

# Homework 4 (Matlab 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 structure array `graphVector`, where each element of the array is a struct with fields:

- `neighbors` (dim.  $[N\text{Neighbors} \times 1]$ ): an array containing the indexes (in `graphVector`) of the vertices that are adjacent to the current one.
- `neighborsCost` (dim.  $[N\text{Neighbors} \times 1]$ ): an array, with the same dimension as the field `neighbors`, containing the cost to move to each neighbor.
- `g` (dim.  $[1 \times 1]$ ): scalar variable to store the cost from the starting location along the path through the backpointer. Use a default value of `[]` if the backpointer is not set.
- `backpointer` (dim.  $[1 \times 1]$ ): index of the previous vertex in the current path from the starting location. Use a default value of `[]` 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 `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. Note that Matlab will fill the fields with the default values for *all* the elements in the array whenever they are set for a single element.

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.  $[N\text{Nodes} \times 1]$ , 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 green 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`).
- `'heuristic',flag`: Enable/disable additional node labels with the heuristic (requires the function `graph_heuristics(.)`) (default: `false`).
- `'start',idxStart`, where `idxStart` is an index in `graphVector`: mark the starting node with a red cross.
- `'goal',idxGoal`, where `idxGoal` is an index in `graphVector`: mark the goal node with a red diamond.
- `'best',idxBest`, where `idxBest` is an index in `graphVector`: mark the node currently being expanded with a green square.
- `'neighbors',idxNeighbors`, where `idxNeighbors` is a  $[N_{\text{Neighbors}} \times 1]$  vector of indices in `graphVector`: mark a set of nodes (e.g., the neighbors of the active node) with green crosses.
- `'closed',idxClosed`, where `idxClosed` is a  $[N_{\text{Closed}} \times 1]$  vector of indices 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 indices 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 `graphVectorMedium`. Additionally, you can access `graphVector_solved`, and `graphVectorMedium_solved`, which are the same as `graphVector`, and `graphVectorMedium`, but with the fields `g` and `backpointer` populated.

**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_query,k_nearest )
```

**Description:** Returns the  $k$  nearest neighbors in the graph for a given point.

### Input arguments

- `graphVector` (dim.  $[N\text{Nodes} \times 1]$ , type `struct`): the structure describing the graph of the roadmap, as specified in Homework 4.
- `x_query` (dim.  $[2 \times 1]$ ): coordinates of the point of which we need to find the nearest neighbors.
- `k_nearest` (dim.  $[1 \times 1]$ ): number of nearest neighbors to find.

### Output arguments

- `idxNeighbors` (dim.  $[N\text{Neighbors} \times 1]$ ): indices 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_struct )
```

**Description:** The function returns a `graphVector` structure described by the inputs. See Figure 1 for an example of the expected inputs and outputs.

### Input arguments

- `grid_struct` (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.  $[N\text{Nodes} \times 1]$ , 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, it uses the Euclidean distance.

**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.

## 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 `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 the two-link manipulator in the next).

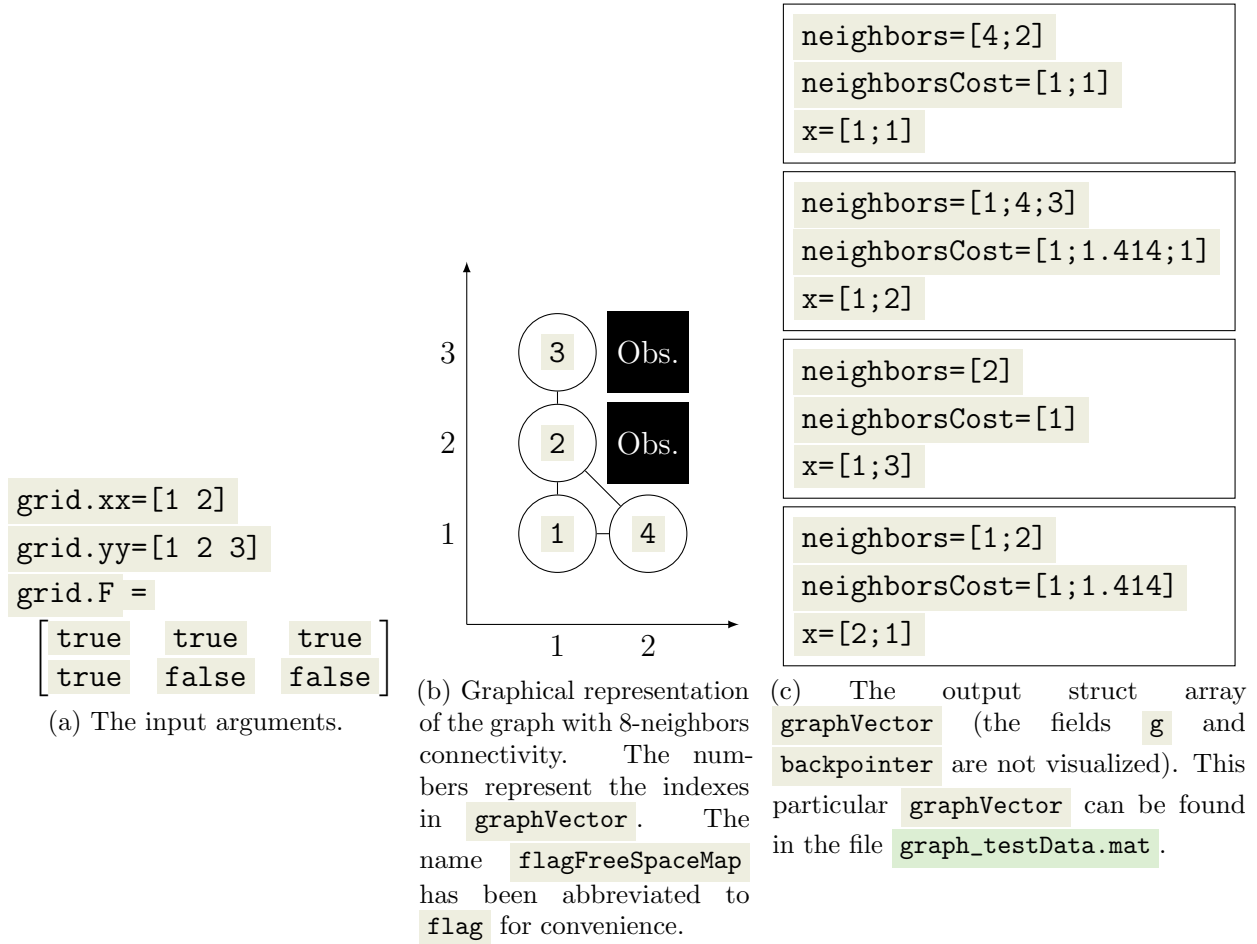


Figure 1: Example of the input and output arguments for `grid2graph` ( ).

You will be required to implement the A\* algorithm for which the reference pseudo-code is reproduced in Algorithm 1 (this is 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.

[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**, **idxGoal**: indexes of the elements in **graphVector** to use to compute the heuristic.

**Output arguments**

- **hVal**: the heuristic (Euclidean distance) between the two elements.

---

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

---

```
1: 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  
2: repeat  
3:   Pick  $n_{best}$  from  $O$  such that  $f(n_{best}) \leq f(n_O)$  for all  $n_O \in O$ . ▷ PriorityMinExtract  
4:   Remove  $n_{best}$  from  $O$  and add it to  $C$ .  
5:   Ver. 1 if  $n_{best} = n_{goal}$  then ▷ Path  
6:     Return path using backpointers from  $n_{goal}$ .  
7:   end if  
8:   Ver. 2 if  $g(n_{goal})$  is set and  $g(n_{goal}) \leq f(n_O)$  for all  $n_O \in O$  then ▷ Path  
9:     Return path using backpointers from  $n_{goal}$ .  
10:  end if  
11:    $S = \text{ExpandList}(n_{best})$   
12:   for all  $x \in S$  that are not in  $C$  do ▷ ExpandElement  
13:     if  $x \notin O$  then  
14:       Set the backpointer cost  $g(x) = g(n_{best}) + c(n_{best}, x)$ .  
15:       Set the backpointer of  $x$  to  $n_{best}$ .  
16:       Compute the heuristic  $h(x)$ . ▷ Heuristic  
17:       Add  $x$  to  $O$  with value  $f(x) = g(x) + h(x)$ . ▷ PriorityInsert  
18:     else if  $g(n_{best}) + c(n_{best}, x) < g(x)$  then ▷ A better path to  $x$  has been found  
19:       Update the backpointer cost  $g(x) = g(n_{best}) + c(n_{best}, x)$ .  
20:       Ver. 2 Update the value of  $f(x) = g(x) + h(x)$  in  $O$  (might require computing  
     $h(x)$  again). ▷ Heuristic  
21:       Update the backpointer of  $x$  to  $n_{best}$ .  
22:     end if  
23:   end for  
24: until  $O$  is empty
```

---

**Question provided 1.1.** The following function demonstrate the use of `graph_plot`, and visualizes two graphs (already solved with A\*) from the provided file `graph_testData.mat`:

```
graph_testData_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_testData_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`.

**Question code 1.2.**

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

**Description:** Finds the neighbors of element `idxNBest` that are not in `idxClosed` (line 12 in Algorithm 1).

**Input arguments**

- `graphVector` (dim.  $[N\text{Nodes} \times 1]$ , type `struct`): the structure describing the graph, as specified above.
- `idxNBest`: the index of the element in `graphVector` of which we want to find the neighbors.
- `idxClosed` (dim.  $[N\text{Closed} \times 1]$ ): array of indexes containing the list of elements of `graphVectors` that have been closed (already expanded) during the search.

**Output arguments**

- `idxExpand` (dim.  $[N\text{NeighborsNotClosed} \times 1]$ ): array of indexes of the neighbors of element `idxNBest` in `graphVector`

This correspond to Line 11 of Algorithm 1.

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

```
[ 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 `pqOpen`.

**Input arguments**

- `graphVector` (dim.  $[N\text{Nodes} \times 1]$ , type `struct`): the structure describing the graph, as specified above.
- `idxNBest`, `idxX`, `idxGoal`: indexes in `graphVector` of the vertex that has

been popped from the queue, its neighbor under consideration, and the goal location.

- `pqOpen` (dim.  $[N_{\text{NodesOpen}} \times 1]$ , type `struct`): structure array with the priority queue of the open nodes.

#### Output arguments

- `graphVector` (dim.  $[N_{\text{Nodes}} \times 1]$ , 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.  $[N_{\text{NodesOpen}} \times 1]$ , type `struct`): 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.

```
[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.  $[N_{\text{Nodes}} \times 1]$ , type `struct`): the structure describing the graph, as specified above. `idxStart`, `idxGoal`: indexes in `graphVector` of the starting and end vertices.

#### Output arguments

- `xPath` (dim.  $[2 \times N_{\text{Path}}]$ ): 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 `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.  $[N_{\text{Nodes}} \times 1]$ , type `struct`): the structure describing the graph, as specified above.
- `idxStart` (dim.  $[1 \times 1]$ ), `idxGoal` (dim.  $[1 \times 1]$ ): indexes in `graphVector` of the starting and end vertices.

#### Output arguments

- `xPath` (dim.  $[2 \times N_{\text{Path}}]$ ): array where each column contains the coordinates of



the points of the path found from `idxStart` to `idxGoal`.

- `graphVector` (dim.  $[N_{\text{Nodes}} \times 1]$ , 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.

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 `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.  $[N_{\text{Closed}} \times 1]$ ), 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.

`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_testData.mat` file (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.

**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 an structure `grid` initialized with fields `xx` and `yy`, each one containing `NCells` values linearly spaced values from `-10` to `10`.
- 3) Use the `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 `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. **optional** Instead of `grid_eval( )`, you can also use nested for loops.
- 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 (and is actually implemented using) the function `graph_search( )`, except that the start and end locations are specified using actual coordinates instead of indices 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.  $[NNodes \times 1]$ , type `struct`): the structure describing the graph, as specified in the previous problem.
- `xStart` (dim.  $[2 \times 1]$ ), `xGoal` (dim.  $[2 \times 1]$ ): vectors describing the initial and final points for the path search.

#### Output arguments

- `xPath` (dim.  $[2 \times NPath]$ ): 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.

In total, you should produce six different images (three choices for `NCell` times two goals, five trajectories in each plot). 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  $(\theta_1, \theta_2)$  for the two links (as was specified in Homework 2).

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).

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

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

`twolink_testPath` ( )

**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 `'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- $2\pi$  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 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.  $[2 \times 1]$ ), `thetaGoal` (dim.  $[2 \times 1]$ ): vectors describing the initial and final joint angles for the path search.

#### Output arguments

- `thetaPath` (dim.  $[2 \times \text{NPath}]$ ): a sequence of pairs of angles describing a feasible

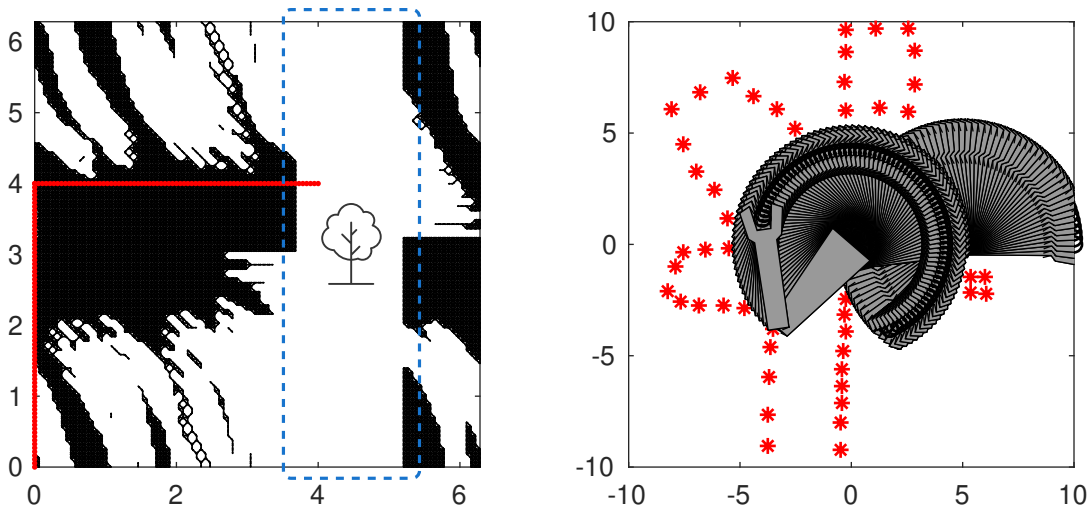


Figure 2: Output of `twolink_testPath` ( )

path. By definition `thetaPath(:,1)=thetaStart`, `thetaPath(:,end)=thetaGoal`, and all the other columns are those returned by `graph_search` (`_`).

**Question report 3.3.** Plot the points `obstaclePoints` in `twolink_testData.mat`, call the function `graph_search_startGoal` (`_`) (or `twolink_search_startGoal` (`_`), if you completed the previous optional question), and then `twolink_animatePath` (`_`), for the following start/goal configurations:

- *Easy*: `thetaStart`= $\begin{bmatrix} 0.76 \\ 0.12 \end{bmatrix}$ , `thetaGoal`= $\begin{bmatrix} 0.76 \\ 6.00 \end{bmatrix}$ .
- *Medium*: `thetaStart`= $\begin{bmatrix} 0.76 \\ 0.12 \end{bmatrix}$ , `thetaGoal`= $\begin{bmatrix} 2.72 \\ 5.45 \end{bmatrix}$ .
- **optional** *Hard*: `thetaStart`= $\begin{bmatrix} 3.30 \\ 2.34 \end{bmatrix}$ , `thetaGoal`= $\begin{bmatrix} 5.49 \\ 1.07 \end{bmatrix}$ . 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 configuration that is plotted is not in collision. For each path, include two plots in your report

- 1) The plots produced by `twolink_animatePath` (`_`) in your report.
- 2) A plot of the graph with `graph_plot` (`_`) (as in Question report 3.1), and superimpose the plot of `thetaPath` to see the path found by A\* in the graph.

**optional** Use the Matlab functions `tic` (`_`) and `toc` (`_`) to measure the time required by the graph search function in each case. **rtrn** 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.** 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 12 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.

**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).