# Supplementary TestVision Examples

# 1 Introduction

This document contains descriptions of three programs implemented in the TestVision interface. The programs are implementations of algorithms that are commonly taught in CS1 or introductory courses on algorithms and data structures, namely insertion into a binary search tree (BST), the sorting algorithm Bubble Sort, and depth-first search (DFS). The purpose of documenting them here is to help the reader better understand how users interact with the system and to demonstrate how it can be used to work with more complex programs than the Greatest Common Divisor and Dot Product examples presented in the paper. We showcase both how the underlying algorithms are implemented and how TestVision can be used to (i) test the program implementations and (ii) assess the quality of one's tests.

Each example is described in detail with illustrative screenshots. It is also possible to explore them directly as pre-configured examples with interactive tutorials on the TestVision website, which is linked in the paper:

#### https://testvision.algot.org/

In the system, the examples are presented as new "Challenges" and can be accessed in the "Workspace" tab, located below the existing tutorials, as shown in Figure 1.

My Workspace - Algot	× +				~
$\leftarrow \rightarrow \mathbf{C}$ $\bigcirc \mathbf{C}$	] ≅º localhost:3000/66f2b2	15aed80af5106c08be	菜 <sub>4</sub> 120% ☆	0	③ œ ☆ Ξ
● Workspace My Operations	五 Testing	State View	× ×	Operations	(?) Queries
Workspace Name				Search	
My Workspace		0		New	
Tutorials				+ New Operation	
		<u> </u>		Tutorial Operations	
Quick Introduction		0		A BST Insert	
A Operation Testing				A BST Find	
				Basics	
$\phi$ Mutation Testing				New Node	N
				Add Child	A
Challenges				+1 Increment	1
Binary Search Tree				-1 Decrement	
				Add Edge To	
Bubble Sort				Swap Values	
				Nelete Edges	
1 Depth-First Search				Documentation	
				Select a tool to see documentation.	its

Figure 1: The "Challenges" containing more complex examples below the tutorial

# 2 Examples

## 2.1 Binary Search Tree Insertion

For this example, we consider an operation that inserts nodes into a Binary Search Tree (BST). The implementation is shown in Figure 2 and can be accessed in TestVision by opening the "Binary Search Tree" challenge.

The Binary Search Tree Insertion operation takes two nodes as inputs. The first node is expected to be the root of the BST, and the second node is the one to be inserted into the tree. These input nodes are represented by blue-colored nodes in Figure 2. The input also defines two possible children for the root, representing the rest of the BST, which are referred to as pattern nodes in Algot and are shown in pink in Figure 2. To define these pattern nodes, the programmer can apply the "Append Child" pattern to the root node, found in the "Patterns" tab of the operation editor.

Algot uses these pattern nodes to check whether the input matches the expected structure. If the input matches, all actions defined in the operation are executed. If actions are defined on nodes that do not exist during the actual execution, the operation simply returns. This behavior is useful, for example, when iterating over a list and stopping recursion once the end is reached.



Figure 2: Implementation of the BST insert operation

The operation is defined with five queries (at the top of Figure 2), which serve as conditions for actions in the operation. The first two queries of the BST insert operation check whether the root node has no children. If this is the case, the new node is simply appended as a child of the root, and the operation terminates. If the root node has two children, the value of the node to be inserted is compared with the value of the root using the third query (at the top of Figure 2). If the new value is less than the root node's value, the operation is recursively called on the left child of the root. If the value is greater than or equal to the root node, it is recursively inserted into the right subtree.

If the root node has one child, and the values of both the single child and the node to be inserted are either larger than, less than, or equal to the root node, the operation is recursively called on the single child to maintain the structure of the BST. These conditions are checked by the third, fourth, and fifth queries (at the top of Figure 2). In all other cases, the new node is simply added as a right child of the root node. If the value of the new node is less than that of the left child, both nodes (and their subtrees) are swapped to maintain the BST structure. This step is not commonly used in BST implementations, but in Algot, it is not possible to add a right child without first adding a left child. This implementation works around the limitation by always appending the left child first and swapping the nodes if necessary to maintain the BST structure.

For this example, we assume that the user has written such a function and now wants to use TestVision's features to ensure the implementation is correct. As seen in Figure 2, the BST insert operation must cover multiple cases to ensure that new nodes are inserted in the correct location and that the structure of the BST is maintained, which makes the implementation more prone to errors and testing non-trivial.

To ensure that the operation is working as expected, the user can navigate to the testing tab and create new tests that cover different insertion scenarios, checking whether the nodes are inserted correctly and the properties of the BST are maintained. Each test is defined by creating an input graph, as seen in Figure 3, and an output graph, as seen in Figure 4, which is compared with the actual output of applying the operation to the input graph. However, the user might overlook adding tests that cover edge cases, potentially leading to undetected bugs.



Figure 3: Input graph of a BST insert test

My Workspace - Algot ×	+				~			
← → C O D = k	O D ē <sup>2</sup> localhost: 3000/66f2b215aed80af5106c08be							
O     B     Morkspace My Operations     Testing	×	2	Operations	(7) Queries				
BST Insert v +	1 Input Stage	2 Output Stage		Search				
🍷 Passing Rate: Run tests				Tutorial Operations				
📽 Branch Coverage: Run tests	Copy input graph		Clear graph	BST Find				
Run all				Basics				
Mutation testing     Show all info				New Node     Add Child	N			
BST Insert Test 🕨 😡 🔟 🛈				+1 Increment	1			
	<b></b>			-1 Decrement				
BST Insert Test 1 🕨 🧐 🔟 🛈	4			Add Edge To				
BST Insert Test 2 🕨 😡 🔟 🛈		<mark>0</mark>		Nelete Edges				
	2 6			Copy Value	с			
		<u>o</u>		Delete Node	D V			
	5			Math				
				+ Sum				
				Documentation Select a tool to see it	s			

Figure 4: Output graph of a BST insert test

To assist with this, TestVision displays coverage metrics for individual tests as well as for the entire test suite, as shown in Figure 5. These metrics allow the user to determine whether all possible insertion scenarios are covered by tests and evaluate whether the test suite is comprehensive enough. By leveraging these metrics, the user can create new test cases and improve the overall quality of the test suite.

My Workspace - Algot × +				
$\leftrightarrow$ $\rightarrow$ $\bigcirc$	0/66f2b215aed80af5106c08be	\$A 120% €	? (	\$) <r th="" ∑="" ≡<=""></r>
Image: Constraint of the second se	State View	N N	Operations	(?) Queries
BST Insert v +	_		Search	
Passing Rate: 100%	3		New Operation	
t Branch Coverage: 72%			Tutorial Operations	
🕨 Run all	0 5		BST Insert	
# Mutation testing			A BST Find	
			Basics	
BST Insert Test 🕨 💡 🔟 🛈			New Node	N
Passed			🖽 Add Child	A
Test branch coverage: 50%			+1 Increment	I
BST Insert Test 1 🕟 😡 🔟 🛈			-1 Decrement	
Passed			Add Edge To	
Test branch coverage: 61%			Swap Values	
			Delete Edges	
BST Insert Test 2 🕨 😡 🔟 🛈			Documentation	
Passed Test branch coverage: 61%			Select a tool to see it documentation.	s

Figure 5: An incomplete test suite for the BST insert operation, reaching 72% branch coverage

### 2.2 Bubble Sort

In this example, we consider an implementation of Bubble Sort. The implementation is shown in Figure 6 and can be inspected in TestVision by starting the Bubble Sort challenge and reviewing the added operations. To create a simple interface that only requires the head of the list, this implementation uses two helper functions. The use of helper functions is a common pattern in the Algot programming language and serves as a realistic example of implementing more complex functions in Algot.

The Bubble Sort operation creates a new node and stores the length of the list in it, then calls the Bubble Helper operation on the head of the list to be sorted along with the length of the list. This simplifies the interface for users, as they only need to call the operation on the head of a list without providing multiple arguments.

The Bubble Helper function uses the second argument, which contains the length of the list, as a counter and decreases it by one. It then calls the Swap operation on the list and a copy of the counter, ensuring the counter's value is not modified by the Swap operation. If the counter is not equal to 0, the Bubble Helper operation recursively calls itself.

The Swap operation takes two nodes as inputs: the first is a node (from the list) that potentially has a child, and the second is a counter. The child is defined as a pattern node, meaning the operation terminates when the end of the list is reached, and no child node exists. The operation first decreases the counter by one, then swaps the values of the first node and its child if the child's value is smaller. It then recursively calls itself on the child and the decreased counter. In this implementation, the counter is used to terminate the recursion before the end of the list is reached, as with each iteration, one more element at the end of the list is already in its correct, sorted position.

For simplicity, this implementation does not terminate early if no swaps are made during a call of the Bubble Helper operation.

•••	My Workspace - Algot						
		00 ==	localhost:3000/66f2b215aed80af5106c08be 🕱				
() Workspace	My Operations	즈 Testing	× Bubble Sort	5	22 Patterns	Operations	(2) Queries
Tutorial Operati	ions				Search		
∘     P Bubble Sw	ар	Ō					
o <sup>O</sup> <sub>e</sub> Bubble Hel	lper	Ō			Bub	erations	
₀₀ Bubble Sor	rt	Û			Bubi	ale Helper	
					Bubi	ble Sort	
			0		Basics		
			•		🗉 New	Node	N
					😐 Add	Child	А
					+1 Incre	ment	1
	Ξ				-1 Decr	ement	
	Steps				- Add	Edge To	
🗄 Create a ni	ew node with value 0	Ô			🖃 Swa	p Values	
Ount the and store t     Count the store t     Count t	number of descenda the result in A	nts of a 📋			🗞 Dele	te Edges	
+1 Increment	the value of A	ñ			Copy	/ Value	с
-0 Analy Bala					🗵 Dele	te Node	D
<ol> <li>wbblA Bript</li> </ol>	pre Heiper to a, A	U			🖳 Set V	/alue	V
<ul> <li>Delete nod</li> </ul>	ie A	Ō			Math		
					C.m		
					Documen	tation	
					Select a to document	ool to see its ation.	

a) Bubble Swap Operation

My Workspace - Algot				
$\leftarrow \rightarrow C$	0 0 ==	localhost:3000/66f2b215aed80af5106c08be	රී ක වී	=
③ ♣ Workspace My Operations	즈 Testing	× Bubble Helper	Patterns Operations Qu	(2) uerie:
Tutorial Operations			Search	
₀₽ Bubble Swap	Ô			
o <sup>♀</sup> Bubble Helper	Ō	is the value of b equal to 0?	Rubble Swap	
₀ <sup>©</sup> Bubble Sort	Ũ		Bubble Helper	
			8 Bubble Sort	
			Basics	
			New Node	1
			Add Child	,
		0 -1723	-1723 +1 Increment	
i=		ab	F -1 Decrement	
Steps			Add Edge To	
Create a new node with value 0	Ô		🚅 Swap Values	
Copy value from b to A	Ō		Nolete Edges	
	Ó		Copy Value	C
-1 Decrement the value of h			🗵 Delete Node	C
- Decrement the value of b	U		Set Value	\
<ul> <li>Delete node A</li> </ul>	Ō		Math	
<ul> <li>If the value of b is not 0</li> <li>Apply Bubble Helper to a, b</li> </ul>	Ō		C. Crow	
			Documentation Select a tool to see its documentation.	

b) Bubble Helper Operation



c) Swap Operation Figure 6: Implementation of Bubble Sort using two helper operations

We now assume that the user creates a simple test, as shown in Figure 7. This test passes and achieves 100% branch coverage. However, the user may want to further assess the adequacy of this single test and opens the mutation testing screen to do so. Next, a suitable mutation operator must be selected. Since the operation only takes a single input and does not operate on a tree, operators from the "Input Mutation" or "Pattern Mutation" categories would not be appropriate. Additionally, as the operation does not use any mathematical operators, the "Math operation mutation" and "Absolute value insertion" options are also unsuitable. In this example, we consider the "Operation mutation" option, which swaps operations called by the function with other operations of the same arity.

• • • My Workspace - Algot × +				~
$\leftarrow \rightarrow$ C O $\square$ $\approx$ locally	st:3000/66f3e32107fb04dfd1f12613	🕺 11056 ☆	♥ @	ск € ≡
My Operations     Testing	× Bubble Sort Test	ē s	Operations	(2) Queries
Bubble Sort ~ +	1 Input Stage	2 Output Stage	Search	
Passing Rate: 100%			Testing	
*# Branch Coverage: 100%	Input to Bubble Sort [1/1] III		→ Select Test Input	
	→ 3		Tutorial Operations	
■ Runai			Bubble Helper	
Mutation testing     Hide all info			Bubble Sort	
Bubble Sort Test 🗈 😡 🗊 🛈			Basics	
Passed			New Node	Ν
	3		Add Child	A
			+1 Increment	I
	2		Add Edge To	
	—		Swap Values	
			Delete Edges	
			Documentation	
			Select a tool to see its documentation.	

Figure 7: Input of a passing test for the Bubble Sort operation

This generates 34 mutants, which are displayed in the list on the right-hand side of the screen, as seen in Figure 8. After running the mutation test, we observe that all but 3 mutants are killed, resulting in a mutation score of 91%. By inspecting the first mutant, shown in Figure 9, we can see that the "Count descendants" operation was swapped with the "Copy value" operation. Using the step-through debugger, we find that the head of the list in the test we wrote has 3 descendants, which is equal to its value. Because of this, our test is unable to kill the mutant, even though it is not equivalent to the original operation.

→ C O D == localhost:3000/66f3e32107fb04dfd1f12613				nos 🏠			
t Mu	tation Testing	- Bubble Sort Suite			;		
Mutation Settings		Bubble S	ort Test Suite St	ats			
"When debugging, novices insert corrective code; experts remove defective code." -Richard Pattis		<b>1</b> tests	100% Branch coverage	34 mutants	0% mutation score		
		Test suite	performance				
		Mutant pe	rformance				
nput Mutation 🛛	<						
Vutants are generated by changing the inputs to the operation.		Weak Mut	ation <b>@</b>				
Action Mutation 🕖							
Mutants are generated by altering the actions of the operation.		1. Muta	nt 34	(Action Mutation)	(Not yet run )		
× Math operation mutation							
× Query input-order mutation × Query mutation × Absolute value ins	sertion	2. Muta	nt 35	Action Mutation	Not yet run		
Pattern Mutation 🛛		3. Muta	nt 36	(Action Mutation)	Not yet run		
Autants are generated by altering the input graph structure.							
, , , , , , , , , , , , , , , , , , , ,							

Figure 8: Mutation testing screen with "Operation mutation" selected

	O D ≂2 localbost 3000/6663e321076b04d6d1f12613		Ż.	10%	
•	Mutation Testing	Bubble Sort Suite	×	ω	
Mutant 40	ant alive) (Mark as equivalent)	Bubble \$	Sort Test Suite St	ats	
Mutant	↑ Shift to show original s to step-through x to close	1 tests	<b>100%</b> Branch coverage	<b>34</b> mutants	91% mutation score
Channe		Test suite	e performance		
0   Create a nev	v node with value 0	Mutant p	erformance		
1   Copy value f	rom a to A				
2   Increment th	e value of A	Weak Mu	tation 🛛 🔵		
3 Apply Bubbl	e Helper to a, A	🗈 Run te	est Show killed mutant	s	
4   Delete node	A				
Operation Inputs		1. Muta	ant 40	Action Mutation	(Mutant alive)
→ a		2. Muta	ant 41	Action Mutation	(Mutant alive)
In such Dathans		3. Muta	ant 42	Action Mutation	(Mutant alive)

Figure 9: Mutant for the Bubble Sort operation that is not killed by the test suite

To address this issue, we create a new test case where the number of descendants of the head of the list is not equal to its value, as shown in Figure 10. After re-running the mutation tests, we observe that all but one mutant are killed after adding this new test case.

🔴 🔴 🌒 My Workspace - Algot X +							
$\leftrightarrow$ $\rightarrow$ C O D $\stackrel{\text{\tiny end}}{\rightarrow}$ local	host:3000/66f3e32107fb04dfd1f12613		💫 110% ☆				
<ul> <li>(i)</li></ul>	×	× Bubble Sort Test 1					
Bubble Sort v +	1 Input Stage	2 Output Stage		Search			
Passing Rate: 100%				Tutorial Operations			
M Denesk Courses 100%				Bubble Swap			
- Branch Coverage: 100%	Copy input graph	🔟 Clear graph		Bubble Helper			
🕨 Run all				Bubble Sort			
# Mutation testing				Basics			
				🕒 New Node	Ν		
Bubble Sort Test 🕒 9g 🔟 🛈				Add Child	A		
Passed	0			+1 Increment	1		
Test branch coverage: 100%	1			-1 Decrement			
Bubble Sort Test 1 🕞 😡 🕅 🕥				Add Edge To			
Passed	<u></u> 0			Swap Values			
Test branch coverage: 100%	2			Nelete Edges			
				Copy Value	С		
				Delete Node	D		
	5			-			
				Documentation			
				Select a tool to see its documentation.			

Figure 10: New test that kills the mutant from Figure 9

If we inspect the last remaining mutant, as seen in Figure 11, we observe that the "Count descendants" operation was swapped with the "Get tree height" function. However, in the case of a list, both operations are equivalent, making it impossible to kill this mutant. To address this, we can mark the mutant as equivalent, which allows us to achieve a mutation score of 100%.

We can now iteratively enable more mutation operators and create new tests to kill any surviving mutants, continuously improving our test suite.

→ C O D ≈ localhost:3000/66f3e32107fb04dfd1f12613		×4 -	110% 😭	S @ . ?
Mutation Testing	Bubble Sort Suite			
Autant 42	Bubble S	ort Test Suite St	ats	
Action Multiple (Mark as equivalent)	2 tests	<b>100%</b> Branch coverage	<b>34</b> mutants	97% mutation score
Stone	Test suite	performance		
0   Create a new node with value 0	Mutant pe	rformance		
1 Get the height of a tree where a is the root node and store the result in A				
2   Increment the value of A	Weak Mut	ation 🛛 🔵	_	
3   Apply Bubble Helper to a, A	🗈 Run tes	Show killed mutant	ts	
4   Delete node A				
Operation Inputs	1. Muta	nt 42	Action Mutation	(Mutant alive)
Insut Pattern				

Figure 11: Equivalent mutant of the Bubble Sort operation

### 2.3 Depth-first Search

For the final example, we consider an implementation of Depth-first Search (DFS) that numbers the nodes in the order they are visited. An example of how the operation numbers the nodes in a graph can be seen in Figure 12.

My Workspace - Algot × -				$\checkmark$
$\leftrightarrow$ $\rightarrow$ $\mathbb{C}$ $\bigcirc$ $\bigcirc$ $\bigcirc$ $\bigcirc$ $\bigcirc$ $\bigcirc$ local	host:3000/66f419c907fb04dfd1f12614	落 110% ☆	v .	œк ₹} ≡
●	State View	2 1	Operations	2 Queries
Workspace Name			Search	
My Workspace			New	
			+ New Operation	
Tutorials			Tutorial Operations	
Quick Introduction	2 5		<mark>ា</mark> DFS	
I Operation Testing			Basics	
Operation resting			New Node	N
$\phi$ Mutation Testing			Add Child	A
			+1 Increment	I
Challenges			-1 Decrement	
A Binary Search Tree			Add Edge To	
Bubble Sort     Bubble Sort     Sort			Delete Edges	
	7		Copy Value	с
1) Depth-First Search			Delete Node	D
			=	
			Documentation Select a tool to see its documentation.	5

Figure 12: Execution of the DFS operation on a binary tree of height 2

The operation takes two nodes as inputs: first, the current node being traversed by the DFS, and second, a node containing a counter. The precondition for the operation is that every node of the tree being searched has a value 0, indicating it has not yet been visited. The value of the second node (the counter) should be 0 on the initial call if the user wants the numbering to start at 1. In the operation definition, the currently visited node may have one child node, and the parent of the current node may also have

another child, as shown in Figure 13. This demonstrates how operations in Algot can work on inputs with specific graph structures.

If the value of the current node is 0 (i.e., it has not been visited), the counter is increased by one and copied to the current node. If the child node's value is also 0, DFS is recursively called on it with the updated counter. Next, if the sibling node's value is 0, the function is recursively called on the sibling with the updated counter.



Figure 13: Implementation of the DFS operation

By starting the "Depth-first Search" challenge in TestVision, the DFS operation is added to the sidebar and can be inspected. In this example, we consider a user who wants to test whether this function works on different kinds of graphs. To do this, the user creates new tests for the DFS operation. For the first test, we consider a binary tree, as shown in Figure 14. After running it, we observe that it passes as expected.



Figure 14: Binary tree test input for the DFS operation

For the next test, the user might introduce multiple paths leading to the same node. An example of this is shown in Figure 15.



Figure 15: Expected output of a failing test that introduces multiple paths to the same node

After running all tests, we observe that the new test fails: not all nodes are visited by the DFS implementation. The issue is that the operation checks the first child of a node and only recurses over the other children if the first child has not been visited yet. However, by introducing multiple paths to the same node, the first child may have already been visited in a previous iteration, causing the DFS to terminate even though there may still be unvisited children.

The user can observe this in more detail by opening the step-through debugger for the failing test by clicking on the footsteps icon, as shown in Figure 15. In the step-through debugger, all executed actions and their order can be stepped through, revealing why the recursion terminates early. An example of the step-through debugger on the failing test is shown in Figure 16.



Figure 16: The step-through debugger opened on a failing DFS test

					Novelty			Us	efulness			intui	tive use				clarity		
ID	Age	Gender	creative	inventive	leading edge	innovative	useful	helpful	beneficial	rewarding	easy	logical	plausible	conclusive	well grouped	structured	ordered	organized	Average
1			7	5	6	7	7		5	7 5		3	6	7 6		5	7	5	5 5.8125
2			6	7															6.5
3			6	4	4	5	4		5	5 5		4	6	5 6		6	4	5	4.875
4			6	6	5	6	6		6	7 6		6	7	7 7		6	6	6	6.1875
5			6	6	6	7	5		6	6 6		5	5	6 5		6	5	5 6	5.6875
6			7	7	7	7	5		7	7 5		7	7	7 6		7	7	7	6.6875
7			6	6	5	6	7		7	7 5		4	5	7 7		6	6	6 6	6 6
8			6	4	2	5	5		4	5 3		7	1	2 3		6	7	6	6 4.5
9			7	7	6	6	5		6	6 7		6	6	6 6		5	5	6	5 5.9375

ID	What part of the course material do you consider to be the most challenging or difficult?		
1	- Symbolic execution (Klee in particular, not very intuitive results) Data Flow Analysis (difficult content)		
2	I think recursion is hard to grasp. Especially for new students of programming or math. This needs a deeper introduction I think.		
3	I think the design of the dot product algorithm is the most challenging one.		
4	Mutation testing. How to compute mutation score and how to efficiently find the test cases that are useful.		
5	The fundamental of software testing as there was little experience prior to taking the course.		
6	I found the abstract interpretation lecture a bit confusing. The rest was fine, especially after doing the exercises		
7	Most of the concepts were ppretty straightforward. Some homeworks were difficult to setup. The project is a bit challenging to practically apply the concepts we learnt in lecture.		
8	Mutation testing		
9	Symbolic execution		
	Could the material be presented in a different way to help you meet your learning goals?		
1	Data How Analysis could be presented initially from a more practical perspective (examples, etc).		
2	It is a good complement to the learning material in the lectures.		
3	it s preux deer in my opmon.		
5	I mink it is a good introduction, by demonstrating mutation testing in a visual way, we can understand it easily. I mink it would be better by discussing more details about dimetent kinds of mutatits.		
5	Inclusion, include in the supervises and excitance to below it is understanding.		
7	vouing voites or innut inte die excluses and assignments or rap war anderstanding.		
8	its, we feating every uning in a very unconcervation appreciated some concepts to be shown practically used in the rectance. Multiplicing were poly evolution been mentioned number of times in exemples. Having evolutioned what multiplicing are would have believed		
9	inductions were only explained are naving been membried interest of answers in examples, instrumed explained what mutations are would have negled.		
5	Base describe your impression with the software		
1	Vervice idea to offer a visual language The interface is modern and clear. Sometimes it's difficult to navigate around the different functionalities of the language		
2	It is a pool software with a clear U. Great with hover information on buttons. Nice overview of lear cases easy to follow.		
3	The software is usable, but there can be improvements such as when designing an operation, the close button "x" can be more obvious. It took me long to find where it is. Also, adding children can be done pressing "A" repeatedly instead of having to click on the newly a	d	
4	It is a very fancy software. It's my first time to use it, vet I find it quite easy to start with. It's useful to learn new knowledge by experimenting with this software.		
5	Design very good, minimal and easy to understand. User functionality, everything is strate forward and implemented in a user friendly way		
6	It looks good and easy to navigate.		
7	At first glance, the software seems very helpful to solve/understand problems before coding them eg in intro to programming		
8	Nice to see visually what happens during mutation testing.		
9	It seems very well thought out and beautifully designed. It just seems to have some hiccups.		
	Could you describe any challenges you can identify with the software?		
1	It is not possible to give names to nodes ("variables" naming). There should be more explanations describing the different functionalities (on the interface). It could be difficult to implement more complex operations.		
2	Not immediately.		
3	The challenge I can identify is how to deploy the written program in Algot to real life.		
4	Add more tutorials, it's still a little confusing about how to use this software, like how to add inputs to testcases.		
5	not right of the bat, but some at the short cuts and case of user flow could be improved with more information but that could also waste it harder on the user		
6	No.		
7	I found it difficult to understand. There was an introduction but it was fast and then when working with it, I forgot how to do multiplication operation.		
8	People unfamiliar with Algot may need some time to wrap their head around all of these interfaces		
9	General usability kinks, like placement of nodes, being able to delete nodes from the input graph in demonstration node without making it an operation		
	Do you think that the software can help you with your learning goals?		
1	Yes it can be useful to learn recursion for beginners and the feature about mutation testing is very well done and I would us it to actually understand better how mutation testing works.		
2	It is for sure easier to grasp mutation testing visually. Also equivalent test cases are easier to understand and compare.		
3	Yes. The visual nature of the software helps a lot with that.		
5	Tes.		
5	Inaversing a state stepper is a group rearming you to see how these tests are implemented us over it could neep Vise accessfully the multiplement from being state and the understand the constrained by the multiplement		
7	res, especially the induction testing part. The visualizations are note to understand the operations done by the inductions.		
9	its.		
9	Onsaire. In nave to really up deep to by it injects to the		
	What improvements or additional features could improve the software?		
1	Name to variables: make testing section clearer, documentation: visualization of branch coverage and other lyoes of coverage metrics: clearer separation between phase of operation definition and actual use of it		
2	Algot in general ctrl+z for regret.		
3	More features such as OOOP and floating points.		
4	Maybe more testing technics could be added. Such as dataflow testing.		
5	A guiding hand for users that are finding this on there own to get them up to speed with usage		
6	Being able to name the nodes would be nice. If this is alread possible I don't see how.		
7	Maybe have a user guide that I can click on when I get stuck.		
8	A some sort of "guided form." I would never have figured out how to use it without the verbal explanation.		
9	* Further testing concepts like symbolic execution * Sometimes the demonstration node tends to make some assumptions, e.g. what input for nodes match what graphs, and these seem to be made super explicit in the software, which is confusing		
	Do you have any other comments regarding the software that you would like to share?		
1	I've noticed that there are different releases of the software, it would be nice to know what is the difference between them :)		
2	Really good implementation at east on the front end. Hopefully backend is good as well.		

3	The tutorial is really clear and the software only needs minor adjustments.	
4	It's very good!	
5	Slower a bit, getting last could be a problem and going over more tests in a creative way. P.S. the tutorial was really good	
6		
7	In the testing part. When it adds a node at the bottom and I repeatedly click on it, it labels 'a, b, c usually clicking the background exists from that. However, for these nodes, I had to click the background at the top near the other nodes to exit from the labelling.	
8	Ton of material to cover within 15 minutes, especially to people who have never heard of Algot before	
9		