Distributed Systems
Assignment 1
Maximum Independent Set
March 24

Jacob Nielsen
jacon18

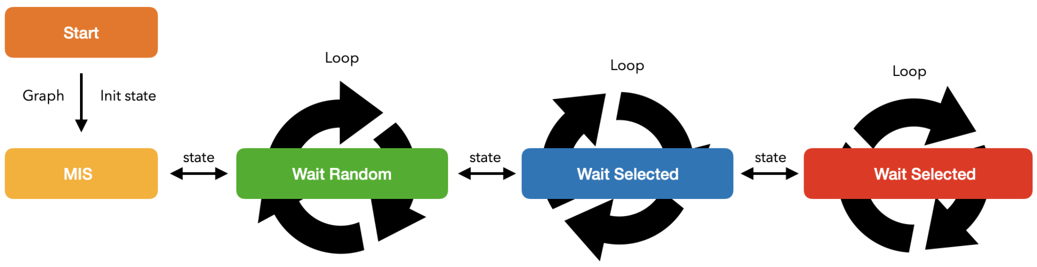
24. marts 2022

1 Introduction

This report designs and implements Luby's Algorithm for Maximal Independent Set (Algorithm 5.12, Section 5.6 Red book). The implementation is designed on basis of the provided pseudocode and is expressed en Erlang. This is to gain the benefits from functional languages and to get an easy process-simulation thanks to Erlang's VM, that makes it very easy to spawn reliable connections to in-systems processes.
The Algorithm implemented finds the maximal independent set between all participating nodes.

2 Design/Implementation

The algorithm is designed and implemented according to very standard functional language structures. What is a design decision made by implementor is, is not just to recursive calls until termination, adding records to the stack. The sub-routines, the loops waiting for messages from neighbors, are returning their state into the function whole called them, this state is then put on the program-stack, within the callers record. It saves a little memory, but more important it makes the code a lot more readable and makes it easier to understand with an imperative setting in mind. An abstract overview how this is achieved in the internal functions, and how the state is then parse parsed around, in the unfolding function is shown in figure 1. The loops can output the result, as this ia part of the looping conditions, therefore these should be seen as mechanisms working and not a direct flow-diagram.



Figur 1: Algorithm Overview

Overall the design the provided pseudocode

2.1 The Process State

The process state keeping track of every single process state is seen below.

```
    -record(state, {
        vertex,
        random,
        neighbours,
        nodes,
        randoms
        }).
```

Vertex represent the given vertex identifier. In our case this is an integer due to the implementation of the `simulator_util` module, but it will work as long it is a comparable value. Random is a uniform chosen value between 1-1000 generated by a internal module. This should be reasonbly high in relation to the number of nodes in thee graph. This could be implemented to not be static. Neigbours is a list of vertex-ids representing the neighbours for the given node/process. Nodes a tuple-list of all the nodes in the graph, where we can extract Pid()'s from when the process needs to communicate with a specific vertex. Randoms is used to collect a list of other process' random value, which is used to reason on, as a whole.

2.2 `start()`

The start function is the algorithms initiator. This function takes no input, but outputs a simulation. In deployed environment or in a real test-environment this function should take the graph directly as argument. For now, this is implemented to make the graph-construction within this function itself, using the provided `simulator_util` module. `start()` is the only exported function.
Next the simulation is started using `start/2` from the `simulator` module. The graph is parsed as argument and an anonymous is formulated, taking the given `Vertex` and `Neighbours` for each node in the graph (hence the process), and lastly all the `Nodes` is parsed into the body.
The function has random sleep capability, that is commented out currently as it has no functionality other than simulating/ randomizing the message-parses
Within the body the initial state is made for the running process and the function `mis` is called. `mis` is abbreviation for Maximal Independent Set, from here just mentioned as `mis`.

2.3 `mis() - Maximal Independent Set`

`mis` is the heart of the algorithm coordinating the further calls and the running invariant. The function is parsed the state initiated from start. From here snippet 1.b is evaluated, to see if the process has any neighbors (potentially left). If this is not the case, the function outputs, that there is only one vertex present, with its identifier. On the other hand, if this is not the case. The pseudo-code dictates that each process must send its random number to all other processes (these values is generated using the build-in function `rand:uniform`). This is done by iterating over the neigbours list, consulting the tuple-list nodes getting the `Pid()` for the given neighbors, before parsing the message the given process.
snippet 1.f dictates, that we must wait for all neighbors to response, to test the condition in snippet 1.g. Therefore is `get_random_loop()` introduced, to mimick the loop-mechanism with recursion.

2.4 `get_random_loop()`

The functions mimicks the loop for waiting all neighbors to response with their random number. The function is called with a counter, a atom-value called `Count`. If count is smaller than the number of neighbors, we have not received a response for each neighbor yet. Please be aware, that the implementation assume a ideal network without, where we are guaranteed messages arrival. It is possible to accommodate a real scenario with the `receive`-constructs `after` operation. In this implementation, we use a basic recieve construct, that waits for a message of the form: { `random, Random_j ¶`} , that accumulate the `Random_j` atom in the states `randoms` list, before make a recursive call on the function again, with the Count argument increased by one.
When all neighbors has send a message, this is catched by the `true`-branch. This branch construct a predicate and uses the filters `:any` function, to tests if any of the received random values is smaller than the current process' own random value.
Next this is tested in a if/true construction. if true, then the process's random value is not the smallest among all. In this case the snippet 1.k is implemented and sending {`selected, false`}, to all neighbors, before going into a selected loop, awaiting the responses, as dictated in snippet 1.l. This is done in another loop function called `get_selected_loop`. Select loop returns a tuple with 3 values. The new state, the selected list, with all the neighbours sending af {`selected, false`} message, and last a value `J_true` which is true if a neighbour send a {`selected, true`} message (snippet 1.m). In this case this case we send an {`eliminated, true`}, to all neighbors, that send {`selected, false`}, to the given process. This eliminated the process/ vertex that can declare it is not in MIS.
If `J_true` is false, we send {`eliminated, false`}, to any neighbor in the states neighbor-list and call the function `get_eliminated_loop`. This function returns a new state that we recurse on `get_random_loop`, called with the Counter incremented by one.
on the other hand, if there was received no smaller random values from the neighbors, then the vertex can declare itself as part of the MIS-set. This is communicated to the other nodes with a {`selected, true`} message.

2.5 `get_selected_loop()`

Takes a state, a counter (initalizsed to 0), a selected list (initalized to []), and J_true, which is initalised to false. The function tests if we have recieved an answer from all neighbours, and returns a tuple of form `S, Selected_list, J_True` if true. otherwise the function goes into a `recieve` constructor, awaiting messages of form { `selected, _, true` }, in case the function will recurse calling itself, with the same arguments with the Counter increased by one or form { `selected, _, false` }. In the latter case the recived vertex is appended to the `Selected_list` list before recursing on the function itself.

2.6 `get_eliminated_loop`

Is called with the current state S and a Count. Returns/evaluates if all neighbors have answered or goes into a `recieve` constructor. When `eliminated, Vertex, true` is recieved, then the given vertex is removed from the list of neighbors in state. The function is called again with new parameters. If `eliminated, Vertex, false` is recieved then stage is just called again, with updated count value incremented.

3 Test

The test 1.a-1.c is done using a complete graph generated by `simulator_util:complete_graph` function provided.
The different tests are simply based of different graph topologies, with different number of vertices, they can be seen in the `start()`- function in the `asg1.erl` file.

3.1 1.a) 1 Vertex - Complete Graph

Only one vertex, hence the program must output only that single vertex, this is of course due to due condition, that the node is selected if it doesn't have any neighbours.

3.2 1.b) 2 Vertices - Complete Graph

There is only two vertices, the program must output one of them, but not both. It can be seen an test-example in appendix 6.3, where it can see the output is as expected. If we test runs multiple times the result will alternate between the two nodes, this is expected behavior in a distributed system and is dependent on which process send/recieve a given message first.

3.3 1.c) 5 Vertices - Complete Graph

In this situation there is multiple possibilities in what could represent a MIS. This is showcased in the appendix 6.4, where there is multiple cases, showcasing multiple Maximum Independent Sets. We expect only one node to be in the MIT-set because of the graph-completeness.

3.4 1.d) Undirected Ring Graph

Test of on undirected graph topology. This works as expected, refer 6.5, for output for designated MIS-set.

3.5 1.e) Ring Graph

Ring, all vertices, processes has two neighbors. This works as expected. See output in appendix 6.6.

4 Discussion

A Distributed Maximal independent set algorithm is implemented, and tested on various graph topologies, where it behave as expected referring to different tests and their individual results as stated in the section above. Regarding the complexity, the tests showed that only a single iteration of the loop was sufficient for these types og graphs. Algorithm states $O(logn)$, which is an upper bound stated in the research paper. What is important her, is that the pratical complexity has something to do with the inter-connection/ edges of the graph, where we in these test don't have any extreme cases.

5 Conclusion

All in all can we conclude that we have a working distributed algorithm for finding the maximum independent set in a given graph topology with different number of vertices. The algorithm works properly in the edge-cases where the graph consists of one or two vertices. The complexity could be tested a lot more thoroughly and on a lot bigger graphs, making it possible to reason a lot more about the theoretical complexity versus the one in experiments. Lastly, there could be added another dimension to this strategy, that is the one, where we don't rely on a perfect network, which would be meaningful in relation to time-complexity.

6 Appendix

6.1 readme.md

```
### Assignment 1 - Distributed Maximum Independent Set

## Contains
The folder contains the provided simulator_util.erl and simulator.erl files.
The algorithm is in the file asg1.erl. The different tests is in this file,
within the start function, that is used to call the MIS()- function itself.

## RUN
1. open erl shell
2. compile: c(asg1).
3. execute binary: asg1:start().
```

6.2 Test 1a

Test performed on a graph containing only one node.

```
1: neighbours: []
<0.123.0>
1: ready!
Only one vertex in graph: 1
1: terminated with result: ok
simulation stopped: no nodes running.
```

6.3 Test 1b

Test performed on a complete graph with two nodes.

```
8> asg1:start().
1: neighbours: [2]
2: neighbours: [1]
<0.114.0>
1: ready!
2: ready!
Vertex: 2 in MIS
Vertex: 1 NOT in MIS
2: terminated with result: ok
1: terminated with result: ok
simulation stopped: no nodes running.
```

6.4 Test 1c

6.4.1 case 1

```
19> asg1:start().
1: neighbours: [2,3,4,5]
2: neighbours: [1,3,4,5]
3: neighbours: [1,2,4,5]
4: neighbours: [1,2,3,5]
5: neighbours: [1,2,3,4]
<0.208.0>
1: ready!
2: ready!
3: ready!
4: ready!
5: ready!
Vertex: 5 in MIS
Vertex: 1 NOT in MIS
Vertex: 2 NOT in MIS
Vertex: 3 NOT in MIS
Vertex: 4 NOT in MIS
5: terminated with result: ok
1: terminated with result: ok
2: terminated with result: ok
3: terminated with result: ok
4: terminated with result: ok
simulation stopped: no nodes running.
```

6.4.2 case 2

Same setting, just shown another result.

```
18> asg1:start().
1: neighbours: [2,3,4,5]
2: neighbours: [1,3,4,5]
3: neighbours: [1,2,4,5]
4: neighbours: [1,2,3,5]
5: neighbours: [1,2,3,4]
1: ready!
2: ready!
3: ready!
4: ready!
5: ready!
<0.201.0>
Vertex: 3 in MIS
Vertex: 1 NOT in MIS
Vertex: 2 NOT in MIS
Vertex: 4 NOT in MIS
3: terminated with result: ok
Vertex: 5 NOT in MIS
1: terminated with result: ok
2: terminated with result: ok
4: terminated with result: ok
5: terminated with result: ok
simulation stopped: no nodes running.
```

6.5 Test 1.d

```
{ok,asg1}
27> asg1:start().
1: neighbours: [2,5]
2: neighbours: [3,1]
3: neighbours: [4,2]
4: neighbours: [5,3]
5: neighbours: [1,4]
1: ready!
2: ready!
3: ready!
4: ready!
<0.258.0>
5: ready!
Vertex: 4 in MIS
Vertex: 2 in MIS
Vertex: 5 NOT in MIS
Vertex: 3 NOT in MIS
4: terminated with result: ok
Vertex: 1 NOT in MIS
2: terminated with result: ok
3: terminated with result: ok
5: terminated with result: ok
1: terminated with result: ok
simulation stopped: no nodes running.
```

6.6 Test 1.e

Test performed on a complete graph with two nodes.

```
15> asg1:start().
1: neighbours: [2]
2: neighbours: [3]
3: neighbours: [4]
4: neighbours: [5]
5: neighbours: [1]
<0.182.0>
1: ready!
2: ready!
3: ready!
4: ready!
5: ready!
Vertex: 3 in MIS
Vertex: 5 in MIS
Vertex: 1 in MIS
Vertex: 4 NOT in MIS
3: terminated with result: ok
Vertex: 2 NOT in MIS
5: terminated with result: ok
1: terminated with result: ok
4: terminated with result: ok
2: terminated with result: ok
simulation stopped: no nodes running.
```