Continuous Space Pathfinding

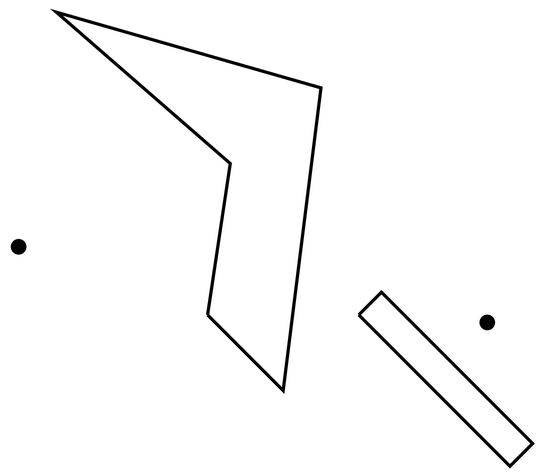
Daniel Wisdom

28 April 2017

1 Introduction

All the traditional pathfinding algorithms, such as BFS, A*, or Dijkstra work with a graph of nodes, often on a grid. None of these algorithms really works when we need to pathfind in continuous space. For this lecture, continuous space will mean that obstacles, the traveler, and the start and end have floating-point coordinates. The traveler can travel at any angle, or even follow a curve if he wants. Usually it is easiest to decompose the given obstacles into a graph, and the use standard shortest path algorithms on that graph.

2 Pathfinding with Polygons



The key observation for this problem is that the shortest path between the start and end must consist only of straight lines between vertices, the start, and the end. We can prove this because any curved path could be shortened by going straight between its endpoints. The curve can only be shortened until it hits a vertex of one of the polygons. Intuitively, the traveler should cut corners as close as possible to get the shortest path, therefore he will go straight from one corner to another. This suggests constructing a graph of all the straight lines between vertices, start, and end. To do this, we loop through every pair i,j of vertices and check if there is a clear path between them.

Checking for a clear path is surprisingly hard. We must first check whether the line segment from i to j intersects any edge of any polygon. Note that we must exclude the edges adjacent to i and j because they share endpoints. If the line segment collides with any edge it is not a valid path. One special case we need to consider is an i, j in the same polygon. For example, the diagonals of a square are not a valid path, but do not collide with any edges. To account for this, we must test whether some point on the line segment, the midpoint is easiest, lies inside the polygon.

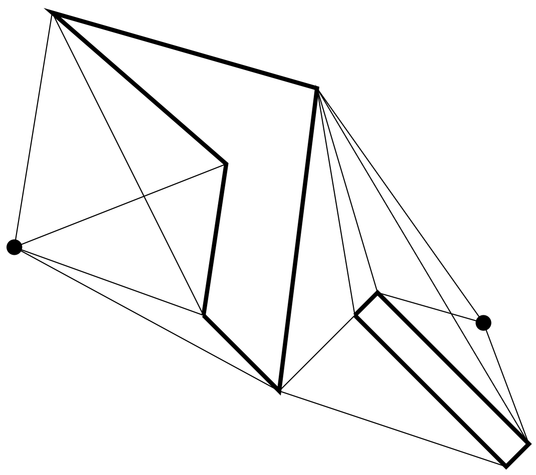
The easiest way to do this is to take a ray out from the midpoint and count the number of intersections with the polygon. With each intersection with an edge of the polygon the ray alternates from inside to outside the polygon. Therefore, if the number of intersections is odd, the original point was inside the polygon. If the number is even, the point is outside. When making this ray, it should not directly hit any vertices because that causes special cases I don't know how to handle. Instead just repick the ray if it hits a vertex, or assume a random ray probably won't perfectly hit a vertex.

Algorithm 1 Polygon Graph

```
function CONSTRUCTGRAPH(vertices, polygons)
    for all vertices, start point, and destination $i$, $j$ do
        if $i$ and $j$ from same polygon then
            if insidePolygon(($i$+$j$)/2, the polygon) then
                next $i$,$j$
        for all vertices $v1$,$v2$ which are adjacent do
            if neither $v1$ nor $v2$ is $i$ or $j$ and intersects($i$,$j$,$v1$,$v2$) then
                next $i$,$j$
        add $i$,$j$ to the graph

function INTERSECTS(points s1, e1, s2, e2)
    orient1=orientation(s1,e1,s2)
    orient2=orientation(s1,e1,e2)
    orient3=orientation(s2,e2,s1)
    orient4=orientation(s2,e2,e1)
    if orient1=0 and orient2=0 and orient3=0 and orient4=0 then
        return whether the x and y ranges overlap
    else
        return orient1 $\neq$ orient2 and orient3 $\neq$ orient4

function ORIENTATION(p1,p2,p3)
    val=(p2.y-p1.y)*(p3.x-p2.x)-(p3.y-p2.y)*(p2.x-p1.x)
    if val<0 then return -1
    else if val>0 then return 1
    else return 0

function INPOLYGON(point m, polygon poly)
    endpoint=(99999+random(),99999+random())
    inside=false
    for all edge between $i$, $j$ in poly do
        if orientation(m,endpoint,$i$)=0 or orientation(m,endpoint,$j$)=0 then return inPolygon(m,poly)
        if intersects($i$,$j$,m,endpoint) then
            inside=not inside
    return inside
```

Note that (99999,99999) is completely arbitrary, so long as it falls outside the polygon. The check is orientation equals zero is in case we directly hit a vertex, in which case we rerun the method and assume our randomness will miss all vertices most of the time.

After running the above algorithm, we get a graph with all the vertices connected if they have a line of sight. This graph includes all the edges of the polygons. There are V^2 pairs *i*,*j* and it takes V intersection tests per pair, so our overall run time is $O(V^3)$. For the beginning example, it should look like:



3 Pathfinding with a Graph

Now we can run Dijkstra's or any other pathfinding algorithm we like on this graph. A* works well because distance is a good heuristic. One advantage of this type of pathfinding algorithm is that its runtime is dependent only on the number of obstacles, not the size of the grid. This means for a million by million grid with five obstacles, it is probably faster to used this algorithm than a grid-based one. That would require using Manhattan distance instead of the usual Euclidean distance.

4 Bug Pathfinding

Bug pathfinding is an algorithm that needs only local knowledge and takes little computation power. It will rarely take the **best** path, but if a path exists it will eventually get to the destination. Bug pathing probably won't show up on algorithmic competitions, but it is huge in real world robot pathfinding and in MIT Battlecode, a strategic programming competition. For bug we think of a robot, karel, traveling through some map. All it can see is what's next to it.

A naive form of the bug algorithm is a wall follower. Karel can always follow the wall to his left, and this will lead him out of mazes, as long as there are no loops in the maze. We'll use this idea as the basis of our bug algorithm. For our algorithm, karel will have a destination point, *dest*. Karel always knows his own location, *current*. Karel can then easily find the direction and distance to *dest*.

Karel can have two states, traveling and wall-following. Karel will also remember the closest he has ever been to his goal and the obstacle he is currently following.

Algorithm 2 Full Bug Pathfinding

```
minDist=$\infty$
while not at destination do
    curDist=distance($current$,$dest$)
    if onWall then
        if curDist $\leq$ minDist and can move toward $dest$ then
            onWall=false
        else
            move left along obstacle
    else
        if can move toward $dest$ then
            move toward $dest$
        else
            curObstacle=what is obstructing karel's move
            onWall=true
    minDist=min(curDist,minDist)
```

This is a very high level form of a bug algorithm. Some complications which make bug harder and much messier are changable terrain, other robots also using bug pathing in the way, or map edges. These are all achievable, but my bug algorithm ended up being about 300 lines long.

5 Challenges

Do Battlecode 2018. It starts early January and has $50,000+ in prizes. In my opinion the best coding competition I've ever done.
How could you modify the polygon pathfinding algorithm to instead pathfind around circles?
Go back to karel and write yourself a bug. Try to make it work with many robots going in diferent directions.