Praxis der Software-Entwicklung

Development of a Framework for Graph Games

SS 2012

Institute of Theoretical Informatics
Prof. Dr. Dorothea Wagner

April 24, 2012

1 Important Preliminary Remark

This is *your* project. This document is not a checklist of exercises that should be processed point by point to obtain the credit. It is just a list of suggestions and hints of what you can do and what we expect you to do. It is *your* decision how *your* software will look like.

2 Task

The general task of this project is the design and implementation of a *graph game framework*, a system for implementation of various games based on operations in graphs. To prove the functionality of your framework, you should use it to implement several graph games. It should have an attractive user interface and provide a possibility to play either alone or in small groups.

In the following we introduce some common notation concerning graphs, followed by a list of graph games. It is not necessary to restrict yourself to these specific games. You are free to choose your own favorite games. After the description of the games we present objectives of the project.

2.1 Definitions

A *graph* is an abstract mathematical structure that models a set of objects and a pairwise relations between them. The objects are called *nodes* or *vertices* and the pairwise relations are called *edges* of the graph. A graph is usually denoted by $G = (V, E)$, where V denotes the set of vertices and E is the set of edges. To model ordered pairwise relations between objects, *directed graph* have been introduced. In a directed graph one of the two ends of an edge is represented by an arrow. Such an edge is called *directed edge* or *arc*. The end of a directed edge without arrow is called *source* and the end with arrow it called *sink*. Two nodes of a graph are called *adjacent* if they are connected by an edge. An edge (u, v) is said to be *incident* to vertices u and v. The number of edges incident to a vertex u is called *degree* of vertex u, and is denoted by $degree(u)$. A vertex of a graph is called *isolated* if it has degree zero.

A *drawing* Γ of a graph G is a mapping p of every vertex v of G to a distinct point $p(v)$ on the plane and each edge $e = (u, v)$ of G to a simple open curve joining $p(u)$ with $p(v)$. A drawing is called *planar* if any pair of distinct edges do not intersect except at their common end-vertices. A graph is called *planar* if it has a planar drawing; see Figure 1 for an example. A drawing is called *straight-line* if straight line segments are used to draw the edges. Given a planar drawing Γ of a planar graph G, the set of points of the plane that can be connected by a curve that does

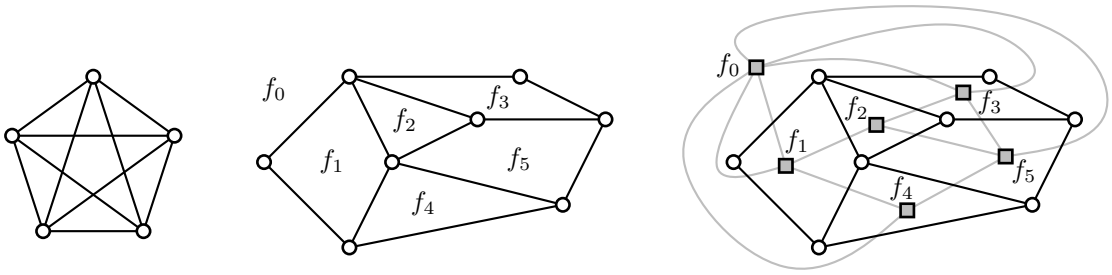

Figure 1: A non-planar graph on the left, a planar graph in the middle, and its dual on the right.

not intersect any vertex or edge of the drawing are said to belong to the same *face* of the drawn graph (the faces of the graph are denoted by $f_0, \ldots, f_5$ in the middle Figure 1). Each face of a drawing can be indicated by the sequence of edges that surround it. An *embedding* of a planar graph G is the equivalence class of planar drawings of G that define the same set of faces or, equivalently, of face boundaries.

Let G be a planar graph. The *dual* graph of G (see Figure 1) is the planar graph denoted by $G^\star$ such that: (i) $G^\star$ has a vertex for each face of G; (ii) for each edge in G incident to the faces f and g the dual $G^\star$ contains an edge (f, g).

The *subdivision* of some edge $e = (u, v)$ yields a graph containing one new vertex w, and with an edge set replacing e by two new edges, (u, w) and (w, v).

A *path* in a graph is a sequence of vertices and edges $v_1, e_1, v_2, \ldots, v_k$ such that edge e_i, $1 < i < k - 1$, is incident to the vertices v_i and v_{i+1}. A *cycle* in a graph is a sequence of vertices and edges $v_1, e_1, v_2, \ldots, v_k, e_k$ such that edge e_i, $1 < i < k - 1$, is incident to the vertices v_i and v_{i+1} and e_k is incident to vertices v_k and v_1. A cycle is called *simple* if it does not pass twice through the same vertex. A *Hamiltonian cycle* is a simple cycle that contains all vertices of the graph. A graph is called *connected* if there exists a path between each pair of its vertices.

A *cut* of a graph is a partition of its vertices into two disjoint subsets. The *cut-set* of the cut is the set of edges whose end points are in different subsets of the partition.

2.2 Some Possible Graph Games

The following list of games is just a suggestion. You may think of (or search the web for) other games related to graphs as well. On the other hand, your framework does *not* need to be able to represent all the graph games listed here.

TwixT, Bridj-It. The two-player games TwixT (`http://en.wikipedia.org/wiki/TwixT`) and Bridj-It (`http://www.sites4all.co.uk/bridjit/bridjit2.php`) can be seen as the same game with different parameters. The players are allowed to place vertices and edges of pre-specified lengths between their own vertices, without crossing the edges inserted by the opponent. The players play on a grid, i.e., they can only place vertices at the points of the grid. The first player has to build a path (a wall) from the left side of the grid to the right. The second player has to build a path from the top to the bottom. Possible parameters of the game are: (1) the lengths of the edges, (2) whether one first needs to place both endvertices (tower between two wall-segments) before inserting an edge connecting them.

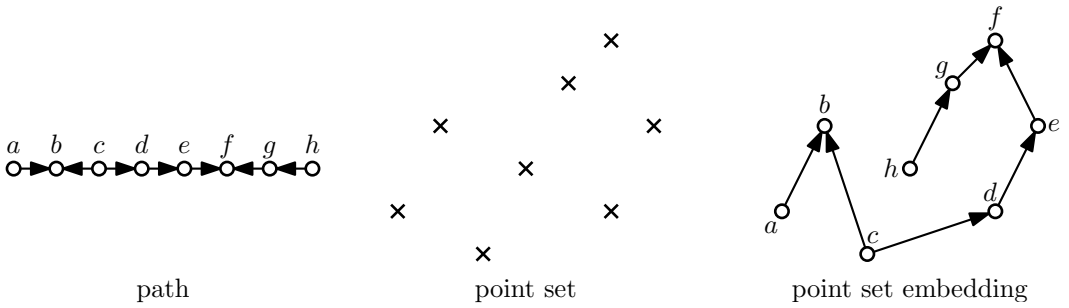

Figure 2: The input of this game is a path and a point set. The task is to find an upward point set embedding as shown on the right.

Shannon Switching Game. The Shannon switching game (`http://en.wikipedia.org/wiki/Shannon_switching_game`, or `http://kryshen.net/games/graphg.html`) starts with a graph with two special vertices s and t. Each move of the first player is to color an edge green, each move of the second player is to remove an uncolored edge. The first player wins if he is able to create a green path from s to t, otherwise, the second player wins. For planar graphs this game can be reformulated in another interesting way. Both players try to create a path, player one in the primal graph, player two in the dual graph. With this interpretation each move consists of choosing an edge not crossing an edge chosen by the opponent. Note that this is quite similar to some variants of the TwixT/Bridj-It games. In fact it is a generalization of Bridj-It.

Upward Point Set Embedding of a Path. This game is a single-player game. Each instance consists of a path of directed edges and a set of points in the plane (see Figure 2). The players task is to map the vertices of the path to the given points in the plane such that the resulting straight-line drawing of the path is planar and each edge is drawn upward.

Hamiltonian Cycle. This two-player game starts with a set of isolated vertices. One move of a player consists of picking one of the vertices and connecting it to as many other vertices as he likes (at least one edge has to be added). The first player who creates a Hamiltonian cycle wins the game. Note that this game is already almost impossible to play with more than seven vertices.

Sprouts. The two-player game Sprouts (`http://en.wikipedia.org/wiki/Sprouts_game`) starts with a planar drawing of a graph with maximum degree 3. Each move consists of two parts. First, adding an edge in a planar way without exceeding the degree of 3 property. Second, subdividing the edge inserted in the first step. The player making the last move wins. It may also be interesting to extend this game to more than two players.

Graph-Coloring. The instance of a graph-coloring game (`http://en.wikipedia.org/wiki/Map-coloring_games`) may be an arbitrary graph and a set of available colors. An allowed move is to color an uncolored vertex in such a way that none of its neighbors has the same color. The player making the last move wins. Apart from this classical version there are other variants. For example, the colors available may depend on the player, the colored region may be required to be connected or neighbored vertices must have the same color instead of different ones. For

the case that the input graph is planar it may be interesting to color the faces instead of the vertices.

2.3 Main Targets

Different graph games of course have differences, but they also have similarities. It is your task to fix a set of features that, in your opinion, every graph game should have. The implementation of these common features forms your framework that should significantly simplify the implementation of specific graph games. In the following we give a list of possible common features, some of them are required others are optional. As this is your project we encourage you to also think of features not listed here. Note that you do not have enough time to create a perfect framework implementing everything one could think of. You have to decide on key aspects and focus on them.

Game Specification. The two main quality measures of your framework are: (1) the effort that needs to be put into the specification (or implementation) of a specific graph game, and (2) its generality and flexibility making it possible to implement a wide range of different games in a customizable way. As these two requirements contradict each other, you have to find a good trade-off between them. You may for example provide a bunch of parameters making it possible to customize each game but specify reasonable default values to simplify the game specification.

There are two possibilities how a single game may be specified. One is through the inheritance of an abstract game class your framework provides. The second is by introducing a metalanguage providing the possibility to formally describe a game. The latter possibility is not an easy task. It is necessary to find a trade-off between the generality of your metalanguage and its complexity as it should support a wide range of graph games and, on the other hand, should not be too complicated. If you choose the option to specify a metalanguage, you should plan your time very carefully and restrict other parts of your framework to necessary aspects.

Data Structures and Graph Algorithms. As all games are based on graphs, you can store the current state in the game in a graph data structure. Depending on the game, the data structure needs to be enhanced by additional information such as vertex positions, colors, etc. You should develop a general data structure that can be used for all your games.

You will probably need some basic graph algorithms such as testing connectivity for the "Shannon Switching Game", or planarity for "Sprouts" and "Upward Point Set Embedding of a Path". You may use an existing graph library (for example JUNG[1]) or develop the algorithms yourself.

Graphics. A data structure storing the current state is worth nothing if it is not presented to the user. Thus, you have to create a graphical representation showing it to the user. Following the model view controller principle it should be decoupled from the internal data structure. When implementing a specific game you should not need to deal with the graphical representation, but you may want to have easy accessible options to customize the appearance of each game. For example in the game TwixT you may want the framework to draw a small image showing a tower for each vertex.

[1] `http://jung.sourceforge.net/`

Input Model. Of course the user needs the possibility to interact with the system. When implementing a specific game you do not want to deal with user interactions such as "player A clicks at position (287, 17)" but with higher level inputs such as "player A deleted edge e". The framework should provide such a higher level communication between the system and the user.

Artificial Intelligence. As an optional feature you may implement an adversary controlled by an artificial intelligence. As you should create a framework, the classical minimax algorithm[2], perhaps enhanced by an alpha-beta pruning[3], is well suited. Once this general algorithm is implemented, it "only" remains to define a position evaluation function for each game, assigning a value to every possible situation, specifying how "good" the current state is.

Network. A game is not a game if it cannot be played with a friend. Thus, you may think of connecting two players over a network. Then you have to specify a protocol describing each move with a short string that has to be sent from one computer to another. If you do not want to care about building a network connection between different computers, you may instead show this string to the player, who then sends it using an external messenger such as Skype or ICQ (although it is less comfortable than a build in network connection). If your framework supports playing over network without any external tools, you may also think of a simple chat allowing to manipulate the opponent.

Note that your System should be sufficiently modular that it does not make such a big difference whether the next move comes from a player clicking somewhere, from an AI or from the network.

Tutorial. Your system should be self-explaining, that is a user should be able to play without knowing the rules of the games in advance. You may for example simply provide a PDF-file containing the manual with your system. A more sophisticated possibility is to develop a tutorial mode teaching a new player interactively. If you decide to make a tutorial mode, the additional effort that has to be put into the implementation of a specific game to contain a tutorial should be as small as possible.

3 Working Process

3.1 Operating Principles

In this course you should use software design and quality assurance methods in practice, use and improve implementation skills, and collaborate in a team distributing the work among its members. Accordingly, the organization of this course is guided by software engineering principals, as you can also see in the schedule in Section 3.4.

It is not the aim to somehow implement a system that somehow works, but to go step by step through the phases of software engineering as prescribed by the schedule. You do not only have to hand in a working system at the end but all documents corresponding to each phase:

- Functional specifications document (dt. Pflichtenheft; about 20 pages)
- Software design (dt. Entwurf; about 40 pages)

[2] `http://en.wikipedia.org/wiki/Minimax#Combinatorial_game_theory`
[3] `http://en.wikipedia.org/wiki/Alpha-beta_pruning`

- Implementation report (about 20 pages)
- Test report (about 20 pages)

At the end of the course you have to present your finished and working system. Moreover, a colloquium takes place at the end of each phase. The particular appointments will be arranged during the semester.

The schedule in Section 3.4 is not just a suggestion to help you developing your system, we expect you to follow it. To this end, in each phase a *team leader* is determined who is responsible for a correct execution of this phase and for submitting the documents in time.

3.2 Documentation

Creating a documentation is a major part of your project. Without a complete and extensive documentation of your work it is not possible to successfully pass this course.

We expect you to document your planning and the execution of your plans precisely, completely, and consistently. Concerning the content and the form of the documents we explicitly refer to the software engineering lecture.

You must set up and use a version management tool for both, the source code **and** the documentation. We recommend to use Subversion or GIT.

3.3 Recommended Tools

The recommended programming language for this course is Java (version 1.6). Using a different language is only in exceptional cases and after agreement allowed.

The JUnit Testing Framework[4] is the de facto standard tool for unit tests when using java. We strongly recommend to use this tool. Basic literature on how to handle JUnit can be found on the web or in the library. The tool CodeCover[5] helps you computing your code coverage. Both tools are integrated as plug-ins into the development environment Eclipse[6] that we recommend to use as IDE. For the software design it is recommended to use an UML-tool. Umbrello[7] is part of the KDE-project and thus free to use.

3.4 Schedule

- **First group meeting:** 24th of April, 2012
- **Delivery of the functional specifications document:** (30.04–20.05). Shortly after this you are required to explain your document in the first colloquium (date for this colloquium by arrangement).
- **Delivery of the software design document:** (21.05–17.06). Shortly after this date you have to defend your software design in the second colloquium (date for this colloquium by arrangement).

[4] `http://junit.org/`
[5] `http://codecover.org/`
[6] `http://www.eclipse.org/`
[7] `http://www.umbrello.org/`

- **Implementation:** (18.06–15.07)
- **Delivery of the implementation report:** (18.06–15.07), shortly after this date there will be a colloquium where you need to explain your implementation report (date for this colloquium by arrangement).
- **Break (exams):** from 16th to 29th of July
- **Delivery of the test report:** (30.07–19.08)
- **Final presentation:** (10.09 – 16.09)

Please note that this schedule is preliminary only. It is possible that we change some of the dates slightly. If this is the case we will make an announcement well enough in advance.

3.5 Contact Information

- **Thomas Bläsius**, Institute of Theoretical Informatics, Prof. Dr. Dorothea Wagner. Email: `thomas.blaesius@kit.edu`, tel.: 0721 608-44322, office: room 316, building 50.34.
- **Andreas Gemsa**, Institute of Theoretical Informatics, Prof. Dr. Dorothea Wagner. Email: `gemsa@kit.edu`, tel.: 0721 608-47331, office: room 322, building 50.34.
- **Tamara Mchedlidze**, Institute of Theoretical Informatics, Prof. Dr. Dorothea Wagner. Email: `mched@iti.uka.de`, tel.: 0721 608-44245, office: room 307, building 50.34.

4 Grading

4.1 Minimum Capability Characteristics

Your framework should provide basic building blocks to simplify the implementation of games based on graphs. Basic ingredients your system must contain are the following.

- A model that is able to represent the current state of a game and a graphical user interface respecting the model view controller principle.
- Your framework must provide the functionality to easily formulate new games.
- You should implement at least two not too similar games to prove the generality of your framework.

4.2 Reference System

Your software must be able to run on a Linux PC. For testing purposes there are Linux PCs (with Suse 11.3) available in room 304. These PCs are equipped with an Intel E7200@2.66 Ghz processor and 4GB of RAM.

4.3 Grading

The grade for your project depends on the following criteria:

- quality of all your delivered documents
- quality of all colloquia
- meeting all of the minimum capability characteristics (see Section 4.1)
- useful extensions to the software that surpass the minimum capability characteristics
- robustness of your software

This list has *no* specific order. In particular the order of this list *does not* reflect any weighting of the criteria for the final grade.

The grade for the team-work in this practical course is a grade for the so called *soft skills*. The most important aspects for these skills are the presentations you give, how well you work as a team and the performance of the team leader during each phase.

For passing the practical course it is required that all documents mentioned in Section 3.1 are delivered *on time*. For a schedule see Section 3.4.

Each phase of your project is graded. The grades of each phase will be combined for the final grade of your project. See Table 1 for a detailed overview on how much the grade for each phase influences the final grade for the project.

phase	percentage
Functional specifications document (dt. Pflichtenheft)	10%
Software design (dt. Entwurf)	30%
Implementation report	30%
Test report	20%
Final presentation	10%

Table 1: weighting of the grade for each phase