Kalamazoo Route Planner Design Document

1. Overview

1.1. Team Members

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

1.2. 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.

1.3. 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

2. Application Architecture

2.1. 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

2.2. 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

3. Functionality

3.1. 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

3.2. 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