Department of Computer Science

Submitted in part fulfilment for the degree of BEng.

A GPU-Accelerated stigmergic BCMM swarm

Naums Mogers

June 3, 2015

This document is an extract from the original report. For the Introduction, Literature Review, full Design, Evaluation and Conclusion sections contact Naums via naums.mogers@gmail.com

Abstract

Multiple studies describe the efficiency of swarm systems in nature, such as ant foraging and termite nest building. These systems are simple, robust and fault-tolerant; their models are successfully applied to solving various problems in mathematics and computer science. This report proposes a stigmergic swarm intelligence system based on a model of an ant colony. The system is shown to be effective at mapping the maze.

Agents of the proposed swarm are augmented using the Binary Correlation Matrix Memory (BCMM), which is used to navigate in an unknown heterogeneous environment. The agents are able to learn and recognize different types of objects in an unsupervised fashion using the trial and error method.

The system is evaluated using a GPU-accelerated simulator, which is also developed within this project. The simulator takes advantage of GPU parallelization by simulating each agent on a separate GPU processor.

An evaluation of the swarm intelligence system concludes with an optimal swarm density per maze square; BCMM is shown to implement quick learning with a compact input data encoding. Finally, the simulator performance is evaluated under varied workloads.

Contents

Design

1.1 Problem statement

1.1.1 The maze

The maze conforms to the following rules:

- It consists of squares, which are bigger than boids.
- A wall fills the whole square.
- There are multiple types of walls and passages, which are recognized by agent sensors differently. The objects can be oriented North, East, South or West – agent sensors perceive the same object in different orientations differently, i.e. as different object types.
- All passages are penetrable; however, some passages belong to the class called 'swamps'. Much like their real-life prototypes, swamps can hinder boid locomotion and are thus to be avoided.

1.1.2 Agents

Each agent has the following 'virtual hardware':

- 5 *'object analysers':*
  - An object analyser returns a single integer representing the object (wall or passage) it is sensing.
  - Object analysers are pointed North, East, South, West and down (below the agent).
  - These sensors do not rotate with an agent, i.e. the northern sensor always points north.
  - The downside sensor is used to detect swamps.
- *Absolute positioning system* returns an exact absolute location of the agent as a floating-point number. The unit is maze squares.
- *Swarm positioning system* gives access to exact absolute locations of all other agents in the swarm.

1.1.3 Time domain

The execution is discrete and split in 'time steps', where each step consists of the following actions:

1. Sensor data is generated by the simulator.
2. Agents generate an action based on sensor data.
3. Environment response is generated by the simulator.

1.2 Swarm intelligence

1.2.1 System requirements

In order to implement a true swarm intelligence system, the following restrictions are imposed:

- No centralized control is to be used. As discussed previously, it is an expensive feature, which is difficult to make scalable and fault-tolerant.
- Swarm individuals must be driven by simple programming, and only short-term future should be planned explicitly. This introduces natural fault-tolerance – less potential points of failure – and, if the system were to be implemented in hardware, this requirement would lower the costs.
- Swarm individuals must reason only within a local context. This includes:
    - Limited sensing ability – this allows to save on both computational complexity and sensor costs.
- Local planning – an agent should not be aware of the programming of the whole swarm; it should only be concerned with itself. This reduces the computational load.
    - Agent is allowed to know positions and velocity vectors of other agents to be able to calculate Reynolds' Rules vector. As discussed previously, this ability is observed simple species in nature; hence, it is not expensive.
- Agent communication protocol should be as simple as possible: the system should not require any routing, buffering or other networking methodologies.

To summarize, the resulting system must resemble a product of natural evolution, i.e. be simple, economic and effective in its domain.

1.2.2 System design

The resulting system bears strong resemblance to an ant colony:

- Agents are 'foraging': they scout the maze looking for 'food'.
- 'Food' represents unexplored squares. This reflects the primary goal of the swarm – to map the maze, i.e. visit all unexplored squares.

Pheromone-related logic

Each agent has an attractant pheromone level *(LA_{boid}):*

- Agent attractant pheromone level is increased when an agent encounters an unexplored square, but leaves it behind unexplored. This may happen when multiple unexplored squares are available: the agent chooses only one and the rest remain unexplored.
- Agent attractant pheromone level is increased proportionally to the number of unexplored paths left behind. For example, if an agent enters junction with three

unexplored paths and explores only one of them, its attractant pheromone level is increased by two.

$$LA^{i}_{agent} = LA^{i}_{agent} + n\_paths^{x,y}_{unexplored} \qquad \text{(Equation 1)}$$

Where *x* and *y* are maze node coordinates and *i* is agent's unique ID in the swarm.

- Agent attractant pheromone level is decreased each time step by the constant value Agent Pheromone Dissolution Coefficient *(D_{agent})*.

Each maze node can get two types of pheromones disposed onto it.

- The attractant pheromone: when an agent enters a node, it increases node attractant pheromone level *(LA^{node})* by its own attractant pheromone level *(D_{agent})*.

$$LA^{x,y}_{node} = LA^{x,y}_{node} + LA^{i}_{agent} \qquad \text{(Equation 2)}$$

- The marker pheromone: when an agent enters a node, it marks it with its unique pheromone *(LM_{node})*. The marker pheromone is used to indicate that the agent has visited the node recently, i.e. that LA^{node} is still up-to-date in relation to D_{agent}. The agent does not increase LA^{node} if the node still has a high level of agent's marker pheromone.

Both node pheromones dissolute over time:

- The information that is represented by the attractant pheromones gets outdated as paths are explored, so there is no need to keep high pheromone levels.
- LM_{node} must be big only for some time after the agent has visited the node – afterwards it should dissolute.

Choosing between paths of equal attraction

A separate logic is necessary for the following cases:

- Multiple unexplored paths are available.
- All available paths are explored and multiple paths share the same LA^{node}.

The first problem is solved using Reynolds' Flocking rules described in Section **Error! Reference source not found.**:

- The direction, which correlates with the average velocity of the swarm, is attractive; thus, swarm maintains the density of the swarm on the level, which is necessary for stigmergy to be efficient.
- The agent is to avoid proximity with other agents, so that the swarm is distributed across the area efficiently.
- The direction towards the centre of the maze is attractive; thus, the swarm does not spread out.

These three rules are combined in a single vector; the agent chooses the unexplored path that is aligned with Reynolds' vector to a maximum degree.

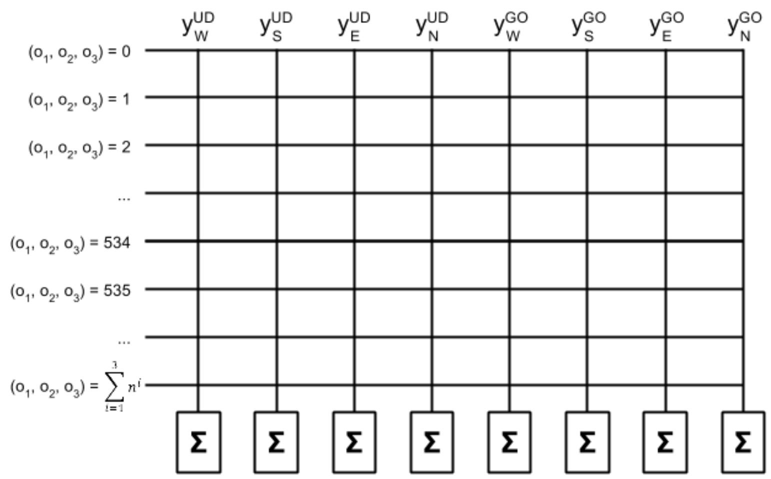

Figure 1: agent BCMM model
(where o_i is an object with index i, and n is a number of object types)

The problem of choosing one of the explored paths of the same attraction is solved using randomness: an agent chooses a random direction, which it follows for a certain number of time steps. This approach allows avoiding local minima, i.e. agent stagnation.

Synchronous approach

Out of the two parallelized ACO algorithm implementation strategies proposed by Bullnheimer et al [1], which are described in the section X (*removed from the web version*), this project implements the synchronous approach, i.e. device and host exchange with data (routes, sensor readings) on each time step. The reason for this decision is following. The study of Bullnheimer et al considers an abstract form of ACO, while this project is concerned with its embodied implementation, where the agent's action causes a reaction from the environment. Accelerating the environment response is out of the scope of this project, so it is computed on the host side. As the environment response depends on the actions of an agent, they have to be transferred from the device to the host on each cycle. On the other hand, as agent behaviour is influenced by the environment response, it is also necessary to transfer the environment response back to the device on each cycle.

1.3 Neural network

Each agent has its own neural network, which is used to classify objects into three groups:

- GO class: the objects of this class are penetrable and should be used as passages.
- NO-GO class: the objects of this class should not be used as passages.
- UD (Undesirable) class: the objects of this class are penetrable, but undesirable. A GO class path is more preferable; a NO-GO class is less preferable.

```
1. Get sensor readings for northern, eastern, southern and western
squares and order them in clockwise direction: (on, oe, os, ow).
2. Choose the first reading on as $o_0$.
3. Generate a trigram of three readings by 'rotating' the pointer in
clockwise direction starting from the current reading $o_0$, i.e. when $o_0$ = $o_n$,
return (on, oe, os).
4. Choose the next $o_0$ in the clockwise direction.
5. Repeat steps 3 and 4 until all four readings have been assigned to $o_0$.
```

Listing 1: Sliding Window algorithm (pseudocode)

The task requires that agents do not know in advance, what object types can be encountered in the maze; hence, this information is acquired by the trial and error method. The agent tries penetrating the unknown object – if successful, the object is considered a passage; otherwise, it is a wall. If the object is a passage, the downside sensor is used to detect a swamp. Finally, the neural network is updated with the acquired knowledge.

It is worth clarifying why the use of neural network is justified at all – after all, an agent could have a simple continuous array, where indices correspond to different object types, and values contain object classes. The justification is following:

- Noise can be present in the input data; therefore, the data structure must be noise-tolerant.
- The logic behind swamps is non-trivial – they are to be used as paths only if no normal passages are available. The logic could be implemented in agent reasoning; however, it is possible to reduce the computational load by associating the class of a swamp to a particular context and storing this link in memory. In other words, the swamp can be 'remembered' as a GO path in a context where no true GO paths are available; and as a NO-GO path if there is at least one true GO path.
    - This is similar to how human mind works: certain objects or events are associated (i.e. remembered) with some property (e.g. good or bad) in memory so that this information does not have to be derived again.

The neural network type is Binary Correlation Matrix Memory (BCMM). As discussed in section X *(removed from the web version)*, BCMM is easy to implement and analyse; it is quick and robust at both learning and recalling.

A model of BCMM used in this project is shown in Figure 1 and described below.

1.3.1 Input layer

There are $\sum_{i=1}^{3} n^i$ input neurons, where n is a number of object types that can be encountered. The way this number of input neurons is derived is described below.

It should be noted, that objects have four possible orientations, which are perceived as different object types by agent sensors.

1.3.2 Pre-processing and input encoding

Trigram generation

The sensor readings are pre-processed using a Sliding Window algorithm. Different variations of the algorithm are defined by Alberg et al [2] and Keogh et al [3]; the algorithm used in this project is shown in Listing 1. The algorithm returns four trigrams: (o_n, o_e, o_s), (o_e, o_s, o_w), (o_s, o_w, o_n) and (o_w, o_n, o_e), where o_j is the object in the respective square, e.g. o_s is an object in the southern square relative to the agent.

Trigram encoding

Each trigram is converted to a single unique numeric value using the *scalarize* function in the following equation:

$$scalarize(o_0, o_1, o_2) = o_0 + n * o_1 + n^2 * o_2 \qquad \text{(Equation 3)}$$

As o_i are positive and not bigger than n, *scalarize* is similar to a base conversion algorithm, with the target base being equal to n.

Input data encoding

The BCMM contains a row (an input neuron) for each possible trigram that can be generated by the Sliding Window algorithm. The input neurons to be activated are selected using the *scalarize* function – the values it returns are the neuron indices.

In order to calculate how many rows (input neurons) will BCMM contain, a maximum value of *scalarize* needs to be calculated:

$$scalarize(n, n, n) = n + n * n + n^2 * n = \sum_{i=1}^{3} n^i \qquad \text{(Equation 4)}$$

The input vector thus conforms to the following regular expression:

$$0^a\,1\,0^b\,1\,0^c\,1\,0^d\,1\,0^e \qquad \text{(Equation 5)}$$

Where 0_a notation means "repeat 0 a times", $a >= 0$, $b >= 0$, $c >= 0$, $d >= 0$, $e >= 0$, and the size of the vector is $\sum_{i=1}^{3} n^i$.

The input is thus always complete – the number of ones stays the same.

Motivation and analysis of the trigram approach

Splitting the sensor readings in trigrams yields the following advantages:

- 'Contextuality' is preserved: information about an object is recorded in association with its neighbours.
- Generalization:
    - If two observed patterns partially coincide, the information about one is relevant to the other.

    - Same pattern can be encountered in four variations (oriented North, East, South and West). When a set of sensor readings is pre-processed, the orientation of readings is lost – the trigrams are converted to scalar values, which will not reflect the trigram order. For example, the trigram (o_n, o_e, o_s) can have bigger or smaller value than (o_e, o_s, o_w) depending on what values are associated with the objects. Trigram encoding allows storing the information about a pattern only once and associating it with an actual orientation again during the post-processing stage.

1.3.3 Output layer and initial weights

The output layer contains eight neurons, which can be functionally split in two groups:

- Four GO neurons return one if the respective object belongs to the GO class and zero if it belongs to the NO-GO class.
- Four UD neurons return one if the respective object belongs to the UD (Undesirable) class and zero otherwise.

Initial weights

- Weights of all inputs to GO neurons are set to one – the objects are thus treated as passages until it is confirmed that they are not penetrable. Such initial weights ensure that agents are motivated to try all squares, and therefore train neural network themselves.
- Weights of all inputs to UD neurons are set to zero – agents set them to one only when swamps are detected by the downside sensors.

Output value range

While this structure allows for two-class membership, the learning algorithm outlined in Section 1.3.6 ensures the following:

- An object can belong to both GO and UD classes – this means that the object was confirmed to be a swamp, but it might be the only available passage.
- An object cannot be in both NO-GO and UD classes – if the object was marked as a NO-GO class, it is definitely a wall, not a swamp.

1.3.4 Thresholding function

The thresholding functions can be first analysed in terms of their limitations:

- L-max requires the output vector to have the same number of ones for any input. In the context of this problem, it is clearly not the case: there might be various numbers of walls and passages surrounding different squares.
- The Willshaw threshold is a good match as the input is always complete. Additionally, there are always only four ones in the input, so the Willshaw threshold automatically transforms into the Fixed Willshaw threshold (4).

```
1. Pass a set of readings R to the neural network. Based on its
output, select a path and try to enter the goal square.
2. If the goal square is a swamp, set the four UD weights of the
corresponding side of R to one.
3. If the goal square is a normal passage, check if any of the other
passages in R are known to be swamps. If some are, change their class
to NOT-UD and NO-GO.
4. If the agent has run into a wall, set the four GO weights of the
corresponding side of R to zero.
```

Listing 2: BCMM learning algorithm (pseudocode)

The Fixed Willshaw seems to meet the requirements; however, due to the input encoding it will produce a high error rate. Consider a pattern A, which split in four trigrams, some of which are also included in a pattern B. If the agent confirms that the western square in pattern A is a wall, it will update the weights of all four trigrams of the pattern A, including those that are included in pattern B. When pattern B is encountered, the sum for one of the walls will be smaller than 4 and hence it will not pass the threshold. The network will report that the object is a wall, while it might be a passage.

To deal with the problem of trigrams appearing in multiple patterns the *logical OR* function is used as a threshold function in this BCMM: the output neuron will be activated if at least one input equals one.

- As described in Section 1.3.6, when the object is confirmed to be a wall, all weights of the corresponding output neuron are set to zero by the learning algorithm. This means that if the sum of weights is higher than zero, but smaller than 4, some of the trigrams were reused in other patterns. The logical OR thresholding function thus allows for 3 errors per output without any effect on the correct neuron activation. Presence of 4 errors means that the neural network became saturated.

1.3.5 Post-processing

As mentioned in Section 1.3.2, during the pre-processing stage the readings set loses association with orientation. It is thus necessary to restore this information during post-processing.

Select the trigram *S* with the smallest value.

1. Get index *d* (delta) of the trigram S in the initial readings set: $(o_n, o_e, o_s) = 0$, $(o_e, o_s, o_w) = 1$, $(o_s, o_w, o_n) = 2$ and $(o_w, o_n, o_e) = 3$.
2. 'Rotate' the output of the neural network by *d* clockwise. For example, if d = 1 the output of the neuron y_n is relevant to the eastern square, neuron y_e – to the southern square, y_s – to the western square etc.

In other words, when neural network is updated, the algorithm assumes that the first object of the smallest trigram points north. The position of the smallest trigram in the initial readings set is a *skew value*, applying which the output we can revert the skew.

1.3.6 Learning algorithm

BCMM of this project has no formal training phase – the learning is unsupervised and happens on the go. The learning algorithm is outlined in pseudocode in Listing 1.

- The reasoning behind step three is that if an agent discovers a normal passage, the swamps are not just undesirable; they should be treated as walls so that the normal passage is always preferred.

1.3.7 BCMM usage model

The interaction with this BCMM happens in the following way:

- The four sensor readings are pre-processed and fed to the neural network.
- The neural network returns classes of the four surrounding objects.
- The agent chooses one of GO or UD class objects and tries penetrating it.
- Based on the trial result (the agent detects a passage, a wall or a swamp), the neural network weights are updated for the previous sensor readings.

Implementation

An example execution of the simulator developed within this project is shown in Figure 2. GUI elements are described in Section 2.2.1.

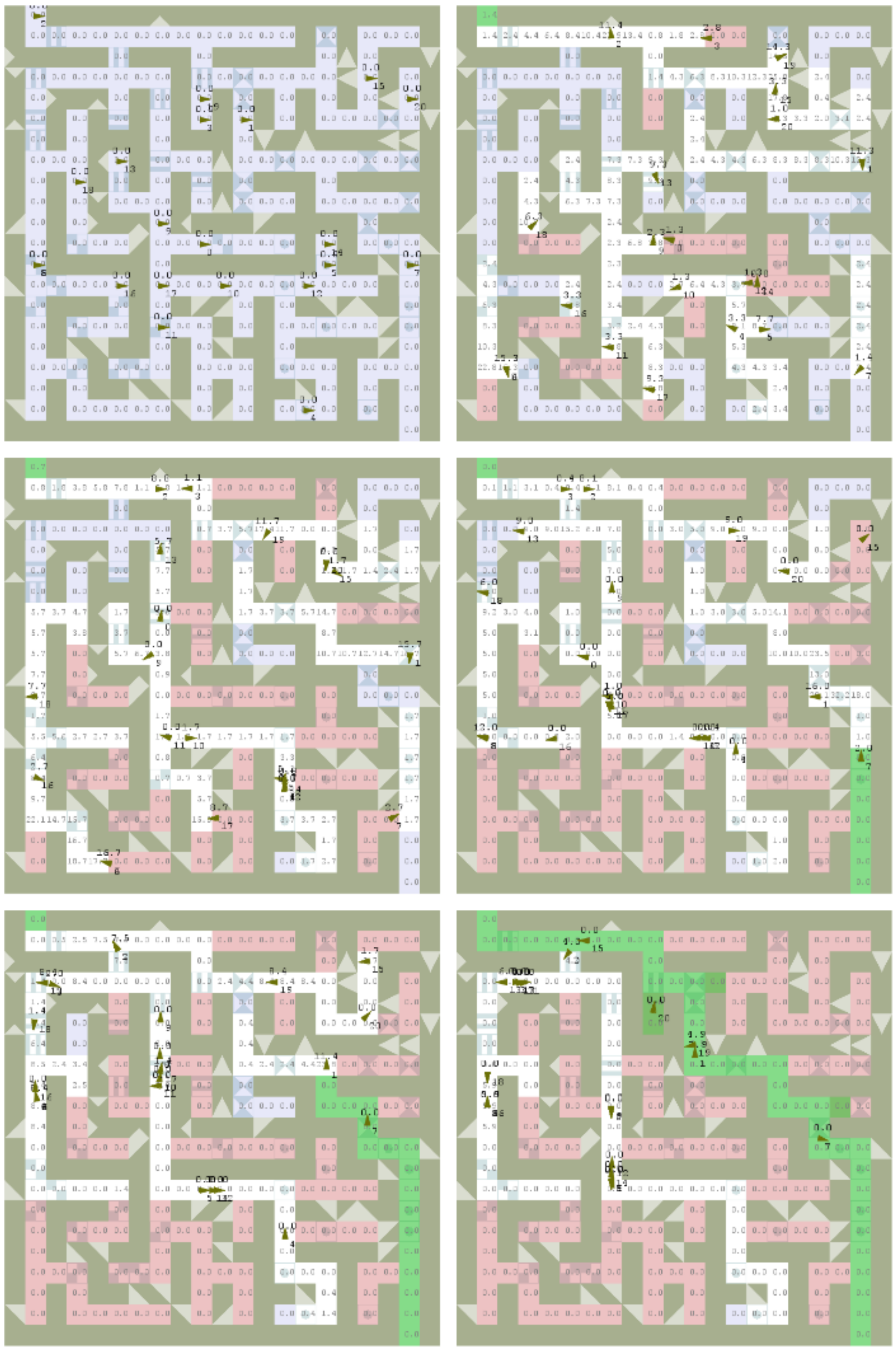

Figure 2: an example of the simulation developed within this project

2.1 Project structure

2.1.1 Functional structure

The software developed within this project can be functionally split in two parts:

- Simulator – this is the framework of the project, which is responsible for:
    - Initializing OpenCL
    - Generating the maze and the initial flock configuration
    - Iteratively executing agent code
    - Iteratively generating environment response, which includes collision processing and sensor readings generation
    - Rendering the simulation
    - Saving the animation as a set of images or a video file
- Agent Artificial Intelligence (agent AI)

The simulator is the biggest part of the project and consists of approximately 1900 lines of Python code and 70 lines of OpenCL code as reported by the CLOC utility [4]. Agent AI (Artificial Intelligence) is implemented in 1610 lines of OpenCL code as reported by the CLOC utility. The separation between the two parts is relative though as the simulator is also responsible for configuring, initializing and executing agent AI.

2.1.2 Code structure

The project consists of a set of Python modules *(*.py)*, OpenCL source *(*.cl)* and header *(*.h)* files:

- *main.py:* as the name suggests it is the root module of the project. Its responsibilities include running other modules, managing their inputs and outputs and recording the time it takes to run the modules.
- *configs.py* is a list of assignments.
    - Both simulator and agent AI are highly configurable and this module contains all the settings. The settings include simulation duration, number of agents, size of the maze, coefficient values for simulation and agent AI equations, GUI settings, and framerate.
    - The data that Python and OpenCL components exchange with is unstructured, so this module contains the symbols used by both sides to access specific memory parts.
- *maze.py* provides support for generating, storing, accessing and analysing the maze-related information. This includes:
    - Providing information about square types
    - Generating, serializing and de-serializing the maze
    - Converting coordinates between square numbers and pixels
    - Responding to the questions about an instance of the maze such as:

    - Is a wall on the edge of the maze?
    - Is a coordinate outside of the maze?
    - What squares surround a particular square?
- *agents.py* provides support for generating, storing, accessing, serializing and de-serializing the state of the swarm. The state data includes the list of agents, their positions and velocity; there is a separate state for each time step of the simulation.
- *simulator.py* is the second most-important module of the project. It is responsible for the main simulation loop, which iteratively runs the agent AI, processes agent collisions with walls and generates sensor readings for each agent.
- *opencl_computations.py* is a wrapper for all PyOpenCL-related code. It is responsible for initializing OpenCL, building the OpenCL program, preparing the host and device memories, transferring data between the host and the device, running the kernels, and post-processing the data.
- *collision_detection.py* encapsulates all collision-related functionality. It allows checking whether an agent has 'teleported' partially or fully inside a wall. If overlapping is detected, an agent is shifted to a position where only a collision occurs, but no overlapping happens. This produces realistic collision processing: when agents bump into a wall, they slide along it.
- *renderer.py* is run after the simulation has finished, i.e. when swarm states for all time steps are computed. The following components are rendered:
    - The maze: different types of walls and passages are represented by different patterns, which are defined in *maze.py*.
    - Square statuses: a square can be either unexplored (blue), explored (white), leading to a dead-end (red) or leading to an exit (green).
    - Boids: a boid are represented by an isosceles triangle, which is oriented according to boid's velocity. Boid ID is rendered below its body.
    - Pheromone levels: attractant pheromone level of a node is rendered in the centre of the node square. Attractant pheromone level of a boid is rendered above its body.
- *displayer.py:* responsibility of this module is to abstract saving the rendered frame sequence from the rest of the project. Depending on *configs.py*, Displayer generates either a set of images – one of each frame – or a single video file using the FFmpeg command-line utility.
- *main.cl* is the main OpenCL source file, which contains the following:
    - The list of macros and corresponding conversion classifiers. On building of the program, Python code traverses through the list and replaces conversion classifiers with literals – this is the build-time data transfer, which is discussed in more detail in section 2.2.7. The macros are used throughout all OpenCL source files.
    - Implementations of all pre-processing, initialization, updating, simulation and agent AI kernels. The kernels are wrappers for the functionality implemented in *agents.cl* and *simulation.cl*; the kernels themselves are responsible for managing

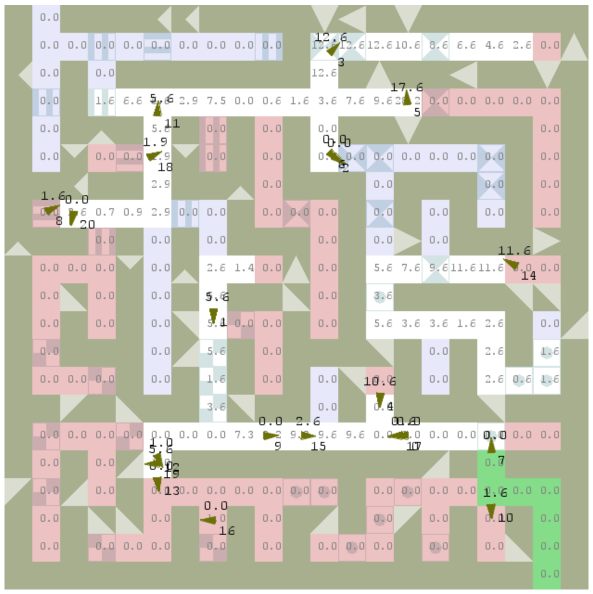
Figure 3: Simulator GUI

and synchronizing work-items and swapping the data between global, local and private memories for the purposes of synchronization and optimization.

- *agents.cl* contains agent AI including all pre-processing and reasoning. *agents.h* is a header for this source file.
- *simulation.cl* contains the GPU-accelerated components of the simulation: node pheromone dissolution, traction, inertia and maze bound collision detection.
- *utilities.cl* contains helper functions that used throughout other OpenCL source files. The functions include atomic float addition in global memory for scalars and vectors, array fetching from global and local memories and conversion between *floats* and *ints*. *utilities.h* is a header for this source file.

2.2 Simulator

2.2.1 GUI

An example of simulator GUI is shown in Figure 3.

- The dark squares represent walls.
- The list squares represent passages.
- The squares with patterns are exotic walls and passages.
- Numbers in squares show node attractant pheromone levels.
- Unexplored passages are blue.
- Explored passages are white.
- Dead-ends are red.
- The path to one of the exits (target path) is green.
- The dark triangles are agents.

```
global_generated_flocks[INDEX_IN_ALL_FLOCKS((int)(*global_iteration),
                                             global_id,
                                             BOID_VEL_VAR)]
```

Listing 4: an example of complex data structure
implemented using pre-processor macros (OpenCL)

- The number above an agent shows agent attractant pheromone level, the number below is agent's ID.

2.2.2 Objects

The three most important objects in this project are:

- *Maze* object – it contains a 2D array of maze squares, where each square is represented by the *Square* object containing a tuple *(shape, orientation)*. Both elements are to be used with corresponding enumerators, which allows for high readability of the code as demonstrated in Listing 3.
- *Boid* object stores orientation of a boid, which is calculated on each cycle of the simulation loop from boid's velocity. The object also provides a static method *init_boid*, which returns a tuple (*position*, *velocity*, *pheromone_level*), where *position* is a Numpy array of random coordinates, *velocity* is a zero Numpy array and *pheromone_level* equals zero. *Pheromone_level* represents boid's attractant pheromone level. The tuple type is a continuous Numpy array.
    - It is important to note that the *Boid* object does not store the above-mentioned tuple.
- *Flock* class stores the state of a flock. It contains two data structures:
    - A list of *Boid* objects.
    - A Numpy array of tuples (*position*, *velocity*, *pheromone_level*) representing boid states.

  The list is used only by Python modules to store boid orientations, while the Numpy array is used both by Python and by OpenCL modules. The reason behind this design is that OpenCL supports transferring only flat continuous memory structures.
  It is worth noting that the simulator stores an array of *Flock* objects, one per each time step of the simulation.

```
maze[x].append(Square(Wall.normal, Orientation.north))
 if isinstance(maze[x][y].shape, Wall):
    print("The square contains a wall!")
if matrix[x][y].shape == Wall.small_triangular:
    print("The square contains a small triangular wall!")
```

Listing 3: an example of maze node manipulation (Python)

2.2.3 OpenCL arrays

The host allocates on the OpenCL device a set of global arrays, which are used throughout most of the kernels to share data between work-items and devices. Due to OpenCL limitations, the arrays are flat; that is why complex data structures were implemented within this project using pre-processor macros. An example of such structure is shown in Listing 4. Layout for complex structures is defined in Python module *configs.py*; it is passed to OpenCL via build-time memory transfer (see Section 2.2.7).

- *Map* is shared between all agents; it contains such node information as node exploration status, whether the node is dead-end or a part of the target path and node pheromone levels.
- *Agent programmes* are agent memories: they contain agent goals for the next time step, information about previous time step and the randomly chosen direction, which is to be followed for some period.
- *Agent neural networks* are stored as a single array, but each agent has access only to its own network.
- *Generated flocks* store agent positions and velocities.
- *Experiments* store the experimental data used for performance evaluation.

2.2.4 Maze generation

Simulator can operate in two modes: either it reuses the previously generated maze by de-serializing the relative *pickle* file, or it generates a new random maze. Maze generation is implemented in *maze.py* using a randomized version of the Prim's algorithm.

Prim's algorithm is an iterative tree-growing process of finding minimum spanning trees in weighted graphs [5] [6]. Listing 1 outlines Prim's algorithm in pseudocode.

The randomized version of the algorithm is described by Manen et al in the context of visual recognition [7] and by Buck in the context of maze generation [6]. The algorithm operates on an unweighted graph and replaces step 3 in the original algorithm with rule outlined in Listing 6.

In the context of maze generation, the maze can be represented as a connected weighted undirected graph, where graph nodes are maze squares, each node can have up to four edges to horizontal and vertical neighbour squares, and the minimum spanning tree is a pattern of passages that connects all squares of the maze. Listing 5 outlines the pseudocode of Randomized Prim's algorithm that is applied to the maze generation.

After the main body of the maze is completed, simulator adds the following:

- Maze frame – the maze edges are filled with walls.
- Two exits. For each exit an edge wall is selected, which adjoins a passage; the wall is changed to a passage of the type 'exit'.
- 'Exotic' walls and passages. Initially the maze consists of ordinary walls and passages, so at this point the simulator replaces randomly selected walls and passages with alternatives.

```
1. Initialize an empty list of walls W.
2. Fill all squares in the maze with walls.
3. Randomly choose the initial square i from the maze and make it a
passage. 4. Add the walls of i to W.
5. Randomly choose a wall w in W.
6. If the square s behind the wall is not a passage, make s a passage
and add its walls to the list W.
7. Remove w from W.
8. Repeat steaps 4 to 7 until W is empty.
```

Listing 5: Randomized Prim's algorithm in the context of maze generation (pseudocode)

- There are four exotic passages (swamps): *striped*, *crossed*, *tiled* and *dotted*.
- There are four exotic walls: *small_triangular*, *big_triangular*, *forward_slash* and *backward_slash*.
- The exotic wall and passage density is configurable.
- The maze is split into four rectangular quarters, and each quarter contains its own types of exotic walls and passages. This introduces heterogeneity to the maze and allows evaluating how an agent performs in an environment, where an accumulated knowledge is irrelevant. Additionally, this allows testing the knowledge exchange between agents, but this feature was not implemented. It is discussed in more detail in the section X *(removed from the web version)***Error! Reference source not found.**.

Initially all nodes are marked as unexplored; the node pheromone levels equal zero.

2.2.5 Flock initialization

The first state of the flock is generated using the *init_boid* static method of the class *Boid*. The boids are placed in centres of random squares throughout the maze. Initial velocities and pheromone levels equal zero.

2.2.6 OpenCL kernels

Most kernels conform to the stream programming paradigm (see Section X *(removed from the web version)*):

- *Initialization* includes declaration of variables, filtering out excessive work-items and pre-buffering the data from global and local memory (*data gathering*).
    - In agent AI-related kernels, all work-items with global IDs higher than total number of agents are considered excessive and are shut down immediately.

```
3. Choose a random node w in G, which meets all of the following
conditions:
    - w is not in V.
    - w is connected to one of the nodes in V.
```

Listing 6: Third step of the Randomized Prim's algorithm (pseudocode)

```
#define NUMBER_OF_BOIDS    %(number_of_boids)d
#define MAZE_WIDTH         %(maze_width)d
#define MAZE_HEIGHT        %(maze_height)d
#define TRACTION           %(traction)f
```

Listing 7: OpenCL pre-processor macros that are defined during Python runtime (OpenCL)

Excessive work-items are allowed as a trade-off for efficiency: each OpenCL device recommends the work-group size to be a multiple of a particular number (e.g. 64), and it is usually more effective to have nonoperating agents than work-groups of non-recommended sizes.

- o Pre-buffering is used to improve performance: the first work-item in the group fetches the global memory data to the local memory so that other work-items can access it faster. This is implemented using local memory barriers: all work-items halt until the first work-item finishes writing to the local memory.

- *Execution* is a list of calls to the specialized functions in *simulation.cl* and *agents.cl*.
- *Wrap-up* implements the *scatter* component: the data computed by the work-group is transferred from local to global memory by a single work-item.

The list of kernels includes:

- *Reynolds' Rules preprocessors* are used to precompute data for Reynolds' Rule calculation. For example, one of these kernels is executed by N^2 work-items (where N is the number of agents in swarm) to generate avoidance vectors. For each agent N work-items check positions of other agents and compute the vector that allows avoiding collision.
- *Agent AI and primary simulation response kernel* runs the agent AI and simulates traction, intertia and maze bounds collision detection.
- *Secondary simulation response kernel* receives the agents positions, which are altered according to the full simulation response computed by Python modules. The kernel makes these update values accessible to agent AIs in the next time step.

2.2.7 OpenCL data transferring methods

Two types of data transfers between host and device are used in this project:

- *The run-time data transfer:* this method is provided by OpenCL API. It is scheduled using a command queue and returns an event on completion.
- *The build-time data transfer:* this method exploits the fact that the OpenCL program is built during run-time. OpenCL code contains a list of pre-processor macros and corresponding Python formatted keywords (see Listing 7); before building, the program Python replaces the keywords with corresponding values. This is a very quick and simple process, which allows for compatibility of Python and OpenCL modules, and configurability of OpenCL code.

2.2.8 Simulation response

The agents generate impulses, which are used by the simulator to generate agent velocity and new position. The following features are supported by the simulator:

- *Traction* and *inertia* – the agent's velocity in previous time step influences its next velocity vector.
- *Collision detection* is implemented in the following way:
    - This simulator calculates agent orientation and checks if in its initial position agent overlaps with any walls. If it does, then the simulator 'pushes' the agent away from the wall. This allows simulating agent rotation on the same spot.
    - The simulator splits agent impulse vector in points and checks if the agent collides with a wall in any of these points. If collision is detected, the function returns the last point on the impulse vector on which the agent does not collide with anything.
- *Sensor readings generation* – the information about surrounding squares is combined into packets and sent to agent AI.
- *Maze node pheromone dissolution* – the simulator runs the OpenCL kernel, which simulates attractant and marker pheromone dissolution.

2.3 Neural network

The BCMM neural network implementation benefits from the low-level nature of OpenCL:

- The network is stored as an array of bytes, where each byte represents a single row in the Correlation Matrix Memory.
- Each bit contains the binary weight of the corresponding output neuron.
- Scalar values of trigrams are used as indices for the array; thus, recall takes almost no time.
- Weights are updated using bitwise operations: *AND*, *OR*, *XOR* and *shifting*.

2.4 Node status propagation and maze solving

A node can have the following statuses: unexplored, explored, leads to a dead-end (dead-end node), leads to the exit (target node). A simple reasoning is implemented in the agent AI to back-propagate the dead-end and target statuses: "if a junction consists out of only two paths and one of them is a dead-end node or a target node, the respective status is to be assigned to the junction".

In some cases, this reasoning is sufficient to solve the maze. Often it is not sufficient; for these cases, Dijkstra's algorithm [8] for constructing trees of minimum total lengths was implemented in the simulator. On each loop cycle, the algorithm checks whether there are enough explored nodes to construct a path between two exits. When the check is successful, the maze is considered solved. Thus, maze solving can be outsourced to the simulator; however, the more complex task of mapping the maze is always performed by the swarm.

References

[1] B. Bullnheimer, K. Gabriele, and S. Christine, “Parallelization Strategies for the Ant System,” *High Perform. Algorithms Softw. inNonlinear Optim.*, vol. 24, no. 8, 1998.

[2] D. Alberg and Z. Laslo, “Segmenting Big Data Time Series Stream Data,” pp. 2126–2134, 2014.

[3] E. Keogh, S. Chu, D. Hart, and M. Pazzani, “An online algorithm for segmenting time series,” *Proc. 2001 IEEE Int. Conf. Data Min.*, 2001.

[4] A. Danial, “CLOC -- Count Lines of Code,” 2015. [Online]. Available: http://cloc.sourceforge.net/. [Accessed: 27-Apr-2015].

[5] R. C. Prim, “Shortest Connection Networks And Some Generalizations,” *Bell Syst. Tech. J.*, vol. 36, no. 6, pp. 1389–1401, 1957.

[6] J. Buck, “Maze Generation: Prim’s Algorithm,” 2011. [Online]. Available: http://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm. [Accessed: 27-Apr-2015].

[7] S. Manen, M. Guillaumin, and L. Van Gool, “Prime Object Proposals with Randomized Prim’s Algorithm,” *2013 IEEE Int. Conf. Comput. Vis.*, pp. 2536–2543, 2013.

[8] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numer. Math.*, vol. 1, no. 1, pp. 269–271, 1959.