CS/ECE/ISyE 524 — Introduction to Optimization — Spring 2023

Final Course Project: Due 5/5/23

Ideal Apartment Locations Near Campus of UW - Madison

Skylar Hou whou25@wisc.edu

Gillian Wang gwang2324@wisc.edu

Isabel An ban22@wisc.edu

Table of Contents

1. Introduction

Finding a suitable place to live is one of the most crucial aspects to consider when relocating to a new city or town for work or school. At the University of Wisconsin-Madison, individuals have diverse requirements when it comes to selecting an ideal apartment location. In this project, we aim to figure out the most convenient location if a property plans to build a new apartment. Our question comes about by identifying a real-world problem of selecting an ideal apartment location for a property company to build the most convenient student apartment near the University of Wisconsin-Madison. Our project has the potential to have an impact on the housing market in the University of Wisconsin-Madison area. By determining the most convenient location, property developers can attract potential tenants who value convenience, potentially increasing the occupancy rates and profitability of the new apartments.

To find the best location with the most convenient access to health services ("hospital.csv"), restaurants ("restaurants.csv"), university buildings, groceries ("supermarkets.csv"), and free bus stops ("bus_stops.csv"), we downloaded and scraped address and location data (longitude and latitude) of these places using Nominatim API (https://nominatim.openstreetmap.org/search.php?q=Madison+WI&format=jsonv2) and

Stops information of Metro Transit in CityOfMadison (https://www.cityofmadison.com/metro/routes-schedules/accessible-bus-stops). To consider as many aspects of students' life as possible, we divided the data of university buildings into three datasets, separately focusing on all university buildings ("university.csv") (since there are many buildings on campus, when considering the distance to buildings in this dataset, more consideration is given to the distance to the entire campus area), the center academic buildings of nine main schools or colleges ("academic_buildings.csv"), and the most popular libraries and recreation centers ("library_recreation.csv") on campus. Technically, we utilized the Geopy.geocoders package and Google Maps to extract the latitude and longitude of some addresses without location information.

Specifically, in the process of cleaning the bus stops data ("bus_stops.csv"), we filtered Routes 80, 81, 82, and 84 from the original stops tables to measure the accessibilities of free metro services provided by the university considering that some students may not have or lose their bus pass to take other routes. Otherwise, as some of the closed bus stops presented by CityOfMadison (https://www.cityofmadison.com/metro/routes-schedules/accessible-bus-stops) are not updated, we deleted them to make sure the authenticity of our data.

In the rest of our report, we build our model considering two conditions (does not include and includes the metro transit). We will explain our methodologies of these two models in detail, elaborate on the mathematical models, present steps we solve our MIQP and MISOCP problem using Gurobi, and talk about our results with or without taking bus, conclusion, and limitations in the last sections.

2. Methodology

To take into account the different modes of transportation that people prefer, we build two models, separately prioritizing walking and taking the bus to commute.

2.1 Model Without Metro Transit

Assuming that most students do not own a car, they tend to live on or near the campus area. Thus, we set a border rectangle to make our data points more concentrated. For measuring the levels of convenience of places, we calculated convenience scores for places we choose by taking the averages of Manhattan distances from each location to other ones, as the amounts of different-type places (health services, restaurants, university buildings, supermarket) are different.

2.2 Model With Metro Transit

Rather than minimizing the walking distance as we did in 2.1, at this time we take the bus routes into consideration. Since the number of bus stops is numerous and would be hard

to implement, we idealize a rectangle as the bus routes 80, 81, 82, and 84. In the process of calculating the cost of the starting point and the ending point, we assume that the difference between places to get on the bus, and the distance on the bus are not considered. We calculate the minimum Manhattan distances between the starting points and the border of the rectangle, and the minimum Manhattan distances between the border of the rectangle and the ending point. In addition, we set two types of logical constraints, first to calculate the distances between the origion and the rectangle, and second to compare the walking distances with or without taking bus in the middle.

3. Mathematical model

3.1 Model Without Metro Transit

Assumptions:

- Calculate distances to different label of buildings to measure the convinience.
- The actual distance between the two points is approximated as the Manhattan distance by taking into account the real situation.

Constants:

- $weight_i$ means the weight for label i.
- lat_i, lon_i is the vector stores latitudes and longtitudes for buildings of label i.
- n_i is the amount of buildings of label i.
- num_i is the number of nearest buildings took into consideration of label i.

The fomulation of this model:

$$\min_{loc\in\mathbb{R}^2,\, b_i,\, dist_i,\, t_{1i}\,,\, t_{2i}\,\in\mathbb{R}^{\mathrm{ni}}} \quad \sum_{i\ in\ labels} \quad \sum_{j\ in\ 1:n_i} dist_{ij} \ * \ b_{ij} \ * \ weight_i$$

$$subject\ \ to:$$

$$dist_i \ = \ t_{1i} \ + \ t_{2i}$$

$$-t_{1i}[j] \leq loc[1] \ - \ lat_i[j] \leq t_{1i}[j]$$

$$-t_{2i}[j] \leq loc[2] \ - \ lon_i[j] \leq t_{2i}[j]$$

$$\sum_{j\ in\ 1:n_i} b_i[j] = num_i$$

$$b_i[j] \in [0, \ \ 1]$$

$$\text{for i in labels, j in 1:ni}$$

Dicision variables:

- $location = [lat, \ \ lon]$, the optimal location for new apartment.

For buildings of each {label} from $labels \ = \ [hos, \ rest, \ mkt, \ camp, \ univ, \ lib]$:

- $b = [b_i]$, for i in [1, amount of buildings with {label}], b_i is binary, to decide how many nearest buildings to take into consideration.
- $dist = [d_i]$, for i in [1, amount of buildings with {label}], d_i is the manhatten distance from the optimal location to each bulding with {label}.
- $t_1[t_{1i}]$ and $t_2[t_{2i}]$, for i in [1, amount of buildings with {label}], to get the abstract value of latitude and longitude distance.

Constraints:

- The constraints in this model are used to calculate the Manhattan distance, and to select the nearest num_i buildings to calculate average distance.

Objective function:

- The average Manhattan distance to each building of each type is used to express the level of convenience, and they are weighted by given $weight_i$.

Model type:

- It is a MIQP. Binary variables is used because of we want to only consider the nearest buildings (e.g. minimize the distance to nearest 1 hospital). Although theoratically Gurobi cannot solve this, after some attempts for other solvers and with different implemantations, solving this problem with Gurobi is still the way that prints the accurate result with much simpler implementation.

3.2 Model With Metro Transit

Assumptions:

- Calculate distances to different types of buildings, but are the minimum between walking distances and bus distances, which is a change compared to previous model.
- The actual distance between the two points is approximated as the Manhattan distance by taking into account the real situation.
- Idealize multiple bus routes in reality as a rectangle, where buses can be taken anywhere at the rectangle bound with a fixed duration by $bustime$.
- The distance from the origin or destination to the bus rectangle is still calculated with the Manhattan distance.

Constants:

- $weight_i$ means the weight for label i.
- lat_i, lon_i is the vector stores latitudes and longtitudes for buildings of label i.
- n_i is the amount of buildings of label i.
- num_i is the number of nearest buildings took into consideration of label i.
- bus_{2i} is the vector stores the smallest manhattan distance from bus regcantle to each destination building of label i. It is culculated by a for loop outside the model.

- $bustime$ is the fixed time of taking bus.
- $busx_1$, $busx_2$, $busy_1$, $busy_2$ are the longitude and latitude of two vertices of the bus rectangle.
- $recx_1$, $recx_2$, $recy_1$, $recy_2$ are the longitude and latitude of two vertices of the range rectangle we selected from the map to consider in this project.
- M_1, m_1 are the max bound and min bound for some expressions.
- eps is the epsilon for the logic constraint.

The fomulation of this model:

$$\min_{\substack{loc\in\mathbb{R}^2 \\ u_i\in\mathbb{R}^4,\, z_i\in\mathbb{R}^{\mathrm{ni\ x\ 2}} \\ bus_i\in\mathbb{R}^1 \\ b_i,\, dist_i,\, t_{1i}\,,\, t_{2i}\in\mathbb{R}^{\mathrm{ni}}}} \sum_{i\ in\ labels} \sum_{j\ in\ 1:n_i} dist_{ij} * b_{ij} * weight_i$$

$$subject\ to:$$

$$dist_i[j] = z_i[j,\ 1] * (t_{1i}[j] + t_{2i}[j]) + z_i[j,\ 2] * (bus_i + bus_{2i}[j] + bustime)$$

$$-t_{1i}[j] \le loc[1] - lat_i[j] \le t_{1i}[j]$$

$$-t_{2i}[j] \le loc[2] - lon_i[j] \le t_{2i}[j]$$

$$(t_{1i}[j] + t_{2i}[j]) - (bus_i + bus_{2i}[j] + bustime) \ge m_1 * z_i[j,\ 1]$$

$$(t_{1i}[j] + t_{2i}[j]) - (bus_i + bus_{2i}[j] + bustime) \ge m_1 * (1 - z_i[j,\ 2])$$

$$(t_{1i}[j] + t_{2i}[j]) - (bus_i + bus_{2i}[j] + bustime) \le M_1 * z_i[j,\ 2] - eps * (1 - z_i$$

$$(t_{1i}[j] + t_{2i}[j]) - (bus_i + bus_{2i}[j] + bustime) \le M_1 * (1 - z_i[j,\ 1]) - eps * z$$

$$bus_i = u_i[1] * (loc[2] - busx_2) + u_i[2] * (busx_1 - loc[2]) + u_i[3] * (loc[1] - busy_2) + u_i[4] * (busy_1 - loc[1])$$

$$loc[2] - busx_1 \ge (rectx_1 - busx_1) * u_i[2]$$

$$loc[2] - busx_2 \le (rectx_2 - busx_2) * u_i[1]$$

$$loc[2] - busx_2 \ge (rectx_1 - busx_2) * (1 - u_i[1]) + eps * u_i[1]$$

$$loc[2] - busx_1 \le (rectx_2 - busx_1) * (1 - u_i[2]) - eps * u_i[2]$$

$$loc[1] - busy_1 \ge (recty_1 - busy_1) * u_i[4]$$

$$loc[1] - busy_2 \le (recty_2 - busy_2) * u_i[3]$$

$$loc[1] - busy_2 \ge (recty_1 - busy_2) * (1 - u_i[3]) + eps * u_i[3]$$

$$loc[1] - busy_1 \le (recty_2 - busy_1) * (1 - u_i[4]) - eps * u_i[4]$$

$$\sum_{j\ in\ 1:n_i} b_i[j] = num_i$$

$$b_i[j],\ u_i[l],\ z_i[j,k] \in [0,\ 1]$$

$$\text{for i in labels, j in 1:ni, k in 1:2, l in 1:4}$$

Dicision variables:

- $location = [lat,\ \ lon]$, the optimal location for new apartment.

For buildings of each {label} from $labels\ = [hos,\ rest,\ mkt,\ camp,\ univ,\ lib]$:

- $b = [b_i]$, for i in [1, amount of buildings with {label}], b_i is binary, to decide how many nearest buildings to take into consideration.
- $dist = [d_i]$, for i in [1, amount of buildings with {label}], d_i is the manhatten distance from the optimal location to each bulding with {label}.
- $t_1[t_{1i}]$ and $t_2[t_{2i}]$, for i in [1, amount of buildings with {label}], to get the abstract value of latitude and longitude distance.
- $z \in \mathbb{R}^{\mathrm{ni\ x\ 2}}$, $u\ \in \mathbb{R}^4$ are vectors of binary, used for logical constraints. The detail explanation are in comments of the code.

Constraints:

- Compared to previous model, plenty of logical constraints are added, to compute the minimum manhattan distance from origin to bus rectangle, sum all distance and bus time as cost and compare that with the manhattan distance from origin to destination, use the logical constraints to choose the smaller one and store it as final distance to $dist$.

Objective function:

- The average distance to each building of each type is used to express the level of convenience, and they are weighted by given $weight_i$.

Model type:

- It is a MISOCP. Binary variables is used for nearest buildings and logical constraints. It is a SOCP because of the logical constraints.

4. Solution

In [1]:
```
using CSV, DataFrames, NamedArrays
using PyPlot, JuMP, HiGHS, Ipopt, Gurobi, Juniper
using PyCall
@pyimport folium
```

Organize the fetched data into csv and read them as array, storing their parameters.

In [2]:
```
# convert csv to name and info array

function csv_to_name_info(filename)
    # read and convert into a named array
    raw = CSV.read(filename, DataFrame);
    # list of name
    name = raw[1:end,1]
```

```
    # list of info
    info = Array(raw[1:end, 2:end])

    return name, info
end

# read data to arrays

bus_name, bus_info = csv_to_name_info("bus_stops.csv");
hos_name, hos_info = csv_to_name_info("hospitals.csv");
rest_name, rest_info = csv_to_name_info("restaurants.csv");
apt_name, apt_info = csv_to_name_info("apartments.csv");
camp_name, camp_info = csv_to_name_info("university.csv");
mkt_name, mkt_info = csv_to_name_info("supermarkets.csv");
lib_name, lib_info = csv_to_name_info("library_recreation.csv");
univ_name, univ_info = csv_to_name_info("academic_buildings.csv");

bus_size = size(bus_name)[1]
hos_size = size(hos_name)[1]
rest_size = size(rest_name)[1]
apt_size = size(apt_name)[1]
camp_size = size(camp_name)[1]
mkt_size = size(mkt_name)[1]
lib_size = size(lib_name)[1]
univ_size = size(univ_name)[1]

bus_num = 1:size(bus_name)[1]
hos_num = 1:size(hos_name)[1]
rest_num = 1:size(rest_name)[1]
apt_num = 1:size(apt_name)[1]
camp_num = 1:size(camp_name)[1]
mkt_num = 1:size(mkt_name)[1]
lib_num = 1:size(lib_name)[1]
univ_num = 1:size(univ_name)[1]


# location of madison capital
center = [43.074761 -89.3837613];

# total range
rect = [43.061969, -89.457811, 43.082614, -89.381480];
rect_xrange = [-89.457811, -89.381480];
rect_yrange = [43.061969, 43.082614];

# bus range
bus_xrange = [-89.428955, -89.400552];
bus_yrange = [43.072107, 43.076629];
```

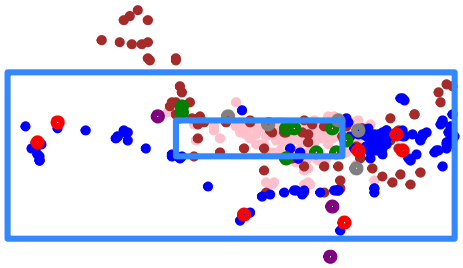
Visualize the data.

In [3]:

```
# Mark the locations of buildings with different colors,
# and the rectangle scope to consider
function draw_map(my_map)

    polygon_rec = folium.Polygon(
        locations=[[43.082614, -89.457811], [43.082614, -89.381480], [43.061969,
    )

    polygon_bus = folium.Polygon(
```

```
        locations=[[43.076629, -89.428955], [43.076629, -89.400552], [43.072107,
    )

    for i in camp_num
        folium.CircleMarker(location=[camp_info[i, 1], camp_info[i, 2]],radius=0
    end
    for i in bus_num
#         if bus_info[i, 3] == 1 || bus_info[i, 5] == 1
          folium.CircleMarker(location=[bus_info[i, 1], bus_info[i, 2]],radius=
#         end
    end
    for i in rest_num
        folium.CircleMarker(location=[rest_info[i, 1], rest_info[i, 2]],radius=1
    end
    for i in hos_num
        folium.CircleMarker(location=[hos_info[i, 1], hos_info[i, 2]],radius=2,c
    end
    for i in mkt_num
        folium.CircleMarker(location=[mkt_info[i, 1], mkt_info[i, 2]],radius=2,c
    end
    for i in univ_num
        folium.CircleMarker(location=[univ_info[i, 1], univ_info[i, 2]],radius=2
    end
    for i in lib_num
        folium.CircleMarker(location=[lib_info[i, 1], lib_info[i, 2]],radius=2,c
    end

    polygon_bus.add_to(my_map)
    polygon_rec.add_to(my_map)

    return my_map
end

my_map = folium.Map(location=center, zoom_start=12)
draw_map(my_map)
```

Out[3]: Make this Notebook Trusted to load map: File -> Trust Notebook



Leaflet (https://leafletjs.com) | Data by © OpenStreetMap (http://openstreetmap.org), under ODbL (http://www.openstreetmap.org/copyright).

Code of model without metro transit.

In [4]:

```
# function to measure the convinience to all type of buildings
# solving a QP with Gurobi
# lambda is the weight for each convinience to consider
# return the optimal location

function manhatten_conv(lambda)
    m = Model(Gurobi.Optimizer)
    set_silent(m)

    # variable to find an optimal location for new apt
    @variable(m, location[1:2])
    # set start value near captial
    set_start_value.(location, center)

    # distance to hospital
    @variable(m, hos_dist[hos_num])
    # binary variable to get the distance to nearest n hospitals
    @variable(m, hos_b[hos_num], Bin)
    @variable(m, hos_t1[hos_num] >= 0)
    @variable(m, hos_t2[hos_num] >= 0)

    # get the manhattan distance, stored in hos_t1 for delta latitude and hos_t2
    for i in hos_num
        @constraint(m, location[1] - hos_info[i, 1] .<= hos_t1[i])
        @constraint(m, -hos_t1[i] .<= location[1] - hos_info[i, 1])
        @constraint(m, location[2] - hos_info[i, 2] .<= hos_t2[i])
        @constraint(m, -hos_t2[i] .<= location[2] - hos_info[i, 2])
    end

    # get the manhattan distance to every hosptial, stored in hos_dist
    @constraint(m, hos_dist .== hos_t1 .+ hos_t2)
    # use the binary to get the nearest 1 hospital
    @constraint(m, sum(hos_b[i] for i in hos_num) == 1)
    @expression(m, hos_score, sum(hos_dist[i] * hos_b[i] for i in hos_num))


     # distance to restaurant
    @variable(m, rest_dist[rest_num])
    @variable(m, rest_t1[rest_num] >= 0)
    @variable(m, rest_t2[rest_num] >= 0)

    for i in rest_num
        @constraint(m, location[1] - rest_info[i, 1] .<= rest_t1[i])
        @constraint(m, -rest_t1[i] .<= location[1] - rest_info[i, 1])
        @constraint(m, location[2] - rest_info[i, 2] .<= rest_t2[i])
        @constraint(m, -rest_t2[i] .<= location[2] - rest_info[i, 2])
    end

    @constraint(m, rest_dist .== rest_t1 .+ rest_t2)
    @expression(m, rest_score, sum(rest_dist[i] for i in rest_num)/rest_size)



    # distance to supermarket
    @variable(m, mkt_dist[mkt_num])
    @variable(m, mkt_b[mkt_num], Bin)
    @variable(m, mkt_t1[mkt_num] >= 0)
    @variable(m, mkt_t2[mkt_num] >= 0)
```

```
for i in mkt_num
    @constraint(m, location[1] - mkt_info[i, 1] .<= mkt_t1[i])
    @constraint(m, -mkt_t1[i] .<= location[1] - mkt_info[i, 1])
    @constraint(m, location[2] - mkt_info[i, 2] .<= mkt_t2[i])
    @constraint(m, -mkt_t2[i] .<= location[2] - mkt_info[i, 2])
end

@constraint(m, mkt_dist .== mkt_t1 .+ mkt_t2)
@constraint(m, sum(mkt_b[i] for i in mkt_num) == 1)
@expression(m, mkt_score, sum(mkt_dist[i] * mkt_b[i] for i in mkt_num))


# distance to campus
@variable(m, camp_dist[camp_num])
@variable(m, camp_t1[camp_num] >= 0)
@variable(m, camp_t2[camp_num] >= 0)

for i in camp_num
    @constraint(m, location[1] - camp_info[i, 1] .<= camp_t1[i])
    @constraint(m, -camp_t1[i] .<= location[1] - camp_info[i, 1])
    @constraint(m, location[2] - camp_info[i, 2] .<= camp_t2[i])
    @constraint(m, -camp_t2[i] .<= location[2] - camp_info[i, 2])
end

@constraint(m, camp_dist .== camp_t1 .+ camp_t2)
@expression(m, camp_score, sum(camp_dist[i] for i in camp_num)/camp_size)


# distance to university main buildings
@variable(m, univ_dist[univ_num])
@variable(m, univ_t1[univ_num] >= 0)
@variable(m, univ_t2[univ_num] >= 0)
@variable(m, univ_b[univ_num], Bin)

for i in univ_num
    @constraint(m, location[1] - univ_info[i, 1] .<= univ_t1[i])
    @constraint(m, -univ_t1[i] .<= location[1] - univ_info[i, 1])
    @constraint(m, location[2] - univ_info[i, 2] .<= univ_t2[i])
    @constraint(m, -univ_t2[i] .<= location[2] - univ_info[i, 2])
end

@constraint(m, univ_dist .== univ_t1 .+ univ_t2)
@constraint(m, sum(univ_b[i] for i in univ_num) == 3)
@expression(m, univ_score, sum(univ_dist[i] * univ_b[i] for i in univ_num)/u


# distance to main libraries and recreation buildings
@variable(m, lib_dist[lib_num])
@variable(m, lib_t1[lib_num] >= 0)
@variable(m, lib_t2[lib_num] >= 0)
@variable(m, lib_b[lib_num], Bin)

for i in lib_num
    @constraint(m, location[1] - lib_info[i, 1] .<= lib_t1[i])
    @constraint(m, -lib_t1[i] .<= location[1] - lib_info[i, 1])
    @constraint(m, location[2] - lib_info[i, 2] .<= lib_t2[i])
    @constraint(m, -lib_t2[i] .<= location[2] - lib_info[i, 2])
end

@constraint(m, lib_dist .== lib_t1 .+ lib_t2)
```

```
    @constraint(m, sum(lib_b[i] for i in lib_num) == 2)
    @expression(m, lib_score, sum(lib_dist[i] * lib_b[i] for i in lib_num)/lib_s



    @objective(m, Min, hos_score * lambda[1] + rest_score * lambda[2] + mkt_scor

    optimize!(m)
    println(value.(location))

    return value.(location)
end
```

Out[4]: manhatten_conv (generic function with 1 method)

Code of model with metro transit.

In [5]:
```
# helper function to get the distance from the bus rectangle to
# destination.

function get_bus2(info, size)
    dist = zeros(size)
    for i in 1:size
        # y >= y2, dist += y-y2
        if info[i, 1] >= bus_yrange[2]
            dist[i] = dist[i] + info[i, 1] - bus_yrange[2]
        # y <= y1, dist += y1-y
        elseif info[i, 1] <= bus_yrange[1]
            dist[i] = dist[i] + bus_yrange[1] - info[i, 1]
        end

        # x >= x2, dist += x-x2
        if info[i, 2] >= bus_xrange[2]
            dist[i] = dist[i] + info[i, 2] - bus_xrange[2]
        # x <= x1, dist += x1-x
        elseif info[i, 2] <= bus_xrange[1]
            dist[i] = dist[i] + bus_xrange[1] - info[i, 2]
        end

    end

    return dist
end

# call the function to get the second distance
mkt_bus2 = get_bus2(mkt_info, mkt_size);
univ_bus2 = get_bus2(univ_info, univ_size);
rest_bus2 = get_bus2(rest_info, rest_size);
lib_bus2 = get_bus2(lib_info, lib_size);
hos_bus2 = get_bus2(hos_info, hos_size);
camp_bus2 = get_bus2(camp_info, camp_size);

# the upper bound and lower bound for logical constraint in the model
min_walk_bus = 2 * (bus_yrange[1] - rect_yrange[1] + bus_xrange[1] - rect_xrange
max_walk_bus = 2 * (bus_yrange[2] - bus_yrange[1] + bus_xrange[2] - bus_xrange[1
```

In [6]:
```
# function to measure the convinience to all type of buildings
# solving a SOCP with Gurobi
# lambda is the weight for each convinience to consider
```

```
# bustime is the fixed duration to take bus
# return the optimal location

function manhatten_conv_with_bus(lambda, bus_time)
    epsilon = 1e-10

    m = Model(Gurobi.Optimizer)
    set_silent(m)

    # variable to find an optimal location for new apt
    @variable(m, location[1:2])
    # set start value near captial
    set_start_value.(location, center)

    # distance to supermarket
    @variable(m, mkt_dist[mkt_num])
    # binary variable to get the distance to nearest n hospitals
    @variable(m, mkt_b[mkt_num], Bin)
    @variable(m, mkt_t1[mkt_num] >= 0)
    @variable(m, mkt_t2[mkt_num] >= 0)

    @variable(m, mkt_bus1)
    # binary variables to set the logical constraints
    @variable(m, mkt_z1[1:4], Bin)
    @variable(m, mkt_z[mkt_num, 1:2], Bin)

    # the distance from the origin to the bus rectangle
    @constraint(m, mkt_bus1==mkt_z1[1]*(location[2]-bus_xrange[2])+mkt_z1[2]*(bu

    # culculate the manhattan distance to the rectangle
    # x <= x1, z12 = 1
    @constraint(m, location[2] - bus_xrange[1] >= (rect_xrange[1] - bus_xrange[1
    # x >= x2, z11 = 1
    @constraint(m, location[2] - bus_xrange[2] <= (rect_xrange[2] - bus_xrange[2
    # x <= x2, z11 = 0
    @constraint(m, location[2] - bus_xrange[2] >= (rect_xrange[1] - bus_xrange[2
    # x >= x1, z12 = 0
    @constraint(m, location[2] - bus_xrange[1] <= (rect_xrange[2] - bus_xrange[1

    # y <= y1, z14 = 1
    @constraint(m, location[1] - bus_yrange[1] >= (rect_yrange[1] - bus_yrange[1
    # y >= y2, z13 = 1
    @constraint(m, location[1] - bus_yrange[2] <= (rect_yrange[2] - bus_yrange[2
    # y <= y2, z13 = 0
    @constraint(m, location[1] - bus_yrange[2] >= (rect_yrange[1] - bus_yrange[2
    # y >= y1, z14 = 0
    @constraint(m, location[1] - bus_yrange[1] <= (rect_yrange[2] - bus_yrange[1

    # get the abstraction for walking distance
    for i in mkt_num
        @constraint(m, location[1] - mkt_info[i, 1] .<= mkt_t1[i])
        @constraint(m, -mkt_t1[i] .<= location[1] - mkt_info[i, 1])
        @constraint(m, location[2] - mkt_info[i, 2] .<= mkt_t2[i])
        @constraint(m, -mkt_t2[i] .<= location[2] - mkt_info[i, 2])
    end

    # compare the walking distance and bus distance, store the smaller one to di
    for i in mkt_num
        @constraint(m, mkt_dist[i] == (mkt_t1[i] + mkt_t2[i]) * mkt_z[i, 1] + (m
```

```
    # mkt_walk <= mkt_bus, z1 = 1
    @constraint(m, mkt_t1[i] + mkt_t2[i] - (mkt_bus1 + mkt_bus2[i] + bus_tim
    # mkt_walk <= mkt_bus, z2 = 0
    @constraint(m, mkt_t1[i] + mkt_t2[i] - (mkt_bus1 + mkt_bus2[i] + bus_tim
    # mkt_walk > mkt_bus, z2 = 1
    @constraint(m, mkt_t1[i] + mkt_t2[i] - (mkt_bus1 + mkt_bus2[i] + bus_tim
    # mkt_walk > mkt_bus, z1 = 0
    @constraint(m, mkt_t1[i] + mkt_t2[i] - (mkt_bus1 + mkt_bus2[i] + bus_tim

end

# select the nearest some
@constraint(m, sum(mkt_b[i] for i in mkt_num) == 2)
@expression(m, mkt_score, sum(mkt_dist[i] * mkt_b[i] for i in mkt_num))


# distance to university main buildings
@variable(m, univ_dist[univ_num])
@variable(m, univ_b[univ_num], Bin)
@variable(m, univ_t1[univ_num] >= 0)
@variable(m, univ_t2[univ_num] >= 0)

@variable(m, univ_bus1)
@variable(m, univ_z1[1:4], Bin)
@variable(m, univ_z[univ_num, 1:2], Bin)

@constraint(m, univ_bus1==univ_z1[1]*(location[2]-bus_xrange[2])+univ_z1[2]*

# x <= x1, z12 = 1
@constraint(m, location[2] - bus_xrange[1] >= (rect_xrange[1] - bus_xrange[1
# x >= x2, z11 = 1
@constraint(m, location[2] - bus_xrange[2] <= (rect_xrange[2] - bus_xrange[2
# x <= x2, z11 = 0
@constraint(m, location[2] - bus_xrange[2] >= (rect_xrange[1] - bus_xrange[2
# x >= x1, z12 = 0
@constraint(m, location[2] - bus_xrange[1] <= (rect_xrange[2] - bus_xrange[1

# y <= y1, z14 = 1
@constraint(m, location[1] - bus_yrange[1] >= (rect_yrange[1] - bus_yrange[1
# y >= y2, z13 = 1
@constraint(m, location[1] - bus_yrange[2] <= (rect_yrange[2] - bus_yrange[2
# y <= y2, z13 = 0
@constraint(m, location[1] - bus_yrange[2] >= (rect_yrange[1] - bus_yrange[2
# y >= y1, z14 = 0
@constraint(m, location[1] - bus_yrange[1] <= (rect_yrange[2] - bus_yrange[1


for i in univ_num
    @constraint(m, location[1] - univ_info[i, 1] .<= univ_t1[i])
    @constraint(m, -univ_t1[i] .<= location[1] - univ_info[i, 1])
    @constraint(m, location[2] - univ_info[i, 2] .<= univ_t2[i])
    @constraint(m, -univ_t2[i] .<= location[2] - univ_info[i, 2])
end

for i in univ_num
    @constraint(m, univ_dist[i] == (univ_t1[i] + univ_t2[i]) * univ_z[i, 1]

    # univ_walk <= univ_bus, z1 = 1
    @constraint(m, univ_t1[i] + univ_t2[i] - (univ_bus1 + univ_bus2[i] + bus
    # univ_walk <= univ_bus, z2 = 0
```

```
    @constraint(m, univ_t1[i] + univ_t2[i] - (univ_bus1 + univ_bus2[i] + bus
    # univ_walk > univ_bus, z2 = 1
    @constraint(m, univ_t1[i] + univ_t2[i] - (univ_bus1 + univ_bus2[i] + bus
    # univ_walk > univ_bus, z1 = 0
    @constraint(m, univ_t1[i] + univ_t2[i] - (univ_bus1 + univ_bus2[i] + bus

end

@constraint(m, sum(univ_b[i] for i in univ_num) == 3)
@expression(m, univ_score, sum(univ_dist[i] * univ_b[i] for i in univ_num))



# distance to main libraries and recreation buildings
@variable(m, lib_dist[lib_num])
@variable(m, lib_b[lib_num], Bin)
@variable(m, lib_t1[lib_num] >= 0)
@variable(m, lib_t2[lib_num] >= 0)

@variable(m, lib_bus1)
@variable(m, lib_z1[1:4], Bin)
@variable(m, lib_z[lib_num, 1:2], Bin)

@constraint(m, lib_bus1==lib_z1[1]*(location[2]-bus_xrange[2])+lib_z1[2]*(bu

# x <= x1, z12 = 1
@constraint(m, location[2] - bus_xrange[1] >= (rect_xrange[1] - bus_xrange[1
# x >= x2, z11 = 1
@constraint(m, location[2] - bus_xrange[2] <= (rect_xrange[2] - bus_xrange[2
# x <= x2, z11 = 0
@constraint(m, location[2] - bus_xrange[2] >= (rect_xrange[1] - bus_xrange[2
# x >= x1, z12 = 0
@constraint(m, location[2] - bus_xrange[1] <= (rect_xrange[2] - bus_xrange[1

# y <= y1, z14 = 1
@constraint(m, location[1] - bus_yrange[1] >= (rect_yrange[1] - bus_yrange[1
# y >= y2, z13 = 1
@constraint(m, location[1] - bus_yrange[2] <= (rect_yrange[2] - bus_yrange[2
# y <= y2, z13 = 0
@constraint(m, location[1] - bus_yrange[2] >= (rect_yrange[1] - bus_yrange[2
# y >= y1, z14 = 0
@constraint(m, location[1] - bus_yrange[1] <= (rect_yrange[2] - bus_yrange[1


for i in lib_num
    @constraint(m, location[1] - lib_info[i, 1] .<= lib_t1[i])
    @constraint(m, -lib_t1[i] .<= location[1] - lib_info[i, 1])
    @constraint(m, location[2] - lib_info[i, 2] .<= lib_t2[i])
    @constraint(m, -lib_t2[i] .<= location[2] - lib_info[i, 2])
end

for i in lib_num
    @constraint(m, lib_dist[i] == (lib_t1[i] + lib_t2[i]) * lib_z[i, 1] + (1

    # lib_walk <= lib_bus, z1 = 1
    @constraint(m, lib_t1[i] + lib_t2[i] - (lib_bus1 + lib_bus2[i] + bus_tim
    # lib_walk <= lib_bus, z2 = 0
    @constraint(m, lib_t1[i] + lib_t2[i] - (lib_bus1 + lib_bus2[i] + bus_tim
    # lib_walk > lib_bus, z2 = 1
    @constraint(m, lib_t1[i] + lib_t2[i] - (lib_bus1 + lib_bus2[i] + bus_tim
```

```
    # lib_walk > lib_bus, z1 = 0
    @constraint(m, lib_t1[i] + lib_t2[i] - (lib_bus1 + lib_bus2[i] + bus_tim

end

@constraint(m, sum(lib_b[i] for i in lib_num) == 3)
@expression(m, lib_score, sum(lib_dist[i] * lib_b[i] for i in lib_num))



# distance to restaurants
@variable(m, rest_dist[rest_num])
@variable(m, rest_b[rest_num], Bin)
@variable(m, rest_t1[rest_num] >= 0)
@variable(m, rest_t2[rest_num] >= 0)

@variable(m, rest_bus1)
@variable(m, rest_z1[1:4], Bin)
@variable(m, rest_z[rest_num, 1:2], Bin)

@constraint(m, rest_bus1==rest_z1[1]*(location[2]-bus_xrange[2])+rest_z1[2]*

# x <= x1, z12 = 1
@constraint(m, location[2] - bus_xrange[1] >= (rect_xrange[1] - bus_xrange[1
# x >= x2, z11 = 1
@constraint(m, location[2] - bus_xrange[2] <= (rect_xrange[2] - bus_xrange[2
# x <= x2, z11 = 0
@constraint(m, location[2] - bus_xrange[2] >= (rect_xrange[1] - bus_xrange[2
# x >= x1, z12 = 0
@constraint(m, location[2] - bus_xrange[1] <= (rect_xrange[2] - bus_xrange[1

# y <= y1, z14 = 1
@constraint(m, location[1] - bus_yrange[1] >= (rect_yrange[1] - bus_yrange[1
# y >= y2, z13 = 1
@constraint(m, location[1] - bus_yrange[2] <= (rect_yrange[2] - bus_yrange[2
# y <= y2, z13 = 0
@constraint(m, location[1] - bus_yrange[2] >= (rect_yrange[1] - bus_yrange[2
# y >= y1, z14 = 0
@constraint(m, location[1] - bus_yrange[1] <= (rect_yrange[2] - bus_yrange[1


for i in rest_num
    @constraint(m, location[1] - rest_info[i, 1] .<= rest_t1[i])
    @constraint(m, -rest_t1[i] .<= location[1] - rest_info[i, 1])
    @constraint(m, location[2] - rest_info[i, 2] .<= rest_t2[i])
    @constraint(m, -rest_t2[i] .<= location[2] - rest_info[i, 2])
end

for i in rest_num
    @constraint(m, rest_dist[i] == (rest_t1[i] + rest_t2[i]) * rest_z[i, 1]

    # rest_walk <= rest_bus, z1 = 1
    @constraint(m, rest_t1[i] + rest_t2[i] - (rest_bus1 + rest_bus2[i] + bus
    # rest_walk <= rest_bus, z2 = 0
    @constraint(m, rest_t1[i] + rest_t2[i] - (rest_bus1 + rest_bus2[i] + bus
    # rest_walk > rest_bus, z2 = 1
    @constraint(m, rest_t1[i] + rest_t2[i] - (rest_bus1 + rest_bus2[i] + bus
    # rest_walk > rest_bus, z1 = 0
    @constraint(m, rest_t1[i] + rest_t2[i] - (rest_bus1 + rest_bus2[i] + bus
```

```
end

@constraint(m, sum(rest_b[i] for i in rest_num) == 5)
@expression(m, rest_score, sum(rest_dist[i] * rest_b[i] for i in rest_num))



# distance to hospitals
@variable(m, hos_dist[hos_num])
@variable(m, hos_b[hos_num], Bin)
@variable(m, hos_t1[hos_num] >= 0)
@variable(m, hos_t2[hos_num] >= 0)

@variable(m, hos_bus1)
@variable(m, hos_z1[1:4], Bin)
@variable(m, hos_z[hos_num, 1:2], Bin)

@constraint(m, hos_bus1==hos_z1[1]*(location[2]-bus_xrange[2])+hos_z1[2]*(bu

# x <= x1, z12 = 1
@constraint(m, location[2] - bus_xrange[1] >= (rect_xrange[1] - bus_xrange[1
# x >= x2, z11 = 1
@constraint(m, location[2] - bus_xrange[2] <= (rect_xrange[2] - bus_xrange[2
# x <= x2, z11 = 0
@constraint(m, location[2] - bus_xrange[2] >= (rect_xrange[1] - bus_xrange[2
# x >= x1, z12 = 0
@constraint(m, location[2] - bus_xrange[1] <= (rect_xrange[2] - bus_xrange[1

# y <= y1, z14 = 1
@constraint(m, location[1] - bus_yrange[1] >= (rect_yrange[1] - bus_yrange[1
# y >= y2, z13 = 1
@constraint(m, location[1] - bus_yrange[2] <= (rect_yrange[2] - bus_yrange[2
# y <= y2, z13 = 0
@constraint(m, location[1] - bus_yrange[2] >= (rect_yrange[1] - bus_yrange[2
# y >= y1, z14 = 0
@constraint(m, location[1] - bus_yrange[1] <= (rect_yrange[2] - bus_yrange[1


for i in hos_num
    @constraint(m, location[1] - hos_info[i, 1] .<= hos_t1[i])
    @constraint(m, -hos_t1[i] .<= location[1] - hos_info[i, 1])
    @constraint(m, location[2] - hos_info[i, 2] .<= hos_t2[i])
    @constraint(m, -hos_t2[i] .<= location[2] - hos_info[i, 2])
end

for i in hos_num
    @constraint(m, hos_dist[i] == (hos_t1[i] + hos_t2[i]) * hos_z[i, 1] + (h

    # hos_walk <= hos_bus, z1 = 1
    @constraint(m, hos_t1[i] + hos_t2[i] - (hos_bus1 + hos_bus2[i] + bus_tim
    # hos_walk <= hos_bus, z2 = 0
    @constraint(m, hos_t1[i] + hos_t2[i] - (hos_bus1 + hos_bus2[i] + bus_tim
    # hos_walk > hos_bus, z2 = 1
    @constraint(m, hos_t1[i] + hos_t2[i] - (hos_bus1 + hos_bus2[i] + bus_tim
    # hos_walk > hos_bus, z1 = 0
    @constraint(m, hos_t1[i] + hos_t2[i] - (hos_bus1 + hos_bus2[i] + bus_tim

end

@constraint(m, sum(hos_b[i] for i in hos_num) == 1)
```

```
    @expression(m, hos_score, sum(hos_dist[i] * hos_b[i] for i in hos_num))



    # distance to campus
    @variable(m, camp_dist[camp_num])
#     @variable(m, camp_b[camp_num], Bin)
    @variable(m, camp_t1[camp_num] >= 0)
    @variable(m, camp_t2[camp_num] >= 0)

    @variable(m, camp_bus1)
    @variable(m, camp_z1[1:4], Bin)
    @variable(m, camp_z[camp_num, 1:2], Bin)

    @constraint(m, camp_bus1==camp_z1[1]*(location[2]-bus_xrange[2])+camp_z1[2]*

    # x <= x1, z12 = 1
    @constraint(m, location[2] - bus_xrange[1] >= (rect_xrange[1] - bus_xrange[1
    # x >= x2, z11 = 1
    @constraint(m, location[2] - bus_xrange[2] <= (rect_xrange[2] - bus_xrange[2
    # x <= x2, z11 = 0
    @constraint(m, location[2] - bus_xrange[2] >= (rect_xrange[1] - bus_xrange[2
    # x >= x1, z12 = 0
    @constraint(m, location[2] - bus_xrange[1] <= (rect_xrange[2] - bus_xrange[1

    # y <= y1, z14 = 1
    @constraint(m, location[1] - bus_yrange[1] >= (rect_yrange[1] - bus_yrange[1
    # y >= y2, z13 = 1
    @constraint(m, location[1] - bus_yrange[2] <= (rect_yrange[2] - bus_yrange[2
    # y <= y2, z13 = 0
    @constraint(m, location[1] - bus_yrange[2] >= (rect_yrange[1] - bus_yrange[2
    # y >= y1, z14 = 0
    @constraint(m, location[1] - bus_yrange[1] <= (rect_yrange[2] - bus_yrange[1


    for i in camp_num
        @constraint(m, location[1] - camp_info[i, 1] .<= camp_t1[i])
        @constraint(m, -camp_t1[i] .<= location[1] - camp_info[i, 1])
        @constraint(m, location[2] - camp_info[i, 2] .<= camp_t2[i])
        @constraint(m, -camp_t2[i] .<= location[2] - camp_info[i, 2])
    end

    for i in camp_num
        @constraint(m, camp_dist[i] == (camp_t1[i] + camp_t2[i]) * camp_z[i, 1]

        # camp_walk <= camp_bus, z1 = 1
        @constraint(m, camp_t1[i] + camp_t2[i] - (camp_bus1 + camp_bus2[i] + bus
        # camp_walk <= camp_bus, z2 = 0
        @constraint(m, camp_t1[i] + camp_t2[i] - (camp_bus1 + camp_bus2[i] + bus
        # camp_walk > camp_bus, z2 = 1
        @constraint(m, camp_t1[i] + camp_t2[i] - (camp_bus1 + camp_bus2[i] + bus
        # camp_walk > camp_bus, z1 = 0
        @constraint(m, camp_t1[i] + camp_t2[i] - (camp_bus1 + camp_bus2[i] + bus

    end

#     @constraint(m, sum(camp_b[i] for i in camp_num) == 1)
    @expression(m, camp_score, sum(camp_dist[i] for i in camp_num))
```

```
    @objective(m, Min, hos_score * lambda[1] + rest_score * lambda[2] + mkt_scor

#     @objective(m, Min, mkt_score)

    optimize!(m)
    println(value.(location))
#     println("Minimum of walking and bus distance: ", value.(mkt_dist))
#     println("The score convinience: ", value.(mkt_score))

    return value.(location)
end
```

Out[6]: manhatten_conv_with_bus (generic function with 1 method)

Call the 2 model functions above with different weight parameters. The optimal locations with different weights are printed.

We considered the following possible needs for the apartment location:

- Weight [1, 1, 1, 1, 1, 1] means no preference.
- Weight [1, 5, 5, 0, 1, 1] means not to consider the proximity of the campus, really need to be near to resaturant and supermarket, and need to be near hospitals, main university buildings and libraries.
- Weight [3, 1, 5, 0, 0, 3] means not to consider the proximity of the campus and main university buildings, really need to be near to supermarket, and need to be near hospitals and libraries.
- Weight [1, 1, 1, 3, 5, 5] means really need to be near to university main buildings and libraries, and also need to be near campus.
- Weight [1, 5, 0, 0, 1, 0] means really need to be near to restaurants, and only need to be near hospitals and university main buildings.

In [7]:
```
# The number means the weights on [hos, rest, mkt, camp, univ, lib],
# considering different needs of different people.

location_val0 = manhatten_conv([1, 1, 1, 1, 1, 1]);
location_val1 = manhatten_conv([1, 5, 5, 0, 1, 1]);
location_val2 = manhatten_conv([3, 1, 5, 0, 0, 3]);
location_val3 = manhatten_conv([1, 1, 1, 3, 5, 5]);
location_val4 = manhatten_conv([1, 5, 0, 0, 1, 0]);
```

```
Set parameter Username
Academic license - for non-commercial use only - expires 2024-02-24
[43.072995, -89.401413]
Set parameter Username
Academic license - for non-commercial use only - expires 2024-02-24
[43.073002, -89.39766]
Set parameter Username
Academic license - for non-commercial use only - expires 2024-02-24
[43.065791, -89.400245]
Set parameter Username
Academic license - for non-commercial use only - expires 2024-02-24
[43.074396, -89.40156]
Set parameter Username
Academic license - for non-commercial use only - expires 2024-02-24
[43.073449, -89.397009]
```

In [8]:
```
# And the same weights with metro bus, but the nearest number of buildings
# are different. There are so may buildings with label like restaurants,
# which will always leads to same location even with weights.

location_val_with_bus0 = manhatten_conv_with_bus([1, 1, 1, 1, 1, 1], 1e-2);
location_val_with_bus1 = manhatten_conv_with_bus([1, 5, 5, 0, 1, 1], 1e-2);
location_val_with_bus2 = manhatten_conv_with_bus([3, 1, 5, 0, 0, 3], 1e-2);
location_val_with_bus3 = manhatten_conv_with_bus([1, 1, 1, 3, 5, 5], 1e-2);
location_val_with_bus4 = manhatten_conv_with_bus([1, 5, 0, 0, 1, 0], 1e-2);
```

```
Set parameter Username
Academic license - for non-commercial use only - expires 2024-02-24
[43.076629, -89.414195]
Set parameter Username
Academic license - for non-commercial use only - expires 2024-02-24
[43.074513, -89.4289550001]
Set parameter Username
Academic license - for non-commercial use only - expires 2024-02-24
[43.076629, -89.400552]
Set parameter Username
Academic license - for non-commercial use only - expires 2024-02-24
[43.076629, -89.414195]
Set parameter Username
Academic license - for non-commercial use only - expires 2024-02-24
[43.072107, -89.405395]
```

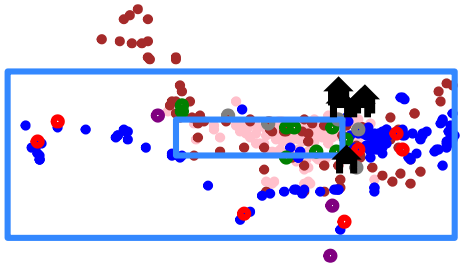
5. Results and discussion

In the model without Metro Transit, from the generated map, there are 5 ideal apartment locations: (1) near Memorial Library, (2) at the corner of West Dayton Street and North Broom Street, (3) at the university square and on the side of North Lake Street, (4) near Grainger Hall and Law Building, (5) and near Monona Bay and South Park Street.

In [9]:
```
my_map1 = folium.Map(location=center, zoom_start=12)
my_map1 = draw_map(my_map1)

folium.Marker([location_val0[1], location_val0[2]],
              icon=folium.Icon(icon="home",color="blue"),).add_to(my_map1)
folium.Marker([location_val1[1], location_val1[2]],
              icon=folium.Icon(icon="home",color="green"),).add_to(my_map1)
folium.Marker([location_val2[1], location_val2[2]],
```

```
            icon=folium.Icon(icon="home",color="red"),).add_to(my_map1)
folium.Marker([location_val3[1], location_val3[2]],
            icon=folium.Icon(icon="home",color="orange"),).add_to(my_map1)
folium.Marker([location_val4[1], location_val4[2]],
            icon=folium.Icon(icon="home",color="purple"),).add_to(my_map1)

# show the map
my_map1
```

Out[9]: Make this Notebook Trusted to load map: File -> Trust Notebook

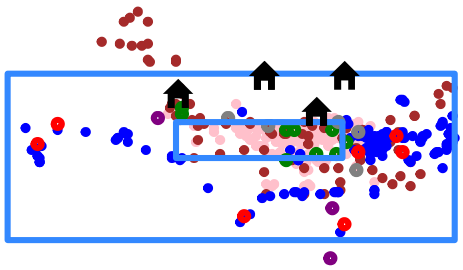


Leaflet (https://leafletjs.com) | Data by © OpenStreetMap (http://openstreetmap.org), under ODbL (http://www.openstreetmap.org/copyright).

In the model with Metro Transit, the generated map shows 5 new ideal apartment locations, they are: (1) next to Allen Centennial Garden and the Cole Sand Volleyball Courts, (2) on the side of Highland Avenue and next to Highland & VA Hospital (SB), (3) across form the Chemistry Building, (4) the place between College Library and Shannon Hall.

In [10]:
```
my_map2 = folium.Map(location=center, zoom_start=12)
draw_map(my_map2)

folium.Marker([location_val_with_bus0[1], location_val_with_bus0[2]],
            icon=folium.Icon(icon="home",color="blue"),).add_to(my_map2)
folium.Marker([location_val_with_bus1[1], location_val_with_bus1[2]],
            icon=folium.Icon(icon="home",color="green"),).add_to(my_map2)
folium.Marker([location_val_with_bus2[1], location_val_with_bus2[2]],
            icon=folium.Icon(icon="home",color="red"),).add_to(my_map2)
folium.Marker([location_val_with_bus3[1], location_val_with_bus3[2]],
            icon=folium.Icon(icon="home",color="orange"),).add_to(my_map2)
folium.Marker([location_val_with_bus4[1], location_val_with_bus4[2]],
            icon=folium.Icon(icon="home",color="purple"),).add_to(my_map2)

# show the map
my_map2
```

Out[10]: Make this Notebook Trusted to load map: File -> Trust Notebook



Leaflet (https://leafletjs.com) | Data by © OpenStreetMap (http://openstreetmap.org), under ODbL (http://www.openstreetmap.org/copyright).

6. Conclusion

When bus routes are not considered, most of the optimal apartments are located on State Street because of its proximity to restaurants and campus. When bus routes are considered, the apartments are located near bus routes and their locations vary because of different weights for needs.

We conclude that our model is reasonable as the location of the ideal apartments we got is consistent with the existing popular apartments on campus, such as Domain (at the corner of West Dayton Street and North Broom Street) and Lucky Apartment (at the university square and on the side of North Lake Street), according to the model without metro transit.

However, figuring out the ideal location in order to make residents feel convenient is not enough. The property company of the apartments still needs to make appropriate amenities plans to maximize the pricing of their units. A possible future direction may be to find the top amenities that residents most care about, such as whether the unit is furnished, has gyms, an in-unit washer/dryer, and pets, etc. We could set these amenities as binary variables and create a regression model to determine the most profitable amenity plan for the new apartment.

In [ ]: