“CheeseFinder” Progress Report

Sashi Kumar, Cody Addison, Oluwasoji Omiwade, Dr. Albert Cheng (Advisor)

University of Houston, Department of Computer Science
{cody.addison, yeskaym}@gmail.com, ooo00a@yahoo.com, cheng@cs.uh.edu

Abstract: We discuss what algorithms and procedures we use to find to make our robot, CheeseFinder, locate the beacon in a given map for the Ciber-Rato competition. We have found that algorithms like that used by Sutherland [1] apply more to powerful obstacle sensors, unlike the ones used in this competition. We use an algorithm like that of Lumelsky’s [2] proposed by Sankaranarayanan. Noise in sensors and motors is a problem that we are still giving much attention.

1. Introduction and Initial Plan

In our planning phase when we had our first discussions as to what algorithms to use to make the robot find the beacon, we initially agreed upon using Sutherland's algorithm [1]. That is, we would identify spurs (shown in Figure 1), points defining spurs like **P1** and **P2**, openings and dead ends, in order to help us create a map of the whole terrain. After identifying spurs, we would use Moore's algorithm to find the shortest path from our starting point to the cheese and then back.

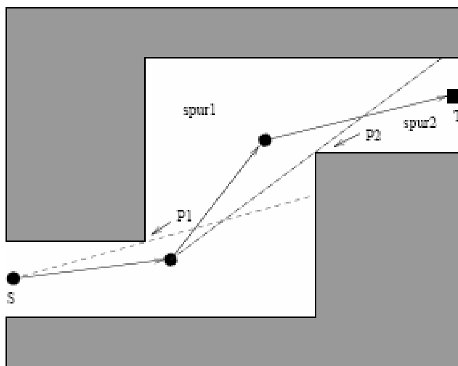

Figure 1: Using the notions of spurs, Sutherland lets the robot learn the whole terrain. Once as many spurs as necessary have been identified, the robot moves from one spur to a neighboring spur and does not hit any obstacles.

2. Refinements

For one main reason, we have decided that such an approach may not be appropriate for this real time systems programming competition. The primary reason why an algorithm like Sutherland's is not appropriate is that we have chosen to approximate the obstacle sensors as touch sensors as opposed to vision sensors. By this we do not mean that we use the bumper to crash into walls in order to determine that a wall is present. Instead, we have noticed that the obstacle sensors begin to trigger only when a robot is very close to the wall. If the robot is not very close to the wall, then the robot cannot detect the obstacle. And so the algorithms that we use are touch sensing algorithms, even though we never touch the walls.

With this approach, we have noticed very good results. Our robot is able to find the beacon assuming the noise in the robot motors and sensors are not large (We are still working to accommodate large noises). Concerning noise, the hardest problem right now is figuring out how to navigate the robot to the beacon with the minimum number of obstacle collisions. Of course we could decide to travel at a slow constant speed, in which case, the values from the obstacle sensors are most reliable. But if we move too slowly, then some other robot moving at a faster speed that *varies* the speed of the robot, with respect to the proximity to a wall, will reach the beacon before us.

We therefore must vary the speed of the robot according to the probability that there is a wall in front of us. Since hitting any obstacle does not mean the end of the game, then we can afford to take risks at times. By risk, what I mean is the following: If the noise level is not sufficiently large, and it seems likely that we have no choice but to go through a given tight opening, then we will take the risk. If our bumper then senses that we hit something, then we will mark that in our map that we cannot go through that area, and try to find some alternate means.

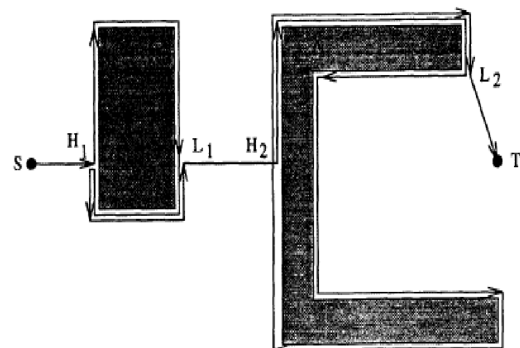

Fig 2: Here Lumelsky goes round every obstacle that he comes across. A detailed algorithm in pseudocode form is given in [2].

3. Implementation

The algorithm that we choose to implement is based on Lumelsky's maze algorithm. Lumelsky's algorithm basically makes the robot go round every obstacle that is on the path between the beacon and the robot. There are two advantages to using such an algorithm. The first advantage is that for each obstacle that a robot goes round, the robot has more knowledge about the entire maze than it did before exploring the obstacle. The second advantage with this approach is that sensor requests are minimal. Other schemes like the scheme we intend to use will request more sensors than Lumelsky's approach. An illustration of how Lumelsky's algorithm works is shown above in Figure 2. *S* is the robot's starting location and *T* is the beacon location.

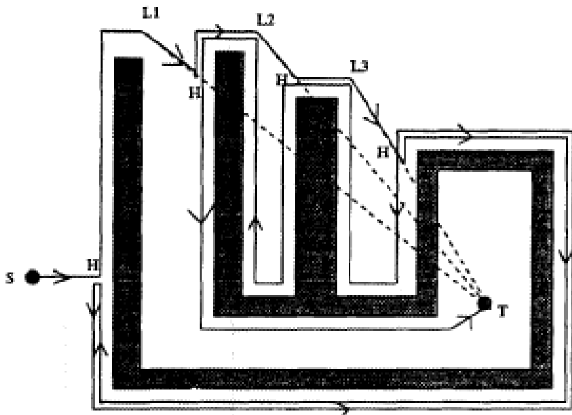

Figure 3: Execution of Sankaranarayanan and Vidyasagar's algorithm to solve the beacon finding problem.

Our algorithm is that of Sankaranarayanan and Vidyasagar. It is a modification of that of Lumelsky. An illustration of how it works is shown above in Figure 3.

1. The robot moves along a straight line as in Lumelsky's algorithm toward the destination *T* until it comes across an obstacle.
2. This time however, the robot leaves the obstacle boundary at a point *L* if and only if the following two condition are satisfied: (a) at *L*, the robot can move along a line segment *LT* that does not enter the obstacle, and (b) *L* is closest to *T* among all *x* ever visited by the automaton prior to visiting *L*. In Lumelsky's algorithm the robot left the boundary only once it had traversed the whole obstacle. Here lies the advantage of Lumelsky's approach over Sankaranarayanan: Sensors are queried less with Lumelsky's. However as we will see, we still cannot use Lumelsky's algorithm because it does not seem to be optimal.
3. If a previously defined hit or a leave point is met then the following rule is applied: if returning to a hit point *H*, the robot moves along the unvisited section of the obstacle boundary, which starts at *H*. Algorithm adapted from [2].

In this algorithm the worst case length (longest distance) of the path generated is greater than the worst case length of Lumelsky's algorithm. One would ask why we want to use such an algorithm if we know that Lumelsky's algorithm guarantees a shorter worst case distance. We have found out that the worst case distance of Sankaranarayanan's approach (the one we implement) is more because in addition to going round the entire obstacle, we must also add the distance traversed when we try to get to the beacon using the shortest path possible.

More important to us is the fact that it seems strongly intuitive that Sankaranarayanan's approach on average would perform better than Lumelsky's. With this in mind we use that of Sankaranarayanan to lead us to the beacon.

4. Hidden Beacon; High Walls

We create a map of the whole terrain. The idea behind how we create our map can be found in [3]. The created map is four times the size of the map specified in the Rules and Specifications Guide of the Ciber-Rato contest. We then assume that our starting point is always

(0,0), which is the center of our created map. The motivation behind creating a map that is four times the size of the original is that no matter where we are placed on the map, our map will always *contain* all of the original map.

Each unit in our matrix (i.e, map) is 0.1 units. We have judicious reasons for making these units so small. Based on the values returned every time we request the beacon sensor, we always fine tune the exact location of the beacon. And so whenever the beacon is hidden by a high wall, we can consult our map, to remember the beacon's approximate location based on our previous calculations.

References

1. I. Sutherland. A Method for Solving Arbitrary Wall Mazes by Computer. *IEEE Trans. On Computers,* C-18(12): 1092-1097, 1969.
2. N. Rao et. al. Robot Navigation in Unknown Terrains: An Introductory Survey of Non Heuristic Algorithms. *Oak Ridge National Laboratory*. 1993
3. Pedro Ribeiro. YAM (Yet Another Mouse) Um Robot Virtual Com Planeamento de Caminho a Longo Prazo. *Revista do DETUA,* Vol. 3, Nr. 7, pp 672-674, Setembro de 2002.