CSE 362 : Final Presentation

Hamiltonian Cycle Problem

Group 7:
Adrita Hossain Nakshi (1705019)
Md. Shafqat Talukder (1705026)
Simantika Bhattacharjee Dristi (1705029)
Joy Saha (1705030)
Fahmid Al Rifat (1705087)

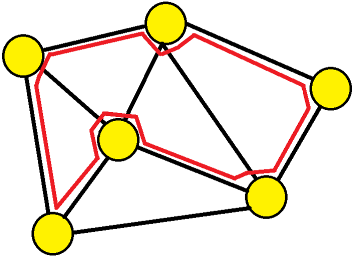
February 23, 2023

Table of Contents

Hamiltonian Cycle

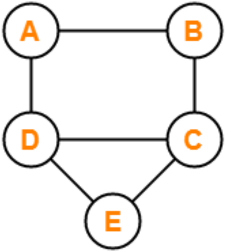
Definition

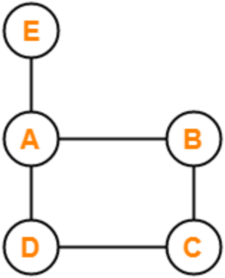
In the mathematical field of graph theory, a Hamiltonian Cycle is a cycle in an directed or undirected graph that visits **each vertex exactly once**.



Hamiltonian Cycle Problem

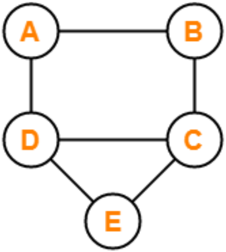
Not all graphs have Hamiltonian Cycle

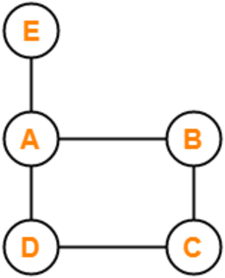




Hamiltonian Cycle Problem

Not all graphs have Hamiltonian Cycle





Infact, detecting Hamiltonian Cycle in a graph, G is a NP-complete problem!

Applications of HCP

- **Transportation Planning**
  - Finding a route that visits a number of different locations exactly once and returns back to the initial location.
- **Genome Assembly**
  - Determining the order in which to sequence the DNA fragments of a genome in the process of reconstructing the complete DNA sequence of an organism from short DNA fragments.
- **Computer security:**
  - Identifying potential vulnerabilities in network systems, as the presence of a Hamiltonian cycle in a network can indicate that an attacker could gain access to all parts of the network.

More Applications

- Route of a postal carrier in postal system
- Robotics and automation
- Electronic circuit design
- Network analysis

Motivation

Apart from practical applications, Hamiltonian Cycle Problem is also theoretically important in the sense that-

- The NP-completeness of Hamiltonian cycle problem has important implications for the study of algorithms and computational complexity.

Motivation

Apart from practical applications, Hamiltonian Cycle Problem is also theoretically important in the sense that-

- ☞ The NP-completeness of Hamiltonian cycle problem has important implications for the study of algorithms and computational complexity.
- ☞ Hamiltonian cycle problem has connections to other important problems in graph theory and computer science, such as the traveling salesman problem.

Motivation

Apart from practical applications, Hamiltonian Cycle Problem is also theoretically important in the sense that-

- ☞ The NP-completeness of Hamiltonian cycle problem has important implications for the study of algorithms and computational complexity.
- ☞ Hamiltonian cycle problem has connections to other important problems in graph theory and computer science, such as the traveling salesman problem.

Hence, there is always more to explore about Hamiltonian Cycle Problem!

Problem Formulation

Hamiltonian Cycle Problem: Determine an hamiltonian cycle of the graph G.

- **Input:** A graph **G(V,E)**

Problem Formulation

Hamiltonian Cycle Problem: Determine an hamiltonian cycle of the graph G.

- **Input:** A graph **G(V,E)**
- **Output:** A hamiltonian cycle if there is one or more present, otherwise print there is no cycle.

Hamiltonian Cycle decision version

Decision version: Does graph G have a hamiltonian cycle?.

- **Input:** A graph **G(V,E)**

Hamiltonian Cycle decision version

Decision version: Does graph G have a hamiltonian cycle?.

- **Input:** A graph **G(V,E)**
- **Output:** Yes if there exists a hamiltonian cycle, otherwise no.

Basic Idea

- First we create an empty path array and add vertex 0 to it. This vertex 0 becomes the root of our implicit tree.

Basic Idea

- First we create an empty path array and add vertex 0 to it. This vertex 0 becomes the root of our implicit tree.
- The next vertex added to the path if it is adjacent to the previously added vertex and not already visited during this path.

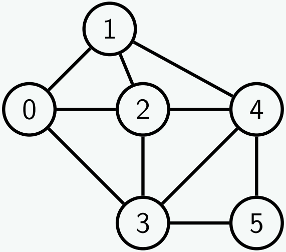
Basic Idea

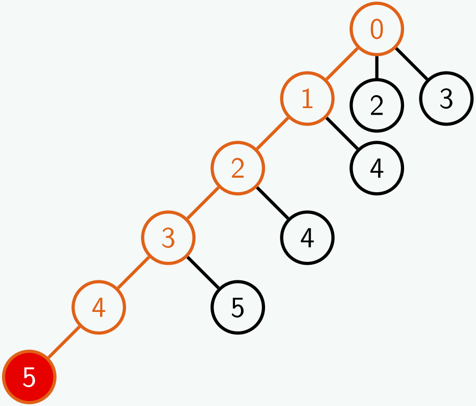
- First we create an empty path array and add vertex 0 to it. This vertex 0 becomes the root of our implicit tree.
- The next vertex added to the path if it is adjacent to the previously added vertex and not already visited during this path.
- If at any stage, no unvisited adjacent vertex can be found then we can say that a **'dead end'** is reached.
  In this case, we backtrack one step, and again the search begins by selecting another adjacent vertex.

Basic Idea

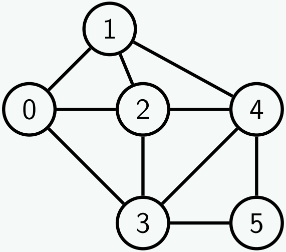
- First we create an empty path array and add vertex 0 to it. This vertex 0 becomes the root of our implicit tree.
- The next vertex added to the path if it is adjacent to the previously added vertex and not already visited during this path.
- If at any stage, no unvisited adjacent vertex can be found then we can say that a **'dead end'** is reached.
  In this case, we backtrack one step, and again the search begins by selecting another adjacent vertex.
- The search using backtracking is successful if an Hamiltonian Cycle is obtained otherwise return false.

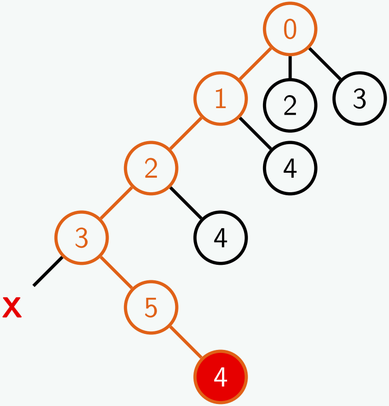
Simulation



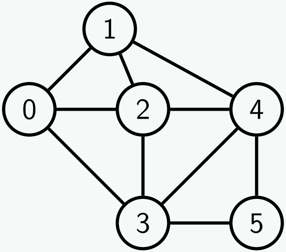


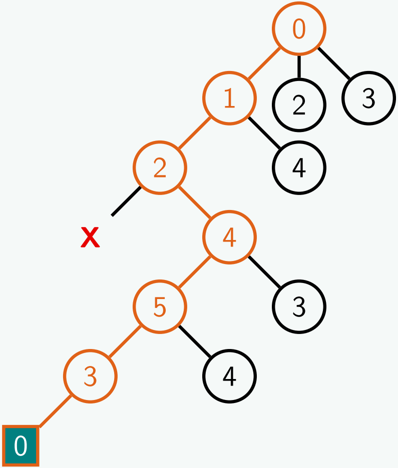
Simulation





Simulation





Code Outputs

```
current node =  0  visited node mask =  000001
current node =  1  visited node mask =  000011
current node =  2  visited node mask =  000111
current node =  3  visited node mask =  001111
current node =  4  visited node mask =  011111
current node =  5  visited node mask =  111111
current node =  5  visited node mask =  101111
current node =  4  visited node mask =  111111
current node =  4  visited node mask =  010111
current node =  3  visited node mask =  011111
current node =  5  visited node mask =  111111
Memory used for node =  5  mask =  111111
current node =  5  visited node mask =  110111
current node =  3  visited node mask =  111111

Hamiltonian cycle found:  [0, 1, 2, 4, 5, 3, 0]
```

Implementation Details

```
def hamiltonian_cycle(graph):
    n = len(graph)  # Number of vertices in the graph
    memo = {}  # Memorization  dictionary to store previously computed results

    # Recursive function to compute the Hamiltonian cycle
    def dp(start, visited):
        print("current node = ", start," visited node mask = ", format(visited,
'06b'))
        # If already computed the result for this node and visited set, return it
        if (start, visited) in memo:
            print("Memory used for node = ", start," mask = ", format(visited, '06b'))
            return memo[(start, visited)]

        # If we have visited all vertices and
        # there is an edge from the last vertex to the first vertex,
        # we have found a Hamiltonian cycle
        if visited == (1 << n) - 1:
            if graph[start][0]:
                return [start, 0]
        ...
```

Implementation Details

```python
def hamiltonian_cycle(graph):
    n = len(graph)  # Number of vertices in the graph
    memo = {}  # Memorization  dictionary to store previously computed results

    # Recursive function to compute the Hamiltonian cycle
    def dp(start, visited):
        ...
        # Try each unvisited neighbor of the current vertex to extend the path
        for neighbor in range(n):
            if graph[start][neighbor] and not visited & (1 << neighbor):
                cycle = dp(neighbor, visited | (1 << neighbor))
                if cycle is not None:
                    if cycle[-1] != 0:
                        continue
                    # Store the result in the memorization  dictionary and return it
                    memo[(start, visited)] = [start] + cycle
                    return memo[(start, visited)]

        # If we reach this point, no Hamiltonian cycle was found and return None
        memo[(start, visited)] = None
        return None

    # Call the recursive function with the starting vertex and its corresponding bit in the visited set
    cycle = dp(0, 1 << 0)

    if cycle is not None:
        return cycle # If a Hamiltonian cycle was found, return it
    return None # If no Hamiltonian cycle was found, return None
```

Complexity Analysis

Time complexity : $O(n!)$

Space complexity : $O(n * 2^n)$

Heuristic Algorithm

Input: A graph **G(V, E)**

Output: Whether there is any Hamiltonian cycle in graph **G**; if there is, output a cycle, if there is not, output "No Hamiltonian cycle".

Used Heuristic: The heuristic information of each vertex is a set composed of its possible path length values from the starting vertex.

Reference: Jin, D., Li, Q. Lu, M. A heuristic search algorithm for Hamiltonian circuit problems in directed graphs. Wireless Netw 28, 979–989 (2022).

Basic Idea

A Hamiltonian cycle on a graph G(V,E) is a loop starting from the starting point S and passing through the remaining vertices in the graph once and only once and back to the starting point. Therefore, the path length from the starting point to any other vertex can be 0, 1, 2, ...n.

The algorithm involves two steps:

- **Step 1:** The heuristic information acquisition step.
- **Step 2:** The heuristic search step.

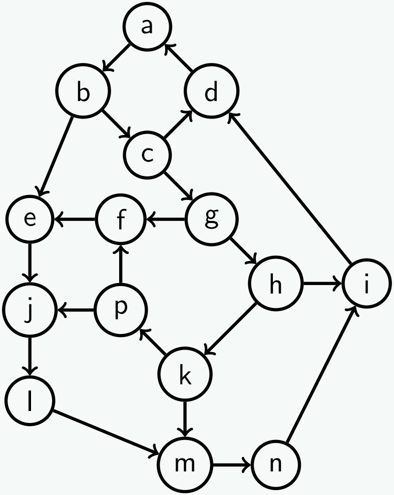
Pseudocode

HeuristicAlgorithm(G):

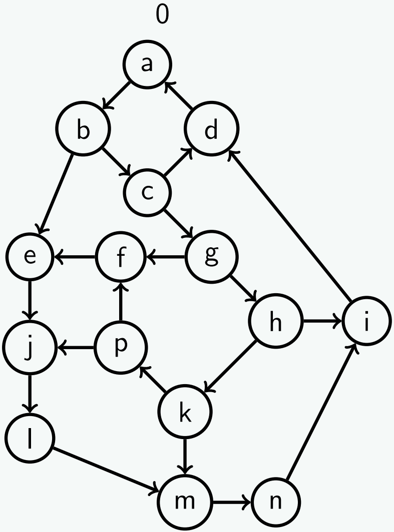
Step 1:

Compute the distance from the starting vertex to all other vertex using all paths possible. Add the distances to the list of corresponding vertex one by one.

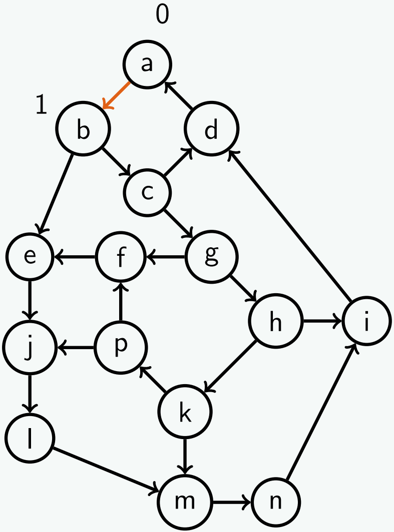
Simulation



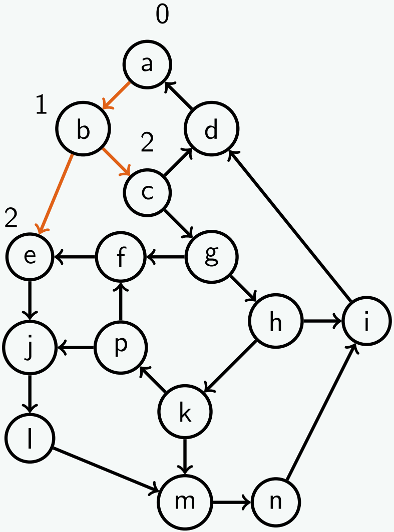
Continued...



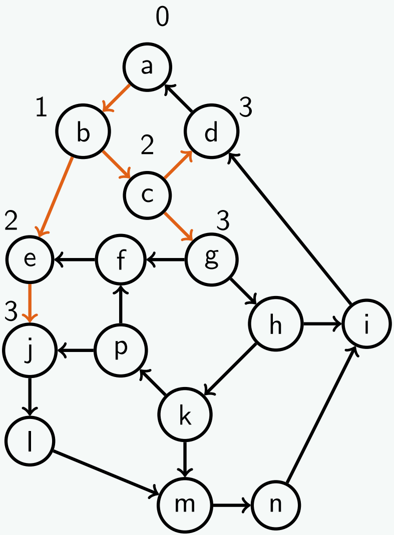
Continued...



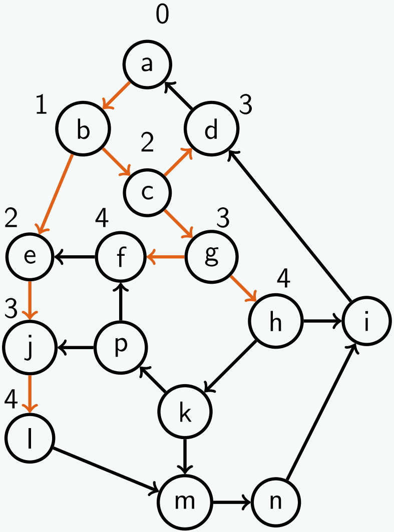
Continued...



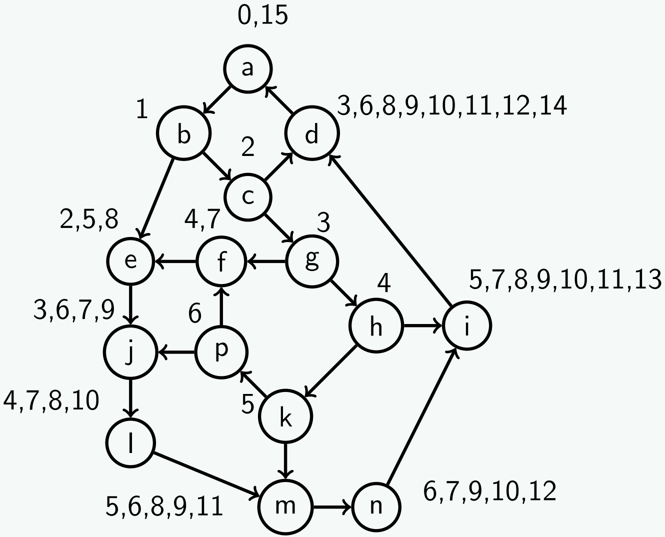
Continued...



Continued...



Continued...


0,15
a
1
b
2
d
3,6,8,9,10,11,12,14
c
2,5,8
4,7
3
e
f
g
4
3,6,7,9
6
h
i
5,7,8,9,10,11,13
j
p
4,7,8,10
5
k
l
5,6,8,9,11
m
n
6,7,9,10,12

Pseudocode

HeuristicAlgorithm(G):

Step 1:

Compute the distance from the starting vertex to all other vertex using all paths possible. Add the distances to the list of corresponding vertex one by one.

Step 2:

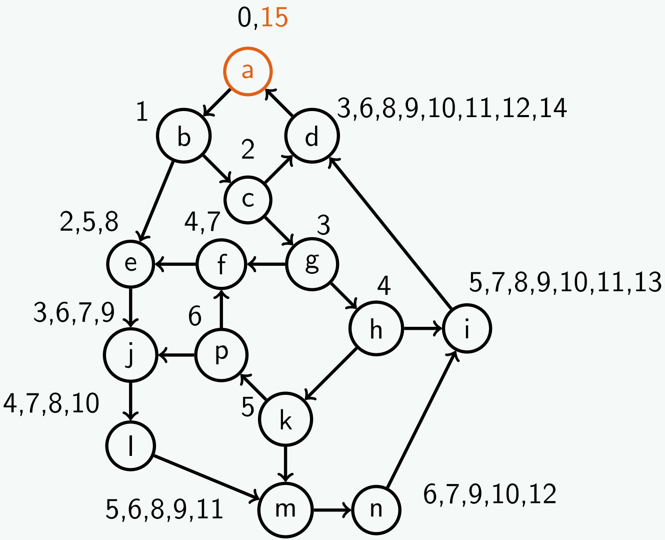
Starting from the start node determine if any i^{th} node from the starting vertex contains value N-i in it's list. Add the vertex to cycle list. Stop when starting vertex is reached.

if (No such vertex found at any point)

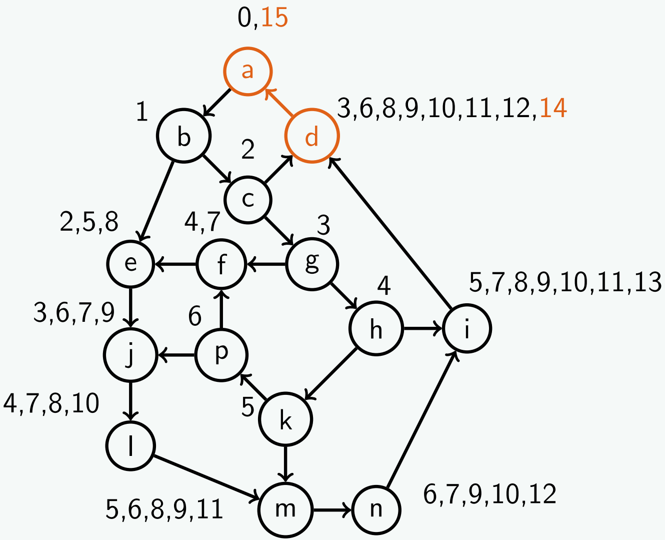
then return No Hamiltonian cycle

else return the Hamiltonian cycle

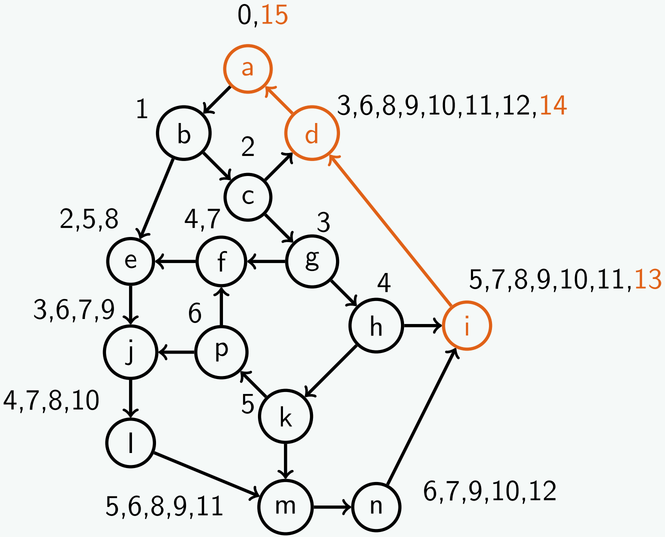
Simulation


0,15
a
1
b
2
d
3,6,8,9,10,11,12,14
c
2,5,8
4,7
3
e
f
g
4
h
i
5,7,8,9,10,11,13
3,6,7,9
6
j
p
4,7,8,10
5
k
l
m
n
5,6,8,9,11
6,7,9,10,12

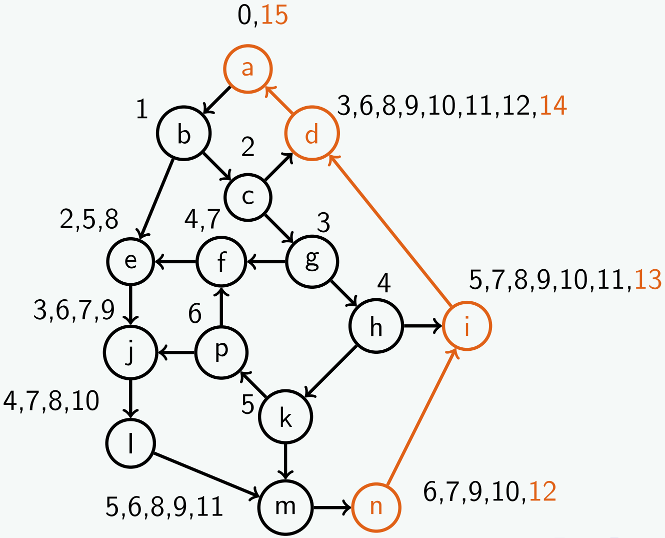
Continued...


0,15
a
1
b
2
d
3,6,8,9,10,11,12,14
c
2,5,8
4,7
3
e
f
g
4
5,7,8,9,10,11,13
3,6,7,9
6
h
i
j
p
4,7,8,10
5
k
l
5,6,8,9,11
m
n
6,7,9,10,12

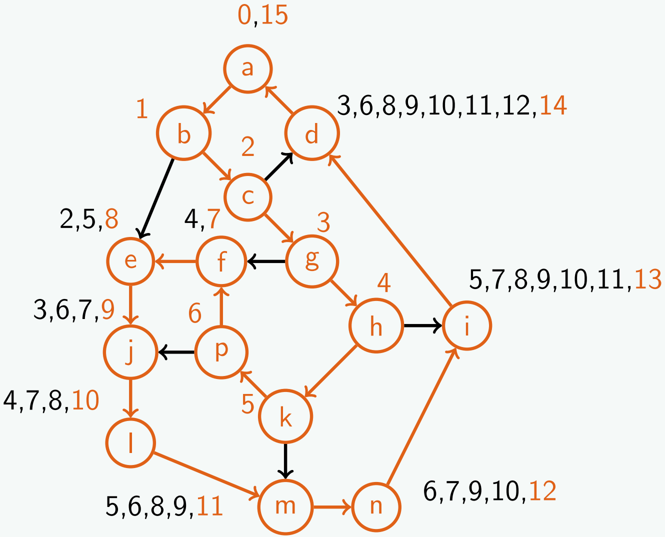
Continued...


0,15
a
1
b
2
d
3,6,8,9,10,11,12,14
c
2,5,8
4,7
3
e
f
g
4
3,6,7,9
6
h
i
5,7,8,9,10,11,13
j
p
4,7,8,10
5
k
l
5,6,8,9,11
m
n
6,7,9,10,12

Continued...


0,15
a
1
b
2
d
3,6,8,9,10,11,12,14
c
2,5,8
4,7
3
e
f
g
4
3,6,7,9
6
h
i
5,7,8,9,10,11,13
j
p
4,7,8,10
5
k
l
5,6,8,9,11
m
n
6,7,9,10,12

Continued...


0,15
a
1
b
2
d
3,6,8,9,10,11,12,14
c
2,5,8
4,7
3
e
f
g
4
5,7,8,9,10,11,13
3,6,7,9
6
h
i
j
p
4,7,8,10
5
k
l
m
n
5,6,8,9,11
6,7,9,10,12

Implementation Details

```python
def apply_heuristic(graph):
    n = len(graph) # Number of nodes in the graph

    # Initialize two list of lists to store heuristic information
    L = [[] for i in range(n)] # List of all possible distances from starting node(0) to vertex i
    R = [[] for i in range(n+1)] #List of nodes that are at distance i from starting node(0)

    curv, curl = 0, 0
    L[curv].append(curl)
    R[curl].append(curv)

    ...
```

Implementation Details

```python
def apply_heuristic(graph):
    ...

    while curl <= n:
        if curl==n:
            # Add starting node(0) to the cycle before returning from the function
            for p in R[curl-1]:
                if(0 in graph[p]):
                    L[0].append(n)
                    break
            return R, L
        elif len(R[curl])==0: # If there is no vertex left to explore, terminate (No instance)
            return None,None

        # Traverse through all neighbours of nodes stored at R[curl]
        for i in R[curl]:
            for j in graph[i]:
                if(j==0) :# Skip if starting node(0)
                    continue
                # For each distance stored at L[i], add (distance+1) to L[j] if j is a neighbour of i
                for x in L[i]:
                    L[j].append(x+1)

                R[curl+1].append(j) # Update the R list
                L[j] = list(set(L[j])) # Remove duplicates from L[i]
        curl += 1
```

Implementation Details

```python
def find_cycle(graph, L, R):
    n = len(graph)  # Number of nodes in the graph
    visited = []
    previous_node = 0
    current_node = None

    for i in range(n-1, -1, -1):
        possible_nodes = R[i]           # Nodes whose list contain value i
        if(len(possible_nodes)==0):     # If no node contains value i, then no cycle
            return False, None
        for j in range(len(possible_nodes)):    # Iterate over all nodes containing value i
            # If a possible node is adjacent to the previous node and it is not visited yet
            # then make it current node and mark as visited
            if((previous_node in graph[possible_nodes[j]]) and (possible_nodes[j] not in visited)):
                current_node = possible_nodes[j]
                visited.append(current_node)
                break
            # If none of the possible nodes can be taken, then there is no cycle
            if(j==len(possible_nodes)-1):
                return False, None

        previous_node = current_node
    return True, visited
```

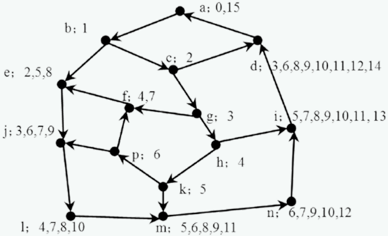
Complexity Analysis

Time complexity :

Best Case : $\Omega(\mathrm{n}^2)$
Worst Case : $\mathrm{O}(n!)$

Space complexity : $O(n^2)$

Comparison



No of vertex: 15

No of edges: 21

Algorithm Name	Time Complexity	Space Complexity	Run Time
Backtracking	O(n!)	O($n2^n$)	51300 ns
Heuristic Algorithm	O(n!)	O(n^2)	40200 ns

Conclusion

- In the example graph the heuristic based algorithm needs to explore 15 nodes to determine whether there is a hamiltonian circuit whereas backtracking method needs at least 21 nodes. Therefore, the heuristic search algorithm can greatly reduce the number of processing nodes compared to the backtracking algorithm.

Conclusion

- In the example graph the heuristic based algorithm needs to explore 15 nodes to determine whether there is a hamiltonian circuit whereas backtracking method needs at least 21 nodes. Therefore, the heuristic search algorithm can greatly reduce the number of processing nodes compared to the backtracking algorithm.
- The implemented heuristic method gradually loses its effectiveness as the number of loops, more specifically the number of loops without the starting vertex increases in the graph. Additional efforts can be undertaken to enhance the heuristic information acquisition algorithm (step 1) in order to effectively address this issue.

Thank You!!