Visualizing Minimax, Alpha-Beta Pruning and Constraint Satisfaction Problems on the Web

Alexander Costa **ALEXANDER.COSTA@UGA.EDU**
Department of Computer Science, University of Georgia

ABSTRACT

Classical artificial intelligence (AI) algorithms like Minimax and Alpha-Beta pruning for adversarial search problems and Backtracking (BT), Forward checking (FC), and Maintaining Arc Consistency (MAC) for constraint satisfaction problems are integral to any introductory artificial intelligence course. In spite of their ubiquity, the methods for teaching these algorithms often center around a block of abstract, and at times elusive, pseudocode which may contain a number of undefined functions or constructs. Consequently, these pseudocodes and accompanying explanations are a difficult framework from which to learn and understand these classical algorithms to any enduring degree. Frequently, visualizations are sought after to supplement learning of these algorithms by providing a more intuitive framework from which to construct a lasting understanding. While visualizations of many basic tree and graph searching algorithms exist, there are few which cover the specific aforementioned algorithms. In this work we present a selection of easily accessible and intuitive web-based interactive visualizations of the Minimax and CSP algorithms for use in an introductory-level academic environment.

Index Terms: *algorithm visualization, minimax, alpha-beta pruning, constraint satisfaction problems, CSP, backtracking, forward checking, arc consistency*

I. INTRODUCTION

There are a number of algorithms central to any level of education in artificial intelligence and which are covered in virtually all introductory level artificial intelligence course. Among these classical algorithms are the adversarial search algorithms Minimax and Alpha-Beta pruning and the constraint satisfaction problem (CSP) solving techniques Backtracking (BT), Forward Checking (FC), and Maintaining Arc Consistency (MAC). Though they are not prohibitively complex, the learning of these algorithms often constitute a student's first foray into the world of search algorithms and, for many new students, the task of developing an enduring understanding these algorithms is of considerable difficulty. In spite these algorithms' ubiquity, the pedagogical techniques employed to teach these algorithms are frequently unintuitive, inaccessible, or inefficient.

Most often, students are presented with a block of high-level pseudocode (which may lack a number of undefined functions or constructs) and an accompanying explanation covering one or two toy examples of the algorithm at work. Though the presentation of a formal definition is, of course, critical to the teaching of these algorithms, the pseudocode and toy examples alone are often not enough to furnish students with a lasting impression of these algorithms. The pseudocode presented is generally too abstract and elusive to commit to memory, and the toy examples explored are often too trivial and leave students with an incomplete and ungeneralizable understanding of these algorithms.

Even though these teaching methods get the job done, and with time could provide a thorough enough understanding, they could easily be improved with the use of interactive computer visualizations. Visualization is a powerful technique for the learning of abstract information and constitutes the perfect learning tool for these

classical algorithms whose search spaces can all be visualized as a graph or tree structure. In providing visual constructions of these algorithms' steps and search spaces, students are presented with a more concrete and intuitive framework around which to base their understanding. Visual representations are easier to commit to memory than pseudocode, and visualizations themselves deliver the same benefits as the exploration of toy examples and more since they may be extended beyond the purely trivial. Interactive visualizations facilitate even greater advantage as the student is equipped with the ability to define search spaces to their liking, explore all different kinds of examples, and investigate any particular misunderstandings.

The rest of this paper is organized into a number of sections II-IV. Section II describes existing implementations of adversarial search and CSP algorithm visualizations as well as their strengths and shortcomings. Section III describes our implementations of adversarial search and CSP algorithm visualizations, as well as their strengths and shortcomings. Section IV provides a conclusion and a discussion of the ways in which the current work could be extended.

II. RELATED WORKS

Algorithm visualization tools can generally be split into two categories: web-based tools, which are accessible over the internet and render in an internet browser, and non-web-based tools which are downloaded and run on a client machine. In this section we limit our purview to live, web-based implementations due to the inaccessibility and unpopularity of non-web-based tools. Students are generally uninterested in the downloading of programs and accompanying software packages merely for the visualization of these algorithms.

Though many visualization programs exist for classical graph searching techniques likes depth-first search and breadth-first search, visualization programs for adversarial search algorithms and CSP solving techniques are few and far in between, or in the case of CSP solving techniques, completely nonexistent.

In the case of adversarial search algorithms two popular implementations exist: one by José Manuel Torres (Torres) [1] and one by Leandro Ricardo Neumann, et al. (Neumann) [2]. Both of which implement the Minimax algorithm as well as Alpha-Beta pruning.

TORRES MINIMAX VISUALIZATION

The Torres program presents the search space of the Minimax algorithm as tree under a control panel used to define the structure of the tree. The program uses a textual input for tree construction, meaning search spaces are defined by two strings of numbers: a list of numbers defining the number of children of each node and a list of numbers defining the utilities (or heuristic values) of each leaf node. Once a search space is constructed the user may step through the Alpha-Beta pruning algorithm one instruction at a time or opt to run the entire algorithm in an animated sequence. The Torres implementation only allows you to run the Alpha-Beta pruning algorithm and not the simple Minimax algorithm alone. A screenshot of the Torres program is given in Figure 1.

Visually, the program is of mediocre quality and the tree definition mechanism is poorly defined and hard to understand, often times leading to the construction of invalid trees or tree formations for which the user did not intend. An incomprehensible "message" is given which presumably defines the tree in a structured text format. When large trees are constructed, nodes often occlude one another, making the visualization hard to follow. MAX states are indicated by upwards pointing triangles and MIN states are indicated by downwards pointing triangles, following the generally accepted convention.

During algorithm execution, current game states are highlighted in yellow and state utilities are presented to the left of each node. Alpha and beta values are also presented to the left of each game state, indicating the values of alpha and beta at the time of that state's evaluation. There are cases when the alpha and beta values are not visually updated (though the algorithm still functions correctly), and the presentation of alpha and beta values at each

state is minorly misleading since alpha and beta are global to the entire algorithm and each state does not have its own associated alpha and beta value. If a state is pruned during execution, the edge connecting that state and its parent is marked with a red X. After the algorithm has finished executing, the optimal path for MAX is highlighted in blue. The user may not specify that MIN goes first.

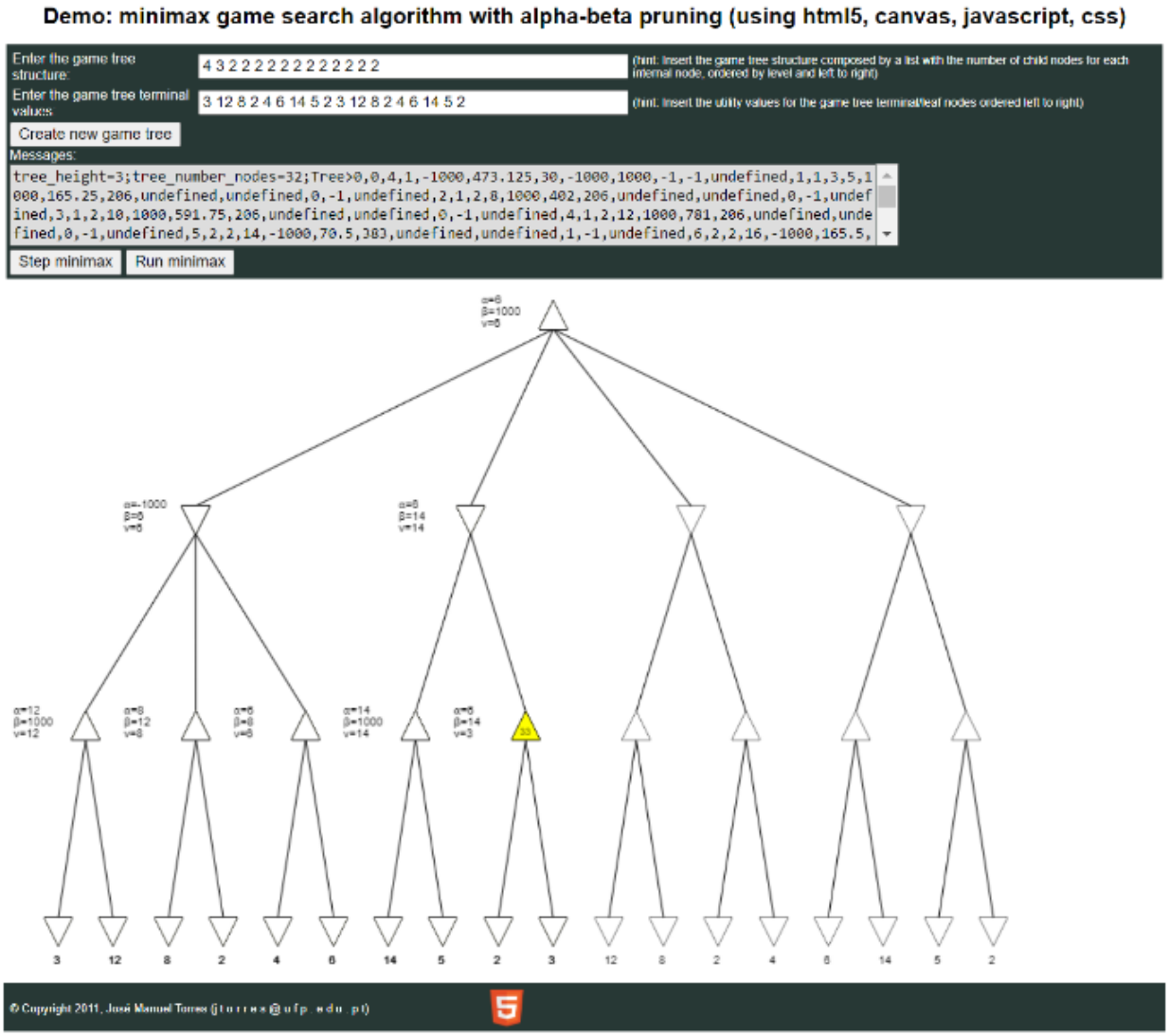


Figure 1: Screenshot of the Torres visualization.

Overall, the Torres program gets the job done, but the textual input for tree construction is extremely unintuitive and difficult to use, there are a number of issues with the presentation of alpha and beta values as mentioned above, and the program lacks any explanatory or pedagogical material about the Minimax or Alpha-Beta pruning algorithms.

NEUMANN MINIMAX VISUALIZATION

The Neumann program presents the search space of the algorithm as a tree under a control panel for executing and stepping through the algorithm instructions. The program uses a graphical, click-based input for tree construction, meaning the search space is constructed by clicking on nodes to add child nodes or edit utility values. Once a search space is constructed the user may choose to execute

either the simple Minimax algorithm or the Alpha-Beta pruning algorithm after which they may step through each instruction of the algorithm or skip to the end result of the program. The program also allows users to step backwards through the algorithm.

Visually, the program is of better quality than the Torres program but is still lacking in many areas. Large trees can be constructed without node occlusion, but wide trees will occlude the MAX/MIN labels presented at each layer of the tree. The tree definition mechanism is much more intuitive than that of the Torres program, but at the cost of efficiency. It takes quite a long time to construct large trees with many clicks and repeated tasks, though a default example can be generated. MAX and MIN states are both presented as circular nodes, going against accepted conventions, and instead indicated by labels to the left and right of each tree layer. A screenshot of the Neumann program is given in Figure 2.

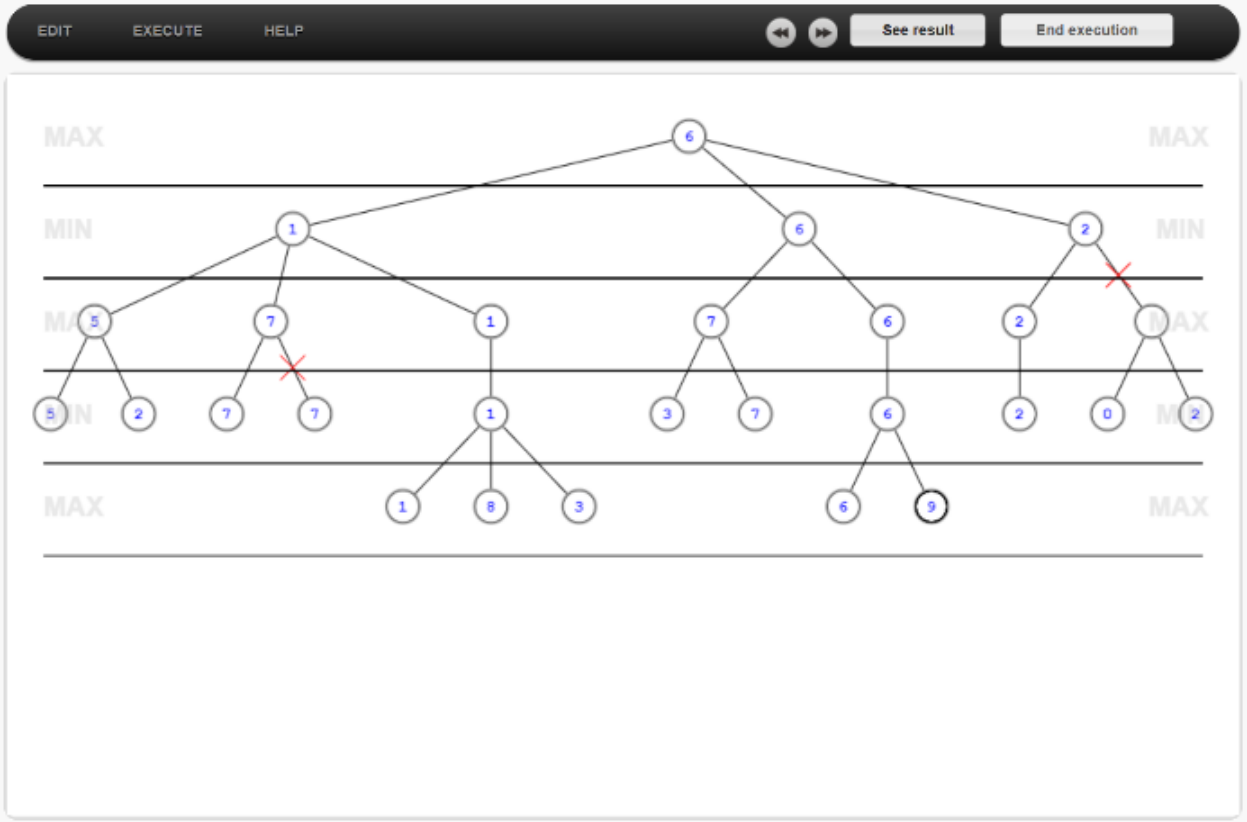


Figure 2: Screenshot of the Neumann visualization.

During algorithm execution, current games states are highlighted by a red border and state utilities are presented in the center of each node. Alpha and beta values are not visually represented during execution of the Alpha-Beta pruning algorithm. If a state is pruned during execution, the edge connecting that state and its parent is marked with a red X, just as in the Torres program. After the algorithm has finished executing, the optimal path for MAX is not indicated and the user must follow the child states of the root with the same utility as the root. The user may not specify that MIN goes first.

Overall, the Neumann program also gets the job done and works accurately, but the graphical tree definition input, while more intuitive, is extremely slow to use. The program also lacks visual indicators of important information like the values of alpha and beta during Alpha-Beta pruning and the optimal path for MAX after execution. The program lacks any explanatory or pedagogical material about

the Minimax or Alpha-Beta pruning algorithms, as well. In our work we shall seek to ameliorate many of the shortcomings of these two works as well as combine their strengths into one program.

As previously mentioned, our investigation yielded no live, web-based CSP interactive visualizations, so our discussion of related works concludes here.

III. OUR WORKS

In this work we aimed to create two interactive, web-based visualization tools: one for Minimax and Alpha-Beta pruning and one for the Backtracking (BT), Forward Checking (FC), and Maintaining Arc Consistency (MAC) solving techniques for CSPs.

TECHNOLOGIES USED

Because of their high familiarity, accessibility, and ease-of-use, we opted to create a web-based program, meaning it is universally accessible to anyone with an internet browser and internet connection and renders directly in a user's internet browser. Our program is implemented mostly in JavaScript using the React user interface library, with a small amount of additional HTML and CSS code and is hosted on the author's personal website (https://alexcostaluiz.com/visualizations/).

MINIMAX PROGRAM

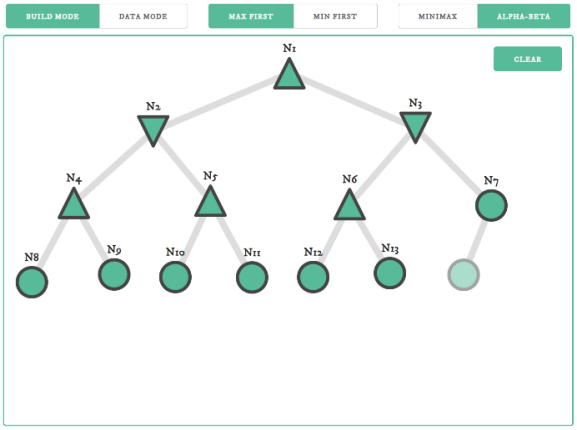
For the Minimax and Alpha-Beta pruning visualization program, hereby referred to as the Minimax program, we aimed to create a similar but enhanced version of the programs discussed in Section II, bridging their respective strengths while avoiding their shortcomings and also pursuing new ideas for enhancement. Important design choices to considered while constructing this program include the following:

- How best to accept input for tree construction.
- How best to visualize the search space (incl. node shapes, labels, edges, utility values).
- How best to indicate the different instructions of the Minimax and Alpha-Beta pruning algorithms.
- How best to provide helpful explanatory material which might aid a student in their understanding of these algorithms.

INPUT FOR TREE CONSTRUCTION

To the first consideration, we opted to follow suit of the Neumann program and use a graphical (or click-based) input for tree construction. To avoid the efficiency shortcomings of the Neumann program, we implemented a number of changes which speed up construction times and limit unnecessarily repetitive tasks. First we split the tree construction and utility specification into two distinct phases of the visualization. By doing so, we can avoid any small, difficult-to-click dropdown menus on node clicks and instead directly perform the desired task based on the visualization phase. Furthermore, during tree construction, child-nodes-to-be follow the cursor as they are placed in such a manner that allows for speedy construction of successive nodes.

a)



b)

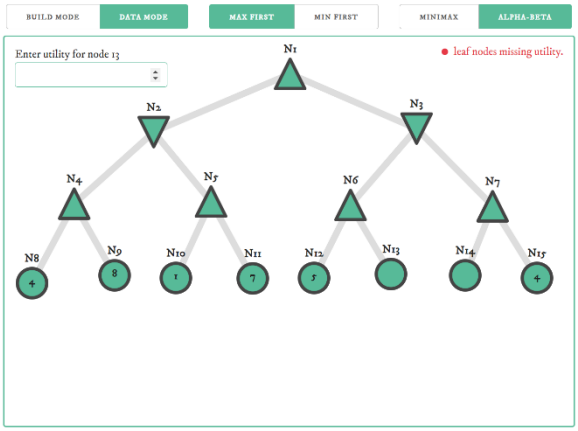


Figure 3: a) An example of our Minimax program during the tree construction mode. In the top left corner are mutually exclusive buttons BUILD MODE and DATA MODE which allow users to toggle between tree construction and utility specification modes. The child node of N7 is currently

being placed. Its slight transparency indicates that it does not yet exist, and its position follows the user's cursor making it easy for users to place nodes wherever they desire, and the creation child nodes of the node being currently placed require no mouse movements after the current node is placed. b) An example of our Minimax program during the utility specification mode. The single input text box located at the top left accepts input for all leaf nodes and automatically updates to the next leaf node with a missing utility after a user enters a utility for the previously selected leaf node.

Our last optimization in this category was a streamlined utility specification procedure which uses only one text box and automatically selects successive leaf nodes to be supplied a value. Effectively, the user need only to type n numbers, where n is the number of leaf nodes, with no clicking or use of the cursor. Overall, our tree construction procedure takes on a hybrid form of both the textual and graphical forms seen in Section II, maintaining the efficiency of the textual form while also supplying the user with an intuitive and fail-safe experience. Figure 3 presents examples of each of these optimizations.

SEARCH SPACE VISUALIZATION

To the second consideration, we opted for simple, convention-conforming visuals for the search space. Like the Torres and Neumann programs, we visualize the search space with a tree structure placed under a control panel. Like the Torres program, we identify game states with the conventional shapes: an upwards pointing triangle for MAX states and a downwards pointing triangle for MIN states. Leaf nodes are drawn as circles to avoid conflating them with the idea of a MAX or MIN state. Node labels are provided above each node to help distinguish each state from one another. Edges are simple grey lines which may take on other colors during algorithm execution. Lastly, node utilities are rendered in the center of each node. Figure 4 presents an example of a fully constructed tree.

COLOR CODING FOR ALGORITHM STEPS

To the third consideration, we follow suit of the Torres and Neumann programs by highlighting nodes when they are relevant to the current instruction of the algorithm. Unlike the existing programs, however, we extend this highlighting method with a color-coded scheme wherein different colors indicate different types of instructions of the algorithm.

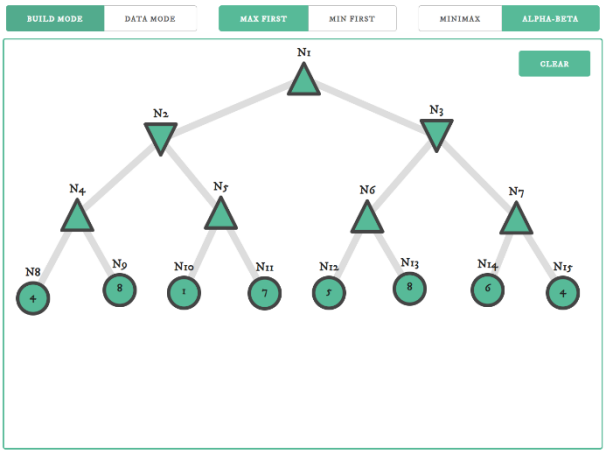


Figure 4: An example of fully defined tree in our Minimax program. Notice the clear shape distinction between MAX, MIN, and leaf nodes.

For example, a currently selected game state's node is highlighted in yellow, but a node whose utility is being updated is highlighted in blue. The full color code is as follows:

Yellow ⇒ a selected game state
Blue ⇒ a state whose utility is being updated
Pink ⇒ leaf node whose utility is being evaluated
Purple ⇒ two nodes being compared by utility

Not only does this color code scheme offer more information about the state of the algorithm to the user than the existing programs, but after repeated use of the program, users may become familiar with the meaning of each color, allowing for quick and intuitive understanding of any state of the visualization. Edges also support a degree of color coding as pruned edges are colored red, the edge connecting two nodes that are being compared is colored purple, and the final optimal path for the root node is colored green. For Alpha-Beta pruning, a single display of the current alpha and beta values in the top left corner of the visualization window is provided. This display is color coded as well, appearing blue when either alpha or beta are being updated to a new value and purple when alpha and beta are being compared to one another. The use of a single display for alpha and beta helps to solidify the understanding that alpha and beta are global to

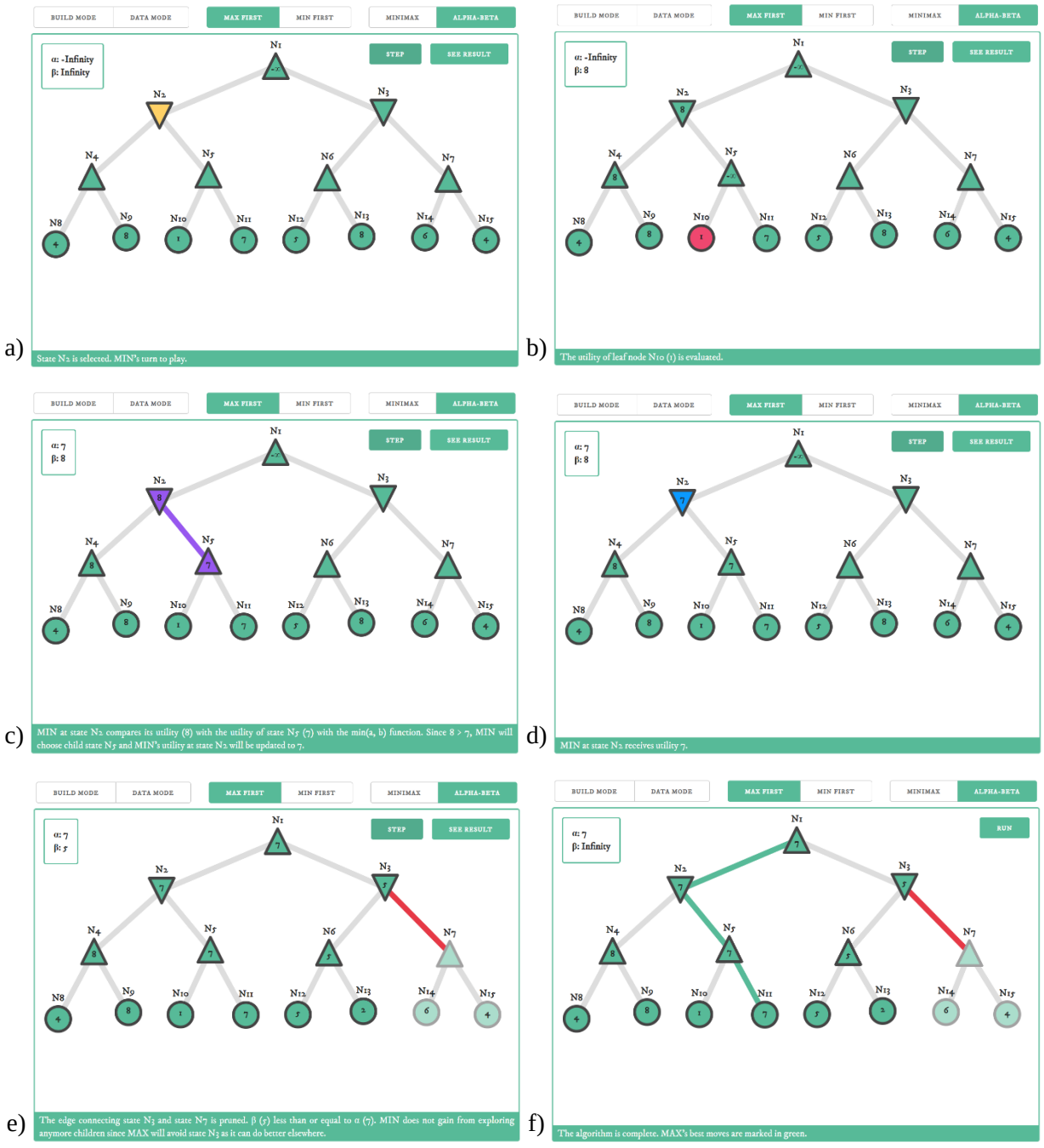


Figure 5: a) An example of a currently selected node. Here N2 is highlighted yellow, as the game state at N2 is currently being considered. b) An example of a leaf node being evaluated. Here N10 is highlighted pink. c) An example of two nodes being compared. Here nodes N2 and N5 and the edge connecting them is highlighted purple to indicate that they are being compared by MIN. d) An example of a node's utility being updated. Here the utility of state N2 is updated to 7 since the minimum of N2's successor states is 7. e) An example of a pruned edge. Here the edge connecting nodes N3 and N7 is highlighted red to indicate that there was no use for MIN to explore the children of state N3 any further. f) An example of an optimal path for MAX. Here the edges connecting nodes N1 to N11 are highlighted green.

the entire algorithm and not a property of each game state. Figure 5 presents examples of color-coded elements during algorithm execution.

PEDAGOGICAL TOOLTIP SYSTEM

To the fourth consideration, we implemented a tooltip system wherein explanations of each algorithm instruction as well as the motivation behind the more obscure ones are given at the

bottom of the visualization window during algorithm execution. So as the user steps through each algorithm, they are presented with a description of what is actually occurring behind the scenes, and in cases where the meaning of an operation is harder to grasp, an explanation of the purpose and consequences of the operation may also be included. Figure 6 presents examples of our tooltip system.

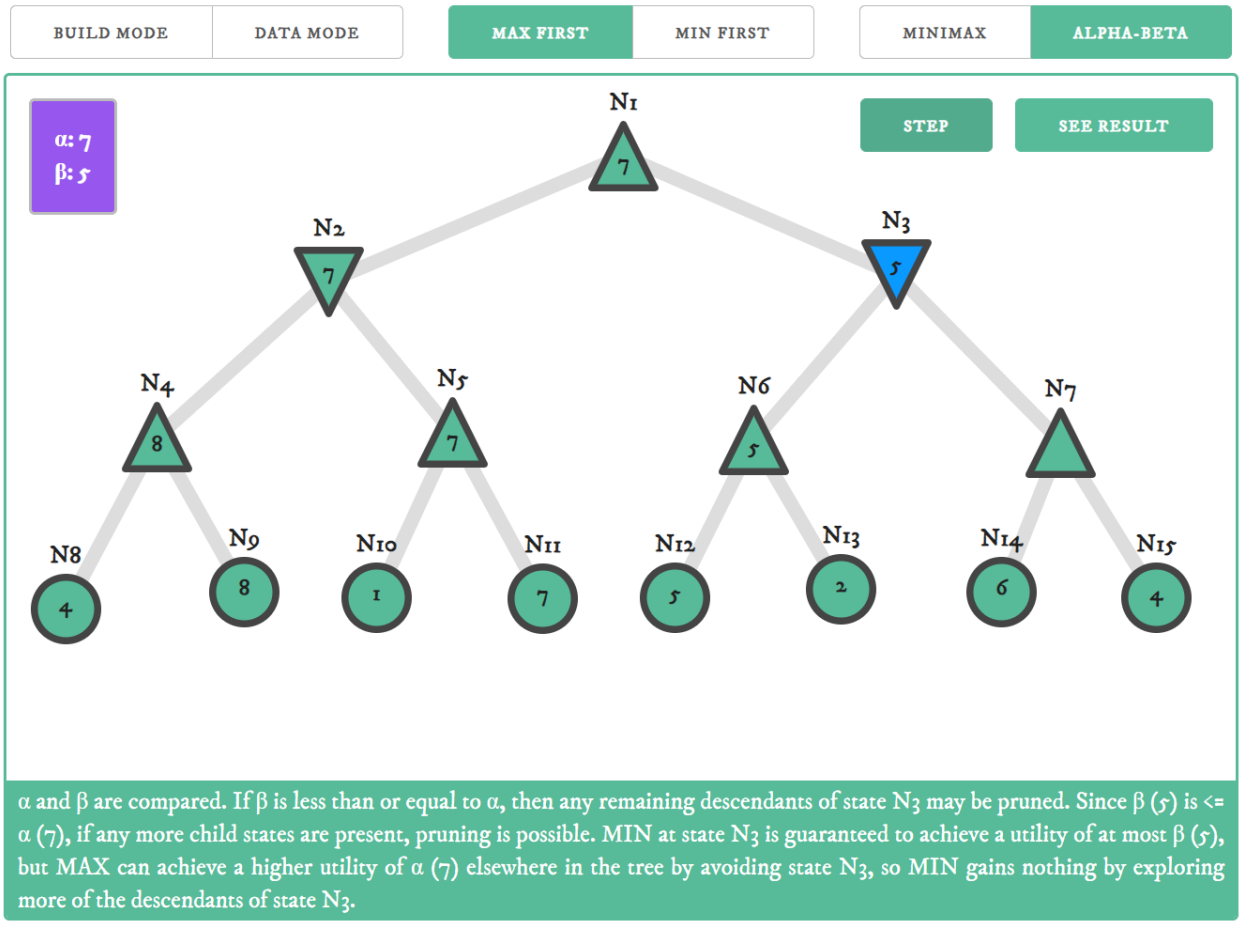


Figure 6: An example of a tooltip for an alpha-beta compare operation. As the alpha-beta compare operation may be one of the hardest aspects of the Alpha-Beta pruning algorithm to grasp, our tooltip here includes detailed information about the motivation and consequences behind the alpha-beta compare instruction, as it relates to the current algorithm state.

Beyond the aforementioned considerations, our Minimax program boasts a few other improvements over existing implementations. Firstly, either MAX or MIN may be specified as the player who moves first. This simple improvement makes the program more generalizable and can help users understand the different consequences that arise due to the order of play. Secondly, we provide a legend below the visualization window detailing the semantic meaning of different shape and coloring schemes which help to make the program more user-friendly and intuitive, in general. Finally, our program presents a cleaner user-interface than the existing implementations which avoids the occlusion of important elements even with large trees and gives clear instructions to the user before they begin using the program. Figure 7 presents the entire interface for the Minimax program.

SHORTCOMINGS

While we believe our Minimax program to constitute a significant improvement over the existing implementations in terms of usability, intuitiveness, generalizability, and effectiveness at education users about the Minimax algorithm, a number of shortcomings still exist.

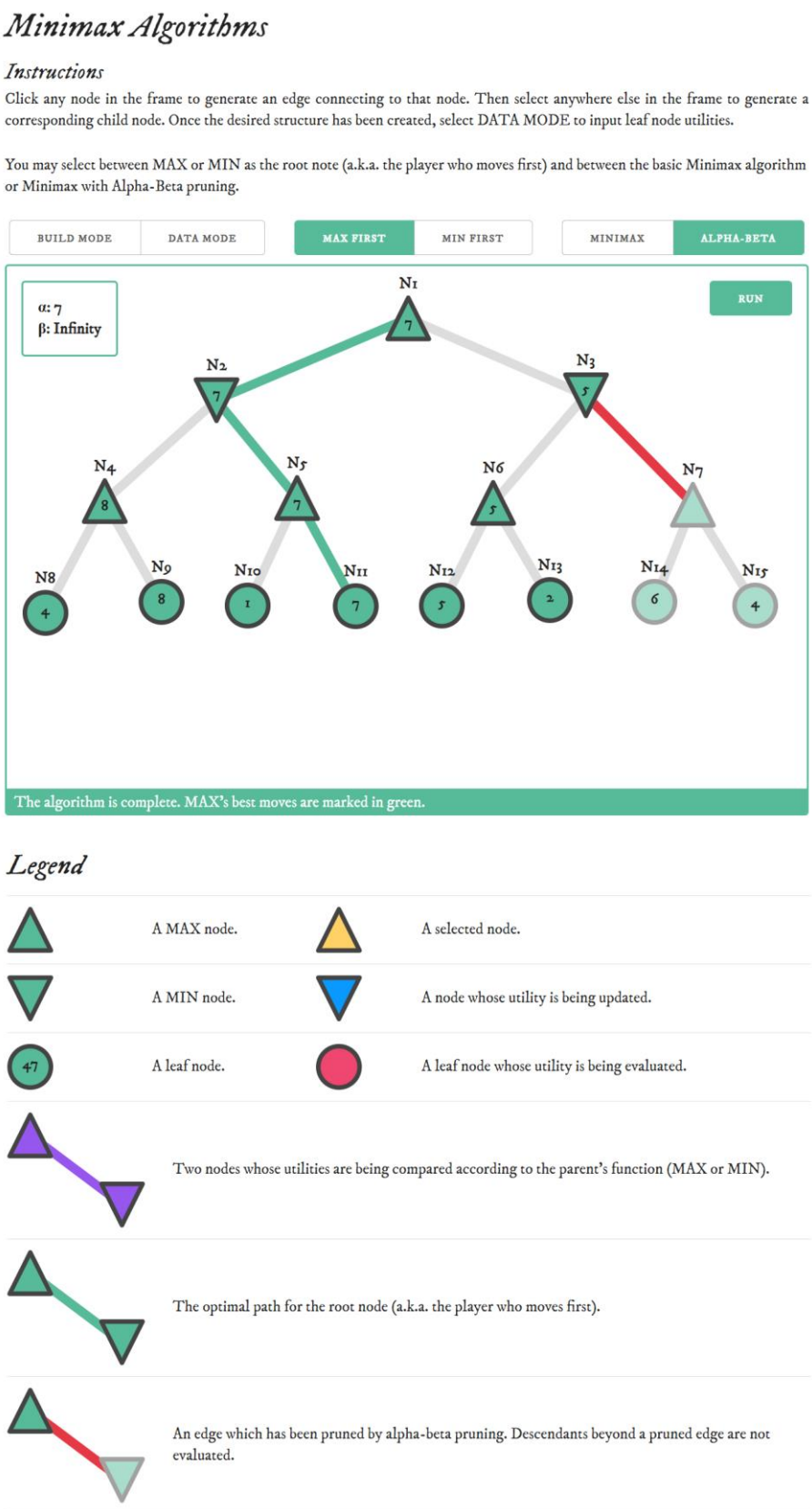


Figure 7: The entire user interface for the Minimax program.

Firstly, the visualization window is rather small relative to the size of each node, meaning you cannot fit large trees within the visualization window without being forced to place nodes atop one another. While considerable research and effort were put into the pursuit of a zoomable interface, no solutions were of satisfactory performance, yet. For now, to get around this issue and construct larger trees, the entire window may be zoomed out (Ctrl+- or Cmd+- (Mac)). In doing so, the visualization window will stay the same size, but the size of nodes will get considerably smaller, allowing more to fit within the visualization window.

Secondly, the program is missing the Expectiminimax algorithm, which is a fundamental enough algorithm for adversarial search that we believe it to be a considerable drawback that our program does not implement it.

Lastly, tree construction could be made more efficient by the ability to delete single nodes instead of only the option to delete all nodes. The complexity of implementing such a feature, however, quickly outweighed the benefit of the feature itself, though with more time it would certainly be worth implementing.

CSP PROGRAM

Unfortunately, due to time constraints, an implementation of the CSP program we wished to create could not be completed. Though we cannot offer a live, web-based program for the CSP algorithms discussed in this work, the planning and design stages were completed. In this section we present all of our planning and design work completed for the CSP program, regrettably without helpful visual aids.

For the CSP visualization program we aimed to create a first-of-its-kind, interactive visualization program which visualizes a few of the fundamental CSP solving techniques including Backtracking (BT), Forward Checking (FC), and Maintaining Arc Consistency (MAC). Many of the more complex CSP solving techniques build off of these fundamental methods by adding heuristic measures to different parts of the algorithm, so we believe it sufficient to constrain our CSP program to visualizing only these three algorithms.

As previously stated, no live, web-based CSP visualization programs were discovered by our investigation. Based on the CSP programs that were discovered in our investigation, it seems that programs implementing CSPs primarily opt to focus on being useful for industrial purposes instead of academic purposes. This may help explain the lack of live, web-based CSP visualization tools since industrial uses of CSP tools are less likely to warrant a visualization component than academic uses and are more likely to desire implementations in much faster programming languages than JavaScript. As no implementations exist from which to begin our planning and design, we present a novel approach to interactive CSP visualization designed from scratch.

Some important design choices to consider before implementing the CSP program are as follows:

- How best to visualize the CSP itself.
- How best to accept input for CSP construction.
- How best to visualize the search space of each algorithm.
- How best to indicate the different instructions of the BT, FC, and MAC algorithms.
- How best to provide helpful explanatory material which might aid a student in their understanding of these algorithms.

CSP VISUALIZATION

To the first consideration, we opt to visualize the CSP itself as a constraint graph. Constraint graphs are graphs in which each node corresponds to a variable of the CSP problem, and an edge connects any two variables that participate in a constraint with one another [3]. Constraint graphs are a very useful and intuitive visualization for a CSP problem as they capture all variable and constraint information into one unified structure and leave only the domains of variables to be represented elsewhere. Much like in the Minimax program, we will present the constraint graph in a window below a control panel consisting of various buttons for manipulating the visualization state.

INPUT FOR CSP CONSTRUCTION

To the second consideration, we will borrow from the Minimax program the graphical, point-and-click input for tree construction and extend it to be capable of creating graphs. The only change necessary will be to allow newly constructed edges to connect to already existing nodes instead of only to new child nodes. At this point, we will have a method for defining all the variables that exist in the CSP as well as the number of constraints and which variables participate in each constraint. Left to define are the domains of each variable and the relations which define each constraint. For defining the domains of each variable, we will again borrow from the Minimax program the utility specification procedure and extend it to accept a list of numbers instead of only a single utility value. This way, a finite domain of numbers can be specified for each variable in the CSP. As domains are specified, a node table below the visualization window will be populated with the domain of each node since they are not easily representable inside the node themselves. Finally, only the definition of constraint relations remains.

Defining constraint relations, however, constitutes the most difficult definition for which to define an input procedure. Constraint relations may be defined by an explicit list of all variable assignments which satisfy the constraint or an abstract logic and/or arithmetic relation (e.g., $X + 5 < Y$, $(X > 10) \wedge (X * 2 < Y)$, etc.). To support both these conventions we will extend the Minimax program utility specification procedure to automatically select edges missing relation data and to accept input as either a list of lists of numbers representing the list of variable assignment which satisfy the given constraint (so in this case, variable assignments are represented as lists of numbers) or a JavaScript code expression which uses the variable names of the given constraint and JavaScript logical and/or arithmetic operators to define the constraint relation. Only JavaScript expressions which strictly evaluate to a boolean will be accepted. By using JavaScript expressions, we avoid the burden of defining our own logical and arithmetic parser and can simply execute the inputted JavaScript expression using JavaScript itself. As constraint relation definitions are specified, an edge table below the visualization window will be populated with the variables and relation of each edge.

SEARCH SPACE VISUALIZATION

To the third consideration, we opt to visualize the search space as a tree since all three algorithms considered in this work explore it as such. During algorithm execution, the constraint graph representing the CSP in the visualization window is swapped out for the tree of the search space. Instead of completely obscuring the constraint graph from view, however, we opt to place it in a small, scaled down window within the top right corner of the main visualization window. Clicking this scaled down inset window allows the constraint graph to be shown in full view again.

COLOR CODING FOR ALGORITHM STEPS

To the fourth consideration, we follow suit of our Minimax program and use a color-coded system for different algorithm instructions. For all three algorithms explored in this work, the procedure followed by each algorithm can be decomposed into three main parts: variable selection, inference, and the recursive call. For variable selection, all unassigned variables will be highlighted purple in the domain table and compared using whichever variable selection procedure has been selected. Once a variable is chosen according to the selected procedure it will be highlighted blue and the search space tree will be expanded by one layer with all the potential assignments for the selected variable. All potential assignments are then iterated over and inference and a recursive call to the algorithm are performed on each potential assignment. When a potential assignment is the current assignment being considered in the iteration, it is highlighted yellow. During inference it is highlighted pink. If a value choice leads to failure, then it is highlighted red and a subsequent assignment is selected, or if the end of the iteration has been reached, backtracking occurs. If a recursive call is made, the whole procedure is started over again.

PEDAGOGICAL TOOLTIP SYSTEM

To the fifth consideration, we follow suit of our Minimax program and provide a tooltip system which explains each step of the chosen algorithm as the user steps through them. We shall display tooltips in the same manner as they are displayed in the Minimax program, at the bottom of the visualization window. For instructions which are harder to understand the motivation for or consequences of, we will provide additional explanatory information which hopefully effectively communicates the motivation and/or consequences.

Beyond our solutions to the above aforementioned considerations, we shall provide a legend, below all other elements, of the color code used, in the same manner that a legend is presented in the Minimax program. Additionally, instructions will be provided at the top of the CSP visualization page detailing the input mechanisms for constraint graph construction, domain specification, and constraint relation specification.

IV. CONCLUSION

In this work we have discussed the shortcomings of current techniques for teaching classical AI algorithms and the potential for powerful interactive visualizations to ameliorate many of these shortcomings. We have covered the strengths and shortcomings of related software programs which implement the Minimax and Alpha-Beta pruning algorithms. And lastly we have covered the programs of this work, along with the considerations assessed and decisions made during the planning and design processes of our program creation and the resulting strengths and shortcomings of the finished products.

As is rather obvious, there are a number of ways in which the work herein may be continued or extended. Firstly, completion of the CSP program detailed in Section III, which regrettably could not be reached due to time constraints, could be pursued. Secondly, there are a number of algorithm variations or optimizations which could be added to our visualization programs as a toggleable feature much like the Alpha-Beta pruning in the Minimax program which would serve to make the programs even more generalized and allow users to explore or become familiar with even more algorithms. For the Minimax program, support for the Expectiminimax algorithm which allows for the existence of chance nodes in the search space, could be added to the program. For the CSP program, support for many toggleable heuristics including Minimum Remaining Values (MRV), the degree heuristic, the least-constraining-value heuristic, conflict-directed backjumping, and the min-conflicts heuristic could be added to the program.

In spite of our failure to create a CSP program, we believe our Minimax program to constitute a significant improvement over those previously existing implementations and to be appropriate for use in an introductory academic environment.

The Minimax program may be visited on the author's personal website: https://alexcostaluiz.com/visualizations/.

REFERENCES

[1] *Minimax game search algorithm with alpha-beta pruning*. (2011). [Online]. Available: http://homepage.ufp.pt/jtorres/ensino/ia/alfabeta.html

[2] *UNISC - Trabalho de Inteligência Artificial*. (2012). [Online]. Available: https://raphsilva.github.io/utilities/minimax_simulator/

[3] S. J. Russell, P. Norvig, and E. Davis, *Artificial intelligence : a modern approach*, 3rd ed. (Prentice Hall series in artificial intelligence). Upper Saddle River: Prentice Hall, 2010, pp. xviii, 1132 p.