JÜLICH
Forschungszentrum

Supplementary Material to: A Network Simulator for the Estimation of Bandwidth Load and Latency created by Heterogeneous Spiking Neural Networks on Neuromorphic Computing Communication Networks

Version: 3.1 | Date: February 25, 2022

Forschungszentrum Jülich GmbH
ZEA-2 | System Modeling
52425 Jülich

Contents

Abbreviations

ACA	Advanced Computing Architectures
BC	Broadcast
BCF	Broadcast First
BCL	Broadcast Last
ESPR	Enhanced Shortest Path Routing
JSON	JavaScript Object Notation
MC	Multicast
NC	Neuromorphic Computing
NN	Neural Network
RNDC NN	Randomly Connected Neural Network
UC	Unicast
VLSI	Very Large Scale Integration

1 Modelling and Simulation

This folder contains the following files:

- *network_sim.py*
- *netlist_generation.py*
- *hardware_graph.py*
- *sim_log.py*
- *config.ini*

1.1 Concept

The purpose of the simulator is to allow the evaluation of different communication network architectures in order to make a well-considered decision for the design of the physical implementation. We are not (yet) interested in the actual behaviour of the **Neural Network (NN)** or individual neurons, but solely in the performance of a chosen communication infrastructure. Therefore, to simplify the model, the nodes internal behaviour such as neuronal membrane potentials, synaptic behaviours, etc. are not considered and only the movement of a spike from the source node to its destinations is modelled. After the spike reaches its destination it is dropped.

In a **Neuromorphic Computing (NC)** system, the communication packets are generated by neurons firing and transmitting a notion of this event, a so called spike packet, to their connected synapses. As mentioned before, the model does not consider the nodes internals and thus is unable to determine which neuron is firing and has send out a spike. Instead, every neuron is assumed to fire once and the corresponding traffic generated by each neuron firing is calculated. The total network traffic can then be determined by superimposing the traffic generated by each neuron. A consequence of this assumption is that the simulation would only be valid for NN in which all neurons have approximately the same level of activity. At high levels of abstractions, when precise NN configurations are unknown, this is not a problem. However, it is not representative for biological NN in which neurons typically have different levels of activity. To work around this limitation, neurons in the model are assigned a firing rate. During the superimposition, this firing rate is used as a multiplier for the contribution of that neuron.

The final output of the simulation is the amount of spikes going over each link within a given time frame corresponding to the chosen firing rate. For example, if the firing rate of every neuron is set to 1 with a biological average firing rate of 10Hz, the simulation output corresponds to the amount of traffic generated within a tenth of a second of running (not considering the acceleration factor).

1.2 Model Description

The current model is implemented in Python (3.7) and is version 3.1 of the model[1]. The choice for this programming language was made because of the versatility and ease of use as well as compatibility with partner institutes within the **Advanced Computing Architectures (ACA)** project.

[1] The earlier versions were based on the same concept but lacked some additional features and optimizations.

During the simulation, the following steps are performed:

1. Read out the config file and set simulation parameters,
2. Set up neural network,
3. Generate a directed graph, representing the communication network,
4. Map neurons from the NN to a node in the graph,
5. Mimic a (weighted) spike event for each neuron in the NN, sending a spike packet to all connected neurons locations.

The top level module of the simulator is *network_sim.py*. This module reads the *config.ini* which contains the simulation settings, controls the execution of all submodules and writes the results of the simulation in a **JavaScript Object Notation (JSON)** file.

1.2.1 Generation of the Neural Network

The generation of the neural network happens in one of two ways. The connectivity is either determine prior to the simulation by the module *netlist_generation.py* and stored in a python dictionary, referred to as the netlist, or the connectivity is determined on-the-fly, during the simulation itself.

Netlist based simulation

The first approach allows the repetition of a simulation with exactly the same connectivity, however, the memory required by the netlist dictionary, limits the size of the NN to be simulated. As the models purpose is to calculate the spike traffic, the neural network generated in this step is not a fully defined NN and only contains the connectivity information and firing rates of the neurons. Accessing a neuron in the netlist returns a sub-dictionary which includes the firing rate of the particular neuron and a list of connections the neuron has to other neurons. In the *config.ini*, the user can define the type of network that is to be generated. The module includes a couple of different network types, ranging from abstract networks, such as the **Randomly Connected Neural Network (RNDC NN)**, to more defined network structures, such as the ACA project test cases. The module generates the specified type of network netlist based on the given parameters in the *netlist_dict_template*. This dictionary is passed to the netlist generator function and overwrites the default values with the given parameters. The format of the resulting netlist is shown below for an arbitrary exemplary network. This dictionary is then also saved in the *netlists* subfolder of the simulator, as a JSON file, to be used again if desired.

{

"neuron$_0$": {"FR": 1.1, "connected_to": [neuron$_1$, neuron$_4$, neuron$_{12}$, ...]},

"neuron$_1$": {"FR": 1.8, "connected_to": [neuron$_3$, neuron$_5$, neuron$_8$, ...]},

"neuron$_2$": {"FR": 0.3, "connected_to": [neuron$_1$, neuron$_2$, neuron$_{24}$, ...]},

...

...

"neuron$_n$": {"FR": fr_n, "connected_to": [...]}

}

The different types of networks implemented at this point in time, for this type of network generation, and their corresponding input parameters can be found in table S1.1. As the project continues, other network types might be added. If the simulation folder already contains a netlist for the given parameters, the module will recognize this and load the old netlist file instead of creating a new one. If desired, users can also load their own netlist files into the simulator by writing or generating their own netlist file.

Network Type	Input Parameters
Hopfield NN	Number of neurons: n
RNDC NN	Nr. of neurons: n Connectivity probability: epsilon
RNDC NN with fixed out degree	Nr. of neurons: n Out-degree: connections
Two population network - Brunel ACA project testcase	Number of neurons: n Ratio exhibitory neurons: beta Connectivity probability: epsilon
Two population network - Vogels-Abbott ACA project testcase	Number of neurons: n Ratio exhibitory neurons: beta Connectivity probability: epsilon Nr. of input neurons: m
Cortical microcircuit model - Potjans-Diesman ACA project testcase	Network size scaling: scale_factor

Table S1.1: Implemented netlist types.

On-the-Fly network generation

The second approach does not store the connectivity, instead, the connectivity of a neuron is determined when it's being processed and is forgotten straight after. This connectivity is determined with a connection probability, which can be different for different pairs of source and target neurons and are directional, i.e. $p(AconnectstoB) \neq p(BconnectstoA)$. The probability can be set in two different ways, as a "hardware dependend" probability or as a connectivity matrix which contains the statistical information needed to create a NN. In the former scenario, the connection probability is given as a function of the distance between the source and potential target neuron. This probability function is the same for every source neuron, thus resulting in a homogeneous neural network. This way of network generation is explained more in section 1.2.3. The latter option, uses predefined connectivity statistics given as a matrix file. Each row in the connectivity matrix (CM) contain the size of the neuron population and the probabilities that a neuron from the corresponding population connects to a neuron in the population of the corresponding column. This way of network generation is very similar to the netlist based network generation. The difference is the storing of the connectivity in the netlist based approach, making the simulation deterministic, and the mapping of the neurons. In this approach the mapping can only be done on a statistical level and only the type of neuron, i.e. the population name, is stored to the node. In the netlist based approach, the population name and the neurons index within the populations needs to be stored. An example of a matrix file is shown below, the header row, printed in blue, is not included in the actual file, and is added to the example for clarity.

Population	Nr. Neurons	L2/3E	L2/3I	L4E	L4I	L5E	L5I	L6E	L6I	TC
L2/3E	20683	0.1009	0.1346	0.0077	0.0691	0.1004	0.0548	0.0156	0.0364	0
L2/3I	5834	0.1689	0.1371	0.0059	0.0029	0.0622	0.0269	0.0066	0.0011	0
L4E	21915	0.0437	0.0316	0.0497	0.0794	0.0505	0.0257	0.0211	0.0034	0
L4I	5479	0.0818	0.0515	0.135	0.1597	0.0057	0.0022	0.0166	0.0005	0
L5E	4850	0.0323	0.0755	0.0067	0.0033	0.0831	0.06	0.0572	0.0277	0
L5I	1065	0	0	0.0003	0	0.3726	0.3158	0.0197	0.008	0
L6E	14395	0.0076	0.0042	0.0453	0.1057	0.0204	0.0086	0.0396	0.0658	0
L6I	2948	0	0	0	0	0	0	0.2252	0.1443	0
TC	902	0	0	0.0983	0.0619	0	0	0.0512	0.0196	0

1.2.2 A Directed Graph Representing the Communication Network

The model of the communication network is stored as a directed graph. Each node in the graph corresponds to a computational node of the system while the edges represent the links between nodes. In the currently existing NC systems, nodes are connected with bidirectional links, whereas the graph uses directed edges - a problem which can be circumvented with ease by creating 2 separate links between nodes, one for each direction. The choice for these directed edges was made, as this is a better representation of the actual hardware. In many cases, bidirectional links are nothing more than two parallel connections, operating completely independent from each other. By modelling it this way, the possibility that a high load in one direction gets cancelled out by low traffic in the other, is eliminated and a higher level of insight can be reached. If the bidirectional links are not implemented as parallel unidirectional links, the sum over the two edges can be used to get the traffic load on the combined link. The combined number of spikes going over the link is not effected by the chosen approach and thus remains valid.

The graphs are generated as class objects. The class declarations and all related class methods are encoded in the *hardware_graph.py* module. The foundation of this module is the *Network*-class. This class functions a (grand)parent class for all the other types of network, each of which have their own class. This (grand)parent class covers all the basic functions the communication network needs to fulfil, such as adding/removing nodes or edges to/from the graph, returning all the traffic in the network after a simulation run, and resetting the network. The (grand)parent class also contains the more complex methods which are independent of the topologies, such as uniform filling or random mapping of the network (more on this in section 1.2.3), and the Dijkstra routing algorithm. The hierarchical structure of the classes in this module is shown in figure S1.1. Besides the network methods, the network class also contains a node class and an edge class[1]. Instances of these classes are used to construct the graph and store its properties, as well as the results of a simulation run. Table S1.2 shows the data stored in each instance of these objects.

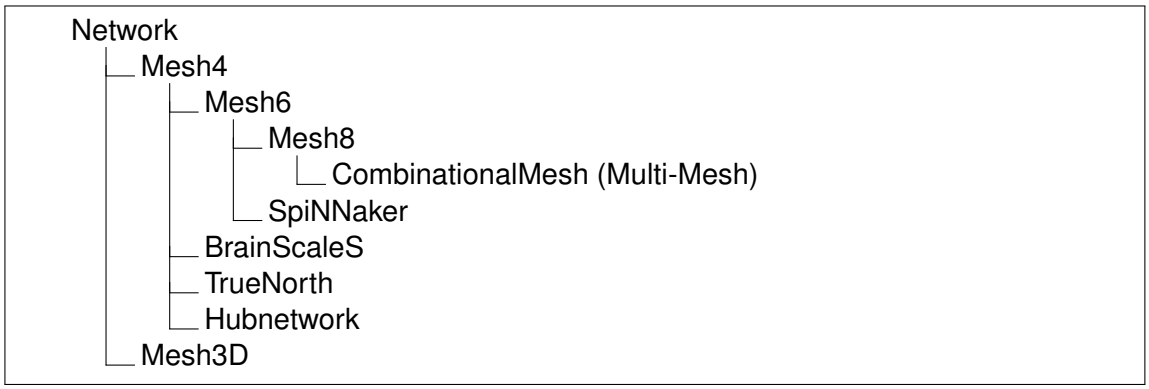


Figure S1.1: Hierachical structure of network classes of the *hardware_graph.py* module.

[1] These are nested classes, not to be mistaken with the subclasses discussed earlier.

Node	
ext_packets_handled	Counts the number of spike packets passed to the node by other nodes
int_packets_handled	Counts the number of spike packets send by the nodes internal computational unit
neurons	A list of neurons mapped to this node
edges	A dictionary with all outgoing edges. {"target node": "Edge object"}

Edge	
weight	A normalized length unit, used to indicate the cost of using this link
packets_handled	The number of packets send over this link

Table S1.2: Overview of the Node- and Edge-classes.

1.2.3 Mapping Neurons to Hardware Graph

In order to model the spike traffic, the location of the neurons in the hardware graph, i .e. which node models which neuron, needs to be known. The determination of these neuron locations is referred to as placement or mapping. A comparison can be made here to the "Place & Route" process performed in **Very Large Scale Integration (VLSI)** design. The mapping algorithm has to designate a location for every neuron[1] in the NN with a given maximum number of neurons per node, NpN, and can be seen as optimization problem. Placing neurons which are connected closer to each other results in a shorter travel distance for the spike and in turn a lower traffic load. Unlike VLSI "Place & Route" algorithms, the mapping does not have to consider constraints like fitting all wires or the driving power of one neuron to the other and even can, in regards to the model, neglect timing constraints. Free of these constraints, the mapping should be much simpler, but the level of connectivity in NN is over 3 orders of magnitude larger than basic VLSI design, which greatly increases the complexity. First attempts to adapt and implement existing VLSI placement algorithms in the first version of the model, turned out to be slow and resulted in only slight improvements of traffic loads.

Currently, the third version only contains some basic mapping algorithm. These algorithms search for an optimized solution, but simply place the neurons randomly on a node or place the neurons in the order they appear in the netlist, on the first available n ode. Obviously, the former does not attempt to solve the optimization problem and will act as a benchmark for other mapping algorithms. The latter on the other hand, sequential placement referred to from here onwards, does reduce traffic load in certain scenarios, even if it might not seem so at first. A requirement for this is that the NN that is used as test case contains populations or clusters of neurons which have a higher level of connectivity to other neurons within the same group. On top of that, the populations and clusters need to be declared in succession in the netlist as well. If both of these requirements are met, sequential placement will place the neurons of the same population in the same general area of the network, as it goes through the nodes in chronological order[2]. While the average traffic reduction of sequential placement compared to the random placement is significant, it is far from an optimized s olution. A slightly larger gain was achieved with the adapted VLSI algorithms, the recursive min-cut algorithm and the quadratic wire length algorithm. Unfortu-

[1] As biological NN inherently possess a great level of redundancy, the mapping algorithm could omit some neurons while the NN remains functional. However, as the goal of the ACA project is to build a NC system for computational neuroscience and exact behaviour is to be investigated, this is not preferred.

[2] In case of the auto-generated meshes, this means the network is filled row-by-row.

nately, the runtime of these algorithms quickly rose up to several hours, even for a relatively small test case of 5000 neurons, making them infeasible to use. Further improvement of the mapping definitely shows potential and is incorporated in the planning. The currently available mapping algorithms are incorporated into the network classes and can be found in the *hardware_graph.py* module. The connection based mapping methods are currently only available for the netlist based NN approach. For the matrix based approach, similar algorithms could be used, however, they would have to be based on the statistical connection probabilities instead of the absolute number of connections. Unfortunately, this adaption has not yet been made in the current version, but might be added to a later version.

Hardware Dependent Neural Network Generation

A alternative to mapping a given NN to the hardware graph, is the generation of a NN in dependence of the hardware graph. The former, i.e. the netlist based and matrix based approaches, have been the main focus during the development of the simulator. Either a NN is given or the properties for a NN are given and a hardware graph with an appropriate size: $NpN \cdot N \geq n$, is generated and the neurons are mapped (efficiently) to the graph. The alternative approach changes dependencies and with it the order of process steps. In this approach, a hardware graph is created with a given size and all nodes are filled with the given number of neurons NpN. However, no connections between the neurons are defined as of yet. In the second step, connections for each neuron are determined at random with a connectivity probability distribution. A uniform distribution, i.e. $p(x) = const.$ results in a NN comparable to the RNDC NN network type discussed in section 1.2.1. Other possibilities are a distance dependent probability distribution, $p(d_{i,j}) = \frac{C}{d_{i,j}}$ where $d_{i,j}$ is the distance between neuron i and j, or a given biologically inspired probability distribution. The benefit of using this approach is the fact that no mapping is required. A good mapping algorithm tries to place connected neurons close to each other, but proves to be complex and time expensive. Here, with the distance dependent probability distribution, neurons which are located close to each other are more likely to be connected, inherently solving the mapping problem. In general, this is also true for biologically inspired probability distribution as these contain distance dependent component as well. The limitation of this approach is that it results in a homogeneous network, all neurons are alike. To create a network in which different groups of neurons have different characteristics, the probability distribution needs to be changed, depending on the node currently being processed. This however, has the undesired effect that the inherent mapping solution might become less efficient. And besides this, it would still not be possible to create the same type of detailed network structure as for example in the Potjans Diesmann test case.

1.2.4 Modelling of the Spike Packets

All the previously described steps are part of the initialization of the model. The next step of the simulation is the emulation of spikes travelling through the network. In the old version, the simulator looped through every node in the network and the routes were calculated for each neuron on that node individually. This step is not significant on its own in regards of computation time, but dominates overall run time, due to the large number of repetitions. So, instead of doing this for each source neuron individually, the second and later versions of the model combine the neurons that are located on the same node. How the neurons are combined depends on the type of casting used and is visualized in figure S1.2.

The simplest case here is **Broadcast (BC)**, in which every neuron communicates with every node, regardless of the neurons it is connected to. As the list of destinations is the same for all neurons, the routes resulting from this destination list is equal for all neurons in a node. Thus, instead of processing each neuron individually and multiplying the resulting routes with the neurons

firing rate, the firing rates off all neurons on the node are accumulated Ⓐ. The routes to all nodes from the current node can be determined once and then weighted by this accumulated firing rate.

The other types of casting require some additional steps to calculate the routes. For each neuron mapped to the current node, the "connected_to" list is read out and every neuron in this list is converted to the node ID on which it is located Ⓑ. From here onwards, this list will be referred to as a destination list of a neuron. When using **Unicast (UC)**, the destination lists are not equivalent for all neurons, but a route from Node_i to Node_j will remain the same and only has to be calculated once. To prevent unwanted repetitions of the same calculation, the destinations list are converted to a weighted list (implemented as Python dictionary) with unique entries Ⓒ_{UC}. Each entry in the destination list of a neuron, adds a value equal to the neurons firing rate to the weight of that entry in the weighted list. Because the routes only depend on the source node and not the actual source neuron, this process can be repeated for all neurons in the current node, resulting in a single weighted list with unique entries for each node Ⓓ_{UC}. With this weighted list, the routes to all destinations can then be calculated once and weighted appropriately.

The local-**Multicast (MC)** case is very similar to UC. However, UC sends a separate spike to each neuron, whereas local-MC sends a separate spike to each node which has at least one neuron it needs to communicate with. If neuron a_i needs to communicate with two neurons, b_j and b_k, located on the same node B, only one spike packet will be send to node B. The local router of node B will then branch off this spike to both neurons. Here, this means that any duplicate nodes in the destination list of the individual neurons can be removed $\text{Ⓒ}_{\text{local-MC}}$. After this, the conversion to a weighted list can be done to combine the destination lists of all neurons in the node $\text{Ⓓ}_{\text{local-MC}}$.

This optimization becomes more complex in case of MC. This is because the branch nodes - nodes at which a spike packet splits off to multiple other nodes - are calculated by superimposing the routes from one neuron to all its destinations and removing any du-

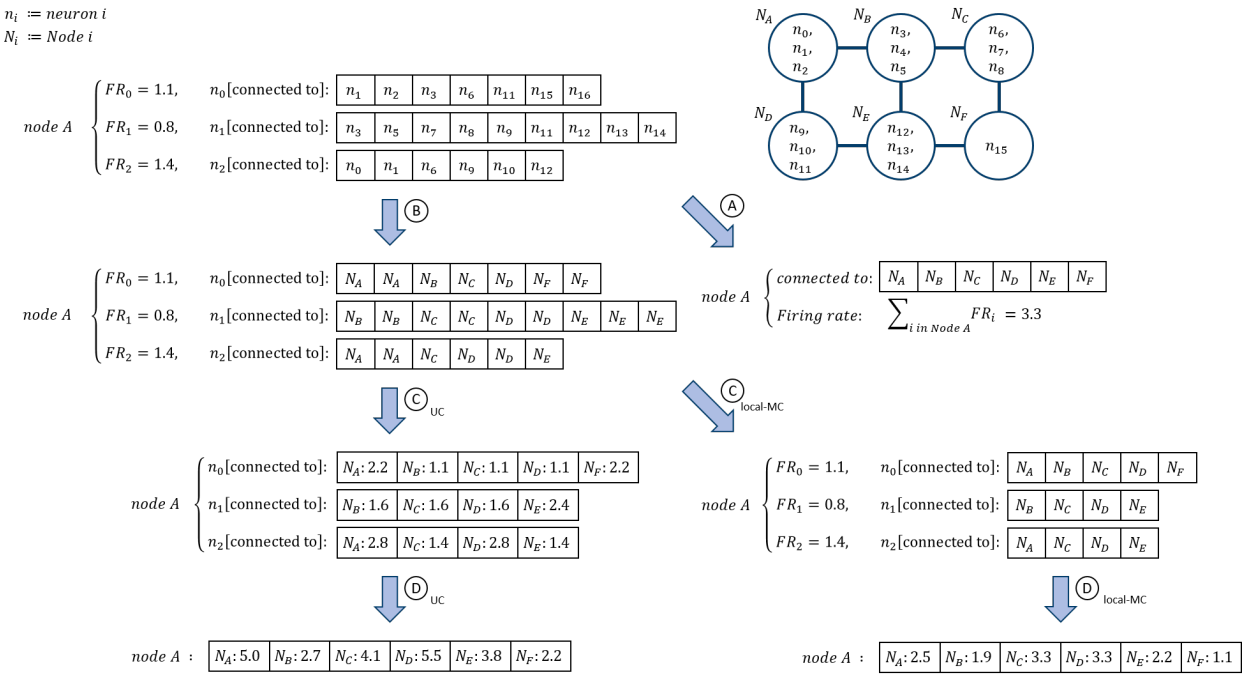


Figure S1.2: Combining the route calculations of neurons to speed up the simulation.

plicate links (Fig. S1.3a). The resulting superimposed route depends on the combination of routes, not just the individual destinations and thus has to be done per neuron individ-ually. This makes it impossible to use the same approach as before. It is still possible to reduce the number of redundant calculations on a lower level in some cases. When the route from node A to node B does not depend on the other destinations (Fig. S1.3b (*DOR*)), this calculation can still be combined for all neurons on a node. Then, only the superimposition of the routes has to be done individually for every neuron.

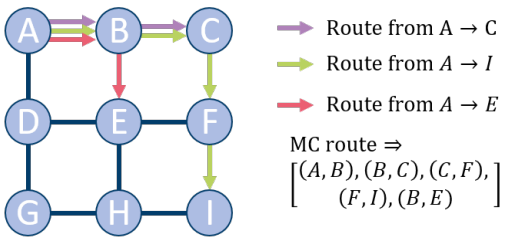
However, some MC routing types, such as **Enhanced Shortest Path Routing (ESPR)**, are optimized for MC and calculate the path from the source to a destination depending on the other destinations as well. This means that the routes from two neurons on node A, to node I are not necessarily the same as shown in figure 1.3b. Unfortunately, this also means that the route calculation has to be done individually for each neuron, as was done in the previous model version[1].

For the Hubnetwork topology, as well as the Combinational-/Multi- Mesh topology, a combination of different casting types have been implemented as well. As the interconnect network of topologies can be divided into multiple levels, different casting types can be used on the different levels. One of the possibilities are the **Broadcast First (BCF)** casting types. Here, the spike is broadcasted over the upper level (after being transported to a hub in case of the Hubnetwork) in the first phase. Then in the second phase, the spikes are communicated from the hubs to their respective destinations by one of the previously described casting types. The second group of additional casting types is the exact opposite of BCF, **Broadcast Last (BCL)**. This group of casting types uses on of the previously described casting types to communicate the spike packets to the hubs located near a target and from these hubs, the spike packets are broadcasted to all nodes surrounding the hub.

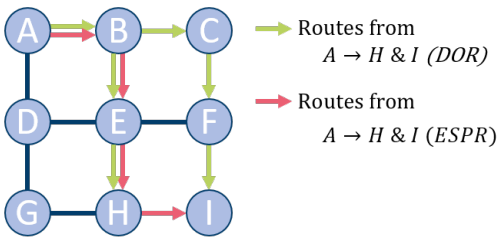
The Path Dictionary

As these different types of casting calculate and combine their routes in slightly different ways, a small adaption was necessary in order for the routing to function properly. In the model, the different routing methods don't return the actual route from a source node to the destination node, but return a 'path' dictionary. The *(key, value)* pairs in this dictionary give a route description in the opposite direction. A value belonging to a key entry in the dictionary indicates the previous node on the route from source to destination. The route

[1] Because of some other improvements, this version will still perform better for MC compared to the old code, but does not reach the same level of speed-up as it did for the other casting types.



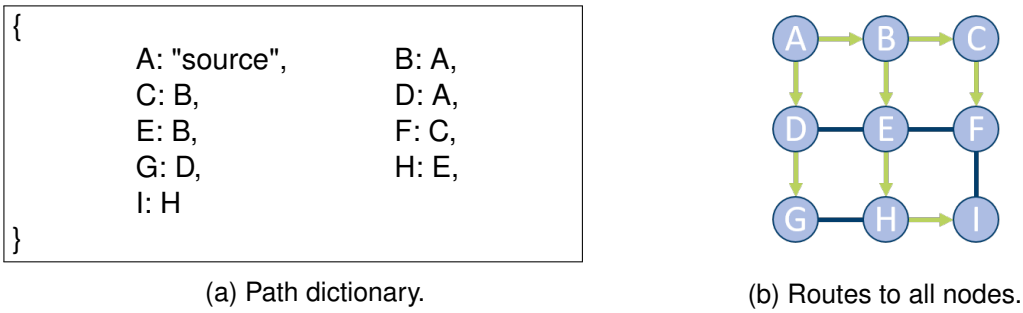
(a) Omitting duplicates entries in the route to get the multicast route.



(b) Routing depends on other destinations for optimized MC routing algorithms.

Figure S1.3: Multicast routing.

to a node can then be determine from this path dictionary by going backwards from the destination to the source and reversing the resulting route. An exemplary path-dictionary is shown in figure S1.4a. To determine the route from node A to node I in this example, we can go backwards through the dictionary until the source is reached, resulting in this case in the sequence I $\rightarrow$ H $\rightarrow$ E $\rightarrow$ B $\rightarrow$ A. And thus the route from A to I is therefore [(A, B), (B, E), (E, H), (H, I)]. The path dictionary contains all necessary routing information for all nodes in the dictionary. Figure S1.4b shows the routes from A to all other nodes in the exemplary network.



(a) Path dictionary.

(b) Routes to all nodes.

Figure S1.4: Exemplary routing case.

1.2.5 Simulation Log

The model files contain one other module which has not been discussed so far, *sim_log.py*. This module serves debugging purposes and offers the user additional information about a (failed) simulation run. After a run is started, a log file is generated and the specified parameters of this run are saved first. Further along the run, the log keeps track of completed tasks, the execution times of these tasks, as well as notices, warnings, errors or in the case of a failed run, fatal errors that occurred during a run. Even after a successful run, the user should evaluate any notions and/or warnings raised during the run, to validate the correctness of the run.