Lecture 14
Intro To Sampling-Based Planning Methods (2)

Instructor: Jingjin Yu
Outline (for Next 3-4 Lectures)

Drawbacks of combinatorial motion planning methods

Probabilistic roadmap (PRM) introduction

Components of sampling-based motion planning methods

- Sampling
- $k$-d tree and nearest neighbor search
- Distance metric
- Collision detection

PRM in more detail

A new notion of completeness

Rapidly-exploring random trees (RRT)

When would sampling-based method work well?

Optimality issues
Recall the illustration of the Piano Mover’s Problem

- Modeling of the free configuration space $C_{free}$ can be a daunting task – they have to be represented as semi-algebraic sets
- The associated computation is also prohibitive for even just a few degrees of freedom

To the rescue: sampling based methods – instead of representing $C_{free}$ explicitly and globally, we instead “probe” the space locally, as necessary
Sampling-Based Planning

\[ x \rightarrow I \rightarrow G \]
Key Components of Sampling-Based Planning

Sampling-based planning requires several important subroutines

⇒ An **efficient sampling routine** is needed to generate the samples. These samples should **cover** $C_{free}$ well in order to be effective.

⇒ **Efficient nearest neighbor search** is necessary for quickly building the roadmap: for each sample in $C_{free}$ we must find its $k$-nearest neighbors.

⇒ The neighbor search also requires a **distance metric** to be properly defined so we know the distance between two samples.

⇒ This can be tricky for certain spaces, e.g., $SE(3)$.

⇒ **Collision checking** - Note that $C_{free}$ is not computed explicitly so we actually are checking collisions between a complex robot and a complex environment.
Nearest Neighbor Search

Connecting the samples

⇒ Building the graph requires connecting the samples
⇒ This cannot be done for all pairs of points!
  ⇒ For \( N \) sample points, this requires \( N^2 \) operations
  ⇒ But \( N \) can be very large, e.g., \( > 10^5 \)
  ⇒ For \( N = 10^6 \), \( N^2 = 10^{12} \)
⇒ We have to do it more efficiently!
⇒ This is known as nearest neighbor search

Variants of useful nearest neighbor search

⇒ 1-NN: finding a single nearest neighbor
  ⇒ Can be done with \( k \)-d trees
  ⇒ We will look at this in more detail
⇒ \( k \)-NN: finding \( k \) nearest neighbors
  ⇒ Note the \( k \) here is not the same as the \( k \) in \( k \)-d trees
  ⇒ Can run 1-NN algorithms \( k \) times
\textbf{k-d Tree}

\textit{k-d} tree stands for \textit{k-dimensional trees}

\begin{itemize}
  \item A data structure for storing points in \textit{k} dimensions
  \item Assumes a tree like structure
  \item Useful for finding points w/ certain properties
  \item Can be used for solving 1-NN
\end{itemize}

Construction of a \textit{k-d} tree for \textit{n} points

\begin{itemize}
  \item Pick dimension \textit{i}, pick a point with coordinates \textit{x} = (x_1, \ldots, x_i, \ldots, x_k)
  \item Split the points based on \textit{x}_i (greater or less than)
  \item Repeat the above two steps recursively
    \begin{itemize}
      \item Increase \textit{i} (modulo \textit{k}) each time
      \item I.e., pick a new dimension each time
    \end{itemize}
  \item Depth: \text{log} \textit{n} if balanced
  \item Construction takes \textit{O}(kn \log n) time
    \begin{itemize}
      \item Each dimension needs sorting \textsim{O}(n \log n)
    \end{itemize}
  \item Can speed up to \textit{O}(n \log n)
  \item Balancing is important
\end{itemize}
Nearest Neighbor Search w/ $k$-d Tree

Finding nearest neighbor of a query point $q$

- Basically, traverse the tree, e.g., using BFS
- Maintain a current best candidate $x$
- Also maintain a queue of subtree distances to $q$
- Uses the subtree distances to prioritize search

Example

- Start with root $(0.42, 0.42)$
  - $x = (0.42, 0.42)$, both left and right subtrees are active
- Examine $(0.07, 0.55)$
  - Three trees on the queue afterward
- Examine $(0.82, 0.3)$, update $x = (0.82, 0.3)$
  - Truncate the left subtrees
  - Two subtrees left
- Examine $(0.72, 0.68)$, update $x = (0.72, 0.68)$
  - We are done since the last subtree is further from $q$ than $x$
Performance of $k$-d Tree and Generalization

General performance of balanced $k$-d tree

- Construction: $O(n \log n)$ w/ $O(n)$ median computation
- Construction through presorting the points: $O(kn \log n)$
- Inserting/deletion of a new point: $O(\log n)$
- Nearest neighbor search: $O(\log n)$ for randomly distributed points

$k$-d trees can be used for $k$-NN ($k$ nearest neighbor search) as well

- Naïve implementation: simply run 1-NN $k$ times
- This yields $O(k \log n)$ running time
- Improvement
  - Keep up to $k$ candidates
  - Only discard a subtree if worse than all $k$ candidates

Applications of $k$-NN

- Widely used in classification tasks, e.g.,
  - Optical character recognition (OCR)
  - Pattern recognition
A Brief Look at the Issue of Distance Metric

Nearest neighbor queries requires a distance metric

⇒ Given two points \(x\) and \(q\), need to know their distance \(d(x, q)\)
⇒ Otherwise, cannot compare!
⇒ This is easy in Euclidean space:
\[
d(x, q) = ||x - q||_2 = \sqrt{\sum(x_i - q_i)^2}
\]
⇒ But what about \(T^2\) or \(\mathbb{R}^2 \times S^1\), or more complex settings?
  ⇒ For \(T^2\), \(\theta_1\) seems to be more important
  ⇒ For \(\mathbb{R}^2 \times S^1\), a small change in \(\theta\) can be hard to make
  ⇒ Sometimes, we can work with the workspace or task space
    ⇒ This however will make the sampling more difficult
⇒ There is no universal solution – requires some creativity
Sampling based methods need to check whether robot is in collision

− Generally, given two sets of points $A$ and $B$, we want to check the distance between them

$$d(A, B) = \min_{a \in A, b \in B} |a - b|$$

− Clearly, $A$ and $B$ intersect (collide) if and only if $d(A, B) = 0$

$d(A, B) > 0$: no collision

$d(A, B) = 0$: collision
Collision checking can be difficult for general objects, e.g., $d(A, B)$ are hard to compute directly!

Often, simpler bounding volumes are used to approximate the shapes

- However, bounding volumes over approximate the shapes
- No collision between bounding volumes $\rightarrow$ no collision between the shapes
- Collision between bounding volumes $\rightarrow$ possible collision
- Need to refine hierarchically if a possible collision is detected
- Such a method is called bounded volume hierarchy (BVH)
Bounded Volume Hierarchy (BVH)

BVH breaks objects into smaller pieces
Which yields a hierarchy, represented as a **tree**

This is carried out incrementally
⇒ Finer hierarchies are created as needed and then saved for later
Starting from the roots and check for collision (how?)
- No collision → done with the branch
- Otherwise, check pairs of children on the trees
- Recursively call the procedure
- Traverse down the tree

How many possible checks in total (say each object has \( n \) pieces)?
- At most \( n^2 \) checks
- Using BVH can save some checks
Types of Bounding Volumes

Many types of bounding volumes are possible

- Spheres are simple and **orientation invariant** but do not fit tightly
- AABBs are even simpler, but not orientation invariant, not tight
- OBBs are orientation invariant, reasonably tight
- Convex hulls are tight and orientation invariant, but require more computation

Figure 5.9: Four different kinds of bounding regions: (a) sphere, (b) axis-aligned bounding box (AABB), (c) oriented bounding box (OBB), and (d) convex hull. Each usually provides a tighter approximation than the previous one but is more expensive to test for overlapping pairs.
Probabilistic Roadmap in More Detail

$C_{free}$, generally high dimensional
Generating Random Samples

Random sample
Rejecting Samples Outside $C_{free}$
Collecting Enough Samples in $C_{free}$
Connect to $k$ Nearest Neighbors (If Possible)
Connect to $k$ Nearest Neighbors (If Possible)
Query Phase

\[ x \]  
\[ I \]  
\[ G \]
Path Smoothing
First proposed by Kavraki et al.

**Algorithm 6 Roadmap Construction Algorithm**

**Input:**
- $n$: number of nodes to put in the roadmap
- $k$: number of closest neighbors to examine for each configuration

**Output:**
- A roadmap $G = (V, E)$

1. $V \leftarrow \emptyset$
2. $E \leftarrow \emptyset$
3. while $|V| < n$ do
    4. repeat
        5. $q \leftarrow$ a random configuration in $Q$
        6. until $q$ is collision-free
        7. $V \leftarrow V \cup \{q\}$
    8. end while
9. for all $q \in V$ do
   10. $N_q \leftarrow$ the $k$ closest neighbors of $q$ chosen from $V$ according to $\text{dist}$
   11. for all $q' \in N_q$ do
        12. if $(q, q') \notin E$ and $\Delta(q, q') \neq \text{NIL}$ then
            13. $E \leftarrow E \cup \{(q, q')\}$
        14. end if
   15. end for
16. end for
Algorithm 7 Solve Query Algorithm

Input:
- \( q_{\text{init}} \): the initial configuration
- \( q_{\text{goal}} \): the goal configuration
- \( k \): the number of closest neighbors to examine for each configuration
- \( G = (V, E) \): the roadmap computed by algorithm 6

Output:
- A path from \( q_{\text{init}} \) to \( q_{\text{goal}} \) or failure

1: \( N_{q_{\text{init}}} \leftarrow \) the \( k \) closest neighbors of \( q_{\text{init}} \) from \( V \) according to \( \text{dist} \)
2: \( N_{q_{\text{goal}}} \leftarrow \) the \( k \) closest neighbors of \( q_{\text{goal}} \) from \( V \) according to \( \text{dist} \)
3: \( V \leftarrow \{q_{\text{init}}\} \cup \{q_{\text{goal}}\} \cup V \)
4: set \( q' \) to be the closest neighbor of \( q_{\text{init}} \) in \( N_{q_{\text{init}}} \)
5: repeat
6: \( \text{if } \Delta(q_{\text{init}}, q') \neq \text{NIL} \text{ then} \)
7: \( E \leftarrow (q_{\text{init}}, q') \cup E \)
8: \( \text{else} \)
9: set \( q' \) to be the next closest neighbor of \( q_{\text{init}} \) in \( N_{q_{\text{init}}} \)
10: \( \text{end if} \)
11: until a connection was successful or the set \( N_{q_{\text{init}}} \) is empty
12: set \( q' \) to be the closest neighbor of \( q_{\text{goal}} \) in \( N_{q_{\text{goal}}} \)
13: repeat
14: \( \text{if } \Delta(q_{\text{goal}}, q') \neq \text{NIL} \text{ then} \)
15: \( E \leftarrow (q_{\text{goal}}, q') \cup E \)
16: \( \text{else} \)
17: set \( q' \) to be the next closest neighbor of \( q_{\text{goal}} \) in \( N_{q_{\text{goal}}} \)
18: \( \text{end if} \)
19: until a connection was successful or the set \( N_{q_{\text{goal}}} \) is empty
20: \( P \leftarrow \) shortest path\((q_{\text{init}}, q_{\text{goal}}, G)\)
21: \( \text{if } P \text{ is not empty then} \)
22: \( \text{return } P \)
23: \( \text{else} \)
24: \( \text{return failure} \)
25: \( \text{end if} \)
A Look at Completeness

Sampling-based algorithms are no longer **complete**!

- If a solution exists, it will eventually find one and stop
- When there is no solution, the algorithm may keep running forever (so we need to have a timeout for these methods)

We need a new notion of completeness
A New Notion of Completeness

Define a new notion of completeness based on **denseness** of sampling

⇒ A set of samples is **dense** if dispersion $\delta(P) \to 0$ as $|P| \to \infty$

⇒ This means that the roadmap will get into any opening

⇒ But it is hard to predict when if we do not know how big is the opening

**Resolution completeness**

⇒ For deterministic sampling (e.g., using a Halton sequence)

⇒ An algorithm is **resolution complete** if it samples deterministically and densely

**Probabilistic completeness**

⇒ For probabilistic methods

⇒ An algorithm is **probabilistic complete** if it samples probabilistically, e.g., uniformly random, and densely