The genesis and philosophy of Algovision

Luděk Kučera

2000-2023

Introduction

The present text describes Algovision (www.algovision.org), a collection of algorithm visualizations, developed at Charles University (Prague, Czech Republic) as a tool for teaching/learning algorithms.

I assume that a reader of this text is a CS teacher, who already knows the presented algorithms. I only add features of Algovision that are not used in standard algorithm visualizations that can be found in the web.

I had an interesting experience when I presented Algovision at STIU (Schweizer Tag für den Informatikunterricht - a seminar for Swiss CS teachers, organized by ETH Zürich). When I was explaining, using Algovision visualization, the idea of some algorithm, one of the participants asked me "Why you don't show first the *algorithm itself*?"

I was surprised by the question, because I thought I was explaining the "algorithm itself". After a short discussion, I understood that for the inquirer, the term "algorithm itself" meant a pseudocode of the algorithm.

Another talk at STIU, delivered by Dennis Komm, gave me a partial explanation of our different points of view. Dennis presented results of his research showing that young students tend to "personalize" computers - students believe that computers behave much like human beings, understanding what a user wants and trying to work in that way.

I realize that we, computer science teacher, tend to "computerize" our students - we act as believing that the way students accept algorithms is much like the way computers accept algorithms: the knowledge of an algorithm is its formally correct and complete description (e.g., a pseudocode).

However, we, computer science researchers, know pretty well that a pseudocode is the last phase of algorithm development, one short step from a true computer code that can be unconsciously executed by a computer, but the true knowledge of the algorithm involves the idea that led the discoverer to its final goal.

Unfortunately, the algorithm idea is often hidden deep in the corresponding pseudocode or code and not everybody is able to find it.

At this point, I use to tell my another experience. On Feb. 5, 2006 I happened to arrive to Pittsburgh to meet a colleague at CMU. That day Pittsburgh Steelers played against Seattle Seahawks in the Super bowl. "Steelers go" was posted at every corner and I understood as my social obligation to watch the game in my hotel room. I saw strong and swift young men running and holding and kicking the ball, but, as a European without any idea of the rules, I was even unable to figure out who won. I had to go to the lobby to learn that Steelers won 21-10.

The same happens when a learner observes a standard algorithm animation that is showing the progress of an algorithm by updating a visual representation of a data structure. Circles move and colors change, letters appear and leave, but if a learner does not know first the idea that makes and controls the algorithm to reach its final goal (like the rules in football), he or she sees just an entertaining Brownian movement.

I believe that this is the main reason of still rather low acceptance of algorithm animations as a teaching and learning tool (see references in the list at https://kam.mff.cuni.cz/~ludek/superreference.html).

The philosophy of Algovision is to present first of all, using visual means, the algorithm idea. The goal is often achieved by showing what I sometimes call the "prenatal development of an algorithm" - a sequence of ideas in the algorithm discoverer's mind that had led or might have led to finding the algorithm. Of course, the true path to a new computational method is always full of dead-ends and detours and Algovision shows a cleaned version, but

I feel important to present to students a *path* and not just the final product.

As a consequence, an Algovision lesson often goes in the opposite direction than a standard one: instead of starting with the algorithm (pseudo)code, an Algovision lesson *finishes* with it at the moment when a learner already understands the algorithm and the ideas that are behind it.

Algovision lectures, based on the presented philosophy, are conceived as full lectures that are intended as primary tools in teaching/learning algorithms and data structures. My experience is that students highly appreciate this approach and like it. For example, the enrollment in the algorithm and data structure course at my university that is based uniquely on Algorithm visualizations is 168 students in the actual semester.

Now, particular algorithms covered by Algovision will be commented. It is recommended to read the text when watching Algovision visualizations at
www.algovision.org/Algovision/pool.html.

Data Structures

Binary search tree

Scene "Binary Search Tree" and many other scenes

One visual feature that is quite simple and useful, but not standard in BST visualizations, is showing the value stored in a node graphically by a column below the node. Students like it and it contributes to deeper understanding of deletion of the value in a node with both children.

Scene "Depth of BST"

Select either Uniform of Monotone in the menu in the control bar and keep clicking Add Nodes. The scene shows that the depth of a BST is logarithmic if insertions are random (the max depth about $2\log_2 n$, the average node depth about $\log_2 n$), but it can be linear (a tree degenerated to a path) in the worst case.

Scene "Rotation in a Tree" and the subsequent scenes

A "playground" that prepares a learner for tree balancing (AVL-trees and red-black trees). Click any edge to see its rotation. A supervisor asks students to perform different tasks using rotations: to promote a given node to the root, transform a tree to a perfectly balance tree or to a path (the worst possible tree). The scene "**Rotation Challenge**" is an exercise to transform the yellow tree to match the gray one. In the scene "**BST Operations**" one can edit tree by adding new nodes or erasing existing ones and rotate edges afterwards.

The playground is appreciated by students as a tool that builds excellent understanding rotation a BST tree balancing.

AVL Trees

All scenes of AVL-trees

The main rule of visualization is: what is important must be immediately visible. The most important attribute of an AVL-tree node is its *balance* - the difference of the depths of its side subtrees (it must have values -1, 0, and +1 only). I do not know why no AVL-tree visualization I have seen in the web draw nodes as circles that hide the node balance. Algovision does it by drawing nodes as balances (and using red color for illegal nodes). Simple, but very useful and highly appreciated by students.

Scene "Make AVL by Rotations"

Another playground, similar to BST playgrounds: using rotation, a learner has to transform the tree in the window to a correct AVL-tree. One of the most preferred scenes among students.

Scene "AVL Simple Insertion" and some others

Keep clicking Go on until a node is inserted. No AVL-tree visualization I know tells a learner that there are two values related to a node balance. The actual balance is the first one. It would be time consuming to recompute node balances all the time. Therefore, an AVL-tree node contains a variable that stores the node balance to avoid necessity of re-computation.

When a new node is inserted, the actual balance of certain nodes can change. However, until the balance variable of such a node is updated, it is invalid for certain time interval.

Algovision shows a node as a pair of two balances: a yellow one, representing the actual balance, and a green one behind, representing the balance variable.

When the balance variable is correct, the green body is invisible, being hidden by the yellow one. When the balance variable needs update, the green body is partially visible to remind a user that an update is necessary.

B-trees

The scene "Insert with Split" and many subsequent scenes

The last phases of the operation show a split of a node - the operation which is a tool that allows to manage *all* complicated situations that occur during B-tree insertions and deletions. When deleting, it is performed in the reversed direction as a node join.

Note, for example, how a join followed immediately by a split (using another moving data box) allows to perform adoption of a data box from a sibling node in scenes "**Delete with Left/Right Adoption**".

Red-Black Trees

The scene "Rotation and Moving Black"

This scene is a playground of red-black tree lesson. A learner can rotate edges, learn which rotations are allowed (in order not the violate that main rule of red-black trees about the black depth of nodes.

It is also possible to perform color flips: a change of colors of a node and its both children, which is visualized as moving black colors from a node to its children or back. To send black color from a parent to its children, push the mouse on the parent and drag it down, when sending black colors from children to a parent, push on the parent and drag up.

In some difficult cases of delete operation, it is convenient to accept *doubly black* nodes. Try to send black colors of children nodes to their black parent.

The scene "Red-Black and 2-3-4"

An extremely important scene showing an equivalence of red-black trees and B-trees with 2, 3, or 4 pointers per node (2-3-4-trees). Click Show clusters and red nodes of the tree move up to touch their black parents. Click Show B-tree and clusters are transformed into nodes of a B-tree - and the equivalence proof is finished.

The equivalence is far reaching - a color flip in a red-black tree corresponds to a split or join operation in the corresponding 2-3-4-tree. Red-black trees allow rotating edges with a red lower node only. Such rotations do not change the corresponding B-tree node, in a cluster of the intermediate red-black tree they just change the internal structure of a cluster.

Standard (Binary) Heap

Scene "Array representation" and the subsequent scenes

The only non-standard feature of this visualization is animation of a transformation of the tree representation of a heap to its representation as an array. Click Change view.

Binomial Heap

A binomial heap and its operations mimic addition of binary numbers. To show this fact to a learner, when showing a binomial heap, the digits of the number(s) are always shown in the background.

Fibonacci Heap

Scene "Fibonacci Trees" and two subsequent scenes

Visualizations of a Fibonacci heap that can be found in the web show execution of operations only. Algovision has two extra goals. The first one is to let a learner to know as many forms of trees that can appear in a Fibonacci heap. The scene "**Fibonacci Trees**" is a playground where a supervisor or a learner can build Fibonacci trees by themselves.

The following scene is an exercise: a learner is invited to build a tree isomorphic to a tree drawn in the background. For example, in order to properly understand certain operations, a learner is invited to construct a long simple path (the worst possible case for BST appears here as an admissible form).

Minimum Fibonacci Tree

The second goal of Algovision is to show, for a given k, the smallest Fibonacci tree in which the root has k children. Clicking Go on shows (for $k > 2$) that a minimum tree for a given k is like a "union" of trees for $k-1$ and $k-2$.

Sorting

Mergesort

Scene "Bottom-up Merge Sort"

The scene and most of other scenes point out that the activity of Mergesort is almost independent on the input data. The scene shows a framework for particular subsequence merges that is given and does not change with the input.

Quicksort

Nothing interesting is presented in animation of Quicksort

Bubblesort

Nothing interesting is presented in animation of Quicksort

Median

Scene "Linear Time"

When the median of medians of 5-tuples is determined (it appears as a green square), checking Not Greater or Not Smaller visualizes the idea of the proof of linear time complexity (the values of the emphasized items are not greater/smaller than the value of the median of 5-tuple medians).

Bitonic Sort

Scenes "Find Partition", "Compare Items"

Most bitonic sort visualizations are just showing moving of columns representing sorted items without any explanation. No one visualizes a proof of the algorithm correctness, which Algovision does in these two scenes.

Graph Algorithms

Extremal Path

This lesson visualizes 3 algorithms: critical path method in an acyclic directed graph, Dijkstra's algorithm and Bellmen-Ford algorithm. The CPM is just a straightforward animation.

Scene "Dynamic Dijkstra"

Once more I repeat the main rule of visualization: what is important must be immediately and clearly visible. The most important parameter of a node is the estimate of the distance from the source that decreases during the computation to be equal to the distance from the source at the end of computation.

All visualizations I know show the value of the estimate as a number written in the node or just next to it. This is anything else but not clear and immediately visible.

Algovision presents a standard Dijkstra visualization ("**Static Dijkstra**"), but the main scene is "**Dynamic Dijkstra**", where nodes are allowed to slide along horizontal lines, their x-coordinate being proportional to the value of the estimate.

This makes not only visible the value of estimate, but makes it trivial to choose a node for the processing (the leftmost among open nodes) and the two invariants that are used in the correctness proof, namely
the estimate of any closed node is less or equal to the estimate of any open node, and
the estimate of a node is the length of the shortest *closed* path into the node (a path composed of edges that have a dark color)
are now so obvious that students are often able to guess them without a help of a teacher.

Scene "Bellman-Ford"

Finish the computation (by clicking Go on or a single click to Finish), check Show Phases, reset the computation and run computation again step-by-step using Go on. It can now be seen easily how the computation progresses from left to right which gives better insight into the proof of termination and complexity. For details, use the guide of Algovision for this algorithm.

Spanning Tree

Scene "Correctness"

A relatively standard visualization is finished by animating of the correctness proof that I haven't seen anywhere in the web.

Network Flows

Scene "Dinitz Algorithm"

In Ford-Fulkerson max flow algorithm if an augmenting path is processed, at least one general arrow that can be a member of an augmented path is saturated and can not be used again (in that moment), while some new arrows appear (those that are reversed orientations of the arrows of th augmenting path).

Dinitz published its algorithm about the same time as Edmonds and Karp; his algorithm introduces one technical feature that make its implementation slightly faster. The feature is very useful for visualization: click Go on twice. The graph is arranged to layers by the distance from the source node.

The idea of Dinitz, Edmonds and Karp is to look always for for the shortest augmenting path, i.e., a path which, in the present visualization, go from left to right. Now, keep clicking Go on until a green path from the source to the sink is completely built.

During the process, edges that do not go from left to right, and then those that lead to a dead-end, are colored yellow and not considered. (You can click Hide Yellow to hide them).

When the green path is processed, at least one of the left-to-right arrows disappears, and arrows that appear are all oriented as opposite to the path arrows, i.e., their orientation is *right-to-left* and the shortest path rule *forbids* their use in the present phase of the algorithm.

Consequently, the present phase (defined by the length of the shortest source-sink path) will finish soon, because the set of usable edges decreases, and clearly any subsequent phase has longer shortest path, and hence the number of phases is limited by $N-1$ (N being the number of nodes), whee N is the number of nodes.

Scene "3 Indians"

Once the Dinitz-Edmonds-Karp idea is understood, the algorithm of Malhotra-Kumar-Maheshwari reduces to a standard animation of the pseudocode.

Section "Goldberg"

The algorithm of Goldberg (or push-relabel) has a very simple description and its function is visualized by very simple examples of paths with possible branching. However, the termination and correctness proof is not trivial and not well suited for visualization. An attempt to visualize to proof is provided.

Graph Min Cut

Scene "Minimum Cut"

How to find the minimum cut in a regular graph of the degree D using eigenvectors? Check Show spectrum to see the second eigenvector of the incidence matrix of the graph. Click Change View to see that a randomly looking graph is in fact composed of two unconnected parts of the same size and the signature of the eigenvector elements show how to partition the graph to minimize the cut.

Click Step to change the graph so that it is not any more unconnected; there are two edges connecting the parts. Click one of two green columns that are visibly smaller than the other ones - you will see that it correspond to a node that has a neighbor in the other part of the graph, similarly for the black columns. Click Step several times to see how connectivity of the graph changes and the changes are reflected by the eigenvector.

The question is why we are using eigenvectors and why this works. The visualization suggests to simplify the method by looking for a *score* of nodes which also give a good cut by signature partition. One promising strategy is to define score so that my score is always close to score of my neighbors, because if neighbors of a node tend to be in the same part of the graph as the node, the corresponding cut is hopefully small.

What about my score to be the average of score of my neighbors? Click Formula to see the corresponding

formula. Click Formula again to rewriting the formula using the incidence matrix of the graph.

The problem is that the equation usually has no non-trivial solution. What about replacing D, the degree of nodes, by a general λ (click Formula again). Yes, this works well - and what you see in the frame is a definition of an eigenvector corresponding to the eigenvalue λ.

Why we are using the second eigenvector? Because, in the case of a regular graph, the first eigenvector is constant and it gives a solution "put all nodes into the same group with the score 1", which perfectly solves the problem of my score being the average of the scores of my neighbors, but is is useless in search of a minimum cut.

Scene "Regular Matrix Eigenvalues"

A technical scene that shows why D is the largest eigenvalue corresponding to a constant vector, and why, in the case of a completely separated graph, D has the multiplicity (at least) 2 and the second eigenvector looks like we already know from the first scene.

Arithmetic Algorithms

Addition

Scene "Random Split"

Carry look-ahead (CLA) adders are algorithms that are not a object of visualization. The key scene in Algovision visualization is "**Random Split**", which shows that the decision whether a block of operand columns generates, propagates, or kills carry can be decided by a circuit that, if split properly, has logarithmic depth (with respect to the width of the block).

Different methods of uniform splitting give known algorithms, for example Kogge-Stone, Ladner-Fischer, Brent-Kung and some others.

Discrete Fourier Transform

Scene "Spectral Analysis"

The Discrete Fourier Transform (DFT) is an important computational methods, used, e.g., in signal processing and data compression. Unlike for other problems (like shortest paths), where the formulation and applications of the problems are obvious, DFT needs clear explanation of possible applications (e.g., CLRS, a "bible" of algorithm learning, offers DFT as a method for formal multiplication of polynomials - not a very practical application, taking into account that the DFT-based number multiplication algorithms are not widely used in practice).

The Algovision visualization shows how to find a spectrum of a periodic function. Draw a graph of a function by the mouse into the upper rectangle, the spectrum of the function appears in the lower rectangle. A linear combination of sin and cos of different frequencies, and const with the coefficients graphically represented in the lower rectangle (the red graph) hits the black function in the sampling points represented by vertical line segments.

Try to draw a "wild" black function to make red function more different from the black function (outside the sampling points).

Scene "Spectrum search"

Once more draw a black function into the upper rectangle. Now it is your turn to find coefficients. Try to guess one value by dragging the mouse in the corresponding column in the lower rectangle. Then, click the cross below to get the correct answer. A very valuable playground.

Scene "Spectrum compression"

Click to the left of the right part of the lower rectangle. A part of the spectrum disappears, and the red function is computed as a linear combination of trigonometric functions that correspond to columns that remained.

The red function is not any more equal to the black one in the sample points, but still looks similar. This is the principle of DFT data compression.

Scene "Cosine transformation"

Draw a black function in the left part of the upper rectangle, the function in the right part is its mirror image. DFT coefficients of sin members of the spectrum are 0, only cos members are important. This gives a cosine transformation, a spin-off of the DFT.

Fast Fourier Transform

A sequence of modifications of the original formula for DFT gives eventually a butterfly network to compute the transformation.

Conjugate Gradient Method

The CGM is perhaps the most illustrative example of the Algovision philosophy. I am not aware of any similar visualization in the web.

Scene "Computer Simulations"

Very short motivation for CGM and its use in CFD (Computational Fluid Dynamics) that is based on solving very large sparse systems of linear equations. The Algovision author cooperates with the data center of the Czech car maker Škoda Auto (Volkswagen group), where the CGM based simulations are used to optimize the automobile drag coefficient of they cars. E.g., the coefficient of the model Škoda Octavia (see figure in the window) was reduced from 0.32 (Octavia 1) to 0.24 (Octavia 4).

Scene "Gaussian Elimination"

Since matrices of linear systems that are used in CFD and similar simulations are typically very sparse, extremely large matrices would fit into a computer memory if they are properly compressed.

However, after Gaussian elimination, even sparse matrices become usually dense and would overflow the memory. Thus, Gaussian elimination is not well suited for simulation purposes and other methods of solving linear systems are needed.

The scene shows a visual representation of a randomly generated sparse 300×300 matrix (non-zeroes represented by white pixels). After Gaussian elimination (click Gauss), the matrix is much denser.

Scene "Linear Equation"

Even though CGM is supposed to be a very difficult algorithm, it in fact follows from two simple observations. The first one is explained in the scenes "**Linear Equation**" and "**Systems of Linear Equations**".

If $a > 0$ the solution of $ax = b$ is the same number as x that minimizes $f(x) = \frac{1}{2}ax^2 - bx$ (a primitive function to $ax - b$). For $a < 0$ we could replace the minimum by the maximum.

The next scene "**Systems of Linear Equations**" generalizes the theorem to higher dimensions, see the theorem displayed in the window (keep clicking Formula).

In fact, the solution of $\mathbf{Ax} = \mathbf{b}$ is the point, where all partial derivatives of Φ are 0. For a positive definite matrix, this represents the minimum.

Scene "Positive Definite Matrix"

The scene shows how the graph of a functional Φ looks like in for a positive definite 2×2 matrix. A positive definite matrix is a matrix that gives rise to a graph that tends to $+\infty$ in all directions.

Scene "Indefinite Matrix"

If a matrix is indefinite, the graph of the functional looks typically like shown in the window, the saddle point still represents the solution of $\mathbf{Ax} = \mathbf{b}$, but the surface has neither minimum nor maximum.

Scene "Steepest Descent"

Click the quadratic surface in an arbitrary point and then keep clicking Go down to see how the method of the steepest descent converges to the minimum of the surface.

Scene "Conjugate Gradients"

The second principal idea of the CGM due to Hestenes and Stiefel (1952). In order to speed up the convergence, let us look for the direction that points directly to the lowest point of the surface.

Click the surface to select a starting point. Click Spheric to compress/expand the space in the directions of the eigenvectors of the matrix (axes of elliptic contour lines on the surface), using different compression/expansion rates in different directions to get eventually circular contour lines.

Apply the standard steepest descent that now points to the minimum (click Go down). Return to the original uncompressed space (click Elliptic, remembering the direction.

Keep clicking Formula to see a mathematical record of this sequence of actions to learn the direction to the minimum is the one that is *conjugated* to the tangential direction of the contour line in the considered point. We say that two vectors **u** and **v** (with respect to a matrix **A**, if the product **uAv** is 0.

Once we are able to recognize the direction to the surface minimum, the following scenes, showing the Conjugate Gradient Method in 2D and 3D are purely technical and do not need any other deeper idea. At the end, in the scene "**The CGM Algorithm**", we match actions made during visualizing scenes with the corresponding lines of the standard CGM code.

Convex Hull

Scene "Complexity"

This visualization is rather standard, but the last scene presents a visual proof of linear time complexity (provided the input points are sorted by, e.g., their x-coordinates.

Voronoi Diagram

Scene "Cones"

Visualization of Fortune's algorithm for construction of a Voronoi diagram of a finite set of points (sites) in the plane is one of the most illustrative examples of Algovision philosophy. While all Fortune's algorithm animations available in the web (including the one in Wikipedia) are similar to the animation in the scene "**Animation**", Algovision uses interactive 3D graphics to show the original plane embedded as a horizontal plane into the 3D space with cones hanging below each site with their summits being the sites.

Viewed vertically, visible parts of cone surfaces give exactly Voronoi domains of sites that we seek.

Scene "Sweep plane"

The animation of the scene "**Animation**" is a vertical view of cones of the previous scene that are swept by an inclined plane (a rectangular section of the plane is shown, using the green color). This scene brings immediately understanding of the underlying idea of Fortune's algorithm.

This and the previous scenes need 3D graphics. It is necessary that the graphics is interactive so that a learner can see cones under different angles to understand how cones are built. This is why the 3D view is not used in textbooks, and, consequently, Fortune's algorithm visualizers have obviously no idea of cones (that had been folklore among experts for years, Einstein's theory of relativity uses similar 4D cones in the space-time analysis; cones are brought into a seemingly linear problem by the quadratic nature of Euclidean distance).

Scene "Original Fortune cont'd"

This scene shows that Steve Fortune used the 3D view shown in scenes "**Cones**" and "**Sweep Plane**", but viewed the cones under a different angle.

String Matching

Knuth-Morris-Pratt and Aho-Corasick

Scene "Floating pattern"

Our experience is that students have no problems to understand an easy idea of a floating pattern. However, there is deep intention in introducing the term: it illustrates the most important invariant of the KMP algorithm, namely that the state of the computation is given by the longest prefix of the pattern that is in the same time a suffix of the read text (the read text and the longest matching prefix use white backgrounds of squares, the longest matching prefix is represented by the part of the floating pattern that protrudes left of two red triangles).

Scene "Matching Prefixes"

Another way of representing the algorithm data shows in the same time so many copies of the pattern that all potentially matching prefixes of the pattern are shown. In this way it is possible to show all matching prefixes, among them those that can be extended by a newly read letter of the text, and relation among prefixes and their properties.

When learners understand well the first two scenes of the visualization, it is quite easy to complete the lesson.

Rabin-Karp

No really deep idea is needed to show how the algorithm is working.

Linear Programming

Simplex Algorithms

As the name of the method suggests, geometric view of the problem of linear programming had been known when the simplex algorithm was described for the first time.

The algorithm is visualized for the case of 2 and 3 variables (unfortunately, we have no intuition for 4D and higher dimensional Euclidean spaces).

The use of interactive 3D graphics is absolutely necessary in the case of 3 variables so that a learner can observe the simplex under different angles.

Scene "Moving Along Edges"

There are 2 scenes of this name, one in the 2 variable case, another one in the case of 3 variables.

The reason why Algovision uses these scenes is to build bridge between the geometric intuition and a formal matrix formulation of the problem.

When working in the dimension N (N variables), then in a non-degenerated case a vertex of a simplex is a point that verifies all inequalities of the problem, exactly N of them being satisfied as equality.

Moving along edge of the simplex means to drop one of N equalities for the given vertex, which defines a *line* and moving along the satisfying portion of the line until another vertex is found (this explains the notion of a *pivot*).

Perhaps a slight innovation of the present visualization is the way the linear functional to be maximized/minimized is shown (both in 2D and mainly in 3D).