Laura Calzoni 0001058438 Jacopo Merlo Pich 0001038025 Francesca Paradiso 0001037825

Autonomous and Mobile Robotics

Project Presentation

Sanitizer Robot

Task 1 – Environment setup

In a house, a sanitizer robot is required to map and navigate environment, in order to be able to reach the different rooms and perform a sanification of the areas.
The ROS2 TurtleBot3 Burger robot working in Gazebo simulation environment and the TurtleBot3 Big House scenario have been chosen for simulative purposes.

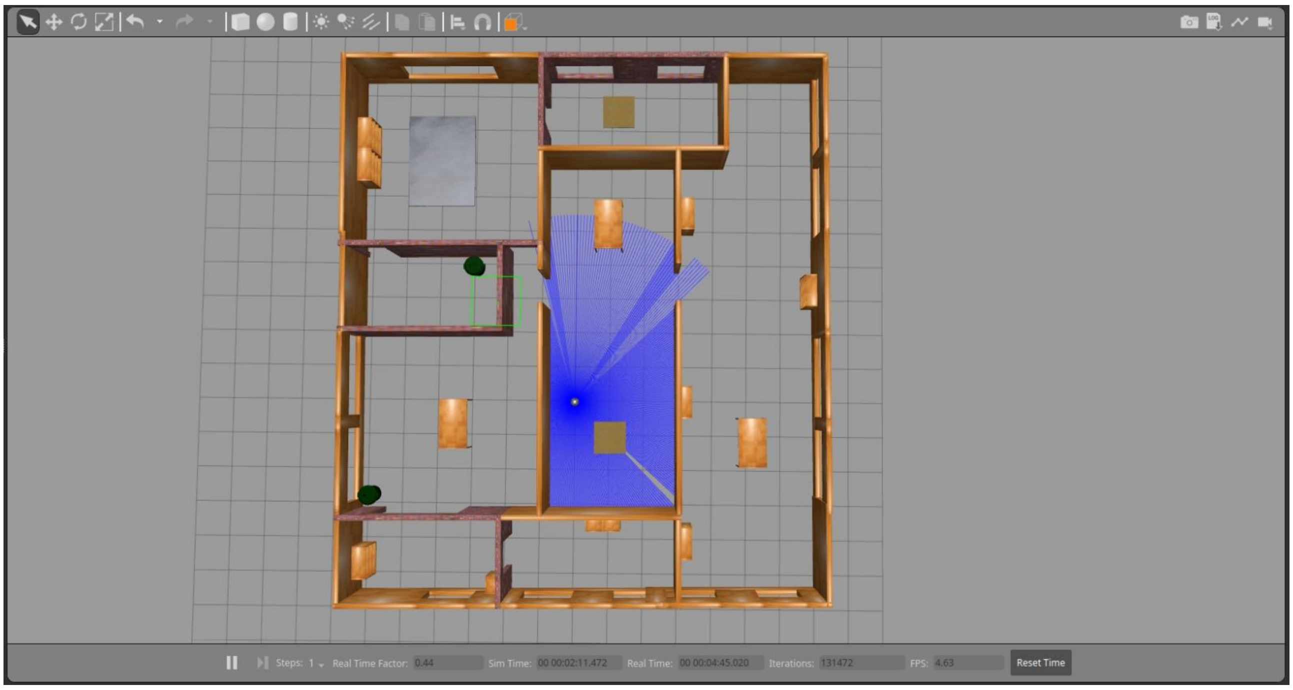

Figure 1 - Big House environment in Gazebo

The robot is initially spawned in a random position in the house and should avoid obstacles that may dynamically change, exploiting the information collected through a Lidar sensor.

To start the simulation, source the folder `AMR_project/task1` and launch the provided file `turtlebot3_big_house.launch.py`:

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_gazebo turtlebot3_big_house.launch.py
```

In this launch file we run the `gzclient.launch.py` and `gzserver.launch.py` files, which can be found in the system directory `opt/ros/galactic/share/gazebo_ros/launch/`, and are responsible for the initialization of the client and server for the simulation in Gazebo. The `gzserver` launch argument in charge of describing the big house world is the `burger.model` file, sourced in the `world` directory of the package provided to us.
Finally, the robot is spawned in the world through the `robot_state_publisher.launch.py` file with argument `use_sim_time` set to true, that allow to use the same clock of the gazebo simulation.

Task 2 – Mapping of the environment

The robot must be able to move autonomously in the environment to create a map as precise as possible. To perform such task, the `explore-lite`, a port of the ROS package `m-explore`, has been used. This package implements a greedy frontier-based exploration, wherein frontiers are regions on the boundary between open space and unexplored areas in the environment. When this node is running, the robot will greedily explore its environment until no frontiers could be found.
`Explore_lite` uses `move_base` for navigation, which provides an implementation of an action that, given a goal in the world, will attempt to reach it (within a user-specified tolerance) linking together a global and local planner. Recovery behaviours can be planned when the robot perceives itself as stuck: by default, the robot performs an in-place rotation to clear out space. Each frontier is given a score based on a combination of different parameters, such as distance from the robot and size, and the frontier with the highest score is classified as the main one, which is the firstly chased.

To succeed in mapping the environment, we carried out different tests and we changed the following parameters:

in `/task2/src/m-explore-ros2/explore/config/params.yaml`:

- ❖ `min_frontier_size` reduced from 0.5 m (default) to 0.03 m. It tunes the minimum size the frontier must have to be set as the exploration goal. Frontiers with smaller size are discarded. Reducing it, the exploration behaviour of the robot is facilitated.
- ❖ `progress_timeout` reduced from 30.0 s (default) to 10.0 s. When the robot, chasing a frontier, does not make any progress for `progress_timeout` seconds, the current goal is abandoned and a new one is selected. The decrease of this parameter is needed in order to not slow down the exploration too much. Indeed, the reduction of `min_frontier_size` causes an increase in the number of frontiers to analyse.

in `/opt/ros/galactic/share/nav2_bringup/params/nav2_params.yaml`:

- ❖ `inflation_radius` augmented from 0.55 m (default) to 0.85 m in the local costmap and from 0.35 (default) to 0.6 in the global costmap. Outside the inflation radius the cost of a cell is not inflated by the cost scaling function. Therefore, by increasing it, we assure a safer exploration.

in `/task2/src/turtlebot3_simulations/turtlebot3_gazebo/models/turtlebot3_burger/model.sdf`:

- ❖ `<ray><range>` increased from 3.5 m (default) to 5.5 m. This parameter defines the maximum distance reachable by the lidar rays, and therefore the maximum distance from which an obstacle can be spotted. A sensor that goes farther allows to be more precise in the detection of obstacles and the subsequent update of the map, leading to a prompt response in the selection of new frontiers. As a consequence, the environment is more likely to be left with no unexplored areas.

To start the mapping, go in the `task2` workspace and then launch the following commands in two separate terminals command:

```
$ ros2 launch task2 task2.launch.py
$ ros2 launch m-explore-ros2 explore.launch.py
```

The latter file launches the `m-explore` node, while the former is in charge of starting the simulation and launching the packages exploited by the `m-explore`:

- ❖ the Gazebo simulation (as in `task1`)
- ❖ the node for Rviz simulation

- the navigation nodes necessary for exploring the environment through `navigation_launch.py`, that is sourced in `/opt/ros/galactic/share/nav2_bringup/launch`/. The navigation parameters passed as launch argument to such file are the ones we find in `nav2_param.yaml` file and have been modified as we addressed above.
- the Asynchronous Slam through `online_async_launch.py`, that is sourced in `/opt/ros/galactic/share/nav2_bringup/launch/`.
  Both this and the `navigation_launch.py` are run with the parameter `use_sim_time` set to True.

As final result of the mapping process, we have obtained a full map of the environment, which is saved at `home/luca/AMR_project/task2/src/m-explore-ros2/` and is portrayed below.

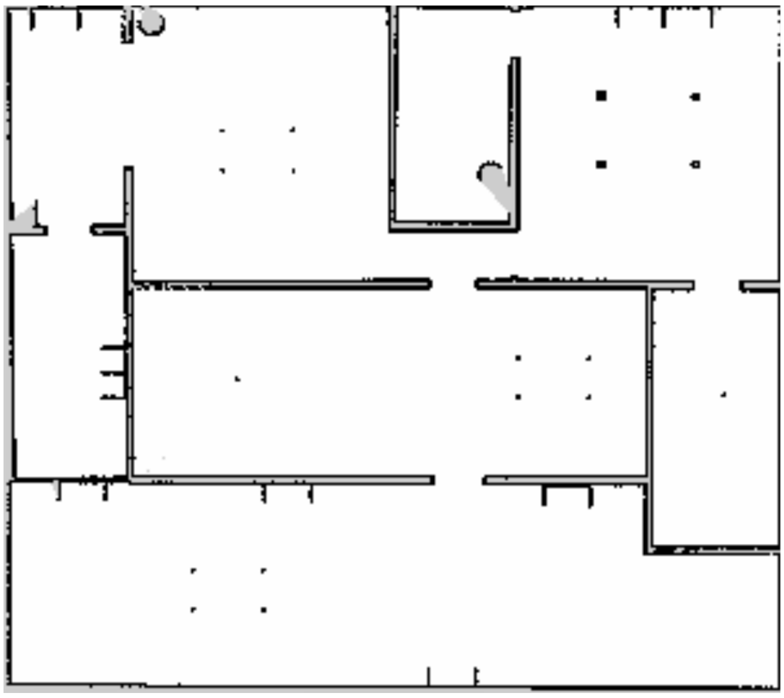

Figure 2 - Map of Big House

Task 3 – Localization and Navigation

Localization

Before being able to navigate the environment and reach a goal point in a room, we need to localize the current position and orientation (i.e. pose) of the robot.
In order to do so, we exploited the `nav2_amcl` node which is part of the `Navigation2` library and emulates the Adaptive Monte Carlo Localization (AMCL). This algorithm exploits a particle filter that spans the state space with a high number of molecules in order to localize the robot. Each of these particles has a position and orientation, representing a guess of where the robot is located, and they propagate following the robot's motion. As new data are provided by sensors, each particle will evaluate its accuracy by checking how likely it would receive such sensor readings at its current pose. Next, the algorithm will redistribute (resample) particles to bias the ones that are more accurate. Iterating these steps, all particles should converge to a single cluster near the true pose of the robot, if the localization is successful.

A key problem with particle filter is maintaining the random distribution of particles throughout the state space when the problem is high dimensional. Due to this reason, it is much better to use an adaptive particle filter which converges faster and is computationally more efficient. The key idea is to bound the error introduced by the sample-based representation of the particle filter.

The AMCL package maintains a probability distribution over the set of all possible robot poses and updates it using data coming from odometry and laser range-finders. The map obtained in the former section is used to compare the observed sensor values.
At the implementation level, the AMCL package of `Navigation2` represents the probability distribution using a particle filter. The filter is “adaptive” because it dynamically adjusts the number of particles in the filter: when the robot’s pose is highly uncertain, the number of particles is increased; while, when the robot’s pose is well determined, the number of particles is decreased. This permits a trade-off between processing speed and localization accuracy.

The `amcl_node` mainly subscribes to the following topics:

- `/scan(sensor_msgs/LaserScan)` which contains data from the lidar sensor, used to detect obstacles and identify the clear path;
- `/tf(tf/tfMessage)`, containing the transformations between the various reference frames of the system, used to calculate the position of the robot with respect to the map of the environment;
- `/initialpose` (`geometry_msgs/PoseWithCovarianceStamped`), the topic used to initialize the robot's position in the map;

while among the topics published there are:

- `/amcl_pose (geometry_msgs/PoseWithCovarianceStamped)`, the estimate of the robot position and its covariance;
- `/tf (tf/tfMessage)`, that publishes the transformation from `odom` to `map`.

Consequently, to perform the localization we created a node called `robot_mover` in which we subscribe to `/amcl_pose` to get the estimated pose and location of the robot from the AMCL node, while we publish on `/initialpose` to communicate with it the initial position whereby we want to spawn the particle cloud.

In order to make the robot avoiding obstacles during the localization process, we subscribe to the `/scan` topic with the variable `scan_msgs` of type `sensor_msgs/LaserScan.` In particular, we exploited the attribute `scan_msgs.ranges` from which we get information about the proximity of possible obstructions within the laser scan width. Even though such message returns a vector of 360 elements (one for each angle covered by the laser scan), we decided to consider just the first 20° degrees on both the right and left side of range:

```
self.dist_from_obst = scan_msg.ranges

right_range = self.dist_from_obst[340:]
left_range = self.dist_from_obst[:20]
```

In this way, if the robot detects an object within 0.8 m in the first 20° on the right it stops and we require it to turn left, while, vice versa, if the obstacle is present within 0.8 m in the first 20° on the left, the robot is demanded to make a right turn.
In order to overcome the problem of getting stuck in an angle, we defined two flags, `flag_right` and `flag_left`, with the aim of keeping track of the movement performed during the previous iteration. In this way we can prevent the robot to turn left if he has done a right turn just before (and vice versa). In this case, as a consequence, to avoid obstacles the robot maintains the angular speed it previously had, until it doesn't see any more obstructions in the given range. At that point, it restarts moving straight, resetting the flags.

To set the robot velocity, we publish on the /`cmd_vel` topic of type `geometry_msgs.msg/Twist` with the variable `msg`. In particular, the robot is controlled performing two simple motions:

- Straight motion:
```
msg.linear.x = RobotMover.LINEAR_SPEED
msg.linear.y = 0.0
msg.linear.z = 0.0
msg.angular.x = 0.0
msg.angular.y = 0.0
msg.angular.z = 0.0
```

- Left/right rotation:
```
msg.linear.x = 0.0
msg.linear.y = 0.0
msg.linear.z = 0.0
msg.angular.x = 0.0
msg.angular.y = 0.0
msg.angular.z = (-)RobotMover.ANGULAR_SPEED
```

where `RobotMover.LINEAR_SPEED` and `RobotMover.ANGULAR_SPEED` respectively set to 0.4 and to 0.3.

The whole localization process is generated by a timer with period `timer_period = 0.1`. Then, we implement all the actions described before in the callback function of the timer, which is executed each 0.1 seconds. At each iteration we check the AMCL particles covariance in order to be able to stop the localization as soon as it reaches a small value (`max_cov = 0.1`).

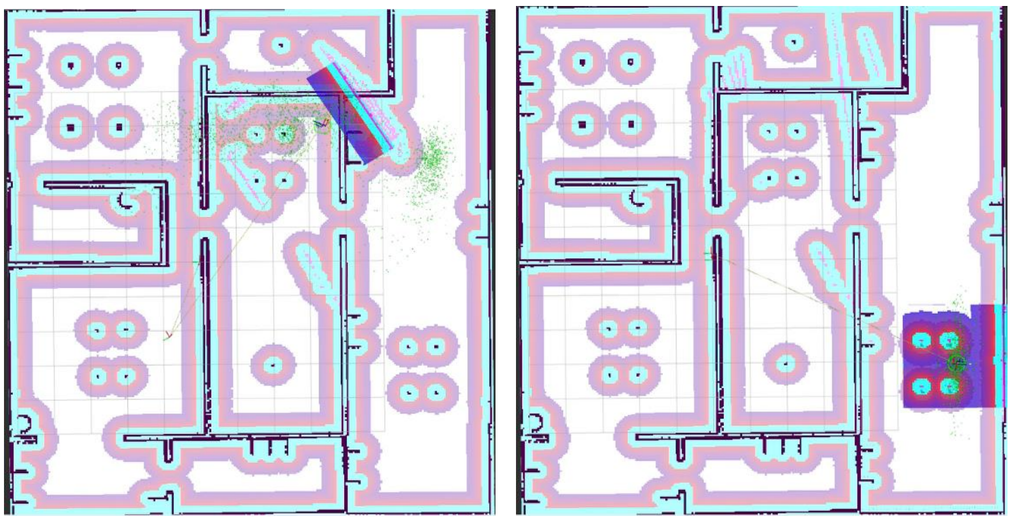
Figure 3 - Localization process

Navigation

Once the localization is completed, we are ready to navigate the environment and reach a specific location in the map. We created three text files (found in `/AMR_project/task3/src/loc_and_nav/loc_and_nav/`) with different goal positions, in order to test the performances in different configurations.

	goal1.txt	**goal2.txt**	**goal3.txt**
position.x	5.0	-5.0	-6.0
position.y	4.0	-4.0	-1.0
orientation.z	0.0	0.0	0.0
orientation.w	1.0	1.0	1.0

So, with that purpose, we created a node called `navigate_to_pose_client` where we generate an action client to the `NavigateToPose` action of the `nav2_msgs.action`. The goal pose, which is a message of type `NavigateToPose.Goal()`, is then read from the selected text file and sent to the action client through the `send_goal_async()` function.
Once the navigation is finished, the script displays a message of goal succeeded or failed depending on which `GoalStatus` message has been returned (for the navigation to be succeed of course the resulting status must be `GoalStatus.STATUS_SUCCEEDED`).

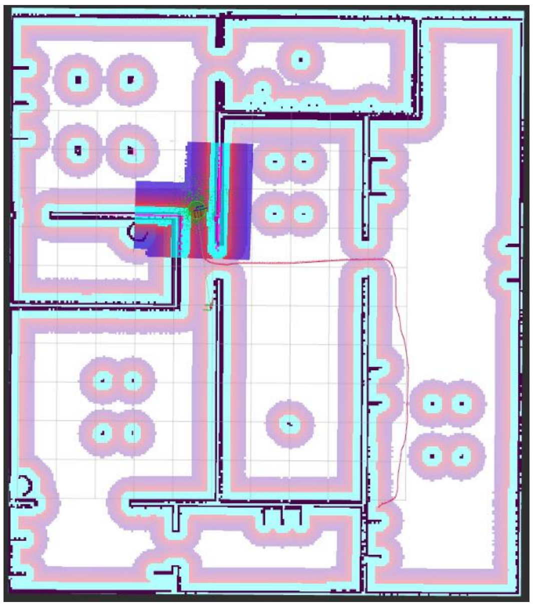
Figure 4 - Navigation process

Both the localization and navigation assignment are implemented in a self-made python script that can be run by sourcing the `task3` workspace and launching:

```
$ ros2 run loc_and_nav loc_and_nav
```

Moreover, in order to start the simulation and all the packages necessary for the third task, we created a single launch file, also located in the `task3` workspace. It must be launched simultaneously to the `loc_and_nav` script:

```
$ ros2 launch loc_and_nav task3.launch.py
```

Such file contains the code for running:

- Gazebo simulation (as in `task1` and `task2`)
- Node for the Rviz simulation
- `Navigation2` bringup, sourced at `/opt/ros/galactic/share/nav2_bringup/launch/`, that launches the scripts necessary for the localization (`localization_launch.py`) and for the navigation (`navigation_launch.py`), as in Figure 3.
  The `bringup_launch.py` file is launched with parameter file `nav2_params.yaml` sourced in the directory `/opt/ros/galactic/share/nav2_bringup/params/`. Here the setting parameters of all the nodes in the graph above can be found, some of which have been already presented in the explanation of task 2.

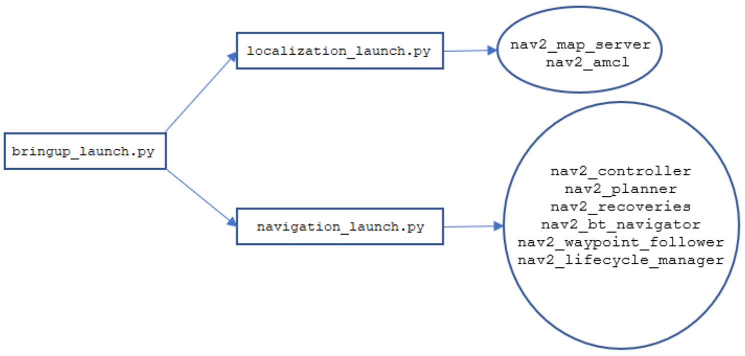

Figure 5

Then, to obtain a better localization of the Big House environment, we have changed the `max_particles` and `min_particles` AMCL parameters, which have both been increased respectively from 2000 to 5000 and from 500 to 2000.

Task 4 - Reinforcement Learning

In the fourth task we had to design a path in order to entirely sanify a room, following the energy propagation rule $E(x, y, t)$ emitted by the UV lamp:

$$E(x, y, t) = \int_0^t \frac{P_I}{(x - p_x(\tau))^2 + \left(y - p_y(\tau)\right)^2} d\tau$$

where $p_x(t)$ and $p_y(t)$ represent the robot position along the x and the y axis respectively at time t.

In order to achieve this goal, we have adopted a Reinforcement Learning approach trying to establish the most efficient path to follow to sanify the room. In particular, we have relied on SB3 (Stable Baselines3), which is a set of consistent implementations of reinforcement learning algorithms in PyTorch. Especially, we have used a Deep Q Network (DQN) model with an MLP (Multi-Layer Perceptron) policy. For the observation and action space, we have used Gym.

Specifically, DQN is a deep learning algorithm that uses a neural network to approximate the Q-function in Q-learning. This allows the agent to learn a more complex and accurate representation of the State-Action Value function, which can improve its performance in complex environments with large state and action spaces.

The MLP policy takes the current state of the agent as input and produces a probability distribution over possible actions as output. The neural network is trained using a form of supervised learning called policy gradient descent, where the goal is to maximize the expected reward received by the agent over a series of actions.

Environment setup

In order to correctly train the agent, we have designed a specific observation of the environment to simulate room as much generic as possible. It is represented by a grid of a width and height which are randomly taken between 15 and 20 cells (corresponding to 3 and 4 meters). The agent position and the obstacles (between 3 and 5) are arbitrarily spawned on the grid.

The actions that the agent can do are 5:

- Stay in position
- Move up
- Mode down
- Move left
- Move right

To each action is associated a specific reward, depending on the scenario in which that action leads us:

- -1 for each instant of time that passes (until the task is completed)
- +1 for each cell sanified in that moment
- -10 for every collision, either with obstacles or walls

The design of the observation is fundamental to achieve good performance. So, we have studied multiple typologies of observations with different sizes, shapes and logics. The DQN turned out to perform better with vector of single elements and, after several tests, we finally obtained good results using a local observation of the agent's neighborhood and assigning to each observed cell:

- 1 if the cell is still not sanitized,
- -1 if the cell has reached the minimum value of UV energy
- -2 if the cell is occupied by an obstacle or is out of the map

The value of the grid follows a clockwise logic with respect to the agent position.

Training

We created a DQN model with an “MlpPolicy” customized with the following parameters:

`buffer_size`	`1000000`	**`gradient_steps`**	`-1`
`learning_starts`	`100000`	**`exploration_fraction`**	`0.3`
`batch_size`	`128`	**`exploration_rate`**	`0.05`
`gamma`	`0.1`	**`learning_rate`**	`1e-05`
`tau`	`1.0`		

We trained the model for 1.5 million time steps using the library function `.learn()` available on the `stable_baseline3` library, while the performances are tracked using `TensorBoard`.

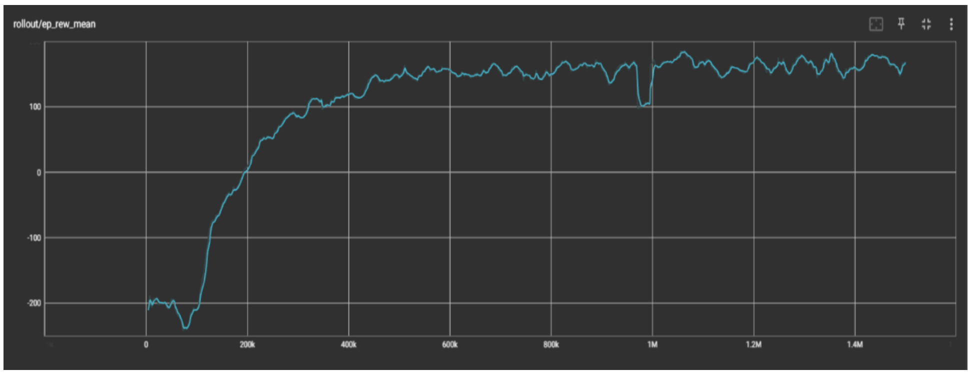

Figure 6 - `TensorBoard` graph of the mean reward computed over 4 episodes

The best result achieved is an agent able to sanitize at least 95% of the room 9 times over 10 on average.

Problematics and Analysis

However, problems arise due to the local nature of the observations. Particularly, a loop phenomenon occurs when the agent is surrounded only by sanitized cells, as in the observation only -2 and -1 elements are present. In this scenario the agent gets stuck in a loop policy, even penalizing through high negative rewards the cells of the path previously followed. To solve this issue, we attempted to both expand and change the shape of the neighborhood. In particular, we changed it into a rhombus or a star shape to better spot unexplored areas in the corners of the map, that a square window failed to do.

We have benchmarked the performance to find the most efficient way to achieve the result, with different shapes and different radius sizes. Despite the star shape seemed more effective in finding not sanitized areas, the rhombus shape has proved to assure more stability and to achieve better performance in the path generation.

The best results have been obtained with a radius of almost 20. In fact, higher values resulted in a deterioration of the performances, while smaller radiuses increased the loop phenomena.

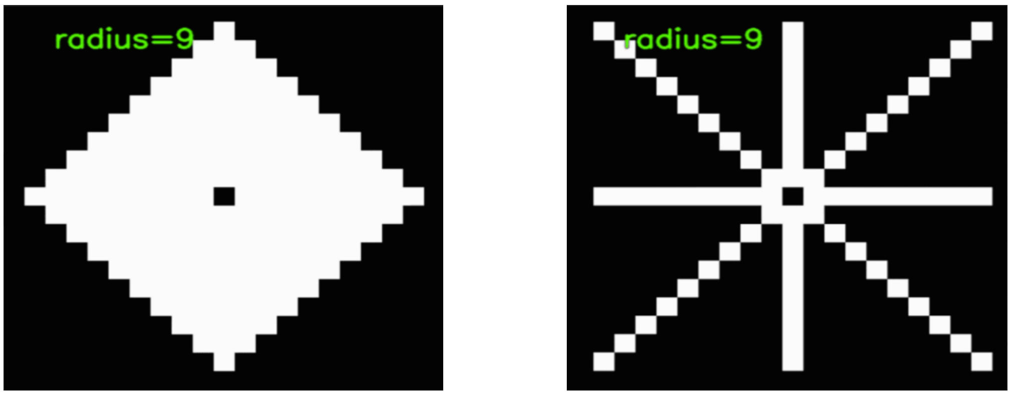

Figure 7 - Changing the shape of the neighbourhood window

Moreover, we tried multiple designs for the reward, as implementing a “punitive path” to penalize the agent in case it returns on cells already covered, but no significant results were achieved.

Finally, we performed some tests using a CNN (Convolutional Neural Network) and a custom CNN in order to use a full observation of the map. However, also this strategy didn’t work, as the mean reward does not improve significantly.