Visualization of Network Dynamics

A document submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science
at
Goethe University Frankfurt

Abstract

This goal of this project was to create an interactive website, that visualizes different network dynamics. This tool can help people understand more complex networks and concepts. The tool will allow users to dynamically explore network data and reveal patterns and trends that are not apparent from static representations of the network. Various visualization techniques will be implemented, such as node-link diagrams and time-series plots, to provide a comprehensive view of network dynamics. Additionally, the website allows users to interactively manipulate and control network parameters, to observe the effects on the network behavior.

Contents

I Introduction

The visualization of network dynamics has become increasingly important in recent years, as the study of complex systems has gained popularity in various scientific disciplines. Networks, such as social networks, biological networks, and communication networks, are omnipresent in our daily lives and can be represented as interconnected nodes and edges. Network dynamics refers to how the connections between nodes change over time, which can reveal important insights into the behavior and evolution of the network. [Bar16]

Visualizations of social media network data are an essential tool for discovering and interpreting structures and patterns of behavior[DLH10] and can help to identify patterns and trends that might not be apparent through static representations [Sco12]. This can help in the discovery of new phenomena and relationships, and can also aid in the prediction and control of network behavior.

Moreover, the ability to visualize network dynamics can help wider audiences to understand complex systems. The visualization of network dynamics are helpful for identifying events that might not be perceivable in a static representation. [KB07] This can and already has important applications for addressing real-world problems such as disease outbreaks, transportation networks, and social networks.

In summary, the visualization of network dynamics is a crucial tool for analyzing and understanding complex systems, and can provide important insights into the behavior and evolution of networks.[JS12] It has important implications for a range of scientific disciplines and real-world applications, and is likely to continue to be an area of active research and development.

2 Preliminaries

This chapter provides an overview of the necessary background and definitions used in this thesis.

2.1 Networks

The dynamic networks $G = (V, E)$ considered throughout the project will be undirected and connected.
Each vertex v ∈ V has attributes. These attributes of a vertex represents the changing factor of the current network type, for example opinions in an opinion network. The edges E display a relationship between two vertices.
The first network gets initialized and then saved in an array of graphs T. The index of this array represents the time step. Whenever a protocol gets executed on a network in T, the network gets changed and the results are saved at index step+1. This way, the network can be represented at different stages in time.

2.2 Protocols

Several different network protocols are implemented and visualized in this project. As established earlier, the state of a vertex, and therefore the state of the entire network, may change on a time step. Network protocols are a set of rules, which determine how a network changes in one time step. An edge between two vertices may alter the attribute of either of them, when the protocol gets executed.

2.2.1 Rumor spreading

The network represents a group of people, with each node representing one person. The state of each vertex represents, whether a rumor is known or unknown. The rumor spreading model simulates the spreading of a rumor among a social group. [DD64]

```
input  : G = (V, E)
output: G = (V, E)
for v ∈ V do
    if v.state == known then
        for neighbor ∈ v.neighbors do
            neighbor.state ← known;
        end
    end
end
```

2.2.2 SIR model

The SIR model is an epidemic model that is quite similar to the rumor spreading model. It simulates the spreading of an illness among a social group. The state of the vertex represents whether the person is susceptible, infectious or recovered. The chance of an infectious person to infect one of its neighbors is denoted in β. The chance of an infectious person to recover is γ.

```
input  : G = (V, E)
output: G = (V, E)
for v ∈ V do
    if v.state == infectious then
        for neighbor ∈ v.neighbors do
            if neighbor.state == susceptible  infectionChance < β then
                neighbor.state = infectious
            end
        end
        if recoveryChance < γ then
            v.state = recovered
        end
    end
end
```

2.2.3 Glauber dynamics

Glauber dynamics is a protocol that simulates the Ising model. The vertices of the network represent atomic particles, which can either spin up (+1) or spin down (-1). Each particle has neighbors, which are represented by the edges of the network. The glauber dyanmics simulates the particles change of spin on time, by implementing the following ruleset:

```
input  : G = (V, E), T, n
output: G = (V, E)
for i ← 0 to n do
    v ← pickRandomVertex (V);
    sum ← v.state;
    for neighbor ∈ v.neighbors do
        sum ← sum + neighbor.state;
    end
    ΔE ← 2· v.state · sum;
    flipChance ← e^{−ΔE/T}/(1 + e^{−ΔE/T});
    flip v.state with probability flipChance
end
```

2.2.4 VOTER MODEL

This network represents a social group, similar to the rumor spreading model. The key difference is that the attribute of each node represents the opinion of the person, regarding a certain topic. The implemented protocol simulate the change of opinions in a social group over time. This voter model was introduced by Richard A. Holley and Thomas M. Liggett[RAH75]:

```
input  : G = (V, E), n
output: G = (V, E)
for i ← 0 to n do
    v ← pickRandomVertex (V);
    u ← pickRandomVertex (v.neighbors);
    v.state ← u.state;
end
```

2.2.5 MAJORITY MODEL

The h-majority model is a more complex version of the voter model. The network is a social group and one vertex gets randomly picked. Next not one but h neighbors of the vertex get picked. Lastly, the opinion, that is most represented among the h neighbors and the vertex, gets picked by the vertex. If there is a tie, which can happen if h is an odd number.[Gal90]

2.3 IMPLEMENTATION

The networks and protocols are implemented using a combination of javascript and the d3.js (Data Driven Documents) library. The library is used to create interactive data visualizations. The network, network dynamics as well as data plots, to visualize the change of the

```
input  : G = (V, E), n
output: G = (V, E)
for i ← 0 to n do
    v ← pickRandomVertex (V);
    u ← pickRandomVertices (v.neighbors);
    max ← mostOpinions (u,v) v.state ← max
end
```

network over time, are all created and handled using d3.js. This project is partially built on the color refinement project by Holger Dell.[Del20] The used functions from color refinement draw an interactive network and update and detect setting changes, for example the number of vertices in the network. Every other aspect of this project is self-implemented, using this foundation.

3 Implementation

This chapter goes indepth on how the networks, protocols and visualization are implemented and handled.

3.1 State

One of the key features of this project is to store all parameters needed for the generation and handling of the networks in the URL. This allows for identical networks and outcomes of network dynamics to be generated with the same URL, allowing easy sharing of networks. These parameters are stored in the state object in the defaults file. The function updateState is used to stringify, encode and set the state object as the window.location.hash. The getState function decodes the hash back into the state object. This way the state can be recieved and updated while also updating the URL. To keep track of changes during the last updateState call, the getStateChanges function is utilized. All three state functions and their respective helper functions are adopted from Holger Dell's color refinement project.
The parameters in state are used to generate new networks and execute protocols on them. To ensure the independence of network generation, coloring, and protocol execution, the project uses three different seeds.

3.2 Network

The network object is composed of two arrays: one for vertices and one for edges. Each vertex is an object in itself, containing a name, which is a number between 0 and n-1, a neighbors array, which are vertices saved in an array, and a level, which keeps track of the dynamic attribute of the vertex and is used to color the vertices and a fix property which is used for opinion protocols.
The network gets initialized with empty vertices and edges arrays. The colorSeed from the state object is used to initialize a random number generator. The vertices are then created by numbering them from 0 to n-1, and giving them a random level between 0 and numberOfColors - 1. Each vertex's neighbors array is empty and the fix variable is set to false.

To ensure a connected network, a random walk algorithm is used to add edges. irst, an empty visited set and a set of unvisited vertices containing all vertices are created. One vertex named currentVertex is then chosen by a random function utilizing the networkSeed from state. This vertex is added to visited and removed from unvisited. The random walk repeatedly picks a new vertex, named nextVertex. If nextVertex is not in visited, an undirected edge between currentVertex and nextVertex is added. Additionally the neighbors array of both vertices are updated to include the other vertex. Afterwards nextVertex is added to visited. The currentVertex then gets replaced by nextVertex. This continues until every vertex has been visited. This continues until all vertices have been visited. With this algorithm, every generated network is guaranteed to be connected, and will have at least n-1 edges.

If the random walk is finished and m is greater than n-1 new edges get randomly added to the network. This is done by picking two vertices with the same random function used during the random walk. If there is no edge between them, a new edge gets pushed to edges. Otherwise two new vertices get picked. This is done until the network contains either m or the maximum number of edges. This approach works fine for sparse networks and small to medium sized networks, which are the networks this project covers. This approach would be too slow for large, close to complete networks, since the chance of picking two vertices that are not connected are getting increasingly small.

This results in an undirected, connected network at time state.t = 0.

3.3 NETWORK NAVIGATION

The user can choose various different protocols by interacting with the navbar on the left hand side. These protocols are then executed by clicking the forward or the start/stop button on the bottom of the screen. The forward function does one of two things; if no changes are available for the current step, the protocol function gets executed and the changes are saved at changes[state.step]. Otherwise, the function uses the changes in changes[state.step] to repeat the saved change. The backwards button functions in a similar way. It applies the changes in changes[state.step-1] to iterate backwards through the changes. The start/stop button simply executes the forward function, in an interval chosen by the user, until clicked again.
Additionally to the forward, start/stop and backwards buttons, the user has the option to directly enter the new time step t' they want to jump to. This allows for a fast transition to a certain step. The program executes the forward function until the step is reached. A large

jump can cause lags for the user interface, since Javascript only uses a single thread to handle every process on the website. One way to avoid this is to throttle the forward function by setting up a timeout after a few execution, to keep the user interface responsive. While this approach certainly does work, the timeout duration has to be set to at least 50 milliseconds per step. This makes the approach inefficient, since it would take 50 seconds to execute the protocol 1000 times. An alternative and much more efficient approach is the utilization of the web worker API.

3.4 WEB WORKER

A web worker is utilized in this project to perform heavy computing outside of the main thread, which results in a fast execution and a responsive user interface. The worker consists of a worker file and a worker instance in the main thread. The worker file represents the second thread and cannot import any of the functions or variables from the main thread. Communication between the two threads is possible through worker.postMessage() and worker.onmessage() functions, which transfer data between the threads. Functions cannot be transferred, so protocol functions are copied to the worker thread. The protocol functions can simply be copied to the worker thread. Howeever, the network, random function and changes need information from the main thread. The random function can be used by messaging the protocol seed to the worker thread and generating the random function in the worker file. The changes array is instanced and updated in the worker file, too. The network transferation posed more of a problem. At first the current network was transferred with the postMessage function. This worked and the worker function was able to use the network and execute the protocol functions on it. A problem occurred, where the vertices picked by the random function differed, depending on the steps taken. This resulted in two different networks, when either skipping ten steps at once or going one step at a time for ten steps. The issue resolved, when saving an instance of the current network in the worker file and accessing and updating it individually from the main thread network. With this setup the worker is capable of executing any protocol. After the protocols have been executed, the new network, new changes and the new state get posted back to the main thread.

3.5 NETWORK AND CHANGES ARRAY

During the development of the program, a crucial decision was whether to save the entire network or only the changes that occur between two steps. Both approaches have advantages and disadvantages. Saving the entire network for each step consumes significantly more space,

as every vertex has to be saved for every time step. The changes array, on the other hand, takes up less space as it only saves the relevant vertices and their changes. It is perfectly suited for going one step forward or backwards at a time. The changes array is suitable for going one step forward or backward at a time, but jumping to a particular time step may be less efficient, as the program has to iterate through every change until it arrives at the desired step.
However, this project uses a web worker to execute the network dynamics, which makes the difference in speed between the two approaches negligible. As a result, the changes array was implemented for this project. Nonetheless, it is worth noting that if the user needs to jump to a time step that has already been calculated, the networkArray approach would be faster, as the user can access it in O(1) time.

3.6 PROTOCOLS

The protocols object contains all the protocols with a pickVertices, pickNeighbors, and changeProtocol function. They are used in the protocolExecution function to execute one protocol step. Firstly, the pickVertices function selects a specified number of vertices and places them in an array. The number of vertices can be modified by the user in the UI. Next, the pickNeighbors function takes this vertices array and creates an object with each vertex as a key and an array of the selected neighbor vertices as its value. The neighbors chosen and the method used to select them are dependent on the protocol that is being executed. After that the actual protocol is being executed by calling the changeProtocol function. This function changes the level values of the chosen vertices and saves them. Several visualization functions are used between these three functions, to visualize and showcase the changes happening throughout one protocol step. Section 4.2 offers a closer look at the functions and methods utilized.

3.6.1 VOTER

The voter protocol is a type of opinion dynamics protocol that operates as follows. First, a set number of vertices is randomly selected from the graph.vertices array. The random function is generated with the state.protocolSeed. This ensures that the protocol execution is independent of the coloring and generation of the network and allows the user to execute the voter protocol on the same network, but with different vertices picked. Next, the pickNeighbors function is used to randomly select a number of neighbors for each vertex. In the case of the voter model, the number of neighbors to pick is set to one. It then returns an object where each vertex is a key and their corresponding neighbor is the value. This object is then used to execute the changeVoterOpinion function. The function first iterates through the neighbors object and checks if the fix value of the vertex is set to false. If it is, the vertex adapts the level

value of its neighbor. If it is not, the vertex is skipped since its opinion cannot be changed. This process is repeated for each vertex in the randomly selected set.

3.6.2 MAJORITY

The majority protocol is similar to the voter protocol in terms of implementation. However, there are some key differences. The pickVertices function picks state.numberOfVertices vertices randomly from the graph.vertices array using the random function generated by the state.protocolSeed. Then, the pickNeighbors function is called, which picks state.majority number of neighbors for each vertex randomly from its neighbors array. The pickNeighbors function also uses the random function generated by the state.protocolSeed. The vertex/neighbors pairs are stored in an object with the vertex as the key. The changeMajorityOpinion function is then called to execute the protocol. It iterates through the vertices and checks for the fix value. If the fix value is false, the function counts the different opinions of the vertex and its chosen neighbors. The vertex then adopts the opinion with the highest count. In case of a tie, a random opinion is selected from the list of highest opinions. Overall, the majority protocol allows for more complex opinion dynamics compared to the voter protocol. The user can control the number of neighbors to be considered, which can lead to different outcomes and opinion clusters.

3.6.3 RUMOR SPREADING

The rumor spreading protocol functions differently from the two opinion protocols mentioned earlier. The graph starts out with one or multiple vertices with vertex.level=0. The pickSpreaders function picks every vertex in the graph with graph.vertices[vertex].level = 0. These vertices know the rumor and want to spread it during the next time step. The pickSpreaderNeighbors function takes the list of vertices and returns every neighbor. The changeRumor function then changes the level of every selected neighbor to 0, making them a spreader for the next round.

3.6.4 SIR MODEL

The SIR model is an epidemic model used to simulate the spreading of a disease in a social group. It works quite similar to the rumor spreading protocol, but is more complex. The network starts out with one or more vertices being infectious and the rest being susceptible. The infected vertices have a level of 0, susceptibles are level 1 and recovered are level 2. When an infected comes into contact with a susceptible, there is a chance β that the suscpetible will become infectious for the next time step. Additionally, each time step, the infected recover with

a rate γ. β and γ can be adjusted by the user to view differences. The pickSpreaders function picks every single vertex with level 0, just like the rumor spreading version. The pickSpreaderNeighbors function picks every single neighbor of the spreader. The changeSIR function then turns the susceptible neighbors into infected vertices with chance β. Afterwards the infected vertex turns into a recovered vertex with chance γ.

3.6.5 GLAUBER MODEL

The glauber model functions similarly to the opinion models with the addition of chance. Each vertex has a downspin (level 0) or an upspin (level 1). The protocol uses the pickVertices function from the opinion protocols, to select state.numberOfVertices vertices. Afterwards it uses the pickSpreaderNeighbors function to select every neighbor for each vertex. The changeGlauber function then takes each vertex/neighbors pairing and calculates the flipChance for the vertex. Afterwards, a random function generated with the state.protocolSeed generates a number between 0 and 1. If it is lower than the calculated flipChance, the vertex changes its spin, otherwise it stays the same.

3.7 SYNCHRONOUS AND ASYNCHRONOUS

Each protocol can additionally be executed in a synchronous or asynchronous way. The asynchronous mode executes the protocol on the picked vertices one by one. This means that the changed value of a vertex can affect the result of the next vertex picked. Synchronous mode, on the other hand, changes the values of every single vertex at the same time. Therefore no vertex change effects the other vertices in the same protocol execution.

4 Results

This chapter focuses on the user interface and the visualization of the website. The user inter-

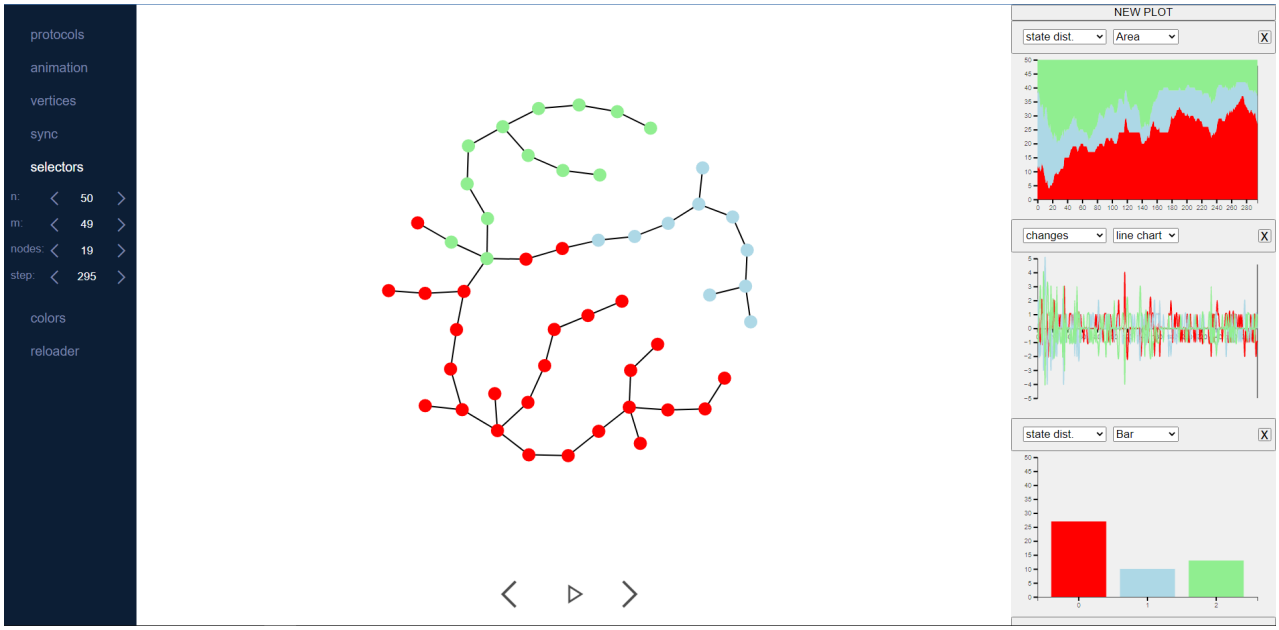

Figure 4.1: The website divided into main area, navbar, plotbar and controlbar

face of the website is divided into four sections; the main area, marked red, where the network gets drawn, the navbar, marked blue, where network and protocol settings can be adjusted, the plot bar, marked green, where data plots get drawn and the control bar, marked yellow, where the protocols can be executed.

4.1 Navbar

The navbar is located on the left hand side of the website and allows the user to interact with the network and protocols. The navbar is divided into two parts; protocol settings and network settings.

4.1.1 PROTOCOL SETTINGS

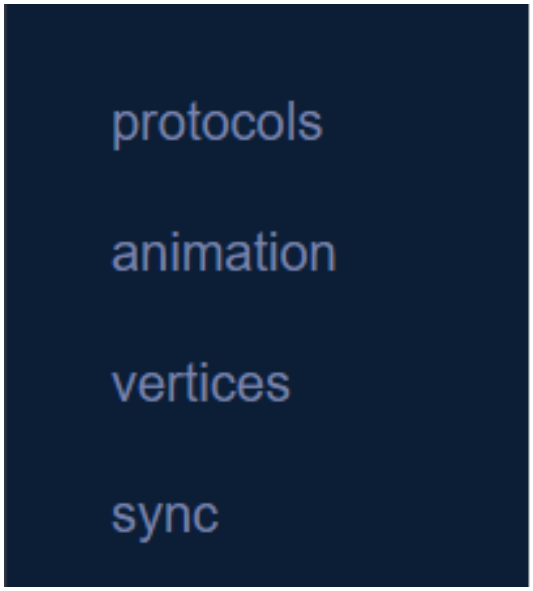

Figure 4.2: available protocol settings

The protocol settings allow the user to select a protocol from the available list. The user then can select one of the animation options to fit their needs and display the selected protocol in different ways. The last protocol setting allows the user to switch between synchronous and asynchronous mode. The speed bar dictates how fast the protocol is executed.

4.1.2 NETWORK SETTINGS

The network settings allow alterations of the network that the protocol is being executed on. The user can either increase or decrease the values by pressing the arrow buttons on screen. Alterantively the user can enter a new number by inputting it at the display of the value. This way users can make changes quickly and efficiently. The state step allows users to jump to the inputted step. This way users can simulate thousands of steps without having to iterate through every single one of them. At the bottom of the network settings is a color selection tool which allows the user to change the preset colors of the network to fit their needs. The colors can be changed by click the select tool of the corresponding color. The colors on the network as well as the plots change instantly, which allows the user to preview the color before making a choice. At the bottom of the navbar is a button to switch between light and dark mode.

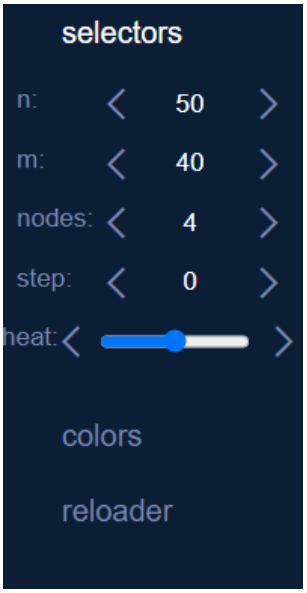

Figure 4.3: available network settings

The navbar can be opened and closed with the button attached to it.

4.2 NETWORK AREA

The network gets randomized using the user settings and then drawn onto the main area. The vertices are color depending on their vertex.level and the corresponding colors[vertex.color], that the user can adjust in the navbar. The vertices of the network can be interacted with. The user can click the vertices to highlight them. When rightclicking any vertex a context menu appears. This menu allows the user to unhighlight every vertex, fix or unfix the currently highlighted vertices, unfix every vertex or change the color of the selected vertices.

4.2.1 NO VISUALIZATION

If the user just wants to skip through the data, they have the choice to select the no visualization option, letting them immediately see the next step without delay.

4.2.2 SIMPLE

The simple mode first grays the entire network out. It then highlights the chosen vertices by returning their color and giving them a black border. After that it picks the selected neighbors and highlights them too. The protocol then gets executed and the color of the first vertex may

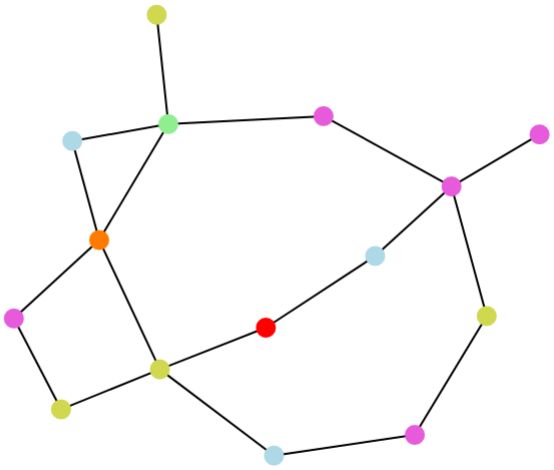

Figure 4.4: example opinion network

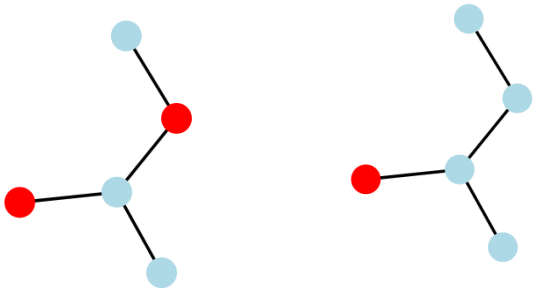

Figure 4.5: Without animation

get adjusted. Lastly the grayout and highlights of the network get lifted. This visualization works well when the user picks a few vertices per round. The closer the picked vertices go to n, the more confusing this approach gets.

4.2.3 ANIMATION

The animation approach works similarly to the simple visualization, but adds edge and vertex animations to make the process more understandable. At first the entire network grays out again and the picked vertices are highlighted. Then the picked vertices lose their color and an edge animation shows the neighbors. The same animation is used to go back to the original

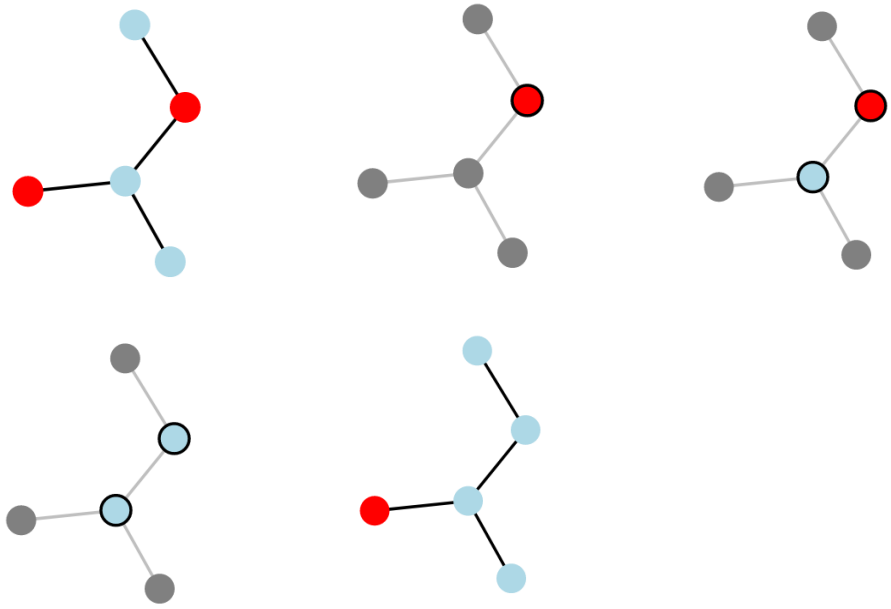

Figure 4.6: Simple animation from top left to bottom right

vertex and color them accordingly. This approach works better than the simple approach, especially with more vertices picked.

4.2.4 SELECT OPTIONS

The user has the option to visualize the protocol execution of only one vertex, only the ones that have changes or every picked vertex. This is only a visual effect and the protocol will be executed on every selected vertex, regardless of the chosen visuals.

4.2.5 ASYNC VS SYNC

The difference between asynchronous and synchronous protocol execution is shown by delaying the animations of the vertex and edge animations. This way, if a vertex is both picked as a vertex to change and a neighbor, the changes of the vertex will be visualized before the animation to change the next vertex starts.

4.2.6 FIXED OPINIONS

Fixed opinions are visualized by having a square shape instead of a circle.

4.3 CONTROL BAR

The control bar is located at the bottom of the screen. It contains 3 buttons to control the protocol execution. The forward button executes the chosen protocol once and animates it using the settings from the navbar. The start/stop button executes the forward function until pressed again. The backwards button undoes the last step in the network dynamic.1

4.4 PLOT BAR

The plot bar is located on the right hand side of the screen. The plot bar contains a single ”new plot” button. When the button is pressed, a new button appears underneath it. This button represents a new plot. The button can be interacted with in two ways. When pressed, the button reveals the svg to the corresponding plot. Secondly, the type of displayed plot can be changed by clicking the name of the plot and choosing a different type of plot. The user can generate multiple plots and set them up individually to display the data in different ways. The plots can be deleted by pressing the 'X' button in the top right corner. The plots can be interacted with by zooming into them and panning by dragging left or right. This allows users to view the data in more detail, especially when a lot of steps are displayed at once. Another feature is that the current step is shown by a vertical line. This helps to keep track of the current step. The user can click anywhere on the chart to jump the that time step. There are several different d3 plots to visualize both the current state of the network and the dynamic changes over time. The user can choose between a line chart, an area chart, a bar chart and a pie chart.

4.4.1 LINE CHART

Line charts display the network data over time. The x-axis represents state.step and the y-axis represents the chosen data of the network at state.step. The line chart draws the progress of the network state over time.

4.4.2 AREA CHART

Area charts work similar to line charts. The difference is that the areas are stacked ontop of each other, filling out the entire chart.

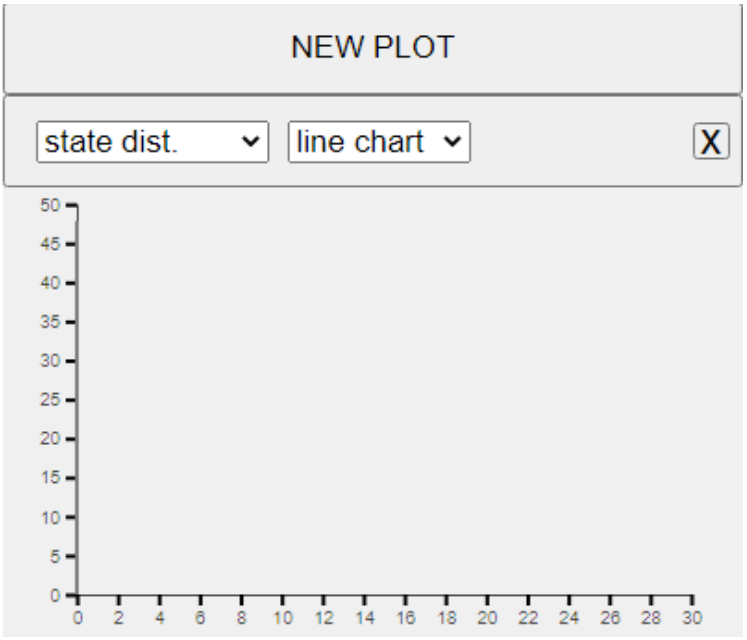

Figure 4.7: empty plot. the user can choose the plotType and dataType with the select menus.

4.4.3 ADDITIONAL PLOTS

The static plots, namely the pie chart and bar chart, represent the data of the current network. It is not possible to extract information about the network dynamics from these plots, but they have their use in displaying the current state.

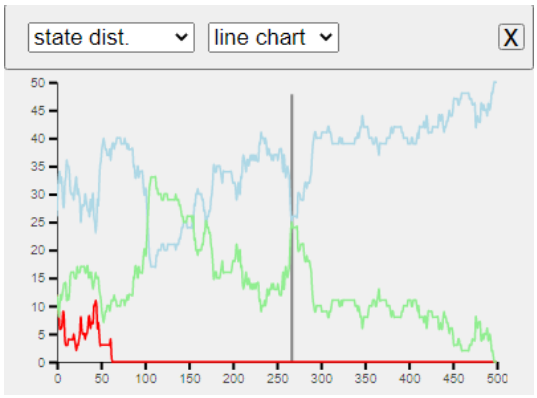

Figure 4.8: example line chart, displaying the state distribution over time. The vertical line shows the current step.

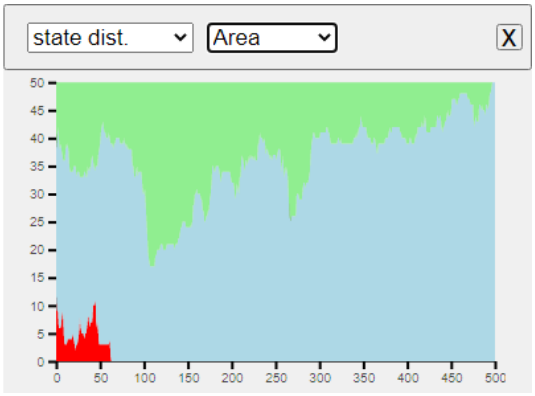

Figure 4.9: example area chart, displaying the same network dynamic as in Figure 4.8

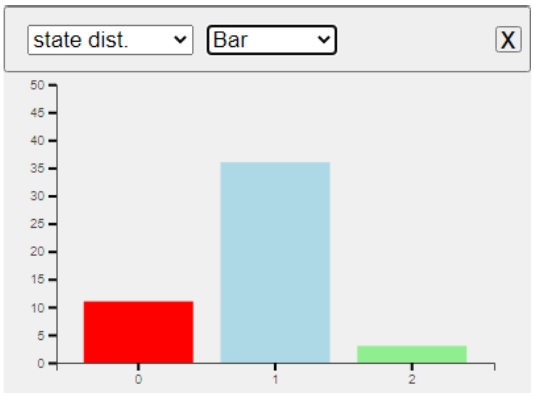

Figure 4.10: example bar chart

5 Conclusion

The goal of the project was to simulate and visualize different network dynamics on small to middle sized sparse networks. The project implements various protocols. The user has many different possible ways to adjust the network and protocol execution. The protocols are visualized with different animations and the data is presented through different plots. These plots are interactive and deliver additional insight on the dynamics and current network state. The project runs entirely in the browser and is compatible and tested in Mozilla Firefox, Google Chrome and Microsoft Edge. Javascript and the d3 library were used for the entire project. Users can send each other networks by share their URL. All of these goals were implemented and reached with this project. The performance of the website suffers as soon as many steps of a big network are simulated. The performance is satisfying for small step sizes. The project does not provide a solution for visualizing many vertices in a protocol step at once. The best alternative is to only visualize the changes or to sample one vertex. It is not possible to upload networks into the website. It is not possible to compare two plots with different outside variables, for example two glauber dyanmics with different temperature, directly. This project can be used as a base for a visualization project of network protocols on big, connected networks. There are still more network types and protocols that could be visualized, like GNNs or hyperbolic graphs.

Bibliography

[Bar16] Albert-László Barabási. *Network Science*. 2016.

[DD64] D. G. Kendall D.J. Delay. *Epidemics and Rumours*. 1964.

[Del20] Holger Dell. Color refinement. https://holgerdell.github.io/color-refinement/#seed=bytwb, 2020.

[DLH10] Marc A. Smith Derek L. Hansen, Ben Shneiderman. *The visual display of quantitative information*. Elsevier, 2010.

[Gal90] Serge Galam. *Social paradoxes of majority rule voting and renormalization group*. 1990.

[JS12] Petter Holme Jari Saramäki. *Temporal networks. Physics reports, 519(3), 97-125.* 2012.

[KB07] Alessandro Vespignani Katy Börner, Soma Sanyal. *Network visualization*. 2007.

[RAH75] Thomas M. Liggett Richard A. Holley. *Ergodic Theorems for Weakly Interacting Infinite Systems and the Voter Model*. 1975.

[Sco12] John Scott. *Social Network Analysis*. Sage Publications, 1 edition, 2012.