Inteligent and Mobile Robotics - 1st project

Development of a reactive agent for a simulated robot that solves labyrinths

Vedran Semenski

Abstract **- This paper presents and describes an implementation of an reactive robotic agent using a library of developed behaviors, all structured in a subsumption architecture. This agent was developed for a project that is part of the "Inteligent and mobile robotics" course and is one of three in total.**

I. INTRODUCTION

The goal of this project is to explore the notion of a reactive robotic agent and the subsumption architecture and understand the benefits and limits of using this kind of setup in a robotic agent. This solution was developed using Java in NetBeans development environment. It was tested using a provided simulator and interface, and will be explained in detail in this document.

II. OBJECTIVES

In this project 3 main objectives were determined:

1. Develop a reactive robotic agent capable of solving labyrinths, using the subsumption architecture. [1]
2. Understand the advantages and limitations of this architecture. [1]
3. Develop a library of behaviours for robotic agents. [1]

III. ROBOTIC AGENT

In this case particular case the robotic agent is 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 robot and the "outside world" are in this project virtual and generated by a simulator.

The solution communicates with the simulator via the ciberIF class. The ciberIF class provides a way to get to initialize the robot in the simulator, fetch sensor readings and give commands to the wheel motors and enable the robot to move. In this solution the functionality available threw ciberIF is divided into movements, sensors and interface actions. The goal of this is to abstract the notion of the outside world and everything that the agents communicates with. This way varius simulators and/or real sensors and motors could be associated and connected to the agent without the need of changing anything in the behaviors, agent, subsumption architecture structure and all the code associated with those parts. In the BasicMovements class there is a list of methods that control the motors and as such provide a more intuitive set of commands that result in movements. BasicSensors class gathers data from the sensors, normalizes some values to more intuitive ones and provides a set of easy to use methods for fetching data. The BasicIF class just has methods for initializing, finishing...

The agent is contained in the BasicAgent class and is created via the AgentFactory. This functionality and abstraction of the agent is added for the possibility of creating new agents with different workflows and/or behaviors. The agent contains basic methods for initialization and starting the agent.

IV. SUBSUMTION ARCHITECTURE

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

This architecture is implemented by organizing behaviors in separate classes and creating instances of the behaviors and than inserting the behaviors in a structure that gives every behavior a priority. The resulting list of behaviors is than ordered by priority from high priority to low priority. Than in the testing phase the algorithm just goes threw the ordered list and executes the behavior that is the first one to satisfy its set of conditions. As the list is ordered it is certain that there is no other behavior with a higher priority that has its conditions satisfied. This way isn't 100% percent consistent with the Subsumption architecture and reactive navigation but it captures the main idea and accomplishes the goal. The reason why it is done this way is because the behaviors do not calculate and memorize any aspects of the measurements and so the testing of the conditions would just be a waste of time and resources. The solution is written in such a way so it is easily reconfigurable and enables the programmer to easily modify the way behaviors are tested and executed.

V. Behaviors

Every behavior is associated with its own class and all behaviors inherited from the base BasicBehavior class. The base class contains abstract methods for tasting behavior conditions and executing behaviors. It also contains methods shared by all behaviors and methods for fetching sensor readings from an instance of the BasicSensors class.

Prioritized list of behaviors:

1. AvoidColision
2. Finish
3. GoToBeacon
4. GoToAproxBeacon
5. FollowWall
6. Wounder

The list is organized in a way which the first behaviors have a higher priority. In the implementation the highest priority behavior is set with a priority of 100 and lower priority behaviors have incrementally smaller priorities of 99, 98, 97 and so on.

AvoidColision behavior activates when the robot comes very close to an obsticle. The actions consists of the robot stopping and rotating itself around its axis in the direction of the sode that is more free of 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.

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

GoToBeacon activates when the beacon is visible by the robot and the and the distance sensors indicate that there is no obstacle between the robot and beacon. The second condition was added in the process of testing the robot to stop the robot from getting stuck between the AvoidColision and GoToBeacon behaviors and to let the lower priority actions be execute and add variety to the resulting robot behavior. The behavior consists of moving straight the beacon.

GoToAproxBeacon is essentially the same as the GoToBeacon behavior but instead of following the beacon from the beacon sensor it follows an approximate location of the beacon location. This approximation is calculated by monitoring the x, y coordinates of the robot, the compas and the direction of the beacon when it is visible. The idea is that when the robot first senses the beacon and gets the direction of the beacon, it saves the values for its location and direction and the direction of the beacon. After the robot comes to a point that is just far enough from the first point that it could calculate the approximate position of the beacon using the two points and directions it saves the approximate position of the beacon and starts calculating the approximate direction of the beacon if the beacon isn't visible. This action is useful if the robot comes to a place surrounded by high walls so the robot wanders "more or less" in the right direction. This behavior is contradictory in alight way to the purely reactive navigation but has proven to be a simple and useful asset to the behavior list.

FollowWall behavior activates when the sensors indicate that there is an obstacle near the robot. The behavior consists of the robot maintaining the distance between itself and the obstacle and moving alongside it.

Wander behavior has no conditions that need to be satisfied for the behavior to activate. This behavior has the lowest priority of all so it always has to be available for execution. The behavior consists of a randomized movement. It is tuned in a way so it moves straight for about 25% of the time and moves in a turn for about 75% of the time. The direction of the turn (left or right) is randomly set on initialization of the behavior and there is a 5% chance that the initialized direction changes.

VI. Testing

Test results indicate that this agent has a very small tendency of colliding with walls and getting stuck in a loop. It found its way to the beacon area on all tested maps almost always. On more difficult maps the robot fails to find the beacon area in time about 10% of the time and on easier maps the margin is even smaller. The agent is set to a fairly low and conservative speed so it can follow walls with a smaller gap and find its way threw smaller openings. For this reason when the robot fails to find the beacon it is usually the result of not finding it in time because it wandered in the wrong direction rather than getting stuck or colliding with obstacles.

VII. Conclusion

The subsumption architecture provides a very flexible platform for expanding the behavior library and further developing and improving the agent. It also enables easy tuning, especially with a large library of behaviors. The reactive aspect of the agent simplifies the implementation but proves to result in a very satisfying behavior 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 behaviors.

By the very nature of this kind of setup the agent has no learning aspect and does not remember any aspect of the environment it is placed in and does not adapt accordingly. As a result this setup is limited in changing and improving the agent in a major way.

Having that in mind it is still a very simple and fast way to get good results and should be used in conjunction with other solutions for an even beter result.

References

[1] "Robótica Móvel e Inteligente, 2014/15, Project 1, Development of a reactive agent for a simulated robot that solves labyrinths ", http://elearning.ua.pt/ , October 2014.