INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO CEARÁ
PRÓ-REITORIA DE ENSINO
COORDENADORIA DE CIÊNCIA DA COMPUTAÇÃO DO CAMPUS ARACATI
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DANIEL VICTOR SARAIVA

ALGORITMOS EVOLUCIONÁRIOS MULTIOBJETIVOS PARA A SELEÇÃO DE CASOS DE TESTE PARA SISTEMAS INTELIGENTES

ARACATI - CE
2017

DANIEL VICTOR SARAIVA

ALGORITMOS EVOLUCIONÁRIOS MULTIOBJETIVOS PARA A SELEÇÃO DE CASOS DE TESTE PARA SISTEMAS INTELIGENTES

Trabalho de conclusão de curso apresentado à Coordenadoria de Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia do Ceará - Campus Aracati como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Área de concentração: Inteligência Artificial.

Orientadora: Prof[a]. Msc. Francisca Raquel de Vasconcelos Silveira

ARACATI - CE
2017

Dados Internacionais de Catalogação na Publicação (CIP)

S243a Saraiva, Daniel Victor.

Algoritmos evolucionários multiobjetivos para a seleção de casos de teste para sistemas inteligentes./ Daniel Victor Saraiva. – Aracati: IFCE, 2017. 73f.:

Orientador: Profª. Msc. Francisca Raquel de Vasconcelos Silveira.
Monografia (Graduação em Ciência da computação) – IFCE.

1. Inteligência artificial. 2. Algoritmos evolucionários. 3. Agente racional. I. Título.

IFCE/BIBLIOTECA/ARACATI CDD: 006.3

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO CEARÁ
COORDENADORIA DE CIÊNCIA DA COMPUTAÇÃO DO CAMPUS ARACATI
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DANIEL VICTOR SARAIVA

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do Grau de Bacharel em Ciência da Computação, sendo aprovado pela Coordenadoria de Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia do Ceará - Campus Aracati e pela banca examinadora:

Fca. Raquel de V. Silveira

Profª. Msc. Francisca Raquel de Vasconcelos Silveira
Instituto Federal do Ceará - IFCE
Orientadora

Diego Rocha Lima

Prof. Msc. Diego Rocha Lima
Instituto Federal do Ceará - IFCE

Adriano Patrick do Nascimento Cunha

Prof. Esp. Adriano Patrick do Nascimento Cunha
Universidade de Fortaleza - UNIFOR

Aracati, 02 de Maio de 2017

A meus pais, familiares e colegas pelo incentivo e apoio constante.

AGRADECIMENTOS

Tendo em vista que esta é uma conquista que não é construída sozinha, agradeço aos meus pais, Francisco Freire Saraiva e Dagna Maria Victor Saraiva por me proporcionarem o melhor que suas condições permitiam e por me ensinarem os valores da humildade, paciência, honestidade e amor. Agradeço também aos demais familiares, a minha esposa, irmãos e amigos que acreditam em mim e torcem pelo meu sucesso.

Sou muito grato à minha estimada professora e orientadora Francisca Raquel de V. Silveira por todo o conhecimento compartilhado durante todos esses anos e pelo grande exemplo de pessoa e de intelectual que és.

Agradeço aos membros da banca pela disponibilidade de participar deste momento ímpar da minha formação acadêmica e pelas valiosas contribuições que enriquecerão o nosso trabalho.

Muito obrigado a todos os professores que contribuíram grandemente para a minha formação acadêmica e para o meu crescimento enquanto ser humano. Aos meus colegas de curso, obrigado por todos os momentos partilhados e parcerias formadas.

Ao Instituto Federal de Educação de Educação, Ciência e Tecnologia do Ceará por me proporcionar o acesso ao conhecimento que muitas vezes é negado aos filhos da classe trabalhadora mais pobre.

Por fim, agradeço a todos que, de algum modo, contribuíram para a concretização deste momento.

"Meus filhos terão computadores sim, mas antes terão livros. Sem livros, sem leitura, os nossos filhos serão incapazes de escrever - inclusive a sua própria história."

(Bill Gates)

RESUMO

Com o aumento da utilização de sistemas computacionais, agente racional surge como uma tecnologia promissora na solução desde problemas relativamente simples, bem como, problemas complexos que fazem parte de sistemas ainda maiores. Devido sua autonomia, testes de agentes tem se tornado um grande desafio, pois esses agentes podem apresentar diferentes resultados em uma mesma entrada de testes. Além disso, nem sempre os melhores casos estão disponíveis nos primeiros testes, de maneira que, dependendo do ambiente de tarefa do agente, pode existir uma grande quantidade de casos a serem analisados. Dessa forma, um caso de teste considerado ótimo, é aquele em que o agente obtem o valor mínimo possível de desempenho, revelando as falhas durante o ciclo de testes. Devido aos algoritmos evolucionários serem capazes de solucionar problemas com múltiplos objetivos, este trabalho apresenta a proposta de um agente testador que realiza busca local no espaço de estados de casos de teste orientado por utilidade e utiliza os algoritmos evolucionários multiobjetivos, NSGAII, SPEA2, PAES e MOCell para selecionar casos de testes eficientes para o teste de agentes racionais. Com os resultados da aplicação dos testes, são realizadas comparações entre os algoritmos a fim de identificar quais deles são mais eficientes na geração de casos de testes que sejam capazes de gerar informações relevantes para o projetista identificar aqueles episódios problemáticos que estão causando o desempenho insatisfatório dos agentes e assim realizar melhorias em seus programas.

Palavras-chave: Inteligência Artificial , Agentes Racionais, Algoritmos Evolucionários.

ABSTRACT

With the increase use of computer systems, rational agent emerges as a promising technology in solution from relatively simple problems, as well as complex problems that are part of even larger systems. Due to its autonomy, agent testing has become a major challenge because these agents can present different results in the same test entry. In addition, the best cases are not always available in the first tests, so that, depending on the agent's task environment, there may be a large number of cases to be analyzed. Thus, a test case considered optimal is one in which the agent obtains the minimum possible value of performance, revealing the failures during the test cycle. Due to evolutionary algorithms being able to solve problems with objective multiples, this work presents the proposal of a tester that performs local search in the state space of utility-oriented test cases and uses multiobjective evolutionary algorithms, NSGAII, SPEA2, PAES and MOCell to select efficient test cases for the rational agents test. With the results of the application of the tests, comparisons are made between the algorithms in order to identify which of them are more efficient in the generation of test cases that are able to generate information relevant to the designer to identify those problematic episodes that are causing the unsatisfactory performance of the agents and thus make improvements in their programs.

Keywords: Artifical Intelligence, Rational Agents, Evolutionary Algorithms.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AA	Aprendizagem Artificial
ACO	*Ant Colony Optimization Algorithm*
AIS	*Artificial Immune Systems*
AM	Aprendizagem de Máquina
AG	Algoritmo Genético
CLONALG	*Clonal Selection Algorithm*
EAs	*Evolutionary Algorithms*
IA	Inteligência Artificial
JADE	*Java Agent Development Framework*
JMetal	*Metaheuristic Algorithms in Java*
OR	*Operations Research*
PAES	*Pareto Archived Evolution Strategy*
PEAS	*Perfomance*, *Environment*, *Actuators*, *Sensors*
NSGA-II	*Non-dominated Sorting Genetic Algorithm*
MOCell	*MultiObjective Cellular*
MOEAs	*Multiobjective Evolutionary Algorithms*
MOPs	*Multiobjective Optimization Problems*
SAC	Sistema Adaptativo Complexo
SMA	Sistema Multi-Agente
SOEAs	*Single-objective Evolutionary Algorithms*
SPEA2	*Strength Pareto Evolutionary Algorithm*

SUMÁRIO

1 INTRODUÇÃO

Neste primeiro capítulo serão apresentados os principais elementos e discussões que fundamentam, contextualizam e motivam o desenvolvimento deste trabalho. Apresentaremos também os objetivos a serem alcançados e como a pesquisa está organizada.

1.1 Contextualização

Os softwares estão cada vez mais complexos e não dependem, em última análise, do aumento do poder computacional. Mesmo que esse poder seja duplicado com o passar do tempo (Lei de Moore[1]), isso não significa que os problemas tenham se tornado mais fáceis de solucionar. Desse modo, o objetivo central da Inteligência Artificial (IA) é procurar incorporar nos programas e sistemas, conhecimentos e capacidades normalmente associadas ao ser humano, a fim de encontrar soluções viáveis (COSTA; SIMõES, 2015).

Para Luger (2013), a IA é um ramo promissor da ciência da computação que se ocupa da automação do comportamento inteligente, cujo interesse principal é encontrar um modo efetivo de entender e aplicar técnicas inteligentes na solução de problemas. Consequentemente, essas técnicas devem ser baseadas em princípios teóricos que incluem as estruturas de dados utilizadas na representação do conhecimento, em algoritmos necessários para aplicar essa representação e nas linguagens e técnicas de programação usadas em sua implementação.

Atualmente, a Inteligência Artificial abrange diferentes subcampos, desde os gerais (aprendizagem e percepção), até tarefas específicas, como jogos, demonstrações de teoremas matemáticos, condução de veículos e diagnósticos de doenças, tornando-se assim, relevante para inúmeras tarefas (RUSSELL; NORVIG, 2013). A despeito disso, Coppin (2013, p. 4) menciona que: "em muitos casos, técnicas de Inteligência Artificial são utilizadas para solucionar problemas relativamente simples ou problemas complexos que fazem parte de sistemas mais complexos".

Costa e Simões (2015) enfatiza que existe uma generalização de que os computadores só fazem aquilo para que foram programados. Porém, eles são utilizados cada vez mais em situações para as quais se torna difícil antecipar o que pode acontecer. Em situações como essas, capacidades como autonomia, flexibilidade, aprendizagem, entre outras, se tornam fundamentais. Assim sendo, Luger (2013) considera que é papel dos pesquisadores da IA a tarefa de criar uma

[1] Lei de Moore surgiu em 1965 através de um conceito estabelecido por Gordon Earl Moore onde o poder de processamento dos computadores dobraria a cada 18 meses.

inteligência genérica, desenvolvendo ferramentas de diagnósticos, prognósticos ou visualização que facilite solucionar essas situações.

Agentes racionais surgem, então, como uma tecnologia promissora. Um agente racional deve possuir atributos capazes de distinguí-lo dos demais programas, tais como: operação sob controle autônomo; percepção do ambiente; persistência por tempo prolongado; adaptação às mudanças e capacidade de criação e perseguição de metas. Sua racionalidade pode ser definida quando o agente atua para alcançar o melhor resultado. Quando há incertezas, deve retornar o melhor resultado esperado conforme sua medida de desempenho (RUSSELL; NORVIG, 2013).

Para Costa e Simões (2015), o termo autonomia está relacionado não apenas ao poder que o agente possui em tomar decisões sem intervenções diretas ou indiretas de outros agentes, sendo eles humanos ou não, como também, em se relacionar a um ambiente, estando em permanente interação com ele e podendo modificá-lo em suas ações. Devido a essa característica de permanência em um ambiente, por exemplo, é possível distinguir esses agentes da maioria dos programas de computadores tradicionais.

Russell e Norvig (2013) enumeram quatro tipos básicos de programas de agentes que incorporam os princípios subjacentes a quase todos os sistemas inteligentes. O primeiro é o agente reativo simples, o tipo mais simples de agente. Ele seleciona suas ações com base na sua percepção atual, ignorando o restante do histórico de percepções. O segundo agente enumerado é o agente reativo baseado em modelo, que seleciona ações com base no seu histórico de percepções e assim reflete sobre alguns dos aspectos não observados do estado atual. O terceiro é denominado de agente baseado em objetivos que possui como característica, selecionar ações com base nos objetivos que descreve situações desejáveis. Porém, sozinhos, os objetivos não são suficientes para gerar comportamento de alta qualidade em todos os casos. Por isso, o quarto agente denominado de agente baseado em utilidades, seleciona suas ações que maximizam a utilidade esperada dos resultados da ação.

Ainda nesta mesma linha, Russell e Norvig (2013) descrevem um quinto agente denominado de agente com aprendizagem. Esse agente converte todos os agentes básicos citados anteriormente e pode melhorar o desempenho de seus componentes de modo a gerar melhores ações. Esse programa, a princípio, opera em ambientes inicialmente desconhecidos e com o conhecimento adquirido com passar do tempo, tornam-se mais eficientes do que quando tinham apenas o conhecimento inicial[2].

[2] Em Costa e Simões (2015), os autores, em seu livro, nomeiam esses cinco tipos agentes da seguinte forma: Agentes Reativos; Agentes de Procura; Agentes Baseados em Conhecimento; Agentes Aprendizes e Agentes

1.2 Motivação

Com o crescimento da Inteligência Artificial e das metodologias de desenvolvimento de agentes, preocupações sobre sua autonomia foram surgiram. Estas devem-se ao fato de que os agentes com autonomia interna estão assumindo cada vez mais o controle e o gerenciamento de suas atividades. Podemos citar por exemplo, os veículos automatizados e os sistemas de *e-commerce*. Assim sendo, a realização de testes para esses agentes tem se tornado um grande desafio, devido a esses programas com conhecimento e objetivos mutáveis apresentarem resultados diferentes em uma mesma entrada de teste ao longo do tempo (NGUYEN *et al.*, 2012).

Outro aspecto levantado por Padgham *et al.* (2013) em relação a seleção de entrada de teste, é que essas entradas apresentam problemas devido ao fato de que os agentes estão destinados a operar de forma robusta em condições que os desenvolvedores não consideram e, portanto, seria improvável testar. Nguyen *et al.* (2012) destacam que testes de agentes autônomos devem lidar com as características de que esses programas podem ser capazes de operar em ambientes abertos, onde circunstâncias diferentes, e até mesmo imprevisíveis, podem surgir.

Diferentes técnicas foram estudadas ao longo dos anos para encontrar formas eficientes de testar agentes autônomos, tais como os estudos empreendidos por: Zhang *et al.* (2007); Houhamdi (2011b); Nguyen *et al.* (2012). Nesse contexto, Houhamdi (2011a) afirma que uma boa avaliação para um agente depende dos casos de testes selecionados. Bons casos de teste devem gerar informações sobre o desempenho insatisfatório dos componentes na estrutura do agente e do funcionamento desses componentes.

Ao referir-se a realização de testes, Silveira (2013) menciona que nem sempre os melhores casos estão disponíveis nos primeiros testes, de maneira que, dependendo do ambiente de tarefa do agente, pode existir uma grande quantidade de casos a serem analisados. Outro aspecto levantado pela autora, é que: "um caso de teste ótimo é aquele em que o agente obtém o valor mínimo possível de desempenho", revelando assim, as falhas durante o ciclo de testes.

Em trabalhos mais recentes relacionados a testes de agentes, Silveira *et al.* (2014) apresenta uma abordagem baseada na seleção de casos de teste utilizando o algoritmo genético (AG) para testar o desempenho de agentes racionais. Em Carneiro *et al.* (2015), é apresentado a aplicação de sistemas imunológicos artificiais (AIS - *Artificial Immune Systems*), por meio do algoritmo de seleção clonal (CLONALG - *Clonal Selection Algorithm*), para a seleção de casos de teste. Em ambas as abordagens, um agente testador utiliza os algoritmos na geração inicial e

Adaptativos.

na seleção de novos casos de testes de acordo as características de cada algoritmo. Entretanto, em seus trabalhos, os autores deixam claro a possibilidade e a necessidade de aplicação de outros algoritmos multiobjetivos na otimização de seleção de casos de teste.

1.3 Objetivos

Devido aos algoritmos evolucionários serem capazes de solucionar problemas com múltiplos objetivos, este trabalho tem por finalidade utilizar uma abordagem baseada em um agente testador que utiliza quatro tipos diferentes de algoritmos evolucionários multiobjetivos (MOEAs - *Multiobjective Evolutionary Algorithms*) na geração de casos de testes que sejam capazes de contribuir com os testes de agentes racionais. Os algoritmos *Non-dominated Sorting Genetic Algorithm* (NSGA-II), *Strength Pareto Evolutionary Algorithm* (SPEA2), *Pareto Archived Evolution Strategy* (PAES) e *MultiObjective Cellular* (MOCell) foram escolhidos por serem largamente utilizados e representam diferentes estratégias de evolução para lidar com problemas de otimização.

O programa testador, orientado por uma função utilidade[3] e emprega estratégias de busca local dos MOEAs[4], baseadas em populações para encontrar conjuntos de casos de teste satisfatórios, ou seja, casos de testes onde o testador consiga minimizar o desempenho dos agentes testados. Ao final desse processo, são realizadas comparações entre os algoritmos utilizados a fim de identificar quais MOEAs são mais eficientes na geração de casos de testes que sejam capazes gerar informações relevantes sobre o desempenho insatisfatório e sobre as falhas do agente na escolha de suas ações.

1.4 Organização do trabalho

O conteúdo deste trabalho está organizado ao longo de seis capítulos, incluindo a atual introdução. Os demais capítulos são descritos a seguir:

Capítulo 2 – Fundamentação Teórica: Descreve os conceitos relacionados aos agentes racionais, seus tipos de programa agente e sua relação com a natureza dos ambientes onde atuam. O referido capítulo discorre, ainda, sobre testes de agentes em comparação aos testes de softwares tradicionais, e por fim, são apresentados os conceitos fundamentais sobre a

[3] A função utilidade é essencialmente uma internalização de uma medida de desempenho para o agente.

[4] Os algoritmos de busca local são úteis para resolver problemas de otimização, nos quais o objetivo é encontrar o melhor estado de acordo com uma função objetivo (RUSSELL; NORVIG, 2013).

otimização multiobjetivo e os algoritmos utilizados neste trabalho.

Capítulo 3 – Trabalhos Relacionados: Apresenta um referencial teórico destacando as principais discussões e resultados dos trabalhos que abordam a realização de testes de agentes destacando suas principais características.

Capítulo 4 – Abordagem Proposta para o Teste de Agentes: Descreve a metodologia utilizada com os principais aspectos da abordagem; a representação dos ambientes de tarefa utilizados; a história de um agente a ser testado em um ambiente; a medida de avaliação de desempenho; o funcionamento do agente testador e a utilização dos algoritmos multiobjetivos.

Capítulo 5 – Avaliação Experimental e Resultados: Apresenta como foram conduzidos os experimentos para demonstrar o funcionamento do agente testador e dos algoritmos multiobjetivos. Apresenta também o plano experimental, enfatizando os objetivos e a metodologia dos experimentos. Por fim, é contextualizada a avaliação e os resultados alcançados comparando os algoritmos multiobjetivos utilizados nos testes com quatro tipos diferentes de agentes.

Capítulo 6 – Conclusões: Relaciona as principais contribuições deste trabalho na resolução do problema de seleção de casos de testes, destacando também, as conclusões dessa pesquisa e os trabalhos futuros para o aperfeiçoamento e a continuidade deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é apresentada a fundamentação teórica sobre as áreas relacionadas à abordagem proposta. Inicialmente, examinamos os agentes e o ambiente de tarefa, e em seguida, são contextualizadas as características dos agentes racionais, destacando os tipos de programas de agentes racionais propostos por Russell e Norvig (2013) e Costa e Simões (2015). É contextualizada, ainda, a aplicação de testes de agentes, realizando uma comparação com os testes de softwares tradicionais. Finalizando este capítulo, a última seção aborda a otimização multiobjetivos e os algoritmos utilizados para a geração e seleção de casos de teste.

2.1 Agentes Racionais

A palavra agente vem do latim *agere*, que significa fazer, ou seja, um agente assim como todos os programas de computadores, é simplesmente algo que age. Agente é um programa computacional capaz de perceber seu ambiente por meio de sensores e agir em ambientes diferentes por intermédio de seus atuadores (RUSSELL; NORVIG, 2013).

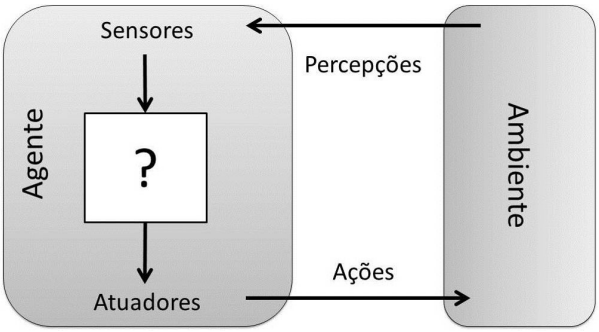
Costa e Simões (2015) entendem por agente, toda entidade capaz de interagir com o ambiente guiada, em geral (mas não necessariamente), por objetivos. Para os autores, um agente possui um mecanismo que permite recolher informações do ambiente (percepção), mecanismos que lhe permitem atuar sobre o ambiente (ação) e processos que definem qual a melhor ação a realizar (decisão).

Outro aspecto ainda levantado pelos mesmos autores, é que os agentes possuem ainda um conjunto de propriedades importantes:

> têm um **corpo**, existem no espaço e no tempo (têm uma **localização**), têm um conjunto de instrumentos para perceber e atuar sobre o ambiente (têm **capacidades**) e têm mecanismos de suporte à ação inteligente (mecanismo de **decisão**). Com base nestas propriedades, mas não necessariamente em todas, um agente constrói **estratégias** que lhe permitam ser eventualmente bem sucedidos nas **tarefas** que tem de resolver, sejam impostas pelo ambiente (externas), sejam determinadas por objetivos do agente (internas) Costa e Simões (2015, p. 29, grifo do autor) .

Na Figura 1, o agente possui sensores capazes de perceber alterações no ambiente e desempenha ações por meios de seus atuadores. Por exemplo, um agente de software pode receber sequências de teclas digitadas, conteúdo de arquivos e pacotes de redes e poderá atuar sobre o ambiente exibindo algo na tela, escrevendo em arquivos e enviando pacotes de redes com respostas.

Figura 1 – Agente interagindo com o ambiente.



Fonte – Russell e Norvig (2013).

Ao referir-se sobre agentes racionais, Coppin (2013) enfatiza que um agente é uma entidade capaz de realizar determinada tarefa mesmo quando os parâmetros de tarefas mudam ou quando há situações inesperadas. Outro aspecto levantado pelo autor, é que além de inteligência, outra característica importante é a autonomia, a capacidade de agir e tomar decisões independentes do programador ou do usuário do agente.

A inteligência de um agente está relacionada a uma ação racional. O conceito de racionalidade pode ser utilizado em diferentes tipos de agentes que operam em ambientes distintos. Uma forma de agir racionalmente, é raciocinar de modo lógico até uma ação pretendida e depois agir de acordo com suas conclusões (RUSSELL; NORVIG, 2013).

Para a definição de agente racional em qualquer instante dado, Russell e Norvig (2013) enfatizam que um agente dependerá de quatro fatores: (i) a medida de desempenho que define o critério de sucesso, (ii) o conhecimento anterior que o agente tem do ambiente, (iii) as ações que o agente pode executar e (iv) a sequência de percepções do agente até o momento.

Para cada sequência de percepções possível (história completa de tudo que o agente já percebeu), um agente racional deve selecionar uma ação que venha a maximizar sua medida de desempenho, dada a evidência fornecida pela sequência de percepções e por qualquer conhecimento interno do agente. Em termos matemáticos, o comportamento do agente é descrito pela função do agente f que mapeia qualquer sequência de percepções P^* específica para uma ação A (RUSSELL; NORVIG, 2013).

$$f : P^* \rightarrow A \tag{2.1}$$

Russell e Norvig (2013) destacam ainda que um agente racional, a princípio, age ao acaso ou recebe alguma assistência do seu projetista. A partir do momento que ele se baseia nas suas próprias percepções, aprendendo o que puder para compensar um conhecimento prévio

parcial ou incorreto e depois de adquirir experiência sobre seu ambiente, seu comportamento torna-se efetivamente independente de seu conhecimento anterior e com a incorporação do aprendizado, permite-se ainda, projetar um único agente racional que poderá atuar em uma ampla variedade de ambientes.

Em determinadas situações, agentes vivem em ambiente que envolve outros agentes. Temos então uma população de agentes onde possivelmente alguns possuem estratégias que possibilitam sua diferenciação de outros. Quando nos interessa considerar como um todo as suas estratégias e o resto do ambiente, temos então o conjunto chamado de sistema. Caso existam fortes interações entre os agentes da população, este sistema é dito como complexo[1]. Quando um sistema complexo contém agentes que buscam a melhoria de seu desempenho, dizemos que estamos em um sistema adaptativo complexo (SAC)[2] (COSTA; SIMõES, 2015).

Vimos que um agente se comporta tão bem quanto possível para maximizar sua medida de desempenho, dada a sequência de percepções recebida até o momento. Contudo, a medida da qualidade do comportamento de um agente depende da natureza dos ambientes encontrados por ele, haja vista que existem ambientes que são mais difíceis que outros (RUSSELL; NORVIG, 2013). Dessa forma, a próxima seção descreve as diferentes características desses ambientes de tarefas.

2.2 A natureza dos ambientes

O modo como um agente aprende as características do ambiente, dependerá efetivamente de sua capacidade. Logo, a complexidade de um ambiente, em grande parte, deverá estar relacionada à forma como os agentes são desenvolvidos (COSTA; SIMõES, 2015).

A primeira etapa ao projetar um agente deve ser sempre especificar os ambientes de tarefas da forma mais completa possível. Esses ambientes são chamados de PEAS (*Perfomance*, *Environment*, *Actuators*, *Sensors* – desempenho, ambiente, atuadores, sensores) que afetam diretamente o projeto apropriado para o programa do agente (RUSSELL; NORVIG, 2013).

Devido à variedade desses ambientes, Costa e Simões (2015) classificam as diferentes características em três aspectos principais:

• **Acessíveis ou não acessíveis:** se o agente puder tirar toda informação que necessita

[1] Os agentes da população podem ter ou não consciência de que estão interagindo entre si. Essa cooperação tem como objetivo, a resolução de uma determinada tarefa.

[2] Sistemas SAC são um tipo de sistema complexo onde as entidades e o ambiente são incentivados a se adaptar e interagir uns com os outros para atingir os objetivos e fornecer uma abstração mais realista de cenários da vida real (BIRDSEY, 2016).

para a escolha da melhor ação, esse ambiente será acessível. Caso contrário, o ambiente será não acessível. Exemplos de ambientes acessíveis são os casos de jogos abertos como o xadrez.

• **Deterministas ou não deterministas:** quando a evolução do ambiente não pode ser determinada de forma única diante da situação corrente e da ação do agente sobre o ambiente, este ambiente será denominado de não determinista. Caso o próximo estado seja determinado, o caso será de ambiente determinista. Um exemplo de ambiente não determinista é o caso de um sistema de diagnóstico médico, onde um paciente e o tratamento não evolui de forma determinista, visto que, para uma mesma doença, nem todos os pacientes reagem da mesma maneira ao mesmo tratamento.

• **Estáticos ou não estáticos:** um ambiente é estático, se ele não se alterar enquanto um agente está realizando uma ação. Caso ocorra alteração, esse ambiente não é estático. Ambientes de previsão do tempo tem que lidar com situações não estáticas.

Em Russell e Norvig (2013)[3], além dos três tipos de ambientes mencionados anteriormente, os autores consideram ainda outras três categorias importantes que determinam o projeto apropriado de agentes e a aplicabilidade das técnicas de implementação:

• **Episódico ou sequencial:** em ambientes episódicos, a experiência do agente é dividida em episódios atômicos , em outras palavras, não depende de ações passadas; caso contrário, é sequencial quando a decisão afeta as ações futuras. Muitas tarefas de classificação são episódicas. Por exemplo, para localizar peças defeituosas em uma linha de montagem o agente baseia cada decisão na peça atual. Por outro lado, jogar xadrez e dirigir um táxi são ambientes sequenciais. Em ambos os casos, ações em curto prazo podem ter consequências a longo prazo.

• **Discreto ou contínuo:** se as percepções e as ações são contáveis e distintas, o ambiente é discreto; caso contrário, é contínuo. Por exemplo, um ambiente de jogo de xadrez tem um número finito de estados distintos (excluindo o relógio), como também um conjunto discreto de percepções e ações. Já dirigir um táxi é um problema de estado e tempo contínuo.

• **Agente único ou multiagente:** se o ambiente é composto da interação de múltiplos agentes, o ambiente é multiagente; caso contrário, é um ambiente com agente único. Um exemplo do segundo ambiente é o caso do agente que resolve um jogo de palavras cruzadas sozinho, enquanto um agente que joga xadrez está em um ambiente de dois agentes, ou seja, multiagente.

Russell e Norvig (2013) realizam ainda uma distinção entre conhecido *versus* des-

[3] Em (RUSSELL; NORVIG, 2013), os três primeiros tipos de ambiente são denominados como: completamente observável ou parcialmente observável, determinístico ou estocástico e estático ou dinâmico.

conhecido. Essa distinção não se refere ao ambiente em si, mas ao estado de conhecimento do agente (ou do projetista). Em ambiente conhecido, as saídas são fornecidas para todas as ações. Se o ambiente for desconhecido, o agente terá de aprender como funciona, a fim de tomar boas decisões.

Essa distinção não é a mesma de ambientes totalmente e parcialmente observáveis de Russell e Norvig (2013) ou acessíveis ou não acessíveis citados por Costa e Simões (2015). Um ambiente conhecido pode ser parcialmente observável. Por exemplo, em jogos de cartas solitárias, nós conhecemos as regras, mas somos incapazes de ver as cartas que ainda não foram viradas. Por outro lado, um ambiente desconhecido pode ser totalmente observável. Neste caso, em um novo jogo, a tela pode mostrar o estado inteiro do jogo, mas não é possível saber o que os botões fazem até experimentá-los[4].

Após o que vimos nessa seção sobre os tipos e características de ambientes, podemos então representar um ambiente de tarefa como sendo um conjunto de estados $E = \{e_1, e_2, ...\}$. Em uma sala, exemplo citado em Costa e Simões (2015), os níveis de temperatura podem variar ao longo do tempo. Em algum instante, a temperatura pode estar alta (e_1), boa (e_2) ou a temperatura pode estar baixa (e_3). Já o agente interage com esta sala, capitando o seu estado e realizando um conjunto de ações sobre o ambiente $A = \{a_1, a_2, ...\}$ podendo levar à mudança do seu estado. Por exemplo, ligar o aquecimento (a_1), desligar o aquecimento (a_2) ou não fazer nada (a_3). Dessa forma, matematicamente, um agente pode ser definido pela seguinte função:

$$Agente : E^* \rightarrow A \tag{2.2}$$

Dada uma sequência de estados do ambiente (E^*), o agente reage com uma dada ação (A). Como veremos ainda na próxima seção, nos casos dos agentes mais simples (por exemplo: agente reativo simples em 2.3.1) essa condição não se verifica.

Ainda em relação ao ambiente, a mudança de estado também pode ser modelizada por uma função (Função 2.3), onde $\wp(E)$ representa o conjunto potência de E, significando que, dado um estado do ambiente (E) e uma ação do agente (A), existe não determinismo relativamente ao estado para que transita o ambiente.

$$Ambiente : E^* xA \rightarrow \wp(E) \tag{2.3}$$

[4] De acordo com as características apresentadas de cada categoria de ambientes, podemos observar que o caso mais difícil de ambiente é *parcialmente observável*, *multiagente*, *estocástico*, *sequencial*, *dinâmico*, *contínuo* e *desconhecido.*

2.3 Estrutura dos Agentes

O trabalho da Inteligência Artificial é projetar um programa de agente que implementa a função do agente f que mapeia percepções P^* em ações A . Assumimos que esse programa será executado em algum tipo de dispositivo de computação com sensores e atuadores físicos chamados de arquitetura:

Agente = arquitetura + programa

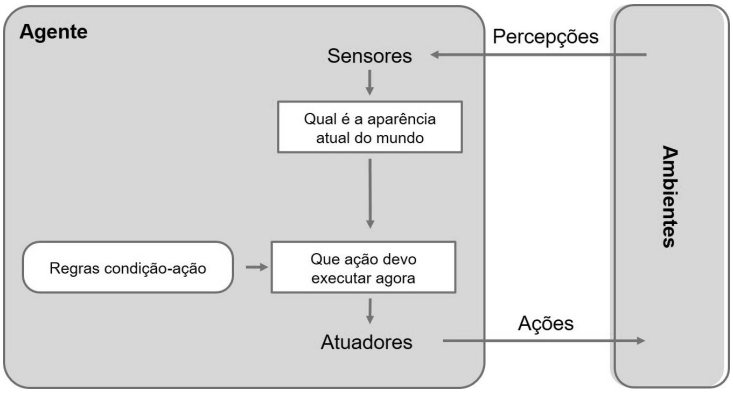
Obviamente, o programa que devemos escolher, tem que ser apropriado para a arquitetura. Se o programa vai realizar ações como *Caminhar*, a arquitetura teria que ter pernas. Arquiteturas podem ser apenas um computador comum, ou podem ser um carro robótico com vários computadores, câmeras e outros sensores a bordo (RUSSELL; NORVIG, 2013).

Cada tipo de agente combina componentes específicos de maneiras diferentes para gerar suas ações. Nas subseções a seguir, mostraremos os tipos de agentes que incorporam os princípios subjacentes à maioria dos sistemas inteligentes.

2.3.1 Agente reativo simples

A função *Agente* não é mais do que a composição das funções *Percepção* e *Ação* (Figura 2). Como visto anteriormente, podem existir agentes sem a capacidade de ter uma visão da história ocorrida anteriormente, ou seja, esse agente não possui a capacidade de guardar os estados passados do ambiente. Neste caso, o agente pode ser definido pela função 2.4 que faz o mapeamento da percepção do estado corrente em uma ação (COSTA; SIMõES, 2015).

Figura 2 – Representação de um agente reativo simples.



Fonte – Russell e Norvig (2013)

$$Agente : E \rightarrow A \tag{2.4}$$

Por exemplo, um agente aspirador de pó[5], é um agente reativo simples porque sua decisão se baseia apenas na posição atual e no fato de essa posição conter ou não sujeira. Esse aspirador deve possuir a habilidade de limpar um ambiente estático contendo duas salas (A e B) que podem conter sujeira ou não. Este ambiente é parcialmente observável, ou seja, o sensor do agente disponibiliza apenas informações locais a respeito da sala (A ou B) e do estado da sala (limpa ou suja) onde o agente está localizado. Assim, esse aspirador por meio de seus atuadores pode realizar as seguintes ações: virar à direita, virar à esquerda e aspirar. A Figura 3 mostra um programa para esse tipo de agente reativo simples.

Figura 3 – Programa agente reativo simples aspirador de pó.

```
função AGENTE-ASPIRADOR-DE-PÓ-REATIVO ([posição, situação]) retorna uma
ação*
    se situação = Sujo então retorna Aspirar
    senão se posição = A então retorna Direita
    senão se posição = B então retorna Esquerda
```

Fonte – Russell e Norvig (2013)

Em uma abordagem mais geral e flexível consiste em primeiro lugar, em construir um interpretador de uso geral para regras condição-ação e depois criar conjuntos de regras para ambientes de tarefas específicos. Essas regras permitem ao agente fazer a conexão entre percepção e ação. O programa desse agente é apresentado na Figura 4.

Figura 4 – Programa de um agente reativo simples.

```
função AGENTE-REATIVO-SIMPLES (percepção) retorna uma ação
    variáveis estáticas: regras, um conjunto de regras condição-ação
     estado ← INTERPRETAR-ENTRADA (percepção)
     regra ← REGRA-CORRESPONDENTE (estado, regras)
     ação ← AÇÃO-DA-REGRA [regra]
retornar ação
```

Fonte – Russell e Norvig (2013)

A função INTERPRETAR-ENTRADA gera uma descrição abstrata do estado atual

[5] Um agente aspirador de pó é utilizado por Russell e Norvig (2013) para exemplificar as características do comportamento de alguns tipos de agentes. Neste trabalho, será utilizada essa mesma analogia no comportamento dos agentes a serem testados

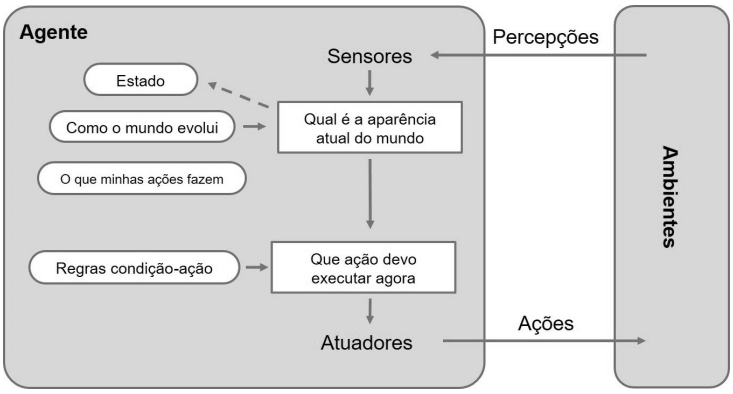
a partir da percepção, e a função REGRA-CORRESPONDENTE retorna à primeira regra no conjunto de regras que corresponde à descrição de estado (RUSSELL; NORVIG, 2013).

2.3.2 Agente reativo baseado em modelos

Para Russell e Norvig (2013), o modo mais eficaz de lidar com a possibilidade de observação parcial é o agente monitorar a parte do mundo que ele não pode ver, ou seja, o agente deve manter algum tipo de estado interno que dependa do histórico de percepções e assim analisar alguns dos aspectos não observados no estado atual.

A atualização dessas informações internas, à medida que o tempo passa, exige que dois tipos de conhecimento sejam codificados no programa do agente. Em primeiro lugar, precisamos de algumas informações sobre o modo como o mundo evolui, independentemente do agente. Em segundo, precisamos de algumas informações sobre como as ações do próprio agente afetam o mundo. Esse conhecimento é chamado de modelo do mundo e o tipo de agente é denominado de agente baseado em modelo (RUSSELL; NORVIG, 2013).

Figura 5 – Agente reativo baseado em modelo



Fonte – Russell e Norvig (2013)

A Figura 5 mostra como a percepção atual é combinada com o estado interno antigo para gerar a descrição atualizada do estado atual, baseado no modelo do agente de como o mundo funciona. Em relação ao programa do agente (figura 6), a parte interessante é a função ATUALIZAR-ESTADO, responsável pela criação da descrição do novo estado interno (RUSSELL; NORVIG, 2013).

Figura 6 – Programa do agente reativo baseado em modelo.

```
função AGENTE-REATIVO-BASEADO-EM-MODELOS (percepção) retorna uma ação
    persistente: estado, a concepção do agente do estado atual do mundo
                 modelo, o próximo estado depende do estado atual e da ação
                 regras, um conjunto de regras condição-ação
                 ação, a ação mais recente, inicialmente nenhuma
    estado ← ATUALIZAR-ESTADO (estado, ação, percepção, modelo)
    regra ← REGRA-CORRESPONDENTE (estado, regras)
    ação ← regra, AÇÃO
retornar ação
```

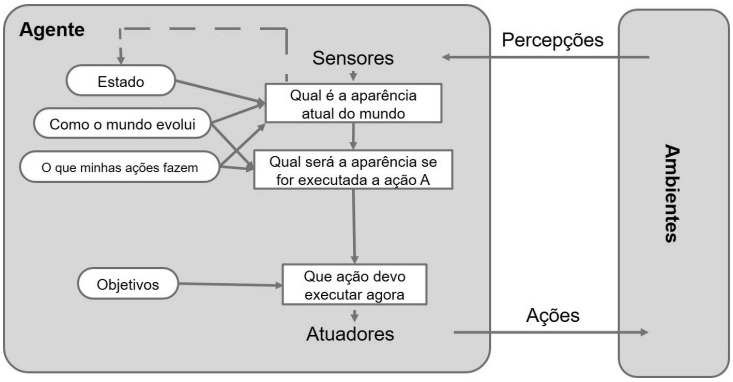
Fonte – Russell e Norvig (2013)

2.3.3 Agente baseado em objetivos

Nas seções anteriores analisamos uma classe de agentes simples. Esses agentes eram constituídos por dois módulos, um de percepção e um segundo de ação. Ao passarmos para um novo tipo de agente, adicionamos aos agentes a capacidade de decisão (COSTA; SIMõES, 2015).

Em outras palavras, da mesma forma que o agente precisa de uma descrição do estado atual, ele também precisa de alguma espécie de informação sobre objetivos que descreva situações desejáveis. O programa de agente pode combinar isso com o modelo (Agente baseado em modelo em 2.3.2), a fim de escolher ações que alcancem o objetivo. A estrutura desse agente baseado em objetivos é evidenciado na figura 7 (RUSSELL; NORVIG, 2013).

Figura 7 – Agente baseado em objetivos



Fonte – Russell e Norvig (2013)

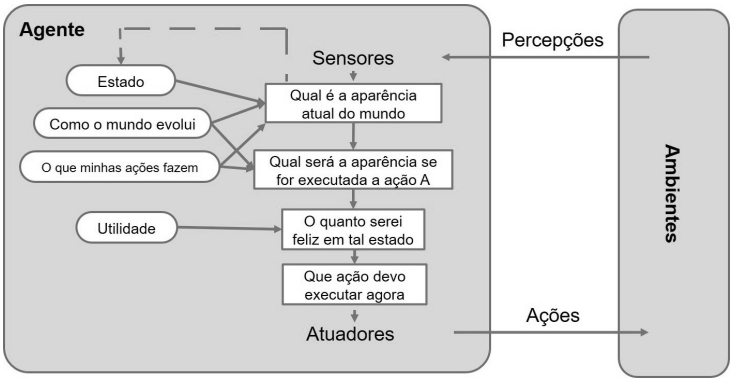
Embora o agente baseado em objetivos pareça menos eficiente, ele é mais flexível porque o conhecimento que apoia suas decisões é representado de maneira explícita e pode ser modificado. O comportamento pode ser alterado com facilidade para atingir um objetivo diferente, simplesmente, especificando um novo objetivo (RUSSELL; NORVIG, 2013).

2.3.4 Agente baseado na utilidade

Russell e Norvig (2013) destacam que sozinhos os objetivos não são realmente suficientes para gerar um comportamento de alta qualidade na maioria dos ambientes. Os objetivos simplesmente permitem uma distinção binária entre “estados felizes” e “infelizes”, enquanto uma medida de desempenho mais geral deve permitir uma comparação entre diferentes estados do mundo, de acordo com o grau exato de felicidade que proporcionariam ao agente. Essa medida é chamada de função utilidade.

Vimos que uma medida de desempenho atribui uma pontuação para qualquer sequência de estados do ambiente, e assim, ela pode distinguir facilmente entre formas mais e menos desejáveis de chegar aos objetivos. Se a função utilidade interna e a medida externa de desempenho estiverem em acordo, um agente que escolhe ações que maximizem a sua utilidade será racional de acordo com a medida de desempenho externa (RUSSELL; NORVIG, 2013).

Figura 8 – Um agente baseado em modelo e orientado para a utilidade.



Fonte – Russell e Norvig (2013)

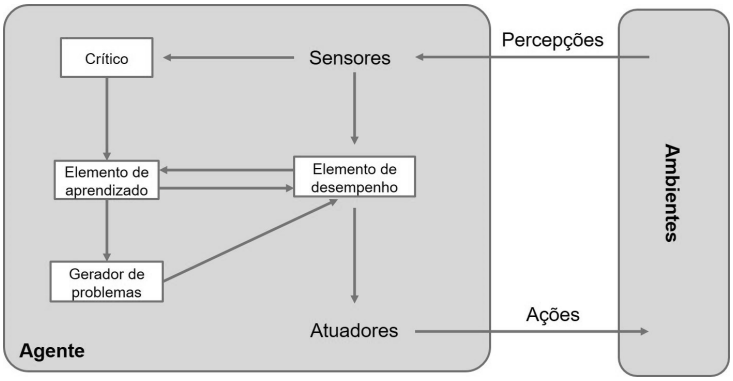
Um agente racional baseado em utilidade escolhe a ação que maximiza a utilidade esperada dos resultados da ação, isto é, a utilidade que o agente espera obter, em média, dadas as probabilidades e as utilidades de cada resultado. Esse agente tem de modelar e monitorar seu ambiente, tarefas que envolvem grande quantidade de pesquisas sobre percepção, representação, raciocínio e aprendizagem. A escolha da maximização de utilidade do curso da ação é uma tarefa difícil, exigindo algoritmos engenhosos. Mesmo com esses algoritmos, geralmente o raciocínio perfeito é inatingível na prática por causa da complexidade computacional. A estrutura desse agente baseado na utilidade aparece na Figura 8 (RUSSELL; NORVIG, 2013).

2.3.5 Agentes com aprendizagem

Aprender é uma característica essencial dos seres inteligentes. A área de Aprendizagem Artificial (AA) tem se tornado uma área central em Inteligência Artificial. Esse área tem três objetivos maiores: (i) o desenvolvimento de teorias computacionais da aprendizagem; (ii) a implementação de sistemas com capacidade de aprender; (iii) análise teórica e o desenvolvimento de algoritmos genéricos de aprendizagem (COSTA; SIMõES, 2015).

O aprendizado em agentes inteligentes pode ser resumido como um processo de modificação de cada componente do agente a fim de levar os componentes a um acordo mais íntimo com as informações de realimentação disponíveis, melhorando assim o desempenho global do agente. Esse tipo de agente pode ser dividido em quatro componentes conceituais (Figura 9). A distinção mais importante se dá entre o elemento de aprendizado, responsável pela execução de aperfeiçoamentos, e o elemento de desempenho, responsável pela seleção de ações externas (RUSSELL; NORVIG, 2013).

Figura 9 – Um modelo geral de agentes com aprendizagem.



Fonte – Russell e Norvig (2013)

O elemento de desempenho é o que foi considerado como sendo o agente completo: ele recebe percepções e decide sobre ações. O elemento de aprendizado utiliza realimentação do crítico sobre como o agente está funcionando e determina de que maneira o elemento de desempenho deve ser modificado para funcionar melhor no futuro. O último componente é o gerador de problemas. Ele é responsável por sugerir ações que levarão a experiências novas e informativas (RUSSELL; NORVIG, 2013).

2.4 Testes de Agentes

Teste de software é uma fase do desenvolvimento que visa realizar a avaliação da qualidade do produto e melhorá-lo por meio da detecção de erros. Esses testes são uma atividade na qual um sistema ou componente é executado em condições específicas. Os resultados são observados ou registrados e comparados com as especificações ou resultados esperados (HOUHAMDI, 2011a).

Malik e Khan (2016) salientam que testes tem um papel primordial no ciclo de desenvolvimento de software e para a arquitetura geral do sistema, pois, é vital garantir a qualidade do software antes de sua implantação e durante a sua manutenção.

Para Pressman (2011), a qualidade de um software pode ser definida como: uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam. O teste proporciona o último elemento a partir do qual a qualidade pode ser estimada e, mais pragmaticamente, os erros podem ser descobertos.

O teste de sistemas de software tradicionais, que têm comportamento estilo reativo (ou entrada-saída), é conhecido por ser não-trivial. O teste de agentes autônomos é ainda mais difícil, porque eles poderão engajar-se em um comportamento proativo particular (HOUHAMDI, 2011a).

A autonomia, por um lado, ajuda os agentes de software a lidarem com ambientes complexos e abertos. Por outro lado, faz com que testes de agentes se tornem uma tarefa desafiadora. Em uma mesma entrada de teste, os resultados podem ser diferentes em diferentes execuções. Por esta razão, os testes de agentes requerem um procedimento que sirva para uma grande variedade de contextos de casos de testes e que devem ser aplicados nos casos de testes mais exigentes. Desse modo, testar os sistemas para certificar-se de que eles se comportam adequadamente, torna-se uma premissa fundamental (NGUYEN *et al.*, 2012).

A referir-se a esse assunto, Houhamdi (2011a), Houhamdi (2011b) afirmam que existe uma procura por novas técnicas eficazes e adequadas para a avaliação do comportamento particular dos agentes autônomos, a fim de gerar confiança em seus resultados.

Em relação aos testes de softwares tradicionais, Malik e Khan (2016) realizam uma classificação conforme os seguintes níveis: unidade (teste individual de código fonte); integração (módulos do software são combinados e testados como um grupo); sistema (testa o sistema simulando o ambiente de execução); aceitação (usuário do sistema irá realizar os testes) e

componentes (teste de manipulação em vários componentes do sistema).

Entretanto, ao considerarmos os testes de agentes, Nguyen *et al.* (2010) classifica os testes em cinco tipos: unitário, agente, integração ou grupo, sistema ou sociedade e aceitação. Os objetivos e escopo de cada tipo são descritos a seguir:

• **Unitário**: testa unidades de códigos e módulos que compõem os agentes com metas, planos, convicções e raciocínio.

• **Agente**: testa funcionalidade interna e recursos do agente em cumprir seus objetivos de perceber e agir no ambiente de execução.

• **Integração ou Grupo**: testa a interação dos agentes, os protocolos de comunicação, a interação dos agentes com o ambiente e com os recursos compartilhados e observa também, as propriedades emergentes. Além disso, certifica-se de que um grupo de agentes e recursos do ambiente funcionem corretamente em conjunto.

• **Sistema ou Sociedade**: testa as propriedades do Sistema Multi-Agente (SMA); as propriedades de qualidade que o sistema deverá atingir, como adaptação, tolerância a falhas e desempenho.

• **Aceitação**: testa o SMA no ambiente de execução do cliente e verifica se ele atingiu os objetivos pretendido pelos interessados.

Ainda nesta mesma linha, Rouff (2002) identifica alguns tipos de falhas que os desenvolvedores buscam detectar ao analisar um único agente: (i) endereçamento incorreto de mensagem para outro agente; (ii) envio de solicitação incorreta em uma mensagem de modo que o agente recebedor não reconheça a mensagem; (iii) análise incorreta da mensagem recebida; (iv) verificação errônea de performance em uma mensagem recebida; (v) envio de mensagens erradas para outro agente e, (vi) o desempenho insatisfatório no código para processar todas as mensagens que são consideradas como parte da responsabilidade do agente.

Outro aspecto levantado por Rouff (2002), são os testes de comunidade de agentes. Nesses testes, está envolvida a certificação dos agentes da comunidade que devem trabalhar de forma conjunta como projetados. Além disso, desenvolvedores verificam se cada agente recebe as mensagens, se fornecem as respostas, e se interagem com o ambiente de forma correta. Para esses testes, os tipos de falhas são classificados da seguinte forma: (i) agente recebe mensagem errada; (ii) agente fornece uma resposta errada; (iii) agente que não responde ao ambiente de forma correta; (iv) agente não documenta a interação de forma correta e, (v) o acontecimento de *Deadlocks*.

2.5 Algoritmos Evolutivos Multiobjetivos

Problemas com múltiplos objetivos surgem de forma natural na maioria das áreas e sua solução tem sido um desafio para os pesquisadores. Apesar da considerável variedade de técnicas desenvolvidas em Pesquisas Operacionais (OR - *Operations Research*) e outras disciplinas, a complexidade da sua solução exige abordagens alternativas (COELLO GARY B. LAMONT, 2007).

O escopo de um problema mono-objetivo pode ser definido por um vetor n-dimensional $x = \{x_1, x_2, ..., x_n\}$ de variáveis de decisão, pertencentes ao universo de busca Ω, avaliados por uma função $f(x)$ de minimização ou maximização dos objetivos. Os problemas de natureza mais complexa, conhecidos como multiobjetivos, envolvem a manipulação de várias funções objetivos: $f_1(x), f_2(x), ..., f_k(x)$, onde k é o número de funções. Assim, esses multiobjetivos formam um vetor função $f(X) = [f_1(X), f_2(X), ..., f_k(X)]$ (COELLO GARY B. LAMONT, 2007).

Ao realizar uma comparação entre duas soluções U e V encontradas durante uma otimização mono-objetivo de minimização, por exemplo, envolve somente comparar a relação $f(U) < f(V)$, permitindo a fácil identificação de qual é a solução mais adequada. Entretanto, em uma comparação entre duas soluções de minimização com dois objetivos conflitantes, pode ocorrer uma situação no qual $f_1(U) < f_1(V)$ e simultaneamente $f_2(U) > f_2(V)$, ou seja, a solução U é melhor para o primeiro objetivo (f_1) porém a solução V é melhor para o outro objetivo (f_2). Neste caso, as duas soluções podem ser consideradas desejáveis.

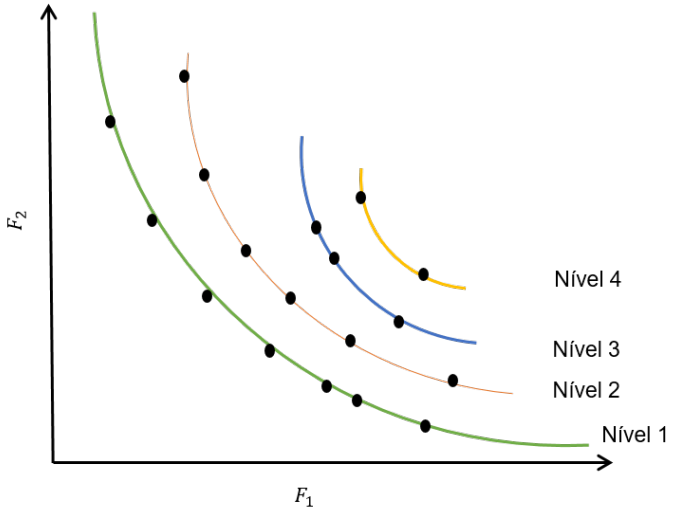
Muitos problemas de otimização são multiobjetivos, no sentido de que múltiplos e conflitantes critérios precisam ser otimizados simultaneamente. Devido ao conflito entre objetivos, geralmente, não existe uma única solução ideal. Em vez disso, o ideal corresponde a um conjunto de soluções, chamadas de Pareto-ótimo, que possuem os melhores valores em todos os objetivos (BROCKHOFF, 2011)

Dados duas soluções $F = (f_1, f_2, ..., f_m)$ e $F' = (f_1', f_2', ..., f_m') \in R^m$, F domina F' se $x_i \leq y_i$ para todo $i = 1, 2, ...m$ e $F \in F'$. Um ponto $X^* \in \Omega$ é chamado de Pareto ótimo se não houver um $X \in \Omega$ tal que $F(X)$ domine $F(X^*)$ (FAN *et al.*, 2016).

A figura 10 representa como as soluções são ordenadas por níveis de dominância: no primeiro nível (menor índice) estão as melhores soluções e no último nível (maior índice) estão as piores soluções em termos de dominância. Portanto, qualquer solução de um certo nível de dominância, é dominada por pelo menos uma solução de cada nível mais baixo. As soluções mais importantes são as não dominadas (nível 1), mas entre as dominadas podem existir vários

níveis de dominância (níveis 2, 3, 4).

Figura 10 – Classificação das soluções por níveis de dominância.



Fonte – Autor (2017)

Dentre os algoritmos de busca utilizados para solucionar problemas de otimização multiobjetivo, os Algoritmos Evolutivos têm se mostrado promissores (COELLO GARY B. LAMONT, 2007). Diante disso, esses algoritmos foram utilizados no presente trabalho para selecionar casos de testes para o teste de agentes racionais.

2.5.1 *Multiobjective Evolutionary Algorithms* - MOEAs

Inicialmente, problemas de grande importância tinham suas soluções baseadas nos métodos matemáticos. Porém, a complexidade desses modelos levaram os pesquisadores a concentrar seus esforços no desenvolvimento de *heurísticas*[6] com soluções baseadas nos fenômenos biológicos, sociais ou físicos observados na natureza (PONSICH *et al.*, 2013).

O uso de algoritmos evolutivos (*evolutionary algorithms* - EAs) para resolver problemas desta natureza, tem sido motivado principalmente por causa da natureza populacional que permite a geração de várias soluções em uma mesma execução. Além disso, a complexidade de alguns problemas de otimização multiobjetivos *(multiobjective optimization problems* - MOPs), como por exemplo, problemas com grande espaço de busca e de incertezas, podem impedir a utilização ou aplicação de técnicas tradicionais de Pesquisas Operacionais (OR) (COELLO

[6] O nome *heurística* é derivado da palavra grega *heuriskein*, que significa descobrir. Hoje esse termo é usado para descrever um método "que, baseado na experiência ou julgamento, parece conduzir a uma boa solução de um problema, mas que não garante produzir uma solução ótima".

GARY B. LAMONT, 2007).

Esta classe de *Metaheurísticas*[7] baseia-se na emulação da teoria *Darwiniana* (ou seja, o mecanismo de "sobrevivência do mais apto"), para evoluir de uma população de soluções para uma boa adaptação ao ambiente (para produzir soluções que são uma boa aproximação do ótimo global). Desde sua origem, EAs foram adaptados com sucesso para resolver problemas em muitas áreas (PONSICH *et al.*, 2013).

Uma característica essencial desejada ao abordar problemas multiobjetivos é a realização de uma distribuição uniforme das soluções eficientes ao longo da Fronteira de Pareto. Neste contexto, existe uma grande motivação para o uso de algoritmos evolucionários multiobjetivos (MOEAs - *Multiobjective Evolutionary Algorithms*) em vez de algoritmos evolutivos de mono-objetivo (SOEAs - *Single-objective Evolutionary Algorithms*) ou outras técnicas. A maioria dos MOEAs implementam procedimentos específicos que aplicam técnicas de preservação da diversidade (como o uso de métricas de densidade) como um critério secundário para avaliar a adequação da solução. Por outro lado, métodos de otimização mono-objetivo podem se concentrar em uma determinada região da Fronteira de Pareto e negligenciar outras (PONSICH *et al.*, 2013).

Após essa introdução sobre Algoritmos Evolutivos Multiobjetivos, são apresentados nas seções a seguir, quatro algoritmos multiobjetivos utilizados neste trabalho: *Non-dominated Sorting Genetic Algorithm* (NSGA-II), *Strength Pareto Evolutionary Algorithm* (SPEA2), *Pareto Archived Evolution Strategy* (PAES) e *MultiObjective Cellular* (MOCell).

2.5.2 *Non-dominated Sorting Genetic Algorithm* (NSGA-II)

O *Non-dominated Sorting Genetic Algorithm* (NSGA-II) Deb *et al.* (2002) é um algoritmo de otimização multiobjetivo baseado em algoritmos evolutivos e como característica principal, apresenta forte estratégia de elitismo. Esse algoritmo é baseado na classificação dos indivíduos da população em diversos níveis de não-dominância. Esse algoritmo varia do algoritmo genético simples apenas na forma em que o operador de seleção trabalha, uma vez que os operadores de cruzamento e mutação são aplicados da mesma forma do GA tradicional. O pseudocódigo do NSGA-II é apresentado na figura 11.

O algoritmo é inicializado com as entradas: tamanho da população N', números

[7] O termo *metaheurística* deriva da composição de duas palavras gregas: *heurística*, e o prefixo “meta”, que significa “após”, indicando um nível superior de descoberta. Pode-se dizer que *metaheurísticas* são mecanismos de alto nível para explorar espaços de busca, cada uma usando um determinado tipo de estratégia.

Figura 11 – Pseudocódigo que ilustra o algoritmo NSGA-II.

```
Entrada: N', g, f_k(X)
1   Inicializar população P'
2   Gerar população aleatória - Tamanho N'
3   Avaliar valores dos objetivos
4   Atribuir rank baseado na dominância de Pareto - Ordenação
5   Gerar população filho
6       Seleção por torneio binário
7       Cruzamento e Mutação
8   para i =1 até g faca
9       para cada Pai e Filho na População faca
10          Atribuir rank baseado na dominância de Pareto - Ordenação
11          Gerar fronteiras não dominadas
12          Ordenar cada solução das fronteiras considerando a distancia de multidão
            e Percorrer todas as fronteiras adicionando para a próxima geração do
            primeiro ao N'individuo
13      fim
14      Selecionar indivíduos das melhores fronteiras e com maior distancia de multidão
15      Gerar população filho
16          Seleção por torneio binário
17          Cruzamento e Mutação
18  fim
```

Fonte – Adaptado de Coello Gary B. Lamont (2007).

de gerações g e a função $f_k(X)$ a ser otimizada, onde k representa o número de objetivos. Em cada nova geração, o algoritmo ordena os indivíduos das populações de pais e filhos de acordo com a dominância entre as soluções, formando, assim, diversas fronteiras (linhas 10 e 11 do pseudocódigo da figura 11). A primeira fronteira é composta pelas soluções não dominadas de toda a população, a segunda é composta pelas soluções que passam a ser não dominadas após serem retiradas as soluções da primeira fronteira, a terceira fronteira é composta por soluções que passam a ser não dominadas após retiradas as soluções da primeira e segunda, e assim sucessivamente até todas as soluções estarem classificadas em alguma fronteira.

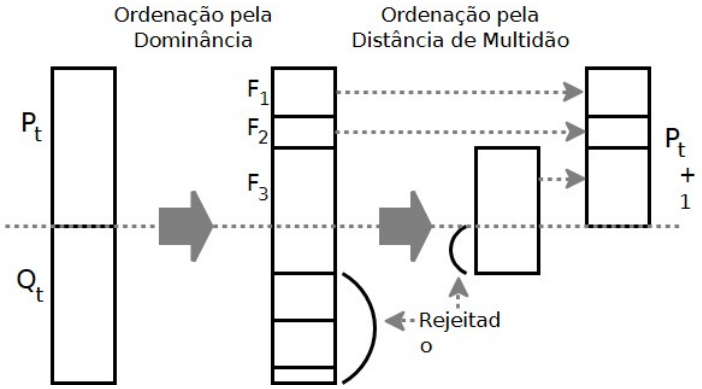
Para cada fronteira outra ordenação é feita usando uma medida, chamada distância de multidão (*crowding distance*), que tem como objetivo manter a diversidade das soluções e calcular o quão distante está a solução de seus vizinhos da mesma fronteira visando estabelecer uma ordem decrescente que privilegia as soluções mais espalhadas no espaço de busca. Como as soluções que estão no limite do espaço de busca apresentam só um vizinho, mas são as mais diversificadas da fronteira, elas recebem altos valores para estarem no topo da ordenação.

Ambas ordenações, de fronteiras e de distância de multidão, são usadas pelo operador de seleção (linhas 6 e 16 do pseudocódigo) e para determinar os indivíduos que sobrevivem

para a próxima geração. O NSGA-II utiliza a seleção por torneio, selecionando soluções de fronteiras com maior dominância e em caso de empate na dominação é utilizado como critério de desempate a distância de multidão.

A criação dos novos indivíduos é efetuada mediante a aplicação dos operadores de cruzamento e mutação (linhas 7 e 17 do pseudocódigo).O processo de ordenação em fronteiras, ordenação pela distância de multidão, e o elitismo são ilustrados na Figura 12, onde P_t é a população dos pais; Q_t é a população dos filhos; F_1, F_2 e F_3 são fronteiras de soluções já ordenadas da união de P_t e Q_t; e $P_t + 1$ representa o conjunto de soluções que serão usadas na próxima geração.

Figura 12 – Diagrama de Funcionamento do Elitismo no NSGA-II.



Fonte – Adaptado de Coello Gary B. Lamont (2007).

2.5.3 *Strength Pareto Evolutionary Algorithm* (SPEA2)

O algoritmo *Strength Pareto Evolutionary Algorithm* (SPEA2) Zitzler *et al.* (2001) apresenta como principal diferença em relação ao NSGA -II, a forma de cálculo do *fitness* e a utilização de um arquivo externo, separado da população regular, que é utilizado para armazenar as soluções não dominadas encontradas durante o processo evolutivo (Figura 13).

Os parâmetros de entrada do SPEA2 são os mesmos dos algoritmos NSGA-II, com o inserção do parâmetro referente ao tamanho do arquivo externo $A^{'}$. Em cada geração desse algoritmo é calculado para todas as soluções um valor chamado de *strength* que é utilizado para definir o *fitness* da solução. Esse valor de uma solução i corresponde ao número de indivíduos, pertencentes ao arquivo externo e a população regular, que dominam a solução i.

O *fitness* de uma solução i é calculado pela soma de todos os valores de *strength* das soluções dominadas por i, tanto do arquivo externo quanto da população regular (linha

Figura 13 – Pseudocódigo que ilustra o algoritmo SPEA2.

```
Entrada: N', A', g, f_k(X)
1   Inicializar população P' - Tamanho N'
2   Cria um arquivo vazio E'
3    para i =1 até g faca
4          Calcular o fitness para cada individuo de P' - Tamanho E'
5          Copiar todos os indivíduos não dominados de P' e E' para E'
6   se Tamanho de E' maior que A' então
7          Usar operador de eliminação de soluções de E'
8    senão se E' menor que A'então
9          Usar soluções dominadas de E' para completar E'
10  Fim
11  Executar seleção por torneio binário para preencher a mating pool
12  Aplicar Cruzamento e Mutação para a mating pool
13   Fim
```

Fonte – Adaptado de Coello Gary B. Lamont (2007).

4 do pseudocódigo da Figura 13). Valor de *fitness* igual a 0 indica que um indivíduo não é dominado por nenhuma outra solução, por outro lado, valores altos de *fitness* representam soluções dominadas por vários outros indivíduos.

Como o arquivo externo tem um tamanho fixo determinado por parâmetro, então durante o preenchimento do arquivo externo duas situações podem ocorrer: o arquivo pode estar com mais soluções não dominadas que seu limite, então um operador de eliminação de soluções é efetuado (linha 7 do pseudocódigo), calculando-se a distância das soluções para seus vizinhos e removendo-se as mais próximas. Por outro lado, caso o número de soluções não dominadas seja menor que o tamanho do arquivo, então este é preenchido com soluções dominadas (linha 9). Somente os indivíduos que compõem o arquivo externo sobrevivem para uma próxima geração. Para compor uma nova população, a criação dos novos indivíduos é efetuada mediante a aplicação dos operadores de cruzamento e mutação em indivíduos selecionados da população anterior e no arquivo externo (linha 12).

2.5.4 *Pareto Archived Evolution Strategy* (PAES)

No processo evolutivo do algoritmo *Pareto Archived Evolution Strategy* (PAES) Knowles e Corne (1999) o conceito de população é diferente das estratégias tradicionais de algoritmos evolutivos, pois apenas uma solução é mantida em cada geração. A estratégia para gerar novos indivíduos consiste em utilizar somente o operador de mutação, como pode ser observado na linha 3 do pseudocódigo da figura 14

Figura 14 – Pseudocódigo que ilustra o algoritmo PAES.

Entrada: $f_k(X)$
1 **Repita**
2 Inicializar população com um único pai C e adicionar para o arquivo A
3 Mutar C para produzir um filho C' e avaliar seu *fitness*
4 **se** $C < C'$ **então**
5 Descartar C'
6 **senão se** $C > C'$ **então**
7 Substituir C por C', e adicionar C' para A
8 **senão se** $\exists_{C'' \in A}(C < C')$ **então**
9 Descartar C'
10 **Senão**
11 Testar (C, C', A) para determinar qual será a solução que continuara no processo evolutivo, possibilitando adicionar C' para A
12 **Fim**
13 Até *atingir critério de parada;*

Fonte – Coello Gary B. Lamont (2007).

Uma vez que o algoritmo trabalha com apenas uma solução por geração não existe possibilidade de utilizar o operador de cruzamento. Assim como no SPEA2, existe um arquivo externo de soluções que é populado com as soluções não dominadas encontradas durante o processo evolutivo.

A cada geração, o algoritmo PAES cria uma nova solução filho que é comparada com a solução pai. Se a solução filho é dominada pela solução pai, a solução filho é descartada (linhas 4 e 5 do pseudocódigo da Figura 14), se a solução filho domina a solução pai, o filho toma o lugar do pai e o filho é acrescentado ao arquivo externo (linhas 6 e 7), se a solução filho for dominada por alguma solução do arquivo, o filho é descartado (linhas 8 e 9), e caso nenhuma das soluções (pai, filho e do arquivo) for dominante, a escolha da solução que vai permanecer no processo evolutivo é feita considerando a diversidade entre as soluções (linhas 10 e 11). É importante destacar ainda que o pseudocódigo do algoritmo PAES apresentado na figura 14 representa uma otimização de minimização.

Caso o tamanho do arquivo externo seja excedido, é aplicada uma estratégia de diversidade sobre este conjunto de soluções, eliminando soluções similares, mantendo a exploração de um espaço de busca maior.

2.5.5 *MultiObjective Cellular* (MOCell)

O algoritmo *MultiObjective Cellular* (MOCell) Nebro *et al.* (2007) usa um arquivo externo para armazenar as soluções não dominadas encontradas durante o processo de execução

como nos algoritmos evolutivos multiobjetivos PAES e SPEA2. O pseudocódigo desse algoritmo é apresentado na figura 15.

Figura 15 – Pseudocódigo que ilustra o algoritmo MOCell.

```
Entrada: f_k(X)
1   proc Steps Up(mocell) //Algorithm parameters in 'mocell'
2   Pareto front = Create Front() //Creates an empty Pareto front
3   while not TerminationCondition() do
4       for individual ← 1 to mocell.popSize do
5           n list←Get Neighborhood(mocell,position(individual));
6           parents←Selection(n list);
7           offspring←Recombination(mocell.Pc,parents);
8           offspring←Mutation(mocell.Pm,offspring);
9           Evaluate Fitness(offspring);
10          Replacement(position(individual),offspring,mocell,aux pop);
11          Insert Pareto Front(offspring);
12      end for
13      mocell.pop←aux pop;
14      mocell.pop←Feedback(mocell,ParetoFront);
15      end while
16      end proc Steps Up;
```

Fonte – Nebro *et al.* (2007).

Nesse algoritmo, a Fronteira de Pareto é uma população adicional (um arquivo externo) composta das soluções não-dominadas. O arquivo tem um tamanho máximo e, portanto, a inserção de soluções na Fronteira tem de ser cuidadosamente gerenciada para obter um conjunto diversificado. Assim, um estimador de densidade é necessário para remover soluções do arquivo, quando ele está completo.

O algoritmo começa com a criação de uma fronteira de Pareto vazia (linha 2 do pseudocódigo da Figura 15). Os indivíduos são organizados em um plano de duas dimensões, e os operadores genéticos são aplicados sucessivamente a eles (linhas 7 e 8 do pseudocódigo) até a condição de parada ser atendida (linha 3). Portanto, para cada indivíduo, o algoritmo seleciona dois pais na sua vizinhança, realizando a combinação entre os dois a fim de obter um descendente. Em seguida, é realizada a mutação e avaliação do indivíduo resultante. Ao final, o algoritmo decide se o novo indivíduo substitui o atual (linha 10). O próximo passo (linha 11) é a inserção do descendente no arquivo externo.

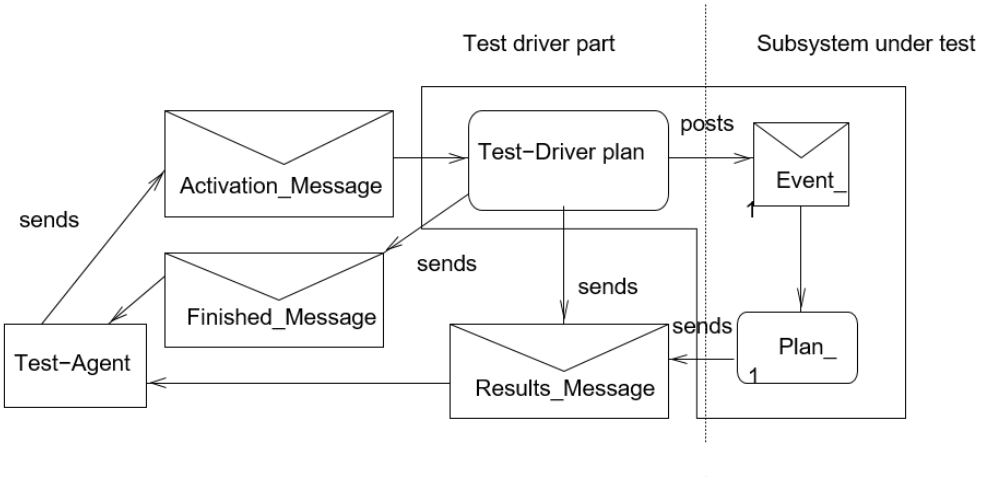
Finalmente, depois de cada geração, a população antiga é substituída pela auxiliar (linha 13) e um processo de *feedback* é chamado para substituir um número de indivíduos escolhidos aleatoriamente por um número de soluções do arquivo (linha 14).

3 TRABALHOS RELACIONADOS

Diferentes agentes autônomos foram testados usando diferentes técnicas ao longo dos anos (por exemplo, algoritmos evolutivos, metodologias de testes de softwares orientados a objetivos, agentes testadores) com o objetivo de verificar o comportamento dos agentes para atender as especificações e os desejos dos usuários. Este capítulo apresenta uma revisão bibliográfica sobre algumas abordagens presentes na literatura para realização de testes em agentes, independentemente de suas características e de sua arquitetura interna.

Em relação a testes de agente, cabe citar o trabalho de Zhang *et al.* (2007) que apresenta alguns aspectos de uma estrutura de testes desenvolvida para sistemas baseados em agentes. A estrutura é uma abordagem baseada em modelos de projeto da metodologia de desenvolvimento de agentes *Prometheus*. O trabalho focaliza no teste unitário apresentando mecanismos para gerar casos de teste adequados e determinar a ordem em que as unidades devem ser testadas (Figura 16).

Figura 16 – Quadro de Testes Abstratos para um Plano.



Fonte – Zhang *et al.* (2007).

O *Test-Agent* gera os casos de teste, e cada caso é executado por meio do envio de uma mensagem de ativação para o *test-Driver Plan*. O plano de *Test-Driver Plan* define a entrada e ativa o subsistema em teste que executa e envia informações para o *Test-Agent*. Quando o teste é concluído o *Test-Agente* gera um relatório sobre o desempenho do agente.

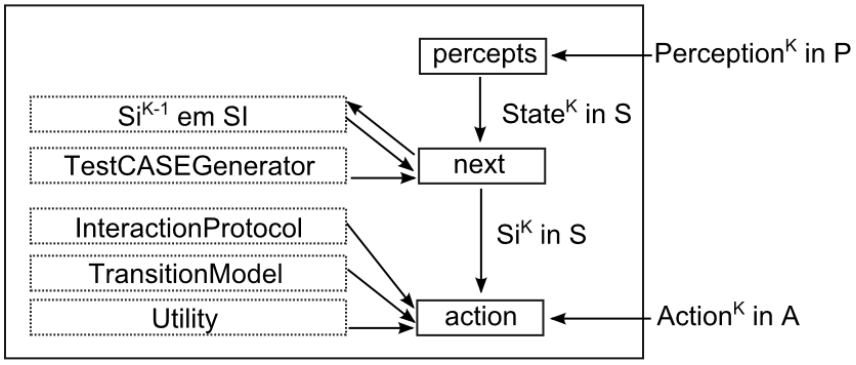
Também vale enfatizar a contribuição de uma proposta de teste orientada a objetivos de Houhamdi (2011b). Neste trabalho, o autor especifica um processo estruturado de testes que explora a relação entre análise de objetivos e casos de teste. Esta abordagem tem como referência

a metodologia Tropos (MYLOPOULOS; CASTRO, 2000). O autor ainda define um processo estruturado para a geração de testes nos agentes, fornecendo uma forma de derivar casos de teste a partir da análise de artefatos de requisitos orientado aos objetivos para a realização de dois níveis de teste: unitário e agente.

Uma abordagem evolucionária para a realização dos testes de agentes autônomos é adotada por Nguyen *et al.* (2012). O objetivo é estudar a eficácia dos testes evolutivos multiobjetivos, sob a observação de que, na realidade, o comportamento indesejável surge apenas quando agentes envolvidos em situações difíceis. A metodologia representa os objetivos dos *stakeholders* como função de qualidade e faz uso de algoritmos evolucionários guiados pela função de qualidade para gerar uma variedade de casos de teste exigentes para os agentes. A abordagem propõe aplicar um recrutamento dos melhores casos de teste para evoluir os agentes. Para cada agente, é dado um período experimental em que os testes com diferentes níveis de dificuldade são executados.

No trabalho de Silveira *et al.* (2014), os autores apresentaram uma abordagem para testar programas de agentes racionais, levando ao projetista, as informações relevantes sobre o desempenho do programa agente para melhorar a sua concepção e eficiência. O agente testador elaborado pelos autores, incorpora e processa as informações na formulação do problema de seleção e outras informações enviadas pelo projetista a fim de selecionar uma solução satisfatória com a finalidade de melhorar o desempenho do agente, se necessário. Este esquema de interação entre o testador deve ser contínuo até que o desempenho do agente ser considerado satisfatório. A Figura 17 ilustra a estrutura do agente testador.

Figura 17 – Estrutura do Agente Testador.



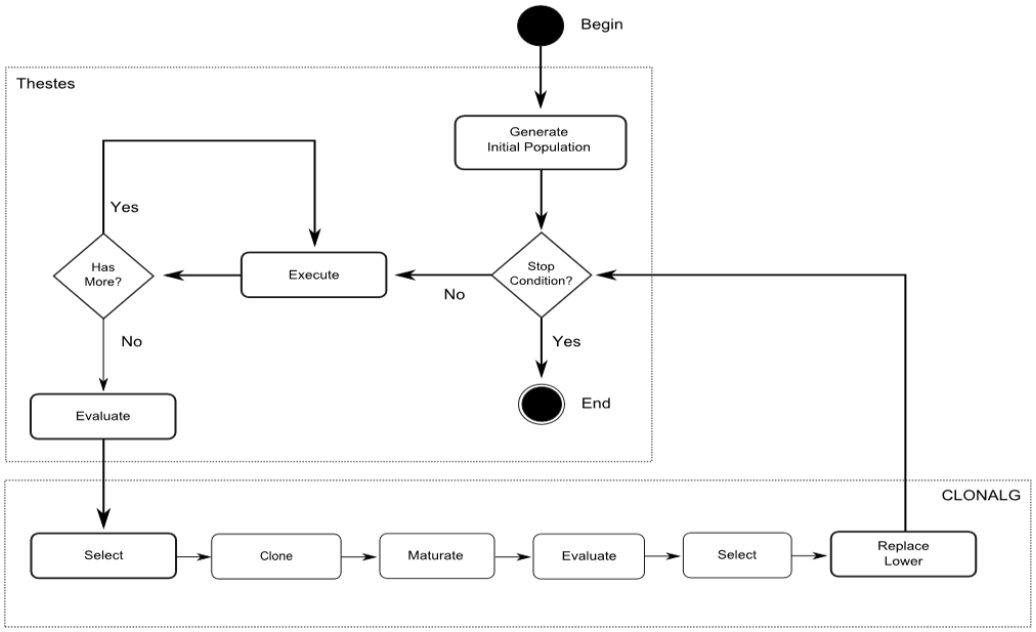
Fonte – Silveira *et al.* (2014).

O subsistema de percepção, *percepts*, é responsável por mapear as informações necessárias para testar o agente em uma representação computacional, $State^k$, útil para o proces-

samento dos outros dois subsistemas (*next* e *action*). O subsistema *next* atualiza o estado interno em $State^k$, e gera um conjunto inicial de casos de testes para testar certos aspectos da estrutura interna do agente. Considerando o estado interno atualizado, a função *action* inicia um processo de busca local para encontrar uma ação satisfatória. Esta função usa informações sobre uma transição de estado para gerar novos casos de teste. Ao final dos testes, o testador envia para o projetista as informações geradas pela função *action*. Silveira *et al.* (2014) ainda reforça que, a implementação do modelo baseia-se na análise populacional, baseados em *metaheurísticas* utilizando algoritmo genético (AG).

Ainda nesta mesma linha de um agente testador, em Carneiro *et al.* (2015), os autores apresentam uma aplicação de sistemas imunológicos artificiais (AIS - *Artificial Immune Systems*), por meio do algoritmo de seleção clonal (CLONALG - *Clonal Selection Algorithm*), para o problema de otimização de seleção de casos de teste para o teste de sistemas computacionais baseados em agentes inteligentes (Figura 18).

Figura 18 – Fluxograma de um agente testador utilizando o CLONALG.



Fonte – Carneiro *et al.* (2015).

O processo começa com a geração da população inicial de casos de testes e todos os casos da geração são executados, o que significa submeter o agente testado a cada um dos cenários configurados no passo anterior. Durante a execução, as histórias são armazenadas para serem avaliadas e submetidas ao CLONALG, que seleciona os melhores casos de testes e gera clones proporcionalmente à sua avaliação. Os clones gerados são submetidos a um processo

de maturação, em que cada indivíduo irá sofrer modificações com uma taxa inversamente proporcional à sua avaliação. Na fase final do CLONALG, os piores indivíduos da geração são substituídos pelos melhores clones e por novos indivíduos gerados de forma aleatória, para induzir a diversidade da população.

Os autores na validação de sua proposta, realizam comparação entre as técnicas de algoritmos genéticos (AG) e algoritmos de otimização por colônia de formigas (ACO - *Ant Colony Optimization Algorithm*). Seus experimentos foram realizados com agentes inteligentes com diferentes tipos de arquitetura em tipos de ambientes diferentes.

A partir da análise dos trabalhos supracitados, podemos destacar que uma boa avaliação de um agente autônomo depende dos casos de testes selecionados. Nem sempre, os melhores casos estão disponíveis nos primeiros testes, de maneira que, dependendo do ambiente de tarefa do agente testado, pode existir uma grande quantidade de casos a serem analisados. Dessa forma, a necessidade da geração automatizada de testes e a aplicação de diferentes algoritmos multiobjetivos para a seleção de casos de teste, foco deste trabalho, podem gerar de forma automatizada bons casos de testes que proporcione informações relevantes sobre o desempenho insatisfatório dos componentes dos agentes testados.

4 ABORDAGEM PROPOSTA PARA O TESTE DE AGENTES

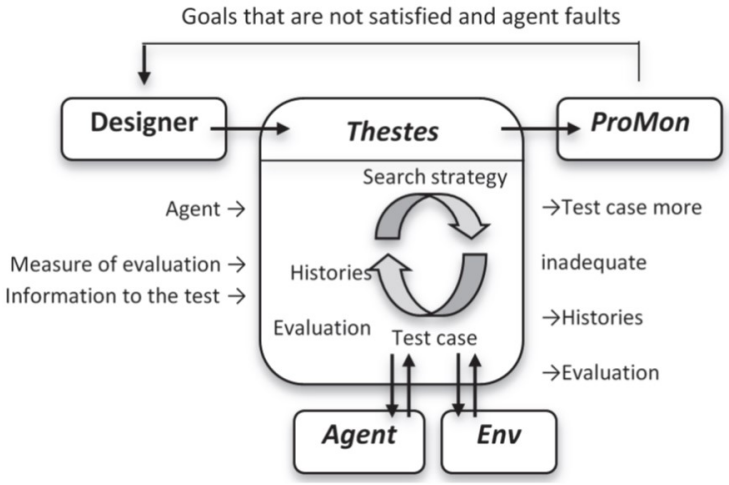
Este capítulo apresenta a abordagem proposta para a realização de teste de agentes racionais. Aqui são contextualizados os aspectos envolvidos na realização dos testes como: representação dos casos de testes; história de um agente em um ambiente de tarefa; o funcionamento do agente testador e o papel dos algoritmos evolutivos multiobjetivos (apresentados na seção 2.5) para selecionar os casos de teste mais apropriados.

4.1 Visão geral

A abordagem fundamenta-se na noção de agentes inteligentes de Russell e Norvig (2013), na interação dos agentes abordada em Silveira *et al.* (2015), na utilização de casos de teste gerados de acordo com os objetivos na medida de avaliação de desempenho do agente testado e nas estratégias de busca local multiobjetivo orientada por utilidade para encontrar casos de teste e histórias correspondentes em que o agente não foi bem avaliado.

Silveira *et al.* (2015) considera na abordagem, que além do *Designer*, existem quatro programas agentes envolvidos na seleção e testes: (i) o programa agente a ser testado, denominado *Agent*, concebido pelo projetista; (ii) o programa ambiente de tarefa, *Env*; (iii) um programa agente para a geração e seleção de casos de testes, *Thestes*, e (iv) um programa agente monitorador, *ProMon*. A Figura 19 ilustra as interações entre os agentes citados por Silveira *et al.* (2015).

Figura 19 – Visão geral da abordagem.



Fonte – Silveira *et al.* (2015).

Primeiramente, o *Designer* é o responsável pelo desenvolvimento do *Agent*, pela definição da medida de avaliação de desempenho e por outras informações necessárias para o agente *Thestes* executar o processo de teste do *Agent* em *Env*. O *Thestes* consiste em um agente de resolução de problemas de seleção de casos de teste que realiza busca local no espaço de estados de casos de teste[1] orientado por utilidade. Esse agente envia para o agente *ProMon* os casos de teste em que *Agent* obteve o comportamento mais inadequado, um conjunto de histórias correspondentes e seus valores de utilidade. O agente *ProMon* após receber essas informações, envia para o *Designer* os objetivos não satisfeitos adequadamente pelo *Agent* e as falhas cometidas por ele em *Env*.

Neste trabalho, o agente *Thestes* utiliza estratégias de busca local dos MOEAs, primeiramente na geração inicial de um conjunto de casos de testes e em seguida, entre os conjuntos avaliados pela função utilidade estabelecido em termos do valor de inadequação associado ao desempenho de *Agent* em *Amb*, a geração de um novo cojunto mais útil que o anterior.

Casos de teste consiste em uma representação do cenário em que o agente deverá atuar para ser avaliado. Dessa forma, a próxima seção define como será representado esses casos de testes durante este trabalho.

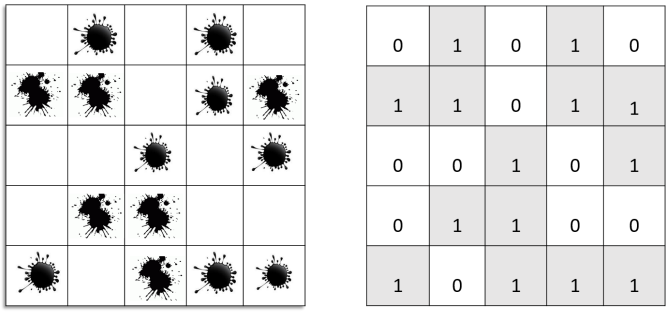
4.2 Casos de Teste

Um ambiente com diferentes representações faz com que um agente opere em um cenários com diferentes níveis de dificuldade. Para ilustrar isso, a Figura 20 consiste em um ambiente de tarefa para um agente aspirador de pó em mundo simplificado. O ambiente consiste em uma área contendo *n x n* salas que podem estar sujas (manchas escuras) ou limpas (Figura 20(a)).

Em uma outra alternativa possível para a representação desse tipo de ambiente, pode-se adotar um esquema de representação formada por *n* bits, em que o estado da sala é representado por 1, quando o agente está em uma sala que contém sujeira, e 0 quando ele estiver em uma sala limpa (Figura 20(b)). Novos casos de teste podem ser gerados apenas modificando a localização dos elementos no ambiente.

[1] Os algoritmos de busca local executam uma busca no espaço de estados, avaliando e modificando um ou mais estados atuais, em vez de explorar sistematicamente os caminhos a partir de um estado inicial e frequentemente podem encontrar soluções razoáveis em grandes ou infinitos (contínuos) espaços de estados para os quais os algoritmos sistemáticos são inadequados (RUSSELL; NORVIG, 2013).

Figura 20 – Ambiente de tarefa composto de 25 (n = 5) salas.



Fonte – Elaborado pelo autor (2017).

A partir de uma história do comportamento de um agente específico, é possível realizar sua avaliação, tendo em vista que todas as informações necessárias para esta avaliação sejam armazenadas. A seguir será explicado como esse comportamento é armazenado.

4.3 História de um Agente em um Ambiente

A história de um agente a ser testado em um ambiente, descreve a experiência do agente no ambiente em uma sequência de episódios da forma:

$$h : (Caso_i) : (Percepcao^0, Acao^0) \rightarrow (Percepcao^1, Acao^1 \rightarrow ... \rightarrow (Percepcao^k, Acao^k) \quad (4.1)$$

onde, o agente testado *Agent*: $P^* \rightarrow A$, em um ambiente, *Env*: $PxA \rightarrow P(P)$, obtém uma sequência de percepções, P^*, e escolhe uma ação para executar, A. A partir de uma percepção P e uma ação A, o ambiente envia para o agente um subconjunto de percepções, $P(P)$, obtidas a partir de P.

Com isso, a experiência do agente pode ser dividida em episódios atômicos, representados formalmente por meio de pares ordenados do tipo $Ep^K = (Percepcao^K, Acao^K)$, cujos elementos representam a percepção e a ação do agente na K-ésima interação com o ambiente.

4.4 Avaliação de Desempenho

Considerando que um agente racional deve realizar os objetivos implícitos na medida estabelecida pelo projetista, o processo de teste deve avaliar o desempenho do programa em seu ambiente de tarefa, identificando os objetivos que não estão sendo satisfeitos (casos satisfatórios) no ambiente, como também os componentes do ambiente interno do programa que estão restringindo o alcance dos objetivos esperados.

Nesse contexto, o projetista deve informar para o agente testador, a medida de avaliação do agente considerando um ou mais objetivos. A Tabela 1, mostra um exemplo de uma medida de avaliação de desempenho para um agente, que tem como objetivo limpar um ambiente, maximizando os níveis de limpeza e minimizando a energia gasta pelo agente.

Tabela 1 – Medida de Avaliação de Desempenho.

$Percepcao^k$	$Acao^K$	$av_E(Ep^k)$	$av_L(Ep^k)$
0	*Asp*	-1.0	0.0
0	*Dir, Esq, Ac, Ab*	-2.0	1.0
0	*N-op*	0.0	0.0
1	*Asp*	-1.0	2.0
1	*Dir, Esq, Ac, Ab*	-2.0	-1.0
1	*N-op*	0.0	-1.0

Fonte – Adaptado de Silveira *et al.* (2015).

A primeira coluna, $Percepcao^k$, descreve uma parte das informações nas percepções do agente em cada episódio possível, onde o número 0 representa salas que estão limpas e o número 1 identifica salas que possuem sujeiras. A coluna $Acao^K$ descreve as ações possíveis nesses episódios (Aspirar (*Asp*), Direita (*Dir*), Esquerda (*Esq*), Acima (*Ac*), Abaixo (*Ab*) e Não operar (*N-op*)). Na terceira e quarta colunas, $av_E(Ep^k)$ e $av_L(Ep^k)$ respectivamente, representam duas funções que estão associadas aos objetivos de energia e de limpeza para medir o desempenho do agente em cada episódio de sua história no ambiente. Assim, neste trabalho, consideramos esse tipo de medida de desempenho na avaliação dos agentes testados.

4.5 Agente Testador

O agente testador fundamenta-se na estrutura do programa agente orientado por utilidade (subseção 2.3.4) e emprega uma estratégia de busca local, baseado em algoritmos evolutivos multiobjetivos (MOEAs - *Multiobjective Evolutionary Algorithms*) apresentados na Seção 2.5, na geração de casos de testes. Essa busca é orientada por uma função utilidade para encontrar um conjunto de casos de testes satisfatórios, ou seja, os casos que possuem baixo desempenho entre as histórias associadas de *Agent* em *Env*.

O processo de teste deve avaliar o desempenho dos agentes em seu ambiente de testes, identificando os objetivos que não estão sendo satisfeitos e os componentes do programa que estão restringindo a satisfação desses objetivos. Assim, a eficácia do processo de teste do agente dependerá dos casos de teste selecionados.

Dessa forma, a seleção de um conjunto de casos de teste é um problema de busca em um espaço de estados composto por um grande número de conjuntos de casos possíveis. Um caso de teste considerado ótimo é aquele em que o agente obtém o valor mínimo possível, ou seja, minimização do desempenho do agente. Assim sendo, o problema de seleção de conjuntos de casos de teste para um agente racional foi formulado como um problema de otimização multiobjetivo (MOP - *multiobjective optimization problem*), tendo em vista que os agentes testador tem como objetivos, maximizar os níveis de limpeza do ambiente e minimizar a energia gasta em suas ações. Dependendo dos objetivos, pode não existir um conjunto solução que alcance os objetivos em todas as funções. Neste caso, a tarefa consiste em encontrar um conjunto de casos satisfatórios, levando o projetista a perceber as propriedades limitativas do programa.

A Figura 21 apresenta o esqueleto do programa agente testador. Internamente, $Estado^K$ recebe o mapeamento das informações necessárias em uma representação computacional pela função *ver*. A função *próximo*, é responsável por gerar novos *CasosTEST* a partir das informações contidas em um $Estado^K$ e gera um conjunto inicial de casos. Considerando o estado interno atualizado, a função *ação* inicia um processo de busca local visando encontrar uma ação satisfatória $Ação^K$ que utiliza informações a respeito de *ModeloTransição* de estados, para gerar novos casos de teste, do *ProtocoloInteração*, e uma função *Utilidade*, para obter as histórias correspondentes aos casos de teste no conjunto e avaliar o desempenho de *Agent* nestas histórias.

Figura 21 – Esqueleto do agente *Thestes*

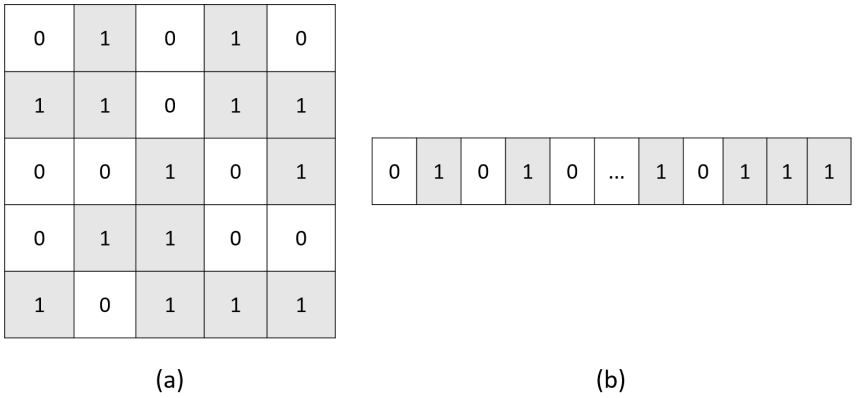
função *Thestes*(Percepção) **retorna** uma ação
  entradas: Percepção em $P_{Thestes}$
  var: Estado em E, Ei em EInterno, Ação em $A_{Thestes}$, GeradorCasosTEST,
    ProtocoloInteração, ModeloTransição, Utilidade
  $Estado^K \leftarrow$ ver($Percepção^K$)
  $Ei^K \leftarrow$ próximo($Estado^K, Ei^{K-1}$, GeradorCasosTEST)
  $Ação^K \leftarrow$ ação(Ei^K, ModeloTransição, ProtocoloInteração, Utilidade)
retornar $Ação^K$

Fonte – Adaptado de Silveira (2013)

A Figura 22 apresenta o ambiente de tarefa ilustrado na Figura 20(b) e um cromossomo equivalente expresso como uma cadeia de genes. No contexto dos algoritmos evolutivos aplicado ao problema de seleção proposto, um conjunto *CasosTEST* corrente, contendo as descrições de *m* ambientes de tarefa compostos de *n x n* salas, é representado por uma população

de indivíduos, ou seja, genes em cadeias de comprimento n^2, onde cada indivíduo na população codifica um ambiente e o valor de cada gene na cadeia codifica o estado da sala correspondente em termos de sujeira ou limpeza.

Figura 22 – Codificação de um caso de teste.



Fonte – Elaborado pelo autor (2017).

O processo do agente *Thestes* inicia com a geração da população inicial de casos de testes (uma população formada por um conjunto de indivíduos), que pode ser de maneira aleatória ou, dependendo dos parâmetros, casos específicos conforme o *Designer*. No próximo passo, todos os casos de testes da geração inicial são executados, o que significa submeter o *Agent* a cada um dos cenários. Na avaliação das histórias, o agente *Thestes* calcula o valor de cada medida de avaliação de desempenho. As histórias avaliadas são submetidas aos algoritmos evolucionários multiobjetivos, que selecionam os melhores casos de testes e geram novas populações ou modificam as populações existentes. Nesta fase é possível selecionar indivíduos mais capazes que possuem maior probabilidade de gerar mais descendentes, enquanto que os menos capazes poderão ainda gerar descendentes, porém em uma escala menor.

5 AVALIAÇÃO EXPERIMENTAL E RESULTADOS

Este capítulo apresenta como foram conduzidos os experimentos para demonstrar o funcionamento do agente testador dos algoritmos multiobjetivos utilizados. Assim, o estudo considerou o esqueleto do programa agente *Thestes* resolvendo problemas de seleção de casos de teste para programas agentes aspiradores de pó. Nos experimentos, foram testados dois tipos de programas agentes: um aspirador reativo simples (subseção 2.3.1) e outro baseado em modelos (subseção 2.3.2).

Este capítulo foi dividido em duas seções. A Seção 5.1 descreve o plano experimental, enfatizando os objetivos dos experimentos, definindo melhor o domínio em que a abordagem foi avaliada e a metodologia empregada. A Seção 5.2 descreve e analisa os resultados.

5.1 Plano Experimental

Na realização dos experimentos, a abordagem de comunicação entre os agentes foi implementada por meio do *framework* de desenvolvimento de agentes JADE[1] (*Java Agent Development Framework*) (BELLIFEMINE *et al.*, 2000). Na utilização dos MOEAs apresentados na subseção 2.5, foi utilizado o *framework* JMetal[2] (DURILLO; NEBRO, 2011), escrito na linguagem Java, cujos os algoritmos foram adaptados para atender ao problema de seleção de conjuntos de casos de testes deste trabalho.

Na validação, os experimentos consideraram o teste de programas agentes reativos simples (seção 2.3.1) onde suas decisões se baseiam apenas na posição atual e no fato de essa posição conter ou não sujeira e em um agente reativo baseado em modelo (seção 2.3.2), que seleciona ações com base no seu histórico de percepções e analisa os aspectos não observados do estado atual.

Esses dois tipos de agentes foram concebidos para funcionarem como aspiradores de pó em um ambiente determinístico e estático destacados na seção 2.2. Em relação à observabilidade do ambiente, os agentes atuaram em ambientes parcialmente e totalmente observáveis (seção 2.2). No primeiro tipo de ambiente, cada iteração, pressupõe que o aspirador possui sensores que percebem o ambiente, mas que a sua função *ver* consegue mapear apenas uma pequena representação desta informação. Em ambientes totalmente observáveis, em cada iteração,

[1] Sua arquitetura de comunicação tenta oferecer mensagens flexíveis e eficientes, escolhendo de forma transparente o melhor transporte disponível (BELLIFEMINE *et al.*, 2000).

[2] JMetal é uma estrutura baseada em Java, orientada a objetos voltada para o desenvolvimento, experimentação e estudo de *metaheurísticas* para resolver problemas de otimização multiobjetivo (DURILLO; NEBRO, 2011).

pressupõe-se que a função *ver* do agente consegue mapear o estado de todas as salas presentes no ambiente.

Os casos de testes iniciais, foram gerados de forma aleatória pelos algoritmos multiobjetivos testados, e os critérios de minimizar gastos de energia gasta para realizar uma ação e maximizar o número de salas limpas, foram utilizados como medidas de desempenho. Os experimentos realizados visam a validação e a análise de desempenho dos algoritmos NSGA-II, SPEA2, PAES e MOCell na seleção de conjuntos de casos de testes.

5.1.1 Ambiente de testes

O ambiente considerado para os experimentos dos agentes possui 25 salas, dispostas em forma de uma matriz 5 x 5, em que cada sala pode conter ou não sujeira (Figura 22). Em cada iteração com o ambiente, independente da sala, a função do agente pode escolher uma das seguintes Ações: aspirar (*Asp*), não operar (*N-op*), ou mover-se para outra sala vizinha (*Acima, Esquerda, Direita, Abaixo*). Como o ambiente é estático, cada caso de teste é instanciado em um programa ambiente (*Amb*) que obedece ao modelo determinístico.

Tabela 2 – Modelo Determinístico do Ambiente

$Percepcao^K$	$Acao^K$	$Percepcao^{K+1}$
L	*N-op*	*No, Oe,* ***L****, Le, Su*
L	*Asp*	*No, Oe,* ***L****, Le, Su*
L	*Ac*	***No****, Oe, L, Le, Su*
L	*Esq*	*No,* ***Oe****, L, Le, Su*
L	*Dir*	*No, Oe, L,* ***Le****, Su*
L	*Ab*	*No, Oe, L, Le,* ***Su***
S	*N-op*	*No, Oe,* ***S****, Le, Su*
S	*Asp*	*No, Oe,* ***L****, Le, Su*
S	*Ac*	***No****, Oe, S, Le, Su*
S	*Esq*	*No,* ***Oe****, S, Le, Su*
S	*Dir*	*No, Oe, S,* ***Le****, Su*
S	*Ab*	*No, Oe, S, Le,* ***Su***

Fonte – Elaborado pelo autor (2017).

A tabela 2 apresenta informações sobre as leis que governam as mudanças de estado de qualquer ambiente descrito em um caso de teste. Com base nessas mudanças, pode-se detectar que a percepção do próximo estado, $Percepcao^{K+1}$ presente na terceira coluna, é obtida quando a $Acao^K$ é executada sobre a $Percepcao^K$. Quando $Percepcao^K$ for igual a *L*, descreve o efeito das ações quando o agente está em uma sala limpa. Quando $Percepcao^K$ for igual a *S* descreve o

efeito das ações quando o agente está em uma sala suja. E quando a $Percepcao^{K+1}$ for igual a *No, Oe, Le, Su*, representa movimentos para as salas vizinhas à sala em que o aspirador está.

5.1.2 Experimentos

O primeiro agente testado foi denominado de *RS-Parcial* do tipo reativo simples, no qual a seleção da ação a ser executada baseia-se apenas na percepção atual do ambiente parcialmente observável, de forma que o agente consegue identificar apenas a sua localização e a presença ou não de sujeira (Tabela 3). As regras condição-ação do programa desse agente são apresentadas da seguinte forma: (i) se o estado da sala é *S*, então o agente realizará a ação de limpar; (ii) se o estado da sala é *L*, ou seja, está limpa, então ele realiza movimentos aleatórios (Acima, Esquerda, Direita, Abaixo, Não-Operar).

Tabela 3 – Regras condição-ação de RS-Parcial

$Percepcao^{K}$	$Acao^{K}$	$Percepcao^{K+1}$
S	*Asp*	*No, Oe,* ***L****, Le, Su*
L	*Ac ou Esq ou Dir ou Ab*	***No, Oe,*** *L,* ***Le, Su***

Fonte – Elaborado pelo autor (2017).

O segundo agente testado foi denominado de Agente reativo simples com alteração, *RS-Parcial-alterado*. Esse agente foi desenvolvido com o objetivo de verificar a sensibilidade da abordagem proposta e as possíveis falhas para que fosse possível avaliar o desempenho dos algoritmos com outros agentes que realizam suas ações avaliando suas percepções e seu histórico de salas já visitadas. Esse agente, toma suas ações independe do estado da sala na qual o agente se encontra. As regras condição-ação são apresentadas a seguir e na Tabela 4: (i) se o estado da sala é *S* ou *L*, o agente realiza movimentos aleatórios (Acima, Esquerda, Direita, Abaixo, Não-Operar).

Tabela 4 – Regras condição-ação de RS-Parcial-alterado

$Percepcao^{K}$	$Acao^{K}$	$Percepcao^{K+1}$
S	*Asp ou Ac ou Esq ou Dir ou Ab ou N-op*	***No, Oe, S, L, Le, Su***
L	*Asp ou Ac ou Esq ou Dir ou Ab ou N-op*	***No, Oe, L, Le, Su***

Fonte – Elaborado pelo autor (2017).

O terceiro agente testado (*RS-Total*) diferencia-se dos dois primeiros agentes no seguinte aspecto: a observabilidade do ambiente. Nesse agente, foi considerada a capacidade de

uma visibilidade total do ambiente. Assim, o agente movimenta-se em direção à sujeira mais próxima. A realização das regras condição-ação são apresentadas na tabela 5.

Tabela 5 – Regras condição-ação de RS-Total

$Percepcao^K$	$Acao^K$	$Percepcao^{K+1}$
S	*Asp*	*No, Oe,* ***L****, Le, Su*
L e PróximaSalaSuja(No)	*Ac*	***No****, Oe, L, Le, Su*
L e PróximaSalaSuja(Su)	*Ab*	*No, Oe, L, Le,* ***Su***
L e PróximaSalaSuja(Le)	*Dir*	*No, Oe, L,****Le****, Su*
L e PróximaSalaSuja(Oe)	*Esq*	*No,* ***Oe****, L, Le, Su*

Fonte – Elaborado pelo autor (2017).

O quarto e último agente testado (*RM-Parcial*) foi do tipo baseado em modelo com visibilidade parcial do ambiente, no qual uma representação interna do ambiente é mantida com todas as salas já visitadas, a fim de evitar que o agente visite salas já percorridas. As regras condição-ação do *RM-Parcial* são apresentadas na Tabela 6.

Tabela 6 – Regras condição-ação de *RM-Parcial*

$Percepcao^K$	$Acao^K$	$Percepcao^{K+1}$
S	*Asp*	*No, Oe,* ***L****, Le, Su*
L e NãoVisitou(No)	*Ac*	***No****, Oe, L, Le, Su*
L e NãoVisitou(Su)	*Ab*	*No, Oe, L, Le,* ***Su***
L e NãoVisitou(Le)	*Dir*	*No, Oe, L,****Le****, Su*
L e NãoVisitou(Oe)	*Esq*	*No,* ***Oe****, L, Le, Su*
L e visitou todas	*Asp ou Ac ou Esq ou Dir ou Ab ou N-op*	***No, Oe, L, Le, Su***

Fonte – Elaborado pelo autor (2017).

Neste contexto, os quatro agentes aspiradores de pó, para atingir um resultado satisfatório, devem limpar a quantidade máxima de sujeira presente no seu ambiente de teste, utilizando a menor quantidade de energia possível.

Para avaliar esse desempenho, a função *Utilidade* avalia os conjuntos gerados, considerando valores de pesos iguais a 0,5 tanto para o atributo limpeza quanto para o atributo energia ($w_L = w_E = 0{,}5$). Além desses valores, acrescentou-se a esta função um ganho por caso de teste que é igual ao número de salas sujas ao final das iterações de *Agent* com *Amb*. Assim, com a inclusão desse valor faz com que *Thestes* busque selecionar casos de teste em que *Agent* teve um comportamento inadequado em termos de energia e limpeza, dando privilégios aqueles casos em que o ambiente permaneceu com a maior quantidade de salas sujas ao final das interações.

A Tabela 7 apresenta os parâmetros de simulação utilizados em todos os experimentos. São gerados 20 casos (*NCasos*), ou seja, representa o tamanho da população, e cada caso de teste é formado por 25 salas. A avaliação de desempenho final em cada caso, leva em consideração a realização de 5 simulações (*Ns*), onde cada simulação dá origem a uma história contendo 25 episódios correspondentes às iterações de *Agent* em *Amb* (*Nint*).

Tabela 7 – Informações *ParâmetrosSimulação*

NCasos	n^2	N_{int}	*Ns*
20	25	25	5

Fonte – Elaborado pelo autor (2017).

O *ModeloTransição* genérico proposto deve considerar os *NCasos* no conjunto *CasosTEST* corrente e indicar como escolher os casos do conjunto corrente que serão modificados e as mudanças que serão realizadas nestes casos. Os parâmetros para utilização dos MOEAs foram ajustados para que cada um execute seu processo evolutivo específico, entretanto, diante de um ambiente similar. Os Valores dos Parâmetros Utilizados pelos MOEAs são apresentados na Tabela 8.

Tabela 8 – Valores dos Parâmetros Utilizados pelos MOEAs

Parametrização usando NSGA-II	
Método de seleção	BinaryTournament
Operador e taxa de cruzamento	SinglePointCrossover, probabilidade = 0.9
Operador e taxa de mutação	BitFlipMutation, probabilidade = 0.6
Parametrização usando SPEA2	
Método de seleção	BinaryTournament
Operador e taxa de cruzamento	SinglePointCrossover, probabilidade = 0.9
Operador e taxa de mutação	BitFlipMutation, probabilidade = 0.6
Tamanho do Arquivo Externo	20
Parametrização usando PAES	
Operador e taxa de mutação	BitFlipMutation, probabilidade = 0.6
Tamanho do Arquivo Externo	20
Parametrização usando MOCell	
Método de seleção	BinaryTournament
Operador e taxa de cruzamento	SinglePointCrossover, probabilidade = 0.9
Operador e taxa de mutação	BitFlipMutation, probabilidade = 0.6
Tamanho do Arquivo Externo	20

Fonte – Elaborado pelo autor (2017).

A Tabela 9 apresenta os experimentos realizados com a indicação dos números dos experimentos, dos tipos de agentes e dos ambientes empregados. Para cada experimento com

objetivo de gerar casos de testes satisfatórios, utilizou-se o agente testador *Thestes* e as estratégias de buscas utilizadas pelos quatro tipos de algoritmos evolucionários multiobjetivos utilizando as informações em *ParâmetrosSimulação* e em *ParâmetrosBUSCA*. Além disso, foi considerado como critério de parada, o número máximo de 30 gerações (iterações) para cada algoritmo.

Tabela 9 – Experimentos realizados.

Expe.	Nome	Tipo de agente	Ambiente
1	*RS-Parcial*	Reativo simples	Parcialmente observável
2	*RS-Parcial-alterado*	Reativo simples	Parcialmente observável
3	*RS-Total*	Reativo simples	Completamente observável
4	*RM-Parcial*	Baseado em modelo	Parcialmente observável

Fonte – Elaborado pelo autor (2017).

5.2 Apresentação e Análise dos Resultados

5.2.1 Experimento 1 - Resultados com RS-Parcial

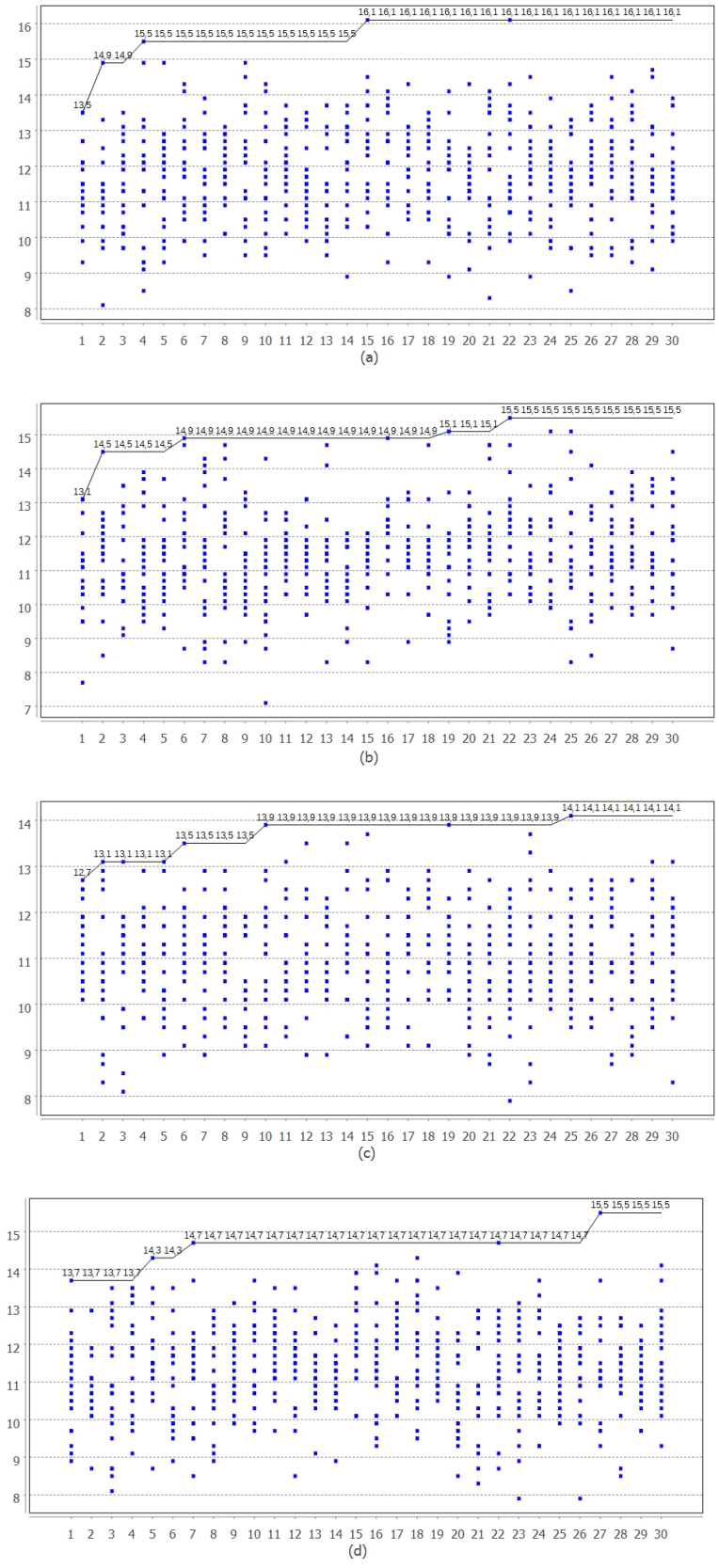
A Figura 23 ilustra os resultados do primeiro experimento obtidos por *Thestes* considerando a função *Utilidade*, conforme os valores de pesos $w_L = w_E = 0{,}5$ para os objetivos energia e limpeza ao longo de 30 gerações utilizando os quatro algoritmos multiobjetivos. Na Figura 23(a) são apresentados os resultados do algoritmo NSGA-II; na Figura 23(b) os resultados para o algoritmo SPEA2; na Figura 23(c) para o algoritmo PAES e na Figura 23(d) os resultados para o algoritmo MOCell.

Os pontos não lineares representam os 20 casos de teste na população (*CasosTEST*) de cada algoritmo em cada geração. Os pontos marcados linearmente identificam o melhor caso de teste na população a cada geração. A utilização do elitismo pelos algoritmos, tem como objetivo, prevenir a perda do melhor caso de teste encontrado em uma geração anterior.

A Figura 24 ilustra a variação do valor de utilidade dos 20 casos de testes em cada geração. O box-plot é formado pelos quartis inferior (25%), mediana (50%) e superior (75%), assim como os valores máximo e mínimo entre os casos. Em (a) é apresentado o box-plot do NSGA-II, em (b) do SPEA2 e (c) o algoritmo PAES e em (d) o box-plot do algoritmo MOCell.

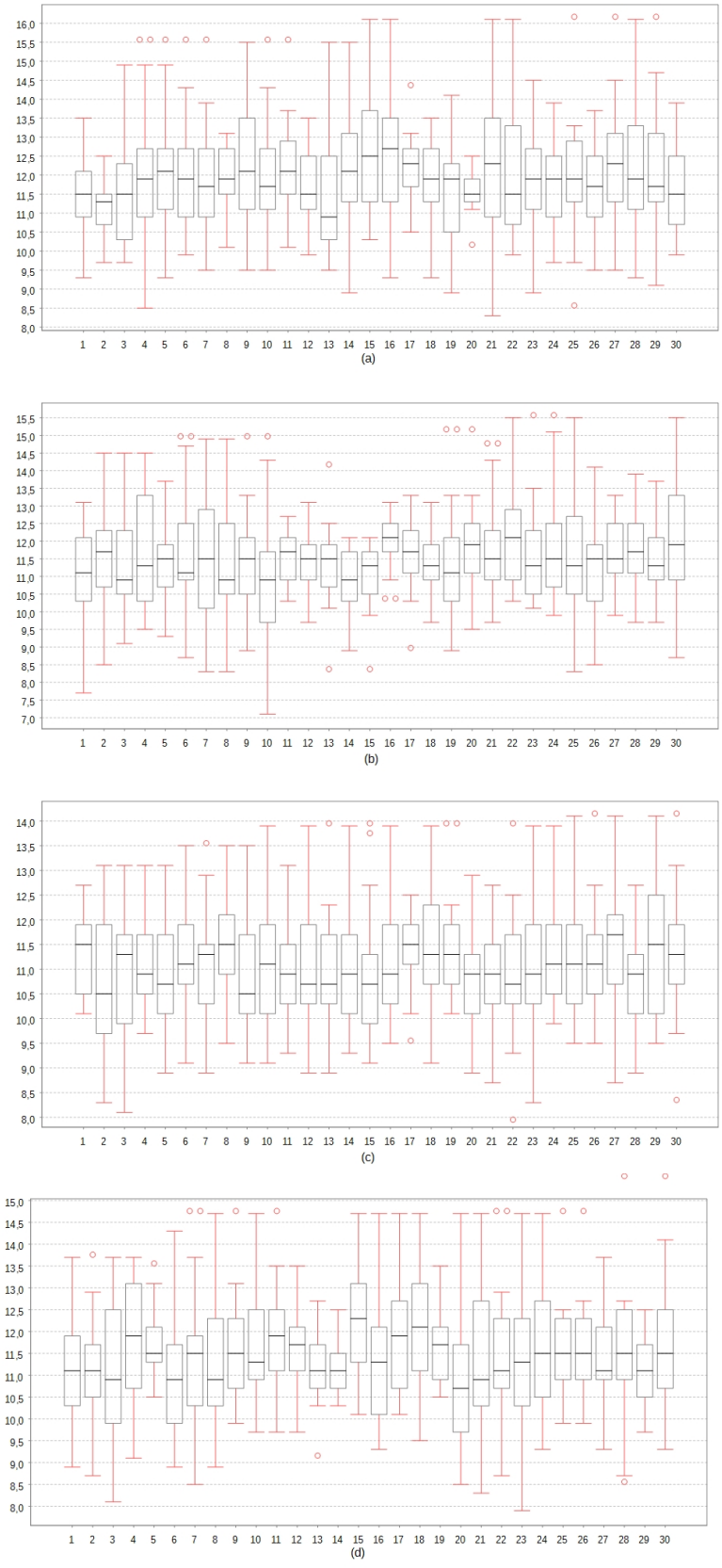
O algoritmo NSGA-II (Figura 23(a)) apresentou o melhor valor do *fitness* em dois momentos, na 15ª e 22ª geração com utilidade = 16,1. Na utilização do algoritmo SPEA2 (Figura 23(b)), o melhor valor da função é obtido também na 22ª geração, entretanto, com

Figura 23 – Utilidade de casos de testes em 30 gerações (Experimento 1).



Fonte – Elaborado pelo autor (2017).

Figura 24 – Box-plot dos valores de utilidade em 30 gerações (Experimento 1).



Fonte – Elaborado pelo autor (2017).

utilidade = 15,5. Para o algoritmo PAES (Figura 23(c)), o melhor valor é obtido na 25ª geração com utilidade = 14,1 e para o algoritmo MOCell, o melhor valor é obtido apenas na 27ª geração com utilidade = 15,5 (Figura 23(d)).

A Tabela 10 destaca a geração e o melhor valor de utilidade encontrado por cada algoritmo.

Tabela 10 – Geração e Utilidade do melhor caso (Experimento 1)

Algoritmo	Geração	Utilidade
NSGA-II	15 e 22	16,1
SPEA2	22	15,5
PAES	25	14,1
MOCell	27	15,5

Fonte – Elaborado pelo autor (2017).

Podemos destacar também, o número de vezes (4 vezes) que os algoritmos SPEA2 e o PAES conseguiram evoluir suas populações de casos de testes. O segundo algoritmo começou com a pior geração (utilidade = 12,5) e alcançou no final uma geração de utilidade = 14,1, entretanto, não conseguiu ser melhor que os demais algoritmos.

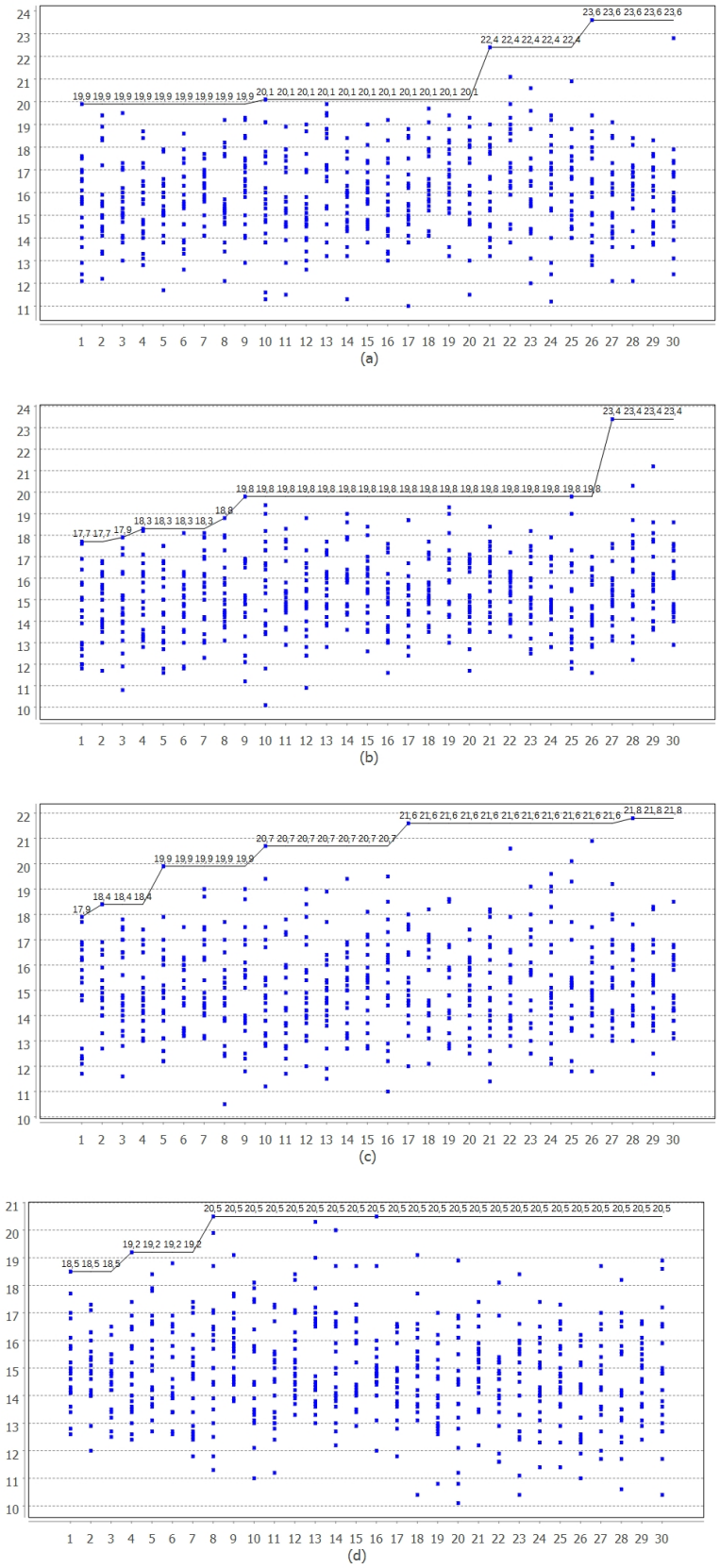
Assim, temos ao final desse primeiro experimento, considerando os melhores casos de teste, o NSGA-II com o melhor resultado, seguido pelos algoritmos SPEA2 e MOCell e, com o pior desempenho, o algoritmo PAES.

5.2.2 Experimento 2 - Resultados com RS-Parcial-Alterado

Nesse segundo experimento (Figura 25), podemos observar que o programa agente *RS-Parcial-alterado* é mais inadequado que o agente *RS-Parcial*. Isso ocorre devido as regras condição-ação do agente *RS-Parcial-alterado* serem definidas aleatoriamente, possibilitando que o subsistema de tomada de decisão seja capaz de selecionar ações que podem produzir episódios com falhas, tornando o valor de Utilidade mais inadequado. A Figura 26 ilustra a variação do valor de utilidade dos 20 casos de testes em cada geração.

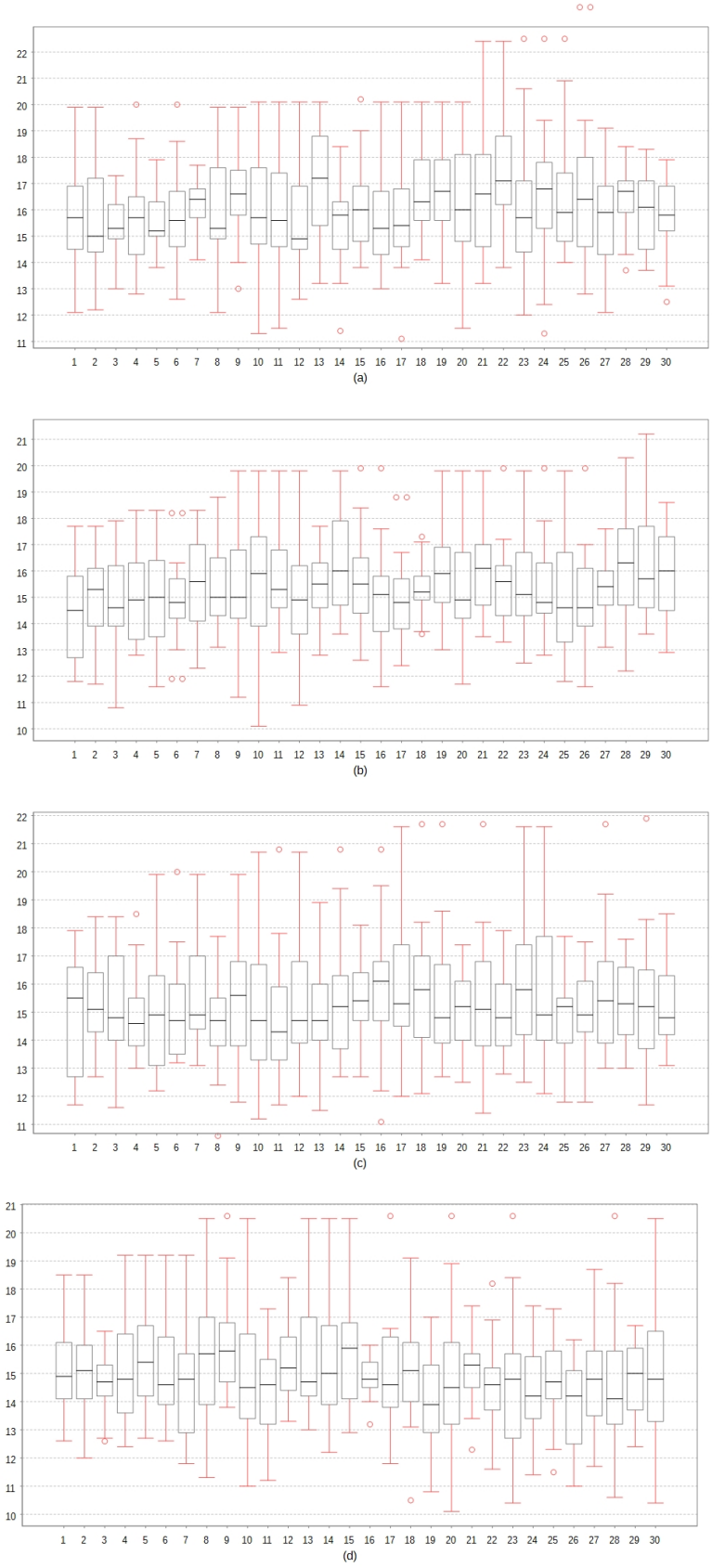
A Tabela 11 destaca a geração e o melhor valor de utilidade encontrado por cada algoritmo nesse segundo experimento. Na utilização do algoritmo NSGA-II (Figura 25(a)), o melhor valor da função *Utilidade* é obtido na 26ª geração com utilidade = 23,6. Nas próximas gerações, esse valor é tomado apenas como referência e não é selecionado nenhum caso de teste com valor de utilidade superior. Para o algorítimo SPEA2 (Figura 25(b)), o melhor valor da

Figura 25 – Utilidade de casos de testes em 30 gerações (Experimento 2).



Fonte – Elaborado pelo autor (2017).

Figura 26 – Box-plot dos valores de utilidade em 30 gerações (Experimento 2).



Fonte – Elaborado pelo autor (2017).

função é obtido na 27ª geração com utilidade = 23,4.

Tabela 11 – Geração e Utilidade do melhor caso (Experimento 2)

Algoritmo	Geração	Utilidade
NSGA-II	26	23,6
SPEA2	27	23,4
PAES	28	21,8
MOCell	8	20,5

Fonte – Elaborado pelo autor (2017).

Para o algoritmo PAES (Figura 25(c)), o melhor valor é obtido na 28ª geração com utilidade = 21,8 e para o último algoritmo, MOCell (Figura 25(d)), o melhor valor da função *Utilidade* é obtido na 8ª geração com utilidade = 20,6, entretanto, o algoritmo não conseguiu mais gerar casos de testes com utilidade superior a partir dessa geração.

Dessa forma, considerando os melhores casos de teste, o algoritmo NSGA-II e o SPEA2, foram os algoritmos que obtiveram os melhores resultados, seguidos pelo algoritmo PAES e o algoritmo MOCell apresentou o pior desempenho.

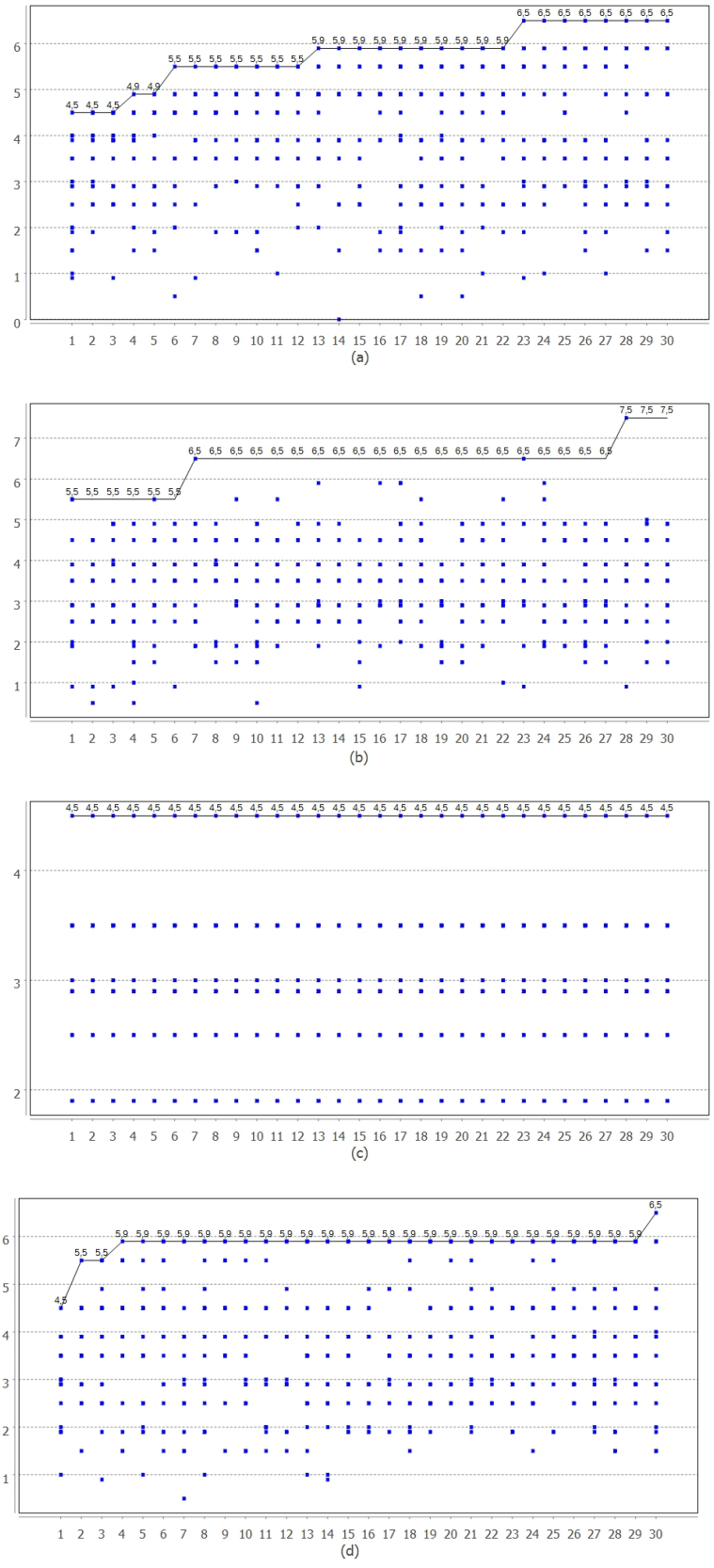
5.2.3 Experimento 3 - Resultados com RS-Total

A Figura 27, apresenta os experimentos do terceiro Agente. O valor de inadequação do agente reativo simples em um ambiente totalmente observável é bem menor que os valores dos outros dois agentes em um ambiente parcialmente observável. *RS-Total* percebe todo o ambiente e isso permite a concepção de um subsistema de tomada de decisão capaz de selecionar as ações que sejam realmente racionais em cada interação com o ambiente. A Figura 28 ilustra a variação do valor utilidade nos 20 casos em cada geração.

Na utilização do NSGA-II (Figura 27(a)), o algoritmo obteve o melhor valor da função *Utilidade* na 23 geração com utilidade = 6,5. Na utilização do algoritmo SPEA2 (Figura 27(b)), o melhor valor da função é obtido também na 28 geração com utilidade = 7,5. Na utilização do PAES (Figura 27(c)), o algoritmo não conseguiu gerar casos de testes mais eficientes, tendo o mesmo valor de utilidade em todas as gerações. Para o algoritmo MOCell, o melhor valor é obtido apenas na 30 geração com utilidade = 6,5 (Figura 27(d)).

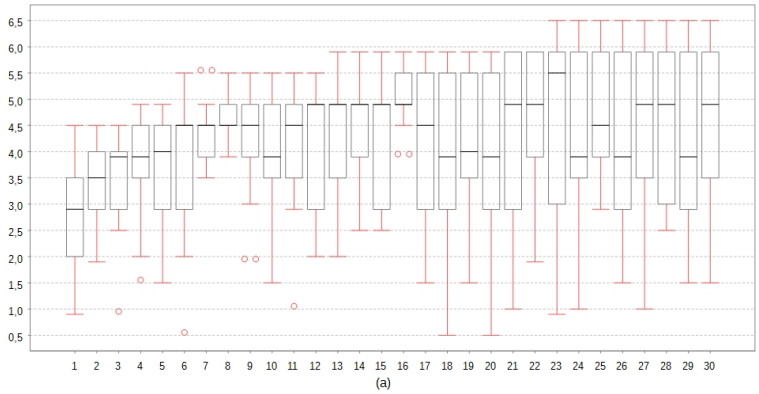
A Tabela 12 destaca a geração e o melhor valor de utilidade encontrado no terceiro experimento pelos quatro algoritmos com o agente reativo simples em um ambiente totalmente observável. Nesse experimento, temos como maior destaque negativo, a ineficiência do algoritmo

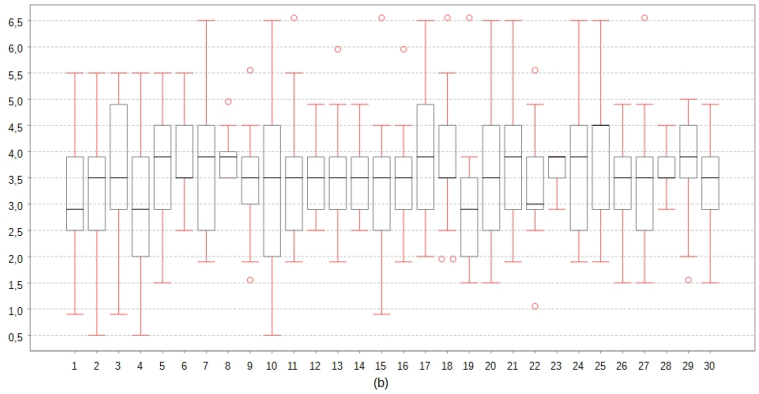
Figura 27 – Utilidade de casos de testes em 30 gerações (Experimento 3).

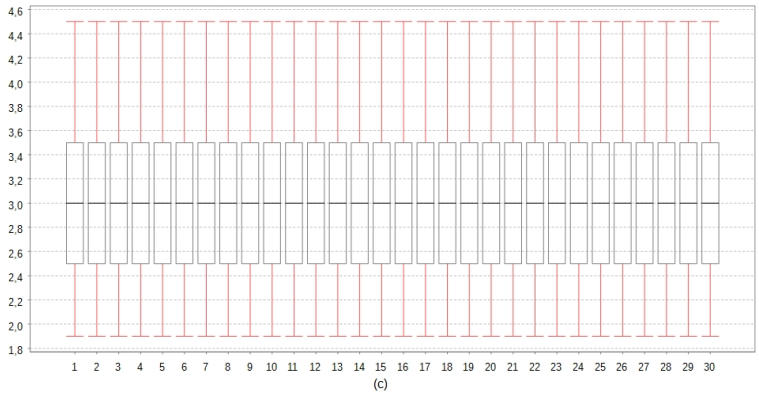


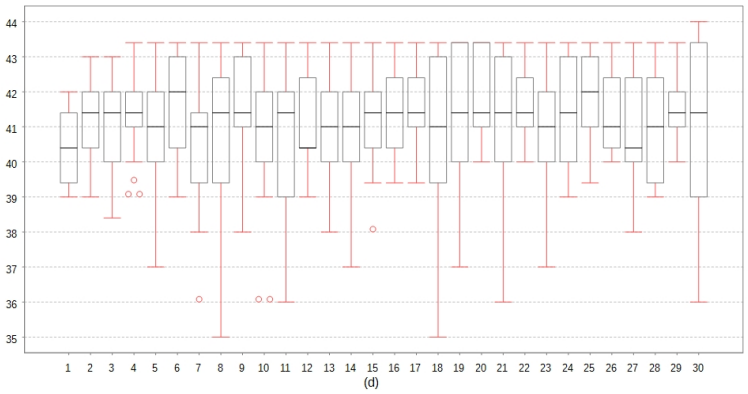
Fonte – Elaborado pelo autor (2017).

Figura 28 – Box-plot dos valores de utilidade em 30 gerações (Experimento 3).









Fonte – Elaborado pelo autor (2017).

PAES em gerar casos de testes para um agente reativo simples que possua a observabilidade total do seu ambiente de tarefa. Isso pode ser justificado devido o algoritmo em seu processo evolutivo, manter apenas a melhor solução em cada geração e a estratégia na geração de novos indivíduos, consiste em utilizar somente o operador de mutação diferente das estratégias tradicionais de algoritmos evolutivos.

Tabela 12 – Geração e Utilidade do melhor caso (Experimento 3)

Algoritmo	Geração	Utilidade
NSGA-II	23	6,5
SPEA2	28	7,5
PAES	01 a 30	4,5
MOCell	30	6,5

Fonte – Elaborado pelo autor (2017).

Podemos destacar também ainda, o valor dos melhores *fitness* se mantém iguais ao longo das gerações na maioria dos algoritmos testados antes de encontrar uma nova melhor solução. Por exemplo, no caso do algoritmo MOCell, o segundo melhor valor alcançado na 4ª geração, é mantido nas gerações seguintes até a 29ª geração quando há uma melhora no valor da função, ou seja, o melhor indivíduo das gerações seguintes a partir a 4ª também atingiu o mesmo valor de utilidade. Isso está relacionado a observabilidade do agente testado, que mantém resultados semelhantes entre os indivíduos, ocasionando assim, pouca variação durante a geração de novos casos de testes para induzir a diversidade das populações.

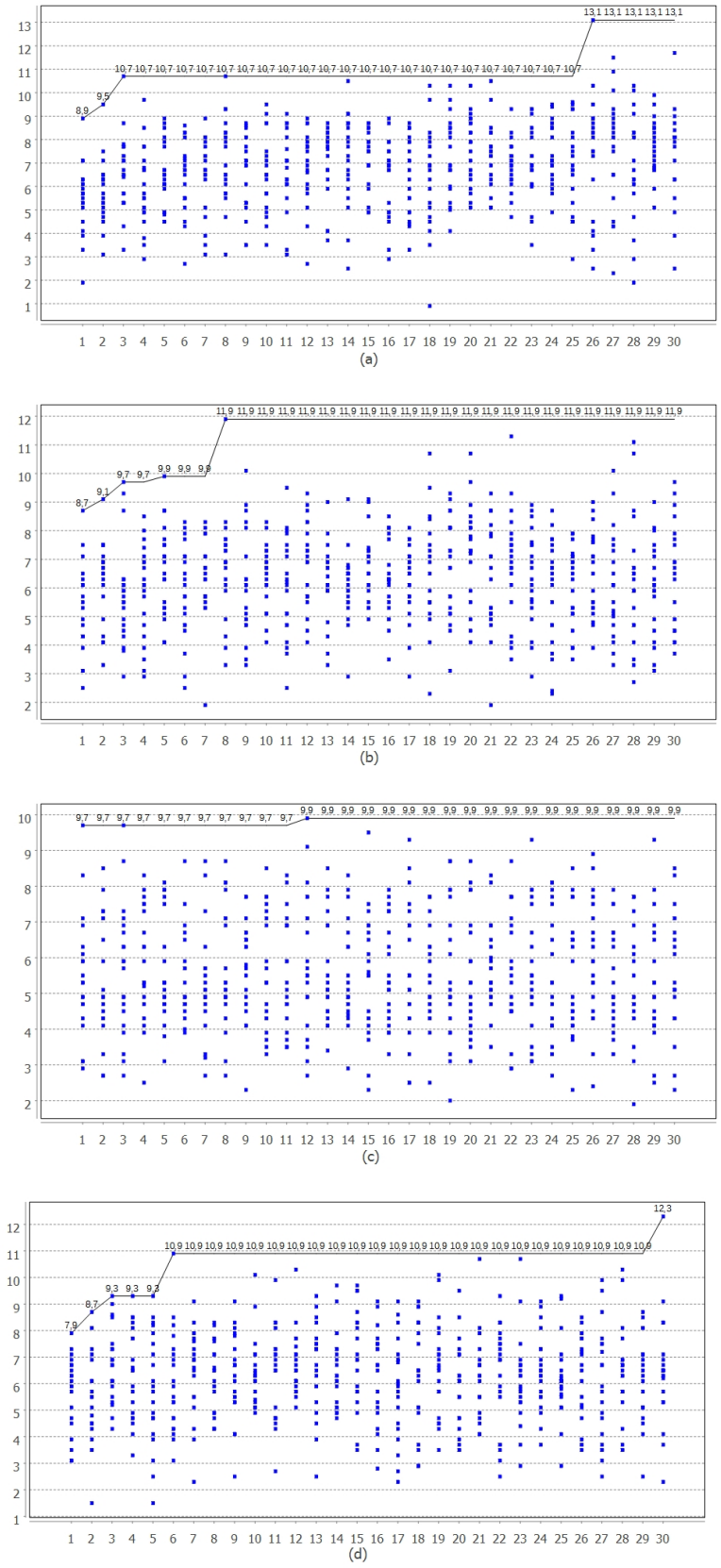
Temos ao final do terceiro experimento, o algoritmo SPEA2 com sendo o melhor algoritmo em gerar casos de teste para esse tipo de agente em comparação com os demais algoritmos. O algoritmo NSGA-II foi o segundo algoritmo que obteve o melhor resultado. Porém, deve ser ressaltado, que o algoritmo conseguiu gerar casos de testes melhores que o anterior durante quatro oportunidades e iniciou com sua geração com o valor de utilidade abaixo do algoritmo SPEA2.

5.2.4 Experimento 4 - Resultados com RM-Parcial

A Figura 29, apresenta os experimentos do agente *RM-Parcial* e a Figura 30 ilustra a variação do valor de utilidade dos quatro algoritmos.

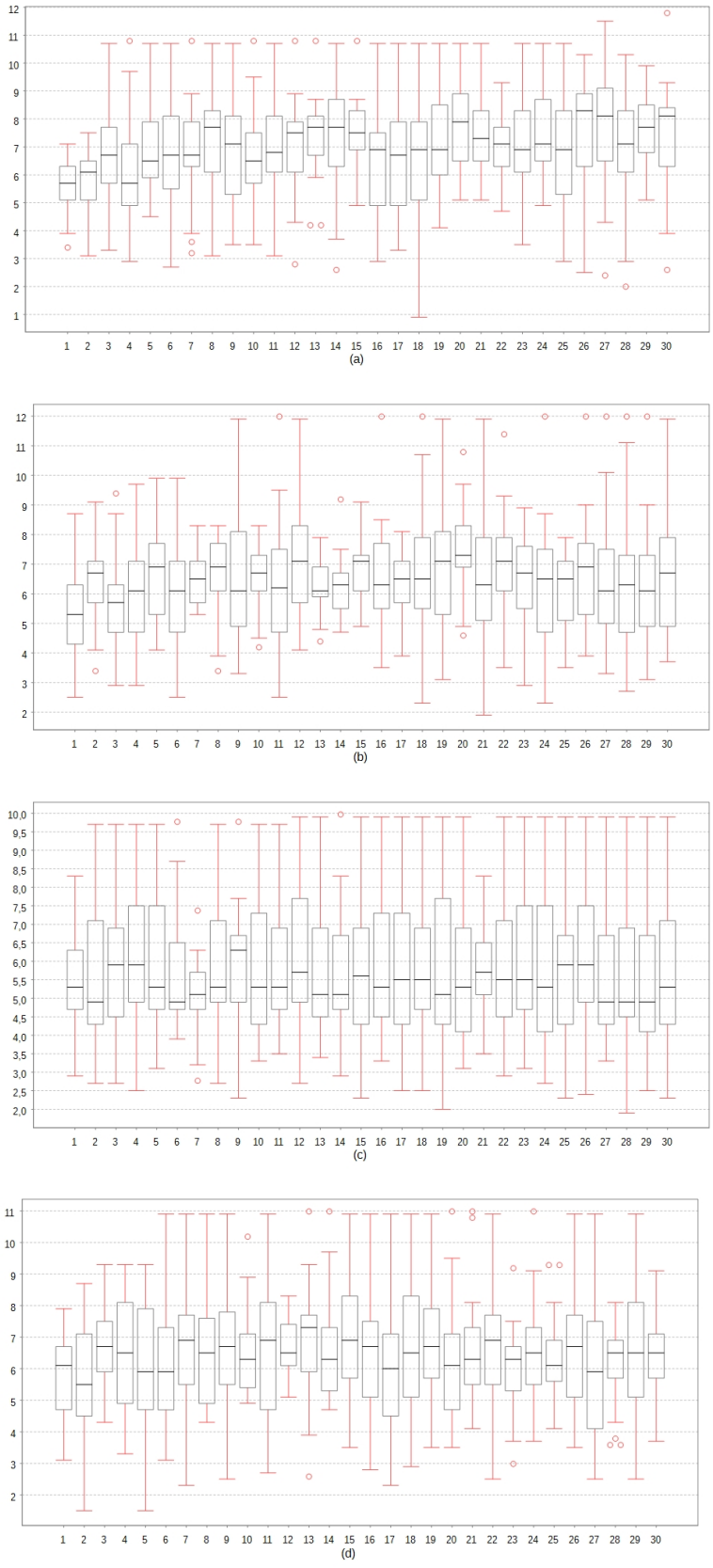
Considerando os valores de utilidade dos melhores casos de teste, o agente mais inadequado para o ambiente parcialmente observável é o reativo simples (experimento 1 e 2),

Figura 29 – Utilidade de casos de testes em 30 gerações (Experimento 4).



Fonte – Elaborado pelo autor (2017).

Figura 30 – Box-plot dos valores de utilidade em 30 gerações (Experimento 4).



Fonte – Elaborado pelo autor (2017).

quando comparado com o agente baseado em modelos. Neste caso, a anotação das salas que foram visitadas pelo subsistema de atualização de estado interno (próximo) do agente baseado em modelos possibilitou a concepção de um subsistema de tomada de decisão (ação) mais refinado, evitando que o agente retornasse desnecessariamente às salas que já foram visitadas em iterações anteriores.

A geração e o melhor valor de utilidade encontrado pelos quatro algoritmos com agente reativo baseado em modelo em um ambiente parcialmente observável é mostrado na Tabela 13.

Tabela 13 – Geração e Utilidade do melhor caso (Experimento 4)

Algoritmo	Geração	Utilidade
NSGA-II	26	13,1
SPEA2	8	11,9
PAES	12	9,9
MOCell	30	12,3

Fonte – Elaborado pelo autor (2017).

Na utilização do algoritmo NSGA-II (Figura 29(a)), o melhor valor da função *Utilidade* é obtido na 26ª geração com utilidade = 13,1. Na utilização do algoritmo SPEA2 (Figura 29(b)), o melhor valor da função é obtido na 8ª geração com utilidade = 11,9. Na utilização do PAES (Figura 29(c)), o melhor valor é obtido na 12ª geração com utilidade = 9,9. Para o algoritmo MOCell, o melhor valor é obtido na 30ª geração com utilidade = 12,3 (Figura 29(d)). Nesse experimento, o NSGA-II foi o algoritmo que obteve o melhor resultado, seguido pelos algoritmos MOCell e SPEA2 e com o pior desempenho, o algoritmo PAES.

5.2.5 Conclusão dos resultados

Nesta seção foi apresentado e discutido os resultados obtidos na avaliação dos experimentos. Começamos apresentando e discutindo na Seção 5.2.1 os resultados do Agente reativo simples (*RS-Parcial*), que executa ações baseados na sua percepção atual. A Seção 5.2.2 apresentou os resultados do agente que executa ações aleatórios (*RS-Parcial-alterado*). A Seção 5.2.3 apresentou os resultados do Agente reativo simples com observabilidade total do ambiente (*RS-Parcial*) e por último, Seção 5.2.4 apresentou os resultados do Agente reativo baseado em modelo (*RM-Parcial*), que seleciona ações com base no seu histórico de percepções.

Conclui-se que, com os resultados obtidos por meio dos quatro experimentos para

cada algoritmo, a viabilidade da utilização da estratégia NSGA-II que identificou, em média, maiores situações de casos de teste em que o agente obteve um maior valor da função *Utilidade*, ou seja, foi pior avaliado. Em segundo lugar ficou o algoritmo SPEA2 que conseguiu atingir um desempenho acima das demais técnicas implementadas, o que a credenciou como uma boa técnica de otimização para o problema estudado juntamente com NSGA-II. O algoritmo evolutivo MOCell obteve o terceiro melhor desempenho e o algoritmo PAES foi o algoritmo com o pior desempenho que obteve em média, o menor valor da função, se mostrando uma opção ineficiente na geração de casos de testes para essa problemática.

6 CONCLUSÕES

Com os softwares cada vez mais complexos, a Inteligência Artificiar (IA) surgiu como um ramo promissor na aplicação de técnicas inteligentes na solução de problemas. Essas técnicas podem ser utilizadas para solucionar desde problemas relativamente simples ou complexos.

Visto que um agente racional é uma entidade autônoma, a realização de testes para esses agentes tem se tornou um grande desafio, devido a esses programas com conhecimento e objetivos mutáveis apresentarem resultados diferentes em uma mesma entrada de teste ao longo do tempo. Dessa forma, esta pesquisa apresentou a aplicação de um agente testador (*Thestes*), orientado por uma função utilidade e pelas estratégias de busca local dos algoritmos evolucionários multiobjetivos (MOEAs) baseadas em populações para encontrar conjuntos de casos de teste satisfatórios para o problema de otimização de seleção de casos de teste.

A abordagem selecionou um conjunto de casos de teste satisfatório em termos dos resultados gerados sobre o desempenho irregular do agente, principalmente com a utilização dos algoritmos NSGA-II e SPA2. Com base nesses resultados, a abordagem possibilita ao projetista informações que possam realizar mudanças objetivas na estrutura interna do agente de forma a melhorar seu desempenho, ressaltando a importância dessa pesquisa, tanto na área de Inteligência Artificial (IA) como de Engenharia de Software (ES).

No entanto, acredita-se que a aplicabilidade da abordagem utilizada possa ser melhor avaliada por meio do emprego de um agente com maior complexidade e cujas ações realizadas sejam baseadas na racionalidade em vez da aleatoriedade.

O aperfeiçoamento e a continuidade deste trabalho, constituem-se em oportunidades para trabalhos futuros que incluem além do emprego de agentes com maiores complexidades, uma proposta de um agente testador que usa aprendizagem de máquina para resolução do problema de seleção de casos de testes. Para isso, o agente testador poderá utilizar algoritmos de aprendizagem de máquina para aprender o desempenho dos agentes testados a partir de um conjunto de atributos e propriedades obtidas das características do ambiente e do problema que o agente busca solucionar. Essa proposta é viável devido aos algoritmos de Aprendizagem de Máquina (AM), uma técnica associada à Inteligência Artificial, utilizarem um princípio de inferência denominado indução, no qual se obtém conclusões genéricas a partir de um conjunto particular de exemplos. Assim, os algoritmos aprendem a induzir novas soluções capazes de resolver problemas a partir de dados que representam instâncias do problema.

REFERÊNCIAS

BELLIFEMINE, F.; POGGI, A.; RIMASSA, G. Developing multi-agent systems with jade. In: SPRINGER. **International Workshop on Agent Theories, Architectures, and Languages**. [S.l.], 2000. p. 89–103.

BIRDSEY, L. A framework and language for complex adaptive system modeling and simulation. In: **2016 Winter Simulation Conference (WSC)**. [S.l.: s.n.], 2016. p. 3670–3671.

BROCKHOFF, D. Evolutionary multiobjective optimization. In: **Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation**. New York, NY, USA: ACM, 2011. (GECCO '11), p. 1111–1136. ISBN 978-1-4503-0690-4. Disponível em: <http://doi.acm.org/10.1145/2001858.2002129>.

CARNEIRO, S. M.; SILVA, T. A. R. da; RABêLO, R. d. A. L.; SILVEIRA, F. R. V.; CAMPOS, G. A. L. de. Artificial immune systems in intelligent agents test. In: **2015 IEEE Congress on Evolutionary Computation (CEC)**. [S.l.: s.n.], 2015. p. 536–543. ISSN 1089-778X.

COELLO GARY B. LAMONT, D. A. V. V. a. C. A. C. **Evolutionary Algorithms for Solving Multi-Objective Problems: Second Edition**. 2. ed. Springer US, 2007. (Genetic and Evolutionary Computation Series). ISBN 978-0-387-33254-3,978-0-387-36797-2. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=E14DAD9680795E083639D060B9BD8247>.

COPPIN, B. **Inteligência Artificial**. [S.l.]: Rio de Janeiro: LTC, 2013.

COSTA, E.; SIMõES, A. **Inteligência Artificial: Fundamentos e Aplicações**. 3. ed.. ed. [S.l.]: Lisboa: FCA, 2015.

DEB, K.; PRATAP, A.; AGARWAL, S.; MEYARIVAN, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. **IEEE Transactions on Evolutionary Computation**, v. 6, n. 2, p. 182–197, 2002.

DURILLO, J. J.; NEBRO, A. J. jmetal: A java framework for multi-objective optimization. **Advances in Engineering Software**, v. 42, p. 760–771, 2011. ISSN 0965-9978. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0965997811001219>.

FAN, Z.; HU, K.; LI, F.; RONG, Y.; LI, W.; LIN, H. Multi-objective evolutionary algorithms embedded with machine learning - a survey. In: **2016 IEEE Congress on Evolutionary Computation (CEC)**. [S.l.: s.n.], 2016. p. 1262–1266.

HOUHAMDI, Z. Multi-agent systems testing: A survey. **International Journal of Advanced Computer Science and Applications**, p. 2156–5570, 2011.

HOUHAMDI, Z. Test suite generation process for agent testing. **Indian Journal of Computer Science and Engineering (IJCSE**, v. 2, 2011.

KNOWLES, J.; CORNE, D. The pareto archived evolution strategy: A new baseline algorithm for multiobjective optimization. In: **Proceedings of the 1999 Congress on Evolutionary Computation**. Piscataway, NJ: IEEE Press, 1999. p. 9–105.

LUGER, G. F. **Inteligência Artificial**. 6. ed.. ed. [S.l.]: São Paulo: Pearson Education, 2013.

MALIK, M. F.; KHAN, M. N. A. An analysis of performance testing in distributed software applications. v. 7, p. 5360, 2016.

MYLOPOULOS, J.; CASTRO, J. Tropos: A framework for requirements-driven software development. **Information Systems Engineering: State of the Art and Research Themes, Lecture Notes in Computer Science, Springer**, 2000.

NEBRO, A. J.; DURILLO, J. J.; LUNA, F.; DORRONSORO, B.; ALBA, E. Design issues in a multiobjective cellular genetic algorithm. In: OBAYASHI, S.; DEB, K.; POLONI, C.; HIROYASU, T.; MURATA, T. (Ed.). **Evolutionary Multi-Criterion Optimization. 4th International Conference, EMO 2007**. [S.l.]: Springer, 2007. (Lecture Notes in Computer Science, v. 4403), p. 126–140.

NGUYEN, C. D.; MILES, S.; PERINI, A.; TONELLA, P.; HARMAN, M.; LUCK, M. Evolutionary testing of autonomous software agents. **Autonomous Agents and Multi-Agent Systems**, v. 25, n. 2, p. 260–283, 2012. ISSN 1573-7454. Disponível em: <http://dx.doi.org/10.1007/s10458-011-9175-4>.

NGUYEN, C. D.; PERINI, A.; TONELLA, P. Goal-oriented testing for mass. **Int. J. Agent-Oriented Softw. Eng.**, Inderscience Publishers, Inderscience Publishers, Geneva, SWITZERLAND, v. 4, n. 1, p. 79–109, dez. 2010. ISSN 1746-1375. Disponível em: <http://dx.doi.org/10.1504/IJAOSE.2010.029810>.

PADGHAM, L.; ZHANG, Z.; THANGARAJAH, J.; MILLER, T. Model-based test oracle generation for automated unit testing of agent systems. **IEEE Transactions on Software Engineering**, v. 39, n. 9, p. 1230–1244, Sept 2013. ISSN 0098-5589.

PONSICH, A.; JAIMES, A. L.; COELLO, C. A. C. Asurvey on multiobjective evolutionary algorithms for the solution of the portfolio optimization problem and other finance and economics applications. **IEEE Transactions on Evolutionary Computation**, v. 17, n. 3, p. 321–344, June 2013. ISSN 1089-778X.

PRESSMAN, R. S. **Engenharia de Software: uma abordagem profissional**. 6. ed.. ed. [S.l.]: Porto Alegre: AMGH, 2011.

ROUFF, C. A test agent for testing agents and their communities. In: **Aerospace Conference Proceedings, 2002. IEEE**. [S.l.: s.n.], 2002. v. 5, p. 5–2638 vol.5.

RUSSELL, S.; NORVIG, P. **Inteligência Artificial**. 3. ed.. ed. [S.l.]: Rio de Janeiro: Elsevierl, 2013.

SILVEIRA, F. R. de V.; CAMPOS, G. A. L. de; CORTéS, M. I. A problem-solving agent to test rational agents - a case study with reactive agents. In: **Proceedings of the 16th International Conference on Enterprise Information Systems**. [S.l.: s.n.], 2014. p. 505–513. ISBN 978-989-758-027-7.

SILVEIRA, F. R. V. **Uma abordagem fundamentada em agentes racionais para o teste de agentes racionais.** Dissertação (Mestrado) — Universidade Estadual do Ceará - UECE. Centro de Ciências e Tecnologia - CCT. Mestrado Acadêmico em Ciência da Computação., Fortaleza, 2013.

SILVEIRA, F. R. V.; CAMPOS, G. A. L. de; CORTéS, M. Monitoring and diagnosis of faults in tests of rational agents based on condition-action rules. In: **Proceedings of the 17th International Conference on Enterprise Information Systems**. [S.l.: s.n.], 2015. p. 585–592. ISBN 978-989-758-096-3.

ZHANG, Z.; THANGARAJAH, J.; PADGHAM, L. Automated unit testing for agent systems. Citeseer, 2007.

ZITZLER, E.; LAUMANNS, M.; THIELE, L. **SPEA2: Improving the Strength Pareto Evolutionary Algorithm**. [S.l.], 2001.