Quantum Optimization and the Traveling Salesman Problem

Matthew Torre and Kai Roybal

The purpose of this paper and our project is to introduce the Quantum Approximate Optimization Algorithm (QAOA for short) as a method for solving the Traveling Salesman Problem (TSP), a famous combinatorial optimization problem. These are problems where the objective is to **find the best solution from a finite set of possible solutions x**. Our main focus is solving the Traveling Salesman Problem (TSP) but other examples of combinatorial optimization problems including the Max-Cut problem, and Minimum Spanning Tree.

We provide an overview of both classical and quantum approaches to the TSP, discussing computational complexities, quantum states, and the fundamental principles of quantum algorithms. We also conduct some experiments and demonstrations, where we can highlight the potential advantages of quantum computing in optimization problems.

1 Introduction

The Traveling Salesman Problem (TSP) is a classic problem in combinatorial optimization. Formulated in the 1930s, the TSP asks: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" This problem remains relevant and important because it is NP-hard, which allows it to act as a benchmark for many optimization methods as well as having a wide variety of applications in logistics, economic optimization, and engineering optimization.

2 Time Complexity of TSP

The Traveling Salesman Problem (TSP) is known to be NP-hard, which means that there no known polynomial-time algorithm that can solve all instances of the problem. This computational complexity would imply that as the number of cities increases, the time required to find an exact solution grows exponentially.

Exact algorithms, such as Dynamic Programming (e.g., the Bellman-Held-Karp algorithm), are used to find precise solutions but they require exponential time. For instance, the Bellman-Held-Karp algorithm has a time complexity of $O(n^2 \times 2^n)$, where n represents the number of cities. This makes exact

algorithms impractical for large instances of the TSP because of how computationally expensive these viable classical algorithms are.

To address this, approximation algorithms have been developed to provide near-optimal solutions in a more efficient manner. One prominent example is Christofides' Algorithm, which guarantees a solution within 1.5 times the optimal solution. While these approximation algorithms do not always yield the exact shortest path, they significantly reduce the computational burden and provide solutions that are sufficiently close to the optimal for many practical purposes.

2.1 Classical Algorithms

Exact algorithms typically have an exponential time complexity. The Bellman-Held-Karp algorithm has a time complexity of $O(n^2 \cdot 2^n)$. Approximation algorithms, such as Christofides' algorithm, provide near-optimal solutions with a guarantee within 1.5 times the optimal solution.

2.2 Resources and Scalability

Classical algorithms, as we've explored require significant computational power and memory as the number of cities increases. Although this idea of parallel processing can be used but is still limited by classical computation constraints. Classical methods struggle with scalability due to the factorial growth in complexity.

3 Quantum Approach to TSP

The Quantum Approximate Optimization Algorithm (QAOA) is designed to solve combinatorial optimization problems by combining quantum and classical techniques. The key idea here is to encode the problem into a quantum state using cost and mixer Hamiltonians and iteratively optimize parameters to find an approximate solution efficiently.

3.1 Time Complexity

Quantum algorithms can offer potential polynomial speedup. QAOA leverages quantum parallelism and entanglement to explore multiple solutions simultaneously. This exciting optimization has such great potential in generalized forms of TSP and can be applied to a wide variety of contexts in logistics, finance, and in combinatorial mathematics.

3.2 Resources and Scalability

Quantum computers use qubits, which as we know can represent multiple states simultaneously. These Q-Computers which require fewer qubits compared to

classical bits for large search spaces are strong candidates to help us solve optimization problems. In it's current state, the limitations are due to qubit coherence times and error rates, but advancements in quantum hardware are promising. Quantum algorithms show better scalability for **certain optimization problems,** though they are still in the experimental stage.

4 The Traveling Salesman Problem

The TSP is one of the most studied problems in combinatorial optimization. It asks for the shortest possible path that visits each city at least once and returns to the original city. This problem is important because it belongs to NP-complete problems. If an efficient solution is found, then all problems in the NP-complete class would be solved.

4.1 Computational Complexity

The TSP's computational complexity is NP-hard, meaning there is no known polynomial-time algorithm to solve all instances. Exact algorithms require exponential time, such as Dynamic Programming (Bellman-Held-Karp): $O(n^2 \cdot 2^n)$. Approximation algorithms provide near-optimal solutions, like Christofides' Algorithm, which guarantees within 1.5 times the optimal solution.

5 Quantum States

This section looks to affirm the fundamental building blocks to arrive at quantum algorithms built in this class. Qubits are the basic unit of quantum information and can be in a superposition of 0 and 1 and in the context of TSP, represent the problem's solution space. A superposition allows a qubit to represent both 0 and 1 simultaneously. In the context of our TSP, superposition properties allow for the simultaneous exploration of multiple solutions. Entanglement means the state of one qubit can depend on the state of another which in essence captures the dependencies between different parts of the problem. Quantum Gates, such as Pauli-X, Hadamard, and CNOT, are operations that change the state of qubits and as a result manifest as the necessary operations for evolving the quantum state to find optimal/near-optimal solutions for TSP.

6 Quantum Algorithms

Quantum algorithms look to be faster than classical counterparts by using quantum mechanical concepts such as superposition and entanglement. This efficiency can manifest in the following forms: they might use fewer gates and have efficient polynomial (linear) circuit complexity. Famous quantum algorithms include Grover's Algorithm, which provides a quadratic speedup for searching an unsorted database, and Shor's Algorithm that efficiently factors large integers.

7 Quantum Optimization Algorithms

Quantum optimization algorithms combine mathematics, computer science, quantum computing, and physics to solve complex problems. The Quantum Approximate Optimization Algorithm (QAOA) is used to find approximate solutions to combinatorial optimization problems by combining quantum and classical techniques.

8 History of QAOA

The Quantum Approximate Optimization Algorithm (QAOA) was introduced by Edward Farhi, Jeffrey Goldstone, and Sam Gutmann in 2014 as a quantum algorithm designed to solve combinatorial optimization problems. QAOA uses principles from quantum mechanics like thes previously introduced concepts of superposition and entanglement, to provide approximate solutions very efficiently.

NOTE: The distance assignments for pairs of nodes were intentionally chosen to be easy to follow (hence the chronological TSP tour and simplified bit strings)

8.1 Stepping Through QAOA

- **Problem Setup:** Optimize an objective function that represents the cost.
- **Objective Function $C(x)$:** A function that assigns a cost to each possible solution x.
- **Binary String x:** Solutions are represented as binary strings of length n.
- **Hamiltonians:** Cost Hamiltonian H_C encodes the objective function, and Mixer Hamiltonian H_M helps explore different solutions.

8.1.1 Initialization

- **Quantum State $|\psi_0\rangle$:** The initial quantum state, a superposition of all possible binary strings of length n.

8.1.2 Cost and Mixer Unitaries

- **Unitary Operators:** Describe the evolution of quantum states.
- **Cost Unitary $U(C, \gamma)$:** Applies the cost Hamiltonian to the quantum state.
- **Mixer Unitary $U(B, \beta)$:** Applies the mixer Hamiltonian to the quantum state.

8.1.3 Forming the Quantum State

Alternating applications of cost and mixer unitaries form the quantum state.

8.1.4 Measurement and Optimization

- **Measure the Final State $|\psi(\gamma, \beta)\rangle$:** Collapse the quantum state to obtain a classical outcome.
- **Expectation Value $\langle C\rangle$:** Represents the average value of the cost function.
- **Classical Optimization:** Adjust parameters γ and β to maximize $\langle C\rangle$.

8.1.5 Algorithm at a High Level

- Initialize: Prepare the superposition state.
- Alternating Unitaries: Apply p layers of cost and mixer unitaries.
- Measure: Measure the final state.
- Optimize: Optimize parameters using classical algorithms.
- Iterate: Repeat until optimal parameters are found.
- Output: Measure the optimized state to obtain the solution.

9 Demonstration of TSP Solving with QAOA

You may find our implementation in our Google Collab Python Script: Colab Notebook

9.1 Problem Formulation: Phase 1

- **Cities and Distances:**
- Cities: [0, 1, 2, 3]
- Distances: $\{(0,1):1,(0,2):4,(0,3):6,(1,2):2,(1,3):5,(2,3):3\}$

Results:

- Most common bitstring: 1111
- TSP Tour: [0, 1, 2, 3]

The most common bitstring line gives us the binary string that most frequently represents the result of running the quantum algorithm. The bitstring is then mapped to a tour sequence based on our implemented logic.

9.2 Problem Formulation: Phase 2

- **Cities and Distances:**
- Cities: list(range(8))
- Distances: $\{(0,1) : 2, (0,2) : 9, (0,3) : 10, (0,4) : 7, (0,5) : 3, (0,6) : 1, (0,7) : 6, (1,2) : 6, (1,3) : 4, (1,4) : 3, (1,5) : 8, (1,6) : 5, (1,7) : 2, (2,3) : 1, (2,4) : 8, (2,5) : 4, (2,6) : 7, (2,7) : 3, (3,4) : 3, (3,5) : 6, (3,6) : 2, (3,7) : 5, (4,5) : 1, (4,6) : 6, (4,7) : 4, (5,6) : 2, (5,7) : 3, (6,7) : 8\}$

Results:

- Most common bitstring: 11111111
- TSP Tour: [0, 1, 2, 3, 4, 5, 6, 7]

9.3 Problem Formulation: Phase 3

Define a TSP problem with 15 cities and generate random distances for the TSP problem.

Results:

- Most common bitstring: 111111111111111
- TSP Tour: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

10 Future Directions in Quantum Computing

Advancements in quantum hardware are increasing qubit counts and reducing error rates which can bring us closer to quantum supremacy—achieving tasks that classical computers cannot solve in a reasonable time. Applications of quantum computing are expanding into fields such as cryptography, material science, logistics optimization, economics, finance, and a wide variety of other optimization problems

Some other Specific Applications: Analysis of crystal structures Overhauling of gas turbine engines In material handling in a warehouse Cutting stock problems Clustering of data arrays Assigning plane fleet routes Genome sequencing

11 Replication of the Coding Process

12 Implementation of TSP Solving with QAOA

In this section, we describe the process of implementing the Traveling Salesman Problem (TSP) solution using the Quantum Approximate Optimization Algorithm (QAOA) with the Google's Cirq library. The implementation is generalized for three different cases with varying numbers of cities and distances (n=4, n=8, and n=15).

12.1 Setup and Visualization

First, we define the TSP problem for each case with different sets of distances between cities. We create a graph to represent the TSP problem using the NetworkX library and visualize the initial graph with the specified distances.

```
# Import necessary libraries
import cirq
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from scipy.optimize import minimize

# Define the TSP problem for different cases
cases = [
    {
        "cities": [0, 1, 2, 3],
        "distances": {(0, 1): 10, (0, 2): 15, (0, 3): 20,
                      (1, 2): 35, (1, 3): 25,
                      (2, 3): 30},
        "title": "Initial Graph of TSP Problem with 4 Cities and New Distances"
    },
    {
        "cities": list(range(8)),
        "distances": {(0, 1): 2, (0, 2): 9, (0, 3): 10, (0, 4): 7, (0, 5): 3, (0, 6): 1, (0, 7): 6,
                      (1, 2): 6, (1, 3): 4, (1, 4): 3, (1, 5): 8, (1, 6): 5, (1, 7): 2,
                      (2, 3): 1, (2, 4): 8, (2, 5): 4, (2, 6): 7, (2, 7): 3,
                      (3, 4): 3, (3, 5): 6, (3, 6): 2, (3, 7): 5,
                      (4, 5): 1, (4, 6): 6, (4, 7): 4,
                      (5, 6): 2, (5, 7): 3,
                      (6, 7): 8},
        "title": "Initial Graph of TSP Problem with 8 Cities and New Distances"
    },
    {
        "num_cities": 15,
        "distances": {(i, j): np.random.randint(1, 100) for i in range(15) for j in range(i+1, 15)},
        "title": "Initial Graph of TSP Problem with 15 Cities and New Distances"
    }
]

for case in cases:
    cities = case["cities"]
    distances = case["distances"]

    # Create a graph for the TSP problem
    G = nx.Graph()
    for (i, j), dist in distances.items():
        G.add_edge(i, j, weight=dist)
```

```
    # Visualize the initial graph with distances
    pos = nx.spring_layout(G)
    plt.figure(figsize=(12, 12))
    nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=500, edge_color='gray')
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
    plt.title(case["title"])
    plt.show()
```

12.2 Creating the Hamiltonians

Next, we define the cost Hamiltonian and the mixer Hamiltonian for the QAOA. The purpose of the cost Hamiltonian is to encode the distances between cities, and the mixer Hamiltonian helps in exploring different solutions.

```
# Define a function to create a cost Hamiltonian
def create_cost_hamiltonian(graph):
    qubits = cirq.LineQubit.range(len(graph.nodes))
    cost_hamiltonian = sum(graph[i][j]['weight'] * (1 - cirq.Z(qubits[i]) * cirq.Z(qubits[j])) for i,
    return cost_hamiltonian, qubits

# Create the cost Hamiltonian
cost_hamiltonian, qubits = create_cost_hamiltonian(G)

# Define the mixer Hamiltonian
def create_mixer_hamiltonian(qubits):
    mixer_hamiltonian = sum(cirq.X(qubit) for qubit in qubits)
    return mixer_hamiltonian

mixer_hamiltonian = create_mixer_hamiltonian(qubits)
```

12.3 QAOA Circuit Creation

We then define a function to create the QAOA circuit, which alternates between applying the cost Hamiltonian and the mixer Hamiltonian for a specified number of layers p.

```
# Define a function to create the QAOA circuit
def create_qaoa_circuit(cost_hamiltonian, mixer_hamiltonian, qubits, gamma, beta, p):
    circuit = cirq.Circuit()
    for i in range(p):
        circuit.append(cirq.ZZPowGate(exponent=-gamma[i])(qubits[j], qubits[j+1]) for j in range(len(q
        circuit.append(cirq.X(qubit) ** beta[i] for qubit in qubits)
    # Add measurement operations
    circuit.append(cirq.measure(*qubits, key='result'))
    return circuit

# Correct the number of elements in gamma and beta
p = 1
gamma = [np.pi / 4] * p
```

```
beta = [np.pi / 4] * p

# Create the QAOA circuit
qaoa_circuit = create_qaoa_circuit(cost_hamiltonian, mixer_hamiltonian, qubits, gamma, beta, p)
```

12.4 Simulation and Analysis

After creating the QAOA circuit, we simulate it using Cirq's simulator and analyze the results to find the most common bitstring, which represents the TSP tour.

```
# Simulate the circuit
simulator = cirq.Simulator()
result = simulator.run(qaoa_circuit, repetitions=1000)

# Analyze the results
def get_most_common_bitstring(result):
    hist = result.histogram(key='result')
    most_common = max(hist, key=hist.get)
    # Convert integer to bitstring
    bitstring = format(most_common, f'0{len(qubits)}b')
    return bitstring

most_common_bitstring = get_most_common_bitstring(result)

def bitstring_to_tour(bitstring):
    tour = []
    for i in range(len(bitstring)):
        if bitstring[i] == '1':
            tour.append(i)
    return tour

tsp_tour = bitstring_to_tour(most_common_bitstring)
```

12.5 Visualization of the TSP and Solutions (in Reference Section

Finally, we visualize the TSP solution by highlighting the optimal tour on the initial graph for each case.

```
# Visualize the TSP solution
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=500, edge_color='gray')
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

# Highlight the optimal tour
if len(tsp_tour) > 1:
    tour_edges = [(tsp_tour[i], tsp_tour[i+1])
    for i in range(len(tsp_tour)-1)] + [(tsp_tour[-1], tsp_tour[0])]
```

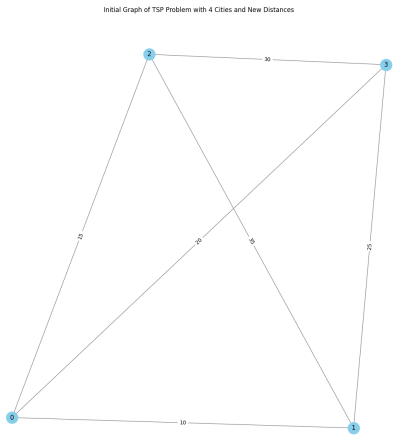

Figure 1: Drawing of Call Stack

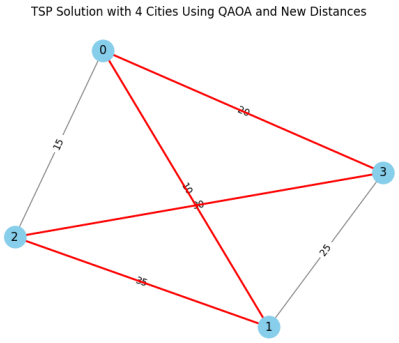

Figure 2: Drawing of Call Stack

```
    nx.draw_networkx_edges(G, pos, edgelist=tour_edges, edge_color='r', width=2)

plt.show()
```

This generalized implementation demonstrates how quantum computing, specifically QAOA, can be applied to solve the Traveling Salesman Problem for various problem sizes, showcasing the potential advantages of quantum algorithms in combinatorial optimization problems.

13 References

- Goldreich, O. (2008). Computational complexity: a conceptual perspective. ACM Sigact News, 39(3), 35-39.

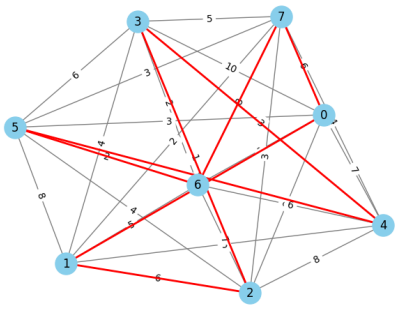

Figure 3: Drawing of Call Stack

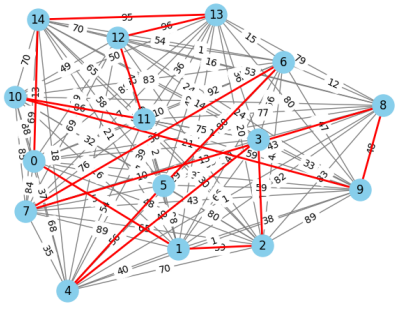

Figure 4: Drawing of Call Stack

- Jain, S. (2021). Solving the traveling salesman problem on the d-wave quantum computer. Frontiers in Physics, 9, 760783.
- Salehi, Ö., Glos, A., & Miszczak, J. A. (2022). Unconstrained binary models of the travelling salesman problem variants for quantum optimization. Quantum Information Processing, 21(2), 67.
- Wong, T. G. (2022). Introduction to Classical and Quantum Computing. Rooted Grove.
- Hoffman, K. L., Padberg, M., & Rinaldi, G. (2013). Traveling salesman problem. Encyclopedia of operations research and management science, 1, 1573-1578.
- Fakhimi, R., & Validi, H. (2023). Quantum Approximate Optimization Algorithm (QAOA).
- Stackpole, Beth (2024). Quantum Computing: What leaders need to know now.
- UC Berkeley (2024). Quantum algorithms.
- Arxiv (2023). Quantum Optimization: Potential, Challenges, and the Path Forward.
- OpenAI. (2024). Conversation with an AI assistant on solving the Traveling Salesman Problem using classical and quantum computing approaches. OpenAI ChatGPT.
- `https://quantumai.google/cirq/start/start`
- `https://github.com/aubreycoffey/TSP-Quantum-Computing/blob/main/quantum_algorithms/objective.py`

This comprehensive write-up delves into the fundamental concepts of solving the Traveling Salesman Problem using both classical and quantum computing approaches, highlighting the potential of quantum algorithms to revolutionize optimization problems.