Catalan

Prinzipien der Programmierung - Objektorientierte Programmierung - WiSe 2024/2025

Addie Jordon

December 18, 2024

1 The Game of Catalan

In this assignment, you will implement and solve the game of Catalan (see `https://catalan.algochem.techfak.de`.

1.1 How to play Catalan

In the game of Catalan, the player is presented with a graph. The player can collapse any vertex of degree 3 in the graph (ie. any vertex with exactly 3 neighbours). The player must then choose which vertices to collapse and in which order. The player wins if they are able to successfully collapse all vertices except one, that is, the final graph has $|V| = n = 1$.

2 A Refresher on Graphs

An introduction to the relevant parts of graph theory was given in the lecture. You will be working closely with the *graph* abstract data type (ADT) throughout this assignment. A short summary of graph definitions is provided here for reference.

A graph $G = (V, E)$ is a set of vertices, V, and edges, E, where each edge in E is an unordered pair of vertices.

There are many different properties graphs may have, but in this assignment all graphs are **assumed to be simple: connected, undirected, and without self-loops or parallel edges**.

For a vertex $u \in V$, a vertex $v \in V$ is said to be *adjacent* to u if there exists edge $e = \{u, v\} \in E$; v is said to be a *neighbour* of u. Additionally, the edge e is said to be *incident* to both u and v. The *degree* of a vertex v is the number of edges incident to v.

2.1 Graph ADT implementations

Graphs can be represented (on computers) in many different ways. Some common implementations are summarized in the table below.

Implementation	Notes
Edge list	A list of all edges in G
Vertex list	A list of all vertices in G with links to adjacent vertices
Adjacency matrix	A two-dimensional matrix A where $A[u, v] = 1$ if edge $\{u, v\} \in E$, and 0 otherwise.

Figure 1: An example of a graph written in GML notation

```
graph [
    node [ id 0 label "0" ]
    node [ id 1 label "1" ]
    node [ id 2 label "2" ]
    node [ id 3 label "3" ]

    edge [ source 0 target 1 label "-" ]
    edge [ source 0 target 2 label "-" ]
    edge [ source 0 target 3 label "-" ]
]
```

2.2 GML notation

Graphs will be provided in `.gml` files for this assignment.

Each `.gml` file will begin with `graph [` on the first line. The following lines will be indented and will list all the vertices in the graph one by one as `node [ id x label "y" ]` where `x` and `y` do not necessarily need to match. Each node must be on a separate line from any others and no two nodes may share the same id. `id` can be any integer. **Important**: when reading in a `.gml` file for this assignment, only the node ids will be used. DO NOT USE NODE LABELS.

After all nodes are written, all edges are written, each on a new line (the same as the nodes) in the format `edge [ source u target v label "-" ]` to denote an undirected edge between vertices with id `u` and id `v` in the graph. There may or may not be a line (or multiple lines) of whitespace separating the node section from the edge section. Each edge *must* have a label, but the label will always be a single dash `"-"` for this assignment. (GML notation allows for other edge attributes to be written in the `label` field, such as edge weights or chemical bond type.)

Note: **all nodes will always come before all edges**.

The end of the file will close the opening square bracket with an unindented `]` on the final line.

3 Logistics and Details

The goal of this assignment is for you to put together a working program with very little to get you started and therefore a lot of flexibility. As a result, you will be responsible for writing nearly all aspects of the program. The methods you must write are outlined in each subsection below: **you may not change the name or omit these methods, and you must keep the exact specified return type**. You **must** implement **all** the methods listed; you may implement more for added functionality if you find it helpful, but additional separate methods will not be marked.

You are allowed to and encouraged to write your own helper methods when appropriate. If a single method feels like it is 'becoming too long' or covering too much functionality, you may want to split it into smaller pieces using helper methods.

You will need to work on and complete the following four files: `Vertex.java`, `Graph.java`, `Move.java`, and `Catalan.java`. It is recommended that you work on the files in the listed order, as later files will depend on methods from previous files.

3.1 `Vertex.java`

All methods are non-static.

3.1.1 Methods

Method	Return type	Description
`getID()`	`int`	Returns the ID of the vertex

3.2 `Graph.java`

All methods are non-static.

3.2.1 Methods

Method	Return type	Description
`Graph()`	`Graph`	A default constructor which accepts no arguments
`readGraphFromFile(String filepath)`	`boolean`	Reads in the GML file given by `filepath` and returns `true` if successful, `false` otherwise. **Note:** please do not assume any pathing; your program may be tested with files outside of the given `gml-files` folder. For example, your program should be able to handle the following arguments upon running: `java Catalan any/path/level.gml`
`numVertices()`	`int`	Returns the number of vertices in the graph
`getVertices()`	`ArrayList<Vertex>`	Returns a list of all vertices in the graph
`areNeighbours(Vertex u, Vertex v)`	`boolean`	Returns `true` if `u` and `v` are neighbours, `false` otherwise
`getNeighbours(Vertex u)`	`ArrayList<Vertex>`	Returns a list of all neighbours of `u`
`collapseNeighbours(Vertex u)`	`Graph`	If `u` has degree 3, returns a new `Graph` where everything is the same except the neighbours of `u`'s neighbours are now `u`'s neighbours (that is, if `v` is a neighbour of `u`, and `y` is a neighbour of `v`, then `y` is now a neighbour of `u`) and all original neighbours of `u` are removed. If `u` does not have degree 3, returns a copy of the current graph. **Should not directly modify the graph but return a new graph.**

3.3 `Move.java`

A move in Catalan consists of three parts: the state of the game before the move was played, the vertex whose neighbours were collapsed, and the state of the game after the neighbours were collapsed.

All methods are non-static.

3.3.1 Methods

Method	Return type	Description
`getSelectedVertex()`	`Vertex`	Returns the vertex whose neighbours were collapsed
`getBeforeState()`	`Graph`	Returns the graph before the neighbours were collapsed
`getAfterState()`	`Graph`	Returns the graph after the neighbours were collapsed
`toString()`	`String`	Overrides the default `toString()` method so that each move should have the string `select vertex x`, where `x` is the vertex whose neighbours were collapsed. Note: you can format the string how you like so long as it includes `x` somewhere easily identifiable in the string.

3.4 `Catalan.java`

The final boss of the assignment. You should now have `Vertex.java`, `Graph.java`, and `Move.java` completed and working. Come, weary traveler. It is now time to solve the game of Catalan once and for all.

Note: the `solve()` method must be implemented non-statically.

3.4.1 Methods

Method	Return type	Description
`Catalan(String path)`	`Catalan`	Constructor which takes a single argument `path`, which is the path to a `.gml` file
`solve()`	`ArrayList<Move>`	Returns an ordered list of the **fewest** moves required to solve Catalan with the initial graph given by the constructor. If there are multiple solutions, return the solution with the lowest numerical order. For example, if both vertex 3 and vertex 5 must be selected and it does not matter in which order, then the solution should select vertex 3 first (since 3 is smaller numerically than 5). If it is impossible to solve, should throw an `UnsolvableGameException` (`UnsolvableGameException.java` provided)
`main(String[] args)`	`void`	After compiling, the program is run using `java Catalan <path/to/gml>`. Thus, the main method should create a `Catalan` object, call `solve()`, print whether the game is solvable or not, and, if solvable, print the lexicographically smallest ordering of moves (in correct order) required to solve it.

3.4.2 Example

Consider the following graph (Fig. 2) which can be written in GML (Fig. 3).

Figure 2: An example graph, 'Simple Blossom'.

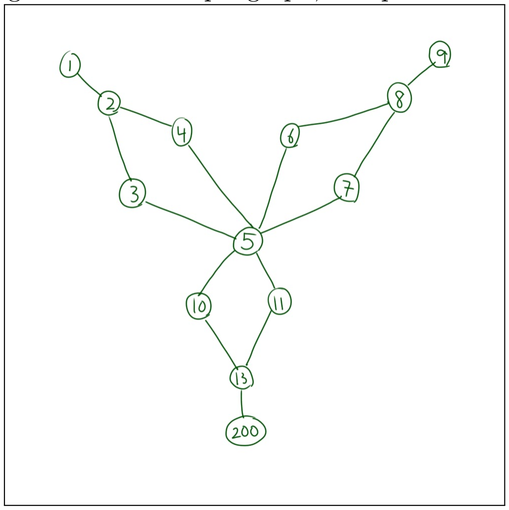

Figure 3: 'Simple Blossom' graph in GML.

```
graph [
    node [ id 1 label "0" ]
    node [ id 2 label "1" ]
    node [ id 3 label "2" ]
    node [ id 4 label "3" ]
    node [ id 5 label "x" ]
    node [ id 6 label "x" ]
    node [ id 7 label "x" ]
    node [ id 8 label "x" ]
    node [ id 9 label "x" ]
    node [ id 10 label "x" ]
    node [ id 11 label "x" ]
    node [ id 13 label "x" ]
    node [ id 200 label "x" ]

    edge [ source 1 target 2 label "-" ]
    edge [ source 2 target 3 label "-" ]
    edge [ source 2 target 4 label "-" ]
    edge [ source 3 target 5 label "-" ]
    edge [ source 4 target 5 label "-" ]
    edge [ source 5 target 6 label "-" ]
    edge [ source 5 target 7 label "-" ]
    edge [ source 6 target 8 label "-" ]
    edge [ source 7 target 8 label "-" ]
    edge [ source 8 target 9 label "-" ]
    edge [ source 5 target 10 label "-" ]
    edge [ source 5 target 11 label "-" ]
    edge [ source 10 target 13 label "-" ]
    edge [ source 11 target 13 label "-" ]
    edge [ source 13 target 200 label "-" ]
]
```

Your program should output something like the following (see Fig. 4). Note that since vertices 13, 2, and 8 can be selected in any order, the program chooses the solution ordered numerically from smallest to largest: 2, 8, and then 13. Vertex 5, however, must be selected last.

Figure 4: The output of `Catalan` after running it with the above graph.

```
$ java Catalan ../solveable/graph2.gml
SOLUTION
===========
[Select vertex 2, Select vertex 8, Select vertex 13, Select vertex 5]
```

3.5 Attributes and constructors

You are in charge of determining which attributes each class should keep and what constructors the class should provide (if no constructor methods are listed in the table).

3.6 Accessibility Modifiers

All classes and methods outlined in the sections above **must** be `public`. You are in charge of determining which accessibility modifiers (`public`, `private`, `protected`) to use for any other methods and class attributes. It is good practice to think about what needs to be seen by the user and what can be hidden.

4 Hints

4.1 Hints on using the command line

- part of the joy of working on a (relatively) small project is being able to run it easily from the command line. While it is perfectly fine to use an IDE, this project is an excellent opportunity to familiarize yourself with CLI (command-line interface). As per convention, do not include `<>` angled brackets when running the commands below. They are written with angled brackets so it is easier to identify which terms must be substituted.

- use `javac <file.java>` to compile a Java file. If the file cannot be compiled, compilation errors will be printed to the screen. When compiling a file that depends on others, Java will automatically compile the dependencies. For example, when compiling `Graph.java` (which requires `Vertex.java`), you only need to use the command `javac Graph.java` to compile both `Graph.java` and `Vertex.java`

- use `java <classname>` to run the `main` method of `classname`. For example, use `java Catalan` to run the main method of your `Catalan` class

- note: if you have added `package catalan;` at the top of your files, you should run your Catalan program from the `src` directory using `java catalan.Catalan <args>` where the lowercase `catalan` refers to the package `catalan`. Java will automatically search the directory named `catalan` for the class `Catalan`.

- if you decide you like working from the command-line so much that you never want to leave, `vim` is a very lightweight, very fun IDE that uses only CLI. After installing, use `vim <filename>` to open an existing file or create a new one. `vim` separates working modes into **insert** mode and **normal** (command) mode, where you can only edit a document if you are in insert mode. Use `i` to enter insert mode and `esc` to exit insert mode. Use `:w` to save your work and `:q` to quit. (Note: you must be in normal mode to use `:w` and `:q`). For more information, here's a basic guide: `https://www.freecodecamp.org/news/vim-beginners-guide/`. While you can achieve everything a normal IDE does using vim alone, it is often more practical to use a GUI-based IDE for larger projects.

4.2 Hints for parsing command-line arguments

- as the name suggests, command-line arguments are passed through the command-line and are usually names of files or folders that the program will need to use
- when running your Catalan program from the command-line, use `java Catalan <arg0> <arg1> ...` to pass arguments to `String[] args` in your `public static void main (String[] args)` method
- you can then get `<arg0>` by using array indexing: `args[0]`, and extract the rest of the arguments similarly
- note that all arguments are always passed as strings
- `args.length` will tell you how many arguments were passed to the program

4.3 Hints for file parsing

- it may be helpful to familiarize yourself with Java's `Scanner`: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Scanner.html`
- specifically: `scanner.hasNextLine()` and `scanner.nextLine()`
- methods for String manipulation such as `s.trim()` and `s.split("<pattern>")` may also be useful (where `s` is a string)
- parsing integers using `Interger.parseInt(String)` may also come in handy

4.4 Hints on ArrayLists

- familiarize yourself with Java's `ArrayList<E>`: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html`, particularly `add()` and `size()`
- depending on your implementation, you may find it helpful to make two-dimensional lists using ArrayLists. Java's generics allows for this: `ArrayList<ArrayList<E>>`
- If you would like resources on Generics in Java, see `https://www.baeldung.com/java-generics`, `https://docs.oracle.com/javase/tutorial/java/generics/types.html` or `https://www.geeksforgeeks.org/generics-in-java/`

4.5 Hints on `solve()`

- it may be helpful to write a helper method that computes all possible immediate next moves given a `Graph`
- this method can be written in a variety of ways. Some suggestions are using Djikstra's algorithm or implementing using recursion
- no level should take more than a few seconds to solve (it is possible to solve each level in well under one second)
- you are asked to return the solution ordered numerically from smallest to largest. You may want to be able to compare `Vertex` and `Move` classes in order to achieve this, which you can do by using the interface `Comparable` and implementing `compareTo()`: `https://jenkov.com/tutorials/java-collections/comparable.html`

- additionally or alternatively, you may want to use the `Comparator` interface and implement `compare()` to create your own comparator class. You can then use `Collections.sort(list, comparator)` to sort an `ArrayList` according to your own specific sorting criteria.
- For information on both `Comparable` and `Comparator`: `https://www.baeldung.com/java-comparator-comparable`

4.6 General hints

- it may be useful to implement `toString()` (for debugging purposes) for more than just `Move.java`
- only write comments when you think the grader will not understand what you are doing. To minimize the amount of comments, name your variables appropriately
- helper methods should almost always be `private`
- Java's 'for-each' loops may help your code readability and are useful when iterating over `ArrayLists`: `https://www.geeksforgeeks.org/for-each-loop-in-java/`
- depending on your implementation of the `Graph` class, Java's `HashMap` class might be useful: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/HashMap.html`

5 Optional Additions

If you would like to add more work to your plate, a suggestion is to add a graphical user interface (GUI) on top of your program. The GUI should **should not interfere** with the standard functionality. A suggestion is to only run the GUI when a command-line argument flag is present.

You are also welcome to add any other functionality that you desire, again, so long as it does not interfere with the base functionality.

Any extra functionality is purely optional and only for fun; it will not contribute toward your grade.

6 Submission

You must submit all your work in Java. You must submit four Java files: `Vertex.java`, `Graph.java`, `Move.java`, and `Catalan.java`. **Do not change the names of these files.**

You are given one complete Java file: `UnsolvableGameException.java`. It should not be modified in any way. You are also given four incomplete Java files that can be used as templates: `Vertex.java`, `Graph.java`, `Move.java`, and `Catalan.java`.

6.1 Written Report

Alongside your code, you must submit a document of approximately four pages which provides an introduction to the problem, an overview of all four classes (Vertex, Graph, Move, and Catalan), and a detailed explanation of your solution. You should also discuss which programming principles you employed, any additional functionality you implemented, and the challenges or bugs you encountered and how you solved them.

A LaTeXtemplate file is provided for you (`main.tex`) in which you should write your report. A suggestion is to use the free online site, Overleaf: `www.overleaf.com` to edit your Latex document online; this way you will not need to install Latex on your machine (although it is fairly simple to do so). You will need to create an account.

When submitting your written report, please export it as a PDF and **submit the PDF only**.

Your TAs will read your report prior to your in-person presentation.

7 Evaluation

The programming assignment is pass/fail and will be evaluated primarily by in-person presentation. You must prepare a 5-7 minute presentation to give to your TAs, after which they may ask questions. The specifics of the presentation are the same as in previous assignments. The following percentages are provided to give a sense of which parts will be most difficult and therefore weighted more heavily; they are to be used only as a guideline.

Part	Percentage (weight)
`Vertex.java`	5%
`Move.java`	10%
`Graph.java`	45%
`Catalan.java`	40%