Final Project

Jurassic Park

Larry Freil, Farzon Lotfi, & Alex Strange

4/30/2011

Table of Contents

Part 1: Escaping

Question 1 overview:

There were several alternative options considered for the parts of the algorithm. One was to use the approximate cell decomposition algorithm we implement in project 3. This algorithm might have been able to handle moving obstacles more easily, because we could have merely updated each grid cell whenever an obstacle moved in and out of it. Then we could have just checked to see if any of the cells it moved into were part of our desired path; this would quickly tell us if we needed to re-plan. With the visibility graphs, each time we want to re-calculate our obstacles, we have to re-generate the visibility graph, and therefore have to re-plan every time. However, this re-planning was fairly quick, and it took much less time during that first run to generate the first plan, so overall visibility graphs were much faster, which would hopefully allow us to avoid the enemy robot more easily.

The other options were in the execution of robot movement. In project 3 we used an algorithm that attempted to line us up in a straight line towards our desired point, and then travel in a straight line till it reached the point. However, this algorithm was not robust enough to handle the robot traveling in the slightly non-straight line that usually occurred, since Rovio has an extremely imprecise response to turn commands. We decided to change this algorithm for the third project, to one that would allow the robot to drive sideways, which would reduce a lot of our uncertainty in turning, and to continually try to check and update the direction it is traveling in, even on the straight line path. This allowed us to reliably get from the current location and target location.

Improvements from the previous project

For this project there was a lot we wanted to do different from the first 3 projects. The most important step was to rework our algorithm for orientation and movement to allow more degrees of freedom. We noticed from LOLCodes success in project three that we needed to use more of the utility that the Rovio provided us. There was another, even more important reason - in project three we realized that even with re-planning, our attempts to detect if we turned the right amount meant that we could never go the angle we intended. Both of these failed because blob detection is not perfect which causes noise to throw off your orientation direction enough that you can't tell if you overshot the angle. To fix this we made turning a much less prevalent part of our code and decided to use moving right, left, and diagonal a lot more. To make our

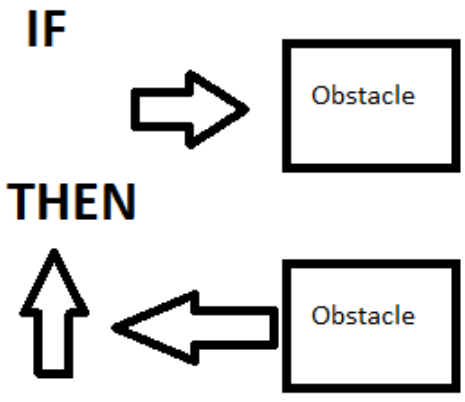

Figure 1: This example represents how we would get Arrows depict robot movement

movement more robust we also tried to analyze when the Rovio was stalling or moved out of the picture. We did this by storing the current point globally and after each movement checking if the pointed had moved out of our target range. If the target range was not achieved we would move the opposite perpendicular vector/direction from that point.

Theoretical algorithms we implemented but did not use

Using knowledge from taking CS3600 last fall we came to the conclusion that a good algorithm to implement for this project would be a game tree.As such we made the assumtion that our enemy would pick the most optimal path. The first problem we encountered was those algorithms are really best for a more defined world. So we thought about switching from our visibility graph back to our approximate cell decomposition algorithm from the first project. While this would have solved the first problem, it came nowhere close to solving the second problem, which is that game trees like the alpha-beta tree we used assume optimality of the enemy and complete control of the user. Everytime we made a wrong move we would have to rebuild the tree. Furthermore after seeing some of the robots the night before the demo we realized that an assumption of an optimal enemy would be flawed. And thus we decided to go with more simple algorithms but keep the path weighting aspect of alpha-beta.

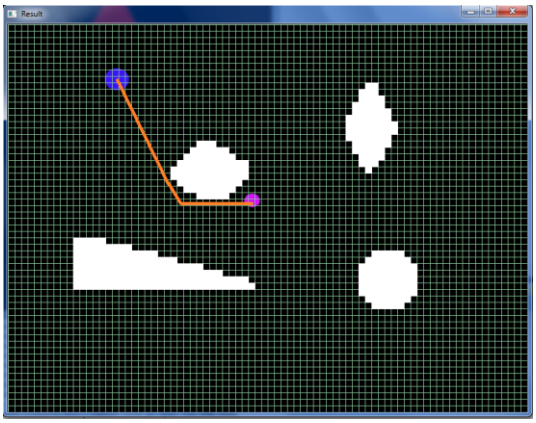

Figure 2: example of our approximate cell decomposition running with alpha beta

Alpha-Beta Tree psedo-code

```
class AlphaBetaAgent:
  def alphaBeta(self, dNode, alpha, beta):
    LActions = dNode.getGameState().getLegalActions(dNode.getAgentTurn())
    #removing STOP
    if dNode.getAgentTurn() == 0:
        if Directions.STOP in LActions:
            LActions.remove(Directions.STOP)
    #terminal test
    if (dNode.getGameState().isLose() or dNode.getGameState().isWin() or
dNode.getDepth() == 0):
        return self.evaluationFunction(dNode.getGameState())
    else:
        #our robot case
        if (dNode.getAgentTurn() == 0):
            for action in LActions:
                if  alpha >= beta:
                    return alpha
                nextNode =
```

```
decisionNode((dNode.getAgentTurn()+1)%dNode.getAgents(),dNode.getGameState().gene
rateSuccessor(dNode.getAgentTurn(),action), dNode.getDepth()-1)
                value = self.alphaBeta(nextNode,alpha, beta)
                if value > alpha:
                    alpha = value
            return alpha
        #their robot
        else:
            for action in LActions:
                if  alpha >= beta:
                    return beta
                nextNode =
decisionNode((dNode.getAgentTurn()+1)%dNode.getAgents(),dNode.getGameState().gene
rateSuccessor(dNode.getAgentTurn(),action), dNode.getDepth())
                value = self.alphaBeta(nextNode,alpha, beta)
                if value < beta:
                    beta = value
            return beta
  def getAction(self, gameState):
    #robt = 0
    alpha = -float("inf")
    beta = float("inf")
    LActions = gameState.getLegalActions(0)
    if Directions.STOP in LActions:
        LActions.remove(Directions.STOP)
    for legal in LActions:
        dNode = decisionNode(1,gameState.generateSuccessor(0,legal),self.depth)
        value = self.alphaBeta(dNode,alpha, beta)
        if value > alpha:
            alpha = value
            bestAction = legal
    return bestAction
```

The simpler algorithms

For our goal-finding algorithm, we decided to implement a simple method that selected a goal that was both nearby and far away from the enemy, planned the shortest path using a visibility graph, and then followed the path with occasional checks to see if the enemy robot was obstructing us.

The first part of the algorithm was to select which goal we should travel towards. In order to do this, we first calculated the Euclidean distance between ourselves and the goal, then calculated the Euclidean distances between the enemy and the goal. From this we tried to choose a goal which was far from the enemy and close to us. So, if we were right next to a goal, and the enemy was a little further away, we would still travel to this goal. However, if the enemy was blocking the goal, and therefore closer to it than we were, we would select a farther goal in order to attempt to avoid them.

The second part was to use a visibility graph to determine the desired path around obstacles that would get us to the goal. We chose this particular algorithm because, although we had also implemented approximate cell decomposition in the previous project, the visibility graph took significantly less time to calculate. Therefore, since we wanted the option to recalculate several times, we wanted the algorithm that completed more quickly. Also, since the visibility graph should give us one of the shortest paths (we added a buffer space around each obstacle in an attempt to avoid collisions, so skirting closer to these obstacles would give us a shorter path) we believed this would give us the best possible chance of getting to the goal before the enemy got to us.

The third part was to actually follow the path and recalculate the path if necessary. This part was achieved because our visibility graph returned us a set of points on the total path we needed to travel to in order to reach the goal, so we could create a simple algorithm for our robot that traveled in a straight line between these points. Once we reached a point in the path, we checked to see if the enemy robot had gotten in between us by rechecking their distance. If the enemy had gotten ahead of us, we switched our intended goal, and re-planned to the other goal. If the enemy was not in the way, we just traveled down the next segment of our path. We repeated this process until we reached the goal.

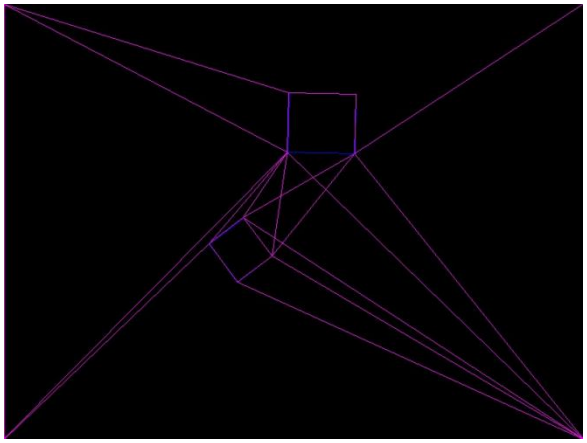

Figure 3: Example of our visibility graph

Why we choose this algorithm

The reason why we chose this particular algorithm is because we believed that it best balanced the trade-offs of speed, accuracy, and optimality. Because we chose a faster planning method, we had to lose the option of easily and accurately calculating interceptions by the enemy robot, but we could recalculate much more quickly, which allowed us to actually check frequently. We chose planning algorithm also because it would give us a close to shortest path to the goal, while still avoiding the obstacles correctly. As for optimality, if you assume optimal to be the path that gets you to the goal most quickly while still avoiding the enemy, then our algorithm should achieve close to these results. The main loss of optimality is that we don't calculate the lengths of the paths that avoid obstacles to either goal, just the straight line distances. The same goes for when we are calculating the enemy's distances from the goal. However, we believe that our algorithm gives a close approximation of the optimal plan/path.

Why we did not choose the other algorithms

As mentioned above we decided against the Alpha-beta tree simulation for two reasons. The first reasons were the high upfront time cost approximate cell decomposition the second was the exponential growth in memory and time of alpha beta trees even with pruning. We realized that this would not be as feasible as we initially hoped because of error in our movements as well as the enemy

movements. Other reasons that factored into this decision were flexibility and creativity. If we could design our own algorithms instead of constraining ourselves to what's already been implanted we thought we might have an advantage to surprise other teams.

Improved movement algorithm

```
void giveOrders(CvPoint* point) {
		if(!running){
			savePoint = *point;
			running = 1;
		}
		if( !(point->x >= savePoint.x+5)||!(point->x <= savePoint.x-5) &&(!(point->y >= savePoint.y+5)||!(point->y <= savePoint.y-5))){
 //this code sees if the robot point has moved out of the threshold for changing directions
// if it has not it first goes orthogonal to the previous direction then  goes opposite the previous direction. Then returns
}
   int counter = 0;
   double temp_angle = Angle;
		if(( (temp_angle >= -110.0) &&  (temp_angle <=-80.0) ) || ((temp_angle <=280.0) && (temp_angle >= 260.0)))
		{
			rovio_forward(4);
			moveMessage = 'f';
			return;
		}
		if(( (temp_angle <= 110.0) &&  (temp_angle >= 80.0) ) || ((temp_angle <=-260.0) && (temp_angle >= -280.0))){
			rovio_backward(4);
			moveMessage = 'b';
			return;
		}
		//225 degrees
		if(((temp_angle >=-145.0) && (temp_angle <=-125.0)) || ((temp_angle <=235.0) && (temp_angle >= 215.0))){
			rovio_DiagForRight(4);
			moveMessage = 'e';
			return;
		}
		//315 degrees
		if(((temp_angle >=-55.0) && (temp_angle <=-35.0)) || ((temp_angle <=325.0) && (temp_angle >= 305.0))){
			rovio_DiagForLeft(4);
			moveMessage = 'k';
			return;
		}
		if(((temp_angle >=-10.0) && (temp_angle <=10.0)) || ((temp_angle <=370.0) && (temp_angle >=
```

```
350.0))){
                        rovio_driveRight(4);
                        moveMessage = 'r';
                        return;
                }
                if(((temp_angle >=-190.0) && (temp_angle <=-170.0)) || ((temp_angle <=190.0) && (temp_angle >=
170.0))){
                        rovio_driveLeft(4);
                        moveMessage = 'l';
                        return;
                }
   temp_angle = abs(temp_angle);

                while(temp_angle >=0) {
      temp_angle-=20;
      counter++;
   }
  if ((side == DirLeft) || ((Angle <= 90) && (Angle >= -90)) || ((Angle <=  450) && (Angle >= 270))){
                        rovio_turnLeftByDegree(counter);
                        moveMessage = 'T';
                        rovio_forward(4);
                }

                else{
                        rovio_turnRightByDegree(counter);
                        moveMessage = 'Y';
                        rovio_forward(4);
                }
                savePoint = *point;
}
```

Full Algorithm

Pseudocode:

Select Goal:

Calculate the Euclidean distance between the center of our robot and each goal.
Calculate the Euclidean distance between the enemy robot and each goal.
for (each goal)
if(our_distance < their_distance):

if AddPathWeights(select goal) < AddPathWeights (select other goal)
select other goal.

else
selectGoal
else:
select other goal.

travel_to: target point
Calculate robot position and orientation using camera.
Calculate relative orientation to target point (the number of degrees we would need to turn to face this point).
while(not near target point):
Call giveOrdersFunction above and give it the current robot point.
Once at point, return.

Follow Path:
Using the obstacles, generate a visibility graph.
Using the selected goal and the visibility graph, find a path as a series of verticies between you and the goal
For(each vertex along path)
travel_to(vertex)
if(select_goal selects a different goal):
restart follow path algorithm.

How the algorithm performed.

Under the conditions we tested under our algorithm worked very well. These conditions included obstacles of varying shapes, sizes, and colors, and moving enemies of a different color than our robot. The main difficulties that emerged occurred when the enemy had the same coloring scheme as us, or when our robot or an enemy robot hit an obstacle of a similar color to our robot, especially if the obstacle moved. This caused the obstacle to appear after background subtraction, which threw off our blob detection. Luckily it was not as damaging to our orientation algorithm because we only ran our orientation algorithm when both of our colors were detected and intersecting circles could be drawn. Even so lighting conditions at the demo because of the chaired arena made the entire state space darker which made color detection much harder. As a result there were frames were we did not detect any an entire side of our robot causing our robot to stall. Because of these unknown variables before the demo our robot did not do as well as in our timed trial runs. Watching our videos that will be sent in with this report will show the effectiveness of our algorithm under the same conditions as the first three projects. The graphs below show the time trials of our robot running the above algorithms to reach the goal under conditions with and without an enemy robot.

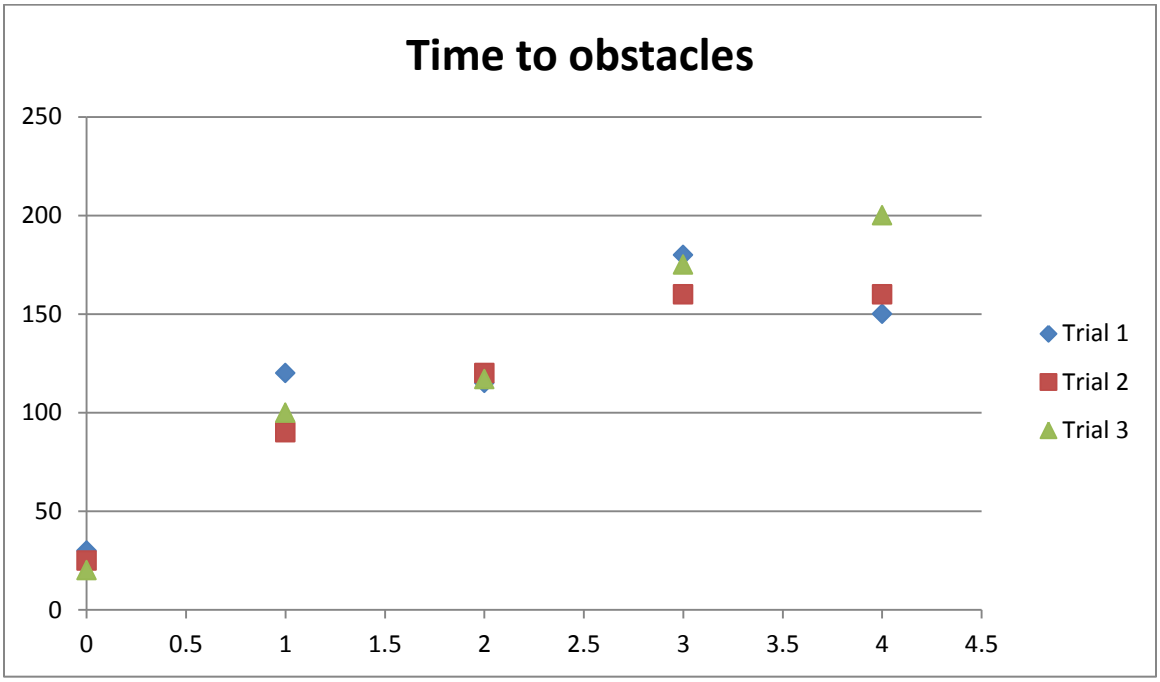

Figure 4: The above graph represents the ability of our algorithm to decide on a goal based on the number of obstacles. The maximum number of obstacles we tested was 4 with the minimum being 0. The time scale for reaching was at worst 3 and a half minutes and at best 20 seconds. It's important to note that this data does not represent the time when an enemy is present. It's also important to note that the obstacles were of variable size and that their colors were never the same as the robot.

Table of data

Obstacles	Trial 1	Trial 2	Trial 3	Trial Averages
0	30	25	20	25
1	120	90	100	103.33
2	115	120	117	117.33
3	180	160	175	171.66
4	150	160	200	170

Figure 4: The above time is represented in seconds

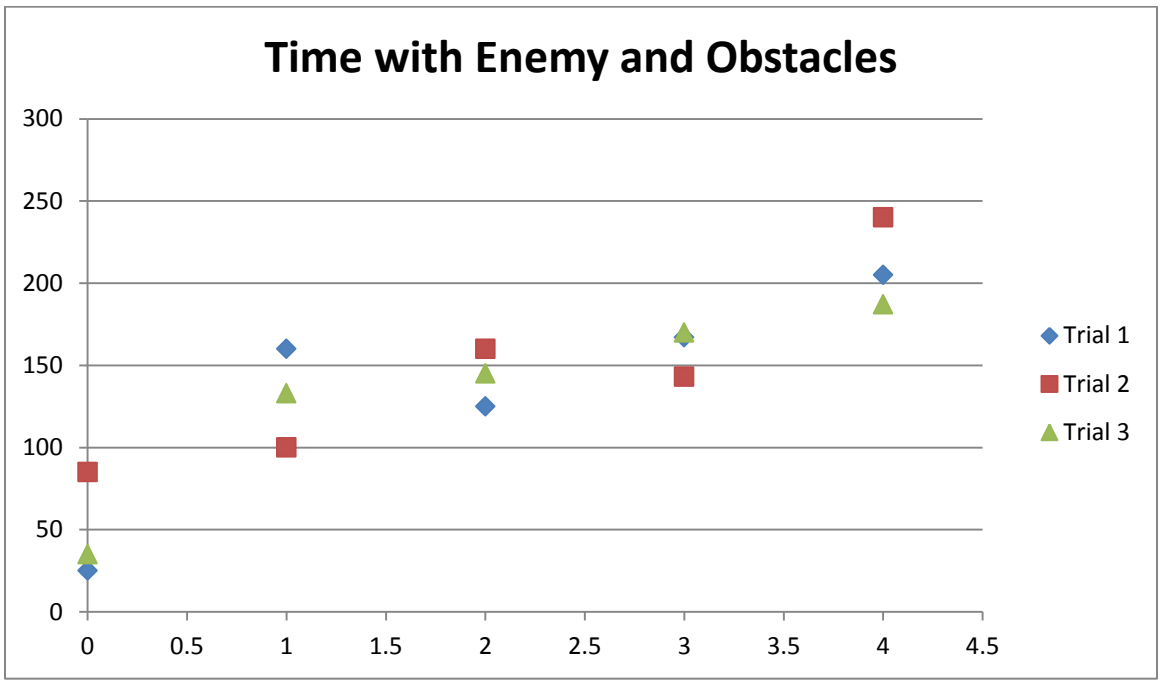

Figure 5: The most noticeable trend in the above data is how much farther off all of the data is from trial to trial. We figured this out after the demo. After we construct our path using the visibility graph, we add weights to each position of the graph based on how close the enemy robot is from us. We do this to determine if we need to re-plan. Unfortunately it resulted in our robot literally being to scared to move for long periods of time until it realized that the enemy robot had moved. We also built in some fail safe movement code that would use a combination of dead reckoning and obstacle collision to try and get to the closest goal in the event that we lost orientation (this would only happen if we hit an obstacle or a robot of the same colors as our own).

Table of data

Obstacles	Trial 1	Trial 2	Trial 3	Trial Averages
0	25	85	35	48.33
1	160	100	133	131
2	125	160	145	146
3	167	143	170	160
4	205	240	187	210.66

Figure 6: The above time is represented in seconds

Future Improvements: Getting to Goal

If we had more time to work on the algorithm, the first thing we would do is work on improving the vision for case where we hit an obstacle. This was a major problem for us in the demo because when we or the opponent hit obstacles we would have blob detection of our robot on obstacles. Our orientation algorithm was smart enough to try and only detect the robot when both of its sides were detected in the image. Still, there were instances where we failed to detect one of our colors on our robot which would cause us to loose orientation for long enough that we would stall. If we could detect the robot more reliably, we would achieve much greater performance. The first step to this might be trying to avoid detecting the robot by colors of any sort, and try to rely on the shape of the robot, or of the design we covered the robot with. Doing this might allow us to detect the robot in even the poorest of lighting conditions, allowing us to more easily control the robot. Also, we could have implemented something that would make sure that the halves of the robot we 'detected' were right next to each other as additional insurance that we had detected the robot correctly. Something like a Kinect that gives cloud laser imaging might make this easier, but learning some shape classification algorithm might work just as well.

The second part of the algorithm we would improve would be the planning around the enemy. If we modified the goal selection algorithms to plan a path between the each robot and each goal, and then used the lengths of these paths to decide which goal to select, it would allow us to guarantee we were close to the goal than the enemy. Because we were a Rovio group, we generally assumed that we could probably travel any path as fast as any of the other robots on the field, so we didn't have to worry about them getting ahead of us too much if we had less distance to travel. Also, we would have updated our path planning algorithm to maybe try less direct paths in favor of paths that take us further from the enemy robot.

Other improvements we would have liked to include, if we had time, would be to figure out a way to implement some kind of Bayesian network with particle filtering in order to better predict the path of our enemy. Our current algorithm used use blob detection to find the enemy then had a 5 element array/list of cvPoints – in every frame we added another point to this list, and when the list fills up we would have a predicted path; based on intersections with our path to the goal and its nearness to our robot we would add weights to our path which would be grounds for re-planning if our threshold for re-planning was met. Then we would clear the list/array. This resulted in problems where our robot would freeze and was not always accurate with Rovios because of the many directions they can go. Thus it was possible for a robot to go right and trick our algorithm into thinking it was oriented in that direction. With particle filter it might have been possible to have real- time analysis not dependent on orientation of the enemy robot.

Part 2: The Third algorithm -Playing defense

Our third algorithm was the one we used to defend against the enemy robots. Because of implantation time, and the uncertainty of the enemy's planning algorithm we chose the simplest option of just planning and following path to the enemy robot. After each portion of the path, we would re-plan the shortest path to the enemy robot.

Options considered

We considered several other options before selecting this algorithm. One option was to plan a path to the goal that the robot was most likely headed towards. This would make it easier in that we would haven't to keep re-planning the path to account for the enemy robot moving. This could save time in planning, but it could be more complicated trying to pick the correct goal, and determining how far in front of it to wait. Also, it could be possible that we might initially move towards the wrong goal if we guessed the incorrect goal.

A second option we thought of was trying to calculate the most likely plan between the enemy robot and its best goal, and then try to calculate a path between us and some midpoint on their path. This would avoid making us always travel to the goal, and instead just try to get us somewhere in between them and the goal. We would still be blocking, but hopefully have to spend less time and distance traveling around the map. The downside to this, is we cannot guarantee which goal the enemy is heading towards, much less the path they will choose to get to that goal. If we pick the wrong path, we could find ourselves on a path the enemy robot won't take, and in order to detect and prevent these cases, we would have to re-plan almost constantly in order to make sure that they don't start traveling a different path.

Why this algorithm

In the end we chose the simple chasing the enemy robot algorithm because it removes a lot of the uncertainty about the expected plan of the enemy robot. Since we can't know exactly what planning algorithm, or goal selection criteria, the other robot is using, we cannot perfectly predict the enemy planned movement. This could end up in our robot wandering around the map continually trying to re-plan. This could be less effective overall than just chasing down the enemy robot. This algorithm should only average give a relatively optimal solution. This is because though we may not always find the shortest path to block the enemy robot, it won't ever accidentally plan a path to the wrong end of the game area, and have to travel significantly further in order to correct later. Also, with this algorithm we would hopefully earn the points for eventually running into the enemy robot.

Pseudo-code:

Find enemy:

When using background subtraction for any blob of pixels that is not our robot or an obstacle.
If (only one blob not us or obstacle):
    Select blob as enemy robot.
else:
    Track each blob for 5 frames.
    Select blob that moved the most over 5 frames as enemy.

Defending:

Find enemy.
Using visibility graphs, plan path to enemy.
Move to first vertex on path (using the movement algorithm from the previous part of this project).
if(not at enemy):
    Restart algorithm.
else:
    Finish.

How it performed

This algorithm performed well in that it would always reach the enemy robot eventually. Since we only followed the enemy, there was a chance that the enemy robot would make it to the goal anyway before we reached them. However, we figured that overall, this would be the most likely way to touch the enemy before they reached the goal. The main times our algorithm failed completely was when our vision detection of the enemy robot failed. This rarely occurred unless their color scheme matched our own. If we couldn't find the enemy robot we couldn't know where to plan our path to. This sometimes resulted in us having to just travel to the last known location of the robot, and hope we rediscovered it at some point. Also, in one or two cases, even with the tracking the movement of the unknown blobs, we would sometimes still detect an obstacle that got moved as the enemy robot. This would cause us to chase down an obstacle instead of the enemy robot.

The data below Is based off of reaching an enemy robot that spins around in a circle. For the most part the robot is immobile. As few of the other groups wanted to spar, moving our second robot around in a circle was as close to the real thing as we could get. We had a couple of off trials where the robot both spun and hit an obstacle which screwed up a couple of frames and made it take longer to find the robot. In other cases we got lucky and found it quickly. There was also more variability in this trial than the

previous two we think this probably has more to do with the difficulty of hitting a moving object is much higher than that of a still one. Other factors could have been that the obstacles where never set up exactly as they were for each previous run.

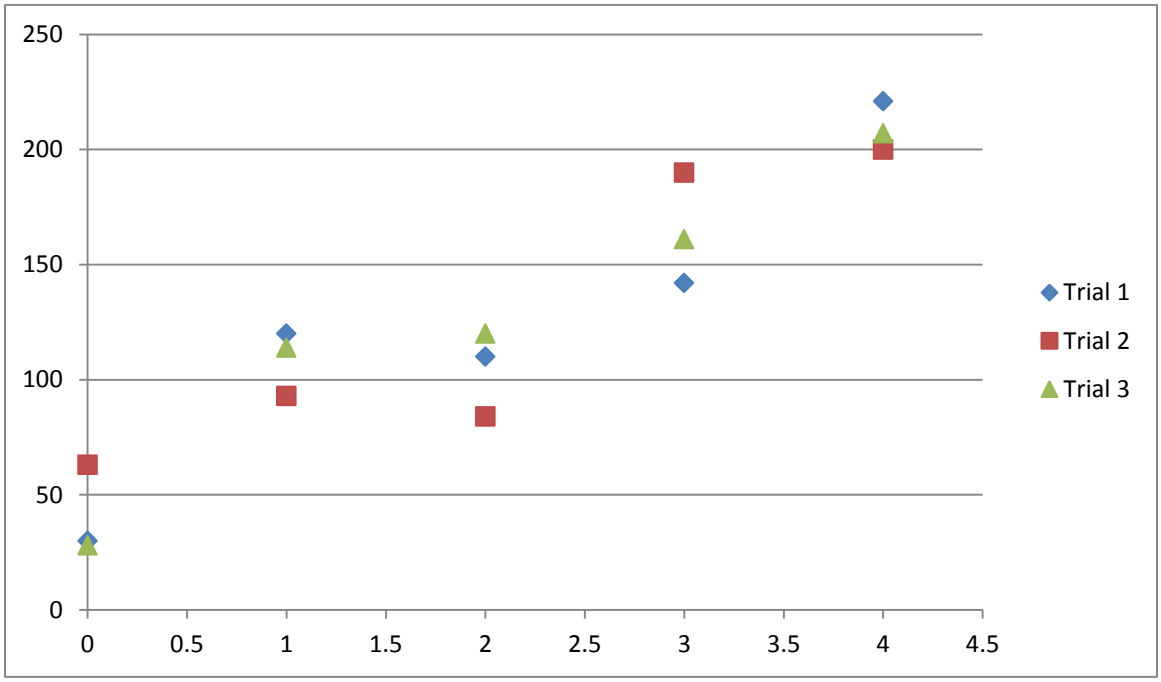

Figure 7: The data bellow represents only the successfully completed trials

Table of data

Obstacles	Trial 1	Trial 2	Trial 3	Trial Average
0	30	63	28	40.33
1	120	93	114	109
2	110	84	120	314
3	142	190	161	164.33
4	221	200	207	209.33

Figure 8: the numbers above are time in seconds

Future Improvements: Attacking Robot

If we had more time, we would have probably implemented some sort of path tracking algorithm for the enemy robot. If we could track their position over several frames, we might be able to determine their most likely path (or provide additional weight to their most likely path) and using this we could then plan an intercept path with much greater confidence. Also, if we could more reliably handle moving obstacles, we would have liked to implement an algorithm that would have tried to move lighter weight

obstacles (using a trial and error strategy to see which ones could be moved), to try to block off the goals because many of the enemy robots would either have difficulty re-calculating their paths if the goals moved, or would have difficulty finding a path around the obstacles because we believe most of the other teams tried to avoid obstacles at all costs, even if they might be moveable. However, as with the first part of the project, given more time we would have worked much more on getting our vision detection of our robot to be much more reliable under all lighting conditions.

Video

http://www.youtube.com/watch?v=ZOO36BOUfes&feature=channel_video_title

http://www.youtube.com/watch?v=fJdwRdgYH8I&feature=channel_video_title

http://www.youtube.com/watch?v=14_B97a-wbI

References

http://cbcl.mit.edu/publications/ps/yokoyama-ICCV-2005.pdf

http://www.sarnoff.com/downloads/research-and-development/vision-technologies/embedded-vision/moving_object.pdf

Contributions

Larry –

- Wrote the attack enemy algorithm
- Wrote the escape algorithm
- Designed the obstacle
- Wrote the re-planning code
- Worked on the write-up

Farzon –

- Rewrote the movement code from scratch to include more degrees of motion
- Rewrote the orientation code
- Wrote the Approximate cell decomposition with Alpha-beta
- Worked on the write-up

Alex –

- Improved the background subtraction algorithm
- Improved the orientation code and made it only work if both sided of the robot are detected
- Wrote visibility graph reused from last project.
- Cleaned up code from last project.
- Brought the obstacle
- Edited the write-up