Heuristics in Optimization Project

Jack Francis and Christian Zamiela

August 27, 2020

1 Project Summary

For our project, we solved an optimization problem that was part of a 2019 Kaggle competition (Santa's Workshop Tour 2019). Kaggle is a data-science competition website that also includes optimization challenges. The problem is an allocation problem, where families have preferred days to attend Santa's Workshop. Santa accrues a cost depending on the number of families that attend and the preferences of each family. Thus, the objective of the problem is to minimize the overall cost accrued by Santa. The full details of the problem are provided in Section 2. To solve this problem, we use random search, guided local search, and the genetic algorithm to identify the optimal allocation of families to attend Santa's Workshop. Our implementation of each heuristic is provided in Section 3.2 and the code is provided in the Appendix. We test the effectiveness of the heuristics and heuristic hyperparameters in Section 4 and report our major findings in Section 5.

2 Research Problem

The research problem is an allocation problem, where families want to attend Santa's Workshop and Santa wants to minimize the total cost of families attending. Santa has opened his workshop for families to attend for each of the 100 days before Christmas, where day n is n days before Christmas (i.e. December 25). There are 5000 families, which combined are 21003 individuals, that have signed up to attend Santa's Workshop. Each family has provided a list of 10 preferred days, in order, that they would like to visit Santa's Workshop. An example of a few family's preferred visit days are shown in Table 1.

Table 1: Examples of family's preferences and varying number of people in each family. Choice is abbreviated as "ch".

family id	ch_0	ch_1	ch_2	ch_3	ch_4	ch_5	ch_6	ch_7	ch_8	ch_9	num people
0	52	38	12	82	33	75	64	76	10	28	4
1	26	4	82	5	11	47	38	6	66	61	4
2	100	54	25	12	27	82	10	89	80	33	3
3	2	95	1	96	32	6	40	31	9	59	2

The preferred days for attendance by each family are crucial to Santa because he does not want families to have an unhappy trip to his workshop. If a family does not attend on their most preferred day (i.e. choice 0), they are given financial compensation to make the trip positive. The list of compensation based on choice is shown below:

1. Choice 0: No consolation gifts
2. Choice 1: \$50 gift card
3. Choice 2: \$50 gift card + \$9 off Santa's Buffet for each family member
4. Choice 3: \$100 gift card + \$9 off Santa's Buffet for each family member
5. Choice 4: \$200 gift card + \$9 off Santa's Buffet for each family member
6. Choice 5: \$200 gift card + \$18 off Santa's Buffet for each family member
7. Choice 6: \$300 gift card + \$18 off Santa's Buffet for each family member
8. Choice 7: \$300 gift card + \$36 off Santa's Buffet for each family member
9. Choice 8: \$400 gift card + \$36 off Santa's Buffet for each family member
10. Choice 9: \$500 gift card + \$36 off Santa's Buffet for each family member + \$199 off the North Pole Helicopter Ride per family member
11. Otherwise: \$500 gift card + \$36 off Santa's Buffet for each family member + \$398 free North Pole Helicopter Ride per family member

Santa strongly prefers to have families attend on their preferred day, however, his workshop only has capacity for 300 people each day. In addition, his accountants have informed him that to make money, there must be at least 125 people attend. Thus, there must be between 125 and 300 people assigned for each day, or the proposed solution is infeasible. Further, his team of accountants has identified that there is a significant cost relating to the number of people that attend each day. This cost arises due to crowded gift shops, additional cleaning, and other miscellaneous costs. The accounting cost is defined as:

$$\text{accounting cost} = \sum_{d=100}^{1} \frac{(N_d - 125)}{400} N_d^{\left(\frac{1}{2} + \frac{|N_d - N_{d+1}|}{50}\right)} \tag{2.1}$$

where N_d is the occupancy of the current day, and N_{d+1} is the occupancy of the previous day. We are counting backward from Christmas until the first day. The first day is N_{100} and it is given that $N_{101} = N_{100}$. The accounting cost incentivizes a relatively stable number of attendees from day to day. However, by examining the preferred day for each family, which is shown in Figure 1, it is clear that this is challenging.

The optimization challenge is to minimize Santa's total cost for inviting all families to the workshop. Since the costs for both family preference and accounting are well-defined, optimization provides a method for identifying the best allocation of families across the 100

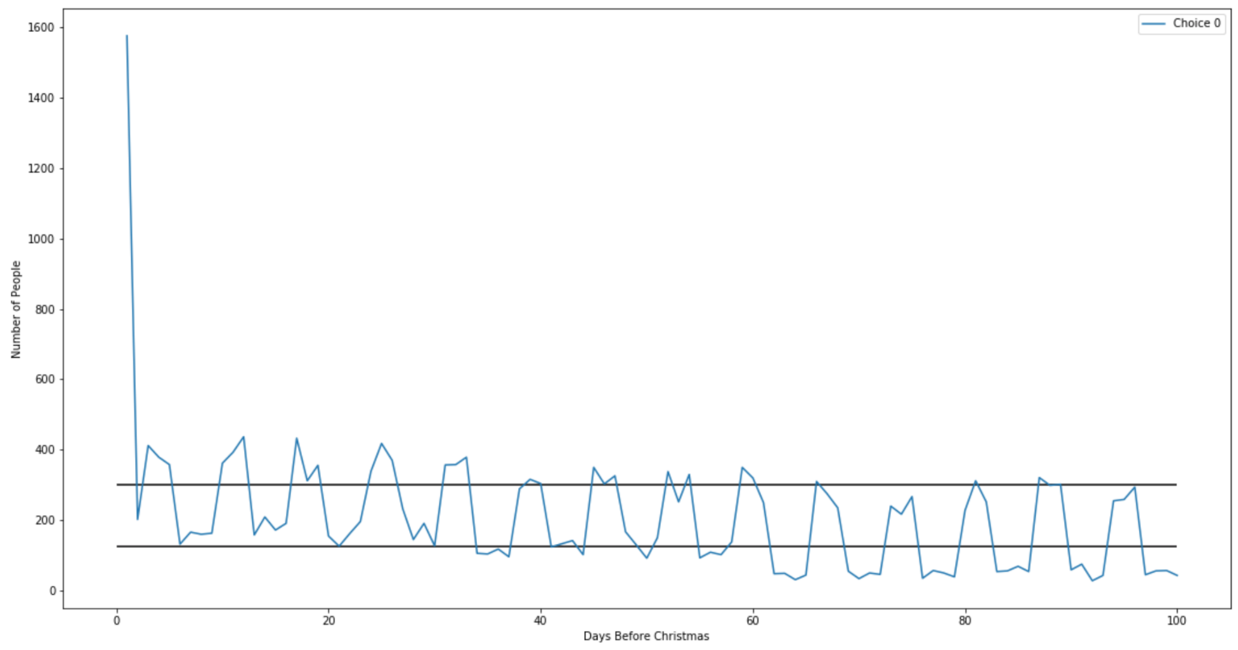


Figure 1: First Choice

days. Heuristics provide a fast approach to find solutions to this challenging problem, which gives Santa a good starting point for reducing his overall costs.

This problem was part of a Kaggle competition in December 2019 and solved to optimality using Mixed-Integer Programming. The optimal solution is $68,888.04 and was found after more than 2 days of runtime using Gurobi. We compare the quality and time of our solutions to the winner, however, Kaggle did not release the optimal allocation so we cannot compare how many families we accurately assigned.

3 Solution Approaches

While many of the heuristics presented in class were motivated by or explained through a TSP example, our problem is a resource allocation problem, which is fundamentally different from the TSP. Instead of identifying the shortest path by changing which cities are on the route, our problem involves changing which family is invited to the North Pole on a given day. We also have to incorporate the preferences of each family and the constraints on the number of people that can be invited for a given day. To do this, we developed helper functions along with the three heuristics that we tested for our problem. In this section, we first explain our rationale for the helper functions and then proceed to describe the heuristics we applied for our problem.

3.1 Helper Functions

To assist with our heuristic implementations, we created two helper functions: a relaxed-constraint cost function and a greedy initialization function. The first, which is largely adapted from the original Kaggle competition, provides an alternative cost formulation that significantly penalizes solutions that do not satisfy the constraints. The key constraint for this problem is the number of people who are invited each day must be between 125 and 300, inclusive. While it is possible to check for this before assigning a cost for a given solution, implementing a large penalty (say 10e8) within the cost function is a computationally efficient way to discourage infeasible solutions. We use this method and find that a penalty of 10e8 for each day that violates the constraints is appropriate. For reference, randomly assigning 50 families to each of the 100 possible days leads to a cost of 10e7. So the penalty we propose is sufficiently large to discourage infeasible solutions.

While the cost function is robust against infeasible solutions, finding a starting feasible solution is difficult. One approach is to randomly assign 50 families to each of the 100 days. However, the downside of this approach is information about the family's preferences is not included in this initial solution. This would lead to very slow convergence because in the worst case 5000 families would need to be moved. Instead of a random initialization, we use a greedy random initialization that begins with all families assigned to their top preference. As shown in Figure 1, most families want to attend close to Christmas (i.e. near day 0) and fewer want to attend farther from Christmas (i.e. near day 100). The black lines in Figure 1 represent 125 and 300 people, which are the hard constraints on attendance. Our approach is to randomly assign families on days with over 300 people to days with less than 125 people. For example, a family who has a preference for attending on day 0 will have a probability (ϵ) to be moved from day 0. If the random number generated is less than ϵ, they will be randomly assigned to one of the days with less than 125 people. The greedy initialization continues until all days meet the occupancy constraints. For heuristics that require a starting population (i.e. Genetic Algorithm) instead of a single starting solution, we run the greedy initialization n times until we have a starting population of size n. It is important to note that these initial solutions will not be the same, because the chance a family stays on their preferred day is probabilistic and if the family is moved, the new day they are moved to is also probabilistically determined.

3.2 Heuristics

For our problem, we test three heuristics: Random Search, Guided Local Search, and the Genetic Algorithm. We chose random search as a baseline to compare the other heuristics. As random search is comparatively bad to most other heuristics, this provides a meaningful way to assess the performance increase by the use of smarter heuristics. The next heuristic we selected was guided local search. Our reasoning for using this heuristic is the availability of domain knowledge (i.e. the family's preferences) to guide the optimization. Finally, we selected the Genetic Algorithm, which is typically a higher-performing method compared to random search and guided local search. This will allow us to make statements about the effectiveness of using a higher-performing heuristic for our problem.

3.2.1 Random Search Day

We implement two versions of random search to test the effectiveness of each. In the first random search method, for each iteration, all families assigned to day j are reassigned to day k, and simultaneously all families assigned to day k are reassigned to day j. For each day j, there is a probability (defined in code as "swap_rate") that day j will be selected for swapping. If selected, another day (k) will be randomly selected for reassignment. If the new assignment reduces the overall cost, the assignment is made permanent. However, if the new assignment leads to a higher cost it is removed.

3.2.2 Random Search Family

Similar to the first version, we also develop a random search that occurs at the family level. A family assigned to day j is randomly assigned to day k and if the cost is reduced, the assignment is made permanent. "swap_rate" is also used here to randomly determine if a family is selected to be moved.

3.2.3 Guided Local Search

Extending upon random search, we use guided local search, which is guided by family preferences. In each iteration, we choose some number of families n (in code as "num_fam_swap") to potentially swap at each iteration. For each n, we randomly pick a family to assign to a new day. However, unlike random search which randomly assigns them to a new day, in guided local search, we try to assign them to a day that they prefer more. For example, if family i is currently on day j, which is their 3rd preference, we check if they can be moved to a day which is their 1st or 2nd preference, otherwise family i stays on day j. If moving a family to a new day reduces the overall cost, they are assigned to that day. If no improvement in cost can be found, the family remains on the previously assigned day. To speed up the computation, we implement two strategies. First, we begin by checking the most preferred day and continue until the least preferred day is checked. If the family can be assigned to a day on a preferred day, we do not check additional days because these will necessarily lead to higher overall costs. Second, if we find that the family is assigned to the day currently being checked we do not perform any further checks for the family. Checking any additional days will also necessarily lead to higher overall costs.

3.2.4 Genetic Algorithm

The final heuristic that we implement is the genetic algorithm. For GA, we chose to use a 5000x1 chromosome, where each value in the chromosome is the day that the family attends Santa's workshop. We begin by initializing a population and checking the fitness (i.e. cost) of the population. Here, lower fitness values represent higher-quality solutions, since the objective is to minimize total cost. We use 2-point crossover for reproduction and single point mutation. The mutations are informed by each family's preferences, so a mutation of family i will mutate to one of the 10 preferred days of family i. We assign probabilities to each day, such that the most preferred day has more probability mass than the second most preferred day and so on. Once the crossover and mutation are completed, the fitness of the new population is calculated. We select the top n individuals as parents for the next generation, where n is a tunable parameter in our heuristic. We investigated choosing parents probabilistically based on their fitness values, however, this led to very poor solution quality. In the beginning iterations, some parents would have infeasible solutions. If these parents were selected for reproduction, the heuristic would be unable to find a feasible solution after a few iterations. Because of this, we instead choose the top n individuals to use as parents for the next generation.

4 Results

We implement each of the heuristics detailed in Section **??** and examine the effect of various hyperparameters in regards to solution quality and computation time. Unless otherwise noted, each test was completed 10 times for 1000 iterations.

4.1 Random Search Day

The first optimization heuristic we use is random search day. We find that this algorithm performed the worst of all the heuristics but we use it as a reference to benchmark the improvements made with the guided local search and genetic algorithm. In Table 2, we explore how changing the swap rate for greedy random search day can decrease the cost for Santa. Total cost only slightly decreases when increasing the swap rate but the time increases exponentially. We expect a small decrease in total cost because with more swaps there is a higher chance that one of the random swaps will lead to an improvement in the solution. For this heuristic, we do not test both the greedy and random initialization, because there would be no improvement for the random initialization. Since all days are initialized to 50 families in the random initialization, swapping families would lead to only a marginal decrease in total cost.

Table 2: Effect of varying the swap rate for random search day.

Swap Rate	0.1	0.3	0.5
Average Time	61.85	213.54	323.98
Best Time	60.53	181.57	304.01
Worst Time	65.67	352.41	343.13
Average Cost	4656939.70	4548308.74	4462029.16
Best Cost	4434598.00	4319533.90	4274140.16
Worst Cost	4914232.58	4854436.32	4935572.18

4.2 Random Search Family

In Table 3 we show the results of the random search family heuristic. We test whether using a greedy initialization improves the performance of the random search family heuristic. We find that the greedy initialization incorporated into the random search family heuristic decreases the time and provides a significantly lower cost for all swap rates. In Table 3, we find that increasing the swap rate linearly increases the total computation time for the heuristic. A larger swap rate tends to produce better results at the cost of more computation time. Thus, if Santa decides to use this heuristic, he must weigh the benefit of a higher quality solution to the computation cost.

Table 3: Effect of varying the swap rate for random search family.

Initialization	Greedy			Random		
Swap Rate	0.0025	0.005	0.01	0.0025	0.005	0.01
Average Time	87.02	178.30	366.21	114.37	341.53	656.76
Best Time	81.30	164.90	346.47	107.13	241.86	533.53
Worst Time	91.33	204.84	404.21	134.14	948.25	1014.06
Average Cost	3011154.64	2497276.08	1879373.14	8665322.45	7127551.76	4958784.36
Best Cost	2953132.53	2473589.46	1829475.85	8570963.08	7029788.14	4859625.86
Worst Cost	3060902.31	2556058.68	1933661.47	8722342.21	7244433.75	5124672.87

4.3 Guided Local Search

In Table 4 we report our findings on using the guided local search heuristic to find the optimal cost for Santa's Workshop. We find that the greedy initialization did not reduce the total cost for guided local search. One explanation for this result is that the greedy initialization begins in a local optimum that is difficult to escape, while the random initialization is freer to explore the solution space. Table 4 suggests that increasing the number of family's swapped does not significantly reduce costs. Swapping more families may decrease solution quality because guided local search is unable to make small incremental changes towards the optimal solution. Increasing the number of families swapped increases the time to run the heuristic exponentially. Thus, there is not much benefit to increasing the number of families swapped. We recommend that Santa not incorporate the greedy initialization and swap fewer families when using guided local search.

Table 4: Effect of swapping varying number of families for each iteration of guided local search.

Initialization	Greedy			Random		
Num Family Swap	50	100	200	50	100	200
Average Time	307.99	619.57	4278.15	326.39	1613.48	3013.19
Best Time	298.27	598.33	1230.26	309.97	571.05	1157.92
Worst Time	319.62	636.66	30760.61	341.75	6130.30	9894.24
Average Cost	1472201.80	1474506.74	1452171.29	1021369.54	1026363.05	1001950.64
Best Cost	1434081.48	1397889.21	1405674.65	976536.18	974873.10	935984.52
Worst Cost	1525239.01	1504736.55	1496172.42	1098208.07	1134194.37	1068883.05

4.4 Genetic Algorithm

Dissimilar to the previous heuristics, the Genetic Algorithm has multiple tunable hyperparameters that we investigate, specifically the number of parents, the population size, and the mutation rate. We display the results of these parameter tests in Tables 5, 6, and 7, respectively. As a baseline, we set the number of parents to 10, the population size to 30, and the mutation rate to 0.01. For varying hyperparameter tests, the remaining two hyperparameters are set to the baseline values. We find that, in general, the greedy initialization helps the genetic algorithm find higher-quality solutions compared to the random initialization. Unlike guided local search, the genetic algorithm can escape the local optima and search closer to the global optima. As the number of parents increases, the total cost increases, and the computation time decreases. This is due to too few children being populated to effectively explore the space. Additionally, the reproduction of children is much more time-intensive compared to assigning a parent to the next population so we also see a decrease in the computation time as the number of parents increases. We find that increasing the population size leads to improved solutions. This can be attributed to having more children which allows for a greater probability of reproducing high-performing children. From this result shown in Table 6 and the result shown in Table 5, we conclude that as the ratio of population size and the number of parents increases, we tend to find higher-quality solutions. This shows that the main driver of improved solution quality in the genetic algorithm, for this problem, comes from the reproduction, crossover, and mutation steps. Finally, we test varying the mutation rate. Table 7 shows that the lowest cost is obtained with a mutation rate of 0.01 but it has a slightly longer run time. We find that our initial value is the highest-performing mutation rate. A mutation rate that is too low does not perform well, due to too few chances to generate high-performing offspring. A mutation rate that is too high leads to large changes in the individuals, which prevents the genetic algorithm from learning.

One additional test performed was to increase the number of iterations for the genetic algorithm. As can be seen from the results of the genetic algorithm, the heuristic takes a very long time to run, even for the small values of hyperparameters we chose. We performed one run with the following hyperparameters: population size 80, number parents 10, mutation rate 0.01, iterations 40000. This test took 13.31 hours and yielded a final cost of 250,000. This result is an order of magnitude better than any other test we performed. So, with more time it is beneficial to use the genetic algorithm, however for this problem the convergence is extremely slow.

Table 5: Effect of changing the number of parents that are chosen for the next generation for the genetic algorithm.

Initialization	Greedy			Random		
Number of Parents	5	10	20	5	10	20
Average Time	382.26	360.58	350.53	617.00	527.95	518.38
Best Time	359.83	353.01	344.94	534.48	509.43	506.50
Worst Time	438.98	375.87	360.07	773.51	541.87	533.52
Average Cost	1097260.58	1436775.09	2588371.25	2556314.17	3370445.28	6532292.24
Best Cost	1038609.94	1340482.68	2424103.33	2447989.92	3214728.64	5997786.77
Worst Cost	1204982.79	1549722.83	2879679.19	2650546.10	3491944.57	6805878.06

5 Summary of Major Findings

We summarize our major findings below:

- We find that the genetic algorithm was the best heuristic for solving our research

Table 6: Effect of varying population size for the genetic algorithm.

Initialization	Greedy			Random		
Population Size	20	30	40	20	30	40
Average Time	278.56	408.30	3939.34	375.17	547.33	769.38
Best Time	246.67	365.76	524.32	355.66	531.56	699.45
Worst Time	295.22	452.36	34086.68	391.81	571.70	942.36
Average Cost	2073006.23	1449354.54	1150444.14	5266768.11	3396212.62	2719246.85
Best Cost	1924112.74	1367930.45	1096639.88	4718575.75	3207402.06	2615420.03
Worst Cost	2199036.37	1539936.75	1246558.62	6027044.73	3710643.95	2907312.89

Table 7: Effect of changing the mutation rate for the genetic algorithm.

Initialization	Greedy			Random		
Mutation Rate	0.001	0.01	0.1	0.001	0.01	0.1
Average Time	456.38	630.51	510.68	587.74	596.21	471.34
Best Time	356.09	362.66	449.50	568.43	519.12	464.69
Worst Time	667.25	1651.03	595.26	601.83	788.41	478.04
Average Cost	1914617.36	1456659.24	3316072.42	6897307.85	3330038.00	8112442.44
Best Cost	1785883.72	1385023.21	3109924.52	6845988.47	3180167.27	7804385.72
Worst Cost	2067404.30	1559655.50	3505076.19	6987254.20	3440080.44	8551568.78

problem. In addition to having the lowest overall test, an additional test for larger population size and 40000 iterations led to a total cost of 250,000.

- We find that our greedy initialization helps on random search family and the genetic algorithm, but does not help improve performance for guided local search. Our idea for this result is that guided local search is stuck in the local optima created by the greedy initialization and is unable to escape.
- As the number of families to swap increases, guided local search becomes much slower than the other methods. However, in general, there is not a significant benefit to increasing the number of families to swap. So guided local search can be implemented with a small number of families to swap and still yield good results.
- The performance of the genetic algorithm is very sensitive to the mutation rate for our problem. This is a potential downside to the genetic algorithm because small changes in the mutation rate can lead to very different convergence patterns.

Heuristics_Project_Code

April 26, 2020

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
import time
```

[2]:
```
family_data = pd.read_csv("../Data/family_data.csv", index_col= "family_id")
sample_submission = pd.read_csv("../Data/sample_submission.csv")
```

[3]:
```
# For our prediction df, we assume it has 3 columns, family id, num_people, and␣
 ↪assigned day.
# We can initialize this df at the beginning and make updates to assigned_day as␣
 ↪algo progresses.
sample_submission = pd.concat([sample_submission, family_data.iloc[:, -1]], axis␣
 ↪= 1)
sample_submission = sample_submission[['family_id', 'n_people', 'assigned_day']]
```

[4]:
```
family_data.head()
```

[4]:
```
           choice_0  choice_1  choice_2  choice_3  choice_4  choice_5  \
family_id
0                52        38        12        82        33        75
1                26         4        82         5        11        47
2               100        54        25        12        27        82
3                 2        95         1        96        32         6
4                53         1        47        93        26         3

           choice_6  choice_7  choice_8  choice_9  n_people
family_id
0                64        76        10        28         4
1                38         6        66        61         4
2                10        89        80        33         3
3                40        31         9        59         2
4                46        16        42        39         4
```

[5]:
```
sample_submission.head()
```

[5]:
```
   family_id  n_people  assigned_day
0          0         4           100
1          1         4            99
2          2         3            98
3          3         2            97
4          4         4            96
```

[6]:
```
family_data.shape[0]
```

[6]:
```
5000
```

[7]:
```
def get_pref_cost(ch, fam_size):
    # Calculate the penalty for not getting top preference
    # This comes directly from the Kaggle competition
    if ch == 0:
        return 0
    elif ch == 1:
        return 50
    elif ch == 2:
        return 50 + 9 * fam_size
    elif ch == 3:
        return 100 + 9 * fam_size
    elif ch == 4:
        return 200 + 9 * fam_size
    elif ch == 5:
        return 200 + 18 * fam_size
    elif ch == 6:
        return 300 + 18 * fam_size
    elif ch == 7:
        return 300 + 36 * fam_size
    elif ch == 8:
        return 400 + 36 * fam_size
    elif ch == 9:
        return 500 + 36 * fam_size + 199 * fam_size
    else:
        return 500 + 36 * fam_size + 398 * fam_size
```

[8]:
```
def get_cost(fam_data, pred):

    # Preference Cost
    preference_cost = 0

    # Calculate preference cost for each family
    for i in range(fam_data.shape[0]):

        # Find the day they were assigned and check to see if this was one of␣
↪their choices
```

```
        assigned_day = pred.iloc[i, 2]
        if (assigned_day in fam_data.iloc[i, :-1].values):

            # If it was a choice, find out which number choice
            choice = np.where(fam_data.iloc[i, :-1].values == assigned_day)[0][0]
        else:
            choice = 10

        # Compute the preference cost for the family
        preference_cost += get_pref_cost(ch=choice, fam_size=fam_data.iloc[i,␣
↪-1])

    # Accounting Cost
    accounting_cost = 0

    # Need to get number of people assigned to each day
    people_per_day = pred.groupby(by="assigned_day").sum()
    people_per_day = people_per_day.sort_index(axis=0, ascending=False)

    # Calculate daily accounting cost
    for day in range(100):
        num_p_today = people_per_day.iloc[day, 1]
        people_difference = np.abs(people_per_day.iloc[day, 1] - people_per_day.
↪iloc[day - 1, 1])

        # Special case for day 0 (i.e. 100 days before christmas)
        if day == 0:
            accounting_cost += ((num_p_today - 125.0) / 400.0) * num_p_today **␣
↪(0.5)
        else:
            accounting_cost += ((num_p_today - 125.0) / 400.0) * num_p_today **␣
↪(0.5 + (people_difference / 50.0))

    return (preference_cost + accounting_cost)
```

[9]:
```
# Testing Cost function on known data
# This should equal 10641498.40
start = time.time()
total_cost = get_cost(fam_data=family_data, pred=sample_submission)
print("Total Cost is %s" % (np.round(total_cost, decimals=2)))
print("Total Time is %s" % (np.round(time.time() - start, decimals=3)))
```

```
Total Cost is 10641498.4
Total Time is 1.441
```

0.1 Problem Information

From Kaggle - Santa has exciting news! For 100 days before Christmas, he opened up tours to his workshop. Because demand was so strong, and because Santa wanted to make things as fair as possible, he let each of the 5,000 families that will visit the workshop choose a list of dates they'd like to attend the workshop.

Now that all the families have sent Santa their preferences, he's realized it's impossible for everyone to get their top picks, so he's decided to provide extra perks for families that don't get their preferences. In addition, Santa's accounting department has told him that, depending on how families are scheduled, there may be some unexpected and hefty costs incurred.

Santa needs the help of the Kaggle community to optimize which day each family is assigned to attend the workshop in order to minimize any extra expenses that would cut into next years toy budget! Can you help Santa out?

The total number of people attending the workshop each day must be between 125 - 300; if even one day is outside these occupancy constraints, the submission will error and will not be scored.

Our notes - We are given each family's 10 preferred days to visit Santa. If the family does not visit Santa on the first day, then Santa incurs a preference cost to accomodate them. There is also an accounting cost that penalizes large variations in the number of people per day.

Using the family's first choice as a starting point provides a good method for initializing the optimization. This is because the more families that are given their top choice, the lower the overall cost will be.

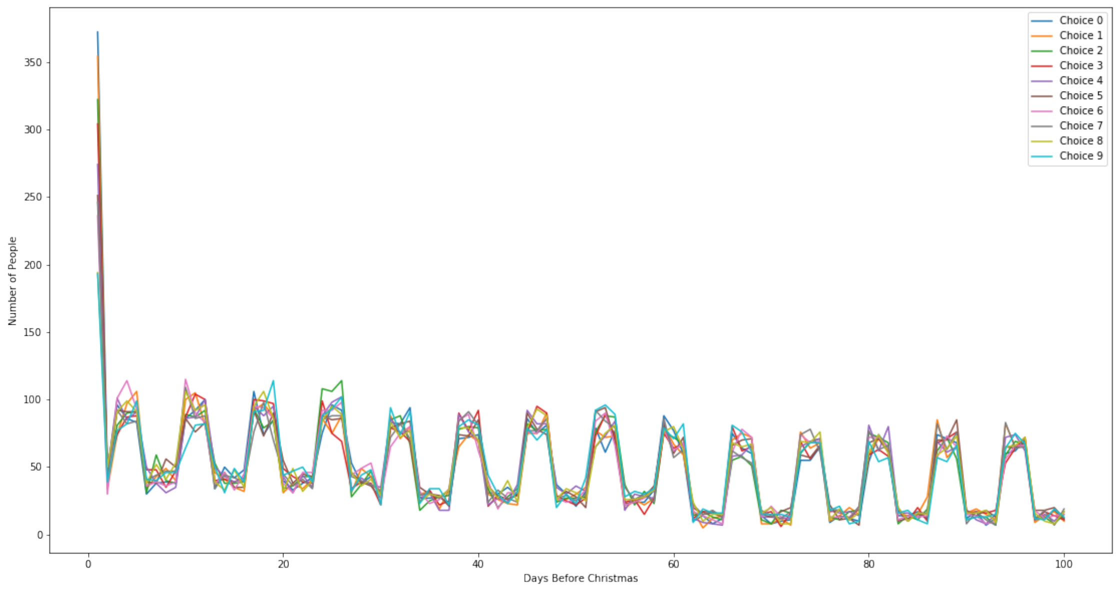
A first thing to examine is the distribution of the preferred days (by number of people).

[10]:
```
family_preferences = pd.read_csv("../Data/family_data.csv", index_col=␣
↪"family_id")
```

[11]:
```
family_preferences.head()
print(family_preferences.shape[1])
```

```
11
```

[12]:
```
plt.figure(figsize=(19, 10))
for i in range(family_preferences.shape[1] - 1):
    choice_count = family_preferences.iloc[:,i].value_counts().sort_index()
    plt.plot(choice_count, label = "Choice %s" % i)
plt.legend()
plt.xlabel("Days Before Christmas")
plt.ylabel("Number of People")
plt.show()
```

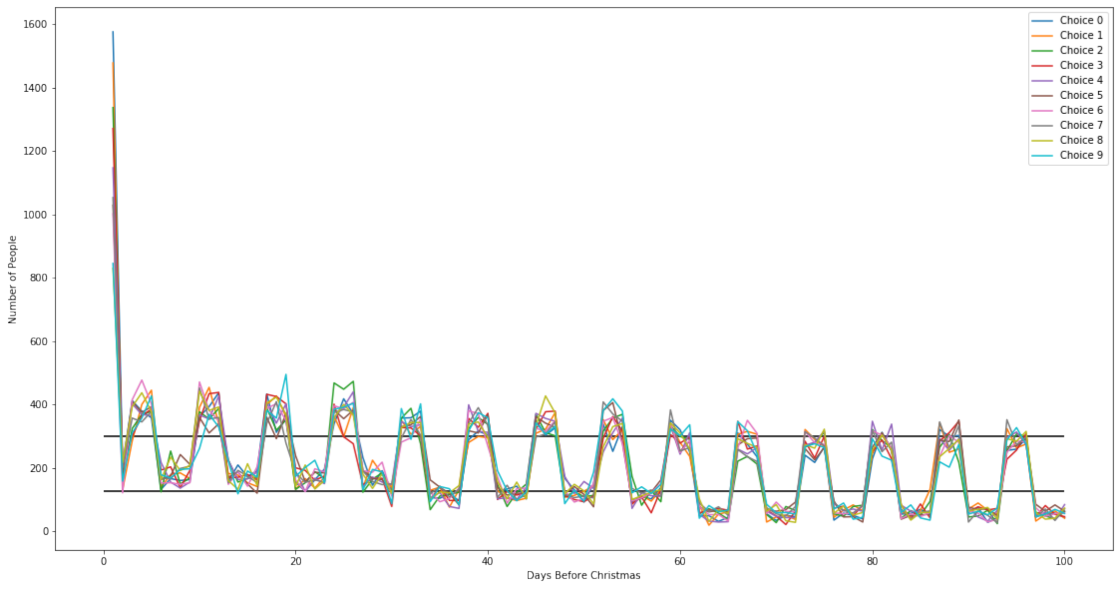


Here, day 0 is Christmas Day, so many families want to visit on Christmas Eve. For our heuristic approach, we will need to incorporate some penalty for being on Day 1. Since so many people want to attend Santa's workshop on Day 1, it will be beneficial to give them their 2nd choice.

Another note is that there are spikes on the Weekends, and lower values on weekdays. So another challenge will be ensuring that 125 people are attending on weekdays, especially far from Christmas.

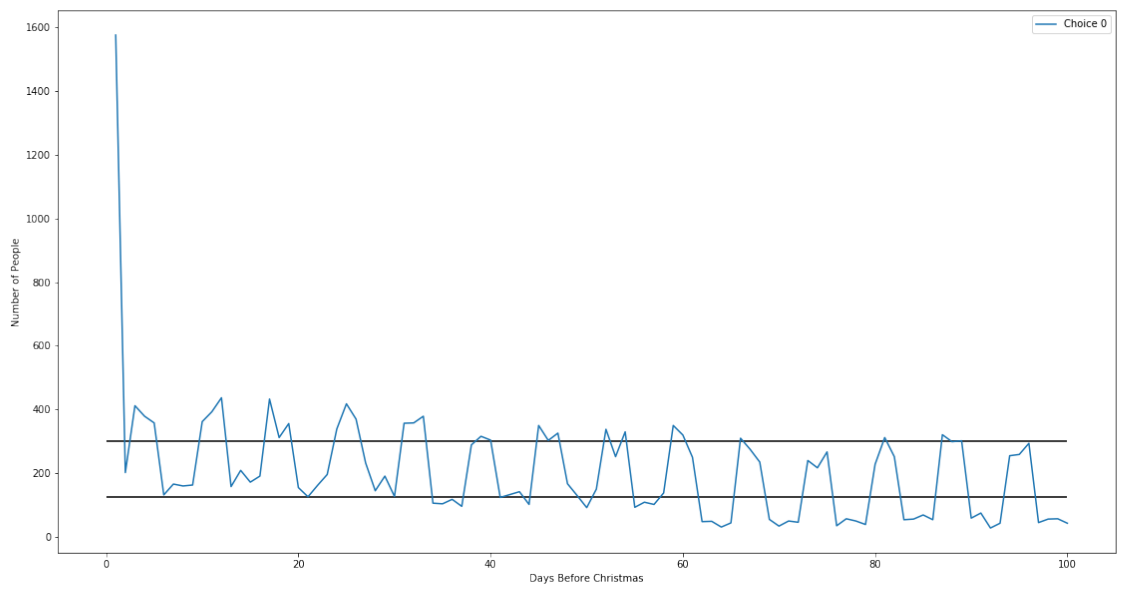
Next, let's look at the number of people that request each day. The previous figure only examined families, but families have different number of people.

[13]:
```
plt.figure(figsize=(19, 10))
for i in range(family_preferences.shape[1] - 1):
    choice_count = family_preferences.iloc[:,[i,-1]]
    choice_count = choice_count.groupby(choice_count.columns[0]).sum()
    plt.plot(choice_count, label = "Choice %s" % i)
plt.legend()
plt.xlabel("Days Before Christmas")
plt.ylabel("Number of People")
plt.hlines(300, 0, 100, colors = "k")
plt.hlines(125, 0, 100, colors = "k")
plt.show()
```



The two lines denote 125 and 300 people, respectively. For each day, the total number of people must be between these two lines. Let's plot only the first choice and see how it is distributed.

[14]:
```
plt.figure(figsize=(19, 10))
for i in range(1):
    choice_count = family_preferences.iloc[:,[i,-1]]
    choice_count = choice_count.groupby(choice_count.columns[0]).sum()
    plt.plot(choice_count, label = "Choice %s" % i)
plt.legend()
plt.xlabel("Days Before Christmas")
plt.ylabel("Number of People")
plt.hlines(300, 0, 100, colors = "k")
plt.hlines(125, 0, 100, colors = "k")
plt.show()
```



We have some days that have the required number of people, but days closer to Christmas tend to have more than 300 people and days far away from Christmas tend to have fewer people. In total, there are 21003 people, so an average of 210.03 per day.

[15]:
```
print(sum(choice_count.iloc[:,0]))
```

```
21003
```

0.2 Helper Functions

Two helper functions are defined below to help for heuristic evaluation

1) Function to evalutate the cost quickly. This code was provided by Kaggle for the competition and was much faster than the code we initially developed for calculating the cost
2) Function to get to a feasible point quickly. Instead of starting at a random location, it is favorable to start with each family's preferred day to visit. However, this leads to some days having too many or too few people. We fix this by randomly assigning people on days with too many people to days with fewer people. Another possible starting point would be too randomly assign people to individual days, but we believe that this starting point will be more beneficial.

[16]:
```
def get_cost(pred, fs_d, ch_d):

    days = list(range(100,0,-1))

    penalty = 0

    # We'll use this to count the number of people scheduled each day
```

```
daily_occupancy = {k:0 for k in days}

# Looping over each family; d is the day for each family f
for f, d in enumerate(pred):

    # Using our lookup dictionaries to make simpler variable names
    n = fs_d[f]
    choice_0 = ch_d['choice_0'][f]
    choice_1 = ch_d['choice_1'][f]
    choice_2 = ch_d['choice_2'][f]
    choice_3 = ch_d['choice_3'][f]
    choice_4 = ch_d['choice_4'][f]
    choice_5 = ch_d['choice_5'][f]
    choice_6 = ch_d['choice_6'][f]
    choice_7 = ch_d['choice_7'][f]
    choice_8 = ch_d['choice_8'][f]
    choice_9 = ch_d['choice_9'][f]

    # add the family member count to the daily occupancy
    daily_occupancy[d] += n

    # Calculate the penalty for not getting top preference
    if d == choice_0:
        penalty += 0
    elif d == choice_1:
        penalty += 50
    elif d == choice_2:
        penalty += 50 + 9 * n
    elif d == choice_3:
        penalty += 100 + 9 * n
    elif d == choice_4:
        penalty += 200 + 9 * n
    elif d == choice_5:
        penalty += 200 + 18 * n
    elif d == choice_6:
        penalty += 300 + 18 * n
    elif d == choice_7:
        penalty += 300 + 36 * n
    elif d == choice_8:
        penalty += 400 + 36 * n
    elif d == choice_9:
        penalty += 500 + 36 * n + 199 * n
    else:
        penalty += 500 + 36 * n + 398 * n

# for each date, check total occupancy
#  (using soft constraints instead of hard constraints)
```

```
    for _, v in daily_occupancy.items():
        if (v > 300) or (v < 125):
            penalty += 100000000

    # Calculate the accounting cost
    # The first day (day 100) is treated special
    accounting_cost = (daily_occupancy[days[0]]-125.0) / 400.0 *␣
↪daily_occupancy[days[0]]**(0.5)
    # using the max function because the soft constraints might allow occupancy␣
↪to dip below 125
    accounting_cost = max(0, accounting_cost)

    # Loop over the rest of the days, keeping track of previous count
    yesterday_count = daily_occupancy[days[0]]
    for day in days[1:]:
        today_count = daily_occupancy[day]
        diff = abs(today_count - yesterday_count)
        accounting_cost += max(0, (daily_occupancy[day]-125.0) / 400.0 *␣
↪daily_occupancy[day]**(0.5 + diff / 50.0))
        yesterday_count = today_count

    penalty += accounting_cost

    return penalty
```

[17]:
```
def get_days_over_n(df, n):
    df = df.groupby(df.columns[-1]).sum().sort_values(by = "n_people")
    return df.index[df['n_people'] > n].tolist()

def get_days_under_n(df, n):
    df = df.groupby(df.columns[-1]).sum().sort_values(by = "n_people")
    return df.index[df['n_people'] < n ].tolist()

def check_days(df):
    return len(get_days_over_n(df, 300)) + len(get_days_under_n(df, 125))

def greedy_random_assignment(pred_df, eps):
    # Idea is to initialize all families with their first choice and then␣
↪randomly assign families
    # on days where there are too many people (i.e. over 300) with preference␣
↪towards assigning them to days
    # with less than 125 people.
    num_day_list = 1

    # Find an initial assignment that is feasible
    while num_day_list > 0:
        days_over = get_days_over_n(pred_df, 300)
        days_under = get_days_under_n(pred_df, 125)

        # If there are no days with over 300 people, take people from days with␣
↪more than 250 people
        if ((len(days_over) == 0) & (len(days_under) > 0)):
            days_over = get_days_over_n(pred_df, 250)

        # If there are no days with under 125 people, give people to days with␣
↪less than 175 people
        if ((len(days_over) > 0) & (len(days_under) == 0)):
            days_under = get_days_under_n(pred_df, 200)

        for day in days_over:
            fam_assigned = pred_df.index[pred_df['assigned_day'] == day].tolist()
            for fam in fam_assigned:
                if np.random.random() < eps:
                    pred_df.iloc[fam, -1] = np.random.choice(days_under)

        num_day_list = check_days(pred_df)
    return pred_df
```

0.3 Heuristic 1: Random Search

For our first heuristic, we initially assign all families to their first choice. We then probabilistically randomly assign families on days with more than 300 people to days with less than 125 people. In this heuristic, we continue moving people until a feasible solution is found. Once a feasible solution is found, we use stochastic 2-opt to successively find new feasible solutions. We move all people assigned from day j to day k and vice versa. Since the original solution was feasible, each stochastic 2-opt is also guaranteed to be feasible.

This solution will not be great, but provides a baseline to compare future heuristics. An additional point to note is that the optimal cost for this problem is 68888.04.

[18]:
```
def random_search(assign_df, epsilon, num_iterations, swap_rate, fam_size_dict,␣
↪choice_dict):

    # Get an initial feasible assignment
    assign_df = greedy_random_assignment(pred_df=assign_df, eps=epsilon)
    best_cost = 10e10
    for i in range(num_iterations):
        for j in range(1,100):
            # Probabilistic 2-opt
            if np.random.random() < swap_rate:

                # Assign all people on day j to another day k and vice versa
                other_days = [x for x in range(1,100) if x != j]
                new_day = np.random.choice(other_days)
                assign_df = assign_df.replace({j:new_day, new_day:j})

                # Only want to accept the swap if it reduces the overall cost
                new_cost = get_cost(pred = assign_df["assigned_day"].tolist(),␣
↪fs_d = fam_size_dict, ch_d = choice_dict)
                if new_cost < best_cost:
                    best_df = assign_df.copy(deep=True)
                    best_cost = new_cost
                else:
                    # Reverse the assignment
                    assign_df = assign_df.replace({j:new_day, new_day:j})
        if i % 50 == 0:
            print(best_cost)

    return best_df, best_cost
```

0.4 Heuristic 1(a): Random Search Family

[19]:
```
def random_search_2(assign_df, epsilon, num_iterations, swap_rate,␣
↪fam_size_dict, choice_dict):

    # Get an initial feasible assignment
    assign_df = greedy_random_assignment(pred_df=assign_df, eps=epsilon)
    assigned_list = assign_df["assigned_day"].tolist()
    best_cost = 10e10
    for i in range(num_iterations):
        for j in range(1,5000):
            # Probabilistic 2-opt
            if np.random.random() < swap_rate:

                cur_day = assigned_list[j]

                # Assign all people on day j to another day k and vice versa
                other_days = [x for x in range(1,101) if x != cur_day]
                new_day = np.random.choice(other_days)
                assigned_list[j] = new_day

                # Only want to accept the swap if it reduces the overall cost
                new_cost = get_cost(pred = assigned_list, fs_d = fam_size_dict,␣
↪ch_d = choice_dict)
                if new_cost < best_cost:
                    best_assignment = assigned_list
                    best_cost = new_cost
                else:
                    # Reverse the assignment
                    assigned_list[j] = cur_day
        if i % 50 == 0:
            print(best_cost)

    return best_assignment, best_cost
```

0.5 Heuristic 2: Guided Local Search

Idea is to give preference to family choices. Try to put them on the best day to minimize overall cost.

```
[20]: def guided_local_search(assign_df, epsilon, num_iterations, fam_size_dict,␣
      ↪choice_dict, greedy, num_fam_swap):

          # Get an initial feasible solution
          if greedy:
              assign_df = greedy_random_assignment(pred_df=assign_df, eps=epsilon)

          # Initialize the assignment and cost based on the feasible solution
          best_assignment = assign_df["assigned_day"].tolist()
          best_cost = get_cost(pred = best_assignment, fs_d = fam_size_dict, ch_d =␣
      ↪choice_dict)
          for i in range(num_iterations):

              # I just chose to swap 100 families each iteration. This could also be␣
      ↪probabilistic
              for j in range(num_fam_swap):

                  # Pick a random family
                  family = np.random.choice(range(5000))
                  for pick in range(10):

                      # Try to put them on the preferred day
                      choice = choice_dict[f'choice_{pick}'][family]

                      # If they are already on a day they choose, break out of the␣
      ↪loop. No improvement can be found
                      if choice == best_assignment[family]:
                          break

                      temp_assignment = best_assignment.copy()
                      temp_assignment[family] = choice
                      new_cost = get_cost(pred = temp_assignment, fs_d =␣
      ↪fam_size_dict, ch_d = choice_dict)
                      if new_cost < best_cost:
```

```
                    best_assignment = temp_assignment.copy()
                    best_cost = new_cost

                    # If a new best cost is found, there is no need to keep␣
↪checking their choices.
                    # Giving them a less preferred choice will increase cost
                    break

        if i % 50 == 0:
            print(best_cost)
    return best_assignment, best_cost
```

0.6 Heuristic 3: Genetic Algorithm

[21]:
```
def genetic_algorithm(population_size, assign_df, epsilon, fam_size_dict,␣
↪choice_dict, parents_count, mutation_rate, num_iter, family_matrix):

    # Initial population has 5000 entries, one for each family
    population = np.zeros((population_size, 5000))
    new_population = np.zeros((population_size, 5000))
    fitness = np.zeros((population_size, 1))
    cumulative_fitness = np.zeros((population_size, 1))
    best_cost = 10e10

    for iteration in range(num_iter):
        if iteration == 0:

            # Initialize population and fitness scores
            for i in range(population_size):
                temp_df = assign_df.copy(deep = True)
                temp_assignment = greedy_random_assignment(pred_df=temp_df,␣
↪eps=epsilon)["assigned_day"]
                population[i,:] = temp_assignment
                fitness[i] = get_cost(pred=temp_assignment, fs_d=fam_size_dict,␣
↪ch_d=choice_dict)
        else:
            # Add parents to new population
            for i in range(parents_count):
                new_population[i,:] = population[new_parents[i], :]

            # Crossover to make offspring: Using 2 point crossover
            for i in range(parents_count, population_size):
                parent_1 = np.random.choice(range(parents_count))
                parent_2 = np.random.choice([x for x in range(parents_count) if␣
↪x != parent_1])
```

```
                crossover_1 = np.random.choice(range(1,4999))
                crossover_2 = np.random.choice([x for x in range(1,4999) if x !=␣
↪crossover_1])
                if crossover_1 > crossover_2:
                    crossover_2, crossover_1 = crossover_1, crossover_2
                new_population[i, :crossover_1] = new_population[parent_1, :
↪crossover_1]
                new_population[i, crossover_1:crossover_2] =␣
↪new_population[parent_2, crossover_1:crossover_2]
                new_population[i, crossover_2:] = new_population[parent_1,␣
↪crossover_2:]

            # Mutation: Calculate number of mutations and randomly apply then
            num_mutations = int(mutation_rate * population_size * 5000)
            # Assign a decreasing probability that their worse choices are␣
↪selected
            choice_prob = [0.181818182,0.163636364,0.145454545,0.127272727,0.
↪109090909,0.090909091,0.072727273,0.054545455,0.036363636,0.018181818]
            fam_mutate = np.random.choice(range(5000), size = num_mutations)
            mutated_choice = np.random.choice(range(10), size = num_mutations, p␣
↪= choice_prob)
            chrom_choice = np.random.choice(range(population_size), size =␣
↪num_mutations)
            for mut in range(int(num_mutations)):
                new_population[chrom_choice[mut], fam_mutate[mut]] =␣
↪family_matrix.iloc[fam_mutate[mut], mutated_choice[mut]]

            # Calculate fitness
            for i in range(population_size):
                fitness[i] = get_cost(pred=new_population[i,:].astype(int),␣
↪fs_d=fam_size_dict, ch_d=choice_dict)

            # Assign new population to population
            population = new_population

        # Take the top n parents for the next population.
        new_parents = np.argsort(-fitness.ravel())[-parents_count:]
        if min(fitness.ravel()) < best_cost:
            best_cost = min(fitness.ravel())

        if iteration % 25 == 0:
            print(f'Iteration {iteration} cost {best_cost}')

    return population, best_cost

# Some initial test
# Pop, Par, Mut, Iter, Cost,   Time
# 40, 10, 0.001, 1000, 1186619, 807.91
# 80, 20, 0.001, 1000, 1014474, 1563.84
# 20, 5, 0.001, 1000, 1268749, 395.94
# 25, 5, 0.001, 1000, 1221041, 484.79
# 80, 10, 0.001, 1000, 793211, 1557.66
```

0.7 Data import

[22]:
```
# Data importing
family_data = pd.read_csv("../Data/family_data.csv", index_col= "family_id")
initial_submission = pd.read_csv("../Data/sample_submission.csv")
initial_allocation = pd.concat([initial_submission, family_data.iloc[:, -1],␣
 ↪family_data.iloc[:,0]], axis = 1)
initial_allocation = initial_allocation[['family_id', 'n_people', 'choice_0']]
initial_allocation = initial_allocation.rename(columns={"choice_0":␣
 ↪"assigned_day"})


# Kaggle provided a cost function, which is very fast. Need these values to use␣
 ↪it.
family_size_dict = family_data[['n_people']].to_dict()['n_people']
cols = [f'choice_{i}' for i in range(10)]
family_choice_dict = family_data[cols].to_dict()
```

0.8 Running Heuristics

[ ]:
```
# Run Random Search Family
best_assignment, best_cost = random_search_2(assign_df=initial_allocation,
                                              epsilon=0.5,
                                              num_iterations = 1000,
                                              swap_rate = 0.01,
                                              fam_size_dict = family_size_dict,
                                              choice_dict = family_choice_dict)
print(best_cost)
```

[ ]:
```
# Run Guided Local Search
best_assignment, best_cost = guided_local_search(assign_df=initial_allocation,
                              num_iterations = 1000,
                              epsilon = 0.5,
                              fam_size_dict = family_size_dict,
                              choice_dict = family_choice_dict,
                              greedy = True,
                              num_fam_swap = 100)
```

```
print(best_cost)
```

[ ]:
```
# Run Guided Local Search without Greedy Assignment
best_assignment, best_cost = guided_local_search(assign_df=initial_submission,
                             num_iterations = 1000,
                             epsilon = 0.5,
                             fam_size_dict = family_size_dict,
                             choice_dict = family_choice_dict,
                             greedy = False,
                             num_fam_swap = 100)
print(best_cost)
```

[ ]:
```
# Run Genetic Algorithm with Greedy Initialization
final_population, best_cost = genetic_algorithm(population_size = 30,
                                                assign_df = initial_allocation,
                                                epsilon = 0.5,
                                                fam_size_dict=family_size_dict,
                                                choice_dict=family_choice_dict,
                                                parents_count = 10,
                                                mutation_rate = 0.01,
                                                num_iter = 1000,
                                                family_matrix = family_data)
```