UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Guilherme Pacheco de Oliveira

Métodos de Inteligência Artificial aplicados em jogos baseados em turnos

Uberlândia, Brasil

2018

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Guilherme Pacheco de Oliveira

Métodos de Inteligência Artificial aplicados em jogos baseados em turnos

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Christiane Regina Soares Brasil

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2018

Guilherme Pacheco de Oliveira

Métodos de Inteligência Artificial aplicados em jogos baseados em turnos

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 09 de Julho de 2018:

Christiane Regina Soares Brasil

Paulo Henrique Ribeiro Gabriel

Renan Gonçalves Cattelan

Uberlândia, Brasil
2018

Resumo

O mercado de jogos eletrônicos vem crescendo rapidamente, junto com ele, também há uma grande evolução no poder de processamento e das tecnologias que envolvem esse ramo. Isso possibilita que, além de questões gráficas, os jogos possam explorar o uso de técnicas da Inteligência Artificial. A melhoria da tecnologia também possibilita que novos algoritmos sejam investigados em diferentes etapas do desenvolvimento de um jogo. Neste trabalho foram desenvolvidos um algoritmo de aprendizado por reforço, o *HAQL*, e um Algoritmo Evolutivo para realizar modelagem de jogadores utilizando cinco agentes que possuem objetivos distintos dentro de um jogo baseado em turnos chamado *MiniDungeons*. As técnicas utilizadas são úteis para serem aplicadas durante o desenvolvimento de jogos, simulando comportamentos de jogadores reais e fazendo com que partes do conteúdo de um jogo possam ser testadas de modo mais rápido e barato. Os resultados obtidos a partir dos experimentos realizados indicam que o algoritmo de aprendizado por reforço foi eficiente em otimizar um objetivo isolado, confirmando resultados mostrados na literatura, enquanto o AE apresentou um conjunto de soluções mais versáteis, buscando otimizar não somente um objetivo específico, aproximando-o mais à realidade de um jogador humano.

Palavras-chave: Algoritmos Evolutivos, Aprendizado de Máquina, Inteligência Artificial, Aprendizado em Jogos, Modelagem de Jogadores.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AG	Algoritmo Genético
AE	Algoritmo Evolutivo
IA	Inteligência Artificial
RPG	*Role Playing Game*
MDP	*Markov Decision Process*
HAQL	*Heuristically Accelerated Q-Learning*

Sumário

1 Introdução

1.1 Visão Geral

A indústria de jogos vem crescendo significativamente desde a sua criação, por volta de 1971. Pode-se perceber que 155 milhões de americanos jogavam regularmente em 2014 e a indústria chegou a movimentar US$22.41 bilhões (*Entertainment Software Association*, 2015).

No passado, o maior desejo nos jogos era de atingir imagens cada vez mais realistas, no entanto, a evolução desse quesito logo já não era mais prioridade (FAIRCLOUGH; et al., 2001) (SWEETSER, 2002). A diferença entre a qualidade gráfica de jogos lançados em um espaço de tempo curto passou a ser menos perceptível, isso fez com que outros pontos fossem explorados como diferencial de vendas, um desses pontos é a Inteligência Artificial (IA) (SWEETSER, 2002). Além desse ponto, a evolução da tecnologia permitiu maior poder de processamento e unidades de processamento dedicadas a gráficos, fazendo com que ainda mais recursos pudessem ser utilizados em técnicas de IA cada vez mais complexas.

No início dos anos 2000, as técnicas mais comuns de Inteligência Artificial em jogos eram máquinas de estados finitos, *scripting*, agentes inteligentes, *flocking*, entre outras (SWEETSER, 2002). Devido aos acelerados avanços tecnológicos, novas técnicas estão sendo utilizadas, sendo as principais atualmente a computação evolutiva, busca em árvores e *planning* (YANNAKAKIS; TOGELIUS, 2014).

Dentro da Computação Evolutiva, estão os Algoritmos Genéticos (AG), que são uma família de algoritmos inspirados na Teoria da Evolução (GOLDBERG, 1989)(HOLLAND, 1992)(DARWIN, 1859). Os AGs geram um conjunto de potenciais soluções de um problema, onde esse conjunto é denominado população e cada possível solução é um indivíduo. Cada indivíduo é representado por uma estrutura correspondente a um cromossomo, e sobre estes indivíduos são aplicados operadores de reprodução a fim de buscar por novas soluções, guiados por uma função de avaliação.

No contexto de jogos, um AG pode ser utilizado para se adaptar ao comportamento do jogador e explorar suas fraquezas; bem como para definir estratégias e evoluir comportamento de personagens controlados pela máquina. Conforme investigado no trabalho de Crocomo (CROCOMO, 2008), havia a impressão de que esse método era lento e ineficiente, porém com os avanços tecnológicos essa técnica se mostrou robusta e capaz de ser executada em tempo real conforme demonstrado nesse trabalho. Por esses atributos, atualmente a computação evolutiva é um dos principais métodos utilizados em pesquisas em jogos (YANNAKAKIS; TOGELIUS, 2014).

Uma área de pesquisa que vem ganhando atenção é a Modelagem de Jogadores, que possui quatro objetivos: descrever, predizer, interpretar e, em alguns casos, representar o comportamento de jogadores, sendo que cada objetivo é abordado individualmente (YANNAKAKIS et al., 2013).

Este trabalho está inserido na área de representação de comportamento de jogadores e leva em consideração a afirmação de que modelos de jogadores possuem características que os permitem serem utilizados como abstrações do processo de tomada de decisão humana. Os agentes resultantes desses modelos podem ser utilizados como ferramentas de teste no processo de *design* de jogos (HOLMGÅRD et al., 2014b).

Além disso, este trabalho visa expandir a linha de investigação feita por Holmgård, utilizando uma outra técnica ainda não explorada por ele. Tais trabalhos serão melhores abordados na Seção 3.5 (Trabalhos Relacionados).

O trabalho *Generative Agents for Player Decision Modeling in Games* (2014b) utiliza de um algoritmo de aprendizado de máquina, o *Q-Learning* para fazer a representação de comportamentos de jogadores. Em sua seção de Discussões é argumentado que uma possível abordagem futura seria utilizando um sistema evolutivo baseado em regras.

Deste modo, este Trabalho de Conclusão de Curso propõe um Algoritmo Evolutivo para implementar uma abordagem evolutiva baseada em regras, inspirada na dissertação de Crocomo (2008), para representar modelos de jogadores previamente definidos no jogo *MiniDungeons* (HOLMGÅRD; LIAPIS; TOGELIUS,

2014). A eficácia dessa abordagem é analisada por meio da comparação com o método de Aprendizado por Reforço *HAQL*, que é uma alternativa mais eficiente a um outro algoritmo de aprendizado, o *Q-Learning*.

1.2 Motivação

A principal motivação para representar modelos de decisão de jogadores é a possibilidade de realizar uma avaliação prévia de conteúdos gerados para jogos, poupando tempo e custos que seriam utilizados em testes com pessoas reais.

A utilização de Algoritmos Evolutivos é motivada pela possibilidade de ser uma abordagem melhor que o *Q-Learning* nesse problema, conforme citado na Seção de Trabalhos Futuros da literatura (HOLMGÅRD et al., 2014a), visando confirmar a ideia de que a computação evolutiva é um método de grande utilidade no contexto de desenvolvimento de jogos.

Finalmente, o trabalho também é motivado pela necessidade de expandir os trabalhos de referência, aplicando mais uma técnica ao problema e realizando uma análise comparativa com seus pontos positivos e negativos.

1.3 Objetivos

O objetivo geral deste Trabalho de Conclusão de Curso é explorar métodos de inteligência artificial e aprendizado de máquina, mais especificamente: Algoritmos Evolutivos e *Q-Learning*, na representação de modelos de decisão de jogadores em jogos baseados em turnos. O jogo utilizado para testes foi *Mini-Dungeons*, por ser um jogo já utilizado na literatura para realizar modelagem de comportamento de jogadores (HOLMGÅRD et al., 2014b) (HOLMGÅRD et al., 2014a) (HOLMGÅRD et al., 2015).

Para tanto, os objetivos específicos do trabalho foram:

1. Implementar um algoritmo evolutivo capaz de evoluir conjuntos de regras.

2. Implementar um controlador no jogo *MiniDungeons* que possa utilizar conjuntos de regras para jogar.
3. Implementar o algoritmo de aprendizado de máquina *Q-Learning* e sua versão mais eficiente, *HAQL*.
4. Implementar um controlador capaz de jogar *MiniDungeons* utilizando *Q-Learning*.
5. Implementar um controlador capaz de jogar *MiniDungeons* utilizando *HAQL*.
6. Comparar o resultado dos três algoritmos.
7. Comparar a abordagem com Algoritmos Evolutivos com outros métodos encontrados na literatura.

1.4 Organização do Trabalho

O trabalho está organizado da seguinte forma: o Capítulo 2 aborda a fundamentação teórica acerca das técnicas utilizadas no trabalho. O Capítulo 3 apresenta aspectos de jogos e a utilização de algoritmos de IA nos mesmos, bem como trabalhos relacionados. O Capítulo 4 fala sobre o desenvolvimento dos algoritmos utilizados. O Capítulo 5 apresenta os resultados obtidos, comparação entre os algoritmos e com a literatura. Por fim, o Capítulo 6 relata as conclusões obtidas pelos experimentos realizados e aborda possíveis abordagens que podem ser exploradas no futuro.

2 Metodologia

2.1 Algoritmos Evolutivos

Os algoritmos evolutivos (AEs) (GOLDBERG, 1989)(HOLLAND, 1992) são meta-heurísticas bioinspiradas criadas por Holland e popularizadas por Goldberg, as quais foram baseadas na Teoria da Evolução (DARWIN, 1859). Segundo essa teoria, uma população é capaz de evoluir utilizando mecanismos biológicos, como mutação, recombinação, seleção natural e sobrevivência dos indivíduos mais adaptados. Computacionalmente, sua implementação é simples, de fácil adaptação e possibilita encontrar soluções para problemas complexos.

Existem várias abordagens dos algoritmos evolutivos, como a programação evolutiva (PE), as estratégias evolutivos (EEs) e os algoritmos genéticos (AGs) (GABRIEL; DELBEM, 2008)

Programação Evolutiva: Na programação evolutiva, cada indivíduo é representado por uma Máquina de Estados Finitos [1], que processa uma sequência de símbolos. A reprodução é feita somente utilizando de mutação, caracterizando uma reprodução assexuada.

Estratégias Evolutivas: As estratégias evolutivas foram desenvolvidas com o objetivo de solucionar problemas de otimização de parâmetros. Nessa abordagem, um pai gera apenas um descendente e ambos competem entre si pela sobrevivência. A representação dos dados é feita em ponto flutuante.

Algoritmos Genéticos: Os algoritmos genéticos, também chamados de algoritmos evolutivos, têm como principal diferença a utilização do operador de recombinação, além da mutação, podendo também apresentar métodos de seleção para definir quais indivíduos irão gerar descendentes.

Em muitas referências pode-se encontrar os termos Algoritmo Genético e

[1] Também chamados de Autômatos Finitos, são modelos matemáticos capazes de reconhecer linguagens (MENEZES, 2000).

Algoritmos Evolutivos como sinônimos.

O funcionamento geral de um algoritmo evolutivo é ilustrado na Figura 1.

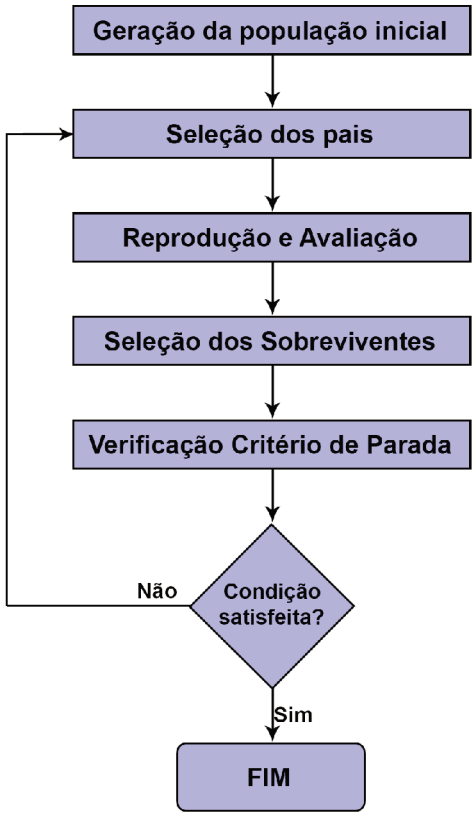


Figura 1 – Fluxograma de um AE típico. Adaptado de: (GABRIEL; DELBEM, 2008).

Para um melhor entendimento do funcionamento de um algoritmo evolutivo, alguns termos específicos precisam ser explicados. As soluções são codificadas em estruturas chamadas de cromossomo, que se associam de modo a formar uma solução. Os cromossomos são codificados em um conjunto de símbolos chamados genes, que podem possuir diferentes valores, chamados de alelos.

Esses métodos possuem uma função objetivo (ou de aptidão), que deve ser minimizada ou maximizada, dependendo do problema. Primeiramente, é gerado

um conjunto aleatório de soluções, que corresponde à população inicial, e aplica-se a função objetivo em cada solução, a fim de medir quão boa é, atribuindo-lhes um valor chamado *fitness*. A função objetivo é definida para cada problema especificamente.

Com base no *fitness*, as melhores soluções são escolhidas para a próxima geração. Por meio de operadores de reprodução (recombinação e/ou mutação) novas soluções são geradas e competem com as antigas. Ao final de cada geração, as soluções com os melhores *fitness* permanecem na população, ocasionando uma evolução. Esse processo se repete até que a condição de parada seja satisfeita.

O operador de recombinação (também conhecido como cruzamento ou *crossover*) é o mecanismo de obtenção de novos indivíduos pela troca dos alelos de dois ou mais indivíduos (GABRIEL; DELBEM, 2008). O resultado da operação é um indivíduo que combina as características de dois ou mais pais.

O tipo de recombinação mais simples é a recombinação de um ponto. Nesta, seleciona-se um ponto aleatório igual para os dois pais, então são gerados dois filhos, cada um com a parte à esquerda de um pai e à direita de outro pai. Esse tipo de recombinação é ilustrado na Figura 2 (a).

A ideia da recombinação de um ponto pode ser generalizada para recombinação de n pontos, sorteando n cortes aleatórios nos pais e criando "seções" nos indivíduos, onde os descendentes vão herdar seções de pais diferentes. Esse tipo de recombinação é ilustrado na Figura 2 (b).

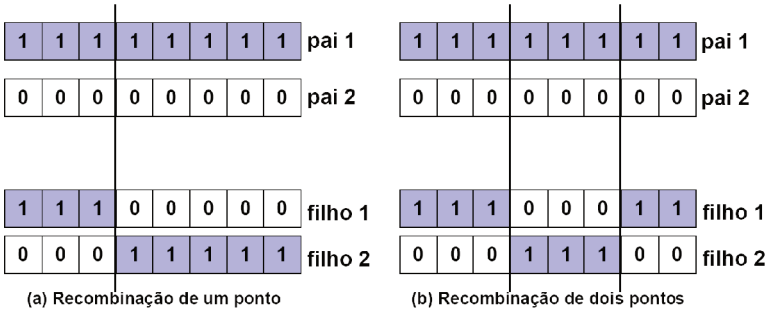


Figura 2 – Ilustração de operadores de recombinação. Adaptado de: (GABRIEL; DELBEM, 2008).

O operador de mutação modifica aleatoriamente um ou mais genes de um cromossomo. Essa operação acontece com uma probabilidade chamada de taxa de mutação.

Um exemplo de operador de mutação pode ser selecionar um gene aleatoriamente no indivíduo e trocá-lo por outro gene. Usando uma codificação binária como exemplo, caso um gene seja "0", ele passa a ser "1". Isso é ilustrado na Figura 3.

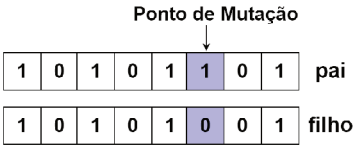


Figura 3 – Ilustração de um operador de mutação. Adaptado de: (GABRIEL; DELBEM, 2008).

A seguir, serão descritos os principais conceitos de Aprendizado de Máquina e apresentado o algoritmo *Q-Learning*.

2.2 Aprendizado de Máquina

A Inteligência Artificial é uma área de pesquisa considerada entre as mais recentes, surgida na década de 1950 (RUSSEL; PETER, 2010). Essa área possui quatro principais abordagens que a definem:

1. criar sistemas que são capazes de pensar de forma humana, fazendo computares aprender, tomar decisões e resolver problemas;
2. criar sistemas que pensam racionalmente, que podem perceber o ambiente, processar a informação de modo racional e, então, agir;
3. criar sistemas que agem de forma humana, que são capazes de realizar tarefas como seres humanos as realizam;

4. criar sistemas que agem de forma racional, ou seja, podem manifestar comportamentos racionais.

Historicamente, as quatro abordagens de IA foram estudadas por diferentes pessoas e métodos. É uma área que permanece em ascensão até os dias atuais, tendo em vista que pode ser aplicada em uma grande variedade de tarefas, desde algumas mais específicas, como jogar xadrez, provar teoremas matemáticos e dirigir um carro, por exemplo, a algumas mais amplas, como percepção e aprendizado de máquina.

O Aprendizado de Máquina é uma subárea da Inteligência Artificial que consiste em viabilizar a computadores a capacidade de aprender a relizar uma tarefa sem que seja explicitamente programado (SAMUEL, 1959). Nessa área, deseja-se utilizar dados de experiências passadas para otimizar o desempenho em problemas no presente (ALPAYDIN, 2010), ou seja, são desenvolvidos algoritmos que possibilitem a máquina a aprender (SILVA, 2008). O aprendizado pode ser classicado em Aprendizado Supervisionado e Não-Supervisionado.

2.2.1 Aprendizado Supervisionado

O Aprendizado Supervisionado consiste na inserção de um "Supervisor" como intermediador, que determina se as predições realizadas estão corretas ou não durante o processo de aprendizagem. Essa abordagem é bastante útil em aplicações que envolvem classificação e regressão, onde já se tem a informação prévia do resultado de alguns elementos do problema. São utilizadas no treinamento de Redes Neurais Artificias [2] e de Árvores de Decisão [3] (SILVA, 2008) (REZENDE, 2003).

2.2.2 Aprendizado Não-Supervisionado

Ao contrário do Aprendizado Supervisionado, o Não-Supervisionado não possui nenhum tipo de orientação para verificar se a sua predição está correta

[2] Construção de unidades de processamento semelhantes aos neurônios humanos (SILVA, 2008).
[3] Uma estrutura hierárquica que pode ser utilizada para classificação e regressão (ALPAYDIN, 2010).

ou não. Diante disso, uma das formas de realizar o aprendizado é criar um sistema onde o agente é premiado para certas ações (SILVA, 2008). Esse paradigma de aprendizado também pode ser chamado de Aprendizado por Reforço (SUTTON; BARTO, 2012), (SILVA, 2008). Entre os algoritmos de aprendizado não-supervisionado, podemos citar o *Q-Learning* (HOLMGÅRD et al., 2014b)(WATKINS; DAYAN, 1992) e o *Dynamic Scripting* (SPRONCK; SPRINKHUIZEN-KUYPER; POSTMA, 2004). Na literatura é muito comum a utilização do termo Aprendizado por Reforço, portanto, esse será o termo adotado neste trabalho.

O Aprendizado por Reforço consiste em aprender qual ação tomar para que se possa maximizar uma recompensa a longo prazo. (SUTTON; BARTO, 2012). As duas características mais importantes desse método são: a busca por tentativa e erro e a recompensa.

Um exemplo de aplicação desse tipo de aprendizagem seria um robô móvel que coleta lixo. A decisão de entrar em um novo cômodo para procurar lixo ou voltar para recarregar sua bateria vai depender de suas experiências passadas.

O aprendizado por reforço busca resolver um problema em que é necessário se atingir um objetivo (SUTTON; BARTO, 2012). Para tal, toma-se uma ação e é verificado se o resultado dessa ação foi positivo ou negativo. O problema é composto por duas partes: um *agente*, que irá aprender e tomar decisões, que interage com o *ambiente*, que irá responder as ações do agente e apresentar novas situações.

Existem quatro elementos principais nesse método de aprendizado (SUTTON; BARTO, 2012):

- uma política que define como um agente deve se comportar em um determinado instante;
- uma função de recompensa que define o objetivo a se atingir no ambiente em um instante, utilizada para tomar uma decisão no curto prazo;
- uma função de valor que especifica uma recompensa que o agente pode acumular a partir de um estado, estimando o quão bom é estar naquele estado, sendo melhor a longo prazo, e

- um modelo do ambiente, que é uma representação do ambiente em que o agente está inserido e quais ações ele pode tomar.

Considerando o processo de aprendizado, no aprendizado por reforço surge um problema interno que é a decisão entre exploração e aproveitamento. Um agente tem que *explorar* o ambiente para que, no futuro, ele possa fazer melhores decisões, mas ele também deve *aproveitar* seu conhecimento para obter as melhores recompensas.

2.2.3 Definições Matemáticas

Os principais conceitos que fundamentam o Aprendizado por Reforço possuem forte embasamento matemático. As principais definições serão discutidas a seguir.

Primeiramente, deve ser definido um método para avaliar o valor das ações, a fim de decidir qual ação realizar em um determinado momento. Uma forma de estimar quão boa é uma ação consiste em calcular a média das recompensas geradas por aquela ação (SUTTON; BARTO, 2012), seguindo a forma:

$$Q_t(a) = \frac{R_1 + R2 + ... + R_{K_a}}{K_a} \tag{2.1}$$

onde $R_1 + R2 + ... + R_{K_a}$ é a soma das recompensas obtidas por aquela ação e K_a é a quantidade de vezes que essa ação foi escolhida. Se $K_a = 0$ então um valor pré definido para Q_t é utilizado, como por exemplo, 0.

Com a função ação-valor definida, são determinados alguns métodos para selecionar as ações que serão tomadas em determinado momento do problema. As abordagens mais comuns são: *Greedy*, ϵ-*greedy* e *Softmax* (BARBOSA, 2013)(SUTTON; BARTO, 2012). O método *Greedy* sempre opta pela ação que promove a maior recompensa, o ϵ-*greedy* escolhe uma ação de aproveitamento com probabilidade 1 - ϵ, onde ϵ é uma pequena probabilidade de optar por uma ação aleatória para explorar o ambiente. No *Softmax*, a ação gulosa (*greedy*) ainda acontece com maior probabilidade, mas as outras ações possíveis são escolhidas baseadas em es-

timativas de distribuição, como por exemplo distribuição de Gibbs ou Boltzman (BARBOSA, 2013).

Um agente e um ambiente interagem entre si em uma sequência de intervalos discretos $t = 0, 1, 2, 3, ...$(SUTTON; BARTO, 2012). Em cada intervalo t o agente recebe uma representação do ambiente, chamada de *estado*, $S_t \in S$, onde S é o conjunto de todos estados possíveis. Baseado nisso, é escolhida uma ação $A_t \in A(S_t)$, onde $A(S_t)$ é o conjunto das ações disponíveis no estado S_t.

A cada instante, o agente possui uma função que mapeia os estados nas probabilidades de escolhea de cada ação possível. Esse mapeamento caracteriza a política do agente e é denotada por π_t

O objetivo do agente é maximizar a recompensa no longo prazo. Isso pode ser formalmente definido propondo uma função que descreva o retorno esperado G_t, da seguinte forma:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.2}$$

onde T é a última ação a ser tomada para solucionar o problema e $R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T$ é a soma das recompensas recebidas depois de um instante t.

Ainda deve ser introduzido o conceito de desconto. A taxa de desconto, definida como γ, que está entre $0 \leq \gamma \leq 1$, é uma taxa que determina o valor atual de recompensas futuras e a alteração de seu valor pode fazer o agente tomar decisões visando mais o curto prazo ou o longo prazo. Com esse conceito, a função do retorno esperado passa a ter a forma:

$$G_t = R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + ... + \gamma^n R_T = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.3}$$

Propriedade *Markoviana*

No problema de aprendizado por reforço, o agente toma uma decisão baseada no estado do ambiente (SUTTON; BARTO, 2012), sendo o estado toda a informação disponível para o agente.

Um estado é dito *Markoviano* ou possui a propriedade *Markoviana* se um estado contém todas as informações relevantes do problema, ou seja, dado um estado com essa propriedade, é possível predizer o próximo estado e a recompensa esperada nele, considerando apenas o estado atual.

A propriedade *Markoviana* é importante para o aprendizado por reforço pois as decisões e valores são assumidos como uma função que considera apenas o estado atual do ambiente.

Se uma tarefa de aprendizado satisfaz essa propriedade, ela é chamada de Processo de Decisão Markoviano(MDP, do inglês *Markov Decision Process*) e se essa tarefa possuir finitos estados e ações, é um Processo de Decisão Markoviano Finito (MDP finito)

Resolvendo Tarefas de Aprendizado

Utilizando o conceito de MDP finito, podem ser definidas políticas ótimas que vão maximizar a recompensa no longo prazo. Essas políticas possuem em comum uma função ótima de estado valor, denotada por v_* e uma função ótima de ação-valor, denotada por q_*, que possuem a forma:

$$v_*(s) = max v_\pi(s) \tag{2.4}$$

onde $s \in S$ e $v_\pi(s)$ é o valor de um estado s na política π e

$$q_*(s,a) = max q_\pi(s,a) \tag{2.5}$$

onde $s \in S$, $a \in A(s)$ e $q_\pi(s,a)$ é o valor de uma ação a em um estado s.

2.2.4 Algoritmos de Aprendizado

Problemas de aprendizado simples podem ter sua função de estado-valor representada por uma lista ou uma tabela, pois os estados e as ações possíveis são suficientemente pequenos e a solução exata pode ser encontrada facilmente. Problemas reais geralmente apresentam espaços de busca maiores e, deste modo,

encontrar a solução exata é difícil e exige a utilização de uma técnica de aprendizado (SUTTON; BARTO, 2012).

Existem três classes de técnicas de aprendizado: *Dynamic Programming* (Programação Dinâmica), métodos de Monte Carlo e *Temporal-Difference Learning* (em português: Aprendizado de Diferença Temporal (BARBOSA, 2013)), categoria onde o *Q-Learning* está inserido.

O Aprendizado de Diferença Temporal é o resultado de uma combinação entre ideias de Programação Dinâmica e métodos Monte Carlo. Nessa categoria encontra-se o *Q-Learning*, algoritmo que foi responsável por inovar a área de aprendizado por reforço (SUTTON; BARTO, 2012).

2.2.4.1 *Q-Learning*

O *Q-Learning* é um algoritmo *off-policy*, ou seja, é um algoritmo onde o comportamento do agente e a avaliação do ambiente são feitos de modo independente, um não tem influência sobre o outro.

De forma simplificada, o algoritmo *Q-Learning* mantém uma tabela chamada tabela-Q que é atualizada para cada estado-ação com o valor-Q, que é o nível de recompensa de cada ação. Isso acontece para cada instante do aprendizado (BARBOSA, 2013).

O algoritmo foi proposto e teve a sua convergência comprovada matematicamente por Christopher Watkins (WATKINS; DAYAN, 1992).

A forma geral da função Q do *Q-Learning* é da forma:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.6}$$

onde:

- Q(S_t, A_t) é o valor atual da recompensa estimada (valor-Q) para o movimento A_t no estado S_t;
- α e γ são, respectivamente, as constantes de fator de aprendizado e desconto;

- R_{t+1} é a compensa do estado $t+1$, onde esse estado é consequência do movimento escolhido no estado atual;
- $maxQ(S_{t+1}, a)$ é o valor máximo da tabela Q levando em consideração o estado $t+1$ e todos os movimentos possíveis de serem realizados naquele estado.

Tendo esse formato, o valor aprendido para a função ação-valor Q se aproxima de q_*, que é a função ação-valor ótima e isso não depende da política que está sendo seguida pelo algoritmo.

O funcionamento procedural do *Q-Learning* pode ser visto a seguir (SUTTON; BARTO, 2012).

Inicializar $Q(s,a), \forall s \in S, a \in A(s)$ aleatoriamente e
$Q(estadoFinal, -) = 0$;
foreach *episódio* **do**
  Inicializar S
  repeat
    Escolher A de S usando política derivada de Q
    Realização ação A, observar R e S'
    $Q(S,A) = Q(S,A) + \alpha[R + \gamma max_a Q(S', a) - Q(S,A)]$
    $S = S'$
  until *S é um estado final*;
end

2.2.4.2 *HAQL*

O *HAQL* (do inglês, *Heuristically Accelerated Q-Learning*) é um método proposto por Reinaldo Bianchi (BIANCHI, 2004) e tem como intuito adicionar heurísticar à busca feita na política do *Q-Learning* para que o algoritmo seja mais rápido e tenha uma convergência mais ágil.

Este método introduz uma função heurística na regra de transição de estados que escolhe uma ação a_t a ser tomada em um estado s_t.

A estratégia considerada é a estratégia $\epsilon - greedy$, onde além da recompensa para um dada ação, também é considerado o valor de uma função H. Sendo assim,

a política de transição de estados para o *HAQL* é dada pela equação:

$$\pi(s_t) = \begin{cases} a_{aleatorio} & \text{se q} \leq \epsilon, \\ max_a[F_t(s_t, a_t)\Theta\xi H_t(s_t, a_t)^\beta] & \text{caso contrário} \end{cases} \tag{2.7}$$

onde:

- $F_t(s_t, a_t)$ é uma função que estima a recompensa acumulada para aquela ação naquele estado, no caso do *Q-Learning*, $F_t(s_t, a_t) = Q_t(s_t, a_t)$.
- $H_t(s_t, a_t)$ é a função heurística que influencia a escolha da ação, define a importância de executar a ação a_t no estado s_t
- Θ é uma função matemática que deve operar sobre os números reais e produzir um valor pertencente a um conjunto ordenado que suporte a operação de maximização.
- ξ e β são variáveis reais utilizadas para ponderar a influência da função heurística.
- q é um valor aleatório escolhido entre 0 e 1 e ϵ é o parâmetro que define a taxa de exploração/aproveitamento. Quanto menor seu valor, menor a probabilidade de se efetuar uma ação aleatória.

e a função heurística é definida como:

$$H_t(s_t, a_t) = \begin{cases} max_a Q(s_t, a_t) - Q(s_t, a_t) + \eta & \text{se } a_t = \pi^H(s_t), \\ 0 & \text{caso contrário} \end{cases} \tag{2.8}$$

Ou seja, a função apenas assumirá algum valor para um par de estado-ação quando o par for escolhido pela política.

3 Jogos

As pesquisas na área de jogos realizaram um dos primeiros grandes objetivos da IA em 1996 (FURNKRANZ, 2004): fazer um programa capaz de ganhar de um campeão mundial de xadrez. Desde então, esse ramo de pesquisa tem se intensificado e novos algoritmos foram propostos para diversos jogos.

Na década de 1990, a Inteligência Artificial tornou-se um ponto forte para a venda de jogos (MILLINGTON; FUNGE, 2009), sendo o diferencial para jogos como *Goldeneye 007* (*Rate Ltd.*, 1997) e *Metal Gear Solid* (*Konami Corporation*, 1998). Foi na mesma época que os jogos de estratégia em tempo real (RTS, do inglês *Real Time Strategy*) começaram a aparecer. *Warcraft* (*Blizzard Entertainment*, 1994) realizava toda a movimentação do jogo utilizando algoritmos de busca de caminho (*pathfinding*).

Atualmente, a Inteligência Artificial cuida de três necessidades básicas dos jogos: mover personagens, tomar decisões e pensar taticamente ou estrategicamente (MILLINGTON; FUNGE, 2009). Além disso, a IA é empregada em diversos estilos de jogos e necessidades, como por exemplo, controladores para jogos de tiro em primeira pessoa (FPS, do inglês *First Person Shooter*), oponentes em jogos de esporte, diálogos e inimigos em jogos de RPG (do inglês *Role Playing Game*), etc.

Um exemplo de aplicação de IA para um jogo pode ser visto na Figura 4. Nesse modelo, a tarefa da IA é divida em três seções: movimentação, decisão e estratégia, onde em volta desses componentes existem vários elementos adicionais referentes ao jogo. Nem todos os jogos precisam das três seções, isto é, alguns podem utilizar apenas da seção de estratégia, como xadrez, enquanto outros podem utilizar apenas de movimentação ou decisão.

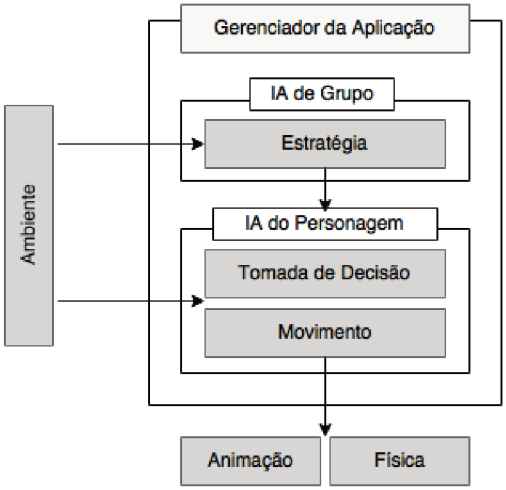


Figura 4 – Esquema de uma IA para jogos. Adaptado de: (MILLINGTON; FUNGE, 2009).

3.1 Decisões em Jogos

Grande parte dos jogos utilizam sistemas de tomada de decisão simples (MILLINGTON; FUNGE, 2009) como máquinas de estados finitos e árvores de decisão. Em alguns casos menos comuns, um sistema baseado em regras é utilizado. Porém, recentemente o interesse em outras técnicas como lógica *fuzzy* [1] e redes neurais vem crescendo.

Embora existam muitas técnicas para realizar uma decisão, todas podem ser vistas de uma forma similar (MILLINGTON; FUNGE, 2009). Um personagem processa um conjunto de informações para gerar uma ação. A entrada para o sistema de decisão é todo o conhecimento que o personagem possui, enquanto a saída é uma ação.

3.1.1 Comportamento Baseado em Metas

É possível fazer com que um personagem controlado pela máquina aparente ter objetivos e desejos, até mesmo com as técnicas mais simples de tomada de

[1] Teoria que se assemelha à maneira humana de se efetuar uma classificação, atribuindo um grau de pertinência às variaveis classificadas (SILVA, 2008) (ZADEH, 1965).

decisão. Esse é o objetivo dos algoritmos de comportamento baseado em metas (MILLINGTON; FUNGE, 2009).

Esses personagens possuem um conjunto de possíveis ações e metas. A abordagem mais simples é escolher a meta com maior prioridade e escolher uma ação que irá cumpri-la ou maximizá-la dentre as possíveis ações.

Essa abordagem é simples, rápida e pode fornecer bons resultados, principalmente em jogos com uma quantidade limitada de ações disponíveis, como jogos de tiro, ação em terceira pessoa, jogos de aventura ou RPGs.

3.1.2 Sistemas Baseados em Regras

Sistemas baseados em regras foram a vanguarda na pesquisas de IA na década de 70 e 80 (MILLINGTON; FUNGE, 2009). Muitas das mais famosas aplicações foram feitas com essa técnica e, provavelmente, é a técnica mais conhecida.

Esse tipo de sistema geralmente é dividido em duas partes: um banco de dados contendo informações sobre o ambiente e um conjunto de regras se-então.

As regras podem verificar no banco se as suas condições são cumpridas, se isso acontecer então a ação relacionada à aquela condição é executada.

Um exemplo visual de como funciona um sistema baseado em regras pode ser visto na Figura 5. Na figura tem-se um exemplo de conjunto de regras e uma base de dados, há também um intermediador que verifica quais das regras que tiveram suas condições cumpridas serão executadas. A implementação desse intermediador é opcional.

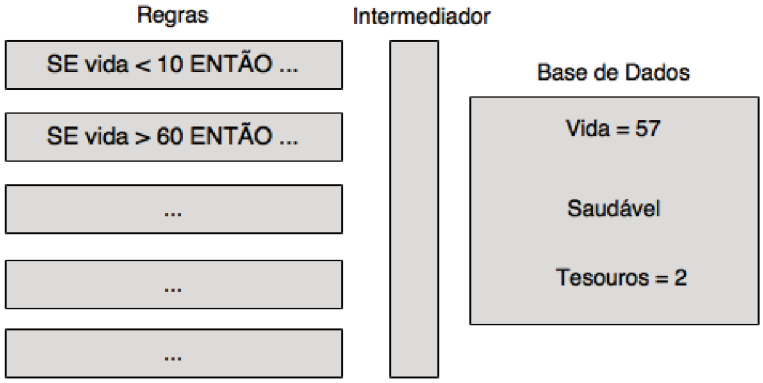


Figura 5 – Esquema de um sistema baseado em regras. Adaptado de: (MILLINGTON; FUNGE, 2009).

Dada o estado na base de dados e as regras, o intermediador deve verificar quais das regras tem suas condições cumpridas e escolher uma para executar utilizando algum critério.

3.2 Aprendizado em Jogos

O aprendizado é um tópico muito explorado em jogos (MILLINGTON; FUNGE, 2009). Ele faz com que o jogo possa se adaptar ao jogador e prover um maior desafio, além de apresentar o potencial de produzir personagens mais realísticos.

Existem duas formas de uma IA aprender: uma é aprender enquanto o jogo está sendo executado, chamada de aprendizado *on-line*; outra forma, largamente usada, é feita de forma *off-line*, seja na transição de cenas ou até mesmo antes de um jogo ser lançado no mercado. O aprendizado *off-line* permite com que mais algoritmos sejam testados e resultados mais satisfatórios sejam alcançados.

Neste trabalho, é utilizado o aprendizado *off-line*.

3.2.1 Aprendizado de Decisão

Para desenvolver todo o potencial do aprendizado de uma inteligência artificial, é necessário que ela seja capaz de aprender a tomar decisões (MILLINGTON; FUNGE, 2009).

O processo de aprendizado de decisão pode ser descrito por meio de um personagem que possui um conjunto de seus possíveis comportamentos. Junto a isso, ele também possui um conjunto de valores observáveis obtidos no ambiente. É necessário, deste modo, aprender a relacionar as decisões (qual comportamento escolher) com as observações feitas.

O aprendizado de decisão deve ser feito de modo que haja uma avaliação do algoritmo de aprendizado. Isso pode ser feito por meio de um supervisionamento forte ou um supervisionamento fraco. Supervisão Forte consiste em um conjunto de respostas corretas que, geralmente são fornecidas por um jogador humano: o agente registra as decisões do jogador e aprende a agir de acordo. Por outro lado, a Supervisão Fraca não exige respostas corretas, apenas uma função que avalia quão boa uma ação pode ser.

Neste trabalho, será utilizada a técnica de Supervisionamento Fraco.

Em cenários realistas de jogos, a quantidade de elementos observáveis é muito grande e a possibilidade de ação relativamente pequena. É desejável, no entanto, aprender algumas regras gerais e de forma rápida, de modo que tais regras sejam realistas, assim alguns elementos do ambiente podem ser desconsiderados pois não é necessário um aprendizado muito profundo ou específico.

Existem quatro grandes técnicas para se realizar o aprendizado em jogos: classificação *Naive Bayes*, Árvores de Decisão, Aprendizado por Reforço e Redes Neurais (REZENDE, 2003).

Neste trabalho, será aplicado o Aprendizado por Reforço.

3.2.2 Aprendizado por Reforço em Jogos

Em sua forma mais geral, um algoritmo de aprendizado para reforço possui três componentes: uma estratégia de exploração do jogo, uma função que deter-

mina quão boa é cada ação, e uma regra de aprendizado que liga as outras duas componentes.

Um algoritmo que é bastante simples e bastante utilizado é o *Q-Learning*, descrito na Seção 2.2.4.1 (MILLINGTON; FUNGE, 2009)(HOLMGÅRD et al., 2014b)(WATKINS; DAYAN, 1992), por ser simples de implementar, possuir resultados satisfatórios e ter sido muito explorado fora do contexto de jogos.

A seguir, serão descritos os estilos de jogos abordados neste trabalho.

3.3 Jogos *Dungeon Crawlers* e *Roguelike*

Video Games estão longe de ser uma mídia uniforme, possuindo vários gêneros que podem ter diferentes representações e manifestações (APPERLEY, 2006). Dentre esses gêneros, dois podem ser destacados: os jogos de estratégia e os *role-playing games* (RPG).

Os jogos de estratégia são geralmente divididos em estratégia em tempo real (RTS, *Real Time Strategy*) e estratégia baseada em turno (TBS, *Turn-based Strategy*). São jogos com uma alta exposição de informações que, geralmente, necessitam de uma associação entre essas informações e as melhores ações a serem tomadas, como exemplo desse tipo de jogo, temos *Starcraft*, *Warcraft* e a série *Total War*.

Por outro lado, o gênero RPG é intimamente ligado ao gênero de fantasia, onde o jogador toma o controle de um personagem em um determinado ambiente e decide o que o personagem deve fazer, para esse gênero pode ser citado *World of Warcraft*, a série *Final Fantasy* e a série *Pokémon*.

Existem também jogos que são, de certa forma, uma sub-categoria de um desses gêneros e que podem apresentar características dos dois. Na literatura aparecem os termos *Roguelike* (SMITH; BRYSON, 2014) e *Dungeon Crawler* (HOLMGÅRD et al., 2014c).

Ambos os gêneros possuem características de jogos de RPG, como tomar o controle de um personagem e tomar decisões por ele e ambos possuem características de jogos de estratégia, por serem jogos de ação baseada em turno e a

necessidade de se observar o ambiente para determinar a melhor ação.

Embora os dois gêneros possam ser tratados como o mesmo, é importante observar uma pequena diferença entre eles: ***Dungeon Crawlers*** são jogos onde o *design* dos níveis são feitos e testados antes dos testes ou lançamento do jogo; enquanto os níveis em jogos ***Roguelikes*** são gerados de modo procedural, ou seja, em tempo de execução. Dentro da categoria *Dungeron Crawler*, temos o *MiniDungeons*, descrito na seção a seguir.

3.4 *MiniDungeons*

MiniDungeons é um jogo simples que começou a ser desenvolvido em 2014 pelos pesquisadores Christoffer Holmgård, Antonios Liapis e Julian Togelius. É um jogo que implementa as mecânicas básicas de jogos *Roguelikes* e *Dungeon Crawlers* (HOLMGÅRD et al., 2014b), onde o objetivo é explorar *dungeons*[2] (HOLMGÅRD; LIAPIS; TOGELIUS, 2014). É um jogo de pesquisa que grava as ações do jogador de forma anônima e as envia para os desenvolvedores do jogo para ajudá-los em futuras pesquisas. O jogo é implementado com uma arquitetura que facilita a sua utilização para testar geração procedural de níveis e experimentos com agentes de Inteligência Artificial.

O jogo acontece em um ambiente 2D de dimensão 12x12 *tiles*[3], onde os *tiles* podem ser livres ou bloqueados. *Tiles* bloqueados não permitem que o jogador passe por eles. Os *tiles* livres podem conter um monstro, um tesouro, uma poção, a entrada ou a saída da *dungeon*. A interface do jogo pode ser vista na Figura 6.

[2] *Dungeon* pode ser traduzido como 'masmorra' e refere-se a um ambiente, em um jogo, com características de um labirinto onde existem monstros que oferecem algum tipo de ameaça ao jogador e recompensas para estimular o jogador a explorar o ambiente.

[3] Um *tile* é definido como uma área, caracterizada por um quadrado. É o elemento mínimo de jogos que utilizam esse recurso, sendo os jogos constituídos por um conjunto de *tiles*, chamado de *grid*.

Figura 6 – Captura de Tela do jogo Minidungeons.

O jogador possui um contador de vida e de tesouros. O jogador perde o jogo caso seus pontos de vida (*Hit Points*, HP) cheguem em zero. Cada nível é iniciado com 40 de HP e a cada turno o jogador pode mover para um *tile* adjacente e livre.

Quando o jogador move para um *tile* de monstro, o mesmo é removido e o jogador perde entre 5 e 14 de vida, decidido de forma aleatória, a posição dos monstros é fixa no mapa. Mover para um *tile* de tesouro remove o tesouro do mapa e aumenta o contador de tesouros em uma unidade. Mover para um *tile* com uma poção recupera ao jogador 10 pontos de vida (até um máximo de 40). Se o jogador se move para a saída do mapa o nível é concluído.

O fato dos níveis serem pequenos faz com que a densidade de decisões seja alta (HOLMGÅRD et al., 2014b), fazendo com que cada decisão tenha um grande

impacto e permitindo que cada nível seja completado mais rapidamente.

Além disso, após um agente terminar um nível é gerada uma imagem indicando a movimentação realizada pelo mapa. Deste modo, pode-se entender melhor como ocorreu as decisões e entender o percurso feito por um agente. Uma imagem que ilustra isto pode ser visualizada na Figura 7.

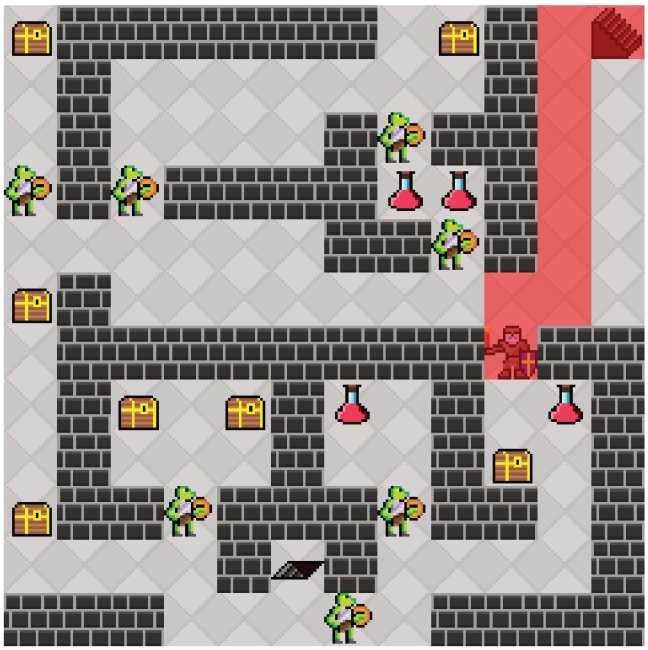


Figura 7 – Regiões do mapa pelas quais o jogador passou.

Após vários trabalhos feitos utilizando *MiniDungeons*, foi-se discutido que o jogo poderia ser visto como muito simples e que, para trabalhos futuros, ele deveria ser ampliado para prover uma maior complexidade (HOLMGÅRD et al., 2014a). Os desenvolvedores, então, começaram a trabalhar em uma segunda versão do jogo em 2015 (HOLMGÅRD et al., 2014c), que ainda não está disponível. A nova interface de *MiniDungeons* 2 pode ser vista na Figura 8.

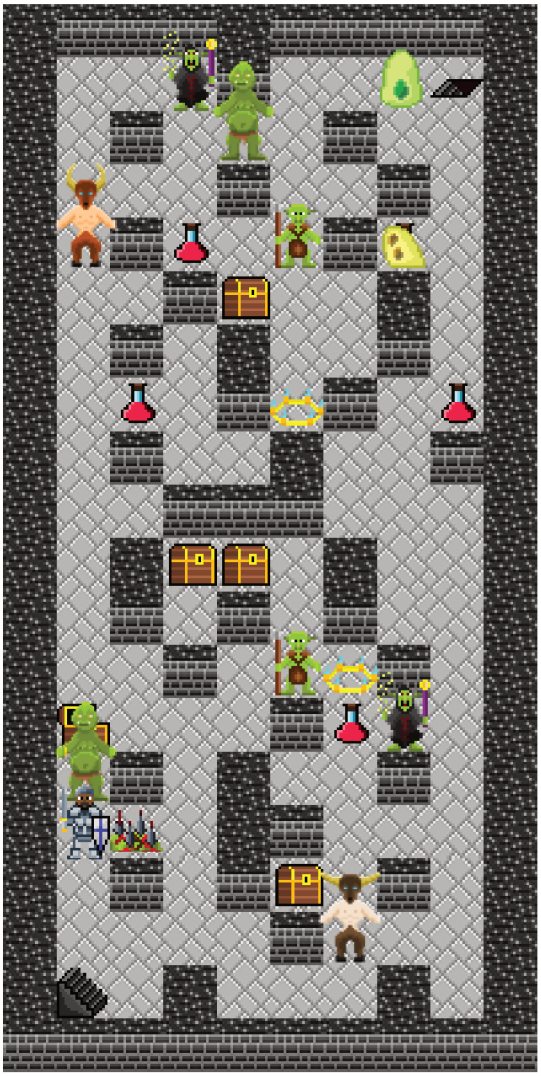


Figura 8 – *MiniDungeons* 2 (HOLMGÅRD et al., 2014c).

3.5 Trabalhos Relacionados

Esse Trabalho de Conclusão de Curso é inspirado principalmente em duas fontes bibliográficas: uma que utiliza algoritmos evolutivos em jogos (CROCOMO, 2008) e outra que aborda a representação de modelos de jogadores utilizando inteligência artificial (HOLMGÅRD et al., 2014b) (HOLMGÅRD et al., 2014a).

Nesta seção serão abordados esses trabalhos e como foram usados neste estudo.

3.5.1 Algoritmos Evolutivos em Jogos

O principal trabalho utilizado como referência foi a dissertação "Um Algoritmo Evolutivo para Aprendizado On-Line em Jogos Eletrônicos" por Marcio Kassouf Crocomo, de (CROCOMO, 2008).

Essa dissertação desenvolve um algoritmo evolutivo para o aprendizado *on-line* em jogos. Seu propósito foi responder a seguinte pergunta: "É possível construir um AE para o aprendizado *on-line* de jogos que possa ser utilizado na prática, por satisfazer os requisitos de rapidez, efetividade, robustez e eficiência?". O trabalho responde positivamente essa pergunta e argumenta ainda que o algoritmo evolutivo pode ser mais eficiente que a técnica de *Dynamic Scripting*, também implementada no trabalho.

O AE proposto por Crocomo utiliza como solução um conjunto de regras e apresenta uma lógica de geração de população inicial e operadores de recombinação e mutação para esse tipo de estrutura.

O Algoritmo Evolutivo implementado para nosso trabalho foi baseado no AE do Crocomo, apenas adaptando a estrutura e as operações para o problema de representação de modelos de jogadores e para o *Minidungeons*.

3.5.2 Representação de Modelos de Jogadores

Para o estudo da representação de modelos de jogadores, utilizou-se como principal referência os trabalhos de um grupo composto de quatro pesquisadores (Christoffer Holmgård, Antonios Liapis, Julian Togelius e Georgios Yannakakis). Atualmente existem três trabalhos desse grupo abordando esse tema, cada um aplicando uma técnica diferente.

O primeiro trabalho, *"Generative Agents for Player Decision Modeling in Games"* (2014b), utiliza da técnica de aprendizado de máquina *Q-Learning* para realizar a representação.

Neste artigo, foram usados agentes artificiais, que possuem propriedades

que permitem com que eles seja utilizados como uma abstração do processo de decisão humana.

Esta abordagem é baseada em três suposições:

1. Jogadores exibem um certo estilo ou uma certa tendência em suas decisões enquanto jogam e essa tendência pode ser aproximada por meio de uma função de utilidade, que modela as decisões dos jogadores.
2. A função de utilidade deve ser explicada considerando conceitos de psicologia e a mentalidade do jogador.
3. Modelos previamente definidos do processo de decisão de jogadores podem ser validados comparando os resultados dos modelos artificiais com observações empíricas de decisões de jogadores reais.

Para treinar o *Q-Learning* foram definidas cinco diferentes agentes com suas respectivas funções de recompensa, cada um representando um tipo diferente de jogador. Esses agentes e seus objetivos podem ser vistos na Tabela 1.

Tabela 1 – Descrição dos agentes.

Agente	Estilo de Jogo
Jogador Base (*Baseline*, B)	Busca a saída
Corredor (*Runner*, R)	Minimiza os movimentos
Sobrevivente (*Survivalist*, S)	Minimiza os riscos
Caçador de Monstros (*Monster Killer*, M)	Mata todos os monstros no cenário
Coletor de Tesouros (*Treasure Collector*, T)	Coleta todos os tesouros no cenário

As recompensas atribuídas a cada um desses agentes podem ser vistas na Tabela 2. O valor das recompensas foi estimado pelos desenvolvedores do jogo de forma que melhor representasse cada tipo de jogador.

Após o treino, foram utilizados cenários onde jogadores reais jogavam. Dada as mesmas condições no jogo, foram comparadas as decisões tomadas pelo jogador e pelo algoritmo *Q-Learning*.

Com os resultados, foi argumentado que todos os agentes são aproximações relevantes de diferentes jogadores. Na seção de discussão é reiterado que a função

Tabela 2 – Recompensa de cada Agente no *Q-Learning*.

	Agentes				
Evento	B	R	S	M	T
Matou Monstro				1	
Foi Morto			-1		
Chegou à saída	0.5	0.5	0.5	0.5	0.5
Pegou Tesouro					1
Movimento		-0.01			

objetivo deve ser uma abstração do jogador e que para trabalhos futuros poderiam ser utilizados Redes Neurais com *Q-Learning*, Busca de Monte Carlo e sistemas evolutivos baseados em regras.

Por fim, o trabalho conclui que o algoritmo *Q-Learning*, dada uma função objetivo que representa o modelo de decisão de um jogador, pode ser utilizado para compreender como o jogador interage com o cenário e que isso pode ser aplicado para modelagem de jogadores bem como para o *design* e desenvolvimento de jogos.

O segundo trabalho, *"Evolving Personas for Player Decision Modeling"* (2014a), utiliza uma Rede Neural para escolher as ações no jogo e uma Estratégia Evolutiva para encontrar os pesos atribuidos à rede.

Esse trabalho expande o anterior utilizando técnicas de inteligência computacional. São utilizadas sete *Perceptrons* Lineares, formando uma rede neural, que é combinada em um controlador evolutivo. A rede neural armazena oito informações: a vida atual do personagem e as distâncias de *Manhattan* entre o jogador e sete outros pontos:

- o monstro mais próximo;
- o tesouro mais próximo;
- o tesouro mais próximo evitando monstros;
- a poção mais próxima;
- a poção mais próxima evitando monstros;

- a distância até a saída;
- a distância até a saída evitando monstros.

Para o controlador evolutivo, é utilizada uma estratégia evolutiva em que a pior metade da população é totalmente descartada e depois preenchida com variações dos indivíduos sobreviventes. Em seguida, todas as soluções passam pela mutação, exceto os melhores 2% da população. Os experimentos foram feitos com uma população de 100 indivíduos, que foram treinados por 100 gerações.

A função *fitness* é obtida dividindo a quantidade de utilidade obtida em uma jogada pela quantidade máxima possível de utilidade a ser obtida naquele cenário.

Os valores de recompensa para a função de utilidade provém da intuição dos desenvolvedores do jogo e são similares aos valores mostrados para o trabalho anterior, ilustrados na Tabela 2. Nesse trabalho, os valores foram alterados para se adequar melhor ao novo algoritmo utilizado e são mostrados na Tabela 3.

Tabela 3 – Recompensa de cada Agente na Estratégia Evolutiva.

	Agentes				
Evento	B	R	S	M	T
Matou Monstro				1	
Foi Morto			-1		
Chegou à saída	0.5	0.5	0.5	0.5	0.5
Pegou Tesouro					1
Movimento	-0.01	-0.02	-0.01	-0.01	-0.01

Com os experimentos executados e os resultados, é argumentado que a abordagem evolutiva foi melhor que o *Q-Learning* em apresentar resultados similares a jogadores e em otimizar as recompensas.

O terceiro trabalho, *"Monte-Carlo Tree Search for Persona Based Player Modeling"* (2015), utiliza da técnica de busca e simulação *Monte-Carlo* para fazer a representação dos modelos e utiliza também a versão melhorada e mais complexa do *Minidungeons*, ainda não disponível.

Por utilizar uma versão diferente e melhorada do jogo, esse trabalho não faz comparação de resultados com os anteriores. Nele é argumentado que a busca *Monte Carlo* tem como grande vantagem ser um algoritmo *on-line* e poder se adaptar a diferentes situações.

4 Desenvolvimento

Neste Trabalho de Conclusão de Curso foram implementadas três técnicas de IA para o jogo *MiniDungeons*: um Algoritmo Evolutivo, o algoritmo de aprendizado de máquina *Q-Learning* e uma alternativa mais eficiente ao *Q-Learning*, o *HAQL*.

A implementação do Algoritmo Evolutivo foi realizada em dois módulos: o primeiro é o módulo propriamente do AE, implementado na linguagem C++. O segundo é o jogo *MiniDungeons*, onde foi desenvolvido um controlador capaz de utilizar as regras evoluídas pelo AE para jogar.

O *MiniDungeons* é um jogo pronto onde foi necessário apenas desenvolver um controlador que recebe os conjuntos de regras gerados pelo AE e retorna os resultados das jogadas. O jogo e o controlador são feitos na linguagem Java.

O desenvolvimento do *Q-Learning* e do *HAQL* ocorre totalmente dentro do *MiniDungeons*, sendo implementados por meio de um controlador, portanto, também utilizando a linguagem Java.

4.1 Algoritmo Evolutivo para *MiniDungeons*

A comunicação entre os dois módulos foi realizada por meio de um sistema cliente/servidor simples, implementado também neste trabalho.

O fluxo do programa ocorre em duas etapas principais. Na primeira etapa é feita a inicialização do Algoritmo Evolutivo, gerando a população inicial que representa o conjunto de possíveis regras. A população inicial é então enviada para o controlador, que utiliza as possíveis soluções para jogar e retorna os resultados. O módulo do algoritmo evolutivo pode, deste modo, calcular o *fitness* da população.

A segunda etapa acontece após a população inicial ter sido gerada e avaliada. Com o *fitness* já atribuído para a população, são aplicados operadores de recombinação e mutação, gerando novos candidatos, que neste caso, são novas

regras.

Os novos candidatos são, desta maneira, reavaliados pelo controlador no jogo, retornando os dados para o módulo do AE. Em seguida, a população passa por um processo de seleção natural, onde apenas os melhores candidatos são selecionados para a próxima geração.

O AE executa de modo cíclico, onde a cada nova geração as soluções são recombinadas e avaliadas, até que se alcance um critério de parada; neste caso, o número de gerações.

O funcionamento geral do algoritmo é ilustrado na Figura 9.

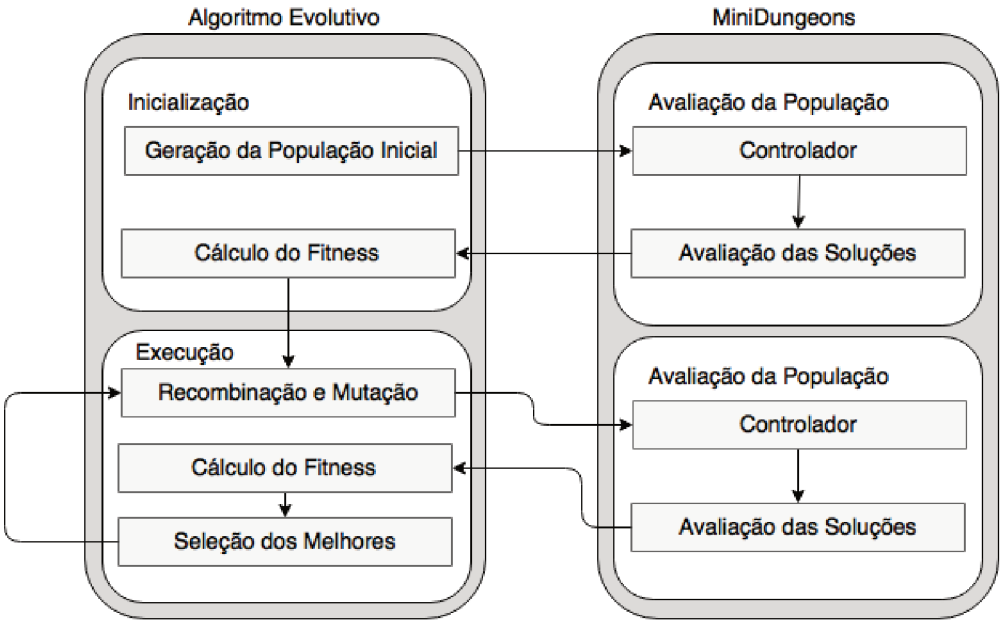


Figura 9 – Inicialização e Execução do Algoritmo Evolutivo.

A seguir serão explicados a estrutura do indivíduo e os operadores que foram utilizados na implementação.

4.1.1 Indivíduo

O indivíduo consiste de um conjunto de regras que são avaliadas e evoluídas no decorrer da execução do AE. No jogo *MiniDungeons* o jogador possui quatro

possíveis ações: mover-se para uma poção de vida, mover-se para um tesouro, matar um monstro ou mover-se para a saída.

Cada ação pode ser transformada em uma regra e, para adicionar mais possibilidades, cada ação pode ser feita evitando monstros ou não. Para melhor se adequar ao trabalho de referência (CROCOMO, 2008), é necessário que haja uma ordem de prioridade nas regras e que as regras possam estar ativas ou inativas. Sendo assim, a estrutura do indivíduo pode ser representada na Tabela 4.

Tabela 4 – Estrutura de um Indivíduo.

Regra	Prioridade	Ação	Condição	Ativa
1	Entre 1 e 7	Poção de Vida	c1	Verdadeiro ou Falso
2	Entre 1 e 7	Poção de Vida Evitando Inimigos	c2	Verdadeiro ou Falso
3	Entre 1 e 7	Pegar Tesouro	c3	Verdadeiro ou Falso
4	Entre 1 e 7	Pegar Tesouro Evitando Inimigos	c4	Verdadeiro ou Falso
5	Entre 1 e 7	Matar Inimigo mais Próximo	c5	Verdadeiro ou Falso
6	Entre 1 e 7	Ir para a Saída Evitando Inimigos	c6	Verdadeiro ou Falso
7	Entre 1 e 7	Ir para a Saída	c7	Verdadeiro ou Falso

A condição para execução de cada regra é interpretada pelo controlador. Por exemplo, para a regra 1, o agente irá buscar uma poção de vida, sem ignorar monstros, caso sua vida atual seja menor que a variável de condição c1.

No algoritmo evolutivo a condição foi representada por um número inteiro que é gerado aleatoriamente junto com a criação da população e pode ser alterado no operador de mutação.

No controlador, a interpretação das regras segue como descrito na Tabela 5.

O conjunto de regras será representado por uma lista de regras, onde cada regra, na implementação, tem o formato demonstrado na Tabela 6.

Tabela 5 – Intepretação das Regras pelo Controlador.

Regra	Interpretação
1	O jogador irá pegar a poção de vida mais próxima se a vida atual for menor que a variável de condição.
2	O jogador irá pegar a poção de vida mais próxima evitando inimigos se a vida atual for menor que a variável de condição.
3	O jogador irá pegar o tesouro mais próximo se a vida atual for maior que a variável de condição.
4	O Controlador não leva em consideração a vida atual para ir ao tesouro mais próximo evitando monstros.
5	O jogador irá matar o monstro mais próximo se sua vida atual for maior que a variável de condição.
6	O Controlador não leva em consideração a vida atual para ir à saída evitando monstros.
7	O jogador irá para a saída se sua vida atual for maior que a variável de condição.

Tabela 6 – Estrutura de codificação de uma regra.

Prioridade	Variável Condição	Ativada	Ação
Inteiro	Inteiro	Booleano	String

4.1.2 Seleção dos Pais

Antes de se realizar a recombinação, é necessário que se faça a seleção dos dois pais a serem recombinados. São implementadas duas possibilidades de seleção: Seleção Aleatória e Seleção por Torneio.

A Seleção Aleatória sorteia dois indivíduos da população onde cada um possui a mesma probabilidade de ser escolhido.

A Seleção por Torneio, por sua vez, sorteia n indivíduos aleatoriamente que, então, competem entre si, sendo selecionado o que tem o melhor *fitness*. Isso é feito uma vez para cada pai.

4.1.3 Recombinação

O operador de recombinação (também chamado de cruzamento ou *crossover*) foi explicado anteriormente na Seção 2.1.

O operador implementado é chamado de cruzamento cíclico. Ele foi selecionado pelo fato de que as prioridades dos indíviduos não podem ser repetidas, favorecendo a utilização desse operador.

De acordo com Gabriel e Delbem (2008), no Cruzamento Cíclico, dado dois pais, a primeira posição no filho recebe o primeiro gene do primeiro pai. O próximo gene do filho é obtido do segundo pai. Este gene é copiado para o filho na posição em que este mesmo gene aparece no primeiro pai.

O algoritmo segue copiando para o filho os genes do segundo pai na posição correspondente do primeiro pai até que se forme um ciclo. Em outras palavras, até que a posição a ser copiada já existe no filho. Neste caso, se restarem posições não preenchidas no filho, elas são preenchidas da esquerda para a direita com os genes do primeiro pai na ordem relativa que aparecem. Trocando-se o primeiro pai pelo segundo é possível obter outro descendente.

No algoritmo implementado há uma pequena modificação: ao invés do cruzamento ocorrer sempre na primeira posição do filho e do pai, é sorteada uma posição aleatória para iniciar o cruzamento. Após isso, o algoritmo segue como descrito.

Essa mudança foi feita para estimular uma maior diversidade de indivíduos resultantes de um mesmo cruzamento.

A taxa de recombinação é de 80%, destacando que os filhos gerados sempre são diferentes dos pais que os geraram.

4.1.4 Mutação

O operador de mutação foi explicado anteriormente na Seção 2.1.

A mutação utilizada neste trabalho foi criada especificamente para ele com o objetivo de simular os mesmos efeitos da mutação utilizada por Crocomo (2008).

O operador é aplicado com uma probabilidade de 5% de acontecer e a alteração pode se dar de três formas, sendo que a forma com que ela ocorre é escolhida aleatoriamente. As possibilidades de mutação são:

1. **Mutação na Prioridade:** Troca a prioridade de duas regras selecionadas aleatoriamente.
2. **Mutação na Variável Condição:** Adiciona ou subtrai aleatoriamente um valor à variável de condição.
3. **Mutação na Ativação da Regra:** Ativa ou desativa uma regra aleatória.

4.1.5 Cálculo do *Fitness*

O *fitness* adotado será o mesmo adotado por Holmgård em seus trabalhos de representação de modelos de jogadores (HOLMGÅRD et al., 2014a). O cálculo do *fitness* é dado pela fórmula:

$$y = \frac{utilidade\_obtida}{utilidade\_max} \tag{4.1}$$

Onde o cálculo da utilidade obtida e máxima podem variar de acordo com o agente sendo utilizado. Os valores de recompensa para as funções de utilidade foram definidos na Tabela 2 e na Tabela 3.

4.1.6 Seleção de Indivíduos para a Próxima Geração

Após aplicar os operadores de recombinação e mutação e calcular o *fitness* dos novos indivíduos, é necessário selecionar quais indivíduos da população estão mais aptos a sobreviver para a próxima geração. Nesse trabalho foi implementado a seleção por Elitismo.

Na seleção por Elitismo, apenas n indivíduos da população sobrevivem, sendo n menor que o tamanho da população original. Nesse trabalho, são mantidos 20% dos melhores indivíduos de uma geração para a outra.

4.2 *Q-Learning* para *MiniDungeons*

O algoritmo de aprendizado de diferença temporal *Q-Learning* foi explicado de modo genérico na Seção 2.2.4.1. Nessa seção será explicada a sua adaptação para o problema abordado.

A implementação do *Q-Learning* foi feita usando o método utilizado por Holmgård (2014b) com o intuito de replicar seus resultados e comparar com o algoritmo evolutivo desenvolvido.

No problema em questão, o agente possui quatro possíveis movimentos: mover para cima, para baixo, para esquerda e para a direita. Ele deve observar o estado atual s, executar um movimento e observar o estado resultante s'.

No trabalho de referência, o valor $Q(s,a)$ é aumentado por $\alpha[r+\gamma max_a Q(s',a)-Q(s,a)]$, onde r é a recompensa no estado s', α é a taxa de aprendizado e γ é a taxa de desconto para futuras recompensas.

O treinamento dos agentes foi feito com $2.5.10^5$ jogadas, com fator de aprendizado $\alpha = 0.5$ e taxa de desconto $\gamma = 0.9$. Durante o treinamento, a ação com maior valor para a função Q será selecionada pela política ϵ-*greedy*, onde ϵ começa como 1 e tem seu valor diminuído até 0.1. Essa diminuição de ϵ começa após 2500 jogadas e decai linearmente até o fim do treinamento. Quando não é selecionado o melhor valor da função Q, a ação tomada é a ação menos explorada naquele estado.

Os mesmos valores e condições serão adotados nesse trabalho, visto que o objetivo era atingir os mesmos resultados de Holmgård (2014b).

A implementação desse algoritmo apresentou algumas dificuldades: (i) A representação da tabela de estados não é explicitada no artigo de referência, (ii) não é explicada a posição em que o agente deve iniciar um episódio, (iii) o desempenho do algoritmo, no fator tempo, foi insatisfatório, levando à limitação da quantidade de ações em cada episódio do treinamento, porém, na referência não é dito se o treinamento possui essa limitação ou não, embora a execução após o treinamento tenha. Cada um desses itens será detalhado em seguida.

A representação da tabela de estados (Tabela Q) é de grande importância

para o *Q-Learning*, pois é onde são armazenados os resultados de cada movimento e onde é avaliado o quão bom é um movimento de um estado para o outro. No artigo de Holmgård (2014b) um estado do jogo deve armazenar todas as informações do ambiente: a posição do jogador, sua quantidade de vida, monstros, tesouros e poções, bem como a informação se um monstro está vivo ou morto, se um tesouro pode ser ou se já foi coletado e se uma poção está disponível ou não.

O modo mais intuitivo de se representar essa estrutura é utilizando uma matriz, que garante rapidez ao consultar os dados na memória. Para esse problema específico, a matriz seria caracterizada por 6 dimensões (posição no eixo horizontal, posiçao no eixo vertical, monstros, tesouros, poções, quantidade de vida), o que significa que a exigência de memória seria muito alta.

Na prática, não foi possível codificar a representação por matrizes por esta utilizar mais memória do que o disponível. A alternativa foi representar utilizando uma estrutura *hash* que, embora mais lenta, é alocada de modo dinâmico. Assim, posições não acessadas não estão presentes na memória, evitando o problema de utilizar mais espaço que o disponível.

Sobre a posição inicial, o algoritmo *Q-Learning* geralmente possui uma política em que o agente é colocado em uma posição aleatória em cada episódio, mas não há informações se foi feito assim, de fato. Além disso, inicializar em uma posição aleatória em *MiniDungeons* pode significar que o agente iniciará na mesma posição de um monstro, um tesouro, uma poção ou até da saída do mapa. Nesse trabalho foi adotado uma política de inicializar o agente em uma posição aleatória, desde que fosse uma posição vazia.

Em questão de desempenho, cada episódio tomava mais de 1 minuto, somado ao fato de que a execução possuia 250.000 episódios. Isto tornava um ambiente onde demorariam dias para executar o treinamento de apenas um agente, impossibilitando, na prática, a execução do algoritmo para esse trabalho.

Uma possível explicação para o desempenho ruim é a utilização de uma tabela *hash* para representar os estados do algoritmo, que é lenta em comparação à utilização de matrizes.

Para melhorar o desempenho, a quantidade de ações que um agente pode

tomar em um episódio foi limitada em 300 ações. Essa quantidade de ações é a quantidade limite que um agente treinado pode tomar no trabalho de Holmgård (2014b), porém não há informações de que esse limite era utilizado para treiná-los.

4.3 *HAQL* para *MiniDungeons*

O desempenho do *Q-Learning* implementado, embora seguisse todas as especificações fornecidas no artigo utilizado como base (HOLMGÅRD et al., 2014b), não foi satisfatório, tendo tempos de execução muito elevados e apresentando um comportamento não-convergente para alguns mapas.

Considerando isso, foi necessário buscar alternativas para a execução do *Q-Learning*. O primeiro passo foi pesquisar e atestar que, de acordo com a literatura, a execução do treinamento desse algoritmo é realmento lento (WHITEHEAD, 1991) (HOLMGÅRD et al., 2014b) (HOLMGÅRD et al., 2014a) (BIANCHI; RIBEIRO; COSTA, 2004) (HASSELT; GUEZ; SILVER, 2016) (MARTINS; BIANCHI, 2018). Verificado isso, o segundo passo foi procurar por uma alternativa, sendo que o algoritmo escolhido foi o *HAQL*, que foi introduzido na Seção 2.2.4.2.

Esse método utiliza-se de variáveis parametrizadas, onde seus valores foram os mesmos valores utilizado na tese de Bianchi (BIANCHI, 2004). Tais valores são: $\eta = \beta = \xi = 1$ e para θ é utilizada a função matemática de adição.

Os valores do fator de aprendizado (α), desconto (γ) e de política de exploração (ϵ) são os mesmos utilizados no *Q-Learning*, que derivam do trabalho de Holmgård (2014b). Os valores são: $\alpha = 0.5$, $\gamma = 0.9$ e ϵ tem seu valor iniciado como 1 e após 2500 episódios começa a decair linearmente até 0.1.

5 Experimentos e Resultados

Para avaliar a eficiência dos dois algoritmos implementados, o algoritmo evolutivo e o *HAQL*, foram executados dois experimentos. Um experimento consistiu em utilizar os algoritmos para jogar todos os mapas, com todos os agentes, a fim de verificar os resultados para essa ocasião. Outro experimento consistiu em utilizar os algoritmos para jogar apenas um mapa simples, com todos os agentes, com o objetivo de medir a eficiência de tempo de cada algoritmo, bem como seus resultados específicos para aquele mapa.

O primeiro consiste em treinar todos os cinco tipos de agentes: *Baseline*, *Runner*, *Survivalist*, *Treasure Collector* e *Monster Killer* em todos os 11 mapas disponíveis em *MiniDungeons*. Para o *HAQL* cada agente joga mais 20 vezes após o fim do treinamento; para o AE, cada agente joga também mais 20 vezes com o conjunto de regras com melhor *fitness*.

O jogo *MiniDungeons* é capaz de gerar estatísticas individuais das execuções, bem como suas médias. Também é possível gerar a imagem mostrando o caminho percorrido pelo agente em um mapa em cada uma das execuções.

As estatísticas geradas pelo *HAQL* e pelo AE podem ser comparadas para verificar a eficiência de cada algoritmo e seus pontos fracos e fortes. As imagens podem ajudar a entender como o treinamento e os conjuntos de regras se manifestaram nas diferentes situações. As imagens também auxiliam uma análise visual, verificando se o resultado é bom ou não e se é possível comparar a eficiência de cada algoritmo com a eficiência de um jogador humano.

A execução do algoritmo evolutivo foi feita com os parâmetros listados na Tabela 7.

Os parâmetros para a execução do *HAQL* e do *Q-Learning* estão mostrados na Tabela 8.

Na próxima seção estão listados os resultados das estatísticas de cada algoritmo, bem como uma análise de acordo com cada agente e o que se era esperado.

Tabela 7 – Parâmetros do AE.

Parâmetro	Valor
Tamanho da População	25
Gerações	100
Vezes Jogadas	20
Fator Recombinação	80%
Fator Mutação	5%

Tabela 8 – Parâmetros do *HAQL* e *Q-Learning*.

Parâmetro	Valor
Episódios	250000
Fator Aprendizado	0.5
Fator Desconto	0.9

O resultado visual da melhor execução de cada mapa, considerando todos os agentes e os dois algoritmos do trabalho, pode ser visto na Seção de Apêndices, na página 69.

5.1 Resultados do Algoritmo Evolutivo

A execução do AE foi feita com os parâmetros indicados na Tabela 7 para os cinco agentes, cada um com sua função objetivo, especificados na Tabela 3. Ao fim da execução, foi-se utilizado o melhor conjunto de regras para jogar, os resultados dessas jogadas estão ilustrados na Tabela 9.

Tabela 9 – Resultados do Algoritmo Evolutivo.

Estatística	B	R	S	M	T
Monstros	20.85	20	20	54.55	36.1
Tesouros	8.4	11.75	11.65	35.5	40.15
Poções	2.45	2	2.05	21.95	11.4
Explorados	229.4	249.5	249.45	445.25	414.9
Mortes	20	5	6	34	6

A interpretação de cada agente pode ser vista na Tabela 1.

O agente *Baseline* (B) tem como único objetivo chegar na saída, por isso não possui números ótimos em nenhum aspecto. Justamente por ter esse caráter, esse agente é muito útil para fazer comparações e verificar se os outros agentes realmente otimizam os objetivos especificados.

O agente *Runner* (R), tem como único objetivo chegar na saída porém possui uma penalidade maior para cada movimento tomado. O AE não foi eficiente em otimizar essa estatística, visto que a quantidade de estados explorados é maior que a do agente *Baseline* e ligeiramente maior que a do agente *Survivalist*.

O agente *Survivalist* (S), também visa chegar à saída, porém possui uma penalidade alta para cada morte, por isso o agente manifesta uma média menor de monstros mortos em relação a B e, por consequência, um número de mortes menor. Um aspecto interessante a se notar é que o agente S explorou menos estados que R e teve número de mortes maior.

Os dois próximos agentes possuem objetivos diferentes em relação aos anteriores e se aproximam mais a jogadores humanos, o agente *Monster Killer* (M) possui uma alta recompensa em matar monstros e o *Treasure Collector* (T) possui uma alta recompensa em coletar tesouros, ambos possuem recompensas menores em alcançar a saída do mapa.

A primeira grande diferença a ser notada nos resultados é a quantidade de estados explorados, que aumenta de forma notável nesses dois agentes. Também são os agentes com maiores números poções encontradas e, no caso do *Monster Killer*, o maior número de mortes. Isso pode acontecer pois, devido a alta exploração, os agentes se envolvem em mais jogadas arriscadas, resultando em mais perda de vida e, em alguns casos, a morte. As poções são uma forma de minimizar o risco das jogadas, permitindo explorar mais estados com uma boa margem de segurança.

O agente M é o que mais mata monstros e o agente T é o que mais coletou tesouros, cumprindo com seus objetivos.

5.2 Resultados do *HAQL*

A execução do *HAQL* foi feita com os parâmetros mostrados na Tabela 8 para os cinco agentes, cada um com a sua função objetivo especificada na Tabela 2. O resultado obtido é ilustrado na Tabela 10

Tabela 10 – Resultados do *HAQL*.

Estatística	B	R	S	M	T
Monstros	20.5	21.95	21.05	50.85	50.25
Tesouros	6.4	6.8	6.85	49.45	49.85
Poções	4.5	4.9	3.6	7.05	5.55
Explorados	226.3	233.6	227.85	344.95	346
Mortes	11	14	17	163	169

A análise dos resultados do *HAQL* é similar a análise feita ao AE na seção anterior, com algumas pequenas mudanças.

Para esse algoritmo, o agente B foi mais eficiente em explorar estados em relação a R e foi mais eficiente a evitar mortes do que o agente S. Os três possuem dados parecidos para monstros e tesouros.

Por outro lado, os agentes M e T, possuem um número muito mais alto de mortes pois engajam mais em jogadas arriscadas e não há uma recompensa direta por tomar poções de vida.

Vale notar que os resultados desses dois agentes foram muito próximos na questão de monstros, tesouros e estados explorados.

5.3 Comparação entre o Algoritmo Evolutivo e *HAQL*

Conforme especificado na Seção 1.3, um dos objetivos deste trabalho foi fazer a comparação do AE implementado com a literatura e outros métodos para verificar seus pontos fortes, fracos e onde ele pode ser utilizado.

Nessa seção, seus resultados serão comparados com o outro algoritmo implementado baseado na literatura, o *HAQL*.

5.3.1 Comparação de Desempenho dos Agentes

Tabela 11 – Resultados do agente *Baseline*.

	HAQL	AE
Monstros	20.5	20.85
Tesouros	6.4	8.4
Poções	4.5	2.45
Ações Tomadas	226.3	229.4
Mortes	11	20

Para o agente *Baseline* (Tabela 11) é possível perceber uma certa similaridade nos resultados, principalmente nos monstros mortos e quantidade de ações tomadas. Isso pode indicar que os dois algoritmos percorrem caminhos similares nos mapas. O sistema baseado em regras utilizado no algoritmo evolutivo permite que o agente explore também outros objetivos desde que não prejudique no *fitness*, isso pode explicar a maior quantidade de mortes e tesouros.

Esse acontecimento também pode ser visto como um modelo de decisão mais complexo. Em alguns mapas é possível buscar tesouros sem nenhum tipo de risco, o algoritmo evolutivo pode ser capaz de criar um conjunto de regras que obedeça o estilo de decisão do agente (nesse caso, chegar até a saída) e que ainda faça essas explorações sem risco, ou até jogadas arriscadas, desde que, no fim, consiga chegar à saída.

Tabela 12 – Resultados do agente *Survivalist*.

	HAQL	AE
Monstros	21.05	20
Tesouros	6.85	11.65
Poções	3.6	2.05
Ações Tomadas	227.85	249.45
Mortes	17	6

O aspecto mais importante do agente *Survivalist* (Tabela 12) é a minimização da quantidade de mortes, nesse caso, pode-se perceber que o AE foi mais eficiente, tendo quase um terço das mortes do HAQL.

Tabela 13 – Resultados do agente *Runner*.

	HAQL	AE
Monstros	21.96	20
Tesouros	6.8	11.75
Poções	4.9	2
Ações Tomadas	233.6	249.5
Mortes	14	5

O agente *Runner* (Tabela 13) tem como objetivo minimizar as ações tomadas. Analisando esses resultados, o AE possui um resultado inferior, realizando, em média, 16 ações a mais.

No entanto, deve-se ressaltar que a execução do *HAQL* não converge para um dos mapas (Mapa 3), fazendo com que o agente não chegue à saída nesse mapa e, por consequência, realize menos movimentos que o necessário. Esse pode ser um dos motivos para que o *HAQL* tenha em média menos ações realizadas.

Tabela 14 – Resultados do agente *Monster Killer*.

	HAQL	AE
Monstros	58.05	54.55
Tesouros	49.45	35.5
Poções	7.05	21.95
Ações Tomadas	344.95	445.25
Mortes	163	34

O agente *Monster Killer* (Tabela 14) tem como maior recompensa matar monstros. Analisando unicamente esse aspecto, o *HAQL* tem melhor desempenho.

Porém outro aspecto que pode ser analisado, é a quantidade de poções, quantidade de ações realizadas e a quantidade de Mortes. A maior quantidade de poções e menor de mortes indica que, mesmo tendo como maior objetivo engajar em combate (que possui alto risco), o AE foi capaz de encontrar um conjunto de regras que tenta minimizar o risco ainda mantendo a característica principal de matar monstros. A quantidade maior de ações tomadas pode ser um reflexo do AE buscando mais poções de vida.

Em outras palavras, os dois algoritmos foram eficientes em cumprir seu

objetivo principal para esse agente: Matar monstros, mas enquanto o *HAQL* visa apenas fazer isso, o AE foi capaz de manifestar um comportamento mais complexo e minimizar riscos. Pode-se argumentar que esses dois comportamentos são dois estilos de jogadores diferentes dentro da categoria *Monster Killer*.

Tabela 15 – Resultados do agente *Treasure Collector*.

	HAQL	AE
Monstros	50.25	36.1
Tesouros	49.85	40.15
Poções	5.55	11.4
Ações Tomadas	346	414.9
Mortes	164	6

A análise do agente *Treasure Collector* (Tabela 15) é similar ao do agente *Monster Killer*. O *HAQL* foi superior em números brutos, porém o AE explorou um espaço maior do mapa, obteve mais poções e morreu menos vezes, minimizando o risco.

Uma diferença é que o *HAQL* mata mais monstros do que coleta tesouros, mesmo o objetivo principal sendo fazer a coleta, enquanto o AE engaja menos em combate, isso atesta a afirmação de que o AE é mais eficiente em minimizar riscos.

Conforme argumentado no agente *Monster Killer*, os resultados de cada algoritmo pode refletir diferentes estilos de jogadores dentro de uma mesma categoria, sendo que o AE é capaz de reproduzir um comportamento mais complexo.

5.3.2 Comparação de Eficiência de Tempo

Para fazer uma comparação entre o tempo de cada algoritmo, foi executado um treinamento para o *HAQL* com 250.000 episódios e para o AE o algoritmo foi executado com os mesmos valores da Tabela 7. O experimento foi executado no Mapa 0, o mais simples, e foi feito para todos os agentes. O resultado pode ser visto na Tabela 16.

Como pode ser observado, o AE com os parâmetros utilizados é executado em um intervalo de tempo muito menor que o *HAQL*, além de ter um tempo de execução muito mais previsível.

Tabela 16 – Comparação de tempo entre AE e *HAQL*.

	HAQL	AE
Baseline	192min 12seg	21min 8seg
Runner	223min 28seg	18min 32seg
Survivalist	175min 15seg	20min 3seg
Monster Killer	204min 7seg	19min 58seg
Treasure Collector	290min 33seg	21min 3seg
Média	217min 7seg	20min 9seg
Desvio Padrão	2679.15 (s)	63.17 (s)

Na próxima seção, serão comparados os resultados numéricos da execução dos agentes para esse teste executado, a fim de verificar se os resultados do AE são tão bons quanto os do *HAQL*.

5.3.3 Análise dos Resultados

Conforme feito na primeira comparação entre os dois algoritmos, será comparado, agente a agente, os resultados do Mapa 0.

Os resultados para o agente *Baseline* podem ser vistos na Tabela 17, o agente *Runner* na Tabela 18 e o agente *Survivalist* na Tabela 19.

Tabela 17 – Resultados do agente *Baseline* para o Mapa 0.

	HAQL	AE
Monstros	2	2
Tesouros	0	0
Poções	1	1
Ações Tomadas	20	20
Mortes	0	0

O que pode ser notado na análise desses três agentes para o Mapa 0, é que os resultados da execução dos três são idênticas e ambos algoritmos chegaram aos mesmos resultados. Ao menos nesses casos, os dois algoritmos são equivalentes.

Os resultados para os agentes *Monster Killer* podem ser vistos na Tabela 20 e do agente *Treasure Collector* na Tabela 21.

Tabela 18 – Resultados do agente *Runner* para o Mapa 0.

	HAQL	AE
Monstros	2	2
Tesouros	0	0
Poções	1	1
Ações Tomadas	20	20
Mortes	0	0

Tabela 19 – Resultados do agente *Survivalist* para o Mapa 0.

	HAQL	AE
Monstros	2	2
Tesouros	0	0
Poções	1	1
Ações Tomadas	20	20
Mortes	0	0

Tabela 20 – Resultados do agente *Monster Killer* para o Mapa 0.

	HAQL	AE
Monstros	6.9	6.85
Tesouros	2	6.55
Poções	2	3
Ações Tomadas	41.2	92.65
Mortes	20	9

Para o agente *Monster Killer* pode-se observar que o AE obtém resultados superiores em todos os aspectos, obtendo uma média similar de monstros mortos porém sobrevivendo em mais da metade das jogadas, explorando um espaço maior.

Tabela 21 – Resultados do agente *Treasure Collector* para o Mapa 0.

	HAQL	AE
Monstros	3.35	1.15
Tesouros	5.45	4.15
Poções	0.9	0
Ações Tomadas	190.2	46.85
Mortes	0	0
Vezes sem Completar	11	0

Para o agente *Treasure Collector*, o AE não conseguiu gerar um conjunto de regras que coletasse tantos tesouros quanto o *HAQL*, porém, o AE chegou ao fim em todas as 20 jogadas, ao contrário do *HAQL* que não alcançou a saída em 11 tentativas.

5.4 Comparação e Análise do AE com a literatura

Nessa seção estão sumarizados os resultados do AE. Os resultados são os mesmos da Seção 5.1 colocado lado a lado com os resultados da literatura (HOLMGÅRD et al., 2014b) (HOLMGÅRD et al., 2014a) e separados por agentes para facilitar a comparação.

O algoritmo de *Q-Learning* foi treinado por 250.000 episódios antes de jogar, o AE e o algoritmo adaptativo (HOLMGÅRD et al., 2014a) foram previamente executados para evoluir os conjuntos de regras ou parâmetros utilizados para se jogar. Todos os algoritmos jogaram 20 vezes cada mapa. Os resultados refletem as estatísticas de jogar os mapas 1 ao 10.

O algoritmo adaptativo foi explicado na Seção 3.5.2 e consiste em utilizar, em conjunto, uma rede neural e uma estratégia evolutiva para jogar *MiniDungeons*.

A comparação para o agente *Baseline* pode ser visto na Tabela 22, para o agente *Survivalist* na Tabela 23, para o agente *Runner* na Tabela 24, para o agente *Monster Killer* na Tabela 25, para o agente *Treasure Collector* na Tabela 26. O AE se refere ao algoritmo implementado nesse trabalho e é apresentado na primeira coluna, a próxima coluna apresenta os resultados do *Q-Learning* apresentado em (HOLMGÅRD et al., 2014b), a última coluna apresenta um algoritmo evolutivo e adaptativo, apresentados em (HOLMGÅRD et al., 2014a).

Será feita uma comparação geral do desempenho dos três algoritmos.

Entre o AE e o algoritmo adaptativo pode-se perceber uma grande diferença mesmo ambos utilizando as mesmas funções de *fitness*, isso ocorre pois o AE utilizado nesse trabalho evolui um conjunto de regras e, mesmo se não for recompensado, o agente pode tomar certas ações como coletar um tesouro, matar um monstro, etc... desde que consiga de modo satisfatório atingir o objetivo principal,

Tabela 22 – Resultados do agente *Baseline* comparando com a literatura.

	AE	Q-Learning 2014b	Adaptativo 2014a
Monstros	20.85	22.7	22.1
Tesouros	8.4	9.4	5.5
Poções	2.45	2.1	1.6
Ações Tomadas	229.4	236	217
Mortes	20	13	10

que nesse caso, é alcançar a saída.

Levando em consideração os três algoritmos, é dificil apontar um que obteve melhores resultados pois esse agente não visa minimizar ou maximizar nenhuma dessas estatísticas analisadas, porém ele é util ao ser comparado com os resultados dos outros agentes.

Tabela 23 – Resultados do agente *Survivalist* comparando com a literatura.

	AE	Q-Learning 2014b	Adaptativo 2014a
Monstros	20	21.4	21.6
Tesouros	11.65	11	7.2
Poções	2.05	3.1	1.2
Ações Tomadas	249.45	244	227.2
Mortes	6	0	2

O agente *Survivalist* visa minimizar o número de mortes. O AE consegue diminuir muito a quantidade de mortes em relação ao agente *Baseline*, porém não de modo tão eficiente quanto o algoritmo adaptativo, no entanto, o AE explorou mais estados e coletou mais tesouros. Novamente, isso pode acontecer pois a utilização de conjuntos de regras propicia que o agente tome outras ações que podem beneficiar não somente a estatística a ser minimizada.

Levando em consideração os três algoritmos, as execuções dos *Q-Learning* foram as mais eficientes, porém os algoritmos evolutivos apresentaram também um número muito baixo de mortes, 2 e 6 em 200 execuções.

O agente *Runner* visa minimizar as ações tomada (estados explorados), o AE e o adaptativo apresentaram resultados diferentes, o segundo apresentou uma estatística similar ao *Q-Learning*. Por outro lado, o AE apresentou uma maior

Tabela 24 – Resultados do agente *Runner* comparando com a literatura.

	AE	Q-Learning 2014b	Adaptativo 2014a
Monstros	20	22.6	24.1
Tesouros	11.75	7.8	4.7
Poções	2	2	1.7
Ações Tomadas	249.5	230	230.2
Mortes	5	22	15

quantidade de ações tomadas, porém, conseguiu minimizar o risco apresentando um número menor de mortes.

Comparando todos os algoritmos os melhores resultados vieram dos algoritmos utilizados na literatura.

Tabela 25 – Resultados do agente *Monster Killer* comparando com a literatura.

	AE	Q-Learning 2014b	Adaptativo 2014a
Monstros	54.55	53.8	69.5
Tesouros	35.5	9.4	10.3
Poções	21.95	16.1	30.9
Ações Tomadas	445.25	302	413.9
Mortes	34	63	89

O AE apresenta uma estatística de monstros mortos similares ao algoritmo de aprendizado por reforço, que é um número menor que o conseguido pelo algoritmo adaptativo, porém, ele engaja menos em ações mais arriscadas, morrendo menos e coletando mais tesouros.

Considerando os três algoritmos, o adaptativo possui uma eficiência muito maior em matar monstros, que é o objetivo principal desse agente, porém, o AE também consegue um número elevado de monstros enquanto conseguiu minimizar riscos e coletar mais tesouros.

Entre os algoritmos evolutivos, novamente a diferença é grande e, do mesmo modo que o agente *Monster Killer*, o adaptativo é mais eficiente considerando apenas o critério de tesouros coletados, mas o AE, embora tenha sido o pior nesse quesito, conseguiu minimizar muito os ricos, engajando em menos combate, tomando mais poções e tendo significativamente menos mortes.

Tabela 26 – Resultados do agente *Treasure Collector* comparando com a literatura.

	AE	Q-Learning 2014b	Adaptativo 2014a
Monstros	36.1	48.2	50.1
Tesouros	40.15	48.9	57.5
Poções	11.4	3.7	6.5
Ações Tomadas	414.9	328	413.9
Mortes	6	169	151

Levando em consideração os três algoritmos, o melhor em coletar tesouros foi o adaptativo, mas o AE apresentou um comportamento mais complexo, ainda caracterizado pela busca por tesouros, bem como minimizando fortemente os riscos.

6 Conclusão

Esse Trabalho de Conclusão de Curso apresentou um algoritmo evolutivo adaptado para um jogo baseado em turnos, comparando-o com as abordagens de aprendizado por reforço.

Comparando o algoritmo evolutivo com os algoritmos de aprendizado por reforço, o AE permite a manifestação de comportamentos mais complexos pelos agentes, podendo manifestar diferentes conjuntos de regras e jogadas dentro de um mesmo tipo de jogador, representado por uma função de *fitness*. Por outro lado, os algoritmos que utilizam Aprendizado de Máquina apresentaram resultados superiores na maioria dos casos, considerando apenas o objetivo específico do agente.

A versatilidade do AE é um ponto muito vantajoso pois jogadores do mundo real não podem, na prática, ser simplificados em apenas 5 grupos. Podem existir jogadores que reunem características de mais de um dos grupos e jogadores que, embora seriam considerados do mesmo grupo, possuem comportamentos diferentes dentro do jogo.

Analisando o aspecto técnico do algoritmo utilizado, o AE possui, em relação ao *Q-Learning*, vantagens em relação ao tempo de execução e facilidade de implementação, não consumindo tanta memória.

Comparando os resultados, o AE possui resultados mais flexíveis e que podem ser generalizados entre um mapa e outro, algo que não acontece no *Q-Learning*. Além disso, o algoritmo evolutivo pode ser expandido, podendo ser utilizado hibridizando com outros algoritmos de IA e pode ser aprimorado com técnicas multiobjetivo, fazendo com que o AE utilize mais de uma função objetivo. Na prática, isso significa trabalhar explicitamente com agentes que possuem comportamentos que combinam dois ou mais grupos de jogadores.

Por outro lado, os algoritmos de aprendizado por reforço (*Q-Learning* e *HAQL*) foram mais efetivos em relação ao objetivo específico de cada agente, de

modo geral, confirmando os resultados mostrados na literatura. A desvantagem é que são métodos lentos, como foi relatado nos experimentos, e menos realistas, em relação a modelagem do comportamento de um jogador real.

Neste trabalho, notou-se possibilidades de estudos futuros.

Uma possibilidade é investigar as diferentes manifestações de conjuntos de regras que o algoritmo pode gerar dentro do mesmo tipo de jogador. Ainda sobre o conjunto de regras, pode-se estudar cenários em que o AE evolui regras por alguns mapas e aplica as mesmas regras em outros mapas, verificando o resultado obtido.

Outra possibilidade é incrementar o AE implementado fazendo-o capaz de lidar com um *fitness* multiobjetivo, trabalhando com agentes híbridos. Para o *Q-Learning*, pode-se estudar outras técnicas e melhorias, como o *Deep Learning* para tentar encontrar resultados satisfatórios em resultado e tempo para esse tipo de problema.

Por fim, uma outra opção é aplicar o algoritmo, adaptando o conjunto de regras utilizado, em outros jogos baseado em turnos para verificar se o AE proposto possui um funcionamento adequado em jogos diferentes e mais complexos.

Referências

ALPAYDIN, E. *Introduction to Machine Learning*. 2nd. ed. [S.l.]: The MIT Press, 2010. ISBN 026201243X, 9780262012430. Citado na página 17.

APPERLEY, T. Genre and game studies: Toward a critical approach to video games genres. *Simulation & Gaming*, 2006. Citado na página 30.

BARBOSA, C. S. *Algoritmos Baseados em Estrategia Evolutiva para a Selecao Dinamica de Espectro em Radios Cognitivos*. Dissertação (Mestrado) — Universidade Federal de Goias, Goiania, 2013. Citado 3 vezes nas páginas 19, 20 e 22.

BIANCHI, R. A. C.; RIBEIRO, C. H. C.; COSTA, A. H. R. Heuristically accelerated q–learning: A new approach to speed up reinforcement learning. In: *Advances in Artificial Intelligence – SBIA 2004*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 245–254. ISBN 978-3-540-28645-5. Citado na página 48.

BIANCHI, R. A. da C. *Uso de Heurísticas para a Aceleração do Aprendizado por Reforço*. Tese (Doutorado) — Universidade de São Paulo, São Paulo, 2004. Citado 2 vezes nas páginas 23 e 48.

CROCOMO, K. M. *Um Algoritmo Evolutivo para Aprendizado On-line em jogos Eletrônicos*. Dissertação (Mestrado) — Universidade Estadual de São Paulo, São Paulo, 2008. Citado 5 vezes nas páginas 10, 34, 35, 42 e 44.

DARWIN, C. R. *On the Origin of Species*. [S.l.]: J. Murray, 1859. Citado 2 vezes nas páginas 9 e 13.

Entertainment Software Association. *Essential Facts About the Computer and Video Game Industry*. 2015. Acessado em 10 de Julho de 2018. Disponível em: <http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf>. Citado na página 9.

FAIRCLOUGH, C.; et al. Research directions for ai in computer games. In: *Proceedings of the 12 Irish Conference on AI and Cognitive Science*. [S.l.]: AICS, 2001. p. 1–12. Citado na página 9.

FURNKRANZ, J. Machine learning in games: A survey. Austrian Research Institute for Artificial Intelligence, 2004. Citado na página 25.

GABRIEL, P.; DELBEM, A. *Fundamentos de algoritmos evolutivos*. [S.l.], 2008. Citado 6 vezes nas páginas 4, 13, 14, 15, 16 e 44.

GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201157675. Citado 2 vezes nas páginas 9 e 13.

HASSELT, H. van; GUEZ, A.; SILVER, D. Deep reinforcement learning with double q-learning. In: *AAAI Conference on Artificial Inteligence*. [S.l.: s.n.], 2016. p. 2094–2100. Citado na página 48.

HOLLAND, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992. ISBN 0262082136. Citado 2 vezes nas páginas 9 e 13.

HOLMGÅRD, C. et al. Monte-carlo tree search for persona based player modeling. In: . [S.l.: s.n.], 2015. Citado 2 vezes nas páginas 11 e 38.

HOLMGÅRD, C.; LIAPIS, A.; TOGELIUS, J. *Minidungeons*. 2014. Acessado em 10 de Julho de 2018. Disponível em: <http://minidungeons.com>. Citado 2 vezes nas páginas 11 e 31.

HOLMGÅRD, C. et al. Evolving personas for player decision modeling. In: *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG))*. [S.l.: s.n.], 2014. Citado 10 vezes nas páginas 11, 33, 34, 37, 45, 48, 58, 59, 60 e 61.

HOLMGÅRD, C. et al. Generative agents for player decision modeling in games. *Foundations of Digital Games*, 2014. Citado 15 vezes nas páginas 10, 11, 18, 30, 31, 32, 34, 35, 46, 47, 48, 58, 59, 60 e 61.

HOLMGÅRD, C. et al. *MiniDungeons 2 - An Experimental Game for Capturing and Modeling Player Decisions*. 2014. Citado 4 vezes nas páginas 4, 30, 33 e 34.

MARTINS, M. F.; BIANCHI, R. Comparação de desempenho de algoritmos de aprendizado por reforço no domínio do futebol de robôs. 07 2018. Citado na página 48.

MENEZES, P. B. *Linguagens Formais e Autômatos*. [S.l.: s.n.], 2000. Citado na página 13.

MILLINGTON, I.; FUNGE, J. (Ed.). *Artificial Intelligence for Games*. [S.l.]: CRC Press, 2009. Citado 7 vezes nas páginas 4, 25, 26, 27, 28, 29 e 30.

REZENDE, S. *Sistemas inteligentes: fundamentos e aplicações*. [S.l.]: Manole, 2003. ISBN 9788520416839. Citado 2 vezes nas páginas 17 e 29.

RUSSEL, S.; PETER, N. (Ed.). *Artificial Intelligence, A Modern Approach*. [S.l.]: Pearson, 2010. Citado na página 16.

SAMUEL, A. L. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, IBM Corp., Riverton, NJ, USA, v. 3, n. 3, p. 210–229, jul. 1959. ISSN 0018-8646. Citado na página 17.

SILVA, M. M. *Uma Abordagem Evolucionaria Para o Aprendizado Semi-Supervisionado em Maquinas de Vetores de Suporte*. Dissertação (Mestrado) — Universidade Federal de Minas Gerais, Belo Horizonte, 2008. Citado 3 vezes nas páginas 17, 18 e 26.

SMITH, A.; BRYSON, J. A logical approach to building dungeons: Answer set programming for hierarchical procedural content generation in roguelike games. 01 2014. Citado na página 30.

SPRONCK, P.; SPRINKHUIZEN-KUYPER, I.; POSTMA, E. Online adaptation of game opponent ai in theory and practice. In: *International Conference On Intelligent Games and Simulation*. [S.l.: s.n.], 2004. p. 45–53. Citado na página 18.

SUTTON, R.; BARTO, A. *Reinforcement Learning: An Introduction*. [S.l.]: MIT Press, 2012. Citado 5 vezes nas páginas 18, 19, 20, 22 e 23.

SWEETSER, P. Current ai in games: A review. Brisbane: University of Queensland, 2002. Citado na página 9.

WATKINS, C. J. C. H.; DAYAN, P. Q-learning. *Machine Learning*, v. 8, n. 3, p. 279–292, May 1992. ISSN 1573-0565. Citado 3 vezes nas páginas 18, 22 e 30.

WHITEHEAD, S. D. A complexity analysis of cooperative mechanisms in reinforcement learning. In: *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 2*. [S.l.]: AAAI Press, 1991. (AAAI'91), p. 607–613. ISBN 0-262-51059-6. Citado na página 48.

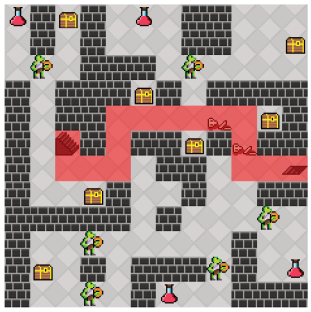
YANNAKAKIS, G. N. et al. Player modeling. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. *Dagstuhl Follow-Ups*. [S.l.], 2013. v. 6. Citado na página 10.

YANNAKAKIS, G. N.; TOGELIUS, J. A panorama of artificial and computational intelligence in games. v. 7, p. 1–1, 01 2014. Citado 2 vezes nas páginas 9 e 10.

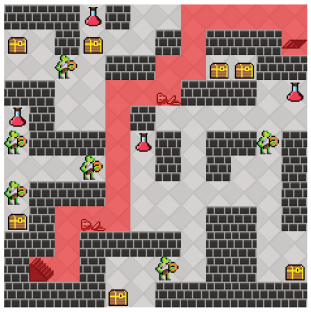
ZADEH, L. Fuzzy sets. *Information and Control*, v. 8, n. 3, p. 338 – 353, 1965. ISSN 0019-9958. Citado na página 26.

Apêndices

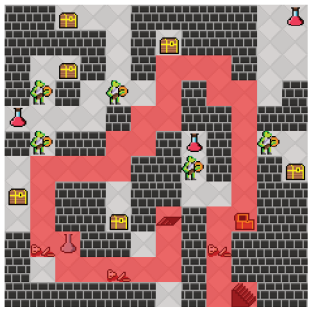
APÊNDICE A – Execuções do Algoritmo Evolutivo



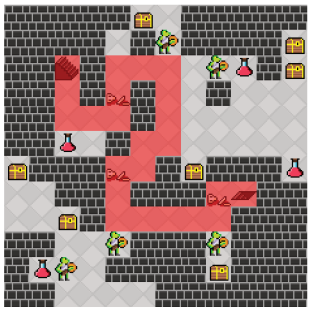
(a) Mapa 1



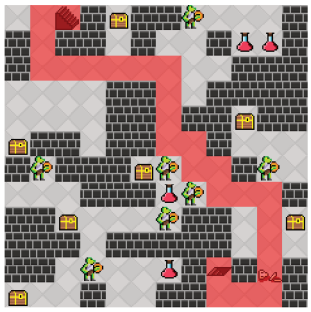
(b) Mapa 2



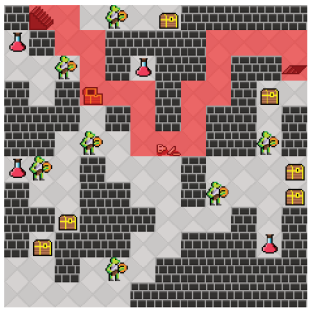
(c) Mapa 3



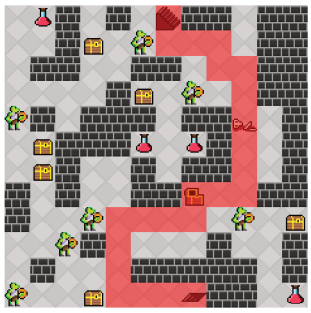
(d) Mapa 4



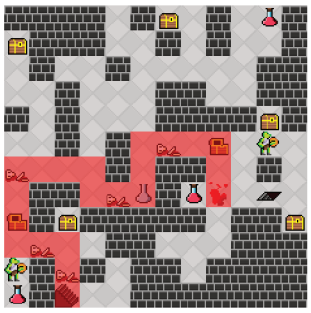
(e) Mapa 5



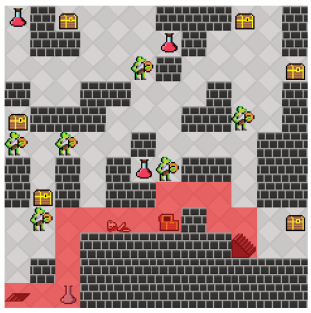
(f) Mapa 6



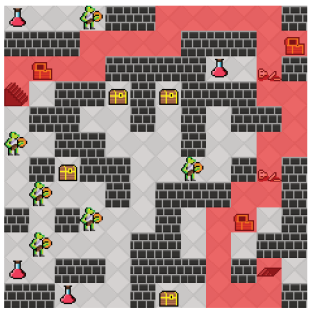
(g) Mapa 7



(h) Mapa 8

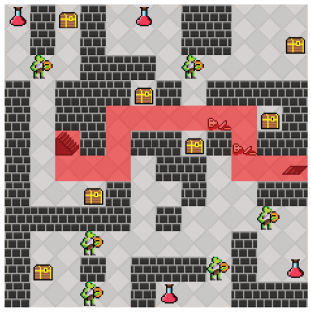


(i) Mapa 9

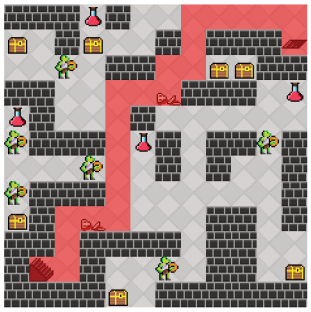


(j) Mapa 10

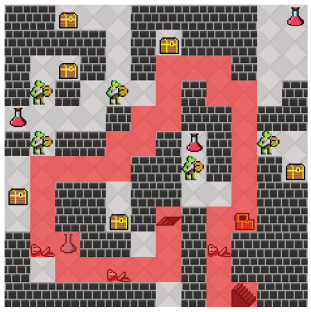
Figura 10 – Agente *Baseline*



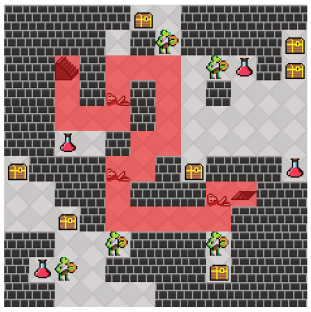
(a) Mapa 1



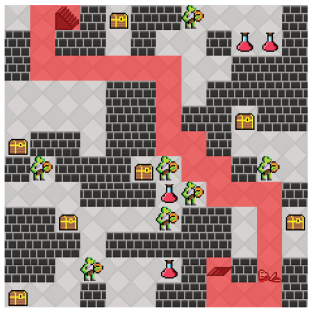
(b) Mapa 2



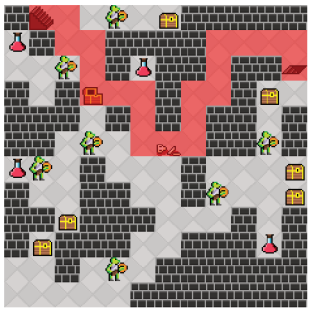
(c) Mapa 3



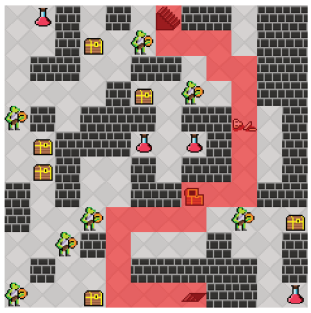
(d) Mapa 4



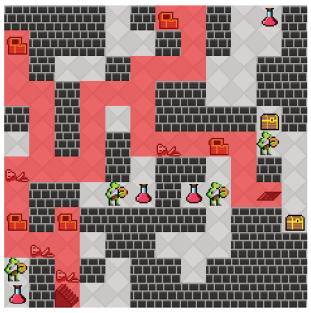
(e) Mapa 5



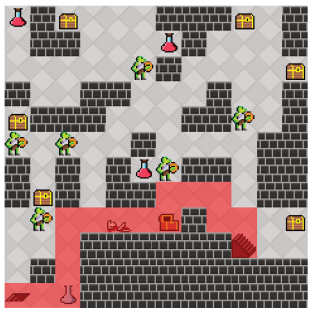
(f) Mapa 6



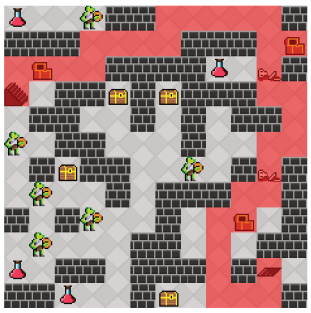
(g) Mapa 7



(h) Mapa 8

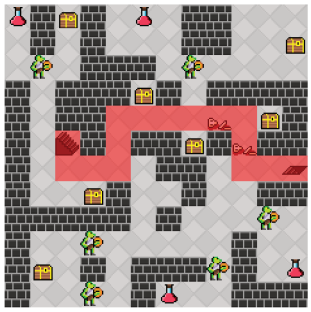


(i) Mapa 9

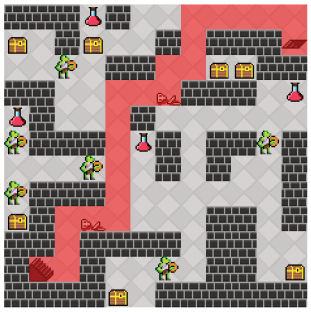


(j) Mapa 10

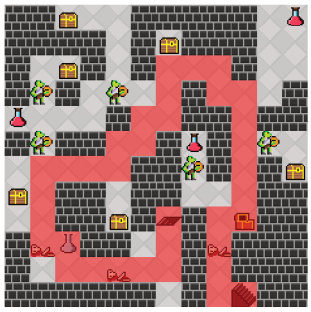
Figura 11 – Agente *Runner*



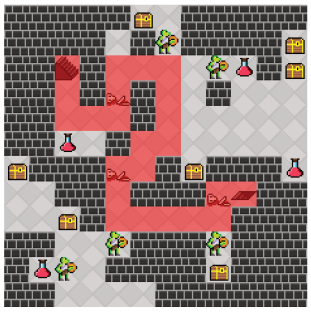
(a) Mapa 1



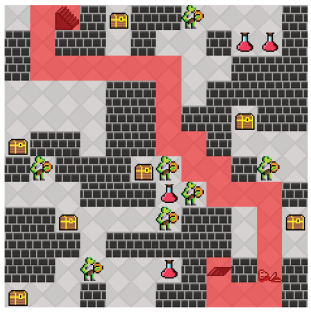
(b) Mapa 2



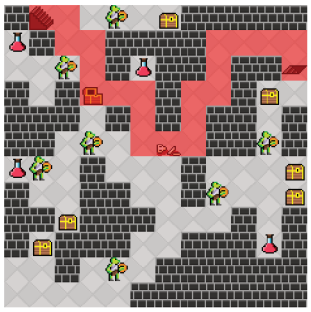
(c) Mapa 3



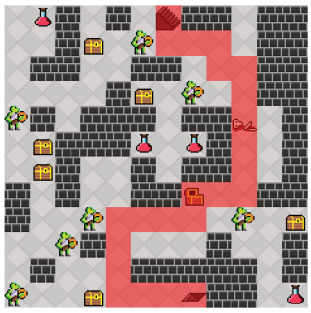
(d) Mapa 4



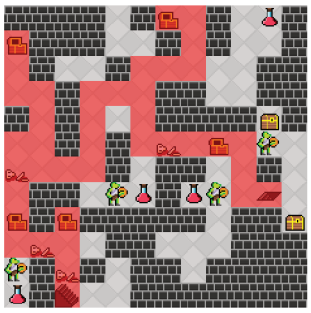
(e) Mapa 5



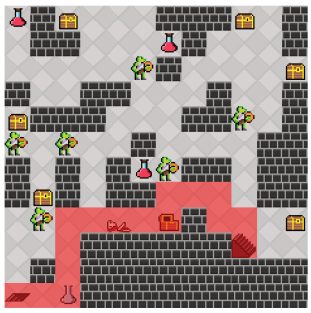
(f) Mapa 6



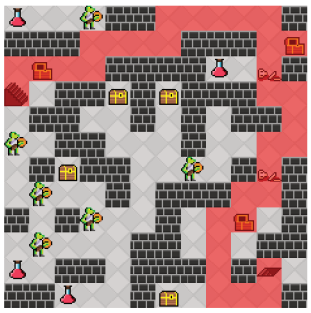
(g) Mapa 7



(h) Mapa 8

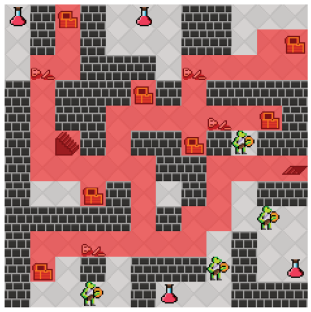


(i) Mapa 9

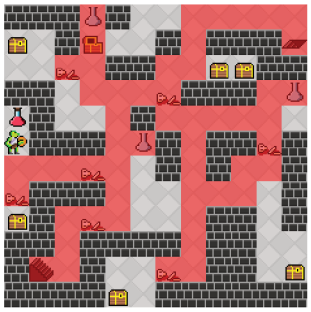


(j) Mapa 10

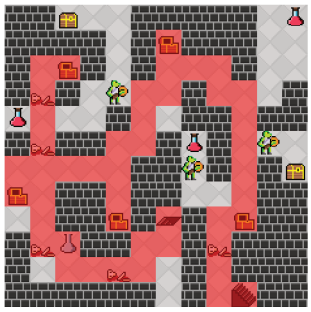
Figura 12 – Agente *Survivalist*



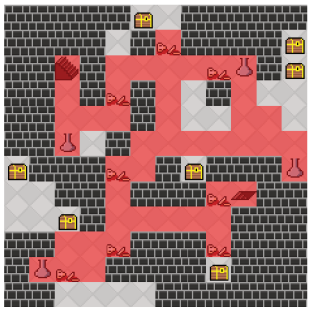
(a) Mapa 1



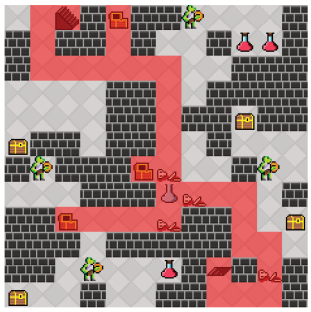
(b) Mapa 2



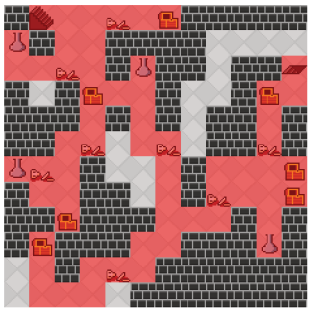
(c) Mapa 3



(d) Mapa 4

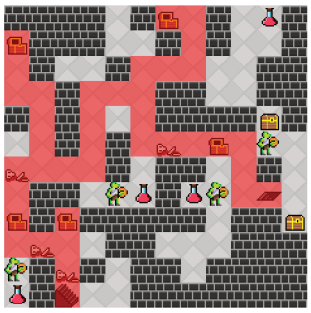


(e) Mapa 5

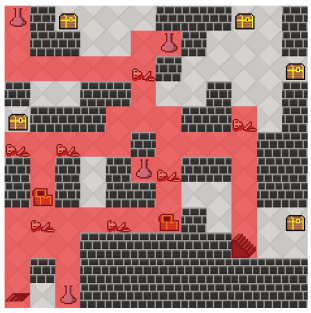


(f) Mapa 6

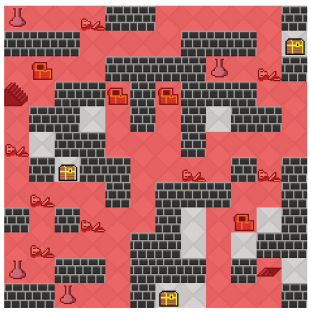
(g) Mapa 7



(h) Mapa 8

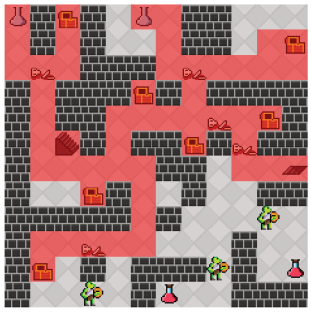


(i) Mapa 9

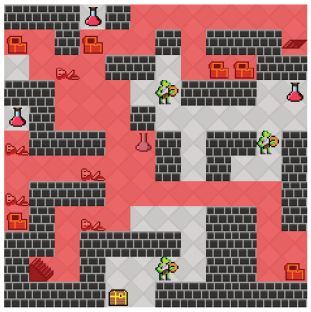


(j) Mapa 10

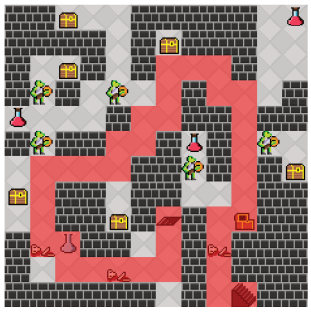
Figura 13 – Agente *Monster Killer*



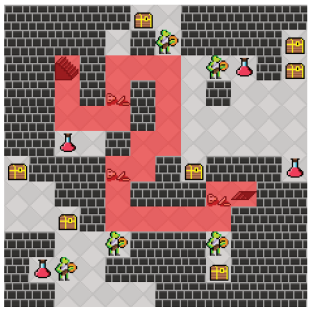
(a) Mapa 1



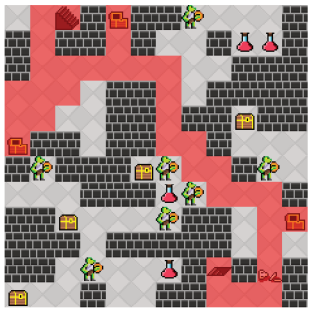
(b) Mapa 2



(c) Mapa 3

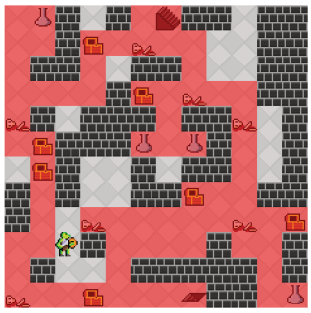


(d) Mapa 4

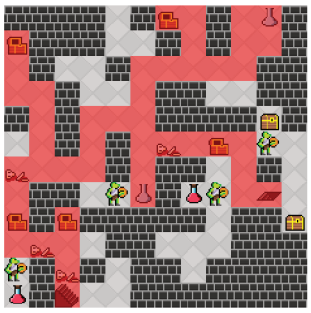


(e) Mapa 5

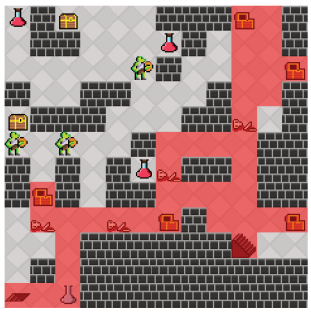
(f) Mapa 6



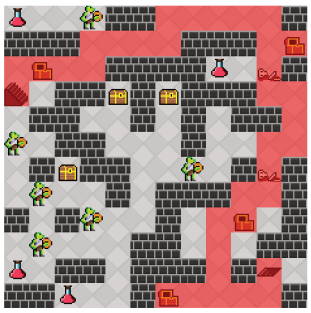
(g) Mapa 7



(h) Mapa 8



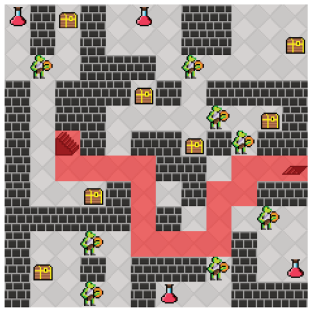
(i) Mapa 9



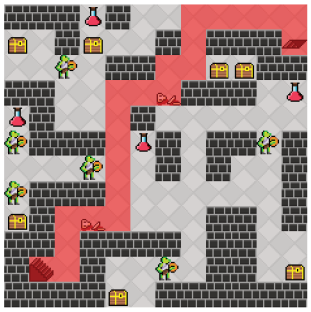
(j) Mapa 10

Figura 14 – Agente *Treasure Collector*

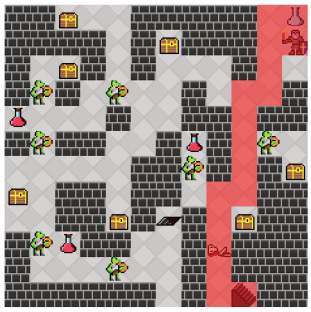
APÊNDICE B – Execuções do *HAQL*



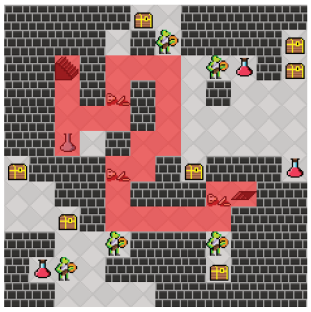
(a) Mapa 1



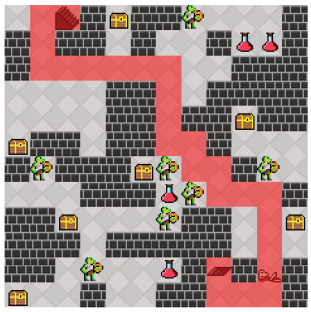
(b) Mapa 2



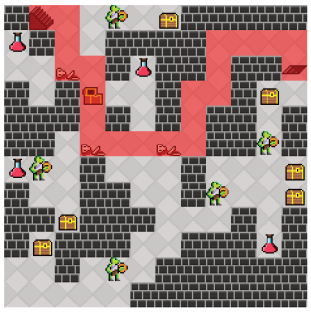
(c) Mapa 3



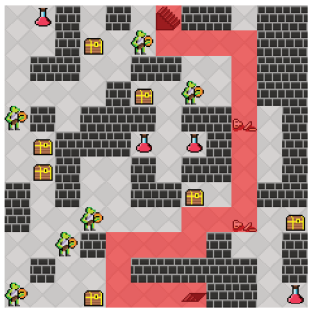
(d) Mapa 4



(e) Mapa 5

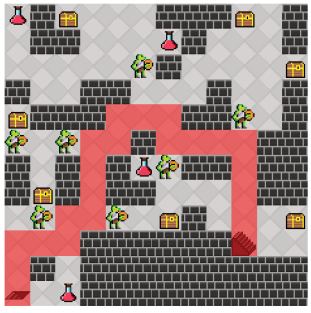


(f) Mapa 6

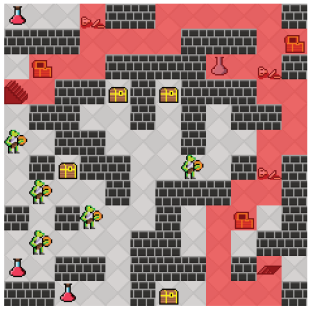


(g) Mapa 7

(h) Mapa 8

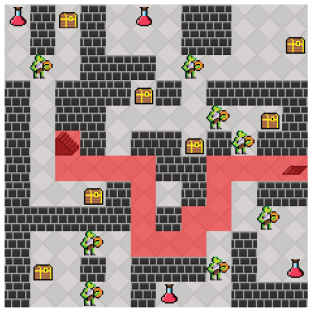


(i) Mapa 9

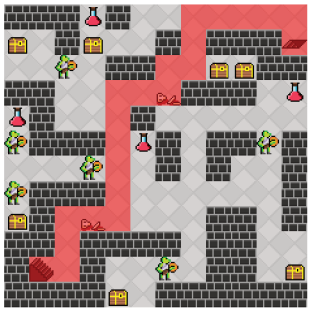


(j) Mapa 10

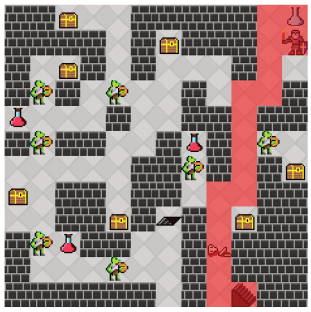
Figura 15 – Agente *Baseline*



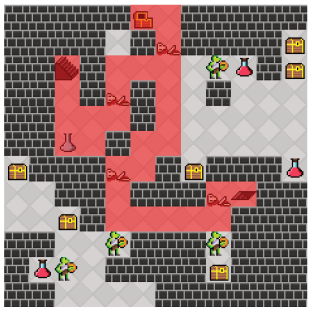
(a) Mapa 1



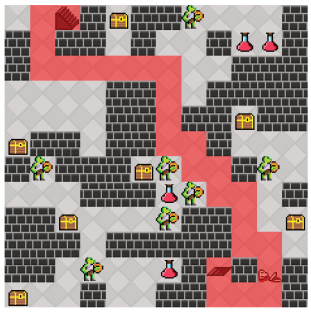
(b) Mapa 2



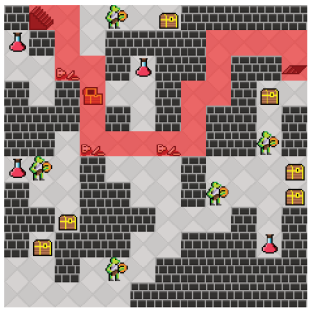
(c) Mapa 3



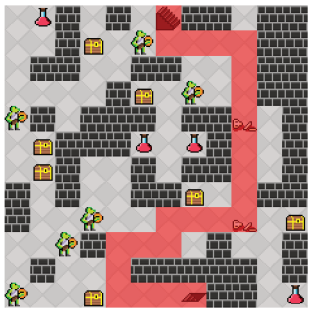
(d) Mapa 4



(e) Mapa 5

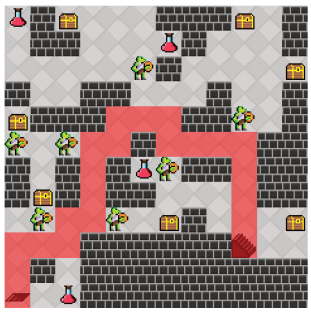


(f) Mapa 6



(g) Mapa 7

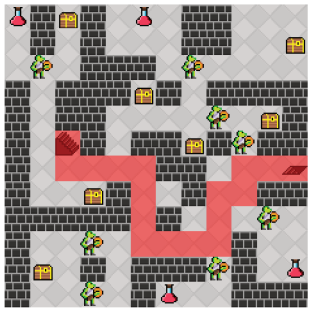
(h) Mapa 8



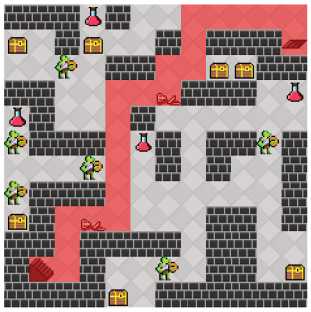
(i) Mapa 9

(j) Mapa 10

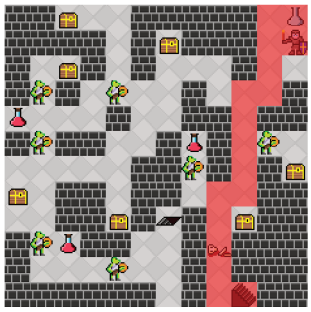
Figura 16 – Agente *Runner*



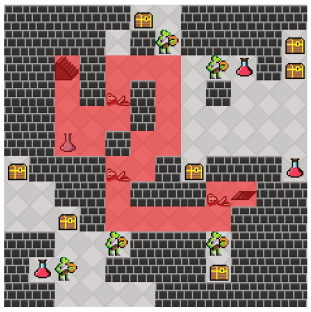
(a) Mapa 1



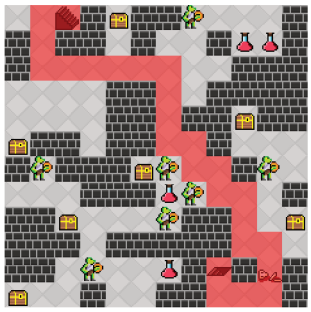
(b) Mapa 2



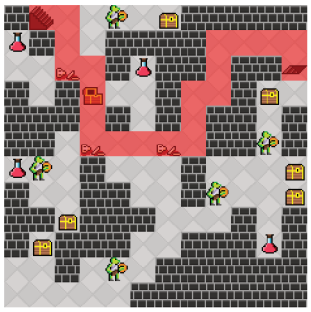
(c) Mapa 3



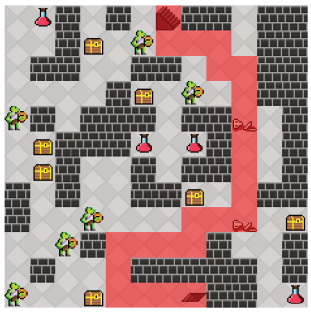
(d) Mapa 4



(e) Mapa 5

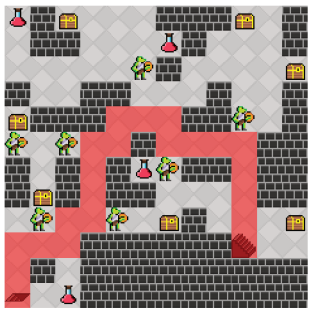


(f) Mapa 6

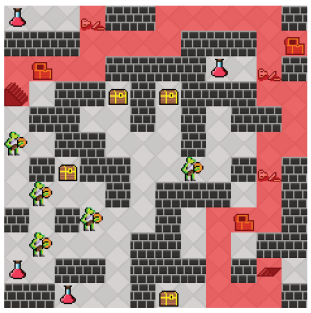


(g) Mapa 7

(h) Mapa 8

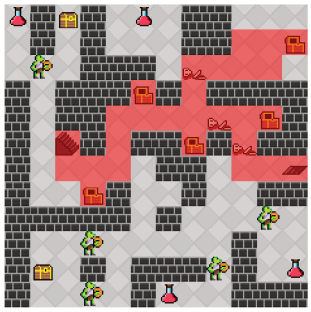


(i) Mapa 9

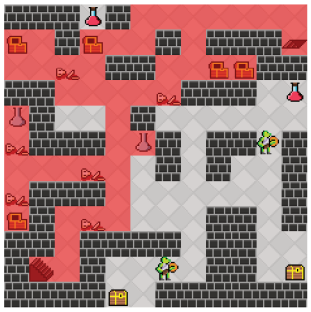


(j) Mapa 10

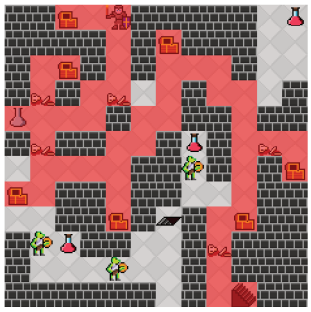
Figura 17 – Agente *Survivalist*



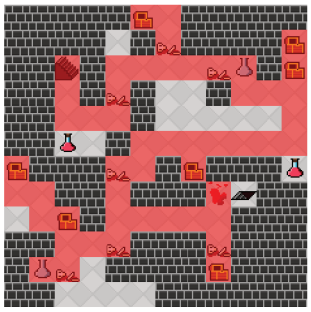
(a) Mapa 1



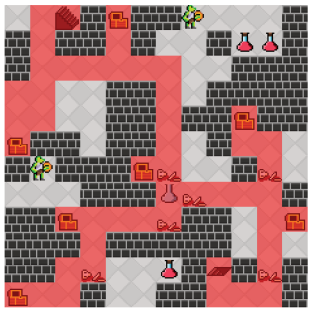
(b) Mapa 2



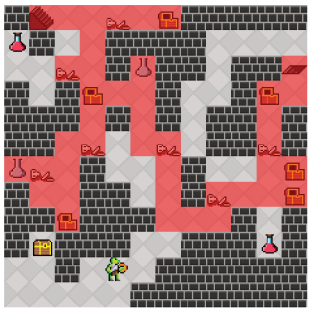
(c) Mapa 3



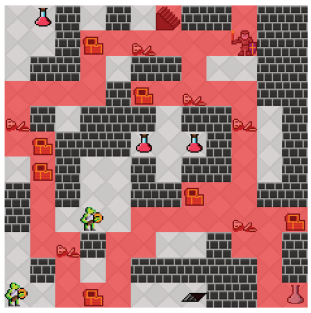
(d) Mapa 4



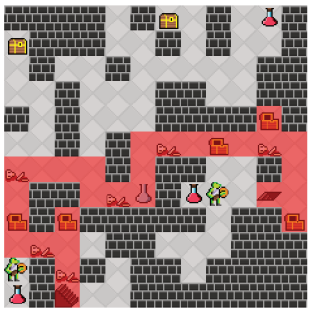
(e) Mapa 5



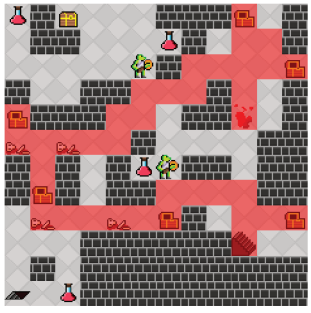
(f) Mapa 6



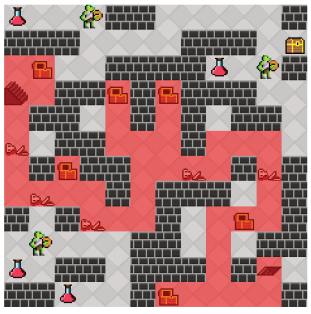
(g) Mapa 7



(h) Mapa 8

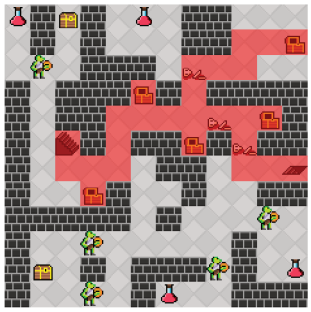


(i) Mapa 9

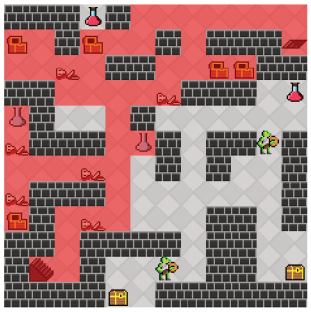


(j) Mapa 10

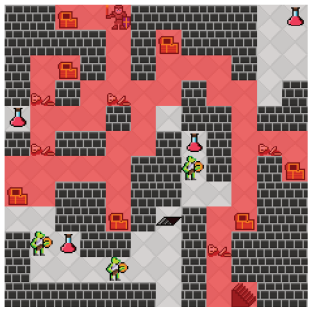
Figura 18 – Agente *Monster Killer*



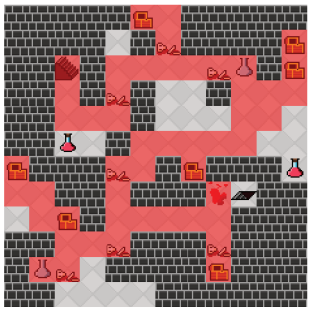
(a) Mapa 1



(b) Mapa 2

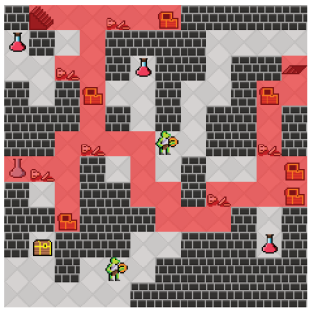


(c) Mapa 3

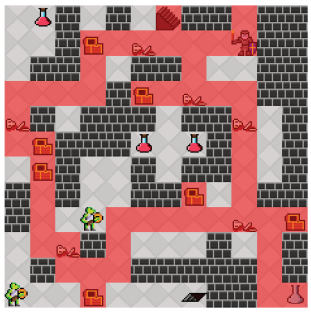


(d) Mapa 4

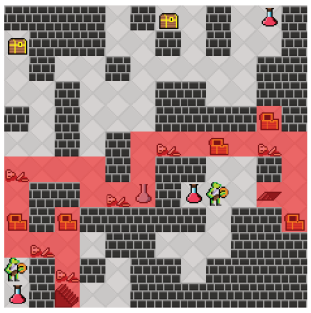
(e) Mapa 5



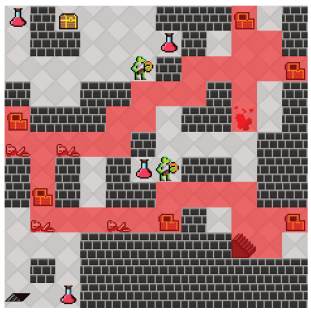
(f) Mapa 6



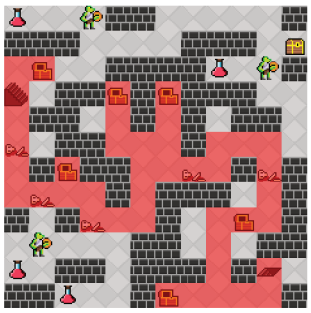
(g) Mapa 7



(h) Mapa 8



(i) Mapa 9



(j) Mapa 10

Figura 19 – Agente *Treasure Collector*