Aufgabe 1: Zimmerbelegung

Team: “throw new Exception();“

Einsendenummer: 00025

26. November 2017

**Inhaltsverzeichnis**

[Lösungsidee 1](#_Toc492675188)

[Umsetzung 1](#_Toc492675189)

[Beispiele 2](#_Toc492675190)

[Quellcode 4](#_Toc492675191)

# Lösungsidee

Die Lösungsidee für dieses Problem ist sehr simpel.

Zunächst wird eine Schülerin ausgewählt die das erste Mitglied eines neuen Raumes sein soll.

Anschließend sollen alle Schülerinnen gefunden werden, die auf eine von zwei Arten in diesem Raum benötigt werden:

1. Ein Mitglied des Raumes mag diese Schülerin. (Beziehung 1)
2. Die Schülerin mag ein Mitlgied des Raumes. (Beziehung 2)

Wenn alle relvanten Schülerinnen gefunden und hinzugefügt wurden, muss überprüft werden ob es Widersprüche gibt. Dabei werden aus einer Liste, die alle Schülerinnen enthält (welche von mindestens einem der Mitglieder nicht gemocht werden) Namen gesucht, die auf der Mitgliederliste stehen. Sollte es Übereinstimmungen geben, ist der Raum automatisch ungültig.

Auf diese Art und Weise können alle gültigen Räume gefüllt werden und Widersprüche nicht entstehen.

# Umsetzung

Umgesetzt wurde die o.g. Lösungsidee wiefolgt in Visual C#:

Für die Implementation des o.g. Algorithmus‘ werden zwei Klassen benötigt: Schülerin und Raum.

Eine Schülerin hat einen Namen, eine Mag-Liste und eine Hass-Liste. Diese beiden Listen enthalten Namen anderer Schülerinnen, die die Schülerin mag bzw. hasst.

Ein Raum hat eine Mitglieder-Liste, eine Hass-Liste, die alle einzigartigen Namen der Hass-Listen aller Mitglieder enthält und eine Funktion, die diese beiden Listen vergleicht. Bei Übereinstimmung wird „true“ zurückgegeben.

Nachdem alle Namen und Beziehungen aus einer Input-Datei gelsen wurden, wird solange versucht gültige Räume zu erstellen, bis keine Schülerinnen mehr auf der Input-Liste stehen.

Dabei wird ein gefundener Verbund von Name nur dann auf die finale Liste gesetzt, wenn kein Widerspruch besteht.

Gefunden wird solch ein Verbund wie folgt:

Eine Startschülerin wird ausgewählt, in diesem Fall die erste von der Input-Liste. Diese Schülerin wird nun einem neuen Raum hinzugefügt und aus der Input-Liste entfernt. Des Weiteren werden alle gehassten Schülerinnen auf den Raum übertragen. Anschließend wird eine „toAdd“-Liste erstellt, welche schlicht mit Namen von Schülerinnen gefüllt werden soll, die einerseits dem Raum hinzugefügt werden müssen und andererseits noch gemochte und gehasste Schülerinnen haben könnten, die es ebenfalls zu berücksichtigen gilt. Dieser Liste werden sofort alle gemochten Schülerinnen der Startschülerin hinzugefügt.

Folgendes wird solange wiederholt, bis keine Namen mehr zu bearbeiten sind, d.h. keine mehr auf der „toAdd“-Liste stehen.

Zunächst wird die Schülerin passend zum ersten Namen der „toAdd“-Liste gesucht. Wenn gefunden, wird sie dem aktuellen Raum hinzugefügt. Anschließend wird sie aus der „toAdd“-Liste und der Input-Liste entfernt. Außerdem wird der Raum inklusive seiner Hass-Liste und die „toAdd“-Liste mit der Schülerin so fusioniert, dass alle gemochten Schülerinnen, die weder Mitglied im Raum sind, noch auf der „toAdd“-Liste stehen auf die „toAdd“-Liste geschrieben werden und alle gehassten Schülerinnen, die noch nicht von anderen Mitlgieder des Raumes gehasst werden, einen Eintrag auf der Hass-Liste erhalten.

Im Folgenden wird die „toAdd“-Liste nocheinmal geupdated. Dazu wird die Input-Liste nach all jenen Schülerinnen durchsucht, die nach aktuellem Stand eines der Raummitglieder mögen.

Sollte es nun noch Elemente auf der „toAdd“-Liste geben, wird das Ganze wiederholt, sonst ist der Raum vollständig und kann nach Widersprüchen durchsucht werden. Die Methode dazu Vergleicht jedes Element der Mitlgieder-Liste mit jedem Element der Hass-Liste.

Wenn es mindestens einen Widerspruch gibt, wird der Ansatz verworfen und mit dem nächsten Raum fortgefahren, gibt es keine, so wird vor dem Fortfahren der Raum auf der finalen Liste gespeichert.

Wenn alle Berechnungen abgeschlossen sind, werden die Ergebnisse sowohl in der Konsole, als auch in einer Datei ausgegeben.

# Beispiele

Aus praktischen Gründen wurde das Programm so geschrieben, dass im Verzeichnis der eingelesenen Datei eine weitere Datei mit dem Namen [Originalname] + „-output“+“.txt“ erstellt wird. Diese dient offensichtlich als Output und enthält folgende Informationen: Zeilenweise werden die Namen der Mitglieder aller widerspruchsfreien Räume aufgelistet und am Ende zusammengefasst, wie viele Schülerinnen der Eingabe in einen Raum sortiert werden konnten. Sie finden sowohl Eingabe, als auch Ausgabedateien im Anhang.

Bei den vorgegebenen Beispielen wurden folgende Ergebnisse erzielt:

* zimmerbelegung1.txt:
  + keine Räume
  + 0 / 7 Schülerinnen wurden eingeteilt
* zimmerbelegung2.txt:
  + drei Räume mit 2, 3 und 1 Schülerin(nen)
  + 6 / 6 Schülerinnen wurden eingeteilt
* zimmerbelegung3.txt:
  + fünf Räume mit 14, 1, 5, 13 und 10 Schülerinnen
  + 43 / 43 Schülerinnen wurden eingeteilt
* zimmerbelegung4.txt:
  + vier Räume mit 6, 19, 17 und 1 Schülerin(nen)
  + 43 / 43 Schülerinnen wurden eingeteilt
* zimmerbelegung5.txt:
  + 34 Räume mit 31 x 1, 2, 3 und 6 Schülerin(nen)
  + 42 / 43 Schülerinnen wurden eingeteilt
* zimmerbelegung6.txt:
  + 5 Räume mit 25, 11, 2, 2, 3 Schülerinnen
  + 43 / 43 Schülerinnen wurden eingeteilt

Zusätzlich befindet sich im Anhang eine Testdatei, die als „zimmerbelegung0.txt“ benannt wurde. Mit dieser wurde das Programm getestet.

Auswertend gilt es zu sagen, dass die Wünsche in einigen Fällen zwar theoretisch erfüllbar sind, aber praktisch schlicht keine passenden Räume existieren. Wenn im 6. Fall ein Raum für 25 Schülerinnen benötigt wird, ist recht sicher, dass die Zuständigen keine passende Unterkunft finden werden.

Damit unterscheidet sich das gegebene Problem vom Hochzeitsproblem, wo keine ideale, sondern eine nachhaltige Lösung gesucht wird. Dabei wird ignoriert, ob die einzelnen jemand anderen bevorzugen würden, solange dieser Zustand nicht auf Gegenseitigkeit beruht. Um das Problem realistisch zu Lösen, sollten also Raumgrößen genannt werden und statt der „schwarz –weiß–Einteilung“ ein Beliebtheitsranking aller anderen Schülerinnen aus Sicht einer einzelnen bekannt sein. Somit könnte ein Algorithmus versuchen die Schülerinnen so anzuordnen, dass eben keine Schülerinnen miteinander einen Raumtausch bevorzugen würden.

# Quellcode

Da die Implementation bereits zuvor erläutert wurde, werden die folgenden Codeabschnitte weitesgehend unkommentiert sein. Das vollständige Projekt finden sie im Abgabeordner.

public class Room

{

public Room()

{

this.Members = new List<string>();

this.PeopleMembersHate = new List<string>();

}

public List<string> Members { get; set; }

public List<string> PeopleMembersHate { get; set; }

public bool MembersAreHated ()

{

foreach (string member in Members)

{

foreach (string hated in PeopleMembersHate)

{

if (member == hated)

{

return true;

}

}

}

return false;

}

}

public class Student

{

public Student(string name)

{

this.name = name;

this.LikedPeople = new List<string>();

this.HatedPeople = new List<string>();

}

public string name { get; set; }

public List<string> LikedPeople { get; set; }

public List<string> HatedPeople { get; set; }

}

static void Main(string[] args)

{

string tempPath = Path();

rawStudents = Relations(tempPath);

rawCount = rawStudents.Count();

// calculation

List<Room> finalRooms = new List<Room>();

while (rawStudents.Count() > 0)

{

Room tempRoom = NextRoom();

if (tempRoom.MembersAreHated() == false)

{

finalRooms.Add(tempRoom);

}

else

{

Console.WriteLine("");

Console.WriteLine("The given instructions are contradictory!");

}

}

// output

Console.WriteLine("");

ReturnRooms(finalRooms, tempPath);

Console.ReadKey();

}

private static Room NextRoom()

{

Room tempRoom = new Room();

Student startStudent = rawStudents[0];

tempRoom.Members.Add(startStudent.name);

rawStudents.RemoveAt(0);

tempRoom.PeopleMembersHate.AddRange(startStudent.HatedPeople);

List<string> toAdd = new List<string>();

toAdd.AddRange(startStudent.LikedPeople);

while (toAdd.Count() > 0)

{

(Room room, List<string> toAdd) result   
 = AddRelevantStudents(tempRoom, toAdd);

tempRoom = result.room;

toAdd = result.toAdd;

}

return tempRoom;

}

private static (Room room, List<string> toAdd) AddRelevantStudents

(Room room, List<string> toAdd)

{

(Student student, bool exists) listEntry = FromList(toAdd[0]);

if (listEntry.exists)

{

room.Members.Add(toAdd[0]);

}

toAdd.RemoveAt(0);

rawStudents.Remove(listEntry.student);

if (listEntry.exists)

{

(Room room, List<string> toAdd) merged

= MergeIntoRoom(room, listEntry.student, toAdd);

room = merged.room;

toAdd = merged.toAdd;

}

toAdd = UpdateMemberLikedBy(room, toAdd);

return (room, toAdd);

}

private static (Student student, bool exists) FromList(string name)

{

Student currentStudent = new Student(name);

bool exists = false;

foreach (Student potentialRawStudent in rawStudents)

{

if (potentialRawStudent.name == name)

{

currentStudent = potentialRawStudent;

exists = true;

break;

}

}

return (currentStudent, exists);

}

private static (Room room, List<string> toAdd) MergeIntoRoom

(Room room, Student currentStudent, List<string> toAdd)

{

return (UpdateHated(room, currentStudent),

UpdateLiked(room, currentStudent, toAdd));

}

private static List<string> UpdateLiked

(Room room, Student currentStudent, List<string> toAdd)

{

foreach (string liked in currentStudent.LikedPeople)

{

bool newEntry = true;

foreach (string alreadyThere in room.Members)

{

if (liked == alreadyThere)

{

newEntry = false;

break;

}

}

if (newEntry)

{

foreach (string alreadyLiked in toAdd)

{

if (liked == alreadyLiked)

{

newEntry = false;

break;

}

}

if (newEntry)

{

toAdd.Add(liked);

}

}

}

return toAdd;

}

private static Room UpdateHated(Room room, Student currentStudent)

{

foreach (string hated in currentStudent.HatedPeople)

{

bool newEntry = true;

foreach (string alreadyHated in room.PeopleMembersHate)

{

if (hated == alreadyHated)

{

newEntry = false;

break;

}

}

if (newEntry)

{

room.PeopleMembersHate.Add(hated);

}

}

return room;

}

private static List<string> UpdateMemberLikedBy

(Room room, List<string> toAdd)

{

foreach (Student potentialMember in rawStudents)

{

bool needsToBeAdded = false;

foreach (string liked in potentialMember.LikedPeople)

{

foreach (string member in room.Members)

{

if (liked == member)

{

needsToBeAdded = true;

break;

}

}

if (needsToBeAdded)

{

break;

}

}

if (needsToBeAdded)

{

toAdd.Add(potentialMember.name);

}

}

return toAdd;

}

Es gibt weitere Methoden, diese sind aber für das Erfassen eines Pfades, für das Einlesen einer Datei und für das Zurückgeben des Ergebnisse relevant und betreffen nicht die Lösung des Problems. Bei Interesse sind sowohl diese Codeabschnitte, als auch alle anderen in kommentierter Fassung in der VS Solution.