Universidade de Coimbra
Faculdade de Ciências e Tecnologia
Departamento de Engenharia Informática

Evolutionary Computational Intelligence for Multi-Agent Simulations

Telmo de Lucena Torres de Menezes

Coimbra
December 2008

Thesis submitted to the
University of Coimbra
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Informatics Engineering

Evolutionary Computational Intelligence for Multi-Agent Simulations

Telmo de Lucena Torres de Menezes

University of Coimbra
Faculty of Sciences and Technology
Department of Informatics Engineering
December 2008

Financial support by Fundação para a Ciência e Tecnologia (FCT)
through PhD grant SFRH/BD/19863/2004

This work was performed
under the supervision of

Doctor Ernesto Jorge Fernandes Costa
Full Professor
of the Department of Informatics Engineering
of the Faculty of Sciences and Technology
of the University of Coimbra

Agradecimentos

Sinto-me profundamente privilegiado pela ajuda que recebi de diversas pessoas nesta fase da minha vida. Neste aspecto, não podia ter tido mais sorte.

Agradeço ao meu orientador, o Professor Ernesto Costa, pela liberdade que me concedeu ao longo do percurso que resultou nesta tese, assim como pela amizade e sabedoria com que me guiou nos momentos decisivos. Teve uma contribuição de valor inestimável no meu crescimento científico e também humano.

Durante este período, tive o prazer de pertencer a um pequeno, mas muito especial, grupo de investigação. Agradeço a todos os meus amigos do ECOS, por tudo o que aprendi com eles e por tantas conversas estimulantes.

Não sei se há forma adequada de expressar a minha gratidão às pessoas mais próximas de mim, sem as quais nada disto seria possível. Aos meus pais agradeço, não só a ajuda material ao longo da minha vida, mas ainda mais os valores que me transmitiram. Gosto de pensar que herdei alguma da honestidade, coragem e persistência da minha Mãe e alguma da curiosidade científica, pensamento filosófico e humor do meu Pai.

A Gisela, minha companheira de vida, foi quem mais sofreu com a turbulência emocional que me causou esta viagem. Nunca esquecerei o apoio incondicional que recebi dela. Ela é capaz de acreditar em mim mesmo quando eu não acredito.

Coimbra, 25 de Outubro de 2008

Telmo Menezes

Abstract

The growing interest in multi-agent simulations, influenced by the advances in fields like the sciences of complexity and artificial life is related to a modern direction in computational intelligence research. Instead of building isolated artificial intelligence systems from the top-down, this new approach attempts to design systems where a population of agents and the environment interact and adaptation processes take place.

We present a novel evolutionary platform to tackle the problem of evolving computational intelligence in multi-agent simulations. It consists of an artificial brain model, called the *gridbrain*, a simulation embedded evolutionary algorithm *(SEEA)* and a software tool, *LabLOVE*.

The gridbrain model defines agent brains as heterogeneous networks of computational building blocks. A multi-layer approach allows gridbrains to process variable-sized information from several sensory channels. Computational building blocks allow for the use of base functionalities close to the underlying architecture of the digital computer. Evolutionary operators were devised to permit the adpative complexification of gridbrains.

The SEEA algorithm enables the embedding of evolutionary processes in a continuous multi-agent simulation in a non-intrusive way. Co-evolution of multiple species is possible. Two bio-inspired extensions to the base algorithm are proposed, with the goal of promoting the emergence of cooperative behaviors.

The LabLOVE tool provides an object model where simulation scenarios are defined by way of local interactions. The representation of simulation object features as symbols mitigates the need for pre-defined agent sensory and action interfaces. This increases the freedom of evolutionary processes to generate diversified behaviors.

Experimental results are presented, where our models are validated. The role of the several genetic operators and evolutionary parameters is analyzed and discussed. Insights are gained, like the role of our recombination operator in bloat control or the importance of neutral search. In scenarios that require cooperation, we demonstrate the emergence of synchronization behaviors that would be difficult to achieve under conventional approaches. Kin selection and group selection based approaches are compared. In a scenario where two species are in competition, we demonstrated the emergence of specialization niches without the need for geographical isolation.

Keywords: Computational Intelligence; Multi-Agent Simulations; Evolutionary Computation; Genetic Programming; Complex Systems; Artificial Life; Emergence of Group-Behavior; Bloat Control.

Contents

List of Figures

List of Tables

List of Pseudo-Code Listings

Resumo Alargado em Português

Introdução

O crescente interesse em simulações multi-agente, influenciado por avanços em áreas como as ciências da complexidade e a vida artificial, motiva uma nova direcção de investigação em inteligência computacional [Floreano and Mattiussi, 2008]. Em vez de de construir sistemas de inteligência artificial do geral para o particular, esta nova metodologia propõe o desenho de sistemas onde uma população de agentes interage entre si e com o ambiente, sendo que tomam lugar processos de adaptação. Holland propõe que a inteligência pode ser vista como o modelo interno de agentes [Holland, 1995]. Nos sistemas adaptativos, este modelo interno pode desenvolver-se à medida que o agente se adapta ao ambiente. Na natureza, a adaptação e um processo transversal que ocorre a vários níveis, desde a evolução *neo Darwinista* até à aprendizagem no cérebro ou no sistema imunitário. O nosso trabalho incide na adaptação baseada em evolução, que consideramos ser o principal mecanismo de diversificação na natureza.

Objectivos

O principal objectivo desta tese é o estudo de mecanismos de evolução de inteligência computacional em simulações multi-agente. Mais concretamente, focamo-nos em ambientes virtuais contínuos, simulados em tempo real e com dimensões espaciais. Utilizamos uma metodologia descentralizada onde comportamentos globais emergem de interacções locais.

Os objectivos a que nos propomos configuram um conjunto de desafios:

- A concepção de um modelo de cérebro de agente adequado;
- A definição de um interface agente-ambiente que permita a configuração do ambiente a um nível de interacções locais, de objecto para objecto;
- A criação de algoritmos evolucionários eficientes mas que não imponham restrições à simulação, e que sejam capazes de manter ambientes com várias espécies.

O modelo de cérebro do agente deve ter uma representação genética e um conjunto de operadores genéticos que possam ser utilizados por um processo evolucionário na procura de indivíduos de alta qualidade. Esta representação deve ser capaz de expressar estruturas computacionais não-triviais, incluindo processos de decisão, generalização, memória e sincronização. Os requisitos computacionais do cérebro devem ser compatíveis com as restrições impostas por uma simulação em tempo real. O cérebro deve ser capaz de processar informação sensorial proveniente de diferentes fontes e relativa a um número variável de entidades percepcionadas. A dimensão interna e complexidade do cérebro não devem ser pré-determinadas, mas adaptáveis aos desafios colocados pelo ambiente.

Em consonância com um modelo descentralizado e desenhado do particular para o geral, deve ser possível definir interfaces entre o agente e o ambiente, incluindo outros agente e objectos, de uma forma que imponha a menor quantidade possível de pré-definições à simulação. É assim importante limitar estes interfaces a relações um-para-um entre as entidades da simulação.

O algoritmo evolucionário dever ter a capacidade de se adaptar aos requisitos da simulação, e não o inverso. Nos sistemas biológicos é observado que a diversidade de espécies desempenha um papel importante no desenvolvimento de organismos, comportamentos e cérebros cada vez mais complexos. Um ambiente heterogéneo de cooperação e competição entre espécies parece ser um aspecto importante dos processos evolucionários capazes de gerar comportamentos inteligentes. O nosso algoritmo evolucionário, deve assim, suportar simulações com várias espécies.

Um objectivo secundário do nosso trabalho é o desenvolvimento de uma ferramenta de *software* que implemente os algoritmos e mecanismos que iremos propor. Esta ferramenta será disponibilizada à comunidade científica, de forma a que o nosso trabalho possa ser

verificado e expandido. Acreditamos que o nosso trabalho tem aplicações práticas, tanto de natureza científica como de engenharia. A plataforma de simulação que propomos poderá ser aplicada a simulações científicas, nomeadamente nas áreas dos sistemas complexos e da vida artificial. Acreditamos ainda que o nosso trabalho pode servir de base ao desenvolvimento de agentes inteligentes para mundos virtuais e jogos de computador.

Contribuições

Uma nova plataforma algorítmica par a evolução de inteligência computacional é apresentada nesta tese. As principais contribuições consistem nas três partes da plataforma:

- Um modelo evolucionário de cérebro para agentes autónomos, designado *gridbrain*. O gridbrain é uma rede multi-camada de blocos construtores computacionais, capaz de formar decisões baseadas em informação sensorial de dimensão variável, proveniente de múltiplos canais de percepção.

- Um algoritmo evolucionário não geracional (SEEA), capaz de embutir o processo evolucionário em simulações multi-agente contínuas e em tempo real.

- Uma ferramenta de simulação designada *LabLOVE*. O LabLOVE define um modelo de agente baseado em interacções locais e descentralizadas. Inclui ainda um sistema de visualização do ambiente e módulos de recolha de dados estatísticos. Foi desenvolvido um ambiente específico para experimentação com as ideias apresentadas nesta tese. Devido à sua natureza modular, é possível implementar outros tipos de ambientes no LabLOVE.

Conceitos Centrais

Nesta secção apresentamos um conjunto de conceitos e abordagens centrais ao trabalho descrito nesta tese.

Complexidade, Emergência e Auto-Organização

Complexidade, emergência e auto-organização são propriedades altamente relacionadas entre si, que podem ser exibidas por sistemas variados [Axelrod and Cohen, 1999; Holland, 1995; Camazine *et al.*, 2001]. A definição destas propriedades tem-se mostrado difícil, sendo até hoje assunto de intenso debate na comunidade científica.

Um sistema complexo pode ser definido como um sistema constituido por componentes individuais e uma rede de interacções entre eles. Um sistema complexo exibe propriedades que não são aparentes nos componentes individuais que o constituem. Este tipo de estrutura pode ser hierárquica, onde sistemas complexos são componentes de outros sistemas complexos a um nível mais elevado da hierarquia. Tipicamente, o comportamento de sistemas complexos é não-linear e difícil de modelar através da utilização de ferramentas matemáticas convencionais, como as equações diferenciais.

A organização hierárquica está ligada ao conceito de emergência. De facto, é comum em terminologia das ciências da complexidade, dizer-se que uma camada de hierarquia emerge de outra. A emergência pode ser vista como o surgimento de novas propriedades num sistema complexo, sendo que estas propriedades não estão presentes nos componentes individuais do sistema. As interacções formam um sistema de nível mais elevado, diferente dos seus componentes.

A auto-organização é um processo através do qual um sistema, sem controlo do exterior, aumenta a sua complexidade interna. Muitos tipos de mecanismos de auto-organização podem ser encontrados em diversos níveis de abstracção na natureza.

Sistemas Multi-Agente

Um sistema multi-agente pode ser definido como um conjunto de agentes em interacção num dado ambiente. Este tipo de sistema permite a modelação de fenómenos descentralizados, tendo portanto uma diversidade de aplicações em áreas onde este tipo de fenômeno é relevante, nomeadamente a biologia [Adamatzsky and Komosinski, 2005], as ciências sociais [Schelling, 1971; Deffuant, 2006], a economia [Tesfatsion, 2002; Izquierdo and L.R., 2006; Borrelli *et al.*, 2005; Caillou and Sebag, 2008] e a inteligência artificial [Steels, 2004; Schut, 2007].

Um tipo de sistema multi-agente de especial relevância para a área da inteligência artificial é aquele que é constituído por agentes inteligentes. Neste contexto, um *agente inteligente* é caracterizado pelas seguintes propriedades: *autonomia*, *capacidade social*, *rectividade* e *proactividade* [Wooldridge and Jennings, 1995]. A autonomia significa que não existe nenhum controlador central no sistema que determine as acções do agentes. A capacidade social é a capacidade de interagir com outros agentes no ambiente. Reactividade é a capacidade de reagir a estímulos do ambiente e proactividade é a capacidade de tomar decisões de acordo com objectivos.

Outro sub-conjunto dos sistemas inteligentes relevante para esta tese é o dos sistemas evolucionários multi-agente. Neste tipo de sistema, os agentes sofrem um processo adaptativo estocástico inspirado na evolução Darwinista. O trabalho apresentado foca-se em sistemas multi-agentes que são simultâneamente evolucionários e baseados em agentes inteligentes.

Sistemas Complexo Adaptativos

A área dos sistemas complexo adaptaticos (CAS) tem o objectivo de abstrair processos fundamentais encontrados a diversos níveis e que apresentam uma característica unificadora: coerência perante a mudança. Os CAS são um tipo de sistema complexo. Num modelo CAS, os componentes constituintes do sistema são agentes. Num esforço de teorização, John Holland propõe sete características fundamentais que definem um CAS: agregação, não-linearidade, fluxos, diversidade, marcadores, modelos internos e blocos constructores [Holland, 1995].

Vida Artificial

A vida artificial é uma área que abrange uma diversidade de abordagens, procurando estudar os processos da vida através da replicação, em meios artificiais, de fenómenos biológicos. Estas abordagens resultam de uma combinação de conceitos biológicos e das ciências da computação. A vida artificial partilha objectivos e abordagens com os sistemas adaptativos complexos e sistemas multi-agente. Alguns trabalhos podem ser considerados como pertencentes a estas três áreas, como é o caso da simulação ECHO [Holland, 1995].

Um dos objectivos por atingir da vida artificial é a criação de sistemas artificiais em *evolução aberta*. A evolução aberta pode ser definida como a capacidade de um sistema evolucionário de produzir perpétuamente inovação [Standish, 2003]. É comum associar a evolução aberta à ausência no processo evolucionário de objectivos explicitamente definidos. Esta associação está, no entanto, por demonstrar.

Algoritmos Evolucionários

Os algoritmos evolucionários [Eiben and Smith, 2008] são inspirados nos processos evolucionários biológicos. Formulam abstracções sobre a evolução Darwinista e a genética de Mendel. O aplicação mais comum destes algoritmos é a optimização e pesquisa de soluções em problemas com explosão combinatória, mas são usados em outras áreas como a criação de programas de computador, a engenharia de sistemas e a criatividade artificial.

De uma forma genérica, um algoritmo evolucionário cria gerações sucessivas de populações. Estas populações são constituidas por indivíduos que representam possíveis soluções no domínio de um problema. A qualidade de cada indivíduo é quantificada por uma função de aptidão, que é especificada para o problema em causa. Os indivíduos com maior aptidão têm uma probabilidade mais elevada de propagar a sua informação genética, uma vez que é mais provável que seja seleccionados como progenitores ou sobreviventes para a geração seguinte. A reprodução com variação, permitida pelos operadores genéticos, gera diversidade na população. A combinação da preferência selectiva por indivíduos de maior aptidão com a diversidade na população pode permitir ao algoritmo uma pesquisa eficiente do espaço de soluções. Os indivíduos têm uma representação, ou *genótipo*, que codifica uma solução ou *fenótipo*. Os operadores genéticos actuam no espaço dos genótipos, enquanto que a função de adaptação avalia fenótipos. Se definirmos uma forma de organizar os genótipos num espaço n-dimensional, a inclusão de uma dimensão para os valores de aptidão associados a cada genótipo define uma *paisagem de aptidão* a $n + 1$ dimensões. A pressão selectiva para encontrar soluções de maior aptidão leva a população a escalar cumes de aptidão, enquanto que a diversidade tende a impedir que a população fique presa em cumes sub-óptimos ou *máximos locais*.

A grande variedade de algoritmos evolucionários advém da diversidade de abordagens

possíveis para os diversos aspectos do mecanismo: representação genética, operadores genéticos e processo de selecção de progenitores e sobreviventes.

As representações pode ser directas ou indirectas [Tavares, 2007]. Nas representações directas, não existe distinção entre o genótipo e o fenótipo. As representações indirectas utilizam uma diversidade de codificações. Uma possível vantagem das representações indirectas é o aumento da redundância. A correspondência de múltiplos genótipos ao mesmo fenótipo permite a existência de mutações neutrais. As mutações neutrais alteram o genótipo mas não o fenótipo, tendo sido demonstrado que podem ter um impacto positivo no desempenho de algoritmos evolucionários [Galván-López and Rodríguez-Vázquez, 2006]. Mesmo no domínio da biologia, há proponentes das mutações neutrais como parte fundamental dos processos evolucionários [Kimura, 1983].

Os tipos mais comuns de operadores são os de recombinação e de mutação. Os operadores de recombinação estão relacionados com a reprodução sexuada em organismos biológicos. Este tipo de operador combina material genético de dois ou mais progenitores para gerar os descendentes. Os operadores de mutação causam alterações no genótipo. Os mecanismos específicos destes operadores são muito dependentes da representação em uso.

Os algoritmos evolucionários são classicamente divididos em quatro ramos principais: Algoritmos Genéticos [Holland, 1975], Estratégias Evolucionárias [Beyer and Schwefel, 2002], Programação Evolucionária [Fogel *et al.*, 1966] e Programação Genética [Koza, 1992; Poli *et al.*, 2008]. Os primeiros dois estão vocacionados para a resolução de problemas de pesquisa e optimização combinatória. Os últimos dois, com destaque para a Programação Genética, são mais relevantes para o trabalho apresentado nesta tese, uma vez que visam a evolução de programas de computador.

Em programação genética (GP), são evoluidas populações de programas de computador. A aptidão de um indivíduo é determinada pela execução de programa de computador que ele representa, e comparação dos resultados produzidos pelo programa com os resultados desejados. Um desafio significativo na evolução de programas de computador é a sua representação. Um problema semelhante existe na definição de linguagens de programação para humanos. As linguagens de programação criam um nível de abstracção sobre o código

máquina, sendo mais amigáveis à forma de pensar dos programadores humanos. Da mesma forma, são definidas representações que facilitem a acção de operadores genéticos. Diversas representações foram sugeridas na área da programação genética, sendo a mais popular a baseada em árvores. O trabalho apresentado nesta tese está no entanto mais relacionado com as representações baseadas em grelhas.

Na programação genética baseada em árvores, os programas são representados como *árvores sintáticas*. Esta representação está fortemente relacionada com a programação funcional. Cada nó da árvore representa uma função ou um terminal. Os nós de função ramificam para nós mais profundos. Os nós a que uma função está ligada num nível mais profundo são os seus parâmetros, que podem ser terminais ou outras funções. Os terminais são geralmente valores constantes ou valores de entrada. Os operadores genéticos mais usados são o *cruzamento de subárvores* e a *mutação de subárvore*. O cruzamento de subárvores é um operador de recombinação. Um nó de cruzamento é seleccionado aleatoriamente na árvore de cada progenitor. Estes nós são as raízes das subárvores seleccionadas. Os descendentes são gerados através da cópia da árvore do primeiro progenitor e substituição da subárvore nele seleccionada pela subárvore do segundo progenitor. A mutação de subárvores consiste na selecção aleatória da raiz de uma subárvore num indivíduo e substituição por uma subárvore gerada aleatoriamente.

A representação baseada em árvores, apesar de ter sido aplicada com sucessos a uma diversidade de problemas, tem limitações importantes. Uma delas é a sua natureza sequencial. As representações baseada em grelha foram criadas com o intuito de tornar possível a programação distribuída e paralela. É o caso da *programação genética paralela e distribuída (PDGP)* [Poli, 1996, 1999]. Em PDGP, os programas são representados como grafos com nós legendados e ligações orientadas. Os nós dos grafos representam funções ou terminais, enquanto que as ligações representam os parâmetros das funções. Os nós do grafo são dispostos numa grelha multi-dimensional com tamanho pré-definido.

Outra abordagem baseada em grelhas é a *programação genética cartesiana (CGP)* [Miller, 1999; Miller and Thomson, 2000]. Tem objectivos semelhantes à PDGP mas utiliza uma representação diferente. Em PDGP existe pouca distinção entre genótipo e fenótipo. A CGP utiliza uma representação mais indirecta. Assim como em PDGP, os nós são

dispostos numa grelha, mas o genótipo consiste em duas listas de inteiros de tamanho fixo.

Um conjunto de parâmetros é usado para definir a topologia da grelha: $P = \{n_i, n_o, n_n, n_j, n_r, n_c, l\}$. O significado destes parâmetros é o seguinte: n_i é o número de entradas do programa, n_o é o número de saídas, n_n é o número de entradas por função, n_j é o número de funções, n_r é o número de funções por linha, n_c é o número de funções por coluna e l é o número máximo de colunas entre a origem e o destino de uma ligação. Estes valores mantêm-se fixos ao longo do processo evolucionário. Os conjuntos de inteiros no genótipo são G e F, onde G contém os valores que representam ligações e F contém os valores que determinam que função existe em cada posição da grelha.

Além de problemas convencionais, a CGP foi aplicada com sucesso à evolução de controladores de robots [Harding and Miller, 2005]. Foram gerados controladores com a capacidade de evitar obstáculos e navegar em labirintos.

Neuroevolução

As *redes neuronais artificiais* [Costa and Simões, 2008] são um modelo computacional inspirado no sistema nervoso central dos animais vertebrados. Trata-se de um modelo conexionista, onde a computação é realizada através da propagação e processamento de informação numa rede, onde os nós são *neurónios artificiais* e as ligações têm pesos.

Os neurónios artificiais são unidades simples de processamento, constituidas por três funções: entrada, activação e saída. Geralmente, a função de entrada é o somatório dos sinais de entrada multiplicados pelo peso das respectivas ligações. As funções de activação mais usadas são a linear, a linear saturada, o degrau e a sigmóide. Tipicamente, a função de saída devolve apenas o valor produzido pela função de activação. As redes neuronais podem ter um diversidade de topologias, e permitir apenas propagação para a frente ou também ligações recorrentes.

As redes neuronais pode ser evoluidas, através do uso de algoritmos evolucionários. Este processo é designado por *neuroevolução*(NE) [Angeline *et al.*, 1993; Moriarty, 1997; Siebel and Sommer, 2007]. Dois aspectos das redes podem ser evoluidos: os pesos das ligações e a topologia. As abordagens inicias evoluem apenas os pesos, tornando o pro-

blema mais simples [Moriarty, 1997; Gomez and Miikkulainen, 1998; Igel, 2003]. Estas abordagens têm no entanto limitações importantes. É necessária intervenção humana para determinar uma topologia adequada ao problema. As soluções obtidas ficam limitadas a esta topologia, o que pode impedir a descoberta de soluções interessantes. Existem além disso problemas de escalabilidade. Domínios de apicação que requiram topologias complexas podem resultar em espaços de procura demasiado grandes.

Existem abordagens mais recentes que permitem a evolução simultânea das topologias e dos pesos, como é o caso da *neuroevolução de topologias aumentativas (NEAT)* [Stanley, 2004; Stanley and Miikkulainen, 2004] e a *codificação genética analógica (AGE)* [Mattiussi, 2005].

Gridbrain

O sistema que controla o comportamento individual de um agente é um aspecto central em simulações multi-agente. Usando uma analogia com as formas de vida animais que podemos encontrar na natureza, referimo-nos a esse sistema como o *cérebro* do agente. A investigação em inteligência artificial tem vindo a gerar uma variedade de abordagens para a criação de cérebros de agentes. Mesmo no caso mais específico das simulações multi-agente evolutivas, diversas abordagens são usadas. Nesta secção propomos um novo modelo, o *gridbrain* [Menezes and Costa, 2007b, 2008b,a], que foi concebido com os seguintes objectivos:

- Ter a capacidade de realizar computações genéricas;
- Aproveitar a arquitectura do computador digital;
- Não requerer processamento intensivo;
- Ser capaz de processar informação de dimensão variável, proveniente de diversos canais sensoriais;
- Usar uma representação que permita melhorias incrementais, através de um processo evolucionário;

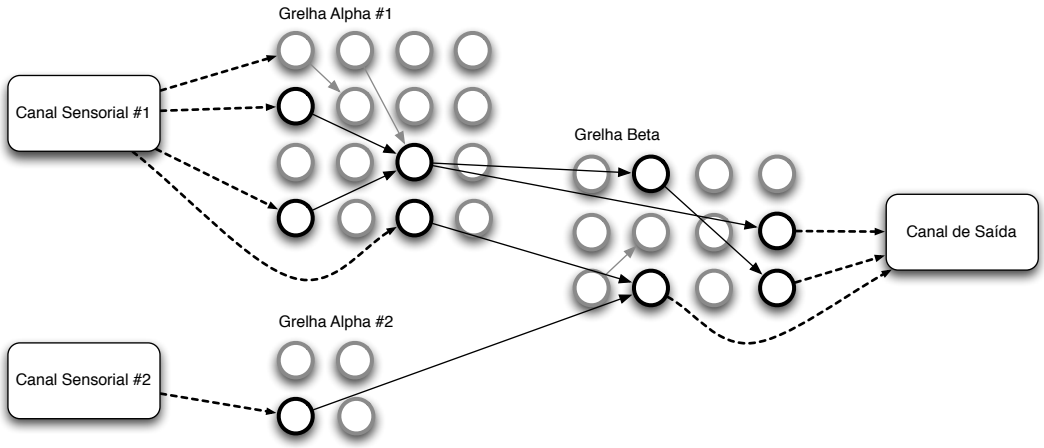

Figura 1: Modelo computacional do gridbrain.

- Ter complexidade adaptável, evitando o *bloat*;
- Produzir resultados compreensíveis para seres humanos.

Propomos um sistema que pertence à família da programação genética, e que procura atingir estes objectivos.

Modelo Computacional

O gridbrain é uma máquina virtual inspirada na arquitectura de von Neumann. Foi concebido para ser o cérebro de agentes autónomos. Como pode ser observado na figura 1, consiste numa rede de componentes computacionais dispostos em grelhas rectangulares. Existem dois tipos de grelha: *alpha* e *beta*. As grelhas alpha estão associadas a canais sensoriais e são responsáveis pelos processamento de informação perceptual. A grelha beta recebe informação das grelhas alpha e produz decisões. Um gridbrain pode ter um qualquer número de grelhas alpha (uma por cana sensorial), mas apenas uma grelha beta. Esta arquitectura é inspirada na organização do cérebro dos animais, que têm zonas dedicadas ao processamento de informação sensorial e outras à tomada de decisões, planeamento e controlo muscular. Não pretende ser um modelo rigoroso destes cérebros, mas apenas uma aplicação do conceito de camadas dedicadas numa rede de componentes. Também não pretende ser um modelo reducionista com hierarquias rígidas. A única limitação imposta é a de que a informação sensorial de entrada seja fornecida às grelhas alpha, e

que a informação de saída seja retirada da grelha beta. Como iremos detalhar mais à frente, isto permite que o gridbrain lide com informação sensorial de dimensão variável, enquanto que o processo evolucionário tem liberdade para testar uma grande diversidade de estruturas.

As ligações entre componentes representam fluxos de informação. Uma ligação do componente A para o componente B significa que, durante cada ciclo de computação, o valor de saída do componente A é usado como valor de entrada do componente B. A informação é propagada sob a forma de valores de vírgula flutuante de grandeza ilimitada. Dois tipos de ligações são possíveis: ligações para a frente dentro de uma grelha, e ligações de qualquer componente de uma grelha alpha para qualquer componente da grelha beta. Nas redes neuronais, as ligações recorrentes permitem que o sistema conserve informação sobre o passado. As redes neuronais constituidas apenas por ligações para a frente permitem modelar apenas agentes puramente reactivos. No gridbrain, o sistema está dotado de mecanismo explícitos de memória. Os componentes podem ter a capacidade de conservar o seu estado interno ao longo dos ciclos de computação. Assim sendo, e para os estudos realizados neste trabalho, optámos pela restrição a ligações para a frente para maior simplicidade.

Em cada ciclo de computação do gridbrain, o valor de saída de cada componentes é processado por ordem. Definimos um sistema de coordenadas onde cada possível posição para um componente é identificada por um tuplo (x, y, g), sendo x o número da coluna, y o número da linha e g o número da grelha. Os componentes são processados da primeira para a última coluna, e dentro de cada coluna, da primeira para a última linha. A restrição de ligações para a frente é imposta usando a regra de, dentro da mesma grelha, permitir apenas ligações para um componente que tenha um número de coluna mais alto que o do componente de origem. A forma rectangular das grelhas facilita a definição de processos de computação paralelos e também sequenciais.

Tanto os componentes como as ligações podem encontrar-se num estado activo ou inactivo. Na figura 4.1, as ligações e componentes activos estão representados a preto, e os inactivos a cinzento. Este estado activo/inactivo não é explicitamente codificado na representação genética do gridbrain, mas extraído da configuração da rede. Para descrever

este mecanismo, necessitamos primeiro de introduzir os conceitos de *componente produtor* e *componente consumidor*. Os componente produtores são aqueles introduzem informação no gridbrain, enquanto que os componentes consumidores são os que enviam informação do gridbrain para o exterior. Os *componentes de entrada* são os componentes das grelhas alpha que introduzem informação sensorial na rede, em cada ciclo de computação. Estes componentes são produtores. Outros componentes que produzem valores diferentes de 0 na ausência de estímulos também são considerados produtores. Exemplos disto são componentes que geram valores aleatórios, ou componentes que produzem o valor 1 no caso da soma dos valores das suas entradas ser 0. Os *componentes de saída* são os componentes da grelha beta de onde é extraída a informação correspondente a decisões. Num contexto de agentes, estão associados com o disparo de acções. Os componentes de saída são consumidores. Um *caminho activo* é uma sequência de ligações que liga um componente produtor a um componente consumidor. Uma ligação é considerada activa se pertencer a um caminho activo. Um componente é considerado activo se tiver pelo menos uma ligação de entrada ou de saída que esteja activa.

A computação levada a cabo pelo gridbrain é determinada apenas pelas ligações e componentes activos. De uma maneira genética, podemos afirmar que os elementos activos são aqueles que têm expressão fenotípica, enquanto que os elementos inactivos não se exprimem no fenótipo. As experiências que realizámos, e que iremos discutir mais à frente, demonstraram a importância dos elementos inactivos no processo evolucionário.

```
para cada grelha g em conj_grelhas:
    seq = seq_components da gelha g
    desde x = 0 a largura de g:
        desde y = 0 a altura de g:
            comp = componente de g na posição (x, y, g)
            se comp activo:
                insere comp no fim de seq
```

Listagem 1: Processo de sequencialização de gridbrains.

Depois de um gridbrain ser gerado, sofre um processo de sequencialização. O algoritmo de sequencialização é apresentado na listagem 1. Este processo consiste na criação, para cada grelha, de uma lista ordenada de componentes. Esta ordenação corresponde à

sequência do processamento que ocorre durante um ciclo de computação, e é determinada pelas posições na grelha. Os componentes inactivos são ignorados.

Durante um ciclo de computação, estas sequências de componentes são processadas. Existem duas fases de processamento, alpha e beta. Na primeira fase, apenas as grelhas alpha são processadas, uma vez por cada entidade que estiver presente no canal sensorial a que estão associadas. Na segunda fase, a grelha beta é processada uma vez. Durante a fase alpha ocorrem duas passagens, de forma a que certos componentes, a que chamamos *agregadores*, tenham a possibilidade de calcular as suas saídas com base na totalidade da informação disponível sobre todas as entidade presentes no canal sensorial. Um exemplo disto é o componente de maximização, que produz um valor de 1 à sua saída, se o valor de entrada actual for o mais alto de todos os valores de entrada, presentes durante este ciclo de computação, ou 0 no caso contrário. Desta forma, os agregadores podem calcular o seu estado interno durante a primeira passagem, e produzir as saídas daí resultantes durante a segunda. Durante a primeira passagem, as ligações inter-grelha encontram-se inactivas, uma vez que os valores correctos de saída das grelhas alpha ainda não foram determinados. Ainda usando o exemplo do maximizador, a primeira passagem é usadas para determinar o valor máximo, enquanto que a segunda é usada para sinalizar os casos em que este valor é reencontrado.

Modelo de Componente

Os componentes são unidade de processamento de informação, e são os blocos construtores computacionais do gridbrain. O modelo genérico do gridbrain não define um conjunto específico de componentes. De forma semelhante às instruções de código máquina, os componentes pertencem a diversas classes de funcionalidade: entrada/saída, aritmética, lógica booleana, agregação de informação, sincronização e memória. O conjunto específico de componentes a utilizar pode ser determinado em função de ambientes específicos ou dos tipos de problemas em causa. Neste trabalho, utilizamos um conjunto de componentes concebido para o controlo de agentes em mundos contínuos com simulação física.

Os componentes possuem um número arbitrário de entradas e saídas, o que confere um elevado grau de liberdade às possíveis topologias da rede. O gridbrain difere neste

Tipo de Componente	**Persistência do Estado**
Operador	Processamento da grelha
Agregador	Fase alpha
Memória	Tempo de vida do gridbrain

Tabela 1: Classificação de componentes segundo a persistência do seu estado interno.

aspecto das abordagens tradicionais de programação genética, onde as funções têm um número pré-determinado de parâmetros, que tem que ser respeitado para que os programas resultantes sejam válidos. Procuramos definir um modelo conexionista mais próximo dos cérebros encontrados na natureza.

O gridbrain foi desenhado seguindo uma abordagem orientada a objectos, onde os interfaces dos componentes são definidos por uma classe base abstracta. Os componentes são assim implementados derivando classes desta classe base. Os seus mecanismos internos dos componentes são implementados nas nestas classes derivadas. Podemos assim ter gridbrains constituidos por uma conjunto heterogéneo de componentes, e por outro lado permitir que o processo evolucionário e o algoritmo de ciclo de computação tratem os componentes como caixas pretas, dotadas de interfaces compatíveis entre si.

Os componentes possuem três interfaces: *limpeza*, *entrada* e *saída*. O interface de limpeza é utilizado para sinalizar aos componentes que um novo ciclo de processamento da sua grelha começou. O interface de entrada é utilizado para fornecer valores provenientes de componentes anteriores, e o interface de saída é utilizado para obter os valores que serão enviados aos componentes seguintes.

Os componentes têm um estado interno. A estrutura de dados que mantém este estado é definida especificamente para cada tipo de componente. Classificamos os componentes como *operadores*, *agregadores* ou *memórias*, consoante a forma como o seu estado interno persiste, conforme é mostrado na tabela 1.

Os operadores são unidade de processamento de informação que realizam, por exemplo, operações aritméticas ou booleanas. A sua saída é determinada apenas pela informação presente nas suas entradas durante cada ciclo de processamento de uma grelha. Os agregadores extraem informação geral de um conjunto de entidades presentes no canal sensorial de uma grelha alpha, e o seu valor de saída é determinado pelos valores recebidos durante

a fase alpha. As memórias conservam o seu estado ao longo dos ciclos de computação do gridbrain, e permitem que seja mantida informação sobre o passado.

Os componentes têm ainda um valor de parâmetro, que é um número de vírgula flutuante pertencente ao intervalo $[0, 1]$. Os componentes podem usar este valor para ajustar o seu comportamento. Por exemplo, um componente amplificador pode usar o valor de parâmetro para determinar o seu factor de amplificação, ou um relógio para determinar a sua frequência de disparo.

Ciclo de Computação

Um ciclo de computação do gridbrain é executado cada vez que se pretende fornecer informação sensorial e obter um decisão. O pseudo-código que descreve o ciclo de computação é apresentado na listagem 2.

Neste código, podemos ver como as funções de interface dos componentes são chamadas durante um ciclo. Note-se que os componentes de entrada têm um campo especial, *tipo_de_entrada*, que é usado para identificar o tipo de informação que recebem do canal sensorial a que a sua grelha está associada. Da mesma forma, os componentes de saída tem um campo especial, *tipo_de_saida*, que identifica o tipo de informação de saída a que estão associados. Num cenário típico de utilização do gridbrain enquanto controlador de agentes autónomos, cada tipo de saída está associado a uma acção que o agente pode executar.

Representação

O gridbrain foi concebido para ser usado em ambiente de computação evolutiva, pelo que é necessário definir a sua representação. É sobre esta representação que os operadores genéticos actuam.

Um gridbrain com N grelhas e M ligações é representado como uma sequência de N segmentos de grelha e M segmentos de ligação. Ambos os tipos de segmento são estruturas de dados compostas por diversos campos.

O segmento de uma grelha contém o conjunto de componentes por ela utilizados. Este conjunto é usado pelos operadores genéticos para gerar ou substituir componentes

```
// Fase alpha
para cada grelha alpha g em conj_grelhas:
    para pass em [0, 1]:
        para cada entidade no canal sensorial de g:
            seq = seq_componentes da grelha g
            para cada comp em seq:
                comp.reset(pass, entity)
                se comp é entrada:
                    tipo_entrada = comp.tipo_entrada
                    valor = valor para tipo_entdada da entidade
                    comp.input(valor, 0)
                saída = comp.output()
                para cada ligação lig de comp:
                    se pass == 1 ou lig é interna à grelha:
                        comp_dest = componente de destino de lig
                        comp_dest.input(saída, comp.id)

// Fase beta
g = grelha beta
seq = seq_componentes da grelha g
para cada comp em seq:
    comp.reset(0, 0)
    saída = comp.output()
    se comp é saída:
        vector_saidas_gridbrain[comp.tipo_saída] = saída
    para cada ligação lig de comp:
        comp_dest = componente de destino de lig
        comp_dest.input(saída, comp.id)
```

Listagem 2: Ciclo de computação do gridbrain.

na grelha. O segmento contém também a largura (W) e altura (H) da grelha. Contém ainda uma matriz de segmentos de componente, que definem o componente específico que se encontra associado a cada posição na grelha. Esta matriz tem, assim, o tamanho $W.H$. O segmento do componente da grelha com as coordenadas x, y pode ser encontrado na posição $p = x.H + y$ da matriz. O segmento de grelha contém ainda duas listas de identificadores de coluna/linha. Estes identificadores são utilizados pelos operadores genéticos para encontrar posições equivalentes entre grelhas. O segmento de componente tem três campos. O primeiro é um valor inteiro que determina o tipo do componente. O segundo é um valor de vírgula flutuante que determina o parâmetro e o último e um valor booleano que armazena o estado do componente (se está activo ou inactivo).

O segmento de ligação tem seis campos inteiros que especificam as coordenadas de origem e destino da ligação. Tem também um campo booleano que armazena o estado da ligação (activa ou inactiva). Tem ainda três campos para marcadores de ligação. Os marcadores de ligação são um mecanismo utilizado para determinar equivalências entre ligações.

Durante o processo de clonagem e recombinação de gridbrains, é necessário determinar a equivalência entre colunas e linhas. Como será descrito mais à frente, os operadores genéticos podem alterar a forma de uma grelha, através da inserção e remoção de colunas e linhas. Daqui resulta uma impossibilidade de determinar equivalências usando apenas o sistema de coordenadas. Concebemos um mecanismo baseado em árvores binárias que permite a identificação de colunas ou linhas através da história de inserções e remoções.

Cada coluna e linha de uma grelha recebe um identificador (ID), que é uma estrutura de dados com dois campos: *profundidade* e *ramo*. Estes valores determinam a posição da linha ou coluna na árvore binária. São mantidas duas árvores distintas para cada grelha: uma para as linhas e outra para as colunas. Referimo-nos genericamente ás linhas ou colunas, representada por nós nestas árvores, como *elementos*.

Existe uma correspondência entre as posições dos elementos na árvore e na sequência a que pertencem. É possível extrair uma ordenação única de elementos da estrutura de uma árvore, e formar assim as sequências de linha e colunas das grelhas. Ao primeiro elemento inserido numa árvore é atribuído o ID $(0, 0)$. Aos elementos inseridos posteriormente, são

atribuidos IDs relativos as IDs já existentes. Os ramos da árvore exprimem relações de posicionamento relativo na sequência. Suponhamos que temos um nó para um elemento A com dois ramos ligados, respectivamente, aos nós dos elementos B e C. B está no primeiro ramo e C no segundo. Isto significa que, na sequência, B está posicionado antes de A e C depois de A. Assim, se contiver apenas estes três nós, a árvore traduz-se na sequência BAC. Suponhamos agora que queremos inserir um novo elemento, D. A sua posição na árvore será determinada pela posição que desejamos que tenha na sequência. Podemos optar por o inserir antes ou depois de qualquer dos elementos já presentes na sequência. Para isso, atribuímos-lhe um ID que o coloque no primeiro ou segundo ramo de um nó já existente na árvore.

Os valores do ID de um elemento inserido antes de um dado elemento são dados pelas expressões:

$$profundidade_{antes} = profundidade + 1$$

$$ramo_{antes} = 2.ramo$$

Os valores do ID de um elemento inserido depois de um dado elemento são dados pelas expressões:

$$profundidade_{depois} = profundidade + 1$$

$$ramo_{depois} = 2.ramo + 1$$

Como se pode ver, os elementos inseridos com uma posição relativa a um elemento já existente situam-se sempre a um nível de profundidade a mais que o elemento já existente. O valor de ramo é duplicado devido ao factor de ramificação de 2 de uma árvore binária. O maior número de nós que pode existir no nível de profundidade n de uma árvore binária é de 2^n.

Os operadores genéticos, que serão apresentados mais à frente, requerem as seguintes operações: inserir novo elemento antes do primeiro elemento da sequência, depois do último elemento da sequência e entre elementos existentes. Quando um elemento é inserido entre

elementos existentes, examinamos o ID desses elementos. Se a profundidade do ID do primeiro elemento for igual ou superior à profundidade do ID do segundo, o novo ID é gerado para estar no segundo ramo do primeiro elemento. Caso contrário, é gerado para estar no primeiro ramo do segundo elemento. Desta forma, garantimos que o novo ID não colide com um ID já existente na árvore, e corresponde à posição desejada na sequência.

Durante o processo evolucionário, um indivíduo herda uma ligação dos seus progenitores, os respectivos marcadores de ligação são também herdados. Quando uma nova ligação é criada, é necessário atribuir-lhe um marcador. Neste caso existem duas alternativas: ou é encontrada um ligação equivalente na população, e o seu marcador é copiado para a nova ligação, ou um novo marcador é gerado. Os valores do novo marcador são gerados incrementando uma variável global que mantém o último valor atribuido.

Este processo requer que, quando um gridbrain com novas ligações é acrescentado à população, estas novas ligações sejam comparadas com todas as ligações já existentes nos gridbrains da população, na tentativa de encontrar uma equivalência. Uma ligação sem marcador atribuido é considerada equivalente a uma outra se, da perspectiva da rede de ligações, tanto os seus componentes de origem como de destino forem equivalentes. Duas origens são equivalentes destas perspectiva se as seguintes condições se verificarem:

- Os componentes de origem são do mesmo tipo;
- Nenhum dos componentes de origem tem ligações de entrada, ou, têm em comum pelo menos uma ligação de entrada equivalente.

Do mesmo modo, dois componentes de destino são equivalentes se:

- Os componente de destino são do mesmo tipo;
- Nenhum dos componentes de destino tem ligações de saída, ou, têm em comum pelo menos uma ligação de saída equivalente.

Estas regras configuram um mecanismo construtivo que se interliga com o processo de complexificação progressiva dos gridbrains durante a evolução. Permitem-nos identificar ligações que correspondem à mesma funcionalidade.

Operadores Genéticos

No modelo do gridbrain não especificamos uma algoritmo evolucionário. Definimos apenas um conjunto de operadores genéticos que podem ser utilizados por um algoritmo desse tipo. Mais à frente iremos propor um algoritmo evolucionário apropriado a simulações multi-agente contínuas, e que tira partido destes operadores.

Definiremos três classes de operadores genéticos: mutação, recombinação e formatação. As duas primeiras estão convencionalmente presentes em algoritmos evolucionários. A última é responsável pela adaptação da forma das grelhas ao longo do processo evolucionário, e está relacionada com a complexificação dos gridbrains.

Os operadores de mutação podem actuar ao nível das ligações ou ao nível dos componentes.

Propomos dois pares de operadores ao nível das ligações: *insere/remove* e *divide/junta*. Os operadores em cada par são simétricos, um realizando a operação inversa do outro.

No par insere/remove, o primeiro operador acrescenta uma nova ligação válida ao gridbrain, enquanto que o segundo remove um ligação existente. Estas mutações ocorrem com as probabilidade respectivas de p_a e p_r. As probabilidades são relativas ao número de ligações existentes no gridbrain. Para cada ligação, há uma probabilidade p_a de que uma nova seja gerada. Cada ligação existente tem uma probabilidade p_r de ser removida. Múltiplas ligações podem ser inseridas ou removidas durante um etapa de mutação do gridbrain. Fica assim definido um processo que se adapta ao tamanho da rede. Se $p_a = p_r$, e na ausência de pressão evolutiva, o número de ligações tenderá a permanecer estável. Este mecanismo de simetria é parte das medidas que tomamos para combater o *bloat*.

O operador de divisão reencaminha uma ligação existente através de um componente intermédio. Considerando uma ligação do componente A para o componente B, duas novas ligações serão criadas: uma de A para o componente intermédio C, e uma de C para B. A ligação original é removida. Se A e B estiverem na mesma grelha, o componente C deverá ter um número de coluna mais alto que A e mais baixo que B. Se estiverem em grelhas diferentes, C deverá estar, ou na grelha de origem e ter um número de coluna mais alto que A, ou na grelha de destino e ter um número de coluna mais baixo que B.

Os marcadores de ligação consistem em tuplos (t_c, t_o, t_t), onde t_c é o identificador de

ligação, t_o o identificador da origem e t_t o identificador de destino. Quando uma ligação é dividida, as novas ligações resultantes herdam o valor t_c da ligação original. A primeira ligação nova herda o valor t_o e a segunda o valor t_t. Um novo identificador é gerado, e atribuído a t_t da primeira ligação e a t_o da segunda. Desta forma, os componentes de origem e destino da ligação original mantém-se equivalentes no que toca à comparação de marcadores. Além disso, torna-se possível determinar se duas ligações tiveram origem numa divisão, verificando a igualdade dos identificadores t_c dos seus marcadores. A operações genética inversa de juntar é realizada apenas sobre uma ligação que seja adjacentes a uma ligação seguinte que tenha tido origem na mesma divisão. Uma divisão seguida de uma junção resulta na ligação inicial, com o marcador de ligação original.

As divisões e junções ocorrem com as probabilidades respectivas de p_s e p_j. Novamente, estas probabilidades são por ligação existente. Com $p_s = p_j$ e na ausência de pressão evolutiva, o número de ligações tenderá a manter-se estável. Note-se que, quando um ligação é dividida, apenas a primeira das ligações resultantes será elegível para uma junção, pelos que as probabilidades ficam equilibradas.

Propomos dois operadores genéticos ao nível dos componentes: *alterar componente* e *alterar parâmetro*.

A operação de alterar componente substitui um componente existente por um novo. O novo componente é seleccionado aleatoriamente do conjunto de componentes associado à grelha em questão, e o seu parâmetro é inicializado com um valor aleatório, extraído de uma distribuição uniforme em $[0, 1]$. Estas mutações ocorrem com uma probabilidade de p_c por cada componente do gridbrain. Uma variação deste operador é o de *alterar componente inactivo*, que funciona da mesma forma mas afecta apenas componentes inactivos. Esta variação é menos destrutiva, uma vez que produz apenas mutações neutrais.

A operação de alterar parâmetro modifica o parâmetro de um componente adicionando-lhe um valor x. O resultado é truncado para um intervalo $[0, 1]$. O valor x é extraído de uma distribuição normal $x \sim N(\mu, \delta^2)$, onde a média, μ, é 0.

O operador de recombinação ter que ser capaz de lidar com grelhas e topologias diferentes. O processo que propomos realiza recombinação semântica, tirando partido do mecanismo de marcadores de ligação. Este mecanismo permite-nos identificar ligações ou

grupos de ligações que partilham a mesma funcionalidade. O operador consiste assim nas seguintes etapas:

- Criar o gridbrain descendente, constituido por grelhas com as mesma formas e os mesmos identificadores de coluna/linha que o progenitor A;
- Recombinar os grupos de ligações do parente A e B no descendente;
- Recombinar os componentes dos progenitores A e B no descendente.

Um grupo de ligações é o conjunto das ligações de um gridbrain que possuem o mesmo valor t_c no seu marcador. Um grupo com mais de uma ligação resulta de uma ou mais operações de divisão. Para recombinar ligações, começa-se pelo progenitor A e itera-se através do conjunto das suas ligações. Para cada grupo de ligações encontrado, verifica-se se este grupo está presente no progenitor B. Se estiver presente, importa-se o grupo de um dos progenitores com igual probabilidade. Se não estiver, o grupo pode ser importado ou descartado, com igual probabilidade. De seguida itera-se através do conjunto de ligações do progenitor B. Novamente, verifica-se a presença de cada grupo de ligações no outro progenitor. Se existir, já foi recombinado. Se não, o processo de importar ou descartar com igual probabilidade é aplicado.

Importar um grupo de ligações consiste em importar cada uma das ligações do grupo. Para importar ligações não se usam coordenadas, uma vez que a grelha do progenitor e a grelha de destino podem ter formatos diferentes. Em vez disso, tira-se partido dos identificadores de coluna/linha para determinar a localização correcta da ligação no descendente. Neste processo, traduzem-se tuplos de coordenadas da ligação de origem, (x_{origem}, y_{origem}, g_{origem}, $x_{destino}$, $y_{destino}$, $g_{destino}$), para tuplos de identificadores: ($IDcol_{origem}$, $IDlinha_{origem}$, g_{origem}, $IDcol_{destino}$, $IDlinha_{destino}$, $g_{destino}$). Depois convertem-se os tuplos de identificadores para tuplos de coordenadas na grelha do descendente. No caso de não existir equivalência no descendente, as colunas ou linhas necessárias são criadas.

A última etapa é a recombinação de componentes. Para cada componente do descendente, verifica-se se existe uma ligação equivalente em cada um dos progenitores. Se existir em ambos, um deles é escolhido aleatoriamente, com igual probabilidade, para fornecer o componente para essa posição. Se a posição só existir num dos progenitores, o componente

é copiado deste. Se não existir em nenhum dos progenitores, um componente é selecionado aleatoriamente do conjunto de componentes da respectiva grelha.

O operador de recombinação produz sempre gridbrains válidos. Na ausência de pressão evolutiva, tende a manter um número estável de ligações.

O operador de formatação adapta a forma das grelhas à rede de ligações contida no gridbrain. Configura um mecanismo adaptativo de natureza não-estocástica. As alterações que realiza não afectam o fenótipo dos indivíduos. O propósito da formatação é a regulação do espaço de busca do processo evolucionário. É, além disso, parte do processo de complexificação. Procuramos evoluir populações que são inicializadas com cérebros vazios, e em que soluções mais complexas vão sendo criadas consoante os requisitos do ambiente.

O operador de formatação opera ao nível das grelhas e realiza alterações tendo em conta a *rede activa*, que é o conjunto de todas as ligações activas. Para cada grelha, o operador determina se linhas ou colunas devem ser inseridas, removidas ou mantidas.

Os causadores directos de complexificação no gridbrain são os operadores de mutação ao nível das ligações. A formatação altera as formas das grelhas de maneira a que a rede activa se mantenha inalterada, e a etapa de mutações seguinte tenha liberdade para complexificar a rede de diversas maneiras:

1. Inserir uma ligação válida entre quaisquer componentes activos;
2. Inserir uma ligação interna à grelha com origem em qualquer componente activo;
3. Inserir uma ligação interna à grelha com destino a qualquer componente activo;
4. Ramificar a saída de qualquer componente activo para um componente inactivo em qualquer das colunas seguintes;
5. Dividir qualquer ligação activa.

A primeira condição é garantida pela não alteração da rede activa. A segunda é garantida através da manutenção nas grelhas de uma primeira coluna sem componentes activos, e a terceira pela manutenção de uma última coluna sem componentes activos. A quarta condição é garantida através da manutenção de pelo menos uma linha sem componentes activos. A última condição é garantida mantendo pelo menos uma coluna

intermédia em cada ligação activa. A formatação altera a grelha inserindo e removendo colunas e linhas, com o objectivo de garantir estas condições usando a grelha mais pequena possível.

O operador de formatação deve ser aplicado antes da etapa de mutação de um gridbrain. Desta forma, no momento em que os operadores de mutação actuam, o gridbrain já foi formatado para respeitar as condições descritas. A sequência de operações genéticas a efectuar quando um novo gridbrain é gerado é: $recombina(progenitor1, progenitor2) \rightarrow formata \rightarrow muta$ se a recombinação estiver a ser usada, ou: $clona(progenitor1) \rightarrow formata \rightarrow muta$ para reprodução a partir de apenas um progenitor.

Algoritmo Genético Embebido em Simulação

Os algoritmos evolucionários mais conhecidos são geracionais. Isto significa que um conjunto de indivíduos é criado, avaliado, e um conjunto seguinte é gerado através de seleção, recombinação e mutação de indivíduos do conjunto anterior, atendendo à sua aptidão. Este tipo de abordagem cria problemas em simulações contínuas, onde não é desejável remover todos os agentes do mundo e criar uma população inteiramente nova, introduzindo assim uma descontinuidade. Precismos de um algoritmo que se integre de forma menos intrusiva na simulação.

Propomos o *algoritmo evolucionário embebido em simulação (SEEA)*. Trata-se de um algoritmo de *estado estável*, ou seja, sem gerações. Suporta múltiplas espécies com isolamento reprodutivo, que podem co-evoluir, partilhando o mesmo ambiente. O algoritmo SEEA foi concebido para ser usado com o modelo gridbrain, como parte dos objectivos deste trabalho. Não possui, no entanto, dependências específicas com o gridbrain e pode ser usado com outros modelos que suportem operadores genéticos de mutação e recombinação.

O algoritmo SEEA mantém um repositório de indivíduos, de tamanho fixo, para cada espécies existente. O ambiente de simulação interage com o algoritmo através de chamadas a duas funções: *onRemoval()* e *requestAgent()*. A função *onRemoval()* processa a remoção de agents do ambiente e a função *requestAgent()* gera e envia um novo agente para o

ambiente.

O repositório constitui uma memória genética para a espécie, contendo alguns dos indivíduos de maior aptidão encontrados até ao momento. Assumimos que o ambiente de simulação tem a capacidade de produzir uma avaliação quantitativa da aptidão de um indivíduo, no momento da sua morte. Quando um agente termina o seu período de vida na simulação, a sua avaliação de aptidão é comparada com a de um indivíduo numa posição aleatória do repositório. Se for igual ou maior, passa a ocupar este lugar no repositório. É desta forma que a pressão evolutiva é introduzida no sistema.

Se, numa comparação de aptidão, o indivíduo removido tiver uma aptidão mais baixa que o indivíduo no repositório, toma lugar o processo de *envelhecimento de aptidão*. Isto causa a redução da aptidão do indivíduo no repoistório por um certo factor. Esta redução é calculada usando a expressão:

$$f_{nova} = f.(1 - a),$$

onde f é a avaliação de aptidão do indivíduo e a é o *factor de envelhecimento de aptidão*. O objectivo do mecanismo de envelhecimento de aptidão é duplo: manter o processo evolucionário dinâmico e promover a diversidade.

O algoritmo não controla a dimensão das populações na simulação. Compete à simulação decidir quando remover um agente ou requisitar um novo. Neste trabalho utilizamos sempre populações de dimensão fixa, mas o algoritmo SEEA pode ser directamente aplicado a simulações com populações variáveis.

O algoritmo base é capaz de criar pressão evolutiva de modo a que os indivíduos tendam a melhorar a sua aptidão individual. Tem, no entanto, a limitação de só promover a emergência de comportamentos que beneficiem individualmente os agentes. Isto pode ser insuficiente para a emergência de comportamentos colectivos, onde algum nível de altruísmo é necessário.

Na natureza, diversos exemplos podem ser encontrados de comportamentos cooperativos. Em alguns casos, os indivíduos exibem comportamentos que lhes são individualmente prejudiciais, mas benéficos para o grupo a que pertencem. O aparecimento de altruísmo coloca problemas ao modelo Darwinista clássico. De um ponto de vista biológico dois

modelos explicativos são avançadas: *selecção de parentesco* e *selecção de grupo* [Smith, 1964].

Indivíduos com parentesco partilham, até certo ponto, informação genética. Quanto mais próximo for o parentesco, mais informação genética é partilhada. A selecção de parentesco é baseada na ideia que, um aumento nas probabilidades de sobrevivência e reprodução de um indivíduo também é benéfica para a propagação da informação genética dos indivíduos com quem tem parentesco. Se a proximidade genética foi suficientemente alta, um comportamento altruísta, mesmo que com custo para o indivíduo, pode ser benéfico para a probabilidade de propagação da sua informação genética [Hamilton, 1963, 1964]. A hipótese da selecção de grupo é baseada na ideia de selecção Darwinista ao nível de conjuntos de organismos [Smith, 1964; Williams, 1966; Wilson, 2005].

Propomos duas extensões ao algoritmo SEEA, criadas com o objectivo de promover a emergência de comportamentos colectivos nas espécies: *super irmãs* e *aptidão de grupo*. A primeira é baseada no modelo de selecção de parentesco e a segunda no modelo de selecção de grupo.

A extensão *super irmãs* aumenta a proximidade genética dos agentes que coexistem numa simulação. É inspirada nos insectos sociais [Wilson, 1971], onde as fêmeas têm um elevado grau de proximidade genética, devido a haplodiploidia. Esta extensão altera a função *onRequest()*, para que, em vez de gerar um novo descendente em cada pedido, o faça apenas a cada n_{seq} pedidos. Quando um novo descendente é gerado, é armazenado na estrutura de dados da espécie. A função *onRequest()* em modo super irmãs devolve sempre uma cópia exacta do último descendente gerado. O ambiente é assim povoada por sequências de indivíduos idênticos.

De forma a aumentar ainda mais a proximidade genética, é utilizado um mecanismo de *rotação de progenitores*. Como o SEEA não é baseado em gerações, as sequências de super irmãs vão-se sobrepor, e agentes de sequência consecutivas irão existir simultâneamente no ambiente. A rotação de progenitores aumenta a proximidade genética entre sequências consecutivas. A primeira vez que um descendente é gerado, os seus progenitores são seleccionados como habitualmente. A posição do primeiro progenitor no repositório de espécie é memorizada. A próxima vez que um progenitor for gerado, apenas um progenitor

é seleccionado aleatoriamente do repositório, sendo o outro o que tinha sido memorizado. O posição do que foi gerado aleatoriamente passa a ser a memorizada. Assim, indivíduos de sequências consecutivas terão sempre um progenitor em comum.

Uma variação do sistema de super irmãs é a introdução de mutações nos clones, de forma a que os indivíduos nas sequências não sejam exactamente iguais. Isto permite ao algoritmo evolucionário testar um maior número de variações, mantendo igualmente uma elevada proximidade genética entre indivíduos coexistentes na simulação.

A extensão de *aptidão de grupo* tem o mesmo objectivo que as super irmãs, mas usa uma abordagem diferente. Consiste em fazer o valor de aptidão de um indivíduo depender, em parte, dos valores de aptidão dos outros indivíduos da mesma espécie com que coexistiu na simulação.

O ambiente de simulação e a avaliação de aptidão não têm que ser alterados de nenhuma forma para que a aptidão de grupo seja aplicada. Em vez de se utilizar directamente a aptidão individual determinada pelo ambiente, f_i, uma *aptidão composta* é calculada. A aptidão composta, f_c, é determinada usando a expressão:

$$f_c = (1 - g).f_i + g.f_g,$$

onde f_g é a *componente de aptidão de grupo* e g é o *factor de grupo*. O factor de grupo é um valor real no intervalo $[0, 1]$. Quanto mais alto for, mais importante é o sucesso do grupo para a aptidão composta de cada indivíduo.

O componente de aptidão de grupo reflecte as variações de aptidão em outros indivíduos da mesma espécie, durante o tempo de vida do indivíduo para o qual está a ser calculado. O princípio aplicado é o de reflectir apenas as variações que podem ter sido obtidas com a ajuda deste indivíduo. Uma forma eficiente de calcular o valor do componente de aptidão de grupo é manter um somatório de aptidão de grupo, G, para cada indivíduo. Quando um indivíduo entra no ambiente, o seu valor G é inicializado pela aplicação da expressão:

$$G = -\sum_{a \in S(t)} f_a(t),$$

onde $S(t)$ é o conjunto dos indivíduos que se encontram no ambiente no momento t

e que pertencem à sua espécie, e $f_a(t)$ é a aptidão individual actual do agente $a \in S(t)$, nesse mesmo momento. Durante o tempo de vida de um indivíduo, sempre que outro indivíduo da mesma espécie é removida da simulação, G é incrementado pelo valor de aptidão individual final do indivíduo removido. Quando um indivíduo termina o seu período de vida, o valor G final é determinado por:

$$G' = G + \sum_{a \in S(t)} f_a(t).$$

Desta forma, no fim do seu período de vida, G contém o somatório das variações individuas de aptidão em outros agentes da mesma espécie, durante esse período. Finalmente, o componente de aptidão de grupo é calculado por:

$$f_g = \frac{G}{pop - 1},$$

onde *pop* é o tamanho da população da espécie do individuo no ambiente. Assim, f_g contém a variação individual de aptidão por indivíduo no ambiente.

As extensões de aptidão de grupo e super irmãs são compatíveis entre si e podem ser usadas em simultâneo.

A Ferramenta de Simulação

Por forma a realizar experiências sobre os modelos descritos, desenvolvemos uma ferramenta de simulação a que chamámos *LabLOVE* [Menezes, 2008b]. Esta ferramenta permite-nos realizar simulações multi-agente em tempo real. Inclui implementações do gridbrain e do SEEA. Foi desenhada seguindo uma abordagem modular e orientada a objectos. Os três módulos principais, que são definidos através de classes abstractas, são os seguintes:

- Simulação (classe *Simulation*)
- Objecto da Simulação (classe *SimObj*)
- Dinâmica de População (classe *PopDyn*)

O módulo de simulação é responsável por manter o ambiente. Actualiza o estado dos objectos em cada ciclo de simulação, de acordo com regras pré-definidas. Por exemplo, pode simular física Newtoniana. Computa o resultado das acções efectuadas pelos agentes e fornece-lhes informação sensorial. Inclui ainda um sistema de visualização, que permite a um observador humano seguir o estado da simulação.

Os objectos de simulação são as entidades que povoam o ambiente. Incluem os agentes, que são controlados por gridbrains. As características dos objectos são definidas através de tabelas de símbolos. Cada objecto contém um conjunto de tabelas de símbolos, de acordo com a configuração da simulação. Os símbolos têm tipos, de acordo com a informação que representam. Por exemplo, a cor de um objecto pode ser representada por um símbolo do tipo RGB, que consiste num tuplo de três bytes, um para cada coordenada de cor (vermelho, verde e azul).

O módulo de dinâmica de população define os mecanismos de criação de novos objectos e remoção de objectos do ambiente. O algoritmo SEEA é implementado sob a forma de um módulo de dinâmica de população.

No modelo LabLOVE, um agente é um caso especial de objecto de simulação, sendo controlado por um gridbrain, e tendo a capacidade de percepcionar o ambiente e executar acções. As percepções constituem a informação sensorial que é fornecida ao gridbrain, de acordo com o canal sensorial a que correspondem. As acções e percepções podem estar associadas a símbolos.

Existem diversas formas através das quais um agente pode formar uma percepção. A informação sensorial pode ser recebida directamente do ambiente, de outros objectos, ou do estado interno do próprio agente. Nos primeiros dois casos, a informação sensorial pode ser obtida através da comparação de símbolos. Em qualquer caso, a informação que é fornecida ao gridbrain toma a forma de valores de vírgula flutuante.

A informação sensorial relativa a um estado interno do agente é uma representação numérica desse estado. Um exemplo possível é o seu nível de energia. A informação sensorial recebida de outros objectos é um representação numérica de algum aspecto do estado ou características desses objectos. Exemplos possíveis de informação sensorial relativa ao estado de outro objecto pode ser as suas distância, velocidade ou orientação

relativas. Este tipo de percepção não tem que ser relativa ao ponto de vista do agente. Por exemplo, podemos permitir que os agentes percepcionem os níveis de energia de outros objectos.

Um outro tipo de informação sensorial que resulta da percepção de outros objectos é a relativa a características representadas por símbolos. Neste caso é utilizado um mecanismo onde o símbolo que representa esta característica no objecto é comparado com um símbolo interno do agente. Desta comparação resulta um valor de vírgula flutuante que constitui a informação sensorial a ser fornecida ao gridbrain. Também no caso de percepções sobre o ambiente, a informação sensorial pode ser obtida directamente ou por comparação de símbolos.

As acções também podem estar associadas a símbolos. Neste caso, o símbolo é utilizado para determinar o efeito da acção. Por exemplo, considere-se um agente capaz de pintar marcas no terreno. A acção de pintura pode ser associada a um símbolo interno de cor, determinado assim a cor em que a marca é pintada. Este tipo de agente pode evoluir diversas acções de pintura com cores diferentes.

Para dar um exemplo mais completo, considere-se um espécie de agentes que é capaz de comunicar através de sons. O som é representado pela sua frequência, intensidade e duração. Um tipo de símbolo é definido para representar os sons, consistindo num tuplo (f, i, d). Estes agentes possuem uma tabela de símbolos para sons. As acções de emissão de sons estão associadas a um símbolo de som. Quando executadas, causam a emissão de um som com as características determinadas pelo seu símbolo. O ambiente de simulação fornece a percepção associada a este som a todos os agentes dento de um determinado raio. Os outros agentes podem percepcionar os sons a partir de um tipo de percepção que está também associado a um símbolo na sua tabla de símbolos de som. A respectiva informação sensorial é obtida através do mecanismo de comparação de símbolos, e fornecida ao gridbrain.

Sim2D

Para efeito de experimentação com os modelos anteriormente descritos, desenvolvemos um ambiente de simulação específico, denominado *Sim2D*. A implementação deste ambiente é

realizada através da extensão das classes *Simulation* e *SimObj*, por forma a definir regras, mecanismos e visualizações. O ambiente Sim2D é bidimensional, contínuo e simula física Newtoniana. Os objectos deste ambiente são modelados como corpos circulares. Exitem num mundo rectangular com dimensões pré-definidas.

Na implementação actual, os agentes do ambiente Sim2D têm as seguintes acções disponíveis: impulso, rotação, comer, disparar projecteis e emitir sons. Possuem ainda dois canais sensoriais: visão e audição.

O canal sensorial visual fornece aos agentes informação sobre outros objectos presentes no seu campo de visão. Os tipos de percepção possíveis sobre este canal são: posição, orientação, distância, tamanho, contacto, alvo, linha de fogo e comparação de símbolos. O canal auditivo fornece informação sobre sons recebidos, suportando as seguintes percepções sobre cada som: posição distância, intensidade e comparação de símbolos.

Experiências

O método de experimentação usado foi o de realizar conjuntos de 30 simulações para cada parametrização. A significância estatística dos resultados foi determinada usando ANOVAs Kruskal-Wallis com $p = 0.01$. Foram utilizadas ANOVAs não-paramétricas uma vez que não existe garantia que os dados sigam distribuições normais.

Benchmark de Operadores e Parâmetros Genéticos

Para o efeito de testar a contribuição dos diversos operadores genéticos e efeitos da parametrização, definimos um cenário experimental que denominámos *poison*. Neste cenário, o ambiente é povoado com objectos de dois tipos: agentes e comida. Os itens de comida podem ser nutritivos ou venenosos para os agentes, em graus variados. Os agentes têm à sua disposição um tipo de percepção que lhes permite avaliar se um dado item é nutritivo ou venenoso, e em que grau. A aptidão dos agentes depende da sua capacidade de adquirir e conservar energia, o que é conseguido consumindo itens nutritivos, evitando os venenosos e evitando o desperdício de energia através da optimizando as suas acções.

Foram realizados conjuntos de experiências onde foi testada uma diversidade de valores

de parametrização, nomeadamente de probabilidades de ocorrência dos vários operadores genéticos e de parametrização do SEEA.

Os resultados mostraram que os pares de operadores de mutação, inserir/remover e dividir/juntar, são importantes para o processo evolucionário. A exclusão de qualquer destes pares resultou num impacto negativo significativo na aptidão final dos agentes. Isto indica que ambos os pares produzem alterações topológicas na rede úteis à pesquisa de soluções.

Por outro lado, não foi verificado que o operador de mutação de alteração de parâmetro tivesse utilidade, e a sua presença não altera significativamente as aptidões finais dos agentes. Consideramos que este é um bom resultado, uma vez que quanto menor for o conjunto de operadores necessários, menos parametrização tem que ser feita para executar simulações bem sucedidas.

O operador de mutação de alteração de componente foi testado inicialmente na sua modalidade de alterar apenas componentes inactivos. A importância deste operador foi também verificada. A sua remoção causa uma diminuição significativa na aptidão final dos agentes. Este resultado indica a importância da pesquisa neutral no processo evolucionário, uma vez que esta modalidade do operador produz apenas mutações neutrais. A modalidade do operador que altera qualquer componente produziu resultados comparáveis quando usado com uma probabilidade mais baixa.

Os pares de operadores insere/remove e divide/junta foram concebidos para serem simétricos, por forma a permitir a adaptabilidade da complexidade da rede de ligações, e também como uma mecanismo de prevenção de *bloat*. De facto, a experimentação mostrou que a diminuição da probabilidade dos operadores remove e junta relativamente aos operadores insere e divide resulta em mais *bloat*, apesar de não afectar significativamente a aptidão dos agentes. Enquanto que em todas as outras experiências foi encontrada uma elevada correlação entre a aptidão dos agentes e o número de ligações no gridbrain, neste caso essa correlação foi baixa.

A variação da probabilidade de recombinação não causou impacto significativo na aptidão, mas causou impacto significativo no *bloat*. De facto, verificou-se que o operador de recombinação actua como um mecanismo de controlo de *bloat*. Pelo observação dos dados

gerados pela execução de simulações, concluímos que para experiências onde não se usa recombinação, mutações inúteis tendem a ser acumuladas ao longo do processo evolucionário. O operador de recombinação permite que as mutações inúteis sejam mais facilmente descartadas. Em experiências sem recombinação, a probabilidade de uma mutação anular uma anterior mutação inútil sem causar efeitos adversos é demasiado baixa.

Verificou-se que ambos os parâmetros do algoritmo SEEA, tamanho de repositório e factor de envelhecimento de aptidão, têm impacto significativo na aptidão final dos agentes. Em ambos os casos foram produzidos resultados inferiores para valores muito elevados ou muito baixos.

De modo mais geral, verificou-se a capacidade do processo evolucionário de gerar gridbrains altamente adaptados ao ambiente, a partir da complexificação de uma população inicial de gridbrains vazios.

Sincronização e Cooperação

Definimos dois cenários para realizar experiências sobre a emergência de comportamentos de sincronização e cooperação entre agentes. Nestes cenários testamos as duas extensões do algoritmo SEEA para promoção de comportamentos cooperativos.

O primeiro cenário, denominado *synch*, promove a sincronização de emissão de sons entre os agentes. O segundo, denominado *targets*, promove a cooperação dos agentes na destruição de alvos em movimento através do disparo de projécteis. O disparo de um projéctil não é suficiente para a destruição de um alvo, pelo que os agentes necessitam de cooperar para atingir bons desempenhos.

Foi verificado experimentalmente que, no primeiro cenário, nenhuma cooperação emerge na ausência das extensões, enquanto que no segundo as extensões não são necessárias. No primeiro cenário, ambas as extensões conduzem a resultados de alta qualidade, sem diferenças significativas. No segundo, a extensão de aptidão de grupo produz resultados de alta qualidade, enquanto que a extensão de super irmãs leva a resultados significativamente inferiores. Em ambos os cenários, verificou-se que a extensão de super irmãs provoca uma diminuição significativa de diversidade no repositório da espécies, enquanto que o algoritmo base sem extensões ou com a extensão de aptidão causam níveis de diversidade sem

diferenças significativas.

Na definição de cenários de simulação, pode ser difícil determinar a importância da cooperação ou a necessidade de utilização de uma extensão de comportamento de grupo. Estes resultados indicam que a extensão de *aptidão de grupo* é uma boa escolha para uso genérico, uma vez que parece ser capaz de promover comportamentos de grupo sem prejudicar significativamente a evolução de comportamentos individuais.

Um Cenário de Competição

Definimos um cenário, denominado *battle*, onde duas espécies co-evoluem com objectivos antagónicos. As duas espécies têm definições iniciais idênticas, excepto pela sua cor, sendo uma vermelha e a outra azul. Este cenário constitui um primeiro passo no sentido de definir simulações de vida artificial utilizando a plataforma evolutiva que propomos. A aptidão dos agentes de cada espécie é determinada pela sua capacidade de disparar contra indivíduos da outra espécie. Além disso, estão sempre disponíveis no ambiente itens comestíveis, que os agentes podem consumir para elevar o seu nível de energia.

Foi verificado que este cenário conduz a histórias evolucionárias diversas e imprevisíveis.

Numa das experiências que efectuámos, a espécie vermelha especializou-se em destruir a azul, enquanto que a azul se especializou em consumir item de comida para sobreviver mais tempo. Este é um resultado interessante, uma vez que mostra um fenómeno de especiação a ocorrer por motivos sistêmicos, sem que o ambiente tenha separação geográfica.

Notra experiência, ambas as espécies especializaram-se na destruição de indivíduos da outra espécie, adquirindo também capacidades de consumo de comida. Acabaram por atingir um estado de equilíbrio, com aptidões médias a oscilar em torno do mesmo valor.

De modo mais geral, este cenário aponta para a possibilidade de comportamentos complexos e diversificados serem gerados a partir de funções de avaliação de aptidão simples, através da exploração das interacções entre a função de avaliação de aptidão, as regras do ambiente e a dinâmica de co-evolução entre espécies.

Conclusões

Consideramos que os objectivos a que nos propusemos no início deste trabalho foram atingidos através da concepção e validação experimental de uma plataforma evolucionária. Esta plataforma é composta por um modelo de cérebro, algoritmo evolucionário e ambiente de simulação, respectivamente: o gridbrain, o SEEA e o LabLOVE.

Os principais conceitos subjacentes ao modelo gridbrains são a abordagem multi-camada, com camadas de percepção associadas a diferentes canais sensoriais e que processam esta informação para a fornecer à camada de decisão; o cérebro artificial como um rede de blocos construtores computacionais, tomando partido da arquitectura do computador digital e a adaptabilidade em termos de complexidade, através da acção de operadores genéticos. Consideramos que todas estas características do gridbrain foram demonstradas através de experimentação.

As nossas abordagens produziram resultados experimentais promissores. Em todos os cenários experimentais, a plataforma evolutiva produziu gridbrains capazes de tirar partido da arquitectura alpha/beta, configurando uma camada de percepção constituida por uma ou mais grelhas alpha, que fornece informação à grelha de decisão beta. No cenário *poison*, foram evoluidos gridbrains com grelhas alpha que utilizam componentes agregadores para transformam informação sensorial de dimensão variável numa representação unidimensional, adequada à formação de decisões. No cenário *targets* foram evoluídos com sucesso gridbrains com duas grelhas alpha para dois canais sensoriais distintos: visão e audição.

Verificou-se a capacidade da plataforma evolucionária de combinar um conjunto heterogéneo de blocos construtores computacionais para definir comportamentos adaptados ao ambiente. Nos diversos cenários experimentais, combinações diversas detes blocos computacionais foram usadas nos gridbrains gerados. Nos cenários *synch* e *targets*, foram gerados gridbrains que usam componentes de relógio para temporizar e sincronizar acções com outros agentes. No cenário *poison* foram gerados gridbrains que tomam partido de componentes de memória para formar decisões na ausência de estímulos sensoriais. De forma geral, a plataforma demonstrou capacidade de adaptação aos desafios dos diversos ambientes, sem necessidade de ajustes específicos. O algoritmo SEEA revelou-se eficiente

na criação de processos evolucionários embebidos numa simulação multi-agente em tempo real.

Uma consequência interessante do nosso trabalho foi a formulação de uma nova abordagem ao problema do controlo de *bloat*. Grande parte da investigação relativa a este problema tem vindo a ser realizada no contexto da programação genética baseada em árvores [Luke and Panait, 2006; Silva, 2008]. Em contraste com o que se verifica nesta última forma de programação genética, o operador de recombinação revelou-se como parte importante da nossa abordagem ao controlo de *bloat*. Através dele, o processo evolucionário tem a capacidade de realizar uma seleção estocástica das melhores características dos diversos gridbrains. Atribuímos esta diferença em relação à programação genética baseada em árvores à diferente representação genética e operadores que utilizamos. Enquanto que na programação genética baseada em árvores, a pesquisa de novas soluções é principalmente baseada na recombinação de sub-árvores, no gridbrain a pesquisa é baseada principalmente nas mutações ao nível das ligações. Através do mecanismo de marcadores de ligação, definimos a unidade de recombinação como sendo o grupo de ligações. Grupos de funcionalidade equivalente são identificados, pelos que a recombinação seleciona uma versão da funcionalidade de apenas um dos progenitores.

Foi tomado um primeiro passo no sentido de definir simulações de vida artificial baseadas na nossa plataforma. Uma experiência simples de co-evolução de espécies produziu resultados interessantes. Fomos, por exemplo, capazes de observar fenómenos de especialização em espécies sem a existência de separações geográficas no ambiente. Espécies com definições iniciais idênticas divergiram para nichos separados de especialização, apenas por efeito da dinâmica do sistema. Verificou-se que este cenário gera histórias evolucionárias variadas e imprevisíveis. A co-evolução na nossa plataforma mostrou a potencialidade de gerar comportamentos diversos e surpreendentes.

O trabalho apresentado tem aplicação possível em diversas áreas, como a simulação de sistemas biológicos e sociais, a robótica e a criação de agentes inteligentes para ambientes virtuais.

Uma tendência actual na biologia e ciências sociais é o uso de simulações como ferramentas de investigação. Esta tendência esta relacionada com o novo paradigma in-

terdisciplinar das ciências da complexidade. A nossa plataforma é aplicável a este tipo investigação, uma vez que permite a definição de cenários de simulação onde o impacto de diversos parâmetros pode ser testado. A facilidade de análise por humanos dos gridbrains gerados ajuda à interpretação dos resultados produzidos pelas simulações.

A robótica evolucionária é uma área de investigação onde controladores de robots são gerados através de processos evolucionários. O gridbrain, devido às suas características previamente descritas, define um novo modelo para controladores de agentes autónomos com possível aplicação à robótica. A ferramente LabLOVE fornece uma plataforma adequada à definição de simulações físicas realistas, onde a evolução de sistemas de robots pode ser testada. Como discutido, a nossa plataforma evolutiva tem a capacidade de gerar comportamentos de grupo, que podem ser aplicados à evolução de equipas de robots com comportamentos cooperativos.

Os ambiente virtuais estão-se a tornar cada vez mais populares, quer na forma de jogos de computador, quer na forma de realidades simuladas, como é o caso do *Second Life*. Estes sistemas estão a tornar-se cada vez mais ricos em termos de interacções entre agentes controlados por humanos e agentes autónomos dotados de inteligência artificial. A plataforma que propomos pode ser usada na criação de agentes autónomos para este tipo de ambientes.

Chapter 1

Introduction

In this chapter we present the motivations behind our work and state the goals that we aimed at. We detail our main contributions and describe the structure for the rest of the thesis.

1.1 Motivation

The human brain can be viewed as an information processing system, far more powerful than any contemporary computer. It vastly exceeds any human created device in both complexity and capabilities. In fact, the field of artificial intelligence [Russel and Norvig, 2002; Costa and Simões, 2008] strives to replicate the several aspects of human intelligence in digital computers, so far with limited success.

Traditional approaches to the creation of artificial intelligence follow classic scientific and engineering methodologies. The several aspects of intelligence are analyzed and isolated. The general problem of developing artificial intelligence is divided into smaller problems. Algorithms are devised to attempt to replicated several aspects of intelligence: decision-making, finding solutions to a problem, formulating plans of action, extracting patterns from large data sets, learning and even having emotions and being creative. Some of these aspects have proven to be easier to replicate than others. From this effort resulted a large variety of algorithms and techniques, some of which found application in conventional software engineering or culminated in landmark technological achievements.

Examples of successes include Internet search engines, industrial automation and robotics, data mining and artificial intelligence in video game agents.

Engineering methodologies are normally of a *divide and conquer* nature, splitting the problem into more manageable sub-problems. This leads to *top down* approaches, resulting in hierarchical systems where modules implement contained functionality and are distributed across layers of abstraction. Modules in higher-level abstraction layers make use of modules in lower-level layers. These methodologies have proven successful in varied problem domains, from aeronautics to software engineering. As a field of computer science with a strong relation to software engineering, it is natural that much artificial intelligence work follows these methodologies.

It is uncertain why the development of artificial intelligence comparable to human intelligence is proving to be so hard to achieve. Several possibilities may be advanced:

- The problem is inherently hard;
- The problem is ill defined, as the definition of intelligence is controversial and an open problem in itself;
- Theoretical limitations exist, like Gödel's incompleteness theorems [Gödel, 1931];
- The von Neumann architecture [von Neumann, 1945] has fundamental limitations that prevent human-like intelligence, as proposed by mathematical physicist Roger Penrose in his book "The Emperor's New Mind" [Penrose, 1991];
- Computational power is still insufficient;
- Intelligence should not be viewed in isolation, it exists in an environment where intelligent and non-intelligent entities interact and results from the evolution of biological systems.

It is likely that the answer is a combination of some of all of these possibilities, or others that are not listed. However, this work will focus on the exploration of the last possibility presented.

Human thinking is not a compartmented process. Scientific, artistic, economic and social trends influence each other. *Top down* approaches and *divide and conquer*, for

example, are strongly related to the industrial era. During the industrial revolution, the drive to optimize industrial processes and achieve mass production led to the ideas of division of labor and, at a greater extreme, *scientific management* or *Taylorism* [Taylor, 1911]. These concepts were a good match to the classical physics model of the Universe as a gigantic mechanical system, governed by linear laws and based on three continuous and fixed dimensions of space and one of time. This type of thinking naturally leads to a model of the human brains as an information processing device that controls the several components of the body, and that is itself an hierarchy of modules that implement the various aspects of intelligence.

During the 20^{th} century, scientific paradigm shifts, technological innovations and social changes initiated what many authors consider a transition into a new era. Examples of this are Alvin Toffler's idea of the *third wave* [Toffler, 1980] and Daniel Bell's *post-industrial* society [Bell, 1974]. The establishment and growth of the Internet accelerated this process. Aspects that characterize this new era include a great improvement in communication across all levels of society and a decrease in compartmentalization. A new scientific field that relates to this new way of thinking aims at the study of complexity.

Complex systems may be defined as networks of components that exhibit properties that result from the decentralized interaction of these components, whereas these properties are not apparent on the components themselves. The connections and interactions between the components may be of a varied nature. Complex systems are thus very diverse, including for example ant colonies, the earth ecosystem, cities, economic markets, the Internet and the human brain. The study of complex systems is an interdisciplinary endeavor that attempts to abstract generic properties from the multitude of practical cases.

Looking at reality through the lenses of complex systems we naturally tend to distance ourselves from reductionist thinking and adopt a more holistic view of systems. Instead of looking for ways to compartmentalize phenomena and study them in isolation we look for layers of systems that emerge from other layers of systems. Scientific disciplines can be viewed as the study of specific layers. Chemical processes result from physical laws. Cells emerge from the interactions of chemical processes. Very large amounts of cells

interact to form tissues, organs and then sophisticated biological entities like the *homo sapiens sapiens*. From the interactions of humans social processes arise, leading to the appearance of larger structures like cities. Cities themselves behave, in many ways, like very large organisms. The processes that take place in these several layers of abstraction are of very different natures and occur at different time scales. They do however influence each other and cannot be studied in separation. Examples of how this way of thinking is influencing science are the modern field of systems biology or the use of decentralized computer simulations to study sociological processes.

The field of artificial intelligence holds the capabilities of the human brain as a standard against which many of its efforts are to be compared. The famous Turing test [Turing, 1950], for example, presents a challenge where an artificial intelligence system has to be able to hold a conversation with a human being through a text terminal, in such a way that the human being cannot decide if he is interacting with another human being or a machine. Scientific models of the human brain are thus of great importance to AI. Classical reductionist approaches attempt to study the brain as an isolated device, reverse-engineer its functionalities and further divide it in hierarchies of modules that can themselves be studied in isolation. This kind of approach is not without merit, but we believe it is also of great value to study the human brain from a complex systems perspective. Not as an isolated device but as part of as much larger network of systems, developing and evolving along time. This leads to an approach to AI where we do not attempt to directly create artificial brains but, instead, establish a system where brains arise and evolve. It is clear that such system must possess the same properties that allow for the evolution of brains to occur in nature.

A current direction in AI research is bio-inspiration [Floreano and Mattiussi, 2008]. It becomes important to understand how sophisticated systems like the brain come to existence in the natural world. Although the origin of life itself is still an open problem, the progressive evolution of very simple, initial life forms into much more sophisticated and complex organisms is explained by *modern evolutionary synthesis* [Huxley, 1942]. *Modern evolutionary synthesis* is a combination of Darwin's theory of the evolution of the species [Darwin, 1859] with Mendel's genetics [Mendel, 1865]. It states that selective

pressure causes organisms to evolve. Organisms that possess traits that benefit their chances of survival and reproduction will be more likely to create copies of themselves. Random genetic mutations will some times cause changes in organisms. Most of these changes will be counter-adaptive but a few will be beneficial. The latter will tend to propagate to future generations because of selective pressure. The accumulation of these changes over large time scales will produce increasingly more complex organism, with more sophisticated systems to increase survival chances and reproductive success.

The appearance of species with reproductive isolation causes the specialization and adaptation of organisms to niches. Competition for resources between species causes arms race scenarios which increase the pressure for improvements, in the need to outperform opponents.

In his seminal work *Hidden Order* [Holland, 1995], John Holland proposes that Darwinian evolution is an instance of a broader phenomenon called adaptation. Other instances of this phenomena include animal's central nervous systems learning behaviors, immune systems learning to distinguish self from non-self or economical markets changing over time. Others suggest that evolutionary processes take place even outside biological evolution. An example of this is Richard Dawkins' idea of the evolution of human culture through the transmission of units of cultural information called *memes* [Dawkins, 1976]. Adaptation and evolution appear to be key elements in the generation of Nature's complexity.

One property present in most complex systems is nonlinearity. A function is said to be linear if has the properties of *additivity* and *homogeneity*. A function f has the property of *additivity* if $f(x+y) = f(x) + f(y)$ and the property of *homogeneity* if $f(k.x) = k.f(x)$. Most mathematical tools that can be used to analyze systems assume linearity. This is the case, for example, of differential calculus which is broadly used in classical physics. In fact, nonlinear systems are very hard to model mathematically. It has been found more effective to study many of these systems through computer simulations. In these simulations, local interactions are modeled and then the global behavior is observed. It is likely not a coincidence that the development of chaos theory coincided with the availability of digital computer in research labs. One example of this is the accidental discovery of the Lorenz

attractors by Edward Lorenz, who was working with a computer simulation to model weather systems [Lorenz, 1963]. Another example is the visualization and exploration of fractals, like the Mandelbrot set, made possible by the digital computer.

The ability to simulate large stochastic processes allows for the use of evolutionary principles in search and optimization. Evolutionary computation is nowadays an important branch of computer science. Multi-agent simulations allow for the study and observation of nonlinear behaviors arising from the local interactions of agents. This type of simulation is used in the field of Artificial Life to study biological processes and in the broader field of Complex Systems to study all kinds of systems, e.g., financial markets and social structures. The scientific paradigm shift of complexity combined with the increasing availability of computational power establish simulation as an important scientific tool in the 21^{st} century.

With the spread of the personal computer and Internet access, simulations are increasingly becoming more popular as a means for human being to interact in shared virtual realities [Smart *et al.*, 2007]. This kind of environment is being used for entertainment, collaboration, socialization and education. Notable examples include the *Second Life* [SecondLife, 2008] virtual world and *Massive Multi-player Online Role Playing Games* (MMORPGs) like *World of Warcraft* [WorldOfWarcraft, 2008]. Virtual reality worlds can be privileged environments for human and computational intelligences to interact.

1.2 Goals

The main goal of this thesis is to study mechanisms to evolve computational intelligence in multi-agent simulations. More specifically, we focus on real-time, continuous virtual environments with spatial dimensions. We favor bottom-up, decentralized approaches where global behaviors emerge from local interactions.

In the pursuit of this goal, the following challenges are to be addressed:

- Conception of a suitable agent brain model;
- Definition of an agent-environment interface that allows for the establishment of simulation parameters on a local interaction, object-to-object basis;

- Creation of effective evolutionary algorithms that do not impose unnecessary restrictions on the simulation and are able to maintain multi-species environments.

The agent brain model must have a genetic representation and a set of genetic operators that allow the evolutionary process to effectively search for higher-quality individuals. This representation must be capable of expressing non-trivial computational constructs, including decision-making, generalization, memory and synchronization. The brain must be able to operate under the hard computational constraints of a real-time simulation. It must be capable of processing sensory information from different perceptual sources and relative to a variable number of external entities. The internal size and complexity of the brain must not be fixed and pre-determined, but adaptable to the demands of the environment.

In the spirit of the decentralized, bottom-up approach, it should be possible to define interfaces between the agent and the environment, including other agents and simulation objects, in a way that imposes the least possible amount of pre-conceptions on the process. It is thus important to limit these interfaces to one-to-one relationships between entities in the simulation.

The evolutionary algorithm should be able to adapt to the requirements of the simulation and not the other way around. From the observation of biological systems, it is found that species diversity plays an important role in the development of more and more complex organisms, behaviors and brains. An heterogeneous environment of cooperation and competition between species appears to be an important aspect of evolutionary systems capable of producing agents with intelligent behavior. Our evolutionary algorithms must thus support multi-species simulations.

Secondary goals of our work include the development of a software simulation tool that implements the algorithms and methods that we will propose. We wish to make this tool available to the community so that our work may be validated and expanded upon. It is also our goal to study practical application of our work, both of a scientific and engineering nature. We believe the simulation framework we will propose can be applied to scientific simulations, namely in the fields of complex sciences and artificial life. We also believe that our work can serve as a basis for the development of intelligent agents for video games

and virtual worlds.

1.3 Contributions

A new algorithmic framework for the evolution of computational intelligence in multi-agent simulations is presented in this thesis. The main contributions consist of the parts of this framework:

- An evolvable brain model for autonomous agents, called the *gridbrain*. The gridbrain is a multi-layered network of computational building blocks capable of forming decisions based on variable-sized information from multiple sensory channels;
- A *steady state* evolutionary algorithm for the non-intrusive embedding of evolutionary processes in real-time, continuous multi-agent simulations, called *Simulation Embedded Genetic Algorithm (SEEA)*;
- A simulation tool called *LabLOVE (Laboratory of Life On a Virtual Environment)*. LabLOVE defines an agent model that relies on decentralized, local interactions. It also includes a visualization system as well as data gathering modules. One particular environment was developed for purposes of experimentation for this thesis, but due to its object-oriented and modular design, LabLOVE is extensible and other environments can be created.

The work described in this thesis was partly published in articles in peer-reviewed international conferences and journal.

The concepts underlaying our work were first published in the proceedings of the *IEEE Symposium on Computational Intelligence and Games (CIG'06)* [Menezes *et al.*, 2006]. An agent brain model that led to the gridbrain was published in the proceeding of the *2006 European Conference on Complex Systems* [Menezes and Costa, 2006]. The gridbrain model was first published in the proceedings of the *First IEEE Symposium on Artificial Life* [Menezes and Costa, 2007b] and then on the *International Journal of Information Technology and Inteligent Computing* [Menezes and Costa, 2008b]. The concepts underlaying the agent model described in chapter 6 were published in the *proceedings of the*

9th European Conference on Artificial Life [Menezes and Costa, 2007a]. The final gridbrain model and part of the experimental results presented in chapter 8 were published in the *proceedings of the 5th European Conference on Complex Systems* [Menezes and Costa, 2008a].

The LabLOVE simulation tool, including the gridbrain library, an implementation of the SEEA algorithm and the experiment definitions used in this thesis, is available to the scientific community under the open source GPL license version 2[1] [Menezes, 2008b].

1.4 Organization

In this chapter we described the motivations, goals and contributions of this thesis. In the following two chapters we present important background information. In chapter 2 we will discuss fundamental concepts that underly this work. In chapter 3 we address the state of the art in evolutionary multi-agent simulations. In the following chapters we describe the models we propose. In chapter 4 we present the gridbrain model, in chapter 5 the SEEA algorithm and in chapter 6 the LabLOVE agent model and simulation tool. We then move to the presentation and discussion of experimental results, in chapters 7, 8 and 9. We end with conclusions and the discussion of future work.

A companion CD is included with this thesis. The CD contains the LabLOVE source code and instruction on how to use the tool, published papers related to this work and logs of experimental results.

[1] The text of this license can be found at http://www.gnu.org/licenses/old-licenses/gpl-2.0.html

Chapter 2

Background

In this chapter we present theoretical concepts, techniques and algorithms that are relevant to the ideas we will present in the following chapters. We will progress from generic concepts to more specific ones, as they pertain to the models to be proposed in following chapters.

2.1 Complexity, Emergence and Self-Organization

Complexity, emergence and self-organization are properties that systems can have, and are highly related [Axelrod and Cohen, 1999; Holland, 1995; Camazine *et al.*, 2001]. The definition of any of these properties has proven to be difficult, and is under heavy debate on scientific communities.

A complex system can be defined as a system formed by individual components and a network of interactions between them. A complex system exhibits properties that are not apparent from the individual components that they are made of. This type of structure may be hierarchical, with complex systems serving themselves as components for higher-level complex system. Typically, the behavior of a complex system is non-linear and is difficult to model using conventional mathematical tools, like differential equations.

This hierarchical organization is tied to the concept of emergence. In fact, it is common to say in the language of complexity science, that one layer of the hierarchy emerges from another. Emergence can be seen as the appearance of novel properties in a complex system,

where these properties are not present on the individual components of the system. The interactions form a higher-level system, which is *different* from its components.

Self-organization is any process by which a system, without control from the outside, increases its internal complexity. Many types of self-organizing mechanisms can be found at all levels of abstraction in nature. Examples are phase transitions in physics, molecular self-assembly in chemistry, Darwinian evolution and animal swarms in biology, brain learning in neuroscience, trophic networks in ecology, organization of human societies in sociology and stock markets in economy.

2.2 Multi-agent Systems

A *Multi-Agent System (MAS)* can generically be defined as a set of agents interacting in an environment. This type of system allows for the modelling of decentralized phenomena, and thus has a wide range of application in areas where this kind of phenomena is relevant, namely biology [Adamatzsky and Komosinski, 2005], social sciences [Schelling, 1971; Deffuant, 2006], economy [Tesfatsion, 2002; Izquierdo and L.R., 2006; Borrelli *et al.*, 2005; Caillou and Sebag, 2008] and artificial intelligence [Steels, 2004; Schut, 2007]. Many of these systems are computer simulations, but other mediums may be used to implement a MAS, for example, robotics.

There is great variety of MAS, both in the way the environment and the agents are modelled. This variety is explained by the heterogeny of the fields of application, but also by the wide range of modelling approaches that have been devised. Environmental models may be purely conceptual [Palmer *et al.*, 1994] or have spacial dimensionality. In the latter case many possibilities exist, from discrete, grid-based, two-dimensional to physically simulated three-dimensional environments. Purely conceptual environments have been used, for example, to simulate financial markets. The discrete, cellular automata like approach is common in social simulations [Epstein and Axtell, 1996], while three-dimensional MAS environments have been successfully applied to video games [Electronic Arts, 2008].

Another source of variety is the internal mechanism and capabilities of the agents,

which are potentially as diverse as the multitude of systems studied in the field of artificial intelligence.

A type of MAS of special interest to the field of artificial intelligence is that which is made of intelligent agents. In this context, an *intelligent agent* is typically considered to have the following properties: *autonomy*, *social ability*, *reactivity* and *proactivity* [Wooldridge and Jennings, 1995]. Autonomy means that there is no global controller determining the agents' actions. Social ability is the ability to interact with other agents in some way. Reactivity is the ability to react to stimulus from the environment and proactivity is the ability to form decisions in accordance to individual goals.

Another subset of MAS that is pertinent to this work is that of evolutionary multi-agent systems. In this type of MAS, agents undergo some form of stochastic adaptive process inspired by Darwinian evolution. The work we present focuses on MAS that are both evolutionary and based on intelligent agents.

Multi-agent systems are strongly related to complexity sciences, and are increasingly being used as a scientific tool to study and develop systems that contain certain properties that are difficult to analyse using a more classical mathematical modelling. This idea will be expanded upon on the following section.

2.3 Complex Adaptive Systems

The field of Complex Adaptive Systems or CAS was born in the Santa Fe Institute and resulted from the interdisciplinary nature of the research performed there. CAS is an effort to abstract the fundamental processes in diverse complex systems that present one common property: coherence under change. CAS are a subgroup of complex systems. As any complex system, they consist of a network of interacting components. They are called *adaptive* because they have the ability to change in response to changes in the environment, in a way that preserves some type of coherence. One example of this is the immune system. The fundamental coherence of the immune system is the ability to distinguish self from non-self. Since it is impossible for the immune system to know in advance all the situations that it may encounter, it must be able to adapt to new cases.

When exposed to a never before encountered pathogen, the immune system is capable of learning a new distinction. It changes itself in order to preserve its fundamental coherence.

The components of a CAS are abstracted as agents. Agents in CAS are as diverse as the systems under study. They can be the neurons in the central nervous system, the antibodies in the immune system, the investors in a financial market, the citizens in a country or the insects in a swarm. Since a CAS is a network of interactions, it is important to identify the stimulus and responses of the component agents. In neurons these are electrical impulses, while in humans these can be sensory perceptions and muscular activity.

In an effort towards theory, John Holland proposes seven basic characteristics that define CAS [Holland, 1995]. This seven characteristics consists of four properties and three mechanisms. The properties are: aggregation, nonlinearity, flows and diversity. The mechanisms are: tagging, internal models and building blocks.

Aggregation can be observed in two ways. Firstly, it is typical of the viewpoint of CAS. Systems have aspects that are relevant to a CAS model and others that are not. By concentrating only on the relevant aspects, phenomena and entities that operate in different environments and time scales may reveal themselves to be of a similar nature. The second way in which we observe aggregation in CAS is in the emergence of *meta-agents* from agent structures. A network of agents operating at a certain level may cause the emergence of *meta-agents* that operate on an higher level. An example of this is the emergence of the human mind from the interactions of neurons. At the social level, humans may be seen as individual agents, but the behaviors of these agents emerge from the interactions of lower level agents - the neurons. This process may continue *ad infinitum*, with *meta-meta-agents* emerging from *meta-agents*.

Nonlinearity is a common property of complex systems in general, as discussed in section 1.1. One example of nonlinearity in a CAS found in nature is the variation of the populations of two species in a predator-prey scenario. The Lotka-Volterra model [Lotka, 1956] allows us to predict these population using the following equations:

$$U(t+1) = U(t) - dU(t) + bU(t) + r.c.U(t).V(t)$$

$$V(t+1) = V(t) - d'V(t) + b'V(t) - r'.c.U(t).V(t)$$

$U(t)$ is the number of predators per area unit at time t, and likewise, $V(t)$ is the number of preys. b and b' are birth rates for predators and preys, d and d' are the death rates. c is the efficiency of the predator in searching for preys. r is the efficiency of the predator in converting food into offspring. r' is the rate of predator-prey interactions that result in the prey being captured and killed. This system of equations defines a nonlinear system. Computer simulations show that in many cases, both populations oscillate, going through cycles of abundance and famine.

Flows is a property strongly related to the network of interactions that constitutes the CAS. It refers to sequences of transferences that take place between agents. These transferences are typically of resources or information. In the central nervous system, electrical impulses activate neurons that generate other electrical impulses that activate other neurons and so on, generating a flow of information. Another example of flow of information is the spreading of a rumor through cell phones in human societies. In an economy, companies subcontract other companies to perform a task, creating a flow of money. Industries create flows of goods by buying, transforming and selling to other economic agents. There are two effects usually observed in flows. These are the *multiplier* and the *recycling* effects. The multiplier effect occurs when the injection of resources in an agent results in a cascading of this resource to other agents through the network. Supposing a simple case where an agent receiving an amount of resource a passes a fraction $r.a$ of this resource to another agent, the total effect to the system is given by:

$$a + a.r + a.r^2 + a.r^3 + ... = a.(1 + r + r^2 + r^3 + ...) = a.(\frac{1}{1-r})$$

The effect of a is thus multiplied by $\frac{1}{1-r}$ which is greater than 0 because $0 \leq r < 1$. The recycle effect occurs when there are cycles of resources in the network. Let us suppose a network where resource A is transformed into resource B with an a ratio and resource B is transformed into resource C with a b ratio. Under these conditions, each unit of A will result in the production of $a.b$ units of C. However, if some ratio of C can be reconverted

into the resource B, an increase in production of C is obtained without an increase in the consumption of the base resource A. Recycling effects are commonly observed in economies and in nature, for example in trophic networks.

Diversity is the property that CAS have of generating a variety of agent types. In a CAS, a significant part of the environment and context of an agent is provided by other agents. The interactions of agents create *niches* that other types of agents may explore. The adaptation of new agents to their *niches* create new types of interactions that create new *niches* and so on. Diversity in CAS is thus dynamic. The removal of a type of agent tends to result in the appearance of a new type of agent that explores the *niche* that was left free. Again, diversity can be observed in nature (species), in economy (business models), in brains (neural network topologies) and many other systems.

Tagging is the mechanism that allows agents to establish distinctions in other agents and objects in the environment. Tagging is the fundamental process in the formation of aggregates, because without the ability to make distinctions, there would be no basis for the formation of preferential interactions. Without tagging, the system would be completely symmetrical, and thus, regular. The tag is an abstract concept that can take many forms. It may be a visual characteristic like a shape or a color that allows elements of a species to identify potential mates or preys. It may be a device that defines a type of agent in human societies, for example the use of suits to convey seriousness and professionalism, or the use of body piercings to convey non-conformity. It may be a molecular configuration that allows chemical binding with specific molecules.

The internal model is a mechanism that is highly related to the goal of computational intelligence. Internal models allow agents to anticipate some aspect of the environment. Internal models exist in a great range of sophistication, from simple reactive behaviors like that of a thermostat to the cognitive capabilities of the human brain. The thermostat for a heating system switches to an *on* state in an implicit anticipation that this action will increase the temperature in the environment. This simple class of internal model is called *tacit* and relates to reactive behaviour. Many instances of it can be found in nature, for example in bacteria moving up a nutrient gradient or ants following pheromone trails. In this case, the agent does not possess an explicit internal representation of the environment,

but the anticipation ability is implicitly contained in some reactive mechanism. Certain artificial intelligence algorithms are capable of maintaining internal representation of the environment and explore a tree of possible future states. An example of this is the *minimax* algorithm [Diamand and Diamand, 1996; Samuel, 1959], normally used in strategic board games like checkers or chess. This type of internal model is called *overt*. The brains of advanced animal species maintain *overt* internal models and use them to formulate plans of action, for example navigating a certain known territory.

The final mechanism is that of building blocks. The concept of this mechanism is that everything in a CAS is made of a combination of some fundamental components or parts, the building blocks. This allows for both reusability and novelty. Building blocks depend of both the type of system and the level of abstraction being considered. In fact, this mechanism is related to the property of aggregation, in that building blocks at an higher level of abstraction are made of building blocks from a lower level. Quantum particles are the building blocks of low level physical worlds, but organize into higher level structures like atoms and then molecules, which are the building blocks of chemical processes and so on. We can find building blocks in other domains, for example letters as the building blocks of words and words as the building blocks of messages, or machine code instructions as the building blocks of computer programs. There is also a strong relation with the mechanisms of tagging and internal models. Tagging involves an abstraction of some aspect of an entity. This is possible because of the repetition provided by building blocks. This in turn is important for the internal model to be able to create useful generalizations about the world. Also, the internal model is itself made of building blocks taking advantage from reusability. This can be observed in the way that an artificial intelligence system is made of computer instructions or a brain is made of interconnected neurons.

2.4 Artificial Life

Artificial life (or alife) is an umbrella field for a variety of approaches that attempt to study life by artificially replicating biological phenomena. These approaches result mainly from a cross-fertilization of concepts from biology and computer science. The name of the

field was proposed by Christopher Langton, one of its fathers. He envisioned artificial life as the study of *life as it could be* [Langton, 1989].

There are three types of medium where artificial life experiments are performed: software, hardware and biochemistry. These medium correspond respectively to *soft*, *hard* and *wet* alife [Bedau, 2003].

Artificial life shares common ground with complex adaptive systems and multi-agent systems. In fact, some work can be considered to belong to the three fields, as is the case of Holland's echo simulation [Holland, 1995], which we will addressed in the next chapter.

A base concept in alife is the construction of systems from the *bottom-up*. A source of diversity in alife approaches is the abstraction level of the base components of the system. In previous sections we discussed classes of systems based on the agent concept. In alife, much work exists with lower level abstractions, notably using *cellular automata*. A cellular automaton consists of a grid of cells with a finite number of dimensions, with each cell being in one of a finite number of states [Schiff, 2008]. There is an universal rule for updating the state of each cell, based on the state of its neighbours. Time advances in discrete steps, with the state of every cell being updated in each step. The *cellular automata* abstraction allows for the emergence of complex phenomena from very simple settings. In fact, this model was first created in the context of the first recognized effort related to alife, von Neumann's work in creating a self-replicating machine [von Neumann, 1966]. They where later used by Langton in another classical alife system, *Langton's loops*, and its variations [Sayama, 1998].

Another low-level approach to alife is that of *artificial chemistries* [Dittrich *et al.*, 2001], where processes similar to chemical reactions are simulated by creating interaction rules between molecules that attempt to replicate or abstract real chemistry. This approach attempts to emerge biological phenomena from chemical building blocks, in a similar fashion to what can be observed in nature.

Higher-level approaches to alife use the concept of agent, while still taking advantage of abstracted nature-inspired processes like Darwinian evolution. In chapter 3 we will provide several examples of such systems. The work presented in this document also fall into this category.

One of the unsolved goals of alife is to attain *open-ended* evolution. Open-ended evolution may be defined as the ability of an evolutionary system to perpetually produce novel forms [Standish, 2003]. No artificial life system that we know of, has to this date shown the capacity for perpetual innovation. It is common to associate open-ended evolution with the absence of explicitly defined goals in the evolutionary process, however this connection is not proven. In fact, we can argue that the co-evolution of multiple species with explicit goals may have the potential for open-evolution. If the goals result in situations of cooperation and competition between the species, this can lead to a dynamic environment where the best strategies to achieve the goals change constantly. It is also important to notice that it is very hard, if not impossible, to determine if a human designed system is *goal free*, because goals can be indirectly defined and smuggled into the foundational rules of the simulation.

2.5 Evolutionary Algorithms

Evolutionary algorithms (EA) [Eiben and Smith, 2008] are a subset of Evolutionary Computation. EAs draw inspiration from biological evolutionary processes and apply the underlying abstract concepts of Darwinism and genetics to perform a stochastic search for solutions to a problem. The most common use of evolutionary algorithms is in search and optimization problems, but application exist in many areas, including the generation of computer programs, systems engineering and artificial creativity. The field of evolutionary computation is a breeding ground for a wide range of approaches and techniques. However, a general, high-level algorithm that encompasses the great majority EA approaches may be described. In listing 2.1 we present a possible description of this generic algorithm in pseudo-code.

The evolutionary algorithm creates successive generations of populations of candidate solutions to the problem being addressed. The quality of each individual is quantified by a fitness function which is specific to the problem domain. Individuals with higher fitness values have higher chances of propagating their genetic information because they are more likely to be selected as parents and surviving to the next generation. Reproduction

```
t = 0
genotypes[t] = generateRandom()
phenotypes[t] = map(genotypes[t])
fitnesses[t] = eval(phenotypes[t])

while !stop_criterion:
    offspring[t] = selectParents(genotypes[t], fitnesses[t])
    offspring[t] = applyGeneticOperators(offspring[t])
    genotypes[t + 1] = selectSurvivors(genotypes[t], offspring[t])
    phenotypes[t + 1] = map(genotypes[t + 1])
    fitnesses[t + 1] = eval(phenotypes[t + 1])
```

Listing 2.1: Generic evolutionary algorithm.

with variation, provided by the genetic operators, creates diversity in the population. This combination of a selective bias towards higher quality with diversity may allow the algorithm to efficiently explore the solution space. Individuals have a representation, or genotype, that encodes a candidate solution or phenotype. Genetic operators work in genotype space while the fitness function evaluates phenotypes. If we consider a way to organize genotypes in an n-dimensional space, the addition of a dimension for the fitness value associated with each genotype defines a *fitness landscape* in $n+1$ dimensions. Broadly speaking, the selective bias towards higher quality solutions will cause the population to climb fitness peaks, while diversity will prevent the population from being stuck in sub-optimal peaks, or *local maxima*.

The wide diversity in evolutionary computation systems stems from the many possible approaches to the various aspects of the generic evolutionary algorithm: genetic representation, genetic operators, parent selection strategy and survivor selection strategy.

Representations may be direct or indirect [Tavares, 2007]. In direct representations, there is no difference between the genotype and the phenotype. One very simple example of this would be the maximization of a function $f(x)$, where both the genotype and the phenotype of individuals are possible values of x. Indirect representations may use a variety of encodings. Possible encodings and best approaches are problem specific. One advantage of indirect representations can be an increase in redundancy, with multiple genotypes corresponding to the same phenotype, allowing for neutral mutations. Neutral mutations change the genotype but not the phenotype. In biology, it has been proposed that neutral

mutations are an important part of the evolutionary process [Kimura, 1983]. Likewise, neutral mutations have been shown to have a positive influence in the performance of some evolutionary algorithms [Galván-López and Rodríguez-Vázquez, 2006; Vassilev and Miller, 2000; Yu and Miller, 2001; Miller and Smith, 2006; Yu and Miller, 2006].

The most usual classes of genetic operators used are recombination and mutation. Recombination operators are related to sexual reproduction in biological organisms. This class of operators combines the genetic material of two or more parents to generate offspring. Mutation operator perform random changes in the genome. The specific mechanics of these operators are highly dependent of the representation and the approach used.

Common selection strategies are *roulette wheel* and *tournament*. In *roulette wheel*, the probability of an individual being selected is given by the relative proportion of its fitness value to the summation of all fitness values in the population. If f_i is the fitness of the individual under consideration and N the size of the population, the probability of selection p_i is given by:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

In *tournament selection*, n individuals are chosen at random from the population, the one with the highest fitness being selected. The purpose of these strategies is to maintain the bias towards higher quality without compromising diversity. More *naive* approaches like just selecting the individuals with higher fitness would cause most algorithms to get stuck early in local maxima.

EAs are classically divided into four main branches: Genetic Algorithms [Holland, 1975], Evolution Strategies [Beyer and Schwefel, 2002], Evolutionary Programming [Fogel *et al.*, 1966] and Genetic Programming [Koza, 1992; Poli *et al.*, 2008]. The former two branches deal with solving optimization and search problems. The latter two, especially Genetic Programming, are more relevant to the work presented in this document because they deal with the evolution of computer programs.

2.6 Genetic Programming

In genetic programming (GP), populations of computer programs are evolved. The fitness of an individual is determined by executing the computer program it represents and comparing the output of this program with a desired optimal output. One important challenge in evolving computer programs is how to represent them. The same problem exists in facilitating the work of a human programmer. Programming languages exist mainly to abstract machine code into a representation that is more suitable to the way humans think. In the same fashion, representations are devised to facilitate the efficient operation of genetic operators. Various representations for programs have been studied in the field of GP. We will describe the tree-based representation, which is the most popular one, and then discuss other representations that relate to the ideas presented in this thesis.

2.6.1 Tree-based Genetic Programming

Tree-based GPs represent programs as *syntax trees*. This representation has a strong relationship with functional programming. Each node in the tree is either a function or a terminal. Functions nodes branch to deeper nodes. The nodes a function connects to in the next level are its parameters. They can be terminals or other functions. Terminals are usually constant values and input variables. A tree-based GP must define a terminal set and a function set. These constitute the building blocks with which programs are constructed. As an example, we could have the terminal set $T = \{x, y, \mathfrak{R}\}$, where R is an ephemeral constant that can take any value in $\mathbb{R}$, and the function set $F = \{+, -, *, \%\}$. One possible program is: $(3.1 + x) * (x - (y * y))$. It is easy to discern the tree structure that corresponds to this program by writing it in prefix notation: $(* \ (+ \ 3.1 \ x) \ (* \ y \ y))$.

The most common genetic operator used are *subtree crossover* and *subtree mutation*. Subtree crossover is a recombination operator. A crossover node is randomly selected in each parent tree. These nodes are the roots of the selected subtrees. The offspring is generated by creating a copy of the first parent and replacing its selected subtree with a copy of the selected subtree from the second parent. Subtree mutation consists of randomly selecting a node from the tree and then replacing the subtree defined by this node with a new, randomly generated subtree.

One interesting extension to the basic tree-based model is that of *automatically defined functions (ADF)* [Koza, 1994]. A common technique in computer programming is code reuse. In it's most usual form, programmers write functions that implement some functionality that can be used by different parts of the program. Programs can be defined as hierarchies of functionalities, with modules designed to solve sub-problems. This can be achieved in tree-based GP by defining two types of branches originating from the program root: automatically defined functions and result production branches (RPBs). This way, RPBs can call ADFs and ADFs can call each other. The availability of this mechanism does not mean that evolution will make productive use of it. Achieving evolutionary processes where useful reusable modules are generated is a challenge in itself.

2.6.2 Parallel Distributed Genetic Programming

Tree-based GPs use parse trees, a powerful representation that has been successfully applied to diverse problem domains. However, this representation has limitations. One of them is that it is highly sequential. Other representations have been created to make parallel and distributed programming possible, as is the case of *parallel distributed genetic programming (PDGP)* [Poli, 1996, 1999]. PDGP programs are represented as graphs with labeled nodes and oriented links. Graph nodes correspond to functions and terminals, while links determine function arguments. Graph nodes are place in a multi-dimensional grid with a pre-determined size. Output nodes are usually placed in the top of the grid and computation takes places from the bottom to the top. The height of the grid determines the maximum graph depth supported.

Nodes may not be active. Inactive nodes do not contribute to the computation, so although PDGP has fixed genotype sizes, phenotype sizes is variable (but limited to a maximum value determined by the size of the genotype grid). The basic PDGP representation allows only upward connections between nodes in adjacent rows, but extensions are proposed that lift these restrictions.

The basic crossover operator for PDGP is a generalization of subtree crossover for graphs, and it is called *sub-graph active-active node crossover (SAAN)*. It consists of the following steps:

- An active node is selected in each parent. These are the crossover points.
- The sub-graph that contains all the nodes that contribute to the computation of the crossover node in the first parent is determined.
- The sub-graph from the first parent is inserted in the crossover point of the second parent. If the sub-graph does not fit in the second parent's grid, it is wrapped around.

Restrictions have to be put in place so that the insertion of the sub-graph in the second parent does not cause the resulting graph to have a larger depth than what the grid supports. A way to do this is to select the first crossover node at random, but restrict the choice of the second crossover node to the ones that will not cause this limit to be exceeded. The placement of the sub-graph can also cause the resulting graph to exceed the width supported by the grid. In this case the sub-graph is wrapped around. Nodes that horizontally exceed the width of the grid on one side are wrapped around to the other side of the grid.

Several variations of the basic crossover operation are possible. Three of them consist of allowing crossover points to be selected amongst both active and inactive nodes. There are: *sub-graph inactive-active node (SIAN)*, where the first crossover point may be inactive; *sub-graph active-inactive node (SAIN)*, where the second crossover point may be inactive and *sub-graph inactive-inactive node (SIIN)*, where both crossover points may be inactive. Other variations consist of selecting just part of the sub-graph in the first parent, as is the case of *sub-sub-graph active-active node (SSAAN)*. All combinations of sub-sub-graphs with inactive crossover points are also possible. To our knowledge, no conclusive study exists so far on the relative merits of the different types of crossover operators.

The creators of PDGP propose two mutation operators: *global mutation* and *link mutation*. Global mutation is similar to the subtree mutation used in tree-based GP. A random node is selected and a random subtree is generated to be placed in this node. Link mutation operates at connection level. A random connection is selected and its origin randomly changed. Changing only the origin guarantees that the program remains valid, as the correct number of parameters for functions is preserved.

Three extensions are proposed to the base PDGP model that improve its generality. One is to allow connections between non-adjacent nodes, so that any acyclic graph may be represented. Another is to allow backward connections so that cycles can be defined. To implement this extension, the program interpreter must be able to limit the amount of times a cycle is executed, as to prevent infinite loops in program execution. The crossover operator must in this case also perform a vertical wrap-around. The final extension proposed is to have labeled links. Labels may be of different data types. Real number labels can be used as connection weights, so that graphs can represent neural networks. One other possibility is to have labels be symbols of a language, allowing the graph to represent state machines. An new mutation operator is proposed for when labels are used, called *label mutation*. This operator replaces a label from a randomly selected connection with a random element from a pre-defined label set.

PDGP was successfully applied to the following problem domains:

- Evolving an exclusive-or (XOR) function, which is a boolean function that takes two parameters ($f(a, b)$) and outputs 0 if $a = b$, 1 otherwise.

- The *even-3 parity problem*, which consists of evolving a boolean function that takes three parameters and outputs 1 if and even number of parameters is 1, 0 otherwise.

- The *lawnmower problem* [Koza, 1994], where a program is evolved to control the movement of a lawnmower that must cut all the grass in a lawn. In this case, functions with side-effects are used.

- Symbolic regression of the function $f(x) = x^6 - 2x^4 + x^2$ by using 50 data samples produced by this function.

- A *MAX problem* [Gathercole and Ross, 1996]. MAX problems consist of finding a function that outputs the maximum possible value for given functional and terminal sets. The MAX problem for the function set $\{*, +\}$ and the terminal set $\{0.5\}$ was used.

- Encoding and decoding of 4-bit long binary strings.

- Evolving a finite state automata capable of recognizing a language [Brave, 1996]. For this problem, connection labels were used and backward connections allowed. The language used was $L = a^*b^*a^*b^*$, which consists of all sentences formed by 0 or more a symbols, followed by 0 or more b symbols, followed by 0 or more a symbols, followed by 0 or more b symbols.

- The *MONK's problems* [Thrun *et al.*, 1991], which are a collection of binary classification problems commonly used to benchmark learning systems.

The first two problems were used mainly for the study of PDGP behavior under different parametrizations. The lawnmower problem is hard for simple tree-based GP to solve because it benefits greatly from code reuse, and for the same reason better results are achieved using ADFs. For the instances of the problem studied, PDGP is shown to outperform GP with ADFs while creating less complex programs. An interesting consequence of these result is that, for this problem, the function output reuse naturally provided by the PDGP representation performs better than the explicit modularity provided by ADFs. The symbolic regression problem was chosen to further compare ADFs modularity with PDGPs output reuse. Tree-based GP with and without ADFs have similar performances for this problem, although code reuse could be theoretically beneficial. Again, PDGP was shown to outperform both forms of tree-based GP, finding simple solutions that benefit from output reuse.

The MAX problem was chosen because it poses difficulties to depth-limited tree-based GPs. PDGPs have a further restriction of width, so it was found interesting to investigate if it suffered from the same limitation in attacking this problem. It was found that is not the case, and an instance of the MAX problem for which depth limited tree-based GPs fail was easily solved.

The encode-decoder problem was used to test the evolution of programs with non rectangular grids. A sand-glass shaped grid was used, with 9 rows. All rows have a width of 4 except for the middle one, which has a width of 2. Input nodes are only allowed on the bottom 4 rows and output nodes are on the top, as usual. The basic model of PDGP is uses, with only adjacent connections allowed. The program is evolved to output the same

values that are input. The idea is that the program is forced to evolve an encoding in the bottom half that allows information to go through the two-bit wide central row and then decode it in the top half. The experiment was performed for the following set of possible inputs: $\{1000, 0100, 0010, 0001\}$. A program was produced that successfully encoded and decoded the string using an intermediary 2-bit representation.

PDGP was shown to be capable of evolving a finite state automaton capable of recognizing a language, which means, deciding if a certain sentence belongs to this language or not. This experiment allowed the testing of a representation that uses labels. In this cases the label set consisted of the possible input symbols: $\{a, b\}$, and the function set consisted of automaton states: $\{N1, N2, E1, E2\}$. N states are non-stop while E states are terminal, with the number representing the amount of outgoing connections for this state. All states receive a sentence on input and check if its first symbol corresponds to the label of one of their output connections. If it does, the first symbol is removed and the remaining sentence is passed to the target node of the matching connection. If it does not, false is returned. Terminal states return true if they receive an empty sentence. A very compact, 4 node solution was found.

MONK's problems provide an harder challenge than the previous cases. These problems consists of finding rules that correctly classify a data set. The data set has 432 instances. Each instance consists of 6 attributes, which can take integer values. There are three MONK problems, that differ on the underlying classification rule. Classification is binary. For example, the rule for the MONK-1 problem is: $(a_1 = a_2) \vee (a_5 = 1)$, where a_n is attribute n. The MONK-3 problem also introduces noise. Experimentation was performed with the MONK-1, MONK-2 and MONK-3 problems. Completely correct solutions were found for MONK-1 and MONK-3, but no solutions could be found for MONK-2.

2.6.3 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) [Miller, 1999; Miller and Thomson, 2000] has been developed with similar goals to PDGP but uses a different representation. Like PDGP, CGP models programs as directed graphs. While in PDGP there is little distinction between genotype and phenotype, in CGP a more indirect representation is used, allowing

for less sophisticated genetic operators. Like in PDGP, function nodes are placed in a grid, but the genotype consists of two fixed-sized sets of integers.

A set of parameters is used to defined the topology of the grid: $P = \{n_i, n_o, n_n, n_j, n_r, n_c, l\}$. The meaning of these parameters is as follows: n_i is the number of inputs of the program, n_o is the number of outputs, n_n is the number of inputs per function, n_j is the number of functions, n_r is the number of functions per row, n_c is the number of functions per column and l is the maximum allowed column gap in connections. This means that with $l = 1$ connections are only allowed between adjacent columns, with $l = 2$ connections are allowed between adjacent columns and columns with one column gap and so on. These values are fixed throughout the evolutionary process. The sets of integers in the genotype are G and F, with G values representing the connections and F values representing the function for each grid position.

The functions in the function set are indexed by integer numbers. The F set of the genotype is a sequence of n_j integers representing the function assigned to each node in the grid. The G set determines the connection origins for each connection target in the grid. Connection targets include node inputs and program outputs. Connection origins include program inputs and node outputs. All connection origins are indexed in a fixed way, with its total number being $n_i + n_j$. The G set contains a number of integers equal to the number of connection targets, which is $n_o + n_n.n_j$, with each integer representing a connection origin index. The output of a node may not be assigned to any connection target (node input or program output), in which case this node is inactive. The possibility of inactive nodes leads to variable phenotype sizes, although the genotype size is fixed. The phenotype size is, however, limited by the size of the grid.

One advantage of the fixed-size representation is that it facilitates the definition of recombination operators. A simple uniform crossover is proposed by the creators of CGP, where each value in the G and F sets is randomly chosen from one of the parents.

The mutation operator change values in the set to other random values, but some constraints have to be maintained so that valid programs are always generated. Let us consider the values:

$$e_{min} = n_i + (j - l).n_r$$

$$e_{max} = n_i + j.n_r$$

$$h_{min} = n_i + (n_c - l).n_r$$

$$h_{max} = n_i + (n_c - l).n_r$$

A program will be valid provided the following constraints are kept:

$$c_{kj} < e_{max}, j < l$$

$$e_{min} \leq c_{kj} < e_{max}, j \geq l$$

$$h_{min} \leq c_k^o < h_{max}$$

$$0 \leq c_k^f < n_f$$

In the expressions above, c_{kj} is the value of G for the k^{th} input of a node of column j, c_k^o is the vale of G for the k^{th} program output and c_k^f is the value of F for the k^{th} function node.

Initial experimentation with CGP was performed using these problems: symbolic regression of the polynomial $x^6 - 2x^4 + x^2$, the Santa Fe ant trail [Koza, 1992], even-parity functions (3, 4 and 5 bits) and the 2-bit multiplier. The n-bit even-parity problem consists of finding a boolean function with n inputs that outputs 1 if an even number of the inputs is 1, 0 otherwise. The n-bit multiplier problem consists of finding a boolean function with 2 n-bits inputs and one $2n$-bits output that returns the binary multiplication of the input values. These two last problems where chosen because they were known to pose difficulties to traditional GP [Koza, 1992], and where expected to benefit from circuit-like representations.

The algorithm was shown capable of finding good solutions for the first two problems, although no conclusive performance comparison to other methods was provided [Miller and Thomson, 2000]. For the last two problems, CGP was shown to be significantly more

effective than tree-based GP [Miller, 1999].

CGP was also applied to the evolution of robotic controllers [Harding and Miller, 2005]. Obstacle avoidance and maze navigation problems where addressed, and good quality solutions were found. These experiments were performed in a simulated environment modelling a Kephera-like robot with two wheels with independent motors and two distance sensors in the front.

Further experimentation shown that the simple crossover operator used in CGP not only does not help but hinders performance, so a new recombination operator was recently proposed [Clegg *et al.*, 2007]. This operator requires a change in the CGP representation. Instead of integers, genes are now real values in the $[0, 1]$ interval. These values encode the conventional integer values but from a more indirect representation. The decoding to integer values can be achieved by using the following expressions:

$$floor(gene_i^f * func_{total})$$

$$floor(gene_i^t * index_i)$$

The $floor(x)$ function returns the largest integer number not greater than x. The first expressions performs decoding for function genes ($gene^f$), the second for terminal genes ($gene^t$). The value $func_{total}$ is the number of functions in the functions set, while $index_i$ is the connection terminal number associated with the terminal gene $gene_i^t$. The idea is to create a more redundant and continuous genotype to phenotype mapping. The new crossover operator consists of uniformly generating a random number $r \in [0, 1]$, and then calculating each gene (g_o) in the offspring by combining the homologous genes in the parent (g_{p1} and g_{p2}) the following way:

$$o = (1 - r).g_{p1} + r.g_{p2}$$

The mutation operator needs to be adapted to this new representation. It is similar to the conventional CGP operator, but genes are changed to random real values uniformly generated in the $[0, 1]$ interval. This mutation operator is functionally equivalent to the

old one, and if no recombination is used, CGP with this new representation is equivalent to conventional CGP.

Experimentation with this new crossover operator was performed on the symbolic regression of the $x^6 - 2x^4 + x^2$ and $x^5 - 2x^3 + x$ polynomials, and it was shown to significantly improve the performance of the algorithm in both cases. The authors believe the effectiveness of this operator is due to its ability to search for solutions by sliding values in a continuous space, instead of performing discrete jumps in search space like the conventional recombination operator.

Embedded Cartesian Genetic Programming

Programs in CGP and PDGP are, by the nature of their representation, capable of reusing the output of functions. This is a step forward when compared to tree-based GP, but it is not as generic as ADF mechanisms, because functions can only be reused with the same input values. Recently, an extension to CGP has been proposed that attempts to address this limitation by evolving reusable modules [Walker and Miller, 2004, 2005]. It is called *Embedded Cartesian Genetic Programming (ECGP)*.

In ECGP, a row number of one ($n_r = 1$) is always used so the grid is reduced to a one-dimensional sequence of nodes. Unlike in standard CGP, the genotype size is variable, although bounded. Modules are represented in the same fashion as the main program, with sequences of integers that define function nodes and connections. The function set contains the primitive functions, as in CGP, but it also contains the defined modules for the individual. The number of modules is variable but bounded. The genotype segment for a module begins with a header with four integer values: the module identification number, the number of inputs, the number of nodes in the module and the number of outputs. Another difference to the representation in CGP is that node inputs are defined by two numbers. Since ECGP evolves modules with a variable number of ouputs, the source of a node input is defined by the source node number and the output number in this node.

A set of mutation operators are defined for the creation, removal and changing of modules: *compress*, *expand*, *module point mutation*, *add input*, *add output*, *remove input*

and *remove output*. The standard point mutation of CGP is also used.

The compress operator is the one that creates new modules. It selects two nodes from the main program and creates a new module formed by these nodes and the ones between them. The module is initialized with a number of inputs equal to the number of connections from nodes external to the sequence that defines the module or program inputs to nodes inside the sequence. In the same fashion, the initial number of module outputs is determined by the number of connections from nodes inside the sequence to external nodes or program outputs. The new module is added to the function set and the sequence of selected nodes is removed from the main program and replaced by one single *type I* node. This latter node is associated with the identification number of the new module and its inputs and output connections are defined in a way that preserves the exact same program as the one existing before the compress operation. *Type I* nodes are immune to point mutation and can only be removed from the genotype by action of the expand operator. The expand operator removes existing modules. It substitutes a *type I* node with the nodes from the module it was associated with, again updating connections in a way that preserves the resulting program. The compress and expand operators only perform structural changes to the genotype. They do not affect the associated phenotype. After a module is defined by a compress operation, it can be reused in the main program. Point-mutations can cause a node to be associated with a module, in which case this node is marked as *type II*. When selecting sequences for module creation, the compress operator is restricted in two ways:

- Do not select sequences that exceed in length the maximum allowed module size;
- Do not select sequences that contain *type I* or *type II* modules.

The second restriction prevents modules to be used inside modules. This restriction is put in place to prevent bloat and infinite loops. In artificial evolution, bloat is an useless increase in the size of evolved entities [Langdon, 1998], and can potentially affect any representation of unbounded size. The authors of ECGP claim that they will work to remove this last restriction in future work.

The remaining operator are responsible for changing modules. The module point

mutation is similar to the point mutation used in the main program, except that it is not allowed to introduce *type II* nodes, for the same reasons stated above for the compress operator restrictions. The other ones add and remove inputs and outputs to the modules. A module must have at least two inputs and one output, a maximum of inputs of twice the number of its nodes and a maximum of outputs equal to the number of its nodes. The operators follow these restrictions. When the number of inputs or outputs is changed in the modules, the genotype of the main program is changed to remain valid.

ECGP performance was compared with CGP on evolving four classes of boolean functions: even-parity [Walker and Miller, 2004], digital adders, digital multipliers and digital comparators [Walker and Miller, 2005]. ECGP was found to significantly outperform CGP on even-parity, digital adders and digital multipliers, but performed worse than CGP in digital comparators.

Further experimentation was performed with two other problems: the lawnmower problem [Koza, 1994] and the Hierarchical If-and-Only-If (H-IFF) problem [Watson *et al.*, 1998]. For the lawnmower problem, ECGP was found to perform better than CGP and also PDGP, with performance advantage increasing with the problem difficulty. For the H_IFF problem, however, CGP was found to perform better than ECGP. The authors provide no explanation for this difference in results, but hypothesize that allowing modules withing modules may provide better results for the latter problem.

Initial experimentation with ECGP only used mutations in the evolutionary process. This is possibly due to the fact that the conventional recombination operator hinders performance in standard CGP [Clegg *et al.*, 2007]. However, a multi-chromosome approach to recombination was proposed and found to cause significant speedups in the evolution of boolean circuits with multiple outputs [Walker *et al.*, 2006]. This approach consists of dividing the genotype into sections of equal length, which are the chromosomes. Each chromosome corresponds to a program output. The inputs of nodes in a chromosome can only be connected to the outputs of nodes in the same chromosome, or to input terminals. This results in a division of the problem into a set of sub-problems, each one related to the evolution of an individual sub-program per output. This technique was also applied with success to standard CGP.

2.7 Neuroevolution

2.7.1 Artificial Neural Networks

The *artificial neural network* [Costa and Simões, 2008] is a computational model inspired in the central nervous system of vertebrates. It is a connectionist model, where computation is performed by the propagation and processing of information in a network, where nodes are *artificial neurons* and connections are weighted.

Artificial neurons are simple processing units that can be characterized by three functions: input (f_i), activation (f_a) and output (f_o). Usually the input function is the weighted summation of the input signals:

$$f_i(j) = \sum_{k=0}^{n} w_{jk} \cdot x_k$$

, where w_{jk} is the weight of the connection from neuron j to neuron k, n is the number of incoming connections to neuron j and x_k is the current activation level of neuron k. The output function is usually:

$$f_o(j) = a_j,$$

where a_j is the activation level of neuron j.

Commonly used activation functions are the linear function, the saturated linear, the step and the sigmoid.

Another source of diversity in artificial neural networks is their topology. Simple networks have just input and output layers, while more complex ones have a number of hidden intermediary layers with any number of neurons. Networks may be *feed-forward*, in which case connections are only allowed to neurons in layers ahead, or *recurrent*, with connections allowed to any neuron.

Artificial neural networks can be used as adaptive systems, employing learning algorithms to improve their performance. Learning may be supervised, by way of human prepared training examples and an algorithm like backpropagation, or unsupervised, as in the system described in section 3.3.1.

2.7.2 Evolutionary approaches

Neural networks may be directly evolved, for example using a genetic algorithm. This process is called neuroevolution (NE) [Angeline *et al.*, 1993; Moriarty, 1997; Siebel and Sommer, 2007].

With evolutionary algorithms, it is common to use fixed-sized genomes. In many genetic representations used in optimization problems, for example, the size of the genome that encodes a solution is either known in advance or limited to a range. This approach tends to work when the complexity of phenotypes in the search domain is constant or fairly homogeneous. There are other problems for which possible phenotypes vary greatly in complexity. That is typically the case with the evolution of agent controllers like artificial neural networks.

Two aspects of neural networks may be evolved: the connection weights and the topology. Defining neuroevolutionary algorithms is easier if a fixed topology is established and only connection weights are evolved. Fixed-sized genomes may be used and the definition of genetic operators is straightforward. In fact, many neuroevolutionary algorithms, especially the earlier ones, follow this approach [Moriarty, 1997; Gomez and Miikkulainen, 1998; Igel, 2003]. Fixed topology approaches have several important limitations. Human intervention is needed to estimate a suitable topology to use. This may not be an easy task. Evolved networks will always be constrained by the chosen topology, which may not allow for more interesting solutions. In open evolution scenarios, diversity may be compromised. Fixed topology may also create scalability problems. Complex topologies required by harder problems can cause the search space to become to large.

Neuroevolutionary algorithms that evolve both connection weights and network topology are designated *Topology and Weight Evolving Neural Networks (TWEANNs)* [Yao, 1999]. TWEANNs do not suffer from the problems caused by fixed topologies but pose other difficulties. Establishing genetic representations become harder because they must include the structure of the network and variable-sized genomes must be used. Mutation operators must perform not only connections weight changes but also topological changes. Topological changes are designated complexification if the number of connections is increased and simplification if it is decreased [James and Tucker, 2004]. Viable recombina-

tion of networks with different topologies is much harder than recombination with fixed topologies. Some TWEANN approaches discard recombination altogether [Angeline *et al.*, 1993].

2.7.3 Neuroevolution of augmenting topologies

Neuroevolution of augmenting topologies (NEAT) [Stanley, 2004; Stanley and Miikkulainen, 2004] is one of the most well-known TWEANNs at the moment. It was applied to real time brain evolution in a multi-agent simulation, the video game NERO [Stanley *et al.*, 2005, 2006].

In NEAT, a genome consists of two lists, one of nodes and another one of connections. Both these lists are of unlimited size, allowing for unbound complexification. Node genes contain the node number and the node type, which can be input, output or hidden. Connection genes contain the numbers of the origin and target nodes, the connection weight and a flag that indicates if the gene is enabled or not. Only enabled genes are expressed in the phenotype.

NEAT uses three mutation operators: change weight, add connection and add node. Change weight is an operator common to most neuroevolutionary approaches that alters the weight of connections. Add connection creates a connection between two unconnected nodes. Add node splits an existing connection in two with a new node in the middle. The old connection is deactivated. The first new connection is given a weight of one and the second new connection is given the weight of the old connection. Add connection and add node are the operators responsible for network complexification.

To solve the problem of performing recombination between two individuals with different topologies, NEAT uses *global innovation numbers*. When a new gene is created, the system attributes an innovation number to it. Each time a new number is needed, it is generated by incrementing a global value. Genes propagated to descendants inherit their innovation number. This way, when performing recombination, it is possible to known which genes match on each progenitor. Genes that do not match are considered disjoint if their innovation number is within the range of the other progenitor or excess otherwise. Genes that match are inherited randomly from each progenitor, while genes that do not

match are inherited from the fittest progenitor. If both progenitors are of equal fitness, genes that do not match are also inherited randomly. Disabled genes have a 0.25 probability of being enabled after recombination, so evolution keeps attempting to reuse older genetic code.

Another problem is that evolution is not capable of maintaining topological innovations. Smaller networks are easier to optimize than larger networks, and most complexification mutations cause an initial decrease in fitness. For complexification to be possible, it is necessary to introduce a mechanism that protects innovation in the population. In NEAT this mechanism is a form of speciation. Individuals are divided in species according to topological similarity, so that individuals compete mainly within their own niche. This way, topological innovations have a chance to optimize before having to compete with the population at large. Speciation is also used as strategy to avoid bloat, since smaller networks are kept while their fitness is competetive and tend not to be unnecessarily replaced by more complex topologies.

To divide the population in species, a distance δ between pairs of networks is used. This distance is given by the expression:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W}$$

.

E is the number of excess genes, D is the number of disjoint genes, $\overline{W}$ is the average weight difference in matching genes and N is the number of genes in the largest genome. c_1, c_2 and c_3 are coefficients that adjust the importance of the three factors. Genomes are compared one at a time with existing species. If a species is found for which the distance to a species member of the previous generation is lesser than a compatibility threshold (δ_t), the genome is assigned to this species. If the genome is not found to belong to any of the existing species, a new one is created. To force diversity and promote and prevent one species from dominating the population, fitnesses are adjusted using a mechanism called *explicit fitness sharing*. Adjusted fitness is computed by the following expression:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))}$$

The sharing function sh returns 0 if its parameter is greater than δ_t, 1 otherwise. This way, fitness will be adjusted to the number of individuals in the same species. This prevents species from becoming too dominant and rewards originality.

2.7.4 Neuroevolution with Analog Genetic Encoders

Analog Genetic Encoders (AGE) [Mattiussi, 2005] is an evolutionary representation for general purpose analog networks that can be applied to neural networks [Durr *et al.*, 2006].

The AGE representation consists of a set of chromosomes that are finite sequences of ASCII characters. The set of admissible characters form the *genetic alphabet*. Certain specific sequences of characters are called *tokens*. Tokens represent types of nodes in the network or control sequences. A set of node types is defined for the type of network being evolved. They may be components in an electronic circuit of types of artificial neurons in a neural network. Each type is represented by a specific, predefined sequence of characters. Control sequences may signal the end of a terminal sequence or a parameter sequence. Terminal sequences represent the interface of a certain terminal of the node. The number of terminals in each node depends on its type. Artificial neurons, for example, have an input and an output terminal, while a transistor has three terminals: collector, base and emitter. The specific terminal that each sequence corresponds to is determined by the order in which it appears after the node type token, and is predefined. Parameter sequences again relate to the type of node. They can represent, for example, activation levels in artificial neurons.

Types of nodes have a predefined number of terminals and parameters that they required. A sequence of characters in a chromosome may not translate to a valid node definition, in which case it is *non-encoding*.

The translation from genotype to phenotype is performed by extracting the circuit represented by the sequences of characters. This consists of generating the set of valid nodes, parameterizing them and then establishing the connections between them. Connec-

tions have a strength, that can have different meanings depending on the type of network. This can mean a connection weight in a neural network or conductivity in an electronic circuit. The strength is determined by matching the terminal strings for every two pairs of terminals that exist. The lesser the number of character substitutions that have to be performed to transform one string into the other, the higher the connection strength. A substitution matrix is defined, where a score for each transformation from one character to another (including blank spaces for strings with different sizes) is defined.

Genetic operators are defined at several levels: insertion, deletion and substitution of characters; duplication, duplication of complement, deletion, transposition and insertion of tokens in chromosomes; duplication and deletion of entire chromosomes; recombination of pairs of chromosomes and duplication and trimming of the entire genome.

AGE could potentially be applied to evolutionary multi-agent simulations, but we have no knowledge of such work to this date.

Chapter 3

Evolution in Multi-Agent Systems

This chapter is dedicated to the state of the art in evolutionary multi-agent systems. We describe several evolutionary multi-agent systems, grouped by agent brain model. A large number and variety of these type of simulation has been developed in the last decades, so we focus on systems that represent hallmark ideas in the field and that served as inspiration to the models that we will present on the following chapters.

3.1 Symbolic and rule-based systems

3.1.1 ECHO

Echo is a class of simulation models conceived to perform experimentations with CAS [Holland, 1995, 1999; Jones and Forrest, 1993; Forrest and Jones, 1994; Schmitz and Booth, 1996; Hraber *et al.*, 1997; Hraber and Milne, 1997]. These models are very abstract, more geared towards thought experiments than practical, real-world scenarios. They were designed with the goal of providing the simplest possible basis for the study of systems with the characteristics described in section 2.3.

Echo is defined as a simple model that can be extended in several ways to support the full range of CAS characteristics. We will begin by describing the base model and then the several extension models proposed by Holland. It is important to notice that some changes were introduced to the model along time, as can be seen in publications related to Echo. We present what we believe to be the most recent description of the models.

The Echo world consists of a network of sites. The geographic configuration and neighbourhood of these sites is to be determined by the designer of the simulation and a large degree of freedom is allowed. Each site contains a resource fountain that emits a certain number of resources to the site per simulation step. There are four types of resources, represented by the letters $[a, b, c, d]$. Each fountain is characterized by the number of each one of these resources outputted per simulation step. An agent consists of two things: a resource reservoir and a chromosome string. The chromosome string is a sequence of the resource letters and the reservoir is capable of holding an arbitrary number of resources.

In the basic model, an agent is capable of reproducing once its reservoir contains sufficient resources to create a copy of the chromosome. The basic chromosome defines two tags. These are the offense and the defense tags. When two agents interact, the offense tag of one agent is compared to the defense tag of the other and *vice-versa*. Tags are matched by left-aligning them and then comparing letters one by one. A table is defined that gives a score for each letter pair, which may be positive or negative. A score is also defined for cases where one of the tags has no letter at that position, which happens if tags are not of the same size. The matching score is the summation of these comparisons, and determines an amount of resources that the attacking agent can retrieve from the defending agent's reservoir.

The basic model is capable of creating interesting scenarios, but does not allow for the full range of CAS characteristics.

The designation "offense tag" is somewhat misleading, in that the extension models we will describe in the following section use it as a generic organism identification tag, to which other tags that will be introduced are compared.

Extension models

The first extension model defines conditional exchanges, allowing agents to reject interactions. This is achieved by adding an exchange condition to the chromosome. From this point on, we will consider the chromosome divided in two regions, tag and control. The previous offense and defense tags belong to the tag region, while definitions like the ex-

change condition belong to the control region. When an interaction takes place, each agent tests the other agent's offense tag against its exchange condition. The d letter is selected to work as a "don't care" symbol. Since tags may have different sizes, for the purpose of condition checking each exchange condition is assumed to have an infinite number of d letters at its end. Conditions are satisfied if all position of the tag match the condition segment. An interaction between two agents will thus test two conditions (one in each direction) and is aborted if neither condition is satisfied. If only one condition is satisfied, there is a pre-defined probability that the interaction is aborted. Conditional exchanges may serve as a building block for mechanisms like message passing.

The second extension model provides agents with the ability to transform resources. This ability is expressed by extra segments in the control region of the chromosome, which we will call *transformation segments*. This transformation ability applies to resources present in the agent's reservoir. An example is the transformation of resource a into b, which would be expressed by the transformation segment ab. The transformation rate for each segment is defined as two resource units per simulation cycle, so that even the shortest lived agents can benefit from the mechanism. The shortest lifespan for an agent is one simulation cycle. Such an agent would consume one unit of a certain resource to define a transformation segment that produces that resource. This way, with a transformation rate of two units per simulation cycle, the agent can still take advantage of the mechanism. Several copies of the same transformation segment will increase the transformation rate. This extension model is related to the *flows* property discussed in section 2.3.

The third extension model is called *adhesion*. It is clearly related to the *aggregation* property discussed in section 2.3, as it provides a mechanism for the creation of agent aggregates. This is achieved by the introduction of a new concept: boundaries. The original environment with the fountains and simple agents may be considered the outermost boundary. Each boundary can contain an arbitrary number of other boundaries, forming a tree of limitless depth. Every agent belongs to a boundary and can only interact with agents in the same of adjacent boundaries. Adjacent boundaries are the one directly containing another and the ones directly contained by another. By directly we mean that they are only one depth level apart in the boundaries tree. We can thus see that bound-

aries constitute, in practice, agent aggregates. They allow for the formation on layers in the simulation. With this mechanism, agent aggregates can serve as building blocks for higher level agents. Each agent is assigned to a boundary at the moment of its creation. For this purpose, a new tag is included in the chromosome, the *adhesion tag*. When an agent is created, another agent is selected for adhesion matching. Usually this agent is a progenitor, but it may be another agent in the interaction realm of a progenitor according to a predefined probability, to allow for some mobility. Agents in this pair are matched to each other, the adhesion tag of one with the offense tag of the other, and two scores are calculated. If both scores are close to zero, the agents do not adhere. The offspring is placed in a new boundary that contains only itself. If scores are close to each other, the offspring is placed in the boundary of the other agent. If the match score of the other agent is significantly higher than that of the offspring, the offspring is placed in a boundary directly interior to the other agent's one. If no such boundary exists, one is created. Conversely, if the match score of the offspring is much higher that that of the other agent, the other agent is moved to the interior of the offspring's boundary.

The forth extension model is called *selective mating*. It allows for the emergence of species in Echo. It also allows for a type of reproduction that involves two progenitors and genetic recombination operators. Again a control segment is created to implement a mating condition. When an agent has acquired the necessary resource for reproduction, it performs a search for a mate. This search involves checking mating conditions with suitable agents until a match is found. Suitable agents may be the set of agents in the interaction realm of the agent that initiates mating, and that have themselves acquired sufficient resources for reproduction. The check to see if mating is possible is done by comparing each agent's mating condition to the other's offense tag. The mating is performed if both scores are high enough. The reproductive process itself consists of two steps: first each progenitor produces a copy of its chromosome, using the resources in its reservoir. The two chromosomes are then crossed. The resulting chromosome are subjected to mutation and then the respective new agents are placed in the environment.

The fifth and final extension model is that of *conditional replication*. In nature, multicellular organisms are capable of developing from a single initial cell. Embryogenesis in

animals, for example, is possible because of *cell differentiation*. The process of cell differentiation is possible because certain cells are capable of developing in more specialized types of cells. This is the case of stem cells, meristematic cells, zygotes and early embryonic cells. The process of embryogenesis has analogies in other CAS systems, for example in the formation of social structures. This type of generative process is included in Echo by way of the *multiagent* structure. The multiagent is an extension of the aggregate. It too is internally constituted by a tree of agents and agent aggregates. The differences reside in two aspects: the process of reproduction and the form of interaction. The purpose of the multiagent is to allow for agents with the same genetic material to diversify and develop into distinct and varied forms.

A multiagent reproduces as a whole. The multiagent shares its internal resources. This way, reproduction becomes possible when the total resources present in all the reservoirs of the agents that constitute the multiagent are sufficient to create a copy of the entire genetic material of these same agents. Agents that constitute the multiagent maintain their individual chromosomes that can be different from each other. The goal is to define a replication process where the same genetic material can result in the expression of different multiagents. For this, two things are added: an *active flag* to agents and a new segment to the control region, called the *replication condition*. An agent becomes active when it participates in an interaction. At the moment of reproduction, the replication condition of every agent chromosome is compared to the offense tags of active agents in the multiagent aggregate. The agent is expressed in the multi-agent offspring only if a match is found. Every chromosome is still copied to the offspring, and has a chance of being expressed in later reproductions. This mechanism ties the multiagent activity to its genetic expression in offspring.

When a multiagent interacts with another multiagent or simple agent, an agent in its outer boundary is randomly selected to perform the interaction. Another random agent is selected each time. Apart from this, interactions take place in the previously described ways.

Multiagents emerge in the simulation through a new mutation mechanism that randomly promotes aggregate boundaries to multiagent boundaries.

```
while simulation running:
    for each site:
        generate list of aggregate pairs in exchange contact

        for each exchange pair:
            determine point of contact agents

        for each exchange pair:
            calculate match score
            exchange resources
            calculate adhesion score

        for each agent:
            calculate resource transformation

        generate list of mating contacts

        for each successful mating pair:
            recombine chromosomes and generate offspring
            apply mutations to offspring
            determine offspring boundary

        perform agent migrations

        process agent deaths
```

Listing 3.1: Main loop of an Echo simulation.

Simulation cycle

An Echo simulation is initialized with a number of simple agents with empty control segments. They simply reproduce upon acquiring sufficient resources from the environment. From this initial, sustainable state, complexity is expected to emerge by way of the mechanisms previously detailed.

The pseudo-code that describes an Echo simulation cycle implementing all the extension models is presented in listing 3.1. Two general processes of the simulation where not yet discussed: agent migration and agent death. Both these processes are left open and can be implemented in different ways. Agent migration takes care of moving agents to other sites, so that the agent population can propagate across the territory. Agent death takes care of continually removing some agents from the simulation, otherwise the population size would explode, the simulation would become computationally unfeasible

and evolutionary pressure would not exist.

A very simple mechanism for migration is to randomly select agents to move to contiguous sites. A more adaptive possibility is to select agents with low resources to move. As for agent death, the simplest mechanism is to randomly select an existing agent for deletion when a new one is generated. A more sophisticated alternative is to define maintenance costs, that the agents must pay with resources from their reservoir. Failure to pay the cost results in an increased probability of deletion.

Experimental results

A study was performed to assess species diversity in Echo simulations [Forrest and Jones, 1994]. The authors use clustering techniques to group agents by genotypes. They found some of the resulting distributions to approximate Preston's canonical lognormal distributions [Preston, 1948], which are a model of species distribution based on data collected in natural ecosystems. The authors found these results to be encouraging in terms of Echo's ability to generate complex adaptive systems.

In [Smith and Bedau, 2000] experimental results are gathered from simulation runs in order to access if Echo does indeed present the characteristics of CAS. This work was performed with version 1.3 beta 2 of the Santa Fe Institute implementation of Echo. This is still the last version available of this implementation as of the writing of this document. It does not implement the *adhesion* nor the *conditional replication* extension models. The mechanisms of *tags*, *internal model* and *building blocks* are present in the system by design, so the author focus on verifying if the four properties of CAS, as defined by Holland, are present. They conclude that the system does indeed show non-linear behavior and that resource flows occur, although of a simple nature. Genotypic diversity emerges, especially at higher mutation rates, but it does not translate into phenotypic diversity. There was no evidence of aggregation nor of the emergence of hierarchies of phenomena. The authors conclude that the simulation, while being an important step forward in research in the field, is not a CAS. It is important to notice that two extension models related to aggregation are not present in the implementation used.

More recently, a new implementation of Echo was developed [McIndoe, 2005] that

includes the *adhesion* extension model. The author of this work collected experimental results that indicate the emergence of aggregates, bringing the simulation closer to displaying the full range of CAS properties. It is still uncertain if this type of simulation is capable of true ongoing diversity, as observed in nature. To our knowledge, there is no implementation of Echo that includes the last extension model, so far.

3.1.2 New Ties

New Ties is a socio-biological multi-agent simulation [Gilbert *et al.*, 2006; Eiben *et al.*, 2007]. Agents live in an environment modeled as a discrete rectangular grid and the state of the world is updated in discrete time steps. Agents are capable of performing actions in this environment, like moving, turning, mating, talking and picking up objects. They also have properties like weight, color and shape. They have a controller that determines their next action based on sensory information from the environment.

The agent controller used is a *Decision Q-Tree (DQT)*. The DQT is a weighted tree with two types of intermediary nodes, *test* and *bias* and leaf-nodes that represent actions to take. Test nodes represent a view of the agent's current situation based on its sensory information. Examples of used test nodes are `my-energy-low`, `food-ahead` and `female-nearby`. Bias nodes represent probabilistic branching points. The probability of a branch from a bias node to be chosen is proportional to the relative weight of the subtree of this branch to the other subtrees of the other branches of the node. A decision is formed by traversing the tree.

The authors of New Ties classify it as a *Population Based Adaptive System*, and its agents are capable of three types of adaptation: *evolutionary learning*, *individual learning* and *social learning*. These adaptations take place at controller level. Evolutionary learning results from reproduction with variation. When two agents mate, their original controllers are recombined and mutated to produce the offspring controller, using genetic operators similar to the ones used in tree-based genetic programming. The controller of an agent can change during its lifetime due to the other types of adaptation, but the original controller is used in evolutionary learning to keep the process non-Lamarkian. No explicit fitness functions are used, nor a central population control algorithm. Agents take the initiative

to reproduce, and the more successful ones will produce more offspring.

Individual learning is a form of reinforcement learning that takes place during the lifetime of the agents. There is a rewarded system based on energy variations that causes the weights of the edges of the tree to be adjusted. Social learning is performed by agents communicating subtrees from their DQTs to other agents. When this process takes place, the agent that accepts the information incorporates the received tree at an appropriate location of its own DQT, by creating a bias node that branches to the accepted subtree and to the original subtree that was already there. Weights in these branches are created based on the relative age and energy level of sender and receiver.

Using this three types of learning, New Ties defines a simulation model that merges biological and social processes. It is the goal of the project to study the interaction between these processes in the context of a multi-agent simulation.

3.2 Artificial organisms as computer programs

One possible approach to the synthesis of life in a digital medium is to abstract the fundamental processes of life found in nature and apply them to the low level building blocks of computation. The building blocks of biological life can be found at the molecular level, emerging to higher level structures through chemical interactions. The building blocks of computation in *von Neumann* machines are processor instructions. The execution of sequences of these instructions give rise to higher level behaviors. In the conventional use of digital computers, a human programmer writes sequences of these instructions to achieve a certain goal. The hybrid approach of synthesising life as computer programs takes Darwinian evolution and applies it to computational environments where the processor instruction is the fundamental building block.

3.2.1 The Core War game

The Core War game [Dewdney, 1984, 1988] was one of the first instances of the idea of modelling virtual organism as machine code. In Core War, programs created by humans compete for control of a virtual machine by causing competing programs to terminate.

This virtual machine consists of a circular memory array of 8000 addresses and a machine code interpreter. The language used, Redcode, is similar to assembly languages used with modern processors but much simpler, consisting of a set of 10 instructions in its first implementation. In the begining, two competing programs are loaded into arbitrary positions in memory. Programs are executed in parallel, following a simple algorithm of time sharing that executes the next instruction from each program at a time. Programs begin execution in a single process but can create new ones. Processes are time-shared among the owner program's execution time.

The simulation stops when a program causes an execution error and this program loses the game. For a program to win the game it must damage the other one by altering its code. Defensive strategies can also be used, like self-repair and replication for the purpose of changing position in memory, hiding from the predator. Replication and process forking can be used to create new organisms.

Core War contains an important ingredient for life simulation, which is replication, but misses another equally important one: evolution. It served as inspiration to Tierra, an artificial life simulation based in machine code organisms.

3.2.2 The Tierra environment

In the early 1990s, ecologist Thomas S. Ray created Tierra [Ray, 1992], an artificial life simulation where machine code creatures compete for computational resources on a virtual machine. Like Core War, the Tierra simulation consists of a virtual machine executing programs in a memory array. The fundamental differences introduced in Tierra are fault injection and garbage collection. These two features introduce mutations and selective pressure into the system, thus enabling evolution.

Each creature/program has its own *Central Processing Unit* or CPU. A CPU contains five registers (two for values, two for addresses and one for errors), a ten word stack, a stack pointer and an instruction pointer. The CPU keeps executing a loop of fetching the next instruction indicated by the instruction pointer, executing the instruction and incrementing the instruction pointer. When the last instruction is reached, the instruction pointer is assigned the beginning address of the memory block, so that programs run in

an endless loop.

CPUs execute instructions of a language called Tierran, that is similar to machine code computer languages used by modern processors, although simpler. The Tierran set of 32 instructions is listed on Table 3.1.

Parallel execution of the several CPUs is assured by a time sharing algorithm called the slicer. The slicer has a circular queue of processes and a number of instructions to be awarded by time slice to each process. The slicer may be parameterized to assign time slices proportional or inversely proportional to the instruction size of programs, thus promoting larger or smaller creatures.

Programs are only allowed to write in memory blocks allocated to themselves and their child, but can read and execute code from any memory position.

In the beginning of simulations, memory is initialized with a number of instances of a human created program, called the ancestor. The ancestor is capable of self replication. To do so it finds its own size, allocates a block of memory of that size for its descendent, copies its own code to that memory block and ends by calling the Tierran instruction `DIVIDE` that creates a new process for the descendent. This new process starts executing the code of the descendent in its own memory block.

Execution of self replicating programs will necessarily result in the exhaustion of memory space. Another algorithm, called the reaper, is used to remove excess creatures. The reaper consists of a FIFO queue. New creatures are inserted at the beginning. When memory usage exceeds a certain pre-defined percentage, creatures are removed from the end of the queue and killed. Killing a creature consists of deallocating its memory block and removing it from the slicer queue. The data in the memory block is kept, but this memory space is now available for allocation by other programs. Generation of error codes by programs is penalized by moving them up in the reaper queue, and correct execution of certain instructions is rewarded by moving them down.

The described system would reach equilibrium and not evolve if it was not for a final ingredient: mutations. A rate is defined for random bit flipping in the entire memory. Bit flipping also occurs in copy operations, according to another rate. There is also an error probability in the execution of instructions, making the outcome of programs non-

deterministic.

Mutations and non-deterministic execution combined with finite life times enforced by the reaper algorithm introduce selective pressure. The basic ancestor will evolve into other types of organisms according to an increase in implicit fitness. In an analogous fashion to nature, organism's fitness is determined by its ability to survive and reproduce as well as destroy or take advantage of competing organisms.

The Tierran language is an effort in bridging computer technology and biological technology. Although based in conventional machine code, several aspects of it are designed to make it a suitable target for evolution. One of the main problems with conventional machine code from an evolutionary stand point is brittleness. As the author of Tierra states in [Ray, 1992]:

> "Von Neumann type machine languages are considered to be *brittle*, meaning that the ratio of viable programs to possible programs is virtually zero. Any mutation or recombination event in a real machine code is almost certain to produce a non-functional program. The problem of brittleness can be mitigated by designing a virtual computer whose machine code is designed with evolution in mind."

To address this problem and reduce the combinatorial explosion in generated programs, Tierran instructions are designed to not need explicit parameters. Even Redcode, mentioned in section 3.2.1, although only consisting of 10 instructions in its first incarnation, will result in much greater combinatorial explosion than Tierran because of parameters. As an example, the `MOV` instruction in Redcode accepts two parameters, origin and target, where origin is an integer value and target is a relative memory address. This leads to an explosion of possible combinations of origins and targets in a single `MOV` instruction. Tierran, on the other hand, uses implicit parameters. As can be seen in Table 3.1 , Tierran defines three `MOV` instructions with predefined origins and targets. The same strategy is used for arithmetic and stack operations.

Explicit parameters are avoided in instructions that need an addressing mechanism by using template addressing. Template addressing is a bio-inspired mechanism based

Hex code	Name	Description
00	NOP_0	no operation
01	NOP_1	no operation
02	OR1	flip low order bit of cx
03	SH1	shift left cx register
04	ZERO	set cx register to zero
05	IF_CZ	if cx is 0 execute next instruction
06	SUB_AB	subtract bx from ax
07	SUB_AC	subtract cx from ax
08	INC_A	increment ax
09	INC_B	increment bx
0A	DEC_C	decrement cx
0B	INC_C	increment cx
0C	PUSH_AX	push ax on stack
0D	PUSH_BX	push bx on stack
0E	PUSH_CX	push cx on stack
0F	PUSH_DX	push dx on stack
10	POP_AX	pop top of stack into ax
11	POP_BX	pop top of stack into bx
12	POP_CX	pop top of stack into cx
13	POP_DX	pop top of stack into dx
14	JMP	move ip to template
15	JMPB	move ip backward to template
16	CALL	call a procedure
17	RET	return from a procedure
18	MOV_CD	move cx to dx
19	MOV_AB	move ax to bx
1A	MOV_IAB	move instruction at address in bx to address in ax
1B	ADR	address of nearest template to ax
1C	ADRB	search backward for template
1D	ADRF	search forward for template
1E	MAL	allocate memory for daughter cell
1F	DIVIDE	cell division

Table 3.1: The Tierran instruction set.

in the way that protein molecules interact. Protein molecules that have complementary surfaces bind. The conformations of molecules define the mechanism by which they "find" each other. Tierran provides two instructions that do nothing but allow the definiton of templates: `NOP_0` and `NOP_1`. A specific sequence of these instructions can be searched for by instructions that want to address a certain part of memory. For example, the `JMP` instruction moves the instruction pointer and thus program execution to a certain position in memory. To find this position, it searches forwards and backward in memory for the complement of a certain pattern. The sequence of instructions `JMP NOP_0 NOP_1` will cause a jump to the nearest occurrence of the sequence `NOP_1 NOP_0`.

Some interesting organisms with behaviors analogous to biological scenarios where shown to emerge from the Tierra simulation, mostly related to parasitism and an arms race between hosts and parasites.

Parasites in Tierra are creatures that do not have their own copy mechanism but are capable of making use of the copy mechanism of other organisms. Parasites evolve to match the template that signals the copy code in other organisms and send its instruction pointer there. Organisms have been shown to evolve resistance to parasites, and parasites to evolve circumventions to this resistance, in an arms race scenario.

3.2.3 Avida - spatial locality through a two dimensional grid

In 1994, Charles Ofria, Chris Adami and Titus Brown introduced Avida [Adami and Brown, 1994; Ofria and Wilke, 2004], an artificial life simulation based on Tierra. Avida extends the Tierra concept to incorporate spacial locality through a two dimensional grid topology. Avida is under active development to the current day and has achieved mainstream scientific acceptance as a theoretical biology research tool [Lenski *et al.*, 1999; Wilke *et al.*, 2001; Lenski *et al.*, 2003; Chow *et al.*, 2004].

We will proceed to describe the architecture of Avida as specified by its technical manual [Ofria *et al.*, 1997].

Avida organisms are placed in a toroidal two dimensional grid and are only allowed to interact with neighbouring cells. This introduces a spatial dimension to the simulation and enforces locality, expecting to approximately model biological self-organizing systems.

Like in Tierra, organisms are strings of machine code instructions. A similar language is used, extended to support the spatial aspects of Avida.

Another innovation introduced is fitness rewards that are not directly related to reproduction. Creatures contain an input and an output buffer that allows them to read values from the environment and write values to the environment. Energy rewards are given to creatures which successfully execute pre-defined operations, like adding two values. Since phenotypical features are promoted, creatures are free to evolve to one of the many possible genotypes that express this feature. This is expected to contribute to the achievement of open-ended evolution.

Like in Tierra, energy is modeled as processor time and a time slicing algorithm is needed to allocate time slices to organisms. Several time slicing modes exist, but we will discuss "integrated", the default and the one that mostly relates to Avida's goals. The "integrated" time slicer allocates time slices to each creature with a number of instructions proportional to its merit. Merit is a metric that has an initial base value and can be increased by successful execution of simulation predefined input/output operations. Different merit increases can be awarded by the simulation for different operations, typically in a fashion proportionally related to its difficulty. Base merit is related to the creature size. It can be configured to be proportional to full size, executed size, copied size or the minimum of full size and copied size. The default strategy is the last one. If base merit is awarded only for full size, creatures will tend to develop a large amount of junk instructions. Junk instructions are never executed or copied to descendants, and are used for the sole purpose of gaining larger time slices. By awarding base merit only for executed or copied instructions, this behaviour is avoided.

The Avida time slicer and merit system superimposes a fitness landscape over the organisms evolutionary process, by encouraging organisms to adapt to the execution of tasks at the same time that they optimize their reproductive mechanisms.

As stated, reproduction in Tierra consists of three basic steps:

- Parent allocates memory for child.
- Parent program writes to child program's memory, usually copying itself.

- Parent executes division instruction and launches child as independent process.

Avida introduces a forth step:

- Placement of the child program into a cell in the grid.

While the three initial steps are under the control of the organism, the fourth step is managed by the environment and the programs have no way of interfering with it. Child organisms can only be placed in the immediate neighbourhood of the parent cell, consisting of the 8 adjacent cells or of the parent cell itself. The placement algorithm will always look for a free cell in the neighbourhood for child placement. Only if none is found will it replace the organism in an occupied cell. Various strategies are available to determine the organism to be replaced, the default one being to select the oldest organism, including the parent. Other possible strategies are: choose randomly, choose highest age/merit ratio and choose empty (reproduction is only allowed if empty neighbouring cell is available).

Since a grid cell can only be occupied by one organism at a time and the default reproduction algorithm causes the replacement of older organisms, there is no need for a reaper algorithm. The number of possible living organism is implicitly limited by the size of the grid and the reproduction process itself causes the removal of older organisms.

It is interesting to notice that the introduction of a spatial dimension on the Tierra model forced the system designers to make the environment rules more complex.

Three types of mutations are used:

- Point
- Copy
- Divide

Point mutations flip bits randomly chosen from the entire simulation memory at Poisson-distributed random times. They are inspired on the effects of cosmic rays. Copy mutations can occur when organism are copying its instructions to their child, causing a flawed copy. Divide mutations can occur in the parent's memory after a divide event and consist of changing, inserting or deleting of instructions. All these mutations are random

events happening according to predefined rates. Two instructions are available, although not used by default, that allow organisms to alter their own copy mutation rates.

Code that was corrupted by mutations may commit errors when copying itself to child memory, thus causing implicit mutations. These types of mutations have been observed but, to our knowledge, not thoroughly studied. One example of such implicit mutation behaviors was named *necrophilia* and consists of an organism only partially copying itself to memory previously occupied by another organism, resulting in a merge of the two codes [Ofria *et al.*, 1997].

The CPU architecture is similar to Tierra's, introducing two buffers, one of input and another for output. Each buffer has a pointer indicating the current position within it. The CPU also contains a *facing* register that holds the relative position of the neighbouring cell that the organism is pointing towards. The instruction set is extended with input and output operations for the buffers and rotation operations to change the organism facing direction.

Contributions to Theoretical Biology

The Avida platform is well-known in the Artificial Life community for its effectiveness as a theoretical biology research tool, having been used in recent years to provide actual contributions to this field.

One example of such a contribution is a study of three types of selective pressures on the molecular evolution of genomes [Ofria *et al.*, 2003]. An interesting aspect of this work is that it helps bridge the gap between computer science and molecular and evolutionary biology by applying Shannon information theory to the study of Darwinian evolution. Evolution at the molecular level is modeled as a set channels through which messages are sent, from progenitors to offspring. Genomes are seen as codifications of these messages and mutations as noise in the channel. The three types of selective pressures are studied under this light. The authors of the work name them: compression, transmission and neutrality selection.

Compression selection results from the fact that information transmission has a cost, and the larger the message, the larger the cost. One obvious way for an organism to

reduce the total cost of its replication is by shortening its genome or, under this model, the message. This behavior was simulated in Avida by letting digital organisms adapt to a complex environment and then lowering the complexity of the environment. It was experimentally observed that the size of the organism's programs shortened after some time of adaptation to the simpler environment.

Transmission selection is based on the concept that organism genomes contain information about the environment that is relevant to survival. Progenitors will transmit this information to offspring. It is expected that the amount of information stored by the organisms about their environment will be a function of environmental complexity. From an information theory perspective, the information channel from progenitors to offspring is expected to increase its bandwidth if the environment becomes more complex. This scenario was tested in Avida by running a set of three simulations. In these simulations, time slices awarded to organisms were defined to be proportional to program size, to remove compression selection. The three simulations were set to have an increasing number of rewards for the execution of logical operations by organisms: none, 10 and 78. As expected, program size of organisms after some time was larger the higher the complexity of the environment, in this case determined by the number of rewards.

Considering mutations as noise in the communication channels, neutrality selection is a pressure to increase robustness of messages passing through these channels. Not all mutations affect fitness. This results from the fact that different messages may correspond to the same meaning. These different messages may, however, have different success rates in transmission through the noisy communication channel. It is expected that neutral mutations that increase fault tolerance in messages have a tendency to be selected, even though they do not directly affect organism's ability to survive. This is because the information expressed in these messages has an higher chance of being preserved across several transmissions.

Neutrality selection was tested by defining a simulation scenario where the program length of all organisms is set to 100, thus disabling compression and transmission selective pressures. The complexity of the environment is also kept constant, and only the mutation rate is changed. Genetic neutrality in a population is measured by generating all possible

one point mutations of the most frequent genotype and then determining their fitness. A mutation is considered neutral if the fitness change is less than $1/N$, with being N the population size. Neutrality is then calculated as the fraction of these mutations that are not detrimental to fitness, as those are the ones that do not delete information from the message. Experimental results shown that neutrality increases with mutation rate. It is concluded that as higher mutation rates introduce stronger selective pressure for message robustness, thus causing an increase in neutrality.

Both selective and transmission selection are well studied and established in theoretical biology. This work nevertheless provides a new experimental tool to test and study these behaviors. Neutrality selection is a less studied idea. The experimental results of this work help understand why most mutations are neutral in terms of the organism phenotype in biological systems [Kimura, 1983].

3.3 Agents controlled by neural networks

3.3.1 Open evolution in the Polyworld multi-agent simulation

Polyworld simulates an ecology of agents under open-ended evolution [Yaeger, 1994]. These agents and the simulated world are less abstract than formerly described models. Polyworld agents operate in a continuous, physically simulated world. They interact with their environment through sensors and actuators. An artificial neural system controls the agents. The specification and details of the system are described in [Yaeger, 1994].

The Environment

The Polyworld software provides a real time, three-dimensional visualization of the world. The simulation is, however, two-dimensional. Agents are polygonal entities. Physically, they are characterized by a set of parameters: size, strength, maximum speed and color. They perceive the world through a simple vision system. The vision perception consists of a one-dimensional strip of colored pixels rendered from the point of view of the agent. A set of actions are available for agents to perform: eating, mating, fighting, moving, turning, focusing and lighting. All these action except eating have and energy cost. Energy

may be gathered by eating from food patches provided by the environment or by eating other agents. Agents may assume a vegetarian behaviour or a carnivorous/predatory behavior. Mating produces offspring when two agents fire the action at the same time while overlapping. Fighting causes energy loss on another agents and also only has effect while physically overlapping the target. Moving and turning allow for spatial navigation in the world. Focusing alters the vision range. Lighting alters the brightness of the agent's color and can be used as a basic form of communication with other agents.

Energy stored in agents is divided by two containers: "health" and "food". Eating stores energy in both containers and all energy costs are payed from both containers. Fighting, however, only removes energy from the "health" container. When the "health" value drops to zero the agent dies. Its body is then converted to food, with the energy value of the "food" container. This model intends to allow for predator-prey behavior, allowing a predator to kill its prey and still obtain an energy reward from it.

The environment imposes several trade-offs between physical characteristics. The maximum amount of energy an agent can store, for example, is determined by its size. The size also affects the energy cost of movement and the result of a fight with other agents. In the same way, relations are also established between strength and energy costs as well as strength and fighting. These trade-offs are defined by linear equations. They are established to allow for the evolution of agents in different ecological niches. Different strategies can evolve, promoting, for example, energy conservation or predatory advantage.

Apart from the cost of actions, there is also an energy cost that the agent pays for maintaining its structure. This cost is a high level simulation of biological organisms base metabolism. In the case of Polyworld, base metabolic energy consumed per cycle depends on factors such as size and brain complexity. Brain complexity for the purpose of calculating metabolic energy cost is determined by comparing the number of neurons and synapses of the agent to the maximum number of neurons and synapses present in the world at the moment. This strategy attempts to force increases of complexity in the neural network to be relate to an increased quality in the agent's behavior.

Physical parameters
ID
Size
Strength
Maximum speed
Life span
Reproduction
Mutation rate
Number of crossover points
Fraction of energy to offspring
Brain structure
Number of visual neurons - red component
Number of visual neurons - green component
Number of visual neurons - blue component
Number of internal neuronal layers
Number of excitatory neurons per internal layers
Number of inhibitory neurons per internal layers
Connection density for all pairs of neuronal layers and neuron types
Topological distortion for all pairs of neuronal layers and neuron types
Learning
Initial bias of neurons per internal group
Bias learning rate per internal group
Learning rate for all pairs of neuronal groups and neuron types

Table 3.2: Polyworld Agent genome.

Evolution

Table 3.2 shows the list of genes that define an Agent in Polyworld. Each gene is an 8-bit value. Except for the "ID" gene, all others define a value inside a pre-defined range. This range is different for each gene and its limits are global parameters of the simulation.

Notice that the genes are organized in groups that define very different aspects of the agent, from physical parameters to the neural networks topologies and learning rates. The Polyworld agent model is heterogeneous in its nature.

In the terminology of the Polyworld author [Yaeger, 1994], agents can be *born* or *created*. *Born* agents are the result of a successful mating, their genome resulting from the crossover of the genome of their progenitors, with possible mutations. The parameters of these genetic operations are determined by the genetic code of the progenitors, as shown in

table 3.2. *Created* agents are placed in the world by the system, either to create the initial population or to prevent the total population from falling beneath a certain pre-defined threshold. In the early stages of a simulation, agents have not developed the ability to mate, so an initial evolutionary process is needed to allow that ability to emerge. This initial evolutionary process is a *steady state genetic algorithm* that uses a pre-defined fitness function that rewards agents for obtaining energy, conserving energy, reproducing, having higher lifespans and moving. Again in the terminology of the author, a simulation is said to have attained a *successful behavior strategy* when it no longer requires the creation of agents through this algorithm to maintain the population, as agents mate successfully in a high enough rate.

Brains

Agents in PolyWorld are controlled by Artificial Neural Networks. The general principle used is to evolve the topology of these networks and let them adapt to the environment during the agent's lifetime by way of an unsupervised learning algorithm. This is a simplified model of animal nervous system development.

The networks consist of layers of input, output and internal artificial neurons. These neurons may be excitatory or inhibitory. Connections originating in excitatory neurons have positive weights while connections originating in inhibitory neurons have negative weights. All non-input neurons also have a bias. Table 3.2 shows the topological parameters that are encoded in genes. These parameters specify the number of neural groups and the quantity of type of neuron by layer. It also specifies the density of connections between layers and the topological distortion of these connections. Topological distortion is a value between 0 and 1, where 0 means that neurons will connect to adjacent neurons in the contiguous layers while 1 means that neurons will connect randomly to neurons in the contiguous layers.

In order to separate topological evolution from learning and avoid Lamarckism, there is no connection weight information stored in the genome. Instead, learning parameters per group are stored: learning rate, bias learning rate and initial bias. A Hebbian learning algorithm [Hebb, 1961; Paulsen, 2000] is used, with the following update rule:

$$w_{ij}^{t+1} = w_{ij}^{t} + \alpha_{ckl}(a_i^t - 0.5)(a_j^t - 0.5)$$

w_{ij}^t is the weight of the connection from neuron i to j at the time t, α_{ckl} is the learning rate for connections to type c from layer k to layer l and a_i^t is the output of neuron i at time t. The type of the connection refers to one of the possible combinations of excitatory/inhibitory to excitatory/inhibitory neurons. The initialization of the brain is done by randomly assigning weights to the connections and then applying the learning rule to the brain for a number of steps while providing arbitrary sensory inputs to the network. This process is inspired in a simulation of the visual cortex [Linsker, 1988, 1987].

The topology of agent brains is fully encoded in the genotype and does not change during the agent's lifetime. As described, learning takes place at connection weight level only.

Speciation

PolyWorld uses two basic mechanisms to promote speciation: spatial isolation through physical barriers and reproductive isolation through a miscegenation function. The barriers can partly or completely isolate parts of the world from each other, thus causing the agents to reproduce in separate groups. This is expected to promote diversity, as geographic isolation is believed to be a key factor in biological speciation. The miscegenation functions determines the probability that two organisms produce viable offspring based on genetic similarity. The higher the similarity between the genetic information of two organisms, the higher the likelihood of successful reproduction. This way, groups of agents that evolve in geographical isolation from each other may become incapable of reproducing when coming in contact, thus forming separate species.

In [Yaeger, 1994], the author of PolyWorld identifies several species that are known to emerge from the PolyWorld simulation.

The *frenetic joggers* are very simple agents that just move ahead as fast as possible while constantly firing the "reproduce" and "eat" actions. These agents tend to emerge in low-hostility worlds, with no barriers or miscegenation function and with wrap-around edges.

The *indolent cannibals* move very little, reproduce with each other and also feed from each other's corpses. These agents tend to emerge in simulations where the parents are not required to transfer energy to their offspring.

Edge runners are similar to the *indolent cannibals* and tend to appear when a physical limit is placed on the edges of the world. They keep moving close to the edges, thus reducing their freedom of movement to one dimension. They adjust their speed when near food or a potential mate.

By making the edges deadly to cross, a more complex species, the *dervishes* tends to emerge. These agents make use of the turn action to explore the world while avoiding the edges. Populations of *dervishes* where observed to regulate their predation and cooperation levels for the optimization of the exploration of common resources.

It is interesting to observe that a higher level of behavioral complexity in agents is obtained by introducing restrictions in the world.

Many different Artificial Life simulations have in common the goal of simulating an increase in complexity driven by evolutionary processes. A proposal of several complexity metrics and the application of these metrics to a PolyWorld simulation is presented in [Yaeger and Sporns, 2006]. The metrics proposed are all related to the artificial neural networks that control the agents. They belong to two categories: network and information theoretic. In this study, a slight increase in complexity was found to exist in the operation of neural networks throughout several simulation runs.

3.3.2 NERO - Neuroevolution in a video game

The creators of NEAT decided to apply their neuroevolutionary algorithm to a multi-agent video game, NERO (Neuroevolutionary robotic operatives [Stanley *et al.*, 2005]). In NERO, the player trains a team of units to confront other teams in a physically simulated world. Training is performed by creating training scenarios and defining a fitness function. The fitness function is defined by adjusting the weights of several criteria, by using sliders in a graphical interface. Units are controlled by evolving neural networks.

NEAT, as most neuroevolutionary algorithms, was designed to operate under traditional generational evolution. In this kind of evolution, the entire population is removed

and a new one generated as generations succeed. This kind of mechanism would be too disruptive to be used in a continuous multi-agent simulation. The creators of NEAT devised a variation of the algorithm to address this type of scenario, called real-time NEAT (rtNEAT) [Stanley *et al.*, 2006].

In rtNEAT, the following sequence of operations is performed every n clock ticks:

- Remove the worst agent
- Re-calculate average fitnesses
- Choose the parent species
- Adjust compatibility threshold (δ_t) and reassign all agents to species
- Place new agent in the world

In the first step, the agent with the lowest adjusted fitness is selected from the group of agents that are past a certain age and removed. This age limit is introduced so that all agents have a chance to perform before competing. Average species fitnesses are used to choose the next parent species, so the average fitness for the species of the removed agent and the global average fitness must now be recalculated. The next parent species is randomly selected according to a probability of selection for each species k given by:

$$P_k = \frac{\overline{F_k}}{\overline{F}},$$

where $\overline{F_k}$ is the average fitness of species k and $\overline{F}$ is the global average fitness.

The creators of rtNEAT decided it was preferable to keep the number of species constant in this version of the algorithm. This is done by dynamically adjusting the compatibility threshold (δ_t). In the fourth step of the sequence, all agents are reassigned to species according to the new threshold. Finally, two progenitors are selected from the parent species according to conventional genetic algorithm techniques and the offspring is placed in the world.

The neural networks evolved in NERO have fixed input and output layer, while hidden layers are topologically evolved. Inputs nodes are the following: enemy radars, *on target*

sensor, object rangefinders and line-of-fire sensor. Enemy radars divide the 360^o around the agent in slices. An enemy in a slice activates the corresponding sensor in a level proportional to its closeness to the agent. The *on target* sensor is activated if a ray projected from the front of the agent intersects an enemy. For object rangefinders, rays are projected around the agent at angle intervals. The activation level of these sensors is proportional to the length of the ray before intersecting an object. Line-of-fire sensors give the orientation of the nearest enemy. They can be used to detect if the agent is currently intercepting the enemy's line-of-fire. Three output nodes are used: left/right movement, front/back movement and fire.

During training, the human player can define scenarios by placing enemy units and obstacles in the playing field. The type of behaviors to evolve is determined by adjusting the components of the fitness function. There are fitness components available for basic behavioral aspects like "approaching enemy", "avoiding enemy" or "hitting target".

3.4 CGP Computational Networks and Biological Plausibility

Most of the existing approaches for artificial brain evolution use a shortcut that goes against biological plausibility, in that they directly evolve the brain. In nature, genetic information does not directly encode the complete structure nor the contents of animal brains. Instead, it encodes adaptive and learning mechanisms, that allows the brains to develop according to the demands of the environment. Animal brains are not static, final systems, but they adapt dynamically during the lifetime of organisms. In the brain model we will present in the following chapter, we too take the shortcut of directly evolving the final structure of the brain. The focus of this thesis is on the creation of bio-inspired algorithms for computational intelligence purposes, but not on strict biological plausibility. This is not to say that the idea of evolving learning brains is not promising for computational intelligence purposes, but it falls outside the scope of this work.

Of the approaches already discussed, Polyworld and New Ties allow for learning during the lifetimes of agents. Neither one, however, encodes the learning mechanism itself in

agent genotypes. The learning mechanism is, thus, predetermined in the simulation.

One very recent research path worthy of mention is that of *CGP Computational Networks (CGPCN)* [Khan *et al.*, 2001, 2008b,a, 2007]. In this model, brains consist of neural networks, with neurons placed in two-dimensional grids. Neurons receive information through dendrite branches, pass it through the soma and output information through axon branches. The network is adaptive, in that, during its lifetime, neurons and branches may replicate or die, and branches can grow or shrink, thus moving from one grid point to another. In CGPCN, neuron behavior is determined by seven ECGP programs. The genotype of brains represents these programs, and does not directly encode brain structure or content. Instead, it encodes the mechanisms by which the brain grows and adapts and the neurons process information.

The dendrite program, D, receives dendrite and soma potentials and updates these values. The soma program, S, updates the value of the soma potential after dendrite inputs are processed. The neuron fires if the soma potential is higher than a threshold value. If it fires, it becomes inactive for a refractory period of some computation cycles. When a neuron fires, its axo-synaptic potential and neighbouring dendrite potentials are updated by running AS programs in axon branches. The three programs, D, S and AS, thus determine how neurons process information. The topology of the brain adapts by way of mechanisms defined by three other programs, DBL, SL and ASL. These are, respectively, the dendrite branch, soma and axo-synaptic branch life cycle programs. They are run in computation cycles where their respective part of the neuron is active. The branch life cycle programs, DBL and ASL, update the *resistance* and *health* values of their respective branches. Resistance is used to determine if a branch will grow, shrink or stay the same. Health is used to determine if a branch will die or replicate. The soma life cycle program, SL, updates *health* and *weight* soma values. Health is used to decide if the neuron should replicate or die, while weight scales the soma output. The final program, WP, is the wight processing program. After all other processing is done on a neuron, it updates the weights of neighbouring branches.

Experiments were performed with evolving brains that learn how to play checkers and operate in the Wumpus World [Russel and Norvig, 2002], with interesting results. In

	Environment	**Building Blocks**	**Reproduction Triggering**	**Explicit Fitness**
ECHO	abstract	a,b,c,d symbols	system	no
New Ties	2D grid	decision trees	agent	no
Tierra	virtual machine	machine code instructions	agent	no
Avida	virtual machine + 2D grid	machine code instructions	agent	arguable, merit system
Polyworld	3D continuous	artificial neurons	system/agent	early stages
NERO	3D continuous	artificial neurons	system	yes
CGPCN	2D grid	CGP neurons	system	yes

Table 3.3: Comparison of evolutionary multi-agent simulations.

Wumpus, for example, brains sometimes displayed a good capacity to adapt to environments different for that which they were evolved on. Also, they appear to have developed a form of instinctive behavior, in that they always started to look for gold in the place where it was in the environment they were evolved on, despite the fact that the genotype does not explicitly contain brain information [Khan *et al.*, 2008b].

3.5 Summary

In table 3.3 we provide a comparison of some of the main characteristics of the approaches discussed. It is possible to observe that there is considerable diversity in the approaches. Research in evolutionary multi-agent systems has been mainly driven by experimentation with simulations. The experimentalist mind-set is reflected in this diversity.

Chapter 4

The Gridbrain

The system that controls the individual behaviour of agents is a central aspect of a multi-agent simulation. In an analogy to animal lifeforms in nature, we will refer to this system as the agent's *brain*. The field of Artificial Intelligence has generated many approaches to the design of agent brains. As seen in previous chapters, even in the more specific case of evolutionary multi-agent simulations, different systems are used. In this chapter we present such a system, the *gridbrain* [Menezes and Costa, 2007b, 2008b,a], that we conceived with the following goals in mind:

- Have the ability to perform generic computations;
- Take advantage of the digital computer architecture;
- Do not require heavy computations;
- Be capable of processing variable-sized information from several sensory channels;
- Use a representation that lends itself to iterative improvement, through evolutionary processes;
- Have adaptable complexity while preventing bloat;
- Produce human-understandable designs.

Since we are aiming to evolve brains in computer simulations, it makes sense to take advantage of the underlying architecture. Evolution in biological systems resulted in the

emergence of highly parallel and asynchronous brains. The digital computer, however, is a sequential, synchronized device. The approach we follow is to abstract mechanisms found in nature and apply them to the medium we work on. While we use algorithms based on Darwinian evolution and connectionism, we use building blocks that are closer to machine code instructions than neurons. This way we hope to explore the medium as an advantage instead of a limitation.

The computational load needed to maintain each agent's brain system directly affects the speed at which the simulation can be run and the amount of agents that we can simulate at the same time. When creating simulations that are meant to be observed or participated by humans, real-time requirements are important. The amount of agents supported is not just of quantitative importance. There is a well known informal saying in complexity sciences that "more is different". The notion of "heavy computation" is subjective. We guide ourselves by the capabilities of contemporary conventional computers.

Autonomous agents make decisions based on sensory information received from the environment. In multi-agent systems, the environment is populated by other agents and entities. Following the ideas of complexity sciences, global phenomena emerge from local interactions. We thus believe it is best to model the environment in entity-to-entity relations. The agent will thus receive sensory information from entities that are within its perception range. The quantity and type of these entities is variable, which poses a difficulty in brain modelling. In simpler simulations, the problem does not arise. One common case is that of worlds modeled as two-dimensional grids, where each cell can only contain one object. At a certain moment, an agent only perceives the cell it is facing, or a fixed number of neighbouring cells.

When moving to continuous environments, this problem cannot be avoided as easily. One common approach is to perform pre-processing on the sensory information, so that a fixed set of symbols is provided to the brain. Let us consider and environment populated with objects, where some are food. Sensory information like *food_visible* or *direction_of_nearest_food* could be generated. The problem with this approach is that the preprocessing conditions and limits the range of possible behaviors developed by the agent. In neural-network based agents, another approach is to use radars. Radars are a sequence

of input neurons, equally distributed across the view range of the agent, that fire according to angle or distance from a certain type of object. This technique still causes the filtering of information that could be used by the agent. Consider the case where two object types are being scanned by two radars, and both objects can have the same properties. If both radars fire on the same position, there is no way to know if two objects are being perceived, each with one of the properties, or only one with the two properties, or even more objects with combinations of these properties. A radar could be provided for property conjunctions, but then we fall again into a form of preprocessing.

One could argue that, as we move towards more realistic, physically simulated worlds, we can just provide the agent with visual renders or sound waves from its point of view, and let it extract all the information it needs from there. There are obvious problems with this approach. Image recognition is a very hard and computationally demanding problem. Under the current state of the art, it would not be feasible to run real-time multi-agent simulations using such a system. Also, many times there is no need or purpose in modelling the world with that level or realism. Even not considering these issues, a system that decomposes sensory information into individual entities would still output variable-sized, entity based information. In this work we focus on the problem of evolving brains that are capable of processing this later type of data.

Evolutionary processes in Nature depend on the accumulation of small changes that improve the survival probability and reproductive success of organisms. As the organisms develop more sophisticated mechanisms, their structure becomes increasingly complex. In the pursuit of evolutionary computational intelligence, it is important to define brain representations that allow for small step improvements and adaptable complexity. Sophisticated solutions should be able to evolve from initial, simple instances. In the context of the direct use of evolutionary processes to generate brains, as discussed in section 3.4, this calls for brain representations that can adapt their size and structure during evolution, without imposing arbitrary limits. One challenge in working with such systems is that of avoiding bloat, which is a useless increase in size. Bloat can compromise several goals we are aiming at. It leads to a waste of computational resources, both in processor time and memory. It can also lead to stagnation in evolution, by causing a type of unwanted

redundancy in the system that prevents simple mutations from causing relevant changes. Bloated systems also tend to be more difficult to be interpreted by humans.

Easy interpretation of the evolved systems is an interesting feature. It allows us to inspect the outcomes of the evolutionary process and access its limitations while giving us insight for improvements.

As far as we know, none of the systems in use nowadays meets all these requirements to satisfaction. Limitations in the sensory system have already been addressed. As detailed in previous chapters, two popular approaches for evolutionary agent brains are neural networks and rule systems. Neural networks are of difficult interpretation and tend to be treated as black boxes. Also, neural networks do not take full advantage of the inherent capabilities of the digital computer, using neurons as the building blocks. Rule systems, on the other hand, produce more readable results and are closer to high level computing languages, but tend to be limited in their capabilities. Most rule systems are sequences of *if/then* rules, lacking a wider range of constructs that are possible in von Neumann type machines.

We propose a system that belongs to the family of genetic programming, and that attempts to address all the stated requirements.

4.1 Computational Model

The gridbrain is a virtual machine inspired in the von Neumann architecture. It was designed to serve as a brain for an autonomous agent. As can be seen in figure 4.1, it consists of a network of computational components placed on rectangular grids. There are two types of grids: alpha and beta. Alpha grids are associated with sensory channels and are responsible for processing perceptual information. The beta grid receives inputs from the alpha grids and outputs decisions. A gridbrain can have any number of alpha grids (one of each sensory channel), but only one beta grid. This architecture is inspired on the organization of animal brains, which have areas for processing sensory information, and others for decision making, planning and muscle control. It is not intended to be an accurate model of such brains, but only an application of the concept of dedicated layers

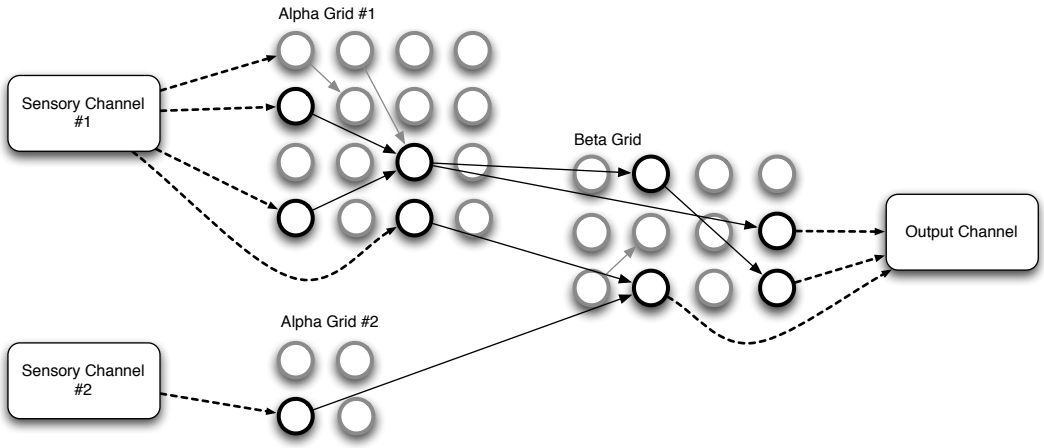

Figure 4.1: Gridbrain computational model.

in a network of components. Furthermore, it is not intended to be a reductionist model with rigid hierarchies. The only limitation imposed is that sensory information is fed to the alpha grids and output is provided by the beta grid. As we will detail later, this allows the gridbrain to deal with variable sized perceptual information, while the evolutionary process remains free to test a diversity of structures.

Connections between components represent flows of information. A connection from component A to component B means that, during each computation cycle, the output value of component A is fed as an input to component B. The information propagated takes the form of unbounded floating point values. Two types of connections are possible: feed-forward connections inside a grid and connections from any component in an alpha grid to any component in a beta grid. In neural networks, recurrent connections allow the system to keep information about the past. Feed-forward neural networks model purely reactive agents. In the gridbrain, to overcome this limitation, we provide the system with explicit memory mechanisms. Components may be able to conserve their state across computation cycles. We thus chose to only allow feed-forward connections in the models studied in this work for simplicity.

In each gridbrain cycle, the outputs of components are computed in order. We define a coordinate system where each component position is identified by a tuple (x, y, g), x being the column number, y the row number and g the grid number. Components are processed

from the first to the last column, and inside each column from the first to the last row. The feed-forward restriction is imposed by only allowing connections inside the same grid to target components with an higher column number than the origin. The rectangular shape of the grids facilitates the definition of both parallel and sequential computational processes.

Both components and connections may be either active or inactive. In figure 4.1, active components and connections are represented in black, inactive ones are grey. Active status is not explicitly encoded in the gridbrain, but derived from the network configuration. To explain how this works, we must first introduce the concept of *producer* and *consumer* components. *Producer components* are the ones that introduce information into the gridbrain, while *consumers components* are the ones that send information from the gridbrain to the outside. *Input components* are the alpha grid components associated with sensory information. They are updated with current sensory data in the beginning of each alpha grid computation cycle. Input components are producers. Other components that output values different than 0 without any input present are also considered producers, because they can generate a signal without a stimulus. Examples of this are a random value component, that outputs a random value, or a component that outputs 1 if the sum of its inputs is 0. *Output components* are the beta grid components from which decision information is extracted after a gridbrain computation cycle. In an agent environment, they are associated with triggering actions. Output components are consumers. An active path is a sequence of connections that links a producer component to a consumer component. A connection is considered active if it belongs to an active path. A component is considered active if it has at least one ingoing or outgoing active connection.

Only active components and connections influence the computation performed by the gridbrain. From a genetic perspective, we can say that active elements are the ones that have phenotypical expression, while inactive elements are not expressed in the phenotype. As we will show later, inactive elements are valuable to the evolutionary process. This has been shown to be true in CGP, a variant of genetic programming that has some characteristics in common with the gridbrain model [Vassilev and Miller, 2000; Yu and Miller, 2001; Miller and Smith, 2006; Yu and Miller, 2006]. In both approaches building

```
for each grid g in grid_set:
    seq = component_sequence of grid g
    for x = 0 to width of g:        // columns
        for y = 0 to height of g:   // rows
            comp = component of g at position (x, y, g)
            if comp is active:
                add comp to end of seq
```

Listing 4.1: Gridbrain sequentialization.

blocks are placed in grids.

After a gridbrain is generated, it is sequentialized. The sequentialization algorithm is presented in listing 4.1. This process consists of creating, for each grid, a list of components in the order that they should be computed in a gridbrain cycle. This order is determined by the positions in the grid. Inactive components are ignored. This sequentialization step is performed for performance reasons. As gridbrain computation is a very frequent event, relatively to gridbrain generation, it is useful to transform the more complex representation into a format that discards inactive components and connections, which are irrelevant to the computation, and provides a list of components that can be iterated through at a low computational cost.

In a computation cycle, the sequences are executed for each grid. There are two evaluation stages, alpha and beta. In the first stage, alpha grids are evaluated, once for each entity in the sensory channel they are associated with. In the second stage, the beta grid is evaluated once. Alpha grid evaluation consists of two passes. This is done so that certain alpha grid components, which we call *aggregators*, have the chance of calculating their output based on information about the entire set of entities. We will discuss aggregators in more detail in later sections. An example of this is a maximizer component that outputs 1 if the current value inputed is the maximum for the set of entities present, 0 otherwise. For this to be possible, a first pass is performed on the grid where only intra-grid connections are active. In this phase, aggregators can compute their internal state so that they produce the correct outputs on the second pass. On the second pass, inter-grid connections are also active so that information can be propagated to the beta grid. In the example of the maximizer, the first pass is used to determine the

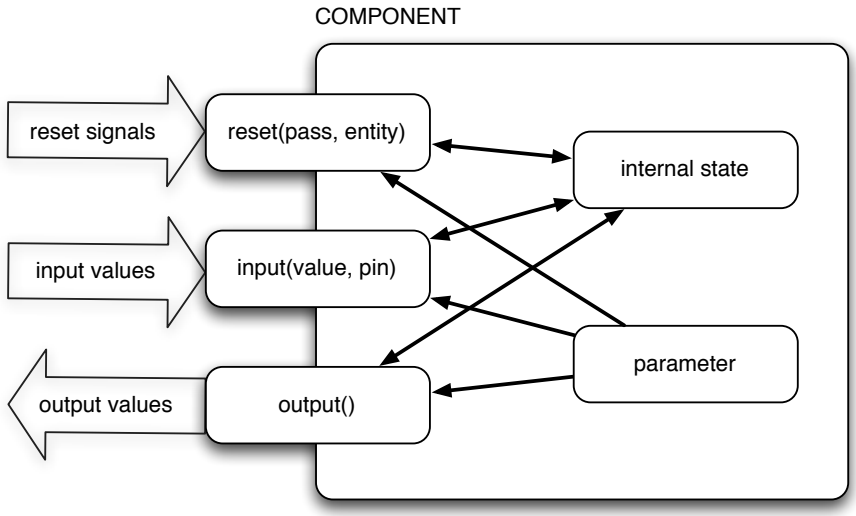

Figure 4.2: Component model.

maximum value, while the second is used to signal this value when it is found.

Data aggregation could be done in other ways. For example, we could just provide the gridbrain with special input components that preprocessed their input vector by calculating averages, maximums or other global views on the input data. This, however, would invalidate the possibility of more general mechanisms. Notice that our aggregators do not have to receive data directly from input components. They can receive data that is already processed by other components. For example, we can have an aggregator returning the average of the sum of two input components.

4.1.1 Component Model

Components are information processing units. They are the computational building blocks of gridbrains. The generic gridbrain model does not define a specific set of components. Much like machine code instructions, components belong to classes of functionalities: input/output, arithmetic, boolean logic, information aggregation, synchronization and memory. Component sets may be conceived for different environments and problem domains. However, we will propose a set of components to be used in physically simulated worlds. These were applied in the experimentation presented in later chapters.

Components have an arbitrary number of inputs and outputs, to allow for a high degree

Component type	State persistence
Operator	Grid evaluation
Aggregator	Alpha stage
Memory	Gridbrain lifespan

Table 4.1: Component classification according to state persistence.

of freedom in network topologies. This contrasts with conventional genetic programming systems, where functions have fixed numbers of parameters that must be respected for the resulting programs to be valid. As discussed in section 2.6.3, in ECGP an arbitrary number of inputs and outputs is possible at the module level, as modules can be generated with any number of terminals, and these can later be changed by way of mutation operators. In the gridbrain however, this freedom is present at the fundamental building block level. We strive for a connectionist model closer to natural brains. To achieve this, we developed the component model presented in figure 4.2.

The gridbrain design follows an object oriented approach, where a base abstract component class defines a set of interfaces for interaction. Specific components are defined by inheriting from this base class and defining the internal mechanisms. This way we can have gridbrains made up of an heterogeneous mix of component, and yet allow the evolutionary process and computation cycle algorithm to treat these components as black boxes with a fixed set of external interfaces.

A component has three interfaces: *reset*, *input* and *output*. The reset interface is used to signal the component that a new grid evaluation has started. The input interface is used to feed a value from downstream components, and the output interface is used to produce a value to be fed to upstream components.

Components have internal states. The data structure that maintains the state is defined by each specific component. We classify components as *operators*, *aggregators* or *memories* according to the persistence of the state, as shows in table 4.1.

Operators are information processing components that perform, for example, arithmetic or boolean operations. Their output is determined only by information present in the current grid pass. Aggregators extract general information from a set of entities present in the sensory channel of an alpha grid, and their output is determined by all the

values received during an alpha stage. They could, for example, provide minimum, maximum or average values for that stage. Memories conserve their state across computation cycles and provide the gridbrain with information about the past.

Components also have a parameter value, which is a floating point number in $[0, 1]$. Components may use this value to adjust their behavior. For example, an amplifier component can use the parameter to determine its amplification factor or a clock component may use the parameter to determine its ticking frequency.

4.1.2 Computation Cycle

A gridbrain computation cycle is performed each time we want to feed current sensory information and obtain a decision. The pseudo code that describes how the cycle algorithm is presented in listing 4.2.

In the code shown it can be seen how the interface functions of the components are called during the cycle. Notice that input components have a special *input_type* field that is used to identify the type of information from the sensory channel they are associated with. Likewise, output components have an *output_type* field that identifies the type of output information they are associated with. In a typical use of the gridbrain as an autonomous agent controller, each output type is associated with an action that the agent can perform.

4.2 A Component Set

In this section we present a component set to be used with the gridbrain. As stated before, the gridbrain model does not specify a component set, but only an abstract component model. Components may be tailored to specific applications and environments. We present this set to better illustrate the gridbrain model and because it will be used in the experimentation described in later chapters. The components presented result from experimentation and a trial and error process, where we attempted to evolve gridbrains in specific environments. We did however try to create general purpose components, and expect the ones presented to have broad application.

```
// Alpha stage
for each alpha grid g in grid_set:
    for pass in [0, 1]:
        for each entity in the sensory channel of g:
            seq = component_sequence of grid g
            for comp in seq:
                comp.reset(pass, entity)
                if comp is input:
                    input_type = comp.input_type
                    value = input_type value of entity
                    comp.input(value, 0)
                output = comp.output()
                for each connection conn from comp:
                    if pass == 1 or conn is intra grid:
                        targ_comp = target component of conn
                        targ_comp.input(output, comp.id)

// Beta stage
g = beta grid
seq = component_sequence of grid g
for comp in seq:
    comp.reset(0, 0)
    output = comp.output()
    if comp is output:
        gridbrain_output_vector[comp.output_type] = output
    for each connection conn from comp:
        targ_comp = target component of conn
        targ_comp.input(output, comp.id)
```

Listing 4.2: Gridbrain computation cycle.

Name	Description	Type	Consumer/Producer
IN	Input	Input/Output	Producer
OUT	Output	Input/Output	Consumer
AND	Boolean AND	Operator	
NOT	Boolean NOT	Operator	Producer
SUM	Sum	Operator	
MUL	Multiply	Operator	
INV	Inverse	Operator	
NEG	Negative	Operator	
MOD	Module	Operator	
AMP	Amplify	Operator	
RAND	Random value	Operator	Producer
EQ	Is equal	Operator	
GTZ	Is greater than zero	Operator	
ZERO	Is zero	Operator	Producer
MAX	Maximum	Aggregator	
MIN	Minimum	Aggregator	
AVG	Average	Aggregator	
MEM	Memory cell	Memory	
SEL	Select entity	Memory	
DMUL	Delayed multiplier	Memory	
CLK	Clock	Memory	Producer
TMEM	Temporary memory	Memory	

Table 4.2: A Component Set.

In table 4.2 we list the components we will describe.

It is important to notice that we present these components in groups for clearness of presentation, but they are defined as interoperable building blocks. The evolutionary process is free to combine them without restrictions.

4.2.1 Input/Output

Input and output of values in the gridbrains is done by way of the IN and OUT components. These are the only two components that are mandatory in any set, because they are fundamental to gridbrain operation.

Previously we showed that alpha grids are associated with sensory channel processing and the beta grid with decision making. Alpha grids are where sensory information enters the system and the beta grid is where decision values are output. This way, IN components are only to be placed in alpha grid sets and OUT components in beta grid sets.

Both IN and OUT contain a floating point value state. In both components, the reset interface changes this state to 0 and the output interface returns the current state value. The input interface of IN writes a value received to the internal state, while the input interface of OUT writes a value received that is different from zero to the internal state.

We arrived at this mechanism for the input interface of OUT after experimenting with alternatives. The two other approaches we tried were:

- Overwrite the internal state of the component for every input value it receives;
- Sum every input value to the internal state value.

The problem with the first approach is that all but the last connection path to provide its input to the OUT component will be ignored. This, in practice, excludes the possibility of having several connection paths contributing to the same OUT component.

The problem with the second approach is that it leads to initial solutions that are not easy to complexify and further evolve. In the type of scenarios we experimented with, and which we will describe in following chapters, gridbrains were used to control agents in physically simulated environments. OUT components were related to the triggering of actions with a certain intensity. One example is the action of thrusting the agent forward. The intensity of the thrust is determined by the output value of the OUT component associated with the action. Suppose that it is advantageous to increase the intensity with which the action is performed. One way is to make the connection path that leads to the OUT component go through a component that scales its value up. However, if input values arriving at the OUT component are summed to its internal state, it is much more likely that evolution finds another way to increase the intensity: adding more connection paths that arrive at the OUT component. This is especially likely to happen in the initial stages of evolution, when the first very simple beneficial behaviors are found. The reinforcement of these behaviors by way of multiple connection paths leads to a kind of structure that is hard to further evolve, because simultaneous mutation operators would be needed to produce meaningful change.[1] Simultaneous good mutations are very unlikely.

[1] In later sections we will discuss how mutation operators change the topology of the network. It will then become apparent why several simultaneous mutations would be needed to change the functionality of the network in this case.

The approach of updating the state on non-zero input values allows several connection paths to contribute to the same decision while preventing trivial behavior reinforcement that leads to stagnation. On one hand, reinforcement does not work because input values are not summed. On the other hand, multiple paths may contribute to the behavior because any path can activate the OUT component while the other paths are deactivated. In practice this is the rule we observed to work better.

Input and output components have a type field, in the form of an integer value. This is used by the environment where the gridbrain is contained to determine which kind of sensory information or output information the component is associated with. Supposing that the gridbrain is the controller for an autonomous agent, we could have an IN type that specifies the sensory information as the distance to an object and another one that specifies the sensory information as the brightness of an object. When the sensory information is fed from the agent perception systems to the gridbrain, the type of the IN components dictates which specific sensory value it should receive. In the same way, an OUT type could be associated with the action of moving forward and another one with the action of turning on a light. After the gridbrain performs a computation cycle, the agent uses the types of the OUT components to interpret their values and perform the appropriate actions.

4.2.2 Boolean Logic

The two boolean logic components are NOT and AND. These components treat their input values as boolean values. They define a threshold value t to determine if the input is considered on or off. If $|input| \leq t$ it is considered off, otherwise on. NOT outputs a value of 1 if none of its inputs is on. AND outputs a 1 or -1 if all of its inputs are on. If it triggers, the signal of the output of AND is equal to the signal of the product of all its inputs.

Both these operators have a floating point value internal state. The NOT component reset interface sets the internal value to 1. The input interface checks if the module of the input value is greater then the threshold, in which case it changes the internal value to 0. The output interface returns the internal state. This way, the NOT component returns 1

```
reset(pass, entity):
    first_input = 0
    fp_state = 0

input(value, pin):
    if first_input:
        first_input = false
        if abs(value) > t:          // abs(x) gives the modulus
                                    //of x
            fp_state = sig(value)   // sig(x) gives the signal
                                    //of x (-1 or 1)
        else:
            fp_state = 0
    else:
        if abs(value) > t:
            fp_state = fp_state * sig(value)
        else:
            fp_state = 0

output():
    return fp_state
```

Listing 4.3: Pseudo-code for the AND component.

if none of its inputs is on.

The AND component has a boolean flag in its internal state, additionally to the floating point value, *fp_state*. This flag signals if the current input is the first one in the grid evaluation. The flag is necessary because the first input is a special case. An initial input signal above the threshold activates the component despite the fact that its internal state is 0, unlike the following signals, which always fail to activate it if the internal state is 0. The pseudo-code for the AND component is presented in figure 4.3.

The NOT component is considered a producer because it outputs a value different from 0 when not receiving any input. This makes it capable of activating outputs in the absence of sensory information.

Arithmetic components can also be used as logic gates, albeit without the threshold. In fact, we decided to not include an OR component, as several of those components can be used as such. The AND component with a single input connection can be seen as a converter from continuous to boolean values.

```
reset(pass, entity):
    first_input = false
    equals = false
    fp_state = 0

input(value, pin):
    if first_input:
        first_input = false
        equals = true
        fp_state = value
    else:
        if value != fp_state:
            equals = false

output():
    if equals:
        return 1
    else:
        return 0
```

Listing 4.4: Pseudo-code for the EQ component.

4.2.3 Arithmetic

The ten arithmetic components are: SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ and ZERO. These operators have a floating point value internal state, except for RAND. RAND outputs an uniformly distributed floating point value in the $[0, 1]$ interval, regardless of its inputs, and is a source of randomness.

The MUL component computes the product of all its inputs. If only one input exists, its value is output. Like the AND component, it has a first input flag in its internal state and uses a similar mechanism, without the threshold.

The function of the EQ component is to output 1 if all of its inputs are equal, and 0 otherwise. It uses a first input flag in its state, and another boolean flag that keeps the current state of the comparison. We call this flag $equals$. It also uses a floating point value in its state, fp_state. The pseudo code for the EQ component is presented in listing 4.4.

All other components have the same input interface: the input value is summed to the internal state. Their state is always set to 0 by the reset interface, before a grid evaluation.

SUM outputs the value stored in the state, thus providing the summation of its inputs. NEG outputs the negative: $-state$. MOD outputs the modulus: $|state|$. GTZ returns 1

if $state > 0$, 0 otherwise. ZERO returns 1 if $state = 0$, 0 otherwise.

INV truncates the internal state value to $[-1, 1]$ and then outputs $-1 - state$ if $state < 0$, $1 - state$ if $state > 0$. This can be used to invert the meaning of certain sensory data, that we assume to be defined in the [-1, 1] interval. One example is distance to an object, which the INV component transforms into proximity.

AMP operates as an amplifier. It outputs the product of its internal state by an amplification factor, which is determined by its parameter. The amplification factor is given by the expression:

$$a(p) = (1/(1-p)) - 1$$

where a is the amplification factor and p the parameter. This way, for $p < 0.5$, $a < 1$ and the signal will be attenuated. The expression defines a curve where:

$$\lim_{p \to 1} a(p) = +\infty$$

We map a value in the $[0, 1]$ parameter interval to $[0, +\infty]$, resulting in a possible interval of amplification factors which is unbounded at the top.

Arithmetic operators may produce indeterminations, in which case the value is set to 0.

As with the NOT component, ZERO and RAND are considered producers, because they can output a value different from 0 when not receiving any input.

4.2.4 Aggregators

As defined, aggregator components conserve an internal state across the alpha grid evaluations of a gridbrain computation cycle. They serve the purpose of extracting information from the entire set of entities in a sensory channel. Aggregators are of no use in beta grids, so they should only be included in alpha grid component set. The aggregator components we describe here are MAX, MIN and AVG.

Aggregator computations have two phases, related to the two passes in alpha grid evaluations. In the first phase, an internal state is computed from the entire set of entities

in the sensory channel of the containing grid. In the second phase, this internal state is used to produce an output.

Aggregator components keep track of the current alpha pass, 0 or 1, by storing the *pass* value they receive from the reset interface in an internal variable.

The function of the MAX component is to signal the appearance of the maximum value in an input vector. In the first phase, the MAX component finds the maximum value different from 0 produced by its inputs, across the alpha grid evaluations in a gridbrain computation cycle. In the second phase, a value of 1 is output when it is again found. In a gridbrain computation cycle, the MAX component outputs 1 during the alpha grid evaluation linked to the entity that produced this maximum, and 0 in other evaluations. If more than one entity produce a maximum, the 1 value is output only for the first occurrence. The pseudo-code for the MAX component is presented in listing 4.5.

The MIN component operates in a similar fashion to MAX, except that it signals the minimum value found.

The AVG component computes the average of its inputs with values different from 0, across the alpha grid evaluations. Its input interface is the same as MAX and MIN, storing the product of the values from its input connections in an internal input state. It has two more internal variables: *input summation* and *input count*. These two internal variables are reset to 0 in the beginning of a gridbrain computation cycle. In the first computation phase, the output interface checks if the input state is not 0. If it is not, it sums it to the input summation and increments input count by one. In the second computation phase, the output interface always returns the value obtained by dividing the input summation by the input count.

The multiplier input interface is used in these components so that an input connection can easily be used to select values to participate in the aggregation. A simple scenario is a MAX component with two connections, A and B. A is linked to a sensory value while B produces a signal of a boolean nature, either 0 or 1. In this case, MAX will signal the maximum value found for all the cases where B is on.

```
reset(pass, entity):
    input = 0
    first_input = true

    // New gridbrain cycle
    if (pass == 0) and (current_pass == 1):
        current_max = 0
        cycle_flag = false
        triggered = false

    current_pass = pass

input(value, pin):
    if first_input:
        input = value
        first_input = false
    else:
        input = input * value

output():
    output = 0

    if input != 0:
        if not cycle_flag:
            output = 1
            current_max = input
            cycle_flag = true
        else:
            if current_pass == 0:
                if input > current_max:
                    current_max = input
                    output = 1
            else if (input == current_max) and (not triggered):
                triggered = true
                output = 1

    return output
```

Listing 4.5: Pseudo-code for the MAX component.

4.2.5 Memory and Synchronization

The components that are able to conserve a state across the lifespan of the gridbrain are: MEM, SEL, DMUL, CLK and TMEM. These components are used for persistent memory and synchronization.

The MEM component is a simple memory cell. Its persistent internal state is a floating point variable which we will call *memory state*. The memory state is initialized to zero when the component is created. It also has an input state which is reset to 0 in the beginning of all grid evaluations. The input interface sums the input value to the input state. The output state checks if the input state is not 0. If so, the input state is written to the memory state.

The CLK component is a clock device, producing a 1 signal at fixed intervals, otherwise outputing 0. The interval between ticks is the number of simulation cycles which is determined by the component parameter. It is given by the expression $I(p) = p.I_{max}$, where p is the parameter value in $[0, 1]$, and I_{max} is the maximum possible value for the period. I_{max} is a pre-configured value relative to the environment the gridbrain is operating on. In a multi agent simulation, it is a good choice to make I_{max} equal to the maximum lifespan of the agent.

The clock component stores the amount of cycles left until it triggers again in an internal state variable that we will call *time to trigger*. The output interface of the clock checks if time to trigger is 0. If it is, it outputs a 1 value and sets the time to trigger to $I(p)$. If not, it outputs 0 and decrements time to trigger by one.

A clock may be synchronized by way of its input connections. The input interface is equal to the MEM component, keeping a summation of the current input values in an input state variable. It has another internal state variable that keeps the value of the previous input state. We call it *last input state*. A clock is forced to fire and restart its firing interval if its input state changes from 0 to another value. After updating the input state, the input interface checks if the input state is not 0 and the last input state is 0. If so, the time to trigger variable is set to 0. In any case, the input state value is then written to last input state.

The TMEM component is a temporary memory. It combines the functionalities of

the MEM and the CLK components, providing a computational building block with a functionality that would be hard to evolve by using just memories and clocks. It keeps a memory state like the MEM component, and has a periodic triggering mechanism like the CLK component. It always outputs the current value stored in the memory state. When the clock mechanism triggers, the memory state is set to 0. It is a memory cell with the capacity of *forgetting*.

The DMUL component is a delayed multiplier. It waits until all of its input connections have produced a value different from 0, and then outputs the product of the last value different from 0 received from each connection. Its internal state consists of an array of float values, with one value for each input connection the component has. When the component is created, the array is initialized to 0 values. The input interface checks if the input value is not 0. If so, this value is written to the corresponding position in the array. The output interface verifies the values in the array. If all of them are not 0, the output is the product of these values and the array is reset to 0 values. If at least one of the values in the array is 0, the output is 0.

The SEL component is an entity selector for alpha grids. It selects one entity present in the sensory channel of their containing grid, and keeps producing outputs only during the grid evaluation corresponding to this entity, while it is present. When it is no longer present, it selects a new one.

The input interface of selector components is a multiplier, like, for example, the one used in MUL components. Only entities for which the input state is not 0 are considered for selection. Its persistent internal state includes a variable that stores the current entity ID, *current entity*, a variable that stores the selected entity ID, *selected entity*, a variable that stores a select candidate ID, *select candidate* and a flag that informs if the selected entity was found in the current alpha evaluation, called *selected found*. We assume all entities have an ID greater than 0, and that a 0 value means "no entity". When the component is created, *selected entity* is initialized to 0.

In the beginning of the first alpha pass, the *selected found* flag is checked. If it is false, the *select candidate* value is written to the *selected entity* variable. The *selected found* flag is then set to false. In all alpha evaluations, the *current entity* is set to the entity value

received by the reset interface. The output interface checks if *input state* is not 0. If so, it writes the *current entity value* to *select candidate*. It then checks if *select candidate* equals *selected entity*. If so, it sets *selected found* to true and outputs *input state*. In all other cases, it outputs 0.

The selector component allows the gridbrain to keep track of a specific entity, using its inputs as a filtering condition. It only makes sense to use this component in alpha grids.

4.3 Representation

The gridbrain was conceived to be used in evolutionary environments, so it is important to define its representation. It is upon this representation that genetic operators act.

As can be seen in figure 4.3, a gridbrain with N grids and M connections is represented as a sequence of N grid segments and M connection segments. Both these types of segments are data structures composed of several fields.

The grid segment contains the component set for the grid. This set is a list of valid components for the grid, and it is used by genetic operators that need to generate or replace components. It also contains the values for the with (W) and height (H) of the grid. It contains an array of component segments that defines the specific components that constitute this grid. This array is $W.H$ in size. The segment for a component of this grid with coordinates x, y can be found at position $p = x.H + y$ in this array. The grid segment also contains two lists of row/column ID segments. The column ID list is W in size and the row ID list is H in size. These IDs are used by genetic operators to determine row/column equivalence. The component segment has two fields. The first is an integer that identifies the type of the component and the second is a floating point in $[0, 1]$ that gives the parameter value.

The connection segment has integer fields to specify the coordinates of the origin and target of the connection. It also contains three fields for the connection tags. The tag mechanism is used by genetic operators to determine connection equivalence.

In figure 4.4 we exemplify with a possible genotype and a graphical representation of its corresponding gridbrain.

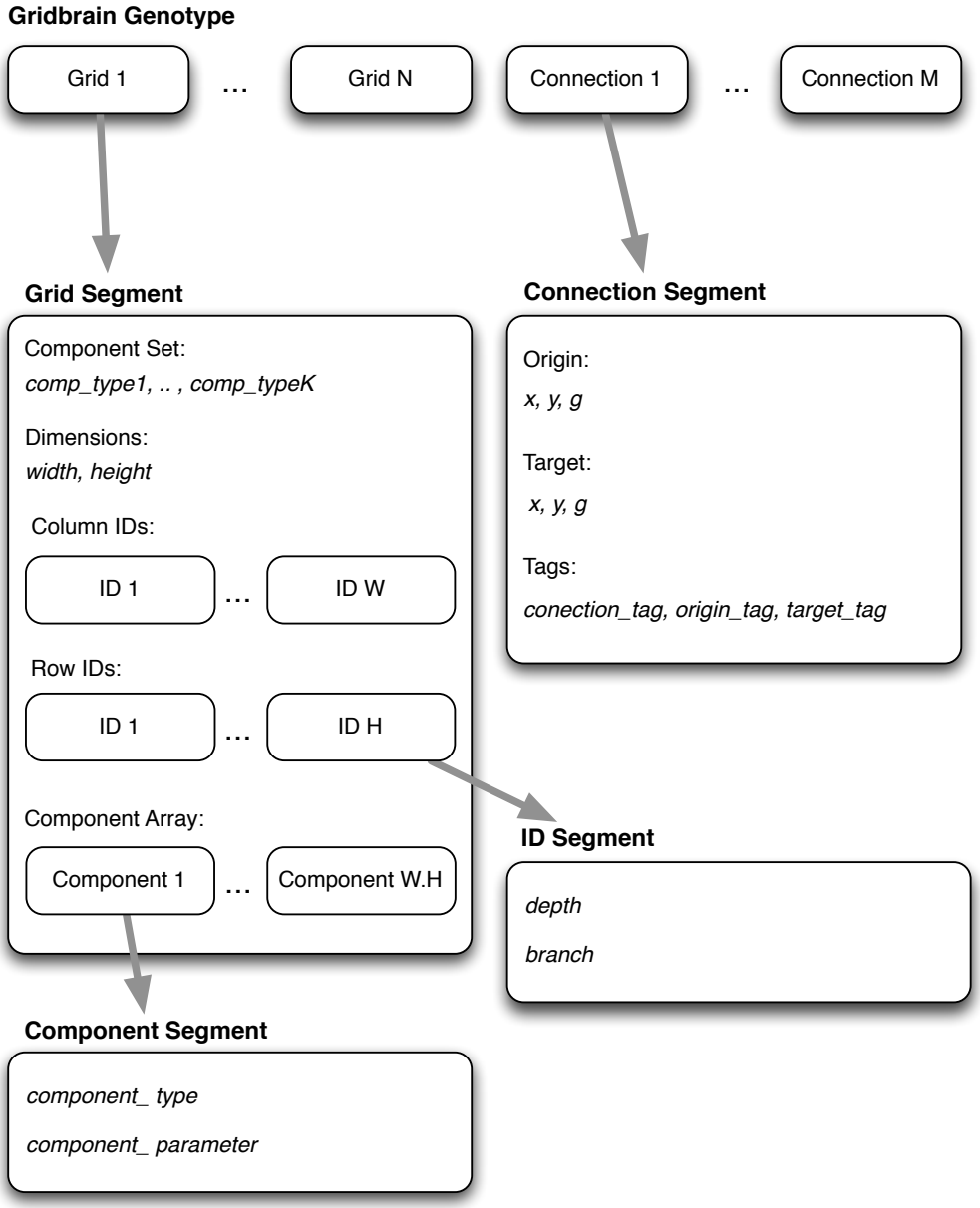

Figure 4.3: Gridbrain genetic representation.

```
(                                                   // Gridbrain
    (                                               // Grid segment
        ('A', 'B', 'C', 'D'),                       // Component Set
        (2, 3),                                     // Dimensions
        ((0, 0), (1, 2)),                           // Column IDs
        ((1, 0), (0, 0), (1, 1)),                   // Row IDs (ID segments)
        ((0, 0.3), (2, 0.8), (1, 0.23), (3, 0.45),
         (1, 1.0), (1, 0.1))                        // Component array (Component segments)
    ),
    (                                               // Grid segment
        ('C', 'D', 'E', 'F'),                       // Component Set
        (2, 2),                                     // Dimensions
        ((1, 0), (2, 2)),                           // Column IDs
        ((0, 0), (1, 1)),                           // Row IDs (ID segments)
        ((3, 1.0), (0, 0.98), (2, 0.5), (2, 0.87)) // Component array (Component segments)
    ),
    ((0, 0, 0), (1, 1, 0), (0, 1, 2)),              // Connection segment (origin, target, tag)
    ((1, 1, 0), (0, 0, 1), (0, 2, 3)),              // Connection segment (origin, target, tag)
    ((1, 2, 0), (0, 1, 1), (4, 5, 6))               // Connection segment (origin, target, tag)
)
```

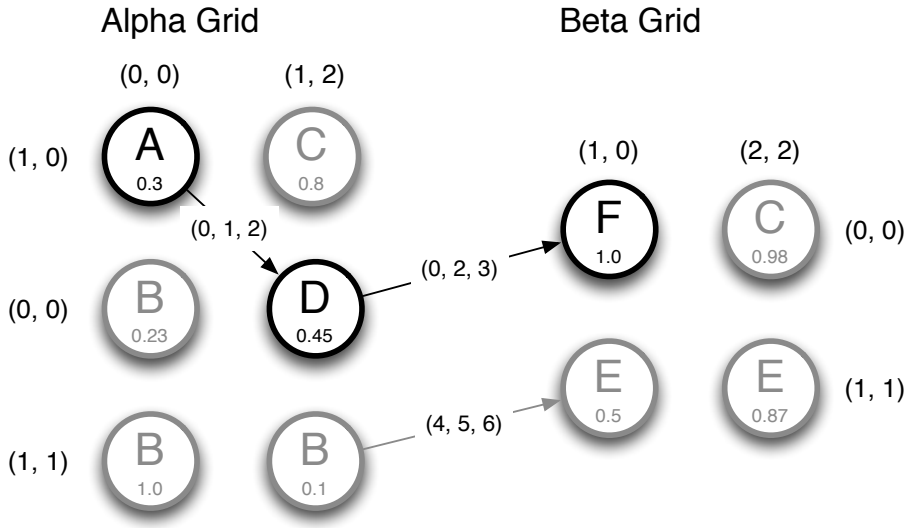

Figure 4.4: Example of a genotype and graphical representation of the corresponding gridbrain.

4.3.1 Row/Column IDs

In the processes of cloning and recombining gridbrains, it is important to know which columns and rows are equivalent. As will be described later, genetic operators can change the shape of a grid, by inserting or deleting rows and columns. This makes it impossible to determine equivalence using the coordinates system. We devised a binary tree mechanism that allows us to identify a column or row based on the insertion/deletion history.

Every column and row in a grid receives an ID, which is a data structure with two integer fields: *depth* and *branch*. These values establish the position of the row or column in a binary tree. In a grid, two separate binary tree structures exist: one for rows and one for columns. We will generically refer to a row or a column represented by a node in the tree as an *element*.

There is a mapping between tree positions and sequential positions of the elements. From a tree, it is possible to derive a unique sequence of elements, forming the sequence of rows or columns of a grid. The first element inserted in a tree is given the ID $(0, 0)$. Following elements are given IDs relative to an ID that already exist. The branches of the tree represent relations of relative position in the sequence between two IDs. Suppose we have a node for element A, with two branches connecting it to B and C. B is in the first branch and C in the second. This means that B comes before A on the sequence, and C comes after A. If the tree just consists of these three nodes, it corresponds to the sequence BAC. Now suppose we want to insert a new element D. Its position on the tree is determined by the position we want to to have on the sequence. We can choose to insert it before of after any of the existing elements in the sequence. This is done by assigning an ID to the element that places it as the first or second branch of an existing node in the tree.

The values of the ID for an element inserted before a given element are determined by the following expressions:

$$depth_{before} = depth + 1$$

$$branch_{before} = 2.branch$$

The values of the ID for an element inserted after a given element are determined by the expressions:

$$depth_{after} = depth + 1$$

$$branch_{after} = 2.branch + 1$$

As can be seen, elements inserted with a position relative to an exiting element are one level deeper in the tree. The branch value is duplicated because of the branching factor of 2 of the binary tree. The maximum number of nodes in tree level n is 2^n.

Genetic operators, which will be described in following sections, require the following insertion operations: insert new element before the first element in the sequence, after the last one or between existing elements. When inserting a new element between existing elements, we examine the ID of these elements. If the depth of the ID of the first existing element is greater or equal than the other one, the new ID is generated as the second branch of the first element. If the depth of the second ID is greater, the new ID is generated as the first branch of the second element. This guarantees that new IDs do not clash with existing IDs in the tree and corresponds to the intended position in the sequence.

In figure 4.5 we present and example of row insertions and deletion. The generated IDs are shown according to their position in the binary tree. The IDs are shown as $(depth, branch)$ tuples. We should point out that, although the binary tree is the underlaying structure for ID generation and comparison, no actual tree structure is explicitly maintained, only ID tuples.

In listing 4.6 we present the pseudo code for an ID comparison function. This algorithms allows us to determine the relative position of two ID tuples in the sequence, even if they are not directly connected by a branch in the tree. The strategy used can by explained as follows: either the two IDs have the same depth or not. The relative position of two IDs at the same depth can be derived from comparing their branch numbers. The one with the lower branch number comes before the one with the higher branch number. This is so because they are either leafs of the same node, or we can find subtrees that contain them and which have roots that are leafs of the same node. Being leafs of the same node

```
// Return value 0 means id1 == id2
// 1 means id1 after id2
// -1 means id1 before id2
int compare(id1, id2):
    depth1 = id1.depth
    branch1 = id1.branch
    depth2 = id2.depth
    branch2 = id2.branch

    if depth1 == depth2:
        if branch1 == branch2:
            return 0
        else if branch1 > branch2:
            return 1
        else:
            return -1

    inversor = 1

    if depth1 > depth2:
        depth_aux = depth1
        branch_aux = branch1
        depth1 = depth2
        branch1 = branch2
        depth2 = depth_aux
        branch2 = branch_aux
        inversor = -1

    delta_depth = depth2 - depth1

    branch2 = branch2 / (2 ^ (delta_depth - 1)) //integer division
    branch1 = branch1 * 2

    result = 1
    if branch1 <= branch2:
        result = -1

    return result * inversor
```

Listing 4.6: ID comparison function.

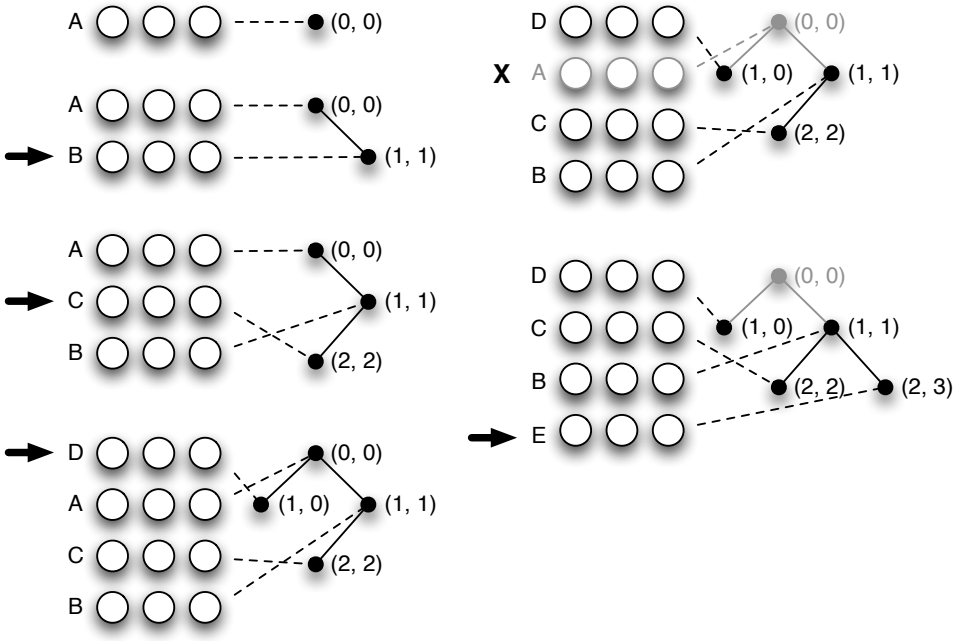

Figure 4.5: Example of ID generation.

determines an implicit position relationship in the sequence, because the first leaf comes before the node and the second after it. If the two IDs have different depths, we will call ID_1 the ID with the lower depth and ID_2 the ID with the higher depth. We can find a node ID that the root of a subtree that contains ID_2 and that is one level deeper than ID_1. We do this by performing an integer division of the branch number of ID_2 by $2^{\delta-1}$, where δ is the depth of ID_2 minus the depth of ID_1. We thus arrive at the branch number of this auxiliary node, which we will call b_{aux}. Also, we will call the depth of this node d_{aux}. All nodes that come before ID_1 in the sequence and have a depth of, at least, d_{aux}, belong to a subtree with root at level d_{aux}. Considering b_1 the branch number of ID_1, all such subtrees that contain nodes that come before ID_1 in the sequence must have a root with a branch number of, at most, $2b_1$, and all that contain nodes that come after ID_1, a branch number of, at least, $2b_1 + 1$. Thus, ID_2 comes before ID_1 if $b_{aux} <= 2b_1$, after otherwise.

Having this comparison function, it is trivial to generate an ordered list of IDs, and to keep it ordered when inserting new IDs.

4.3.2 Connection tags

Connection tags are a mechanism used to identify which connections are equivalent. It consists of assigning three integer values to each connection. There is one value for the connection, one for the origin an one for the target. We use three values instead of just one because of the split/join mutation operators that we will describe later. Two connections are equivalent if their tag values are equal.

During the evolutionary process, when a connection is passed from the parent to a child, its connection tags are also passed. When a new connection is created, tags have to be assigned to it. In this case there are two alternatives: either we find an equivalent connection in the population and copy its connection tags to the new one, or we generate new values. New values are generated by simply incrementing a global variable in the systems that holds the last value assigned.

This process requires that when a gridbrain with new connections is added to the population, these new connections are compared against all connections in all existing gridbrains to attempt to find an equivalent. A connection with unassigned tags is considered equivalent to another one if, from the connection network perspective, both its origins and targets are equivalent. Two origins are equivalent from a connection network perspective if the following conditions are met:

- Origin components are of equal types;
- Origin components both have no incoming connections or they share at least one equivalent incoming connection.

In the same vein, two targets are equivalent if:

- Target components are of equal types;
- Target components both have no outgoing connections or they share at least one equivalent outgoing connection.

These rules define a constructive mechanism that ties in with the progressive complexification of the gridbrains in the evolutionary process. They allow us to match connections that create the same functionality.

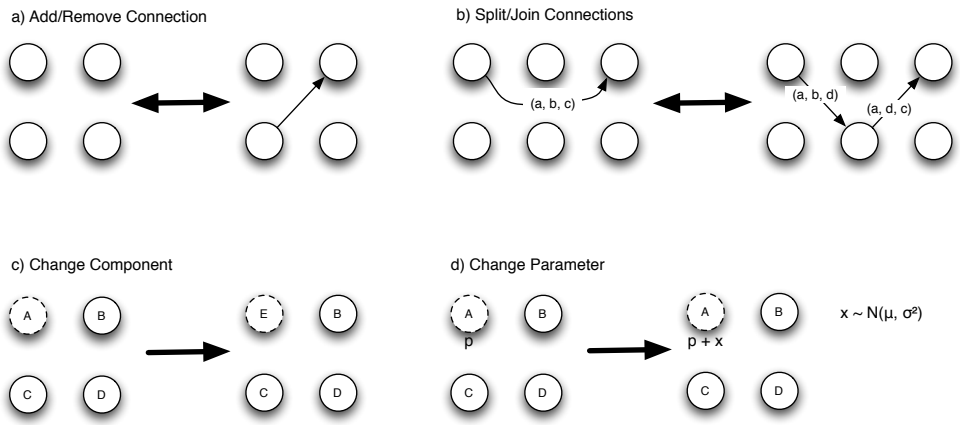

Figure 4.6: Gridbrain mutation operators.

4.4 Genetic Operators

In the gridbrain model we do not specify the evolutionary algorithm. We just provide a set of genetic operators that can be used by such an algorithm to produce reproduction with variation. In the following chapter we will propose an evolutionary algorithm suitable for continuous multi agent simulations that takes advantage of these operators.

We provide two types of genetic operators that are usual in evolutionary computation systems: mutation and recombination. We also provide a formatting operator that deals with adapting the shape of the grids as evolution progresses and is related to the complexification of gridbrains.

4.4.1 Mutation

We define mutation operators at connection level and component level.

There are two pairs of connection level operators: *add/remove* and *split/join*. The operators in each pair are symmetrical, one performing the inverse operation of the other.

As showed in figure 4.6a), the add operator inserts a new valid connection and the remove operator deletes an existing connection from the gridbrain. These mutations occur with respective probabilities of p_a and p_r. These probabilities are relative to the number of connections in the gridbrain. For each existing connection there is a p_a probability that a new one is generated. Each existing connection has a p_r probability of removal. Multiple

connections may be added or removed in the same mutation step. This defines a process that adapts to the size of the gridbrain. The number of mutations tends to increase as the gridbrain size increases, but the probability per connection remains the same. Also, if $p_a = p_r$, and disregarding evolutionary pressure, the number of connections will tend to remain stable. This is part of the measures we take to avoid bloat.

In figure 4.6b) we show the split/join operators. The split operator routes an existing connection through an intermediary component in the grid. If we have a connection from component A to component B, two new connections will be created, from A to the intermediary component C and from C to B. The original connection is removed. If the B or C connection already existed, their respective add operation is just ignored. If A and B are on the same grid, the component C must be in a column with a number higher than A and lower than B. If they are on different grids, C must be in either the origin grid with a column number higher than A or in the target with a column number lower than B.

In the figure, connection tags are shown as (t_c, t_o, t_t) tuples, where t_c is the connection value, t_o is the origin value and t_t is the target value. As can be seen, when a connection is split, both new connections inherit the t_c value from the original connection. The first new connection inherits the t_o value and the last new connection inherits t_t. A new tag is generated, and is assigned to t_t of the first connection and t_o of the second. This way, the origin and target components of the original connection remain equivalent for the purpose of generating new connection tags. Also, it can be determined if two connections originated from the same connection via split, by checking if they share the same t_c value. The reverse operation, join, is performed only to adjacent connections with equal t_c values. A split followed by a join of the resulting connections will result in the original connection, with the original connection tags.

Splits and joins occur with respective probabilities of p_s and p_j. Again, these probabilities are per existing connection. Due to the way the split/join operators work, only one connection at most will meet this condition. The join will be performed to the selected connection and the one that meets the condition. With $p_s = p_j$ and no evolutionary pressure, the number of connections will tend to remain stable. Notice that when a connection

is split, only the first resulting connections will be eligible for the symmetrical join, so the overall probabilities remain balanced.

There are two component level operators: *change component* and *change parameter*.

In figure 4.6c) we can see that change component replaces an existing component with a new one. The new component is randomly selected from the grid component set, and its parameter is initialized with a random value extracted from a uniform distribution in $[0, 1]$. These mutation occur with a probability of p_c per component in the gridbrain. A variation of this operator is *change inactive component*, which works the same way but only affects inactive components. This variation is less destructive as it only produces neutral mutations, which are mutations that do not affect the phenotype.

The *change parameter* operator, as shown if figure 4.6d), alters the parameter of a component by adding a value x to it. The resulting value is truncated to the $[0, 1]$ interval. The value x is randomly generated from a normal distribution $x \sim N(\mu, \delta^2)$, where the average, μ is 0. The operator may be parameterized by the standard deviation, δ. An higher standard deviation will produce larger changes in parameters.

4.4.2 Recombination

Creating a viable recombination operator for the gridbrain was a challenge. Such an operator has to deal with the recombination of differently shaped grids, containing networks with different topologies. Furthermore, there are different types of components, so the network nodes are heterogeneous. Several approaches were attempted, using both syntactic and semantic recombination. We will present the only viable approach we found. It performs semantic recombination of connections, taking advantage of the previously described tag mechanism. This mechanism allows us to identify connections or groups of connections that correspond to the same functionality.

The operator we present performs the following steps:

- Create the child gridbrain, with grids the same size and with the same row/column IDs as parent A;
- Recombine connection groups from parents A and B into child;

- Recombine components from parents A and B into child.

A connection group is a set of connections that have the same t_c tag value. A connection group with more than one element results from one or more split mutations. When recombining connections, we start with parent A and iterate through its connection set. For each connection group we find, we check if it is present in parent B. If it is, we import the group from one of the parents with equal probability. If it is not, the group may be imported or discarded, with equal probability. Then we iterate through the connection set of parent B. Again we check for each connection group if it exists on the other parent. If it does, we ignore it, has it was already recombined. If it does not, the same random process of importing or discarding is used, as with the connection groups that only exist on parent B.

Importing a connection group consists of importing all of its connections. When importing connections, we do not use grid coordinates, as the grids in each parent may have different shapes. We take advantage of the row/column ID mechanism that we described previously to import the connection to the correct location in the child grid. In the parents, we translate connection coordinate tuples $(x_{origin}, y_{origin}, g_{origin}, x_{target}, y_{target}, g_{target})$ to ID tuples $(IDcol_{origin}, IDrow_{origin}, g_{origin}, IDcol_{target}, IDrow_{target}, g_{target})$. Then we convert the ID tuples to child coordinate tuples and add the connection. As the child is created with the same shape and row/column IDs of parent A, it may happen that a connection from parent B has no equivalent row or column in the child. In this case, the equivalent row or column is created. We add it in the correct position and import its ID from parent B.

Translating a coordinate to a row/column ID consists of obtaining the element in the corresponding ID list, at the index position given by the coordinate. An x coordinate corresponds to the ID at position x in the column ID list and an y coordinate corresponds to the ID at position y in the row ID list. The reverse translation is performed by finding the position of an ID in the list.

The last step is to recombine components. For each component in the child, we check to see if a component in the equivalent position (given by its column and row IDs) exists in each parent. If it exists in both parents, one of them is chosen with equal probability to

have the component copied from. If it exists in only one parent, the component is copied from it. If it exists in neither parent, a new component is randomly selected from the grid connection set. A component in a child may have no equivalent component in any of the parents if it is situated in an intersection of a column that only exists in one of the parents with a row that only exists on the other.

This recombination operator always produces valid gridbrains and is able to recombine functionalities in a meaningful way. In the absence of evolutionary pressure, it does not introduce any probabilistic bias towards the increase or decrease in the total number of connections.

4.4.3 Formating

Formating is an operator that adapts the shape of the grids according to the network contained in the gridbrain. It is of a non-stochastic nature and can be seen as an adaptation mechanism. The changes it performs do not affect the phenotypical expression of the individual. The purpose of formating is to regulate the search space of the evolutionary process. It is part of the complexification process. We attempt to create systems that are initialized with empty brains, and that undergo an increase in complexity as higher quality solutions are found. Solutions are constructed through iterated tweaking, in a similar fashion to what happens in nature. Our goal is to have the size and complexity of the gridbrains to be determined by the demands of the environment.

Formating operates at grid level and performs changes taking into account the current *active network*, which is the set of active connections. For each grid, this operator determines if rows and columns should be added, removed or kept.

Under the genetic operator framework we are describing, complexification is driven by connection level mutations. Add/remove and split/join are the operations that directly affect network complexity. Formating alter the shape of the grid in such a way that the active network is kept unaltered, but the following mutation step has freedom to complexify the network in all the manners available to connection level operators. This means that the following should be possible:

1. Add a new valid connection between any of the active components;

2. Add an intra-grid incoming connection to any active component;

3. Add an intra-grid outgoing connection to any active component;

4. Branch the output of any active component to an inactive component in any of the following columns;

5. Split any of the active connections.

The first condition is guaranteed by not altering the active network. The second condition is met by having no active components in the first column of the grid, and the third by having no active components in the last one. The fourth condition is met by having at least one line with no active components, and the last condition is met by having all active connections skip at least one column. Formating alters the grid by inserting and deleting rows and columns, making sure that all these conditions are met with the smallest possible grid, without changing the active network.

If new rows or columns are created in a formating step, the new components are selected randomly from the grid component set. Furthermore, if a grid is null, meaning it has no columns or rows, a row and column is always created so that the evolutionary process can move forward. In fact, in the experiments we will describe, we just initialize the gridbrains with null grids.

In listing 4.7 we present the pseudo code for the format operator. It is in a high level format to be more clear. A column is active if it contains at least one active component. The *jump* value of a connection is the difference between its target column coordinate and its origin column coordinate. It only applies if the origin and target are in the same grid. The *minimum jump* of a column is the lowest jump value of all the intra-grid connections originating from it. If the column has no intra-grid connections originating from it, its minimum jump is 0. As can be seen, this value is used in the algorithm to determine if a new column has to inserted as to allow for future connection splits, or if an inactive column can be deleted, and still allow for the split of any connection in the gridbrain.

The format operator can work at grid level and still enforce the restrictions stated above for the entire gridbrain. Maintaining inactive columns at the beginning and ending

```
if grid is null:
    add_column()
    add_row()
    return

jump = 0

for each column in grid:
    x = column coordinate

    if column is active:
        jump = minimum jump from this column
        if x == 0:
            add_column_before(column)
        else if x == grid width - 1:
            add_column_after(column)
        else if jump == 1:
            add_column_after(column)
            jump = 2
    else:
        if no active column before and x >= 1:
            delete_column(column)
        else if no active column after and x < grid with - 1:
            delete_column(column)
        else if jump > 1:
            delete_column(column)

    jump = jump - 1

max_active = 0

for each row in grid:
    active = 0

    for each component in row:
        if component is active:
            active = active + 1

    if active == 0:
        delete_row(row)
    else if active > max_active:
        max_active = active

if max_active == grid height:
    add_row_to_end()
```

Listing 4.7: Format operator.

of all grids guarantees that any inter-grid connection is splittable, and that it can be split by a component in the origin or target grids.

The format operator should be applied before the mutation step in the generation of a new gridbrain. This way the gridbrain is formated according to the above mentioned rules when connection level mutations are performed. The sequence of genetic operations when generating a gridbrain is $recombine(parent1, parent2) \rightarrow format \rightarrow mutate$ if recombination is being used, or $clone(parent) \rightarrow format \rightarrow mutate$ for a single parent reproduction. We are considering the *mutate* operation to be the combined application of all the mutation operators described above, according to their respective probabilities.

4.5 Redundancy and Neutral Search

The gridbrain representation is highly redundant, because many genotypes map to the same phenotype. There are several reasons for this. One is the inactive part of the gridbrain, consisting of inactive components and connections. These components and connections are not expressed in the phenotype. Multiple permutations of inactive components and connections can exist in a gridbrain with the same active network. Furthermore, the same active network may be represented in gridbrains of different sizes. Another reason is that the same active network may appear in different configurations. For example, the order in which active components are placed in a column does not alter the functionality of the active network. Even active components in different columns may lead to the same phenotype, if the equivalent active connections are possible and present. Yet another reason is that even different active networks may have equivalent phenotypes, by the presence of multiple connections with the same functionality.

Changes in unexpressed parts of genotypes during evolution constitute a process knows as *neutral search*. As discussed in section 2.5, neutral search is acknowledged as an important part of the evolutionary process, both in biology and evolutionary computation. The gridbrain representation and genetic operators implicitly promote neutral search. No operator explicitly activates or deactivates parts of the gridbrain. As discussed before, this happens as a consequence of topological changes, by active connection paths being

formed or broken.

In the later chapters we will present experimental results on the role of neutral search in gridbrain evolutionary processes.

While most forms of redundancy in the gridbrain appear to be either helpful or neutral to the performance of the evolutionary process, we identified one type that is potentially detrimental. This latter type is the formation of multiple equivalent active connection paths that include redundant origin and target components.

Let us suppose that the addition of a connection from input component I to output component O leads to an increase in the quality of an individual. If multiple components equal to I and O can appear in the gridbrain, it is possible that this connection is formed several times. In fact, it is likely that it is formed several times as this creates a protection against the removal of that type of connection. Now, let us suppose that splitting this connection through an intermediary component C leads to a further improvement in the quality of the individual, and that more improvement can be attained by establishing certain connections to the C component. Experimentation has shown this to be a common scenario in gridbrain complexification. If multiple connections of the type $A \rightarrow B$ are present, the effect of the beneficial split mutation on one of them may be mitigated or nulled. It is not likely that the same beneficial mutation will occur at the same time in the several equivalent connections. The unchanged connections may override the behavior of the changed one, preventing the expression of the behavior that leads to an increase in quality.

Notice that this may seem counter-intuitive, as the formation of redundant paths is thought to be important in both natural and artificial evolution. There is an important distinction to be made here. The type of redundancy we described includes input and output component redundancy. To our knowledge, this problem is not present in other evolutionary algorithms, as free placement of input and output terminals is not commonly allowed. The problem is that the free form placement of components used in the gridbrain makes it too easy for the evolutionary process to create many copies of the same input-output path, since it can place many instances of the same input and output components in the grids. This was found to be especially problematic in the early stages of evolution,

as the first fitness improvements tend to be direct input-output connections. It was observed that the process tends to create several copies of the first successful paths found, including the redundant input/output components, leading to structures that are difficult to further evolve. While some degree of path redundancy is likely helpful, as attested by the referenced literature on the importance of neutral search in genetic programming, the situation we described leads to an excess of this form of redundancy.

One simple addition to the gribrain model prevents the appearance of most cases of the above mention redundancy. This is to make input and output components unique. When randomly selecting a component from the component set to add to the gridbrain, which happens in both the formating and change component operators, a check is added to verify if the new component is unique and already exists in the gridbrain. If so, another one is chosen. This prevents the formation of multiple active paths from equivalent inputs to equivalent outputs.

4.6 Gridbrain Distance and Diversity Metrics

In evolutionary systems, it is useful to be able to measure the diversity in a population. To help measure pheotypical diversity in a population of agents using the gridbrain, we define a way to quantify the distance between two gridbrains. The distance, d, is computed by comparing connections in the two gridbrains, with the help of the tag mechanism. It is determined by the following expression:

$$d = 1 - \frac{ms(G_1, G_2)}{max(C_1, C_2)},$$

where C_1 and C_2 are the number of connections in each gridbrain and ms is a matching score function for the connections sets (G_1 and G_2) of the two gridbrains. The matching score is computed by iterating through each connection in one of the gridbrains. It is initialized to zero and then, the tag of each connection is compared with the tags of the connections of the other gridbrain. If a tag is found with equal origin and target IDs, 1 is added to the score. If a tag if found with only an equal origin or target ID, 0.5 is added to the score.

The overall diversity of a population of gridbrains can be computed by averaging the distance of all possible pairs of individuals.

4.7 The Gridbrain Library

We implemented the gridbrain model in the form of a C++ library, *libgridbrain*. This library and its code are provided free for non-commercial uses under the GPL open source license. The C++ language was chosen for performance reasons. The gridbrain is intended to be use as the brain of autonomous agents in real time multi-agent simulations, so speed of execution is of great importance. The library may be bound to other languages, providing integration with simulation environments. We provide one such binding, to the LUA language [Ierusalimschy *et al.*, 2006]. We take advantage of this when integrating the gridbrain library with LabLOVE [Menezes, 2008b], our multi-agent simulation environment. We have written libgridbrain in portable code and used the CMake tool [Hoffman and Martin, 2003] to provide a portable compilation environment. This way, libgridbrain can be easily compiled in the main operating systems.

We developed libgridbrain using a test-driven methodology [Beck, 2003]. As a consequence of this, the library includes a test suit that is used to test the correctness of the code every time it is compiled. We believe this was an important contribution to the experimentation process, as it allowed us to ensure that modifications to the code did not introduce side effects that would compromise the comparison with previous results. Also, as we developed the library in a desktop computer and performed experimentation in clusters with different hardware and operating systems, the test suits allowed us to ensure that the change in platform did not introduce side effects.

The library has an object oriented architecture and was developed with modularity in mind. We attempted to define simple and clear interfaces to create and interact with gridbrains. We were also concerned with the ability to extend the library, particularly in the creation of new types of components.

New components may be created by extending the abstract *Component* class and implementing the *reset*, *input* and *output* interfaces. We include in the library the component

set described above, and these component were implemented by this same method.

The gridbrain library also provides the ability to write the graphical representation of a gridbrain to a file, in the *scalable vector graphics (svg)* format [Andersson *et al.*, 2003].

Chapter 5

Simulation Embedded Evolutionary Algorithm

In this chapter we describe an evolutionary algorithm aimed to be used in continuous multi agent simulations. The most well known population based evolutionary algorithms are generational. This means that a set of individuals is created, evaluated and then the next set is generated by selecting, recombining and mutating the individuals in the previous set according to their quality. This poses a problem in continuous agents simulations, where we do not want to remove all the agents from the world and create a new batch, thus introducing a discontinuity. In this case we need an algorithm that integrates in a seamingless as possible manner with the simulation.

We present a *simulation embedded evolutionary algorithm (SEEA)*. This is a *steady state* evolutionary algorithm, meaning that there are no generations. Its goal is not only to integrate with the simulation environment, but also to perform its evaluations in a way that is spread across execution, instead of condensed in certain moments, which could pose problem with real-time requirements. Moreover, it supports multiple species. A species of agents has reproductive isolation. This means that selection and genetic operations are performed within a species, and only amongst individuals that belong to it. Multiple species allows for the co-evolution of agents, that can have conflicting or complementary goals. Multiple species with conflicting goals can lead to *arms race* scenarios, which are one of the acknowledge sources of complexity increase in nature.

SEEA was conceived to be used with the gridbrain, as part of our goal of developing a framework of evolutionary computational intelligence for multi agent simulations. However, it has no dependencies to the gridbrain, and can be used with any agent brain representation that supports mutation and recombination operators.

5.1 Open evolution and fitness

In the field of evolutionary multi agent simulations, a common goal is to attain open evolution, with similar generative power with what can be observed in nature. The simulations described in chapter 3 follow different approaches towards this goal. Although they model agents differently, they all rely on the co-evolution of diverse agents in the simulated environment. It is generally accepted that co-evolution is one of the mechanisms behind the emergence of complex life forms in nature.

Another thing that these simulations have in common is that they attempt to avoid the explicit definition of a fitness function. They attempt to introduce reproductive mechanisms into the simulation that emulate the natural world. These mechanisms are related to the level of abstraction and model of the simulation. The Tierra model is perhaps the lowest level in terms of reproduction. The reproduction mechanism itself is evolved in the form of sequences of machine code instructions. In fact, the entire dynamics of Tierra is based on reproductive technology. Tierra can be seen as a game where computer programs try to out-reproduce each other. This leads to the emergence of the several reproductive strategies previously described. The Tierra model is very elegant, but it suffers from an initialization problem. The evolutionary process will not start until some program finds a way to copy itself. Finding such a program is not trivial. If the Tierra simulation were to be initialized with random data, the search for this program would be essentially by *brute force*. Finding a program that initialized the evolutionary process would be highly unlikely. The solution used in Tierra is to initialize the environment with human designed programs that are capable of copying themselves. This introduces information into the simulation and influences it.

Avida expands the Tierra model, in an effort to model phenomena that are not limited

to reproductive dynamics. Programs can improve their reproductive and survival chances by performing predefined tasks. The execution of this tasks results in the agents being awarded *merit* points. In our perspective, this is equivalent to defining a fitness function.

Echo models reproduction as an event that is triggered by the environment. It happens when an agent has the necessary resources to create a copy of itself. This mechanism may be considered higher level than Tierra, in that it is provided by the environment, but on the other hand allows for a great degree of freedom in interactions between the agents. It is a very free form simulation, created in an attempt to generate a diversity of inter-relationships and diversity of species. Although interesting results have been produced, this goal was not yet achieved. In accordance with its goals, Echo simulation are uncontrollable. They have great potential for scientific research in the field of complex adaptive systems but are not amenable for engineering pursuits.

Polyworld is a higher level simulation. Here, reproduction is provided as an action available for the agents to execute. This action as a cost in terms of energy. To be able to perform the action, agents have to acquire energy from the environment. The ways in which energy may be obtained, for example by eating, are predefined in the simulation. Again we will claim that there is an hidden fitness function in this simulation, in the form of energy values. Furthermore, Polyworld suffers from a similar initialization problem as Tierra. In Polyworld this is solved by running a genetic algorithm with an explicit fitness function until agents that can gather enough energy and reproduce appear. Again this influences and conditions the simulation at its start.

In this work we chose to use fitness functions. We aim at engineering applications, and thus, some amount of control and the ability to create simulations where reproduction is not necessarily an explicit part of the model. The existence of an explicit fitness function does not preclude the emergence of complexity. Agents live in an environment that can possess some or all of the previously described characteristics of CAS. It is this environment, according to its present conditions, that generates the fitness evaluation. On the other hand, by using fitness functions, we have some control on the behavior of the system.

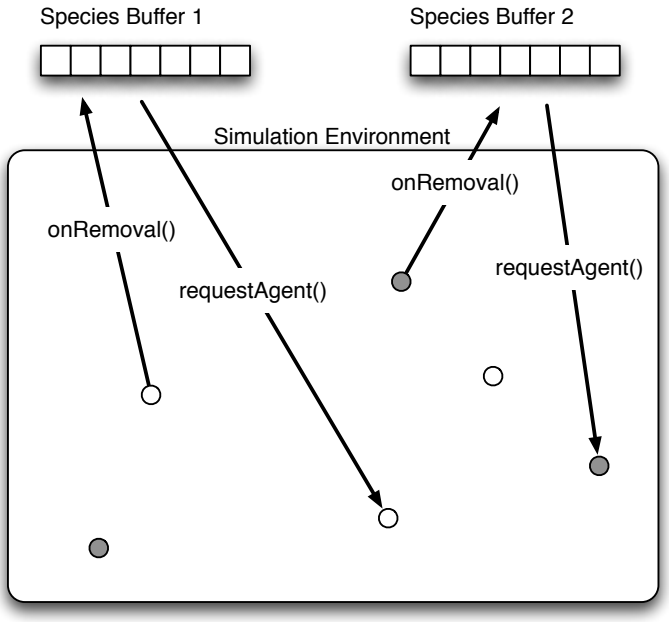

Figure 5.1: SEEA integration with a simulation with two agents species.

5.2 The basic algorithm

The SEEA algorithm maintains a fixed sized buffer of individuals for each species that exists. As shown in figure 5.1, it interfaces with the simulation environment through two function calls: *onRemoval()* and *requestAgent()*. The *onRemoval()* call processes the removal of agents from the environment, while the *requestAgent()* call provides a new agent to be sent to the environment.

The species buffer constitutes a genetic memory of the species, containing some of the individuals found so far. We assume that the environment is capable of producing a fitness value for an agent when it is removed. This fitness value reflects the performance of the agent during its lifespan in the simulation, by some predefined criterion. When and agent ends its lifespan and is removed from the simulation, its fitness is compared against the fitness of an individual in a random position in its species buffer. If it is equal or greater, it takes the place of the original individual in the buffer. This is how evolutionary pressure is introduced in the system.

If, in a fitness comparison, the removed individual has a lower fitness that the buffer individual, the process of *fitness ageing* takes place. This consists of reducing the fitness

of the buffer individual by a certain factor. The fitness ageing process in parameterized by the *ageing factor* $a \in [0, 1]$. The fitness of the buffer individual is updated by using the following expression:

$$f_{new} = f.(1 - a)$$

This means that, the higher a is, the larger the impact of fitness ageing in the evolutionary process.

The purpose of fitness ageing is twofold: keeping the evolutionary process dynamic and promoting diversity. SEEA aims at evolving agents that operate in complex environments. The fitness evaluation of an agent is relative to the period in which the agent interacted with the environment. As the environment changes, this evaluation becomes less meaningful. The ageing process allows lower fitness but more recent agents to replace the ones in the buffer, but it still conserves evolutionary pressure. The higher the original fitness of an agent in the buffer, the more comparisons it is likely to win. The higher the fitness of the challenging agent, the more likely it is to win the comparison.

SEEA without fitness ageing is an elitist evolutionary algorithm. In this case, only the individuals with the highest fitnesses enter the buffer. Ageing allows for lower fitness individuals to replace individuals in the buffer that suffered the fitness ageing process. This helps counteract the effects of elitism, which can lead the population to be stuck at local maxima.

Fitness ageing is an adaptive mechanism, because it is driven by agent death events from the simulation. The more agents are removed, the more ageing takes place at the buffer, as more possibilities are tested.

The criteria for removal of an agent from the simulation is defined by the simulation itself. Agents may just have a limited lifetime, or they can die from lack of energy or be destroyed by other agents, or a combination of these. This makes no difference to the SEEA algorithm, which just reacts to the events of agent removal and agent request.

In listing 5.1 we present the data structure used to define a species with SEEA. Any number of species may exist, with one instance of this structure for each species. The list of species is kept in *species_list*. In the case of the work presented in this thesis, the

```
species:
    id : int
    buffer_size : int
    buffer : array
    recombine_prob : float
    ageing : float

species_list : list of species
```

Listing 5.1: SEEA data structures.

individual is an agent controlled by a gridbrain. The data structure contains a unique identifier, *id*. This identifier is propagated to organisms when they are generated, so that when removed, they can be matched to the corresponding species. The buffer, an array of agents, is also contained in this structure, along with *buffer_size*. The two other fields, *recombine_prob* and *ageing*, parameterize the evolutionary process.

If figure 5.2 we present the pseudo code for a function that adds a species to SEEA. It receives a *base_agent* parameter, which is an instance of the agents that are to constitute the species. The buffer is initialized with *buffer_size* clones of the base agent. The species *id* is generated by incrementing the *id* of the last included species by 1 or, if the species list is empty, set to 0.

In the work presented in this document we evolve gridbrains which are initialized as empty. In other cases where initial diversity is relevant, the initialization procedure may be altered to populate the buffer with mutations of the base agent, instead of unaltered clones.

When an agent is removed from the simulation environment, the *onRemoval()* SEEA function is called, passing as parameter the object that defines the removed agent. The pseudo code for the *onRemoval()* function is presented in listing 5.3. We assume that the agent object includes a field for its fitness value, and that the simulation updated this value before calling *onRemoval()*. As can be seen, a comparison is performed with an agent at a random position in the buffer, which can result in replacement or ageing.

When the simulation needs a new agent, the *requestAgent()* SEEA function is called, passing as parameter the species *id* for the requested agent. A new agent is created by either performing a recombination of two agents in the buffer, followed by a mutation of the

```
function addSpecies(base_agent,
                    buffer_size,
                    recombine_prob,
                    ageing):
    s = create species instance

    if species_list is empty:
        s.id = 0
    else:
        last_s = last species in species_list
        s.id = last_s.id + 1


    s.buffer_size = buffer_size
    s.recombine_prob = recombine_prob
    s.ageing = ageing
    s.buffer = array[s.buffer_size]

    for i = 0 to s.buffer_size - 1:
        org = base_agent.clone()
        s.buffer[i] = org

    species_list.add(s)
```

Listing 5.2: Add species function.

```
function onRemoval(agent):
    s = species with id == agent.species_id
    pos = rand_int(s.buffer_size)
    buf_agent = s.buffer[pos]

    if agent.fitness >= buf_agent.fitness:
        s.buffer[pos] = agent
    else:
        aged_fitness = buf_agent.fitness * (1 - s.ageing)
        buf_agent.fitness = aged_fitness

    remove agent from simulation
```

Listing 5.3: Agent removal funtion.

```
function requestAgent(species_id):
    s = species with id == species_id

    pos1 = rand_int(s.buffer_size) // random integer between
                                   //0 and s.buffer_size - 1
    parent1 = s.buffer[pos1]

    prob = rand_float(1.0) // random float in [0, 1]

    child : agent

    if prob <= s.recombine_prob: // recombine
        pos2 = rand_int(s.buffer_size - 1)

        if pos2 >= pos1:
            pos2 = pos2 + 1

        parent2 = s.buffer[pos2]

        child = parent1.recombine(parent2)
    else:
        child = parent1.clone()

    child.mutate()

    child.species_id = species_id
    send child to simulation
```

Listing 5.4: Agent request function.

resulting child, or just a mutation of the clone of one agent in the buffer. Recombination is performed according to the recombination probability for the species. The resulting child is then sent to the simulation.

Notice that mutation probabilities are not defined in the species structures. These are specific to the representations and operators being used. We assume that it is possible to call a *mutate()* function on agent objects that performs mutations according to the operators and mutation probabilities being used. In the case of the gridbrain, this is done in the ways described in the previous chapter. Only the recombine probability and ageing factor are defined in SEEA species, because these are the parameters that affect the operation of the algorithm.

SEEA does not control the sizes of populations in the simulation. It is up to the simulation to decide when to remove and agent or request a new one. In the work presented in this thesis we work with fixed sized populations. This is because we are interested in focusing on the study of the evolution of computational intelligence, and not on variable population dynamics. Clearly the two can co-exist, but we decided to remove the latter for simplicity. The simulation performs an initial number of *requestAgent()* calls to initialize the species populations, and then performs a *requestAgent()* every time an agent is removed, keeping the population numbers constant. SEEA can however be used with variable-sized populations with no change required.

An advantage of the buffers is that they allow the evolutionary process to work with low population sizes on the simulation. As is common to all population based algorithms, a small population size can compromise the evolutionary process. This happens because small populations are less robust to bad mutations and do not provide diversity, becoming easily stuck in local maxima. It is common for evolutionary agent simulations, like several artificial life simulations, to use the population of agents in the simulation as the basis for evolution. New agents are created directly from agents present in the simulation environment. While this provides for a more realistic simulation of certain biological processes, it may not be ideal if the simulation of such processes is not important.

With sufficiently large species buffers, the evolutionary process works even if we are working with small sized populations. It can even work with populations of just one

individual. In physically simulated worlds, the computational cost of maintaining the simulation tends to grow exponentially with the number of simulated entities, due to collision detection and vision processing. With SEEA, we are able to evolve multi-species simulations without a prohibitive number of agents in the simulated world.

5.3 Alternative selection methods

In the basic algorithm presented, the selection of agents from species buffer is random. Evolutionary pressure arises from selection performed when agents die. It is at this point that the value of the final fitness of the dead agent is compared against the fitness value of an agent in the buffer, and a decision is made to replace the agent in the buffer with the new one or not. However, selection can also be performed when selecting parents from the buffer, when the creation of a new agent is requested. We will propose two methods that are commonly used in genetic algorithms: *tournament* and *roulette*. We will refer to the base method of random selection as *simple*.

The tournament method consist of choosing a set of n random individuals from the buffer. The individual from this set with higher fitness is selected to be the parent. If two parents are necessary, the tournament selection method is applied again to find the second parent.

The roulette method consists of randomly selecting the parents in a way that the probability of an agent being selected is proportional to its fitness. This is done by computing the total fitness of the buffer, F, by summing all the fitnesses of all the agents in the buffer. The relative fitness of an agent a is determined by the expression: $p_a = \frac{f_a}{F}$, with f_a being the fitness of the agent. A random value, p, is then selected from an uniform distribution in $[0, 1[$. An agent a is selected if:

$$\sum_{i=0}^{a-1} p_i \leq p < \sum_{j=0}^{a} p_j.$$

5.4 Group behavior

The basic SEEA algorithm presented is capable of creating evolutionary pressure so that, as the simulation advances, agents tend to increase their fitness value. One limitation of it is that it only promotes the emergence of behaviors that benefit each agent directly. It may be insufficient to promote the emergence of collective behaviors, where some degree of altruism is needed.

In nature, many examples can be found of cooperative behavior. In some of these cases, individuals even display behaviors that are detrimental to their own survival in favor of the common good. The evolution of altruism poses problems to classical Darwinian theory. At the biological level, two main hypothesis have been advanced: *kin selection* and *group selection* [Smith, 1964].

Individuals who are related share genetic information to a degree. The closest the relation, the more genetic information is shared. Kin selection is based on the idea that an improvement in the survival and reproductive chances of an individual also helps the propagation of the genetic information of its relatives. If genetic similarity is high enough, an altruistic behavior, even with a cost to the individual, may be beneficial to the probability of propagation of its genetic information. This idea has been formalized by Hamilton in the following inequality, known as *Hamilton's Rule* [Hamilton, 1963, 1964]:

$$rB > C$$

where r is the genetic proximity of two individuals, B is the survival and reproductive benefit to to target of the behavior and C is the cost in terms of survival and reproductive chances to the originator of the behavior. If the inequality holds true for a behavior, it will tend to be selected, despite of its cost to the individual.

Group selection is based on the idea of Darwinian selection at group level. The hypothesis is that genetic traits may be selected not because they are beneficial to the individual, but to the group it belongs to. Groups selection is a controversial hypothesis, dividing theoretical biologists to the present day [Smith, 1964; Williams, 1966; Wilson, 2005].

One of the first models of group selection is called the *haystack model* [Wilson, 1987].

It can be informally formulated this way: suppose a group of animals of the same species live in groups. Each group lives isolated in an haystack. From time to time, they all leave their haystacks at the same time, mate, and then form new groups that return to haystack isolation. During the isolation periods, mating within the group takes place. A genetic trait that is beneficial to the individual will tend to be selected during the isolation periods, as normal per Darwinian evolutionary theory. However, genetic traits that are beneficial to the group will cause an increase in population in these groups. When all the groups meet and mate, groups that had individuals with more traits that are beneficial to the collective will be represented in larger numbers. This will cause evolutionary pressure for the selection of traits that are beneficial to the group. If these traits have a cost to the individual, they will tend to be removed during the isolation period, but if the isolation period is not too long they will survive to the next meeting of groups.

Theoretical studies suggested that group selection based on models like the haystack has no significant impact on real ecosystems, and does not explain the emergence of cooperative behavior in species [Williams, 1966].

A recent model of group selection has been proposed, called *multilevel selection theory* [Wilson, 2005]. Multilevel selection proposed that there are units of selection above the gene. The hypothesis is that groups exist in nature as super-organisms, with functional units form by the individuals of the species. This way, selection also takes place at a high-level than the individual. D. S. Wilson, one of the proposers of this theory, suggests that the Hamilton rule should be extended to:

$$(r.B_k + B_e) > C$$

where B_k is the benefit to kin and B_e is the benefit to the group.

In this chapter we present two extensions to the basic SEEA algorithm, aimed at promoting the emergence of collective behaviors in species: *super sisters* and *group fitness*. The first one is inspired by kin selection models, while the second one is inspired by group selection.

5.4.1 Super Sisters

The *super sisters* extension to the base SEEA algorithm increases genetic proximity in agents in the simulation. It is inspired by social insects [Wilson, 1971], where female siblings have a great genetic similarity due to haplodiploidy. It his theorized that the increase in kin proximity in females leads to the emergence of eusociality [Hughes *et al.*, 2008], because helping siblings improve their fitness increases the chances of propagation of the own agent's genetic information.

The super sisters mechanism increases genetic proximity in the environment by generating sequences of equal or very similar agents. It is implemented by changing the *onRequest()* SEEA function. Instead of generating a new offspring from the buffer in every request, it does so only every n_{seq} requests. This new offspring agent is stored in the species structure. The agent produced by the function is a copy of the stored offspring, so that the world is populated with a sequence of n_{seq} equal agents. We call this sequence of agents *super sisters*, in a reference to the genetic proximity of female siblings in certain species of social insects, like ants, bees and wasps.

To further increase genetic proximity, *parent rotation* is performed. Since SEEA is not generation based, sequences will overlap, and agents from consecutive sequences will be present in the environment at the same time. While the super sisters mechanism provides genetic proximity for a sequence of agents, parent rotation increases genetic proximity between consecutive sequences. The first time an offspring is generated for a species, two parents are randomly selected as usual. The position of the first parent in the buffer is stored in the species data structure. The next time an offspring is generated, the stored parent is used and a new one randomly selected. The new one takes the place of the old one in the data structure. If a recombination probability of 1 is used, agents from consecutive sequences will have one parent in common.

The super sisters mechanism is parameterized by the size of the sequence, n_{seq}. This value can be set in relation to the average size of the species population on the environment, *pop*. The *sequence factor* parameter, f_{seq} is used, with $n_{seq} = round(f_{seq}.pop)$. With $f_{seq} = 1$, super sister sequences the size of one population are generated. The higher the f_{seq}, the less overlapping of sequences in the environment, and thus the higher the genetic

```
species:
    id : int
    buffer_size : int
    buffer : array
    recombine_prob : float
    ageing : float

    // Super sisters fields
    seq_size : int     // can be set using the sequence factor
    seq_pos : int      // initialized to seq_size
    parent_pos : int   // initialized to -1
    child : agent
```

Listing 5.5: Species data structure with super sisters extension.

proximity of coexisting agents. On the other hand, the higher the f_{seq}, the less possibilities are tested by the evolutionary algorithm.

A variation of the super sisters mechanism is to perform mutations on the clone, so that agents in a sequence are not exactly equal. This allows the evolutionary algorithm to test more possibilities while still increasing genetic proximity between coexisting agents in the environment. This option is parameterized by a boolean flag, *sister mutation*. The super sisters extension to SEEA requires that both the *sequence size* and *sister mutation* parameters are stored in species data structures, as well as the current child agent, buffer position of one of the parents and current position in the super sisters sequence.

In listing 5.5 we presented the species structure with the extra super sisters fields.

In listing 5.6 we present the pseudo code for the altered *requestAgent()* function, implementing the super sisters extension.

5.4.2 Group Fitness

Group fitness aims at the same goal as the super sisters extension, but uses a different approach.

Group fitness in SEEA is a mechanism to reflect the success of other members of a species that coexisted with an agent in the simulation on the agent's own fitness value. The principle is to allow an agent to increase its fitness by developing behaviors that benefit the fitness of other agents of its species.

```
function requestAgent(species_id):
    s = species with id == species_id

    if s.seq_pos == s.seq_size:
        if s.parent_pos == -1:
            s.parent_pos = rand_int(s.buffer_size)

        pos1 = s.parent_pos
        pos2 = rand_int(s.buffer_size - 1)
        if pos2 >= pos1:
            pos2 = pos2 + 1
        s.parent_pos = pos2  // rotate parents

        parent1 = s.buffer[pos1]

        prob = rand_float(1.0) // random float in [0, 1]

        if prob <= s.recombine_prob: // recombine
            parent2 = s.buffer[pos2]
            s.child = parent1.recombine(parent2)
        else:
            s.child = parent1.clone()

        s.child.mutate()

    sister = s.child.clone()

    if s.sister_mutation:
        sister.mutate()

    sister.species_id = species_id
    send sister to simulation
```

Listing 5.6: Super sisters agent request function.

The simulation environment and fitness evaluation do not have to be altered in any way for group fitness to be applied. Instead of directly using the individual fitness evaluation provided by the environment, f_i, a *composite fitness* is used. The composite fitness, f_c is calculated by the expression:

$$f_c = (1 - g).f_i + g.f_g$$

where f_g is the *group fitness component* and g is the *group factor*. The group factor is a real value in the $[0, 1]$ interval. The higher it is, the more important the group success is to the composite fitness of each agent.

The group fitness component reflects the variation in the fitness of other agents, during the lifetime of the agent for which it is being calculated. The principle is to only reflect fitness variations that the agent may have helped cause in others. An effective way to compute the group fitness component is to maintain a group fitness sum, G for each agent. When an agent is sent into the environment, its G is initialized by applying the expression:

$$G = -\sum_{a \in S(t_0)} f_a(t_0),$$

where t_0 is the simulation time at which the agent was created, $S(t_0)$ is the set of agents in the world belonging to the same species as the agent we are calculating G for, at simulation time t_0, and $f_a(t_0)$ is the current individual fitness for agent $a \in S(t_0)$. Then, during the lifetime of the agent, each time another agent of the same species dies, we increment G by that agent's final individual fitness. When an agent dies, it's final group fitness sum is calculated by applying the expression:

$$G' = G + \sum_{a \in S(t)} f_a(t)$$

This way, in the end of the agent lifespan, G contains the summation of the variations of individual fitnesses in other agents of the same species, during that lifespan. Finally, the group fitness component is given by:

$$f_g = \frac{G}{pop - 1}$$

where *pop* is the population size of the agent's species in the environment. In case this population is variable, an average can be used. This way, f_g gives us the individual fitness variation per other agent in the environment.

For the group fitness extension, the field *group_factor* is included in the SEEA species data structure.

The group fitness and super sisters extensions to SEEA are compatible and may be used in conjunction.

Chapter 6

LabLOVE: The Simulation Tool

To perform the experiments with the models presented in the previous chapters, we developed a simulation tool called *LabLOVE (Laboratory of Life On a Virtual Environment)*. This tool allows us to run real time multi agent simulations. It uses the gridbrain library and provides an implementation of SEEA.

LabLOVE was designed following an object-oriented, modular approach. The three main modules, which are defined by base abstract classes are the following:

- Simulation (Simulation class)
- Simulation Object (SimObj class)
- Population Dynamics (PopDyn class)

The simulation module takes care of simulating the environment. It updates the state of simulation objects in each simulation cycle, according to environmental rules. For example, it may perform a physical simulation, using Newtonian laws. It computes the outcome of actions performed by agents and feeds sensory information to them. Furthermore, it provides visualization, drawing the current state of the environment for human observation.

Simulation objects are the entities that populate the environment. Simulation objects include agents, which are controlled by a gridbrain. The characteristics of objects are defined by symbol tables. Every object contains a set of symbol tables, according to the

settings of the simulation. Symbols have types, according to the type of information they represent. For example, the color of an object may be represented by an RGB symbol, which consists of a tuple of three bytes, one for each color coordinate (red, green and blue).

The population dynamics module defines the mechanism of removal of objects from the environment and generation of new objects. We provide and use an implementation of the SEEA algorithm as a LabLOVE population dynamics module. Other population dynamics modules may be created, by extending the base PopDyn class. For example, a classical genetic algorithm could be implemented. However, for the purpose of this thesis we will focus on SEEA.

The tool also includes an extendible logging module, to gather information about simulation runs. The information is stored in the format of *comma separated values (csv)* files, which is compatible with most external statistical analysis tools.

LabLOVE is written in portable C++, for performance reasons. We attempted to make it as easy as possible for modules to be developed, so that the algorithms we present in this thesis can be tested by other researchers in different fields. However, it is not a GUI based tool, and requires C++ programming abilities to extend.

For experiment configuration we use the LUA scripting language [Ierusalimschy *et al.*, 2006]. The LUA interpreter is embedded in the system, and LUA scripts are used to configure experiments. Researchers wishing to test scenarios with modules that are already implemented can easily define experiments using these scripts.

Visualization is done with the aid of the *Artist graphics library* [Menezes, 2008a], also developed by the author of this thesis.

LabLOVE is available to the scientific community under the GPL open source license.

6.1 The Simulation Model

The *cycle()* function is at the heart of the simulation. It is run repeatedly in a loop until the simulation ends. In listing 6.1 we present the pseudo code for the function. This type of simulation loop is common in computer games. It broadly consists of updating the

```
function cycle():

    for every obj in objects_to_kill:
        popdyn.onOrganismDeath(obj)
    objects_to_kill.clear()

    drawBeforeObjects()

    onCycle()

    for every obj in objects:
        obj.process()

    for every obj in objects:
        if obj is agent:
            obj.perceive()
            obj.compute()
            obj.act()

        obj.draw()

    drawAfterObjects()

    popdyn.onCycle(sim_time)

    sim_time = sim_time + 1
```

Listing 6.1: LabLOVE simulation cycle.

world and the state of each object in the world and then starting over.

The base Simulation class contains two lists: *objects* and *objects_to_kill*. The first one contains all the objects currently in the environment, while the second contains the objects that are to be removed. All the objects that belong to the second one also belong to the first. The Simulation class also contains a reference to the population dynamics object, *popdyn*. As can be seen, the first step in the cycle is to iterate through the *objects_to_kill* list and send them to the population dynamics module, through the *onOrganismDeath()* function. The population dynamics module must then react to this event according to its internal mechanisms and then perform the removal. The other event that the population dynamics receives is the cycle end, through its *onCycle()* function.

The actual update of the world state is performed by calling a set of functions of the simulation and objects. These functions are abstracted on the basis of Simulation and SimObj classes, and are implemented when creating specific simulation environments. To an extension of the *Simulation* base class, defining a type of environment, a corresponding extension of the *SimObj* class must exist. In fact, implementing a specific simulation environment in LabLOVE consists of extending these two classes.

As shown in the pseudo code, a generic world update is performed by calling the Simulation *onCycle()* function, followed by a call of the *process()* function in each object in the world. In a physical simulated world, for example, these functions may update the position, speed and acceleration of each object and apply forces. Then every agent performs a decision cycle, where the *perceive()*, *compute()* and *act()* functions are called in sequence. This first feeds sensory information to the agent, the second allows its brain to perform its computation cycle and the third executes action based on the decision produced by the brain. The *compute()* function will perform a gridbrain computation cycle.

During the simulation cycle, the current visualization is drawn. This is done by calling the *drawBeforeObjects()* and *drawAfterObjects()* functions on the *Simulation* object, and the *draw()* object function so that each object draws itself. These function are also to be implemented in class extensions for specific environments.

The cycle ends by incrementing the simulation time.

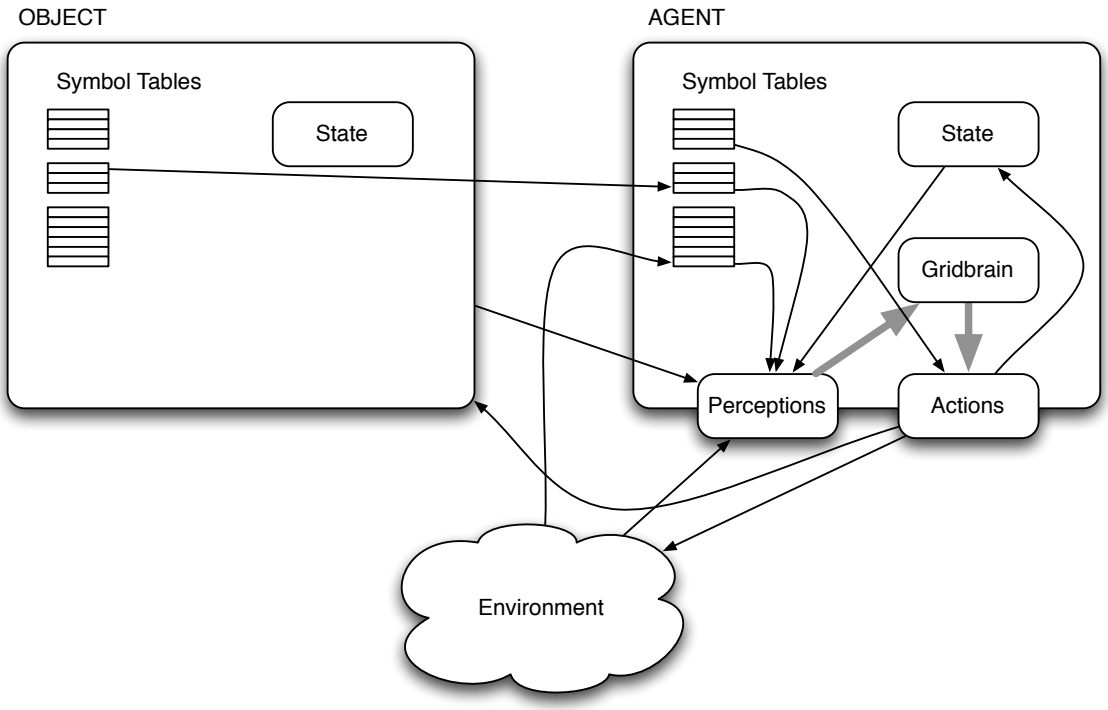

Figure 6.1: Object/Agent Model.

6.2 The Object/Agent Model

In figure 6.1 we present the object/agent model used in LabLOVE.

Objects contain two types of data: symbols and a state structure. The state structure contains information that changes during the lifetime of the object and pertains to current conditions of the object in relation to its environment. For example, in a two dimensional environment the state will include a (x, y) current position tuple, while in a three dimensional environment the state will include a (x, y, z) current position tuple. Other examples of information that can exist in the state are energy level, rotation or current speed. The data structure to use for the state is dependent of the specific needs of the environment being modeled.

Symbols contain information about the object that does not change during its lifetime. They represent object features, analogous to genetic traits. Examples of this could be the agent color or shape. Symbols have types, similarly to variables in computer languages. There is a *Symbol* abstract base class in LabLOVE that can be extended to implement

new symbol types as needed.

Symbols are organized in tables, according to the class of feature they relate too. The symbols in a table must be all of the same type. The table structure is predefined for each species of objects in an experiment.

In the LabLOVE model, and agent is an extension of an object that contains a gridbrain, as well as the ability to form perceptions and produce actions. These perception are used as sensory information fed to the gridbrain, according to its corresponding sensory channel. The gridbrain outputs trigger the actions to be performed by the agent. In the case of agents, symbols may be associated with actions and perceptions.

6.2.1 Actions and Perceptions

As seen in figure 6.1, there are several ways in which an agent may form a perception. Sensory information may be received directly from the environment, from other objects or from the agent's own internal state. The first two types of sensory information may be preprocessed by comparison with one of the agent's symbols. In any case, all the sensory information that is fed to the gridbrain takes the form of floating point values. Specific sensory mechanism are implemented in simulation environments.

Sensory information received from the agent's internal state is a floating point representation of one aspect of this state, for example energy level. Information received from other objects is a floating point representation of some aspect of this agent's current state or features.

An example of information received from the state of another object could be distance. The simulation environment uses the position of the agent and the perceived object to calculate their distance, and this value is used by the agent to form a distance perception. Other examples could include relative speed or orientation. These kinds of perception do not have to be relative to the agent. For example, we could allow the agent to perceive the energy level of visible objects.

Another type of sensory information that can be perceived from other objects is relative to their symbol-represented features. In this case a mechanism is used where a symbol of the perceived object is compared to an internal symbol of the agent. This comparison

results in a floating point value, and thus, the sensory data is obtained. In LabLOVE we define two types of symbol-to-symbol comparisons: *equality* and *distance*. The first returns 1 if the symbols are equal, otherwise it returns 0. The second applies a distance metric which is appropriate for the symbol type. These comparisons are defined as abstract functions in the base *Symbol* class, to be implemented in extensions of this class.

To illustrate, we will take the example of the RGB symbol type, representing colors by their red, green and blue components. In this type, each component is a byte. Let us consider two of these symbols, with the component tuples (r_1, g_1, b_1) and (r_2, g_2, b_2). The equality comparison returns 1 if $(r_1 = r_2) \wedge (g_1 = g_2) \wedge (b_1 = b_2)$, 0 otherwise. The distance comparison returns the cartesian distance distance between the tuples, using the component as coordinates, divided by the greatest distance possible ($\sqrt{3 \cdot 255^2}$). This way it returns a value in the $[0, 1]$ interval:

$$d = \frac{\sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}}{\sqrt{3 \cdot 255^2}}$$

Another example could be a fixed-size binary string symbol. In this case the equality comparison checks if all bits in corresponding positions are equal. The distance comparison uses Hamming distance divided by the size of the string, which is the maximum possible Hamming distance, again producing a value in the $[0, 1]$ interval.

Sensory information can also be received directly from the environment, or through symbol comparison. Example of sensory data received directly could be temperature or the intensity of a smell. Symbols may be also used to model environmental traits, for example the color of terrain patches. An agent could perceive the color of the terrain patch it currently occupies by comparison with an internal color symbol.

Actions may also be associated with symbols. In this case the symbol is used to help determine the outcome of the action. For example, consider an agent that is capable of painting a mark on the terrain. The paint action can be associated with an internal color symbol, to determine the color of the painted mark. This agent species may even evolve several paint actions with different colors.

To give a full-circle example, consider agents that are capable of communicating by emitting sounds. The sound is represented by its frequency, intensity and duration. A

specific symbol type is defined for sounds, that contains the tuple (f, i, d) to store these values. Agents have a symbol table for sounds. Actions that emit a sound are associated with a sound symbol. When triggered, they emit a sound with the characteristics determined by the symbol. This is an action that directly affects the environment. The simulation delivers this sound to all objects within range. Other agents can sense sounds through a specific perception that is also associated with sound symbols. Sound sensory information is generated by applying the mechanism formerly described, and fed to the sound sensory channel of the gridbrain.

The connection between LabLOVE agents actions and perceptions and the gridbrain input/output system is implemented by extending the *IN* and *OUT* gridbrain component classes, to respectively *PER* and *ACT*. This extension includes fields to store possible associations between the component and a symbol. A component to symbol association is represented by the integer fields *origin_table*, *origin_symbol*, *target_table* and *target_symbol*. These fields store the respective table and symbol identifiers, or 0 is no association exists.

When forming perceptions or producing actions, the gridbrain of the agent is scanned for components of respectively the *PER* and *ACT* types. This way the simulation knows which perceptions to form and actions to produce for each agent.

In figure 6.1 we show perception/action relationships between an agent and an object, but they could likewise exist between an agent and another agent, because, as already stated, the agent is a special case of an object.

6.2.2 Agents and Genetic Operators

In LabLOVE, evolving populations are constituted by agents. It is thus important to detail how genetic operators work at this level. The offspring gridbrain is generated by applying the genetic operators detailed in chapter 4, according to their defined probabilities of occurrence. Then, the symbol tables for the offspring are populated, by importing symbols from the parent or parents. Only symbols that are used by the child are imported. A symbol is considered to be in use in any of the three following cases:

- The symbol represents an agent feature;

- The symbol is associated with at least one action or perception component of the agent's gridbrain;

- The symbol is explicitly marked as used, by an *used flag*.

The last case is related to the symbol acquisition mechanism, that we will describe in the following section.

Symbols that exist in both parents are only imported once.

6.2.3 Symbol Acquisition and Generation

Two mechanisms are defined to expand the symbol tables of agents: *symbol acquisition* and *symbol generation*. The purpose of these mechanism is to allow the gridbrain's input and output interfaces to be adaptively complexified, according to the demands of the environment. Symbols that are added to agents' symbol tables become available for association with action and perception components, through the action of the genetic operators.

Symbol acquisition consists of, during the lifetime of an agent, adding symbols that are perceived in the environment to the agent's tables. Let us suppose that an agent has a table for symbols that represent object colors. This table is initialized with one symbol that represents the color of the agent itself. During its lifetime, the agent may perceive objects that have different colors, represented by different color symbols. Through the acquisition mechanism, these color symbols are copied to the agent's color table. They are propagated to the agent's child and become available for association with action and perception components on the child's gridbrain. To this end, when a symbol is acquired, it is explicitly marked as used, by setting its *used flag*. When a new agent is sent to the environment, its symbol's *used flags* are reset. If an agent acquires a symbol that is present in its table because it had already been acquired by one of its parents, the symbol's *used flag* is set, so that the symbol again propagates to the offspring.

Symbol generation is a mechanism that creates new symbols that may not yet exist in any table of any object in the simulation. It consists of generating a random symbol and adding it to a table before the genetic operators are applied to the agent. This way, the genetic operators have the opportunity of incorporating the new symbol in the gridbrain's

input and output interfaces. For example, considering agents that communicate through messages represented by symbols, this allows for different types of messages to emerge.

These mechanisms work in conjunction with the inclusion of action and perception components in the grids' component sets that are associated with origin tables, but not with specific origin symbols. Before genetic operators are applied, these components are expanded in the sets, with one instance being created in association with each symbol in the table.

The symbol acquisition and generation mechanisms may be selectively enabled in each symbol table. This is part of the initial definitions of an experiment.

6.3 Sim2D

For the purpose of experimentation with the models described in this thesis, we developed a specific simulation environment called *Sim2D*. As stated, the implementation of this environment was done by extending the *Simulation* and *SimObj* classed to define specific environmental rules, mechanisms and visualizations.

The Sim2D environment is two-dimensional, continuous and uses a simple Newtonian physics simulation.

In figure 6.2 we present a screen shot of a LabLOVE visualization using the Sim2D environment. Two species of agents can be seen, red and blue, as well as food items in green. The grey areas in front of agents represent their vision range. Large colored circles represent sounds being propagated in the environment. A red agent can be seen shooting at another agent near the center of the screen. Agents are represented as triangles and food items as squares. Although different shapes are used to distinguish object types, all objects have a circular physical body model.

6.3.1 Object Model

Objects in Sim2D are physically modeled as two-dimensional bodies with bounding circles. They exist in a limited, rectangular world with a pre-defined width and height. Their physical parameters are shown in table 6.1.

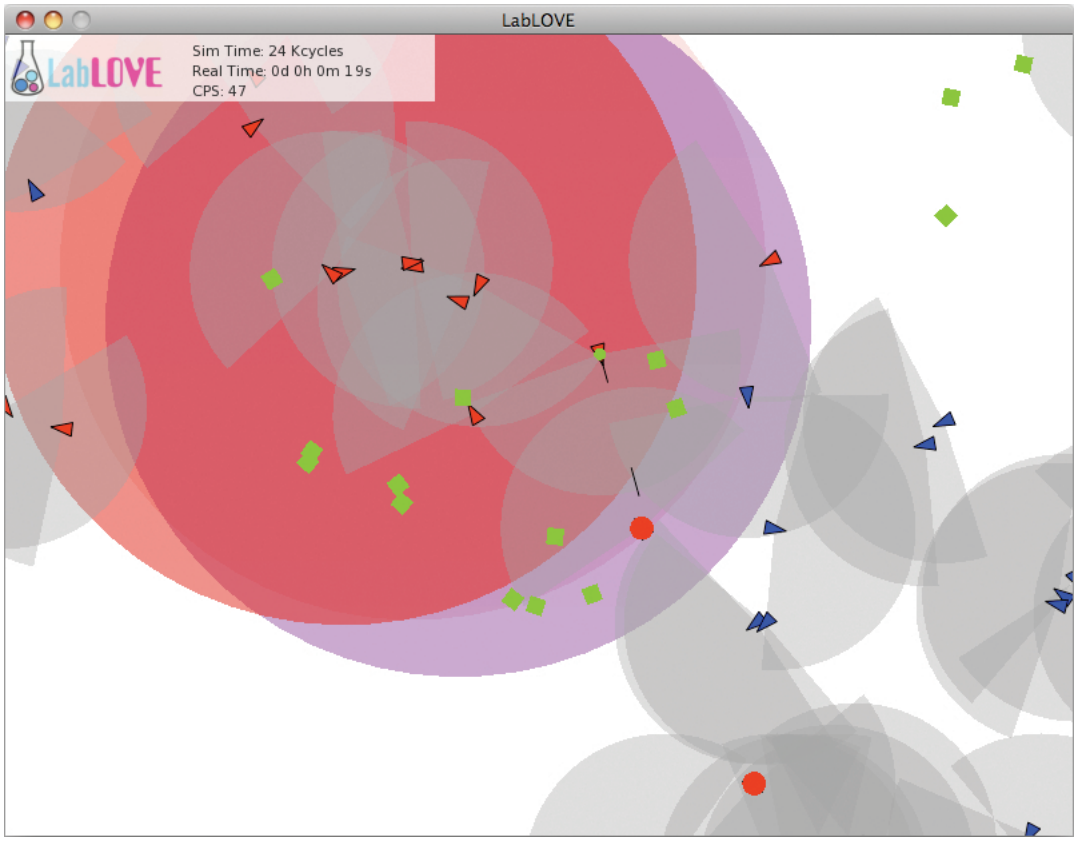

Figure 6.2: LabLOVE visualization screen shot using the Sim2D environment.

Parameter	**Description**
x (x)	x coordinate of the current position
y (y)	y coordinate of the current position
angle (a)	current angle of rotation
size (s)	radius of the body
velocityX (v_x)	x component of the current velocity vector
velocityY (v_y)	y component of the current velocity vector
velocityRot (v_r)	current rotational velocity
drag (d_l)	linear drag constant
rotDrag (d_r)	rotatinal drag constat
impulseX (I_x)	x coordinate of total impulse currently applied
impulseY (I_y)	y coordinate of total impulse currently applied
impulseRot (I_r)	total rotational impulse currently applied

Table 6.1: Physical parameters of Sim2D objects.

In each simulation cycle, the physical state of each object is updated. First, linear and rotational velocities are updated by calculating the effects of impulse and drag. The effect of impulse is calculated by using the expressions:

$$v'_x = v_x + \frac{I_x}{s}$$

$$v'_y = v_y + \frac{I_y}{s}$$

$$v'_r = v_r + \frac{I_r}{s}$$

The meaning of the variables is given in table 6.1.

After impulses are applied, impulse values are reset to 0. Impulse values result from the application of forces to the agent, as a consequence of the triggering of actions that will be detailed later. For simplicity, in Sim2D we consider the mass of an object to be equal to its size.

Drag is then applied, by way of the expressions:

$$v'_x = v_x \cdot (1 - d_l)$$

$$v'_y = v_y \cdot (1 - d_l)$$

$$v'_r = v_r \cdot (1 - d_r)$$

Finally, the object position and angle is updated:

$$x' = x + v_x$$

$$y' = y + v_y$$

$$a' = a + v_r$$

When the position is updated, it is verified if the object remains within the boundaries of the world. If boundaries are crossed, position coordinates are truncated so that the object remains within bounds.

Besides physical parameters, Sim2D objects have metabolic parameters, which model

Parameter	Description
energy (x)	current energy level
maxAgeLow (y)	low limit of maximum age interval
maxAgeHigh (a)	high limit of maximum age interval
maxAge (s)	maximum age
birthTime (v_x)	time of creation

Table 6.2: Metabolic parameters of Sim2D objects.

energy levels and age limits. These are shown in table 6.2. When an agent is created, its maximum age is chosen randomly inside the interval defined by *maxAgeLow* and *maxAgeHigh*. If the energy level reaches 0 or the maximum age is reached, the agent dies and is removed from the world.

Sim2D objects also have a color, which is determined by an RGB symbol.

We defined a set of capabilities for agents in Sim2D. In the current implementation, capabilities included are: movement, eating, shooting projectiles (that we will refer to as lasers) and communicating through sounds.

6.3.2 Vision

Agents in Sim2D can have a vision sensory channel. This channel provides the gridbrain with information about other objects that are within the agent's vision field. The vision field of agents is defined by a range (r_{vision}) and an angle (a_{vision}). An object is visible to an agent if it is at a distance inferior or equal to r_{vision} and its relative angle to the agent is within $[-\frac{a_{vision}}{2}, \frac{a_{vision}}{2}]$. The distance between two objects in Sim2D is the distance between the bounding circles that define the frontiers of their bodies.

In table 6.3, we present the visual perception types defined in Sim2D.

POSITION gives the relative position of an object to the agent. It is given by the angle ($a_{position}$) between a line connecting the center of the agent and the center of the object and a line going through the center of the agent and with the same direction as the agent. The value of POSITION is normalized by dividing $a_{position}$ by $\frac{a_{vision}}{2}$. This way, a position of -1 means that the object is on the far left of the visual field of the agent, a position of 1 means that it is on the far right and a position of 0 means that it is in the center of the vision field.

Type	Possible values
POSITION	$[-1, 1]$
ORIENTATION	$[-1, 1]$
DISTANCE	$[0, 1]$
SIZE	$]0, +\inf[$
IN_CONTACT	$\{0, 1\}$
EAT_TARGET	$\{0, 1\}$
LASER_TARGET	$\{0, 1\}$
LOF	$\{0, 1\}$
SYMDIST	$[0, 1]$
SYMEQ	$\{0, 1\}$

Table 6.3: Visual perception types in Sim2D.

ORIENTATION gives the direction the object is facing, relative to the agent's own facing direction. Being a_{agent} the current angle of the agent and a_{object} the current angle of the object, the value of this perception is given by:

$$\frac{norm(a_{object} - a_{agent})}{2\pi}$$

where $norm$ is a function that normalizes an angle in radians to the $]-\pi, \pi]$ interval.

DISTANCE is the distance between the agent and the object, divided by the agent's r_{vision}.

SIZE is the relative size of the object compared to the agent. Being s_{agent} the size of the agent and s_{object} the size of the object, it is given by $\frac{s_{object}}{s_{agent}}$.

IN_CONTACT gives a value of 1 if the agent's body is overlapping the object's body, 0 otherwise. EAT_TARGET gives a value of 1 if the object is the current agent's target for eating, 0 otherwise. LASER_TARGET gives a value of 1 if the object is the current laser target for the agent, 0 otherwise. LOF stands for *line of fire*, and gives a value of 1 if the agent is currently in the line of fire of the object, 0 otherwise.

SYMDIST and SYMEQ are perceptions that compare symbols in the agent's tables with symbols in the object's tables that represent features. SYMDIST uses distance symbol comparison, while SIMEQ uses equality symbol comparison. They could be used, for example, to provide the agent with information about the object's color.

Type	Possible values
POSITION	$[-1, 1]$
DISTANCE	$[0, 1]$
VALUE	$]0, +\inf[$
SYMDIST	$[0, 1]$
SYMEQ	$\{0, 1\}$

Table 6.4: Sound perception types in Sim2D.

6.3.3 Sounds

Agents in Sim2D have the ability to produce and perceive sounds. We did not attempt to model these sounds in a physically realistic way, as we were more interested in easiness of observation of the phenomena in LabLOVE's visualization system. This way, sounds are modeled as data structures that, when produced in a certain point, can be received by agents within the sound range pre-defined for the simulation run. This sound model is used in experimentations dealing with the emergence of synchronization and cooperation process between agents, that we will describe later.

A sound is defined by an RGB symbol and a floating point value. The floating point value can be considered the intensity of the sound, while the RGB symbol represents its unique characteristics. In a more realistic model, frequencies could be used. However, the RGB value allows us to visually display the emission of different sounds. When visualizing a LabLOVE simulation run, sound emission are represented as transparent circles centered on the sound source, with a radius equal to the sound range defined for the simulation and the color defined by the RGB symbol used in generating the sound.

In table 6.4, the sound perception types available to agents are listed.

POSITION gives the relative position of the sound sorce. It is given by the angle ($a_{position}$) between a line connecting the center of the agent and the sound source and a line going through the center of the agent and with the same direction as the agent. The value of POSITION is normalized by dividing $a_{position}$ by 2π. DISTANCE is the distance between the agent and the sound source, divided by the sound range. VALUE gives the intensity of the sound. SYMDIST and SYMEQ compare symbols in the agent's tables with the symbol that represents the sound.

Type	Description
GO	Apply a linear force in the direction the agent is facing
ROTATE	Apply a rotational force
EAT	Attempt to eat the current eat target
SPEAK	Produce a sound
FIRE	Fire a laser shot

Table 6.5: Action types in Sim2D.

6.3.4 Actions

In table 6.5 we show the list of action types available to agents.

The GO action causes a linear force to be applied to the center of the agent's body. The direction of the force depends on the signal of the input value, i, received by the component that triggered the action. It is applied in the direction the agent is facing if the input is positive, or in the opposite direction if it is negative. The intensity of the force applied is proportional to i. The intensity of the force applied to the agent is $F = i \cdot g_{const}$, where g_{const} is a constant defined in the simulation run settings. The impulse caused by this force is added to the total accumulated impulse of the agent in the current simulation cycle by the expressions:

$$I'_x = I_x + cos(a) \cdot F$$

$$I'_y = I_y + sin(a) \cdot F$$

The action has an energetic cost to the agent which is proportional to the amount of force applied. It is calculated by the expression $E = F \cdot g_{cost}$, where g_{cost} is a constant defined in the simulation run settings.

The ROTATE action causes a torque to be applied to the center of the agent's body. The intensity of the torque applied is proportional to i, and given by the expression: $T = i \cdot r_{const}$, where r_{const} is a constant defined in the simulation run settings. The rotational impulse caused by this force is added to the total accumulated rotational impulse of the agent in the current simulation cycle by the expression:

$$I'_r = I_r + T$$

The action has an energetic cost to the agent which is proportional to the amount of torque applied. It is calculated by the expression $E = T \cdot r_{cost}$, where r_{cost} is a constant defined in the simulation run settings.

The EAT action causes the agent to attempt to eat the closest overlapping object. When the action is successfully executed, it results in an energy transfer from the target object to the agent. Feeding interaction rules are defined with the aid of symbols. Objects and agents that are to participate in this type of interaction must possess two features defined by symbols: *food type* and *feed interface*. EAT action components are set to used *feed interface* symbols as origins and *food type* symbols as targets. When an EAT action is attempted, the distance symbol comparison method is used to determine the distance between both symbols, d_{feed}. If the *feed interface* symbol does not exist on the agent or the *food type* symbol does not exist on the target, the action fails. If the action succeeds, the target object looses all its energy, e. The energy received by the agent is determined by the expressions:

$$e_{gain} = \begin{cases} -\frac{c_{feed}+d_{feed}}{c_{feed}} \cdot e & \text{if } d_{feed} < c_{feed} \\ \frac{d_{feed}-c_{feed}}{1-c_{feed}} \cdot e & \text{if } d_{feed} \geq c_{feed}, \end{cases}$$

where c_{feed} is a constant defined in the experiment run settings. This means that if the symbol distance is greater than this constant, the agent will gain energy, if it is smaller, the agent will lose energy. The greater the distance above the constant, the more energy gained. The smaller the distance below the constant, the more energy is lost. We define an object that, when eaten by an agent, causes it to lose energy, as a *poison* to that agent and one that causes the agent to gain energy as *food*.

Any symbol type may be used to define the feeding interface. This mechanism allows the experiment designer to create complex food networks.

The SPEAK action causes the emission of a sound, centered on the agent's current position. The sound emitted is represented by the origin symbol pointed to by the action

component that triggered the action. The intensity of the sound is equal to the input i received by the component. A *speak interval* is defined for the simulation run, determining the minimum number of simulation cycles between two sound emissions from the same agent. If an agent attempts to perform a speak action before this amount of simulation time elapsed, the action fails.

The FIRE action causes a laser shot to be fired from the agent, in the direction it is facing. A laser shot is modeled as a line segment, with a length of l_{laser}, traveling at a constant velocity of v_{laser} distance units per simulation cycle. The amount of damage that it is capable of doing to an object is determined by the expression:

$$l_{damage} = max(\frac{t_{lock}}{l_{interval}}, 1) \cdot s_{laser},$$

where t_{lock} is the amount of simulation cycles that the agent as spent targeting the current target object, and $l_{interval}$ and s_{laser} are predefined constants. The first one represents the minimum amount of simulation cycles that and agent has to be locked on a target for the effectiveness of the shot to be maximum, and the second is a constant laser strength factor.

When a laser shot collides with an object, it is added to a list of received shots on this object. In each simulation cycle, when objects update their states, they run through their list of received shots. Shots that are older than t_{laser} simulation cycles are removed from the list. The other have their damage value added to the object's total damage for that cycle. The age of a laser shot is equal to the current simulation cycle number minus the simulation cycle number in which it was created. If the total damage to an object is greater or equal than its current energy, the object is destroyed, otherwise nothing happens.

6.3.5 Application

In the following chapters, we will present and discuss experimental results obtained with the Sim2D environment. The movement, eating, shooting and sound emitting capabilities of agents were explored to formulate different challenges for the evolutionary process. By establishing fitness functions and environmental conditions, we devised scenarios where agents evolve to survive by eating, distinguish food from poison, communicate, and coop-

erate at shooting at moving targets and each other. The Sim2D provided us with a base platform upon which these scenarios were created with little configuration needed.

Chapter 7

Genetic Operators and Parameters Benchmark

In this chapter we present and discuss results from a set of simulation runs aimed at testing the contribution of the several genetic operators to evolving gridbrains, as well as the impact of SEEA parameters on the evolutionary process.

For this purpose we defined an experimental scenario called *poison*. This scenario is defined in the `experiments/poison.lua` file that is included in the LabLOVE distribution. The Sim2D simulation environment is used. The world is populated with two types of objects: an agent species and an object species that serves as food for the agents. The agents are provided with an action set that allows them to move around the world and eat the food objects. Some of the food objects are poisonous, while others are nutritive. In this scenario, evolutionary pressure is created so that the agents evolve strategies to find and eat nutritive food objects, while avoiding poisonous objects and not wasting energy.

Sets of experiment runs where executed for ranges of simulation parameters, namely genetic operator probabilities and SEEA ageing factor and buffer size.

7.1 Experimental Setup

The eating interface is defined by floating point symbols. These symbols consist of a floating point value x, in an interval of possible values, $[x_{min}, x_{max}]$. The equality comparison

Grid	Type	Components
Vision (Alpha)	Computational	AND, NOT, SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ, ZERO, MAX, MIN, AVG, DMUL, SEL, MEM
	Perceptions	POSITION, DISTANCE, EAT_TARGET, SYMDIST(food)
Beta	Computational	AND, NOT, SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ, ZERO, CLK, DMUL, MEM, TMEM
	Actions	GO, ROTATE, EAT

Table 7.1: Grid component sets of poison agents.

between two symbols of this type consists of verifying if the two values are equal. The distance comparison uses the expression:

$$d = \frac{|x' - x|}{x_{max} - x_{min}}$$

In this scenario we use $x_{min} = 0$ and $x_{max} = 1$. Food objects have a *food type* symbol. Its value is randomly generated using an uniform distribution in the $[0, 1]$ interval each time a food object is created. Agents have a *feed interface* symbol that is set to 1, and the environment feed center (c_{feed}) is set to 0.3. This way, there is a 30% probability that a given food object is poisonous to agents, and a 70% probability that it is nutritive. These values were chosen empirically to provide an interesting but not overly hostile environment to the agents. There are various degrees of poisonous or nutritive values. A food item with a food type symbol with the value 0 will be the most poisonous, while one with the value of 1 will be the most nutritive.

Agent gridbrains are defined to have two grids: one alpha grid for vision and one beta grid. In table 7.1 we present the grid component sets used in this scenario. The computational components are the ones described in section 4.2. Aggregator components are only used in the vision grid, because, as described, they are only useful in alpha grids. Clock components are only used in the beta grid. Since the alpha grid is evaluated a variable number of times per simulation cycle, clocks in this type of grid produce meaningless out-

Parameter	Value
World width	1000
World height	1000
Number of agents	25
Number of food items	50
Agent size (s_{agent})	10
Food item size (s_{food})	10
Initial agent energy	1.0
Initial food item energy	1.0
Agent maximum age	5000
Food maximum age	5000
Go cost (g_{cost})	0.005
Rotate cost (r_{cost})	0.005
Go force constant (g_{const})	0.3
Rotate force constant (r_{const})	0.006
Drag constant (d_l)	0.05
Rotational drag constant (d_r)	0.05
Feed center (c_{feed})	0.25
Vision range (r_{vision})	150
Vision angle (a_{vision})	170^0

Table 7.2: Poison experiment parameters.

put. The beta grid is always evaluated one time per simulation cycle, so we reserve clock components for this type of grid.

The vision grid includes a SYMDIST perception component that compares the agent's feed interface symbol with the visible object's food type. This way, the agent may know the nutritive value of any visible object, if its alpha grid includes that perception component. The remaining action and perception components were described in section 6.3.

In table 7.2 we present the values used to parametrize the experiment. Experiment runs are performed with a duration of $2{\times}10^5$K simulation cycles.

These values where chosen using an empiric approach, with the goal of defining an environment that is not too challenging nor too easy for the evolutionary process, and also taking into consideration computation power constraints. For example, the computational cost of maintaining a simulation increases with the number of objects/agents in the world, as well as with their density. Too long lifetimes make evolution slow, while too short ones prevent agents from performing actions that can increase their fitness. Many other sets of parameters could be used, but this one was found to work well for our benchmarking

purposes.

When new elements are generated, both agents and objects, they are placed in a random location in the world, facing a random direction.

Agent fitnesses are calculated using the expression:

$$f = \sum_{j=0}^{100j \leq t_{max}} max(0, e(100j) - e_0),$$

where e_0 is the initial energy of the agent, $e(t)$ is the energy of the agent at age t and t_{max} is the maximum age that the agent reached. This expression performs a summation of samples of the energy level of the agent, at intervals of 100 simulation cycles. If the energy level of the agent at a sample point t is greater than e_0, its fitness is incremented by $e(t)$, otherwise it remains unaltered. The purpose of this fitness function is to simultaneously reward agents for gathering energy, conserving energy and living longer.

7.2 Results

In this section we present the experimental results obtained for this scenario. The methodology used was to perform sets of 30 simulation runs, each set varying one or more evolutionary parameters. This way we are able to access the impact of the various evolutionary mechanism proposed on the evolutionary process. Data is collected by logging the average of several metrics for all the agents dead in 100K simulation cycle intervals. We use the agent's final fitness as a performance metric, and simple gridbrain bloat metrics include number of connections and fitness per number of connections.

The statistical significances null hypothesis of no differences was determined with Kruskal-Wallis ANOVAs at $p = 0.01$. For experiment sets where parameterization was found to be significantly relevant, pairwise Wilcoxon rank sum tests with Holm's p-value adjustment method where applied. Non-parametric ANOVAs and pairwise tests were used because the data is not guaranteed to follow a normal distribution.

7.2.1 Mutation Operators

In this set of experiments we test the impact of mutation operators.

In the first experiment we vary all the connection level mutation operators at the same time. We set the change inactive component probability to $p_c = 0.1$, the change parameter to $p_p = 0$, the recombination probability to $p_{rec} = 1$, the ageing factor to $a = 0.1$ and the agent species buffer size to $s_{buf} = 100$. We chose the change inactive component at a high rate because it is a non-destructive neutral mutation. We chose to disable the change parameter mutation for now, because we empirically expected it to not have a great contribution to evolution and wanted to keep the experiment as simple as possible. In the same vein of simplicity, we work with a recombination probability of 1 for now. The ageing factor and buffer sizes where chosen to use values that appeared empirically reasonable.

In each set of runs we use the same value x for the add connection probability, $p_a = x$, remove connection probability, $p_r = x$, split connection probability, $p_s = x$ and join connections probability $p_j = x$. We perform sets of runs with different x values.

The results of this experiment can be observed in figure 7.1 and table 7.3. Box plots use average values for the last 100K cycle period of each simulation run. The best median fitnesses are obtained for the sets $x = 0.01$ and $x = 0.02$, with a very slight advantage to $x = 0.01$. The variation in mutation rates was found to cause a significant difference in final average fitnesses, with a p-value of 6.784×10^{-9}. Pairwise tests show that $x \geq 0.005$ significantly outperforms lower x values, and $x = 0.02$ significantly outperforms the higher value of $x = 0.05$.

It can also be observed that the number of final active gridbrain connections tends to be higher for sets with higher final average fitness. We used Spearman's ranks correlation coefficient, which is a non-parametric measure of correlation, to calculate the correlation between the average final fitness and the average final number of gridbrain connections in all the runs in this experiment. It was found that $\rho = 0.778024$.

In the experiment corresponding to the results shown in figure 7.2 and table 7.4, we tested the impact of the change inactive component mutation operator. For this purpose, we performed simulation runs with $p_a = 0.01, p_r = 0.01, p_s = 0.01, p_j = 0.01, p_c = 0.2, p_p = 0, a = 0.1, p_{rec} = 1, s_{buf} = 100$, while varying the p_c value. As can be observed the best median was obtained for $p_c = 0.2$. Sets where found to be significantly different

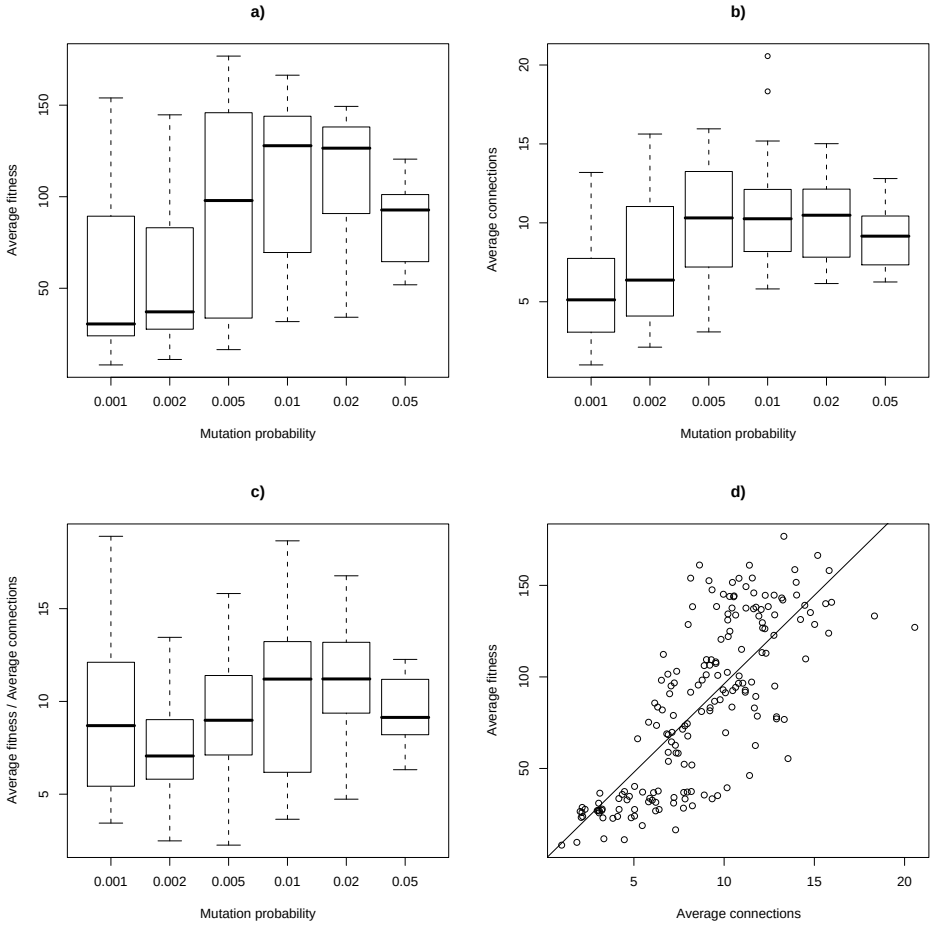

Figure 7.1: Results of experiments with various mutation probabilities: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections, with best fit line.

	0.001	*0.002*	*0.005*	*0.01*	*0.02*
0.002	1.00000	-	-	-	-
0.005	**0.00780**	0.02018	-	-	-
0.01	**0.00016**	**0.00025**	1.00000	-	-
0.02	$\mathbf{3.4 \times 10^{-05}}$	$\mathbf{3.9 \times 10^{-06}}$	1.00000	1.00000	-
0.05	**0.00519**	**0.00164**	1.00000	0.23116	**0.00042**

Table 7.3: Pairwise comparisons of fitness samples using the Wilcoxon rank sum test. The p-value for each pair of mutation probability values is presented.

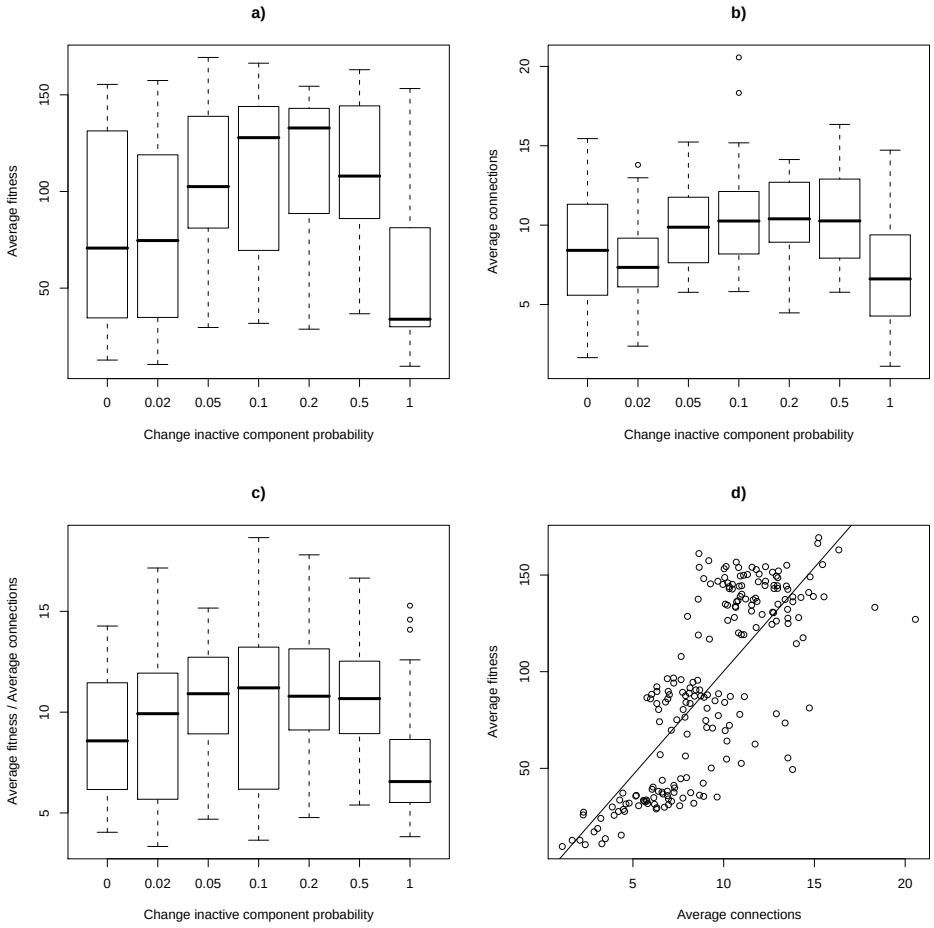

Figure 7.2: Results of experiments with various change inactive component probabilities: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections, with best fit line.

	0	*0.02*	*0.05*	*0.1*	*0.2*	*0.5*
0.02	1.00000	-	-	-	-	-
0.05	0.35135	0.24833	-	-	-	
0.1	0.35135	0.24833	1.00000	-	-	-
0.2	0.19110	0.07249	1.00000	1.00000	-	-
0.5	0.08250	0.04420	1.00000	1.00000	1.00000	-
1	0.55004	0.63782	**0.00130**	**0.00049**	**0.00030**	$\mathbf{5.6\times10^{-05}}$

Table 7.4: Pairwise comparisons of fitness samples using the Wilcoxon rank sum test. The p-value for each pair of change inactive component probability values is presented.

in terms of fitnesses, with a p-value of 2.019×10^{-6}. For this entire experiment, a high correlation between fitness and number of connections in the gridbrain was found, with $\rho = 0.7596641$. Pairwise tests revealed $p_c = 1$ to perform significantly worse than any of $p_c \in \{0.05, 0.1, 0.2, 0.5\}$. There were no significant differences in performance between $p_c = 1$ and $p_c = 0$ or $p_c = 0.02$.

In the experiment corresponding to the results shown in figure 7.3 and table 7.5, we compared the two modalities of the change component mutation operator: *change any* and *change inactive*. We performed simulation runs with $p_a = 0.01, p_r = 0.01, p_s = 0.01, p_j = 0.01, p_p = 0, p_{rec} = 1, a = 0.1, s_{buf} = 100$, with various p_c values for *change any*, and compared it with the best value previously found for *change inactive*: $p_c = 0.2$, shown in the plots as *in_0.2*. As can be observed *change inactive* outperformed all the values tested for *change any*. Sets where found to be significantly different in terms of fitnesses, with a p-value of 2.2×10^{-16}. For this entire experiment, a high correlation between fitness and number of connections in the gridbrain was found, with $\rho = 0.9354398$. Pairwise tests show that the change inactive modality performs significantly better than the change any modality with $p_c \geq 0.05$. For lower values of p_c on change any, no significant differences were found. The three lower values tested for p_c ($0.01, 0.02$ and 0.05) where shown to significantly outperform the higher values of 0.1 and 0.2.

In the experiment corresponding to the results shown in figure 7.4 and table 7.6, we tested the impact of the add/remove mutation operators. We performed simulation runs with $p_s = 0.01, p_j = 0.01, p_c = 0.2, p_p = 0, p_{rec} = 1, a = 0.1, s_{buf} = 100$, and $p_a = p_r = x$. We run sets of experiences with various x values. As can be observed the best median was obtained for $x = 0.01$. Sets where found to be significantly different in terms of fitnesses, with a p-value of 3.958×10^{-8}. For this entire experiment, a high correlation between fitness and number of connections in the gridbrain was found, with $\rho = 0.6953678$. Pairwise tests show that $x \geq 0.005$ significantly outperforms $x = 0$.

In the experiment corresponding to the results shown in figure 7.5 and table 7.7, we tested the impact of the split/join mutation operators. We performed simulation runs with $p_a = 0.01, p_r = 0.01, p_c = 0.2, p_p = 0, p_{rec} = 1, a = 0.1, s_{buf} = 100$, and $p_s = p_j = x$. We run sets of experiences with various x values. As can be observed the best median

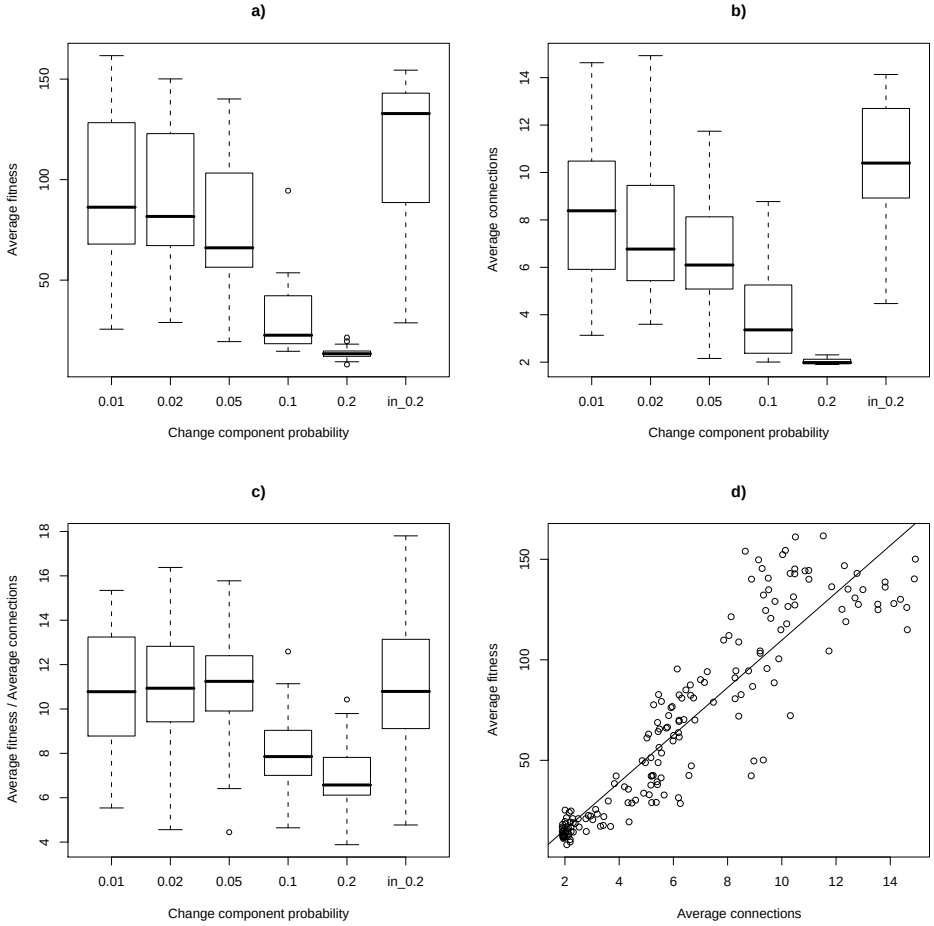

Figure 7.3: Results of experiments with various change component probabilities and comparison with the change inactive component operator, indicated as "in_0.2": a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections, with best fit line.

	0.01	*0.02*	*0.05*	*0.1*	*0.2*
0.02	0.49944	-	-	-	-
0.05	0.04218	0.09627	-	-	-
0.1	$\mathbf{5.6\times10^{-09}}$	$\mathbf{1.3\times10^{-07}}$	$\mathbf{1.6\times10^{-06}}$	-	-
0.2	$\mathbf{1.9\times10^{-15}}$	$\mathbf{1.9\times10^{-15}}$	$\mathbf{5.6\times10^{-15}}$	$\mathbf{1.9\times10^{-10}}$	-
in_0.2	0.08758	0.04218	**0.00020**	$\mathbf{4.3\times10^{-10}}$	$\mathbf{5.1\times10^{-16}}$

Table 7.5: Pairwise comparisons of fitness samples using the Wilcoxon rank sum test. The p-value for each pair of change active/inactive component probability values is presented.

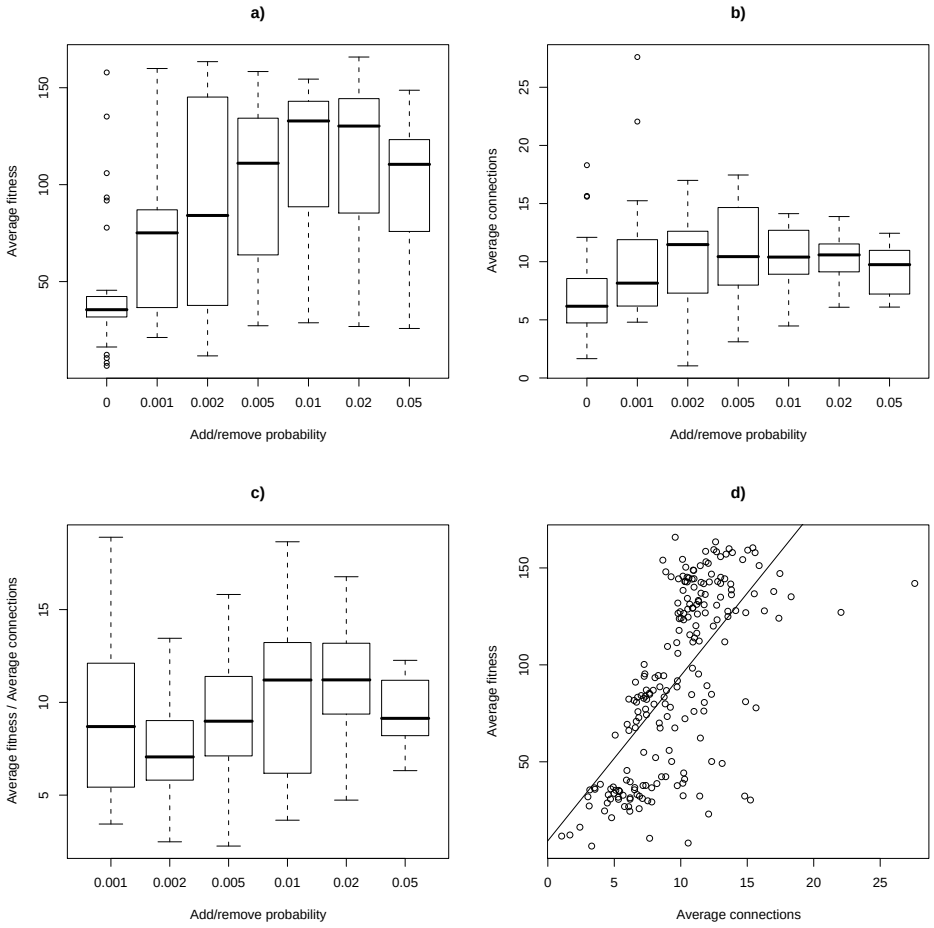

Figure 7.4: Results of experiments with various add/remove connection probabilities: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections, with best fit line.

	0	*0.001*	*0.002*	*0.005*	*0.01*	*0.02*
0.001	0.08971	-	-	-	-	-
0.002	0.01193	1.00000	-	-	-	-
0.005	**0.00059**	0.49037	1.00000	-	-	-
0.01	$\mathbf{1.1\times10^{-05}}$	0.02692	1.00000	1.00000	-	-
0.02	$\mathbf{2.4\times10^{-07}}$	**0.00640**	0.79002	0.79002	1.00000	-
0.05	$\mathbf{1.1\times10^{-05}}$	0.49037	1.00000	1.00000	0.08248	0.03640

Table 7.6: Pairwise comparisons of fitness samples using the Wilcoxon rank sum test. The p-value for each pair of add/remove connection probability values is presented.

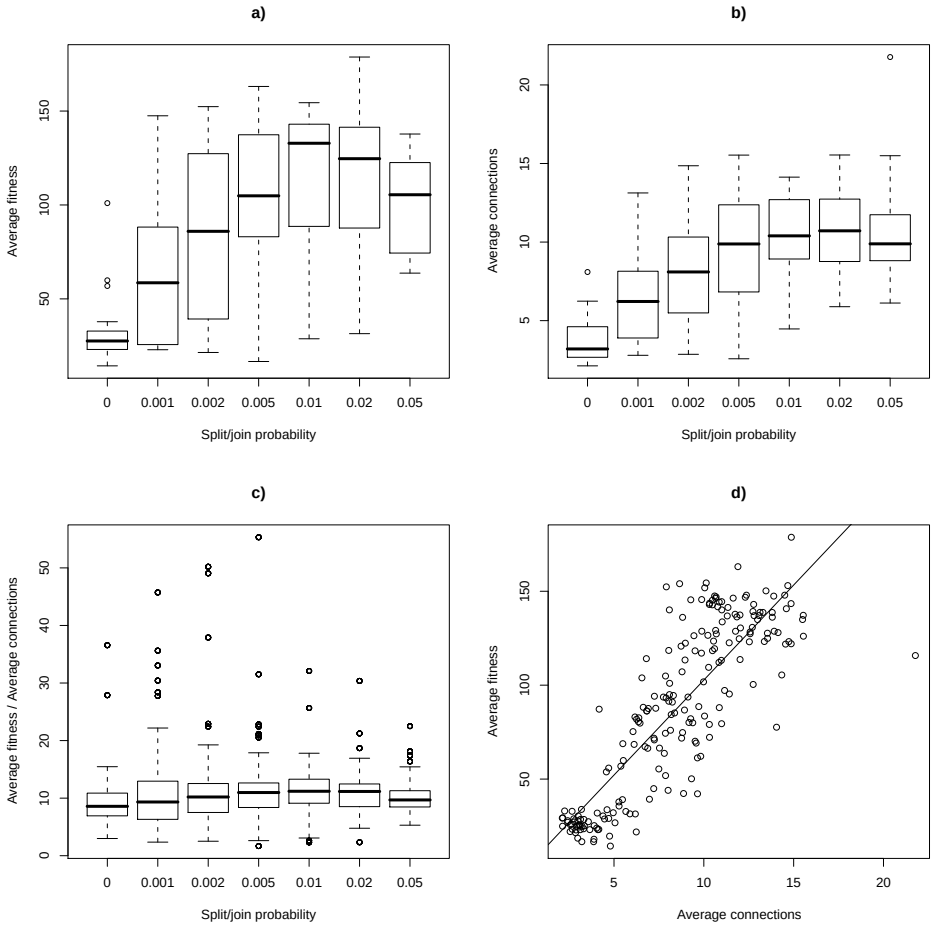

Figure 7.5: Results of experiments with various split/join connection probabilities: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections, with best fit line.

	0	*0.001*	*0.002*	*0.005*	*0.01*	*0.02*
0.001	**0.00596**	-	-	-	-	-
0.002	$\mathbf{5.3\times10^{-06}}$	0.42781	-	-	-	-
0.005	$\mathbf{9.8\times10^{-08}}$	0.01134	0.97143	-	-	-
0.01	$\mathbf{5.7\times10^{-11}}$	**0.00049**	0.04744	0.97143	-	-
0.02	$\mathbf{7.1\times10^{-13}}$	**0.00013**	0.08774	1.00000	1.00000	-
0.05	$\mathbf{2.6\times10^{-13}}$	**0.00969**	1.00000	1.00000	0.09758	0.18854

Table 7.7: Pairwise comparisons of fitness samples using the Wilcoxon rank sum test. The p-value for each pair of split/join connection probability values is presented.

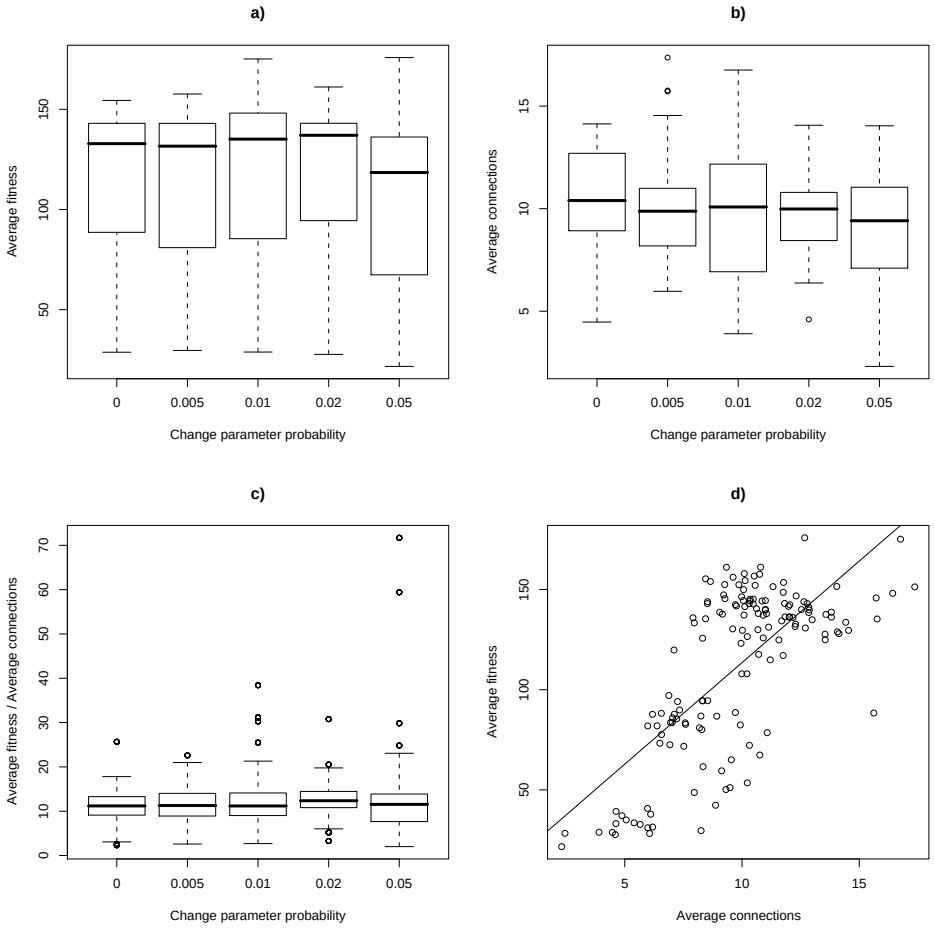

Figure 7.6: Results of experiments with various change parameter probabilities: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections, with best fit line.

was obtained for $x = 0.01$. Sets where found to be significantly different in terms of fitnesses, with a p-value of 9.466×10^{-15}. For this entire experiment, a high correlation between fitness and number of connections in the gridbrain was found, with $\rho = 0.838462$. Pairwise tests show that $x > 0$ significantly outperforms $x = 0$.

In the experiment corresponding to the results shown in figure 7.6, we tested the impact of the change parameter mutation operator. We performed simulation runs with $p_a = 0.01, p_r = 0.01, p_s = 0.01, p_j = 0.01, p_c = 0.2, \delta_p = 1, p_{rec} = 1, a = 0.1, s_{buf} = 100$, and various p_p values. There where no significant differences in terms of fitnesses, with a p-value of 0.7293. For this entire experiment, a high correlation between fitness and

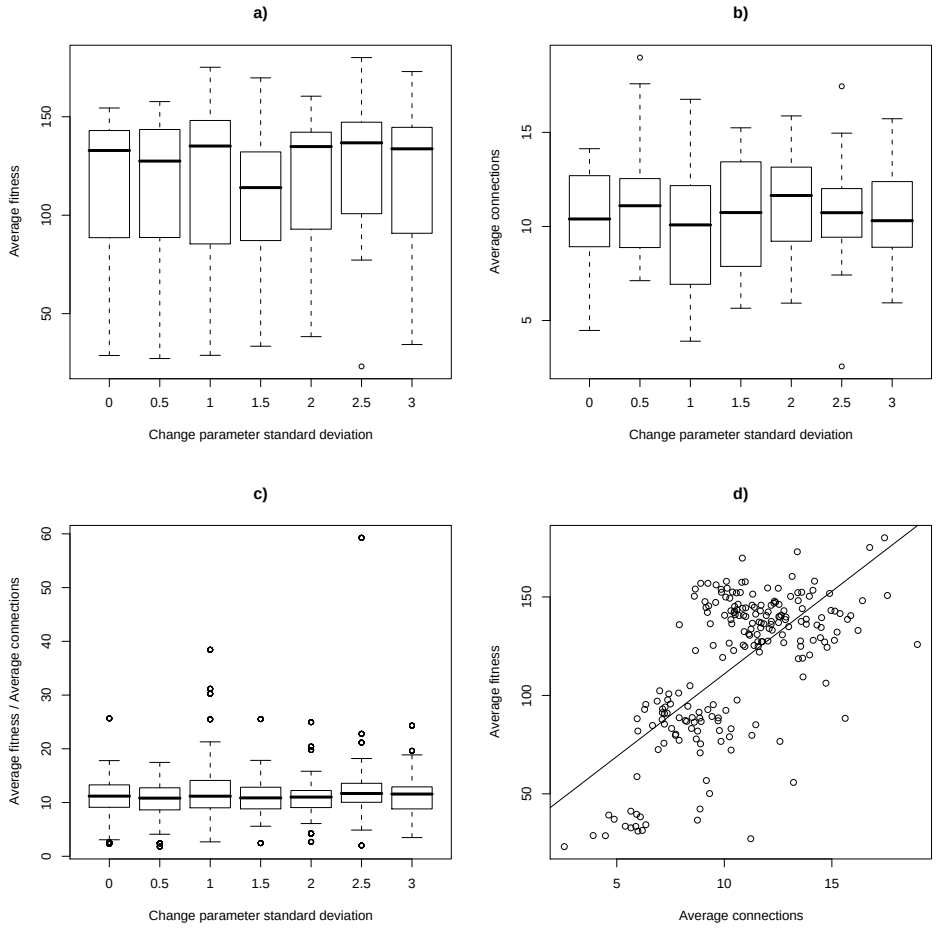

Figure 7.7: Results of experiments with various change parameter standard deviations: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections, with best fit line.

number of connections in the gridbrain was found, with $\rho = 0.6064714$.

In the experiment corresponding to the results shown in figure 7.7, we tested the impact of the standard deviation of the change parameter mutation operator. We performed simulation runs with $p_a = 0.01, p_r = 0.01, p_s = 0.01, p_j = 0.01, p_p = 0.01, p_c = 0.2, \delta_p = 1, p_{rec} = 1, a = 0.1, s_{buf} = 100$, and various δ_p values. There where no significant differences in terms of fitnesses, with a p-value of 0.4381. For this entire experiment, a correlation between fitness and number of connections in the gridbrain was found, with $\rho = 0.5534466$.

In the experiment corresponding to the results shown inf figure 7.8 and table 7.8,

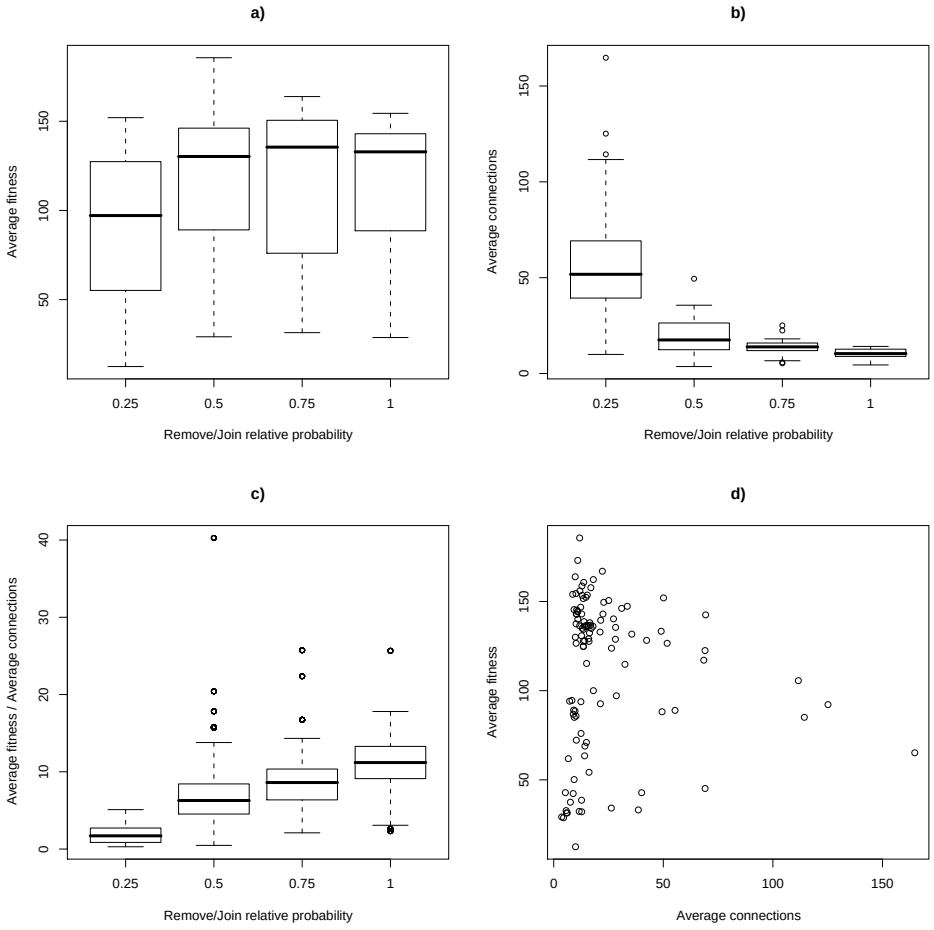

Figure 7.8: Results of experiments with various unbalanced remove/join probabilities: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections.

	0.25	*0.5*	*0.75*
0.5	$\mathbf{2.0\times10^{-06}}$	-	-
0.75	$\mathbf{2.2\times10^{-07}}$	**0.00960**	-
1	$\mathbf{7.4\times10^{-09}}$	$\mathbf{4.3\times10^{-06}}$	**0.00048**

Table 7.8: Pairwise comparisons of number of connections samples using the Wilcoxon rank sum test. The p-value for each pair of remove/joint to add/split probability ratios is presented.

we test the impact of the remove connection and join connections operators in controlling bloat. For this purpose we performed experiments with $p_a = 0.01, p_s = 0.01, p_c = 0.2, p_p = 0, p_{rec} = 1, a = 0.1, s_{buf} = 100$, and $p_r = p_j = x \cdot p_a$, where x is the probability factor of remove/join relative to the probability of add/split. Various factors where tested. In terms of fitness, no significant differences were found, with a p-value of 0.1455. However, the lower the factor, the more bloat was obtained. This could be observed both in the number of connections and in fitness per connection values, the first with a p-value of 1.969×10^{-11} and the second with a p-value lower then 2.2×10^{-16}. For this entire experiment, a low correlation between fitness and number of connections in the gridbrain was found, with $\rho = 0.1358262$. Pairwise tests show that bloat significantly increases as x decreases, in all cases covered by the experiment.

7.2.2 Recombination

In the experiment corresponding to the results shown in figure 7.9 and table 7.9, we tested the impact of the recombination operator. We performed simulation runs with $p_a = 0.01, p_r = 0.01, p_s = 0.01, p_p = 0, p_c = 0.2, a = 0.1, s_{buf} = 100$, and various p_{rec} values. There where no significant differences in terms of fitnesses, with a p-value of 0.1996. However, the average number of connections per gridbrain was higher for lower recombination probabilities, and the average fitness per number of connections was lower for lower recombination probabilities. Both these results were shown to be significant, with both p-values lower than 2.2×10^{-16}. For this entire experiment, a low correlation between fitness and number of connections in the gridbrain was found, with $\rho = 0.2859944$. Pairwise tests show that $p_r \geq 0.5$ produces significantly less bloat than $p_r = 0$.

7.2.3 SEEA parameters

In the experiment corresponding to the results shown in figure 7.10 and table 7.10, we tested the impact of the buffer size. We performed simulation runs with $p_a = 0.01, p_r = 0.01, p_s = 0.01, p_j = 0.01, p_c = 0.2, p_p = 0, p_{rec} = 1, a = 0.1$, while varying s_{buf}. As can be observed, the best medians were obtained for $s_{buf} = 100$ and $s_{buf} = 1000$. Sets where found to be significantly different in terms of fitnesses, with a p-value of 1.778×10^{-10}.

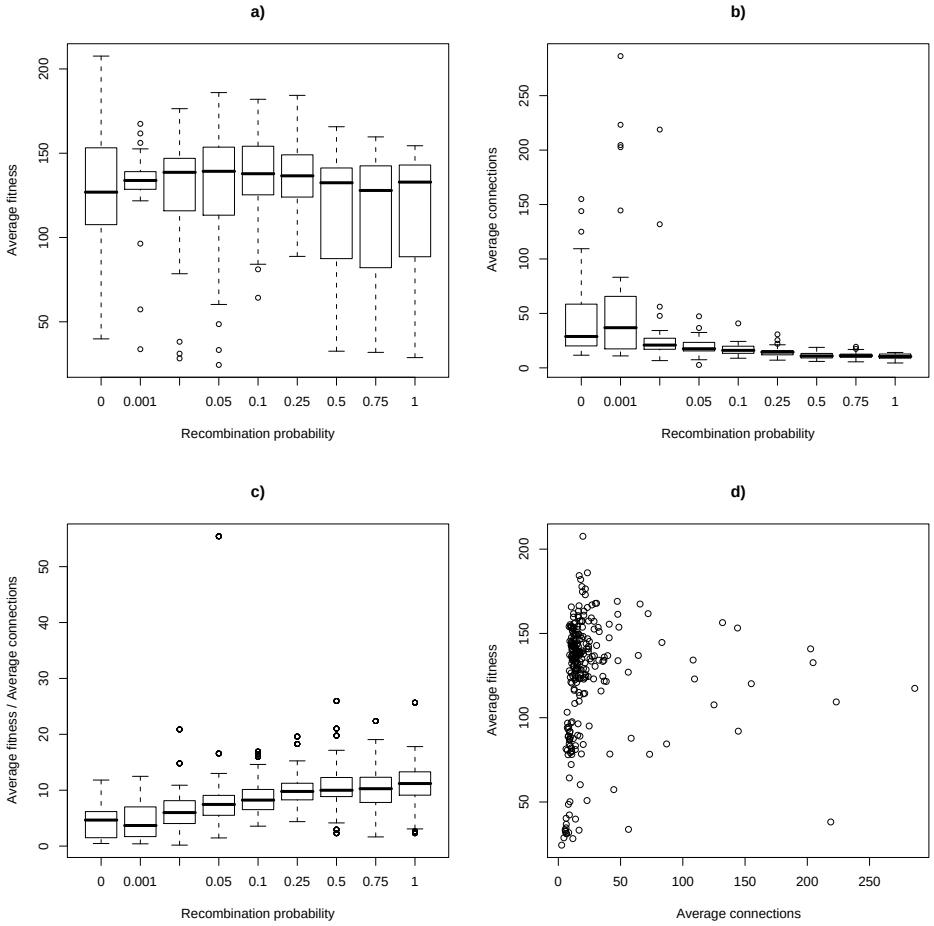

Figure 7.9: Results of experiments with various recombination probabilities: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections.

	0	*0.1*	*0.25*	*0.5*	*0.75*
0.1	0.74068	-	-	-	-
0.25	0.01932	0.80522	-	-	-
0.5	$\mathbf{3.9\times10^{-06}}$	**0.00016**	0.01775	-	-
0.75	$\mathbf{2.6\times10^{-05}}$	**0.00093**	0.05242	1.00000	-
1	$\mathbf{3.2\times10^{-07}}$	$\mathbf{7.5\times10^{-06}}$	**0.00055**	1.00000	0.85299

Table 7.9: Pairwise comparisons of number of connections samples using the Wilcoxon rank sum test. The p-values for some pairs of recombination probability values are presented.

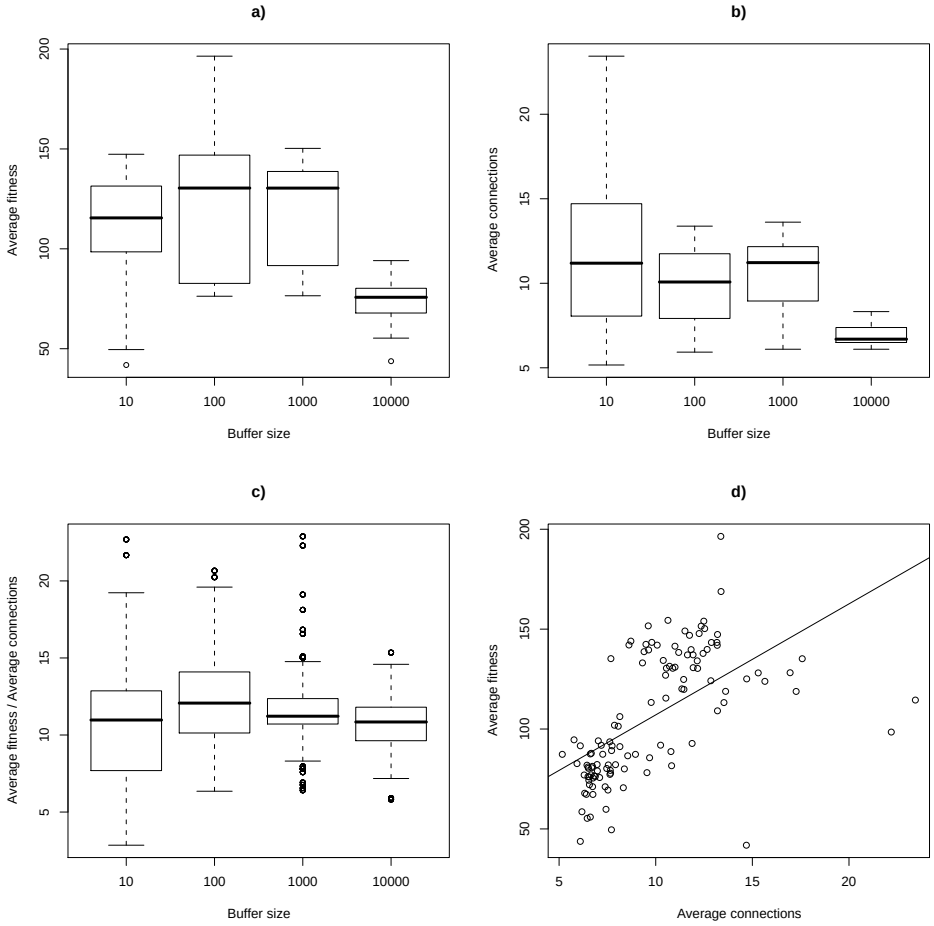

Figure 7.10: Results of experiments with various buffer sizes: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections, with best fit line.

	10	*100*	*1000*
100	0.64	-	-
1000	0.62	0.71	-
10000	$\mathbf{3.6\times10^{-08}}$	$\mathbf{1.1\times10^{-08}}$	$\mathbf{6.5\times10^{-11}}$

Table 7.10: Pairwise comparisons of fitness samples using the Wilcoxon rank sum test. The p-value for each pair of buffer size values is presented.

For this entire experiment, a high correlation between fitness and number of connections in the gridbrain was found, with $\rho = 0.6974644$. Pairwise tests show that $s_{buf} = 10000$ performs significantly worse than the lower sizes considered.

In the experiment corresponding to the results shown in figure 7.11 and table 7.11, we tested the impact of the fitness ageing factor. We performed simulation runs with $p_a = 0.01, p_r = 0.01, p_s = 0.01, p_j = 0.01, p_c = 0.2, p_p = 0, p_{rec} = 1, s_{buf} = 100$, while varying a. As can be observed, a too low or high ageing factor negatively affects the fitnesses achieved in runs. Sets where found to be significantly different in terms of fitnesses, with a p-value of 1.539×10^{-5}. For this entire experiment, a high correlation between fitness and number of connections in the gridbrain was found, with $\rho = 0.7181274$. Pairwise tests show that $0.1 \leq a \leq 0.5$ performs significantly better than the lowest ageing factor used, $a = 0.02$.

In the experiment corresponding to the results shown in figure 7.12 we tested different selection methods for the SEEA algorithm. These methods are applied when choosing agents from the species buffer for reproduction, and were described in section 5.3. So far, we have used the *simple* method in all experiments. In these experiments we compare the *simple* method against *tournament of two* and *roulette*.

As can be observed, the *simple* method outperformed the other two in terms of fitness, but only with borderline significance, at a p-value of 0.01054. No significant differences were found in terms of number of connections between the several methods. For this entire experiment, a high correlation between fitness and number of connections in the gridbrain was found, with $\rho = 0.6137742$.

7.2.4 Details of a simulation run

In this section we present the more detailed results of one simulation run, performed with the most successful combination of parameters found in previous experimentations: $p_a = 0.01, p_r = 0.01, p_s = 0.01, p_p = 0, p_c = 0.2, p_{rec} = 0.25, a = 0.5, s_{buf} = 100$.

Figure 7.13 shows the evolution of several metrics during the simulation run. This data was collected in $100K$ simulation cycle intervals. Each metric consists of an average of final values collected from agents that die during an interval.

In plot a) it can be observed that fitness increases in rapid bursts, followed by periods

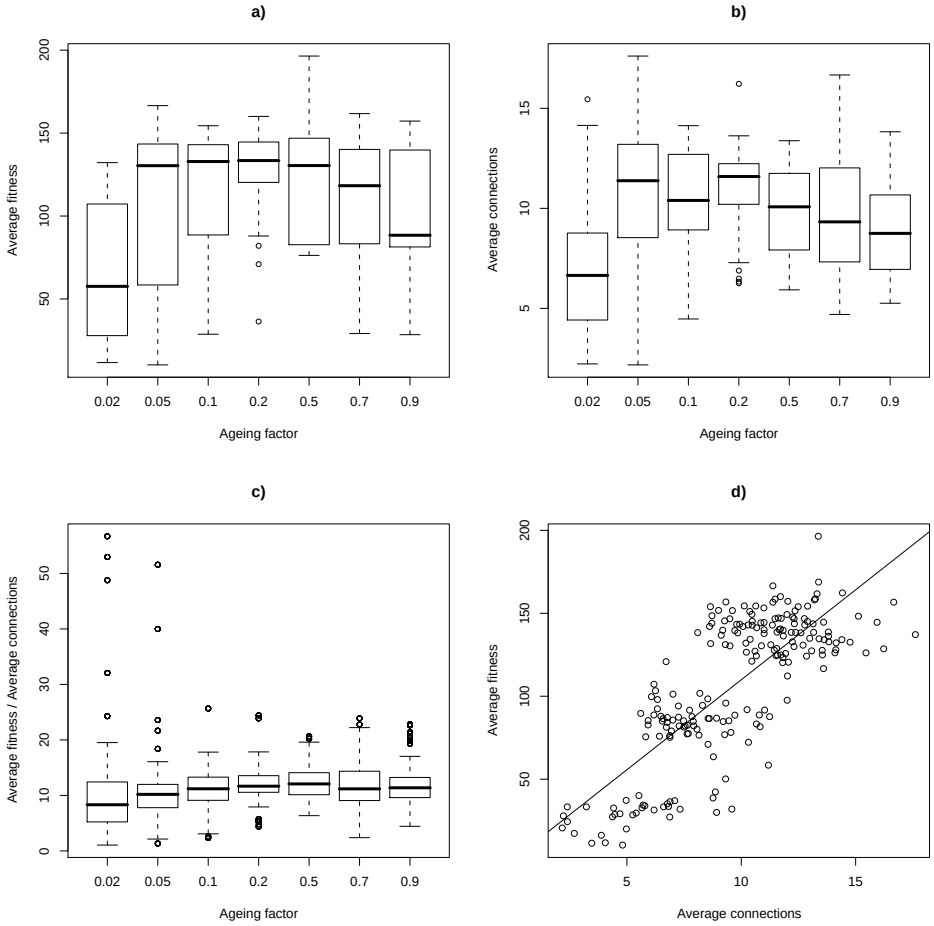

Figure 7.11: Results of experiments with various ageing factors: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections, with best fit line.

	0.02	*0.05*	*0.1*	*0.2*	*0.5*	*0.7*
0.05	0.02407	-	-	-	-	-
0.1	**0.00015**	1.00000	-	-	-	-
0.2	$\mathbf{4.6\times10^{-07}}$	1.00000	1.00000	-	-	-
0.5	**0.00039**	1.00000	1.00000	1.00000	-	-
0.7	0.01168	1.00000	1.00000	1.00000	1.00000	-
0.9	0.03371	1.00000	1.00000	0.17979	1.00000	1.00000

Table 7.11: Pairwise comparisons of fitness samples using the Wilcoxon rank sum test. The p-value for each pair of ageing factors is presented.

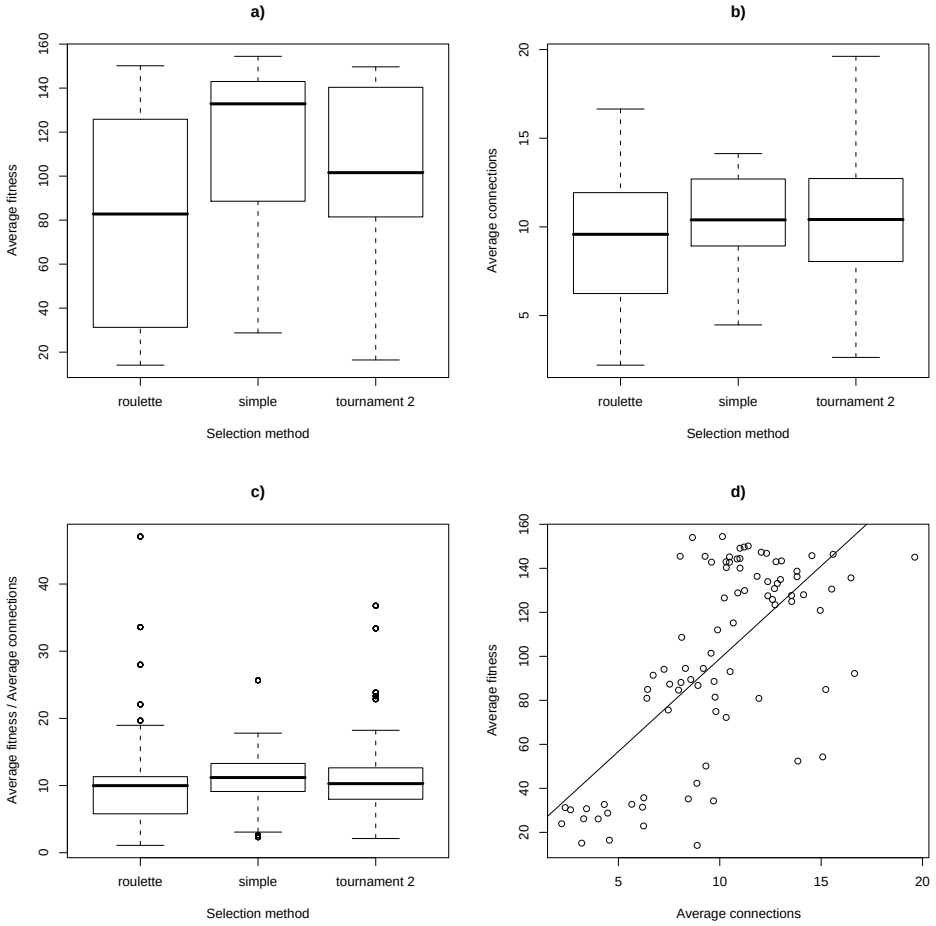

Figure 7.12: Results of experiments with alternative selection methods: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections, with best fit line.

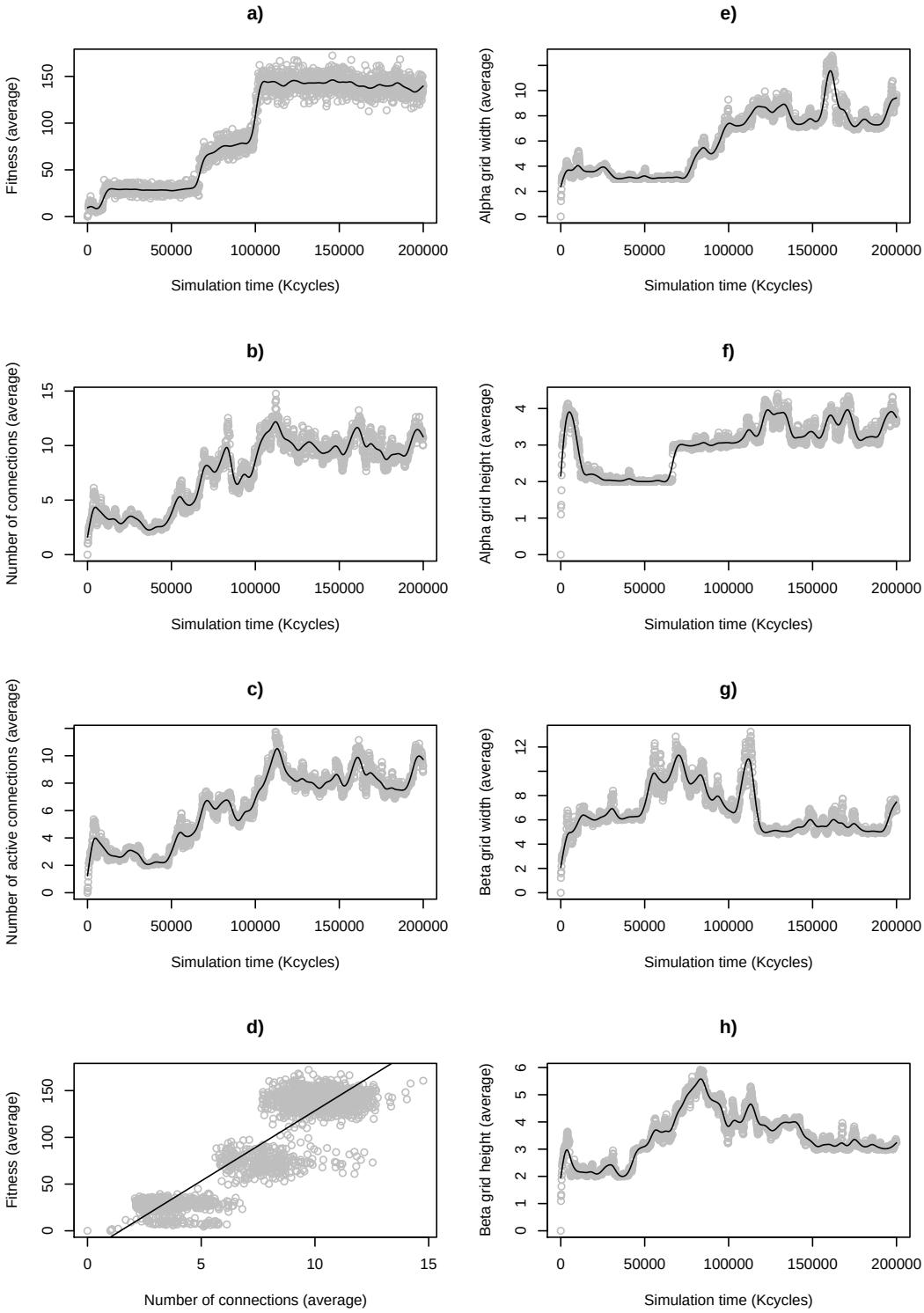

Figure 7.13: Evolution of several average metrics in a poison experiment run: a) fitness; b) number of connection in the gridbrain; c) number of active connections in the gridbrain; d) plot of fitness against number of connections; e) alpha grid width; f) alpha grid height; g) beta grid width; h) beta grid height.

of plateauing. This is consistent with biological models on evolutionary history, where species are observed to go through phases of rapid change (sometimes referred to as major evolutionary transitions), followed by large periods of stasis. In plot b), we can see that the general increase in fitness corresponds to an increase in the number of connections in gridbrains. The overall trend of fitness increase from time 50000 to time 110000 corresponds to and overall trend of increase in the number of connections, although with more oscillation. On the other hand, the final and longest fitness plateaus corresponds to a plateau in the number of connections. Another interesting detail to observe is that the sudden rise in fitness to the last plateau, and thus the break off from the previous one is closely preceded by a sharp decline in the number of connections. All these behaviors appears to display an adaptive behavior in the dimension of gridbrains. Useless complexifications are discarded and useful ones maintained. In fact, a high correlation of fitness and the number of connections throughout the experiment was found, with $\rho = 0.8006546$. This correlation is apparent in plot d), where plateau regions can also be observed.

In plot c) we can see that the number of active connections follows a progression very similar to the total number of connections, although, as expected, in lower values. In plots e), f), g) and h) we can observe the progression of the dimensions of the grids. Adaptive behaviors may also be observed. Each dimension produces a different curve, which reflects the different functionalities assigned to each grid, and the fact that evolutionary tweaking is performed mainly in different parts of the gridbrains, depending on the stage of the evolutionary process. All the bloat metrics appear to indicate that bloat is successfully contained.

In figure 7.14, the gridbrain from the agent with the highest final fitness in the last $100K$ cycles period is shown. The interpretation of the evolved circuit is the following:

- The agent rotates in the direction of the visible food object with the maximum nutritional value;
- The agent remembers the last direction of rotation, and keeps rotating in that direction, looking for food if no food is currently visible;
- The agent moves forward, applying a force with an intensity proportional to the

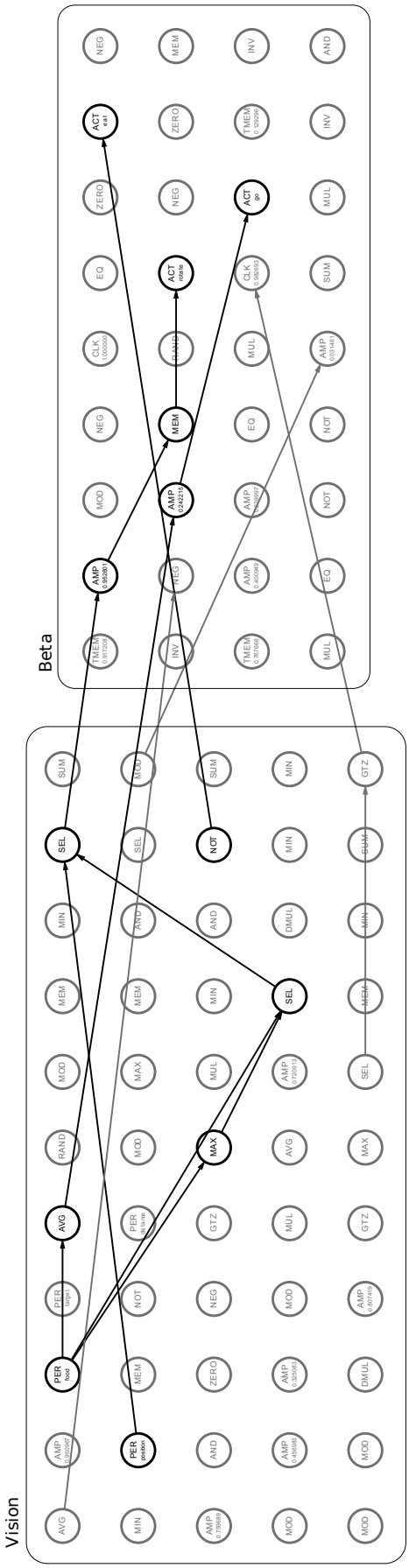

Figure 7.14: Evolved gridbrain from the poison scenario.

average nutritional value of visible objects;

- The agent always attempts to eat any object it is in contact with.

We will dissect the workings of the part of the circuit that controls rotation. The food perception component outputs a value that expresses the nutritional value of a visible object. This value is sent to a MAX aggregator, which triggers only when the highest value is found, outputting 1. The outputs of this aggregator and of the food perception component itself converge at a SEL component, which acts as a multiplier. This SEL component will only trigger for the object with the highest nutritional value, outputting this value. The outputs of the SEL and of the position perception component converge at another SEL component, which acts as a multiplier. The signal of this final value gives the direction of rotation, while the module of the value is proportional to both the nutritional value of the target object and its angular distance. This value is fed to the beta layer, arriving at an AMP component which scales it. The scaled value is then sent to a MEM component, which finally sends its output to the rotate action component. The intermediary MEM component causes the last received rotation value different from zero to be used in the absence of stimulus. The behavior or SEL components, as compared to MULs, was possibly useful in earlier stages of evolution. Likely, SELs remain as an artifact from this time.

Overall, notice that the evolutionary process was capable of combining different components to generate an highly adapted controller. Other successful runs of this same experiment produced similar mechanisms, while using different components in different ways to generate them.

7.3 Analysis of Results

In table 7.12 we present a summary of the results obtained with this scenario. For genetic operators, a positive impact means that we found that there is an activation probability range for this operator, at which significantly better results are produced than in the absence of the operator. A negative impact means that significantly worse results are produced using the operator than those obtained in its absence, at all activation probabil-

Operator / Operator Set / Parameter	Impact on Fitness	Impact on Bloat Control
Balanced Add / Remove	↗	↗
Balanced Split / Join	↗	↗
Unbalanced Add / Remove, Split / Join		↘
Change component	↗	
Change inactive component	↗	
Change parameter		
Recombination		↗
Buffer size	↗	
Fitness ageing	↗	
Use Roulette / Tournament	↘ ∗	

Table 7.12: Genetic operators and parameters benchmark: summary of results. The symbol ↗ indicates positive impact, while the symbol ↘ indicates negative impact. The absence of a symbol indicates no impact. An ∗ after a symbol indicates borderline significance.

ities tested. For mandatory evolutionary parameters, a positive impact means that better results can be achieved by fine-tuning the parameter. No symbol means that results where not significantly affected at any activation probability of parameterization tested. An ∗ after a symbol indicates borderline significance.

Results from this experiment set showed both the add/remove and split/join mutation operator pairs to be important in evolving gridbrains. The removal of any of these pairs resulted in considerable negative impact to the final fitness of agents. This indicates that both produce network topological changes that are important to evolutionary tweaking.

The change parameter mutator, on the other hand, was not found to be useful. Although final evolved gridbrains displayed signs of parameter tweaking (in AMP components), this tweaking was clearly achieved by the other operators. We consider this to be a good result, because it reduces the set of operators, and thus parametrization needed to perform successful runs.

The change component mutation operator was first tested in its *change inactive* modality. This operator was found to significantly affect results. As can be observed in figure 7.2, the worst results were produced when using a too high or too low probability of application of the operator. However, pairwise tests failed to produces conclusive results in terms of the ideal parameterization range. It is important to notice that the formating operator, when inserting and removing new rows and columns, also has the effect of varying the

inactive components available to the gridbrain. This means that, even at a probability of 0 for this operator, there is still a mechanism taking place that changes inactive components. This explains that activating this operator has a less dramatic effect than the connection level ones, and also suggests why the pairwise tests are inconclusive.

Pairwise tests, did however show that a *change inactive* probability of 1 performs significantly worse than values in the middle range (0.05 to 0.2). Furthermore, figure 7.2 show that the complete removal of neutral search produced the worst performance of all. Notice that assigning a probability of 1 to this operator removes neutral search altogether, since all the inactive components of the gridbrain are changed at each reproduction. This also affects inactive connections, because their origins and targets change. These results are a strong indication of the importance of neutral search in gridbrain evolution. The fact that a probability of 1 is significantly worse than a probability in the middle range, but not significantly different from lower probabilities (0 and 0.02) also suggests that the change inactive operator performs better in the middle range tested.

The *change any* modality of change component was shown to perform significantly better at lower probabilities (0.01 to 0.05) than it *change inactive* counterpart. This is not unexpected, as this modality is more disruptive. Unlike *change any*, it can produce mutations with phenotypical expression. *Change inactive* at 0.2 probability was found to perform significantly better than *change active* with a probability equal or higher than 0.05, but not significantly different from *change active* at probabilities lower than 0.05. Figure 7.3 shows that the inactive modality performed better than all the parametrizations of *change any*, but pairwise tests were inconclusive in establishing the significance of these results. Nevertheless, and based on these results, we lean towards recommending the *change inactive* modality, as it appears to be, in the worse case, as effective as *change any*, potentially better, and less disruptive.

The add/remove and split/join mutation operator pairs were designed to be symmetrical, one being able to reverse the effects of the other. This was conceived as a mechanism to make the size of gridbrains adaptable, and as a method to prevent bloat. In fact, experimentation showed that lowering the relative probability of the remove and join operators in relation to add and split resulted in more bloat, although not significantly affecting

fitness. These unbalanced probabilities seemed to affect gridbrain size adaptability. While in all other experiments with mutation operators a high correlation between fitness and number of connections was found, in this one the correlation was low.

Varying the probability of recombination showed no significant impact in fitness, but significant impact on bloat. In fact, the recombination operator clearly acts as a bloat control mechanism. The lower the recombination probability, the lower the fitness per number of connections was. Correlation between fitness and number of connections was also found to be low for this experiment set. Observing experimental data and gridbrains generated, we came to the conclusion that in a purely mutation based run, useless mutation are accumulated along an agent's lineage. The recombination operator allows the evolutionary process to select the most successful aspects of gridbrains while discarding the useless ones. In pure mutation, the probability of a useless mutation being discarded without adversely affecting the quality of the gridbrain is too low, and bloat is accumulated.

Both SEEA parameters, buffer size and ageing factor were found to significantly affect the fitness achieved. Both produced inferior results for values that were too low or too high. The size of the buffer was found to be important, although a wide range of values is admissible in this experimental scenario, as both buffer sizes of 100 and 1000 produce similar results. Pairwise testing only showed significant performance degradation for the larger buffer size of 10000. Buffer size plays a similar role to population size in conventional evolutionary algorithms, and it is thus not surprising that a too low or two high value has a negative effect on fitness. The importance of the fitness ageing mechanism was validated by the results, again with both too low or high values for this parameter having an advert effect on fitness.

The roulette and tournament variations of the SEEA selection mechanism were not found to be useful. In fact, they were borderline prejudicial. This seems to indicate that the simple version of the algorithm is a better choice.

The detailed results from an experiment run with the best parameters confirmed that, with a good choice of parameters, the evolutionary mechanisms of gridbrain genetic operators in conjunction with SEEA, result in adaptable gridbrain sizes and the prevention of bloat.

Overall, the evolutionary process was shown to be capable of successful complexification of gridbrains, starting with a population of agents with empty grids (zero width, zero height). Gridbrains adapted to the environment emerged, making use of several of the component types previously described. Notably, the gridbrain presented uses aggregation and memory components.

Chapter 8

Synchronization and Cooperation

In this chapter we present the results obtained from two experimental scenarios aimed at promoting synchronization and cooperation behaviors between agents. We test the two group behavior extensions to the SEEA algorithm, presented in chapter 5, as well as the gridbrain's ability to support this kind of behaviors. Furthermore, in these scenarios the agents are provided with two sensory channels, vision and sound.

The first scenario is called *synch*. It promotes the emergence of synchronization of sound emissions between the agents in the world. The second scenario is called *targets*. In it, the agents are evolved to acquire the capability of destroying moving targets through shooting. A single shot is not enough to destroy a target, so the agents have to cooperate to be successful in their goal.

8.1 The *Synch* Scenario

The *synch* scenario is defined in the `experiments/synch.lua` file that is included in the LabLOVE distribution. The Sim2D simulation environment is used.

8.1.1 Experimental Setup

Agent gridbrains are defined to have three grids: one alpha grid for vision, one alpha grid for sounds and one beta grid. In table 8.1 we present the grid component sets used in this scenario. The computational components are the ones described in section 4.2 and

Grid	Type	Components
Vision (Alpha)	Computational	AND, NOT, SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ, ZERO, MAX, MIN, AVG, DMUL, SEL, MEM
	Perceptions	POSITION, DISTANCE
Sound (Alpha)	Computational	AND, NOT, SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ, ZERO, MAX, MIN, AVG, DMUL, SEL, MEM
	Perceptions	POSITION, DISTANCE, VALUE, SYMEQ(color)
Beta	Computational	AND, NOT, SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ, ZERO, CLK, DMUL, MEM, TMEM
	Actions	GO, ROTATE, SPEAK

Table 8.1: Grid component sets of synch agents.

the same used in previous experiments. The action and perception components where described in section 6.3. Agents have a color symbol table, that is initialized with one symbol representing the agent's own color. Agents' color is set to the RGB value $(0, 0, 255)$, pure blue. The color symbol table is shared by the sound mechanism, being used both by the SYMEQ(color) perception component and the SPEAK action component. The mechanism of *symbol acquisition* and *symbol generation*, as described in section 6.2.3 are enabled for this table. The SYMEQ(color) component compares a color in the agent's symbol table with the color that defines a received sound message.

In table 8.2 we present the parameters used in this scenario. Physical parameters are the same ones used in the *poison* scenario. We decided to use a smaller world, with a smaller number of agents here, because this way we were able to run experiments faster. Instead of setting all the agents to a fixed 5000 simulation cycles maximum age, we set them to have a random maximum age in the $[4500, 5500]$ interval. We did this to prevent the emergence of synchronizations from the simultaneous creation of agents. Although the initial population of agents in a LabLOVE simulation is always created with random maximum ages in the $[0, max\_age]$ interval to prevent this, we decided to take this further step, as synchronization is the phenomena under observation in this scenario.

Parameter	Value
World width	500
World height	500
Number of agents	10
Agent size (s_{agent})	10
Initial agent energy	1.0
Agent maximum age, low limit	4500
Agent maximum age, high limit	5500
Go cost (g_{cost})	0.005
Rotate cost (r_{cost})	0.005
Go force constant (g_{const})	0.3
Rotate force constant (r_{const})	0.006
Drag constant (d_l)	0.05
Rotational drag constant (d_r)	0.05
Vision range (r_{vision})	150
Vision angle (a_{vision})	170^0
Sound range (r_{sound})	500
Speak interval (i_{speak})	250

Table 8.2: Synch experiment parameters.

Agent fitness is the best *synchronization score* obtained during the lifetime of the agent. Two variables are used: *best syncronization score* and *current synchronization score*. They are both initialized to 0. The *current synchronization score* is updated the following way:

- Every time the agents speaks, the *current synchronization score* is set to 1, the simulation time of the event is stored, as well as the symbol corresponding to the message sent;
- Every time the agent receives a message, the *current synchronization score* is incremented by 1 if the current simulation time is no more than 5 cycles ahead of the time the last message was sent by the agent and the symbol of the received message is equal to the symbol of the last message sent.

Every time the *current synchronization score* is updated, it is compared against the *best synchronization score*. If it is greater, the *best synchronization score* is set to the *current synchronization score*.

Unless otherwise indicated, *synch* experiment runs where performed with the following evolutionary parameters: $p_a = 0.01, p_r = 0.01, p_s = 0.01, p_p = 0.01, \delta_p = 1, p_c = 0.2, p_{rec} =$

$0.25, a = 0.5, s_{buf} = 100$. The *change inactive* modality of the *change component* mutation operator was used.

Experiment runs are performed with a duration of $2{\times}10^5$K simulation cycles.

8.1.2 Results

In the experiment for which results are presented in figure 8.1 and tables 8.3 and 8.4, we compared several group behavior configurations of the SEEA algorithm, as described in section 5.4. The configurations tested are the following: *simple* uses only the base SEEA algorithm; *grp* uses the group fitness extension; ss uses the super sisters extension; *ssM* uses the super sisters extension with mutations; *ss+grp* uses both the super sisters extension and group fitness and *ssM+grp* uses the super sisters extension with mutations and group fitness. A *group factor* of 0.5 was used in *group fitness* and a *sequence factor* of 1 was used with *super sisters*. Pairwise tests show that *simple* performs significantly worse than all others. These tests also show that *simple* and *grp* present significantly higher buffer diversity than all others.

As can be observed in plot a), the *simple* configuration performs significantly worse than all the other in terms of average final fitnesses, with a p-value for average fitness in all sets of $2.402{\times}10^{-10}$. All the configurations except *simple* attain similar performances. The p-value for average fitness in all sets except *simple* is 0.747, confirming that no significant difference exists in the performance of these sets.

In plot b) it can be observed that the *simple* and *grp* configurations present higher diversity in the buffer, by the end of runs, than the other configurations. The sets are significantly different in terms of buffer diversity, with a p-value of $3.03{\times}10^{-10}$.

In the experiment for which results are presented in figure 8.2 and table 8.5, we perform simulation runs using the *group fitness* extension of the SEEA algorithm and test the impact of the *group factor* parameter. For this purpose we performed 30 sets of simulation runs at different group factors. As can be observed, the best performances are obtained with a group factor larger than 0.3. As the group factor increases from 0 to 0.3 there is an increase in average fitnesses, after that the results remain similar. These results were found to be significant, with an overall p-value of $2.048{\times}10^{-11}$. Pairwise tests show that

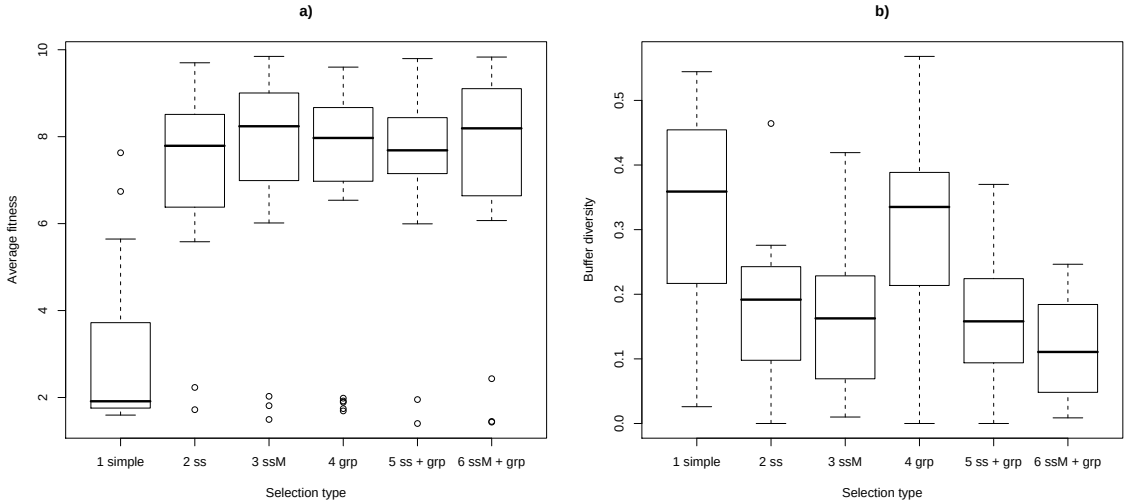

Figure 8.1: Comparison of group behavior configurations in the synch scenario: a) final average fitnesses; b) final diversity in the buffer.

	1 simple	*2 ss*	*3 ssM*	*4 grp*	*5 ss + grp*
2 ss	$\mathbf{3.2\times10^{-10}}$	-	-	-	-
3 ssM	$\mathbf{7.1\times10^{-09}}$	1	-	-	-
4 grp	$\mathbf{2.7\times10^{-07}}$	1	1	-	-
5 ss + grp	$\mathbf{8.1\times10^{-10}}$	1	1	1	-
6 ssM + grp	$\mathbf{4.3\times10^{-08}}$	1	1	1	1

Table 8.3: Pairwise comparisons of fitness samples using the Wilcoxon rank sum test. The p-value for each pair of group behavior configurations is presented.

	1 simple	*2 ss*	*3 ssM*	*4 grp*	*5 ss + grp*
2 ss	**0.00023**	-	-	-	-
3 ssM	$\mathbf{5.8\times10^{-05}}$	1.00000	-	-	-
4 grp	1.00000	**0.00127**	**0.00106**	-	-
5 ss + grp	$\mathbf{7.5\times10^{-05}}$	1.00000	1.00000	**0.00081**	-
6 ssM + grp	$\mathbf{6.4\times10^{-07}}$	0.18432	1.00000	$\mathbf{3.2\times10^{-05}}$	0.41954

Table 8.4: Pairwise comparisons of buffer diversity samples using the Wilcoxon rank sum test. The p-value for each pair of group behavior configurations is presented.

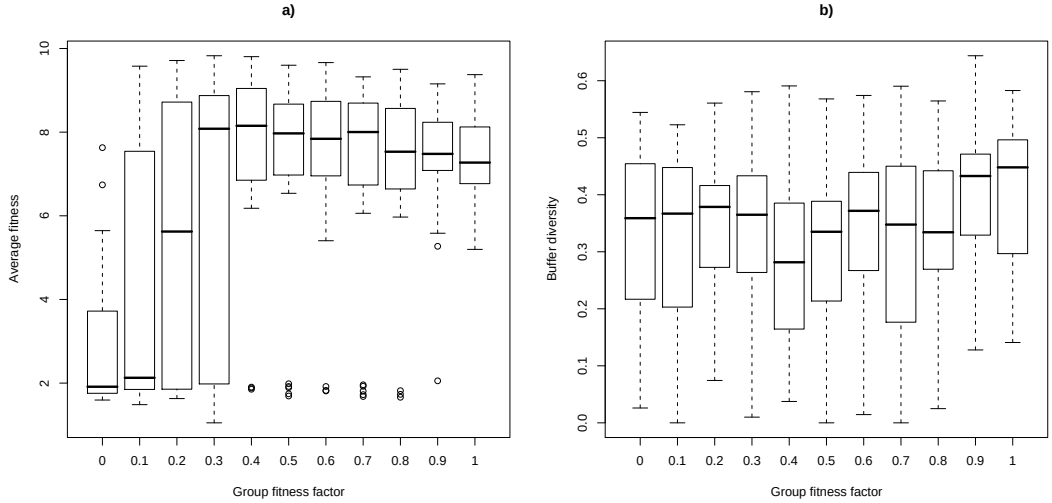

Figure 8.2: Impact of group factor in the synch scenario: a) final average fitnesses; b) final diversity in the buffer.

	0	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*
0.1	1.0000	-	-	-	-	
0.2	0.2075	1.0000	-	-	-	
0.3	**0.0022**	0.5476	1.0000	-	-	
0.4	$\mathbf{1.4\times10^{-08}}$	**0.0052**	0.1040	1.0000	-	
0.5	$\mathbf{1.2\times10^{-06}}$	0.0620	1.0000	1.0000	1.0000	
1	$\mathbf{2.4\times10^{-08}}$	0.0128	0.9479	1.0000	1.0000	

Table 8.5: Pairwise comparisons of fitness samples using the Wilcoxon rank sum test. The p-values for some pairs of group factor values are presented.

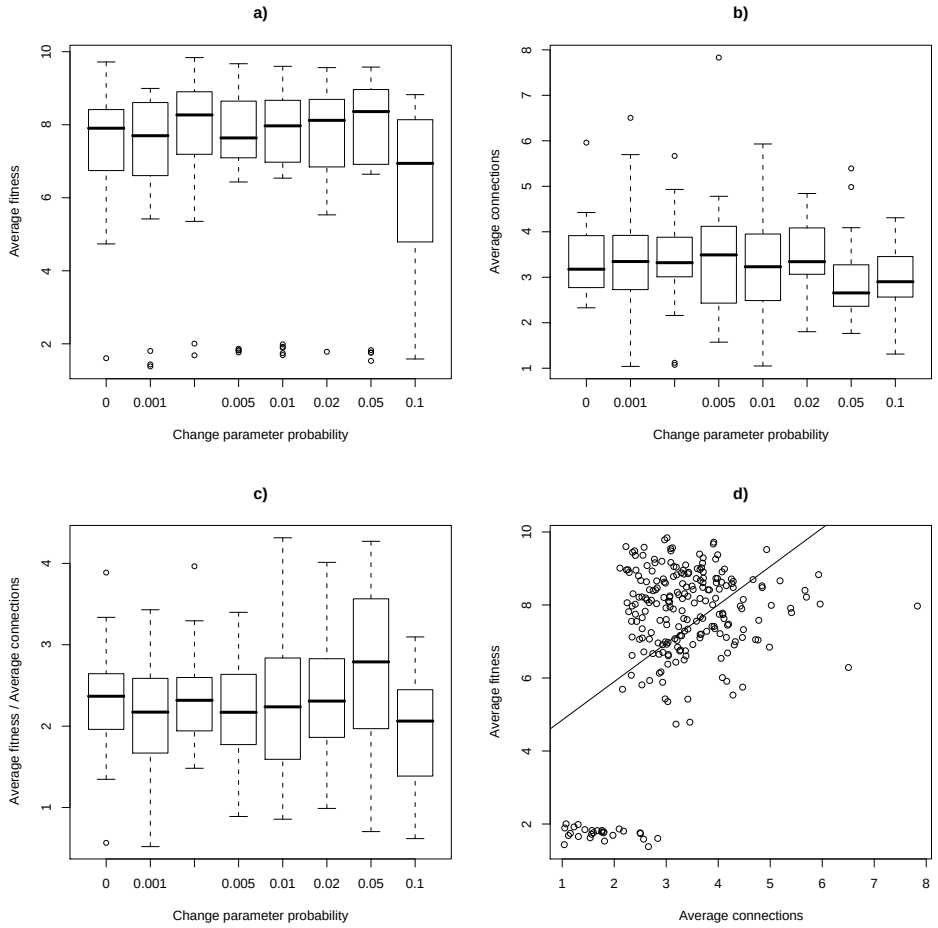

Figure 8.3: Results of experiments with various change parameter probabilities in the synch scenario: a) box plot of average final fitnesses; b) box plot of average final number of gridbrain connections; c) box plot of average final fitness per average final number of gridbrain connections; d) plot of average final fitnesses against final number of gridbrain connections, with fit line.

a group factor equal or higher than 0.3 performs significantly better than a group factor of 0.

As can be seen in figures 8.1 and 8.2, diversity in the buffer was not found to depend on the group factor. In fact, differences in buffer diversity were not found to be significant, with a p-value of 0.02172.

In the experiment corresponding to the results shown in figure 8.3, we again tested the impact of the change parameter mutation operator. Since it was found to have no impact in the *poison* scenario, we decided to also test its impact in the *synch* scenario. We performed simulation runs with $p_a = 0.01, p_r = 0.01, p_s = 0.01, p_j = 0.01, p_c = 0.2, \delta_p =$

$1, p_{rec} = 1, a = 0.1, s_{buf} = 100$, and various p_p values. We used the *group fitness* SEEA extension with a group factor of 0.5. Again there where no significant differences in terms of fitnesses, with a p-value of 0.02937.

In figure 8.4 we show the evolution of fitness for simulation runs using the several SEEA configurations tested. These runs were all done using the same random seed. It is apparent that different configurations produces different fitness curves. The *simple* configuration fails to achieve the high fitness values of all others. However, it is interesting to notice that there is an initial peak in fitness, that is not sustained. This patterns was observed in many of the instances of this experiment. It is clear that agents learn to synchronize with the sound emissions of other agents, but then rely solely on exploring this reaction. There is not individual value in maintaining sound emission in the absence of stimulus, so the population falls back to a lower-fitness, individual behavior. All the other combinations of cooperation extensions achieve higher fitnesses. Setups that include the *super sisters* extension, except *ss+grp*, take longer to achieve higher fitness plateaus.

8.2 The *targets* Scenario

The *targets* scenario is defined in the `experiments/targets.lua` file that is included in the LabLOVE distribution. The Sim2D simulation environment is used.

8.2.1 Experimental Setup

Agent gridbrains are defined to have three grids: one alpha grid for vision, one alpha grid for sounds and one beta grid. In table 8.6 we present the grid component sets used in this scenario. The computational components are the ones described in section 4.2 and the same used in previous experiments. The action and perception components where described in section 6.3. Agents have a color symbol table, that is initialized with one symbol representing the agent's own color. Agents' color is set to the RGB value $(0, 0, 255)$, pure blue. The mechanisms of *symbol acquisition* and *symbol generation*, as described in section 6.2.3 are enabled for this table. The SYMEQ(color) component compares a color in the agent's symbol table with the color of a visible object.

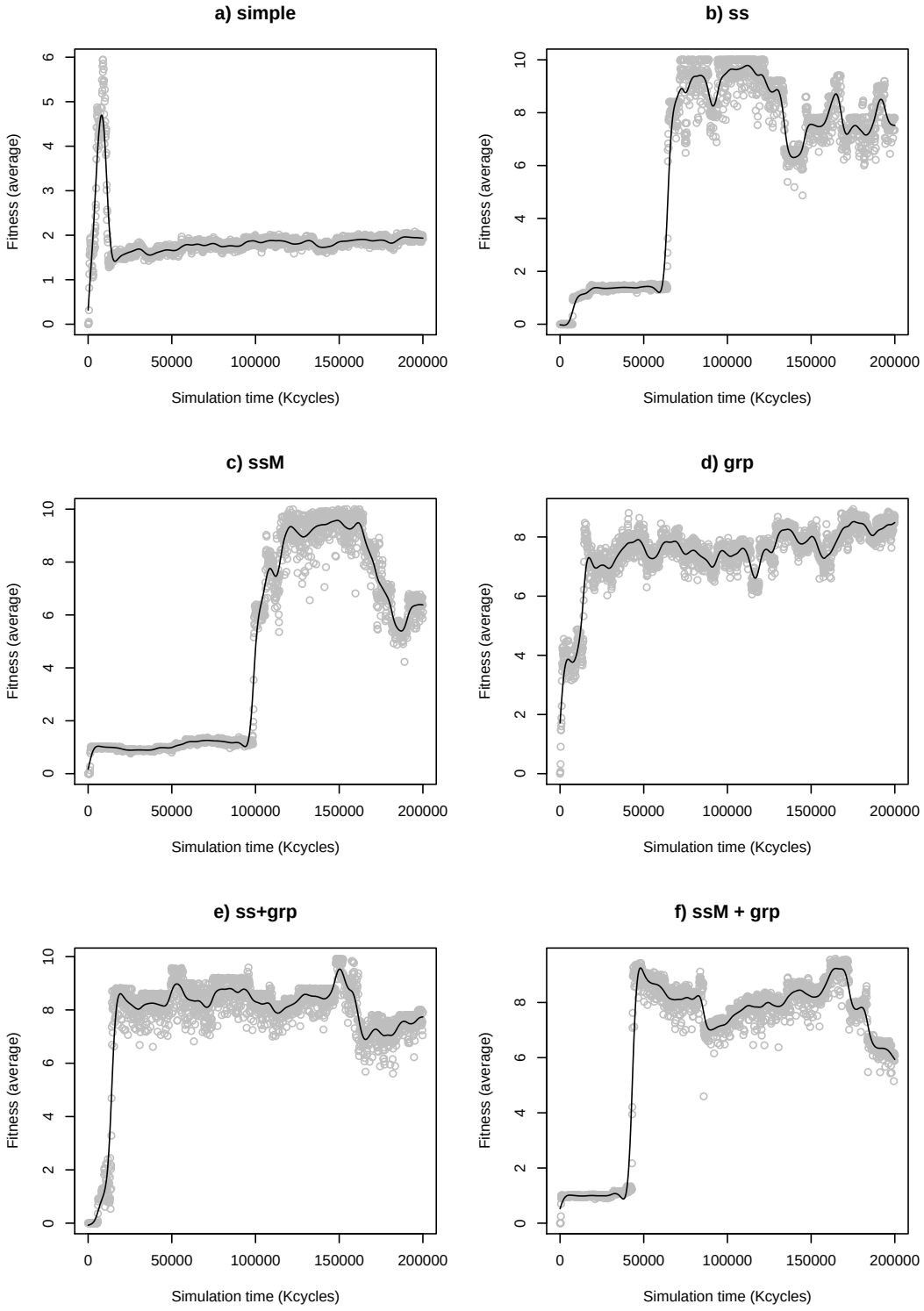

Figure 8.4: Evolution of fitness in synch experiment runs with different SEEA configurations: a) simple; b) super sisters; c) super sisters with mutations; d) group fitness; e) super sisters and group fitness; f) super sisters with mutation and group fitness.

Grid	Type	Components
Vision (Alpha)	Computational	AND, NOT, SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ, ZERO, MAX, MIN, AVG, DMUL, SEL, MEM
	Perceptions	POSITION, DISTANCE, LASER_TARGET, LOF, SYMEQ(color)
Sound (Alpha)	Computational	AND, NOT, SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ, ZERO, MAX, MIN, AVG, DMUL, SEL, MEM
	Perceptions	POSITION, DISTANCE, VALUE, SYMEQ(color)
Beta	Computational	AND, NOT, SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ, ZERO, CLK, DMUL, MEM, TMEM
	Actions	GO, ROTATE, FIRE

Table 8.6: Grid component sets of targets agents.

In table 8.7 we present the parameters used in this scenario. Physical parameters are the same ones used in the *synch* scenario. In this scenario, agents are set to have an average maximum lifespan that is twice as long as the ones used in previous scenarios. This is because we determined in preliminary experimentation that the type of behavior that we are attempting to emerge takes longer to be achieved. The agents must lock on and follow a target for some time to produce an effective shot. The agent's vision range and angle were also increased, as compared to previous scenarios. We chose to increase them because we were looking to emerge behaviors of cooperation in shooting, so we wanted to increase the probability of several agents observing the same target at the same time.

Targets are Sim2D objects that move through the world at a constant speed. Their initial position and direction are initialized to random values when they are created. When a target hits a world limit it bounces off, as if hitting a wall. This bouncing is modeled as a simple Newtonian physics deflection with perfect elasticity. A target that is placed in the world never stops or loses speed. It has no maximum age, and is only removed from the world if it is destroyed. A destroyed target is always replaced by a new, randomly generated one.

Target energy is set to 1.1, so that a perfect shot from a single agent is not enough to

Parameter	**Value**
World width	500
World height	500
Number of agents	10
Number of targets	5
Agent size (s_{agent})	10
Target size (s_{target})	5
Target speed ($v_{targets}$)	0.1
Initial agent energy	1.0
Target energy	1.1
Agent maximum age, low limit	9500
Agent maximum age, high limit	10500
Go cost (g_{cost})	0.005
Rotate cost (r_{cost})	0.005
Go force constant (g_{const})	0.3
Rotate force constant (r_{const})	0.006
Drag constant (d_l)	0.05
Rotational drag constant (d_r)	0.05
Vision range (r_{vision})	300
Vision angle (a_{vision})	350^0
Laser fire interval ($l_{interval}$)	2000
Laser length (l_{laser})	25
Laser speed (v_{laser})	100
Laser strength factor (s_{laser})	1
Laser cost factor (l_{cost})	0.1
Laser hit duration (t_{laser})	2

Table 8.7: Targets experiment parameters.

destroy it. As the laser strength factor is set to 1, the maximum energy contained in an agent shot is 1. To destroy a target at least two agents must successfully shoot at it, with shots arriving at the target in an interval of two or less simulation cycles. Laser shots are set to move at a very high speed (100 distance units per simulation cycle), so that agents simultaneously shooting at the same target from different distances have a good chance of having their shots arrive at the target within the t_{laser} interval. The laser cost factor of 0.1 makes it possible for an agent to produce at most 10 shots during it's lifetime with the initial energy it has available. The laser fire interval of 2000 makes it possible for the agent to produce at most 5 perfect shots during it's lifetime.

A simple fitness function that, at first glance, appear appropriate for this experiment is the amount of targets that the agent collaborated in successfully destroying. The problem with this function is that, to achieve one single target destruction, complex behavior is needed. To achieve the destruction of one target, an agent must follows the movement of the target for long enough time and then shoot at it both after enough target locking time has passed and in synchronization with another agent. This function would not be very useful because it would reduce the evolutionary process to brute force search.

Instead, we defined a fitness function that rewards both the effectiveness of a shot and its synchronicity with shots from other agents. Agent fitness is the best *shooting score* obtained during the lifetime of the agent. Two variables are used: *best shooting score* and *current shooting score*. They are both initialized to 0. The *current shooting score* is updated the following way:

- Every time an agent shoots, its *current shooting score* is reset to 0;
- Every time a target receives a shot, the energy of all shots currently accumulated in targets is added to the *current shooting score* of their respective originating agents.

Every time the *current shooting score* is updated, it is compared against the *best shooting score*. If it is greater, the *best shooting score* is set to the *current shooting score*.

Targets experiment runs where performed with the following evolutionary parameters: $p_a = 0.01, p_r = 0.01, p_s = 0.01, p_p = 0, p_c = 0.2, p_{rec} = 0.25, a = 0.5, s_{buf} = 100$. The *change inactive* modality of the *change component* mutation operator was used.

8.2.2 Results

In the experiment for which results are presented in figure 8.5 and tables 8.8 and 8.9, we compared several group behavior configurations of the SEEA algorithm. The configurations tested are the following: *simple* uses only the base SEEA algorithm; *grp* uses the group fitness extension and ss used the super sisters extension. A *group factor* of 0.5 was used in *group fitness* and a *sequence factor* of 1 was used in *super sisters*. Pairwise tests show that the *simple* and *grp* configurations perform significantly better than *ss*. These tests also show that *simple* and *ss* promote significantly more diversity in the buffer than *ss*.

Both *simple* and *grp* were found to perform significantly better than *ss*, with a p-value for the overall fitnesses per set of 8.118×10^{-12}. Also, final buffer diversity for both the first sets was found to be significantly higher that the one for *ss*, with a p-value for the overall buffer diversity per set of 2.054×10^{-06}.

The *simple* and *grp* configurations were not found to have significantly different performances, with a p-value for fitness between these two sets of 0.7117 and a p-value for buffer diversity of 0.8303.

In figure 8.6, we can observe the progression of several metrics during a successful simulation run using the *simple* configuration. In plot a) it can be seen that fitness goes through sudden increases followed by plateaus of different sizes, as in previous scenarios. The increase in fitness to the final plateau corresponds to a sudden increase in the number of targets destroyed per agent, from near 0 to around 0.5. The number of connections in gridbrains again shows adaptability, as shown in plot b), with an initial exploration phase with a large increase in this number, followed by a sudden decrease that coincides with an increase in fitness. The number of connections stabilizes in the end of the run.

In plot d) it can be observed that, in the beginning of the simulation, there is a spike in the number of agents dead per sample interval. Analysis of gridbrains produced at this stage showed that this corresponds to the initial phases of agents learning to shoot. They are not yet capable of temporizing the shooting, so they shoot too many times and exhaust their energy quickly. Learning to temporize corresponds to a sharp decrease in this value. A smaller peak at about 2×10^5 simulation cycles corresponds to an increase

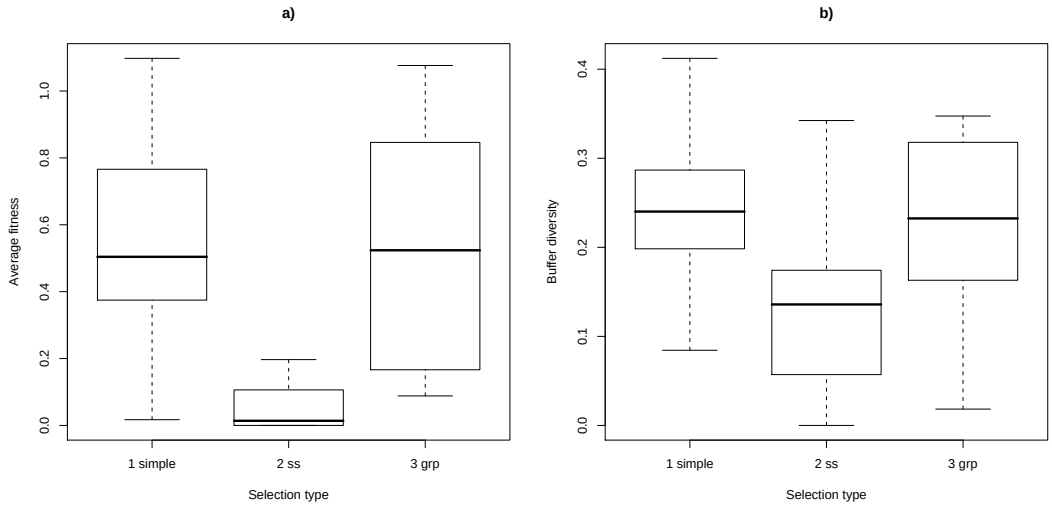

Figure 8.5: Comparison of group behavior configurations in the targets scenario: a) final average fitnesses; b) final diversity in the buffer.

	1 simple	*2 ss*
2 ss	$\mathbf{1.5{\times}10^{-09}}$	-
3 grp	0.72	$\mathbf{2.0{\times}10^{-09}}$

Table 8.8: Pairwise comparisons of fitness samples using the Wilcoxon rank sum test. The p-value for each pair of group behavior configurations is presented.

	1 simple	*2 ss*
2 ss	$\mathbf{4.8{\times}10^{-06}}$	-
3 grp	0.83602	**0.00013**

Table 8.9: Pairwise comparisons of buffer diversity samples using the Wilcoxon rank sum test. The p-value for each pair of group behavior configurations is presented.

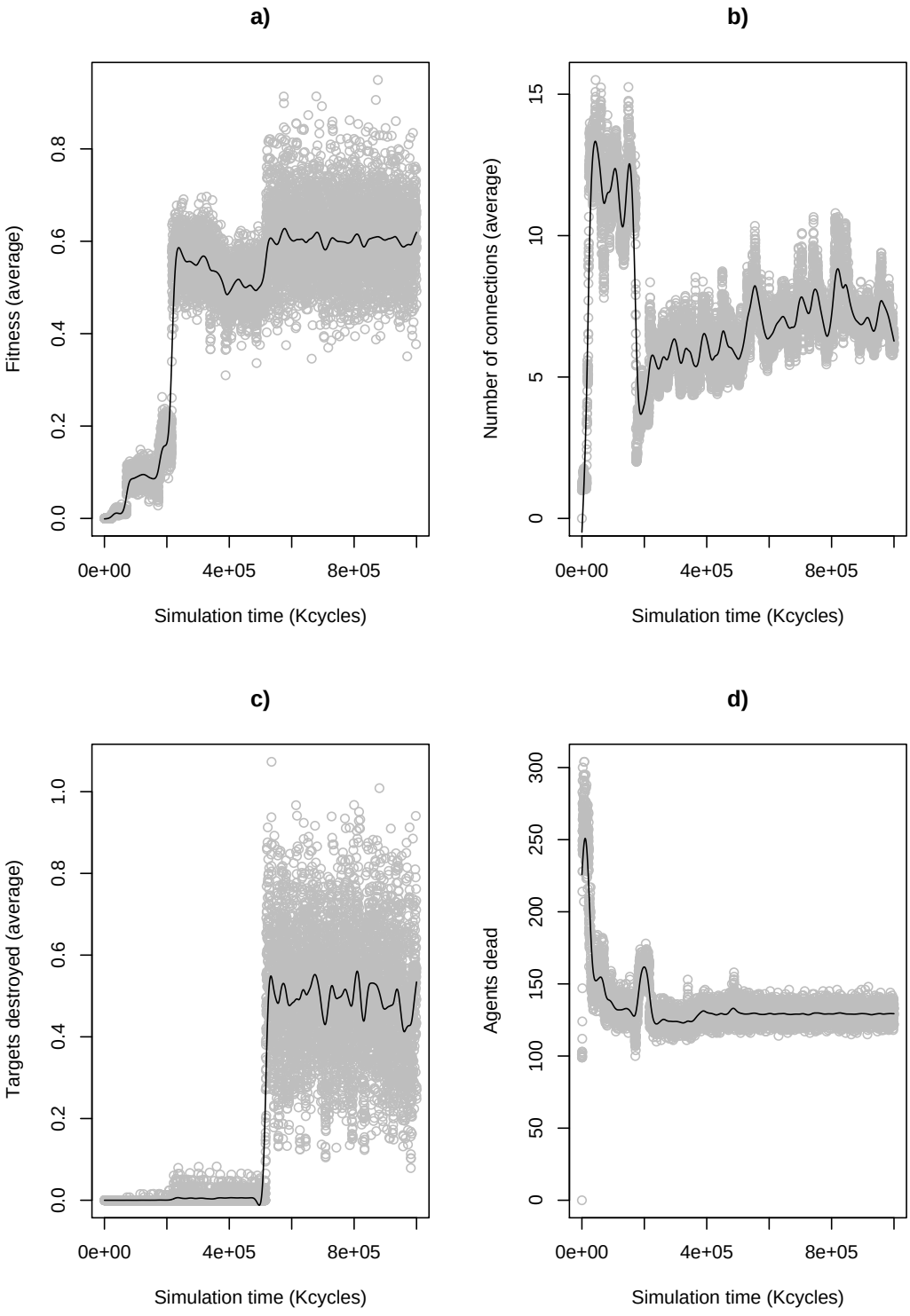

Figure 8.6: Evolution of several metrics in a successful targets experiment run using the simple configuration: a) average fitness; b) average number of gridbrain connections; c) average targets destroyed per agent; d) agents dead in sampling interval.

in the agent's ability to lock at targets and produce more effective shots, while not being able to distinguish targets from other agents. The ability to distinguish and only shoot at targets corresponds to a final decrease in the number of agents dead per sample cycle. This value stabilizes for the rest of the run at around 130, indicating that the majority of the agents live until their maximum allowed age.

In figure 8.7, the best gridbrain from the last sample interval of a simulation run is shown. As can be seen, a SEL component is used in the vision alpha grid to select a direction of rotation, based on visible objects' colors and relative positions. The gridbrain selects an object that has a target color and keeps rotating in its direction. The intensity of rotation is scaled by an AMP component in the beta grid. A CLK component in the beta grid triggers firing at regular intervals. The clock is synchronized by shooting sounds from other agents, received by the sound alpha grid. The agent also performs forward and backward movements, triggered by shooting sounds and proportional to the sound origin's relative position. It is not clear why this last behavior is beneficial, but it was found in several of the evolved gridbrains for this scenario. Maybe it helps in target following, or maybe it helps agents to stay out of the firing line of other agents.

8.3 Analysis of Results

In the two scenarios presented in this chapter, we analyze the importance of group behavior extensions to SEEA in the emergence of cooperative behaviors. In the *synch* scenario, no cooperation is achieved without using one of the extensions, while in the *targets* scenario, extensions are shown not to be necessary. We believe that this difference is caused by the nature of the mechanism that evolve in response to the demands of the environment.

It can be observed in plot a) of figure 8.4, for a *synch* run with no extensions, that a high peak in fitness is achieved in the beginning, but it is not sustained. The experimental data we collected reveals that this kind of peak is formed in several of the simulation runs with no extensions. It happens because, when agents achieve synchronization, from an egoistic perspective it is sufficient for each agent to send a message when a message is received to achieve high fitness. There is no individual evolutionary pressure to maintain

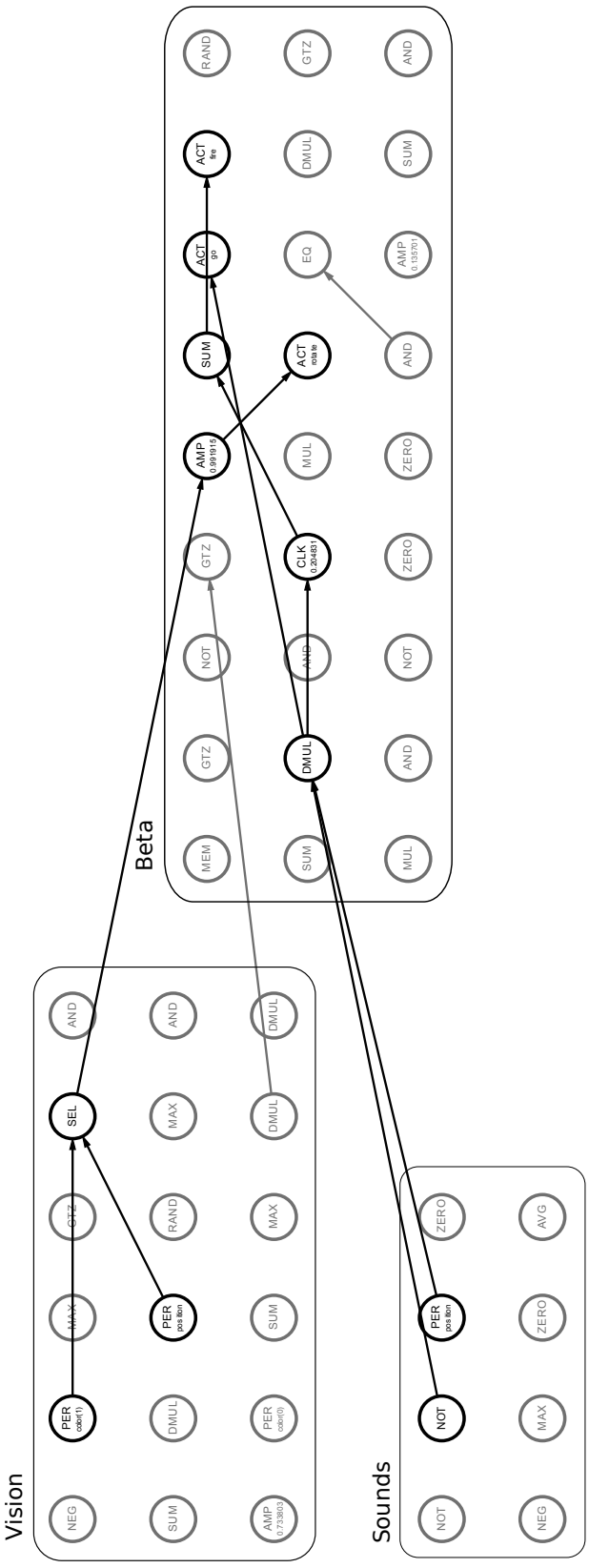

Figure 8.7: Evolved gridbrain from the targets experiment.

more complex synchronization mechanisms with other agents, by using CLK components. It then happens that the simpler egoistical agents become dominant and the population regresses to no cooperation. The use of any of the setups that include group behavior extensions solves this problem, by making the reproductive success of each agent dependent on the reproductive success of coexisting agents.

It was expected that the same thing happened for the *targets* scenario, but it did not. Here, the simple configuration is sufficient to achieve good performances. In this case, more sophisticated synchronization mechanisms that use CLK components are always necessary, because even from an egoistic perspective, the agents need to temporize their shots to gain fitness, because the energetic value of the shot is dependent on the amount of time that they spent locking on a target. These more sophisticated mechanism are in place before synchronization is achieved, and there is evolutionary pressure not to remove them. This then leads to a situation where, even from and egoistic perspective, agents are better off synchronizing their clocks by the firing of shots from other agents, because they are more likely to collect repeated scores for the same shot.

While *synch* agents would also be more likely to achieve higher scores by synchronizing, the synchronization mechanism itself is not maintained until this selective pressure would come into play.

Another interesting thing to notice is that, in the *targets* scenario, the *group fitness* extension performs as well as base SEEA , while the *super sisters* extension performs much worse (figure 8.5). The clue to why this happens is contained in the buffer diversity values collected for both scenarios. Both in *synch* (figure 8.1) and *targets* (figure 8.5), buffer diversity is significantly lower when the *super sisters* extensions is used, either by itself or in conjunction with *group fitness*. This is not unexpected, as the *super sisters* extension causes populations to be generated with individuals with similar genetic codes. When a good variation is found, several individuals with very similar genetic code will get to the buffer. The accumulated effect of this is a reduction in buffer diversity. Lower diversity causes the evolutionary process to be less effective, as wildly recognized in the field of evolutionary computation.

In the *synch* scenario, the lack of diversity caused by the *super sisters* extension does

not affect the performance of the evolutionary algorithm, but this is likely due to the simplicity of the problem. In fact, in figure 8.4 we can see that during evolutionary runs, configurations that include the *super sisters* extensions tend to go through initial plateaus of low fitness.

On the other hand, it is also interesting that while not being necessary in the *targets* scenario, the *group fitness* extension performs as well as the simple configuration.

When defining simulation scenarios, it may be difficult to access the importance of cooperation or the need for a group behavior extension in SEEA. These results indicate that a good practice is to always use the *group fitness* extension in this kind of evolutionary multi-agent simulations.

In the *synch* scenario we further investigated the importance of the *change parameter* operator. As in the *poison* scenario, it was found to have no impact (figure 8.3). Both the *synch* and the *targets* scenario were capable of generating gridbrains with fine-tuned parameters in CLK and AMP components without the need for this operator. Our conclusion is that neutral mutations that generate components with random parameter values, coupled with topological mutations are sufficient for the evolutionary search to optimize component paramters, and no further improvement is achieved by the presence of the *change parameter* operator.

8.4 Comparison with an Evolutionary Robotics Experiment

A research work in evolutionary robotics, with parallels to the scenario we presented in this chapter, has been recently published [Floreano *et al.*, 2007]. It deals with the emergence of cooperation and communication in groups of robots. It does not, however, deal with synchronization.

In this work, the environment used is the following: 10 robots are placed in a square arena. The arena contains a food source and a poison source, in opposite corners. Both sources constantly emit a red light that can be sensed by robots from anywhere in the arena. The food source has a white paper underneath it, and the poison source has a black paper. The robots are capable of detecting the color of these papers if very near the

source. The robots move by way of two independent tracks and can also activate a blue light that is visible to other robots. They are controlled by a very simple, fully connected neural network with only input and output layers.

The purpose of this experiment was to create a situation where the robots could increase their performance by cooperative behavior and communication. Robots can use their blue light to signal either the food source or the poison source, providing useful information to the other robots.

The genotype of the robot consists of one gene for each connection weight in the neural network. The genetic operators used where single-point crossover and bit flip mutations. The population of evolving robots consisted of 100 colonies of 10 robots. Several selection strategies are used to select individuals for succeeding generations. In individual-level selection, the 200 best individuals from the entire population are chosen to generate the next population. Another strategy is to populate each colony with equal individuals, in order to explore the effect of high genetic relatedness. In this case, 100 individuals are created from the selected 200 progenitors, by using only one of the recombinations from each pair of progenitors. Each generated individual is then used to populate its colony, with exact copies of itself. A colony-level strategy chooses the 200 individuals from the 20 best performing colonies. Finally, a combination of high relatedness and colony-level selection is used, combining the two former strategies.

Results showed that the best performance was obtained by colony-level selection with high relatedness, followed by individual-level selection with high relatedness and then individual-level selection with no relatedness. The worst performance was from colony-level selection with low relatedness. All the selection strategies except individual-level selection with low relatedness led to the emergence of behaviors of food or poison signaling, coupled respectively with blue light approaching and avoidance. Some of the runs converged to food signaling and others to poison signaling, with a greater tendency for the first.

High relatedness explores kin selection and can be compared to our super sisters extension, while colony-level selection explores group selection and can be compared to our group fitness extension. Both our experiments and the ones discusses in this section arrive

to a common conclusion: either some form of kin or group selection may be necessary for cooperation to emerge. However, in our experiments, group selection performs better while in the evolutionary robotics experiment, kin selection or a combination of kin selection and group selection perform better. There are several possible reasons for this difference.

The evolutionary robotics experiment uses a generation based evolutionary algorithm with synchronous creation and destruction of agents, while SEEA is a steady-state evolutionary algorithm and we use a continuous simulation. Our extensions to promote cooperation share the same natural inspiration, but are more fuzzy. We do not have clear generation limits nor contained colonies. It is possible that these different conditions lead to different performances in kin and group selection.

Another possible reason is that the evolutionary robotics experiment uses a much simpler representation and phenotype for the controller. It is fixed-sized, with a fixed topology of only two layers and uses simple artificial neurons. This contrast with our variable sized representation, with evolving topologies and heterogeneous components. It is thus possible that the diversity decrease caused by kin selection is less hurtful to performance in the evolutionary robotics experiment.

Yet another reason for the difference can simply be the different nature of the environments in the evolutionary robotics experiments and ours.

Chapter 9

A Competition Scenario

In this chapter we present the experimental results from a scenario with two evolving species in competition, called *battle*. The two species are defined with equal initial settings, but are assigned to different SEEA buffers.

The goal of this scenario is to experiment with the simultaneous evolution of multiple species and to observe the dynamics of co-evolution of two species with antagonic goals. Previous scenarios where more focused on the goal of evolving computational intelligence in a multi-agent environment. This one is a first step in defining artificial life simulations using our evolutionary framework. We believe it leads to interesting future paths of research.

9.1 Experimental Setup

The *battle* scenario is defined in the `experiments/battle.lua` file that is included in the LabLOVE distribution. The Sim2D simulation environment is used.

In table 9.1 we present the grid component sets used in this scenario. As in the *targets* scenario, agents have three grids: two alpha, one for vision and one for sound and one beta for decision making. The component sets are a combination of the components used in the *poison* scenario with the components used in the *targets* scenario.

In table 9.2 we present the parameters used in this scenario. Physical parameters for agents are the same ones used in the *targets* scenario and physical parameters for food

Grid	Type	Components
Vision (Alpha)	Computational	AND, NOT, SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ, ZERO, MAX, MIN, AVG, DMUL, SEL, MEM
	Perceptions	POSITION, DISTANCE, LASER_TARGET, LOF, EAT_TARGET, ORIENTATION, SYMEQ(color), SYMDIST(food)
Sound (Alpha)	Computational	AND, NOT, SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ, ZERO, MAX, MIN, AVG, DMUL, SEL, MEM
	Perceptions	POSITION, DISTANCE, VALUE, SYMEQ(color)
Beta	Computational	AND, NOT, SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ, ZERO, CLK, DMUL, MEM, TMEM
	Actions	GO, ROTATE, FIRE, EAT, SPEAK

Table 9.1: Grid component sets of battle agents.

items are the same used in the *poison* scenario. This time we have two species evolving at the same time, and the world is constantly populated with a fixed number of food items that the agents can consume to increase their energy level. The only difference between the two species is their color, one being red (RGB: $255, 0, 0$) and the other being blue (RGB: $0, 0, 255$). We will refer to them as the red species and the blue species. Food items always provide an increase in energy level of 1 and are green (RGB: $0, 255, 0$).

The fitness function used is the same as in the *targets* scenario, with one species being defined as the target for the other. Agents will thus evolve to increase their ability to destroy elements of the other species.

9.2 Results and Analysis

This scenario was found to be more computationally demanding than previous ones. This is due to the appearance of a larger number of costly to compute interactions between agents, namely shots and sound emissions. Four simulation runs were performed. In this section we will present and analyze the evolutionary history of two of them.

In figure 9.1 we present the evolution of several metrics for the first simulation run.

Parameter	**Value**
World width	1000
World height	1000
Number of agents per species	20
Number of food items	20
Agent size (s_{agent})	10
Food item size (s_{food})	10
Initial agent energy	1.0
Initial food item energy	1.0
Agent maximum age, low limit	9500
Agent maximum age, high limit	10500
Go cost (g_{cost})	0.005
Rotate cost (r_{cost})	0.005
Go force constant (g_{const})	0.3
Rotate force constant (r_{const})	0.006
Drag constant (d_l)	0.05
Rotational drag constant (d_r)	0.05
Vision range (r_{vision})	300
Vision angle (a_{vision})	350^0
Laser fire interval ($l_{interval}$)	2000
Laser length (l_{laser})	25
Laser speed (v_{laser})	100
Laser strength factor (s_{laser})	1
Laser cost factor (l_{cost})	0.1
Laser hit duration (t_{laser})	2

Table 9.2: Battle experiment parameters.

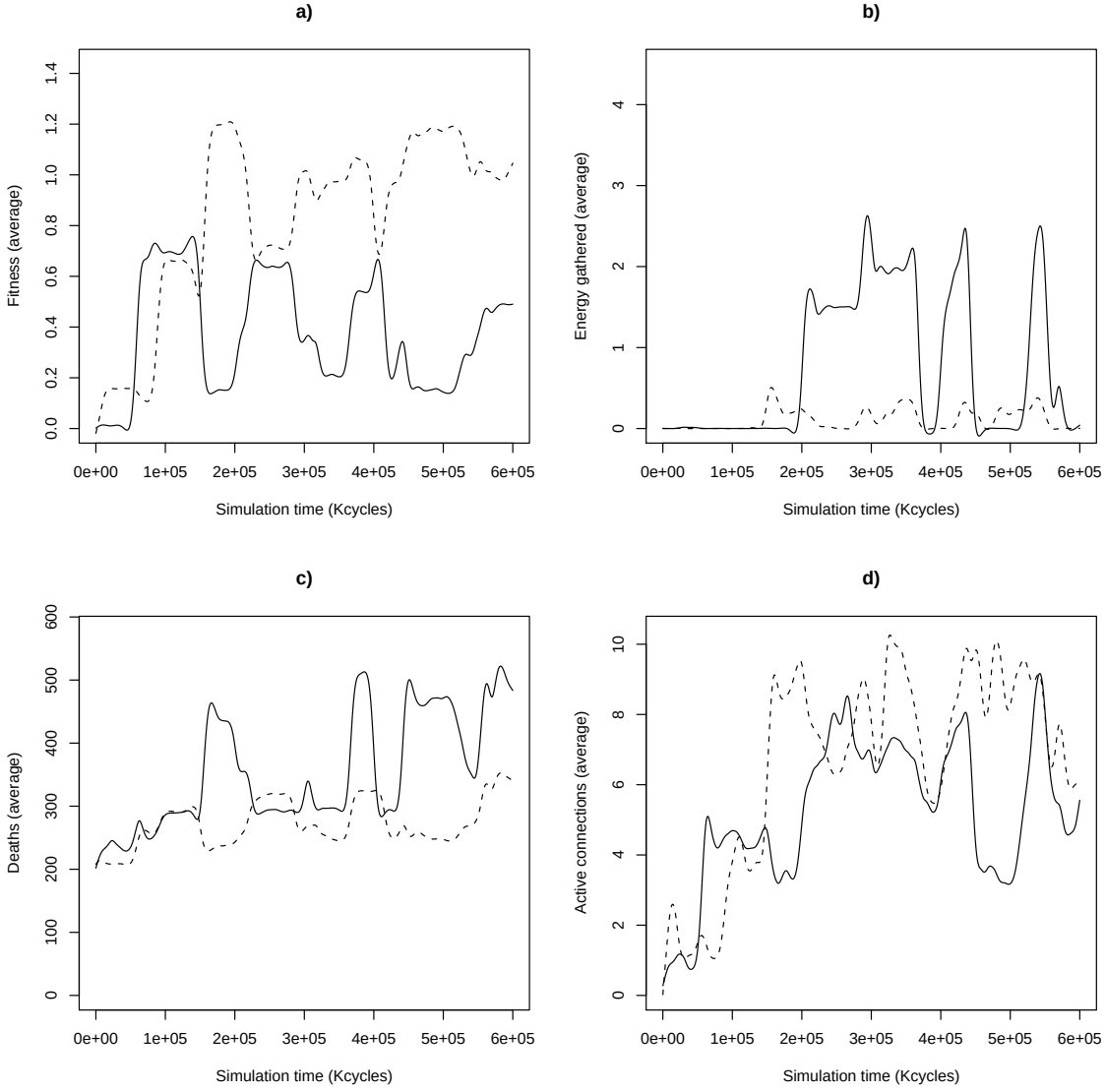

Figure 9.1: Evolutionary history of a battle simulation run. Several metrics are presented for the two species. The blue species is represented in solid line, the red species in dotted line. a) Average fitness; b) Average energy gathered; c) Deaths; d) Average active connections.

Each panel present the evolution of a metric for the two species. The blue species is represented in solid line, while the red species is represented in dotted line. In panel a) it can be observed that both species go through a period of fitness increase until about 1×10^5 Kcycles. From this moment on, a sharp increase in the fitness of the red species affects the performance of the blue species, causing a decline in its fitness. As can be observed in panel c), this relates to the red species gaining the ability to destroy blues agents, and thus causing an increase in the blue species death rate. Until the end, the simulation run goes through periods of convergence and divergence in the fitness of both species. Almost symmetrical fitness curves are created, with the axis of symmetry being an horizontal line coinciding approximately with the 0.6 fitness level. At several moments the fitness of both species becomes almost the same, only to diverge again, with the red species always taking the lead.

This is the first scenario we tested where the average fitness of a species drops significantly during a simulation run, and this is caused by the antagonism of the other evolving species.

In panel b) it can be observed that, for several times, the blue species shows the ability to gather energy by consuming food items. This is an interesting result because the consumption of food items is not directly rewarded by the fitness function. It helps indirectly, because by sustaining higher energy levels, agents are harder to destroy, and by living longer they have a greater chance of producing good shots.

In this simulation run, the two species develop different strategies. The red species relies more on its ability to destroy opponents, while the blue species relies more on increasing its survival chances by consuming food items. This is interesting because both species are defined with the same initial conditions. It is an indication of the possibility of generating diversity through species co-evolution in this type of simulation. It is likely caused by initial, random changes, sending each species in a different evolutionary path. This is consistent with non-linear, complex system behavior. Also, it is a niching phenomenon caused, not by physical separation, but by the dynamics of the system.

In panel d) we can observe that average gridbrain complexity is more unstable than in previous scenarios. Species go through stronger fluctuations in the average number of

active connections. We believe this is cause by the *arms race* [Dawkins and Krebs, 1979] between the two species leading to a more dynamic environment.

In figure 9.2 we present the evolutionary history of another run for the same scenario. The reason we do it is to illustrate that this scenario leads to diverse evolutionary histories. In this second run, it can be observed that the species attain a similar capability of destroying each other, leading to similar fitness curves oscillating around the 0.6 fitness value. The death count average for both similar also increases in a similar fashion and overall there is equilibrium between the species. As can be seen in panel b), the red species goes through a period of greater ability in consuming food items, but to the end both species stabilize their consumption level around similar values.

Overall this scenario indicates the possibility of generating complex and diverse behaviors from simple fitness functions, by exploring the interaction between the fitness function, the environmental basic rules and the dynamics of co-evolution of species.

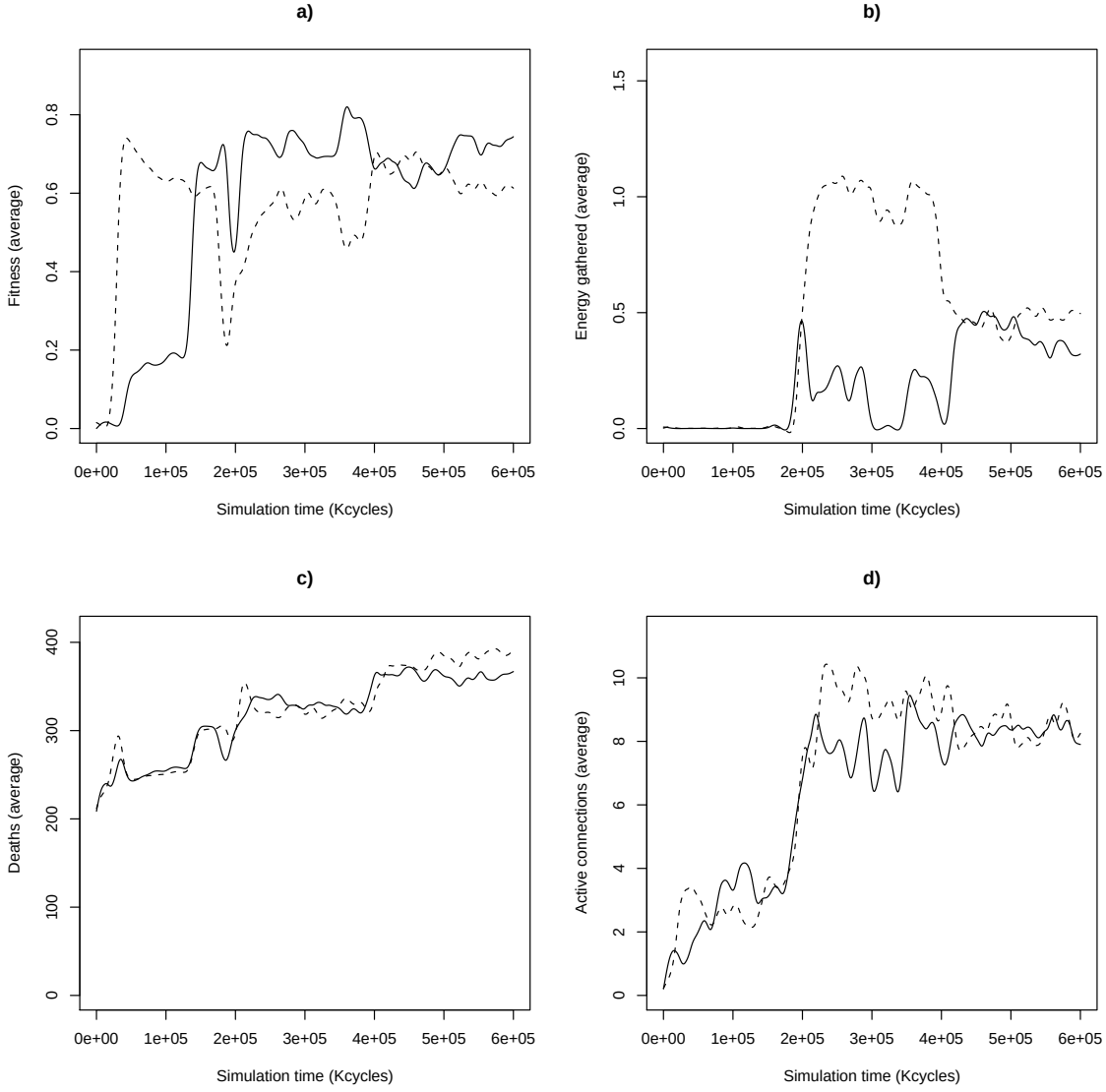

Figure 9.2: Evolutionary history of another battle simulation run. Several metrics are presented for the two species. The blue species is represented in solid line, the red species in dotted line. a) Average fitness; b) Average energy gathered; c) Deaths; d) Average active connections.

Chapter 10

Conclusions and Future Work

In this chapter we will present a summary of the contributions of this work, as well as an overall discussion of the experimental results achieved. Design principles are extracted from our experience in creating evolutionary multi-agent simulations with out framework. We will end with a set of proposals of future work.

10.1 General Discussion

The goal of our work as been to research novel mechanisms to evolve computational intelligence in multi-agent simulations, following ideas from the complexity sciences of favoring bottom-up, decentralized approaches where global behaviors emerge from local interactions. Our work belongs to a current trend of bio-inspiration in computational intelligence, and we looked for new ways to achieve autonomous agent intelligence.

We used abstract concepts inspired from processes found in nature, while adapting them to the digital computer medium.

It is our belief that the goal of the work has been achieved by the conception and experimental validation of an evolutionary framework that consists of three parts: the gridbrain agent brain model, the SEEA algorithm and the LabLOVE simulation environment.

The main concepts behind the gridbrain model are the multi-layered brain approach, with perception layers tied to different sensory channels feeding a decision layer; the ability to process streams of variable-sized sensory information; the brain as a network of compu-

tational building blocks that take advantage of the underlying architecture of the digital computer and the adaptability in terms of complexity, through evolutionary operators, to the demands of the environment.

We consider that all these characteristics of the gridbrain have been demonstrated through experimentation. In fact, as is common in research related to complex systems, the methodology followed was one of experimentation with simulations. To this end, the LabLOVE simulation environment proved to be an invaluable tool. Its development was a considerable part of the effort devoted to the goals of this thesis. It was made available to the scientific community as open source under the GPL license.

In terms of the goal of this work, we consider that our novel approach has shown promising results. In all the experimental scenarios, the evolutionary framework was capable of producing gridbrains that explored the alpha/beta grid duality, generating a sensory layer, made of one or more alpha grids, that fed information to the decision layer, the beta grid. In the *poison* experiment, alpha grids were evolved that used several aggregator components to transform the variable-sized stream of visual sensory data into an unidimensional representation capable of being used by the beta layer to *form* decisions. Components like MAX, AVG and SEL were successfully used for this purpose. In the *targets* experiment, gridbrains with two alpha grids for two different sensory channels, vision and sound, were evolved. The evolutionary framework was shown capable of producing gridbrains that transform sensory information from different channels and feed it to the beta grid, with this grid then combining the received and transformed information from alpha grids to form decisions.

The evolutionary framework was shown capable of combining an heterogeneous set of computational building blocks to form behaviors that are useful in the defined environment. In the several experimental scenarios, different combinations of building blocks were used in the evolved networks. In the *synch* and *targets* scenarios, gridbrains were found to use CLK components in order to temporize actions and synchronize with other agents in the environment. In the *poison* scenario, evolved gridbrains used MEM components to form decisions in the absence of sensory stimulus. Overall, we found that the framework showed adaptability to the challenges of the different environments, with no need for

specific tweaking.

We arrived at a set of genetic operators that where shown to be necessary and sufficient for gridbrain evolution and complexification, in the context of the experiments performed. We understood their respective roles in evolution. The minimal and optimal set of operators we found is: *add/remove connection (mutation)*, *split/join connection (mutation)*, *change inactive component (mutation)*, *recombination* and *formating*. The *change parameter* operators was shown to have no impact on evolution, while the unrestricted *change component* operator was found to be harmful. These two operators were discarded.

Connection level mutations are responsible for performing topological changes on gridbrain networks. These mutations are the ones that ultimately generate computational processes. Each pair of operators performs a different type of topological change, and both were found to be important to the evolutionary process, as experimentation showed that the ablation of one of them decreases the quality of the generated gridbrains. This is not surprising, as *add/remove* creates new connection paths, while *split/join* complexifies existing paths, routing them through intermediary components. *Change inactive component* is responsible for providing the connection level operators with a diversity of components to use in the network. The fact that a too high or low probability for this operator hurts performance is a strong indication that neutral search is taking place. The set of unexpressed components in the genotype is evolving alongside the active network, and there is a synergy between the two.

The formulation of an effective recombination operator for populations of variable-sized networks is not a trivial problem, as is patent in the field of evolutionary neural networks [Yao, 1999]. Only recently have viable recombination operators for neural networks of variable topology been devised [Stanley, 2004]. The same difficulty holds true for limited-sized networks of heterogeneous node types, as is the case of network-based approaches to genetic programming like PDGP and CGP. Only very recently has a viable recombination operator for CGP been proposed [Clegg *et al.*, 2007]. We were able to devise a viable recombination operator for a variable, non-limited size heterogeneous network.

The SEEA algorithm was shown to be effective in creating an evolutionary process embedded in a real-time multi-agent simulation. It is capable of doing so without intro-

ducing intrusive constraints to the simulation, namely the need for disruptive sequences of generations. The *group fitness* extension demonstrated to be a good choice in a general purpose mechanism to stimulate cooperative behavior without hurting performance.

Overall, the framework presented was shown to be both effective in creating an evolutionary process that generates computational brains for agents, while allowing for a great degree of freedom in creating multi-agent simulation scenarios.

One interesting consequence of our work was a new approach to bloat control in evolutionary systems that undergo complexification. Most of the work done in this area has been in the context of tree-based genetic programming [Luke and Panait, 2006; Silva, 2008].

In all the experiments previously presented we initialized the evolutionary runs with a population of agents with zero-sized, empty gridbrains. The evolutionary search caused the gridbrains to increase in complexity in a controlled way, as long as the symmetric connection level mutation operators have equal probabilities and recombination is used. In these cases, we found high correlation between the final fitness value of individuals and their gridbrain complexity, measured in number of connections. Analyzing the evolutionary history of such simulation runs, we observed periods were the complexity of gridbrains was increased in an exploratory fashion, returning to previous levels if no increase in fitness was gained or being sustained if it was fruitful.

Formating is a neutral operator that adapts the grids to the active network they contain. It provides space for further complexification to take place, while bounding the search space. It was found to be effective in doing so. As connection networks, grid dimensions were shown to be adaptable throughout simulation runs.

In contrast to what happens in tree-based genetic programming, recombination was found to be important in controlling bloat. Through it, the evolutionary process is capable of stochastically selecting the best aspects of gridbrains. In its absence, there is not sufficient pressure to discard parts of the genotype that are not useful nor prejudicial. In fact, experiment runs with pure mutation showed the accumulation of such features in the genotype, leading to significant increase in bloat that wastes computational resources.

We attribute this difference to tree-based genetic programming to the different rep-

resentation and genetic operators that we use. While in tree-based GP the search for new solutions is performed mainly by recombination of sub-trees, in the gridbrain the search is performed mainly by connection-level mutations. Through our connection tag mechanisms, the basic units of recombination we use are connection groups. Equivalent connection groups are identified by our tagging mechanism, so our recombination operator constructs a new individual by selecting a version of each functionality from either parent, and possibly incorporating functionalities that are present in only one of them. This, in conjunction with the formatting operator that maintains an adaptive search space, leads to new and different evolutionary dynamics.

A first step towards the creation of artificial life simulations under our framework was made. A simple co-evolutionary experiment with two species showed interesting results. For example, we were able to observe a phenomenon of species specialization without the need for geographical separation. Species with equal initial conditions were shown to diverge to different specialization niches by way of the systems dynamics. Another observation we made in this scenario is the unpredictable nature of evolutionary runs. Co-evolution under our framework shows the potential for generating diverse and surprising agent behaviors.

Overall, the evolutionary framework we propose constitutes a novel approach for the emergence of intelligence from computational building blocks. The ideas we tested with the gridbrain model can be applied outside the realm of multi-agent simulations, with application to the more general problem of generating systems made of simple building blocks through evolutionary complexification.

10.1.1 Comparison with other approaches

In chapters 2 and 3, several approaches that relate to our work and goals have been discussed.

The gridbrain model uses computational building blocks that are close to the base functionalities of the modern digital computer. This idea was followed by the Tierra/Avida artificial life systems. However, the system we propose is less conceptual, and is geared towards the emergence of behaviors in more realistic simulations. Tierra and Avida envi-

ronments are highly conceptual, and were developed with the goal of studying biological processes. Our evolutionary framework was developed with engineering goals in mind.

We achieved results that generate computational constructs that are more complex than the ones achieved by rule based/symbolic approaches. These latter class of approaches, for example *New Ties* or the seminal *Sugar* and all its derivations produce behaviors based on IF/THEN rules. Our approached generated brains that take advantage of boolean and arithmetic operations, information aggregation, clocks and memory cells.

Evolutionary neural network approaches, like *Polyworld* or *NERO* are theoretically capable of the same computational expressiveness as the gridbrain model, due to fully recurrent neural networks being Turing complete. However, the theoretical possibility of computational constructs does not mean that they can be found by the evolutionary process. We showed the ability of our framework to evolve certain types of behaviors that, to our knowledge, have not been achieved with evolutionary multi-agent neural network based system. This is the case of cooperation through synchronization, as demonstrated in the *targets* experiment set.

Another advantage of our model is the ability to interpret the solutions achieved, while neural networks tend to be difficult for human interpretation. This provided us with greater insight on the behavior of the system, and aided in the design of genetic operators and components.

One other novel characteristic of our model is the ability to process information from a variable number of perceived entities, with no need for fixed, predefined preprocessing of this information. This allows the definition of simulation scenarios as entity-to-entity local interactions, which is congruent with the concepts from the field of complexity sciences. Also, it provides for a greater degree of freedom in the way that brains can evolve.

Our grid approach merits comparison with the variants of genetic programming that use similar representations: PDGP and CGP. One important difference in our approach is that the grid dimensions are not fixed, and can adapt their size during the evolutionary complexification process. Another is the use of several grids, for purposes already discussed. Unlike conventional genetic programming, we do not use functions as building

blocks, but components with internal states. This allows us to evolve computational networks that have persistent memory distributed amongst its nodes. Finally, we use different genetic operators. Evolutionary search is based on mutation operators, that perform two basic types of topological modifications on the network (add/remove and split/join connections), both of each having been shown to be necessary. Working with unbounded genotype sizes, we must deal with the problem of bloat, as happens in tree-based genetic programming. The formating operator combined with symmetrical connection level mutations and the recombination operator constitute an innovative approach to bloat control, that we showed experimentally to be effective.

10.1.2 Design principles

The design of experiments to validate our evolutionary framework allowed us to gain some insight on the process of creating multi-agent simulations using the gridbrain model, that we believe to have broader applicability to other complex and evolutionary systems. Although we did not arrive at a formalization of these insights, we attempt to convey them in the form of design principles.

It is interesting to notice that these principles are aligned with current wisdom from complexity sciences.

Simplicity

To arrive at the component set that we use in this work, we experimented with a diversity of components. We found that simpler components are more useful in the evolutionary process, and that it is a mistake to try to endow components with too much internal intelligence. Simpler components have a greater chance of being used in unforeseen ways, as there is a greater degree of freedom in the ways they can be utilized. More complicated components lead to the introduction of human design into a process that, as explained throughout this document, is meant to be as free from explicit control as possible. Furthermore, we realized several times that human intuition does not translate well to the evolutionary process. Designing computational components for evolutionary artificial intelligence is not the same as conceiving a programming language that is friendly to the

human thought processes. For example, we experimented with memory cells with functionalities that are closer to variable manipulation in conventional computer languages. Evolution not only failed to use these cells as we expected, but it found no use for them at all. On the other hand, the very simple memory cell we ended up favoring was successfully used by evolution in ways that we did not expect. There is a trade-off between exposing basic functionality from the medium that supports the simulation, in our case the digital computer, and the internal complexity of the building blocks that expose this functionality. The clock and memory cell components are prime examples of translating digital computer base functionality with very low internal complexity.

Tweaking

Darwinism can be viewed as a process of cumulative tweaking. Complex organism evolve from simpler ones by the accumulation of small tweaks that increase their adaptation. We found this to be an important principle in designing our simulations. Successful components have been the ones that can lead to an increase in fitness of the agent with small deviations from the current network. In the evolutionary process, there is a strong relationship between the functionality of the components and the available genetic operators. For example, the amplifier component is useful because, by way of the split operator, allows for the fine tuning of a behavior that is already present. A connection path that causes an agent to thrust forward can be regulated by an amplifier, so that this thrust is performed with greater or less intensity. The inclusion of clocks in an active path may tweak the rate at which an action is performed. The fact that a clock can use an input signal to synchronize but does not need it, allows for evolution to arrive at a behavior that happens at a certain rate, and then further refine this behavior by adding a synchronization system that uses an output from another part of the gridbrain. Another way to promote tweaking potential is to define components that do not change the behavior of the gridbrain if intercalated in a connection path, but that constitute a target for other incoming connections that can then refine that behavior. This is the case of the AND boolean logic component and of several arithmetic components like MUL and SUM. Even the memory cell adheres to this property as much as possible, not causing a change in

behavior unless zero signals are present. Components should be designed with tweaking potential in mind.

The tweaking principle is also important when defining fitness functions. As with any evolutionary algorithm, the effectiveness of search is related to the fitness landscape generated by the fitness function. If fitness cannot be improved by sequences of small changes, the evolutionary search falls back to brute force (or perhaps worse because stochastic search is not systematic). It is important to formulate the behaviors we want to promote in terms of a function that provides as soft as possible gradients of rewards. One example of this is the *targets* experiment discussed in chapter 8, where the fitness functions is defined as a score that evaluates the quality of a shot. If the function simply rewarded agents for destroying a target, the complexity of the gridbrain needed for the smallest possible increase in fitness would be too great, making it unlikely for solutions to be found.

Scaffolding

We found that the usefulness of components in the evolutionary search cannot be uniquely accessed by their role in the best solutions found after a simulation run. We identified several cases where components provide scaffolding for more complex solutions to be constructed. One common example of this are the NOT or ZERO components, that in the absence of input connections provide constant signals. These components are frequently used in the early stages of evolution to activate behaviors in a simple fashion, later being replaced with more complex constructs. Another example of scaffolding was found in the *poison* experiment, where SEL components where found to be used as MULs in final evolved brains. This can be explained by the specific functionalities of the SEL component being used in earlier stages and becoming obsolete later.

Assumption avoidance

In conventional engineering dealing with linear systems, it is common to think in terms of trade-offs. Intuition can be easily developed on the impact of parameterization on a system. This is not true of the systems we are dealing with in this work. We found several times that the results of parametrization can be much different from what common

sense would indicate. One informal experience we performed was to increase the vision range in the *poison* scenario. Common sense indicates that agents would benefit from the availability of more information. This is not the case. The increased vision range allow agents to have a larger array of food items in sight. More agents see the better food items at the same time, so competition between them increases. Individual-level evolution favors this behavior, which leads to a worse performance overall, as agents disregard acceptable food items and compete for the best. It turns out that sub-optimal behavior serves as a beneficial regulatory mechanism to the species. The introduction of the collective behavior extensions of SEEA may allow them to regulate their behavior with higher view ranges. However, this shows that a simple parameter change can lead to a different system. This case is an interesting opportunity for further research.

10.2 Areas of Application

We believe our work to have application in several areas, namely biological/social simulation research, robotics and virtual environments.

An important current trend in both biology and social sciences is the use of simulation as a research tool. This is related to the new paradigm provided by the inter-disciplinary complexity sciences. Our evolutionary framework is applicable to such research, as it allows for the definition of simulation scenarios were the impact of different parameters may be tested. The readability of gridbrains and LabLOVE visualization aids in the analysis and understanding of the results of such simulations.

Evolutionary robotics is an established area of research, were robot controllers are generated through evolutionary processes. The gridbrain, because of its previously described characteristics, defines a new model for agent controllers that may be applied to robotics. LabLOVE provides a good platform for the definition of realistic physical simulations were the evolution of such robots may be tested. As shown in this work, the framework is amenable for the generation of group behaviors, which can be applied to the generation of robot teams with cooperative behaviors.

Virtual environments are increasingly popular in popular culture, both through video

games and simulated realities, like Second Life. In combination with the Internet, these environments are increasingly rich in interactions between artificial agents and human controlled agents. The framework we propose here may be used in the generation of artificial agents for such worlds.

10.3 Future Work

As is to be expected in the type of work presented in this thesis, many avenues of research were left to be explored. In this section we will present a series of ideas we believe to be worthy of exploration in future work.

The gridbrain model was intentionally kept as simple as possible, so that the impact of its several aspects could be easily tested and analysed. There are two interesting extensions that we considered adding to the model: recurrent connections and connection types.

Recurrent connections could be added to the model with little change to the genetic operators proposed, but the algorithm for a gridbrain computation cycle would have to support them. A simple solution for this is to add a recurrent input interface to the components. This interface would receive values from recurrent connections during a computation cycle and include them in the processing of the following computational cycle. The advantage we see to this extension is the increase in the potentiality for the reuse of gridbrain computations, placing use nearer the modularity of conventional programming languages. In fact, this level of reuse is pursued in several genetic programming models, namely through automatically defined functions in tree-based genetic programming. It is however uncertain how easily can this potential for reuse result in actual modularity though the evolutionary process.

Connection types could allow for components to distinguish incoming connections, treating their input in different ways. This can allow for the definition of interesting components, for example a memory component that treats one type of input as a write instruction and another type as a reset instruction, or a clock that treats one type of input as a synchronization signal and another as a frequency parameter. On the other hand, connection types make the model more complicated, which can lead to unforeseen

problems.

The overall dynamics of the type of systems we studied in this thesis has much left to be explored. The framework proposed and others developed by other researchers are still very far from the ability to generate the level of complexity found in nature. It is important to study the mechanism necessary to foster the evolution of much more complex gridbrains. We believe that it is necessary to study the interplay between fitness functions, component types and operators. One interesting study to be done is the comparison of the complexity of the fitness function with that of the generated gridbrains, possibly through known metrics like *Kolmogorov complexity*.

We have done initial work in applying our evolutionary framework to artificial life simulations. Much explorations is left to be done in this area. Even simple variations of the scenario we presented in chapter 9 may lead to interesting results, for example the use of larger numbers of species.

Another interesting research path is the co-evolution of gridbrains and agent morphologies. In fact, the gridbrain was conceived with this idea in mind. The adaptability of its sensory and actuator layers are a good match to agent models where the morphology of the agent itself is evolving. Consider situations similar to the simulations of Karl Sims. The gridbrain may incorporate actions possible by new muscles or other components in the agent body, as well as perceptions received through new organs, while still allowing for complex computations to be performed in the agent brain.

The gridbrain model may be applied to problems outside the realm of agent control. It is worth investigating the performance of gridbrains in tackling classical genetic programming problems, namely the ones in which grid-based approaches like PDGP and CGP outperform tree-based GP.

We hope that our work can be used by researchers in different areas, and that this results in further insights, analysis and refinements of the framework we proposed.

Bibliography

Adamatzsky, A. and Komosinski, M., 2005. *Artificial Life Models in Software*. Springer-Verlag, London.

Adami, C. and Brown, T., 1994. Evolutionary Learning in the 2D Artificial Life System Avida. In *Proc. Artificial Life IV*, 377–381. MIT Press, Cambridge, MA.

Andersson, O., Armstrong, P., Axelsson, H., and Berjon, R., 2003. Scalable Vector Graphics (SVG) 1.1 Specification. http://www.w3.org/TR/SVG11/.

Angeline, P. J., Saunders, G. M., and Pollack, J. B., 1993. An Evolutionary Algorithm that Constructs Recurrent Neural Networks. *IEEE Transactions on Neural Networks*, **5**(1):54–65.

Axelrod, R. and Cohen, M. D., 1999. *Harnessing Complexity: Organizational Implications of a Scientific Frontier*. The Free Press.

Beck, K., 2003. *Test-Driven Development by Example*. Addison Wesley.

Bedau, M. A., 2003. Artificial Life: Organization, Adaptation, and Complexity from the Bottom Up. *Trends in Cognitive Science*, **7**(11):505–512.

Bell, D., 1974. *The Coming of Post-Industrial Society*. Harper, New York.

Beyer, H. G. and Schwefel, H. P., 2002. Evolution Strategies: A Comprehensive Introduction. *Journal of Natural Computing*, **1**(1):3–52.

Borrelli, F., Ponsiglione, C., Iandoli, L., and Zollo, G., 2005. Inter-organizational Learning and Collective Memory in Small Firms Clusters: An Agent-Based Approach. *Journal of Artificial Societies and Social Simulation*, **8**(3).

Brave, S., 1996. Evolving Deterministic Finite Automata Using Cellular Encoding. In Koza, J., Goldberg, D. E., Fogel, D. B., and Riolo, R. L. (editors), *Genetic Programming 1996: Proc. of the First Annual Conference*, 39–44. MIT Press, Stanford University, CA, USA.

Caillou, P. and Sebag, M., 2008. Modelling a Centralized Academic Labour Market: Efficiency and Fairness. In *Proc. of the 5th European Conference on Complex Systems*. Jerusalem, Israel. URL `http://www.jeruccs2008.org/node/237`.

Camazine, S., Deneubourg, J., Franks, N., Sneyd, J., Theraulaz, G., and Bonabeau, E., 2001. *Self-Organization in Biological Systems*. Princeton University Press.

Chow, S. S., Wilke, C. O., Ofria, C., Lenski, R. E., and Adami, C., 2004. Adaptive Radiation from Resource Competition in Digital Organisms. *Science*, **305**(5680):84–86.

Clegg, J., Walker, J. A., and Miller, J. F., 2007. A New Crossover Technique for Cartesian Genetic Programming. In *GECCO '07: Proc. of the 9th Annual Conference on Genetic and Evolutionary Computation*, 1580–1587. ACM, London, England.

Costa, E. and Simões, A., 2008. *Inteligência Artificial: Fundamentos e Aplicações (2^{nd} ed.)*. FCA - Editora de Informática.

Darwin, C., 1859. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, London. URL `http://www.talkorigins.org/faqs/origin.html`.

Dawkins, R., 1976. *The Selfish Gene*. Oxford University Press.

Dawkins, R. and Krebs, J., 1979. Arms Races Between and Within Species. *Proceedings of the Royal Society of London*, **205**(1161):489–511.

Deffuant, G., 2006. Comparing Extremist Propagation in Continuous Opinion Models. *Journal of Artificial Societies and Social Simulation*, **9**(3):8.

Dewdney, A. K., 1984. In the Game Called Core War Hostile Programs Engage in a Battle of Bits. *Scientific American*, **250**(5):14–22.

Dewdney, A. K., 1988. *The Armchair Universe: An Exploration of Computer Worlds*. W. H. Freeman, New York.

Diamand, R. and Diamand, M., 1996. *A History of Game Theory: From the Beginnings to 1945*. Routledge, Urbana, IL.

Dittrich, P., Ziegler, J., and Banzhaf, W., 2001. Artificial Chemistries - A Review. *Artificial Life*, **7**(3):225–275.

Durr, P., Mattiussi, C., and Floreano, D., 2006. Neuroevolution with Analog Genetic Encoding. *PPSN2006*, **9**:671–680.

Eiben, A. E. and Smith, J. E., 2008. *Introduction to Evolutionary Computing, 2nd edition*. Springer.

Eiben, G., Griffioen, R., and Haasdijk, E., 2007. Population-Based Adaptive Systems: an Implementation in NEW TIES. In *Proc. of ECCS 2007 - European Conference on Complex Systems*. Dresden, Germany. Paper 158 on proceedings CD.

Electronic Arts, 2008. The Sims Celebrates 100 Million Sold Worldwide. http://info.ea.com/news/pr/pr1052.pdf.

Epstein, J. and Axtell, R., 1996. *Growing Artificial Societies: Social Science From the Bottom Up*. Brookings Institution Press and MIT Press.

Floreano, D. and Mattiussi, C., 2008. *Bio-Inspired Artificial Intelligence: Theories, Methods and Technologies*. Universal Music International.

Floreano, D., Mitri, S., Magnenat, S., and Keller, L., 2007. Evolutionary Conditions for the Emergence of Communication in Robots. *Current Biology*, **17**:1–6.

Fogel, L., Owens, A., and Walsh, M., 1966. *Artificial Intelligence through Simulated Evolution*. John Wiley.

Forrest, S. and Jones, T., 1994. Modeling Complex Adaptive Systems with Echo. Working paper 94-12-064, Santa Fe Institute. URL `http://ideas.repec.org/p/wop/safiwp/94-12-064.html`.

Galván-López, E. and Rodríguez-Vázquez, K., 2006. The Importance of Neutral Mutations in GP. *Parallel Problem Solving from Nature - PPSN IX*, **4193**:870–879.

Gathercole, C. and Ross, P., 1996. An Adverse Interaction Between Crossover and Restricted Tree Depth in Genetic Programming. In Koza, J., Goldberg, D. E., Fogel, D. B., and Riolo, R. L. (editors), *Genetic Programming 1996: Proc. of the First Annual Conference*, 291–296. MIT Press, Stanford University, CA, USA.

Gilbert, N., den Besten, M., Bontovics, A., Craenen, B. G. W., Divina, F., Eiben, A. E., Griffioen, A., Hevizi, G., Lorincz, A., Paechter, B., Schuster, S., Schut, M. C., Tzolov, C. K., Vogt, P., and Yang, L., 2006. Emerging Artificial Societies Through Learning. *Journal of Artificial Societies and Social Simulation*, **9**(2). URL `http://ideas.repec.org/a/jas/jasssj/2005-53-2.html`.

Gödel, K., 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *I. Monatshefte für Mathematik und Physik*, **38**:173–198.

Gomez, F. and Miikkulainen, R., 1998. 2-D Pole-Balancing with Recurrent Evolutionary Networks. In *Proc. of the International Conference on Artificial Neural Networks*, 425–430. Springer-Verlag, New York.

Hamilton, W. D., 1963. The Evolution of Altruistic Behavior. *American Naturalist*, **97**(896):354–356.

Hamilton, W. D., 1964. The Genetical Evolution of Social Behaviour. *Journal of Theoretical Biology*, **7**(1):1–16 and 17–52.

Harding, S. and Miller, J. F., 2005. Evolution of Robot Controller Using Cartesian Genetic Programming. In *Genetic Programming*, 62–73. Springer Berlin / Heidelberg, Cambridge, MA.

Hebb, D. O., 1961. Distinctive Features of Learning in the Higher Animal. In Delafresnave, J. F. (editor), *Brain Mechanisms and Learning*, 37–46. Oxford University Press.

Hoffman, W. and Martin, K., 2003. The CMake Build Manager - Cross Platform and Open Source. *Dr. Dobbs*. URL `http://www.ddj.com/cpp/184405251`.

Holland, J. H., 1975. *Adaptation in Natural and Artificial Systems.* Univerisity of Michigan Press, Ann Arbor, MI.

Holland, J. H., 1995. *Hidden Order - How Adaptation Builds Complexity.* Addison-Wesley.

Holland, J. H., 1999. Echoing Emergence: Objectives, Rough Definitions, and Speculations for Echo-class models. In Cowan, G. A., Pines, D., and Meltzer, D. (editors), *Complexity: Metaphors, Models and Reality*, 309–342. Perseus Books, Cambridge, MA.

Hraber, P. and Milne, B. T., 1997. Community Assembly in a Model Ecosystem. *Ecological Modelling*, **103**:267–285.

Hraber, P. T., Jones, T., and Forrest, S., 1997. The Ecology of Echo. *Artificial Life*, **3**(3):165–190.

Hughes, W. O. H., Oldroyd, B. P., Beekman, M., and Ratnieks, F. L. W., 2008. Ancestral Monogamy Shows Kin Selection Is Key to the Evolution of Eusociality. *Science*, **320**(5880):1213–1216.

Huxley, J., 1942. *Evolution: The Modern Synthesis.* Allen & Unwin, London.

Ierusalimschy, R., Figueiredo, L. H., and Celes, W., 2006. *Lua 5.1 Reference Manual.* Lua.org.

Igel, C., 2003. Neuroevolution for Reinforcement Learning using Evolution Strategies. In *Proc. of the Congress on Evolutionary Computation 2003 (CEC 2003)*, 2588–2595. IEEE Press, Piscataway, NJ.

Izquierdo, S. and L.R., I., 2006. The Impact on Market Efficiency of Quality Uncertainty Without Asymmetric Information. *Journal of Business Research*, **60**(8):858–867.

James, D. and Tucker, P., 2004. A Comparative Analysis of Simplification and Complexification in the Evolution of Neural Network Topologies. In *Proc. of the 2004 Conference on Genetic and Evolutionary Computation (GECCO-2004).* Seattle, WA.

Jones, T. and Forrest, S., 1993. An Introduction to SFI Echo. Technical report 93-12-074, Santa Fe Institute.

Khan, G., Halliday, D., and Miller, J., 2001. Breaking the Synaptic Dogma: Evolving a Neuro-Inspired Developmental Network. In *Proc. of the 7th International Conference on Simulated Evolution and Learning (SEAL)*, 11–20. LNCS.

Khan, G., Halliday, D., and Miller, J., 2007. Coevolution of Intelligent Agents using Cartesian Genetic Programming. In *Proc. of Genetic and Evolutionary Computation Conference*, 269–276. ACM, New York, NY, USA.

Khan, G., Halliday, D., and Miller, J., 2008a. Coevolution of Neuro-Developmental Programs the Play Checkers. *Evolvable Systems: From Biology to Hardware*, **5216**:352–361.

Khan, G., Halliday, D., and Miller, J., 2008b. Emergent Instinctive Behavior in Evolved Neuro-Inspired Agents. In *Proc. of Modelling Adaptive and Cognitive Systems (AdapCOG), at SBIA/SBRN/JRI Joint Conference.*

Kimura, M., 1983. *The Neutral Theory of Molecular Evolution.* Cambridge University Press, Cambridge.

Koza, J. R., 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA.

Koza, J. R., 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA.

Langdon, W. B., 1998. The Evolution of Size in Variable Length Representations. In *Proc. of the 1998 IEEE International Conference on Evolutionary Computation*, 633–638. IEEE Press, Anchorage, Alaska.

Langton, C., 1989. *Artificial Life.* Addison-Wesley.

Lenski, R. E., Ofria, C., Collier, T. C., and Adami, C., 1999. Genomic Complexity, Robustness, and Genetic Interactions in Digital Organisms. *Nature*, **400**:661–664.

Lenski, R. E., Ofria, C., Pennock, R. T., and Adami, C., 2003. The Evolutionary Origin of Complex Features. *Nature*, **423**:139–145.

Linsker, R., 1987. Towards an Organizing Principle for a Layered Perceptual Network. In Anderson, D. Z. (editor), *Neural Information Processing Systems*, 485–494. American Institute of Physics, New York.

Linsker, R., 1988. Self-Organization in a Perceptual Network. *Computer*, **21**(3):105–117.

Lorenz, E. N., 1963. Deterministic Nonperiodic flow. *Journal of Atmospheric Sciences*, **20**:130–141.

Lotka, A. J., 1956. *Elements of Mathematical Biology*. Dover, New York.

Luke, S. and Panait, L., 2006. A Comparison of Bloat Control Methods for Genetic Programming. *Evolutionary Computation*, **14**(3):309–344.

Mattiussi, C., 2005. *Evolutionary Synthesis of Analog Networks*. Laboratory of Intelligent System (LIS) - EPFL - Lausanne, Switzerland.

McIndoe, B. W., 2005. *jECHO: A New Implementation of Holland's Complex Adaptive Systems Model With the Aim of Seeking Answers to Some Fundamental Open Questions*. University of Liverpool.

Mendel, G., 1865. *Experiments in Plant Hybridization*. URL `http://www.mendelweb.org/Mendel.html`.

Menezes, T., 2008a. Artist Graphics Library. http://sourceforge.net/projects/artist.

Menezes, T., 2008b. LabLOVE - Laboratory of Life On a Virtual Environment. http://telmomenezes.com/lablove.

Menezes, T., Baptista, T., and Costa, E., 2006. Towards Generation of Complex Game Worlds. In *Proc. of the IEEE Symposium on Computational Intelligence and Games (CIG'06)*, 224–229. Reno, NV.

Menezes, T. and Costa, E., 2006. A First Order Language to Coevolve Agents in Complex Social Simulations. In *Proc. of the European Conference on Complex Systems 2006*. Oxford, UK. URL `http://complexsystems.lri.fr/FinalReview/FILES/PDF/p146.pdf`.

Menezes, T. and Costa, E., 2007a. Designing for Surprise. In *Advances in Artificial Life - 9th European Conference on Artificial Life (ECAL2007)*, 1079–1088. Lisbon, Portugal.

Menezes, T. and Costa, E., 2007b. The Gridbrain: an Heterogeneous Network for Open Evolution in 3D Environments. In *Proc. of the The First IEEE Symposium on Artificial Life*, 155–162. IEEE, Honolulu, USA.

Menezes, T. and Costa, E., 2008a. Artificial Brains as Networks of Computational Building Blocks. In *Proc. of the 5th European Conference on Complex Systems*. Jerusalem, Israel. URL `http://www.jeruccs2008.org/node/606`.

Menezes, T. and Costa, E., 2008b. Modeling Evolvable Brains - An Heterogeneous Network Approach. *International Journal of Information Technology and Intelligent Computing*, **2**(2).

Miller, J. and Smith, S., 2006. Redundancy and Computational Efficiency in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation*, **10**(2):167–174.

Miller, J. F., 1999. An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach. In *GECCO 1999: Proc. of the Genetic and Evolutionary Computation Conference*, 1135–1142. Morgan Kaufmann, Orlando, Florida.

Miller, J. F. and Thomson, P., 2000. Cartesian Genetic Programming. In *Proc. of the 3rd European Conference on Genetic Programming*, 121–132. Springer Verlag Berlin, Edinburgh.

Moriarty, D. E., 1997. *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. Department of Computer Sciences, The University of Texas at Austin.

Ofria, C., Adami, C., and Collier, T. C., 2003. Selective Pressures on Genomes in Molecular Evolution. *Journal of Theoretical Biology*, **222**(4):477–483.

Ofria, C., T., B. C., and Adami, C., 1997. *The Avida Technical Manual*. http://www.krl.caltech.edu/ charles/avida/manual/.

Ofria, C. and Wilke, C. O., 2004. Avida: A Software Platform for Research in Computational Evolutionary Biology. *Journal of Artificial Life*, **10**:191–29.

Palmer, R., Arthur, W., Holland, J., LeBaron, B., and Taylor, P., 1994. Artificial Economic Life: a Simple Model of a Stockmarket. *Physica D*, **75**:264–274.

Paulsen, T. J., O.; Sejnowski, 2000. Natural Patterns of Activity and Long-Term Synaptic Plasticity. *Current Opinion in Neurobiology*, **10**:172–179.

Penrose, R., 1991. *The Emperor's New Mind*. Penguin (Non-Classics).

Poli, R., 1996. Parallel Distributed Genetic Programming. Technical report CSRP-96-15, The University of Birmingham, UK.

Poli, R., 1999. Parallel Distributed Genetic Programming. In D. Corne, e. a. (editor), *Optimization, Advanced Topics in Computer Science*, chapter 27, 403–431. McGraw-Hill, Maidenhead, Berkshire, England.

Poli, R., Langdon, W. B., and McPhee, N. F., 2008. *A Field Guide to Genetic Programming*. Lulu Enterprises, Ltd., UK.

Preston, F. W., 1948. The Commonness, and Rarity, of Species. *Ecology*, **29**(3):254–283.

Ray, T. S., 1992. Evolution, Ecology and Optimization of Digital Organisms. Working paper 92-08-042, Santa Fe Institute.

Russel, S. and Norvig, P., 2002. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Samuel, A., 1959. Some Studies in Machine Learning Using the Game of Checkers. *IBM J. Res. Dev.*, **3**(3):210–219.

Sayama, H., 1998. Introduction of Structural Dissolution into Langton's Self-Reproducing Loop. In *Proc. of the Sixth International Conference on Artificial Life*, 114–122. MIT Press, Los Angeles, California.

Schelling, T., 1971. Dynamic Models of Segregation. *Journal of Mathematical Sociology*, **1**:143–186.

Schiff, J. L., 2008. *Cellular Automata: A Discrete View of the World*. Wiley & Sons, Inc.

Schmitz, O. J. and Booth, G., 1996. Modeling Food Web Complexity: The Consequence of Individual-Based Spatially Explicit Behavioral Ecology on Trophic Interactions. *Evolutionary Ecology*, **11**(4):379–398.

Schut, M., 2007. *Scientific Handbook for Simulation of Collective Intelligence*. http://www.sci-sci.org (Creative Commons).

SecondLife, 2008. Second Life. http://www.secondlife.com.

Siebel, N. T. and Sommer, G., 2007. Evolutionary Reinforcement Learning of Artificial Neural Networks. *International Journal of Hybrid Intelligent Systems*, **4**(3):171–183.

Silva, S., 2008. *Controlling Bloat: Individual and Population Based Approaches in Genetic Programming*. Departamento de Engenharia Informatica, Faculdade de Ciencias e Tecnologia, Universidade de Coimbra.

Smart, J., Cascio, J., and Paffendorf, J., 2007. Metaverse Roadmap Overview. http://www.metaverseroadmap.org.

Smith, M. J., 1964. Group Selection and Kin Selection. *Nature*, **201**:1145–1147.

Smith, R. and Bedau, M. A., 2000. Is Echo a Complex Adaptive System? *Evolutionary Computation*, **8**(4):419–442.

Standish, K. R., 2003. Open-Ended Artificial Evolution. *International Journal of Computational Intelligence and Applications*, **3**(2):167–175.

Stanley, K. O., 2004. *Efficient Evolution of Neural Networks Through Complexification*. Department of Computer Sciences, The University of Texas at Austin.

Stanley, K. O., Bryant, B. D., Karpov, I., and Miikkulainen, R., 2006. Real-Time Evolution of Neural Networks in the NERO Video Game. In *Proc. of the Twenty-First National Conference on Artificial Intelligence (AAAI-2006)*, 1671–1674. AAAI Press, Meno Park, CA.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R., 2005. Evolving Neural Network Agents in the NERO Video Game. In *Proc. of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG'05)*. IEEE, Piscataway, NJ.

Stanley, K. O. and Miikkulainen, R., 2004. Competitive Coevolution through Evolutionary Complexification. *Journal of Artificial Intelligence Research*, **21**:63–100.

Steels, L., 2004. The Evolution of Communication Systems by Adaptive Agents. In Alonso, D. K., E. and Kazakov, D. (editors), *Adaptive Agents and Multi-Agent Systems*, volume 2636 of *Lecture Notes in AI*, 125–140. Springer Verlag, Berlin.

Tavares, J., 2007. *Evolvability in Optimization Problems: the Role of Representations and Heuristics*. Universidade de Coimbra.

Taylor, F. W., 1911. *The Principles of Scientific Management*. Harper Bros., New York.

Tesfatsion, L., 2002. Growing Economies From the Bottom Up. *Artificial Life*, **8**(1):55–82.

Thrun, S., Bala, J., Bloedorn, E., Bratko, I., Cestnik, B., Cheng, J., Jong, K., Dzeroski, S., Hamann, R., Kaufman, K., Keller, S., Kononenko, I., Kreuziger, J., Michalski, R. S., Mitchell, T., Pachowicz, P., Roger, B., Vafaie, H., Velde, W., Wenzel, W., Wnek, J., and Zhang, J., 1991. The MONK's Problems: A Performance Comparison of Different Learning Algorithms. Technical Report CMU-CS-91-197, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

Toffler, A., 1980. *The Third Wave*. Bantam Books.

Turing, A., 1950. Computing Machinery and Intelligence. *Mind*, **59**:433–460.

Vassilev, V. and Miller, J., 2000. The Advantages of Landscape Neutrality in Digital Circuit Evolution. In *Proc. of the 3rd International Conference of Evolvable Systems: From Biology to Hardware*, 252–263. Springer LNCS.

von Neumann, J., 1945. First Draft of a Report on the EDVAC. Technical report, Moore School of Electrical Engineering, University of Pennsylvania.

von Neumann, J., 1966. *The Theory of Self-reproducing Automata*. Univ. of Illinois Press.

Walker, J., Miller, J., and Cavill, R., 2006. A Multi-Chromosome Approach to Standard and Embedded Cartesian Genetic Programming. In *Proc. of the 2006 Genetic and Evolutionary Computation Conference (GECCO 2006)*, 903–910. ACM Press.

Walker, J. A. and Miller, J. F., 2004. Evolution and Acquisition of Modules in Cartesian Genetic Programming. In *Genetic Programming*, 187–197. Springer Berlin / Heidelberg, Cambridge, MA.

Walker, J. A. and Miller, J. F., 2005. Investigating the Performance of Module Acquisition in Cartesian Genetic Programming. In *GECCO '05: Proc. of the 2005 conference on Genetic and Evolutionary Computation*, 1649–1656. ACM, Washington DC, USA.

Watson, R. A., Hornby, G., and Pollack, J. B., 1998. Modeling Building-Block Interdependency. In *PPSN V: Proc. of the 5th International Conference on Parallel Problem Solving from Nature*, 97–108. Springer-Verlag, London, UK.

Wilke, C. O., Wang, J. L., Ofria, C., Lenski, R. E., and Adami, C., 2001. Evolution of Digital Organisms at High Mutation Rate Leads to Survial of the Flattest. *Nature*, **412**:331–333.

Williams, G. C., 1966. *Adaptation and Natural Selection: A Critique of Some Current Evolutionary Thought*. Princeton University Press.

Wilson, D. S., 1987. Altruism in Mendelian Populations Derived from Sibling Groups: The Haystack Model Revisited. *Evolution*, **5**:1059–1070.

Wilson, D. S., 2005. Kin Selection as the Key to Altruism: its Rise and Fall. *Social Research*, **72**(1):159–166.

Wilson, E. O., 1971. *The Insect Societies*. Belknap Press of Harvard University Press, Cambridge, Massachusetts.

Wooldridge, M. and Jennings, N. R., 1995. Intelligent Agents: Theory and practice. *Knowledge Engineering Review*, **10**(2):115–152.

WorldOfWarcraft, 2008. World of Warcraft. http://www.worldofwarcraft.com.

Yaeger, L., 1994. Computational Genetics, Physiology, Metabolism, Neural Systems, Learning, Vision, and Behavior or PolyWorld: Life in a New Context. In *Artificial Life III, Proc. Volume XVII*, 263–298. Addison-Wesley, Reading, Massachusetts.

Yaeger, L. and Sporns, O., 2006. Evolution of Neural Structure and Complexity in a Computational Ecology. In *Artificial Life X*, 330–336. MIT Press, Cambridge, MA.

Yao, X., 1999. Evolving Artificial Neural Networks. *Proceedings of the IEEE*, **87**(9):1423–1447.

Yu, T. and Miller, J., 2001. Neutrality and Evolvability of a Boolean Function Landscape. In *Proc. of the 4th European Conference on Genetic Programming (EuroGP2001)*, 204–217. Springer LNCS.

Yu, T. and Miller, J., 2006. Through the Interaction of Neutral and Adaptive Mutations Evolutionary Search Finds a Way. *Artificial Life*, **12**(4):525–551.