Logical Architecture for Robot Path Planning

Román Katz and Claudio Delrieux

Universidad Nacional del Sur, Dept. Electrical Eng. and Computers
(8000) Bahia Blanca - ARGENTINA
(voice) (54)(291) 4595101 ext. 3381 - (fax) (54)(291) 4595154
claudio@acm.org

Abstract. In this work we present a robot path planning architecture based on Lee's routing algorithm. This algorithm was originally conceived to obtain minimal conductor nets among terminals on VLSI circuits. In our system it provides a simple and robust mechanism to compute near to shortest free paths in two-dimensional configuration environments. We incorporate this algorithm in a planning system, and the resulting architecture can be used to generate trajectories in order to guide a mobile agent from an initial point to a final (static or moving) position of the working space, avoiding not only static but also moving obstacles.
The guidance engine is implemented as a rule-based version of Lee's algorithm, embedded in a multi-paradigm software system that performs logical inferences in a Prolog component, whose results are provided to an imperative and numerical processing compiled component. This allows a neat factorization of the functionality of the system: high level representation of the navigation algorithm is implemented through its logical formulation, and easy and natural means to acquire the layout of the environment with computer vision and image processing techniques is implemented via numerical programming. The same imperative approach is also natural to provide 2D or 3D visualization and simulation features to aid to understand the ongoing process. Thus, our architecture features both the speed and versatility of a visual language application, and the abstraction level and modularity of a logical description.

1 Introduction

The problem of robotic navigation is usually approached in two very different ways, one involving kinematics and dynamical modeling and other involving spatial and geometrical modeling [1]. In the former, a feedback control law that deals with torques to manipulator joints and drive wheels is generated, in order to track a supplied reference trajectory. The use of *artificial potential fields* [2, 3] is commonly implemented under this viewpoint. In the latter, a trajectory or path is generated through the associated space of possible configurations of the

robot and the objects in the working area, considering their positions and the desired final position of the robot [1].

Certainly, both approaches have advantages and disadvantages, and their well defined and disjoint fields of applications . The kinematics and dynamical view —as the one in the artificial potential fields— is used mainly in real-time guidance applications [1] and the programming techniques are based essentially on numerical processing. On the other hand the spatial and geometrical modeling is considered in the realm of planning [4, 5], with areas of application mainly in high level off-line navigation tasks [6]. Here the techniques employed are closer to symbolic processing [7, 8], where the domain knowledge is represented in a declarative manner, for example, in first order logic. Therefore, it seems sensible to consider the implementation of hybrid guidance systems, which integrates the path planning engine based on a Prolog version of Lee's routing algorithm [9] into a visual programming language.

The paper is organized as follows: Section 2 gives a brief summary of the Lee's algorithm and how it can be used for path planning purposes. Section 3 extends the proposed path planning architecture for moving obstacles avoidance. Sections 4 and 5 present the simulation and experimental results and the conclusions, respectively.

2 Lee's Routing Algorithm

In this Section we give some details of Lee's algorithm and how to use it for path planning purposes. This algorithm was originally conceived to obtain minimal conductor nets among terminals on VLSI circuits, and in the proposed system it provides a simple, robust mechanism to calculate near to shortest free paths in real environments. First, consider the two-dimensional working space represented as the regular grid shown in the layout in Fig. 1. The working space may contain obstacles (blue boxes), and then the problem consists in finding an adequately short free path between two specified points. The solid black line joining this two positions (red and blue small circles) represents the shortest path between these points.

A Prolog version of Lee's algorithm [9] adapted for this path planning problem, is shown in Fig. 2. In our algorithm, points are represented by their Cartesian coordinates (denoted X-Y) and the paths are calculated by the program as a list of points. Initially the program displays the relation `lee_route` and its two correspondent stages `waves` and `path`. The predicate `waves` generates successive neighboring grid points, starting from one initial point, until the other point is reached. These *waves* are sets of points that neighbor a point in the previous wave and have not already appeared. The predicate `path` gets the *path* using

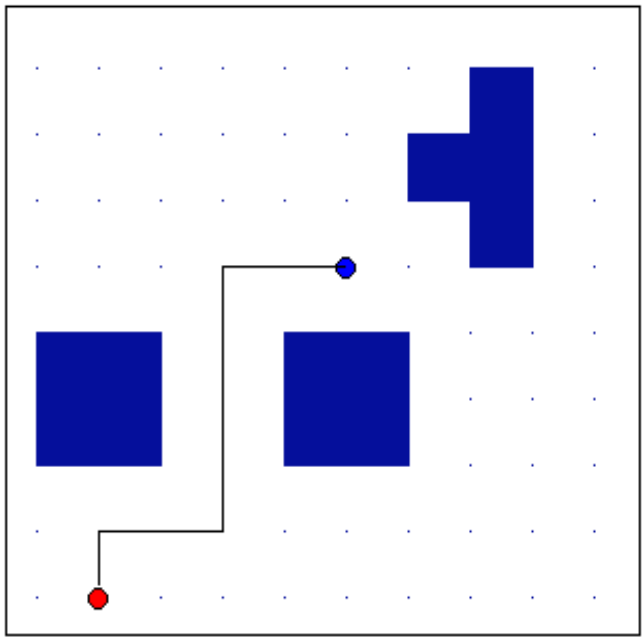


Fig. 1. Generic layout of the working space.

the found *waves* downwardly, then the order of the points ranges from the final point to the initial one.

In this implementation, rectangular block obstacles and complex obstacles are both considered. The former are represented by terms of the form `obstacle(L,R)`, where L is the set of 2-D coordinates of the lower left-hand corner and R of the upper right-hand corner. The latter are handled using a list of their composing rectangular blocks with the generic structure described above (i.e., the list `[obstacle((2-3),(4-5)), obstacle((3-3),(4-5))]`). In both cases the obstacles could be easily included into the knowledge database through the Prolog built-in predicates `assert` or `assertz` [10].

As we pretended a system capable of getting the layout of the working space dynamically, we have made some changes to the original Lee's Algorithm presented in [9]. For instance, we define the predicate `get_obstacles` that makes effortlessly the process of recognizing the obstacles at any moment during the runtime. This is something highly profitable for the moving obstacle avoidance scheme that will be further described (Section 3). In particular the predicate `get_obstacles` must be computed before `test_lee` in order to update the *obstacles* that requires the correct execution of `lee_route` and also of `test_lee`.

```
lee_route(A,B,Obstacles,Path):-
             waves(B,[[A],[]],Obstacles,Waves),
             path(A,B,Waves,Path).

waves(B,[Wave|Waves],Obstacles,Waves):-member(B,Wave),!.
waves(B,[Wave|[Lastwave|Lastwaves]],Obstacles,Waves):-
             next_wave(Wave,Lastwave,Obstacles,Nextwave),
             waves(B,[Nextwave,Wave,Lastwave|Lastwaves],Obstacles,Waves).

next_wave(Wave,Lastwave,Obstacles,Nextwave):-
             setof(X,admissible(X,Wave,Lastwave,Obstacles),Nextwave).

admissible(X,Wave,Lastwave,Obstacles):-
             adjacent(X,Wave,Obstacles),
             not member(X,Lastwave),
             not member(X,Wave).

adjacent(X,Wave,Obstacles):-
             member(X1,Wave),
             neighbor(X1,X),
             not obstructed(X,Obstacles).

neighbor((X1-Y),(X2-Y)):-next_to(X1, X2).
neighbor((X-Y1),(X-Y2)):-next_to(Y1, Y2).

next_to(X,X1):- X1 is (X+1).
next_to(X,X1):- X>0,X1 is (X-1).

obstructed(Point,Obstacles):-
             member(Obstacle,Obstacles),
             obstructs(Point,Obstacle).

obstructs(X-Y,obstacle((X-Y1),(X2-Y2))):-Y1≤Y, Y≤Y2.
obstructs(X-Y,obstacle((X1-Y1),(X-Y2))):-Y1≤Y, Y≤Y2.
obstructs(X-Y,obstacle((X1-Y),(X2-Y2))):-X1≤X, X≤X2.
obstructs(X-Y,obstacle((X1-Y1),(X2-Y))):-X1≤X, X≤X2.

path(A,A,Waves,[A]):-!.
path(A,B,[Wave|Waves],[B|Path]):-
             member(B1,Wave),
             neighbor(B,B1),
             !, path(A,B1,Waves,Path).

get_obstacles(Previous,List):-
             obstacle(A,B),
             not member(obstacle(A,B),Previous),
             conc([obstacle(A,B)],Previous,Next),
             get_obstacles(Next,List),!.
get_obstacles(List,List).

test_lee(Source,Destination, Path):-
             lee_route(Source, Destination, Obstacles, Path).
```

Fig. 2. Prolog version of Lee's Algorithm.

Finally, we will now specifically formulate the navigation problem, so that the previous algorithm can be used to generate trajectories in order to guide a mobile agent of a regular size from an initial point to the desired final position of the working space. Fortunately, the translation of the trajectory-solving strategy provided by Lee's algorithm described before into a path planning architecture is almost immediate. For instance, the actual position of the mobile robot and the desired final position are respectively the previous initial point (red small circle) and final point (blue small circle) and the path joining these points among the obstacles represents the shortest free sought trajectory.

On the other hand the acquisition methods of the environmental conditions in the working space are independent of the Prolog guidance algorithm, whatever the method applied the knowledge database can be always updated in the same manner using the predicates `assert` or `assertz`. As a matter of fact there are several different ways to obtain the layout containing the set of obstacles, being those provided by the Computer Vision and Image Processing techniques [11, 12] the most versatile and commonly employed.

3 Moving Obstacle Avoidance

In the preceeding Section we introduced an adaptation of Lee's algorithm for its use as a mobile robot trajectory generator in environment containing fixed obstacles. However the typical situation in real world working environments is much more complex. There may be errors and uncertainty in control and sensing [13] and the obstacles and the target can move during the tasks [14]. In this chapter we will discuss how to adapt the proposed algorithm to the situation where the obstacles and/or the target move.

To address the problem of possible presence of moving obstacles in the working space during the navigation task we consider initially the following restriction: it is mostly certain that the complete set of obstacles varies smoothly in the majority of the interesting situations. Therefore, only small changes may and does occur from a moment to another. These changes could be due either to alteration in the dimensions of the obstacles or modifications in their position. This relatively slow rate of change of the layout allows us to consider the implementation of a hybrid guidance system, which integrates a high level path planning engine with sensory data, regarding dynamically the situation of the environment and then adjusting on the fly the pragmatically best trajectory.

Under the mentioned circumstances, we propose a system that embeds the planning component with the sensory interface component, as well as the numerical processing tools and visualization features that helps the user to understand the ongoing process. In fact, the proposed general scheme for moving obstacle avoidance turns to be remarkably simple. It combines the use of some of the

predicates of Fig. 2 into a *repeat* loop, in which the *obstacles* and/or the destination are iterative refreshed and the *path* is recalculated if necessary. The pseudo-code of the procedure for avoiding moving obstacles is shown in Fig. 3.

```
set(Initial_point,Final_point)
get_obstacles(Obstacles)
test_lee(Path)
Actual_point←Initial_point
(Start moving)
repeat          (Show path)
        move_nextpoint
        Actual_point←Next_point
        get_obstacles(Obstacles)
        If interfer(Obstacles,Path) then test_lee(Path)
until (Actual_point=Final_point)
```

Fig. 3. Pseudo-code of the moving obstacle avoidance procedure.

We first have to define the extreme conditions of the required path, setting the initial an final points (*Initial_point* and *Final_point*) with the command `set`. After this the earliest layout with the present *Obstacles* is obtained (`get_obstacles(Obstacles)`) and the first *Path* is calculated (`test_lee(Path)`). The opening sequence is already done, the *Actual_point* is set as the *Initial_point* and the mobile robot can start its movement (Start moving). The sequence enters now into a *Repeat until* loop. It entails the repetition of several steps until the ending condition is reached.

During every iteration, several tasks are performed. First the current position of the mobile is displayed (Show path) Second, the mobile is driven to the next position in the 2-D working space (`move_nextpoint`). Third, the actual position *Actual_point* is updated to the *Next_point* newest position. Then, the algorithm performs the most important part of the loop (and also of the whole procedure), which is the evaluation of the impact of the changes in the environment. This is achieved through two consecutive steps. First, the layout is reloaded (`get_obstacles(Obstacles)`), and then any possible interference between these new set of *Obstacles* and the already computed *Path* is tested with the command `interfer(Obstacles,Path)`. If this evaluation shows interferences, a new trajectory towards the final position from the actual position will be computed (`test_lee(Path)`) among the new arrangement of *Obstacles*.

That set of procedures loops until the mobile gets the final position, this is when *Actual_point* is equal to *Final_point*. While this procedure is running, the

system continuously loads the information of the obstacles and their eventual changes into the knowledge database. This is done using the Prolog predicates `assert` and `assertz` (as we described in Section 2) and sensory devices.

In the situation where the target moves, then it is most likely that the previously computed path is rendered useless. For this reason, the algorithm should reconsider a new path every time that the target moves. This has a negative impact in the efficiency of the procedure, but several optimizations can be implemented (we discuss this issue in the conclusions). The pseudo-code of the procedure for path planning to moving targets while avoiding moving obstacles is shown in Fig. 4.

```
set(Initial_point)
set(Final_point)
get_obstacles(Obstacles)
test_lee(Path)
Actual_point←Initial_point
(Start moving)
repeat          (Show path)
       move_nextpoint
       Actual_point←Next_point
       get_obstacles(Obstacles)
       get_target(New_target)
       If interfer(Obstacles,Path) or New_target ≠ Final_point then
             begin
             Final_point ← New_target
             test_lee(Path)
             end
until (Actual_point=Final_point)
```

Fig. 4. Pseudo-code of the moving target procedure.

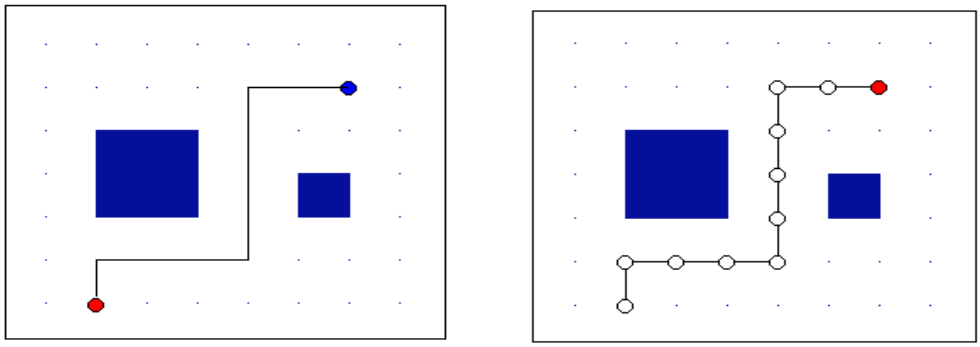
As we stated before, an adequate sensing interface is supplied by the standard Computer Vision and Image Processing (CVIP) techniques. This is the most adequate strategy, since many experimental and commercially available mobile robots employed in autonomous navigation have built-in cameras, image adquisition, and DSP devices [11, 3]. Therefore, a component-based architecture seems the most sensible way to integrate the hardware-provided module with the logical representation of the navigation strategy, and the visualization and other ancillary features. In this way, our system can be easily integrated with any device that can be programmed with standard general-purpose programming systems. CVIP techniques very frequently combine both numerical and

logical processing [15–17], and hybrid architectures as the one presented here are particularly interesting and useful.

4 Implementation and Experimental Results

However, integration of software units of different programming platforms in an application may be hard. This is why we chose the strategy of using a component-based programming system as Kylix$^{(TM)}$ (or Delphi$^{(TM)}$ under MS Windows$^{(TM)}$). In this way, a numerical software component may be integrated with a symbolic software component, in particular, a Prolog interpreter, within the same application. In this stage of our work, we are using copyright-free and freeware components (Kylix$^{(TM)}$ is distributed free for academic institutions, and the Prolog interpreter was downloaded as freeware from `www.trinc.nl`).

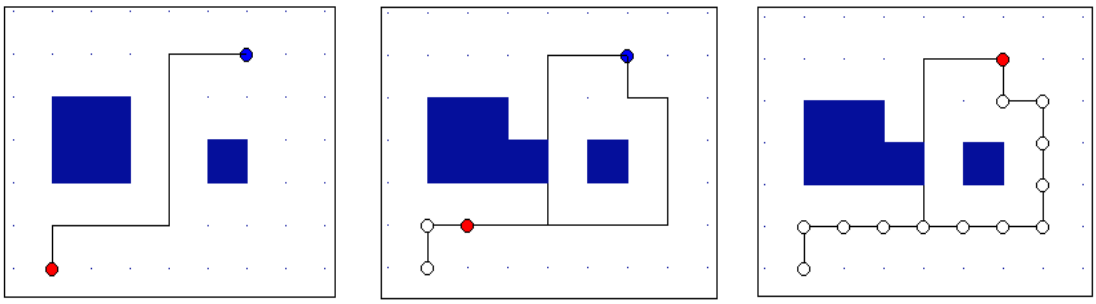
In Fig. 5 we show the resulting path of a simulation where the situation involved the avoidance of a set of static obstacles. Fig. 5(a) shows this static layout, with the actual position (small red circle), the target or desired final position (small blue circle) and the initially calculated trajectory joining these two points. Fig. 5(b) shows the mobile robot reaching the final position (small red circle), after having tracked the 2-D coordinates (small white circles) defined by the calculated path.



(a) (b)

Fig. 5. Simulation of the system solving a situation of static obstacle avoidance.

We show in Fig. 6 the system working in a situation of moving obstacles avoidance. Fig. 6(a) shows the initial layout, the actual position (small red circle) and the desired final position (small blue circle) of the mobile robot and the initial calculated trajectory joining these two points. Fig. 6(b) shows the mobile robot in an intermediate position, when a change in the position of the obstacles



(a) (b) (c)

Fig. 6. Simulation of the system solving a situation of moving obstacle avoidance.

produce interference with the first computed path. The trajectory is therefore recomputed, and the robot can avoid this obstacle, reaching in Fig. 6(c) the final position.

Various successful simulations were realized, testing the avoidance of static and moving obstacles, under several conditions of complex sets of obstacles and composed movements in the obstacles. In all the cases the results were remarkable, both in performance and in time of computation. The presented algorithm is robust enough to solve any situation where the obstacles move with linear trajectories. Deadlocks may appear if objects oscillate around a given position where the actual path the robot must traverse. A possible solution for this will be discussed in the conclusions.

5 Conclusions and Further Work

In this work we have presented a hybrid path planning architecture. The proposed system can be used to generate trajectories in order to guide successfully a mobile agent of a regular size from an initial point to a final position, avoiding not only static but also moving obstacles.

The guidance engine is accomplished through a rule-based version of Lee's routing algorithm, implemented within a Prolog interpreter component. The image processing and visualization components are implemented in a general purpose programming language. This allows neat high level representation of the navigation algorithm through its logical formulation and easy and natural means to acquire the layout of the environment with computer vision and image processing techniques implemented via numerical programming, while providing tools for 2D or 3D visual simulation and visualization of the ongoing process.

Two problems arose within this development, one concerning optimization of the path-generation loop when the target moves, and the other relating to dead-

lock avoiding when any obstacle oscillates across the actual path that the robot must traverse. We are currently considering a solution for these two problems, where the solution involves a meta-planning procedure that regards the actual and the previously considered paths to see if any pattern can be extracted.

References

1. Dean, T., Wellman, M.: Planning and Control. Morgan Kaufmann Publishers, Los Altos, CA (1990)
2. Khatib, O.: Real-time obstacle avoidance for manipulators and mobile robots. International Journal of Robotics Research **5** (1986) 90–99
3. Latombe, J.C.: Robot Motion Planning. Kluwer Academic Publishers, Boston, Massachusetts (1990)
4. Korf, R.: Planning as Search: A Quantitative Approach. Artificial Intelligence **33** (1987) 65–88
5. Genesereth, M., Nilson, N.: Logical Foundations of Artificial Intelligence. Morgan Kaufmann Publishers, Los Altos, California (1987)
6. Poole, D., Mackworth, A., Goebel, R.: Computational Intelligence: a Logical Approach. Oxford University Press, New York (1998)
7. Bratko, I.: Prolog Programming for Artificial Intelligence. Addison Wesley, Singapore (1990)
8. Rich, E.: Artificial Intelligence. McGraw-Hill (1983)
9. Sterling, L., Shapiro, E.: The Art of Prolog. second edn. The MIT Press, Cambridge, Massachusetts (1994)
10. Clocksin, W.F., Mellish, C.S.: Programming in Prolog. third edn. Springer-Verlag (1988)
11. K. S. Fu, R.C.G., Lee, C.S.G.: Robotics: Control, Detection, Vision and Intelligence. McGraw-Hill, New York (1988)
12. Umbaugh, S.E.: Computer Vision and Image Processing. Prentice-Hall, New York (1998)
13. Latombe, J.C., Lazanas, A., Shekhar, S.: Robot Motion Planning with Uncertainty in Control and Sensing. Artificial Intelligence **52** (1991) 1–47
14. Nilsson, N.: Teleo-reactive programs for agent control. Journal of Artificial Intelligence Research **1** (1994) 139–158
15. Batchelor, B.G.: Intelligent Image Processing in Prolog. Springer-Verlag, London (1991)
16. Delrieux, C., Katz, R.: Hybrid image recognition architecture. In: SPIE's 16th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Controls, Florida USA, SPIE-The International Society for Optical Engineering (2002)
17. Delrieux, C., Katz, R.: Integrating Symbolic and Numerical Image Processing. In: The 2002 International Conference on Imaging Science, Systems, and Technology (CISST'02), Las Vegas USA, Proceedings of the CISST'02 (2002)