Navigation Method Selector for an Autonomous Explorer Rover with a Markov Decision Process

Simon Le Gloannec, Abdel-Illah Mouaddib

Navigation Method Selector for an Autonomous Explorer Rover with a Markov Decision Process

Simon Le Gloannec and Abdel-Illah Mouaddib

GREYC, UMR 6072
Université de Caen Basse Normandie
14000 Caen, France

Abstract. Many high level navigation methods exists for autonomous rovers. Usually, fixed methods is choosen for the rover to follow its trajectory. In some cases, a rover can have several embedded navigation methods, but there is currently no automatic approach to select the best method. In this paper, we propose an automatic navigation method selector using Markov Decision Process (MDP) framework. Navigation method are modelized in the MDP transitions functions. Results we achieved were promissing.

1 Introduction

In the robotic application fields, robots often have several actuators and sensors in order to interact with the environment (cameras, telemeters, GPS, wheels, arms...). Some high-level methods are then implemented in the robot's kernel to produce complex behaviours. For example, some robots recognize objects (methods) with a camera (sensor) in order to follow or grab them (behaviour). Many methods can also produce the same behaviour with different sensors.

We consider here the problem[1] of a robot that have to follow a given trajectory. The trajectory is composed of points that the rover must approach as best as possible. There exist many methods to go from one point A to another point B. A robot can move forward by using only its odometer, or by following a wall detected with a telemeter. These methods produce more or less good results but they are more or less difficult to get start. Following a wall by using the telemeter has a cost because at the same time, the robot can not use it to perform an other task. The main problem in this context is to **select the appropriate method** to perform the desired behaviour, i.e. follow the trajectory. In our case the methods are : move towards an object, move along a wall and move without help except the robot odometer. Some objects in the environment can then help the robots : walls, trees, houses, other vehicules. Methods are implemented to use these objects in order to follow this path. We do not present these methods in details but we are concerned with the module that permits the robot to select the best method at the right time.

[1] This work is supported by DGA (Délégation Générale pour l'Armement) and THALES company.

Markov Decision Processes (MDP) [Bel57, Put94] are used to modelize this selection problem [LMC08],[MZ00]. Even if most of the time the policy computation is "*only*" quadratic in the problem state space, the state space is generally huge. Moreover, if a new useful object appears in the environment, the policy must be recomputed online. We tackle this two problems in this paper with an acyclic directed graph in the MDP.

The first part of this article shows the autonomous rover context. Then we modelize the problem into an MDP. We describe the algorithm that produces the policy. In the same section, we show how to recompute a part of this policy when a new object appears in the environment. Finally we present some experimental results

2 The Target Application

This work is developped in a project in collaboration with THALES company under grant of DGA. The project consists of an autonomous rover has to follow a given trajectory. This path is defined as a set of points (see Fig. 1). The objective for the rover is to pass throw these points or as near as possible. It has three high-level methods to move to the next point. The first method is to use an odometer that is quite imprecise. The second method is to recognize a particular object in front of the rover, it will then move toward this object during some time [CBC+05], [SRC06], [BM07]. The third method is to detect a wall on the side. The rover has to move along the wall.

The three methods end up with in the same behaviour which is to move toward a particular point. The first method is less precise than the others but the environment does not always provide good objects to use. Between some points, the method to apply is fixed. For example, during the first part of the trajectory depicted in Fig. 1, the rover must move along the wall. In some region, no methods are selected (third part in Fig. 1). We propose in this case an automatic navigation method selector, We formalize this problem with an MDP. In addition, the rectangles offers a simple strucutre to perform easily computation since it allows us to transform the space as a grid. If the rover is far from it's path, we consider it is in a failed state. We define a rectangle around each pair of points. Beyond these rectangles, the rover is in a failed state.

We also suppose that the rover never turn back. This assumption is important because it permits us to have an acyclic structure problem. We will show in the next section how to use this acyclic property to solve the problem more efficiently.

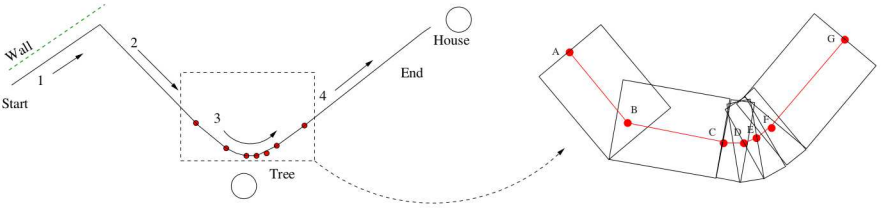

Fig. 1. The rover must follow a trajectory and select navigation methods

3 Formal Framework

The problem is formalized in a Markov Decision Process MDP model. We calculate a policy with this model that indicates the action to select given a particular state.

3.1 Background : MDP Model

An MDP is a tuple S, A, T, C where

- S is a finite set of states,
- A is a finite set of actions,
- T is a transition function. It gives the probability of being in the next state given the selected action and the current state,
- C is a cost function. This cost includes many features like distance and changing action.

Actually a state is the rover position combined with its orientation. $S = \mathcal{R}^2 \times]-\pi, \pi]$. But not all states are accessible according to the previous assumption. We transform the state space into an acyclic graph that is much easier to solve with an MDP. As shown in Fig. 1 (right side), we transform every single segment of the path into a rectangular shape area. States that are outside these boxes are failed states. All the failed states are considered as a unique failed state.

At this point the transformation is not finished. We combine three geometrical tranformations (rotation, scale, translation) in order to obtain a simple grid for each rectangular shape. The grid ratio Δ transforms a $l \times L$ rectangle into a $M \times N$ matrix where $M = L/\Delta$ and $N = l/\Delta$. An α rotation is needed to straighten up the grid. Finally, the left lower corner is moved to the origin (translation). We do this transformation for each rectangle.

The graph is not yet acyclic. To reach this, we only keep angles that are not moving backward. We select N_ω angles (for example five in Fig. 2) appropriate to our target application. The state space is now composed by n $M \times N \times N_\omega$ matrix. The state space is acyclic : while moving forward, the rover cannot not enter the same state twice.

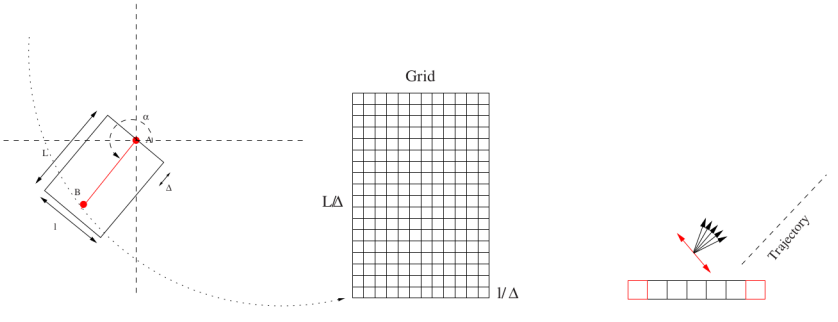

Fig. 2. Transformation from a continuous state space into a 3D discrete $M \times N \times N_\omega$ matrix

There are 3 kinds of actions

1. a_m consists of moving along the path without help (except an odometer),
2. a_{o_i} consists of moving towards the object i,
3. a_{w_j} consists of moving along the wall j.

In each rectangle/grid, the rover can only follow one of two possible objects or one of two walls. Thus, the rover can perform 5 actions in a given grid $a_m, a_{o_1}, a_{o_2}, a_{w_1}, a_{w_2}$, where o_1, o_2, w_1, w_2 are respectively object 1, object 2, wall 1 and wall 2. The transition function represent the uncertainty model the uncertainty reliable to the rover movement method. When the rover helps itself with an object or a wall we consider the transition as perfect (i.e. deterministic)

$$T(\langle x, y, \omega\rangle, a_o, \langle x + v.\cos\omega, y + v.\sin\omega, \omega\rangle) = 1. \tag{1}$$

where v represent the rover speed.

Without helping, the Transition is uncertain. The rover can deviate from $\delta\omega$

$$T(\langle x, y, \omega\rangle, a_m, \langle x + v.cos(\omega), y + v.sin(\omega), \omega\rangle) = 1 - 2p \tag{2}$$
$$T(\langle x, y, \omega\rangle, a_m, \langle x + v.cos(\omega + \delta\omega), y + v.sin(\omega + \delta\omega), \omega + \delta\omega\rangle) = p \tag{3}$$
$$T(\langle x, y, \omega\rangle, a_m, \langle x + v.cos(\omega - \delta\omega), y + v.sin(\omega - \delta\omega), \omega - \delta\omega\rangle) = p \tag{4}$$

where v represents the rover velocity.

The cost function C is a sum of four atomic cost functions:

1. the distance cost $C_d(\langle x, y, \omega\rangle)$ measures the distance between the current state and the ideal path
2. the cost of using an action a $C_a(a)$ occurs each time the rover performs the action.
3. the changing cost $C_c(a, a')$ is paid each times de rover changes its current action
4. the distance $C_i(\langle x, y, \omega, n\rangle)$ that we will discuss later on.

In this context, T and C are sligthly different from the usual definition because we have to take the cost of the action change into account. We will detail all costs in the next section.

To solve the MDP we have to calculate a value function. This Value function V is as usually calculated with the Bellamn Equation. But in our context, we have to take the change cost into account. Thus, in order to memorize the last action, we calculate Q values on pairs of (state, action). The objective here is to minimise the cost.

$$Q(s, a) = \min_{a' \in A} \sum_{s \in S} C(s, a, s', a') + T(s, a, s').Q(s', a') \tag{5}$$

We compute the Cost value to the last line of the last grid. Then, the calculation is done backward from the last grid to the first one. In the last grid, the algorithm computes the cost value to the last line. Then, the entire Q-Value function is calculated backward on the grid according to the previous equation. The policy is finally saved in a file. Then, the algorithm step back to the previous grid until the first grid is reached.

$$\pi^*(s) = argmin_a Q(s, a) \tag{6}$$

Only grid G_{i+1} remains in memory during the calculation of grid G_i's values. All previous ones are freed after being saved in a file. The state space is strutured on ordered grids where each grid is a subpace that can be considered solely for computation. This is an advantage of having an acyclic graph structure. The program does not need a lot of memory, since the memory space is an important issue of the trooper rover from THALES. Some states are sometimes declared two times when many rectangles overlap each other. In this case, the algorithm does consider the latest state, i.e. the one in the grid G_{i+1}.

Once the policy is entirely calculated, the rover only has to read the files in order to select the best action.

3.2 Cost Functions

The cost function is divided into four parts.

1. the distance cost $C_d(\langle x, y, \omega \rangle)$ measures the distance between the current state and the ideal path
2. the cost of using an action a $C_a(a)$ occurs each time the rover performs the action.
3. the changing cost $C_c(a, a')$ is paid each time the rover changes its current action
4. the distance cost $C_i(\langle x, y, \omega, n \rangle)$.

C_d measures the distance cost between the current state and the ideal state. Equation 8 calculates this cost (also see Fig. 3). This measure is divided into two parts. It measures first the euclidian distance from the straight line Δ_i in relation to the rectangle width l. Secondly, it measures the angle deviation from the current ideal orientation α_i. There is no penalty when the angle brings the rover closer to the straight line Δ_i. Thus, equation 8 is applied to the left part of the rectangle only. We introduce a β weighting factor (generally equal to $0,5$) in order to give more importance to the distance or to the deviation.

$$C_d(\langle x, y, \omega \rangle) = \beta . \frac{d((x, y), \Delta_i)}{l/2} + (1 - \beta) . \frac{max(0, \omega - \alpha_i)}{\alpha_{max}} \tag{7}$$

In order to avoid to use the odometer that is quite imprecise, we introduce a using cost for each action. $C_a(a)$ is a constant positive integer. When the rover activates an high-level method like move toward an object, it monopolizes an effector like a camera. It

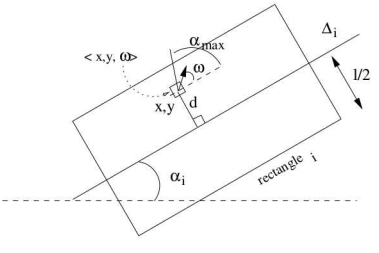

Fig. 3. Distance cost in rectangle i

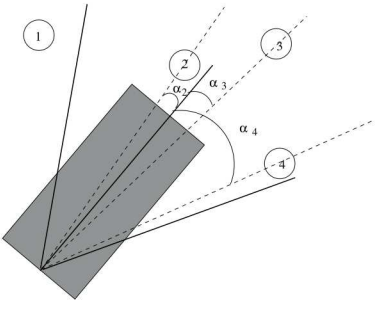

Fig. 4. Objects eligibility

also takes some time to locate the target. To take this into account, we define a change cost $C_c(a, a')$ as a 2 dimensionnal matrix.

Finally all costs are added together to form the global cost. Factors w_d, w_a, w_c (and w_u) weight this sum, they could be adapted during a learning step.

3.3 Object Eligibility

The rover can only move toward an unique object at a given time. But it has many sensors (or camera) ready to capture an object. Thus in each rectangle of the path, we have to select 0, 1 or 2 objects. This selection is made like it is shown in Fig. 4. All objects outside a $\pi/6$ angle are excluded from the selection. Then we keep the two closest objects if there are any. In this Fig., object 2 and 3 will be selected.

The Wall selection is quite similar. We select the closest wall from the left/right side of the rover if it is not too far from the trajectory (e.g. three meters).

3.4 Including Uncertainty in the State Description

While moving with the help of the odometer, the rover has an uncertain position. The longer it moves without assigning a helping object, the more uncertain it will be about its the position. To modelize this, we define N_u levels of uncertainty. Thus we add a variable to the state. It becomes $\langle x, y, \omega, u\rangle \in \mathcal{R} \times \mathcal{R} \times]-\pi, \pi] \times [1, N_u]$. Each uncertainty level corresponds to an elliptical area E that represents where the rover can be. A normalized gaussian curve indicates the position probability ($P(\langle x, y, \omega\rangle) \rightarrow]0, 1]$). In Fig. 5, there are 4 level of uncertainty. At the 4^{th} level, the rover could be in a failed state. Each time the rover moves with the odometer, the uncertainty increases to the next level $(s_1, \ldots, s_4)$. If the uncertainty level is equal to the maximum N_u, the level stays at this maximum. Elliptical range and their associated gaussian are predefined (as the number N_u).

With this uncertainty level, we can introduce the uncertainty cost. It is the probability for being in a failed state mulitplied by the failed state cost.

$$C_u(\langle x, y, \omega, u\rangle) = V_{FAIL} \int_{x,y,\omega}^{E\times]-\pi,\pi]} P(\langle x, y, \omega\rangle = FAIL).dx.dy.d\omega \tag{8}$$

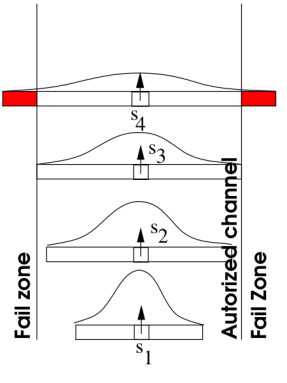

Fig. 5. Uncertainty in the model

Where FAIL is the failed state and V_{FAIL} a constant value. It has an high cost. When the rover uses an high-level method, uncertainty automatically decreases to 1. The object and the walls give a good localization to the rover. This cost is in our opinion more realistic that the cost C_a. But it also increases the algorithm complexity.

3.5 Complexity

The algorithm we presented previously is linear in the state space. The result of this policy is a policy that the rover must follow once it is calculated. Thus the complexity depends on the path total lenght L_T, the channel width l, the grid ratio size Δ, the angle number, the uncertainty levels and the action number (here 5). This leads to the following complexity :

$$L_T * l/\Delta^2 * N_\alpha * N_i * A \tag{9}$$

In order to have a very precise policy, we can increase N_α and N_i or decrease Δ. This model is interesting because it's adaptive. The policy can be calculated very quickly if necessary. Drawback of MDP model is their lack of adaptivity when th environment changes after the policy has been calculated. In this paper, we can take some changes in the environment into account without calculating the policy entirely.

4 Dynamic Environment

During the exploration mission, the path never changes. But sometimes, new objects or walls that the rover had not detect yet could appear and be very helpfull. Generaly, problems modelized with MDP model calculates a definitive policy that indicates the rover the action to select during execution time. But here, the graph induced by the MDP is acyclic. Thus, if something change the policy or the value function in the i^{th} rectangle of the path, only the i^{th} first rectangle of the path are affected. Moreover, if this change appears during the mission, the change affects the rectangle from j to i if the

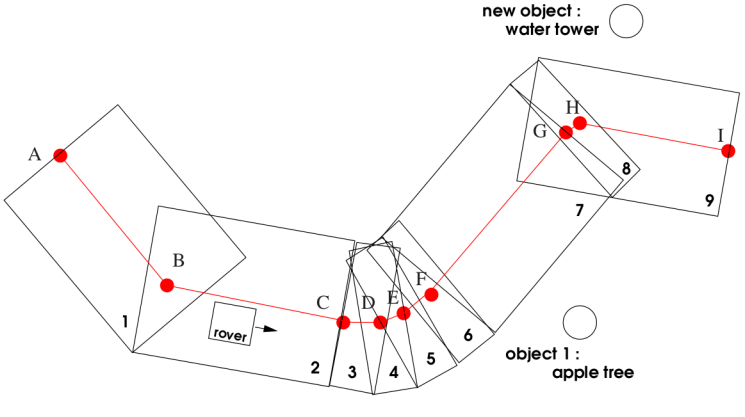

Fig. 6. A new object in the environment does not affect the entire policy

rover is currenctly in rectangle j. Fig. 6 explains this principle. The rover in rectangle 2 has detected a water tower in the distance. This object can not be usefull for now. But it will be for rectangles 4, 5, 6, and 7. Then it has to calculate a new policy from rectangle 7 to rectangle 2.

When a new interesting object is detected, we first search the first rectangle where this object is eligible. If no rectangle is affected, the policy does not change (this object was not so interesting). Otherwise, we load the policy in rectangle i+1 and calculate the new policy from i to j very quickly. Then, the rover selects new actions according to the new policy.

5 Experimental Results

We develop a C++ program that calculates the MDP policy on a given trajectory. We have included a graphical interface that shows the policy and also the Q-Value functions.

5.1 Interface

This is a very helpful tool to simulate the rover's behaviour. With this graphical interface, you can : load a trajectory from a file, see the corresponding channel and rectangle around this trajectory, add a new object or a new wall, calculate the Q-Values and the policy, see the Q-Values and the policy, change the parameters Δ, N_α ...

A trajectory file is a set of points. After being loaded into the main application, the user can see the corresponding channel inside which the rover has to move (Fig. 7).

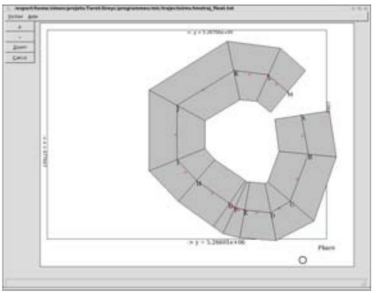

Fig. 7. A trajectory and the corresponding channel

We define a color gradient green, yellow, orange, red that represents values. Green corresponds to 0 while red corresponds to V_{FAIL}. Value functions are painted for different reasons.

1. showing the policy value function, i.e. the value of the best action for each state.
2. it also shows all QValue function values, in order to see the values for a particular function.

The goal is to minimize the cost. Thus the typical result is to have a red on the channel's border and green in the middle. The beginning of the path is more red than the end because the state's values are an accumulation (or sum) of atomic using costs (plus other costs). The value function decreases slowly from the beginning to the end of the path.

The parameters can also be changed directly in the interface. When Δ is very small, there are lots of states. In this case the solution comes slowly. The cost function weighting factors $w_d, \ldots w_u$ can be changed before calculating the policy. It has a great influence on the value function. We tuned this parameters in order to have a "good" value

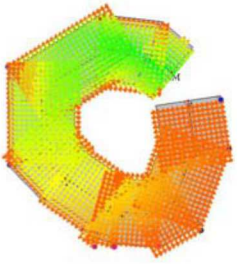

Fig. 8. A value function without object

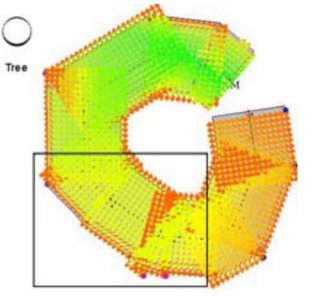

Fig. 9. With an Object

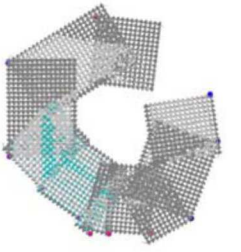

Fig. 10. Policy

function. We do not want to have a completely red value function neither a completely green function but a gradation of colors. Without helping objects , we obtain the value function shown in Fig. 8

In Fig. 9, the rover follow the tree in the dotted rectangle area. We see that the values are lower in this region than in Fig. 8. The policy of the first case (Fig. 8) is not depicted because there is only one action : trust the odometer.

5.2 Running Time and Memory Requirements

We measure time and memory required for some cases.

Total length L_T (meter)	1672,64	1672,64	3092,43	3092,43
Channel width l (meter)	20	20	20	20
Grid ration Δ	10	1	1	1
Number of angle N_α	21	21	21	5
State space	8.10^5	4.10^6	$7,3.10^6$	$1,7.10^6$
running time (s)	1	62	119	27

We used a 2.60GHz CPU to perform these tests. The function that take the more time is the state's transformation. The memory requirement is not so huge because grids are saved into file one per one during the calculation.

5.3 Behaviour Analysis

We add some objects in order to see whether the behaviour change or not. In this example, we add an object in the rigth up corner. It can help the rover at the beginning of the turn. Effectively, it does. The value function turns into green in the rectangle where this object is elligible. As a result, the corresponding policy also change : the rover move towards this object when it is on the right part of the rectangles. It does not move towards it on the left side because in this case it would come off [or leave] the channel. In

Fig. 10, the rover moves towards an object when the states are blue. Otherwise, it just follows the path without help.

6 Conclusion

In this paper we consider the problem of an autonomous rover that have to follow a path. This is developped in a project in collaboration with THALES company under the grant of DGA. The rover has many methods to direct itself. It can move towards object, follow a wall side by side or use his imprecise odometer. We showed how the rover can select the best method to use in order to follow the trajectory. This selection is based on a Markov Decision Process framework. A policy is calculated off-line. On-line, the rover reads the instructions. MDP framework often suffer from a lack of adaptivity, but we found a method that transforms the problem into an acyclic structure. Then, the policy calculation time is linear in the state space. This allows us to consider dynamic changes during running time : it is now possible to compute a policy with a new environment quickly. Some experiments have been developped to test running time and memory requirement. The behaviour of the rover is interesting : it helps itself with objects in order to follow the path. This automatic selection method is under integration on a six-wheel Trooper rover soon equiped with these behaviours.

References

[Bel57] Bellman, R.E.: Dynamic Programming. Princeton University Press, Princeton (1957)

[BM07] Benhimane, S., Malis, E.: Homography-based 2D Visual Tracking and Servoing. The International Journal of Robotics Research 26(7), 661–676 (2007)

[CBC+05] De Cabrol, A., Bonnin, P., Costis, T., Hugel, V., Blazevic, P.: A new video rate region color segmentation and classification for sony legged robocup application (2005)

[LMC08] Gloannec, S.L., Mouaddib, A.-I., Charpillet, F.: Adaptive multiple resources consumption control for an autonomous rover. In: Bruyninckx, H., Přeučil, L., Kulich, M. (eds.) Proc. European Robotics Symposium 2008 (EUROS 2008), pp. 1–11. Springer, Heidelberg (2008)

[MZ00] Mouaddib, A.-I., Zilberstein, S.: Optimizing resource utilization in planetary rovers. In: Proceedings of the 2nd NASA International Workshop on Planning and Scheduling for Space, pp. 163–168 (2000)

[Put94] Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., New York (1994)

[SRC06] Segvic, S., Remazeilles, A., Chaumette, F.: Enhancing the point feature tracker by adaptive modelling of the feature support. In: Leonardis, A., Bischof, H., Pinz, A. (eds.) ECCV 2006. LNCS, vol. 3952, pp. 112–124. Springer, Heidelberg (2006)