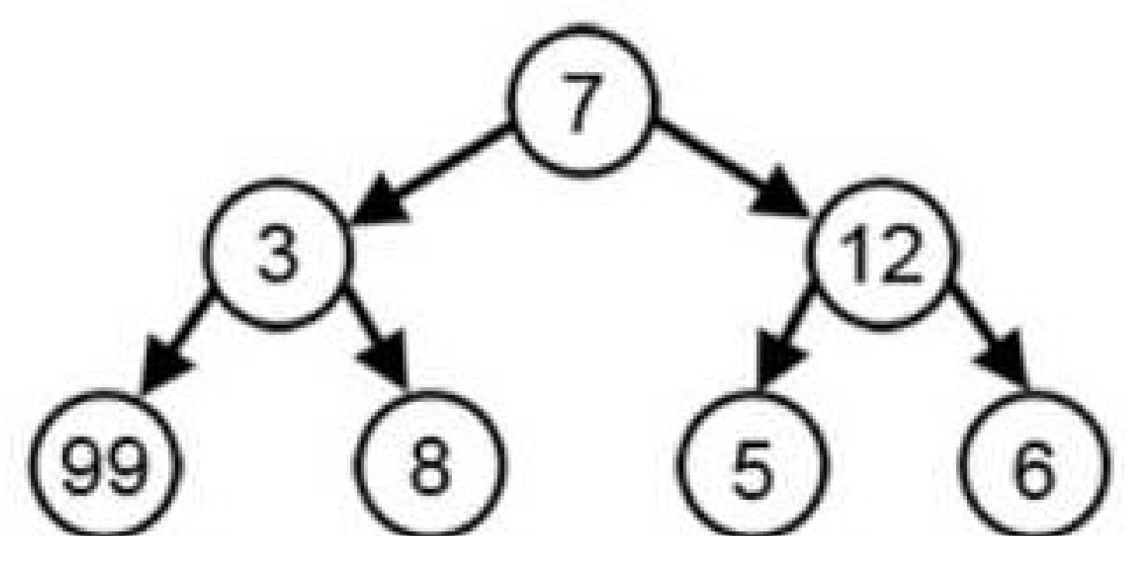
# EXPLORING GREEDY IN REAL WORLD MATCHING

Zachary George, Professor: Eric Balkanski

## INTRODUCTION TO THE RESEARCH

Even though greedy has the worst case out of all the possible matching algorithms in this study it focuses on figuring out why it is the most effective in real world capabilities. For instance, companies like Uber/Doordash use greedy algorithm to match the server (driver) to the user (person requesting). The idea of greedy is essentially matching the newest user to the unmatched server. In experimental studies there are real-time taxi with 10000 users and servers and results are surprising cause even though its worst-case greedy beats out the other matching algorithms like HST and is almost on par with offline matching (most effective however can't be applied in real-world due to it always being online). Therefore, the goal of my project is to test why through programming test trials in python that will allow me to use randomized data

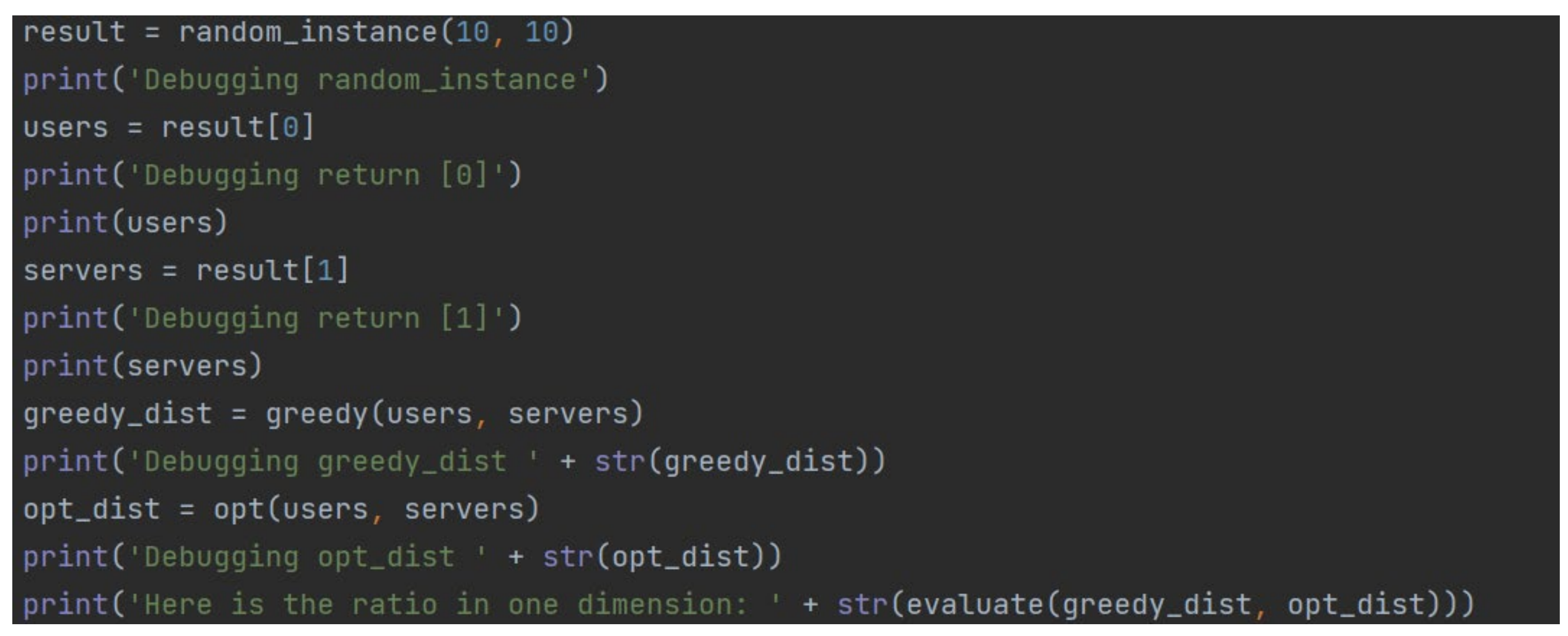


Greedy algorithm being bad because the path 7+12+6 is not the greatest sum (theoretical)

## METHODS

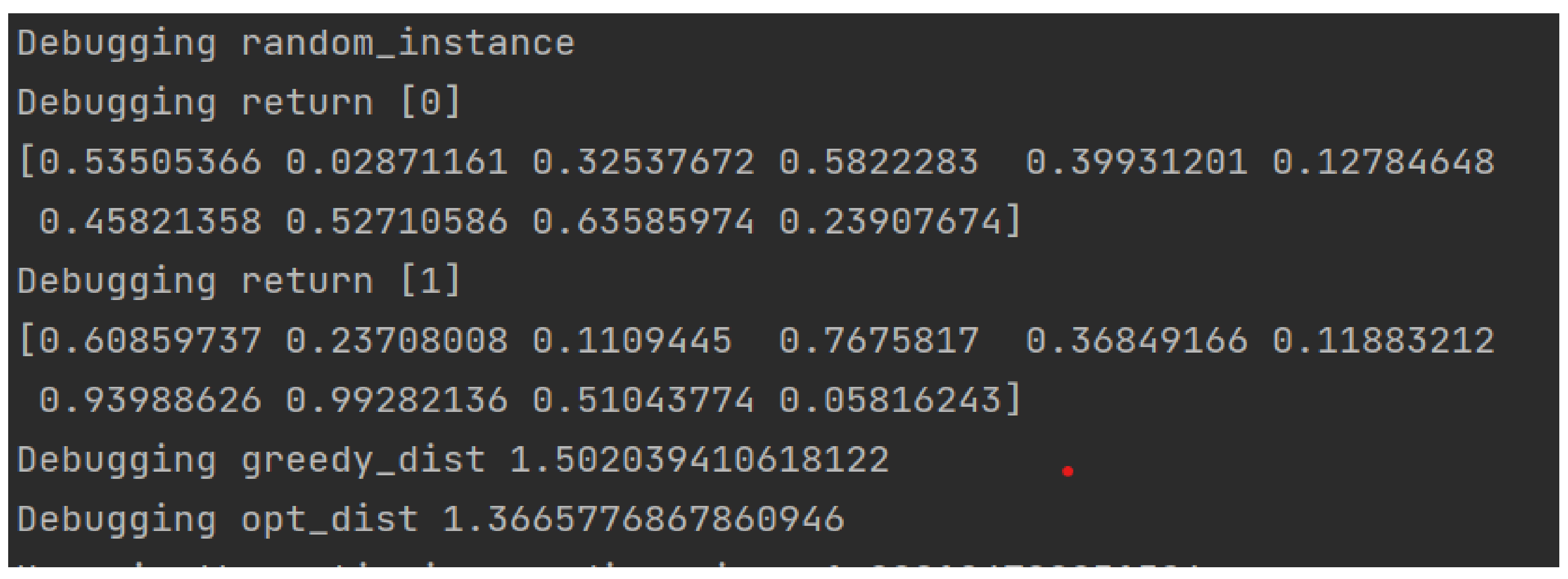
The methods used for this study is python testing where essentially I utilized the PyCharm IDE. Then through using python made it so that it generated random users/servers as coordinate points and then applied the greedy algorithm. After that there was an optimal algorithm in where it would find the optimal through methods defined, often looking at the minimum value present in the randomly generated coordinate points. Additionally, did it in a two-dimensional scale which is more realistic such and through utilizing the distance formula was able to get a methodology similar to how companies like Uber/Doordash have their set of users/servers and apply the algorithm

## OUTPUT ONE DIMENSION



```python
result = random_instance(10, 10)
print('Debugging random_instance')
users = result[0]
print('Debugging return [0]')
print(users)
servers = result[1]
print('Debugging return [1]')
print(servers)
greedy_dist = greedy(users, servers)
print('Debugging greedy_dist ' + str(greedy_dist))
opt_dist = opt(users, servers)
print('Debugging opt_dist ' + str(opt_dist))
print('Here is the ratio in one dimension: ' + str(evaluate(greedy_dist, opt_dist)))
```

Main method above that is applied to one-dimension



```
Debugging random_instance
Debugging return [0]
[0.53505366 0.02871161 0.32537672 0.5822283  0.39931201 0.12784648
 0.45821358 0.52710586 0.63585974 0.23907674]
Debugging return [1]
[0.60859737 0.23708008 0.1109445  0.7675817  0.36849166 0.11883212
 0.93988626 0.99282136 0.51043774 0.05816243]
Debugging greedy_dist 1.502039410618122
Debugging opt_dist 1.3665776867860946
```

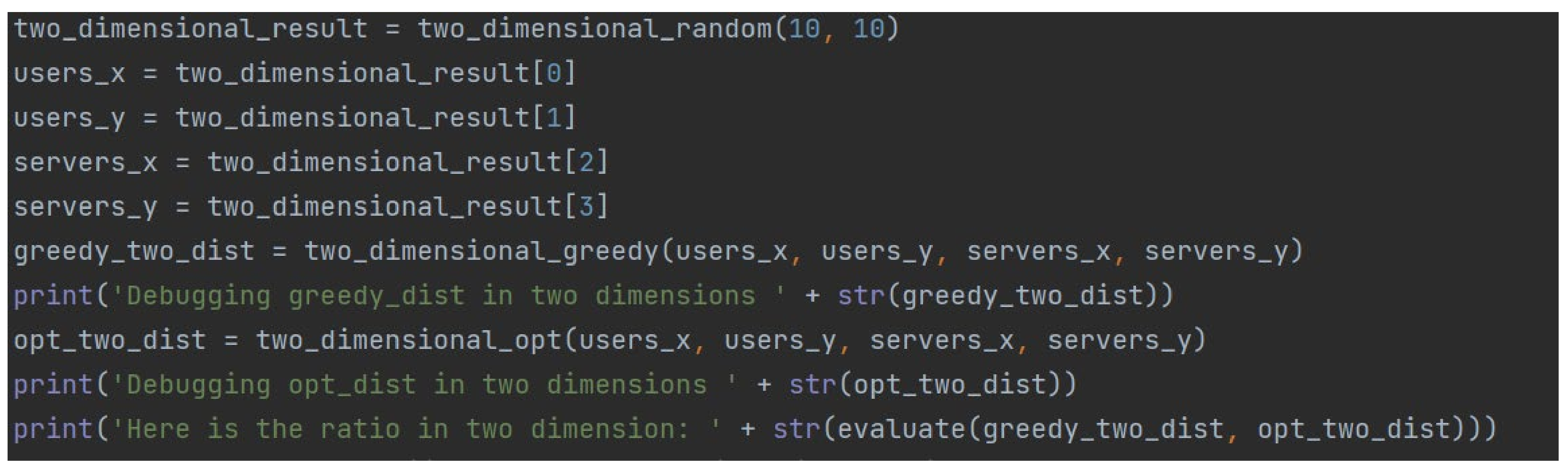
## HIGHLIGHTED FACTS HERE

- Theoretically the greedy algorithm has the least optimal worst-case run time
- In theory there would be better algorithms like HST (tree)
- Therefore, the research project is aimed at looking at why the greedy algorithm works in real-world situations
- In real-world situations it'd be two-dimensional data
- Therefore, the research was done on one-dimensional at first and then two-dimensional which represents what Uber/Doordash would map it as
- Through trial/testing we see the outcome of experiment

## CONCLUSIONS

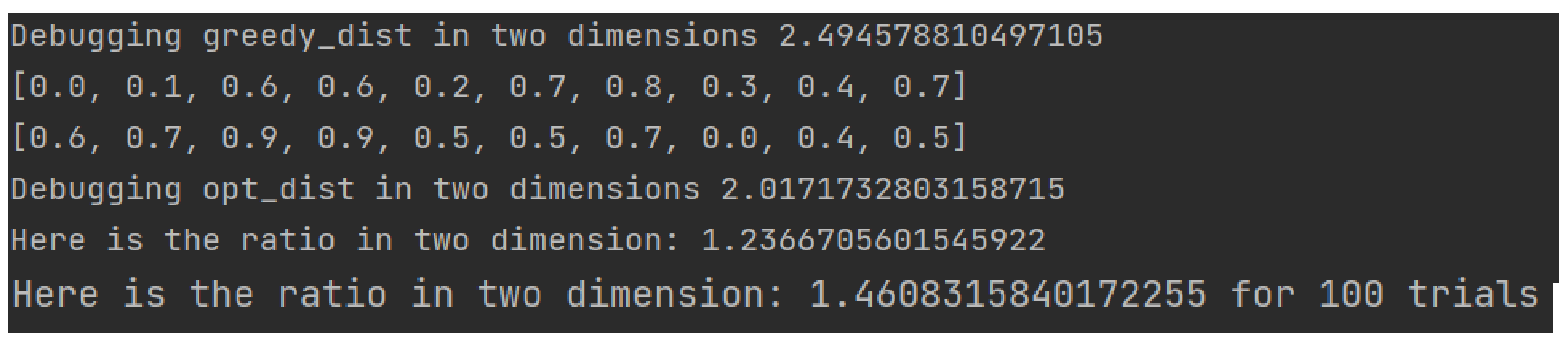
Therefore, the conclusion proves that greedy algorithm is optimal for figuring out how to match servers/users despite having the worst run time theoretically. As noted in the sample the ratio of one and two dimensions are extremely close to the value of ~1 which means greedy is super close to being optimal and is great for real-world applications, the two-dimensional output can be referenced below, and it shows this exact fact of how the optimal and greedy are both close to 1 when you take the ratio of them which means that they are the around the same proving that greedy is great in a real world scenario with receivers and servers being Uber drivers and those who call the Ubers. Furthermore, this can be applied to numerous real-world applications where there are a server and receiver, and this research helps contribute to the fact that for data matching greedy algorithm is optimal in these situations.

### Main method calling two dimensional



```python
two_dimensional_result = two_dimensional_random(10, 10)
users_x = two_dimensional_result[0]
users_y = two_dimensional_result[1]
servers_x = two_dimensional_result[2]
servers_y = two_dimensional_result[3]
greedy_two_dist = two_dimensional_greedy(users_x, users_y, servers_x, servers_y)
print('Debugging greedy_dist in two dimensions ' + str(greedy_two_dist))
opt_two_dist = two_dimensional_opt(users_x, users_y, servers_x, servers_y)
print('Debugging opt_dist in two dimensions ' + str(opt_two_dist))
print('Here is the ratio in two dimension: ' + str(evaluate(greedy_two_dist, opt_two_dist)))
```

### Output of two dimension greedy and opt-distance



```
Debugging greedy_dist in two dimensions 2.494578810497105
[0.0, 0.1, 0.6, 0.6, 0.2, 0.7, 0.8, 0.3, 0.4, 0.7]
[0.6, 0.7, 0.9, 0.9, 0.5, 0.5, 0.7, 0.0, 0.4, 0.5]
Debugging opt_dist in two dimensions 2.0171732803158715
Here is the ratio in two dimension: 1.2366705601545922
Here is the ratio in two dimension: 1.4608315840172255 for 100 trials
```

## REFERENCES + ACKNOWLEDGEMENTS

1. Yongxin Tong † Jieying She § Bolin Ding ‡ Lei Chen § Tianyu Wo † Ke Xu † †SKLSDE Lab, NSTR, and IRI, Beihang University, China §The Hong Kong University of Science and Technology, Hong Kong SAR, China ‡Microsoft Research, Redmond, WA, USA. 2016;9(12):1053-1064

Acknowledgements: Columbia Summer Undergraduate Research Program

In collaboration with