Aufgabe 2: Schwierigkeiten

Team-ID: 00852

Team-Name: Team_Magdeburg_an_die_Macht

Bearbeiter/-innen dieser Aufgabe:
David Noskov

18. November 2024

Inhaltsverzeichnis

1 Lösungsidee **3**
1.1 Hauptaufgabe . . . 3
1.2 Konflikte . . . 3

2 Umsetzung **3**

3 Pseudocode **4**

4 Beispiele **4**

5 Quellcode **5**
5.1 Algorithm.cs . . . 5
5.2 Graph.cs . . . 6

1 Lösungsidee

1.1 Hauptaufgabe

Um die Aufgaben der Klausuren nach der Schwierigkeit zu ordnen, basiert meine Lösung darauf, das Problem in einen gerichteten Graphen zu modellieren, in Form einer Adjazenzliste. Jede Aufgabe ist ein Knoten und jede gerichtete Kante zeigt auf eine leichtere Aufgabe.
Anschließend wird die Liste topologisch sortiert, um eine Reihenfolge zu erhalten, die den gegebenen Schwierigkeitsgraden der Aufgaben entspricht. Eine topologische Sortierung liefert eine gültige lineare Ordnung der Knoten in einem gerichteten azyklischen Graphen (DAG), die den Schwierigkeitsrelationen entspricht.

1.2 Konflikte

Bei einem Konflikt herrscht ein Zyklus in dem Graphen.
$A \rightarrow B$
$B \rightarrow A$
Um diese Konflikte aufzulösen, wird mithilfe eines Deep-First-Search ein Zyklus gesucht. Wird ein Zyklus gefunden, wird ein Knoten entfernt, um den Zyklus zu entfernen und sicherzustellen, dass der Graph azyklisch ist.

2 Umsetzung

Die Umsetzung und Implementierung des Algorithmus orientiert sich eng an der entwickelten Lösungsstrategie. Der Algorithmus verwendet eine Kombination aus Graphenrepräsentation, topologischer Sortierung und Konfliktbehandlung, um eine gültige Anordnung der Aufgaben gemäß den gegebenen Schwierigkeitsrelationen zu erstellen.

Für die Modellierung des Problems wurde der Graph als Adjazenzliste realisiert. Hierzu wird ein Dictionary verwendet, das jeden Knoten (Aufgabe) mit einer leichteren Aufgaben verbindet.

Bevor der Graph topologisch sortiert werden kann, muss der Graph azyklisch sein, also kein Zyklus haben. Dazu wird mithilfe einer Tiefensuche nach einem Zyklus gesucht. Wird ein Zyklus gefunden, wird eine zufällige Kante entfernt unter Berücksichtigung, dass der Graph zusammenhängend bleibt bzw, nicht in zwei geteilt wird.

Für die endgültige Anordnung der Aufgaben wird eine topologische Sortierung durchgeführt. Dies wird mithilfe eines weiteren Deep-First-Search ausgeführt.

3 Pseudocode

```
Funktion find_task_order(lines):
    // Einlesen der Eingabedaten
    info = parse(lines[0])
    klausuren = lines[1:info[0]]
    aufgaben = lines[lines.Length - 1]

    // Erstelle den Graphen
    graph = Graph()
    für i in 0 bis info[1] - 1:
        graph.add_node(alphabet[i])
    für klausur in klausuren:
        für j in 0 bis klausur.length - 2:
            graph.add_edge(klausur[j], klausur[j+1])

    // Konfliktbehandlung
    graph.remove_cycle()

    // Topologische Sortierung
    liste = graph.topological_sort()

    // Filtere die relevanten Aufgaben und gebe sie aus
    ergebnis = ""
    für aufgabe in liste:
        wenn aufgabe in aufgaben:
            ergebnis += aufgabe
    Rückgabe ergebnis
```

4 Beispiele

Das Programm wurde auf einem Windows System mit einem AMD Ryzen 7 5800HS Prozessor ausgeführt.

schwierigkeiten0.txt
Laufzeit: 00:00:00.0066154
BEDFC

schwierigkeiten1.txt
Laufzeit: 00:00:00.0000727
AGCDF

schwierigkeiten2.txt
Laufzeit: 00:00:00.0000197
ABDEFG

schwierigkeiten3.txt
Laufzeit: 00:00:00.0000180
MNHILJKAEFGBDC

schwierigkeiten4.txt
Laufzeit: 00:00:00.0000375
BIFWN

schwierigkeiten5.txt
Laufzeit: 00:00:00.0000323
ZRQKHCSENOMJLFVAGPXBWIYDUT

5 Quellcode

Der Code wurde in C geschrieben. Die Logik des Programms, ist in zwei Dateien unterteilt; Algorithm.cs und Graph.cs

5.1 Algorithm.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;


namespace Schwierigkeiten.src
{
    internal class Algorithm
    {
        /*
         * 0 = Anzahl der Klausuren
         * 1 = Gesamtanzahl an Aufgaben
         * 2 = Anzahl an Aufgaben für welche eine gute Anordnung gefunden werden soll
         */
        private readonly int[] info;
        // Liste aller Klausuren
        private readonly List<string> klausuren;
        // Aufgaben, für die eine gute Anordnung gefunden werden soll
        private readonly string aufgaben;

        Stopwatch stopwatch;

        public Algorithm(string[] lines)
        {
            stopwatch = new();
            stopwatch.Start();
            string infoLine = lines[0];
            info = infoLine
                .Split(new[] { ' ' })
                .Select(int.Parse)
                .ToArray();
            klausuren = [];
            for (int i = 1; i <= info[0]; i++)
            {
                klausuren.Add(removeClutter(lines[i]));
            }
            aufgaben = removeClutter(lines[lines.Length - 1]);
        }

        private string removeClutter(string line)
        {
            string newLine = line;
            newLine = newLine.Replace(" ", "");
            newLine = newLine.Replace("<", "");
            return newLine;
        }

        public void Solve()
        {
```

```
            Graph graph = CreateGraph();
            graph.RemoveCycle();
            List<char> liste = graph.TopologicalSort();
            //liste.ForEach(x => Console.Write($"{x} <"));
            string ergebnis = "";
            for (int i = 0; i < liste.Count; i++)
            {
                for (int j = 0; j < info[2]; j++)
                {
                    if (aufgaben[j] == liste[i])
                    {
                        ergebnis += liste[i];
                        break;
                    }
                }
            }
            Console.WriteLine($"Laufzeit: {stopwatch.Elapsed}");
            Console.WriteLine(ergebnis + "\n");
            stopwatch.Stop();
        }

        private Graph CreateGraph()
        {
            string alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
            Graph graph = new();
            for (int i = 0; i < info[1]; i++)
            {
                graph.AddNode(alphabet[i]);
                //Console.WriteLine(alphabet[i]+ " :node created");
            }
            foreach (string klausur in klausuren)
            {
                for (int i = 0; i < klausur.Length - 1; i++)
                {
                    graph.AddEdge(klausur[i], klausur[i+1]);
                }
            }
            return graph;
        }
    }
}
```

5.2 Graph.cs

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

namespace Schwierigkeiten.src
{
    internal class Graph
    {
        private Dictionary<char, List<char>> adjList;

        public Graph()
```

```
{
    adjList = [];
}

public void AddNode(char node)
{
    if (!adjList.ContainsKey(node))
    {
        adjList[node] = [];
    }
}

public void AddEdge(char from, char to)
{
    if (!adjList[from].Contains(to))
        adjList[from].Add(to);
}

public void RemoveEdge(char from, char to)
{
    adjList[from].Remove(to);
}

public List<char> TopologicalSort()
{
    HashSet<char> visited = [];
    Stack<char> resultStack = [];

    foreach (var node in adjList.Keys)
    {
        if (!visited.Contains(node))
        {
            DFS(node, visited, resultStack);
        }
    }
    return resultStack.ToList(); // Rückgabe der topologisch sortierten Liste
}

public void RemoveCycle()
{
    // Suche nach einem Zyklus und entferne eine Kante, die keine Brücke ist
    foreach (var node in adjList.Keys)
    {
        var visited = new HashSet<char>();
        var stack = new Stack<char>();
        var resultStack = new Stack<char>();

        if (FindCycle(node, '\0', visited, stack))
        {
            var cycleNodes = stack.ToList();

            for (int i = 0; i < cycleNodes.Count - 1; i++)
            {
                char u = cycleNodes[i];
                char v = cycleNodes[i + 1];

                // Überprüfen, ob (u, v) eine Brücke ist
                if (!IsBridge(u, v, [], resultStack))
```

```
                {
                    adjList[u].Remove(v);
                    adjList[v].Remove(u);
                    return;
                }
            }
        }
    }
}

// Hilfsfunktion zur Zyklenerkennung mit DFS
private bool FindCycle(char current, char parent, HashSet<char> visited, Stack<char> stack)
{
    visited.Add(current);
    stack.Push(current);

    foreach (var neighbor in adjList[current])
    {
        if (neighbor == parent)
            continue;

        if (visited.Contains(neighbor))
        {
            stack.Push(neighbor);
            return true;
        }

        if (FindCycle(neighbor, current, visited, stack))
        {
            return true;
        }
    }

    stack.Pop();
    return false;
}

// Funktion zur Überprüfung, ob eine Kante eine Brücke ist
private bool IsBridge(char u, char v, HashSet<char> visited, Stack<char> resultStack)
{
    // Entferne die Kante temporär
    adjList[u].Remove(v);
    adjList[v].Remove(u);

    // Überprüfen, ob u und v noch im selben Teil des Graphen sind
    DFS(u, visited, resultStack);

    // Füge die Kante wieder hinzu
    adjList[u].Add(v);
    adjList[v].Add(u);

    // Überprüfen, ob beide Knoten noch in derselben Komponente sind
    return !visited.Contains(v);
}

// DFS-Funktion zur Überprüfung der Erreichbarkeit oder für topologischen Sort
private void DFS(char node, HashSet<char> visited, Stack<char> resultStack)
{
```

```
            visited.Add(node);

            foreach (var neighbor in adjList[node])
            {
                if (!visited.Contains(neighbor))
                {
                    DFS(neighbor, visited, resultStack);
                }
            }

            // Fügt den Knoten dem resultStack hinzu, um später den topologischen Sort zu haben
            resultStack.Push(node);
        }

    }
}
```