RENATO RESENDE RIBEIRO DE OLIVEIRA

PROGRAMAÇÃO GENÉTICA APLICADA À GERAÇÃO AUTOMATIZADA DE APLICAÇÕES PARA REDES DE SENSORES SEM FIO

LAVRAS - MG

2014

RENATO RESENDE RIBEIRO DE OLIVEIRA

PROGRAMAÇÃO GENÉTICA APLICADA À GERAÇÃO AUTOMATIZADA DE APLICAÇÕES PARA REDES DE SENSORES SEM FIO

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Redes de Computadores e Sistemas Embarcados, para a obtenção do título de Mestre.

Orientador

Dr. Tales Heimfarth

Coorientador

Dr. Raphael Winckler de Bettio

LAVRAS - MG

2014

Ficha Catalográfica Elaborada pela Coordenadoria de Produtos e Serviços da Biblioteca Universitária da UFLA

Oliveira, Renato Resende Ribeiro de.
Programação genética aplicada à geração automatizada de aplicações para redes de sensores sem fio / Renato Resende Ribeiro de Oliveira. – Lavras : UFLA, 2014.
71 p. : il.

Dissertação (mestrado) – Universidade Federal de Lavras, 2014.
Orientador: Tales Heimfarth.
Bibliografia.

1. Redes de sensores sem fio. 2. Programação genética. 3. Algoritmo genético. 4. Middlewares. I. Universidade Federal de Lavras. II. Título.

CDD – 004.6

RENATO RESENDE RIBEIRO DE OLIVEIRA

PROGRAMAÇÃO GENÉTICA APLICADA À GERAÇÃO AUTOMATIZADA DE APLICAÇÕES PARA REDES DE SENSORES SEM FIO

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Redes de Computadores e Sistemas Embarcados, para a obtenção do título de Mestre.

APROVADA em 24 de janeiro de 2014.

Dr. Raphael W. de Bettio	UFLA
Dr. Luiz Henrique A. Correia	UFLA
Dr. Cláudio Fabiano M. Toledo	USP

Dr. Tales Heimfarth

Orientador

LAVRAS - MG

2014

Dedico esta dissertação à minha amada esposa Ana Cristina, que me apoiou plenamente durante toda esta jornada.
Me acompanhou durante todas as noites sem dormir e as correrias de prazos e deadlines...

AGRADECIMENTOS

Agradeço ao professor Cláudio, que apesar de estar longe continua me orientando e aconselho até hoje.

Agradeço aos meu orientador professor Tales e ao meu coorientador professor Raphael pelas dicas, conselhos e a plena disponibilidade.

Agradeço meus pais, que apesar das dificuldades sempre me apoiaram e me deram condições para chegar até aqui.

Agradeço ao meu tio professor João Chrysostomo pelo amparo e apoio desde que cheguei em Lavras com 18 anos.

Agradeço à todos os meus colegas do PPGCC da UFLA pela ajuda durante as disciplinas do mestrado.

Agradeço à todos os professores do PPGCC da UFLA pelas instruções e conhecimentos passados.

Agradeço à secretaria do PPGCC da UFLA pela pronta disponibilidade.

Agradeço à FAPEMIG pela bolsa de estudos concedida.

RESUMO

A programação de redes de sensores sem fio (RSSF) é uma tarefa complexa devido à programação em linguagens de baixo nível e à necessidade de uma aplicação distinta para cada sensor. Além disso, sensores sem fios possuem grandes limitações de hardware, como baixo poder de processamento, pouca memória e limitação energética. Portanto, a programação automática de RSSF é desejável, uma vez que pode-se contemplar essas dificuldades automaticamente, além de economizar em custos, pois elimina a necessidade de alocar um desenvolvedor para programar a RSSF. A geração automática de códigos-fonte para RSSF utilizando programação genética foi pouco estudada na literatura até o momento. A programação genética mostrou-se promissora na geração de código em diversas áreas de aplicação. Dessa forma, o presente estudo propõe o desenvolvimento e a aplicação de algoritmos evolutivos para gerar e evoluir códigos-fonte que solucionem problemas de RSSF. O objetivo é que os algoritmos evolutivos desenvolvidos sejam capazes de resolver problemas distintos de RSSF de forma correta (atendendo o objetivo geral do problema) e com uma eficiência satisfatória (principalmente no quesito energia gasta pelos nós sensores). Os resultados obtidos mostram que a ferramenta é capaz de solucionar de maneira ótima o Problema de Detecção de Eventos para RSSF com topologia em grade e de forma satisfatória para RSSF com topologia randômica. Sendo assim, o presente estudo traz contribuições para a área de RSSF, uma vez que a programação automática de RSSF reduz consideravelmente a mão de obra humana na programação das mesmas, além de reduzir os custos da realização desta tarefa.

Palavras-chave: Redes de Sensores sem Fio. Middlewares. Programação Genética. Algoritmo Genético.

ABSTRACT

The wireless sensor networks (WSN) programming is a complex task due to the low-level programming languages and the need of a specific application for each sensor. Furthermore, wireless sensors have many hardware limitations such as low processing power, small memory and energetic limitations. Hence, the automatic programming of WSNs is desirable since it can automatically address these difficulties, besides saving costs by eliminating the need to allocate a developer to program the WSN. The automatic code generation for WSNs using genetic programming has been poorly studied in the literature so far. The genetic programming has proved to be promising in code generation for many application areas. This study proposes the development and application of evolutionary algorithms to generate source codes that solve WSNs problems. The developed evolutionary algorithms should be able to solve different problems of WSNs correctly (achieve the main goal of the problem) and with satisfatory efficiency (mainly on energy savings). The obtained results show that the proposed framework is able to find optimal solutions for the Event Detection Problem for WSN with grid topology and to find satisfatory solutions for WSN with randomized topology. Thus, this study brings many contributions to the WSN area since the automatic programming of WSNs drastically reduces the human programming effort, besides saving costs on executing this task.

Keywords: Wireless Sensor Networks. Middlewares. Genetic Programming. Genetic Algorithm.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

AG	Algoritmo Genético
GRUBI	Grupo de Redes Ubíquas
MV	Máquina Virtual
MW	*Middleware*
PDE	Problema de Detecção de Eventos
PG	Programação Genética
RSSF	Redes de Sensores sem Fio
UFLA	Universidade Federal de Lavras
USP	Universidade de São Paulo

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização e Motivação

Redes de sensores sem fios (RSSF) são redes compostas por vários nós sensores. Esses nós sensores são compostos por um processador de baixo poder de processamento, um rádio de baixa potência e memória de tamanho reduzido. Eles são alimentados por uma bateria e por isso possuem limitações energéticas na sua utilização. Além disso, existem várias limitações e dificuldades no projeto de uma RSSF, como a programação em baixo nível dos nós sensores, a especificidade do comportamento de cada nó sensor, entre outras questões que afetam sistemas massivamente distribuídos.

As RSSF são muito versáteis. Além do custo de implantação relativamente reduzido, elas possibilitam o monitoramento e mapeamento de regiões de alto risco. Pode-se utilizar RSSF para monitoramentos sismográficos em vulcões, monitoramento de temperatura e pressão em plataformas petrolíferas, mapeamento de florestas e de regiões hostis (YANG, 2010).

Apesar da versatilidade das RSSF, a programação da rede é uma tarefa complexa. A aplicação deve ser desenvolvida e customizada para cada nó sensor da rede, uma vez que o comportamento de cada nó depende de fatores específicos como posição geográfica, estado do nó, função do sensor, entre outros. Essa heterogeneidade das RSSF torna o processo do desenvolvimento e da programação dos nós sensores da rede uma tarefa árdua. Portanto, a programação e configuração automática de aplicações para RSSF é um recurso muito desejável, uma vez que diminui drasticamente o esforço humano na programação de aplicações em RSSF. Assim, a contribuição desse trabalho será o desenvolvimento de um *framework* que

proporcionará a programação automática de aplicações para RSSF, facilitando a utilização desse tipo de rede nos diversos problemas do mundo real. Além disso, na literatura atual se encontram poucos trabalhos que aplicam a PG com o intuito de programar uma RSSF como um todo. Dessa forma, o presente trabalho investiga novos aspectos da utilização de PG em RSSF.

1.2 Objetivos Gerais e Específicos

O objetivo geral do presente trabalho é desenvolver um *framework* capaz de gerar, de forma automática, o código-fonte de aplicações para RSSF. Esse *framework* é composto por três camadas: Um simulador de RSSF, um *middleware* que provê uma linguagem *script* de alto nível para programação de aplicações para RSSF e um método baseado em programação genética para gerar automaticamente aplicações para este middleware.

Os seguintes objetivos específicos foram estabelecidos:

- Desenvolver uma linguagem *script* que permita a descrição de aplicações para RSSF. Essa linguagem deve ser expressiva o suficiente para permitir o desenvolvimento de diferentes tipos de aplicações.
- Desenvolver um *middleware* que seja capaz de interpretar essa linguagem *script* e possibilite a programação de RSSF em alto nível.
- Adaptar o simulador GRUBIX (GRUBI, 2013) para realizar simulações de RSSF mais rápidas e simplificadas, uma vez que as simulações dos *scripts* serão realizadas de forma intensiva no *framework* proposto.
- Desenvolver diferentes algoritmos de programação genética para modelar o objetivo geral de uma RSSF e gerar automaticamente aplicações que visam alcançar esse objetivo.

- Integrar os três módulos (módulo de simulação, *middleware*, módulo de geração de aplicações) e desenvolver um *framework* para geração automática de aplicações para RSSF.
- Desenvolver um gerador de instâncias para testes do *framework* proposto. Esse gerador deve gerar instâncias que representem um problema de detecção de eventos utilizando RSSF.
- Criar um *benchmark* de instâncias para testes utilizando o gerador desenvolvido. Essas instâncias devem contemplar diferentes tamanhos e características de uma RSSF.
- Avaliar o desempenho do *framework* proposto ao solucionar o *benchmark* de instâncias criado.

1.3 Metodologia

Um *middleware* específico para facilitar a geração automática de aplicações para RSSF foi desenvolvido. Esse *middleware* provê uma linguagem de programação *script* que possibilita a programação das RSSF em alto nível e a reprogramação dos nós sensores de forma eficiente. Esse *middleware* é incorporado à um módulo de simulação, adaptado do simulador GRUBIX (GRUBI, 2013), para proporcionar um ambiente de programação de RSSF em alto nível com suporte a simulações dessas aplicações. Este *middleware* recebeu o nome de **Odin**.

Um módulo de Programação Genética foi construído contendo diversos algoritmos para realização da programação automática de aplicações para RSSF. Foi desenvolvido um algoritmo baseado em Algoritmos Genéticos (AG). Este módulo de PG utiliza-se de uma função objetivo que deve ser definida pelo projetista da RSSF. Através dessa função objetivo, o módulo é

capaz de gerar populações de programas e evoluí-los por meio de operadores genéticos (como mutações e recombinações). Esse módulo de PG recebeu o nome de **GeneticNet**.

A integração do módulo GeneticNet com o ambiente de simulação composto pelo *middleware* Odin e o módulo de simulação forma um *framework* específico para geração automática de aplicações para RSSF. Este *framework* foi testado e avaliado analisando o desempenho das aplicações geradas para solucionar problemas de detecção de eventos utilizando RSSF.

1.4 Trabalhos Publicados

Os resultados obtidos neste estudo foram publicados em eventos científicos de ampla divulgação:

- The 2013 IEEE Congress on Evolutionary Computation, Cancún, México (OLIVEIRA et al., 2013a).
- Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, Paddeborn, Alemanha (HEIMFARTH et al., 2013).
- III Workshop de Sistemas Distribuídos Autonômicos, Brasília, Brasil (OLIVEIRA et al., 2013b).

1.5 Organização do Trabalho

O trabalho está organizado da seguinte forma, na Seção 2 será apresentado um referencial teórico acerca dos conceitos utilizados. Na Seção 3 será apresentada a metodologia do trabalho desenvolvido. Na Seção 4 serão apresentadas as simulações computacionais realizadas e os resultados obtidos. As conclusões e os trabalhos futuros serão apresentadas na Seção 5.

2 REFERENCIAL TEÓRICO

2.1 Redes de Sensores sem Fio

As Redes de Sensores sem Fio (RSSF) são uma tecnologia recente que tem crescido muito nos últimos anos. As RSSF são compostas por dispositivos eletrônicos (nós sensores) independentes que possuem capacidade de comunicação sem fio (*wireless*). Os nós sensores possuem capacidade de processamento computacional, comunicação sem fio, sensores que são capazes de realizar medições no ambiente (sismógrafos, termômetros, barômetros, entre outros) e atuadores que podem interagir com o ambiente. Esses nós sensores normalmente são alimentados por baterias, o que torna o consumo energético uma questão chave neste tipo de tecnologia. Esse tipo de tecnologia de sistema embarcado tem sido utilizada como uma forma de computação ubíqua, sendo aplicada em diversas áreas da vida humana, como agricultura, monitoramento, rastreamento e até mesmo para controlar componentes domésticos simples, como luzes, cortinas, eletrodomésticos, entre outros (KARL; WILLIG, 2005). A computação ubíqua é definida como uma forma onipresente de computação, é um conjunto de aparelhos e dispositivos eletrônicos que realizam computação de forma transparente e imperceptível para os usuários (HANSMANN et al., 2003).

Apesar de cada nó sensor da rede possuir a capacidade de processamento e sensoriamento de forma individual, em aplicações de RSSF um nó sensor não é capaz de atingir o objetivo da rede individualmente. É necessário que os nós sensores cooperem entre si, através de comunicação sem fio, para que consigam alcançar o objetivo da RSSF. Essa característica acarreta em diversas dificuldades no desenvolvimento do software que con-

trolará a operação da RSSF. A flexibilidade e a heterogeneidade das RSSF requerem que o software de controle seja capaz de tratar a abstrair as diferenças e características de cada nó sensor individual. É necessário que este software seja capaz de tratar a RSSF como um todo para que o objetivo seja alcançado.

Devido à estas características especiais das RSSF, diversas pesquisas são realizadas no intuito desenvolver arquiteturas e protocolos que sejam mais adequados às necessidades deste tipo rede. Akyildiz et al. (2002) realizaram uma revisão da literatura englobando todas as áreas de pesquisa em RSSF. Diversos protocolos e arquiteturas já desenvolvidas para RSSF são detalhados por Karl e Willig (2005). Uma revisão mais recente da literatura é realizada por Guo e Zhang (2014), que focam suas análises em protocolos inteligentes para RSSF.

2.2 Simuladores de RSSF

Devido às características das RSSF e por serem sistemas massivamente distribuídos, os experimentos com dispositivos (nós sensores) reais são caros e muito demorados, em alguns casos se tornando impraticáveis (XUE et al., 2007). Essa dificuldade trouxe a necessidade de se desenvolver simuladores para que protocolos e arquiteturas desenvolvidas em pesquisas pudessem ser previamente testados e avaliados. Essa necessidade existe porque a implantação e construção de arquiteturas em nós sensores reais é difícil e consome muito tempo e recursos. Após essa avaliação preliminar com simuladores, esses protocolos e arquiteturas podem ser implantados e avaliados em RSSF reais, com nós sensores físicos.

Existem vários simuladores de redes de computadores (não apenas

RSSF) disponíveis atualmente. Muitos desses simuladores suportam a simulação de redes sem fio e ad hoc, permitindo que sejam usados para simular RSSF. Esses simuladores são complexos e simulam desde o meio físico de comunicação (cabo, ar, entre outros) até as características de software e hardware dos nós da rede, como baterias, rádios de comunicação, sensores e pilhas de protocolos.

Dentre os diversos simuladores de redes disponíveis atualmente, pode-se destacar alguns importantes. O ns-3 (CONSORTIUM, 2013) é um simulador de redes escrito em C++ e Python muito popular no meio acadêmico. Ele é principalmente utilizado para simulações de redes sem fio, o que contempla o universo das RSSF. O ns-3 é um software gratuito e de código-fonte aberto (*open source*), o que incentiva sua utilização em pesquisas.

Outro simulador de redes de código-fonte aberto é o GNS3 (GROSSMANN; SARAIVA, 2013), que tem como principal característica a simulação de redes complexas, englobando diversos roteadores, *switches* e vários tipos de dispositivos com capacidade de se conectar à uma rede. O GNS3 possui uma interface gráfica para modelagem e configuração da rede que será simulada, o que facilita estes processos para quem não possui conhecimento de programação de computadores.

Um outro simulador gratuito é o Grubix (GRUBI, 2013), que é desenvolvido na linguagem de programação Java e é focado em simulações de redes sem fio ad hoc. Este simulador é implementado seguindo uma arquitetura orientada à eventos (*discrete event network simulation*).

Devido ao fato dos simuladores de redes contemplarem quase todas as características de uma rede real, as simulações realizadas com estes softwares demandam muito tempo para serem executadas. No contexto do

atual estudo, é necessário que as aplicações geradas pelo *framework* proposto sejam simuladas de forma intensiva (dezenas a centenas de vezes por segundo). Os primeiros experimentos realizados utilizaram o simulador Grubix para avaliar os programas gerados pela PG. Para solucionar um cenário simples com poucos nós sensores, o *framework* necessitava de 3 dias de execução. Portanto é impraticável a utilização de um simulador de redes completo para simular e avaliar as aplicações geradas pela PG. Dessa forma, foi desenvolvido um módulo de simulação simplificado para realizar a avaliação das aplicações geradas.

O módulo simplificado desenvolvido foi utilizado para solucionar o mesmo cenário em que o simulador Grubix foi aplicado, o tempo de execução caiu de 3 dias para 2 minutos. Esse módulo de simulação foi desenvolvido baseado na arquitetura interna do simulador Grubix (GRUBI, 2013). Essa foi escolha deve-se ao fato de que o Grubix é um simulador específico para simulações de redes sem fio adhoc, o que torna sua arquitetura mais simples e mais adequada às necessidades do *framework* proposto.

2.3 Middlewares

Middlewares são abstrações na camada de aplicação das RSSF que permitem simplificar a programação de uma rede heterogênea, facilitando a gestão de recursos e energia (WANG et al., 2008). Eles devem prover a coordenação dos nós sensores, através da distribuição de tarefas de baixo nível equivalentes às tarefas de alto nível recebida como entrada, fundir os resultados obtidos pelos nós individualmente e apresentar um *feedback* para o usuário, além de ter mecanismos que promovam a eficiência energética, robustez e escalabilidade (JIN; JIANG, 2010).

Os middlewares (MW) funcionam entre a camada do Sistema Operacional (SO) dos nós sensores e a camada de aplicação e podem ser classificados considerando duas camadas, horizontal e vertical (IBRAHIM, 2009). A camada horizontal define o tipo de MW, ou seja, a sua implementação. Já a camada vertical se preocupa com as propriedades que cada MW possui considerando os ambientes em que estão inseridos.

Existem outras classificações de diversos MWs. O MW proposto se encaixa melhor na categoria de MWs orientados à eventos. O paradigma orientado a eventos é um dos mais recentes tipos de MW existente, embora seja muito promissor há poucos estudos nesta área (JIN; JIANG, 2010). O MW mais notável na categoria de MWs orientados à eventos é o *Impala* (LIU; MARTONOSI, 2003). O *Impala* foi criado visando dar suporte à um projeto maior, denominado ZebraNet, que visa o rastreamento de animais e mapeamento do meio-ambiente. O *Impala* possui algumas características similares ao Odin, como a implantação sem fio de novas aplicações, gerenciamento das características específicas de hardware e do rádio de comunicação, entre outras. A principal diferença é a linguagem de programação em *scripts* provida pelo Odin. Essa linguagem foi desenvolvida e especializada para ser utilizada pela PG, visando aspectos específicos da geração automatizada de código-fonte.

2.4 Algoritmos Genéticos

O Algoritmo Genético (ou *Genetic Algorithm*) é um algoritmo de computação evolutiva inicialmente proposto por Holland (1975). Essa metaheurística baseia-se principalmente no processo de seleção natural. O conceito de seleção natural foi proposto por Charles Darwin em 1859 e consiste

na ideia de que as espécies com características mais adaptadas ao meio onde vivem tendem a sobreviver, enquanto as espécies pouco adaptadas tendem a ser extintas. O mesmo raciocínio ocorre para as características genéticas de uma espécie. Uma característica bem adaptada ao meio, se mantém nas gerações seguintes, já as características pouco adaptadas ao meio, tendem a desaparecer.

Darwin acreditava que o meio ambiente seleciona naturalmente os seres que nele vivem. De forma resumida, os seres que são adaptados ao meio em que vivem sobrevivem. Já os seres que não se adaptam morrem e são extintos, assim como características genéticas de uma espécie. Esse conceito de seleção natural leva em conta o conceito de hereditariedade, que é a capacidade dos seres vivos de transmitirem características genéticas aos seus descendentes. Isso ocorre, por exemplo, durante a meiose das células humanas para geração dos gametas (espermatozóide e óvulo). Durante o processo de meiose pode ocorrer um fenômeno chamado *crossing-over*. Este fenômeno ocorre quando um par cromossomos de DNA trocam trechos de sua codificação genética. Esse fenômeno pode trazer variabilidade nas características provenientes do progenitor. Além disso, podem ocorrer mutações aleatórias na codificação genéticas dos gametas produzidos na meiose.

Todos esses conceitos são levados em conta no Algoritmo Genético. Inicialmente define-se uma população de indivíduos. Cada indivíduo representa uma solução do problema em questão. Essa população tem seus indivíduos inicializados de forma randômica (ou através de alguma rotina específica para o problema tratado). Em seguida, executa-se o chamado laço de evolução, que "evolui" os indivíduos durante um certo número de gerações. Primeiramente é executada uma rotina de seleção, que seleciona dois

indivíduos da população para se realizar o cruzamento (ou recombinação, *crossover*). Um novo individuo é gerado nessa recombinação, em seguida ele é processado por uma rotina que realiza mutações em sua codificação. Esse procedimento de seleção, cruzamento e mutação é executado até que a população da próxima geração seja totalmente gerada.

Para avaliar a aptidão dos indivíduos (qualidade das soluções), o AG utiliza uma função de avaliação (ou função de *fitness*). Essa função de avaliação é uma rotina que analisa a codificação de um indivíduo do AG e calcula um valor que indica a qualidade da solução representada por aquele indivíduo. A função de *fitness* leva em conta características do problema para calcular a aptidão do indivíduo, dessa forma são necessárias diferentes funções de *fitness* para problemas distintos.

Existem diversas variações de AG, Toledo et al. (2009) propõe um AG que possui múltiplas populações que evoluem independentes. Cada população organiza seus indivíduos em uma estrutura hierárquica baseada em árvores. Periodicamente é executada uma rotina de migração, que copia os melhores indivíduos das populações para a população vizinha. Foram realizados experimentos com este tipo de AG, porém os resultados do AG simples se mostraram mais promissores durante o desenvolvimento do estudo.

2.5 Programação Genética

Programação Genética (PG) é um conjunto de técnicas de computação evolutiva que visa realizar a geração de programas de computador (KOZA, 1992). Segundo Baeck, Fogel e Michalewicz (1997), a programação genética utiliza algoritmos evolutivos para evoluir estruturas (indivíduos) que são programas de computador. A função de avaliação (*fit-*

ness) em programação genética consiste em uma medida relacionada à execução desses programas. Segundo os mesmos autores, trata-se de uma busca evolutiva em um espaço possível de programas de computador que sejam capazes de produzir os melhores resultados quando executados. Os programas gerado pela PG podem ser programas independentes ou parte de programas maiores.

A PG é utilizada em diversos campos de aplicação, sendo explorada para gerar programas visando solucionar diversos problemas. Langdon e Gustafson (2010) apresentam uma revisão das pesquisas realizadas nas áreas de PG e máquinas evolutivas em um período de dez anos. São apresentados livros sobre o assunto e mostradas algumas estatísticas acerca das pesquisas em PG e máquinas evolutivas. Já Mckay et al. (2010) realizam uma revisão bibliográfica acerca de aplicações de PG baseada em gramáticas. O artigo se concentra em mostrar como as gramáticas formais contribuem na solução dos problemas, além de realizar uma análise das prováveis tendências da pesquisa nessa área.

2.6 Estado da Arte

Existem na literatura alguns estudos da aplicação de PG na área de redes de sensores. Ohtani e Baba (2005) desenvolvem um sensor de posicionamento ótico utilizando a PG para realizar a separação da luz proveniente de um sinal ótico e da luz proveniente do fundo. O sensor é composto por um conjunto de sensores distribuídos em uma superfície de duas dimensões. Um algoritmo genético (AG) é utilizado para realizar a PG. Pattananupong, Chaiyaratana e Tongpadungrod (2007) propõem um sistema para detectação de deformidades em superfícies táteis utilizando uma pequena rede de

sensores. Uma rede neural combinada com PG é utilizada para determinar o estado de contato dos sensores e detectar as deformidades nas superfícies. Tripathi et al. (2011) apresentam um método híbrido que une a PG à um AG para realizar o posicionamento de nós sensores em uma RSSF. Enquanto a PG otimiza a estrutura de implantação dos nós sensores, o AG realiza o posicionamento de fato dos nós.

Uma aplicação de PG para geração de protocolos distribuídos de redes (não especificamente para RSSF) é apresentada em Weise e Zapf (2009). Os métodos propostos são utilizados para evoluir um algoritmo de eleição. O método denominado "programação genética padrão com memória" utiliza a PG baseada em genomas organizados em árvores. O programa é organizado em vários procedimentos, entre eles um procedimento de inicialização e um procedimento que é chamado quando o nó recebe uma mensagem.

Além da PG, algoritmos evolutivos são largamente utilizados na área de RSSF para diversos fins. Nan e Li (2008) apresentam uma revisão bibliográfica de diversas aplicações de abordagens evolutivas para solucionar problemas da área de RSSF. AGs também são utilizados para diversos objetivos dentro da área de RSSF. Bhondekar et al. (2009) apresentam um método de posicionamento de nós sensores em RSSF baseados em AGs.

Apesar de Programação Genética, Algoritmos Genéticos e algoritmos evolutivos serem estudados na área de RSSF, abordagens que visam desenvolver e configurar automaticamente a rede de sensores como um todo ainda são limitadas. Um método que visa este objetivo é apresentado por Markham e Trigoni (2011). O artigo apresenta um *framework* para geração automática de controladores de redes reguladoras de genes (RRG) para

RSSF. Esses controladores gerados foram utilizados para determinar a atividade e a comunicação dos nós sensores para uma aplicação de rastreamento de alvos. Não ficou claro no artigo se a abordagem é extensível para outros tipos de problemas de RSSF, pois a RRG trabalha baseada em níveis de proteínas e esses níveis de proteínas foram desenvolvidos especificamente para controlar os ciclos de trabalho local (o ciclo de trabalho de cada nó é controlado localmente, sem considerar os outros nós).

Uma abordagem utilizando PG para programar RSSF é apresentada em Weise (2006). O trabalho apresenta um *framework* para gerar algoritmos distribuídos utilizando PG. Este *framework*, de forma similar ao método proposto, não utiliza um simulador de RSSF com a caracterização física da comunicação, mas uma simulação simplificada específica para obter o *fitness* dos programas gerados.

Ainda em Weise (2006), os problemas do máximo divisor comum (MDC) entre dois números e da eleição de nó foram solucionados através de uma RSSF. Diferentemente do método proposto, apenas mensagens em *broadcast* foram utilizadas, portanto problemas que envolvem mensagens direcionadas não foram tratados. Dessa forma os algoritmos distribuídos gerados são restritos à inundar a rede com mensagens, eliminando a complexidade das interações específicas entre os nós sensores, o que é bem similar à uma solução centralizada. O objetivo do presente artigo é mais extenso neste aspecto. No método desenvolvido os nós sensores podem enviar mensagens especificamente para um de seus vizinhos, permitindo a geração de soluções espacialmente distribuídas na rede.

Os estudos limitados motivam a investigação e desenvolvimento de um método aplicando essas ideias visando a programação e a configuração

automática de RSSF para solucionar diversos problemas.

O presente estudo obteve resultados promissores na programação automatizada de RSSF através de técnicas de PG. Alguns destes resultados foram publicados em eventos científicos de ampla divulgação. Oliveira et al. (2013b) apresentam resultados preliminares deste estudo. Resultados de experimentos mais completos podem ser encontrados em Heimfarth et al. (2013) e Oliveira et al. (2013a).

3 METODOLOGIA

A presente seção detalha a metodologia utilizada na execução deste estudo. Serão apresentados todos os detalhes do *framework* desenvolvido, contemplando os três módulos que compõem a ferramenta.

3.1 Visão Geral do Sistema

O *framework* desenvolvido é um sistema que trabalha inserido no contexto de uma RSSF. A Figura 1 apresenta uma ilustração esquemática de um nó sensor que trabalha neste sistema. Uma cópia do *middleware* proposto é instalada e executada em todos os nós sensores da RSSF. Este *middleware* possui uma máquina virtual capaz de interpretar *scripts* escritos em uma determinada linguagem. Esses *scripts* são programas que descrevem em alto nível o ciclo de trabalho dos nós sensores da RSSF. Esse ciclo de trabalho pode variar conforme as características individuais de cada nó sensor, como posicionamento geográfico, variáveis do ambiente e variáveis de controle interno do nó sensor.

O *middleware* deve ser implementado em linguagens de programação específicas para programação de nós sensores. Ele deve prover a máquina virtual que interpreta a linguagem de *script* especificada e deve controlar os aspectos de *hardware* dos nós sensores. Dessa forma, o projetista é abstraído dos aspectos específicos de cada nó sensor e pode se preocupar apenas com os aspectos globais da RSSF, mesmo que a rede seja heterogênea em termos de *hardware* dos nós sensores.

Outro aspecto importante que o *middleware* deve contemplar é a implantação do *script* que controlará a RSSF. O *middleware* precisa ser instalado na RSSF através do acesso físico a cada nó sensor. Porém, é

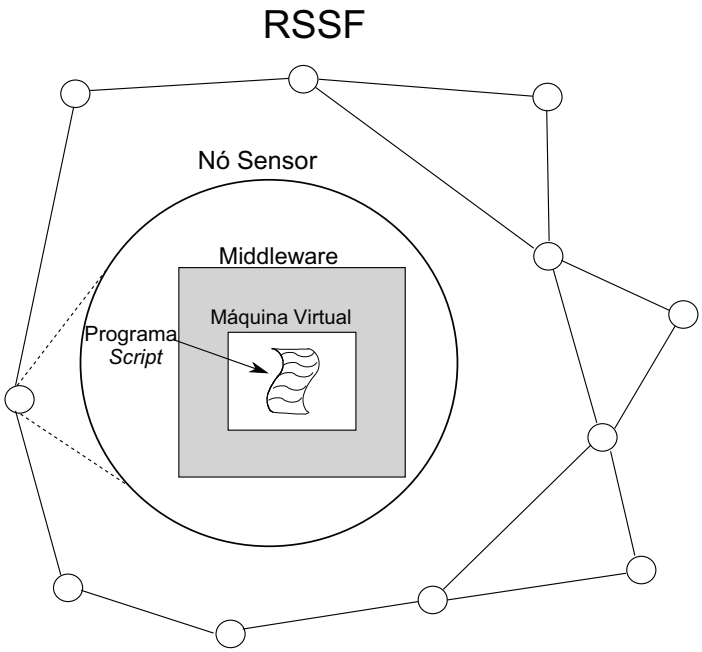

Figura 1 Ilustração esquemática de uma RSSF executando o sistema proposto.

conveniente que o *middleware* possua um recurso para instalação do *script* controlador através da comunicação sem fio da RSSF (*wireless*). Este recurso facilita os testes na RSSF real e economiza tempo na implantação ou atualização do *script* de controle da rede.

Pode-se encontrar na literatura várias abordagens de *middlewares* orientados a agentes móveis para a programação de RSSF, como o *Impala* (LIU; MARTONOSI, 2003). Estes *middlewares* fornecem, além de outros recursos, a capacidade de se programar (ou reprogramar) uma RSSF através da comunicação sem fio, sem precisar ter acesso físico aos nós sensores toda a vez que o programa de controle precisa ser atualizado. Dessa forma, o presente estudo tem o foco principal no desenvolvido do *framework* para gerar *scripts* de controle de forma automatizada.

A Figura 2 apresenta uma visão geral do *framework* desenvolvido. O fluxo de funcionamento da ferramenta é dividido em três partes. A primeira parte é definida pelo projetista da RSSF (Figura 2(a)). Esta etapa deve ser a menor em termos de quantidade de tarefas e custo de tempo, uma vez que o objetivo principal do *framework* é minimizar o trabalho do projetista da rede. Assim, o projetista deve executar uma tarefa apenas: definir a função objetivo (*fitness*). Esta função deve contemplar todos os aspectos que se deseja otimizar durante a geração do *script* de controle, como número de mensagens enviadas durante a execução, tempo de vida da bateria dos nós sensores, entre outros aspectos. A definição desta função objetivo não é uma tarefa trivial, mas uma vez definida, o *framework* será capaz de gerar *scripts* otimizados para qualquer tamanho e topologia conexa de RSSF.

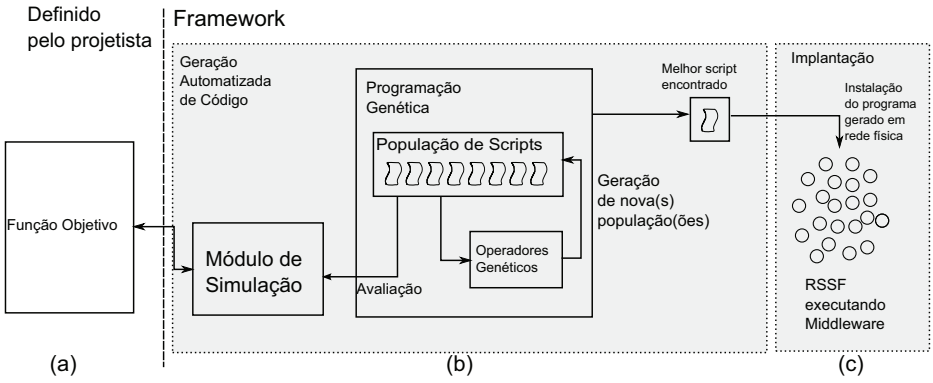

Figura 2 Visão geral do funcionamento do *framework* desenvolvido.

A segunda parte é definida e executada pelo *framework* desenvolvido (Figura 2(b)). A ferramenta realizará a geração de programas através de um método de Programação Genética (PG). Na PG existe uma população de indivíduos, onde cada indivíduo representa um *script* de controle que o *middleware* é capaz de interpretar e executar. Essa população de *scripts* é

evoluída através de operadores genéticos de mutação e recombinação. Cada novo indivíduo gerado pela PG precisa ser avaliado para saber a qualidade do programa de controle. Esse processo de avaliação é realizado através da simulação do funcionamento deste *script* em uma RSSF. Essa simulação é realizada pelo módulo de simulação desenvolvido neste estudo. A simulação fornece estatísticas da execução do *script* para a função objetivo. Assim, a função objetivo fornece um valor que representa a qualidade deste *script* em atender aquele objetivo. Este valor é utilizado como a aptidão (*fitness*) do indivíduo que representa esse *script*.

É importante observar que a avaliação dos *scripts* gerados não pode ser feita executando estes programas em uma RSSF real, uma vez que isso demandaria muito tempo de execução e a avaliação é uma tarefa que é executada intensivamente durante o processo de geração do programa. Por isso, a necessidade do desenvolvimento de um módulo de simulação especializado que seja rápido, porém que contemple os aspectos funcionais e não funcionais da RSSF em questão.

A PG gera novas populações de indivíduos baseada no *fitness* da população atual, buscando evoluir os programas para soluções melhores, ou seja, que atendam melhor o objetivo definido pelo projetista. Quando um determinado critério de parada é atendido (esse critério é configurável), o *script* representado pelo indivíduo com o melhor valor de *fitness* é fornecido para o projetista. Esse *script* é o programa controlador da RSSF que realiza a tarefa descrita pela função objetivo.

Em seguida, na terceira parte, já com a solução final em mãos, o projetista deve implantar (*deployment*) esse *script* na RSSF em que o *middleware* está instalado (Figura 2(c)). Essa implantação é feita através da

comunicação sem fio e consiste basicamente em copiar o *script* fornecido para todos os nós sensores da RSSF. Após o processo de implantação a RSSF já estará executando a tarefa definida pelo projetista através da função objetivo.

As seções a seguir detalharão cada aspecto do processo de geração automatizada de aplicações para RSSF. Neste estudo o *framework* foi aplicado para o problema de detecção de eventos utilizando RSSF. Dessa forma uma função objetivo para avaliar uma solução para este problema também será apresentada.

3.2 Problema de Detecção de Eventos (PDE)

O problema de detecção de eventos (PDE) visa comunicar ao nó concentrador de informações (nó *sink*) a ocorrência de um determinado evento em alguma região da RSSF. Este nó concentrador de informação normalmente é uma central de monitoramento de uma região. Essa central recebe informações da RSSF e precisa ser notificada caso o evento sob monitoramento seja detectado.

Neste tipo de problema as RSSF têm algumas características comuns, por exemplo:

- A rede fica "parada" (sem enviar informações entre os nós) realizando medições periódicas no ambiente.
- Quando um determinado comportamento do ambiente é detectado, um evento ocorre.
- A detecção do evento gera uma mensagem que deve ser enviada até o nó *sink* comunicando a ocorrência do evento.

Este é o comportamento padrão de uma aplicação que visa solucionar um problema de detecção de eventos através de RSSF. Existem vários problemas reais que seguem esse padrão de comportamento de problemas de detecção de eventos. A detecção de incêndios é um problema clássico desse tipo. Uma rede de sensores é espalhada por uma região, esses sensores fazem medições constantes de temperatura e pressão de seus arredores. Quando a temperatura e a pressão chegam em valores determinados, é disparado um evento que indica que provavelmente está ocorrendo um incêndio naquela área. Essa informação deve ser encaminhada na RSSF até o nó *sink*, que pode ser um computador e/ou uma central de monitoramento na região.

Neste estudo o problema de detecção de eventos será tratado de forma genérica. Quando um evento é detectado, o *middleware* registra essa

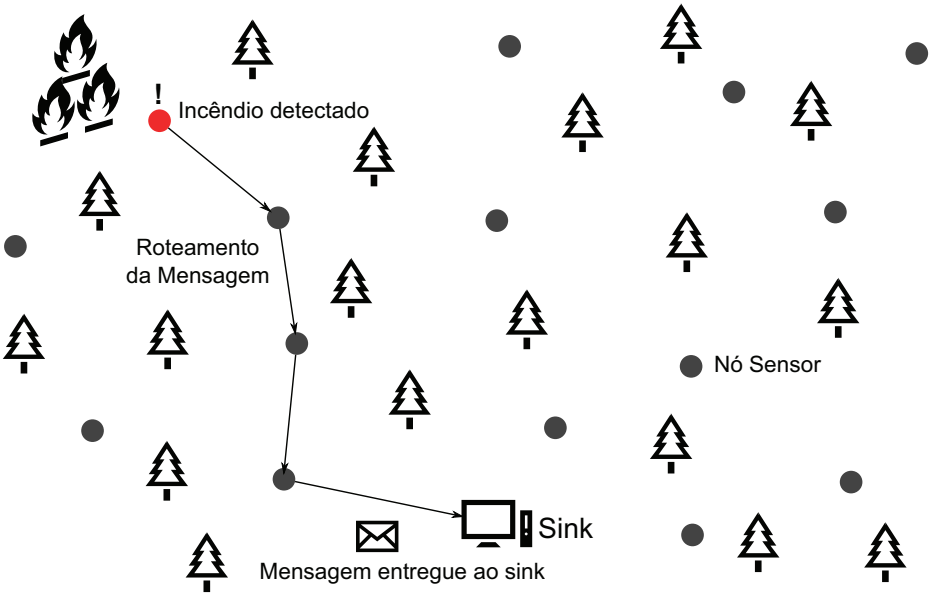

Figura 3 Exemplo de RSSF em uma aplicação do PDE (detecção de incêndios).

informação (um alarme) na memória dos nós sensores. Essa memória pode ser lida pelo *script* de controle que está sendo executado na RSSF, permitindo que ações sejam realizadas mediante à detecção de um evento. Dessa forma, a solução obtida pelo *framework* é aplicável a qualquer tipo de PDE, até mesmo para uma rede que precise monitorar mais de um evento simultaneamente. A Figura 3 apresenta um exemplo de RSSF tratando um evento detectado.

3.3 Função Objetivo para o PDE

A função objetivo (ou função de avaliação) calcula a qualidade de uma determinada solução (*script*) gerada pela PG. Essa qualidade representa a eficiência com que o *script* de controle soluciona um problema determinado. Essa eficiência é representada por um valor real positivo. A função objetivo é descrita como uma minimização, portanto valores menores representam uma qualidade maior do programa avaliado. Neste estudo foi desenvolvida uma função objetivo que representa o PDE. Essa função objetivo foi utilizada para avaliar a capacidade do *framework* de gerar aplicações (*scripts* de controle) capazes de solucionar com eficiência satisfatória este problema.

Uma das principais características do *framework* desenvolvido é que o projetista da RSSF necessita definir apenas uma função objetivo que descreva bem o objetivo final da rede. Essa função objetivo será utilizada pela PG como a função de avaliação (função de *fitness*) do algoritmo evolutivo. Devido à isso, a chave para se obter boas soluções através deste *framework* é uma função objetivo bem definida.

A função objetivo baseia-se em dados (estatísticas) obtidas através

da simulação de um *script* gerado pela PG. Além destes dados, alguns parâmetros devem ser definidos para balancear o peso de cada característica da rede. Estes dados e parâmetros são formalizados na Tabela 1 e na Tabela 2:

Tabela 1 Parâmetros da função objetivo.

C_{el}	Custo de eventos que não forem entregues ao *sink*.
C_{ms}	Custo do envio de mensagens.
C_{pm}	Custo do envio prematuro de mensagens.
C_{pa}	Custo de ações prematuras.
C_{mwd}	Custo de mensagens enviadas para direção incorreta.
C_{ed}	Custo da distância mínima entre a mensagem e o *sink*.

Tabela 2 Dados de simulação utilizados pela função objetivo.

PS	Tamanho do *script* gerado.
EL	Número de eventos que não foram comunicados ao *sink*.
PM	Número de mensagens enviadas antes que algum evento ocorra.
PA	Número de ações executadas antes que um evento seja detectado.
MWD	Número de mensagens enviadas para a direção contrária do *sink*.
ED	Distância mínima entre a mensagem comunicando o evento do nó *sink*.
MS	Número total de mensagens enviadas durante a simulação.

Considerando estes parâmetros e dados de simulação, a função objetivo é definida a seguir na Equação 1.

$$\begin{aligned} Min\ F(\cdots) = \quad & C_{ms} \cdot MS + C_{el} \cdot EL + C_{pa} \cdot PA + C_{pm} \cdot PM \\ + \quad & PS + C_{ed} \cdot (ED)^2 + C_{mwd} \cdot MWD \cdot EL \end{aligned} \tag{1}$$

Esta função objetivo define basicamente 2 requisitos. Primeira-

mente, entregar a mensagem da ocorrência de um evento ao nó *sink*, que é o requisito que diz se o PDE foi solucionado ou não por um programa. O segundo requisito é realizar essa entrega com um número mínimo de mensagens enviadas pela RSSF. O rádio de comunicação sem fio é um dos recursos que mais gastam energia em um nó sensor. Como a energia é um recurso limitado na maioria das RSSF, minimizar o número de mensagens enviadas significa prolongar o tempo de vida da RSSF. Outro aspecto é que, em algumas vezes, menor número de mensagens enviadas significa um menor intervalo entre a detecção do evento e a entrega da mensagem ao nó *sink*.

Dados estes requisitos, o problema é definido sem nenhuma restrição formal. Dessa forma, todas as soluções obtidas são factíveis em termos de modelagem do problema (algumas podem ser infactíveis no mundo real). Em detrimento de restrições formais, as violações ao comportamento desejado são penalizadas na função objetivo. Este tipo de tratamento das violações é melhor para PG, uma vez que o algoritmo evolutivo não precisa tratar e gerenciar indivíduos infactíveis. Outra vantagem é que o espaço de possíveis soluções se torna mais homogêneo sem soluções infactíveis. Uma desvantagem porém é que este espaço de busca se torna maior devido à falta de restrições formais.

Quando um evento é detectado por um nó sensor, este deve encaminhar uma mensagem para o nó *sink* comunicando a ocorrência deste evento. Quando a mensagem de evento detectado não consegue chegar até o nó *sink* a contagem EL é incrementada. Esta contagem é penalizada na função objetivo pois é uma violação direta aos requisitos do problema tratado ($C_{el} \cdot EL$).

Outra penalidade que é considerada é quando um nó sensor realiza alguma ação antes que o evento que está sendo monitorado tenha ocorrido.

Por exemplo, o nó sensor pode começar a escrever na memória baseado em uma *trigger* (uma condição) inconsistente. Estas ações consomem energia do nó sensor, o que é uma das características que se deseja otimizar durante o funcionamento de uma RSSF. Essas ações realizadas prematuramente normalmente são inúteis e não contribuem para solução do problema. Dessa forma, cada mensagem enviada antes da ocorrência do evento é penalizada em $C_{pm} \cdot PM$. Já as ações (que não sejam envio de mensagens) realizadas prematuramente são penalizadas em $C_{pa} \cdot PA$.

Um comportamento desejável da rede é que a cada mensagem enviada por um nó sensor, esta mensagem fique mais próxima do nó *sink*. Por isso cada vez que uma mensagem é enviada na direção oposta à direção do *sink* uma penalização é adicionada ao valor de *fitness*. Esta penalização é adicionada em $C_{mwd} \cdot MWD \cdot EL$. A penalização é multiplicada pela contagem de eventos não entregas ao *sink*, pois em alguns casos, é necessário que algumas mensagens sejam enviadas na direção oposta ao *sink* para que ela possa ser entregue à ele (redes esparsas). Neste caso, se a RSSF é capaz de entregar todas as mensagens de eventos ao nó *sink*, esta penalização é ignorada.

Uma penalização muito importante incluída na função objetivo é aquele que penaliza a distância mínima entre o último nó sensor avisado da ocorrência do evento e o nó *sink*. Esta componente da função objetivo trata do requisito principal do PDE, a entrega da mensagem comunicando a ocorrência do evento ao nó *sink*. Esta penalização é tratada em $C_{ed} \cdot (ED)^2$. Quando o evento é comunicado com sucesso ao nó *sink*, essa distância mínima (ED) é zero e portanto não há penalizações no *fitness*. Uma característica importante obtida através desta penalização é a diferenciação de um

script que consegue levar a mensagem para mais próximo do *sink* do que um outro *script*. Essa diferenciação ajuda o algoritmo evolutivo da PG à guiar melhor a evolução da população de *scripts*.

Por último, são adicionadas duas penalizações que não contemplam aspectos funcionais da RSSF, mas sim eficiência na solução do PDE. A primeira é o tamanho total do programa (*script*) gerado (PS). A segunda penalização é aplicada sobre o número total de mensagens enviadas pela RSSF durante toda a simulação. Essa componente da função objetivo visa minimizar a quantidade de mensagens enviadas, economizando energia das baterias dos nós sensores. Essa penalização é adicionada em $C_{ms} \cdot MS$.

3.4 Módulo de Simulação de RSSF

Simular uma RSSF por completo é uma tarefa complexa e que contempla diversos aspectos físicos do ambiente, além de características de hardware e software dos nós sensores (pilhas de protocolos, sinal de comunicação sem fio, entre outros). Devido à necessidade de realizar simulações de forma muito intensiva, foi implementado um módulo de simulação de RSSF que não contempla a maior parte dos aspectos de uma rede real e simplifica vários outros aspectos.

Além da simulação da RSSF, foi necessário implementar uma versão simulada do *middleware* proposto. Essa implementação do *middleware* provê para a simulação todas as funcionalidades que o *middleware* em um nó sensor real deve prover.

A Figura 4 apresenta um conjunto de informações que o simulador necessita para realizar simulações de programas gerados pela PG. O simulador foi desenvolvido de forma que ele seja capaz de realizar simulações do

PDE com RSSF da forma mais eficiente possível. Esse conjunto de parâmetros que o simulador necessita compõe uma instância de testes no contexto deste estudo. De maneira geral, a função do simulador de RSSF é avaliar a qualidade de um programa gerado pela PG quando executado no contexto de uma instância de teste.

simulationMaxTime
Duração da simulação que será realizada (em unidades virtuais de tempo).

nodesSleepCycle
Intervalo de tempo entre as execuções do programa em cada nó sensor.

numberOfNodes
Número de nós sensores na rede.

sinkNode
ID do nó *sink*.

eventTriggers
Lista com o ID de cada nó sensor que detectará um evento durante a simulação. Contém também o instante virtual em que cada evento ocorrerá.

sensorNodes
Lista com a configuração de cada nó sensor. Os nós possuem uma posição geográfica (coordenadas), alcance do rádio de comunicação (em metros) e o tamanho de sua memória (ativa e passiva, veja a seção 3.5).

Figura 4 Conjunto de parâmetros do simulador de RSSF que compõe uma instância de testes.

O simulador implementado é executado através de passos (laços de repetição), onde cada iteração deste laço de execução representa uma unidade de tempo virtual. A duração da simulação é definida pelo parâmetro *simulationMaxTime*. Em cada iteração deste laço de repetição será verifi-

cado se algum evento contido na lista *eventTriggers* necessita ser disparado. Caso o tempo de ocorrência de algum evento tenha sido alcançado, um comando é disparado para a implementação simulada do *middleware* do nó sensor específico. O *middleware* então se encarrega de tratar a detecção deste evento. O funcionamento do *middleware* será melhor detalhado na seção 3.5 a seguir.

```
   Data: script
 1 begin
 2     middleware ← Implementação simulada.;
 3     for node ∈ sensorNodes do
 4         node.instalaMiddleware(middleware);
 5         node.middleware.implantaScript(script);
 6         node.middleware.inicializa();
 7     for currentTime ← 1 até simulationMaxTime do
 8         for event ∈ eventTriggers do
 9             if event.ocorreu(currentTime) then
10                 sensorNodes.disparaEvento(event);
11                 eventTriggers.removeEvento(event);
12         for node ∈ sensorNodes do
13             node.middleware.executaScript();
```

Figura 5 Pseudocódigo da execução do simulador.

Após a verificação e a ativação dos eventos detectados, o simulador dispara um comando de execução do *script* de controle para o *middleware* em cada nó sensor da rede. O *middleware* verifica se a memória do nó sensor foi alterada desde o último comando de execução do *script* de controle. Caso tenha havido alterações na memória, o *script* de controle é executado pela máquina virtual. Caso contrário a execução do *script* não é realizada nesta iteração da simulação.

A Figura 5 apresenta um pseudocódigo da execução básica do simulador para um programa gerado pela PG. Durante a execução da simulação dados estatísticos da simulação são armazenados e repassados para a função objetivo, que por sua vez realiza o cálculo de *fitness* do *script* simulado. Os dados coletados durante a simulação são formalizados na seção 3.3, na Tabela 2.

A implementação do simulador e da versão simulada do *middleware* é bem mais complexa do que a parte retratada na Figura 5. Entretanto, esses detalhes não serão especificados pois muitos deles são detalhes de implementação.

3.5 Arquitetura Geral do *Middleware*

O *middleware* atua como um sistema intermediário entre a estrutura do nó sensor sem fio e o código *script* gerado pela PG. Ele atua como um pequeno sistema operacional que gerencia os recursos do nó sensor, fornece acesso a serviços e funcionalidades pré-programadas. Além disso ele controla como é interpretada a representação do indivíduo gerada pela PG. O *middleware* proposto consiste em uma interface pré-definida que determina quais funcionalidades ele deve prover. A implementação real pode ser feita diretamente para o microcontrolador do nó sensor ou como uma aplicação para um sistema operacional específico para nós sensores (como o TinyOS).

A Figura 6 apresenta uma ilustração esquemática da arquitetura geral do *middleware*. Nela são representados diversos componentes que compõem o *middleware* e algumas das interações entre eles. Também estão representados alguns componentes do nó sensor que o *middleware* pode fa-

zer uso.

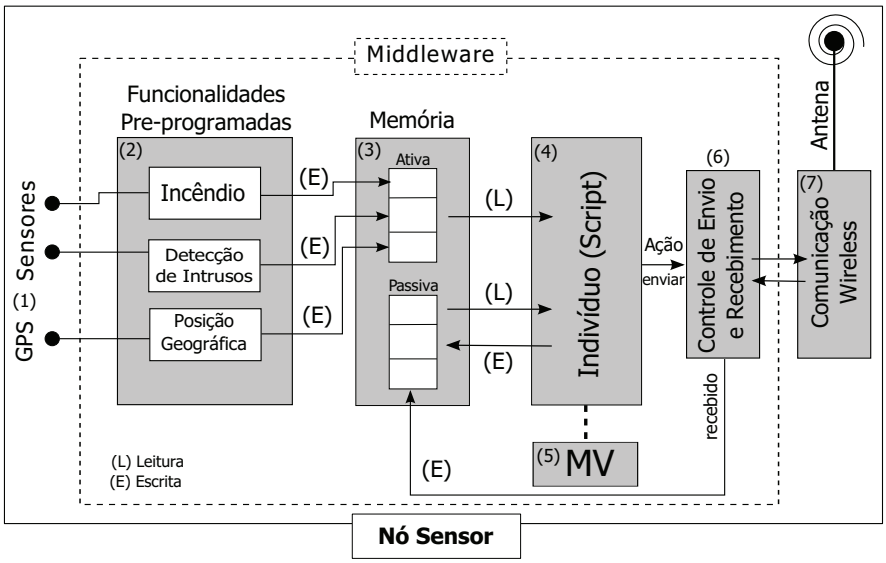

Figura 6 Arquitetura geral do *middleware* desenvolvido.

O nó sensor possui diversos sensores, como sensores de temperatura, pressão, umidade e até mesmo podem estar equipados com um sistema de GPS (Figura, 6(1)). Estes sensores e sistemas de sensoriamento são acessados pelo *middleware* com o intuito de prover funcionalidades pré programadas para o *script* de controle (Figura, 6(2)). Essas funcionalidades podem ser a detecção de incêndios baseada nas leituras de sensores de temperatura e pressão, a detecção de intrusos baseada na leitura de um sensor de movimento ou então o posicionamento geográfico obtido da leitura de um GPS.

Apesar destas funcionalidades serem utilizadas pelo *script* de controle, este não tem acesso direto à estes blocos de código pré programados. A interação do *script* de controle com essas funcionalidades é através da

memória ativa do *middleware*, que também possui uma memória passiva (Figura, 6(3)). As funcionalidades pré programadas escrevem dados na memória ativa que podem ser lidos posteriormente pelo *script* de controle. Esses dados podem ser a posição geográfica do nó *sink*, a posição do intruso detectado, um valor booleano dizendo se um incêndio está ocorrendo ou não, entre outros. O *script* tem acesso "somente leitura" à memória ativa e não pode alterar os valores nela armazenados. A memória passiva do *middleware* tem um objetivo diferente e é uma forma que o *script* de controle tem para armazenar informações internas de controle e enviar e receber mensagens dos nó vizinhos. Na memória passiva o *script* pode ler e escrever dados sem restrições.

O *script* de controle fica armazenado dentro do *middleware* (Figura, 6(4)) e é interpretado pela máquina virtual (MV) que também faz parte do *middleware* (Figura, 6(5)). Quando um comando de envio de mensagem para um nó vizinho é executado, esse comando é repassado para o controle de envio e recebimento de mensagens do *middleware* (Figura, 6(6)). O controle envia a mensagem para o nó sensor de destino através do rádio de comunicação sem fio do nó sensor (Figura, 6(7)). Quando este recebe a mensagem, os dados recebidos na mensagem são escritos na memória passiva do nó de destino. Esses dados podem ser lidos e utilizados pelo *script* de controle posteriormente.

O envio e recebimento de mensagens realizados pelo *script* de controle é basicamente uma escrita de dados na memória passiva de um nó sensor vizinho. Dessa forma, a comunicação entre dois nós sensores da rede é transparente para o *script* de controle, simplificando o código de controle.

O objetivo final é que todos os nós sensores da RSSF possuam o *mid-*

dleware instalado executando o mesmo *script* de controle, porém o estado (valores armazenados) de suas memórias ativa e passiva modifiquem o comportamento de cada nó. Neste estudo não foi realizada uma implementação do *middleware* para um nó sensor real, foi realizada apenas uma versão simulada em software para que experimentos computacionais pudessem ser realizados.

3.6 Linguagem *Script* do *Middleware*

O *middleware* proposto possui uma máquina virtual (MV) que é capaz de interpretar *scripts* descritos em uma linguagem específica. Esta linguagem é composta basicamente de uma estrutura condicional (chamada de *trigger*) e comandos imperativos pré programados na implementação da MV. Neste estudo a linguagem foi desenvolvida de forma que os *scripts* de controle possuam um expressividade suficiente para solucionar o PDE. A Tabela 3 apresenta as estruturas existentes na linguagem proposta.

Tabela 3 Estruturas existentes na linguagem proposta.

Estrutura	Exemplo	Descrição
trigger	if(*ev*1 *op ev*2)	Estrutura condicional composta de dois operandos e um operador.
send command	send(*ev*,*dest*)	Envia um dado para um nó vizinho.
up event command	up(*ev*)	Guarda o valor *true* em um evento.
down event command	down(*ev*)	Guarda o valor *false* em um evento.

As *triggers* são "gatilhos" que são utilizados como estruturas para condicionar a execução de um conjunto de comandos. Como apresentado na

Tabela 3, uma *trigger* é composta por dois eventos (*ev*1 e *ev*2) e um operador (*op*). Os eventos, no contexto do *script* de controle, são posições na memória do *middleware* (memória ativa ou passiva, veja a seção 3.5). Essas posições de memória são chamadas de eventos pois só guardam valores booleanos (verdadeiro ou falso). Dessa forma, um evento está inativo quando está com o valor falso e está ativo quando está com o valor verdadeiro. Na prática, ativar um evento significa mudar o valor em sua posição de memória para verdadeiro. O operador de uma *trigger* pode ser o operador lógico "E" (*and*) ou o ooperador lógico "OU" (*or*).

O comando de envio de mensagens (*sendcommand*) possui dois parâmetros, um evento (*ev*) e um destinatário (*dest*). Ele consiste basicamente em ativar um evento (escrever o valor "verdadeiro" em uma posição da memória) na memória de um nó sensor vizinho. O nó sensor vizinho é definido pelo parâmetro *dest*, que na verdade é uma direção na qual o nó vizinho se encontra. Essa direção pode ser norte, sul, leste ou oeste (*up*, *down*, *right* e $left$). O *middleware* processa essa direção e envia a mensagem para o nó sensor vizinho mais próximo naquela direção. Ao receber a mensagem o *middleware* do nó vizinho ativa o evento especificado na mensagem.

Os outros dois comandos existentes na linguagem são o comando de *upevent* e *downevent*. Esses comandos ativam e desativam, respectivamente, um determinado evento (*ev*) na memória passiva do *middleware*.

A Figura 7 apresenta um exemplo completo de *script* de controle que pode ser executado pela MV do *middleware* proposto. Os *scripts* são executados na ordem em que os comandos aparecem, por isso a ordem em que as *triggers* são organizadas influenciam no fluxo de execução dos comandos. Uma observação importante é que o *script* de controle, após executado, só

é executado novamente se alguma alteração na memória houver ocorrido na última execução. Caso nenhuma alteração tenha sido feita, o *script* só executará novamente quando algum evento ativo seja alterado ou caso o nó sensor receba uma mensagem de algum vizinho ativando um evento passivo.

```
1  if A1 and A2 then
2      up(P2);
3      down(P3);
4      send(P3, up);
5      down(P1);
6  if P1 or A2 then
7      donw(P1);
8      send(P2, right);
9  if P3 and P2 then
10     up(P2);
11     down(P1);
12     send(P1, down);
```

Figura 7 Exemplo de *script* de controle que pode ser interpretado pela MV do *middleware* proposto.

No exemplo da Figura 7, a *trigger* na linha 1 verifica se os eventos da memória ativa na posição 1 (A1) e na posição 2 (A2) estão ativos. Caso essa condição seja satisfeita, os comandos na linha 2 a 5 serão executados sequencialmente. Primeiramente o evento passivo na posição 2 (P2) será ativado e o evento passivo na posição 3 (P3) será desativado. Em seguida o evento passivo na posição 3 do nó vizinho ao norte (*up*) será ativado através de uma mensagem. Por fim o evento passivo na posição 1 (P1) é desativado. Na sequência da execução, a MV avalia as outras *triggers* e executa os comandos daquelas cuja a sua condição seja satisfeita.

O *framework* desenvolvido tem o objetivo de obter um *script* des-

crito nesta linguagem para que possa ser executado em uma RSSF com o *middleware* instalado. Apesar da linguagem ter sido desenvolvida com o intuito de facilitar a geração automática de código, o projetista da RSSF pode escolher escrever um *script* manualmente, ou até mesmo alterar o *script* encontrado pelo *framework*.

3.7 Programação Genética

O *framework* desenvolvido possui um módulo responsável pela realização da Programação Genética (PG), que realiza a geração automatizada de código fonte apoiada nos outros módulos que compõem a ferramenta. Este módulo foi implementado de uma forma que ele seja facilmente estendido. Um usuário do *framework* pode criar de maneira simples novos algoritmos evolutivos para realizar o processo de PG. Os operadores genéticos utilizados pelos algoritmos evolutivos também podem ser facilmente customizados e estendidos.

Nesta seção o módulo de PG será descrito de forma detalhada contemplando o algoritmo evolutivo utilizado nos experimentos deste estudo.

3.7.1 Representação do Indivíduo

Um indivíduo na PG representa um programa, neste caso um *script* de controle que pode ser executado pelo *middleware* proposto. A representação desses *scripts* nos indivíduos da PG foi feita utilizando-se diretamente a estrutura da versão simulada do *middleware*.

Cada indivíduo carrega um programa que é representado por uma lista de *triggers*. Cada *trigger* possui um cabeçalho e uma lista de comandos. O cabeçalho de uma *trigger* representa a condição para que a lista de

comandos seja executada. A Figura 8 apresenta uma ilustração da representação do individuo que foi utilizada. Esta é a representação do *script* apresentado na Figura 7.

A1 **and** A2	P1 **or** A2	P3 **and** P2
up(P2)	down(P1)	up(P2)
down(P3)	send(P2,→)	down(P1)
send(P3,↑)		send(P1,↓)
down(P1)		

Figura 8 Exemplo de representação (codificação genética) do indivíduo da PG.

A representação não possui uma estrutura diferente do *script* na linguagem desenvolvida, por isso ela pode ser lida diretamente pela versão simulada do *middleware*, sem a necessidade de realizar processos de codificação/decodificação.

3.7.2 Operadores Genéticos

Nessa seção, uma descrição dos operadores genéticos responsáveis pela geração de novos indivíduos durante a evolução da PG.

3.7.2.1 Recombinação

Os operadores de recombinação propostos visam permitir uma troca efetiva de informações entre os indivíduos pais na geração de um indivíduo filho. São considerados os diversos *triggers* e comandos que compõem a representação de cada indivíduo. Foram criados um total de quatro operadores de recombinação, dois deles baseados em recombinação de um ponto e outros dois baseados em recombinação uniforme.

O primeiro operador de recombinação é a recombinação de um ponto em *trigger* e é ilustrado na Figura 9. Um novo indivíduo é criado a partir de dois indivíduos já existentes utilizando uma recombinação de um ponto no nível das *triggers* do programa. A linha tracejada indica o ponto de corte. O novo indivíduo herda todos os *triggers* do primeiro pai até o ponto de corte e todos os *triggers* do segundo pai após o ponto de corte. O ponto de corte é sorteado aleatoriamente baseado no pai que possui menos *triggers*.

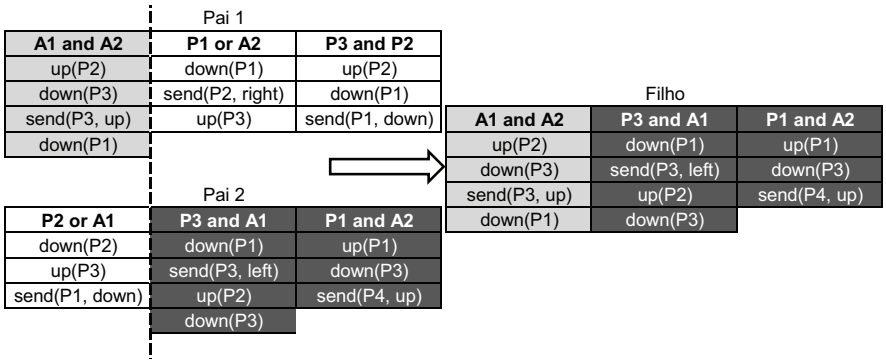

Figura 9 Operador de recombinação de um ponto em *trigger*.

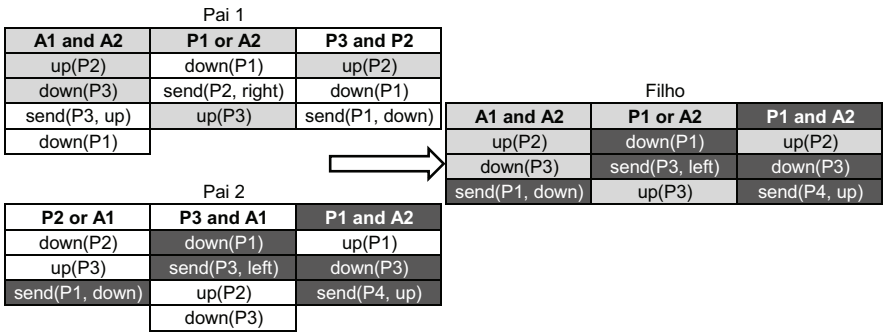

Figura 10 Operador de recombinação de um ponto em comando.

A recombinação de um ponto em comando ocorre dentro de cada

trigger conforme exemplificado na Figura 10. Inicialmente é selecionado um ponto de corte para cada lista de comandos em todos os *triggers*. Um novo indivíduo é criado pela lista de comandos até o ponto de corte do primeiro pai mais a lista de comandos após o ponto de corte do segundo pai. Listas de tamanho diferentes terão um ponto de corte que ocorra em ambas. A escolha de qual cabeçalho será utilizado para compor a *trigger* do novo indivíduo é feita aleatoriamente entre os dois pais, a probabilidade é de 50% para cada pai.

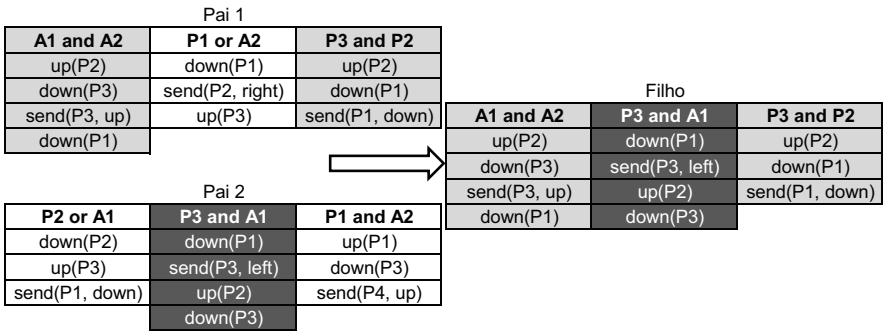

Figura 11 Operador de recombinação uniforme em *trigger*.

A Figura 11 ilustra o terceiro operador de recombinação, a recombinação uniforme por *trigger*. Nesta recombinação cada *trigger* dos pais é escolhida aleatoriamente entre os dois. Existe um probabilidade de 50% de escolher uma *trigger* de um dos pais. Caso um pai possua mais *triggers* do que o outro, cada *trigger* excedente terá uma probablidade de 50% de ser incluída ou não no indivíduo filho.

Por último, a Figura 12 apresenta um exemplo da recombinação uniforme por comando. Este operador funciona de forma semelhante à recombinação de um ponto por comando. Para cada lista de comandos dos pais, um cabeçalho é sorteado com 50% de probabilidade para cada pai.

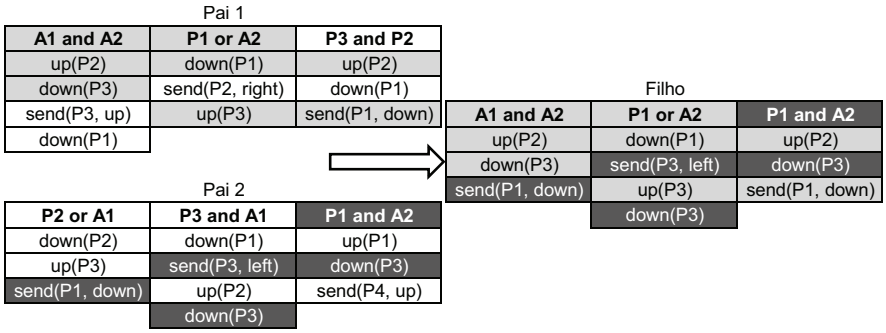

Figura 12 Operador de recombinação uniforme em comando.

Em seguida, para cada comando existente nos pais, uma probabilidade de 50% é dada a cada um deles e apenas o comendo de um deles é transferido para o indivíduo filho. Assim como na recombinação uniforme por *trigger*, se um pai possui mais comandos do que o outro, cada comando excedente tem 50% de probabilidade de ser incluído no indivíduo filho.

3.7.2.2 Mutação

Além de operadores de recombinação, operadores de mutação são utilizados para a exploração de novas áreas do espaço de soluções. Operadores genéticos de mutação são aplicados aos indivíduos gerados pela recombinação com o intuito de criar novos "genes" nos indivíduos. Os operadores de recombinação exploram o rearranjo dos genes já existentes dentro da população de indivíduos. Para se criar novos genes são necessários os operadores de mutação. Foram criados um total de sete operadores de mutação para realizar esta tarefa. A Tabela 4 apresenta e explica estes sete operadores de mutação.

3.7.3 Algoritmo Evolutivo

O algoritmo evolutivo utilizado para realizar a PG é ilustrado por meio de um pseudocódigo na Figura 13. Este algoritmo evolutivo é baseado em um AG clássico com utilização de elitismo.

Primeiramente, todos os indivíduos de uma população são inicializados (linha 2). Cada indivíduo é preenchido com uma lista de *triggers* gerada aleatoriamente. Em seguida todos estes indivíduos são simulados e avaliados (linha 3), sendo atribuído um valor de *fitness* à cada um deles. A simulação de cada indivíduo é feita utilizando o módulo de simulação

Tabela 4 Operadores genéticos de mutação utilizados pela PG.

Operador:	Descrição:
Substituir comando:	Sorteia um comando de forma aleatória e troca este por outro comando gerado também aleatoriamente.
Reiniciar comandos:	Faz o mesmo que "Substituir comando", porém ele substitui todos os comandos de um *trigger* escolhido aleatoriamente.
Remover e inserir:	Sorteia um comando em um *trigger* aleatório, remove o comando deste *trigger* e o insere em uma posição aleatória em outro *trigger*.
Trocar comandos:	Sorteia dois comandos dentro de um *trigger* aleatório e troca os dois de posição na lista de comandos deste *trigger*.
Alterar comando:	Sorteia um comando e modifica os parâmetros do mesmo, como eventos e destinos.
Trocar *triggers*:	Troca a ordem de dois *triggers* sorteados aleatoriamente na lista de *triggers* de um indivíduo.
Alterar cabeçalho:	Modifica os parâmetros do cabeçalho de um *trigger* sorteado aleatoriamente, alterando os operadores e operandos.

```
1  begin
2      inicializar(populacao);
3      simularEAvaliar(populacao);
4      geracaoAtual ← 1;
5      while geracaoAtual ≤ numeroDeGeracoes do
6          populacaoIntermediaria ← ∅;
7          inserir(melhorIndividuo(populacao),
           populacaoIntermediaria);
8          for i ← 2 to numeroDeIndividuos do
9              (pai1, pai2) ← torneio(populacao);
10             if probabilidade(taxaDeRecombinacao) then
11                 filho ← recombinacao(pai1, pai2);
12             else
13                 filho ← copiar(melhor(pai1,pai2));
14             if probabilidade(taxaDeMutacao) then
15                 filho ← mutacao(filho);
16             inserir(filho, populacaoIntermediaria);
17         simularEAvaliar(populacaoIntermediaria);
18         populacao ← populacaoIntermediaria;
19         geracaoAtual ← geracaoAtual + 1;
20     return melhorIndividuo(populacao)
```

Figura 13 Pseudocódigo do algoritmo evolutivo utilizado na PG.

desenvolvido (seção 3.4) e o valor de *fitness* é calculado através da função objetivo proposta (seção 3.3).

O contador de número de gerações é inicializado como a primeira geração da população (linha 4). Em seguida, o laço de evolução da população é iniciado (linha 5). A evolução da população será repetida até que o contador de número de gerações atinja o paramêtro que define o número máximo de gerações (*numeroDeGeracoes*).

Na linha 6, a população intermediária é definida como um conjunto

vazio de indivíduos. Em seguida o conceito de elitismo é aplicado inserindo o melhor indivíduo da população atual diretamente na população intermediária (linha 7). Na linha 8 um laço de repetição é iniciado para realizar a gerações dos outros indivíduos da população intermediária. Ao final deste laço de repetição a população intermediária terá o mesmo número de indivíduos que a população atual.

Os indivíduos que serão utilizados como pais na recombinação são selecionados através de um torneio (linha 9). O torneio escolhe o melhor indivíduo dentro alguns selecionados aleatoriamente da população atual. O número de indivíduos selecionados aleatoriamente para o torneio é definido por um parâmetro da PG (tamanho do torneio). São realizados dois torneios e os dois pais para a recombinação são selecionados ($pai1$ e $pai2$).

Em seguida um teste de probabilidade é aplicado ao parâmetro que define a taxa de recombinação. Caso o teste seja positivo, o indivíduo filho é gerado através de um operador de recombinação (selecionado aleatoriamente dentre os 4 desenvolvidos) aplicado aos pais selecionados no torneio. Caso contrário o melhor indivíduo selecionado nos torneios (melhor($pai1$,$pai2$)) é copiado para o indivíduo filho (linhas 10 a 13).

Após a criação do indivíduo filho, um novo teste de probabilidade é aplicado ao parâmetro que define a taxa de mutação. Caso este teste seja positivo um operador de mutação (selecionado aleatoriamente dentre os 7 criados) será aplicado ao indivíduo filho (linhas 14 e 15). Na linha 16 este indivíduo filho é inserido na população intermediária.

Em seguida, após todos os indivíduos da população intermediária terem sido criados, toda a população intermediária é simulada e avaliada, atribuindo-se um valor de *fitness* à todos os novos indivíduos (linha 17).

Dessa forma, toda a população intermediária é copiada para a população principal (linha 18). É importante observar que todos os indivíduos anteriormente contidos na população principal são descartados neste ponto. Por último o contador do número de gerações é incrementado na linha 19.

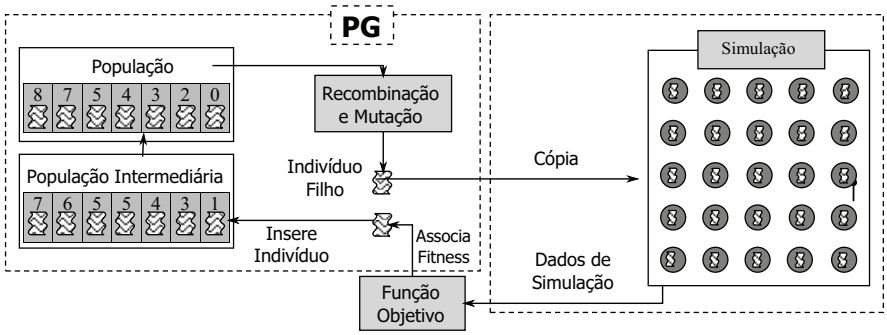

Figura 14 Visão geral da execução da PG e sua interação com o *framework* porposto.

Ao final da execução da PG o melhor indivíduo da população principal é retornado como a melhor solução encontrada (linha 20). A Figura 14 apresenta uma visão geral do fluxo de execução da PG e sua interações com os outros módulos do *framework*.

4 RESULTADOS E DISCUSSÃO

Com o intuito de avaliar o *framework* proposto, alguns experimentos computacionais foram realizados para avaliar a capacidade da ferramenta em gerar aplicações para RSSF de forma automatizada. Para isso foi implementado um gerador de instâncias de testes e posteriormente um *benchmark* de instâncias de teste para realizar os experimentos. Nesta seção serão apresentados o gerador de instâncias, o *benchmark* criado e os resultados obtidos com os experimentos computacionais.

4.1 Gerador de Instâncias

Um gerador de instâncias foi criado para gerar diferentes RSSF que representem o PDE. Este gerador de instâncias recebe uma configuração de parâmetros que diz quais características as instâncias que serão geradas devem possuir. A Tabela 5 apresenta estes parâmetros e suas respectivas descrições.

4.2 Instâncias de Teste Criadas (*Benchmark*)

Para avaliar o desempenho da abordagem proposta, oito instâncias de teste foram criadas utilizando o gerador de instâncias desenvolvido. Esse *benchmark* foi criado pois não existe na literatura um conjunto de instâncias similares para que o experimentos fossem realizados. Cada instância de teste representa um RSSF para solucionar o PDE, onde cada nó sensor da rede possui uma posição fixa. Além disso, uma instância de teste possui outras informações como qual nó da RSSF é o nó *sink*, quais nós da rede detectarão um evento durante uma simulação, o alcance de transmissão sem fio dos nós

Tabela 5 Parâmetros do gerador de instâncias.

Parâmetro:	Descrição:
fieldSizeX:	Largura da área onde a RSSF estará (tamanho da dimensão X).
fieldSizeY:	Altura da área onde a RSSF estará (tamanho da dimensão Y).
numberOfNodes:	Número total de nós sensores na RSSF.
activeMemorySize:	Tamanho da memória ativa dos nós sensores da rede.
passiveMemorySize:	Tamanho da memória passiva dos nós sensores da rede.
transmissionRange:	Alcance (em metros) da transmissão sem fio dos nós sensores.
numberOfEventTriggers:	Número total de eventos que serão detectados durante a simulação.
simulationMaxTime:	Tempo total (virtual) que a simulação terá.
nodePlacementType:	Tipo de posicionamento dos nós sensores. Este parâmetro define a topologia da RSSF que será gerada. Existem três tipos, o primeiro posiciona os nós em grade, o segundo posiciona os nós aleatoriamente no campo, o terceiro posiciona os nós aleatoriamente também, porém eles só podem ocupar uma posição no campo que seja conexa aos outros nós já posicionados na rede.

sensores e o número total de nós na rede. A Figura 4 na seção 3.4 apresenta e descreve essas informações. Dentre estas oito instâncias geradas, foram contempladas instâncias com dois tipos de topologia, em grade (*grid*) e topologia randômica (aleatória).

A Tabela 6 apresenta as principais características de cada uma das oito instâncias que compõem o *benchmark* de testes. O 'R' no nome das instâncias indica que esta instância possui topologia randômica. Já o 'G'

indica que a instância possui topologia em grade. O número contido no nome das instâncias indica o número total de nós sensores da RSSF representada por esta instância. Essa tabela apresenta também as dimensões (em metros) da área monitorada pela RSSF.

Tabela 6 Principais características das instâncias de teste do *benchmark*.

Instância	Num. de Nós	Dimensões da Área	Topologia
G25	25	40m x 40m	Em Grade
G49	49	60m x 60m	Em Grade
G225	225	140m x 140m	Em Grade
G625	625	240m x 240m	Em Grade
R25	25	40m x 40m	Randômica
R49	49	60m x 60m	Randômica
R225	225	140m x 140m	Randômica
R625	625	240m x 240m	Randômica

A Figura 15 apresenta uma ilustração da instância R25. Na figura os círculos representam os nós sensores da RSSF e as linhas representam a conectividade de comunicação sem fio dos nós. Os círculos maiores (rotulados como "Evento") representam os nós sensores que irão detectar algum evento durante a simulação. Já o círculo central (rotulado como "Nó Sink") representa o nó *sink* que deve ser informado sobre a detecção de eventos na rede. A instância R25 é a mais simples dentre as instâncias com topologia randômica, dessa forma, fica mais fácil de entender o que uma instância de testes representa.

As instâncias com topologia em grade são mais fáceis de se solucionar. Isso ocorre pois a solução ótima (*script* de controle ótimo) é simples e idêntica para qualquer tamanho de instância (número de nós na rede). Com isso a PG consegue obter a solução ótima para essas instâncias muito mais

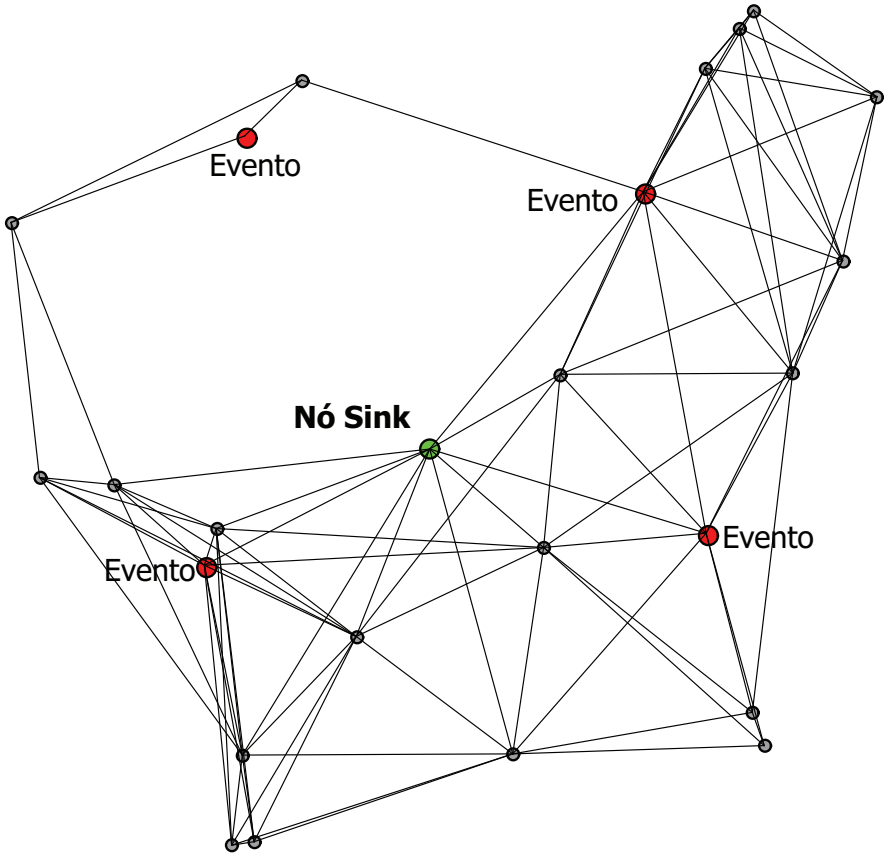

Figura 15 Ilustração da topologia de rede da instância R25.

rápido do que para as instâncias de topologia randômica.

Em vários casos a PG precisa desenvolver comportamentos específicos para algumas regiões da RSSF quando está solucionando uma instância de topologia randômica. Isso ocorre pois essas instâncias podem conter regiões mais difíceis no contexto do roteamento das mensagens até o nó *sink*. Por exemplo, na Figura 15 a região noroeste da rede precisa ser tratada de forma específica pelo *script* de controle, uma vez que os nós nesta região não têm um caminho direto até o nó *sink*. Esse problema já não ocorre quando

trata-se de instâncias com topologia em grade, como pode-se observar na Figura 16 que apresenta uma ilustração da instância G49.

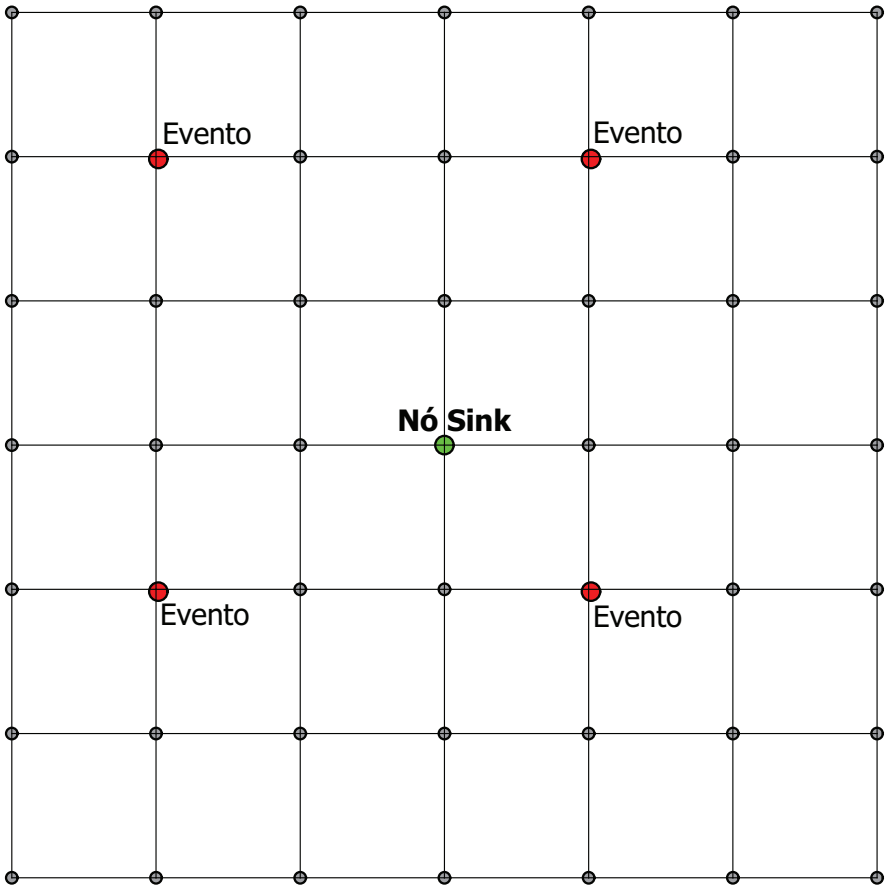

Figura 16 Ilustração da topologia de rede da instância G49.

4.3 Experimentos Computacionais e Resultados Obtidos

4.3.1 Configurações dos Experimentos

Para avaliar o *framework* desenvolvido, este foi executado para solucionar as oito instâncias de testes descritas na seção 4.2. O método desen-

volvido foi executado 10 (dez) vezes em cada uma das instâncias de teste e os valores médios obtidos são considerados nas análises que seguem (também são analisados valores máximos e mínimos). A utilização de um valor médio de várias execuções nas análises é essencial, uma vez que a PG é um método estocástico e pode fornecer diferentes soluções quando executada mais de uma vez com os mesmos parâmetros. Além disso, a análise de valores médios serve também como uma análise de estabilidade do método proposto.

Todos os experimentos aqui reportados foram realizados em computadores idênticos. Esses computadores possuem cada um deles, um processador Core 2 Duo 2,66GHz, 2GB de memória RAM e executam o sistema operacional Gentoo Linux 64 bits. O *framework* foi implementado utilizando a linguagem de programação C++ e compilado através do GNU Cross Compiler (GCC).

A PG foi configurada com os parâmetros descritos na Tabela 7. Todos estes parâmetros foram ajustados através de testes empíricos. A alta taxa de mutação utilizada justifica-se pelo fato de que novos comandos e novas *triggers* só são criados por operadores de mutação. Operadores de recombinação só conseguem reorganizar os genes já existentes dentre os indivíduos da população. Veja a seção 3.7.2 para mais detalhes sobre os operadores genéticos utilizados.

Os custos utilizados na função objetivo (veja a seção 3.3 para mais detalhes) foram ajustados tentando traduzir da forma mais fiel possível a importância de cada componente da função objetivo. A Tabela 8 apresenta os custos utilizados nos experimentos. Alguns testes empíricos foram realizados para ajudar no ajuste desses custos.

Tabela 7 Configuração de parâmetros utilizada na PG.

Parâmetro:	Valor Utilizado:
Número de Indivíduos	20
Taxa de Recombinação	0,65
Taxa de Mutação	0,85
Tamanho do Torneio	3
Elistismo	Sim
Número de Gerações	2.000

Tabela 8 Configuração dos custos utilizados na função objetivo da PG.

Custo:	Valor Utilizado:
C_{ms}	2
C_{pa}	2.500
C_{el}	5.000
C_{pm}	500
C_{mwd}	500
C_{ed}	500

4.3.2 Resultados dos Experimentos

Os resultados obtidos pelo *framework* após ser executado dez vezes para cada uma das oito instâncias de testes são apresentados na Tabela 9. São exibidos os valores de *fitness* do melhor indivíduo encontrado após as 2.000 gerações da PG. Para cada instância, são apresentados os valores médios, mínimos e máximos de *fitness*. O coeficiente de variação do *fitness* nas dez execuções também é reportado. Por último é apresentado o tempo médio de execução das 2.000 gerações para cada instância. O tempo de execução é um valor médio das dez execuções. O valor de *fitness* considerado é o *fitness* do melhor indivíduo da população da PG após cada execução de cada instância.

Tabela 9 Resultados obtidos nos experimentos computacionais.

	Fitness da Melhor Solução			Coeficiente	Tempo
Instância	Média	Mínimo	Máximo	de Varia.	Total(s)
G25	22,6	20	27	13,55%	55,1
G49	29	29	29	0,00%	116,6
G225	61	61	61	0,00%	654,7
G625	101	101	101	0,00%	2.135
R25	32,9	30	53	22,22%	88,7
R49	84,8	43	189	70,75%	208,4
R225	1.157	376	2.722	77,74%	1.927,3
R625	2.756.2	461	4.384	47,27%	4.447,4

Inicialmente, os resultados mostram claramente a diferença de dificuldade entre as instâncias com topologia de rede em grade e as instâncias com topologia de rede randômica. Os resultados nas instâncias com topologia em grade mostram uma grande estabilidade do método, além de uma alta qualidade das soluções encontradas. As melhores soluções encontradas para estas instâncias foram analisadas e foi possível observar que elas possuem o comportamento ótimo. Isto é, em todas as instâncias de topologia de rede em grade o *framework* foi capaz de encontrar a solução ótima para elas. A solução ótima para estas instâncias pode ser facilmente escrita manualmente por um projetista, dessa forma, comparando o *fitness* da solução ótima escrita manualmente com o *fitness* da solução obtida pelo *framework*, é fácil verificar se o comportamento das soluções encontradas pela PG é o comportamento ótimo.

A Figura 17 apresenta o *script* de comportamento ótimo encontrado para a instância G49. O evento ativo A0 é a posição da memória ativa que representa o evento detectado. Já os eventos ativos A1 até A4 são eventos escritos pela versão simulada do *middleware* que indicam a direção

geográfica (norte, sul, leste ou oeste) que o nó *sink* se encontra. As direções são A1 para o norte, A2 para leste, A3 para sul e A4 para oeste. No caso das instâncias de topologia em grade, a solução ótima nada mais é que enviar a mensagem do evento para o nó vizinho que está na mesma direção do nó *sink*. Este *script* precisou enviar 12 mensagens durante toda a simulação para solucionar a instância G49 (Figura, 16).

```
1  if P0 and A3 then
2      send(P0, down);
3  if P0 and A1 then
4      send(P0, up);
5  if A2 and P0 then
6      send(P0, right);
7  if A4 and P0 then
8      send(P0, left);
9  if A0 or P0 then
10     up(P0);
```

Figura 17 *Script* de comportamento ótimo econtrado pelo *framework* na instância G49.

Já nas instâncias com topologia de rede randômica, a qualidade e a estabilidade do método caem consideravelmente. Essas instância necessitaram de aproximadamente o dobro do tempo de execução do que as instâncias com topologia de rede em grade. Isso ocorre pois a complexidade das redes com topologia randômica é bem maior, por isso a simulação de cada indivíduo para o cálculo de *fitness* é mais demorada, já que as soluções necessitam de mais mensagens enviadas para comunicar o nó *sink* da ocorrência do evento. Observou-se que 2.000 gerações da PG não foram suficientes para garantir uma alta qualidade de soluções para redes com

topologia complexa.

Apesar da falta de estabilidade em topologias complexas, todas as execuções para todas as oito instâncias de testes foram capazes de encontrar soluções que conseguem entregar todos os eventos ao nó *sink*. Isto é, em todas as execuções dos experimentos realizados, o *framework* foi capaz de solucionar o PDE. A variação de *fitness* das soluções obtidas foi devido à variação da quantidade necessária de mensagens para comunicar o nó *sink* da ocorrência dos eventos.

```
1 if P0 and A1 then
2     send(P0, up);
3 if P0 and A2 then
4     send(P0, down);
5 if A4 and P0 then
6     send(P0, down);
7 if P0 or A0 then
8     up(P0);
```

Figura 18 *Script* com o melhor comportamento econtrado pelo *framework* na instância R25.

A Figura 18 apresenta o *script* com o melhor comportamento encontrado para a instância R25. Os eventos passivos e ativos têm o mesmo significado que no *script* da instância G49 (Figura, 17). A principal diferença que se nota em relação ao *script* da instância G49 é que os comandos de envio de mensagens não são todos para o vizinho na direção do nó *sink*. Por exemplo, seguindo o *script*, para os nós sensores cujo o nó *sink* esteja à leste, as mensagens são encaminhadas para o vizinho mais próximo ao sul. Essas regras de roteamento são desenvolvidas para tratar aspectos

específicos da instância R25.

Os resultados obtidos nos experimentos mostraram que o *framework* desenvolvido é capaz de solucionar o PDE. Mesmo em instâncias com uma rede extensa (mais de 600 nós sensores) e topologias complexa o método proposto foi capaz de solucionar o PDE com uma qualidade satisfatória. Isso mostra que a PG em união com a arquitetura desenvolvida neste estudo é promissora no contexto da geração de aplicações para RSSF de forma automatizada.

5 CONCLUSÃO E TRABALHOS FUTUROS

O presente estudo propõe o desenvolvimento de um *framework* que seja capaz de gerar aplicações para RSSF de forma automatizada. Esta ferramenta foi desenvolvida em três módulos que se complementam, um simulador de RSSF, um *middleware* para RSSF que provê uma linguagem de *scripts* para descrever aplicações, e um módulo de Programação Genétic (PG).

O *middleware* proposto gerencia os recursos de hardware dos nós sensores da RSSF e fornece funcionalidades pré programadas para os usuários. Ele também possui um máquina virtual capaz de interpretar *scripts* de controle escritos em uma linguagem pré definida. O módulo de simulação é utilizado para realizar a avaliação dos programas gerados de forma automatizada pelo módulo de PG. A PG utiliza um algoritmo evolutivo para gerar programas descritos nessa linguagem de *scripts*.

A junção destes três módulos forma um *framework* para que projetistas de RSSF possam gerar aplicações para diferentes problemas de forma automatizada. Basta o projetista definir uma função objetivo que descreva o problema que ele quer tratar com a RSSF e o *framework* se encarrega de gerar um *script* de controle que solucione este problema com uma boa performance.

Para avaliar a qualidade e a capacidade do *framework* proposto em atingir seu objetivo, o Problema de Detecção de Eventos (PDE) foi utilizado como caso de teste. Foi criado um gerador de instâncias para o PDE e um *benchmark* com oito instâncias de testes. Experimentos computacionais foram realizados utilizando-se este *benchmark*.

Os resultados obtidos nos experimentos mostram que a ferramenta

desenvolvida é capaz de solucionar de maneira ótima o PDE para RSSF com topologia em grade, independente do número de nós sensores considerados. Em instâncias com topologia randômica, a ferramenta foi capaz de solucionar o PDE com uma qualidade de soluções satisfatória, mesmo contemplando redes extensas (RSSF com mais de 600 nós sensores). Apesar disso, o método mostrou-se um pouco instável para instâncias cuja a rede possui uma topologia complexa. A qualidade das soluções encontradas em diversas execuções para as mesmas instâncias variou em alguns casos.

Contudo, o *framework* foi capaz de solucionar o PDE em todas as oito instâncias tratadas nos experimentos computacionais. Isso mostra que o este trabalho conseguiu atingir seus objetivos iniciais e que o método desenvolvido é promissor no contexto de geração automática de aplicações para RSSF. Essa geração automatizada de aplicações para RSSF pode reduzir bastante os custos de implementação, uma vez que visa eliminar a maior parte do trabalho humano na programação dos nós sensores da rede.

Apesar do sucesso em solucionar o PDE, o *framework* ainda possui vários pontos de atenção que não foram tratados no presente estudo. Como trabalho futuro, a ferramenta pode ser testada em instâncias mais complexas do PDE, por exemplo instâncias que contemplem simultaneamente mais de um tipo de evento. Além disso, experimentos com nós sensores reais são necessários, uma vez que são importantes para constatar definitivamente a qualidade da abordagem proposta.

Em um estudo mais avançado, o *framework* pode ser melhorado para que a linguagem *script* fornecida pelo *middleware* tenha uma maior expressividade. Dessa forma, a ferramenta poderia ser utilizada para solucionar outros problemas de RSSF que sejam mais complexos.

REFERÊNCIAS

AKYILDIZ, I. et al. Wireless sensor networks: a survey. **Computer Networks**, Amsterdam, v. 38, n. 4, p. 393-422, 2002. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1389128601003024>. Acesso em: 10 dez. 2013.

BAECK, T.; FOGEL, D.; MICHALEWICZ, Z. **Handbook of evolutionary computation**. New York: Taylor and Francis, 1997. 1130 p. Disponível em: <http://books.google.com.br/books?id=n5nuiIZvmpAC>. Acesso em: 10 dez. 2013.

BHONDEKAR, A. P. et al. Genetic algorithm based node placement methodology for wireless sensor networks. In: INTERNATIONAL MULTICCONFERENCE OF ENGINEERS AND COMPUTER SCIENTISTS, 2009, Hong Kong. **Proceedings...** Hong Kong: IEEE, 2009. 1 CD-ROM.

CONSORTIUM, N. **The network simulator (ns-3)**. Disponível em: <http://www.nsnam.org/>. Acesso em: 20 dez. 2013.

GROSSMANN, J.; SARAIVA, F. J. **Graphical Network Simulator (GNS3)**. Disponível em: <http://www.gns3.net/>. Acesso em: 20 dez. 2013.

GRUBI. **GRUBIX Simulator**. Disponível em: <http://asteroide.dcc.ufla.br/∼ grubi/grubix/>. Acesso em: 20 dez. 2012.

GUO, W.; ZHANG, W. A survey on intelligent routing protocols in wireless sensor networks. **Journal of Network and Computer Applications**, London, v. 38, p. 185-201, Feb. 2014.

HANSMANN, U. et al. **Pervasive computing**. New York: Springer, 2003. 448 p.

HEIMFARTH, T. et al. Automatic generation and configuration of wireless sensor networks applications with genetic programming. In: SOFTWARE TECHNOLOGIES FOR FUTURE EMBEDDED AND UBIQUITOUS SYSTEMS (SEUS), WORKSHOP ON, 2013, Paderborn. **Proceedings...** Paderborn: IEEE, 2013. p. 9-16.

HOLLAND, J. H. **Adaptation in natural and artificial systems**. Ann Arbor: University of Michigan, 1975. 183 p.

IBRAHIM, N. Orthogonal classification of middleware technologies. In: MOBILE UBIQUITOUS COMPUTING, SYSTEMS, SERVICES AND TECHNOLOGIES, 3., 2009, Sliema. **Proceedings...** Sliema: IEEE, 2009. p. 46-51.

JIN, H.; JIANG, W. **Handbook of research on developments and trends in wireless sensor networks:** from principle to practice. Hershey: Information Science Reference, 2010. 556 p.

KARL, H.; WILLIG, A. **Protocols and achitectures for wireless sensor networks**. Chichester: J. Wiley, 2005. 526 p.

KOZA, J. R. **Genetic programming:** on the programming of computers by means of natural selection. Cambridge: MIT, 1992. 835 p.

LANGDON, W. B.; GUSTAFSON, S. M. Genetic programming and evolvable machines: ten years of reviews. **Genetic Programming and Evolvable Machines**, London, v. 11, p. 321-338, Mar. 2010.

LIU, T.; MARTONOSI, M. Impala: a middleware system for managing autonomic, parallel sensor systems. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 9., 2003, San Diego. **Proceedings...** San Diego: ACM, 2003. p. 1-12.

MARKHAM, A.; TRIGONI, N. The automatic evolution of distributed controllers to configure sensor networks operation. **The Computer Journal**, London, v. 54, n. 3, p. 421-438, 2011.

MCKAY, R. I. et al. Grammar-based genetic programming: a survey. **Genetic Programming and Evolvable Machines.** London, v. 11, p. 365-396, Mar. 2010.

NAN, G.; LI, M. Evolutionary based approaches in wireless sensor networks: a survey. In: INTERNATIONAL CONFERENCE ON NATURAL COMPUTATION, 4., 2008, Jinan. **Proceedings...** Jinan: IEEE, 2008. p. 217-222. Disponível em: <http://dl.acm.org/citation.cfm?id=1473247.1473821>. Acesso em: 10 dez. 2013.

OHTANI, K.; BABA, M. A smart optical position sensor with genetic programming technique. In: INSTRUMENTATION AND MEASUREMENT TECHNOLOGY CONFERENCE, 2005, Ottawa. **Proceedings...** Ottawa: IEEE, 2005. v. 2, p. 1166-1171.

OLIVEIRA, R. R. R. de et al. A genetic programming based approach to automatically generate wireless sensor networks applications. In: EVOLUTIONARY COMPUTATION (CEC), 2013, Cancún. **Proceedings...** Cancún: IEEE, 2013a. p. 1771-1778.

OLIVEIRA, R. R. R. de et al. Programação automática de redes de sensores sem fio utilizando programação genética. In: WORKSHOP DE SISTEMAS DISTRIBUÍDOS AUTONÔMICOS, 3., 2013, Brasília. **Anais...** Brasília: SBC, 2013b. p. 45-48.

PATTANANUPONG, U.; CHAIYARATANA, N.; TONGPADUNGROD, R. Genetic programming and neural networks as interpreters for a distributive tactile sensing system. In: EVOLUTIONARY COMPUTATION (CEC), 2007, Singapore. **Proceedings...** Singapore: IEEE, 2007. p. 4027-4034.

TOLEDO, C. F. M. et al. Multipopulation genetic algorithm to solve the synchronized and integrated two-level lot-sizing and scheduling problem. **International Journal of Production Research**, London, v. 47, n. 11, p. 3097-3119, 2009.

TRIPATHI, A. et al. Wireless sensor placement using hybrid genetic programming and genetic algorithm. **International Journal on Intelligent Information Technologies**, Hershey, v. 7, n. 2, p. 63-83, 2011.

WANG, M. M. et al. Middleware for wireless sensor networks: a survey. **Journal of Computer Science and Technology**, London, v. 23, n. 3, p. 305-326, May 2008.

WEISE, T. **Genetic programming for sensor networks.** Kassel: University of Kassel, 2006. 16 p.

WEISE, T.; ZAPF, M. Evolving distributed algorithms with genetic programming: election. In: ACM/SIGEVO SUMMIT ON GENETIC AND EVOLUTIONARY COMPUTATION, 5., 2009, New York. **Proceedings...** New York: ACM, 2009. p. 577-584.

XUE, Y. et al. Performance evaluation of ns-2 simulator for wireless sensor networks. In: CANADIAN CONFERENCE ON ELECTRICAL AND COMPUTER ENGINEERING (CCECE), 2007, Vancouver. **Proceedings...** Vancouver: IEEE, 2007. p. 1372-1375.

YANG, H. C. The application of the wireless sensor network (wsn) in the monitoring of fushun reach river in china. In: COMPUTER AND NETWORK TECHNOLOGY, 2., 2010, Bangkok. **Proceedings...** Bangkok: IEEE, 2010. p. 331-333.