**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

**Constructing Valid Arrangements of Sequential Pairs**

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

**BACHELOR OF ENGINEERING**

**IN COMPUTER SCIENCE ENGINEERING**

**Submitted by**

**D SAKTHI SARAVANAN**

**Under the Supervision of**

**Dr. K.V. Kanimozhi**

**SEPTEMBER 2024**

**DECLARATION**

I am D. **Sakthi Saravanan**, student of **Computer Science Engineering**, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled is **Constructing Valid Arrangements of Sequential Pairs** the outcome of our own bonafide work and is correct to the best of our knowledge and this work has been undertaken taking care of Engineering Ethics.

(D. Sakthi Saravanan 192210553)

**CERTIFICATE**

This is to certify that the project entitled **“Constructing Valid Arrangements of Sequential Pairs”** submitted by **D. Sakthi Saravanan** has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B.E Computer Science Engineering.

Teacher-in-charge

Dr. K.V. Kanimozhi

**Abstract**

The problem at hand involves arranging a list of integer pairs in such a way that each consecutive pair in the arrangement shares a matching end and start value. Specifically, for each index iii where 1≤i<pairs.length1 \led if < \text {pairs. length} 1≤i<pairs.length, the end value of the previous pair must equal the start value of the current pair. This requirement establishes a sequence that can be visualized as a directed graph, where each pair represents a directed edge between two vertices.

In this context, the challenge of finding a valid arrangement can be likened to the problem of locating a Eulerian path within this directed graph. A Eulerian path is defined as a trail that visits every edge exactly once, making it an ideal framework for solving our arrangement problem. The existence of a valid arrangement is guaranteed by the constraints of the problem, ensuring that the necessary conditions for a Eulerian path—such as balancing in-degrees and out-degrees for each vertex—are inherently satisfied.

To implement a solution, we employ Hierholzer's algorithm, a well-known method for finding Eulerian paths in directed graphs. This algorithm operates by starting at a designated vertex (or start node) and recursively traversing the graph, following each edge until returning to the starting point. During this traversal, edges are marked as visited, ensuring that each is included in the final path exactly once. By carefully managing the edges and utilizing a stack to backtrack, when necessary, we construct the desired arrangement of pairs.

The solution not only guarantees a valid arrangement but also efficiently constructs it, leveraging the properties of Eulerian paths to ensure that all pairs are connected correctly, adhering to the specified requirements.

**Introduction**

In graph theory, the task of finding a valid arrangement of pairs can be effectively conceptualized as the problem of identifying an Eulerian path. In this framework, each pair is treated as a directed edge connecting two vertices, where the first element of the pair serves as the starting vertex and the second element as the ending vertex. The primary objective is to traverse each edge exactly once, ensuring that the end of one pair corresponds to the start of the next pair in the sequence.

Given the nature of the problem, it is stipulated that a valid arrangement exists, which simplifies certain aspects of the solution. The challenge lies in accurately identifying and constructing this Eulerian path by leveraging essential graph properties, including the in-degrees and out-degrees of the nodes involved. In a directed graph, an Eulerian path can exist if, at most, one vertex has an out-degree that exceeds its in-degree by one (potentially the starting vertex), while at most one vertex can have an in-degree that exceeds its out-degree by one (potentially the ending vertex). All other vertices should have equal in-degrees and out-degrees. This balance is crucial for the continuity of traversal and ensures that every edge can be visited without retracing steps.

The construction of the Eulerian path is typically achieved through algorithms such as Hierholzer's algorithm. This approach facilitates the systematic exploration of the graph, allowing for the traversal of edges while maintaining the necessary conditions for a valid arrangement. By employing a depth-first search strategy, the algorithm navigates through the graph, marking edges as visited and utilizing backtracking when necessary to explore alternative paths.

Beyond its theoretical significance, this problem has numerous practical applications in various domains, such as routing, scheduling, and resource allocation. In routing applications, for example, maintaining continuity of connections is essential for optimizing travel paths in logistics and transportation networks. Similarly, in scheduling contexts, ensuring that tasks are carried out in a specific sequence can enhance efficiency and minimize delays. By applying graph-theoretic principles to these real-world scenarios, we can design more effective solutions that adhere to continuity requirements while maximizing resource utilization.

**Coding**

#include <stdio.h>

#include <Sliabh>

typedef struct {

int start;

int end;

} Pair;

typedef struct Node {

int value;

struct Node\* next;

} Node;

typedef struct {

int key;

int value;

} HashMap;

Node\* create Node (int value) {

Node\* new Node = (Node\*) malloc (size of (Node));

new Node->value = value;

new Node->next = NULL;

return new Node;

}

void add Edge (Node\*\* adjacency List, int index, int value) {

Node\* new Node = create Node(value);

new Node->next = adjacency List[index];

adjacency List[index] = new Node;

}

int find Key (HashMap\* HashMap, int size, int key) {

for (int i = 0; i < size; ++i) {

if (HashMap[i].key == key) {

return i;

}

}

return -1;

}

void valid Arrangement (Pair\* pairs, int pairs Size) {

int i, j, index;

int map Size = 0;

int start Node = pairs [0]. start;

HashMap map [100];

Node\* adjacency List [100] = {NULL};

int indegree [100] = {0}, outdegree [100] = {0};

int path [100], path Size = 0;

for (i = 0; I < pairs Size; ++i) {

index = find Key (map, map Size, pairs[i]. start);

if (index == -1) {

map [map Size].key = pairs[i]. start;

map [map Size].value = map Size;

index = map Size;

map Size++;

}

add Edge (adjacency List, index, pairs[I]. end);

outdegree[index]++;

index = find Key (map, map Size, pairs[I]. end);

if (index == -1) {

map [map Size].key = pairs[I]. end;

map [map Size].value = map Size;

index = map Size;

map Size++;

}

indegree[index]++;

}

for (I = 0; I < map Size; ++I) {

if (outdegree[I] - indegree[I] == 1) {

start Node = map[I].key;

break;

}

}

int stack [100], stack Size = 0;

stack [stack Size++] = start Node;

while (stack Size > 0) {

int current Node = stack [stack Size - 1];

index = find Key (map, map Size, current Node);

if (adjacency List[index]! = NULL) {

stack [stack Size++] = adjacency List[index]->value;

Node\* temp = adjacency List[index];

adjacency List[index] = adjacency List[index]->next;

free(temp);

} else {

path [path Size++] = stack [--stack Size];

}

}

for (I = path Size - 1; I > 0; --I) {

print ("[%d, %d] ", path[I], path [I - 1]);

}

print("\n");

}

int main () {

Pair pairs [] = {{5, 1}, {4, 5}, {11, 9}, {9, 4}};

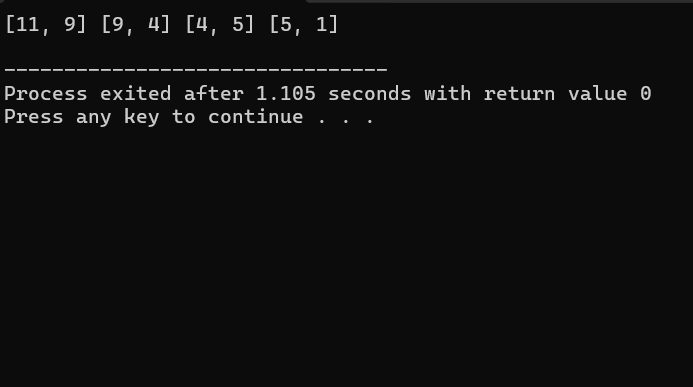
int pairs Size = size of(pairs) / size of (pairs [0]);

valid Arrangement (pairs, pairs Size);

return 0;

}

**Output**



**Complexity Analysis**

**1.Graph Construction (Best, Average, and Worst Cases):**

**Time Complexity: O(N),** where N is the number of pairs. Each pair is processed exactly once to construct the graph and update the degree counts for each node.

**Space Complexity: O(N),** for storing the graph structure using an adjacency list and mappings for in-degrees and out-degrees.

**2. Finding Start Node (Best, Average, and Worst Cases):**

**Time Complexity: O(N),** iterating over nodes to find the starting node based on the difference between out-degrees and in-degrees.

**Space Complexity: O(1),** as no additional space beyond the degree mappings is required.

**3. Hierholzer’s Algorithm (Eulerian Path) (Best, Average, and Worst Cases):**

**Time Complexity: O(N),** since each edge (pair) is traversed exactly once while constructing the Eulerian path.

**Space Complexity: O(N)**, storing the resultant path and the recursion stack for backtracking.

**Overall Complexity:**

**Time Complexity: O(N)**

**Space Complexity: O(N)**

**Conclusion**

In conclusion, the problem of finding a valid arrangement of pairs can be elegantly framed within the context of graph theory as the search for an Eulerian path. By representing pairs as directed edges in a graph, we establish a framework that allows us to traverse connections in a manner that ensures continuity between consecutive pairs. The existence of a valid arrangement, guaranteed by the problem's constraints, simplifies the task of identifying this path.

Utilizing Hierholzer's algorithm, we can efficiently construct the desired arrangement while adhering to the necessary conditions of node degrees. This approach not only highlights the beauty of mathematical principles in solving complex problems but also has practical implications across various fields, including logistics, scheduling, and resource management. By understanding and applying these concepts, we can optimize processes that rely on maintaining sequence and connectivity, ultimately leading to more efficient and effective solutions in real-world applications.

The intersection of graph theory and practical problem-solving underscores the value of mathematical models in addressing everyday challenges, paving the way for further exploration and innovation in algorithmic design and implementation.