Path planing of mobile robot

Business

During execution, the robot must react to unforeseen events (e. g. obstacles) in such a way so as to still reach the goal. Suppose that a robot M at a time i has a map Mi and an initial belief sate bi. The robot’s goal is to reach a position p while satisfying some temporal conditions: locg(R) = p; (g? n).

Thus the robot must be at p before the nth step. Although the goal of the robot distinctly physical, the robot can only really sense its belief state; not its physical location, and therefore we map the goal of reaching location p to reaching a belief state bg, corresponding to the belief that locg(R) = p.

With this formulation a plan q is nothing more than one or more trajectories form bi to bg if the plan is executed from a world state consistent with both bi and Mi. Completeness of a robot The robot is complete if and only if, for all possible problems (i. e. , initial belief states, maps, and goals), when there exists a trajectory to goal belief state, the system will achieve the goal belief state.

2. Path planning 1. Configuration space Path planning for manipulator robots and indeed, even for most mobile robots, is formally one in a representation called Configuration space.

Suppose that a robot arm has k degree of freedom. Every state or configuration can be described with k real values: q1, .

.. , qn. The values can be regarded as a point p in a k-dimensional configuration space C. If we define configuration space obstacle as O as the subspace of C, then the free space F= C-O in which the robot can move safely.

For mobile robotics opening on flat ground, we generally robot position with 3 variables (x, y,?). Nowadays, it is often assumed that a robot is simply a point. Thus we can reduce the configuration space looks essentially identical to 2D.

The first step of any path-planning is to transform this possibly continuous environmental model into a discrete map suitable for the chosen path-planning algorithm. Path planners differ as to how they effect this discrete decomposition.

We can identify 3 general strategies for decomposition: 1. Road map: identify a set of routes within the free space. 2. Cell decomposition: discriminate between free and occupied cells. 3. Potential field: impose a mathematical function over the space.

2. Road map approach

This approach is dependent upon the concepts of configuration space and a continuous path. A set of one-dimensional curves, each of which connect two nodes of different polygonal obstacles, lie in the free space and represent a roadmap R. That is, all line segments that connect a vertex of one obstacle to a vertex of another without entering the interior of any polygonal obstacles are drawn. This set of paths is called the roadmap. If a continuous path can be found in the free space of R, the initial and goal points are then connected to this path to arrive at the final solution, a free path.

If more than one continuous path can be found and the number of nodes in the graph is relatively small, Dijkstra’s shortest path algorithm is often used

to find the best path. There are various types of roadmaps, including the visibility graph, the Voronoi diagram, the freeway net, and the silhouette. Discussion of all the types is beyond the scope of this report; however, the first two approaches are discussed in the following sections. 1. Visibility Graph The graph for a polygonal configuration space consists of edges joining all pairs of vertices that can see each other (including initial and final positions as vertices as well).

The unobstructed straight line joining those corners is the shortest distance between them.

Now our task is simplified and thus just required to find shortest line of all the roads. There are two important caveats when implementing visibility graph search. First, the size of the representation and the number of edges and nodes increases with the number of obstacle polygon. These can make the program run slow and be inefficient. The second caveat is a much more serious potential flaw: the solution paths found by this method tend to take the robot as close as possible to obstacle on the way to the goal.

This implies visibility graph is optimal only in terms of length of travel but not safety.

2. Voronoi Graph Contrast to visibility graph, Voronoi graph tries to maximize the distance between robot and the obstacle. Considering the adjacent figure we plot the distance as a height coming out of the screen. The height increases as we move away from the obstacle. At points that are equidistant from two or more obstacle has sharp ridges. The Voronoi diagram consists of the edges formed by these sharp ridges.

When the configuration space obstacles are polygon, the Voronoi diagram consists of a line and parabolic segment.

The one of the major disadvantage of Voronoi graph when we use a limited-range localized sensors. Since this path planning algorithm maximizes the distance between robot and obstacles in the environment, any short-ranged sensor will fail to sense faraway obstacle. However, given a particular planned path via Voronoi graph, the robot can move on it using ultraviolet or IR sensors. 3. Cell decomposition approach The basic idea behind this method is that a path between the initial configuration and the goal configuration can be determined by subdividing the free space of the robot's configuration into smaller regions called cells.

After this decomposition, a conneactivity graph, as shown below, is constructed according to the adjacency relationships between the cells, where the nodes represent the cells in the free space, and the links between the nodes show that the corresponding cells are adjacent to each other. From this conneactivity graph, a continuous path, or channel, can be determined by simply following adjacent free cells from the initial point to the goal point. These steps are illustrated below using both an exact cell decomposition method and an approximate cell decomposition method.

The first step in this type of cell decomposition is to decompose the free space, which is bounded both externally and internally by polygons, into trapezoidal and triangular cells by simply drawing parallel line segments from each vertex of each interior polygon in the configuration space to the

exterior boundary. Then each cell is numbered and represented as a node in the conneactivity graph.

Nodes that are adjacent in the configuration space are linked in the conneactivity graph. A path in this graph corrresponds to a channel in free space, which is illustrated by the sequence of striped cells.

This channel is then translated into a free path by connecting the initial configuration to the goal configuration through the midpoints of the intersections of the adjacent cells in the channel. This method is exact cell decomposition. Having discussed two strategies, the last strategy (i.

e. potential field approach) is now discussed. 4. Potential field approach The potential field method involves modelling the robot as a particle moving under the influence of a potential field that is determined by the set of obstacles and the target destination.

This method is usually very efficient because at any moment the motion of the robot is determined by the potential field at its location.

Thus, the only computed information has direct relevance to the robot’s motion and no computational power is wasted. It is also a powerful method because it easily yields itself to extensions. For example, since potential fields are additive, adding a new obstacle is easy because the field for that obstacle can be simply added to the old one. The method’s only major drawback is the existence of local minima.

Because the potential field approach is a local rather than a global method (it only considers the immediate best course of action), the robot can get “

stuck” in a local minimum of the potential field function rather than heading towards the global minimum, which is the target destination.

This is frequently resolved by coupling the method with techniques to escape local minima, or by constructing potential field functions that contain no local minima. 3. Virtual Field Force The idea of having obstacles conceptually exerting forces onto a mobile robot has been suggested by Khatib [20].

Krogh [21] has enhanced this concept further by taking into consideration the robot’s velocity in the vicinity of obstacles. Thorpe [26] has applied the Potential Fields Method to off-line path planning. Krogh and Thorpe [22] suggested a combined method for global and local path planning, which uses Krogh’s Generalized Potential Field (GPF) approach.

Furthermore, none of the above methods has been implemented on a mobile robot that uses real sensory data. The closest project is that of Brooks [7, 8], who uses a Force Field method in an experimental robot equipped with a ring of 12 ultrasonic sensors.

Brooks’s implementation treats each ultrasonic range reading as a repulsive force vector. If the magnitude of the sum of the repulsive forces exceeds a certain threshold, the robot stops, turns into the direction of the resultant force vector, and moves on. 1.

The Basic VFF Method This section explains the combination of the Potential Field method with a Certainty Grid. This combination produces a powerful and robust control scheme for mobile robots, denoted as the Virtual Force

Field (VFF) method. As the robot moves around, range readings are taken and projected into the Certainty Grid, as explained above.

Simultaneously, the algorithm scans a small square window of the grid. The size of the window is 33×33 cells (i.

e. , 3. 30×3. 30m) and its location is such that the robot is always at its center. Each occupied cell inside the window applies a repulsive force to the robot, “ pushing” the robot away from the cell. The magnitude of this force is proportional to the cell contents, C(i, j), and is inversely proportional to the square of the distance between the cell and the robot: [pic]x +[pic]] Where, Fcr = Force constant (repelling) d(i, j) = Distance between cell (i, j) and the robot

C(i, j) = Certainty level of cell (i, j) x0, y0 = Robot’s present coordinates xi, yj = Coordinates of cell (i, j) The resultant repulsive force, Fr, is the vectorial sum of the individual forces from all cells: At any time during the motion, a constant-magnitude attracting force, Ft, pulls the robot toward the target.

Ft is generated by the target point T, whose coordinates are known to the robot. The target-attracting force Ft is given by where Fct = Force constant (attraction to the target) dt = Distance between the target and the robot xt, yt = Target coordinates

Notice that Ft is independent of the absolute distance to the target. The resultant R is given by R= Ft+Fr The direction of R, = R/| R| (in degrees), is used as the reference for the robot’s steering-rate command ?. Ks = Proportional constant for steering (in /sec) ? = Current direction of travel (in

degrees) 2. Low Pass Filter for Steering Control For smooth operation of the VFF method, the following condition between the grid resolution ‘ s’ and the sampling period T must be satisfied: In our case s = 0. 1 m and TVmax = 0.

1*0. 78 = 0. 078 m, and therefore the above condition is satisfied.

Since the distance dependent repulsive force vector Fr (see Eq. 2) is quantized to the grid resolution (10×10 cm), rather drastic changes in the resultant force vector R may occur as the robot moves from one cell to another (even with condition (6) satisfied).

This results in an overly vivacious steering control, as the robot tries to adjust its direction to the rapidly changing direction of R. To avoid this problem, a ddigital low-pass filter, approximated in the algorithm by t= 0. 4 sec, has been added at the steering-rate command. The resulting steering rate command is given by where i = Steering-rate command to the robot (after low-pass filtering) ? i-1 = Previous steering-rate command ?’i = Steering-rate command (before low-pass filtering) T = Sampling time (here: T = 0. 1 sec) t= Time constant of the low pass filter Ideally, when the robot encounters an obstacle, it would move smoothly alongside the obstacle until it can turn again toward the target. At higher speeds (e.

g. , V > 0. 5 m/sec), however, the robot introduces a considerable relative delay in responding to changes in steering commands caused by the combined effects of its inertia and the low-pass filter mentioned above.

Due to this delay, the robot might approach an obstacle very closely, even if the algorithm produces very sthrong repulsive forces. When the robot finally

turns around to face away from the obstacle, it will depart more than necessary, for the same reason.

The resulting path is highly oscillatory, as shown in Fig. 2a. One way to dampen this oscillatory motion is to increase the strength of the repulsive forces when the robot moves toward an obstacle, and reduce it when the robot moves alongside the obstacle.

The general methodology calls for variations in the magnitude of the sum of the repulsive forces Fr as a function of the relative directions of Fr and the velocity vector V. Mathematically, this is achieved by multiplying the sum of the repulsive forces by the directional cosine (cos?) of the two vectors, Fr and V, and using the product as follows: where F’r is the adjusted sum of the repulsive forces and w is a weighting factor. The directional cosine in Eq.

8 is computed by where Vx, Vy = x and y components of velocity vector V Frx, Fry = x and y components of the sum of the repulsive forces, Fr

The effect of this damping method is that the robot experiences the repulsive forces at their full magnitude, as it approaches the obstacle frontally (with -cos? = 1). As the robot turns toward a direction alongside the obstacle’s boundary, the repulsive forces are weakened by the factor 0. 75*cos? , and will be at their minimum value when the robot runs parallel to the boundary. Notice that setting w= 0 is undesirable, since the robot will eventually run into an obstacle as it approaches it at a very small angle. Careful examination of Eq. reveals the fact that the damped sum of repulsive forces, F’r, may become negative (thereby actually attracting the robot), as the robot moves away from the obstacle (and cos? ; 0).

We found the attraction-effect to improve damping and reduce oscillatory motion. 3. Speed Control The intuitive way to control the speed of a mobile robot in the VFF environment is to set it proportional to the magnitude of the sum of all forces, R = Fr + Ft. Thus, if the path was clear, the robot would be subjected only to the target force and would move toward the target, at its maximum speed.

Repulsive forces from obstacles, naturally opposed to the direction of Ft (with disregard to the damping effect discussed above), would reduce the magnitude of the resultant R, thereby effectively reducing the robot’s speed in the presence of obstacles. However, we have found that the overall performance can be substantially improved by setting the speed command proportional to cos? (see Eq.

9). This function is given by: Vmax for | Fr| 0 (i. e. in the absence of obstacles) Vmax*(1- | cos? |) for | Fr| > 0

With this function, the robot still runs at its maximum speed if no obstacles are present. However, in the presence of obstacles, speed is reduced only if the robot is heading toward the obstacle (or away from it), thus creating an additional damping effect. If, however, the robot moved alongside an obstacle boundary, its speed is almost not reduced at all and it moves at its maximum speed, thereby greatly reducing the overall travel-time.

Fig. 2b shows the joint effect of both damping measures on the resulting path. 4. A-Star (A*) Algorithm Like most of the algorithm, which focuses on cost optimization, A-star is o exception. This algorithm is useful to find the best path on any type of terrain.

The algorithm is discussed in details in the following sections. 1. The Arena Lets suppose that we have to go from green block (say A) to red block (say B), but a wall (shown in blue colour) blocks our way. The arena is divided into square grid. Simplifying the search area, as we have done here, is the first step in path finding.

This particular method reduces our search area to a simple two dimensional array. Each item in the array represents one of the squares on the grid, and its status is recorded as walkable or unwalkable.

The path is found by figuring out which squares we should take to get from A to B. Once the path is found, our person moves from the center of one square to the center of the next until the target is reached. These center points are called “ nodes”. It is possible to divide up path finding area into something other than squares.

They could be rectangles, hexagons, triangles, or any shape, really. And the nodes could be placed anywhere within the shapes – in the center or along the edges, or anywhere else. Because of simplicity, this system is used to understand. 2.

Starting the Search Once our search area is simplified into a manageable number of nodes, as done with the grid layout above, the next step is to conduct a search to find the shortest path.

We do this by starting at point A, checking the adjacent squares, and generally searching outward until we find our target. We begin the search by doing the following: 1. Begin at the starting point A and add it to an “ open

list” of squares to be considered. The open list is kind of like a shopping list. Right now there is just one item on the list, but we will have more later.

It contains squares that might fall along the path we want to take, but maybe not. Basically, this is a list of squares that need to be checked out. 2. Look at all the reachable or walkable squares adjacent to the starting point, ignoring squares with walls, water, or other illegal terrain. Add them to the open list, too.

For each of these squares, save point A as its “ paren’t square”. This paren’t square stuff is important when we want to trace our path. It will be explained more later. 3. Drop the starting square A from our open list, and add it to a “ closed list” of squares that we don’t need to look at again for now.

In this illustration, the dark green square in the center is the starting square.

It is outlined in light blue to indicate that the square has been added to the closed list. All of the adjacent squares are now on the open list of squares to be checked, and they are outlined in light green. Each has a gray pointer that points back to its paren’t, which is the starting square. Next, one of the adjacent squares is chosen on the open list and more or less repeats the earlier process, as described below. The one with the lowest F (explained in the following section of the report) cost is chosen. 3.

Path Scoring

The key to determining which squares to use when figuring out the path is the following equation: F = G + H Where, • G = the movement cost to move from the starting point A to a given square on the grid, following the path

generated to get there. • H = the estimated movement cost to move from that given square on the grid to the final destination, point B. This is often referred to as the heuristic, as the actual distance is still unknown until we find the path, because all sorts of things can be in the way (walls, water, etc.). Our path is generated by rrepeatedly going through our open list and choosing the square with the lowest F score.

This process will be described in more detail a bit further in the article. First let’s look more closely at how we calculate the equation. As described above, G is the movement cost to move from the starting point to the given square using the path generated to get there. In this example, we will assign a cost of 10 to each horizontal or vertical square moved, and a cost of 14 for a diagonal move. We use these numbers because the actual distance to move diagonally is the square root of 2 (don’t be scared), or roughly 1. 414 times the cost of moving horizontally or vertically.

We use 10 and 14 for simplicity’s sake.

The ratio is about right, and we avoid having to calculate square roots and we avoid decimals. This isn’t just because we are dumb and don’t like math. Using whole numbers like these is a lot faster for the computer, too. As we will soon find out, path finding can be very slow if we don’t use short cuts like these.

Since we are calculating the G cost along a specific path to a given square, the way to figure out the G cost of that square is to take the G cost of its paren’t, and then add 10 or 14 depending on whether it is diagonal or orthogonal (non-diagonal) from that paren’t square.

The need for this method will become apparen’t a little further on in this example, as we get more than one square away from the starting square. H can be estimated in a vvariety of ways. The method we use here is called the Manhattan method, where one calculates the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement, and ignoring any obstacles that may be in the way. We then multiply the total by 10, our cost for moving one square horizontally or vertically.

This is (probably) called the Manhattan method because it is like calculating the number of city blocks from one place to another, where we can’t cut across the block diagonally.

One might guess that the heuristic is merely a rough estimate of the remaining distance between the current square and the target “ as the crow flies. ” This isn’t the case. We are actually trying to estimate the remaining distance along the path (which is usually farther). The closer our estimate is to the actual remaining distance, the faster the algorithm will be.

If we overestimate this distance, however, it is not guaranteed to give us the shortest path.

In such cases, we have what is called an “ inadmissible heuristic. ” Technically, in this example, the Manhattan method is inadmissible because it slightly overestimates the remaining distance. But we will use it anyway because it is a lot easier to understand, and because it is only a slight overestimation. F is calculated by adding G and H. The results of the first step in our search can be seen in the illustration below.

The F, G, and H scores are written in each square.

As is indicated in the square to the immediate right of the starting square, F is printed in the top left, G is printed in the bottom left, and H is printed in the bottom right [Figure 3] 4. Calculations In the square with the letters in it, G = 10. This is because it is just one square from the starting square in a horizontal direction. The squares immediately above, below, and to the left of the starting square all have the same G score of 10. The diagonal squares have G scores of 14.

The H scores are calculated by estimating the Manhattan distance to the red target square, moving only horizontally and vertically and ignoring the wall that is in the way.

Using this method, the square to the immediate right of the start is 3 squares from the red square, for an H score of 30. The square just above this square is 4 squares away for an H score of 40. And so the other values are calculated as above. The F score for each square, again, is simply calculated by adding G and H together. 5.

Continuing the Search

To continue the search, the lowest F score square is chosen from all those that are on the open list. We then do the following with the selected square: 4) Drop it from the open list and add it to the closed list. 5) Check all of the adjacent squares. Ignoring those that are on the closed list or unwalkable (terrain with walls, water, or other illegal terrain); add squares to the open list if they are not on the open list already. Make the selected square the “

paren’t” of the new squares. 6) If an adjacent square is already on the open list, check to see if this path to that square is a better one.

In other words, check to see if the G score for that square is lower if we use the current square to get there. If not, don’t do anything. On the other hand, if the G cost of the new path is lower, change the paren’t of the adjacent square to the selected square (in the diagram above, change the direction of the pointer to point at the selected square). Finally, recalculate both the F and G scores of that square. If this seems confusing, we will see it illustrated below. 6.

Working Of our initial 9 squares, we have 8 left on the open list after the starting square was switched to the closed list.

Of these, the one with the lowest F cost is the one to the immediate right of the starting square, with an F score of 40. So we select this square as our next square. It is highlight in blue in the following illustration. First, we drop it from our open list and add it to our closed list (that’s why it’s now highlighted in blue). Then we check the adjacent squares.

Well, the ones to the immediate right of this square are wall squares, so we ignore those. The one to the immediate left is the starting square. That’s on the closed list, so we ignore that, too.

The other four squares are already on the open list, so we need to check if the paths to those squares are any better using this square to get there, using G scores as our point of reference. Let’s look at the square right above our selected square.

Its current G score is 14. If we instead went through the current square to get there, the G score would be equal to 20 (10, which is the G score to get to the current square, plus 10 more to go vertically to the one just above it). A G score of 20 is higher than 14, so this is not a better path.

It’s more direct to get to that square from the starting square by simply moving one square diagonally to get there, rather than moving horizontally one square, and then vertically one square. When we repeat this process for all 4 of the adjacent squares already on the open list, we find that none of the paths are improved by going through the current square, so we don’t change anything.

So now that we looked at all of the adjacent squares, we are done with this square, and ready to move to the next square.

So we go through the list of squares on our open list, which is now down to 7 squares, and we pick the one with the lowest F cost. Interestingly, in this case, there are two squares with a score of 54. So which do we choose? It doesn’t really matter. For the purposes of speed, it can be faster to choose the last one to get added to the open list.

This biases the search in favor of squares that get found later on in the search. But it doesn’t really matter. So let’s choose the one just below, and to the right of the starting square, as is shown in the following illustration. pic] [Figure 5] This time, when we check the adjacent squares we find that the one to the immediate right is a wall square, so we ignore that. The same goes for the one just above that.

We also ignore the square just below the wall, because we can’t get to that square directly from the current square without cutting across the corner of the nearby wall. We really need to go down first and then move over to that square, moving around the corner in the process. This rule on cutting corners is optional. Its use depends on how our nodes are placed.

That leaves five other squares.

The other two squares below the current square aren’t already on the open list, so we add them and the current square becomes their paren’t. Of the other three squares, two are already on the closed list (the starting square, and the one just above the current square, both highlighted in blue in the diagram), so we ignore them. And the last square, to the immediate left of the current square, is checked to see if the G score is any lower if you go through the current square to get there.

No dice. So we’re done and ready to check the next square on our open list. We repeat this process until we add the target square to the closed list, at which point it looks something like the illustration below.

[pic] [Figure 6] We note that the paren’t square for the square two squares below the starting square has changed from the previous illustration. Before it had a G score of 28 and pointed back to the square above it and to the right. Now it has a score of 20 and points to the square just above it.

This happened somewhere along the way on our search, where the G score was checked and it turned out to be lower using a new path – so the paren’t was switched and the G and F scores were recalculated. While this change

doesn’t seem too important in this example, there are plenty of possible situations where this constant checking will make all the difference in determining the best path to our target. We start at the red target square, and work backwards moving from one square to its paren’t, following the arrows.

This will eventually take you back to the starting square, and that’s our path.

It should look like the following illustration. Moving from the starting square A to the destination square B is simply a matter of moving from the center of each square (the node) to the center of the next square on the path, until you reach the target. [pic] [Figure 7] 7. Summary of the A* Method Okay, now that we have gone through the explanation, let’s lay out the step-by-step method all in one place: 1) Add the starting square (or node) to the open list.

2) Repeat the following: a) Look for the lowest F cost square on the open list.

We refer to this as the current square. b) Switch it to the closed list. c) For each of the 8 squares adjacent to this current square … • If it is not walkable or if it is on the closed list, ignore it. Otherwise do the following.

• If it isn’t on the open list, add it to the open list. Make the current square the paren’t of this square. Record the F, G, and H costs of the square. • If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. A lower G cost mearns that this is a better path.

If so, change the paren’t of the square to the current square, and recalculate the G and F scores of the square.

If you are keeping our open list sorted by F score, you may need to resort the list to account for the change. d) We stop when: • Add the target square to the closed list, in which case the path has been found (see note below), or • Fail to find the target square, and the open list is empty. In this case, there is no path. 3) Save the path. Working backwards from the target square, go from each square to its paren’t square until you reach the starting square.

That is our path.

Note: In earlier versions of this article, it was suggested that you can stop when the target square (or node) has been added to the open list, rather than the closed list. Doing this will be faster and it will almost always give you the shortest path, but not always. Situations where doing this could make a difference are when the movement cost to move from the second to the last node to the last (target) node can vary significantly — as in the case of a river crossing between two nodes, for example. 8. Variable Terrain Cost

In this report and my accompanying program, terrain is just one of two things – walkable or unwalkable. But what if we have terrain that is walkable, but at a higher movement cost? Swamps, hills, stairs in a dungeon, etc.

– these are all examples