Hamiltonian Cycle Problem

Group 7 :

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

Computer Science & Engineering
Bangladesh University of Engineering & Technology

March 3, 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Definition

Hamiltonian Cycle

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. In other words, it is a path that starts and ends at the same vertex and visits every other vertex exactly once.

Hamiltonian Cycle Problem

Not all graphs have Hamiltonian cycle. Determining whether a graph has a Hamiltonian cycle or not is an important problem of graph theory. The Hamiltonian cycle problem is the problem of determining a Hamiltonian cycle in the input graph. It is an NP-complete problem, which means that it is computationally difficult to solve for large graphs.

Problem Formulation

Problem: Determine an Hamiltonian cycle of the graph $\boldsymbol{G}$.
Input: A graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$
Output: A Hamiltonian cycle if there is one or more present, otherwise print there is no cycle.

Decision Version

Problem: Does graph G have a hamiltonian cycle?
Input: A graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$
Output: Yes if there exists a hamiltonian cycle, otherwise no.

1.2 Variations

Hamiltonian Path Problem

The Hamiltonian Path Problem asks whether there is a route in a directed graph from a beginning node to an ending node, visiting each node exactly once. The Hamiltonian Path Problem is NP-complete, achieving surprising computational complexity with modest increases in size.
If a Hamiltonian path exists whose endpoints are adjacent, then the resulting graph cycle is a Hamiltonian cycle. All hamiltonian cycles contain hamiltonian paths, but a hamiltonial path not necessarily will always form a hamiltonian cycle.

Travelling Salesman Problem

The travelling salesman problem (also called the travelling salesperson problem or TSP) asks the following question: ”Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?” It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.
So, in TSP we basically find the least weighted hamiltonian cycle.

Chapter 2

Reductions to and from

2.1 Reduction from 3-SAT to Hamiltonian Cycle Problem

We reduce from the 3-SAT problem to Hamiltonian Cycle Problem. We will show that

$$\textbf{3-SAT} \leq_p \textbf{Hamiltonian Cycle}$$

Input of 3-SAT :

A **3-CNF** formula over variables $\boldsymbol{x_1}$, $\boldsymbol{x_2}$, ..., $\boldsymbol{x_n}$ is the conjunction of m clauses $\boldsymbol{C_1 \wedge C_2 \wedge ... \wedge C_m}$, where each clause is the disjunction of 3 literals, $\boldsymbol{C_i = l_{i1} \vee l_{i2} \vee l_{i3}}$, and each literal $\boldsymbol{l_{ij}}$ is either a variable or the negation of a variable.
For example, $\mathbf{Y} = \boldsymbol{(x_1 \vee \neg x_2 \vee x_3)\ (\neg x_1 \vee \neg x_2 \vee \neg x_3)}$

Input of Hamiltonian Cycle problem :

A graph $\mathbf{G(V,E)}$ with $\mathbf{V}$ vertices and $\mathbf{E}$ edges

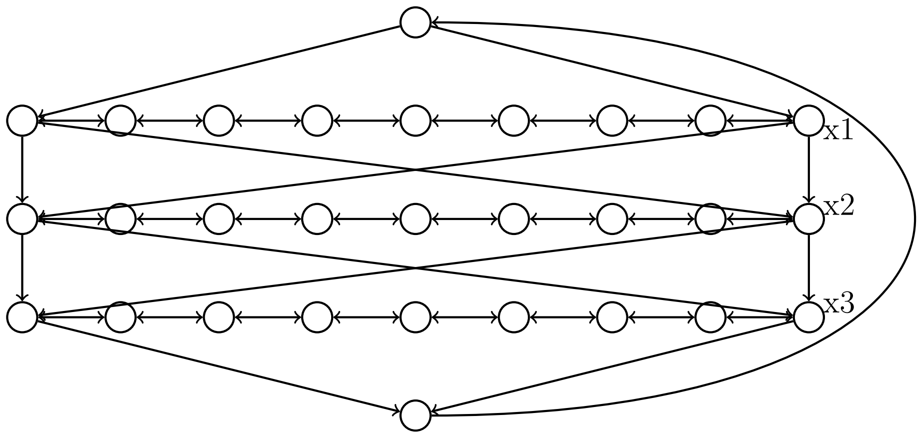
2.1.1 Reduction

Given 3-SAT formula $\mathbf{Y}$ create a graph $\boldsymbol{G_Y}$ such that

- $\boldsymbol{G_Y}$ has a Hamiltonian cycle if and only if $\mathbf{Y}$ is satisfiable.
- $\boldsymbol{G_Y}$ should be constructed from $\mathbf{Y}$ by a polynomial time algorithm $\mathbf{A}$.

2.1.2 Reduction: First Ideas

- Construct graph with $\mathbf{2^n}$ Hamiltonian cycles, where each cycle corresponds to some boolean assignment.
- Then add more graph structure to encode constraints on assignments imposed by the clauses.
- Each clause has 3 ways in which it can be visited.



This graph contains $\mathbf{2^n}$ hamiltonian cycle.

$$\mathbf{Y} = (\boldsymbol{x_1} \vee \neg \boldsymbol{x_2} \vee \boldsymbol{x_3})\ (\neg \boldsymbol{x_1} \vee \neg \boldsymbol{x_2} \vee \neg \boldsymbol{x_3})$$

2.1.3 The Reduction: Phase - I

- There will be a path for each variable.
- Each path has **3(m + 1)** nodes where **m** is number of clauses in **Y**; nodes numbered from left to right is **(1 to 3m + 3)**.

2.1.4 The Reduction: Phase - II

- Add vertex $\boldsymbol{c_j}$ for clause $\boldsymbol{C_j}$. $\boldsymbol{c_j}$ has edge from vertex **3j** and to vertex **3j + 1** on path **i** if $\boldsymbol{x_i}$ appears in clause $\boldsymbol{C_j}$, and has edge from vertex **3j + 1** and to vertex **3j** if $\neg\boldsymbol{x_i}$ appears in $\boldsymbol{C_j}$.

2.1.5 Correctness of proof

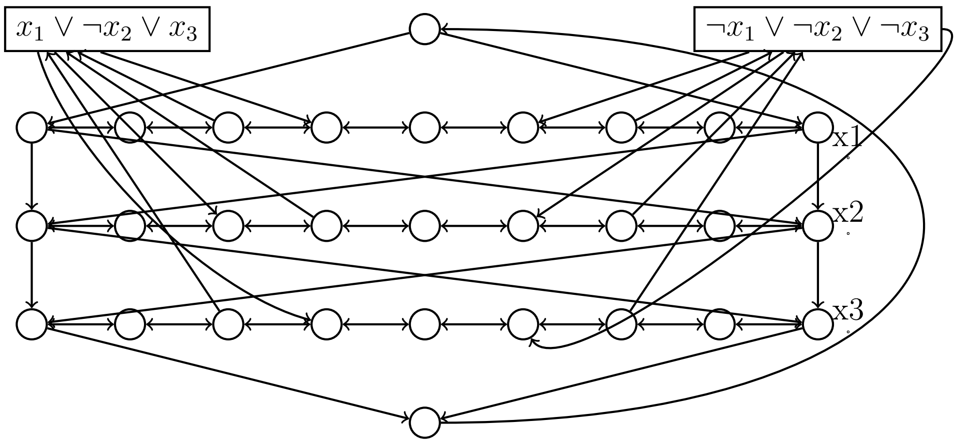
Proposition

Y has a satisfying assignment if and only if $\boldsymbol{G_Y}$ has a Hamiltonian cycle.

Proof :

Let a be the satisfying assignment for **Y**. Define Hamiltonian cycle as -

- If a$(x_i) = 1$ then traverse path i from left to right
- If a$(x_i) = 0$ then traverse path i from right to left


$x_1 \lor \neg x_2 \lor x_3$
$\neg x_1 \lor \neg x_2 \lor \neg x_3$
x1
x2
x3

$$\mathbf{Y} = (\boldsymbol{x_1} \lor \neg \boldsymbol{x_2} \lor \boldsymbol{x_3})\ (\neg \boldsymbol{x_1} \lor \neg \boldsymbol{x_2} \lor \neg \boldsymbol{x_3})$$

2.1.6 Hamiltonian Cycle $\implies$ Satisfying assignment

Let **X** be a Hamiltonian Cycle in $\boldsymbol{G_Y}$

- Vertices visited immediately before and after C_i are connected by an edge
- We can remove c_j from cycle, and get Hamiltonian cyle in $\boldsymbol{G - c_j}$
- Consider Hamiltonian cycle in $\boldsymbol{G - \{c_1, ..., c_m\}}$; it traverses each path in only one direction, which determines the truth assignment
- If **X** enters c_j (vertex for clause C_j) from vertex 3j on path i then it must leave the clause vertex on edge to **3j + 1** on the same path i
  - If not, then only unvisited neightbor of **3j + 1** on path i is **3j + 2**
  - So, no two unvisited neighbors (one to enter from, and the other to leave) to have a Hamiltonian Cycle
- Similarly, if **X** enters c_j from vertex **3j + 1** on path i then it must leave the clause vertex c_j on edge to **3j** on path i
- So, **X** will centainly provide a satisfiable assignment

2.2 Reduction from Hamiltonian Cycle to TSP

We reduce from the Hamiltonian Cycle Problem to the Traveling Salesman Problem in Graphs. Given an instance of Hamiltonian Cycle, namely a graph $G = (V, E)$ with $n = |V|$ vertices, our reduction outputs the following instance of TSP in Graphs:

- A complete graph G' on the same vertex set V, where each edge is labeled with weight 1 if it is in E, and with weight $n + 1$ otherwise.
- The target weight $t = n$.

Having described the reduction, we need to prove two properties about it. Firstly, we need to show that our reduction is in polynomial time. It takes $O(n)$ time to count the number n of vertices and $O(n^2)$ time to create the weighted complete graph G', so our reduction is certainly in polynomial time.

Secondly, we need to show that our reduction is correct—that is, the resulting TSP in Graphs instance (G', n) always has the same answer as the original Hamiltonian Cycle instance G. We will do this by showing that a solution to either instance implies the existence of a solution to the other instance in both directions.

In one direction, suppose that there exists a solution to the Hamiltonian Cycle instance, that is, that G has a Hamiltonian cycle C. We can view C as a cycle in G' as well. Because C is Hamiltonian, it visits all n vertices exactly once. Thus C has length n and so it has weight n in G′, as it uses only edges in E. Cycle C visits every vertex at least once and has $weight \leq n$, so it is also a solution to the TSP in Graphs instance.

In the other direction, suppose that there exists a solution to the TSP in Graphs instance, that is, a cycle C in G' visits every vertex at least once and has $weight \leq n$. Because C visits every vertex at least once, it has $length \geq n$. On the other hand, C has $weight \leq n$ and thus $length \leq n$ because all weights are positive integers. So the cycle C must have length exactly n, which means it visits every vertex exactly once.

Finally, C must be a cycle in G because, if it included any edges not in E, then that edge alone would cause it to have weight at least $n + 1$ because weights are non-negative, a contradiction. Therefore, C is a Hamiltonian cycle on G, so it is also a solution to the Hamiltonian Cycle Problem instance.

Thus we reduced Hamiltonian Cycle problem to TSP.

Chapter 3

Polynomial solvable variants

3.1 Polynomial Algorithm

Polynomial algorithms are a type of algorithm that are designed to solve problems in polynomial time. These algorithms are particularly useful for solving large-scale optimization problems efficiently, as they ensure that the running time of the algorithm grows at most polynomially with the size of the problem instance.

Unlike exponential algorithms, which have a running time that grows exponentially with the size of the problem instance, polynomial algorithms are able to efficiently explore a large number of possible solutions by exploiting the structure of the problem. This allows them to find a near-optimal solution in a reasonable amount of time.

Polynomial algorithms rely on mathematical models and algorithms that are specifically designed to solve problems in polynomial time. For example, polynomial algorithms may use techniques such as dynamic programming, linear programming, or graph theory to efficiently explore possible solutions and find the best possible solution for the given problem instance. Overall, polynomial algorithms are a powerful tool for solving optimization problems efficiently and effectively.

3.2 Algorithm

3.2.1 Basic Idea

The algorithm give accurate results for all types of graphs, including perfect graphs, planar bipartite graphs, grid graphs, and 3-connected planar graphs. The implementation of the algorithm is presented in this paper.

Preprocessing is a crucial step in the algorithm, and it involves simplifying the input graph by removing parallel edges and self-loops before searching for a Hamiltonian circuit. After this, the algorithm checks for preprocessing conditions. If any of these conditions are met, it is determined that the graph does not have a Hamiltonian circuit.

The two preprocessing conditions are:

- No node should have a degree of 1.
- No node should have more than two adjacent nodes with a degree of 2.

If none of the preprocessing conditions are met, the algorithm proceeds with the Hamiltonian circuit search.

The implementation of these preprocessing conditions ensures that the algorithm runs efficiently and accurately on a wide range of graph types. After preprocessing main algorithm will be :

3.2.2 Algorithm

- While not all nodes have been processed, do the following:
  - If standing at a St-stack node and no unprocessed node is left, select the starting node as the next node.
  - If an adjacent node of degree 2 exists:
    - If more than one adjacent node of degree 2 exists, call Backtrack().
    - If only one adjacent node of degree 2 exists, select that node as the next node.
      - If the selected node is a St-stack node:
      - If there is only one St-stack node left and all other nodes are processed, select the node as the next node.
      - If there is only one St-stack node left and other nodes are unprocessed, call Backtrack().
  - If only one St-stack node is adjacent:
    - If the St-stack has more than one node, pick this node as the next node and remove it from the St-stack.
    - If the St-stack has only one node:
      - If all other nodes are processed, choose this node as the next node and remove it from the St-stack.
      - If other nodes are unprocessed, mark the connecting edge and continue.
  - If multiple St-stack nodes are adjacent, select the St-stack node with the least degree.
  - If no appropriate adjacent node is found, call Backtrack().
  - If the next node is not selected:
    - Select the node with the highest degree that is not yet visited.
    - If multiple nodes have the same highest degree, select the node with the highest degree and the least number of unvisited neighbors.
  - Mark all edges from the current node to adjacent nodes.

    - Decrement the degree of adjacent nodes (except the previous node).
    - Add the current node to the stack ”hamil”.
    - Rename the next node as the current node.
    - If all nodes are visited:
        * If the current node has an edge to the starting node, add the starting node to the stack ”hamil” and return ”Hamiltonian circuit found”.
        * If the current node does not have an edge to the starting node, call Backtrack().
- If no Hamiltonian circuit is found, return ”No Hamiltonian circuit exists in G”.

In this for finding a Hamiltonian Circuit in a graph is a polynomial time algorithm. The use of selection conditions and proper node selection reduces the chances of backtracking and makes the algorithm more efficient. The backtracking only occurs on junction nodes, and the algorithm prompts a message if no HC exists. The algorithm encounters blocking conditions if it makes a wrong decision on junction nodes, which minimizes the backtracking. Our algorithm provides a solution to the NP-Complete problem of finding a Hamiltonian Circuit in a graph in polynomial time.[1]

3.2.3 Complexity

Time Complexity:

The time complexity of the algorithm is $\mathcal{O}(n^3)$, as the algorithm traverses all the nodes and edges in the graph and performs constant time operations for each

Space Complexity:

The space complexity of the algorithm is also $\mathcal{O}(n^3)$, as the algorithm needs to store the graph, the stack for storing the Hamiltonian circuit, and the junction stack for storing junction nodes. However, in practice, the space complexity can be improved by using an adjacency list to represent the graph, which reduces the space complexity to $\mathcal{O}(n+m)$, where m is the number of edges in the graph.

Time Complexity: $\mathcal{O}(n^3)$

Space Complexity: $\mathcal{O}(n^3)$ (or $\mathcal{O}(n+m)$ with an adjacency list representation of the graph)

Chapter 4

Exact Algorithm

4.1 Exact Algorithm

An exact algorithm is a computational method that guarantees to find the optimal solution to a problem. That is, an exact algorithm will always return a solution that is guaranteed to be the best possible solution for the problem instance.

Exact algorithms are usually designed to solve optimization problems, such as finding the shortest path between two points, the maximum flow in a network, or the minimum cost to connect a set of nodes. These algorithms are typically based on mathematical models and involve the exhaustive exploration of all possible solutions, or a subset of solutions that is guaranteed to contain the optimal solution.

While exact algorithms provide a guaranteed optimal solution, they can be computationally expensive, particularly for large problem instances. Therefore, they are often used when the problem size is small or when the optimal solution is essential and cannot be approximated.

4.2 Algorithm 1 : Brute Force

4.2.1 Basic Idea

Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be n! (n factorial) configurations.

4.2.2 Algorithm

Algorithm 1 Brute Force Algorithm for Hamiltonian Cycle

```
procedure HamiltonianCycle($G$)
    $n \leftarrow$ number of vertices in $G$
    for $p \leftarrow 1$ to $n!$ do
        $perm \leftarrow$ permutation of vertices represented by $p$
        if $perm$ forms a Hamiltonian cycle in $G$ then
            return $perm$
        end if
    end for
    return "No Hamiltonian cycle exists in G"
end procedure
```

In this algorithm, $\boldsymbol{G}$ is the input graph, $\boldsymbol{n}$ is the number of vertices in $\boldsymbol{G}$, and $\boldsymbol{p}$ represents the permutation of vertices that we are testing which is different from previous permutations. The algorithm loops through all possible permutations of the vertices in $\boldsymbol{G}$ and checks if any of them form a Hamiltonian cycle. If a Hamiltonian cycle is found, the algorithm returns the corresponding permutation. If no Hamiltonian cycle is found after exploring all possible permutations, the algorithm returns a message indicating that no Hamiltonian cycle exists in $\boldsymbol{G}$.[2]

4.2.3 Complexity

Time Complexity:

The brute force algorithm for Hamiltonian cycle involves generating all possible permutations of the vertices, which takes $\boldsymbol{O(n!)}$ time, where n is the number of vertices in the input graph. Checking each permutation for a Hamiltonian cycle takes $\boldsymbol{O(n)}$ time, as we need to traverse the entire cycle once. Therefore, the total time complexity of the algorithm is $\boldsymbol{O(n! \times n)}$.

Space Complexity:

The space complexity of the algorithm is $\boldsymbol{O(n)}$, as we need to store the current permutation of vertices being tested and the input graph $\boldsymbol{G}$. The space required to store the permutation is proportional to the number of vertices in the graph, which is $\boldsymbol{n}$. Therefore, the space complexity of the algorithm is $\boldsymbol{O(n)}$.

Time Complexity: $\boldsymbol{O(n! \times n)}$

Space Complexity: $\boldsymbol{O(n)}$

4.3 Algorithm 2 : Backtracking Algorithm

4.3.1 Basic Idea

The basic idea of the backtracking algorithm to solve the Hamiltonian cycle problem is to recursively build a potential Hamiltonian cycle by exploring all possible paths in the graph. At each step, the algorithm selects an unvisited vertex and adds it to the current cycle. The algorithm then continues exploring the graph, adding more vertices to the cycle until either a complete Hamiltonian cycle is found, or it is determined that no Hamiltonian cycle exists.

If the current path cannot be extended to form a Hamiltonian cycle, the algorithm backtracks to the previous vertex and tries a different path. This process continues until either a Hamiltonian cycle is found or all possible paths have been explored without success.

The key to the backtracking algorithm is the use of a boolean array to keep track of which vertices have been visited in the current path. As the algorithm explores the graph, it marks each vertex as visited or unvisited as necessary. If the current path cannot be extended to form a Hamiltonian cycle, the algorithm unmarks the vertices and backtracks to the previous vertex, allowing for other potential paths to be explored.

The backtracking algorithm is an improvement over the brute force algorithm as it prunes the search space by avoiding exploring paths that cannot lead to a Hamiltonian cycle. However, it still has exponential time complexity in the worst case, making it impractical for large graphs. Nonetheless, it can be a useful method for smaller graphs or as a heuristic for larger graphs.

4.3.2 Algorithm

Algorithm 2 Backtracking Algorithm for Hamiltonian Cycle

```
1: procedure HamiltonianCycle($G$)
2:     $n \leftarrow$ number of vertices in $G$
3:     $visited \leftarrow$ array of size $n$ initialized to false
4:     $path \leftarrow$ empty list
5:     add vertex 1 to $path$
6:     $found \leftarrow$ HamiltonianCycleUtil($G$, $visited$, $path$, 1)
7:     if $found = true$ then
8:         return $path$
9:     else
10:        return "No Hamiltonian cycle exists in G"
11:    end if
12: end procedure
```

Algorithm 3 Hamiltonian Cycle Utility Function

```
procedure HamiltonianCycleUtil($G$, $visited$, $path$, $v$)
    $n \leftarrow$ number of vertices in $G$
    if $len(path) = n$ and $G_{v,1} = 1$ then
        return true
    end if
    $found \leftarrow$ false
    for $i \leftarrow 2$ to $n$ do
        if $G_{v,i} = 1$ and $visited_i = false$ then
            add vertex $i$ to $path$
            $visited_i \leftarrow$ true
            $found \leftarrow$ HamiltonianCycleUtil($G$, $visited$, $path$, $i$)
            if $found = true$ then
                return true
            end if
            remove vertex $i$ from $path$
            $visited_i \leftarrow$ false
        end if
    end for
    return false
end procedure
```

In this algorithm, $\boldsymbol{G}$ is the input graph, $\boldsymbol{n}$ is the number of vertices in $\boldsymbol{G}$, $\boldsymbol{visited}$ is the boolean array indicating which vertices have been visited in the current path, and $\boldsymbol{path}$ is the list of vertices in the potential Hamiltonian cycle. The algorithm begins by initializing the variables and calling the utility function with the starting vertex (in this case, vertex 1). The utility function recursively explores all possible paths from the starting vertex, marking visited vertices as it goes. If a complete Hamiltonian cycle is found, the function returns true, indicating success. Otherwise, it backtracks to the previous vertex and tries a different path.[2]

4.3.3 Complexity

Time Complexity:

Time complexity is $\boldsymbol{\mathcal{O}(n!)}$, where $\boldsymbol{n}$ is the number of vertices in the input graph. This is because the algorithm explores all possible permutations of the vertices in the worst case.

Space Complexity:

Space complexity is $\mathcal{O}(n)$, where n is the number of vertices in the input graph. This is because the algorithm uses a boolean array of size n to keep track of which vertices have been visited in the current path, and a list of size n to store the potential Hamiltonian cycle. The recursive call stack also contributes to the space complexity, but the maximum depth of the call stack is n in the worst case, so the overall space complexity is still $\mathcal{O}(n)$.

Time Complexity: $O(n!)$

Space Complexity: $O(n)$

Chapter 5

Approximation Algorithm

5.1 Approximation Algorithm

An approximation algorithm is a way of dealing with NP-completeness for an optimization problem. This technique does not guarantee the best solution. The goal of the approximation algorithm is to come as close as possible to the optimal solution in polynomial time. Such algorithms are called approximation algorithms. For example, for the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.

α-approximation algorithm: An α-approximation algorithm for an optimization problem is a polynomial-time algorithm that for all instances of the problem produces a feasible solution whose value is within a factor of α of the value of an optimal solution.
As a convention, we consider $\alpha \geq 1$. Then the calculation of α for maximization problem and minimization problem will be different. For maximization problem,

$$\alpha = \frac{\text{Optimal Solution}}{\text{Obtained Solution}} \tag{5.1}$$

And for minimization problem,

$$\alpha = \frac{\text{Obtained Solution}}{\text{Optimal Solution}} \tag{5.2}$$

5.2 A 2-approximation algorithm

5.2.1 Basic Idea

We can devise an approximation algorithm for a similar problem ”traveling salesman problem” and use it for the hamiltonian path problem.

Problem: Given complete, undirected graph $G = (V, E)$ with non-negative integer cost $c(u, v)$ for each edge, find the cheapest hamiltonian cycle of G.

Here can be two cases: with and without triangle inequality. c satisfies triangle inequality if it is always cheapest to go directly from some u to some w; going by way of intermediate vertices can't be less expensive. This algorithm can solve TSP with triangle inequality with an approximation ratio of 2. As, hamiltonian cycle follows triangle inequality, we can use this algorithm for finding hamiltonial cycle too.

5.2.2 Algorithm

We use function $\mathbf{MST\text{-}PRIM(G, c, r)}$, which computes an MST for G and weight function c, given some arbitrary root r.

Algorithm 4 Approx-TSP-Tour

```
Input:
$G = (V, E), c : E \to \mathbf{R}$
Steps:
1: Select arbitrary $v \in V$ to be “root”
2: Compute MST $T$ for $G$ and $c$ from root $r$ using $\mathbf{MST\text{-}PRIM(G, c, r)}$
3: Let $L$ be a list of vertices visited in a pre-order tree walk of $T$
4: Return the hamiltonian cycle that visits the vertices in the order $L$
```

5.2.3 Complexity and Approximation Ratio

Running Time

Simple MST-PRIM takes $\Theta(V^2)$, as computing preorder walk takes no longer. So, the running time is **polynomial**.

Approximation Ratio

This algorithm has an approximation ratio of 2.

Proof: Let H^* denote an optimal tour for a given set of vertices. Deleting any edge from H^* gives a spanning tree.
Thus, the weight of the minimum spanning tree is lower bound on the cost of the optimal tour,

$$c(T) \le c(H^*)$$

A full walk of T lists vertices when they are first visited, and also when they are returned to, after visiting a subtree.

Ex: a, b, c, b, h, b, a, d, e, f, e, g, e, d, a

Full walk W traverses every edge exactly twice (although some vertex perhaps way more often), thus,

$$c(W) = 2c(T)$$

Together with $c(T) \leq c(H^*)$, this gives,

$$c(W) = 2c(T) \leq 2c(H^*)$$

Chapter 6

Randomization

6.1 Randomization

Randomization in solving NP-complete problems involves the use of randomized algorithms to try to find solutions to problems that are believed to be intractable using deterministic algorithms.
Randomized Algorithm: A *randomized algorithm* can be defined as one that receives, in addition to its input, a stream of random bits that it can use in the course of its action for the purpose of making random choices.
Randomized algorithms for solving NP-complete problems typically work by making random choices during the algorithm's execution. These choices are usually made in a way that is probabilistically biased towards good solutions. By making many random choices and combining the results in a clever way, the algorithm can sometimes find a solution that is close to the optimal solution.Randomization is not a guaranteed way to find optimal solutions to NP-complete problems, but it can be a useful tool for finding good solutions when deterministic algorithms are not feasible or are computationally expensive. In this chapter, we will look into two basic randomized algorithms for Hamiltonian Cycle Problem.

6.2 Algorithm 1

6.2.1 Basic idea

This algorithm works by choosing random permutation of vertices in the input graph and checking if the random permutation is a Hamiltonian cycle.

6.2.2 Work Principle

The steps of the algorithm are given below-

1. Choose a random permutation of the vertices of the graph.
2. For each pair of adjacent vertices in the permutation, check if there is an edge between them. If any pair of adjacent vertices in the permutation is not connected by an edge, then the permutation is not a Hamiltonian cycle.

3. If all pairs of adjacent vertices in the permutation are connected by an edge, then the permutation is a Hamiltonian cycle.

6.2.3 Run time and success probability

For one iteration, the running time is polynomial $\boldsymbol{O(n^2)}$ but the success probability is at least $\boldsymbol{1/n!}$ where n is the number of vertices in the graph. So, as the size of the graph increases, the probability of success decreases rapidly. One way to improve the success probability of the algorithm is to repeat it multiple times with different random choices and combine the results to obtain a better solution. Usually, the number of repetition is such that the run time is polynomial and at the same time the success probability is reasonable.

6.3 Algorithm 2

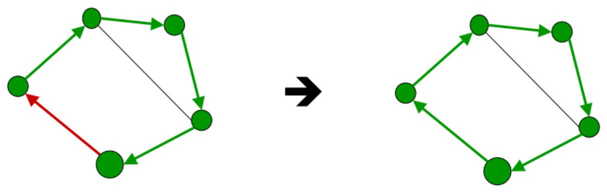
6.3.1 Basic idea

The algorithm is like the game snake/nibbles and make use of a simple operation called **rotation**. We start by picking a random vertex in the graph as the head (of the snake). Then, repeatedly, we choose an adjacent edge of the head. The snake "eats" this edge and updates the position of the head. When the snake tries to eat itself, we either get a Hamiltonian cycle or perform a rotation.

6.3.2 Work Principle

The steps of the algorithm are given below-

1. Choose a random vertex as the head of the snake.
2. Choose an adjacent edge of the head. The snake "eats" this edge and updates the position of the head.
3. Let $P = v_1, , v_2, ..., v_k$ be the current snake, v_k = head of snake. Suppose the snake now eats the edge (v_k, v_j) That is, the snake eats itself.
    - If $v_j = \text{tail} = v_1$ and if all vertices are explored, we obtain an Hamiltonian cycle



    - Otherwise, we change perform rotation to change the snake into: $P = v_1, v_2, ..., v_{j-1}, v_j, v_k, v_{k-1}, v_{k-2}, ..., v_{j+2}, v_{j+1}$
4. When the snake does not have any edge to eat next (before a Hamiltonian cycle is found), we stop and report "FAIL".

6.3.3 Pseudocode

Suppose $P = v_1, v_2, ..., v_k$ is current path, v_k = head Now, let us define three subroutines first and then provide the full algorithm

```
IsHamilton(P, v) {
    1. If v = $v_1$ and all vertices are visited, return TRUE
    2. Else, return FALSE
}

Rotate(P, j) {
    1. Update head to $v_{j+1}$
    2. Update $P = v1, v_2, ..., v_j, v_k, v_{k-1}, ..., v_{j+2}, v_{j+1}$
}

Extend(P, v) {
    1. Update the head to v
    2. Update $P = v_1, v_2, ..., v_k, v$
}
```

Algorithm 5 Snake-Nibble Algorithm

```
Input:
$G = (V, E), |V| = n$
Steps:
(i): Choose a random node as head
(ii): Repeat until no unused edge is adjacent to head
    1. Let $P = v_1, v_2, ..., v_k$ where $v_k$= head
    2. Choose an unused edge adjacent to $v_k$ ,say $(v_k, v)$, uniformly at random.
       Mark this edge as used
        (a) If IsHamilton(P,v), DONE
        (b) Else, if v is unvisited, Extend(P, v)
        (c) Else, $v = v_j$ for some j. Rotate(P, j)
(iii) If there is no edge left to "eat", return "No Hamiltonian Cycle"
```

6.3.4 Success probability

With some modifications in the given algorithm, it can be proved that for sufficiently large n, the algorithm can find a Hamiltonian cycle in a random graph, $G_{n,p}$, whenever $\boldsymbol{p >= (40lnn)/n}$. This immediately implies that when p is at least $(40lnn)/n$, a random graph, $G_{n,p}$ will contain a Hamiltonian cycle.

Chapter 7

Heuristics

7.1 Heuristics

Heuristic algorithms are problem-solving methods that utilize practical approaches to finding solutions. These methods do not guarantee optimal solutions, but they often provide good approximations that can be found more efficiently than exact methods.

Heuristics can be applied to a wide range of problems, including optimization, decision-making, and search problems. They are especially useful in situations where the problem is too complex to be solved using exact methods or when the solution needs to be found quickly. One of the key features of heuristic algorithms is their ability to learn from experience. They use past performance data to improve their performance and refine their approach over time. This makes heuristic algorithms highly adaptable to changing problem conditions and environments.Some popular heuristic algorithms include simulated annealing, genetic algorithms, ant colony optimization, and tabu search. Each of these algorithms has its own unique strengths and weaknesses, and the choice of algorithm often depends on the specific problem being addressed.

Overall, heuristic algorithms provide a powerful and flexible tool for solving complex problems that are difficult or impossible to solve using exact methods.

7.2 Algorithm

7.2.1 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. [3]

The algorithm involves two steps:

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

7.2.2 Work Principle

The algorithm involves two steps:

- **Step 1:** The heuristic information acquisition step.
- **Step 2:** The heuristic search step.
  - if no such vertex found at any point, then return ”No Hamiltonian cycle”
  - else return the Hamiltonian cycle.

7.2.3 Our Python Implementation Code

```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)

    # Iterate n times
    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]:
                # Skip if starting node(0)
                if(j==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



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 (possibl
                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
```

7.2.4 Example analysis

The time complexity of the algorithm is related to the size and structure of the input directed graph. Therefore, this paper chooses three concrete examples to analyze the difference between the heuristic search algorithm and the non-information backtracking method for the Hamiltonian circuit problem. Because the time complexity of the heuristic information generation algorithm is polynomial, this paper compares the number of nodes that need to be processed by the algorithm as the benchmark. A directed graph of the Tutte subgraph is shown in Fig. 1, in which the vertex labels are shown as a, b, c and so on, and the number of vertices $n = 15$. Extend from the starting point a (path length value is 0) until the extended path length value is $n - 1$, and the extended length list of each vertex is given after the vertex label. Check whether the vertex with the path length $n - 1 = 14$ has a directed edge to the starting point a, i.e., if a node d with path length value 14 and directed edge $< d, a >$ exists. Search the vertex sequences with path lengths $n-1, n-2, \ldots, 1, 0$ in order from the starting point in the reverse direction, that is, $a(15) \to d(14) \to i(13) \to n(12) \to m(11) \to l(10) \to j(9) \to e(8) \to f(7) \to p(6) \to k(5) \to h(4) \to g(3) \to c(2) \to b(1) \to a(0)$. The reverse order of this sequence constitutes a Hamiltonian circuit. As shown in Fig. 2(a, b), the directed graph and its extended length list, $n = 8$, can return to the starting point in extension step (8). However, there are no Hamiltonian circuits in graph (a). While searching from vertex a according to the depth-first algorithm with heuristic information, we obtain the sequence $a(8) \to f(7) \to d(6) \to g(5) \to e(4) \to c(3)$. At this point, only $d(2)$ can meet the requirements of vertex c, but $d(2)$ has appeared in the existing path sequence; thus, we can only backtrack to vertex a. Then, no vertices can be searched, so there is no Hamilton circuit in Fig. 2(a).

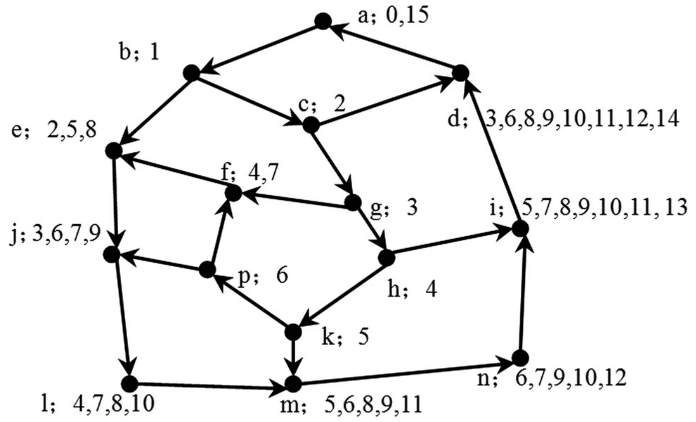


Figure 7.1: A directed graph of the Tutte subgraph

However, for Fig. 2(b) a Hamiltonian circuit does exist; while searching from vertex a according to depth-first algorithm with heuristic information, we obtain the sequence $a(8) \to f(7) \to d(6) \to g(5) \to e(4) \to h(3) \to c(2) \to b(1) \to a(0)$, or the other sequence $a(8) \to f(7) \to g(6) \to e(5) \to h(4) \to c(3) \to d(2) \to b(1) \to a(0)$. The reverse

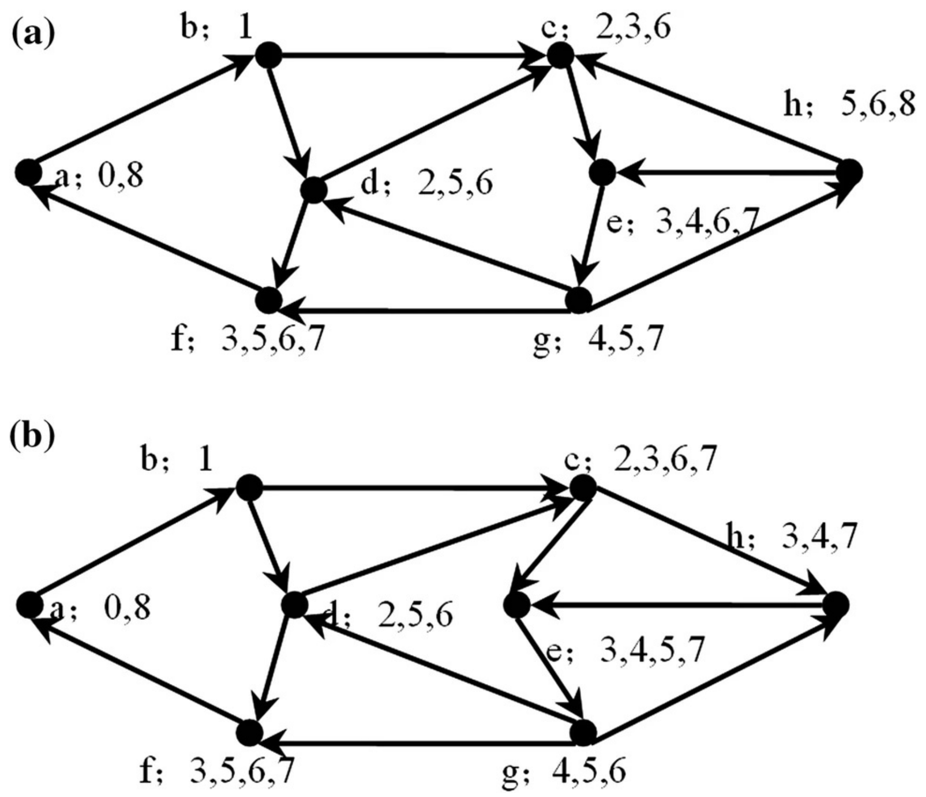


Figure 7.2: A directed graph with loops

7.2.5 Complexity Analysis

The storage method for a directed graph in the computer mainly includes the adjacency matrix and the adjacency table. With a different storage structure, the time complexity of the algorithm is not very different.

In this paper, we use the adjacency matrix in algorithm 2 for acquiring heuristic information. For n extension steps, the number of vertices for each step is $O(1)$ in the best case and $O(n)$ in the worst case. Thus, when storing using an adjacency matrix, the time complexity is $O(n^2)$ in the best case and $O(n^3)$ in the worst case. Thus, the heuristic information is obtained with polynomial time complexity.

In the heuristic search algorithm (algorithm 3) of the Hamiltonian circuit problems, since the extended path length value set is stored in an $n \times n$ matrix, the number of searches for the precursor nodes of each vertex in step (4) is generally $O(n)$. In the best case, one search could locate a simple circuit, and the time complexity is $O(n^2)$. In the worst case, the algorithm needs to traverse the whole graph, and the time complexity is $O(n!)$.

Time complexity : Best Case: $\Omega(n^2)$ **Worst Case:** $O(n!)$

Space complexity : $O(n^2)$

Chapter 8

Meta Heuristic

8.1 Meta-heuristics

A **metaheuristic** is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms. In other words, a **metaheuristic algorithm** is a search procedure designed to find, a good solution to an optimization problem that is complex and difficult to solve to optimality. Genetic algorithm is one kind of meta heuristic algorithm. Here, we will be using an improved version of the genetic algorithm that uses Dijkstra's algorithm and solve hamiltonian cycle problem. This algorithm is named as **IGA** (Improved Genetic Algorithm).[4]

8.2 Improved genetic algorithm

8.2.1 Basic Idea

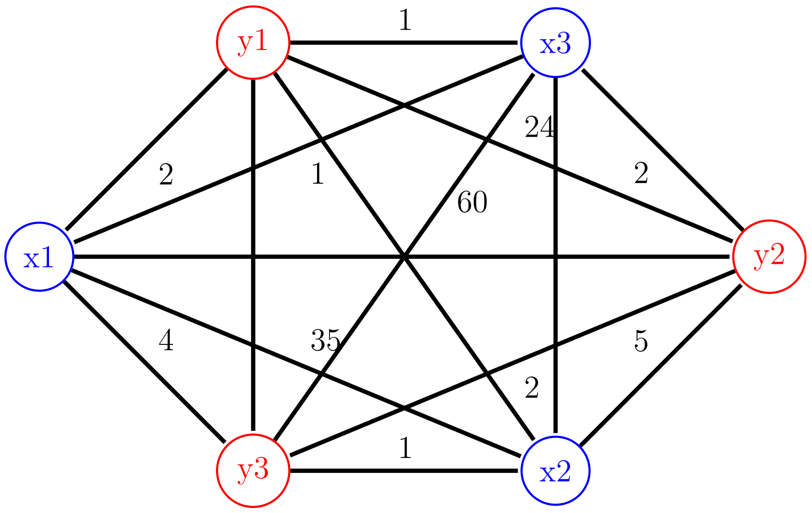
Suppose, there is a superimposed graph $\mathbf{G((X \cup Y), E)}$. This graph contains a set of significant vertices $\mathbf{|X| = n}$ and a set of non-significant vertices $\mathbf{|Y| = m}$.
In the algorithm 2, we can see that we have the start vertex **V0** (which is also the finish vertex in the Hamiltonian cycle), a square symmetric matrix **M** which represents the superimposed graph, and the **L** is a list of non-significant vertices in the matrix.
Two sets of vertices (visited and non-visited) are used to form the superimposed graphs. Only the significant vertices will appear in the final solution. So, a local research method is introduced to find the shortest path between two significant vertices, if it exists, using non-significant vertices. For the fitness function, we used the Dijkstra's algorithm to find the best fitness cost.
The **IGA** uses the roulette wheel selection method. The One point crossover is used as a crossover method.

Initial population and chromosome structure

The initial population is a randomly generated chromosome set. It respects the constraints of the **CSP** formalism. These constraints are applied only on the significant vertices **X** :

1. **Constraint 1** : We should never visit the significant vertices more than one time **Alldiff**$(\boldsymbol{X_i})$.
2. **Constraint 2** : We should start and finish the cycle in the same vertex $\boldsymbol{X_1 = X_{n+1}}$.
3. **Constraint 3** : We must visit all the significant vertices.

The size of the chromosome is equal to the size of the total number of the significant vertices. We should note that, an individual is a complete demonstration of the problem's variables. Its genes represent the passing order of the vertices in the superimposed graph.
The example in figure below present a superimposed graph with $\boldsymbol{x_1, x_2, x_3}$ a set of significant vertices and $\boldsymbol{y_1, y_2, y_3}$ a set of non-significant vertices.



Example of shortest Hamiltonian cycle

Here, The best chromosome could be equal to the array : 1,2,3,1. So, the shortest path is: $\boldsymbol{x_1, x_2, x_3, x_1}$.

Fitness function and Dijkstra

We define an augmented fitness function **F** that will be used by the optimization process for a given **CSOP** (V, D, C, F) modeling the shortest Hamiltonian problem in an agglomeration of superimposed graphs. The objective function for potential solution is be defined as :

$$\boldsymbol{F = min \sum_{i=1}^{n+1} Cost(shortest_path(X_i, X_{i+1}))} \tag{8.1}$$

Here,

- **Cost()** : is a function that calculates the cost or the distance between two vertices $\boldsymbol{X_i} \in \boldsymbol{X}$ and $\boldsymbol{X_{i+1}} \in \boldsymbol{X}$ already presented in the matrix.
- **shortest__path** : is a function that calculate the shortest path between $\boldsymbol{X_i} \in \boldsymbol{X}$ and $\boldsymbol{X_{i+1}} \in \boldsymbol{X}$, passing throw a set function of non-significant vertices $\boldsymbol{Y = \{Y_1, Y_2, Y_3, ..., Y_m\}}$

$$\boldsymbol{Shortest\_Path(X_i, X_j) = Dijksra\_shortest\_path((X_i, X_j), \{\{X_i, X_j\} \cup Y\})} \tag{8.2}$$

This equation uses the Dijkstra's algorithm to find the shortest path between $\boldsymbol{X_i} \in \boldsymbol{X}$ and $\boldsymbol{X_{i+1}} \in \boldsymbol{X}$. Dijkstra algorithm is one of the well-known algorithms in solving shortest path problem. As a solution it gives the shortest path between the significant vertices $\boldsymbol{X_s}$ and $\boldsymbol{X_f}$ passing throw a set of non- significant vertices only if this equation is satisfied.

$$\boldsymbol{\sum_{k=1}^{m} Cost(X_s, Y_k) + Cost(Y_k, X_f) < Cost(X_s, X_f)} \tag{8.3}$$

This equation compares the cost between the given vertices $\boldsymbol{X_s}$ and $\boldsymbol{X_f}$, passing by a set of non-significant vertices.

Input :

- **Weighted graph** : The cost matrix **(M)**.
- $\boldsymbol{X_s}$: A Start significant vertex.
- $\boldsymbol{X_f}$: A finish significant vertex.

Output :

An array that presents the shortest path that includes a Start significant vertex, a finish significant vertex and (if possible) one or more non -significant vertex.

As the equation shows, two scenarios exist, after calculating the shortest path we compare both costs. If the new cost, including the new path, is higher than the initial cost, nothing will happen. Else, we take the new value as the cost between $\boldsymbol{X_i}$ and $\boldsymbol{X_{i+1}}$.

Selection

In this phase of the algorithm, we use roulette wheel selection method. The selection of every chromosome in the genetic algorithm is done by a probability $\boldsymbol{P_i}$.

$$\boldsymbol{P_i = \frac{F'(ch_i)}{\sum_{j=1}^{n} F'(ch_j)}} \tag{8.4}$$

With $\boldsymbol{F'(ch_i)}$ is the fitness function and **n** is the size of the initial population. The new survivors will be recombine and mutate to create another generation.

Crossover method

There exist various crossover techniques with different data structures and different principles. In this case, we use a one-point crossover method. From each two chromosomes that are considered as parents, we select only one point, and we swipe the genes from one to another to create two offspring.

Mutation

Mutation is applied to each child individually after crossover. It randomly alters each gene with a small probability. Mutation only affects the genes that represent the non-significant vertices.

8.2.2 Algorithm

Algorithm 6 Improved Genetic Algorithm

```
Input :

V0 : Start vertex
M : Cost matrix (n+m)*(n+m)
L : A list of non-significant vertices in the matrix

Output : Shortest Hamiltonian circuit presented as an array of vertices, table
with size (n).

1. Generate the initial population
2. Repeat while the termination condition is not reached
3. {
4. Evaluation of fitness value of chromosomes by calculating objective function of
the significant and non-significant vertices using Dijkstra's algorithm;
5. Chromosomes selection;
6. Crossover;
7. Mutation;
8. }
9. Return Solution (Best Chromosomes)
```

Chapter 9

List of open problems

There are several list of open problems for hamiltonian cycle problem. Some of the open problems are as follows :

1. Is there a polynomial time algorithm to solve the Hamiltonian cycle problem in general graphs?
2. Can the performance of existing approximation algorithms for the Hamiltonian cycle problem be improved?
3. Is it possible to develop heuristics for the Hamiltonian cycle problem that work well on large graphs and have provable performance guarantees?
4. What is the computational complexity of the Hamiltonian cycle problem in restricted graph classes, such as chordal graphs, interval graphs, or permutation graphs?
5. Can the Hamiltonian cycle problem be solved efficiently in sparse graphs or graphs with certain structural properties?
6. What is the relationship between the Hamiltonian cycle problem and other problems in graph theory and combinatorics, such as the graph isomorphism problem, the matroid intersection problem, or the polyomino packing problem?
7. Are there new techniques from other areas of mathematics or computer science that can be applied to the Hamiltonian cycle problem to develop new algorithms or insights?

Note that this is not an exhaustive list and there are many other open problems related to the Hamiltonian cycle problem.

Chapter 10

Conclusion

In conclusion, the Hamiltonian cycle problem is a classic NP-complete problem that has been extensively studied in computer science and mathematics. It is the problem of finding a cycle that visits every vertex exactly once in a given graph. Variations of the problem include Hamiltonian path problem and the Traveling salesman problem. Reductions have been developed from a NP-complete problem(**3-SAT**) to the **Hamiltonian cycle problem** and from **Hamiltonian cycle problem** to another NP-complete problem(**Traveling salesman problem**).

Polynomial algorithms have been developed to efficiently solve optimization problems, such as the Hamiltonian cycle problem, in polynomial time by using mathematical models and techniques like dynamic programming, linear programming, or graph theory. Special cases, like the bipartite graph Hamiltonian cycle problem and the planar graph Hamiltonian cycle problem, have been solved exactly in polynomial time. However, the general problem remains NP-complete and requires exponential time in the worst case for exact solutions. Polynomial algorithms exploit the problem's structure to find solutions more efficiently than exponential algorithms, making them a promising area of research for solving the Hamiltonian cycle problem.

Various approximation algorithms, randomized algorithms, heuristics, and metaheuristics have been developed to solve the Hamiltonian cycle problem in practice, with varying levels of success. These algorithms are often used in applications such as routing and logistics planning.

Despite decades of research, the Hamiltonian cycle problem remains an active area of study, with many open problems still to be solved. Some of these open problems include finding better approximation algorithms, developing faster heuristics for large graphs, and exploring the connections between the Hamiltonian cycle problem and other areas of mathematics and computer science. Overall, the Hamiltonian cycle problem is an important and challenging problem that continues to inspire research and innovation in computer science and mathematics.

Bibliography

[1] M. R. Garey, D. S. Johnson, and R. E. Tarjan, "The planar hamiltonian circuit problem is np-complete," *SIAM Journal on Computing*, vol. 5, no. 4, pp. 704–714, 1976.

[2] "Hamiltonian cycle using exact algorithm." `https://www.geeksforgeeks.org/hamiltonian-cycle/`. Accessed: 2023-03-03.

[3] D. Jin, Q. Li, and M. Lu, "A heuristic search algorithm for hamiltonian circuit problems in directed graphs," *Wireless Networks*, pp. 1–11, 2022.

[4] S. B. Khaoula BOUAZZI, Moez HAMMAMI, "Application of an improved genetic algorithm to hamiltonian circuit problem," *Procedia Computer Science*, vol. 192, pp. 4337–4347, 2021.