GraphBench: Exploring the Limits of Complexity with Educational Software
Exploring the limits of complexity with educational software

Author(s):
Brändle, Markus A.

Publication Date:
2006

Permanent Link:
https://doi.org/10.3929/ethz-a-005128663

Rights / License:
In Copyright - Non-Commercial Use Permitted
GraphBench: Exploring the Limits of Complexity with Educational Software

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Sciences ETH Zürich

presented by

Markus Andreas Brändle
Dipl. Informatik-Ing. ETH

born 22.05.1977
citizen of Mosnang SG

accepted on the recommendation of
Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Jürg Nievergelt, co-examiner
Prof. Dr. Angelika Steger, co-examiner

2006
Acknowledgements

This dissertation and the software GraphBench resulted from the support and contributions of many people to whom I am very thankful and wish to express my gratitude.

I first would like to thank Jürg Nievergelt and Werner Hartmann for inspiring me to work on this thesis. Jürg Nievergelt has been an extraordinary supervisor, advisor and personal mentor throughout my whole dissertation. His interest, commitment, ideas and visions are the cornerstones of my work and for that I am very grateful. Werner Hartmann was the didactic voice and his comments and suggestions account for the usability of GraphBench. I would especially like to thank him for being a friend I could always talk to.

I am also indebted to Bertrand Meyer for allowing me to finish my thesis under his supervision. I greatly appreciate his contributions and his interest in a topic originally not within his research area.

I was very glad that Angelika Steger agreed to be co-advisor. Peter Widmayer and Juraj Hromkovic deserve my gratitude for the time I spent in their groups and for their support.

Matthias Dreier was involved in the beginning of the project and was very much responsible for the good start and the solid software foundation of GraphBench. Raimond Reichert, Ruedi Arnold and Judith Zimmermann greatly supported my work with their time, comments and suggestions.

Many others have directly or indirectly contributed to my work and deserve to be mentioned: Nina Huggenberger, Floris Tschurr, Hon Wai Leong and the whole reasearch group of Betrand Meyer, in particular Michela Pedroni and Till Bay.

Finally, I thank Ursina, my parents and my brother for putting up with me and taking my mind of computers every once in a while.
Abstract

In today’s information society it has become ordinary for students to use computers to study. Educational software is increasingly being developed for all kinds of subjects and school levels. Although a broad range of computer science areas is covered, topics that admit intuitively meaningful graphic representations are almost exclusively considered. Mathematically abstract topics that are more difficult to present, as well as to understand, still challenge our ability to create effective computer support. The theory of NP-completeness in particular has been almost neglected by educational software, despite its importance in the theory of computation and in computer science education. Yet it is precisely in such a highly specialized, abstract topic that computer-based learning environments might do their best job.

In this thesis we have developed GraphBench, a highly interactive learning environment for the theory of NP-completeness. GraphBench, as well as the techniques and principles on which it is built, is the main contribution of this dissertation. Our software provides students with an intuitive approach to an otherwise abstract and complex topic. GraphBench features eight different NP-complete problems and nine different polynomial time reductions. Our software offers separate environments for all featured NP-complete problems and polynomial time reductions.

GraphBench combines traditional didactic with new computer-based approaches to foster an intuitive understanding of the basic underlying concepts. In this thesis we identify difficulties that arise when developing educational software for abstract topics and present several didactic concepts and implementation approaches to overcome them.

We have successfully used GraphBench in various courses on the theory of computation at the Swiss Federal Institute of Technology ETH, at the National University of Singapore and at the Free University of Bolzano. The highly positive feedback from professors and students justifies our approach and shows the need for learning environments for highly specialized, abstract topics.
vi
Zusammenfassung


Im Rahmen dieser Arbeit haben wir GraphBench entwickelt, eine hochgradig interaktive Lernumgebung für NP-Vollständigkeit. GraphBench, wie auch die zugrunde liegenden Technologien und Prinzipien, stellen die Hauptbeiträge dieser Dissertation dar. Die Lernumgebung bietet eine intuitive Einführung in ein sonst abstraktes und komplexes Thema. GraphBench beinhaltet acht verschiedene NP-vollständige Probleme und neun verschiedene Reduktionen. Die Lernumgebung bietet separate Umgebungen für alle NP-vollständigen Probleme und Reduktionen.

GraphBench verbindet traditionelle didaktische und neue computergestützte Ansätze um ein intuitives Verständnis grundlegender Konzepte zu vermitteln. Wir haben Schwierigkeiten die beim Entwickeln von Lernumgebungen für abstrakte Themen auftreten identifiziert und präsentieren verschiedene didaktische Konzepte und Implementierungsansätze um diese zu überwinden.

Wir haben GraphBench erfolgreich in Vorlesungen zur theoretischen Informatik an der ETH Zürich, der National University of Singapore und der Freien Universität Bozen eingesetzt. Die sehr positiven Rückmeldungen von Professoren und Studenten bestätigen unseren Ansatz und zeigen die Notwendigkeit von Lernumgebungen für spezialisierte, abstrakte Themen.
# Contents

1 Overview
   1.1 Motivation .......................................................... 1
   1.2 Contributions ...................................................... 2
   1.3 Conclusions ........................................................ 3

2 Computer-based learning environments: State of the art 5
   2.1 Computer uses in education: Medium of presentation and
      subject matter tutor .............................................. 5
   2.2 Learning environments in computer science education .... 7
   2.3 Tackling abstraction and complexity .......................... 15
   2.4 Our contribution: GraphBench ................................. 17

3 Subject matter: Theory of Computation 19
   3.1 The foundation of computer science education ............... 19
   3.2 Complexity theory: NP-completeness and polynomial time
      reductions .......................................................... 21
   3.3 A need for computer-based learning environments .......... 23
   3.4 GraphBench: Choosing the right content ..................... 24

4 A first session with GraphBench 27
   4.1 Satisfiability ..................................................... 28
   4.2 Reducing 3CNF to Graph Colorability ......................... 37

5 Didactic concepts in GraphBench 45
   5.1 Example generation .............................................. 45
   5.2 Solving problem instances ...................................... 47
   5.3 Pseudo-code ..................................................... 49
   5.4 Algorithm animation ............................................. 50
   5.5 Multiple problem representations ............................. 53
   5.6 Relations among structures for reductions ................. 54
   5.7 Selective level of detail ....................................... 55
Chapter 1

Overview

In this chapter we give a short overview of this thesis. We summarize our motivation and the work that has been done. We further give a short introduction to our learning environment GraphBench and shortly look at its use.

1.1 Motivation

The theory of computation teaches students the limits of computers, e.g. what a computer can and cannot compute. It is fundamental to computer science and thus an important part of any computer science education. Despite of its importance students often have difficulties with the theory of computation and lack motivation to study it. Among the topics that students typically struggle with is the theory of NP-completeness and polynomial time reductions. This theory defines a class of problems that are difficult, i.e. very time consuming, for a computer to solve.

Computer-based learning environments can help students overcome their difficulties and help them gain a better understanding. While some topics of the theory of computation are well covered by educational software, the theory of NP-completeness has been almost neglected. It is a highly abstract and rigorously formal topic. This not only makes the theory difficult for students to understand, but also the development of a learning environment for this topic challenging. Topics that are either only highly abstract or only rigorously formal are easier to understand. Students have for instance little problems to use the learning environment “Tarkis world” (Barwise and Etchemendy, 1997) that covers the very formal but not very complex topic of first-order logic. Computer games on the other hand are often highly complex, but because they are also quite intuitive they are well feasible to master. It is the combination of abstraction and formality that challenges the human mind and makes the
design of a learning environment difficult (figure 1.1).

Fig. 1.1: Software for either a only highly complex or a only rigorously formal topic is easier to master than software for a topic that is both.

The importance of the theory of NP-completeness, students’ problems with the topic and the lack of learning environments motivated us to approach the theory of complexity and design a learning environment for NP-completeness.

1.2 Contributions

In this thesis we focus on complexity theory. We have developed GraphBench, a highly interactive learning environment for the theory of NP-completeness, i.e. for NP-complete problems and polynomial time reductions. GraphBench provides students with an intuitive approach and combines traditional didactic concepts with new computer-based methods. It fosters an intuitive understanding of the basic underlying concepts and allows students to experiment and thus to discover the theory of NP-completeness.

GraphBench features eight different NP-complete problems, e.g. Satisfiability or Graph Colorability, and nine different polynomial time reductions, e.g. reducing 3CNF Satisfiability to Clique or Vertex Cover to Hamilton Circuit. GraphBench offers a separate environment for every problem and for every reduction. The environments all share a common
user interface and similar functionality. The problems and reductions were selected according to several criteria, e.g. their frequent use in computer science education.

GraphBench actively involves students and provides them with a “hands-on” approach. Students can for instance manipulate and create new problem instances, or actively investigate polynomial time reductions. The learning environment allows students to solve arbitrary examples by hand or to use built-in solution algorithms. GraphBench fosters an intuitive understanding by providing various, intuitive graphical representations of the NP-complete problems, polynomial time reductions, and solution algorithms.

In this thesis we have identified difficulties that arise when developing educational software for abstract topics and present several didactic concepts and implementation approaches to overcome them, e.g. “Selective Level of Detail”. We illustrate the concepts with examples and state their learning benefits.

This thesis consists of work in the field of computer science education and didactics as well as work in the field of software engineering. With GraphBench we have developed a Java software framework for computer-based learning environments. It consists of several independent libraries that can be reused to create learning environments for topics other than complexity theory. The framework is designed to facilitate maintaining and extending existing content, e.g. adding new NP-complete problems or solution algorithms.

We have successfully used our learning environment GraphBench in various courses on the theory of computation at the Swiss Federal Institute of Technology ETH, at the National University of Singapore and at the Free University of Bolzano. It was used as a demonstrational tool in class as well as for student assignments and semester projects.

1.3 Conclusions

With GraphBench we have developed a learning environment for a topic that has otherwise been almost neglected by educational software. The fundamental importance of complexity theory in computer science and student’s struggle with the topic show the need for such a learning environment.

GraphBench conforms to today’s criteria for educational software, e.g. a high level of interactivity or active user involvement. We have developed a software framework and new didactic approaches that can be adapted for other topics.

The widespread use of GraphBench at universities worldwide, and the highly positive feedback from professors and students, convince us that our approach is successful and worth pursuing.
Chapter 2

Computer-based learning environments: State of the art

Computer-based learning environments are omnipresent in today’s education society and can be found for a wide variety of topics. In this chapter we first analyze the different uses of computers in education and show our idea of utmost computer benefit on learning success. We point out requirements and criteria for successful computer-based learning environments.

We then focus on computer science education and show the wide range of topics currently addressed by learning environments. We briefly introduce several learning environments and point out where we see room for improvement.

2.1 Computer uses in education: Medium of presentation and subject matter tutor

In today’s information society it is not uncommon for students to use computers to study. Educational software is increasingly being developed for all kinds of subjects and school levels. Traditional learning environments like classrooms include infrastructure, information, exercises, tests, teachers or tutors. How do we apply these traditional concepts to computer aided learning? What is needed to create a computer-based learning environment?

Today the computer is mostly used as versatile presentation medium that combines the capabilities of a number of traditional media such as books, movies or audio. The cognitive theory of multimedia learning supports the use of such multimedia presentations. [MM02] state for instance, that animations can improve learning when used in ways that are consistent to the theory of cognitive multimedia learning. This theory is based on three assumptions: 1) dual-channel assumption – the idea that humans possess separate information
processing channels for visual and auditory material, 2) limited capacity assumption – the idea that humans are limited in the amount of information they can process at one time in each channel, 3) active processing – the idea that meaningful learning occurs when learners engage in cognitive processes, e.g. selecting relevant material (see e.g. [May01]).

Using computers and multimedia does not exhaust the strength of computers in learning by far. Computer-based technologies present a number of possibilities to further improve learning. Educational software can for instance provide access to a large collection of documents and facts. An additional advantage, compared to traditional media, is obtained if the data is kept up-to-date. In business and finance, for example, the software might access daily currency exchange rates and stock quotations that are freely available on the Internet.

Today’s educational software often does not allow students to change the texts, narrations, sounds or animations, e.g. to change the initial configuration of an animated experiment. However, allowing students to manipulate underlying objects, such as examples or solutions, can improve learning. For instance, instead of seeing a pre-computed animation, the student can now configure the initial parameters and then watch a simulation based on his settings.

An even greater benefit arises when the software is able to produce new content depending on actions by the student, e.g. generate an unlimited number of exercises with various difficulty levels, solve arbitrary examples or provide meaningful feedback to actions or questions by the student. This implies that it must embody a model of its domain of discourse that enables it to derive facts not explicitly foreseen and stored. We refer to such systems as subject matter tutors. In this thesis we have developed a learning environment containing several subject matter tutors, we introduce the system and the subject matter tutors in detail in chapters 6, 7.

Schulmeister introduced six levels of interactivity to classify educational software [Sch02b]. Serving as a medium of presentation allocates educational software to levels 1 or 2, depending on the user’s ability to choose from different forms of representations. If the user can vary the form of the representation, e.g. by scaling or rotating a diagram, the third level is reached. The fourth level of interactivity is characterized by allowing the user to manipulate the content, e.g. setting the parameters for a simulation. A high level of interactivity can be achieved if the user cannot only manipulate the content, but can create new content as well. A subject matter tutor reaches level 5 if it allows students to create individualized exercises and solutions. If it additionally provides meaningful feedback the highest level of interactivity is attained.

A computer-based learning environment combines aspects of all three func-
tions: medium of presentation, data base, and subject matter tutor and thus exploits the computer’s power of computation to the utmost. [Rie02] stated that “Producing more referential connections should be expected when a user has the opportunity to interact with information in meaningful ways, especially given a variety of multiple representations.” The question is: under what conditions, in what disciplines, for what audience, to what extent can software be developed to achieve all three goals? This question is unlikely to admit a general answer, and must be investigated anew for any specific situation. However, computer science education seems to be well suited, mainly due to its formal nature.

2.2 Learning environments in computer science education

Computer science benefits from numerous learning environments ranging from small, stand-alone, web-based applets to full sized applications. The demand and the domain for such software is almost endless, from introductory programming to the theory of computation, from logic to compiler design. While certain areas, such as introductory algorithms and data structures, are well covered, others have almost been neglected, such as advanced topics in the theory of computation.

In this thesis we focus on the theory of NP-completeness. An advanced topic in the theory of computation that is only sparsely covered by educational software. In the following we survey the current state of learning environments concentrating on introductory programming and the theory of computation. We will briefly introduce our educational software GraphBench and list its contributions.

Introductory programming

Among the first learning environments were ‘mini-environments’ for learning programming. These mini-environments use a small programming language to control an actor within a virtual world. Starting with Logo in 1970 and Karel the robot [Pat95] in 1981 various mini-environments have been created and are widely used, such as Kara [Rei03].

For more advanced students a large variety of simulations and animations of abstract data type implementations, such as arrays, linked lists, trees or hash tables are available. They typically provide a graphically animated visualization of the standard operations (i.e. insert, delete, traversal) and user
interaction is often limited to calling these standard operations and defining insert values.

Educational software can be found for even more advanced topics in learning programming, such as ‘Concurrency’ (figure 2.1) or ‘Distributed Algorithms’. They are typically animation and simulation tools, rather than highly interactive systems.

![Simulation for the producer consumer problem](http://cities.lk.net/approco.html). The applet simulates one producer and up to four consumers and allows the user to start and stop processes.

Fig. 2.1: A simulation for the producer consumer problem (http://cities.lk.net/approco.html). The applet simulates one producer and up to four consumers and allows the user to start and stop processes.

We describe two representative systems for introductory programming, Kara and RoboCode, in more detail.

**Kara**

Kara is a beginner’s programming environment and is based on the concept of finite state machines. It allows students to program the ladybug Kara, a virtual actor in a simple, grid-like world, using finite state machines. The lady-
bug has sensors such as **tree in front?**, **on leaf?**, and it has commands like **move**, **turn left**, or **pick up leaf** (figure 2.2). The Kara environment allows students to execute their finite state machines step by step or in “movie-mode” with different speeds. Students can choose from a collection of exercises, that come with different world settings and solutions. They can also create their own problem settings.

Figure 2.2 shows an example where the task is to guide Kara out of the maze and onto the cloverleaf. The solution is a finite state machine with the two states “Exit left?” and “Exit right?” . The state “Exit left?” is used to make Kara walk from left to right, while looking for an exit on the left side. If an exit is detected, Kara will move up to the next row. The state “Exit right?” works similar with the difference that Kara walks from right to left and looks for an exit on the right.

Besides the Kara environment the software also includes the MultiKara and the JavaKara environments. The MultiKara environment allows students
to control up to four actors and introduces the concept of monitors and critical sections, giving students a first look at concurrent programming. Figure 2.3 shows an example with two ladybugs, where the task is for them to walk endlessly along a corridor of trees (in opposite directions) without colliding. The corridor itself is wide enough for the Karas to pass each other without problems, but there are sections narrowed by mushrooms that only leave room for a single Kara. JavaKara, on the other hand, allows students to write the programs using Java instead of finite state machines. It aims to decrease the gap between the learning environment and real world programming.

In addition to the three environments aimed for introductory programming, the Kara learning environment also includes TuringKara. With TuringKara students can design and operate two-dimensional Turing machines [Tur37]. The use of a two dimensional ‘tape’ does not affect the computational power of the Turing machine, but it is useful for educational purposes because it facilitates problem solving.
CeeBot

c is an other environment for novice programmer, where the student programs a robot acting in a continuous 3D world [Eps00]. CeeBot supports several levels of complexity, ranging from simple turtle graphics which are programmed non-textually, to an object oriented model similar in syntax to Java. An integrated debugger, on-line compiler error feedback and a browsable code library support the student during the exercises. CeeBot has a rich library of problems, complete with task descriptions, code stubs and full solutions. The sophisticated nature of the environment is visually attractive, but can limit the multitude and openness of solutions.

Fig. 2.4: The CeeBot environment

Theory of computation

The theory of computation is also a well covered domain in computer science education, offering a large collection of learning environments. Among the best represented topics are finite automata, formal languages and grammars and Turing machines (figure 2.5). Educational software for these topics often offers a high level of interactivity and user involvement. Because these systems rely on a precise mathematical framework, they can generate examples automatically, and let students not only select from predefined examples but make up their own.
The best covered topic in the theory of computation by educational software is Turing machines. The learning environments that can be found range from simple animations to highly visual ‘programming’ environments which let students design and execute Turing machines of their own. We introduce Turing’s world as an example. We also take a look at Exorciser, a learning environment covering various basic topics of the theory of computation.

**TURING’S WORLD** (Barwise and Etchemendy, 1993)

Turing’s World aims to give students a hands-on introduction to Turing machines. It allows students to graphically generate Turing machines by drawing their state diagram and executing them. Turing’s World helps students to debug their Turing machines by offering different execution speeds and allowing them to step through the computation - forward and backward - one step at a time. In addition to deterministic Turing machines the software also supports the design and execution of finite state machines and non-deterministic automata.

**EXORCISER**

Exorciser is a learning environment for an introductory course on the theory of computation, covering standard topics such as finite automata, formal languages and computability [TLN02], [Tsc04]. Exorciser is able to generate
an unlimited number of distinct problem instances of varying complexity with parameter values that are set by the user or at random. The key characteristic of Exorciser is that it grades not only the student’s final solution, but provides meaningful feedback to every step of the solution, if the student so desires. The ability to provide step-by-step comments is based on the concept of a solution space for a given type of problem. A structure that captures all the consistent sequences of operations for solving problems of that type by following appropriate algorithms.

The scope of learning environments is not limited to introductory programming and the theory of computation. There are probably no topics for which no learning environment can be found. Operating systems, databases, logic or neural networks are only a few other topics for which educational software exists. We look at Tarski’s World as a representative in more detail.

Fig. 2.6: Exorciser: interactive state minimization
TARSKI’S WORLD (Barwise and Etchemendy, 1997)
The goal of Tarski’s World is to introduce students to the theory of first-order logic. Students quickly master the meaning of operators and quantifiers using this program. The software allows students to construct three-dimensional worlds using blocks of various shapes and sizes and to describe them in first-order logic. The students can evaluate their sentences for correctness, receiving feedback that shows where their solution is wrong.

Fig. 2.7: Tarski’s World

The high level of interactivity and the possibility to solve exercises make students feel involved with the subject matter. Tarski’s World allows students to test their mastery of basic concepts while the systems fine-grained feedback pinpoints the source of any misunderstanding.
Theory of complexity

Only a few learning environments can be found for the theory of NP-completeness covering only a small fraction of the topic. For NP-complete problems the Traveling Salesman and Satisfiability problems seem to be the only two examples that offer several tools (figure 2.8 and 2.9). For polynomial time reductions educational software seems not to exist.

Fig. 2.8: Java applet for the Traveling Salesman problem

2.3 Tackling abstraction and complexity

Although educational software covers a broad range of computer science areas, coverage is biased in favor of topics that admit intuitively meaningful graphic representations. Mathematically abstract topics that are more difficult to present, as well as to understand, still challenge our ability to create effective computer support. Among these challenging topics is the theory of complexity addressed in this thesis.
One explanation might be the high level of abstraction itself. [BM85] lists the following definitions:

abstract: detached from any specific instance or object; difficult to understand, abstruse

abstraction: an abstract idea or term rigorously stripped of any concrete application or reference

The constructivist learning theory states that the construction of new knowledge is performed individually, based on previous knowledge. The process of acquiring knowledge becomes more difficult for abstract topics because students lack real world objects or problems to relate to. [Tri99] states that “Cognitively plausible representation is a fundamental problem to every theory of instruction”. Therefore the requirements for educational software introducing abstract subject matters must include finding good representations to facilitate the construction of knowledge. Fulfilling this additional requirement is a demanding task and thus might be a hindrance to the development of learning environments for abstract topics.
In addition to conceptual difficulty, a quantitative problem arises: meaningful examples for abstract topics when presented in a rigorous manner often generate more data than the human mind can absorb [Mil56]. A typical example to illustrate a theorem, for instance, easily claims several pages of highly compressed argumentation in a textbook, and may require an hour to present and understand in detail. The problem is not that any single step in working such an example is particularly difficult. The challenge lies in the complexity of the entire argument. It requires keeping in mind simultaneously concepts, notations, facts, and assumptions that stretch over several pages of a book, or over a significant part of a lecturer’s presentation. Helping the student to understand the core of such an argument, rather than merely following its individual steps, is an intrinsically difficult task using any medium of presentation.

It is precisely in such a highly specialized, abstract topic such as the theory of complexity that educational software might do its best job. It can help a student truly understand the interconnection among the many elements that make up a lengthy chain of reasoning in a complicated mental structure. Rather than linearly following the trail laid down by an author or a lecturer, the student can explore this structure under his own initiative, heading in any direction that arouses his curiosity. The learning environment serves as a tutor offering information and support depending on the student’s behavior. In the presentation of advanced topics we may assume that the student is sufficiently motivated to undertake the sustained work necessary to take advantage of the exploratory features offered by the software.

Abstract topics in ‘exact’ sciences (e.g. Mathematics, Physics, Computer Science) are often based on proofs and abstract theory. However, there are topics for which a full knowledge of these proofs and theory is not necessary for most students. Often an intuitive knowledge, an understanding of the basic concepts, is sufficient. When designing learning environments this should be kept in mind and the goal of educational software should be to facilitate such an intuitive understanding. This kind of approach has been around for years. We do not teach children the law of gravity by showing them physical formulas, but simply by showing them the effect of gravity. We let them discover.

2.4 Our contribution: GraphBench

We have developed GraphBench a learning environment for the theory of NP-completeness. This topic has otherwise been almost neglected by educational software. Our learning environment provides students with a hands-on approach to a selection of NP-complete problems and polynomial time reductions and promotes an intuitive understanding of the underlying concepts.
GraphBench combines aspects of a multimedia presentation, data base and subject matter tutor while offering a high level of interactivity. Similar to Kara for instance it allows students to execute algorithms with different initial configurations and in different modes. Our learning environment is for example also capable to generate arbitrary examples and allows students to create examples of their own.
Chapter 3

Subject matter: Theory of Computation

In this chapter we first argue why we believe that the theory of computation is a fundamental part of any computer science curriculum. We review basic concepts of complexity theory, the domain of discourse for our learning environment. We introduce our educational software GraphBench by means of a typical example and list its contributions.

3.1 The foundation of computer science education

Done properly, the theory course puts the "science" into computer science, giving aspiring practitioners a basis for understanding the fundamental laws that govern their discipline: there are problems that cannot be solved, there are intractable problems, there are limitations on the efficiency of the solutions to problems, and so on. [RGKL02]

The theory of computation teaches students the limits of computers, e.g. what a computer can and cannot compute. The concepts are as fundamental to computer science as Newton’s laws to physics. [Nie95] presents a layered tower of computer science with the theory of computation as the foundation (figure 3.1).

Despite its importance, students often lack motivation to study the theory of computation and have problems understanding it. There are various explanations for this phenomenon, such as the complexity of the topic itself, or students’ view that theory is too “biased towards mathematics” [CGM04].
Nevertheless knowledge of the theory of computation is indispensable for computer science students. [Hro04] identifies five of the most significant reasons for the importance of the theory of computation in computer science education, summarized as:

**Philosophical depth:** Theoretical computer science explores knowledge and develops new concepts and notions that influence science in its very core. It gives partial or complete answers to philosophical questions such as “Are there problems that are not automatically solvable by algorithms?”.

**Applicability and spectacular results:** The theory of computation gives students insights that influence the handling of algorithmic problems. It provides directly applicable concepts and methods and helps solving problems that might otherwise have been considered unsolvable.

**Lifespan of knowledge:** Computer science is one of the fastest moving sciences today. What was cutting edge yesterday might already be out of
date tomorrow. However the theory of computation has been around for decades. Fundamental theories and concepts have consistently survived all changes and will continue to do so in the future.

Interdisciplinary orientation: The theory of computation can contribute to a large variety of interesting research areas. Genome research, diagnostic investigation, optimization in economics and technical science, voice recognition and space exploration are only a few examples.

Way of thinking: Mathematicians attribute the special role mathematics play in education through development, enrichment and shaping the way of thinking. A similar argument can be made about theoretical computer science, it offers the possibility to learn to combine theoretical knowledge with practical experience and thus to develop a way of thinking that is powerful enough to attack complex real-world problems.

The theory of computation has many fundamental concepts; we will focus our attention on complexity theory and time complexity in particular.

3.2 Complexity theory: NP-completeness and polynomial time reductions

Indeed, NP-complete problems are now so pervasive that it is important for anyone concerned with the computational aspects of these fields to be familiar with the meaning and implications of this concept. [GJ79]

Complexity theory deals with the resources required during computation. The most common resources are time (how long does it take to solve a problem) and space (how much memory does it take to solve a problem). Problems that are intrinsically computable can turn out to be practically insolvable because they require too many resources.

In this thesis we have limited our focus to time complexity, because time is the one resource typically dealt with in introductory courses on the theory of computation. Time complexity classifies algorithmic problems according to the time needed for the computation, depending on the size of the input. The two most important complexity classes are:

P: The class of problems that can be solved in polynomial time by a deterministic machine.

NP: The class of problems that can be solved in polynomial time by a non-deterministic machine.
NP-Completeness

In the early 1970s Stephen Cook [Coo71] and Leonid Levin [Lev73] defined the concept of \textit{NP-complete} problems. These problems are part of the class NP and have an individual complexity that is related to that of the entire class. If a deterministic polynomial time algorithm for one such problem can be found, all problems in NP will be solvable in polynomial time. This would also yield that $P = NP$, answering one of the most important open questions in the theory of computation today.

To prove that a problem is NP-complete polynomial time reductions are typically used.

Polynomial time reductions

A reduction transforms a given problem into an equivalent one, which is for instance easier but equivalent to solve. In the case of a “polynomial time reduction” the time needed to transform the original problem is polynomial in the size of the input.

Figure 3.2 shows the process of a polynomial time reduction. A problem $A$, known to be NP-complete, and a problem $B \in NP$ are given. To show that $B$ is NP-complete a polynomial time reduction from $A$ can be used. The reduction takes a given input $I$ for problem $A$ and in polynomial time transforms it to an input $I^*$ for $B$ using a coding function $c$. The output $O^*$ of $B$ is then transformed into an output $O$ of problem $A$ using a decoding function $d$, also in polynomial time. If such a polynomial time reduction from $A$ to $B$ can be found, $B$ is proven to be NP-complete.

![Fig. 3.2: Polynomial time reduction from problem $A$ to problem $B$](image)

Polynomial time reductions are fundamental to complexity theory and to NP-completeness in particular. Therefore they are also an important part of most introductory courses on the theory of computation.
3.3 A need for computer-based learning environments

Teaching NP-completeness and polynomial time reductions is a challenging task. Many students struggle with the topic and are never able to understand the fundamental concepts behind it and its importance. Besides the previously mentioned reasons for reservations towards the theory of computation, the high level of abstraction must be recognized as an additional hindering factor. To understand a reduction, students must deal with two different abstractions at once: the problems used in the reduction and the reduction process itself.

This barrier can be diminished if students are first given the opportunity to fully understand the problems the reductions deal with. Because of the abstract nature of most of the typical NP-complete problems, teaching such a full understanding in reasonable time might not be feasible using traditional teaching approaches such as lectures or books. Providing students with a learning environment that lets them experiment with the different NP-complete problems may promote positive learning results. If students are able to get a “hands-on” feeling and are able to discover the problems they will gain a more intuitive understanding. This will also help them understand that a problem is computationally hard to solve and that it belongs to the class NP.

This is only the first step, because understanding a polynomial time reduction is still a very demanding task. Presented in rigorous detail, reductions can overwhelm students without allowing them to understand the underlying fundamental concepts. With books and lectures students are bound to the level of detail presented by the authors and teachers. Educational software on the other hand can allow students to alter the level of detail between global general concepts and exact details. A more intuitive understanding of the underlying concepts can for instance be promoted by allowing students to ‘play’ with the reductions. Additionally students will find learning the theory to be more fun [RGKL02].

In 1998 and 1999 Boroni et al. identified the need for educational software to cover the theory of computation, including NP-completeness [BGGR98], [BGG+99]. They outlined a possible PhD thesis as part of the ‘Webworks’ project at the CS department of Montana State University [BGGR]. They stated that “The goal is to provide interactive animations of the fundamental concepts of the theory, including problem reductions and NP-completeness”. While their site offers elaborate applets for finite automata or context free grammars learning environments covering NP-completeness and polynomial time reductions are still missing today. The presented thesis addresses their challenge and realizes their PhD idea.
The clearly identified need on one hand, and the lack of learning environments for polynomial time reductions and NP-completeness on the other hand, motivated us to design and develop GraphBench, a learning environment for complexity theory.

3.4 GraphBench: Choosing the right content

[GJ79] contains a list of over 300 NP-complete or NP-hard problems. For GraphBench we selected a subset. In addition to the Satisfiability problem Garey and Johnson identify six of the most frequent used problems (figure 3.3). They suggest that these six problems should serve as the “basic core” of known NP-complete problems for beginners.

Fig. 3.3: Diagram of the six basic NP-complete problems and reductions

Besides the criteria introduced by Garey and Johnson, we also took intuitive graphical visualization into account for selecting the NP-complete problems and reductions to be part of GraphBench. Figure 3.4 shows the 9 problems and 12 reductions currently featured in GraphBench.
Fig. 3.4: The content of GraphBench. The boxes denote NP-complete problems and the arrows polynomial time reductions.
Chapter 4

A first session with GraphBench

In this chapter we give a first look at our learning environment GraphBench on the basis of a virtual session. We schematically show how GraphBench can be used to study. We show the two tutors for SATISFIABILITY and for the reduction 3CNF-SATISFIABILITY to GRAPH COLORABILITY. Both are similar in their use and user interface to the other tutors for NP-complete problems and polynomial time reductions available in GraphBench.
4.1 Satisfiability

Choosing the problem tutor

After starting GraphBench we see an overview of all NP-complete problems and polynomial time reductions available. Moving the mouse over the SATISFIABILITY box displays a short description of the problem. Clicking the box starts the tutor.

Fig. 4.1: The overview page of GraphBench
The empty tutor

The tutor opens with an empty workspace, i.e. without a Boolean formula set. The main part of the workspace holds the four different problem views: a formal textual view of the formula, graphical views of the variables and clauses and an additional view displaying the formula as a “Boolean circuit”. Additionally the workspace contains the toolbar, a panel displaying problem information, and a panel for controlling solution algorithms. The toolbar allows us to return to the overview, open and save Boolean formulas, manipulate a formula, and display help about the problem and the tutor itself.

Fig. 4.2: The empty Satisfiability tutor
Creating a random problem instance

We first create a random problem instance, i.e. a random Boolean formula, by clicking the “New random formula” button. This opens a dialog that asks us to specify the parameters of the Boolean formula to be created. We can specify the number of variables and clauses, as well as the number of literals within a clause.

Fig. 4.3: Specifying parameters for a random formula
Investigating the problem instance

After GraphBench has created a random Boolean formula, according to the parameters specified, we can investigate the formula before trying to solve it. GraphBench not only allows us to use Boolean values \((true, false)\), but also continuous ones in the interval \([0,1]\). We can select any variable with the mouse to highlight the variable and all “its” clauses within all views or we can select any clause to highlight its variables.

Fig. 4.4: Inspecting the problem instance by selecting a variable
Solving a problem instance by “hand”

To solve the problem, i.e. find a satisfying variable assignment, we change the values of the variables by moving their “value-sliders”. The tutor instantly adapts the values of the clauses.

Figure 4.5 shows a possible solution process. We first set the value of variable $x_1$ to true, which satisfies clause $C_4$. Setting the value of variable $x_2$ to true will additionally satisfy clauses $C_2$ and $C_3$, while setting the value of clause $C_0$ to false. In our solution process we change the values of the variables until we reach a satisfying variable assignment (figure 4.6).

![Fig. 4.5: A possible solution process (from left to right and top to bottom)](image-url)
Fig. 4.6: Reaching a satisfying variable assignment
Executing a solution algorithm

Besides solving a problem instance by hand we can also use the solution algorithms provided by GraphBench to find a satisfying variable assignment. From the list of possible algorithms we choose Limited Local Search (figure 4.7).

![GraphBench](image)

Fig. 4.7: Selecting a solution algorithm

Before executing the algorithm we can first read a brief textual description of the algorithm and study its pseudo-code (figure 4.8). We then start to execute the algorithm step by step. During the execution the tutor highlights the current line of code, displays statistics and highlights the data currently being worked on within the different views (figure 4.9). After the first steps we change to “movie-mode” and watch the algorithm finish.
Fig. 4.8: Brief textual description and pseudo-code of the Limited Local Search algorithm

Fig. 4.9: The tutor during the execution of the Limited Local Search algorithm
Creating further problem instances

After examining and solving a first problem instance we now want to create different problem instances. We can first create additional random examples by using the same parameters. We can also add or remove single variables or clauses.

We then create Boolean formulas using the built-in formula editor that match our interest. We modify an existing Boolean formula by adding, removing and renaming variables (figure 4.10) or create completely new formulas. Additionally we can choose examples from a list of predefined problem instances, which cover for instance worst case scenarios (figure 4.11).

![Fig. 4.10: Editing an existing Boolean formula](image1)

![Fig. 4.11: Choosing from a list of predefined examples](image2)
4.2 Reducing 3CNF to Graph Colorability

The reduction tutor

We have started the tutor for the 3CNF-Satisfiability to Graph Colorability reduction and have already created a first Boolean formula. The formula and the variables are displayed in the top of the main view. The bottom of the main view hosts the graphical representation of the reduced Graph Colorability instance. This view is still empty because we have not yet executed the reduction algorithm.

Fig. 4.12: The tutor for the the 3CNF SATISFIABILITY to GRAPH COLORABILITY reduction before the reduction algorithm has been executed.
Executing a polynomial time reduction

We have opened the pseudo-code view and click the “step”-button to execute the reduction algorithm step by step (figure 4.13).

Fig. 4.13: The reduction tutor after executing the first line of the reduction algorithm.

The instance of the Graph Colorability problem is generated incrementally by the reduction algorithm. The tutor highlights the elements of both problem instances that are currently being worked on. Figure 4.14 shows the tutor in different states during the execution of the reduction algorithm. The bottom right image shows the tutor after the algorithm has finished.
Fig. 4.14: States of tutor during the execution of the reduction algorithm (from left to right and top to bottom).
Investigating the corresponding problems

After the reduction algorithm has finished we start to investigate the two corresponding problem instances. We try to understand how the reduced problem instance has been created and what the dependencies among the elements of the two instances are. We select elements of either problem instance to see the elements of the other problem instance that are dependent.

Fig. 4.15: Investigating the dependencies among the elements of the two problem instances.
Observing effects of changing the original problem

We now want to see how changes in the original problem instance affect the reduced instance. We first select the “auto update” option. This causes the tutor to automatically execute the reduction algorithm whenever the Boolean formula is changed. We add a single clause to the Boolean formula (figure 4.16). This results in a new vertex C4 and several new edges within the graph. We proceed by adding and removing different clauses and variables.

Fig. 4.16: Observing the effects of adding a single clause.
Solving the reduced problem instance

At the end we want to solve the reduced problem instance, i.e. find a valid coloring. Figure 4.17 shows the tutor during the solution process. Figure 4.18 shows the tutor after we have successfully colored the graph. The solution has been automatically transformed to the corresponding solution of the Satisfiability problem instance.

![Graphical editor](image-url)

Fig. 4.17: Solving the reduced problem instance by hand.
Fig. 4.18: A valid coloring and the corresponding solution of the Satisfiability problem instance.
Chapter 5

Didactic concepts in GraphBench

Designing an educational learning environment is a challenging task because a large variety of requirements have to be fulfilled. A high level of interactivity or the functionality of a subject matter tutor, i.e. allowing students to design and solve individualized exercises, are just two of them. In this chapter we introduce the didactic concepts and implementation approaches we have used for building GraphBench. We illustrate all concepts with examples and state their learning benefits.

5.1 Example generation

Examples are an important part in teaching, regardless of the medium used. A good and effective example can make an otherwise complex subject more accessible for students. Books and lectures limit students to the examples selected by the author or teacher. With computer-based learning environments this is no longer the case. Educational software can offer a large library of examples and it can allow students to create examples of their own or at random.

Random example generation

All tutors in GraphBench are capable to generate random problem instances. They allow students to specify parameters that describe characteristics of the examples to be created. To make it easy for students to generate random instances, the number of parameters is kept small and students can also use default settings. The number of parameters and their type depend on the problems. Figure 5.1 shows two dialogs, one for Boolean formulas and one for graphs, that allow students to specify parameters. For Boolean formulas
students can specify the number of variables and clauses in the formula. They can also specify the number of literals within a clause, this to create formulas in $k$-CNF. For graphs students can specify the type of graph to be created (i.e. random graph, complete graph or tree), the number of vertices and the number of edges. Additionally they can require for a graph to be connected.

Fig. 5.1: Setting parameters to create random problem instances for Boolean formulas (left) and graphs (right)

**Example generation by students**

Besides random generation, tutors in GraphBench also allow students to manually create problem instances and modify existing ones arbitrarily. Depending on the tutor, this can either be done directly within the graphical representation of the problem or by using a specific editor. Generating or editing graphs for instance can be done directly within the view of the graph. Students can add vertices by double clicking the left mouse button within the view or they can delete existing vertices using the right mouse button. The Satisfiability tutor on the other hand uses a specific problem instance editor to create and modify Boolean formulas (figure 5.2).

Fig. 5.2: The editor to enter and modify Boolean formulas
Offering students the possibility to generate and modify problem instances freely, as well as being able to create problem instances at random brings several benefits:

- Students are not limited to a predefined set of examples.
- Students can examine as many examples as they want, matching their personal preferences and difficulties.
- Students can experiment, discover and come up with special or worst case examples.
- It increases the level of interactivity according to Schulmeister.

There are also drawbacks when allowing students to create examples of their own. [Rie02] points out that students are prone to the confirmation bias and tend to design experiments that support their hypotheses. GraphBench therefore contains a library of predefined examples. In particular worst-case scenarios that students might otherwise miss.

5.2 Solving problem instances

All tutors in GraphBench allow students to solve examples by hand, but also offer automatic solving. An important point is that arbitrary examples can be solved and not only a predefined set of problem instances.

Solutions by students

All NP-complete problem tutors in GraphBench allow students to solve examples using simple and intuitive graphical commands. For most problems in GraphBench solving them means to create a subset of elements that fulfill problem-specific constraints. Solving such a problem corresponds to the process of incrementing the solution element by element. During this problem-solving process, i.e. before the student reaches a correct solution or aborts, the partial solution can either fulfill or violate the problem constraints.

The tutors in GraphBench support students by identifying the state of their solution process and by providing explaining information if it is incorrect, i.e. violating constraints. In case of a partial solution that does not violate any constraints, GraphBench does not alert students if it cannot be extended to a correct one. There are two reasons:

1. Testing whether a partial solution can be extended to a correct one is a computationally difficult and possibly very time consuming task.
2. Even assuming the computer can in reasonable time determine whether a complete solution can be reached or not, displaying and explaining the reasons for the answer would most likely be infeasible.

Figure 5.3 shows three solutions to an instance of the Independent Set problem that admits an independent set of 4 vertices. The different states of the solutions are depicted using the colors orange, green and red for partial, complete and incorrect solutions, respectively. In case of the incorrect solution (figure 5.3 right) the reason for the incorrectness is also shown in red (i.e. the edge connecting vertices 2 and 6).

![Figure 5.3: Three states of solutions for the Independent Set problem. A partial solution not violating any constraints, a complete solution and an incorrect solution (v.l.t.r). The incorrect solution violates constraints because the two vertices 2 and 6 are connected by an edge.](image)

Allowing students to solve examples by hand helps them gain a better understanding of the problems and their characteristics. The learning environment should ensure that students do not get frustrated, because they cannot find a correct solution within reasonable time. Providing them with feedback and guidance will help avoid such frustrations. Successfully solving a problem is also very motivation for students and enhances learning results [FFE00].

**Solutions generated by GraphBench**

Besides allowing solutions generated by students, all tutors in GraphBench are also capable of solving arbitrary problem instances automatically. NP-complete problems can be solved either by exhaustive search to find an optimal solution or by heuristics, which do not always find an optimal solution. In GraphBench we give students the opportunity to work with both approaches.
For all problems we offer at least one exhaustive search algorithm and often additional heuristics. For the Traveling Salesman problem *Branch and Bound* and *Backtracking* can be used to solve the problem exhaustively. In addition four heuristics are available: *Nearest neighbor, Greedy, Two-opt local search* and *Convex hull*.

For both exhaustive search algorithms and heuristics the solution process is made as comprehensible as possible. Any solution presented by a tutor comes with an explanation of the process that created it. Two mechanism are used in GraphBench to support this goal: pseudo-code and algorithm animation.

### 5.3 Pseudo-code

All algorithms part of GraphBench come with a pseudo-code representation to impart knowledge of how the algorithm works. The pseudo-code is written in a high-level language rather than an actual particular programming language. [SSMN04] points out that pseudo-code should be easy to understand and that a high-level language is desirable. Additionally GraphBench uses syntax-highlighting and semantical variable names to further ease understanding of the pseudo-code. The following listing shows the pseudo-code for the Traveling Salesman *Branch and Bound* algorithm:

```plaintext
openPaths = createSortedList()
currentPath = createPath(first vertex)
while notValid(currentPath) do
    foreach vertex not in currentPath do
        newPath = createPath(currentPath, vertex)
        if containsAllVertices(newPath) then
            closePath(newPath)
        end if
        insert(openPaths, newPath)
    end foreach
    currentPath = removeShortestPath(openPaths)
end while
```

The example shows how we focus on the logical steps of the algorithm that directly deal with solving the problem and hide details by using methods with semantic names.

**Loop variants and invariants**

In addition to the lines of code that are being executed, the pseudo-code also allows the user to view loop variants and invariants (figure 5.4). The invariant
of a loop is an assertion that must hold before every iteration of the loop’s body. The variant of a loop is an integer-valued expression that is decreased with every iteration of the loop’s body and never takes on a negative value [Mey92].

% The algorithm works on graph G = (V,E), with n = |V|
currentPath = createPath(first vertex)
while sizeOf(currentPath) < n do
  % INVARIANT: 1 <= sizeOf(currentPath) <= n
  % VARIANT: n - sizeOf(currentPath)
  nextVertex = findClosestVertex()
  append(currentPath, nextVertex)
end while

Fig. 5.4: The Nearest Neighbor algorithm for the Traveling Salesman problem, including loop variant and invariant.

Including loop variants and invariant within the pseudo-code brings two main advantages:

**Loop correctness:** variants and invariants help students check the correctness of a loop. I.e. the invariant determines the properties of a loop on exit and the variant serves to guarantee that the body of the loop’s execution terminates.

**Clarity:** variant and invariants clarify the purpose of a loop and help students understand what the loop is doing.

The pseudo-code, including the loop variants and invariants, is not just a static text, but is displayed dynamically during algorithm animations.

## 5.4 Algorithm animation

For students to understand an algorithm, presenting a static, textual description and the result of its execution might not be sufficient. GraphBench therefore also offers animated visualizations of the algorithms during execution. [Low04] points out that students might be disadvantaged if their comprehension process cannot keep up with the pace of the algorithm visualization. In order to avoid such shortcomings GraphBench gives students control over the execution speed. Additionally students can execute algorithms step-by-step and set arbitrary breakpoints. This significantly helps improve learning outcomes [SSMN04].

During the animation different aspects of the algorithm are displayed:
Figure 5.5 shows GraphBench during the visualization of the *Backtracking* algorithm for the Graph Colorability problem. During the execution of the algorithm a vertex can be in one of three states: *colored*, *currently processed* or *open*. These vertex states are visualized within the view of the graph by highlighting or coloring the vertices. Additionally the vertex states are also displayed within a view of the internal data structure of the algorithm.

Research has shown differences in students cognitive thinking strategies. [Sch04] identifies functional and predicative problem solving skills and argues that new learning technologies have to address both in order not to disad-
vantage anyone. Offering different views of the algorithm gives students the possibility to work with the one that suits them most. Some students might prefer the visualization of the pseudo-code, while others favor the internal data structure.

In order to get more information about an algorithm, and its running time in particular, students can use simple statistics. The number of times a single line has been executed can be displayed within the pseudo-code view (figure 5.6). This kind of runtime analysis is especially useful to show students the exponential nature of exhaustive search algorithms.

Fig. 5.6: The pseudo-code view during algorithm animation. The line currently executed is highlighted and statistical information is displayed.
5.5 Multiple problem representations

Personal preferences and ways of thinking play a big role in students’ understanding of problem visualizations. While some students might favor a formal presentation of the problem, others might appreciate a more intuitive, graphical depiction. This holds especially for topics that lack a standard, ‘off-the-shelf’ visualization. To achieve better learning results and satisfy students’ needs, it is necessary to offer different representations.

In GraphBench two problems in particular demand multiple representations: SATISFIABILITY and 3-DIMENSIONAL MATCHING. Both problems are typically presented in a formal, textual manner, without a more intuitive, graphical visualization.

The 3-Dimensional Matching problem deals with the formal mathematical objects sets, triples and elements. The problem is often depicted by simply showing a textual representation of the sets and triples. This might make it difficult for students to see the effects of choosing a particular triple to be part of the solution. GraphBench therefore offers an additional, more intuitive representation of the problem: the three-dimensional visualization (figure 5.7).

An important point to note is that the different problem views are displayed in relation to each other. They are dynamically connected in order to show the dependencies among them. Selecting a component in one view highlights the same component in all other views. In figure 5.7 the user has selected a point in the three-dimensional view of the problem (pointed at by the mouse cursor).
The meaning of this point is shown by highlighting the triple it represents within the list of triples.

Providing students with multiple problem representations and showing the relations and dependencies among them helps students comprehend the problem and gain a more intuitive understanding. Emphasizing relations and dependencies is not only valuable for different problem views, but also for a problem and its solution.

5.6 Relations among structures for reductions

Figure 5.8 shows the reduction from Satisfiability to Graph Colorability. The top frame shows the Boolean formula of the Satisfiability problem and the bottom frame the corresponding instance of the Graph Colorability problem.

Every reduction in GraphBench comes with an algorithm that generates corresponding instances. That is, it takes the original problem instance (e.g. a Boolean formula of the Satisfiability problem) and generates an instance of the problem the reduction reduces to (e.g. a graph for the Colorability problem). Reduction algorithms are treated as ‘normal’ algorithms within GraphBench, in the sense that they can be animated and executed step by step if desired.
To help students understand a reduction and why it is correct, it might not suffice simply to provide the two corresponding problem instances. Even animating the generating algorithm might not be enough and more information could be necessary. In GraphBench we therefore allow students to investigate the dependencies of two corresponding problem instances and thus gather further information about the reduction. This can be done by graphically selecting components of one problem instance to receive information about the dependencies to components in the other problem instance.

When selecting a vertex in the example shown in figure 5.8, the reduction tutor highlights the components of the Satisfiability problem responsible for or dependent on the vertex. This helps students understand why the vertex has been created and why it is relevant to the correctness of the reduction. Students can also select any component of the Satisfiability problem to see how they have been involved in the creation of the graph. This kind of investigation allows students to take a closer look at separate parts of the reduction and thus gain a better understanding overall.

5.7 Selective level of detail

Small, simple examples are an effective help in understanding a topic. But what if even the smallest examples are complex and difficult to grasp? Good examples are too important to be left out and must be presented in a manner that fits students capacities. If examples and their solutions are complicated, the learning environment should support students in dealing with this complexity. It should function as a guide, focusing the student’s attention to the important parts. Unnecessary information or information not of immediate importance should be put out of focus and only displayed on demand.

Separating important information from unnecessary information not of immediate importance for students is a challenging task. For GraphBench this is done by the teacher or programmer who design the individual subject matter tutors. A different approach would be to have the learning environment itself identify important information, e.g. by using artificial intelligence.

Figure 5.9 shows the reduction from an instance of the Vertex Cover problem to the Hamilton Circuit problem. The original problem instance consists of nothing more than a triangle with three vertices and three edges. The resulting graph consists of over 30 vertices and almost 40 edges. The whole graph (figure 5.9 right) is impractical to survey.

Instead of burdening students with such a graph, the tutor hides vertices that are not of immediate importance and lets the student focus on the important ones (figure 5.10). An important point to note is that even if certain
vertices are hidden, they are not kept secret. GraphBench gives students the possibility to view the complete graph if desired.

Fig. 5.9: Reduction from VERTEX COVER to HAMILTON CIRCUIT.

Fig. 5.10: Using “selective level of detail” to hide unnecessary information.

An expert user knows what is important to focus on. Novices do not have this expert knowledge and can easily get lost. The concept of “selective level of detail” is a powerful way of helping students to cope with large-scale problem instances.
5.8 Intuitive graphical representations

One important requirement of GraphBench were intuitive graphical representations of the problems. Depending on the problem type, finding such intuitive visualizations is a demanding task that requires several iterations. We use 3-DIMENSIONAL MATCHING as an example to illustrate how didactic concepts guided the design of a problem tutor. We describe the 3-DIMENSIONAL MATCHING tutor in detail in 6.5.

For the 3-Dimensional Matching problem the challenge was to find a representation of the abstract sets and triples that is intuitively understandable. An additional requirement was to help students understand the meaning of the solution and the impact of a single triple. In order to find a suitable visualization several designs were implemented and tested. In the context of this thesis a systematic evaluation of GraphBench has not been performed (see also chapter 10. The following assessments are, instead, based on informal feedback from test users and students.

Figure 5.11 shows the textual visualization of the abstract sets and triples as part of the current version of GraphBench. In the left image the user has selected the one triple that has been added to the solution so far, while a complete solution has been found in the right image.

Besides this “textual” visualization we wanted to provide students with a more tangible one. Our first approach was to use a graph to represent the sets, triples and the solution (figure 5.12). We used three groups of vertices to represent the elements of the sets. A triple belonging to the solution was visualized by connecting the three respective vertices with edges to form a triangle.
We realized that this representation was not satisfactory because it did not provide more insights into the problem. It also did not allow support displaying all possible triples, because this would have led to an accumulation of triangles that would have been too complex. Our second approach was to display the triples in form of entries in a matrix (figure 5.13).

The $x$ and $y$ coordinates of a cell corresponded to the first two entries of the triple. The third element of the triple was then displayed as an entry in the matrix. The triple $(3,2,1)$ was for example shown with an entry ‘1’ in the cell $[3,1]$.

The 2-dimensional representation made it possible to display all possible triples and highlight the ones belonging to the solution. The impact of adding a triple to or removing it from the solution could be shown by shading all affected entries gray (figure 5.13 left). Usability tests showed, however, that
the visualization was not intuitive, despite the desirable characteristics. Test users had problems to make the connection between an entry in the matrix and the corresponding triple. The feedback from test users led to the graphical visualization part of GraphBench now, a 3-dimensional representation (figure 5.14). The three sets are represented by the three dimensions and triples as points in space. Making the connection between a triple and its point in space became easier for students and the visualization nicely shows the impact of a single triple on the solution.

![Fig. 5.14: The final 3-dimensional representation. Left: a single triple has been added to the solution. Right: a complete solution.](image)

5.9 Conclusions

The didactic concepts we have introduced help approach abstract topics. Students are actively involved with the subject matter by allowing them to generate examples and solve them by themselves. GraphBench offers mechanisms to automatically generate random examples and is able to solve arbitrary problem instances. We have shown how a learning system can support students in solving problems, how it can be responsive to students personal interests and questions and how it can help them focus on important aspects.
Chapter 6

NP-Complete problems in GraphBench

GraphBench features nine different NP-complete problem tutors. The problems were selected based on their frequent use in computer science education and their use in polynomial time reductions. An additional criterion was the existence of intuitive graphical representations. In this chapter we list the NP-complete problems part of GraphBench and show the resulting subject matter tutors.

6.1 Satsifiability

Satisfiability was the first problem identified to be NP-complete. In 1971 Steven Cook proved that any Turing Machine, including its input, can be transformed into a Boolean formula such that the formula is satisfiable if and only if the Turing Machine accepts the input [Coo71].

In GraphBench we are only dealing with Boolean formulas in conjunctive normal form (CNF), i.e. with formulas expressed as an AND of clauses, each of which is the OR of one or more literals. GraphBench contains two Satisfiability tutors, one for arbitrary CNF formulas and one for formulas in 3CNF. This does not in any way change the fundamental impact, the role in polynomial time reductions or the complexity of the Satisfiability problem. The problem is defined as follows:

| INSTANCE: A set $U$ of variables and a collection $C$ of clauses over $U$ |
| QUESTION: Is there a satisfying truth assignment for $C$? |
Continuous variable values

GraphBench does not only use Boolean values (true and false), but also allows continuous values in the interval [0,1]. In the beginning we expected that the use of continuous values would allow us to create more intuitive visual representations and solution algorithms. It turned out that the benefits gained in respect to these two criteria where not as high as expected. However, from a users’ perspective continuous values have three advantages nevertheless:

1. Variables that do not contribute to a satisfied solution can be identified easily by having a value other than 0 or 1.

2. For reductions from Satisfiability the values of the variables can be set to 0.5 until the reduced problem instance has been solved. This shows the user that the variable values are derived from the solution of the reduced problem and are undefined until a solution has been found.

3. Changes in variable and clause values can be made visible continuously and are easier for the user to spot than sudden, discrete value jumps.

Calculation of continuous values

In the case of continuity, variables can take on any value between 0 and 1. For a variable \( v \) with value \( val(v) \in [0,1] \) its negation \( \neg v \) has the value \( val(\neg v) = 1 - val(v) \) assigned.

The Boolean operators AND and OR are treated as MIN, respectively MAX functions when using continuous values. Therefore the value of a formula in CNF corresponds to the minimum value of its clauses, with the clauses taking on the maximum value of their literals.

\[
val_F(F) = MIN(val_c(c_0), val_c(c_1), \ldots, val_c(c_m)) \quad F = c_0 \land c_1 \land \ldots \land c_m
\]

\[
val_c(c) = MAX(val(l_{i0}), val(l_{i1}), \ldots, val(l_{ik})) \quad c_i = l_{i0} \lor l_{i1} \lor \ldots l_{ik}
\]

An example of a formula \( F \) and its evaluation would thus look as follows:

\[
F = (x_1 \lor x_2 \lor \neg x_3) \land (x_2 \lor x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4)
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
\hline
\text{MAX}(0.5,1,0.2) & 1 & 0.2 & 1 & 0 & 0.5 & 1 & 0.8 \\
\hline
\text{MIN}(1,1,1) & 1 & 1 & 1 & \text{MAX}(0.5,1,0.8) & 1 & \text{MAX}(0.5,1,0.8) & 1 \\
\hline
\end{array}
\]

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>1</td>
<td>0</td>
<td>0.8</td>
</tr>
</tbody>
</table>

The given variable assignment satisfies the formula, i.e. the value of the formula is 1 (\( val_F(F) = 1 \)). The variables \( x_1 \) and \( x_3 \) do not contribute to satisfying the formula. They have values other than 0 or 1 and could take on any value without affecting the value of the formula.
Visualization

GraphBench offers different views for the Satisfiability problem (6.1). The complete formula is displayed in a formal, textual representation. This representation displays the current values of the literals and clauses dynamically. True and false values are emphasized using the colors green and red, while “don’t care” values are shown in black. This coloring is consistently applied to all views, green always denotes a true and red always a false value. A second and third view show the values of the variables, respectively the values of the clauses. For both, variables and clauses, a single value is displayed as a ‘slider’ ranging from 0 to 1. Students can change the value of a variable simply dragging its slider with the mouse. Finally a fourth view shows a “Boolean circuit” of the formula.

The different views are synchronized to show their relationships and dependencies. When the user selects any component (e.g. a variable) of one view, the same component is highlighted in all other views. Figure 6.1 shows the four views with the clause C4 selected by the user within the clause view. The same clause is emphasized within the formula view and all the variables occurring in the clause are highlighted in the variable view. Finally the parts of the Boolean circuit that are connected to the clause, i.e. its variables and the clause itself, are highlighted.

Fig. 6.1: Connecting the different views for the Satisfiability problem
Solving Satisfiability in GraphBench

GraphBench provides three algorithms to solve Boolean expressions in CNF: Backtracking, Limited Local Search Heuristic and Physical Model Heuristic. The Backtracking algorithm uses exhaustive search to test all possible variable assignments, stopping only if a satisfying one is found. It is a straightforward solution and we will not present it in detail.

Limited Local Search Heuristic

The Limited Local Search Heuristic was presented by U. Schöning [Sch02a] in 2002. In its original version the algorithm works with Boolean formulas in k-CNF, i.e. every clause contains exactly k literals. For GraphBench we disregard this requirement and allow the algorithm to run on formulas with clauses of different size. For the calculation of the maximal number of tries we set k to be the maximum size of all clauses.

Physical Model Heuristic

The Physical Model Heuristic is based on the idea of physical forces. Every clause $C_i = (l_1 \lor l_2 \lor \ldots \lor l_m)$ tries to become satisfied by exerting forces on its literals. The forces ‘push’ the value of the variables towards 1 if the variable occurs as a positive literal and towards 0 if it occurs as a negative literal. The strength of the force varies for every variable and depends on the amount of work needed to reach the desired variable value. If the value of the variable is close to the value desired by the clause a bigger force is exerted than if the value would be further away (figure 6.2).

Fig. 6.2: Forces exerted by the clauses during a single step of the physical model heuristic. $F = (\neg x_2 \lor x_4 \lor \neg x_1) \land (\neg x_1 \lor x_2 \lor x_3) \land (x_2 \lor x_3 \lor \neg x_4)$
In every step the heuristic calculates the forces the clauses exert on the variables. These forces are summed up for every single variable and the variable value is then adjusted according to the combined forces. The algorithm terminates as soon as a satisfying variable assignment is found.

6.2 Vertex Cover, Clique and Independent Set

The similarity of the three problems Vertex Cover, Clique and Independent Set resulted in three similar tutors. We thus introduce them together. All three problems deal with a graph \( G = (V, E) \) and a positive integer \( K \leq |V| \) and they all have the goal of finding a subset of vertices that fulfill specific requirements. These requirements are defined as follows:

**Vertex Cover:** Does \( G \) contain a *vertex cover* of size \( K \), that is, a subset \( V' \subseteq V \) such that \( |V'| \leq K \) and, for each edge \((u, v) \in E\), at least one of \( u \) and \( v \) belongs to \( V' \)?

**Clique:** Does \( G \) contain an *clique* of size \( K \), that is, a subset \( V' \subseteq V \) such that \( |V'| \geq K \) and every two vertices in \( V' \) are joined by an edge in \( E \)?

**Independent Set:** Does \( G \) contain an *independent set* of size \( K \), that is, a subset \( V' \subseteq V \) such that \( |V'| \geq K \) and no two vertices in \( V' \) are joined by an edge in \( E \)?

Graph problems have the advantage that their graphical representation is standardized and well known by students. The challenging task for a learning environment is to visualize additional information within the graph, for instance about the solution. Figure 6.3 shows the Clique tutor and in particular how students can modify the solution using context menus and how GraphBench provides supporting information. Vertices belonging to the solution are displayed in green, yellow or red, depending on the state of the solution: green if the solution is correct, yellow if it is correct but not of the required size and red if there is a constraint violation.
Both problems, Hamiltonian Circuit and Traveling Salesman, have the goal of finding a closed path, an ordering of all vertices, within their graph. Traveling Salesman is probably one of the best known NP-complete problems. The fact that it describes a "real world" situation makes it suitable for a non-expert audience. Hamiltonian Circuit on the other hand is more abstract:

**INSTANCE:** A graph $G = (V, E)$  
**QUESTION:** Does $G$ contain a Hamiltonian circuit, that is, an ordering $< v_1, v_2, ..., v_n >$ of the vertices of $G$, where $n = | V |$, such that $v_n, v_1 \in E$ and $v_i, v_{i+1} \in E$ for all $1 \leq i < n$?

Figure 6.4 shows the visualization for Hamiltonian Circuit during a student’s solution process. GraphBench offers support by visualizing the state of
the vertices. These states depend on the number of edges incident to the vertex and are defined as follows: unvisited (0 incident edges), visited (1), valid (2) or invalid (>2).

Fig. 6.4: The HAMILTONIAN CIRCUIT tutor helps students to find a solution by displaying the state of a vertex using different colors.

TRAVELING SALESMAN is a variant of HAMILTONIAN CIRCUIT in the form of an optimization problem. It is defined as follows:

INSTANCE: Set $C$ of $m$ cities, distance $d(c_i, c_j) \in \mathbb{Z}^+$ for each pair of cities $c_i, c_j \in C$

QUESTION: Which is the shortest tour $C$, that is, a permutation $<c_{\pi(1)}, c_{\pi(2)}, ..., c_{\pi(m)}>$ of $C$ such that $\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(m)}, c_{\pi(1)})$ is minimal?

The tutor for TRAVELING SALESMAN addresses two versions of the problem, one for weighted graphs and one for Euclidean graphs in the plane (figure 6.5). Because both versions deal with optimization problems, their visualizations differ from the other graph problems in GraphBench. Instead of displaying only the partial or complete solution currently being computed it also shows
the best solution found so far. The state of the vertices are colored similar to the \textsc{Hamilton Circuit} tutor.

![Fig. 6.5: The \textsc{Traveling Salesman} tutor displays the best solution found so far and the current solution at the same time.]

\section{6.4 \textsc{Graph Colorability}}

\textsc{Graph Colorability} is well suited for a learning environment about complexity theory. Its problem description is relatively intuitive and its solution does not ask for a subset of elements that have to fulfill abstract constraints:

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{INSTANCE:} A graph \( G = (V, E) \) and a positive integer \( k \leq |V| \) \\
\textbf{QUESTION:} Is \( G \) \( k \)-colorable, that is, does there exist a function \( f : V \to 1, 2, \ldots, k \) such that \( f(u) \neq f(v) \) whenever \( u, v \in E \)? \\
\hline
\end{tabular}
\end{center}

The task of assigning a color to every vertex, such that neighboring vertices have different colors, is easily and quickly understood by students. Figure 6.6 shows the \textsc{Graph Colorability} tutor of GraphBench. It shows how students can solve problems by hand, by assigning a color to each vertex. The list of possible colors is shown using a context menu for every vertex.
GraphBench supports students by indicating which colors are good choices and which color would lead to a conflict. Additionally constraint violations, i.e. two vertices that have the same color and that are connected by an edge, are pointed out by showing the edge in red.

### 6.5 3-Dimensional Matching

3-DIMENSIONAL MATCHING is probably the most difficult NP-complete problem in GraphBench for students to understand. It is defined as follows:

\[
\text{INSTANCE: A set } M \subseteq W \times X \times Y, \text{ where } W, X \text{ and } Y \text{ are disjoint sets having the same number } q \text{ of elements}
\]

\[
\text{QUESTION: Does } M \text{ contain a matching, that is, a subset } M' \subseteq M \text{ such that } | M' | = q \text{ and no two elements of } M' \text{ agree in any coordinate?}
\]

One reason for its difficulty is the formal representation of the sets and triples typically used. This representation makes an intuitive approach difficult for students, especially when trying to understand the meaning of the solution.
With GraphBench we try to provide a graphical visualization, that is more intuitive and helps students understand the problem and its solution. Figure 6.7 shows the GraphBench tutor for 3-DIMENSIONAL MATCHING. Besides the formal representation of the sets, triples and the solution, it also contains a 3-dimensional view, that uses the sets as dimensions and displays the triples as points in space.

Fig. 6.7: Providing graphical information about the elements and the relations between the different views for 3-DIMENSIONAL MATCHING.

The tutor not only displays the data, but provides additional, graphical information about the elements. Coloring is used for instance to visualize the state of a triple. A triple that is part of the solution is assigned a unique color. A triple that agrees in at least one coordinate with a solution triple is shaded in gray, to indicate that it is ‘covered’ and cannot become part of the solution. A triple that does not agree in any coordinate with any solution triple is shown in black, indicating that it is ‘open’ and could be part of the solution. This coloring is used not only in the 3-dimensional view, but also in the formal, textual representation.

GraphBench provides additional information to help students understand the different representations and find a solution. When a student selects a triple with the mouse, in either the 3-dimensional or the triple view, the triple is highlighted in both views. Additionally the 3-dimensional view shows the
effects of the selected triple, i.e. it points out all the triples that agree in at least one coordinate and would thus be affected if the triple would be added to or removed from the solution.

Similar to all other tutor environments in GraphBench students can freely create and modify problem instances for 3-DIMENSIONAL MATCHING. An edit mode allows students to add or remove triples, both graphically and textually.
Chapter 7

Polynomial time reductions in GraphBench

GraphBench contains a total of twelve different polynomial time reductions. The most important selection criteria was their applicability in computer science education. Most of them are standard reductions that can be found in books about the theory of computation and complexity theory (e.g. [Sip97],[Sch97]). We introduce all reductions briefly to give an overview.

7.1 Reducing from Satisfiability

Satisfiability plays a major role in polynomial time reductions especially in computer science education. As the first known NP-complete problem it lies at the root of the ‘reduction tree’.

GraphBench offer tutors to reduce 3CNF-Satisfiability to Vertex Cover, Clique and Graph Colorability. The complexity of the three reductions is similar in that they are demanding to understand and result in complex graphs of medium size. All three tutors support students by displaying the graphs in a semantically meaningful manner. Vertices that originated from the same component of the formula, i.e. the same clause, are visually grouped. Figure 7.1 shows the tutor for the reduction 3CNF SATISFIABILITY to VERTEX COVER. The tutor has semantically labelled and grouped the vertices to help students see “where they came from”. Additionally all tutors emphasize the relations between the components of two corresponding problem instances. Selecting any component (i.e. an edge, vertex, variable or clause) in either problem instance highlights the components in the other instance that are directly connected.
7.2 Reducing to Satisfiability

Besides the three reductions from 3CNF-Satisfiability GraphBench also allows to reduce Graph Colorability to Satisfiability. Even though reducing another problem to Satisfiability is not very common the example is well suited for educational purposes. The formula generated is easy to understand and can be formulated intuitively. Every clause makes a constraint statement of the form “Vertex i and vertex j cannot both be colored yellow”. These statements are comprehensible for students and they can also easily understand that the formula is satisfiable iff the graph is colorable.

The implementation of the reduction algorithm and its visualization are also designed to support students. Figure 7.2 shows the tutor during the execution of the reduction algorithm. It illustrates the reduction process by providing information about the current execution step. In the situation shown clauses are added to ensure that vertices connected by an edge have different colors. The tutor highlights the two vertices and the edge that are being processed. Additionally it displays information about the variables and the
colors currently dealt with.

Fig. 7.2: Reducing Graph Colorability to Satisfiability. The tutor highlights the vertices and the edge currently being processed and provides additional information.

7.3 Reductions among graph problems

GraphBench contains several tutors for polynomial time reductions dealing with two graph problems. All except one use reduction algorithms that work only with edge operations to transform the origin graph.

The three problems Vertex Cover, Clique and Independent Set can be pairwise reduced to each other in both directions. GraphBench therefore offers three tutors, one for each pair. They differ from other reduction tutors in two aspects:

1 Students can modify and solve both problem instances freely.
2 The tutors constantly synchronize the two problem instances and do not feature an explicit reduction algorithm. I.e. changes in one of the problem instances are instantaneously applied to the other problem instance.
A similar tutor in GraphBench deals with the reduction from HAMILTONIAN CIRCUIT to TRAVELING SALESMAN for weighted graphs (figure 7.3). The reduction algorithm also only uses edge operations, but does not work in both directions. The reduced graph (i.e. the TRAVELING SALESMAN instance) consists of a copy of the original graph (i.e. the HAMILTONIAN CIRCUIT instance) with edges having weight 1. This reduced graph is extended to a complete graph by adding edges with bigger edge weights. The tutor makes the reduction process more transparent by showing the dependencies between the edges of the two graphs. Selecting an edge in one graph highlights the corresponding edge in the other graph, if such an edge exists. If the corresponding edge does not exist the tutor shows a dotted red edge in place of it, to indicate its absence.

Finally GraphBench also contains a tutor for the reduction from VERTEX COVER to HAMILTONIAN CIRCUIT. The reduction algorithm and the resulting graphs are more complex than the other graph related reductions. Even for a small VERTEX COVER instance the corresponding HAMILTONIAN CIR-
CUIT graph becomes large and difficult to understand. An instance with three vertices and three edges will for example result in a graph with more than 40 vertices and more than 40 edges (figure 5.9). To support students the tutor therefore uses ‘selective level of detail’ to help them focus on the most important components of the reduced graph. As with all other reduction tutors the elements of the two problem instances are dynamically connected and relations are shown upon user actions.
Chapter 8

Writing algorithms with GraphBench

Besides the tutors for NP-complete problems and polynomial time reductions GraphBench also offers a programming environment. It allows students to design, implement and run graph algorithms using Java™.

The programming environment includes a simple programming editor, originally designed by Raimond Reichert for the JavaKara environment [Rei03] and has been adapted for GraphBench. It provides a Java code template to help students write their first programs (figure 8.1).

```java
import ch.ethz.graphbench.toolbox.*;

public class MyAlgorithm extends GraphAlgorithm{
    // Define your methods here
    public void executeAlgorithm(){
        // put your main program code here, for example:
        for(int i = 0; i < graph.getVertexCount(); i++){
            graph.getVertex(i).setVertexState(Vertex.PROCESSED_STATE);
        }
    }
}
```

Fig. 8.1: The code template of the GraphBench programming environment
All programs must inherit from the class `GraphAlgorithm`. This class is an interface to the GraphBench programming environment and provides access to an instance of the class `Graph`. The class was designed to give students access to the functionality they need, while hiding implementation details needed to interact with GraphBench not relevant for them.

Any algorithm written by students in the programming environment of GraphBench must contain a method called `executeAlgorithm`. It defines the entry point into their program and is called from GraphBench. Students are otherwise free to create their own methods, variables and classes. There is only one technical restriction: if the class defines a constructor, it must be parameterless. The reason is that the system instantiates an object of the students class, and it cannot know what values it should pass to the constructor.

Before executing their program students have to compile it. The programming environment provides this functionality including a display for possible compilation errors. When a compilation error occurs, students can click on the error message to highlight the corresponding line of code. If a program has compiled successfully GraphBench allows to execute it at various speeds.

The programming environment provides a simple data structure (figure 8.2) and a visualization for graphs. The graph data structure provides standard operations to modify graphs, e.g. adding and removing vertices and edges, moving vertices, or setting edge weights. All changes in the graph data structure are instantaneously visualized by GraphBench.

![Data structure provided by GraphBench to handle graphs.](image)

Fig. 8.2: Data structure provided by GraphBench to handle graphs.
Besides the standard operations on graphs and the visualization of the graph data structure, GraphBench also provides functionality to illustrate the execution process of an algorithm. Students have the possibility to mark vertices and edges as being queued, active, processed, valid or invalid. Figure 8.3 shows the visualization of the different states for a vertex. The states have no actual, semantical meaning. They are simply a help for students to visualize the execution process of their algorithm, without having to program any graphical output.

![Different states of a vertex](image)

Fig. 8.3: Different states of a vertex.

In addition to the graph data structure and its functionality the programming environment also provides the class `Toolbox` (figure 8.4). It provides functionality for drawing arbitrary lines and shapes within the view of the graph. This might for instance be helpful to display the working front of a plane sweep algorithm. The class `Toolbox` additionally provides the possibility to output text to a separate window. This feature was requested by students, for instance to be able to print debug information.

![Class Toolbox](image)

Fig. 8.4: The class `Toolbox`
Chapter 9

Design and Architecture of GraphBench

In this chapter we describe how we designed the software GraphBench and show the underlying architecture. We introduce important libraries and present several implementation issues.

9.1 Overall System Architecture

GraphBench was designed to allow the integration of a variety of tutors for various NP-complete problems and polynomial time reductions. While some tutors are similar, others have almost nothing in common. The architecture of GraphBench was therefore designed to a) support the integration of various different problem tutors, b) offer a common code base that makes the creation and integration of new tutors as comfortable as possible, and c) provide a framework that makes such a large system manageable.

Figure 9.1 shows a high-level overview of the GraphBench architecture consisting of a highly scalable framework of Java components. The overall GraphBench software consists of over 370 classes in more than 110 packages with more than 90'000 lines of code.

At the foundation of the GraphBench system lies the GraphBench framework consisting of several general purpose libraries and classes that provide a common code base. The main task of the framework is to create, integrate, display and handle the different tutors. Further it was designed to incorporate functionality that is used by all or most tutors, such as file-handling and the use of pseudo-code to illustrate algorithms. Additionally we added libraries for handling and displaying graphs. This because a majority of tutors deal with graph related problems. We introduce the GraphBench framework in detail in
chapter 9.2.

Built on the GraphBench framework is the Problem library. It consists of packages and classes that implement the different tutors for NP-complete problems. They provide problem-specific functionality, e.g. creating and displaying problem instances or handling content-specific user interaction. The Problem library was designed to a) reuse as much code from the GraphBench framework as possible and to b) provide as much functionality for the Reduction library as possible. The Reduction library contains packages and classes dealing with tutors for polynomial time reductions.

Finally the GraphBench system contains a library for the programming environment that is built onto the GraphBench framework. It mainly takes advantage of the graph and file-handling libraries provided by the GraphBench framework. Because the programming environment is different from the tutors for NP-complete problems and polynomial time reductions it was designed independently of them.

9.2 The GraphBench Framework

The GraphBench framework was designed to allow the integration of independent environments without imposing restrictions on the types of the environments or the topics covered. However a special interest was given to the development and integration of tutor environments for NP-complete problems and polynomial time reductions.

Figure 9.2 shows the main classes and a selection of their methods and at-
tributes from the top level package `ch.ethz.graphbench.framework`.

Fig. 9.2: The main classes of the package `ch.ethz.graphbench` that constitute the basis of the GraphBench framework.

At the heart of the package lies the class `Browser`. It provides functionality for adding, removing and displaying environments and for handling basic user interaction, i.e. displaying the help window. All environments to be displayed by the browser must implement the interface `EnvironmentInterface` to provide the browser with information about their content, e.g. a short description or a help file. Environments that implement the interface `SavableEnvironmentInterface` must implement additional functionality to read and write files, e.g. describing a problem instance such as a graph, and must provide information about the files to be written and read. They however must not deal with high level file handling, e.g. displaying a “save dialog”, since this functionality is
implemented by the Browser class.

The graphical user interface (GUI) of the environments is not created by
the Browser or the environments themselves, but by instances of GuiFactoryInterface. Every environment has a reference to its GuiFactoryInterface and passes it to the Browser when being displayed within the Browser. Separating
the creation of the GUI (GuiFactoryInterface) from the functionality of the environment (EnvironmentInterface) makes the development of new environments easier and facilitates code reuse. While the functionality of two environments
differ, their GUI might be very similar. The tutor environments for CLIQUE, VERTEX COVER and INDEPENDENT SET for instance all use the same implement-
ation of GuiFactoryInterface.

Instantiating the different environments and adding them to the Browser
is done dynamically by the class EnvironmentBuilder. It uses a configuration
file specifying the environments to be created and integrated into the system.
The main advantage compared to a static creation of the environments with
hard-coded instructions is that altering the featured content is simpler. A new
tutor can for instance be added without having to modify any classes of the
GraphBench framework. A developer only has to extend the Problem library
and modify the configuration file.

The configuration file is XML based and provides information about the
environments to be created (figure 9.3). It specifies the main class and the
type (e.g. tutor for a polynomial time reduction) of the environments.

<environments>
  <environmentlist>problem, reduction</environmentlist>
  <problem>
    <list>clique, sat, ... </list>
    <clique>ch.ethz.graphbench.problem.clique.CliqueContent</clique>
    <sat>ch.ethz.graphbench.problem.sat.SatContent</sat>
    ...
  </problem>
  <reduction>
    <list>satcolorability, satclique, ... </list>
    <satcolorability> ... </satcolorability>
    <satclique> ... </satclique>
    ...
  </reduction>
</environments>

Fig. 9.3: The configuration file specifying the environments to be created.

The class EnvironmentBuilder not only uses this information to create the
environments, but also to create an instance of the class IndexPage. The class
IndexPage is the entry point to the application from a user's point of view, because it is the first environment shown. It displays a graphical overview of all environments and offers more detailed information about them. This is also done dynamically using information from the configuration file. The file provides information about the overview image to be displayed and about the coordinates of the environments within the image to create an interactive image map (figure 9.4).

<environments>
  <hasoverview>true</hasoverview>
  <overview>
    <img>overview.gif</img>
    <coordinates>
      <problem>
        <etsp>2,416,142,100</etsp>
        <hamilton>186,417,142,100</hamilton>
        <colorability>1,276,142,100</colorability>
        ...
      </problem>
      <reduction>
        <satcolorability>146,241,40,37</satcolorability>
        <satclique>330,241,40,37</satclique>
        <colorabilitysat>66,84,194,181,273,274,104,100</colorabilitysat>
        ...
      </reduction>
      ...
    </coordinates>
  </overview>
  ...
</environments>

Fig. 9.4: The configuration file providing information about the overview displayed by the IndexPage.

Besides the core classes listed above the GraphBench framework also contains the libraries graph and pseudocode described next.

Graph library

The majority of tutors part of GraphBench deal with graph related NP-complete problems and polynomial time reductions. To prevent code duplication and to facilitate the implementation of such graph related tutors we created the library ch.ethz.graphbench.graph. It was designed as a stand-alone, extendable library and can be used independently of GraphBench. The library provides general functionality for generating, manipulating and display-
ing graphs. It was designed according to the Model-View-Controller design pattern (figure 9.5).

Fig. 9.5: An overview of the graph library implementing the Model-View-Controller design pattern.

**Model**

The model of the graph library consists of several interfaces and classes with the interface `GraphModelInterface` and its default implementation `GraphModel` as the basis. The `GraphModelInterface` provides functionality for basic graph operations, i.e. creating, adding, removing and accessing edges and vertices or changing graph parameters, e.g. directed or weighted graph.

Vertices and edges are represented by instances of `VertexInterface` and `EdgeInterface` and both come with a default implementation (`Vertex` and `Edge`).
We realized that for algorithm visualization and animation more specialized representations of vertices and edges are desirable. The possibility to store information about the algorithm process directly with the components (e.g. vertices and edges) was needed. We therefore introduced the interface \texttt{AlgorithmComponentInterface}. It defines functionality for any objects to be used in an algorithm (figure 9.6) and defines methods for specifying the state of the object. These states can be queued, active, processed, valid or invalid (see also chapter 8). They can be used for visualization purposes and have no actual semantical meaning.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{algorithm_vert_edge.png}
\caption{Classes specialized for algorithm animation and visualization.}
\end{figure}

\texttt{AlgorithmVertex} and \texttt{AlgorithmEdge} are two implementations of \texttt{AlgorithmComponentInterface}. They inherit from \texttt{Vertex}, respectively \texttt{Edge} and are used in most of the graph-related algorithms of GraphBench.
View

Figure 9.7 shows an overview of the classes responsible for displaying a graph on the screen. At the core of the package is the class GraphView.

The class GraphView manages all instances of RenderableInterface. This interface defines a representation of data model objects to be drawn (e.g. vertices). It holds information about the position and the dimension of the graphical representation and provides functionality for handling user interactions. Two implementations are RenderableEdge and RenderableVertex.

Instances of RenderableInterface are not drawn by the GraphView directly but by an instance of ComponentRendererInterface. This interface defines a single method drawRenderable and has the sole purpose to draw instances of RenderableInterface. Separating the management of the objects to be drawn from the actual drawing process has three main advantages. First, different objects of the same type (e.g. vertices) can be drawn differently by having different instances of ComponentRendererInterface assigned. Second, changing the visualization of single objects or the whole graph can be done by simply registering different instances of ComponentRendererInterface, even at runtime. Third, creating a new visualization is simple, because implementing a new class of type ComponentRendererInterface that provides the method drawRenderable suffices.
Controller

The class `GraphControl` is responsible for receiving and processing user interaction. It is an extension of the Java class `MouseInputAdapter` and handles all mouse events, e.g. dragging of vertices. It uses the instances of `ComponentRendererInterface` to help determine where a mouse action as occurred and how to handle it, e.g. if and how to display a context menu.

Pseudo-code library

One feature of GraphBench is the ability to execute algorithms stepwise or at variable speed and dynamically display a pseudo-code representation. Users can set breakpoints and view statistical information. The basis for these functionalities lies in the package `ch.ethz.graphbench.framework.pseudocode` (figure 9.8) that was designed jointly with Matthias Dreier [Dre03].

![Diagram of Pseudo-code library](image)

Fig. 9.8: The package `ch.ethz.graphbench.framework.pseudocode`
The class **AbstractAlgorithm** is the superclass for all algorithms that can be executed in GraphBench. It implements two interfaces:

**AlgorithmInterface** defines an interface to classes interested in the algorithm and its execution status. Algorithms must provide a textual representation of their pseudo-code, a description of the algorithm and information about the line currently executed (within the pseudo-code). They must additionally allow for listeners to be notified in case of changes in the algorithm, e.g. execution of a step.

**StepperInterface** is an interface of the package defined in the Kara software package. By implementing the **StepperInterface** algorithms can be executed using the classes of the package `ch.educeth.interpreter` written by Raimond Reichert, as described in [Rei03].

The pseudo-code of an algorithm is represented by the class **PseudoCodeProgram**. It implements simple functionality for creating and accessing algorithm information. The actual pseudo-code is stored as a text in HTML format.

Displaying the pseudo-code is done by the class **PseudoCodeHtmlPanel** an extension of the class **SimpleHtmlPanel**. Even though the pseudo-code is written in HTML for which Java provides a specific editor, we implemented a proprietary HTML viewer with the class **SimpleHtmlPanel**. The reason is a better performance both with respect to time and memory required to display the code. The pseudo-codes use only a small fraction of tags and functionality available by HTML. Implementing a viewer that is focused on high performance in respect to the tags and functionality used by the pseudo-code therefore proved to be very valuable. Besides the functionality inherited from **SimpleHtmlPanel** the class **PseudoCodeHtmlPanel** defines methods for managing user-defined breakpoints and statistical information and implements the **AlgorithmListener** interface.

### 9.3 Implementation issues

**Efficiency and Memory**

Choosing Java™ as the programming language for GraphBench has the main advantage of having a system that is platform-independent. However the fact that Java programs are interpreted using a virtual machine causes performance drawbacks. This is especially the case when working with Swing, Java’s graphical user interface library.
We therefore used Borland’s Optimizeit [Bor02] to optimize GraphBench both with respect to time and memory space needed. The first major improvement was achieved by omitting the class JEditorPane when displaying the HTML pseudo-codes. We implemented a proprietary HTML viewer specifically designed for the requirements by the pseudo-code. Using this viewer resulted in significantly faster displaying times.

The second major improvement was reducing the amount of memory used by GraphBench. In the beginning the application used more than 40MB of memory and more than 600’000 objects were instantiated. Extensive analysis of the system showed that the use of standard Swing components was extremely memory consuming. We therefore rigorously limited the number of Swing components, which reduced the memory space needed by GraphBench by 75% to 10MB with only about 150’000 instantiated objects.

Multiple Language

GraphBench was designed not only to be platform independent but also to be language independent. The current version of GraphBench is in English, but this is neither a requirement nor a limitation. All texts within the software are not hard coded, i.e. they are not part of the source code. They are stored in an external configuration file using XML. Therefore changing the language only requires a translation effort, but no programming is required.

File Format for Input and Output

GraphBench uses XML as the file format to store problem instances, e.g. a graph or a Boolean formula. To create the XML representations the Java Architecture for XML Binding (JAXB) is used [Sun02]. The architecture provides an API to automate the mapping between XML documents and Java objects. DTDs (document type definitions) are used to define the XML structure of the files to be written. JAXB implicitly validates the XML structure according to the specified DTD. This makes it extremely easy to assure that a created XML structure is valid and that it can later be used to recreate the Java objects.
Chapter 10

Experience and Evaluation

In this chapter we describe the use of GraphBench and the experience with it. We show a first evaluation and explain its outcome.

10.1 GraphBench at the ETH Zurich

The first use of GraphBench was in spring 2003 in an introductory course to the Theory of Computation at ETH Zurich. The software was used as a demonstrational tool to illustrate NP-complete problems and polynomial time reductions. After a formal introduction to the theory of NP-completeness several NP-complete problems and polynomial time reductions were presented. GraphBench was used to visualize and clarify the theory. The software made it possible to present various instances of the different problems and reductions, ranging from simple to complex ones.

10.2 GraphBench at the National University of Singapore

In fall 2003 we used GraphBench at the National University of Singapore in the course “Combinatorial and Graph Algorithms”. GraphBench was again primarily used as a demonstrational tool in class to introduce various NP-complete problems and polynomial time reductions.

However the software was also used by students for homework assignments. They were required to write basic graph algorithms, such as finding a minimum spanning tree. Students used the built-in Java editor to write their algorithms and used GraphBench to execute and visualize their programs. In addition a majority of the students used GraphBench for their term project. They were
required to write various algorithms using a programming environment of their choice.

An evaluation of GraphBench was not conducted. However students expressed their opinions about GraphBench in a discussion in class and with personal feedback. They showed a very positive attitude towards GraphBench in general. They rated the use of GraphBench to introduce problems and polynomial time reductions as helpful. They stated that the visual introductions and demonstrations of NP-complete problems, polynomial time reductions and algorithms fostered their understanding of the topic. Students also expressed their positive expectations about the use of GraphBench as an educational tool.

10.3 GraphBench at the Free University of Bolzano

In 2005 we conducted a first assessment of GraphBench with 15 students at the Free University of Bolzano in Italy. The assessment took place in a computer lab and students were given two hours to complete an exercise about polynomial time reductions. Students were given a special version of GraphBench that recorded their actions. During the evaluation we observed students and interrogated them from time to time. At the end of the two hour lab session students were asked to fill in a questionnaire about GraphBench.

The general conditions of the assessment does not allow to draw final conclusions about the use of GraphBench as an educational tool. The number of students is to small to derive reliable statistical results. Additionally the limited resources to observe students during the lab session did not allow to conduct close and in detail evaluation of students behavior. However the collected data suggest certain tendencies and allow hypothesis about GraphBench.

Hypothesis 1: GraphBench is useful

The results of the questionnaire show that a significant majority of students judge the use of GraphBench as an educational tool as high or very high. The same can be said about the graphical user interface of GraphBench. A vast majority classified the GUI as being of high or very high quality (figure 10.1).

This result was also supported by students comments during the lab session. They had little or no problems to use GraphBench and repeatedly stated that they felt GraphBench helped them understand the nature of the NP-complete problems and polynomial time reductions.
Fig. 10.1: Students assessments of GraphBench

Fig. 10.2: Students assessments of GraphBench features
When asked about the helpfulness of specific GraphBench features student’s answers indicate that all feature contribute to their understanding of NP-completeness and polynomial time reductions (figure 10.2). The possibility to animate algorithms and to execute them step by step were judged as being most helpful.

A different indication about the usefulness of GraphBench are the number of downloads registered. GraphBench is published on the web and is freely available. The webserver EducETH registered about 200 downloads of the software per month (as of march 2005).

Hypothesis 2: Dynamic dependencies of problems are being used

The evaluation of the logged student activities indicates that the possibility to investigate relations between components of the two problems of a polynomial time reduction is used extensively by students. For our experiment the log files show that for polynomial time reductions students spent about 50% of their time investigating the relations among components.

10.4 Future work

We did not perform an in-depth evaluation of GraphBench, because we concentrated on the design and implementation of the learning environment. A statistically profound evaluation could be part of future work. Such an evaluation must be carried out with a number of students large enough to derive reliable statistical results. One possibility would be to conduct the evaluation as part of a course on the theory of computation. The students will have to be split in two groups, one working with GraphBench and a control group working without GraphBench.

The evaluation will have to consist of several assessments of students’ knowledge. Knowledge should be tested on different levels, for instance:

- Knowledge of facts, e.g. definition of NP-completeness.
- Ability to replicate knowledge, e.g. simulate a solution algorithm that has been studied, reduce a given problem instance for a reduction that has been studied.
- Ability to create new knowledge, e.g. develop a reduction algorithm for a reduction that has not been studied.
A first assessment will have to take place prior to the test phase. This to record the previous student knowledge.

Besides evaluating knowledge the evaluation must also record additional information, for instance about how motivated a student is to study, or how often a student studies and for how long.

Conducting an evaluation as described might be very difficult to achieve with a single class. For a mandatory course with a final exam it might not be feasible to divide students into groups using different learning approaches. Students of either group could feel disadvantaged.

10.5 Conclusions

Our use and evaluation of GraphBench do not allow to make significant statements about the usability of the software. However they indicate that GraphBench is well accepted by students and that they feel GraphBench is useful as a learning environment. Additionally the functionality offered by GraphBench seems to fit the requirements of students.
Appendix A

Algorithms in GraphBench

A.1 NP-complete problems

Clique: Backtracking

% The algorithm works on a graph G=(V,E), with n=|V|.
% It tries to find a clique of size k
\textit{currentVertex} = getFirstVertex()
\textit{solution} = createFixedSizeStack(k)
\textbf{while} notEmpty(\textit{solution}) \textbf{or} isDefined(\textit{currentVertex}) \textbf{do}
\textbf{if} isFull(\textit{solution}) \textbf{then}
\textbf{if} isValid(\textit{solution}) \textbf{then}
\hspace{1em} halt()
\textbf{else}
\hspace{1em} currentVertex = pop(\textit{solution})
\textbf{end if}
\textbf{else if} isDefined(\textit{currentVertex}) \textbf{then}
\hspace{1em} push(\textit{solution}, currentVertex)
\textbf{else}
\hspace{1em} currentVertex = pop(\textit{solution})
\textbf{end if}
currentVertex = getNextVertex(currentVertex)
\textbf{end while}
Independent Set: Backtracking
% The algorithm works on a graph G=(V,E), with n=|V|.
% It tries to find an independent set of size k
\texttt{currentVertex} = \texttt{getFirstVertex()}
\texttt{solution} = \texttt{createFixedSizeStack(k)}
\texttt{while notEmpty(solution) or isDefined(currentVertex) do}
  \% VARIANT: numberPassesLeft
  \texttt{if isFull(solution) then}
  \texttt{if isValid(solution) then}
    \texttt{halt()}
  \texttt{else}
    \texttt{currentVertex} = \texttt{pop(solution)}
  \texttt{end if}
  \texttt{else if isDefined(currentVertex) then}
    \texttt{push(solution, currentVertex)}
  \texttt{else}
    \texttt{currentVertex} = \texttt{pop(solution)}
  \texttt{end if}
  \texttt{currentVertex} = \texttt{getNextVertex(currentVertex)}
\texttt{end while}

Vertex Cover: Backtracking
% The algorithm works on a graph G=(V,E), with n=|V|.
% It tries to find a clique of size k
\texttt{currentVertex} = \texttt{getFirstVertex()}
\texttt{solution} = \texttt{createFixedSizeStack(k)}
\texttt{while notEmpty(solution) or isDefined(currentVertex) do}
  \% VARIANT: numberPassesLeft
  \texttt{if isFull(solution) then}
  \texttt{if isValid(solution) then}
    \texttt{halt()}
  \texttt{else}
    \texttt{currentVertex} = \texttt{pop(solution)}
  \texttt{end if}
  \texttt{else if isDefined(currentVertex) then}
    \texttt{push(solution, currentVertex)}
  \texttt{else}
    \texttt{currentVertex} = \texttt{pop(solution)}
  \texttt{end if}
  \texttt{currentVertex} = \texttt{getNextVertex(currentVertex)}
\texttt{end while}
Traveling Salesman: Backtracking

current\_vertex = getFirstVertex()
current\_tour = createPath(current\_vertex)
while notEmpty(current\_tour) do
  \% VARIANT: number\_Untested\_Paths
  current\_vertex = nextVertexNotInTour(current\_tour, current\_vertex)
  if containsAllVertices(current\_tour) then
    closePath(current\_tour)
    current\_vertex = removeLastVertex(current\_tour)
  else if isDefined(current\_vertex) then
    appendVertex(current\_tour, current\_vertex)
  else
    current\_vertex = removeLastVertex(current\_tour)
  end if
end while

Traveling Salesman: Branch and Bound

open\_Paths = create\_Sorted\_List()
current\_Path = create\_Path(first\_vertex)
while notValid(current\_Path) do
  \% VARIANT: number\_Untested\_Paths
  foreach vertex not in current\_Path do
    newPath = create\_Path(current\_Path, vertex)
    if containsAllVertices(new\_Path) then
      closePath(new\_Path)
    end if
    insert(open\_Paths, newPath)
  end foreach
  current\_Path = remove\_Shortest\_Path(open\_Paths)
end while

Traveling Salesman: Nearest Neighbor Heuristic

\% The algorithm works on graph G = (V, E), with n = |V|
current\_Path = create\_Path(first\_vertex)
while sizeOf(current\_Path) < n do
  \% INVARIANT: 1 <= sizeOf(current\_Path) <= n
  \% VARIANT: n - sizeOf(current\_Path)
  next\_Vertex = find\_Closest\_Vertex()
  append(current\_Path, next\_Vertex)
end while
**Traveling Salesman: Greedy Heuristic**

```plaintext
while current tour is not complete do
  % VARIANT: numberMissingEdges
  addShortestValidEdge()
end while
```

**Traveling Salesman: Convex Hull Heuristic**

% The algorithm works on graph G = (V,E), with n = |V|
```plaintext
currentPath = createConvexHull()
while sizeOf(currentPath) < n do
  % INVARIANT: 2 <= sizeOf(currentPath) <= n
  % VARIANT: n - sizeOf(currentPath)
  insert(vertex closest to currentPath)
end while
```

**Traveling Salesman: Two-Opt Local Search**

```plaintext
while there are swappable edges do
  swapEdges()
end while
```

**Hamiltonian Circuit: Backtracking**

```plaintext
currentPath = createPath(first vertex)
openPaths = createStack(currentPath)
while notEmpty(openPaths) do
  % VARIANT: numberUntestedPaths
  currentPath = pop(openPaths)
currentVertex = getLastVertex(currentPath)
foreach vertex not in currentPath do
  if connected(vertex, currentVertex) then
    newPath = createPath(currentPath, vertex)
    if isValidCircuit(newPath) then
      halt()
    else
      push(openPaths, newPath)
    end if
  end if
end foreach
end while
```
Graph Colorability: Backtracking

\[
colored\text{Vertices} = \text{createStack()}
\]
\[
open\text{Vertices} = \text{createStack(all vertices)}
\]
\[
current\text{Vertex} = \text{pop(openVertices)}
\]
\[
useNextColor(current\text{Vertex})
\]
\[
\textbf{while} \text{ solution not found} \textbf{do}
\]
\[
\% \text{ INARIANT: notEmpty(openVertices) or solution found}
\]
\[
\% \text{ VARIANT: numberOfRemainingColorings}
\]
\[
\textbf{if} \text{ existsConflictingEdge()} \textbf{then}
\]
\[
current\text{Vertex} = \text{pop(coloredVertices)}
\]
\[
\textbf{else}
\]
\[
current\text{Vertex} = \text{pop(openVertices)}
\]
\[
\textbf{end if}
\]
\[
useNextColor(current\text{Vertex})
\]
\[
\textbf{while} \text{ isNotColored(current\text{Vertex})} \textbf{do}
\]
\[
\% \text{ INARIANT: current\text{Vertex} is uncolored}
\]
\[
\% \text{ VARIANT: sizeOf(coloredVertices)}
\]
\[
push(open\text{Vertices}, current\text{Vertex})
\]
\[
\textbf{if} \text{ isEmpty(coloredVertices)} \textbf{then}
\]
\[
halt()
\]
\[
\textbf{end if}
\]
\[
current\text{Vertex} = \text{pop(coloredVertices)}
\]
\[
useNextColor(current\text{Vertex})
\]
\[
\textbf{end while}
\]
\[
push(colored\text{Vertices}, current\text{Vertex})
\]
\[
\textbf{end while}
\]

Graph Colorability: Greedy Heuristic

\[
open\text{Vertices} = \text{createList(all vertices)}
\]
\[
\textbf{while} open\text{Vertices} \text{ is not empty} \textbf{do}
\]
\[
\% \text{ VARIANT: sizeOf(openVertices)}
\]
\[
current\text{Vertex} = \text{removeBiggestVertex(openVertices)}
\]
\[
\text{if findColor(current\text{Vertex}) is successful} \textbf{then}
\]
\[
\textbf{foreach} \text{ uncolored neighbor of current\text{Vertex} do}
\]
\[
\text{if findColor(neighbor) is successful} \textbf{then}
\]
\[
\text{removeElement(openVertices, neighbor)}
\]
\[
\textbf{else}
\]
\[
halt()
\]
\[
\textbf{end if}
\]
\[
\textbf{end foreach}
\]
\[
\textbf{else}
\]
\[
halt()
\]
\[
\textbf{end if}
\]
\[
\textbf{end while}
\]
Graph Colorability: Edge Optimization

createRandomColoring()
while solution is not valid do
  currentEdge = getRandomConflictEdge()
  if findBetterColor(getStartVertex(currentEdge)) is unsuccessful then
    findBetterColor(getEndVertex(currentEdge))
  end if
end while

Satisfiability: Backtracking

variableList = createList(all variables)
setVariableValues(variableList, false)
while formula is not satisfied do
  % VARIANT: numberUntestedAssignments
  currentVariable = getFirstElement(variableList)
  while currentVariable is true do
    % INVARIANT: currentVariable is defined
    % VARIANT: numberUntestedVariables
    setVariableValue(currentVariable, false)
    currentVariable = getNextElement(variableList, currentVariable)
    if currentVariable is not defined then
      halt()
    end if
  end while
  setVariableValue(currentVariable, true)
end while

Satisfiability: Limited Local Search

randomVariableAssignment()
numberTries = 0
while(numberTries < t) do
  % VARIANT: triesLeft = t - numberTries
  if formula is satisfied then
    numberTries = numberTries + 1
  end if
  currentClause = findUnsatisfiedClause()
  currentLiteral = findRandomLiteral(currentClause)
  changeLiteralValue(currentLiteral)
end while
Satisfiability: Physical Model

setAllVariableValues(0.5)
while formula is not satisfied do
  calculateForces()
  updateVariableValues()
end while

3-Dimensional Matching: Backtracking

currentTriple = getFirstTriple()
while notValid(solution) do
  numberPassesLeft
  if isFull(solution) then
    currentTriple = pop(solution)
  else if isDefined(currentTriple) then
    push(solution, currentTriple)
  else if notEmpty(solution) then
    currentTriple = pop(solution)
  else
    halt()
  end if
  currentTriple = getNextOpenTriple(currentTriple)
end while

A.2 Polynomial time reductions

3-CNF Satisfiability to Graph Colorability

subGraph = createCompleteGraph(numberVariables + 1);
foreach variable in formula do
  variableVertex = createVertex(variable);
  negatedVariableVertex = createVertex(not variable);
  addEdge(variableVertex, negatedVariableVertex);
  foreach vertex in subGraph do
    if indexOf(vertex) is not equal indexOf(variable) then
      addEdge(variableVertex, vertex);
      addEdge(negatedVariableVertex, vertex);
    end if
  end foreach
end foreach
falseColorVertex = getFirstVertex(subGraph);
foreach clause in formula do
  clauseVertex = createVertex(clause);
  addEdge(clauseVertex, falseColorVertex);
  addEdgesToAllLiteralVerticesExceptOccuring(clause);
end foreach
3-CNF Satisfiability to Clique

```plaintext
foreach clause in formula do
  foreach literal in clause do
    newVertex = createVertex(literal)
  endforeach
  foreach vertex in graph do
    if i not equal p and dontClash(zij, zpq) then
      addEdge(newVertex, currentVertex)
    end if
  endforeach
end foreach
end foreach
```

3-CNF Satisfiability to Vertex Cover

```plaintext
foreach variable in formula do
  literalVertex = createVertex(variable);
  negatedLiteralVertex = createVertex(not variable);
  createEdge(literalVertex, negatedLiteralVertex);
end foreach
foreach clause in formula do
  clauseVertexList = createEmptyList();
  foreach literal in clause do
    clauseVertex = createVertex(literal);
    addElement(clauseVertexList, clauseVertex);
    literalVertex = getVertex(literal);
    addEdge(clauseVertex, literalVertex);
  endforeach
  createCompleteGraph(clauseVertexList);
end foreach
```

Vertex Cover to Hamiltonian Circuit

```plaintext
counter = 0
while counter < sizeOf(vertex cover) do
  % VARIANT sizeOf(vertex cover) - counter
  addVertex(counter)
  counter = counter + 1
end while
foreach edge in graph do
  createSubgraph(edge)
end foreach
foreach vertex in graph do
  createEdges(vertex)
end foreach
```
Hamiltonian Circuit to Traveling Salesman

\[ \text{numberVertices} = \text{getNumberVertices(hamiltonGraph)} \]
\[ \text{tspGraph} = \text{createCompleteGraph(numberVertices)} \]
\[ \text{foreach edge in tspGraph do} \]
\[ \quad \text{if containsEdge(edge, hamiltonGraph) then} \]
\[ \quad \quad \text{setEdgeWeight(edge, 1)} \]
\[ \quad \text{else} \]
\[ \quad \quad \text{setEdgeWeight(edge, numberVertices + 1)} \]
\[ \text{end if} \]
\[ \text{end foreach} \]

Graph Colorability to Satisfiability

\[ \text{foreach vertex in graph do} \]
\[ \quad \text{foreach color in color list do} \]
\[ \quad \quad \text{addVariable(currentVertex, color)} \]
\[ \text{end foreach} \]
\[ \text{currentVariables = getVariablesForVertex(vertex)} \]
\[ \text{addClause(currentVariables)} \]
\[ \text{addClausesLimitingToAtMostOneTrueVariable(currentVariables)} \]
\[ \text{end foreach} \]
\[ \text{foreach edge in graph do} \]
\[ \quad \text{startVertex = getStartVertex(edge)} \]
\[ \quad \text{endVertex = getEndVertex(edge)} \]
\[ \quad \text{foreach color in color list do} \]
\[ \quad \quad \text{firstVariable = getVariable(startVertex, color)} \]
\[ \quad \quad \text{secondVariable = getVariable(endVertex, color)} \]
\[ \quad \quad \text{addClausesLimitingToAtMostOneTrueVariable(firstVariable, secondVariable)} \]
\[ \text{end foreach} \]
\[ \text{end foreach} \]
Bibliography


[BGGR] Boroni, C. M., Goosey, F. W., Grinder, M. T., and Ross, R. J. Webworks.


Curriculum Vitae

Education

1993 - 1997  Kantonsschule Baden, AG
1997  Matura, Type C

1997 - 2002  Studies of Computer Science, ETH Zurich
            2002  Master’s degree in Computer Science (Dipl. Inf. Ing. ETH)

2000 - 2005  Studies in Education Sciences, ETH Zurich
            2005  Didaktischer Ausweis ETH in Computer Science

2002 - 2005  Ph. D. studies in Computer Science, ETH Zurich
            Advisor: Prof. Dr. Bertrand Meyer
            Co-referees: Prof. Dr. Angelika Steger, Prof. Dr. Jürg Nievergelt

Awards

2002  European Academic Software Award for the project ‘KaraToJava’