MATHEMATIQ

The Newsletter of MathSIG
(Special Interest Group within Mensa Austria)

Special Edition on Theoretical Computer Science

http://www.hugi.scene.org/adok/mensa/mathsig/

Editorial

Dear Readers:

Welcome to a special edition of MATHEMATIQ. On popular request I have translated several articles of mine that deal with theoretical computer science to the English language so that members of Mensa International and other people interested in this branch of science will be able to read it.

The articles in this issue introduce laymen to the fascinating area of theoretical computer science and two particular open problems. I focused on theoretical computer science during my studies at the Vienna University of Technology and received a very good training in this area. Altruistic as I am, I do not keep this knowledge to myself but try to convey it to others as well for the sake of advancement of education and science.

MATHEMATIQ is the official newsletter of the MathSIG, a special interest group within Mensa Austria. Regularly it issues in the German language, which is the official language of the Republic of Austria (no kangaroos in Austria!).

Note: All authors are responsible for the contents of their respective articles. The articles in MATHEMATIQ solely represent the opinions of the individual authors and not the opinion of Mensa as a whole. Submission of contributions implies agreement to publication in MATHEMATIQ.

Enjoy reading and studying!

Claus D. Volko, cdvolko@gmail.com

Formal Languages

A formal language is a set of words. This set can be either finite or infinite. If a formal language is finite, it can be specified by listing all words that belong to it. This is not possible if it is infinite. How can an infinite formal language be defined? A formalism is required. The type of formalism depends on the type of the language. Unfortunately no formalism is known that would enable one to specify any language. But for special sets of languages there are diverse elegant formalisms.

The sets of formal languages form a hierarchy. This hierarchy is named after the American linguist Noam Chomsky. Higher languages are proper subsets of lower languages in this hierarchy. There are the following relations:

Regular languages
$\subset$ Context-free languages
$\subset$ Context-sensitive languages
$\subset$ Recursive languages
$\subset$ Recursively enumerable languages
$\subset$ General languages,

Recursive langagues
$\subset$ Co-recursively enumerable languages
$\subset$ General languages.

All finite languages are regular. Moreover, languages that allow an unlimited repetition of parts of a word belong to the set of regular languages as well. Regular languages can be defined by regular expressions, that is strings which in addition to literals (characters that belong to the word) may also contain parentheses and two peculiar symbols. One of these symbols is usually expressed as the plus sign, the other as the multiplication sign. The plus sign signifies that the preceding part of the word may be omitted. Due to this, the regular expression *ab+c* stands for the two words *ac* and *abc*. The plus sign is usually only applied to the previous literals, except in case several literals were embraced by a parenthesis, in which case the plus sign relates to the part of the word between the parentheses. The regular expression *a(bc)+d* for example stands for the two words *ad* and *abcd*. With the multiplication sign it is quite similar; its meaning is that the marked part of the word can be repeated an unlimited number of times. The regular expression *a(bc)*d* creates an infinite set of words, containing the words *ad*, *abcd*, *abcbcd*, *abcbcbcd*, *abcbcbcbcd* and infinitely many more.

Some readers might raise the question how this is related to computer science; well, actually formal languages are a central concept of theoretical computer science. Each formal language corresponds to a decision problem: Does a particular word belong to the formal language or not? The belonging to a certain set of formal languages can be decided by a computational model, a theoretical formalism that mimicks the behaviour of a computer. In general any decision problem that can

be solved by a computer can also be solved by a Turing machine and vice versa. However, a Turing machine is not able to decide any conceivable language - it is only suitable for recursively enumerable languages. There is no known formalism for more general languages. I will talk about Turing machines later on, but now let us come back to regular languages.

Regular languages can be modelled as finite automata. An automaton is a set of states with defined transitions. There is exactly one starting state and at least one finishing (accepting) state. Any automaton begins at the starting state and reads the first literal. If there is a transition from the current state to another state with this transition accepting the literal, the transition to the new state can be made. Otherwise execution stops. A word is considered an element of the given formal language if and only if all literals have been accepted in the given order and a finishing state has been reached this way. Note that there may be more than one transition from the current state that accepts a given literal. If there are several different transitions accepting the same input, such an automaton is called non-deterministic. If such an automaton is used to check if a word is in a given language, all possible paths of execution must be considered; it is enough if a single path leads to an accepting state. The other type of automata is called deterministic; with deterministic finite automata, it is sufficient to execute them once to solve a decision problem. As the intelligent reader might suspect, deterministic automata are usually more complex than non-deterministic ones; they consist of more states. Is it possible to construct a deterministic automaton for any decision problem that can be solved by a non-deterministic automaton? Yes, it is. Since a deterministic automaton is actually a special type of a non-deterministic automaton, the opposite relation applies as well. Deterministic and non-deterministic automata have the same strength of expression. In many cases, however, it is easier to construct a non-deterministic finite automaton that accepts a given language.

The next level in the hierarchy is occupied by context-free languages. These languages can be specified by context-free grammars. There are various notations for these, one of the better known ones being the *Extended Backus-Naur-Form (EBNF)*. It has the following syntax:

rule $\rightarrow$ *literal* rule* literal**

Again the multiplication sign means that the preceding element can be repeated an unlimited time, including zero times. This enables one to define rules such as

$A \rightarrow abc$,

which means that any occurence of the rule A may be replaced by the string *abc*, but also rules such as

$A \rightarrow a\ B\ de$,
$B \rightarrow B\ c$,

which mean that B may be replaced by an arbitrary number of literals c and A by literal a, followed by rule B and literals de. What distinguishes this from regular languages is that several possibilities can be defined for each rule. It holds:

$A \rightarrow B|C$

is equal to

$A \rightarrow B$,
$A \rightarrow C$.

So it is possible to choose an option, and this makes it possible to describe languages that cannot be defined by regular expressions. For instance, the context-free grammar

$A \rightarrow a\ A\ b|\epsilon$,

where ϵ signifies the empty word, enables one to form the following words: ϵ, ab, $aabb$, $aaabbb$ etc. There is no regular expression for this language. For this reason this is not a regular, but a context-free language.

Context-free languages can be defined by automata as well. For this purpose pushdown automata are used. These work with a stack, that is a data structure that enables one to push anything onto the top of the stack any time and to derive ("pop") the upmost element of the stack, by which this element is removed from the stack, but not to directly access any other element of the stack. Stacks are also called LIFO memories (*last in, first out*). A pushdown automaton uses the topmost value of the stack as an additional criterion to decide whether a particular transition is allowed. Moreover, each transition may push a new element onto the top of the stack. With such an automaton it is possible to decide whether a given word is an element of the context-free language which is represented by that automaton.

It is easy to show that such an automaton is able to model a context-free language: If the right-hand side of a rule contains only literal, the stack is not needed. If there are references to other rules on the right-hand side, the stack can be used to save where the automaton should continue after processing the rule that is referred to. For instance, if the rule A contains a reference to the rule B, the automaton processes the word following rule A until the reference is reached. Then it saves on the stack where it must continue as soon as the processing of the rule B is finished, and goes on by processing the rule B. Once that is finished, the automaton looks up on the stack to see where it must continue. After finishing the processing of the rule A it realizes that the stack is empty and ends at an accepting state.

What is missing is the proof that such a pushdown automaton is only able to process context-free languages and not also languages that appear in the next level

of the Chomsky hierarchy. Of course it is possible to show that a pushdown automaton is not able to process context-sensitive languages which are not context-free at the same time. Context-sensitive languages can be defined by grammars in which literals may also appear on the left-hand side, for example

$a\ A\ b \rightarrow b\ C\ d$.

I leave the proof to the readers as an exercise. A hint: It is related to the sequential processing of the input (one literal after the other in the very order they appear in the input word). Why may this be a problem with context-sensitive languages? Would it, in theory, be possible using pushdown automata to jump back to literals that have already been processed? Why does this not suffice to define context-sensitive languages by means of pushdown automata?

A formalism that allows to jump back to already processed literals while saving the additional pieces of information needed to process context-sensitive languages ia Turing machines. Turing machines are far more powerful than automata. They can have different states, process the input from left to right as well as in the other direction, and overwrite the input. A Turing machine is represented by states and transitions just like an automaton, exactly one state being the starting state and at least one state being a finishing (accepting) state. The input word is accepted when such a finishing state is reached. Which transitions are possible depends on the current state on the one hand and on the literal located at the current position of the input/output head on the other. Each transition not only defines the following state but also the value the current input data element is overwritten with and the direction where the input/output head will move next.

Turing machines allow to describe more general languages than just context-sensitive ones. Turing machines which have to either accept or reject an input but must not enter an infinite loop are also called Turing deciders. Turing deciders represent recursive (also called decidable) languages. If you allow a Turing machine to enter an infinite loop if the word does not belong to the language but demand from it that it accepts the word in any other case, the set of languages that can be represented is called the set of recursively enumerable or semi-decidable languages. By contrast, if the Turing machine must always reject the word if it is not in the language but may either accept or enter an infinite loop otherwise, these languages are called co-recursively enumerable; and the set of recursive languages is the intersection of recursively enumerable and co-recursively enumerable languages.

Claus D. Volko, cdvolko@gmail.com

Gödel and the Limits of Computability

Kurt Gödel showed in the first half of the 20th century that a formal system in which mathematical and logical statements can be expressed must be either incomplete or inconsistent (First Incompleteness Theorem). Furthermore, he stated that a consequence of a formal system being consistent is that this consistency is not provable within this very formal system (Second Incompleteness Theorem).

Both theorems can be easily deduced using computability theory when considering that completeness basically means that each statement must be decidable and consistency that whenever a mechanism associated with the formal system (e.g. a Turing machine) comes to the conclusion that a statement is provable, this statement must indeed be provable, and when the mechanism comes to the conclusion that a statement is not provable, this statement must indeed not be provable. Paradoxical statements having the property that the assumption that they are provable leads to the conclusion that they cannot be provable and the opposite assumption that they are not provable leads to the conclusion that they must be provable cannot be decided by a consistent system. For this reason consistent systems are not complete, which is equivalent to the statement that complete systems cannot be consistent. The Second Incompleteness Theorem in particular can be shown by arguing that a formal language in the sense of Gödel must be recursively enumerable but must not be recursive in order to be consistent, since recursive languages cannot be consistent as each statement of a recursive language must be decidable and paradoxical statements are not decidable. The problem of deciding whether there is a undecidable statement is itself undecidable.

This is especially relevant to the question of the limits of computability. Gödel himself believed the human mind to be much more powerful than any computer since the human mind is able to compute things not computable by a computer. I believe that it would be interesting in this context to conduct research on detecting paradoxical statements by a Turing machine. Of course the more general problem whether a statement is decidable is undecidable itself since it is equal to the so-called Halting problem, that is the problem whether a Turing machine terminates on any input or whether there is some input which will lead to an infinite loop. However, this concerns decidability in general. Paradoxical statements are a particular subset of the set of statements whose provability cannot be decided. Perhaps it would be possible to program a computer to work correctly at least with this subset.

If the hypothesis that intelligent reasoning is nothing but a form of computation is true, this implies that any computer would be able to reason in a way as intelligent as a human being if anything the human mind is able to compute can also be computed by a computer. Gödel doubted that. It is currently not possible to judge whether Gödel was right, and if he is really right, maybe this will never be provable (as it is infinitely more difficult to prove that something is impossible than to prove that it is possible - cf. the *P-NP* problem).

Claus D. Volko, cdvolko@gmail.com

The P-NP Problem

This being one of the most difficult open problems of computer science, Clay Mathematics Institute has announced that it will pay 1 million dollars to the first person to find the solution.

P and NP are sets of decision problems with particular complexities. That means: There is an algorithm that solves the decision problem whose execution time does not exceed a particular limit with respect to the size of the input data. An algorithm belonging to P will never need more than a polynomial number of steps in terms of the input data size. By contrast NP means that it is possible to verify a candidate solution in polynomial time.

P is a subset of NP. The open question is whether it is a proper subset, or whether the two sets are identical. Most computer scientists believe P is a proper subset of NP, but they have no idea how to prove it.

If the two sets were identical it would suffice to prove it for a single NP-complete problem. The set of NP-complete problems is a subset of NP with the property that any NP-complete problem is at least as hard as any other problem in NP. One of these problems is the Satisfiability Problem of Propositional Logic (SAT, cf. Cook-Levin-Theorem). To show that another particular problem is NP-complete, it is enough to show that it is in NP and that it is at least as hard as SAT. This can be done by showing that it is possible to reduce SAT to this problem, i.e. show that an algorithm for this new problem would enable one to solve instances of SAT and that a SAT solver could be used to solve instances of this new problem. So far, not a single polynomial algorithm has been found for an NP-complete problem.

Probably P is a proper subset of NP, but how should this be shown? In general it seems to be far harder to show that something is not possible than to show that it is possible (*if* it is possible).

From time to time, "proofs" are released, but usually errors are found in them after a short (sometimes longer) while. Interesting enough, some people claim that the P-NP problem is unsolvable. How should this be judged?

First of all the P-NP problem is not a decision problem, so those publications that deem the P-NP problem "undecidable" definitely err - but maybe it is just a matter of terminology. What in theory is possible is to do research on the decidability of a related decision problem, such as the set $NP \setminus P$. If and only if this set is empty, P is equal to NP. I leave it as an exercise to the readers whether $NP \setminus P$ is decidable. In any case, it will not help settle the matter since it is irrelevant: If the set is undecidable, the P-NP problem might be solvable nonetheless, and if it is decidable, this alone will not bring the solution to the P-NP problem.

Claus D. Volko, cdvolko@gmail.com