5-2023

Fuel Prediction: Determining the Desirable Stops for the Cheapest Road Trips

Maxx Smith
University of Arkansas, Fayetteville

Fuel Prediction: Determining the Desirable Stops for the Cheapest Road Trip

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering

College of Engineering

University of Arkansas

Fayetteville, AR

April 2023

by

Maxx Smith

Advised by

Dr. Susan Gauch

Abstract

Current technology has given rise to many advanced route-planning applications that are available for use by the general public. Gone are the days of preparing for road trips by looking at a paper map for hours on end trying to determine the correct exits or calculate the distance to be traveled. However, with the use of modern technology, there is a certain aspect of forward-thinking that is now lost with planning a road trip. One of the biggest constraints that often gets left on the backburner is deciding when and where to stop to refuel the car. This report is the holistic overview of a web application designed to assist drivers with deciding when and where to stop for gas in an effort to ultimately reduce unnecessary expenses.

1. Introduction

1.1 Objectives

The main goal of this project is to create an application that can be used to plan the gas station stops during a road trip ahead of time so that the user can stop to refuel at the station with the cheapest gas listed. Even within a single city, gas prices can vary between stations. While in some cases it may only be a few cents difference, in others it may climb up to 20 cents. At face value this may seem like a mostly negligible difference, but this change in price can dramatically add up over the course of a road trip with two or three stops to fully refuel.

In order to provide ease of access to the user, this project is designed as a web application. This allows the application to be accessed with multiple devices by being hosted through a server on a network. The application should be able to take in any two addresses or locations, as well as any make and model of a car and correctly calculate the trip distance, the car’s expected miles per gallon, the expected number of stops, and a short list of gas stations at those expected stops. Once the user's data is collected and the results are calculated, they should be displayed in an easily readable format for the user on the application web page.

1.2 Background

Before starting work on this project it was essential to first understand a few crucial parts of the anticipated coding and usable algorithms. Since this project is focused less on comparing the same route with various algorithms and more on optimizing the stops along the route, this project will use a third party API, Openrouteservice, to decide the best route between two locations ahead of any calculations [5]. Dijkstra's algorithm is one of the most commonly used

shortest path algorithms that can accommodate for path weight [2]. However, it is not usable for a case like this because while it works well for shortest path calculations, the context of this project actually requires a fastest path calculation. To achieve this, the routing API used for this project uses one of two preprocessing speed-up methods depending on the situation: contraction hierarchies (CH) or ALT. Both of these are commonly used improvements to shortest-path algorithms [6, 8]. Openrouteservice creates an optimal route by combining "individual segments computed separately between consecutive pairs of points," [1]. This results in a fast and accurately optimized route to be used for the anticipated road trip.

With the route planning phase completed, the next obstacle to overcome is locating gas stations along the route. This is achieved by using a separate API, Gas Prices Scraper by Natasha Lekh, which takes in a specified location and returns a number of gas stations in the given area based on Google Maps data [7]. This ensures that the data will stay current and also be as accurate as Google provides.

Lastly the application needs to be able to calculate when the user's car is expected to run out of gas. In order to accomplish this, the application scrapes from a website dedicated to graphing the recorded miles per gallon for various car models, which is data provided by the website's dedicated community [4]. Fuelly.com is the website containing the vehicle data, and the scraping tool used was Beautiful Soup 4. Fueleconomy.gov, while technically reports more official miles per gallon listings, does not permit scraping robots to obtain data from the website.

2. Related Works

The main source of inspiration used for this project came from a combination of Google Maps and the Waze navigation app. Google Maps allows the user to search ahead along the route to look for gas stations, but only does so when prompted and also does not sort by price in any form. Waze, on the other hand, does sort the gas stations by price and even categorizes them into high, medium, and low prices according to the area, but these prices are not updated as often as they are through Google. Waze also does not anticipate when the user is going to need to stop for gas, and therefore uses a shortened list of gas stations along the route. This inherently limits the user's options and ability to choose which gas station they may prefer.

3. Approach and Functionality Development

3.1 Overview

As stated before, the goal of this project is to ultimately create an easy-to-use web application. To produce this, the application runs with a Python backend connected to a server through implementation of the Flask web framework [3]. The website takes the user's input and sends it to the backend for the main calculations to be done and for the main API's to be called. The Python script then sends data back to the web front end through use of the "render_template" function in Flask. This function allows Python variables to be passed in, which can then be retrieved and manipulated by JavaScript before ultimately being formatted and displayed to the user in HTML.

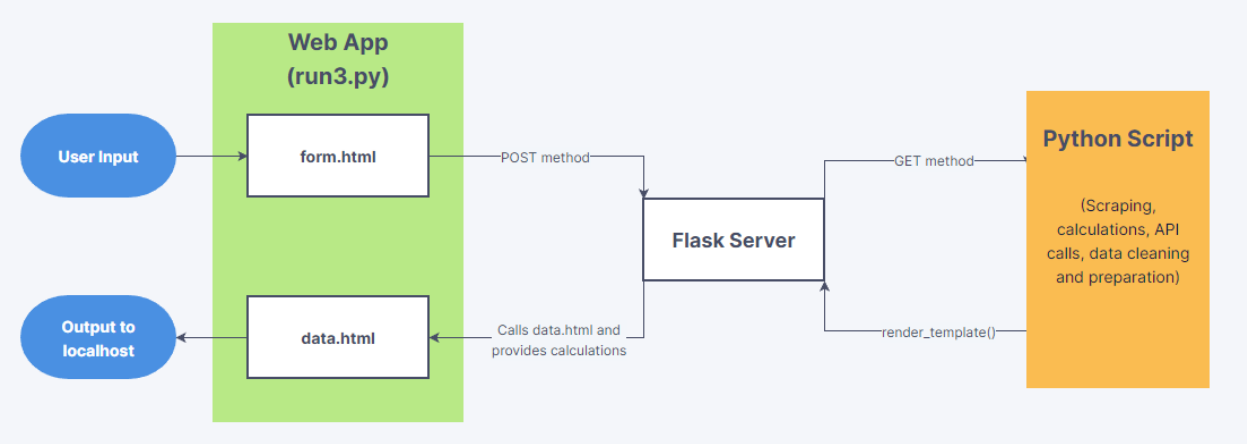


Figure 1: Project Application Architecture

Throughout the development timeline, the application itself naturally went through several iterations, with each one building on the previous version. The most effective way to have a reasonable development timeline was to build each new piece of code to run on its own as much as possible, and then implement it into the actual project application code. This style of development helped reduce errors and allowed for much cleaner code. Below is a summarized list of each of the versions and their respective, significant changes.

3.2 Calculating MPG and Travel Distance

This was the most basic module of the application and was the most straightforward to accomplish. The first version developed, web.py, would accept the user input for the car make and model, and then it would navigate to and scrape the corresponding fuelly.com page for the miles per gallon data of that specific car. It also established the usage of the ‘re’ library, which is a module that allows the code to feed strings into regular expressions, scanning for any matches to the written formula.

```python
# import libraries
from bs4 import BeautifulSoup
import requests
import re #regex

# Gather the user's car info
# make = input("\nWhat is your car's make? ")
# model = input("What is your car's model? ")
make = "cadillac"
model = "srx"


print("\nSearching for a " + make + " " + model + " in the Fuelly database...")

# Request to website and download HTML contents
url='https://www.fuelly.com/car/' + make + '/' + model
print("URL is: " + url)
req=requests.get(url)
content=req.text

print("Souping the website - ")
soup = BeautifulSoup(content, 'html.parser')
# Finds the car page's title, useful for making sure the car actually exists
print(soup.title)
# Searches for the page's highcharts div
print("Finding Highchart div...")
chart = soup.find("div", {"id" : "highchart-model-mpg-byvehicle"})
print(chart)
print("Chart is of type: ")
print(type(chart))

print("Trying regex")
#chart is a 'Tag' BS4 object, needs to be typecasted
print("Count locations: ")
print([m.start() for m in re.finditer('count', str(chart))] )
print("MPG locations: ")
print([m.start() for m in re.finditer('rnd_mpg', str(chart))] )

```

Figure 2: web.py depicting basic scraping functionality using Beautiful Soup 4

Equation 1: m.start() for m in re.finditer(‘count’, str(chart)).

Equation 2: m.start() for m in re.finditer(‘rnd_mpg’, str(chart)), *combined with Eq. 1 will return MPG values and the corresponding counts.*

With the fuel data obtained, the next goal was to translate it into a usable average. Version webv2.py improves on the previous version of the code by interpreting and making sense of the data. This data is averaged from the obtained Highchart to calculate the car’s miles per gallon, as well as applied to some placeholder location data to simulate already having a route gathered. This was also the first instance where the differences in regular expression modules between languages would start to appear. For example, in Python it is generally good practice to input a string into a regex formatted as r”...” as it tells the code to interpret each of the contained characters literally.

```
print('\n')
#split data-avgs portion of chart string by commas (highchart and BS4 made it a comma separated list)
dataList = chart['data-avgs'].split(',')
dict = []
total = 0
countSum = 0

for i in range(len(dataList)):
    if i%2 == 0:
        #findall works better than finditer because it returns a list of strings, not an object
        #When writing strings in python that contain backslashes, start with "r" to write raw. Good for regex.
        count = int(re.findall(r"(\d+)",dataList[i])[0])
    else:
        mpg = int(re.findall(r"(\d+)",dataList[i])[0])
        dict.append({count:mpg})
        total += (count*mpg)
        countSum += count
avgMPG = total/countSum
tankValue = math.ceil((drivingDistance/(avgMPG*tankSize)))

print(dict)
print("\nYour car's expected MPG is: " + '{:.2f}'.format(avgMPG))

print("You will have to drive " + str(drivingDistance) + " miles to get from " + loc1 + " to " + loc2 + ".")

print("You will be stopping for gas " + str(tankValue) + " times on your trip.")
```

Figure 3: MPG is calculated and applied to trip distance to determine number of stops.

In order for the car's averaged MPG to actually be used, the next stage of the project was to calculate the distance to be traveled during the course of the road trip. Version webv3.py shows the first implementation of Openrouteservice which is being used to calculate the route between two locations. The application is also correctly identifying the starting location and destination inputs.

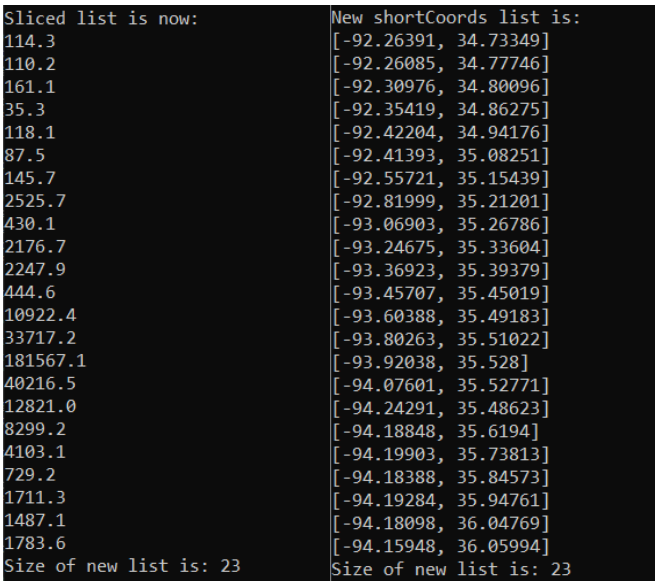
```
    #retrieve summary distance

    m = re.search(r"(\d+)", str(routes))
    #m.group(1) ensures you grab the first number
    print("\nretrieved distance is: " + m.group(1))
    milesDist = float(m.group(1)) * 0.000621371
    print("\ndistance converted is: " + str(milesDist))

What is your starting location? Little Rock
What is your destination? Fayetteville
Little Rock, Pulaski County, Arkansas, United States
Fayetteville, Washington County, Arkansas, United States
34.7465071 -92.2896267
36.0625843 -94.1574328

retrieved distance is: 306212

distance converted is: 190.271256652
```

Figure 4: Openrouteservice returns data in meters, so it must be converted for MPG usage

Equation 3: milesDist = float(m.group(1)) + 0.000621371, *converts the returned route distance in meters into miles to be used for MPG calculations.*

3.3 Determining Stops and Retrieving Gas Prices

This stage of the project is where the application begins to take form. Version webv5.py now has a much higher functionality when compared to the previous models as it is now capable of deciding ahead of time where along the route the user is going to need to stop for gas. It is important to note that the approximation algorithm for deciding where to stop in this version is not the same algorithm used in the final version. To elaborate, it is helpful to first understand how the graphics for a route-planning service are drawn. Because routes almost never consist solely of straight lines, the routes are drawn using an extensive list of coordinates instead, which can be found in the geometric polyline generated by the route-planning service. This version of the script attempts to match the coordinates found in the polyline with the driving instructions and distances that would be given to the user. Unfortunately, the coordinates often fail to accurately represent the intended location. Because the driving instructions are based on distance traveled and the coordinates are simply based on the shape of the route, the two are not correlated and should not be attributed to one another. Specific test cases showed that the algorithm used in this version struggles with longer stretches of roads.



Sliced list is now:	New shortCoords list is:
114.3	[-92.26391, 34.73349]
110.2	[-92.26085, 34.77746]
161.1	[-92.30976, 34.80096]
35.3	[-92.35419, 34.86275]
118.1	[-92.42204, 34.94176]
87.5	[-92.41393, 35.08251]
145.7	[-92.55721, 35.15439]
2525.7	[-92.81999, 35.21201]
430.1	[-93.06903, 35.26786]
2176.7	[-93.24675, 35.33604]
2247.9	[-93.36923, 35.39379]
444.6	[-93.45707, 35.45019]
10922.4	[-93.60388, 35.49183]
33717.2	[-93.80263, 35.51022]
181567.1	[-93.92038, 35.528]
40216.5	[-94.07601, 35.52771]
12821.0	[-94.24291, 35.48623]
8299.2	[-94.18848, 35.6194]
4103.1	[-94.19903, 35.73813]
729.2	[-94.18388, 35.84573]
1711.3	[-94.19284, 35.94761]
1487.1	[-94.18098, 36.04769]
1783.6	[-94.15948, 36.05994]
Size of new list is: 23	Size of new list is: 23

Figure 5: The left side depicts the distance traveled for each "step" of the route. The right side depicts the selected corresponding coordinate pair for each step. Closer inspection shows that because there are many relatively small steps and few large steps, this method of attempting to relate coordinates to the steps will not produce accurate results.

Equation 4: for i in slicedDistances: if(currFuelMeters - float(i) <= 0): , *this equation iterates through the list of distance-steps to be traveled, reducing the car's fuel with each step. When the car's fuel would drop below the available fuel amount, the previous set of coordinates would be retrieved and alert the user to stop there.*

Webv5.py marks the major development of finishing the module responsible for retrieving the necessary gas stations and their respective prices. This is the final version of the back end Python script before fully implementing it to be run through the Flask server instead of standalone. With the stop locations decided from the previous algorithm, which was inaccurate but functional enough, the script now extracts the data from the gas station API and formats it into a JSON to be submitted back through the Flask server through the render_template function.

```
        #Implement apifygas.py
        run_input = {
            "location": stopCity,
            "maxCrawledPlacesPerSearch": 3, #change for locations
        }

        # Run the actor and wait for it to finish
        print("\nRunning actor")
        run = clientA.actor("natasha.lekh/gas-prices-scraper").call(run_input=run_input)
        dummyPrice = 0.00

        # Fetch and print actor results from the run's dataset (if there are any)
        print("\nFetching list")
        JSONdataName = clientA.dataset(run["defaultDatasetId"])
        print(JSONdataName)

        #create JSON bones for later use
        listObj = []

        for item in JSONdataName.iterate_items():
```

```
            try:
                print(priceVar[0]) #finds the first, almost always regular. Could specify which fuel type

                #append new data to listObj
                listObj.append({
                    "Title": item.get('title',''),
                    "Address": item.get('address',''),
                    "Price": priceVar[0]
                })

                print(listObj)

            except IndexError:
                print("Gas prices within API undefined.")
                listObj.append({
                    "Title": item.get('title',''),
                    "Address": item.get('address',''),
                    "Price": "Unavailable"
                })
```

Figure 6: Full implementation of apifygas.py is complete, and the data is prepared to be sent to the frontend side

3.4 Web Interface Development

This is the stage of the project that would first boot the Flask server and allow the user to access the web application from their computer. Along with the new run.py script that starts the server, this stage of development also necessitated the creation of several templates to be used by the web app. These are base.html, form.html, and data.html. Form.html is effectively the homepage that greets the user, and the data.html page is rendered with a set of variables to display once all calculations are complete.

```python
from flask import Flask
from flask import render_template, request
#access through running this test.py file and navigate to localhost:5000

# import libraries
from bs4 import BeautifulSoup
import requests
import re #regex
import math
import openrouteservice as ors
#import geopy
from geopy.geocoders import Nominatim

app = Flask(__name__)

@app.route('/form')
@app.route('/')
def form():
    return render_template('form.html')

@app.route('/data', methods = ['POST', 'GET'])
def data():
    if request.method == 'GET':
        return f"This is currently in GET mode. Navigate to /form to enter your data.\n"
    if request.method == 'POST':
        #this is where your python script goes. It runs when the form is posted.
        #form_data = request.form
        form_data1 = request.form.get('start')
        form_data2 = request.form.get('end')
        form_data3 = request.form.get('make')
        form_data4 = request.form.get('model')

```

Figure 7: Because Flask is a framework new to this point in the app's development, the project's code needed to be slightly restructured through the usage of @app.route()

As the pre-final version of the web app script, run2.py has all of the functionality necessary for the project to be considered "complete" but it is still missing the last few tweaks and adjustments to speed up the program where possible and increase data accuracy. This version still uses the coordinate-distance step system from early development.

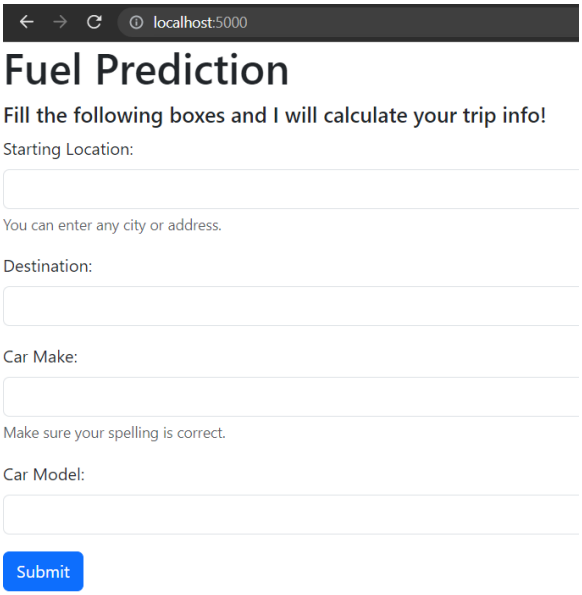


Figure 8: The form.html (or home) page. The user enters data and submits it. This data is sent to the Python script backend using a POST/GET request exchange.

Form submitted. You will be travelling:

from

Little Rock, Pulaski County, Arkansas, United States

...to...

Austin, Travis County, Texas, United States

driving a

cadillac srx

To get from your starting location to your destination with the car provided, you will be travelling 514.35 miles with an expected MPG of 19.75, and will be stopping for gas 1 time(s).

Gas Stations Near to Your Projected Stops:

- Stop #1:
- $3.36, Exxon, 3381 TX-276, Quinlan, TX 75474
- $3.33, Shell, TX-34, TX-276, Quinlan, TX 75474
- Unavailable, Brookshire's Fuel Center, 8934 TX-34, Quinlan, TX 75474
- Unavailable, 76, 900 Wolfe City Dr, Greenville, TX 75401
- $3.40, Shell, 1523 E Quinlan Pkwy, Quinlan, TX 75474
- Unavailable, Pritchett Oil LLC, 11400 TX-34, Quinlan, TX 75474
- Stop #2:
- Unavailable, Shop N Go, 110 E Crest Dr, Waco, TX 76705
- $2.95, Flying J Travel Center, 2409 S New Rd, Waco, TX 76711
- $3.10, Shell, 10400 Wortham Bend Rd, Waco, TX 76708
- $2.98, H-E-B Fuel, 1821 S Valley Mills Dr, Waco, TX 76711
- $3.40, Chevron Waco, 6218 Gholson Rd, Waco, TX 76705
- Unavailable, Corner Store, 6312 N Interstate 35 Frontage Rd, Waco, TX 76705

Figure 9: The data.html page. This is the page returned once all calculations are complete. Note the discrepancy between the anticipated stops for gas and the actual number of projected stops. This is a now fixed error in the fuel calculation algorithm.

The final version of the project code, run3.py, has all the same functionality of the previous version, but it also has an improved gas tracking algorithm. This new algorithm ditches the coordinate-distance-step matching system from before in favor of an algorithm that instead divides the entire list of coordinates by the number of calculated stops to get a key number. This key number indicates what set of coordinates should be returned to provide the stop locations. The key number is multiplied by whole numbers until it exceeds the length of the full set of polyline coordinates. It also runs slightly faster, as the new algorithm no longer requires the distance-step arrays to be made from the directions provided by the Openrouteservice API. It also fixes a previous bug in which the city to stop in could not properly be found in the reverse geolocator. This was caused by the Nominatim client actually being too specific and occasionally referring to cities as a "hamlet" or a "town" instead, which would not be detected by the regex equation looking for a "city."

Equation 5: key = int(coordsCount/(tankValueRaw + 1))

Equation 6: for x in range(tankValue): stopCoords = coordsFull[key * (x+1)] , *the new fuel tracking algorithm uses the "key" variable, which indicates how many coordinates you will visit before needing to stop for gas. "Key" is obtained by dividing the number of coordinates by the number of predicted stops, plus one.*

4. Testing and Metrics

For the sake of testing, the application will be run with three different road trips, and each of these trips will be tested with two different vehicles. The stretches are 1. Little Rock, AR to Fayetteville, AR, 2. Little Rock, AR to Lexington, VA, and 3. Austin, TX, to Phoenix, AZ. These trips will be tested using a Cadillac SRX and a Honda Accord. This ensures that the application is put through tests over various distances and stops, along with demonstrating the capability of switching vehicles.

Once the calculations are made and the results are displayed, each of the trips will be manually plugged into Google Maps so that the stop distances can be compared with one another and ensure that these stops line up with the projected travel distances. It is important to note that the "fuel range" of any car is not entirely precise. To be more specific, the miles per gallon displayed to the user and used for calculations is a total average gathered from many users over a period of time. However, because most of a road trip is driven on an interstate, this miles per gallon listing is slightly lower than what the user would likely experience. A small set of calculations found that there would be a 20-30% increase in MPG from average to highway. Another crucial piece of information to bear in mind before examining the data is that these MPG ratings are expecting the driver to obey traffic laws and avoid excessive speeding. The gas tank size is also hard-coded at 19 gallons, and is set to not deplete below the 15% fuel level.

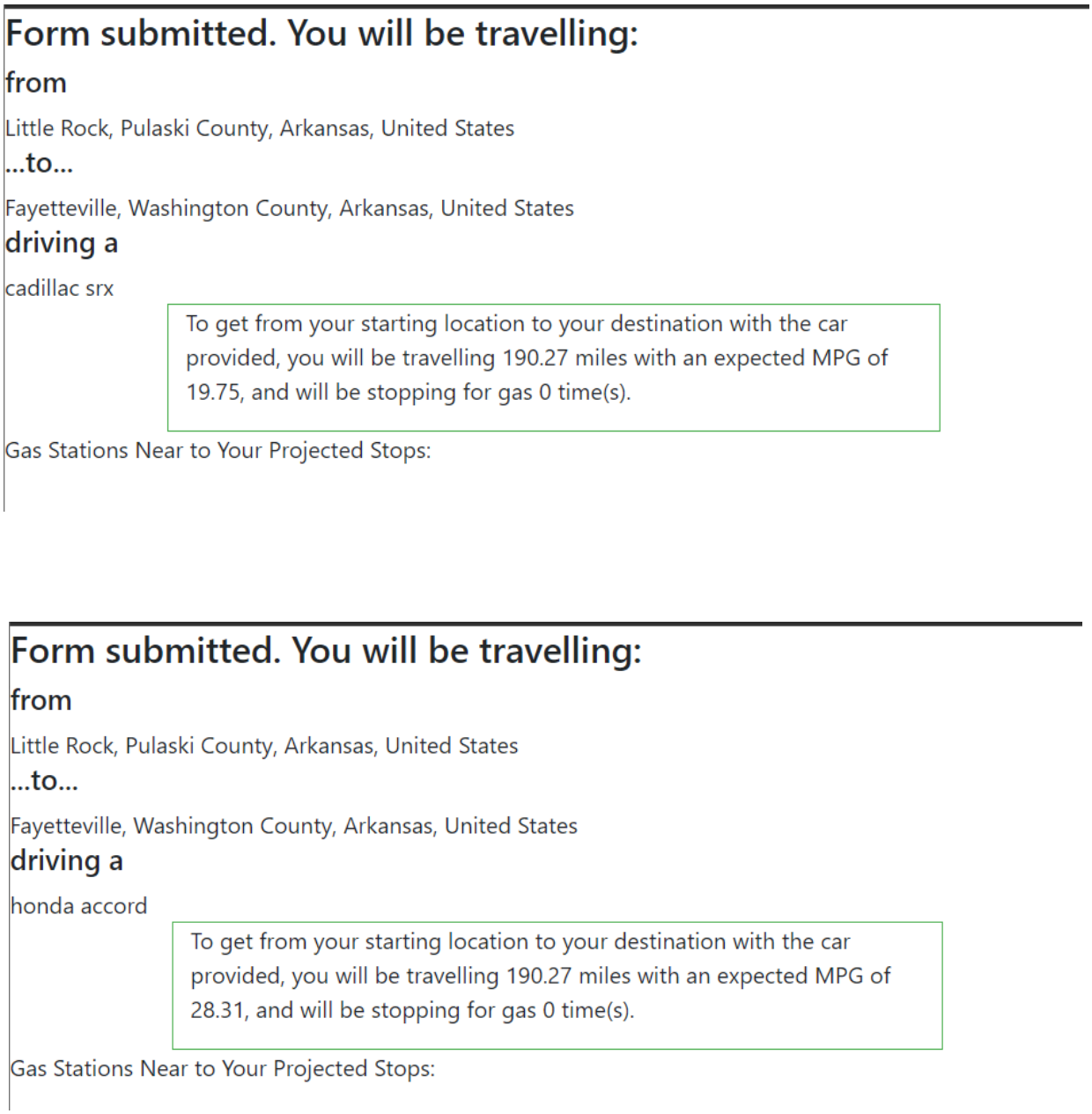


Figure 10: From Little Rock to Fayetteville, the program correctly identifies that neither car would require a refueling to complete the trip.

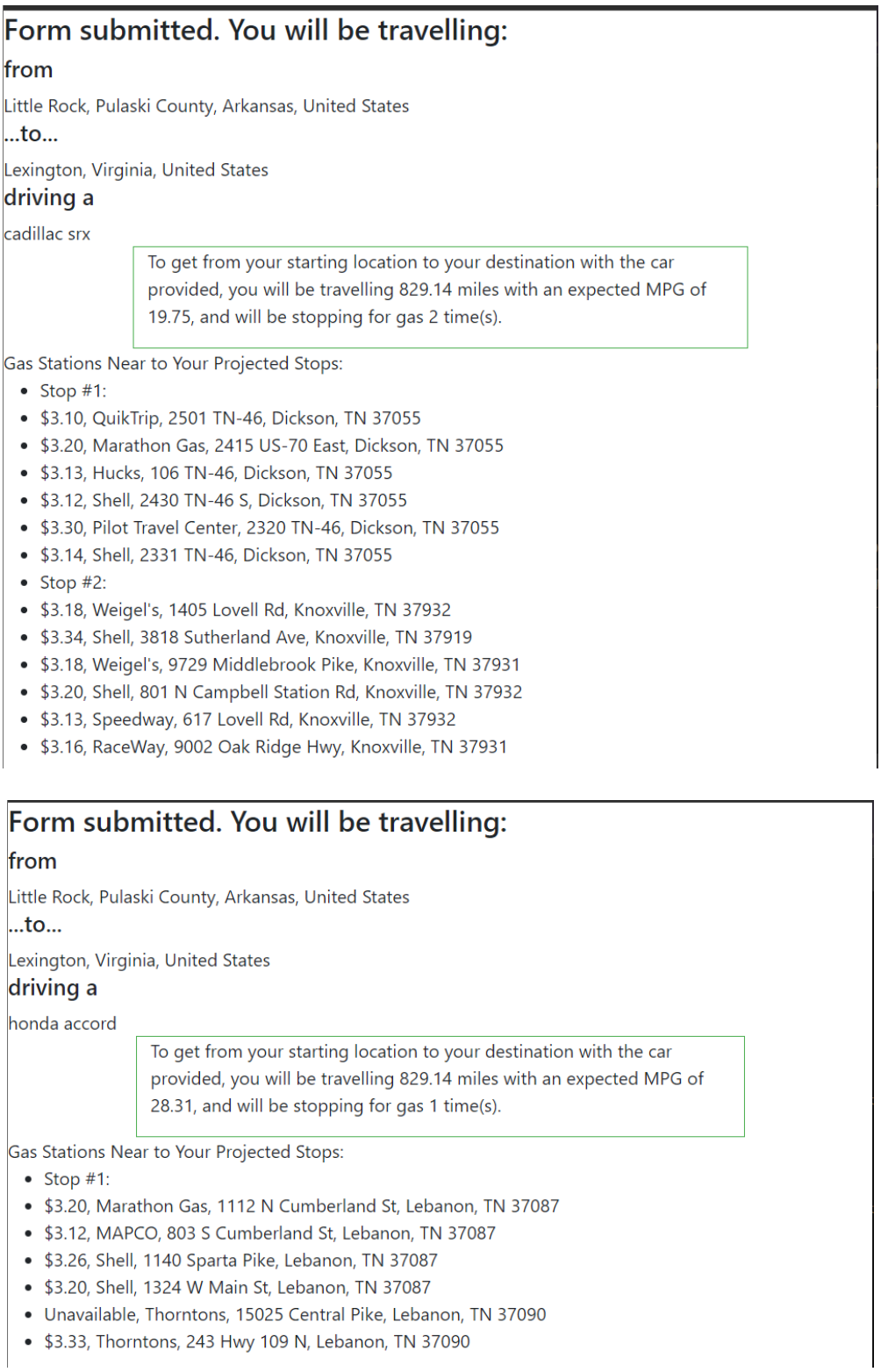


Figure 11: From Little Rock to Lexington, the SRX would need to stop twice and the Accord would need to stop once, due to having a higher MPG.

Form submitted. You will be travelling:

from

Austin, Travis County, Texas, United States

...to...

Phoenix, Maricopa County, Arizona, United States

driving a

cadillac srx

To get from your starting location to your destination with the car provided, you will be travelling 1005.91 miles with an expected MPG of 19.75, and will be stopping for gas 3 time(s).

Gas Stations Near to Your Projected Stops:

- Stop #1:
- Unavailable, Stage Coach, 5629 US-290, Fredericksburg, TX 78624
- Unavailable, Vp Racing Fuels Gas Station, 501 S Washington St, Fredericksburg, TX 78624
- $3.11, Sunoco Gas Station, 2204 HWY 16 South, Fredericksburg, TX 78624
- Unavailable, Texaco Fredericksburg, 701 E Main St, Fredericksburg, TX 78624
- Unavailable, Sinclair, 5244 N State Hwy 16, Fredericksburg, TX 78624
- $3.15, D AND DS, 11031 S State Hwy 16, Fredericksburg, TX 78624
- Stop #2:
- Unavailable, Kent Kwik Convenience Stores, 3301 W Dickinson Blvd, Fort Stockton, TX 79735
- $3.30, ALON, 701 E Dickinson Blvd, Fort Stockton, TX 79735
- Unavailable, Walmart Fuel Station, 2610 W Dickinson Blvd, Fort Stockton, TX 79735
- Unavailable, Uncle's Convenience Store/Gas - Valero, 1507 W Dickinson Blvd, Fort Stockton, TX 79735
- $3.30, Flying J Travel Center, 2571 N Front St, Fort Stockton, TX 79735
- $3.36, Love's Travel Stop, V4VW+VV, 2723 E US Hwy 290, Fort Stockton, TX 79735
- Stop #3:
- $3.60, Food Mart, 5701 N Jornada Rd, Las Cruces, NM 88012
- $3.50, Pilot Travel Center, 2681 W Amador Ave, Las Cruces, NM 88005
- $3.44, Circle K, 801 Thorpe Rd, Las Cruces, NM 88007
- $3.80, Circle K, 2601 Dona Ana Rd, Las Cruces, NM 88007
- Unavailable, Pic Quik (Drive-thru), 1501 E Amador Ave, Las Cruces, NM 88001
- Unavailable, Alon, 2601 Dona Ana Rd, Las Cruces, NM 88007

Form submitted. You will be travelling:

from

Austin, Travis County, Texas, United States

...to...

Phoenix, Maricopa County, Arizona, United States

driving a

honda accord

To get from your starting location to your destination with the car provided, you will be travelling 1005.91 miles with an expected MPG of 28.31, and will be stopping for gas 2 time(s).

Gas Stations Near to Your Projected Stops:

- Stop #1:
- $3.28, Phillips 66, 2350 N Main St, Junction, TX 76849
- $3.28, Chevron Junction, 2415 N Main St, Junction, TX 76849
- $3.28, Pilot, 2342 N Main St, Junction, TX 76849
- $3.28, Shell, 2416 N N Main St, Junction, TX 76849
- Unavailable, Les Williams Inc, 1928 Main St, Junction, TX 76849
- $3.36, Conoco, 1014 Main St, Junction, TX 76849
- Stop #2:
- Unavailable, Chevron Sierra Blanca, I-10 & FM 1111 SWC, Sierra Blanca, TX 79851
- Unavailable, Exxon, 316 E El Paso St, Sierra Blanca, TX 79851
- $3.90, Shell, 1422 Knox Ave, Fort Hancock, TX 79839
- $3.60, Valero, 316 US-80 E, Sierra Blanca, TX 79851
- Unavailable, Dell Valley Oil - Gas Station, 109 E Broadway St, Dell City, TX 79837
- Unavailable, TxDot Maintenance Facility, Ranch Rd 1111, Sierra Blanca, TX 79851

Figure 12: From Austin to Phoenix, the SRX would need to stop 3 times, and the Accord would only need to stop twice.

5. Evaluation

The goal of this project was to create an application that can take a starting location, a destination, and the information of a car and determine when the driver is expected to stop and follow up by recommending a few potential gas stations at that stop location. In order to evaluate the effectiveness of this program, the same trips will be calculated by hand using Google Maps and compared against the program. The distances between stops will be compared and examined to see if they are realistically achievable.

5.1 Final Model

As shown above, the final model is proven to work reliably and effectively for the listed trips and different cars. Theoretically, this model should work well with any car that has enough statistical data stored on the Fuelly.com website. It also works with any address or location that is contained by the OpenStreetMap API, which is utilized by Openrouteservice to determine routes between locations [5]. The only limitation to this is with extremely long distances, for example from Los Angeles, CA to New York City, NY. It is at distances such as these that the route planning service will reach its max nodes available and cannot plot the route.

5.2 Results

To illustrate which stops the application selects, the routes have been plotted using Google Maps, with each of the vehicle's respective stops circled, and the distances recorded. The Cadillac SRX has stops circled in red, and the Honda Accord has stops circled in yellow.

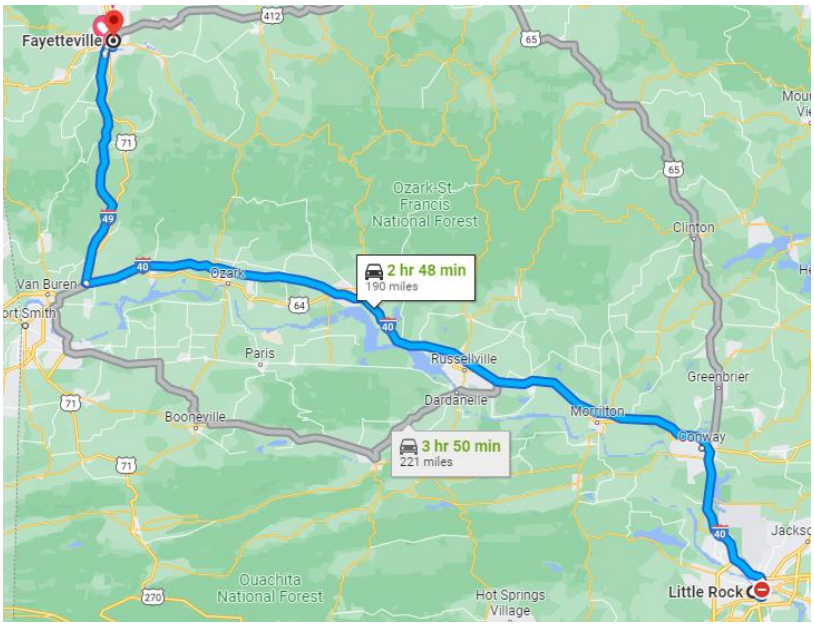


Figure 13: Little Rock to Fayetteville is 190 miles (program reports 190.27), therefore no stops would be necessary.

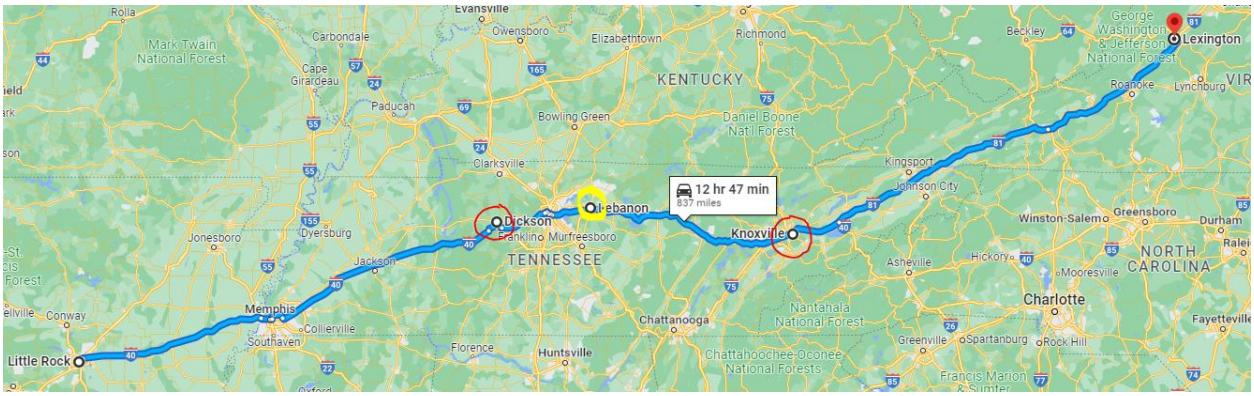


Figure 14: Little Rock to Lexington is 837 miles (program reports 829.14), which requires stops in Dickson and Knoxville for the SRX, and requires a single stop in Lebanon for the Accord.

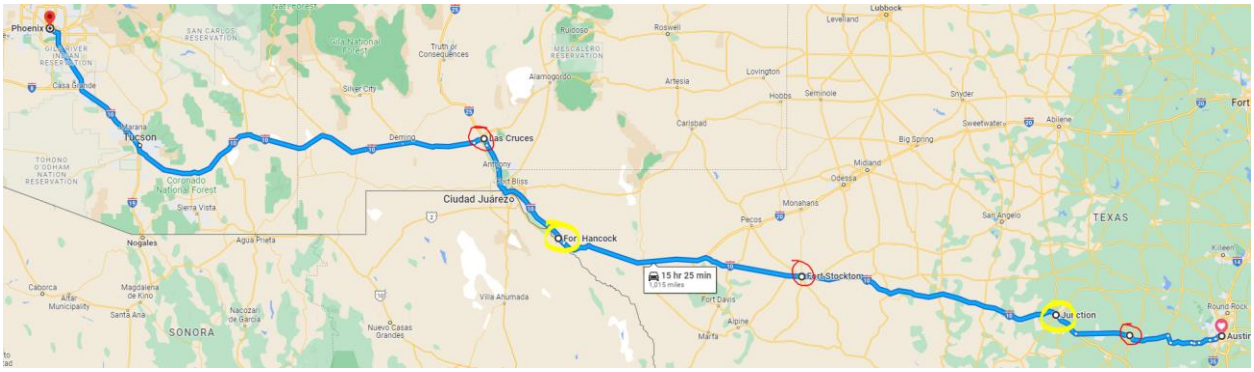


Figure 15: Austin to Phoenix is 1015 miles (program reports 1005.91), which requires stops in Fredericksburg, Fort Stockton, and Las Cruces for the SRX and requires stops in Junction and Fort Hancock for the Accord.

Cadillac SRX				Honda Accord		
	Total Distance	From Previous Stop			Total Distance	From Previous Stop
Dickson	312	312		Lebanon	380	380
Knoxville	531	219		N/A		
Lexington	837	306		Lexington	837	457
Fredericksburg	75	75		Junction	136	136
Fort Stockton	331	256		Fort Hancock	519	383
Las Cruces	625	294		N/A		
Phoenix	1015	390		Phoenix	1015	496

Figure 16: Excel chart depicting each vehicle's projected stops and distances between each stop.

As shown in the above chart, the Accord can travel much further between refuelings due to its better gas mileage, which is reflected in the "From Previous Stop" category. The longest stretch made without stopping to refuel is from Fort Hancock, TX to Phoenix, AZ at 496 miles. Noticeably, during the SRX's trip to Phoenix the application clearly prioritizes having a final stretch as long as possible. There is a similar trend seen in all of the stops decided by the refueling algorithm. While the Accord is realistically capable of driving long distances before needing to stop for gas, these numbers are slightly inflated. This is due to the application's gas

consumption algorithm assuming a gas tank size that is in fact larger than the Accord's actual tank size.

6. Discussion

6.1 Conclusions

The purpose of this project was to build a web application that is capable of taking in a start and a destination as well as the make and model for the car, and plot the route between those two locations while accommodating for the necessary fuel stops for the given car. This was done through the use of the Openrouteservice API, the Gas Prices Scraper API, and the reverse geocoding Nominatim. The web app, hosted on a Flask server, prompts the user for the previously mentioned inputs, sends the data through a POST/GET method exchange, calls the necessary API's, and returns a rendered HTML template with the calculated stops displayed to the user. The algorithm used by the application is found to work best used with mid-size to SUV-sized vehicles.

6.2 Future Work

Throughout the process of writing this report and deciding this program there are many areas in which the application can be taken to further improve on either its functionality or efficiency and accuracy. The immediately most noticeable of which is that the program simply uses a catch-all gas tank size when calculating potential fuel range. Part of the reason this was used was to accommodate individuals wanting to stop more often than completely necessary, but if someone were to run their tank completely empty, it would likely not line up correctly with the

predicted stop locations. A solution to this is to allow the user to input the car's tank size on their own and also input their starting fuel level. This would allow for a much more realistic scenario as the cars expected fuel range would be much closer to what the individual would likely experience, and it prevents them from having to completely fill up their tank before actually starting the trip for the calculations to be accurate.

The next, much more intensive improvement would be to allow for another gas station searching algorithm to be used. This algorithm could search along the desired route for major cities and towns and pick one or two gas stations from each one reporting those back to the user. The program could then compare these returned gas prices and through a much more in-depth set of calculations provide the exact cheapest trip possible with a given number of refueling stops. However, an algorithm like this is not practical with the current parameters and API's being used by the program. Currently one of the biggest issues with the Gas Prices Scraper API being used is that for modern-day technology it is incredibly slow. Retrieving six gas stations from a single county takes up to an entire minute. If the same API were to be used with every city along the given route, the program could likely take anywhere from 5 minutes to 30 minutes to run. It is likely that the source of this slowdown comes from the way the gas station API is written or from the hosting site, Apify. An alternative would be to use Google's own landmark listing API, Google Places, which would remove the need for the gas station specific API and would likely result in a much faster run time. This faster run time would allow for the use of the previously mentioned algorithm that would search every major location along the given route.

The final two improvements would be much smaller changes, but would result in a much higher quality of life experience for the user. The reverse geocoder being used for this program

often has timeout errors, which is in no part on behalf of this program but instead on the geolocator being used.

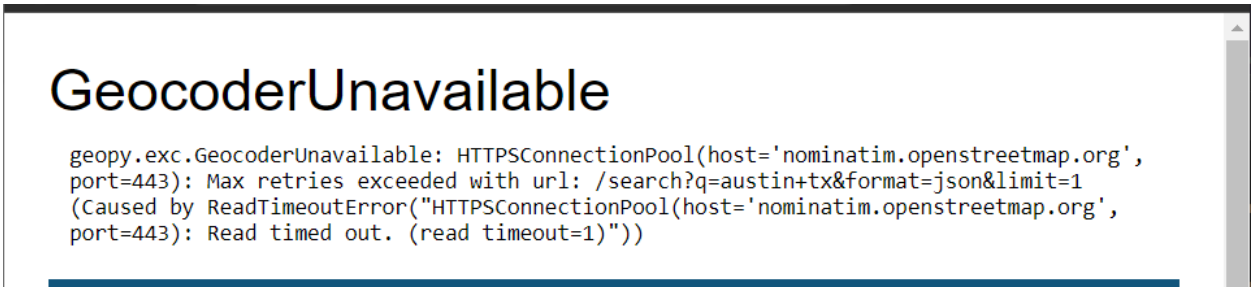


Figure 17: Nominatim timing out during runtime.

A possible improvement on this would be to simply use another geocoder that is more consistent with its availability. Lastly, it would benefit the user greatly if the program was adjusted to omit gas stations with "unavailable" prices. As these gas stations with unlisted prices are likely lower quality than those that do, most users would prefer to ignore these options regardless.

7. References

[1] Adam and Andrzej, “What Is The Algorithm Used by ORS API for QGIS,” *Openrouteservice*, Feb-2021. [Online]. Available: https://ask.openrouteservice.org/t/what-is-the-algorithm-used-by-ors-api-for-qgis/2476/2. [Accessed: 25-Apr-2023].

[2] D. Rachmawati and L. Gustin, “Analysis of Dijkstra’s Algorithm and A* Algorithm in Shortest Path Problem,” *Journal of Physics: Conference Series*, vol. 1566, no. 1, p. 012061, 2020.

[3] “Flask,” *Welcome to Flask - Flask Documentation (2.3.x)*, Apr-2023. [Online]. Available: https://flask.palletsprojects.com/en/2.3.x/. [Accessed: 25-Apr-2023].

[4] “Fuelly - Track and Compare Your MPG,” *Fuelly*, 2023. [Online]. Available: https://www.fuelly.com/car. [Accessed: 25-Apr-2023].

[5] heiGIT gGmbH, *Openrouteservice*, 2022. [Online]. Available: https://openrouteservice.org/. [Accessed: 25-Apr-2023].

[6] J. Dibbelt, B. Strasser, and D. Wagner, “Customizable Contraction Hierarchies,” *ACM Journal of Experimental Algorithmics*, vol. 21, no. 1.5, pp. 1–49, Apr. 2016.

[7] N. Lekh, “Gas Prices Scraper,” *Apify*, 01-Jan-2023. [Online]. Available: https://apify.com/natasha.lekh/gas-prices-scraper. [Accessed: 25-Apr-2023].

[8] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner, “Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm,” *ACM Journal of Experimental Algorithmics*, vol. 15, no. 2.3, pp. 2.1–2.31, Mar. 2010.