REVOLUCIONARIOS de QUBITS

Computación cuántica para todos
(Gpo 101)

Revolucionarios de qubits equipo 18

INTEGRANTES

A07106565 - LUIS FELIPE MARINO PALAFOX

LUIS ÁNGEL GONZÁLEZ ROMO A01235962

MAYRA STEFANY GÓMEZ TRIANA A01625609

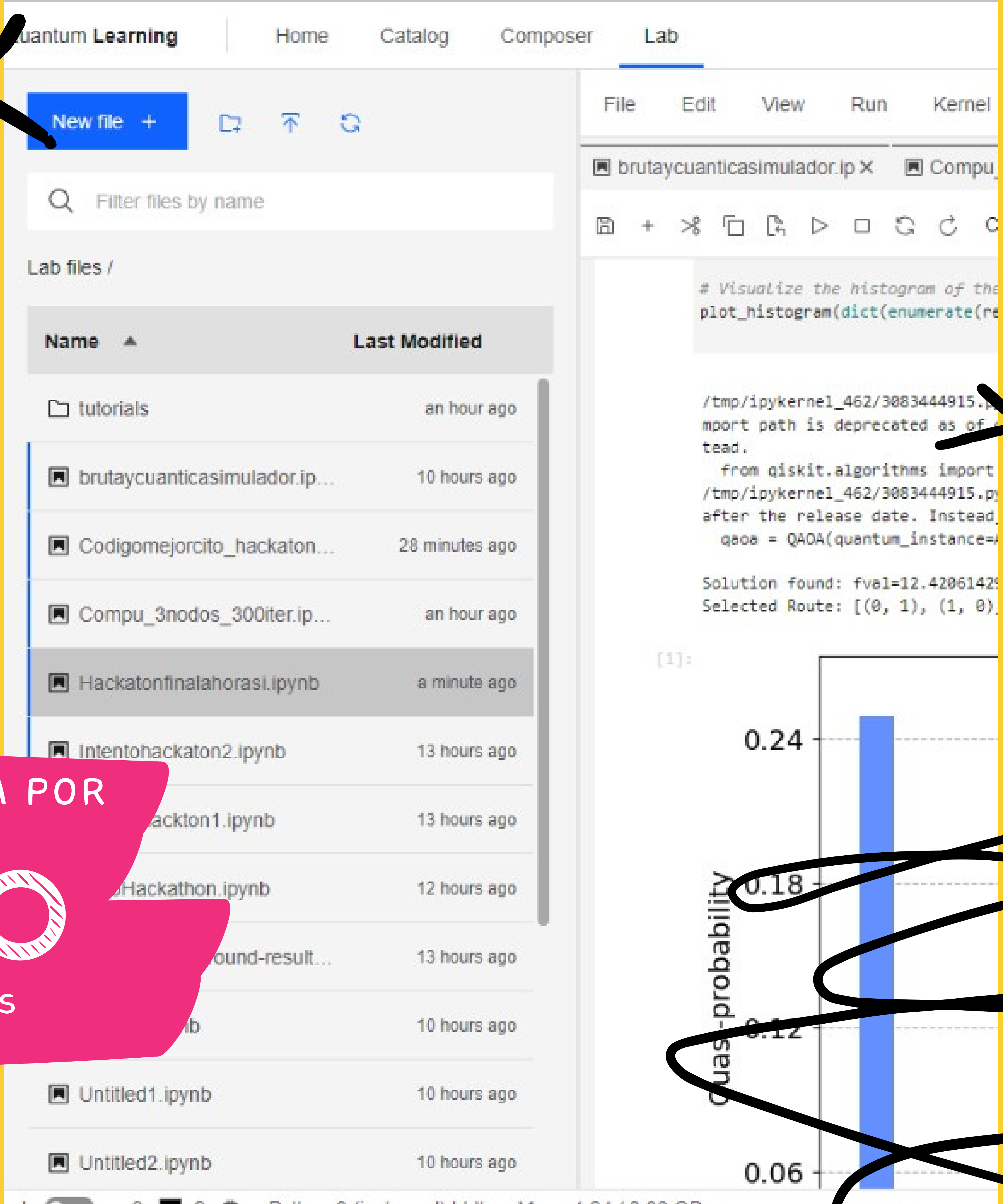
JUAN PABLO SOLÍS RUIZ A01067387

A01721790 - ADRIÁN GÓMEZ ORTIZ

OSCAR CRUZ ZEPEDA A01639263

Computación cuántica para todos (Gpo 101)

NUESTRA PELEA



VEA NUESTRA PELEA POR TAN SOLO

$4,200

pesos mexicanos

ESTRATEGIA

TSP

1. Representación de datos: Usamos una matriz o grafo para representar ciudades y distancias.

2. Algoritmo inicial: Aplicamos un algoritmo como el del vecino más cercano o mínimo árbol de expansión para obtener una solución inicial.

3. Optimización (opcional): Mejoramos la solución con métodos de computación cuántica.

CODIGO

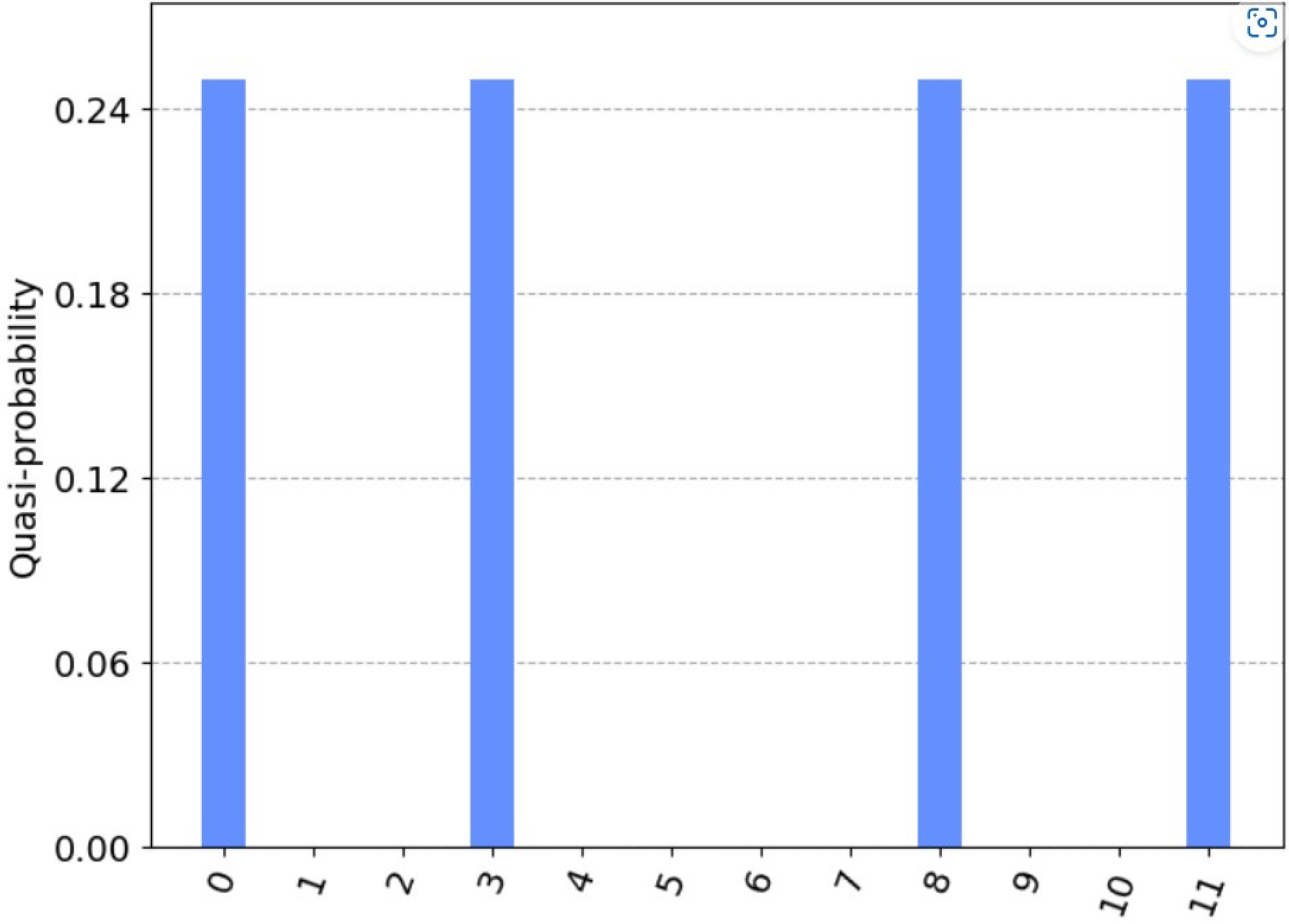
```python
import numpy as np
from qiskit import Aer
from qiskit.visualization import plot_histogram
from qiskit.utils import QuantumInstance
from qiskit.algorithms import QAOA, NumPyMinimumEigensolver
from qiskit_optimization import QuadraticProgram
from qiskit_optimization.algorithms import MinimumEigenOptimizer
from qiskit_optimization.problems import QuadraticProgram

# Define the coordinates of the cities
coordinates = np.array([[0.61817,6.23198], [3.50095,2.84561], [5.43635,3.70334], [5.42186,1.94033],[4.46827,1.96438],[5.81491,1.54681],[6.10634,1.41956],[8.8509,2.0681]])

# Calculate the distances between the cities (using Euclidean distance)
def calculate_distance(coord1, coord2):
    return ((coord1[0] - coord2[0]) ** 2 + (coord1[1] - coord2[1]) ** 2) ** 0.5

num_cities = len(coordinates)
distance_matrix = np.zeros((num_cities, num_cities))

for i in range(num_cities):
    for j in range(num_cities):
        distance_matrix[i][j] = calculate_distance(coordinates[i], coordinates[j])

# Create a quadratic optimization problem for the TSP
problem = QuadraticProgram()
for i in range(num_cities):
    for j in range(num_cities):
        if i != j:
            problem.binary_var(f'x_{i}_{j}')

# Constraints: each city must be visited exactly once
for i in range(num_cities):
    constraint = {f'x_{i}_{j}': 1 for j in range(num_cities) if j != i}
    problem.linear_constraint(linear=constraint, sense='E', rhs=1, name=f'visit_once_{i}')
```

```python
# Constraints: each city must be visited exactly once
for i in range(num_cities):
    constraint = {f'x_{i}_{j}': 1 for j in range(num_cities) if j != i}
    problem.linear_constraint(linear=constraint, sense='E', rhs=1, name=f'visit_once_{i}')

# Constraints: from each city, there should be exactly one transition
for j in range(num_cities):
    constraint = {f'x_{i}_{j}': 1 for i in range(num_cities) if i != j}
    problem.linear_constraint(linear=constraint, sense='E', rhs=1, name=f'leave_once_{j}')

# Objective function: minimize the total distance
linear_coeff = {f'x_{i}_{j}': distance_matrix[i][j] for i in range(num_cities) for j in range(num_cities) if i != j}
quadratic_coeff = {}
problem.minimize(linear=linear_coeff, quadratic=quadratic_coeff)

# Solve the problem using the QAOA algorithm
qaoa = QAOA(quantum_instance=Aer.get_backend('qasm_simulator'))
optimizer = MinimumEigenOptimizer(qaoa)

# Solve the problem
result = optimizer.solve(problem)
print('Solution found:', result)

# Map binary variable names to indices
var_to_indices = {(i, j): idx for idx, (i, j) in enumerate([(i, j) for i in range(num_cities) for j in range(num_cities) if i != j])}

# Extract the selected route from the binary variables
selected_route = []
for (i, j), idx in var_to_indices.items():
    if result.x[idx] == 1:
        selected_route.append((i, j))

# Print the selected route
print('Selected Route:', selected_route)

# Visualize the histogram of the solutions
plot_histogram(dict(enumerate(result.x)), figsize=(8, 6), bar_labels=False)
```

DISCUSION

Trata de minimizar tu impacto en el entorno natural y respeta las tradiciones y costumbres de la comunidad local.

[1]:



¡Muchas gracias por tu

CONCLUSION