Homework 4

ME 570 - Prof. Tron

2020-11-09

In this homework, you will implement the `A*` graph search algorithms, and apply it to a discretization of the sphere world and the two-link manipulator from previous assignments.

General instructions

Programming For your convenience, together with this document, you will find a `zip` archive containing Matlab files with stubs for each of the questions in this assignment; each stub contains an automatically generated description of the function and the function header. You will have to complete these files with the body of each function requested. The goal of this files is to save you a little bit of time, and to avoid misspellings in the function or argument names. The functions from the parts marked as **provided** (see also the *Grading* paragraph below) contain already the body of the function.

Please refer to the guidelines on Blackboard under Information/Programming Tips & Tricks/Coding Practices.

Homework report Along the programming of the requested functions, prepare a small PDF report containing one or two sentences of comments for each question marked as **report**, and including:

- Embedded figures and outputs that are representative of those generated by your code.
- All analytical derivations if required.

You can include comments also on the questions marked as **code**, if you feel it necessary (e.g., to explain any difficulty you might have encountered in the programming), but these will not be graded. In general, *do not* insert the listing of the functions in the report (I will have access to the source files); however, you can insert *short* snippets of code if you want to specifically discuss them.

A small amount of *beauty points* are dedicated to reward reports that present their content in a professional way (see the *Grading criteria* section in the syllabus).

Analytical derivations To include the analytical derivations in your report you can type them in LaTeX(preferred method), any equation editor or clearly write them on paper and use a scanner (least preferred method).

Submission

The submission will be done on Gradescope through two separate assignments, one for the questions marked as **code**, and one for those marked as **report**. See below for details. You can submit as many times as you would like, up to the assignment deadline. Each question is worth 1 point unless otherwise noted. Please refer to the Syllabus on Blackboard for late homework policies.

Report Upload the PDF of you report, and then indicate, for each question marked as **report**, on which page it is answered (just follow the Gradescope interface). Note that some of the questions marked as **report** might include a coding component, which however will be evaluated from the output figures you include in the report, not through automated tests. Many of these questions are intended as checkpoints for you to visually check the results of your functions.

Code questions Upload all the necessary `.m` files, both those written by you, and those provided with the assignment. The questions marked as **code** will be graded using automated tests. Shorty after submission, you will be able to see the results for *some* of the tests: green and red mean that the test has, respectively, passed or not; if a test did not pass, check for clues in the name of the test, the message provided on Gradescope, and the text of this assignment. You can post questions on Blackboard at any time.

Note 1: The final grade will depend also on additional tests that will become visible only after all the grades have been reviewed as a whole. It is therefore important that you examine the results in the figures that you will generate for the report.

Note 2: The automated tests use Octave, a open-source clone of Matlab. For the purposes of this class, you should not encounter any specific compatibility problem. If, however, you suspect that some tests fail due to this incompatibility, please contact the instructor.

Optional and provided questions. Questions marked as **optional** are provided just to further your understanding of the subject, and not for credit. Questions marked as **provided** have already a solution provided in the file accompanying this assignment, which you can directly use unless you want to answer it by yourself. If you submit an answer for optional or provided questions I will correct it, but it will not count toward your grade.

Maximum possible score. The total points available for this assignment are 29.0 (28.0 from questions, plus 1.0 that you can only obtain with beauty points).

Hints

Some hints are available for some questions, and can be found at the end of the assignment (you are encouraged to try to solve the questions without looking at the hints first). If you use these hints, please state so in your report (your grading will not change based on this information, but it is a useful feedback for me).

Use of external libraries and toolboxes All the problems can be solved using Matlab's standard features. You are **not allowed** to use functions or scripts from external libraries or toolboxes (e.g., mapping toolbox), unless specifically instructed to do so (e.g., CVX).

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 structure array `graphVector`, where each element of the array is a struct with fields:

- `neighbors` (dim. [`NNeighbors` × `1`]): an array containing the indexes (in `graphVector`) of the vertices that are adjacent to the current one.
- `neighborsCost` (dim. [`NNeighbors` × `1`]): an array, with the same dimension as the field `neighbors`, containing the cost to move to each neighbor.
- `g` (dim. [`1` × `1`]): scalar variable to store the cost from the starting location along the path through the backpointer.
- `backpointer` (dim. [`1` × `1`]): index of the previous vertex in the current path from the starting location.
- `x` (dim. [`2` × `1`]): the physical (x, y) coordinates of the vertex.

Note that, in the above, the dimension `NNeighbors` is in general different for each element in `graphVector`. The graph is defined by the fields `x`, `neighbors`, `neighborsCost`; 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.

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

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

> `graph_plot`(`graphVector,...`)
>
> Description: The function plots the contents of the graph described by the `graphVector` structure, alongside other related, optional data.
>
> Input arguments
>
> - `graphVector` (dim. [`NNodes` × `1`], type `struct`): .
>
> Optional arguments
>
> - `'nodeLabels',flag`: Enable/disable index labels for each node (default: `false`).
> - `'edgeWeights',flag`: Enable/disable the use of weights as edge labels, shown in black (default: `false`).
> - `'backpointers',flag`: Enable/disable visualization of backpointers, shown as short blue arrows; requires a non-empty value for the field `backpointer` in `graphVector` (default: `true`).
> - `'backpointerCosts',flag`: Enable/disable additional node labels with backpointer costs, shown in blue; requires a non-empty value for the field `g` in `graphVector` (default: `false`).
> - `'idxStart',idxStart`, where `idxStart` is an index in `graphVector`: mark the starting node with a red cross.
> - `'idxGoal',idxGoal`, where `idxGoal` is an index in `graphVector`: mark the goal node with a red diamond.
> - `'idxBest,idxBest`, where `idxBest` is an index in `graphVector`: mark the node currently being expanded with a green square.

- `'idxNeighbors',idxNeighbors`, where `idxNeighbors` is a [NNeighbors × 1] vector of indeces in `graphVector`: mark a set of nodes (e.g., the neighbors of the active node) with green crosses.
- `'idxClosed',idxClosed`, where `idxClosed` is a [NClosed × 1] vector of indeces in `graphVector`: mark a set of closed nodes with blue squares.
- `'pqOpen',pqOpen`, where `pqOpen` is a struct vector storing a priority queue (see Homework 1), with keys being indeces in `graphVector`: mark the set of nodes in the priority queue with red circles.

Requirements: Note that the visualization may become slow when a large number of labels (e.g., more than 1000) are included in the plot (with the options `'nodeLabels'`, `'edgeWeights'`, or `'backpointerCosts'`).

Question provided 0.2. The file `graph_testData.mat` includes already-made graphs, stored in the variables `graphVector` and `graphVector_medium`. Additionally, the file contains the graphs `graphVector_solved`, and `graphVector_medium`, which are the same as `graphVector`, and `graphVectorMedium`, but with the fields `g` and `backpointer` populated.

You can see a couple of examples of use of `graph_plot` and visualize the contents of the file `graph_testData.mat` with the following.

`graph_testData_plot`( )
Description: Visualizes the contents of the file `graph_testData.mat` using `graph_plot`( ) and different sets of visualization options.

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

[`idxNeighbors`]=`graph_nearestNeighbors`(`graphVector,x,k`)
Description: Returns the k nearest neighbors in the graph for a given point.
Input arguments

- `graphVector` (dim. [NNodes × 1], type `struct`): the structure describing the graph of the roadmap, as specified in Homework 4.
- `x` (dim. [2 × 1]): coordinates of the point of which we need to find the nearest neighbors.
- `k` (dim. [1 × 1]): number of nearest neighbors to find.

Output arguments

- `idxNeighbors` (dim. [NNeighbors × 1]): indeces in `graphVector` of the neighbors of `x`. Generally, `NNeighbors=k`, except when `graphVector` 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 `graphVector` structure.

> [`graphVector`]= `grid2graph` (`grid`)
>
> Description: The function should return a `graphVector` structure described by the inputs. See Figure 1 for an example of the expected inputs and outputs.
>
> Input arguments
>
> - `grid` (, type `struct`): a structure with fields `xx` , `yy` , `F` as used in Homework 2 and 3. The field `F` should contain a logical array such that `F(i,j)` is `true` if there is a cell (i.e., no collision) at the (`xx(i)` , `yy(j)`) location, and `false` otherwise.
>
> Output arguments
>
> - `graphVector` (dim. [`NNodes` $\times$ `1`], type `struct`): structure array (as discussed above), obtained from the points on the grid using a 8-neighbors connectivity. For the cost to move from one neighbor to the other, use the Euclidean distance. The size of array should be equal to the number of vertices in the graph.
>
> 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 involving the two-link manipulator, they correspond to angles.

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. In particular, you will apply this to the two-link manipulator from Homework 3.

The graph search function you will develop will be generic, because it can work on a `graphVector` 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 a roadmap in the next).

You will be required to implement the `A*` algorithm, for which the reference pseudo-code for the algorithm can be found on page 531 of the book, and is reproduced in Algorithm 1 with some additional minor clarifications.

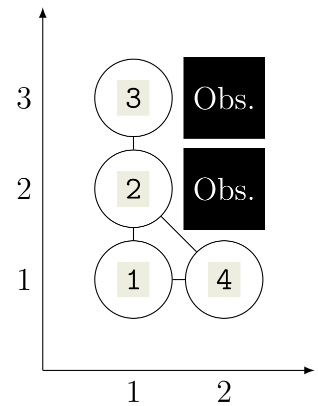
Data structures The algorithm uses a priority queue O, and a list of closed edges C. For the priority queue O, you are expected to use the corresponding set of functions from Homework 1. For the list C, you should use a simple array. See Question code 1.5 for further details.

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` ( ) together with the `pause` command 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).

```
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 `graphVector`. The name `flagFreeSpaceMap` 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 struct array `graphVector` (the fields `g` and `backpointer` are not visualized). This particular `graphVector` can be found in the file `graph_testData.mat`.

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

Question code 1.1.

[`hVal`]= `graph_heuristic`(`graphVector,idxX,idxGoal`)

Description: Computes the heuristic `h` given by the Euclidean distance between the nodes with indexes `idxX` and `idxGoal`.

Input arguments

- `graphVector` (dim. [`NNodes` × `1`], type `struct`): the structure describing the graph, as specified above.
- `idxX` (dim. [`1` × `1`]), `idxGoal` (dim. [`1` × `1`]): indexes of the elements in `graphVector` to use to compute the heuristic.

Output arguments

- `hVal` (dim. [`1` × `1`]): the heuristic (Euclidean distance) between the two elements.

Question code 1.2.

[`idxExpand`]= `graph_getExpandList` (`graphVector,idxNBest,idxClosed`)

Description: Finds the neighbors of element `idxNBest` that are not in `idxClosed` (line 8 in Algorithm 1).

Input arguments

- `graphVector` (dim. [`NNodes` × `1`], type `struct`): the structure describing the graph, as specified above.
- `idxNBest` (dim. [`1` × `1`]): the index of the element in `graphVector` of which we want to find the neighbors.
- `idxClosed` (dim. [`NClosed` × `1`]): array of indexes containing the list of elements of `graphVectors` that have been closed (already expanded) during the search.

Output arguments

- `idxExpand` (dim. [`NNeighborsNotClosed` × `1`]): array of indexes of the neighbors of element `idxNBest` in `graphVector` that are not in `idxClosed` and that are not `idxNBest` (i.e., avoid self-loop edges in the graph).

Question code 1.3 (2 points).

[`graphVector,pqOpen`]= `graph_expandElement` (`graphVector,idxNBest,idxX,idxGoal,pqOpen`)

Description: This function expands the vertex with index `idxX` (which is a neighbor of the one with index `idxNBest`) and returns the updated versions of `graphVector` and

---

Algorithm 1 The `A*` algorithm.

1: Add the starting node n_{start} to O, set $g(n_{start}) = 0$, and set the `backpointer` of x to be empty. ▷ Initialization
2: **repeat**
3: Pick n_{best} from O such that $f(n_{best}) \leq f(n)$ for all $n \in O$.
4: Remove n_{best} from O and add it to C.
5: **if** $n_{best} = q_{goal}$ **then**
6: Exit.
7: **end if**
8: **for** all $x \in \text{Star}(n_{best})$ that are not in C **do** ▷ Expand n_{best}
9: **if** $x \notin O$ **then**
10: Set the value of $g(x)$ to $g(n_{best}) + c(n_{best}, x)$.
11: Set the `backpointer` of x to n_{best}.
12: Add x to O with value $f(x)$.
13: **else if** $g(n_{best}) + c(n_{best}, x) < g(x)$ **then**
14: Update the value of $g(x)$ to $g(n_{best}) + c(n_{best}, x)$.
15: Update the `backpointer` of x to n_{best}.
16: **end if**
17: **end for**
18: **until** O is empty

`pqOpen`.

Input arguments

- `graphVector` (dim. [NNodes × 1], type `struct`): the structure describing the graph, as specified above.
- `idxNBest` (dim. [1 × 1]), `idxX` (dim. [1 × 1]), `idxGoal` (dim. [1 × 1]): indexes in `graphVector` of the vertex that has been popped from the queue, its neighbor under consideration, and the goal location.
- `pqOpen` (dim. [NNodesOpen × 1], type `struct`): structure array with the priority queue of the open nodes.

Output arguments

- `graphVector` (dim. [NNodes × 1], type `struct`): Same as the homonymous input argument, but with the values of the `g` and `backpointer` fields updated for the affected cells.
- `pqOpen` (dim. [NNodesOpenNew × 1], type `struct`): Same as the homonymous input argument, but updated with the new nodes that have been opened.

Requirements: This function corresponds to lines 9–16 in Algorithm 1.

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

[`xPath`]= `graph_path` (`graphVector,idxStart,idxGoal`)

Description: This function follows the backpointers from the node with index `idxGoal` in `graphVector` to the one with index `idxStart` node, and returns the *coordinates* (not indexes) of the sequence of traversed elements.

Input arguments

- `graphVector` (dim. [NNodes × 1], type `struct`): the structure describing the graph, as specified above.
- `idxStart` (dim. [1 × 1]), `idxGoal` (dim. [1 × 1]): indexes in `graphVector` of the starting and end vertices.

Output arguments

- `xPath` (dim. [2 × NPath]): 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 `xPath(:,1)=graphVector(idxStart).x`, and `xPath(:,end)=graphVector(idxGoal).x`.

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

[`xPath,graphVector`]= `graph_search` (`graphVector,idxStart,idxGoal`)

Description: Implements the `A*` algorithm, as described by the pseudo-code in Algorithm 1.

Input arguments

- `graphVector` (dim. [NNodes × 1], type `struct`): the structure describing the graph, as specified above.
- `idxStart` (dim. [1 × 1]), `idxGoal` (dim. [1 × 1]): indexes in `graphVector` of the starting and end vertices.

Output arguments

- `xPath` (dim. [2 × NPath]): array where each column contains the coordinates of the points of the path found from `idxStart` to `idxGoal`.
- `graphVector` (dim. [NNodes × 1], type `struct`): same as the corresponding argument, but with the fields `backpointer` and `g` populated by the search.

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.

For the purposes of this homework, you can assume that a path always exists (although this can be optionally relaxed 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, see below). The cost $c(n, x)$ between two vertices is the one stored in the `neighborsCost` field.

Your implementation of `graph_search`( ) must contain the following elements:

- a priority queue `pqOpen` of the *opened* vertices (the structure O in Algorithm 1); this structure must be manipulated with the functions `priority_*`( ) from Homework 1; use the index of the vertex for the *key* and the function $f(n)$ described above for the `cost`.
- a vector `idxClosed` (dim. [NClosed × 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 optional 1.3 (recommended). Make a function `graph_search_test`( ) that calls `graph_search`( ) to find a path between arbitrary two nodes in the graphs provided in `graph_testData.mat`, and then plots the results. Visually inspect if the results make sense, try to also use your own graphs. This will give you the confidence that your `A*` algorithm works well. Since this routine is the backbone of the next questions, I strongly encourage you to make sure that it works properly before moving on.

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.

Question code 2.1. Create a function that discretizes the Sphere World environment.

[`graphVector`]=`sphereworld_freeSpace_graph`(`NCells`)

Description: The function performs the following steps:

1) Load the file `sphereworld.mat`.
2) Initializes a structure `grid` with fields `xx` and `yy`, each one containing `NCells` values linearly spaced values from `-10` to `10`.
3) Populates the field `grid.F` following 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 best way to manipulate the output of `potential_total`( ) (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.
4) Call `grid2graph`( ).
5) Return the resulting `graphVector` structure.

Input arguments

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

Output arguments

- `graphVector` (dim. [NNodes × 1], type `struct`): the structure describing the graph, as previously specified.

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 `graph_search`( ), except that the start and end locations are specified using actual coordinates instead of indeces to nodes in the graph.

[`xPath`]=`graph_search_startGoal`(`graphVector,xStart,xGoal`)

Description: This function performs the following operations:

1) Identifies the two indexes `idxStart`, `idxGoal` in `graphVector` that are closest

to `xStart` and `xGoal` (using `graph_nearestNeighbors`(␣) twice, see Question provided 0.3).

2) Calls `graph_search`(␣) to find a feasible sequence of points `xPath` from `idxStart` to `idxGoal`.

3) Appends `xStart` and `xGoal`, respectively, to the beginning and the end of the array `xPath`.

Input arguments

- `graphVector` (dim. [NNodes × 1], type `struct`): the structure describing the graph, as specified in the previous problem.
- `xStart` (dim. [2 × 1]), `xGoal` (dim. [2 × 1]): vectors describing the initial and final points for the path search.

Output arguments

- `xPath` (dim. [2 × NPath]): a sequence of pairs of points describing a feasible path. By definition `xPath(:,1)=xStart`, `xPath(:,end)=xGoal`, and all the other columns are those returned by `graph_search`(␣).

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

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

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

3) The graph is much finer than necessary (`NCells` 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:

`sphereworld_search`(`NCells`)

Input arguments

- `NCells` : Size of the discretization grid to use (as number of cells on one side).

Description:

1) Load the variables `xStart`, `xGoal` from `sphereworld.mat`

2) For each of the three values for `NCells`:

   (a) Run the function `sphereworld_freeSpace_graph`(␣) for the given value of `NCell`.

   (b) For each goal in `xGoal`:

      i. Run `graph_search_startGoal`(␣) from every starting location in `xStart` to that goal.

      ii. Plot the world using `sphereworld_plot`(␣), together with the resulting trajectories.

> Requirements: In total, this function should produce six different images (three choices for `NCell` times two goals).

Include all the images in the report. Please make sure that images from different choices of `NCell` but the same goal appear together in the same page (to help comparisons).

Question **report** **2.3.** Comment on the behavior of the `A*` planner with respect to the choice of `NCell`.

Question **report** **2.4.** Comment on the behavior of the `A*` planner with respect to the potential planner from Homework 3.

Problem 3: Application of `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 `graphVector().x` will represent the pairs of angles (θ_1, θ_2) for the two links (as was specified in Homework 2).

Question **report** **3.1.** The file `twolink_freeSpace_data.mat` contains a struct `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). For this question, you need to implement the following function:

> `twolink_freeSpaceGraph` ( )
> Description: The function performs the following steps
>
> 1) Loads the contents of `twolink_freeSpace_data.mat`.
> 2) Calls `grid2graph` (.)
> 3) Stores the resulting `vectorGraph` struct array in the file `twolink_freeSpace_graph.mat`.

Use the function `graph_plot` ( ) to visualize the contents of the file twolink_freeSpace_graph.mat. Include the figure in your report.

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 `'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 `'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 1for 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 instead of $2\pi - 0.4$.

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 handle).

4) Make a function `twolink_search_startGoal`(_) with the specifications below.

> [`thetaPath`]=`twolink_search_startGoal`(`thetaStart,thetaGoal`)
> Description: This function works in the same way as `graph_search_startGoal`(_) in Question code 2.2, but it loads `graphVector` from `twolink_freeSpace_Graph.mat` instead of obtaining it via an input argument.
> Input arguments
> - `thetaStart` (dim. [2 × 1]), `thetaGoal` (dim. [2 × 1]): vectors describing the initial and final joint angles for the path search.
>
> Output arguments
> - `thetaPath` (dim. [2 × NPath]): a sequence of pairs of angles describing a feasible path. By definition `thetaPath(:,1)=thetaStart`, `thetaPath(:,end)=thetaGoal`, and all the other columns are those returned by `graph_search`(_).

Question report 3.2. Call the function `graph_search_startGoal`(_) (or `twolink_search_startGoal`(_) if you completed the previous optional question), and the `twolink_animatePath`(_), for the following start/goal configurations:

- *Easy*: `thetaStart=[0.76; 0.12]`, `thetaEnd=[0.76; 6.00]`.
- *Medium*: `thetaStart=[0.76;0.12]`, `thetaEnd=[2.72;5.45]`.
- **optional** *Hard*: `thetaStart=[3.30;2.34]`, `thetaEnd=[5.49; 1.07]`. For this case, the planner will find a feasible path only if you implement and pass the `'torus'` option to `grid2graph`(_).

Note that all values for the angles are in radians.

Question report 3.3 (2 points). For the *Easy* case in the question above, comment on the *unwinding* phenomenon that appears if you do not use the `'torus'` option for `grid2graph`(_) (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 3.4. Comment on how close the planner goes to the obstacles, and what you could do about it in a practical situation. Include all the final figures in your report.

Question optional 3.2. If you implemented the `method` option for `graph_search`, repeat the above with the different strategies, and compare the computation times (e.g., using the `tic` and `toc` functions in Matlab).

Question optional 3.3. 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 8 in Algorithm 1), instead of performing it for all the nodes at the beginning. Make a function `twolink_graph_search`( ) that is the same as `graph_search`( ) but:

- The input `graphVector` does not contain neighbor information (the fields `neighbors` and `neighborsCost` are empty).
- The subfunction `getExpandList`( ) uses `twolink_checkCollision`( ) to find the neighbors of the node being expanded.

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

Question **optional** **3.4.** Adapt the functions above to solve the sphere world planning problem from Homework 3.

Problem 4: Homework feedback

Question **report** **4.1.** Indicate an estimate of the number of hours you spent on this homework (possibly broken down by problem). Explain what you found hard/easy about this homework, and include any suggestion you might have for improving it.

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