Problem 1786: Number of Restricted Paths From First to Last Node

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

There is an undirected weighted connected graph. You are given a positive integer

n

which denotes that the graph has

n

nodes labeled from

1

to

n

, and an array

edges

where each

edges[i] = [u

i

, v

i

, weight

i

]

denotes that there is an edge between nodes

u

i

and

v

i

with weight equal to

weight

i

.

A path from node

start

to node

end

is a sequence of nodes

[z

0

, z

1

,

z

2

, ..., z

k

]

such that

z

0

= start

and

z

k

= end

and there is an edge between

z

i

and

z

i+1

where

0 <= i <= k-1

.

The distance of a path is the sum of the weights on the edges of the path. Let

distanceToLastNode(x)

denote the shortest distance of a path between node

n

and node

x

. A

restricted path

is a path that also satisfies that

distanceToLastNode(z

i

) > distanceToLastNode(z

i+1

)

where

0 <= i <= k-1

.

Return

the number of restricted paths from node

1

to node

n

. Since that number may be too large, return it

modulo

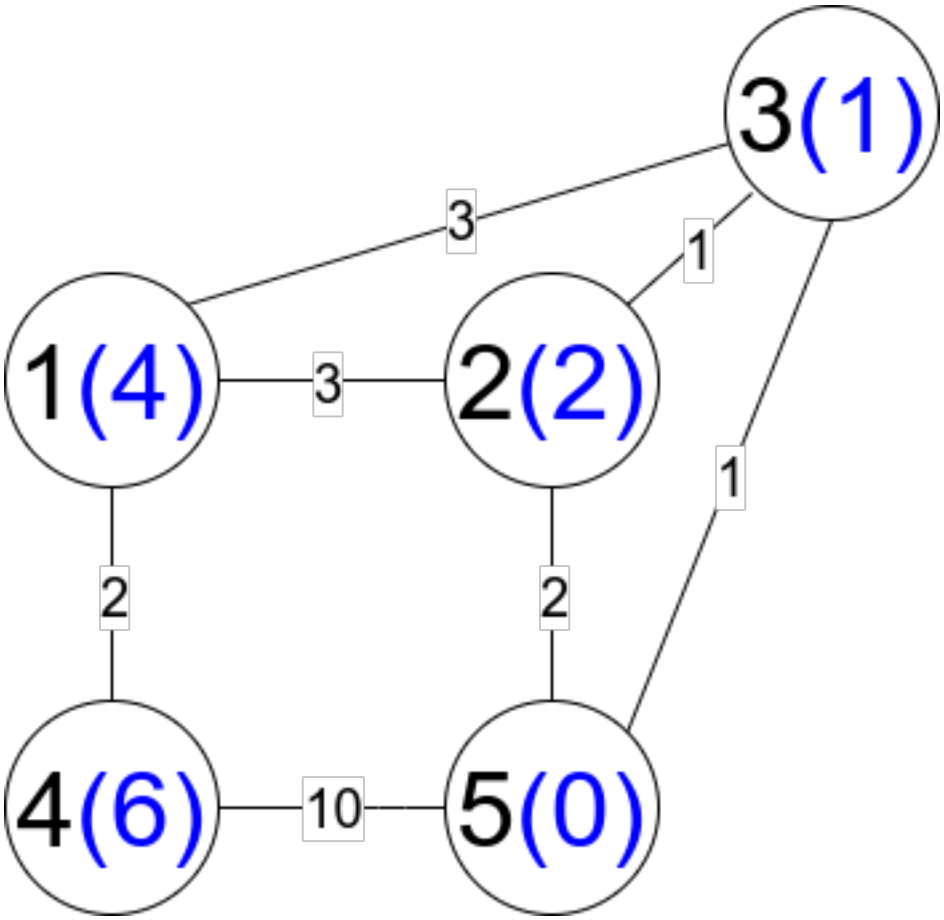
10

9

+ 7

.

Example 1:



Input:

n = 5, edges = [[1,2,3],[1,3,3],[2,3,1],[1,4,2],[5,2,2],[3,5,1],[5,4,10]]

Output:

3

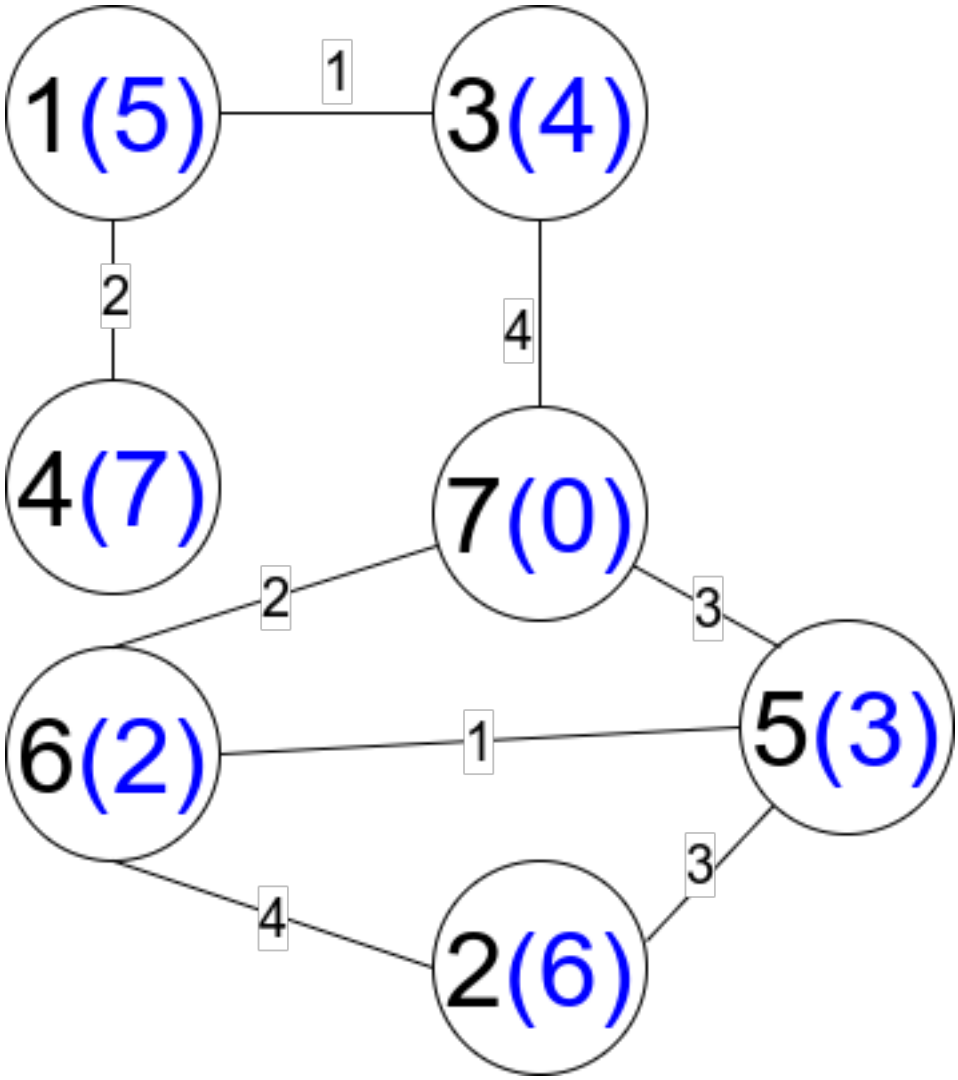
Explanation:

Each circle contains the node number in black and its

distanceToLastNode value in blue.

The three restricted paths are: 1) 1 --> 2 --> 5 2) 1 --> 2 --> 3 --> 5 3) 1 --> 3 --> 5

Example 2:



Input:

n = 7, edges = [[1,3,1],[4,1,2],[7,3,4],[2,5,3],[5,6,1],[6,7,2],[7,5,3],[2,6,4]]

Output:

1

Explanation:

Each circle contains the node number in black and its

distanceToLastNode value in blue.

The only restricted path is 1 --> 3 --> 7.

Constraints:

1 <= n <= 2 * 10

4

n - 1 <= edges.length <= 4 * 10

4

edges[i].length == 3

1 <= u

i

, v

i

<= n

u

i

!= v

i

1 <= weight

i

<= 10

5

There is at most one edge between any two nodes.

There is at least one path between any two nodes.

Code Snippets

C++:

```
class Solution {
public:
int countRestrictedPaths(int n, vector<vector<int>>& edges) {

}
};
```

Java:

```
class Solution {
public int countRestrictedPaths(int n, int[][] edges) {

}
}
```

Python3:

```
class Solution:
def countRestrictedPaths(self, n: int, edges: List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def countRestrictedPaths(self, n, edges):
"""
:type n: int
:type edges: List[List[int]]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number} n
* @param {number[][]} edges
* @return {number}
*/
var countRestrictedPaths = function(n, edges) {

};
```

TypeScript:

```
function countRestrictedPaths(n: number, edges: number[][]): number {

};
```

C#:

```
public class Solution {
public int CountRestrictedPaths(int n, int[][] edges) {

}
}
```

C:

```
int countRestrictedPaths(int n, int** edges, int edgesSize, int*
edgesColSize) {

}
```

Go:

```
func countRestrictedPaths(n int, edges [][]int) int {

}
```

Kotlin:

```
class Solution {
fun countRestrictedPaths(n: Int, edges: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func countRestrictedPaths(_ n: Int, _ edges: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn count_restricted_paths(n: i32, edges: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer[][]} edges
# @return {Integer}
def count_restricted_paths(n, edges)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $edges
* @return Integer
*/
function countRestrictedPaths($n, $edges) {

}
}
```

Dart:

```
class Solution {
int countRestrictedPaths(int n, List<List<int>> edges) {
```

```
}
}
```

Scala:

```
object Solution {
def countRestrictedPaths(n: Int, edges: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec count_restricted_paths(n :: integer, edges :: [[integer]]) :: integer
def count_restricted_paths(n, edges) do

end
end
```

Erlang:

```
-spec count_restricted_paths(N :: integer(), Edges :: [[integer()]]) ->
integer().
count_restricted_paths(N, Edges) ->
.
```

Racket:

```
(define/contract (count-restricted-paths n edges)
(-> exact-integer? (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Number of Restricted Paths From First to Last Node
```

```
* Difficulty: Medium
* Tags: array, graph, dp, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int countRestrictedPaths(int n, vector<vector<int>>& edges) {

}
};
```

Java Solution:

```
/**
* Problem: Number of Restricted Paths From First to Last Node
* Difficulty: Medium
* Tags: array, graph, dp, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int countRestrictedPaths(int n, int[][] edges) {

}
}
```

Python3 Solution:

```
"""
Problem: Number of Restricted Paths From First to Last Node
Difficulty: Medium
Tags: array, graph, dp, sort, queue, heap

Approach: Use two pointers or sliding window technique
```

```
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def countRestrictedPaths(self, n: int, edges: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def countRestrictedPaths(self, n, edges):
"""
:type n: int
:type edges: List[List[int]]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Number of Restricted Paths From First to Last Node
* Difficulty: Medium
* Tags: array, graph, dp, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number} n
* @param {number[][]} edges
* @return {number}
*/
var countRestrictedPaths = function(n, edges) {

};
```

TypeScript Solution:

```
/**
* Problem: Number of Restricted Paths From First to Last Node
* Difficulty: Medium
* Tags: array, graph, dp, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function countRestrictedPaths(n: number, edges: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Number of Restricted Paths From First to Last Node
* Difficulty: Medium
* Tags: array, graph, dp, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int CountRestrictedPaths(int n, int[][] edges) {

}
}
```

C Solution:

```
/*
* Problem: Number of Restricted Paths From First to Last Node
* Difficulty: Medium
* Tags: array, graph, dp, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
```

```
*/

int countRestrictedPaths(int n, int** edges, int edgesSize, int*
edgesColSize) {

}
```

Go Solution:

```
// Problem: Number of Restricted Paths From First to Last Node
// Difficulty: Medium
// Tags: array, graph, dp, sort, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func countRestrictedPaths(n int, edges [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun countRestrictedPaths(n: Int, edges: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func countRestrictedPaths(_ n: Int, _ edges: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Number of Restricted Paths From First to Last Node
// Difficulty: Medium
```

```
// Tags: array, graph, dp, sort, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn count_restricted_paths(n: i32, edges: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[][]} edges
# @return {Integer}
def count_restricted_paths(n, edges)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $edges
* @return Integer
*/
function countRestrictedPaths($n, $edges) {

}
}
```

Dart Solution:

```
class Solution {
int countRestrictedPaths(int n, List<List<int>> edges) {

}
```

```
}
```

Scala Solution:

```
object Solution {
def countRestrictedPaths(n: Int, edges: Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec count_restricted_paths(n :: integer, edges :: [[integer]]) :: integer
def count_restricted_paths(n, edges) do

end
end
```

Erlang Solution:

```
-spec count_restricted_paths(N :: integer(), Edges :: [[integer()]]) ->
integer().
count_restricted_paths(N, Edges) ->
.
```

Racket Solution:

```
(define/contract (count-restricted-paths n edges)
(-> exact-integer? (listof (listof exact-integer?)) exact-integer?)
)
```