Mphasis FOUNDATION
INNOVATE • INCLUDE • INVEST

INTER IIT TECH MEET 12.0

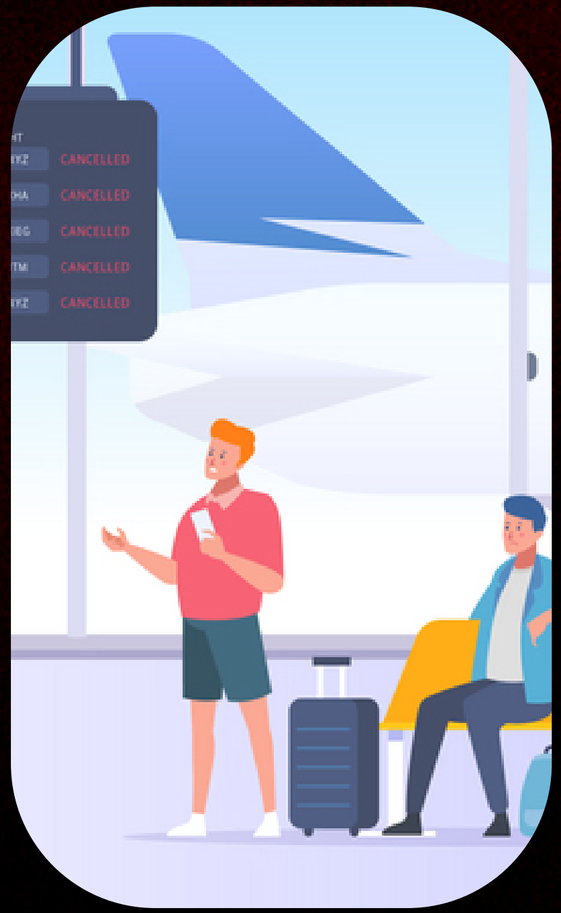
QUANTUM COMPUTING

Team 59

Passenger Recovery during Airline Disruptions

Agenda

Team 59



PROBLEM DESCRIPTION

Airlines routinely change their flight schedules for various reasons like seasonal demands, to pick new routes, time changes needed based on daylight savings, changes to flight numbers, operating frequency, timings, etc. Many passengers get impacted due these schedule changes and they need to re-accommodate to the alternate flights. Airlines need a solution to analyse the impact to the passengers with their planned schedule changes and automatically identify the suitable alternate flights for the impacted passengers.

TASK 1	Find Alternate Flight Routes for Impacted PNRs
TASK 2	Reallocate the PNRs to the flight plans found

Team 59

Our Goals

Expected outcomes of the models

GOAL # 1

Flexible Business Rules

GOAL # 2

Handling Multiple Disruptions while optimizing the solution

GOAL # 3

Time is Everything. It has to be Fast . . .
No Compromise over Correctness though. . .

Team 59

Assumptions

Logical Assumptions we took/can take

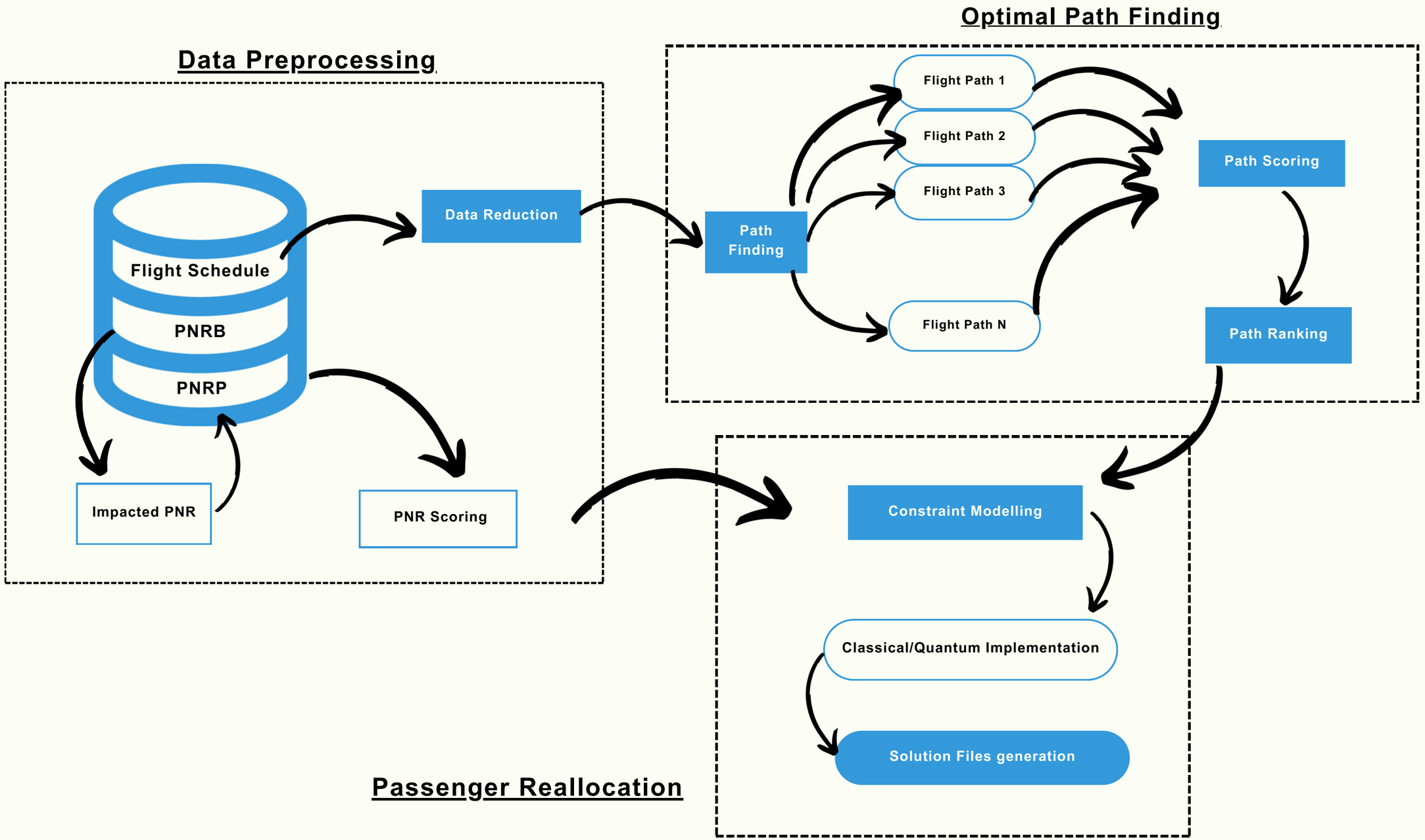
There is a maximum of ‘k’ stopovers allowed in any flight plan
Solution Space Reduction from O(N !) –> O(k! * N) –> O(N)

It is not allowed to visit an already visited airport in a flight plan
Loop Removal allows for further reduction in complexity

There is a set A of airports were it is allowed for a stopover
Targeted Searches <=> Faster Speed (Not Implemented)

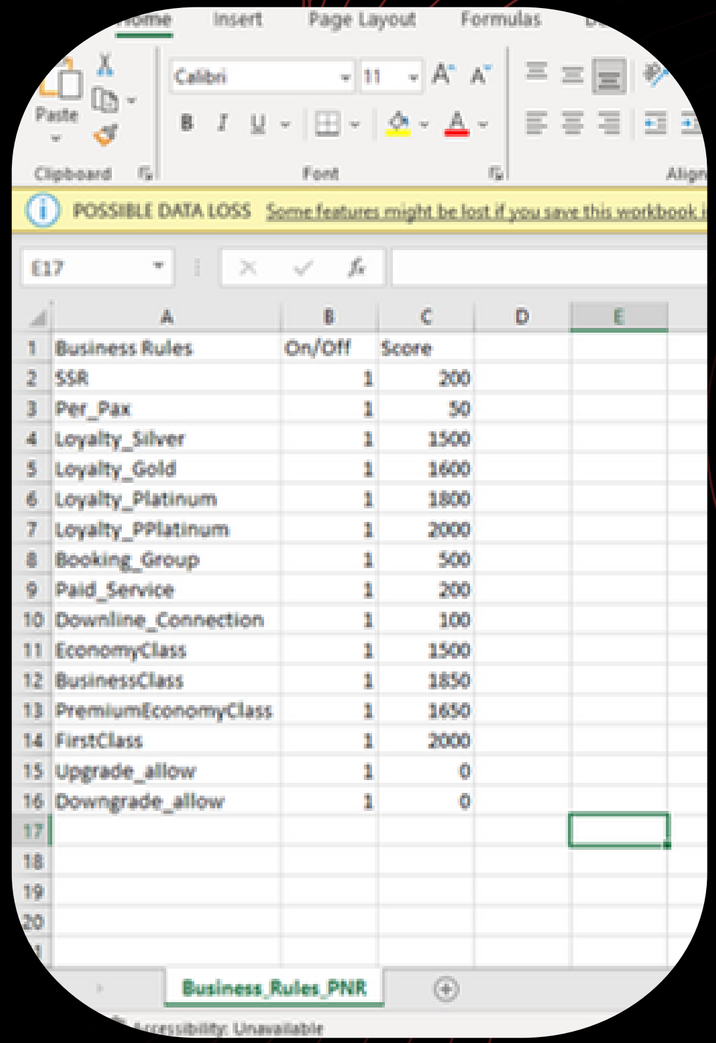
Priority is to get maximum passengers accommodated, then to get most PNRs in default solution, while maximizing PNR scoring

PIPELINE FOR ITINERARY SEARCHING

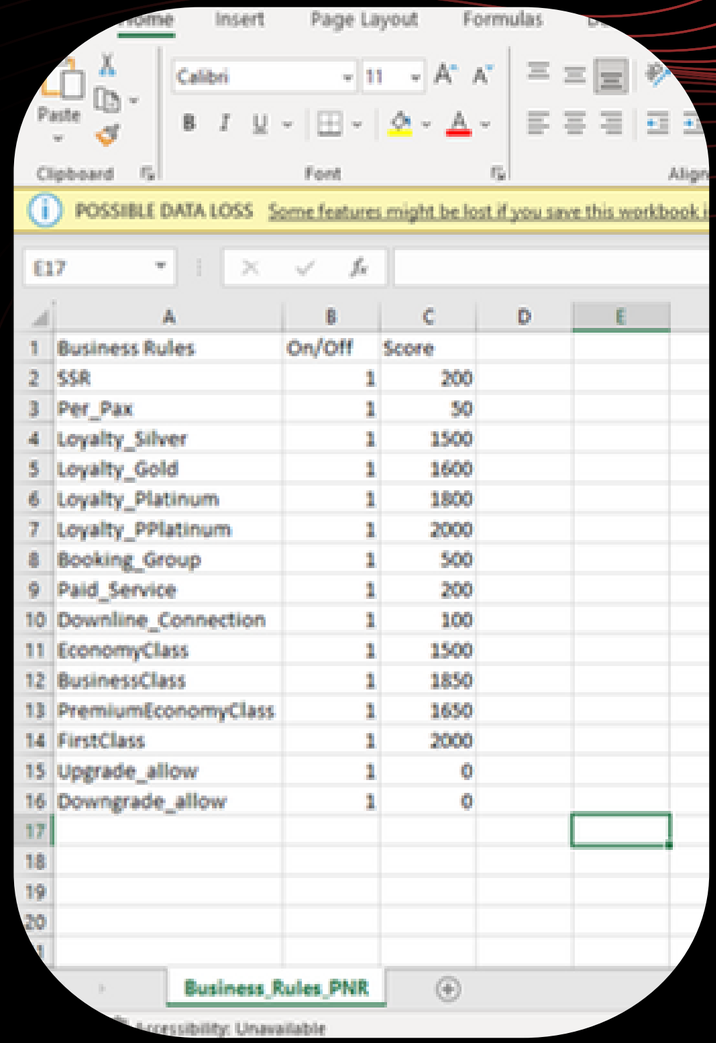


RULE ENGINE

Team 59

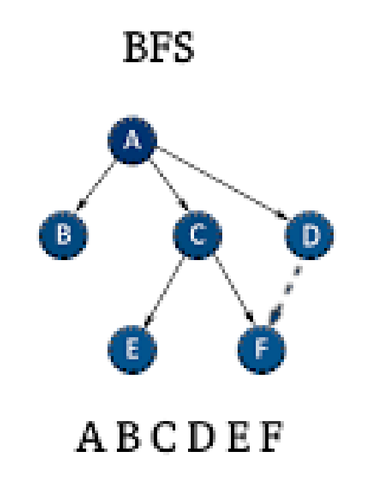


	A	B	C	D	E
1	Business Rules	On/Off	Score		
2	SSR	1	200		
3	Per_Pax	1	50		
4	Loyalty_Silver	1	1500		
5	Loyalty_Gold	1	1600		
6	Loyalty_Platinum	1	1800		
7	Loyalty_PPlatinum	1	2000		
8	Booking_Group	1	500		
9	Paid_Service	1	200		
10	Downline_Connection	1	100		
11	EconomyClass	1	1500		
12	BusinessClass	1	1850		
13	PremiumEconomyClass	1	1650		
14	FirstClass	1	2000		
15	Upgrade_allow	1	0		
16	Downgrade_allow	1	0		
17					
18					
19					
20					



	A	B	C	D	E
1	Business Rules	On/Off	Score		
2	SSR	1	200		
3	Per_Pax	1	50		
4	Loyalty_Silver	1	1500		
5	Loyalty_Gold	1	1600		
6	Loyalty_Platinum	1	1800		
7	Loyalty_PPlatinum	1	2000		
8	Booking_Group	1	500		
9	Paid_Service	1	200		
10	Downline_Connection	1	100		
11	EconomyClass	1	1500		
12	BusinessClass	1	1850		
13	PremiumEconomyClass	1	1650		
14	FirstClass	1	2000		
15	Upgrade_allow	1	0		
16	Downgrade_allow	1	0		
17					
18					
19					
20					

STAGE 1: PATH FINDING



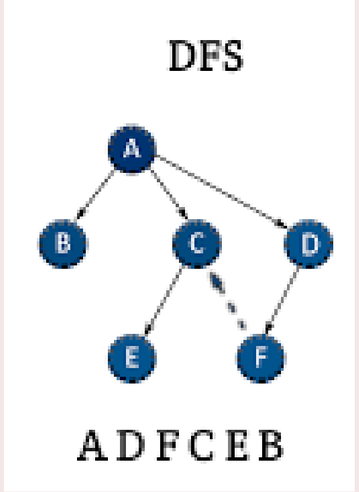
Breadth First Search

Disadvantage

- Creating a Graph of possible paths while considering constraints burns through the resources
- Searching takes O(N logN)

REJECTION LIST

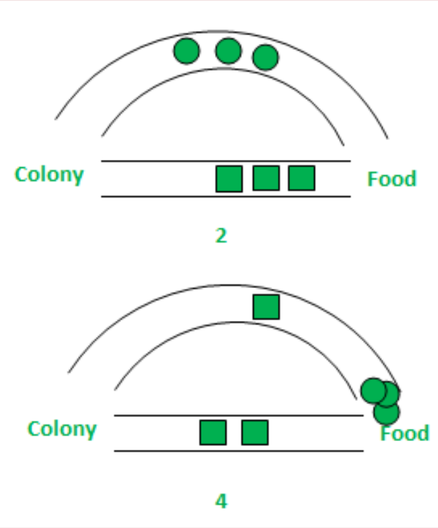
Team 59



Depth First Search

Disadvantage

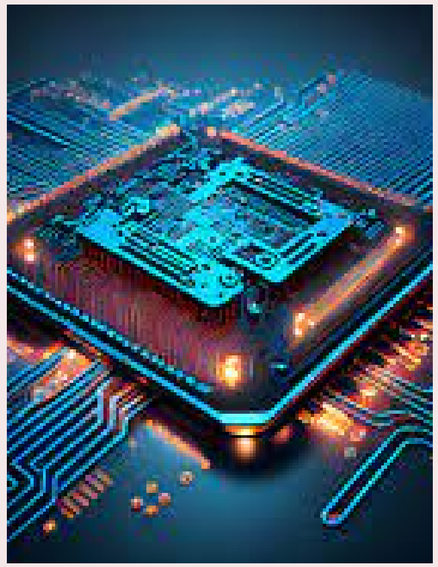
- Creating a Graph of possible paths while considering constraints burns through the resources
- Searching takes O(N logN)



Ant Colony Optmization

Disadvantage

- Slow Convergence to Optimal Solution
- Pheromone Deposition can miss global Optima
- High number of ant makes the runtime blow up quadratically



QACO and Quantum Walk

Disadvantage

- Theoretically it was the most promising with time complexity O(√N). After actual implementation, we found out a different issue !!!

References: Andris Ambainis. Quantum walks and their algorithmic applications. International Journal of Quantum Information, 1(04):507–518, 2003.

IMPLEMENTATION STAGE 1: PATH FINDING

The methods we tried but rejected

```
PS H:\My Drive\QC Inter IIT> & "C:/Users/Sukhvansh Jain/AppData/Local/Programs/Python/Python310/python.exe" "
h:/My Drive/QC Inter IIT/QACO_updated.py"
{'1110': 36, '0111': 43, '1111': 253, '0010': 33, '1000': 42, '0000': 233, '0011': 30, '1010': 37, '1101': 35
, '0110': 45, '0101': 31, '1100': 33, '1001': 39, '0100': 47, '0001': 40, '1011': 47}
PS H:\My Drive\QC Inter IIT>
```

Quantum Ant Colony Optimization

Rejection reason: Unstable with current quantum systems, Need more Qubits

Classical Ant Colony Optimization

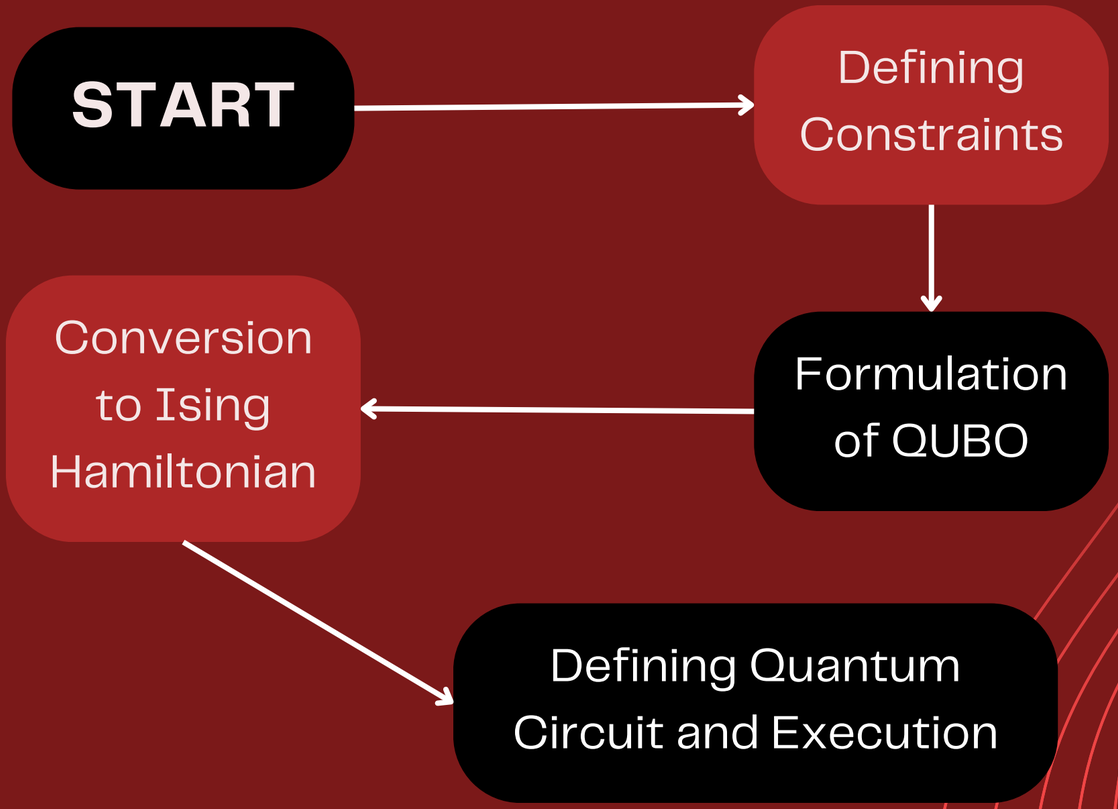
Rejection reason: Runtime is too high

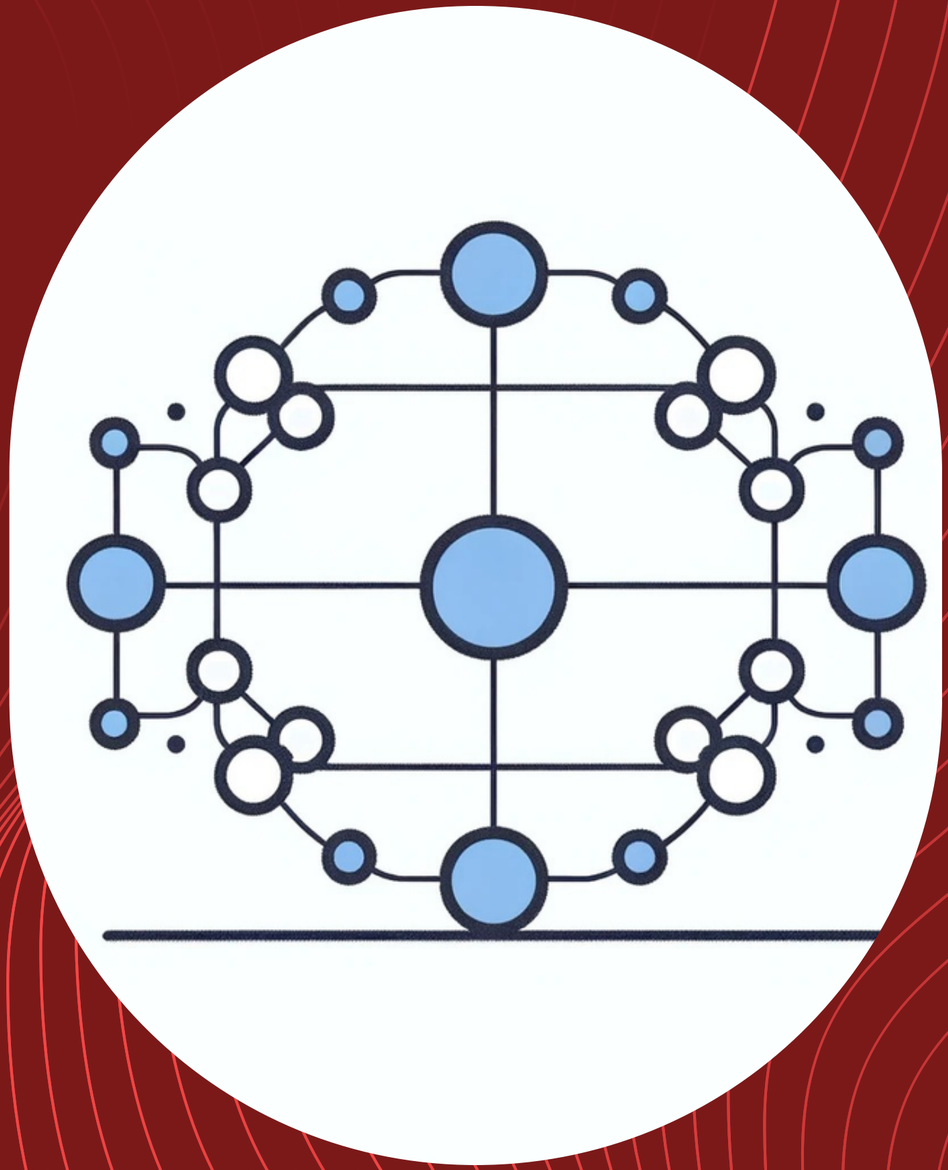
```
SIMULATION STATISTICS:

Airports Count:  30
Time Span:  101

Flight ID of Disruption:  13485
Source Node of Disruption:  14
Destination Node of Disruption:  16

Departure Time of Disruption:  4 // 1
Arrival Time of Disruption:  3 // 2

{'Path': None, 'Path_cost': inf, 'Epoch': 1, 'Clock': 1.017286200003582}
{'Path': None, 'Path_cost': inf, 'Epoch': 2, 'Clock': 2.7474938000086695}
{'Path': None, 'Path_cost': inf, 'Epoch': 3, 'Clock': 5.15698010000051}
{'Path': None, 'Path_cost': inf, 'Epoch': 4, 'Clock': 7.3786158000002615}
{'Path': [14, 16.0], 'Path_cost': 25, 'Epoch': 5, 'Clock': 8.947287200004212}
{'Path': [14, 16.0], 'Path_cost': 25, 'Epoch': 6, 'Clock': 10.719987500007846}
{'Path': [14, 16.0], 'Path_cost': 25, 'Epoch': 7, 'Clock': 12.7915151000052}
{'Path': [14, 16.0], 'Path_cost': 25, 'Epoch': 8, 'Clock': 14.521899100000155}
{'Path': [14, 16.0], 'Path_cost': 25, 'Epoch': 9, 'Clock': 16.948991199999}
{'Path': [14, 16.0], 'Path_cost': 25, 'Epoch': 10, 'Clock': 18.764366200004588}
{'Path': [14, 16.0], 'Path_cost': 25, 'Epoch': 11, 'Clock': 21.050808400003007}
{'Path': [14, 16.0], 'Path_cost': 25, 'Epoch': 12, 'Clock': 23.126063900010195}
{'Path': [14, 16.0], 'Path_cost': 25, 'Epoch': 13, 'Clock': 24.864134600007674}
{'Path': [14, 16.0], 'Path_cost': 25, 'Epoch': 14, 'Clock': 26.586843099998077}
{'Path': [14, 16.0], 'Path_cost': 25, 'Epoch': 15, 'Clock': 28.319043299998157}
```

For quantum Reference : Mrityunjay Ghosh, Nivedita Dey, Debdeep Mitra, Amlan Chakrabarti. A novel quantum algorithm for ant colony optimization

QAOA

Optimization Algorihtm





Reference: M Alpuente, D Ballis, S Escobar, and J Sapi˜ na. Journal of logical and algebraic methods in programming. 2021

Team 59

Quantum Annealing

Benefits of Quantum Annealing

Efficiency: For optimization tasks, quantum annealing can outperform Gate based approach

Rapid Convergence: For its target problem areas, quantum annealing can converge to a solution more quickly than other methods

Error Tolerance: Quantum annealers typically have a higher tolerance for operational errors

Optimization Efficiency: It's specifically designed for optimization problems

Noise tolerance: as quantum annealing approach focus on finding minina through tunneling, small noise wont make any impact

Decoherence Resilience: quantum annealers leverage quantum tunneling, which can be effective even in the presence of some level of decoherence.

OPTIMIZATION USING QUANUTM ANNEALING

BQM

- D waves' Annealing Version of QUBO
- Solved Using Leap's Hybrid Solver

OBJECTIVE FUNCTION

$$min \sum_{e}^{E} \sum_{j}^{S_2} \sum_{k}^{A_2} \sum_{l}^{D_T} |T'_{jekl} - \beta| q_{jekl}$$

$$min \sum_{i}^{S_1} \sum_{k}^{A_1} \sum_{l}^{D_T} |T_{Sikl} - \alpha| q_{Sikl}$$

CONSTRAINTS

$$\sum_{i}^{S_1} \sum_{k}^{A_1} \sum_{l}^{D_T} q_{Sikl} = 1$$

$$\sum_{e}^{E} \sum_{j}^{S_2} \sum_{k}^{A_2} \sum_{l}^{D_T} q_{jekl} = 1$$

PATH FINDING

$$\sum_{o} \sum_{k} \sum_{l} q_{ojkl} - \sum_{j} \sum_{k} \sum_{l} q_{jqrs} = 0 \quad \forall j \notin S, E$$

$$\sum_{k'} \sum_{m'} \sum_{l'} T_{jk'l'm'} q_{jk'l'm'} - \sum_{i} \sum_{k} \sum_{l} q_{ijkl} \left(T'_{ijkl} + \lambda_2 \sum_{k'} \sum_{m'} \sum_{l'} q_{jk'l'm'} \right) \leq 0 \quad \forall j \notin S, E \qquad (7)$$

MIN/MAX TIME DELAY

$$\sum_{k'} \sum_{m'} \sum_{l'} T_{jk'l'm'} q_{jk'l'm'} - \sum_{i} \sum_{k} \sum_{l} q_{ijkl} \left(T'_{ijkl} + \lambda_1 \sum_{k'} \sum_{m'} \sum_{l'} q_{jk'l'm'} \right) \geq 0 \quad \forall j \notin S, E \qquad (8)$$

CQM

Team 59

Same constraints as BQM (NOTE: not QUBO)

Does not form a penalty function to be added in the objective function

Provides Efficiency and Speed over BQM

NOTE: CQM is more than 250% faster than BQM !!!

Can provide solutions with no violation of constraints (NOTE: Not the case with BQM)

Team 59

Classical Approach

Path Finding Using Classical Recursion

The exploration of alternative flight routes is facilitated through a recursive search mechanism.

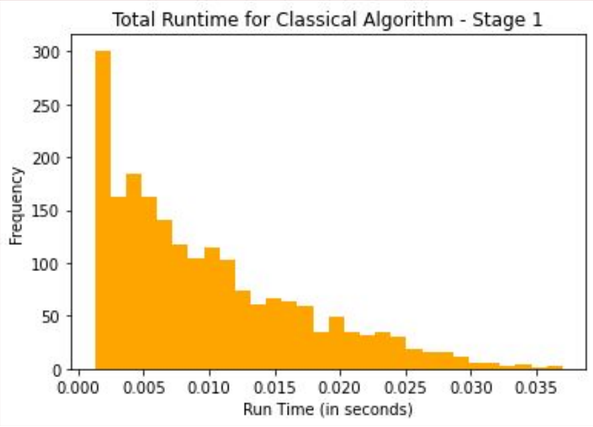
Despite the theoretical concern of poor time complexity associated with recursion, real-world constraints dictate that stopovers rarely exceed three flights

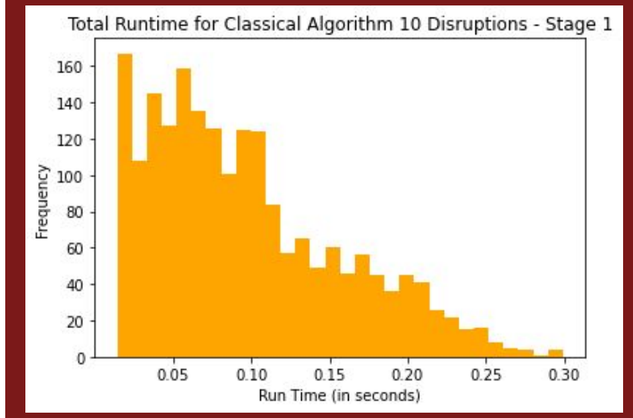
This limitation ensures a low recursion depth, contributing to the code's remarkable speed.

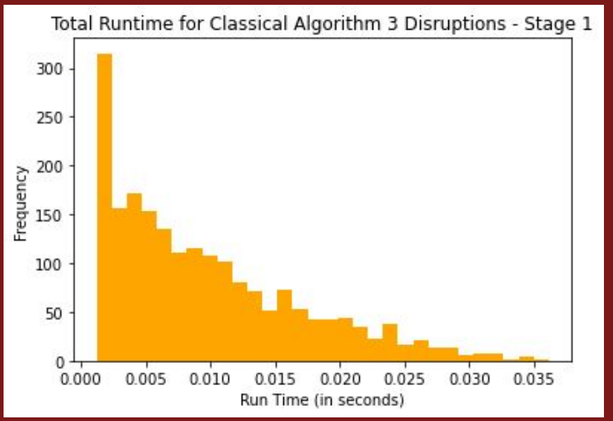
We used Pandas as our data handler, with Numpy under the hood. The Original creator of pandas, Wes McKinney, was overly obsessed with efficiency and speed.

Stage 1: Runtime Comparison

BEST CLASSICAL APPROACH







Average Runtime:
0.029935202146299557 seconds

Accuracy: 100%
(Since a deterministic classical approach is used with no approximations with all constraints satisfid)

Paths Found: All possible paths given a constraint

BEST QUANTUM APPROACH

For “INV-ZZ-6029799" as cancelled

```
Top 5 unique solutions:

Unique Solution 1:
Energy: 0.0 Occurrences: 1
Sample: {'1': 0.0, '5': 0.0, '6': 0.0, '7': 0.0, 'q(1,3,1,5)': 0.0, 'q(2,0,0,4)': 1.0, 'q(2,1,2,0)': 0.0, 'q(2,1,2,2)': 0.0, 'q(2,1,2,7)': 0.0, 'q(3,
```

```
print('cqm solver time: ',end-start)
```

✓ 0.0s Python

```
cqm solver time:  6.313271999359131
```

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16    energy num_oc.
0  0  1  0  0  0  0  0  0  1  1  0  1  1  0  0  1  1 -289952.0       1
['BINARY', 1 rows, 1 samples, 17 variables]
```

```
# Mapping of variable indices to names
if(total_flight_after_reduction>0):
    index_to_name = {index: var_name for index, var_name in enumerate(qubo.va

# Print variable names with their values
    for index, value in best_solution.items():
        var_name = index_to_name.get(index)
        print(f"{var_name}: {value}")
```

✓ 0.0s

```
q(1,3,1,5) (binary): 0
q(2,0,0,4) (binary): 1
q(2,1,2,0) (binary): 0
q(2,1,2,2) (binary): 0
q(2,1,2,7) (binary): 0
q(3,0,3,1) (binary): 0
q(3,0,3,3) (binary): 0
q(3,0,4,6) (binary): 0
```

Average QPU_Runtime:
0.016 seconds

Accuracy: Approx 95%
(Reason: Constrained Quadratic Modelling)

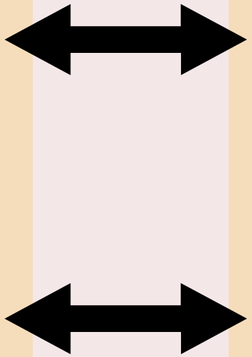
Paths Found: Top K paths, with time complexity increasing with K linearly

STAGE 2: PASSENGER REALLOCATION

GA

(Genetic Algorithm)

- Inspired from the nature, it is one of the most promising algorithm to solve combinatorial optimization problems.
- For its implementation to such kind of problems, it needs a problem specific approach for the algorithm to converge faster.
- The drawback of this algorithm is that it is prone to failure if it gets stuck in a local minima.



QGA

(Quantum Genetic Algorithm)

- QGA has sizeably large search space in comparison to its classical counter-part which provides faster convergence.
- It is more robust to getting stuck in local optima due to its adaptive rotation angle strategy.
- The drawback faced in it is due to the gate based model due to which errors accumulate for large problem sizes and it does not work optimally in that scenario.

REJECTION LIST

QAOA

(Quantim Optimization Algorithm)

- QAOA is gate based model which aims to solve combinatorial optimization problems by slowly taking the initial ansatz towards convergence by using different operators.
- When problem size becomes large it has to apply a very large number of gates to reach convergence due to which errors accumulate due to solution cannot be deemed to be feasible anymore.
- QAOA by qiskit utilises a classical solver for hyper tuning the parameterised circuits due to which its runtime scales up substantially for large problem sizes.

IMPLEMENTATION – STAGE 2: REALLOCATION

The methods we tried but rejected

```
Weights:  [95, 4, 60, 32, 23, 72, 80, 62, 65, 46]
Values:  [55, 10, 47, 5, 4, 50, 8, 61, 85, 87]
Knapsack Size:  269

[10.775767588452917, 31.690732432267 71, 9.597849187232892, 18.325826107939626, 26.893666051874746, 15.323100008233538, 8.548038078105542, 5.845498199398683, 19.580448863
248538, 8.325104282202659]
Best String:  1000000111
Best Value:  288
```

Genetic Algorithm

Rejection reason: Runtime too high, slow Convergence, High Number of Generations Needed

Quantum Genetic Algorithm

Rejection reason: Gate Based model accumulates errors for large sizes, Runtime too high

```
import time

start = time.perf_counter()
p = [3,1,2]
w = [1,2,3]
C = 4
theta_max = np.pi
theta_min = -np.pi
t_max = 100

print(Q_Genetic_Algorithm(p,w,C,theta_max,theta_min,t_max))
end = time.perf_counter()
print("RUNTIME: ", end-start)
```

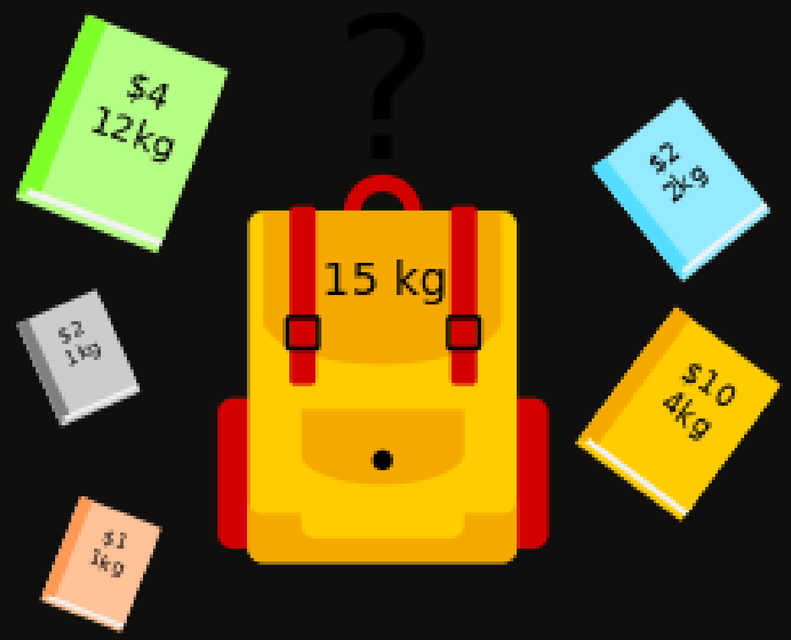
```
['000', '100', '100', '101', '101', '101', '101', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '
111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '11
1', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111'
, '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111',
'111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '111', '1
11', '111', '111', '111', '111', '111', '111', '111', '101', '101', '101', '110', '011']
RUNTIME:  28.632366063000518
```

Knapsack Problem

The art of Linear Programming helps to refine this problem as a multiknapsack problem with different classes.

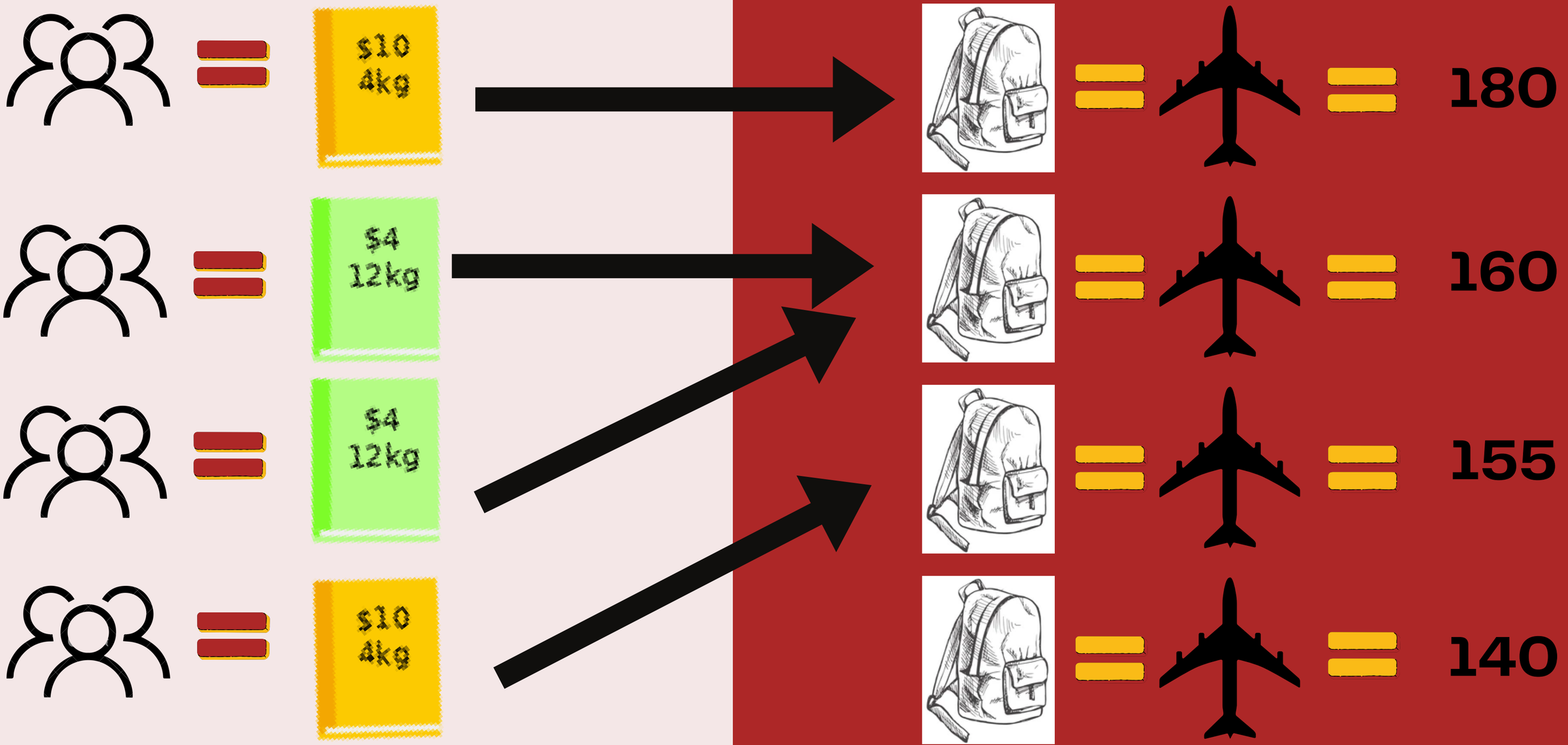
$$max \sum_{j=1}^{n} p_j x_j$$

$$max \sum_{j=1}^{n} w_{ij} x_j \leqslant c_i$$



Reference: A Awasthi et al. Quantum computing techniques for multi-knapsack problems, 10.48550. arXiv preprint ARXIV.2301.05750, 2301, 2023

MODIFIED MULTIPLE KNAPSACKING



Constraints

$$max \sum_{c \in S_p} \sum_{i \in c} \alpha_i \sum_{j}^{N} \sum_{k}^{K} p_{jk} x_{cijk}$$

Objective function which aims to maximize total PNR scoring while taking into account flight scoring

$$\sum_{j}^{N} \sum_{c \in S_f} x_{cijk} . w_j \leq C_{ik} \quad \forall k \in K, \quad \forall i \in c$$

Capacity constraint for the class k of flight i

$$\sum_{i \in S_1} \sum_{k}^{K} x_{cijk} \leq 1 \quad \forall j \in (1, 2, ..., N), \quad \forall c \in S_p$$

A passenger at best can be accommodated to one of the flights leaving from the departure airport

$$\sum_{i \in S_2} \sum_{k}^{K} x_{cijk} \leq 1 \quad \forall j \in (1, 2, ..., N), \quad \forall c \in S_p$$

A passenger at best can be accommodated to one of the flights reaching the destination airport.

$$\sum_{c \in S_p} \left(\sum_{i \in F_I} \sum_{k}^{K} x_{cijk} - \sum_{i \in F_O} \sum_{k}^{K} x_{cijk} \right) = 0 \quad \forall j \in 1, 2, ..., N$$

Ensures if a PNR reaches an intermediate airport then it should leave it as well

$$\sum_{c \in S_p} \left(\sum_{i \in F_c} \sum_{k}^{K} x_{cijk} - N_c . \sum_{k}^{K} x_{cfjk} \right) = 0 \quad \forall j \in 1, 2, ..., N$$

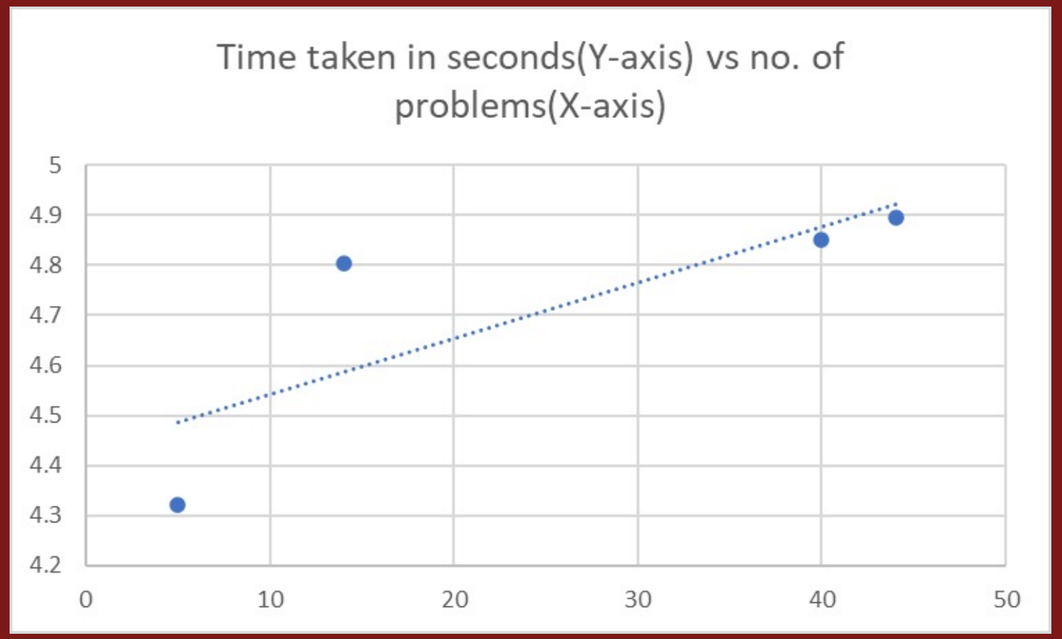
Deals with preserving the path of the PNR on which it travels

HYPERPARAMETER NOTE:

How did we choose our alpha parameter?

RESULTS

Quantum based knapsacking



If Problem size is scaled by 8 times, then the increase in QPU runtime is 3.3% !!!

QPU_ACCESS_TIME

00 h : 00 m : 00.016 s

CHARGE_TIME

00 h : 00 m : 05.000 s

RUN_TIME

00 h : 00 m : 05.091 s

QPU_ACCESS_TIME

00 h : 00 m : 00.032 s

CHARGE_TIME

00 h : 00 m : 04.759 s

RUN_TIME

00 h : 00 m : 04.759 s

QPU_ACCESS_TIME

00 h : 00 m : 00.016 s

CHARGE_TIME

00 h : 00 m : 04.320 s

RUN_TIME

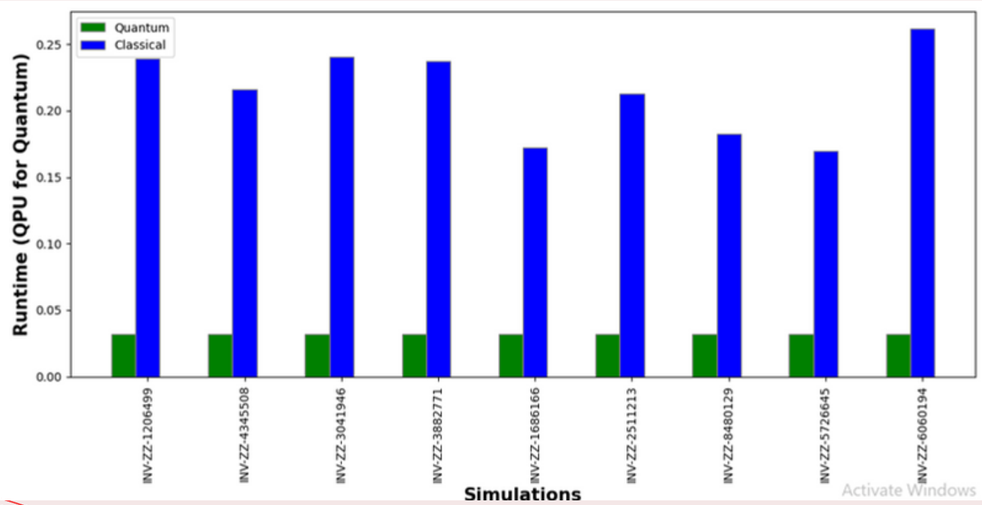
00 h : 00 m : 04.320 s

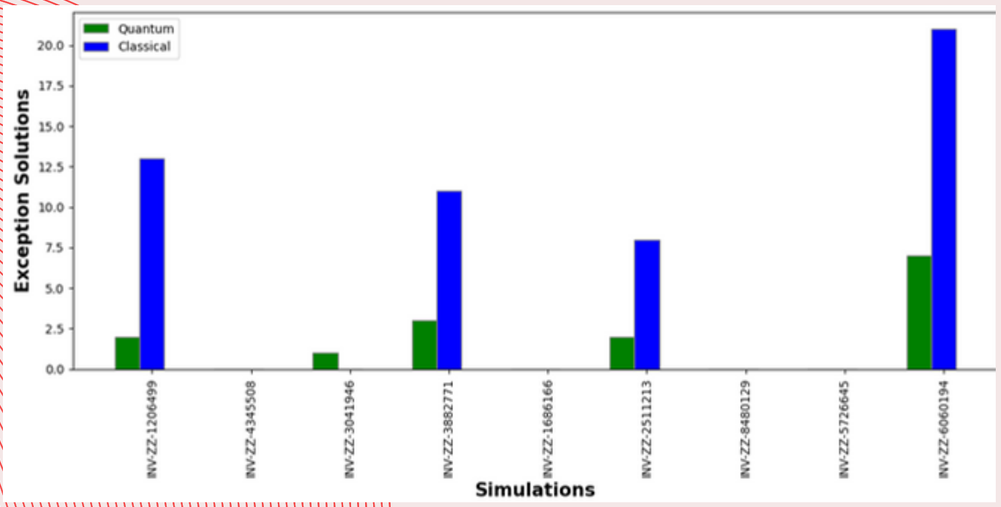
BEST CLASSICAL APPROACH

- For the classical approach to this problem we were unable to use class constrained multi - knapsack analog (NP - HARD problem)

- Instead we used a simple greedy approach to allocate highest ranking PNRs to the highest ranking flights

Stage 2 (Reallocation): Runtime Comparison

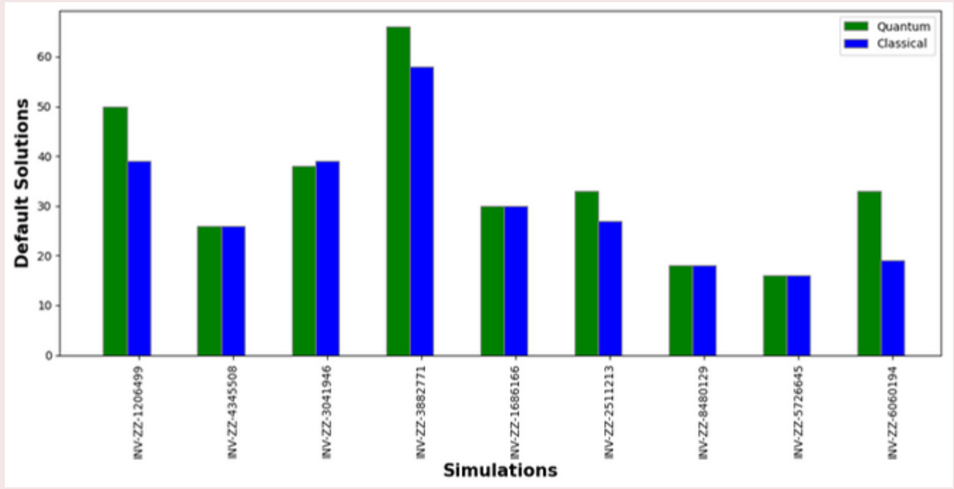




Classical	Quantum
Average Number of Paths: 4.89 Paths	Average Number of Paths: 4.89 Paths
Average Ratio of Default to Exception: 87.0778698 %	Average Ratio of Default to Exception: 96.22529242 %
Average Runtime: 0.27394458 seconds	Average Runtime: 0.032 seconds
Class Violations: Nil	Class Violations: Nil

8.5 Times Faster !!!

Higher Overall Score with better Optimizations !!!



Overall Comparison

Classical – Classical

- Paths Found: All possible Paths found 100% of the time
- Solution Correctness/ Efficiency: Fails to allocate efficiently. Not all passengers are allocated in some cases
- Scaling of Problem Size: Capable of finding paths efficiently but runtime blows up exponentially for reallocation
- Total Average Runtime: 0.30394458 seconds

Classical – Quantum

- Paths Found: All possible Paths found 100% of the time
- Solution Correctness/ Efficiency: Efficiently Allocates the passengers while maximizing scores
- Scaling of Problem Size: Grows roughly linearly with increase in problem size (low value of coefficient)
- Total Average Runtime: 0.0299+0.032 = 0.0619 seconds (Excluding Sever Access Time)

Quantum – Quantum

- Paths Found: Top K Paths found with accuracy approx. 95%
- Solution Correctness/ Efficiency: Efficiently Allocates the passengers while maximizing scores
- Scaling of Problem Size: Grows roughly linearly with increase in problem size (low value of coefficient)
- Total Average Runtime: 0.016+0.032 = 0.0.048 seconds (Excluding Server Access Time)

FINDING NEARBY FLIGHTS

Unique Selling Point

- Implemented Code to automatically change the search radius for city pairs as a Business Rule
- This can also be manually setup for faster search times
- Similar method can be used for finding connection flights used in real life

```python
def nearby_airport(cancelled_flight_departure_airport, radius):
    df = pd.read_csv('GlobalAirportDatabase.csv')

    reference_airport_iata = cancelled_flight_departure_airport
    search_radius_km_D = radius

    df_airports = df

    if reference_airport_iata in df_airports['IATA'].values:
        reference_airport = df_airports[df_airports['IATA'] == reference_airport_iata].iloc[0]
        ref_lat, ref_lon = reference_airport['Decimal Latitude'], reference_airport['Decimal Longitude']

        df_airports['Distance_km'] = df_airports.apply(
            lambda row: haversine(ref_lon, ref_lat, row['Decimal Longitude'], row['Decimal Latitude']),
            axis=1
        )
        df_nearby_airports = df_airports[df_airports['Distance_km'] <= search_radius_km_D]

        df_nearby_airports = df_nearby_airports[df_nearby_airports['IATA'] != reference_airport_iata]

        df_nearby_airports.dropna(subset=['IATA'], inplace=True)
        E_prime_d = df_nearby_airports['IATA'].tolist()
    else:
        return []
        pass
    return E_prime_d

print(nearby_airport('DEL', 150))
print(nearby_airport('DEL', 750))
print(nearby_airport('DEL', 1000))
```

```
[]
['LYP', 'LHE', 'MFG', 'MUX', 'RYK', 'ISB', 'RAZ', 'BHO', 'IDR', 'JLR', 'HJR', 'UDR', 'YOP', 'AGR', 'I
XD', 'ATQ', 'VNS', 'KUU', 'IXC', 'DED', 'GWL', 'JOH', 'JAI', 'JSA', 'IXJ', 'KNU', 'KTU', 'LUH', 'IXL'
, 'LKO', 'IXP', 'PGH', 'SXR', 'BWA', 'PKR']
['JAA', 'LYP', 'GIL', 'HDD', 'LHE', 'MFG', 'MJD', 'MUX', 'WNS', 'PEW', 'RYK', 'ISB', 'RAZ', 'SKZ', 'S
DT', 'SUL', 'BDN', 'PZH', 'AMD', 'AKD', 'IXU', 'PAB', 'BHJ', 'BDQ', 'BHO', 'BHU', 'IDR', 'JLR', 'JGA'
, 'IXY', 'HJR', 'NAG', 'RAJ', 'RPR', 'STV', 'UDR', 'YOP', 'GAY', 'PAT', 'AGR', 'IXD', 'ATQ', 'VNS', '
KUU', 'IXC', 'DED', 'GWL', 'JOH', 'JAI', 'JSA', 'IXJ', 'KNU', 'KTU', 'LUH', 'IXL', 'LKO', 'IXP', 'PGH
', 'SXR', 'BWA', 'KTM', 'PKR', 'SIF', 'HTN']
```

Business Proposal and USPs

U

Solved Class Constrained Multiple Knapsack Problem with Quantum Annealing (Never solved before in Literature)

S

New Business Rule to Automatically Solve for City Pairs

P

Solves for multiple solution files using a single iteration (rather that looping repeatedly for different solutions)

WE ARE NOW OPEN TO ANY AND ALL QUESTIONS