Dean Boerner
ECON 641
12/14/2022

Final Project Description

For my ECON 641 final project, I created an interactive implementation of my proposed solution to the problem of garden-style apartment community parking spot reallocation. Specifically, I coded up an implementation of the Top Trading Cycles algorithm (TTC) for its utilization within a simple web application. Here, I'll describe how the app works and can be used to achieve my goal of moving toward my optimality definition of Pareto efficiency, wherein there exists no pair of agents (i.e. apartment residents, in this case) who both want to switch parking spots with one another after the algorithm's use.

Because of residential turnover and heterogeneous valuations of qualities such as distance, dimensions, and coverage, a given apartment complex's parking spot matching is likely suboptimal in this regard. But TTC has properties that make it a suitable solution. Foremost among these is that it has been theoretically and empirically shown to result in my goal of Pareto efficiency. Secondarily, it also has the attractive property of strategy-proofness, which means that it's in agents' best interest to report their true preferences.

The algorithm entails each agent figuratively pointing to the assignee of their top available preference at a given moment, with those forming a cycle getting assigned that preference and the market as a whole shrinking accordingly. Consistent with this, my code works to shrink a Python dictionary of apartment residents and their preferences down to emptiness, keeping track of assignments along the way and ultimately outputting them in a simple table. It identifies cycles by extracting the latest info from each iteration of the dictionary and then applying a function from within the NetworkX Python module that uses Johnson's algorithm. Upon further research following presentation of my solution in class, I discovered that a large block of my Python program reserved for tie-breaking—specifically, a point in the process wherein there exist no cycles—was unnecessary; in fact, there being a finite number of goods and agents ensures there will always be a cycle. This results in less verbose code that still transforms—as illustrated in the miniature example below—a dictionary representing current assignments and preferences (L) into a more efficient allocation (R).

```
{1700: [1, [5, 4, 2, 1, 3]],
 1701: [2, [2, 4, 3, 5, 1]],
 1702: [3, [1, 3, 2, 4, 5]],
 1703: [4, [5, 2, 4, 1, 3]],
 1704: [5, [1, 5, 3, 4, 2]]}
```

	1700	1701	1702	1703	1704
Allocation	5	2	3	4	1

Key aspects of the program are briefly described and given below.

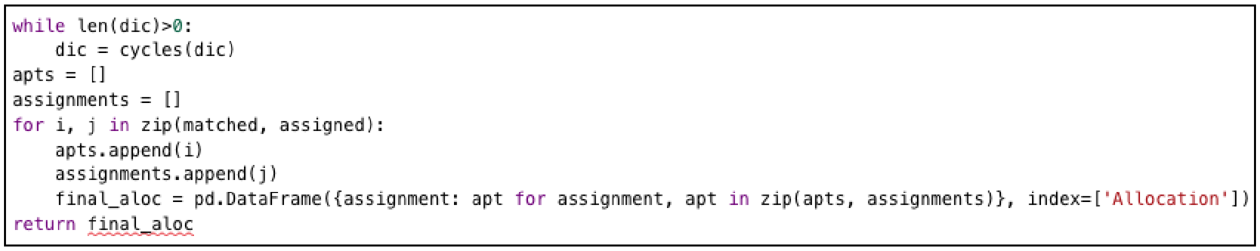
This function takes in the latest preference dictionary and returns edges of a directed graph.

```
def edges(dic):
    top_prefs = []
    for apt in dic: #for each apt, find what apartment has their top pref as its current assignment
        best = dic[apt][1][0]
        for other in dic:
            if dic[other][0]==best:
                match=other
                top_prefs.append((apt, match))
            else:
                pass
    return(top_prefs) #return a list of tuples (i, j), where j has i's most preferred assignment
```

This function iterates through cycles of that graph and updates the dictionary accordingly.

```
def cycles(dic):
    #create a graph object from which to identify a list of cycles or lack thereof
    G = nx.DiGraph()
    G.add_edges_from(edges(dic))
    cycles = list(nx.simple_cycles(G))
    for cycle in cycles:
        first = cycle[0]
        for apt in cycle:
            #if the apartment is the last in the cycle, they get the first's spot
            if cycle.index(apt)==(len(cycle)-1):
                matches.append((apt, first))
                matched.append(apt)
                assigned.append(dic[first][0])
            else: #otherwise, people just get whose after them
                next_apt = cycle[cycle.index(apt)+1]
                matches.append((apt, next_apt))
                matched.append(apt)
                assigned.append(dic[next_apt][0])
    dic = {key:value for (key, value) in dic.items() if ((key not in matched)|(value[0] not in assigned))}
    for apt in dic: #update each apartment's preference list to remove any spots that were just assigned
        dic[apt][1] = [pref for pref in dic[apt][1] if pref not in assigned]
    return(dic) #return the latest dic
```

This block specifies how long the program should be run and returns the end result.

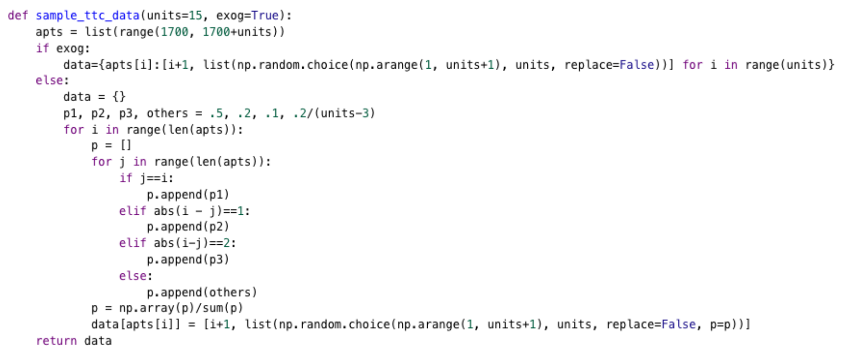


```
while len(dic)>0:
    dic = cycles(dic)
apts = []
assignments = []
for i, j in zip(matched, assigned):
    apts.append(i)
    assignments.append(j)
    final_aloc = pd.DataFrame({assignment: apt for assignment, apt in zip(apts, assignments)}, index=['Allocation'])
return final_aloc
```

The app code in its entirety is shown here.

Use of the application is very straightforward. Once it's open, a user simply needs to specify the number of required assignments and then upload a CSV file wherein the first two columns give an apartment unit and its existing assignment, and the remaining give that row's preferences from most preferred to least. While the sample data generator by default provides columns for 15 preferences, it could also easily serve as a template. After deleting the fake data, the user could use an autocomplete feature of a spreadsheet application like Microsoft Excel or Google Sheets to add more columns, or they could delete some if necessary. Finally, while uploading the file automatically outputs the TTC-determined final assignment, one can also get a picture of who, if anyone, would be made better off from the reallocation by checking the "Observe Efficiency Gains" box. Then a row's existing assignment is highlighted in orange, while re-assignments (if any) are in green. Screenshots of the app in action are on page 6.

I use multiple sample data types to gauge the performance of the solution, both in terms of algorithmic efficiency and Pareto efficiency. I consider data simulating a situation in which pre-existing assignments are completely independent of preferences/apartment location—wherein there are large improvements to be made—and one in which preferences/apartment location are very predictive of pre-existing assignments—wherein improvements to be made are relatively few. The former of those two data-generating processes is self-explanatory, while the latter is a process in which there's a 50% chance that a unit already has its ideal parking spot and an 80% chance it has one of its three most preferred spots. Specifics are below.



```python
def sample_ttc_data(units=15, exog=True):
    apts = list(range(1700, 1700+units))
    if exog:
        data={apts[i]:[i+1, list(np.random.choice(np.arange(1, units+1), units, replace=False))] for i in range(units)}
    else:
        data = {}
        p1, p2, p3, others = .5, .2, .1, .2/(units-3)
        for i in range(len(apts)):
            p = []
            for j in range(len(apts)):
                if j==i:
                    p.append(p1)
                elif abs(i - j)==1:
                    p.append(p2)
                elif abs(i-j)==2:
                    p.append(p3)
                else:
                    p.append(others)
            p = np.array(p)/sum(p)
            data[apts[i]] = [i+1, list(np.random.choice(np.arange(1, units+1), units, replace=False, p=p))]
    return data
```

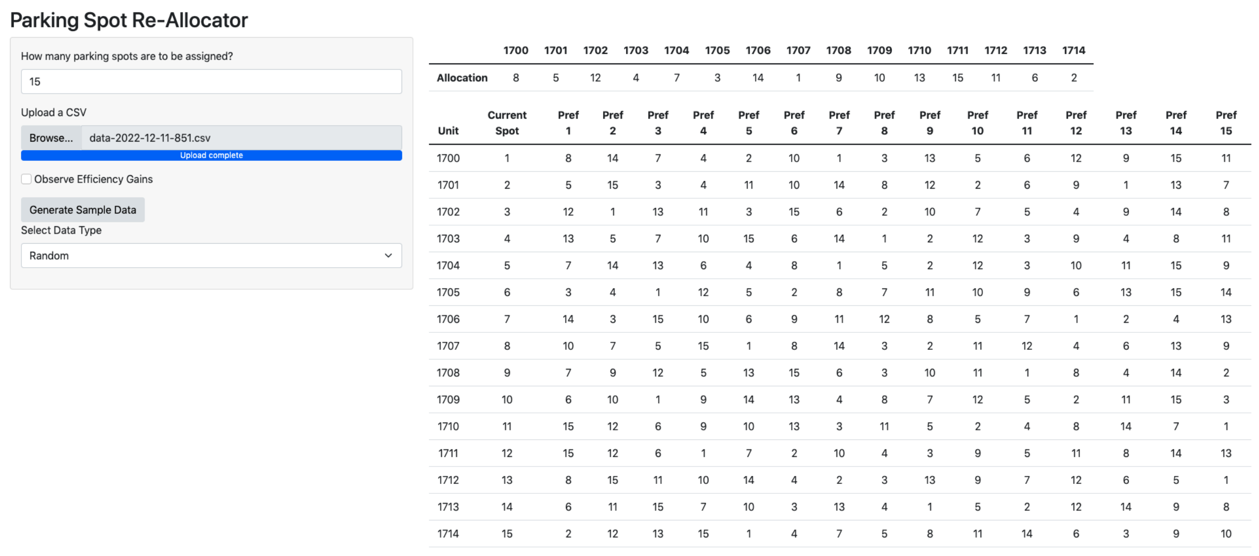
Using both hypothetical data types, I ran the TTC application program thousands of times to see how its runtime scales with larger allocative problems and to see what sort of improvements came about on average. Regarding the former of those two concerns, I observed how average runtime changes as agent/parking spot count increases from 5 to 100, using a sample size of 10 computations for each agent/parking spot count. While it appears that the program is of polynomial time complexity, runtime increases more slowly for the data simulating the more realistic, endogenous pre-reallocation assignments. However, for two reasons, I don’t think runtime should be much of a concern. Firstly, for N=1,000, runtime still comes out to less than one minute (see page 7), and I don’t think uses of the program for problems of that scale would be very common anyway. Secondly, because most residents’ preferences should be much higher for the limited set of parking spaces proximate to their residence (e.g. 15 or so) than the community’s remainder, sufficiently dispersed communities could apply the algorithm in separate blocks of the complex, likely without the loss of much or any efficiency gains.

Regarding the latter of the two concerns, efficiency improvements, testing is promising. The mean average per-person improvements in allocation—according to place among a given preference list—across 2,000 simulations is about 1.15 for the endogenous data type and 7.00 for the exogenous data type. In addition, no apartment units are left with worse allocations according to their preference rankings in any of the simulations (see page 8). Finally, indicating Pareto efficiency, I find that re-running TTC after simulating implementation of a prior

TTC-suggested allocation (i.e. updating assignments in accordance with the TTC results) never results in any changes whatsoever (see page 9).

In conclusion, I think this TTC solution is suitable for this parking-spot reallocation problem and improvement goals, despite some obvious limitations. One is that more mechanism design (e.g. an auction) is needed for the allocation of unassigned parking spaces, as TTC doesn't directly address this component. Furthermore, eliciting truthful preferences may be difficult to do or taxing on an apartment community's residents. Nevertheless, this solution's strengths may outweigh its shortcomings. In addition to having improvements in efficiency and an acceptable runtime, it also could easily be utilized periodically (e.g. biannually) because of turnover in residents and preferences. Furthermore, assuming a given resident's preferences are stable over time, they can be kept on file, so demands on residents are limited.

App Screenshot 1



Parking Spot Re-Allocator

How many parking spots are to be assigned?

15

Upload a CSV

Browse... data-2022-12-11-851.csv

Upload complete

☐ Observe Efficiency Gains

Generate Sample Data

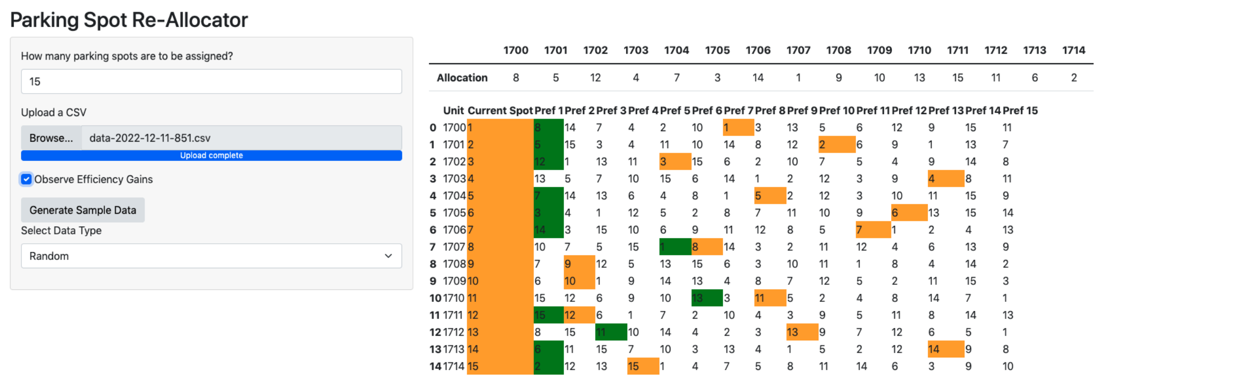
Select Data Type

Random

	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711	1712	1713	1714
Allocation	8	5	12	4	7	3	14	1	9	10	13	15	11	6	2

Unit	Current Spot	Pref 1	Pref 2	Pref 3	Pref 4	Pref 5	Pref 6	Pref 7	Pref 8	Pref 9	Pref 10	Pref 11	Pref 12	Pref 13	Pref 14	Pref 15
1700	1	8	14	7	4	2	10	1	3	13	5	6	12	9	15	11
1701	2	5	15	3	4	11	10	14	8	12	2	6	9	1	13	7
1702	3	12	1	13	11	3	15	6	2	10	7	5	4	9	14	8
1703	4	13	5	7	10	15	6	14	1	2	12	3	9	4	8	11
1704	5	7	14	13	6	4	8	1	5	2	12	3	10	11	15	9
1705	6	3	4	1	12	5	2	8	7	11	10	9	6	13	15	14
1706	7	14	3	15	10	6	9	11	12	8	5	7	1	2	4	13
1707	8	10	7	5	15	1	8	14	3	2	11	12	4	6	13	9
1708	9	7	9	12	5	13	15	6	3	10	11	1	8	4	14	2
1709	10	6	10	1	9	14	13	4	8	7	12	5	2	11	15	3
1710	11	15	12	6	9	10	13	3	11	5	2	4	8	14	7	1
1711	12	15	12	6	1	7	2	10	4	3	9	5	11	8	14	13
1712	13	8	15	11	10	14	4	2	3	13	9	7	12	6	5	1
1713	14	6	11	15	7	10	3	13	4	1	5	2	12	14	9	8
1714	15	2	12	13	15	1	4	7	5	8	11	14	6	3	9	10

App Screenshot 2



Parking Spot Re-Allocator

How many parking spots are to be assigned?

15

Upload a CSV

Browse... data-2022-12-11-851.csv

Upload complete

☑ Observe Efficiency Gains

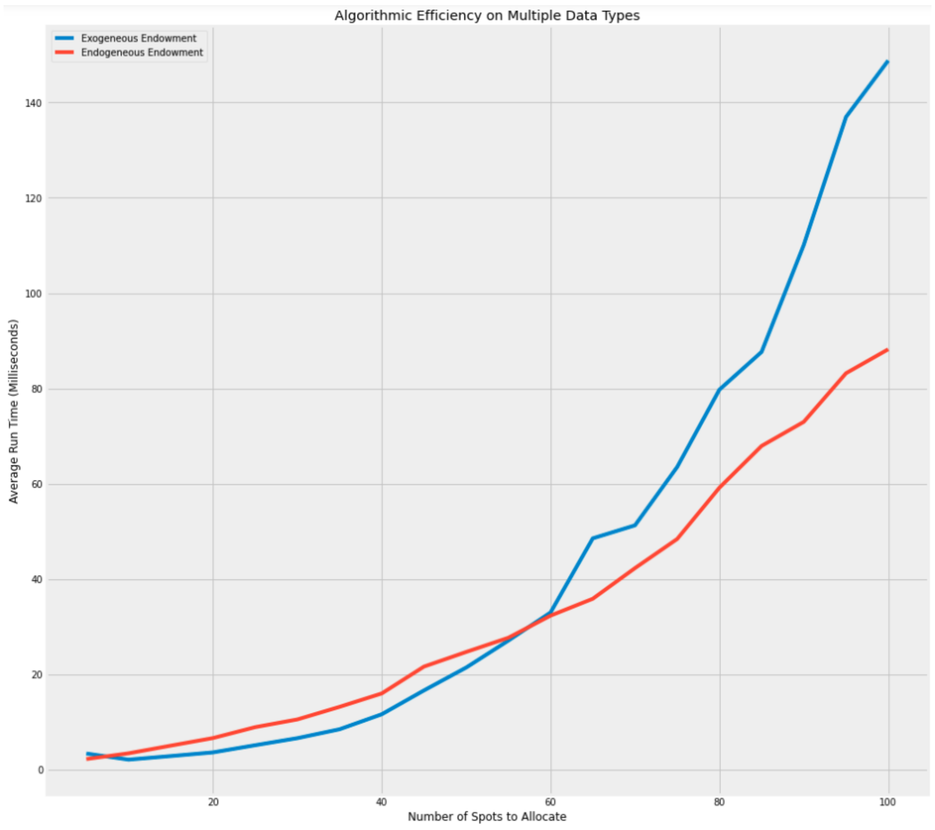
Generate Sample Data

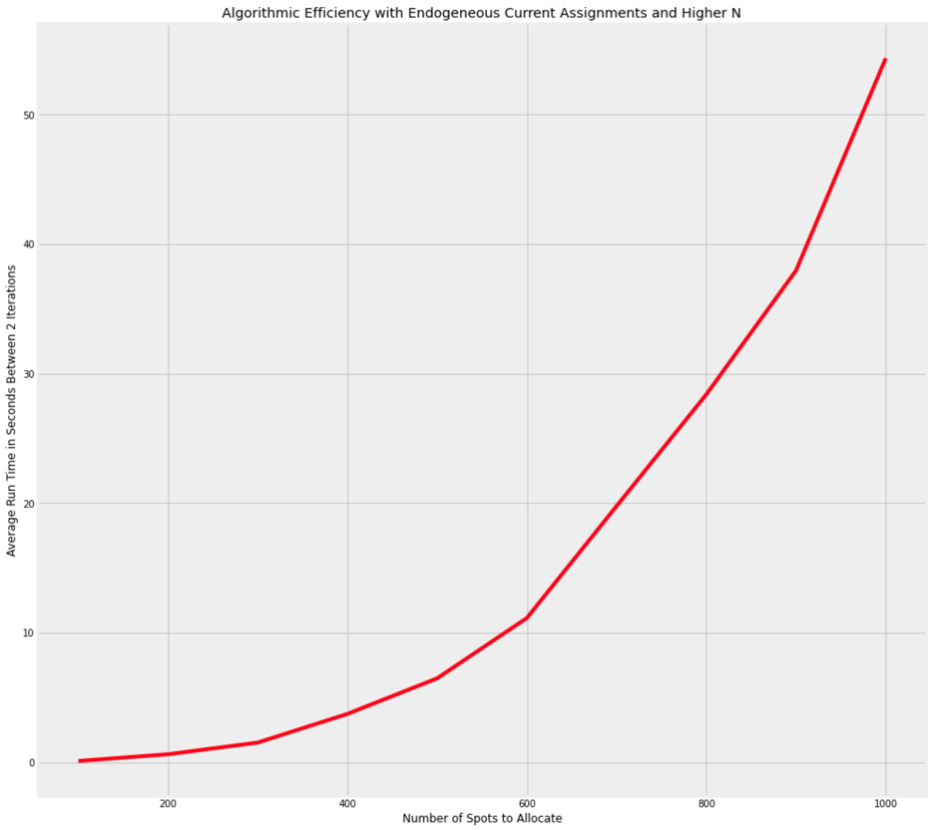
Select Data Type

Random

	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711	1712	1713	1714
Allocation	8	5	12	4	7	3	14	1	9	10	13	15	11	6	2

	Unit	Current Spot	Pref 1	Pref 2	Pref 3	Pref 4	Pref 5	Pref 6	Pref 7	Pref 8	Pref 9	Pref 10	Pref 11	Pref 12	Pref 13	Pref 14	Pref 15
0	1700	1	8	14	7	4	2	10	1	3	13	5	6	12	9	15	11
1	1701	2	5	15	3	4	11	10	14	8	12	2	6	9	1	13	7
2	1702	3	12	1	13	11	3	15	6	2	10	7	5	4	9	14	8
3	1703	4	13	5	7	10	15	6	14	1	2	12	3	9	4	8	11
4	1704	5	7	14	13	6	4	8	1	5	2	12	3	10	11	15	9
5	1705	6	3	4	1	12	5	2	8	7	11	10	9	6	13	15	14
6	1706	7	14	3	15	10	6	9	11	12	8	5	7	1	2	4	13
7	1707	8	10	7	5	15	1	8	14	3	2	11	12	4	6	13	9
8	1708	9	7	9	12	5	13	15	6	3	10	11	1	8	4	14	2
9	1709	10	6	10	1	9	14	13	4	8	7	12	5	2	11	15	3
10	1710	11	15	12	6	9	10	13	3	11	5	2	4	8	14	7	1
11	1711	12	15	12	6	1	7	2	10	4	3	9	5	11	8	14	13
12	1712	13	8	15	11	10	14	4	2	3	13	9	7	12	6	5	1
13	1713	14	6	11	15	7	10	3	13	4	1	5	2	12	14	9	8
14	1714	15	2	12	13	15	1	4	7	5	8	11	14	6	3	9	10


Algorithmic Efficiency on Multiple Data Types
Exogeneous Endowment
Endogeneous Endowment
Average Run Time (Milliseconds)
Number of Spots to Allocate
0
20
40
60
80
100
120
140
20
40
60
80
100

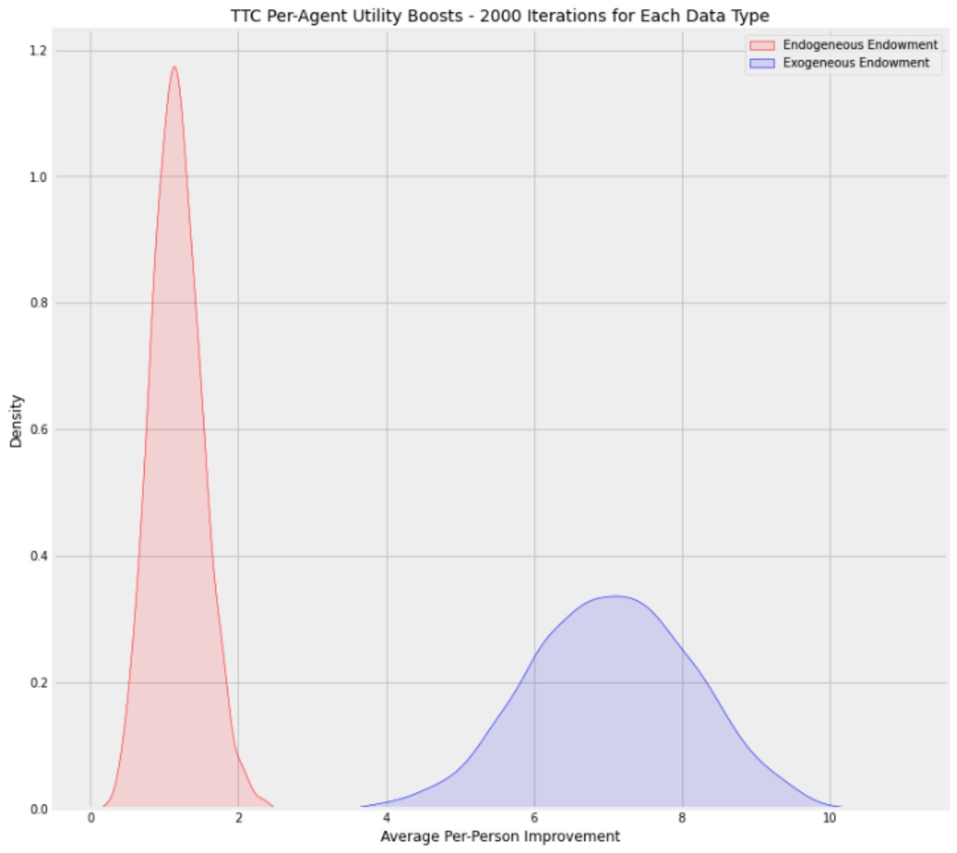

Algorithmic Efficiency with Endogeneous Current Assignments and Higher N
Average Run Time in Seconds Between 2 Iterations
Number of Spots to Allocate
0
10
20
30
40
50
200
400
600
800
1000

```python
average_ut_boost_endog = [None]*2000
any_negatives_endog = [None]*2000
#code to run TTC 2000 times and record improved efficiency
for i in range(2000):
    existing_situation = sample_ttc_data(units=15, exog=False) #record an initial state
    init_ut = []
    final_ut = []
    for apt in existing_situation: #for each apartment in that initial state,
        #record what order their current assignment is in in their true preferences
        init_ut.append(15 - existing_situation[apt][1].index(existing_situation[apt][0]))
    #then run TTC to allocate
    final = ttc(existing_situation)
    for apt in list(final.columns): #for each column/apt, record its final allocation and the utility from it
        final_aloc = final[apt].values[0]
        final_ut.append(15 - existing_situation[apt][1].index(final_aloc))
    #record the average difference per apartment
    ut_imp = np.array(final_ut) - np.array(init_ut)
    any_negatives_endog[i] = int(np.any(ut_imp < 0))
    average_ut_boost_endog[i] = np.mean(ut_imp)

average_ut_boost_exog = [None]*2000
any_negatives_exog = [None]*2000
#code to run TTC 2000 times and record improved efficiency
for i in range(2000):
    existing_situation = sample_ttc_data(units=15, exog=True) #record an initial state
    init_ut = []
    final_ut = []
    for apt in existing_situation: #for each apartment in that initial state,
        #record what order their current assignment is in in their true preferences
        init_ut.append(15 - existing_situation[apt][1].index(existing_situation[apt][0]))
    #then run TTC to allocate
    final = ttc(existing_situation)
    for apt in list(final.columns): #for each column/apt, record its final allocation and the utility from it
        final_aloc = final[apt].values[0]
        final_ut.append(15 - existing_situation[apt][1].index(final_aloc))
    #record the average difference per apartment
    ut_imp = np.array(final_ut) - np.array(init_ut)
    any_negatives_exog[i] = int(np.any(ut_imp < 0))
    average_ut_boost_exog[i] = np.mean(ut_imp)
```

```python
if sum(any_negatives_exog)==0 and sum(any_negatives_endog)==0:
    print('No one would be made worse off.')
else:
    print('Uh-oh.')
exog_boost = round(np.mean(average_ut_boost_exog), 2)
endog_boost = round(np.mean(average_ut_boost_endog), 2)
print(f'The mean average utility boost from states of exogeneous assignment is {exog_boost}.')
print(f'The mean average utility boost from states of endogeneous assignment is {endog_boost}.')
```

```
No one would be made worse off.
The mean average utility boost from states of exogeneous assignment is 7.04.
The mean average utility boost from states of endogeneous assignment is 1.17.
```



Re-Running TTC

```
initial_data = sample_ttc_data(units=15, exog=True)
data_after_aloc = copy.deepcopy(initial_data)
alocs = ttc(initial_data) #first TTC
for apt in data_after_aloc:
    data_after_aloc[apt][0] = alocs[apt].values[0] #allocating according to first TTC
re_alocs = ttc(data_after_aloc) #second TTC
```

```
#first TTC results
alocs
```

	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711	1712	1713	1714
Allocation	10	7	5	13	2	4	9	8	3	12	11	1	6	14	15

```
#second TTC results
re_alocs
```

	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711	1712	1713	1714
Allocation	10	7	5	13	2	4	9	8	3	12	11	1	6	14	15

```
#difference
alocs-re_alocs
```

	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711	1712	1713	1714
Allocation	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Re-Running TTC Across 2,000 Simulated Preference Scenarios

```
any_changes = [None]*2000
for i in range(2000):
    initial_data = sample_ttc_data(units=15, exog=True)
    data_after_aloc = copy.deepcopy(initial_data)
    alocs = ttc(initial_data)
    for apt in data_after_aloc:
        data_after_aloc[apt][0] = alocs[apt].values[0]
    re_alocs = ttc(data_after_aloc)
    diff = alocs-re_alocs
    any_changes[i] = np.any(diff.iloc[0, :]!=0)
if ~np.any(any_changes):
    print(f'None of the {len(any_changes)} re-allocations following TTC resulted in any changes.')
```

```
None of the 2000 re-allocations following TTC resulted in any changes.
```