Motion Planning Using Random Networks

This article gives a short overview of methods used for planning the movements of abstract objects with several degrees of freedom, followed by a complete discussion of a probabilistic roadmap-based approach. The algorithm described is well-suitable to many motion planning problems in (mostly) static environments. The article describes the algorithm in general terms and also gives a discussion of implementation details and possible optimizations.

Table of Contents

1. Introduction

No matter if we're dealing with probes sent to the distant surface of Mars to take pictures, or with the less peaceful Quake-bots programmed to take your virtual life in a computer-simulated arena, motion planning and pathfinding are crucial elements to these applications. It is easy to come up with a variety of other examples, some in present day and some in future technologies, where algorithms to automatically find a path, a roadmap or a sequence of movements an entity of some sort has to follow, play an important role.

2. Defining the Problem

For simplicity reasons I will from now on use the term '*robot*' for the object whose motion is to be planned rather than speaking about abstract entities, but depending on the application you have in mind you may freely substitute 'sprite', 'spaceship', 'car' and many more. They all fit in the same framework:

- They can perform several kinds of movement (e.g. move forward, rotate certain parts,...). The types of movement a robot can perform will be referred to as *degrees of freedom (dof)* in the following discussion.
- At every point in time, they have exactly one location and orientation that is described by giving a numeric value for each dof. Those values taken together form a so-called *configuration*. The set of all numerically possible combinations of dof values is called the *configuration space* C.
- While the configuration space may contain many more configurations that are numerically possible, only certain configurations are conceptually or physically legal. A robot can only move between obstacles, not through them for instance. Configurations that are allowed will be referred to as *free configurations* and the subset of C made up by them is the *free configuration space* C_{free}.
- Moving from one configuration to a different configuration is not instantaneous (we don't have teleporting robots, yet). Intermediate configurations have to be passed through that must all be within C_{free} to be feasible. Computing a sequence of free configurations (a *path*) that connects a start configuration with a goal configuration (e.g. moves the robot around obstacles from point A to point B) is non-trivial.

The last point already describes the problem discussed in this article:

Given a certain set C (derived from a robot's dof), a subset C_{free} of C, a start configuration q_{start} and a goal configuration q_{goal} (both elements of C_{free}), compute a path within C_{free} that connects q_{start} with q_{goal}.

3. Different Approaches

There are several approaches to the problem of motion planning most of which fall into one of the following categories (hybrid approaches also exist):

cell decomposition

The configuration space is split into disjoint subsets (*cells*) which are represented in a graph. Nodes correspond to cells and edges connect nodes if the respective cells are connected. Usually this approach is only appropriate in low dimensional C-spaces where "natural" cell boundaries exist (e.g. walls in 3D space) and when the robot is very simple.

potential fields

The idea behind this approach is simple. When an electron (having negative charge) is put into an electric field it will be attracted by a positive charge (the goal) and repelled by other negative charges (obstacles). Starting at a certain point it will move towards the goal, driven by the force exerted by the electric field. A robot's configuration is just a point in the configuration space C, just like the electron is just a point in our natural 3D space.

So if a mathematical potential field function can be constructed that adequately models q_{goal} and the obstacles just as the electric field models the distribution of charges, then the virtual forces exerted on a virtual electron positioned in the point q_{start} can be computed. This leads to a path from q_{start} to q_{goal} that avoids the obstacles.

This method has been successfully applied to many path planning problems. However, with increasing number of dof and complexity of the obstacles, finding an appropriate potential field function becomes increasingly difficult.

roadmaps

Roadmap-based planners are somewhat similar to those using cell decomposition. However, instead of using a graph with nodes that describe subsets of C-space, roadmap graphs use nodes that represent individual configurations. This makes them even easier to use than cell decomposition graphs. There are many different ways to construct roadmaps manually and automatically. The main drawback of roadmap planners is that because every node in the graph contains only little information they often require much more space than other methods to capture the whole C_{free} adequately.

A roadmap-based planner finds a path connecting 2 configurations by looking for a path within the roadmap. This path is a sequence of nodes from C_{free} used as waypoints. These waypoints are usually not a complete path for the robot because they are too far apart and intermediate configurations are required. This situation is comparable to driving directions like "take the first street to the left, then the second one right, then the first one left". These are just waypoints. A car can not teleport from one to the other, it has to drive along the streets that connect them. This corresponds to the edges of the graph. These edges however are usually reduced to the information q is connected with q'. The actual connection, i.e. the intermediate configurations that a robot has to go through are usually not stored explicitly. They are dynamically regenerated as the robot goes from one waypoint to the next. This function is performed by a so-called *local planner* which is a sub-planner within the roadmap planner.

4. The Probabilistic Roadmap Planner

4.1. Algorithm Overview

The planner described in this article is the roadmap-based planner described in [1]. The method that is used to construct the roadmap does not require any structural information about C-space. The only primitive required is the ability to determine whether a given configuration is free or not. This allows this planner to be applied to almost any kind of robot with arbitrary dof. In order to choose the nodes for the roadmap, this planner uses a probabilistic approach. It generates configurations at random, tests if they are free and if so integrates them into the roadmap. This step is repeated until a certain limit (# of nodes, # of edges, time,...) is reached. The finished roadmap is then used to find a path from q_{start} to q_{goal} by first connecting path from q_{start} and q_{goal} to nodes path from q'_{start} and q'_{goal} that are within the graph, followed by a search in the graph for a connection between q'_{start} and q'_{goal}. The complete path is then q_{start}, q'_{start},..., q'_{goal}, q_{goal}

Because many nodes have to be created at random, tested for collisions and integrated into the roadmap, this can not be done in real-time. Finding a path within a graph on the other hand is a simple and fast operation. For that reason the roadmap planner is divided in 2 phases, a preprocessing phase that constructs the roadmap for a given (static) environment and a query phase that uses the generated roadmap to answer requests for paths between configurations. It is possible to interweave these phases to a certain extent by generating only a small roadmap in the preprocessing phase and adding nodes dynamically as queries are performed. That way the roadmap will adapt to the queries.

4.2. Preprocessing Phase

The preprocessing phase is done in 2 steps, roadmap construction and roadmap refinement. The roadmap construction step is the core of the planner. The refinement phase is optional and tries to improve the roadmap by making connections that the construction step missed.

4.2.1. Definitions

For the following algorithm description, let `R= ( N,E )` be the roadmap graph. `N` is the set of nodes that correspond to configurations in C_{free}. `E` is the set of edges connecting the nodes from `N`. D: C x C -> $|R^{+}_{0}$ is a pseudo-metric (i.e. symmetrical and non-degenerate, the triangle inequation is not required) that defines a distance between any 2 configurations.

4.2.2. Roadmap Construction

The algorithm used to construct the roadmap is the following (pseudo-code):

```
1.    N:={};
2.    E:={};

      repeat
3.       q:=random configuration from C;
4.       if q is free (i.e. no collision) then
         begin
5.          add q to N;
6.          choose subset Nq of N with candidate neighbours for q

7.          for all q' in Nq sorted ascending by D(q',q) do
            begin
8.             if local planner can connect q with q'
9.             then add (q,q') to E
            end;
         end;
10.    until limit is reached;
```

The algorithm is simple and straightforward. Begininning with an empty roadmap (steps 1 and 2), it repeatedly adds nodes to the roadmap until a certain limit is reached (step 10). This limit has to be chosen based on the application. One possibility would be to impose a limit on the size of the roadmap. Another possibility would be to terminate the algorithm after a specified time has elapsed. Both methods don't take into account that depending on the given environment more time/nodes might be required for creating an adequate roadmap. With more knowledge about the structure of `C` and C_{free} more useful limits can be chosen. The choice of limit is not crucial as the algorithm is incremental and thus can be called again anytime should it turn out that not enough nodes have been created.

More important is the choice made for step 6. The algorithm attempts to connect the new configuration `q` to all configurations from N_q. N_q therefore has an influence on how large and how connected the roadmap will be. The more nodes are considered as candidates for a connection the more well-connected will the roadmap be. However, the preprocessing time is dominated by the time spent in the inner-most loop which is proportional to the size of N_q. Furthermore, more connections mean more edges in the graph which may slow down query processing later as a larger graph has to be searched. The choice of N_q will be discussed in more detail later together with other implementation details.

The most important part in the roadmap construction algorithm (and also the query phase described later) is played by the local planner used in step 8. This sub-planner within the roadmap planner is used to compute paths between 2 configurations within the roadmap. Unlike the roadmap planner, this local planner is simple and not very powerful. An example for a possible local planner is a straight line planner that tries to connect the configurations q and q' with a straight line within C-space. It is obvious that this planner will fail if an obstacle is present between q and q'. For that reason the local planner is not very useful on its own. Together with an algorithm to find a way within the roadmap graph, however, the local planner can be used to find very complex paths around arbitrary obstacles. The general idea is that the roadmap graph describes the global structure of C_{free} while the local planner is responsible for the details. Only the local planner actually computes the intermediate configurations of the final path. The roadmap is just used to define waypoints which the local planner is able to connect.

4.2.3. Roadmap Refinement

Depending on how many nodes were created in the roadmap construction step and on how complex the environment the robot is moving in is, the roadmap constructed may need to be improved before it is adequate. The biggest difficulty for this algorithm is finding ways through narrow passages. Because the planner picks a sample at random, the probability is small that the sample lies within a narrow passage and in addition to this, the narrower the passage is, the more likely is a collision of the random configuration for the robot with the walls of the narrow passage. As a consequence, the constructed roadmap may contain several components that are not connected within the graph, even though there is a connection (said narrow passage) between the parts of C_{free} the 2 graph components represent. Furthermore, even if everything is adequately connected, it may be desirable to add more nodes in regions of C_{free} that only contain few nodes after roadmap construction. This adds alternative routes which is important in situations where several robots move within the same environment so that some paths might be temporarily blocked by other robots. Likewise it may be desirable to remove redundant nodes from the graph. If there is a large empty region in C_{free} where a robot can move freely from one configuration to another in a straight line, it would be a waste of space to store many nodes for this region.

Only a few general suggestions will be given here regarding the improvement step. When a specific application is given, more effective strategies can possibly be found. One way to improve the roadmap is to add nodes in difficult regions, especially if this allows connecting previously unconnected components within the graph. Again a probabilistic method can be used. An effective way of adding nodes and connections to the roadmap that the roadmap construction step did not (and possibly could not) create is to use random-bounce walks. A configuration q is picked at random and from this starting point a path is generated by walking into a random direction till an obstacle is hit. On collision a different direction is chosen at random, and so on. This generates paths that have the potential of crossing difficult regions that the local planner failed at, like moving around a corner in a narrow tunnel. If it is possible to assign every node q in the graph a certain weight w(q) so that w(q) is high if q lies in a region where planning is hard for the local planner, it is usually a good idea to choose nodes with a high value for w(q) with a higher probability than those with lower weights. That way, random-bounce paths are not needlessly generated in regions where the local planner is sufficient. This is important because, unlike the paths generated by the local planner which can be regenerated any time by running the local planner again, the paths generated by random-bounce walks have to be store explicitly in the roadmap. Just inserting an edge is not sufficient. An effective way of storing a random-bounce path is to store the seed for the random number generator used. Once the new node q' that is the endpoint of the random-bounce walk, together with the path connecting it to q, has been stored in the roadmap, it should be attempted to connect q' with components previously unconnected to that of node q. There is the chance that the random-bounce walk has cleared a difficult obstacle and reached a point from where the local planner is able to connect to previously unreachable nodes. Connections should be attempted with nodes close to q' from components not connected to that of q.

If the weight function mentioned in the previous paragraph is available, there is another way of improving the roadmap. Again a node q from a difficult region is chosen at random but instead of doing a random-bounce walk, another configuration q' is just picked at random with a certain maximum distance from q. The local planner is used to connect q and q' as in the roadmap construction step. New roadmap nodes will thus be generated in difficult regions without the need to compute and store random-bounce paths. Note, however, that this method is not as powerful as the random-bounce method which is even able to find paths that the local planner is conceptually unable to find.

Another way to improve the roadmap is to choose to configurations q and q' from yet unconnected components and attempt to connect them using a very powerful planner if one is available to find a connection between q and q' (if one exists). That way a powerful planner, even if is too slow for regular use, can still be used productively by improving the roadmap for the fast probabilistic roadmap planner.

Sometimes it may even be advisable to add nodes manually. One configuration added in the middle of a narrow tunnel with the proper orientation of the robot for crossing the tunnel may be enough to allow the planner to make a connection through that tunnel. If possible, manual additions of nodes should already occur before the initial roadmap construction step as this may decrease the number of failed configurations tried and thus increase connectivity and speed of the roadmap construction.

4.3. Query Phase

In the query phase, the roadmap is used to answer requests for connections between given start and goal configurations. Let q_{start} be a given start configuration and q_{goal} be a given goal configuration. If q_{start} and/or q_{goal} are not among the configurations explicitly stored in the roadmap, the first step is to connect them to nodes q'_{start} and q'_{goal} within the roadmap. In order to do this for node q_{start}, nodes from the roadmap are chosen in ascending order of distance from q_{start}. The local planner is used to find a connection to each of those nodes. If a q'_{start} is found that can be connected to q_{start} by the local planner, the same is attempted for q_{goal}. Of course, q'_{start} and q'_{goal} must lie within the same connected component of the roadmap. If it is not possible to find appropriate q'_{start} and q'_{goal} configurations, random-bounce walks may be attempted starting at q_{start} and q_{goal} in order to reach configurations from which the local planner can connect to suitable q'_{start} and q'_{goal} respectively. If this also fails the whole query fails.

If appropriate configurations q'_{start} and q'_{goal} are found, the next thing is to connect q'_{start} and q'_{goal} within the graph. This yields a sequence of waypoints. Finally the requested path can be constructed by using the local planner to regenerate the path from q_{start} to q'_{start} (possibly preceded by a random-bounce path) and to regenerate the paths between the waypoints. Finally the path from q'_{goal} to q_{goal} is generated by reversing the path from q_{goal} to q'_{goal} that is regenerated using the local planner (if a random-bounce walk was executed, this path is also reversed and concatenated with the rest).

If the roadmap created in the construction phase is adequate, queries will be answered quickly and reliably. If this is not the case, queries will fail frequently and it is necessary to improve the roadmap. It is not necessary to regenerate the whole roadmap. As the graph is an incremental structure, new nodes (preferably in difficult regions) can just be added to the old roadmap.

4.4. Solving Problems with Narrow Spaces

The algorithm described above behaves very poorly if C_{free} contains many narrow spaces, not only because the probability of picking a configuration within a narrow space is low but also because the possible orientations of the robot within the narrow space may be limited. Because the orientation is also chosen at random it is very likely that a configuration picked in a narrow tunnel, for instance, is in collision with the walls. As a consequence, the roadmap constructed in the preprocessing phase will contain considerably fewer configurations in narrow spaces than in open spaces, even though narrow spaces would actually require more samples to be adequately covered. In the following I will present 2 ways to improve this situation. The first method retracts collision configurations onto the medial axis and is described in [2]. The second method is my own idea which I have unfortunately not yet had the time to implement and test. Both methods are intended for the use with rigid body robots in an environment with polygonal (2D) or polyhedral (3D) obstacles and are probably not applicable to other situations.

4.4.1. Retracting Configurations onto the Medial Axis

The planner described in [2] is based on the medial axis or general Voronoi diagram. The medial axis is the set of all points that have the same distance to at least 2 obstacle walls. It is obvious that these points are at a maximum distance to the blocked part of C-space. A path on the medial axis that goes through a tunnel, for instance, would have the robot move exactly in the middle of the tunnel. Even though the medial axis has a lower dimension than C-space itself (e.g. for a 2D C-space the medial axis is just a network of 1 dimensional lines) it is still a complete representation with regards to roadmap planning in the sense that for any roadmap constructed using points outside the medial axis there exists a roadmap that just contains points from the medial axis but still allows a path to be found for any (q_{start},q_{goal}) pair that the original roadmap could find a path for. It is assumed that the straight line planner is used as local planner.

Unfortunately the medial axis is not easy to compute. However, it turns out that a given configuration can be retracted onto the medial axis without computing it explicitly. The only geometric primitive required is a method to find the obstacle point that is closest to the configuration q to be retracted. Note that although I will still use the term configuration, this is actually just the location component of the configuration and of C-space. Only this component will be adjusted. Likewise the medial axis referred to here is not the medial axis of C-space. It's the medial axis of C-space projected onto the location component.

If a random configuration `q` is picked during the roadmap construction phase, it is usually not on the medial axis. This means that it has exactly one well-defined closest point `p` on an obstacle. This point is computed. To retract `q` onto the medial axis, we move `q` away from `p` on a straight line to a location `q'` with the property that it is as far away from `p` as from another point `p'` on a different obstacle. `q'` is now on the medial axis. This works for any configuration `q` in C_{free}. If `q` is a collision configuration, i.e. it is inside an obstacle it must first be moved into C_{free}. This is easily achieved by moving it towards instead of away from `p` until it has passed `p` by a little delta.

Using the above method it is possible to use even random configurations that are not in C_{free} as a basis for finding roadmap nodes. If it is desired that all paths have maximum distance to obstacles, this method should be applied to all nodes prior to entering them in the roadmap. It can be proven that this method significantly improves the number of configurations in the roadmap that lie in narrow areas of C_{free}. If only this effect is desired it is sufficient to only retract non-free configurations during roadmap construction and to use free configurations as is.

4.4.2. Growing The Robot Into Position

A different method for improving the number of configurations in narrow spaces is what could be called blow-up method. The only geometric primitive being used is the computation of collision points between the robot and obstacles. This information is usually available as a side-effect from the collision test that is always necessary for roadmap construction. The general idea is that roadmap construction is done as described in the basic algorithm but instead of throwing away non-free configurations in step 4, it is first attempted to find a near-by configuration that is free. In order to do this the robot is reduced to a single point (the reference point of the robot also used to describe the robot's position). This point is less likely to be within an obstacle (often it is even possible to avoid generating random configurations where the robot's reference point is inside an obstacle altogether). Now the robot's size is increased using a binary search-like method. First the robot is blown up to half its size. If it still collides, a quarter of its size is attempted. If the half-sized robot does not collide 3/4 are attempted, and so on until the maximum size is found that is collision free. As a next step the robot is enlarged just a little. This way it is likely to only collide in very few places. Using those points as reference it is easy to rotate and/or move the robot away from these collision points. If this is not possible without generating a new collision the attempt is aborted and the configuration is discared. If the attempt was successful the moved robot is enlarged further towards its natural size. If more collisions are encountered the above process is repeated until either the robot has reached its real size or the configuration is discarded.

The idea behind this method is to have the robot move into a collision-free position while being blown up just as a ballon would do if inflated close to a wall. It can improve the number of nodes in narrow passages because it allows more configurations to be used that would otherwise be discarded. Because it also allows modifying the rotation component of a configuration to make it collision-free it is especially suitable for situations where collisions are frequently caused by a misoriented robot, as would be the case when moving a long stick-like robot through a tunnel. In the latter case most configurations generated at random within the tunnel would only collide because the robot is not aligned with the tunnel's direction. This is something the blow-up algorithm should be able to fix easily. As mentioned above I have not tested this method for applicability.

4.5. Implementation Issues

In this section I will discuss several implementation issues with the probabilistic roadmap planner.

4.5.1. Saving Space in the Roadmap

As the dimension of C-space corresponds to the robot's dof, the size of the necessary roadmap graph grows, too. This is because every entry in the roadmap graph only corresponds to exactly one possible combination of dof values. This means that as more dimensions are added, more configurations must be sampled to have a representative set of samples. However, very often several nodes can be joined together. Take a car in an empty parking lot for example. In almost every location of the parking lot the car can point into any direction without colliding. It would be a waste of space to save 100 nodes in the roadmap with (almost) the same location, only differing in orientation. It is better to modify the algorithm to check several orientations for every random location picked during roadmap construction and to store information like "angles 0 to 270 degrees free" in the node. This also improves the paths found during query phase as the car may move more smoothly, not being forced to go exactly through fixed orientations dictated by the waypoints. This approach can be taken a step further by not only storing several free configurations in one node but by testing several possible connections when adding an edge to the roadmap graph. Such an extended edge would contain information like "location 1 with orientations between 0 to 90 degrees can be connected to location 2 with orientations 90-180 degrees". This way a single edge can store many possible movements from location 1 to location 2 that differ in orientation. Again this allows for smoother paths to be chosen.

4.5.2. Choosing the Local Planner

When choosing the local planner there are basically 2 main questions to be answered. The first is the question whether the planner should be deterministic or use a probabilistic approach (like the random-bounce walk). The second question is how powerful the planner should be, considering that more powerful planners take more time to compute. Experimental results have shown that fast and deterministic planners are best. Speed is more important than power because the running time of the local planner is the main component in the running time of the preprocessing algorithm as well as the query algorithm. Handling the complexities of the environment is better left to the roadmap. The planner should be deterministic so that no additional information is necessary to reconstruct paths between nodes in the roadmap.

4.5.3. Choosing the Candidate Neighbours

In step 6 of the preprocessing phase a set of candidate neighbours is chosen for which connection with the random configuration will be attempted. Because the local planner is likely to fail for configurations far away from each other the set of candiate neighbours N_q should be chosen near the location of q which is also the reason why the nodes in N_q are used in the order of ascending distance from q in step 7. If the length and shape of the paths found during query phase is not important an optimization that can be done is to discard, once a succesfull connection between q and q' has been established, all those nodes from N_q within the same connected component as q'. Otherwise there are redundant connections to this component.

4.5.4. Choosing the Distance Function

The distance function used to find configurations close to another configuration should reflect the local planners ability to connect 2 given nodes. The distance of 2 nodes that the planner is likely to fail for should be high. If the local planner being used is the straight line planner, the Euclidean distance serves this purpose well.

4.5.5. Data Structures that Support the Planner

For the performance of the planner as described in this article, the proper choice of data structures is crucial. When the candidate neighbours for q are chosen during roadmap construction and when q'_{start} and q'_{goal} are to be found in the query phase it is important to be able to quickly find those nodes in the roadmap that are close to q, q_{start} and and q_{goal} respectively. Searching the whole graph for those nodes is extremely inefficient. An index structure specialized on next neighbour queries or window queries like an R* tree is advisable.

4.5.6. Improving Paths

The paths returned in the query phase may be twisted and overly long, especially if no redundant nodes were added in the construction phase by not attempting connections to the same component twice. For that reason smoothing and short-cutting the returned path is often required. One possible way is to try and skip waypoints. This is done by trying if the local planner can find a path between waypoint i and waypoint i+2, thus skipping waypoint i+1. A way to smooth the path for a rigid body robot would be to compute some kind of spline using the waypoints as orientation, instead of connecting them with the local planner. Care must be taken to ensure that this spline path does not collide with an obstacle.

4.5.7. Splitting Responsibilities

When implementing this algorithm it is best to keep the local planner, the global (roadmap) planner and the structure used to store the graph separate so that they can be easily exchanged depending on application. In an object-oriented language this naturally done by having separate classes for all three that only communicate via interfaces. Although generating a random configuration appears in the preprocessing phase of the global planner it should be implemented as part of the local planner because the local planner always has to know the robot whereas this knowledge would have to be added to the global planner in order to find a random configuration, which would destroy the application-independence of the global planner. If a separate class is used for collision detection it is also possible to give this class responsibility for generating the random free configurations. This would also be a natural choice as this class also has knowledge about the robot so that adding this function to its interface would decrease its flexibility. Of course the above is only one possible realization. There are many more but for a flexible design it is important that the different parts of the algorithm do not know more about the application than is strictly required.

5. Literature

This article is based on

[1] L. Kavraki, J.-C. Latombe. Probabilistic Roadmaps for Robot Path Planning. (chapter 3 in the book "Gupta et al. Practical Motion Planning in Robotics. ISBN 0-471-98163-X)

[2] S. Wilmarth, N. Amato, P. Stiller. Motion Planning for a Rigid Body Using Random Networks on the Medial Axis of the Free Space. Proceedings of the 15th Annual ACM Symposium on Computational Geometry, 1999, pp. 173-180