KE5207 COMPUTATIONAL INTELLIGENCE II

GENETIC ALGORITHM FOR ROUTING OPTIMIZATION

Applying the genetic algorithm technique to optimize Christmas gifts delivery across the globe for Santa Claus

SUBMISSION DATE: 21 OCTOBER 2018

TEAM: FAMOUS FIVE
SIDDHARTH PANDEY
PRANSHU RANJAN SINGH
EDUARD ANTHONY CHAI
NYON YAN ZHENG
TAN KOK KENG

INSTITUTE OF SYSTEMS SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

Origin of Problem

We have adopted the problem from a Kaggle competition in December 2015 with a prize money of USD 20,000. The title of the problem is *Santa's Stolen Sleigh*. The problem can be found at https://www.kaggle.com/c/santas-stolen-sleigh.

We have chosen this problem as it closely resembles our real-world vehicle routing problem for logistics companies.

Kaggle's winning score

The winning score, weighted reindeer weariness for this problem on Kaggle is 12.384 billion. This serves as our benchmark as we work through the problem.

Problem Statement

Santa Claus needs to deliver 100,000 gifts around the world by flight. All the gifts have a unitless weight ranging from 1 to 50. The total weight for all the gifts are 1,409,840 units and all the trips start from the North Pole. Each trip could only carry up to a weight of 1,000 units of weight and the reindeers will be weary.

Goal

The goal is to minimize the weighted distance (weighted reindeer weariness) under the following constraints.

Constraints

- Santa needs to deliver all the 100,000 gifts
- Every trip needs to start from and end at North Pole (Lat = 90, Long = 0)
- Sleigh has base weight = 10
- Sleigh has a weight limit = 1000 (excluding the sleigh base weight)
- There is no limit to the number of trips
- All gifts must be travelling with the sleigh at all times until the sleigh delivers it to the destination. Gifts are not allowed to be dropped off anywhere before it is delivered.

Fitness function

The fitness function is defined as the weighted reindeer weariness. The formula to calculate weighted reindeer weariness is given by:

$$WRW = \sum_{j=1}^{m} \sum_{i=1}^{n} \left[\left(\sum_{k=1}^{n} w_{kj} - \sum_{k=1}^{i} w_{kj} \right) \cdot Dist(Loc_i, Loc_{i-1}) \right]_j,$$

Where:

m = the number of trips

n = the number of gifts for each trip j

w_{ij} = the weight of the i^{th} gift at trip j

$Dist()$ = calculated with Haversine Distance between two locations

Loc_i = the location of gift i

Data Collection

The dataset is available for download from:
https://www.kaggle.com/c/santas-stolen-sleigh/data

Data Understanding

The dataset consists of 100,000 gifts that need to be delivered by Santa. The gifts information provided in the dataset are gift id, latitude, longitude, and the weight. A snapshot of the data is given in the table below

Table 1 Snapshot of Data

GiftId	Latitude	Longitude	Weight
1	16.34577	6.303545	1
2	12.49475	28.6264	15.52448
3	27.79462	60.03249	8.058499
4	44.42699	110.1142	1
5	-69.8541	87.94688	25.08889

Distribution of gifts based on their weight

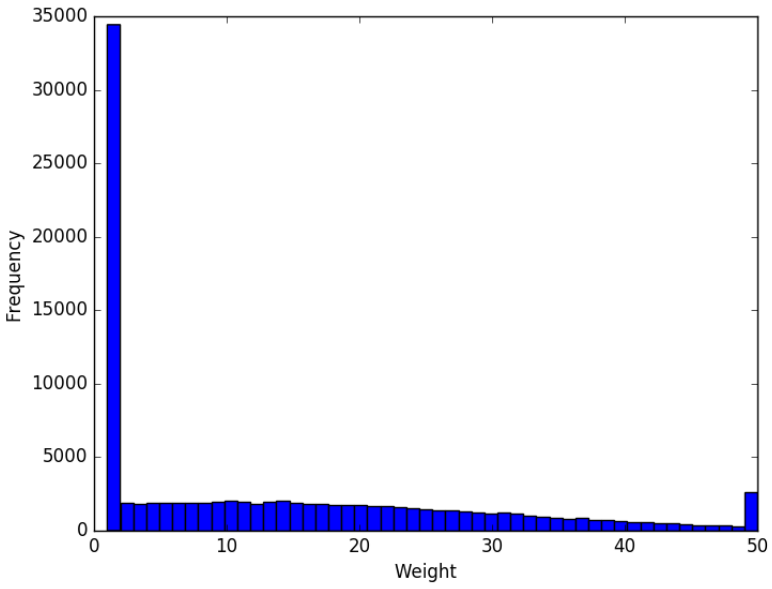

Figure 1 Distribution of Gifts by Weight

From the weight distribution, we can see that most of the gifts are weighted 1 unit. The rest of the gifts are considered well distributed. The number of gifts with weight 50 units is also slightly more.

Distribution of the gifts based on location

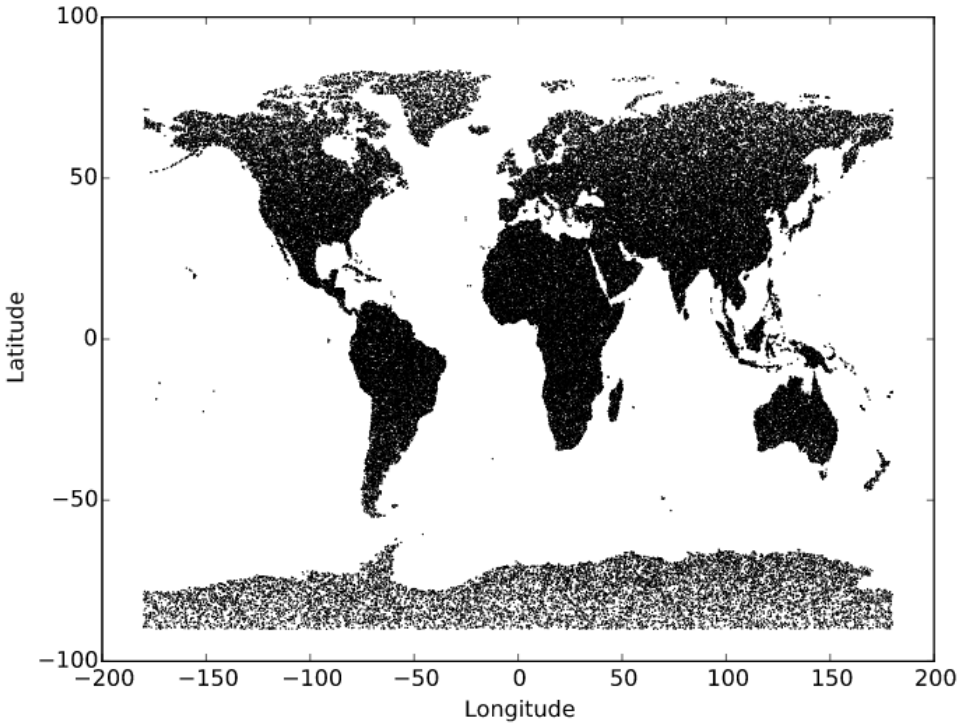

Figure 2 Plot of gifts location by latitude and longitude

The plot above shows that the gifts are well distributed around any land area across the globe. We might be able to do some clustering like Antarctica cluster and non-Antarctica cluster.

Genetic algorithm (GA)

We have tried 2 approaches to solve this problem:

1. Allow GA to optimize the routes from the whole search space of 100,000 gifts. Each individual or chromosome is a sequence of all trips and each gene consist of the list of gifts with a total weight limit of 1000 unit or less.
2. Cluster the gifts along the longitudes to reduce the search space. Each cluster is a trip and we run GA independently on each trip to optimize the sequence of gift delivery that minimizes the fitness function. In this case, each individual or chromosome is a trip and each gene in the chromosome are the gifts.

Approach 1

Hardware specification

For our problem, computation and memory requirement are very high. We have tried to solve the problem using our local machine and we are unable to run even single iteration, if not days for an iteration. To solve the computation and memory issue, we spawned a cloud instance with **42 cores**. With this instance, we were able to run 1 iteration in 1.5 to 2 hours.

Strategy overview

The proposed GA flow chart is given in Figure 3.

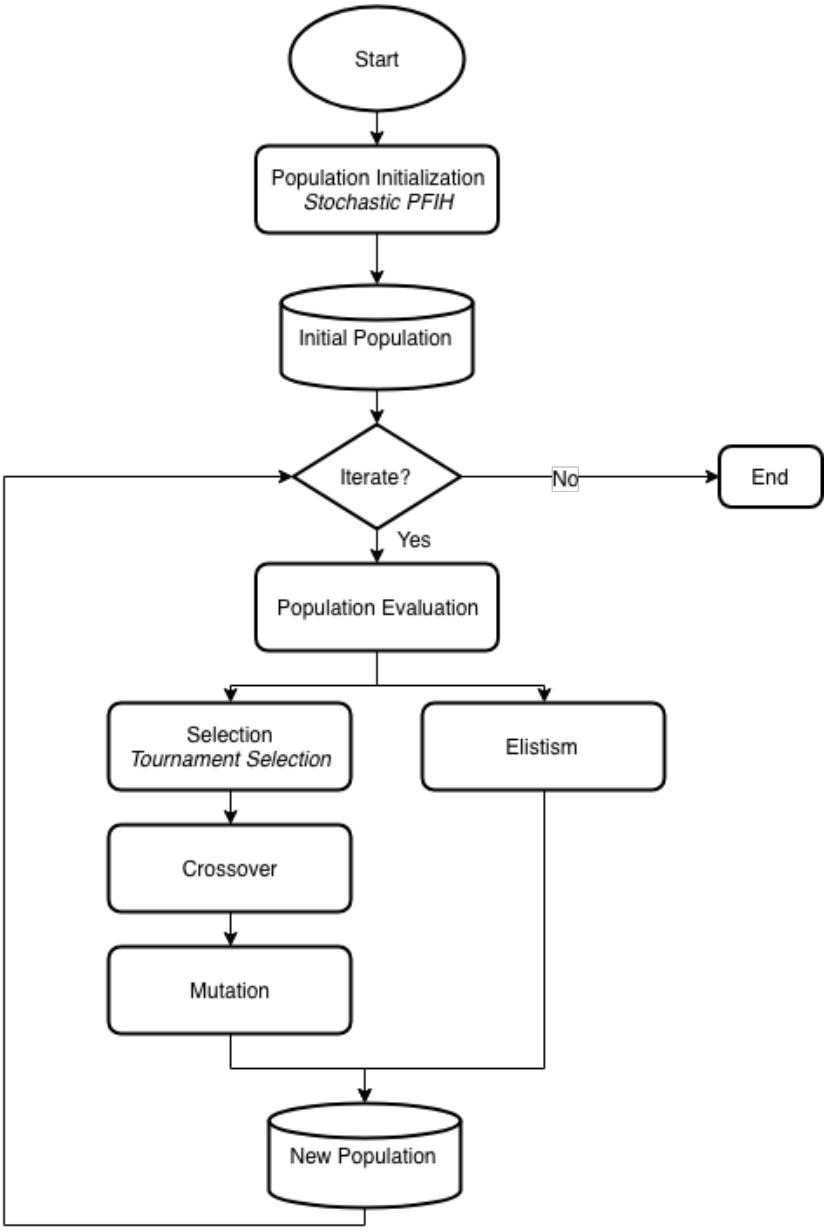

Figure 3 GA Flow Chart

Steps overview:

1. We have generated **200** individuals for the initial population.
2. We preserved **30% individuals with the best fitness score** and carry them over to the next generation.
3. To allow the elites to improve their score, we cloned them and then subject them to mutation process. So, for our new generation we have **30% elites + 30% mutated elites = 60% individuals.**
4. The other **40% individuals** are generated from the crossover, which means we need to select 80% from the initial population as 2 parents will produce 1 offspring.
5. The offspring from the crossovers are subjected to mutation too.
6. So, the new population consist of **30% elites + 30% mutated elites + 40% crossover and mutation**.

Initial population
200 individuals
⇓
Elites
30% * 200 = 60 individuals
⇓
Crossover selection
80% * 200 = 160
Crossover
160 / 2 = 80 individuals (2 parents produce 1 offspring)
⇓
Mutation
60 from elites + 80 from crossover = 140 individuals
⇓
New population
60 from elites + 140 from mutation = 200 individuals

1. Population initialization

Strategy 1: Basic random without heuristics

In our first attempt to solve the problem, we tried a very basic way to initialize our population without any heuristics. We initialized the population in the following steps:

1. Shuffle 100,000 gifts
2. Pick gifts in sequence until it reaches the weight constraint of 1000
3. Group the picked gifts into a trip
4. Repeat steps 2 and 3 until all gifts are picked
5. Repeat steps 1 to 4 as many times as the number of initial populations needed

This approach is the cheapest amongst other approaches that we tried to initialize the population. However, it gave us the worst quality of the individuals. Each individual has an average of **1430 trips** and **60 to 70 gifts** per trip. It gave us **approximately 400 billion** individual weariness on average.

Strategy 2: Stochastic Push Forward Insertion Heuristic (SPFIH)

In this approach, we were adopting the strategy proposed by Solomon [1], called *Push Forward Insertion Heuristic* (PFIH). In our project, we applied a modified version of PFIH called *Stochastic PFIH* (SPFIH) [2]. In SPFIH, we:

1. Randomly select 1 gift from available gifts (start with 100,000 gifts)
2. Calculate distance to all available gifts and find the next closest gift to deliver
3. Repeat step 2 until we reach the weight constraint of 1000
4. Group selected gifts into a trip
5. Repeat steps 1 to 4 until all gifts are delivered
6. Repeat steps 1 to 5 as many times as the number of initial populations is needed

This approach is expensive in terms of computation power but produces high quality individuals. Because of its computation power consumption, we were unable to execute this with a simple loop.

To optimize the computation, we created a lookup table to store distance between gifts. Even though in theory it will help us save computational power, we faced memory issue when generating 100,000 x 100,000 distance matrix. When we computed 100,000 x 20,000, it has already used approximately 30GB memory. It means that we need approximately 30GB x 5= 150GB of memory to generate a 100,000 x 100,000 distance matrix. It is simply too expensive for us to achieve.

To be able to achieve this with limited computation power and memory, we assumed that the next closest available gift to current gift should be fall within 10,000 closest gifts. If it goes beyond 10,000 closest gifts, it is better for Santa to return to the North Pole and load more gifts. With this assumption, we created a lookup table that store 10,000 closest gifts to each gift which gave us matrix with dimension 100,000 x 10,000.

To further optimized computation and memory, we did some testing and found that the average index of next closest available gift is approximately within 3,000 gifts. So, we reduced our lookup table dimensions even further to 100,000 x 5,000. This is our final lookup table and it is stored in ***Redis*** for fast lookup and persistency. It took us 3 to 4 hours to generate the lookup table.

Without lookup table, we were simply limited by computation power and unable to generate even a single individual. With lookup table, we were able to generate a single individual in 600 to 700 seconds. This approach gave us **approximately 14 billion** individual weariness on average with **1410 trips** per individual and **60 to 70 gifts** per trip.

2. Population evaluation

For our GA, we evaluate the population and process them with 2 different strategies.

1. The first strategy is elitism.
2. The second strategy is selection, crossover, and mutation.

The combination of these 2 strategies will give us a new population.

3. Elitism

We picked **60** individuals with the best score from the population (**30%** of the population). We adopted elitism strategy to guarantee that our GA never retreat quality solution.

4. Selection, crossover, and mutation

Selection

We picked ***Tournament Selection*** as our selection strategy. In this strategy, we randomly picked **30** individuals from the population and picked **2 winners**. The 2 winners will be used for crossover in the next step and will produce 1 offspring. This process is repeated until we have **160** individuals (**80%** of the population).

Crossover

After the selection process, we crossover the 2 winners to get an offspring. The algorithm that we used for the crossover is detailed below:

1. Copy random trip from parent A
2. Copy random trip from parent B
3. Validate the 2 trips by checking if trip copied from parent A and parent B does not have overlapping gifts
4. If no overlaps occur, inherit trips to offspring
5. Otherwise, repeat 1-3
6. Repeat 1-5 until there are no more trips can be inherited
7. Insert undelivered gifts into inherited trips in offspring if possible
8. Generate trips for remainder undelivered gifts using SPFIH approach

Mutation

The population for mutation consists of individuals from the crossover and the elites. The algorithm used for mutation:

1. Picked two random trips from an individual
2. Switch gifts between 2 trips
3. If the switch gave a better result, proceed with mutation
4. Otherwise, do more iterations

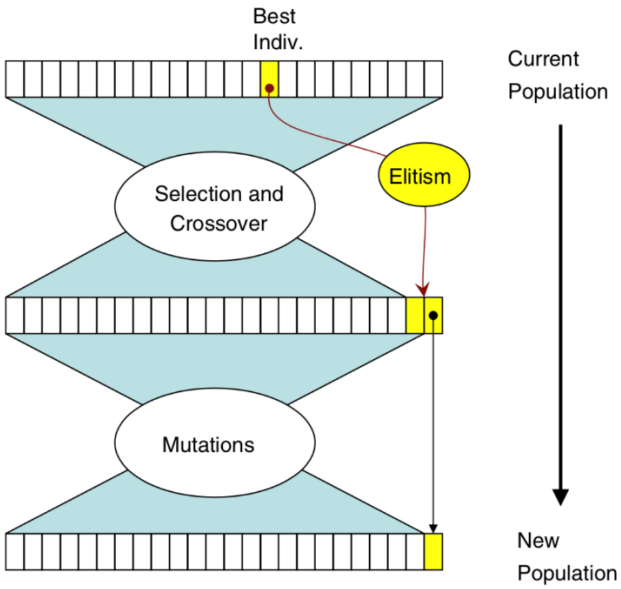

Figure 4 Selection, crossover and mutation strategy

Result

Running the algorithm took 1.5 to 2 hours per iteration with 42 cores machine. We run the algorithm for 1 day and the results can be seen below:

Initialization Strategies	Initial Population	Last Best Solution
Random initialization	~400 billion	~350 billion
SPFIH initialization	~14 billion	13.89 billion

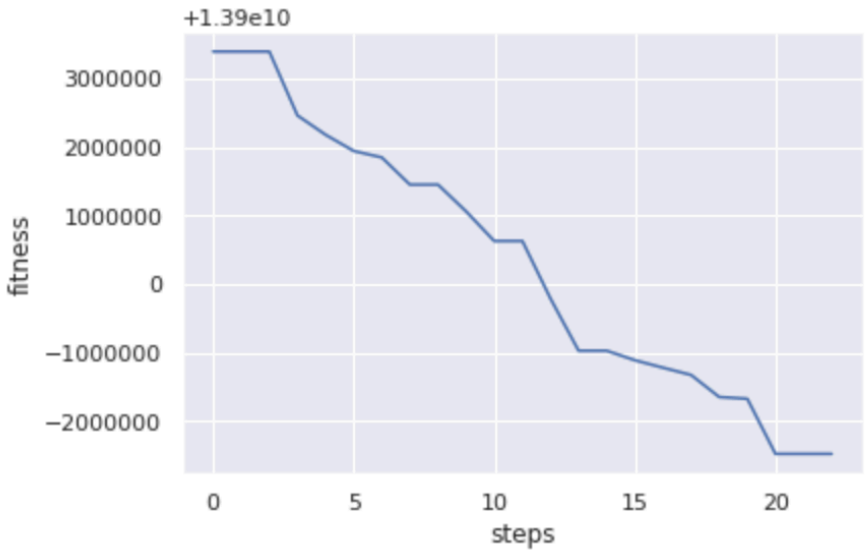

Figure 5 Fitness value over 20 iterations with SPFIH strategy as initial population

Based on the result, we can conclude that using SPFIH as the initialization strategy is the best solution overall. It Is expensive to initialize, but the quality of the solution given is justifiable.

The random strategy is cheaper to execute but it gave the worst performance overall. Given the bad initial population quality, the improvement rate given by GA is only logical.
Link to source code and data for this approach can be found here.

Approach 2

Heuristics and Setup

From the list of 100,000 gifts, we have first split the gifts into 2 main groups:

1. Gifts location in the non-Antarctica regions (latitude more than -60): 11,343 gifts
2. Gifts located in the Antarctica region (latitude less than -60): 88,657 gifts

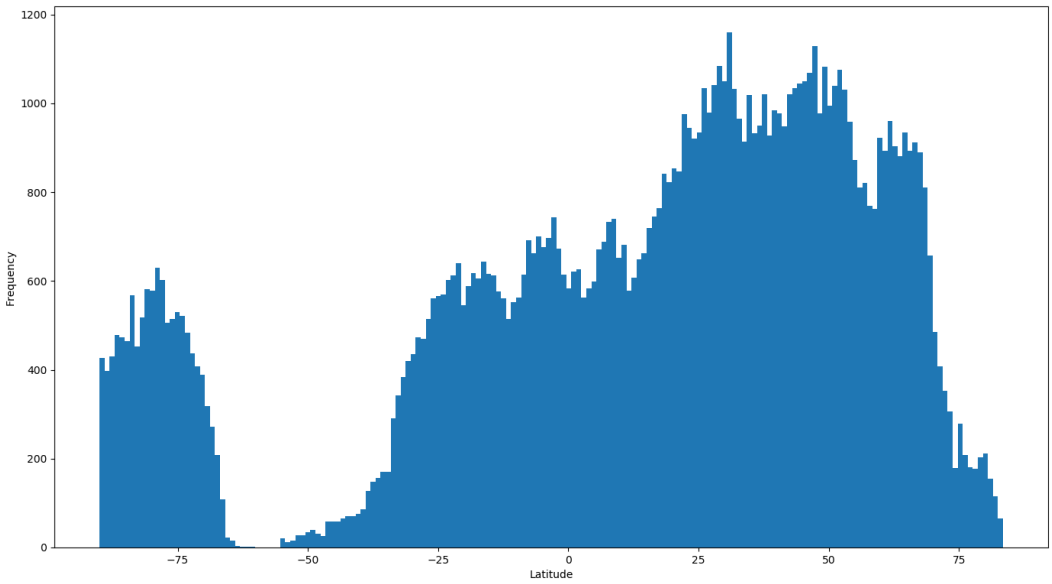

Figure 6 Distibution of gift by Latitude

The rationale behind this split is that there are no gifts located on the latitude -55.2 to -60.5 hence it may not be efficient to travel south across this region just to deliver gifts at the Antarctica region. Furthermore, the distances within the Antarctica region at the south pole is small and hence we can have separate trips to this region on its own.

From the non-Antarctica and Antarctica split above, we split the gifts along the longitudes. Each trip consists of gifts located along the similar longitude, as straight as possible and with weights less than 1000 units. This is based on the understanding that it might be more efficient to deliver the gifts along the paths that are as straight as possible.

Hence, all trips, which originates from the north pole will travel in a "straight" route down to the south and goes back up to the north pole. The GA will work on each trip independently to optimize the sequence of delivery of the gifts within each trip.

This arrangement resulted in a total of 1432 trips, with an average of 70 gifts in each trip:

1. 1268 non-Antarctica trips
2. 164 Antarctica trips

Each trip is then treated as a travelling salesman problem and we use GA to optimise each of the 1432 trips. The GA will optimise the delivery sequence of the gifts in each trip, depending on the weights, that minimises the fitness function which is the weighted reindeer weariness.

Library

We have used the DEAP (Distributed Evolutionary Algorithms) library in Python for this approach.

Parameters for GA

For this approach, each chromosome is a trip and each gene is a gift in the GA. To see how parameters of the GA affect fitness and computation time, we have randomly chosen a trip to inspect. The gifts in the selected trip is given in Appendix 1.

There are 68 gifts in this trip with a total weight of 999.851. With a random initialization, the fitness value at the start, before GA is **16,410,173**.

We have computed the time (in seconds) for the GA to run on our sample trip for population size 100 to 500 and for generations from 100 to 1000 in Table 2 below.

Table 2 Time in seconds for GA to run for different population size and generations in sample trip

		Population Size				
		100	**200**	**300**	**400**	**500**
Number of Generations	**100**	5	11	16	22	27
	200	10	22	32	43	61
	300	17	31	47	63	81
	400	22	48	67	93	109
	500	28	53	82	108	141
	600	32	69	98	133	167
	700	38	76	113	150	189
	800	43	86	129	173	215
	900	50	98	145	196	246
	1000	54	108	161	218	278

In Table 2, we can see that computation time definitely increases with increasing population size and number of generations. We have decided to use population size 300 and 500 generations in our final GA, with computation time between 82 to 90 seconds for each GA.

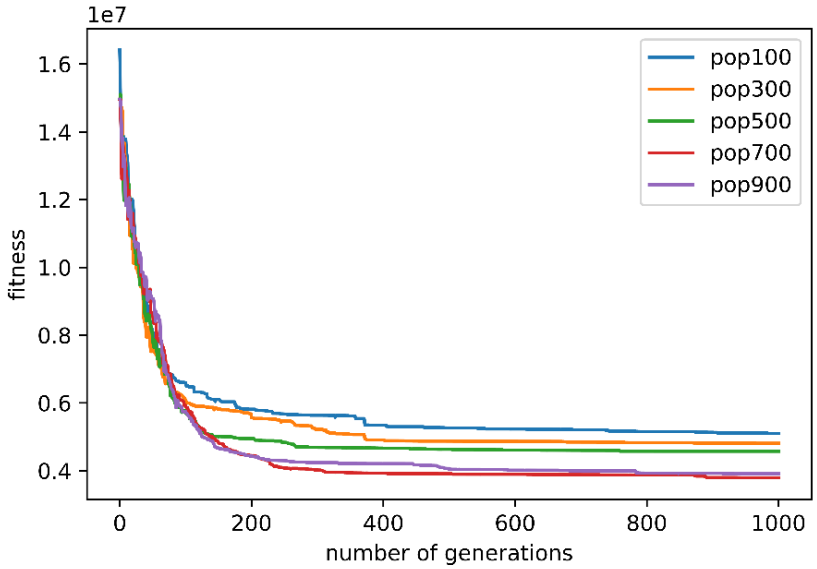

Figure 7 Fitness value across 1000 generations for different population size

In Figure 7, we have plotted the fitness level across 1000 generations for population size 100, 300, 500, 700 and 900. We can see that generally larger population generally will produce lower fitness value. We also see that fitness becomes stagnant after 400 generations.

The decrease in fitness value after 1000 generations are given in the table below:

Table 3 Fitness Score after 1000 generations

Population Size	Fitness After 1000 generations	% Decrease from Initial Fitness of 16,410,173
100	5,095,718	69%
200	4,101,818	75%
300	4,805,183	71%
400	4,780,080	71%
500	4,569,081	72%
600	4,515,294	72%
700	3,788,759	77%
800	4,164,748	75%
900	3,906,790	76%
1000	4,426,833	73%

We can see that the fitness generally decreases with larger population size. After considering the computing time involved, we have chosen to set population size to 300 and generation number to 500 in our final GA run.

Final GA setup

Weighing between computation time and fitness, we have chosen the parameters below to run the GA for all the trips.

Each trip is optimized independently using the above GA setup, which means there is 1432 GAs being run for the whole problem. Each GA took approximately 90 seconds to evolve through 500 generations, summing up to 35 hours to complete the problem.

Population Size	300
Crossover	We have used the **partially matched crossover** method from the library. This crossover involves two individuals being modified in place. It generates two children by matching pairs of values in a certain range of the two parents and swapping the values of those indexes.
Mutation	We have used the **mutShuffleIndexes** mutation from the DEAP library. This mutation method shuffles the attributes of the input individual and returns the mutant. The independent probability for each attribute to be exchanged to another position is set to **0.05**.
Tournament	**The tournament size is set to 3**, where this is the number of individuals participating in each tournament. The tournament will select the best individual among 3 randomly chosen individuals.
Algorithm	We have used DEAP simplest evolutionary algorithm. **Probability of crossover = 0.7** **Probability of mutation = 0.2** **Number of generations = 500** It is worth noting here that since each trip is set up like a travelling salesman problem, the crossover and mutation essentially is behaving in the same manner where both crossover and mutation occur within the individual itself.

Results

The GA resulted in the fitness, which is the weighted reindeer weariness of 23.54 billion. This is worse than the best solution in Approach 1 mentioned in the previous section.

Summary

Given the time constraint and cost of spawning 42 cores instance, we only run our GA for 1 day for Approach 1 and for Approach 2 for approximately 35 hours. For Approach 1, the fitness score is still decreasing when we stop the scripts.

Nonetheless, the results are summarized in Table 4 below.

Table 4 Summary of the 3 approaches

Approach	Strategies	Last Best Solution
1	Strategy 1: Random initialization + GA	~350 billion
1	Strategy 2: SPFIH initialization + GA	13.89 billion
2	Longitudinal Slicing	23.54 billion
-	**Kaggle benchmark**	**12.384 billion**

Future improvement

We can explore other formulations of the problem for GA. An alternative formulation of the problem makes use of the general set partitional problem (SPP). In this problem formulation, an individual comprises 1 chromosome in which each gene is a possible trip which can be made to and from Santa's depot at the North Pole. Immediately, it can be seen that this problem formulation is difficult given large number of gifts (100, 000) we have, which results in a combinatorial explosion of possible trips.

A literature search on vehicle routing problems (VRP) show a possible approach using the SPP formulation [1]. In this approach, a multi-phase solution is adopted. In phase 1, the solution which we implemented in our approach 1 is used to generate good quality individuals. The GA process is run multiple times to generate a set of such individuals. The trips from these individuals are collected together to form a set of possible trips for the GA based on the SPP formulation. This provides a reasonably sized set of possible trips for the SPP-based GA process to further search for an optimized solution.

The basis for this two-phase approach lies in the fact that a local minimum for a VRP problem has a significant possibility of containing routes that are also found in the global optimum. Therefore, if several local minimum solutions are found, it is possible to join these solutions into a search space and apply the SPP-based GA.

Also, we could also explore using heuristics to arrange gifts into trips as describe in Approach 2 and further use GA to optimize the arrangement of gifts to allow for crossover and mutation between trips.

References

[1] Solomon MM. Algorithms for the vehicle routing and scheduling problems with time window constraints. Operations Research 1987;35(2):254–65.
[2] Guilherme Bastos Alvarenga, Geraldo Mateus, Giorgio De Tomi. A genetic and set partitioning two-phase approach for the vehicle routing problem with time windows.

Appendix

List of gifts from 1 trip selected to check how the parameters changes the fitness value and computation time in Approach 2.

GiftId	Weight	Latitude	Longitude
55620	8.624	66.982	-132.824
74475	11.392	65.719	-133.320
92887	1.000	61.797	-133.082
44453	26.171	65.273	-132.742
92977	12.902	62.996	-132.883
46204	3.689	58.315	-132.735
17412	9.537	68.850	-132.731
49197	26.874	58.906	-132.806
61363	9.721	55.266	-133.357
75969	6.808	59.244	-133.196
84808	1.000	55.965	-133.078
5398	1.000	61.569	-133.115
70393	37.429	60.767	-133.377
68473	41.368	63.637	-132.528
1768	24.812	60.175	-133.034
56812	21.457	63.641	-133.246
84712	1.000	62.532	-133.002
48416	1.000	66.373	-132.789
5498	1.000	59.170	-133.356
45734	34.134	68.521	-132.805
89958	15.615	67.829	-132.710
61425	43.207	67.949	-133.306
8588	30.799	66.848	-132.765
40048	13.648	55.341	-132.681
73696	20.405	69.263	-132.846
71351	1.000	65.375	-132.836
88756	3.692	69.244	-133.225
79885	1.000	67.400	-133.211
15617	21.119	62.994	-133.132
98001	23.591	58.868	-133.081
49105	3.989	63.607	-133.240
74207	6.070	66.328	-132.592
74610	50.000	63.323	-133.415
46644	1.000	59.952	-132.697

GiftId	Weight	Latitude	Longitude
45211	1.000	68.444	-133.358
62594	1.000	62.275	-132.531
40777	1.000	67.116	-132.770
89180	2.127	56.140	-133.067
91244	2.130	55.288	-133.101
64420	1.000	60.976	-132.816
6926	37.153	62.600	-133.257
34326	48.712	68.306	-132.954
48271	40.541	66.375	-133.097
72697	32.651	55.475	-133.069
59652	38.860	55.658	-132.903
57823	14.819	63.120	-132.894
86260	19.697	68.144	-133.189
62631	17.373	69.168	-133.247
12125	1.000	66.266	-133.046
98826	3.805	68.232	-133.273
36143	1.000	63.712	-133.009
43477	9.938	60.383	-132.746
27897	42.722	63.921	-133.187
1435	1.000	58.225	-132.862
5796	1.000	57.762	-133.241
15895	1.264	69.019	-132.734
17317	3.670	62.322	-133.373
98744	50.000	59.517	-132.984
94702	13.267	65.332	-133.078
62119	1.000	59.849	-132.638
91629	1.000	67.080	-133.290
99248	10.943	62.634	-132.572
42116	41.044	63.637	-132.491
14072	1.000	61.702	-133.237
39160	4.025	63.563	-133.282
68990	1.000	57.577	-133.112
25789	19.806	57.997	-132.664
95804	17.249	68.072	-132.984