AMBIENTES VIRTUAIS PARA O ESTUDO DA INTELIGÊNCIA ARTIFICIAL

VIRTUAL ENVIRONMENTS AS FOR ARTIFICIAL INTELLIGENCE TEACHING

Ruy de Oliveira[1]
Mario A. de Oliveira[2]
Gabriela G. Santos[3]
Matheus C. Teixeira[4]
Vitor B. O. Barth[5]

Resumo

A Inteligência Artificial (IA) é parte integrante de diversos cursos nas áreas de Engenharia e Ciências da Computação, e o seu estudo é desafiador para professores e estudantes. Neste contexto, os Ambientes Virtuais (AV) podem potencializar o processo de ensino-aprendizagem desse assunto. Os AVs proporcionam cenários gráficos e interativos para as soluções dos vários problemas de busca em IA, de modo que o ensino e a aprendizagem se tornam bem mais intuitivos. Dessa forma, este trabalho propõe três AVs para soluções de problemas de busca clássicos, cotidianamente abordados nas disciplinas de IA: o problema do Caixeiro Viajante, do Labirinto e do Quebra-Cabeça de N Peças. Os AVs desenvolvidos possibilitam ao usuário não apenas alterar o código do algoritmo implementado, mas também incluir novos algoritmos, ou ainda testar novas heurísticas de seu interesse. Ao utilizar esses ambientes, o usuário poderá visualizar o desenvolvimento passo a passo de cada solução, e comparar o desempenho dos diferentes algoritmos, suas configurações, eficácia, completeza, entre outros parâmetros.

Palavras-chave: Ambientes Virtuais, Ensino, Busca, Inteligência Artificial.

[1] Professor Dr. do IFMT, Campus Cuiabá "Octayde Jorge da Silva", Cuiabá – MT – Brasil.
E-mail: ruy@cba.ifmt.edu.br
[2] Professor Dr. do IFMT, Campus Cuiabá "Octayde Jorge da Silva", Cuiabá – MT – Brasil.
E-mail: mario.oliveira@cba.ifmt.edu.br
[3] Graduando em Engenharia da Computação pelo IFMT, Campus Cuiabá "Octayde Jorge da Silva", Cuiabá – MT – Brasil. Graduando em. E-mail: gabi.gsantos@hotmail.com
[4] Graduando em Engenharia da Computação pelo IFMT, Campus Cuiabá "Octayde Jorge da Silva", Cuiabá – MT – Brasil. E-mail: matheuscandido2009@gmail.com
[5] Graduando em Engenharia da Computação pelo IFMT, Campus Cuiabá "Octayde Jorge da Silva", Cuiabá – MT – Brasil. E-mail: vitor.barth@gmail.com

Abstract

Artificial Intelligence (AI) courses are part of many Computer Sciences and Engineering majors, and its learning process is challenging for both the students and teachers. In this context, the Virtual Learning Environments (VLE) can improve the teaching-learning process on this issue. The VLE provide graphical and interactive scenarios that show solutions on various AI search problems, making the teaching-learning process becomes much more intuitive. Thus, this work proposes three VLEs to solve classical search problems: the Travelling Salesperson Problem, the Maze Problem and the N-Pieces Puzzle Problem. The developed VLEs for these problems include the most famous search algorithms presented by the literature, allowing the user not only to change the implemented algorithms code, but also to develop new algorithms and heuristics. By using these VLEs, the user can visualize the step-by-step development of each solution, and compare the performance of the distinct algorithms, their efficiency, completeness, among other parameters.

Keywords: Virtual Learning Environments, Teaching, Search Algorithms, Artificial Intelligence.

1. INTRODUÇÃO

Técnicas e algoritmos de Inteligência Artificial (IA) podem ser utilizados para solucionar problemas complexos de maneira mais eficiente que os métodos clássicos, o que é de vital importância para o desenvolvimento econômico e social de qualquer país ou região. Devido à sua característica multidisciplinar, soluções que fazem uso de IA são aplicadas em uma variedade de ramos, como engenharia, medicina, direito, entre outras (PINTO e colab., 2003; TRINDADE e colab., 2008).

Apesar de essencial para a formação de profissionais das ciências da computação e engenharias, cursos introdutórios à IA são desafiadores para professores e alunos, dada a diversidade e quantidade de conteúdos discutidos em sala de aula. Por conta disso, técnicas tradicionais de ensino têm se mostrado não muito eficazes no que se refere à assimilação dos conteúdos (CRESPO GARCÍA e colab., 2006). Para minimizar tais efeitos, novas metodologias de ensino têm sido elaboradas conforme encontrada na literatura.

Em Grivokostopoulou e colab. (2014) é proposto um Ambiente Virtual (AV), utilizando conceitos de Inteligência Artificial, como forma de complementar o ensino da disciplina de IA. O Ambiente desenvolvido é capaz de se adaptar ao nível de conhecimento do aluno, apresentando conceitos e problemas teóricos de acordo com as dificuldades de cada estudante.

Os autores em Ali & Samaka (2013), Dogmus e colab. (2015), Markovic e colab. (2015) e Pantic e colab. (2005) desenvolveram Ambientes Virtuais utilizando Videogames, onde o estudante pode desenvolver e testar algoritmos de IA. O desempenho destas ferramentas e outras semelhantes são avaliados em DeNero & Klein (2010), Fernandes (2016) e Yoom & Kim (2015), onde foram aplicados questionários aos alunos de turmas que utilizaram os Ambientes Virtuais. Segundo os autores, os resultados foram majoritariamente positivos.

Em Ali & Samaka (2013), Barella e colab. (2009) e Fernandes (2016), é utilizada a metodologia de Aprendizagem Baseada em Problemas (PBL, *Problem-based Learning*), a qual se mostra eficaz para este tipo de disciplina. Por meio de atividades práticas, a PBL permite aos estudantes compreender melhor os conceitos ensinados no decorrer do curso, aplicando a teoria para a solução de problemas reais.

As abordagens encontradas em DeNero & Klein (2010) e Markovic e colab. (2015) fazem uso de ferramentas de *software* ou *hardware* construídas com o intuito de aprimorar as competências do estudante. Tais ferramentas devem ser atrativas para o estudante,

incentivando-o a resolver o problema apresentado, e facilitando a visualização da solução por ele desenvolvida.

Ambientes Virtuais, como os desenvolvidos por Grivokostopulou e colab. (2014), Ali & Samaka (2013), Dogmus e colab. (2015), Markovic e colab. (2015) e Pantic e colab. (2005), são *softwares* que tornam o processo de ensino-aprendizagem mais ativos e dinâmicos, permitindo ao aluno realizar experiências, e ao professor acompanhar o desenvolvimento do aluno (PEREIRA e colab., 2007).

Com base nos trabalhos analisados, Ambientes Virtuais de Aprendizagem se mostraram úteis durante o processo de aprendizagem por possibilitar ao estudante implementar sua própria versão dos algoritmos de Inteligência Artificial e visualizar a execução da sua implementação. As referências de DeNero & Klein (2010) e Markovic e colab. (2015) representam os principais trabalhos encontrados na literatura com esta finalidade.

Algoritmos de Busca são a forma mais simples de resolução de problemas utilizando IA. Normalmente, esses algoritmos são abordados na parte introdutória dos cursos, por serem mais simples, servindo como base para os demais métodos estudados. Observa-se que, apesar de sua importância para a disciplina de Inteligência Artificial, não foi encontrada nenhuma ferramenta específica de auxílio ao ensino de Algoritmo de Busca.

Na *web* estão disponíveis diversas ferramentas de visualização de execução de algoritmos de busca em problemas clássicos, mas estes não são totalmente didáticos, visto que a implementação dos algoritmos não pode ser alterada facilmente (AIAI, 2011b, a; BRYUKH, 2018; DRAMAIX, 2012; LOTZ, 2014).

Como forma de facilitar, e consequentemente melhorar, o ensino de IA, em específico acerca do funcionamento de algoritmos de busca, este trabalho propõe três ambientes virtuais. A ideia principal destes AVs é apresentar visualmente ao usuário problemas de busca clássicos, possíveis de serem resolvidos através de Inteligência Artificial. Além de fornecer uma visualização da execução do algoritmo, estes AVs permitem ao aluno implementar seus próprios algoritmos de busca ou adaptar os existentes.

Aborda-se aqui os problemas mais comuns em livros de IA (HAYKIN, 2000; RUSSEL; NORVIG, 1995): o Problema do Caixeiro Viajante, do Labirinto e o do Quebra-Cabeça de N peças. Foram desenvolvidos AVs que representam graficamente cada um destes problemas e, por terem seu código-fonte aberto, permitem a fácil alteração ou desenvolvimento de algoritmos que os resolvam.

Os AVs deverão conter pelo menos quatro algoritmos de busca (BFS, DFS, UCS e

A*), e apresentar passo-a-passo a execução de cada um destes algoritmos. O usuário deve ser capaz de inserir o Estado Inicial e acompanhar métricas, tais como número de estados avaliados e espaço ocupado em memória, durante a execução do algoritmo. Este trabalho apresentará a estrutura e o funcionamento de cada um dos AVs.

As demais seções deste trabalho estão estruturadas como segue: na seção 2 são apresentados brevemente os algoritmos de busca mais comuns, e o seu funcionamento. Os AVs desenvolvidos são expostos na seção 3, e sua forma de utilização, com exemplos de funcionamento e de avaliação de desempenho, é discutida na seção 4. Por fim, na seção 5, estão as principais conclusões obtidas.

2. SOLUÇÃO DE PROBLEMAS POR MEIO DE ALGORITMOS DE BUSCA

A solução de problemas por meio de algoritmos de busca é em geral o primeiro e mais simples método de solução de problemas das ementas dos cursos introdutórios de IA. Nele, são apresentados vários algoritmos capazes de, através da observação do espaço de estados, encontrar um conjunto de passos que, a partir de um estado inicial, chegue até um estado final desejado (RUSSEL; NORVIG, 1995).

Após a execução do algoritmo, espera-se encontrar uma solução, que é o conjunto de ações que levem o problema de um estado inicial até um estado objetivo. Pode-se, por exemplo, utilizar-se de estrutura de dados em Árvore, chamada de Árvore de Busca, para armazenar, de modo organizado, o Espaço de Estados do problema. Nesse contexto, a exploração de Árvores de Busca consiste num método eficiente utilizado para encontrar uma solução particular dentro de uma grande quantidade de dados.

A essência da solução de problemas por meio de busca é percorrer a Árvore de Busca de maneira organizada, até que seja encontrado um caminho, ou seja, uma sequência de ações que leve o problema do estado inicial ao estado objetivo.

Nem sempre a primeira solução encontrada é a mais eficiente dentre todas as soluções possíveis: problemas clássicos de IA são, em geral, do tipo NP (*Non-Deterministic Poynomial time)*. Difícil, e, portanto, não se pode verificar, em tempo polinomial, se uma solução é ótima (RUSSEL; NORVIG, 1995). Deste modo, muitas vezes a busca pela solução ótima exige que todo o espaço de estados do problema seja explorado (KIRKPATRICK e colab., 1983).

A avaliação de desempenho de métodos que utilizam Árvores de Busca é realizada através de parâmetros como a complexidade de tempo e de espaço, medidos em termos do fator de ramificação (*Branching Factor,* ou *b*), que corresponde à média de operadores que

podem ser aplicados a qualquer estado, gerando B filhos, e da profundidade (*depth*, ou *d*), que determina a quantidade de passos para se chegar a um estado, seja ele final ou não, à partir do estado inicial (KORF; SCHULTZE, 2005).

O desempenho de um Algoritmo de Busca também depende de fatores como a completeza, que garante sempre encontrar uma solução, se esta existir, e a otimização, que garante que a solução encontrada seja ótima.

As subseções abaixo apresentam algumas estratégias de busca por soluções.

2.1 BUSCA: NÃO-INFORMADA E INFORMADA

Busca Não-Informada é um meio de resolução de problemas onde a ideia é transformar um problema de raciocínio em um problema de navegação no espaço de estados. Neste tipo de algoritmo não se sabe qual é o melhor nó a ser expandido. As principais estratégias para definir qual nó deve ser expandido são as buscas em largura, em profundidade, e de custo uniforme (RUSSEL; NORVIG, 1995).

A busca em largura (*Breadth-First Search*, ou BFS) expande sempre o nó menos profundo ainda não expandido, ou seja, consiste em explorar todos os nós existentes no nível de profundidade atual antes de avaliar o próximo nível. O algoritmo BFS utiliza uma estrutura de dados FIFO (*First In, First Out*), onde o primeiro nó armazenado é o primeiro a ser expandido (SILVA; PARPINELLI, 2015).

A busca em profundidade (*Depth First Search*, ou DFS) expande sempre o nó mais profundo ainda não expandido. Quando um nó que não pode ser expandido (nó folha) é encontrado, o algoritmo retorna ao nó mais profundo ainda não explorado, recomeçando a busca a partir dele.

A busca de custo uniforme (*Uniform Cost Search*, ou UCS) é uma variação da busca em largura, porém expande primeiramente o nó ainda não expandido que possui menor custo. O custo de caminho de cada nó é calculado por uma função *g(n)*, determinada a partir da soma dos custos de caminho do estado inicial até o estado atual (SILVA; PARPINELLI, 2015).

A Busca Informada, por outro lado, estima o melhor nó a ser expandido de acordo com uma função heurística, usando atalhos para eliminar subespaços sem precisar explorá-los. Este tipo de busca é capaz de encontrar soluções de forma mais eficiente que uma estratégia sem informação, porque este tipo de busca utiliza o conhecimento específico do problema.

Uma heurística é uma função denominada *h(n),* onde *h* é uma estimativa de custo do caminho do estado atual ao estado objetivo. As principais estratégias de busca informada são a busca gulosa, e o algoritmo A* (RUSSEL; NORVIG, 1995).

O algoritmo de busca gulosa (*Greedy Best First Search,* ou GBFS) tem como estratégia expandir o nó mais próximo da origem de acordo com uma heurística, mas sofre dos mesmos problemas enfrentados pela busca de custo uniforme, pois podem ocorrer caminhos redundantes e, assim, *a* busca pode não ser completa. A solução encontrada pela busca gulosa nem sempre é ótima, uma vez que esta estratégia sempre expande o nó mais perto do objetivo, sem levar em consideração o custo do caminho (ROCHA; DORINI, 2004).

O algoritmo A* busca expandir o nó com menor custo, representado por função *f(n)* que corresponde à soma do custo de caminho percorrido *g(n)* e o valor da heurística *h(n)*. Este algoritmo garante encontrar a melhor solução caso a heurística seja admissível. (RUSSEL; NORVIG, 1995). Este algoritmo combina a vantagem do BFS, que garante explorar todo o espaço de estados sem caminhos redundantes, juntamente com os benefícios do UCS, que encontra rapidamente uma solução. Desta forma, o A* sempre tentará minimizar a função *f(n),* para determinar o menor caminho ao estado objetivo (SILVA; PARPINELLI, 2015). A desvantagem do algoritmo A* está no gasto de memória, pois esta técnica exige o armazenamento de todos os nós anteriores ao atual.

3. AMBIENTES VIRTUAIS DESENVOLVIDOS

Os AVs desenvolvidos permitem diversas personalizações pelo usuário, como a implementação de novos algoritmos ou heurísticas. Apesar de simples, a busca por soluções aos problemas apresentados requer quantia razoável de recursos computacionais, e os AVs devem prover uma boa experiência de usuário mesmo sob condições de estresse.

3.1 PROBLEMA DO CAIXEIRO VIAJANTE

O Problema do Caixeiro Viajante (TSP, do inglês *Travelling Salesman Problem*) é um dos mais famosos e importantes problemas da área de IA. Este é um problema de roteamento, que essencialmente busca encontrar o menor percurso fechado entre um conjunto de pontos.

O TSP é do tipo NP-difícil e possui complexidade $O(N!)$. Comumente, o TSP é modelado com grafos ponderados não-direcionais onde, cada cidade (ponto) é um vértice, as arestas são os caminhos e o peso é a distância entre duas cidades. Com isto, este problema se

torna um problema de minimização de custo de caminho, que começa e termina no mesmo vértice, após visitar cada nó uma única vez.

O AV desenvolvido possui um *framework* desenvolvido em *JavaScript* que certamente o faz extremamente útil para o ensino de IA, pois fornece ao usuário um conjunto de métodos que podem ser utilizados para implementar algoritmos de busca compatíveis com a interface, baseados na sintaxe de pseudocódigo do livro *Artificial Intelligence: A Modern Approach* (RUSSEL; NORVIG, 1995).

Conforme ilustrado na Figura 1, o AV desenvolvido utiliza a representação de grafos como forma de modelagem do problema. Graficamente, os vértices são apresentados no primeiro quadrante do plano cartesiano.

Cada vértice pode ser movido livremente dentro do espaço disponível, mostrado na Figura 1 (A) e a quantidade de cidade é variável, conforme Figura 1 (C), desde que obedeça ao mínimo de 4 vértices. A quantidade máxima de cidades é, teoricamente, infinita, entretanto esta foi limitada em 20 para melhor experiência do usuário.

Estão disponíveis inicialmente cinco algoritmos: BFS, DFS, UCS e A*, escolhidos na posição indicada pela Figura 1 (C). É também fornecida a heurística 2-*opt*, que pode ser selecionada na Figura 1 (F). Esta heurística tem por objetivo reduzir o número de cruzamentos entre as arestas do caminho. Para isto, é atribuído um custo à quantidade de intersecções existentes entre os caminhos, e este custo é minimizado durante a execução do algoritmo A*.

A implementação dos algoritmos de busca é realizada de maneira iterativa, o que permite ao usuário total controle sobre a execução. A iteração pode ser feita de maneira manual ou automática, e caso a execução seja automática, esta pode ser pausada e reiniciada a qualquer momento. Isso possibilita ao usuário visualizar passo a passo a execução do algoritmo, e assim compreender as características fundamentais de cada um deles.

O uso de recursos computacionais e a convergência do algoritmo podem ser vistas numérica ou graficamente, no local indicado pelas Figura 1 (D) e Figura 1 (E), respectivamente. São apresentadas informações da distância do caminho avaliado atualmente, a taxa de convergência do algoritmo, a quantidade de estados em memória e a taxa de processamento de estados, permitindo ao usuário comparar a eficácia, completeza e otimização de cada um dos algoritmos implementados.

No painel mostrado na Figura 1 (B) o usuário é capaz de controlar a execução do algoritmo, podendo iniciar, pausar, parar ou avançar passo a passo o algoritmo. Para permitir este controle de execução, é necessário implementar apenas dois métodos: a inicialização e a

iteração. As estruturas de dados necessárias são fornecidas, e os valores nela armazenados são utilizados para a alteração do mapa.

No método de inicialização, cabe ao usuário inicializar quaisquer ferramentas que serão utilizadas a cada iteração. O método de iteração é responsável por controlar a execução do algoritmo propriamente dito.

O problema fornece a estrutura de dados *problem.frontier*, que pode ser utilizada como pilha ou fila, e armazena a ordem dos estados a serem explorados. São fornecidos também os métodos *problem.getPath* e *problem.goalCheck*, que, ao receberem o estado atual, respectivamente, retornam o caminho percorrido e se o estado atual é o objetivo.

Para se adicionar um novo algoritmo ou heurística, o usuário deverá criar um novo arquivo na pasta *algorithms* e implementar uma interface contendo dois parâmetros, *name*, que é um texto correspondente ao nome do algoritmo e *use_heuristics*, que indica se este algoritmo requer ou não uma heurística, e dois métodos: *init* e *step*, que correspondem à inicialização e iteração daquele algoritmo. Após a implementação, deverá ser chamada a função *add_algorithm*, que receberá estes quatro parâmetros e permitirá a chamada deste algoritmo na interface.

Ao final da execução de cada iteração, o algoritmo deve retornar o nó atualmente avaliado, para que a interface seja atualizada com as informações atuais da execução. Essa abstração em se desenhar em tela simplifica o desenvolvimento de algoritmos pelo usuário, permitindo assim uma melhor compreensão do algoritmo implementado.

A implementação de uma nova heurística se dá de modo semelhante à implementação de um algoritmo: inicialmente devem ser instanciados o nome da heurística e os métodos de início e de interação. Ao final da iteração do algoritmo de busca, o algoritmo de heurística receberá o estado atual e todos os estados possíveis, e a partir disso recalcula a heurística para todos os nós

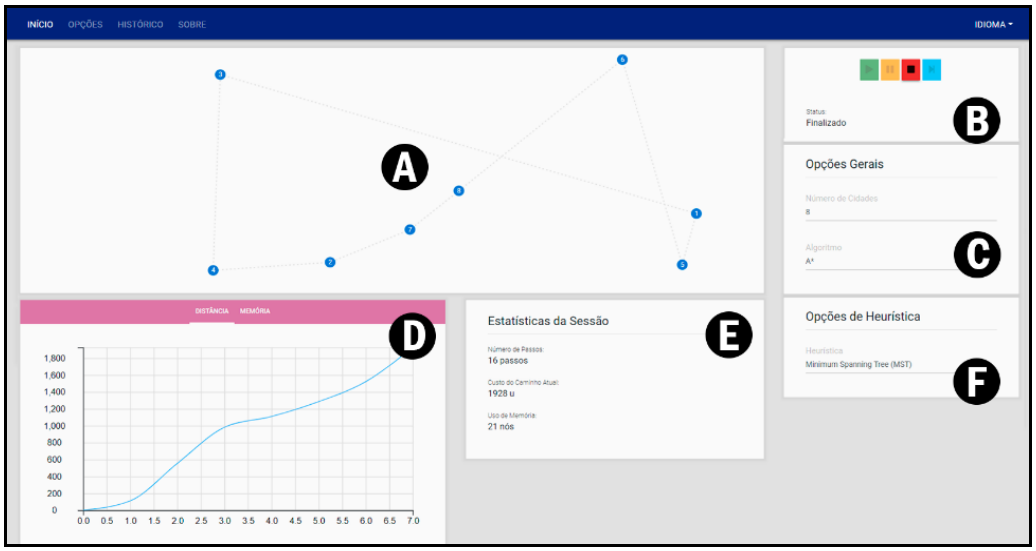
3.2 PROBLEMA DO LABIRINTO

Outra aplicação clássica de algoritmos de busca corresponde ao problema de se encontrar a saída de um labirinto (GOLDBARG; GOLDBARG, 2012; MISHRA, 2008). O objetivo deste problema é encontrar um caminho que chegue ao estado final a partir de um estado inicial. Um agente de resolução de problemas não é capaz de observar ou passar através das paredes. Caso se encontre cercado por paredes, o agente precisa retornar. Se não houver caminho entre o estado inicial e o estado final, a execução não terá sucesso.

O AV para o problema do labirinto foi desenvolvido utilizando o motor de interfaces *Angular 7* baseado em *TypeScript* (GOOGLE, 2018). A modelagem do labirinto no AV é realizada através de uma matriz com dimensões variáveis, onde cada célula corresponde a um nó do espaço de estados.

Neste AV, cada célula pode ser definida pelo usuário como caminho, parede, início ou fim. Visualmente, conforme apresentado na Figura 2 (A), as células em cinza representam as paredes do labirinto e as brancas representam o caminho livre. O início e o fim são definidos pelas cores verde e vermelho respectivamente.

Figura 1 - Tela do AV para o problema do caixeiro viajante. (A) Mapa das cidades; (B) Opções de execução; (C) Opções de algoritmo; (D) Gráficos de convergência; (E) Estatísticas de execução. (F) Opções de heurística.



Fonte: Desenvolvido pelos autores.

A interface desenvolvida permite ao usuário desenhar o labirinto de forma dinâmica, através do uso do *mouse* na região indicada na Figura 2 (A). Também é possível abrir um arquivo JSON contendo um modelo de labirinto salvo previamente pelo usuário, por meio da aba de opções, mostrada na Figura 2 (B).

Para o cálculo da solução, quatro algoritmos de busca estão disponíveis: BFS, DFS, UCS e A*. Também são implementadas as heurísticas da Distância Euclidiana e Distância de Manhattan. No local apresentado na Figura 2 (C) o usuário define o algoritmo de busca que deseja utilizar para encontrar a solução do labirinto. Caso o usuário selecione um algoritmo de busca informada, é necessário também, que seja indica uma heurística.

Na aba de execução, mostrada na Figura 2 (D), o usuário pode controlar o processo de busca, executado pelo algoritmo selecionado, que pode ser interrompido e reinicializado a qualquer instante.

Assim como para o problema do caixeiro viajante, este ambiente virtual foi desenvolvido com foco em uma execução flexível e didática, que permite ao usuário o controle total do processo de execução.

Por ser desenvolvida usando o motor de interfaces *Angular 7*, o código fonte da interface pode ser facilmente alterado para se adicionar mais funcionalidades gráficas, ou mesmo adicionar novos algoritmos ou heurísticas, visto que o código fonte é ricamente comentado.

Para que o usuário implemente novos algoritmos ou heurísticas, deve ser alterado o serviço *algoritmos.service*. O desenvolvimento de um novo algoritmo de busca requer a implementação de dois métodos: o método de inicialização e o de iteração.

O método de inicialização deve instanciar a estrutura de dados responsável por armazenar o espaço de estados explorado, contendo inicialmente o estado inicial. Ao final deste método, é necessário realizar uma chamada ao método de iteração, que dará início a execução do algoritmo.

No método de iteração é definido o código do algoritmo de busca, que será executado com base nas informações do espaço de estados explorado. São fornecidas pelo AV métodos que avaliam os movimentos possíveis de serem realizados pelo agente de busca e verificam se o nó atual avaliado é o estado objetivo.

A implementação de uma nova heurística também deve ser realizada no serviço *algoritmos.service*. A função *gerarHeurística* realiza uma chamada a um método responsável por calcular o valor da heurística para um nó, de acordo com a seleção realizada pelo usuário.

3.3 PROBLEMA DO QUEBRA-CABEÇA DE N PEÇAS

O Quebra-Cabeça de N Peças é um problema no qual apenas uma peça pode ser movida por vez. Esta peça pode ser deslocada apenas em uma posição, na vertical ou na horizontal. Após o movimento, a peça movida troca de lugar com a peça que ocupa o local pretendido.

Este problema possui dois estados importantes: o estado inicial e o estado final. O objetivo é atingir o estado final a partir do estado inicial através de uma série de movimentos.

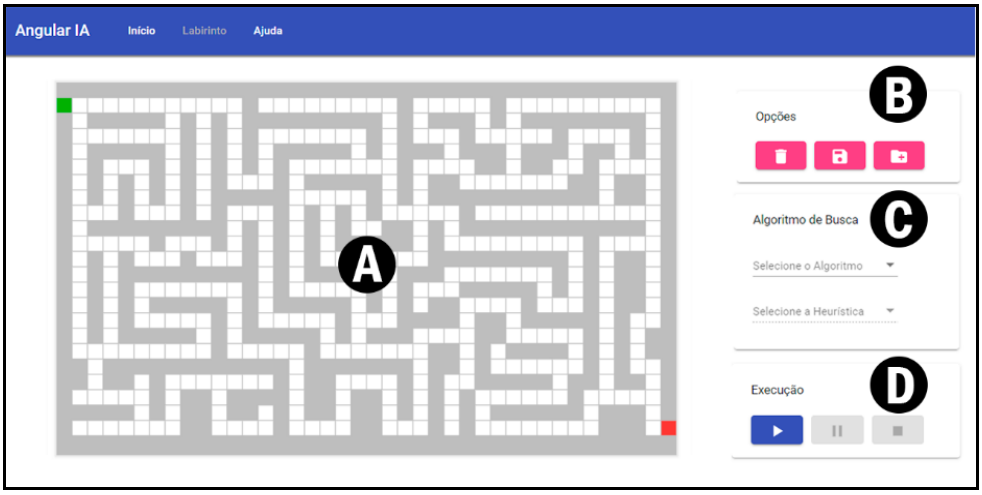
A interface gráfica da implementação do AV para o problema do Quebra-Cabeça de N Peças pode ser vista na Figura 3. A peça que pode ser deslocada possui cor mais escura que as demais. Os possíveis movimentos também são destacados, porém, com uma cor mais clara.

O AV foi desenvolvido em Python e faz uso da biblioteca PyQt, que é uma das ligações mais populares para a framework *cross-plataform* em C++ do *Qt Framework.*

É possível modificar o estado final e inicial. Para isso, é necessário configurar o tabuleiro na configuração do estado final desejado e pressionar o botão *Set Final State* e o estado representado no tabuleiro será definido como estado objetivo.

Há também dois modos de configuração do tabuleiro, *Play Mode* e *Edit Mode.* No primeiro, o deslocamento das peças é realizado de acordo com as regras originais do problema, isto é, deslocando a peça na horizontal ou na vertical deslocando apenas uma posição no tabuleiro em relação à posição atual. No segundo, o valor da peça pode ser atribuído livremente, independente das restrições, bastando clicar no local desejado e selecionar o valor pretendido para esta peça que o valor será atribuído a ela.

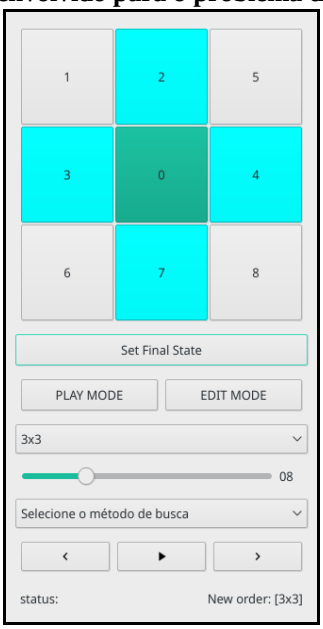
Figura 2 - Imagem do AV desenvolvido para o problema do Labirinto.
(A) Labirinto; (B) Opções do labirinto; (C) Opções do algoritmo e de execução;



Fonte: Desenvolvido pelos autores.

É possível também alterar a ordem de complexidade do tabuleiro, que pode variar de 3 a 9. Esta limitação é apenas para melhor a experiência do usuário, pois o algoritmo é capaz de lidar com qualquer ordem, desde que haja disponibilidade de recursos de armazenamento e processamento.

Figura 3 - Imagem do AV desenvolvido para o problema do Quebra-Cabeça de N Peças.



Fonte: Desenvolvido pelos autores.

Estão disponíveis os algoritmos de busca BFS, DFS e A*, com as heurísticas: número de peças Fora do lugar e Distância de Manhattan. Alguns algoritmos, como o DFS, possuem opções adicionais, como a limitação da profundidade de busca, que pode ser configurada através de uma barra deslizante, a qual assume valores inteiros de 1 até 30, conforme mostrado na Figura 3. Após encontrar a solução, é possível executá-la passo a passo, avançando ou retrocedendo.

Ao se deslocar pela solução, o tabuleiro é modificado para ilustrar o estado de transição atual, facilitando a compreensão da natureza do algoritmo de busca selecionado. Também são mostrados dados sobre o número de nós expandidos e o tempo gasto até a solução encontrada.

4. RESULTADOS E DISCUSSÕES

Esta seção apresenta as especificidades dos Ambientes Virtuais aqui propostos, bem como a avaliação das otimizações propostas para o problema do Quebra-Cabeça de N Peças.

4.1 PROBLEMA DO CAIXEIRO VIAJANTE

O AV desenvolvido para representar o Problema do Caixeiro Viajante possui por padrão uma grande quantidade de algoritmos de busca já implementados. Esta diversidade é

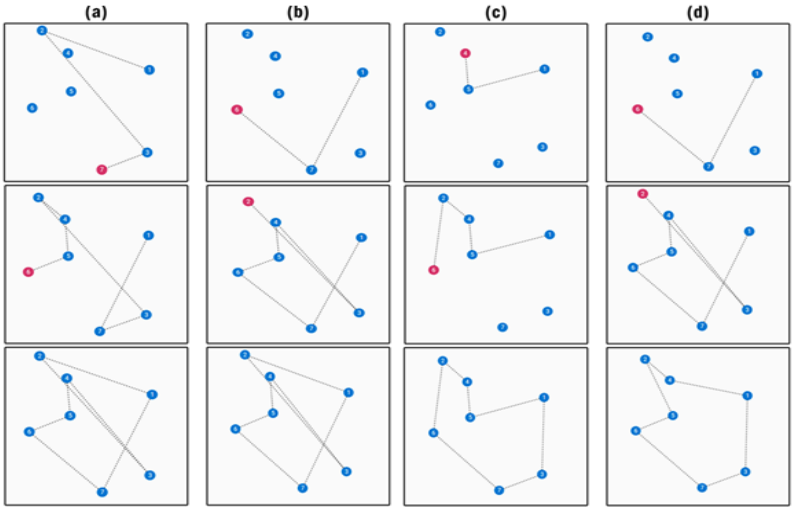
importante para que o usuário possa experimentar com diversos mapas e algoritmos e, assim, compreender bem o funcionamento de cada algoritmo.

Os algoritmos de busca Não-Informada são ineficientes, e assim, a execução completa de algoritmos DFS e BFS, para o cenário com 20 cidades, pode durar muitas horas para completar a execução e pode causar *overflow* de memória, dependendo do computador utilizado. Este cenário é essencial para que o usuário compreenda bem os problemas de algoritmos de busca clássica. A busca informada, por outro lado, é capaz de encontrar rapidamente um caminho, ótimo mesmo no pior cenário, com 20 cidades.

Foram então elaborados dois cenários de teste: um com 7 cidades e outro com 20 cidades. No primeiro cenário, o algoritmo BFS, cuja solução está apresentada na Figura 4 (a), necessitou de 235 segundos para encontrar a primeira solução. Esta não era uma solução otimizada. Os algoritmos DFS e UCS levaram 3 segundos para encontrar a primeira solução não otimizadas (Figura 4 (b) e Figura 4 (c)).

O algoritmo A* gastou 5 segundos para encontrar o melhor caminho, sendo este o único com solução ótima. As demais informações de execução dessas soluções estão apresentadas na Tabela 1. Nota-se que a execução dos algoritmos BFS e DFS resultou no mesmo caminho, mas o algoritmo BFS necessitou de 217 vezes mais passos e 45 vezes mais memória que o DFS.

Figura 4 - Imagem do AV desenvolvido para o problema do caixeiro viajante. Cenário 1: 7 cidades. (a) Resolução por BFS. (b) Resolução por DFS. (c) Resolução por UCS. (d) Resolução por A*



Fonte: Desenvolvido pelos autores.

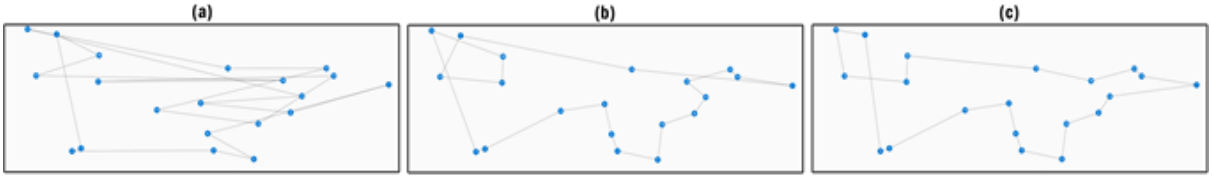
No segundo cenário, de pior caso, contendo 20 cidades, o algoritmo BFS não foi executado, visto que após 2 horas de execução, com mais 200.000 nós em memória, não havia sido encontrada uma solução. Os resultados de execução deste cenário estão expostos numericamente na Tabela 2, e graficamente na Figura 5.

Tabela 1 Resultados dos testes do AV desenvolvido para o problema do caixeiro viajante. Cenário 1: 7 cidades.

Algoritmo	Nº de Passos	Uso de Memória	Custo da Solução
BFS	1 959	720 nós	2 224 u
DFS	9	16 nós	2 224 u
UCS	9	16 nós	1 719 u
A*	13	16 nós	1 458 u

Fonte: Desenvolvido pelos autores.

Figura 5 - Imagem do AV desenvolvido para o problema do caixeiro viajante. Cenário 1: 7 cidades. (a) Resolução por BFS. (b) Resolução por DFS. (c) Resolução por UCS. (d) Resolução por A*



Fonte: Desenvolvido pelos autores.

Como esperado, o algoritmo A* é capaz de encontrar caminhos otimizados com maior eficácia, mas necessita de cerca de 4 vezes mais processamento. Os algoritmos UCS e DFS são capazes de encontrar rapidamente uma solução para o problema, mas possuem custos elevados: 2.66 vezes maior para o caso do DFS, e 1,15 vezes maior para o caso do UCS. Com isto, este cenário é capaz de demonstrar as vantagens e falhas de cada algoritmo, ajudando o usuário a compreender experimentalmente qual é o melhor método para resolução de um problema.

Tabela 2 Resultados dos testes do AV desenvolvido para o problema do caixeiro viajante. Cenário 2: 20 cidades.

Algoritmo	Nº de Passos	Uso de Memória	Custo da Solução
BFS	+50 000	+200 000 nós	?
DFS	22	172 nós	14 963 u
UCS	22	172 nós	6 458 u
A*	98	172 nós	5 615 u

Fonte: Desenvolvido pelos autores.

4.2 PROBLEMA DO LABIRINTO

O AV para o problema do labirinto é o que gera menor espaço de estados neste trabalho. Assim, pode-se criar mapas com maior quantidade de nós e verificar o impacto no tempo de execução, pois mesmo em cenários de pior caso, a execução finda em poucos minutos.

O AV desenvolvido foi avaliado com o modelo de labirinto ilustrado na Figura 2, para cada algoritmo implementado (BFS, DFS, UCS e A*). Para cada um dos algoritmos, o trajeto traçado é diferente, de modo que o usuário entenda, ainda que empiricamente, o modo de funcionamento do algoritmo. Como mostrado na Figura 6, a solução encontrada, corresponde à cor rosa no labirinto e as células visitadas pelo agente de busca são representadas pelo azul.

A Figura 6 (A) mostra que o algoritmo de busca BFS explorou quase que todos os caminhos do Labirinto, porque o BFS avalia todos os nós em uma mesma profundidade, que neste caso corresponde a avaliar todos os nós equidistantes à posição inicial antes de prosseguir.

O DFS, mostrado na Figura 6 (B), por explorar os nós em profundidade, escolhe um caminho e o percorre até o final, e se não encontrar uma solução, escolhe outro caminho. Então, apesar da busca em profundidade explorar menos nós que o BFS, nota-se que a solução encontrada possui um custo de caminho maior que a encontrada pelo BFS.

No caso do A*, apesar de ser semelhante ao UCS na implementação de código, há o uso de uma heurística somada ao custo do caminho. A Figura 6 (D) e a Figura 6 (E) mostram a execução deste algoritmo utilizando as heurísticas de Manhattan e Euclidiana, respectivamente.

Como o custo de um movimento no problema do labirinto é fixo, sempre igual a um, o algoritmo UCS é equivalente ao BFS. Isto ocorre porque a principal característica do algoritmo UCS é explorar sempre o nó com menor custo de caminho e, assim, segue uma ordem igual ou parecida à do algoritmo BFS. A Figura 6 (C) mostra passo a passo a execução do UCS.

A Tabela 3 mostra o total de nós expandidos na solução de cada algoritmo juntamente com a quantidade total de estados visitados no espaço de estados. Como esperado, o algoritmo DFS possui menor quantidade de nós explorados, mas a solução não é ótima.

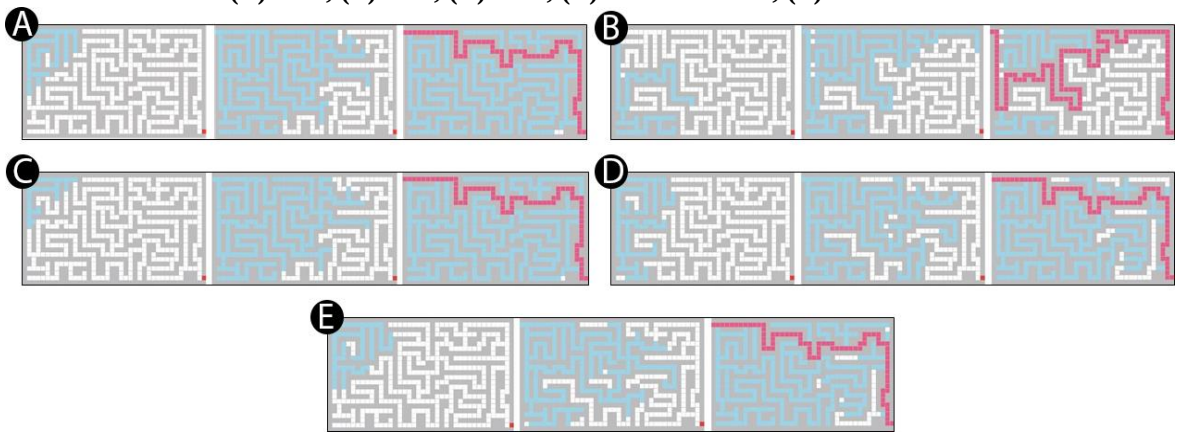
Tabela 3 - Resultados dos testes para cada algoritmo de busca

Algoritmo	Profundidade	Solução
BFS	446	81
DFS	236	139
UCS	447	81
A* Manhattan	389	81
A* Euclidiana	415	81

Fonte: Desenvolvido pelos autores.

Todos os outros algoritmos encontraram a mesma solução, ótima, e o A*, utilizando heurística de Manhattan, gastou o menor tempo para isso.

Figura 6 - Imagem do AV desenvolvido para o problema do labirinto (A) BFS; (B) DFS; (C) UCS; (D) A* Manhattan; (E) A* Euclidiana.



Fonte: Desenvolvido pelos autores.

4.3 PROBLEMA DO QUEBRA-CABEÇA DE 9 PEÇAS

O problema do Quebra-Cabeça de N Peças é um dos problemas mais utilizados para a demonstração de algoritmos de busca em IA. Conforme já mencionado, trata-se de um problema NP-Difícil. Por esta razão, algoritmos de busca, em especial os de busca Não-Informada, não são eficazes para este problema em cenários com espaço de estados muito profundos. Por isso, para este problema optou-se por explorar as otimizações possíveis para a redução de processamento, de modo que o problema possa ser resolvido para grandes espaços de estados. Os autores entendem que a compreensão dessas estratégias de otimização é também muito importante para o aprendizado do usuário acerca das otimizações de implementação.

Uma forma de otimizar o desempenho de algoritmos de busca é através da não expansão de estados já encontrados em outros caminhos. Porém, nesse caso surge a questão de qual dos caminhos deve ser propagado quando eles alcançarem um estado comum, o que depende do algoritmo de busca utilizado. Para o BFS, o caminho que atingir primeiro o estado em questão deve ser mantido, e todos os demais caminhos, que também atingirem este estado, podem ser descartados sem prejuízo.

No caso do DFS, o caminho que atingir o estado com a menor profundidade (a menor quantidade de movimentos) deve ser mantido e os demais podem ser descartados, independentemente de quem o atingiu primeiro, e isso não irá interferir na solução.

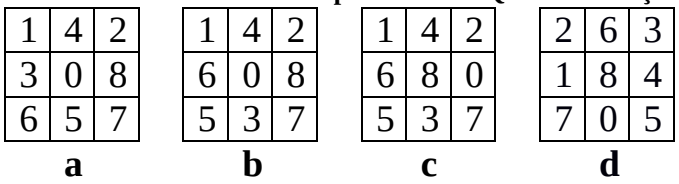
No caso do algoritmo A*, com qualquer heurística, o caminho que alcança o estado repetido com a menor soma, do custo e da heurística, deve ser mantido. Todas os demais caminhos podem ser descartados sem prejuízo para a solução.

Para impedir a propagação de estados já conhecidos, é necessário verificar qual estado já foi visitado. Como não há nenhuma forma de se comparar dois estados através de uma relação "*maior que"* ou "*menor que*", não é possível ordenar os estados já conhecidos de forma a reduzir o tempo de verificação de estados já processados. O tempo necessário para se realizar a verificação de estados cresce proporcionalmente com a quantidade de nós expandidos pelo algoritmo de busca utilizado. Somado a isso, a quantidade de verificações de nós com potenciais para serem inseridos na borda, cresce de acordo com número de novos estados que precisam ser avaliados.

Portanto, para se reduzir o tempo de busca foi utilizado uma estrutura de *hash*, cuja função de espalhamento têm como argumento o estado do tabuleiro e retorna o *hash* associado a esse estado. A principal vantagem das tabelas de *hash* sobre outras estruturas de dados é a velocidade. O tempo de busca utilizando tabelas *hash* é idealmente constante (CORMEN e colab., 2009). A Tabela 4 apresenta esses resultados, considerando os estados iniciais a, b, c e d mostrados na Figura 7.

De acordo com a Tabela 4, há redução efetiva no número de nós gerados quando não se permite estados repetidos. A redução do número de nós para os algoritmos de busca cega é considerável, chegando a até 10 vezes menos nós, se comparado com as implementações em que não há bloqueio de estados já analisados.

Figura 7 - Estados iniciais avaliados no problema do Quebra-Cabeça de 8 peças.



Fonte: Desenvolvido pelos autores.

Para os algoritmos de busca informada, em sua maioria, não houve grandes vantagens porque esta classe de algoritmos já evita repetições implicitamente.

A próxima avaliação dedicou-se a aferir o impacto da utilização da tabela *hash (c),* na redução do escopo de estados, visando reduzir o tempo de busca. A Figura 8 apresenta os resultados para um cenário em que todos os algoritmos de busca implementados foram avaliados, considerando o estado inicial da Figura 7 (d) que demanda no mínimo 18 movimentos para a solução, ou seja, um problema cuja solução requer elevado processamento. Os resultados, com diferentes tamanhos de *hash*, confirmam que o tamanho

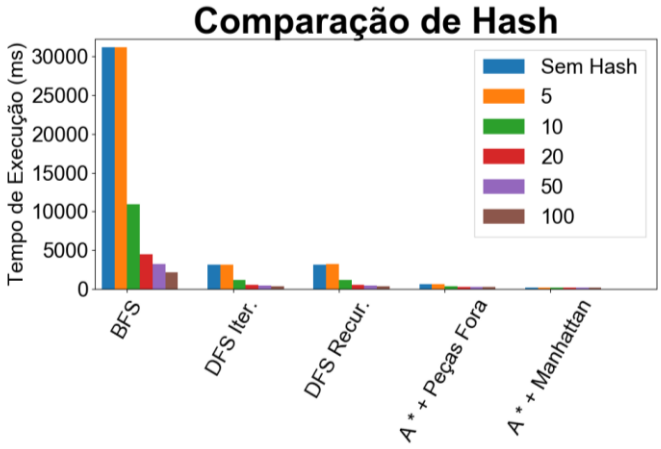
da tabela é inversamente proporcional ao tempo de busca, para todos os algoritmos de busca. Esses resultados podem ser facilmente obtidos pelo usuário que utilizar esse AV, e certamente o ajudará a assimilar a importância das otimizações nos problemas de busca em IA.

Tabela 4 - Comparação de desempenho para os algoritmos utilizados.

Estado	Algoritmo de Busca	Nós gerados (Com repetição)	Nós gerados (Sem repetição)
a	BFS	751	96
	DFS iterativo (Profundidade 9)	1 477	165
	DFS recursivo (Profundidade 9)	1 470	161
	A* + Peças Fora do Lugar	28	26
	A* + Distância de Manhattan	25	24
b	BFS	45 295	669
	DFS iterativo (Profundidade 10)	29 805	389
	DFS recursivo (Profundidade 10)	29 795	383
	A* + Peças Fora do Lugar	122	122
	A* + Distância de Manhattan	44	44
c	BFS	120 194	1 162
	DFS iterativo (Profundidade 11)	104 704	1 105
	DFS recursivo (Profundidade 11)	104 694	1 101
	A* + Peças Fora do Lugar	498	171
	A* + Distância de Manhattan	50	47

Fonte: Desenvolvido pelos autores.

Figura 8 - Comparação de desempenho dos algoritmos de busca com diferentes comprimentos de *hash*.



Fonte: Desenvolvido pelos autores.

5. CONSIDERAÇÕES FINAIS

Este trabalho apresentou três Ambientes Virtuais (AVs) desenvolvidos para aprimorar o processo de ensino-aprendizagem, do tópico de resolução de problemas por meio de busca, em cursos introdutórios à IA. Esses ambientes são altamente personalizáveis, e possuem

visualizações didáticas da execução de algoritmos em problemas clássicos da área: o problema do Caixeiro Viajante, o do Labirinto e o do Quebra-Cabeça de N Peças.

Os AVs desenvolvidos para os problemas do caixeiro viajante e do labirinto oferecem um conjunto de ferramentas que facilitam a compreensão de algoritmos de busca Não-Informada e Informada, sem otimizações. Estes AVs possuem também um ambiente de programação que permite aos usuários implementar seus próprios algoritmos e heurísticas, de modo muito intuitivo que visam facilitar a aprendizagem do assunto.

Já o AV para o problema do Quebra-Cabeça de N Peças foi desenvolvido com o propósito de exemplificar o impacto das otimizações de desempenho para algoritmos de busca. Este ambiente possui diversas configurações de desempenho, como heurísticas variadas, e a não repetição de estados já conhecidos, o que tem o potencial para simplificar a compreensão da importância do uso dessas técnicas na resolução de problemas reais, em termos de demanda de processamento e memória.

O principal diferencial dos AVs desenvolvidos neste trabalho em relação a outros existentes na literatura está na forma como eles foram concebidos. Os AVs elaborados neste trabalho pressupõem que o usuário fará a sua própria implementação e verá os resultados no AV. Ademais, todos os códigos aqui criados estão disponibilizados na web para livre download.

Por fim, ressalta-se que embora os AVs deste trabalho tenham sido desenvolvidos para aprimorar a assimilação dos algoritmos de busca em cursos de IA, nesta fase não foi possível avaliar a sua eficácia com muitos estudantes, o que deverá ser feito na próxima fase deste trabalho. Espera-se, ainda, desenvolver AVs mais avançados baseados em Algoritmos Genéticos, Classificadores e Redes Neurais.

REFERÊNCIAS

AIAI. *MissionariesAndCannibalsApp*. . [S.l.]: Artificial Intelligence Applications Institute. School of Informatics. The University of Edinburgh. , 2011a

AIAI. *TravellingSalesmanApp*. . [S.l.]: Artificial Intelligence Applications Institute. School of Informatics. The University of Edinburgh. , 2011b

ALI, Zeyad; SAMAKA, Mohammad. ePBL: Design and Implementation of a Problem- based Learning Environment. 2013, [S.l: s.n.], 2013. p. 1209–1216.

BARELLA, Antonio; VALERO, Soledad; CARRASCOSA, Carlos. JGOMAS: New approach to AI teaching. *IEEE Transactions on Education*, v. 52, n. 2, p. 228–235, 2009.

BRYUKH. *Labyrinth Algorithms*.

CORMEN, Thomas H. e colab. *Introduction to Algorithms*. 3rd. ed. [S.l.]: MIT Press, 2009.

CRESPO GARCÍA, Raquel M.; ROMÁN, Julio Villena; PARDO, Abelardo. Peer review to improve artificial intelligence teaching. *Proceedings - Frontiers in Education Conference, FIE*, p. 3–8, 2006.

DENERO, John; KLEIN, Dan. Teaching Introductory Artificial Intelligence with Pac-Man. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, p. 1885–1889, 2010.

DOGMUS, Zeynep; ERDEM, Esra; PATOGLU, Volkan. ReAct!: An interactive educational tool for AI planning for robotics. *IEEE Transactions on Education*, v. 58, n. 1, p. 15–24, 2015.

DRAMAIX, Julien. *Sliding puzzle*.

FERNANDES, Marcelo A.C. Problem-based learning applied to the artificial intelligence course. *Computer Applications in Engineering Education*, v. 24, n. 3, p. 388–399, 2016.

GOLDBARG, Marco; GOLDBARG, Elizabeth. *Grafos: Conceitos, algoritmos e aplicações*. [S.l.]: Elsevier, 2012.

GOOGLE. *Angular*.

GRIVOKOSTOPOULOU, Foteini; PERIKOS, Isidoros; HATZILYGEROUDIS, Ioannis. Using semantic web technologies in a web based system for personalized learning AI course. *Proceedings - IEEE 6th International Conference on Technology for Education, T4E 2014*, p. 257–260, 2014.

HAYKIN, S. S. *Redes Neurais*. 2nd. ed. Porto Alegre: Bookman, 2000.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. *Optimization by Simulated Annealing*. . [S.l.]: SCIENCE. , 1983

KORF, Richard E; SCHULTZE, Peter. Large-Scale Parallel Breadth-First Search. p. 1380–1385, 2005.

LOTZ, Sebastian. *Travelling Salesman Problem*.

MARKOVIC, Marko e colab. INSOS - Educational system for teaching intelligent systems. *Computer Applications in Engineering Education*, v. 23, n. 2, p. 268–276, 2015.

MISHRA, Swati. Maze Solving Algorithms for Micro Mouse. p. 86–93, 2008.

PANTIC, Maja; ZWITSERLOOT, Reinier; GROOTJANS, R.J. Teaching Introductory Artificial Intelligence Using a Simple Agent Framework. *IEEE Transactions on Education*, v. 48, n. 3, p. 382–390, Ago 2005. Disponível em: <http://ieeexplore.ieee.org/document/1495645/>.

PEREIRA, Alice Theresinha Cybis; SCHMITT, Valdenise; DIAS, Maria Regina Álvares C. AVA: Ambientes Virtuais de Aprendizagem em Diferentes Contextos. *Livraria Cultura*, p. 232, 2007.

PINTO, L e colab. Simulador de Mercados Baseado em Jogos Evolucionários e Algoritmos Genéticos. p. 698–704, 2003.

ROCHA, Anderson; DORINI, Leyza Baldo. *Algoritmos gulosos: definições e aplicações*. . Campinas: [s.n.]. , 2004

RUSSEL, Stuart; NORVIG, Peter. *Artificial Intelligence: A Modern Approach*. 3rd. ed. [S.l: s.n.], 1995.

SILVA, Rafael Castro G; PARPINELLI, Rafael S. Analise do Desempenho de Algoritmos Clássicos de Busca de Caminho em Ambiente de Navegacão. 2015.

TRINDADE, Eder L e colab. Algoritmos de busca em tempo real aplicados a jogos digitais. n. September 2014, 2008.

YOON, Du Mim; KIM, Kyung Joong. Challenges and Opportunities in Game Artificial Intelligence Education Using Angry Birds. *IEEE Access*, v. 3, p. 793–804, 2015.