Intro to Mobile Robots 2017

Homework #4

Global Motion Planning

Note to myself - This file has conditional text formatting to hide the answers.

Generally, I'd prefer if you submit all homeworks entirely in electronic form. PDF files are best. See the web page for how to get PDF tools from Adobe for free.

You should do this and all homeworks alone without any undue help from your fellow students. Do not collaborate with others in the class on the technical content of the assignment. Its OK to help each other with Eclipse, Java etc..

Global motion planning is the problem of determining optimal paths through a cost field or an obstacle field which is sufficiently large that significant search is necessary. In this task, you will develop a series of solutions of increasing sophistication starting with a straightforward but too slow solution, and ending with a state of the art real-time replanner.

You can use the code to help you answer many of the questions asked below. If you do not know the right answer, change the code temporarily and see what happens. Some of the issues raised here have not been covered in any depth, or maybe even at all, in class. Play around with the code and learn the answers if necessary. Its also legal to do your own literature search. A surprising amount of info on A* is on the web. For Dstar lite, you can find the following paper on the web:

"Fast Replanning for Navigation in Unknown Terrain" by Sven Koenig and Maxim Likhachev. IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION.

We will implement the second version of Dstar Lite from this paper. Read this paper about 10 times until you have a rough idea what you are going to do.

1. Introduction and Software Design

This assignment will take you all the way from Motion Planning 101 to the implementation of a state of the art motion planner published only a few years ago.

The software implements a simulation of a mobile robot operating in a flat world populated by polygonal obstacles.

A simulated obstacle sensor is able to perceive the obstacles as "cost" values within its circular field of view out to some maximum range. The simulated sensor is not a rangefinder in this case. Instead, it directly reads a hidden "cost map" that encodes the severity of obstacles and makes the data available to the planner according to rules that you specify. The code can be configured to provide a "prior" map of all the obstacles before the robot moves, or to reveal the obstacles as they are perceived.

For simplicity in computing C space, the robot is shaped circularly. The robot is not moved by motion (speed, steering) commands in this case. Rather, a path is computed by the planner which is a sequence of the states and the robot position is set to occupy these states in sequence as the path is executed.

1.1 World View and Planner View

The graphics have been set up to draw the state of the simulated world in the left DrawingArea and the internal state of the planner in the right DrawingArea. Things have been set up so you should not have to make any significant changes to the logical flow of any of the code.

1.2 Code Overview

The file applicationHW1MotionPlanner.java is provided in source form. It contains the main driver routines with all of the parameters, world creation, graphics etc. It configures and then drives the simulator in the Simple Mobile Robot Development Environment ("smrde") to produce fake sensor readings for the robot. It also configures and invokes the main motion planner in an incremental manner and extracts its state to the screen so you can watch the motion planner run.

Javadocs are provided for the entire SMRDE system. It is almost impossible to proceed without figuring out how to access the javadocs. The simplest way is to read them with an HTML browser. The best way is to configure Eclipse so that it integrates them with the source code editor as outlined in the "How to get started in Java" document on the web. Doing this will save you hours and allow you to see the javadocs on anything including the run-time library by hovering the mouse over the variable of interest. Understanding all of the SMRDE environment will take more time than you have, so don't spend too much time with that. Read what you need when you need it. What you need is outlined below.

Figure 1 shows the main files of interest for this assignment:

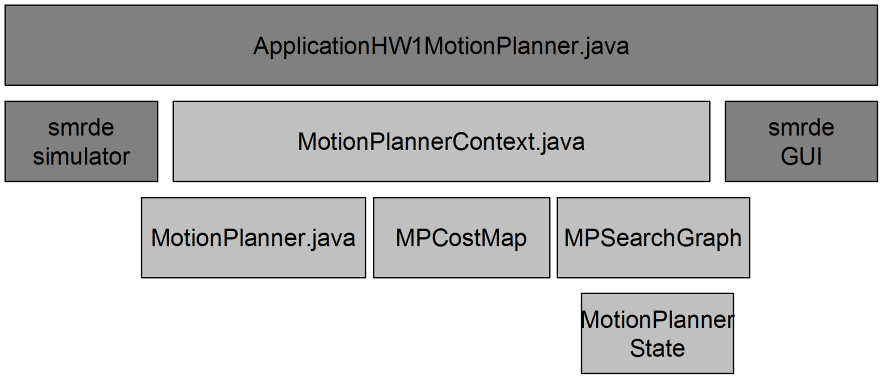

Figure 1: Rough Invocation Hierarchy for the Software for this Assignment.

- `applications.homeworks.ApplicationHW1MotionPlanner` - the main program tied to the MotionPlanner application button. Most of the code that is related to setting up the graphics and simulation is in here. You should need to do nothing in here except tweak some numbers and flags. Most of what you want to tweak can be done with the GUI. From now on this will be called the "main program".
- `applications.homeworks.unlocked.hw1....`
  - `MotionPlanner.java`. A generic discrete motion planner designed for ease of visualization while it runs. Grassfire, Dykstra's, Astar, and Dstar algorithms are supported.

- `MotionPlannerContext.java`. The more concrete context in which the motion planners operate. Provided to keep the planners as ignorant as possible of certain implementation decisions. Owns both the search graph and the cost map which together specify the discretization of the world and how to compute the cost of edges (representing motions) between states.
- `MotionPlannerCostMap.java`. The representation of the world used for motion planning. Converts an initially and/or incrementally specified list of discrete polygonal obstacles into appropriate cost cell regions in the map.
- `MotionPlannerSearchGraph.java`. The representation of the network of possible motions used for motion planning. Supports 4 connected and 8 connected topologies.
- `MotionPlannerHeuristicCalculator.java`. An interface to allow computation of the cost heuristic external to the planner graph. The intention is to keep the planner graph unaware of any planner specific data. `MotionPlannerContext` implements the interface.
- `MotionPlannerState.java`. A state to be used in motion planning. Records the position of the robot, cost data and a backpointer (parent) to be used in extracting the path. Overrides standard methods like equals() and compareTo() so that these objects can be handled directly by all of the Java collections framework - especially `Queues` and `PriorityQueues`.
- `MotionPlannerPerceptionSimulator.java`. A simple simulator to uncover the costs in cells that are within a prescribed radius of the sensor.

Read all of these files, especially the comments, carefully before proceeding.

What are you doing here now :)

Go back and read all the java files carefully - especially the comments.

1.3 JavaDocs

Read the Javadocs for:

- `Random` and learn about how you can supply an integer argument in its constructor for a special kind of repeatable randomness.
- `Queue, LinkedList` and `PriorityQueue`. These are perhaps the most basic data structures used to implement any motion planner. We will use Java's versions but if you ever write your own motion planner, you will care a lot about how these perform. The last two are particular forms of the first. Queues are often classified by whether they add and delete from the front or back and whether they are sorted. The fact that java provides a built in (sorted) priority queue makes this assignment much easier to do.
- `Set` and `LinkedHashSet`. The second is the basic Set used in the code to store references to states that will be displayed in various colors by the graphics. The great thing about Sets is if you try to add something that is already in them, the addition is automatically ignored - so you never have to worry about multiple copies. Testing membership is fast too.
- `Point` and `Point2D`. These are used in a few places to represent points.
- `ArrayList`. This is probably the most versatile data structure in Java (C's STL has a version too) and well worth your time to learn. We will use it for things like `ArrayList <MotionPlannerState>` which is an array of unspecified size (it grows automatically as needed) that contains a list of planner states. Once such object is the solution path. The path is cleared by a call to the `clear()` method of `ArrayList`.
- `smrde.math.RectangleMap` and `smrde.math.RectangleMapField` and learn the basic mechanisms for converting between user coordinates and discrete rectangle coordinates. The conversion from the continuum to discrete states and discrete cost cells is also fundamental to vehicle motion planning. I have provided these for you but in general you would need to write your own.
- `smrde.math.Pose2D`. You may have seen this already in an earlier assignment.
- You may find that you want to know about what java means by the word "interface" and in particular the interface called `Comparable` but I tried to set things up so you would not have to. `MotionPlannerHeuristicCalculator` is also an interface whose methods are implemented in `MotionPlannerContext.`

1.4 First Run & GUI

The main program (`ApplicationHW1MotionPlanner)` implements a GUI that provides useful debugging information. There are two graphics windows. The left is "ground truth" data about the content of the simulation and the right is the "application" data representing the internal state of the motion planner. Polygonal obstacles can be added and deleted in the left window by clicking with the left and right mouse buttons respectively. The planner is invoked in a loop in this file which may or may not:

- complete the entire planning task before redrawing the display
- stop planning once the solution is found (you may want the nav function for all goals)
- move the robot to follow the path generated

Run the SMRDE program now and select homeworks then motion planning and then **read the instruction panel very carefully.** Among other things, it tells you how to use the GUI to change the configuration of the application. This is much more convenient than recompiling in many cases. If you press run now the program will probably print `Path size: 0` many times or crash since too many things are missing.

The line:

```
ranGen = new Random();
```

in the main application file can be changed to provide an integer argument to the constructor. If you do this, the program will generate the same test world over and over. This is useful for debugging but you will want to switch the argument or switch back to no argument in order to test your code well. The RandWorld button in the GUI changes this variable as well if you prefer that method. You can only change GUI parameters if the planner is paused or stopped and if you do a reset after all changes to reload the variables from the GUI to the application.

Various questions are asked in the following. Please answer them all in a single document and pass it in with your code.

Start writing your own solution as outlined below.

2. PlannerGraph and Grassfire Planner [32 points]

It is typical in the design of motion planning software to define a class, here called `MotionPlannerSearchGraph` which encapsulates many characteristics of the search space including discretization, cost heuristics, and the topology of the discrete states.

a) [15 points (12 Qs, 3 code)] Complete the empty or partially empty functions in the file `MotionPlannerSearchGraph.java` and answer questions as outlined below. The functions to complete are:

- `cellCenterInWorld()`
- `generateLegalNeighbors()`

i) [2 points] Read the header for `RectangleMap`. In a separate document draw a square with x and y axes in the bottom left corner and another square next to it with i , j axes in the bottom left corner. The "cells" in the outRect fit exactly and entirely inside the outRect in `RectangleMap` so it is the peripheral cell boundaries that correspond to the edges of the inRect. If you are not quite sure what RectangleMap is doing, write some test code to test it. Write an expression for the coordinates of the center of cell (i,j) in the outRect. Based on this, complete the function `cellCenterInWorld()` in `MotionPlannerSearchGraph`. This is the basic mapping from the continuum to discrete states that turns continuum motion planning into a sequential search problem.

ii) [3 points] Complete the function `generateLegalNeighbors()` in `MotionPlannerSearchGraph` for both the 4 connected and the 8 connected case. This is the basic process which establishes the topological properties of the discrete search space in motion planning.

iii) [5 points] In a separate document discuss briefly how the choice of one or the other of 4 or 8 connectedness affects the types of paths produced and suggest a case when each may be appropriate. Answer this after you implement Grassfire in the next question (part b) because you can then try both and see. Produce a screen capture of the right window (described later) for both cases where the screen has part of the search tree (arrows) drawn and all of the path drawn. The simplest way to do this is to set minExpansions to one and wait for the second time the solution is generated and stop it midway. You can use the pause button to stop drawing so you can capture an image of the screen. Comment on the shapes of the planner horizon in both cases.

iv) [5 points] A basic design decision in motion planning representations is whether to treat obstacles as "high cost" regions or as "no-go" areas where the robot is forbidden to go. In the latter case, one can encode obstacles directly in the planner graph. Suggest how, if the obstacles were known in the function `generateLegalNeighbors()`, they could be considered and processed in the algorithm that generates successors. What is an advantage of this approach? You may be better equipped to answer this question after implementing any of the planners after Grassfire in the assignment.

b) [17 points = 11 Qs + 6 code] Complete the empty or partially empty functions in the planner file `MotionPlanner.java` and answer questions as outlined below. Make sure the main program is set like so:.

```
final double USER_DS = 0.5;
algorithm = Grassfire;
boolean makeRandomWorld = false;
boolean executePlan = false;
boolean doNavFunction = false;
```

Before you do anything read the course notes and form your own ideas about how you would implement the Grassfire algorithm. Blindly following the comments without first knowing what you intend to do is not a strategy for success. Picture the open list as the advancing horizon on screen if that helps you visualize things.

i) [3 points] Look for every reference to `open` in the file `MotionPlanner.java` and see how the methods `remove` and `add` are used to add and delete states to a `Queue`.

> **You should use the provided routines which access `open` rather than do so directly yourself. Using the provided routines will manage the sets `statesChanged` and `statesClosed` etc. which are used for graphics to help you debug and answer questions.**

Complete the function:

- `expandGrassfire(MotionPlannerState state).`

You should now be able to click the run button and watch the planner finding a path (arrows on screen) around a single obstacle in the center of the screen. It will not execute it until you do the next step.

ii) [3 points] Complete the function:

- `extractPath()`

The path is all states from goal to start <u>including the goal and the start</u>. Notice the comment in the header that you must add nodes at the front. In case you think you do, you do not need to implement `getBestHorizonNode()` at this stage because the best horizon node is simply the goal node.

You should now be able to set (in GUI or code):

```
      boolean executePlan = true;
```

and watch the planner plan a path and execute it. After the world appears, you have to press the run button for execution to start.

Congratulations.!@#$%^&*()!

The planner is configured, for the moment, to know the entire map before planning starts and to replan from scratch after each move. Now that it is working, you can temporarily turn on `makeRandomWorld (RandWorld` in the GUI`)` and test your first motion planner on many random worlds. Be aware that motion planning cannot succeed if no collision free path exists. Keep pressing reset to make new worlds as necessary to get one where a solution exists - or temporarily reduce the number of obstacles generated if needed.

You can also left click in the left window to create obstacles and right click on them to delete them. Its fun to block the robot path while it is moving and see it figure out a new path. Obstacle surgery takes place in the Java event dispatching thread, and I have done nothing special to make sure this is robust. Random things might happen if you change the world precisely when the planner is in the middle of reading it. You can add obstacles before clicking run to avoid this issue. It is also fun to set MinExpansions to 1 (press reset) and then watch the search wavefront expand.

iii) [11 points]

iii a) [1 points] Comment on the basic function of the set of closed cells (regardless of how it is implemented) often called the closed "list". What is its impact on the search from an efficiency perspective?

iii b) [1 points] Why do you suppose there is no explicit `closed` list in the code but there is a flag in each state instead?

iii c) [1 points] What happens to the speed of the grassfire algorithm if you remove just the test for being on closed (leave the test for "not on open" alone)? Do it to be sure and explain what you see. Set minExpansions to 1 so you can watch it in slow motion. Red arrows are open states and green ones are closed.

iii d) [2 points] What happens to the completeness of the grassfire algorithm if you remove this test on closed? Do it to be sure. [HINT: Consider both aspects of completeness. Enclose the robot start state in a wall of obstacles (after configuring and pressing reset) and see what happens].

iii e) [1 points] There <u>is</u> an explicit `open` list, so why do you think there is an explicit `open` flag in each state also. [HINT: look where it is read and consider why this might be better].

iii f) [1 points] Why does the above `open` flag actually hurt when the planner is used a second time for a different start and goal.

iii g) [4 points] Why is the removal of edges that penetrate obstacles necessary in Grassfire. Remove the associated code and see what happens if you are not sure.

3. PlannerState, Dykstra's Algorithm and Astar Planner [36 Points]

Grassfire is a very simple form of dynamic programming. As you may have noticed, its not particularly fast even though it is already quite efficient. Dykstra's algorithm is a more general implementation of dynamic programming. Astar adds the branch and bound algorithm to Dykstra's algorithm to produce the most famous path planning algorithm of them all.

a) [12 points = 9 Qs + 3 code] Dykstra's algorithm. Complete the empty or partially empty functions in the file `MotionPlannerContext.java` and answer questions as outlined below. Make sure the main program is set up like so:

```
boolean do8ConnectedTopology = true;
algorithm = Dykstra;
boolean executePlan = true;
boolean makeRandomWorld = false;
```

i) [3 points] Many motion planners can be used in continuous cost graphs where the edges have costs (also called weights) which may be different from each other. In this case, and in the rest of the homework assignment, obstacles are represented as regions of high cost <u>which can be driven over</u> if the cost is justified. In motion planning, we often compute the cost of edges as a line integral of a cost field along each edge. A line integral is an integral of the form:

$$c(x_0, x_f) = \int_{x_0}^{x_f} c(x) \cdot ds$$

In this assignment we will consider the cost field $c(x)$ to be an array of square cells c_{ij} <u>within</u> each of which the cost $c(x)$ is uniform. The uniform cost is potentially different for each cell in the array. You will need a formula in the next question for the cost of an edge between two <u>adjacent</u> cells with two different costs in them. Draw a figure and present a formula which computes the edge cost as a function of the two cell costs and the distance between their centers. It should work whether the cells are related diagonally or otherwise.

ii) [3 points] Complete the following two functions in `MotionPlannerContext.java`.

- `gValue(MotionPlannerState state, MotionPlannerState parent)`
- `distanceMetric(double dx, double dy)`

Also complete the following function in `MotionPlanner.java`.

- `expandDykstra(MotionPlannerState state)`

Make sure the `distanceMetric` works for both Manhattan distance (sum of absolute displacements in x and y) and Euclidean distance (square root of sum of squares) metrics[1]. You should be able to run Dykstra's algorithm now. Produce a screen capture of the right window for one obstacle (the world generated when `makeRandomWorld` is false) for both cases (Manhattan and Euclidean) where the screen has part of the search tree (arrows) drawn and all of the path drawn. Comment on the different shapes of the planning horizon in each case.

iii) [4 points] Dykstra's Algorithm is designed for continuous cost graphs - where all edges may have different costs. It differs from Grassfire in the sorting of the open list. Notice in the constructor for `MotionPlanner` that the open list is a `PriorityQueue` rather than a `LinkedList`. The former is automatically sorted and the latter is not. Give a reason why this sorting is necessary for Dykstra's algorithm [If you are not sure, change the constructor to use a `LinkedList` and see what happens to the insides of obstacles. Consider what this means to the quality of the plan].

iv) [2 points] Consider again the 8 connected grassfire result. Grassfire does not use a distance metric but you can always measure the length of a resulting path any way you like. Is the Grassfire result optimal under a Euclidean definition of distance for 8 connected topology? Why or why not?

b) [8 points = 5 Qs + 3 code] Heuristics and Comparisons. Complete the empty or partially empty functions in the file `MotionPlannerContext.java` and answer questions as outlined below.

i) [3 points] Complete the function:

- `hValue(double x, double y)`

ii) [2 points] Assume for this question that the cost field in the absence of obstacles is uniform throughout the search space so that edge cost through free space is proportional to the distance metric in use. Obstacles have a higher cost than "free" cells. For 8-connected topology, comment on the admissability of heuristics based on both Manhattan and Euclidean metrics. For our problem, is each admissable?

iii) [2 points] Use the same assumptions for the last question, particularly 8-connected topology. One heuristic is more informed than another if it is closer to the true cost between states. Suggest a more informed heuristic than Euclidean which is also admissable in an 8-connected state space.

iv) [1 points] Look at the function `compareTo()` near the end of the file `MotionPlannerState.java`. This function overrides the notion of < and > for MotionPlannerStates so that Java will sort the open list based on the f() value of the states it contains. What do you think is the rationale for the treatment of the tie-breaking case where the two f values are equal? Don't tell me what the code is doing. Tell me why it is doing this rather than nothing at all. A test for lower `h()` is equivalent to one for higher `g()` since the two `f()` values are already known to be equal.

1. Be careful from now on not to confuse the definition of distance (the "metric") from the topology of the graph. 2D rectangular lattices (graphs) like ours can be 4 or 8 connected and you can define distance between cells in either case as Euclidean or Manhattan.

c) [16 points = 12 Qs + 4 code] A* algorithm. Complete the empty or partially empty functions in the file `MotionPlannerContext.java` and answer questions as outlined below. Make sure the main program is set up like so:

```
    boolean do8ConnectedTopology = true;
    algorithm = Astar;
    boolean executePlan = true;
    boolean makeRandomWorld = false;
```

i) [4 points] Complete the function:

- `expandAstar(MotionPlannerState state)` - this is one of the hardest parts of the assignment but it is still < 10 lines of code.

You should now be able to run the Astar algorithm. Its fun to watch how smart it is in a number random worlds by temporarily turning on `makeRandomWorld`. Alternatively, you can turn on execution and click to add obstacles in the left drawing plane to block its path while it is driving. if you add to the left and then the right of a “wall” it will keep changing its mind about which way to go.

Now that is a cool algorithm.

ii) [6 points] What is the point of testing the `f()` values of nodes on the closed list and is the resulting path optimal if you do this check and propagate the new costs downward in the tree?

iii) [4 points] Consider the value of `greedyFactor` which is defined at the top of the main application and used in the method `hValue()` which is defined in `MotionPlannerContext.java`. It can be edited in the GUI. Why may it be valuable to configure a planner with `greedyFactor` > 1.0? In what kinds of environments does this work well? What is the cost of doing so?

iv) [2 points] What happens to some nodes on `closed` when the planner is configured with `greedyFactor` > 1.0?

4. Real-Time Astar [8 Points]

Of course, in a real situation, robots do not plan long paths and then execute them blindly. The environment may be changing (moving obstacles) and new information may be gathered while the robot moves whether the environment is changing or not. The technique of continously planning and executing is referred to as real-time *replanning*, or *interleaving* planning and execution. If sensor information is being gathered and it triggers replans, it is also known as *sensor-based* replanning. If replanning is unable to search all the way to the goal in the available time between executing actions, the search is called a *limited horizon* search.

You have probably noticed that the code is configured to execute one step of any plan and then plan all over again. However, here you will learn the value of this when there is insufficient time to plan all the way to the goal.

a) [8 points = 6Qs + 2 Code] Real-Time A* algorithm. Complete the empty or partially empty functions in the file `MotionPlanner.java` and answer quesions as outlined below. Make sure the main program is set up like so:

```
boolean do8ConnectedTopology = true;
algorithm = Astar;
boolean executePlan = true;
boolean makeRandomWorld = true; // NOTE true!
int maxExpansions = 50;
int minExpansions = 1; //
```

Consider the function `getBestHorizonNode()`. As supplied, it simply returns the goal node on the assumption that a solution was found. However, the need to make a timely decision often outweighs the desire for an optimal plan when there is insufficient time for the latter. As you probably are aware by now, the `open` list contains the horizon nodes of the search. Every node on `open` is a potential source of new unvisited nodes. Conversely, nodes on `closed` have already been expanded and are therefore surrounded by nodes on `open` or on `closed` or by cells in collision. Consider the process of:

- picking the node on `open` which is "best" in some way
- "backing up" the node to the root by following backpointers in the `extractPath()` function
- moving to the first state on the path to the best node.

This is accomplished by editing `getBestHorizonNode()` to return the best node to the path extraction routine.

i) [2 points] Complete the function:

- `getBestHorizonNode()` - Modify it so that it can be configured (use // comments to comment out one or the other) to return the state on open with the lowest `f()` or the lowest `h()` in the case <u>when the goal has not been reached</u>. It should return the goal state otherwise.

If you set `maxExpansions` to the small value indicated above, you should now be able to run the program in real-time replanning mode based on your implementation of `getBestHorizonNode()`. Run it several times until you feel you have seen most of the behaviors that result. Remember that you can add obstacles before clicking the "run" button (after a reset) in order to set up specific cases.

ii) [2 points] Note that when the environment is known and static, the action computed at any position in limited horizon real-time replanning is the same regardless of when the position is occupied. A vector field can be drawn which points to the next cell that would be occupied by the robot with such a planner for every cell in the search space. Oscillations therefore occur precisely whenever the robot occupies a state on a cyclic path in this vector field. In minimum `f()` limited horizon search the planner often fails to make it to the goal because it starts to oscillate. Describe one event which guarantees that the robot has just begun cycling back and forth in a trivial cycle of two cells.

iii) [2 points] Why is minimum `f()` limited horizon search so much more prone to cycles than minimum `h()` ?.

iv) [2 points] One somewhat effective technique to avoid cycles is for the robot to remember where it has been before and avoid going back. Is there any foolproof method when searching to the goal is probibited? Why or why not?

5. Dstar Lite [24 Points] (Bonus question)

Hopefully, it is now clear that planning half the way to the goal is a pretty risky approach. The Dstar family of algorithms is designed to efficiently "repair" the policy encoded in the search tree backpointers without redoing the search from scratch like the code has been doing so far.

In Dstar, an initial navigation function is calculated before the planner runs. Replanning is accomplished by continuous repairing this initial plan as new information becomes available. New information can become available in at least two ways. First, if the environment is dynamic, or if the perception sensor is noisy, changes in the map occur even if the robot does not move. Second, when the robot moves, new information is generated as the periphery of the sensor field of view.

which is unaware of any obstacles beyond the sensor radius.

a) [12 points = 6 Qs + 6 code] D* Lite algorithm for stationary robot. This question will walk you through a debugging exercise to make sure your Dstar implementation is working correctly. Set the code up like so:

```
final double USER_DS = 0.5;
algorithm = Astar;
boolean makeRandomWorld = false;
boolean executePlan = false;
boolean doCostMapDrawing = false;
boolean doNavFunction = true;
int minExpansions = 4500;
```

If you run Astar with the above settings and click to add or delete the central obstacle you will see a set of arrows flash in blue to indicate those cells whose costs or backpointers have changed as a result of the obstacle change. You can click the pause button to do a screen capture to save a record of the right answer. Your job is to make the same thing happen using Dstar when you add or remove an obstacle - which is a lot more efficient. The "random" obstacles produced by mouse clicks are actually repeated if you restart the program and click in roughly the same place.

Complete the empty or partially empty functions in the file `MotionPlanner.java` and answer questions as outlined below.

i) [6 points] Complete the functions:

- `expandDstar(MotionPlannerState state)`
- `updateVertex(MotionPlannerState state)`
- `getRhs(MotionPlannerState state)`

Note that the java idiom

```
for(State neighbor : myList){
}
```

will sequence through all `States` in the collection `myList` assigning them successively to the variable `neighbor`. Look up the java "for" loop for more on this construct which leads to short, clean, readable code.

Once you have the code ready to debug, turn on Dstar and rework your code and TEMPORARILY turn on doReverseDstar (ReverseD* in the GUI) until you get the same answer as A* when you add or delete an obstacle. You can set the variable `searchTreeDisplayTimeInMillis` in the main program to a number like 10,000 in order for the results to display on screen long enough to capture.

ii) [3 points] Do two screen captures as described above - one for Astar and one for Dstar that show the identical set of cells are updated when the same obstacle is added to the center of an empty world (i.e. delete it, then add it, then capture the screen).

iii) [3 points] Look at the function MotionPlannerState.hasChanged() which is used to determine which cells should have their backpointers flashed in blue. Explain the test for when a cell has changed from the perspective of how the test would fail if only one or the other condition were used.

b) [12 points = 10Qs + 2 code] D* Lite algorithm for moving robot.

Now we will move the robot in Dstar. As the robot moves, a simulated perception system becomes aware of those costs in the cost map which are within the maximum radius of the sensor, known as `DStarSensorRadius`. Up to this point in the homework, the maximum radius has been infinite but now it will be finite.

Complete the empty or partially empty functions in the file `MotionPlannerContext.java` and answer questions as outlined below.

i) [2 points] Complete the function:

- `updateHeuristicForMovedGoal(Point2D pt);`

This function cleverly updates the search tree to account for vehicle motion by changing the value of a single variable. Read the Dstar Lite paper for the reasoning behind this. You should now be able to turn execution back on and run Dstar lite in random worlds to get a sense for its behavior.

Make sure the main program is set up like so:

```
final double USER_DS = 0.5;
boolean do8ConnectedTopology = true;
```

```
algorithm = Dstar;
boolean makeRandomWorld = true;
boolean executePlan = true;
int maxExpansions = 30000;
boolean doCostMapDrawing = true;
```

ii) [4 points] Explain the behavior in the following case (i.e. why does it happen, not what happens). The robot is hemmed in by a "wall" of obstacles so there is no way to the goal without going through them. However, it explores the entire boundary of the wall of obstacles over and over before deciding to drive through it. Remember you can add or delete obstacle from random worlds by clicking on screen to make this case occur. Its best to do that before clicking "run". [HINT: The robot continues to explore BEYOND the point where it knows it is encircled by obstacles. Sensor radius matters.]

iii) [4 points] Based on your answer above, speculate on how you can make the robot more or less willing to "explore" or to enter costly unknown terrain. [Hint1: do not use the greedyFactor] [Hint2: You can probably test your ideas by modulating `obsCost` in `MotionPlannerCostMap` (also in the GUI)] .

iv) [2 points] Notice that when the goal is reached, the application layer prints how many cells have been traversed. Also, if you let Dstar run after it reaches the goal it will start over with memory of the map cells that were uncovered the first time. Hence, each re-run of Dstar is potentially more informed than the last. Observe this behavior in a few worlds, and/or conduct your own experiments and then comment on the relationship between the `DStarSensorRadius` (in the main program, not in the GUI) and the "exploration efficiency" defined as the ratio of the cells traversed to that of the optimal path if the entire cost map were known at the start.