Robot Motion Planning
PhD Qualifying Exam Written Component

Sterling McLeod
Department of Computer Science, UNC Charlotte

I. Introduction

Robot Motion Planning is the problem of moving a robot from an initial state to a goal state in an environment that may contain obstacles. States can be planar positions or high-dimensional configurations. They can also include dynamic information, such as a specific velocity and/or acceleration. Motion planning is required in almost every robotics task that exists. Unfortunately, several aspects of this problem make it difficult:

- *Kinematic structure of a robot*
  Most path-planning algorithms operate in a robot's Configuration Space (Section II). The number of dimensions in this space scales with the degrees of freedom in a robot's kinematic structure. Mobile bases will typically only have 2-3 dimensional C-spaces, but manipulators can have up to 7. Humanoid robots can have over 20. Finding a path in such high-dimensional spaces is extremely computationally expensive.
- *Complexity of an environment*
  Environment complexity is defined by the obstacles existing in it. If all obstacles are static, then the planning will not be too difficult. If some or all of the obstacles are dynamic, then the planning is made much more complicated because the representation of an environment created at one time may not be accurate for future times. Some planning algorithms build graphs that assume the obstacle are static. If the obstacles are dynamic, then the algorithms must rebuild a graph representing the state space. Rebuilding this space can be too computationally expensive to execute at real-time.
- *Amount of obstacle information*
  The inherent complexity of dynamic obstacles can be diminished if a planner knows useful information about the obstacles. If we know exactly how an obstacle will move, then a planner can know where it will be in the future and plan accordingly. Even partial information about obstacle movement would significantly reduce the planning problem complexity because it allows a system to make assumptions. For instance, in the domain of self-driving cars, an assumption can be made that the surrounding cars will only move forward. This means that a planner does not have to consider the case where an obstacle may move directly at the robot. Other domains, such as warehouse robots, cannot have this assumption applied. In order to have a general planner, one must assume as little as possible about obstacles and the algorithm must be developed to handle lots of various situations.

The earliest works revolve around complete planners operating on a roadmap of the robot's state space. However, building such a roadmap in a high-dimensional space is too computationally expensive for real-time planning. Artificial Potential Fields came about as an attempt to move a robot without computing such roadmaps. However, these methods still require an exact representation of a robot's C-space, and they suffer from local minima. Sampling-based algorithms gave roboticists a way to build an approximation of a robot's C-space suitable for planning, which made many high-dimensional planning problems feasible to solve.

This survey will discuss these various approaches to motion planning. First, the Configuration Space of a robot is defined. Then, the problem is formulated in Section III. Combinatorial methods and their shortcomings are described in Section IV. Artificial Potential Functions are briefly discussed in Section V and are discussed more thoroughly when covering Elastic Strips in Section VIII. Sampling-based approaches are covered in Section VI, followed by Kinodynamic Planning in Section VII. Real-time Motion Planning is discussed in Section VIII, which covers the Elastic Strips and RAMP frameworks. A brief overview of Collision Detection in covered in Section IX. Section X concludes the paper.

II. Configuration Space

A robot's *Configuration Space*, or C-space, is a manifold mapping the set of all transformations that can be applied to a robot to Euclidean space. The dimensionality of this space is determined by the number of degrees of freedom in a robotic system. A robot's *configuration* is a complete specification of the positions of a robot's points. It is typically represented as a vector specifying the values for each of a robot's degrees of freedom. Each point in the C-space is a vector specifying a transformation of the robot. A more intuitive way to think of C-space is as the set of all possible configurations that a robot can have.

In order to move a robot's end-effector, actuators have to produce rotations or translation around the joints of a robot. This means that any motion of a robot is entirely controlled by the motion of its joints, or the change in configuration. Therefore, robot motion planning is concerned with how to move a robot from an initial configuration to a goal configuration.

To define the C-space for a robot more formally, both the range and type of motion of the robot must be considered. If a robot is a rigid-body object translating in a plane, then its transformation can be expressed in terms of x and y. This yields a manifold $C = \mathbb{R}^2$. If rotations can be applied to the robot, then another dimension must be added to the C-space. Let $\theta = [0, 2\pi]$ be the range of rotation for the robot. The C-space for a rigid body robot that can translate along x and y and rotate $[0, 2\pi]$ in a plane becomes $C = \mathbb{R}^2 \times \mathbb{S}^1$.

For rigid bodies moving in a 3D space, the C-space becomes more complicated. Generally, objects in a 3D space can translate in three dimensions and can rotate about three axes independently. This set of transformations is characterized by a position vector, p, and a rotation matrix, R, relative to a fixed frame. The resulting C-space will be six-dimensional: $C = \mathbb{R}^3 \times \mathbb{RP}^3$.

Manipulators consist of multiple rigid bodies that can either translate or rotate independently of one another. When computing the C-space of articulated bodies, the C-spaces of *each* body must be combined. If a manipulator is a chain of n links connected by revolute joints and each joint can achieve full rotation $\theta_i = [0, 2\pi]$, then the C-space is defined as

$$C = \mathbb{S}^1 \times \mathbb{S}^1 \times ... \times \mathbb{S}^1 = \mathbb{T}^n \tag{1}$$

Visually, this C-space is an n-dimensional torus. If a manipulator can translate in a 3D space, then an additional $\mathbb{R}^3$ must be considered: $C = \mathbb{R}^3 \times \mathbb{T}^n$. In the real world, however, it may not be possible to achieve a full rotation around each joint. If that is the case, then the topological space representing rotation in the C-space would be $\mathbb{R}^n$ instead of $\mathbb{T}^n$.

The C-space in a motion planning problem is separated into two subspaces, C_{obs} and C_{free}. The subspace C_{obs} represents the set of configurations containing obstacle regions and $C_{free} = C \backslash C_{obs}$. Let A be a rigid body robot and O be an obstacle region. Both A and O exist in a world space W. The C-space of A is C and a configuration $q \in C$ is defined as $q = (x, y, \theta)$. The obstacle region, C_{obs}, is defined as

$$C_{obs} = \{q \in C | A(q) \cap O \neq \emptyset\} \tag{2}$$

In order to solve a motion planning problem, C_{obs} must be computed or approximated. In the case of a rigid body robot translating in a plane that contains polygonal obstacles, C_{obs} can be found by *growing* the obstacles. For instance, if the robot is a circle robot, all obstacles have each vertex extended to the size of the robot's radius. There is a more systematic way to compute how the obstacles grow if the robot is a convex polygon. The Minkowski Sum is computed for the set of robot and obstacle vertices:

$$A \oplus B = \{(a, b) | a \in A, b \in B\} \tag{3}$$

where A is the set of robot vertices and B is the set of obstacle vertices. The convex hull of $A \oplus B$ represents the final C-space obstacle region. To picture this algorithm visually, it is the result of sliding the robot along the obstacle's edges at the robot's reference vertex. This sliding process was first proposed in [1]. The works [2] and [3] show how the sliding algorithm can be extended to 3 dimensions and manipulator models. The overall result is the same (obstacles are grown based on the robot size), however, the time to compute the grown obstacles is significantly increased because this algorithm has complexity $O(n + m)$ where n is the number of robot edges and m is the number of obstacle edges. For a robot that can also rotate in the plane, only considering translations is not enough - this process must also include rotations of the robot. For an articulated body robot, i.e. manipulator, this process must be repeated for each link on the robot. Thus, computing the full C-space scales with the number of dimensions in a robot's C-space.

Path-planning problems usually operate in a robot's C-space. Since motion planning problems need to obtain a trajectory, some algorithms operate in a robot's Configuration-Time Space (CT-space). A robot's CT-space only differs from its C-space by adding a dimension to represent time.

III. Path Planning

Let Q_{free} denote a robot's Configuration-free space. Given an initial configuration, q_S and a goal configuration q_G, the goal of path planning is to find a path $\rho[0, 1] \rightarrow Q_{free}$ such that $\rho(0) = q_S$ and $\rho(1) = q_G$. This path will enable a robot to move from the initial configuration to the goal configuration without colliding with any obstacles in the environment.

A solution to the full motion planning problem uses the points in a path as knot points along a trajectory. Trajectory following can be done after a path has been generated by interpolating intermediate points using a control algorithm, or the trajectory can be generated while a path is being generated, which is known as *Kinodynamic Planning* and will be covered in Section VII.

This paper focuses on the planning aspect of motion planning because the essence of motion planning is in exploring a state space as rapidly as possible. Trajectory-following is closer to the field of *control* and is performed only after a state space has been searched to form a path.

IV. Combinatorial Methods

Combinatorial motion planning methods are complete planning algorithms that construct a roadmap in a known continuous space to search through for path planning queries. These methods compute the exact configuration space for a robot and its environment to construct the roadmap. These algorithms work well for simple robotics problems, but do not scale up to many real-world applications seen today.

A. Roadmaps

Let G be a graph structure of a topological space that maps to a robot's configuration-free space, C_{free}. The set S is the set of points reachable in C_{free} by G. In order for G to be a roadmap, two conditions must be satisfied:

1) A graph-search algorithm can compute a path from any initial node to any goal node in the space represented by G.

2) Given initial and goal configurations q_I and q_G, respectively, in C_{free}, it is possible to connect corresponding nodes s_1 and s_2 in S. More formally, if there exists a path $\tau : [0,1] \rightarrow C_{free}$, such that $\tau(0) = q_I$ and $\tau(1) = q_G$, then there also exists a path $\tau'[0,1] \rightarrow S$, such that $\tau'(0) = s_1$ and $\tau'(1) = s_2$, where s_1 and s_2 are nodes on the roadmap.

An important implication of the second condition is that solutions to path-planning queries are not missed due to the roadmap not capturing the connectivity of the configuration space. Essentially, the second condition ensures that algorithms searching on a roadmap can be complete. The first condition makes it always possible to connect an initial and goal configuration to a roadmap. This is a necessary condition because the initial and goal nodes are not connected to the roadmap when it is constructed. They are only connected when beginning a query.

B. Cell Decomposition

The goal of cell decomposition is to partition a topological space into cells that lie entirely in C_{free}. From such a partitioning, a roadmap can be constructed in order to reduce the path planning problem to a graph-search problem. There are several choices for decomposition. Some methods are best suited for polygonal spaces, some are better suited for higher-dimensions. However, all cell decompositions must satisfy three properties:

1) A path from any point in a cell to any point in a separate cell should lie in C_{free}.
2) Adjacency information is easy to extract.
3) Given any configuration q, calculating which cell q lies in should be simple.

The first condition is the most important. If it is known that the robot can move from one cell to another without encountering C_{obs}, then the problem is reduced to finding a path among a small number of cells, i.e. a graph-search problem. The second and third conditions are necessary to make the querying fast.

1) Complexes: A *complex* is defined as a collection of cells and their boundaries. The difference between the terms *cell decomposition* and *complex* is that complexes contain additional pieces of information about how cells fit together, whereas cell decompositions only contain the actual cells themselves.

The type of complex for a space will dictate the cell decomposition method that can be derived for the space. Two types of complexes are described below.

A *simplicial complex* satisfies the following conditions:

1) Any face of the complex is also in the complex.
2) The intersection of any two simplexes in the complex is a common face of both simplexes

Essentially, all of cells must be connected and share boundaries.

A *singular complex* is a generalized version of simplicial complexes that can be defined on a manifold of $\mathbb{R}^n$. Singular complexes also satisfy two conditions:

1) For each dimension k, the set $S_k \subseteq X$ must be closed.
2) Each k-cell is an open set in the topological subspace S_k. Note that 0-cells are open in S_0, even though they are usually closed in X.

2) Approaches: Vertical Cell Decomposition partitions the free C-space into rectangles, triangles and trapezoids. These simple shapes represent the cells of C_{free}.

At each vertex in C_{obs}, vertical rays are extended above and below the vertex until another point in C_{obs} is reached. These rays define edges for the polygonal cells that partition C_{free}. The polygons that form C_{obs} can be either convex or concave, and the cells in C_{free} will be either triangular or trapezoidal.

Building a roadmap from the partitioning amounts to choosing sample points from each cell and connecting them to edges of the cell. More formally, a roadmap $G(V, E)$ is defined as follows: For every cell, C_i, define an interior point q_i. For each vertical edge in C_i, define a point s_i. Add the points q_i and s_i to V. Then, create an edge, e_i, between q_i and the sample points on its cell's vertical edges and add e_i to E. The resulting graph is a roadmap that can be used for efficient path planning.

The running time of this algorithm is $O(nlogn)$ when using the *line-sweep* principle. This principle is the idea of a line sweeping across the space and noting where there are interesting points in the space. In this application, the line is vertical and the interesting points are the vertices of obstacles. An *event* occurs each time the line finds an interesting point.

Triangulation is an alternative to vertical decomposition for polygonal spaces. The cells formed from triangulation will be triangles. The running time of a naive approach to triangulation is $O(n^3)$. That is far too long, even for some 2D spaces. However, the running time can be reduced to $O(n^2 logn)$ with radial sweeping.

Cylindrical decomposition is similar to vertical decomposition in that it works by extending rays from all the obstacle vertices, but in the cylindrical method the rays do not stop until they reach the boundary of the C-space. This results in cells that alternate between regions of C_{free} and C_{obs}.

3) 3D Decomposition: Cell decomposition in 3 (or more) dimensions is desirable since most motion planning problems have a C-space of more than 2 dimensions. The methods discussed previously work well for 2D, but only cylindrical decomposition generalizes well to higher dimensions. Vertical decomposition can be recursively applied to k-cells of higher dimensions, but it only succeeds if C_{obs} is represented as a semi-algebraic model with linear primitives.

4) Querying: To solve a path planning query using the roadmap generated by a cell decomposition method, the initial and goal configurations, q_I and q_G respectively, need to be added to the roadmap. The third condition for cell decomposition listed in the previous section comes into play here. It should be simple to determine which cells q_I and q_G are in. Let C_I and C_G denote the cells that contain q_I and q_G, respectively. Once C_0 and C_G are identified, edges are found between the configurations and the corresponding

cell's sample point. Thus, an edge is formed between q_I and the sample point of C_I. Similarly, an edge is formed between q_G and the sample point of C_G. Both edges are added to the roadmap $G(V, E)$. Graph search algorithms can be applied to the roadmap to find a path from q_I and q_G or find if a path does not exist.

C. Combinatorial Motion Planning Complexity

There are two major issues with combinatorial methods. The first is that the upper bound on the number of possible paths in a roadmap scales factiorally with the number of edges and vertices in a graph.

Consider the simpler problem of a robot moving in a directed acyclic grid with dimensions $m \times n$ and each edge is the same length. If the robot is at an inital location (s_x, s_y) and the goal is given as (g_x, g_y), then the total number of solutions is

$$|P| = \binom{|g_x - s_x| + |g_y - s_y|}{|g_x - s_x|} \tag{4}$$

Given (4), the total number of paths to move from one corner to the opposite corner can be computer as:

$$|P| = \binom{m+n}{m} \tag{5}$$

These equations assume that the robot would never have a cyclic path. This assumption may hold for path planning in static environments, but it is one that cannot be made for the more general problem of path planning in dynamic environments. A robot may need to move back to a previous position due to the environment changing or the robot made need a path with a new direction to avoid an obstacle. Thus, the number of possible paths for a real-time motion planning system would be even more than the directed acyclic graph case.

The second major issue with these methods is that the exact C-space must be computed to perform the decomposition methods. As noted in Section **??**, constructing the C-space will scale with the degrees of freedom of a robot (i.e. its C-space dimensionality) and the number of obstacle edges in the workspace.

While it is good to understand the theoretical components of these algorithms, it would be far too time-consuming to run any combinatorial method on a real-time motion planning system.

V. Artificial Potential Fields

In an artificial potential field approach [4], a robot is treated like a particle in a gradient field. The goal state emits an attractive force that acts on the robot and the obstacles emit repulsive forces. The combination of these forces results in a vector field that guides the robot from its initial state to the goal state. At each location, a robot's motion is dictated by the magnitude and direction of the gradient vector at the location. Let $U_{att}(q)$ denote the attractive force emitted from the goal at configuration q, and $U_{rep}(q)$ denote the repulsive force from the closest obstacle in an environment at configuration q. The overall force acting on the robot is:

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{6}$$

The motion of the robot is determined by ∇U. The speed at q is the magnitude of $\nabla U(q)$, and the direction of motion is the direction of $\nabla U(q)$.

Encountering local minima is a problem when using this approach. If a robot is led into an area dominated by repulsive forces, then that area may cause the robot to move in a loop. The most common method of escaping local minima is random walks. When it is detected that the robot has encountered a local minima, a random direction is chosen and the robot is moved in that direction for a variable amount of time. After moving in a random direction, it is hoped that the robot will have escaped the local minima region and can resume moving towards the goal.

Another issue with using potential fields to move a robot is that the exact C-space must be computed. As discussed before, this is usually infeasible when the C-space is high-dimensional.

This approach inspired the Elastic Strips approach described in Section VIII-B that addresses these problems by using a higher-level planner to maintain global path information, and reduce the necessary parts of the C-space that must be computed to use potential field controllers.

VI. Sampling Based Methods

Computing the exact configuration space in high dimensions becomes extremely expensive. This makes previous methods inapplicable to motion planning problems of high dimensions, such as a 6-DOF manipulation problem. Sampling-based methods were introduced in order to handle motion planning problems with such high complexity. Sampling algorithms sacrifice completeness and optimality by using an approximation of the C-space, but are able obtain solutions to high-dimensional problems in feasible amounts of time.

Sampling based methods do not access the C-space directly. An associated collision detection algorithm will test a sampled configuration of the robot against obstacles in the environment. Based on the result of collision detection, the sampling algorithm knows if a sample lies in the free or obstacle portions of the C-space. Connections between samples are made through local planner. After sampling many points, the result is an approximation of the C-space, rather than a complete description.

Sampling has become the method of choice for most modern robotics applications. A sampling based method was first proposed in 1996 [5] that builds a graph representation of the C-space approximation and then uses graph-search techniques to find a final path. A tree-based approach was introduced shortly after that could take kinodynamic constraints into account [6]. These two approaches have been extended and iterated upon many times and remain among the top approaches in the field.

A. Probabilistic Roadmaps

The basic Probabilistic Roadmaps algorithm [5] works in two phases: learning and querying.

1) Learning Phase: The objective of the *learning* phase is to build the C-space approximation. An overview of the learning phase is summarized below in Algorithm 1.

Algorithm 1 PRM Learning Phase

```
N : number of nodes to include in the roadmap
V ← ∅ // Initialize set of vertices
E ← ∅ // Initialize set of edges
i ← 0
while i < N do
    q ← randomly sampled configuration
    if q ∈ C_free then
        V ← V ∪ {q};
        V_c ← neighborhood set of q;
        for each v ∈ V_c do
            if Can_Connect(v, q) then
                e ←new edge(q, v)
                E ← E ∪ {e}
            end if
        end for
    end if
end while
```

The goal is to build an undirected graph containing N nodes. These nodes correspond the collision-free points in a robot's configuration space. When a point is sampled and determined to be collision-free, a neighborhood of points from the existing graph is found. For each of these neighborhood points, if a local planner can find a collision-free path from the sampled point to the neighborhood points, an edge is formed connecting the nodes. If a sampled node is found to be in collision, the node is thrown away.

A local planner is utilized in order to create edges within the graph. Designing a local planner can be difficult, however, because it needs to plan in a high-dimensional space. Cheap and fast planners are possible since there is no need to consider a global objective. Even if fast local planners are not complete, they are desired over more powerful methods because they allow for more connections to be considered during the learning phase. This is true because if a local planner is fast, then checking connections between many nodes is fast. Thus, the faster a local planner is, the more connections can be made in a small amount of time. It is desirable to sample as many nodes and make as many connections as possible since more nodes and connections will generally lead to a more accurate C-space approximation. For these reasons, local planners that connect nodes via straight lines are commonly used.

Determining the neighborhood of a node is based on the value of a distance function. The distance function is closely related to the choice of local planner because the distance should be proportional to the chance that the local planner can connect the two nodes with a collision-free path. For straight line local planners, the distance function can be the Euclidean distance between two nodes' positions in the robot's workspace. For more complicated local planners, however, more involved distance functions must be defined.

Narrow passages are a critical concern during the learning phase. A narrow passage is a region that is mostly enclosed by obstacles. These regions are of concern because it is difficult to sample inside of them and to find connections to nodes inside the regions. If a robot must pass through a narrow passage to reach its goal, then the learning phase may be prolonged to the point of being impractical.

The PRM method tries to address narrow passages by performing an *expansion* step after constructing the graph, in order to improve the graph's connectivity. To accomplish that, nodes that lie in narrow passages are selected and an attempt to find neighboring points in C_f is made.

The main difficulties in the expansion step are to identify which nodes lie in a narrow passage (those are the nodes that need expansion) and how to find neighboring points.

To find potential expansion nodes, a heuristic needs to be defined that predicts the likelihood of some node c being unconnected. In [5], the following was used:

$$r_f(c) = \frac{f(c)}{n(c)+1} \tag{7}$$

where $n(c)$ is the number of times the planner tried to connect node c and $f(c)$ is the number of successful connects made to c. This function is the *failure ratio* of node c.

Random walks are a common method for actually expanding a node. A direction is chosen at random and the direction is followed until an obstacle is reached. An edge is then formed between node c and a new node n that is a node somewhere in between c and the obstacle that was reached in the chosen direction. This process repeats by selecting new random directions for a pre-specified amount of time.

2) Query Phase: The *query* phase searches the roadmap constructed during the learning phase to find a path from an initial configuration to a goal configuration.

The first step in this phase is to connect the starting configuration, s, and goal configuration, g, to the roadmap. The local planner used in the learning phase is used for this task. The procedure for connecting s to the roadmap, R, begins by identifying the closest node on R to s. The local planner attempts to form a path between s and the closest node. If it cannot find a path, the planner uses the next-closest node on R to connect to s. This process repeats until a connection is successful. Once s has been connected, the goal configuration g is connected in the same way. Once both s and g are connected to R, any graph-search algorithm will return a path from s to g in the robot's configuration space.

3) Further Discussion: The PRM planner is considered *probabilistically* complete, meaning that in most cases it will not explore every point in the C-space, but it will explore all points in the C-space given enough time. The likelihood of determining if a path exists is proportional to the time spent in the learning phase.

Dynamic environments are difficult to address with the PRM approach. Constructing the roadmap can take significant computation time and is only useful for the state of the environment while constructing the roadmap. If any obstacles move, then the roadmap needs to be modified or completely remade based on the new state of the environment.

B. Rapidly-Exploring Random Trees

Rapidly-exploring Random Trees (RRT) is a popular sampling based approach for high dimensional spaces [6]. The key difference between RRT and PRM is that RRT uses a tree structure to approximate a robot's C-space, rather than the undirected graph structure used in PRM. This approach leads to several attractive features: it is suitable for handling non-holonomic and kinodynamic constraints (Section VII), it requires few parameters, and is more apt to handle real-time planning in dynamic environments.

Pseudocode for building an RRT can be seen below:

Algorithm 2 RRT Overview

```
1: function CONSTRUCT RRT(x_init, x_goal, Δt)
2:     T.init(x_init)
3:     while x_goal ∉ T do
4:         x_rand = newly sampled state
5:         x_near = nearest neighbor of x_rand
6:         u = input needed to move from x_near to x_rand
7:         x_new = {x_near, u, Δt}
8:         T.add_vertex(x_new)
9:         T.add_edge(x_near, x_new, u)
10:    end while
11:    Return T
12: end function
```

At each iteration, a new state, x_{rand}, is randomly sampled. The parent of x_{rand} will be the vertex on the tree that is nearest to the sample. The control input, u, is then selected to move from x_{near} to x_{rand} without moving into any regions occupied by obstacles in the state space. The new state, x_{new}, is formed from the nearest neighbor, the control input, and a small time increment. The new node is added as a vertex on the tree, and the edge connecting x_{near} to x_{new} by applying u is added to the tree as a new edge. This process repeats until the goal state has been added to the tree. Since every state has a parent on the RRT, adding the goal state to the tree means that a path has been found.

The RRT approach has several nice properties. Similar to PRM, the RRT algorithm is probabilistically complete. The chance of determining whether or not a path exists increases as times goes on. As previously discussed, RRT is able to consider kinodynamic constraints. This is largely due to using a tree over an undirected graph. Since each state will have only one directed connection, it is easy to ensure that a robot's dynamic constraints will be satisfied as it move from configuration to configuration. Another attractive property is that the tree will always remain connected, even in narrow passages. Graphs constructed with the PRM are not guaranteed to remain connected because the connections are based on a vertex's neighborhood. If there are no existing vertices nearby the new vertex, then the vertex will have no connections, but will still be added to the graph.

One of the most appealing aspects of RRT is that it is bias towards unexplored areas of a space. This occurs because the selection of new states is based on the nearest neighbor calculation. As RRT explores a space, the regions separated by the tree's edges become close to voroni regions of the space.

Bi-directional searching was presented in [8] to significantly increase the performance of RRT. Two trees are instantiated - one at the initial state x_{init} and one at the goal state x_{goal}. When the trees connect, the query is complete. At each iteration of the algorithm, one of the trees, T_0, is extended by the usual set of procedures in RRT. Let x_{new} be the newly added state to T_0. A new procedure, $Connect$, is called to attempt to add x_{new} to the second tree, T_1. If this connection is unsucessful, then the trees are swapped so that T_1 can be extended.

C. Discussion

Both sampling-based approaches sample stochastically in order to build an approximation of a C-space. They differ in the data structure used as the C-space approximation. Because each approach builds an approximation of the C-space, the basic versions are not able to find an optimal path to the goal. Several extensions have been made to these approaches to improve the optimality of the resulting paths.

For PRM, most of the immediate work following [5] focused on the learning phase of PRM to improve its ability to capture narrow passageways in the C-space. This is because narrow passageways are the most significant challenge when using a sampling-based approach. Extensions of the PRM algorithm target one of two aspects: the node generation process or the process of finding a neighborhood of candidate connections.

Obstacle Based PRM (OBPRM) [9] generates nodes based on obstacle locations. The obstacle information guides the node generation process. It can generate both contact configurations (for manipulation planning) and configurations in free space for general path-planning. Each sampled state is translated towards an obstacle until collision is made. Then, the robot is rotated randomly (while maintaining collision) and the resulting configuration is recorded as c_{in}. A random direction is selected to translate c_{in} on until a free configuration, c_{out}, is found. The contact configuration between c_{in} and c_{out} can be found through binary search. Medial Axis PRM (MAPRM) [10] proposes a node generation strategy based on the medial axis of a space. Each new sampled state is retracted onto the medial axis of the robot's configuration-free space. The exact medial axis does not need to computed with this approach. Each sampled configuration is moved towards its nearest neighbor on the graph via bisection until there is no unique closest node on the graph. An approach to sampling based on Gaussian distribution was presented in [11]. This approach assigns probability values to C-space regions based on their distance from C-space obstacles. Higher

values are given to areas closer to obstacles. Each iteration of building the graph does the following: a configuration is obtained via uniform sampling, then a distance value d is chosen based on a Gaussian distribution uncertainty, and a second configuration is obtained at distance d from the first sampled configuration, i.e. closer to an obstacle. All of these strategies help the planner deal with narrow passageways by generating nodes in regions close to obstacles, which are the hardest regions to plan in.

A visilibity roadmap approach [12] significantly reduces the number of nodes by only keeping nodes that either connect two connected components or are not visible by any "guard" nodes in the roadmap. A node is visible by another node if they can be connected with a straight-line that lies entirely in the free portion of a robot's C-space. Guard nodes are ones that are not visible by other guard nodes.

Significant reductions in planning time were achieved by taking a "lazy" approach [13]. This work builds a graph without considering collision. After a shortest path is found, collision checking is performed on the path. If collision is found, then the corresponding nodes and/or edges are removed from the graph, and a new shortest path is found. This process repeats until a shortest path is found that contains no collision.

The RRT approach also received a significant amount of work to improve its overall speed and sampling. Using RRT for real-time robot navigation was presented in [14]. This work proposed the Extended RRT (ERRT) algorithm in order to use RRT on a real robot in a dynamic environment. To use the basic version of RRT in a dynamic environment, an entirely new tree must be created each time the robot needs to change trajectories to adapt to its environment. Using the intuition that prior trees contain states that may lead to the goal, the authors used a *waypoint cache* to reduce the iterations needed to find the goal. A waypoint cache is a set of states that were used in a previous path. When a new path must be found, the points in the waypoint cache are used as samples. Simulation experiments show that this approach can significantly reduce planning time when the queries become complicated to solve. This work was also implemented onto the RoboCup F180 robot system, which was the first time RRT was shown to work on a real robot.

Dynamic-Domain RRT [15] is an approach to improve the node generation of RRT by limiting the size of the sampling area. If v is a boundary point at some distance R from a C-space obstacle, then the *boundary domain* is the intersection of the Voroni region of v and a sphere of radius R centered at v. The boundary domains of each existing point on the RRT create the *Dynamic Domain* for the RRT. Rather than adding nodes sampled from the entire state space, only nodes that are within the Dynamic Domain of an RRT are considered.

An Obstacle Based RRT approach [16] uses a group of growth operators to generate nodes that are close to obstacles. These operators use vectors between an obstacle's vertices to determine what direction to move a sample in.

The work done in [17] retracts nodes to C_{free} with a local optimization approach. Given a sample q_r that is in collision with an obstacle, the algorithm makes an initial guess q_n, and then creates a new sample q_c in the contact space that is found between q_r and q_n. A set of nodes, S, can be found in the contact space that are within a distance threshold of q_r. The algorithm then performs an optimization loop in order to find the node in S closest to the original sample q_r.

Using PRM and RRT to find an optimal path was explored in [18]. Three algorithms, PRM*, RRG, and RRT*, were proposed that extend the standard PRM and RRT algorithms. Analysis in this paper shows that the three algorithms presented are 1) asymptotically optimal, and 2) within a constant factor of the computational complexity of the standard sampling algorithms.

The PRM* algorithm is only slightly different from the standard version. In the standard version of PRM, the nearest neighbors for a sample are obtained with a fixed radius of the sample. The PRM* algorithm uses a variable radius that is a function of the number of nodes in the roadmap. Let n denote the number of vertices in the roadmap. The radius function used in PRM* is

$$r(n) = 2(1 + 1/d)^{1/d} * (\mu(X_{free})/\zeta_d)^{1/d} * (log(n)/n)^{1/d} \tag{8}$$

where d is the dimension of the state space X, $\mu(X_{free})$ is the volume of the obstacle-free space, and ζ_d is the volume of the unit ball in the d-dimensional Euclidean space. The radius decreases as n increases and the rate of decay is proportional to $log(n)$. Similar to PRM, the PRM* algorithm is meant to handle multiple-query problems.

The *Rapidly exploring Random Graph* (RRG) algorithm is more suited for single-query problems. It is an incremental algorithm that builds a connected roadmap to use for planning. It is an extension of the RRT algorithm with one key difference: when a new vertex is added to the vertex set, all vertices that already exist in the set within a certain radius are connected to the new vertex. This means that the resulting data structure is a graph rather than a tree.

RRT* modifies RRG by maintaining a tree structure rather than a graph. A subset of edges is considered redundant and removed so that cycles are not formed. These edges are ones that are not part of the shortest path from the root of the tree to a vertex.

Speaking generally, sampling-based approaches work well for solving most robot motion planning problems. However, the time it takes to rebuild roadmaps, plan a new path, and recompute a new trajectory can be too expensive to avoid obstacles of unforeseen motion. The RRT approach is the most successful in handling real-time robot motion planning because it is designed to handle single-query problems. The RRT approach extends intuitively to Kinodynamic Planning (Section VII) which can significantly speed up the time to compute an entire trajectories (instead of only a path).

VII. Kinodynamic Motion Planning

Kinodynamic Planning is an approach to robot motion planning that integrates the kinematic and dynamic constraints of a robot into the planning aspect of the solution.

In other words, it solves both path-planning and trajectory-following simultaneously. Prior motion planning approaches required these two problems to be done separately. Path-planning would be performed without considering a robot's dynamic constraints to produce a geometric path, and then a trajectory-following algorithm would interpolate points to move the robot along the path.

Since an implicit goal of planning is usually to find an optimal solution (or the most-optimal available solution), it is possible that a kinodynamic planner may lead to an unsafe trajectory based on the robot's dynamic constraints. Therefore, a safety margin can be used to ensure that a solution maintains a certain distance from obstacles. A safety margin, δ, is defined based on a robot's dynamic constraints. A δ_v-*safe kinodynamic solution* will move a robot from a starting state (position and velocity) to a goal state while avoiding obstacles by a safety margin δ_v. Mathematically, the effect of the safety margin is the following:

$$\delta_v(c_0, c_1)(\dot{p}(t)) = c_0 + c_1||\dot{p}(t)|| \tag{9}$$

where c_0 and c_1 are positive scalar values.

Additional parameters to this problem are l, the side length of a cube that bounds all free configurations, a_{max}, the maximum acceleration, and v_{max}, the maximum speed.

The problem was first proposed in [19], where an approximate solution was presented based on graph-search. The state space for this approach, denoted as TC, is the phase space of a robot's C-space. A point in this space represents both the robot's position and velocity, e.g. $S = (s, \dot{s})$. The state space is transformed into a directed graph where the vertices are discretized values from TC and the edges represent trajectory segments.

The trajectory segments are formed from (a, τ)-*bangs*. Given a maximum acceleration, a, let the set of constant acceleration values $[-a, a]$ be denoted as A. A timestep τ can be chosen such that the velocity bound v_{max} is an integer multiple of $a\tau$. A (a, τ)-*bang* is the result of applying an acceleration value from A for duration τ.

The timestep τ is based on a function of a_{max}, v_{max}, an approximation parameter ϵ, and scalar values c_0 and c_1. The scalar values define a safety margin and the approximation parameter defines a bound for closeness to the desired start and goal states. The chosen timestep is the largest value satisfying the following condition:

$$\tau \leq \frac{\epsilon}{13} min\left(\sqrt{\frac{2c_0\epsilon}{a(c_1+1)}}, \frac{c_0\epsilon}{a(c_1+1)}\right) \tag{10}$$

Given τ, the initial state $S^* = (s, \dot{s})$ is chosen such that $s^* = s$ and $\dot{s}^*$ is the multiple of $a\tau$ closest to $\frac{\dot{s}}{1+\epsilon}$.

Breadth-first Search is then applied to the embedded graph in TC. The initial state is S^* and the goal state is any vertex within a certain threshold of the desired goal state, e.g. any vertex within $\left(\frac{a\tau^2}{2}, \frac{a\tau}{2}\right)$ of $\left(g, \frac{\dot{g}}{1+\epsilon}\right)$. Safety margin constraints are checked for each trajectory segment that the search algorithm considers.

A theorem is presented in this work to show that the complexity of the algorithm. Given the kinodynamic bounds of a problem, the upper bound for this algorithm's running time is

$$O\left(n\left[\frac{lv\gamma^3}{\epsilon^6}\right]^d\right) \tag{11}$$

where $d = 2, 3$, n is the number of bounding halfplanes on all obstacles, and

$$\gamma = max\left(\frac{a_{max}(c_1+1)}{c_0}, \sqrt{\frac{a_{max}(c_1+1)}{2c_0}}, \frac{a_{max}}{v_{max}}\right). \tag{12}$$

This work can be used for kinodynamic planning of a rigid robot translating in a 3D space that contains convex polyhedra. This initial work is sufficient for problems with low degrees of freedom, but could not be shown to be a feasible approach to more complicated problems. Randomized approaches were proposed to handle kinodynamic planning problems with high degrees of freedom.

An RRT approach was proposed in [7]. RRT lends itself to kinodynamic planning very easily because part of forming the connections in RRT is based on the control needed to move from node to node. The main difference is to change the state space from the robot's C-space to the phase-space of the C-space so that each state is a (*position*, *velocity*) pair.

Differential constraints, such as the non-holonomic constraint, are represented as the implicit function

$$\dot{x} = f(x, u) \tag{13}$$

where $u \in U$ and U is the set of all inputs to a system. A numerical approximation for (13) is needed to find the resulting state from applying u to x. The work in [7] uses fourth-order Runge-Kutta integration, but a similar numerical approximation technique will suffice.

The randomized kinodynamic planner based on RRT assumes that an environment is static. It extends the notion of an obstacle region in a state space to be a region of inevitable collision. This region contains all the states where the robot is either in collision or at a state where it will not be able to avoid an obstacle due to its dynamics. The region of inevitable collision subsumes the obstacle region in a state space.

This approach has been shown to work for several systems. For each system, the dynamic model must be defined as well as a distance metric in order to approximate a nearest neighbor for each node. The most complicated system described in experimental results is a three-dimensional spacecraft with a 12-dimensional state space. A bidirectional search method can also be applied to this work as in [8].

The RRT approach did not consider dynamic obstacles, however. Dynamic obstacles in kinodynamic planning were considered using a different randomized approach [20]. This work plans in the state×time space of a robot by sampling to

build an approximation of the search space. A vision module runs simultaneously with the planner to estimate obstacle motion.

Let S denote the state space of a robot, e.g. in the case of a car-like robot a state $s \in S$ would contain $(x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$. Let ST denote the state$\times$time space defined as $S \times [0, +\infty)$. Given an initial state$\times$time pair (s_0, t_0) and a goal pair (s_g, t_g), the output of this planner is a function $u : [t_0, t_g] \rightarrow \Omega$ that leads to a collision-free trajectory $\tau : t \in [t_0, t_g] \rightarrow \tau(t) = (s(t), t)$ such that $s(t_0) = s_0$ and $s(t_g) = s_g$.

A query is solved by iteratively building a tree-shaped roadmap T whose root is the initial state$\times$time pair through sampling until the goal is connected to the tree. Each iteration randomly selects an existing node on the roadmap, (s, t), a time $t' \leq t_g$, and a control u. Then, the control input u is applied to (s, t) for time t'. This will form a trajectory in the state$\times$time space. If this trajectory lies in free space, then the endpoint (s', t') is added as a node in the roadmap. The trajectory itself is stored as a directed edge between (s, t) and (s', t') and u is also stored with the new edge. The algorithm exits when a trajectory leads to a new state that is in a neighborhood of the goal state. If some number of iterations N occur and a solution has not been found, then the algorithm will return a failure to find a trajectory.

This approach was applied to two robot systems. The first system was a pair of non-holonomic bases that must remain within a minimum and maximum distance of each other. The state space of this system had 6 dimensions: $(x_1, y_1, \theta_1, x_2, y_2, \theta_2)$. The second system was an air-cushioned robot that moves frictionlessly on a flat table. The state space of this system had 4 dimensions: $(x, y, \dot{x}, \dot{y})$. When adding the time dimension to the state spaces, the algorithm built a roadmap in 7 and 5 dimensions, respectively.

In simulation, obstacle motion is given to the planner before executing the algorithm. The results verify that the planner is able to find a collision-free trajectory in the precense of moving obstacles for the majority of the time over the three cases shown. In experiments with real robots, obstacle motion is captured with an overhead camera. Obstacle motion is passed to the planner before executing and the planner assumes constant velocity for each obstacle. If the obstacle velocities significantly change, the camera will alert the planner with new information so that it can replan a new trajectory with new obstacle trajectories. Several real robot experiments show that the planner was able to navigate the robot around moving obstacles when obstacle motion is known beforehand.

The best kinodynamic planners are sampling-based due to planning under differential constraints being so difficult. Because of this, recent work is still based on the approaches laid out above.

VIII. Real-time Motion Planning

Real-time Motion Planning is concerned with enabling a robot to react to unforeseen changes in an environment. These changes are unknown to the robot beforehand and the planning algorithms must enable the robot to adapt to these changes on the fly. Real-time planners can be categorized by the way that they deal with changes in the environment, or how they update a robot's motion when an initial path cannot be followed.

A. D* Re-planning

D* [21] is an iterative graph-search alogrithm that incorporates dynamic edge costs into the planning in order to update a robot's path at real-time. Similar to A* search, it is an optimal and complete algorithm. As the robot moves on the optimal path towards the goal, sensing may find that something obstructs the path so much that the robot cannot continue on the path. When this happens, D* updates the associated nodes and propagates necessary changes to the node's neighbors.

The algorithm plans from the goal node to the start node. Similar to other graph-search algorithms, it maintains a sorted list that contains nodes to expand called the OPEN list. Initially, no nodes in the graph have any costs. The goal node is pushed onto the OPEN list with a cost of 0. As nodes are expanded, the cost of its neighbors are computed before they are entered into the OPEN list. Once the start node is expanded, the shortest path is computed and given to the robot to follow.

A robot may not be able to follow the initial path found by D* for numerous reasons, such as a door being closed, an obstacle obstructing the path, etc. In this case, D* will update the point in the path that the robot cannot reach, then update the point's neighbors, and re-compute a path to the goal. Generally, the node that the robot cannot reach is set to be obstacle space so it's path cost is dramatically increased. Then, the neighboring nodes are updated by changing the transition costs of these nodes, and they are inserted into the OPEN list. If one of the neighbors is popped off the OPEN list, the actual cost of the node is computed, and then the node is re-inserted into the OPEN list with the cost that includes the updates based on the environment changes. After this propagation of cost changes has been completed, a new path from the robot's current node to the goal node is computed and passed to the robot to follow.

The process of modifying costs at real-time and re-computing a path is faster than applying A* (or other graph-search algorithm) from scratch. This re-planning approach was initially proposed for moving a robot in a partially known environment. It was extended to work in the prescense of moving obstacles in [22]. The approach inspired a similar re-planning algorithm called D*-lite [23] that generated behavior as if it were D*, but the algorithm itself is simpler.

Some issues with this approach are that 1) it's unclear what the robot should do while re-planning, 2) D* has only been shown to work with mobile bases, and 3) the discretization of the environment can siginficantly affect the performance of this approach. For the first issue, if the robot is meant to stop and not move while re-planning, then it is very vulnerable to collision while stopped. The second issue implies that this

approach may not work for manipulators because it can take too long to re-plan in a high dimensional space to avoid obstacles at real-time. The third issue warrants a discussion on the trade-off between accuracy and efficiency since a more fine grid will take a longer time to plan on.

B. Elastic Strips

Elastic Strips [24] is an approach to real-time motion planning in unstructured dynamic environments that leverages potential fields and a decomposition-based path-planning strategy [25] to produce motion that allows a robot to avoid obstacles while also moving towards a goal.

An elastic strip is a volume formed by the union of a set of homotopic paths. These paths are slight modifications to a candidate path that satisfies global constraints of a robot's task, such as moving to a goal state. The volume, also called a *tunnel*, is obtained by the decomposition-based strategy in [25]. The volume is "elastic" because it can be slightly modified by modifying one of the homotopic paths while maintaining topological properties of the candidate path and local constraints, such as obstacle avoidance.

1) Obstacle Avoidance: The Elastic Strips framework employs local potential fields to modify trajectories in response to changes in the environment. The potential fields act on the future points along the robot's path.

A path is represented as a discrete set of configurations. Let P_c be the candidate path for a robot. To avoid obstacles, a new candidate path, P_c', must be formed by modifying P_c such that P_c' lies within the elastic tunnel. To modify a path, two work space forces are created from two potential fields and applied to the path's configurations. The two forces are $V_{external}$ and $V_{internal}$. The external force is based on the robot's proximity to obstacles. For a point p on the robot, V_{ext} is defined as

$$V_{ext} = \begin{cases} \frac{1}{2}k_r(d_0 - d(p))^2 & \text{if } d(p) < d_0 \\ 0 & \text{otherwise} \end{cases} \quad (14)$$

where $d(p)$ is the distance between p and the closet obstacle, d_0 is the region of influence around obstacles, and k_r is a scalar. The resulting force from V_{ext} is

$$\mathbf{F}_p^{ext} = -\nabla V_{ext} = k_r(d_0 - d(p))\frac{d}{||d||} \quad (15)$$

where d is the vector between p and the closest point on an obstacle. This repulsive force, $\mathbf{F}_p^{rep}$, will modify the existing configurations by pushing them away from obstacles. If the robot is not close to an obstacle, then the repulsive force has no effect.

If an obstacle moves away from the robot after applying a repulsive force, then the robot's path should contract in order to make the path shorter. This is achieved by an internal force generated by virtual springs at each of the robot's control points along a path. This force is meant to bring each configuration closer to its adjacent configurations on a path. The internal force acting at control point p_j^i on the jth link at configuration q_i connected to configurations q_{i-1} and q_{i+1} is defined as

$$\mathbf{F}_{i,j}^{int} = k_c\left(\frac{d_j^{i-1}}{d_j^{i-1} + d_j^i}(p_j^{i+1} - p_j^{i-1}) - (p_j^i - p_j^{i-1})\right) \quad (16)$$

There are two scenarios that result in a failure to find a new path. One scenario is when a topological change occurs in the robot's configuration free space. Because the modification method maintains the topological properties of the candidate path, there will not be a successful modification if a topological change occurs.

The second scenario is when a structural local minimum occurs. A structural local minima is when the configuration free space around the candidate path has become so narrow that it cannot be modified enough to avoid an obstacle. Note that this is not a local minima in the workspace of the robot. A separate planner can be employed to move through the narrow passageway.

2) Discussion: The Elastic Strips method was extended to the Elastic Roadmaps method [26] in order to further reduce the necessary C-space information needed. The Elastic Roadmaps method utilizes the OBPRM method to create milestones for a robot's path, and these milestones are labelled as task-consistent, valid, or invalid based on the robot's ability to execute the path in the current environment. This approach has been shown to successfully plan a mobile manipulator with task constraints in the presence of unpredictable moving obstacles. However, it still may suffer issues moving a robot between milestones, or not be able to find a path due to the incompleteness of OBPRM.

Using potential fields as controllers as an approach to generate motion while satisfying other constraints has inspired much work for general robot motion planning. These approaches, combined with the operational space [27], have been most realized in planning for highly constrained motion planning, such as humanoid models that need to move toward a goal while maintaining balance and posture. These problems, however, do not generally have strict real-time requirements.

C. Real-time Adaptive Motion Planning (RAMP)

One of the key difficulties in robot motion planning is planning in high-dimensional state spaces, such as mobile manipulator models, at real-time. Sampling-based methods are able to find a path in large state spaces, but they can be far from optimal and re-building roadmaps can be too expensive to avoid obstacles at real-time. The Real-time Adaptive Motion Planning (RAMP) [28] framework utilizes evolutionary computation for real-time planning and execution of high-DOF robots in dynamic environments.

RAMP is a framework that utilizes evolutionary computation for real-time planning and execution in dynamic environments. Evolutionary computation is a field designed to leverage sophisticated data structures with genetic algorithms to rapidly search through a solution space [29]. They are useful for motion planning because an evolutionary

computation approach to planning allows us to drastically change a robot's trajectory with simple, fast operators and incorporate optimization into our state spaces searches. Using evolutionary computation for path planning was first proposed in [30].

1) Overview: A set of trajectories, called a *population*, is maintained throughout the entire run time. Each trajectory in the population is a chromosome. At each *generation*, 1-2 randomly selected chromosomes are modified via modification operators. The new chromosome(s) may replace other trajectories in the population.

The framework interweaves three separate procedures called planning cycles, control cycles, and sensing cycles. At each planning cycle (or *generation*), the population of trajectories is modified by a modification operator. Control cycles are procedures that select a trajectory from the population to be executed by the robot and update the starting motion state of each trajectory in the population. Sensing cycles are procedures that update information about the environment.

2) Modification: The basic modification operators are described in [30]. They are *Insert*, *Delete*, *Swap*, *Change*, and *Crossover*. These modification operations are computationally inexpensive so it is possible to perform many of them within a small amount of time. The operators can result in drastic changes to a path. This means that a new feasible path may be found very quickly, which facilitates real-time path planning that can react to changes in an environment. Modification operators can be added, removed, or replaced based on the application of the robot. For instance, a *Stop* operator was introduced in [28] that stops the base of a mobile manipulator for a random amount of time. This operator promoted decoupling of the base and manipulator, which was desirable to exploit redundancy.

3) Evaluation: Evolutionary computation utilizes an evaluation function to select the best chromosome from a population. In the RAMP framework, this corresponds to evaluating the population to choose the best trajectory for a robot to execute. In RAMP, feasible trajectories are evaluated differently than infeasible trajectories. A trajectory's feasibility is determined by collision with obstacles and if all constraints can be satisfied when switching to the trajectory.

A feasible trajectory is evaluated by a weighted, normalized sum based on optimization criteria:

$$Cost = \sum_{i=1}^{N} C_i \frac{V_i}{\alpha_i} \tag{17}$$

where C_i is a weight, V_i is a value for an optimization criterion, and α_i is a normalization value. Common criteria are time costs, energy costs, orientation change, and manipulability costs.

Infeasible trajectories are trajectories that are in collision or cannot be executed due to not satisfying constraints. Rather than optimization criteria, they are evaluated based on penalties. For a mobile robot system, the following would be an evaluation function for infeasible trajectories:

$$Cost = P_T + P_\theta \tag{18}$$

$$P_T = \frac{Q_T}{T_{coll}} \tag{19}$$

$$P_\theta = Q_\theta \frac{\Delta\theta}{M} \tag{20}$$

where Q_T and Q are large constant values, T_{coll} is the time of the first collision in the trajectory, $\Delta\theta$ is the orientation change, and M is a normalization term.

The evaluation for infeasible trajectories should be designed to differentiate which infeasible trajectories are better than others. For instance, the term P_T ensures that trajectories with collision much later in time will be evaluated as better than trajectories with collision occurring very soon.

4) Sensing: Sensing cycles update the latest environment changes for the evaluation function. For dynamic obstacles, simple trajectories are predicted to use for collision detection in CT-space. An obstacle trajectory is predicted by assuming the sensed velocity is constant. Therefore, the predicted trajectory will be either a straight-line or circular arc. These are used for collision detection when evaluating the population. The obstacles may not follow what RAMP predicts, but future changes will be captured in future sensing cycles. Since sensing cycles occur frequently, using simple trajectories is sufficient for navigation, as shown through various experiments.

5) Discussion: RAMP excels when planning with high-DOF robots, such as mobile manipulators. A video of results shown in [28] has several mobile manipulators being planned independently (using each other robot as dynamic obstacles) to work in a task environment.

While RAMP was shown in [28] to perform well with mobile manipulators, it only considered mobile bases that were capable of holonomic motion. The framework was extended to work with non-holonomic bases in [31]. This work modified the original RAMP approach by converting holonomic trajectory segments to non-holonomic segments at real-time for the robot to execute. A method to switch trajectories with smooth curves was presented, and the control cycles were made to adapt to the robot's dynamics, rather than being fixed.

RAMP has also been applied to continuum manipulator robot models [32] and to the pursuit-evasion domain [33].

IX. Collision Detection

Collision detection is the problem of determining whether one or more objects in a virtual environment intersect. Collision detection is the most computationally expensive task in sampling-based robot motion planning because it has to be performed with respect to each sample to determine whether or not a point lies in C_{free} or C_{obs}. This is made even more time consuming by the complexity of the obstacles and the number of obstacles in an environment.

A. Convex Shapes

Convex polygons are the simplest form of objects to check for collision. There are two main approaches to checking collision between such objects - the GJK algorithm and the Lin-Canny algorithm.

The Gilbert-Johnston-Keerthi (GJK) algorithm [34] is an approach for collision detection between two convex polygons or polyhedra. Let A and B be two nonequal sets of vertices for convex polygons. The GJK algorithm finds a Minkowski sum of the two sets, $C = A - B = \{p_1 - p_2 | p_1 \in A \text{ and } p_2 \in B\}$. If the origin is contained in C, then the two objects A and B are in collision.

It is often useful to know the distance between two objects to maintain a certain distance from obstacles. The GJK algorithm provides a simple method to compute this. Let C be the Minkowski Sum between two convex polygons. Let q denote the closest point in C to the origin. The length of the vector $\vec{\mathbf{q}}$ from the origin to q is equal to the distance between the closest pair of points on A and B.

It is important to note that the entire Minkowski Sum does not need to be computed. An alternative is to iteratively build a polygon inside the Minkowski Sum by connecting a subset of its vertices and checking to see if the origin is contained in any polygons built this way.

The Lin-Canny algorithm [35] is another approach for collision detection between two convex objects. This approach finds the closest pair of features (vertices, edges, or faces) on two objects, determines if collision exists, and uses past knowledge to speed up future queries.

Given an initial feature pair (f_a, f_b) (chosen randomly or manually specified), the closest points on these features, (p_a, p_b), must be computed. If these points lie in the voroni space of the other object's feature, then f_a and f_b are the two closest features. Otherwise, a new set of features is found and the process of finding and checking the closest points repeats. Once the closest feature pair has been found, if the distance between the features is less than some distance threshold, then collision exists.

If the objects move as time goes on, motion history can exploited by using the previous closest feature pair each time the Lin-Canny algorithm begins. The closest feature pair is unlikely to change unless the time between collision detection queries is very large. Therefore, using the previous closest pair will drastically reduce the amount of time spent on the query.

These algorithms can be extended to work with concave objects simply by decomposing a concave object into multiple convex objects. However, real-life objects can be extremely complicated, e.g. containing over 1 million triangles. Therefore, many approaches use a bounding volume hierarchy to represent objects.

B. Complex Objects

Bounding Volumes (BV) are convex polyhedra that encompass a set of smaller polyhedra. A hierarchy is formed by placing a BV over an entire object, then sub-dividing the object into smaller regions, and placing BVs over each smaller region. To perform collision detection, a test on the root BV is performed. If there's no collision, the computation stops. Otherwise, collision detection is performed on the BV at the next level of the hierarchy. This process continues until either 1) no collision is found at any level or 2) collision is found at the lowest levels of the hierarchies. The motivating idea behind this is that collision detection queries can be answered extremely quickly when objects are significantly far apart.

There are several types of Bounding Volumes one can use to approximate a 3D object. Sphere Trees [36] use spheres to bound pieces of the objects. Sphere Trees are nice because collision can be detected between two spheres extremely quickly. Axis Aligned Bounding Boxes (AABB) [37] are cubes that surround a shape and the edges of the cube are parallel to the world coordinate system's axes. Oriented-Bounding-Box (OBB) Trees [38] are another commonly used representation for hierarchical object approximation that are similar to AABBs, but the edges of the cube are parallel to the object's orientation.

C. Discussion

Collision Detection has seen significant work for the field of haptics, which requires collision detection queries to be performed at 1KHz. While contemporary work is mostly aimed at haptics, motion planning also benefits from such advances in collision detection. Two examples of such work are [39] and [40].

There are still challenges for collision detection in mobile robots as well. Mobile robots tend to follow trajectories that are curves, such as B-splines or clothoids. Even though the curves are only planar, it is still a challenging problem to compute exact collision between such curves in a practical amount of time. Most collision detection queries on higher-order curves are performed as approximations for performance reasons, such as in [41].

X. Conclusions

One of the biggest open problems in motion planning is the integration of sensing uncertainty into motion planning algorithms. Currently, a great deal of motion planning research assumes that good sensing is available to a robot and that changes in the environment can be found quickly and accurately. However, this is often not the case. All sensors contain a significant deal of uncertainty, and can take significant time to process. These issues can be detrimental to a robot if not accounted for while planning motion.

References

[1] Tomás Lozano-Pérez and Michael A Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979.

[2] Tomas Lozano-Perez. Automatic planning of manipulator transfer movements. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(10):681–698, 1981.

[3] Tomas Lozano-Perez. Spatial planning: A configuration space approach. *Computers, IEEE Transactions on*, 100(2):108–120, 1983.

[4] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research (IJRR)*, 5(1):90–98, 1986.

[5] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[6] S. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, C.S. Dept., Iowa State Univ., 1998.

[7] Steven M LaValle and James J Kuffner. Randomized kinodynamic planning. *IJRR*, 20(5):378–400, 2001.

[8] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[9] Nancy M. Amato, O. Burchan Bayazit, Lucia K. Dale, Christopher Jones, and Daniel Vallejo. Obprm: An obstacle-based prm for 3d workspaces, 1998.

[10] Steven A Wilmarth, Nancy M Amato, and Peter F Stiller. Maprm: A probabilistic roadmap planner with sampling on the medial axis of the free space. In *ICRA 1999*, volume 2, pages 1024–1031.

[11] Valérie Boor, Mark H Overmars, and A Frank van der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In *ICRA 1999*, volume 2, pages 1018–1023.

[12] Carole Nissoux, Thierry Siméon, and J-P Laumond. Visibility based probabilistic roadmaps. In *Intelligent Robots and Systems (IROS), 1999. Proceedings. IEEE/RSJ International Conference on*, volume 3, pages 1316–1321.

[13] Robert Bohlin and Lydia E Kavraki. Path planning using lazy prm. In *ICRA 2000*, volume 1, pages 521–528. IEEE, 2000.

[14] James Bruce and Manuela Veloso. Real-time randomized path planning for robot navigation. In *IROS 2002*, volume 3, pages 2383–2388. IEEE, 2002.

[15] Anna Yershova, Léonard Jaillet, Thierry Siméon, and Steven M LaValle. Dynamic-domain rrts: Efficient exploration by controlling the sampling domain. In *ICRA 2005*, pages 3856–3861. IEEE, 2005.

[16] Xinyu Tang, Jyh-Ming Lien, NM Amato, et al. An obstacle-based rapidly-exploring random tree. In *ICRA 2006*, pages 895–900. IEEE, 2006.

[17] Liangjun Zhang and Dinesh Manocha. An efficient retraction-based rrt planner. In *ICRA 2008*, pages 3743–3750. IEEE, 2008.

[18] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[19] Bruce Donald, Patrick Xavier, John Canny, and John Reif. Kinodynamic motion planning. *Journal of the ACM (JACM)*, 40(5):1048–1066, 1993.

[20] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. *The International Journal of Robotics Research*, 21(3):233–255, 2002.

[21] Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *ICRA 1994*, pages 3310–3317. IEEE, 1994.

[22] Anthony Stentz et al. The focussed dˆ* algorithm for real-time replanning. In *International Joint Conference on Artifical Intelligence*, volume 95, pages 1652–1659, 1995.

[23] Sven Koenig and Maxim Likhachev. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, 21(3):354–363, 2005.

[24] Oliver Brock and Oussama Khatib. Elastic strips: A framework for motion generation in human environments. *The International Journal of Robotics Research*, 21(12):1031–1052, 2002.

[25] Oliver Brock and EE Kavraki. Decomposition-based motion planning: A framework for real-time motion planning in high-dimensional configuration spaces. In *ICRA 2001*, volume 2, pages 1469–1474. IEEE, 2001.

[26] Y. Yang and O. Brock. Elastic roadmapsmotion generation for autonomous mobile manipulation. *Autonomous Robots*, 28(1):113–130, 2010.

[27] Oussama Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Journal on Robotics and Automation*, 3(1):43–53, 1987.

[28] J. Vannoy and J. Xiao. Real-time adaptive motion planning (ramp) of mobile manipulators in dynamic environments with unforeseen changes. *IEEE Transactions on Robotics*, 24(5):1199–1212, 2008.

[29] Z. Michalewicz. *Genetic algorithms+ data structures= evolution programs*. Springer Science & Business Media, 2013.

[30] J. Xiao, Z. Michalewicz, L. Zhang, and K. Trojanowski. Adaptive evolutionary planner/navigator for mobile robots. *IEEE Transactions on Evolutionary Computation*, 1(1):18–28, 1997.

[31] Sterling McLeod and Jing Xiao. Real-time adaptive non-holonomic motion planning in unforeseen dynamic environments. In *IROS 2016*. IEEE, 2016.

[32] Jing Xiao and Rayomand Vatcha. Real-time adaptive motion planning for a continuum manipulator. In *IROS 2010*, pages 5919–5926. IEEE, 2010.

[33] Jonathan Annas and Jing Xiao. Intelligent pursuit & evasion in an unknown environment. In *IROS 2009*, pages 4899–4906. IEEE, 2009.

[34] Elmer G Gilbert, Daniel W Johnson, and S Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation*, 4(2):193–203, 1988.

[35] Ming C Lin and John F Canny. A fast algorithm for incremental distance calculation. In *ICRA 1991*, pages 1008–1014. IEEE, 1991.

[36] Philip M Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics (TOG)*, 15(3):179–210, 1996.

[37] Herbert Edelsbrunner and Hermann A. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 13(4):177–181, 1981.

[38] Stefan Gottschalk, Ming C Lin, and Dinesh Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180. ACM, 1996.

[39] Miguel A Otaduy and Ming C Lin. Sensation preserving simplification for haptic rendering. In *ACM SIGGRAPH 2005 Courses*, page 72. ACM, 2005.

[40] Jernej Barbič and Doug L James. Six-dof haptic rendering of contact between geometrically complex reduced deformable models. *IEEE Transactions on Haptics*, 1(1):39–52, 2008.

[41] Knut Mørken, Martin Reimers, and Christian Schulz. Computing intersections of planar spline curves using knot insertion. *Computer Aided Geometric Design*, 26(3):351–366, 2009.

[42] Mike Stilman. Task constrained motion planning in robot joint space. In *IROS 2007*, pages 3074–3081. IEEE, 2007.

[43] Dmitry Berenson, Siddhartha S Srinivasa, and James Kuffner. Task space regions: A framework for pose-constrained manipulation planning. *The International Journal of Robotics Research*, page 0278364910396389, 2011.