A parallel implementation of current-reinforced random walks

David Jäderberg, Kristoffer Jonsson

Project in Computational Science: Report

February 2015
Abstract

Reinforced random walks occur in many complex systems, in particular biological systems such as construction of blood vessels or neural networks and trail-laying ants can at some level be described by reinforced random walks. One particular type of reinforced random walk, the current-reinforced random walk, is implemented in C++ with the library OpenMP for parallelism.

Results of simulating current-reinforced systems show that both the characteristics of the constructed network and the execution time needed until a solution is found strongly depend on the path maintenance parameter, $\mu$, as well as on the number of sources versus the number of sinks and their placement. If in the linear state, i.e. $\mu = 1$, the network will consist of the shortest paths from each source to each sink and it takes a long time to construct a stable network. If in the non-linear state, i.e. $\mu > 1$, there is instead a preference to share paths rather than taking the shortest path and the time needed to find a solution is much smaller. Results also show that if the value of $\mu$ is large enough the simulation breaks down and the solution found does not represent any meaningful network.

The model is also tested on a graph over the street network of the Uppsala, Sweden, where source are placed in the residential areas and sinks are placed in the city center. This can be seen as a simulation of traffic flow during rush hours and used to understand how to deal with traffic congestion.

In order to improve the performance of the implementation, the nodes are ordered with the graph partitioning software METIS. Measurements of fixed-size speedup and scaled speedup show that the parallel implementation significantly increases the overall performance.
Contents

1 Introduction .................................................. 4

2 Methodology and implementation ........................... 5
   2.1 Algorithm ................................................. 5
      2.1.1 Preparation ......................................... 6
      2.1.2 Confirmation ......................................... 7
   2.2 Implementation and parallelism ......................... 8
   2.3 Graph partitioning ....................................... 9

3 Result ......................................................... 10
   3.1 Homogeneous grid ......................................... 10
   3.2 City ..................................................... 10
   3.3 Parallelism ............................................... 14

4 Discussion ..................................................... 16

5 Appendix ....................................................... 20
   5.1 Object oriented programming ......................... 20
      5.1.1 Improvements ....................................... 21
1 Introduction

Networks created by biological systems such as construction of blood vessels or neural networks and trail-laying ants can be described by reinforced random walks [1, 2]. These biological systems can be seen as a massive cluster of tiny computational units, e.g. ants, generating the network in parallel, with every tiny component only knowing its local environment [3, 4]. Modelling complex systems with reinforced random walks is often done by simulating particles traveling from node to node in a specified graph, constructing networks. The paths taken by the particles depend on local environmental parameters at each node in the graph. The parameters for a node can for example be the particle density within the node or the flow of particles to another node. The probability for a particle to move to another node then depends on these parameters. Ants for example, as presented in [2] and [5], move around in their search for food or building material drop pheromone in certain patterns. If an ant has found food it drops an amount of pheromone which other ants can then register. Ants want to move along paths where the pheromone concentration is high, i.e. where other ants are moving [6]. This causes the ants to create the so called current-reinforced patterns through their transport network.

Previous studies have shown how biological systems can be modelled by current-reinforced random walks. In current-reinforced random walks, the probability for a particle to move to another node depends on the flow of particles to that node, i.e. the current of particles. The basic idea of a current-reinforced random walk is to find the shortest path between a source and a sink for a given graph. This optimises the path length for each individual path from the sources to the sinks. The algorithm for network creation by current-reinforced random walks used in [7] converges to the shortest path through the graph [8]. When utilising non-linear current-reinforced random walks a combination of path length and path maintenance is minimised. For example, ants may want to construct one big road rather than many small ones to easier keep needles and dirt off the road while not making each individual path that much longer. The actual function that is minimised in non-linear cases is not yet known, a suggestion for this function is given in [7].

Another example of networks that can be modelled by current-reinforced random walks is when modelling certain types of electrical systems. As presented in [7], the movement of the electrons in special types of current dependent conductors can be modelled by current-reinforced random walks. In these systems the conductivity in the conductors is proportional to the amount of electrons flowing through it, i.e. the current. This property enables the analogy to ant trail networks where pheromone concentration corresponds to the conductivity in the electrical network. What all these different networks have in common is that the particles flowing within the network have no global information of the network itself, i.e. they can only know and affect their local environment.

When simulating these systems it may seem natural to implement the algorithms within a parallel setting due to the parallel nature of the systems themselves. However,
previous simulations of current-reinforced random walks have been implemented serially in low performance environments [7, 9]. This is mainly because high performance parallel programming is not that simple as it may seem at first glance. Since all particles affect and react to their local environment synchronisation is needed when moving particles around in the network, otherwise the computation would be corrupted. This project aims to be the first high performance parallel implementation of the current-reinforced random walk network creation presented in [7], utilising high performance languages such as C++ along with graph partitioning schemes such as METIS [10] for better performance. The purpose is to decrease computation time and increase the problem size. Relevant measures for this are fixed-size speedup and scaled speedup as a function of number of utilised parallel cores.

2 Methodology and implementation

2.1 Algorithm

The core of modelling current-reinforced random walks is that there are sources, sinks, nodes, particles and edges all organised within a graph representing a network. The fundamental idea is that sources send out information, sinks collect information and particles represent information. Edges are connections between sources, sinks and nodes and represent a path for the information to spread. Sources have a production rate, i.e. the rate at which new particles are being injected into the system. Sinks have a removal rate, i.e. the rate at which particles can be removed from the system. Sources, sinks and nodes can all contain particles. For a system to have a viable solution, i.e. a stable network, the cumulative production rate of all the sources must be less than or equal to the cumulative removal rate of all the sinks, otherwise no flow equilibrium can be established.

There are some necessary parameters for this model to make sense. These parameters and their analogy in both electrical networks and ant trail networks are shown in Table 1. How these parameters are used in the algorithm is explained in detail in Section 2.1.1 and 2.1.2.

The underlying algorithm for the current-reinforced random walk can be described by some key steps. At each time step the algorithm can be divided into two parts, one preparation part and one confirmation part. The reason for this is that synchronisation is needed when implementing the algorithm in parallel. This is further explained in Section 2.2. An outline of this algorithm is shown in Algorithm 1 and is explained in more detail in sections 2.1.1 and 2.1.2.
Table 1: Table shows the necessary parameters for the current-reinforced random walk modelling and their analogy in electrical networks and ant trail networks. The index $i$ denotes that the parameter corresponds to node $i$ and $ij$ denotes that the parameter corresponds to the edge between node $i$ and node $j$.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Electrical network</th>
<th>Ant trails</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_{ij}$</td>
<td>edge length</td>
<td>edge length</td>
</tr>
<tr>
<td>$N_i$</td>
<td>number of electrons</td>
<td>number of ants</td>
</tr>
<tr>
<td>$P_i$</td>
<td>potential</td>
<td>density of ants</td>
</tr>
<tr>
<td>$I_{ij}$</td>
<td>current</td>
<td>flow of ants</td>
</tr>
<tr>
<td>$\bar{I}_{ij}$</td>
<td>expected value of current</td>
<td>mean flow of ants</td>
</tr>
<tr>
<td>$D_{ij}$</td>
<td>conductivity</td>
<td>pheromone concentration</td>
</tr>
<tr>
<td>$C_i$</td>
<td>capacitance</td>
<td>total pheromone density</td>
</tr>
<tr>
<td>$q$</td>
<td>reinforcement intensity</td>
<td>pheromone drop rate</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>conductivity decrease rate</td>
<td>pheromone evaporation rate</td>
</tr>
</tbody>
</table>

Algorithm 1: Outline of the algorithm used in the current-reinforced random walks in [7].

2.1.1 Preparation

The preparation begins with calculating the mean flow $\bar{I}_{ij}$ from each node to each of the node’s neighbours as
\[ I_{ij} = \frac{(N_i - N_j)D_{ij}}{l_{ij}}. \] (1)

From (1) it can be seen that the mean flow depends on the conductivity of each edge, \( D_{ij} \), which is initialised at and kept at or above some minimal value. The actual flow along the edge \( ij \) can then be calculated as

\[ I_{ij} = \text{Poi}(\bar{I}_{ij}|\Delta t), \] (2)

where \( \text{Poi} \) is the Poisson distribution with the given expected value.

In terms of implementation, it is important that nodes that are connected to the same edge agree on the same number of particles to transfer. This is ensured by allowing only the node with a larger number of particles to randomise the value and then both nodes use that value when updating the number of particles. This will also help to ensure that there are never fewer than zero particles at a node, since the number of particles will decrease at the node with the largest number of particles. So, allowing only the nodes with the largest number of particles to randomise the flow means that nodes can be stopped from sending away more particles than are already at the node.

The order of randomising the number of particles to move to each neighbouring node is uniformly randomised at each time step for each node. This means that there is no bias in choosing flow calculation order, i.e. no node will always have its number of particles randomised first. This is done to minimise bias in the randomisation when using large time steps, since large time steps leads to large flows and possibly cases where all particles (or more, if it was possible) at a node move during the same time step.

When a node has a smaller number of particles than a given neighbour, the node always calculates the flow to that neighbour as zero and then waits for the neighbouring node to tell it how many particles should be moved. After all nodes have calculated and stored \( \bar{I} \) and \( I \), the preparation step is finished.

### 2.1.2 Confirmation

At the beginning of the confirmation part, the data regarding how many particles will move along each edge is available. And so the confirmation step begins by updating the conductivity at each node, which is calculated as

\[ D_{ij}(t + \Delta t) = D_{ij}(t) + q|\bar{I}|^\mu - \lambda D_{ij}(t)\Delta t, \] (3)

where \( q \) is the reinforcement intensity caused by per unit flow, \( \mu \) is the path maintenance parameter, \( \lambda \) is the conductivity decay parameter and \( \Delta t \) is the time step. If \( \mu \) is set to one, the current-reinforced random walk is linear and will converge to the same solution every time. If \( \mu \) is set to a value greater than one the current-reinforced random walk becomes non-linear and will not converge to the same solution every time that a system is simulated. In the algorithm’s linear state it optimises the path length for each individual path from the sources to the sinks, i.e it minimises the cost function.
\[ \sum_{j \in \text{Neighbours}(i)} l_{ij} \bar{I}_{ij}, \]  

(4)

where \( l_{ij} \) is the edge length between node \( i \) and \( j \) and \( \bar{I}_{ij} \) is the corresponding mean flow of particles. This means that the algorithm will always find the shortest path through the graph. When utilising the algorithm in its non-linear state a combination of path length and path maintenance is optimised. No cost function for the non-linear case has analytically yet been found. A proposition for such a cost function is suggested in [7].

The number of particles can be updated either before or after the conductivity is updated. The new number of particles is calculated as

\[ N_i(t + \Delta t) = N_i(t) + \sum_{j \in \text{Neighbours}(i)} (I_{ji} - I_{ij}). \]  

(5)

The capacitance is then calculated as

\[ C_i = \sum_{j \in \text{Neighbours}(i)} D_{ij}/l_{ij}, \]  

(6)

and finally the potential is calculated as

\[ P_i = \frac{N_i}{C_i}. \]  

(7)

When this is done the time step is finished and the next time step can be started.

2.2 Implementation and parallelism

In order to achieve short execution times it is important to implement the algorithm within a high performance environment. An ideal programming language for applications with high performance requirements is C++, which gives the programmer good control over optimisations and provides support for low level memory manipulation. It is also a good language for object oriented programming, which makes application design much easier. Other good features of C++ are that it has generic and imperative programming features and is also very well suited for implementing parallel applications, both when utilising smaller shared memory models and big scale message passing systems. Since the simulations done in [7] were implemented serially in Matlab, C++ is a very suitable language of choice for significantly increasing the performance and taking the simulations into the parallel realm.

The algorithm shown in Algorithm 1 can be parallelised in each of the outer loops, with synchronisation after each preparation step and after each confirmation step. The synchronisations are necessary since the update of the conductivity depends on the flow from the neighbouring nodes and since the update of the mean flow depends on the potential of the neighbouring nodes. Therefore, a node and its neighbours must all have done the necessary steps before moving on to the confirmation part. So strictly speaking, global synchronisation is not necessary, i.e. all nodes must not wait for all nodes to
finish the preparation part, a node must only wait for its neighbours to finish. However, global synchronisation is much simpler implementation-wise and may not be slower on the shared memory machines that are used, since it avoids complicated synchronisation patterns.

The actual implementation uses OpenMP [11] to parallelise the computations, the outer for-loops in Algorithm 1 are parallelised using the directive #pragma omp parallel with a #pragma omp barrier in between them. Each of the for-loops explicitly assigns the nodes within a partition to each core, perfectly matching the nodes assigned to a core with the graph partitions generated by the method described in Section 2.3.

The source code has been made freely available at [12].

2.3 Graph partitioning

When utilising multicore systems it is important to minimise cache invalidation, i.e. communication between cores, in order to get good performance. This can be done in numerous ways but when working with graphs a natural way of minimising cache invalidation is to partition the graph. Partitioning the graph must be done in a clever way so that there are as few edges as possible connecting nodes belonging to different cores. A rigorous way of partitioning graphs is presented in [10]. A schematic view of such a graph partitioning is shown in Figure 1. Graph partitioning has been studied extensively, hence many softwares for partitioning graphs in good ways have been developed through the years. The particular software used for the graph partitioning in this project is METIS [10].

The graph partitioning method for lowering communication is applied in such a way that each core being used in the simulation is assigned one partition of the graph, hence the number of partitions and the number of cores must be equal. In both the preparation step and the confirmation step in Algorithm 1 cache is invalidated since nodes write and read data that other nodes use. With the graphs partitioned a lot less cache is invalidated since the number of edges connecting nodes belonging to different cores are much smaller when each core handles a partition rather than nodes scattered all over the graph. The only cache that is invalidated now is the cache used by nodes laying on the boundary of the partitions.
3 Result

3.1 Homogeneous grid

A homogeneous grid containing 1024 nodes where the position of each node has been perturbed by a small amount was used to execute the algorithm with several different choices of parameters, with a focus on the maintenance parameter, $\mu$. Figure 2 shows the simulations for various values of $\mu$ when having many sources and only one sink in the graph. Figure 3 shows the simulations for various values of $\mu$ when having many sinks and only one source in the graph. The solutions that is found when $\mu = 1$ are the shortest paths between each source and sink. When $\mu > 1$, the path chosen may not be the shortest path and there also exists a preference to share paths. Observe that the results presented are when the solution is stable, i.e. when the network pattern it creates tend not to drastically change.

3.2 City

Another graph was created using road data from OpenStreetMaps for the city of Uppsala, Sweden. Figure 4 shows the result of the simulation when sources are placed in the residential areas and sinks are placed in the city centre. This can be seen as a simulation of traffic flow during rush hours.
Figure 2: Solutions found by running the simulation on a homogeneous grid containing 1024 nodes where the position of each node has been perturbed a small amount. The blue lines show the conductivity along each edge, where the thickness of the line is proportional to the conductivity. The shortest path found with Dijkstra’s algorithm from each source to each sink has been marked with a thin black line [13]. The parameters used are $q = 10^{-4}$, $\lambda = 10^{-3}$, and $D_{min} = 5 \cdot 10^{-2}$ with a time step of one. In this graph there are in total one sink and 20 sources. Each source (red) produces 1000 particles each time unit and the sink (green) can remove as many particles as are produced each time unit.
(a) The simulated solution when $\mu = 1$. Solution at approximately 1400000 time steps.

(b) The simulated solution when $\mu = 1.05$. Solution at approximately 120000 time steps.

(c) The simulated solution when $\mu = 1.4$. Solution at approximately 20000 time steps.

(d) The simulated solution when $\mu = 1.5$. Solution at approximately 20000 time steps.

Figure 3: Solutions found by running the simulation on a homogeneous grid containing 1024 nodes where the position of each node has been perturbed a small amount. The blue lines show the conductivity along each edge, where the thickness of the line is proportional to the conductivity. The shortest path found with Dijkstra’s algorithm from each source to each sink has been marked with a thin black line [13]. The parameters used are $q = 10^{-4}$, $\lambda = 10^{-3}$, and $D_{min} = 5 \cdot 10^{-2}$ with a time step of one. In this graph there are in total one source and 20 sinks. The source (red) produces 1000 particles each time unit and all sinks (green) can together remove as many particles as are produced each time unit (evenly divided between the sinks).
Figure 4: Solution at 1000000 time steps found by running the simulation on a grid of the city of Uppsala, Sweden containing 5926 nodes. Sources (red) are placed in the residential areas and sinks (green) are placed in the city centre. The total production rate is 80000 per time unit, equally divided between all sources. Similarly, the total removal rate is 80000 per time unit, equally divided between all sinks. The blue lines show the conductivity along each edge, and the thickness of the line is proportional to the conductivity. The parameters used are $\mu = 1.1$, $q = 10^{-4}$, $\lambda = 10^{-3}$, and $D_{\text{min}} = 5 \cdot 10^{-2}$ with a time step of 0.1.
3.3 Parallelism

The measurements used to benchmark the performance of the parallel OpenMP implementation was fixed-size speedup and scaled speedup. The fixed-size speedup with \( n \) cores is how many times faster the program solves the same problem, compared with using one core. The scaled speedup is how many times larger the problem can be (measured in number of nodes in the graph), so that the execution time with \( n \) cores is the same as with one core. Figure 5 shows the result from the fixed-size speedup and scaled speedup measurements. All benchmarking were done on the Uppsala University servers \textit{vitsippa.it.uu.se} and \textit{tussilago.it.uu.se}, both with specifications:

- CPU: AMD Opteron (Bulldozer) 6282SE, 2.6 GHz, 16-core, dual socket
- Memory: 128 GB
- OS: Scientific Linux 6.5.

All C++ code was compiled with gcc 4.8.2 with the \texttt{-fopenmp (OpenMP 3.1), -O3 and -static-libstdc++ flags}. The graphs were partitioned with METIS 5.1.0.
(a) The fixed-size speedup of the parallel implementation. A uniformly randomized grid containing 10000 nodes was used for measuring the fixed-size speedup. The simulation was run for 1000 iterations with a time step of 0.1.

(b) The scaled speedup of the parallel implementation. The reference size, i.e. the size of the problem when scaled speedup is one was a uniformly randomized grid containing 5000 nodes. The computation time used for measuring the scaled speedup was 60 s with an allowed variation of ±3 s.

Figure 5: Different measurements of the parallel implementation of the algorithm presented in Algorithm 1. All benchmarking was run three times per data point and the best performance among those data points was extracted and used in these results.
4 Discussion

Considering the results in Figure 2 and 3, it can be seen that when $\mu = 1$, the particles move along the shortest path between each source and the sink, while even a small non-linearity where $\mu = 1.05$ shows a clear preference to share paths from the sources to the sink. Notable is also the difference in amount of time steps required to achieve a stable solution. Since the behaviour fundamentally changes when utilizing a $\mu > 1$ this is not so surprising. Non-linearity emphasizes the tendency to create big roads and suppresses small roads. Hence small roads will faster be eliminated in a non-linear simulation compared to a linear simulation where small roads, if not being on the shortest path, only will disappear by statistical properties, i.e. they will only be eliminated since it is more probable to take the shortest path than any other path.

For larger values of $\mu$, e.g. 1.4 and 1.5, the preference to share paths is much more pronounced judging by the result in Figure 2, but it is not clear that 1.5 leads to a stronger effect than 1.4. This may be due to the stochastic nature of the algorithm and due to the non-linear reinforcement, which means that differences in the early stages of the simulations may persist forever. This can also lead to slightly different solutions when running the simulation multiple times with the same parameter values.

When increasing $\mu$ and running the simulation on a graph with only one source and many sinks the network creating pattern breaks down. This can be explained by the fact that the non-linear simulation has multiple stable solutions as opposed to the linear simulation which only has one solution (shortest path). Due to the non-linearity, the simulation can be stuck in a state where particles basically only are moving between some, often only two, nodes reinforcing the flow and conductivity for only those nodes and edges, which creates a feedback loop. This is a state where the non-linearity is strong enough to prohibit the simulation from finding any meaningful solution. For low values of $\mu$ the stochastic diffusion of particles can overcome this phenomenon and instead find stable network patterns.

The reason for this phenomenon not occurring for $\mu = 1.4$ and $\mu = 1.5$ when running on a graph with many sources and only one sink, seen in Figure 2, could be explained by a combination of the difference in production rate for the sources and the difference in distance to nearest source. The fact that the source in Figure 3 has 20 times higher production rate than the individual sources in Figure 2 adds non-linearly to the local current-reinforcement increasing the chance of a stable feedback loop. An idea to deal with this problem could be to reduce the non-linear reinforcement for a node as the number of particle within that node increases. This could perhaps prohibit feedback loops. The difference in distance to the nearest source could also add to the difference in the behavior. If there are sources evenly spread throughout the graph it is more probable for the sources to connect to each other due to the current-reinforcement. Hence if there are many sources it is basically enough that only one source finds the sink and all the other sources will connect to each other and form a network. This cannot happen between sinks, since they have no tendency to connect to each other. When looking at the city centre in Figure 4, the difficulty of creating networks in between sinks can also
there be observed.

The simulation when run on the graph of Uppsala shows that the non-linear current-reinforced random walk approximates the traffic flow to a city centre quite well. Looking at the different residence areas it can be seen that they are drained of particles in a very natural way, everyone are seeking the big roads to faster transport themselves to the city centre. By running a simulation like this the necessary road sizes for sustaining traffic demands, or if to put a road block in a residence area to prohibit traffic flow through there can be predicted. This area could be much further investigated, this simulation is merely scraping the surface of possible road map simulations.

Judging by the results presented in Figure 5 the parallelisation gave significant fixed-size speedup and scaled speedup. The resulting fixed-size speedup is roughly 0.5 times the number of cores, which is a good fixed-size speedup considering that there are two synchronisations in each iteration of the algorithm. The scaled speedup is also good at almost 0.5 times number of cores. The drop in fixed-size speedup at 16 cores can be due to that the servers vitsippa.it.uu.se and tussilago.it.uu.se are dual socket and therefore have non uniform memory access times. Since the servers each have 32 cores, using 16 of them might result in conflicts with other processes running on the servers, meaning that threads will be placed on both sockets which raises memory access times for some threads. This increases the execution time since all cores need to synchronise. The probability for this phenomena to happen increases with the number of cores being used.

The reason for the scaled speedup being less than the speed up may be due to cache misses. Since the memory requirements increase when the problem size is increased there will be more cache misses for bigger problems, hence the scaled speedup will not be as great as the fixed-size speedup.

To improve performance of the implementation, it may be useful to weight each node in the graph partitioning according to the number of neighbours that each node has, since the computational demand scales with the number of neighbours. It may also help to dynamically re-partition the graph, taking advantage of the fact that no computations need to be performed at nodes that do not contain any particles.

A natural progression of this work is to introduce communication between cores that do not share memory, e.g. via message passing. This would likely only provide performance improvements for large problems, since it relies on higher-latency communication.
Acknowledgements

Thanks to our supervisors, David Sumpter and Jarmo Rantakokko, for support and advice.

Road data from ©OpenStreetMaps contributors was used for creating the graph of Uppsala, Sweden. This data is made available under the Open Database License: http://opendatacommons.org/licenses/odbl/1.0/. Any rights in individual contents of the database are licensed under the Database Contents License: http://opendatacommons.org/licenses/dbcl/1.0/ - See more at: http://opendatacommons.org/licenses/odbl/#sthash.Ts31Orqp.dpuf
References


5 Appendix

5.1 Object oriented programming

There are two main class structures used in the program, one to represent graphs and one to represent different computational methods.

The class NodeSet exists to model a graph. This is done by storing all nodes, each in an instance of the class Node, and their neighbours, where each Node has a list of pointers to all its neighbours. To model sources and sinks, the classes Source and Sink exist, and they are both subclasses of Node. This is in itself enough to model a graph, but there are also some extensions to help dealing with graphs where every node has a given position in a Euclidean space. To do this, the classes PositionedNodeSet, PositionedNode, PositionedSource, and PositionedSink are introduced, which are all similar to their non-positioned versions. These classes are then used as tools when implementing the actual computations.

The class structure allows to implement different computation algorithms using the same framework. To do this, there exists the abstract class Algorithm, which contains a few pure virtual methods that must be implemented by each subclass. The main subclass of Algorithm is CurrentWalk, which implements the current-reinforced random walks. Each instance of Algorithm expects to be given a single Node and uses that as the basis for its calculations.

These two groups of classes are then used by the main program to perform simulations. Given a method to create instances of some subclass of Algorithm, the main program is then able to construct a NodeSet with a set of Nodes and an Algorithm for each Node. The algorithm advances using the takeStep() method in NodeSet. A schematic overview of this structure is shown in Figure. (6).

![Figure 6: An overview of the relation between elements in the graph and different classes in the program structure. It can be seen that each NodeSet contains both a set of Nodes and a set of Algorithms.](image)
5.1.1 Improvements

The original program design did not include the Algorithm class, but rather held all computation methods and state in Nodes. As such, the current design still has some artefacts, such as Nodes still containing a few select members that store the state of the computation. This is currently needed because the tool to draw graphs is unaware of the existence of Algorithm, meaning that it requires all state that it uses (e.g. mean flow or conductivity) to be stored in the Node object. Similarly, Nodes are unaware of the Algorithm that computes on them, leading to issues if it is necessary to access algorithm-specific data of ones neighbours, which is the case with current-reinforced random walks if the flow is considered algorithm specific. Allowing each Node to hold a pointer to the Algorithm that operates on it may be preferable, although the naïve approach may require dynamically typecasting all Algorithms every iteration.

Another issue with the current implementation is that it does almost no error handling, which is particularly problematic when dealing with IO, especially when manually specifying which files to use. This should likely be done with use of exceptions, which are not used at all in the current design.