Problem 882: Reachable Nodes In Subdivided Graph

Problem Information

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

Problem Description

You are given an undirected graph (the

"original graph"

) with

n

nodes labeled from

0

to

n - 1

. You decide to

subdivide

each edge in the graph into a chain of nodes, with the number of new nodes varying between each edge.

The graph is given as a 2D array of

edges

where

edges[i] = [u

i

, v

i

, cnt

i

]

indicates that there is an edge between nodes

u

i

and

v

i

in the original graph, and

cnt

i

is the total number of new nodes that you will

subdivide

the edge into. Note that

cnt

i

== 0

means you will not subdivide the edge.

To

subdivide

the edge

[u

i

, v

i

]

, replace it with

(cnt

i

+ 1)

new edges and

cnt

i

new nodes. The new nodes are

x

1

,

x

2

, ...,

x

cnt

i

, and the new edges are

[u

i

, x

1

]

,

[x

1

, x

2

]

,

[x

2

, x

3

]

, ...,

[x

cnt

i

-1

, x

cnt

i

]

,

[x

cnt

i

, v

i

]

.

In this

new graph

, you want to know how many nodes are

reachable

from the node

0

, where a node is

reachable

if the distance is

maxMoves

or less.

Given the original graph and

maxMoves

, return

the number of nodes that are

reachable

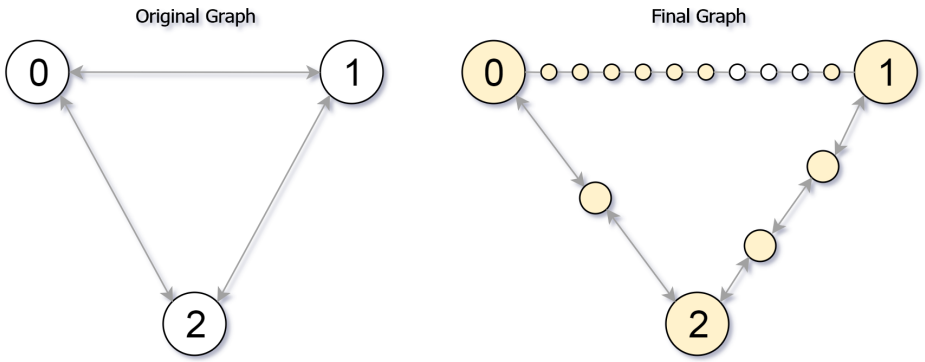
from node

0

in the new graph

.

Example 1:



Input:

edges = [[0,1,10],[0,2,1],[1,2,2]], maxMoves = 6, n = 3

Output:

13

Explanation:

The edge subdivisions are shown in the image above. The nodes that are reachable are highlighted in yellow.

Example 2:

Input:

edges = [[0,1,4],[1,2,6],[0,2,8],[1,3,1]], maxMoves = 10, n = 4

Output:

23

Example 3:

Input:

edges = [[1,2,4],[1,4,5],[1,3,1],[2,3,4],[3,4,5]], maxMoves = 17, n = 5

Output:

1

Explanation:

Node 0 is disconnected from the rest of the graph, so only node 0 is reachable.

Constraints:

0 <= edges.length <= min(n * (n - 1) / 2, 10

4

)

edges[i].length == 3

0 <= u

i

< v

i

< n

There are

no multiple edges

in the graph.

0 <= cnt

i

<= 10

4

0 <= maxMoves <= 10

9

1 <= n <= 3000

Code Snippets

C++:

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

}
};
```

Java:

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

```
}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

```
int reachableNodes(int** edges, int edgesSize, int* edgesColSize, int
maxMoves, int n) {

}
```

Go:

```
func reachableNodes(edges [][]int, maxMoves int, n int) int {

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (reachable-nodes edges maxMoves n)
(-> (listof (listof exact-integer?)) exact-integer? exact-integer?
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Reachable Nodes In Subdivided Graph
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
```

```
};
```

Java Solution:

```
/**
* Problem: Reachable Nodes In Subdivided Graph
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
}
```

Python3 Solution:

```
"""
Problem: Reachable Nodes In Subdivided Graph
Difficulty: Hard
Tags: array, graph, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Reachable Nodes In Subdivided Graph
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

};
```

TypeScript Solution:

```
/**
* Problem: Reachable Nodes In Subdivided Graph
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

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

};
```

C# Solution:

```
/*
* Problem: Reachable Nodes In Subdivided Graph
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
}
```

C Solution:

```
/*
* Problem: Reachable Nodes In Subdivided Graph
* Difficulty: Hard
* Tags: array, graph, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

int reachableNodes(int** edges, int edgesSize, int* edgesColSize, int
maxMoves, int n) {

}
```

Go Solution:

```
// Problem: Reachable Nodes In Subdivided Graph
// Difficulty: Hard
// Tags: array, graph, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func reachableNodes(edges [][]int, maxMoves int, n int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Reachable Nodes In Subdivided Graph
// Difficulty: Hard
// Tags: array, graph, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

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

```
}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

```
}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (reachable-nodes edges maxMoves n)
(-> (listof (listof exact-integer?)) exact-integer? exact-integer?
exact-integer?)
)
```