UNIVERSITY OF CALIFORNIA, BERKELEY
IEOR 169

Division of Inheritance

for the consideration of the heirs of Mr Logan Roy

Jonathan Bodine - jonathan.m.bodine@berkeley.edu
Ethan Ding - yifan.ding@berkeley.edu

Executive Summary

05/08/2020

The division of the contents of former multi-billionaire, Mr Logan Roy's estate has proved a never-before-seen modern challenge for the legal representation of the Roy family. Before his death, Mr Roy had an enormously large number of assets, some of which are liquid, others not so much, which need to be divided up among his heirs. This is compounded by the fact that many of these assets are laden with varying amounts of debt which, in some cases, sum up to a net negative valued asset for the person inheriting the item, although what this sum adds up to varies depending on which one of Mr Roy's heirs is appraising that asset; for example, Mrs Siobhan Roy, as a politically active professional would value the synergies she acquires from the acquisition of Mr Roy's newscaster channel as greater than the fines this channel is currently paying out as a result of former misconduct.

Additionally, as his lawyers dig through the intricacies of Mr Roy's estate, more assets are emerging on a daily basis. Exacerbating this trouble even further is Mr Roy's apparent sexual history; a slew of illegitimate children and mistresses are emerging every day to stake a claim to parts of Mr Roy's assets. Finally, Mr Roy's directive in his will when it came to divvying up his assets was also especially unhelpful for lawyers to parse: "Greed runs in my blood, just keep them from seeing green."

Ultimately, the legal representation of these parties have agreed to have some operations researchers come together and put together this report to attempt to establish an envy-free allocation of Mr Roy's assets. Due to the constantly changing landscape facing this problem, with DNA tests coming back with mixed results, we have chosen to come up with a set of algorithms which is generalizable to any situation which is finalized, where we have x number of heirs and y number of illiquid assets to divvy up between them, and the amount each heir values each assets is randomly distributed. Under those parameters we have built models with the following objectives in mind:

- Approximately envy free: where the total amount of envy post-allocation is minimal in general
- Envy-free up to one item: where the total amount of envy post-allocation is minimal, minus one item for each person
- Envy-free with cash: highly applicable, where the illiquid assets being divvied up are augmented by parts of Mr Roy's liquid assets
- Analysis with a more intelligent set of relative value assessments

Strategy and Approach

In order to formulate a set of decisions (optimal solution) with the goal of minimizing various forms of envy, a Linear Programming formulation will be constructed and carried out through a Mathematical Programming Language known as `(AMPL)` — pronounced “ample”. On a more technical level, we approached the problem with the following steps (all of the files mentioned below can be found attached to this project, should the reader want to run those files themselves):

- Write a python script that generates random .dat files for any specified number of people and items, which can be fed into the AMPL mathematical problem solver
- Write an AMPL model that captures the constraints of the problem for each version of our objective
- Run a grid-search of each AMPL model for different permutations of people / item combinations to assess the speed and performance of each model and how well we can resolve the different formulations of the question

Technical Overview

Approximate Envy Free

First we built an AMPL model for our assessment of approximate envy-freeness; we used an array of i people, j items, and a matrix of i x j valuations where the entry in the ith row and jth column represented how much the ith person valued the jth item, as a percentage of how much they valued the total sum of the items. In other words, each row in matrix summed to win, as the sum of the values they attribute all the items was the denominator for each of the values

Having defined this matrix, we set a matrix of i x j binary conditions which represented whether or not each jth item was allocated to the ith person, and introduced the condition that the sum of each column had to be 1, as every item needed to be allocated to someone.

Finally, we introduced p, which was calculated as the maximum amount of envy any individual feels for any other individual, as a percentage of how much they value the total amount. For example, a p-envy value of 1 would imply that there is an individual who values someone else's allocated products as 100% of the total value more than their own (true when there's only one item).

Simple Example

Our allocation for the simple example with our p-envy algorithm was:

	Car	Painting
Alice	1	0
Bob	0	1

With overall optimal p-envy = 0.5

Larger Example

	Bicycle	Dog	Motorcycle	Painting	Television
Alice	0	0	1	0	0
Bob	1	0	0	0	0
Carol	0	1	0	0	1
David	0	0	0	1	0

With overall optimal p-envy = 0.0465116

In Depth Analysis

For further analysis, we ran this model for every permutation possible from 0-10 people and items, for 30 separate sets of values randomly generated by our script that made .dat files which were compatible with the way our AMPL model accepted inputs. A wall clock measured how

long the algorithm took to run for each of the iterations we ran it for. Fig 1a below shows the mean log wall clock town in seconds for each of the 30 iterations we ran for every permutation up to 10 of people and items. Fig 1b shows the average p-envy value we produced for each iteration.

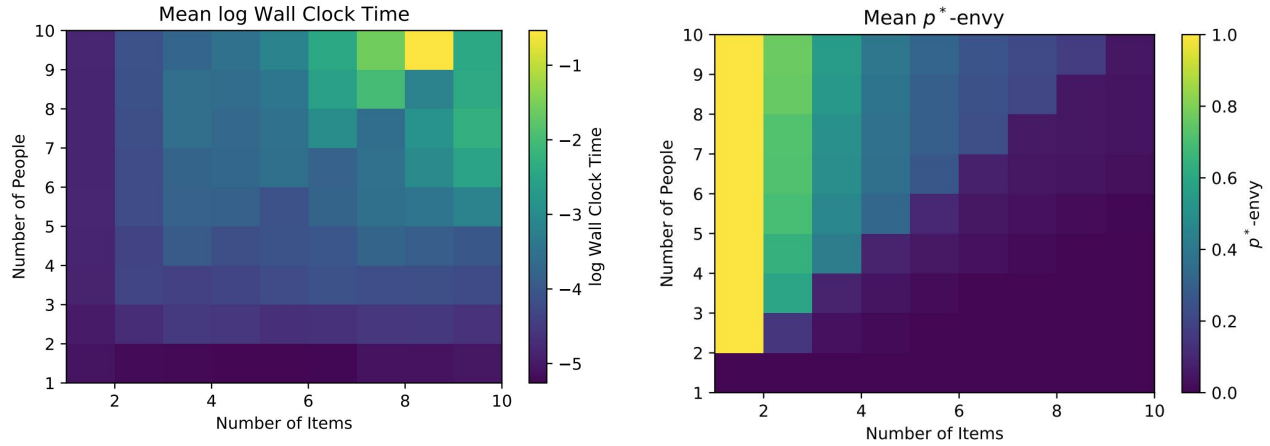

Figure 1a (left): a heat map showing the log wall clock time taken for an AMPL algorithm to run, for up to 10 people and 10 items. Figure 1b (left): a heat map showing the mean p-envy achieved for the 30 sets of data we used

Obviously it makes sense that for one item and more than one people, the p-envy free score is 100%, since 100% of the value as assessed by everyone is attributed to someone for everyone who was allocated nothing. This dramatically decreases for a second item since the most any one person can be jealous of is approximately half of the total value.

And interesting find was that the most difficult to compute allocations appeared when the number of items was just short of the number of people, implying that the most challenging thing would be to figure out who gets the short end of the stick; while we were able to run a 20p20o (20 people 20 objects) system reasonably quickly, 20p12o took the algorithm 5810s (1.6hrs) to run.

Also, increasing the number of people didn't always lead to more complexity in terms of runtime; the 15p13o data took 4584s (1.27hrs) to run, while the 16p13o data ran in only 335s. Even more interestingly, a 17p13o ran for >24hrs before we gave up on that iteration.

1-Item Envy Free

We added into the code the ability to select one item for removal when comparing two different peoples sets. This was accomplished by introducing another binary variable that allowed for one item to be selected per person for the comparing person to ignore. This was able to greatly reduce the complexity of the problem as can be seen by the following analysis.

Simple Example

	Car	Painting
Alice	0	1
Bob	1	0

With overall optimal p-envy = 0 (insignificant as both items are disregarded)

Larger Example

	Bicycle	Dog	Motorcycle	Painting	Television
Alice	1	0	0	0	0
Bob	0	0	0	1	0
Carol	0	0	1	0	1
David	0	1	0	1	0

With overall optimal p-envy = 0

In Depth Analysis

For further analysis, we followed the same methodology as the further analysis of approximate envy-freeness; we ran 30 iterations of the algorithm over a fresh dataset. A similar set of figures showing the wall clock time and mean p-envy level up to one item as in figure 1 is shown below.

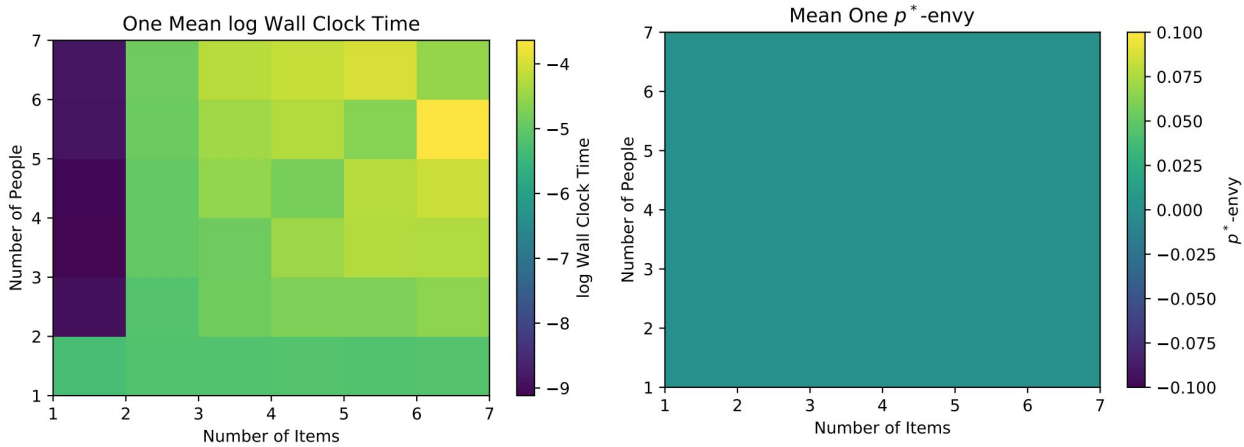

Figure 2a (left): a heat map showing the log wall clock time taken for the AMPL algorithm to run, for up to 10 people and 10 items, 30 iterations of the p-envy up to one item. Figure 12 (left): a heat map showing the mean p-envy achieved for the 30 sets of data we used

From heat map scale in figure 2a, we can see that the order of magnitude of time that this problem ran for is approximately 4 orders of magnitude faster; the problem is effectively made

simpler by a significant margin when we're dealing with p-envy free up to one item, because the one items makes it incredibly easy to achieve 0 envy, as seen in figure 2b.

When dealing with 1-item envy free, we weren't gated by runtime as much as the limitations of our licensed version of AMPL. While the problem is made simpler, the number of variables that need to be tracked goes up, and consequently we hit the 500 var and 500 cons limit very quickly and weren't able to go beyond 7 people / objects.

Envy Free w/ Cash

Adding to our original AMPL code, we added a new parameter for cash, ensuring that allocated cash to each person was greater or equal to 0 and that total cash allocated was equal to cash. Then we included the amount of cash allocated as part of the value assessment equation, as a percentage of the total value, the same way we might incorporate another object (as the previous constraints ensured that we wouldn't need to allocate cash in a binary way).

Simple Example

Assuming that there is only $1000 in cash, this is the optimal allocation.

	Car	Painting	Cash
Alice	0	1	1000
Bob	1	0	0

With overall optimal p-envy = 0.33

We also found that the minimum amount fo cash needed to achieve p-envy = 0 is $3000, with all 3000 allocated to Alice

Larger Example

With the larger example, we don't need any more than $1000 to achieve p-envy = 0

	Bicycle	Dog	Motorcycle	Painting	Television	Cash
Alice	0	0	1	0	0	400
Bob	1	0	0	0	0	0
Carol	0	1	0	0	1	600
David	0	0	0	1	0	0

With overall optimal p-envy = 0.

The minimum amount of money needed to acquire p-envy = 0 is $200, with the same allocation of items as above, but Alice has 0 and Carol has $200 in cash.

In Depth Analysis

Augmenting our existing python script to fit the format for cash was simple enough, and since we were averaging the overall runtime and p-envy value of our 30 runs, not using the same

existing data as the previous examples shouldn't have made a difference, especially since we're generating the new .dat files using the same script.

For the purpose of effectively implementing our algorithm, the values which were fed to the AMPL file underwent normalization; consequently, the largest a value could be 1, and the smallest it could be was 0. When it came to generating a value for cash, we ran iterations for $0.25, $0.50, $1.00 and $2.00, with $1.00 effectively meaning: equivalent to the largest valuation an individual attributes to any given item. Our overall times can be seen below:

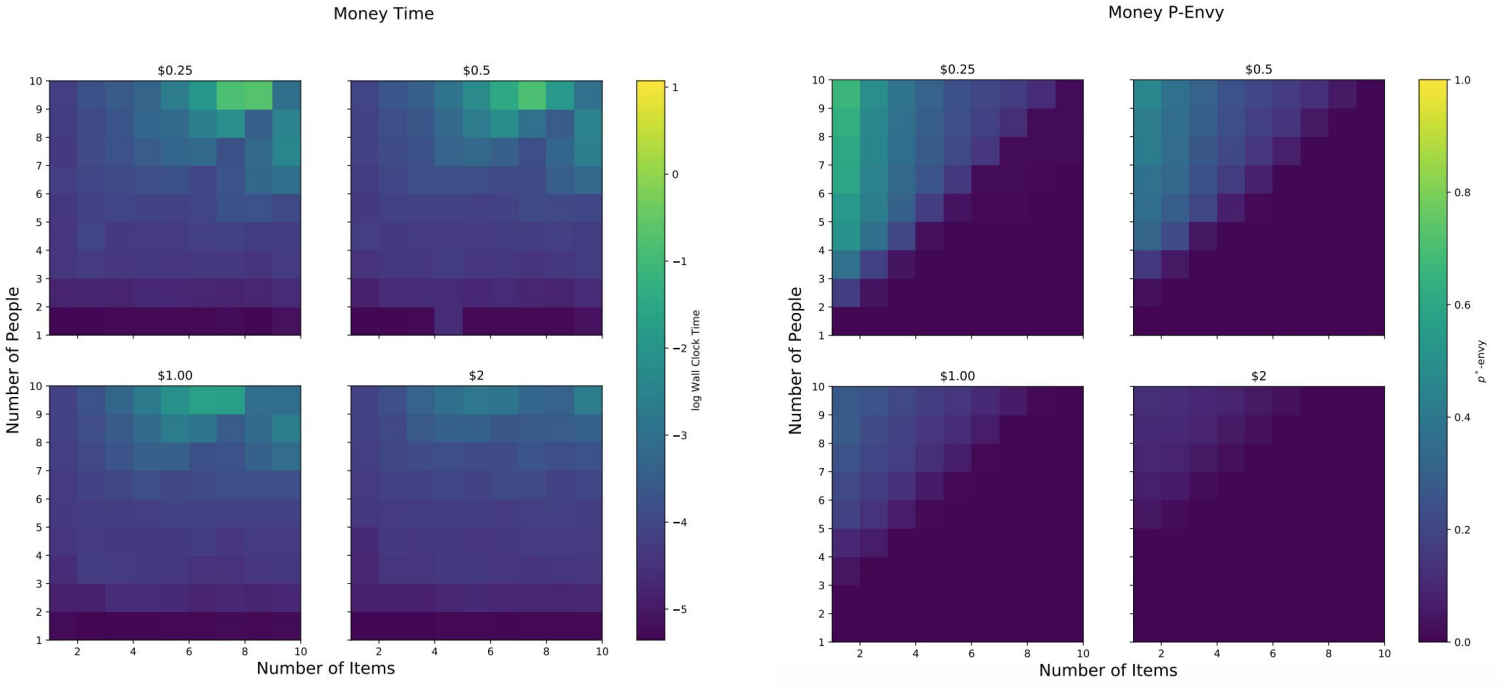

Figure 3a (left): a heat map showing the amount of time it took for the algorithm to run for the various amounts of cash we generated; the overall complexity of the problem has returned to a similar point to the original p-envy free problem. Figure 3b (right) shows the optimal p-envy value for the various cash quantities.

In terms of trends, obviously the more cash is in the system, the easier it is to offset any inequity felt by the people who are allocated less. Once a certain threshold of cash is hit, further allocation simply means giving everyone who wants more equivalently higher amounts of cash. It is also interesting to note that computation time is higher for lower cash quantities.

Improved Data Heuristics

However it isn't realistic to have data generated in a random way; while any one person might value something marginally more or less than someone else, the fact that any asset can be sold or bought for a fixed price means that there is a limit to subjective value perception. Consequently, we ran the same algorithm set up for a different type of data generation method, which wasn't entirely random, but instead generated a seed for each item and added variation that scaled with 20% of that said, to account for the fact that anyone's valuation of any one item

is probably not likely to be 20% more or less of anyone else's, although this specific value can be adjusted as well.

After making the necessary changes to the python script, we ran the original p-envy algorithm over it, and found conducted a side-by-side comparison of the two algorithm runs respective time and final p-envy produced.

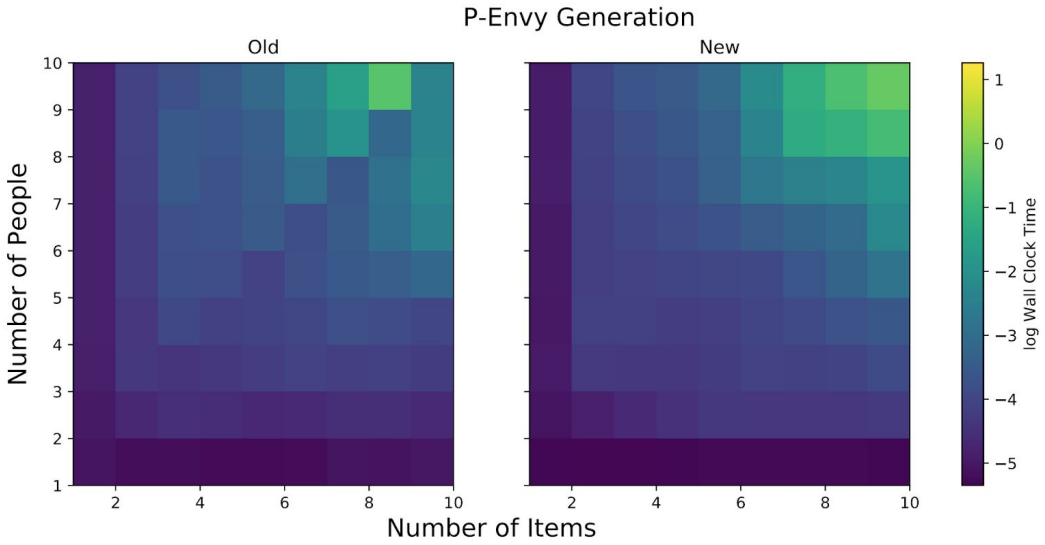

Figure 4a: a heat map showing the change in clock time taken to run the algorithm with the new heuristics in place. Note that NpNo data is no longer significantly easier to allocate than NpN-1o data; an interesting byproduct of our set up now having each item valued a similar amount by everyone

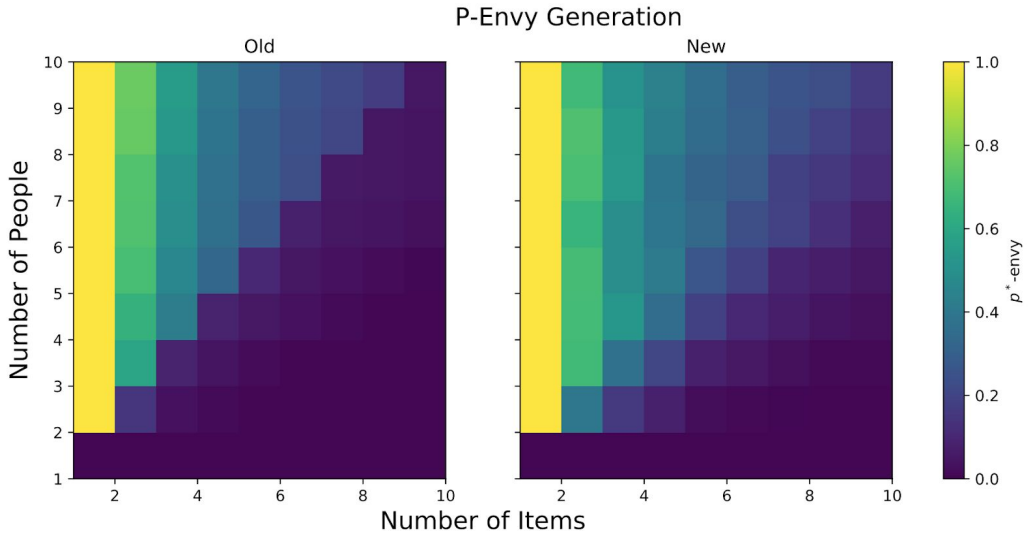

Figure 4b: a heat map showing the overall p-envy score comparison between our old data generation method and this new one. It is apparent that it is now more difficult to achieve lower values of p-envy, which make sense because there are fewer instances of people valuing one item so drastically differently from someone else that the allocation of that item becomes almost trivial.

Conclusions

As analysis has been pretty extensive throughout the report, we'll summarize our findings for anyone who skipped straight to the end:

- For the base version of p-envy free allocation, it was easier to achieve with more items and fewer people, and the longest algorithm run times were for iterations with just fewer items than people
- 1-item p-envy free is an incredibly fast algorithm without a lot of complexity, and produces a near-perfect result every single time it's run, for any combination of people and items.
- P-envy with cash is an interesting problem that has a similar order of magnitude to base p-envy allocation, but achieves better p-envy values, with more cash in the system producing progressively better p-envy values
- Creating a better system for generating valuations and data smooths out the difficulty of allocating datasets of n people and n-1 objects and produces weaker values of p-envy, but these values are more reflective of the real world.

Based off our findings, we recommend the legal team of Mr Roy's estate attempt to explain the concept of p-envy free up to one item to Mr Roy's heirs, as that is the only effective way to distribute his estate to minimize envy without requiring overly complex models that have no hope of effectively resolving the problem.

Appendix:

Data generation for parts 1 through 3:

```
def generate(obj = 5, ppl = 5):
	#set ITEM := 1 2 3 4;
	str1 = "set ITEMS :="
	for i in range(obj):
		str1 += " {}".format(i+1)
	str1 += ';\n'

	#set PEOPLE := 1 2 3 4;
	str2 = "set PEOPLE :="
	for i in range(ppl):
		str2 += " {}".format(i+1)
	str2 += ';\n \n'

	#param values: 1 2 3 4 :=

	str3 = "param values:"
	for i in range(obj):
		str3 += " {}".format(i+1)
	str3 += ':='

	str4 = ""
	for i in range(ppl):
		str4 += "\n{}".format(i+1)
		for i in range(obj):
			str4 += " {}".format(random.random())
	str4 += ';'
	str4

	filename = "/home/jon/Berkeley/ieor169/project/random_data/{}ppl{}obj".format(ppl, obj)
	filename += ".dat"

	file = open(filename, "w")
	file.write(str1)
	file.write(str2)
	file.write(str3)
	file.write(str4)
	file.close()
```

Data Generation for part 4:

```
def generate(obj=10, ppl=5):
    # set ITEM := 1 2 3 4;
    str1 = "set ITEMS :="
    for i in range(obj):
        str1 += " {}".format(i+1)
    str1 += ';\n'

    # set PEOPLE := 1 2 3 4;
    str2 = "set PEOPLE :="
    for i in range(ppl):
        str2 += " {}".format(i+1)
    str2 += ';\n \n'

    # param values: 1 2 3 4 :=

    str3 = "param values:"
    for i in range(obj):
        str3 += " {}".format(i+1)
    str3 += ':='

    heuristic = []
    for i in range(obj):
        heuristic.append(random.randint(1, 1000))

    str4 = ""
    for i in range(ppl):
        str4 += "\n{}".format(i+1)
        for j in range(obj):
            x = heuristic[j]
            str4 += " {}".format(x + random.randint(0, (x//5)))
    str4 += ';'
    str4

    filename = "/Users/jonathan/Desktop/random_data/{}ppl{}obj_heuristic".format(ppl, obj)
    filename += ".dat"

    file = open(filename, "w")
    file.write(str1)
    file.write(str2)
    file.write(str3)
    file.write(str4)
    file.close()
```

General code for running the experiments:

```
def run_one_experiment(ampl, obj = 5, ppl = 5):
	generate(obj = obj, ppl = ppl,)
	ampl.reset()
	ampl.read('/Users/jonathan/Desktop/169projectPEnvy.mod')

ampl.readData("/Users/jonathan/Desktop/random_data/{}ppl{}obj_heuristic.dat".format(ppl, obj))
	ampl.setOption('solver', 'cplex')
	start_time = time.time()
	ampl.solve()
	finish_time = time.time()
	p_envy = ampl.getObjective('p_envy').value()

	return p_envy, finish_time-start_time

ampl =
AMPL(Environment("/Users/jonathan/Documents/files/amplide.macosx64/amplide.macosx64"))

with open('part4.csv', 'a', newline='') as csvfile:
	fieldnames = ['Number of People', 'Number of Items', 'Optimal P-Envy', 'Wall Clock
Time']
	writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
	writer.writeheader()
	for i in range(1,11):
		for j in range(1,11):
			for _ in range(30):
				p_envy, run_time = run_one_experiment(ampl, i, j)
				out = {'Number of People': j,
					'Number of Items': i,
					'Optimal P-Envy': p_envy,
					'Wall Clock Time': run_time}
				print(out)
				writer.writerow(out)
				csvfile.flush()
```

Mod file for part 1:

```
set ITEMS;
set PEOPLE;

param values{i in PEOPLE, j in ITEMS};
param normalized_values{i in PEOPLE, j in ITEMS} = values[i,j] / sum{k in ITEMS}(values[i,k]);

var allocated{i in PEOPLE, j in ITEMS}, binary;
var p >= 0, <= 1;

minimize p_envy: p;

subject to

allocation_requirement{j in ITEMS}: sum{i in PEOPLE}(allocated[i, j]) = 1;
p_envy_free{i in PEOPLE, k in PEOPLE}: sum{j in ITEMS}((allocated[i,j] -
allocated[k,j])*normalized_values[i,j]) >= -p;
```

Mod file for part 2:

```
set ITEMS;
set PEOPLE;

param values{i in PEOPLE, j in ITEMS};
param normalized_values{i in PEOPLE, j in ITEMS} = values[i,j] / sum{k in ITEMS}(values[i,k]);

var allocated{i in PEOPLE, j in ITEMS}, binary;
var remove{i in PEOPLE, j in ITEMS, k in PEOPLE}, binary;
var p >= 0, <= 1;

minimize p_envy: p;

subject to

allocation_requirement{j in ITEMS}: sum{i in PEOPLE}(allocated[i, j]) = 1;
can_remove{i in PEOPLE, j in ITEMS, k in PEOPLE}: remove[i,j,k] <= allocated[k,j];
one_remove{i in PEOPLE, k in PEOPLE}: sum{j in ITEMS} (remove[i,j,k]) = 1;
p_envy_free{i in PEOPLE, k in PEOPLE}: sum{j in ITEMS}((allocated[i,j] + remove[i,j,k] -
allocated[k,j])*normalized_values[i,j]) >= -p
```

Mod File for part 3:

```
set ITEMS;
set PEOPLE;

param values{i in PEOPLE, j in ITEMS};
param cash;
param normalized_values{i in PEOPLE, j in ITEMS} = values[i,j] / (sum{k in ITEMS}(values[i,k])
+ cash);
param normalized_cash{i in PEOPLE} = 1/(sum{k in ITEMS}(values[i,k]) + cash);

var allocated{i in PEOPLE, j in ITEMS}, binary;
var cash_alloc{i in PEOPLE} >= 0, <= cash;
var p >= 0, <= 1;

minimize p_envy: p;

subject to

allocation_requirement{j in ITEMS}: sum{i in PEOPLE}(allocated[i, j]) = 1;
p_envy_free{i in PEOPLE, k in PEOPLE}: (cash_alloc[i] - cash_alloc[k])*normalized_cash[i] +
sum{j in ITEMS}((allocated[i,j] - allocated[k,j])*normalized_values[i,j]) >= -p;
cash_restriction: sum{i in PEOPLE}(cash_alloc[i]) = cash;
```