An Algorithm for Edge Generation in Graphs of Real World Vertex Fields for Travel with Momentum Constraints

Ethan Reyes

Table of Contents

Abstract

The use of Dijkstra's algorithm in finding the shortest distance between two points that can be represented by a graph is commonplace. Existing variants of the algorithm, while in some cases accounting for other constraints, fail to take into account constraints on movement in physical space by a vehicle's momentum and direction of travel at any given vertex, or if they do, are limited by angle limits that involve rigid movement. This is particularly needed in a field of vertices that represent a largely unrestricted travel area such as a body of water or airspace. Existing algorithms also do not perform realtime edge calculations based on previous travel conditions that affect edge distances. This research addresses these factors to create realistic shortest-path results by extending Dijkstra's algorithm. This algorithm has applications to both sea vessel and aircraft travel. The functionality of this algorithm is demonstrated using real world satellite imagery superimposed with calculated paths that represent likely applications.

Introduction

A common scenario in graph theory is needing to know the shortest distance between two vertices. One of the most common algorithms for accomplishing this goal is known as Dijkstra's algorithm, a concept conceived by Edsger Dijkstra in 1956. (Figure 1) The algorithm begins by defining arrays holding values for the lowest distance to and the previous vertex for reaching each vertex. Initially all of these values are set to infinity for the lowest distance and undefined for the previous vertex, as neither of these have been determined. All vertices are then added to a queue for processing. The algorithm then loops through the vertices in the queue, each time processing the vertex with the lowest distance, starting with the first point at distance zero. The algorithm then calculates distance values for all the vertices with which the currently in process vertex, can reach via an edge. If the new distance is lower than the previously defined distance to reach a vertex, then the distance to that vertex is updated in the array and the previous vertex is updated to the vertex in process (Sedgewick). This is a computationally efficient way to process the shortest path in a graph and has a worst case performance of $O(|V|^2)$ where V is the number of vertices (Schrijver).

```
 1  function Dijkstra(Graph, source):
 2
 3      for each vertex v in Graph.Vertices:
 4          dist[v] ← INFINITY
 5          prev[v] ← UNDEFINED
 6          add v to Q
 7      dist[source] ← 0
 8
 9      while Q is not empty:
10          u ← vertex in Q with min dist[u]
11          remove u from Q
12
13          for each neighbor v of u still in Q:
14              alt ← dist[u] + Graph.Edges(u, v)
15              if alt < dist[v]:
16                  dist[v] ← alt
17                  prev[v] ← u
18
19      return dist[], prev[]
```

Figure 1: Dijkstra's Algorithm Pseudocode (Abba)

A commonly used application that utilizes Dijkstra's algorithm is Google Maps (Mitchell). Google Maps allows a user to enter an origin and destination to travel to in a car with a goal of providing the user with that fastest way to reach their destination. The application uses real time traffic information and speed limits between intersections to assign edge distance values in a graph and then uses Dijkstra's algorithm to find the fastest route. Dijkstra's algorithm makes a number of assumptions about traversal of edges. When an edge is traversed, the algorithm assumes that no conditions exist that would prevent the validity of subsequent edges. In the case of Google Maps, this works well for its goal of finding routes for cars, as

roads are designed to be the proper width to accommodate automobiles and every car has the ability to stop at an intersection and change direction. However, different modes of transportation require different considerations. All edges in a graph being processed by Dijkstra's algorithm are assumed to be similar to roads being traversed by cars and all vertices to be intersections with stop signs. This makes the algorithm unusable in real world scenarios that involve using the algorithm to calculate shortest paths for vehicles with momentum that do not have the ability to stop and make abrupt turns. For example, a boat or an airplane does not have the ability to stop abruptly at a point in space and instantaneously make a 90 degree turn. Traveling between an airplane's current point and another point may require a 90 degree turn instantaneously for the shortest distance but that does not fit into the constraints of an airplane's abilities. Boats and airplanes also have other restrictions that aren't addressed by the algorithm. Airplanes require the ability to be routed around potential weather hazards and restricted airspace and boats often encounter situations with dimension differences where a waterway may not be wide enough to accommodate a vehicle. The considerations for these modes of transportation involve accounting for momentum and clearance of the vehicles. To address these issues, an extension is required for real world vehicles that utilize graph based navigation to determine the shortest path to their destination. This project's goal is to develop such an extension.

To accomplish this task, very little is needed to be changed in Dijkstra's algorithm itself with the majority of the extension algorithm applying to the real-time calculation of edges. Code segments in this paper are expressed in Python 3.9 code as some of the critical aspects of optimization can not be expressed in pseudo code.

Initial Considerations

To begin the task, the vertices set, $\mathbb{V}$, of the graph needs to be defined. As the path needs to take place in two dimensional physical space, the vertices will be defined by x and y values. By keeping the algorithm unit-agnostic, the x and y values can be restricted to integers. The user of the algorithm can then pick a unit that suits their case. A convenient real world mapping to GPS coordinates using the Universal Transverse Mercator (UTM) system can be achieved using either meters or kilometers as a unit. The UTM system utilizes easting and northing numbers at one meter intervals broken into one kilometer grids which function the same way as x and y values in a Cartesian coordinate system (Moore). A critical factor in determining the unit to use is choosing a unit that is not too large that a vehicle would likely change course between the two points. A unit that is too large could potentially not include an obstruction between two vertices as it would not be sampled accurately. This would make edges considered valid that should not be. Choosing a unit that is excessively small would result in a lot of extra processing time but

will still result in an accurate path. A convenient way to create a list of vertices is to take a satellite image of an area to be traversed, write a simple program to sample it at different pixels, and analyze the color at the sampled pixels to determine if they are valid vertices. This method will work for cases like traversing waterways or avoiding weather conditions from Doppler radar maps as there are color differences that indicate whether or not a point is valid. While these vertices give us a position in two dimensional space, with this extension there needs to be added another dimension to account for the direction of travel at that point. Due to this, each vertex will be defined by ($x \in \mathbb{Z}, y \in \mathbb{Z}, t \in [0, 2\pi)$).

Edge Calculation

While the shortest distance between any two lattice points is a straight line, vehicles with momentum can't instantaneously halt and change directions to follow that straight line. To calculate the distance and other relevant data traveling between two points, it must be considered **how** the vessel will travel between the points. When taking a turn in physical space, the path traveled is an arc. In the initial considerations, it was outlined how to choose a unit between vertices so as to not be large enough that a course correction between points is required. If a course correction won't take place, then a circular arc exists; one with a fixed radius. At the beginning of edge traversal, the direction of travel and initial location will be known. Since the vehicles will be traveling along a circular arc, the direction of travel will be the angle of the tangent to that arc at the starting point, which is where the calculations start:

The following variables with input constraints (variables denoted as $undefined$ are being solved for)(Figure 2):

- $(x_0 \in \mathbb{Z}, y_0 \in \mathbb{Z})$ is the coordinate of the origin point
- $(x_1 \in \mathbb{Z}, y_1 \in \mathbb{Z})$ is the coordinate of the destination point
- (x_c, y_c) is the coordinate of the center point of the circular arc connecting point (x_0, y_0) and point (x_1, y_1) $(undefined)$
- $t_0 \in \mathbb{p}$ is the angle of the tangent line to the circular arc at point (x_0, y_0). For this application, this is the angle of departure from the origin point. Set $\mathbb{p}$ is defined in the **Precision** section of this paper.
- t_1 is the angle of the tangent line to the circular arc at point (x_1, y_1). For this application, this is the angle of arrival at the destination point $(undefined)$
- θ_0 is the angle of point (x_0, y_0) with respect to point (x_c, y_c) $(undefined)$
- θ_1 is the angle of point (x_1, y_1) with respect to point (x_c, y_c) $(undefined)$
- θ is the angle of the arc between (x_0, y_0) and (x_1, y_1) $(undefined)$

- r is the radius of the circular arc with center at (x_c, y_c) beginning at point (x_0, y_0) and ending at point (x_1, y_1) $(undefined)$
- d is the length of the circular arc, For this application this is the distance of the edge $(undefined)$

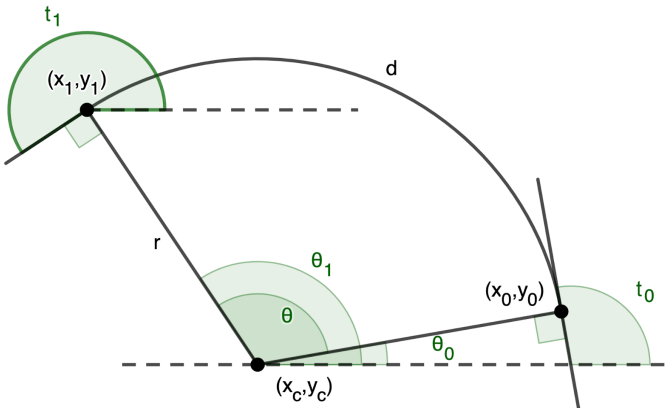

Figure 2: Visualization of Variables (Reyes)

The angles t_0 and t_1 are such that $t = 0$ is in the direction of the x-axis with ascending x values, increasing in a counter-clockwise direction in radians. The ultimate goal, for the purposes of edge calculation, is to find the tangent angle at the destination point (t_1) and the length of the circular arc between the origin point (x_0, y_0) and destination point (x_1, y_1), which is *d*. The center point of the circular arc (x_c, y_c) must be found to do this. This point must be on a line that passes through point (x_0, y_0) that is perpendicular to t_0. Using the common trigonometry equations $x = r{\cdot}cos(\theta)$ and $y = r{\cdot}sin(\theta)$ (Nykamp), equations can be created to represent this line as a function of r. The starting angle is the tangent to the arc, but the line that needs to be drawn is perpendicular to that line, so the $sin()$ and $cos()$ functions to $cos()$ and $sin()$ need to be changed, respectively. The starting point is not necessarily $(0, 0)$, so the equation needs to be changed to account for the x and y offset which results in the following functions:

$$x_{toc}(r) = x_0 - (r{\cdot}sin(t_0))$$
$$y_{toc}(r) = y_0 + (r{\cdot}cos(t_0))$$

It's important to note that it is possible to have a negative r value depending on where point $(x_{toc}(r), y_{toc}(r))$ exists in relation to point (x_0, y_0). While the sign distinction of r is needed for further calculation, the radius of the circular arc, or the distance between (x_0, y_0) and $(x_{toc}(r), y_{toc}(r))$, is actually $|r|$. Since point (x_c, y_c) will be the only point on the line represented by the previously defined functions and is also equidistant to point (x_0, y_0) and point (x_1, y_1), the formula

for the distance between two points can be used for each $(x_{toc}(r), y_{toc}(r)) \rightarrow (x_o, y_0)$ and $(x_{toc}(r), y_{toc}(r)) \rightarrow (x_1, y_1)$ and setting those distances equal to each other:

$$\sqrt{(x_{toc}(r) - x_0)^2 + (y_{toc}(r) - y_0)^2} = \sqrt{(x_{toc}(r) - x_1)^2 + (y_{toc}(r) - y_1)^2}$$

In this particular case, both sides can be squared without affecting the result of the equation and we can now solve for the radius, r, which results in the following:

$$r = \frac{(x_1 - x_0)^2 + (y_1 - y_0)^2}{2(y_1 - y_0)cos(t_0) - 2(x_1 - x_0)sin(t_0)}$$

Now that the distance between (x_0, y_0) and the center of the circular arc can be calculated, the coordinate values of (x_c, y_c) can be determined by solving directly for (x_c, y_c) using the original functions:

$$x_c = x_0 + sin(t_0)\left(\frac{(x_1 - x_0)^2 + (y_1 - y_0)^2}{2(x_1 - x_0)sin(t_0) - 2(y_1 - y_0)cos(t_0)}\right)$$

$$y_c = y_0 - cos(t_0)\left(\frac{(x_1 - x_0)^2 + (y_1 - y_0)^2}{2(x_1 - x_0)sin(t_0) - 2(y_1 - y_0)cos(t_0)}\right)$$

Since the coordinate (x_c, y_c) is defined and both the given beginning and end points,(x_0, y_0) and (x_1, y_1), are known, the two argument arctangent function can be used to calculate the angles of the radii from the center point (x_c, y_c) to origin point (x_0, y_0), defined as θ_O, and to destination point (x_1, y_1), defined as θ_1 (Ellis) :

$$\theta_0 = atan2(y_0 - y_c, x_0 - x_c)$$
$$\theta_1 = atan2(y_1 - y_c, x_1 - x_c)$$

The sign of r will have an effect on both direction the arc is drawn and the angle of the tangent at destination (x_1, y_1) . The two argument arctangent function also returns a value in set [-π, π] and the range in use is [0,2π) (Haber). To account for these an angle for the arc and the tangent equations need to be defined based on the following conditionals:

$$\theta_{arc} = \begin{cases} \theta_1 - \theta_0 + 2\pi, \; if \; (\theta_1 - \theta_0 < 0) \; and \; r > 0 \\ \theta_1 - \theta_0 - 2\pi, \; if \; (\theta_1 - \theta_0 > 0) \; and \; r < 0 \\ \theta_1 - \theta_0, \; otherwise. \end{cases}$$

$$t_1 = \begin{cases} \theta_1 - \pi/2, \; if \; r < 0 \\ \theta_1 + \pi/2, \; otherwise. \end{cases}$$

If point (x_1 , y_1) lies at an angle of t_0 or $t_0 + \pi$ from point (x_0 , y_0), then the radius will be infinitely large, as a straight line exists between the two points at the angle of departure . This will result in an illegal operation as the denominator of the r equation will be zero. In the occurrence of the angle being t_0, the distance can be calculated using the distance between two points formula. If the angle is $t_0 + \pi$, the destination point is behind the origin point with its exact direction of travel, and thus can be disregarded as it's an invalid neighboring node. Since the slope of angles t_0 and $t_0 + \pi$ are the same, the difference can be determined by using the atan2 function and comparing the result with tangent angle t_0 (the comparison should be approximated to account for precision limitations of floating point numbers). In all other cases, the the edge length is simply the absolute value of the radius multiplied by the angle of the arc (Burns):

$$d = \begin{cases} \theta_{arc} \cdot |r|, \; if \; 2(y_1 - y_0)cos(t_0) - 2(x_1 - x_0)sin(t_0) \approx 0 \\ \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}, \; else \; if \; atan2(y_1 - y_0, x_1 - x_0) = t_0 \\ undefined, \; otherwise. \end{cases}$$

Figure 3 shows these calculations in Python. This code utilizes the Precision class defined in the next section, but has had code from other sections of this paper removed to reflect only code pertaining to edge calculation (specifically the code outlined in the Requisite Points section).

```
def edgeCalculation(__x0, __y0, __x1, __y1, __t0, __b, __minRadius, P):

   if math.isclose((2 * (__y1 - __y0) * math.cos(__t0)) - (2 * (__x1 - __x0) *
math.sin(__t0)), 0, abs_tol=1e-10):
       if P.effectiveTangent(math.atan2(__y1 - __y0, __x1 - __x0)) ==
P.effectiveTangent(__t0):
           __d = math.sqrt((__x1 - __x0) ** 2 + (__y1 - __y0) ** 2)
```

```
            return {'t1': P.effectiveTangent(__t0), 'd': __d, 'requisiteVertices':
__requisiteVertices, 'coords': __coords}
        return False

    __r = (((__x1 - __x0) ** 2) + ((__y1 - __y0) ** 2)) / ((2 * (__y1 - __y0) *
math.cos(__t0)) - (2 * (__x1 - __x0) * math.sin(__t0)))
    if abs(__r) < __minRadius:
        return False
    __xc = __x0 + (math.sin(__t0) * (((__x1 - __x0) ** 2) + ((__y1 - __y0) ** 2)) / ((2 *
(__x1 - __x0) * math.sin(__t0)) - (2 * (__y1 - __y0) * math.cos(__t0))))
    __yc = __y0 - (math.cos(__t0) * (((__x1 - __x0) ** 2) + ((__y1 - __y0) ** 2)) / ((2 *
(__x1 - __x0) * math.sin(__t0)) - (2 * (__y1 - __y0) * math.cos(__t0))))
    __theta0 = math.atan2(__y0 - __yc, __x0 - __xc)
    __theta1 = math.atan2(__y1 - __yc, __x1 - __xc)

    if __theta1 - __theta0 < 0 < __r:
        __theta = __theta1 - __theta0 + 2*math.pi
    elif __theta1 - __theta0 > 0 > __r:
        __theta = __theta1 - __theta0 - 2*math.pi
    else:
        __theta = __theta1 - __theta0
    if __r < 0:
        __t1 = __theta1 - math.pi/2
    else:
        __t1 = __theta1 + math.pi/2

    __d = abs(__theta) * abs(__r)

    return {'t1': P.effectiveTangent(__t1), 'd': __d, 'radius': __r}
```

Figure 3: Edge Calculation Python Code (Reyes)

Precision

With the added dimension of arrival angles at each vertex in the graph, the number of possible vertices is potentially much larger than is practical. Each time a t_1 value is calculated for a vertex, that number may be overly precise and in many cases effectively identical to a t_1 value created when calculating an edge from a different vertex. There would be no point in processing vertices (x , y, t) and (x , y, $t \pm \pi/180$) if the vehicle the algorithm is calculating for doesn't have the ability to steer with 1° accuracy. To consolidate these vertices, a precision value, p, will be defined that will divide the possible tangent values in [0, 2π) into ranges that will be considered equivalent based on the use case. To ensure the divisions are the same in all four quadrants, the constraint of $p \bmod 4 = 0$ is needed to create a set with valid tangent values:

$$\mathbb{p} = \{ (2\pi/p)\cdot 0 , (2\pi/p)\cdot 1, (2\pi/p)\cdot 2 \ldots (2\pi/p)\cdot(p-1) \}$$

The sympy package for Python allows the functionality of using a π symbol (among others) to display the tangent in a usable form instead of a decimal number (Meurer et al.). Sympy will be used for display purposes and the decimal version for comparison purposes when determining an effective tangent, the tangent that the t values will be rounded to. To remove the need to generate $\mathbb{p}$ every time an edge calculation is performed, a precision class will be created that is passed to the functions with the set already generated. The Precision class code in (Figure 4) populates $\mathbb{p}$ on initialization as the dictionary

self.validTangents, with the keys being the decimal values and the values being the simplified, displayable values, of each value in $\mathbb{p}$. The effectiveTangent() function returns the rounded tangent value for a supplied tangent and the tangentList() function returns a list of the simplified values in set $\mathbb{p}$, which will be used later in edge discovery.

```
import math
from sympy import *

class Precision:

   def __init__(self, __precision):
       if __precision % 4 != 0:
           return False
       self.validTangents = {}
       self.validTangentKeys = []

       for __precisionCycle in range(0, __precision):
           self.validTangents[2 * math.pi / __precision * __precisionCycle] = simplify(2 *
pi / __precision * __precisionCycle)
           self.validTangentKeys = list(self.validTangents.keys())

   def effectiveTangent(self, __angTan):
       if __angTan < 0:
           __angTan = 2 * math.pi + __angTan
       return
self.validTangents[self.validTangentKeys[min(range(len(self.validTangentKeys)), key=lambda
i: abs(self.validTangentKeys[i] - __angTan))]]

   def tangentList(self):
       return list(self.validTangents.values())
```

Figure 4: Precision Python Code (Reyes)

Requisite Vertices

Another consideration is required to determine whether or not the destination node can be reached from the source node by a circular arc. Since any vehicle traveling the edge path will have a width, and presumably have some additional clearance requirement, it needs to be ensured that this width and clearance exist in the graph. $b \in \mathbb{R}^+$ will be defined to represent the beam, a term commonly used to represent the width of watercraft (Ghosh). The number supplied should include any clearance width in addition to the width of the vehicle to ensure proper clearance. To ensure that the object traversing the edge can reach (x_1 , y_1) from (x_0 , y_0), all lattice points (coordinates where $x \in \mathbb{Z}$, $y \in \mathbb{Z}$) that exist within distance $b/2$ and are perpendicular to the tangents of the arc connecting the vertices, must be present in the graph (Insall). In the event the path is a straight line, then the vertices will be within distance b/2 and perpendicular to the path itself. The method for determining these vertices will be different depending on whether the path is an arc or a line.

For an arc, the area of inclusion for requisite vertices will be a segment of an annulus, or ring shape, with the borders being the arcs with radii $|r| \pm b/2$. For efficiency the extremities of this annulus segment will

need to be determined so the algorithm can iterate through x and y values without testing more coordinates than necessary.

This can be achieved by creating a set for candidate x-values and y-values, $\mathbb{M}^x$ and $\mathbb{M}^y$, respectively, and adding the beginning and ending points of the borders for the inclusion area for requisite vertices. These values can be determined as it is known that they will exist at distances $|r| \pm b/2$ at angles θ_0 and θ_1 from point (x_c , y_c):

$$\mathbb{M}^x = \{ x_c + (|r| \pm (b/2)) \cdot cos(\theta_0), x_c + (|r| \pm (b/2)) \cdot cos(\theta_1) \}$$
$$\mathbb{M}^y = \{ y_c + (|r| \pm (b/2)) \cdot sin(\theta_0), y_c + (|r| \pm (b/2)) \cdot sin(\theta_1) \}$$

In many cases the extremities will be limited to these values, but in some cases the extremities will be more extreme. Since this is a circular arc and its center and radius are known they can be determined via a simple method. If the angles 0, π/2, π, or 3π/2 exist between θ_0 and θ_1, then an extremity will exist at $|r| + b/2$ in the direction of that angle from the center point of the arc. The extremities can now be defined as:

$$x_{min} = \begin{cases} x_c - (|r| + (b/2)), \ if\ \pi\ is\ between\ \theta_0\ and\ \theta_1 \\ min(\mathbb{M}^x), \ otherwise. \end{cases} \qquad x_{max} = \begin{cases} x_c + (|r| + (b/2)), \ if\ 0\ is\ between\ \theta_0\ and\ \theta_1 \\ max(\mathbb{M}^x), \ otherwise. \end{cases}$$

$$y_{min} = \begin{cases} y_c - (|r| + (b/2)), \ if\ 3\pi/2\ is\ between\ \theta_0\ and\ \theta_1 \\ min(\mathbb{M}^y), \ otherwise. \end{cases} \qquad y_{max} = \begin{cases} y_c + (|r| + (b/2)), \ if\ \pi/2\ is\ between\ \theta_0\ and\ \theta_1 \\ max(\mathbb{M}^y), \ otherwise. \end{cases}$$

Note that "between" in the algorithm is defined as the angle value between θ_0 and θ_1, which may fail a less than or greater than comparison depending on the direction of the arc or if the arc crosses the line from (x_0 , y_0) at angle θ_0 (Figure 5).

```
def isBetweenAngles(__a0, __a1, __acheck, __r):

   # Convert scale from 0-2pi to -(pi)-pi if necessary to account for the output from atan2
function
   if __acheck > math.pi:
       __acheck = __acheck - (2*math.pi)
```

```
    if __r > 0 and __a1 >= __a0 and __a0 <= __acheck <= __a1:
        return True
    elif __r > 0 and __a1 <= __a0 and (__acheck >= __a0 or __acheck <= __a1):
        return True
    elif __r < 0 and __a1 <= __a0 and __a0 > __acheck > __a1:
        return True
    elif __r < 0 and __a1 >= __a0 and (__acheck >= __a1 or __acheck <= __a0):
        return True

    return False
```

Figure 5: "Betweeness" Python Code (Reyes)

The extremities defined will form the evaluation block. Only lattice point coordinates that exist within the extremities of a rectangular box formed with opposite corners at ($\lceil x_{min} \rceil, \lceil y_{min} \rceil$) and ($\lfloor x_{max} \rfloor, \lfloor y_{max} \rfloor$) will be evaluated as those are the only vertices that could possibly be requisites. The algorithm will now iterate through all of the coordinates in the evaluation block and test each point to see if it follows two rules to verify it's in the area of inclusion: first-that it's distance from (x_c , y_c) is between $|r| \pm (b/2)$ and second-that it's angle from (x_c , y_c) is "between" θ_0 and θ_1 (Figure 6).

```
__Mx = [__xc + ((abs(__r)-(__b / 2)) * math.cos(__theta0)), __xc + ((abs(__r)+(__b / 2)) *
math.cos(__theta0)), __xc + ((abs(__r)-(__b / 2)) * math.cos(__theta1)), __xc +
((abs(__r)+(__b / 2)) * math.cos(__theta1))]
__My = [__yc + ((abs(__r)-(__b / 2)) * math.sin(__theta0)), __yc + ((abs(__r)+(__b / 2)) *
math.sin(__theta0)), __yc + ((abs(__r)-(__b / 2)) * math.sin(__theta1)), __yc +
((abs(__r)+(__b / 2)) * math.sin(__theta1))]
__xMin = min(__Mx)
__xMax = max(__Mx)
__yMin = min(__My)
__yMax = max(__My)
if isBetweenAngles(__theta0, __theta1, 0, __r):
    __xMax = __xc + (abs(__r) + (__b/2))
if isBetweenAngles(__theta0, __theta1, math.pi / 2, __r):
    __yMax = __yc + (abs(__r) + (__b/2))
if isBetweenAngles(__theta0, __theta1, math.pi, __r):
    __xMin = __xc - (abs(__r) + (__b/2))
if isBetweenAngles(__theta0, __theta1, 3 * math.pi / 2, __r):
    __yMin = __yc - (abs(__r) + (__b/2))

# Compensate for limitations of Python precision
if abs(round(__xMin)-__xMin) < 1e-10:
    __xMin = round(__xMin)

if abs(round(__xMax)-__xMax) < 1e-10:
    __xMax = round(__xMax)

if abs(round(__yMin)-__yMin) < 1e-10:
    __yMin = round(__yMin)

if abs(round(__yMax)-__yMax) < 1e-10:
    __yMax = round(__yMax)

__requisiteVertices = [(__x1, __y1)]

for __x in range(math.ceil(__xMin), math.floor(__xMax)+1):
    for __y in range(math.ceil(__yMin), math.floor(__yMax)+1):
        if not abs(__r) - (__b / 2) < abs(math.sqrt((__xc - __x) ** 2 + (__yc - __y) ** 2)) <
abs(__r) + (__b / 2):
            continue
        if not isBetweenAngles(__theta0, __theta1, math.atan2(__y - __yc, __x - __xc), __r):
```

```
            continue
        __requisiteVertices.append((__x, __y))
```

Figure 6: Arc Path Requisite Point Python Code (Reyes)

When visualizing an evaluation block for an arc path it is noticeable that segments of the arc are clipped due to it not being possible for a lattice point to be in those segments (Figure 7). Less clipping will occur if a smaller unit is used.

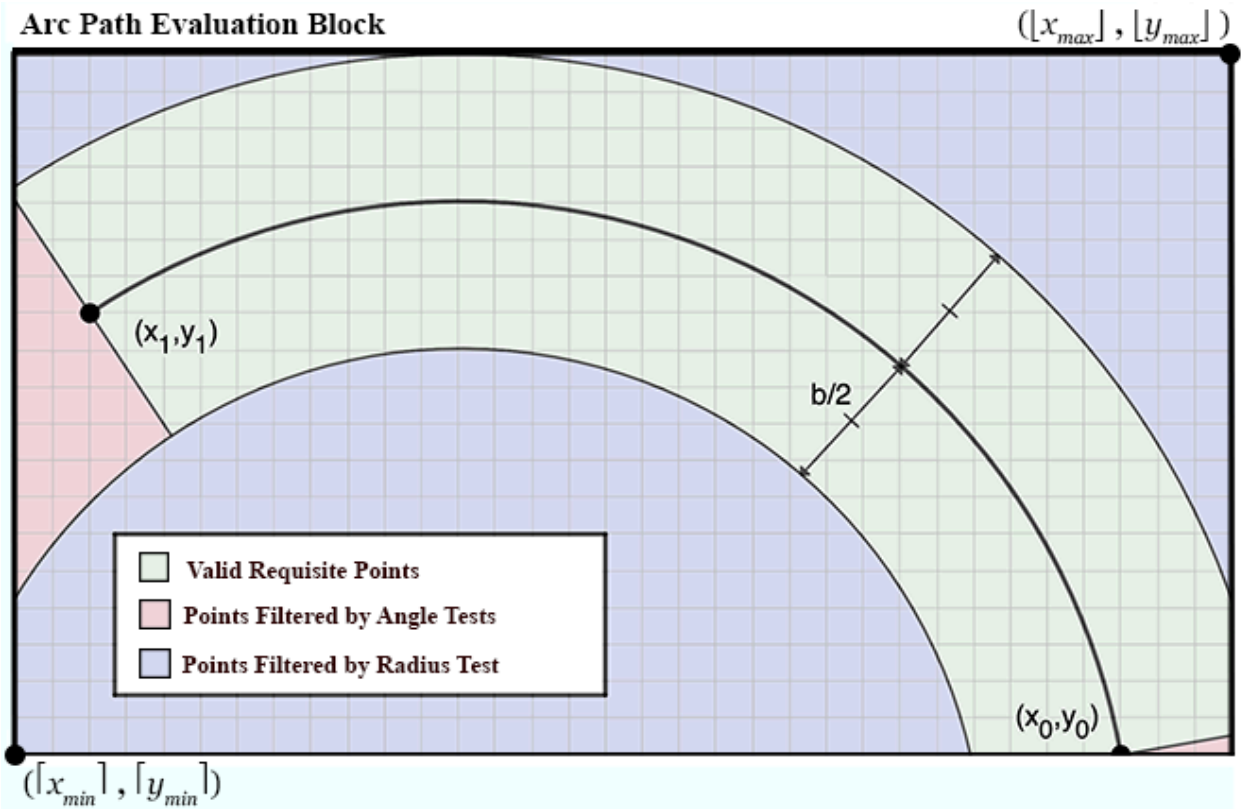

Figure 7: Visualization of Arc Path Evaluation Block (Reyes)

For a path that is a straight line, this process is simpler. Instead of an annulus, the shape formed by the path of a line with a clearance on each side will always form a rectangle (Figure 9). The coordinates for the points that define the rectangle will be at distance b/2 from vertices (x_0, y_0) and (x_1, y_1) in both directions, perpendicular to the line connecting the vertices. Instead of solving for the coordinates directly, sets can be made of all the x and y values for these points so they can be used to determine the extremities of the evaluation block. The same principle will be used that was used to form functions $x_{toc}(r)$ and $y_{toc}(r)$, as vertices at a certain distance and angle from another vertex are being solved for. So the following sets where θ is the angle of the line connecting (x_0, y_0) and (x_1, y_1) are defined:

$$\mathbb{M}^x = \{ x_0 + b/2 \cdot cos(\theta),\ x_0 + b/2 \cdot cos(\theta + \pi),\ x_1 + b/2 \cdot cos(\theta),\ x_1 + b/2 \cdot cos(\theta + \pi) \}$$

$$\mathbb{M}^y = \{ y_0 + b/2 \cdot sin(\theta),\ y_0 + b/2 \cdot sin(\theta + \pi),\ y_1 + b/2 \cdot sin(\theta),\ y_1 + b/2 \cdot sin(\theta + \pi) \}$$

And alternately define the following extremities:

$$x_{min} = min(\mathbb{M}^x) \qquad x_{max} = max(\mathbb{M}^x)$$
$$y_{min} = min(\mathbb{M}^y) \qquad y_{max} = max(\mathbb{M}^y)$$

Similarly to finding requisite vertices for an arc path, the opposite corners of the evaluation block will be the rounded values of the extremities, $(\lceil x_{min} \rceil, \lceil y_{min} \rceil)$ and $(\lfloor x_{max} \rfloor, \lfloor x_{max} \rfloor)$, and the algorithm will iterate through all of the lattice points in that block to test for inclusion. The test for whether a point fits in the area of inclusion will be a polygon inclusion test which will be done using the shapely library in Python (Gillies et al.). To determine whether or not the point is a requisite, it has to be tested with "within" or "intersects", as a "within" test will fail if a point lies on the border of the rectangle.

```
__requisiteVertices= [(__x1, __y1)]
__angle = math.atan2(__y1 - __y0, __x1 - __x0) + (math.pi / 2)
__Mx = [__x0+((__b/2)*math.cos(__angle)), __x0+((__b/2)*math.cos(__angle+math.pi)),
          __x1+((__b/2)*math.cos(__angle)), __x1+((__b/2)*math.cos(__angle+math.pi))]
__My = [__y0+((__b/2)*math.sin(__angle)), __y0+((__b/2)*math.sin(__angle+math.pi)),
          __y1+((__b/2)*math.sin(__angle)),
__y1+((__b/2)*math.sin(__angle+math.pi))]
__coords = [(round(__Mx[0], 1), round(__My[0], 1)), (round(__Mx[1], 1),
round(__My[1], 1)), (round(__Mx[3], 1), round(__My[3], 1)),
           (round(__Mx[2], 1), round(__My[2], 1))]

for __x in range(math.ceil(min(__Mx)), math.floor(max(__Mx)) + 1):
   for __y in range(math.ceil(min(__My)), math.floor(max(__My)) + 1):
       __testPoint = Point(__x, __y)
       if __testPoint.within(Polygon(__coords)) or
__testPoint.intersects(Polygon(__coords)):
           __requisiteVertices.append((__x, __y))
```

Figure 8: Straight Path Requisite Point Python Code (Reyes)

In both sets of code, any coordinate points in the evaluation block that pass the tests, depending on the type of edge between (x_0, y_0) and (x_1, y_1), will form a set of requisite vertices, $\mathbb{R}$. The vertices in $\mathbb{R}$ only contain x and y values, without tangent values. Since the vehicle is traveling through, but not to these points, it only matters that these points exist in the graph. Now that the set is built, it can be confirmed that the edge is clear to be traversed. This is a simple test that if $\mathbb{R} \subseteq \mathbb{V}$ then (x_1, y_1) is reachable from (x_0, y_0) without any obstructions and the edge is valid. Otherwise it is discarded.

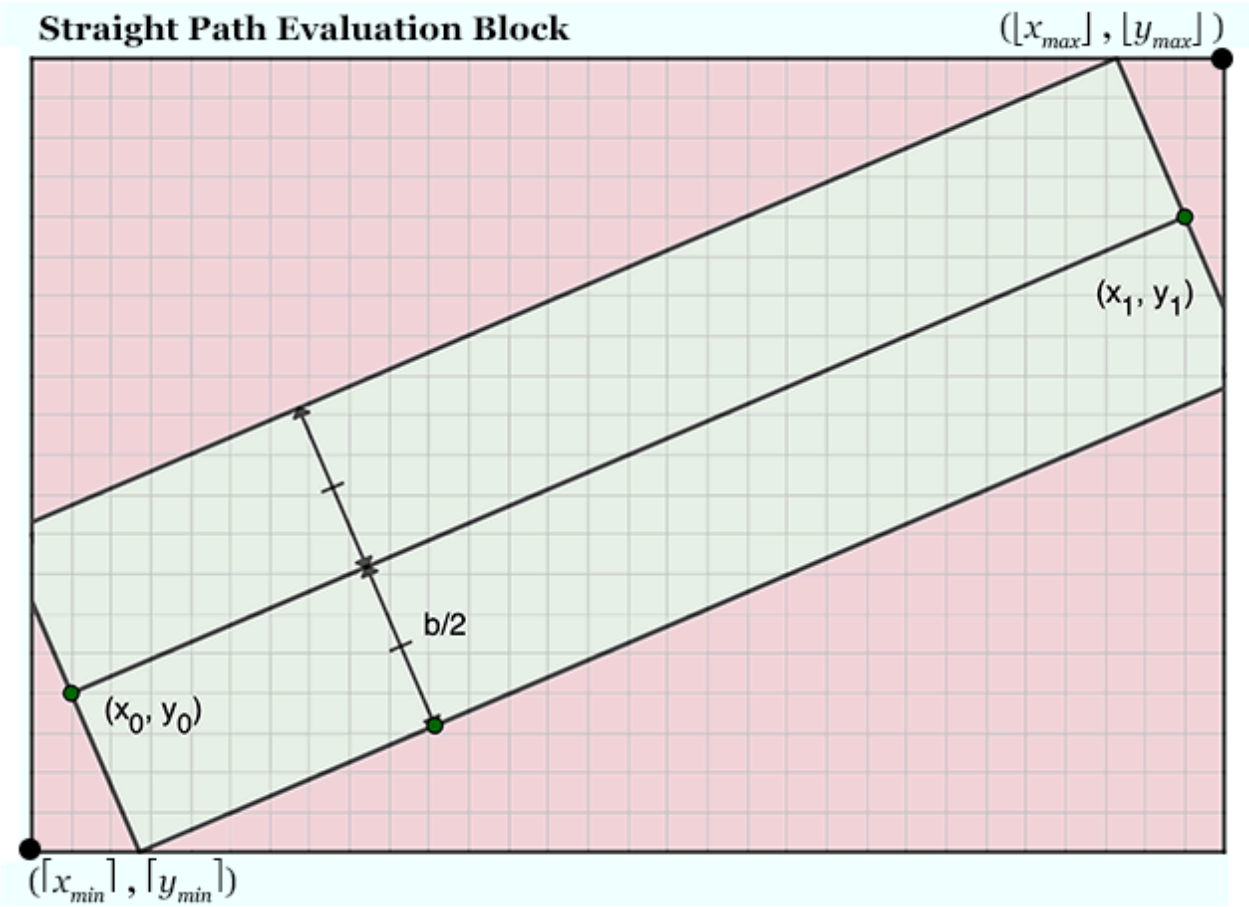

Figure 9: Visualization of Straight Path Evaluation Block (Reyes)

Edge Discovery

Now that the ability to determine the distance, tangent angle on arrival, and requisite vertices for an edge are defined, all of the edges that exist from any vertex (x_0 , y_0) at a certain angle to other vertices must be discovered. Since the algorithm will be dealing with 2d coordinates that in most cases will have large uninterrupted fields of vertices, there will be many edges that span a very large distance that only offer negligible difference in total distance between any two vertices. The user will need to specify a discretionary offset from the vertex to limit how far the algorithm will calculate edges. This offset will be called the *searchDepth*. For any given vertex with coordinates (x_0 , y_0) the edges for vertices with coordinates where $x_1 \in \mathbb{Z}$ and $-\ searchDepth \leq x_1 \leq searchDepth$ and $y_1 \in \mathbb{Z}$ and $-\ searchDepth \leq y_1 \leq searchDepth$ will be calculated (Figure 11).

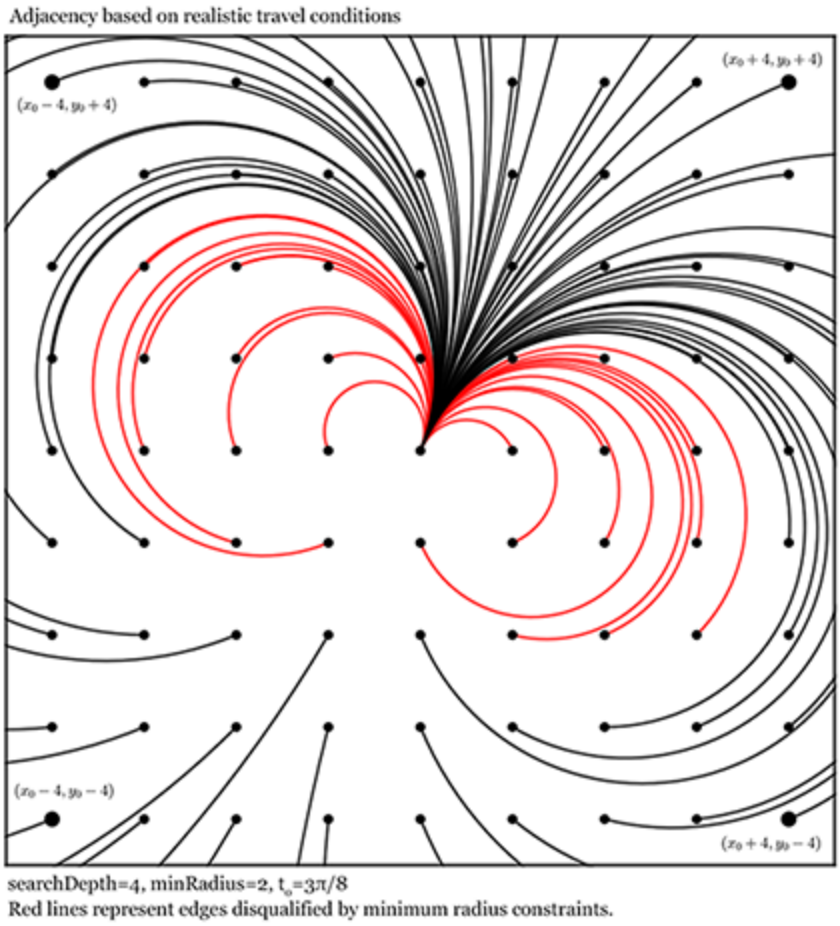

Figure 10: Visualization of Accurate Edge Paths with Proportional Distances (Reyes)

```
import edgeCalculation

def discoverEdges(__searchDepth, __beam, __minRadius, P):

    __edges = {}

    for __x in range(-__searchDepth, __searchDepth + 1):
        for __y in range(-__searchDepth, __searchDepth+1):
            for __precision in P.tangentList():
                __edgeData = edgeCalculation.edgeCalculation(0, 0, __x, __y, __precision,
__beam, __minRadius, P)
                if __edgeData is False:
                    continue

                if __edges.get(__precision) is None:
                    __edges[__precision] = {}

                __edges[__precision][(__x, __y, P.effectiveTangent(__edgeData['t1']))] =
{'distance': __edgeData['d'], 'requisiteVertices': __edgeData['requisiteVertices'], 't1':
__edgeData['t1']}

    return __edges
```

Figure 11: Edge Discovery Python Code (Reyes)

With the edge calculation, precision, and edge discovery combined, all of the elements necessary to

accurately define real world edges and real world destinations for vehicles with momentum and direction are defined. Figure 10 shows the edges of a two dimensional uninterrupted field of vertices for vehicles with momentum with edge length that's representative of actual distance.

Optimization

Optimization is fundamentally important with this extension as every vertex in a graph is expanded to a set of vertices with varying angles. The addition of multiples of the vertices combined with edge generation creates a scenario with a lot more to process than the Djikstra algorithm processes by itself.

Since the angles that will be reachable at any given coordinate is not known, following the initialization per the Dijkstra algorithm of the *dist* array and the *prev* array will lead to many unused entries. Due to the addition of the direction of travel, it will actually lead to many more unused entries, as the total entries would increase from $|\mathbb{V}|$ to $|\mathbb{V}|\bullet|\mathbb{p}|$. In addition, all of these entries will need to be added to the processing queue and processed unnecessarily, as many of these angles will not be reachable. Instead, the *dist* and *prev* arrays can have these vertices added to the queue as each reachable vertex for the current vertex being processed (Mehlhorn and Sanders):

```
if v not in dist:
   dist[v] = alt
   prev[v] = u
   fheappush(Q, (alt, v))
```

Figure 12: Alternate Dijkstra Python Code for Adding Vertices as Reached (Reyes)

Calculating the edges and determining the requisite vertices from a particular vertex at a particular angle is much more computationally expensive than any other single part of the algorithm. Running these calculations each time a vertex is processed from the queue would take a very long time. There is a solution that allows us to calculate these only once and apply the same calculations to each vertex. Since all of the vertices are separated by a fixed unit, they share the same distances and angles from each other. For example, the distance and angle between vertices (0, 0) and (1, 3) is the same as the distance and angle between vertices (2, 8) and (3, 11). The edge calculation will be identical as the relationship between the vertices is $(x,\ y) \rightarrow (x\ +\ 1,\ y\ +\ 3)$ for both. This principle can be applied to all vertices by just calculating all possible edges within the searchDepth block for vertex (0, 0) (whether it exists in the vertices of the graph or not doesn't matter), and using the x and y values as offsets from each vertex that is being processed from the queue. This is done by creating a cache during edge discovery for each valid tangent value for the precision setting. The returned edge discovery dictionary will contain keys for each

of tangent values that contain dictionaries for each vertex offset with its distance and requisite vertices. This requires a change to Djikstra's algorithm (Figure 13).

```
# Pseudocode in Djikstra's algorithm
# for each neighbor v of u:

# Changed to the following

for v, edgeData in edges[u[2]].items():
   v = (u[0] + v[0], u[1] + v[1], edgeData['t1'])
   requisiteVertices = set()
   for requisiteVertex in edgeData['requisiteVertices']:
       requisiteVertices.add((u[0] + requisiteVertex[0], u[1] +
requisiteVertex[1]))
```

Figure 13: Dijkstra's Algorithm Change Required for Requisite Point Functionality (Reyes)

There are a few instances where the data type used for certain operations is critical. There are multiple types that work, but many of them will take a large amount of computing time. Most of the potential for inefficiency revolves around lists of data. The largest list that the algorithm will be searching through is the vertices list, $\mathbb{V}$, when it is compared with the requisite vertice list for each edge, $\mathbb{R}$. For each edge it needs to be ensured that all of the vertices in $\mathbb{R}$ exist in $\mathbb{V}$. Intuitively, $\mathbb{V}$ would be a list, as it's a list of vertices. The problem is that searching for a value in a list data type has a time complexity of *O(n)*. This is because the operation needs to go through and check every value in the list to find a match. This operation needs to be executed for every single requisite vertex for every single edge in the graph. That's a lot of computing time. As an alternative, a data type that utilizes hash tables can be used for a much more efficient process. Data types that use hash tables run the value being searched for through a hash that returns an integer value that points to a specific memory location in the set (Yehoshua). If the location exists then the value is in the set. The *set* data type in Python contains data as a list of values but the "in" operation has a time complexity of *O(1)* (Badr). If *V* represents the number of vertices and *R* represents the number of requisite vertices, utilizing hash tables instead of standard lists will reduce the complexity from *O(R*V)* to *O(R)*.

The original definition of Dijkstra's algorithm requires iterating through the dist array every time a vertex is processed to find which one has the current lowest distance. This is a very slow process with a time complexity of *O(V)*. It's a common practice to use a binary heap tree to manage items that are processed based on some priority, in this case the distance, but there are considerations with using it in this case. A binary heap tree keeps the queue sorted in order of priority so it's fast to retrieve the next vertex to process (Adamchik). There is a heapify process each time another vertex is added to the queue but the speed improvement is so drastic that this processing time is negligible. There is a problem, however, with

updating the heap any time there is a new priority. Every time a better route is found to reach a vertex, its priority needs to be updated, which is not an efficient process. The algorithm only ever decreases the distance of a vertex in the heap, as it only changes when a better, shorter, distance is calculated, so the algorithm is only concerned with the decrease-key speed of the heap. For a standard binary heap the decrease-key time is *O(log n)* but as this is an operation that will be used a lot, a different heap type should be used that can perform this operation faster. A fibonacci heap can perform the same decrease-key operation in *O(1)* time complexity (Brodal et. al.). With the potentially large number of updates, that can create a huge time savings. However, a problem exists with both binary heaps and fibonacci heaps that affect the decrease-key operation. To decrease the key on a value in the heap, first it has to be found. The time to find a value in a heap is *O(n)* which takes away the entire efficiency advantage of using a heap over iterating a list. To get around searching the heap for the node every time its key needs to be decreased, a map of the heap can be created and updated so it doesn't have to be searched for values. To implement this, a dictionary will be created with the vertex as the key, and its location in the heap as the value whenever a node in the heap changes location. Since a dictionary uses hash tables, the insert, update, and retrieval operations all run in *O(1)* time. There are a lot of packages in circulation commonly used to implement fibonacci heaps, as they are commonly used in programming but none of them generate a map for quick node location. Rather than writing a whole custom fibonacci heap class to implement a map, a commonly available fibonacci heap class can just be modified to create or update a map entry whenever a node changes positions in the heap. This can be implemented into Quang Tran's fibonacci heap class that is commonly available with Python distributions by adding "`self.map = {}`" to the class's initialization function and adding "`self.map[x.key[1]] = x`" to each of the class's functions that modify the heap (Tran).

With an up-to-date map of the heap, the location of the node in the heap can now be retrieved in *O(1)* time and its key can be decreased in *O(1)* time with the following code:

```
Q.decrease_key(Q.map[v], (alt, v))
```

Figure 14: Fibonacci Heap Key Decrease Python Code (Reyes)

Results

The culmination of all previously discussed changes and optimizations (in addition to the Precision, edgeDiscovery, and edgeCalcuation functions) results in an altered version of Dijkstra's Algorithm (Figure 15). Additions are highlighted in blue, removals are written in red, and original Dijkstra pseudocode is highlighted in yellow with its Python equivalent written below it highlighted in green (in

some cases the code differs due to reasons already expressed in this paper but functionally accomplish the same idea as the original algorithm).

```
from sympy import *
import edgeDiscovery
import Precision
from fibonacciHeap import *

vertices = set()
# Code to populate vertices set from data source goes here

# Specify values per use case
minRadius = 2
searchDepth = 3
beam = 8
P = Precision.Precision(72)

dist = {}
prev = {}
Q = makefheap()

edges = edgeDiscovery.discoverEdges(searchDepth, beam, minRadius, P)

# Removed from Dijkstra for efficiency
# for each vertex v in Graph.Vertices:
#      dist[v] ← INFINITY
#      prev[v] ← UNDEFINED
#      add v to Q
#      Q.append(vertex)

# Set the starting point
# dist[source] ← 0
dist[(499, 1607, simplify(3*pi/2))] = 0

# Add only the source vertex to the queue
fheappush(Q, (0, (499, 1607, simplify(3*pi/2))))

# while Q is not empty:
while Q.num_nodes != 0:

    # u ← vertex in Q with min dist[u]
    # remove u from Q
    next = fheappop(Q)
    d = next[0]
    u = next[1]

    # for each neighbor v of u still in Q:
    for v, edgeData in edges[u[2]].items():
        # Calculate vertex and requisite vertices from offset
        v = (u[0] + v[0], u[1] + v[1], edgeData['t1'])
        requisiteVertices = set()
        for requisiteVertex in edgeData['requisiteVertices']:
            requisiteVertices.add((u[0] + requisiteVertex[0], u[1] +
requisiteVertex[1]))
```

```
if not requisiteVertices.issubset(vertices):
    continue

# Removed as we're adding points to queue as reached for efficiency
# if v not in Q:
#    continue

# alt ← dist[u] + Graph.Edges(u, v)
alt = dist[u] + edgeData['distance']

if v not in dist:
    dist[v] = alt
    prev[v] = u
    fheappush(Q, (alt, v))
# if alt < dist[v]:
elif alt < dist[v]:
    Q.decrease_key(Q.map[v], (alt, v))
    # dist[v] ← alt
    dist[v] = alt
    # prev[v] ← u
    prev[v] = u
```

Figure 15: Dijkstra's Algorithm Alterations and Comparison (Reyes)

As the majority of the algorithm is an extension to the Dijkstra algorithm, very little has changed in the algorithm itself. To recap, those changes are the disqualification of edges based on requisite points, and the various efficiency modifications. Since these are the only changes, the Dijkstra algorithm proof of correctness still holds true here.

Demonstrating the functionality of the extension can be done using small scale scenarios and having the extension process the paths to see if they are calculated as expected. The first scenario demonstrates how changes in the minRadius value alter the calculated path with a fixed precision, searchDepth, and beam when routing around a 10 unit wall. (Figure 16). It can be seen that increasing the minRadius value creates a wider turn of the appropriate turn radius and a longer path distance while still maintaining the beam clearance of 1 unit (½ unit on each side of the path). The dashed line provides a comparison of how the extension calculated paths vary from one calculated by the standard Dijkstra algorithm.

The next scenario aims to demonstrate that a route will be affected appropriately by changes in the beam value with a fixed precision, searchDepth, and minRadius (Figure 17). The scenario involves calculating a route to traverse a 39 unit by 34 unit graph containing a wall with holes of widths 4, 7, and 12 units using the extension. It can be seen that each calculated path routes through the wall using a hole that allows for adequate clearance for the specified beam. It can also be seen that when a path can fit through multiple holes, it travels through the one that will provide it with the shortest distance to its destination and passes

as close to the wall as possible to ensure that its path is the shortest possible path for its beam.

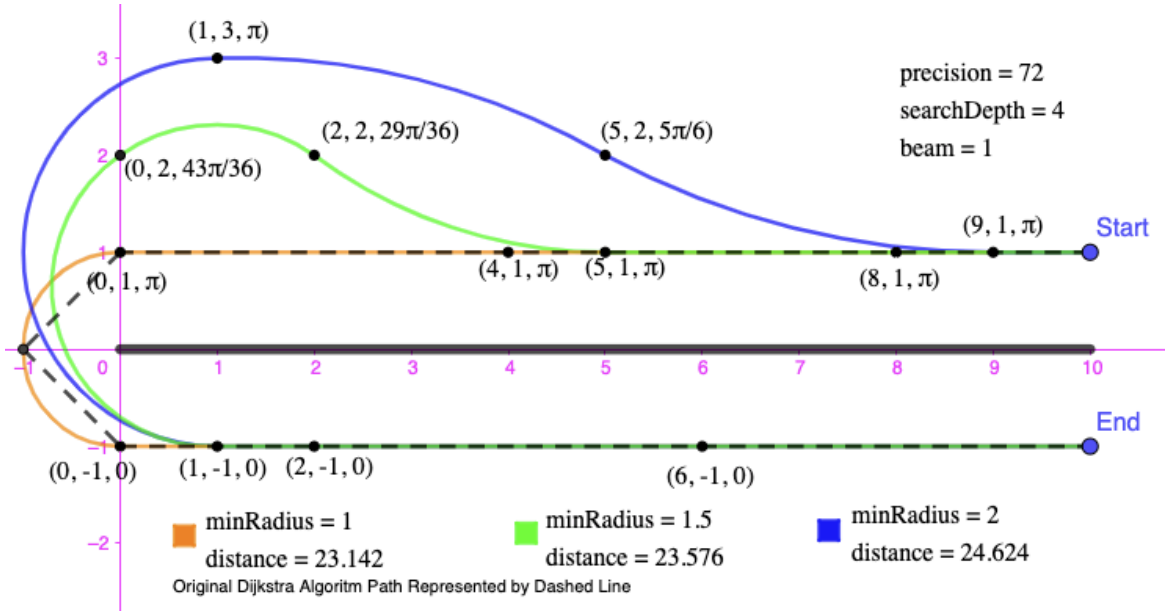

Figure 16: Algorithm Paths Around Wall (Reyes)

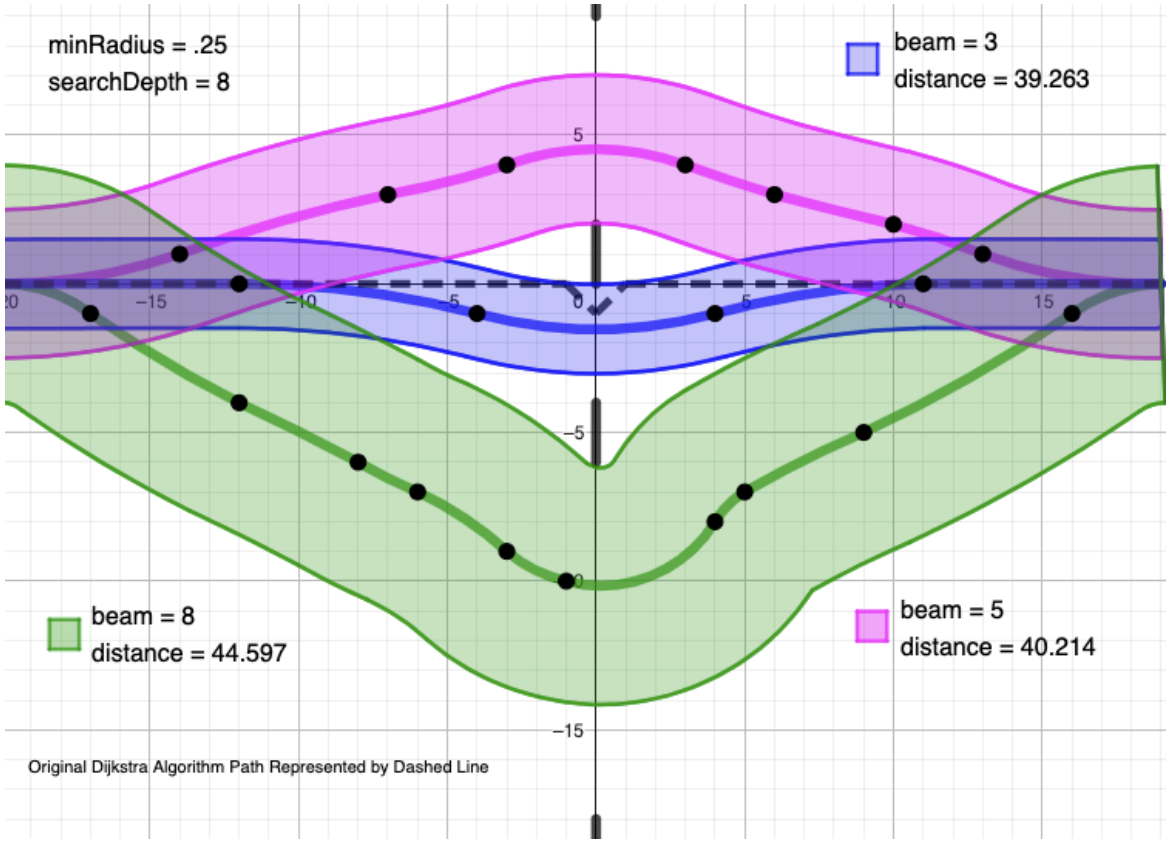

Figure 17: Algorithm Paths Through Holes (Reyes)

The extension's calculated shortest path can be shown on larger scale scenarios by using the Pillow

graphics library in Python to draw the arcs and lines of the path as well as the extremities of the beam clearance. The drawn path can then be overlaid onto the maps used to generate the vertice set. The Head of the Charles Regatta is a rowing race held annually in Boston every fall. Unlike many regattas, the Head of the Charles has numerous turns, five total, which makes the trajectory at any given point a critical factor. A path calculated using this extension can be overlaid onto an image showing turns four and five of the regatta from an arbitrary starting point (Figure 18). The image used to generate the vertice set was manipulated to account for the water passages under the Eliot Bridge, buoy lines, and removal of map text that doesn't exist in the real world (Google, n.d.).

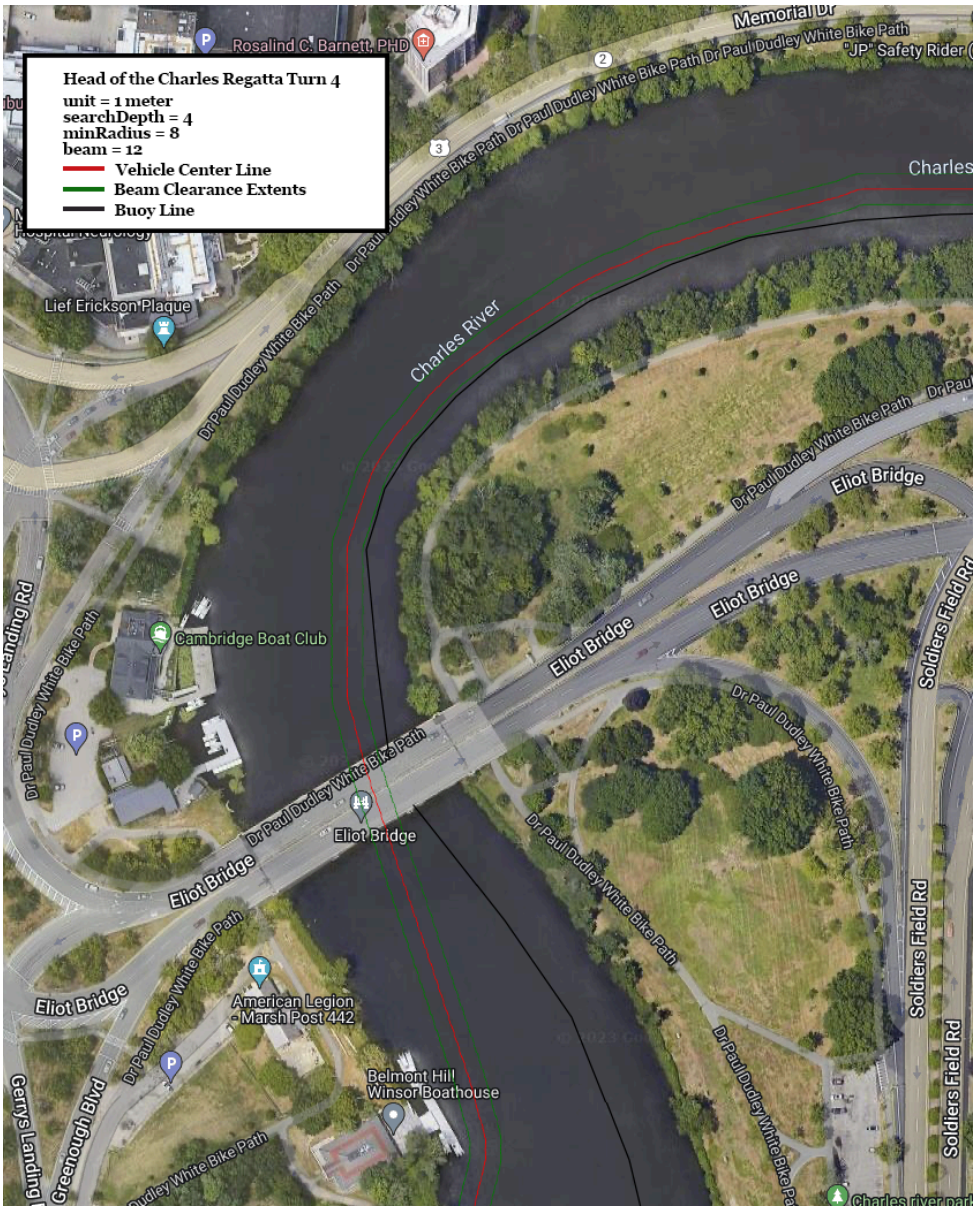

Figure 18: Algorithm Paths for Head of the Charles Regatta Turns 4 and 5 (Reyes)

It can be observed that the path takes what one would reasonably expect to be the shortest path through this turn without violating its clearance requirements or making any instantaneous turns that would be impossible to make. This example is a larger demonstration of one realistic path. The next demonstration involves a much larger scale, using a Google Maps generated map of the Dnieper River (Google, n.d.), a river on the border between Ukraine and Belarus (Figure 19).

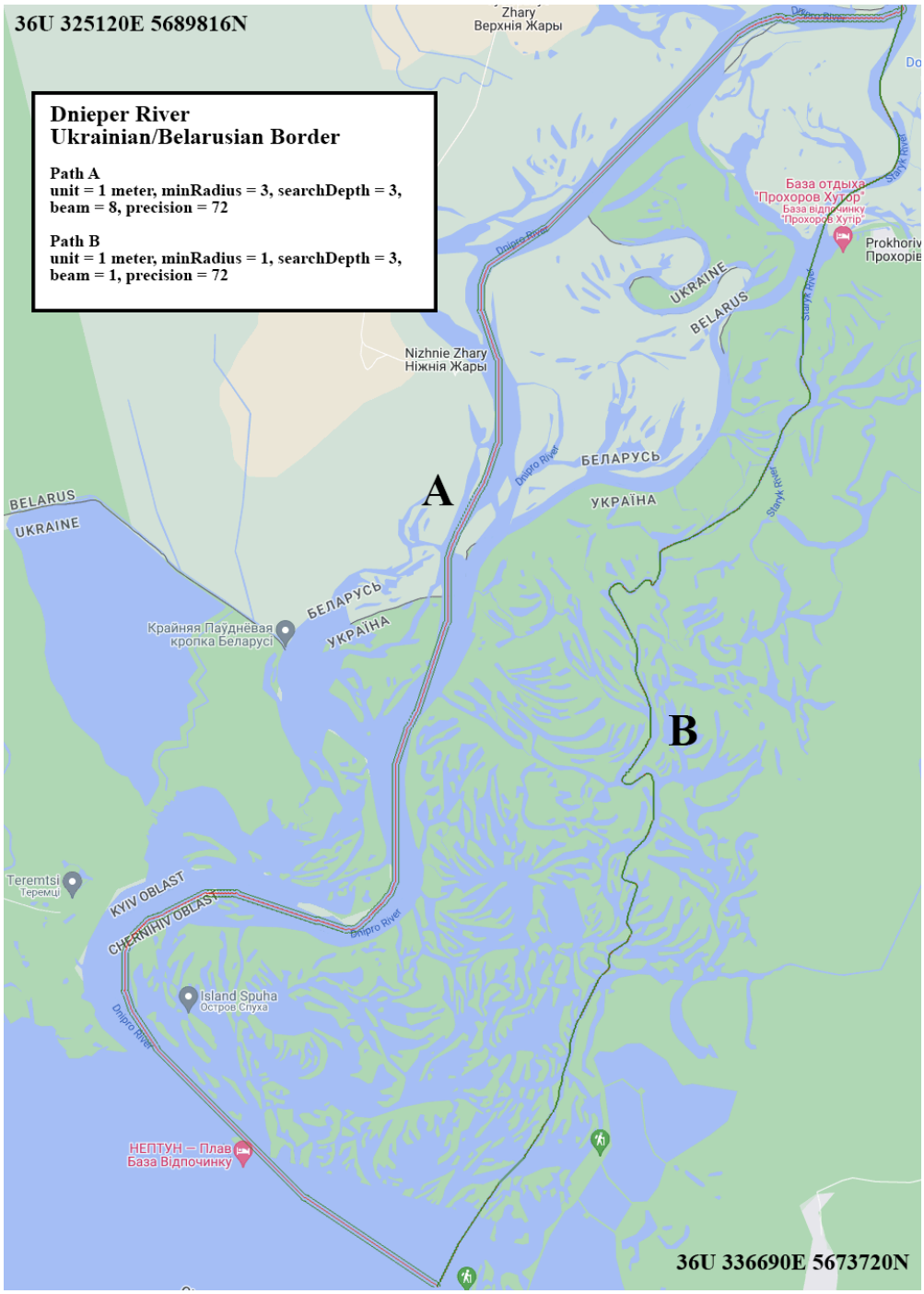

Figure 19: Algorithm Paths for Dnieper River (Reyes)

This river has conditions that change frequently, so the scenario only represents waterways present at the time Google Maps processed the satellite image. This image shows two calculated paths using the extension, each for vessels with varying turn radius and clearance requirements between two arbitrary points. While path B does not appear to show clearance lines, they are in fact much closer to the centerline of the path and appear as one thick line. As can be seen, the vessel that is able to travel via path B is able to navigate much smaller waterways with much tighter turn requirements. Some of them are so narrow you need to zoom in to see them when the path is laid over them visually. The vessel traveling path A must take a longer path due to its constraints. While path A is 24.2% longer than path B, it is still the shortest path that can be traveled by that vessel.

These real world examples are evidence of the algorithm's functionality as well as demonstrating suitable applications.

Future Work

This algorithm in its current state can benefit from additional work in the future in regards to both further constraints and travel efficiency. In regards to further constraints, in the real world while traveling through water or air there will be certain resistance in the form of either water current or air current. Other forms of similar resistance can exist but these are the most likely. The algorithm could be altered to account for a resistance vector at each vertice with the resistance magnitude and direction. In addition to adjusting edge distance, it would also need to account for a difference in vehicle speed, which in turn will affect the vehicle's minimum turn radius. This would add a significant amount of complexity that would likely provide negligible benefit for most scenarios, but for very large graphs, such as an aircraft navigating around weather systems crossing an ocean, the benefits would likely be significant.

For travel efficiency it is sometimes beneficial to reduce speed to take a tighter turn that may not be possible at full speed. In simple systems this would add little benefit, but for complicated graphs with many routes it may open up the possibility of taking much shorter routes if a vehicle were to slow down for a relatively short period of time.

Other algorithms that factor in other constraints that are outside of the scope of this algorithm exist to complement the Dijkstra algorithm. For certain use cases, an effort can be made to integrate this algorithm with others to create a robust system for specific use-cases.

Acknowledgements

Thank you to Regina Reals, Robin Clapper, and John Reyes for their support and guidance in the development of this project.

Bibliography

Abba, I. V. (2022, December 1). Dijkstra's algorithm – explained with a pseudocode example. freeCodeCamp.org. https://www.freecodecamp.org/news/dijkstras-algorithm-explained-with-a-pseudocode-example/

Adamchik, Victor. Binary Heaps. Carnegie Mellon University, https://www.andrew.cmu.edu/course/15-121/lectures/Binary%20Heaps/heaps.html. Accessed 3 July 2023.

Badr, Andrew. "TimeComplexity - Python Wiki." Python.org, 2017, http://wiki.python.org/moin/TimeComplexity. Accessed 15 Aug. 2023.

Brodal, Gerth Stølting; Lagogiannis, George; Tarjan, Robert E. (2012). Strict Fibonacci heaps (PDF). Proceedings of the 44th symposium on Theory of Computing - STOC '12. pp. 1177–1184. CiteSeerX 10.1.1.233.1740. doi:10.1145/2213977.2214082. ISBN 978-1-4503-1245-5.

Burns, Carol. "Length of a Circular Arc (and Related Concepts)." Www.onemathematicalcat.org, 2004, http://www.onemathematicalcat.org/Math/Precalculus_obj/lengthOfCircularArc.htm. Accessed 5 Aug. 2023.

Ellis, Daniel. "Calculating the Bearing between Two Geospatial Coordinates." Medium, 25 May 2020, http://towardsdatascience.com/calculating-the-bearing-between-two-geospatial-coordinates-66203f57e4b4. Accessed 1 July 2023.

Ghosh, Subhodeep. "Understanding the Beam of a Ship." Marine Insight, 17 Nov. 2022, http://www.marineinsight.com/naval-architecture/understanding-the-beam-of-a-ship/. Accessed 1 July 2023.

Gillies, Sean and Shapely contributors.. "Shapely — Shapely 1.8.0 Documentation." Shapely.readthedocs.io, shapely.readthedocs.io/en/stable/. Accessed 5 Aug. 2023.

Google. (n.d.). [Google Maps of Charles River and Dnieper River]. Retrieved May 4, 2023.

Haber, Howard. Physics 116A the Argument of a Complex Number. 2011.

Insall, Matt; Rowland, Todd; and Weisstein, Eric W. "Point Lattice." From MathWorld--A Wolfram Web Resource. https://mathworld.wolfram.com/PointLattice.html

Lewis, Rhyd. A Comparison of Dijkstra's Algorithm Using Fibonacci Heaps, Binary Heaps, and Self-Balancing Binary Trees. 2023.

Mehlhorn, Kurt; Sanders, Peter (2008). "Chapter 10. Shortest Paths" (PDF). Algorithms and Data Structures: The Basic Toolbox. Springer. doi:10.1007/978-3-540-77978-0. ISBN 978-3-540-77977-3

Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP,

Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. (2017) SymPy: symbolic computing in Python. *PeerJ Computer Science* 3:e103 https://doi.org/10.7717/peerj-cs.103

Mitchell, Ryan. (2021, August 2). The algorithms behind the working of google maps. CodeChef. https://blog.codechef.com/2021/08/30/the-algorithms-behind-the-working-of-google-maps-dijkstras-and-a-star-algorithm/

Moore, L. (1997). Transverse mercator projections and US Geological Survey digital products. US Geological Survey, Professional Paper. http://www.geo.utexas.edu/courses/420k/PDF_files/LABS/GIS/map_project.pdf

Nykamp DQ, “Polar coordinates.” From Math Insight. http://mathinsight.org/polar_coordinates. Accessed 18 June 2023.

Schrijver, Alexander (2012). "On the history of the shortest path problem" (PDF). Documenta Mathematica.

Sedgewick, Robert, and Kevin Daniel Wayne. Algorithms. 4th ed, Addison-Wesley, 2011.

Tran, Quang. “Fibheap: An Implementation of Fibonacci Heap.” PyPI, http://pypi.org/project/fibheap/. Accessed 30 July 2023.

Yehoshua, Dr Roi. “List vs. Set in Python.” Medium, 3 Aug. 2023, http://levelup.gitconnected.com/list-vs-set-in-python-4ea183dfe20c. Accessed 30 Aug. 2023.