Path Planning Algorithms for Blind People

Natesh M, Murali S
Department of Computer Science & Engineering, Maharaja Institute of Technology, Belawadi, Srirangapatna Tq, Mandya, Karnataka, India

Abstract—This paper investigates path planning strategies for blind people in partially unknown environments. The aim of this work is to reduce collision risk and time of path following in cases when robot repeatedly traverses between predefined target points (e.g. transportation or surveillance tasks). We consider the case of constrained environments where the robot is represented as a point. For that, we used an approach based on models of evolution; the Probabilistic Roadmap Planner (PRM) algorithms which are an interesting alternative to conventional methods of path planning. A population of paths is obtained firstly using a random distribution strategy. The performance of the Probabilistic Roadmap Planner Algorithm based approach is tested on environments with increasing complexity. Experimental data show that approach is able to reduce collision risk, travel time and distance. The robot is also able to learn and adapt quickly in a changing environment.

Keywords—A*, RRT, PRM, D*, Anytime

I. INTRODUCTION

For vision disabled people (or visually impaired persons), it is always a problem to manage their life like a normal person. Even some very simple jobs like going to job/school, shopping are sometimes causing big problems for them. Before managing their life, they also have problems with obtaining the necessities of a human body. They can overcome these difficulties by themselves or with the help of a guide. There are several studies for visually impaired people to make their life easier which free them from boundaries. Due to the development of modern technology, many different types of navigational aids are now available to assist blind persons. They are frequently known as Electronic Travel Aid (ETA) which have been used widely to help blind people while they are walking from one point to another [1]. ETA systems are efficient navigation systems; however they require hard training due to the complicated sensors which used in such systems. Recently, robotic technology has been involved in guiding and navigating impaired people. The use of robotics is a promising alternative to guide dogs. A well designed navigation robotic system could enhance reliability and reduce high cost required for such complicated systems (i.e. ETA). A project named Eye Blind has started in 2012, which mainly aims to build a robotic system which should be able to guide blind people to reach their destinations with the least cost and the shortest route possible. In addition to track and localize the blind person’s position, in order to help him/her in emergency situations. Usually, navigation systems consist of three main parts to help blind people to travel with a greater degree of comfort and independency. First, sensing the instant environment for obstacles and hazards, second, providing information about location and orientation, and third, providing the optimal route towards the desired locations.

II. RELATED WORK

Most of the developed projects so far achieved for assisting blind people in navigation tasks combine research progress with technological innovations. Most of experiments have been either directed to indoor or outdoor environments, and considering that blind people have different knowledge of the environment where they act in either well-known, partially known or unknown surroundings. There are many methods and devices used to assist and guide blind people. Several research works are being performed by several institutions throughout the world. This section reviews various navigational aids for blind individuals and categorizes them based on the technology used to guide blind people. Based on our knowledge, navigation systems can be divided into 5 categories namely: GPS based systems, Radio Frequency Identification (RFID) based systems, camera based systems, sensor based systems, and inertial navigation based systems. First, GPS based systems are considered. A GPS receiver is used in this kind of systems to navigate blind persons, according to the longitude and latitude values received from GPS satellites. An electronic device was designed in [2], to navigate blind people with no need for any human assistant. The proposed system consists of the following components: GPS receiver, PIC microcontroller, and voice recorder. This work aimed to fabricate a device which is cheaper than using other complicated and expensive devices. In [3], a mobile assistant system to locate and orient blind passengers in a Metrobus environment was introduced. A low cost system was developed based on using a mobile phone, GPS receiver, and a compass device to provide a good functionality. Interaction among these devices is achieved using Bluetooth communications. Furthermore, an audible interface was implemented to inform the blind person about his/her location within the Metrobus environment. Additionally, a new system was proposed in [4] to navigate blind people in indoor and outdoor environments, using GPS receiver and a wireless connection. Second, RFID based systems are taken place in this section. RFID based systems consists of RFID tags and readers. Usually, RFID readers are required to be distributed over...
the navigational area, while the RFID tags might be attached to each robot device, or vice versa. The RFID reader detects the serial number stored in the RFID tag attached to each robot, and then transmits the location information to a base station in order to compute the robot’s location. A navigational robotic system based on a combination of RFID and GPS systems was proposed in [5] which is called Smart-Robot (SR). While the implemented robotic system in [6] based on the RFID technology; RFID passive tags were deployed over the area of interest, whereas RFID reader was attached to the blind person. This work is similar to Blind Interactive Guide System (BIGS) presented in [7]. BIGS is an indoor navigation system, and includes scattering the RFID tags and readers in the same way as in [6]. In [8], an RFID robotic system was introduced, which consists of an RFID reader and a laser range finder. RFID technology was used to determine the location of the blind person, whereas the laser range finder was used to find out and obstacles facing the blind person. On the other hand, Smart Vision is a development of an electronic white cane that helps blind people moving around inside the navigational area and then processing them. The work presented in [18] includes a visual navigational system consists of a computerized cane which can steer blinds around obstacles, whereas the system proposed in [17] includes a new approach to localize visually impaired people indoors, using cane-mounted sensors, and a foot-mounted pedometer. Fourth, camera based systems. A camera device is used to guide and navigate blind people by capturing images around the navigational area and then processing them. The work presented in [18] includes a visual navigational system using ultrasonic sensor and USB camera. Also [19] includes a prototype assistive-guide robot (eyeDog) which provides the visually impaired person with autonomous vision-based navigation and laser-based obstacle avoidance function using USB webcam. A new robotic system based on using camera and laser range finders was proposed in [20], where sensors and a laptop were placed on a trolley walker. Fifth, the Inertial Navigation systems are reviewed in this section, a navigational robotic system is proposed in [2], and a device called ROVI was produced. ROVI is a device that can be used to guide visually impaired people based on using ultrasonic sensors, and digital encoder. The ultrasonic sensor was deployed to estimate the distance between the blind person and obstacles, while the role of the digital encoder was to estimate the distance travelled by ROVI. The work presented in [18] involves a visual odometry system with an assisted inertial navigation filter to produce a precise and robust navigation system which does not rely on particular infrastructure.

III. PATH PLANNING

Planning consists of finding a sequence of actions that transforms some initial state into some desired goal state. In path planning, the states are agent locations and transitions between states represent actions the agent can take, each of which has an associated cost. A path is optimal if the sum of its transition costs (edge costs) is minimal across all possible paths leading from the initial position (start state) to the goal position (goal state). A planning algorithm is complete if it will always find a path in finite time when one exists, and will let us know in finite time if no path exists. Similarly, a planning algorithm is optimal if it will always find an optimal path. The general problem of this paper aims at solving is to find reliable paths for repeated traversal between previously determined target points so that following them minimises collision risk, speeds up the mission and increases predictability of the robot’s behaviour.

According to LaVall [21], the path planning algorithms are mainly divided into two groups: 1) Compound Path planning Algorithms; and 2) Sampling-based path planning algorithms. In Compound path planning algorithms are complete and they have a complicated implementation. An algorithm is considered complete if for any input it correctly reports whether there is a solution in a finite amount of time. If a solution exists, it must return one in finite time [21]. The second group is not complete but it has a simple implementation. Many sampling-based approaches are based on random sampling, which is dense with “probability complete”. This leads to algorithms that with enough points, the probability that it finds an existing solution converges to one. The most relevant information, however, is the rate of convergence, which is usually very difficult to establish [21].The second group is divided into two other groups, namely multiple query and single query. Multiple query methods like (PRM) Probabilistic Road Map method require pre-processing while single query methods like A* (Hart et al.,[23]; Nilsson [24]). Both algorithms return an optimal path (Gelpperin [25]), and can
be considered as special forms of dynamic programming (Bellman [26]). A* operates essentially the same as Dijkstra’s algorithm except that it guides its search towards the most promising states, potentially saving a significant amount of computation.

IV. PATH PLANNING ALGORITHMS

First, we describe Dijkstra’s algorithm that finds shortest paths in a graph from a single source to all nodes in the graph. An extension to Dijkstra’s algorithm is A*, that speeds up the calculation of a shortest path in case one is only interested in a path between a single source and a single goal. Later we discuss about D*, Rapidly-exploring Random Graph (RRT) and Probabilistic Roadmap Planner (PRM) algorithms.

A. Dijkstra’s

If one is given a graph $G = (V,E)$ consisting of a set of vertices $V$ and a set of edges $E \subseteq V \times V$, and for each edge $(u,v) \in E$ an associated nonnegative cost $c(u,v)$, Dijkstra’s algorithm is able to find shortest paths from a single source vertex $u_{\text{start}} \in V$ to all vertices in $V$. This algorithm is used in many path planning algorithms and throughout this thesis to find shortest paths in graphs (or grids, which in fact are graphs as well) that for instance represent roadmaps of valid motion paths through an environment. The cost associated with each edge is often the distance of the motion which it represents, but other criteria such as clearance, safety, etc., can be incorporated into the cost function as well. Dijkstra’s algorithm works by maintaining for each vertex $u$ the shortest path distance $g(u)$ from the start vertex. Further, a backpointer $bp(u)$ is maintained for each vertex indicating from which neighboring vertex the shortest path from the start comes. Hence, a shortest path to some vertex can be read out by following the backpointers back to the start vertex. Initially, of all vertices the shortest path distance is set to infinity, except for the start vertex whose distance is set zero. From the start vertex, the shortest path distances are propagated through the graph until all vertices have received their actual shortest path distance. To this end, the start vertex is inserted into a priority queue, with a key equal to its shortest path distance (zero in this case). In each iteration of the algorithm, the minimal element $u$ is popped from the priority queue, and of all neighbors of $u$ it is checked whether their shortest path distance can be decreased by changing their backpointer to $u$. If the distance of some vertex is decreased, it is inserted (or updated) in the priority queue, as it may enable other vertices to decrease their distances. When the priority queue becomes empty, all vertices are guaranteed to have their actual shortest path distance and correct backpointer associated with it. The algorithm is shown below (Algorithm 2.1)

**Algorithm 2.1: DIJKSTRA**

1: for all $u \in V$ do
2: $g(u) \leftarrow \infty$
3: $g(u_{\text{start}}) \leftarrow 0$
4: Insert $u_{\text{start}}$ into $Q$
5: repeat
6: $u \leftarrow$ element from $Q$ with minimal $g(u)$
7: Remove $u$ from $Q$
8: for all neighbors $v$ of $u$ do
9: if $g(u)+c(u,v) < g(v)$ then
10: $g(v) \leftarrow g(u)+c(u,v)$
11: $bp(v) \leftarrow u$
12: Insert or update $v$ in $Q$
13: until $Q = \emptyset$

B. A*

Despite the speed of Dijkstra’s algorithm, there are cases in which it is desirable to optimize the performance of finding shortest paths (for instance when the roadmap is very large, or the application is very time-critical). Dijkstra’s algorithm finds shortest paths to all vertices in the graph, but often one is only interested in a shortest path to a specific goal vertex $u_{\text{goal}}$. In this case the A* method, which is based on Dijkstra’s algorithm, is favorable. A* focuses the search in the graph towards the goal, whereas in Dijkstra’s algorithm the shortest paths distances are propagated breadthwise. Still, A* is guaranteed to find the optimal path from the start vertex to the goal vertex. There are two differences between Dijkstra’s and A*. The first concerns the key with which the vertices are sorted in the priority queue. In Dijkstra’s this key equals the current shortest path distance to the start, which is $g(u)$ for a vertex $u$. In the A* method a heuristic value $h(u)$ is added to $g(u)$ when computing $u$’s key. This heuristic value must be a lower bound estimate of the distance from vertex $u$ to the goal vertex, so that the key of a vertex $u$ is a (lower-bound) indication of the length of the shortest path between start and goal to which vertex $u$ contributes. The second difference is that the algorithm stops when the goal vertex is popped from the queue, rather than when the priority queue $Q$ has become empty (see Algorithm 2.2).

**Algorithm 2.2: A**

1: for all $u \in V$ do
2: $g(u) \leftarrow \infty$
3: $h(u) \leftarrow$ lower-bound estimate of the distance between $u$ and $u_{\text{goal}}$
4: $g(u_{\text{start}}) \leftarrow 0$
5: Insert ustart into Q
6: repeat
7: u ← element from Q with minimal \((g(u) + h(u))\)
8: Remove u from Q
9: for all neighbors v of u do
10: if \(g(u) + c(u, v) < g(v)\) then
11: \(g(v) ← g(u) + c(u, v)\)
12: \(bp(v) ← u\)
13: Insert or update v in Q
14: until \(u = u_{goal}\) or \(Q = \emptyset\)

C. Anytime Algorithms

When an algorithm name includes the term anytime, it implies that the algorithm quickly finds a sub-optimal solution and then works to refine that solution while time permits [Hansen and Zhou, [29]]. Given enough time for refinement, anytime algorithms eventually converge to an optimal solution. Anytime algorithms are useful for real-time applications, where performing any valid yet sub-optimal action is considered better than performing an invalid action or no action at all. For instance, when driving a car down a highway, most people are willing to settle for a solution that keeps the car on the road and free from collisions—even if the solution is slightly longer than optimal. There are a couple of general frameworks for anytime algorithms we will discuss in the context of A*.

**Algorithm 2.3: Iterative Anytime A**

Require: G, E, vstart, vgoal, \(\varphi, \Delta\varphi\)
1: while time remains do
2: initialize(open-list)
3: for all \(v_i \in v_{goal}\) do
4: insert\((v_i, \varphi h_{start,i}, open-list)\)
5: while the open-list is not empty and time remains do
6: \(v_j = \text{get-best(open-list)}\)
7: if \(v_j \in v_{start}\) then
8: save the path from \(v_j\) to \(v_{goal}\)
9: break
10: for all \(v_i\) such that \(e_{i,j} \in E\) do
11: if \(v_i\) is unexpanded for this value of \(\varphi\) or \(d_{i,goal} > d_{i,j} + d_{j,goal}\) then
12: \(d_{i,goal} = d_{i,j} + d_{j,goal}\)
13: \(c_i' = \phi h_{start,i} + d_{i,goal}\)
14: update\((v_i, c_i', open-list)\)
15: set back-pointer from \(v_i\) to \(v_j\)
16: if \(v_j \in v_{start}\) then
17: \(\varphi = \varphi - \Delta\varphi\)
18: if \(\varphi < 1\) then
19: return SUCCESS
20: else
21: break
22: if a path has been saved then
23: return SUCCESS
24: return FAILURE

The first idea is to quickly perform a search that will find a suboptimal solution in the allotted amount of time, and then continue to perform increasingly more optimal searches as time permits. This can be accomplished with A* by weighting the admissible heuristic \(h_{i,j}\) by a factor \(\varphi \geq 1\) to get \(\phi h_{i,j}\). This causes the heuristic to over-estimate the cost of traveling between two nodes by at most \(\varphi\).

D. Lifelong A*  

Lifelong A* (Likhachev and Koenig [30]) has been developed specifically for applications in which multiple searches need to be performed through a changing representation, but the the start and goal locations are guaranteed to remain the same. If the number of changes is small compared to the size of the graph, then it is more efficient to repair the existing search-tree than to perform an entirely new search from scratch. Given the type of search-tree that we have been considering up to this point, a Lifelong A* algorithm could be implemented by reinserting into the open-list all nodes with modified edge cost or connectivity. Likhachev and Koenig, [10] take a slightly different approach because they do not explicitly store a pointer-based search-tree. Instead, Lifelong A* implicitly models the search-tree structure by storing cost-to-go values \(c_{i,goal}\) at each node. When the start node is reached, a path is extracted using gradient descent over cost-to-go values12, and ties are broken arbitrarily.
Let node vi be a neighbor of vi. The quantity rhsi is defined as follows:

\[ \text{rhsi} = \min_j (c_{i,j} + c_j, \text{goal}) \]

and represents the minimum cost of traveling to the goal from vi through one of its neighbors. For the case vi = vgoal the value rhsgoal = 0. As changes are made to the representation, Lifelong A* updates rhsi if an outgoing edge of vi has been modified. Next, if rhsi ≠ c_{i,goal} then the cost of moving from vi to the goal is no longer accurate, so vi is inserted back into the open-list. When rhsi ≠ c_{i,goal} the node vi is said to be inconsistent. The terms under-consistent and over-consistent are also used to describe the cases when rhsi > c_{i,goal} and rhsi < c_{i,goal}, respectively. The open-list priority heap is now sorted according to a lexicographic ordering of two values \([k1,k2]\) where \(k1 = h_{\text{start}},i + \min(c_{i,\text{goal}}, \text{rhsi})\) and \(k2 = \min(c_{i,\text{goal}}, \text{rhsi})\). Pseudo-code for Lifelong A* is given in Algorithm 10. rhsi values and c_{i,goal} are initialized to \(\infty\). The function top() returns the vertex on the top of the open-list without removing it.

**Algorithm 2.4: Lifelong A***

**Require:** G, E, vstart, vgoal

1: procedure CalculateKey(vi)
2: return \([h_{\text{start}},i + \min(c_{i,\text{goal}}, \text{rhsi}),\min(c_{i,\text{goal}}, \text{rhsi})]\)
3: procedure UpdateVertex(vi)
4: if vi \(\not\in\) vstart then
5: rhsi = \(\min_j c_{i,j} + c_j, \text{goal}\)
6: if vi \(\in\) open-list then
7: remove vi from the open-list
8: if (c_{i,goal} \(\neq\) rhsi) then
9: update(vi,CalculateKey(vi),open-list)
10: procedure PerformSearch()
11: while CalculateKey(top(open-list)) < CalculateKey(vstart) or rhsstart \(\neq\) cstart,goal do
12: vj = get-best(open-list)
13: if (c_{j,goal} > rhsj) then
14: c_{j,goal} = rhsj
15: for all vi such that e_{i,j} \(\in\) E do
16: UpdateVertex(vi)
17: else
18: c_{j,goal} = \(\infty\)
19: UpdateVertex(vj)
20: for all vi such that e_{i,j} \(\in\) E do
21: UpdateVertex(vi)
22: procedure main
23: for all vi \(\in\) vgoal
24: update(vi,CalculateKey(vi),open-list)
25: rhsstart = 0
26: forever
27: PerformSearch()
28: Wait for changes in edge costs
29: for all edges e_{i,j} with changed costs do
30: update UpdateVertex(vi)

When a node vi is under-consistent, its current distance to goal value is too low and the implicit search-tree structure may need be modified so that vi transitions to a different neighbor (line 19). Also, nodes that used to transition to vi may be able to find a better path through a different neighbor. This is accomplished by re-initializing vi (line 19) and then inserting all nodes that might transition to vi onto the open-list (lines 20-21). When a node is over-consistent, its distance to goal value is too high. In that case, the search-tree may need to be modified so that other nodes transition to vi instead of a different neighbor. This is accomplished by having vi adopt the rhsi value as its c_{i,goal} value (line 14), and then reinserting any node that can transition to vi into the open-list (lines 15-16). Lifelong A* inherits all theoretical properties of A*. When an admissible heuristic is used, it is optimal with respect to the graph (after a search is complete and before edge costs change). It is complete in a finite world and resolution complete in a countably infinite world.

**E. D* Algorithm**

Dynamic A* or D* is the next logical algorithmic idea after Lifelong A*. The D* algorithm is similar to lifelong A*, except that vstart is allowed to change between searches. As in Lifelong A*, edge costs are also allowed to change. The first version of D* was presented in (Stentz,[31]) and then extended in (Stentz,[32]). These versions of the algorithm rely on the more traditional notion of explicitly updating back-pointers in the search-tree to reflect graph changes, as opposed to the implicit search-tree used in Lifelong A*. A more recent version called D* lite is presented in (Koenig and Likhachev, [33]) and uses the implicit search-tree representation. As a side-note, the term ‘lite’ has since been adopted to
describe algorithms that use the implicit search-tree representation. Results in (Koenig and Likhachev, [33]) appear to show that D* lite runs more quickly than the original D* algorithm in practice. As with Lifelong A*, D* and D* lite inherit the completeness and optimality properties of the A* algorithm they are built on top of. The following discussion is valid for both D* and D* lite; however, the pseudo-code in Algorithm 11 is for D* lite. As previously noted, the only functional difference between Lifelong A* and D* Lite is that the location of vstart is allowed to move in the latter. This is advantageous in robotic.

Algorithm 2.5 D* Lite

Require: G, E, vstart, vgoal

1: procedure CalculateKey(vi)
2: return \[hstart, i + \min (ci,goal, rhsi) + M, \min (ci,goal, rhsi)\]
3: procedure UpdateVertex(vi)
4: if vi \(\notin\) vstart then
5: rhsi = \(\min j\) ci, j + cj, goal
6: if vi \(\notin\) open-list then
7: remove vi from the open-list
8: if (ci,goal \(\neq\) rhsi) then
9: update(vi, CalculateKey(vi), open-list)
10: procedure PerformSearch()
11: while CalculateKey(top(open-list)) < CalculateKey(vstart) or rhsstart \(\neq\) cstart, goal do
12: vj = get-best(open-list)
13: if (cj,goal > rhsj) then
14: cj,goal = rhsj
15: for all vi such that ei, j \(\in\) E do
16: UpdateVertex(vi)
17: else
18: cj,goal = \(\infty\)
19: UpdateVertex(vj)
20: for all vi such that ei, j \(\in\) E do
21: UpdateVertex(vi)
22: procedure main
23: for all vi \(\in\) vgoal
24: update(vi, CalculateKey(vi), open-list)
25: M = 0
26: rhsstart = 0
27: while robot is not at the goal do
28: PerformSearch()
29: move robot to new vstart
30: M = M+ the magnitude of resulting movement cost
31: wait for changes in edge costs
32: for all edges ei, j with changed costs do
33: update UpdateVertex(vi)

applications where the robot must move through the world on its way to a goal. Recall that the root of the search-tree is at vgoal. Also, ci,goal and rhsi reflect the minimum cost of moving from vi to vgoal. Therefore, the structure of the search-tree is valid, regardless of the location of vstart. This does not happen accidentally. Search is specifically performed in the backward direction (i.e. from goal to start) so that the start can move without destroying the validity of the search-tree.

F. Rapidly-exploring Random Graph

Further development of Rapidly-exploring Random Graph (RRG) is RRT [34]. This algorithm is introduced as an incremental algorithm for making a connected path map (probably includes ring). RRG algorithm in the first attempt tries to connect the nearest node to the new sample, similar to RRT. If this act is successful, the new node will be added to the set. Graph of RRT (direct tree) is a sub graph of RRG graph (indirect graph probability of inclusion of ring). In particular two graphs have the same set of nodes and the edges set of RRT graph is a sub set of edges set of RRG graph. The state-of-the-art algorithm that incorporates dynamics for a single query problem is the Rapidly exploring Random Trees (RRT) algorithm [35]. The main idea in RRT is to randomly pick states in the free space and extending the planning tree towards that state. The pseudo-code of RRT is shown in algorithm 2.6. We also assume that the given free space sampling function f generates uniformly distributed states from the free space. We use rejection sampling to implement this sampler. However, sampling states from the free space can in general be expensive. First, note that RRT is not constructing an abstract problem. Although RRT samples random states and tries to get to them, it considers a sub-goal for only one time step (notice the difference between considering a sub-goal and committing to a sub-goal). Also, because RRT does not build the planning tree in an exhaustive fashion, it is able to efficiently handle the continuity in the
action space. RRT is one of the probabilistic path planners presented by Kuffner and Lavalle in order to solve differential problems [34,35]. The important aspect of quasi RRT algorithms is that they give conclusion to the paths that may be implemented by fundamentally dynamic systems. This algorithm is in fact a search tree which grows incrementally in the intended environment. This tree grows in the manner that at first it considers an initial amount in the start point as parent node. Then, it randomly selects a node in the space, called sampling. The distance between this node and the nodes of tree is calculated and connected to the nearest node of the tree provided that the distance is within the defined area. In this way, the search tree develops incrementally until it reaches destination. Kuffner and Lavalle proved algorithm is probabilistic complete, Hesu et al. developed RRT in the manner that it had a good convergence speed[33].

Algorithm 2.6: RRT
Input: xs (starting state), g (goal function), m (a generative model), κ (a setpoint controller), ρ (a metric), u(X1) (a sampler from the free space), f (the free space indicator function).
Output: A solution trajectory.
Let T be a set of states, initialized by T = ∅.
T = T ∪ {xs}.
while Goal Not Found do
    X = u(X1).
    Y = argminY ∈ T ρ(Y ′, X).
    A = κ(Y, X).
    Z = m(Y, A).
    Add Z as a child to Y.
    T = T ∪ {Z}.
end if
Goal is Found. Return the solution trajectory.
end while

The RRT quickly expands in a few directions to quickly explore the four corners of the square. Although the construction method is simple, it is no easy task to find a method that yields such desirable behavior. Consider, for example, a naive random tree that is constructed incrementally by selecting a vertex at random, an input at random, and then applying the input to generate a new vertex. Although one might intuitively expect the tree to "randomly" explore the space, there is actually a very strong bias toward places already explored (simulation experiments yield an extremely high density of vertices with little other exploration). A random walk also suffers from a bias toward places already visited. An RRT works in the opposite manner by being biased toward places not yet visited. This can be seen by considering the Voronoi diagram of the RRT vertices. Larger Voronoi regions occur on the "frontier" of the tree. Since vertex selection is based on nearest neighbors, this implies that vertices with large Voronoi regions are more likely to be selected for expansion. On average, an RRT is constructed by iteratively breaking large Voronoi regions into smaller ones.

V. PROBABILISTIC ROADMAP METHODS
The methods described above are able to find a shortest path on a given graph. The issue most path planning methods are dealing with is how to create such a graph, or roadmap. To be useful for path planning applications, the roadmap should represent the connectivity of the free configuration space well, and cover the free configuration space such that any query configuration can easily be connected to the roadmap. See Chapter 1 for various approaches for creating such roadmaps. We will describe the most popular method in some more detail here: the Probabilistic Roadmap Method (PRM). PRM is a probabilistically complete method that is able to solve complicated path planning problems in arbitrarily high-dimensional configuration spaces. It is used throughout this thesis. The invention of PRM is usually attributed to Kavraki and Latombe [37], and Overmars [38], who worked independently on the method in the early nineties. The first appearance of the idea in literature, however, was by Glavina in 1990 [39].

The Basic PRM Approach The basic PRM-method is surprisingly simple. It constructs a roadmap by iteratively sampling configurations randomly from the configuration space. Using a collision-checker, it can be determined whether a configuration belongs to the free- or forbidden configuration space. If a configuration is collision-free, it is added as a node to the roadmap. Subsequently, it is attempted to connect this node to other nodes already present in the roadmap. To save time, a connection is only tried to nodes which are close (e.g. to all nodes within a certain distance dmax, or to the k nearest neighbors). The set of nodes to which a connection is attempted is called the neighbor set of the current node. Connections between nodes are tried using a local planner, which is a simple planner that is allowed to fail on all but the simplest queries. In the basic PRM implementation, the local planner simply tries to connect two configurations by a straight line through the configuration space. The local planner succeeds when the straight-line is collision-free, which is determined by collision-checking intermediate configurations, up to some predefined resolution. If the connection succeeds, an edge between the two associated nodes is added to the roadmap. If a connection has been tried to all nodes of the neighbor set, a new node is sampled, and the process repeats. This continues until some application-specific stop-criterion is met. Usually this is when some predefined set of query configurations are inter-connected via the roadmap. In
Algorithm 2.3, a generic sketch of the algorithm is given. If a roadmap has been constructed for a particular scene, it can be queried for motion paths between pairs of configurations in the environment. This is done by connecting the two query configurations to the roadmap, as if they were nodes newly inserted into the roadmap in the construction phase. That is, the local planner is used to connect each of the query configurations to nodes already present in the roadmap. If this succeeds, a path

Algorithm 2.7: PRM
1: repeat
2: $c \leftarrow$ a random configuration in $C$
3: if $c \in C_{\text{free}}$ then
4: $V \leftarrow V \cup \{c\}$
5: $N \leftarrow$ a subset of $V$ containing nodes neighboring $c$
6: for all $c \in N$ do
7: if the local planner connects $c$ and $c$ then
8: $E \leftarrow E \cup \{c, c'\}$
9: until some stop-criterion is met

VI. EXPERIMENTAL RESULTS
A number of scenarios were generated from easy to more challenging ones. The first scenario consisted of a simple single obstacle in the middle of the paths of robots from source to goal. The graph started the generation process from the corner and continued till the sub-graphs met and later the process continued exploring the entire robotic map. The algorithm has two stages: an offline roadmap (graph) building stage and an online planning/query stage. The aim of the offline roadmap (graph) building stage is to randomly draw a small graph across the workspace. All vertices and edges of the graph should be collision-free so that a robot may use the same graph for its motion planning. The PRM selects a number of random points (states) in the workspace as the vertices. In order to qualify being a vertex, a randomly selected point (state) must not be inside some obstacle. Let there be $k$ number of states which is an algorithm parameter. Higher are the number of vertices or $k$, better would be the results with a loss of computational time. The algorithm then attempts to connect all pairs of randomly selected vertices. If any two vertices can be connected by a straight line, the straight line is added as an edge. The concept is shown in Figure 1.

VII. CONCLUSION
This paper investigates path planning strategies in partially unknown environments. The aim of the approach was to minimize collision risk and speed up the mission by adapting to the changes in the dynamic environment. The advantages of the path selection algorithms for generating paths between predefined target points are demonstrated. Over 20 test runs are conducted. The experimental results lead to the following general conclusions. Path planning approach presented in this paper can be used even if very little is known about the environment or when the environment is completely restructured during the mission. The path selection algorithm will efficiently cover the whole space even if the environment is large. This approach helps to reduce time, risk of collisions and increases the predictability of robot’s behaviour.

REFERENCES


Figure 1: Shows the Sample results of PRM