Developing Gato and CATBox with Python: Teaching graph algorithms through visualization and experimentation

Alexander Schliep[1] and Winfried Hochstättler[2]

[1] ZAIK/ZPR, University of Cologne, Cologne, 50937, Germany
[2] Institute of Mathematics, Clausthal University of Technology, Clausthal-Zellerfeld, 38678, Germany

1 Abstract

Teaching algorithms, especially graph algorithms, is one of the natural applications of multi-media in mathematics. The objects considered are of a highly dynamic nature and require an adequate dynamic visualization.

CATBox, the combinatorial algorithm tool-box, is an interactive course combining a textbook with the visualization software Gato, the graph animation tool-box. For the design of Gato, the following simple rules were used to address issues of clearness of presentation, consistency, level of interactivity and software engineering.

- Use a real programming language instead of pseudo-code: By choosing Python for presenting algorithms we bought consistency of the presentation at the expense of using a syntactically slightly more complex language compared to traditional pseudo-code.
- Visualization based on rules: *Interesting events* [5] in the visualization always correspond to changes to data-structures used in the algorithm. Therefore, those changes to data-structures should be tightly coupled to the corresponding visualization commands. For reasons of consistency and maintainability we added these 'rules' as sub-classes of abstract classes implementing data-structures. This occasionally provided surprising insights even in the case of very simple algorithms.
- Choose the appropriate software framework: Over a number of years various predecessors to CATBox, all under the same name but with slightly different feature sets, have been implemented in a number of ways. For the current implementation we chose Python/Tkinter, due to the rules mentioned above and its availability on a large number of computer-platforms.
- Good Software Engineering: The applicable aspects of the 'Extreme Programming' methodology were used for developing Gato.

The application of these rules yields a software which allows learners to experiment with both problem instances and the algorithms themselves.

We will also address software engineering issues arising in an academic setting with special emphasis on software quality control and cross-platform requirements. Additionally, positive experiences resulting from licensing Gato freely under the LGPL (Library GNU Public License) will be discussed.

2 Introduction

Due to the advance of computer technology over the last decades, theoretical computer science, the theory of algorithms, gained tremendously in relevance. Besides its importance in the pure and applied sciences, it also became part of the tool-box in areas as diverse as Biology, Electrical Engineering, Operations Research and Civil Engineering.

When considered merely a tool, the direct educational goal is programming proficiency, and not understanding of theoretical computer science. But in reality, programming proficiency depends on a number of distinct qualities:

1. Algorithmic thinking,
2. proficiency in the implementation language, and
3. engineering aspects of programming; e.g., knowledge of development methodologies such as design patterns, implementation frameworks, development tools, languages appropriate to the problem at hand.

For beginning learners points 1. or 2. alone are usually difficult enough to master. If these distinct areas are subsumed into one programming class, the theoretical aspects of point 1. are obfuscated under the syntactic and technical complexities of 2. – imagine learning Mathematics in an unknown foreign language. Therefore you will find distinct algorithm and programming language classes in most Computer Science curricula.

In the algorithm classes, graphs are the most widely used mathematical model to formalize problems. They are very intuitive models, can be used to model a large number of relevant problems from many different disciplines, support problems from all known complexity classes and are suited to demonstrate a wide range of algorithmic approaches.

Teaching graph algorithms is one of the natural applications of multimedia in mathematics. The objects considered are highly dynamic in nature and require an adequate dynamic visualization. The existing packages for visualization, a partial survey of which can be found in [22], can be roughly separated in two categories.

Instance visualization tools, e.g., BALSA [4] or TANGO [23,15], only show the effect an algorithm has on the problem instance (i.e., the graph the algorithm operates on), without revealing the code of the algorithm. Thus, they occasionally further the understanding of those who already know the algorithm, but rarely offer insights to beginners.

Code and instance visualization tools, as Leonardo [8,19], show both the cause, a statement of an algorithm, and the effect, a change in the problem instance. The link between cause and effect is usually achieved, as it is in the previous category, by interspersing visualization commands between the algorithm's commands. This highly obfuscates the code for complex algorithms. Sometimes the presentation of an algorithm is made more appealing by hiding the actual code executed and presenting a made-up version of the algorithm instead. Either way, this adds a layer of indirection and a possible source of inconsistency.

We decided to develop an environment which addresses these issues by a design based on consistency and immediate visualization and allows for learning through extensive experimentation. CATBox, the Combinatorial Algorithm Tool-Box, is an interactive course combining a textbook with the visualization software Gato, the Graph Animation TOol-box.

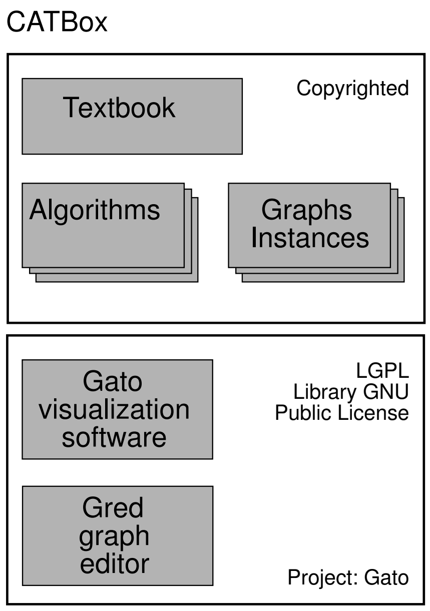


Fig. 1. Overview of the components in the CATBox project.

It is based upon experiences gained in earlier projects at the ZAIK/ZPR, University of Cologne, and from teaching courses in this area at the Technical University Cottbus and the University of Cologne. The target audience at present are undergraduate students in mathematics and computer science.

Among other topics, minimal spanning tree, shortest-path, maximal-flow and both weighted and non-weighted matching problems are covered. Each problem is first motivated by real-world examples. Students learn about different possible solving strategies. This leads to the introduction of algorithms. Relevant mathematical prerequisites and theoretical investigations of the algorithms are then rigorously described. The introduction and refinement of the algorithms go hand in hand with experimentation with Gato, putting an emphasis on extreme cases, such as worst-case examples.

Gato uses a graphical, window-based user interface. The code of the algorithm is displayed in the algorithm window. It also contains the user interface controls. For examination of the algorithm a familiar debugger look-and-feel is employed. A user can start or stop an algorithm, set breakpoints (i.e., lines at which execution is stopped), trace into procedures called (i.e., show the code for the procedure) and continue execution. In a second window the graph the algorithm is operating on is displayed. For every action of the algorithm visual feedback is given by – for example – changing vertex or edge colors. We will explain our rule-based visualization strategy and its implementation in Sect. 4.

By moving the mouse pointer over vertices and edges, additional information (e.g., edge weight) is displayed. Certain operations, such as selecting a vertex, can be performed interactively. Gred, the GRaph EDitor, an easy to use editor for graphs, is integrated. Gato and Gred run on a large number of different operating systems. For editing algorithms a standard text editor can be used.

In the following sections we will discuss the design rationale behind Gato, the choice of Python as a language for representing algorithms, our choice of implementation platform and the design of our visualization engine. Then, we will introduce the rule-based visualization strategy we employed and its technical realization. We will conclude with a discussion of the 'extreme programming' methodology employed in the development of Gato. There, we will give consideration to software engineering issues in an academic setting and positive experiences resulting from licensing Gato under the LGPL (Library GNU Public License).

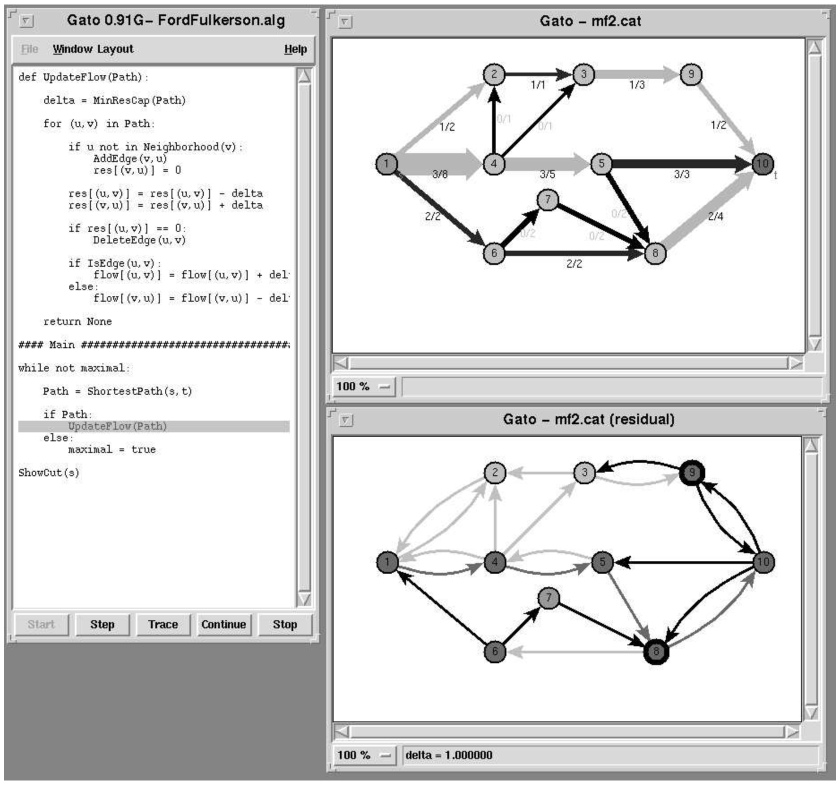
3 The design and implementation of Gato

Gato consists of three major functional building blocks, as shown in Fig. 4. The data we have to deal with are algorithms and the instances they act on; i.e., graphs.

1. The *visualization engine* supplies in its *algorithm engine* support to execute algorithms under full user control, and also the technical foundation to provide visual feedback.
2. The algorithm display shows the algorithm on-screen and is the central component of the graphical-user-interface (GUI).
3. The graph display enables visualization by providing a rich methodic interface for controlling the graphical display of graphs and editing functions[1].

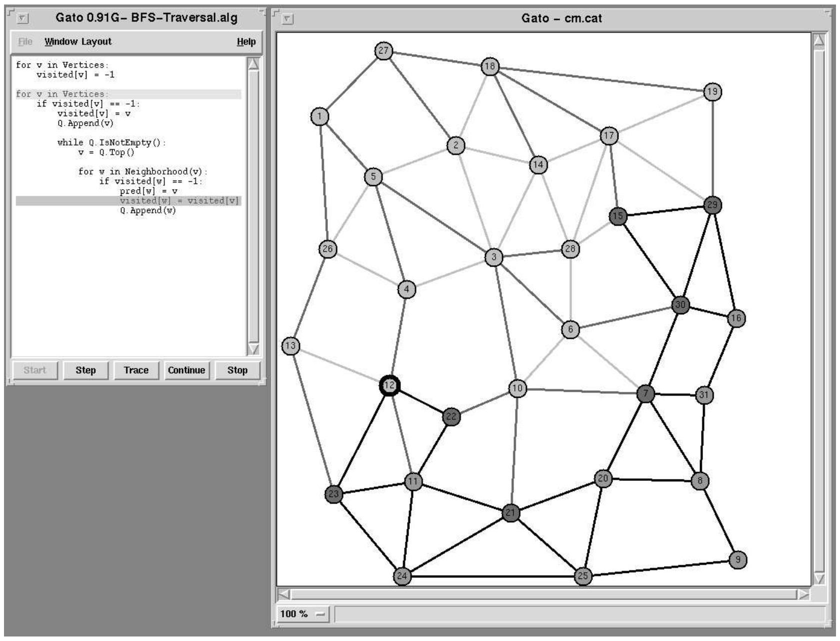
As far as the general graphical-user-interface metaphor was concerned we aimed for a very simple debugger.

[1] The GUI to editing functions is implemented in Gred.



```
def UpdateFlow(Path):

    delta = MinResCap(Path)

    for (u,v) in Path:

        if u not in Neighborhood(v):
            AddEdge(v,u)
            res[(v,u)] = 0

        res[(u,v)] = res[(u,v)] - delta
        res[(v,u)] = res[(v,u)] + delta

        if res[(u,v)] == 0:
            DeleteEdge(u,v)

        if IsEdge(u,v):
            flow[(u,v)] = flow[(u,v)] + del
        else:
            flow[(v,u)] = flow[(v,u)] - del

    return None

#### Main ####################################

while not maximal:

    Path = ShortestPath(s,t)

    if Path:
        UpdateFlow(Path)
    else:
        maximal = true

ShowCut(s)
```

Fig. 2. The Ford-Fulkerson algorithm [1] for computing a maximum flow in a capacitated directed graph is visualized in Gato. The algorithm window is shown on the left, the graph on the right and the corresponding residual network on the bottom right.



```
for v in Vertices:
    visited[v] = -1

for v in Vertices:
    if visited[v] == -1:
        visited[v] = v
        Q.Append(v)

        while Q.IsNotEmpty():
            v = Q.Top()

            for w in Neighborhood(v):
                if visited[w] == -1:
                    pred[w] = v
                    visited[w] = visited[v]
                    Q.Append(w)
```

Fig. 3. Dijkstra's algorithm [1] for computing all shortest paths from a given root vertex is displayed.

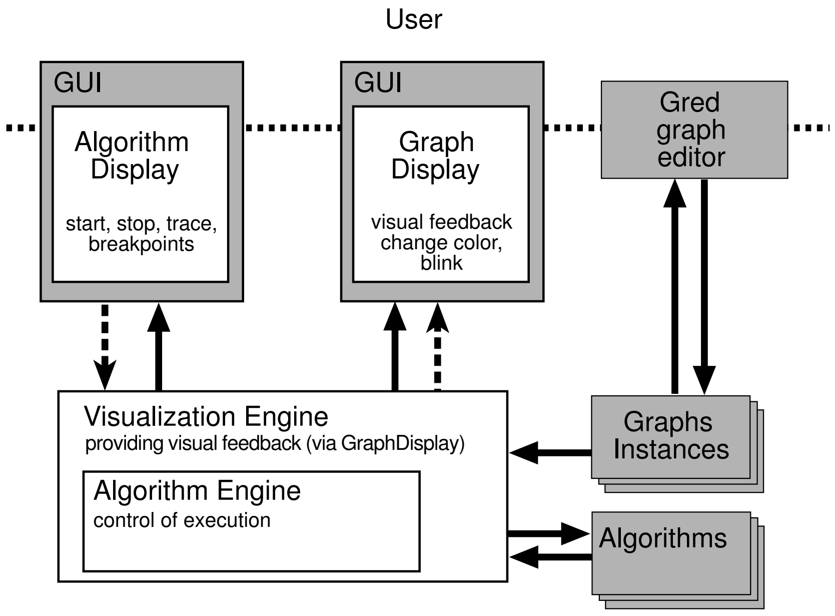


Fig. 4. The functional building blocks in Gato. Dashed lines designate a control relation and solid lines data-flow.

3.1 Choosing Python

According to [20] python can be described as

> ... an interpreted, interactive, object-oriented programming language. It is often compared to Tcl, Perl, Scheme or Java.
>
> Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface.

Python is used in a number of successful projects for teaching aspects of computer science and programming for an audience from middle-school to university students [7,10,18,2]. The special interest group (SIG) on use of Python in education, EDU-Sig [9] provides an active developer mailing list and further links to relevant projects.

What from our point of view really distinguishes Python for use in educational applications from other languages such as C, C++ or Java is its remarkably clear and easy to learn syntax, cf. Fig. 5, which does however look familiar enough to users already used to C, Pascal, etc.. As a matter of fact, for small examples Python code is not more complex than pseudo-code in use in textbooks. The second author has also used Python with great success for practical exercises in a numerical algorithm class [14] taught to second-year students from business information systems ("Wirtschaftsinformatik"), supporting the usability of Python for students with no prior programming experience. Compared to pseudo-code, the formal language definition enforces a higher level of consistency in the formulation of algorithms.

The requirements of the algorithm engine also motivated our choice of Python. Due to its interpreted nature and the fact that parts of the run-time system are written in Python itself, the standard library provides a debugger class [21]. Hence, the development of our algorithm engine consists of sub-classing the existing debugger class and providing the hooks for user interface control and visualization display. Thus, we arrived at a solution where the algorithm code displayed is the actual code executed and also the code triggering the visualization.

3.2 Python as an implementation language

Since a Python interpreter can easily be embedded into other applications [27], the choice of the implementation language was an independent one. From our point of view, the thorough object-orientation and the excellent exception handling in the core language, and the large number of

```
for v in Vertices:
    visited[v] = 0

root = PickVertex()
visited[root] = 1
Q.Append(root)

while Q.IsNotEmpty():
    v = Q.Top()
    for w in Neighborhood(v):
        if visited[w] == 0:
            visited[w] = 1
            Q.Append(w)
```

Fig. 5. An implementation of a Breadth-First-Search (BFS) algorithm in Python.

functional areas[2] covered by the standard libraries are some of Python's very strong points.

Python allows the use of a large number of different GUI-frameworks [25]. We choose to use Tkinter, the library providing bindings to the well known Tk (as in Tcl/Tk [26]) framework, for various reasons. It is the only framework which, besides Unix, Linux and Microsoft Windows, supports Macintosh OS reasonably well. Also, it offered the widest variety of 'rich' widgets at the time we had to make this design decision. For example the canvas, the class to display graphical objects, is thoroughly object oriented. Objects such as lines, circles, text labels placed on a canvas can be modified independently in their properties, location and size at any time. Method call-backs can be directly bound to those objects, so interactive behavior can be easily implemented. Also, export to encapsulated postscript files is supported. Tk is certainly not a reasonable choice for developing a full-blown 3D-CAD application, but it has more than adequate performance for Gato. Today, Java's Swing library [16] used with JPython [17], a Python interpreter written in Java, which makes the Java libraries accessible from Python, might be an alternative.

The language features and the libraries and frameworks available yield a very *expressive* language. In comparison with an unpublished CATBox version, CATBOX++, developed in C++ with the graphical user interface cross-platform-library XVT [28], which was the industry-standard at that time, we were able to make the following observations: Moving to Python reduced the number of lines of code (LOC) by at least a factor of fifteen[3]. Currently Gato has about 6,000 LOC includ-

[2] Even color space conversion, used for increased distinguish-ability among colors of, say, labels and vertices, is among the functionality provided.

[3] We simultaneously added additional features during the re-implementation, making it difficult to obtain more than a lower bound.

ing the complete developer documentation. While LOCs is certainly just one relevant measure in software engineering, this reduction greatly facilitated re-factorization when needed and encourages contributions to sub-systems of the package (typically in the 500 LOC range).

Another important factor is the increased development speed due to avoiding laborious edit-compile-link-cycles and having to make code compliant with compilers on different platforms, as it is often the problem with C++.

3.3 The visualization engine

Having decided to present algorithms as executable Python code, we will now address the question of how to provide a suitable and safe environment for executing algorithms and visualizing their actions.

The algorithm engine, which consists mainly of a sub-class of Python's standard library debugger class, is the core of the visualization engine. Together with the standard library's `exec`-function, it provides the functionality to execute Python code from within an application, while giving the user control over the execution and allowing for reactions to events such as proceeding to the next line or calling functions.

To protect the main application from erroneous code in algorithms, we supply explicit private *local* and *global* name-spaces for algorithms. Name-spaces are implemented as dictionaries containing (*key*,*value*)-pairs. Objects in the main application are made available to algorithms on a need-to-know basis. Since graphs are often modified by algorithms, private copies are inserted into the name-space; for all other objects references are used. Also, for reasons of convenience, standardized shorter variable names for graphs and the 'animator', the visualization engine, are supplied.

Algorithms are stored in two files. In addition to the actual algorithm, the `*.alg`-file, containing the visible algorithm code, there is the so-called prolog, the `*.pro`-file. The prolog will be read and executed without user control before the algorithm is started.

The prolog serves a number of purposes. As we will discuss in detail in the next section, so-called animated data-structures (ADS) are used for coding the visualization. The visualization engine acts as a mediator between the ADSs 'living' in the algorithm engine and the object, in Gato always the `GraphDisplay`, providing the visual feedback. The ADSs are usually implemented in Gato. To facilitate rapid development and easy adaption, ADSs can also be implemented in the prolog. This supports changes to an ADS without touching Gato's code and without even leaving Gato.

The prolog also contains information about the algorithm, e.g., the problem it addresses, its history, complexity, and cues regarding its visualization, e.g. meaning of colors, in form of HTML-code for Gato's built-in simple HTML-viewer. It provides information to the visualization engine about suggested or default breakpoints and so-called *interactive* algorithm code lines. These are code-lines where user interaction

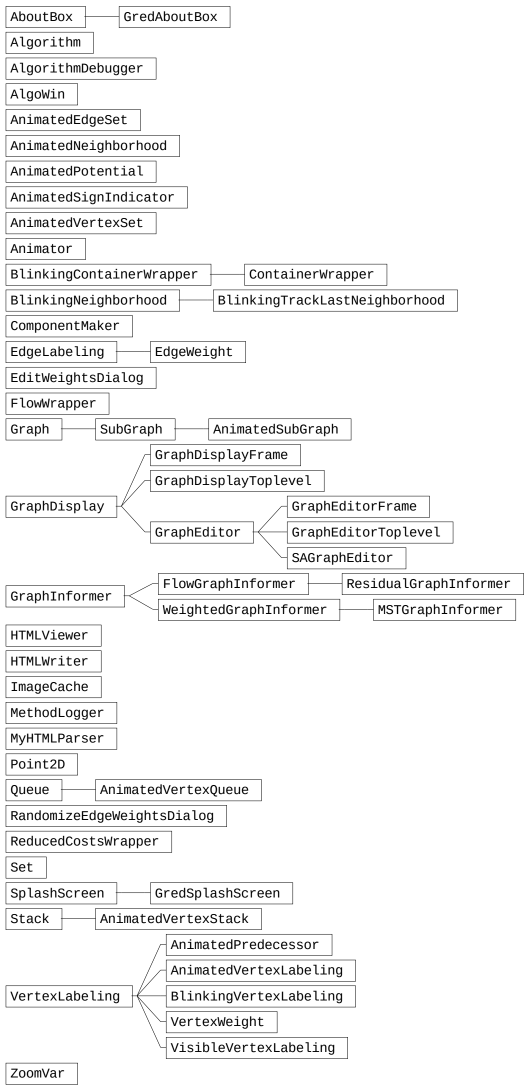


Fig. 6. A diagram showing the classes used in Gato. The shallow class hierarchy is due to the rich standard library.

```
################################################################
#
# This is part of CATBox (Combinatorial Algorithm Toolbox)
...
#
################################################################

# Options --------------------
breakpoints = [9]
interactive = [4]
graphDisplays = 1
about = """<HTML>
<HEAD>
<TITLE>Breadth-First-Search</TITLE>
</HEAD>
<BODY>
...
</BODY></HTML>
"""
pickCallback = lambda v, a=A: A.SetVertexAnnotation(v,"source")
PickVertex = lambda f=pickCallback: self.PickVertex(1,None,f)
Neighborhood = lambda v,a=A,g=G: AnimatedNeighborhood(a,g,v)
Vertices = G.vertices
visited = AnimatedVertexLabeling(A)
Q = AnimatedVertexQueue(A)

# End-of BFS.pro
```

Fig. 7. An example of the code in the prolog, BFS.pro, corresponding to the algorithm code in algorithm BFS.alg as shown in Fig. 5. The additional commands set default values and visual feedback for interactively picking the root of the BFS-tree. Header and HTML-code describing the algorithm are edited for brevity.

such as choosing a vertex or an edge is required. Other uses are the definition of auxiliary data-structures and functions, which are irrelevant to the understanding of the algorithm in question. Just as in the case of the name-spaces, easier and shorter variable names can also be defined in the prolog. The renaming is mainly done to make the initial learning step as easy as possible for students; often it consists of replacements of the form '`self.G.Vertices`' by '`V`'.

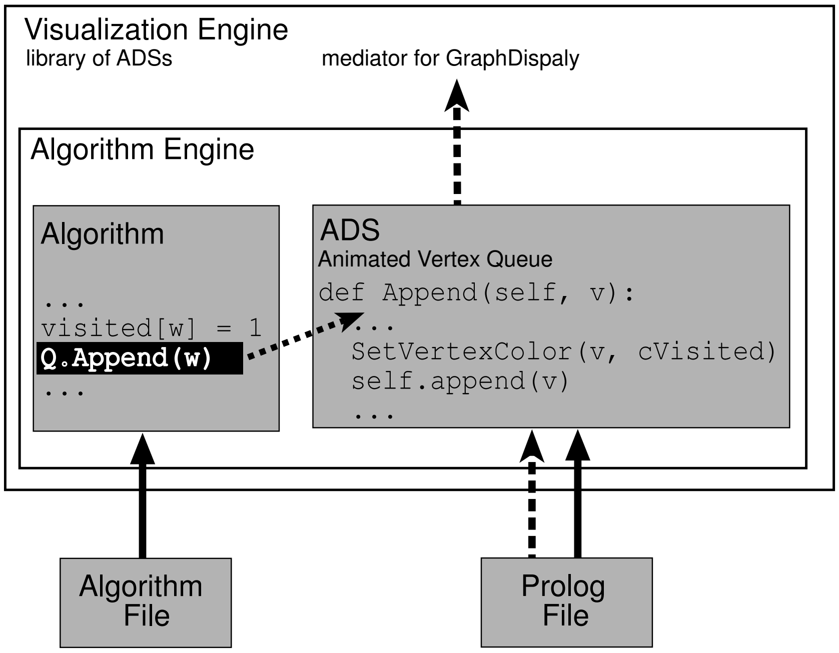


Fig. 8. A schematic view of the visualization engine and the objects involved in visualizing an algorithm. Dashed lines designate a control relation and solid lines data-flow.

4 Rule-based visualization

Consistency and an appropriate definition of interesting events, i.e., events that are crucial with regard to function of an algorithm, are of utmost importance for an educationally successful visualization. When sufficient care is taken that the *implementation* of the algorithm, no matter if it is written in pseudo-code or executable code as Gato's, corresponds in all details to the theoretical, mathematical *idea* behind it, those interesting events can always be defined in terms of changes to the data-structures used in algorithms. If not, our experience shows that this is an indication of divergence between theory and implementation.

A tight coupling of the visualization commands to changes to the data-structures yields a high level of consistency. We achieved the coupling by using the animated data-structures (ADS) mentioned in the previous section. ADSs are data-structures typically used in graph algorithms, e.g., queues, stacks, lists, and arrays with vertices as an index. They provide visualization whenever methods changing the contents of those ADSs are invoked. They can be refined to obtain a specific visual effect by the choice of instantiation parameters or by sub-classing. Implementing rules for the visualization in this way had several advantages. It is technically very easy to realize, assures a a high level of maintainability, it provides a high degree of fine-level control, and it enforces an exactly defined choice of interesting events and their visual feedback.

In the following, we demonstrate rule-based visualization using Dijkstra's algorithm as an example. Dijkstra's algorithm is a label-setting-algorithm for solving the shortest-path-problem [1] from one fixed root. It works by processing all vertices in a graph basically in a breadth-first-search manner while setting labels, which represent the distance from the root and the predecessor on the shortest path to the root for each vertex.

For Dijkstra's algorithm, vertices fall into four distinct categories:

1. Vertices not yet 'seen',
2. vertices 'seen', but not yet processed,
3. the active vertex, and
4. processed vertices.

During the run-time of the algorithm, vertices proceed through the different categories. They are never re-processed or become unseen. The software can determine which category a vertex belongs to by checking its status with regard to the central data-structure in the algorithm, a priority queue. The *theoretical* categories above correspond to the following *implementational* ones:

1. Vertices, which are not in the queue,
2. vertices in the queue,
3. the vertex most recently removed from the queue,
4. vertices, but the most recent one, removed from the queue.

Transitions between the different categories are tied to changes to the data-structure. A vertex becomes seen and is appended to the queue, when transitioning from category 1 to 2. The transition from 2 to 3 happens, when a vertex becomes 'active'; that is, when it is removed from the queue. Finally, an active vertex is moved into category 4 as soon, as another vertex is removed from the queue.

The categories can hence be easily visualized with the help of an ADS called `AnimatedVertexQueue`. Initially, all vertices are in category 1 and displayed with a fixed color. The method for appending a vertex to the queue also changes the color of a vertex, as does the removing of a vertex, which promotes the vertex to active status. The queue keeps track of the active vertex, retiring it to the appropriate color for category 4, when the next vertex is removed from it.

The second component in the visualization concerns the processing step. Here, all the neighbors of the active vertex are visited; i.e., the edge connecting the active vertex with its neighbor is traversed and the distance label of a neighbor is inspected. Here we can use an ADS called `AnimatedNeighborhood`, which creates a temporary highlighting of neighbors and edges, as the algorithm iterates over the complete neighborhood. Finally, the rooted shortest path tree is highlighted through use of an `AnimatedPredecessor`-ADS as the array for storing the predecessor label.

Rule-based visualization enables students to experiment with algorithms. They can either change existing algorithms, or by using the ADSs supplied, obtain visualizations for algorithms implemented from scratch with little additional effort. This is an important factor for the teaching effectiveness of algorithm visualizations [24].

Even the authors sometimes obtained surprising insights already for very simple algorithms. It seems to be an accepted fact, that Breadth-First-Search and Depth-First-Search (DFS) are basically the same algorithm, just using a queue in the former and a stack in the latter algorithm. During implementation we were rather surprised, when code obtained by the exchange of the data-structure in a straight-forward BFS-implementation looked 'strange' in the visualization. As it turns out, a somewhat artificial implementation is necessary to accomodate such an equivalence, illustrating slight differences between theory and implementation of an algorithm.

5 Software Engineering

Extreme programming [11] has become a very popular programming methodology over the last four years. It formalizes the processes which have been successfully used to create large software systems and is in some aspects an antipode to the classical software engineering techniques with their divide-and-conquer and "build up from building blocks" approaches. One fundamental difference is the acknowledgement of the fact that software, compared to, say, a bridge is a rapidly moving target.

We will just briefly mention the aspects of extreme programming that are most relevant to the development of Gato and software development in an academical environment in general.

– Integrate often, release frequently. We tried to have a working version of Gato at all times. Modules, which were developed independently, were integrated into the main application daily. The application would often miss parts of the functionality — the very first implementation could not display graphs, but 'visualized' color changes to vertices or edges as text output — but nevertheless *work*, allowing for hands-on experimentation as a basis for further design decisions.

- Add functionality as needed. A typical error in object-oriented development is to identify foundation classes, and implement those in the fullest generality possible, neverminding the actual need of the application at hand. Similarly, it is tempting to address all fathomable uses in application design. A restriction to what is needed now speeds up the development process and increases software quality. E.g., we choose to restrict the graphical display of graphs.
- Simplicity. Implement the most simple and not the fastest or most elegant solution. Optimization for speed should be done at the very end, when the application is working without errors.
- *Spike solutions* and frequent re-factorization. Spike solutions establish the feasibility of solving a technical problem by a sample implementation, separate from the main application. Introducing spike solutions to the main code basis and adding additional features lead to general code growth. Re-factorization, e.g., moving code to separate classes, adding new methods or classes for often-used tasks, was used to clean the code basis. For this tasks, the small number of LOCs helped tremendously.

6 Licensing Issues

We chose to release the different components of our project under different licenses, cf. Fig. 1. Gato and Gred are published under the Library GNU Public License (LGPL) for the following reasons: Our work has only been possible due to the large number of people contributing quality work to Python and Tk. We feel obliged to return this favor to the community in the most extensive way possible. A point in support: Gato already has been used for the development of 'AsIF — An Authoring System for Interactive Fiction' [3].

As we have seen from requests for updates and bug-fixes to a long-outdated CATBox-predecessor, software and especially educational software seems to have a life of its own and a life-span much larger than anticipated. From our point of view, the only way to address the issue of long-term, say, decades of, software support is to put it into the hands of the users.

Freedom to re-use your own code in further work can only be achieved when your code is free and protected through a water-proof license. As CATBox shows, commercial interests, in this case those of Springer Verlag, protected through copyrights can easily co-exist with free software, to the benefit of both parties.

On a more practical note, besides attracting a larger user base than commercial code, thus increasing the number of people exercising the code, free software also attracts other developers. A free exchange on a technical level regarding design issues is possible and very helpful especially in an academic environment with limited project group size.

7 Evaluation

So far, no formal evaluation studies have been done for Gato and CAT-Box. We do however have a range of experiences using parts of our course in a number of different settings, ranging from second-year business information majors to graduate level mathematics students.

A typical teaching scenario used Gato in the following form: At first, the algorithm was run to completion to give an overview. Then, usually stepping through the algorithm's commands one by one, the finer details were explained. Worst-case instances provided a very vivid picture illustrating the underlying theory. Students could use Gato outside the class-room on their own, and were encouraged to experiment with different instances to intesify their understanding.

In the advanced graduate level course mathematics students were offered extra credit for problems requiring implementation and visualization of non-trivial algorithms from scratch without tutoring in Gato or Python. We observed that the students in the class working on these optional problems were usually both more advanced and had prior programming experience. Their comments, gathered in informal interviews, indicated that they found the problems very challenging, but also the results very rewarding. For large-scale use in the classroom, we suggest to employ teaching-assistants to provide support for programming problems.

8 Outlook

There are several possible areas of expansion. The graphical side of the visualization has so far been very 'schlicht': edges are always lines, vertices always circles and vertex names are simply numbers. To help non-math or computer science majors to understand the graph model as a correct formalization of an application problem from their field, the inclusion of visual cues might be beneficial. Imagine building a phylogenetic tree in Biology and, say, displaying pictures of the species as the vertices in the tree.

Also, to address difficulties of understanding basic data-structures such as e.g., lists, queues and stacks, graphical inspectors are a viable option.

We can and will also extend the areas to which we apply Gato by addressing graph-based stochastic models. As a matter of fact, for Hidden-Markov-Models, the first author's main area of work, a specialized editor based on Gred, is under development [13]. This project will include an interface to a HMM-library developed at the ZAIK/ZPR providing an environment for researching and training HMMs. Markov Models, Bayesian networks etc. can also be addressed.

9 Acknowledgements

CATBox and Gato would not exist without Achim Bachem, who inspired the original idea. We are also very grateful to Sándor Fekete and Cristoph Moll, who contributed ideas and aided through helpful discussions in the design process.

Thanks to Torsten Pattberg, who put the finishing touches on the proof-of-concept algorithm implementations and implemented and visualized a lot of the algorithms included in CATBox from scratch.

Thanks also to Ramazan Buzdemir and Achim Gaedke, recent additions to our CATBox and Gato team. Ramazan Buzdemir implemented the graph layout algorithms in Gred and Achim Gaedke is working on Gred and its extension to an editor for Hidden-Markov-Models.

References

1. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows. Theory, Algorithms and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.
2. Alice - free, easy, interactive 3D graphics for the WWW. URL http://www.alice.org/.
3. AsIf - an authoring system for interactive fiction. URL http://www.videon.wave.ca/~krussell/asif/.
4. Marc H. Brown and Robert Sedgewick. A system for algorithm animation. In *Proceedings of ACM SIGGRAPH '84*, pages 177–186, July 1984.
5. Marc H. Brown and Robert Sedgewick. Interesting events. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization. Programming as a Multimedia Experience*, Cambridge, MA, 1998. The MIT Press.
6. CATBox - Combinatorial Algorithm Toolbox. URL http://www.zpr.uni-koeln.de/~catbox.
7. Computer programming for everybody. URL http://www.python.org/cp4e/.
8. P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. LEONARDO: a software visualization system. In *Proceedings of the 1-st Workshop on Algorithm Engineering (WAE'97)*, pages 146–155, 1997.
9. EDU-Sig: Python in education. URL http://www.python.org/sigs/edu-sig/.
10. Jeffrey Elkner. Using Python in a high school computer science program. In *Proceedings of the Eighth International Python Conference*. PSA, 2000. URL http://www.python.org/workshops/2000-01/proceedings/papers/elkner/pyYHS.html.
11. Extreme programming: A gentle introduction. URL http://www.extremeprogramming.org.
12. Gato - Graph Animation Toolbox. URL http://www.zpr.uni-koeln.de/~gato.
13. A hidden-markov-model editor. URL http://www.zpr.uni-koeln.de/~hmm.
14. Winfried Hochstättler. Algorithmische Mathematik WS 98/99. URL http://www.zpr.Uni-Koeln.DE/AlgoMat98_99/.

15. Scott E. Hudson and John T. Stasko. Animation support in a user interface toolkit: Flexible, robust and reusable abstractions. In *Proceedings of the 1993 ACM Symposium on User Interface Software and Technology, Atlanta, GA*, pages 57–67. ACM, November 1993.
16. The Swing connection. URL http://java.sun.com/products/jfc/tsc/index.html.
17. JPython home. URL http://www.jpython.org.
18. Bernd Kokavecz. Mit leichten Schritten in die objektorientierte Programmierung. URL http://www.b.shuttle.de/b/humboldt-os/python/.
19. Leonardo: A C programming environment for reversible execution and software visualization. URL http://www.dis.uniroma1.it/ demetres/Leonardo/.
20. Python language website. URL http://www.python.org.
21. Python library reference. URL http://www.python.org/doc/current/lib/lib.html.
22. John Stasko, John Domingue, Marg H. Brown, and Blaine A. Price, editors. *Software Visualization. Programming as a Multimedia Experience.* The MIT Press, Cambridge, MA, 1998.
23. John T. Stasko. The TANGO algorithm animation system. Technical Report CS-88-20, Department of Computer Science, Brown University, December 1988.
24. John T. Stasko. Using student-built algorithm animations as learning aids. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 29, 1997.
25. Andy Tai. The GUI toolkit, framework page. URL http://www.geocities.com/SiliconValley/Vista/7184/guitool.html.
26. Tcl/tk. URL http://www.tcltk.com.
27. Guido van Rossum and Jr. Fred L. Drake. Extending and embedding the Python interpreter. URL http://www.python.org/doc/current/ext/ext.html.
28. XVT. URL http://www.xvt.com/.

□