2ME3 Assignment 3

Justin Staples (staplejw)

March 11, 2017

The purpose of this report is to show the missing parts of the specification for the RegionT and Path Calculation modules. Given below is a brief critique of the interface provided by the modules in this project, as well as the specification for the modules. The Region Module and the Path Calculation Module are given in full at the end with the missing specification details filled in.

Critique

There were a few things I would have changed or considered changing in the interface.

Firstly, the constructor for RegionT allows an input type of real. Because it does not make sense in this context to allow a negative value for width or height, an excpetion needs to be added to show that those values give an invalid region. Negative width and height could make sense in some circumstances, but for the lower_left state variable to remain as is, those values must be positive. I added this as part of the excpetion for RegionT.

Another aspect of the interface that I considered adding to or changing was to include another data type in the interface called LineT or SegmentT. The interface, as it stands, deals with segments often. Perhaps in some circumstances, it might be desirable to have another data type called SegmentT that is constructed from two instances of PointT. Then, maybe the path could be thought of as a list/sequence of segments, as opposed to points.

I was also thinking it might be beneficial in some cases to make more of a distinction between the obstacles and the destinations. They are currently both implemented the exact same way, as a GenericList(RegionT). If we wanted to define specific operations for each type of region, it might be better to have them implemented as classes that extend GenericList(RegionT).

The idea of a path calculation module is quite broad, and there are many different qualities of a path that might need to be studied. As the path calculation module stands now, it is basically a library of different methods for analyzing paths. This library could certainly be added to, and I have come up with a few extra methods that I think might be worth adding. Perhaps new methods could be added called totalRightTurns, or totalLeftTurns that would keep track of which direction the robot has turned. Maybe totalTurns could simply be modified to accomodate this feature. Perhaps there could be a method that analyzes exactly how many times a certain path visits each destination, or a method that returns a real distance that represents the closest that the robot comes to an obstacle over the course of its path. Perhaps another method that returns which quandrant of the map the robot spent the most time in. The path calculation library certainly could be expanded in a number of interesting ways!

Overall, the interface is quite complete, as it offers a wide range of services with a number of different data types. It also offers a generic list module, so it is quite general in this sense.

In terms of the specification given, I just have a few comments about potential ambiguities. The units of the velocity constant are not stated explicitly anywhere. This could cause a slight problem in terms of defining and implementing certain method. Because arccos always returns an angle in radians, the unit of the angular velocity might need to be adjusted to suit this specification (what if is not given in radians per second? Then the basic formula does not work). As well, it is not made clear what the original orientation of the robot is. We are left to assume that it always starts with the correct orientation for the first segment of the path, but this might not be the case. Perhaps the robot is always facing north to start. In this case, it would usually have to do extra turn to start.

Region Module

Template Module

RegionT

Uses

PointT, Constants

Syntax

Exported Types

RegionT = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
RegionT	PointT, real, real	RegionT	InvalidRegionException
pointInRegion	PointT	boolean	

Semantics

State Variables

lower_left: PointT *//coordinates of the lower left corner of the region*
width: real *//width of the rectangular region*
height: real *//height of the rectangular region*

State Invariant

None

Assumptions

The RegionT constructor is called for each abstract object before any other access routine is called for that object. The constructor can only be called once.

Access Routine Semantics

RegionT(p, w, h):

- transition: $\mathit{lower\_left}, \mathit{width}, \mathit{height} := p, w, h$
- output: $out := \mathit{self}$
- exception: $exc := ((\neg(p.\text{xcrd}() + w \leq \text{Constants.MAX\_X})) \lor (\neg(p.\text{ycrd}() + h \leq \text{Constants.MAX\_Y})) \lor (\neg(w > 0)) \lor (\neg(h > 0)) \Rightarrow \text{InvalidRegionException})$

pointInRegion(p):

- output: $out := \exists(q : \text{PointT} | (self.\text{lower\_left.xcrd}() \leq q.\text{xcrd}() \leq self.\text{lower\_left.xcrd}() + self.w) \land (self.\text{lower\_left.ycrd}() \leq q.\text{ycrd}() \leq self.\text{lower\_left.ycrd}() + self.h) : q.\text{dist}(p) < \text{Constants.TOLERANCE})$
- exception: none

Path Calculation Module

Module

PathCalculation

Uses

Constants, PointT, RegionT, PathT, Obstacles, Destinations, SafeZone, Map

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
is_validSegment	PointT, PointT	boolean	
is_validPath	PathT	boolean	
is_shortestPath	PathT	boolean	
totalDistance	PathT	real	
totalTurns	PathT	integer	
estimatedTime	PathT	real	

Semantics

State Variables

none

State Invariant

none

Assumptions

none

Access Routine Semantics

is_validSegment(p_1, p_2):

- output: $out := \forall(i : \mathbb{N} | 0 \leq i < \text{Map.get\_obstacles.size}() : \forall(t : \mathbb{R} | 0 \leq t \leq 1 : \neg(\text{Map.get\_obstacles().getval(i).pointInRegion}(tp_1 + (1-t)p_2)))$

- exception: none

is_validPath(p):

- output: out := start_and_end_safe(p) $\wedge$ all_rescue_regions_visited(p) $\wedge$ all_segments_valid(p)
- exception: none

is_shortestPath(p):

- output: $out := (\neg(\exists(q : \text{PathT}|\text{is\_validPath}(q) : \text{totalDistance}(q) < \text{totalDistance}(p))))\wedge (\text{is\_validPath}(p))$ *// p is a valid path and there does not exist any other valid path with total distance less than p*
- exception: none

totalDistance(p):

- output: $out := +(i : \mathbb{N}|0 \leq i < p.\text{size}() - 1 : p.\text{getval(i).dist}(p.\text{getval(i + 1)}))$
- exception: none

totalTurns(p):

- output: $out := +(i : \mathbb{N}|(1 \leq i < p.\text{size}() - 1) \wedge (\text{changes\_direction(i, p)}) : 1)$ *// counts the number of times the robot changes direction, starting at the second point and going until the second last point (turns cannot be made at the first or last point)*
- exception: none

estimatedTime(p):

- output: $out := \frac{\text{totalDistance(p)}}{\text{Constants.VELOCITY\_LINEAR}} + \frac{\text{total\_angle(p)}}{\text{Constants.VELOCITY\_ANGULAR}}$
- exception: none

Local Functions

// for all destinations, there exists a segment, where there exists a point that is in the region

all_rescue_regions_visited: PathT $\rightarrow$ boolean

all_rescue_regions_visited(p) $\equiv \forall(i : \mathbb{N}|0 \leq i <$ Map.get_desinations.size() : $\exists(j : \mathbb{N}|0 \leq j < p.\text{size}() - 1 : \exists(t : \mathbb{R}|0 \leq t \leq 1 :$ Map.get_destinations.getval(i).pointInRegion (tp.getval(j) + (1 - t)p.getval(j + 1)))))

// the first point of the path is in the safe zone and the last point of the path is in the safe zone

start_and_end_safe: PathT $\rightarrow$ boolean

start_and_end_safe(p) $\equiv$ (Map.get_safeZone().getval(0).pointInRegion(p.getval(0))) $\wedge$ (Map.get_safeZone().getval(0).pointInRegion(p.getval(p.size() - 1))

// for all segments in the path, each one is valid

all_segments_valid: PathT $\rightarrow$ boolean

all_segments_valid(p) $\equiv \forall(i : \mathbb{N}|0 \leq i < p.\text{size}()-1 :$ is_valid_segment(p.getval(i), p.getval(i + 1)))

// the robot changes direction if the turn angle from the previous segment to the next segment is not equal to 0

changes_direction: integer $\times$ PathT $\rightarrow$ boolean

changes_direction(i, p) $\equiv$ (turn_angle(i, p) $\neq$ 0)

// uses the inverse cosine function to find the turn angle

turn_angle: integer $\times$ PathT $\rightarrow$ real

$$\text{turn\_angle}(i, p) = \arccos\left(\frac{\text{dot(i, p)}}{\text{mangtiude\_product(i, p)}}\right)$$

// for a given index and path, it will return the product of the magnitude of the previous segement with the magnitude of the next segment, used in calculating turn angle

magnitude_product: integer $\times$ PathT $\rightarrow$ real

magnitude_product(i, p) = (p.getval(i $-$ 1).dist(p.getval(i)))(p.getval(i).dist(p.getval(i + 1)))

// there are two vectors. the first points from the previous point to the current point, and the second points from the current point to the next point. this method returns the dot product of those two vectors. the first term represents the product of the x-components, the second term is the product of the y-coordinates

dot: integer $\times$ PathT $\rightarrow$ real

$\text{dot}(i, p) = (p.\text{getval}(\text{i}).\text{xcrd}() - p.\text{getval}(\text{i} - 1).\text{xcrd}())(p.\text{getval}(\text{i} + 1).\text{xcrd}() - p.\text{getval}(\text{i}).\text{xcrd}()) + (p.\text{getval}(\text{i}).\text{ycrd}() - p.\text{getval}(\text{i} - 1).\text{ycrd}())(p.\text{getval}(\text{i} + 1).\text{ycrd}() - p.\text{getval}(\text{i}).\text{ycrd}())$

// calculates the total turning angle by adding up all the turning angles for each point in the path. The arccos function always returns a non-negative value between 0 and π, so we do not need to worry about whether the robot turned left or right
total_angle: PathT $\rightarrow$ real
total_angle(p) $= +(i : \mathbb{N} | 1 \leq i < p.\text{size}() - 1 : \text{turn\_angle(i, p)})$