Accumulation Problem

Aidela Karamyan
Ani Sakhlyan
Sipan Muradyan

College of Science and Engineering

American University of Armenia

Yerevan, Armenia

May 9, 2016

Submitted as partial fulfillment for Communication Networks course

TABLE OF CONTENTS

1 Accumulation Problem 3
  1.1 Problem Definition 3
  1.2 Applications 4
  1.3 Accumulation on Specific Graphs 4
2 Investigation Program 9
  2.1 Program Structure 9
  2.2 Graph Generator Tool 10
  2.3 Main Application 12
3 Future Work 16

References 17

1 Accumulation Problem

Accumulation is the "reverse" of broadcasting. Here instead of one vertex passing message to all other vertexes the information flows in the opposite way, all vertexes need to pass their message to one designated vertex. This kind of problem can reflect real life examples when a communication strategy and optimal topology are needed for collecting some sort of data from network participants. Examples of such cases can be collecting sensor data in a network, designing reporting strategy in an organization and more.

1.1 Problem Definition

Given a graph G with set of vertexes V and edges E and a vertex $s \in V$. All the vertexes except s contain a distinct piece of information. The communication strategy in which s, let's denote it the *server* vertex, will learn the cumulative message from all $v \in V$ is called accumulation [1].

Beside the main definition there can be several models worth considering. First important criteria is whether a vertex can simultaneously send and receive messages or can do only one action at a time. This communication models are denoted *2-way* and *1-way* mode respectively.

More models can be determined depending on communication buffer criteria, namely how many messages can be sent or received using given connection. Possible models based on this criteria are following:

1. a vertex v can send and receive only 1 message
2. a vertex v can send 1 message only, but receive up to *deg(v)* messages
3. a vertex v can send up to *deg(v)* messages, but receive only one
4. a vertex v can send and receive up to *deg(v)* messages

In model 2, sine communication line buffer limits the number of passed messages, the more than 1 messages received are received from multiple sources. Similarly in model 3, the more than 1 messages are sent to multiple targets. Remarkable that in model 2 accumulation time is equal to the broadcast time on the same graph, with same vertex taken as the server node and the originator. In model 4 accumulation time is equal to the diameter of the graph, at each time unit all messages stored in the vertex are passed forward so accumulation time is determined by the longest path needed to pass.

Vertex storage space can also be a criteria for model determination, vertex with limited storage cannot receive more messages until freeing some space by sending messages.

1.2 Applications

An example case where accumulation problem can be useful is a post office transportation topology. Sometimes it is not justified to send one unit directly to the center (center is the analogy of server node), it can be due to transportation costs or traffic considerations. In this kind of situations units need to be accumulated in local spots, than sent to regional spots, than spots of larger size areas and finally to the center.

Weather station is an example when processing needs to be done on the way to the center. Suppose we have sensors for collecting weather station data. Data from the sensors need to be passed to the middle-ware nodes in the network where it will be filtered. After filtering it can be passed to another layer of middle-ware nodes where some kind of decision making can be done. And finally the processed data will be passed to the center.

Reporting in organizations is a combination of previous two cases. Obviously it is not practical for everyone to directly report to the higher management, so the reporting is done in levels. Lower level employees report to their direct management, those managers report to higher level management and so on. Also at each level some processing can be done such as making reports based on the received reports.

1.3 Accumulation on Specific Graphs

We chose following model for accumulation problem investigation: communication buffer 1-1 mode with unlimited vertex storage buffer and 1-way communication mode. Reason for choosing this model is that it is the most complicated and challenging. Chosen model is reflecting real life problem with hardware limitations, which can occur even with the advanced hardware when huge amounts of data need to be passed as one piece of information.

Unlike broadcasting each edge passes multiple pieces of information in different time units, and vertexes can have many states depending on the data stored. So to ease steps description during accumulation process we have developed a *table method* for determining the states of the vertexes at each time unit. Row number defines the states of the vertexes at that time unit, very first row determines states at time 1 and so on. We assume that at time 0 each vertex only has its own message. Column number determines vertex number.

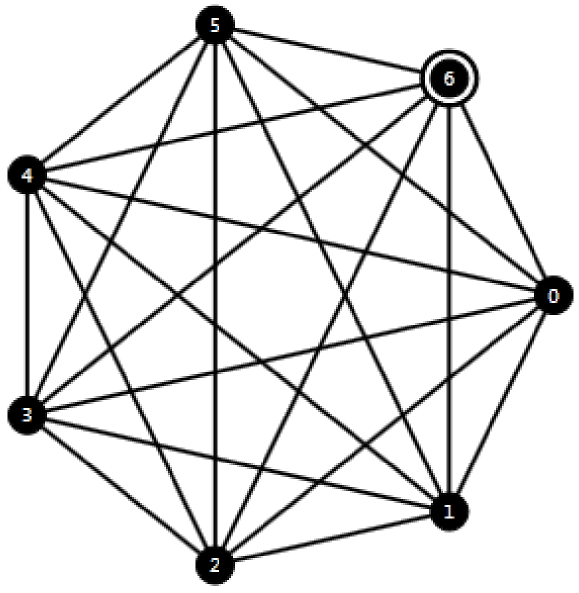
Now let's discuss accumulation problem on several graph typologies and determine the best possible case to complete the task.

Complete Graph

Complete graph is the simplest case for accumulation. With the chosen model it will take $(n-1)$ time units to pass all messages to the server. Server can receive only one message at each time unit, and it is not meaningful to send message to anywhere other than server since it will not optimize communication anyhow.

Figure 1: Complete graph

Source: This and all subsequent graph figures are created/generated in our application and saved with Windows screen print feature

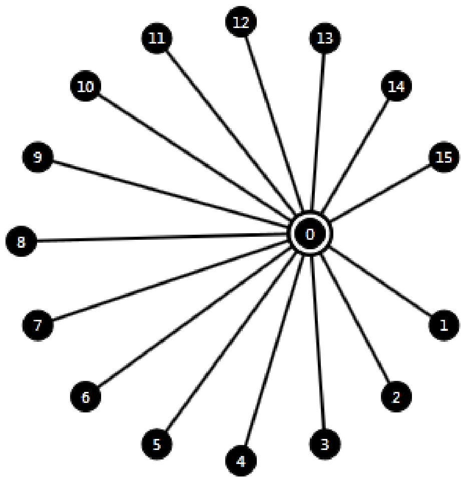


Star Graph

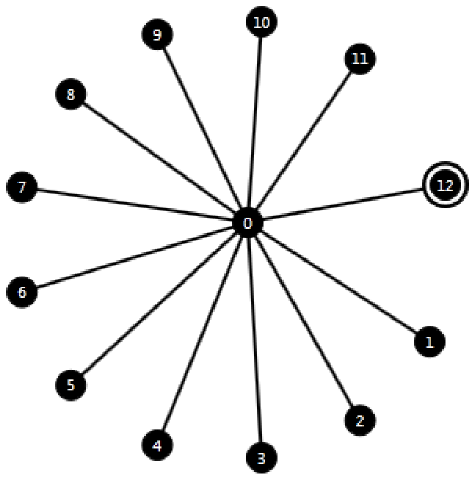
In the star graph we have two different server choices. In first case pictured in figure 2 (a), server is the central node. This is similar to complete graph case and accumulation time is $(n-1)$. When we choose one of the star leafs as a server vertex, see figure 2 (b), all messages need to be passed through the center of the star and one at a time only. It will take $(n-2)$ time units to pass all massages to the center and another $(n-2)+1$ for center to pass all the messages including its own message to the server. Overall, the accumulation with worst case server for star graph is $2*(n-2)+1$ time units.

Figure 2: Star graph

(a) Central vertex as server



(b) Leaf vertex as server

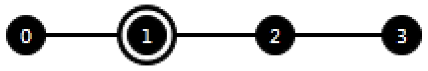


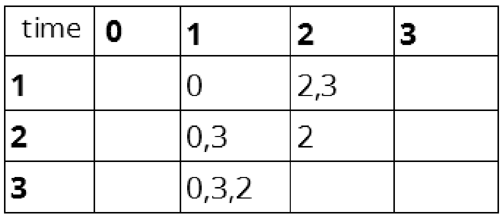
Path Graph

In the path graph case the accumulation scheme is different depending on where the server is located. It can be one of the edge vertexes or any vertex up to the center of that path. Odd or even vertexes can also impact the accumulation time.

Broadcast scheme could be a one similar to the star case, messages are passed sequentially from one end towards the server. That takes $n * (n-1) * (n-2) * ... * 2 * 1$ time units, which is too long. However turns out that dividing path into pairs of vertexes at each time unit can improve accumulation time. At first time unit vertex closest to the server will send its message, the next two vertexes on the path will also communicate one message towards the server, and so on.

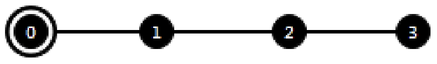
Figure 3: Path graph with central server, even number of vertexes





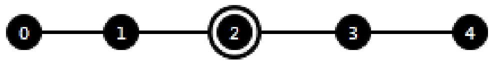
time	0	1	2	3
1		0	2,3	
2		0,3	2	
3		0,3,2		

Figure 4: Path graph with last vertex as server



time	**0**	**1**	**2**	**3**
1	1		2,3	
2	1	3	2	
3	1,3		2	
4	1,3,	2		
5	1,3,2			

Figure 5: Path graph with central server



time	**0**	**1**	**2**	**3**	**4**
1		1,0		3,4	
2		1	0	3,4	
3			0,1	3,4	
4			0,1,4	3	
5			0,1,4,3		

An example of this scheme with server on one of the last vertexes on an even graph is described in the figure 4's table. Cases when server is not one of the last vertexes is similar, however we deal with 2 paths that need to access the server sequentially.

3-ary Tree with Depth 3

Tree pictured in figure 6 is the closest topology to the application examples brought in subsection 1.2. Broadcast scheme for this specific example is following:

- All leaves transfer their messages to their parent vertexes; server's direct children pass their messages to server $: 3$ time units
- Then second level vertexes transfer all their messages to the first level vertexes $: (3 + 1) * 3$
- Finally first level vertexes transfer all the collected messages to the server $: (3 + 1) * 3 * 3$
- Overall $3 + (3 + 1) * 3 + (3 + 1) * 3 * 3 = 51$ time units

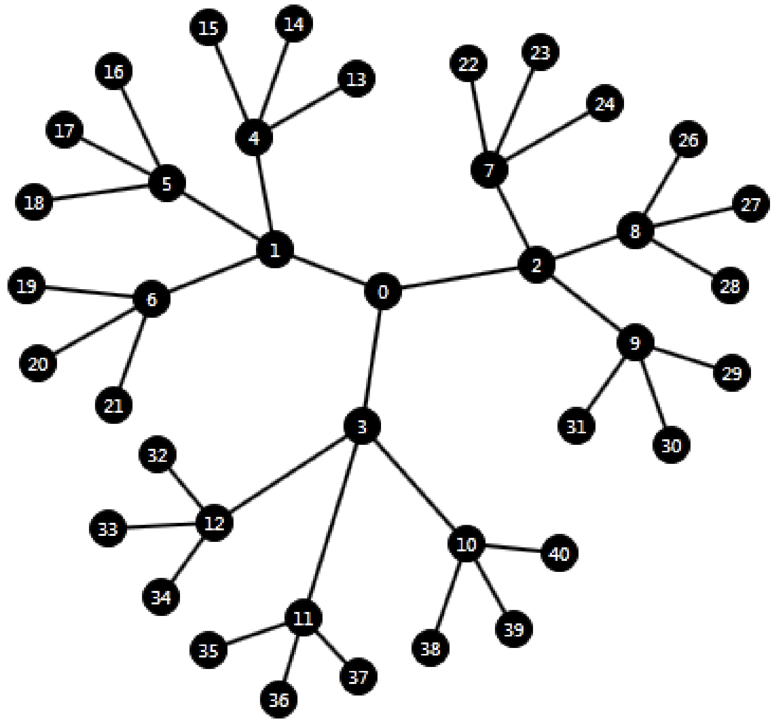


Figure 6: 3-ary Tree with Depth 3

2 Investigation Program

Accumulation is a hard problem in terms of investigation on the paper, compared to broadcasting. Classical broadcasting has one piece of information and in each edge is activated only ones. This makes possible using the paper and pencil method for picturing broadcasting process on the graph itself.

Since accumulation has multiple pieces of information the process is radically different. During the accumulation process each edge can have up to *n* (*n* is the number of nodes in the graph) number of pieces of information passed depending on where the edge is located. As a result it is impractical to picture the whole process on the paper in the same way that classical broadcasting is pictured.

The table method we introduced in subsection 1.3 makes investigation easier. However, since it lacks visual connection with the graph, method requires to jump your eyes from the graph to the table and back during accumulation process multiple times. Investigation process turns out to be not very productive using this method.

Given this difficulties, in the scope of this project we developed a new application which will make investigation process easier. It offers following main features

- conveniently draw and manipulate graphs
- auto-generate common graph types or random graphs
- simulate accumulation process steps in an interactive way

2.1 Program Structure

Our application consists of two parts: command line tool for generating graphs and main application that has all 3 initially planned features. Main application calls command line component internally, however latter can be used as a standalone console application.

Graph generating command line tool is written in *Python* [2]. Python is a multipurpose high-level programming language. Main reason behind choosing Python for graph generating logic is the *NetworkX* [3] library. This library, apart from offering a full-fledged functionality for working with graphs, has a graph generating module supporting more than 50 common graph types and more than 15 different algorithms for generating random graphs [4].

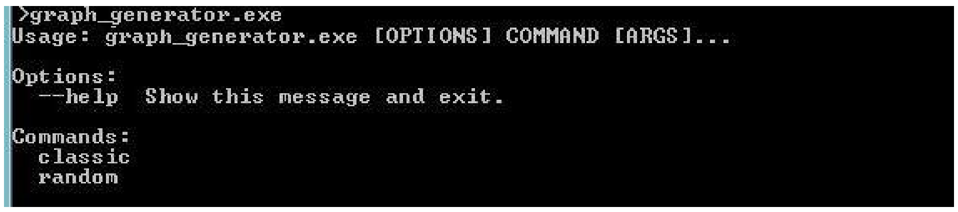
NetworkX library also has a graph export feature. Exporting is done in *GraphML* format [5] which is an XML based text file format for graph representation. This format files are used for passing graphs generated in command line tool to the main application. Since GraphML is a common file format serialization and de-serialization of graphs in this format is made easy with the existing parsing libraries.

Main application is a Windows application with graphical user interface written in *C#* [6] and *Windows Presentation Foundation engine (WPF)* [7]. WPF uses *XAML* [8] interface markup language which allows to easily create and handle custom graphical components. That is a very important criteria for this kind of applications where components such as graph vertexes and edges are required for drawing and manipulating in a user responsive manner.

2.2 Graph Generator Tool

Command line tool is a graph generating application with easy to use command line interface. Application has 2 commands: ″random″ and ″classic″. As mentioned in section 2.1 NetworkX libary is used for generating graphs and exporting them in GraphML format.

Figure 7: Graph generator help output



```
>graph_generator.exe
Usage: graph_generator.exe [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  classic
  random
```

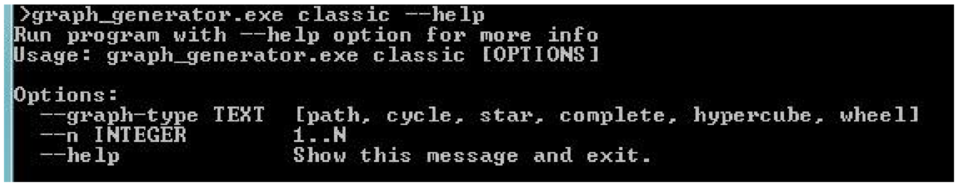
″Random″ command takes as an argument number of vertexes and generates a random graph. Random graph is generated using *Erdõs-Rényi model* [9], where graphs is generated with predetermined probability for edge existence. Our program uses 50% probability by default, this can be changed.

″Classic″ command takes two arguments, the number of vertexes and the type of classical graph. Currently we support path, cycle, star, complete, hypercube and wheel graphs. Code for generating a graph of specific type and exporting it is literally 2 lines of code, so it is only matter of time to add all graph types supported by NetworkX library.

For example, below is a sample code for generating 20 vertex complete graph with NetworkX library.

- graph = nx.complete_graph(20)
- nx.write_graphml(graph, './complete_20.graphml')

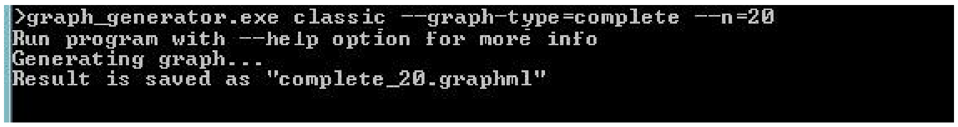
Figure 8: Graph generator help output for "classic" command



```
>graph_generator.exe classic --help
Run program with --help option for more info
Usage: graph_generator.exe classic [OPTIONS]

Options:
  --graph-type TEXT  [path, cycle, star, complete, hypercube, wheel]
  --n INTEGER        1..N
  --help             Show this message and exit.
```

Figure 9: Graph generator "classic" command usage example



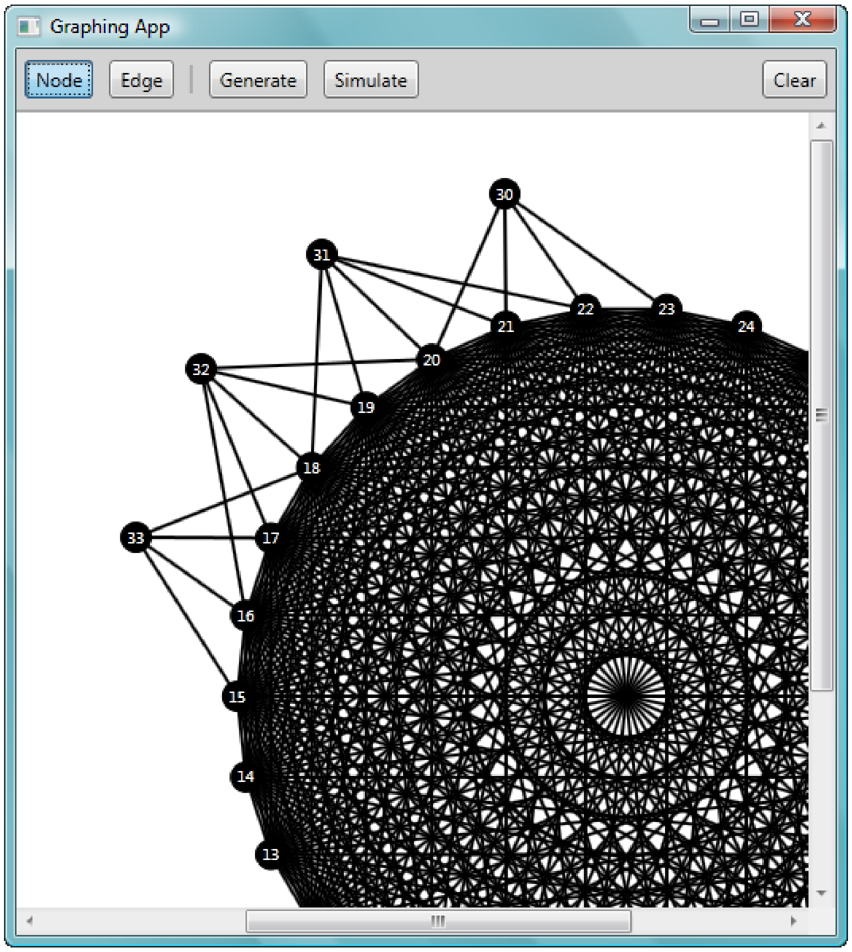
```
>graph_generator.exe classic --graph-type=complete --n=20
Run program with --help option for more info
Generating graph...
Result is saved as "complete_20.graphml"
```

Command line program also uses *Click_* [10] library for designing command line interface in a simple annotated way, that gives flexibility to add more commands or arguments for existing commands. And *PyInstaller* [11] application was used to compile Python code to an executable file to make the command line standalone application and call it as a process from the main application.

2.3 Main Application

Main application has a white canvas for drawing generated graphs and manual drawing; and a control bar.

Figure 10: Main windows screen



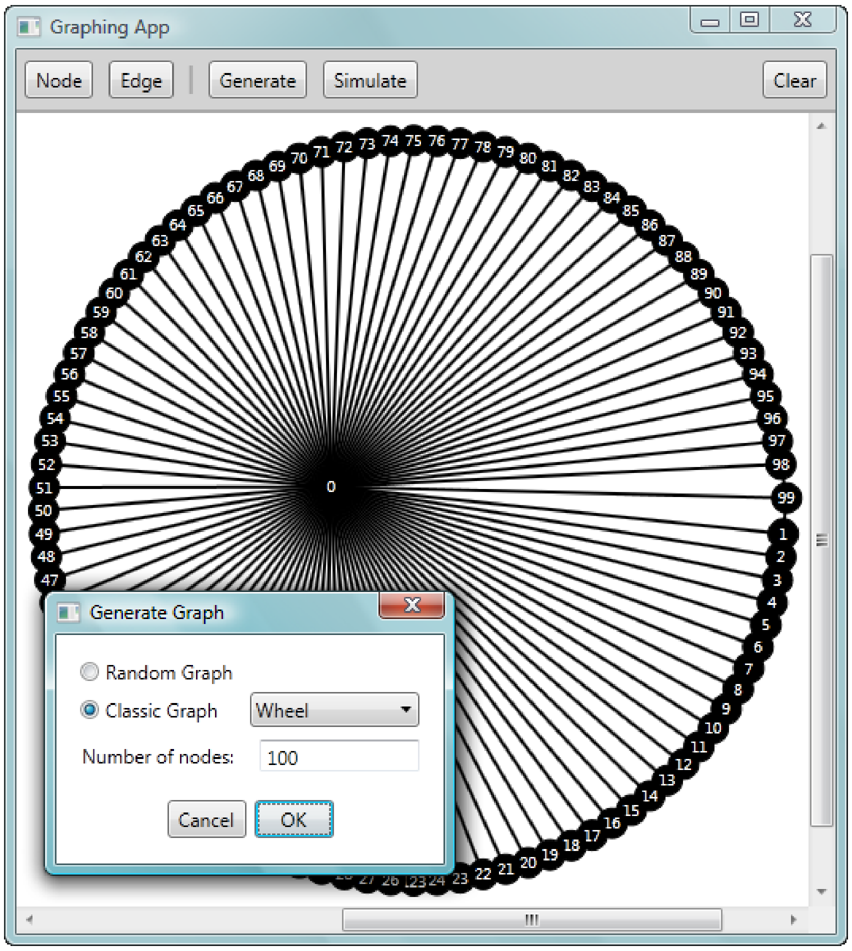
All main features can be accessed with the buttons on the control bar. Toggle buttons on the left corner are manual drawing mode buttons. Two modes are available: node and edge. Both modes can't be active at the same time. When node mode is active mouse left-click on the canvas will create a vertex with sequential id as a label. Id is reset to 0 when all vertexes are deleted. Mouse right-click on the vertex will delete it with all the adjacent edges.

In the edge mode left-click on the vertex will choose it as a first vertex and change its color to red. After clicking on the second vertex an edge will be drawn connecting those two vertexes. Clicking on the same vertex twice will cancel vertex selection. Mouse right-click works as a delete for edges as well. Mouse left-click and move action on a vertex will re-position it with all the adjacent edges. Delete and re-positioning will work regardless if any of the modes is active.

Generate button opens windows with generate options. After choosing graph type and number of nodes and pressing "OK" button, main application calls command line graph generator application described in chapter 2.2 and gets a GraphML file as a result.

GraphML file is parsed using QuickGraph [12] library's de-serializer and resulting graph is drawn on the canvas. NetworkX library only describes the structure of the graph and does not determine the positions of vertexes, so the coordinates are calculated manually using a simple polygon representation method. 12 is the formula used for calculating the coordinates.

Figure 11: Generate options window screen



Random Graph
Classic Graph: Wheel
Number of nodes: 100

Figure 12: Vertex coordinate calculation formula

$$x = centreX + r * \cos(2 * \pi * i/n)$$
$$y = centreY + r * \sin(2 * \pi * i/n)$$

$$i = 1 \ldots n, n = |V|$$
$$r = \min(CanvasWidth, CanvasHeight)/3$$
$$centreX = CanvasWidth/2$$
$$centreY = CanvasHeight/2$$

Graph on the canvas is stored in form of a linked list. Each vertex has an entry with a list of all edges connected to it. Edges in this storage structure are referenced twice to enable deletion of specific vertex with its edged. However in C# stored edges are actually references to real objects, so in case of extra data kept in the edges there will not be any performance issues. ″Clear″ button on the right corner of the control bar will clear the canvas and release graph related data.

The simulator feature is designed for accumulation problem investigation specifically. After clicking on ″simulate″ button a new window is opened and all buttons on control tab and right-click action become disabled to avoid graph alterations during simulation. The opened window contains a terminal for writing simulation script and control tab with 3 buttons: ″start″, ″next″ and ″stop″. Initially only ″start″ button is enabled.

Script is the code determining the steps and what should be done during simulation. Each line of the code should start with the following string *″time:″*, *time* is a time-slot sequential number starting from 0. Server vertex should be determined at time 0 with the following line *″0: server <- vertex_id″*, *vertex_id* is the designated vertex numeric id. All subsequent lines define how data should be passed. Format is following:

number: source_vertex_id ->target_vertex_id, source_vertex_id ->target_vertex_id...

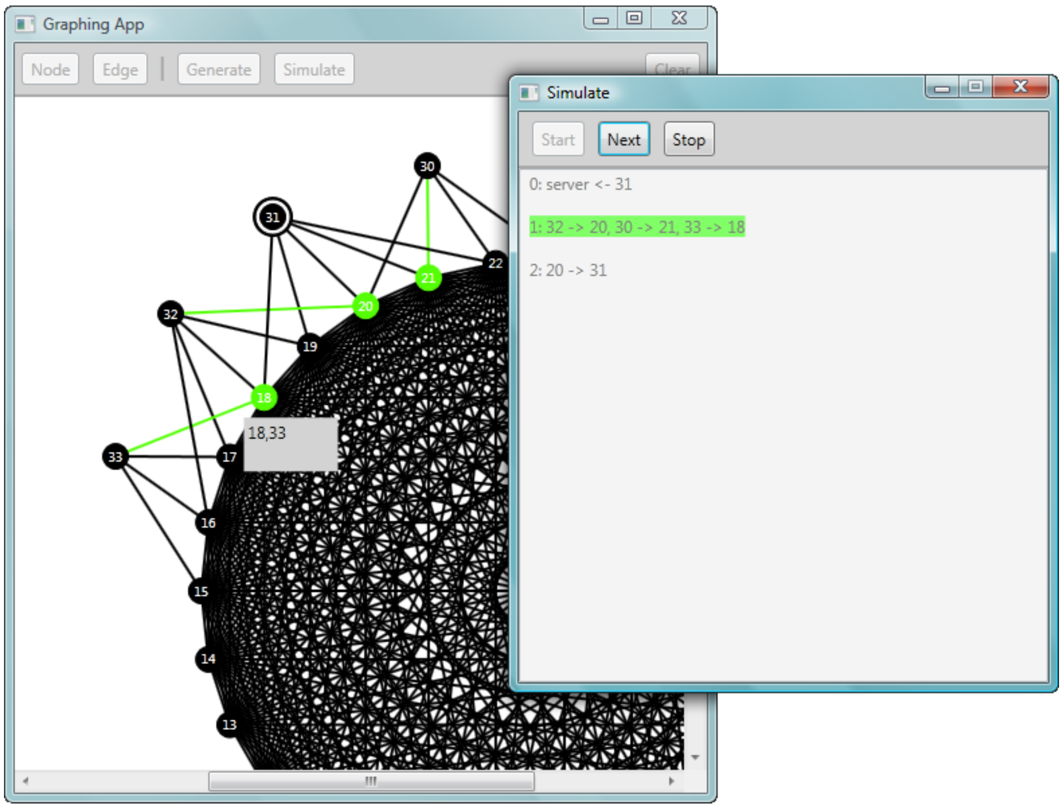
source_vertex_id is the id of the vertex that is passing a piece of message, *target_vertex_id* is the id of the vertex that will receive it.

After writing script and clicking on the ″start″ button simulation starts and ″next″, ″stop″ buttons become enabled. Immediately after clicking ″start″ button the first line of the script is highlighted. When script line does not have any errors it is highlighted with green color indicating it is the currently executed line of code and user can proceed to the execution of the next line with ″next″ button. After proceeding the next line becomes highlighted. Highlight color will be red when script line does not have a correct formatting or the vertex ids are not existing vertex ids, in this case ”next” button will be disabled. Passing data out of empty vertex and passing data from server vertex are also error cases.

During the execution of the code’s first line, server vertex is visualized on the graph. It gets an extra hollow like circle around main circle. All other script lines are for accumulation data passing visualization. Edges that are used to pass the message and the target vertexes will be colored green during those steps. Hovering over a vertex will show what data it contains.

”stop” button is for stopping simulation. It is the only option when error is occurred in the script, can also be used for terminating valid code execution. After terminating simulation graph visualization state is also reset.

Figure 13: Simulate window screen



```
0: server <- 31

1: 32 -> 20, 30 -> 21, 33 -> 18

2: 20 -> 31
```

3 Future Work

Accumulation is an interesting problem which is not thoroughly investigated for limited options cases. We plan to investigate accumulation problem on more graph typologies and try to determine a general algorithm that will at least complete the task even if not in the best possible time. And there are multiple features that can be added to the application to make it easier to use and more useful, some of those are

- Real time syntax warnings in the simulator feature to turn it into a real compiler
- Size option to the graph generate feature to allow users generate graphs with large amount of vertexes on the canvas, not limited to the screen size
- Developing a GraphML de-serialization module ourselves to support all graph types generated by NetworkX library and replacing currently used de-serializer
- Implementing more graph layout algorithms
- Exporting graphs in GraphML and .png image formats

References

[1] J. Hromkovic; R. Klasing; A. Pelc; P. Ruzicka; W. Unger. *Dissemination of Information in Communication Networks: Broadcasting, Gossiping, Leader Election, and Fault-Tolerance*. Springer, 2005.

[2] Python language. https://www.python.org/. [Online; accessed 09-05-2016].

[3] Networkx library. https://networkx.github.io/. [Online; accessed 09-05-2016].

[4] Networkx graph generators. http://networkx.readthedocs.org/en/networkx-1.10/reference/generators.html. [Online; accessed 09-05-2016].

[5] Graph Drawing Steering Committee. Graphml file format. http://graphml.graphdrawing.org. [Online; accessed 09-05-2016].

[6] Microsoft. C sharp language. https://msdn.microsoft.com/en-us/library/kx37x362.aspx. [Online; accessed 09-05-2016].

[7] Microsoft. Windows presentation foundation engine. https://msdn.microsoft.com/en-us/library/mt149842%28v=vs.110%29.aspx. [Online; accessed 09-05-2016].

[8] Microsoft. Xaml - extensible application markup language. https://msdn.microsoft.com/en-us/library/cc189036%28VS.95%29.aspx. [Online; accessed 09-05-2016].

[9] P. Erdos; A. Renyi. On random graphs. i. *Publicationes Mathematicae*, 6, 1959.

[10] Click library. http://click.pocoo.org/5/. [Online; accessed 09-05-2016].

[11] Pyinstaller packager application. http://pyinstaller.readthedocs.io/en/latest/index.html. [Online; accessed 09-05-2016].

[12] Quickgraph library. https://quickgraph.codeplex.com/. [Online; accessed 09-05-2016].