Agustin Forero
Professor Olaf Hall-Holt
CSCI 390A
Fall 2020

OleGraph

And the Importance of Graphically Interactive Collaboration

Abstract

Though several tools exist within the scope of public access to teach people how to use graph traversal algorithms (GTAs), little to none of them place heavy emphasis on graphically interactive collaboration (GIC) as a centerpoint for teaching -- that is, collaboration enabled by an effective graphical user interface. If employed with a thoughtful UI, GIC holds the potential to become an excellent driving force for users to learn GTAs through interactive software. Our research and development team (Rob Berger ‘21, Nick Dahlquist ‘21, and Agustin Forero ‘21) created a tool, OleGraph, hinged on the novelty and significance of this idea. We then narrowed down our work to teaching Dijkstra’s algorithm. Ultimately, test subjects responded positively to having access to a peer to converse with while navigating the algorithm, provided that adequate steps were taken to ensure they were not confused by the interface itself. Our team thus recommends GIC as a new paradigm for the development of GTA teaching tools.

Introduction

Since the early days of computer science education, plenty of attempts have been made to construct an effective teaching tool for the various graph traversal algorithms (hereafter abbreviated as GTAs) that dominate the field. Instructors have found it fit to explain various GTAs through the usage of graphical representations and interactive tools, and young programmers practice to ready themselves for technical interviews. Thus, effective algorithm teaching tools are in high demand. LeetCode[1] and HackerRank[2] serve as recently popular tools for practicing implementation, and many developers have tried their skills at creating flexible new tools for learning GTAs.

Most of these tools are created by programmers with heavily technical backgrounds, with the contextual knowledge needed to implement GTAs accurately. In this respect, there is no shortage of tools that display easy-to-understand graphics, or plot brilliant animations through various front-end tools like JavaScript and React.js. Unfortunately, it is extraordinarily hard to find GTA teaching tools that make graphically interactive collaboration (hereafter referred to as GIC) their centerpiece, where GIC refers to collaboration driven forward by a well-crafted graphical user interface. Most teaching tools available have the user working independently, watching how something like DFS or BFS traverses through a graph.

By negating the ability for collaboration to healthily boost a user’s learning ability, these tools have missed out on a concept that may define GTA teaching tools in the future. Accordingly, our group set out to create such a tool using C++ and React.js, designing it with the significance of GIC as a guiding force. We focused our efforts on Dijkstra’s algorithm specifically.

Background

A simple Google search will provide plenty of GTA teaching tools available on GitHub or directly in a browser. Some such as Programiz[3] or Tutorialspoint[4] contain step-by-step guides on GTAs like DFS running on specific, pre-rendered graphs, whereas others allow for increased user involvement, such as VisuAlgo.[5] None of these aforementioned tools incorporate any notion of collaboration, which has shown to be an invaluable tool within research regarding the teaching of GTAs, such as with J. Paul Gibson’s study on schoolchildren between Kindergarten and 12th grade (Gibson 38). In Gibson’s study, students were encouraged to deliberate amongst themselves when working through a shortest-path algorithm, rather than relying on the teacher for

[1] LeetCode.com
[2] HackerRank.com

[3] Programiz.com DFS
[4] Tutorialspoint.com DFS
[5] VisuAlgo.net DFS/BFS

help; this led to students finding answers through collective discussion and creation of counter-examples, even within groups of students that had never encountered such GTAs before.

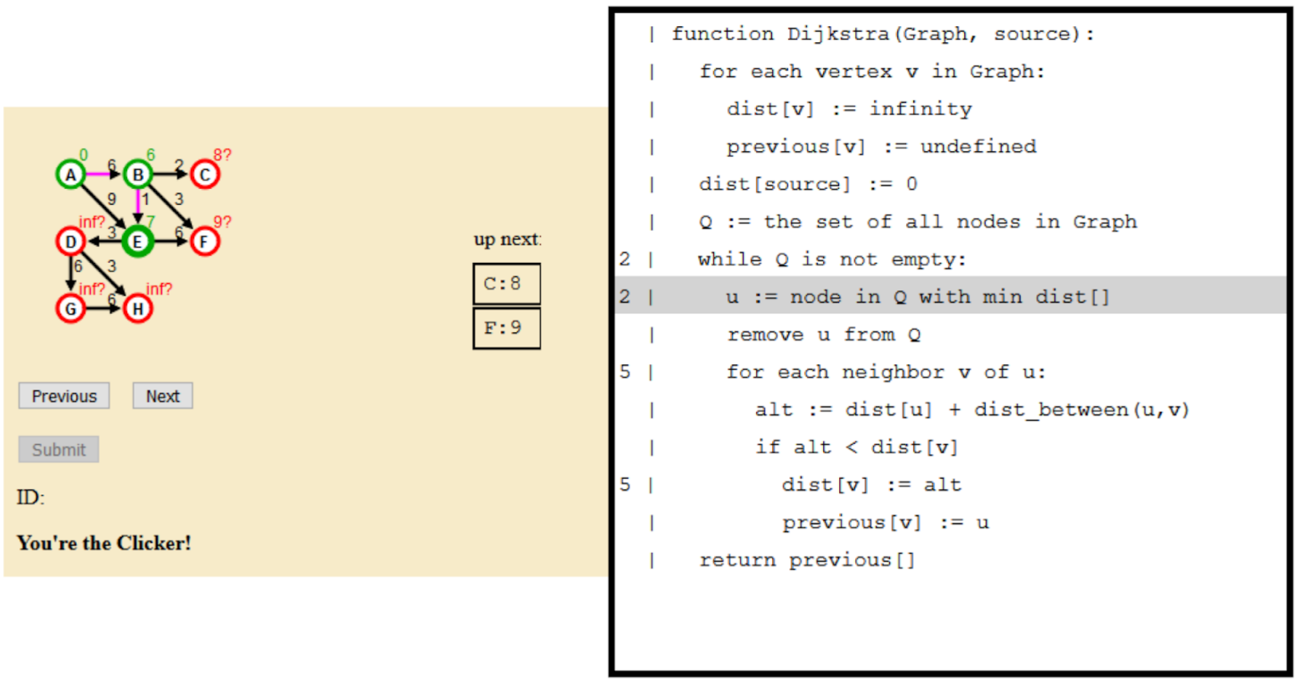
In 1995, professors at the University of Rome developed a collaborative GTA teaching tool, Java Algorithm visualiZer (JAZ), to help teachers showcase GTAs over the internet or intranet (Bongiovanni et al., 73). Much like our own tool, frames holding the different states of a graph during a GTA are held within a server, displayed concurrently to all clients on their own machines. However, the teacher controls all facets of the frames displayed; the user has no ability to move through the animation at their own pace, nor can they edit the structure of the graph or even regenerate it (77). Though the underlying technology used to implement JAZ approaches something that would allow for student-to-student collaboration, this particular implementation of a GTA teaching tool lacks such a feature. It revolves around a teacher-to-student relationship rather than a peer-to-peer learning environment. This disallows students to come to breakthroughs together, or challenge themselves by clarifying concepts that confuse their teammates.

In the fall of 2001, two professors at Auburn University's Computer Science Department wrote a teaching tool that closely resembles the objectives of our own work. Professor Teresa Hübscher-Younger and Professor N. Hari Narayanan created the Collaborative Algorithm Representations Of Undergraduates for Self-Enhanced Learning, or CAROUSEL, to allow students to collaboratively create their own representations of the recursive exponentiation algorithm and Quick Sort (Hübscher-Younger et al., 6). Students would create their own representations and share them with the class, gathering collective input from their peers to help facilitate cognitive growth. Ultimately, this worked; students had a promising 30% increase between their pretest and posttest scores on an evaluative quiz (9). There are two marked differences between their implementation and our own: they primarily focused their studies of collaborative learning on algorithms that alter arrays, while we focused on GTAs; additionally, we had a much smaller pool of students to draw data from. Despite these differences, it is apparent from their research that collaboration is highly effective in teaching students algorithms, a point critical for the successful execution of our own tool.

Finally, though academics have contended the viability of graphics for teaching students GTAs for decades, Professor Ladislav Végh and Professor Veronika Stoffová, both professors in Slovak universities, found exciting data with their own graphical representations of card-sorting algorithms. Within post-evaluation tests, the professors found a 53.1% increase in correctly answered algorithm statement combinations, i.e. "The algorithm always compares two neighboring elements in the array" and "The algorithm compares every element with all elements located behind it" (Végh et al., 133). Accordingly, there was a 39.7% decrease in incorrect pairings, and results were then proven to be statistically significant (134). However, when continuing to assess the students' ability to match pseudocodes to their respective sorting algorithms, they did not find a significant change in performance (137). Though interactive card animations weren't enough to instill a deep understanding of each algorithm within the subjects, Vegh and Stoffová's research proves that graphical representations can at least help students make sufficient strides in their goal of learning algorithms.

```
$ ./graphOutputDriver 4 -nodes 16 \
-g 4 -w 5 > data.json
```

Figure 1: a sample command-line execution of the C++ binary.



```
  | function Dijkstra(Graph, source):
  |   for each vertex v in Graph:
  |     dist[v] := infinity
  |     previous[v] := undefined
  |   dist[source] := 0
  |   Q := the set of all nodes in Graph
2 |   while Q is not empty:
2 |     u := node in Q with min dist[]
  |     remove u from Q
5 |     for each neighbor v of u:
  |       alt := dist[u] + dist_between(u,v)
  |       if alt < dist[v]
5 |         dist[v] := alt
  |         previous[v] := u
  |   return previous[]
```

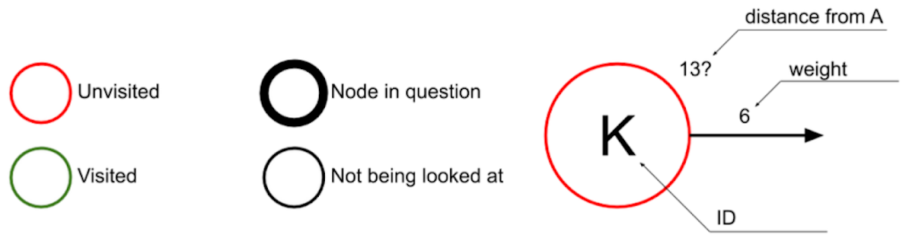


Figure 2: the OleGraph UI.

Approach

OleGraph primarily utilizes two languages to implement a collaborative GTA teaching tool: C++ and React.js, for the back- and front-end respectively. A server is installed and hosted on a St. Olaf College lab machine using Yarn[6], a package manager created by Facebook. After compiling and running the driver for the graph generation, each individual frame containing every iterative state of the graph is outputted to a JSON file, subsequently copied to a directory that Yarn uses to broadcast the data to each user.

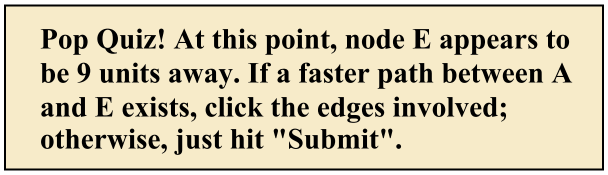
Depending on the flags inputted by our team, the C++ binary outputs JSON-based frames containing the changing nodes, edges and weights that define the graph. For example, to randomly generate a grid with 16 nodes (4 rows of 4 columns) with random edge weights between 1 and 5, a member of our team could run the command seen in Figure 1.

Our design philosophy for the implementation of each GTA was to implement the algorithm exactly as we would normally, only stopping at key points of the algorithm to output the graph's current state. Within our implementation of Dijkstra's, this meant outputting the graph's current state at four key points: when the next closest node was visited, for each neighbor of the aforementioned closest node, whenever a shorter path was found to a certain node, and when the algorithm returned to the top of the `while` loop.[7]

This makes certain that students using the tool can understand, with absolute and meticulous detail, each individual operation involved with Dijkstra's decision handling. Class attributes like `Graph::isBold`[8] help track exactly which node is being considered by the algorithm.

[6] Yarn's Official Homepage

[7] graphAlgos.cpp, line 97.
[8] graphMainHeader.hpp, line 19.



Pop Quiz! At this point, node E appears to be 9 units away. If a faster path between A and E exists, click the edges involved; otherwise, just hit "Submit".

Figure 3: a sample pop quiz question.

User Interaction

OleGraph's UI prompts students to interact with three main features of the page: the graph, pseudocode and supplemental key. Students utilize the "Previous" and "Next" buttons to move through the frames of Dijkstra's animation, with each frame containing states spawned from four key lines on the right. Then, while students move through these frames, they are prompted to complete a task assigned by a pop quiz displayed on the left. These assessments, such as in Figure 3, ask users to update distances when shorter paths are found, and select edges that compose shortest paths between the head and a given node. Students are also asked to decipher which node would be moved to next, so that they could update neighboring distances from that point. For navigation, the graph's `div` features several features visible in Figure 2, including amendable distance annotations to the top-right of each node and color-coding between black and blue to signify if an edge has been selected by a user. Students can also sketch on the `div` itself to point out ideas to other students, allowing classmates to draw an arrow or star near an object of importance. The `div` also features a display of the priority queue, signifying which nodes are next in line to be selected by the algorithm.

Finally, for both the sake of avoiding network-based race conditions, and simplifying teamwork for the students, OleGraph only allows for one student to navigate the graph at a time. This student is aptly called the "Clicker", and they are randomly assigned their role at the beginning of OleGraph's runtime. If a Clicker fails to answer a quiz question correctly, clicking responsibilities are passed to the next student, allowing for all members of a team to take turns asking and receiving input for decision-making. As a student described in a post-usage interview, if they were confused on a facet of a question, it was up to the entire group to pool their knowledge to help continue the activity along. This strengthens the group's understanding as a whole, as weaknesses held by one student are usually made up for by contributions from another.

To help students predict the actions of the GTA, OleGraph also features pseudocode on the right side of the page (Figure 2). The UI highlights whichever line is currently being processed for the bolded node in question. It also keeps count of how many times each line has been encountered, as per the leftmost column displayed in the pseudocode `div` (Figure 2). At the bottom of the page, students may also refer to a key, reducing ambiguity in the graph's representation of the GTA.

By implementing each algorithm with a step-by-step output of JSON data, and combining these frames with a UI that emphasises clarity and interactiveness, we created a tool that allows students to learn by bouncing ideas between one another. Students focus less on navigating a confusing interface and more on compiling knowledge together to learn as a whole, a strength not seen in most GTA teaching tools today.

```
while Q is not empty:
  u := node in Q with min dist[]
  remove u from Q
  for each neighbor v of u:
    alt := dist[u] + dist_between(u,v)
    if alt < dist[v]:
       dist[v] := alt
       previous[v] := u
```

Figure 4: a section of Dijkstra's Algorithm pseudocode.

Results

Two groups, both containing one junior and one senior, tested two slightly different versions of OleGraph. The first tested a version without the key, and the second had both the key and a preemptive tutorial graph to use beforehand.

Both groups were exposed to a 3-minute tutorial on Dijkstra's[9] prior to starting usage, and both were given the same graph to navigate. A student within the first test group had technical difficulties with connecting to the server, and thus piloted the tool indirectly through a streamed display of one of OleGraph's developers.

Results between the two groups had great disparity. Considering the lack of tutorial or key, the first group struggled to navigate the UI, and thus found it difficult to change weights or click edges when OleGraph would prompt. After testing with this group and witnessing confusion related to the UI, our team added the tutorial and key to the tool, so that users wouldn't be barred from learning simply because of a confusing interface.

With these critical changes implemented, the second group had a far superior experience with the tool. The key cleared up any ambiguity with the information displayed in the graph's `div`, and the short tutorial graph beforehand taught the subjects how to answer the different pop quizzes. Once these students moved past the tutorial and onto the main graph, they conversed among themselves to answer quiz questions as they arose. At one point, the students worked together to correctly guess that a certain distance would *not* update, as line 11 in Figure 4 was not fulfilled as `true`. After the successes of the second test group, it was apparent that a user-friendly UI is an essential first step in implementing effective GIC.

Ultimately, what defined a valuable learning experience with OleGraph was GIC enabled by our focus on effective team-building. The studies conducted by Professors Hübscher-Younger, Narayanan, Végh and Stoffová fit this hypothesis well. Hübscher-Younger and Narayanan conveyed how collaboration may assist students to share ideas and overcome challenges together. Végh and Stoffová proved how graphical teaching tools for algorithms help students solidify the basics, even if they don't immediately gain advanced knowledge of whatever algorithms are taught. OleGraph combines these two fundamentals. Collaboration empowers students to work together and learn as a team, and graphics help students grasp the fundamentals in the first stages of mastering algorithms. Graphically interactive collaboration is significant as it weaves these proven methods together, which is exactly why our team worked to make it the defining feature of OleGraph.

Conclusion

The ability of students to collaborate with one another while learning graph traversal algorithms electronically is an unexplored yet lucrative paradigm. Though most tools do not currently incorporate collaboration as key features of their softwares, GIC can help students learn GTAs with speed and efficiency, and holds the potential to be highly significant in GTA teaching tools in the future. Effective GIC hinges on a UI meticulously crafted to dispel any ambiguity in the traversal's progress. Once proper user experience is achieved by GTA teaching tool developers in the future, our research and development team implores developers to consider GIC as a defining centerpiece by which algorithmic knowledge can be attained by up-and-coming software engineers.

Acknowledgements

I'd like to thank Rob and Nick for being excellent collaborators for this term, always quick to fix whatever must be fixed and research whatever must be researched. I'd also like to thank Professor Olaf for his advising on this project, and the anonymous test subjects for their time volunteering.

[9] Dijkstra's algorithm in 3 minutes by Michael Sambol

Sources

- Bongiovanni, Giancarlo, Pierluigi Crescenzi, and Gabriella Rago. "JAZ: Java Algorithm visualiZer. a multi-platform collaborative tool for teaching and testing graph algorithms." *Sixth International Conference in Central Europe on Computer Graphics and Visualization*. 1998.
- Gibson, J. Paul. "Teaching graph algorithms to children of all ages." Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education. 2012.
- Hübscher-Younger, Teresa, and N. Hari Narayanan. "Constructive and collaborative learning of algorithms." *ACM SIGCSE Bulletin* 35.1 (2003): 6-10.
- Végh, Ladislav, and Veronika Stoffová. "Algorithm Animations for Teaching and Learning the Main Ideas of Basic Sortings." *Informatics in Education* 16.1 (2017): 121-140.