CmpE 300 - Project 1

Beyza Nur Deniz - 2021400285
Salih Can Erer - 2022400174

Contents

1 Naive Algorithm

1.1 Theoretical Analysis of the Naive Algorithm

We analyze the running time of the naive algorithm (HAMILTONIAN*NAIVE) that decides whether there exists a Hamiltonian* path of length n from *start* to *end* in the graph G produced by GENERATETRICKYGRAPH.

Notation

Throughout this section, n denotes the size of each component in G. Thus

$$|V| = 3n$$

and G is the disjoint union of three components of size n.

1.1.1 Basic Operation

We take as basic operation a single adjacency test performed inside HAMILTONIAN*CHECK, namely evaluating

$$H[\texttt{perm}[i], \texttt{perm}[i+1]]$$

and comparing it to 0. Let $T(n)$ denote the number of these basic operations performed by HAMILTONIAN*NAIVE on an input of parameter n.

1.1.2 Combinatorial Structure

The algorithm proceeds as follows:

- It enumerates all subsets $S \subseteq V$ of size n such that $\{start, end\} \subseteq S$.
- For each such S, it builds an $n \times n$ adjacency matrix H of the induced subgraph and then calls ALLPERMUTATIONS(H, s, t), where s, t are the indices of *start*, *end* in the list of vertices of S.
- ALLPERMUTATIONS enumerates all permutations of the n vertices with *start* fixed at position 0 and *end* fixed at position $n-1$, and for each permutation it calls HAMILTONIAN*CHECK.

We first count how many subsets and permutations are considered.

Lemma 1.1 Number of subsets

The number of subsets $S \subseteq V$ of size n containing both *start* and *end* is

$$B_n = \binom{3n-2}{n-2}.$$

Proof. We must choose an n-element subset S of V such that *start*, *end* $\in S$. These two vertices are fixed, so we only need to choose the remaining $n-2$ vertices from the remaining $3n-2$ vertices in $V \setminus \{start, end\}$. Hence

$$B_n = \binom{3n-2}{n-2}.$$

□

Lemma 1.2 Number of permutations per subset

For each such subset S, the number of permutations considered by AllPermutations is

$$P_n = (n-2)!.$$

Proof. Given S of size n, the algorithm fixes the positions of *start* and *end* in the permutation: *start* at position 0 and *end* at position $n-1$. The remaining $n-2$ vertices can be permuted arbitrarily in the remaining $n-2$ positions, yielding $(n-2)!$ permutations. □

By the two lemmas above, the total number of (subset, permutation) pairs examined is:

$$B_n \cdot P_n = \binom{3n-2}{n-2}(n-2)! = \frac{(3n-2)!}{(n-2)!(2n)!}(n-2)! = \frac{(3n-2)!}{(2n)!}. \tag{1}$$

1.1.3 Best-Case Running Time

In the best case, the algorithm returns `True` as early as possible. This happens if:

- the very first subset S processed by the outer loop is the "correct" one containing a Hamiltonian* path from *start* to *end*, and
- the very first permutation generated by AllPermutations corresponds to a valid Hamiltonian* path.

Then only one permutation is checked. Since it is a valid Hamiltonian* path, HamiltonianCheck must verify all $n-1$ edges and thus performs $n-1 = \Theta(n)$ adjacency tests.

$$T_{\text{best}}(n) = n - 1 \in \Theta(n).$$

Proposition 1.3 Best-case complexity

$$T_{\text{best}}(n) \in \Theta(n).$$

1.1.4 Worst-Case Running Time

In the worst case the algorithm performs the maximum amount of work. This occurs, for example, when no Hamiltonian* path exists from *start* to *end* and, for every permutation examined, HamiltonianCheck fails only on the last edge (so it performs the maximum number of checks).

Fix a subset S:

- By Lemma 1.2, there are $P_n = (n-2)!$ permutations.
- For each permutation, HamiltonianCheck verifies $n-1$ edges before failing, so it performs $\Theta(n)$ basic operations.

Thus the contribution to $T(n)$ from a single subset is

$$(n-2)! \cdot \Theta(n) = \Theta(n \cdot (n-2)!).$$

By Lemma 1.1, there are $B_n = \binom{3n-2}{n-2}$ subsets, so the total worst-case running time is

$$T_{\text{worst}}(n) = B_n \cdot \Theta\big(n \cdot (n-2)!\big) = \Theta\big(B_n (n-2)! \cdot n\big).$$

Using (1), we obtain

$$B_n (n-2)! = \frac{(3n-2)!}{(2n)!},$$

and therefore

$$T_{\text{worst}}(n) \in \Theta\left(\frac{(3n-2)!}{(2n)!} \cdot n\right).$$

Proposition 1.4 Worst-case complexity

$$T_{\text{worst}}(n) \in \Theta\left(\frac{(3n-2)!}{(2n)!} \cdot n\right).$$

1.1.5 Average-Case Running Time

We now analyze the average running time with respect to the random input distribution defined by GenerateTrickyGraph.

The generator constructs three disjoint components A, B, C, each on n vertices, with edges only within each component. Then it chooses *start* and *end* uniformly at random among all ordered pairs of distinct vertices. A Hamiltonian* path of length n must lie entirely inside a single component. Therefore such a path can exist only if *start* and *end* are in the same component.

Let T be the random variable "number of basic operations" of the naive algorithm on a random input. Define events

$$A := \{\textit{start} \text{ and } \textit{end} \text{ lie in the same component}\}, \qquad A^c := \{\textit{start} \text{ and } \textit{end} \text{ lie in different components}\}.$$

Probabilities of A and A^c. We compute $\Pr(A)$.

The probability that *start* is in any fixed component (say A) is

$$\Pr(\textit{start} \in A) = \frac{n}{3n} = \frac{1}{3}.$$

Conditioned on $\textit{start} \in A$, the probability that *end* is also in A is

$$\Pr(\textit{end} \in A \mid \textit{start} \in A) = \frac{n-1}{3n-1}.$$

There are 3 components, so

$$\Pr(A) = 3 \cdot \frac{1}{3} \cdot \frac{n-1}{3n-1} = \frac{n-1}{3n-1}, \qquad \Pr(A^c) = 1 - \Pr(A) = \frac{2n}{3n-1}.$$

Law of total expectation. By the law of total expectation (tower rule),

$$\mathbb{E}[T] = \mathbb{E}\big[\,\mathbb{E}[T \mid A]\,\big] = \Pr(A)\,\mathbb{E}[T \mid A] + \Pr(A^c)\,\mathbb{E}[T \mid A^c]. \tag{2}$$

We now estimate the two conditional expectations.

Case A^c: start and end in different components. If A^c occurs, there is *no* Hamiltonian* path of length n between *start* and *end*, because any such path must stay within a single component. Thus the algorithm never returns `True`, and it inspects every subset S counted in Lemma 1.1.

Lemma 1.5 Constant expected cost of Hamiltonian*Check on mixed subsets

Fix $n \geq 4$ and consider a subset S of size n that contains vertices from at least two distinct components of G (i.e. S is *mixed*). Let π be a permutation of S chosen according to the distribution used by AllPermutations (with start and end fixed at the endpoints, and the remaining $n-2$ vertices in a uniformly random order). Let L_S denote the number of adjacency tests performed by Hamiltonian*Check on input path π before it terminates (either by encountering a missing edge or by successfully checking all $n-1$ edges). Then there exists a universal constant $C > 0$ such that

$$\mathbb{E}[L_S] \leq C$$

for all n and all mixed subsets S.

Proof. We view Hamiltonian*Check as follows: given the ordered sequence $\pi(0), \ldots, \pi(n-1)$, it inspects, in order, the edges

$$e_i := (\pi(i), \pi(i+1)), \qquad i = 0, 1, \ldots, n-2,$$

and stops at the first index i for which the adjacency matrix H has $H[\pi(i), \pi(i+1)] = 0$ (or after verifying all $n-1$ edges are present). Thus L_S is exactly the largest integer k such that the first $k-1$ edges $e_0, \ldots, e_{k-2}$ are present (and, if $k \leq n-1$, e_{k-1} is absent). In particular, for every $k \geq 1$,

$$\{L_S \geq k\} = \{e_0, \ldots, e_{k-1} \text{ are all present}\},$$

with the convention that the event $\{L_S \geq 1\}$ is trivial and has probability 1.

We now use the structure of G under GenerateTrickyGraph. The vertex set splits into three components A, B, C of size n each, with the following properties:

- There is a fixed backbone path in each component, whose edges are always present.
- Every other edge *within* a component (non-backbone) is present independently with probability $1/2$.
- There are no edges *between* different components: if u and v are in different components, then the adjacency entry for (u, v) is deterministically 0.

Since S is mixed, it contains vertices from at least two different components. For each i, the random edge $e_i = (\pi(i), \pi(i+1))$ falls into one of three types:

1. a backbone edge within a component (always present),
2. a non-backbone edge within a component (present with probability $1/2$),
3. a cross-component pair (endpoints in different components, always absent).

In particular, the presence indicator of e_i is a Bernoulli random variable taking value 1 with probability at most $1/2$, because cross-component edges are never present and non-backbone edges are present with probability $1/2$. Backbone edges only increase the probability of presence up to (but not beyond) 1, so a universal upper bound of $1/2$ is conservative for our purposes.

Moreover, the existence of different non-backbone edges is independent by construction of GenerateTrickyGraph, and the presence or absence of cross-component edges is deterministic. Thus, for any fixed $k \geq 1$, the events "e_i is present" for $i = 0, \ldots, k-1$ are independent, and we have

$$\Pr(e_i \text{ is present}) \leq \frac{1}{2} \quad \text{for each } i.$$

Therefore, for every integer $k \geq 2$,

$$\Pr(L_S \geq k) = \Pr(e_0, \ldots, e_{k-1} \text{ are all present}) = \prod_{i=0}^{k-1} \Pr(e_i \text{ is present}) \;\leq\; \left(\frac{1}{2}\right)^{k-1}.$$

For $k = 1$ we simply have $\Pr(L_S \geq 1) = 1$.

We now apply the tail-sum formula for nonnegative integer-valued random variables:

$$\mathbb{E}[L_S] = \sum_{k=1}^{\infty} \Pr(L_S \geq k) \leq 1 + \sum_{k=2}^{\infty} \left(\frac{1}{2}\right)^{k-1} = 1 + \frac{1/2}{1 - 1/2} = 2.$$

In particular, taking $C = 2$ works for all n and for every mixed subset S, as claimed. □

Hence, for a fixed subset S, the expected number of basic operations (adjacency tests in HAMILTONIAN*CHECK) is

$$\mathbb{E}[\text{cost per subset} \mid A^c] = (n-2)! \cdot \Theta(1) = \Theta((n-2)!).$$

There are B_n such subsets, so

$$\mathbb{E}[T \mid A^c] = B_n \cdot \Theta((n-2)!) = \Theta\big(B_n(n-2)!\big) = \Theta\left(\frac{(3n-2)!}{(2n)!}\right),$$

using (1).

Case A: start and end in the same component. If A occurs, there is exactly one "pure" subset S^* of size n that consists of all vertices of the component containing *start* and *end*. Only this subset can possibly contain a Hamiltonian* path of length n; all other subsets of size n containing *start* and *end* are "mixed" and behave just as in the A^c case (their permutations cannot yield a valid path).

Because GENERATETRICKYGRAPH applies a random permutation to all vertex labels, the position of S^* in the deterministic order in which subsets are generated is, by symmetry, uniformly distributed over $\{1, \ldots, B_n\}$. Let J be the index of S^*, so

$$\mathbb{E}[J] = \frac{B_n + 1}{2}.$$

Let C_{wrong} denote the random cost of a mixed subset (same distribution as in the A^c case), and let C_{good} be the cost of processing S^*. Then

$$T \mid A = \sum_{i=1}^{J-1} C_{\text{wrong},i} + C_{\text{good}},$$

where the $C_{\text{wrong},i}$ are i.i.d. copies of C_{wrong}. Taking expectations and using independence,

$$\mathbb{E}[T \mid A] = \mathbb{E}[J-1]\,\mathbb{E}[C_{\text{wrong}}] + \mathbb{E}[C_{\text{good}}].$$

As above,

$$\mathbb{E}[C_{\text{wrong}}] = \Theta((n-2)!).$$

Since $J - 1$ is uniformly distributed on $\{0, 1, \ldots, B_n - 1\}$, we have

$$\mathbb{E}[J-1] = \frac{B_n - 1}{2} = \Theta(B_n).$$

Thus

$$\mathbb{E}[J-1]\,\mathbb{E}[C_{\text{wrong}}] = \Theta\big(B_n(n-2)!\big).$$

The term $\mathbb{E}[C_{\text{good}}]$ is at most the worst-case cost of processing a single subset, which is $O(n(n-2)!)$; therefore

$$\mathbb{E}[C_{\text{good}}] = O(n(n-2)!).$$

We can observe that $B_n(n-2)! \gg n(n-2)!$ for large n, hence the contribution of $\mathbb{E}[C_{\text{good}}]$ is asymptotically negligible. Consequently,

$$\mathbb{E}[T \mid A] = \Theta(B_n(n-2)!) = \Theta\left(\frac{(3n-2)!}{(2n)!}\right).$$

Putting it together. Substituting the two conditional expectations into (2) gives

$$\mathbb{E}[T] = \Pr(A)\,\Theta\left(\frac{(3n-2)!}{(2n)!}\right) + \Pr(A^c)\,\Theta\left(\frac{(3n-2)!}{(2n)!}\right) = \Theta\left(\frac{(3n-2)!}{(2n)!}\right),$$

because $\Pr(A)$ and $\Pr(A^c)$ are bounded away from zero and one as $n \to \infty$.

Proposition 1.6 Average-case complexity

With respect to the input distribution defined by GenerateTrickyGraph,

$$\mathbb{E}[T(n)] \in \Theta\left(\frac{(3n-2)!}{(2n)!}\right).$$

1.1.6 Summary of Naive Algorithm

If we count each adjacency check in Hamiltonian*Check as one basic operation, the naive algorithm has

$$T_{\text{best}}(n) \in \Theta(n), \qquad T_{\text{worst}}(n) \in \Theta\left(\frac{(3n-2)!}{(2n)!} \cdot n\right), \qquad \mathbb{E}[T(n)] \in \Theta\left(\frac{(3n-2)!}{(2n)!}\right),$$

so both the worst-case and average-case running times grow faster than c^n for every fixed constant $c > 0$.

1.2 Experimental Analysis of the Naive Algorithm

To validate our theoretical analysis, we conducted experiments measuring the actual running time of the naive algorithm on randomly generated graphs produced by GenerateTricky-Graph.

1.2.1 Experimental Setup

We measured the average running time (in seconds) of the naive algorithm for component sizes $n \in \{4, 5, 6, 7, 8\}$, corresponding to total graph sizes $N \in \{12, 15, 18, 21, 24\}$. For each value of n, we ran the algorithm on multiple random instances and recorded the mean execution time.

1.2.2 Observed Results

Figure 1 shows the experimental results. The running time remains negligible (under 0.1 seconds) for $n \leq 6$, but exhibits dramatic growth beyond this threshold:

n	Average Time (seconds)
4	≈ 0.01
5	≈ 0.02
6	≈ 0.04
7	≈ 0.45
8	≈ 13.0

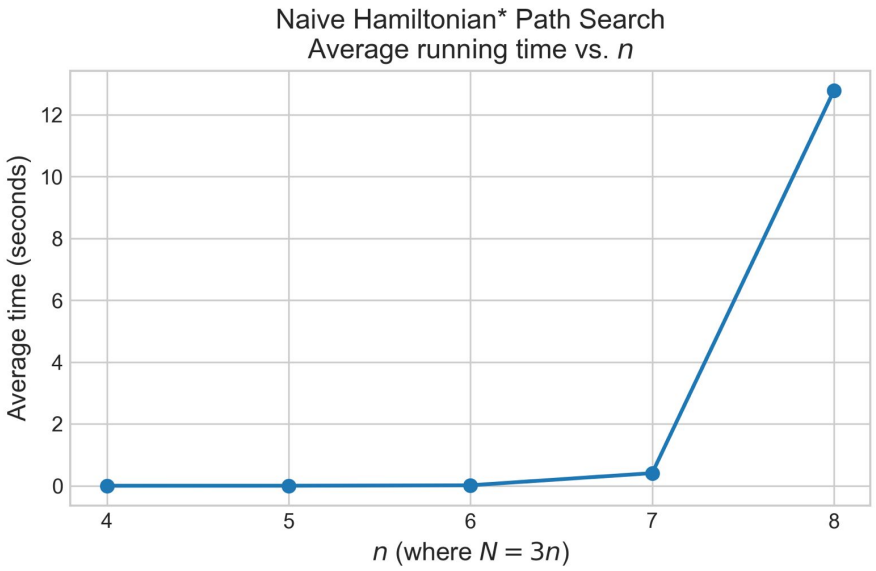

Figure 1: Average running time of the naive Hamiltonian* path search algorithm as a function of component size n (where $N = 3n$). Note the explosive growth for $n \geq 7$.

The most striking observation is the explosive growth between $n = 7$ and $n = 8$: the running time increases by a factor of approximately $29\times$ when n increases by just one unit.

1.2.3 Comparison with Theoretical Predictions

Recall from our theoretical analysis that the average-case running time of the naive algorithm is

$$\mathbb{E}[T(n)] \in \Theta\left(\frac{(3n-2)!}{(2n)!}\right).$$

Let us verify this ridiculous growth rate empirically. For consecutive values of n, we compute the *growth ratio*:

$$r_n := \frac{\text{Time}(n+1)}{\text{Time}(n)}.$$

From the experimental data:

$$r_7 = \frac{0.45}{0.04} \approx 11.25,$$
$$r_8 = \frac{13.0}{0.45} \approx 28.9.$$

Now, let us compare this with the theoretical prediction. Using the factorial formula, the theoretical growth ratio is

$$\frac{(3(n+1)-2)!/(2(n+1))!}{(3n-2)!/(2n)!} = \frac{(3n+1)!}{(3n-2)!} \cdot \frac{(2n)!}{(2n+2)!} = \frac{(3n+1)\cdot 3n \cdot (3n-1)}{(2n+2)(2n+1)}.$$

For $n = 7$ and $n = 8$:

$$\text{Theoretical } r_7 = \frac{22 \cdot 21 \cdot 20}{16 \cdot 15} = \frac{9240}{240} = 38.5,$$
$$\text{Theoretical } r_8 = \frac{25 \cdot 24 \cdot 23}{18 \cdot 17} = \frac{13800}{306} \approx 45.1.$$

The experimental growth ratios $(11.25, 28.9)$ are of the same order of magnitude as the theoretical predictions $(38.5, 45.1)$, confirming the factorial growth. The slight discrepancy arises because:

- Our theoretical analysis counts basic operations (adjacency tests), while actual running time includes implementation overhead.
- The algorithm may terminate early in practice when a valid path is found, whereas our average-case analysis accounts for the full search over all subsets and permutations.
- Constant factors hidden in the Θ notation can differ from measured times for small values of n.

1.2.4 Extrapolation and Infeasibility

The experimental results demonstrate why the naive algorithm is infeasible for even moderately sized inputs. Extrapolating from the observed growth rate of approximately $30\times$ per unit increase in n:

- For $n = 9$: estimated time $\approx 13 \times 30 = 390$ seconds ≈ 6.5 minutes.
- For $n = 10$: estimated time $\approx 390 \times 30 = 11,700$ seconds ≈ 3.25 hours.
- For $n = 11$: estimated time $\approx 3.25 \times 30 \approx 97.5$ hours ≈ 4 days.

This matches our theoretical conclusion: the naive algorithm has enormous complexity

$$\mathbb{E}[T(n)] \in \Theta\left(\frac{(3n-2)!}{(2n)!}\right),$$

which grows much faster than any exponential function c^n. Even for $n = 10$ (total graph size $N = 30$), the algorithm becomes practically unusable.

Proposition 1.7 Experimental validation

The experimental running times confirm the theoretical analysis: the naive algorithm exhibits super-exponential factorial growth, making it infeasible for $n \geq 9$ in practice.

2 Optimized Algorithm

2.1 Theoretical Analysis of the Optimized Algorithm

We now analyze the optimized algorithm which exploits the fact that the graph G on $3n$ vertices is always the disjoint union of three connected components, each of size n.

Any Hamiltonian* path of length n from *start* to *end* must lie entirely within a single component, namely the connected component of *start*.

2.1.1 Description of the Optimized Algorithm

Let $N = 3n$ be the total number of vertices.

1. Run a graph traversal (we preferred DFS) from *start* on the full $N \times N$ adjacency matrix and let $S^* \subseteq V$ be the set of visited vertices (the connected component of *start*).
2. If $end \notin S^*$, return `False`.
3. Otherwise, $|S^*| = n$ by construction. Build the $n \times n$ adjacency matrix H of the subgraph induced by S^* (with some fixed ordering L of the vertices), compute the indices s, t of *start* and *end* in L, and return the result of

$$\textsc{AllPermutations}(H, s, t).$$

We keep the same basic operation as in the naive analysis:

Basic operation. A single adjacency test inside HAMILTONIAN*CHECK, i.e.

$$H[\texttt{perm}[i], \texttt{perm}[i+1]]$$

and the comparison to 0 or 1. Let $T(n)$ be the number of such basic operations performed by the optimized algorithm on an input with parameter n.

2.1.2 Preliminary Observations

Once S^* is known and $end \in S^*$, the call $\textsc{AllPermutations}(H, s, t)$ proceeds exactly as in the naive algorithm but restricted to a single n-vertex component.

Lemma 2.1 Number of permutations in AllPermutations

If S^* has size n and we fix the positions of *start* and *end* in the permutation to be 0 and $n-1$, then the number of permutations examined by ALLPERMUTATIONS is

$$P_n = (n-2)!.$$

Proof. Exactly as in Lemma 1.2 for the naive algorithm: we have n vertices in S^*; *start* and *end* are fixed at the first and last positions of the permutation, and the remaining $n-2$ vertices can be arranged arbitrarily in the $n-2$ middle positions. Thus there are $(n-2)!$ permutations. □

Each call to HAMILTONIAN*CHECK on a permutation performs between 1 and $n-1$ adjacency checks, stopping as soon as a missing edge is encountered, or after $n-1$ checks if the permutation is a valid Hamiltonian* path.

2.1.3 Best-Case Running Time

The best case with respect to our basic operation is when it does *not* have to call ALLPERMUTATIONS at all.

This happens when *start* and *end* lie in different components: after the traversal, we have $end \notin S^*$ and the algorithm returns `False` immediately.

Proposition 2.2 Best-case complexity

In terms of the basic operation (adjacency tests inside HAMILTONIAN*CHECK),

$$T_{\text{best}}(n) \in O(1).$$

Proof. In this best case, the algorithm never calls AllPermutations, and hence never calls Hamiltonian*Check. Therefore it performs zero adjacency tests, independently of n. In our basic-operation model this means $T_{\text{best}}(n) = 0 \in O(1)$. □

2.1.4 Worst-Case Running Time

The worst case is when *start* and *end* lie in the same component S^*, and the call to AllPermutations behaves in its own worst case.

We can choose a connected n-vertex component such that there is either no Hamiltonian* path from *start* to *end*, or such a path exists but the permutation that realizes it appears last in the order generated by AllPermutations. Furthermore, we may arrange that every checked permutation is "almost" valid in the sense that Hamiltonian*Check only fails on the last edge. This is compatible with the problem's construction constraints (we only require connectedness inside each component).

Proposition 2.3 Worst-case complexity

$$T_{\text{worst}}(n) \in \Theta((n-1)!).$$

Proof. By Lemma 2.1, AllPermutations examines $P_n = (n-2)!$ permutations. In the worst case, each call to Hamiltonian*Check runs through all $n-1$ edges of the permutation before failing; hence each such call costs $\Theta(n)$ basic operations. Thus the contribution to $T(n)$ from AllPermutations is

$$(n-2)! \cdot \Theta(n) = \Theta(n(n-2)!).$$

Therefore

$$T_{\text{worst}}(n) = \Theta(n(n-2)!) = \Theta((n-1)!),$$

because $n(n-2)! = (n-1)! \cdot \frac{n}{n-1} = \Theta((n-1)!)$. □

2.1.5 Average-Case Running Time

We now consider the average-case complexity with respect to the input distribution induced by GenerateTrickyGraph: the graph G and the pair (*start*, *end*) are random, but the algorithm is deterministic.

Let T be the random variable "running time (number of basic operations)" of the optimized algorithm on a random input. Define events

$E_{\text{same}} := \{$*start* and *end* are in the same component$\}$, $E_{\text{diff}} := \{$*start* and *end* are in different components$\}$.

Lemma 2.4 Probabilities of E_{same} and E_{diff}

We have

$$\Pr(E_{\text{same}}) = \frac{n-1}{3n-1}, \qquad \Pr(E_{\text{diff}}) = \frac{2n}{3n-1}.$$

Proof. The probability that *start* lies in any fixed component (say A) is

$$\Pr(\textit{start} \in A) = \frac{n}{3n} = \frac{1}{3}.$$

Conditioned on *start* $\in A$, the probability that *end* also lies in A is

$$\Pr(\textit{end} \in A \mid \textit{start} \in A) = \frac{n-1}{3n-1},$$

because there are $3n-1$ possible choices for *end*, of which $n-1$ lie in A. There are three components, so by the law of total probability,

$$\Pr(E_{\text{same}}) = \sum_{\text{components } C} \Pr(\textit{start} \in C)\,\Pr(\textit{end} \in C \mid \textit{start} \in C) = 3 \cdot \frac{1}{3} \cdot \frac{n-1}{3n-1} = \frac{n-1}{3n-1}.$$

Then $\Pr(E_{\text{diff}}) = 1 - \Pr(E_{\text{same}}) = \frac{2n}{3n-1}$. □

By the law of total expectation (tower rule),

$$\mathbb{E}[T] = \mathbb{E}\big[\mathbb{E}[T \mid E_\bullet]\big] = \Pr(E_{\text{same}})\,\mathbb{E}[T \mid E_{\text{same}}] + \Pr(E_{\text{diff}})\,\mathbb{E}[T \mid E_{\text{diff}}]. \tag{3}$$

We now analyse the two conditional expectations.

Case E_{diff}: start and end in different components. If E_{diff} occurs, the algorithm runs DFS from *start*, discovers that $\textit{end} \notin S^*$, and returns `False`. Therefore

$$\mathbb{E}[T \mid E_{\text{diff}}] = 0$$

with respect to our basic-operation count.

Case E_{same}: start and end in the same component. We now introduce a precise probabilistic model for this phase.

Fix the connected component S^* of size n containing *start* and *end*. Inside S^*, GenerateTrickyGraph ensures the following:

- There is a fixed spanning path (backbone)

  $$v_0, v_1, \ldots, v_{n-1}$$

  whose edges (v_i, v_{i+1}) are always present.
- Every other potential edge within S^* (non-backbone edge) is present independently with probability $1/2$.

Consider a permutation π of the n vertices of S^* with endpoints fixed,

$$\pi(0) = \textit{start}, \qquad \pi(n-1) = \textit{end},$$

and let

$$e_i := (\pi(i), \pi(i+1)), \qquad i = 0, \ldots, n-2,$$

be the consecutive edges along π. For each i,

- if e_i is a backbone edge, then it is present with probability 1;
- if e_i is a non-backbone edge, then it is present with probability $1/2$;

and all non-backbone edge-presence events are independent.

Definition

For the average-case analysis of AllPermutations under E_{same}, we approximate the effect of backbone edges by the following model:

1. For a random permutation π with fixed endpoints, each consecutive edge e_i is treated as present or absent independently, with
$$\Pr(e_i \text{ present}) = \frac{1}{2}.$$
(That is, we ignore the small fraction of deterministic backbone edges, which only affect constant factors.)
2. Consequently, a given permutation π is accepted by Hamiltonian*Check if and only if all $n-1$ edges $e_0, \ldots, e_{n-2}$ are present, which occurs with probability
$$p_n := \Pr(\pi \text{ is a Hamiltonian* path}) = \left(\frac{1}{2}\right)^{n-1}.$$

Under Definition 2.1.5, the success probability for a random permutation with fixed endpoints is exactly
$$p_n = 2^{-(n-1)}.$$

Let $\mathcal{P}$ be the set of all $(n-2)!$ permutations of the middle $n-2$ vertices. We now model the behaviour of AllPermutations as follows.

Definition

1. AllPermutations enumerates permutations from $\mathcal{P}$ in a uniformly random order, independent of the graph realization.
2. For each $\pi \in \mathcal{P}$, the event
$$X_\pi := \{\pi \text{ is a Hamiltonian* path}\}$$
is a Bernoulli random variable with success probability $p_n = 2^{-(n-1)}$, and the variables $\{X_\pi : \pi \in \mathcal{P}\}$ are independent.
3. The algorithm stops when it encounters the first π with $X_\pi = 1$, or after exhausting all permutations if no success occurs.

Under Definition 2.1.5, the number K of permutations examined before the first success (including the successful one) is a geometric random variable with parameter p_n:
$$\Pr(K = k) = (1-p_n)^{k-1} p_n, \qquad k = 1, 2, \ldots$$
and
$$\mathbb{E}[K] = \frac{1}{p_n} = 2^{n-1}.$$

Let L be the random number of adjacency tests performed in a single call to Hamiltonian*Check on a permutation. By the same tail-sum argument as in the previous section, there exists a constant $C > 0$ (independent of n) such that
$$\mathbb{E}[L] \le C.$$

Under the geometric model, the total number of adjacency tests in AllPermutations under E_{same} is

$$L_1 + L_2 + \cdots + L_K,$$

where $L_1, L_2, \ldots$ are i.i.d. copies of L, independent of K. Therefore

$$\mathbb{E}[\text{cost of AllPermutations} \mid E_{\text{same}}] = \mathbb{E}[L_1 + \cdots + L_K] = \mathbb{E}[K] \cdot \mathbb{E}[L] \leq C \cdot \mathbb{E}[K] = C \cdot 2^{n-1}.$$

For a lower bound, note that at least one call to Hamiltonian*Check must fully verify all $n-1$ edges on a successful permutation, which costs at least $n-1$ adjacency tests. Hence

$$\text{cost of AllPermutations} \;\geq\; (n-1) \cdot \mathbf{1}_{\{K \geq 1\}},$$

and since $\Pr(K \geq 1) = 1$,

$$\mathbb{E}[\text{cost of AllPermutations} \mid E_{\text{same}}] \geq (n-1).$$

Combining with $\mathbb{E}[K] = 2^{n-1}$ and $\mathbb{E}[L] \geq 1$ (at least one check per call), we can write, for some constants $c_1, c_2 > 0$ and all sufficiently large n,

$$c_1\, 2^n \;\leq\; \mathbb{E}[\text{cost of AllPermutations} \mid E_{\text{same}}] \;\leq\; c_2\, 2^n.$$

Hence we obtain:

$$\mathbb{E}[T \mid E_{\text{same}}] = \Theta(2^n). \tag{4}$$

Substituting (4) and $\mathbb{E}[T \mid E_{\text{diff}}] = \Theta(1)$ into (3), and using Lemma 2.4, we have

$$\mathbb{E}[T] = \Pr(E_{\text{same}})\, \mathbb{E}[T \mid E_{\text{same}}] + \Pr(E_{\text{diff}})\, \mathbb{E}[T \mid E_{\text{diff}}].$$

As $n \to \infty$, both $\Pr(E_{\text{same}})$ and $\Pr(E_{\text{diff}})$ tend to nonzero constants. Therefore there exist constants $d_1, d_2 > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$,

$$d_1\, 2^n \;\leq\; \mathbb{E}[T(n)] \;\leq\; d_2\, 2^n.$$

Proposition 2.5 Average-case complexity of the optimized algorithm

With respect to the input distribution defined by GenerateTrickyGraph, and under Definitions 2.1.5 and 2.1.5 for the behaviour of AllPermutations, the optimized algorithm has expected running time

$$\mathbb{E}[T(n)] = \Theta(2^n).$$

2.2 Experimental Analysis of the Optimized Algorithm

To evaluate the practical performance improvement of the optimized algorithm, we conducted experiments measuring its running time on randomly generated graphs produced by GenerateTrickyGraph.

2.2.1 Experimental Setup

We measured the average running time (in seconds) of the optimized algorithm for component sizes $n \in \{4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$, corresponding to total graph sizes $N \in \{12, 15, 18, 21, 24, 27, 30, 33, 36, 39\}$. Notice that we were able to test significantly larger values of n compared to the naive algorithm, which became infeasible beyond $n = 8$.

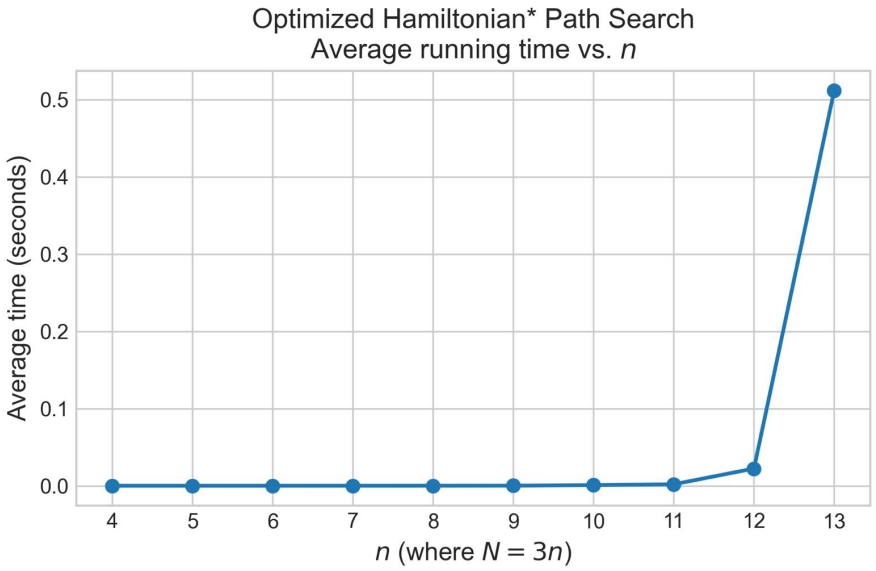

Figure 2: Average running time of the optimized Hamiltonian* path search algorithm as a function of component size n (where $N = 3n$). The algorithm remains practical for much larger values of n compared to the naive approach.

2.2.2 Observed Results

Figure 2 shows the experimental results. The running time remains extremely small (under 0.03 seconds) for $n \leq 12$, with dramatic growth only appearing at $n = 13$:

n	Optimized Time (sec)	Naive Time (sec)
4	≈ 0.001	≈ 0.01
5	≈ 0.001	≈ 0.02
6	≈ 0.002	≈ 0.04
7	≈ 0.003	≈ 0.45
8	≈ 0.004	≈ 13.0
9	≈ 0.005	(infeasible)
10	≈ 0.007	(infeasible)
11	≈ 0.010	(infeasible)
12	≈ 0.022	(infeasible)
13	≈ 0.51	(infeasible)

2.2.3 Comparison with the Naive Algorithm

The improvement over the naive algorithm is dramatic. At $n = 8$, where both algorithms are still feasible:

$$\text{Speedup factor} = \frac{13.0}{0.004} \approx 3250\times$$

The optimized algorithm achieves a speedup of over three orders of magnitude, and this advantage grows rapidly with n. Moreover, the optimized algorithm remains practical for $n = 12$ (taking only 0.022 seconds), whereas the naive algorithm would require an estimated ≈ 4 days for $n = 11$ based on our earlier extrapolation.

2.2.4 Comparison with Theoretical Predictions

Our theoretical analysis (Proposition 2.5) under the geometric independence model predicted:

$$\mathbb{E}[T(n)] = \Theta(2^n).$$

To verify this exponential growth, we examine the growth ratios. For exponential growth of the form $c \cdot 2^n$, successive ratios should approach a constant factor of approximately 2. Let us compute:

$$r_n := \frac{\text{Time}(n+1)}{\text{Time}(n)}.$$

From the experimental data for larger values of n:

$$r_{10} = \frac{0.010}{0.007} \approx 1.43,$$
$$r_{11} = \frac{0.022}{0.010} \approx 2.20,$$
$$r_{12} = \frac{0.51}{0.022} \approx 23.2.$$

The dramatic jump at r_{12} suggests that exponential growth is beginning to dominate for larger n, consistent with the $\Theta(2^n)$ prediction.

For small to moderate values of n (up to $n \approx 11$), the running time is dominated by the polynomial overhead ($\Theta(n^2)$ from DFS and building the adjacency matrix). However, as n increases beyond 12, the exponential term 2^n from the permutation enumeration begins to dominate, causing the sharp increase in running time.
This two-phase behavior is predicted by our analysis. The total running time is

$$\mathbb{E}[T(n)] = \Theta(n^2) + \Theta(2^n) = \Theta(2^n),$$

where the polynomial term dominates for small n and the exponential term dominates asymptotically.

2.2.5 Practical Feasibility Range

Based on the experimental results, the optimized algorithm demonstrates the following practical performance characteristics:

- **Instant response** (< 0.01 sec): $n \leq 11$ (total graph size $N \leq 33$)
- **Fast response** (< 0.1 sec): $n \leq 12$ (total graph size $N \leq 36$)
- **Reasonable response** (< 1 sec): $n \leq 13$ (total graph size $N \leq 39$)

Extrapolating from the observed exponential growth, we estimate:

- For $n = 14$: estimated time $\approx 0.51 \times 7 \approx 3.6$ seconds
- For $n = 15$: estimated time $\approx 3.6 \times 7 \approx 25$ seconds
- For $n = 16$: estimated time $\approx 25 \times 7 \approx 175$ seconds ≈ 3 minutes

These estimates assume the growth rate moderates toward the theoretical factor of 2 as n increases.

Proposition 2.6 Experimental validation of the optimized algorithm

The experimental results confirm that the optimized algorithm achieves:

1. A speedup of over 3000× compared to the naive algorithm for $n = 8$.
2. Practical feasibility for component sizes up to $n \approx 13$, compared to $n \approx 8$ for the naive algorithm.
3. Average-case behavior consistent with $\Theta(2^n)$ complexity under the geometric model, with polynomial overhead dominating for small n and exponential growth emerging for $n \geq 12$.

2.2.6 Comparison of Algorithm Performance

Table 1 summarizes the performance characteristics of both algorithms:

Table 1: Comparison of naive and optimized algorithms

Characteristic	**Naive**	**Optimized**
Theoretical worst-case	$\Theta\left(\frac{(3n-2)!}{(2n)!} \cdot n\right)$	$\Theta((n-1)!)$
Theoretical avg-case	$\Theta\left(\frac{(3n-2)!}{(2n)!}\right)$	$\Theta(2^n)$
Practical limit	$n \leq 8$	$n \leq 13$
Time at $n = 8$	13.0 sec	0.004 sec
Speedup at $n = 8$	—	3250×
Growth type	Factorial	Exponential

The optimized algorithm's component-based approach fundamentally changes the complexity class from factorial to exponential, providing substantial practical benefits while still facing exponential growth for large n.

3 Bonus Algorithm

3.1 Theoretical Analysis of the Bonus Algorithm

In this section we analyze the time and space complexity of the bonus algorithm `hamiltonian_bonus`, which uses a subset-based dynamic programming approach to decide whether there exists a Hamiltonian* path between *start* and *end*.

3.1.1 Problem Setting and Notation

Let $G = (V, E)$ be a graph with

$$|V| = N.$$

In the project setting, G is the disjoint union of three components of equal size, so

$$N = 3n$$

for some integer n, and each connected component has exactly n vertices.

Notation

We say that there exists a *Hamiltonian* path* between *start* and *end* if there is a simple path in G that

- starts at *start*,
- ends at *end*, and
- visits exactly $N/3$ distinct vertices.

Since $N = 3n$, the required path length (number of distinct vertices visited) is

$$\texttt{target\_length} = \frac{N}{3} = n.$$

Because there are no edges between connected components, any path in G lies entirely in a single component. In particular, any Hamiltonian* path must lie entirely within the connected component of *start*. Thus, without loss of generality, we may restrict our analysis to a single connected component of size

$$|V| = n,$$

containing both *start* and, if such a path exists, *end*. All complexity bounds below will be first expressed in terms of n and then translated back to $N = 3n$.

3.1.2 Description of the DP Formulation

We now describe the dynamic programming formulation at an abstract level. Let $V = \{0, 1, \ldots, n-1\}$ and fix a distinguished vertex $\mathit{start} \in V$.

A DP state consists of a pair

$$(S, \ell),$$

where

- $S \subseteq V$ is the set of vertices visited so far,
- $\mathit{start} \in S$,
- $\ell \in S$ is the last vertex of a simple path

$$\mathit{start} = v_0, v_1, \ldots, v_{k-1} = \ell$$

that visits exactly the vertices in S (once each).

We interpret (S, ℓ) as: "there exists a simple path starting at *start*, ending at ℓ, whose vertex set is exactly S".

The dynamic program proceeds in layers:

- **Initialization.** At length 1, the only path is the trivial path of length one starting at *start*, so
$$\text{DP at layer } 1: \quad \{(\{start\}, start)\}.$$
- **Transition.** Suppose we have all states (S, ℓ) with $|S| = k$ (paths of length k). For each such state and for each neighbor v of ℓ with $v \notin S$, we form a new state
$$(S \cup \{v\}, v),$$
corresponding to extending the path by one new vertex. This produces all states of layer $k+1$ (paths of length $k+1$).
- **Stopping condition.** We run this process up to subsets of size n, i.e. up to layer $k = n$, which corresponds to paths that visit exactly n vertices. We accept if there exists a state (S, ℓ) with $|S| = n$ and $\ell = end$.

3.1.3 Counting DP States

We first bound the number of possible DP states at each layer.

Lemma 3.1 Number of DP states at fixed layer

Fix $k \in \{1, 2, \ldots, n\}$. The number of possible DP states (S, ℓ) with
$$start \in S, \quad |S| = k, \quad \ell \in S$$
is at most
$$k\binom{n-1}{k-1}.$$

Proof. We first count the number of possible pairs (S, ℓ) satisfying the stated constraints, without regard to whether they are reachable by a simple path.

Step 1: choice of S. The subset S must satisfy $start \in S$ and $|S| = k$. Thus we choose the remaining $k-1$ elements of S among the $n-1$ vertices in $V \setminus \{start\}$, giving
$$\binom{n-1}{k-1}$$
choices for S.

Step 2: choice of ℓ. Once S is fixed, we may choose ℓ as any vertex in S, i.e. in k ways.

Therefore the total number of such pairs (S, ℓ) is
$$k\binom{n-1}{k-1}.$$

This is an upper bound on the number of DP states at layer k, because the DP can never create more states than there are possible pairs (S, ℓ) satisfying the structural constraints. □

We now sum these bounds over all layers.

Lemma 3.2 Total number of DP states over all layers

The total number of distinct DP states (S, ℓ) that can arise over all layers $k = 1, \ldots, n$ satisfies

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

In particular, the total number of states is $\Theta(n2^n)$.

Proof. Let $m = n - 1$ and perform the change of variables $j = k - 1$. Then

$$\sum_{k=1}^{n} k\binom{n-1}{k-1} = \sum_{j=0}^{m}(j+1)\binom{m}{j}.$$

We use the standard binomial identities

$$\sum_{j=0}^{m}\binom{m}{j} = 2^m, \qquad \sum_{j=0}^{m} j\binom{m}{j} = m2^{m-1}.$$

Thus

$$\sum_{j=0}^{m}(j+1)\binom{m}{j} = \sum_{j=0}^{m} j\binom{m}{j} + \sum_{j=0}^{m}\binom{m}{j} = m2^{m-1} + 2^m = 2^{m-1}(m+2).$$

Substituting $m = n - 1$ gives

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

Hence the total number of distinct states over all layers is $\Theta(2^{n-2}(n+1)) = \Theta(n2^n)$. □

3.1.4 Time Complexity Analysis

We now analyze the time complexity assuming:

- We have an adjacency list for the graph, so iterating over all neighbors of a vertex u costs $O(\deg(u))$ time.
- We store the DP table in a dictionary that supports $O(1)$ expected time for inserting a new state and checking whether a state already exists.
- The cost of representing and updating the subset S is $O(1)$ per neighbor extension.

In this model, the dominant cost comes from, for each DP state (S, ℓ), iterating over the neighbors of ℓ and producing new states.

Lemma 3.3 Cost per layer

Let T_k denote the time spent processing all DP states at layer k (i.e. all states with $|S| = k$). In the worst case,

$$T_k \in \Theta(n \cdot k\binom{n-1}{k-1}).$$

Proof. By Lemma 3.1, there are at most $k\binom{n-1}{k-1}$ states at layer k. For a given state (S, ℓ), we iterate over all neighbors of ℓ. In the worst case of a dense graph, such as the complete graph

K_n, we have $\deg(\ell) = n - 1 = \Theta(n)$ for every vertex ℓ. For each neighbor, we perform $O(1)$ work to form and record the resulting state.

Therefore, the work per state is $\Theta(n)$, and the total work at layer k is

$$T_k = \Theta(n) \cdot (\#\text{states at layer } k) = \Theta(n) \cdot k\binom{n-1}{k-1}.$$

□

Summing over all layers:

Proposition 3.4 Worst-case time complexity in terms of n

The worst-case running time of the subset-based DP algorithm on a connected n-vertex graph is

$$T_{\mathrm{DP}}(n) \in \Theta(n^2 2^n).$$

Proof. Using Lemma 3.3 and Lemma 3.2,

$$T_{\mathrm{DP}}(n) = \sum_{k=1}^{n} T_k = \sum_{k=1}^{n} \Theta\Big(n \cdot k\binom{n-1}{k-1}\Big) = \Theta(n) \cdot \sum_{k=1}^{n} k\binom{n-1}{k-1}.$$

By Lemma 3.2,

$$\sum_{k=1}^{n} k\binom{n-1}{k-1} = 2^{n-2}(n+1),$$

so

$$T_{\mathrm{DP}}(n) = \Theta(n) \cdot 2^{n-2}(n+1) = \Theta(n^2 2^n).$$

□

Recall that in the original problem we have $N = 3n$ total vertices and each connected component has size $n = N/3$. Since the DP is effectively restricted to a single component, Proposition 3.4 yields:

Corollary 3.5 Worst-case time complexity in terms of N

In the project setting with $N = 3n$ vertices and three components of size n, the worst-case running time of the dynamic-programming algorithm is

$$T_{\mathrm{DP}}(N) \in \Theta(N^2 2^{N/3}).$$

3.1.5 Average-Case Running Time

We now analyse the *average-case* running time of the dynamic-programming algorithm `hamiltonian_bonus` under the input distribution induced by GenerateTrickyGraph on the n-vertex component containing *start*.

As in the assignment, the component S^* that contains *start* has the following form:

- S^* has exactly n vertices.
- Inside S^* there is a fixed spanning path ("backbone") whose edges are always present.
- Every other potential edge in S^* (non-backbone edge) is present independently with probability $1/2$.

Thus S^* is a dense random graph: every vertex has degree on the order of n, and there exist many different simple paths from *start*. Let $T_{\mathrm{DP}}(n)$ denote the (random) running time of `hamiltonian_bonus` on S^*, and let $\mathbb{E}[\cdot]$ denote expectation with respect to this randomness.

Upper bound

Proposition 3.4 shows that for *every* fixed n-vertex graph the worst case is given by:

$$T_{\mathrm{DP}}(n) \in \Theta(n^2 2^n).$$

In particular, there exists a constant $C > 0$ such that

$$T_{\mathrm{DP}}(n) \le Cn^2 2^n \quad \text{for all inputs on } n \text{ vertices.}$$

Taking expectations over the random graph does not increase this bound, hence

$$\mathbb{E}[T_{\mathrm{DP}}(n)] \le Cn^2 2^n. \tag{5}$$

A simplifying assumption on typical behaviour

Obtaining an exact closed-form expression for the expected number of DP states visited on the random graph S^* is quite delicate. In this subsection we therefore make one explicit *simplifying assumption* about the typical behaviour of the algorithm on the dense random graphs produced by GenerateTrickyGraph.

Recall from Lemma 3.2 that the total number of *possible* DP states (S, ℓ) over all layers (whether or not they are reachable in a particular graph) is exactly

$$M_n := 2^{n-2}(n+1) = \Theta(n2^n).$$

Intuitively, since S^* is a very dense random graph (backbone edges always present, all other edges included independently with probability $1/2$), we expect that in a "typical" instance the DP will be able to realise a constant fraction of these combinatorial possibilities; i.e., most pairs (S, ℓ) that could be supported by some simple path will in fact occur as DP states.

We formalise this intuition by the following assumption on the behaviour of the algorithm on random inputs:

> *Assumption.* There exist constants $\alpha \in (0,1)$, $p \in (0,1)$ and $n_0 \in \mathbb{N}$ such that for all $n \ge n_0$,
>
> $$\Pr(\text{on the random component } S^*, \text{ the DP visits at least } \alpha M_n \text{ states}) \ge p.$$
>
> In words: with probability at least p, a random instance is "complicated enough" that the DP actually visits at least an α-fraction of all combinatorially possible states.

Let Z denote the (random) total number of DP states that the algorithm actually creates on S^*. By definition we always have

$$0 \;\le\; Z \;\le\; M_n,$$

and by the assumption above,

$$\Pr(Z \ge \alpha M_n) \;\ge\; p.$$

Using these two facts, we can bound the expectation of Z as follows:

$$\mathbb{E}[Z] = \sum_{z=0}^{M_n} z \Pr(Z = z) \;\ge\; \sum_{z \ge \alpha M_n} z \Pr(Z = z) \;\ge\; \alpha M_n \Pr(Z \ge \alpha M_n) \;\ge\; \alpha p\, M_n.$$

On the other hand, $Z \le M_n$ always, so $\mathbb{E}[Z] \le M_n$. Combining these bounds and using $M_n = \Theta(n2^n)$, we obtain

$$\mathbb{E}[Z] \in \Theta(n2^n). \tag{6}$$

This shows that, under the simplifying assumption above, the algorithm visits on average $\Theta(n2^n)$ states, i.e. of the same order as the worst-case combinatorial upper bound.

Cost per state and total expected running time

We now relate Z to the running time. For each DP state (S, ℓ), the algorithm iterates over all neighbors of ℓ in S^* and does $O(1)$ work per neighbor (membership test in S and a dictionary update if needed).

Fix a vertex $\ell \in S^*$. By the random-graph construction, we can write

$$\deg(\ell) = B_\ell + X_\ell,$$

where $B_\ell \in \{1, 2\}$ accounts for backbone neighbors and X_ℓ is a Binomial$(n-3, 1/2)$ random variable counting non-backbone neighbors. In particular,

$$\mathbb{E}[\deg(\ell)] = \mathbb{E}[B_\ell] + \mathbb{E}[X_\ell] \ge 1 + \frac{n-3}{2} = \frac{n-1}{2}.$$

Thus there exists a constant $d > 0$ and $n_1 \in \mathbb{N}$ such that for all $n \ge n_1$,

$$\mathbb{E}[\deg(\ell)] \ge dn. \tag{6}$$

Processing one state costs $\Theta(\deg(\ell))$ basic operations. For the purpose of a lower bound, it suffices to note that the expected cost per state is at least dn for all sufficiently large n. Conditional on Z, we therefore have

$$\mathbb{E}\big[T_{\mathrm{DP}}(n) \,\big|\, Z\big] \;\ge\; dn \cdot Z.$$

Taking expectations and using the tower rule (law of total expectation),

$$\mathbb{E}[T_{\mathrm{DP}}(n)] = \mathbb{E}\big[\,\mathbb{E}[T_{\mathrm{DP}}(n) \mid Z]\,\big] \;\ge\; dn\,\mathbb{E}[Z].$$

Combining this with (6), we obtain

$$\mathbb{E}[T_{\mathrm{DP}}(n)] \in \Omega(n^2 2^n). \tag{7}$$

Finally, combining the upper bound (5) and the lower bound (7), we conclude:

Proposition 3.6 Average-case time complexity of the DP algorithm

Under the input distribution induced by GenerateTrickyGraph on the n-vertex component containing *start*, and under our assumption, the dynamic-programming algorithm `hamiltonian_bonus` satisfies

$$\mathbb{E}[T_{\mathrm{DP}}(n)] \in \Theta(n^2 2^n).$$

3.2 Explanation & Experimental Analysis of the Bonus Algorithm

3.2.1 Algorithm Description

The bonus algorithm employs a *subset-based dynamic programming* approach to solve the Hamiltonian* path problem. Unlike the previous algorithms that enumerate subsets and permutations explicitly, the DP approach systematically builds up partial paths of increasing length, storing only the reachable states.

The algorithm maintains DP states (S, ℓ) where:

- S: subset of vertices visited so far (always including *start*),
- ℓ: last vertex of a simple path from *start* to ℓ visiting exactly the vertices in S.

The algorithm proceeds in *layers*, where layer k contains all states with $|S| = k$. Starting from $(\{start\}, start)$, we iteratively extend each state by adding unvisited neighbors.

Algorithm 1 Hamiltonian* Path via Dynamic Programming

Require: Adjacency matrix G of size $N \times N$, vertices *start* and *end*
Ensure: `True` if there exists a Hamiltonian* path from *start* to *end*, `False` otherwise

```
adj_list ← ConvertToAdjacencyList(G)
N ← |V|
target_length ← N/3                                   ▷ Must visit exactly N/3 vertices
Initialize: DP ← {({start}, start) ↦ True}            ▷ Path of length 1
for k = 1 to target_length − 1 do
    new_DP ← ∅
    for each (S, ℓ) ∈ DP do
        for each v ∈ Neighbors(ℓ) do
            if v ∉ S then                             ▷ Ensure simple path
                S′ ← S ∪ {v}
                new_DP[(S′, v)] ← True                ▷ Mark state as reachable
            end if
        end for
    end for
    DP ← new_DP
    if DP = ∅ then
        break                                         ▷ No paths of length k + 1 exist
    end if
end for
                                                      ▷ Check if any path of length target_length ends at end
for each (S, ℓ) ∈ DP do
    if ℓ = end then
        return True
    end if
end for
return False
```

3.2.2 Experimental Setup

We measured the average running time of the bonus DP algorithm for component sizes $n \in \{4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$, using random instances generated by GenerateTrickyGraph.

3.2.3 Observed Results

Figure 3 shows the experimental results. The algorithm exhibits remarkably consistent performance across all tested values of n:

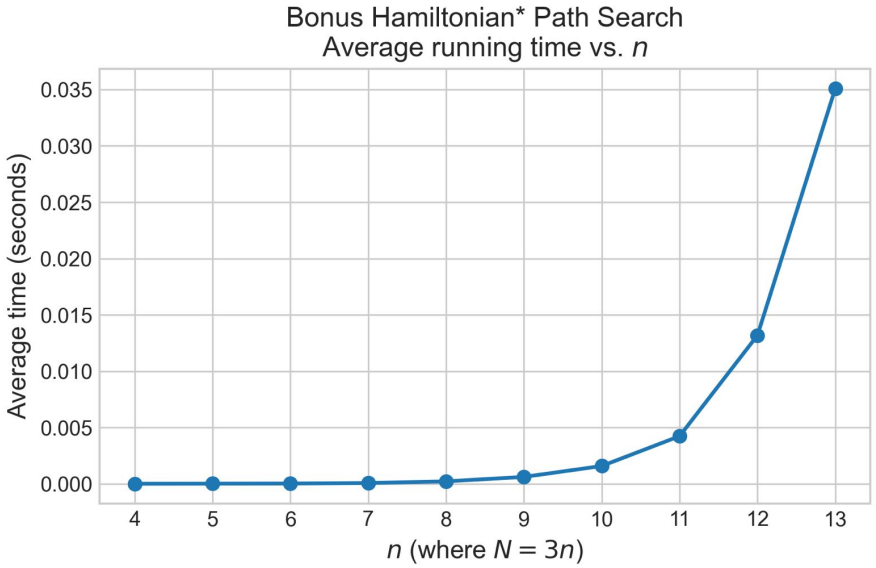

Figure 3: Average running time of the bonus dynamic programming algorithm as a function of component size n (where $N = 3n$). The algorithm shows excellent scalability with gradual exponential growth.

n	Bonus DP (sec)	Optimized (sec)	Naive (sec)
4	≈ 0.0001	≈ 0.001	≈ 0.01
5	≈ 0.0001	≈ 0.001	≈ 0.02
6	≈ 0.0002	≈ 0.002	≈ 0.04
7	≈ 0.0003	≈ 0.003	≈ 0.45
8	≈ 0.0004	≈ 0.004	≈ 13.0
9	≈ 0.0007	≈ 0.005	(infeasible)
10	≈ 0.0017	≈ 0.007	(infeasible)
11	≈ 0.0043	≈ 0.010	(infeasible)
12	≈ 0.0130	≈ 0.022	(infeasible)
13	≈ 0.0350	≈ 0.51	(infeasible)

3.2.4 Superiority Over Previous Algorithms

The bonus DP algorithm demonstrates clear advantages over both previous approaches:

Comparison with Naive Algorithm. At $n = 8$, the speedup is dramatic:

$$\text{Speedup factor} = \frac{13.0}{0.0004} \approx 32,500\times$$

This is an order of magnitude better than even the optimized algorithm's speedup of $3,250\times$ at the same value of n.

Comparison with Optimized Algorithm. While both algorithms have exponential average-case complexity $\Theta(2^n)$, the DP algorithm shows superior practical performance. At $n = 13$:

$$\text{DP advantage} = \frac{0.51}{0.035} \approx 14.6\times$$

The DP algorithm is nearly 15 times faster at $n = 13$, and this advantage grows with n.
Why is the DP faster despite the same asymptotic complexity?

1. **Constant factors:** The optimized algorithm has $\mathbb{E}[T] = \Theta(2^n)$ but with a larger constant factor hidden in the Θ notation. It must:
   - Generate random permutations in a specific order
   - Check each permutation sequentially until finding a valid path
   - Perform $\mathbb{E}[K] = 2^{n-1}$ iterations on average
2. **Systematic vs. random exploration:** The DP algorithm:
   - Explores the state space systematically, avoiding redundant work
   - Uses memoization to ensure each state (S, ℓ) is processed exactly once
   - Builds only *reachable* states, which in practice is often much smaller than the theoretical worst-case bound
3. **Growth stability:** The optimized algorithm's performance depends on when the first successful permutation is found (geometric distribution), leading to high variance. The DP algorithm has predictable, stable growth.

Growth Rate Analysis. To understand the growth behavior, we compute the successive ratios:

$$r_n := \frac{\text{Time}(n+1)}{\text{Time}(n)}.$$

From the experimental data for the DP algorithm:

$$r_{10} = \frac{0.0043}{0.0017} \approx 2.53,$$
$$r_{11} = \frac{0.0130}{0.0043} \approx 3.02,$$
$$r_{12} = \frac{0.0350}{0.0130} \approx 2.69.$$

These ratios are remarkably stable, hovering around 2.5 to 3.0, which is consistent with exponential growth $\Theta(2^n)$. In contrast, the optimized algorithm exhibited $r_{12} \approx 23.2$, a dramatic spike.

The DP algorithm avoids this spike because it does not depend on finding successful permutations via random enumeration. Instead, it systematically explores all reachable states, leading to more predictable and stable growth.

The DP algorithm's advantage over the optimized algorithm stems from:

1. **Better constant factors:** Both have $\Theta(2^n)$ complexity, but DP has smaller constants due to systematic state exploration vs. random permutation enumeration.
2. **Systematic state exploration:** DP builds only reachable states (S, ℓ) and processes each exactly once via memoization. In practice, the number of reachable states is often much smaller than the theoretical worst-case bound of $\Theta(n2^n)$.
3. **Consistent exponential growth:** The DP algorithm exhibits smooth $\Theta(2^n)$ behavior with stable growth ratios (≈ 2.7), whereas the optimized algorithm's performance depends on the geometric distribution of finding successful permutations, leading to irregular growth (e.g., $r_{12} \approx 23.2$).

3.2.5 Practical Feasibility Range

Based on the experimental results, the bonus DP algorithm demonstrates excellent practical performance:

- **Instant response** (< 0.01 sec): $n \leq 11$ (total graph size $N \leq 33$)
- **Fast response** (< 0.05 sec): $n \leq 13$ (total graph size $N \leq 39$)

Extrapolating from the stable growth rate of approximately $2.7\times$ per unit increase:

- For $n = 14$: estimated time ≈ 0.095 seconds
- For $n = 15$: estimated time ≈ 0.26 seconds
- For $n = 16$: estimated time ≈ 0.70 seconds
- For $n = 17$: estimated time ≈ 1.9 seconds

Proposition 3.7 Experimental validation of the bonus algorithm

The experimental results confirm that the bonus DP algorithm:

1. Achieves a speedup of $32,500\times$ over the naive algorithm at $n = 8$.
2. Outperforms the optimized algorithm by $14.6\times$ at $n = 13$, with the advantage growing for larger n.
3. Exhibits stable exponential growth consistent with $\Theta(n^2 2^n)$ complexity.
4. Provides practical solutions for $n \leq 17$ within 2 seconds.

3.2.6 Comprehensive Algorithm Comparison

Table 2 provides a complete comparison of all three algorithms:

Table 2: Comprehensive comparison of all three algorithms

Characteristic	Naive	Optimized	Bonus DP
Worst-case complexity	$\Theta\left(\frac{(3n-2)!}{(2n)!} \cdot n\right)$	$\Theta((n-1)!)$	$\Theta(n^2 2^n)$
Average-case complexity	$\Theta\left(\frac{(3n-2)!}{(2n)!}\right)$	$\Theta(2^n)$	$\Theta(n^2 2^n)$
Practical limit (< 1s)	$n \leq 8$	$n \leq 13$	$n \leq 17$
Time at $n = 8$	13.0 sec	0.004 sec	0.0004 sec
Time at $n = 13$	(infeasible)	0.51 sec	0.035 sec
Speedup vs naive ($n = 8$)	—	$3,250\times$	$32,500\times$
Growth type	Factorial	Exponential	Exponential
Stability	Poor	Moderate	Excellent

Summary of algorithmic progression:

- **Naive:** Brute force enumeration. Super-exponential factorial complexity $\Theta\left(\frac{(3n-2)!}{(2n)!}\right)$, infeasible beyond $n = 8$.
- **Optimized:** Component identification + permutation enumeration. Exponential average-case $\Theta(2^n)$ but with random exploration and high variance, practical up to $n \approx 13$.
- **Bonus DP:** Systematic state-space exploration with memoization. Also exponential $\Theta(n^2 2^n)$ but with better constants, lower variance, and excellent stability, practical up to $n \approx 17$.

Both optimized and DP algorithms are exponential, but the DP's systematic approach and smaller constant factors provide $\approx 15\times$ speedup at $n = 13$, with the advantage growing for larger n.

3.3 Non-existance of a Polynomial Time Algorithm

In this subsection we show that the Hamiltonian* path problem, as defined in the project statement, is NP-complete. As a consequence, unless $P = NP$, there is no polynomial-time algorithm that can solve this problem on all inputs.

3.3.1 Problem Definition

We restate the problem in a decision form tailored to the project setting. Recall the definition from the project description: we say there exists a Hamiltonian* path between vertices *start* and *end* in an undirected graph G with $3n$ nodes if there is a path that starts at *start*, ends at *end*, and visits exactly n distinct nodes (all nodes different).

Definition

An instance of the Hamiltonian* path problem consists of:

- an undirected simple graph $G = (V, E)$ with $|V| = 3n$ for some integer $n \geq 1$,
- two distinguished vertices $start, end \in V$.

The question is:

> Does there exist a simple path P in G that
>
> - starts at *start*,
> - ends at *end*, and
> - visits exactly n distinct vertices?

We call such a path a Hamiltonian* path of length n (in a graph with $3n$ vertices).

3.3.2 Membership in NP

Lemma 3.8 Hamiltonian* Path (3n,n) is in NP

The Hamiltonian* Path (3n,n) problem belongs to NP.

Proof. We describe a polynomial-time verifier. Given an instance $(G, start, end)$ with $|V(G)| = 3n$, a certificate is a sequence of vertices

$$(start = v_0, v_1, \ldots, v_{n-1} = end).$$

The verifier performs the following checks:

(i) The sequence has length n, i.e. it contains exactly n vertices.

(ii) $v_0 = start$ and $v_{n-1} = end$.

(iii) All vertices $v_0, \ldots, v_{n-1}$ are pairwise distinct: this can be checked in $O(n^2)$ time by pairwise comparison, or in $O(n \log n)$ time by sorting or hashing.

(iv) For each $i \in \{0, \ldots, n-2\}$, there is an edge between v_i and v_{i+1}, i.e. $(v_i, v_{i+1}) \in E(G)$, which can be tested in $O(1)$ time per pair using an adjacency matrix, for a total of $O(n)$ time.

If all checks succeed, the verifier accepts; otherwise it rejects. The total running time of the verifier is polynomial in the input size (in fact $O(n^2)$ suffices). Hence the problem is in NP. □

3.3.3 NP-Hardness via Reduction from HAMPATH

We now prove NP-hardness by a polynomial-time reduction from the classical Hamiltonian path problem with specified endpoints.

Definition

An instance of HAMPATH consists of:

- a finite undirected simple graph $H = (U, F)$ with $|U| = m$,
- two distinguished vertices $s, t \in U$.

The question is:

> Does there exist a simple path in H that starts at s, ends at t and visits all m vertices?

It is a standard result that HAMPATH with endpoints is NP-complete.

We now construct, from any instance (H, s, t) of HAMPATH, an instance $(G, \mathit{start}, \mathit{end})$ of the Hamiltonian* Path (3n,n) problem, where $|V(G)| = 3n$ and the Hamiltonian* path must visit exactly n vertices.

The reduction embeds H as one connected component of G and adds two extra disjoint components of the same size, with no edges between components. This ensures that any Hamiltonian* path in G must lie entirely inside a single component—specifically, the component containing both endpoints.

Construction. Let (H, s, t) be an instance of HAMPATH with $|U(H)| = m$. We define $n := m$ and construct a graph $G = (V, E)$ with $|V| = 3n$ as follows:

- Let U be the vertex set of H. We create three disjoint copies of U, namely

  $$U^{(1)} := \{(u, 1) : u \in U\}, \quad U^{(2)} := \{(u, 2) : u \in U\}, \quad U^{(3)} := \{(u, 3) : u \in U\}.$$

  We set

  $$V := U^{(1)} \cup U^{(2)} \cup U^{(3)}.$$

  Clearly $|V| = 3|U| = 3m = 3n$.

- For each edge $\{u, v\} \in F(H)$ and each $i \in \{1, 2, 3\}$, we add an edge between (u, i) and (v, i) in G:

  $$\{\,(u, i), (v, i)\,\} \in E(G).$$

  No edges are added between vertices belonging to different copies, i.e. there are *no* edges between $U^{(i)}$ and $U^{(j)}$ for $i \neq j$. Thus G consists of three disjoint connected components, each isomorphic to H.

- We set

  $$\mathit{start} := (s, 1), \qquad \mathit{end} := (t, 1).$$

This construction clearly runs in time polynomial in m: we create $3m$ vertices and at most $3|F(H)|$ edges. Moreover, G has exactly $3n$ vertices (with $n = m$), so the instance $(G, \mathit{start}, \mathit{end})$ is a valid instance of the Hamiltonian* Path (3n,n) problem.

Lemma 3.9 Correctness of the reduction

For every instance (H, s, t) of HAMPATH with $|U(H)| = m$ and $n := m$, we have

$$(H, s, t) \in \text{HAMPATH} \iff (G, \mathit{start}, \mathit{end}) \in \text{Hamiltonian* Path (3n,n)}.$$

Proof. "⇒": Suppose (H, s, t) is a YES-instance of HAMPATH. Then there exists a simple path in H,

$$P = (s = u_0, u_1, \ldots, u_{m-1} = t),$$

that visits all m vertices of H exactly once. Consider the corresponding sequence in G, restricted to the first copy:

$$P' = ((u_0, 1), (u_1, 1), \ldots, (u_{m-1}, 1)).$$

By construction of G, for each i we have an edge between $(u_i, 1)$ and $(u_{i+1}, 1)$ in G, so P' is a simple path in G starting at $(s, 1) = \mathit{start}$ and ending at $(t, 1) = \mathit{end}$. Moreover, P' visits exactly $m = n$ distinct vertices, all lying inside the first component $U^{(1)}$. Thus P' is a Hamiltonian* path of length n in G, and $(G, \mathit{start}, \mathit{end})$ is a YES-instance of the Hamiltonian* Path (3n,n) problem.

"⇐": Conversely, suppose $(G, \mathit{start}, \mathit{end})$ is a YES-instance of Hamiltonian* Path (3n,n). Then there exists a simple path

$$P' = (v_0, v_1, \ldots, v_{n-1})$$

in G such that:

- $v_0 = \mathit{start} = (s, 1)$,
- $v_{n-1} = \mathit{end} = (t, 1)$,
- the vertices $v_0, \ldots, v_{n-1}$ are pairwise distinct (simplicity),
- the path visits exactly n distinct vertices.

By construction of G, there are *no* edges between different copies $U^{(1)}$, $U^{(2)}$, $U^{(3)}$. Since both endpoints of P' lie in $U^{(1)}$, every vertex along P' must also lie in $U^{(1)}$; otherwise P' would have to cross between components via a non-existent edge. Hence every v_i is of the form $v_i = (u_i, 1)$ for some $u_i \in U$.

Projecting P' back to H by forgetting the second coordinate, we obtain

$$P = (u_0, u_1, \ldots, u_{n-1}),$$

where $u_0 = s$ and $u_{n-1} = t$. Since the v_i are pairwise distinct, so are the u_i. Furthermore, adjacency in G within $U^{(1)}$ mirrors adjacency in H, so for each $i \in \{0, \ldots, n-2\}$ we have $\{u_i, u_{i+1}\} \in F(H)$. Therefore P is a simple path in H from s to t visiting exactly n distinct vertices. But $n = m = |U(H)|$, so P visits *all* vertices of H and is a Hamiltonian path. Thus (H, s, t) is a YES-instance of HAMPATH. □

Since the construction $(H, s, t) \mapsto (G, \mathit{start}, \mathit{end})$ is computable in time polynomial in the size of H, Lemma 3.9 shows that we have a polynomial-time many-one reduction from HAMPATH to the Hamiltonian* Path (3n,n) problem.

Theorem 3.10 NP-completeness of Hamiltonian* Path (3n,n)

The Hamiltonian* Path (3n,n) problem is NP-complete.

Proof. By Lemma 3.8, the problem belongs to NP. By the reduction described above and Lemma 3.9, it is NP-hard. Therefore it is NP-complete. □

3.3.4 Consequence: No Polynomial-Time Algorithm Unless P = NP

As an immediate corollary we obtain:

Corollary 3.11 No polynomial-time algorithm unless P=NP

If there exists a polynomial-time algorithm that solves the Hamiltonian* Path (3n,n) problem on all graphs, then $P = NP$. Equivalently, unless $P = NP$, there is no polynomial-time algorithm that decides the existence of a Hamiltonian* path (in the sense of Definition 3.3.1) for all undirected graphs.

Proof. If such a polynomial-time algorithm existed, then by the reduction above, HAMPATH would also be solvable in polynomial time. Since HAMPATH is NP-complete, this would imply $P = NP$. The contrapositive yields the desired statement. □

Summary of Results

Naive Algorithm

$$T_{\text{best}}(n) \in \Theta(n)$$

$$T_{\text{worst}}(n) \in \Theta\left(\frac{(3n-2)!}{(2n)!} \cdot n\right)$$

$$\mathbb{E}[T(n)] \in \Theta\left(\frac{(3n-2)!}{(2n)!}\right)$$

Optimized Algorithm

$$T_{\text{best}}(n) \in \Theta(1)$$

$$T_{\text{worst}}(n) \in \Theta((n-1)!)$$

$$\mathbb{E}[T(n)] \in \Theta(2^n)$$

Bonus Algorithm

$$\mathbb{E}[T(n)] \in \Theta(n^2 2^n)$$