Project 3

Motion Planning

Larry Freil, Farzon Lotfi, & Alex Strange

4/2/2011

Part1: Navigation Planning

1.1 Methods We Chose

The Two Methods Our Team Chose were a uniform approximate cell decomposition and The Visibility Graph.

1.1.1 Approximate cell decomposition

With approximate cell decomposition we divide the state space into a s high a resolution grid as possible where each cell in the grid contains a flag identify if an obstacle exists or not. For this to work well the resolution of the graph has to be very high in order to capture all the important details. This results in a graph with a very high number of nodes. Thus path planning is not very efficient. Given the required 800x600 camera dimensions It was not possible to compute 480,000 nodes to represent each pixel. Which left our team with two possible solutions, either scale the image or have a larger representation of nodes/unit squares. Multiple reasons such as computational speed, not wanting to possibly distort our image, and not wanting to scale the location of our reference frame contributed to our decision to just increase our unit square. We scaled the unit square to be 10x10 to cut the possible nodes down to 4,800. The next step was to generate the list of edges. We had a simple formula displayed to the side that would make sure a unit did not have any red in it (our color for denoting an obstacle) and would create edges based on the units left, right above below, and diagonal to the current unit

```
Function generateEdges:
FOR x < NewWidth
  FOR  y< NewHeight
      IF !RED: //the flag
    Current = point(x,y)
    IF y>0 :
              //top
              edgePair.add (Current, point(x,y-1))
              IF x+1 < NewWidth :
         edgePair.add (Current, point(x+1,y-1))
            IF x > 0 :
             //midLeft
             edgePair.add (Current, point(x-1,y))
            //botLeft
             IF  y+1 < NewHeight :
                edgePair.add (Current, point(x-1,y+1))
             IF (x>0) and (y>0) :
               //topLeft
                edgePair.add (Current, point(x-1,y-1))

             //midRight
              IF x+1 < NewWidth:
                  edgePair.add (Current, point(x+1,y))

             //bot
              IF  y+1 < NewHeight :
                   edgePair.add (Current, point(x,y+1))

             //botRight
              IF x+1 < NewWidth and y+1 < NewHeight
                   edgePair.add (Current, point(x+1,y+1))

RETURN edgePair;
```

square. This way when we insert this data into our graph we will only construct edges and vertices that are not within the area of the obstacles. The next step is to construct the graph we used an adjacency matrix for more efficient memory use and stored all the vertices in a hash table to make up for any time lost from not using the faster adjacency matrix. The next step is to run our a-Star search algorithm with the Manhattan heuristic. Psedo-code for the a-star search is shown

below.

```
Function aStarSearch(problem, heuristic=nullHeuristic):
  initialState = problem.getStartState()
  frontier = util.PriorityQueue()
  explored = []
  cost = 0
  actions = []
  hurristicVal = heuristic(initialState, problem)
  vertex = pathNodes(initialState,actions,cost,hurristicVal)
  frontier.push(vertex,hurristicVal+cost)
  AlphaTest = True
  while (AlphaTest):
     IF  frontier.isEmpty():
           RETURN NULL
      vertex=frontier.pop()
      currState = vertex.getState()
      actions = vertex.getActions()
      cost = vertex.getCost()
      heuristicVal = vertex.getHeuristic()
       IF problem.isGoalState(currState):
           RETURN actions //list of vertices to go to
       IF currState not in explored:
          explored.append(currState)
          prevCost = cost
           FOR children in problem.getSuccessors(currState):
              cost = prevCost + children[2]
              updateActions = actions +[children[1]]
              newHuristic = heuristic(children[0],problem)
              updatePath = pathNodes(children[0],updateActions,cost,newHuristic)
              frontier.push(updatePath,cost+newHuristic)
```

1.1.2 **Algorithm comparison**

Our team implemented uniform approximate cell decomposition so if we detected any red in a section of the grid we would make sure to avoid and find the shortest path around the grid. This is not as efficient as quad tree decomposition but it works well in enough that with inherent problems such as

drift and slippage inefficiencies in how the robot reaches its path are hardly noticeable such as avoiding grids that have a slight amount of red (the inexactness problem). While inefficient compared to a visibility graph, a uniform approximate cell decomposition can avoid obstacles that do not have corners. Visibility graphs main advantage over other methods is that they are fast. They essentially create highways from the edges of the obstacles. Since it constrains the motion to these lines planning is much faster. Comparing this to creating a connectivity graph of everything that is not an obstacle and you can quickly see why visibility graphs are faster.

1.1.3 **Visibility graphs**

Unlike cell decomposition methods, which create a node for every square of the image, the visibility graph works by storing the corners of obstacles. The data structure used is a graph formed from polygon representations of the obstacles; the vertices are all vertices of the obstacle polygons, plus another four representing the edges of the screen, and there is an edge between two vertices if they are visible from one another.

The graph is based on polygons representing obstacles, but our system detects obstacles by creating an image with the objects painted solid colors. Therefore, to create the graph, we first had to convert the image to points. Reusing the blob detection code from previous projects, we detected each red object in the image as an ellipse, and then chose four points at each axis, reducing it to a diamond shape. As the obstacles are expected to be rectangular, this should be very accurate. We then formed the graph from these points.

The result of both algorithms is a graph; as such, we used the same A* search for both, first adding the center points of the robot and goal object to the graph. The result is a short path from start to finish, where each internal vertex is the edge of an obstacle. Traveling this path presents difficulties, as the robot has a size in real life and cannot actually overlap the obstacles; this can be dealt with by shifting the path one robot radius away from each vertex, into empty space. Also, it tends to contain long straight lines, but driving Rovio on edges with arbitrary angles between them is not always possible. This can be dealt with in the driving control, but cannot be dealt with during search, due to the fewer amount of nodes to search.

The major advantage of this algorithm is that it is extremely fast. As obstacles are the only members of the graph, and are static, the graph does not need to be recalculated for a robot replan and its speed is irrelevant (although realtime). The only thing that must be rerun is the graph search, which is the same algorithm as in cell decomposition and therefore has the same time complexity. However, the graph to be searched is much simpler, faster, and its size is not affected by the camera resolution. The image below (*figure. 1*) shows an example search; with four obstacles marked, the graph contains 20 vertices and 85 edges (in purple). The white line, containing 3 edges, is the shortest path found to the goal; 36 edges were expanded to find it.

1.2.1 **The performance of the two planning methods**

The Approximate cell decomposition methods both complete because a grid is finite and optimal because it searches using A star which is optimal as long as long as our heuristic is optimal. Since we use the Manhattan distance guaranteeing optimality is not a problem. To use

an approximate Cell decomposition we must first construct a connectivity graph which takes O(V+E) construction time where V represents vertices and E represents edges. On top of that it requires a O((E + V) log V) search time for A star. Visibility graphs on the other hand have a worst case construction time of O(V^3) and a Best Case construction time of O(V^2log(V)) depending on how you construct it. Visibility Graphs are also Complete assuming Polygonal Obstacles because of the edge detection requirement and are optimal because of distance traveled as a criterion for decisions on next visited state space. Big O notation and completeness are not enough to compare these two algorithms to determine which one of these algorithms is best you must also look at efficiency. Evaluating these two algorithms on the premise of efficiency shows a clear winner in Visibility graphs as visibility graphs only require analysis on the Edges of their obstacles to become optimal while Approximate cell division requires a breakdown of the state space into very fine unit squares. The more nodes the better the evaluation but also higher demand on memory and time Which makes uniform approximate cell division a lot less practical.

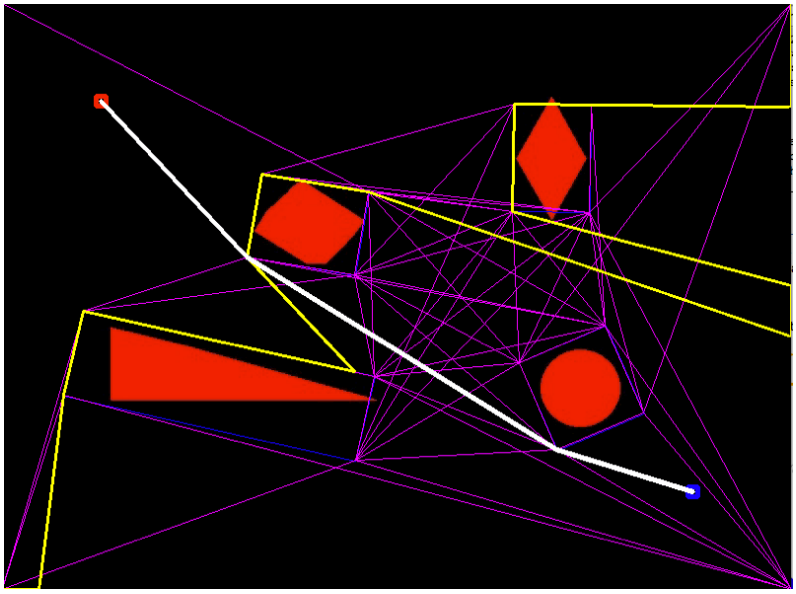

Figure 1: An example of how our visibility graph works

1.3.1 **results vs. the theory**

The Results for the most part met our expectations as described in question 1. We expected our visibility graph to run faster. At times it would approach a corner to close for comfort. We had to deal with a trade of efficiency vs meeting the requirement. This is not to say that approximate decomposition does not have undetected collisions but that using the uniform method we have a larger scale of error to work with. Recalling the implementation of our approximate cell algorithm, we stored our vertices in a hash table to allow for constant time look up. I stress this because it greatly improved the performance of our algorithm during the path searching portion.

1.4.1 **Overcoming online execution errors**

In an attempt to handle the error produced in executing the plan, there were two approaches we tried to implement. First, we planned paths that would try to keep us relatively far away from obstacles, this allows for minor errors in execution without accidental collisions. The second component is to

recalculate the robots positions regularly, and if it is not on the current path, then re-plan. This allows us to not have to wait between each step of the plan to recalculate, yet still maintain accuracy because we recalculate whenever we find ourselves off of the intended path. Because we only re-plan when we find ourselves off path, we can get to the object much more quickly. This technique gives us reliability in execution, while still responding fairly quickly.

Pseudo-code:

```
While not at target:
    Download next frame
    Detect obstacles, robot, and target fruit.
    If we already have a planned path, and are on that path:
        execute next steps of the path.
    Else:
        Generate new plan plan
        Execute the first few steps of new plan.
    Repeat.
```

This algorithm is extremely simple to implement on top of the planning algorithms. Also, it maintains accuracy, because it does not allow the robot to get to far off of the intended plan. Finally, we still avoid having to recalculate the plan every step, so it can still operate much more quickly than a single step per update algorithm.

1.5.1 **further improvements with more time**

With more time we would have attempted to implement some form of multi-threading which would allow us to execute the current plan while checking or planning the next step of the plan. This would allow us to more efficiently use our clock cycles, instead of waiting till the current part of the plan has finished executing to calculate the next part. A second improvement would be to use some of the more advanced vision detection algorithms. There are better ways to detect the position of the robot than the generic background subtraction and color selection such as using the shape and relative features of the robot as the features we are searching for. However these algorithms can be computer resource intensive as well as complicated to learn. So with more time we could have learned how to use libraries (such as leptonlib) with these algorithms, or written versions of these algorithms for our application, and with faster computers we could run them in real time. Also, as for background subtraction, It is possible to use key “features” of the background such as patterns in the carpet to detect changes caused by the camera being bumped, or it auto-adjusting the brightness levels of the whole image. These algorithms however tend to be computationally intensive, so they are hard to do in real-time without high-end computers. Also, though openCV includes functionality that makes this possible to implement relatively easily. It still requires more knowledge of openCV than we have. With more time we could experiment with some of these options more to see how well they work.

Part2: Manipulation Planning

2.1

The simplest algorithms to go to the fruit and bring it back is to use the algorithms from the beginning part of the project, with a few minor modifications. The simplest change is that the plan is executed much slower, this helps prevent the fruit from rolling away from us. This solved the fairly simple problem of the fruit moving away from us, which requires us to recalculate a path to re-secure the fruit. The hardest problem was then planning a path that would actually bring the fruit back to the starting position, because we don't have a way to pull the fruit with us we have to calculate a plan that mostly pushes the fruit to the target position.

In order to solve this problem we had to use a variation of the original algorithms where be basically required that the path always has the robot move either forwards, and can only turn up to 45 degrees at any one time. Before the robot is allowed to turn again, it must travel at least a little bit forwards. This allows the algorithms to find paths that use slow sweeping arcs instead of sharp turns, which allows us to relatively reliably keep the fruit in front of us and push it back to base. Since this is only a concern when we actually posses the fruit, we only have to use this version of the algorithm when we have the fruit in our grasp.

Algorithm:

Grab next camera frame.

Calculate Robot and target information

if(path already calculated, and fruit and robot are on path):

    take next step in path:

else:

    if(next to fruit):

        plan path where the robot travels slower, always travels forwards, and only turns small angles at a time

        execute first step in path

    else:

        plan basic path to fruit using an algorithm from the first part of the assignment

        execute first step in path.

repeat.

This general algorithm allows us to plan simpler routes to get to the fruit, while still being able to calculate the more complicated path once we have the fruit in our possession. Also, this algorithm handles the case where the fruit rolls away from us, or the robot accidentally leaves the fruit behind, because it will re-plan as soon as either the fruit or the robot are off of the plan.

2.2

The motion of the fruit is much harder to predict than the robot because we are not given direct control over its movement. Also, since it is round, it tends to roll around somewhat, and can roll away from the robot even once the robot has stopped pushing on it. In order to solve this problem, we first set the robot to move much more slowly around the fruit. This helps prevent the fruit from building up enough inertia to keep rolling even after the robot has stopped moving. Also, this can give us time to get more frequent updates from the camera, which help us track the fruit and ensure that it is still in front of the robot. If the fruit every drifted away from the robot, we switched back to our original algorithm that would head to the fruit, because if the fruit is not next to us, we don't have to worry about its motion, because it should remain stationary.

Whenever we found ourselves and the fruit off the intended plan, we chose to re-plan, because it could be possible that trying to return to the intended plan could leave the fruit behind. Also, in order to constrict the motion of the fruit, once it is trapped in front of us, we don't take rapid or large angle turns, so it helps prevent the fruit from rolling away. Also, after each turn, there must be at least one move forward step to re-gather the fruit in case it got only slightly ahead of us. With this in place, we take slower more methodic steps, that allow us to slowly herd the fruit back to base. Our paths have to take more sweeping turns, but as long as there aren't a lot of obstacles in the way, this doesn't really cause an issue. Even though it might be possible to take the course faster than we do, we figure that the less frequently we have to chase down the fruit because it got away from us will save time in the long run.

2.3

The relative speed of the planning for the first and second parts of the algorithms were fairly different. Since we always calculated the simplest path that would bring us to the fruit, it usually generated a shorter path that could include sharp turns. This allows it to calculate the absolute simplest path to the fruit, which allows for a much faster search, because we don't have to search as much of the space to find an acceptable answer. Once we had the fruit we had to use a more complicated algorithm that eliminated a lot of the simpler paths because we couldn't ensure proper control of the fruit. This required us to search a significantly larger part of the search space, which consequently results in much longer search times.

Once the plan has been made, there is still the execution time to consider. The trip to the fruit executed much more quickly that the return trip for several reasons. First, the robot was set to move more slowly when in possession of the fruit in order to prevent losing control of the fruit. Secondly the plan to get to the fruit was almost always significantly simpler than the plan to return. This simplicity leads to having to recalculate the path as frequently, as well as requiring the robot to travel a shorter distance. Finally, since the return path had to be more complicated in order to account for the lack of control of the fruit, it was more common for us to get far enough off plan that we had to recalculate more frequently. With the intentionally slower movement time, the more frequent plan recalculations, and longer time to calculate the plan, the return trip was always significantly longer that the trip to the fruit.

2.4

Of all of the components of our algorithm for manipulation planning, the execution of the plan was by far this longest. This is because to execute each successive step of the plan, we have to wait for the previous step to finish. Since this is the only part of the whole algorithm that requires making some change in the real world, it takes the longest. The actual sending of commands is very quick, but we spend large amounts of times waiting for each action to complete.

The second longest component of our algorithm was calculating the return paths, because these require more stems, and a more exhaustive search of our state space. The speed of searching for first path to the fruit was very similarly timed, because even the path there takes a significant amount of time to calculate. Also, the time here varied significantly depending on how close we were to the fruit or home base, because as we got closer to base, the calculated paths are shorter and shorter, and require less searching in order to find.

The second fastest component of our algorithm was the vision detection. The background subtraction is a linear time process, which for the size of images we are dealing with is very fast. Once we had filtered out all of the noise (background pixels), it becomes much easier to identify which category each of the important pixels belongs in. This allows for extremely rapid detection of the important objects. Also, since the obstacles are unmoving, we don't have to recalculate their position as long as we do not run into one of them. The longest part of this whole process is fetching the image over the wireless.

The fastest part of the algorithm is the check to see if we are only the path we planned out. If we don't have a plan yet (like at the start of the program), then we immediately proceed to the planning part of the algorithm. If we do have a path planned out, we merely have to check that the robot is on one of the lines of the plan. If the robot is significantly off these lines, then we start the re-planning process. If not, we continue on to the execution. In order to select which plan to use, we merely have to check whether the robot is facing the fruit and within a certain distance. If it is adjacent to the fruit, and

facing it, then we proceed with the return trip planning, otherwise we use the original trip algorithm to plan.

2.5

With more time and resources, there are a few changes we would like to make to our algorithms. The first would be to come up with a method that breaks down the world into smaller components, which would allow us to more accurately plan and track the robot and the fruit. This however takes more processing time, and in order to prevent extremely long computation times, we had to avoid increasing the resolution on our approximate cell decomposition algorithm. Secondly, we would have tried to implement a single planner that would try to calculate a path for getting to the fruit that would put us in a better position to return the fruit. For example, it would possibly take the long way around an obstacle, in order to make it so it could push the fruit the shorter and hopefully straighter path back to the starting point. However, to calculate this, we would have to calculate a much longer path with significantly more possible paths. This kind of calculation can make calculation times grow exponentially, as well as being difficult to implement.

An implementation that would probably work better, and be somewhere in between, would be to have the simplest path finding algorithm calculate the original path to the fruit, then once we are within a certain distance of the fruit, use an algorithm that calculates the path to the fruit and at least the beginning of a path back to the goal, and chooses from the best options of those. Then a third algorithm for when we possess the fruit which computes the best path back to the goal. Having the in-between algorithm could set us up in a better position while still not taking too long to calculate.

With more time we would also like to try constructing approximate Voronoi diagrams around all of the obstacles. This should make calculating paths much easier, in that it would tell us the paths that would keep us the furthest away from the obstacles until we have to head to the fruit or the goal. Each search decision would only have to follow any braches in the path until it led us to the area that the fruit is in. Once we have the fruit, the same search back would be just as simple, though we might have to favor paths that don't have sharp turns. This process would save us a lot of computing time when it came to calculating the path, because we wouldn't have to recalculate around the obstacles, the best paths would be mostly pre-generated, we just have to select the one we would want to follow. This would allow us to check our plan and replan much more frequently without having to significantly slow down our execution times

References

http://www-cs-faculty.stanford.edu/~eroberts/courses/soco/projects/1998-99/robotics/basicmotion.html

Contributions

Larry –

- Developed the background subtraction algorithm
- Improved the blob detection algorithm to better detect obstacles, the fruit , and the robot
- Worked on the write-up

Farzon –

- Improved our movement algorithm
- Wrote the Approximate cell decomposition
- Worked on the write-up

Alex –

- Wrote the visibility graph
- Worked on the write-up