UNIVERSIDADE DE SOROCABA
PRÓ-REITORIA ACADÊMICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Danilo de Lucas Moraes Dias

APRENDIZADO DE MÁQUINA APLICADO Á RESOLUÇÃO DE PROBLEMAS DE DECISÃO

Sorocaba
2016

Danilo de Lucas Moraes Dias

APRENDIZADO DE MÁQUINA APLICADO Á RESOLUÇÃO DE PROBLEMAS DE DECISÃO

Trabalho de conclusão do curso de graduação apresentado na Universidade de Sorocaba como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Fernando Cesar Miranda

Sorocaba/SP

2016

RESUMO

Os jogos eletrônicos sempre estiveram evoluindo no quesito gráfico, apresentando cada vez mais detalhes em texturas, luzes e sombras mais realistas, além de objetos com detalhes geométricos fotorrealistas. Esta evolução também se estendeu à complexidade dos sistemas que compõem um jogo, trazendo experiências mais imersivas aos jogadores como, por exemplo, inimigos que aprendem o padrão do jogador e se tornam mais difíceis de serem derrotados. Por conta disso, a evolução nos jogos propiciou que fossem desenvolvidas técnicas para aperfeiçoar a implementação de algoritmos inteligentes neste contexto. Trabalhos na literatura demonstraram avanço no desenvolvimento de algoritmos capazes de adaptar suas ações através da utilização de métodos como programação genética e redes neurais, os quais demandam uma quantidade significativa de tempo para serem processados e efetivamente gerar uma resposta em tempo hábil ao sistema. Para contornar essa dificuldade, este trabalho adota a estratégia de combinar métodos de análise e otimização combinatória e classificação através de algoritmos capazes de desenvolver conhecimento a respeito do contexto de um problema e gerar soluções em tempo de execução. Os resultados apresentados foram validados estatisticamente e indicaram que o método proposto obteve tempo de processamento inferior aos métodos que utilizam do treinamento de redes neurais para a resolução de problemas.

Palavras-chaves: Inteligência Artificial; Análise combinatória; Otimização combinatória; Classificação; Aprendizado de Máquina; Jogos eletrônicos.

ABSTRACT

The Games have always been evolving…

Key-words: artificial intelligence; classification; machine learning; games.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

IA	Inteligência Artificial
NPC	Non-player Character
NEAT	Evolving Neural Networks through Augmenting Topologies
SMW	Super Mario World
RNA	Redes Neurais Artificiais
SNES	Super Nintendo Entertainment System
TAS	Tool Assisted Speedrun

SUMÁRIO

1 INTRODUÇÃO

A medida que o processamento, os gráficos e o realismo dos jogos aumentam, a exigência dos jogadores por uma experiência mais imersiva também cresce. Estes fatores, criaram a necessidade de que os elementos controlados pelo computador, tais como obstáculos e NPC's (*non-player characters,* personagens não controlados pelo jogador), tenham não somente reações a interações do jogador, mas também aprendam com o decorrer do jogo para adaptar suas ações. Alguns jogos possuem mecanismos que comportam a utilização de métodos de aprendizado para a implementação de inteligências artificiais (IA), o que indica que elas não precisam ser programadas manualmente (CHAMPANDARD, 2004). Partindo deste princípio, os desenvolvedores buscam criar técnicas e métodos para a implementação de algoritmos que se moldem utilizando os dados gerados no decurso do jogo.

Trabalhos na literatura indicam ser promissora a aplicação de métodos de aprendizado de máquina para a implementação de inteligências artificiais em jogos (STANLEY, 2002; MIIKKULAINEN, 2002). Estes métodos geram dados que são utilizados para o treinamento de redes neurais[1], que posteriormente serão utilizadas para a resolução dos problemas. Contudo, algumas destas implementações demonstram-se lentas, pois utilizam técnicas como programação genética[2] para otimização de soluções, sendo necessárias várias execuções para que uma solução que atenda ao problema seja alcançada.

Em um jogo, um problema pode ser considerado como uma situação que é apresentada ao jogador ou ao computador, onde as variáveis são os atributos dos inimigos e obstáculos, e o contexto é a relação entre as variáveis e o elemento controlado pelo jogador ou computador. Quando se tem um problema que não é conhecido, inferir dados sobre o contexto pode ser uma boa forma de chegar a uma solução para o problema. Para obter estes dados podem ser utilizados mecanismos de busca, que farão a varredura do contexto do problema a procura de soluções, gerando dados sobre o mesmo. Contudo, independente da implementação, seria necessário gerar dados específicos para cada problema proposto, o que não é o propósito de uma IA de acordo com Tanimoto (TANIMOTO, 1987). Portanto é necessária uma forma de generalizar soluções para que as mesmas possam ser aplicadas a outros problemas parecidos, sem que necessariamente seja preciso refazer a busca e gerar novos dados. Uma forma de conseguir essa

[1] Redes neurais artificiais são modelos computacionais que abstraem o funcionamento do sistema nervoso central de um animal.
[2] Programação genética é uma técnica de programação autônoma que aplica princípios da evolução biológica para manipular soluções.

generalização é por meio da aplicação de técnicas de aprendizagem por reforço, que irão utilizar os dados gerados por uma análise combinatória dos possíveis comandos para o seu treinamento, e com isso generalizar soluções para os problemas propostos, utilizando do *feedback* das soluções para otimizar a atribuição de resultados.

Para a busca por soluções e geração de dados destaca-se a utilização de algoritmos recursivos que utilizam processos heurísticos para estimar a procedência positiva ou negativa de uma iteração (GHADERI, 2009). Trata-se de um método de refinamento de busca por força bruta, que faz a varredura do contexto do problema a procura de soluções, utilizando heurísticas para ignorar soluções errôneas e com isso gerar dados específicos sobre os problemas solucionados. A implementação é melhor aproveitada quando utilizada em conjunto com métodos de otimização combinatória para otimizar a atribuição de soluções, que seja capaz de ordenar os dados gerados e priorizar as melhores soluções para o contexto geral da situação. A generalização de soluções demonstra-se eficiente com a utilização de métodos de classificação que consomem dados para o treinamento de árvores de decisão (FUSSELL, 2012). A árvore gerada é ajustada mediante aos valores fornecidos em sua instancia, classificando da melhor forma possível uma dada observação, com base nos valores disponíveis na base durante a geração da árvore.

Nesse cenário, onde é preciso que para um dado problema seja encontrada uma solução, e posteriormente possa ser generalizada para outros problemas, a utilização dos métodos de análise combinatória e classificação, em conjunto, mostra-se viável. Com a aplicação do método de análise combinatória, é possível gerar uma base de dados com soluções especificas para problemas resolvidos. A partir desta base, o método de classificação possibilita a generalização das soluções para problema não tratados, gerando a melhor classificação possível para problemas semelhantes e possíveis boas soluções para problemas muito diferentes. Uma otimização pode ser feita nos resultados gerados pela analise combinatória, de forma a melhorar a atribuição de soluções para problemas nunca antes tratados e aprimorar a classificação das demais situações.

Objetivos e Contribuições

O objetivo deste trabalho é apresentar técnicas para a implementação de inteligência artificial em jogos por meio da geração e manipulação de bases de conhecimento, através da utilização de métodos de classificação e otimização combinatória.

Dentre as contribuições oferecidas neste trabalho, destacam-se:

1. Demonstração da utilização de uma base de dados gerada por um método de análise e otimização combinatória para o treinamento de árvores de decisão;
2. Criação de uma biblioteca para o auxílio na implementação de algoritmos de aprendizado de máquina;
3. Geração de bases de conhecimento que podem ser utilizadas para treinamento de outros algoritmos de classificação.

Organização

Este manuscrito apresenta a seguinte estrutura:

- No Capítulo 2, é introduzida a utilização de inteligência artificial em jogos e os principais trabalhos encontrados na literatura.
- No Capítulo 3, são introduzidos os conceitos de aprendizado de máquina e descritos os métodos de classificação envolvidos no trabalho.
- No Capítulo 4, é apresentado o algoritmo resultante das pesquisas realizadas neste trabalho, assim como a biblioteca desenvolvida para facilitar a implementação do mesmo.
- No Capítulo 5 são apresentados os experimentos realizados e os resultados obtidos pela pesquisa.
- Finalmente, no Capítulo 6, são dadas as conclusões e orientações para trabalhos futuros.

2 INTELIGÊNCIA ARTIFICIAL EM JOGOS ELETRÔNICOS

Inteligência artificial pode ser definida como a capacidade de tornar computadores aptos a executar tarefas que atualmente somente seres humanos e animais são capazes de realizar (MILLINGTON, 2006). É utilizada nos jogos eletrônicos com o objetivo de permitir uma interação mais dinâmica e passar a sensação de pervasividade ao jogador, na tentativa de fazê-lo acreditar que a máquina está de fato pensando e tentando derrotá-lo.

De acordo com Rich (1988), os primeiros problemas de IA a serem resolvidos estavam relacionados a problemas encontrados em jogos e provas de teoremas. Portando, a IA e os jogos sempre estiveram relacionados, e a evolução de um contribui muito com desenvolvimento do outro.

O uso da inteligência artificial é muito vasto dentro do contexto dos jogos eletrônicos (KISHIMOTO, 2004). Sua aplicação vai desde a atribuição de comportamento a NPCs (personagens não controlados pelo jogador), até na modificação da dificuldade do jogo com base na habilidade do jogador. Para tal, são utilizadas diversas técnicas de IA, muitas vezes em conjunto, para a obtenção de resultados cada vez mais interessantes para os desenvolvedores. Consequentemente, permitindo implementações mais complexas e muito mais imersivas para os jogadores.

Trabalhos na literatura demonstram a aplicação de diversos métodos de inteligência artificial para a implementação de algoritmos capazes de realizar tarefas nunca antes imaginadas, como é o caso do algoritmo denominado *MarI/O*, que será tratado brevemente na seção 2.1.

2.1 MarI/O

MarI/O é um algoritmo capaz de passar fases do jogo Super Mario World, desenvolvido por um usuário do site *YouTube* chamado SethBling[3]. No site é possível encontrar um vídeo com o nome *MarI/O - Machine Learning for Video Game*[4]. Nele é demonstrada a aplicação do algoritmo no jogo e é dada uma explicação das técnicas utilizadas e da implementação.

O programa utiliza uma técnica denominada NEAT (*Evolving Neural Networks through Augmenting Topologies)*, que segundo Stanley e Miikkulainen (2002), busca solucionar os principais problemas encontrados durante a utilização de algoritmos genéticos para a

[3] Canal do usuário SethBling no site *YouTube.* Disponível em: <https://goo.gl/enVf6o>. Acessado em 17 de março de 2015.
[4] Video *MarI/O - Machine Learning for Video Game* no site *YouTube*. Disponível em: <https://goo.gl/IAvXGy>. Acessado em 13 de junho de 2015.

otimização da estrutura topológica de uma rede neural artificial. Basicamente, o programa é capaz de passar as fases do jogo que são apresentadas a ele, sem nenhum conhecimento prévio das mesmas e nem mesmo do jogo em si. Inicialmente o computador não conhece nenhuma das características do jogo nem das interações que podem ser realizadas por ele.

O algoritmo utiliza o NEAT para realizar o treinamento de redes neurais artificiais com base nas informações que possui, que são apenas as informações de quais botões podem ser pressionados para realizar ações no jogo e as informações a respeito do ambiente ao redor do personagem. O treinamento é feito por meio de uma série de gerações, onde cada uma das combinações escolhidas pelo computador é testada e as melhores soluções são escolhidas por um processo de seleção. Este processo utiliza uma técnica de programação genética para selecionar os melhores conjuntos de soluções com base na qualidade dos resultados obtidos por eles quando aplicados no jogo. Ele seleciona os que obtiveram os melhores resultados e gera uma mutação em alguns dos elementos deste conjunto. Uma nova série de gerações é realizada e novamente são recolhidas as melhores soluções resultantes do processo. As anomalias resultantes da mutação que fizeram com que a solução se demonstrasse melhor em relação as outras, prevalece nas demais gerações até que outra solução com um resultado melhor ocupe o seu lugar. Desta forma o processo tende a apresentar boas soluções após um grande número de gerações.

A utilização do NEAT para este propósito apresenta uma solução satisfatória, se tratando da precisão dos resultados. Contudo, o tempo necessário para treinar uma rede neural capaz de passar uma fase por completo é muito alto, podendo chegar a dias de processamento para alcançar uma solução sub-ótima.

2.2 Imersão nos Jogos Eletrônicos

Um jogo é considerado muito imersivo quando o jogador é capaz de esquecer o que se passa no ambiente ao seu redor e concentra toda sua atenção no jogo (FERREIRA; OLIVEIRA, 2011). O meio de interação não apresentar grandes obstáculos a forma natural do ser humano fazer as coisas pode ajudar no processo de imersão. Sem levar em consideração a forma como o jogador irá interagir com o jogo, seja por meio de controles ou mesmo a captação de seus movimentos, o tipo de resposta audiovisual que ele deve receber deve se assemelhar o máximo possível ao que ele está acostumado a encontrar no mundo real. Fazendo com que o tempo de resposta para uma interação seja muito importante.

Dependendo do tipo de IA implementada no jogo, o tempo de resposta elevado pode arruinar a imersão do jogador. Por exemplo, em um jogo de combate onde o jogador precisa derrotar o inimigo para vencer, se o adversário for controlado pelo computador, é esperado que a IA do personagem seja capaz de revidar aos ataques do jogador e tentar vencê-lo, respeitando o nível de habilidade do jogador. Contudo, se o jogador for experiente e o computador não se apresentar um desafio para ele, a IA deve ser capaz de se ajustar rapidamente a habilidade do adversário para adequar-se ao nível dele. Se este processo de ajuste demorar para ser realizado, o jogador perderá a imersão, tanto pelo fato de seu adversário não lhe apresentar um desafio, quanto pela perceptível mudança gradual no nível do personagem enfrentado.

TODO Igual eu comentei, aqui poderiam ser dado mais exemplos, incluindo falando de jogos e tal. Enriquece bastante e dá corpo para o capítulo.

3 Métodos de Classificação

Os métodos de classificação exercem um importante papel no mercado de tomada de decisões, classificando as informações disponíveis utilizando algum critério para tal (KIANG, 2002). A implementação de inteligências artificiais que precisam evoluir e aprender durante a execução do programa normalmente é feita através de métodos de aprendizado de máquina. Geralmente o campo do aprendizado de máquina é dividido em três paradigmas: aprendizagem supervisionada, aprendizagem não-supervisionada e aprendizagem por reforço (RUSSEL; NORVIN, 2003).

- Aprendizagem supervisionada: o programa recebe amostras de entradas e suas respectivas saídas, e com isso, gera uma função que pode ser utilizada para classificar observações que não se conhece o resultado.
- Aprendizagem não-supervisionada: o programa trata de identificar os padrões do problema com base nas entradas, sem que nenhuma saída especifica seja disponibilizada.
- Aprendizagem por reforço: o programa interage com o ambiente do problema e aprende conforme o explora.

O método a ser utilizado depende do tipo do problema apresentado. O aprendizado por reforço se mostra o mais adequado ao contexto do trabalho em questão, pois os dados gerados por ele são tipicamente de problemas de classificação, onde se tem dados armazenados em uma certa ordem com base em um critério conhecido. O contexto do problema tratado no trabalho apresenta uma base de observações solucionadas e indexadas por meio de alguns atributos, que pode ser utilizada para o treinamento do programa. O reforço é feito através do algoritmo de otimização combinatória, implementado para a atribuição dos melhores resultados aos problemas nunca antes encontrados.

Na literatura, são utilizados métodos de classificação para as mais diversas tarefas. Em 2009, um artigo disponibilizado pelo *Jornal of Convergence Information Technology* tratou de fazer uma comparação dos métodos de classificação baseando-se no tipo dos atributos e no tamanho das amostras (ENTEZARI-MALEKI, 2009). Os resultados do trabalho demostram os melhores métodos a serem utilizados mediante as circunstâncias do problema.

Dentre os métodos mais comuns utilizados para classificação de dados, destacam-se a indução por arvores de decisão (ID3), métodos baseados em regra, métodos probabilísticos,

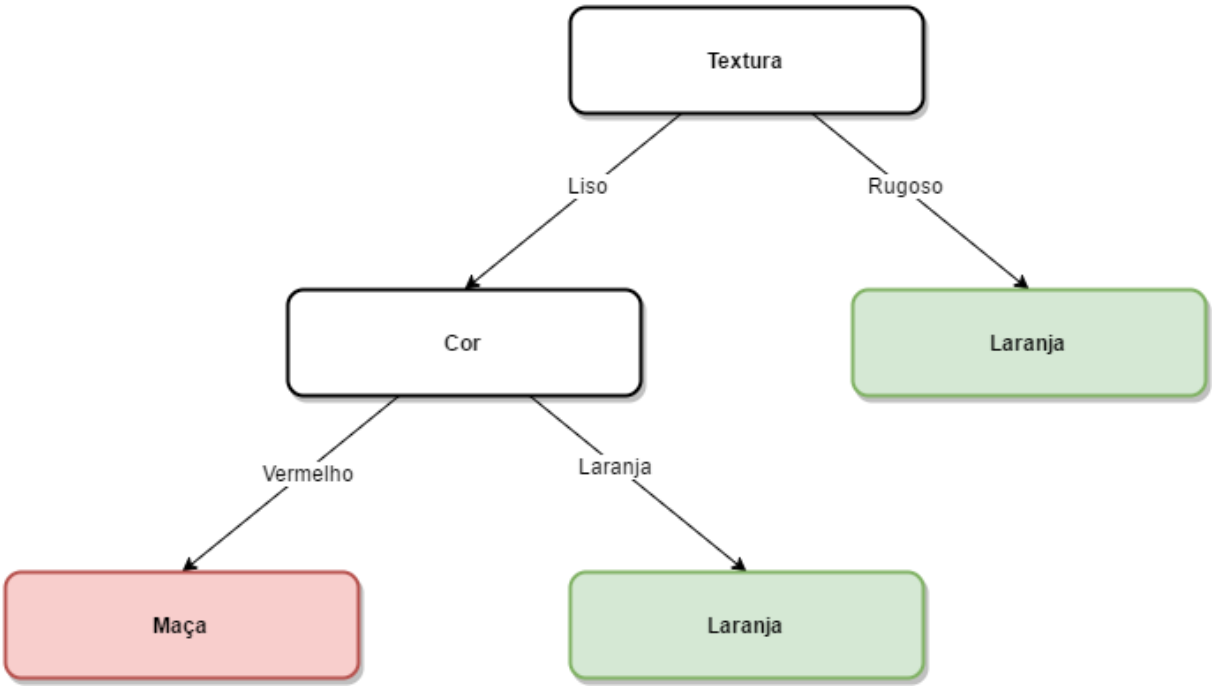
métodos SVM, métodos baseados em instância e redes neurais (AGGARWAL, 2015). Na seção a seguir é apresentada uma introdução sucinta a respeito dos métodos citados, com um enfoque particular nos métodos baseados em árvore, por se tratar de um dos métodos utilizados neste trabalho.

3.1 Indução por Árvores de Decisão (ID3)

Indução por árvores de decisão é um dos mais simples e assertivos métodos de aprendizado de máquina (RUSSEL; NORVIG, 2003). Se trata de um algoritmo de aprendizado de fácil implementação e que apresenta um alto nível de acurácia.

O método gera uma árvore de decisão com base na escolha de alguns atributos, que posteriormente pode ser utilizada para a classificação de observações. Os atributos são identificadores que denotam informações relevantes a respeito do contexto da situação, enquanto as observações são formadas por estes atributos e neles contém os dados reais da situação. Por exemplo, é necessário saber se uma determinada fruta é uma laranja ou uma maçã. Dentre as diversas características de uma fruta, serão utilizadas sua a cor e sua textura para a classificação. Estes são os atributos da classificação. Eles são extraídos com base em algum critério conhecido e serão utilizados como variáveis para as características referentes às frutas. Depois de determinados os melhores atributos, uma série de exemplos de teste é passada para o programa, efetivamente introduzindo as informações de cor e textura de cada fruta e indicando se as amostras se referem a uma laranja ou a uma maça. Desta forma, a árvore será gerada e ajustada para classificar as observações, como é demonstrado na Figura 1. Uma observação é composta pelos atributos de uma nova fruta, porém diferente dos exemplos de teste, a observação não possui a informação de qual fruta se trata. O programa deve então passar estes atributos pela árvore de decisão gerada e apresentar o resultado. Com base nas observações e resultados passados para o programa anteriormente, ele irá indicar a fruta que mais se adequa aos atributos introduzidos.

Figura 1 – Árvore de Decisão gerada para classificação de laranjas e maças.



Fonte: Elaboração própria.

Na geração de uma árvore de decisão, os exemplos de teste introduzidos são descritos por meio de um conjunto de atributos e suas classes, que são utilizados para montar os nós de condição e os ramos e folhas de resultado. Cada nó da árvore corresponde a uma condição composta por um atributo e um objeto de teste. A árvore apresenta um resultado para cada condição, seja ela verdadeira ou falsa, podendo ser um resultado final (representado por uma folha) ou um ramo, que levará a outros nós a serem testados. Após sua geração, a árvore está apta a receber novas observações e predizer a qual classe pertence. Os atributos das observações serão utilizados como objeto de teste para percorrer os nós da árvore e resultar em uma saída booleana (sim ou não). Implementações com saídas mais abrangentes também podem ser representadas, se for necessário que o resultado esteja presente em um conjunto.

É importante que durante a geração da árvore sejam escolhidos os melhores atributos para montar os ramos, pois uma escolha incorreta dos atributos pode acarretar a construção de uma árvore ineficiente. O objetivo então é maximizar a homogeneidade/pureza de cada conjunto por meio de heurísticas, sendo que o melhor atributo do conjunto é o que possui o maior nível de pureza dentre os demais. Para tal, o conjunto de atributos é iterado e são utilizados métodos heurísticos para determinar a homogeneidade do conjunto de observações atual da iteração. A cada nova iteração, o processo é repetido excluindo do conjunto os atributos

que já foram utilizados anteriormente. Os atributos que apresentam a maior pureza são priorizados, a fim de otimizar a classificação, pois os ramos serão divididos de forma a terem muitas soluções de um lado e poucas do outro, fazendo com que a maior parte das soluções inconsistentes seja desconsiderada. Com a pureza dos atributos sendo levada em consideração, a árvore tenderá a ser dividida de forma a priorizar os atributos que menos mudaram durante o treinamento, assim diminuindo o tamanho da árvore e ignorando grande parte dos resultados incoerentes durante a classificação de uma dada observação.

Na construção das árvores de decisão são utilizados métodos como Entropia e Ganho de Informação para ordenar os atributos de forma otimizar as árvores.

3.1.1 Entropia

Em teoria da computação, entropia é considerada a medida do grau de incerteza presente em uma variável aleatória (SHANNON, 1948). A entropia é utilizada para determinar o grau de homogeneidade/pureza de um conjunto.

Dado um conjunto **S**, seus valores são iterados e atribuídos à classe $\boldsymbol{p_i}$, que são multiplicados pelo $\boldsymbol{log_2}$ de $\boldsymbol{p_i}$ e então somados aos resultados das demais iterações para a obtenção da entropia.

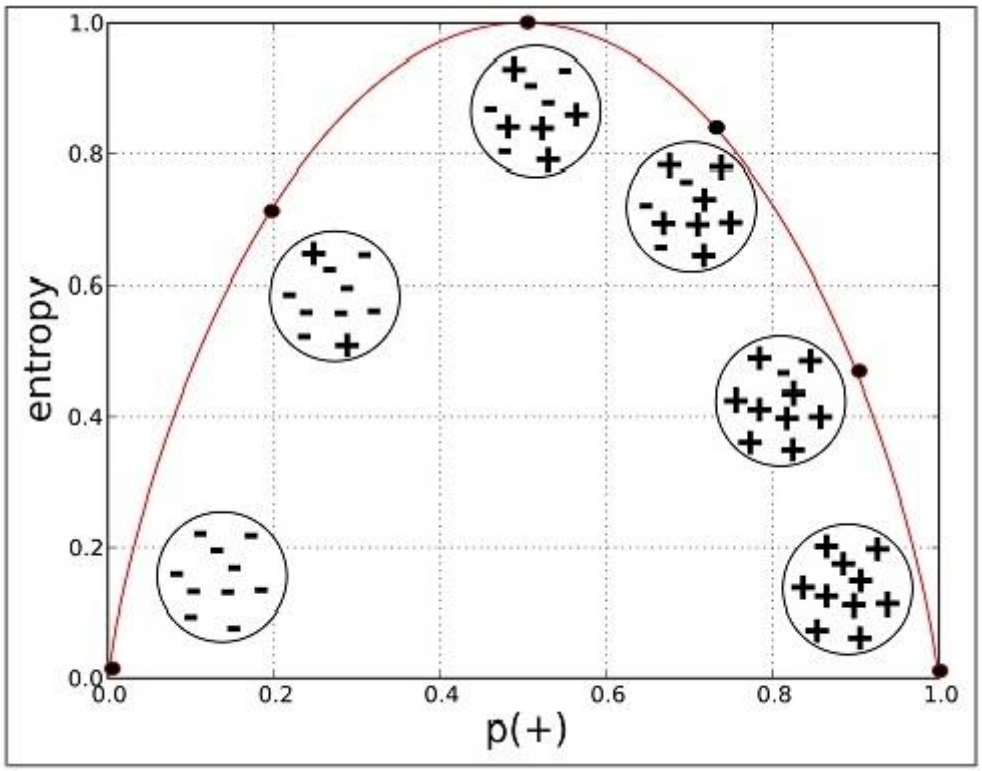
$$\boldsymbol{Entropy(S)} = \sum \boldsymbol{p_i\, log_2\, p_i}$$

Onde:

- **S** é um conjunto de elementos que contém um ou mais tipos de classe.
- $\boldsymbol{p_i}$ é o elemento que representa a iteração das classes contidas em **S.**
- $\boldsymbol{log_2}$ é o logaritmo binário, utilizado para identificar se um valor inserido é uma potência de 2.

Como é demonstrado na Figura 2, quanto mais heterogêneo for o conjunto, maior é a sua entropia. Supondo que em um conjunto existam duas classes, a entropia deste conjunto atinge seu máximo quando ele possui a mesma quantidade de elementos de cada classe, e é nula quando o conjunto é constituído por uma única classe.

Figura 2 – Gráfico demonstrando o crescimento da entropia mediante o aumento na mistura das classes.



Fonte: PROVOST, Foster; FAWCETT, Tom. **Data Science for Business – What You Need to Know About Data Mining and Data-Analytic Thinking.** Disponível em: <https://goo.gl/jYuOCh>. Acesso em 02 julho de 2016.

Neste caso, para a obtenção da entropia, o conjunto **S** apresenta a quantidade de elementos da classe positiva (+) e da classe negativa (-), e é inserido na função de entropia para obtenção do resultado, representado da seguinte forma:

$$\boldsymbol{Entropy(S) = -p_+ \, log_2 \, p_+ - p_- \, log_2 \, p_-}$$

Onde:

- **S** é o conjunto de exemplos de treino;
- $\boldsymbol{p_+}$ representa a quantidade de elementos positivos;
- $\boldsymbol{p_-}$ representa a quantidade de elementos negativos;
- $\boldsymbol{log_2}$ é o logaritmo binário.

A entropia então é dada pela diferença na quantidade de elementos positivos em relação a quantidade de elementos negativos.

3.1.2 Ganho de Informação

O ganho de informação é determinado pelo quão bem um dado atributo separa o conjunto de exemplos de treino de acordo com o objetivo da classificação (MITCHELL, 1997).

Um determinado conjunto é dividido pelo atributo que apresenta o maior ganho de informação dentre os demais. Para isso, os atributos do conjunto são iterados e seus ganhos de informação são calculados. O ganho de informação é obtido dividindo o conjunto pelo atributo atual da iteração e calculando a diferença entre a entropia do conjunto não dividido pela entropia da média dos dois novos conjuntos resultantes da divisão. Os dois novos conjuntos gerados pela divisão serão utilizados nos próximos nós da árvore, onde o mesmo processo se repete até que uma folha seja alcançada. O atributo com o maior ganho de informação encontrado é utilizado e retirado das próximas operações. O ganho de informação é dado pela seguinte equação:

$$\boldsymbol{Gain(S,A) = Entropy(S) - \sum_{v \in values(A)} \frac{|S_v|}{|S|} . Entropy(S_v)}$$

Onde:

- **S** é o conjunto sem divisão.
- **A** é o atributo atual da iteração.

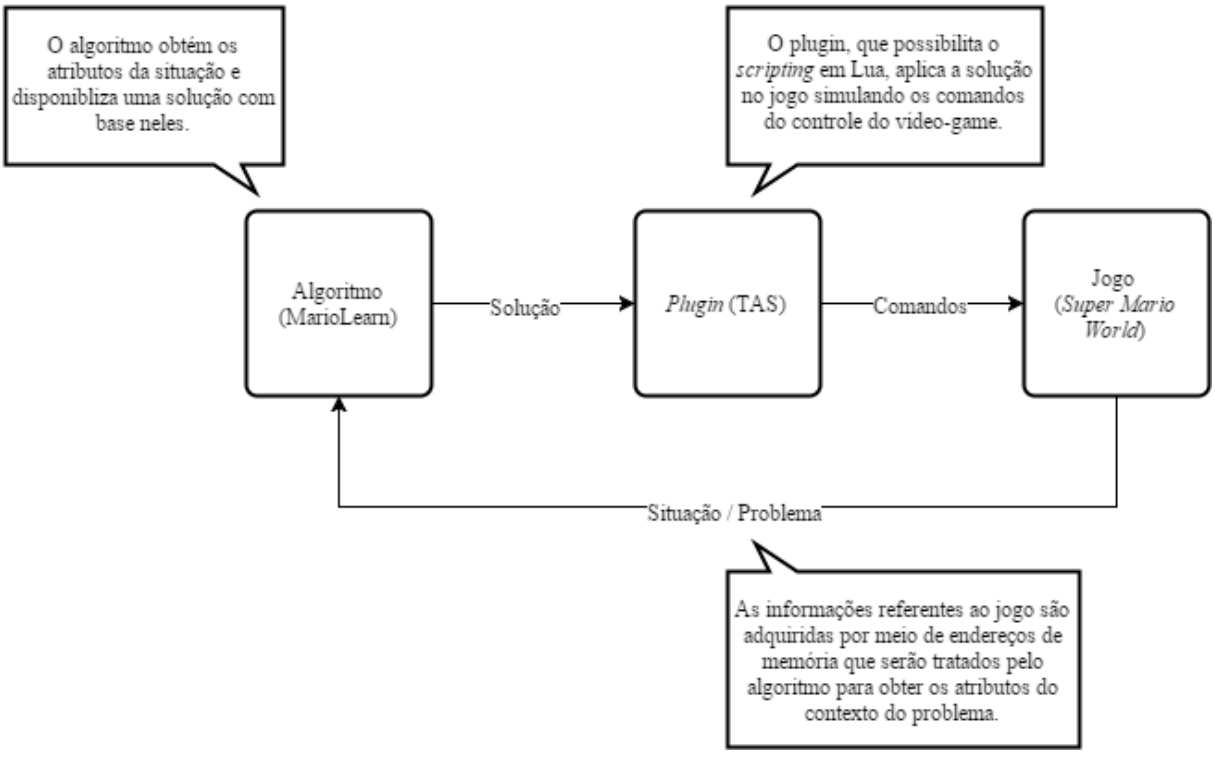
O ganho então é dado pela diferença entre a entropia de **S** e a média da soma das entropias dos conjuntos divididos pelo atributo **A**.

4 MarioLearn

MarioLearn é o nome do algoritmo de IA resultante das pesquisas realizadas neste trabalho. O algoritmo utiliza uma base de soluções gerada por um processo de análise e otimização combinatória, em conjunto com métodos de classificação para a generalização de soluções para problemas não-determinísticos[5]. O acesso e a manipulação dos atributos do jogo é feito a partir de um emulador de jogos do SNES (*Super Nintendo Entertainment System*), por meio da utilização de um *plugin* chamado TAS (*Tool Assisted Speedrun*), que permite a programação de scripts na linguagem de programação Lua[6].

A Figura 3 demonstra, de maneira rudimentar, a utilização do algoritmo em conjunto com o *plugin* para o acesso de informações e injeção de comandos no jogo.

Figura 3 – Diagrama que demonstra o fluxo de funcionamento do programa.



Fonte: Elaboração própria.

O presente capítulo detalha as etapas de funcionamento do algoritmo, divididos por geração de soluções (4.1), classificação e otimização (4.2) e persistência de dados (4.3).

[5] Um problema é dito não-determinístico quando, para uma infinidade de entradas, não é possível expressar um algoritmo que consiga encontrar uma solução ótima para o problema em tempo polinomial.
[6] Lua é uma linguagem de programação de *scripts* de multiparadigma, baseada em tabelas associativas e semântica extensível.

4.1 Geração de soluções

Inicialmente o programa faz uma análise combinatória com base em um objeto denominado *commands*, que contém todos os botões disponíveis no controle do videogame, assim como todos os eventos que podem ser realizados por cada botão (*Down*, *Up*, *Press*). O processo gera uma lista de possíveis soluções para situações (problemas) encontrados no jogo, já que na lista estarão contidas todas as combinações possíveis de serem realizadas por um jogador com o intuído de completar uma fase.

O processo de análise combinatória pode ser descrito como um conjunto de procedimentos que possibilita a construção de grupos diferentes formados por um número finito de elementos de um conjunto sob certas circunstâncias[7]. A seguinte formula descreve o processo de análise combinatória denominado arranjo simples:

$$A_{n,p} = \frac{n!}{(n-p)!}$$

Onde:

- **A** é o conjunto de elementos;
- **n** é o número de elementos do conjunto; e
- **p** é o número de elementos a agrupar.

A análise combinatória é feita por meio de uma função recursiva denominada *generateVariations*. A função recebe dois parâmetros: *action* e *index*.

- *Action* – parâmetro que possui o conjunto de soluções atual da recursão.
- *Index* – identificador do botão que terá seus eventos iterados na recursão.

Na primeira recursão é passado um conjunto vazio no parâmetro *action*, assim como o número 1 no parâmetro *index*, que é utilizado para acessar o objeto *commands* e recuperar o primeiro par de botão e eventos. O par é armazenado em uma variável chamada *command*, utilizada para compor o parâmetro *action*, que por sua vez é adicionada a uma lista denominada *variations* (lista de soluções finais). A recursão seguinte é chamada passando o novo parâmetro *action* e o parâmetro *index* com o incremento de 1 (*index* + 1). Na nova recursão, todo o processo é repetido, porém agora com o novo index do botão a ser iterado e o novo conjunto contido em *action*.

[7] Definição retirada do site **Matemática Essencial**, Disponível em: <https://goo.gl/bKVr0U>. Acessado em 13 de outubro de 2016.

Por fim, a lista *variations* possui todas as soluções possíveis de serem geradas a partir dos botões presentes no controle. Esta lista pode ser considerada como o horizonte de soluções possíveis. Cada solução, além do comando a ser aplicado no jogo, apresenta um peso, representado pela variável *weight*. Este peso é utilizado para ponderar as soluções e melhor a atribuição das mesmas para problemas não conhecidos, e será utilizada pelo processo de otimização tratado mais à frente.

4.2 Classificação e Otimização

Para a geração da árvore de decisão foi desenvolvida uma biblioteca, na linguagem de programação Lua, para auxílio na implementação de algoritmos de aprendizado de máquina. A biblioteca expõe algumas funções que permitem o treinamento de uma árvore de decisão e a sua utilização para classificar observações. As funções expostas são:

- **tree** – função que recebe como parâmetro um conjunto de *features* (atributos) e um conjunto de *labels* (resultados). Retorna a arvore de decisão que é utilizada para a classificação.
- **classify** – função protótipo da árvore retornada pela função *tree*. Recebe como parâmetro um conjunto de *features* e retorna o *label* resultante do treinamento da árvore. É responsável pela predição (classificação) das observações com base na árvore gerada.

Exemplo de código

```lua
local ml = requires "ml"

local features = {
  {"liso", "vermelho"},
  {"rugoso", "laranja"},
  {"liso", "laranja"}
}

local labels = {"maca", "laranja", "laranja"}

local decision_tree = ml.tree(features, labels)

local result = decision_tree.classify({"rugoso", "vermelho"}) -- Laranja
```

Efetivamente para gerar a árvore de decisão que será utilizada para classificar as situações encontradas no jogo, são fornecidas as soluções já encontradas para a biblioteca. A

mesma irá utilizar os métodos descritos no capítulo 3 para retornar uma árvore treinada com base nestas soluções.

Após a árvore ser gerada, já é possível tentar classificar as situações encontradas no jogo. Quando o personagem se depara com um obstáculo ou inimigo, é recolhida uma observação a respeito do contexto do problema. Este contexto é adquirido por meio da janela de observação implementada no algoritmo e representa a área de visão que o algoritmo tem do que está acontecendo no jogo. Nesta observação os atributos são o identificador, o estado e o posicionamento do inimigo ou obstáculo. Esta observação então é passada pela árvore, e o resultado, que é um conjunto de comandos, é aplicado no jogo por meio do *plugin* TAS.

Como dito anteriormente, existe um processo de otimização que é realizado na lista gerada pela análise combinatória conforme as soluções vão sendo classificadas para os problemas. Sempre que o programa se deparar com um problema não conhecido ele irá utilizar a solução com maior peso (*weight)* existente no horizonte de soluções (*variations)*. Após a solução ser atribuída ela é testada e a sua procedência é avaliada. Em caso de sucesso na resolução do problema, o valor de peso (*weight)* da solução bem-sucedida é incrementado, aumentando sua relevância no horizonte de soluções. Já no caso de procedência negativa, o peso é decrementado, fazendo com que a solução tenha menos chance de ser escolhida para os demais problemas.

A otimização do algoritmo se trata de um processo heurístico, ou seja, de uma forma geral ele tende a dar boas soluções para os problemas não conhecidos. Porém em alguns casos pode acontecer de a solução para o problema ser a com o menor peso. Isso fará com que o programa percorra todas as demais soluções até chegar na correta, mas ela será alcançada eventualmente.

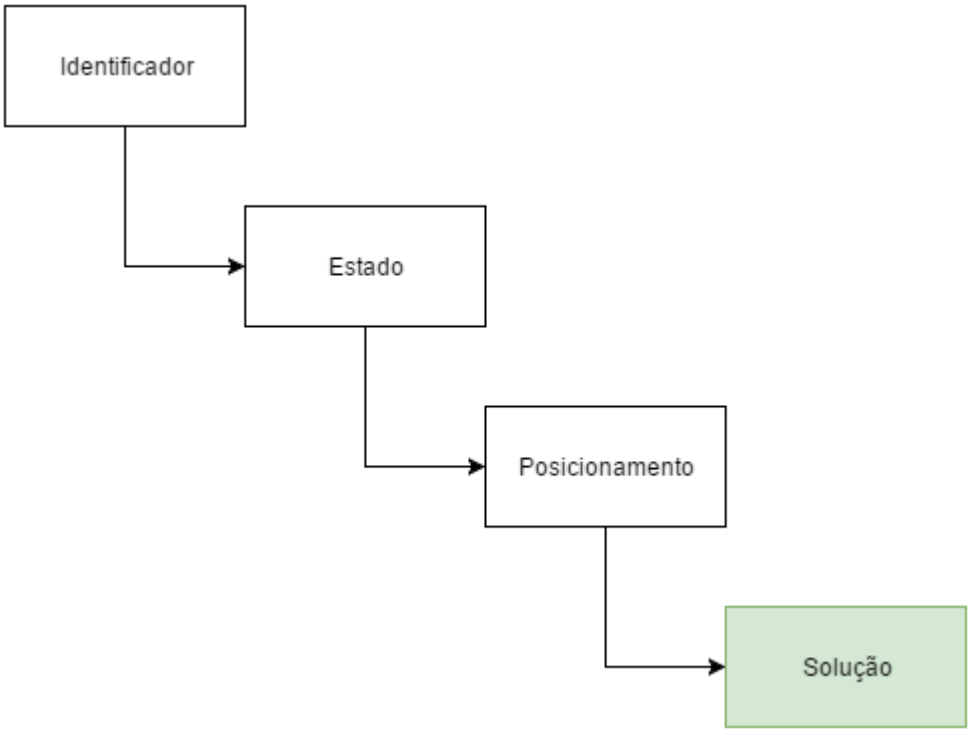
4.3 Persistência de Dados

Por fim, quando as soluções são obtidas, elas serão armazenadas em uma base de dados. Na implementação do programa, é utilizado um sistema de arquivos para armazenar as soluções. A cada vez que uma solução é encontrada ela é persistida no arquivo, que é utilizado para a geração de uma nova árvore de decisão.

O armazenamento é feito indexando as soluções pelos atributos definidos nas observações. Assim como demonstrado na Figura 4, o arquivo apresenta uma estrutura hierárquica que organiza as soluções de forma a agrupar os comandos finais da solução (que serão aplicados ao jogo) em um objeto indexado pelo atributo de identificação, estado e posicionamento. É importante notar que um identificador, assim como um estado e um

posicionamento, pode conter diversas soluções indexadas nele. Portanto, para um identificador podem existir diversas soluções com estados e posicionamentos diferentes.

Figura 4 – Estrutura utilizada para indexar soluções com base nos atributos selecionados.



Fonte: Elaboração própria.

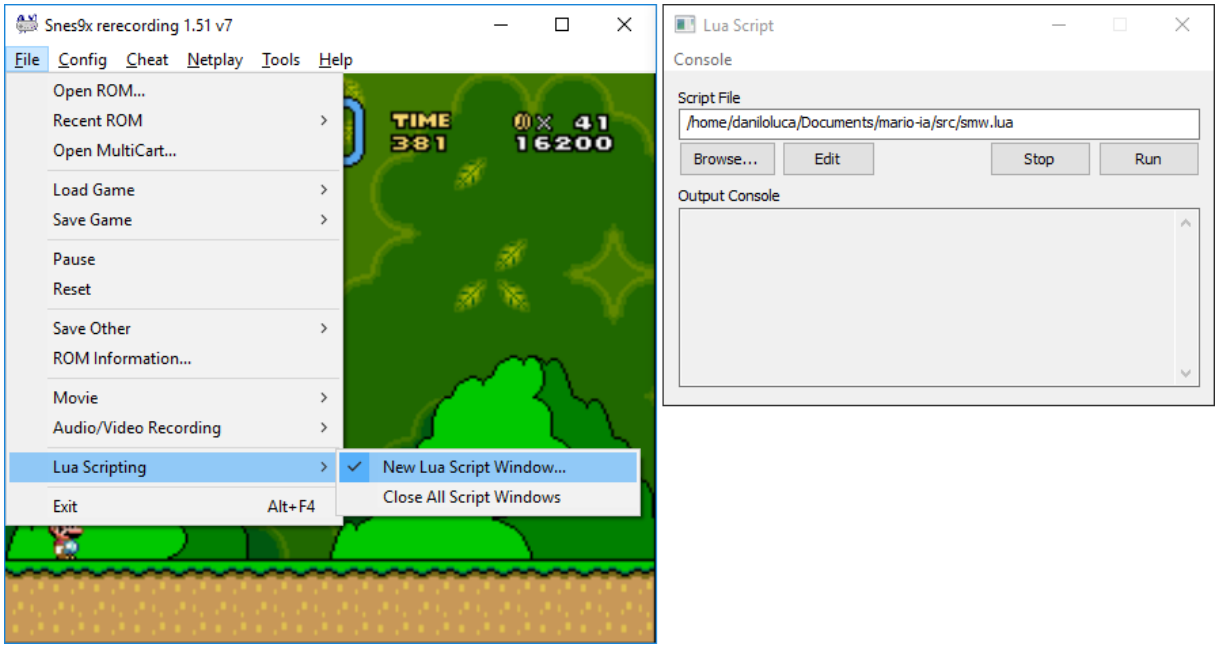
4.4 Execução

A aplicação do algoritmo é feita por meio do *plugin* TAS, que é responsável por injetar os comandos finais no jogo e disponibilizar acesso as variáveis utilizadas por ele para armazenar valores, tais como posicionamento, identificação e tipo dos elementos. O *plugin* é instalado no emulador Snes9x[8], que é um conhecido emulador de SNES. Outros emuladores do gênero também têm suporte ao *plugin*, porém este foi escolhido por se tratar de uma aplicação multiplataforma e de código aberto, podendo ser utilizado no Windows, Mac e Linux.

Como é demonstrado na Figura 5, o emulador apresenta a opção "*New Lua Script Window...*", que abre uma janela onde é possível adicionar um *script* com a extensão ".lua". O *plugin* disponibiliza uma biblioteca[9], que expõe várias funções para possibilitar o acesso a atributos e variáveis do jogo e simular a aplicação de comandos no mesmo (simula a ação de pressionar os botões do controle).

[8] Código fonte do emulador **Snes9x**. Disponível em: <https://goo.gl/h52NeP>. Acessado em 29 de agosto de 2015.
[9] Documentação da biblioteca disponibilizada pelo *plugin* TAS. Disponível em: <https://goo.gl/EJjpvK>. Acessado em 29 de agosto de 2015.

Figura 5 – Demonstração da utilização do *plugin* no emulador.



Fonte: Elaboração própria.

No apêndice A é apresentado um tutorial para auxílio na configuração e execução do algoritmo no emulador. O código fonte do projeto pode ser encontrado visionado no site GitHub[10].

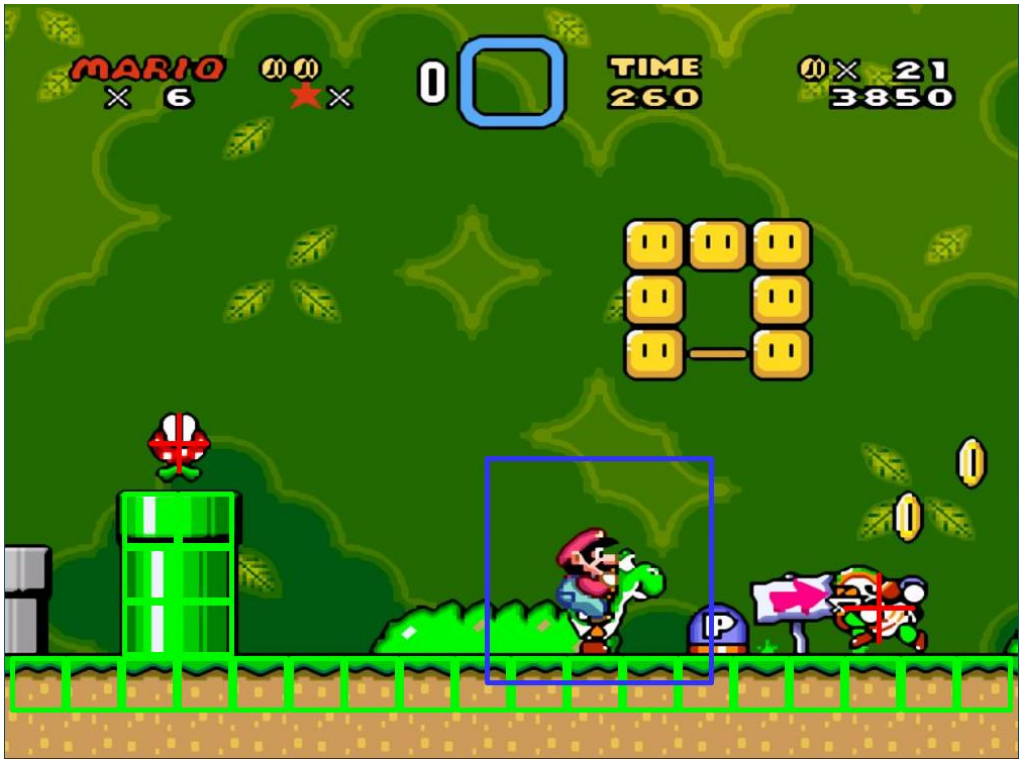
4.5 Informações Visuais

A biblioteca disponibilizada pelo *plugin* TAS também apresenta algumas funções para o gerenciamento da GUI (*Graphical User Interface*)[11]. Estas funções podem ser utilizadas para apresentar, de forma visual, informações relevantes para o jogador. Normalmente são utilizadas para apresentar informações que aumentam a precisão do jogador. Por exemplo, desenhar um retângulo em torno de um personagem, inimigo ou obstáculo para delimitar a sua área de colisão.

Como é demonstrado na Figura 6, as funções de GUI são utilizadas pelo algoritmo para demonstrar qual é a visão que ele tem do que se passa no jogo.

[10] O código fonte do algoritmo está disponível em: <https://github.com/daniloluca/mario-ia>
[11] Uma GUI é um tipo de interface que permite a representação gráfica de informações por meio de elementos gráficos.

Figura 6 – Demonstração do algoritmo em funcionamento no jogo.



Fonte: Elaboração própria.

As informações de localização de identificação dos elementos demarcados pelas funções de GUI são as mesmas utilizadas pelo algoritmo para solucionar as fases. Por meio destas informações o algoritmo identifica se o elemento se trata de um inimigo ou de um obstáculo dentro do jogo, e a partir disso aplica o contorno e as cores corretas ao elemento por meio das funções de GUI.

Cada forma e cor utilizada para demarcar os elementos tem um significado. Este tipo de informação não é relevante para o algoritmo, mas é aplicada no jogo para que o usuário tenha uma visão de como o algoritmo "enxerga" o jogo e entenda suas reações mediante a uma situação.

O jogo *Super Mario World*, assim como muitos outros jogos de SNES, possui um cenário dividido em blocos (*tiles*), que são pequenos trechos do cenário apresentados na forma de uma matriz para constituir a cena completa do jogo. Cada forma aplicada na GUI é referente a um inimigo ou a um bloco do cenário. Desta forma é possível que o algoritmo considere que ao redor do personagem existem diversos elementos, e que ele precisa desenvolver uma solução

que utilize este contexto para fazer com que o personagem chegue mais próximo de seu objetivo (concluir a fase).

As formas desenhadas na GUI representam o tipo de cada elemento, sendo:

- Quadrado verde – um bloco do cenário que o personagem pode interagir, de maneira a poder subir nele ou mesmo colidir com suas laterais. Um obstáculo pode ser representar por um conjunto de quadrados verdes;
- Cruz vermelha – pode ser um inimigo, pelo qual o personagem pode ser morto se entrar em contato com ele. Pode também ser um elemento passivo que atribui alguma melhoria (vantagem) ao personagem.; e
- Área azul – esta área é a janela de observação e representa a área de visão do personagem, todos os elementos que estiverem dentro desta área serão considerados pelo algoritmo como o contexto do problema atual, sendo utilizados para sua resolução. Não necessariamente se trata de um quadrado, pois pode ser modificado, aumentando ou diminuindo sua altura ou largura, para otimizar os resultados, de forma a não sobrecarregar a observação com informações desnecessárias.

Para o algoritmo, todos estes elementos e informações não tem significado logo no início. Eles são apenas entradas que o algoritmo recebe e assimila com seus resultados. Posteriormente ele utiliza o treinamento realizado para classificar problemas que não possuem um resultado introduzido em sua entrada. No fim das contas ele apenas associa um conjunto de entradas a uma possível saída, se baseando nos valores disponíveis durante o treinamento.

O programa também apresenta algumas informações precisas por meio da GUI, tais como coordenadas e velocidade do personagem, número de *resets* feitos no jogo, etc.

Figura 7 – Informações apresentadas pelo algoritmo na GUI.

Fonte: Elaboração própria.

5 Experimentos e resultados

Neste capítulo são apresentados detalhes a respeito dos experimentos realizados para a validação da solução proposta e os resultados obtidos. Os experimentos foram realizados apresentando diversas situações do jogo ao algoritmo onde era necessário que ele fizesse uso do conhecimento adquirido durante a resolução dos demais problemas, e foram constatadas as situações onde ele foi capaz de solucionar os problemas e as ocasiões onde algumas características especificas não o permitiram de conseguir.

As situações apresentadas e seus resultados positivos ou negativos são detalhadas na Seção 5.1. Uma discussão a respeito dos resultados e algumas especulações a respeito de possíveis melhorias são apresentados na Seção 5.2.

5.1 Situações

Como é dito no capítulo 4, o algoritmo obtém os elementos presentes no contexto do problema por meio da janela de observação. A partir destas informações ele identifica qual é a situação que o personagem se encontra, e com isso utiliza seus métodos para tentar solucioná-la. Contudo, algumas situações necessitam que outras variáveis sejam observadas para que possam ser solucionadas. Por conta disso, algumas destas situações não podem ser resolvidas pelo programa.

Foram feitas diversas execuções em certos trechos do jogo, e nas próximas seções serão descritas as situações onde o algoritmo obteve sucesso e onde ele falhou.

5.1.1 Sucesso

O algoritmo apresenta ótimos resultados quando se depara com situações onde é possível observar os elementos presentes ao redor do personagem. Ele é capaz de generalizar grande parte das soluções encontradas para problemas nunca antes observados. Quando ele se depara com uma situação nunca antes vista e a generalização de suas soluções não é efetiva, ele aplica uma das soluções presentes em uma lista que contém todas as combinações possíveis do controle do vídeo game. Quanto mais ele treina, mais eficiente este método se torna, pois quando uma solução é efetiva, seu peso é incrementado, fazendo com que ela tenha mais chance de ser utilizada para uma outra situação.

Na Figura ..., é demonstrada uma situação onde o algoritmo resolve uma das situações e depois generaliza a solução para as demais situações, passando sem a necessidade de refazer a classificação.

Figura 8 – Exemplo de situação onde o algoritmo demonstra a generalização de soluções.

Quando o algoritmo soluciona a primeira situação, ele gera novamente a árvore de decisão com base nas novas informações coletadas. Ele então utiliza esta nova árvore para testar as demais situações. Se a situação for semelhante a anteriormente solucionada, provavelmente a mesma será utilizada e assim subsequentemente até que ele encontre uma situação onde ele precise aplicar o mesmo processo novamente.

5.1.2 Falha

Por conta de alguns fatores, o algoritmo não consegue passar de certas situações específicas. Os problemas normalmente estão relacionados a falta de informações disponibilizadas pelo contexto, a necessidade de observar informações mais específicas no contexto da situação e a necessidade de esperar por eventos como aguardar que uma plataforma em se movimente até que o personagem possa subir.

Nas seções a seguir, são descritas as situações onde o algoritmo, em seu estado atual, não consegue solucionar.

Falta de elementos na janela de observação

Como o algoritmo utiliza os elementos que se dispõem ao redor do personagem para determinar a situação do mesmo, a falta de elementos presentes na janela de observação faz com que o algoritmo erre na maioria das vezes. Quando não existem elementos na janela de observação, a situação apresenta valores nulos em todos os atributos. O algoritmo considera isso como uma situação válida, aplicando uma solução para a mesma. Contudo, por se tratar de uma situação que pode ocorrer em diversos pontos do jogo, ele acaba por alterar essa reação toda vez que a solução não apresenta resultado positivo, modificando assim todas as demais situações que apresentam as mesmas características.

O algoritmo precisa considerar este tipo de situação onde não existem elementos na janela de observação. Nestes casos ele aplica uma combinação de comandos até que alguma das combinações faça com que o personagem ande em direção ao objetivo. Contudo, quando o personagem morre em uma situação igual a que o algoritmo atribuiu a combinação que o faz andar, ele a modifica, fazendo com que o personagem não ande mais para os casos anteriores, aplicando outro tipo de combinação como pular ou abaixar. Isso fará com que todo o desenvolvimento anterior seja inutilizado, pois todas as amostras anteriores levavam em consideração que para uma situação onde não houvessem elementos ao redor do personagem, ele deveria andar.

Figura 9 – Exemplo de situação onde o personagem morre sem nenhum elemento ao seu redor.

Fonte: Elaboração própria.

Na Figura 9, é demonstrada uma situação onde o personagem está prestes a morrer sem nenhum elemento ao seu redor. Isso fará com que seja modificada a reação atribuída a esta situação, que neste momento possui a combinação responsável por fazer o personagem andar. Depois disso, o personagem não irá mais andar quando não observar nada ao seu redor, mas sim executar a nova reação aplicada quando ele morreu, impactando assim todas as demais situações anteriormente solucionadas.

Tamanho da janela de observação

Este problema tem relação com o anterior, porém pode ocorrer mesmo quando existem elementos ao redor do personagem. O Tamanho da janela de observação está diretamente ligado ao nível de especialização ou generalização de uma situação. Quanto maior a janela de observação, mais específica será a solução para a situação observada. Isso faz com que a solução tenha menos chance de se adequar a outras situações parecidas. Quanto menor a janela de observação, menos informações a respeito do contexto serão consideradas pelo algoritmo, mas

as soluções serão mais genéricas, aumentando a chance de poderem se utilizadas para outras situações.

O principal problema relacionado ao tamanho da janela de observação é o de acabar por generalizar demais determinadas situações. Isso faz com que informações importantes a respeito do contexto da situação sejam ignoradas. Como demonstrado na Figura 10, algumas situações requerem que o algoritmo observe mais do que é delimitado pela janela de observação, pois a falta de conhecimento do algoritmo a respeito do contexto da situação fará com que o personagem continue andando e caia. Se a janela de observação fosse maior, a plataforma poderia ser observada pelo algoritmo, que possivelmente faria com que o personagem pulasse, permitindo-o de continuar a fase.

Figura 10 – Exemplo de situação onde o algoritmo falha por falta de observação do contexto.

Fonte: Elaboração própria.

Apesar de o aumento na janela de observação melhorar a classificação da situação, ele também pode acarretar consequências que podem fazer com que o personagem se quer chegue nesta parte da fase. O excesso de informações durante o treinamento pode fazer com que as

soluções fiquem muito específicas, fazendo com que o número de tentativas para terminar a fase aumente consideravelmente.

Espera de eventos

As situações mais complexas, onde o personagem precisa esperar que algo aconteça são as mais difíceis de se tratar. Esta dificuldade é, dentre outros fatores, consequência dos demais problemas apresentados, que descrevem a deficiência do algoritmo em validar situações que não apresentam uma quantidade significativa de informação e sua dificuldade em ajustar a janela de observações para validar situações de forma mais eficiente.

Como é demonstrado na Figura 11, algumas situações requerem que o personagem aguarde até que algum elemento do cenário se mova para possibilitar a passagem. Estas situações são muito decorrentes no jogo e o algoritmo precisa ser capaz de identificar este tipo de comportamento para que tenha êxito em resolver situações que apresentem este tipo de característica.

Figura 11 – Exemplo de situação onde o personagem morre por não aguardar pela plataforma.

Fonte: Elaboração própria.

O algoritmo possui as informações necessárias para identificar que um elemento na janela de observação pode se mover. Contudo, fazer com que o algoritmo pare o personagem e aguarde a plataforma é uma tarefa complexa, pois para que ele apresentasse este tipo de comportamento seria necessário resolver o problema do tamanho da janela de observação e também uma grande quantidade de execuções do programa seria necessária para que ele chegasse a este resultado.

5.2 Melhorias

O algoritmo se mostrou muito eficiente para o objetivo proposto, mediante as características da fase. Se tratando de velocidade e generalização de soluções, o algoritmo consegue percorrer boa parte da fase em apenas algumas execuções. Ele apresenta mecanismos capazes de se ajustar para aprimorar a atribuição de soluções e com isso tornar ainda mais rápida a resolução da fase.

Com o conhecimento das situações onde o algoritmo não apresenta êxito, é possível deduzir que ele não é capaz de completar todo o jogo *Super Mario World*, pois com o passar do jogo as situações que ele não pode tratar ficam cada vez mais decorrentes. Por conta destes fatores, é de interesse que métodos para aprimorar o algoritmo sejam desenvolvidos.

Dentre os itens que podem ser melhorados e implementados no algoritmo destacam-se:

- A capacidade de modificar o tamanho da janela de observação de forma dinâmica, para que o algoritmo possa ter informações mais especificas a respeito de algumas situações, mas sem perder a generalização necessária nas demais;
- A capacidade do algoritmo de observar variáveis como tempo, para que seja possível solucionar situações que apresentem a necessidade de o personagem aguardar por algum acontecimento para prosseguir.

As implementações citadas podem ser desenvolvidas de maneira individual, por meio de outros projetos e posteriormente seus resultados podem ser aplicados no algoritmo, ou podem ser realizadas no próprio algoritmo, visando melhorar seus resultados.

6 Conclusões

A implementação de inteligências artificiais capazes de se adequar...

REFERÊNCIAS

STANLEY, Kenneth O.; MIIKKULAINEN, Risto. **Evolving Neural Networks through Augmenting Topologies.** Massachusetts Institute of Technology, 2002. Disponível em: <http://goo.gl/TccvC>

KISHIMOTO, André. **Inteligência Artificial em Jogos Eletrônicos.** 2004. Disponível em: <http://goo.gl/CS45lZ>

CHAMPANDARD, Alex J. **AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors.** 8 p. Disponível em: <http://goo.gl/Cj8khW>

TANIMOTO, Steven L. **The Elements of Artificial Intelligence.** Washington: Computer Science Press, 1987. Disponível em: <http://goo.gl/O9n6f6>

GHADERI, Hojjat. **CSC384: Introduction to Artificial Intelligence.** University of Toronto, 2009. 5 cap – Backtracking Search. Disponível em: <http://goo.gl/8UhmeO>

FUSSELL, Don. **AI – Decision Trees and Rules Systems.** University of Texas at Austin, 2012. Disponível em: <http://goo.gl/jyp8yK>

RUSSEL, Stuart J.; NORVIG, Peter. **Artificial Intelligence: A Modern Approach. 2. ed.** Pearson Education, 2003.

AGGARWAL, Charu C. **Data Classification: Algorithms and Applications**. IBM T. J. Watson Research Center, Yorktown Heights, New York, USA. Disponível em: <https://goo.gl/4EGygO>

KIANG, Melody Y. **A Comparative Assessment of Classification Methods.** Information System Department, College of Business Administration, California State University, 2002. Disponível em: <https://goo.gl/LcBgHI>

ENTEZARI-MALEKI, R.; RAZAEI, A.; MINAEI-BIDGOLI, B. **Comparison of Classification Methods Based on the Type of Attributes and Sample Size.** Department of Computer Engeneering, Iran University of Science & Technology, Tehran, Iran, 2009. Disponível em: <https://goo.gl/wc2ltw>

SHANNON, C. E. **A Mathematical Theory of Communication.** The Bell System Technical Journal, 1948. Disponível em: <https://goo.gl/jNKkCH>. Acessado em 24 de setembro de 2016.

MITCHELL, Tom M. **Machine Learning.** The Mc-Graw-Hill Companies, Inc. 1997. Disponível em: <https://goo.gl/Wk8gsD>. Acessado em 02 de outubro de 2016.

TENENBAUM, A. A.; LANGSAM, Y.; AUGENSTEIN, M. J. **Data Structures Using C.** Prentice Hall, Inc. 1995. Disponível em: <https://goo.gl/Pt188G>. Acessado em 04 de outubro de 2016.

MILLINGTON, Ian. **Artificial Intelligence for Games.** Elsevier Inc. 2006. Disponível em: <https://goo.gl/868FiM>. Acessado em 18 de outubro de 2016.

RICH, E. **Artificial Intelligence.** McGraw-Hill. 1998. Acessado em 18 de outubro de 2016.

FERREIRA, E; OLIVEIRA, T. **Som, imersão e jogos eletrônicos: um estudo empírico.** SBC – Proceedings of SBGames. 2011. Disponível em: <https://goo.gl/NR07UU>. Acessado em 10 de setembro de 2016.

APÊNDICE A - EXECUÇÃO

Como configurar?

- Baixe o repositorio.
- Abra o diretorio **\your_path\mario-ia\emulator\snes9x-1.51-rerecording-v7-win32**.
- Execute o arquivo **snes9x.exe**
- Selecione a rom no emulador, a rom se contra em **\your_path\mario-ia\emulator**.
- Com o jogo rodando, adicione arquivo **script.lua** ao plugin, ele se encontra em **\your_path\mario-ia\algorithm**.

Como utilizar?

- Existem **3** states pré-configurados.
- Você pode utilizar as teclas **F1**, **F2** e **F3** para navegar entre as fases salvas.