Accelerating the Held-Karp Algorithm for the Symmetric Traveling Salesman Problem

Kazuro KIMURA\(^1\)\(^a\), Shinya HIGA\(^1\), Nonmembers, Masao OKITA\(^1\)\(^\dagger\), and Fumihiko INO\(^1\)\(^\dagger\), Members

SUMMARY In this paper, we propose an acceleration method for the Held-Karp algorithm that solves the symmetric traveling salesman problem by dynamic programming. The proposed method achieves acceleration with two techniques. First, we locate data-independent subproblems so that the subproblems can be solved in parallel. Second, we reduce the number of subproblems by a meet in the middle (MITM) technique, which computes the optimal path from both clockwise and counterclockwise directions. We show theoretical analysis on the impact of MITM in terms of the time and space complexities. In experiments, we compared the proposed method with a previous method running on a single-core CPU. Experimental results show that the proposed method on an 8-core CPU was 9.5–10.5 times faster than the previous method on a single-core CPU. Moreover, the proposed method on a graphics processing unit (GPU) was 30–40 times faster than that on an 8-core CPU. As a side effect, the proposed method reduced the memory usage by 48%.

key words: symmetric traveling salesman problem, Held-Karp algorithm, parallelization, meet in the middle, GPU

1. Introduction

The traveling salesman problem (TSP) is a problem that finds the minimum Hamiltonian cycle for a weighted complete graph \( G = (V, E) \), where \( V \) and \( E \) are the sets of vertices and edges, respectively. A Hamiltonian cycle here is a cycle that visits each vertex exactly once and the minimum Hamiltonian cycle has the lowest-weight cycle, i.e., the lowest sum of edge weights, in the given graph \( G \). Hereafter, we call the minimum Hamiltonian cycle the optimal path. We also call the weight of the optimal path the optimal value.

One special case of the TSP is the symmetric TSP (sTSP), which deals with an undirected graph. That is, the given graph has the same weight between two vertices in each opposite direction: \( w(i, j) = w(j, i) \) holds for all vertices \( i, j \in V \), where \( w(i, j) \) represents the weight of the edge \((i, j) \in E \) from vertices \( i \) to \( j \). Many practical applications include the sTSP as an underlying fundamental problem. For example, welding robots can minimize their power consumption by solving the sTSP [1]. In more detail, an optimal motion plan can be obtained by solving the sTSP for a graph in which vertices represent the welding points and edge weights represent the costs required for moving between the welding points.

Algorithms for the sTSP can be classified into three approaches: exact, approximate, and heuristic approaches. Exact algorithms in [2], [3] find the optimal path but the problem size that can be solved in a reasonable time is limited due to high demand for both compute and memory capacities. By contrast, approximate and heuristic algorithms [4], [5] rapidly find an approximate path but the path is not optimal. Consequently, approximate and heuristic algorithms are typically used if the given graph has many vertices or the approximation error is acceptable for the overlying application. Exact algorithms are preferred otherwise. The latter case is of our interest where applications require an exact solution for the sTSP.

The Held-Karp algorithm [2] is an exact algorithm that deploys dynamic programming for the sTSP. To the best of our knowledge, the Held-Karp algorithm is the fastest exact algorithm [6] and its worst-case time complexity is \( O(n^22^n) \), where \( n \) is the number of vertices. Therefore, accelerating the Held-Karp algorithm increases the problem size \( n \) that can be solved in a reasonable time. The original paper [2] is a theoretical paper that presented the Held-Karp algorithm with its recursive equations and complexity analysis. By contrast, Kubo [7] presented a real implementation with specific data and loop structures. The contribution of [7] over [2] are twofold: the paper (1) shows a data structure that represents a set of vertices with a bit sequence; and (2) finds a loop structure that correctly solves data-dependent subproblems according to sorted bit sequences.

In this paper, we propose an acceleration method for the Held-Karp algorithm, which solves the sTSP. The proposed method achieves acceleration with two techniques, parallelization and meet in the middle (MITM). The former identifies data-independent subproblems from those generated by the Held-Karp algorithm. On the other hand, the latter approximately halves the number of subproblems by exploiting the symmetric attribute embedded in the given graph. Moreover, the proposed method reduces the memory usage as a side effect. We also show theoretical analysis for understanding the impact of MITM in terms of the time and space complexities. The source code of our implementation is available at http://www-ppl.ist.osaka-u.ac.jp/research/code/.

The paper is structured as follows. Section 2 introduces related work to clarify the contribution of the present paper. Section 3 presents an overview of the Held-Karp algorithm...
with [7]. Section 4 describes the proposed method with theoretical analysis. Section 5 shows experiments that evaluate the performance of the proposed method. Finally, Sect. 6 concludes the paper and discusses future work.

2. Related Work

Concorde [3] is an exact sTSP solver that relies on a cutting-plane method. This solver is capable of finding the optimal path for a large graph of \( n = 85,900 \). However, its execution time depends not only on the number \( n \) of vertices, i.e., the problem size, but also on the weight of edges. In fact, Ahammed et al. [8] found that the execution time differed by 30,000 times though \( n \) was fixed. On the other hand, the execution time of the proposed method depends only on the number \( n \) of vertices. Therefore, the proposed method is useful for strict real-time situations where the optimal path must be computed within a certain period of time.

Moffat [9] parallelized the Held-Karp algorithm for the sTSP using a JavaScript application programming interface (API). Similar to the proposed method, this method exploited the data parallelism inherent in the Held-Karp algorithm. However, the maximum problem size was limited by \( n \leq 16 \) because the method simply stored subproblems in \( O(n^2) \) space. By contrast, the proposed method saves the memory consumption by using a bit representation that is useful for identifying data-independent subproblems with \( O(2^n) \) space. Furthermore, the proposed method realizes further acceleration with MITM.

As for approximation approaches, these approaches return a solution that is theoretically guaranteed to be close to the optimal solution. Sahni and Gonzalez [10] proved that there is no algorithm that solves the sTSP in polynomial time. However, a polynomial time algorithm exists for the sTSP if edge weights in the graph satisfy the triangle inequality. Christofides [4] presented a 1.5 approximation algorithm for the sTSP. The worst time complexity of this algorithm is \( O(n^3) \), which is better than that of the Held-Karp algorithm. Therefore, approximation approaches are useful when the underlying application accepts solutions being within a certain bound.

In contrast to the approaches mentioned above, heuristic approaches are usually based on a rule of thumb, which returns a practical solution but not guaranteed to be optimal or within certain error bounds. The Lin-Kernighan algorithm [5] deploys an iterative approach that improves an initial solution by removing two or more edges and adding the same number of edges. The error to the optimal solution was usually 1–2%, making heuristic approaches more attractive than approximation approaches [7]. However, Chandra et al. [11] pointed out that the computed weight could be \( 4 \sqrt{n} \) times larger than the optimal value. Consequently, approximation approaches must be selected if the underlying application requires guarantee on the maximum error.

3. Held-Karp Algorithm

Let \( V = \{0, 1, \ldots, n-1\} \) be the set of vertices that compose a graph \( G \). Each vertex then can be identified with an integer value ranging from 0 to \( n - 1 \). The optimal path can be obtained by determining the best visiting order starting from a vertex and ending at the same vertex. In the following, let vertex \( n - 1 \) be the starting/ending vertex. Accordingly, we determine the visiting order for the remaining vertices, which compose a set \( V' = V \setminus \{n - 1\} = \{0, 1, \ldots, n - 2\} \).

To describe how the Held-Karp algorithm computes the optimal value \( OPT \), we define the function \( tsp \) as follows.

**Definition 1.** For \( S \subseteq V' \) and \( x \in S \), let \( tsp(S, x) \) be the minimum weight of the path that starts from the starting vertex \( n - 1 \), visits all vertices in set \( S \), and reaches vertex \( x \).

The optimal value \( OPT \) and \( tsp(S, x) \) then can be given by Eqs. (1) and (2), respectively [2].

\[
\begin{align*}
OPT &= \min_{x \in V} (tsp(V', x) + w(x, n - 1)), \quad (1) \\
tsp(S, x) &= \begin{cases} 
  w(n - 1, x), & \text{if } |S| = 1, \\
  \min_{g \subseteq S \setminus \{x\}} (tsp(S \setminus \{x\}, y) + w(y, x)), & \text{otherwise.} \quad (2)
\end{cases}
\end{align*}
\]

According to Eqs. (1) and (2), each value of \( tsp(S, x) \), where \( S \in \mathcal{P}(V') \setminus \emptyset \) and \( x \in S \), is accessed \( \max(n - 1 - |S|, 1) \) times to obtain \( OPT \). \( \mathcal{P}(-) \) here is the power set and \( \emptyset \) is the empty set. Figure 1 shows how these references occur during OPT computation, where the values of \( tsp(S, x) \) are accessed multiple times. This data dependence relation motivates the Held-Karp algorithm to deploy dynamic programming to avoid redundant computation; the algorithm stores the values of \( tsp(S, x) \) in table \( D_{tsp} \) to reuse the computed values without duplicated computation. This table can be indexed by pair \((S, x)\), which have \( \sum_{k=1}^{n-1} \binom{n-1}{k} \) cells, where \( \binom{n}{k} \) is the number of \( k \)-combinations from \( n \) elements, because \( x \) can take \( k \) different vertices when \( |S| = k \), where \( 1 \leq k \leq n - 1 \) (Fig. 1).

Next, the optimal path relies on the following definitions.

**Definition 2.** Let \( R = r_1 \rightarrow r_2 \rightarrow \cdots \rightarrow r_n \) be the path that visits \( n \) vertices in the following order: \( r_1, r_2, \ldots, r_n \).

![Fig. 1](image-url) Data dependence relation inherent in OPT computation for \( n = 4 \). The value of \( tsp(\{0\}, 0) \) is required to compute the values of \( tsp(\{0, 1\}, 1) \) and \( tsp(\{0, 2\}, 2) \).
Definition 3. Consider a path that has the weight of tsp\((S, x)\); the path starts from the starting vertex \(n - 1\), visits all vertices in set \(S \subseteq V'\), and reaches vertex \(x \in S\). Let \(\text{pr}(S, x) \in V\) denote the vertex visited immediately before vertex \(x\) on the path.

The vertex \(\text{pr}(S, x)\) then can be computed according to Eq. (3) as follows.

\[
\text{pr}(S, x) = \begin{cases} 
    n - 1, & \text{if } |S| = 1, \\
    \arg \min_{y \in S \setminus \{x\}} (\text{tsp}(S \setminus \{x\}, y) + w(y, x)), & \text{otherwise}, 
\end{cases} 
\]  

(3)

where the function \(\arg \min(\cdot)\) returns an argument that minimizes the value of the expression given to the function.

According to the above definitions, the optimal path \(n - 1 \to v_{n-1} \to \cdots \to v_1 \to n - 1\), where \(v_k\) represents the \(k\)-th vertex visited before reaching the ending vertex \(n - 1\), can be computed as follows.

\[
v_1 = \arg \min_{x \in V'} (\text{tsp}(V', x) + w(x, n - 1)), 
\]  

(4)

\[
v_k = \begin{cases} 
    \text{pr}(V', v_1), & \text{if } k = 2, \\
    \text{pr}(V' \setminus \{v_1, v_2, \cdots, v_{k-2}\}, v_{k-1}), & \text{otherwise}, 
\end{cases} 
\]  

(5)

That is, the optimal path can be identified from \(v_1\) by Eq. (4), then \(v_2, v_3, \cdots, v_{n-1}\) by Eq. (5).

Note here that the value of \(\text{pr}(S, x)\) can be simultaneously computed with that of \(\text{tsp}(S, x)\). Similar to \(\text{tsp}(S, x)\), which we mentioned earlier, redundant computation can be avoided by storing the value of \(\text{pr}(S, x)\) in table \(D_{\text{pr}}\).

3.1 Complexity Analysis

We first analyze \(L(n)\), or the space complexity of the Held-Karp algorithm. We assume that a single value occupies constant space. \(L(n)\) then can be given by

\[
L(n) = L_{\text{top}}(n) + L_{\text{pr}}(n), 
\]  

(6)

where \(L_{\text{top}}(n)\) and \(L_{\text{pr}}(n)\) are the space complexities for tables \(D_{\text{top}}\) and \(D_{\text{pr}}\), respectively. Since these tables have the same indexing scheme \((S, x)\), where \(S \in \mathcal{P}(V') \setminus \emptyset\) and \(x \in S\), we have

\[
L_{\text{top}}(n) = L_{\text{pr}}(n) = \sum_{k=1}^{n-1} k \binom{n - 1}{k}. 
\]  

(7)

Thus, the worst-case space complexity of the Held-Karp algorithm is \(O(n^{2n})\) because the binomial theorem rewrites Eq. (7) into

\[
L_{\text{top}}(n) = L_{\text{pr}}(n) = (n - 1)(n - 2)^{n-2}. 
\]

We next investigate \(T(n)\), or the time complexity of the Held-Karp algorithm, assuming a comparison as the elementary operation. We define subproblem \(P(S, x)\) as follows.

Definition 4. For \(S \subseteq V'\) and \(x \in S\), let \(P(S, x)\) denote the subproblem that computes the values of \(\text{tsp}(S, x)\) and \(\text{pr}(S, x)\), i.e., fills in tables \(D_{\text{top}}\) and \(D_{\text{pr}}\).

\[
T(n) \text{ can be given by } 
T(n) = T_1(n) + T_2(n), 
\]  

(8)

where \(T_1(n)\) is the time complexity for solving a set of all subproblems, \(\{P(S, x) | S \in \mathcal{P}(V') \setminus \emptyset, x \in S\}\), and \(T_2(n)\) is the time complexity for obtaining the optimal value and the optimal path from tables \(D_{\text{top}}\) and \(D_{\text{pr}}\), respectively. \(T_1\) and \(T_2\) can be expressed as follows.

\[
T_1(n) = \sum_{k=2}^{n-1} k(k - 1) \binom{n - 1}{k}, 
\]  

(9)

\[
T_2(n) = n - 1. 
\]  

(10)

Thus, the worst-case time complexity of the Held-Karp algorithm is \(O(n^{7/2}n)\) because the binomial theorem rewrites Eq. (9) into \(T_1(n) = (n - 1)(n - 2)^{n-3}\).

3.2 Previous Method

The previous method [7], which we denote hereafter by \(M_0\), deploys a bit string expression to represent set \(S \subseteq V'\) (Fig. 2). That is, set \(S\) is expressed by bit string \(s\), or a binary sequence of \(n - 1\) bits; the \(x\)-th bit of \(s\), where \(0 \leq x \leq n - 2\), is \(1\) if \(x \in S\) and \(0\) otherwise. Consequently, bit string \(s\) can be interpreted as an integer value in the range \([0, 2^{n-1} - 1]\). This value can be directly used as an index of tables \(D_{\text{top}}\) and \(D_{\text{pr}}\), which can be stored in two-dimensional arrays. As shown in Fig. 2, the tables are indexed with row \(s\) and column \(x\) to store the value for pair \((S, x)\), where \(S \in \mathcal{P}(V') \setminus \emptyset\) and \(x \in S\). Thus, table values can be rapidly accessed in constant time.

This indexing scheme has another benefit in obtaining a correct execution order for solving subproblems. As shown in Fig. 1, some subproblems must be computed before others due to data dependence. Filling up the arrays from lower to higher indices guarantees such a correct execution order (Fig. 2).

4. Proposed Method

The Held-Karp algorithm produces not only data-dependent
subproblems but also data-independent subproblems. The proposed method $M_1$ identifies such data-independent subproblems to process them in parallel. The method $M_1$ further achieves acceleration by MITM, which exploits the symmetric attribute available in the graph. The method $M_1$ also reduces the memory usage.

### 4.1 Parallelization Technique

The proposed method $M_1$ classifies subproblems into $n - 1$ groups such that subproblems in each group can be processed in parallel. In other words, subproblems belonging to different groups must be processed sequentially with $n - 1$ steps. The key idea for realizing this classification is to group subproblems by their size $k$, where $1 \leq k \leq n - 1$.

**Definition 5.** Let $|P(S, x)| = |S|$ be the size of subproblem $P(S, x)$.

More specifically, Eqs. (2) and (3) indicate that subproblems of size $k$ depend only on those of size $k - 1$ (Fig. 3). This implies that (1) at least $n - 1$ steps are required to process all $1 \leq k \leq n - 1$ and (2) subproblems of the same size $k$ are data-independent, allowing them to be solved in parallel.

Because the proposed method adopts the same indexing scheme as the previous method $M_2$ [7], the subproblem size $|S|$ can be easily computed according to the Hamming weight of bit string $s$, where $s$ corresponds to set $S \subseteq V'$.  

**Definition 6.** Let $d(s)$ be the number of 1s in bit string $s$.

Thus, we have $|S| = d(s)$ for any $S$, which means that data-independent subproblems can be classified into the same group by Hamming weight.

Notice here that (data-independent) subproblems that have the same Hamming weight involve non-contiguous rows in each table because the previous method $M_0$ stores subproblems in an increasing order of $s$ (Fig. 2). For example, subproblems of size 1 exist in the first, second, and fourth rows in Fig. 2. Thus, search operations are required to locate subproblems that form the same group, eliminating the advantage of constant-time access. To deal with this issue, our method sorts the subproblems, i.e., bit strings, by Hamming weight so that data-independent subproblems are stored in contiguous rows in each table (Fig. 4). This operation incurs an overhead, but once sorted, the data-independent subproblems can be located in constant time.

...
holds for the weights of paths $A$ and $B$.

If we have the Hamming weights of all bit strings. Since the number of bit strings of length $n - 1$ is $2^{n-1}$, the worst-case time complexity for obtaining all Hamming weights is given by $O(n2^n)$. As for sorting subproblems, the merge sort algorithm has the worst-case time complexity of $O(n2^n)$.

$T_{offset}(n) = O(n2^n)$ holds. The values of offset($k$) can be computed by traversing row indices from the top of the table $D_{bus};$ offset($k$) is the first index that has Hamming weight $k$.

Thus, the terms in Eq. (13) except $T_1$ are asymptotically negligible in comparison to $T_1$. Since $T_1$ is the time complexity for solving all subproblems, parallelizing this performance bottleneck leads to acceleration.

4.2 Meet in the Middle (MITM) Technique

The proposed method $M_2$ integrates the MITM technique into $M_1$. Given an undirected graph, MITM computes the optimal path from both clockwise and counterclockwise directions, solving subproblems of at most size $\lceil n/2 \rceil$ instead of those of at most $n - 1$ (Fig. 5). In comparison to methods $M_0$ and $M_1$, MITM roughly halves the number of subproblems that are to be solved.

We first explain how MITM obtains the optimal value. To do this, we introduce the following operators for paths.

Definition 9. Let + be the binary operator that concatenates two paths. That is, we obtain path $R + Q = r \rightarrow \cdots \rightarrow x \rightarrow \cdots \rightarrow q$ from paths $R = r \rightarrow \cdots \rightarrow x$ and $Q = x \rightarrow \cdots \rightarrow q$.

Definition 10. Let rev be the unary operator that returns the given path in reverse order. That is, we obtain path $\text{rev}(R) = r_n \rightarrow r_{n-1} \rightarrow \cdots \rightarrow r_1$ from path $R = r_1 \rightarrow r_2 \rightarrow \cdots \rightarrow r_n$.

Theorem 1. Obviously, $\text{rev}(\text{rev}(R)) = R$ holds.

Let $m$ be the midpoint of the optimal path (Fig. 5); since the optimal path contains $n$ vertices, $m$ is the $\lceil n/2 \rceil$-th vertex from the starting vertex. Consequently, the optimal path can be given by $A + B$, where $A = n - 1 \rightarrow a_1 \rightarrow \cdots \rightarrow a_{\lceil n/2 \rceil} \rightarrow m$ and $B = m \rightarrow b_1 \rightarrow \cdots \rightarrow b_{n-\lceil n/2 \rceil} \rightarrow n - 1$. Theorem 2 holds for the weights of paths $A$ and $B$.

Theorem 2. Let $S_A$ be the set of vertices that constitute path $A$. The weights of paths $A$ and $B$ are then given by $\text{tsp}(S_A, m)$ and $\text{tsp}(S_A \cup \{m\}, m)$, respectively, where $S_A$ is the complementary set of $S_A$ in $V'$.

Proof. According to Definition 1, the weights of paths $A$ and $B$ are $\text{tsp}(S_A, m)$ and $\text{tsp}(S_A \cup \{m\}, m)$, respectively. Since the graph consists of undirected edges, i.e., weights are symmetric, the weight of path $B$ equals to that of path $\text{rev}(B)$. Therefore, the weight of path $B$ is given by $\text{tsp}(S_A \cup \{m\}, m)$.

Equation (14) gives the optimal value $\text{OPT}$.

$$\text{OPT} = \min_{S \cup \{x\} \in F} \left( \text{tsp}(S, x) + \text{tsp}(S_2 \cup \{x\}, x) \right). \quad (14)$$

where set $F$ is given by

$$F = \left\{ (S, x) \mid S \in \mathcal{P}(V') \setminus \phi, x \in S \text{ such that } |S| = \lceil n/2 \rceil \right\}. \quad (15)$$

We next describe how the proposed method $M_2$ obtains the optimal path with MITM. Algorithm 1 returns the optimal path from set $S_A$ and midpoint $m$, which can be given by

$$S_A(m) = \arg \min_{S \cup \{x\} \in F} \left( \text{tsp}(S, x) + \text{tsp}(S_2 \cup \{x\}, x) \right). \quad (16)$$

We next analyze $T''(n)$, or the time complexity of $M_2$, which is given by

$$T''(n) = T_{\text{basis}}(n) + T_{\text{offset}}(n) + T'_1(n) + T'_2(n). \quad (17)$$

where $T'_1(n)$ is the time complexity for solving subproblems of size $k \leq \lceil n/2 \rceil$ and $T'_2(n)$ is that for obtaining the optimal value and path. The former term is given by

$$T'_1(n) = \sum_{k=2}^{\lceil n/2 \rceil} k(n - 1)B_k. \quad (18)$$

Lemmas 1 and 2 then give Theorem 3 whose proofs are presented in Appendix A, Appendix B and Appendix C, respectively.

Lemma 1. The time complexity $T'_1(n)$ for solving all subproblems of size $k \leq \lceil n/2 \rceil$ is given by

$$T'_1(n) = \begin{cases} \frac{T_1(n)}{2}, & \text{if } n \text{ is even,} \\ \frac{T_1(n)}{2} + \frac{(n - 1)(n - 2)}{2} \left( \frac{n - 3}{n - 3} \right)^2, & \text{otherwise.} \end{cases} \quad (19)$$

Algorithm 1 Optimal path computation with MITM.

1. Compute set $S_A$ and midpoint $m$ by Eq. (16);
2. Compute path $A$ by Eq. (5), which starts from prev($S_A, m$);
3. Compute path $\text{rev}(B)$ by Eq. (5), which starts from prev($S_A \cup \{m\}, m$);
4. Compute the optimal path $A + \text{rev}(\text{rev}(B))$;
Lemma 2. For any even number $n$, $\binom{n}{n/2} \in \Theta(2^n/\sqrt{n})$ holds.

Theorem 3. Given the time complexity $T_1(n)$ for solving subproblems for all $1 \leq k \leq n - 1$ and that $T'_1(n)$ for solving subproblems for all $k \leq \lceil n/2 \rceil$, we have

$$\lim_{n \to \infty} T'_1(n)/T_1(n) = 1/2.$$  \hspace{1cm} (20)

As for the latter term of Eq. (17), we have

$$T'_2(n) = \lceil n/2 \rceil (n - 1) - 1,$$  \hspace{1cm} (21)

because there are $\binom{n-1}{\lfloor n/2 \rfloor}$ sets whose size is $|S| = \lceil n/2 \rceil$. According to Lemma 2, $T'_2(n) \in \mathcal{O}(\sqrt{n}2^n)$ holds. Therefore, similar to $T_{bstr}(n)$ and $T_{offset}(n)$, $T'_2(n)$ is asymptotically negligible in comparison to $T_1$ in Eq. (17). In summary, MITM is expected to double the performance according to Theorem 3.

Note that MITM can be integrated into the previous method $M_0$ instead of $M_1$; this integration allows us to skip subproblems of larger than $\lceil n/2 \rceil$. However, we avoid this because MITM is able to reduce the memory usage if subproblems in tables $D_{bstr}$ and $D_{prev}$ are sorted by Hamming weight. Thus, there exists a common procedure in methods $M_1$ and $M_2$.

4.3 Reducing Memory Usage

The proposed method $M_2$ is allowed to store the solutions for the subproblems of size $k \leq \lceil n/2 \rceil$ whereas the previous method $M_0$ has to store the solutions for those of size $k \leq n - 1$. Therefore, the memory usage of $M_2$ asymptotically reduces to $1/2$ in comparison to that of $M_0$ as follows.

Theorem 4. Let $L'_{bstr}(n)$ and $L'_{prev}(n)$ be the space complexities of tables $D_{bstr}$ and $D_{prev}$ in method $M_2$, respectively. We then have

$$\lim_{n \to \infty} L'_{bstr}(n)/L_{bstr}(n) = 1/2,$$  \hspace{1cm} (22)

$$\lim_{n \to \infty} L'_{prev}(n)/L_{prev}(n) = 1/2.$$  \hspace{1cm} (23)

The proof for Theorem 4 is presented in Appendix D.

The above discussion assumes that necessary rows in tables $D_{bstr}$ and $D_{prev}$ exist in contiguous regions as shown in Fig. 6. In other words, tables in the previous method $M_0$ fail to save the memory usage because necessary rows exist in non-contiguous regions; the entire table must be allocated for execution.

Similar to the original method $M_0$, the tables must be accessed in constant time to achieve acceleration with MITM. To realize this constant time access, we introduce function rank as follows.

Definition 11. Consider the $i$-th bit string $s$, where $0 \leq i \leq 2^{n-1} - 1$ and $s = bstr(i)$, in a sequence of bit strings sorted by ascending order of Hamming weight where bit sequences of the same Hamming weight are further sorted by ascending order of numerical value. Let $\text{rank}(s)$ be the rank of string $s$ in the sequence of bit strings that have the same Hamming weight. $\text{rank}(s)$ is then given by $\text{rank}(s) = i - \text{offset}(d(s)).$

Using the function rank, the value for $(s, x)$ at the $k$-th step can be stored in the element at $(\text{offset}(k) + \text{rank}(s), x)$ of the two-dimensional array. As shown in Fig. 6, this indexing scheme allows necessary rows to be stored in a contiguous region and to be accessed in constant time. For example, a bit sequence of $11010$ has a hamming weight of $3$. Accordingly, this sequence corresponds to the subproblems of size $3$, which are processed at the third step ($k = 3$). We then have $\text{offset}(3) = 16$ according to Definition 8. Furthermore, we have $\text{rank}(11010) = 8$ because $\text{rank}(11010)$ gives the rank of $11010$ in sorted sequences that have a hamming weight of $3$: 00111, 01011, 01101, 01110, 10011, 10101, 10110, 11001, 11010 and 11100, each corresponding to $7, 11, 13, 14, 19, 21, 22, 25, 26$ and $28$ in decimal.

The time complexity $T_{\text{rank}}(n)$ for computing $\text{rank}(s)$ is negligible in comparison to $T'_1$. In more detail, $T_{\text{rank}}(n) \in \mathcal{O}(2^n)$ holds because the values of $\text{rank}(s)$ can be computed by traversing row indices from the top of table $D_{bstr}$. Similarly, the space complexity $L_{\text{rank}}(n)$ for storing $\text{rank}(s)$ is negligible in comparison to $L_{bstr}$ and $L_{prev}$; $L_{\text{rank}}(n) \in \mathcal{O}(2^n)$ holds if we store all values in table $D_{\text{rank}}$.

The space complexity for table $D_{bstr}$ can be further reduced by allocating rows only for two consecutive steps and overwriting the allocated rows that are not to be accessed. More specifically, tsp($S, x$) at the $(k + 1)$-th step depends only on that at the $k$-th step, which means that all values of tsp($S, x$) except those at the last step can be overwritten. Accordingly, the proposed method $M_2$ allocates only the rows for the latest two steps rather than all rows in table $D_{bstr}$. Furthermore, the method deploys a double buffering scheme that stores the rows computed at odd and even steps in tables $D_{bstr}$ and $D_{prev}$, respectively (Fig. 7). Each table holds the value for bit string $s$ in the rank($s$)-th row to realize constant time access to a contiguous region.

Finally, we analyze the space complexities of tables $D_{bstr}$ and $D_{prev}$. The number of values computed at the $k$-th step is maximized when $k = \lceil (n-1)/2 \rceil$. Each table
then requires \( \binom{n-1}{\lfloor (n-1)/2 \rfloor} \) rows and \( n - 1 \) columns to store all values generated at the \( k \)-th step. According to Lemma 2, both the space complexities for tables \( D_{tsp0} \) and \( D_{tsp1} \) can be given by \( O(\sqrt{n2^n}) \), which is negligible in comparison to \( L_{prev} \in O(n2^n) \).

5. Experiments

We evaluated the proposed methods \( M_1 \) and \( M_2 \) by comparing them with the previous method \( M_0 \). The evaluation metrics were the execution time, the speedup ratio, and the memory usage, which demonstrated the impact of parallelization and MITM.

For the experiments, all methods were implemented with the C language and OpenMP directives [12] to obtain performance results on a multi-core CPU. We adopted bucket sort to implement sorting operations on the CPU. Similarly, we implemented all methods except \( M_0 \) with the compute unified device architecture (CUDA) [13] to execute the methods on a GPU. Sort operations were implemented with the Thrust library [14]. Table 1 shows the specification of the deployed machine. Hyper-threading was enabled on the 8-core CPU, and thus, at most 16 threads were executed on 16 logical cores. Our CPU-based implementation parallelizes the \( k \)-th step with all CPU threads. On the other hand, the GPU-based implementation parallelizes the \( k \)-th step with \( \binom{n-1}{k} \) threads; each thread is responsible for processing a bit sequence and each block consists of 1024 threads.

As for the problem size \( n \), we varied the number of vertices from 4 to 29, which was the maximum number that could be solved on the GPU successfully; the execution for \( n \geq 30 \) failed due to exhaustion of GPU memory. Given a complete weighted graph, all the methods returned the optimal path with its value \( OPT \). Each edge in the graph had a random weight specified as an integer value in the range [1, 1023]. For each size of \( n \), we generated 10 graphs that had random weights. For each graph, we executed 50 times (500 times for each \( n \)) and the minimum time was selected for each \( n \). We found that the maximum time was close to the minimum time. For example, the execution times of \( M_2 \) on the GPU ranged from 1.16 s to 1.17 s for \( n = 29 \). Thus, the gap between the minimum and maximum times was negligible on our machine because the time complexity of the Held-Karp algorithm depends only on \( n \).

Figure 8 shows the execution time for different problem sizes (\( 4 \leq n \leq 29 \)). The proposed methods \( M_1 \) and \( M_2 \) were faster than the previous method \( M_0 \) when the problem size was sufficiently large (\( n \geq 13 \)) where efficient parallelization was achieved. In other words, small problem sizes solved within 0.001 s could not be accelerated by parallelization, which involved a synchronization overhead; a synchronization operation is necessary at every step to increase the problem size \( k \). We also found that \( M_2 \) was faster than \( M_1 \) because acceleration by MITM is independent from that by parallelization. That is, the former technique reduces the number of subproblems that are to be solved whereas the latter technique reduces the execution time for solving subproblems. In particular, \( M_2 \) running on the GPU was 30–40 times faster than that on the CPU.

All of execution times in Fig. 8 include data transfer time. The data transfer time of \( M_2 \) on the GPU for \( n = 29 \) was 65.2 \( \mu s \), which was negligible compared to the execution time of 1.16 s. Note that data transfer occurs only at the program initialization and finalization to send the input to the GPU and receive the output from the GPU, respectively. The input size is given by \( O(n^2) \), which corresponds to an adjacency matrix that stores graph information. On the other hand, the output size is given by \( O(n) \), which corresponds to the optimal path and value. Thus, the amount of data transfer is asymptotically negligible to that of computation, which has the time complexity of \( O(n^22^n) \). Note that our GPU-based implementation failed to overcome the CPU-based implementation when \( n < 16 \), because the problem size was not sufficient large to maximize the efficiency of GPU execution. The data transfer time for \( n = 4 \) occupied 12.6% of the execution time.

Figure 9 shows the speedup ratio obtained on the CPU.

---

**Table 1** Specification of experimental machine.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon Silver 4110 (8 cores)</td>
</tr>
<tr>
<td>Memory capacity (CPU)</td>
<td>96 GB</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla V100 PCIe</td>
</tr>
<tr>
<td>Memory capacity (GPU)</td>
<td>16 GB</td>
</tr>
<tr>
<td>GPU driver</td>
<td>396.44</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu 16.04 LTS</td>
</tr>
<tr>
<td>Development environment</td>
<td>GCC 5.4.0 and CUDA 9.2</td>
</tr>
<tr>
<td>Compile option</td>
<td>03 optimization</td>
</tr>
</tbody>
</table>

---

**Fig. 7** Double buffering scheme that alternatively uses tables \( D_{tsp0} \) and \( D_{tsp1} \) (\( n = 8 \)). Instead of allocating all rows in \( D_{tsp} \), limited rows are allocated only for two consecutive steps.

**Fig. 8** Comparison of the previous method \( M_0 \) and the proposed methods \( M_1 \) and \( M_2 \) in terms of execution time.
blems at the
were assigned to 16 CPU threads when
n
to hyper-threading, which dropped the e
ffi
umber of threads from 8 to 9. This slight decrease was due
the speedup slightly decreased when we increased the num-
mer-intensive applications
physical core. A similar behavior can be found in other
execution when multiple threads were assigned to a single
ample, the proposed method generated
prev
ffi
icient to make parallel threads busy. For ex-
ample, the proposed method generated \([n/2]^{(n-1)\choose (n/2)}\) subproblems at the \([n/2]\)-th step, so that 561, 632, 400 subproblems were assigned to 16 CPU threads when \(n = 29\). In Fig. 10, the speedup slightly decreased when we increased the number of threads from 8 to 9. This slight decrease was due to hyper-threading, which dropped the efficiency of thread execution when multiple threads were assigned to a single physical core. A similar behavior can be found in other memory-intensive applications [15]–[17].

Finally, we investigated the memory usage of the pro-
posed and previous methods. The memory usage was com-
puted assuming that tables \(D_{\text{prev}}, D_{\text{osp}}, D_{\text{bstr}}\), and \(D_{\text{rank}}\) have 1 B, 2 B, 4 B, and 4 B values, respectively. Note that we
avoided considering variables that can be stored in poly-
onomial space; In fact, these variables consume several KB
when \(n = 29\), which is negligible to other data. Figure 11
shows the reduction rate of memory usage, \(\gamma = (\alpha - \beta)/\alpha\),
where \(\alpha\) and \(\beta\) are the memory usage of \(M_0\) and that of
\(M_2\), respectively. We observed that the rate \(\gamma\) had a neg-
ative value when \(n \leq 7\) but increased with \(n\). The reason
for this negative value was due to the additional tables \(D_{\text{bstr}}\)
and \(D_{\text{offset}}\). On the other hand, the increase was yielded by
the memory reduction technique presented in Sect. 4.3; the
proposed method replaces table \(D_{\text{osp}}\) with two smaller tables
\(D_{\text{osp0}}\) and \(D_{\text{osp1}}\), which reduces the space complexity from
\(O(n2^n)\) to \(O(\sqrt{n}2^n)\). Consequently, the gap from \(M_0\) to \(M_2\)
increased with \(n\).

Note here that the memory usage showed a zigzag be-
behavior similar to that observed for the speedup (Fig. 9). This
behavior was also theoretically analyzed in Sect. 4.2. The
highest reduction rate of 48% was obtained when \(n = 28\),
which was the largest even problem size processed success-
fully on the deployed GPU. In theoretical, the memory
usage roughly doubles as the problem size increases by 1. Ac-
tually, method \(M_2\) consumed 5.2 GB and 11.2 GB of GPU
memory when \(n = 28\) and \(n = 29\), respectively. Thus, the
problem size of \(n = 30\) required approximately 22 GB of
GPU memory for execution, which was beyond the capacity
available on the deployed GPU.

6. Conclusion

In this paper, we proposed an acceleration method for the
Held-Karp algorithm, which is an exact algorithm for the
\(s\)TSP. The proposed method achieves acceleration with par-
allelization and MITM. We also showed theoretical analysis
for the time and space complexities of the proposed method.

Experimental results show that the proposed method
on an 8-core CPU was 9.5–10.5 times faster than the pre-
vious method on a single-core CPU. Moreover, the pro-
posed method on the latest GPU was 30–40 times faster than
that on the multi-core CPU. As a side effect, the proposed
method reduced the memory usage by 48%, which matches
to the theoretical analysis.

Future work is to deal with large problem sizes that
cannot be naively stored in the GPU memory due to memory
exhaustion. An out-of-core processing scheme [18], [19] may be useful for realizing this large-scale computation; the scheme divides data into small pieces and iteratively processes the pieces with overlapping GPU computation with CPU–GPU data transfer.

Acknowledgments
This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers JP15H01687 and JP16H02801. We are also grateful to the anonymous reviewers for their valuable comments.

References

Appendix A: Proof for Lemma 1

Let \( \mathbb{N}_0 \) be a set of non-negative integers. We introduce Lemmas 3 and 4 to show a proof for Lemma 1.

Lemma 3. For any \( n \in \mathbb{N}_0 \) and \( k \in \mathbb{N}_0 \), where \( k \leq n \),
\[
k(n) = n(n-1)
\]
(A·1)

Proof. Lemma 3 obviously holds according to the definition of the binomial coefﬁcient: \( (\binom{n}{k}) = n!/(k!(n-k)!) \).

Lemma 4. For any \( n \in \mathbb{N}_0 \),
\[
\sum_{k=0}^{n} \binom{2n+1}{k} = 2^{2n}
\]
(A·2)
\[
\sum_{k=0}^{n} \binom{2n}{k} = 2^{2n-1} + \frac{1}{2} \binom{2n}{n}
\]
(A·3)

Proof. First, we show a proof for Eq. (A·2). According to the symmetry rule for binomial coefﬁcients, we have
\[
\binom{2n+1}{k} = \binom{2n+1}{2n+1-k}
\]
(A·4)
for all \( 0 \leq k \leq 2n+1 \). Therefore, we have
\[
\sum_{k=0}^{n} \binom{2n+1}{k} = \sum_{k=n+1}^{2n+1} \binom{2n+1}{k}
\]
(A·5)
Using this equivalence, the left-hand side of Eq. (A·2) can be rewritten as follows.
\[
\sum_{k=0}^{n} \binom{2n+1}{k} = \frac{1}{2} \left( \sum_{k=0}^{n} \binom{2n+1}{k} + \sum_{k=0}^{n} \binom{2n+1}{k} \right)
\]
\[
= \frac{1}{2} \left( \sum_{k=0}^{n} \binom{2n+1}{k} + \sum_{k=n+1}^{2n+1} \binom{2n+1}{k} \right)
\]
\[
= \frac{1}{2} \left( \sum_{k=0}^{n} \binom{2n+1}{k} + \sum_{k=0}^{n} \binom{2n+1}{k} \right)
\]
(A·6)
Applying the binomial theorem to Eq. (A·6) gives
\[
\sum_{k=0}^{n} \binom{2n+1}{k} = \frac{1}{2} \sum_{k=0}^{2n+1} \binom{2n+1}{k} = 2^{2n}
\]
(A·7)
Thus, Eq. (A.2) holds.

We next show a proof for Eq. (A.3). Similar to Eq. (A.4), the symmetry rule for binomial coefficients gives

$$\binom{2n}{k} = \binom{2n}{2n-k}, \quad (A.8)$$

for all $0 \leq k \leq 2n$. Therefore, we have

$$\sum_{k=0}^{n-1} \binom{2n}{k} = \sum_{k=n+1}^{2n} \binom{2n}{k}. \quad (A.9)$$

Using this equivalence, the left-hand side of Eq. (A.3) can be rewritten as follows.

$$\sum_{k=0}^{n} \binom{2n}{k} = \frac{1}{2} \left\{ \sum_{k=0}^{n-1} \binom{2n}{k} + \binom{2n}{n} + \sum_{k=n+1}^{2n} \binom{2n}{k} + \binom{2n}{n} \right\}$$

$$= \frac{1}{2} \left\{ \sum_{k=0}^{n-1} \binom{2n}{k} + \binom{2n}{n} + \sum_{k=n+1}^{2n} \binom{2n}{k} + \binom{2n}{n} \right\}$$

$$= \frac{1}{2} \left\{ \sum_{k=0}^{n} \binom{2n}{k} + \binom{2n}{n} \right\} \quad (A.10)$$

Applying the binomial theorem to Eq. (A.10) gives

$$\sum_{k=0}^{n} \binom{2n}{k} = \frac{1}{2} \left\{ \sum_{k=0}^{n} \binom{2n}{k} + \binom{2n}{n} \right\}$$

$$= 2^{2n-1} + \frac{1}{2} \binom{2n}{n}. \quad (A.11)$$

Thus, Eq. (A.3) holds. ⊓⊔

Using Lemmas 3 and 4, we show a proof for Lemma 1.

**Proof.** By applying Lemma 3 twice to Eq. (18), we obtain

$$T'(n) = \sum_{k=2}^{\lceil n/2 \rceil} k(k-1) \binom{n-1}{k-1}$$

$$= (n-1)(n-2) \sum_{k=2}^{\lceil n/2 \rceil} \frac{(n-3)}{(k-2)}$$

$$= (n-1)(n-2) \frac{(n-3)}{2} \binom{n-4}{2}. \quad (A.12)$$

We then consider two cases.

**Case 1.** $n$ is even. Since $n-3$ is an even number, there is an integer $m$ such that $2m+1 = n-3$. Therefore, $\lceil n/2 \rceil - 2 = m$. Using these equations, we have

$$\sum_{l=0}^{\lfloor n/2 \rfloor-2} \binom{n-3}{l} = \sum_{l=0}^{m} \binom{2m+1}{l}. \quad (A.13)$$

Applying Lemma 4 to Eq. (A.13), we obtain

$$\sum_{l=0}^{\lfloor n/2 \rfloor-2} \binom{n-3}{l} = 2^{2m} = 2^{n-4}. \quad (A.14)$$

Therefore,

$$T'(n) = (n-1)(n-2) \frac{(n-3)}{2} \binom{n-4}{2}$$

$$= (n-1)(n-2)2^{n-4}$$

$$= T(n)/2. \quad (A.15)$$

**Case 2.** $n$ is odd. Since $n-3$ is an even number, there is an integer $m$ such that $2m = n-3$. Therefore, $\lceil n/2 \rceil - 2 = m$. Using these equations, we have

$$\sum_{l=0}^{\lfloor n/2 \rfloor-2} \binom{n-3}{l} = \sum_{l=0}^{m} \binom{2m}{l}. \quad (A.16)$$

Applying Lemma 4 to Eq. (A.16), we obtain

$$\sum_{l=0}^{\lfloor n/2 \rfloor-2} \binom{n-3}{l} = 2^{2m-1} + \frac{1}{2} \binom{2m}{m}$$

$$= 2^{n-4} + \frac{1}{2} \binom{n-3}{(n-3)/2}. \quad (A.17)$$

Therefore,

$$T'(n) = (n-1)(n-2) \frac{(n-3)}{2} \binom{n-4}{2}$$

$$= T(n)/2 + \frac{(n-1)(n-2)}{2} \binom{n-3}{(n-3)/2}. \quad (A.18)$$

Thus, by the two cases above, Lemma 1 holds. ⊓⊔

**Appendix B: Proof for Lemma 2**

**Proof.** According to the Wallis product [20] for $\pi$, we have

$$\lim_{n \to \infty} \frac{2^{2n}}{\sqrt{n} \binom{2n}{n}} = \sqrt{\pi}. \quad (A.19)$$

That is, using the $(e, \delta)$-definition of limit,

$$\forall \varepsilon > 0, \exists n_0 \in \mathbb{N}_0 \text{ s.t. } \forall n \in \mathbb{N}_0,$

$$n > n_0 \Rightarrow \left| \frac{2^{2n}}{\sqrt{n} \binom{2n}{n}} - \sqrt{\pi} \right| < \varepsilon, \quad (A.20)$$

which can be rewritten as

$$\exists \varepsilon_0 \text{ s.t. } 0 < \varepsilon_0 < \sqrt{\pi}, \exists n_0 \in \mathbb{N}_0 \text{ s.t. } \forall n \in \mathbb{N}_0,$

$$n > n_0 \Rightarrow \left| \frac{2^{2n}}{\sqrt{n} \binom{2n}{n}} - \sqrt{\pi} \right| < \varepsilon_0. \quad (A.21)$$
Then
\[
\left| \frac{2^{2n}}{\sqrt{n}} - \sqrt{n} \right| < \epsilon_0
\]
\[
\Rightarrow -\epsilon_0 < \frac{2^{2n}}{\sqrt{n}} - \sqrt{n} < \epsilon_0
\]
\[
\Rightarrow \sqrt{n} - \epsilon_0 < \frac{2^{2n}}{\sqrt{n}} < \sqrt{n} + \epsilon_0. \quad (A-22)
\]
Since \(0 < \epsilon_0 < \sqrt{n}\) gives \(\sqrt{n} - \epsilon_0 > 0\),
\[
\frac{1}{\sqrt{n} + \epsilon_0} < \frac{2^{2n}}{\sqrt{n}} < \frac{1}{\sqrt{n} - \epsilon_0}
\]
\[
\Rightarrow \frac{1}{\sqrt{n} + \epsilon_0} \frac{2^{2n}}{\sqrt{n}} < \left( \frac{2^{2n}}{n} \right) < \frac{1}{\sqrt{n} - \epsilon_0} \frac{2^{2n}}{\sqrt{n}}. \quad (A-23)
\]
Using \(c_1 = 1/(\sqrt{n} + \epsilon_0)\) and \(c_2 = 1/(\sqrt{n} - \epsilon_0)\),
\[
\exists n_0 \in \mathbb{N}_0 \text{ s.t. } \forall n \in \mathbb{N}_0,
\]
\[
n > n_0 \Rightarrow c_1 \frac{2^{2n}}{\sqrt{n}} < \left( \frac{2^{n}}{n} \right) < c_2 \frac{2^{2n}}{\sqrt{n}}. \quad (A-24)
\]
Thus, \(\left( \frac{2n}{n} \right) \in \Theta \left( \frac{2^{n}}{\sqrt{n}} \right)\). Assuming \(n\) be an even number, we obtain
\[
\left( \frac{n}{n/2} \right) \in \Theta \left( \frac{2^{n}}{\sqrt{n}} \right). \quad (A-25)
\]
\[
\square
\]
Appendix C: Proof of Theorem 3

Proof. We consider two cases.

Case 1. \(n\) is even. According to Lemma 1, Eq. (20) obviously holds.

Case 2. \(n\) is odd. According to Lemma 1,
\[
T_1(n)/T_1(n) = 1/2 + \frac{(n-1)(n-2)}{2} \frac{n-3}{(n-3)/2} = \frac{(n-1)(n-2)}{2} \frac{n-3}{(n-3)/2} \frac{T_1(n)}{T_1(n)}.
\]
\[
(A-26)
\]
According to Lemma 2, given an even number \(n - 3\), we have
\[
\left( \frac{n-3}{(n-3)/2} \right) \in \Theta \left( \frac{2^{n}}{\sqrt{n}} \right). \quad (A-27)
\]
Thus,
\[
\frac{(n-1)(n-2)}{2} \frac{n-3}{(n-3)/2} \in \Theta \left( \frac{2^{n}}{\sqrt{n}} \right). \quad (A-28)
\]
Since \(T_1(n) = (n-1)(n-2)2^{n-3} \in \Theta(n^{1.5}2^n)\), we obtain
\[
\lim_{n \to \infty} T_1(n)/T_1(n) = 1/2. \quad (A-29)
\]
Thus, by the two cases above, Theorem 3 holds. \(\square\)

Appendix D: Proof of Theorem 4

Proof. We show a proof for Eq. (22). As for Eq. (23), we omit the proof but it can be presented in a similar manner.

First, \(L_{up}(n)\) can be expressed as follows.
\[
L_{up}(n) = \sum_{k=1}^{[n/2]} \binom{n-1}{k}. \quad (A-30)
\]
By applying Lemma 3 to Eq. (A-30), we obtain
\[
L_{up}(n) = (n-1) \sum_{k=1}^{[n/2]} \binom{n}{k-1} = (n-1) \sum_{l=0}^{[n/2]-1} \binom{n}{l}. \quad (A-31)
\]
We then consider two cases.

Case 1. \(n\) is even. Since \(n - 2\) is an even number, there is an integer \(m\) such that \(2m = n - 2\). Therefore, \([n/2] - 1 = m\). Using these equations, we have
\[
\sum_{l=0}^{[n/2]-1} \binom{n}{l} = \sum_{l=0}^{m} \binom{2m}{l}. \quad (A-32)
\]
Applying Lemma 4 to Eq. (A-32), we obtain
\[
\sum_{l=0}^{[n/2]-1} \binom{n}{l} = 2^{2m-1} + \frac{1}{2} \binom{2m}{m} = 2^{n-3} + \frac{1}{2} \binom{n-2}{(n-2)/2}. \quad (A-33)
\]
Therefore,
\[
L_{up}(n) = (n-1) \sum_{l=0}^{[n/2]-1} \binom{n}{l} = (n-1)2^{n-3} + \frac{1}{2} \binom{n}{(n-2)/2}. \quad (A-34)
\]
According to Lemma 2, given an even number \(n - 2\), we have
\[
\left( \frac{n-2}{(n-2)/2} \right) \in \Theta \left( \frac{2^{n}}{\sqrt{n}} \right). \quad (A-35)
\]
Thus,
\begin{equation}
\frac{n-1}{2} \left( \frac{n-2}{(n-2)/2} \right) \in \Theta(\sqrt{n}2^n).
\tag{A-36}
\end{equation}

Since \( \text{L}_{\text{exp}}(n) = (n-1)2^{n-2} \in \Theta(n2^n) \), we obtain
\begin{equation}
\lim_{n \to \infty} \frac{\text{L}_{\text{exp}}'(n)}{\text{L}_{\text{exp}}(n)} = \lim_{n \to \infty} \left\{ \frac{n-1}{2} \left( \frac{n-2}{(n-2)/2} \right) \right\} = 1/2.
\tag{A-37}
\end{equation}

**Case 2. \( n \) is odd.** Since \( n-2 \) is an odd number, there is an integer \( m \) such that \( 2m+1 = n-2 \). Therefore, \([n/2]-1 = m+1\). Using these equations, we have
\begin{equation}
\sum_{l=0}^{[n/2]-1} \binom{n-2}{l} = \sum_{l=0}^{m+1} \binom{2m+1}{l} = \sum_{l=0}^{m} \binom{2m+1}{l} + \binom{2m+1}{m+1}.
\tag{A-38}
\end{equation}

Applying Lemma 4 to Eq. (A-38), we obtain
\begin{equation}
\sum_{l=0}^{[n/2]-1} \binom{n-2}{l} = 2^{2m} + \binom{2m+1}{m+1} = 2^{n-3} + \binom{n-2}{(n-1)/2}.
\tag{A-39}
\end{equation}

According to Lemma 3,
\begin{equation}
\binom{n-2}{(n-1)/2} = \frac{2(n-2)}{n-1} \binom{n-3}{(n-3)/2}.
\tag{A-40}
\end{equation}

Therefore,
\begin{equation}
\text{L}_{\text{exp}}'(n) = (n-1) \sum_{l=0}^{[n/2]-1} \binom{n-2}{l} = (n-1) \left\{ 2^{n-3} + \frac{2(n-2)}{n-1} \binom{n-3}{(n-3)/2} \right\} = (n-1)2^{n-3} + 2(n-2) \binom{n-3}{(n-3)/2} = \text{L}_{\text{exp}}(n)/2 + 2(n-2) \binom{n-3}{(n-3)/2}.
\tag{A-41}
\end{equation}

According to Lemma 2, given an even number \( n-3 \), we have
\begin{equation}
\binom{n-3}{(n-3)/2} \in \Theta\left(\frac{2^n}{\sqrt{n}}\right).
\tag{A-42}
\end{equation}

Thus,
\begin{equation}
2(n-2) \binom{n-3}{(n-3)/2} \in \Theta(\sqrt{n}2^n).
\tag{A-43}
\end{equation}

Since \( \text{L}_{\text{exp}}(n) = (n-1)2^{n-2} \in \Theta(n2^n) \), we obtain
\begin{equation}
\lim_{n \to \infty} \frac{\text{L}_{\text{exp}}'(n)}{\text{L}_{\text{exp}}(n)} = \lim_{n \to \infty} \left\{ \frac{2(n-2)}{\text{L}_{\text{exp}}(n)} \right\} = 1/2.
\tag{A-44}
\end{equation}

Thus, by the two cases above, Lemma 4 holds. \( \square \)