Kalamazoo Route Planner Design Document

# Overview

## Team Members

* Austin Miller
* Bjarne Wilken
* Christian Fuentes
* Matthew Phillips
* Mike Henke

## Project Description

The Kalamazoo Route Planner is a custom GPS website for the Kalamazoo area. The main focus is generating directions for people looking to walk or bike. Our application takes into account a risk factor meaning you can choose how safe you want the roads you take to be. Once the path is generated you will be able to see it on the map and also will be able to export it to a .gpx file to be used in an app like google maps. Alongside that it also has many points of interest available to show on the map some of which include bike racks, grocery stores, and restrooms.

## Project Objectives

* Have a clean and intuitive user interface
* Display points of interest on the map
* Generate an optimal path to the destination
* The pathfinding should be relatively quick
* Allow the user to set a risk level for the path
* Allow user to export path to .gpx file

# Application Architecture

## Architecture Overview

* **Frontend:** Built using HTML, CSS, and JavaScript, responsible for user interaction and displaying the map and route
* **Backend:** Utilizes Python to handle route calculations and database interactions and the Flask library as the web framework
* **Database:** Uses SQLite to store map information from the OpenStreetMaps api

## Components

* **User Interface:** Provides a dashboard for user input and interaction and displays the map, markers, and the route
* **Pathfinding:** Uses the A\* pathfinding algorithm to calculate a route based on user input
* **Map Service:** Ulilizes the Leaflet and OpenStreetMaps api’s to display and get map information
* **Flask:** Controls the website and handles sending data between the frontend and the backend

# Functionality

## User Features

* **Route Creation:** Allows the user to place two markers that will generate an optimal route between the two
* **Map Display:** Displays the generated route on an interactive map that allows for zooming and panning, it also keeps the user within a bounding box for the Kalamazoo area
* **Amenities:** Allows the user to toggle different points of interest to be overlaid on the map
* **Route Customization:** Adds options for the user to choose their mode of transportation and adjust the risk level of the route being generated
* **Export Route:** Lets the user export their route to a .gpx file to be uploaded to something like google maps

## Technical Features

* **A\*:** This was the best fit for our project because the amount of nodes is so high it would take forever using an algorithm that doesn’t take into account distance to the end. We also modified the algorithm to allow for it to take into account the risk level of the road when generating a route
* **OpenStreetMaps:** Pulled map data from their api that we save in our database and use to generate routes
* **Export Route:** Takes the generated route and creates a file the user can save in the .gpx format, this file can be uploaded to another map software such as google maps that will read the file and give directions based on it
* **Database:** The data from the OpenStretMaps api is parsed and stored in the database creating connections between all of the different nodes as it builds itself

4. Pseudocode

4.1. Frontend

4.1.1. Routeplanning.html

<Head>

Set up initial HTML site settings and external documents and libraries

</Head>

<Body>

Set up side bar with required buttons

Set up sub lists for each drop down

transportation types

risk tolerance

amenities

Other buttons to be set up in navigation bar

Export GPX

Clear

Help

About

Set up direction sidebar

Set up information in About and help popups

import leaflet

import jquery

import map.js

import marker.js

import navbar.js

</body>

4.1.2. Map.js

Set up map boundaries

Create map object

Create boundary line on map

Create scale object

Create zoom object

Create Map layers

Function changeLayer

if osmlayer exists

change to OSMLayer

else if osmcycl layer exists

change to cyclelayer

Else if satellite layer exists

Change to satellite

Start map on OSMLayer

Set up listener on Layer changing button

4.1.3. Marker.js

Set up variables

Function createMarker

if user clicked in the boundaries on the map

if number of markers less than 2

if marker with id of 1 is undefined

create start marker

If marker with id of 2 is undefined

Create end marker

Add markers to marker layer

Add layer to map

If number of markers equals 2

Passtoflask()

Else

Removepathline()

Else

Display alert

Async function passtoflask()

AddDirSidebar()

AddLoader()

Get user risk tolerance

Get user transportation type

Fetch

Then fetch json

Select direction sidebar

Then drawpathline and adddirections

Function drawpathline(path)

For the length of path

create line object

add line to line array

add line to marker

Function adddirsidebar()

If mobile

set up direction sidebar for mobile

Else

set up direction sidebar for desktop

Function addLoader()

Create html element

Set innerhtml

Add loader class

Appen dots to loader

Append to directions sidebar

Function addDirections

AddDirSideBar()

Set avgrisk to 0

Set userRisklower to false

Loop for length of directions

add risk from direction to avg risk

add distance from direction to total distance

set up dirTag html element

set dirTag innerhtml

set dirTag text color to white

if direction risk is more than userRisk

set userRiskLower to True

set dirTag background color

Append dirTag to dirSideBar

If length of directions is 0

Set up errTag html element

Set html elements text, color, and background

Append errTag to dirSideBar

Else

Calculate avg risk of whole path

Create distTag html element

If userRiskLower is true

Set distTag to display distance and warning

Else

Set distTag to display total distance only

Set distTag color

Append distTag to dirSideBar

Function hideDirections()

If mobile

Hide directions to bottom of screen

Else

Hide directions to right of screen

Clear directions from window

Function createAmenMarkers(amens, id)

Create layer group for amens

Based on id create icon object

Loop amens length

set marker icon and location

set marker desc and name

bind pop up to that marker

Push amens onto layer

Add amens to map

Function setDest()

If start marker is undefined

set marker object

add to map

If end marker is undefined

set marker object

add to map

If marker layer size is 2 and no directions

passToFlask()

Function deleteAmenMarkers()

Loop based on layer length

if layer id is amens layer id

clear layers

remove layer from map

splice layers together

break

Function deleteAllMarkers()

Clear all markers

Set lines to empty

Clear marker layer

Remove marker layer from map

HideDirections()

Function removePathLine()

Loop based on length of lines

remove each line from map

Function deleteMarker()

RemovePathLine()

Set lines to empty

Set directions to empty

Remove this marker layer

Delete this id from markers

HideDirections()

Function newCoords()

RemovePathLine()

Set marker id, latitude, and longitude

Clear directions

PassToFlask()

Function mobileAndTabletCheck()

Set check to false

Function to check all possibilities

Return check

Add listener to map

Get dirSideBar

Check for mobileUsers

Open Navigation

4.1.4. Navbar.js

Get html elements

Function openNav()

open navigation bar when icon is clicked

Function closeNav()

close navigation bar when icon is clicked

Function exportGPXFile()

if path is not empty

pass information to backend views

For dropdown length

create drop down based on length

Function open()

open help or about page

Function close()

close help or about page

Function getinput()

store transportation type value

store risk tolerance value

Function changeAmenMarkers(event)

when checkbox is clicked store info from click

create post request

set up await

on return

parse returned JSON

if amen is checked

create that amen marker

else

delete amen marker

create post and send

Function clear

Remove all objects from map and uncheck boxes

Create event listeners for all html elements

4.2. Backend

4.2.1. main.py

import libraries

get\_env():

check for .env file

if exists

load variables

else

set default values

run\_website():

set up flask

register flask blueprint

run the flask server

if main

get system arguments

if system arguments is 1

run\_website()

if –t is a system argument

run tests()

else

print error

4.2.2. install.py

import required libraries

install(package):

check call with system and run given command

call install on all required packages

4.2.3. views.py

import required libraries and packages

set up flask blueprint

set locations as empty

set flask views function call and get or post methods

homepage():

return render html template

set flask views function call and get or post methods

calculate\_route(markerInfo):

print info

load JSON info

set start, end, risk\_tol, and transport variables

set error

get start time

while we have not found a path or risk\_tol too high

initialize pathfinder object with variables

call astar

increment risk\_tol

get end time

get time difference and display on server side

if path not found

return empty list to front end

else

return path and directions to front end

set flask views function call and get or post methods

get\_amenities(amen\_type):

load JSON data from front end

initialize data\_retriver object

connect to database

try to get amenity data

if not print error

close database connection

return amens to frontend as JSON

set flask views function call and get or post methods

get\_gpx(path\_list):

create GPX\_file

get file name created for file

create a binary stream

open binary file for reading

copy data over to new file

set pointer to start of new file

delete old file

set new file name to current date time

return file to frontend to be downloaded

4.2.4. pathfinder.py

import required libraries

Node Class:

init(data):

set up initial variables

set\_g(g):

set g value

set\_h(h):

set h value

set\_parent(parent):

set Node parent

get\_f():

return f value

get\_g():

return g value

Pathfinder Class:

init(start, end, transport, risk\_tol):

set initial values

create data\_retriver object

open database connection

get\_q(node\_list):

set small index to 0

set small value to 9999999

loop over node\_list

if current nodes f value is less than small value

set small value to nodes f value

set small index to i

return small index node

nodify(node\_list, parent):

set list to empty

loop over node\_list

create new Node

set Nodes parent

append node to list

return list

is\_in(node, node\_list):

loop over node\_list

id node data is the same as node\_list data

return True

return False

denodify(node):

loop forever

append node data to path list

set node to node parent

if node is NONE

break from loop

append start nodes to path list

reverse path list

astar():

set start time

find closest nodes in graph to user nodes

find the closest intersections to those nodes

append end nodes to path list

set up open and closed list

append start node to open list

loop while open\_list still ahs nodes

get the q node from open\_list

get that nodes neighbors based on transport type

nodify that data

loop through neighbors

if data equals end node

set found to True

set last\_node to neighbor

break from loop

set neighbor node g and h values

if neighbor is\_in open or closed list

continue at start of loop

else append neighbor to open\_list

append q node to closed list

if end node found:

break from loop

get current time taken

if total time taken is more than 40 secs

last node set to NONE

break from loop

if last node is NONE:

return –1

denodify()

\_assemble\_lat\_lng()

\_assemble\_directions()

return 1

\_assemble\_directions():

get path length

set prev\_dir to empty

set prev\_path\_name to empty

set distance to 0

loop over length of path – 2:

set lat and long differences

increment distance on distance between nodes

set cardinal direction

set direction string if path name or card dir change

append direction to directions list

\_assemble\_lat\_lng():

loop over path

append lat and long to list

\_calculate\_distance\_between\_nodes(n1, n2):

return distance between nodes

\_find\_closest\_connector(node):

if node is already a connector

return node

get ways of node from database

get connector nodes from database from way value

if only 1 connector node

return connector node

set closest index and shortest\_dist to 0 and none

loop over connector nodes

calculate distance from node and conn node

if shortest\_dist is none:

set shortest\_dist

set closest index to this one

if current distance is shorter than shortest dist

set shortest distance

set closest index to this node’s

return closest connector node

\_find\_next\_best\_user\_node(user\_node):

get list of nodes from database of nodes closest to user’s

set best distance of distance of first node

loop over list of nodes

if a node is closer to user’s

set best node to that node

return closest node

\_get\_cardinal\_directions(lat\_diff, lng\_diff):

if lat is more than lng:

if lat in path is positive:

set card\_dir to North

else:

set card\_dir to South

else:

if lng path is positive:

set card\_dir to East

else

set card\_dir to West

return card\_dir

return\_directions():

return directions list

return\_path()

close database connection

return lat long list

4.2.5. data\_retriever.py

import required libraries and packages

Data\_retriever class:

init():

Set up initial variables

Connect():

Try to connect to the database via sqlite connector

If failed try another path and return error

Close():

Try to close the connection

If fails print error

Get\_amenities(amen\_type):

Execute sql query with amen\_type

Fetch all results and store in amens

Create an empty amens\_dict list

Loop through amens

build a dictionary for amenity

append dictionary to amens\_dict list

Return amens\_dict list

Get\_closest\_nodes(user\_marker, transport\_type, risk):

Loop forever:

Set east\_long

Set west\_long

Set north\_lat

Set south\_lat

Execute query based on lats and longs

Fetch query results and store in nodes

Set num of nodes to length of nodes

Loop backwards through returned nodes

set index to last index in list

if transport is walk

if not \_is\_node\_walkable(current node):

remove node from nodes list

if transport is bike

if not \_is\_node\_bikable(current node):

remove node from nodes list

If length of nodes is not 0:

return nodes list

else:

increment mag variable by 50

Get\_connector\_nodes(way\_id):

Set nodes to get\_nodes(way\_id)

Set connectors to empty list

Loop through nodes

set data to get\_node\_info(node id)

if connector status is 1:

append data to connectors list

Return connectors

Get\_node\_info(node\_id):

Execute sql query on database

Fetch and return query results

Get\_nodes(way\_id):

Execute sql query on database with way\_id

Fetch all results and store in data

Set nodes to empty list

Loop through data returned

append to nodes list get\_node\_info(id from data)

Return nodes list

Get\_way(node\_id):

Execute sql query from database with node\_id

Fetch and store results in data

Set ways to empty list

Loop through data

append id from data to ways

Return ways

Get\_way\_info(way\_id):

Execute sql query on database with way\_id

Fetch and return results of query

Get\_node\_neighbors(node\_id):

Execute sql query on all\_links from database with node\_id

Fetch all results and store in temp

Set neighbors to empty list

Loop through temp

append get\_node\_info(ids from temp) to neighbors

Return neighbors

Get\_connector\_node\_neighbors(node\_id):

Execute sql query on connector\_links from database

Fetch all results and store in temp

Set neighbors to empty list

Loop through temp

append get\_node\_info(ids from temp) to neighbors

Return neighbors

Get\_node\_coords(node\_id):

Execute sql query on nodes from database with node\_id

Fetch and return results

Get\_walking\_neighbors(n\_id, risk):

Set walking\_neighbors to empty list

Set neighbors to get\_connector\_node\_neighbors(n\_id)

Set start\_ways to get\_way(n\_id)

Set start\_len to length of start\_ways

Loop through neighbors

set end\_ways to get\_way(node id from neighbors)

set end\_len to length of end\_ways

if start\_len or end\_len is more than 1

find matching ways

else

set way\_id to end\_ways id

set end\_way\_info to get\_way\_info(way id)

if road type is in walking types:

append node to walking\_neighbors

Return walking\_neighbors

Get\_biking\_neighbors(n\_id, risk):

Set biking\_neighbors to empty list

Set neighbors to get\_connector\_node\_neighbors(n\_id)

Set start\_ways to get\_way(n\_id)

Set start\_len to length of start\_ways

Loop through neighbors

set end\_ways to get\_way(node id from neighbors)

set end\_len to length of end\_ways

if start\_len or end\_len is more than 1

find matching ways

else

set way\_id to end\_ways id

set end\_way\_info to get\_way\_info(way id)

if road type is in biking types and risk less risk:

append node to biking\_neighbors

Return biking\_neighbors

\_is\_node\_bikable(n\_id, risk):

Set ways to get\_way(n\_id)

Loop through ways

get way\_info

if road type is in biking types and risk is less risk:

Return True

Return False

\_is\_node\_walkable(n\_id, risk):

Set ways to get\_way(n\_id)

Loop through ways

get way\_info

if road type is in walking types and risk is less risk:

Return True

Return False

Reset\_mag():

Set mag to 50

Get\_path\_name\_risk(n\_id\_one, n\_id\_two, query):

If query is 1:

execute sql query on all\_links from DB

Else:

execute sql query on connector\_links from DB

Fetch and store in way

Execute sql query on ways from DB

Fetch result and store in path\_info

If path\_info name is None:

return road type and risk factor

Else:

return road name and risk factor

4.2.6. gpx\_export.py

import required libraries and packages

GPX\_export Class:

init(path):

set path\_string to path

set path to empty list

set\_path(path):

set path to path

get\_path():

return path

parse\_string\_to\_list(input\_string):

strip away all new lines, tabs, and spaces in input string

remove square brackets from end

get coordinates by themselves as coordinate strings

loop through coordinate strings:

strip away unnessecary characters from string

convert strings to floats and store in new list

set path list to new list

export():

parse\_string\_to\_list(path\_string)

set file as an open file

create new gpx object

create first track on gpx

create first track segment

name the track

add the track to the gpx object

write the gpx file

close the file

Clean\_up():

Delete created gpx file

4.2.7. DB\_create.py

import required libraries and packages

try connecting to DB with sqlite connector

if fails print error and return

create a cursor object to execute sql queries

drop all tables if DB exists already

Create new tables for DB

Query OSM for node data in Kalamazoo and Portage

insert nodes into DB

Query OSM for ways data of Kalamazoo and Portage

insert ways into DB

link nodes together in the process

remove nodes and ways associated with interstates from DB

Set nodes as a connector node or not

link connector nodes together

open csv files with risk levels

update ways with the associated risk value

create indexes in the DB for faster searching

open KML with all amenity data

parse amenity data and insert into DB

commit and close DB connection