7MR10070: Software and Robotic Integration
Semester 2

Assignment 1

Written by Alexandros Megalemos

Path Planning Part 1

Task 1

Algorithm (a)

```
Point p = (x, y, z)
List of points P = [p_1, …, p_n]
List of output points O = [o_1, …, o_n]
For each p in p ∈ P:
      Retrieve pixel intensity c, where c ∈ [0,1]
      If c = 1 then O ← p
```

Algorithm (b)

```
Point p = (x,y,z)
Point ep is a point p representing an entry point
Point tp is a point p representing a target point
List of entry points EP = [ep_1, …, ep_n]
List of target points TP = [tp_1, …, tp_n]
List of ventricle points VP = [p_1, …, p_n]
List of output points O = [(ep_1, tp_1), …, (ep_n, tp_n)]
Line l is a line between two points
For each ep in p ∈ EP:
      For each tp in p ∈ TP:
            l = eptp→
            For each p in p ∈ eptp→
                  If does not p ∈ VP then O ← (ep, tp)
```

Algorithm (c)

```
Point p = (x,y,z)
Point ep is a point p representing an entry point
Point tp is a point p representing a target point
List of entry points EP = [ep_1, …, ep_n]
List of target points TP = [tp_1, …, tp_n]
List of blood vessel points BVP = [p_1, …, p_n]
List of output points O = [(ep_1, tp_1), …, (ep_n, tp_n)]
Line l is a line between two points
For each ep in p ∈ EP:
      For each tp in p ∈ TP:
            l = eptp→
            For each p in p ∈ eptp→
                  If does not p ∈ BVP then O ← (ep, tp)
```

Algorithm (d)

```
Point p = (x,y,z)
Point ep is a point p representing an entry point
Point tp is a point p representing a target point
List of entry points EP = [ep_1, …, ep_n]
List of target points TP = [tp_1, …, tp_n]
List of blood vessel points BVP = [p_1, …, p_n]
List of output points O = [(ep_1, tp_1), …, (ep_n, tp_n)]
Line l is a line between two points
For each ep in p ∈ EP:
      For each tp in p ∈ TP:
            l = eptp→
            l_p = ⊥ eptp→
            angle = ∠ll_p
            If angle < 55 then O ← (ep, tp)
```

Task 2

The constants included in the following algorithms represent the time taken by ***if-statements***. Perhaps they could have been omitted

Algorithm (a)

```
For each point in target points:
      Get the coordinates (x, y, z) of the point
      Retrieve the pixel value of (x, y, z) in the image
      If pixel value == 1:
            Add the point to the list
```

$Big\ O = O(N_{tp} * R_{pv})$

$where\ N_{tp}\ is\ the\ total\ number\ of\ target\ points, R_{pv}\ is\ the\ lookup\ time\ for\ the\ pixel\ value$

Algorithm (b)

```
For each entry point in entry points:
      For each target point in target points:
            Get the line between the two points
            Get the points of the line
            validLine = true
            For each point on the line:
                  If point passes through ventricle:
                        validLine = false
                        break
            if validLine:
                  add (entry, target) point to valid points list
```

$Big\ O = O(2 * N_{ep} * N_{tp} * N_{pl})$

$where\ N_{ep}\ is\ the\ number\ of\ entry\ points, NP_{tp}\ is\ the\ number\ of\ target\ points\ and\ N_{Pl}\ is\ the\ number\ of\ points\ in\ the\ line$

Algorithm (c)

```
For each entry point in entry points:
      For each target point in target points:
            Get the line between the two points
            Get the points of the line
            validLine = true
            For each point on the line:
                  If point passes through blood vessel:
                        validLine = false
                        break
            if validLine:
                  add (entry, target) point to valid points list
```

$Big\ O = O(2 * N_{ep} * N_{tp} * N_{pl})$

$where\ N_{ep}\ is\ the\ number\ of\ entry\ points, NP_{tp}\ is\ the\ number\ of\ target\ points\ and\ N_{Pl}\ is\ the\ number\ of\ points\ in\ the\ line$

Algorithm (d)

```
For each entry point in entry points:
      For each target point in target points:
            Get the line, lineET, between the two points
            For each point on line:
                  If point connects with the cortex:
                        Get perpendicular line where lineET passes
through the cortex
                        Calculate the angle between the two lines
                        If angle < 55:
                              Add (entry, target) point to valid points
list
```

$Big\ O = O(2 * N_{ep} * N_{tp} * N_c * T_a)$

$where\ N_{ep}\ is\ the\ number\ of\ entry\ points, NP_{tp}\ is\ the\ number\ of\ target\ points, N_c\ is\ the\ number\ of\ cortex\ points\ and\ T_a is\ the\ time\ needed\ to\ calculate\ the\ angle\ between\ the\ entry-target\ \ line\ and\ the\ perpendicular\ line\ where\ it\ connects\ on\ the\ cortex$

Task 3

Tests were run using the full data because the test files were crashing for some reason. As in the previous tasks, constants represent the time taken by ***if-statements***.

Validation for each algorithm

For each algorithm in this report the following overall process was used.

1. Pick a small subset of the data
2. Visually look for a trajectory / point that is obviously valid for a task and for one that is obviously invalid for a task. For example, when filtering for targets within the hippocampus, the invalid point could be one outside the image.
3. Run the algorithm
4. Inspect the output in slicer
5. Increase the subset of data gradually

Algorithm (a)

Introduction and Methods: This one is straight forward. We simply transform the input node to an IKJ matrix and then loop over each target image. To decide if a target point is within our target area, we iterate over all the target points and retrieve the pixel value for each one. If its value is greater than zero (or 1), it is a valid target. The function used to do this is called **getFilteredTargets(targets, area)** and should accept any area we want to filter for.

Validation: Here I used the steps specified above.

$$Big\ O = O\left(N_{tp} * R_{pv}\right)$$

$$where\ N_{tp}\ is\ the\ total\ number\ of\ target\ points, R_{pv}\ is\ the\ lookup\ time\ for\ the\ pixel\ value.$$

Changes made to the algorithm affecting the time complexity: No fundamental changes were made. Here we use the GetRASTToIJKMatrix function to represent the image in a different structure.

Time taken: 0.001s (average of 10 runs)

Rejected trajectories: 78336

Algorithm (b)

Introduction and Methods: This one had to change a bit from the original pseudocode. We first create an oriented bounding box tree (OBBTree) of the ventricles. We then iterate over each entry and target pair and check if the pair intersects any of the bounding boxes defined by the OBBTree. If there is an intersection, we reject the path. The function used to do this is called **getTrajectoriesAvoidingArea(entriesAndTargets, area)** and should accept any area we want to avoid. It is important to note that this function uses **isPassThroughArea(tree, entry, target)** which is where the actual intersection check is made. I chose to separate the check so that it can be used in combination with the other constraints, without having to loop through each entry target pair for each one every time.

Validation: Here I used the steps specified above

$Big\ O = O\left(2 * N_{ep} * N_{tp} * log\left(N_{vp}\right) + \ N_{vp} log\left(N_{vp}\right)\right)$

$where\ N_{ep}\ is\ the\ number\ of\ entry\ points, NP_{tp}\ is\ the\ number\ of\ target\ points\ and\ N_{vp}\ is\ the\ number\ of\ ventricle\ points$

Changes made to the algorithm affecting the time complexity: Instead of traversing through all the points in the line between the entry and target point, we instead search using a tree whose lookup time should be $log\left(N_{vp}\right)$ instead of N_{vp}. However, we'll need to create the tree (once), and that should take $N_{vp} log\left(N_{vp}\right)$.

Time taken: 3s (average of 10 runs)

Rejected trajectories: 10205

Algorithm (c)

Introduction and Methods: This one had to change a bit from the original pseudocode. We first create an oriented bounding box tree (OBBTree) of the blood vessels. We then iterate over each entry and target pair and check if the pair intersects any of the bounding boxes defined by the OBBTree. If there is an intersection, we reject the path. The function used to do this is called **getTrajectoriesAvoidingArea(entriesAndTargets, area)** and should accept any area we want to avoid. It is important to note that this function uses **isPassThroughArea(tree, entry, target)** which is where the actual intersection check is made. I chose to separate the check so that it can be used in combination with the other constraints, without having to loop through each entry target pair for each one every time.

Validation: Here I used the steps specified above.

$Big\ O = O\left(2 * N_{ep} * N_{tp} * log\left(N_{bvp}\right) + N_{bvp} log\left(N_{bvp}\right)\right)$

$where\ N_{ep}\ is\ the\ number\ of\ entry\ points, NP_{tp}\ is\ the\ number\ of\ target\ points\ and\ N_{bvp}\ is\ the\ number\ of\ blood\ vessel\ points$

Changes made to the algorithm affecting the time complexity: Instead of traversing all the points in the line between the entry and target point, we instead search using a tree whose lookup time should be $log\left(N_{bvp}\right)$ instead of N_{bvp}. However, we'll need to create the tree (once), and that should take $N_{vp} log\left(N_{bvp}\right)$.

Time taken: 60s (average of 10 runs)

Rejected trajectories: 47855

Algorithm (d)

Introduction and Methods: This one had to change a bit from the original pseudocode. We first create an oriented bounding box tree (OBBTree) of the cortex. We then iterate over each entry and target pair and check if the pair intersects any of the bounding boxes defined by the OBBTree. If there is an intersection, we create a line perpendicular to the intersection. We then create two vectors, one for our entry/target pair and one for the intersecting points and calculate the angle between the two. If the angle is below the specified limit (55), we accept the path. The function used to do this is called **getTrajectoriesWithSpecifiedAngle (entriesAndTargets, area, specifiedAngle)** and should accept any area we want check the angles for. It is important to note that this function uses **isValidAngle()** which is where the actual check for the angle is made. I chose to separate the check so that it can be used in combination with the other constraints, without having to loop through each entry target pair for each one every time.

Validation: Here I used the steps specified above.

$$Big\ O = O\left(2 * N_{ep} * N_{tp} * log\left(N_{cp}\right) * T_a + N_{cp} log\left(N_{cp}\right)\right)$$

$$where\ N_{ep}\ is\ the\ number\ of\ entry\ points, NP_{tp}\ is\ the\ number\ of\ target\ points\ and\ N_{cp}\ is\ the$$

$$number\ of\ points\ cortex\ points\ and\ T_a\ is\ the\ time\ needed\ to\ calculate\ the\ angle\ between\ two\ vectors$$

Changes made to the algorithm affecting the time complexity: Instead of traversing all the points in the line between the entry and target point, we instead search using a tree whose lookup time should be $log\left(N_{cp}\right)$ instead of N_{cp}. However, we'll need to create the tree (once), and that should take $N_{cp} log\left(N_{cp}\right)$.

Time taken: 105s (average of 10 runs)

Rejected trajectories: 17426

Task 4

Here we combine all three constrains under the same nested for loop (between entry and target points). We start with the least complex (in terms of time and space) algorithm and work our way down to the most complex. Therefore:

1. First, create the OBB trees required for each task
2. Then, filter for targets that are within the hippocampus
3. After that, filter for entry/target trajectories that do not pass through the ventricles
4. After that, we filter for entry/target trajectories that do not pass through blood vessels
5. Finally, we filter so that only trajectories of a certain angle (degrees) are accepted

By doing this, we rule out most of the trajectories before we reach the more expensive checks in our overall algorithm. This is clearly demonstrated by the total time taken of ~25 seconds, whereas before filtering just for valid angles would take more than 100 seconds.

Time taken: 25s (average of 10 runs)

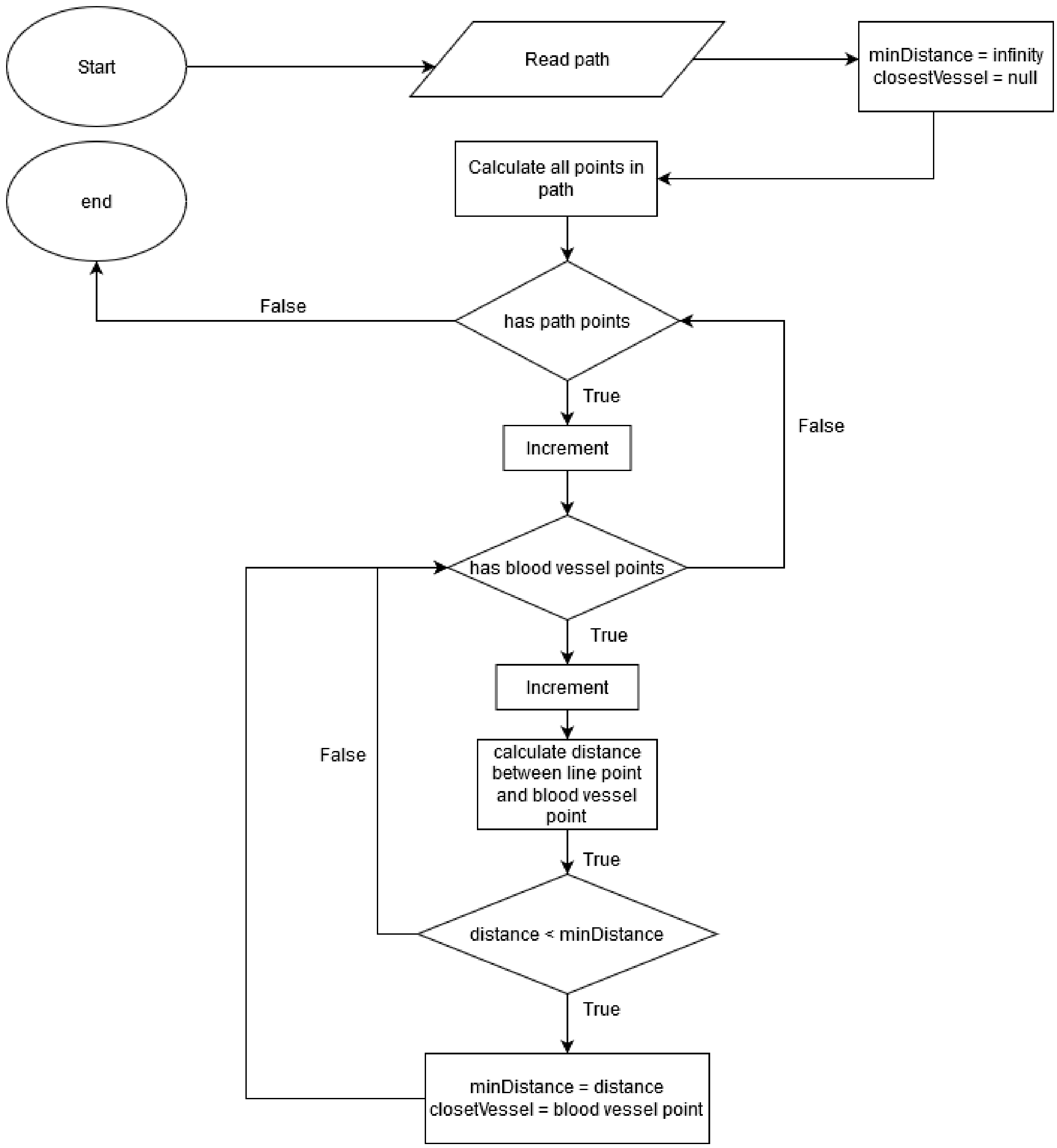
Rejected trajectories: 88603

Path Planning Part 2

Task 1

(a) A naïve/brute-force solution to the problem. It iterates over all possible points and finds the minimum distance between a vessel and a point in the path

```
minDistance = +infinity
closestVessel = null
allPoints = getAllPointsInPath(entry, target)
for each point in allPoints:
	for each bloodVesselPoint:
		Calculate distance (e.g. Euclidean) between the line
point and blood vessel point
		If distance < minDistance:
			minDistance = distance
			closestVessel = bloodVesselPoint
```



1:simplified flowchart of the algorithm

(b) In order to test that my code works, I used a subset of the data outputted by combining all the hard constraints. I visually chose a trajectory with obviously large distance from the blood vessels and another one with an obviously small distance from the blood vessels. I then inspected the output and seeing that it was as expected, I started using more trajectories and larger sets of data.

Code / Git Repository

https://github.com/Meldanen/kcl/tree/master/robotics/slicerTutorials/Tutorial%202%20and%203

The tests should be able to run all code required for the assignment. If you wish you run smaller parts of the code, you can either comment out some of them, or go to *Assignment1Logic.run()* and again, comment/uncomment the parts you want to run.
The submitted version of this only runs the combination of all three constraints when using the GUI to save time.

Tests

testLoadAllData(path): check that data has been loaded successfully
testGetFilteredHippocampusTargets(): check that targets are correctly filtered down to hippocampus targets
testAvoidVentriclesValidPath(): check that the algorithm accepts a path that doesn't pass through the ventricles

testAvoidVentriclesInvalidPath(): check that the algorithm rejects a path that passes through the ventricles

testAvoidBloodVesselsValidPath(): check that the algorithm accepts a path that doesn't pass through the blood vessels
testAvoidBloodVesselsInvalidPath(): check that the algorithm rejects a path that passes through the blood vessels

testAngleValidPath(): check that the algorithm accepts a path that hits the cortex at the correct angle

testAngleInvalidPath(): check that the algorithm rejects a path that hits the cortex at an incorrect angle
testCountRejectedTrajectories(True): To count rejected trajectories and time each part. This is a slow test
testAllTogether(): Just to see if everything is able to run together (pseudo test for task 4). This is a slow test