Homework 5c - EST (Matlab version)

ME570 - Prof. Tron

2023-11-30

In this homework, you will implement the simplest variation of EST, one of the sampling-based methods seen in class. The main effort for its implementation is in the sampling functions (since it builds a tree, there is no need for a separate graph search). If the assignment mentions a file not specifically provided, it refers to a file from a previous assignment (in fact, you will need functions from Homework 2, 3 and 4).

Problem 1: Sampling tree

This problem considers a random *tree* for solving a single path query. One advantage of using a tree instead of a general graph is that you will not need to call `graph_search`( ), as the backpointers in `graphVector` can be set while constructing the tree.

We will use the sphere world from Homework 3 for testing. Please refer to that homework for a detailed description of the `world` data structure.

We will use the sphere world from Homework 3 for testing. Please refer to that homework for a detailed description of the `world` data structure.

Question provided 1.1. This function implements a collision check against all spheres in the world.

> [`flag`]=`sphereworld_isCollision`(`world,x`)
>
> Description: Checks if the points with coordinates `x` is inside any of the spheres in `world`. This is done by iterating over all spheres, using the function `sphere_distance`( ) from Homework 3 on each one of them, and then combining the results.
>
> Input arguments
>
> - `world` (dim. [`NSpheres` × 1], type `struct array`): see definition in Homework 3.
> - `x` (dim. [2 × `NPoints`]): array with the coordinates of the sampled locations (one for each column) that need to be tested for collisions.
>
> Output arguments
>
> - `flag` (dim. [1 × `NPoints`], type `logical`): `flag(iPoint)` is `true` if `x(:,iPoint)` is inside one of the spheres (i.e., it is a collision), `false` otherwise.

Question provided 1.2. Create a function that samples from the free configuration space using rejection sampling.

[`xSample`]= `sphereworld_sample` (`world,...`)

Input arguments

- `world` (dim. [NSpheres × 1], type `struct array`): see Homework 3.

Optional arguments

- `'distribution',distribution` : Selects the type of distribution to be used. Possible values for `distribution` are 'uniform' and 'Gaussian'.
- `'size',sz` : Size parameter for the distribution (default value: `sz=10`). For the `'uniform'` distribution, the resulting domain of the random variable is the square [− `maxSize` , `maxSize`]^2. For the `'Gaussian'` distribution, `sz` represents the variance.
- `'mean',mu` (dim. [2 × 1]): Mean of the distribution (default value: `mu=`$\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$).

Output arguments

- `xSample` (dim. [2 × 1]): a sample in the free space. Can be empty if a sample cannot be generated in `nbTrials=1000` trials.

Description: Generate samples using the specified distribution, until `sphereworld_isCollision` ( ) returns `false` for a sample, or a maximum number of trials `nbTrials` is reached.

Question **code** **1.1.** This function samples locations along a line between two samples, and checks if this line can be used to create an edge in the roadmap.

[`flag`]= `prm_localPlannerIsCollision` (`world,xEdge1,xEdge2,maxDistEdgeCheck`)

Description: Generates `NPoints` equispaced points on the line between `xEdge1` and `xEdge2` , such that the maximum distance between two consecutive samples is less than `maxDistEdgeCheck` . Note that the value of `NPoints` is determined by the distance between the two points and `maxDistEdgeCheck` .

Input arguments

- `xEdge1` (dim. [2 × 1]), `xEdge2` (dim. [2 × 1]): coordinates of the endpoints of the line on which to sample the locations.
- `maxDistEdgeCheck` (dim. [1 × 1]): maximum distance between sampled locations.

Output arguments

- `flag` (type `logical`): `true` if *any* of the sampled locations is in collision, `false` otherwise.

Requirements: This function returns an "all-or-nothing" result, meaning that the edge between `xEdge1` and `xEdge2` even if only one of the samples is in collision.

Question **provided** **1.3.** Implement a function that can sample from any arbitrary discrete probability distribution.

[`x`]= `rand_discrete` (`xDistr,fDistr`)

Description: Returns a sample `x` of a random variable X that can assume values in `xDistr` with probability given by `fDistr` (that is, $p($ `x` $=$ `xDistr(i)` $) =$ `fDistr(i)`).

Input arguments

- `xDistr` (dim. [`NValues` $\times$ `1`]): set of values that X can assume.
- `fDistr` (dim. [`NValues` $\times$ `1`]): array describing the probability of each value in `xDistr`. Note: the function normalizes `fDistr` so that it sums to one.

Output arguments

- `x`: a sample from the distribution. Internally, the function uses `rand` ( ) and the cumulative distribution of X to generate `x`.

Question provided 1.4. A utility function for graphs.

File name: `me570_graph.py`
Class name: `Graph`
Description: See description in Homework 4.

[`deg`]= `graph_degree` (`graphVector`)
Description: The function returns an array with the degree deg(n) of each vertex n in the graph `graphVector`.

Input arguments

- `graphVector` (dim. [`NNodes` $\times$ `1`], type `struct`): the structure describing a graph, as specified in Homework 4.

Output arguments

- `deg` (dim. [`NNodes` $\times$ `1`]): the number of neighbors for each element in `graphVector`.

Question code 1.2. This function performs connection sampling to generate candidates for growing the tree.

[`graphVector,xSample`]= `sampleTree_extend` (`graphVector,radius,world`)
Description: The function performs the following steps:

1) Uses `graph_degree` ( ) to obtain the array of degrees `deg` of the nodes in the tree.

2) Computes an array `fDistr` such that `fDistr(i)` is proportional to `1/(deg(i)+1)`.

3) Use `rand_discrete` ( ) to sample a node with index `idxSample` according to `fDistr`.

4) Use `sphereworld_sample` ( ) with the optional arguments to generate a sample location `xSample` around the location of `idxSample` according to a Gaussian distribution with variance `radius`.

5) Use `graph_nearestNeighbors` ( ) (with `kNeighbors=1`) to find the index `idxNear` of the single closest neighbor of `xSample`.

6) Use `prm_localPlannerIsCollision`( ) to check whether `xSample` can be connected to the closest neighbor. If there is no collision, add `xSample` to `graphVector`, setting its backpointer to `idxNear`, and updating the `neighbor` and `neighborCost` fields of `graphVector[idxNear]`.

7) Repeat the steps above from *3)* until a sample is successfully added, or up to 100 times.

Input arguments

- `graphVector` (dim. [NNodes × 1], type `struct`): the structure describing the tree to extend, as specified in Homework 4.
- `radius`: variance of the sampling around a given node.
- `world` (dim. [NSpheres × 1], type `struct array`): see definition in Homework 3. This is needed to be able to check for collisions.

Output arguments

- `graphVector` (dim. [NNodes+1 × 1], type `struct`): same as the corresponding input argument, but with an additional node (unless the extension fails).
- `xSample` (dim. [2 × 1]): the location of the additional node (empty if the extension fails).

Note that, in Matlab, concatenating struct arrays (e.g., `[a;b]`) might fail even if the two arrays (`a` and `b`) have the same fields. A reliable way of achieving this is to extend the first array (`a`), and then copy the data from the second (`b`) on field at a time; this is implemented with the provided function `struct_cat`( ).

Question optional 1.1. The function `sampleTree_extend`( ) as described above implements a variant of EST. Add an optional argument to this function to transform the algorithm into RRT, RRT with greedy search, or RRT with greedy search and sampling biased toward the goal.

Question code 1.3. In this question you will build around the function `sampleTree_extend`( ) to generate a complete planner.

[`xPath`,`graphVector`]=`sampleTree_search`(`world`,`xStart`,`xEnd`,`goalDistThreshold`)

Description: Implement a tree-based sampling search algorithm. The function performs the following steps:

1) Initializes a `graphVector` containing only a vertex at `xStart`.

2) Call `sampleTree_extend`( ) to grow the tree with a new sample with location `xSample`.

3) If the distance between `xSample` and `xEnd` is less then `goalDistThreshold`, call `prm_localPlannerIsCollision`( ) to try to connect the goal to the tree:

(a) If the connection does *not* succeed, repeat from step *2)*, up to `NTrials=1000` times.

(b) If the connection succeeds, add `xEnd` to `graphVector` (with the correct backpointer), and then use `graph_path`( ) to return the path from `xStart` to `xEnd`.

Input arguments

- `world` (dim. [NSpheres $\times$ 1], type `struct array`): see definition in Homework 3.
- `xStart` (dim. [2 $\times$ 1]), `xEnd` (dim. [2 $\times$ 1]): the coordinates of the initial and final locations.
- `goalDistThreshold`: threshold for trying to connect the goal to a newly generated extension of the tree. If too large, the planner will spend too much time checking if the goal can be reached from the tree. If too small, the planner will be unlikely to generate a sample close enough to the goal to connect it.

Output arguments

- `xPath` (dim. [2 $\times$ NPath]): array where each column contains the coordinates of the points of the path found from `idxStart` to `idxEnd`. If the planner fails, it returns an empty array.
- `graphVector` (dim. [NNodes $\times$ 1], type `struct`): the structure describing the tree built by the algorithm, as specified in Homework 4.

Question **report** **1.1.** In this section you will test your planner.

`sampleTree_search_test`( )

Description:

1) Loads the variables `world`, `xStart` and `xEnd` from the file `sphereWorld.mat`.

2) For each starting location (column) in `xStart`, and each goal location (column) in `xGoal`, calls `sampleTree_search`( ) to find a path.

3) Calls `sphereworld_plot`( ) and then overimposes all the trajectories from the start locations to the goal location.

Start with `goalDistThreshold` and `radius` both around 2, and then try to change them until you are satisfied with the performance. In the report, include a table with the values you tried, and images from a few sets of values that you consider significant.

Question **report** **1.2.** Considering the theory we discussed in class, give at least two meaningful comparisons of the differences between the results you have seen in this homework and the previous homework for the sphere world (i.e., w.r.t. potential-based methods and `A`*); for instance, imagine what you would say if somebody asked you which method to use for a particular version of the problem.

Question **optional** **1.2.** Modify and run `sampleTree_search_test`( ) so that they run the respective planners multiple times on the same data, and show how the variance of the length of the paths found across different trials.

Hint for question code 1.1: To generate the samples, you can use the function `line_linspace`( ) from Homework 2, with `a=x2-x1` and `b=x1` (what should you use for `tMin` and `tMax`?)

Hint for question code 1.3: You can use the function `graph_plot`( ) to plot the tree graph and check that the backpointers are correct.

Hint for question report 1.1: If your planner cannot reliably find a path, or takes very long, there are typically two main causes. First, if the graph contains a lot of samples, but still does not get anywhere near the goal, increase either `goal_dist_sthreshold` or `radius`. Second, if the function `sampleTree_extend`( ) frequently fails, then decrease `radius`.

"In theory, there is no difference between theory and practice. But in practice, there is."
— *Benjamin Brewster via goodreads.com*