Team 14 Documentation
Sustainable Shipment Systems

Table of Contents

Introduction

This project is meant to create a system that organizes online orders in order to reduce amount of incoming/outgoing boxes and distance traveled (to create a reduction in carbon footprint) during product shipment. The system will take grouped data of similar order times, locations, and other factors, and output a list of where to deliver, what boxes to take, and at what times.

This project is aimed to reduce the amount of pollution that is created by excessive driving distances when shipping products to consumers. An efficient system that inputs product titles, shipping addresses, delayed shipping answers, and privacy responses and successfully organizes them would advantageously affect the online shopping industry and its environmental impact.

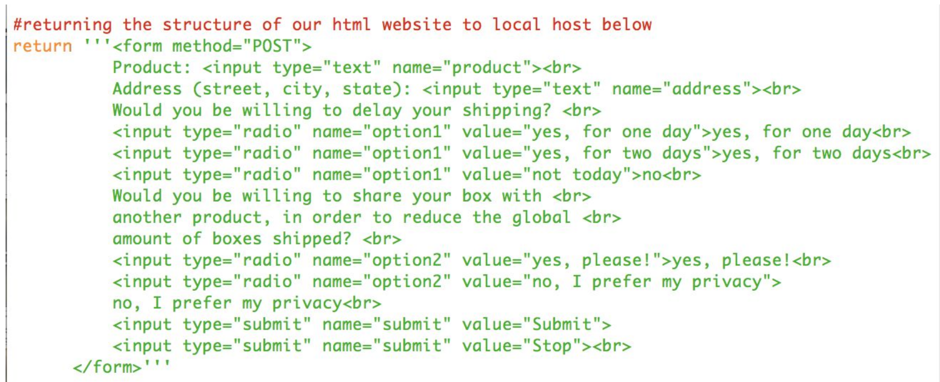
Forms that include similar shipping days (i.e. after considering delays) and addresses would be shipped to consumers at the same time. This system, however, is not entirely unique. What we have done to enhance our project was to group together two products (without the need of privacy) in the same box, to reduce the amount of cardboard boxes that are shipped. An ‘invasion of privacy’ might come to mind, but we have thought of a clever solution to this as well: if they do require privacy, their orders will simply be delivered regularly, similarly to how most orders are shipped today.

Resources

- Python, HTML
- Flask
- Google Maps API
- Folium

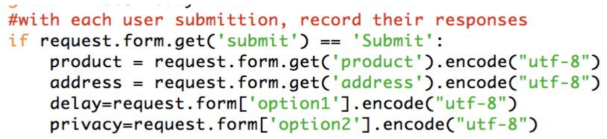
Instructions

- Download Python, Flask, and all libraries necessary (time, googlemaps, folium, json, math)
- Develop frontend:
  - With Flask, ask users for the product, address, willingness to delay, and privacy preferences.
    - user HTML structure such as in figure 1
    - record data using request.form.get, as in figure 2



```
#returning the structure of our html website to local host below
return '''<form method="POST">
          Product: <input type="text" name="product"><br>
          Address (street, city, state): <input type="text" name="address"><br>
          Would you be willing to delay your shipping? <br>
          <input type="radio" name="option1" value="yes, for one day">yes, for one day<br>
          <input type="radio" name="option1" value="yes, for two days">yes, for two days<br>
          <input type="radio" name="option1" value="not today">no<br>
          Would you be willing to share your box with <br>
          another product, in order to reduce the global <br>
          amount of boxes shipped? <br>
          <input type="radio" name="option2" value="yes, please!">yes, please!<br>
          <input type="radio" name="option2" value="no, I prefer my privacy">
          no, I prefer my privacy<br>
          <input type="submit" name="submit" value="Submit">
          <input type="submit" name="submit" value="Stop"><br>
      </form>'''
```

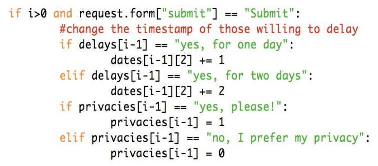
Figure 1: HTML structure



```
#with each user submittion, record their responses
if request.form.get('submit') == 'Submit':
    product = request.form.get('product').encode("utf-8")
    address = request.form.get('address').encode("utf-8")
    delay=request.form['option1'].encode("utf-8")
    privacy=request.form['option2'].encode("utf-8")
```

Figure 2: record data

  - Take time stamp using time.time()
  - Append all user data to different lists (products, addresses, delays, privacies)
  - Now, restructure delay and privacy lists: (see figure 3)

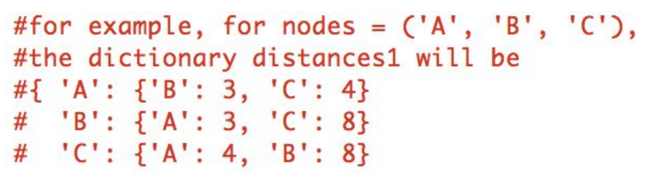


```
if i>0 and request.form["submit"] == "Submit":
        #change the timestamp of those willing to delay
        if delays[i-1] == "yes, for one day":
                dates[i-1][2] += 1
        elif delays[i-1] == "yes, for two days":
                dates[i-1][2] += 2
        if privacies[i-1] == "yes, please!":
                privacies[i-1] = 1
        elif privacies[i-1] == "no, I prefer my privacy":
                privacies[i-1] = 0
```

Figure 3: restructure lists

    - for those who wanted to delay in one day, add a day to their time stamp
    - for those who wanted to delay in two days, add two days to their time stamp
    - for those who wanted privacy, write “0” and later add to privacy orders
    - for those who did not need privacy, write “1” and later add to general orders
  - create a long string of each order, with each of the user input separated with a dash (on which we could later parse and seperate)
- Develop the shipping algorithm:
  - Develop the address-to-coordinate function (latitude, longitude)
    - Use Google’s Geolocation API to take the address strings and turn them into coordinates
    - Calculate distance between locations using the Haversine Formula
  - Develop the shipping lists (General Orders vs. Private Orders)
    - Split the input strings on the “-” to provide access to the user’s order
    - Separate the orders into two separate lists: the General Order List and the Private Order List
      - To do so, use the parsed user data, and check the privacy input value (3rd data value in the list)
      - Add private orders to the Private Order List, and all other orders to the General Order List

- Lump the same day, same address general orders
    - Iterate through the General Order List and combine all the orders for the same day, with the same address
    - Make sure to remove repeat instances of the same address when lumping! Here, turning the address list into an address set is useful for removing duplicates
- Develop most-efficient path algorithm:
  - Taking in a list of addresses of that day’s orders from above, create a dictionary such that:
    - Each key is an address, and each value is another dictionary
    - Each nested dictionary will have all the other addresses (excluding the one in the original key), and the values will be their distance to the original key/address
    - See example in figure 4



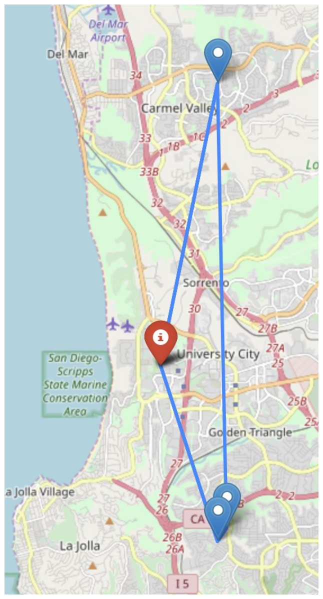
```
#for example, for nodes = ('A', 'B', 'C'),
#the dictionary distances1 will be
#{ 'A': {'B': 3, 'C': 4}
#  'B': {'A': 3, 'C': 8}
#  'C': {'A': 4, 'B': 8}
```

Figure 4: distances dictionary

  - Now, develop an algorithm that maps the route:
    - start at base:
      - check distances to all other points (through the dictionary)
      - choose the shortest path and advance to that point
    - from next chosen point:
      - check distances to all other points (through the dictionary), excluding those already checked
      - Choose the shortest path and advance to that point
    - Do so repeatedly (until back at base), and print out a list of points (tuple coordinates) in the order of the path we just found

- To see result:
  - user folium library to map the base, the coordinate tuples we just produced, and create a line between those coordinate tuples (see result in figure 5).

Figure 5: result map



Conclusion

The most prominent limiting factor we experienced was time. If we were given more time, we would have learned the intricacies of HTML. Our UI in general would be more user-friendly; we would change the webpage so that the input boxes would be detailed and the text would not be so stylistically bland.

We would have loved to dive deeper into Python as well. It was difficult to create a fully-functioning app without a developed knowledge of Python and Flask. Although we were familiar with the language, we did not have the time to thoroughly understand its intricacies and inherent upsides. With Flask, we wish we had more time to develop a complex web app and to understand how to utilize the many libraries available.

We learned during the tedious process of debugging (and debugging, and debugging) that with enough determination, we can surmount the toughest obstacles. At first, we had a rigorous time creating the environment in which we ran our HTML code. Then, we had a plethora of syntax problems along with logical errors in our Python code and algorithm, specifically with duplicate user orders appearing, or user orders not being added to the order lists. But, through it all, we were able to find solutions and together, arrive to presentation day with a fully-developed, functioning platform.

Our end product is something we are genuinely very proud of. Regardless of how hindering time was, we were able to develop an efficient data organizing system that could one day could help in the necessary reduction of our global carbon footprint.

References

- Flask — The microframework for python we used (FrontEnd)
  - http://flask.pocoo.org/
- Google Map Geocoding API — The API we used to calculate distances
  - https://developers.google.com/maps/documentation/geocoding/start
- Python 2.7 — The backbone of our code interpreted and organized user input data
  - https://www.python.org/downloads/
- Haversine Formula - Formula for finding the distance between points on the Earth (Geolocation)
  - https://community.esri.com/groups/coordinate-reference-systems/blog/2017/10/05/haversine-formula

- Shoutout to Dillon Hicks and Bassel Hatoum for the tips and check-ins! Thank you!
- Shoutout to Stack Overflow for keeping us sane!