Metaphors and Analogies for Teaching Algorithms*

Michal Forišek
Department of Computer Science
Comenius University, Bratislava, Slovakia
forisek@dcs.fmph.uniba.sk

Monika Steinová
Department of Computer Science
ETH Zurich, Switzerland
monika.steinova@inf.ethz.ch

ABSTRACT

In this paper we explore the topic of using metaphors and analogies in teaching algorithms. We argue their importance in the teaching process. We present a selection of metaphors we successfully used when teaching algorithms to secondary school students. We also discuss the suitability of several commonly used metaphors, and in several cases we show significant weaknesses of these metaphors.

Categories and Subject Descriptors

K.3.2 [**Computer Science Education**]: Algorithms

General Terms

Algorithms

1. INTRODUCTION

Metaphors and analogies are an important tool in learning unfamiliar concepts. Many authors agree that they can be used not only to communicate the relevant properties of the new concept, but also to facilitate developing new conceptual structures – new mental models and new abstractions [13]. (Kindly refer to the appendix for an explanation of differences between metaphors and analogies.) Abstraction of knowledge is, in some sense, the holy grail of both research and education. A famous example: Among chicken, often a "stronger" chicken pecks a "weaker" one, but not vice versa. Biologists observing chicken groups discovered that this is not always a transitive relation. Surprisingly, it was still always possible to find a linear "pecking order" in which each chicken pecked the next one. The surprise ended with Landau's proof [18] that every tournament graph contains a Hamiltonian path – the existence of a pecking order is a mathematical certainty, not a biological property.

For teaching programming, [5] empirically shows that good explanatory analogies, characterized by clarity and a systematic approach, positively influence all factors, including program comprehension, program composition, and time needed for the given tasks. Metaphors in programming and Computer Science (CS) were also presented and analyzed in [27, 7, 28]. Many nice metaphors and analogies useful for teaching CS to young kids can be found in [2, 29].

Metaphors and analogies can be used in all levels of education, starting with the complete basics. For instance, it is well known that a "labeled box" metaphor for a variable in a program helps explain the potentially confusing "x=x+1" statement. But there is no real upper bound on subject difficulty – metaphors are also used in graduate courses at universities. For instance, amortized time complexity is often explained using the physical metaphor of potential energy [8], and zero-knowledge proofs can be illuminated using the beautiful Ali Baba's cave metaphor [24].

One caveat when using metaphors and analogies in education is that they are very sensitive to cross-cultural aspects [12, 16], and also to cross-gender aspects – e.g., "car analogies" are commonly used in engineering courses and textbooks such as [6]. These tend to be less suitable for female students. Another source of difficulties with metaphors is that the target never matches the source perfectly. The inherent danger is that a subject will use the metaphor to infer incorrect conclusions about the target, and additionally have high confidence in these conclusions. This danger must be foreseen and mitigated by the educator. In Section 7 of this paper we identify several false and/or misleading metaphors and analogies that are widely used in practice.

The danger of flawed metaphors cannot be avoided completely. Spolsky [25] quotes several prominent cases where simple abstractions fail in unexpected ways. One of the examples is iterating over a simple 2D array (a table). The array is a mathematical abstraction that allows us to visualize the operations done with the actual data in memory. When working just with this abstraction it seems that iterating in row major order should be the same as in column major order – either way we process each element once. But in practice this is not the case: row major order is usually a order of magnitude faster due to caching. (It is faster to process data in the order in which it is stored in physical memory.) Spolsky concludes that becoming a proficient programmer is getting harder and harder, even though we have higher and higher level programming tools with better and better abstractions.

Many scientists, famously including Dijkstra in [11], warn against misleading metaphors. Dijkstra explains that CS represents a major breakthrough, hence often "the analogies

*The research is partially supported by Hasler Stiftung.

SIGCSE'12, February 29 – March 3, 2012, Raleigh, North Carolina, USA.

become too shallow, and the metaphors become more misleading than illuminating". Still, nowadays we already have many good metaphors in CS. Their existence can be seen as a measure of our scientific progress – we already managed to fill in many of the gaps perceived by Dijkstra. In fact, one of our results is a good metaphor we successfully used to explain Dijkstra's very own shortest paths algorithm.

We also agree with Dijkstra in that we are convinced that a rigorous formal approach to CS can never be replaced by simply inferring from the metaphors. The metaphors can only provide intuition and insight that makes formal reasoning easier. Dijkstra also argues that anthropomorphic metaphors (e.g., programs "knowing" or "wanting" things) should be avoided at all costs. His argument is that adoption of these metaphors prevents a departure from operational semantics – the programmer is forced to think about the flow of the computational process, without the possibility of an abstraction. In the example scenario given by Dijkstra such problems are plausible indeed. We are not using any anthropomorphic metaphors in our paper.

Paper outline. The paper is organized as follows: Section 2 contains several new metaphors for simple algorithmic concepts. An explanation of Dijkstra's algorithm for shortest paths via a physical metaphor is contained in Section 3. In Sections 4 and 5 we give metaphors for a few geometry problems and for the Knuth-Morris-Pratt substring search algorithm, respectively. In Section 6 we describe our experience with using our new metaphors in education. The paper is concluded by Section 7 where widely used false or misleading concepts are discussed. In Appendix A we define and explain the terminology used in this paper.

2. METAPHORS FOR SIMPLE CONCEPTS

Here we illustrate our intended usage of metaphors on simple examples that do not deserve a separate section. Still, all of these examples are meaningful and in our experience their proper use significantly improves the teaching process.

The stack. When explaining the data structure *stack* (LIFO memory – last in, first out) we realized that many of the students already have a deep understanding of the same principle. In Slovakia ice cream is customarily served as shown in the figure on the right. The vendor takes an empty cone and adds the scoops as they are requested. However, such a cone needs to be eaten from top to bottom, hence to eat the flavors in the desired order, kids are required to place the order in reverse.

(Note that this metaphor is culturally biased, as it only works in countries where kids have the prior experience. Just explaining it with pictures tends to be useless. Also note that the metaphor can be extended to less trivial stack operations, for instance the cone offers "read access" to all its elements, just like a stack stored in an array.)

The queue. The most common metaphor for a queue is the checkout line in a supermarket. This metaphor can be found in many textbooks, for instance [15]. While this metaphor is nice to introduce the "first in, first out" principle, it is woefully inadequate when it comes to an actual implementation of the queue and its time complexity – based on the way actual queues work, the student is coaxed into thinking that it is necessary to move all elements each time we process the first one. Therefore we prefer to follow this metaphor by a better one. In practice we can easily find examples of smarter queuing. For instance, there are waiting rooms with a machine that dispenses tickets with numbers. You take a ticket, sit down anywhere and wait until your number comes up. This corresponds to a queue implemented in an array. (Actually, a cyclic array. The tickets commonly reuse numbers 000 to 999 repeatedly.)

There are also other examples of smart queuing. For instance, in some cultures the doctor's office has a waiting room that serves as a queue. Upon entering the room you just ask who is the last in line, remember them and then take any available seat. When the person before you gets called into the office, you are the next one to go in. This closely resembles the linked list implementation of a queue.

Alternately, we sometimes present the correct queue implementation as a "think out of the box" puzzle after the checkout line metaphor. The solution: instead of moving everyone in the checkout line, move the cash register.

Topological sorting. Our favorite metaphor for the topological sorting problem is *getting dressed*. There are many non-trivial dependencies between various pieces of clothing. Their graph can easily be constructed by students and it is a good example for a test run of the algorithm. The intuitive approach "put on anything you already can" actually translates to a correct algorithm.

3. DIJKSTRA'S ALGORITHM

In this section we present a metaphor that helps our students (including secondary school students) create a deep mental model of Dijkstra's shortest path algorithm.

The input for Dijkstra's algorithm is a graph with positive edge lengths and one marked vertex s. The algorithm computes the shortest paths from s to each other vertex of the graph. We will now give its brief overview, for more details see [8, 10]. The algorithm proceeds as follows: Let the *distance value* of vertex v be the smallest known distance from the vertex s to v. These values are stored in the array D; the value $D[v]$ is updated whenever a smaller distance to v is discovered. Initially, all $D[v]$ are infinite except for $D[s] = 0$. Each vertex v is in one of two states: visited (meaning that $D[v]$ is final) or unvisited. Initially, all vertices are unvisited. The algorithm then repeats the following steps:

1. Let u be an unvisited vertex with the smallest distance value, i.e., the closest unvisited vertex to the vertex s.
2. The vertex u becomes visited and thus $D[u]$ is final.
3. For each edge uv: update $D[v]$ if the path via u to v is shorter. (Its length is $D[u]$ + length of uv.)
4. If there are unvisited vertices left, continue with step 1.

There is a well-known metaphor related to the shortest path problem: the balls-and-string metaphor. Each vertex of the graph corresponds to a small ball. The balls are connected by strings with lengths proportional to edge lengths. To find the shortest path between s and t, one just grabs the corresponding two balls and tries to pull them apart. This process stops when some strings between the balls become stretched. These strings then form the shortest path. One textbook mentioning this metaphor is [10], but the metaphor is not actively used in algorithm presentation.

We will show how to augment the metaphor by adding the aspect of gravity. This will intuitively model all necessary properties of the algorithm. Similar insights can then lead to new algorithm design, as in [22]. The way we use our new metaphor can be summarized as follows:

If we take the balls-and-strings gadget and let it hang by the ball s, ball depths correspond to their shortest distances

from the ball s. Dijkstra's algorithm computes their depths by discovering and processing the balls from top to bottom, while it builds the gadget on the fly.

Now in more detail: Imagine the gadget hanging by the ball s. Due to gravity, some strings from s to other balls become stretched simultaneously. The strings with no slack form precisely all shortest paths from s to other vertices. To find the shortest distances from s, it is sufficient to determine how the gadget looks like in its stable state, hanging by the ball s. It can now be shown that Dijkstra's algorithm corresponds to examining this gadget from top to bottom. For added clarity, we will *build* the gadget, adding balls and strings as we go, focusing on its behavior thanks to gravity.

We start by fixating the single ball s (with no strings attached yet) on the top of a vertical wall. This matches the initialization of Dijkstra's algorithm. We then process the gadget from top to bottom. Whenever we encounter a ball, we add all its remaining strings (and the balls on their other end, if necessary) to the gadget. Remember that string lengths are proportional to edge lengths.

By observing the physical properties of the gadgets we can make one important observation: Imagine that you are adding a piece of string between two balls. Obviously, this can never pull a ball *further downwards*. The only two cases: either the string is long enough and the balls remain where they were, or the string pulls the deeper ball upwards. This intuition is reflected in the algorithm – processing edges in step 3 can only decrease the distances in the array D. (Note that at any moment the values in D directly correspond to current ball depths, measured from the top of the wall.)

The other observation we need: adding a string somewhere *does not influence anything above it*. Once again, this is obvious thanks to intuition from physics. How does this property help us here? Whenever we encounter a ball, we know that the unfinished part of the gadget currently lies below the ball only – and therefore this ball is in its final position. (This corresponds to the step 2 of the algorithm. Adding the strings from this ball is step 3.)

4. COMPUTATIONAL GEOMETRY

Thanks to the close connection between geometry and physics, many involved geometric concepts can be explained more easily when appealing to intuition from physics. One classic metaphor commonly used in computational geometry is "rubber band as a two-dimensional convex hull": Imagine the points as pegs on a board and take a closed rubber band that contains them all. If you let the rubber band go, it will contract until it hits some of the pegs, eventually stabilizing itself as the convex hull of the point set. Rubber bands are an excellent metaphor, because from real life experience we know how they behave: they contract whenever possible. We discovered that many proofs and algorithms in computational geometry can be stated in terms of local optimizations, hence the rubber band metaphor can be used.

Shortest path with obstacles. As a first example, consider a two-dimensional shortest path problem in a plane with obstacles. (This is a very important problem with applications in motion planning, see chapter 13 in [3].) The number of possible paths is infinite, even uncountable. The algorithms for this problem are based on reducing the possible paths to a finite set of candidates. To do so, one has to prove that the shortest path always consists of several special segments, each segment being either a straight line or copying the boundary of an obstacle.

A formal proof of this claim can be based on local optimizations: if we can locally improve a path, it is not the shortest one. The intuition behind this proof is captured by a rubber band metaphor. Imagine that you are looking for the shortest path from point A to point B. We will show that the shortest path must have the shape described above. To see why, take a rubber band, fix one of its ends at A and, keeping the other end in hand, walk to B using any path you like. Of course, the rubber band will try to contract. Once you reach B, it will be at most as long as your path was, and it will have the desired shape.

Distance between line segments. One of the basic problems in computational geometry is to determine the shortest straight-line distance between two line segments. In two dimensions this problem is simple, as there are only three distinct cases to consider. However, the problem becomes much more involved in three dimensions and students often miss some of the cases. Notably, in one tricky case the shortest path does not contain any segment endpoint – it is a line segment orthogonal to both given segments.

With rubber band intuition, the three-dimensional case becomes no harder than the two-dimensional one. Imagine each line segment as a railroad track. Place one tiny railroad car on each segment and connect them by a rubber band. The rubber band will try to contract, thereby pulling the cars along the tracks. And while the rubber band pulls them, their distance is decreasing. Hence in the optimal configuration, when the cars are closest to each other, none of them can be pulled any further. This means that each railroad car is either stuck at the end of its segment, or the rubber band is perpendicular to that car's segment.

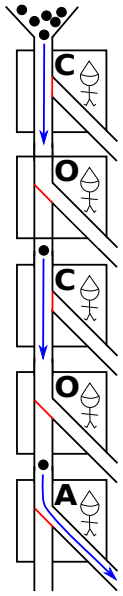
Polygon triangulation. Our last example contains a combination of two metaphors: gravity and our favorite rubber band. We will show an intuitive proof of an involved geometric theorem with many practical consequences: every polygon completely contains one of its diagonals (and hence has a triangulation). In addition to being easy to visualize, our proof is constructive and leads to an efficient algorithm.

To prove the claim, place the polygon on a vertical wall. The polygon has at least one convex angle. Pick one and rotate the polygon so that both sides of this angle point downwards and no edge is horizontal. Take a lead ball on the end of a rubber band, fix the other end of the band to our vertex and drop the ball. Clearly, it will fall straight down until it hits a side of the polygon, and then slide along the side until it reaches a vertex. At this moment, look at the rubber band. If it is straight, it is the diagonal we are looking for. Otherwise, start at the initial vertex and look for the first obstacle it hits. This has to be some other vertex, and the rubber band piece between them is the diagonal.

5. KNUTH-MORRIS-PRATT

The Knuth-Morris-Pratt algorithm (KMP, see chapter 32 in [8]) is one of the most frequently used linear time algorithms for the task of locating a given substring N (the "needle") in a longer string H (the "haystack"). We will denote the lengths of these strings n and h. We developed a physical metaphor for the algorithm that both illuminates how the algorithm works and simplifies its bug-free implementation. Additionally, our metaphor is also useful when analyzing the time complexity of the algorithm.

We will assemble a large gadget consisting of balls falling down a series of tubes. Conceptually, the gadget resembles a finite automaton. The gadget will have n blocks – one containing each letter of N, from top to bottom. An example for the word $N =$ `COCOA` is shown on the right. Each block contains a switch. If the switch is on, a ball coming from above will continue downwards, as in the first and third block in the figure; otherwise, the ball will be redirected to drop out of the gadget. The switch is controlled by a gnome. Whenever he hears his letter, he turns the switch on, and whenever he hears any other letter, he turns it off.



Our time units will be called *ticks*. In a single tick each ball travels through one block of the gadget. To look for the string N in a given string H, we will use our gadget as follows: Once per tick we read aloud the next letter of H and drop a new ball into the gadget. The letter is always heard by all the gnomes.

The figure shows the state of the gadget immediately after reading the last letter of the string `OCOCOC`. Switches are already set according to the last `C`. During the next tick, three balls will travel as shown by the arrows.

Here is the summary of why this is a suitable metaphor for KMP: *Simulating the entire gadget corresponds to the naive string matching algorithm. KMP then achieves speedup by only keeping track of the number of blocks cleared by the currently bottommost ball.*

For the naive string matching, just note that each occurrence of N in H corresponds to a ball successfully traversing the entire gadget and dropping out the bottom after the last letter of the occurrence was read. Then, the core idea of KMP is that we just need to follow the bottommost ball – the one closest to success. With our metaphor it is easy to prove that we do not lose any information by doing so: If we know that the lowest ball b already passed k blocks, we can determine the last k letters we read from H. These uniquely determine the paths taken by all balls thrown in after b.

To make the simulation efficient, we need one more piece of information. What if the bottommost ball drops out of the machine? The next ball above it now becomes the bottommost one. Thus we need to precompute the information "if the bottommost ball is here, where is the one above it?" This is precisely the "failure function" precomputed during the first phase of KMP, and it is closely related to the backlinks in a deterministic finite automaton for locating N.

Finally, the metaphor can illustrate the non-trivial proof of KMP's time complexity: our entire simulation consists only of two types of steps: "move the bottommost ball" and "jump to the next ball above", and each of these can be executed at most h times.

6. PRACTICAL EXPERIENCE

The metaphors presented in Sections 2 to 5 of this paper were used in teaching various groups of students in years 2000 through 2010. In most of the cases, the students' age group was 15 to 19 and the algorithm lectures were given by the authors as a part of an extra-curricular activity for talented kids interested in CS. In all cases, we analyzed whether and/or how using these metaphors helped our students. We observed that the explanations that were accompanied by suitable metaphors were much more successful than ones given without metaphors. With our metaphors we usually saw deeper understanding of the topic in most students. The main way in which this was observed was by asking them to *think* in terms of the newly acquired knowledge. Below, we give concrete accounts how this thinking was directly influenced by the presence of the metaphor in various cases.

For Dijkstra's shortest path problem, we followed the presentation by a series of questions that tested the students' understanding. Here are several of these questions: "What happens to the shortest path tree in a graph if we add another edge to the graph? How would you recompute the new tree? Why does the algorithm require edge lengths to be non-negative? Can you find an instance with a negative edge where the algorithm fails?" Many of these questions are non-trivial; in our experience even university students who passed an introductory algorithm course often have trouble answering them. Our students, having the physical metaphor, were very successful in answering these questions.

Here are examples of similar questions for the KMP substring search: "How is the failure function related to the periods of the substring (the 'needle')? What is the time complexity? Can you prove it?" Additionally, having the metaphor, many of the students were able to independently discover the Aho-Corasick algorithm – a generalization of KMP that searches for multiple patterns at once.

With KMP we observed one additional, very significant difference. Implementations of this algorithm are notoriously prone to off-by-one errors. Before we introduced the metaphor, we routinely observed these in implementations by our students. However, in implementations by students knowing the metaphor these errors are virtually non-existent.

We also note that our KMP metaphor can be easily introduced as a kinaesthetic activity in which each student plays the role of one of the gnomes.

7. MISLEADING METAPHORS

In daily computer usage we can find many metaphors that are confusing for many users. For years, in MacOS a common way to eject a CD-ROM drive was to drag its icon to Trash. The "desktop" and "trash" metaphors were so strong that many users were anxious about losing their data when "dragging the CD to trash" [14]. Stephenson [26] gives another example, so common that proficient computer users do not even realize that it is a metaphor: the words "document" and "save". When we document something in the real world, we make fixed, permanent, immutable records of it – but computer documents are mutable. And if we "save" something in real life, we protect it from harm. But every time you hit "save" on your computer, you annihilate the previous version of the document – the one you supposedly "saved" before. In our research we focused on flawed metaphors that are used both in teaching and in designing algorithms. For this paper, we identified multiple such cases.

Queue. We already mentioned one in Section 2: the "checkout line" metaphor happily used to illustrate the queue. As we explained above, this concept should only be used as an analogy to the FIFO principle. It is not a suitable metaphor, because it leads to incorrect reasoning about the time complexity of operations with the data structure.

Russian dolls ("Matryoshkas"). These nesting dolls are commonly used as metaphors for two distinct concepts: for recursion and for nesting of data structures [9]. In our

experience both usages turned out to be problematic. With recursion, the main caveat is that each of the dolls always directly contains at most one smaller doll – this sometimes caused false impression that a function can only make one recursive call. With nested data structures, the main misconception arising from the metaphor was that the structures have to be of the same type.

Soap and Steiner trees. The last two topics in this section have very much in common. In both of them we document situations where researchers draw false analogies between physical experiments and CS concepts. In some cases their readers then misinterpret those analogies as metaphors, which leads them to wildly inaccurate conclusions about the difficulty of problems solved by computer scientists.

In a classic science experiment two parallel glass plates with pegs between them are dipped into soapy water. After the plates are taken out of the water, a soap film will form between the pegs. Once it reaches a stable state, we will see a soap film network connecting all the pegs. This stable configuration of the soap film has properties similar to the optimal solution for the *Steiner tree problem*. This is a well-known NP-hard problem. Given a set of points in the plane (pegs), their Steiner tree is the shortest collection of line segments that connects all of them. Note that this problem differs from the spanning tree problem, as here we are allowed to create new points where the segments meet.

Due to mistakenly identifying the soap film contraction with solving the Steiner tree problem, several researchers published this claim in their papers. There even was a paper[1] claiming that P=NP and "proving" it via soap bubbles.

It is true that to minimize surface tension, the soap film always contracts into a shape resembling the optimal Steiner tree. However, soap contraction is *not* a suitable metaphor for solving the Steiner tree problem. As shown by Aaronson in [1], the final configuration of the soap is just a local minimum of the energy function, not necessarily the global one. The behavior of soap film should just be used as an analogy to explain one heuristic approach to approximating the Steiner tree. However, there are more suitable metaphors for such approximation algorithms, e.g., "hill climbing".

Animals "solving" hard problems. Simulating the behavior of an animal colony is a frequently used and well researched meta-heuristic for designing approximation algorithms that tend to perform reasonably well for some practical instances of hard optimization problems [21]. Taking inspiration from nature is surely a good idea. However, we noticed that too often the readers (and sometimes also the researchers) interpret the behavior of the animal colony in question as an actual metaphor for the computation, which often leads to incorrect conclusions.

In a recent publication by Reid et al. [23] the authors transform an instance of a classical three disc Towers of Hanoi problem to a physical maze in which the paths from start to exit correspond to solutions of the Towers of Hanoi instance. The authors then let ants navigate the maze to get to a food source. The experiment showed that the ants were able to find the shortest path in the maze, and they were able to adjust the solution if a simple obstacle was added.

The actual biological experiment is scientifically sound. However, the authors then claim that "ants are capable of solving dynamic optimization problems" (i.e., they can optimally solve optimization problems and even adjust the solutions as the instance changes). This is obviously false for optimization problems as known in CS. For n discs the optimal solution of the problem needs exactly $2^n - 1$ moves, which produces an exponentially large maze – unexplorable by any ant colony, real or simulated. The observed results cannot be related to optimal solutions for real optimization problems. At best, they are just another confirmation that studies of ants may help improve heuristic algorithms.

In [20] the authors descibe an experiment on discovering flowers and route optimization by bumblebees. The reoptimization occurred whenever the flowers' locations were manually changed. In the last section of the study authors relate the observed behavior to the famous and provably NP-hard Traveling Salesman Problem (TSP, see [8]) and its dynamic version; they suggest that bumblebees can find an optimal solution for this problem. However, the study [20] never even confirms that the bumblebees' solution in the experiments is indeed always globally optimal, and not just a local optimum. The experiment was done with only four flowers, with no attempt at a generalization for more vertices. And what is even worse, this result was then incorrectly adopted in online articles [19] and [4] that generalized the suggestion to a bold claim that bees can efficiently solve TSP.

We the computer scientists are at least partially responsible for this confusion. One part of the problem here is that the hardness of many optimization problems is pretty subtle. For instance, a person who is not a computer scientist can form an approximate mental model of TSP – "visit all locations as quickly as you can", and also a model of ant colony behavior – "try varying the path and if you find a better one, stick with it". For a computer scientist, these are two very different concepts. We understand that the difficulty of the optimization problem lies in computing its globally optimal solution, or at least in finding a *provably* good approximation. But to a person without a CS background, ant colony behavior can be a very strong metaphor for solving the optimization problem. From this metaphor the person often infers incorrect conclusions about the optimization problem. It should be our duty to clarify such confusion when it occurs, and to look for ways how to prevent it in the future.

8. CONCLUSION

In our paper we presented multiple metaphors that we developed to teach algorithms. To our best knowledge, the metaphors are original, and our experience shows that they are indeed an efficient tool: they help students to form correct mental models, and these mental models lead to deeper understanding of the topics. We also showed that careless usage of flawed metaphors can easily mislead the students, and that it is easy to draw false conclusions based on such flawed metaphors. Teachers should carefully consider the suitability of metaphors before using them. Finally, we stress that metaphors and analogies are just a didactic tool that can help the teacher give a better explanation. They should not (and in principle cannot) replace a formal approach. Instead, they should supplement the formal presentation, helping the students to grasp and visualize it.

9. REFERENCES

[1] Aaronson, S.: NP-complete Problems and Physical Reality. Electronic Colloquium on Computational Complexity (ECCC) 12 (2005)

[1] Bringsjord, S., Taylor, J.: P=NP. arXiv:cs/0406056v1 (2004). It was never published in a peer-reviewed journal.

[2] Bell, T., Fellows, M., Witten, I.: CS Unplugged: Off-line activities and games for all ages (1998).
[3] de Berg, M., Cheong, O., van Kreveld, M., Overmars, M. Computational Geometry: Algorithms and Applications (3rd ed.). Springer-Verlag (2008)
[4] Boyle, R.: Bees solve hard computing problems faster than supercomputers. PopSci (2010).
[5] Chee, Y. S.: Applying Gentner's theory of analogy to the teaching of computer programming. Int. J. of Man-Machine Studies 38, 347–368 (1993)
[6] Cohoon, J. P., Davidson, J. W.: C++ Program Design: An intro to programming and object-oriented design (3rd ed.). McGraw-Hill (2002)
[7] Colburn, T. R., Shute, G. M.: Metaphor in Computer Science. Journal of Applied Logic 6, 526–533 (2008)
[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: Introduction to Algorithms (3rd ed.). The MIT Press (2009)
[9] Danzig, N.: Introduction to Computer Science – C++. Available online: `http://www.danzig.us/cpp/` (2009)
[10] Dasgupta, S., Papadimitriou, C., Vazirani, U.: Algorithms. McGraw-Hill (2006)
[11] Dijkstra, E. W.: On the cruelty of really teaching Computer Science. Comm. of the ACM 32 (1989)
[12] Duncker, E.: Cross-cultural usability of the library metaphor. Proceedings of JCDL 2002, pp. 223–230, ACM (2002)
[13] Gentner, D., Stevens, A. L. (eds.).: Mental Models. L. Erlbaum Assoc. (1983)
[14] Hübscher-Younger, T.: Understanding algorithms through shared metaphors. CHI EA '00 on Human factors in computing systems, pp. 83–84, ACM (2000)
[15] Keogh, J. E., Davidson, K.: Data structures demystified. McGraw-Hill (2004)
[16] Keränen, J.: Using metaphors in Computer Science education – cross cultural aspects. Tech. report, CS Department, Univ. of Joensuu (2005)
[17] Lakoff, G., Johnson, M.: Metaphors We Live By. Univ. of Chicago press (2003)
[18] Landau, H. G.: On dominance relations and the structure of animal societies III. Bull. of Mathematical Biophysics 15, 143–148, Springer, New York (1953)
[19] Levey, S.: Tiny brained bees solve a complex mathematical problem. Queen Mary Univ. of London Media Centre (2010)
[20] Lihoreau, M., Chittka, L., Raine, N. E.: Travel optimization by foraging bumblebees through readjustments of traplines after discovery of new feeding locations. The American Naturalist 176, 744–757, The University of Chicago Press (2010)
[21] Monmarché, N., Guinand, F., Siarry, P. (eds.).: Artificial Ants. Wiley-ISTE (2010)
[22] Narváez, P., Siu, K., Tzeng, H.: New Dynamic SPT Algorithm Based on a Ball-and-String Model. ACM Transactions on Networking 9, 706–718, ACM (2001)
[23] Reid, C. R., Sumpter, D. J. T., Beekman, M.: Optimisation in a natural system: Argentine ants solve the Towers of Hanoi. J. of Exper. Biol. 214 (2011)
[24] Quisquater, J. J., Guillou, L., Annick, M., Berson, T.: How to explain zero-knowledge protocols to your children. Proceedings of CRYPTO 1989, pp. 628–631, Springer-Verlag New York (1989)
[25] Spolsky, J.: The Law of Leaky Abstractions. Joel on Software (2002).
[26] Stephenson, N.: In the beginning... was the command line. Harper Perennial (1999)
[27] Waguespack, L. J.: Visual Metaphors for Teaching Programming Concepts. Proceedings of SIGCSE 1989, pp. 141–145, ACM (1989)
[28] Woollard, J.: The rôle of metaphor in the teaching of computing; towards a taxonomy of pedagogic content knowledge. PhD. Thesis, Univ. of Southampton (2004)
[29] Yim, K., Garcia, D. D., Ahn, S.: Computer Science Illustrated: engaging visual aids for CS education. Proc. of SIGCSE 2010, pp. 465–469, ACM (2010)

APPENDIX

A. TERMINOLOGY

The terms *analogy* and *metaphor* are often misused in practice. To avoid the confusion, we provide definitions and a few examples. For a more detailed treatment of these topics we recommend [17]. An *analogy* is a cognitive process in which a subject transfers information from one particular object to another. The word *analogy* can also be used as a noun describing the similarity between the two particular objects. A sample analogy: CPU is the brain of the machine. It takes input data, processes it and produces outputs.

A (conceptual) *metaphor* is a cognitive process that occurs when a subject seeks understanding of one idea (the target domain) in terms of a different, already known idea (the source domain). The subject creates a conceptual mapping between the properties of the source and the target, thereby gaining new understanding about the target. The metaphor can be conveyed by any means of communication, not only by words. For an *anthropomorphic metaphor (personification)* the source domain is a living being, usually a human. Sample metaphors: calling the workspace in a graphic environment "desktop", calling breadth-first search "flood fill". Personifications are common in colloquial speech of computer scientists, for instance in the expressions "the program is running", "the compiler needs to see a semicolon there", "the client machines periodically ask the server", etc.

By definition every metaphor is an analogy, but not vice versa. In an analogy, only some of the properties of the source are transferred to the target; these properties may be explicitly stated as a part of the analogy. In a good metaphor, the target and source should match on all relevant properties, enabling us to infer information about the target in terms of the source. For example, the analogy "the CPU is the brain" is a poor metaphor – it is impossible to think about the CPU in terms of what we know about our brains. In classroom setting, such poor metaphors may lead to unnecessary confusion.

An *abstraction* is a process by which a subject derives more general, higher level knowledge from patterns observed in multiple particular objects. For example, pseudocode is an abstraction from various imperative programming languages, and the strategy of looking for a greedy algorithm is an abstraction of observing concrete problems for which particular greedy algorithms work.