Path Planning for Supercharger Network

ECEN - 779 Motion Planning

Anthony Angeles
Santa Clara University

February 13, 2026

1 Introduction

For this assignment, we were tasked with implementing a search algorithm to find the minimum time path through a network of supercharging stations. There were a few details that were given to us to begin the problem.

1.1 Key Assumptions

In the project guideline, we were given that each supercharger will refuel the vehicle at a different rate given in km/hr of charge time. We were also given that our route does not have to fully charge at every visited charger, so long as it never runs out of charge between two chargers.

In the project we were also given the liberty to make these assumptions throughout our project:

1. The car begins at the start charger with a full charge of 320 km.
2. The car travels at a constant speed of 105 km/hr along great circle routes between chargers.
3. The Earth is a sphere of radius 6356.752 km.
4. We may look up formulas and reference external material.

The references used throughout this project include resources on the Haversine formula, A* with Landmarks (ALT), and class lectures:

- Haversine Formula – Wikipedia: `https://en.wikipedia.org/wiki/Haversine_formula`
- Haversine Formula – Rosetta Code (Python): `https://rosettacode.org/wiki/Haversine_formula#Python`
- A* with ALT Heuristic – Georgia Tech: `https://faculty.cc.gatech.edu/~thad/6601-gradAI-fall2014/02-search-01-Astart-ALT-Reach.pdf`
- ECEN-779 Class Lecture: `https://santaclarauniversity.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=b1249009-5673-428b-9e3f-b3e20184f055`

2 The Code

2.1 My First Iterations

My initial thoughts were to simply try to brute force the solution as described in the hints of the project, from there we could then optimize. In my brute force solution, I essentially enumerated through all possible paths, using a version of recursive DFS search to find valid paths from start to end, from this we could then find the total time (driving + charging) and return the best path.

This solution worked, but it is also highly inefficient and slow to get the solution. The reason for this is because we are literally exploring every possible path before we decide on the best choice. As our network grows our runtime will balloon quite easily. Realistically speaking this would be okay at best for small networks that do not take much to solve for the best path.

2.2 How I Improved

The key to arriving at a better solution is the A* search algorithm, where instead of exploring all paths, I implement informed searching to prioritize the ideal paths first. I iterated through different versions of A* as well, before landing on my last implementation of ATL.

Also instead of simply just tracking the stations we visited, in our new implementation we also track the charge level in addition to the station: `state = (station_idx, charge_level)`

What this essentially allows us to do is not just keep track of the *where* but also the *when to charge*. Another difference when using A* is the use of heuristics. By adding heuristics we can use estimated time remaining to help push the search in the direction towards the goal.

The charge is put into levels spaced 2 km apart (controlled by the `charge_distance` variable). This means instead of tracking exact charge as a continuous number, the algorithm rounds it into bins. With a 320 km battery and 2 km spacing, there are 161 possible charge levels. This makes the search space manageable while still being precise enough to find near-optimal solutions.

At each step, the algorithm considers two options for every reachable neighbor:

1. **Drive directly** (handled by the `directDrive` function): If the car has enough charge to reach the neighbor without any additional charging, it simply drives there. The arrival charge level is computed, and the total time is updated with just the driving time.

```
    def directDrive(node, charge_level, curr_charge, curr_time,
                    neighbor, dist, drive_time, best_time, parent_info,
                    heuristic_time, heap, numlvls):
        if curr_charge >= dist:
            arrival_charge = curr_charge - dist
            arrival_level = min(int(arrival_charge / charge_distance), numlvls
                - 1)
            new_time = curr_time + drive_time

        ...
```

2. **Charge first, then drive** (handled by the `chargeToDrive` function): The car charges at the current station for just long enough to reach the neighbor with a specific arrival charge level. The function loops over all possible arrival charge levels and calculates how much charging time each one would need. This lets the algorithm consider partial charges—it does not have to fill up completely before leaving.

```
    def chargeToDrive(node, charge_level, curr_charge, curr_time,
                      neighbor, dist, drive_time, charge_rate,
                      best_time, parent_info, heuristic_time, heap, numlvls)
                          :
        if curr_charge >= dist:
            min_arrival_level = int((curr_charge - dist) / charge_distance) +
                1
        else:
            min_arrival_level = 0

        max_arrival_charge = full_charge - dist
        max_arrival_level = min(numlvls - 1, int(max_arrival_charge /
            charge_distance))

        for arrival_level in range(min_arrival_level, max_arrival_level + 1):
```

```
            target_arrival_charge = arrival_level * charge_distance
            charge_needed = target_arrival_charge + dist - current_charge

        ...
```

At each station, the algorithm estimates the remaining time to the destination by computing the straight-line distance and dividing by the driving speed. Since the straight-line distance is always less than or equal to the actual driving distance, this estimate never overestimates the true remaining time, which is the key property that guarantees A* finds the optimal path. A priority queue keeps states sorted so the most promising one is always explored next. When the algorithm reaches the destination, it traces back through stored parent pointers. Each recording the previous station, charge level, and charge duration—to reconstruct and print the full route.

If I had to attribute the one change that I made to my solution when going from brute force to A*, it would be the tracking of the state as previously mentioned. Which allows us to terminate immediately for sub optimal paths.

2.3 Further Optimizations

Beyond the core A* implementation, I made additional changes as I wanted to see how much we could improve really, some worked other methods didn't.

Charge Granularity Tuning. The `charge_dist` parameter controls the spacing between discrete charge levels. We tested values of 8.0, 5.0, 4.0, and 2.0 km. Coarser values (e.g., 8.0) resulted in only 41 charge levels and produced noticeably suboptimal paths. This could be seen for example where Chicago → Cadillac route took 3.7474 hours with 8.0 spacing versus 3.3751 hours with 2.0. The finer 2.0 km spacing which is 160 levels per node, was enough resolution for near-continuous charge decisions while keeping the state space manageable for A*.

ALT Heuristic (A* with Landmarks and Triangle Inequality). The baseline heuristic uses straight-line distance to the goal divided by driving speed. While admissible, this worked but after looking into other versions I came across ATL. With an ALT heuristic we took eight landmark nodes strategically selected: the four geographic extremes, the geographic center, and three additional points chosen by farthest-point sampling:

```python
def landmarks(coords, dist, k=8):
    n = len(coords)
    lat = [coord[0] for coord in coords]
    lon = [coord[1] for coord in coords]
    landmarks = []
    # 1. Four corners (N, S, E, W extremes)
    cardinal = [
        lat.index(max(lat)), lat.index(min(lat)),
        lon.index(max(lon)), lon.index(min(lon))
    ]
    landmarks.extend(list(dict.fromkeys(cardinal)))
    # 2. Geographic center point
    ...
    # 3. Farthest point sampling for remaining slots
    while len(landmarks) < k:
        maxMinDist = -1
        farthest = -1
        for node in range(n):
            if node in landmarks:
                continue
            minDist = min(dist[node][lm] for lm in landmarks)
            if minDist > maxMinDist:
                maxMinDist = minDist
                farthest = node
```

```
        if farthest != -1:
            landmarks.append(farthest)
    return landmarks
```

At search time, the heuristic for each node takes the maximum of the straight-line estimate and the landmark-based lower bound using the triangle inequality:

```
def heuristic(currIndex, goalIndex, distance):
    maxLower = 0.0
    for distanceToLandmark in distance:
        lowBound = abs(distanceToLandmark[currIndex]
                       - distanceToLandmark[goalIndex])
        maxLower = max(maxLower, lowBound)
    return maxLower
```

The two heuristics are then combined during search setup, always taking the tighter bound:

```
h_dist = []
for i in range(n):
    h_straight = dist[i][end_idx]
    h_landmark = heuristic(i, end_idx, distLandmark)
    h_dist.append(max(h_straight, h_landmark))
```

Early Termination. Once a valid path to the destination is found, any state whose current time plus heuristic exceeds the best known cost is dropped.

```
if best[end_idx][0] != float('inf'):
    if t + h_time[u] >= best[end_idx][0]:
        continue
```

Heuristic Weight. We initially used weighted A* with a heuristic multiplier of 1.5 to speed up the search. However, testing revealed this caused issues on certain routes (e.g., Chicago → Cadillac: 3.7474 hrs vs. 3.3751 hrs with standard A*). Reverting to weight 1.0 restored optimal results on all test cases.

2.4 What Did Not Help

Not every idea improved the solution. **caching** was considered but I decided that wouldn't do much for my cause at the end. **Bidirectional A*** was too complex for what would have been small gains. **Charge granularity** when we changed `charge_dist` ultimately a fixed 2.0 km spacing was best. Finally, having k=4 wasn't as efficient as k=8 confirming k=8 as the sweet spot.

3 Results

We validated the script on several different start and end charger pairs, then verified each result against the provided reference checker. Below are the terminal outputs showing the solution paths and checker results.

3.1 Route Outputs

Council_Bluffs_IA → Cadillac_MI:

```
$ python3 solution.py "Council_Bluffs_IA" "Cadillac_MI"
Council_Bluffs_IA, Worthington_MN, 2.49734, Albert_Lea_MN, 0.38119, Onalaska_WI,
    0.69868, Mauston_WI, 2.31389, Sheboygan_WI, 0.53554, Cadillac_MI
```

Albany_NY → Chicago_IL:

```
$ python3 solution.py "Albany_NY" "Chicago_IL"
Albany_NY, Liverpool_NY, 0.59374, Buffalo_NY, 2.18321, Erie_PA, 1.05864, Maumee_OH
    , 2.25044, Angola_IN, 0.09717, Chicago_IL
```

San_Diego_CA → Bellevue_WA:

```
$ python3 solution.py "San_Diego_CA" "Bellevue_WA"
San_Diego_CA, San_Juan_Capistrano_CA, 0.00601, Lebec_CA, 1.56346, Buttonwillow_CA,
    0.36866, Coalinga_CA, 0.76639, Folsom_CA, 1.87715, Roseville_CA, 0.10925,
   Corning_CA, 1.13320, Mt._Shasta_CA, 1.43072, Bend_OR, 1.71540, The_Dalles_OR,
   0.51135, Bellevue_WA
```

Queens_NY → Redondo_Beach_CA:

```
$ python3 solution.py "Queens_NY" "Redondo_Beach_CA"
Queens_NY, Allentown_PA, 0.80936, Harrisburg_PA, 0.80350, Somerset_PA, 1.45620,
   Triadelphia_WV, 0.17575, Grove_City_OH, 2.06039, Dayton_OH, 0.36816,
   Terre_Haute_IN, 2.18552, Effingham_IL, 0.84338, Columbia_MO, 2.77570, Topeka_KS
   , 1.32287, Salina_KS, 1.80150, Hays_KS, 0.93861, Goodland_KS, 1.17998,
   Lone_Tree_CO, 1.69387, Silverthorne_CO, 0.63113, Glenwood_Springs_CO, 0.30698,
   Green_River_UT, 1.16074, Richfield_UT, 1.29038, St._George_UT, 1.74767,
   Primm_NV, 0.76091, Barstow_CA, 1.20930, Hawthorne_CA, 0.02878, Redondo_Beach_CA
```

3.2 Checker Verification

Each route was verified using the provided `checker_linux` binary.

```
$ ./checker_linux 'Council_Bluffs_IA, Worthington_MN, ..., Cadillac_MI'
Finding Path Between Council_Bluffs_IA and Cadillac_MI
Reference result: Success, cost was 17.2531
Candidate result: Success, cost was 16.8597

$ ./checker_linux 'Albany_NY, Liverpool_NY, ..., Chicago_IL'
Finding Path Between Albany_NY and Chicago_IL
Reference result: Success, cost was 17.5703
Candidate result: Success, cost was 17.4835

$ ./checker_linux 'San_Diego_CA, San_Juan_Capistrano_CA, ..., Bellevue_WA'
Finding Path Between San_Diego_CA and Bellevue_WA
Reference result: Success, cost was 26.8863
Candidate result: Success, cost was 26.4978

$ ./checker_linux 'Queens_NY, Allentown_PA, ..., Redondo_Beach_CA'
Finding Path Between Queens_NY and Redondo_Beach_CA
Reference result: Success, cost was 66.3384
Candidate result: Success, cost was 64.6781
```

3.3 Summary

All four routes returned "Success" from the checker, confirming that the car never ran out of charge at any point. The code for this assignment can be found here: `https://github.com/aneangel/Supercharger-path-planning`

4 Key Takeaways

I learned a lot about path planning in this project. **charge granularity** (2.0 km) was essential for finding optimal charging decisions, and you can't assume increasing the value will yield better results. **Heuristics**: the ALT heuristic provided meaningfully improvement than straight-line distance alone, directly reducing the number of states explored. **Standard A* outperformed weighted A***, I totally thought that having the weight times 1.5 would have improved it more than not including it, but I was proven wrong and the standard weight did better. Also doing these changes bit by bit and then testing it with the binary was helpful to see how the cost changed.