Using Interactive Visualization for Teaching the Theory of NP-completeness

Christian Pape
Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
76128 Karlsruhe
pape@ira.uka.de

Abstract

In this paper we investigate the potential of interactive visualization for teaching the theory of NP-completeness to undergraduate students of computer science. Based on this analysis we developed some interactive Java applets which we use to present an NP-complete tiling problem PUZZLE in our lecture. This software is integrated into our hypertext lecture notes and our students also use it to find an NP-completeness proof for PUZZLE as an exercise.

1 Introduction

The notion of NP-completeness plays an important role in computer science. It is commonly agreed that once a problem is known to be NP-complete it is unlikely to find an efficient, i.e. polynomial time, algorithm to solve it. This has practical consequences: Many existent optimization problems are known to be NP-complete and, hence, it is usually a waste of time to try to develop algorithms for these problems with standard techniques like, for example, divide-and-conquer. Instead other techniques like simulated annealing are employed, that often lead to good and acceptable results in practice. Therefore it is important that students of computer sciences understand the concept of NP-completeness, and are able to check whether a problem is NP-complete or not.

Due to the rigorous mathematical treatment of complexity theory, many (if not most) undergraduate students have a serious problem mastering the concept of NP-completeness and, moreover, in developing their own NP-completeness proofs. We will report in this paper on our ongoing activities to use visualization in teaching theoretical computer science at the undergraduate level and show how the education of the theory of NP-completeness can benefit from multi-media techniques.

After recapitulating basic concepts of complexity theory we investigate in Section 3 the potential of interactive visualization to support the development and presentation of NP-completeness proofs. In Section 4 we illustrate our ideas using a certain NP-complete tiling problem as an example, which is presented in our lecture with the help of Java-applets. Furthermore the students use this software to develop an NP-completeness proof as an exercise. These applets and other interactive courseware is integrated into our HTML based hypertext lecture notes.[1] It can be used with every Java capable HTML browser.

2 Basic Definitions

To make sure that we are on common grounds we briefly rehash the definitions of Turing machines, the classes P and NP, and the notion of NP-completeness. For a detailed description we refer to [Kfoury et al., 1986]. Turing machines (TM) are the traditional way to formalize the notion of computable functions. They were invented in 1936 by Alan Turing [Turing, 1936] and because of their simplicity they are still a very attractive and convincing concept.

A *deterministic Turing machine* (DTM) consists of:

1. A two-way-infinite *tape* divided into cells. Each cell contains a *symbol* taken from a fixed set Σ. A special symbol B (*blank*) must occur in Σ.

2. A read/write head scanning exactly one cell at a time and being able to move along the tape.

3. A control unit which at any point in time is known to be in one internal state taken from a fixed set , of states.

The behavior of the TM is described as follows by a transition function

$$\delta : , \times \Sigma \to , \times \Sigma \times \{L, R, U\} \quad :$$

If q is the current state of the TM, s is the symbol currently scanned, and $\delta(q, s) = \langle r, t, d\rangle$ holds, then the TM overwrites s with t, changes to the internal state r, and the read/write head, according to the value of d moves to the left if $d = L$, to the right if $d = R$ or remains at the current position if $d = U$. The Turing machine *stops* if $\delta(q, s)$ is undefined for the current state q and the current symbol s.

[1] `http://i12www.ira.uka.de/~info3/skript/companion/companion.htmlonly`

A *configuration* of a TM T consists of the head position, the current tape, and state of T. A configuration is denoted by the relevant part of the tape written as a string with the current state before the symbol corresponding to the cell under the the head. For example $B00q_1 1B$ is a shorthand for a TM consisting of: the tape with symbols $0, 0, 1$ and blanks to the left and to the right; the current state q_1; and the read/write head over the cell with symbol 1.

A DTM computes a function $f : \Sigma^* \to \Sigma^*$ starting with an initial state and a tape that contains in some way the arguments for f. Instructions are provided to read the function value from the final tape inscription when the TM stops.

Non-deterministic Turing machines (NTM) are defined similar to deterministic TM but with a transition function mapping a state and symbol to a *set* of possible actions:

$$\delta : ,\ \times \Sigma \to 2^{\Gamma \times \Sigma \times \{L,R,U\}}$$

The behavior of the NTM is defined as follows: If q is the current state of the NTM and s is the symbol currently scanned, then one triple $\langle r, t, d \rangle \in \delta(q, s)$ is chosen non-deterministically and the internal state of the NTM changes accordingly.

A (deterministic or non-deterministic) TM *decides* a language $L \subseteq \Sigma^*$ iff it stops on every input $w \in \Sigma^*$ and the final tape contains a 1 if $w \in L$ holds, and a 0 otherwise.

In complexity theory, problems (e.g. the problem PUZZLE introduced in Sec. 4.1) are encoded into formal languages $L \subseteq \Sigma^*$ for a fixed signature Σ to abstract from their informal description. An instance of a problem is a word $w \in \Sigma^*$ and the question is whether w belongs to L or not. This makes it much easier to investigate them in a more general way. Therefore we assume that a decidability problem consists of a language $L \subseteq \Sigma^*$ such that $x \in L$ iff x is solvable. By definition L is decidable iff the question $x \in L$ can be decided for every $x \in \Sigma^*$ with a (deterministic or non-deterministic) TM.

It is commonly agreed among computer scientists that all problems which can be solved by a DTM within a polynomial time bound are feasible, i.e. an efficient algorithm to decide it exists. All other problems are not feasible. This classification leads to the following definition:

The class P and NP are the sets of all problems $L \subseteq \Sigma^*$ which can be decided by a deterministic resp. non-deterministic Turing machine T in at most $p(n)$ steps where p is a polynom and n the length of the input word $w \in L$.

Because every problem in NP is bounded by a polynom the following can be proved:

Theorem 1 *For every language $L \subseteq \Sigma^*$ in NP there exists a non-deterministic TM that stops on every input with resulting output* 1 *iff $w \in L$ holds.*

First of all one might guess that NTM are more powerful, i.e. can decide more problems than deterministic ones. However, this impression is wrong and a proof is presented to undergraduates that both models are equally powerful. But one might expect that non-deterministic computations are less complex than deterministic ones. If this is really the case, i.e. P=NP holds or not, is one of the most important unsolved problems in computer science, because currently only exponential time algorithms are known to solve problems that are in NP but not in P. The technique of reduction and notion of NP-completeness can be employed to find such unfeasible problems:

A problem L' is *polynomial reducible* on a problem L, denoted by $L' \leq_p L$, if there exists a DTM that computes a function $f : \Sigma^* \to \Sigma^*$ in polynomial time such that $w \in L'$ iff $f(w) \in L$.

A problem $L \in$ NP is *NP-complete* if every problem $L' \in$ NP is polynomial reducible on L, i.e. $L' \leq_p L$.

3 Developing and Presenting NP-completeness Proofs

There are two general methods for proving the NP-completeness of a problem $P_1 \subseteq \Sigma^*$:

First, by finding a polynomial reduction from another problem $P_2 \subseteq \Sigma^*$ which is known to be NP-complete, i.e. $P_2 \leq_p P_1$.

Second, by an *elementary* proof, i.e. by giving a transformation, computable in polynomial time by a DTM, that maps every NTM with given input tape into an instance $I_1 \in \Sigma^*$ such that $I_1 \in P_1$ iff the corresponding NTM stops.

The main part of both methods is the construction of a mapping from one domain (an NTM or an NP-complete problem) into another domain (a problem).

Interactive visualization can support the presentation and development of such an NP-completeness proof in several different manners:

If the problem itself is presented in a lecture or textbook, then it is a useful and traditional approach to give a visualization of some examples of the problem. An electronic text, in addition, can visualize the problem in many other ways: Instead of presenting static pictures, it is possible to simulate an instance of the problem, for example, an instance of the traveling salesman problem can be represented as a graph (or map) and the students can search for a short round tour interactively. This may give them a better comprehension of the problem and its complexity (see Sec. 4.1 for an example of such a visualization in case of a tiling problem). Moreover, instead of choosing one instance of the problem, it is possible to provide an editor for typing in an instance of the problem and then investigating it interactively. If a reduction $L \leq_p L'$ is used in a proof, then both problems L and L' can be visualized as described. But what is more, we are able to map an instance of L' into an instance of L and investigate the effects and properties of the transformation visually.

If an NP-completeness proof is not given but has to be carried out by the student, e.g. as an exercise, then an interactive visualization can also help him understanding the problem better. However, the crucial and difficult part of any NP-completeness proof is the construction of the mapping between the two domains. It is likely that most students fail on the first attempt to find the correct transformation. Here an interactive software can help them to check their transformation on suitable problem instances. First of all, the student has to type in the transformation (which depends on the given problems). An appropriate implementation can focus on the relevant and interesting parts of the transformation such that nonsensical inputs are impossible. The student then has the opportunity to check his transformation for errors much easier than doing it by hand. As another advantage, the result can often be checked automatically (for example if only finitely many correct transformations exist). Thus the student get immediate feedback on the correctness of his solution. But even if the solution can not be checked automatically, it may be possible that the student can test his solution on certain instances of the problem interactively with the help of a simulation of the problem (see Sec. 4.2 for an example). In many situations it might also be possible to generate counterexamples for wrong solutions automatically. These counterexamples then can be inspected with the help of the software.

As an additional remark let us note that the technique of reduction is not restricted to the framework of NP-completeness. In computer science reductions are also used, for example, to prove lower time bounds or the undecidability of problems; thus most of the above techniques for visualizing reductions can be used in these areas as well.

4 NP-completeness Proof for a Tiling Problem

Every introduction into the theory of NP-completeness starts with the definition of the classes P, NP and the notion of NP-complete problems, and then presents the technique of reduction to prove the NP-completeness of one problem using the known NP-completeness of another problem. However, this leaves open the question of how to proof the NP-completeness of the latter problem. In our lecture we do this by presenting Cooks Theorem, which states that the problem of deciding the satisfiability of propositional formulas is NP-complete [Cook, 1971], and we give an elementary proof based on a transformation of NTMs into propositional formulas.

To understand this transformation one has to be familiar with propositional logic. Though first year students are teached basic parts of formal logic, many (if not most) of them have problems to understand the proof, in particular its general scheme and the validity of the transformation.

To make sure that the students well understand this proof and the concept behind it, they have to carry out an elementary NP-completeness proof for another problem — a tiling problem — as an exercise. In our hypertext the presentation of this problem is supported by an interactive simulation of some of its instances (Sec. 4.1); and we developed a Java-applet which makes it much easier for the students trying to solve the exercise (Sec. 4.2).

4.1 The Tiling Problem

Consider the following tiling problem PUZZLE [Wagner, 1994]:

INSTANCE: Given an alphabet Σ, a collection $T_1, \ldots, T_n \in \Sigma^4, n \geq 1$ of "tiles" (where $\langle a, b, c, d \rangle$ denotes a type of tile with top side a, right side b, bottom side c, and left side d), and a quadratic frame $F \in \Sigma^{4t}, t \geq 1$.

QUESTION: Is it possible to fill the frame F with tiles of the type $T_1, \ldots, T_n$ such the sides of the tiles match with the corresponding sides of the frame and their neighboring tiles?

In our lecture notes we exemplify this problem with visual representations of two small instances: one which is solvable and another one which has no solutions. Figure 1 (a) shows the paper version of the solvable example, where $\Sigma = \{1, 2, 3, 4, 5\}, t = 3, F = \langle 1, 2, 3, 1, 1, 3, 2, 5, 2, 2, 3, 5 \rangle, T_1 = \langle 2, 3, 2, 4 \rangle, T_2 = \langle 2, 1, 2, 3 \rangle, T_3 = \langle 1, 4, 2, 2 \rangle, T_4 = \langle 4, 4, 5, 5 \rangle, T_5 = \langle 2, 5, 2, 5 \rangle$. The quadratic frame of an instance of PUZZLE is build from F starting from the top left edge in clockwise orientation.

In the hypertext this picture is omitted and replaced by a simulation of the instance (Fig. 1 (b)): Here, the students can pick the tiles with the mouse and try to fill the frame with them. If two tiles do not fit, i.e. the touching borders have different symbols, then these parts are highlited black. The advantage of this simulation over the paper version is the immediate response to a wrong or correct attempt to solve the instance, which makes it easier for the student to understand the problem in detail. After some experiments with the simulation most students should be familiar with PUZZLE, in particular because of its obvious relation to jigsaw puzzles.

For the proof of the NP-completeness of PUZZLE we consider one-way-infinite Turing machines, which differ from two-way-infinite Turing machines

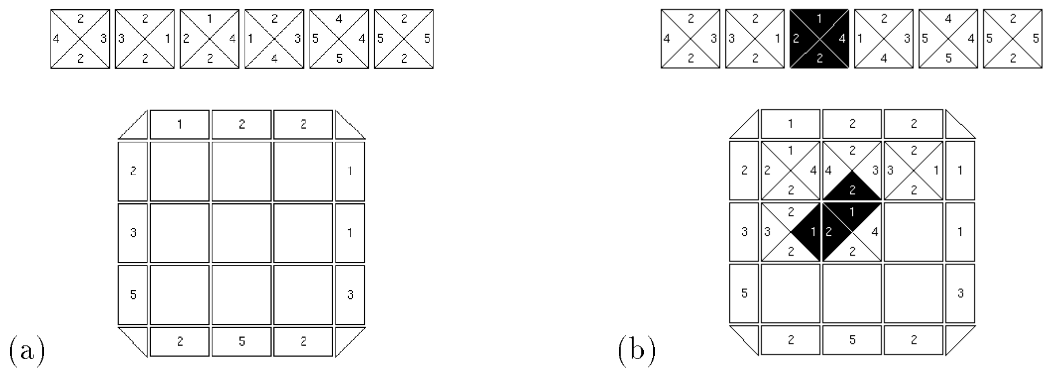

Figure 1: Example of PUZZLE: (a) paper version, (b) simulation in hypertext

only in the restriction that the tape has a leftmost cell (and an infinite supply of cells to the right); in case the transition function attempts to move the head off the tape to the left the machine just stops.

In our lecture we prove that the expressive power of one-way-infinite Turing machines does not differ from that of two-way-infinite ones. Its proof is based on a transformation of one type of Turing machine into the other. The proof is given in our hypertext lecture notes; its central parts are visualized with corresponding simulations of the original and the transformed Turing machine (see [Pape and Schmitt, 1997] for details).

4.2 Supporting the Development of an NP-completeness Proof for PUZZLE

The central part of the NP-completeness proof of PUZZLE is to show that every problem $L \in \mathrm{NP}, L \subseteq \Sigma^*$ is polynomial reducible on PUZZLE. By definition of NP and Theorem 1 we conclude that a (one-way-infinite) NTM T exists that stops on input $w \in \Sigma^*$ with final output 1 after $p(|w|)$ steps (where p is a polynom) iff $w \in L$ holds. From the description of T and given input w we have to construct an instance of PUZZLE that is solvable iff T stops on input w with final output 1.

The main idea of the proof is to simulate T with the corresponding instance of PUZZLE by encoding a configuration of T and all possible transitions into a row of tiles. These tiles must be designed such that a row r of tiles fits under the preceding row iff r corresponds to a next possible configuration of T.

The top side of a tile contains the symbol of the corresponding cell. If the read/write head is over this cell than the current state is added to the top side, too, and the left and right sides of the tiles are used to simulate the head movement over the tape. The bottom side of a tile contains the information for the resulting configuration.

The frame of the puzzle has size $t = p(|w|)$ and is generated as follows: The top side of it is constructed from the given input tape and initial state of T; the bottom side is build from the given output 1 (the rest is filled with blanks); the left and right side is filled with special symbols ($\#$) not occuring in Σ.

Figure 4.2 shows an example for a puzzle related to some TM (the details of the TM need not to concern us here): The top side corresponds to the initial configuration $q_1 01B$ and the bottom side corresponds to a (desired) final configuration $q_2 100B$. The first row of tiles corresponds to the initial configuration $q_1 01B$ (top side of tiles) and the resulting next configuration $1q_1 1B$ (bottom side of tiles); the tile in the top left corner of the puzzle simulates the transition $\langle q_1, 0\rangle \mapsto \langle q1, 1, R\rangle$.

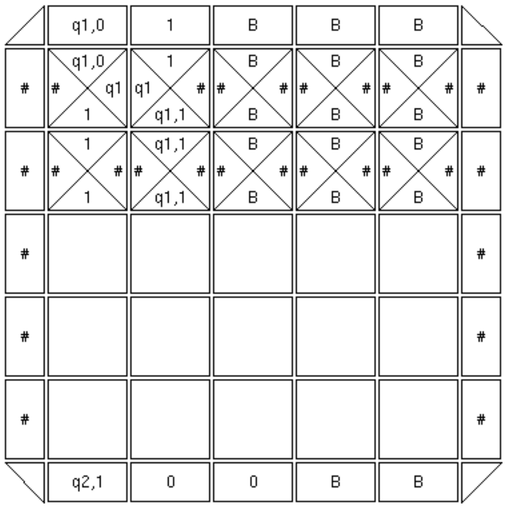

Figure 2: A partial filled PUZZLE related to some non-deterministic Turing machine

The difficult part of the proof is to find this transformation. For an introductional course on complexity theory this task is far to complex to solve without any hints. To make the search for a correct transformation easier, we developed a Java-applet with which it is possible to specify the transformation, to type in a Turing machine to which the transformation is automatically applied, and to try to solve the resulting puzzle.

The frame is build automatically from the input tape and the output; therefore it is not necessary for the student to specify this part of the transformation. Furthermore, the graphical user interface for the input of the transformation is designed to forbid as much nonsensical inputs as possible without giving too much hints.

It is possible to classify the types of the tiles into six categories depending on the different actions and states of the NTM. Recall that a tile is directly related to a cell of the tape:

1. Neither before nor after the next step of the NTM the head is over this cell of the tape.
2. $(p, b, U) \in \delta(q, a)$, i.e. the head is above this cell, a new symbol is written to it, the head moves to the right, and the internal state of the NTM changes. In the applet the correct solution for this category is given as a hint.
3. $(p, b, R) \in \delta(q, a)$, analogous to 2.
4. $(p, b, L) \in \delta(q, a)$, analogous to 2.
5. The head moves into the cell from the left.
6. The head moves into the cell from the right.

In these categories a, b and q, p are placeholders for the corresponding symbols and states of the NTM. The input of the transformation for one category is done by a multiple-choice selection of symbols, states, or pairs of symbols and states (see Fig. 3 for categories 2 and 5). This approach makes it a lot easier for the students to find the correct transformation and it inhibits too much nonsensical input.

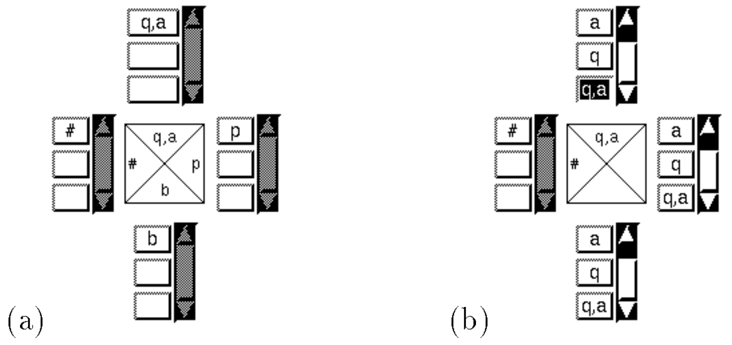

Figure 3: Input of the transformation: (a) correct input for category 2, (b) partially filled out category 5

After the transformation is typed in, the students can feed in an NTM and have a closer look at the corresponding instance of PUZZLE, which is automatically generated applying the given transformation. They now can experiment with their solution and get confident about its correctness or they can try to falsify it by finding counterexamples, i.e. Turing machines which stop on a certain input but where the corresponding instance of PUZZLE has no solution or vice versa. If the latter happens then it is possible to modify the transformation and they can try to experiment with the new version again.

5 Conclusion and Further Work

We investigated the potential of interactive visualization for the development and presentation of NP-completeness proofs in computer science education and reported on our implementation of an example which incorporates most of our ideas. We believe that with this software students can learn certain parts of NP-completeness theory much better than with traditional teaching methods.

The next step in our work will be a practical test during the lecture and tutorials. Based on this evaluation we are going to improve our software, e.g. by generating counterexamples for wrong solutions.

Currently the students' solutions are still checked by human tutors, but we are also working on an fully automatic intelligent tutor for helping students with checking NP-completeness proofs based on the reduction technique for certain variants of 3SAT.

References

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem proving procedures. In *Proc. Third Annual ACM Symposium on the Theory of Computing*, pages 151–158.

[Kfoury et al., 1986] Kfoury, A. J., Moll, R. A., and Arbib, M. A. (1986). *A Programming Approach to Computability. Second printing.* Texts and Monographs in Computer Science. Springer-Verlag.

[Pape and Schmitt, 1997] Pape, C. and Schmitt, P. H. (1997). Visualizations for Proof Presentation in Theoretical Computer Science Education. In Halim, Z., Ottmann, T., and Razak, Z., editors, *Proceedings of International Conference on Computers in Education, Kuching, Sarawak, Malaysia, December 2–6*, pages 229–236, Charlottesville. Association for the Advancement of Computing in Education.

[Turing, 1936] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Mathematical Society*, 42:230–265.

[Wagner, 1994] Wagner, K. W. (1994). *Enführung in die Theoretische Informatik. Grundlagen und Modelle.* Springer.