Catalan Programming Assignment: Written Report

Philip Adams

13. Januar 2025

1 Introduction

Catalan ist ein Spiel, wo es darum geht einen gegeben Graphen, durch Löschen von Knoten, zu lösen. Man bekommt zu beginn einen Graphen mit nummerierten Knoten ($|\{Knoten\}| \equiv 1 \, mod \, 3$) mit definierten ungerichteten Kanten als Verbindungen zwischen den Knoten. Als Spiele kann man nur Knoten zusammenlegen wenn er genau drei Nachbarn hat. Wenn man vier Knoten zusammenlegt, werden alle Nachbarknoten, des Zentralen, gelöscht und ihre Verbindungen werden auf den Zentralen Knoten übertragen. Dadurch kann man das Ziel, nur noch einen Knoten zu haben, je nach Graph, auf verschiedenen Wegen erreichen, oder in Situationen kommen wo der Graph unlösbar wird.

2 Overview of Classes

2.1 Vertex

In der Vertex-Klasse habe ich bei der Implementierung als Klassenparameter nur die ID des Knoten als privat und final Parameter implementiert, da im Verlauf des Spieles kein Knoten verändert werden soll, sondern nur Knoten gelöscht werden. Darüber hinaus habe ich noch eine die `equals()` und `hashCode()` Funktionen überschrieben da ich später um Verlauf des Programmes sowohl Vertex-Objekte, als auch in eine Hash-Map nach dem Hash-Code vergleichen möchte. Die `toString()` Methode hier soll teil der Finalen ausgabe des Programms sein.

2.2 Graph

Zu Beginn der Graph-Klasse stand die Frage, wie ich den Graph als leicht zu veränderndes, gut auszulesendes und effizient zu skalierendes Objekt speichere. Ich habe mich hier für eine HashMap mit dem Inhalt: HashMap<Vertex, ArrayList<Vertex» entschieden, da ich in dieser effizient nach Knoten suchen kann, um ihre Nachbarn zu erhalten. Die Keys stellen die Knoten dar und die Values die Kanten, dafür nutze ich eine ArrayList mit Vertex-Objekten um die akt. Verbindungen darzustellen. Darüber hinaus kann ich die ArrayList, mit den Nachbarn gut bearbeiten, und durch die in der Vertex-Klasse implementierten Vergleichsfunktionen sicher sowohl die Keys als auch die Vertex-Objekte in den Listen vergleichen. Um den Graph aus der GML-Datei einzulesen erstelle ich zuerst in der `readGraphFromFile()` eine ArrayList mit String[] Elementen, wo jeder String[ ] eine Zeile der Datei mit gestrippten, gesplitteten Elementen enthält. Diese ArrayList verwende ich in den Hilfsmethoden `createVertex(ArrayList<String[ ]>)` und `createEdge` die Knoten und Kanten in der HashMap zu erzeugen. Als weitere Besonderheit nutzte ich in der Graph-Klasse das Interface „Cloneable“ um die Methode `clone()` zu überschreiben. Dies ist nötig um eine tiefe Kopie eines Graph-Objektes zu erzeugen. Ich möchte auf dieses Problem später unter Punkt 5 weiter eingehen.
Eine zentrale Methode der Graph-Klasse ist ist die `collapseNeighbours`-Methode, welche dafür sorgt, dass der Graph so verändert wird, dass alle Kanten des Vertex v zu seinen Nachbarn gelöscht werden, alle Kanten der Nachbarn zu dritten Knoten auf v übertragen werden und die Nachbarn zuletzt gelöscht werden. Dazu habe ich eine weitere Version der `setEdge()`-Methode implementiert, welche zwei Vertex-Objekte (v und den zu entfernenden Nachbarn) entgegen nimmt und die Kanten auf v überträgt. Dazu wird vorher überprüft ob schon eine Verbindung im Value von v hinterlegt ist, und wenn nicht wird die Kante in beiden „Richtungen“ angelegt. Zum Schluss wird der zu entfernende Knoten selbst entfernt.

2.3 Move

Ziel der Move-Klasse ist es Objekte erzeugen zu können, welche als „Verlauf“ der Lösungswege des Spieles gespeichert werden können und jeweils den Graph vor und nach einem „Zug“ und den entsprechenden Vertex speichern der zum kollabieren der Nachbarn gewählt wurde. Um dies zu Erreichen wird dem Konstruktor ein Graph übergeben, welcher den Graph vor einem Zug darstellt und der Vertex-Objekt, welcher den zu kollabierende Knoten darstellt. In der `getGraphAfter()`-Methode wird zuerst ein neuer Graph erzeugt

und dann, mit Hilfe der `clone()`-Methode der Graph-Klasse eine Tiefe Kopie des übergebenen Graphen mit der HashMap und allen ArrayLists erzeugt. Dieser kopierte Graph wird nun mit der `collapseNeighbours(Vertex)`-Methode der Graph-Klasse und dem ausgewählten Knoten manipuliert. Dieser Graph wird anschließend übergeben.

2.4 Solution

Um meine Lösungen zu verwalten habe ich das Programm um eine Klasse `Solution` ergänzt, in welcher ich als Klassenparameter eine ArrayList<Move> initialisiere, welche den Graphen lösen. Diese Klasse implementiert das `Comparable` -Interface um später im Programmverlauf dies Objekte Sortieren zu können. Neben einigen Hilfsmethoden, auf welche ich hier nicht weiter eingehen möchte, bietet diese Klasse mit der `compareTo` Methode die Möglichkeit zwei Solution-Objekte vergleichen zu können um am Ende die optimale Lösung zu finden. In dieser werden die Listen von vorne nach hinten verglichen und an der ersten stelle wo ein Unterschied auftritt wird ein positiver oder negativer Wert zurückgegeben, welcher von der sort-Methode zum Sortieren der Liste verwendet wird

2.5 Catalan

Die Catalan-Klasse hat bei mir als Klassenparameter eine neues Objekt vom Typ ArrayList<Solution>, in welcher alle möglichen Lösungen des Graphen gesammelt werden.
Die Funktionalität der `solve()`- und `doNextMove`-Methode werden in 2.4.1 erläutert. Ich möchte hier noch an dieser Stelle noch auf die `sort()`- und `main`-Methode eingehen:
Der `getBestSolution`-Methode wird von der `solve`-Methode aufgerufen und gibt eine Liste ArrayList<Move> als beste Lösung nach der Aufgabenstellung zurück. Dazu wird als erstes auf den Klassenparameter `solutions` sie `Collections.sort`-Methode angewendet und dann den ersten Eintrag (bester) zurückgegeben.
In der `main`-Methode verwende ich beim Aufruf, dass ich dieser einen String-Array übergeben kann, welcher der Pfad der GML-Datei ist. In der Methode gibt es nur einen print-Befehl der den Ausgabekopf druckt, verknüpft mit einem neuen Catalan-Objekt, welches direkt sie `solve`-Methode, mit dem Pfad aus dem `main`-Aufruf. Da `solve` eine ArrayListe zurück gibt wird hier erst der allgemeine `toString`-Methode der Liste und dann die überschriebene `toString`-Methode der Move-Klasse aufgerufen und ausgegeben.

2.5.1 Solution

Ich habe als Lösungsstrategie der `solve`-Methode einen Rekursiven Ansatz gewählt, da ich hier mit überschaubarem Aufwand eine effiziente Skalierung erreichen kann.
Die `solve`-Methode nimmt den Datei-Pfad entgegen und legt zuerst den ursprünglichen Graphen an, indem es ein neuen Graph erzeugt und die `readGraphFromFile`-Methode mit dem Pfad ausführt.
Die eigentliche rekursive Implementierung habe ich über die Methode `doNextMove` realisiert. Diese nimmt eine ArrayListe<Move> und einen Graphen entgegen und prüft als erstes die Exit-Bedingung mit einer if-Abfrage, ob der Graph gelöst ist. Es wird geprüft ob der Graph nur noch aus einem Knoten besteht, wenn ja, wird mit der übergebene Liste ein neues Solution-Objekt erzeugt und dieses dem Klassenparameter `solution` von Catalan hinzugefügt. Wenn es mehr als einen Knoten gibt wird in `else` in den rekursiven Teil der Methode übergegangen. Hier werden alle Knoten des Graphen durchgegangen und geprüft ob sie 3 Nachbarn haben. Wenn dies der Fall ist wird ein neues Move-Objekt erzeugt, welches diesen Knoten und den aktuellen Graphen enthält. Danach erstelle ich eine neue Liste als Kopie des aktuellen Move-Verlaufes und füge den neuen Move hinzu. Mit dieser Liste und dem output von `move.getGraphAfter()` rufe ich nun die Methode `doNextMove` wieder auf. Wenn kein Knoten drei Nachbarn hat, also der Graph auf diesem weg nicht lösbar ist läuft die Funktion ins leere.

3 Programming Principles

KISS

Ich habe beim schreiben des Codes versucht die Methoden so übersichtlich und so kurz wie möglich zu halten. Daher habe ich einiges an Hilfsmethoden verwendet um die Übersichtbarkeit zu erhöhen und an einigen stellen auf streams verzichtet, da diese den Code unübersichtlicher gemacht hätten als mehrere verschachtelte und eingerückte Schleifen.

Single Responsibility Principle (SRP)

Ich habe versucht in meiner Lösung streng auf die Trennung der einzelnen Funktionalitäten der Klassen zu achten. Dies bedeutet jede Klasse kapselt seine eigene Logik und ich kann auch einzeln getestet und verändert werden.

Sicherheit

Jede Methode und jede Klassenvariable ist nur so offen wie sie sein müssen. Abgesehen von den vorgaben sind alle Hilfsmethoden private, wenn sie nur von der eigenen Klasse verwendet werden.

4 Challenges

4.1 Referenzen

Mein größtes Problem bestand darin, eine tiefe Kopie eines Graphen-Objektes zu erzeugen, welches die `getGraphAfter()`-Methode zurückgeben kann. Zuerst hatte ich das Problem, dass ich immer nur den ersten Lösungsansatz erhalten habe, da ich zwar ein neues Objekt erstellt habe, aber die HashMap nicht des neuen Objektes immer noch auf die ursprüngliche referenziert war. Hier habe ich dann das Interface `Cloneable` in der Graph-Klasse eingerichtet und mit dieser die `clone`-Methode überschrieben, so dass eine sichere tiefe Kopie von Graph erzeugt wird mit einer neuen HashMap. Diese HashMap fülle ich noch mit neuen Vertex- und ArrayList-Objekten, indem ich mit einer `for`-Schleife durch die ursprüngliche HashMap gehe. Mit dieser Methode kann ich nun in der Move-Klasse einen komplett neuen, veränderten Graphen erzeugen der keine Überschneidungen in den Referenzen mit dem ursprünglichen Graphen hat.

4.2 Exception

Ich habe für die Exception, für einen Unlösbaren Graphen, an zwei Punkten eingerichtet. Zuerst wird nach dem Einlesen, bevor die eigentliche Lösungssuche beginnt, die Anzahl der Knoten Überprüft. Hier wird getestet ob die Bedingung: $|\{Knoten\}| \equiv 1 \, mod \, 3$ erfüllt ist und ob am ende überhaupt genau ein Knoten über bleiben kann. Wenn diese Bedingung erfüllt ist, wird am ende geschaut ob es überhaupt eine Lösung gibt. Wenn eine der Bedingungen nicht erfüllt ist, wirft das Programm eine UnsolvableGameException um den Fluss zu unterbrechen und dem Benutzer mitzuteilen wo das Problem liegt.