Text Compare - Leetcode 269 two practice

Mode: All
Left file: Leetcode269_AlienDictionary_P4.cs
Right file: Leetcode269_AlienDictionary_warmup.cs

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
	=	using System;
		using System.Collections.Generic;
		using System.Linq;
		using System.Text;
		using System.Threading.Tasks;

namespace _269AlienDictionary	<>	namespace _269AlienDictionary_Review

{
    class Program
    {
        static void Main(string[] args)
        {
            string[] words = { "wrt", "wrf", "er", "ett", "rftt" };
	=	{
		    class Program
		    {
		        static void Main(string[] args)
		        {
		            string[] words = { "wrt", "wrf", "er", "ett", "rftt" };

	-+	// verify result manually here: "wertf"

string result = alienOrder(words);
}
	=	string result = alienOrder(words);
		}

/*
 * source code from blog:
 * http://www.cnblogs.com/yrbbest/p/5023584.html
 *
 * Topological sorting - Kahn's Algorithm
 */
public static String alienOrder(String[] words) {
 	
if(words == null || words.Length == 0) {
	<>	
		
		public static string alienOrder(string[] words)
		{
		if (words == null || words.Length == 0)

return "";	=	**return** "";

}
 	
Dictionary<char, HashSet<char>> graph = new Dictionary<char, HashSet<char>>();
	<>	
		// Graph presentation:
		// nodes, node's dependency list and inDegree array
		// nodes - function getNodes()
		//
		Dictionary<char, HashSet<char>> dependencyList = new Dictionary<char, HashSet<char>>();

	=	

int AlphabeticSize = 26;
int[] inDegree = new int[AlphabeticSize];
	<>	
		int[] inDegree = new int[26];

	=	
graphSetup(words, graph, inDegree);	<>	graphSetup(words, dependencyList, inDegree);
	=	

return topologicalSort(words, graph, inDegree);
}
	<>	return topoligicalSort(words, dependencyList, inDegree);
		}

	=	

/*
 * First time to use HashSet UnionWith api - good practice!
 *
 */
public static HashSet<char> getCharSet(string[] words)
{
HashSet<char> set = new HashSet<char>();
	<>	
		
		public static HashSet<char> getNodes(string[] words)
		{
		HashSet<char> hashset = new HashSet<char>();

	=	

foreach (string word in words) {
 	
set.UnionWith(word.ToList());
	<>	foreach (string s in words)
		{
		hashset.UnionWith(s.ToList());

} 	=	}
return set;	<>	return hashset;
} 	=	}
	-+	
/*	=	/*
* Topological Sort algorithm in Graph	<>	* https://en.wikipedia.org/wiki/Topological_sorting

* Need to review		
*	=	*
* When the node's inDegree's value goes down to zero, the node * is ready to enqueue.	<>	* review the idea of topological sorting: * 1. push all indegree nodes with 0 into the queue * 2. work on queue, * step 1: dequeue the node from the queue, * step 2: update indegree node's value affected - decrement one * step 3: if the dependency list's node indegree value down to zero, * add the node to the queue * 3. construct the output string * * It is just the normal queue process, can you do a bug free writing?
*/	=	*/
public static string topologicalSort(string[] words, Dictionary<char, HashSet<char>> graph,	<>	public static string topoligicalSort(string[] words, Dictionary<char, HashSet<char>> dependencyList,
int[] inDegree) {	=	**int**[] inDegree) {
HashSet<char> set = getCharSet(words);	<>	HashSet<char> nodes = getNodes(words);
	=	

int AlphabeticSize = 26;
 	
// Topological sort - starting from nodes with indegree value 0
// put all those nodes into queue first.
// Go through all 26 chars, and then, add chars with indegree 0 - make
// sure that chars are in the HashSet set
Queue<char> queue = new Queue<char>();
 	
for (int i = 0; i < AlphabeticSize; i++)
	<>	Queue<char> queue = new Queue<char>();
		
		for (int i = 0; i < 26; i++)

{	=	{

char curr = (char)('a' + i);
if (inDegree[i] == 0 && set.Contains(curr))
 	
{
 	
queue.Enqueue(curr);
}
	<>	char runner = (char) ('a' + i);
		if (!nodes.Contains(runner))
		
		continue; // skip it!
		
		if (inDegree[i] == 0)
		queue.Enqueue(runner);

}	=	}
StringBuilder sb = new StringBuilder(); /* * keep updating indgree value based on the queue's operation * once the node's indegree value's 0, push node to the queue. * That is how it works - continue to output chars */	<>	StringBuilder sb = new StringBuilder();

while (queue.Count > 0)
{
	=	**while** (queue.Count > 0)
		{

char node = queue.Peek();
sb.Append(node);
queue.Dequeue(); // bug001 - Do not forget to dequeue
 	
if (!graph.ContainsKey(node))
 	
break; // something is wrong - "all nodes in the queue are from graph"
 	
// check nodes in the dependency list
 	
foreach (char runner in graph[node])
	<>	char runner = queue.Dequeue();
		
		sb.Append(runner);
		if (!dependencyList.ContainsKey(runner))
		continue;
		
		HashSet<char> neighbors = dependencyList[runner];
		foreach (char c in neighbors)

{	=	{
int index = runner - 'a';	<>	int index = c - 'a';
inDegree[index]--; **if** (inDegree[index] == 0)	=	inDegree[index]--; **if** (inDegree[index] == 0)

{
queue.Enqueue(runner);
}
}
	<>	
		queue.Enqueue(c);
		}

}	=	}

// edge case discussion:
// if the graph has cycle, then, ?
// What will be case with "" <- give an example:
return sb.Length != set.Count ? "" : sb.ToString();
	<>	// edge case:
		return sb.Length < nodes.Count? "" : sb.ToString();

}	=	}
/* * June 16, 2016 * Work on the detail - How graph is saved using dependency list * * Construct the graph * Nodes in the graph at most 26 chars, a, b, c,d, ..., z * * int[] inDegree - 26 * Dictionary<char, HashSet<char>> graph * For example, * "wrt", "wrf", "er", "ett", "rftt" * * inDegree: * 'w' - indegree['w'-'a'] = 0 * "wrt", "wrf" -> we can tell that t->f, so this edge t->f, * how to save it in the graph? * * We choose to save the dependency list -> t has a list of dependency, f is one in the list * graph['t'] is a hashset, and then, make sure that 'f' is added to the hashset	<>	/* * Motivation talk: * set up graph for {"wrt","wrf"} * output: * 'f' - add 'f' into dependency list's dictionary, also, update content {'t'} * inDegree array setup for inDegree['f'-'a']
*	=	*
* Next, the smart tip about comparison: * wrt -> wrf, skip first two chars, and then set up third char t->f edge in the graph * You need to figure out what is edge in the graph through this words order. * Extract the information -	<>	* two words, at most one edge
*	=	*
* This function is not easy to maintain bug free * Need to enforce some rules in the code: * Rule 1: ? * Rule 2: ? * * filter out duplicated relationship ["za","zb","ca","cb"], then, a->b will show up twice * read Java code for more discussion on this: * http://blog.csdn.net/feliciafay/article/details/50040985	<>	* two words, no edge - special case discussion: * test case: * case 1: "a", "ab" * case 2:
*	=	*
* Review graph implementation: * http://www.geeksforgeeks.org/graph-and-its-representations/	+-	
*	=	*
* Directed Graph: Edge - From (u) -> To (v) * * Graph setup: * 1. Add node in the graph * 2. update node's dependency list - a HashSet *	<>	* That is it!
*/ **public static void** graphSetup(	=	*/ **public static void** graphSetup(

string[]   words,
Dictionary<char, HashSet<char>> graph,
int[]   inDegree)
{
for (int i = 1; i < words.Length; i++) {
String previous = words[i - 1];
String current = words[i];
	<>	string[] words,
		Dictionary<char, HashSet<char>> dependencyList,
		int[] inDegree
		)
		{
		int len = words.Length;
		if (len == 1)
		return; // cannot do anything
		for (int i = 1; i < len; i++)
		{
		string prev = words[i - 1];
		string curr = words[i];

	=	

int shortLength = Math.Min(previous.Length, current.Length);
for (int j = 0; j < shortLength; j++)
	<>	int start = 0;
		while (prev[start] == curr[start])

{	=	{

Text Compare

char edgeFrom = previous[j];
char edgeTo   = current[j];
if (edgeFrom == edgeTo)
continue;
// start node - need to add a node in the graph
if (!graph.ContainsKey(edgeFrom)) {
graph.Add(edgeFrom, new HashSet<char>());
}
	<>	start++;
		}
		// no edge
		if(start >= Math.Min(prev.Length, curr.Length))
		return;
		// at most one edge
		char edgeFrom = prev[start];
		//char edgeTo = prev[start]; // bug001
		char edgeTo = curr[start];
		if (!dependencyList.ContainsKey(edgeFrom))
		{
		dependencyList.Add(edgeFrom, new HashSet<char>());
		}

	=	

// Avoid bugs
// Do not add same node twice in inDegree array
// For example, wrt->wrf => t->f, 'f''s indgree from 't', should not count
// twice.
// filter out duplicated relationship
// ["za","zb","ca","cb"], then, a->b will show up twice
//
// Try to describe what code is doing here:
// if adjacency list does not contains edgeTo, then, it is the
// first time visited, then, increment one to inDegree array for
// the char
if (!graph[edgeFrom].Contains(edgeTo)) {
inDegree[edgeTo - 'a']++;
}
// For any case, add edgeTo to the HashSet - first time, add,
// second time, will be ignored.
// update dependency list.
graph[edgeFrom].Add(edgeTo);
break;
}
}
}
	<>	if (!dependencyList[edgeFrom].Contains(edgeTo))
		{
		dependencyList[edgeFrom].Add(edgeTo);
		inDegree[edgeTo - 'a']++;
		}
		}
		}

} }	=	} }