Vamonos: Embeddable Visualizations of Advanced Algorithms

Brent Carmer
Oregon State University
carmerb@eecs.oregonstate.edu

Mike Rosulek
Oregon State University
rosulekm@eecs.oregonstate.edu

***Abstract*—We present Vamonos: a new framework for algorithm visualization, designed from the beginning to support embedding, interaction, and unlimited scope. Visualizations are executed entirely client-side on any device that supports a modern web browser; they can be embedded into any website or online textbook. Users can specify breakpoints, watched variables, provide inputs to the algorithm (e.g., by drawing a graph using a mouse), and be prompted for interaction by the visualization. The core framework supports any algorithms and data structures that can be implemented in Javascript. We have implemented a wide range of visualizations of advanced algorithms topics, including dynamic programming and graph algorithms (e.g., spanning tree, max-flow, bipartite matching algorithms).**

I. Introduction

Algorithm visualizations (AVs) have long been appealing to both educators and students alike. Yet, they have not been as widely adopted as one would expect. There has been a great deal of research into effectiveness of AVs in general (e.g., [11], [7]) as well as usability aspects of AV systems which hamper or facilitate their adoption (e.g., [9]).

At the same time, web-based educational resources have become more sophisticated, due to overall advances in web/browser technology and facilitated by the increased popularity of massively open online courses. Specifically, AVs have evolved from static animations (famously [1]) to interactive Java/Flash applets (e.g., [16]), and recently, full-featured web applications (e.g., [6]) that take advantage of standard modern browser features.

Now, with increased effort being placed into high-quality online, *interactive* textbooks (e.g., [14], [17]), interactive visualizations are becoming replacements for the static figures of paper textbooks. Indeed, online textbooks/lessons seem to be the ideal venue for AVs, as [20] found that AVs should be paired with motivational goals and instruction, and [11] found that AVs are most effective in a homework/self-study (rather than exam) setting.

II. Related Work

There are a number of ongoing AV projects that share our goal of leveraging recent developments in webrowsers.

The Javascript Algorithm Visualization Library (JAVL) [10] provides a framework for educators to create browser-based visualizations, with the stated goal of supporting online textbooks. In particular JAVL was created to support the OpenDSA project [19] [5] which seeks to provide a free interactive online data structures and algorithms textbook. JAVL visualizations are highly interactive, with built-in mechanisms (”proficiency exercises”) for quizzing the user. JAVL can follow along as the user simulates the algorithm, providing feedback when mistakes are made. JAVL is very high-level in the sense that it favors visualizing more conceptual rather than procedural aspects of algorithms.

The recent PythonTutor framework [6] is a popular and very successful example of an AV tool that can be embedded into online textbooks. In contrast to JAVL, PythonTutor's focus is on visualizing low-level, introductory programming concepts (Python syntax, flow control, arrays, references, calling conventions) to novices. Indeed, the AV landscape skews heavily towards introductory material (e.g., sorting algorithms). Shaffer et al. [18] surveyed a wide range of AVs and found that coverage of advanced topics (e.g., dynamic programming, network flow) is lacking, by comparison.

III. Our Work

Inspired by the success of PythonTutor, we set out to make a web-based AV framework that supports more advanced algorithmic topics, which are typically expressed in very high-level pseudocode and involve more sophisticated data structures. Since web browsers already have the capability to execute arbitrary Javascript, we also wanted to forego PythonTutor's reliance on a server-side component. In the context of embedded visualizations in textbooks, client-side visualizations allow the material to be learned away from an active internet connection (or after the server-side maintainer has let the service lapse).

In addition, like PythonTutor, we wished to provide a close coupling of a visualization to the code itself - in our case pseudocode. This provides the end-user with more flexibility and transparency than offered with a visualization in JAVL. With Vamonos, users can see exactly how a line of pseudocode changes the underlying data structure. In contrast, JAVL visualizations provide an English description of what occurs in each frame out of context of the rest of the program. Furthermore, JAVL does not support the visualization of loops, recursion, and other control structures as they work inside an algorithm.

Ihantola et al. [8] give a taxonomy of AV *"effortlessness"* — features of a system that reduce barriers for educators to adopt and maintain visualizations. Among our goals was to achieve a high degree of effortlessness. We highlight some important features of Vamonos, using this taxonomy:

Integrability denotes qualities that make the system attractive to use and easy to set up, encompassing installation, customization, platform independence, internationalization, documentation, interactive prediction support, course management support, and integration of hypertext.

Being web-based, Vamonos has very high integrability. Visualizations require no installation barrier beyond a modern web browser (for either AV producers or consumers) so are inherently cross-platform. At the extreme, all of the page data (HTML, Javascript, CSS) of a visualization can be inlined into a single HTML file. We also inherit all of the power of modern HTML for layout, CSS for stylization, as well as accessibility features (i.e., visualizations are natively scalable). And, of course, the visualizations can be easily embedded.

Scope denotes how widely one can apply the system, encompassing difficulty in installing and running the system, and educational context (how many topics can the system be used for?). We created Vamonos to have essentially unlimited scope within the domain of algorithms, supporting both low-level concepts (arrays, calling conventions, basic flow control) and very high-level ones (e.g., graphs and geometric algorithms). The core framework can, at least in principle, support any algorithm & data structure that can be implemented in Javascript.

Interaction denotes how deeply the consumer can modify a visualization. Lawrence et al. [12] showed that students' ability to design their own inputs to algorithms is a key factor in AV effectiveness. In Vamonos, the consumer can specify her own inputs to an algorithm: even graphs. Further, the consumer can set breakpoints and watched variables, and step forward and back through the algorithm's execution. Meta-aspects of the execution are also displayed (e.g., active line of execution, complete call stack). We also support several kinds of interactive quiz capabilities. (See Section IV)

IV. Use Cases / Vignettes

We introduce Vamonos by presenting use cases that take advantage of its highly interactive nature and broad scope.

A. Ford-Fulkerson Algorithm & Eliciting Interesting Inputs

Consider the Ford-Fulkerson algorithm for computing maximum flow. The input to such an algorithm is a flow network — a complicated data structure consisting of a directed graph, with integer edge labels / capacities, and two distiguished vertices (designated source and sink).

Using a Vamonos visualization of Ford-Fulkerson, a student can draw any such flow network using the mouse (see Figure 1). S/he can then execute the algorithm on the graph of her/his choice and see how the max flow is iteratively obtained and how the residual graph is updated.

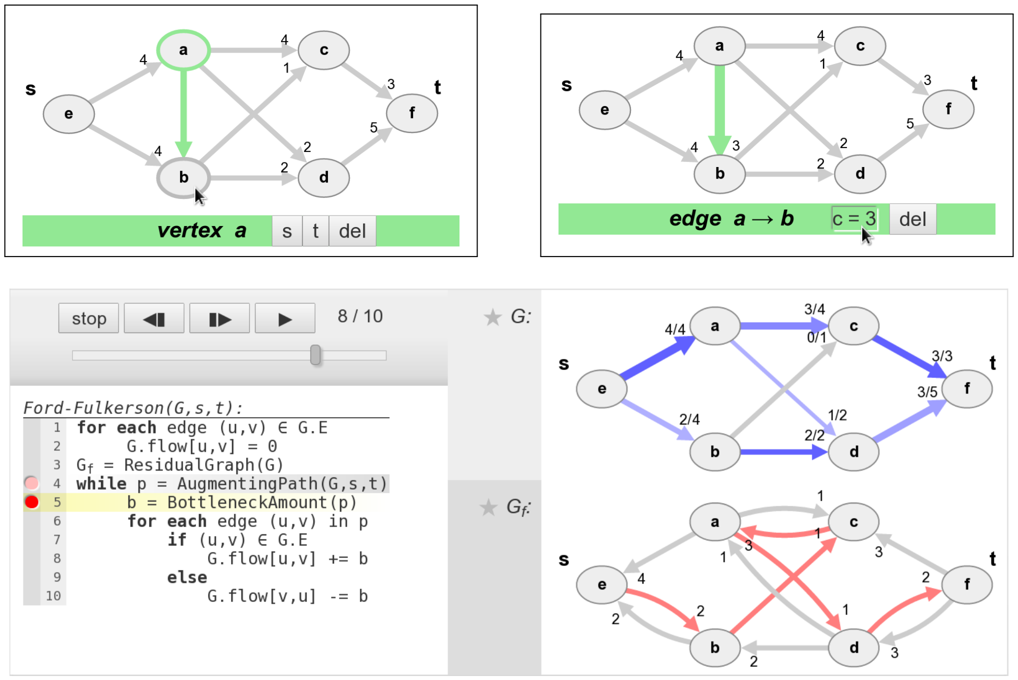
One particularly challenging concept about the Ford-Fulkerson algorithm is that during the course of the algorithm, flow does not monotonically increase along every edge. Consider the following application for a Vamonos visualization embedded in a web-based textbook. After discussing why it might be necessary to reverse the flow along some edge, the textbook can present an embedded visualization and prompt the reader to draw an example input on which such flow-reversal is necessary. Vamonos supports this kind of input-based quizzing, where the user is asked to provide an input that triggers a certain internal condition to happen; for example:

- An edge case in an algorithm is triggered (e.g., there is a tie among shortest paths).
- Dijkstra's algorithm proceeds by maintaining upper bounds on each vertex's distance from the source. These distance estimates are reduced until they converge on the true distance. The user can be prompted to input a graph for which some vertex has its distance estimate modified at least 5 times.

B. Depth-First Search & Topological Sort

In Vamonos, graphs can be manipulated with the mouse not only to draw the input to an algorithm, but to rearrange the graph during the AV's display mode (this is an optional setting, as it would not be advisable to rearrange vertices while running an algorithm that uses the graph's planar *embedding*).

We provide a ready-made visualization of depth-first search (DFS), and, following the textbook of Cormen et al. [4] (CLRS), we label each vertex with its "start" and "finish" time (i.e., the time at which it entered and left the stack). For acyclic graphs, CLRS prove that sorting



```
Ford-Fulkerson(G,s,t):
for each edge (u,v) ∈ G.E
    G.flow[u,v] = 0
Gf = ResidualGraph(G)
while p = AugmentingPath(G,s,t)
    b = BottleneckAmount(p)
    for each edge (u,v) in p
        if (u,v) ∈ G.E
            G.flow[u,v] += b
        else
            G.flow[v,u] -= b
```

Fig. 1. Vamonos is a web-based algorithm visualization framework that supports high-level data structures like graphs. This series of screenshots shows a Vamonos visualization of the Ford-Fulkerson max-flow algorithm. The two upper screenshots show the modal interface by which a user can input a graph using the mouse. In the upper-left screenshot, the vertex a is modally selected, and a green potential edge ab is shown when the user hovers over b. Buttons below can delete vertex a or set it to be source/sink. In the upper-right screenshot, an edge ab is modally selected, and it can be either deleted or its label/capacity changed. In the bottom screenshot, a user has finished drawing the input graph G (labeled G), set two breakpoints, and stepped through 8 of 10 frames of the algorithm's execution. The red path in the residual graph G_f shows the augmenting path p. In the flow network G, the color denotes saturation: edges with zero flow are gray; saturated edges are completely blue; other edges are an appropriate mixture.

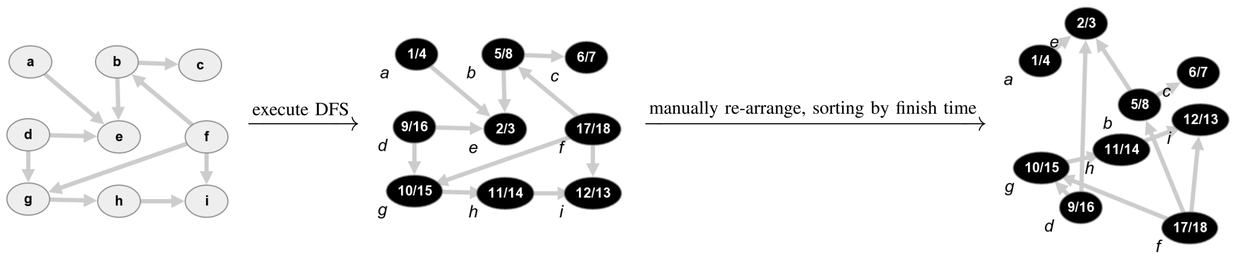


Fig. 2. A visual demonstration of a topological sort based on DFS. Vertex labels represent "start / finish" times. In the final image, it can be visually verified that all edges point upward, hence the ordering corresponds to a topological sorting of the vertices.

the vertices by decreasing finish time yields a topological ordering.

As an in-class demonstration or a guided self-study lesson, a user can execute DFS on a graph so that each vertex is labeled with its finish time. S/he can then drag vertices around to place them in order according to finish time. It can then be verified *visually* that the resulting embedding has all edges pointing in the direction of decreasing finish time. (See Figure 2).

C. Breadth-First Search & Bipartiteness

Other properties of graphs are illustrated well using a similar approach. For instance, consider executing breadth-first search (BFS) on a graph to label each vertex with a distance value. Then arrange the vertices into columns, one column for each possible distance value, in increasing order. In the resulting graph, it can be visually verified that for every edge uv, we have $|u.d - v.d| \leq 1$. That is, no edge can skip a "height/level" in the BFS tree.

Along the same lines, we have included a visualization for a variant of BFS that colors vertices red and blue, where the colors are chosen to disagree along an edge as it is traversed by BFS. If an edge is encountered with monochromatic endpoints, the algorithm stops highlighting the problematic edge.

When an edge has monochromatic endpoints, an

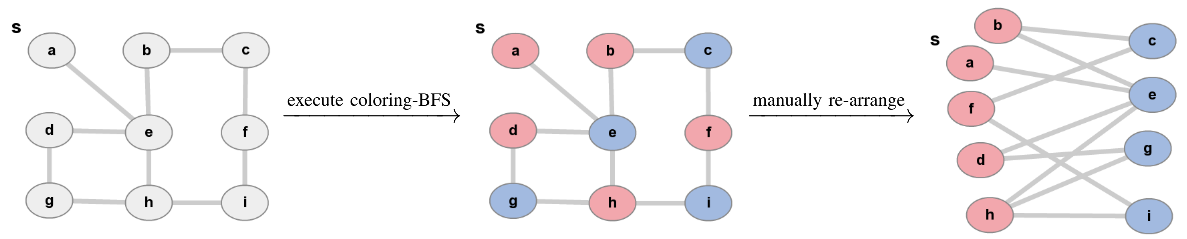


Fig. 3. A visual demonstration of a bipartiteness algorithm based on BFS. After manually rearranging, it can be visually verified that all edges have one red and one blue endpoint.

odd-length cycle can be easily found visually (hence, the graph is not bipartite). If the algorithm terminates without a monochromatic edge, then the graph can be rearranged with red vertices to the left and blue vertices to the right. Students can then verify visually that all edges cross from left to right; hence, the graph is indeed bipartite. (See Figure 3.)

V. Design & Implementation

The Vamonos framework consists of CSS and Javascript libraries (compiled from Coffescript), along with popular Javascript library jQuery and graphical library D3 [2]. Visualizations (of which we provide a collection of examples) are HTML files that load these libraries.

A. *Visualization Engine & Lifecycle*

Vamonos visualizations are managed by an instance of a **Visualizer** object. A **Visualizer** object is instantiated with a **callback** that implements the algorithm to be visualized (more details of this are discussed in Section V-C) and a collection of **widget** objects which implement the user-facing elements of the visualization (described immediately below).

Vamonos visualizations are modal, with two modes. By default, a visualization begins life in **edit mode**. (It is possible to make a visualization that exists only in display mode. This may be useful for visualizations with fixed inputs.) In this mode, widgets are collecting input from the user: inputs to the algorithm, and other parameters of the algorithm execution like breakpoints and watched variables.

When the user is satisfied, she uses the controls to finish edit mode. At this point, the **Visualizer** object will execute the algorithm and periodically collect snapshots of the algorithm's state, called **frames**. Each frame contains the contents of all variables, call stack, active pseudocode line, etc. Indeed, the main responsibility of the **Visualizer** object is to constantly maintain a representation of the call stack, manage the various procedure-local name spaces for variables, and determine the appropriate times to take a snapshot of the current state.

Then the visualization enters **display mode**, in which the widgets display the contents of these frames. Using on-screen controls, the user can navigate through this collection of frames to see the execution of the algorithm. While the user has a sense of being able to pause/rewind the execution of an algorithm, it is important to note that the collection of snapshots is generated all at once and then fixed (until the next edit/display lifecycle). The user cannot change variables, breakpoints, watched variables on the fly. This is primarily because our method for implementing the actual visualized algorithm is not re-entrant, so does not support stepping backwards; again, see Section V-C.

B. *Widgets*

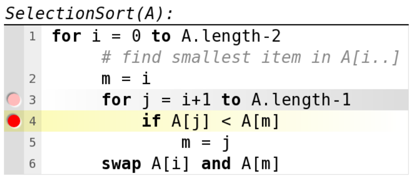
Generally, a **widget** is any part of the visualization that interacts with the user. The **Visualizer** object provides a standardized API for widgets. In edit mode, widgets may set variables and other execution parameters (like breakpoints and watched variables). In display mode, widgets receive the current frame to display. Widgets are also the mechanism by which the **Visualizer** changes modes and navigates through the collection of frames.

We give a summary of the various widgets implemented so far:

Pseudocode displays the pseudocode of the algorithm. In edit mode, allows users to specify breakpoints by clicking in the gutter (mimicking the interface of a standard IDE). In display mode, highlights the pseudocode line that just finished executing, and the line about to execute.

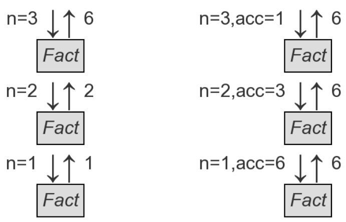
We point out that the lines of code in the pseudocode widget are purely cosmetic, and have no bearing on what code is actually *executed* to generate the visualization (see Section V-C for more details). Hence, the pseudocode widget can be used to show a very high-level or a very low-level algorithm.

In the example below, lines 4 & 5 are set as breakpoints. In this frame, line 4 is about to execute, and line 3 has just finished.



```
SelectionSort(A):
1  for i = 0 to A.length-2
       # find smallest item in A[i..]
2      m = i
3      for j = i+1 to A.length-1
4          if A[j] < A[m]
5              m = j
6      swap A[i] and A[m]
```

CallStack, in display mode, shows a dynamically updating representation of the call stack. The procedure name, arguments, and return values are all displayed. Vamonos supports algorithms with multiple procedures, and also supports tail calls. Below are two call-stack visualizations for different factorial functions (both showing the final result).



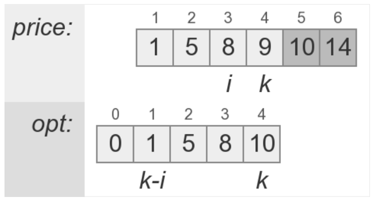
Array displays an array data structure. In edit mode, allows WYSIWYG editing of an input array. In display mode, shows the contents of an array in the given frame. The widget also supports various styling features: the array can be annotated with array index variables from the algorithm, and CSS rules can be applied to cells based on index variables. Array cell zero can also be hidden, to mimic 1-indexed arrays common in many texts.

Below is an example from a dynamic programming visualization that uses the recurrence

$$opt[k] = \max_i \left\{ price[i] + opt[k-i] \right\}.$$

Hence, we show variables `k` and `i` indexing into `price`, we shade the not-yet-used portion of `price` (beyond index `k`), and we show *expression* `k-i` indexing into `opt`.

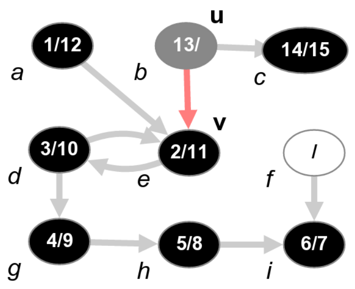
```
new Widget.Array({
  varName:     "price",
  showIndices: ["k", "i"],
  cssRules:    [[">", "k", "shaded"]],
  ...
new Widget.Array({
  varName:     "opt",
  showIndices: ["k", "k-i"],
  ...
```



Graph displays a graph data structure. Edges can be directed/undirected, and optionally weighted. In edit mode, the widget area is a canvas on which the user can draw a graph using mouse/touch actions. **Graph** widgets support a wide variety of informative styling in display mode: vertex labels, edge labels, and CSS styles can be determined according to arbitrary vertex/label attributes set by the algorithm. Variables that take on *vertices* as values can be displayed as annotations similar to how array indices annotate the appropriate cell in an **Array** widget.

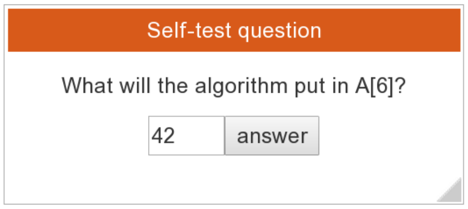
In the depth-first-search example below, a directed graph is shown. The `v.color` attribute determines the CSS class of the vertex; the contents of the vertex are computed as "`v.start` / `v.finish`". Vertex variables `u` and `v` are displayed as annotations *à la* array indices. The edge `u` to `v` is highlighted red.

```
new Widget.Graph({
  vertexLabels: {
    inner: function(v){
      return v.start + "/" + v.finish;
    },
    sw: function(v){ return v.name },
    ne: ["u", "v"],
  },
  vertexCssAttributes: {
    color: ["white", "gray", "black"],
  },
  edgeCssAttributes: { red : "u->v" },
  ...
```



UserQuiz creates a modal user dialog in display mode that cannot be dismissed until the correct answer is given. The quiz is parameterized by a *condition* (on which frames will the dialog be displayed), a *question* to be asked, and a correct *answer*. Each of these parameters can be computed as a function of the current frame. In the following example, a dialog will be displayed whenever algorithm variable `i` > 5 and the frame was taken just before pseudocode line 4 was to be executed.

```
new Widget.UserQuiz({
  condition: function(frame) {
    return frame.i > 5
      && frame._nextLine == 4;
  },
  question: function(frame) {
    return "What will algorithm put "
        + "in A[" + frame.i + "]?"
  },
  answer: function(frame) {
  ...
```



What will the algorithm put in A[6]?

42

ResultProperty provides an easy way to display a message on the page conditioned on some event happening in the execution of the algorithm. This can be used to make visualizations with self-tests of the form "find an input that causes behavior X from the algorithm." (See Section IV.)

C. Internals & Creating Visualizations

An Vamonos visualization is concretely an HTML document that imports the Vamonos library. This HTML file contains Javascript invocations of the **Visualizer** and other widgets, as well as HTML layout information and accompanying page content.

1) Laying out the widgets: Most widgets are user-facing, and draw elements to the page. The relative layout of these widgets is completely unconstrained (though we provide default HTML/CSS templates), so that a visualization's collection of widgets can be placed anywhere within a document.

When generating a visualization, the producer typically specifies the visual location of a widget by inserting an empty `<div>` HTML tag into the page source, e.g.:

```
<div id="some-widget"></div>
```

Then the associated widget object instance is attached to this tag in its constructor arguments:

```
new Widget.Array({
  container: "some-widget"
  ...
```

2) Algorithm callbacks: An AV producer must specify an implementation of the algorithm which is to be run. Naturally, this can be done by passing a callback to the **Visualizer** object. However, there are two major challenges: (1) Stopping the callback mid-execution to take snapshots of its internal state. (2) Providing the **Visualizer** object with access to the internal variables of the algorithm.

We solve these challenges in the following way. For concreteness, consider the following Javascript implementation of Horner's polynomial evaluation method:

```
function Horner(P,x) {
  var res = 0;
  var i = P.length - 1;
  while (i >= 0) {
    res = res * x + P[i];
    i = i-1;
  }
  return res;
}
```

We solve the first challenge by having the the **Visualizer** pass a callback to the algorithm. By convention we name this argument "_". The algorithm should invoke this callback with a number n to indicate that it is about to execute *pseudocode* line number n. This provides a hook to return control flow to the **Visualizer** object so it can generate a new frame if desired. Again we stress that the correspondence between the actual algorithm implementation and lines of pseudocode is arbitrary. Conveniently, the **Visualizer** can also raise an exception if it suspects an infinite loop in the algorithm.

Addressing the second challenge, when the **Visualizer** executes the algorithm callback, it first sets Javascript special global `this` to be a "scope object" in which local variables are stored. As long as the algorithm refers to `this.varName` rather than a local variable `varName`, both the algorithm and the **Visualizer** will have common access to these variables. To further cut down on the extra syntax imposed, we use Javascript's somewhat obscure `with` statement. Assuming that `this.i` exists in the scope object, all references to variable `i` inside the scope of `with(this){...}` will really refer to `this.i`.

In the end, our example from before becomes:

```
function (_) {
  with (this) {
  _(1); res = 0;
        var i = P.length - 1;
  _(2); while (i >= 0) {
  _(3);    res = res * x + P[i];
          i = i - 1;
        }
  _(4); return res;
  }
}
```

This callback can then be passed as a parameter to the **Visualizer** object via:

```
var viz = new Visualizer({
  widgets: [ ... ],
  algorithm: function (_) { ... }
```

3) Multiple procedures: An AV producer can specify an algorithm with several named procedures. Instead of passing in a single callback, an associative array of name$\rightarrow$callback values should be given.

The **Visualizer** engine adds a wrapper around each callback which facilitates the calling conventions in several ways. This wrapper is where the engine can maintain its representation of the call stack. The wrapper also handles argument passing, which is not positional but done by name. Finally, the wrapper populates the `this` "scope object" with the (wrapped) procedures, so that the procedure-calling syntax is natural when writing the visualization. A special argument to these wrapped procedures is used to specify tail-call optimization of the call stack.

4) Watched variables via serialization: Vamonos supports generating snapshot frames whenever a variable is changed. We achieve this functionality for completely arbitrary data types by using the browser-standard JSON (Javascript object notation) serialization features. Each watched variable is serialized to a JSON string, and these strings can be compared to detect when a variable is changed. Hence, one can set a graph variable to be watched, and a frame will be captured whenever any of the constituent vertices has an attribute changed.

Serialization also allows an easy way to save and share user-provided inputs. When a user enters an input to an algorithm, an encoding of that input is placed in the URL. Students can use this mechanism to submit answers to homework questions (of the form "find an input that causes behavior X") or share with peers.

5) Inlining visualization: One of the benefits of basing our framework on native, client-side web technologies is that visualizations can be contained in a single page. We have avoided use of images in our default styles, and as such, the visualizations require only HTML, CSS, and Javascript resources. These can be inlined into the main HTML page, and we provide a simple script to do so. Without taking any effort to minimize the size of our source files, a complete standalone Vamonos visualization (including all prerequisite libraries) is less than 1 megabyte.

VI. Ongoing Work

So far, our focus has been implementing a fully-featured core system. Our next step is to add levels of abstraction facilitating AV production by non-experts. Ultimately, we envision a workflow in which a user compiles a high-level pseudocode-like language into a Vamonos visualizations, similar to PythonTutor [6] and PCIL [13]. Students themselves could then create fully functional Vamonos visualizations without needing to interact with HTML and Javascript boilerplate.

A particularly interesting direction is to explore the extent to which "visualizable aspects" of high-level code can be automatically deduced from the source code, as demonstrated in [15]. It seems likely that a compiler would be able to infer which variables/expressions are used as array indices. However, other more sophisticated run-time properties could be automatically inferred and inform the visual presentation. For example: if an array index i monotonically traverses the array, and only indices at most i are ever accessed, then the portion of the array beyond i can be shaded (we hard-code such visual rules already).

A formal evaluation of the current system's efficacy is also needed, particularly when used in an embedded form in an electronic textbook. User study results would help guide our ongoing development and addition of features.

Acknowledgements

The authors are supported by NSF Award 1149647.

References

[1] R. Baecker, D. Sherman, and U. of Toronto Computer Systems Research Group. Dynamic graphics project. sorting out sorting., 1981.

[2] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2301–2309, 2011.

[3] T. Camp, P. T. Tymann, J. D. Dougherty, and K. Nagel, editors. *SIGCSE*. ACM, 2013.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009.

[5] E. Fouh, D. A. Breakiron, S. Hamouda, M. F. Farghally, and C. A. Shaffer. Exploring students learning behavior with an interactive etextbook in computer science courses. *Computers in Human Behavior*, 41:478–485, 2014.

[6] P. J. Guo. Online Python tutor: embeddable web-based program visualization for CS education. In Camp et al. [3], pages 579–584.

[7] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *J. Vis. Lang. Comput.*, 13(3):259–290, 2002.

[8] P. Ihantola, V. Karavirta, A. Korhonen, and J. Nikander. Taxonomy of effortless creation of algorithm visualizations. In R. J. Anderson, S. Fincher, and M. Guzdial, editors, *ICER*, pages 123–133. ACM, 2005.

[9] V. Karavirta, A. Korhonen, and P. Tenhunen. Survey of effortlessness in algorithm visualization systems. In *3rd Program Visualization Workshop*, pages 141–148, July 2004.

[10] V. Karavirta and C. A. Shaffer. Jsav: the javascript algorithm visualization library. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 159–164. ACM, 2013.

[11] C. M. Kehoe, J. T. Stasko, and A. Talor. Rethinking the evaluation of algorithm animations as learning aids: an observational study. *Int. J. Hum.-Comput. Stud.*, 54(2):265–284, 2001.

[12] A. W. Lawrence, A. N. Badre, and J. T. Stasko. Empirically evaluating the use of animations to teach algorithms. In *VL*, pages 48–54, 1994.

[13] B. Malone, T. Atkison, M. Kosa, and F. Hadlock. Pedagogically effective effortless algorithm visualization with a PCIL. In *IEEE FIE*, pages 1–6, 10/2009 2009.

[14] B. N. Miller and D. Ranum. Beyond PDF and epub: toward an interactive textbook. In T. Lapidot, J. Gal-Ezer, M. E. Caspersen, and O. Hazzan, editors, *ITiCSE*, pages 150–155. ACM, 2012.

[15] J. Mornar, A. Granic, and S. Mladenovic. System for automatic generation of algorithm visualizations based on pseudocode interpretation. In Å. Cajander, M. Daniels, T. Clear, and A. Pears, editors, *ITiCSE*, pages 27–32. ACM, 2014.

[16] T. L. Naps. Jhavé: Supporting algorithm visualization. *IEEE Computer Graphics and Applications*, 25(5):49–55, 2005.

[17] D. Pritchard and T. Vasiga. CS circles: an in-browser python course for beginners. In Camp et al. [3], pages 591–596.

[18] C. A. Shaffer, M. L. Cooper, A. J. D. Alon, M. Akbar, M. Stewart, S. P. Ponce, and S. H. Edwards. Algorithm visualization: The state of the field. *TOCE*, 10(3), 2010.

[19] C. A. Shaffer, V. Karavirta, A. Korhonen, and T. L. Naps. Opendsa: beginning a community active-ebook project. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, pages 112–117. ACM, 2011.

[20] J. T. Stasko, A. N. Badre, and C. Lewis. Do algorithm animations assist learning?: an empirical study and analysis. In S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel, and T. N. White, editors, *INTERACT*, pages 61–66. ACM, 1993.