Algorithmic Thinking: The Key for Understanding Computer Science

Gerald Futschek

Vienna University of Technology
Institute of Software Technology and Interactive Systems
Favoritenstrasse 9, 1040 Vienna, Austria
futschek@ifs.tuwien.ac.at

Abstract. We show that algorithmic thinking is a key ability in informatics that can be developed independently from learning programming. For this purpose we use problems that are not easy to solve but have an easily understandable problem definition. A proper visualization of these problems can help to understand the basic concepts connected with algorithms: correctness, termination, efficiency, determinism, parallelism, etc. The presented examples were used by the author in a pre-university course, they may also be used in secondary schools to help understanding some concepts of computer science.

1 Introduction

In autumn 2005 the Faculty of Informatics at the Vienna University of Technology started to offer a pre-university course [1] (propaedeutic course), similar to other universities, e.g. [2], for all applicants who intended to start one of the bachelor studies in Informatics. This course addresses all beginners and has many-fold reasons:

1. Lack of pre-knowledge, what is Informatics after all
2. Lack of pre-knowledge, how computers work
3. Lack of pre-knowledge about algorithms
4. Lack of pre-knowledge about programming
5. Lack of sufficient knowledge in mathematics

These facts were observed by our lecturers and were confirmed by a survey of our beginners. Although these topics are part of the secondary school curriculum and should be known by all students passing secondary school, most of our beginners have not enough skills and pre-knowledge that is necessary to start a university study in Computer Science. The consequences were a very high drop out rate during the first study year and a low success rate in the topics Programming and Algorithms & Data Structures. The pre-university course should overcome the lack of usual pre-knowledge. Therefore the contents of this propaedeutic course are:

- What is informatics?
- How do computers work?
- Algorithmic thinking
- First steps in programming
- Basics in mathematics

R.T. Mittermeir (Ed.): ISSEP 2006, LNCS 4226, pp. 159 – 168, 2006.

In this contribution we focus on the part on algorithmic thinking and try to answer the question: How can students that cannot program, learn basic facts about algorithms in a very short period of time?

2 What Is Algorithmic Thinking?

Algorithms are defined differently in literature, but for our purpose the following definition is sufficient: “An Algorithm is a method to solve a problem that consists of exactly defined instructions”. Algorithmic thinking is a term that is used very often as one of the most important competences that can be achieved by education in Informatics [3]. Algorithmic thinking is somehow a pool of abilities that are connected to constructing and understanding algorithms:

- the ability to analyze given problems
- the ability to specify a problem precisely
- the ability to find the basic actions that are adequate to the given problem
- the ability to construct a correct algorithm to a given problem using the basic actions
- the ability to think about all possible special and normal cases of a problem
- the ability to improve the efficiency of an algorithm

For me, algorithmic thinking has a strong creative aspect: the construction of new algorithms that solve given problems. If someone wants to do this he needs the ability of algorithmic thinking.

3 How to Teach Algorithmic Thinking?

This question is as hard to answer as “How to teach creativity?” A practical answer can be: try to solve many problems. Especially for beginners in Informatics these problems should be chosen very carefully. In my opinion these problems should be solvable independent from a specific programming language. Especially beginners have many problems to understand the underlying programming language properly, so that they cannot concentrate additionally on the design of a new algorithm. The language to describe the algorithm should be high-level and problem-oriented, e.g. pseudo code fulfills these criteria.

The problems to be solved should be not too simple, but the problem statement should be easily understandable. More complex problems give more space to creativity and to the students individual, sometimes unusual solutions. As is shown in the following examples pre-knowledge of a programming language is not necessary.

Very useful and powerful tools are visualizations of algorithms, e.g. [4]. These tools help to understand algorithms. The possibility to construct easily different input values allows to play with different variations of inputs and allows to study normal and extreme cases. It gives also a feeling why an algorithm works and how an algorithm may be improved. We used in our course a tool, called Theseus, see [1], written by the student Marian Kogler. This tool is able to produce and manipulate mazes and it allows to apply different path-searching algorithms.

4 Example: Paths in Mazes

We want to show with this example that it is possible to gain first insight in problem analysis, algorithm design and effort analysis without prior knowledge of computer programming.

4.1 Problem Analysis

Finding a path in a maze (labyrinth) is a classical task and is not trivial. Usual tasks are:

Find a path out of a maze,
Find a path through a maze, or
Find a path to a specific position inside a maze.

What is to do seems easily understandable, but it is not evident how to do it. A first step in solving the tasks is the analysis of the problem. We can find out that the mentioned tasks have a common generalization:

Find a path from position A to position B

It is also important to know what we know about the maze while we search for position B. In the simplest scenario we know nothing about size and structure of the maze. We can walk on corridors and we can see until the next crossing. At a crossing we can see all the different corridors starting from this crossing. We can recognize the goal B, but we do not know the direction to the goal B. If we reach a crossing that looks like a crossing already visited, we cannot decide if it is the same crossing or a new one.

While walking in a maze, we have to decide at each crossing which corridor to go. Possible strategies are:

Follow the leftmost corridor
Follow a random corridor
Follow systematically all corridors, beginning with the leftmost

We now discuss these three strategies in detail.

4.2 A First Solution Strategy

A well-known strategy to find a path through a maze is:

```
follow the left wall
```

It is not necessarily clear that we find a way through a maze with this strategy. In the case that the starting point A is an entrance of the maze it can be shown that this strategy finds an exit. Reasoning about this could be a good exercise in algorithmic thinking. It should also be considered the case that the maze has no exit at all.

Mazes can differ in size, form, and length of the corridors and crossings, mazes can be planar or even 3-dimensional. An appropriate model of a maze shows only its relevant characteristics.

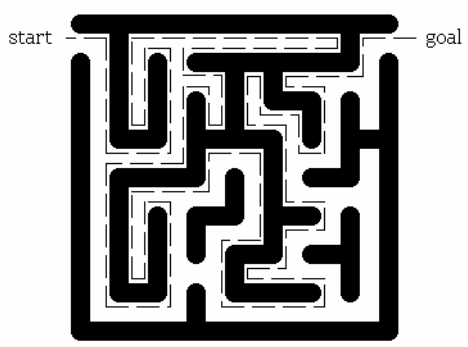

Fig. 1. The Left Wall Algorithm. After entering the maze we follow the left wall. This left wall is connected to (a part of) the surrounding wall. Therefore we will reach an exit of the maze using this strategy. If we exit at the entrance then we know that there exists no other exit in the maze.

4.3 A Model of a Maze

Modeling is abstraction. From a maze we can abstract the length and form of the corridors. The only thing we have to know is: which crossings are connected with which other crossings. An adequate model of a maze is a graph, its nodes are the crossings and its edges are the corridors.

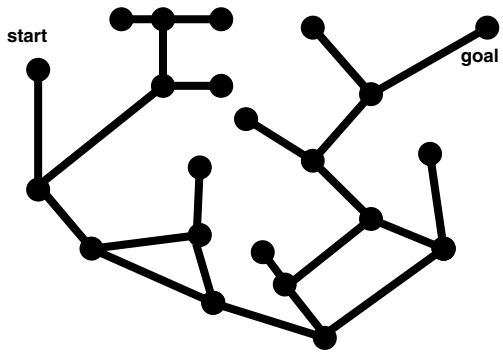

Fig. 2. The Maze of Fig. 1 represented as graph consisting of nodes (crossings) and edges (corridors). The length, form, and direction of the corridors are abstracted, they are not relevant in the graph. A graph just represents the neighborhood relation of crossings. Which pairs of crossings are connected by a corridor (an edge exists) and which pairs of crossings are not connected (no edge exists).

In the left wall algorithm we need to know for each node that can be reached which one of the edges around this node is the leftmost edge. In order to be able to define the leftmost edge of a node, we can define a circular ordering of the edges around each node. Each edge i around a node has in this ordering a successing edge succ(i). When

reaching a node on an edge e, then the leftmost edge is succ(e) according to this circular ordering. In a planar maze there exists a natural circular ordering of the edges according to their direction in the plane. Please note that the leftmost edge of a node is not a fixed edge, the leftmost edge depends on the edge e that reaches the node.

Since a normal graph as an abstract model of a maze does not represent a direction or ordering of edges we have to explicitly and additionally define the circular ordering of the edges around each node. A graph with such a circular ordering of edges is also known as "embedded graph".

4.4 Basic Actions of a Maze-Algorithm

To describe algorithms we need to know the actions that are the elementary instructions that can be used. In this maze example we define the basic actions at a high level (much higher than that of programming languages). So we hope that the semantics of the algorithm will be easier to understand.

We define 4 basic actions that can be performed when a node (crossing) is reached by an edge e:

action 1. Query: Is the goal reached?
action 2. Determine the number of edges (corridors) that leave this node.
(If this number is 0, then a dead end is reached)
action 3. Follow the i^{th} edge and go to the connected node (crossing).
(The 1^{st} edge is called the leftmost edge)
action 4. (Turn around 180 degrees and) go back to previous node,
following edge e.

Using these 4 basic actions we will formulate the algorithms to find paths in mazes.

"Follow the i^{th} edge" in action 3 means: after entering a node on edge e the edge succ(… succ(succ(e))…) has to be followed, where the circular ordering succ is applied i times. The 1^{st} edge is defined as the leftmost edge and can be determined by succ(e). The number of basic actions can be reduced by one since basic action 4 can be expressed by basic action 3: Follow the 0^{th} edge, which is in fact edge e. We think that this special case of action 3 is easier understood in an extra basic action.

4.5 A First Algorithm

Using the 4 basic actions we can formulate the left wall strategy in the following pseudo code algorithm:

```
while goal not reached                          (action 1)
   if dead end                                  (action 2)
   then turn around (180 degrees)and
        return to previous node                 (action 4)
   else follow the leftmost edge                (action 3)
```

For this algorithm we need to give a precondition that guarantees termination and the existence of the leftmost edge:

precondition: the goal is reachable (to guarantee termination) and there exists a starting edge (to be able to determine the leftmost edge at the first node)

This simple left wall algorithm shows already the algorithmic concept of *back-tracking* while returning in dead ends. This algorithm is very simple because there are no additional tools necessary and there are also no visited nodes to remember during execution. While trying out this algorithm with several mazes and with several start- and end-positions, we find out that this algorithm sometimes does not terminate.

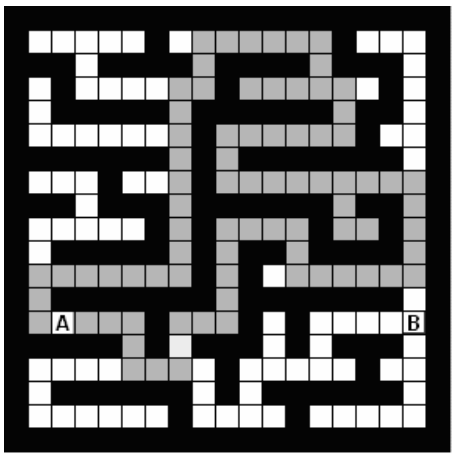

Fig. 3. If the maze contains cycles the algorithm may be trapped in a cyclic path forever, while following the leftmost path. The algorithm does not terminate in these cases.

The left wall strategy always terminates when it is applied to mazes without cycles and may not terminate when it is applied to mazes with cycles.

4.6 Random Paths

How walking in cycles can be avoided? One possible observation is: It seems that the left wall algorithm runs endless because at each crossing always the leftmost path is selected to follow. Why not choose randomly at all crossings a path to go? Now a tool to perform this random decision is needed, a dice or a random number generator. Using this tool at each crossing, an arbitrary path can be selected. Every time the algorithm starts from the same starting point again possibly other paths are used to find the goal. This is a property of non-deterministic algorithms. Sometimes the goal is reached with a small number of steps, sometimes it needs many steps to reach the goal. The question arises: Does this algorithm always terminate? Is it possible that by random algorithm always those paths are chosen that don't reach the goal? All experiments with different mazes and different start- and end-positions show that the algorithm always terminates when the goal is reachable at all. Is this a proof that the random path algorithm always terminates? No. These observations are not a proof. A proof needs some scientific insight into probability theory. But, anyway, it is important to raise such questions to motivate students to study the necessary theories.

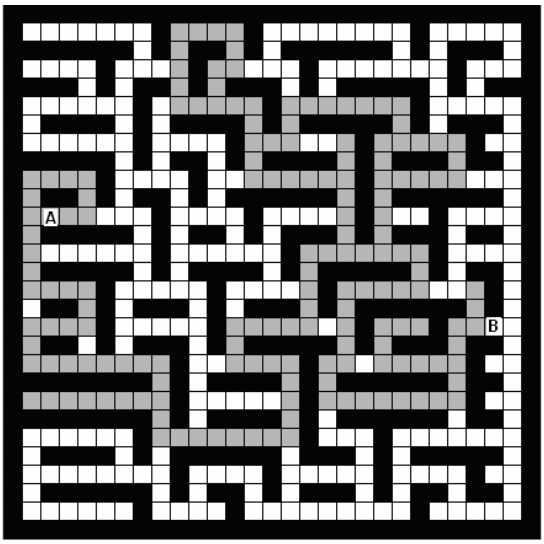

Fig. 4. Random Path Algorithm. A random walk does not search systematically. Each start of the algorithm generates another path. This algorithm is not deterministic. Each reachable position of the maze can be reached in a finite number of steps. Already visited positions may be visited many times again and again. There is no upper bound of the number of steps needed to reach an end-position.

4.7 Using Ariadne's Thread

Another classical tool to avoid cycles is the famous "Ariadne's Thread" that was used by Theseus to find and kill Minotaur in his maze at Knossos. It is the eldest known back-tracking algorithm. The thread shows the path from the start-position to the current position. The basic actions 3 and 4 have to be modified: while walking on new paths the thread is unwound, while back-tracking an unsuccessful path the thread is wound up again. A basic action to find Ariadnes thread has to be added. Every time Theseus finds his thread again he is sure that he found a cycle and returns to the last crossing in the same way as he does at dead ends. If Theseus chooses the leftmost path first, the pseudocode of Ariadne's Algorithm can be formulated very similar to the Left Wall Algorithm:

```
while goal not reached
   if (dead end) or (Ariadne's thread is found)
   then turn around (180 degrees) and return to previous node
   else follow the leftmost edge
```

Notice that the leftmost edge depends on the edge one is coming from. To find the goal the edges of a node are tried out in the sequence of the circular ordering. When returning from a not successful edge, the leftmost edge is the next edge in the circular ordering to be explored. While performing Aridne's Thread Algorithm with several mazes and several start- and end-positions we can observe that this algorithm finds sometimes the end-position very fast. But sometimes it takes very long and sometimes

it takes extremely long (many hours). It can be observed that this algorithm tries very systematically all possible combination of edges. It can be discussed how many combinations of edges are necessary to find all paths from A to B. A rough estimation shows a combinatorial explosion depending on the degree of the nodes and the distance from the starting point. This fact is a good motivation to improve this algorithm …

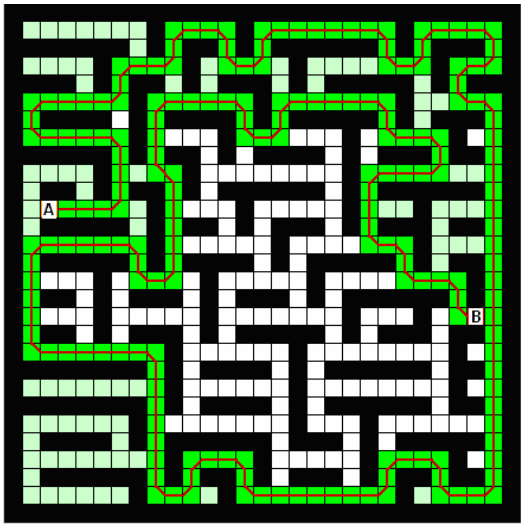

Fig. 5. Ariadne's Thread Algorithm. The thread avoids walking in cycles. The Algorithm produces systematically all possible acyclic paths from the start-position. It stops when the end-position is reached.

There are still many questions that can be discussed:
- What is the necessary length of the thread?
- What is the effect of the algorithm if the goal is not reachable?
- How we can improve this algorithm?
- How can we find the shortest paths?
- etc.

The maze-problem is an interesting starting point to discuss many variants of maze algorithms and is an ideal tool to give first insight in different aspects of algorithms.

5 Example: Parallel Sorting

Sorting of a series of values can be played by the students themselves. To be an active part of an algorithm is very motivating. Each student holds a value that can be seen by all the other students. At the beginning the students stand in a row at arbitrary positions. The task of finding a sorting algorithm consists of defining precise instructions to the students what they have to do. The students are told just to follow

the instructions of the algorithm and they are not allowed to do something else. The first approximation to a sorting algorithm is very often the following instruction to all students in the row:

```
repeat forever:
   exchange your position with a neighbor, whose
   value is not in correct order compared to your value
```

This first algorithm shows the idea of the algorithms very well, but it is not precise enough to be executed properly. There can raise the following problem: both neighbors of a student want to change with him. What should he do? Also the following question cannot be answered exactly at any time: Who is a neighbor? The values change their positions very often. If a neighbor starts changing his position with his other neighbor. Is he still a neighbor? Which one of the two exchanging values is the neighbor?

A more precise definition is necessary. We can solve the questions with *synchronization*: the two exchanging persons shake their right hands while exchanging their positions. The students are free for exchange if they have their right hand free. An improved algorithm with synchronization may look like this:

```
repeat forever:
   look at your left neighbor.
   if (he is free) and (his value is less than your value)
   then synchronize and exchange your position
        with this neighbor
```

This play of a sorting algorithm works very well with a group of students. This algorithm is also very robust: Even if a value is changed from outside or an additional value is inserted, this value will be rearranged until it is on its final position.

An analysis of this algorithm is also very easy: the maximum number of exchanges of a specific value is n-1. Because all persons act independently at their own personal speed, it is not possible to tell the number of steps of this parallel program. This is a perfect program to show the concept of *software agents*. But there is an unsolved point: as the parallel agents will act forever, it is not clear at what time the row is sorted!

We can observe that all odd positions can exchange at the same time or all even positions can do this at the same time. The odd and the even exchanges can be clocked so that they take place at the same time. Furthermore we need at most n parallel exchanges. With this observation we can formulate a terminating algorithm.

Parallel exchanges at odd and even positions take place alternately:

```
repeat n/2 times:
   all even positions perform in parallel:
      look at your left neighbor
      if his value is less than your value
      then exchange your position with this neighbor
   all odd positions perform in parallel:
      look at your left neighbor
      if his value is less than your value
      then exchange your position with this neighbor
```

We know that we need at most n parallel actions, so the order of this sorting algorithm is linear. In fact this is the most efficient algorithm with the restriction that

all persons exchange with their neighbors. Studying computer science will show that there exist much faster parallel sorting algorithms, but all of them exchange positions over longer distance.

6 Summary

Our experience with students shows that a first introduction to algorithms can be learned at a problem oriented level independent from any programming language. A wide series of informatics relevant topics can be addressed and learned by solving interesting and encouraging problems. These problems should be well-chosen. A visualization of the algorithms in form of a program-tool or of an activity play performed by the students themselves can be very helpful. Beginners like to play with algorithms while feeding them with different input values to get an impression how the algorithms work and what problems arise, e.g. non-termination or combinatorial explosion. To improve or modify algorithms abstract models and precise definitions of properties are necessary.

References

1. Propädeutikum in Informatik (prolog), Lessons held at Vienna University of Technology: www.informatik.tuwien.ac.at/prolog
2. Loidl, S., Mühlbacher, J, Schauer, H.: Preparatory Knowledge: Propaedeutic in Informatics. In Mittermeir, R.T. (ed.): From Computer Literacy to Informatics Fundamentals. Lecture Notes in Computer Science, Vol. 3422. Springer-Verlag, Berlin Heidelberg New York (2005) 104-115
3. Snyder, L. Interview by F. Olsen "Computer Scientist Says all Students Should Learn to Think 'Algorithmically'," *The Chronicle of Higher Education*, May 5, 2000: http://chronicle.com/free/2000/03/2000032201t.htm/
4. Mudner, T., Shakshuki, E.: A new Approach to Learning Algorithms. In Proceedings of International Conference on Information Technology: Coding and Computing. (2004) 141-145