Flag Coordinator

Network differentiation game

Author	Jaroslaw Pawlak
Student ID	0923141
Supervisor	Prof Peter McBurney

26/04/2013

King's College London

Originality avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary.

Jaroslaw Pawlak

26/04/2013

Abstract

“The graph colouring problem is a natural abstraction of many human and organizational problems in which it is desirable or necessary to distinguish one's behaviour from that of neighbouring parties.”[1] This project aims to produce a simulation software that will allow to observe the performance of particular behaviours of autonomous agents in various network structures. Produced simulation system allows to both observe agents in GUI interface and perform numerous automated tests without GUI.

The project presents two effective ways of determining the number of colours required to colour a graph under certain constraints:

- bipartite graphs can be coloured with 2 colour
- graphs without edges intersections can be coloured with no more than 4 colours

Finally, the project presents similarities between network differentiation and network colouring problems and proves that these problems are equivalent for graphs that can be coloured with 2 colours. It also presents an algorithm for describing network differentiation as network colouring which increases human observer's abilities to analyse an overall performance of the agents.

Acknowledgement

To Prof Peter McBurney for his help through the year and always being able to answer my questions and suggest improvements.

To Prof Michael Kearns whose human subject experiments on graph colouring problem have inspired me to choose this subject as my final year project.

To my family and friends, who helped me make this report cleaner and always were able to look at it or listen about challenges I have been facing, although they were often unable to help.

Table of Contents

Appendices

1 Introduction

Imagine a game in which 20 people are arranged in a circle. Each person has 2 flags, a red flag and a blue flag. The game proceeds in rounds, with each round requiring everyone to raise just one flag. The joint goal of the game (further referred to as consensus) is for the flags to alternate in colours around the circle (i.e. no two the same colours next to each other). If this joint goal is not achieved on the first round, all flags are lowered, and everyone tries again in the next round. This kind of experiment was first proposed by Michael Kearns, who performed a series of such experiments on human subject networks.[1] The initial problem can be further extended by plenty of extra features, such as playing in any network structure, introducing infections (infected players try to achieve opposite goal), changing visibility in the network, changing the number of available flags and so on.

1.1 Aims and objectives

The main object of this project is to build a simulation of this game and analyse how different features affect the number of rounds required to achieve a consensus. This includes:

- building a platform where players' behaviours will be simulated by autonomous agents and whose behaviour can be easily changed
- finding the best (the most optimal) behaviours of the agents given the type of network and set of enabled features
- calculating the average number of rounds required to achieve a consensus in different scenarios (for various types of network, agents' behaviours, enabled features)

1.2 Software platform

The simulation has been created in Java. The main reasons are listed below:

- debugging is fast and effective
- Swing allows to easily create GUI without extra dependencies
- platform independence allows to run the simulation on any machine and any operating system so it can be easily used in any conditions

2 Background

This chapter presents the relevance of this Flag coordinator game to real life problems and background work that is used in the game.

2.1 Relevance to real life problems

Graph colouring is an abstraction of many common more complex problem. The most common among them is scheduling problem – suppose that university exams office has already published exams timetable (modules and days/times) but did not decide on room allocation yet. Each exam can be represented by a node within a graph, exams that are scheduled in the same time (or overlap even partially) are connected with an edge. Colours of nodes represent rooms. By colouring such graph it is possible to determine both how many different rooms are needed (sizes are ignored, we can assume that there is no exam taken by more students than the capacity of the smallest room) and how to allocate exams to rooms.[1]

This was an example of graph colouring problem, where a single observer was performing this task. In case of solving this problem in centralised fashion, probably exams office would allocate the rooms. But what would happen if exams office asked modules' teachers/examiners to solve it on their own, providing them only with information about which exams overlap with their exam? That would in fact require examiners to perform network differentiation in a distributed way – where they have limited knowledge about the graph and have to coordinate with their graph neighbours.[2]

In network differentiation, each node is represented by a human player or autonomous agent, who is responsible of choosing a colour for this node. These players may have limited visible (usually they can only see their local neighbourhood – i.e. first degree neighbours), hence making a task more difficult to achieve. It could be considered an abstraction of choosing a unique profile picture (or rather being different enough from another to avoid confusions) on Facebook account among one's friends. Each person considers only their first degree friends (local neighbourhood) and does not have any knowledge about entire network structure.

Network differentiation or network colouring problems are also abstraction of distributed computing, where many computers work together to achieve a common goal. If the system is not centralised and there are computers that cannot communicate directly, the network and the task become more complex and it may be worth to introduce that kind of abstraction layer.

Finally, network differentiation and network colouring games may be considered a simple simulation of the Internet. Network vertices are computer nodes (computers, routers etc.) while edges symbolise physical connections that allow two nodes to communicate directly. Agent raising a flag can be considered a computer node broadcasting a message to its all neighbours. With introduction of infections, the simulation system allows to simulate malicious nodes (eaves dropping, ddos attacks), where not-infected players should try to identify and isolate infections (by simply ignoring them).

2.2 Minimal required number of flags required to achieve consensus

The important question that has to be asked before simulation is started is how many different flags should be used? How does the number of flags affect the agents' performance? The answer to the second question is rather simple – the more flags the easier it is for the agents to achieve consensus. As the number of flags decreases, the task is becoming more and more challenging, up to some value below which it is no longer possible to achieve a consensus – the minimal number of flags required.

In the most simple case of circle network with even number of agents the minimal number of flags is equal to two. If there is an odd number of agents there have to be at least three flags as presented on the Figure 2.1.

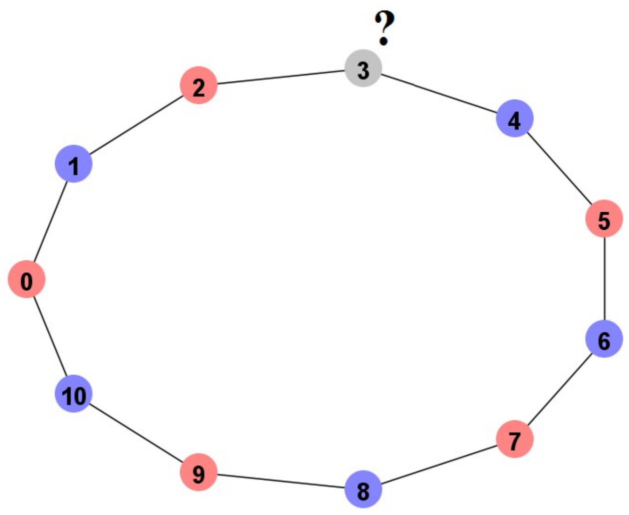


Figure 2.1: eleven players playing on ring network with 2 flags

2.2.1 Why is this important?

If the number of flags is much larger than minimal one, even the random agent will perform relatively well, hence the idea of implementing a good behaviour would be no longer important. Consider a circle network with n number of random agents, where n is even:

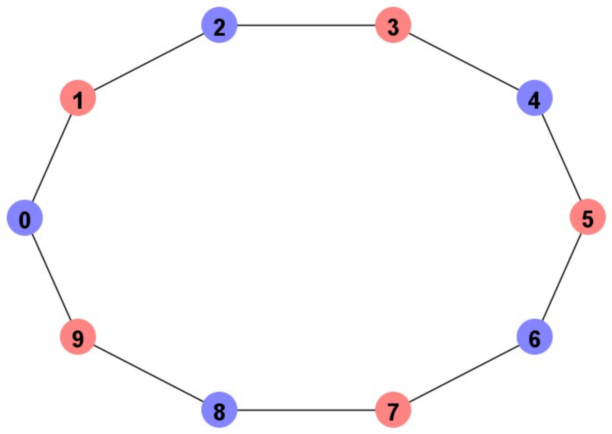


Figure 2.2: One of two possible colourings of 10-node ring network

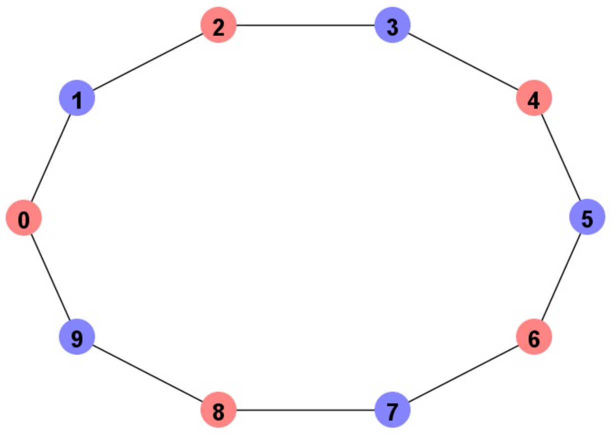


Figure 2.3: One of two possible colourings of 10-node ring network

There are only two possible valid graph colourings – either the first node is assigned blue colour and all the rest are chosen so they avoid conflicts (see Figure 2.2) or similarly with assigning red colour to the first agent (see Figure 2.3). Each agent can raise one of two flags which gives a total of 2^n different (not necessarily valid) colourings. As each agent has 50% chance of raising each flag, the chance that they achieve consensus is:

$$\frac{2}{2^n} = \left(\frac{1}{2}\right)^{n-1}$$

So random agents in this scenario should be able to achieve a consensus on average in 2^{n-1} rounds. In case of 20 agents, the expected average number of rounds is 2^{19} = 524,288. Performed experiment, where 50,000 games were played, resulted in an average equal 526,019.

Consider the same scenario (the same network, n number of agents) but with 4 flags. Let the agent 1 raise any flag, then agent 2 has 75% that it will raise different flag. Agent 3 will have 75% chance that it will raise different flag than agent 2 (so the probability of consensus is $75\%^2$ so far), and so on. Agent n-1 has 75% of raising different flag than agent n-2, with the probability of consensus so far being $75\%^{n-2,}$ as n-2 is the number of edges in the graph we have considered so far. There is only one agent left with 2 neighbours and there are two possible cases – either these two have raised the same flag (25% probability) or they have raised different flags. In the first case the agent has 75% of raising different flag, in the second case the probability is only 50%. Hence the total probability of achieving a consensus is equal:

$$75\%^{n-2} \cdot \left(25\% \cdot 75\% + 75\% \cdot 50\%\right) = \left(\frac{3}{4}\right)^{n-2} \cdot \left(\frac{1}{4}\cdot\frac{3}{4} + \frac{3}{4}\cdot\frac{2}{4}\right) =$$

$$= \left(\frac{3}{4}\right)^{n-2} \cdot \frac{9}{16} = \left(\frac{3}{4}\right)^{n-2} \cdot \left(\frac{3}{4}\right)^{2} = \left(\frac{3}{4}\right)^{n}$$

Different exponent than in 2-flags case may indicate an error. We can validate it by approaching the problem as in 2-flags case. The graph may be coloured in 4^n different ways. How many of them are valid? First agent can raise any flag (chooses from 4 colours). Choice of each next agent is limited to only 3 flags (so it avoids conflict with the previous agent). Last agent is the most difficult to consider as there is 25% chance that its neighbours have raised the same flag (in which case it chooses from 3 flags) and 75% chance that its neighbours have raised different flags (in which case it chooses from 2 colours). This gives

$$4 \cdot 3^{n-2} \cdot \left(25\% \cdot 3 + 75\% \cdot 2\right) = 4 \cdot 3^{n-2} \cdot \left(\frac{1}{4}\cdot 3 + \frac{3}{4}\cdot 2\right) =$$

$$= 4 \cdot 3^{n-2} \cdot \left(\frac{3}{4} + \frac{6}{4}\right) = 4 \cdot 3^{n-2} \cdot \frac{9}{4} = 3^n \text{ valid colourings.}$$

Summing up, n random agents playing with 4 flags on the circle network, have $\frac{3}{4}^n$ chance of achieving a consensus in any round. The expected average number of rounds for 20-agent network is then equal:

$$\left(\frac{4}{3}\right)^{20} \approx 315.3$$

One million games played have resulted in an average equal 314.9.

2.2.2 How to calculate minimal number of flags required

Reasoning presented above shows that the number of different flags is important and directly affects the agents' performance. However, the most important question – how to calculate the minimal number of flags required to achieve a consensus – has not been answered yet. To have the most correct answer, the graph colouring algorithm should be used to actually colour the graph. Graph colouring is however NP-hard and is a complex problem on its own. The fastest algorithms have exponential time complexity, such as $O(2.445^n)$[7] or $O(2^n n)$[8]. It is possible to colour the graph with 3 or 4 colours in time $O(1.3289^n)$[9] and $O(1.7504^n)$[10] respectively (assuming that it is possible). However, there are two fast ways of defining whether a graph can be coloured with 2 or 4 colours (without necessity of doing actual colouring).

Bipartite graph

Bipartite graph is a graph whose nodes can be divided into two disjoint sets such that no edge connects two nodes within the same set (see Figure 2.4). Each tree is bipartite graph. In case of graphs with cycles, a graph is bipartite if each cycle is made from even number of nodes. It is possible to check if graph is bipartite in O(n) time where n is the number of nodes with the following algorithm:

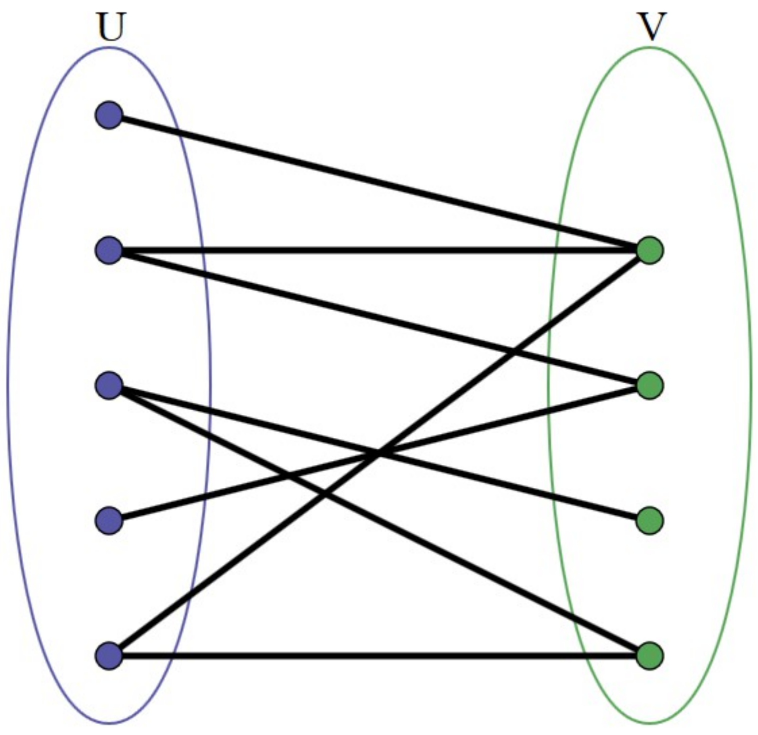


Figure 2.4: Bipartite graph
Source: *http://en.wikipedia.org/wiki/File:Simple-bipartite-graph.svg*
Accessed: 20/04/2013

```
Choose any starting node (root) and assign any of two available colours to it
Traverse through the graph (any algorithm can be used, e.g. BFS or DFS)
    if among neighbours of currently processed node there are two nodes that have been
    assigned different colours, the graph cannot be coloured with two colours – algorithm
    terminates
    assign to current node a colour that was not assigned to any of its neighbours
Once all nodes have been processed, it means that the graph can be coloured with two nodes
```

Algorithm 2.1: Bipartite graph checking

This algorithm in fact colours the graph.

Four Colour Theorem

Four colour theorem states that given a plane divided into regions (further referred to as map), the regions can be coloured, in that way that no two adjacent regions have the same colour, with maximum 4 colours. Corners will be called such points that are shared between at least three regions. Regions are considered adjacent if they share a boundary (i.e. line, curve, polyline etc.) between two corners.

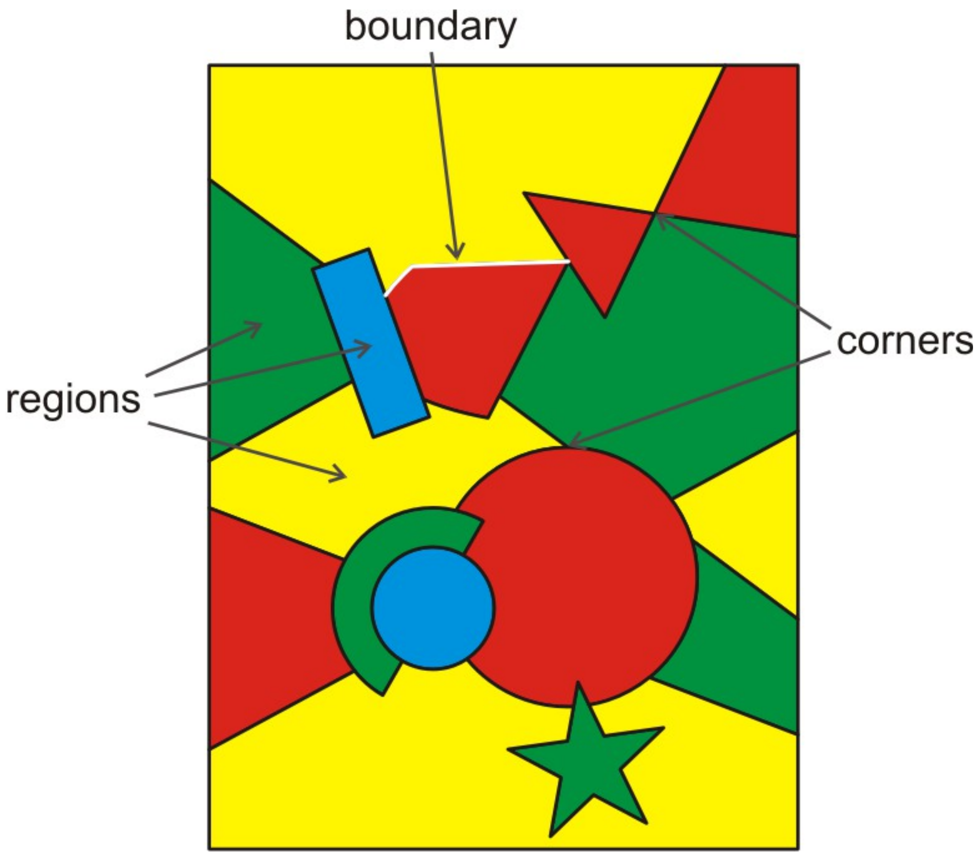


Figure 2.5: Example map

Translating map into graph

At first it may seem that four colour theorem has nothing to do with the graphs, however it is not true. From each map (plane divided into regions) it is possible to build a classic graph with nodes and edges.

```
inside each region create a node
for each pair of adjacent regions
    create an edge between nodes inside these regions
```

Algorithm 2.2: Translating map into graph

When creating edges it may be good idea to avoid intersecting other boundaries or other already created edges. Edges not necessarily have to be lines, it may be necessary for them to be curves (see Figure 2.6). The graph's nodes can be later rearranged so that all edges are straight line segments (see Figure 2.7).

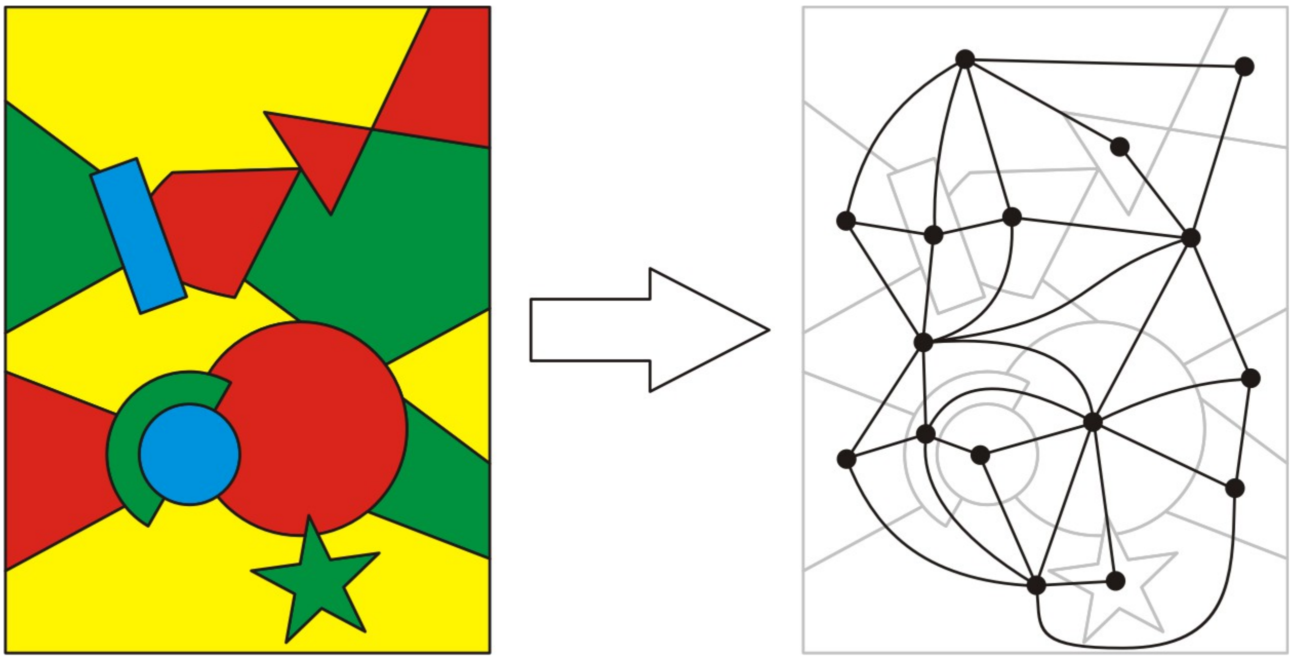


Figure 2.6: Translating map into graph

Translating graph into map

But the question is, is it possible to build a map from a graph or to prove that it is not possible? What constraints does a graph have to meet to make it possible to create a map from it? In the figures above it should be possible to spot that no two edges intersect. It seems to be both necessary and sufficient condition. First, consider what happens if two edges intersect (and obviously it is not possible to rearrange the nodes to avoid this intersection).

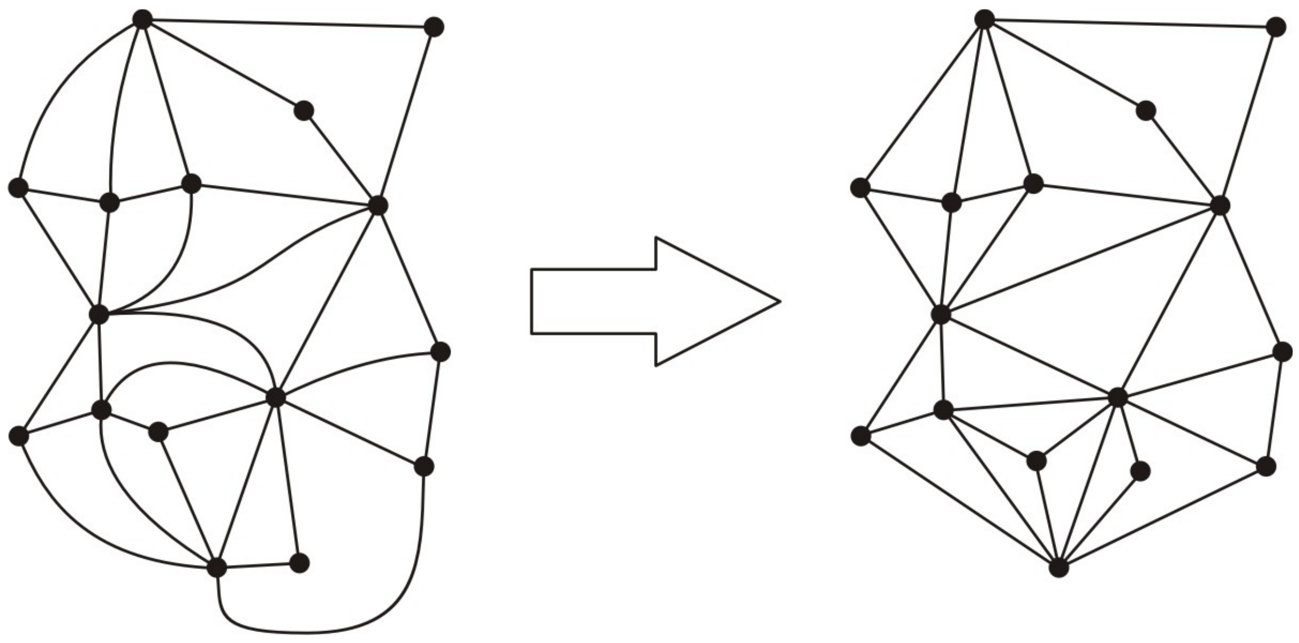


Figure 2.7: Rearranging nodes and straightening edges of the graph obtained by translating a map

Theorem #1: Graph containing edges intersections cannot be translated into map without rearranging its vertices.

Consider four nodes with two intersecting edges and try to create a map from it without rearranging nodes (i.e. under an assumption that it cannot be rearranged that way). This is presented on Figure 2.8.

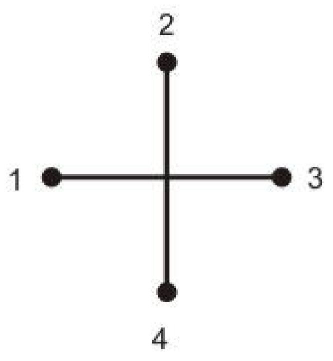


Figure 2.8: Edges intersection

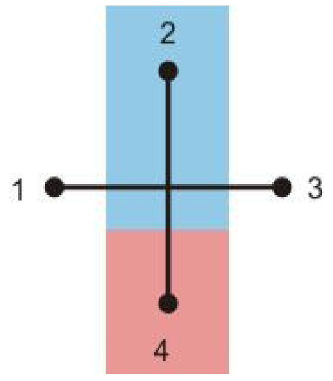


Figure 2.9: Partial map, full map cannot be constructed due to edges intersection

First, create a part of map from nodes 2 and 4 (see Figure 2.9).

These two regions create a belt of non-zero width which cannot be passed through because otherwise regions 2 and 4 would no longer be adjacent. Hence, regions 1 and 3 cannot share a boundary and it is not possible to create a map from this part of graph (how the rest of the graph look like is not important and hence not considered here).

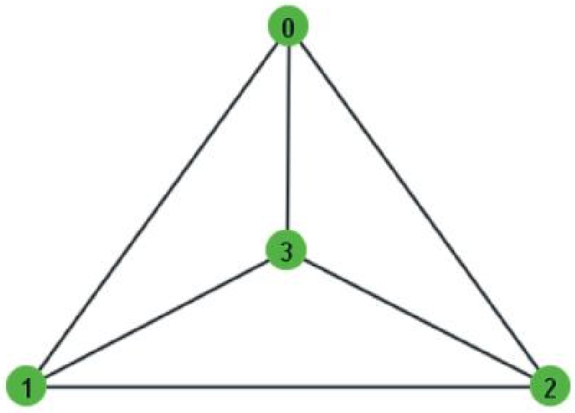


Figure 2.10: 4-node fully-connected graph

One could say that these nodes could be rearranged or that regions 1 and 3 could share a boundary below region 4, at the bottom of the plane. In first case it may not be possible to rearrange a graph in that way. In the second case it would be true only if it would be a whole graph – but it was just a part of it. To better show that it is not possible, consider the smallest possible graph where edges intersections cannot be avoided – a 5-node fully-connected graph. Figure 2.10 presents 4-node fully-connected graph without intersections and one more node has to be added to this. No matter where it is put, there will always be at least one intersection:

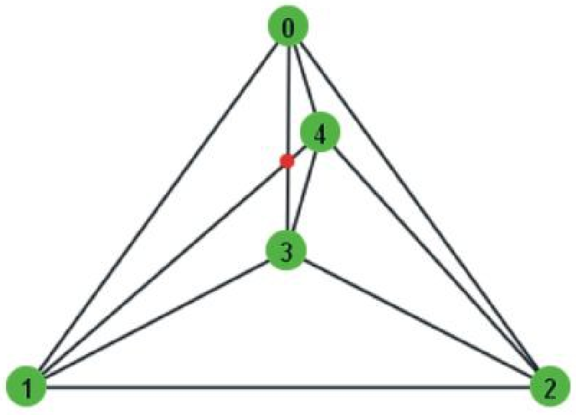


Figure 2.11: 5-node fully-connected graph

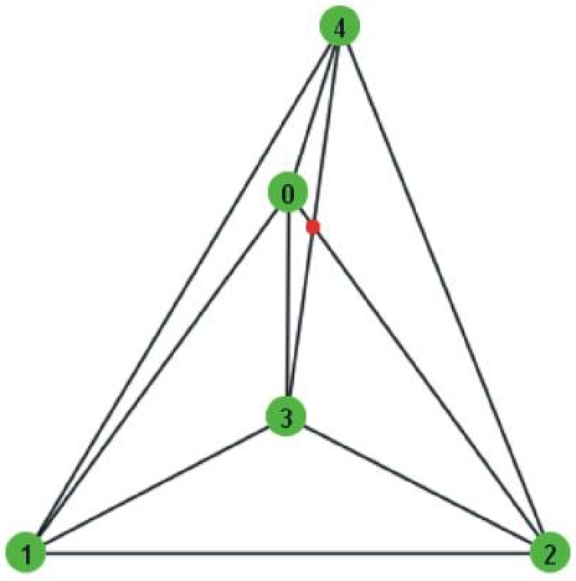


Figure 2.12: 5-node fully-connected graph

What we are trying to do, in map would be equivalent to dividing a plane into five regions, each of them being adjacent to all others. First, consider only three regions:

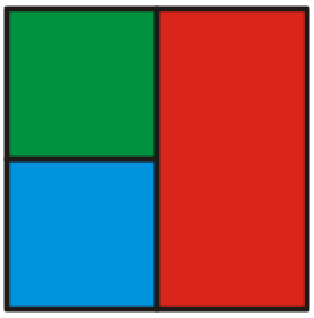


Figure 2.13: Map with three regions adjacent to each other

The fourth region has to be drawn, such that it is adjacent to all other three. This cannot be done without completely surrounding one of the regions:

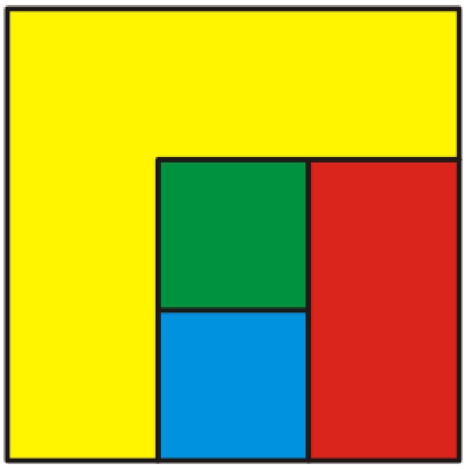


Figure 2.14: Map with 4 regions adjacent to each other, where green region is completely surrounded

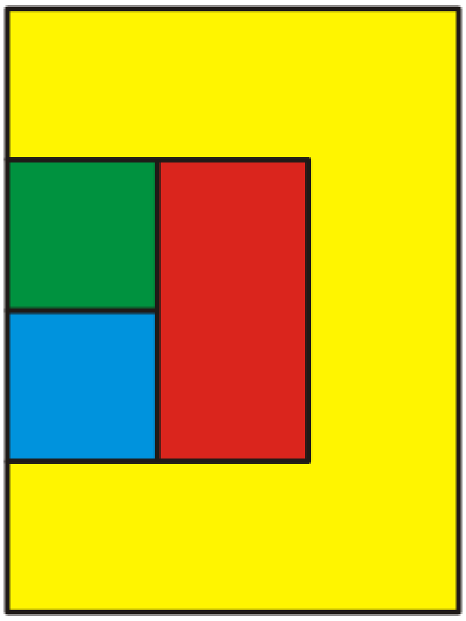


Figure 2.15: Map with 4 regions adjacent to each other, where red region is completely surrounded

It means that if the fifth region is drawn, it cannot be adjacent to the surrounded region without creating "a bridge" over existing region – which would be a graph edge intersection.

Theorem #2: Each graph that does not contain edges intersections, can be translated into map.

Proof (by construction): Suppose G is a graph that does not contain edges intersections. The following algorithm translates G into a map.

```
having a graphical representation of a graph identify all polygons (these are in fact cycles)
create a new corner in each of the polygons
for each two adjacent polygons (i.e. such that they share at least one edge)
        for each edge they share
                connect corners of these two polygons with a line crossing this edge
connections between corners are region boundaries in the map
```

Algorithm 2.3: Translating graph into map

Figures 2.16, 2.17 and 2.18 present a map construction at different steps of the algorithm.

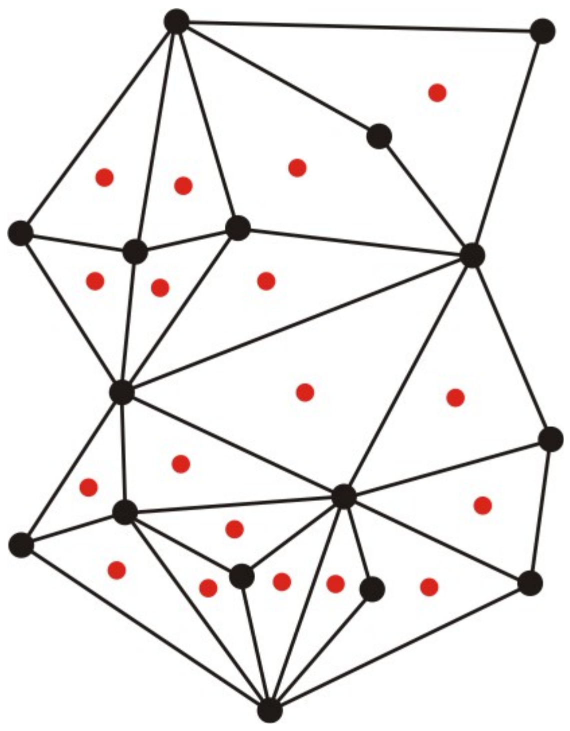


Figure 2.16: Graph being translated into a map, corners of a map in red

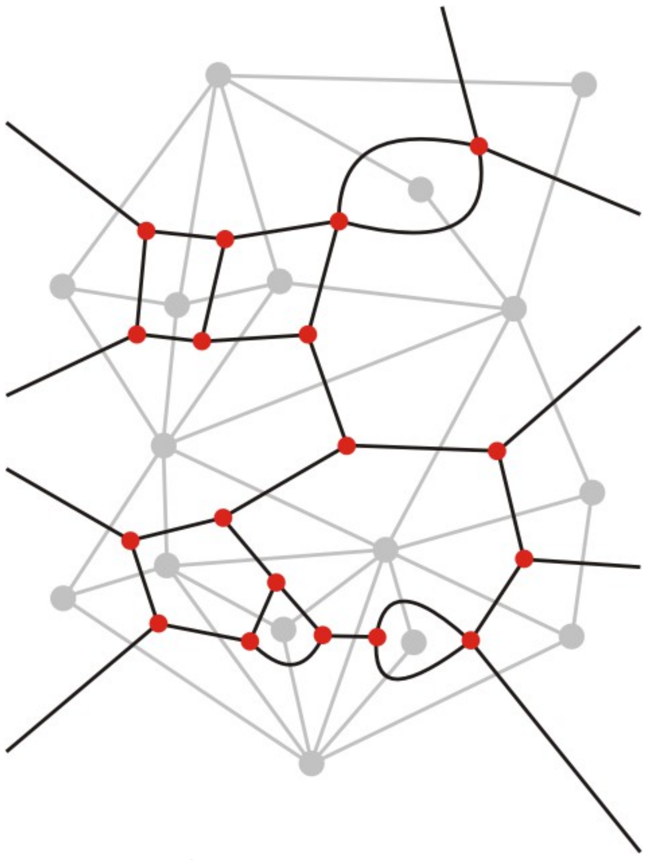


Figure 2.17: Graph being translated into a map, regions boundaries in black

Conclusion

Summing up, every graph that has no edges intersections (or can be represented in that way), based on four colours theorem, can be coloured with no more than four colours.

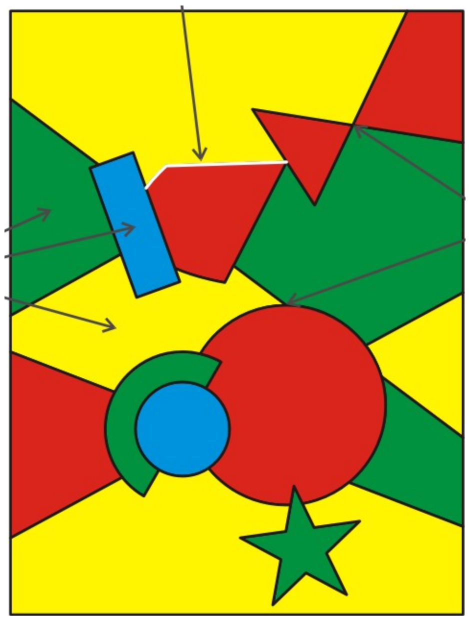


Figure 2.5 (cropped):
Example map

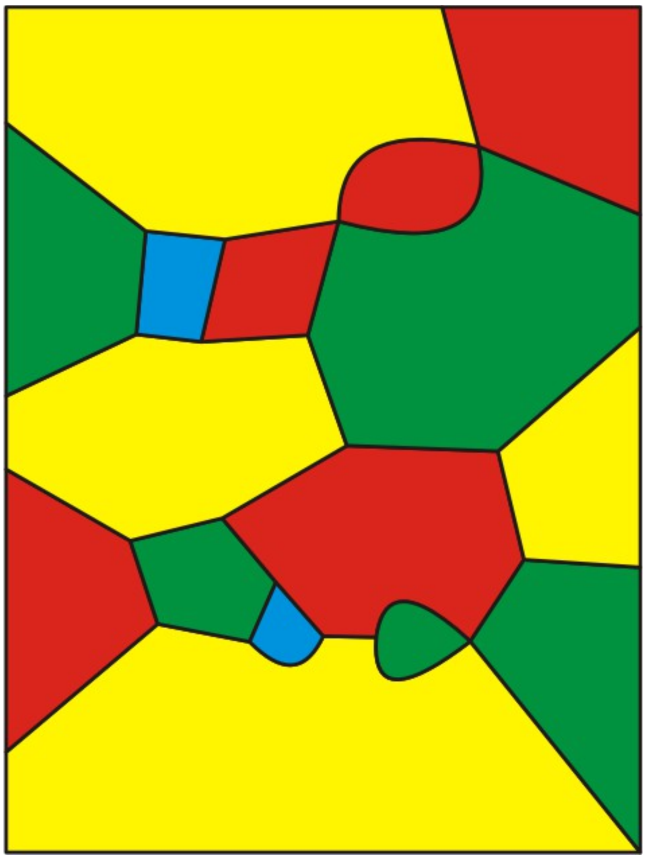


Figure 2.18: Map created by translating map from Figure 2.5 into graph and back into map

2.3 Network differentiation versus network colouring

It has been showed that network differentiation is more difficult to achieve for human-subject networks than network colouring. Furthermore, with the increase of connectivity, differentiation problem becomes harder while colouring easier.[3]

As in this project game is played by autonomous agents, this difference does not affect agents drastically. However, it affects a human observer who is evaluating agents' performance. Initial tests with circle-simulation have shown difficulties in the observations of the whole network's state and estimating how far the agents are from their goal.

A few interesting questions arise here. Is it possible to present network differentiation game as network colouring? Do these two can be freely translated one into another with one to one mapping? Are these potentially so different problems, in fact equivalent?

Not all of these questions can be answered immediately. First, consider a circle network with even number of agents who can raise one of two available flags – red or blue. Let us assign numbers to the agents in circle from 0 to n - 1, where n is the number of agents (these are in fact agents' IDs). Here we need to introduce term “state” and define it as follows:

If agent 0 has raised a red flag, and consensus has been achieved, the circle is in the green state (see Figure 2.19). If agent 0 has raised a blue flag, and consensus has been achieved, the circle is in the yellow state (see Figure 2.20). If agents 0 to 3 have all raised a red flag, agents 0 and 2 are in green state, while agents 1 and 3 are in yellow state (see Figure 2.21). And so on.

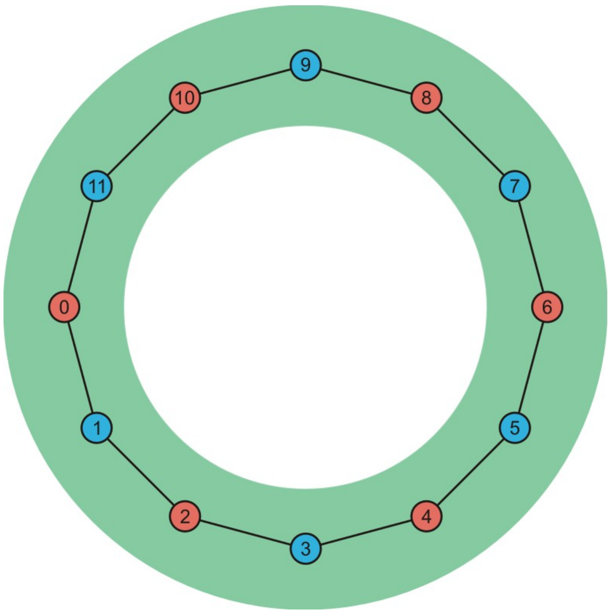


Figure 2.19: Network differentiation achieved, players are all in the same state

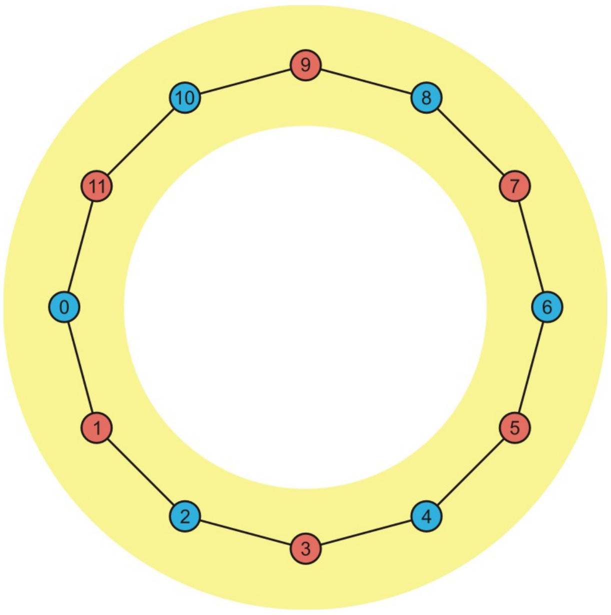


Figure: 2.20: Network differentiation achieved, players are all in the same state

The aim of the game is that all agents are in the same state. Hence, instead of raising red/blue flag symbolising a colour in network differentiation problem, agents may as well raise green/yellow flag symbolising a colour in network colouring problem. For each graph that can be coloured with two colours network differentiation and network colouring problems become equivalent with a one to one mapping.

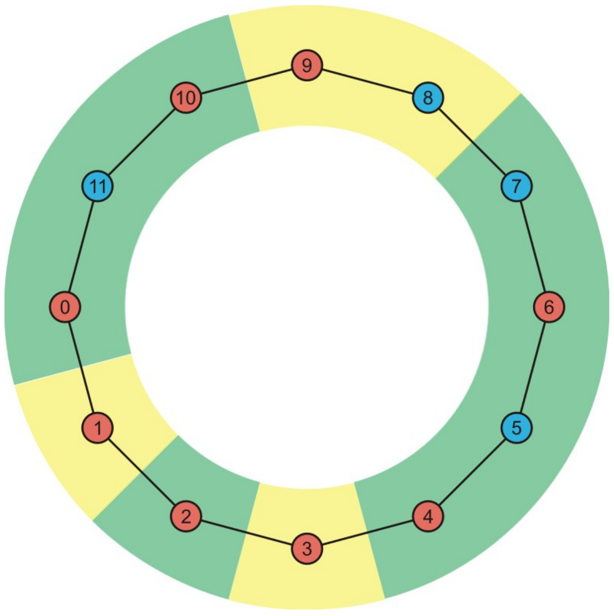


Figure 2.21: Game in progress

This reverse of network differentiation problem becomes more complex as the number of flags increases. In case of ring network played with 3 flags, there is 3^n different valid colourings. It is not possible to introduce that many states as it would not simplify the problem. Hence, it will not be possible to have one to one mapping with this approach.

However, it is worth to notice that it is not really important if one agent has raised a blue flag, while its neighbours red – or if it raised red flag while its neighbours blue. Both these cases could be mapped to the same state. Of course it will not be possible to calculate what flags were actually raised having only agents' states, but this is not a requirement.

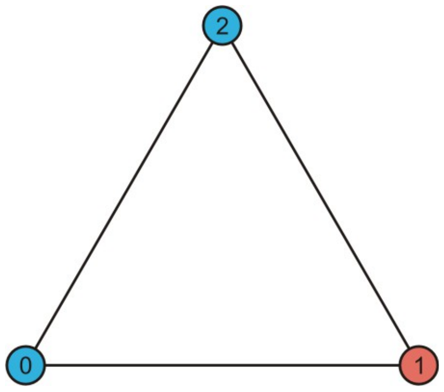


Figure 2.22: fully-connected 3-node network

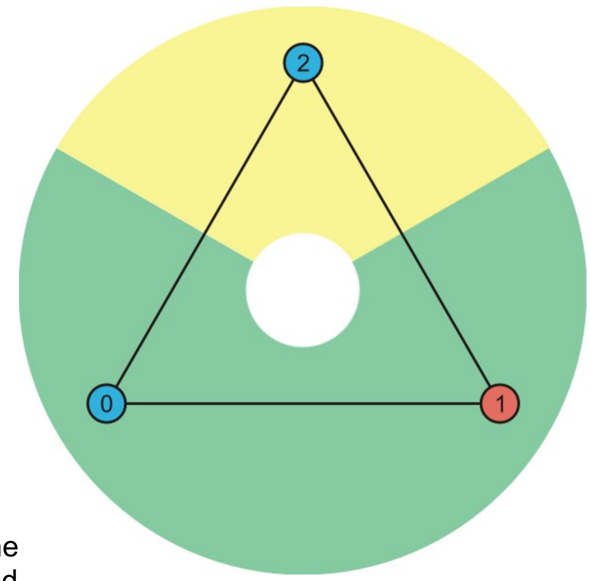


Figure 2.23: States have been assigned to agents from Figure 2.22 using algorithm 2.4

Consider a fully-connected 3-node network, where one agent has raised red flag while two others have raised blue flag (see Figure 2.22). Agents 0 and 1 have raised different flags so they are in the same state. Agents 1 and 2 have raised different flags, so they are in the same state too. But this results in contradiction as agents 0 and 2 are in the same state despite raising the same flag. Hence, the rule of assigning a state has to be slightly changed and the assignment can be performed by the following algorithm:

```
consider only those neighbours that have been already assigned a state
for each neighbour A that has raised different flag
    if there is no other neighbour B such that A's and B's states are the same and B's flag is
    the same as currently processed node's flag
        assign A's state to current node and terminate
otherwise, assign new state (i.e. such that it is not among neighbours) to the current node
```

Algorithm 2.4: Assigning states to agents

Consider a previous example of fully-connected 3-node network again, but this time applying above algorithm:

- Agent 0 is processed and it receives new state – green.
- Agent 1 is processed:
  Agent A = 0 has raised different flag and there is no other agent with green state (A's state) and red flag (1's flag), hence 1 receives the same state as 0 – green.
- Agent 2 is processed:
  Agent A = 1 has raised different flag, but there is an agent B = 0 that has the same state as A (i.e. green) and raised the same flag as 2 (blue). Hence 2 cannot be assigned green state and new state has to be used – yellow.

In this network colouring representation it may seem that it is enough that agent 2 adjusts to other two by just changing its state to green. However, if the game is played with 2 flags only, it is not possible. Implementing network background colouring (as green/yellow on above figure) may still be useful for human observer – it will allow to easily notice where the actual problems occur, but in these places the human observer will have to rely on actual flags and not the background colouring.

Conclusion

Network differentiation and network colouring are equivalent problems if the game is played with two colours. In other cases, problem reverse may be useful in evaluating overall performance of the whole network, but cannot be a substitute of network differentiation.

3 Requirements and specification

Simulation system should provide the following features:

Playing a game on any network

The simulation should provide an ability to play on any network – with any number of nodes and edges, with any number of agent's neighbours. It should provide the functionality necessary to access simulation's state, such as information about each agent or whether a consensus was achieved in the last round. It should be able to run without any GUI components. Simulation should also allow agents to obtain necessary information about their neighbours. As the network is unknown prior to game start, it has to be possible to define the number of available colours to play with.

Tool that allows to easily build networks

Network creator is a GUI component that should allow to easily build or modify networks. Networks should be able to be saved to and loaded from a text file. It should be possible to run a simulation without using network creator GUI, by just loading a network from a file and providing it to a new simulation.

Tool that analyses the network and provides its statistics

Network statistics should provide a short characteristic of the network. It should contain not only simple attributes such as number of nodes/edges, but also information about minimum, average or maximum degree (number of agent's neighbours).

Tool that automatically builds some common networks

User should not be required to build each network from scratch and should be provided with some ready-to-use networks. Instead of providing predefined networks, the simulation should provide network generators, that allow to generate a given type of network with given attributes (these are depended on the the network type). Network types that should be available include:

- ring – with given number of nodes
- star – with given number of nodes
- grid – with given width and height (counted in nodes)

- hex – with given width and height (counted in nodes) and optional extra edges on sides
- fully connected graph – with given number of nodes
- full tree – with given number of children and levels
- random – with the maximum number of nodes and edges

Random network generator can be a simple implementation that does not care about nodes' positions (as long as they do not overlap) or edges intersections. In other words, it may be highly probable that the network will have to be slightly modified to become human-readable. This generator should also have a time limit to ensure that it always returns a network – in case of timeout the number of nodes or edges may be smaller than requested.

Tools for finding the number of colours required to colour a graph

The simulation should provide tools that allow to give a user an idea of how many flags should be used in the game.

- Graph painter

  Simple graph colouring algorithm that is fast and always colours a graph, however does not guarantee that it will be done with a minimal number of colours. This will in fact provide the maximum number of required colours, with minimal number being possibly lower.

- Edges intersections detector

  If no two edges intersect in the graph, based on Four Colours Theorem, no more than four colours are required to colour the graph. A simple algorithm should be implemented that will allow to check if graph contains edges intersections.

Game history

The simulation should provide a user with a game history in GUI mode. It should be possible to disable history in the simulation back-end in order to save system resources when the history is not required – for example when automated testing without GUI is performed.

Consensus mode

The simulation should allow to define whether the goal of the game is to achieve network differentiation or network colouring.

Playing after consensus was achieved

It should be possible to continue the game even after a consensus was achieved. This will allow not only to test how fast agents are able to achieve an agreement, but also to test if they are able to remain in an agreed state.

Infections

The simulation should provide back-end functionality for handling infections. It should provide a proper initialisation that allows to assign different behaviour to the chosen agents. The simulation GUI should provide a user interface that allows to easily choose which agents are infected and what behaviour they should use.

Including infections in consensus

The simulation should allow to define whether the infected agents should be considered in consensus or not. Enabling this option will allow to test whether agents are able to achieve a goal despite minor malfunctions of the whole system.

Accessing agents memory

Once the overall performance of an agent has been analysed, user may want to find out why agent make particular decisions. Simulation should provide a tool that allows to access and modify fields of agent's class. This should be done without requiring the user (who provides agent's behaviour) to implement any functionality, hence this tool should use Java Reflection API to make it agent independent.

Experiment Scheduler

Experiment scheduler should allow user to schedule what tests should be run and how many times. All output should be saved to the text files. Experiment scheduler should ensure that if the simulation crashes (e.g. due to exception thrown in agent's implementation), other experiments will not be affected. It should also provide an ability to perform many experiments at the same time on the given number of threads.

4 Design

Simulation system in fact consists of two simulations. An early prototype where network structure is limited to ring and the final simulation with all features enlisted in the requirements. The simulation has to be implemented separately for circle network and graph network.

Both simulations consist of four main components:

- Agent
- AgentInfo
- Simulation
- GUI

Agent is actual implementation of the particular behaviour. The agent should be able to get previously raised flags by other agents it can see. It cannot access the history or get any other information about the game, unless it is implemented in its behaviour. The Simulation should give the agent the number of the new round and the agent should return the flag it wants to raise.

AgentInfo is the wrapper of Agent for the Simulation. It contains information that Agent should not be able to access, such as Agent’s unique ID. In case of graph network, it will be the node within the network.

Simulation is the core part that joins together agents and defines game rules. It should provide functionality to update the network by one round and to run until consensus is achieved and provide the number of rounds it took – so the automatic testing is possible without GUI. It should be able to provide all information about the network and agents to GUI. It should provide the optional ability to remember the history and interface to access it. It may need to store some extra information about agents such as their position on screen.

Any-network simulation

Graph simulation should be further extended by the network representation that will provide number of nodes, their neighbours and on-screen positions of nodes. Nodes and neighbours definitions will be used when creating the Simulation to connect agents properly. Positions of nodes will be used by GUI components to display a network.

GUI of any-network simulation will be split into four main components:

- Network creator
- Simulation initialisation
- Actual simulation
- Game history

Network creator will allow user to create a network. It should also provide an interface to modify it, save to and load from a file. It should also provide access to network generators.

Simulation initialisation should allow user to choose various features of the game before the game starts. This may include:

- number of flags
- consensus mode (differentiation or colouring)
- agents' behaviour
- infections and their behaviour
- visibility

5 Implementation

Both simulations were implemented using Rapid Application Development. Circle-simulation was an early prototype that allowed to test initial ideas and requirements. Once the simulation became functional, all work has been focused on any-network simulation.

5.1 Implementation results

One of the most difficult decisions (due to the number of different possible approaches) was how to display the network, and, at the same time, give a user access to the agents. Circle-simulation has been mixing drawing on Graphics to draw edges and JLabels with absolute positioning for representing agents (see Figure 5.1). This approach has revealed integration problems between Graphics object and JLabels, as both of them were responsible for the network display. This in turn has resulted in occasional incorrect repainting of the container – especially when resizing the window or using right-click pop-up menu.

Due to these reasons, final simulation is drawing a whole network using only Graphics object while all the other functionality is provided via side or menu bars. Any-network simulation consists of four main GUI components presented on the Figures 5.2, 5.3, 5.4 and 5.5 below.

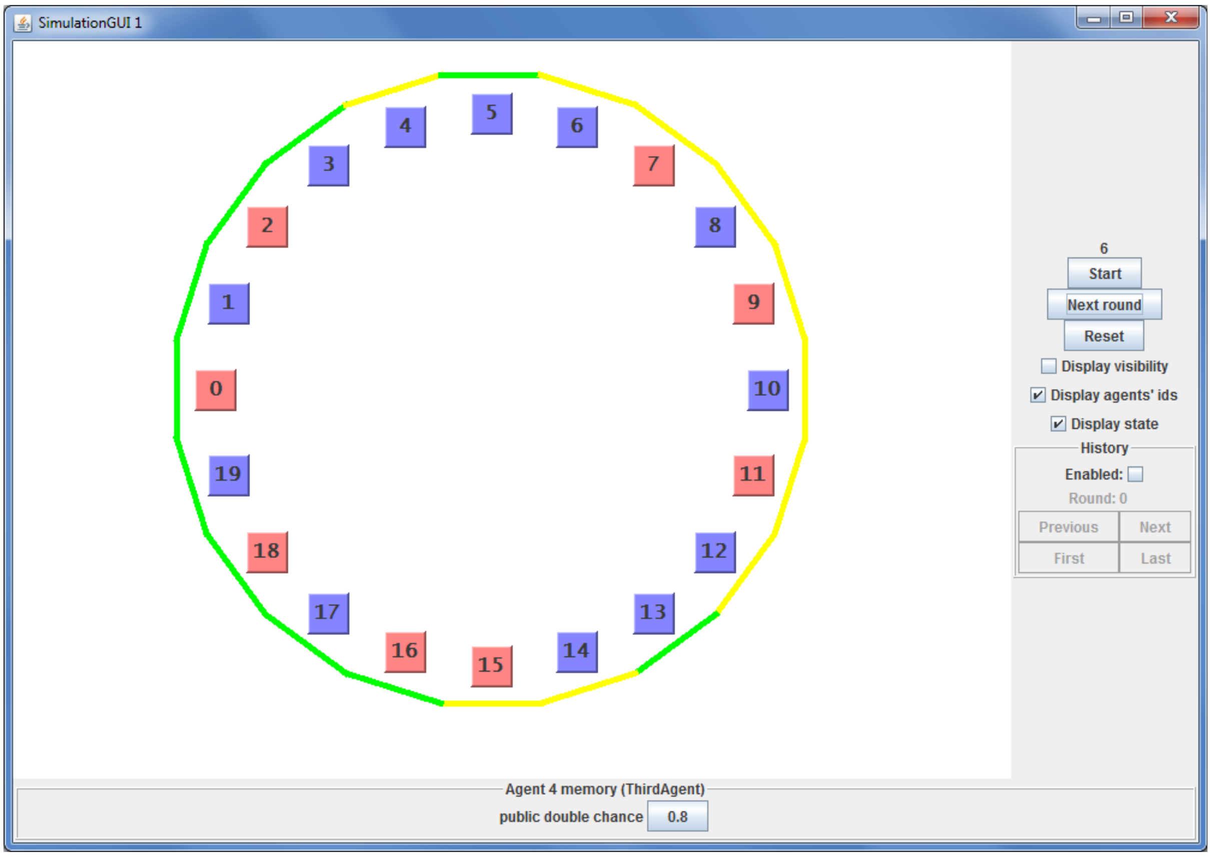


Figure 5.1: Prototype circle-simulation

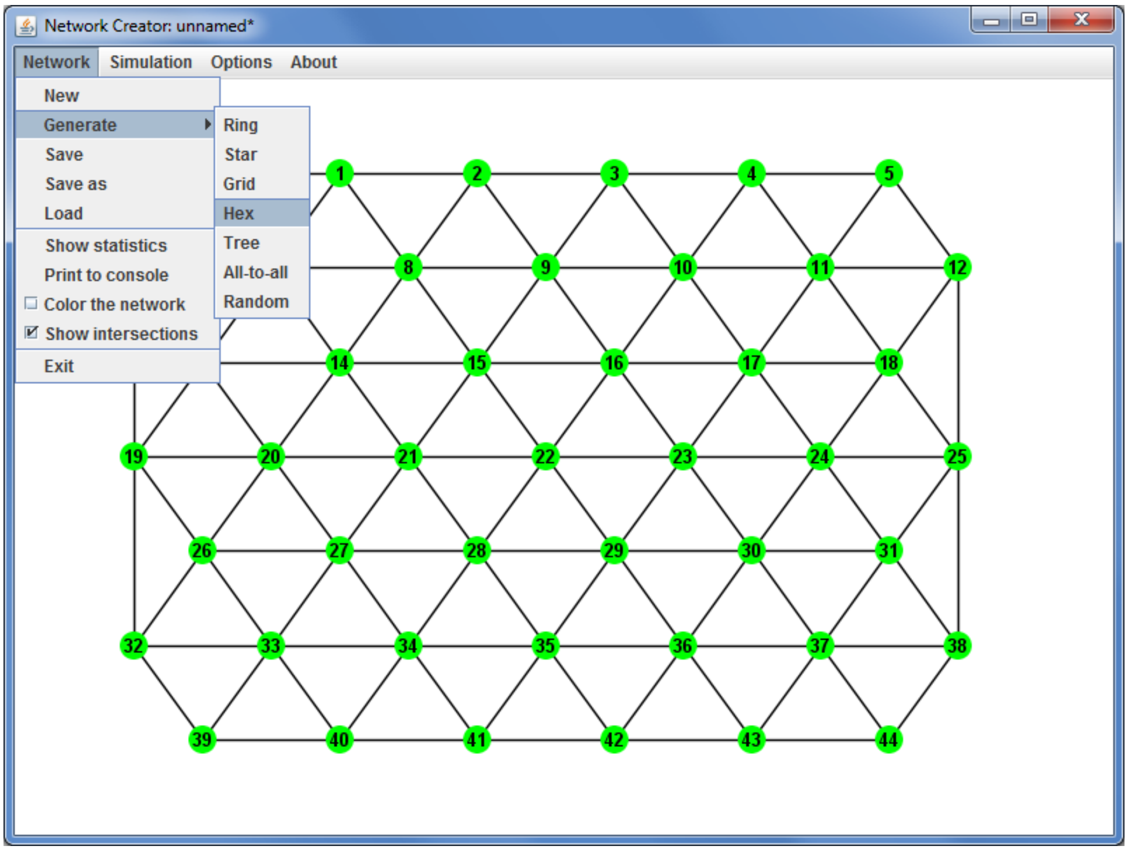
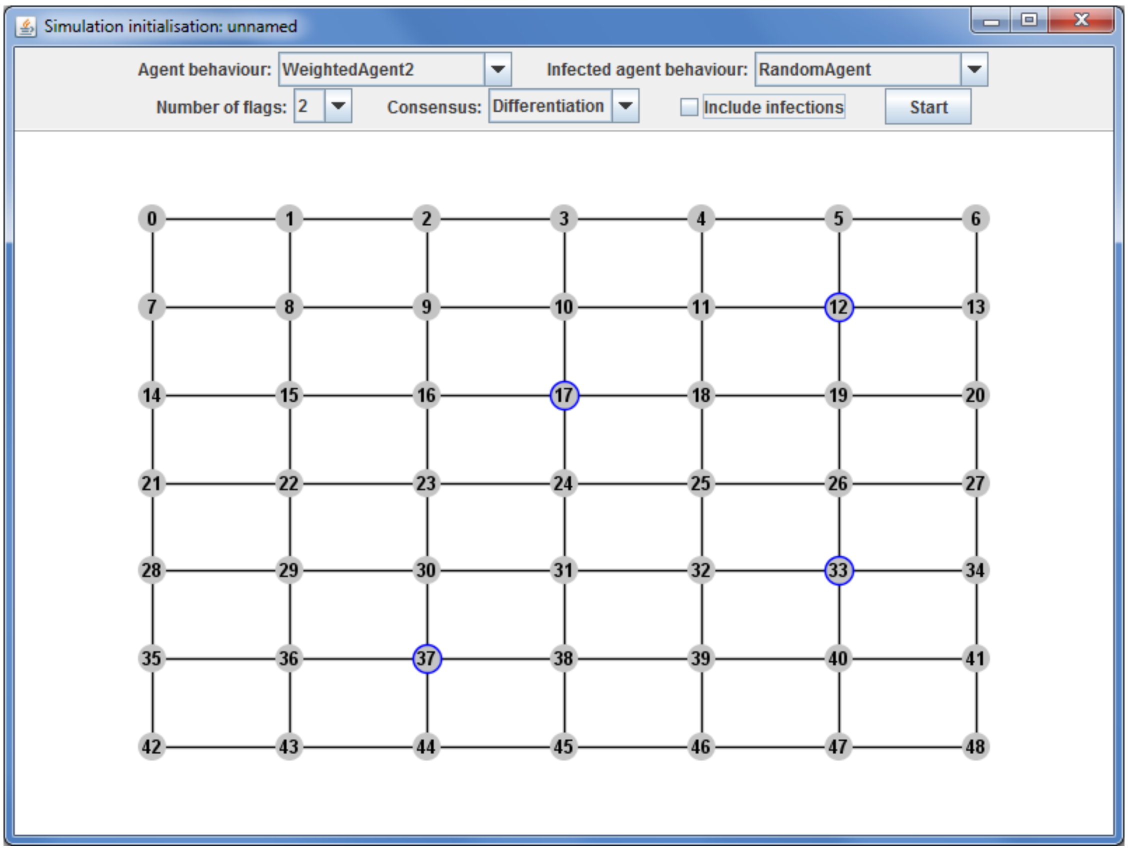


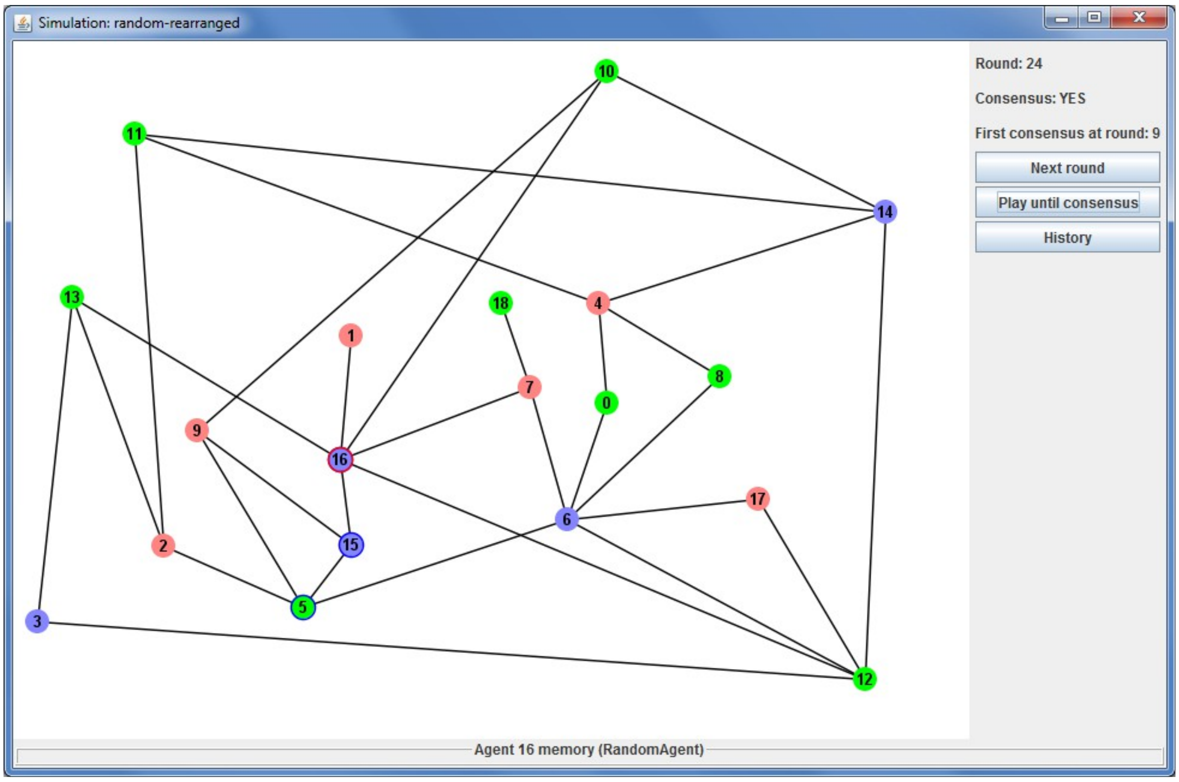
Figure 5.2: Network creator window of final simulation showing 7x7 hex network


Simulation initialisation: unnamed

Agent behaviour: WeightedAgent2 Infected agent behaviour: RandomAgent

Number of flags: 2 Consensus: Differentiation ☐ Include infections

Figure 5.3: Simulation initialisation. Selected agents will use infected behaviour


Simulation: random-rearranged

Round: 24

Consensus: YES

First consensus at round: 9

Agent 16 memory (RandomAgent)

Figure 5.4: Game in progress on random network. Agents 5 and 15 are infected. Agent 16 is shown in agent memory modifier. Infections are ignored in consensus checking.

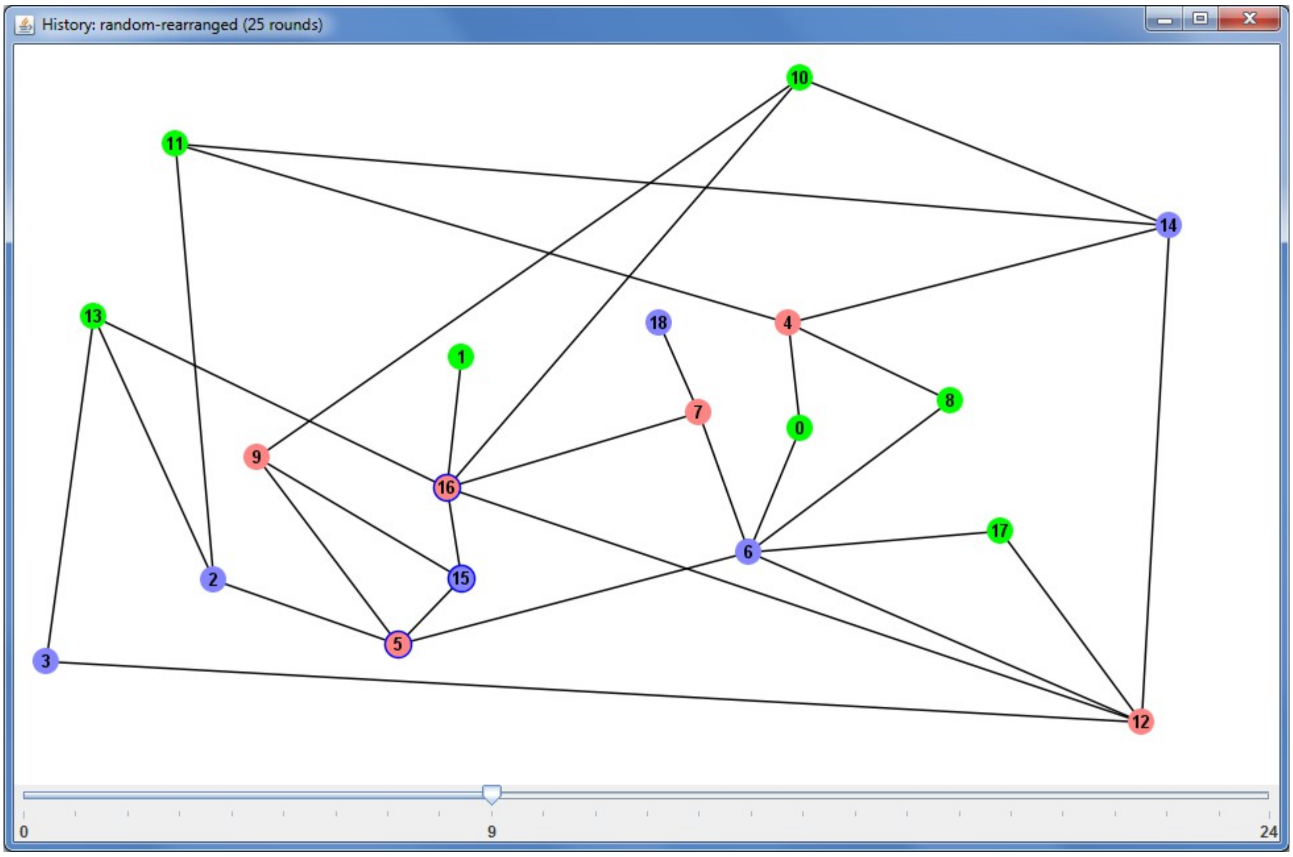


Figure 5.5: History of the game presented in Figure 5.4 currently showing round 9.

5.2 Testing and evaluation

Both final simulation and the prototype circle-simulation have been firstly implemented without GUI and tested using unit testing approach. Once back-end was functional and simulations could be run successfully, GUI components have been created providing a user with easy way of using most of the simulation's features. This work has been followed by integration tests, which allowed to ensure that all implemented features are fully functional via graphical interface.

Performance of the whole system could not be measured due to the fact that it directly depends on agents' implementation – and these are not limited in time or in number of operations when they are making decisions.

5.3 Challenges faced

This chapter briefly describes some of the most interesting challenges faced during the implementation of the simulation system.

5.3.1 Round based simulation versus real-time simulation

The implemented simulation system uses rounds – in each round all agents make a decision at the same time. Agents are not limited in time and they can use extremely complex (and slow) algorithms, as only the number of rounds required to achieve consensus matters. Another option was to give each agent its own thread and let them play in a real time. Agents would be able to make a decision whenever they want while the simulation should provide notifications to avoid agents continuously checking their neighbours' states. Both approaches have their advantages and disadvantages as listed below:

Advantages of round-based simulation over thread-based simulation:

- easy to implement

  It results in faster progress and lower amount of time wasted on implementing complex systems instead of doing more valuable research and analysis.

- easy to control and debug

  A single threaded application can be easily debugged, there are no synchronisation problems.

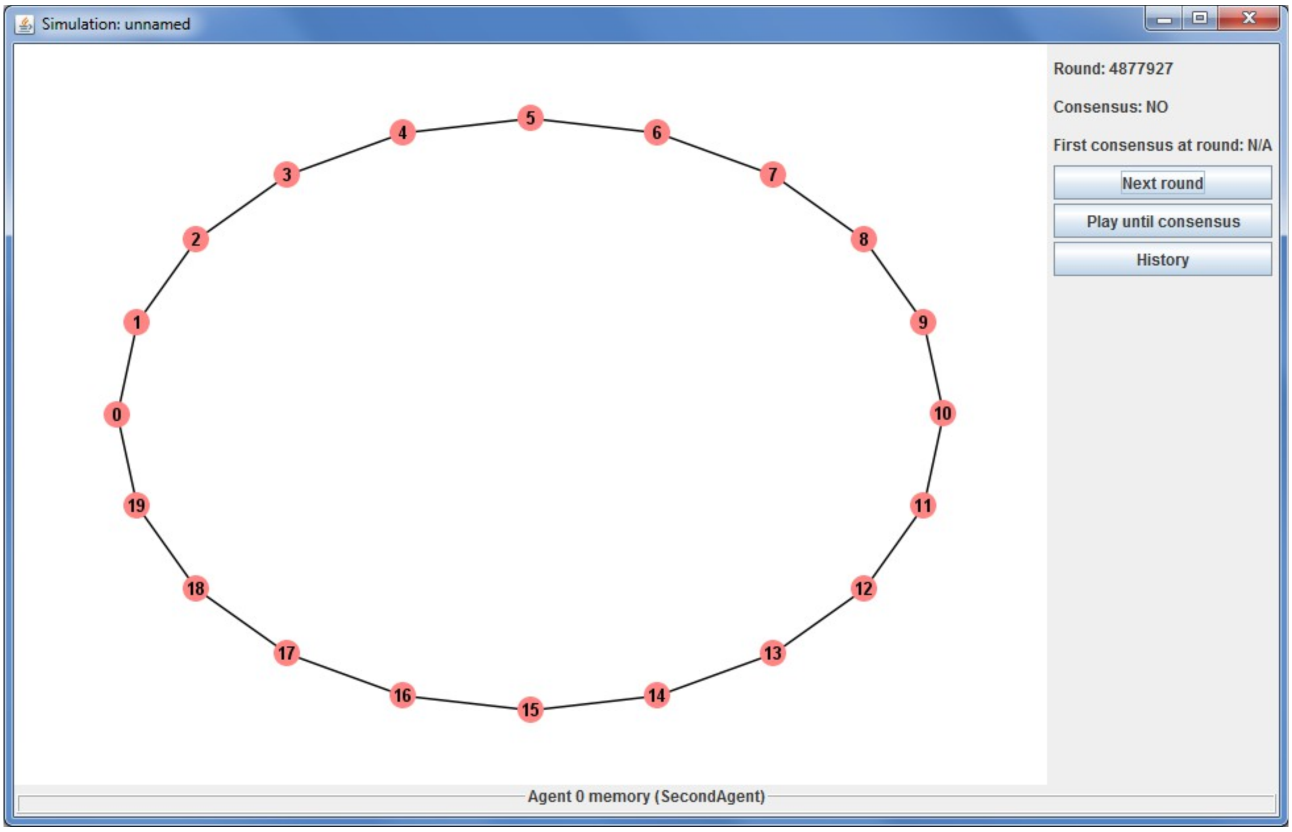
- agents' performance does not depend on the hardware on which simulation is run

  Simulations can be run on any machines (although it will take more time on slower machines) the total number of rounds will be the same and the results of simulations run on different machines can be compared.

Advantages of thread-based simulation over round-based simulation:

- better reflects real-life problems

  In distributed systems or computer networks, nodes are not limited to make their decisions in any particular moments. The common problem of round-based simulation is that agents' behaviour cannot be deterministic. Deterministic behaviour is such behaviour that results in the same output (flag raised) for the same input (neighbours flags). It is best seen in 2-flag game. When agent who raised red flag see that all its neighbours have also raised red flags, it is reasonable to raise blue flag next round. But if all agents within the network have raised red flags, in the next round all of them will raise blue flags, and this results in vicious circle that agents are unable to stop – hence they will never achieve consensus in this game. Figures 5.6, 5.7 and 5.8 on the next page show three consecutive rounds of a stuck game.



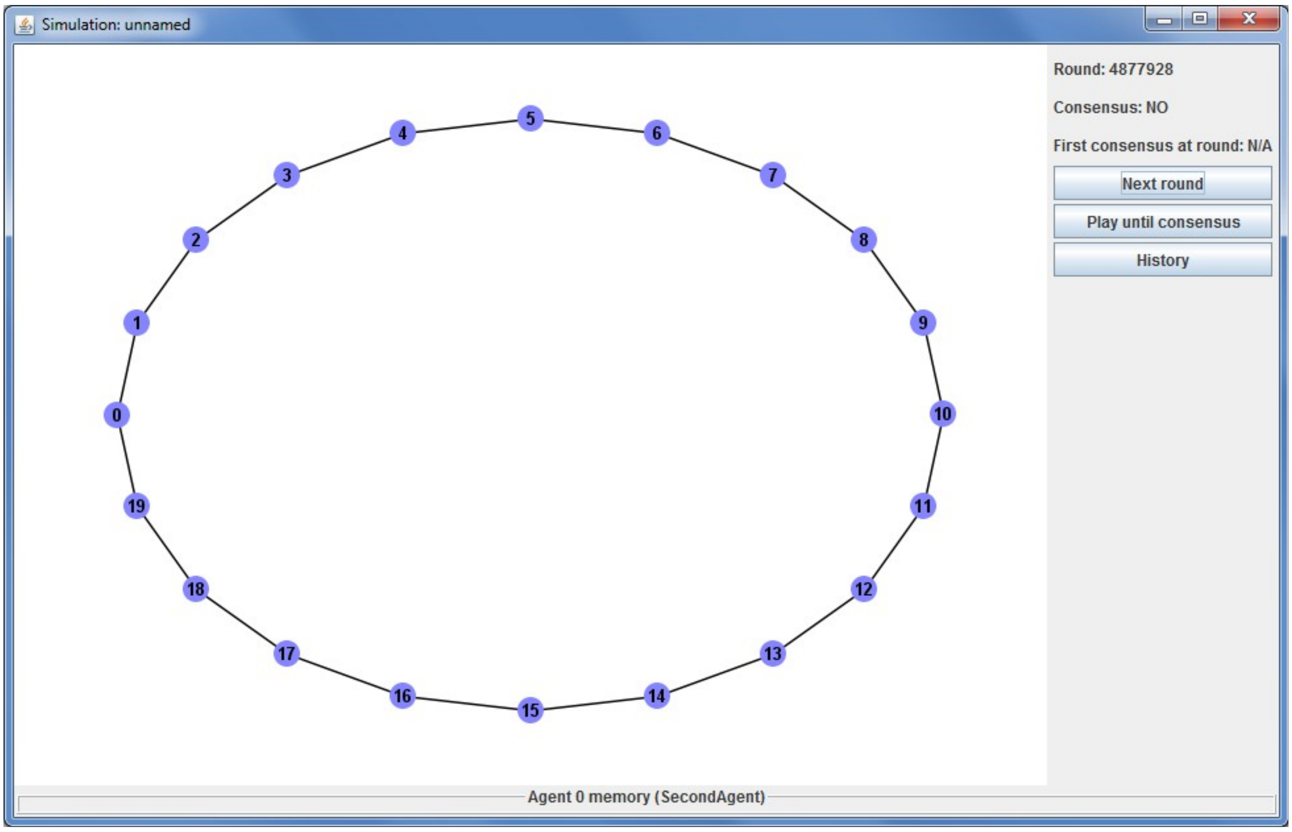
Round: 4877927

Consensus: NO

First consensus at round: N/A

Agent 0 memory (SecondAgent)

Figure 5.6: Deterministic agents in stuck game at round k



Round: 4877928

Consensus: NO

First consensus at round: N/A

Agent 0 memory (SecondAgent)

Figure 5.7: Deterministic agents in stuck game at round k+1

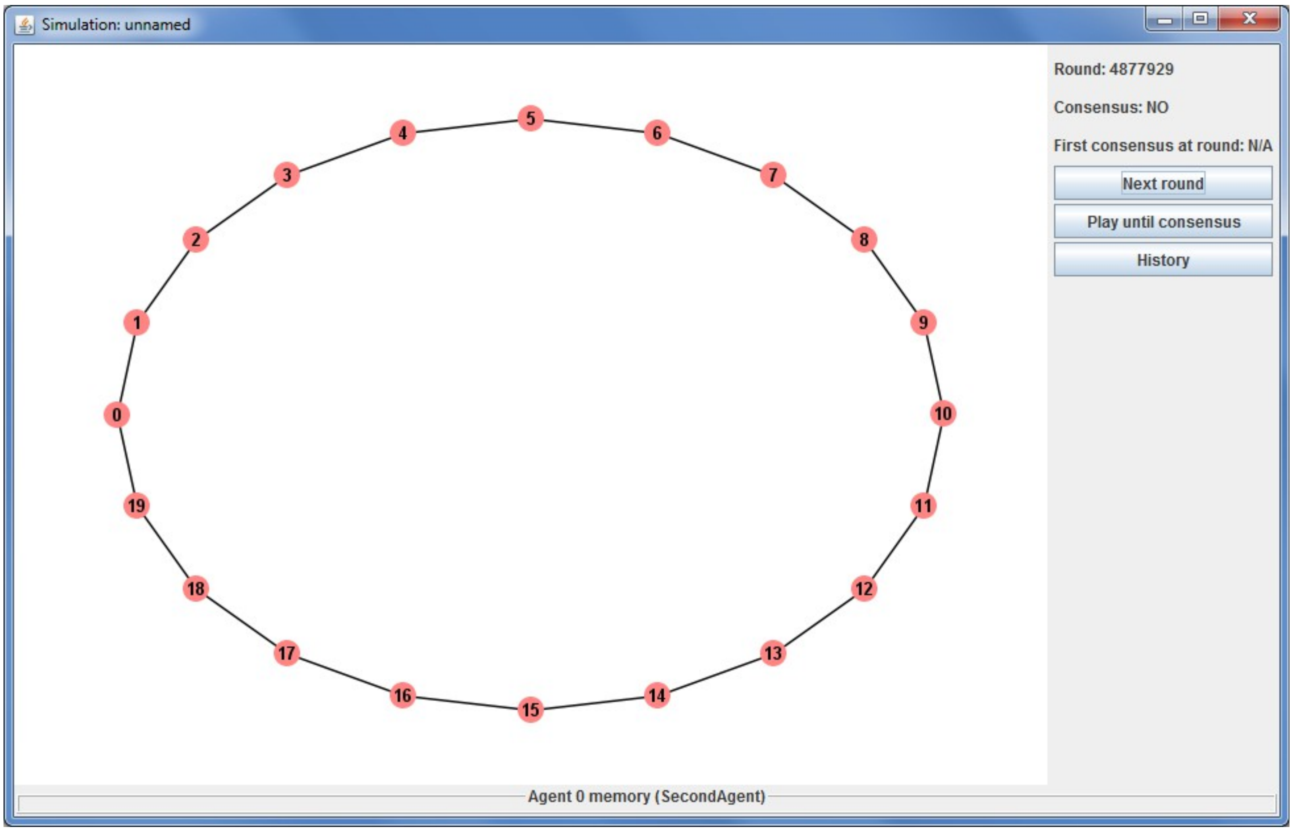


Figure 5.7: Deterministic agents in stuck game at round k+2

5.3.2 Strong interconnections of GUI components

Many design problems were faced when designing any-network simulation. Being over 3 times as large as circle-simulation, its code was more difficult to manage and split properly into packages and classes. It was refactored many times. The back-end of the simulation and all tools are relatively simple, often grouped into 1-3 classes with clear methods and no more than a few hundreds lines of code. However, the front-end brought many problems how different views and functionality should be integrated. The initial idea assumed that a whole any-network simulation will be provided in a single window with four different major views. To simplify the interactions between these four major components, the simulation was split into four independent windows (one for each component) with no interactions between existing windows (i.e. change in one window does not affect any others). Although each of these windows have some common features, they approach them slightly different. For example, each of them displays a network, however in network creator a network is modified frequently, hence it provides options do disable anti-aliasing to speed up performance, while in other components anti-aliasing is always enabled. Other components can use buffers to decrease the time spent on painting the network. Table below presents the main differences and similarities of the four major components:

	creator	initialisation	simulation	history
can modify network	yes	no	no	no
selection	single	multiple	single	no
highlighting	single	multiple	single + multiple	multiple
nodes colouring	yes	no	yes	yes

5.3.3 Code repetition versus code reusage

The design problems described in the above section resulted in interesting problems on balancing between code reusage and code repetitions. On one side, these four major components try to be independent of others, on the other hand, they all provide similar features. For this purpose an abstract drawable pane has been implemented. Its main functionality is to draw a network using buffers for edges and labels (only nodes' colours are painted unless the window size changes and buffers have to be updated). It contains four abstract methods so each of initialisation window, simulation window and history window can specify selected node, infected nodes and nodes colours.

5.3.4 Visibility setting in any-network simulation

The initial simulation assumed agents' performance analysis depending on visibility in the network. This feature (visibility) was implemented in circle simulation. Each agent was given an array with references to other agents where the middle reference was this agent. Agents could use a knowledge of the network structure. They know that in this array an agent at index x have only two neighbours – those at indices x-1 and x+1. Providing this functionality is trivial when implementing the simulation and it is trivial to be used by agents.

However, in case of any-network simulation, this is not so easy to achieve. Network is no longer represented as an array but as nodes, where each node has a list of nodes adjacent to it. Many questions arise on how to design this feature. Agents have to be able to access their 2nd (and further) degree neighbours so the interface via which they access other agents have to provide a functionality to get further neighbours. Most important problem is that traversing algorithms mark nodes as visited, finalised etc. while they traverse. This functionality should not be implemented on the simulation level – each agent should be able to choose the best approach to iterate over visible agents, an approach that is most suitable for its behaviour and algorithms it uses. Hence, it is not sufficient to add a method AgentDelegate[] getNeighbours() to AgentDelegate class. The main difference between circle simulation and any-network simulation is that now agents also need information about edges, only nodes are not enough.

There are a few possible approaches to this problem, each brings different difficulties. In each of them agents have to be provided with a subnetwork of nodes, but the question is what nodes should provide:

- Nodes contain fields for marking the node as visited or finalised. Each agent would have its own copy of a subnetwork it sees, so if both agent 1 and 2 can see agent 3, if agent 1 marks 3 as visited, it will not affect agent 2 in any way. But what would happen if an agent would like to assign for example heuristics to its neighbours? Node structure provided by the simulation would not provide this functionality and agent would be unable to do so.
- Nodes contain nothing else except the list of neighbours and a way to differ two nodes (it could be an id, but should not be the same as this given by the simulation, so agents cannot abuse it; other way would be to let agents compare Java references). Agents would take a responsibility of rebuilding this given subnetwork into a one that meets their needs with their own representation of a node. It would mean that agents have to provide both their own node structure and an algorithm to build their own subnetwork, which would not be too time efficient.

- Nodes contain a dictionary – agents could put there whatever they like, modify it and remove via a few simple methods such as put, get and remove. Agents could put into a dictionary a boolean value (such as visited) or a number (such as last raised flag) or even a list containing the history of that agent. Hence the dictionary elements should be of class Objects what would require agents to do casting and will potentially increase the amount of code for more complex agents.

All these three approaches bring extra effort on both implementing the simulation and implementing the agents, which affects their simplicity – this would require more time to implement a simple and successful agent. Last approaches seem to be the most promising, however any-network simulation does not provide visibility feature at the moment.

6 Experimental results

This chapter presents some of the implemented agents and their performance across range of various networks and game settings. It presents a comparison of various behaviours that have been tested on the following networks:

- Ring-20
  Classic circle network
  20 agents
  2 flags

- Fully-connected-20
  Fully connected graph
  20 agents
  20 flags

- Full-tree-5-3
  Full tree with 5 children and 3 levels
  31 agents
  2 flags

- Grid-7-7
  7 by 7 grid network
  49 agents
  2 flags

- Hex-7-7-e
  7 by 7 hex network with extra edges
  45 agents
  4 flags

- Star-20
  Classic star network
  20 agents
  2 flags

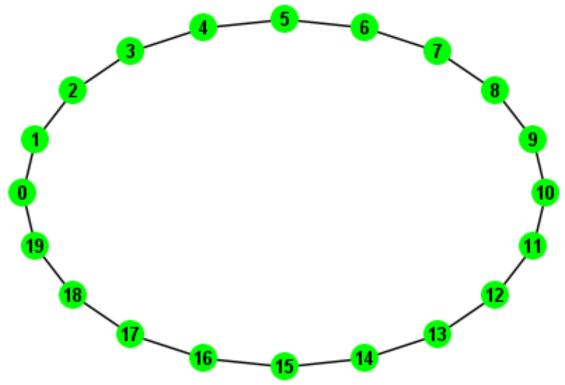


Figure 6.1: Network “Ring-20”

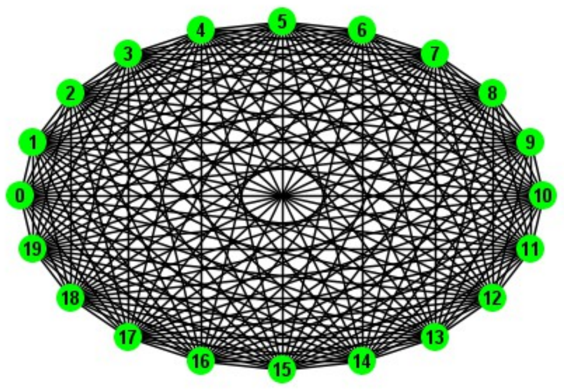


Figure 6.2: Network “Fully-connected-20”

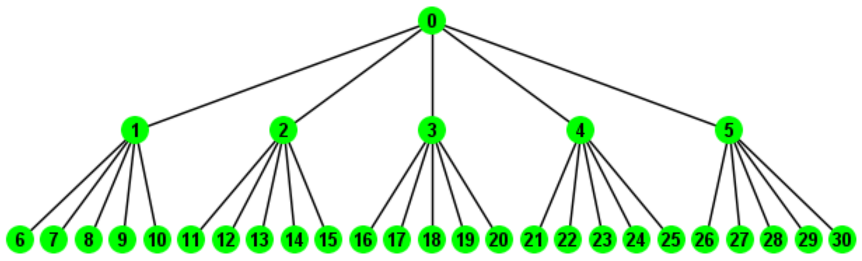


Figure 6.3: Network “Full-tree-5-3”

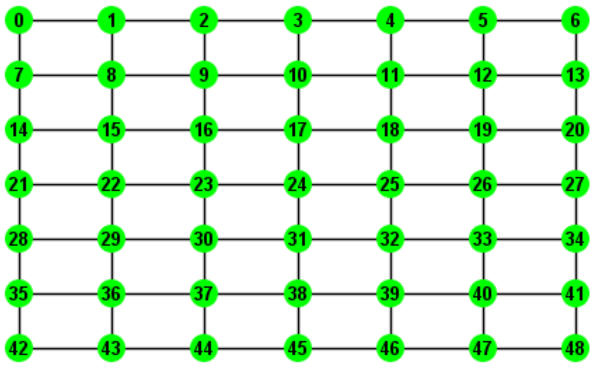


Figure 6.4: Network “Grid-7-7”

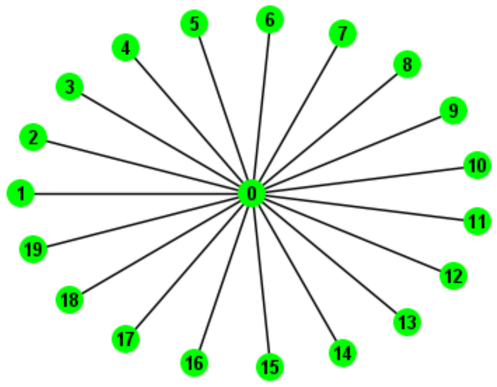


Figure 6.6: Network “Star-20”

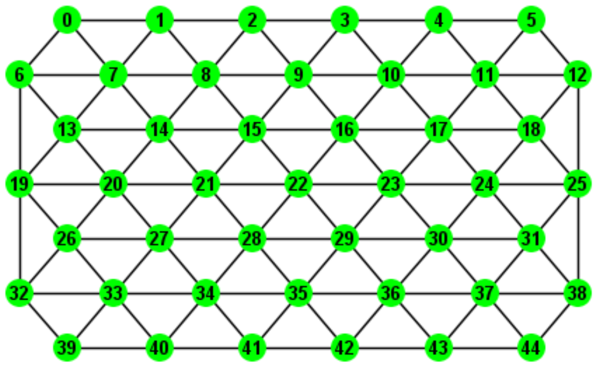


Figure 6.5: Network “Hex-7-7-e”

Each experiment consisted of 100,000 games played with the same settings, unless stated otherwise.

6.1 ThirdAgent

As described previously in chapter 5.3.1, deterministic agents are unable to achieve consensus. ThirdAgent was the first implemented agent (for circle-simulation) that had success rate 100%. Its implementation for any-network simulation is named ThirdAgent2F2N, as it plays with only two flags and exactly two neighbours (if there are more – they are ignored). Its reasonable behaviour is defined as follows:

```
if both neighbours raised red flag
    raise blue
if both neighbours raised blue flag
    raise red
otherwise, raise random flag with 50% probability each
```

Algorithm 6.1: ThirdAgent's reasonable behaviour

However, the agent does not raise a chosen flag immediately. It has a flexible chance of reasonable behaviour, that is initially set to 100%. If agent decides to raise different flag than in the previous round, this chance drops by 10%, while if it is about to raise the same flag, the chance increases by 10%. If check against the chance comes negative, agent raises opposite flag to its choice and the chance is reset back to 100%.

Game played on ring network with 20 such agents ended with consensus achieved on average in 76.4 rounds.

The plot below shows the average number of rounds for ring network with number of agents varying between 4 and 100. This confirm the expectations that with the increase of the network's size (and no change in connectivity), differentiation becomes more difficult.

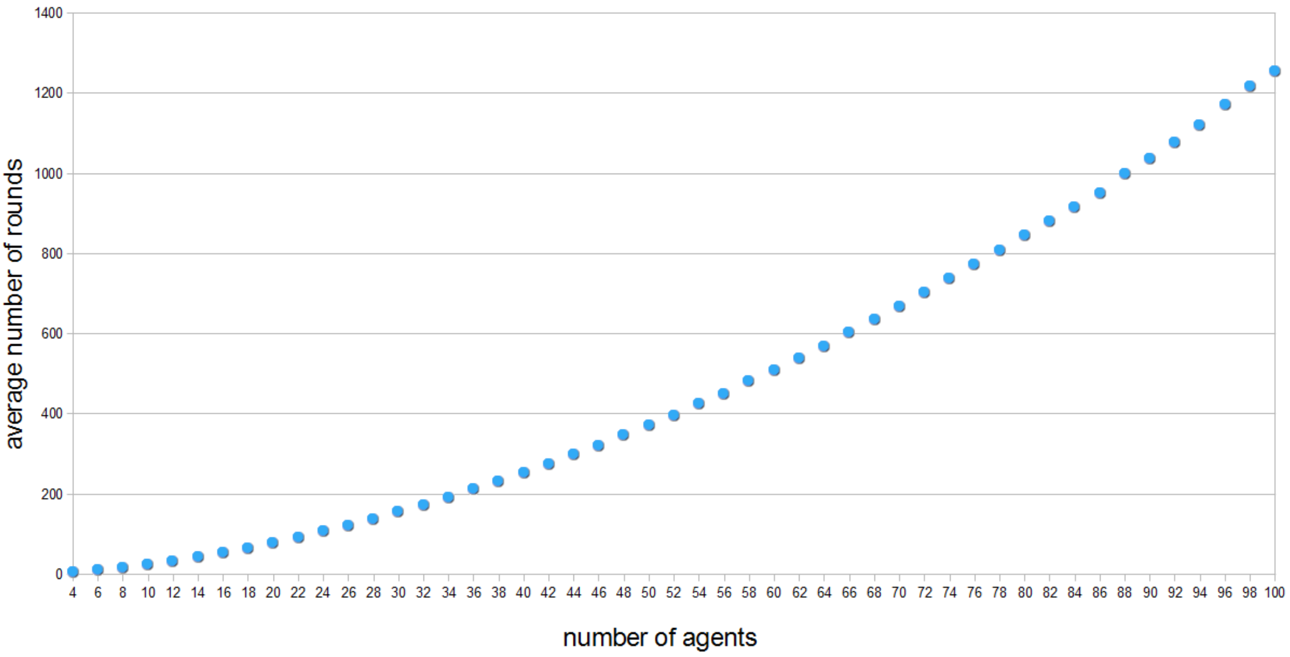


Figure 6.6: Plot of average number of rounds versus number of agents (on ring network)

Interesting question is how the function that modifies the chance affects the agent's performance? Tests on eight extra agents have been performed, where each agent modified the chance differently, as shown in the table below:

Agent class	Increase chance	Decrease chance
ThirdAgent2F2Nc1	+ 5%	- 5%
ThirdAgent2F2Nc2	+ 20%	- 20%
ThirdAgent2F2Nc3	+ 25%	- 25%
ThirdAgent2F2Nc4	+ 33.(3)%	- 33.(3)%
ThirdAgent2F2Nc5	+ 50%	- 50%
ThirdAgent2F2Nc6	* 1.1	/ 1.1
ThirdAgent2F2Nc7	* 1.5	/ 1.5
ThirdAgent2F2Nc8	* 2	/ 2

Tests have been performed on ring network with 20, 30, 40 and 50 agents. In each setting, agents have played 100,000 games and achieved consensus in the average number of rounds presented in the table below:

	20 agents	30 agents	40 agents	50 agents
ThirdAgent2F2N	76.4	153.5	252.0	370.9
ThirdAgent2F2Nc1	82.7	164.6	270.5	396.8
ThirdAgent2F2Nc2	73.1	147.9	243.4	360.8
ThirdAgent2F2Nc3	73.3	147.3	245.6	365.0
ThirdAgent2F2Nc4	74.7	152.5	252.1	377.8
ThirdAgent2F2Nc5	82.4	169.6	286.8	428.6
ThirdAgent2F2Nc6	79.2	158.2	259.4	381.6
ThirdAgent2F2Nc7	76.8	157.5	262.6	392.5
ThirdAgent2F2Nc8	88.3	185.7	314.9	477.7

Agents c2 and c3, that modified the chance by 20% and 25% respectively have achieved better results in all tests, compared to the original agent that modified the chance by 10%. Special attention requires agent c4 that achieved better results in games with 20 and 30 agents, similar results in games with 40 agents and worse results in games with 50 agents. This may suggest that to achieve the best results, chance modifier should depend on the size of the network – this information however is not know to the agents and cannot be used by them.

6.2 LCFAgentND

This agent, whose name stands for “Least Common Flag Non-Deterministic Agent”, is an extension of ThirdAgent so it is able to play with any number of flags and neighbours. Its reasonable behaviour is defined as follows:

```
if there is local differentiation, choose the same flag as last round
otherwise, choose with equal probability from among those flags, that were raised the least number of times among neighbours
```

Algorithm 6.2: LCFAgentND's reasonable behaviour

As an example, consider 3 agents A, B and C playing with 4 flags and:

- agent A has 3 neighbours who raised flags 0, 1, 2
- agent B has 2 neighbours who raised flags 0, 3
- agent C has 4 neighbours who raised flags 0, 1, 2, 3

Then:

- agent A will choose flag 3
- agent B will choose flag 1 or 2
- agent C will choose flag 0, 1, 2 or 3

However, similarly to ThirdAgent, chosen flag is not raised immediately, but there is also a chance of reasonable behaviour. If check against it comes negative, agent raises the same flag as it did in the previous round.

Ring network

This agent should be equivalent to ThirdAgent when playing with 2 flags in ring network, however tests have shown otherwise. Two experiments have been performed – both played on 20-agent ring networks, where in the first case, agent modified its chance in 10% steps, while in the second case – in 20% steps.

First experiment resulted in an average of 64.0 rounds (compared to 76.4 rounds for ThirdAgent2F2N). Second experiment resulted in an average of 66.0 rounds (compared to 73.1 rounds for ThirdAgent2F2Nc2).

These two experiments show that not only these agents are different, but also that chance modify function affects their performance in different ways. A reason of their different behaviours in case of ring networks remain an open question.

Various networks

Further experiments have been performed on aforementioned networks, results are presented in the table below. Success rate represents the number of games that did not end with consensus within 1000 rounds limit, while average is calculated only from successful games.

	Success rate	Average number of rounds
Ring-20	100.0%	64.0
Fully-connected-20	100.0%	6.3
Full-tree-5-3	10.0%	5.8
Grid-7-7	75.5%	22.0
Hex-7-7-e	98.9%	37.0
Star-20	100.0%	3.3

Very low success rate (10% for tree) in a few of the networks indicates that these agents occasionally get stuck. GUI observations revealed that they are simply too stubborn. Consider a stuck game presented on the Figure 6.7. Only agents 0, 1 and 5 did not achieve local differentiation. Agents 1 and 5 have 5 neighbours that raise blue and only 1 that raised red – so this is reasonable to adjust to blue ones and raise red flag. However, for agent 0 the least common flag is red too, as it has more blue neighbours than red ones. All three agents' choices are not affected by the chance.

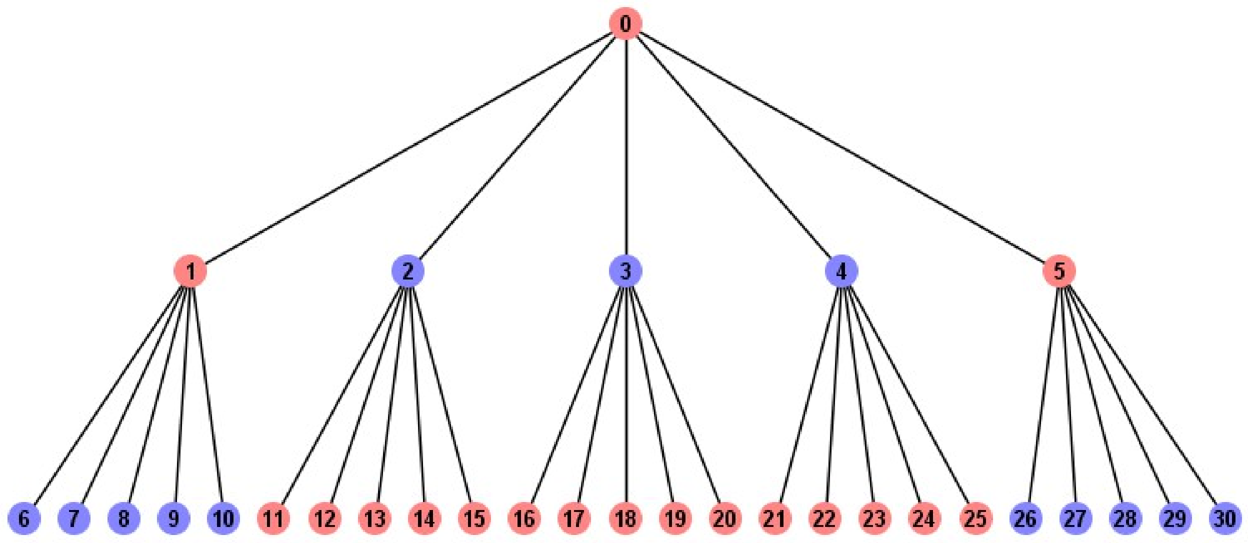


Figure 6.7: LCF agents in stuck game on full-tree-5-3 network

Further work

Possible upgrade of this agent includes modification of the chance in case where local differentiation is not achieved, and no agent (neighbours and the current agent) did not change their flags compared to the previous round. This will require an agent to remember flags raised by its neighbours in last round.

6.3 WeightedAgent2

The idea of weighted agent is to get rid of the LCFAgent's chance of reasonable behaviour and provide a uniformed way of decision making mechanism. The agent assigns weights to available flags and then chooses a random flag from among them. For example, if the game is played with two flags and agent assigned weights 2 and 1 to the flags, it means that it is twice more probable that it will raise a flag with weight 2 than the other one.

Initial implementation did not prevent an agent from making random choice even if local differentiation was achieved, which has been fixed in WeightedAgent2.

The important question is how to translate flags raised by agents into weights? The agent should prefer the flag that was raised the least amount of times among neighbours. Also, no flag should have weight 0, so game does not switch between all-red and all-blue as described in chapter 5.3.1. The function that has been used is defined as follows:

```
weight of flag X:
total number of neighbours – number of neighbours that raised X in last round + 1
```

Algorithm 6.3: Calculating a weight of a flag

So if agent has 3 neighbours who raised flags red, red and blue:

- weight of blue = 3 - 1 + 1 = 3
- weight of red = 3 - 2 + 1= 2

This meets the requirement that the least common flag has the highest weight.

Table below presents the results of experiments on 6 used networks. Games that did not end with consensus within 1000 rounds were considered failed as in the case of LCFAgentND.

	Success rate	Average number of rounds
Ring-20	100.0%	106.4
Fully-connected-20	0.2%	499.7
Full-tree-5-3	100.0%	79.6
Grid-7-7	2.6%	504.9
Hex-7-7-e	0.0%	N/A
Star-20	100.0%	7.9

Compared to LCFAgentND, weighted agent performance is much worse – not only in terms of average number of rounds, but most importantly in the success rate. However it is worth to notice that weighted agent achieved 100.0% success rate on a tree network (compared to 10.0% achieved by LCFAgentND).

Why is this agent so inefficient? One of the possible reasons may be a function for defining weights of flags. Consider what happens if agent has 6 neighbours (common in hex network), who raised flags: red, red, blue, blue, green, yellow.

- weight of red = 6 - 2 + 1 = 5
- weight of blue = 6 - 2 + 1 = 5
- weight of green = 6 - 1 + 1 = 6
- weight of yellow = 6 - 1 + 1 = 6

This gives 27.2% chance of raising green/yellow flag and 22.7% for raising red/blue flag. These values are so close, that in this case the agent raises random (with equal probability) flag in fact. Green/yellow flags are not preferred more enough than red/blue. Possible upgrade of this agent includes changing the weight function so that there are larger differences in weights between these flags that were raised different number of times.

6.4 Infections

Implemented simulation system allows to play coordination games with agents of various behaviours. What would happen if one of the agents behaved differently and for example was raising random flag? How does it affect others' performance? Tests have shown that a few random agents within a network allow deterministic agents to achieve much higher success rates (even up to 100%). Random agents were included in consensus checking. Other agents did not try to identify them in any way.

A series of experiments have been performed on six aforementioned networks. In each network, agent with id 1 was replaced by random agent, who was raising a random flag with equal probability. In one series of experiments all other agents were of class LCFAgentND, while in the second series – of class WeightedAgent2. Results are presented below:

LCFAgentND with single RandomAgent (id 1)		
	Success rate	Average number of rounds
Ring-20	100.0%	98.7
Fully-connected-20	100.0%	26.5
Full-tree-5-3	17.4%	12.7
Grid-7-7	85.4%	29.0
Hex-7-7-e	>99.9%	48.9
Star-20	100.0%	4.3

WeightedAgent2 with single RandomAgent (id 1)		
	Success rate	Average number of rounds
Ring-20	99.4%	197.2
Fully-connected-20	0.1%	559.2
Full-tree-5-3	99.9%	152.5
Grid-7-7	0.6%	494.8
Hex-7-7-e	0.0%	N/A
Star-20	100.0%	10.0

In each experiment agents have achieved consensus in more rounds than they did before infected agent. This confirms expectations that differentiation problem is harder to achieve with infections or just if not all agents have clever behaviour. In some cases, introduction of random agents has resulted in decreased success rate – the reason for this is that rounds limit was set relatively low – hence it should be consider within a mistake margin.

Special attention require full tree network and grid network played by LCF agent, where success rates have increase from 10.0% to 17.4% and from 75.5% to 85.4% respectively. This indicates that a little bit of randomness comes helpful in solving agent's stubbornness and determinism.

7 Conclusions

This project was an invaluable way of gaining experience, especially in fields such as Java Reflection API and designing and implementing large systems. Common functionality of potentially different components required clever use of inheritance, which in result required extra attention to access modifiers.

Produced simulation software has met most of the initial requirements and included also many extra not initially predicted features. It provides effective ways of both observing the autonomous agent's behaviour and measuring its performance in series of multiple tests. Finally, the project has brought up a lot of ideas for future work.

7.1 Further work

Visibility settings in any-network simulation

As already described in Implementation chapter, visibility in any-network simulation has not been implemented and is always limited to one. Agent can see their neighbours, but do not know even whether some of them are connected. As this feature may affect agents, it may be required to implement it as a separate simulation. Also simulation GUI will have to be upgraded to be able to show agents' neighbourhoods.

Spreading the infection

One of the project's initial ideas was to allow infections to spread. This would have to be controlled by the simulation that would replace a chosen agent with an agent of other class. Special care should be taken to ensure that all references to the old agent have been removed and replaced by reference to the new agent. It may require to extend communication between simulation and agents - changes would possibly affect AbstractAgent and AgentInfo classes but all previous implementations of AbstractAgent should not be affected.

But when the infection should spread to adjacent agents? It could be provided to the simulation as an average number of rounds N. After each round, for each agent adjacent to infection, simulation would randomise with probability 1 / N whether this agent should become infected.

Possible extra features also include stopping the game (as failed) if number of infections reaches certain threshold.

Leaders

Concept of leaders was another of the project's initial ideas. It assumed that some agents are able to tell others what flag to raise in the next round. This will in fact require extra communication mechanism for agents. But except these potential implementation problems, it is still unclear whether the concept of leaders is worth testing.

If there is a single leader, it makes no sense for an agent not to listen to it – so it will always behave as told by the leader. In this case part of the network visible to the leader will always be stable and will never change, while the game will proceed on the other part of the network. This can be currently simulated by adding “constant” agents into the game – who always raise the same flag.

If there are multiple leaders, it wood be a good idea if they could communicate. But this would result in coordination problem among nodes in the network – which is an original problem. If leaders could not communicate, all extra effort would be put onto agents, who also had to decide which leader to listen to (assuming that they receive requests from multiple leaders).

Interesting question is if a leader could be infected? It is highly possible that agent will quickly find it out due to the fact that leader requests an agent to raise the same flag as its neighbour.

Reversing network differentiation into network colouring

One of the circle simulation's features was the ability to present a network differentiation as network colouring by introducing a state. In any-network simulation the background of the displayed image would have to be divided into regions that would be coloured as states. This feature may need to be limited only to networks that do not contain edges intersections, in which case an algorithm for translating graph into map could be used to divide a background into regions. Otherwise, it may be impossible to divide the background into continuous regions. The main aim is to allow a human observer to easily see which parts of the network are not yet stable.

Improve graph colouring algorithm

Implemented graph colouring algorithm, that is used to find the minimal required number of flags, does not find an optimal solution. If it was upgraded, it might result in wait periods for the user while the algorithm works. Another option would be to present a user with both algorithms (quick or optimal) and ask which one should be used. This could be also defined in the settings.

Graph “untie'ing” algorithm

This report presents an effective way of finding if the graph can be coloured with 4 colours or less. This however leads to edges intersection detection and the algorithm fails if there is any, even though it could be rearranged. Similarly to Planarity game[6], where a user is asked to rearrange a graph so no two edges intersect, this process could be automated. However, the efficiency of such algorithm is unknown and requires further research.

Tool for multiple testing of a single agent

It has turned out that observing why an agent has made certain decisions is challenging despite simulation GUI and Agent Memory Accessor available. A programmer may be unable to realise that agent's behaviour is different than desired. This tool would simulate only local neighbourhood of the given agent. Decisions of its neighbours would be in fact provided by the tool. It might be required that a few rounds are played for each test (if e.g. an agent reasons based on the history of a few last rounds). Multiple tests would allow to negate a possible mistake due to element of randomness that is usually included in agent's behaviours.

8 Bibliography

1. M. Kearns, S. Suri, N. Montfort (2006)
   "An Experimental Study of the Coloring Problem on Human Subject Networks"
   Science vol. 313
   doi: 10.1126/science.1127207
2. K. Chaudhuri, F. Chung, M. S. Jamall (2008)
   "A Network Coloring Game"
   Workshop on Network and Economics vol. 5385
3. S. Judd, M. Kearns, Y. Vorobeychik (2010)
   "Behavioral dynamics and influence in networked coloring and consensus"
   PNAS vol. 107 no. 34
   doi: 10.1073/pnas.1001280107
4. R. Olfati-Saber, R. M. Murray (2004)
   "Consensus Problems in Networks of Agents with Switching Topology and Time-Delays"
   IEEE transactions on automatic control vol. 49 no. 9
   doi: 10.1109/TAC.2004.834113
5. R. Pagliari, M. E. Yildiz, S. Kirti, K. A. Morgansen, T. Javidi, A. Scaglione
   "A Simple and Scalable Algorithm for Alignment in Broadcast Networks"
   IEEE Journal on Selected Areas in Communications vol. 28 no. 7
   doi: 10.1109/JSAC.2010.100923
6. J. Tantalo (2006)
   "Planarity" game
   www.planarity.net
7. E.L. Lawler (1976)
   "A note on the complexity of the chromatic number problem"
   Information Processing Letters 5 (3): 66–67
   doi: 10.1016/0020-0190(76)90065-X
8. A. Björklund, T. Husfeldt, M. Koivisto (2009)
   "Set partitioning via inclusion–exclusion"
   SIAM Journal on Computing 39 (2): 546–563
   doi: 10.1137/070683933
9. R. Beigel, D. Eppstein (2005)
   "3-coloring in time O(1.3289^n)"
   Journal of Algorithms 54 (2): 168–204
   doi: 10.1016/j.jalgor.2004.06.008
10. J. M. Byskov (2004)
    "Enumerating maximal independent sets with applications to graph colouring"
    Operations Research Letters 32 (6): 547–556
    doi: 10.1016/j.orl.2004.03.002