SUNY
Polytechnic
Institute

Open House Route Planner

Alexander Jansing

Computer Science Department
State University of New York,
Polytechnic Institute
Utica, NY 13502
USA
jansina@sunypoly.edu

15 April 2019

Abstract: Consectetur sit aliquip pariatur anim et proident nisi mollit labore exercitation sunt exercitation fugiat.

$\rightarrow\delta_{\alpha}\tau^{\alpha}\leftarrow$

I. Introduction

A. 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

Over the last year, I had been looking for a house and was trying to visit as many open houses as I could. So the question became, *"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."

B. 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 phrasing stating the problem, the problem 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.

II. 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?

A. 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.

1) Extraction: Data extraction was performed manually as it seems that many realestate 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[9] in the form of ICS files[4].

2) 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 III along with all the other transformations required to consistently gather and use data.

3) 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.

B. Infrastructure

Docker is a modern technology that helps facilitate rapid prototyping, development, and compartmentalization of development of projects [5]. Docker, and subsequently docker-compose[6], 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 and Appendix IV-A1).

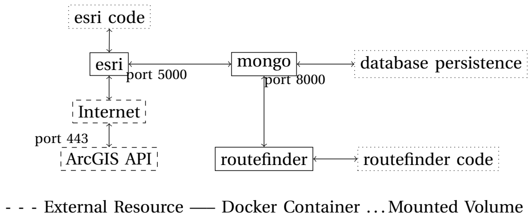

Fig. 1. Docker-compose network diagram.

C. Computation

When the project was started, it was thought that Spark could be used to perform graph processing[7]. 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 was still used for handing the data when the time came to computing routes[1]. The vertices were given IDs, addresses, start and end times, and edges with weights.

III. Preprocessing

For the purposes of this and following sections refer to Figure 2 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 (location1)
END:VEVENT
END:VCALENDAR
```

```
BEGIN:VCALENDAR
VERSION:2.0
BEGIN:VEVENT
URL: REDACTED
DTSTART:20190413T150000Z
DTEND:20190413T170000Z
SUMMARY: REDACTED
DESCRIPTION: REDACTED
LOCATION: REDACTED (location2)
END:VEVENT
END:VCALENDAR
```

Fig. 2. Example ICS files (locations 1 and 2).

Given a directory of n ICS files (as per section II-A1) the data is parsed by the ICSParser (IV-B) 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 [8]. This process yeilds data like that shown in Figure 3.

```
{'_id': ObjectId('5cabffe2671c85002d41afb3'), 'url':
    ↪ 'REDACTED', 'dtstart': '20190414T153000Z',
    ↪ 'dtend': '20190414T170000Z', '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_d, 'DisplayY': y_d, 'Xmin':
    ↪ x_min, 'Xmax': x_max, 'Ymin': y_min, 'Ymax':
    ↪ y_max, 'ExInfo': '', 'OBJECTID': 1}, 'address':
    ↪ <location1 (lower case, no punctuation)>},
    ↪ 'address_hash': <hash of location1>}
```

Fig. 3. Final location JSON for the location1 ICS file.

Gathering directions between locations follows in a similar manner. After the n ICS files have been parsed into data that looks like Figure 3, 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 4).

And at every point, when data has been geocoded (i.e. Figures 3 and 4) that data is saved in an easily retrievable fashion via unique constructable hashes based on the location(s)' address(es)'s.

```
{'_id': ObjectId('5cad42f3671c850b358ab86b'),
    ↪ 'directions': [{'Time of day': '15:59:01',
    ↪ 'Direction text': 'Start at Location 1', '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
    ↪ 2, on the right', 'Duration (min)': 19.67,
    ↪ 'Distance (miles)': 13.85}], 'directions_hash':
    ↪ hash of the concatenation of location1 and
    ↪ location2, 'start': location1, 'stop': location2}
```

Fig. 4. Directions information between location1 and location2.

IV. Appendix

A. Resources

1) Docker Images:

- jupyter/pyspark-notebook:7254cdcfa22b[10] - container renamed *routefinder*.
- esridocker/arcgis-api-python-notebook:1.5[11] - container renamed *esri*.
- mongo:4.1[12] - container named *mongo*.

B. Code

- apjansing/Open-House-Route-Planner - This project.
  - ICSParser - Class for parsing ICS files.
  - 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.
  - 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.
  - OpenHouseGraph - A graph data structure used for computing routes one might take while visiting open houses.
  - 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 GeoJSON Points.
- Data Scientists, The one Graph Algorithm you need to know[1] - Basis for the OpenHouseGraph class.

References

[1] Agarwal, R. (2019). To all Data Scientists - The one Graph Algorithm you need to know. [online] Towards Data Science. Available at: https://towardsdatascience.com/to-all-data-scientists-the-one-graph-algorithm-you-need-to-know-59178dbb1ec2 [Accessed 13 Apr. 2019].

[2] ArcGIS for Developers. (2019). ArcGIS API for Python | ArcGIS for Developers. [online] Available at: https://developers.arcgis.com/python/ [Accessed 13 Apr. 2019].

[3] ArcGIS for Developers. (2019). ArcGIS for Developers. [online] Available at: https://developers.arcgis.com/ [Accessed 13 Apr. 2019].

[4] Wikiwand. (2019). iCalendar | Wikiwand. [online] Available at: https://www.wikiwand.com/en/ICalendar [Accessed 13 Apr. 2019].

[5] Docker Documentation. (2019). About Docker CE. [online] Available at: https://docs.docker.com/install/ [Accessed 13 Apr. 2019].

[6] Docker Documentation. (2019). Install Docker Compose. [online] Available at: https://docs.docker.com/compose/install/ [Accessed 13 Apr. 2019].

[7] Spark.apache.org. (2019). GraphX | Apache Spark. [online] Available at: https://spark.apache.org/graphx/ [Accessed 13 Apr. 2019].

[8] Flask.pocoo.org. (2019). Welcome | Flask (A Python Microframework). [online] Available at: http://flask.pocoo.org/ [Accessed 14 Apr. 2019].

[9] Trulia. (2019). Trulia: Real Estate Listings, Homes for Sale, Housing Data. [online] Available at: https://www.trulia.com/ [Accessed 13 Apr. 2019].

[10] jupyter/pyspark-notebook. (2019). https://hub.docker.com/r/jupyter/pyspark-notebook: MongoDB.

[11] esridocker/arcgis-api-python-notebook. (2019). https://hub.docker.com/r/esridocker/arcgis-api-python-notebook: MongoDB.

[12] mongo. (2019). https://hub.docker.com/_/mongo: MongoDB.