# Time and Space Efficient Algorithms for Hamiltonian Cycles

Jonathan N, Euan M.

November 14, 2022

# Contents


**1 Abstract** **3**

**2 Introduction** **3**
- 2.1 Hamiltonian Paths and Hamiltonian Cycles Definition . . . 3
- 2.2 Mathematical Definition . . . 3
- 2.3 The Hamiltonian Cycle Problem . . . 3
- 2.4 Overview and Restrictions . . . 4

**3 Algorithms** **5**
- 3.1 Backtracking 1: Brute Force DFS . . . 5
- 3.2 Backtracking 2: Eppsteins Algorithm . . . 10
- 3.3 Dynamic Programming: Held-Karps Algorithm . . . 11

**4 Theoretical Complexity** **12**
- 4.1 Backtracking 1: Brute Force DFS . . . 12
- 4.2 Backtracking 2: Eppsteins Algorithm . . . 12
- 4.3 Dynamic Algorithm: Held-Karp . . . 12

**References** **13**

# 1 Abstract

*Hamiltonian cycles* are a very interesting problem lining all undergraduate discrete mathematics textbooks. At a higher level, we understand them as complex to solve. Here we look at the problem and discuss intricate details that have allowed the development of relatively efficient but still exponential algorithms for solving Hamiltonian Cycles.

# 2 Introduction

The *Hamiltonian Cycle Problem* is a problem that has been known for a very long time to be hard. Hamiltonian Cycles are often used to study the fundamental theories and understanding of complexity theory. Here we assume graphs are undirected, the conversion from undirected to directed graphs is not difficult however.

## 2.1 Hamiltonian Paths and Hamiltonian Cycles Definition

The hamiltonian cycle is similar to that of the hamiltonian path where inside an undirected or directed graph, the hamiltonian path, also known as the traceable path, is a path that visits each vertex exactly once. However, differing to the hamiltonian path, the starting point and ending point of the path must also be adjacent to each other such that they are able to create a cycle using an available edge.

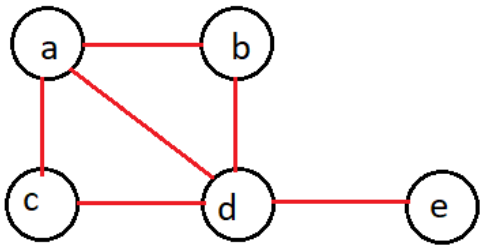



Figure 1.

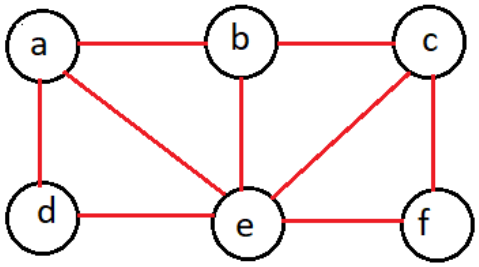



Figure 2.

In figure 1, there are multiple paths inside the undirected graph that result in a hamiltonian path, but due to the requirement of needing to start with node $e$, there is no hamiltonian cycle since there is no edge that accomplishes the requirement of the starting node and ending node of the path to connect.

Figure 2 shows an example of an undirected graph that includes a hamiltonian cycle as, starting with any node, such as $a$, allows for a path such as $abcfeda$ or $adefcba$, fulfilling the criteria of connecting the first and last nodes.

## 2.2 Mathematical Definition

**Definition 1** (Graph). *A graph $G$ is a tuple with a set of verticies $V$ and edges $E$, with an edge connecting two verticies. Notated as $G = (V, E)$*

**Definition 2** (Path for Undirected Graphs [Ros07a]). *Let $n$ be a nonnegative integer and $G$ an undirected graph. A path of length $n$ from $u$ to $v$ in $G$ is a sequence of $n$ edges $e_1, ..., e_n$ of $G$ for which there exists a sequence of $x_0 = u$, $x_1, ..., x_{n-1}$, $x_n = v$ of vertices such that $e_i$ has for $i = 1, ..., n$, the endpoints $x_{i-1}$ and $x_i$.*

**Definition 3** (A Cycle [Ros07a]). *A graph has a cycle if it has a path where $u = v$.*

**Definition 4** (A Simple Cycle or Path [Ros07a]). *A path or cycle is simple if all its edges appear in the path only once.*

**Definition 5** (Hamiltonian Cycle [Ros07b]). *A Hamiltonian Cycle is a simple cycle that visits all the vertices in a graph.*

**Definition 6** (Forced Edge [Epp07]). *A edge is forced if it must appear in the Hamiltonian Cycle.*

## 2.3 The Hamiltonian Cycle Problem

There are many variations of the Hamiltonian Cycle Problem.

The decision problem Hamilton Cycle Problem would be to find if a hamiltonian cycle exists or not. The decision problem can be extended to what is generally understood as the hamiltonian path problem by returning the path if it exists. The decision problem is NP-complete. Another variation of the Hamiltonian Cycle Problem is to find how many hamiltonian cycles exist in a graph.

Not all graphs have equivalent complexity. Some graphs are relatively trivial to solve in exact exponential time with very small, near polynomial bases. This property makes Hamiltonian Cycles an interesting problem to research when looking at a search space for algorithms that solve the hamiltonian cycle problem. Unrelated to our implementations in this report, it has been hypothesised that the hardest graphs to solve lie close to the bound $\frac{1}{2}v \cdot \ln(v) + \frac{1}{2}\ln(\ln(v))$ [KS06]. It also has been shown through empirical methods [SB21] that there exists graphs that are difficult to solve far away from this bound.

The *Travelling Salesman Problem* is reducible to the Hamiltonian Cycle Problem by setting the weight of the graph to 1. As a direct result, many of the algorithms used to solve the hamiltonian cycle problem were originally intended as a solution to the travelling salesman problem.

## 2.4 Overview and Restrictions

Hamiltonian Cycles have been known for a long time to be solvable through a brute-force depth-first search. At the same time, this yields less efficient time complexity. Bellman[Bel62], Held and Karp[HK62] developed a dynamic programming algorithm for solving travelling salesman in exact exponential time that applies to the Hamiltonian Cycle Problem. It was shown after that not all graphs have equal complexity, and by restricting a graph to degree 3 [Epp07], there exists an exact exponential bound with a near polynomial base, as long as the graph was restricted to having at most vertices with degree 3.

# 3 Algorithms

## 3.1 Backtracking 1: Brute Force DFS

The backtracking algorithm is a brute-force approach to problem solving and involves backtracking if the current solution is not suitable. That is, the algorithm attempts to recursively build a solution, one step at a time, and if the resulting solution fails, another solution will be found by trying another solution and/or moving one step back, eventually finding every single solution to the problem.

In regards to the hamiltonian cycle problem:

**Step 0:**

We first create an adjacency matrix to the undirected graph to make finding the hamiltonian cycle easier. This involves writing 1 where there is a connection between the next vertex, and 0 where there is none.

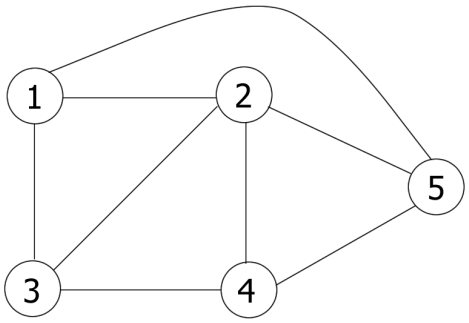



Figure 1: Example Graph 1. Undirected Graph

Table 1: Example Adjacency Matrix

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

**Step 1:**

We will first create an empty array to store the hamiltonian path.

Table 2: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | | | | | |

**Step 2:**

We will then fill in the first node in the graph.

Table 3: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 1 | | | | |

This will result in the following path.


**Step 3:**

We will then attempt to increment the next node into the array.

Table 4: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 1 | 1 | | | |

However, since 1 is already in the path, the value will be incremented again.

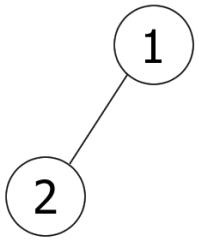
Table 5: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 1 | 2 | | | |

This new value will then be compared against the adjacency matrix to check if the previous node (node 1) is adjacent to the current node (node 2).

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

As there is a 1 in the adjacency matrix when matching node 1 with node 2, this means that the path is valid, making the following path:




**Step 4:**

To show this again, the same steps will be repeated with index 3. We will then attempt to increment the next node into the array.

Table 6: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 1 | 2 | 1 | | |

However, since 1 is already in the path, the value will be incremented again.

Table 7: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 1 | 2 | 2 | | |

However, since 2 is already in the path, the value will be incremented again.

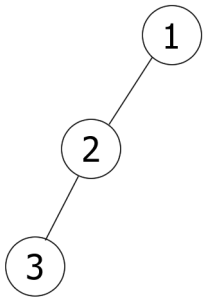
Table 8: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 1 | 2 | 3 | | |

This new value will then be compared against the adjacency matrix to check if the previous node (node 2) is adjacent to the current node (node 3).

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

As there is a 1 in the adjacency matrix when matching node 2 with node 3, this means that the path is valid, making the following graph:




This will be continued until it fills in the entire array. An issue that can occur is if the new node is invalid, in which it will just increment and test the next node. For example, given the following graph and adjacency matrix:

**Invalid Adjacency**

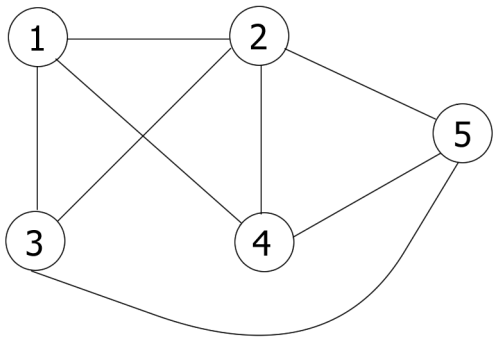



Figure 2: Example Graph 2: Invalid Adjacency

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

Table 9: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 1 | 2 | 3 | 4 | |

As 4 is not a valid adjacent node, it will try incrementing again.

Table 10: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 1 | 2 | 3 | 5 | |

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

As 5 is adjacent to the previous node (node 3), it is valid and will continue until the array is complete.

**Array Complete:**

With reference to graph 1 again, upon completing the array, the path array will look like the following:

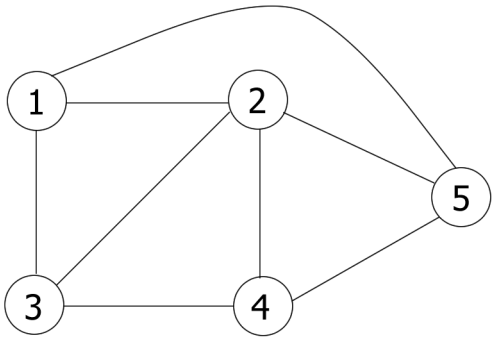
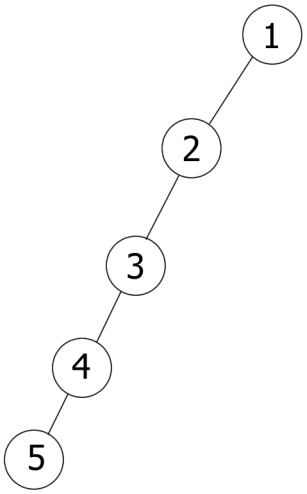



Figure 3: Graph 1: Undirected Graph




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

Table 11: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 1 | 2 | 3 | 4 | 5 |

Where it will also test if the last node (node 5) is adjacent to the first node (node 1), thus fulfilling the criteria of a hamiltonian cycle.

However, as we want to find every single path that results in a hamiltonian cycle, the algorithm will begin to backtrack.

As there is no node after 5, it will remove the last node in the array and attempt to increment the previous node.

Table 12: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 1 | 2 | 3 | 5 | |

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

However, as 5 is not adjacent to 3 and is also the last node, we will remove the node and attempt to increment the previous node.

Since there is a valid adjacency between 2 and 4, we will move onto the next node again. This will repeat recursively, testing if there is an invalid adjacency or not, thus finding every single hamiltonian cycle in the graph.

Table 13: Path Array

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 1 | 2 | 4 | | |

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

## 3.2 Backtracking 2: Eppsteins Algorithm

By considering possible permutations of a graph with degree 3. Eppstein was able to develop an algorithm that had the time-complexity $O(2^{\frac{3n}{8}})$ and in linear space [Epp07]. The algorithm was a careful backtracking technique that identified all possible permutations of a cubic graph and defined a set of back tracking rules.

Eppsteins algorithm looks like as follows.

let $G$ be a graph. let $F$ be the forced edges of a graph.

1. Repeat until they return or they can no longer be applied.
   (a) If degree$(G) < 2$, return empty set $E$.
   (b) If $F$ has three edges meeting at a vertex, return empty set $E$.
   (c) If $F$ has a non hamiltonian cycle, return empty set $E$.
   (d) If $F$ has a hamiltonian cycle for $G$ return cycle.
   (e) If $G$ has a vertex $v$ where degree$(v) == 2$. Add the edges of $v$ to $F$.
   (f) If $G$ has a triangle subgraph with vertices $xyz$, and the non-triangle edge incedent to $x \in F$. Add edge $yz$ to $F$.
   (g) If $G$ has exactly two edges meeting at a vertex, remove from vertex $G$, and all edges incedent. Replace the edges with a single edge connecting the endpoints. If this leads to parallel edges of $G$ remove other edge from $G$.
2. If $|F| > 0$, let $xy$ be any edge in $F$, and $yz$ be an adjacent edge in $G \setminus F$. Else, let $yz$ be any edge in $G$.

3. Call algorithm recursively on $G, F \cup \{yz\}$
4. Call algorithm recursively on $G \setminus \{y, z\}, F$

### 3.3 Dynamic Programming: Held-Karps Algorithm

Stores a bit map of the possible paths in the problem. Uses it as a memoisation table.

# 4 Theoretical Complexity

## 4.1 Backtracking 1: Brute Force DFS

**Lemma 1.** *The depth first search has a running time of $O(n!)$, and a space complexity of $O(n)$.*

*Proof.* The algorithm is a recursive Depth First Search on the Graph. Since the depth first search needs to search all possible permutations of the graph, and there are $n!$ permutations of paths in a graph. The algorithm searches $n!$ paths. The space is required to store the actual path, and to store the graph itself.

The steps 1 and 2 occur in constant time. While the recursive steps occur for the remaining possible graphs. ■

## 4.2 Backtracking 2: Eppsteins Algorithm

**Lemma 2.** *The depth first search has a running time of $O(2^{\frac{3n}{8}})$.*

*Proof.* Much like the depth first backtracking, this algorithm checks all possible permutations of the graph. However, Eppstein proved that there are at most $2^{\frac{3n}{8}}$ possible permutations of a cubic graph. Hence, if you backtrack by only checking the possible permutations, you will achieve that lower bound. ■

## 4.3 Dynamic Algorithm: Held-Karp

**Lemma 3.** *Held-Karps algorithm has a complexity of $O(n^2 \cdot 2^n)$ and a space complexity of $O(n \cdot 2^n)$.*

*Proof.* Due to the nature of results being memoised, instead of trying to find the amount of combinations of paths in the set of edges. Instead you try to find how many paths in the subset of paths that haven't been checked yet. Through expanding the formula for permutation, the complexity can be written as $O(n^2 \cdot 2^n)$

Similarly storing possible values for subsets requires $O(n \cdot 2^n)$ space. ■

# References


[Bel62] Richard Bellman. "Dynamic programming treatment of the travelling salesman problem". In: *Journal of the ACM* 9.1 (1962), pp. 61 63. DOI: 10.1145/321105.321111.

[Epp07] David Eppstein. "The Travelling Salesman Problem for Cubic Graphs". In: *Journal of Graph Algorithms and Applications* 11.1 (2007), pp. 61 –81. DOI: 10.7155/jgaa.00137.

[HK62] Michael Held and Richard M. Karp. "A dynamic programming approach to sequencing problems". In: *Journal of the Society for Industrial and Applied Mathematic* (1962). DOI: 10.1137/0110015.

[KS06] János Komlós and Endre Szemerédi. "Limit distribution for the existence of hamiltonian cycles in a random graph". In: *Discrete Mathematics* 306.10-11 (2006), pp. 1032 1038. DOI: 10.1016/j.disc.2006.03.022.

[Ros07a] Kenneth H. Rosen. "10.4 Connectivity". In: *Discrete mathematics and its applications*. McGraw-Hill, 2007, p. 679.

[Ros07b] Kenneth H. Rosen. "10.5 Euler and Hamiltonian Paths". In: *Discrete mathematics and its applications*. McGraw-Hill, 2007, pp. 698 –703.

[SB21] Joeri Sleegers and Daan van den Berg. *Backtracking (the) Algorithms on the Hamiltonian Cycle Problem*. 2021. DOI: 10.48550/ARXIV.2107.00314. URL: https://arxiv.org/abs/2107.00314.