Hamiltonian Cycle - GeeksforGeeks

Source: https://www.geeksforgeeks.org/hamiltonian-cycle/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Hamiltonian Cycle Last Updated : 31 Jan, 2026 A Hamiltonian Cycle or Circuit in a graph G is a cycle that visits each vertex of G exactly once and returns to the starting vertex. If a graph has a Hamiltonian cycle, it's a Hamiltonian graph; otherwise, it's non-Hamiltonian. Finding a Hamiltonian cycle is an NP-complete problem, meaning there's no known efficient solution for all graph types, but solutions exist for smaller or specific types. The Hamiltonian Cycle problem has applications in logistics, network design, and computer science. Sample Problem Given an undirected graph, the task is to determine if it contains a Hamiltonian cycle. If found, print the path; otherwise, print "Solution does not exist". Examples: Input : N=5, adjMat[][] = [[0, 1, 0, 1, 0], [1, 0, 1, 1, 1], [0, 1, 0, 0, 1], [1, 1, 0, 0, 1], [0, 1, 1, 1, 0]] Output : [0, 1, 2, 4, 3, 0] Input : N=5, adjMat[][] = [[0, 1, 0, 1, 0], [1, 0, 1, 1, 1], [0, 1, 0, 0, 1], [1, 1, 0, 0, 0], [0, 1, 1, 0, 0]] Output : "Solution Does Not Exists" Try it on GfG Practice [Naive Approach] Generate All Configurations Generate all possible vertex configurations and print a configuration that satisfies the given constraints. This results in n! (n factorial) configurations. Therefore, the overall time complexity of this approach is at least O(n!). [Expected Approach] Using Backtracking Backtracking is used to find a Hamiltonian Cycle. Initialize an empty path array, place the starting vertex (e.g., 0) in the first position. Recursively add remaining vertices one by one, ensuring each new vertex is adjacent to the previous one and not already in the path. If a valid vertex is found, proceed; otherwise, backtrack. Exemplification Let's find out the Hamiltonian cycle for the following graph: Start with node 0. Apply DFS to find a Hamiltonian path. When all 'n' vertices are traversed, check if the current node is a neighbor of the starting node. Since node 2 is not a neighbor of 0, return. As no cycle is found in path {0, 3, 1, 4, 2}, return from nodes 2 and 4. Explore node 2. Check for a Hamiltonian cycle. Since node 4 is not a neighbor of 0, return. Return from nodes 4, 2, and 1. Explore node 3. In path {0, 3, 4, 2, 1, 0}, a cycle is found, print this cyclic path. This is our Hamiltonian cycle. Note: The algorithm currently starts from a fixed vertex, but a Hamiltonian cycle allows any vertex. Minor adjustments are needed to enable cycle initiation from any node. C++ #include <iostream> #include <vector> using namespace std ; // Check if it's valid to place vertex at current position bool isSafe (int vertex , vector < vector < int >> & adjMat , vector < int > & path , int pos) { // The vertex must be adjacent to the previous vertex if (! adjMat [path [pos - 1]][vertex]) { return false ; } // The vertex must not already be in the path for (int i = 0 ; i < pos ; i ++) { if (path [i] == vertex) { return false ; } } return true ; } // Recursive backtracking to construct Hamiltonian Cycle bool hamCycleUtil (vector < vector < int >> & adjMat , vector < int > & path , int pos , int n) { // Base case: all vertices are in the path if (pos == n) { // Check if there's an edge from last to first vertex return adjMat [path [pos - 1]][path [0]]; } // Try all possible vertices as next candidate for (int v = 1 ; v < n ; v ++) { if (isSafe (v , adjMat , path , pos)) { path [pos] = v ; if (hamCycleUtil (adjMat , path , pos + 1 , n)) { return true ; } // Backtrack if v doesn't lead to a solution path [pos] = -1 ; } } return false ; } // Initialize path and invoke backtracking function vector < int > hamCycle (vector < vector < int >> & adjMat) { int n = adjMat . size (); vector < int > path (n , -1); // Start path with vertex 0 path [0] = 0 ; if (! hamCycleUtil (adjMat , path , 1 , n)) { return { -1 }; } return path ; } // Driver Code int main () { vector < vector < int >> adjMat = { { 0 , 1 , 0 , 1 , 0 }, { 1 , 0 , 1 , 1 , 1 }, { 0 , 1 , 0 , 0 , 1 }, { 1 , 1 , 0 , 0 , 1 }, { 0 , 1 , 1 , 1 , 0 } }; vector < int > path = hamCycle (adjMat); if (path [0] == -1) { cout << "Solution does not Exist" ; } else { for (int i = 0 ; i < path . size (); i ++) { cout << path [i] << " " ; } cout << path [0]; } return 0 ; } Java import java.util.ArrayList ; import java.util.List ; public class GFG { // Check if it's valid to place vertex at current position private static boolean isSafe (

int vertex , int [][] adjMat , List < Integer > path , int pos) { // The vertex must be adjacent to the previous vertex if (adjMat [path . get (pos - 1)][vertex] == 0) { return false ; } // The vertex must not already be in the path for (int i = 0 ; i < pos ; i ++) { if (path . get (i) == vertex) { return false ; } } return true ; } // Recursive backtracking to construct Hamiltonian Cycle private static boolean hamCycleUtil (int [][] adjMat , List < Integer > path , int pos , int n) { // Base case: all vertices are in the path if (pos == n) { // Check if there's an edge from last to first vertex return adjMat [path . get (pos - 1)][path . get (0)] == 1 ; } // Try all possible vertices as next candidate for (int v = 1 ; v < n ; v ++) { if (isSafe (v , adjMat , path , pos)) { path . set (pos , v); if (hamCycleUtil (adjMat , path , pos + 1 , n)) { return true ; } // Backtrack if v doesn't lead to a solution path . set (pos , - 1); } } return false ; } // Initialize path and invoke backtracking function public static List < Integer > hamCycle (int [][] adjMat) { int n = adjMat . length ; List < Integer > path = new ArrayList <> (n); for (int i = 0 ; i < n ; i ++) { path . add (- 1); } // Start path with vertex 0 path . set (0 , 0); if (! hamCycleUtil (adjMat , path , 1 , n)) { List < Integer > noSolution = new ArrayList <> (); noSolution . add (- 1); return noSolution ; } return path ; } // Driver Code public static void main (String [] args) { int [][] adjMat = { { 0 , 1 , 0 , 1 , 0 }, { 1 , 0 , 1 , 1 , 1 }, { 0 , 1 , 0 , 0 , 1 }, { 1 , 1 , 0 , 0 , 1 }, { 0 , 1 , 1 , 1 , 0 } }; List < Integer > path = hamCycle (adjMat); if (path . get (0) == - 1) { System . out . println ("Solution does not Exist"); } else { for (int i = 0 ; i < path . size (); i ++) { System . out . print (path . get (i) + " "); } System . out . print (path . get (0)); } } } Python # Check if it's valid to place vertex at current position def isSafe (vertex , adjMat , path , pos): # The vertex must be adjacent to the previous vertex if adjMat [path [pos - 1]][vertex] == 0 : return False # The vertex must not already be in the path for i in range (pos): if path [i] == vertex : return False return True # Recursive backtracking to construct Hamiltonian Cycle def hamCycleUtil (adjMat , path , pos , n): # Base case: all vertices are in the path if pos == n : # Check if there's an edge from last to first vertex return adjMat [path [pos - 1]][path [0]] == 1 # Try all possible vertices as next candidate for v in range (1 , n): if isSafe (v , adjMat , path , pos): path [pos] = v if hamCycleUtil (adjMat , path , pos + 1 , n): return True # Backtrack if v doesn't lead to a solution path [pos] = - 1 return False # Initialize path and invoke backtracking function def hamCycle (adjMat): n = len (adjMat) path = [- 1] * n # Start path with vertex 0 path [0] = 0 if not hamCycleUtil (adjMat , path , 1 , n): return [- 1] return path if __name__ == "__main__" : adjMat = [[0 , 1 , 0 , 1 , 0], [1 , 0 , 1 , 1 , 1], [0 , 1 , 0 , 0 , 1], [1 , 1 , 0 , 0 , 1], [0 , 1 , 1 , 1 , 0]] path = hamCycle (adjMat) if path [0] == - 1 : print ("Solution does not Exist") else : for v in path : print (v , end = " ") print (path [0]) C# using System ; using System.Collections.Generic ; class GFG { // Check if it's valid to place vertex at current position static bool isSafe (int vertex , int [,] adjMat , List < int > path , int pos) { // The vertex must be adjacent to the previous vertex if (adjMat [path [pos - 1], vertex] == 0) { return false ; } // The vertex must not already be in the path for (int i = 0 ; i < pos ; i ++) { if (path [i] == vertex) { return false ; } } return true ; } // Recursive backtracking to construct Hamiltonian Cycle static bool hamCycleUtil (int [,] adjMat , List < int > path , int pos , int n) { // Base case: all vertices are in the path if (pos == n) { // Check if there's an edge from last to first vertex return adjMat [path [pos - 1], path [0]] == 1 ; } // Try all possible vertices as next candidate for (int v = 1 ; v < n ; v ++) { if (isSafe (v , adjMat , path , pos)) { path [pos] = v ; if (hamCycleUtil (adjMat , path , pos + 1 , n)) { return true ; } // Backtrack if v doesn't lead to a solution path [pos] = - 1 ; } } return false ; } // Initialize path and invoke backtracking function static List < int > hamCycle (int [,] adjMat) { int n = adjMat . GetLength (0); List < int > path = new List < int > (); for (int i = 0 ; i < n ; i ++) path . Add (- 1); // Start path with vertex 0 path [0] = 0 ; if (! hamCycleUtil (adjMat , path , 1 , n)) { return new List < int > { - 1 }; } return path ; } // Driver Code static void Main () { int [,] adjMat = { { 0 , 1 , 0 , 1 , 0 }, { 1 , 0 , 1 , 1 , 1 }, { 0 , 1 , 0 , 0 , 1 }, { 1 , 1 , 0 , 0 , 1 }, { 0 , 1 , 1 , 1 , 0 } }; List < int > path = hamCycle (adjMat); if (path [0] == - 1) { Console . WriteLine ("Solution does not Exist"); } else { foreach (int v in path) Console . Write (v + " "); Console . Write (path [0]); } } } JavaScript // Check if it's valid to place vertex at current position function isSafe (vertex , adjMat , path , pos) { // The vertex must be adjacent to the previous vertex if (adjMat [path [pos - 1]][vertex] === 0) { return false ; } // The vertex must not already be in the path for (let i = 0 ; i < pos ; i ++) { if (path [i] === vertex) { return false ; } } return true ; } // Recursive backtracking to construct Hamiltonian Cycle function hamCycleUtil (adjMat , path , pos , n) { // Base case: all vertices are in the path if (pos === n) { // Check if there's an edge from last to first vertex return adjMat [path [pos - 1]][path [0]] === 1 ; } // Try all possible vertices as next candidate for (let v = 1 ; v < n ; v ++) { if (isSafe (v , adjMat , path , pos)) { path [pos] = v ; if (hamCycleUtil (adjMat , path , pos + 1 , n)) { return true ; } // Backtrack if v doesn't lead to a solution path [pos] = - 1 ; } } return false ; } // Initialize path and invoke backtracking function function hamCycle (adjMat) { const n = adjMat . length ; const path = new Array (n). fill (- 1); // Start path with vertex 0 path [0] = 0 ; if (! hamCycleUtil (adjMat , path , 1 , n)) { return [- 1]; } return

path ; } // Driver Code const adjMat = [[0 , 1 , 0 , 1 , 0], [1 , 0 , 1 , 1 , 1], [0 , 1 , 0 , 0 , 1], [1 , 1 , 0 , 0 , 1], [0 , 1 , 1 , 1 , 0]]; const path = hamCycle (adjMat); if (path [0] === - 1) { console . log ("Solution does not Exist"); } else { console . log (path . join (" ") + " " + path [0]); } Output 0 1 2 4 3 0 Time Complexity: O(n!) Space Complexity: O(n) Comment Article Tags: Article Tags: Graph Backtracking DSA Amazon DFS + 1 More