



# Solving Travelling salesman problem using Quantum Computing

Under the guidance of Prof. Vijay Kumar Chaurasiya


Kalyani Pharkendekar IIT2020196  
Anurag Harsh IIB2020016  
Aman Utkarsh IIB2020027  
Priyamvada Priyadarshani IIB2020037  
Dhamapurkar Nikita Suresh IIB2020042



# Introduction

Traveling Salesman Problem (TSP) is one of the most widely known Combinatorial Optimization Problems.

It involves finding the shortest path while traveling  $N$  cities and returning back to the starting one, with the constraint of visiting every city only once.

- 
- The Traveling Salesman Problem has been widely studied.
  - The problem has many applications in various fields including deliveries, vehicle routings and crystal theory.



Quantum Computing based on the principle of Quantum Mechanics has shown to provide significant speedups to its classical counterparts on certain problems , Combinatorial Optimization one among them.

In this presentation we aim to present a study and understanding of quantum computing by applying a Quantum Optimization Algorithm on the Traveling Salesman Problem (TSP).



# Earlier Presentations

- In earlier assessments we presented
  - Intro to TSP problem
  - Classical Approach of solving TSP problem
  - our study of quantum computation theory



# Proposed Methodology

- Formulate the problem as a combinatorial optimization problem.
- Map the problem to an Ising problem and calculate the Ising Hamiltonian.
- Use numpy Min EigenSolver to find the lowest energy state of the Ising Hamiltonian.
- That uses a quantum circuit to approximate the ground state (lowest energy state) of the Ising Hamiltonian, which corresponds to the optimal solution of the optimization problem.
- Comparing it with D-wave Solution



# BRUTE FORCE APPROACH

# DRAW GRAPH FUNCTION:

- Defined a function called draw\_graph

G: A NetworkX graph object

colors: A list of colors, one for each node in the graph

pos: A dictionary of node positions

- Draw the graph on the axes object - axes will be visible when the graph is drawn.
- Get the edge labels for the graph - edge labels are stored in a dictionary.
- Draw the edge labels on the graph.

```
[7]: def draw_graph(G, colors, pos):  
      default_axes = plt.axes(frameon=True)  
      nx.draw_networkx(G, node_color=colors, node_size=600, alpha=0.8, ax=default_axes, pos=pos)  
      edge_labels = nx.get_edge_attributes(G, "weight")  
      nx.draw_networkx_edge_labels(G, pos=pos, edge_labels=edge_labels)
```



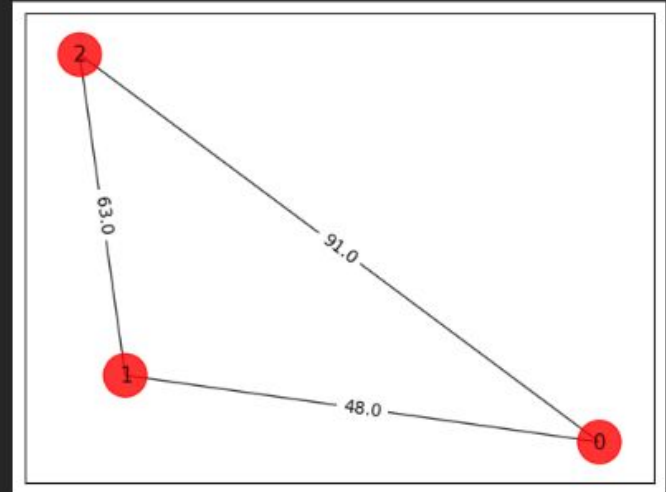
- Set n=3
- Create a TSP instance - the seed parameter ensures reproducible results for the random instance generation.
- Convert TSP graph to an adjacency matrix - converted the NetworkX graph object (tsp.graph) to a NumPy
- Print the distance matrix
- Set the node colours and positions.
- Draw the TSP graph.

```
[8]: n = 3

tsp = Tsp.create_random_instance(n, seed=123)
adj_matrix = nx.to_numpy_array(tsp.graph)
print("distance\n", adj_matrix)

colors = ["r" for node in tsp.graph.nodes]
pos = [tsp.graph.nodes[node]["pos"] for node in tsp.graph.nodes]
draw_graph(tsp.graph, colors, pos)

distance
[[ 0. 48. 91.]
 [48.  0. 63.]
 [91. 63.  0.]]
```





# Brute force function

- Takes two arguments:  $w$  and  $N$ .  $w$  is a 2D NumPy array representing the distances between all pairs of cities, and  $N$  is the number of cities.
- Generate all possible permutations excluding the starting city and store it in a list called 'a'
- $\text{last\_best\_distance} = 1e10$
- Outer loop iterates over all the permutations.
- $\text{distance} \rightarrow$  track total distance from the current permutation
- $\text{pre\_j} \rightarrow$  keep track of previous city visited
- inner loop iterates over the cities of the current permutation
- Add the distance, update  $\text{pre\_j}$  to the current city
- At the end of the nested loop, add distance between the last city visited and the starting city to the "distance" variable
- add city 0 followed by cities in the current permutation to the tuple "order"
- check if the current permutation has a shorter distance than shortest distance so far
- updates the  $\text{best\_order}$  variable to the current permutation and the  $\text{last\_best\_distance}$  variable to the current distance
- return  $\text{last\_best\_distance}$  and  $\text{best\_order}$

# Brute force function

```
[10]: def brute_force_tsp(w, N):
      a = list(permutations(range(1, N)))
      last_best_distance = 1e10
      for i in a:
          distance = 0
          pre_j = 0
          for j in i:
              distance = distance + w[j, pre_j]
              pre_j = j
          distance = distance + w[pre_j, 0]
          order = (0,) + i
          if distance < last_best_distance:
              best_order = order
              last_best_distance = distance
              print("order = " + str(order) + " Distance = " + str(distance))
      return last_best_distance, best_order
```

```
[11]: best_distance, best_order = brute_force_tsp(adj_matrix, n)
      print(
          "Best order from brute force = "
          + str(best_order)
          + " with total distance = "
          + str(best_distance)
      )
```

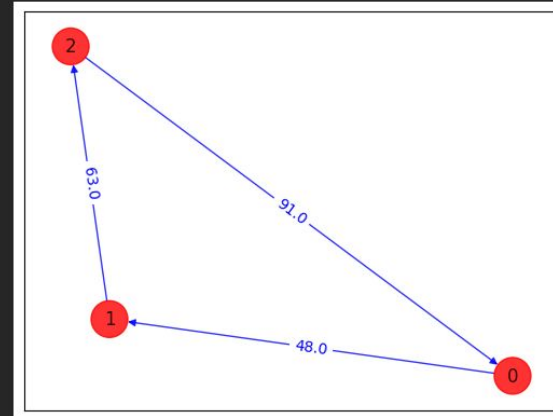
order = (0, 1, 2) Distance = 202.0

Best order from brute force = (0, 1, 2) with total distance = 202.0

# Draw TSP function

- Define a function called `draw_tsp_solution` that takes four arguments: `G`, `order`, `colors`, and `pos`.
- Create a directed graph object (`G2`)
- Add all the nodes from the original graph (`G`) to (`G2`).
- Add edge, create axes, draw `G2`, get edge labels for `G2`, draw the edge labels
- Call `draw_tsp_solution()`

```
[12]: def draw_tsp_solution(G, order, colors, pos):  
      G2 = nx.DiGraph()  
      G2.add_nodes_from(G)  
      n = len(order)  
      for i in range(n):  
          j = (i + 1) % n  
          G2.add_edge(order[i], order[j], weight=G[order[i]][order[j]]["weight"])  
      default_axes = plt.axes(frameon=True)  
      nx.draw_networkx(  
          G2, node_color=colors, edge_color="b", node_size=600, alpha=0.8, ax=default_axes, pos=pos  
      )  
      edge_labels = nx.get_edge_attributes(G2, "weight")  
      nx.draw_networkx_edge_labels(G2, pos, font_color="b", edge_labels=edge_labels)  
  
draw_tsp_solution(tsp.graph, best_order, colors, pos)
```





# Mapping to ising problem



# Converting TSP into a Quadratic Problem (QP)

```
qp = tsp.to_quadratic_program()  
print(qp.prettyprint())
```

Problem name: TSP

Minimize

$$48x_{0_0}x_{1_1} + 48x_{0_0}x_{1_2} + 91x_{0_0}x_{2_1} + 91x_{0_0}x_{2_2} + 48x_{0_1}x_{1_0} + 48x_{0_1}x_{1_2} + 91x_{0_1}x_{2_0} + 91x_{0_1}x_{2_2} + 48x_{0_2}x_{1_0} + 48x_{0_2}x_{1_1} + 91x_{0_2}x_{2_0} + 91x_{0_2}x_{2_1} + 63x_{1_0}x_{2_1} + 63x_{1_0}x_{2_2} + 63x_{1_1}x_{2_0} + 63x_{1_1}x_{2_2} + 63x_{1_2}x_{2_0} + 63x_{1_2}x_{2_1}$$

Subject to

Linear constraints (6)

$$\begin{aligned}x_{0_0} + x_{0_1} + x_{0_2} &= 1 && \text{'c0'} \\ x_{1_0} + x_{1_1} + x_{1_2} &= 1 && \text{'c1'} \\ x_{2_0} + x_{2_1} + x_{2_2} &= 1 && \text{'c2'} \\ x_{0_0} + x_{1_0} + x_{2_0} &= 1 && \text{'c3'} \\ x_{0_1} + x_{1_1} + x_{2_1} &= 1 && \text{'c4'} \\ x_{0_2} + x_{1_2} + x_{2_2} &= 1 && \text{'c5'}\end{aligned}$$

Binary variables (9)

$$x_{0_0} \ x_{0_1} \ x_{0_2} \ x_{1_0} \ x_{1_1} \ x_{1_2} \ x_{2_0} \ x_{2_1} \ x_{2_2}$$

- A Quadratic Problem, also known as QP consists of a set of variables, an objective function that defines the quantity to be minimized or maximized, and a set of constraints that restrict the allowable values of the variables.
- To convert the TSP to a QP, we define a binary variable for each possible edge in the graph. If the corresponding edge is included in the TSP solution, the variable is set to 1; otherwise, it is set to 0. The objective function then becomes the sum of the weights of the selected edges.



# Converting QP into Quadratic Unconstrained Binary Optimization (QUBO) Problem

- QUBO is a special type of QP where the objective function consists only of terms that are either linear or quadratic in the variables. This form is particularly suitable for quantum optimization algorithms, as it can be efficiently represented using quantum circuits.
- The conversion from QP to QUBO involves transforming the quadratic terms into linear terms using additional variables and constraints. This transformation ensures that the problem remains equivalent but can be expressed in the QUBO format.





# Converting QUBO into Ising Problem

- Converting a QUBO (Quadratic Unconstrained Binary Optimization) to an Ising Problem is a crucial step in utilizing quantum algorithms for solving optimization problems. The Ising Hamiltonian represents the energy landscape of the problem in terms of interactions between qubits, enabling quantum computers to explore and identify the optimal solution.
- An Ising Hamiltonian is a specific type of Hamiltonian tailored for problems like TSP. It uses a set of qubits (quantum bits) as its building blocks. Each qubit can be in either a 0 or 1 state, representing the binary choices in the TSP problem (e.g., edge inclusion or exclusion).



```
from qiskit_optimization.converters import QuadraticProgramToQubo
```

```
qp2qubo = QuadraticProgramToQubo()  
qubo = qp2qubo.convert(qp)  
qubitOp, offset = qubo.to_ising()  
print("Offset:", offset)  
print("Ising Hamiltonian:")  
print(str(qubitOp))
```

Offset: 7581.0

Ising Hamiltonian:

```
-1282.5 * IIIIIIII  
- 1282.5 * IIIIIIZI  
- 1282.5 * IIIIIIZII  
- 1268.5 * IIIIIZIII  
- 1268.5 * IIIIZIIII  
- 1268.5 * IIIZIIIII  
- 1290.0 * IIZIIIIII  
- 1290.0 * IZIIIIIII  
- 1290.0 * ZIIIIIIII  
+ 606.5 * IIIIIIZZ  
+ 606.5 * IIIIIIZIZ  
+ 606.5 * IIIIIIZZI  
+ 606.5 * IIIIIZIIZ  
+ 12.0 * IIIIIZIZI  
+ 12.0 * IIIIIZIZI  
+ 12.0 * IIIIIZII  
+ 12.0 * IIIIZIIIZ  
+ 606.5 * IIIIZIIZI  
+ 12.0 * IIIIZIZII  
+ 606.5 * IIIIZZIII  
+ 12.0 * IIIZIIIIZ  
+ 12.0 * IIIZIIIZI  
+ 606.5 * IIIZIIIZI  
+ 606.5 * IIIZIIIZI  
...  
+ 15.75 * ZIIIZIIII  
+ 606.5 * ZIIIZIIII  
+ 606.5 * ZIZIIIIII  
+ 606.5 * ZZIIIIIII
```



# Solving for the optimized value of the objective function

- The ground state of the Ising Hamiltonian corresponds to the optimal solution of the objective function.
- We need to compute the energy associated with the ground state and the state itself.
- These can be obtained by computing the eigenstates and eigenvectors associated with the Ising Operator.



# EigenStates and EigenVectors

- Eigenstates are to operators what Eigenvectors are to matrices.
- For example, if  $|\psi\rangle$  is an eigenstate of the operator  $A$ , then  $A|\psi\rangle = \lambda|\psi\rangle$ , where  $\lambda$  is the corresponding eigenvalue.
- The eigenstates represent the quantum states of a system.
- The ground state of the system corresponds to the eigenstate associated with the lowest eigenvalue.



# Eigenvalues

- The eigenvalues of a Hamiltonian represent the possible energy levels of the quantum system.
- Each eigenvalue corresponds to a specific energy state that the system can occupy.
- The ground state of a system has the lowest eigenvalue and excited states correspond to have higher eigenvalues.
- The differences between eigenvalues are associated with the energy transitions that can occur in the system.



# Calculating the Eigenstates and Eigenvalues

- There are two ways to calculate the eigenstate and the eigenvalues of the Ising Operator.
- NumpyEigenSolver - Used for finding the minimum eigenvalue of a given matrix using NumPy, which is a popular numerical computing library for Python. Takes the Ising Operator as an input and provides the minimum eigenvalue as the output.
- SamplingVQE - Uses an optimizer (SPSA) to find the minimum energy associated with the quantum state. SPSA is a gradient descent method for optimizing systems with multiple unknown parameters.

# Using NumpyEigenSolver

```
# Computing the minimum eigenvalue and thus the solution of the Objective function
```

```
eS = NumPyMinimumEigensolver()  
eigenvalue_result = eS.compute_minimum_eigenvalue(IsingOperator)  
x = tsp.sample_most_likely(eigenvalue_result.eigenstate)  
print("energy:", eigenvalue_result.eigenvalue.real)  
print("feasible:", qubo.is_feasible(x))  
solver = MinimumEigenOptimizer(numpyEigenSolver())  
result = solver.solve(qubo)  
print(result.prettyprint())
```

energy: -7379.0

feasible: True

objective function value: 202.0

variable values: x\_0\_0=1.0, x\_0\_1=0.0, x\_0\_2=0.0, x\_1\_0=0.0, x\_1\_1=1.0, x\_1\_2=0.0, x\_2\_0=0.0, x\_2\_1=0.0, x\_2\_2=1.0

status: SUCCESS



# Using SamplingVQE

```
: optimizer = SPSA(maxiter=300)
ry = TwoLocal(IsingOperator.num_qubits, "ry", "cz", reps=5, entanglement="linear")
vqe = SamplingVQE(sampler=Sampler(), ansatz=ry, optimizer=optimizer)|
eigenvalue_result = vqe.compute_minimum_eigenvalue(IsingOperator)
print("energy:", eigenvalue_result.eigenvalue.real)
x = tsp.sample_most_likely(eigenvalue_result.eigenstate)
print("feasible:", qubo.is_feasible(x))
z = tsp.interpret(x)
print("solution: ",z)
print("solution objective:", tsp.tsp_value(z, adj_matrix))
draw_tsp_solution(tsp.graph, z, colors, pos)
```

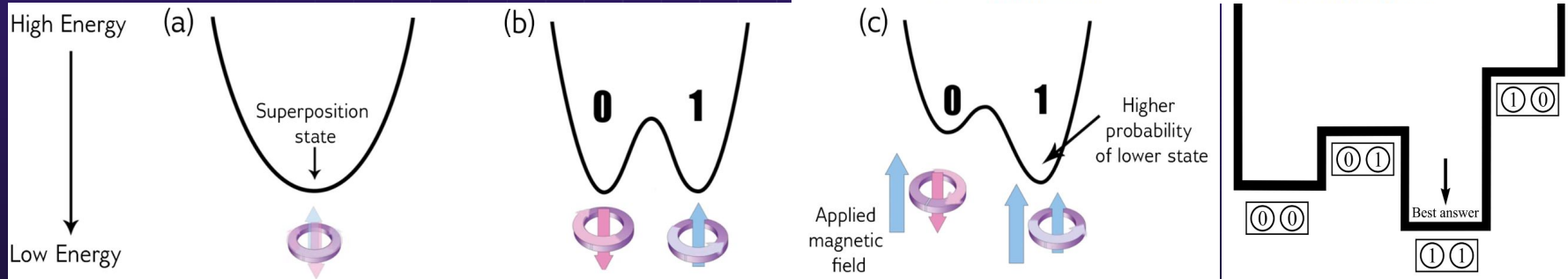
```
energy: -7326.02469952184
feasible: True
solution: [0,1,2]
solution objective: 202.0
```



# Simulated Annealing & Adiabatic Quantum computers

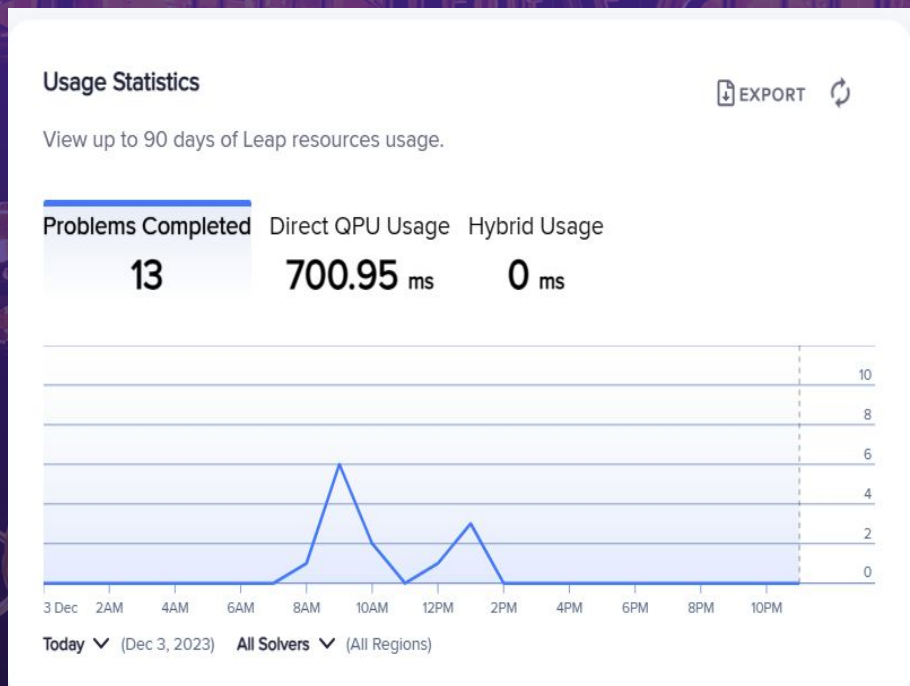
- Simulated annealing: Stochastic global optimization technique which brings a system to minimum energy state (objective function mimics energy).
- Aims to Realize adiabatic quantum computation through quantum annealing where the Hamiltonian is constrained to be an Ising Hamiltonian.
- Ising Hamiltonian is transformed to QUBO problem & solved using D-Wave QPU. Use Lagrange multipliers to add constraints.
- Quantum Tunneling

$$\mathcal{H}_{ising} = \underbrace{-\frac{A(s)}{2} \left( \sum_i \hat{\sigma}_x^{(i)} \right)}_{\text{Initial Hamiltonian}} + \underbrace{\frac{B(s)}{2} \left( \sum_i h_i \hat{\sigma}_z^{(i)} + \sum_{i>j} J_{i,j} \hat{\sigma}_z^{(i)} \hat{\sigma}_z^{(j)} \right)}_{\text{Final Hamiltonian}}$$





# Dashboard view & Peek to D-Wave Annealer Code



[Advantage Performance white paper](#) : Learn more about D-Wave Advantage5.4 & embedding

```
%time embedded_big_route = dnx.traveling_salesperson(G, Embedd
print("Route found with simulated annealing:", embedded_big_route)
```

```
CPU times: user 1min 15s, sys: 525 ms, total: 1min 16s
```

```
Wall time: 1min 20s
```

```
Route found with simulated annealing: [0, 7, 2, 3, 9, 6, 8, 4,
```

```
total_dist=0
```

```
for idx, node in enumerate(embedded_big_route[:-1]):
    dist = data10[embedded_big_route[idx+1]][embedded_big_route[idx]]
    total_dist += dist
print("Total distance exact_route (Without return):" , total_dist)
return_distance= data10[embedded_big_route[0]][embedded_big_route[-1]]
print("Distance between exact_route start and end", return_distance)
distance= total_dist+ return_distance
print("total including return: exact_route", distance)
```

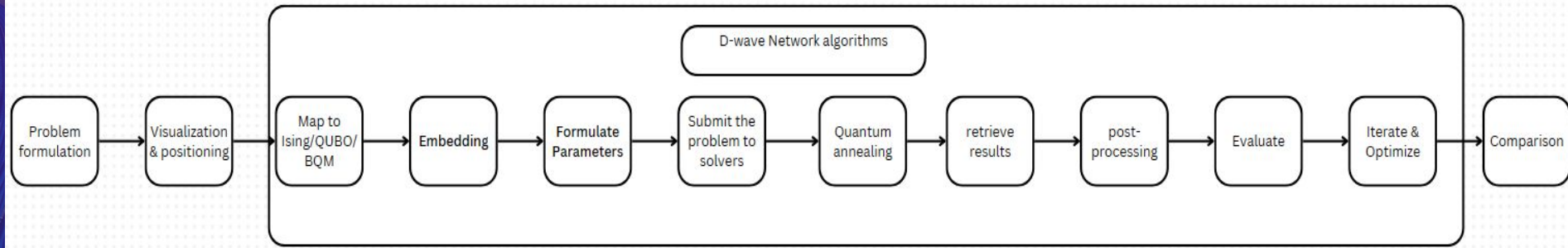
```
Total distance exact_route (Without return): 355
```

```
Distance between exact_route start and end 52
```

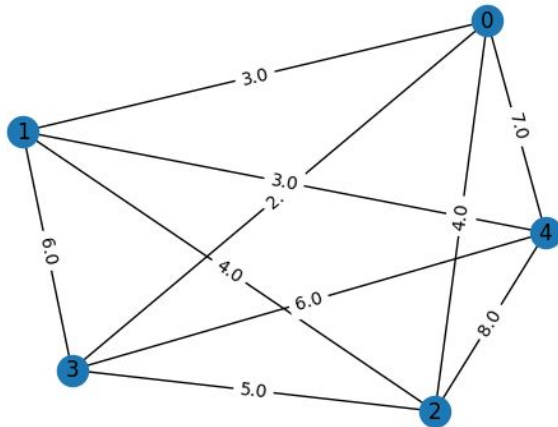
```
total including return: exact_route 407
```

Code link: [https://colab.research.google.com/drive/1uFtkpcsn\\_Gs60HSI0o7JOY\\_aIVvhfxNx#scrollTo=jMuYbb4iJJSX](https://colab.research.google.com/drive/1uFtkpcsn_Gs60HSI0o7JOY_aIVvhfxNx#scrollTo=jMuYbb4iJJSX)

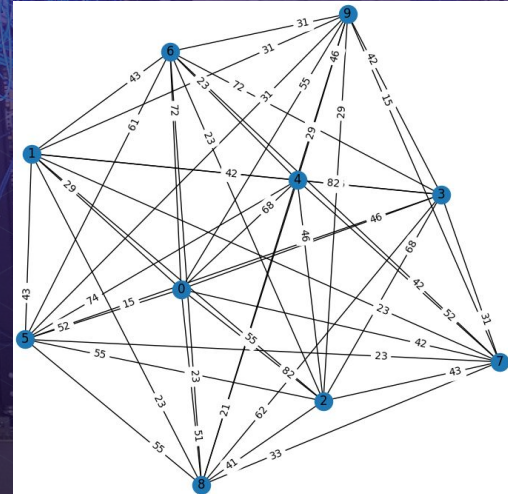
# Proposed Methodology flowchart & Problem formulation



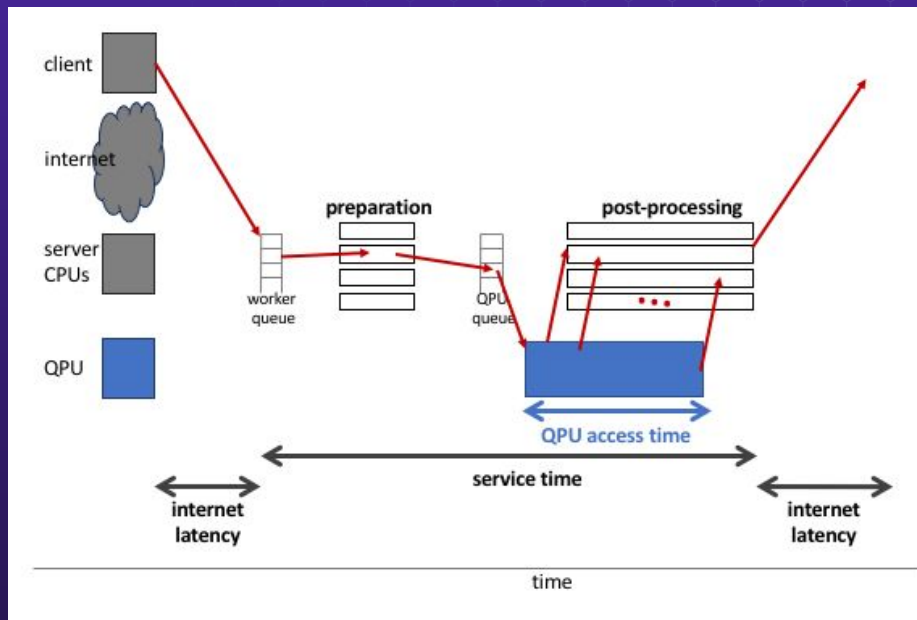
Example Problem 1



Example Problem 2



# Code details, functions used



D-Wave's quantum computers, such as those using the Chimera architecture, require the user to map their problem onto the physical qubits and couplers of the quantum processing unit (QPU).

Exact solver for testing and debugging code using your local CPU.

SimulatedAnnealingSampler: classical solver alternative to quantum annealing for specific optimization problems.

Overview of execution of a single QMI, starting from a client system, and distinguishing classical (client, CPU) and quantum (QPU) execution. Source: [Operation and Timing – D-Wave System Documentation documentation \(dwavesys.com\)](https://dwavesys.com/documentation)





# Quantum Approach

`traveling_salesperson(G[, sampler, ...])`: Defines a QUBO with ground states corresponding to the minimum routes and uses the sampler to sample from it.

`traveling_salesperson_qubo(G[, lagrange, ...])`: Return the QUBO with ground states corresponding to a minimum TSP route.

`num_reads` in D-Wave's quantum annealing API determines how many times the quantum computer attempts to sample solutions, aiding in improving the likelihood of finding optimal outcomes through multiple runs of the probabilistic process.

**Lagrange Multiplier:** Control the trade-off between minimizing the objective function and satisfying constraints. It helps incorporate constraints into the optimization problem. e.g. Lagrange parameter (e.g.,  $\text{lagrange} = G.\text{size}(\text{weight}=\text{weight}) * G.\text{number\_of\_nodes}() / G.\text{number\_of\_edges}()$ ) automates its calculation based on graph characteristics. It balances density and total cost, crucial for TSP optimization by fine-tuning constraints and objectives.

# Comparison

$n$	Number of paths	Time ( $1\mu s/\text{chemin}$ )
5	12	$12\mu s$
10	181,440	0.18s
15	$4.359 \times 10^{10}$	12h
20	$6.082 \times 10^{16}$	1,928 years
61	$4.160 \times 10^{81}$	$13.19 \times 10^{67}$ years

[Traveling Salesman Problem - an overview | ScienceDirect Topics](#)

Number of cities	Brute Force	Gate quantum computer
3	0.00042176s	31s
6	0.0010166s	Kernel Crashed

Number of cities	Simulated Annealing (using Classical computer)	Exact Solver	D-Wave Sampler(Direct QPU Usage(CPU time in notebook))	D-Wave Sampler(num_reads=1000) (Direct QPU Usage)	Custom Lagrange multiplier(density based) & traveling_salesman_qubo
5	4.97s	1m 38s	31.88ms (3.67s-5.45s)	129.27ms	15ms
10	36.2s	None	31.89s (1min 36s)	226.25ms	None

# Comparison with Previous work

Paper: Solving the Traveling Salesman Problem on the D-Wave Quantum Computer Siddharth Jain 2021	Our Method
8 nodes, $n = 9$ ( $m = 36$ ), the D-Wave QPU did not return any valid solution for 5 out of 8 test cases, and for $n = 10$ ( $m = 45$ ), the QPU returned a solution for only 1 out of 8 test cases.	10 + nodes Supported
20 $\mu$ s, excludes the time spent in minor embedding of the problem graph	'qpu_anneal_time_per_sample': 20.0 $\mu$ s,
suboptimal compared to classical solver	optimal ans with repeated runs

# Conclusión

- Solved TSP using VQE on Gate model QC( QUBO-> Ising).
- Solved TSP using D-Wave Annealers.

## Observations:

- Classic Solvers always respects constraint.
- Quantum solution turns the constraints into soft-constraints that can be violated.
- Neither Gate QC or D-Wave QPU seemingly demonstrate any quantum advantage over classical.
- But this is clear that with increasing number of cities the time taken doesn't increase exponentially.
- With advancements in technology the quantum solution might outperform classical method.
- Future: Use QAOA & Hybrid Solvers



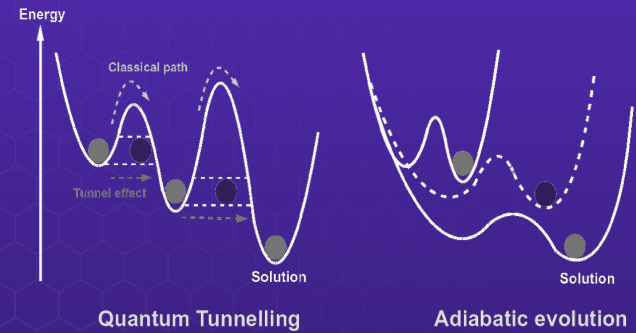
D-Wave Annealers



Gate Based Quantum Computer

# References

- [Variational Quantum Optimization algorithm for Quantum Computing](#)
- [Ising formulation of NP problems](#)
- Au-Yeung R, Chancellor N and Halffmann P (2023), NP-hard but no longer hard to solve? Using quantum computing to tackle optimization problems. Front. Quantum. Sci. Technol. 2:1128576. doi: 10.3389/frqst.2023.1128576
- “Solver Docs” D-Wave. [Solver Docs — D-Wave System Documentation documentation \(dwavesys.com\)](#)(accessed 2023).
- Jain S (2021) Solving the Traveling Salesman Problem on the D-Wave Quantum Computer. Front. Phys. 9:760783. doi: 10.3389/fphy.2021.760783
- Feld S, Roch C, Gabor T, Seidel C, Neukart F, Galter I, Mauerer W and Linnhoff-Popien C (2019) A Hybrid Solution Method for the Capacitated Vehicle Routing Problem Using a Quantum Annealer. Front. ICT 6:13. doi: 10.3389/fict.2019.00013
- Our report: [MINI\\_PROJECT\\_7TH\\_C3 - Google Docs](#)
- Our code:  
[https://colab.research.google.com/drive/1uFtkpcsn\\_Gs60HSI0o7JOY\\_alVVhfxNx#scrollTo=jMuYbb4jIJSX](https://colab.research.google.com/drive/1uFtkpcsn_Gs60HSI0o7JOY_alVVhfxNx#scrollTo=jMuYbb4jIJSX)





**THANK  
YOU!**

