SKKU 2025 Quantum Challenge : Traveling Salesman Problem

Sangyoon Woo

November 3, 2025

Part 1: Leg-variable encoding of the TSP

1-a) Suppose the distances between cities are: $(d_{ab}, d_{ac}, d_{ad}, d_{bc}, d_{bd}, d_{cd}) = (1, 1, 1, 2, 3, 1)$**. What is the smallest integer value of** λ **such that the** Obj **values for all valid tours are strictly less than the** Obj **values for all tours?**

This problem aims to find the smallest integer value of λ such that the Obj values calculated for all valid tours are always smaller than those for any invalid tours.

For valid tours, all constraint conditions are satisfied, so $P = 0$. Therefore, the objective function simplifies to $Obj_{\text{valid}} = D$.

In contrast, for invalid configurations, $P > 0$, and the objective function is expressed as $Obj_{\text{invalid}} = D + \lambda P$. Since P is always positive, if λ is sufficiently large, the Obj value for invalid tours becomes greater than that of valid tours.

Before solving the problem, note that the given distances are $(d_{ab}, d_{ac}, d_{ad}, d_{bc}, d_{bd}, d_{cd}) = (1, 1, 1, 2, 3, 1)$, these can be represented graphically as shown in Fig.1, and the distinct valid tours expressed using the Leg-variable encoding and their corresponding total distances are summarized in Table 1.

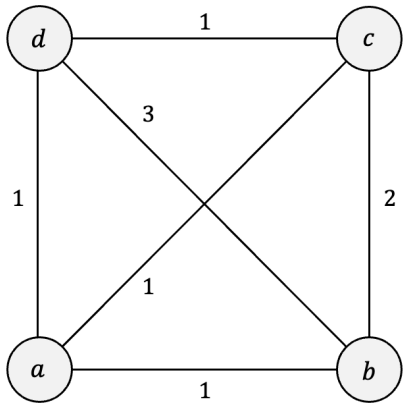

Figure 1: Graph representation of the given distance configuration in the 4-city TSP problem.

Table 1: Leg-variable encoding for three distinct tours in a 4-city TSP.

LEG	VAR	Tour 1	Tour 2	Tour 3	Distance
AB	x_{ab}	1	1	0	1
AC	x_{ac}	1	0	1	1
AD	x_{ad}	0	1	1	1
BC	x_{bc}	0	1	1	2
BD	x_{bd}	1	0	1	3
CD	x_{cd}	1	1	0	1
Sum of distance		**6**	**5**	**7**	

In this case, the distance of Tour 3 is the largest with $D = 7$. Therefore, since the maximum value of Obj_{valid} is 7, we must determine λ so that all Obj_{invalid} values become greater than 7.

The penalty term is defined as $P = (x_{ab} + x_{ac} + x_{ad} - 2)^2 + (x_{ab} + x_{bc} + x_{bd} - 2)^2 + (x_{ac} + x_{bc} + x_{cd} - 2)^2 + (x_{ad} + x_{bd} + x_{cd} - 2)^2$. When $P > 0$, the configuration represents an invalid case.

Invalid cases refer to all combinations that can be formed with the six variables (x_{ij}), excluding the three valid tours described above. Since there are $2^6 = 64$ possible combinations, it is theoretically possible to calculate Obj for every case. However, this approach becomes computationally inefficient as the number of cities increases. Therefore, in this report, we apply a simplified filtering criterion to select a subset of relevant invalid configurations.

Among the invalid cases $(P > 0)$, we filter configurations whose $D + P$ values are smaller than the maximum valid distance of 7. By determining λ such that all filtered cases satisfy this condition, we can ensure that valid tours always yield a smaller objective value. The filtering procedure was implemented as shown in Listing 1, and the resulting representative cases are summarized in Output 1.

Listing 1: Code to screen for specific cases when they are invalid.

```
d = [1, 1, 1, 2, 3, 1]

city_legs = {
    'A': [0, 1, 2],
    'B': [0, 3, 4],
    'C': [1, 3, 5],
    'D': [2, 4, 5]
}

for bits in itertools.product([0, 1], repeat=6):
    x = list(bits)
    D = sum(d[i] * x[i] for i in range(6))
    P = sum((sum(x[i] for i in city_legs[c]) - 2)**2 for c in
        city_legs)
    if P>0 and D+P <= 7:
        print(f'x={x}, D={D}, P={P}, D+P={D+P}')
```

Output 1: Selected invalid case.

```
x=[0, 0, 1, 1, 0, 0], D=3, P=4, D+P=7
x=[0, 0, 1, 1, 0, 1], D=4, P=2, D+P=6
x=[0, 1, 0, 0, 1, 1], D=5, P=2, D+P=7
x=[0, 1, 1, 0, 0, 1], D=3, P=4, D+P=7
x=[0, 1, 1, 0, 1, 0], D=5, P=2, D+P=7
x=[0, 1, 1, 1, 0, 0], D=4, P=2, D+P=6
x=[0, 1, 1, 1, 0, 1], D=5, P=2, D+P=7
x=[1, 0, 0, 0, 0, 1], D=2, P=4, D+P=6
x=[1, 0, 0, 0, 1, 1], D=5, P=2, D+P=7
x=[1, 0, 0, 1, 0, 1], D=4, P=2, D+P=6
x=[1, 0, 1, 0, 0, 1], D=3, P=2, D+P=5
x=[1, 0, 1, 1, 0, 0], D=4, P=2, D+P=6
x=[1, 1, 0, 0, 0, 1], D=3, P=2, D+P=5
x=[1, 1, 0, 0, 1, 0], D=5, P=2, D+P=7
x=[1, 1, 0, 1, 0, 1], D=5, P=2, D+P=7
x=[1, 1, 1, 0, 0, 0], D=3, P=4, D+P=7
x=[1, 1, 1, 0, 0, 1], D=4, P=2, D+P=6
x=[1, 1, 1, 1, 0, 0], D=5, P=2, D+P=7
```

Here, when $\lambda = 1$, as shown in Output 1, the objective values for invalid cases are smaller, and thus the condition is not satisfied. Next, let us examine the case of $\lambda = 2$. For example, when $x = [1, 0, 1, 0, 0, 1]$ and $x = [1, 1, 0, 0, 0, 1]$, the corresponding Obj value is 7, which is equal to the maximum Obj_{valid}. Therefore, $\lambda = 2$ is not sufficient.

In other words, when $\lambda > 2$, all Obj_{invalid} values become strictly greater than 7. Hence, the smallest integer value of λ that distinguishes valid tours from invalid ones is $\lambda = 3$.

The code from Listing 1 was slightly modified to count the number of cases satisfying $Obj \leq 7$ when $\lambda = 3$. This modified code is shown in Listing 2, and the output results are confirmed to match the cases presented in Table 1.

Listing 2: Code for counting the number of cases satisfying $Obj \leq 7$ when $\lambda = 3$.

```
obj_lambda = 3

for bits in itertools.product([0, 1], repeat=6):
    x = list(bits)
    D = sum(d[i] * x[i] for i in range(6))
    P = sum((sum(x[i] for i in city_legs[c]) - 2)**2 for c in
        city_legs)
    if D+obj_lambda*P <= 7 :
        print(f'x={x}, D={D}, P={P}, D+P={D+P}')
```

Output 2: Results for $\lambda = 3$.

```
x=[0, 1, 1, 1, 1, 0] → D=7, P=0, D+P=7
x=[1, 0, 1, 1, 0, 1] → D=5, P=0, D+P=5
x=[1, 1, 0, 0, 1, 1] → D=6, P=0, D+P=6
```

Part 2: Switch-network encoding of the TSP

2-a) Write out the objective D as a function of the six switch variables $x_0, x_1, x_2, x_3, x_4, x_5$. (Hint: rather than keeping track of all the variables manually, use a symbolic processing package like Sympy). Remember that Boolean variables are idempotent, meaning that they equal their own square: $x^2 = x$. Apply this equation for each switch variable to make sure that none of the switch variables appears squared or to any power higher than one.

Based on the switch network structure provided in the problem (see Fig. 2), all possible cases were expressed as shown in Table 2. Each element of the table indicates the switches that a given path passes through in the network. A switch in the *on* state is denoted by its index number, while a switch in the *off* state is represented by $\bar{i}$.

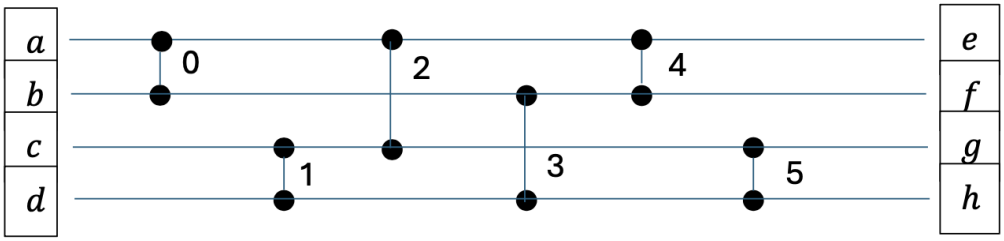

Figure 2: The switch network structure presented in the problem.

Table 2: Switch configurations for all input–output paths. Each entry shows the switches along the corresponding route.

From/To	e	f	g	h
a	$[\bar{0}, \bar{2}, \bar{4}]$, [0, $\bar{2}$, 4]	[0, $\bar{3}$, $\bar{4}$], $[\bar{0}, \bar{2}, 4]$	$[\bar{0}$, 2, $\bar{5}]$, [0, 3, 5]	[0, 3, $\bar{5}$], $[\bar{0}$, 2, 5]
b	[0, $\bar{2}$, $\bar{4}$], $[\bar{0}$, $\bar{3}$, 4]	$[\bar{0}$, $\bar{3}$, $\bar{4}]$, [0, $\bar{2}$, 4]	[0, 2, $\bar{5}$], $[\bar{0}$, 3, 5]	[0, 2, 5], $[\bar{0}$, 3, $\bar{5}]$
c	$[\bar{1}$, 2, $\bar{4}]$, [1, 3, 4]	[1, 3, $\bar{4}$], $[\bar{1}$, 2, 4]	$[\bar{1}$, $\bar{2}$, $\bar{5}]$, [1, $\bar{3}$, 5]	[1, $\bar{3}$, $\bar{5}$], $[\bar{1}$, $\bar{3}$, 5]
d	[1, 2, $\bar{4}$], $[\bar{1}$, 3, 4]	[1, 2, 4], $[\bar{1}$, 3, $\bar{4}]$	[1, $\bar{2}$, $\bar{5}$], $[\bar{1}$, $\bar{3}$, 5]	$[\bar{1}$, $\bar{3}$, $\bar{5}]$, [1, $\bar{3}$, 5]

Using this switch information, the τ_{ij} functions can be constructed. When a switch i is turned on, the variable x_i is used, and when the switch is off, $(1 - x_i)$ is used. A Python code that computes the corresponding τ_{ij} values was implemented based on this principle, as shown in Listing 3. Executing the code yields all possible indicator functions, as shown in Eq. 1a-1p.

Listing 3: Computation of the indicator functions τ_{ij} for all input–output pairs based on Table 2.

```
x = sp.symbols('x0 x1 x2 x3 x4 x5')

tau_table = {
    'a': {
        'e': [['!0', '!2', '!4'], ['0', '!3', '4']], #
```

```
        'f': [['0', '!3', '!4'], ['!0', '!2', '4']], #
        'g': [['!0', '2', '!5'], ['0', '3', '5']], #
        'h': [['0', '3', '!5'], ['!0', '2', '5']] #
    },
    'b': {
        'e': [['0', '!2', '!4'], ['!0', '!3', '4']],
        'f': [['!0', '!3', '!4'], ['0', '!2', '4']],
        'g': [['0', '2', '!5'], ['!0', '3', '5']],
        'h': [['0', '2', '5'], ['!0', '3', '!5']]
    },
    'c': {
        'e': [['!1', '2', '!4'], ['1', '3', '4']],
        'f': [['1', '3', '!4'], ['!1', '2', '4']],
        'g': [['!1', '!2', '!5'], ['1', '!3', '5']],
        'h': [['1', '!3', '!5'], ['!1', '!2', '5']]
    },
    'd': {
        'e': [['1', '2', '!4'], ['!1', '3', '4']],
        'f': [['1', '2', '4'], ['!1', '3', '!4']],
        'g': [['1', '!2', '!5'], ['!1', '!3', '5']],
        'h': [['!1', '!3', '!5'], ['1', '!2', '5']]
    }
}

def term_to_expr(term):
    factors = []
    for t in term:
        if '!' in t:
            idx = int(t.replace('!', ''))
            factors.append(1 - x[idx])
        else:
            factors.append(x[int(t)])
    return sp.Mul(*factors, evaluate=False)

def tau_expr(from_city, to_city):
    term_list = tau_table[from_city][to_city]
    exprs = [term_to_expr(term) for term in term_list]
    return sp.Add(*exprs, evaluate=False)

for from_city in ['a', 'b', 'c', 'd']:
    for to_city in ['e', 'f', 'g', 'h']:
        tau_ij = tau_expr(from_city, to_city)
        print(f'tau_{from_city}{to_city} = {tau_ij}')
```

$$\tau_{ae} = x_0x_4(1-x_3) + (1-x_0)(1-x_2)(1-x_4) \tag{1a}$$
$$\tau_{af} = x_0(1-x_3)(1-x_4) + x_4(1-x_0)(1-x_2) \tag{1b}$$
$$\tau_{ag} = x_0x_3x_5 + x_2(1-x_0)(1-x_5) \tag{1c}$$
$$\tau_{ah} = x_0x_3(1-x_5) + x_2x_5(1-x_0) \tag{1d}$$
$$\tau_{be} = x_0(1-x_2)(1-x_4) + x_4(1-x_0)(1-x_3) \tag{1e}$$
$$\tau_{bf} = x_0x_4(1-x_2) + (1-x_0)(1-x_3)(1-x_4) \tag{1f}$$
$$\tau_{bg} = x_0x_2(1-x_5) + x_3x_5(1-x_0) \tag{1g}$$
$$\tau_{bh} = x_0x_2x_5 + x_3(1-x_0)(1-x_5) \tag{1h}$$
$$\tau_{ce} = x_1x_3x_4 + x_2(1-x_1)(1-x_4) \tag{1i}$$
$$\tau_{cf} = x_1x_3(1-x_4) + x_2x_4(1-x_1) \tag{1j}$$
$$\tau_{cg} = x_1x_5(1-x_3) + (1-x_1)(1-x_2)(1-x_5) \tag{1k}$$
$$\tau_{ch} = x_1(1-x_3)(1-x_5) + x_5(1-x_1)(1-x_2) \tag{1l}$$
$$\tau_{de} = x_1x_2(1-x_4) + x_3x_4(1-x_1) \tag{1m}$$
$$\tau_{df} = x_1x_2x_4 + x_3(1-x_1)(1-x_4) \tag{1n}$$
$$\tau_{dg} = x_1(1-x_2)(1-x_5) + x_5(1-x_1)(1-x_3) \tag{1o}$$
$$\tau_{dh} = x_1x_5(1-x_2) + (1-x_1)(1-x_3)(1-x_5) \tag{1p}$$

The total distance D follows Eq. 2. This represents the complete expression obtained by summing, for every pair of cities that become adjacent in the network, the products of their corresponding indicator functions multiplied by the distance between those two cities:

$$D = \sum_{i<j} d_{ij}(\tau_{ie}\tau_{jf} + \tau_{if}\tau_{je} + \tau_{if}\tau_{jg} + \tau_{ig}\tau_{jf} + \tau_{ig}\tau_{jh} + \tau_{ih}\tau_{jg} + \tau_{ih}\tau_{je} + \tau_{ie}\tau_{jh}) \tag{2}$$

Since each τ_{ij} is a cubic function, the terms in D that involve products of two τ_{ij}'s become sixth-order polynomials. Because the number of such terms is extremely large and complex, it is impractical to compute D manually. Therefore, the process of expanding and evaluating D is carried out programmatically. The code for computing D is shown in Listing 4, and the expanded expression of D is given in Eq. 3.

Listing 4: Code for computing the distance function D based on the calculated τ_{ij} values.

```
city_pairs = [('a','b'), ('a','c'), ('a','d'), ('b','c'), ('b'
    ,'d'), ('c','d')]
adjacent_ports = [('e','f'), ('f','g'), ('g','h'), ('h','e')]

distance_values = {
    ('a','b'): 1,
    ('a','c'): 1,
    ('a','d'): 1,
    ('b','c'): 2,
    ('b','d'): 3,
    ('c','d'): 1
```

```
}

tau_dict = {f'{i}{j}': tau_expr(i,j) for i in ['a','b','c','d'] for j in ['e','f','g','h']}

d_terms = {}
for (city1, city2) in city_pairs:
    terms = []
    for (p, q) in adjacent_ports:
        terms.append(tau_dict[f'{city1}{p}'] * tau_dict[f'{city2}{q}'])
        terms.append(tau_dict[f'{city2}{p}'] * tau_dict[f'{city1}{q}'])
    d_expr = sp.Add(*terms, evaluate=False)
    d_terms[f'D_{city1}{city2}'] = d_expr

weighted_terms = []
for (city1, city2), dist in distance_values.items():
    weighted_terms.append(dist * d_terms[f'D_{city1}{city2}'])

D = sp.Add(*weighted_terms, evaluate=False)
D_expanded = sp.expand(D)
```

$$
\begin{aligned}
D(x) =& 2x_0^2x_2^2x_4^2 - 4x_0^2x_2^2x_4x_5 + 2x_0^2x_2^2x_5^2 + 4x_0^2x_2x_3x_4^2 - 8x_0^2x_2x_3x_4x_5 + 4x_0^2x_2x_3x_5^2 \\
&- 8x_0^2x_2x_4^2 + 8x_0^2x_2x_4x_5 + 4x_0^2x_2x_4 - 4x_0^2x_2x_5 + 2x_0^2x_3^2x_4^2 - 4x_0^2x_3^2x_4x_5 \\
&+ 2x_0^2x_3^2x_5^2 - 8x_0^2x_3x_4^2 + 8x_0^2x_3x_4x_5 + 4x_0^2x_3x_4 - 4x_0^2x_3x_5 + 8x_0^2x_4^2 \\
&- 8x_0^2x_4 + 2x_0^2 + 2x_0x_1x_2^2x_4^2 - 4x_0x_1x_2^2x_4x_5 + 2x_0x_1x_2^2x_5^2 + 4x_0x_1x_2x_3x_4^2 \\
&- 8x_0x_1x_2x_3x_4x_5 + 4x_0x_1x_2x_3x_5^2 - 4x_0x_1x_2x_4^2 + 8x_0x_1x_2x_4x_5 - 4x_0x_1x_2x_5^2 + 2x_0x_1x_3^2x_4^2 \\
&- 4x_0x_1x_3^2x_4x_5 + 2x_0x_1x_3^2x_5^2 - 4x_0x_1x_3x_4^2 + 8x_0x_1x_3x_4x_5 - 4x_0x_1x_3x_5^2 - 8x_0x_1x_4x_5 \\
&+ 4x_0x_1x_4 + 4x_0x_1x_5 - 2x_0x_1 - 6x_0x_2x_3x_4^2 + 12x_0x_2x_3x_4x_5 - 6x_0x_2x_3x_5^2 \\
&+ 4x_0x_2x_4^2 - 6x_0x_2x_4x_5 - x_0x_2x_4 + 2x_0x_2x_5^2 + x_0x_2x_5 - 6x_0x_3^2x_4^2 \\
&+ 12x_0x_3^2x_4x_5 - 6x_0x_3^2x_5^2 + 16x_0x_3x_4^2 - 18x_0x_3x_4x_5 - 7x_0x_3x_4 + 2x_0x_3x_5^2 \\
&+ 7x_0x_3x_5 - 8x_0x_4^2 + 4x_0x_4x_5 + 6x_0x_4 - 2x_0x_5 - x_0 \\
&+ 2x_1^2x_2^2x_4^2 - 4x_1^2x_2^2x_4x_5 + 2x_1^2x_2^2x_5^2 + 4x_1^2x_2x_3x_4^2 - 8x_1^2x_2x_3x_4x_5 + 4x_1^2x_2x_3x_5^2 \\
&+ 8x_1^2x_2x_4x_5 - 4x_1^2x_2x_4 - 8x_1^2x_2x_5^2 + 4x_1^2x_2x_5 + 2x_1^2x_3^2x_4^2 - 4x_1^2x_3^2x_4x_5 \\
&+ 2x_1^2x_3^2x_5^2 + 8x_1^2x_3x_4x_5 - 4x_1^2x_3x_4 - 8x_1^2x_3x_5^2 + 4x_1^2x_3x_5 + 8x_1^2x_5^2 \\
&- 8x_1^2x_5 + 2x_1^2 - 2x_1x_2^2x_4^2 + 4x_1x_2^2x_4x_5 - 2x_1x_2^2x_5^2 - 6x_1x_2x_3x_4^2 \\
&+ 12x_1x_2x_3x_4x_5 - 6x_1x_2x_3x_5^2 + 2x_1x_2x_4^2 - 10x_1x_2x_4x_5 + 3x_1x_2x_4 + 8x_1x_2x_5^2 \\
&- 3x_1x_2x_5 - 4x_1x_3^2x_4^2 + 8x_1x_3^2x_4x_5 - 4x_1x_3^2x_5^2 + 2x_1x_3x_4^2 - 14x_1x_3x_4x_5 \\
&+ 5x_1x_3x_4 + 12x_1x_3x_5^2 - 5x_1x_3x_5 + 4x_1x_4x_5 - 2x_1x_4 - 8x_1x_5^2 \\
&+ 6x_1x_5 - x_1 + 2x_2^2x_4^2 - 4x_2^2x_4x_5 + 2x_2^2x_5^2 - 2x_2x_3x_4^2 \\
&+ 4x_2x_3x_4x_5 - 2x_2x_3x_5^2 + 2x_2x_4x_5 - x_2x_4 - 2x_2x_5^2 + x_2x_5 \\
&+ 6x_3^2x_4^2 - 12x_3^2x_4x_5 + 6x_3^2x_5^2 - 6x_3x_4^2 + 10x_3x_4x_5 + x_3x_4 \\
&- 4x_3x_5^2 - x_3x_5 + 2x_4^2 - 2x_4x_5 - x_4 + 2x_5^2 \\
&- x_5 + 5
\end{aligned}
\tag{3}
$$

However, this equation is exceedingly complex, allowing for simplification. Considering that each x_i takes only binary values of 0 or 1, we can see that $x_i^2 = x_i$ always holds true.

Therefore, since x_i^2 can be replaced with x_i, terms sharing the same monomial structure can be grouped together, allowing the expression to be simplified. The code that performs this substitution and simplification is shown in Listing 5, and the simplified equation is given as Eq. 4.

Listing 5: Code for obtaining the simplified distance function $D_{\text{simplified}}$ by substituting x_i^2 with x_i.

```
for xi in x:
    D_expanded = D_expanded.xreplace({xi**2: xi})
```

$$
\begin{aligned}
D_{simplized} = &-8x_0x_1x_2x_3x_4x_5 \\
&+4x_0x_1x_2x_3x_4 + 4x_0x_1x_2x_3x_5 + 4x_0x_1x_2x_4x_5 + 4x_0x_1x_3x_4x_5 - 2x_0x_2x_3x_4x_5 \\
&+4x_1x_2x_3x_4x_5 \\
&-2x_0x_1x_2x_4 - 2x_0x_1x_2x_5 - 2x_0x_1x_3x_4 - 2x_0x_1x_3x_5 - 8x_0x_1x_4x_5 - 2x_0x_2x_3x_4 \\
&-2x_0x_2x_3x_5 - 2x_0x_3x_4x_5 - 2x_1x_2x_3x_4 - 2x_1x_2x_3x_5 - 2x_1x_2x_4x_5 - 2x_1x_3x_4x_5 \\
&+4x_2x_3x_4x_5 \\
&+4x_0x_1x_4 + 4x_0x_1x_5 + x_0x_2x_4 + x_0x_2x_5 + x_0x_3x_4 + x_0x_3x_5 + 4x_0x_4x_5 \\
&+x_1x_2x_4 + x_1x_2x_5 + x_1x_3x_4 + x_1x_3x_5 + 4x_1x_4x_5 - 2x_2x_3x_4 - 2x_2x_3x_5 \\
&-2x_2x_4x_5 - 2x_3x_4x_5 \\
&-2x_0x_4 - 2x_0x_5 - 2x_1x_4 - 2x_1x_5 + x_2x_4 + x_2x_5 + x_3x_4 + x_3x_5 - 2x_4x_5 \\
&+x_1 + x_4 + x_5 \\
&+5
\end{aligned}
\tag{4}
$$

2-b) Evaluate the objective D for all 64 possible settings of the switch variables. How many times does each of the three possible tours appear?

By substituting all possible combinations of the six switch variables into the simplified equation, we can obtain the tour distance for every switch configuration. The code that performs the substitution of all switch combinations and records the resulting values is shown in Listing 6, and the corresponding output results are presented in Output 3.

Listing 6: Code for evaluating all switch combinations and counting the occurrences of each resulting distance.

```
combinations = list(itertools.product([0,1], repeat=6))

results = []
for combo in combinations:
    subs_dict = {x[i]: combo[i] for i in range(6)}
    D_val = D_expanded.subs(subs_dict)
    results.append((combo, float(D_val)))

```

```
results.sort(key=lambda r: r[1])

from collections import Counter
dist_counts = Counter([r[1] for r in results])

for dist, count in sorted(dist_counts.items()):
    print(f'D = {dist:>4} : {count:>2} times')

min_dist = min(dist_counts.keys())
opt_combos = [r for r in results if r[1] == min_dist]
```

Output 3: Results of all switch combinations substituted into D.

```
D =  5.0 : 24 times
D =  6.0 : 24 times
D =  7.0 : 16 times
```

The resulting distances correspond to the three distinct tours obtained in Part 1. Among them, the number of switch combinations that yield $D = 5$ or $D = 6$ is 24, while the number of combinations producing $D = 7$ is 16. This indicates that the network structure expresses the three tours almost uniformly, though it is not a perfectly symmetric configuration.

In this way, the TSP problem was solved using the switch-network encoding approach. Although this method involves more complex algebraic expressions and expansion processes than the leg-variable encoding used in Part 1, it has the advantage of eliminating unnecessary constraint terms. Furthermore, if the switch-network encoding is extended to a physical device, it suggests the possibility that the switches can be directly mapped onto physical qubit gates.

The detailed implementation of this concept will be discussed in the next part. The procedures carried out in this part provide a foundation for extending the TSP problem to quantum computation.

Part 3: Comparison of QAOA performance

Before getting into the problem, we must create the Quantum Approximate Optimization Algorithm (QAOA) circuit presented in the main text. The steps can be summarized as follows:

1. Build Quadratic Unconstrained Binary Optimization (QUBO) / Higher-order Unconstrained Binary Optimization (HUBO)
2. Spin-variable transformation
3. Quantum Hamiltonian mapping
4. QAOA circuit construction

Step 1

QUBO and HUBO refer to quadratic/higher-order objective functions composed of binary variables, respectively. This process was already calculated in the course of

deriving the Obj function for the Leg-variable encoding method and the D function for the Switch-network encoding method.

Step 2

Each variable x_i is replaced with a spin s_i. This is because using spin enables a physical interpretation. The relationship is as follows:

$$x_i = \frac{1 - s_i}{2} \tag{5}$$

Rearranging this for s_i:

$$s_i = 1 - 2x_i \tag{6}$$

Since $x_i = \{0, 1\}$, it follows that $s_i = \{-1, 1\}$. That is, the binary space is changed to the spin space.

To perform this procedure, the objective function Obj was implemented symbolically using Sympy, and the variable substitution was carried out accordingly. The corresponding code is shown in Listing 7, and the simplified expression after substitution is presented in Eq. 8. The distance information d between each pair of cities and λ were predefined and substituted using the given numerical values.

Listing 7: Code for transforming the objective function Obj of the leg-variable encoding into the spin space.

```
s = sp.symbols('s0 s1 s2 s3 s4 s5')

D = sum(d[i] * x[i] for i in range(6))

P_terms = []
for c in city_legs:
    expr = sum(x[i] for i in city_legs[c]) - 2
    P_terms.append(expr**2)
P = sum(P_terms)

L = 3
Obj = D + L * P
Obj_expanded = sp.expand(Obj)

sub_dict = {x[i]: (1 - s[i])/2 for i in range(6)}

for xi in x:
    Obj_expanded = Obj_expanded.xreplace({xi**2: xi})

Obj_spin = Obj_expanded.subs(sub_dict)
Obj_spin_expanded = sp.expand(Obj_spin)
```

$$
\begin{aligned}
Obj_{spin} =& \frac{3s_0s_1}{2} + \frac{3s_0s_2}{2} + \frac{3s_0s_3}{2} + \frac{3s_0s_4}{2} \\
&+ \frac{3s_1s_2}{2} + \frac{3s_1s_3}{2} + \frac{3s_1s_5}{2} \\
&+ \frac{3s_2s_4}{2} + \frac{3s_2s_5}{2} \\
&+ \frac{3s_3s_4}{2} + \frac{3s_3s_5}{2} \\
&+ \frac{3s_4s_5}{2} \\
&+ \frac{5s_0}{2} + \frac{5s_1}{2} + \frac{5s_2}{2} + 2s_3 + \frac{3s_4}{2} + \frac{5s_5}{2} \\
&+ \frac{33}{2}
\end{aligned}
\tag{7}
$$

Since D was previously derived symbolically using Sympy, the substitution can be directly applied to $D_{simplified}$. The corresponding implementation is presented in Listing 8, and the resulting expression after substitution is shown in Eq. 8.

Listing 8: Code for transforming the distance function D of the switch-network encoding into the spin space.

```
for xi in x:
    D_expanded = D_expanded.xreplace({xi**2: xi})

sub_dict = {x[i]: (1 - s[i])/2 for i in range(6)}
D_spin = D_expanded.subs(sub_dict)
D_spin_expanded = sp.expand(D_spin)
```

$$
\begin{aligned}
D_{spin} =& -\frac{s_0s_1s_2s_3s_4s_5}{8} + \frac{s_0s_1s_2s_3}{8} - \frac{3s_0s_1s_4s_5}{8} - \frac{s_0s_1}{8} \\
&+ \frac{3s_2s_3s_4s_5}{8} - \frac{3s_2s_3}{8} - \frac{3s_4s_5}{8} + \frac{47}{8}
\end{aligned}
\tag{8}
$$

Step 3

Step 3 is the process of converting the mathematical spin variables s_i into matrix operators that can be used in an actual quantum circuit. Each spin variable is replaced with the Pauli-Z operator $\sigma_z^{(i)}$. The Pauli-Z operator acts as a rotation operator about the z-axis for each qubit, and in quantum circuits, it plays the same role as the RZ gate. The matrix representation of σ_z is given in Eq. 9:

$$
\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{9}
$$

When two or more spins appear as a product term, each operator is represented as a tensor product such as $\sigma_z \otimes \sigma_z$, which denotes the interaction between the corresponding qubits. If any spin variable is missing from a term, the identity operator I is inserted at its position.

By applying these rules to Obj_{spin} and D_{spin}, the corresponding Hamiltonian H for each encoding method can be constructed. When representing the Hamiltonian in circuit form, only the qubits associated with a given term are assigned the

corresponding gates, and no operation is applied to qubits that do not appear in that term. Additionally, the constant part of H only affects the global phase of the circuit and can therefore be safely ignored.

Step 4

We can write an ansatz that guides the construction of a quantum circuit based on the Hamiltonian H derived in the previous process. In this step, we explicitly describe the structure of the QAOA circuit.

First, Hadamard gates are applied to all qubits in the circuit to create a uniform superposition of all computational basis states. Next, in the i-th layer, the exponentiated Hamiltonian parameterized by time γ_i is applied, followed by the mixer operator parameterized by time β_i. These two types of Hamiltonians are repeated for p layers. The final state resulting from this iterative process is denoted as $|\gamma, \beta\rangle$.

Before constructing the main circuit, it is necessary to examine how to implement rotations along the Z-axis involving multiple qubits. In Qiskit, the entanglement between two qubits can be implemented using the `rzz` function, but for interactions involving more than two qubits, a different approach is required.

The operator $e^{-i\gamma b_j^{(i)} \sigma_z^{(i)} \sigma_z^{(j)}}$, that is, the R_{ZZ} gate, can be implemented using CNOT and RZ gates. The two qubits to be entangled are first entangled through a CNOT gate, then rotated with an RZ gate. Afterward, another CNOT gate is applied to disentangle them, which has the same effect as applying an R_{ZZ} gate.

This method is not limited to two qubits; it can also be extended to multiple qubits. For example, in a 3-qubit system, a rotation involving all three qubits can be applied as shown in Fig. 3. Listing 9 performs this entangling procedure programmatically.

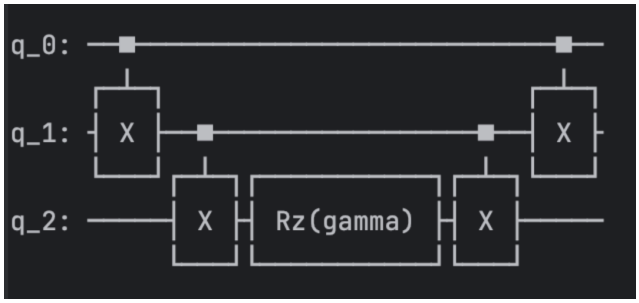

Figure 3: Method for implementing high-order multi-qubit operations using multiple CNOT and RZ gates.

Listing 9: Code for applying multi-qubit operations.

```
def multiple_rz(qc, qubits, coeff, gamma, isBarrier=False):
    for i in range(len(qubits) - 1):
        qc.cx(qubits[i], qubits[i+1])

    qc.rz(2 * gamma * coeff, qubits[-1])

    for i in reversed(range(len(qubits) - 1)):
        qc.cx(qubits[i], qubits[i+1])
```

```
9
10      if isBarrier:
11          qc.barrier()
```

3-a) Using the pairwise distances between cities, build a QAOA circuit for $p = 1$. How many CNOT gates does it contain?

In this problem, we are asked to determine the number of CNOT gates used when constructing the quantum circuit based on the leg-variable encoding. From Eq. 7, we can see that the maximum order of the entangled terms is 2, and there are a total of 12 such entangled terms. Furthermore, as examined earlier in Step 4, when N qubits are entangled, the number of CNOT gates required is $2N - 2$.

Therefore, the number of CNOT gates used in the leg-variable encoding scheme is calculated as $(2 \times 2 - 2) \times 12 = 24$.

Listing 10 presents the temporary quantum circuit for the case of $p = 1$. As shown in Fig. 4, both the visualized circuit and the computation confirm that a total of 24 CNOT gates are used. γ and β used a temporary value of 1 for circuit configuration.

Listing 10: CCode for generating the quantum circuit of the leg-variable encoding when $p = 1$.

```
n_qubits = 6
qc = QuantumCircuit(n_qubits)

gamma = 1
beta = 1

for q in range(n_qubits):
    qc.h(q)

qc.barrier()

qc.rz(2*gamma*5/2, 0)
qc.rz(2*gamma*5/2, 1)
qc.rz(2*gamma*5/2, 2)
qc.rz(2*gamma*2,   3)
qc.rz(2*gamma*3/2, 4)
qc.rz(2*gamma*5/2, 5)

multiple_rz(qc, [0,1], 3/2, gamma, isBarrier=True)
multiple_rz(qc, [0,2], 3/2, gamma, isBarrier=True)
multiple_rz(qc, [0,3], 3/2, gamma, isBarrier=True)
multiple_rz(qc, [0,4], 3/2, gamma, isBarrier=True)

multiple_rz(qc, [1, 2], 3/2, gamma, isBarrier=True)
multiple_rz(qc, [1, 3], 3/2, gamma, isBarrier=True)
multiple_rz(qc, [1, 5], 3/2, gamma, isBarrier=True)

multiple_rz(qc, [2, 4], 3/2, gamma, isBarrier=True)
multiple_rz(qc, [2, 5], 3/2, gamma, isBarrier=True)
```

```
30
31 multiple_rz(qc, [3, 4], 3/2, gamma, isBarrier=True)
32 multiple_rz(qc, [3, 5], 3/2, gamma, isBarrier=True)
33
34 multiple_rz(qc, [4, 5], 3/2, gamma, isBarrier=True)
35
36 for q in range(n_qubits):
37     qc.rx(2 * beta, q)
38
39 qc.measure_all()
40
41 qc.draw('mpl', fold=-1)
```

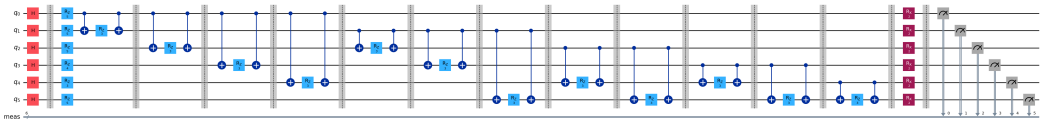

Figure 4: Quantum circuit of the leg-variable encoding when $p = 1$.

3-b) Using the switch network encoding described in Part 2, build a QAOA circuit for $p = 1$. How many CNOT gates does it contain?

The calculation for the switch network encoding method proceeds in a similar manner. Referring to Eq. 8, the number of CNOT gates can be computed as $(2 \times 6 - 2) + (2 \times 4 - 2) \times 3 + (2 \times 2 - 2) \times 3 = 34$. Likewise, the provisional circuit construction code can be found in Listing 11, and the visualized result of the circuit is shown in Fig. 5.

Listing 11: Code for generating the quantum circuit of the switch-network encoding when $p = 1$.

```
1  n_qubits = 6
2  qc2 = QuantumCircuit(n_qubits)
3
4  gamma = 1
5  beta = 1
6
7  for q in range(n_qubits):
8      qc2.h(q)
9
10 qc2.barrier()
11
12 multiple_rz(qc2, [0,1,2,3,4,5], -1/8, gamma, isBarrier=True)
13 multiple_rz(qc2, [0,1,2,3], 1/8, gamma, isBarrier=True)
14 multiple_rz(qc2, [0,1,4,5], -3/8, gamma, isBarrier=True)
15 multiple_rz(qc2, [0,1], -1/8, gamma, isBarrier=True)
16 multiple_rz(qc2, [2,3,4,5], 3/8, gamma, isBarrier=True)
17 multiple_rz(qc2, [2,3], -3/8, gamma, isBarrier=True)
18 multiple_rz(qc2, [4,5], -3/8, gamma, isBarrier=True)
19
```

```
qc2.barrier()

for q in range(n_qubits):
    qc2.rx(2 * beta, q)

qc2.measure_all()

qc2.draw('mpl', fold=-1)
```

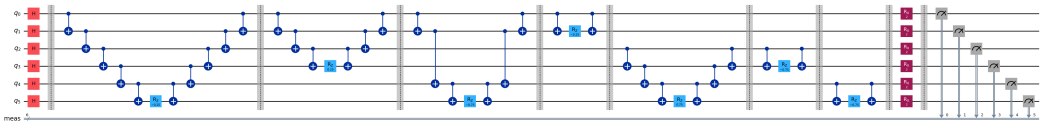

Figure 5: Quantum circuit of the switch-network encoding when $p = 1$.

3-c) Build a QAOA circuit for arbitrary p for the leg-variable encoding and initialize both β_i and γ_i to be linear functions of i. Choose p so that the total number of CNOT gates in the circuit does not exceed 3000. What is the maximum ground state probability you can obtain by varying the linear trajectories for β_i and γ_i?

Before measurement, the primary task is to determine the key parameters of the circuit. The main parameters include the circuit depth (number of layers) p, and the exploration intensity parameters β_i and γ_i.

Both β_i and γ_i take values from $[0, \pi]$ and $[0, 2\pi]$, respectively. Following a linear schedule for the i-th layer, β_i decreases linearly from its maximum value, while γ_i increases linearly from its minimum value.

The depth p must be chosen such that the number of CNOT gates used remains below 3000. In the leg-variable encoding used in this problem (Problem 3-a), the number of CNOT gates for $p = 1$ was 24. Hence, since $3000/24 = 125$, measurements can be performed up to a depth of approximately $p = 125$. While increasing p enhances the expressibility of the circuit, excessively large values make it difficult to find the optimal parameter region. Moreover, on real hardware, the increase in CNOT gates introduces more noise. Therefore, an appropriate value of p must be found to effectively solve the given problem.

In Listing 12, parameter optimization was performed while varying p, aiming to maximize the probability of measuring the ground state. The parameters β_i and γ_i were linearly interpolated. The optimized parameters were found to be $p = 92$, $\beta = \text{linspace}(4.584, -0.764)$, and $\gamma = \text{linspace}(0.091, 7.914)$. The parameters β_i and γ_i are adjusted at the end using a modulo operation to account for their periodicity. At these values, the probability of measuring the target state $|101101\rangle$ was 0.2699. The simulation results of this analysis are shown in Fig. 6.

Listing 12: Code for varying p to find the optimal parameters and calculating the theoretical values of the leg-variable encoding quantum circuit using the statevector.

```
ground_state_bits = '101101'
```

```
ground_state_str_qiskit = ground_state_bits[::-1]

def objective_function_leg_statevector(params, p):
    gamma_start, gamma_end, beta_start, beta_end = params
    gammas = np.linspace(gamma_start, gamma_end, p)
    betas  = np.linspace(beta_start, beta_end, p)

    qc = gen_qaoa_leg(n_qubits, p, gammas, betas)

    sv = Statevector.from_instruction(qc)

    ground_index = int(ground_state_str_qiskit, 2)
    ground_probs = np.abs(sv.data[ground_index])**2

    return -ground_probs

initial_params = [0, 2*np.pi, np.pi, 0]
results = []

for p in range(1, p_max_leg + 1):
    opt_result_leg = minimize(
        objective_function_leg_statevector,
                                initial_params,
                                args=(p,),
                                method='COBYLA',
                                options={'maxiter': 500, 'disp
                                    ': False})

    gamma_start, gamma_end, beta_start, beta_end =
        opt_result_leg.x
    optimized_params = [gamma_start, gamma_end, beta_start,
        beta_end]
    max_prob = -opt_result_leg.fun
    results.append((p, optimized_params, max_prob))

best_p, best_params, best_prob = max(results, key=lambda x: x
    [2])
print(f'best p: {best_p} (prob={best_prob})')
print(f'best parameter: {best_params}')
```

Measurements are performed using the optimized parameters obtained through theoretical analysis. The code used for measurement and visualization is shown in Listing 13, and the resulting outputs and visualizations are presented in Output 4 and Fig. 7, respectively. A deviation of approximately 0.04% from the theoretical value was observed, and it can be confirmed that the measured probability distribution closely matches the theoretical prediction.

Listing 13: Code for performing measurements on the leg-variable encoding circuit using the optimized parameters.

```
simulator = AerSimulator(max_parallel_threads=2)
p_leg_opt = best_p
n_shots = 1_000_000
```

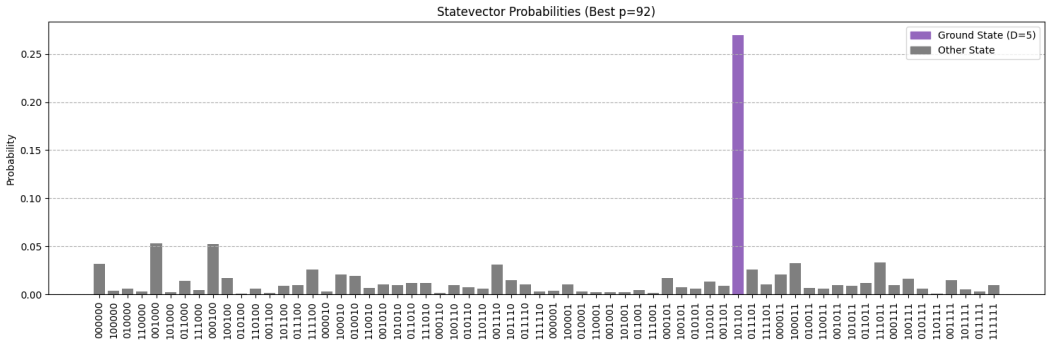

Figure 6: Graph of the theoretical values of the leg-variable encoding at the optimal parameters.

```
4
5  gammas = np.mod(np.linspace(opt_gamma_start, opt_gamma_end,
       best_p), np.pi*2)
6  betas  = np.mod(np.linspace(opt_beta_start, opt_beta_end,
       best_p), np.pi)
7
8  leg_qc = gen_qaoa_leg(n_qubits, p_leg_opt, gammas, betas)
9  leg_qc.measure_all()
10
11 result_opt = simulator.run(leg_qc, shots =n_shots).result()
12 counts_opt = result_opt.get_counts()
13
14 bitstrings = [format(i, f'0{n_qubits}b')[::-1] for i in range
       (2**n_qubits)]
15
16 probs = np.array([counts_opt.get(b, 0) / n_shots for b in
       bitstrings])
17
18 print(f'prob : {counts_opt[ground_state_bits]/n_shots}')
19
20 # ---
21
22 colors = ['tab:purple' if b == ground_state_bits else 'tab:
       gray' for b in bitstrings]
23
24 fig, ax = plt.subplots(figsize=(15, 6))
25 ax.bar(bitstrings, probs, color=colors, edgecolor='black',
       linewidth=0.5)
26
27 ax.set_ylabel('Measured Probability', fontsize=12)
28 ax.set_xlabel('Bitstring', fontsize=12)
29 ax.set_title(f'Measurement Probabilities (Best p={best_p})',
       fontsize=14, pad=10)
30 ax.tick_params(axis='x', labelrotation=90)
31 ax.grid(axis='y', linestyle='--', alpha=0.7)
32
33 purple_patch = mpatches.Patch(color='tab:purple', label=f'
       Ground State ({ground_state_bits})')
```

```
gray_patch = mpatches.Patch(color='tab:gray', label='Other
    States')
ax.legend(handles=[purple_patch, gray_patch], loc='upper right
    ', frameon=True)

plt.tight_layout()
plt.show()
```

Output 4: Probability of observing the target state (101101) when measuring the leg-variable encoding circuit with optimized parameters.

```
prob : 0.270361
```

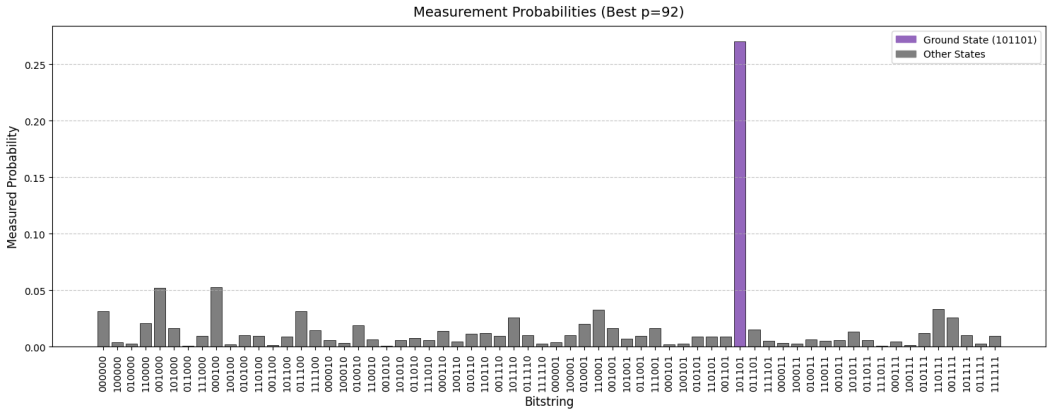

Figure 7: Probability distribution graph when measuring the leg-variable encoding circuit with optimized parameters.

3-d) Build a QAOA circuit for arbitrary p for the switch network encoding and initialize both β_i and γ_i to be linear functions of i. Choose p so that the total number of CNOT gates in the circuit does not exceed 3000, taking advantage of gate cancellation where possible. What is the maximum ground state probability you can obtain? Note that the ground state can be achieved in more than one computational basis state, so the ground state probability is a sum over probabilities obtained from appropriate computational basis states.

As in Problem 3-c, the parameters of the quantum circuit must be determined prior to measurement. Although the maximum number of CNOT gates is limited to 3000, successive pairs of identical CNOT operations can be canceled by exploiting their self-inverse property, a process known as gate cancellation [1]. This optimization can be performed in Qiskit using the `transpile` function with the `optimization_level` parameter. The quantum circuit for $p = 1$ after applying gate cancellation is shown in Fig. 8.

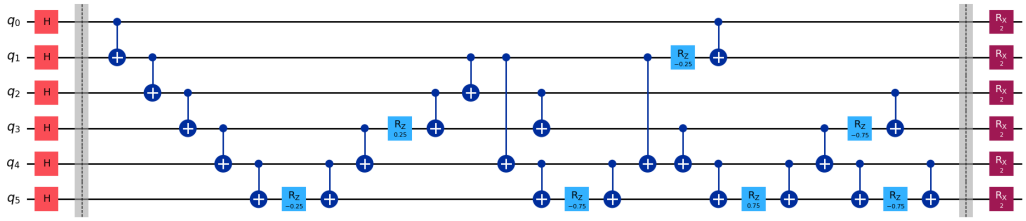

Figure 8: Switch-network encoding quantum circuit with gate cancellation applied ($p = 1$).

In the optimized circuit, each layer contains 22 CNOT gates, which is a significant reduction compared to the original 34 gates, demonstrating a considerable improvement in circuit efficiency.

Since $3000/22 = 136.36$, the switch network encoding method can reach a maximum depth of $p = 136$. In the same manner, the optimal parameters can be searched by increasing p as shown in Listing 14.

Listing 14: Code for varying p to find the optimal parameters and calculating the theoretical values of the switch-network encoding quantum circuit using the statevector.

```
def objective_function_switch_statevector(params, p):
    gamma_start, gamma_end, beta_start, beta_end = params

    gammas = np.linspace(gamma_start, gamma_end, p)
    betas  = np.linspace(beta_start, beta_end, p)

    qc = gen_qaoa_switch(n_qubits, p, gammas, betas)
    transpile(qc, optimization_level=1)

    sv = Statevector.from_instruction(qc)

    ground_probs = 0
    for combo, dist in opt_combos:
        if dist == 5:
            bits = ''.join(map(str, combo))
            qiskit_bits = bits[::-1]
            idx = int(qiskit_bits, 2)
            ground_probs += np.abs(sv.data[idx])**2

    return -ground_probs

initial_params = [0, 2*np.pi, np.pi, 0]
results = []

for p in range(1, p_max_switch + 1):
    opt_result_switch = minimize(
       objective_function_switch_statevector,
                                 initial_params,
                                 args=(p,),
```

```
                                    method='COBYLA',
                                    options={'maxiter': 500, 'disp
                                        ': False})

    gamma_start, gamma_end, beta_start, beta_end =
        opt_result_switch.x
    optimized_params = [gamma_start, gamma_end, beta_start,
        beta_end]
    max_prob = -opt_result_switch.fun
    results.append((p, optimized_params, max_prob))

best_p, best_params, best_prob = max(results, key=lambda x: x
    [2])
print(f'best p: {best_p} (prob={best_prob})')
print(f'best parameters: {best_params}')
```

The optimized parameter values are $p = 111$, $\beta = \text{linspace}(3.147, 0.027)$, and $\gamma = \text{linspace}(0.996, 7.240)$, where both parameters apply modular arithmetic at the final stage to account for periodicity. When these parameters are applied, the measured probability of the target value ($D = 5$) is 0.9815, which is a relatively high probability. The theoretical distribution corresponding to this result is illustrated in Fig. 9.

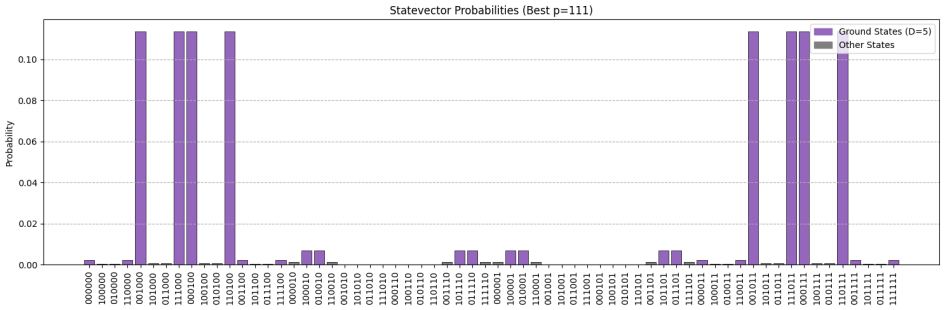

Figure 9: Graph of the theoretical values of the switch-network encoding at the optimal parameters.

As in the previous section, the optimized parameters obtained are then used for actual measurements. The code for performing the measurement and visualization is shown in Listing 15, and the measurement results and visualization correspond to Output 5 and Fig. 10, respectively.

Listing 15: Code for performing measurements on the switch-network encoding circuit using the optimized parameters.

```
simulator = AerSimulator(max_parallel_threads=2)
p_switch_opt = best_p
n_shots = 1_000_000

gammas = np.mod(np.linspace(opt_gamma_start, opt_gamma_end,
    best_p), np.pi*2)
betas  = np.mod(np.linspace(opt_beta_start, opt_beta_end,
    best_p), np.pi)
```

```
7
8  switch_qc = gen_qaoa_switch(n_qubits, p_switch_opt, gammas,
       betas)
9  switch_qc.measure_all()
10
11 result_opt = simulator.run(switch_qc, shots=n_shots).result()
12 counts_opt = result_opt.get_counts()
13
14 bitstrings = [format(i, f'0{n_qubits}b')[::-1] for i in range
       (2**n_qubits)]
15
16 probs = np.array([counts_opt.get(b, 0) / n_shots for b in
       bitstrings])
17
18 d5_states = [''.join(map(str, combo)) for combo, dist in
       opt_combos if dist == 5]
19 d5_states_qiskit = [s[::-1] for s in d5_states]
20
21 total_ground_prob = sum(counts_opt.get(state, 0) for state in
       d5_states_qiskit) / n_shots
22 print(f'prob: {total_ground_prob:.6f}')
23
24 # ---
25
26 colors = ['tab:purple' if b in d5_states_qiskit else 'tab:gray
       ' for b in bitstrings]
27
28 fig, ax = plt.subplots(figsize=(15, 6))
29 ax.bar(bitstrings, probs, color=colors, edgecolor='black',
       linewidth=0.5)
30
31 ax.set_ylabel('Measured Probability', fontsize=12)
32 ax.set_xlabel('Bitstring', fontsize=12)
33 ax.set_title(f'Measurement Probabilities (Best p={best_p})',
       fontsize=14, pad=10)
34 ax.tick_params(axis='x', labelrotation=90)
35 ax.grid(axis='y', linestyle='--', alpha=0.7)
36
37
38 purple_patch = mpatches.Patch(color='tab:purple', label='
       Ground States (D=5, total 24)')
39 gray_patch = mpatches.Patch(color='tab:gray', label='Other
       States')
40 ax.legend(handles=[purple_patch, gray_patch], loc='upper right
       ', frameon=True)
41
42 plt.tight_layout()
43 plt.show()
```

Output 5: Probability of observing switch configurations corresponding to $D = 5$ when measuring the switch-network encoding circuit with optimized parameters.

```
prob: 0.981239
```

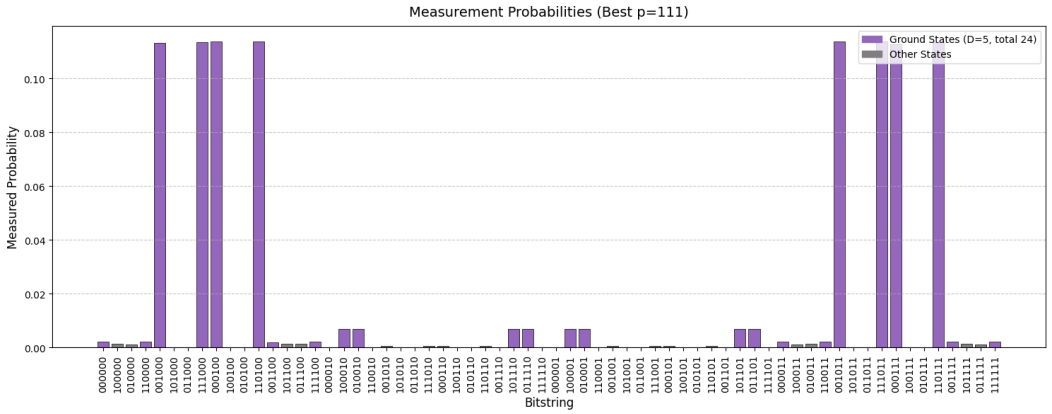

Figure 10: Probability distribution graph when measuring the switch-network encoding circuit with optimized parameters.

Conclusion

Compared with the theoretical result, approximately 0.04% deviation was observed, but the measured distribution appeared very similar to the theoretical one.

Unlike the leg variable encoding method, the switch network encoding approach includes all possible valid routes that can be output, and among them, it finds combinations corresponding to the shortest distance. This structural property explains why the switch network encoding yields higher probabilities.

Through this process, it was confirmed that the leg variable encoding method, although circuit-wise simpler, has a very limited feasible solution space, while the switch network encoding method, though more complex, can include a greater number of valid solutions. Therefore, the appropriate encoding method should be selected depending on the problem situation.

In this problem, only four cities were used as a simple example, and thus a classical computer could solve it quickly. However, as the number of cities increases, the computation time for classical computers increases exponentially, indicating that quantum approaches such as QAOA could provide a computational advantage.

References

[1] Colin Campbell and Edward D Dahl. Enhancing quantum optimization with parity network synthesis. *arXiv preprint arXiv:2402.11099*, 2024.