A Proposal for Self-Recognition in Robot Programming

Kamran Karimi[1], Mehran Mehrandezh[2], Howard J. Hamilton[1]

[1] Department of Computer Science
University of Regina
Regina, Saskatchewan
Canada S4S 0A2
{karimi, hamilton}@cs.uregina.ca

[2] Department of Industrial Engineering
University of Regina
Regina, Saskatchewan
Canada S4S 0A2
Mehran.Mehrandezh@uregina.ca

Abstract

In this paper we propose a way to perform robotic self-recognition in static and quasi-static environments. self-recognition is a process during which the robot discovers the effects of its own actions on the environment and itself. For example, how much would the robot move when its wheels turn once? Such information can be hard-coded into the robot's code, but it can also be discovered automatically. self-recognition can be used when the robot or the environment are unknown a priori or change. Examples are when new wheels or a completely new robot is used, or when obstacles are added to the environment. Instead of re-programming, the user can let the robot "play" and discover how to change its situation. In this way only the self-recognition phase should be repeated, while the high level planning strategy can remain the same.

Keywords: Robot Programming; Data mining; Artificial Intelligence.

1. Introduction

Programming robots that move in environments not known a priori usually is a non-trivial task. A domain expert may be the only source of knowledge about the environment and the task to be performed by a robot, because the semantics of the domain are not evident from a robot's perspective. To achieve a goal, not only the input from the sensors should be analysed and a plan of actions produced, but the results of each action in the environment of the robot must also be known. The knowledge about the effect of each action is commonly extracted by the domain expert based on the physical characteristics of the robot and the environment, which are then hard coded into the plan. The problem with this approach is that if the robot or the environment changes, the program should be changed too. Examples of change in a robot include different or new actuators, or a different design. An example of a change in environment would be a new room layout, or obstacles being added, removed, or moved around. After a change, the same commands as before the change may not result in the same outcome.

In this paper we are not concerned with the high-level plan generator, which we assume is written by a domain expert. Rather, we target the automation of the process of discovering the effects of the actions of a robot. This information is crucial to any high-level plan generator, and facilitates the re-adaptation of a robot motion planner once the robot or its environment has changed. Through a process that we call *self-recognition*, a robot is allowed to explore the effects of its actions. After that rules are extracted to represent the effects of any action by the robot. They are then used by a high-level plan generator, which, knowing the effects of the actions of the robot, comes up with a sequence of actions to guide the robot from the current situation to a desired situation. A robot that operates in a changing environment may be able to perform self-recognition as it explores the effects of it actions on the environment. However, some plans may fail because of lack of information about the environment (e.g. unexplored areas) or because of failed actions (e.g. obstacles were moved in the environment). If a plan fails, the self-recognition part of the planning process should be repeated.

There are three assumptions essential to our method. First, we assume a limited space of operation, and hence a limited search space. This requirement is satisfied for robotic arms or mobile robots that operate in a confined area. The second assumption is that the robot and its environment are static or quasi-static, that is, the shape of the robot or the locations of obstacles do not change often. The reason for these assumptions is that the process of self-recognition is performed off-line, so it cannot learn as it is executing a plan. The third assumption is that the robot can both affect and perceive its environment, so that it can register the effects of its own actions.

The rest of the paper is organised as follows. In Section 2 we present the self-recognition process. Section 3 shows why some of the characteristics of the self-recognition method, such as the preference towards exhaustive searches, can be of use. Section 4 briefly considers a possible implementation of the self-recognition method. Section 5 concludes the paper.

2. How Self-recognition works

The self-recognition problem is defined as automatic discovery of the effects of an action performed by the robot in its environment.

We consider two possible ways to represent the location of the end point of a 2-dof (degrees of freedom) planar robotic arm: at a low level, we can use the angular position of the shoulder and the elbow, as in the proposition Position(θ_s, θ_e) where θ_s is the angular position of the shoulder, and θ_e is the angular position of the elbow.

The problem with this approach is that many different configurations will correspond to the same robot's tip position, and this number increases exponentially with the degrees of freedom of the robot. We need to reduce this number if we are to be able to handle the representation and programming phases. For representing the position of the robot's tip one can use the grid number: Position(N) where *N* is the grid number where the tip is located.

With the location determined, we now consider observing the effects of the robot's moves. First the current situation of the robot and the environment are perceived through the robot's sensors. Then actions are generated and executed one by one. The possible changes in the robot and the environment are observed and recorded. For example if we are observing the robot's position, this method results in a series of data records of the form <$\text{Position}_{\text{current}}$, action, $\text{Position}_{\text{next}}$>. This change of the situations while performing an action has been the basis for Situation Calculus [3]. Since we can get to the same situation through different paths, the result is a graph where nodes are the positions. The nodes are connected to each other by the edges that are actions, and lead from one position to another. The robot thus learns how it can move from one position to the next. In a finite space it is possible for the robot to explore all the positions.

After obtaining the motion graph, we can use it to go to a desired node. To do this a high-level planning strategy selects a desired goal (node) for the robot, and planning is reduced to finding a path within the graph from the current node to the destination. The graph is traversed node by node. Failure recovery is automatically provided. If the results of an action do not lead to the expected situation (the wheels slip, or the user moves the robotic arm away, for example), then the new node is noted and a new path, from the current node/situation to the desired one, is plotted. The plan fails if it is not possible to come up with a path from the current (unexpected) situation to the destination.

After the robot has explored its space, possibly covering some positions many times, the exploration can be stopped. At this point the rule discovery phase begins. This can be carried out by different methods. In a classification problem, a program such as C4.5 can be employed. One example rule could be: if {($x = 1$ AND $y = 3$ AND *action* = Right)} then $x = 2$. This rule determines that when in position (1, 3), a move to the right will result in an increase in the x value and result in position (2, 3).

The main requirement is that all the grid points have been explored by the robot's tip. It should be noted that this requirement is much easier to accommodate than the need to cover all angular values that the robot's joints can accept. When all the grid points have been covered (the tip has been in all of them, and the start position has been visited at least twice), we are guaranteed that we can navigate the tip anywhere.

This is done by backward chaining, which means that to go from one grid location to the next, we plot a route backward from the destination node to the source node, and then follow the route in the forward direction.

The rules can thus be employed by a plan generator, which determines where the robot should go to, and then consults the rules to navigate the robot from its current position to the desired position. If the robot or the environment changes, new rules should be extracted, but the high-level planning strategy can be left unchanged.

We have developed the TimeSleuth software to automatically extract rules from raw data. TimeSleuth is a tool for discovering causality based on time [5]. It is a suitable aide in plan generation because a plan is a sequence of actions that causes change in the environment. In this paper we assume that the effects of each action are known in the next time step, but with TimeSleuth the user can investigate the possibility that the effects will be known after a number of time steps. Also, TimeSleuth can output the rules it discovers as Prolog statements, which can then be readily executed, or converted into complete plans after some modifications [3, 4].

2. Justification of the proposed approach

In this section some problem domains, in the realm of robot motion planning in which the proposed method can be applied are investigated. The classical robot motion planning problem is defined as; "finding a collision-free path for a robot equipped with sensors to move from a start configuration to a goal configuration in environments not known a priori". This is also referred to as a point-to-point motion planning problem [6]. Additional constraints such as the minimum length of the path, its maximum clearance with obstacles, and the overall motion time will contribute to the complexity of the motion planning problem further. Minimizing the overall motion time is of particular interest in scenarios in which the environment is changing (i.e., dynamic environments). An example would be the robotic interception of moving objects, [1].

Many researchers have studied the robotic interception problem in the past two decades. A variety of techniques have been suggested under the assumption that no static obstacle exists in robot's work space, [e.g., 2, 7]. Mehrandezh et al. developed a novel time-optimal motion planning of co-operating robotic arms in a sequential task-sharing mode in presence of obstacles, [8]. Time-optimal robotic interception problem in presence of static obstacles needs an exhaustive search method (i.e., dynamic programming). There is always a trade off between the complexity of the interception algorithm and the deviation from the global optimal motion time. Existing time-optimal interception schemes cannot be implemented in real-time due to the massive calculations needed. Therefore, real-time implementation of these techniques remains a challenging problem. The proposed self-recognition strategy would take the burden off the shoulders of the motion planner, making real-time interception planning realizable.

In general, robotic interception problem can be divided into two stages; (1) coarse, and (2) fine motion planning. In the first stage, the motion planner will bring the robot to the vicinity of the moving object. In the second stage, a fine-tuned motion-tracking algorithm takes over, bringing the robot to precise interception location for capturing the object. The self-recognition technique can provide the coarse motion planning stage with information that can be utilized to achieve a fast (but not very precise) interception planning strategy.

One should note that the low-level rules derived through the self-recognition algorithm are all position/action-based. A mapping will be needed to translate these position/action-based rules to a position-velocity space. Overall motion time of the robot moving between any two configurations (given that these two configurations belong to the graph developed by the self-recognition technique) can be deducted from the information embedded within this position-velocity space. This would facilitate the real-time implementation of the high-level interception planning strategy.

4. Implementation considerations

Implementing self-recognition in a real environment requires some considerations, as we will describe in this section for a robotic arm. Since we are using an exhaustive search method, we need to reduce the size of the physical space. We achieve this by discretising the space (see Figure 1).

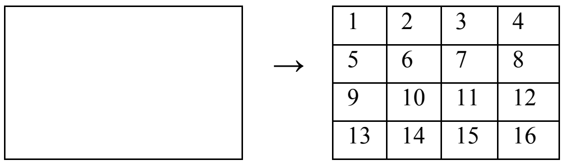

Fig. 1. The space is discretised into 16 different locations

We represent each situation as the place where the tip of the robot lies. The *Markov condition* states that the history of a system is of no effect when performing an operation at the present time. This is a great simplifying assumption, and holds in the artificial robot environment. To illustrate this concept let us assume that the robot's tip is located on the cell 6. The robot can be inside cell 6, but not exactly on the centre. If each time the arm is in cell 6 the exact location differs, then after a while a specific amount of movement may result in positioning the robot's tip in a different cell. For example, in one case the tip may move to position 7, and in another case it may remain in position 6 because it has been placed too far from cell 7. This reliability on the history of the tip's movement means that the Markov condition is violated. The same rule may fail or work, depending on how the tip was moved before. This property complicates planning, and must be resolved.

For this reason we assume that the tip always positions itself on the centre of a grid location. The robot will move from one point to another, along the *x* or *y* axis (and *z,* if the space is three-dimensional). If the robot detects a problem in moving to the destination, either due to a collision or because the robot's limits of movement have been reached, then the tip will be put back on the starting position. For each tip position there can be many angular values for the robot's joints. This number is two for a robot with 2 degrees of freedom, one called the elbow-up position and the other the elbow-down position. For a robot with more degrees of freedom, this number can be very large. With two degrees of freedom we can compute the angular values easily. For more degrees of freedom, the problem can be solved by finding solutions for $min\{w_1 \times d^1 + w_2 \times d^2 + w_3 \times d^3\}$, where we assume the robot has 3 degrees of freedom, *w*s are weights, and *d*s are the angular values for the joints. The computed values are stored in tables, to be used for mapping from the physical space to the configuration space (i.e., a space where the robot can be represented as a point denoting its configuration). To move from one point to another, the source and target angular values will be extracted, and used to guide the robot's tip. The following pseudo code explains the idea:

```
#define X_DISCRETE 10
#define Y_DISCRETE 10
int shoulder_table[X_DISCRETE][Y_DISCRETE];
int elbow_table[X_DISCRETE][Y_DISCRETE];
// generate the data needed for movement, then play
Computer_Angular_Values();
loop {   //  play as long as needed
  $x_0$, $y_0$ = Get_Current_pos();
  M = Generate_Next_Move(); // select destination
  $x_1$, $y_1$ = Determine_Where_M_Is();  // Which grid?
  Shoulder = shoulder_table[$x_1$][$y_1$];
  Elbow = elbow_table[$x_1$][$y_1$];
  Robot_MoveTo(current_pos, next_pos); // try a move
  $x_1$, $y_1$ = Get_Current_Pos();   // where did we end up?
  Output($x_0$, $y_0$ , M, $x_2$, $y_2$ );   // display the results
 }
```

During the self-recognition phase, the destination can be selected by any method. The moves can be generated randomly, systematically, or by some other heuristic.

We assume the environment is quasi static, meaning that changes do not happen often. This allows us to be able to use the rules that are extracted for a while, before being forced to generate them again, as shown in the following algorithm:

```
forever do {
 observations = motion_generation();
 rules = learning(observations); // TimeSleuth
 while(planning(rules) == true); // break if failure
}
```

If a plan fails, then an assumption about the environment has become violated. At this point the robot explores the area again, and generates a new set of rules. In a quasi-static environment this will not happen very often.

A reliable robot motion planning technique would require a huge amount of online collision detection which can be computationally expensive. By using the rules generated at the learning phase, we do not have to check for collisions any more, reducing the computational requirements considerably. Skin-type sensors can be put on the robot in the self-recognition phase to detect collisions and then removed after the environment has been explored.

We have experimented with this method to create a planner for an artificial robot that moves in a 2-Dimensional environment called URAL. The rules governing the movements of the robot are converted into Prolog statements and then combined to guide the robot from a starting position to a finishing position [3, 4].

In what follows X refers to the robot's position along the x axis, and A refers to the action (direction of movement). Table 1 shows parts of an example set of Prolog statements generated by TimeSleuth when the decision attribute is x_2. In general, for an $N \times M$ board, there are $N \times 4$ Prolog statements generated for moving along the x axis (from left, right, top, or bottom positions). Likewise, there are $M \times 4$ statements for moving along the y axis.

class(A1, X1, 0) :- A1 = 2, X1 = 1.
class(A1, X1, 2) :- A1 = 3, X1 = 1.
class(A1, X1, 3) :- A1 = 2, X1 = 4.
class(A1, X1, 3) :- A1 = 3, X1 = 2.

Table 1. Sample Prolog statements

In Table 1 a value of 2 and 3 for action *A1* could mean going to the left and right, respectively. Following a classification terminology, the results are designated by a predicate called "class." The condition attributes (action *A1* and position *X1* in this case) come first, and the value of the decision attribute (the *next* value of x) comes last. In the head of the rules, the condition attributes are used for the decision making process. In our example temporal data, *A1* and *X1* belong to the current time step, while the classification is done for the value of x in the next time step. For example, the first Prolog statement in Table 1 says that if the robot is at position 1, and there is a move towards left (action is 2), then the next value of x is 0.

There is no mention of the y axis in this table because in this example the movements along the x and y axis are independent of each other. These statements can be executed in both a forward and a backward direction. For example, the user can invoke the command class(A1, X1, 2) (what should the robot do if it wants to move to the x location 2? The results will be a list of actions/position combinations that will lead to $x = 2$. Conversely, the user can issue queries like class(2, 4, X2) (where does the robot go from $x = 4$ if it moves to left?) to determine the effects of an action. If we are dealing with more than one sensor attribute (x and y for example) we could rename "class" to something like "classx" to avoid name clashes.

For planning we assume that we are already at the destination and then find a way back to the starting location. For example suppose we want the robot to go from location 1 to location 3 along the x axis. In Table 1, from the fourth statement we find a way to go from location 2 to location 3, (by moving right). Now the problem is to get from location 1 to location 2, and the second statement in Table 1 has the answer (move right from 1). The plan now consists of starting as location 1 and moving to the right twice by executing the second and the fourth Prolog statements. For more details of planning with such Prolog statements see [3, 4].

5. Concluding remarks

In this paper we proposed a method for a robot to discover the results of its own actions. Once this information is available, a planner can put them together and create high level plans. We argue that with self-recognition, the high-level planning algorithm does not need to be changed as the robot or its environment changes. The basic rules regarding the results of an action do change, but self-recognition proposes a play phase in which these results are discovered automatically. The domain- and robot-dependent parts of the problem are handled by automating the task of detecting the effects of the actions of the robot. The main draw back is the need for exhaustive search of the domain, which makes this method less suitable in large domains or when a high amount of uncertainty associated with sensory data or probability of failure is present.

TimeSleuth and URAL can be downloaded freely from http://www.cs.uregina.ca/~karimi/downloads.html.

References

[1] Geering, H. S., Guzzela, L., Hepner, S. A. R., and Onder, C. H., Time-Optimal Motions of Robots in Assembly Tasks, *IEEE Trans. On Automatic Control,* Vol. 3, No. 6, June 186, pp. 512-518.

[2] Hujic, D., Croft, E. A., Zak, G., Fenton, R. G., Mills, J. K., and Benhabib, B., Time-Optimal Interception of Moving Objects – An Active Prediction, Planning and Execution System, *IEEE/ASME Trans. On Mechatronics,* Vol. 3, No. 3, Sept 1998, pp. 225-239.

[3] Karimi, K. and Hamilton, H.J., Learning With C4.5 in a Situation Calculus Domain, *The Twentieth SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence (ES2000)*, Cambridge, UK, December 2000, pp. 73-85.

[4] Karimi, K. and Hamilton, H.J., Logical Decision Rules: Teaching C4.5 to Speak Prolog, *Second International Conference on Intelligent Data Engineering and Automated Learning (IDEAL'2000)*, Hong Kong, December 2000, pp. 85-90.

[5] Karimi, K., and Hamilton, H.J., Distinguishing Causal and Acausal Temporal Relations, *Seventh Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'2003)*, Seoul, South Korea, April/May 2003.

[6] Lozano-Perez, T., and Wesley, M. A., An Algorithm for Planning Collision-Free Paths among Polyhedral Obstacles, *Commun ACM,* 22:560-570.

[7] Mehrandezh, M., Sela, N. M., Fenton, R. G., and Benhabib, B., Robotic Interception of Moving Objects using an Augmented Ideal Proportional Navigation Guidance Technique, *IEEE Trans. On Systems, Man, and Cybernetics – Part A: Systems and Humans,* Vol. 30, No. 3, May 2000, pp. 238-250.

[8] Mehrandezh, M., and Gupta, K., Sequential Task Sharing for Co-operating Robots in Presence of Obstacles, *IEEE International Conference on Systems, Man and Cybernetics (SMC'2002),* Tunisia, October 5-9, 2002.