OPEN HOUSE ROUTE PLANNER

Except where reference is made to the work of others, the work described in this masters project is my own or was done in collaboration with my advisory committee. Further, the content of this masters project is truthful in regards to my own work and the portrayal of others' work. This masters project does not include proprietary or classified information.

_____

Alexander Jansing

Certificate of Approval:

_____

Jorge Novillo
Professor
Department of Computer Science

_____

PLACEHOLDER
Professor
Department of Computer Science

_____

PLACEHOLDER
Professor
Department of Computer Science

_____

PLACEHOLDER
Dean
College of Engineering

Open House Route Planner

Alexander Jansing

A Masters Project

Submitted to the
Graduate Faculty of the
State University of New York Polytechnic Institute
in Partial Fulfillment of the
Requirements for the
Degree of

Master of Science

Utica, New York
April 25, 2019

OPEN HOUSE ROUTE PLANNER

Alexander Jansing

_____
Signature of Author

_____
Date of Graduation

VITA

Alexander Jansing's interest in Computer Science started early; at the age of 4. The first computer his family had had two 5.25" floppy drives and the operating system needed to be loaded into memory at boot. He used that computer to teach himself some basic command line operations and played games like chess. Seeing this interest in computer and logic, his parents started teaching him basic math before he even entered school. He built his first PC when he was 8 years-old and took as many Mathematics, Computer Science, Digital Electronics, and various science courses as he could in grade school. This fascination with mathematical concepts continued through his education and subsequent careers as an F-16 avionics technician, data scientist, and software engineer while working for the United States Air Force, Booz Allen Hamilton, and Lockheed Martin, respectively. He received an Bachelors of Science in Applied Mathematics from SUNY Oswego and is pursuing an Masters of Science in Computer and Information Sciences at SUNY Polytechnic Institute.

MASTERS PROJECT ABSTRACT

OPEN HOUSE ROUTE PLANNER

Alexander Jansing

Master of Science, April 25, 2019
(B.S., SUNY Oswego, 2015)

41 Typed Pages

Directed by Jorge Novillo

This program begins to allow users to enter a series of open houses and finds a route that will allow the user to visit the maximum number of open houses given the constraints of travel time between locations and when the open houses are open.

The program accepts a directory of ICS files [5] that contain open house data. The files are parsed, geocoded, and cached in a database. Then a *directions matrix* is constructed to create a graph. This graph is traversed using edge weights to determine how long it takes to get from one house to the next. Adding the edge weight to *current time* variable, it is then determined if the user would arrive at a given open house in time. If the user arrives at an open house in time, then the traversal continues until either there are either no more houses to travel to, or the user would not be able to arrive at any other houses in time.

ACKNOWLEDGMENTS

This work is dedicated to my parents. Without their guidance, I would not be the person I am today. I joined the United States Air Force after their suggestion, where I gained a greater appreciation for my education and the opportunities afforded to me.

I would like to thank each of my committee members:

Gerard Aiken supplied the Esri ArcGIS Developer credits required to do this work.

I would also like to acknowledge Jennifer Tran, Sylvia Pericles, Zhushun Cai, and Oliver Medonza for aiding the initial code base where we won the Esri API Prize and the Grand Prize at Hack Upstate XI.

This work was funded by Booz Allen Hamilton tuition assistance program.

Style manual or journal used Journal of Approximation Theory (together with the style known as "sunypolyms"). Bibliography follows van Leunen's *A Handbook for Scholars*.

Computer software used The document preparation package TEX (specifically LATEX2e) together with the style-file `sunypolyms.sty`.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

OPEN HOUSE ROUTING PLANNER

1.0.1 Motivation

I have been looking for houses. When I add open houses to my Google Calendar, I am able to request direction to whatever house is open next in time, but I was thinking, *"What if two houses are significantly far apart, open at similar times, and there are other houses in each of their respective neighborhoods that open at different times? Is there a way I can plan my day of house hunting so that I can attend all of the open houses?"* The answer to this question is, "yes, within reason."

1.0.2 Objective

Given a series of open houses the application should find routes that will allow the user to visit the maximum number of open houses given the constraints of *travel time* and *when the open houses are open*.

After stating the problem, it was divided up into several part:

- where the houses were with respect to each other,
- when the open houses were,
- and try to determine the path I needed to take to visit as many open houses as possible.

I will describe how each of these tasks were accomplished and what other work needed to be done to facilitate that work.

1.1 Requirements

As with most projects, a bit of legwork is involved before even starting the main problem. Before routes can be derived from the data, we need to know

- what kind of data the system will accept,
- what kind of ETL (Extraction, Transformation, and Loading) processes will need to occur,
- if/how there data will be cached,
- what infrastructure can we set up to support these requirements,
- and how might we will we compute routes?

1.1.1 ETL

As for the first three points, the following describe what kind of data the program accepts, its transformation, and how there data is cached.

Extraction

Data extraction was performed manually as it seems that many real estate sites did not want to hand over data programatically; or if they did the purchase of an API key was required. The data used for this project was sourced from Trulia [13] in the form of ICS files [5].

Transformation

Open House data is typically provided with in a human readable format with an address written like “100 Seymour Ave, Utica, NY 13502” and times provided in the form "10AM to 12PM." These forms of data need to be handled somehow. Luckily, when downloading calendar data the time data comes in ISO 8601 Notation (*i.e.,* yyyymmddThhmmssZ).

That just leaves the address to be *geocoded*; the process of converting addresses to a coordinate system. The ArcGIS Developer API provides an easy way to geocode address data. This

process will be described in the section 2.1 along with all the other transformations required to consistently gather and use data.

Loading

To avoid having to query the ArcGIS API repeated and spending the credits that was required [3], the data is stored in a MongoDB instance. This not only helps the program be more economically efficient, but cuts down on latency during testing and future delivery.

1.1.2 Infrastructure

Docker is a container platform that helps facilitate rapid prototyping, development, and compartmentalization of development of projects [1]. Docker, and subsequently docker-compose [10], was used on this project to set up network on systems that could easily be deployed on a cloud service.

This docker-compose network consisted of three containers to query ArcGIS API, store geocoded data, and perform computation on graph data structure; esri, mongo, and routefinder, respectively (see Figure 1.1 and Appendix A.1).

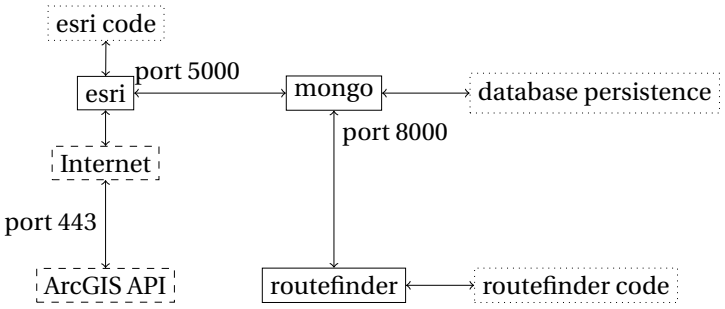

Figure 1.1: Docker-compose network diagram.

1.1.3 Computation

When the project was started, it was thought that Spark could be used to perform graph processing [12]. Later on, it was decided that the graphs were not going to be large enough to have to worry about optimizing the processing. A graph data structure [9] was still used for handing the data and computing routes.

CHAPTER 2

SETUP AND EXECUTION

2.1 Preprocessing

For the purposes of this and following sections refer to Figure 2.1 as examples of ICS files. Even though the information gathered was readily available online, I have redacted addresses, links, summaries, and descriptions to avoid divulging any PII.

```
BEGIN:VCALENDAR
VERSION:2.0
BEGIN:VEVENT
URL: REDACTED
DTSTART:20190414T153000Z
DTEND:20190414T170000Z
SUMMARY: REDACTED
DESCRIPTION: REDACTED
LOCATION: REDACTED (location0)
END:VEVENT
END:VCALENDAR
```

```
BEGIN:VCALENDAR
VERSION:2.0
BEGIN:VEVENT
URL: REDACTED
DTSTART:20190413T150000Z
DTEND:20190413T170000Z
SUMMARY: REDACTED
DESCRIPTION: REDACTED
LOCATION: REDACTED (location1)
END:VEVENT
END:VCALENDAR
```

Figure 2.1: Example ICS files (locations 0 and 1).

Given a directory of n ICS files (as per section 1.1.1) the data is parsed by the ICSParser (Appendix B.1.1) and the address information is *safely geocoded* (querying for existence in MongoDB, otherwise querying ArcGIS API for information) by MongoOps and a Flask REST endpoint

(Appendices B.1.2 and B.1.5 respectively) [8]. This process yields data like that shown in Figure 2.2.

```
{'_id': ObjectId('5cabffe2671c85002d41afb3'), 'url': 'REDACTED', 'dtstart': '20190414T153000Z',
    ↪ 'dtend': '20190414T170000Z', 'summary': 'REDACTED', 'description': 'REDACTED',
    ↪ 'location': {'geometry': {'x': x_0, 'y': y_0, 'spatialReference': {'wkid': 4326,
    ↪ 'latestWkid': 4326}}, 'attributes': {'Loc_name': 'World', 'Status': 'M', 'Score': 100,
    ↪ ... 'X': x_0, 'Y': y_0, 'DisplayX': x_{d0}, 'DisplayY': y_{d0}, 'Xmin': x_{min0},
    ↪ 'Xmax': x_{max0}, 'Ymin': y_{min0}, 'Ymax': y_{max0}, 'ExInfo': '', 'OBJECTID': 1},
    ↪ 'address': <location0>(lower case, no punctuation)}, 'address_hash': sha1(location0)}
```

Figure 2.2: Final location JSON for the location0 ICS file.

```
{'_id': ObjectId('5cad42f3671c850b358ab86b'), 'directions': [{'Time of day': '15:59:01',
    ↪ 'Direction text': 'Start at Location 0', 'Duration (min)': 0.0, 'Distance (miles)':
    ↪ 0.0}, {'Time of day': '15:59:01', 'Direction text': REDACTED, 'Duration (min)': 0.73,
    ↪ 'Distance (miles)': 0.4}, {'Time of day': '15:59:45', 'Direction text': REDACTED,
    ↪ 'Duration (min)': 7.33, 'Distance (miles)': 4.15}, {'Time of day': '16:06:21',
    ↪ 'Direction text': REDACTED, 'Duration (min)': 11.9, 'Distance (miles)': 8.26}, {'Time of
    ↪ day': '16:10:55', 'Direction text': REDACTED, 'Duration (min)': 14.04, 'Distance
    ↪ (miles)': 9.74}, {'Time of day': '16:13:03', 'Direction text': REDACTED, 'Duration
    ↪ (min)': 18.45, 'Distance (miles)': 13.29}, {'Time of day': '16:17:28', 'Direction text':
    ↪ REDACTED, 'Duration (min)': 19.67, 'Distance (miles)': 13.85}, {'Time of day':
    ↪ '16:18:41', 'Direction text': 'Finish at Location 1, on the right', 'Duration (min)':
    ↪ 19.67, 'Distance (miles)': 13.85}], 'directions_hash': sha1(location0 + location1),
    ↪ 'start': location0, 'stop': location1}
```

Figure 2.3: Directions information between location0 and location1.

Gathering directions between locations follows a similar pattern. After the ICS files have been parsed into data that looks like Figure 2.2, the same Flask app has another REST endpoint to accept two locations and return the directions, along with estimated durations as each step (see Figure 2.3). At every point, when data has been geocoded (i.e. Figures 2.2 and 2.3) that data is saved in an easily retrievable fashion via unique constructable hashes based on the location(s)' address(es)'s.

2.2 Creating A Graph

2.2.1 Directions Matrix

Once the ICS files have been processed, a *direction matrix* is created. First, a verbose matrix is created by taking the addresses from the ICS files and querying the database for all $\binom{n}{2}$ combinations of SHA-1 $\left(\text{CONCAT}\left(location_i, location_j\right)\right)$, where $0 <= i < j <= n-1$. Each entry of this matrix would look like that of Figure 2.3 and that is why a simplified form of the matrix is listed along side this matrix in the DirectionMatrix class (Appendix B.1.3).

This verbose directions matrix is entirely usable, but would require the code also be verbose when gathering the travel time when constructing the graph and assigning edge weights. Hence, a simplified matrix that looks like Figure 2.4 is created as a refinement of the original data.

```
   [{'_id': ObjectId('5cad42f3671c850b358ab86b'), 'url': REDACTED, 'dtstart': '20190414T153000Z',
       ↪ 'dtend': '20190414T170000Z', 'summary': REDACTED, 'description': REDACTED, 'location':
       ↪ {'geometry': {'x': x_0, 'y': y_0, 'spatialReference': {'wkid': 4326, 'latestWkid':
       ↪ 4326}}, 'attributes': {'Loc_name': 'World', 'Status': 'M', 'Score': 100, ... 'X': x_0,
       ↪ 'Y': y_0, 'DisplayX': x_{d0}, 'DisplayY': y_{d0}, 'Xmin': x_{min0}, 'Xmax': x_{max0},
       ↪ 'Ymin': y_{min0}, 'Ymax': y_{max0}, 'ExInfo': '', 'OBJECTID': 1}, 'address': REDACTED},
       ↪ 'address_hash': sha1(location0), 'durations': [[1, 13.85]]}
{'_id': ObjectId('5cac003a671c85002d41afb9'), 'url': REDACTED, 'dtstart': '20190413T150000Z',
     ↪ 'dtend': '20190413T170000Z', 'summary': REDACTED 'description': 'REDACTED, 'location':
     ↪ {'geometry': {'x': x_1, 'y': y_1, 'spatialReference': {'wkid': 4326, 'latestWkid': 4326}},
     ↪ 'attributes': {'Loc_name': 'World', 'Status': 'M', 'Score': 100, ... 'X': x_1, 'Y': y_1,
     ↪ 'DisplayX': x_{d1}, 'DisplayY': y_{d1}, 'Xmin': x_{min1}, 'Xmax': x_{max1}, 'Ymin':
     ↪ y_{min1}, 'Ymax': y_{max1}, 'ExInfo': '', 'OBJECTID': 1}, 'address': REDACTED},,
     ↪ 'address_hash': sha1(location1), 'durations': [[0, 14.15]]}]
```

Figure 2.4: Simplified Directions Array (Matrix).

2.2.2 Open House Graph

With the directions matrix derived, a graph can now be constructed and routes can be discovered that direct users to a maximum number of houses. Looking at Figure 2.4 we can see that edge data has been defined by the durations key-value pairing, the start and stop times are included; all that is left is to do one last transformation to what the graph expects and create the graph (Figure 2.5).

```
for i in range(len(sdm)):
  start = str(int(sdm[i]['dtstart'][9:-3])-400)
  start_minutes = int(start[:-2])*60 + int(start[-2:])
  end = str(int(sdm[i]['dtend'][9:-3])-400)
  end_minutes = int(end[:-2])*60 + int(end[-2:])
  sdm[i]['start_minutes'] = start_minutes
  sdm[i]['end_minutes'] = end_minutes

vertices = []
V = None
for i in range(len(sdm)):
    V = {'ID' : i,
           'start' : sdm[i]['start_minutes'],
           'end' : sdm[i]['end_minutes'],
           'edges' : sdm[i]['durations'],
           'address_hash' : sdm[i]['address_hash'],
           'address' : sdm[i]['location']['address']}
    vertices += [V]
ohg = OpenHouseGraph(vertices)
```

Figure 2.5: Last Refinement and Graph Construction.

The ID of a given vertex (as seen in second loop of Figure 2.5), corresponds to the 0^{th} index of an edge (durations in the directions matrix). So that data from Figure 2.4 will looks like Figure 2.6, meaning that it will take 13.85 minutes travel from the house 0 to house 1, and 14.15 in the other direction.

```
    [{'ID': 0 'durations': [[1, 13.85]], ...},
    {'ID': 1 'durations': [[0, 14.15]], ...}]
```

Figure 2.6: Vertex-Edge Relation.

The Graph constructor (see Appendix B.1.4) takes this information and stores the edges as [[0, 1, 13.85], [1, 0, 14.15]], and stores the rest of the JSON as the vertex information. To help keep the explanation as general as possible, the following notation will be used:

- vertices will be referred to as v_i,
- the start and stop time of open houses belong to a vertex will be denoted t_{i0} and t_{i1} respectively,
- edges from v_i to v_j will be denoted e_{ij}.

- values from edges e_{ij} can be accessed as they would be in an array. i.e., $e_{01}[2] = 13.85$ (Figure 2.6).

Figure 2.5 shows some shorthand that I wanted to use. Instead of working with a built in time API, I assumed that the day did not matter. All that needed to work with time was to convert the time to a base-10 numbering system (minutes since midnight), This allows computation in the graph to use duration in minutes returned by the ArcGIS API, and just keep a running tally of the time (scoped to a recursive function call) when figuring out when one would arrive at a given house.

2.2.3 Finding Routes

Starting Vertices

To start the search of paths, a for-loop is used to select starting vertices to enter the graph on. The program is written such that when we arrive at the first vertex, v_i, that the we arrive as soon as the open house starts, t_{i0}. t_{i0} will be used as the initializing value for a time tracking variable, T, and v_i is used to start the current paths' set of visited locations, V, and will later populate the $trip$ variables that contains the possible trips given the arbitrary starting point v_i.

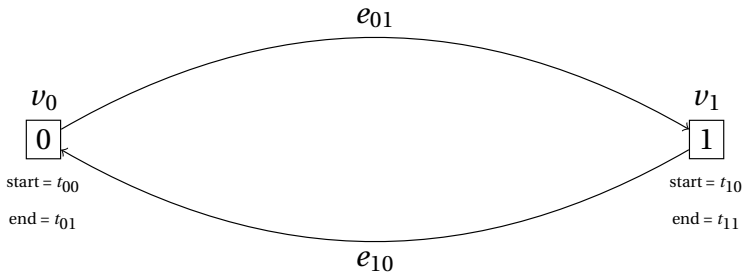

Figure 2.7: A trivial case of the Open House Graph.

Discoving Routes

Now that we are at v_i at time T and having visited V, the visit_next function in OpenHouse-Graph follows this pattern

1. Gather acyclic edges and iterate on them,

2. spend 30 minutes (default value) at the current house and assign that to a departure time variable ($T_d := T + 30$),

3. determine the what the arrival time at v_j would be and assign that to an arrival time variable ($T_a := T_d + e_{ij}[2]$),

4. determine if the open house has opened and determine if it has closed,

   Determining both values allows us to perform additional checks on the time.

   - If the open house has **not opened**, we can tell the function to set the arrival time to the start of the open house ($T_a := t_{j0}$).
   - If the open house is **not closed**, we can tell the function that the arrival time is correct and does not need updating.
   - If the open house is **closed** (or in the unlikely case that the time is in some other state), we can tell the function to set the arrival time to -1 ($T_a := -1$).

   If $T_a > 0$, then then function is allowed to proceed.

5. v_j is added to a list of visited V ($V := [\cdots, v_i, v_j]$, where V previously equaled $[\cdots, v_i]$),

6. finally, a recursive call is made to the visit_next function to start the pattern all over again.

   Once no valid acyclic edges exist,

   - the function will return its current V as long as $|V| > 1$; adding the value of V of a deeper recursive layer to the $trip$ variable one level higher.
   - Otherwise visit_next will return all the $trip$s it has discovered back to the user.

Evaluating The Results

It is at this point where I called my development to a halt. With our trivial case above, the scenario returns $[[0,1],[1,0]]$. This a worst case scenario for systems with more vertices, but

would most likely only occur with 4 or less houses (assuming a 30 minute stay at each house). With larger graphs like that of Figure 2.8, the returned scenario often times looks like Figure 2.9.

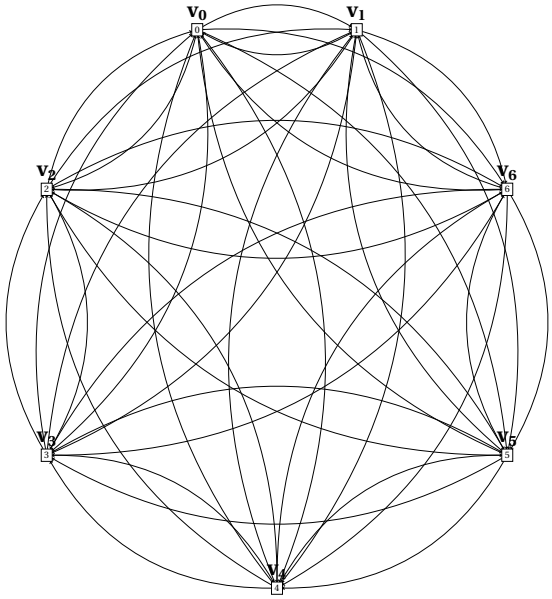

Figure 2.8: Open House Graph containing seven houses.

The sheer number of results may seem intimidating, but it is a simple task a finding the max length of the trips and then displaying a trimmed down version. In the sample results of Figure 2.9, we can see that it is possible to visit all seven houses in a couple of different ways. This will not always be the case because of the limitations of travel and when the houses are open to the public. I tried to stress the system with 10 open houses and the program always completes its task in less than 0.6 seconds.

$$[[0,1,2,3,4,5,6],[0,1,2,3,4,6,5],[0,1,2,3,4,6],[0,1,2,3,4,5],$$
$$\cdots,[0,2,4,5,6],\cdots,[0,4,6],[2,0,1,3,5,4,6],\cdots[6,5],\cdots]$$

Figure 2.9: Simulated results of graph of seven vertices.

CHAPTER 3

CONCLUSIONS AND FUTURE WORK

3.1 Conclusions

3.1.1 Reflection

In the beginning of this project, I wanted to use different higher level graph theory concepts to try to solve the problem of routing the user to different open houses, rather than via brute force. Though after some time I decided to get answers before trying to dive too deeply into what *could* be done. After getting the methodology down, I worked out some scenarios by hand and went house hunting. After confirming that I could find a route (a route in which I found the house I now own), I proceeded to implement the program that is being presented here.

The methodology of routing the user to house is fairly sound, though after nailing down the 30 minute stay as a default time to stay at a house, I asked my realtor how long people usually stayed at an open house and he said it was more like 10 - 15 minutes. A change to how much time the user would potentially stay at a house can be changed easily, but I did not concern myself too much with that because the core logic of the problem was completed and I figured assuming the user stayed a house longer than normal would allow for human error to occur throughout the day.

3.1.2 Justification

I would still like to look into using methods other than the brute force to solve this problem, but for routing a user to a small number of open houses it might not be necessary to do anything more than brute force. Open houses are usually open from about 10am until 5pm; 7 hours. If travel time is elimatinated ($e_{ij} = 0, \forall i, j$) and one spends on average 30 minutes at each house, then the max recursive depth of the visit_next function is 14. Additionally, an interview with

David W. Jansing brought to my attention that with how bounded this particular problem is, more complex functions may potentially require more time to set up than brute force needs to run [11].

In an attempt to stress the program, I have requested an export of open house data in San Antonio, TX, but I foresee the traffic of a large city keeping the number potential routes low (many short routes may be returned, but will be ignored for larger ones).

3.2 Future Work

The future work of this project comes down to some of the stretch goals I provided myself:

- implement other methods of traversing the graph,
- provide a more user friendly way to ingest data,
- and supply a front end that users could interact with and receive detailed information from.

Other things that may be worked on in the future are:

- clean up the information stored in the database. Right now, if you do not clear the database from week to week and you query a house that was open previously, you will receive the hours of the last week.
- And there is an issue with character encoding with the ICSParser. The issue was ignored during the scope of this project, but if this were to be serves up to a client, then it would not be good to just through out whatever is not understood.

BIBLIOGRAPHY

[1] About docker ce. https://docs.docker.com/install/, 2019.

[2] Arcgis api for python. https://developers.arcgis.com/python/, 2019.

[3] Arcgis for developers. https://developers.arcgis.com/, 2019.

[4] esridocker/arcgis-api-python-notebook. https://hub.docker.com/r/esridocker/arcgis-api-python-notebook, 2019.

[5] icalendar | wikiwand. https://www.wikiwand.com/en/ICalendar, 2019.

[6] jupyter/pyspark-notebook. https://hub.docker.com/r/jupyter/pyspark-notebook, 2019.

[7] mongo. https://hub.docker.com/_/mongo, 2019.

[8] Welcome | flask (a python microframework). http://flask.pocoo.org/, 2019.

[9] R. Agarwal. To all data scientists - the one graph algorithm you need to know. https://towardsdatascience.com/to-all-data-scientists-the-one-graph-algorithm-you-need-to-know-59178dbb1ec2, 2019.

[10] Docker Documentation. Install docker compose. https://docs.docker.com/compose/install/, 2019.

[11] David W Jansing and Alexander P Jansing. Traveling salesman interview, Apr 2019.

[12] Spark.apache.org. Graphx | apache spark. https://spark.apache.org/graphx/, 2019.

[13] Trulia. Trulia: Real estate listings, homes for sale, housing data. https://www.trulia.com/, 2019.

APPENDICES

APPENDIX A
RESOURCES

A.1 Docker Images

- jupyter/pyspark-notebook:7254cdcfa22b [6] - container renamed *routefinder*.
- esridocker/arcgis-api-python-notebook:1.5 [4] - container renamed *esri*.
- mongo:4.1 [7] - container named *mongo*.

A.2 Docker Compose

```
version: '3.5'
services:
  mongo:
    container_name: mongo
    image: mongo:${MONGODB_TAG}
    ports:
      - ${MONGO_EXTERNAL}:${MONGO_INTERNAL}
    environment:
      - MONGO_INITDB_ROOT_USERNAME=${MONGO_INITDB_ROOT_USERNAME}
      - MONGO_INITDB_ROOT_PASSWORD=${MONGO_INITDB_ROOT_PASSWORD}
    volumes:
      - ./persistence/mongo/db:/data/db
  pyspark:
    env_file: credentials.env
    container_name: routefinder
    image: jupyter/pyspark-notebook:${PYSPARK_TAG}
    ports:
      - ${PYSPARK_EXTERNAL}:${PYSPARK_INTERNAL}
    volumes:
      - ./data:/data
      - ./dependencies/pyspark.txt:/home/jovyan/requirements.txt
      - ./persistence/pyspark:/home/jovyan/work
    environment:
      - MONGO_INITDB_ROOT_USERNAME=${MONGO_INITDB_ROOT_USERNAME}
      - MONGO_INITDB_ROOT_PASSWORD=${MONGO_INITDB_ROOT_PASSWORD}
  esri:
    env_file: credentials.env
    container_name: esri
    image: esridocker/arcgis-api-python-notebook:${ESRI_TAG}
```

```
    ports:
      - ${ESRI_EXTERNAL}:${ESRI_INTERNAL}
      - ${GET_DIRECTIONS_EXTERNAL}:${GET_DIRECTIONS_INTERNAL}
    volumes:
      - ./data:/home/jovyan/data
      - ./dependencies/esri.txt:/home/jovyan/requirements.txt
      - ./persistence/esri:/home/jovyan/work

networks:
  neighborhood:
    name: ${DC_NETWORK}
    driver: bridge
```

A.2.1 backend/docker/.env

```
# Docker-compose Variables
DC_NETWORK=home_finding_network

# MongoDB Variables
MONGODB_TAG=4.1
MONGO_INITDB_ROOT_USERNAME=admin
MONGO_INITDB_ROOT_PASSWORD=admin
MONGO_EXTERNAL=8000
MONGO_INTERNAL=8080

# PySpark Variables
PYSPARK_TAG=7254cdcfa22b
PYSPARK_EXTERNAL=8001
PYSPARK_INTERNAL=8888

# esridocker/arcgis-api-python-notebook
ESRI_TAG=v1.5
ESRI_EXTERNAL=8002
ESRI_INTERNAL=8888
GET_DIRECTIONS_EXTERNAL=5000
GET_DIRECTIONS_INTERNAL=5000
```

APPENDIX B
SOURCE CODE

B.1 apjansing/Open-House-Route-Planner

B.1.1 ICSParser

Class for parsing ICS files.

```
from datetime import datetime
import dateutil.parser
from sys import argv

class ICSParser():
    """
    docstring for ICSParser
    """
    def __init__(self, filename = None):
        self.filename = filename
        self.event = []
        if(filename != None):
            self.event = self.parse_ics()

    def parse_ics(self, filename = None):
        if filename == None:
            filename = self.filename
        event = []
        with open(filename, "r") as file:
            try:
                for line in file:
                    line = line.strip().split(":", 1)
                    if line[0] in ["URL", "DTSTART", "DTEND", "SUMMARY", "DESCRIPTION", "LOCATION"]:
                        if(len(line)==2):
                            event += [[line[0].lower(), line[1].strip()]]
            except:
                print("There was a problem parsing open house ics file {}".format(filename))
        self.event = event
        return event

    def to_dict(self, parse_datetime = False):
        event_dict = {}
        for E in self.event:
            if parse_datetime and (E[0] == "dtstart" or E[0] == "dtend"):
                E[1] = dateutil.parser.parse(E[1])
            event_dict[E[0]] = E[1]
        return event_dict
```

B.1.2 MongoOps

- Class for:

- loading data to database,
- querying the database for geocoded address,

- querying the database for directions,
- querying the ArcGIS Developer API for geocoded address,
- and querying the ArcGIS Developer API for directions.

```
import os
import re
import json
import string
import hashlib
import requests
import datetime
import pandas as pd
import pymongo as pm
from bson import Binary
from bs4 import BeautifulSoup
from urllib.parse import quote_plus

class MongoOps():
    """
    docstring for MongoOps
    """
    def __init__(self, username="admin",
                 password="admin",
                 host="mongo:27017",
                 testing=False):
        self.username = username
        self.password = password
        self.host = host
        self.uri = "mongodb://%s:%s@%s" % (quote_plus(self.username),
                                    quote_plus(self.password), self.host)

        self.connection = pm.MongoClient(self.uri)
        self.homes_database = self.connection.homes_database

        self.homes_collection = self.homes_database.homes
        self.directions_collection = self.homes_database.directions

        self.create_loc_index(self.homes_collection)
        self.create_loc_index(self.directions_collection)
        self.testing = testing

    def drop_table(self, collection):
        collection.drop()

    def create_loc_index(self, collection, name = "geometry", location_type = "2dsphere"):
        collection.create_index([(name, location_type)])


    def load_dict(self, collection, data):
        collection.insert_one(data)

    def search_collection(self, collection, search_val = None, limit = 10):
        if limit != None:
            return collection.find(search_val).limit(limit)
        else:
            return collection.find(search_val)

    def address_info_in_database(self, collection, address_hash):
        search_val = {"address_hash" : address_hash}
        cursor = self.search_collection(collection, search_val = search_val, limit=1)
        return cursor.count() > 0
```

```
def format_address(self, address):
    formatted_address = address.lower()
    remove_spaces = lambda s : remove_spaces(re.sub(" ", " ", s)) if " " in s else s
    whitelist = string.ascii_lowercase + string.digits + ' '
    formatted_address = remove_spaces(formatted_address)
    formatted_address = ''.join(c for c in formatted_address if c in whitelist)
    return formatted_address

def flask_request(self, url):
    try:
        r = requests.get(url)
    except Exception as e:
        return "proxy service error: " + str(e), 503
    soup = BeautifulSoup(r.content, "html.parser")
    address_info = json.loads(str(soup))
    return address_info

def _query_for_location_info(self, address):
    address = self.format_address(address)
    url = 'http://esri:5000/get_geocode/%s' % quote_plus(address)
    r = self.flask_request(url)
    r['address'] = address
    return r

def safe_query_for_location_info(self, event):
    address = ""
    if type(event)==dict:
        try:
            if len(event.keys()) < 1:
                return None
        except:
            return None
        address = event["location"]
    elif type(event)==str:
        address = event
    address = self.format_address(address)
    address_hash = self.get_hash(address)
    if not self.address_info_in_database(self.homes_database.homes, address_hash):
        self.print_test("%s not found! Gathering data from Esri." % address)
        data = self._query_for_location_info(address)
        print(address)
        if type(event)==dict:
            event["location"] = data
            event["address_hash"] = address_hash
        elif type(event)==str:
            event = {"location": data, "address_hash": address_hash}
        self.load_dict(self.homes_database.homes, event)
    else:
        self.print_test("%s found! Gathering data from MongoDB." % address)
    result = self.search_collection(self.homes_database.homes, {"address_hash" :
     ↪ address_hash}).next()
    return result

def get_hash(self, strings):
    hasher = hashlib.sha1()
    string = "".join(strings)
    hasher.update(string.encode('utf-8'))
    hashed_directions = hasher.digest()
    return hashed_directions

def get_address_from_loc_data(self, loc):
    if isinstance(loc, dict):
        try:
```

```
            loc_str = loc["location"]["address"]
        except:
            print("Dict passed is not properly formatted. Should have {\"location\": {...,
                ↪ \"address\": <address>}} format.")
            return None
    elif isinstance(loc, str):
        try:
            loc_str = json.loads(loc).get("location.address", loc.get("location",
                ↪ loc.get("address")))
        except:
            loc_str = loc
    else:
        print("Location data not string or dict.")
        return None
    return loc_str.strip()

def get_directions(self, start, stop):
    """
    get_directions is designed to receive jsons for the following schema.
    A schema checker may be added at some point in the future.
        {"geometry": {"x" : <float>, "y": <float> }, *}
    """
    start_str = json.dumps(start, skipkeys=True)
    stop_str = json.dumps(stop, skipkeys=True)

    directions_json = {}

    url_path = 'get_directions/{ "location_1": %s, "location_2": %s }' % (start_str, stop_str)
    url = 'http://esri:5000/%s' % url_path

    directions = self.flask_request(url)
    directions_json['directions'] = directions

    hashed_directions = hashed_directions = self.get_hash([start_str, stop_str])

    directions_json["directions_hash"] = hashed_directions

    return directions_json

def direction_in_database(self, collection, direction_hash):
    search_val = {"directions_hash": Binary(data = direction_hash)}
    cursor = self.search_collection(collection, search_val = search_val, limit=1)
    return cursor.count() > 0

def _query_for_directions(self, start, stop):
    directions = self.get_directions(start, stop)
    return directions


def safe_query_for_directions(self, start, stop):
    start_str = self.get_address_from_loc_data(start)
    start_data = self.safe_query_for_location_info(start_str)

    stop_str = self.get_address_from_loc_data(stop)
    stop_data = self.safe_query_for_location_info(stop_str)

    directions_hash = self.get_hash([start_str, stop_str])
    if not self.direction_in_database(self.directions_collection, directions_hash):
        self.print_test("Directions from %s to %s not found! Gathering data from Esri." %
            ↪ (start_str, stop_str))
        directions = self._query_for_directions(start_data["location"], stop_data["location"])
        directions["directions_hash"] = directions_hash
        directions["start"] = start_data["location"]["address"]
        directions["stop"] = stop_data["location"]["address"]
```

```
            self.load_dict(self.directions_collection, directions)
        else:
            self.print_test("Directions from %s to %s found! Gathering data from MongoDB." %
                ↪ (start_str, stop_str))
            directions = self.search_collection(self.directions_collection, {"directions_hash":
                ↪ directions_hash}).next()
        return directions

    def print_test(self, string):
        if self.testing:
            print(string)

if __name__ == "__main__":
    from ICSParser import ICSParser
    mops = MongoOps()
    ics = ICSParser()

    ics.parse_ics("/data/download.ics")
    start = mops.safe_query_for_location_info(ics.to_dict())

    ics.parse_ics("/data/download (1).ics")
    stop = mops.safe_query_for_location_info(ics.to_dict())

    print(mops.safe_query_for_directions(start, stop))
```

B.1.3 DirectionsMatrix

Class for creating a matrix of directions data pertaining to locations passed to it. It utilized the MongoOps class to safely query for directions between open houses.

```
import json
import numpy as np
from os import listdir
from MongoOps import MongoOps
from ICSParser import ICSParser
from os.path import isfile, join


class DirectionsMatrix():
    def __init__(self, locations, mops = MongoOps()):
        self.mops = mops
        self.locations = locations
        self.directions_matrix = None
        self.simplified_directions_matrix = None

    def get_directions_matrix(self):
        directions_matrix = []
        for location1 in self.locations:
            loc1_row = []
            for location2 in self.locations:
                if location1['location'] != location2['location']:
                    loc1_loc2_directions = self.mops.safe_query_for_directions(start = location1, stop
                        ↪ = location2)
                    loc1_row += [loc1_loc2_directions]
                else:
                    loc1_row += [None]
            directions_matrix += [loc1_row]
        self.directions_matrix = np.array(directions_matrix)

    def generate_simplified_directions_matrix(self):
```

```
        try:
            if self.directions_matrix == None:
                self.get_directions_matrix()
        except:
            pass
        simplified_directions_matrix = []
        for i in range(len(self.directions_matrix)):
            row_data = self._get_start_location_info(i)
            simplified_directions_matrix += [self._get_time_between_points(i, row_data)]
        self.simplified_directions_matrix = np.array(simplified_directions_matrix)

    def _get_start_location_info(self, row):
        start = ''
        if self.directions_matrix[row][0] != None:
            start = self.directions_matrix[row][0]['start']
        else:
            start = self.directions_matrix[row][1]['start']
        return self.mops.safe_query_for_location_info(start)

    def _get_time_between_points(self, row, simplified_directions_row):
        durations = []
        for j in range(len(self.directions_matrix[row])):
            if self.directions_matrix[row][j] == None:
                pass
                # durations += [[j, -1]]
            else:
                duration = self.directions_matrix[row][j]['directions'][-1]['Duration (min)']
                durations += [[j, duration]]
        simplified_directions_row['durations'] = durations
        return simplified_directions_row


if __name__ == "__main__":
    sample_data_files = [f for f in listdir("/data") if isfile(join("/data", f)) and f != '.DS_Store'
        ↪ ]
    mops = MongoOps()
    locations = []
    for open_house_file in sample_data_files:
        parser = ICSParser("/data/%s" % open_house_file)
        event = parser.to_dict()
        # print(event)
        if event != None:
            result = mops.safe_query_for_location_info(event)
            locations += [result]
    dir_mx = DirectionsMatrix(locations, mops)
    # dir_mx.get_directions_matrix()
    dir_mx.generate_simplified_directions_matrix()
    # print(dir_mx.directions_matrix[0][1])
    # print(dir_mx.directions_matrix.shape)
    print(dir_mx.simplified_directions_matrix)
```

B.1.4 OpenHouseGraph

A graph data structure used for computing routes one might take while visiting open houses.

- Inspired by: Data Scientists, The one Graph Algorithm you need to know [9] - Basis for the OpenHouseGraph class.

```
import json
import numpy as np
```

```
from pprint import pprint
from os import listdir
from os.path import isfile, join
from MongoOps import MongoOps
from ICSParser import ICSParser
from make_directions_matrix import DirectionsMatrix
import random
import time

def random_combination(iterable, r):
    '''
    This function helps test different scenarios.
    '''
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

class Graph(object):
    """
    A Python Class
    A simple Python graph class, demonstrating the essential
    facts and functionalities of graphs.
    Original implementation from https://www.python-course.eu/graphs_python.php
    Changes to include weighted edges from https://towardsdatascience.com/to-all-data-scientists-the-
        one-graph-algorithm-you-need-to-know-59178dbb1ec2
    Some functions have been removed because they are not
    going to be used, and I would like to protect future
    users from using this graph object incorrectly.
    """
    def __init__(self, graph_dict=None):
        """ initializes a graph object
            If no dictionary or None is given,
            an empty dictionary will be used
        """
        if graph_dict == None:
            graph_dict = {}
        self.__graph_dict = graph_dict
        self.__vertices = self.vertices()

    def vertex_ids(self):
        """ returns the vertices of a graph """
        vertices = []
        for val in self.__graph_dict:
            vertices += [val['ID']]
        self.__vertices = vertices
        return list(self.__vertices)

    def vertices(self):
        """ returns the vertices of a graph """
        vertices = []
        for val in self.__graph_dict:
            ver = self.__removekey(val, 'edges')
            vertices += [ver]
        self.__vertices = vertices
        return list(self.__vertices)

    def get_vertex_from_vid(self, vid):
        '''
        returns the vertex given its id.
        I would LIKE to assume that the vertex's id will match its location in
        graph_dict, but I won't in the case that someone passes in an irregualar
        graph_dict to the Graph object.
        '''
```

```
        vertex = None
        for val in self.__graph_dict:
            if val['ID'] == vid:
                vertex = val
        if vertex == None:
            raise Exception('Vertex ID provided not found.')
        return vertex

    def edges(self):
        """ returns the edges of a graph """
        return self.__generate_edges()

    def get_edges_from_vid(self, vid):
        '''
        returns the edges from a vertex, given its id.
        '''
        vertex = self.get_vertex_from_vid(vid)
        edges = self.get_edges(vertex)
        return edges

    def __generate_edges(self):
        """ A static method generating the edges of the
            graph "graph". Edges are represented as sets
            with one (a loop back to the vertex) or two
            vertices
        """
        edges = []
        for vertex in self.__graph_dict:
            edges += self.get_edges(vertex)
        return edges

    def get_edges(self, vertex):
        """
        returns the edges of a vertex dictionary
        """
        edges = []
        for neighbour in vertex['edges']:
            weight = neighbour[1]
            n = neighbour[0]
            v = vertex['ID']
            if [v, n, weight] not in edges:
                edges.append([v, n, weight])
        return edges

    def __str__(self):
        res = "vertices:\n"
        for k in self.__vertices:
            res += str(k) + " \n"
        res += "\nedges:\n"
        for edge in self.__generate_edges():
            res += str(edge) + " "
        return res

    def adj_mat(self):
        return self.__graph_dict

    def __removekey(self, d, key):
        r = dict(d)
        del r[key]
        return r

class OpenHouseGraph(Graph):
    """
    OpenHouseGraph extends the Graph Object. Base Graph object was inspired by,
```

```
https://towardsdatascience.com/to-all-data-scientists-the-one-graph-algorithm-you-need-to-know-
    59178dbb1ec2
"""

def visit_next(self, current_vertex, arrival_time, visited, trip = [], average_time_at_each_house
    ↪ = 30):
    """
    Recursive function, given by a starting vertex, iterate over outbound
    edges to travel to every house and determine what the time would be
    after getting to a destination.
    """
    acyclic_edges = self.get_acyclic_edges(self.get_edges(current_vertex), visited)
    for a_edge in acyclic_edges:

        # Determine when you leave at the next house.
        departure_time = arrival_time + average_time_at_each_house

        # Determine when you 'arrive' at the next house.
        next_arrival_time = departure_time + a_edge[2]

        idx = a_edge[1]
        a_vertex = self.get_vertex_from_vid(idx)
        opened, closed = self.open_and_closed(next_arrival_time, a_vertex)
        wait_function = self.get_wait_function(opened, closed)
        next_arrival_time = wait_function(a_vertex['start'], next_arrival_time)

        # If you're not too late, continue with this path
        if next_arrival_time > 0:
            step = visited + [idx]
            trip += self.visit_next(a_vertex, next_arrival_time, step)
    return [visited] if len(visited) > 1 else trip


def flatten(self, trips):
    results = []
    for i in range(len(trips)):
        if isinstance(trips[i], int):
            return trips
        else:
            if len(trips[i]) > 0:
                results += [trips[i]]
    return results

def get_acyclic_edges(self, edges, visited):
    """
    Gets edges out of a vertex that have not been visited.
    """
    E = []
    for edge in edges:
        idx = edge[1]
        if self.been_visited(visited, idx):
            continue
        else:
            E += [edge]
    return E

def been_visited(self, visited, v):
    '''
    Checks to make sure you're not going back to a node that has already
    been visited.
    '''
    return v in visited

def open_and_closed(self, arrival_time, next_vertex):
```

```
        """
        Determines whether an open house has started/ended when you arrive.
        """
        opened = arrival_time >= next_vertex['start']
        closed = arrival_time >= next_vertex['end']
        return [opened, closed]

    def get_wait_function(self, opened, closed):
        """
        Given the combination of booleans opened and closed, set the wait variable
        to a given set of values. Return the lambda function corresponding key equal
        to the wait variable.
        """
        if not opened:
            wait = "Wait"
        elif not closed:
            wait = "No need to wait"
        elif closed:
            wait = "Too late"
        else:
            wait = "Time doesn't work that way!"
        wait_function = {
            "No need to wait" : lambda opens_at, current_time : current_time,
            "Wait" : lambda opens_at, current_time : opens_at,
            "Too late" : lambda opens_at, current_time : -1,
            "Time doesn't work that way!" : lambda opens_at, current_time : -1
        }
        return wait_function[wait]

    def convert_mins_to_time(self, time):
        """
        Converts a time in the form of 600 to its more recognisable form.
        600 corresponding to 10:00 (or 600 minutes from midnight).
        """
        hours_minutes = str(time / 60).split('.')
        hours_minutes[0] = hours_minutes[0]
        hours_minutes[1] = str(float('0.'+hours_minutes[1]) * .6)[2:4]
        return hours_minutes[0], hours_minutes[1]

if __name__ == "__main__":

    locations = []
    mops = MongoOps()
    sample_data_files = [f for f in listdir("/data") if isfile(join("/data", f)) and f != '.DS_Store'
        ↪ ]
    for open_house_file in sample_data_files:
        try:
            parser = ICSParser("/data/%s" % open_house_file)
            event = parser.to_dict()
            result = mops.safe_query_for_location_info(event)
            locations += [result]
        except:
            pass

    random_locations = locations #random_combination(locations, len(locations))
    DM = DirectionsMatrix(random_locations, mops)
    DM.generate_simplified_directions_matrix()
    sdm = DM.simplified_directions_matrix


    # Showing off EST/EDT time of day of the open houses and the conversion to minutes from midnight
        ↪ that day
    for i in range(len(sdm)):
        start = str(int(sdm[i]['dtstart'][9:-3])-400)
```

```
    start_minutes = int(start[:-2])*60 + int(start[-2:])
    end = str(int(sdm[i]['dtend'][9:-3])-400)
    end_minutes = int(end[:-2])*60 + int(end[-2:])
    sdm[i]['start_minutes'] = start_minutes
    sdm[i]['end_minutes'] = end_minutes

vertices = []
V = None
for i in range(len(sdm)):
    V = {'ID' : i,
           'start' : sdm[i]['start_minutes'],
           'end' : sdm[i]['end_minutes'],
           'edges' : sdm[i]['durations'],
           'address_hash' : sdm[i]['address_hash'],
           'address' : sdm[i]['location']['address']}
    vertices += [V]
print(vertices)
ohg = OpenHouseGraph(vertices)

### TICK ###
start = time.time()

paths = []
for v in ohg.vertices():
    starting_id = v['ID']
    starting_vertex = ohg.get_vertex_from_vid(starting_id)
    start_time = starting_vertex['start']
    path = ohg.visit_next(starting_vertex, start_time, [starting_id])
    if len(path) > 0:
        for p in path:
            if p not in paths:
                paths += [p]

### TOCK ###
end = time.time()

paths = np.array(paths)

max_len = max([len(path) for path in paths])
P = []
for path in paths:
    if len(path) >= max_len:
        P += [path]

pprint(vertices)
for path in P:
    print('------------------------- Showing path for {}'.format(path))
    for p in path:
        print(vertices[p]['address'])
print('\n')

print('''Given {} locations, Open House routing calculations took {} seconds to execute.
The maximum number of houses that could be visited was {}.'''.format(len(locations), end -start,
    ↪ max_len))
```

B.1.5 Esri Flask App

REST endpoint that is designed to accept information from MongoOps and query the ArcGIS Developer API for geocoded address information or directions between two geometry points.

```
import arcgis.network as network
```

```
import arcgis.geocoding as geocoding

from arcgis.gis import *
from arcgis.geometry import Point
from arcgis.geocoding import geocode, reverse_geocode
from arcgis.features.feature import FeatureSet, Feature

import json
import datetime
import pandas as pd

from flask import Flask
import requests

import os

app = Flask(__name__)

username = os.environ['ESRI_USERNAME']
password = os.environ['ESRI_PASSWORD']

gis = GIS('https://www.arcgis.com', username, password)
route_service_url = gis.properties.helperServices.route.url
route_layer = network.RouteLayer(route_service_url, gis=gis)
start_time = int(datetime.datetime.now().timestamp() * 1000)

@app.route('/get_directions/<coordinates>')
def get_directions(coordinates):
    """
    get_directions is designed to receive jsons for the following schema.
    A schema checker may be added at some point in the future.
    {
        "location_1": { "geometry": {"x" : <float>, "y": <float> }, * },
        "location_2": { "geometry": {"x" : <float>, "y": <float> }, * }
    }
    """
    coordinates = json.loads(coordinates)
    result = route_layer.solve(stops='''%f,%f; %f,%f'''%(coordinates["location_1"]['geometry']['x'],
        coordinates["location_1"]['geometry']['y'],
        coordinates["location_2"]['geometry']['x'],
        coordinates["location_2"]['geometry']['y']),
        directions_language='en-US', return_routes=False,
        return_stops=False, return_directions=True,
        directions_length_units='esriNAUMiles',
        return_barriers=False, return_polygon_barriers=False,
        return_polyline_barriers=False, start_time=start_time,
        start_time_is_utc=True)

    records = []
    travel_time, time_counter = 0, 0
    distance, distance_counter = 0, 0

    for i in result['directions'][0]['features']:
        tod_token = i['attributes']['arriveTimeUTC']
        time_of_day = datetime.datetime.fromtimestamp(tod_token / 1000).strftime('%H:%M:%S')
        time_counter = i['attributes']['time']
        distance_counter = i['attributes']['length']
        travel_time += time_counter
        distance += distance_counter
        records.append( (time_of_day, i['attributes']['text'],
                    round(travel_time, 2), round(distance, 2)) )

    pd.set_option('display.max_colwidth', 100)
```

```
    directions_dataframe = pd.DataFrame.from_records(records, index=[i for i in range(1, len(records)
        ↪ + 1)],
        columns=['Time of day', 'Direction text', 'Duration (min)', 'Distance (miles)'])
    directions_json = json.loads(directions_dataframe.to_json(orient='index'))
    directions_json_array = []
    for bar in directions_json:
        directions_json_array += [directions_json[bar]]
    return json.dumps(directions_json_array)

@app.route('/get_geocode/<address>')
def get_geocoded(address):
    """
    get_geocode expects and address (i.e. 100 Seymour Ave, Utica, NY 13502) in the URL and will
        ↪ return the a json in a form as shown below.
    {
    "geometry": {
        "x": -75.23401051692672,
        "y": 43.08877505712876,
        "spatialReference": {
            "wkid": 4326,
            "latestWkid": 4326
        }
    },
    "attributes": {
        "Loc_name": "World",
        "Status": "M",
        "Score": 95.18,
        "Match_addr": "Seymour Ave, Utica, New York, 13501",
        "LongLabel": "Seymour Ave, Utica, NY, 13501, USA",
        "ShortLabel": "Seymour Ave",
        "Addr_type": "StreetName",
        "Type": "",
        "PlaceName": "",
        "Place_addr": "Seymour Ave, Utica, New York, 13501",
        .
        .
        .
        "X": -75.23401051692672,
        "Y": 43.08877505712876,
        "DisplayX": -75.23401051692672,
        "DisplayY": 43.08877505712876,
        "Xmin": -75.23501051692672,
        "Xmax": -75.23301051692671,
        "Ymin": 43.08777505712876,
        "Ymax": 43.089775057128755,
        "ExInfo": "100",
        "OBJECTID": 1
    }
    """
    return json.dumps(geocode(address=address, as_featureset=True).features[0].as_dict)

@app.route('/')
@app.route('/<path:p>')
def wikiproxy(p = ''):
    import requests
    url = 'https://apjansing.github.io/Open-House-Route-Planner/{0}'.format(p)
    try:
        r = requests.get(url)
    except Exception as e:
        return "proxy service error: " + str(e), 503
    from bs4 import BeautifulSoup
    soup = BeautifulSoup(r.content, "html.parser")
    return str(soup)
```

```
if __name__ == '__main__':
    app.run(host="0.0.0.0")
```

B.2 Dependencies

B.2.1 Dependencies required by system.

Docker	18.09.2
docker-compose	1.23.2
Esri Developer Account	N/A

B.2.2 Python Modules used within docker containers:

ics	0.4
pymongo	3.7.2
requests	2.21.0
pandas	0.24.1
bs4	0.0.1
virtualenv	16.4.0
Flask	0.12.2