Homework 4 (Python version)

ME570 - Prof. Tron

2023-11-06

Graph data structure and utilities

Both problems in this homework represent the configuration space as a graph. In practical terms, the graph will be represented by a list of dictionaries `graph_vector`, where each element of the array is a dictionary with fields:

- `neighbors` (dim. [`nb_neighbors` $\times$ 1]): an array containing the indexes (in `graph_vector`) of the vertices that are adjacent to the current one.
- `neighbors_cost` (dim. [`nb_neighbors` $\times$ 1]): an array, with the same dimension as the field `neighbors`, containing the cost to move to each neighbor.
- `g` (`float` or `0.`): scalar variable to store the cost from the starting location along the path through the backpointer. Use a default value of `0.` if the backpointer is not set.
- `backpointer` (`float` or `None`): index of the previous vertex in the current path from the starting location. Use a default value of `None` if the backpointer is not set.
- `x` (dim. [2 $\times$ 1]): the physical (x, y) coordinates of the vertex.

Note that, in the above, the dimension `nb_neighbors` is in general different for each element in `graph_vector`. The graph is defined by the fields `x`, `neighbors`, `neighbors_cost`; the fields `g` and `backpointer` will be added and used by the graph search algorithm, which will modify these fields while leaving the others constant. **Note about the autograder:** The autograder will assume that *all* the keys are always included for every node (including the `g` and `backpointer` keys), even if the backpointer has not been set (use the default values in that case)

To help you with the homework, the assignment includes a number of utilities.

> Class name: `Graph`
> File name: `me570_graph.py`
> Description: A class collecting a `graph_vector` data structure and all the functions that operate on a graph.

Method name: `__init__`
Description: Stores the arguments as internal attributes.
Input arguments

- `graph_vector` (dim. [`nb_nodes` $\times$ 1], type `dictionary list`): A list of dictionaries as described above.

Since most of the functions that you will implement are methods of the class `Graph`, when the assignment refers to `graph_vector`, this corresponds to the internal attribute in the same class.

Question provided 0.1. The first utility is a function to plot the graph.

Class name: `Graph`
File name: `me570_graph.py`

Method name: `plot`
Description: The function plots the contents of the graph described by the `graph_vector` structure, alongside other related, optional data.
Optional arguments

- `flag_edges` (default: `True`): Show the edges.
- `flag_labels` (default: `False`): Show the labels of the vertices.
- `flag_edge_weights` (default: `False`): Show weights for each vertex.
- `flag_backpointers` (default: `False`): Show arrows for the backpointer.
- `flag_backpointers_cost` (default: `False`): Show the value of `'g'` for the corresponding backpointer.
- `idx_goal` (default: `None`): If not `None`, mark the goal node.
- `flag_heuristic` (default: `True`): If this flag is `True` and `idx_goal` is not `None`, show the value of the heuristic h for each node (requires that the `Graph.heuristic`( ) is correctly implemented).
- `node_lists` (default: `None`): A list or list of lists, of nodes to be marked (e.g., start or goal nodes, a list of closed nodes). Different lists are shown with different markers (in the order `'d'`, `'o'`, `'s'`, `'*'`, `'h'`, `'^'`, `'8'`).

Question provided 0.2. The function `graph_test_data_load`( ) described below allows you to load already-made graphs, stored in the variables `graph_vector` and `graph_vector_medium`. Additionally, you can access `graph_vector_solved`, and `graph_vector_medium_solved`, which are the same as `graph_vector`, and `graph_vector_medium`, but with the fields `g` and `backpointer` populated.

File name: `me570_graph.py`
Function name: `graph_test_data_load`
Description: Loads data from the file `graph_test_data.pkl`.
Input arguments

- `variable_name` (type `string`): name of the variable to load. Available names include: `closedMedium`, `graphVector`, `graphVectorMedium`, `graphVectorMedium_solved`, `graphVector_solved`.

Returns arguments

- `graph_vector` (dim. [`nb_nodes` × `1`], type `dictionary list`): A list of dictionaries as described above, **or a list**.

Question provided 0.3. The last provided utility allows you to find the nodes in the graph that are closest to a given point.

Class name: `Graph`
File name: `me570_graph.py`

Method name: `nearest_neighbors`
Description: Returns the k nearest neighbors in the graph for a given point.
Input arguments

- `x_query` (dim. [`2` × `1`]): coordinates of the point of which we need to find the nearest neighbors.
- `k_nearest` (dim. [`1` × `1`]): number of nearest neighbors to find.

Output arguments

- `idx_neighbors` (dim. [`nb_neighbors` × `1`]): indeces in `graph_vector` of the neighbors of `x`. Generally, `nb_neighbors=k`, except when `graph_vector` contains less than k vertices, in which case all vertices are returned.

In this homework, you will mainly use this function with $k = 1$ to find vertices that approximate start and goal locations.

Question provided 0.4. This function should takes as input a discretized world and outputs the corresponding `graph_vector` structure.

File name: `me570_graph.py`
Method name: `grid2graph`
Description: The function returns a `Graph` object described by the inputs. See Figure 1 for an example of the expected inputs and outputs.
Input arguments

- `grid` (type `dictionary`): An object of class `Grid`. The attribute `fun_evalued` should contain a logical array such that `fun_evalued[i,j]` is `true` if there is a cell (i.e., no collision) at the (`xx_grid[i]` , `yy_grid[j]`) location, and `false` otherwise.

Requirements: Note that the fields `xx` and `yy` in `grid` are to be intended as *generalized coordinate* pairs, and their interpretation could be different than x and y coordinates of points in $\mathbb{R}^2$. For instance, in Problem 3 below, which involves the two-link manipulator, they correspond to angles.

Question provided 0.5. This homework includes an updated version of the class `Grid` from Homework 2. The class now includes an internal attribute that contains the last value returned by the method `eval` ( ). In the following questions, you will use this internal attribute to obtain data that was previously generated.

File name: `me570_geometry.py`

Class name: `Grid`

Description: A class to store the coordinates of points on a 2-D grid and evaluate arbitrary functions on those points.

Attribute name: `fun_evalued`

Description: Stores the value that was returned by the last call to the method `eval` ( ). It is initially set to `None` .

Problem 1: Graph search

In this problem you will implement a graph search algorithm, and apply it to a graph obtained from a grid discretization of a free configuration space.

The graph search function you will develop will be generic, because it can work on a `graph_vector` data structure in a way that is somewhat abstract from the actual problem. For instance, the function manipulates nodes in terms of their indexes in the data structure, instead of, say, using their coordinates. In this way, the same function can be applied to different problems (an occupancy graph in this problem, and the two-link manipulator in the next).

You will be required to implement the `A`* algorithm for which the reference pseudo-code is reproduced in Algorithm 1 (this is a essentially the same as the algorithm on page 531 of the book, but includes more details).

Data structures The algorithm uses a priority queue O, and a list of closed edges C. For the priority queue O, you are expected to use the corresponding set of functions from Homework 1. For the list C, you should use a simple array. See Question code 1.5 for further details.

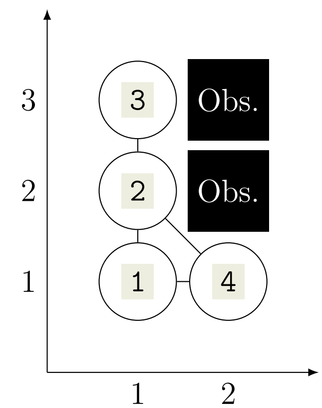
Debugging tips Since `A`* is a somewhat complex algorithm to implement, you should use the provided function `Graph.plot` ( ) and the provided data `graph_testData.mat` to test the individual functions and check that the outputs are consistent with what you would expect. In particular, embedding `Graph.plot` ( ) and pausing the execution at each iteration in the loop of `Graph.search` ( ) during debugging is instructive (but remember to remove it in the final version or, even better, use an optional argument to enable it only when needed).

Question code 1.1.

```
grid.xx=[1 2]
grid.yy=[1 2 3]
grid.F =
```

$$\begin{bmatrix} \texttt{true} & \texttt{true} & \texttt{true} \\ \texttt{true} & \texttt{false} & \texttt{false} \end{bmatrix}$$

(a) The input arguments.



(b) Graphical representation of the graph with 8-neighbors connectivity. The numbers represent the indexes in `graph_vector`. The name `flag_free_space_map` has been abbreviated to `flag` for convenience.

```
neighbors=[4;2]
neighborsCost=[1;1]
x=[1;1]
```

```
neighbors=[1;4;3]
neighborsCost=[1;1.414;1]
x=[1;2]
```

```
neighbors=[2]
neighborsCost=[1]
x=[1;3]
```

```
neighbors=[1;2]
neighborsCost=[1;1.414]
x=[2;1]
```

(c) The output `Graph.graph_vector` (the fields `g` and `backpointer` are not visualized). This particular `graph_vector` can be found in the provided data.

Figure 1: Example of the input and output arguments for `grid2graph`( ).

Class name: `Graph`
File name: `me570_graph.py`

Method name: `heuristic`
Description: Computes the heuristic `h` given by the Euclidean distance between the nodes with indexes `idx_x` and `idx_goal`.
Input arguments

- `idx_x` (type `int`), `idx_goal` (type `int`): indexes of the elements in `graph_vector` to use to compute the heuristic.

Output arguments

- `h_val` (type `float`): the heuristic (Euclidean distance) between the two elements.

Question provided 1.1. The following function demonstrate the use of `Graph.plot`, and visualizes two graphs (already solved with `A*`) from the provided function `graph_load_test_data`( ):

File name: `me570_graph.py`
Method name: `graph_test_data_plot`
Description: Visualizes the contents of the file `graph_testData.mat` using `Graph.plot`( ) and different sets of visualization options.

In particular, the example `graphVectorMedium_solved` corresponds to the `A`* example we covered in class. Note that the functions will work properly only after you completed Question code 1.1.

Question **report** **1.1.** Run the provided function `graph_test_data_plot`( ), and, in your report: *1)* include the two figures with the two solved examples, and *2)* describe the meaning of the arrows, and the numbers labeled with `g`, `h` and `f`.

Algorithm 1 The `A`* algorithm. **Ver. 1** is a simplified, complete but not optimal version. **Ver. 2** is the full optimal version.

Add the starting node n_{start} to O, set $g(n_{start}) = 0$, set the `backpointer` of n_{start} to be empty, initialize C to an empty array. ▷ Initialization
repeat
  Pick n_{best} from O such that $f(n_{best}) \leq f(n_O)$ for all $n_O \in O$. ▷ PriorityMinExtract
  Remove n_{best} from O and add it to C.
Ver. 1
  if $n_{best} = n_{goal}$ **then**
    Return path using backpointers from n_{goal}. ▷ Path
  end if
Ver. 2
  if $g(n_{goal})$ is set **and** $g(n_{goal}) \leq f(n_O)$ for all $n_O \in O$ **then**
    Return path using backpointers from n_{goal}. ▷ Path
  end if
  $S = \text{ExpandList}(n_{best})$
  for all $x \in S$ that are not in C **do** ▷ ExpandElement
    if $x \notin O$ **then**
      Set the backpointer cost $g(x) = g(n_{best}) + c(n_{best}, x)$.
      Set the `backpointer` of x to n_{best}.
      Compute the heuristic $h(x)$. ▷ Heuristic
      Add x to O with value $f(x) = g(x) + h(x)$. ▷ PriorityInsert
    else if $g(n_{best}) + c(n_{best}, x) < g(x)$ **then** ▷ A better path to x has been found
      Update the backpointer cost $g(x) = g(n_{best}) + c(n_{best}, x)$.
      Ver. 2 Update the value of $f(x) = g(x) + h(x)$ in O (might require computing $h(x)$ again). ▷ Heuristic
      Update the `backpointer` of x to n_{best}.
    end if
  end for
until O is empty

Question code **1.2.**

> Class name: `Graph`
> File name: `me570_graph.py`
>
> > Method name: `get_expand_list`
> > Description: Finds the neighbors of element `idx_n_best` that are not in `idx_closed` (line 12 in Algorithm 1).
> > Input arguments
> >
> > - `idx_n_best` (type `int`): the index of the element in `graph_vector` of which we want to find the neighbors.
> > - `idx_closed` (dim. [`nb_closed` $\times$ 1], type `list`): list of indexes containing the list of elements of `graph_vectors` that have been closed (already expanded) during the search.
> >
> > Output arguments
> >
> > - `idx_expand` (dim. [`nb_neighborsnotclosed` $\times$ 1], type `list`): array of indexes of the neighbors of element `idx_n_best` in `graph_vector`

This correspond to Line 11 of Algorithm 1.

Question code **1.3 (2 points).** The function below uses the priority queue from Homework 1.

> Class name: `Graph`
> File name: `me570_graph.py`
>
> > Method name: `expand_element`
> > Description: This function expands the vertex with index `idx_x` (which is a neighbor of the one with index `idx_n_best`) and updates the attribute `graph_vector` and returns an updated version of `pq_open`.
> > Input arguments
> >
> > - `idx_n_best` (type `int`), `idx_x` (type `int`), `idx_goal` (type `int`): indexes in `graph_vector` of the vertex that has been popped from the queue, its neighbor under consideration, and the goal location.
> > - `pq_open` (type `Priority`): object of type `Priority` with the priority queue of the open nodes.
> >
> > Output arguments
> >
> > - `pq_open` (type `Priority`): Same as the homonymous input argument, but updated with the new nodes that have been opened.

This function corresponds to Lines 13–22 in Algorithm 1.

Question code **1.4.** Implement a function that transforms the backpointers describing a path into the actual sequence of coordinates.

Class name: `Graph`
File name: `me570_graph.py`

Method name: `path`
Description: This function follows the backpointers from the node with index `idx_goal` in `graph_vector` to the one with index `idx_start` node, and returns the *coordinates* (not indexes) of the sequence of traversed elements.
Input arguments

- `idx_start` (type `int`), `idx_goal` (type `int`): indexes in `graph_vector` of the starting and end vertices.

Output arguments

- `x_path` (dim. [2 × `nb_path`]): array where each column contains the coordinates of the points obtained with the traversal of the backpointers (in reverse order). Note that, by definition, we should have `x_path[:,0]` equal to `graph_vector[idx_start]['x']` and `x_path[:,-1]` equal to `graph_vector[idx_goal]['x']`.

Question code 1.5. This question puts together the answers to Questions code 1.1–code 1.4.

Class name: `Graph`
File name: `me570_graph.py`

Method name: `search`
Description: Implements the `A`* algorithm, as described by the pseudo-code in Algorithm 1.
Input arguments

- `idx_start` (dim. [1 × 1]), `idx_goal` (dim. [1 × 1]): indexes in `graph_vector` of the starting and end vertices.

Output arguments

- `x_path` (dim. [2 × `nb_path`]): array where each column contains the coordinates of the points of the path found from `idx_start` to `idx_goal`.

Requirements: **optional** Set a maximum limit of iterations in the main `A`* loop on line 2 of Algorithm 1. This will prevent the algorithm from remaining stuck on malformed graphs (e.g., graphs containing a node as a neighbor of itself), or if you make some mistake during development.

This function corresponds to the entire Algorithm 1; you can choose to implement either the lines corresponding to **Ver. 1** (slightly simpler) or to **Ver. 2** (which guarantee optimal paths); the unlabeled lines in black are common to both versions. For the purposes of this homework, you can assume that a path always exists (although this can be optionally generalized in Question optional 1.1).

The function for the cost to use in the priority queue, denoted as $f(n)$ in the book and in Algorithm 1, is $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from node n to the start vertex (going through the backpointer), and $h(n)$ is the heuristic (the Euclidean distance between nodes). The cost $c(n, x)$ between two vertices is the one stored in the `neighbors_cost` field.

Your implementation of `Graph.search`( ) must contain the following elements:

- a priority queue `pq_open` of the *opened* vertices (the structure O in Algorithm 1); this structure must be an object instantiated from the class `Priority` from Homework 1; use the index of the vertex for the *key* and the function $f(n)$ described above for the `cost`.
- a list `idx_closed` (dim. [`nb_closed` $\times$ `1`]), corresponding to the set C in Algorithm 1 containing the indexes of the closed vertices.

Question optional 1.1. Add conditions to return an empty `path` if `A`* cannot find a feasible path.

Question optional 1.2. Add an argument `method` containing a string that determines the behavior of the algorithm. The function $f(n)$ will then depend on the value of the argument `method`:

- $f(n) = g(n)$ if `method` is equal to `bfs`.
- $f(n) = h(n)$ if `method` is equal to `greedy`.
- $f(n) = g(n) + h(n)$ if `method` is equal to `astar`.

Question report 1.2. Make a function `graph_search_test`( ) that test your graph search function on one of the provided graphs.

> File name: `me570_hw4.py`
>
> Method name: `graph_search_test`
>
> Description: Call `graph_search`( ) to find a path between the bottom left node and the top right node of the `graphVectorMedium` graph from the `graph_test_data_load`( ) function (see Question provided 0.2). Then use `Graph.plot`( ) to visualize the result.

Include the figure in your report, and visually check the result with the solution given by `graphVectorMedium_solved` (see Question report 1.1).

This will give you the confidence that your `A`* algorithm works well before moving on.

optional Add an optional argument to `Graph.search`( ) to enable/disable plots of the solution after every node expansion, so that you can verify each step of the algorithm.

Problem 2: Application of `A`* to the sphere world

In this problem you will apply the `A`* graph search function from Problem 1 to a discretized version of the sphere world used in Homework 3. The instructions below assume that you all the functions and data from Homework 3 (that you have developed or from the solution) are in the same directory.

File name: `me570_graph.py`
Class name: `SphereWorldGraph`
Description: A discretized version of the `SphereWorld` from Homework 3 with the addition of a search function.

Question code **2.1.** Create a function that initializes an object with a discretized version of the Sphere World environment.

Class name: `SphereWorldGraph`
File name: `me570_graph.py`

Method name: `__init__`
Description: The function performs the following steps:

1) Instantiate an object of the class `SphereWorld` from Homework 3 to load the contents of the file `sphereworld.mat`. Store the object as the internal attribute `sphereworld`.d
2) Initializes an object `grid` from the class `Grid` initialized with arrays `xx_grid` and `yy_grid`, each one containing `nb_cells` values linearly spaced values from `-10` to `10`.
3) Use the method `grid.eval`(`_`) to obtain a matrix in the format expected by `grid2graph`(`_`) in Question provided 0.4, i.e., with a `true` if the space is free, and a `false` if the space is occupied by a sphere at the corresponding coordinates. The quickest way to achieve this is to manipulate the output of `Total.eval`(`_`) (for checking collisions with the spheres) while using it in conjunction with `grid.eval`(`_`) (to evaluate the collisions along all the points on the grid); note that the choice of the attractive potential here does not matter. **optional** Instead of `Grid.eval`(`_`), you can also use nested for loops.
4) Call `grid2graph`(`_`).
5) Store the resulting `graph` object as an internal attribute.

Input arguments

- `nb_cells`: Number of cells on one side of the grid used for the discretization.

optional It is suggested that you use `Graph.plot`(`_`) to check that the result is consistent with the map shown by `Sphereworld.plot`(`_`).

Question code **2.2.** The function from this question is similar to (and is actually implemented using) the function `Graph.search`(`_`), except that the start and end locations are specified using actual coordinates instead of indeces to nodes in the graph.

Class name: `Graph`
File name: `me570_graph.py`

Method name: `search_start_goal`
Description: This function performs the following operations:

1) Identifies the two indexes `idx_start`, `idx_goal` in `graph.graph_vector` that are closest to `x_start` and `x_goal` (using `Graph.nearestNeighbors`( ) twice, see Question provided 0.3).

2) Calls `Graph.search`( ) to find a feasible sequence of points `x_path` from `idx_start` to `idx_goal`.

3) Appends `x_start` and `x_goal`, respectively, to the beginning and the end of the array `x_path`.

Input arguments

- `x_start` (dim. [2 × 1]), `x_goal` (dim. [2 × 1]): vectors describing the initial and final points for the path search.

Output arguments

- `x_path` (dim. [2 × `nb_path`]): a sequence of pairs of points describing a feasible path. By definition `x_path[:,0]=x_start`, `x_path[:,-1]=x_goal`, and all the other columns are those returned by `Graph.search`( ).

Question **report** **2.1.** Pick three values of `nb_cells` such that, after discretization:

1) Some or all of the obstacles fuse together (`nb_cells` is too low);

2) The topology of the Sphere World is well captured (`nb_cells` is "just right");

3) The graph is much finer than necessary (`nb_cells` is too high).

Include the three values in your report, together with a visualization of the corresponding graphs (using `Graph.plot`( )).

Question **report** **2.2.** Create the following function:

Class name: `SphereWorldGraph`
File name: `me570_graph.py`

Method name: `run_plot`
Description:

1) Load the variables `x_start`, `x_goal` from the internal attribute `sphereworld`.
2) For each goal in `x_goal`:
   (a) Run `search_start_goal`( ) from every starting location in `x_start` to that goal.
   (b) Plot the world using `Sphereworld.plot`( ), together with the resulting trajectories.

Create three objects from `SphereWorldGraph` with the three values of `nb_cells` from the previous question, and call `run_plot`( ) for each one of them. In total, you should produce six different images (three choices for `nb_cell` times two goals, five trajectories in each plot). Include all the images in the report. Please make sure that images from different choices of `nb_cell` but the same goal appear together in the same page (to help comparisons).

Question report **2.3.** Comment on the behavior of the $\mathtt{A}^*$ planner with respect to the choice of `nb_cell`.

Question report **2.4.** Comment on the behavior of the $\mathtt{A}^*$ planner with respect to the potential planner from Homework 3.

Problem 3: Application of $\mathtt{A}^*$ to the two-link manipulator

In this problem you will apply the graph search function you implemented in Problem 1 to the two-link manipulator from Homework 2. In this case, the coordinates in the field `'x'` of `graph_vector` will represent the pairs of angles (θ_1, θ_2) for the two links (as was specified in Homework 2).

The file `twolink_freeSpace_data.mat` contains a dictionary `grid` that describes the configurations of angles for the two-link manipulator that collide with the set of points in `twolink_testData.mat` (see Question provided 0.4 for the format used in `Grid`). This structure is essentially the result of an optional question from Homework 2 (please reread that question for details).

Question provided **3.1.** This method loads the data from the file `twolink_freeSpace_data.mat`

File name: `me570_robot.py`
Class name: `TwoLinkGraph`
Description: A class for finding a path for the two-link manipulator among given obstacle points using a grid discretization and $\mathtt{A}^*$.

File name: `me570_robot.py`
Method name: `load_free_space_grid`
Description: Loads the contents of the file `twolink_freeSpace_data.mat`
Returns arguments

- `grid` (type `Grid`): a grid where the attributes `xx_grid`, `yy_grid`, `fun_evalued` are set to the value loaded from `twolink_freeSpace_data.mat`.

Question **report** **3.1.** For this question, you need to implement the following functions:

Class name: `TwoLinkGraph`
File name: `me570_robot.py`

Method name: `plot`
Description: Use the method `Graph.plot`(␣) to visualize the contents of the attribute `graph`.

Method name: `search_start_goal`
Description: Use the method `Graph.search`(␣) to search a path in the graph stored in `graph`.
Input arguments

- `theta_start` (dim. [2 × 1]), `theta_goal` (dim. [2 × 1]): vectors describing the initial and final joint angles for the path search.

Output arguments

- `theta_path` (dim. [2 × `nb_path`]): a sequence of pairs of angles describing a feasible path.

Use the function `Graph.plot`(␣) to visualize the contents of the attribute `graph` of an object of class `TwoLinkGraph`. Include the figure in your report.

Question **report** **3.2.** The following function visualizes a sample path, both in the graph, and in the workspace.

File name: `me570_hw4.py`
Method name: `twolink_search_test`
Description: Visualize, both in the graph, and in the workspace, a sample path where the second link rotates and then the first link rotates (both with constant speeds).

You should obtain figures similar to those shown in Figure 2. In your report, explain why in the area delimited the dashed rectangle and marked with the symbol there are no nodes.

Question **optional** **3.1.** Modify the functions from the previous problems to work with the topology of the configuration space of the two-link manipulator by following the steps below:

1) Modify `grid2graph`(␣) to allow an additional optional argument `mode='torus'`. If this argument is passed to `grid2graph`(␣), in the final graph the vertices on the left edge become neighbors of those on the right edge, and the vertices on the bottom edge become neighbor of those on the top edge. With this option, we change the topology of the space from $\mathbb{R}^2$ to $\mathbb{S}^1 \times \mathbb{S}^1$, that is, from the plane to the torus.

2) Modify `Graph.heuristic( )` to allow an additional optional argument `mode='torus'`. With this argument, the heuristic will use a mod-2π arithmetic to compute the distance between pairs of angles instead of the Euclidean distance (look at the function `Edge.angle( )` from Homework 1 for inspiration). For instance, with this option the heuristic between the pairs of angles $\begin{bmatrix} 2\pi - 0.1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0.1 \\ 0 \end{bmatrix}$ should be 0.2.

3) Modify `Graph.search( )` with an optional argument that enables the use of `graph_heuristic` with the `'torus'` option (you can either introduce an option `'torus'`, or allow passing the heuristic as a function).

4) Modify the method `TwoLinkGraph.search_start_goal( )` that uses the modified `Graph.search( )` from the previous point.

Question report 3.3. Plot the points `obstacle_points` in `twolink_testData.mat` (see the code the was provided for Question provided 2.2 in Homework 2), call the function `TwoLinkGraph.search_start_goal( )`, and then the method `TwoLink.plotAnimate( )`, for the following start/goal configurations:

- *Easy*: `theta_start=`$\begin{bmatrix} 0.76 \\ 0.12 \end{bmatrix}$, `theta_goal=`$\begin{bmatrix} 0.76 \\ 6.00 \end{bmatrix}$.
- *Medium*: `theta_start=`$\begin{bmatrix} 0.76 \\ 0.12 \end{bmatrix}$, `theta_goal=`$\begin{bmatrix} 2.72 \\ 5.45 \end{bmatrix}$.
- **optional** *Hard*: `theta_start=`$\begin{bmatrix} 3.30 \\ 2.34 \end{bmatrix}$, `theta_goal=`$\begin{bmatrix} 5.49 \\ 1.07 \end{bmatrix}$. For this case, the planner will find a feasible path only if you implement and pass the `'torus'` option.

Note that all values for the angles are in radians. Every time the graph search finds a feasible path, you should see the manipulator move between the obstacle points, where each

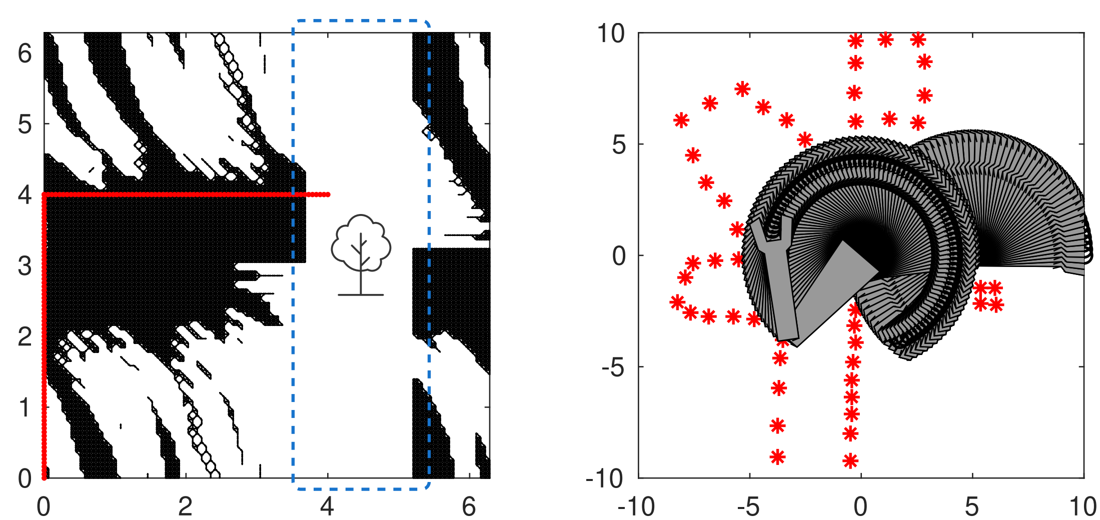


Figure 2: Output of `twolink_search_test( )`

configuration that is plotted is not in collision. For each path, include two plots in your report

1) The plots produced by `TwoLink.plotAnimate()` in your report.

2) A plot of the graph with `Graph.plot()` (as in Question report 3.1), and superimpose the plot of `theta_path` to see the path found by `A*` in the graph.

optional Use the function `time()` from the module `time` to measure the time required by the graph search function in each case. **rtron** There is a problem with the plotting function of the Python version. It does not

Question report 3.4. For the *Easy* case in the question above, comment on the *unwinding* phenomenon that appears if you do not use the `'torus'` option (that is, why the planner does not find the straightforward path that keeps the first link fixed). To obtain full marks, make sure to include the relation between your answer and the visualization of the configuration space from Homework 2.

Question report 3.5. Comment on how close the planner goes to the obstacles, and what you could do about it in a practical situation.

Question optional 3.2. Notice that the majority of the time during planning is spent in checking collisions while generating the free space graph, but most of the graph is never actually explored during search. To significantly speed up the planner, you can use *lazy evaluation*. Lazy evaluation performs collision checking when looking for neighbors in the expansion of a node (line 12 in Algorithm 1), instead of performing it for all the nodes at the beginning. Make a method `TwoLinkGraph.search()` that is the same as `Graph.search()` but:

- The attribute `graph_vector` does not contain neighbor information (the fields `neighbors` and `neighbors_cost` are not used).
- The subfunction `getExpandList()` uses `TwoLink.is_collision()` to find the neighbors of the node being expanded.

Run the function `TwoLinkGraph.search()` on the problems above, and compare the computation times with the previous implementation.

Hint for question code 1.2: Since each element in `graph_vector` already contains a list of indexes of neighbors for each node, this function reduces to compute a set difference (see the `set` class and its method `difference( )`).