Capstone Project Phase B

Self-stabilizing Systems – concept and basic algorithm for mutual exclusion

23-1-R-10

Name: Or Biton **E-Mail:** Orbiton3005@gmail.com

Supervisor:
Prof. Shmuel Zaks

Project code - https://github.com/or3005/algorithm_simulator.git

Table of Contents

1 Project Review

1.1 Abstract

The project involves an in-depth examination of Dijkstra's groundbreaking paper on the subject of self-stabilization, which falls under the domain of distributed computing, specifically addressing issues related to fault tolerance. the initial phase of the project entails a thorough study of the fundamental algorithms outlined in Dijkstra's paper, and using the Kruijer trees method to prove and implement Dijkstra's idea.

The subsequent phase of the project revolves around the development of a software tool that is designed to replicate Dijkstra's 4-State machines self-stabilization algorithms. the primary focus is on creating a visual representation of this algorithm, facilitating their utilization in educational settings such as classrooms. the software allows for a step-by-step simulation of the algorithms, enabling users to comprehend their functionality, in addition, it simulates an automatic run of the algorithm in its entirety.

Additionally, the software presents Kruijer proof regarding a finite number of rotations until the system stabilizes itself by running different scenarios on the system.

1.2 Introduction

In Dijkstra's paper [1] "Self-Stabilizing Systems despite Distributed Control" he introduces a formal representation of self-stabilization, denoting the distributed system's capability to autonomously return from any state of malfunction to a valid state, all without requiring external intervention.

the paper has left a mark on the world of computer science. It played a significant role in shaping the development of self-stabilizing systems, impacting various fields such as distributed systems, and computer networks. The concepts introduced in the paper have contributed to advancing the understanding and implementation of self-recovering systems in the landscape of technology.

Self-stabilizing systems represent a crucial concept in distributed systems, emphasizing their ability to autonomously recover from transient faults and disturbances. these systems aim to achieve stability and correctness regardless of the initial state or any unforeseen disruptions that may occur during their operation. the idea is to enable distributed systems to converge back naturally and efficiently to valid and functional state without relying on external interventions. this approach enhances the resilience and robustness of distributed systems, making them capable of maintaining their desired behavior despite dynamic and unpredictable conditions. self-stabilizing systems play a pivotal role in ensuring the reliability and consistency of distributed computing environments, contributing to the advancement of fault-tolerant and adaptable systems.

Dijkstra's paper was significant, but it lacked a proof or implementation for the second algorithm described as the 4-state machine part. therefore, in 1979, Kruijer wrote a paper [2]

suggesting a data structure for the four-state machine problem and providing proof of the correctness of Dijkstra's algorithm.

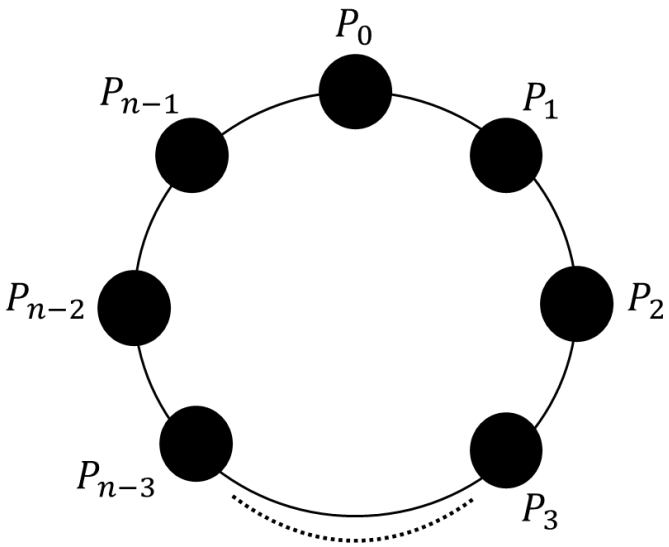
In Kruijer's paper, he offers a two-step solution in the initial part, the nodes are arranged in the tree, where the root is a special node that receives the turn under certain conditions, and in the second part, he performs an algorithm-based on Dijkstra's algorithm, and in addition, he proves that the solution he proposes is correct.

During Phase A of the project, we explore the self-stabilization concept, explaining how it operates and providing evidence that the algorithm stabilizes systems. additionally, we included examples demonstrating the executions of these algorithms.

In Phase B, we developed a software tool designed for educational purposes, for those who teach the field, especially for Dijkstra's algorithms. The software mimics Dijkstra's 4-state machines using the Kruijer tree, aiming to aid to the realization of the algorithm by providing a visual representation of the algorithm's execution. we add user flexibility, allowing them to select the number of processors in the network and determine which processor runs at any given moment, therefore enabling exploration of various algorithm scenarios, even those that do not include the legitimate state. users can also choose for automatic simulation. additionally, we conducted an analysis of the anticipated performance of the algorithm using Theorem 3 from Kruijer's paper which proves a finite number of turns until reaching the perfect state.

1.3 The Algorithm

In his paper [Dijkstra 1974], Dijkstra presented three self-stabilizing algorithms for mutual exclusion on a ring network: one with K-state machines, another with Four-state machines, and lastly with Three-state machines, focus on Four-state machines algorithm, the system composed of processors representant by a special node that will be the root in the tree, in Dijkstra paper all processor connected in a circle ring, and every node tied to left and right neighbors.



the system can self-stabilize if it is in a legitimate state, to reach the legitimate or perfect state the system should pass these conditions –

1. one or more privileges will be possible.

2. after the processor takes a turn the system stays in a legitimate state.

3. every possible privilege will be displayed in a certain legitimate state.

4. for every possible legitimate state there is a sequence of moves to another legitimate state.

1.4 Dijkstra's Four-state Machines Algorithm

In the algorithm each processor P_i has two Booleans variables that represent him, $X_s \; and \; Y_s$ so we can get 4 different states in each processor, all of them can observe its own value and its neighbors values represented by $X_R \; Y_R$ for the right neighbor and $X_L \; Y_L$ for the left neighbor.
The processor P_0 is defined as the bottom processor and by definition he's $Y_S = true$, and P_{n-1} is define as the top processor whereby definition he's $X_s = false$, these two processors are two-state machines.

The processor P_0 can have the privilege when $X_S = X_R \; and \; not \; Y_R$,and then we will change X_s value to he's apposite.
The processor P_{n-1} can have the privilege when $X_s \neq X_L$ and then we and then we will change X_s value to he's apposite.
For all of the other processors that are not unique the privilege is given in two conditions-
$when \; X_s \neq X_L \; then \; X_s = !X_s, Y_s = true$
$when \;\; X_s = X_R \; and \; Y_s \; and \; !Y_R \; then \; Y_s = false$
In addition, we assume that in the system there is a daemon that schedules the processors with a privilege by giving the token to one processor at a time.
Also, we assume that only the processor who has the token can operate has he finishs, he passes the token back to a processor that can have it.
in this way, the algorithm prevents deadlocks and will solve the problem.

1.5 Kruijer Four-state Machines Solution

to solve the problem in Dijkstra [1] self-stabilizing in distributed systems Kruijer in his paper [2] proposes an algorithm that maintains the structure of a tree in a distributed system.
the algorithm refers to a distributed system with an even (K=4)K number of states for each processor as these states are represented inside each node of the tree, Kruijer designed the tree so that in a legitimate state more than one privilege can be displayed, which would allow the parallel operation of the involved processors to occur logically.
we assume that there is a central demon who manges the token passing in an even way.

Kruijer self-stabilizing algorithm tree will consist of n nodes numbered in $0,1,2,3,\dots\dots,n-1$ as the root is in the zero node and the leaves are in the last nodes of the tree.
each node is represented with 3 variables –
$\sup[i]-$ represent the above level of i node in the tree, for example, if K is the node above to i on the path to the root then K is the superior of him and i is the subordinate of K, then $\sup[i]=k\ , 1\le k\le n-1$ as for the special node in the root i_0- $\sup[i_0]=0$ by definition.
as mentioned before every node has two variables that represent its state.
$s[i]$ – an integer with a range of $0,1,2,\dots,K-1$
$eq[i]$- is a Boolean value.
as for the root i_0, $s[i_0]=K$ is a constant state for convenience.
as mentioned before all processors i have two privilege conditions-

1. $\boldsymbol{if\ (not\ eq[i]\ and\ test(i))then\ eq[i]=true}$

in this privilege test function returning true in two conditions
firstly, when node i is a terminal node of T.
secondly, if node i is not above and has $1\ to\ K$ processors as its subordinates, or in another word he is the root and when all $eq[i]=eq[k]=eq[1]\dots.=true$.
and return false if none of the above conditions are met.

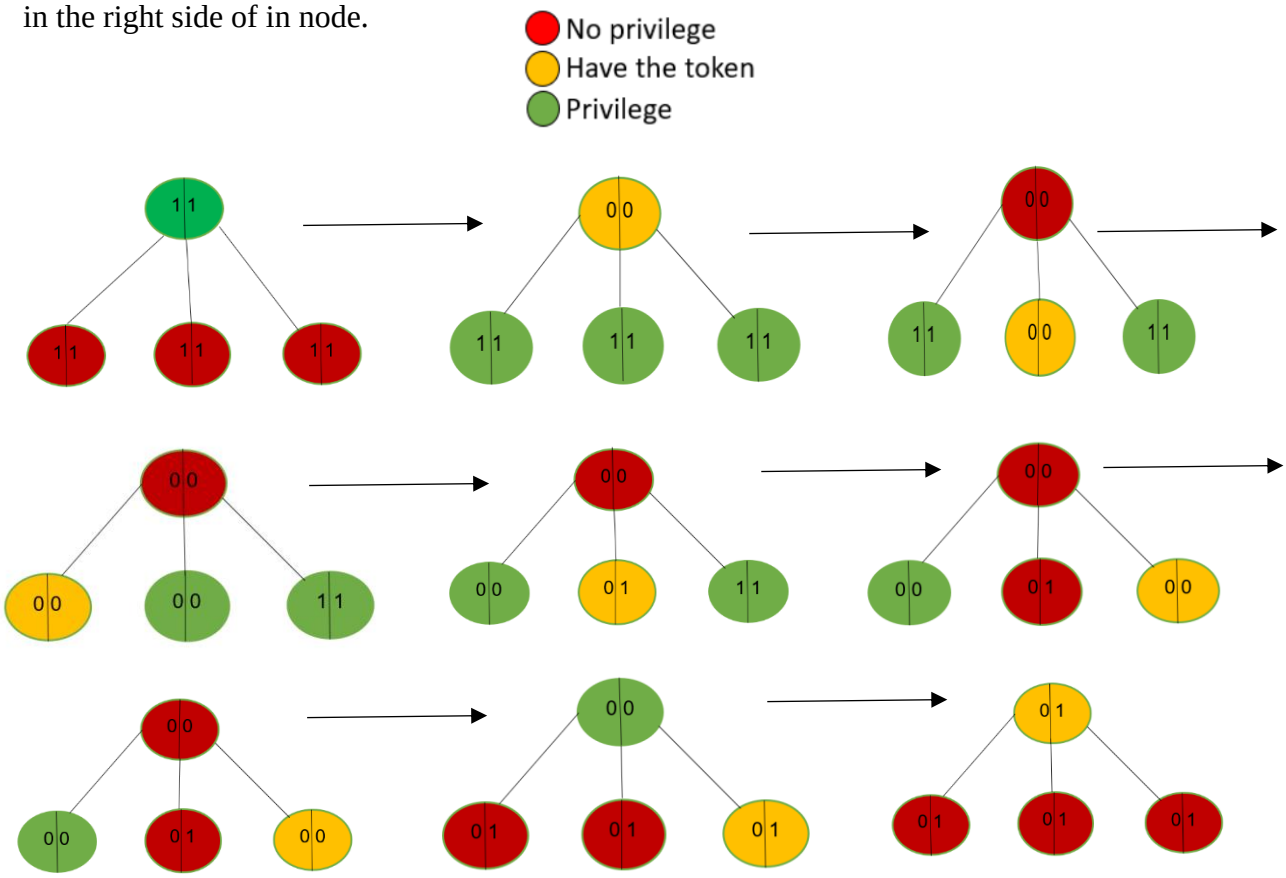
2. $\boldsymbol{.if(eq[i]\ and\ s[i]\neq s[sup[i]])}$
$\boldsymbol{then\ if\ sup[i]=0}$
$\boldsymbol{then\ s[i]=(s[i]+1)modK}$
$\boldsymbol{else\ s[i]=s[sup[i]]}$
$\boldsymbol{else\ eq[i]=false}$

the legitimate state for the system is:
1. **The perfect state** – $s[i_1]=s[i_2]=\cdots=s[i_{n-1}]\ and\ eq[i_1]=eq[i_2]=\cdots=eq[i_{n-1}]=true$
2. The states that emerge at the end of one or more allowed moves.

note that in the legitimate states only the root can make an action.

1.6 Kruijer Tree Solution Example

example of one run as 1,0 represent Boolean values in the node as $s[i]$ in the left side and $eq[i]$ in the right side of in node.



The example above represents the solution with trees as Kruijer suggested - each perfect state can be viewed as a temporary balance point of the system.

Then, the main node is moved one step forward.

This causes a disruption in equilibrium that spreads through the branches of the tree and eventually reaches the terminal node.

When the effect of the disturbance reaches the root, a temporary balance is established again in the system.

1.7 Challenges and solutions

In this project we had several difficulties and challenges, some of them theoretical and some practical, this is a list of some of them and their solutions-

1. Understanding the Algorithm: understanding of the basic logic and complexity of the presented algorithms, so we invested in studying the theoretical foundations and practical applications of the algorithms. by breaking down complex algorithms into small parts and experimenting with different parameters, we gained a deeper understanding of their behavior and performance characteristics.
2. Crafting a User-Friendly Interface: Design a user interface that looks good and is clear about what each button does, and what the algorithm does so that it is easy to learn from the software, we solved it by using JavaFX layout managers like HBox and VBox to ensure that UI elements remain visually pleasing across different devices and resolutions.
3. Ensuring handling of user input and Providing Informative feedback: try to prevent crashes caused by invalid user input by null value or giving the token to an unprivileged node, we solved this by implementing input validation mechanisms within the code to handle unexpected user inputs and prevent application crashes. Additionally, we've incorporated informative pop-up messages to guide users and provide feedback, improving the overall user experience.
4. Making a good simulation of the algorithm in an easy way to understand we try to develop a dynamic visualization component using JavaFX allows users to witness algorithmic operations step by step or automatically. In addition, using colors and an info section with specific algorithmic conditions helps the user understand the algorithm better
5. Creating trees and nodes with different variables that will perform the algorithm in a desired way: as I explained above, each node has two Boolean values, and because everything has to be drawn in the form of a linked tree when in each run each node has a different number of sons with random values for each, therefore we need to use different data structures, one array for all nodes, one array for the circles that represent the nodes themselves to the user, and an array of buttons and more, in addition I linked each node to its circle and its button in a special class called a logic whose job it is to perform these actions.

in conclusion, the project had many difficulties that I had to deal with, and with the help of the supervisor, we were able to deal with all of them optimally.

1.8 Research Process

To create the software, it was essential to research Dijkstra's 1974 article [1], and Kruijer's article [2], comprehending its significance and the algorithms within. proving the correctness of these algorithms constituted the primary focus of our research project, a task accomplished during Phase A. After we studied the algorithm, we had to prove it in the first book when this part was finished, we turned to identifying an appropriate development environment for the software. we chose to work with Java and JavaFX for several compelling reasons: the integrated development environment (IDE) is an adaptable platform for constructing algorithm simulators, with extensive libraries.

1.9 Results and Conclusions

the software was crafted to deliver an interactive and user-friendly interface, empowering users to study and experiment with Dijkstra's algorithms. Here some several key features tailored to enhance the learning experience:

1. **Adaptable Setup:** Users can adjust the number of processors within the network, making it easier to simulate a wide range of network sizes.

2. **Progressive Execution Control:** Users can oversee the algorithm's step-by-step execution by choosing which processor operates at any particular moment. This functionality enables the creation of specific scenarios, including those diverging from the algorithm's intended state, fostering a thorough analysis and understanding.

3. **Visual Representation:** The application displays a tree representation of the network and algorithm execution, streamlining the visualization of the system states. this visual tool is crucial for comprehending the algorithm's behavior and improving understanding.

4. **Rounds Analysis:** shows that Kruijer’s proof of self-stabilization from a legitimate position in a finite number of rotations to a perfect state is correct.

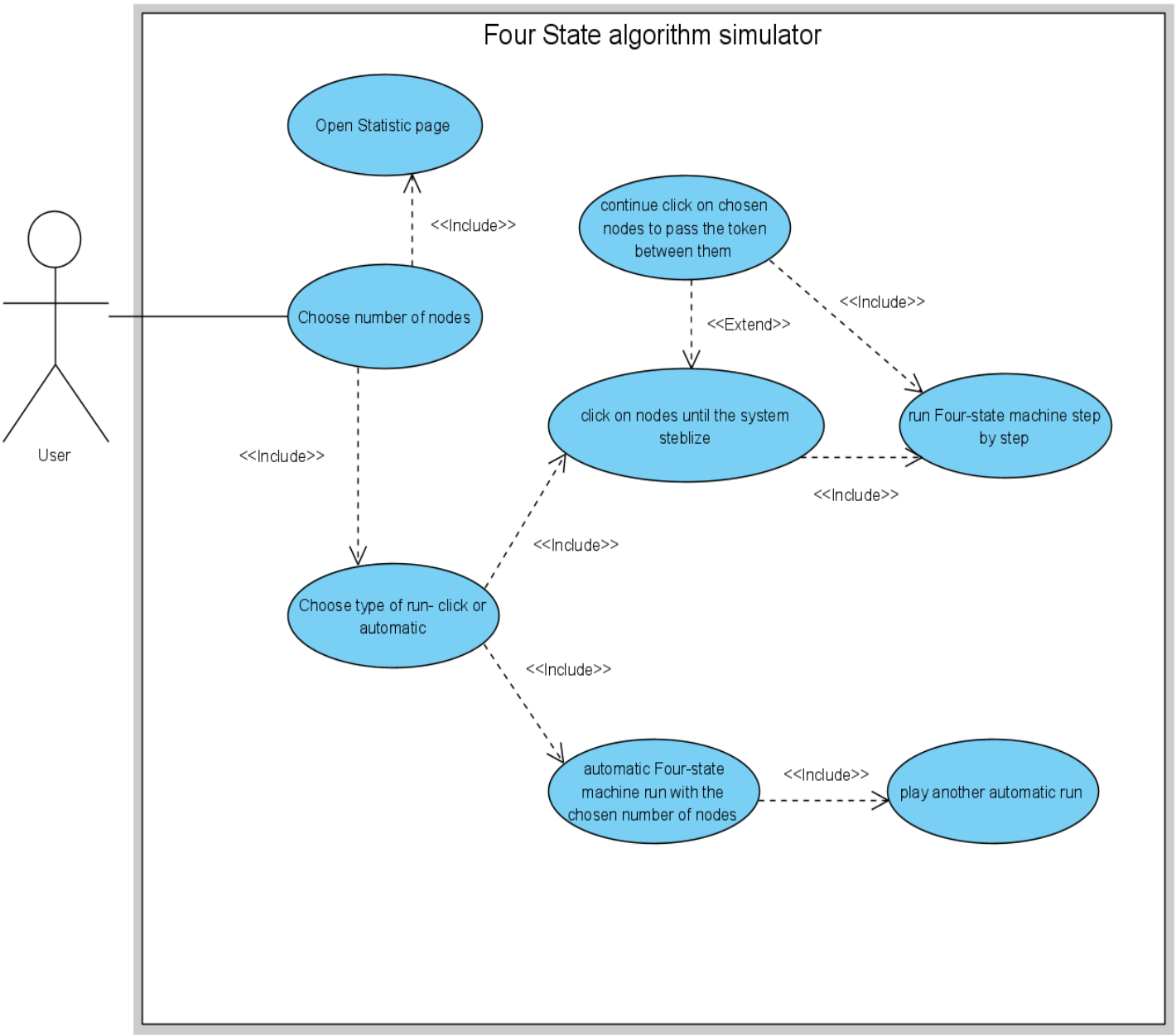
by engaging in practical experimentation and visualization, students can enhance their understanding of Dijkstra's.

the software versatility, with both manual and random configurations, can enable the exploration of a variety of scenarios.

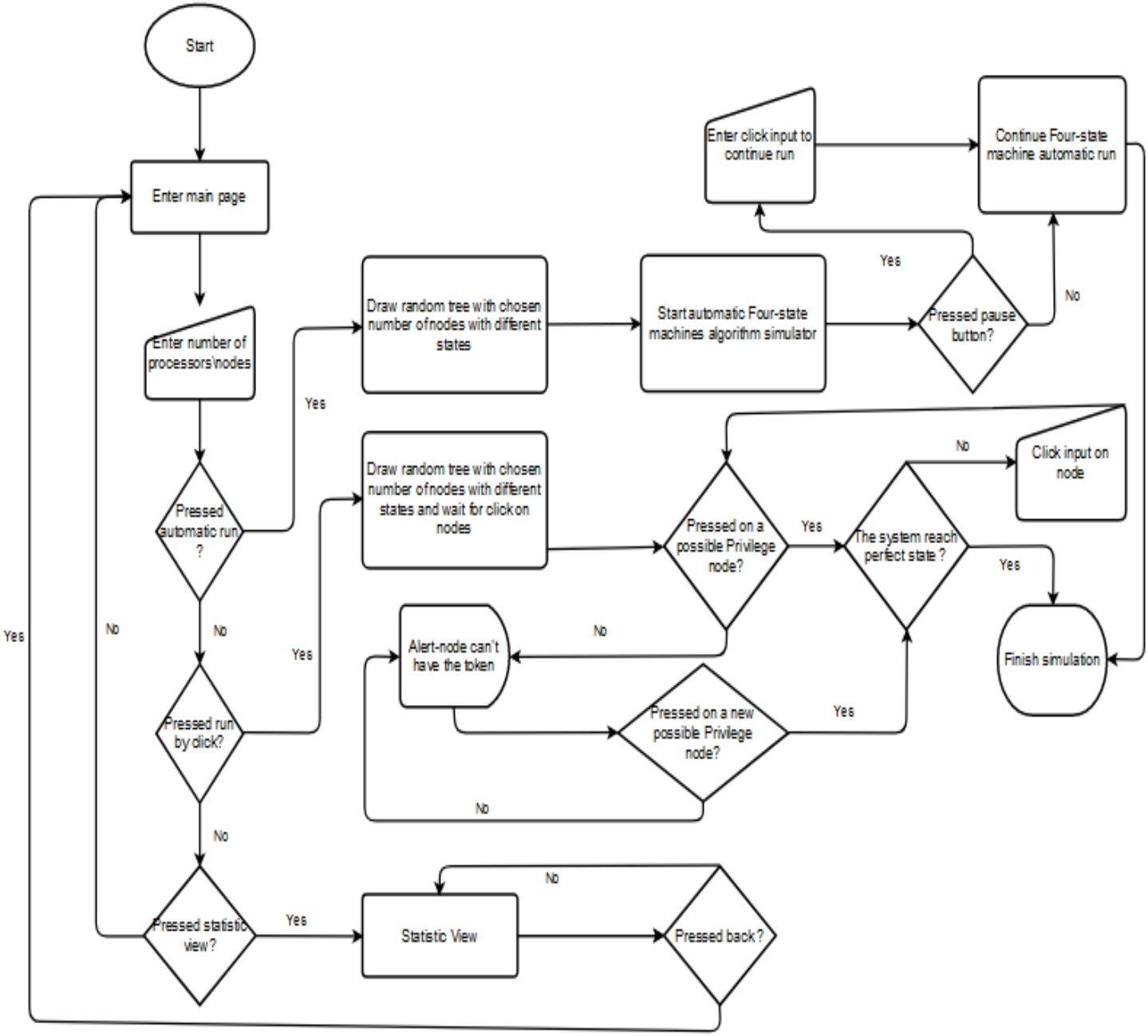
2 User Documentation

2.1 Diagrams

2.1.1 Use Case diagram



2.1.2 Flow Chart



2.2 User Guide

The software aims to replicate Dijkstra's Four-state machines algorithm as outlined in [1] with the implementation of the Kruijer tree as outlined in [2], with a focus on analyzing the number of tokens or turns given to any processor until the system self-stabilized at different scenarios with different trees and different state for every node in the tree.

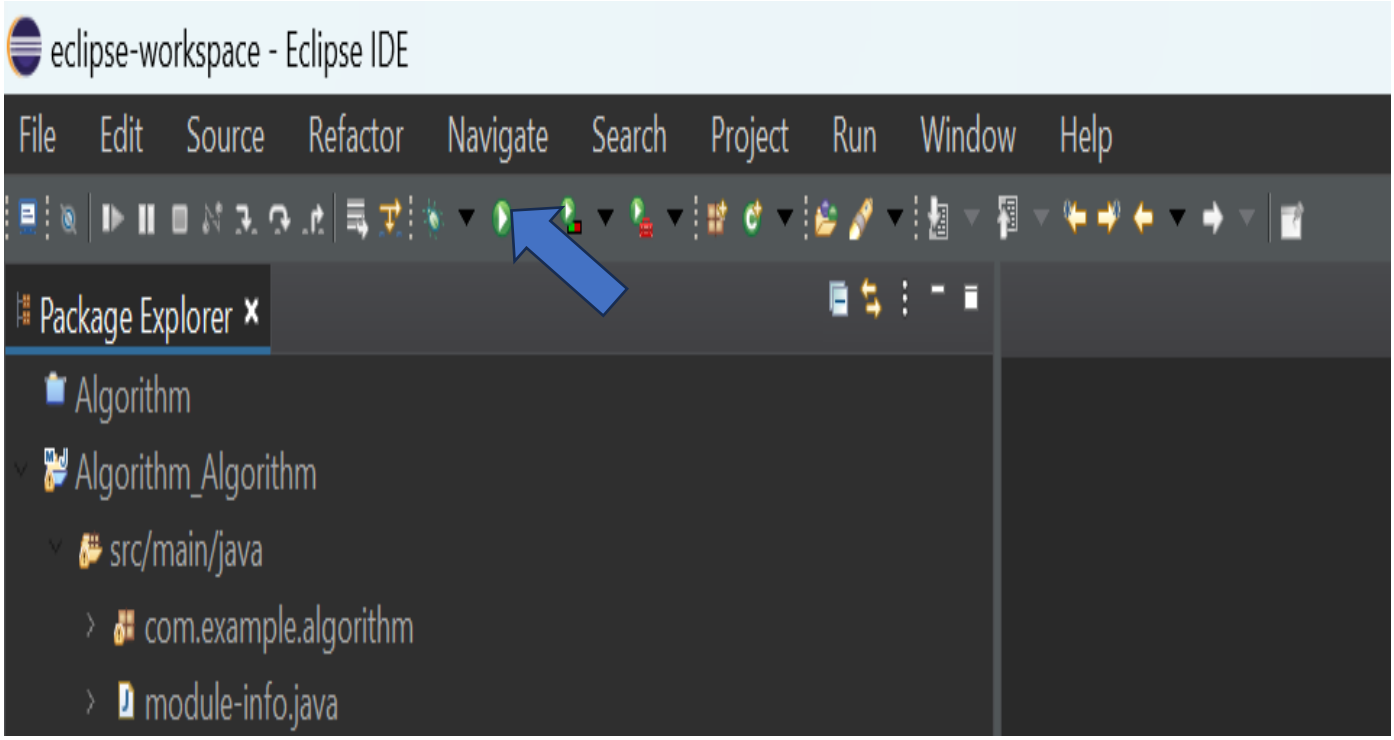
The primary objective of the software is to visually demonstrate the operation of the algorithm. The Four-state machines algorithm is described as it executes on a network of processors arranged in the Kruijer tree.

visualizing the algorithm execution offers an intuitive method to comprehend its functionality, thereby enhancing understanding. users can effortlessly construct a custom number of possessors using a user-friendly interface with randomly generated trees with different numbers of tree levels and random states for each node. The algorithm can be executed in either a step-by-step manner, by clicking the node the user gives the token to the selected node, or choosing automatic run. The status of each processor is represented by color: yellow indicates the node can get the privilege, green indicates privilege and have the turn, and red indicates can’t have the privilege.

The implementation of the algorithm is carried out in Java using Eclipse IDE version 2022-12, leveraging the JRE System library [JavaSe-1.8], and incorporating JavaFX with the javafx-sdk-19 library for the user interface (UI). User interface files are constructed using SceneBuilder version 19.0.0 software.

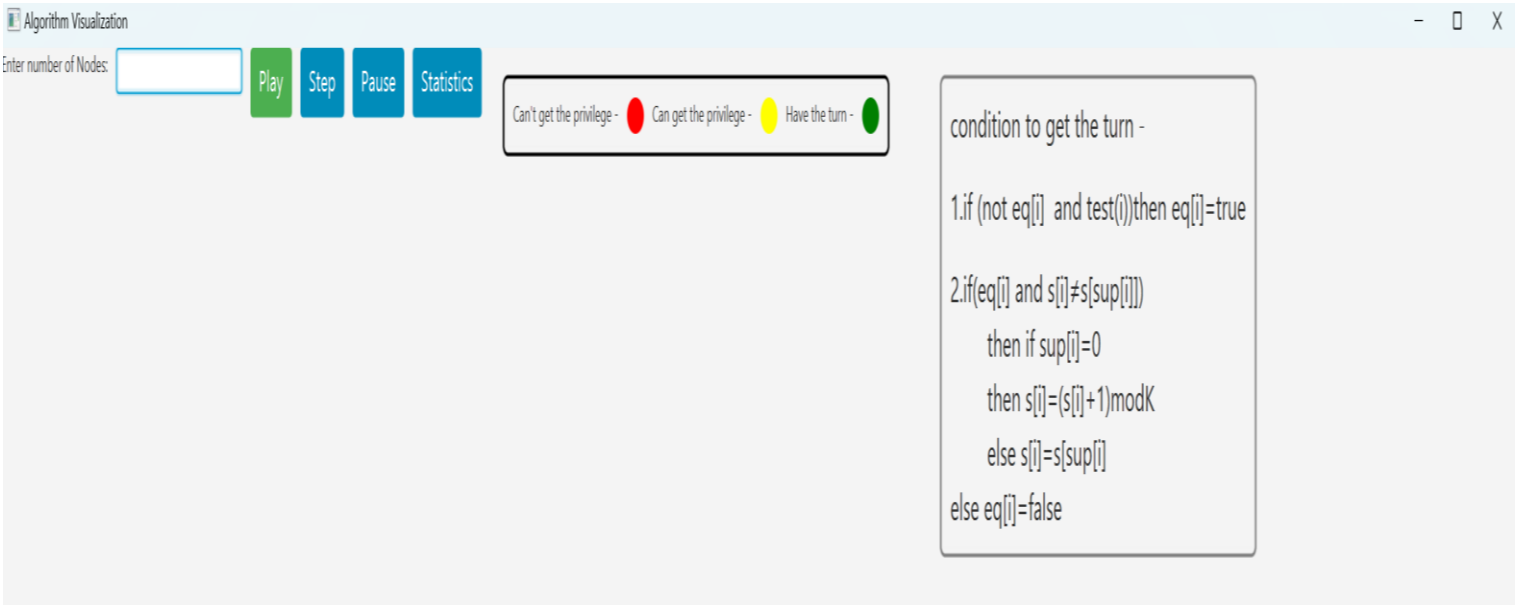
2.2.1 Run the software.

To run the software, you should click the green triangle in the Eclipse workspace with the correct IDE when the project is open.



2.2.2 The main page

Now we will see the main page of the software.

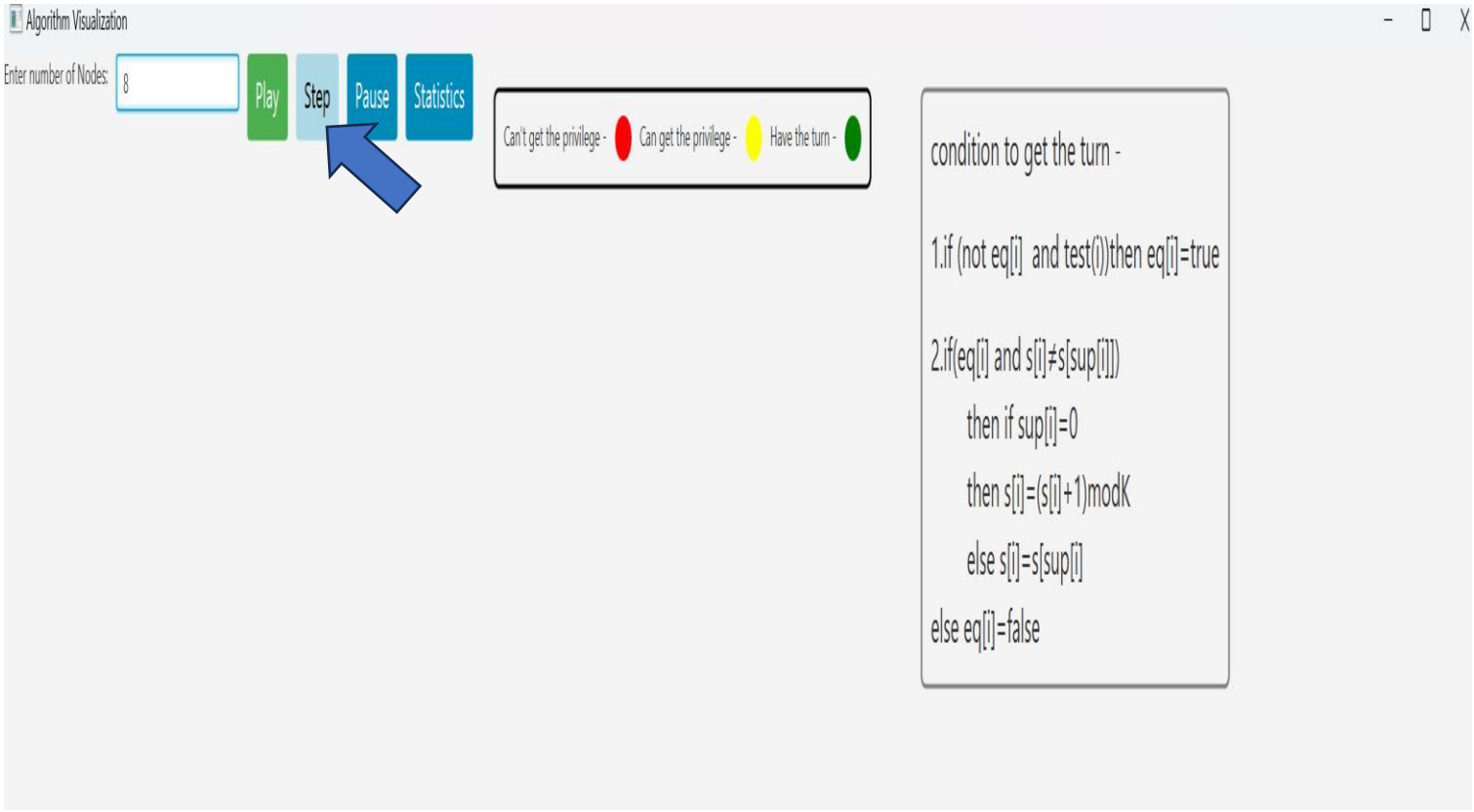


On this page the user can choose the method in which the algorithm will run, automatically or by steps, in addition, the user must enter the number of processors for all the simulations, as we can see the algorithm conditions and the meaning of the nodes color.

Additional options: pause simulation, see statistic, and play/run algorithm.

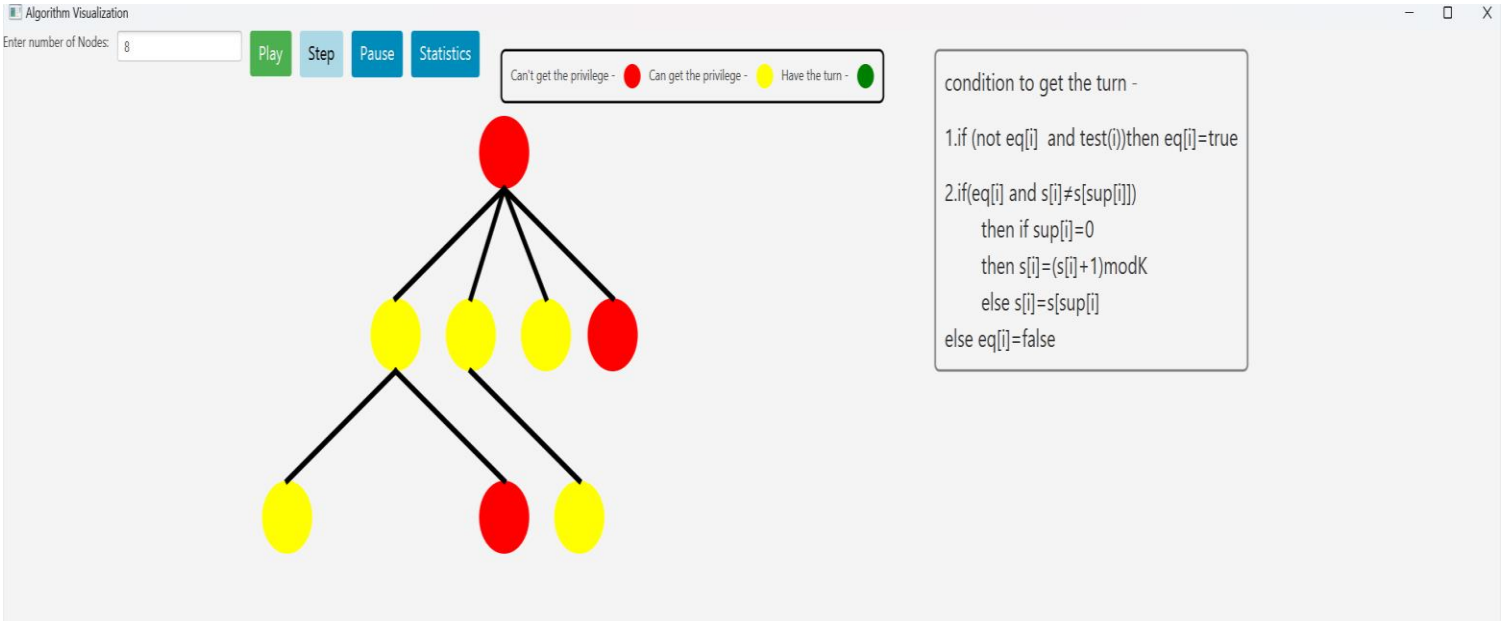
2.2.3 Manual simulation

In this part, we will see step by step simulation.



To pick the step simulation the user needs to click the step button and make sure the button is in highlight mode, choose the number of nodes, and click play.

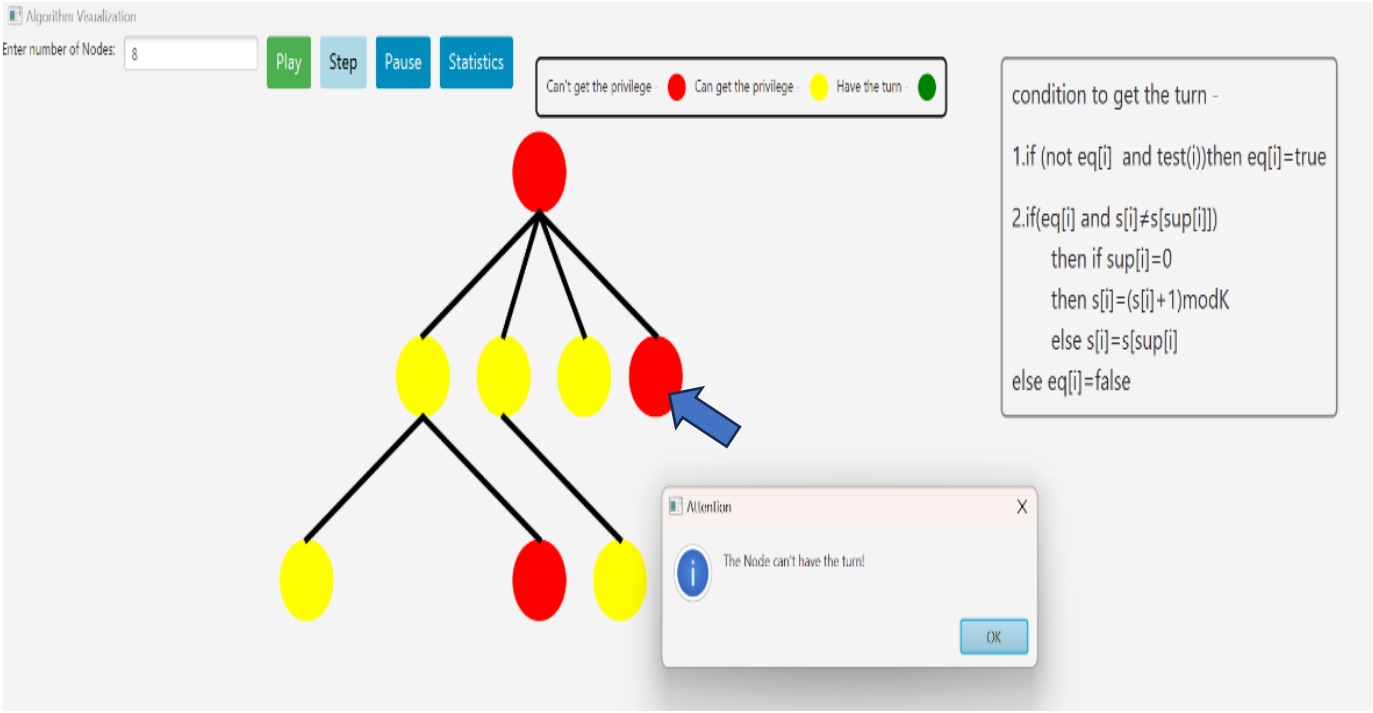
Here we can see the screen after clicking play in step mode.

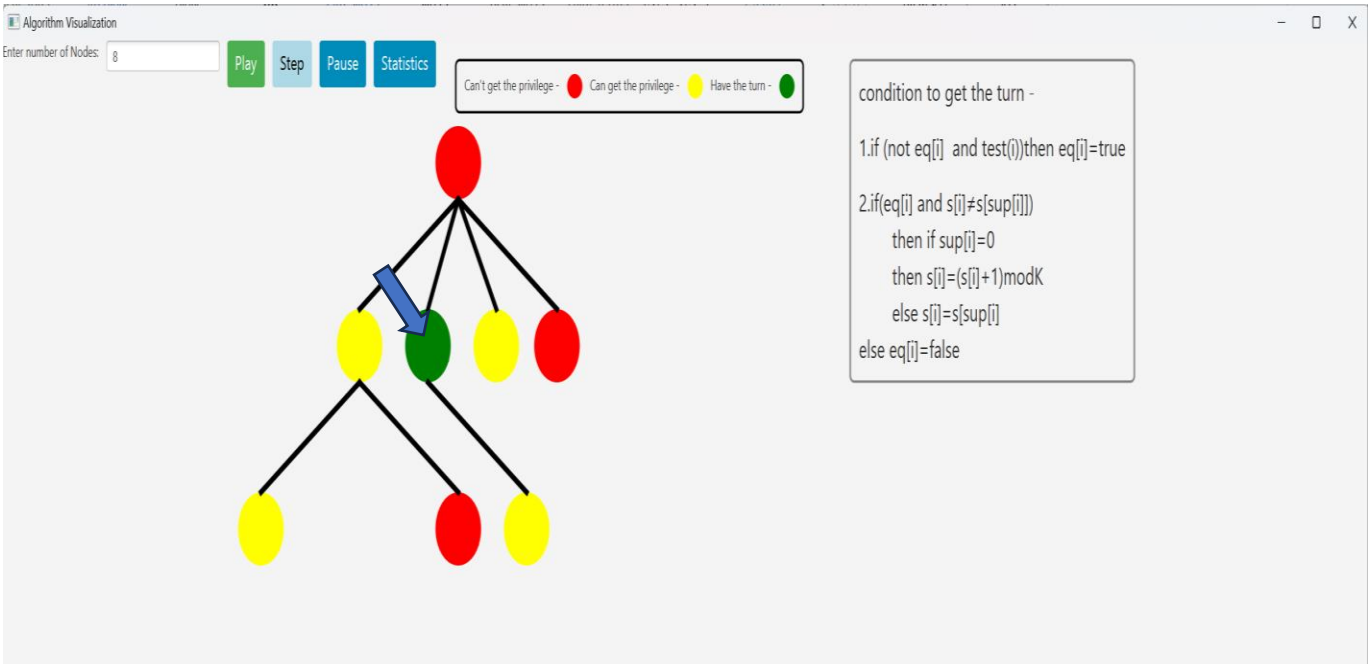


The simulator draws a random tree, and each of the nodes holds different values.

each node has a random number of children.

The user should pick and click the desired node to have the turn, he can choose an unprivileged node, and the software pops an alert message and informs the user of this impossible move and waits for a new pick.





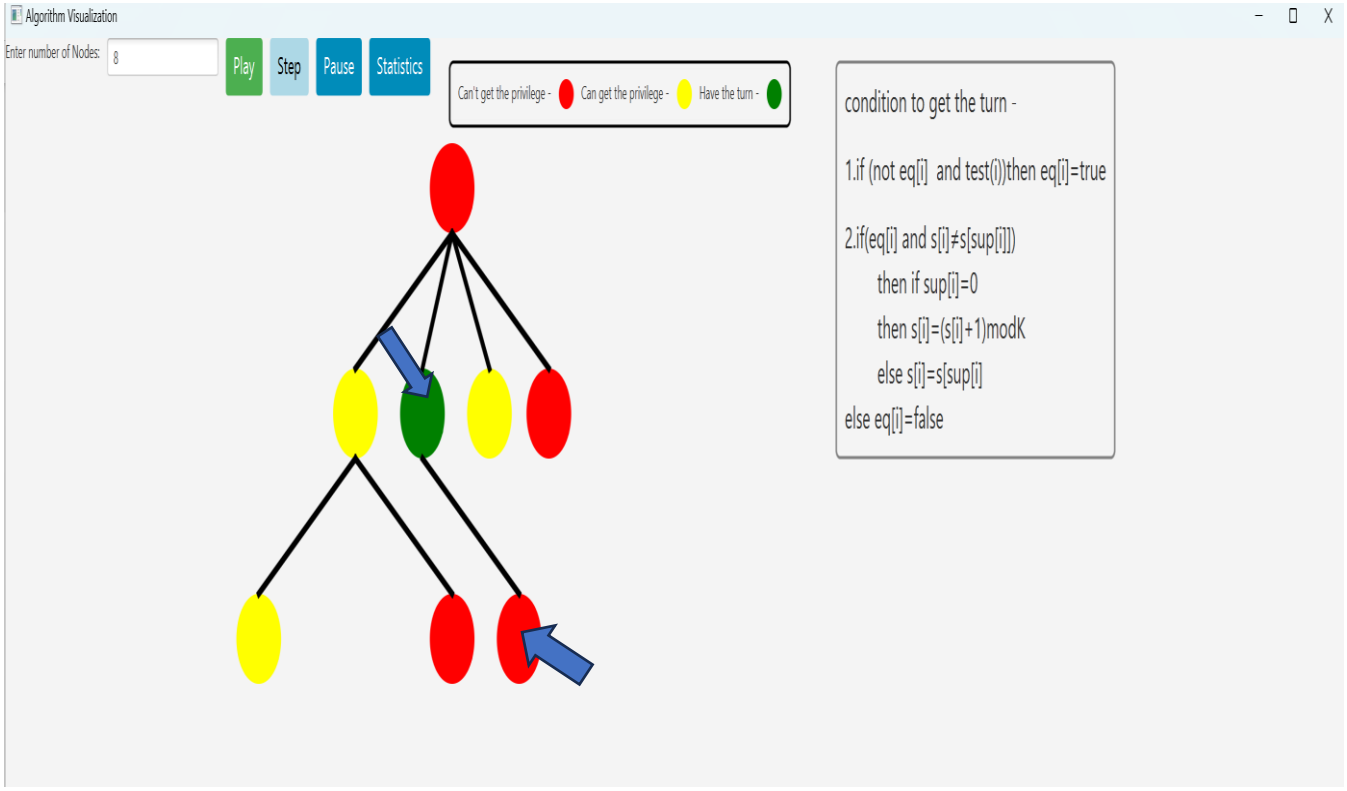
In the first figure on this page, we see the alert, in the second figure, we see the user pick the privilege processor, as the algorithm can continue running.

All nodes hold two Boolean values, as the user clicks on the privilege node, that node gets the token, and the algorithm runs and updates at least one of his values, just like in the condition on the right side.

Reminder: each node can have the turn only if it is privileged in these two conditions -

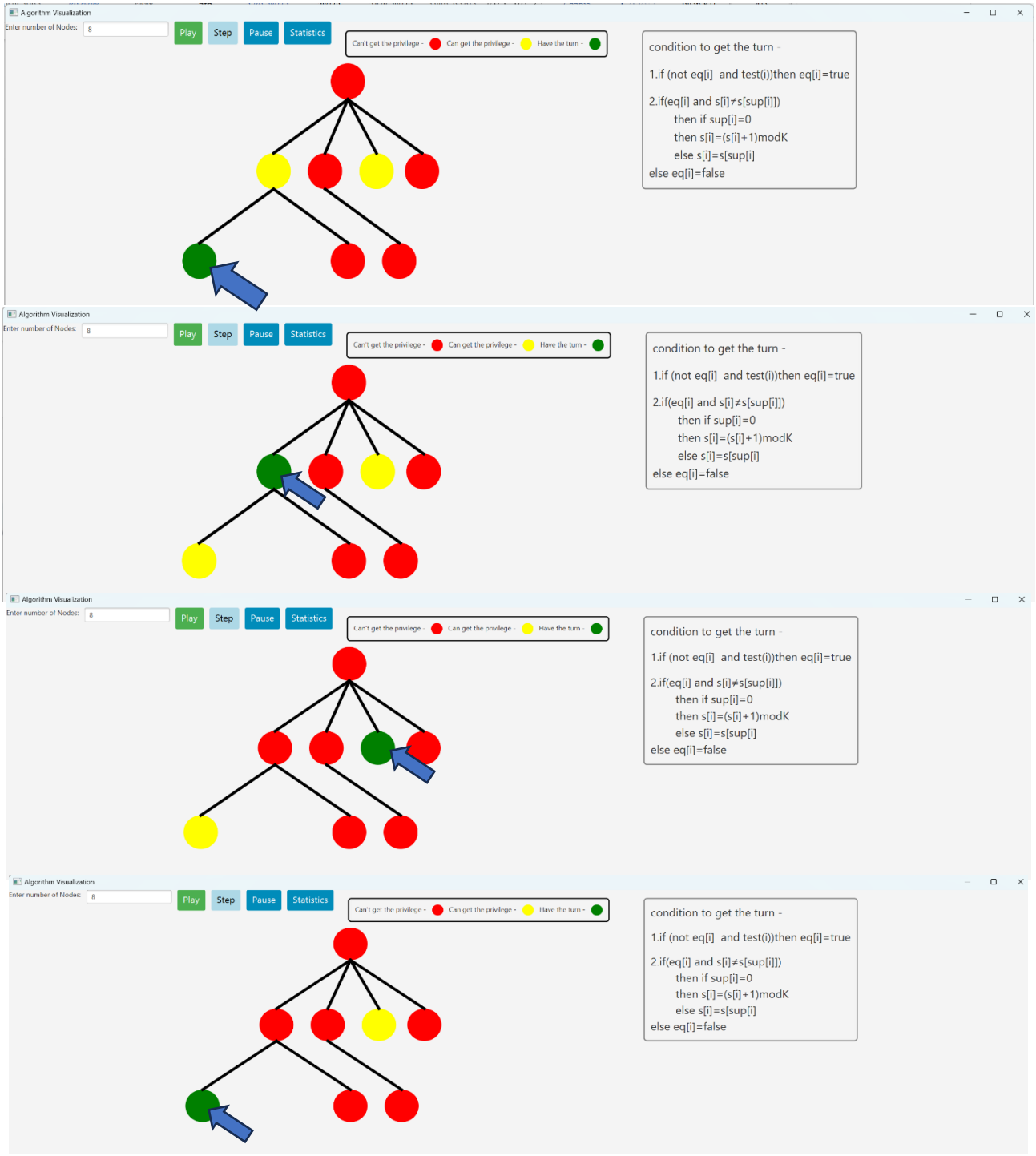
$$\boldsymbol{if\ (not\ eq[i]\ and\ test(i))}$$

$$\boldsymbol{if(eq[i]\ and\ s[i] \neq s[sup[i]])}$$



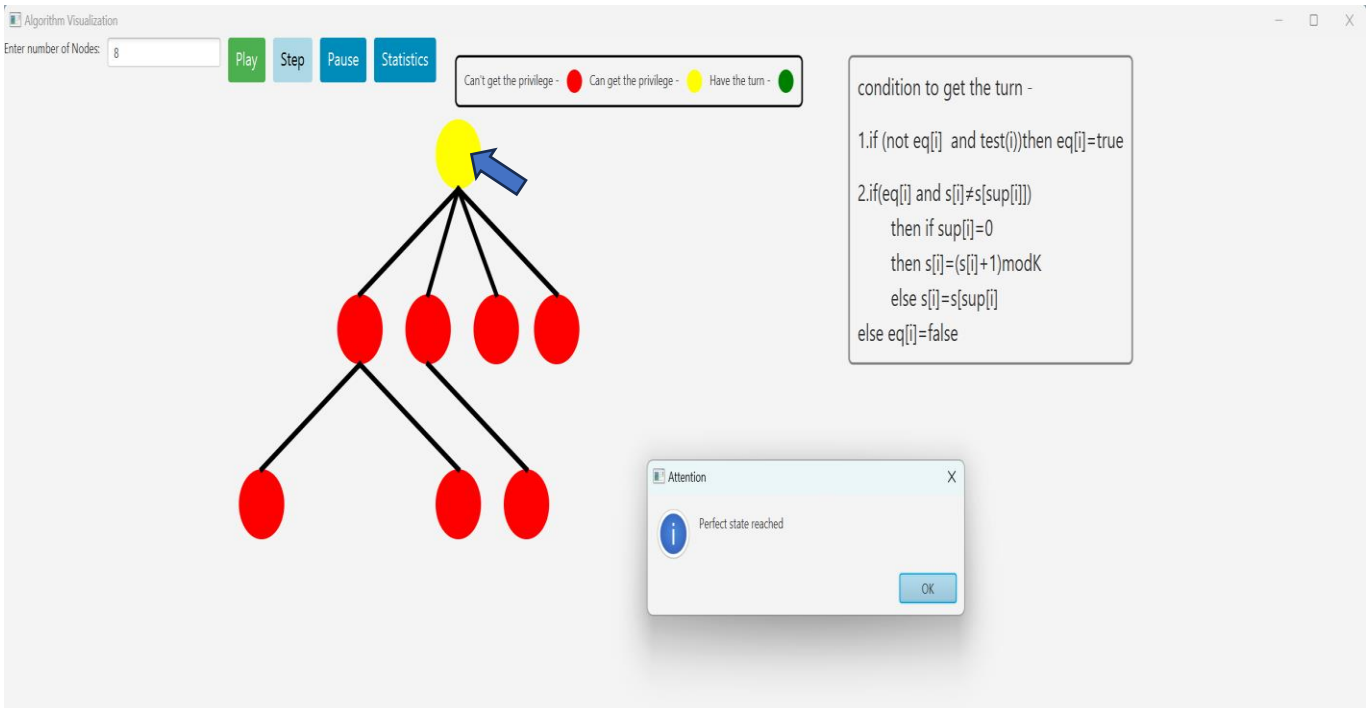
After clicking the last node and then back to over the first privilege we clicked we can see that the last node cannot have the turn and change its color to red, it happened because the algorithm updated one of its Boolean values and now the first privilege node has the token.

after we pass the token the green node will turn to red.



Some of available moves of passing the token by clicking before reaching the perfect state.

Reaching the perfect state

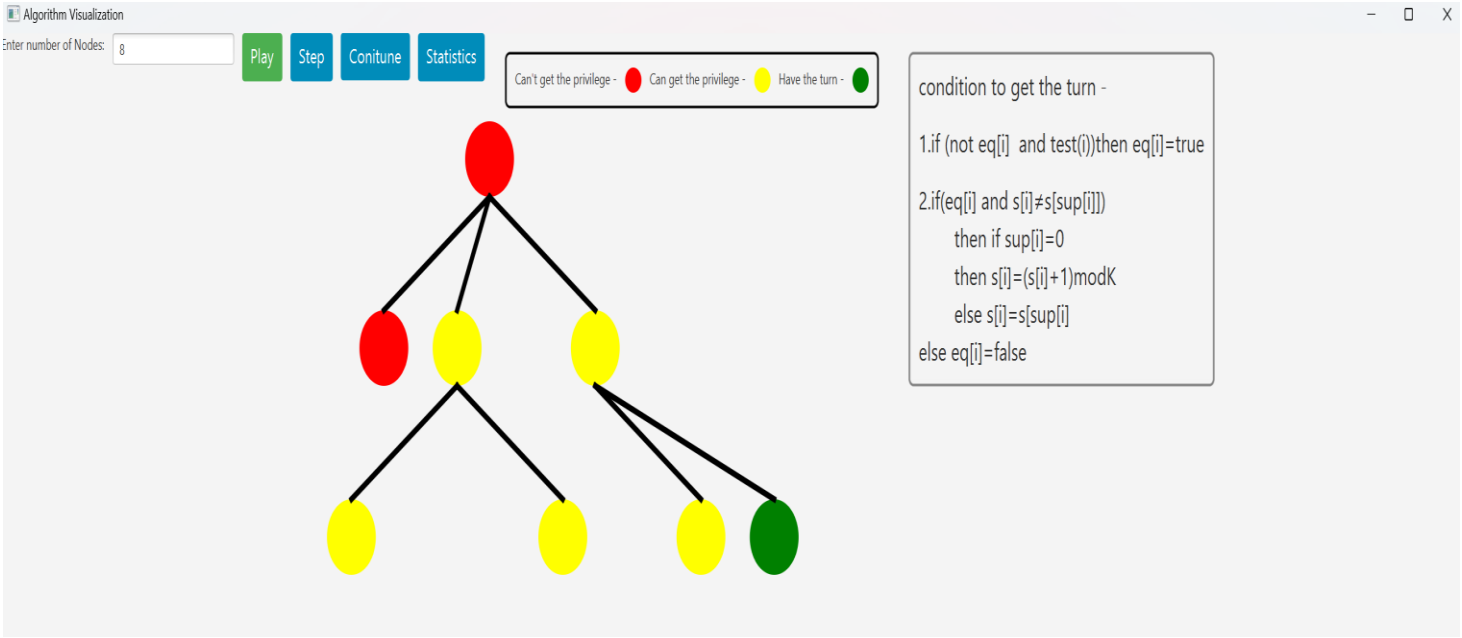


In this figure, we see the perfect state as all of the nodes are red (can't have the privilege), and the only processor that can have the turn is the root-
Reminder the perfect state is reached only when

– $s[i_1] = s[i_2] = \cdots = s[i_{n-1}]$ and $eq[i_1] = eq[i_2] = \cdots = eq[i_{n-1}] = true$

2.2.4 Automatic simulation

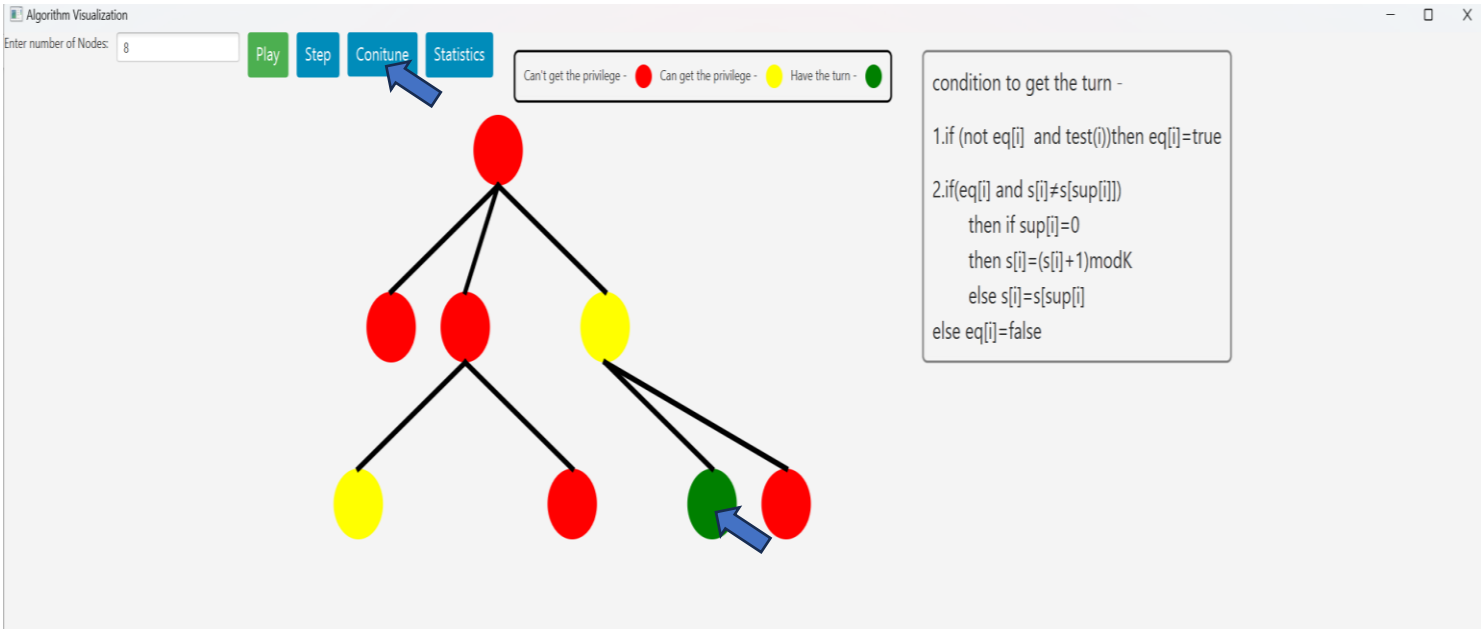
Now we will run a simple automatic run.



Just like before the user must enter the number of nodes.

and from now on the token will pass randomly between the privilege nodes.

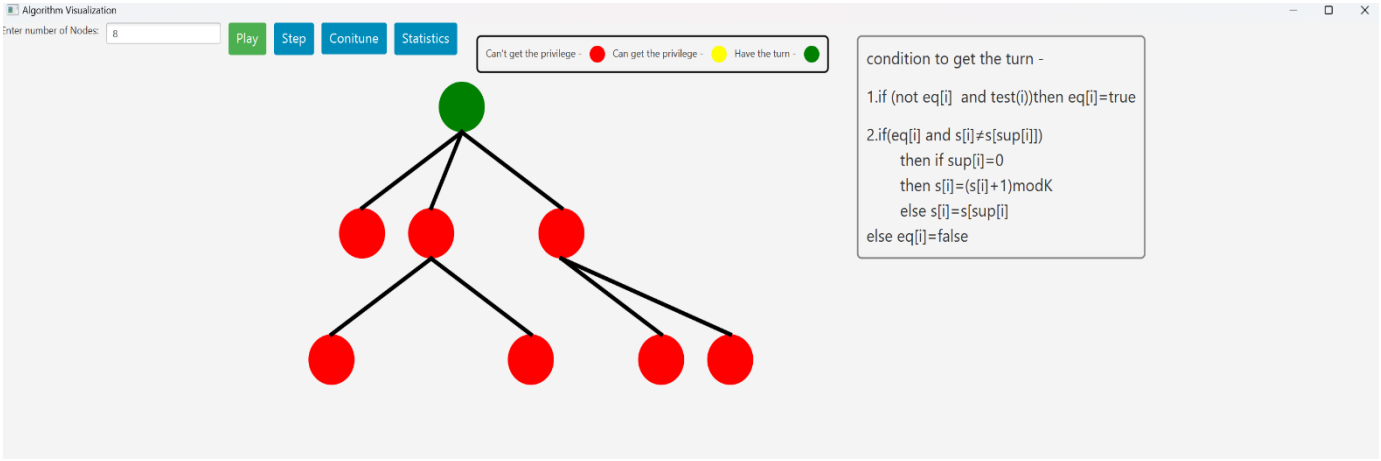
we can see that the step button isn’t highlighted because he is not clicked.



To stop the automatic run the user should click the pause button, and the text change to continue and when this button is clicked again the algorithm will run again from the pause point.

The user can notice the token now in a different node and some of them update their state to can't get the privilege.

Reaching the perfect state

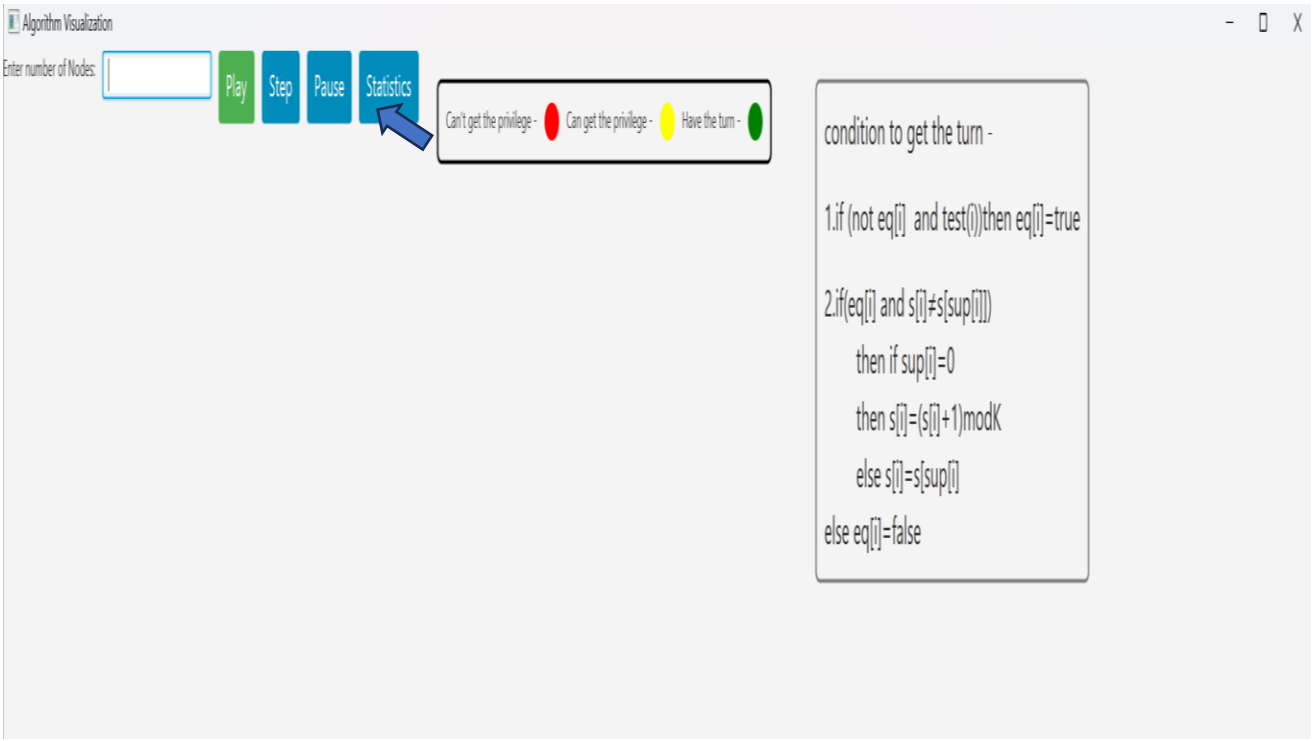


In the automatic run after reaching the perfect state the algorithm will continue to run infinite times until the pause button is clicked, we can see in this figure that the root gets the token after he was privileged at the end of the perfect state.

Reminder: after the perfect state is reached only the root can have the token.

When – $s[i_1] = s[i_2] = \cdots = s[i_{n-1}]$ and $eq[i_1] = eq[i_2] = \cdots = eq[i_{n-1}] = true$
the system will resume passing the token based on the algorithm conditions.

2.2.5 Graph and statistics

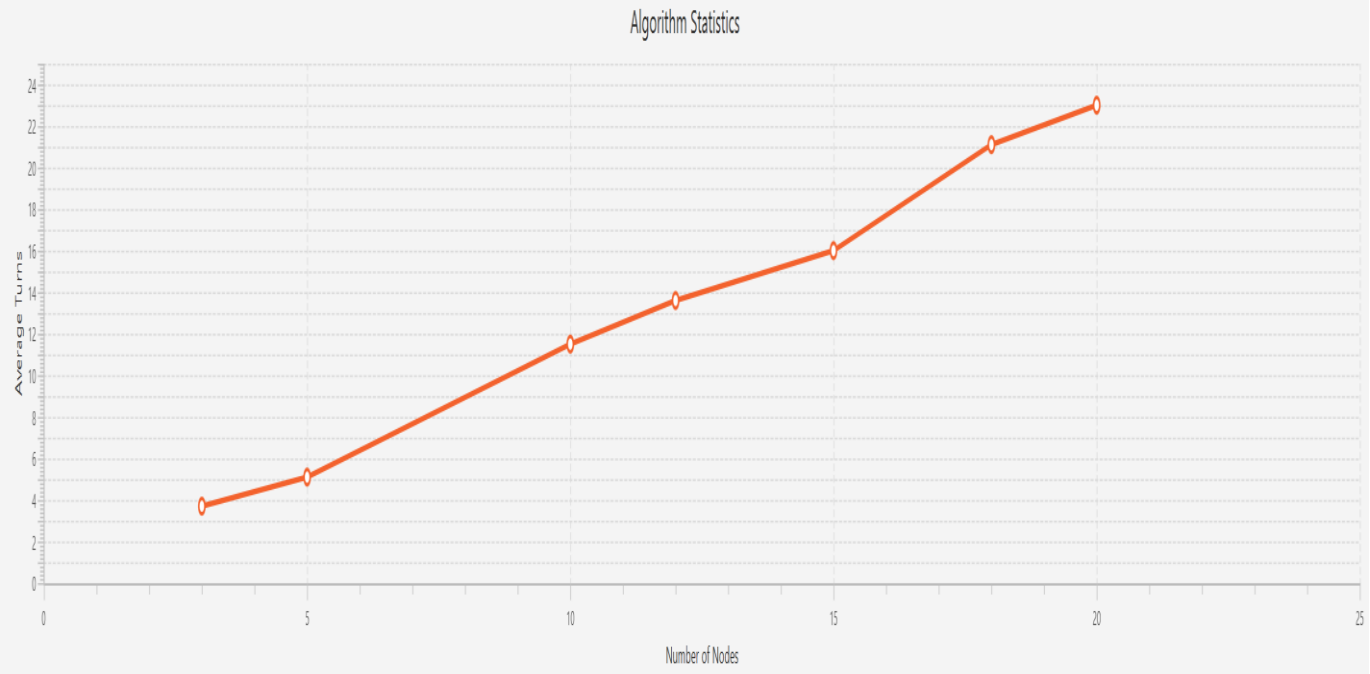


If the user desires to see the data the software collected on this specific Four-state machine algorithm he can click the statistic button, as we compare and analyze different runs of this algorithm by Dijkstra's implements in the Kruijer tree.
in Kruijer's article [2], he proved that –

Theorem 3 *if the system is in a perfect state, the next time the system reaches a perfect state will be after* $2 * n$ *actions.*

this means that for every situation of a tree and nodes, the system will stabilize at most after twice the number of nodes, we checked this proof in depth with the data we collected and according to the data the result is better, and the system will stabilize before reaching this barrier.

2.2.5.1 Line Chart



Here the user can see a line graph of the Four-state machine's algorithm based on the data we collected on a variety of tests on 5,10,12,15,18,20 nodes trees with random values for each node, on each run the algorithm stops when the network reaches perfect state and count how many actions it took. We calculate the average number of actions to reach the perfect state in all the different scenarios, as you can see the number of nodes on the x-axis and the average on the y-axis and we can see we are not even close to the upper bound in Kruijer proof.

2.2.5.2 Data Table

Statistics

Number of nodes	1	2	3	4	5	6	7	8	9	10	Average
3	2	6	4	5	4	4	3	4	2	3	3.7
5	6	6	3	8	5	4	5	5	3	6	5.1
10	10	13	14	13	10	7	13	17	10	8	11.5
12	11	17	15	17	12	10	14	14	12	14	13.6
15	21	11	13	21	18	13	18	17	17	11	16
18	24	20	18	21	19	23	22	25	21	18	21.1
20	22	24	29	27	20	18	21	19	26	24	23

In this table, we can see the data we collected for trees consisting of a different number of nodes (3,5,10,12,15,18,20).

where each tree is randomized in each test with nodes having a random value, and the token passed randomly.

we can see in each index of a column the number of times the token was moved or in other words the number of actions made until the system is stabilized, while in the last column we can see the average of all the tests for the specific number of nodes.

in this data table, we can see the Kruijer Theorem, furthermore in most of the tests we conducted the system will self-stabilize after near n actions, in fact only in one case we reach perfect state in $2 * n\ actions$.

additionally, we had many cases where the system reached a perfect state in less than n steps.

In conclusion, according to our table the theory that Kruijer proved was correct, and even in most cases the result was better.

3 References

[**1**] DIJKSTRA, E. W. 1974. Self-stabilizing Systems in Spite of Distributed Control.
Communications of the ACM 17, 11, 643-644.

[**2**] HSM Kruijer. Self-stabilization (in spite of distributed control) in tree structured.
systems. Information Processing Letters, 8:91–95, 1979.