Artificial Intelligence For Robotics 2
Assignment 2 Report

Zoe Betta 5063114
Francesco Ganci 4143910
Federico Zecchi 4949035
Litong Huang 5058374

05/07/2021

1 Abstract

In the case of a robot moving from a designated start region and reaching four other given regions until the goal region in the environment, along with four landmarks to detect the robot position, a task and motion planning problem is created. The PDDL domain and problem files are modified accordingly in order to obtain the solution. An optimal path should be found for the robot to move from one region to another in a discrete sequence until all regions are visited once after taking the travelled distance of the robot and the cost due to uncertainty into account. Semantic attachments are implemented through the functions 'triggered', 'act-cost', and 'dummy'. Because of that, the task planner can communicate the start and goal region to the external model, and the results can then be stored in 'dummy' variable and assigned to 'act-cost'. By using the popf-tif planner, after the given files related to the way-points and landmarks are parsed, the search of a plan will start taking place. As the external module is called, the semantic attachments are turned on. With the fluent 'triggered' having a value larger than zero, the path from one region to another region is searched, and the cost of that can be estimated by using the Extended Kalman Filter (EKF). Assuming the initial covariance matrix (P) to be $diag(0.02m^2, 0.02m^2, 0.02rad^2)$, once knowing the goal position and detecting the landmark, the uncertainty on the position is reduced, and the cost is calculated with the updated covariance matrix (P). For the PDDL Domain file, two durative actions are included, one is "$goto_region$", which it communicates the start and goal region to the external module; the other one is "$localize$", which the Euclidean distance travelled between two regions is calculated. In addition, the cost due to uncertainty is also computed and then added to the distance cost. With all the aspects above mentioned included, after running the PDDL Domain and Problem files, the optimal path with minimized act-cost for the robot to travel all four regions from the start region is computed. We have all of our code available in a github repository at: `https://github.com/programmatoroSeduto/AIRo2-Assignment-2.git`

2 Introduction

In this assignment, modeling integrated task and motion planning problems are being investigated. A robot is placed at a certain region in the environment, and it must navigate to four different regions in order, as shown in Figure 1. The starting region of the robot is assumed to be the initial position at (0,0). Then, it has to move from one way-point to another, until visiting all four way-points, which are (2,0,0), (0,2,1.57), (-2,0,3.14), and (0,-2,-1.57). In addition, 4 landmarks are included to localize the position of the robot, which are (4,0,3.14), (0,4,-1.57), (-4,0,0), and (0,-4,1.57), respectively. For this assignment, the popf-tif planner, which is a temporal PDDL planner based on a forward heuristic search engine and exploits partial ordering, is used to approximate its effects and to evaluate alternative states in the search space in order to find the minimum time-step. To do so, the Euclidean distance travelled by the robot between two regions is calculated. Other than that, as the robot arrives at each region, by using an odometry model, Extended Kalman Filter (EKF) can be applied to estimate the final robot state with noises being measured. The act-cost, which is the cost due to uncertainty, can be computed as the resulting covariance matrix (P) while assuming the initial covariance matrix (P) to be $diag(0.02m^2, 0.02m^2, 0.02rad^2)$. Eventually, the discrete sequence of region visits performed by the robot, as the optimized solution, is required to be solved while taking the distance travelled and the cost into consideration.

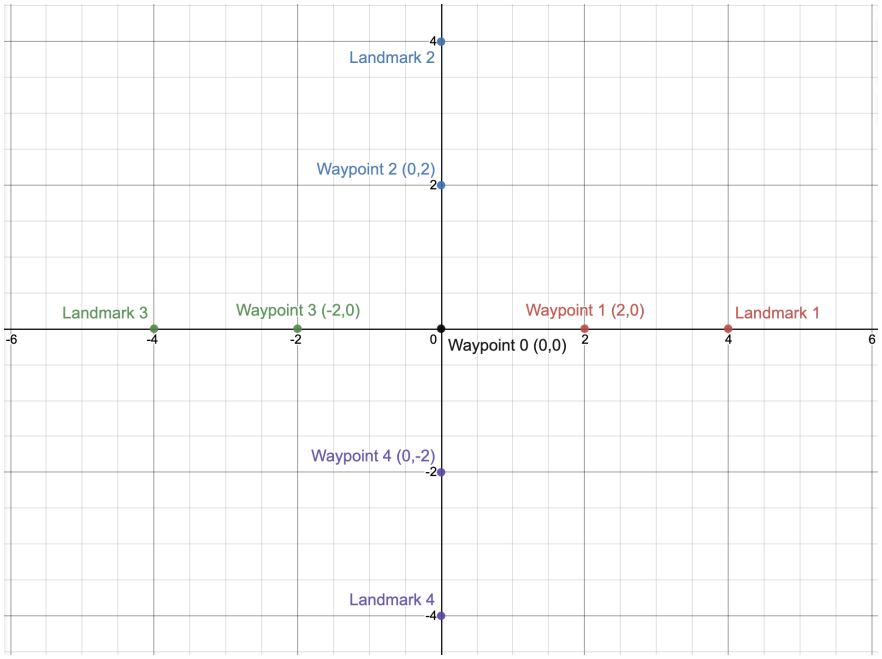


Figure 1: The scheme layout of a robot navigating to 4 different regions

3 Methods

For this project we implemented three files, one in programming language CPP and the other two in PDDL.

3.1 Semantic attachment - PDDL side

The module 'visits', provided along with the instructions for the assignment in a to-modify version, implements the semantic attachment through these PDDL fluents:

- **triggered** : command for executing the motion from region A to region B. The motion is considered from one way-point to another way-point.
- **dummy** : after localization, the cost is computed and assigned to this fluent.

The PDDL code turns on the semantic attachment when the fluent '(triggered ?from_region ?to_region)' is set to a value greater than zero. Otherwise, the semantic attachment is turned off.

3.2 VisitSolver.cpp

Here is how the integration with the module works. Before starting the search of a plan, the planner 'popf_tif' initializes the module using the method 'loadSolver( )' from the class 'visitSolver': here the other files are parsed ('waypoints.txt', 'region_poses' and 'landmarks.txt'), and the class is initialized; here also the fluents involved in the semantic attachment are declared to the planner. Then, the searching can start. The external module is called, after init, the actually considered action is evaluated by the planner, only if this action contains the name declared in the initialization of the module. The method 'callExternalSolver( )' is called when the semantic attachment is turned on: the planner passes only the grounded fluents (map 'initialState') declared before during the initialization step. When called, the method 'callExternalSolver( )' simply analyses the map 'initialState', each element at turn. It searches for a fluent 'triggered' with value greater than zero. If it succeeds, from-region and to-region are extracted from the string of the fluent, and are used for finding the two waypoints referred to. Then, the function executes the Kalman Filter and returns the cost associated to the motion in the fluent 'dummy'.

Here are the methods implemented in this class:

- **VisitSolver::get_waypoint_coordinates** : get the coordinates of a waypoint referring to a region by name
- **VisitSolver::distance_between_regions** : Euclidean distance between two waypoints
- **VisitSolver::closest_landmark** : get the closest landmark to a certain position
- **VisitSolver::KF_localize** : an implementation of the kalman filter, using math library *Eigen*. See the next paragraph.

3.3 Extended Kalman Filter

In order to calculate the cost due to uncertainties that needs to be taken into account for calculating the total cost of the project, we implemented an Extended Kalman Filter inside our CPP file. To do so, we took inspiration from the repository github at the following link `https://github.com/shazraz/Extended-Kalman-Filter`. When the 'localize' durative-action is executed, the CPP file is called, in particular the function calls ExternalSolver that computes the cost due to the distance and the uncertainty.

In order to create the Extended Kalman Filter (EKF), we started with two assumptions: the robot is able to localize only when it is in the region, and the robot calculates the odometry ten times when it goes from the one position to the next one. The first thing we do is to update the covariance matrix (P) assuming a landmark is detected. Our detection of the landmark is disturbed by some white noises with the mean value zero and the standard deviation of 0.5 for the distance and of 0.2 for the orientation. After that, we computed ten cycles to update the P matrix without detecting any landmark. In the end, we reached the goal position and detected again a landmark trying to reduce the uncertainty. We then computed the cost by calculating the trace of the covariance matrix.

For the implementation of the EKF, we had to define some matrices. We started with defining the system model as:

$$\begin{cases} X_{k+1} = f(X_k, U_k) \\ Y_k = g(X_k) \end{cases} \tag{1}$$

Where X is the matrix containing the state of the robot, which in our problem is the x and y coordinates and orientation; U is the vector of inputs, for example, the linear and angular velocities; Y is the measurements, which in our case the distance and orientation from the landmark. Our system has the following system of equations:

$$\begin{cases} x_{k+1} = x_k + \Delta D_k * cos(\theta_k) \\ y_{k+1} = y_k + \Delta D_k * sin(\theta_k) \\ \theta_{k+1} = \theta_k + \Delta \Theta_k \end{cases} \tag{2}$$

In this system, ΔD_k and $\Delta \Theta_k$ are the inputs of the system, the data read from the sensors is used to calculate the odometry in a real implementation of this problem.

$$\begin{cases} d^2 = (x_k - x_{landmark})^2 + (y_k - y_{landmark})^2 \\ \alpha = \theta_k - \theta_{landmark} \end{cases} \tag{3}$$

This system on the other hand describes the data read when a landmark is detected. We assumed that we can get the distance from the landmark and also the difference from our orientation with respect to the absolute reference frame and the orientation of the landmark. Matrix F is the state transition model and it is defined as follows:

$$F = \frac{\partial f}{\partial X} \tag{4}$$

For this reason, from equation 2, we obtained the matrix F implemented in the code:

$$F = \begin{bmatrix} 1 & 0 & -sin(\theta_k) \\ 0 & 1 & cos(\theta_k) \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

Matrix P is the covariance matrix, which is the matrix representing the uncertainty on the position. It is initialized every time a new estimation is required as a 3x3 diagonal matrix that has on the diagonal 0.02 based on the specifications. Matrix P is then updated in the beginning because of the detection of a landmark in the initial region, then for ten times due to a simplified odometry. In the end when the robot is in the final region and it detects a new landmark. This matrix is always updated depending on the odometry uncertainty:

$$P = FPF^T + Q \tag{6}$$

Only when new data on the landmarks are available, it is updated using the following equation:

$$P = (I - KC)P \tag{7}$$

Where I is the identity matrix of the correct dimensions. Matrix C is the measurement matrix, and it expresses how the output varies when the robot moves around the posture X at which the function is calculated. It shows how sensitive the measurements are with respect to the variations of the state. Matrix C is defined as follows:

$$C = \frac{\partial g}{\partial X}(X_{k+1/k}) \tag{8}$$

In the implementation for our model, it is a 3x2 matrix described as:

$$C = \begin{bmatrix} 2(x - x_{landmark}) & 2(y - y_{landmark}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{9}$$

Matrix R encodes the uncertainty with respect of the data acquired while detecting the landmarks. Since from specifications, it was asked to implement a zero mean noise on the measurement. We decided to implement a Gaussian noise with the mean equal to zero and the standard deviation of 0.5 for the x and y coordinates and 0.2 for the orientation angle. In order to so, we used the code presented at the following link `https://stackoverflow.com/questions/32889309/adding-gaussian-noise`. We implemented the iteration one hundred times in order to have a noisier measure since we decided the standard deviation value to be low. After we decided the standard deviation, matrix R, which is a 3x3 matrix, has on its diagonal the standard deviations squared:

$$R = \begin{bmatrix} 0.5^2 & 0) & 0 \\ 0 & 0.5^2 & 0 \\ 0 & 0 & 0.2^2 \end{bmatrix} \tag{10}$$

Matrix Q is an additive noise to the evolution model; for simplicity, it was decided to be zero.

Matrix S is the covariance of $(Y - \hat{Y})$, which is the measurement and the expected measurement, we defined it in order to simplify further calculations of K.

$$S = CPC^T + R \tag{11}$$

K is the Kalman Gain, and it is defined as a matrix of 3x2. It is calculated as the result of the following expressions. Ideally, the Kalman gain should increase as the uncertainty of the position decreases.

$$K = PC^T S^{-1} \tag{12}$$

The difference from the three phases of our EKF is how the covariance matrix (P) is updated. When we detected a landark (first and third phase), the Kalman Gain (K) is updated and so P is updated with the formula 7. To mimic the odometry part, we used the F matrix by calculating the direction the robot needs to go from the start to the goal region by using the atan2 function that is defined for all possible values of x and y, other than (0,0) and is also able to return an angle of pi/2. The matrix F remains constant for the entire simulated path since it only depends on the orientation, and that is constant since the robot should ideally always move in a straight line. After calculating F, we were able to update P using the formula 6.

3.4 PDDL Domain and problem

For the PDDL part of the assignment, we implemented a domain file and a problem file. In the domain file, there are two durative-actions: 'goto_region' and 'localize'. The action 'goto_region' implements the movement part of our problem while the 'localize' action calculates the cost to reach the position.

In order for the 'goto_region' to be called at the beginning of the action, the robot must be in the starting position. In addition, a predicate that states the robot has not arrived yet is needed to avoid calling the two actions multiple times. This action moves the robot from the start region to the goal region by setting the predicate (robot_in ?v ?from) to false at the beginning and setting the predicate (robot_in ?v ?to) to true at the end. It also saves the path in the predicate (plan ?from ?to) in order to communicate it to the 'localize' action that calculates the cost. It sets the goal region as visited, and it also sets (arrived) to true and (not_arrived) to false .

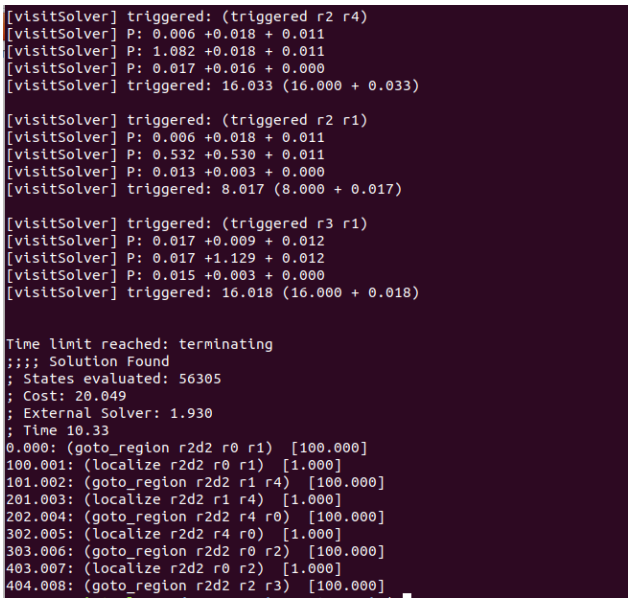
The durative action 'localize' on the other hand can be called when the predicate (arrived) is true and when the predicate (plan ?from ?to) is active. By doing so, the 'localize' will be called always after the 'goto_region', and they will have the same initial and goal region. This action sets the predicate (triggered ?from ?to) to 1, so it turns on the semantic attachement that executes the cpp code and returns the cost for that specific motion in 'dummy'. After that, the predicate (plan ?from ?to) is set to false, and the semantic attachment is turned off. The predicates (arrived) and (not_arrived) are switched, from true to false and vice versa accordingly to avoid having more than once the same 'localize' action. The 'act-cost' function is increased based on the cost calculated and returned in 'dummy'.

The problem file is quite simple. The robot starts from the region 0 and has a goal to visits all the other regions (r1, r2, r3, r4) while minimizing the 'act-cost' function. The predicate (not_arrived) is set to true in the beginning in order to have first the 'goto_region', then the 'localize' action. By setting to true the (arrived) predicate, we would have first the 'localize', then the 'goto_region'. For this problem, we worked with the closed world assumption: every predicate that is not explicitly set to true is false.

4 Results

In order for the planner to search the best solution that has the act-cost minimized, we had to introduce two flags in the script in order to tell the planner to avoid stopping at the first solution found. We needed to work in anytime mode in order to search for a better solution for a predetermined amount of time, it was then added -n -10t where 10 is the time, in seconds. We can notice that the planner never finds the best solution, but it keeps searching for a better one. This happens because we introduced randomness in our assignment by adding noise to the measurement, so each calculation is different from the previous one. Because of this, it will have as a consequence that has a non convergence to the best solution, but only the best solution found in the given time.

Here the results for the planner without modifying the waypoint.txt and landmark.txt files.



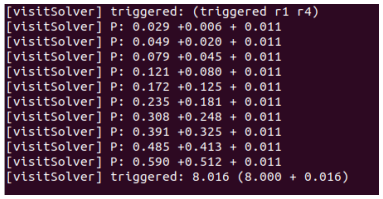
(Fig. 2)

Here, we printed the intermediate results for the diagonal of matrix P, the result after taking into account the first measurement, the result after odometry, and the result after reaching the goal region and taking into account the second landmark. We can notice that the uncertainty decreases after the first detection (it is always less than the initial value of 0.02; it

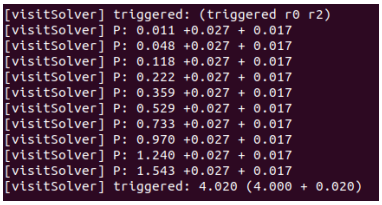
increases during odometry, and it decreases again after the second measurement.

We can notice that the robot does not have the shortest path available in this case, which it would visit the regions diagonally after reaching the first one (for example, r0 r1 r2 r3 r4). However, after reaching r4, it goes back to r0, which is an already visited location; and then reaching r2, which it goes straight up, see Figure 1. More importantly, the robot takes then into consideration the cost due to uncertainties. Given the path for what reason the robot did not move directly from r4 to r2, this might be because having an intermediate localization step will reduce uncertainties.

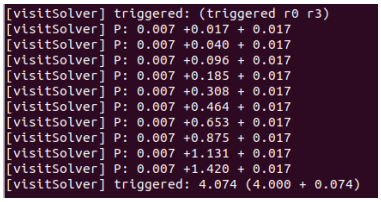
We can observe better what happens during the simulated odometry in Figure 3, 4 and 5.



(Fig. 3)



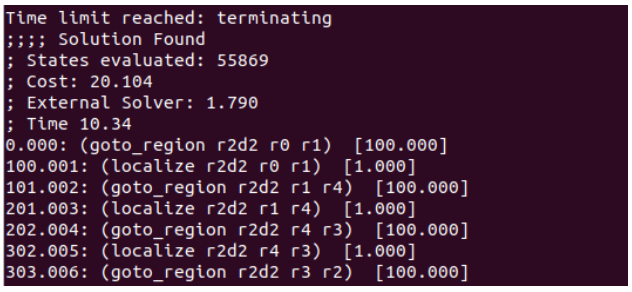
(Fig. 4)



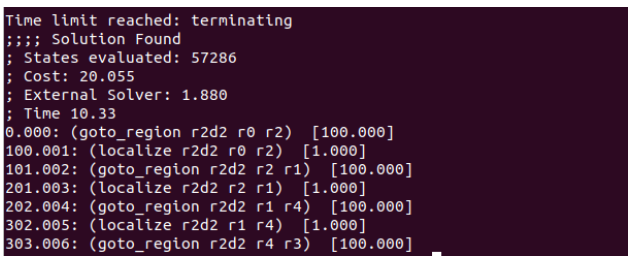
(Fig. 5)

In Figure 3, we have the calculations of odometry for the robot moving from r1 to r4, which it moves diagonally. In this case, the uncertainty increases evenly for the x and y coordinates while remaining constant for the orientation. In Figure 4, the robot is moving from r0 to r2, so it moves vertically. We noticed that the uncertainty increases only for the x coordinate. Therefore, we can deduce that while the robot moves vertically since the uncertainty is only on the horizontal position and not the vertical one. Symmetrically for Figure 5, the robot moves from r0 to r3, which it moves horizontally, and the uncertainty increases only along the y coordinate.

We then modified the location of the way-points in order to verify the correctness of our plan. In particular, we moved r2 to the coordinates (0;20), placing it furthest away from all of the others.



(Fig. 6)



(Fig. 7)

We can notice that since r2 is so far away from the other regions, it is the last one visited, so the robot can stop in that region without having to go back or basically duplicating the distance. We also modified only the landmark.txt file by moving one landmark so far away that it was basically deleted. We moved l1 to the coordinates (40; 0). We can notice that this does not cause any problems since the robot in r1, the region closest to the deleted landmark is still able to localize using one of the other landmarks. For this reason, the uncertainty remains contained.