Project 1-1: GRAPH COLOURING

Academic Year 2020/2021

Courses

Introduction to Data Science and AI (KEN1110)
Introduction to Computer Science 1 (KEN1120)
Discrete Mathematics (KEN1130)
Computational and Cognitive Neuroscience (KEN1210)
Introduction to Computer Science 2 (KEN1220)

Linear Algebra (KEN1410)

Graph Colouring

Summary.

A graph G is essentially a collection of points connected by lines. The chromatic number of G is simply the minimum number of colours you need to colour the points, such that no two points that are connected by a line have the same colour.

- In the first phase of the project (which lasts a single week) a computer program is to be implemented in Java which reads in a small graph from a text file and then computes the chromatic number, or - if this is deemed too computationally intensive – generates lower and upper bounds on the chromatic number. The examiners will provide a suite of test graphs.

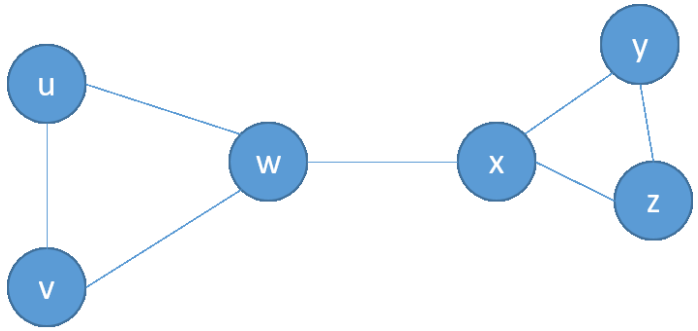
- In the second part of the project, a game will be implemented in which a human player is confronted with a random graph and is asked to colour the graph with as few colours as possible. The game has different playing modes depending on the way the performance of the player is measured. In all modes, the human player should be able to ask for computer-assisted support.

- In the third phase of the project, the work of the first phase will be continued. However, this time the graphs provided by the examiner will be potentially much larger and/or have a hidden structure that makes it much easier to compute the chromatic number than at first glance. The code developed in this third phase will be entered into a competition in which all the groups compete against each other on a hidden set of test graphs developed by the examiners. In this phase, students can optionally consider using a simple machine learning framework to learn which algorithm to use on which graphs.

1. Project description

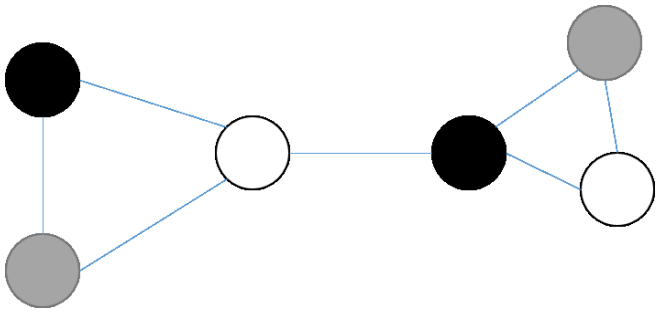
A graph G=(V,E) is simply a set of points V (known as *vertices*) connected by lines E (the *edges*). Each edge connects exactly two vertices. In an *undirected* graph there is no orientation on the edges so E is often modelled as a set of size-two subsets of V. In this project, we will deal exclusively with undirected graphs and for brevity will henceforth omit the prefix "undirected". We say that two vertices u and v are adjacent if E contains an edge {u,v}. Note: a vertex is not allowed to be connected to itself, and it is not permitted to have more than one edge between a given pair of vertices.

Here is an example of a graph on six vertices V={u,v,w,x,y,z} and seven edges E={{u,v},{u,w},{v,w},{w,x},{x,y},{y,z},{x,z}}. This graph is *connected* because it consists of one piece; graphs can also be *disconnected*, consisting of multiple *connected components*.



Graphs are one of the cornerstones of mathematics and computer science. Think, for example, of road/train networks, the wiring of the brain, social interaction networks...these are all graphs! Given their prominent role in applied computing it's important to be able to develop software/ algorithms that can query, explore and manipulate graphs efficiently.

Here we explore the topic graph colouring. A colouring of a graph G is an assignment of colours to its vertices such that no two adjacent vertices have the same colour. We are particularly interested in colourings that use as few different colours as possible. The chromatic number of G, often denoted $\chi(G)$, is defined as the minimum number of colours required to colour G. The chromatic number of the graph shown earlier is 3, because 3 colours are sufficient (as shown below), but 2 are not.



There are good reasons for wanting to know the chromatic number of a graph. For example: automated timetabling and scheduling. If we let V represent a set of university courses, and E represent “these two courses cannot take place at the same time”, then the chromatic number represents the minimum number of time slots required to schedule all the courses. Computing this minimum thus helps to make optimized use of limited resources (time, space, money). As you will discover, computing the chromatic number exactly is surprisingly challenging. It is therefore a good case study in developing code, data structures, and algorithms to tackle difficult computational problems.

Phase 1:

In Block 1.1 we want you to write a Java code which reads a small graph G in from a text file and then, ideally, computes the chromatic number of G exactly. If you encounter a graph that is too large or complex for your code to cope with, then you should aim to output both lower and upper bounds on the chromatic number that are as tight as possible. (Lower bounds are important because they give you some idea of how close your upper bounds are to the chromatic number).

To be more specific: to pass this phase, you need to develop at least one exact algorithm (i.e. an algorithm that computes the chromatic number exactly) and at least one upper bounding algorithm (i.e. an algorithm that is not necessarily guaranteed to use a minimum number of colours, but which nevertheless uses “few” colours). To achieve a higher grade you also need to code a lower bounding algorithm (i.e. an algorithm that produces some non-trivial lower bound on the chromatic number).

Your exact algorithm may not give you a solution in reasonable time for all the graphs in the provided test set. This does not necessarily mean that your algorithm is not working, but it just takes more time.

As a general rule, the greater the variety, sophistication, effectiveness, and accuracy of the algorithms you develop, the higher your score.

The code should be your own: do not use third party software and do not cut and paste code found on the internet. We will use plagiarism tools to check this.

The examiners will provide a test suite of graphs (of increasing difficulty) upon which you can test your code. At this stage you have not been learning Java for long, so the examiners will provide some ready-made Java code that reads the graph from the file into memory for you...but the rest you have to do yourself! Both the java code and the test graphs are available on Canvas.

In the demonstration you will be asked to present your approach(es), your results for the test suite and to explain briefly why your results are mathematically correct. For example, if you claim that the chromatic number of a graph cannot be 3 or less, then you should be able to explain why! Your demonstration should include a short PowerPoint presentation in which you

summarize the algorithms you have coded and your results. You might also be asked to execute your code and to show the source code.

Given that this phase of the project is only one week long, we do not expect hyper-optimized code at this stage, but any insights and optimizations you can make at this stage will help you in phases 2 and (in particular) phase 3.

Phase 2:

In Block 1.2 we want you to build a computer application with a user-friendly interface to play a simple game based on computation of chromatic number. This phase links to skills you will learn in (in particular) Introduction to Computer Science 2 (CS2).

The core of the game is as follows. There is a common core to ensure comparability between project groups. Beyond the core, you are free to customize and extend the game as you see fit.

- It is a game for one, human player.
- The computer generates a random graph which is shown on the screen. (The user should be able to specify some basic characteristics of the graph beforehand e.g. number of vertices, number of edges, etc.) Alternatively, the user should be able to read in a graph from a file.
- The player is challenged to colour the graph with as few colours as possible. He/she does this by (for example) clicking on vertices and indicating which colour they should have. The computer should warn the user if their current colouring is not valid i.e. if there exist two adjacent vertices that have the same colour.
- There are three different game modes possible. All three should be implemented.
    - To the bitter end. The player simply has to colour the graph as quickly as possible. The computer does not allow the player to finish until the minimum number of colours has been reached. (This, of course, means that your code should be able to internally compute the chromatic number…)
    - Best upper bound in a fixed time frame. The player is given a fixed amount of time and they have to find a colouring with as few colours as possible in the given time. (Here it is not necessary that the user finds the minimum number of colours).
    - Random order. Here the computer generates a random ordering of the vertices and the player has to pick the colours of the vertices in exactly that order. Once the colour of a vertex has been chosen, it cannot be changed again. (Note that the player can never get stuck because they are always allowed to introduce a new colour.) The goal for the player is to use as few colours as possible.
- In each game mode, there must be a ‘hint’ button which, when pressed, gives the user some computer-assisted help. For example, the computer might propose which vertex should be coloured next, or indicate a subset of colours that should be considered (or avoided) for a given vertex. More sophisticated hint functions will contribute to a higher grade.

Phase 3:

The third phase is conceptually similar to the first phase. However, this time the test suite of graphs provided by the examiner (at the beginning of phase 3) will be larger and more computationally challenging, so you will need to think more deeply about how to optimize your algorithms. To make life more interesting, some of the graphs will have a special structure that makes the computation of the chromatic number easier than you would expect - assuming you know how to detect, and then exploit this special structure! For example, if a graph is a tree (i.e. has no cycles) then its chromatic number will be (at most) 2, no matter how many vertices or edges it has. There are lots of other special cases. You should explore ways to both enhance the efficiency of your algorithms and to detect these “nice” special cases. If you wish you can re-use the GUI from phase 2, although this is not essential.

If you wish, you can adopt a simple machine learning approach to learn which of your algorithms work best on which graphs. (This is called the algorithm selection problem for graph colouring). However, this is not essential – and there are no guarantees it will work well! It might be a better approach to spend your time aggressively optimizing a smaller number of algorithms, and to hard-code rules for deciding which algorithm to use on which graph. The choice is yours.

At the end of the third phase, the implementations of the different groups will be compared on a hidden third dataset that the examiners will not disclose. To make this possible we will ask you to provide a version of your code without a GUI that reads input and writes output in a very specific way; you must adhere to these conventions. (This convention will be announced in due course). There will be a first, second, and third prize – and eternal glory - for the groups whose implementations perform best on this hidden dataset.

May the odds be ever in your favour!

Project examiners: E. Smirnov, E. Hortal, S. Mehrkanoon, M. Staudigl, S. Chaplick, J. Niehues