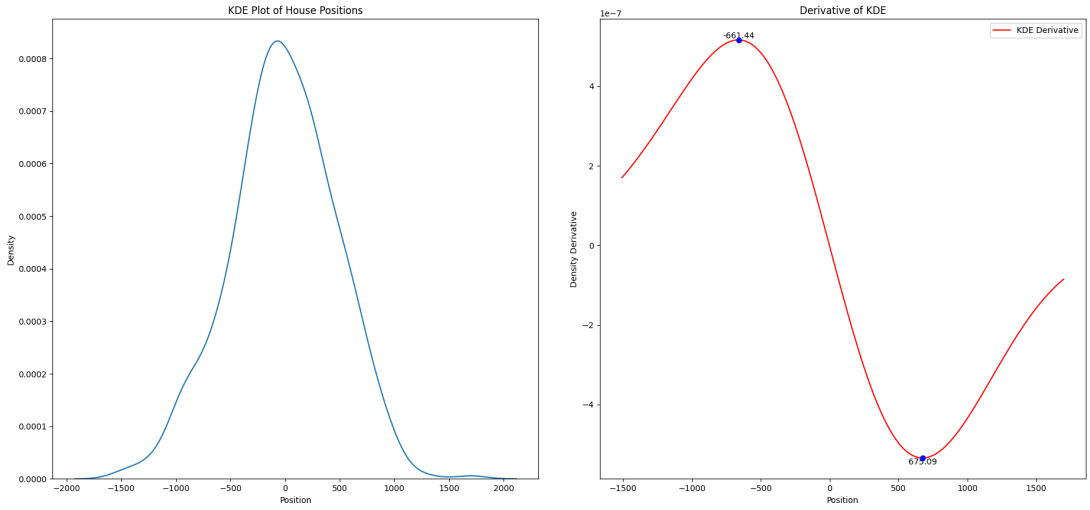
ALGORITHMIC COMPLEXITY REPORT

Written by: Jonathan Layduhur

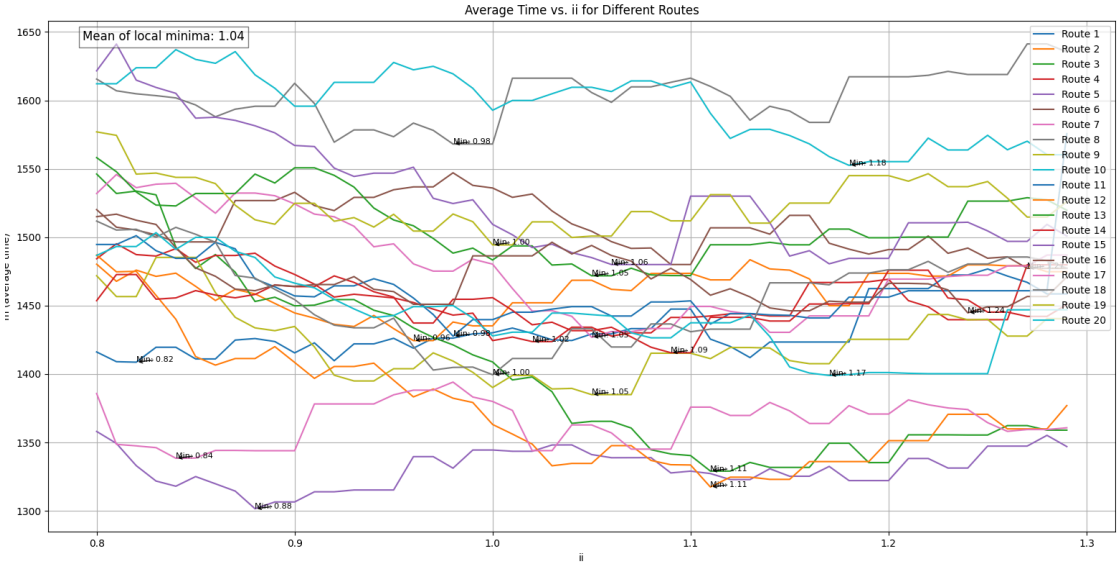
1 PART 1 1

1.1

By observing the samples, we can see that, due to being derived from a normal distribution, the houses are most likely to be around 0. To minimize the overall time, it would be better to clear the two parts around 0, the denser areas, before addressing the outliers. I decided to use Kernel Density Estimation (KDE) to get a better idea of where the houses are located in a specific sample. Then, by calculating the derivative of the KDE, I can identify where there is an "acceleration" or "drop" in density. This helps to determine where to make the "first stop" and then proceed to complete the other dense parts of the sample.

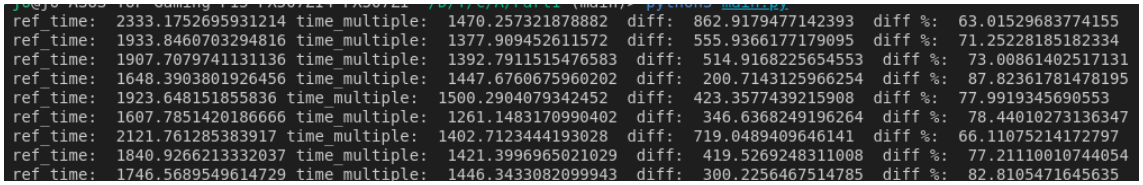


From that we iterate on an offset to have a better result.



in the function “refference()” we have the benchmark algorith, and observe that the new algorith time is 60% -

80% of the gredy algorith.



1.2 Polynomial Time Complexity Analysis 2

1. Function: calculate_average_time

- Insertion at the beginning: Inserting an element at the beginning of the list is O(n).
- For loop: Iterates nnn times, where nnn is the length of houses, contributing O(n).
- Appending to list: Each append operation is O(1), leading to a total of O(n).
- Mean calculation: Calculating the mean is O(n).

Overall Time Complexity: O(n)

2. Function: get_kde

- KDE Initialization: Initializing the KDE is typically O(n log n).
- Range determination: Finding the minimum and maximum values in the list is O(n).
- Array creation: Creating a linearly spaced array is O(n.n_points).
- KDE Evaluation: Evaluating the KDE at each of the n_points points is O(n·n_points).
- Gradient Calculation: Calculating the gradient of the KDE values is O(n_points).
- Finding Min/Max Derivatives: Finding the minimum and maximum values of the derivative is O(n_points).

n_points is proportional to n, the overall complexity is O(n^2).

3. Function: from_step_to_route

- Finding initial index: Finding the index of the minimum absolute value in the list is O(n).
- Outer for loop: The loop runs k times, where k is the length of steps.
- Finding closest index: Finding the closest index in the list is O(n).
- While loop and house deletion: Each deletion operation is O(n), and the loop can run O(n) times per step, making this part O(n2).

Overall Time Complexity: O(n^2)

4. Function: mutiple_index_route

- ii_values generation: Generating the range of values is O(1) since it has a constant number of elements.
- Outer for loop:The loop runs a constant number of times (100).
  - Function calls within loop: Each call to make_simple_route_derivitive and calculate_average_timeis polynomial.
    - make_simple_route_derivitive includes get_kde O(n^2) and from_step_to_route O(n^2).
    - calculate_average_time is O(n).

Since the loop runs a constant number of times, the total complexity is dominated by the polynomial time of the inner functions.

Overall Time Complexity: O(n^2)

5. Conclusion

All provided functions run in polynomial time:

- calculate_average_time: O(n)
- get_kde: O(n^2)
- from_step_to_route: O(n^2)
- mutiple_index_route: O(n^2)

Thus, the combined complexity of the main function mutiple_index_route remains polynomial, specifically O(n^2)

3 Part 3 **3**

Analyzing function_1()

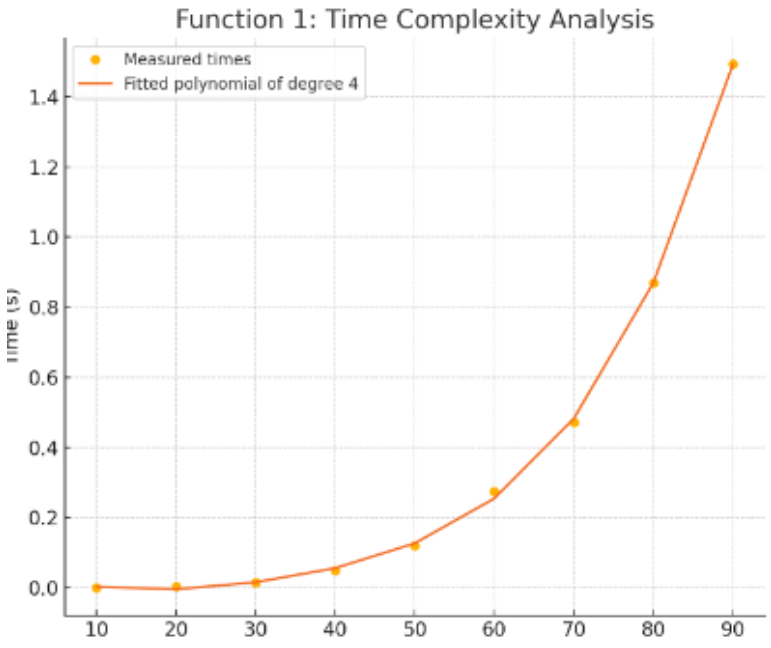
Complexity Analysis:

1. The outer loop runs n^2 times.
2. The inner loop runs itimes for each iteration of the outer loop.
   - When i=0, the inner loop runs 0 times.
   - When i=1, the inner loop runs 1 time.
   - When i=2, the inner loop runs 2 times.
   - And so on, up to i=n^2 - 1.

The total number of iterations of the inner loop is:

$$\sum_{i=0}^{n^2-1} i = \frac{(n^2-1)n^2}{2} \approx \frac{n^4}{2}$$

Therefore, the time complexity of function_1 is O(n^4).



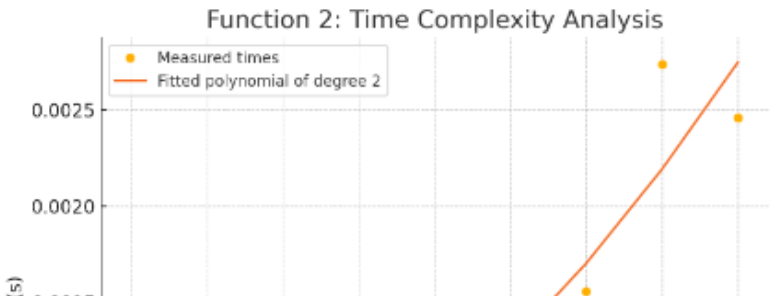
Analyzing function_2()

Complexity Analysis:

1. The outer loop runs n times.
2. Creating temp_list involves a list comprehension, which runs in O(n).
3. Shuffling the list temp_list runs in O(n).
4. Finding the maximum element in temp_list runs in O(n).

Therefore, each iteration of the outer loop runs in O(n)+O(n)+O(n)=O(n).

The total time complexity is n×O(n)=O(n^2).



Analysis of Plots:

- **Function 1:** The measured times fit well with a polynomial of degree 4, confirming our theoretical complexity of O(n^4).
- **Function 2:** The measured times fit well with a polynomial of degree 2, confirming our theoretical complexity of O(n^2).

2 Part 2 4

1) Heuristics

We will choose two degree-based heuristics for matching:

1. **Greedy Matching Based on Degree:**
   - Sort nodes by their degrees in descending order.
   - Iteratively select the highest-degree node and match it with one of its neighbors that has the highest degree.
2. **Random Matching:**
   - Randomly shuffle the nodes.
   - Iteratively select the first unmatched node and match it with one of its available neighbors.

We will implement these heuristics in Python using the networkx library.

2) Verification of matchings

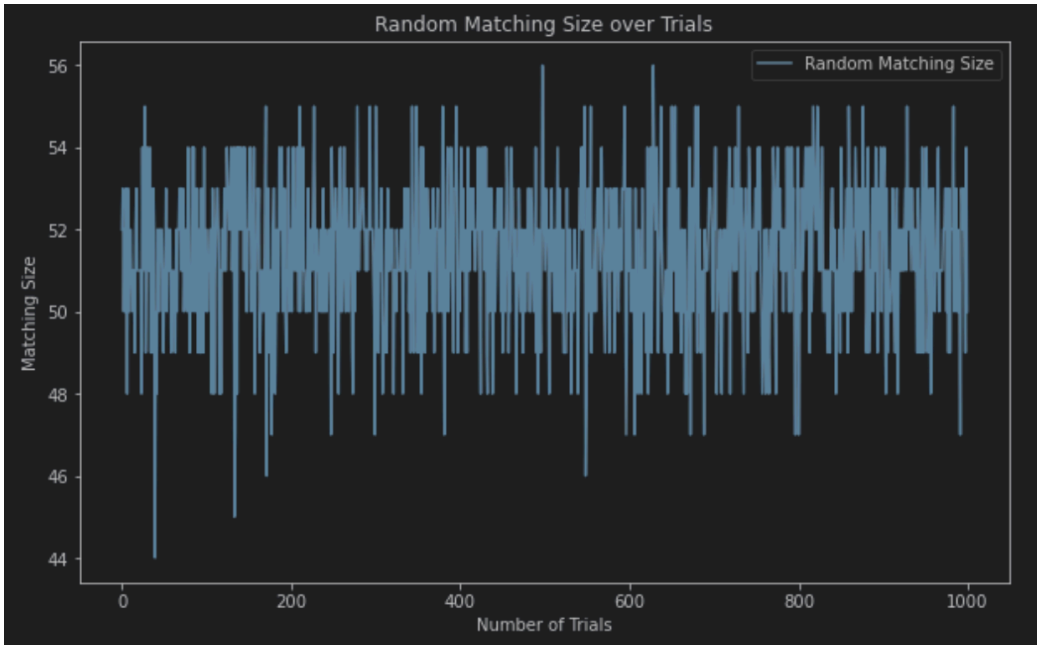
We will create a test file test_matching.py to verify that the matchings are correct.

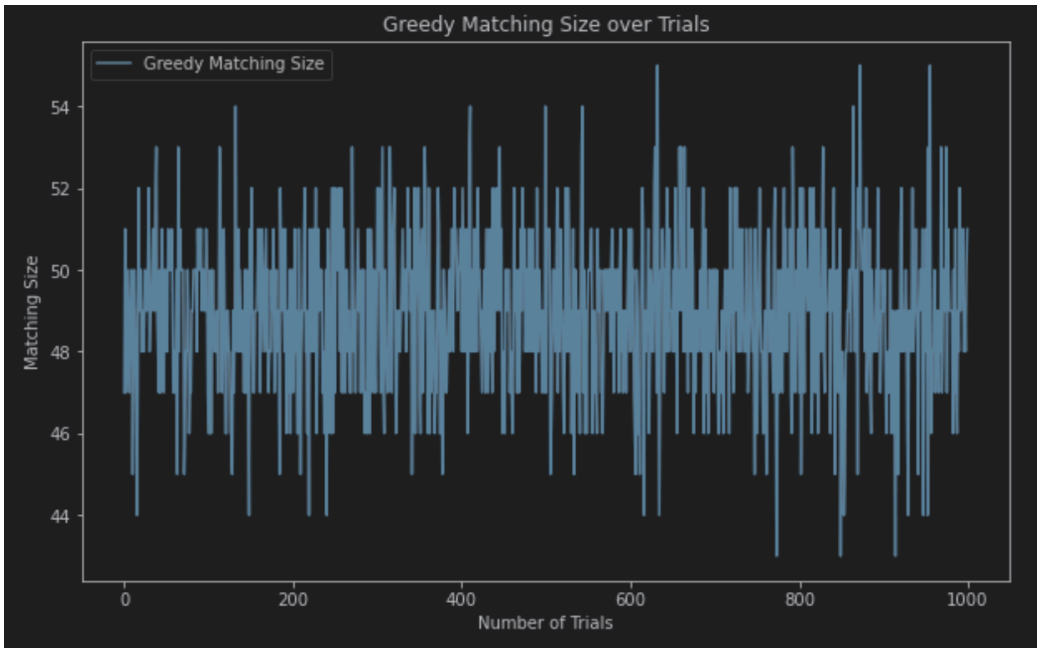
```python
import networkx as nx
from your_module import greedy_matching_based_on_degree,
random_matching
def test_matching(graph, matching):
    matched_nodes = set()
    for u, v in matching:
        assert u in graph
        assert v in graph
        assert (u, v) in graph.edges or (v, u) in graph.edges
        assert u not in matched_nodes
        assert v not in matched_nodes
        matched_nodes.add(u)
        matched_nodes.add(v)
    print("All matchings are correct")
```

3) Comparison of heuristics

We will statistically compare the two heuristics based on:

- The size of the returned matching.
- The computation time.





`3) Polynomial time proof`

Both heuristics must be proven to run in polynomial time. We analyze their complexities as follows:

- **Greedy Matching Based on Degree:**
  - Sorting nodes by degree: O(nlogn)O(n \log n)O(nlogn)
  - Iterating over nodes and their neighbors: O(n+m)O(n + m)O(n+m), where nnn is the number of nodes and mmm is the number of edges.
  - Overall complexity: O(nlogn+n+m)=O(nlogn+m)O(n \log n + n + m) = O(n \log n + m)O(nlogn+n+m)=O(nlogn+m)
- **Random Matching:**
  - Shuffling nodes: O(n)O(n)O(n)
  - Iterating over nodes and their neighbors: O(n+m)O(n + m)O(n+m)
  - Overall complexity: O(n+m)O(n + m)O(n+m)

An Eulerian path is a path in a graph that visits every edge exactly once. The key conditions for a graph to have an Eulerian path are:

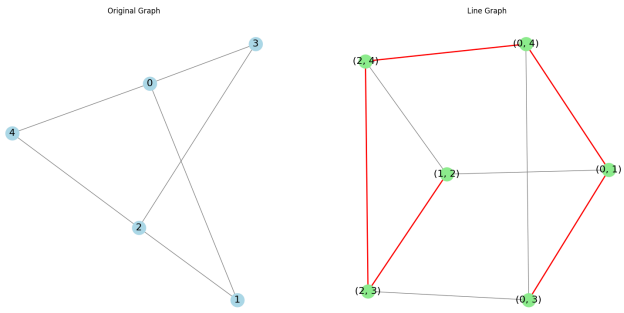
- **Eulerian Circuit**: The graph has an Eulerian circuit if and only if every vertex has an even degree.
- **Eulerian Path**: The graph has an Eulerian path (but not necessarily an Eulerian circuit) if and only if exactly two vertices have an odd degree.

```python
def analyze_eulerian_path(G : nx.Graph):
    degrees = {node: val for (node, val) in G.degree()}
    even_degree_count = sum(1 for degree in degrees.values() if degree % 2 == 0)
    odd_degree_count = sum(1 for degree in degrees.values() if degree % 2 != 0)
    eulerian_path_exists = (odd_degree_count == 2 or odd_degree_count == 0)
    if even_degree_count == len(G.nodes):
        print("Every vertex has an even degree. The graph has an Eulerian circuit.")
    elif odd_degree_count == 2:
        print("Exactly two vertices have an odd degree. The graph has an Eulerian path.")
}
```

A Hamiltonian path is a path in a graph that visits every vertex exactly once. Unlike Eulerian paths, Hamiltonian paths do not have a simple set of degree-based criteria to determine their existence. The problem of determining whether a Hamiltonian path exists is NP-complete, meaning there is no known efficient algorithm to solve it for all graphs.

Connection Between Eulerian and Hamiltonian Paths

The connection between Eulerian and Hamiltonian paths can be explored using the concept of the line graph (also known as the adjacency graph). If a graph G has an Eulerian path, its line graph has a Hamiltonian circuit.



As we have seen, the Eulerian path problem can be solved much more easily than the Hamiltonian path problem.