MetByChance: A Travel Plan Match System

CS411 Project 1
Group37 UNKNOWN--MetByChance
Yan Xu, Saxue Wang, Sheng Hu, Yi Xu

1. Description

Our team built a website, MetByChance, to help people find travel companions when they have a travel plan. On our website, the user could post their travel plans and get them matched with the plans post by other people. When they post plan, the users can specify destinations, expected budgets and date ranges they want to travel in. We allow users to post multiple plans and get the multiple plans matched at the same time. Plus, for users without exact destinations in mind, we provide a complete set of data analysis tools to find popular destinations and travel times. The user can reject the match provided and may still get a new match if another qualifying plan is available. The system is robust when being accessed by multiple interacting users. The website provides a brand new platform for people to have extra fun in their trips by making new friends. We hope our effort can bring people to wonderful places worldwide and get people more connected in the world.

2. Usefulness

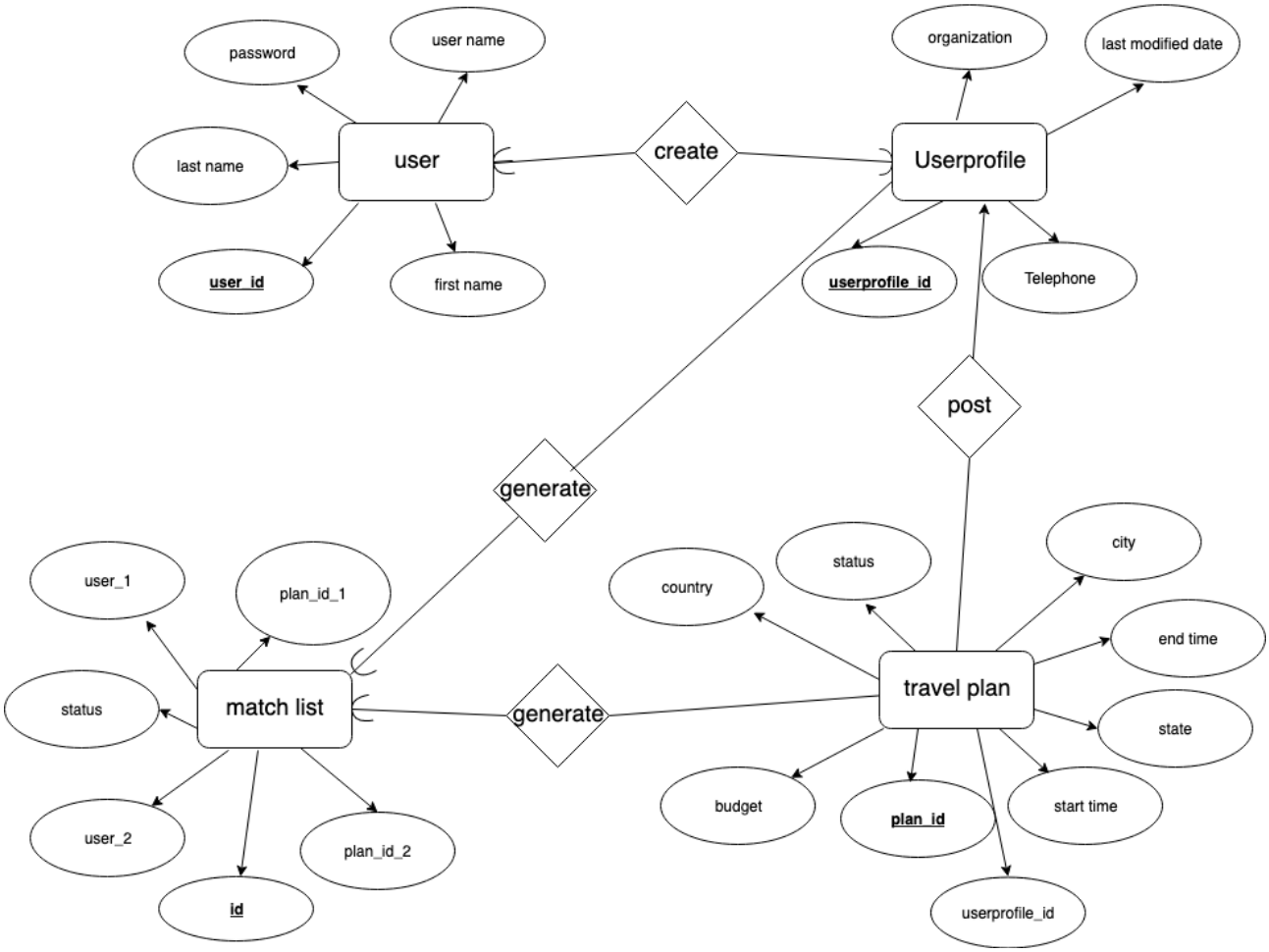
The desire of people to explore the world is increasing these years. However, people are limited to stay in their own communities. Besides that, communication usually happens among people in the same cities but this has prevented people from making more connections and from exploring the world.

Traveling together is a good choice to make new friends or reunite old friends. While traveling, people are easy to relax, and there are lots of new things to do together, which makes it easier to know each other and to build the friendship. Our goal is to provide as many matched options as possible to our users and without leaking too much private information.

3. Data

The data of our website is user-generated. We wrote a python file to generate needed data and import all of them into the SQLite database. For some testing issues, we also manually insert some data on the website to make sure that the matched plans in the table travel_Plan will go to the table match_list.

4 ER Diagram and Schema



users(user_id, password, user name, email, first name, last name)
userprofile(userprofile_id,telephone, last modified date, organization)
travelPlan(plan_id, country, city,status, budget, start_time, end_time,userprofile.userprofile_id)
matchList(id,travelplan.plan_id_1,travleplan.plan_id_2,userprofile.userprofile_id_1,userprofile.user profile_id_2,status)

5 Database Setup

The website www.priceoftravel.com provides the travel prices for many hot destinations in the world. We extracted budget at about 56 major European cities from the website. We assumed reasonable statistical distributions and generated users' profiles and travel plans.

Assumptions include:
- People post their travel plan at least 30 days before they travel;
- From fifteen days before the travel date, the website won't do any match on the specific travel plan;
- We assume a gamma distribution for the time intervals of post plans between the travel dates and current date;
- We assume the expected travel duration to be exponentially distributed with a mean of 2 days;
- We assume that travels have uniform distribution of interests to all cities;

- The budget of post plans are simulated of a log-normal distribution;

```python
# For the realness of the project, we get budget data for top European destinations from https://www.priceoftravel.com/1979/european-backpacker-index/

import pandas as pd
import os
import numpy as np
from datetime import datetime
from datetime import timedelta
os.chdir(r"C:\Users\sheng\Box Sync\phd-4\CS 411\MainProject\WebsiteProject\cs411\auto_generator")

# Read in the csv file created by user_generated.py

user_tab = pd.read_csv("user0.csv")
dest_tab = pd.read_excel("European_Backpackers_Index_2018.xlsx")

# Simulate the other
num_sim = 2 * user_tab.shape[0]

# Sample from the dest_tab for each pseoudo user

country = dest_tab["Country"].sample(n=num_sim, replace=True)
country = country.reset_index()
city = list()
budget = list()
start_time = list()
end_time = list()
state=list()
today = datetime.today().date()

# We ask people to post travel plans at least 30 days before they want to start
# If the travel plan gets less than 15 days away from the today
# the travel plan will be cancelled and we don't try to do any match for it any more
# In the simulation, we assume a gamma distribution for the time interval between the start date and today

# (k-1)*theta =15
# sqrt(k* (theta^2) ) = 5
k_gamma = np.sqrt(-1 + pow(5.5,2)) + 5.5
theta_gamma = 15 / (k_gamma - 1)
sd_samp = 15 + np.random.gamma(k_gamma, theta_gamma, num_sim)


# We assume duration to be exponentially distributed with mean 2 days
lambda_exp = 1/2
duration_samp = np.random.exponential(1/lambda_exp, num_sim)


user_id = user_tab["user_id"].sample(n=num_sim, replace=True)
```

```python
for i in range(num_sim):
    # Simulate cities according to the country
    fil_tab = dest_tab.loc[dest_tab["Country"].values == country['Country'][i]]
    samp = fil_tab.sample(n=1, replace=True)
    city.append(samp.values[0][2])
    # Simulate the budget posted by people
    # Here for each country-city, we simulate a random number using a log-normal distribution
    # with \miu = log("Total USD"), \sigma = log("Total USD")/10
    mu, sigma = np.log(samp['Total USD']), np.log(samp['Total USD'])/10
    budget.append(int(np.random.lognormal(mu, sigma, 1)[0]))
    start_time.append(today + timedelta(days=sd_samp[i]))
    end_time.append(start_time[i] + timedelta(days=sd_samp[i]))
    state.append('unknown')


plan_temp = {'user_id': user_id, 'country': list(country['Country'].values), 'city': city, 'budget': budget, 'start_time': start_time, 'end_time': end_time, 'state':state}

plan_tab = pd.DataFrame(plan_temp, columns=['user_id', 'country', 'city', 'budget', 'start_time', 'end_time','state']).reset_index().drop(['index'], axis=1)

plan_tab.head()

plan_tab.to_csv("plan.csv",index=False)
```

6 Clearly list the functionality

Insert: create user account, create user profile, post travel plan, match travel plan,
Delete: delete travel plan, reject travel plan
Update: update account password, update userprofile, update travel plan
Search: search hottest cities by budget
Advanced function 1: user can match with another travel plan by clicking post, delete, update and reject buttons.
Advanced function 2: data visualization, we create heatmap to see the travel plans' distribution in the map, another chart can help us to see the number of travel plan in each city.

7 One Basic Function

Insert: user can create a user profile, input his/her first name, last name, telephone and organization, the html will save this form to the back-end, then Django could save this form as a tuple in our database.

8 Actual SQL Code

```
def hotbybudget(request):
    if request.method == 'POST':
        budget_low_limit = request.POST.get('budget_low_limit')
        budget_high_limit = request.POST.get('budget_high_limit')
        # '''SELECT COUNT(plan_id), DATE_TRUNC('month',start_time) FROM users_travel_Plan GROUP BY DATE_TRUNC('month',start_time)'''
        # whatishot_result = travel_Plan.objects.raw('''SELECT plan_id, start_time FROM users_travel_Plan GROUP''')
        cursor = connection.cursor()
        cursor.execute("""select
                            city, country, m, sum(c)/2 as s
                        from
                        (select
                            city, country, strftime('%%m', start_time) as m, count(*) as c, "start" as t
                        from
                            users_travel_Plan
                        where
                            budget > %s and budget < %s
                        group by
                            m, city, country
                        union
                        select
                            city, country, strftime('%%m', end_time) as m, count(*) as c, "start" as t
                        from
                            users_travel_Plan
                        where
                            budget > %s and budget < %s
                        group by
                            m, city, country) as temp
                        group by
                            m, city, country
                        order by
                            s DESC
                        LIMIT 5; """, [budget_low_limit, budget_high_limit, budget_low_limit, budget_high_limit])
        whatishot_result = cursor.fetchall()
```

```
try:
    travel_plan_1 = travel_Plan.objects.raw(
        '''SELECT * FROM (users_travel_Plan a LEFT JOIN users_userprofile b ON a.user_id = b.user_id)
        WHERE a.country=%s AND a.state=%s AND a.city=%s AND a.budget>=%s AND a.budget<=%s AND a.start_time=%s AND a.end_time=%s
        AND a.user_id <> %s
        GROUP BY a.user_id''',
        [matched_country_1, matched_state_1, matched_city_1, matched_budget_l_1, matched_budget_h_1,
         matched_start_time_1, matched_end_time_1, my_id])[0]
    matched_user_name = str(travel_plan_1.user).split()[0]
    matched_user_1 = User.objects.filter(username=matched_user_name)[0]
except IndexError:
    travel_plan_1 = None
    matched_user_1 = None
```

9 Data Flow

When a user create an account, his/her email address, username, and password are saved in the ‘user’ table. A use can edit the profile to update the first name, last name into the ‘user’ table. Also he/she can update his/her organization, telephone number into the ‘userprofile’ table.

When the user post a travel plan through our website, the plan is saved in the ‘travelPlan’ table as a tuple until it is deleted. Also he/she can update the plan. When one click the ‘My Plan’ button to check his/her plans, the tuples with his/her own userID are returned.

Once a plan matches another one, two tuples are created in the match list table. When you click the ‘My Match’ button on the top of our website, your potential partners plan detail and his email will

be shown. If you reject your matched plan, these two tuples will be deleted from the match list table and try to match them with remaining plans again.

10 Advanced Functions

10.1 Advanced data visualization:

Other than a straight forward bar chart we provide you, we also create a heat map visualization. On the map, the user can see the distribution of post plans in different European cities and also the distribution of post budgets in each city. The map provides much convenience for users who want to plan a multi-city trip across different cities and users who have geologic preference in choosing their destinations.

Because of the complex coding for rendering such a complex visualization with this much information, we consider this to be an advanced function. The coding consists of two parts. On backend, function “serve” written in Python collects real time plans in our database and save the plans as a json file on the server. Such a json file can facilitate many other future tasks also.

```
def serve(request):
    print("t1")
    cursor = connection.cursor()
    cursor.execute("""select
                            city, country, budget
                        from
                            users_travel_Plan""")
    whatishot_result = cursor.fetchall()
    print("t2")
    BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    path0 = os.path.join(BASE_DIR, 'static', 'files', 'destloc_tab.csv')
    path = os.path.join(BASE_DIR, 'static', 'files', 'planAndCoordstest.json')
    # Load in a csv file for known cities' coordinates

    destloc_tab = pd.read_csv(path0)
    print("t3")
    latCoord = list()
    lonCoord = list()

    num_sim = len(whatishot_result)
    print(num_sim)
    print("t4")
    for i in range(num_sim):

        if whatishot_result[i][0] == '?esk? Krumlov':
            latCoord.append('48.8127')
            lonCoord.append('14.3175')
        else:
            findMe = destloc_tab.loc[(destloc_tab['Country'] == whatishot_result[i][1])
                                     & (destloc_tab['City'] == whatishot_result[i][0]),]
            latCoord.append(findMe['latCoord'])
            lonCoord.append(findMe['lonCoord'])
        # print(i)

    jsonfile = open(path, 'w')
    model_name = "users.travel_plan_and_coordinates"
    jsonfile.write(
        r'{ "type": "FeatureCollection", "crs": { "type": "name", "properties": { "name": "urn:ogc:def:crs:OGC:1.3:CRS84" } } ,"features":')
    jsonfile.write('\n')
    jsonfile.write('[ \n')
    i = 0
    for row in whatishot_result:
        # print((i, lonCoord[i], latCoord[i], whatishot_result[i][1], whatishot_result[i][0]))
        if len(lonCoord[i]) != 0:
            if i not in [0]:
                jsonfile.write(',')
```

```
            #
            todump = {"type": "Feature",
                      "properties": {"id": i, "rate": int(row[2])},
                      "geometry": {"type": "Point",
                                   "coordinates": [float(lonCoord[i]), float(latCoord[i]), 0.0]}}
            jsonfile.write('\n')
            json.dump(todump, jsonfile)
            i = i + 1
    print("t5")
    jsonfile.write('\n   ] }')
    jsonfile.write('\n')
    jsonfile.write('\n')
    jsonfile.write('\n')
    jsonfile.close()
    BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    path = os.path.join(BASE_DIR, 'static', 'files', 'planAndCoordstest.json')
    print(path)
    with open(path, 'r', encoding='utf-8') as myfile:
        data = myfile.read()

    print(data)
    response = HttpResponse(content=data)
    response['Content-Type'] = 'application/json'
    print("t6")
    return response
```

On the frontend, we wrote in javascript and used functions from mapbox (https://www.mapbox.com/). We first write functions to divide the budgets into five levels and set five different colors to each level. We then set up a container for us to render our map. Then we added a layer that contains circle marks, and text. In each layer, we add a filter that facilitate the rendering of the map at different scales. That is to say, when we are at the continent scale, only circle representing the number of plans in that continent will be shown. Many functions also need to be taken care of for some visualization details. For example, function donut Segment decides how the sections of a donut is painted in different colors.

```
{% extends 'base2.html' %}
{% load staticfiles %}
{% block css %}<link href="{% static 'css/other.css' %}" rel="stylesheet"/>{% endblock %}
{% block content %}
<head>
<meta name='viewport' content='initial-scale=1,maximum-scale=1,user-scalable=no' />
<script src='https://api.tiles.mapbox.com/mapbox-gl-js/v0.53.1/mapbox-gl.js'></script>
<link href='https://api.tiles.mapbox.com/mapbox-gl-js/v0.53.1/mapbox-gl.css' rel='stylesheet' />
<style>
body { margin:0; padding:0; }
#map { position:static; width:100%; }
#pam { position:absolute; top:0; bottom:0; width:100%;}
#wrap {
display: flex;
justify-content: space-around;
}
#menu {
position: static;
background: #fff;
padding: 10px;
font-family: 'Open Sans', sans-serif;
}
</style>
</head>
<body>
    <div id='pam'></div>
    <script>
    mapboxgl.accessToken = 'pk.eyJ1IjoiaHVzMTk5MyIsImEiOiJjamtoMDdtODAwb3dlM3BvYW43cG8yaDhyIn0.NlGujGLxQrPeQPfc5_apeA';
    var map1 = new mapboxgl.Map({
        container: 'pam',
        zoom: 0.3,
        center: [0, 20],
        style: 'mapbox://styles/mapbox/light-v10'
    });
    map1.addControl(new mapboxgl.NavigationControl());
    // filters for classifying earthquakes into five categories based on magnitude
    var mag1 = ["<", ["get", "rate"], 20];
    var mag2 = ["all", [">=", ["get", "rate"], 20], ["<", ["get", "rate"], 30]];
    var mag3 = ["all", [">=", ["get", "rate"], 30], ["<", ["get", "rate"], 50]];
    var mag4 = ["all", [">=", ["get", "rate"], 50], ["<", ["get", "rate"], 100]];
    var mag5 = [">=", ["get", "rate"], 100];
    // colors to use for the categories
    var colors = ['#fed976', '#feb24c', '#fd8d3c', '#fc4e2a', '#e31a1c'];
    map1.on('load', function () {
        // add a clustered GeoJSON source for a sample set of earthquakes
        map1.addSource('earthquakes', {
            "type": "geojson",
            "data": "http://ec2-3-17-57-87.us-east-2.compute.amazonaws.com/accounts/whatishot/serve/$",
            "cluster": true,
```

```
        "clusterRadius": 80,
        "clusterProperties": { // keep separate counts for each magnitude category in a cluster
            "mag1": ["+", ["case", mag1, 1, 0]],
            "mag2": ["+", ["case", mag2, 1, 0]],
            "mag3": ["+", ["case", mag3, 1, 0]],
            "mag4": ["+", ["case", mag4, 1, 0]],
            "mag5": ["+", ["case", mag5, 1, 0]]
        }
    });
    // circle and symbol layers for rendering individual earthquakes (unclustered points)
    map1.addLayer({
        "id": "earthquake_circle",
        "type": "circle",
        "source": "earthquakes",
        "filter": ["==", "cluster", 'hah'],
        "paint": {
            "circle-color": ["case",
                mag1, colors[0],
                mag2, colors[1],
                mag3, colors[2],
                mag4, colors[3], colors[4]],
            "circle-opacity": 0.6,
            "circle-radius": 12
        }
    });
    map1.addLayer({
        "id": "earthquake_label",
        "type": "symbol",
        "source": "earthquakes",
        "filter": ["!=", "cluster", true],
        "layout": {
            "text-field": ["number-format", ["get", "rate"], {"min-fraction-digits": 1, "max-fraction-digits": 1}],
            "text-font": ["Open Sans Semibold", "Arial Unicode MS Bold"],
            "text-size": 10
        },
        "paint": {
            "text-color": ["case", ["<", ["get", "rate"], 100], "black", "white"]
        }
    });
    // objects for caching and keeping track of HTML marker objects (for performance)
    var markers = {};
    var markersOnScreen = {};
    function updateMarkers() {
        var newMarkers = {};
        var features = map1.querySourceFeatures('earthquakes');
        // for every cluster on the screen, create an HTML marker for it (if we didn't yet),
        // and add it to the map if it's not there already
        for (var i = 0; i < features.length; i++) {
            var coords = features[i].geometry.coordinates;
            var props = features[i].properties;
            if (!props.cluster) continue;
            var id = props.cluster_id;
            var marker = markers[id];
            if (!marker) {
                var el = createDonutChart(props);
                marker = markers[id] = new mapboxgl.Marker('pam',{element: el}).setLngLat(coords);
            }
            newMarkers[id] = marker;

            if (!markersOnScreen[id])
                marker.addTo(map1);
        }
        // for every marker we've added previously, remove those that are no longer visible
        for (id in markersOnScreen) {
            if (!newMarkers[id])
                markersOnScreen[id].remove();
        }
        markersOnScreen = newMarkers;
    }
    // after the GeoJSON data is loaded, update markers on the screen and do so on every map move/moveend
    map1.on('data', function (e) {
        if (e.sourceId !== 'earthquakes' || !e.isSourceLoaded) return;

        map1.on('move', updateMarkers);
        map1.on('moveend', updateMarkers);
        updateMarkers();
    });
});
```

```
    // code for creating an SVG donut chart from feature properties
    function createDonutChart(props) {
        var offsets = [];
        var counts = [props.mag1, props.mag2, props.mag3, props.mag4, props.mag5];
        var total = 0;
        for (var i = 0; i < counts.length; i++) {
            offsets.push(total);
            total += counts[i];
        }
        var fontSize = total >= 1000 ? 22 : total >= 100 ? 20 : total >= 10 ? 18 : 16;
        var r = total >= 1000 ? 50 : total >= 100 ? 32 : total >= 10 ? 24 : 18;
        var r0 = Math.round(r * 0.6);
        var w = r * 2;
        var html = '<svg width="' + w + '" height="' + w + '" viewbox="0 0 ' + w + ' ' + w +
            '" text-anchor="middle" style="font: ' + fontSize + 'px sans-serif">';

        for (i = 0; i < counts.length; i++) {
            html += donutSegment(offsets[i] / total, (offsets[i] + counts[i]) / total, r, r0, colors[i]);
        }
        html += '<circle cx="' + r + '" cy="' + r + '" r="' + r0 +
            '" fill="white" /><text dominant-baseline="central" transform="translate(' +
            r + ', ' + r + ')">' + total.toLocaleString() + '</text></svg>';
        var el = document.createElement('div');
        el.innerHTML = html;
        return el.firstChild;
    }
    function donutSegment(start, end, r, r0, color) {
        if (end - start === 1) end -= 0.00001;
        var a0 = 2 * Math.PI * (start - 0.25);
        var a1 = 2 * Math.PI * (end - 0.25);
        var x0 = Math.cos(a0), y0 = Math.sin(a0);
        var x1 = Math.cos(a1), y1 = Math.sin(a1);
        var largeArc = end - start > 0.5 ? 1 : 0;
        return ['<path d="M', r + r0 * x0, r + r0 * y0, 'L', r + r * x0, r + r * y0,
            'A', r, r, 0, largeArc, 1, r + r * x1, r + r * y1,
            'L', r + r0 * x1, r + r0 * y1, 'A',
            r0, r0, 0, largeArc, 0, r + r0 * x0, r + r0 * y0,
            '" fill="' + color + '" />'].join(' ');
    }
    </script>
</body>
{% endblock %}
```

10.2 Advanced Match:

This is considered as one advanced function because of the complicated logic. Match here means to match two very similar plans from two different posters. To consider whether two plans can be matched, our rules here restrict that all the attributes except the budget must be exactly the same. The budget is considered matched if it falls in a range from 0.8 to 1.2 times the poster's budget. If two plans are matched, their statuses in table travel_Plan will be changed from unmatched to matched. After that, both two tuples will be stored in the table match_list.

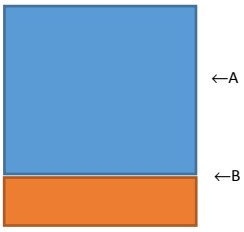
At first, we will start by explaining the logic of choosing the appropriate plan to match. One argument of this match_helper is limit and this limit restricts the range for searching.

```
def match_helper(user_1,plan_1,limit):
    myplan1=travel_Plan.objects.get(user=user_1,plan_id=plan_1)
    try:
        travel_plan_1 = travel_Plan.objects.raw(
            '''SELECT * FROM users_travel_Plan WHERE country=%s AND state=%s AND city=%s AND budget>=%s AND budget<=%s AND start_time=%s
            AND end_time=%s AND user_id <> %s AND plan_id>%s AND status=%s ORDER BY plan_id ASC''',
            [myplan1.country, myplan1.state, myplan1.city, myplan1.budget*0.8, myplan1.budget*1.2,
             myplan1.start_time, myplan1.end_time, user_1, limit, 'unmatched'])[0]
    except IndexError:
        travel_plan_1 = None

    return travel_plan_1
```

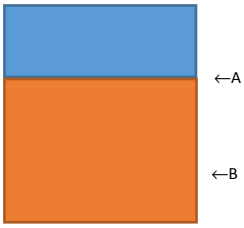
In our scenario, A just posted a plan and limit here is set to 0 which means we search from the very bottom of this table. If so far no matched plan is found, in this case nothing happens and A needs to wait for his matched plan. If A's plan has a matched plan right now, for now, the limit will be set to the plan_id of the current matched plan because any plan_id which is less than the current matched

plan's plan_id has already been checked and did not fulfill the requirement. After this, it comes to the next step in the procedure. Let's call the matched user to be B. Both A and B can choose to reject or confirm his partner based on the provided information. As long as one person in this match chooses to reject his partner, both two tuples will be deleted from the table match_list and their corresponding plans in the table travel_Plan will be marked unmatched. For both A and B, we still provide other opportunities and at this moment the match_helper will be ran again.
For user A, the limit is set to be the plan_id of user B's plan and for the next search, the range starts from this limit.



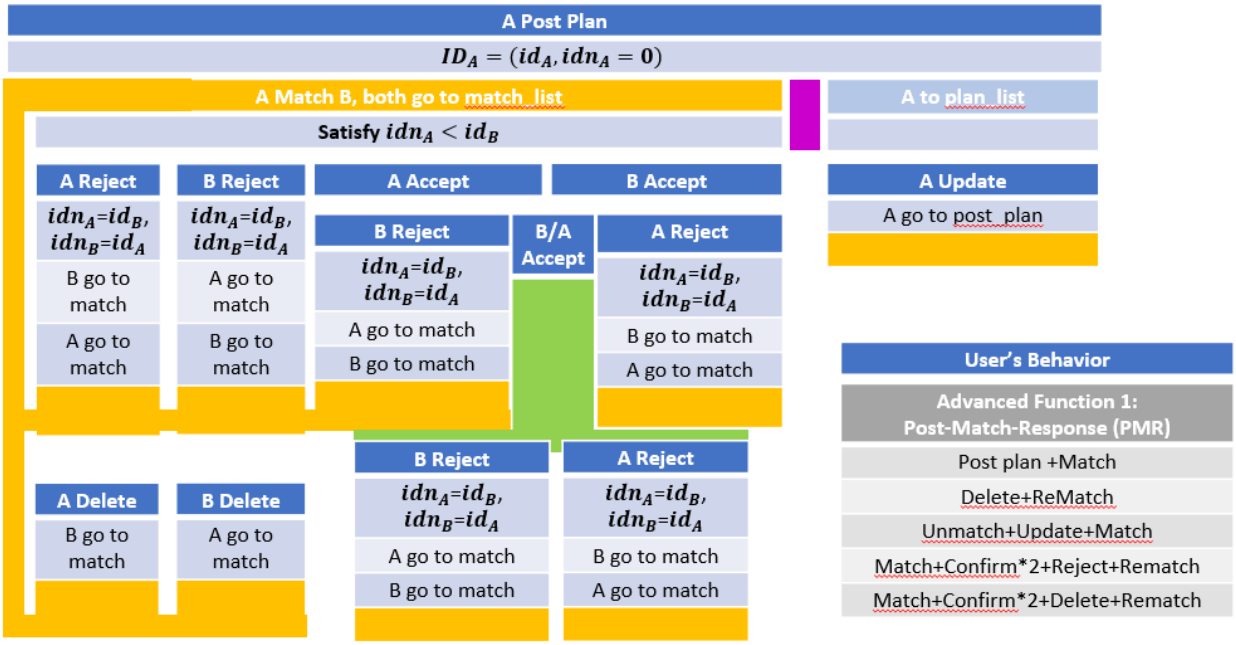
The blue part indicates the appropriate searching range and the orange part indicates the invalid searching range.

For user B, the limit is set to be the plan_id of user A's plan and for the next search, the range starts from this limit.



By restricting the searching range, we can promise that no duplicate plan (such as previously rejected plan) will be matched again.

The graph below precisely shows all the cases we have tested and made sure about its correctness. The blue part stands for the user's behavior. In the beginning, user A posts a plan and our match function starts to look for the matched tuples, here limit is 0. Then there are two cases, one is A matches B and the other one is no matched plan found. For the first one, there are four cases: (A rejects, B does nothing), (A does nothing, B rejects), (A accepts, B rejects), (B accepts, A rejects). For the second one, A can choose to update his plan for future matching.



The grey block on the right-lower corner shows all the combinations we tried. From two plans matched to both two users confirm on their plan, many situations can happen, such as plan deleted and updated and they will definitely affect your future matched plans.

11 One Technical Challenge

The biggest challenge is our website is for multiple users to post, delete and update plans. Any action from any user will make impact on other people's matched plans. For example, some user updates his plan. This may affect his match list. The original matched plan should go away and the new matched plan should appear in his match list. For his process, the corresponding effects will show up in both his previous matched partner and his current matched partner. We did not recognize the difficulty of testing and debugging in the very beginning. At last, it all goes very well. I will say think earlier and never underestimate the difficulty on any multiuser task.

12 Differences from the Initial Plan

We made little change on our final project, in the initial project, we planed to find second connection to help a user to match a travel plan, However, in the final project, we found out that relational database is not sufficient for network, but we did not have enough time to change our schema in the end, so we chose to match plans based on date, budget, and cites.

13 Division of Labor

Saxue Wang	Back-end(basic function, match function and the “Hottest City” chart); Collaborate to write reports and to record the video
Yan Xu	Back-end(basic function, match function); Collaborate to write reports.
Yi Xu	Front-end. Collaborate to write reports and to record the video
Sheng Hu	Database, heat map, searching by budget range and AWS deployment; Collaborate to record the video and to write reports.