Chapter 3

The Distributed Graph Algorithm Playground

3.1 Tutorial

Algorithms are designed with the Erlang programming language, configurations are established through Javascript Object Notation (JSON) files, and interactions are facilitated via the VS Code Command Palette and VS Code Webviews. Hence, users are expected to have a fundamental understanding of Erlang, JSON and VS Code.

3.1.1 Getting Started

Installation

To use the tool, the VS Code extension must be installed by downloading the released vsix file from the Github repository and install it manually into VS Code.

For users, to install a `.vsix` file in VS Code:

1. Go to the Extensions view.
2. Click **Views and More Actions...**
3. Select **Install from VSIX...**

or

in your terminal, run the following command:

```
# if you use VS Code
code --install-extension my-extension-0.0.1.vsix

# if you use VS Code Insiders
code-insiders --install-extension my-extension-0.0.1.vsix
```

Setup

After installing the extension, the next step involves setting up the development environment. The essential setup for the tool should include a dgap.json file. The dgap.json file must be located at the root of the workspace/folder. Such that the current setup will resemble the following structure.

```
/ (root)
```

└── dgap.json

This is very important as the extension will look at the root for that particular name.

3.1.2 Interface

Now that the essentials are in place it is now appropriate to look at how to interact with the tool. The first place to get familiar with is the VS Code Commands. These will be the entry point for all further interactions with the tool. These commands utilize the VS Code window API to relay important information and errors to the user.

COMMANDS

- **DGAP: Start** - Starts the DGAP Erlang OTP as a child process of VS Code. An information message is sent when the process is ready.
- **DGAP: Stop** - Stops the Erlang application. An information message is sent when the process is stopped. This operation discards all previous progress of the Erlang application.
- **DGAP: Compile** - Compiles a .erl file into an algorithm designed for use in a simulation.
- **DGAP: Simulate** - Simulate an algorithm on a specified topology and generate a webview to visualize the graph on which the algorithm was simulated.
- **DGAP: Observer** - Starts the Erlang observer module and connects it to the tool.

This was a high-level summary outlining the functions of various commands. The attention will now shift to providing a comprehensive explanation of the "compile" and "simulate" commands as these are the two configurable commands essential for preparing the first example.

Compile

For configuring the compile command, the dgap.json file should have a compile object. Reading which file to compile and load into an algorithm.

compile object

```
"compile": {
    "file": "string" // path to file
}
```

Simulate

For configuring the simulate command, The dgap.json file should have a simulate object. Reading which algorithm to run on the given topology.

simulate object

```
"simulate": {
    "module": "string", // algorithm module
    "fun": "string", // algorithm start function
    "topology": "object" // topology object
}
```

topology object

```
"topology": {
    "type": "string", // "ring" | "complete" | "random"
    "size": "number", // number of vertices
    "alpha": "number" // decimal between -1 and 1
}
```

The "alpha" property in the topology object is only required when specifying a topology object of type "random". The "alpha" property indicates how many edges the random graph is expected to have. Ranging from -1 is a null graph and 1 is a complete graph.

3.1.3 Basic Functionality

With the knowledge of how to configure commands, it is now time to look at examples on how to use the basic functionality of the tool. The current setup will be extended further for each example shown, to ensure there is a smooth transition when following the tutorial from one step to the next.

Compile a File

To compile a file there must exist a file to compile. Therefore a file is created called hello_world.erl, such that the current setup will resemble the new structure.

```
/ (root)
├── dgap.json
└── hello_world.erl
```

Prior to implementing any algorithm within the hello_world.erl file, it can be preconfigured for compilation upon necessity. A compile object is written inside the dgap.json file to compile the hello_world.erl file as required.

Listing 3.1: dgap.json

```
{
    "compile": {
        "file": "hello_world.erl"
    }
}
```

The hello_world.erl file must be implemented as an Erlang module; otherwise, attempting to compile it will generate an error. In this example, the hello_world.erl file will contain a simple implementation.

Listing 3.2: hello_world.erl

```
-module(hello_world).

-export([start/1]).

start(_Args) ->
  "Hello World!".
```

Listing 3.1 and 3.2 detail the process of creating and configuring a module, which, when initiated by running **DGAP: Start** and subsequently followed by **DGAP: Compile** will yield a fully launched Erlang application. This application will have the hello_world module compiled and loaded, making it ready to be simulated on a graph.

Simulate a Graph Algorithm

For the simulation of a graph algorithm, a compiled module with an exported function must be available. The exported function may only have 1 argument with the argument type defined as:

```
-type vertex() :: {Id :: integer(), Pid :: pid()}.
-type argument() :: {Vertex :: vertex(),
                     Edges :: [Neighbors :: vertex()]}.
```

This all is complied with in the previous example, which means by configuring a simulate object inside the dgap.json file. It is then possible to run a graph simulation using the tool.

Listing 3.3: dgap.json

```
{
    "compile": {
        "file": "hello_world.erl"
    },
    "simulate": {
        "module": "hello_world",
        "fun": "start",
        "topology": {
            "type": "ring",
            "size": 10
        }
    }
}
```

With the extended configuration of the dgap.json file shown in listing 3.3. Running the **DGAP: Simulate** command will generate a new graph simulation featuring a ring topology of size 10, where each vertex within the graph will initiate the start function.

Figure 3.1 demonstrates the execution of the setup designed using Listings 3.2 and 3.3. In Figure 3.3, the graph is visualized within a webview, displaying a ring topology of size 10 as outlined in Listing 3.3. The output on the right side of the webview showcases the printing of the string 'Hello World!' from each vertex specified in Listing 3.2, visible in the result list. The numbers associated with the visualized graph and result list correspond to the unique vertex IDs.

3.1.4 Advanced Features

This section will delve into more specialized functionalities on how to verify and debug the algorithms implemented to ensure the algorithm is executing as intended.

Data

Analyzing the data exchanged within a distributed graph algorithm is crucial for assessing its performance. The tool incorporates three types of data: messages, results, and logs. These can be read from the panel at the right side of the webview.

- **Messages** - Shows all messages sent between vertices in the graph.
- **Result** - Shows all results returned from vertices in the graph.
- **Log** - Shows a new log, every time the log bif is executed. The log bif takes one argument and it can be of any type.

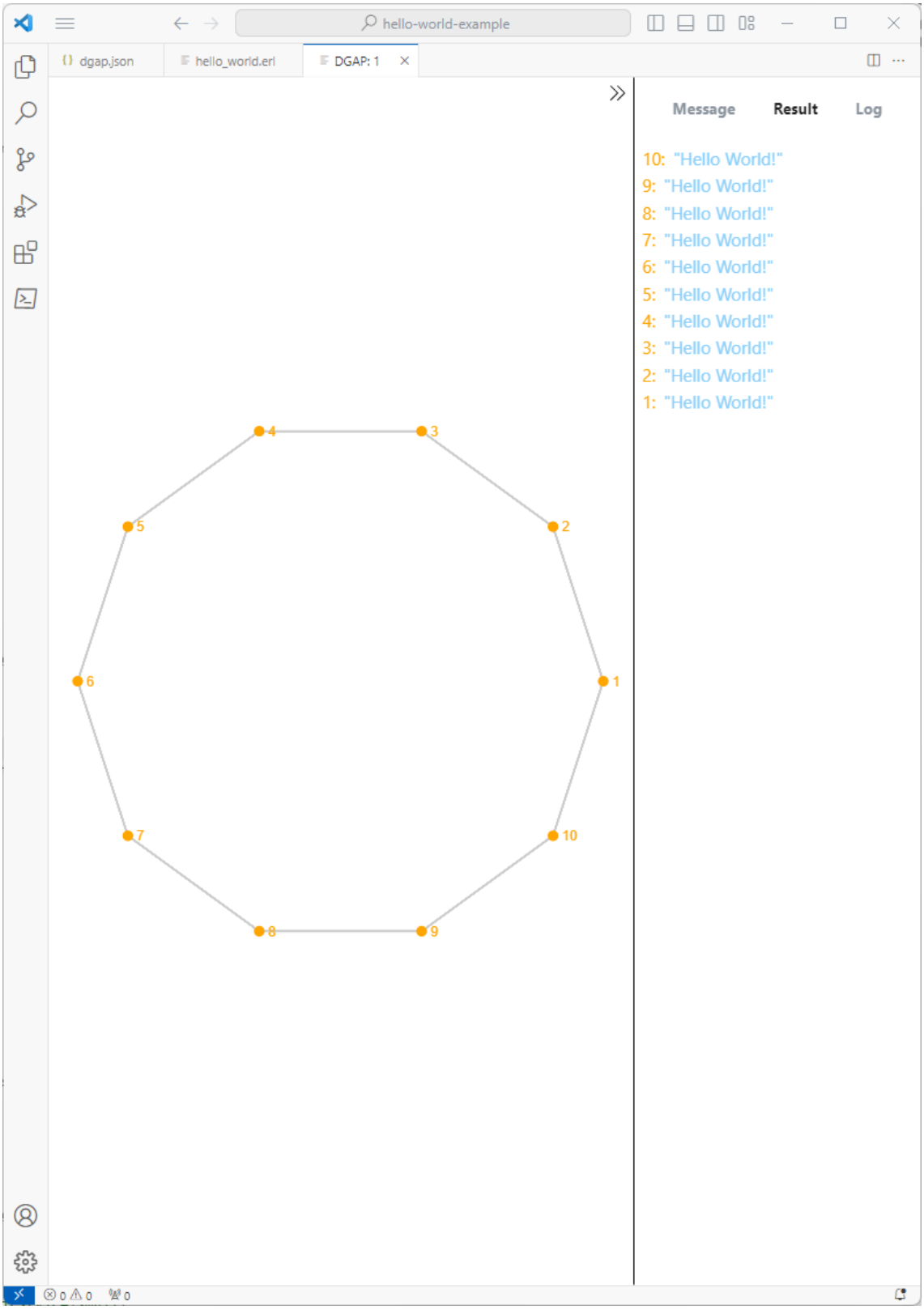


Figure 3.1: Hello World Graph Simulation

Listing 3.4: hello_world.erl

```
1  -module(hello_world).
2
3  -export([start/1]).
4
5  start({_Self, [{_NeighborId, NeighborPid} | _Rest]}) ->
6    NeighborPid ! "Hello World Message",
7    log("Hello World Log"),
8    "Hello World Result".
```

Listing 3.4 extendes the prior hello_world.erl example by incorporating all varieties of data streams for debugging. Figure 3.2 demonstrates the utilization of hello_world.erl from Listing 3.4, simulating a scenario akin to that depicted in Figure 3.1.

Link Failure

Communication links in distributed systems spanning geographical distances often suffer from unreliability. This can be simulated using this tool by clicking on the graph edges. A graph edge with a light color indicates an open communication link between two vertices, while an edge colored red indicates a closed communication link between two vertices, preventing messages from reaching their destination through that particular link.

Figure 3.3 shows a Link Failure at edge {1, 3} and {2, 4}. In the scenario illustrated in Figure 3.3, this implies that all messages sent from vertex 1 to vertex 3 and from vertex 2 to vertex 4 are disregarded by the system.

This means it is possible to test an algorithm in a real world scenario were two vertices suddenly are unable to communicate.

Observer

The observer module serves as a tool to analyze the processes of the Erlang application. However, it's important to note that the Observer module isn't included with the ERTS from the VS Code extension. Therefore, in order to utilize this functionality, Erlang must be installed on the system running the VS Code application. Once the Erlang process is running and ready, invoking **DGAP: Observer** will initiate the observer module using the globally installed Erlang Runtime, establishing a connection to the tool.

3.1.5 Use Cases

In this section three distinct algorithms will be explored. The algorithms will increase sequentially in difficulty. Each will explain its use case, providing an implementation and demonstrating how to interpret the visualization tools to ensure a correct implementation.

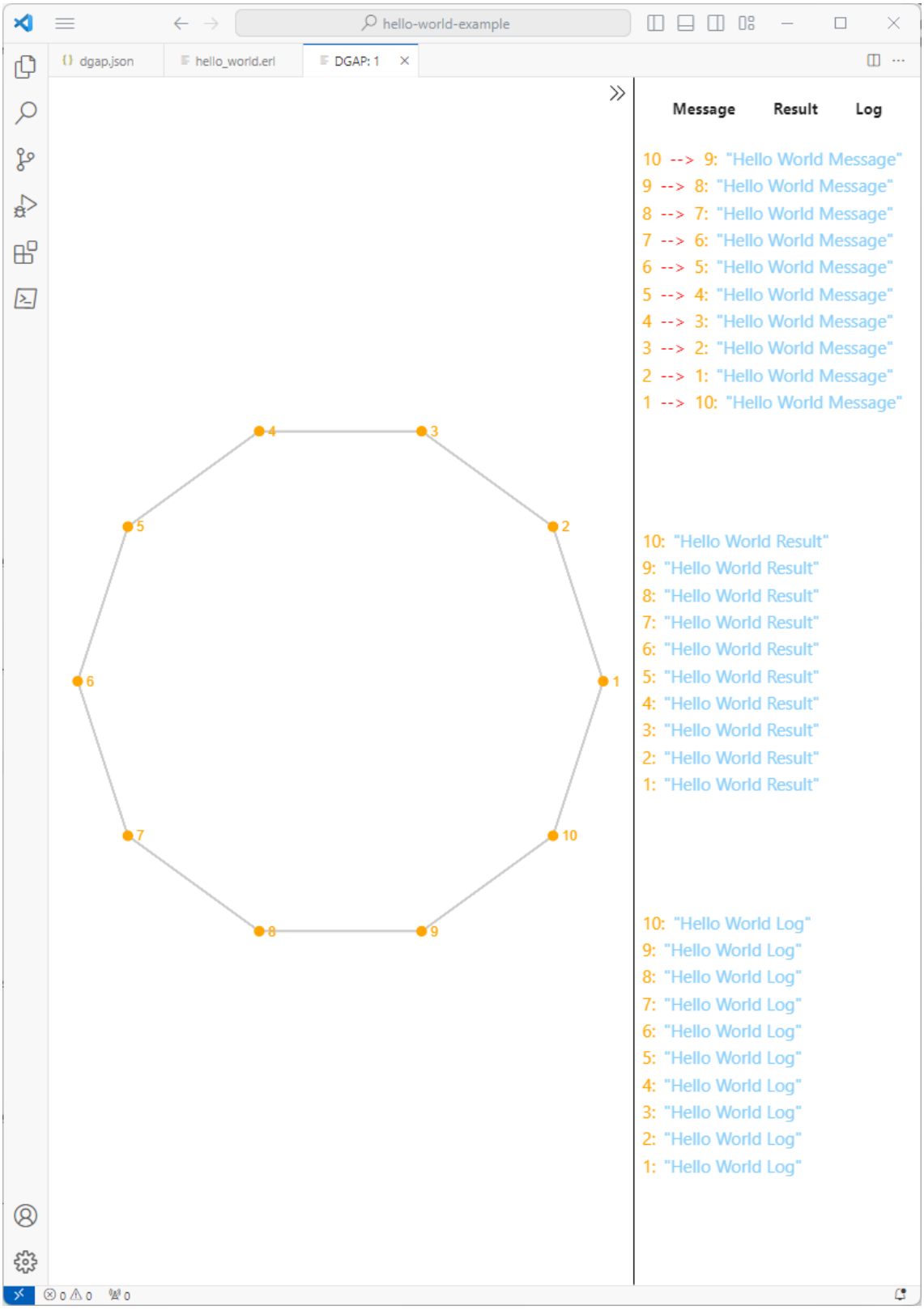


Figure 3.2: Hello World Graph Simulation Debug

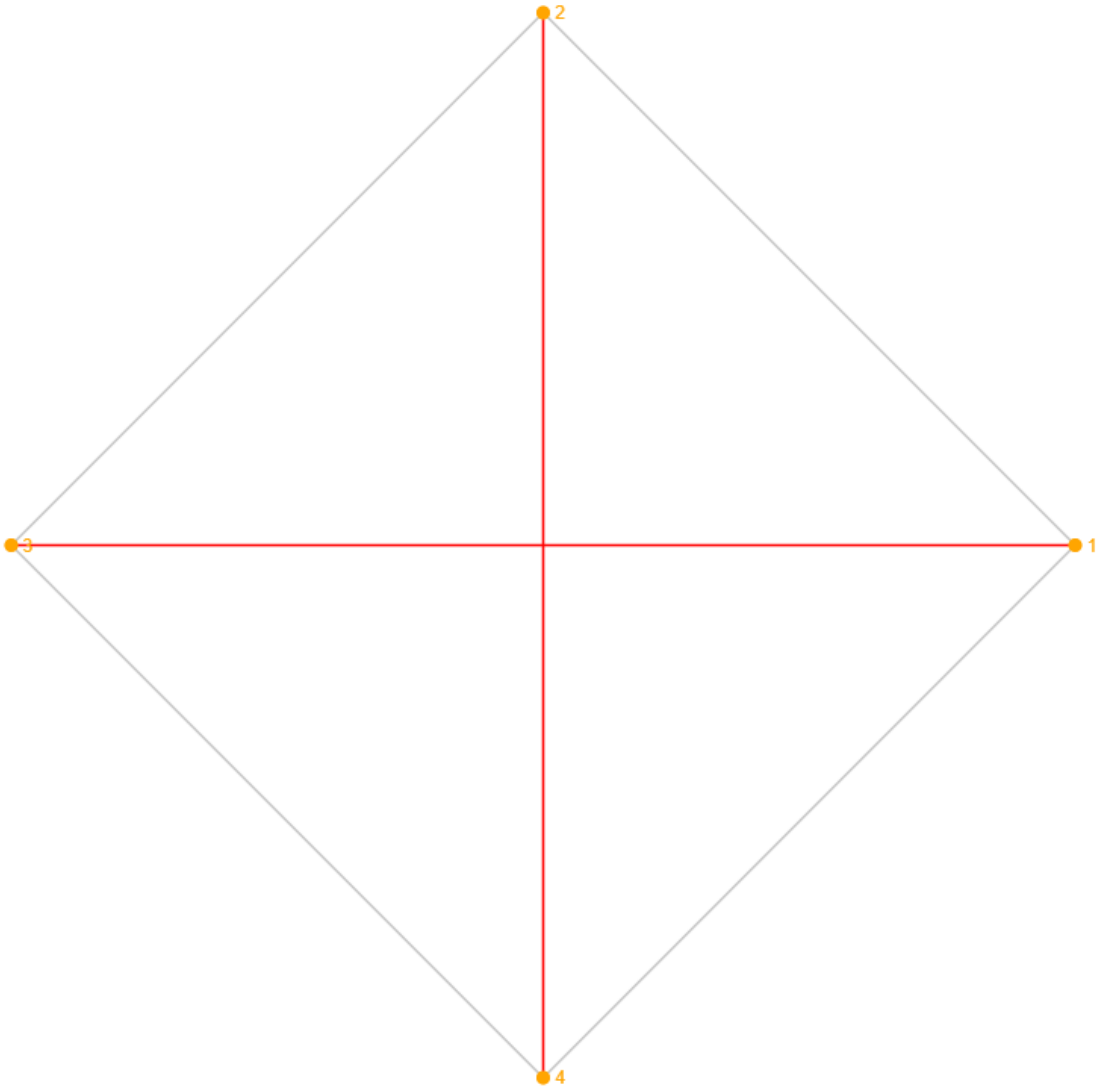


Figure 3.3: Link Failure

Ping Pong

The Ping Pong algorithm serves as an excellent starting point due to its simplicity. In this approach, one process initiates a ping request to another process and anticipates a pong response. This method is frequently employed at regular intervals to verify the continued health of processes.

Listing 3.5: ping_pong.erl

```
-module(ping_pong).

-export([start/1]).

start({{Id1, _Pid1}, [{Id2, Pid2}]}) when Id1 < Id2 ->
  ping(Pid2);
start({_Self, [{_Id1, Pid1}]}) ->
  pong(Pid1).

ping(Pid) ->
  ping(Pid, 3).

ping(Pid, Try) ->
  timer:sleep(1000),
  Pid ! ping,
  receive
    pong ->
      ping(Pid, 3)
  after
    1000 ->
      case Try of
        0 ->
          "Failed to receive pong in 3 attempts!";
        _ ->
          ping(Pid, Try - 1)
      end
  end.

pong(Pid) ->
  receive
    ping ->
      Pid ! pong,
      pong(Pid)
  after
    6000 ->
      "Failed to receive ping!"
  end.
```

Listing 3.5 presents an algorithm for a ping-pong interaction. The algorithm is designed to operate on a linked graph of size two. Within this implementation, the process with the smallest ID initiates a ping to the other process every second. In the event of no response after 1 second, the algorithm retries the ping procedure. The ping process is programmed to terminate if it does not receive a pong response after three attempts. Similarly, the pong process is set to terminate if it has not received a ping after 6 seconds.

In Figure 3.4, the visualization depicts the ping pong algorithm, illustrating that each time process 1 sends a ping to process 2, an immediate response in the form of a pong follows. In Figure 3.4 (c), a scenario is presented where the link between the two processes is removed for a duration exceeding 6 seconds, resulting in both processes terminates.

(a) Graph Topology

2 --> 1: pong
1 --> 2: ping

(b) Communication

1: "Failed to receive pong in 3 attempts!"
2: "Failed to receive ping!"

(c) Termination

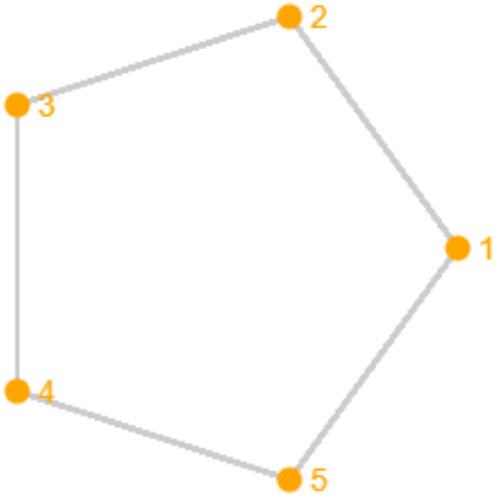
Figure 3.4: Ping Pong Visualization

Leader Election

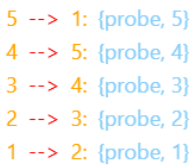
The subsequent algorithm Leader Election introduces a higher level of complexity, as every process executes identical code, and various processes can exist in distinct states. Leader election necessitates a consensus among all processes to designate a specific distinguished process, referred to as the leader. The presence of a leader is crucial in numerous distributed systems due to the inherent asymmetry in algorithms; a decisive initiator is required.

Listing 3.6: leader_election.erl

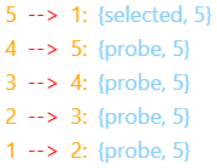
```
1  -module(leader_election).
2
3  -export([start/1]).
4
5  start({{Id, _Pid}, [{_LeftId, _LeftPid}, {_RightId, RightPid}]}) ->
6    RightPid ! {probe, Id},
7    loop(Id, RightPid, none).
8
9  loop(Id, RightPid, Leader) ->
10   receive
11     {probe, ProbeId} ->
12       if
13         Id > ProbeId ->
14           loop(Id, RightPid, none);
15         Id < ProbeId ->
16           RightPid ! {probe, ProbeId},
17           loop(Id, RightPid, none);
18         true ->
19           RightPid ! {selected, Id},
20           loop(Id, RightPid, none)
21       end;
22     {selected, SelectedId} ->
23       case SelectedId =:= Id of
24         true ->
25           RightPid ! agreement,
26           "I am the leader!";
27         false ->
28           RightPid ! {selected, SelectedId},
29           loop(Id, RightPid, SelectedId)
30       end;
31     agreement ->
32       RightPid ! agreement,
33       lists:flatten(io_lib:format("The leader is ~p!", [Leader]))
34   end.
```



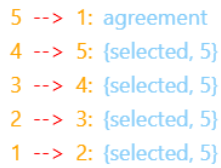
(a) Graph Topology



(b) 1st Round

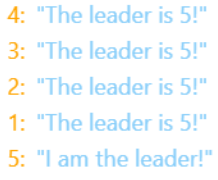


(c) 2nd Round



(d) 3rd Round

(e) 4th Round



(f) Termination

Figure 3.5: Leader Election Visualization

Listing 3.6 presents an algorithm for the Leader Election. The algorithm is designed to operate on a graph with a ring topology. The implementation has been extended to include a mechanism where, upon termination of the process associated with the selected ID, it communicates to its right neighbor the successful establishment of an agreement. This way every process terminates and returns their selected leader.

In Figure 3.5, the visualization depicts the Leader Election algorithm. From Figure 3.5 (b) to (e), each round demonstrates how the algorithm determines a leader. Because all processes has a more suitable leader to its right, except the correct leader. The algorithm efficiently terminates in minimum steps.

Luby's Maximal Independent Set

The last algorithm to explore is the Luby's Maximal Independent Set (MIS) algorithm. For a graph of vertices, an independent set, is a set of vertices where no links are shared between them. An independent set is a MIS if no strict superset of it is an independent set. The MIS problem arises in scenarios such as wireless broadcasting, where the imperative is to ensure that adjacent nodes do not simultaneously engage. This constraint is particularly crucial in situations where transmitters are prohibited from broadcasting on the same frequency within their respective ranges.

Listing 3.7: luby_mis.erl

```
1  -module(luby_mis).
2
3  -export([start/1]).
4
5  -record(state, {
6    random,
7    selected,
8    eliminated
9  }).
10
11 start({Self, Neighbors}) ->
12   loop(Self, Neighbors, #state{ random = rand:uniform() }).
13
14 loop(_Self, _Neighbors, #state{ selected = true }) ->
15   "Selected";
16 loop(_Self, _Neighbors, #state{ eliminated = true }) ->
17   "Eliminated";
18 loop(Self, [], State) ->
19   loop(Self, [], State#state{ selected = true });
20 loop(Self = {Id, _Pid}, Neighbors, State = #state{ random = Random }) ->
21   NLength = length(Neighbors),
22   send_random(Neighbors, Random),
23   Randoms = await_randoms(NLength),
24   case lists:all(fun(NRandom) -> Random < NRandom end, Randoms) of
25     true ->
26       send_selected(Id, Neighbors, true),
27       loop(Self, Neighbors, State#state{ selected = true });
28     false ->
29       send_selected(Id, Neighbors, false),
30       Selections = await_selections(NLength),
31       case lists:any(fun({_NId, NSelected}) -> NSelected end, Selections)
             of
32         true ->
33           FalseSelections = lists:filter(fun({_NId, NSelected}) -> not
                 NSelected end, Selections),
34           FalseNeighbors = lists:map(fun({NId, _NSelected}) -> lists:
                 keyfind(NId, 1, Neighbors) end, FalseSelections),
35           send_eliminated(Id, FalseNeighbors, true),
36           loop(Self, Neighbors, State#state{ eliminated = true });
37         false ->
38           send_eliminated(Id, Neighbors, false),
39           Eliminations = await_eliminations(NLength),
40           TrueEliminations = lists:filter(fun({_NId, NEliminated}) ->
                 NEliminated end, Eliminations),
41           UpdateNeighbors = lists:foldl(fun({NId, _NEliminated}, Acc) ->
                 lists:keydelete(NId, 1, Acc) end, Neighbors,
                 TrueEliminations),
42           loop(Self, UpdateNeighbors, State)
43       end
44 end.
```

Listing 3.7 presents an algorithm for Luby's MIS. An implementation of the private functions can be found in the appendix. [TODO]

In figure 3.6 a visualization is shown of a random graph which used Luby's algorithm, to select a MIS. The MIS selected for the graph is 1, 2 and 10. Which means the set {1,2,10} covers all the vertices of the graph.

proof:
Let

$$links(ID) = set\ of\ links\ for\ vertex\ ID$$

Then

$$links(1) = \{4, 6, 8\},\ links(2) = \{3, 4, 5, 6, 9\},\ links(10) = \{4, 5, 7, 9\}$$

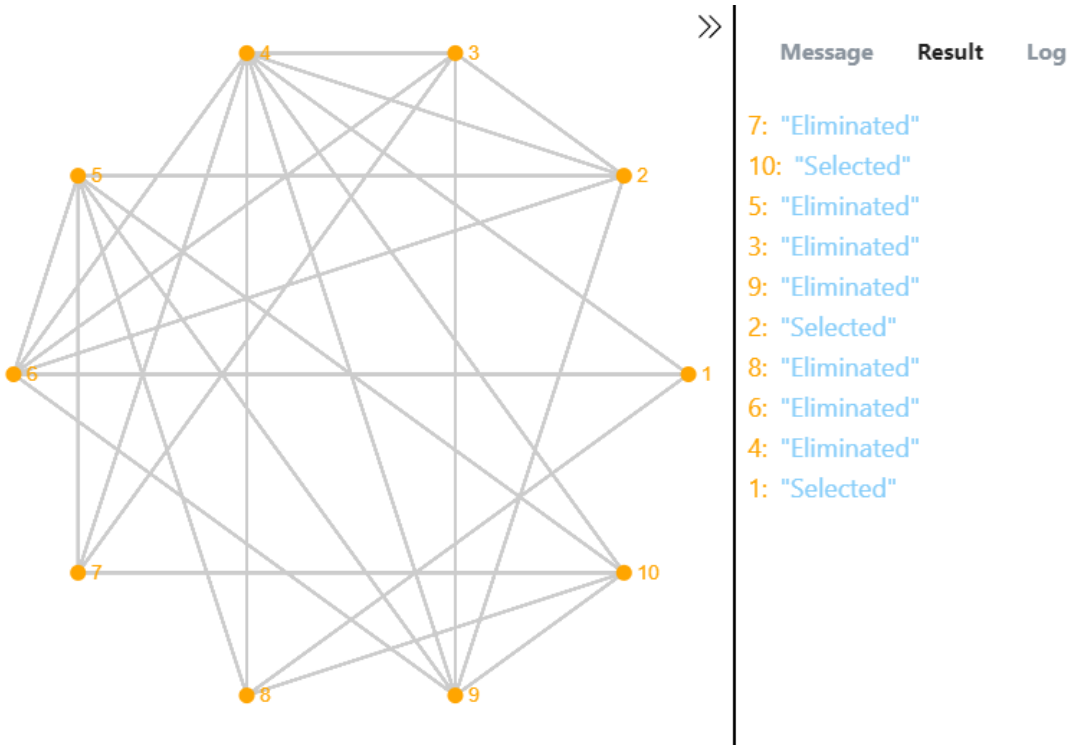


Figure 3.6: Luby's Maximal Independent Set Visualization

$$\{1, 2, 10\} \cup \{4, 6, 8\} \cup \{3, 4, 5, 6, 9\} \cup \{4, 5, 7, 9\}$$
$$= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$