

# Simulating Cooperating Localised Agents on Graphs

K.A. Hawick and H.A. James  
Institute of Information and Mathematical Sciences  
Massey University – Albany  
North Shore 102-904, Auckland, New Zealand  
Email: {k.a.hawick,h.a.james}@massey.ac.nz  
Tel: +64 9 414 0800 Fax: +64 9 441 8181

## Abstract

A microscopic agent formulation is an appealing approach from a simulation perspective for many complex systems involving cooperative behaviour. It is satisfying to construct a detailed localised model of contributing agents and to experiment with a collective to study emergent effects in the overall system without having to build in global heuristics that anticipate known solutions or behaviours. The collective world in which the agents transact their operations can take several different forms, the most general of which is an arbitrary graph. We describe our simulation framework engine for studying cooperative effects amongst agents on graph structures and report on some experiments on path-finder agents that are limited to localised knowledge and heuristics. We explore some consequences of graph connectivity and the interplay between short and long-range agent spatial knowledge and present some preliminary results on autonomous exploration agents. We also describe ideas and issues for generalised simulation engines for interacting agents on graphs.

**Keywords:** agent; graph; simulation engine; simulation visualisation; path-finder experiments.

## 1 Introduction

Simulations that aim at an understanding of complex systems are often usefully formulated in terms of microscopically detailed constituent agents that carry out their programs within a suitable framework. We are especially interested in spatially embedded agents or **animat agents** that model some physical or sociological phenomena. The world in which agents move and interact can be modelled in a number of ways.

A system of real-space Euclidean geometric coordinates can be employed with agents having almost arbitrary locations specified by floating point coordinates. In such a system it is hard to localise agent programs as there is no simple way for an agent to keep track of which other agents are in its proximity without an explicit search and check. More commonly agent worlds are formulated as a set of discrete coordinate values with topologies such as a lattice or mesh. Many models in computational physics, chemistry and engineering are formulated as a lattice or regular mesh of nodes with a connectivity that specifies the nearest neighbours, next nearest, and so forth. Recent work has considered the more general case of an arbitrary graph connecting agents, so that they have definite numbers of neighbours and can readily interact with them, but where the connectivity is more general than a mesh and individual nodes can have widely differing number of neighbours.

There are two interesting ways to explore the intermediate cases between fixed and regular topology lattices and arbitrary graph networks. One is to start with a lattice and remove some of the linking edges and the other is to add extra and usually longer range shortcut links to such a lattice. The former as usually referred to as a damaged lattice and the latter as a small-world network or lattice with shortcuts or “spatial worm-holes.”

In this paper we describe our simulation framework for addressing certain sorts of agent problems on arbitrary graph networks that can be constructed by removing or adding links to a known lattice structure or by other means. We believe this framework is very general and opens up a route to study many interesting cooperating agent problems. In this paper we focus on the problem of path-finding agents that can communicate and cooperate in various ways. We explore the capabilities of our simulation system in the context of the well-known and some new agent oriented path-finding strategies.

Our central goal is to be able to program microscopic agent behaviours entirely locally – in such a way that an individual agent can determine its own actions purely from its own state and knowledge of its immediate surroundings only. Ideally we want our framework to be reusable and separate from the details of a particular sort of animat agent that it hosts. We need to set up some measurement metrics and apparatus to help study collective behaviours of agents as well as to help debug and verify intended microscopic (individual) behaviours.

It is not entirely trivial to implement such a framework scalably so that relatively large numbers of agents may be simultaneously hosted. This is important as many of the complex systems phenomena that we seek to study do require relatively large numbers of microscopic agents to support the length and times scales upon which “interesting” features emerge.

We describe some of the various abstractions we have considered in our design so that agents can “self organise” as they interact at nodes of an arbitrary graph structure of which they have (initially) only localised knowledge.

In section 2 we describe our high-level simulation architecture, and give details on how the graph based world is managed in section 3. We describe how we formulate agent hierarchies in section 4 and report on some results concerning autonomously exploring path-finder agents in section 5. We discuss some of the ramifications of our agent simulation experiments in section 6 and draw some conclusions and suggest areas for further simulation work and engine development in section 7.

## 2 Simulation Architecture

In this section we describe our simulation architecture. The basic model is that of mobile and static agents that “live” (are hosted) on the vertices and arcs of a graph.

We were motivated to develop the framework we describe in this paper by a desire to study the spatial interplay effects of interacting agents. Two applications originally suggested themselves to us: simulating **animat agents** or “artificial life-forms” [7]; and simulating **traffic models** on computer networks and transport networks [6]. In both cases we found similar requirements for agent hosting and a management framework. We needed a new framework since we want to have agents potentially occupy **both** vertices **and** arcs.

Graphs are interesting to us as they have the interesting property of specific neighbour relations, unlike the “soup” world models where agents can “see” or “communicate

with” arbitrary other agents in the system. They are also supportive of more flexible spatial structures than lattices or meshes would be. We further discuss the use of graphs in section 3.

In our model we consider the case where those agents that inhabit the vertices of our graph are static, and those agents that inhabit the arcs are mobile. Of course, this is an arbitrary classification and is only a reflection of our viewpoint of the system. Generally speaking static agents are invoked in a fair manner by the master program to move the whole world forward one time-step. How individual spatially-static agents deal with their local traffic of mobile agents is an issue separate from the overall framework.

This approach gives us a powerful framework for exploring systems that host agents that: move around effectively autonomously; agents that cooperate with agents they encounter by communicating information; agents that consume or destroy agents that they encounter; and agents that spawn new agents. In this paper we focus on ideas and experiments to verify the **autonomous movement capabilities** of animat agents.

The few strict requirements that we have of our framework include: the ability to move simulation time forward in a fair way; to enable to visualisation and measurement of the agents inhabiting our system; the ability to load and save configurations; and to import and export fragments of the system state for analysis in external programs.

The ability to move simulation time forward in a fair way is crucial to the meaningfulness of simulation results. The naïve way of implementing a time-step update rule is to iterate across each of the nodes, in numerical order, instructing each agent to fire. This approach leads to so-called “sweeping effects” which reduces the significance of statistics generated from the model. Other approaches simply guarantee that, *on average*, each agent will be fired once per time step. This, too is also not good enough for a microscopic model. We start off with a vector containing the indices of all the agents currently in the system. The elements in this vector are randomly permuted a number of times to ensure a random order. Agents are then fired in this new order, which changes each time step. Thus, a fair agent firing order is guaranteed. Other agent simulation systems [8] have used operating system level concurrency support such as threads or process management. While this is useful in hiding concurrency issues it can be limited by system constants such as maximum thread or process numbers. Our system supports arbitrary (up to memory limitations) of agents under a fairness control

that we can control and vary at will without experiencing operating systems anomalies.

Static agents in our system are represented by the **Machine** classes: **Source Machine**, **General Machine** and **Sink Machine**. **Thaum** agents are dynamic agents. Each **Thaum** agent has enough intelligence to implement simple instructions based on the local state information available to it. Not surprisingly, the **Source Machine** is responsible for emitting **Thaum** agents, and the **Sink Machine** is responsible for receiving (and perhaps consuming) the **Thaum** agents. **General Machine** agents simply accept mobile agents arriving on their input queues and emit them to either their preferred, or a random, output queue, depending on the **Thaum** agent preferences.

Visualisation and the gathering of statistical information from our framework is enabled by a careful layering of the Java object hierarchy. Instrumentation is achieved through the object hierarchy by allowing each object to provide its own rendering and statistics-collection mechanisms. For example, **Thaum** age is represented by inserting the current time stamp information into the **Thaum** when it is created. When a **Thaum** arrives at the **Sink Machine** the age is calculated as the offset between when it was released and when it arrived.

### 3 Graph Construction

Most of the physical models involving animat agents that we know of have some simplifying graph properties. We do not allow multiple arcs nor self-arcs. Our arcs are directed however, and we can map an edge to a pair of two directed arcs managing traffic of mobile agents in the two directions. There are a number of ways to represent a graph data structure, depending upon the algorithms that will use it. We have used highly compact graph data structures for very large graph models using neighbour lists. In general for experimenting with smaller graphs it is convenient to use linked lists of vertices and arcs when the graph is being constructed, and once it is stable and not likely to change, a full adjacency matrix and individual adjacency in and out lists can speed up most graph traversal algorithms. It is convenient to abstract these data structures and the cache management issues that go with them into the agent simulation framework and away from individual agent codes. We want to present an coding interface for implementation of the Agent's *fire()* method that supports operations like determining how many neighbours a static agent on a vertex has; or determining how many input queues have mobile agents awaiting processing on them.

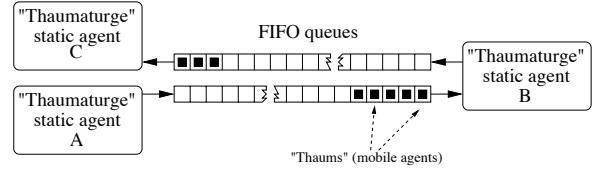


Figure 1: Graph edges consist of two directed arcs, each supporting a queue of mobile agents. The queues can be constrained (in size) or left unconstrained (subject to memory availability)

Figure 1 shows how we implement the traffic of mobile agents in our system. The “Thaumaturges” are the static agents (wielders of magic) and the “Thaums” are the mobile agents (units of magic) that travel along the arc-way queues.

By hiding the data management structures from the agent *fire()* method code, we can make framework optimisation choices depending upon graph sparseness independently of agent algorithm experimentation. This also allows us to maintain two bodies of code support - for Java Agents and for C++ agents. We have been able to use library implementations of the various data structures we discuss above to make it easy to port an agent code between the two implementations. As a general observation we often find it easier to rapid prototype an agent idea in Java but may prefer to later make a more efficient C++ implementation for systems that need large agent numbers or need to run on parallel or distributed computing hardware. It is generally easier to develop visualisation debugging and analysis code to work within the Java framework. In any case we believe it valuable to cross-validate microscopic agent behaviours against two separate implementations.

In the work reported in this paper we set up a number of 2D and 3D meshes and either damaged them (removed links) or added small-world shortcut links to make them generalised graphs representing mazes.

We consider a graph that describes the complete structural composition of the Agent's world as  $G = (V, E)$  where  $v \in V$  are the vertices and  $e \in E$  are the edges. Most of the models of interest have edges as against single one-way arcs.

In the work reported in this paper we experiment with path-finding agents to verify the ability of our framework to manage and track populations of autonomous agents that move around in a spatial world represented as a graph.

We are generally interested in a Path or ordered sequence

of nodes visited, written as  $P_i = [v_O, v_a, v_b, v_c, \dots, v_T]$ , where we take  $v_O$  as the origin and  $v_T$  as the target vertex. In general there may be many paths connecting the agent origin with the target, and we are experimenting with agent communication mechanisms to collectively find (and manipulate or exploit) the shortest path. It is sometimes useful to seek not the shortest path but the spanning structure for an explorer agent seeking knowledge about the entire graph world.

Our model consists of Agent  $i$  being released from the Origin vertex  $v_O$  at time  $t_i^O$ , following path  $P_i$  to the target vertex  $v_T$ , where it (may) arrive at finite time  $t_i^T$ . We study the histograms and statistics pertaining to  $\delta t_i = t_i^T - t_i^O$  as we vary the agent model, the number of agents released over time, and the effect of different combinations of agent species populations.

The problems of informed and un-informed searching are well known in Artificial Intelligence [9] and Operations Research [1, 2]. In this paper we explore how to implement some well-known search agent formulations but also discuss how these might be combined in novel ways using our animat agent framework.

## 4 Animat Agents

We have already discussed the topological arrangement of nodes and vertices in our simulation framework in section 2. Each agent, now spatially located and managed within our simulation framework, has the ability to “fire”. When an agent fires, it is able to affect the **local state** of the system. As previously mentioned, in our simplistic simulation model when a static agent fires, it essentially reads the incoming **Thaums** from the input queue and processes them according to the local rules. **Thaum** agents are mobile agents. They “live” on the arcs of the graph. When **Thaum** agents fire, they have the ability to choose which static node they will be directed towards in the next simulation time step. It is important to bear in mind that there is no set order in which static agents will invoke their fire methods. Thus, it is entirely possible that a **Thaum** may be processed by one static agent in a given time step, and then passed to another static agent, which is then fired *in the same time step*.

Each agent only relies on local information. Thus, static agents only know what their “instruction sequence” is and who their connected neighbours are. They have no idea whether one of their neighbours is the source or sink node, nor do they know how many **Thaum** agents are waiting in input or output queues. Mobile agents have a sim-

ilarly local view of the environment. Depending on the sophistication of the model, the mobile agent may build up partial knowledge of the static agents visited so far, or a path to take to arrive (eventually) at the sink node. We discuss such paths in section 5.

There are many different types of agents that are realisable in our simulation framework. For example, in this paper we have concentrated on path-finder agents, whose goal it is to find an efficient route from the source to a sink node in a graph. Other agents that are possible are communication agents for implementing sensor networks [5], networks of animals in artificial life systems [7] and teams of agents cooperating to accomplish some task [3].

In this paper we concentrate only on the path-finding application and pay particular attention to the way in which our static and mobile agents can be imbued with a modicum of intelligence or memory to enable them to remember and communicate information for the benefit of other agents in the system.

## 5 Path-finder Animats

The path-finding problem is only one interesting problem that we can look at using a suitable agent simulation framework. The path-finding problem is essentially one of finding a “good” path from a source node to a sink node within a given simulation environment.

We have constructed a sequence of graphs, starting from a lattice in which all non-edge nodes have exactly four neighbours (in 2D) or six neighbours (in 3D) and progressively damaged them by removing 10% of the remaining arcs that join neighbours. In all the graphs that we have used in our simulations node 0 is the source node, and the highest-numbered node is the sink node. In no circumstances do mobile agents “know” which node they’re seeking.

There are a number of obvious strategies that one would normally try when performing such goal-seeking. The strategies range from the very simple to quite complex, and also may involve very little (or no) communication between mobile agents to quite complex interaction protocols. In table 1 we list some strategies and compare their relative features. In the context of the path-finder agents we have investigated, some were guaranteed to produce an optimal solution (such as the Depth-First Search – DFS – and Breadth-First Search – BFS – algorithms) while most simply tried to produce what is essentially a “good” solution. In fact, the first three models listed in this table are based on the same premise: a randomly dif-

fusive agent will sometimes accidentally find the optimal path from the source to the sink. Eventually, of course, the agent will stumble onto the sink node, in which case an acceptable path can be generated from the route the agent took by removing any extraneous loops in the circuit. Obviously this is not very useful unless the agent is able to communicate this path information back to the remainder of the agent in the system.

For the purposes of our model we allow the second (“Stealth”) and third (“Gossip”) agents to communicate their path information with other agents in the system by not disappearing when they reach the sink, but by returning to the source node. The Stealth model only communicates its new path information with the agent at the source node, and so each new agent created after the agent arrives back at the source (and those agents which stumble at random back to the source node) are updated with path information that is guaranteed to get them to the sink node, albeit not in an optimal number of steps. The Gossip model takes this approach one step further: each agent that is encountered along the path back to the source node is also updated with the relevant path information. If two agents meet and both have path information that will lead them to the sink, then only the shortest path information will be kept and thus propagated to other agents. Thus these first three agents have important physical analogies: the first model is a solo wanderer; the second only reports to a superior back at base; and the third is an altruistic agent that wants to aid all other agents.

Self-Avoiding Walk agents have the unfortunate pathological property that an agent can become trapped by its own tail - it can fence off possible routes by its previously-walked trail and may no longer be able to make progress. We can introduce the idea of a mostly self-avoiding walk agent that will avoid its trail unless it has to in order to make any more progress. For example, it could leap over its trail up to a fixed distance if it has to. We suspect this could give us a way to investigate the intermediate regime between local and global.

Two or more paths can be combined to find the shortest path between each segment. Crossover points can be identified in a single path and used to eliminate redundant loops. This is useful in obtaining the best currently known path, but it may be advantageous for the explorer agent to retain any apparently redundant loops since they add knowledge about *terra incognita* that may be usable later to connect together other path routes into an even shorter path.

We have run our different agents on eight different graphs

of size  $16 \times 16$  in 2D and  $8 \times 8 \times 8$  in 3D. As previously mentioned these graphs range from a fully-regular square (cubic) grid to grids with a high percentage of arcs removed.

In models 1 and 2 there will always be a certain number of **Thaum** agents that arrive at the sink with large times because until the first **Thaum** actually arrives at the sink and starts its return journey no “valid” path to the sink is known. In model 1 this path is only communicated when the **Thaum** arrives back at the source node. Bear in mind that these graphs are in log-linear scale.

As the fully-connected graph is progressively damaged we find that the number of **Thaum** agents with larger times increases. This is due to the greater dis-connectivity between nodes in the graph, and therefore more opportunities for the **Thaum** to get lost or to go down a “rat-hole”. We find that when the histogrammed population of agent ages or path-length solutions is plotted on a log-linear scale, the “straight-line” portion in each plot indicates the relationship:

$$\log P \approx -|m|l \implies P \approx Ae^{-|m|l} \quad (1)$$

for the probability  $P$  of a path length  $l$ . This is not unreasonable for each model will generate a range of agent solutions that can be exponentially long. We can characterise model behaviour in the different arc connectivity regimes by fitting a gradient to the log-linear plots produced by each model. These  $|m|$  values are (approximately) linear in the arc-perturbation (deletion) fraction. To a good approximation there is a positive correlation and in fact the exponent  $|m|$  is linearly related to the perturbation fraction  $|m| \approx C \times f \pm D$  with  $C \approx 0.002$  and  $D \approx 0.0001$ . Our graphs are too small to infer this linearity is a universal property but we plan to investigate larger systems more thoroughly to verify this. It seems not unlikely that this is a systematic characteristic of the distribution path-length solution as the graph is damaged.

If we consider the representative times taken by **Thaum** agents to reach the sink node, model 0 shows the effects of not allowing communication between agents: all agents take a variable amount of time to reach the sink node – some find it immediately and some take an extraordinarily long time to get there.

Model 1, the Stealth agent, shows the best performance out of all the models that feature random diffusion as a basis. The first three- to five-hundred **Thaum** agents released do a random walk to try to find the sink node. Some find it quickly, leading to a reduction in the average path length for the newly-created **Thaum** agents, but those agents that are still randomly walking will continue

Mobile Agent	Memory Type	Complexity	Will Complete?	Map Generated	Comms	Loops in Path	Optimal Path?
Random Diffusive	None	Trivial	Yes	All Visited	None	Yes	No
Stealth	Followed Path	Simple	Yes	Direct path	Only at source	Yes	No
Gossip	Followed Path	Slightly complex	Yes	Direct path	Every chance meeting	Yes	No
DFS	Stack	More complex	Maybe	World up to sink	None	No	Yes
BFS	Queue	More complex	Yes	World up to sink	None	No	Yes
SAW	Visited nodes	Simple	Maybe	All visited	None	No	No
Wall Follower	Followed path	Simple	Yes	All visited	None	Yes	Yes

Table 1: Characteristics of different path-finding agents. Depending on the requirements of the path-finder algorithm, it may be perfectly adequate to have the agent report an acceptable path, rather than the optimal path through the graph.

to take a potentially long time to arrive at the sink. After enough “good” **Thaums** arrive back at the source, the average transit time flattens out considerably.

In contrast to model 1, model 2, the Gossip agent produces quite interesting results. Remember that in model 2 the returning **Thaum** agent exchanges path information with all the exploring **Thaums** that it meets on its travel. The path taken to return to the source node is the same as it took to the sink node (minus any path loops). If the returning **Thaum** took a tortuous route to the sink node, it will pass that on to the exploring **Thaum**, possibly causing it to take a longer route than absolutely necessary.

## 6 Discussion

We have adopted an object-oriented approach throughout development of our framework. Past experience has shown us that going overboard with every entity being an object makes the code inefficient to run however and since our research programme aims to investigate emergent properties in systems with medium to large numbers of agent constituents, run-time efficiency is important. The object hierarchical management split of static and mobile agents is inevitable, but we have tried to avoid any deeper an object-oriented structure. “Small and many” summarises our approach and the concurrency available in our system can be exploited for optimisation in a number of ways.

Our simulation framework is primarily designed to be run either as an interactive tested (Java version) or a “batch managed” set of background jobs for gathering statistical averages. In fact either of our codes can be wrapped in appropriate scripts to support long-running simulation configurations. In such cases we can exploit any available parallelism through the statistics collation. For example 100 independent runs of stochastic agents can be run independently on 100 nodes of a cluster computer. It is not trivial to determine how to split a large arbitrary agent graph in a way that the “owner-computes” rule can be applied. Indeed one of the simulation applications is to determine how a cluster or computational grid graph might be split in this way [6]. We have experimented with various fine and coarse grained levels of concurrency in our system but expect to report this work elsewhere.

For the most part our target agent models are somewhat unusual and interoperability with other simulation frameworks in the simulation community is not as an immediate concern as efficiency. We remain interested in tracking agent communication and mark-up languages [4]. It is attractive to export or import agents from another system and we envisage further developing our system as special purpose agent hosting framework that is primarily aimed at speed, but which might be able to operate as a component in a broader system of more generalised agents. We are aiming at a middle ground between a super-generalised system that will be slow at runtime, and the super-fast but un-generalisable hand-crafted simulation codes that researchers (including ourselves) construct

for one-off simulation problems.

## 7 Summary & Conclusions

We have investigated some well known and some less-well known combinations of path-finding agents on a spatial graph using our static and mobile agent simulation framework. Our system is flexible enough to manage arbitrary number of both static and mobile agents. We are able to manage the agents without loss or gain and are able to keep track of population statistics centrally. This is important for the management of experiments involving stochastic ensembles of agents. Our system reproduces expected behaviour for the microscopic path-finding agents we have tested, and our architecture is implement-able in both Java and C++.

We have found some interesting scaling properties in the distributions of path-length solutions found by cooperating and non-cooperating agents in the context of systematically damaged graphs and. We have also identified stark contrasts in solutions found when small-world shortcut links are applied. We plan to explore the numerical scaling properties in more depth with experiments on more and larger stochastically generated graphs. We believe we have shown that our framework is capable of supporting controlled large-scale simulation experiments on systems of agents.

We plan to investigate other agent behaviours that specifically involve changes in the agent population sizes - such as predating and spawning agents. In this paper we focused on agents that cooperate by communicating information. We also plan to investigate “Byzantine General” agents and teams of competing agents that co-exist in the same world simultaneously. We believe these effects, which appear in many real-world complex systems, are worthy phenomena for more detailed simulation and can only be effectively tackled by a statistical simulation approach.

## Acknowledgements

We thank the Allan Wilson Centre for the use of the Helix Cluster Supercomputer and Massey University for the Monte Cluster.

## References

- [1] Deo, N. and Pang, C.-Y. Shortest path algorithms: Taxonomy and annotation. *Networks*, 14(2), 275–323, 1984.
- [2] Dreyfus, S.E. An appraisal of some shortest-paths algorithms, *Operations Research*, 17, 395–412, 1969.
- [3] Ferber, J. *Multi-Agent Systems*, Addison-Wesley, 1999.
- [4] Finin, T., et. al., Specification of the KQML Agent-Communication Language <http://www.cs.umbc.edu/kqml/kqmlspec>
- [5] Hawick, K.A. and James, H.A., Small-World Effects in Wireless Agent Sensor Networks, To appear in Int. J. Wireless and Mobile Computing, Special Issue on Mobile Systems, E-Commerce and Agent Technology.
- [6] Hawick, K.A. and James, H.A., Simulating a Computational Grid with Networked Animat Agents, Proc. 4th Australasian Symposium on Grid Computing and e-research, (AusGrid 2006), Hobart, January 16-19, 2006.
- [7] Hawick, K.A., James, H.A. and Scogings, C.J., Roles of Rule-Priority Evolution in Animat Models Proc. Second Australian Conference on Artificial Life (ACAL 2005), Sydney, Australia, 2005.
- [8] Robertson A.R. and Ibbett, R.N., HASE: A Hierarchical Architecture Design and Simulation Environment for Computer Architects, UKSS '93, Keswick, UK, UK Simulation Society, 1993.
- [9] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*, Prentice Hall Series in Artificial Intelligence, 2003.