CMRoboBits (15-491) Fall 2007 – Homework4

Due Wednesday, October 22th 2008 at the beginning of lab (12:30pm)

Part 1. Overhead Vision and Probabilistic Path Planning (RRT) (90%)

In this homework you will create a VPL/C# behavior that makes use of an overhead vision service we provide. Your program will have to localize your robot and determine its orientation based on colored markers. It should then allow your robot to reliably drive to a designated point using a closed-loop approach.

Additionally, there will be obstacles in the domain which will also be marked with colored patterns. You should implement an RRT (Rapidly exploring Random Tree)-based algorithm to compute a collision free path from one side of the field to the other. Once your robot reaches that side, it should make its way, back, looping back and forth forever without hitting any obstacles. The obstacles will be placed randomly and might move slowly between and/or during iterations, so it is crucial that you constantly re-plan as you go along. For the re-planning, you are to use the ERRT algorithm.

The Vision Service

On the lab machines, you will find a service called “VisionClient”. This service will automatically connect to the vision server located in the REL and return a list of colored regions (called Blobs).

If you need to install the vision client on your laptop, you can do so by simply extracting the zip-file provided at the course website to “c:\microsoft robotics studio (1.5)\bin”.

The only function you need to call from this service is “getBlobs”. The resulting list will be sorted primarily by color and each color will be sorted by size.

Each Blob will provide you the following information (very similar to the ColorSegmentation Service you already used):

- “color”, An integer determining which color-label the blob is (**see further below for explanation**)
- “area”, The area of the blob in pixels

- “cx”,”cy”, The x and y coordinates of the blob’s centroid (its “center of mass”).
- “x1”,”x2”,”y1”,”y2”, The x and y coordinates of the blob’s bounding box from x1,y1 to x2,y2

Note that the camera coordinates are in the range from (0,0) to (720,480).

Also note, that the vision coordinates are in camera-coordinates (e.g. the camera is not geometrically calibrated in any way). Since distortion is fairly small and the camera is mounted overhead, you can ignore these issues, and should assume that camera-coordinates are in direct linear relationship to world-coordinates.

The “color” integer is one of the following labels:

1 = “iRobot White”, the iRobot’s color
2 = “Scribbler Blue-ish”, the Scribbler’s color, since it is fairly de-saturated you probably don’t want to use it for anything, but instead rely on colored patterns
3 = “Yellow Paper”
4 = “Bright Blue Paper”
5 = “Orange Paper” **= Obstacle**
6 = ”Red Paper”
7 = “Bright Green Paper”

To use this service in C# you should be able to place it inside a custom activity, connect it to your activity’s input/output calling its getBlobs function, and then select “Compile as Service” as you did for the previous lab. This should give you programmatic access to the Vision Services result.

General Vision Hints

The Vision service will send all regions back at you which are greater than 8 pixels in area. Depending on noise and lighting conditions this could be a lot of different regions. You might want to perform additional filtering to make sure your Blob is large enough to really represent a robot. You can also look at its bounding box to make sure that its height/width bounding ratio is roughly square (don’t be too strict on your filtering though). Keep in mind that within each color, regions are sorted by size, so picking the first “n” of each color is usually a good strategy, unless you intend to use some more sophisticated tracking algorithm.

Obtaining position/orientation

The probably easiest way to obtain your robot’s position and orientation is by using two different colored markers on opposite sides of the robot (e.g. one blue, one green, see image further below). Your markers should be as big as possible without overlapping for giving you the best vision result. This should give you all the information you need to compute the robot’s position and orientation.

RRT path planning

You should implement your RRT-based planner in C#. You should know the positions of all your obstacles (which will be of color orange). As for a radius, they will be about the size of an iRobot Create, so you should use that plus your own radius plus a tolerance for your avoidance distance check. You will need to come up with some very basic "line-segment and circle" intersection algorithm. Note that if you can compute the shortest distance from a point to the line-segment, this will inherently do the same trick.

You should continuously re-run your planner as your robot drives along. Obstacles should be able to move while your robot is driving and your robot should adjust its plan accordingly.

The RRT algorithm, as introduced in the lectures, itself is as follows (in extreme pseudo-code):

```
Let your tree T consist of only the root, which is your current
     position.

While (GoalFound==false && iterations < MaxIterations) {
     Let x be a random value in the range from 0 to 1
     If (x  < alpha) {
          Let p be a random point, anywhere on the field
     }  else  {
          Let p = the goal point
     }
     Find the node n of the existing Tree T that's closest to p.
     Add a new node from n extending a constant distance d in the
     direction towards p. ONLY IF this extension will not hit any
     obstacles, add this new node to the Tree T, and make sure to
     store with it an index to its parent node n. If your new node is
     close enough to the goal, then you're done: GoalFound=true;
     iterations++;
}

Now, going backwards from the node that found the goal, you can back-
     track the final path by following its chain of parental nodes
     until you reach the starting node.
```

"alpha" in this case, is a parameter you chose. It determines your algorithm's probabilistic bias towards the goal. "MaxIterations" is another parameter that will limit your algorithm to plan a certain number of iterations and then give up if it hasn't found a goal. Otherwise, if no path exists, your planner would run on indefinitely. Your step-distance "d" should be a value that's small enough for

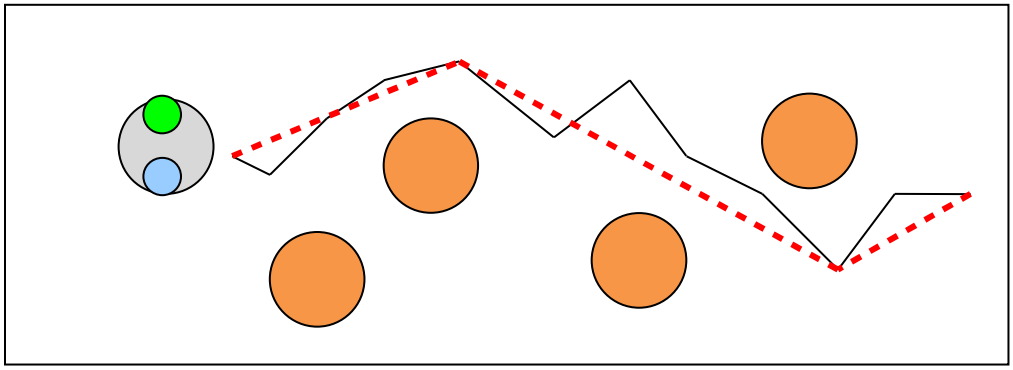
accuracy, but large enough for speed. A value in the range of a few cm (or pixels respectively) should suffice.

ERRT planning

Because your robot will need to do re-planning, you should extend the above algorithm to an ERRT by adding a third probabilistic possibility besides letting p be a random point or the goal. In this third case, p would be a random point from the previous iteration's successful path (if one was found). Depending on your probability for this option, your algorithm will be biased much more strongly on re-finding a path it has previously found.

RRT path smoothing (optional)

Keep in mind that RRT might return some very edgy and non-optimized plan. It thus is a good idea to "smooth out" the path using a very simple technique. In essence, you should check from your current position (the root node) whether you can reach a node beyond the next planned node by driving to it in a straight line without hitting any obstacles. If you can, then try a straight line from your current node the very next one, etc. This will smooth out any unnecessary edges of your existing path, as depicted below (the red dotted line is the smoothed out path):



Demo

You will need to demo your path planning behavior at lab-time. As mentioned before, your robot should go back and forth from one side of the field to the other, while there are **Orange (color ID 5)** obstacles on the field. These obstacles might slowly change configuration between runs.

Submission

Your entire homework should be submitted by copying it into the "**dropbox/lab04**" folder on your personal AFS space before the due time. **Make sure to submit your C# code as well!**

Write-Up

Please also submit a write-up for this lab (one write-up per group is fine). In it, describe what internal data-structures you used in your C# code and which path planning algorithm you implemented.

Part 2. Write-Up about the Course and Robotics Studio (10%)

In addition to your normal write-up that you will submit for part 1, we would also like you to quickly give us some feedback by answering the following three questions:

1) Learning/Course

Please give us some feedback about your course experience, in terms of learning the main concepts of perception, cognition, and action in robots. Refer also to your experience in the Labs - have they been too hard, too easy, or just right, at the conceptual, and at the implementation levels. Any comments to support your opinions will be particularly appreciated. You may want to mention also what you would be most curious to learn about for the remaining of the course. Furthermore, feel free also to make suggestions of projects that you may be interested in making, for the final evaluation.

2) Robotics Studio Bug Reports (for all homeworks)

Please let us know about any particular Robotics Studio Bugs, especially if they are grave and/or reproducible.

3) Robotics Studio Annoyances / Improvement Ideas / Feature Requests (for all homeworks)

If you have any particular ideas on how to improve Robotics Studio in a way that would make future students' lives easier, please let us know.

Please place your answers to these three questions in a separate file called "eval.txt" in your "dropbox/lab04". Thank You!