UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO

BEATRIZ PACHECO CAMPOS
GABRIEL FERNANDES RIBEIRO

PATH DISCOVERY: UMA NOVA ABORDAGEM SOBRE ALGORITMOS DE PATHFINDING

NITERÓI
2019

BEATRIZ PACHECO CAMPOS
GABRIEL FERNANDES RIBEIRO

PATH DISCOVERY: UMA NOVA ABORDAGEM SOBRE ALGORITMOS DE PATHFINDING

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense para obtenção do grau de Bacharel em Ciência da Computação.
Orientador: Flavio Luiz Seixas
Orientador: Troy Costa Kohwalter

NITERÓI
2019

BEATRIZ PACHECO CAMPOS
GABRIEL FERNANDES RIBEIRO

PATH DISCOVERY: UMA NOVA ABORDAGEM SOBRE ALGORITMOS DE PATHFINDING

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense para obtenção do grau de Bacharel em Ciência da Computação.
Orientador: Flavio Luiz Seixas
Orientador: Troy Costa Kohwalter

Data da Aprovação: _____

BANCA EXAMINADORA

_____
Prof. Dr. Flávio Luiz Seixas – Orientador - UFF

_____
Prof. Dr. Troy Costa Kohwalter – Orientador - UFF

_____
Prof. Dr. Esteban Walter Gonzalez Clua - UFF

_____
Prof. Dr. Leandro Augusto Frata Fernandes - UFF

NITERÓI
2019

AGRADECIMENTOS

Beatriz Pacheco Campos

À minha mãe Fabiane, que é a base da minha vida e também meu maior refúgio em todas as dificuldades. Ao meu pai Eric, que sempre me orientou acadêmica e profissionalmente e proporcionou toda base educacional que possuo hojes. Ao meu irmão Lucas, meu maior exemplo de vida e à minha gatinha e companheira Bella.

Às minhas avós Sylvia e Rosinha pelos ensinamentos da vida, dedicação para com a família e tanto amor. Aos meus avôs Pacheco e Luiz Olavo, que, mesmo não estando presentes em vida, tenho certeza que comemoram minhas conquistas.

À melhor amiga dessa vida, minha irmã de alma Amanda, por todo incentivo e ajuda quando mais precisei.

Por fim, aos meus colegas de trabalho, em especial Jackie, Silveira e Alex, por tamanha compreensão pelo tempo que tive que dedicar a este trabalho.

Gabriel Fernandes Ribeiro

Agradeço aos meus pais Olga e Jorge por moldarem o que sou hoje e terem feito de tudo ao seu alcance para que este momento chegasse.

À minha avó Lucilene e tia Lídia por sempre me apoiarem.

Ao Madruguinha por me fazer companhia em todas as noites em claro.

Aos professores Troy Kohwalter e Flavio Seixas pela orientação e disposição para concluirmos este trabalho.

Aos professores do Institudo de Computação, em especial Fabio Protti, Leonardo Murta e Aline Nascimento por todo o conhecimento passado e por não hesitarem em ajudar quando precisei.

À Beatriz, pela parceria nestes longos anos de curso e no desenvolvimento deste trabalho.

Ao Gerson Barbosa, que ajudou um estranho e nem sabia que o estava ajudando a se formar.

Também aos colegas da Dumativa pela compreensão e apoio nos momentos que precisei.

Por fim, aos meus amigos: Raiane, Yuri, Felipe, Jéssica, João, Matheus, Lucas, Karina, Diego, Camila, Gabriel, Manuella, René, Leon, Igor, Flávia, Thiago, Carolina,

são tantos… Eu sei que vocês duvidaram! E confesso que eu também! Mas este trabalho não seria possível sem vocês!

RESUMO

Um dos crescentes desafios na área de jogos é relacionado a movimentação de personagens secundários. Para isso, são utilizados algoritmos de Pathfinding que determinam, por vezes o melhor, caminho entre um ponto de partida e um ponto de chegada. No entanto, para obtenção desses resultados a partir de algoritmos conhecidos, como a busca em largura, em profundidade, Dijkstra ou A*, precisamos conhecer o destino a priori e ter mapeado todo o terreno do jogo. Essa característica torna a busca em tempo real impossível, ou seja, não conseguimos traçar um caminho desconhecendo o objetivo. O presente trabalho foi desenvolvido com o intuito de criar um algoritmo de busca onde não é necessário saber a localização do destino nem possuir conhecimento do mapa, tornando o comportamento da Inteligência Artificial (IA) desenvolvida, intitulada Path Discovery, mais próximo possível ao de um humano.

Palavras-chave: Algoritmo de busca; Pathfinding; Jogos; Mapeamento.

ABSTRACT

One of the greatest challenges in gaming is related to the movement of secondary characters. In this way, Pathfinding algorithms are used to determine the best path between a starting and an ending point. However, to obtain these results from known algorithms such as Breadth-first search, Depth-first search, Dijkstra or A* search, we need to know a priori the destination point and have mapped the whole terrain of the game. This feature makes real-time searching impossible, meaning we cannot trace a path unaware of the goal. The present work was developed in a way to create a search algorithm where it is not necessary to know the destination's location nor having knowledge of the map, making the behavior of the developed Artificial Intelligence (AI), called Path Discovery, as close as possible to a human.

Key-words: Search Algorithms; Pathfinding; Games; Mapping.

SUMÁRIO DE FIGURAS

SUMÁRIO DE TABELAS

SUMÁRIO

CAPÍTULO 1 - INTRODUÇÃO

Com a constante evolução das tecnologias de informação, cada vez mais as pessoas estão imersas nesse mundo de interações com as máquinas. Mais especificamente na área dos jogos, essa interação se dá por um contexto onde o jogador, interage, muitas vezes, com personagens controlados por uma Inteligência Artificial (IA). Um dos crescentes desafios na área é a movimentação destes agentes secundários.

O que se espera, então, é que essa relação homem-máquina (jogador-IA) seja imperceptível e suave, ou seja, que o jogador não consiga notar grandes diferenças entre um personagem guiado por uma terceira pessoa ou por uma IA. Neste sentido, é necessário adaptar implementações existentes a fim de otimizar e melhorar o sistema das IAs para que seu comportamento seja humanizado, reagindo de forma espontânea frente a diversos acontecimentos e situações.

Pathfinding, ou busca de caminhos, é uma estratégia comumente implementada como a essência de um sistema de movimentações em uma IA. Essa estratégia tem a responsabilidade de encontrar ao menos um caminho, desde que exista, entre dois pontos quaisquer (de partida e chegada) inseridos no mundo do jogo. Por trás dos panos, os algoritmos, empregados para este fim, tratam o mapa do jogo como uma estrutura de grafo, onde os pontos são nós e as conexões são arestas, podendo ter peso ou não. A partir dele, diferentes lógicas são utilizadas para descobrir e traçar, por vezes o melhor, caminho entre os pontos pré-determinados. Uma grande dificuldade se encontra no processamento desses dados para que o melhor caminho seja alcançado.

Diante deste contexto, o objetivo do presente trabalho é a elaboração de um algoritmo de Pathfinding, denominado **Path Discovery**[1], que seja capaz de tornar o comportamento da IA o mais humanizado possível, não abrindo mão de uma boa capacidade de processamento em termos de otimização.

[1] Disponível em <https://gitlab.com/gfribeiro/TCC-Labirinto.git>.

1.1. Contribuição

Uma vez apresentadas as motivações para o desenvolvimento do Path Discovery, se faz necessário comentar sobre suas contribuições, dado que refletem justamente nas limitações dos algoritmos de busca já existentes. A característica básica do algoritmo implementado é a não necessidade de conhecer a localização do objetivo final e nem ter o terreno mapeado previamente. Este princípio permite não apenas a dinamicidade nos jogos como também reações mais humanizadas para o personagem controlado pela IA.

Desta forma, as principais contribuições para a academia são mostrar que existem outras alternativas para algoritmos de Pathfinding e estimular os estudos nessa área, dado que ainda há muito o que ser explorado.

1.2. Organização do Texto

Este trabalho está dividido da seguinte forma: o Capítulo 2 traz o detalhamento de cada um dos principais algoritmos de Pathfinding existentes na literatura e também levanta seus principais déficits, o que gerou a motivação para a IA desenvolvida e descrita no Capítulo 3.

No 4º capítulo serão apresentados e analisados os resultados referentes à comparação dos algoritmos. E, no último capítulo, será dada uma conclusão sobre o estudo e desenvolvimento do algoritmo proposto, assim como pontos de melhoria para trabalhos futuros.

CAPÍTULO 2 - ALGORITMOS DE PATHFINDING EM JOGOS

Inserido no mundo dos jogos, a Inteligência Artificial (IA) é responsável por dar vida aos personagens que não são controlados pelo jogador. Uma de suas muitas responsabilidades é a movimentação do agente dentro do cenário. Isto significa escolher um destino e a rota que se deve seguir para alcançá-lo. Tal escolha é baseada na priorização de algum critério como custo, distância, regra do algoritmo ou interesse que se deseja atingir. Para isso, são utilizados algoritmos chamados de Pathfinding.

Portanto, Pathfinding é o processo de busca por um caminho dentro do mundo virtual, desde que a solução satisfaça aos critérios predeterminados. Assim, os algoritmos nessa abordagem funcionam baseados em duas etapas principais (CUI e SHI, 2012)[2]:

1. **Mapeamento do terreno**: um algoritmo que escaneia todo terreno é executado e então são mapeados todos os nós e suas respectivas conexões, gerando um grafo
2. **Busca de um caminho**: é utilizado um algoritmo de busca de caminhos em grafos (geralmente o A*) e é retornado um caminho de um ponto A (origem) a um ponto B (destino)

Vale ressaltar que, para o algoritmo funcionar corretamente, é necessário que o grafo com informações como as conexões e os custos entre os nós seja conhecido a priori.

Sendo assim, neste capítulo detalharemos os algoritmos de busca mais difundidos e que foram utilizados como base para o presente trabalho.

O capítulo está organizado da seguinte forma: na seção 2.1 serão apresentadas algumas abordagens para algoritmos de pré-processamento; em seguida, na seção 2.2 serão detalhados alguns dos algoritmos de busca existentes e, na seção 2.3, indicaremos algumas de suas limitações, que motivaram o desenvolvimento do algoritmo Path Discovery.

---

[2] CUI, Xiao; SHI, Hao. An Overview of Pathfinding in Navigation Mesh. **International Journal of Computer Science and Network Security**, Australia, v. 12, n. 12, dez. 2012.

2.1. Mapeamento

O primeiro passo para realizar uma busca por um caminho é o de preparar o mapa do jogo, ou seja, escanear e mapear a topologia do terreno a fim de transformá-lo numa estrutura reconhecível a partir de um algoritmo de busca.

Há diversas maneiras de atingirmos esse objetivo e em Algfoor, Sunar & Kolivand (2015)[3] destacam-se os usos a partir de grids regulares ou irregulares. Grids são compostos por vértices conectados a arestas e que, juntos, representam um grafo. Dependendo do tipo de estrutura utilizada, a performance da navegação durante a busca pode ser afetada.

O tamanho do grafo também influencia diretamente no desempenho do algoritmo. Quanto maior a granularidade e a quantidade de conexões dos nós, melhor será o caminho entre dois pontos, porém, serão necessários mais passos para alcançar o resultado.

2.1.1. Grids Regulares

Bastante conhecidos e largamente utilizados por desenvolvedores de jogos, os grids regulares descrevem a pavimentação a partir de polígonos regulares. De acordo com Algfoor, Sunar & Kolivand (2015), os grids podem ser classificados em:

- **Grid Quadrangular**: um grid quadrangular é gerado a partir de quadrados regulares. Altamente popular na indústria de vídeo games, é utilizado em jogos onde o mapa pode ser dividido em quadrados. Por exemplo: um labirinto, uma casa;
- **Grid Triangular**: composição a partir de triângulos regulares. Sendo o menos popular da lista, é mais utilizado em projetos que possuem otimizações para esse tipo de grid;
- **Grid Hexagonal**: composição feita por hexágonos regulares. Muito utilizado em jogos táticos e de estratégia onde o tabuleiro ou mapa são compostos por nós com muitas conexões;

[3] ALGFOOR, Zeyad Abd; SUNAR, Mohd Shahrizal; KOLIVAND, Honshang. A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. **International Journal of Computer Games Technology**, v. 2015.

A Figura 1 exemplifica os diferentes tipos de grid regular citados acima

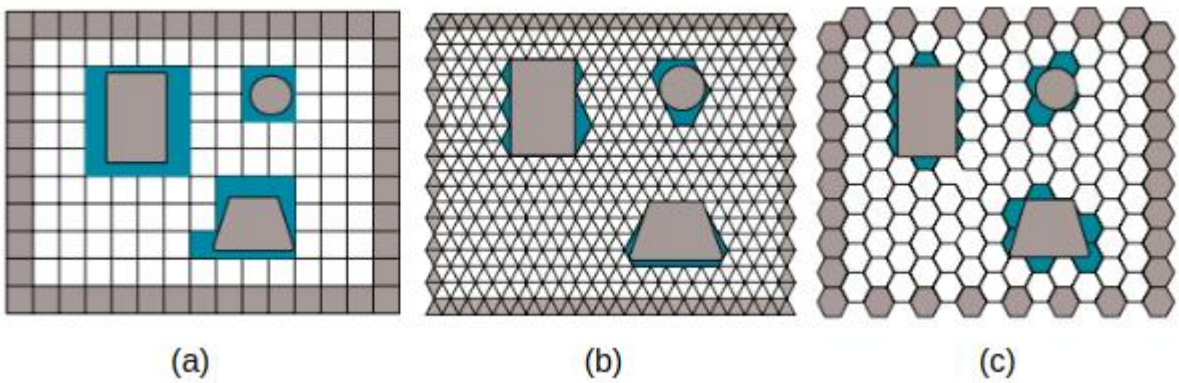


Figura 1. (a) representação de um grid quadrangular com três obstáculos (b) representação de um grid triangular com três obstáculos e (c) representação de um grid hexagonal com três obstáculos

2.1.2. Grids Irregulares

Compostos por polígonos irregulares, cujas formas se dão de acordo com as características do terreno, os grids irregulares são incorporados em diversas aplicações, de acordo com Algfoor, Sunar & Kolivand (2015):

- **Navigation Mesh:** NavMesh, como também é conhecido, é um método para representar todo o ambiente do jogo através de polígonos convexos, cujas arestas não se intersectam. A convexidade garante que o personagem do jogo, controlado pela IA, seja capaz de se movimentar livremente de/para qualquer ponto dentro do polígono. O conjunto de polígonos descreve uma superfície caminhável em um ambiente 3D, onde cada um deles pode ser interpretado como um nó para um algoritmo de pathfinding.

    a. **Triangulação** é um caso especial do **NavMesh**, onde todos os polígonos são triângulos, cujos ângulos agudos (menores de 90°) devem ser maximizados. Essa propriedade garante que um caminho ótimo não atravessa um mesmo triângulo mais de uma vez. Em outros casos, o número de lados do polígono pode variar de 4 a 6 e, se maior do que isso, pode haver um aumento significativo do uso de memória. A Figura 2 elucida este processo.

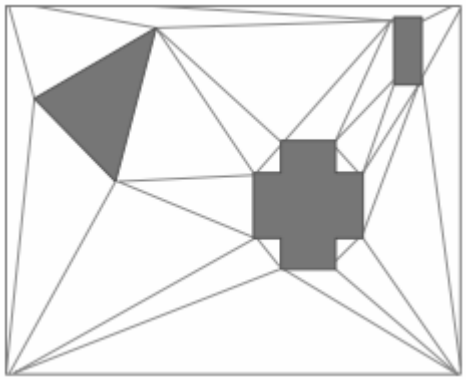


Figura 2. Representação do navmesh por triangulação com obstáculos destacados

2.2. Algoritmos de Busca

Os algoritmos de busca têm o propósito de encontrar um caminho de um nó A (origem) a um nó B (destino) em um grafo. Alguns dos algoritmos mais utilizados, e que serão detalhados mais à frente, de acordo com Korf (1996) [4], são:

- Busca em Largura (KORF,1996)
- Busca em Profundidade (KORF,1996)
- Dijkstra (KORF,1996)
- A* e (KORF,1996 e BOTEA; *et al*., 1998)

É importante salientar que não há um algoritmo melhor que outro, e sim que cada um cumpre um papel atacando um determinado objetivo. Em outras palavras, significa que a escolha pelo melhor algoritmo depende exclusivamente de quais atributos devemos priorizar em detrimento de outros relacionados ao resultado que se deseja atingir. Atributos como:

- **Performance**: complexidade do algoritmo
- **Completude**: se o algoritmo sempre retorna uma solução
- **Solução ótima**: se o algoritmo sempre retorna a melhor solução
- **Soluções para múltiplos destinos**: se o algoritmo aceita, e resolve, múltiplos destinos

[4] KORF, Richard E. **Artificial Intelligence Search Algorithms**. Los Angeles: Universidade da Califórnia, jul. 1996.

- **Conhecimento prévio do terreno**: necessidade de ter todo o grafo mapeado
- **Tipo de busca**: se é direcionada ou não direcionada (exaustiva)

De acordo com Botea et al. (1998)[5], há diversas abordagens para construção de IAs em busca de caminhos, que são divididas em duas principais categorias: **direcionada** e **não direcionada**.

Analogamente à um rato em um labirinto, na abordagem não direcionada o rato despende toda sua energia andando “às cegas” simplesmente tentando achar uma fuga. Eventualmente o rato acabará alcançando becos sem saída. E é neste contexto de busca exaustiva que os algoritmos de busca em largura e profundidade estão inseridos. Ambos serão abordados com maior profundidade posteriormente.

Em contraposição, todos os algoritmos de pathfinding inseridos em uma abordagem direcionada definitivamente não caminhariam às cegas pelo labirinto. Pelo contrário, em todos os casos, a IA consegue, de alguma maneira, recuperar informações de seu progresso até o momento para definir o caminho a ser tomado a partir dali, ou seja, o próximo nó a ser visitado. Alguns algoritmos podem levar em consideração o custo desta transição, que normalmente é medido pela distância entre os nós, fazendo ser possível encontrar a solução mais eficiente possível, isto é, o menor caminho/caminho de menor custo. As estratégias utilizadas nos algoritmos para atingir esse objetivo são:

- **Busca de custo uniforme**: ajusta o algoritmo para sempre escolher o menor custo até o próximo nó
- **Busca por heurística**: estima o custo do próximo nó até o objetivo

Os algoritmos comumentes empregados em jogos e que estão inseridos na abordagem direcionada são Dijkstra e A*. Ambos utilizam a(s) estratégia(s) descrita(s) acima como forma de implementação. Existem diversas variações na literatura baseadas no A* e aplicadas no mundo dos jogos, como citado em Cui e Shi (2011)[6] e Silver (2006)[7].

[5] BOTEA, Adi; et al. Pathfinding in Computer Games. **IBM Research**, Dublin, 1998.
[6] CUI, Xiao; SHI, Hao . A*-based Pathfinding in Modern Computer Games. **International Journal of Computer Science and Network Security**, Australia, v. 11, n. 1, jan. 2011.
[7] SILVER, David. Cooperative Pathfinding. Edmonton: Universidade de Alberta, 2005.

A seguir, serão providos detalhes e descrição de características de cada um dos principais algoritmos da literatura. A fim de ilustrar o processo de busca de cada um deles, consideremos o exemplo da Figura 3, que descreve a seguinte situação:

Um grafo acíclico G, um conjunto de vértices V e um conjunto de arestas E, onde G = (V,E), V = {o, a, b, c, d} e E = {(o,a), (o,b), (o,c), (a,d), (b,d), (c,d)} apresentado em forma de mapa e em sua estrutura de árvore, onde o objetivo é atingir o nó *d* (destino) a partir do nó inicial *o* (origem).

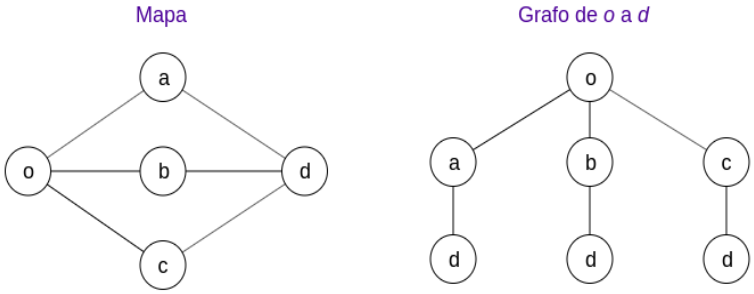


Figura 3. Um exemplo de grafo representado em forma de mapa e também em sua estrutura de árvore

2.2.1. Buscas em Largura e em Profundidade

Enxergando o mundo virtual como um grande grafo com nós conectados em estrutura de árvore, a cada iteração do algoritmo da busca em largura, o grafo se expande a partir do nó atual para todos os seus vizinhos (CUI e SHI, 2011). Em outras palavras, dado um nó pai, serão mapeados os caminhos/arestas até seus filhos e estes serão visitados sucessivamente até que a busca atinja o nó objetivo, sendo este o critério de parada.

Na **busca em largura**, os custos não são levados em consideração para a escolha do próximo nó a ser visitado. Além disso, se existe uma solução do ponto de partida até o objetivo, a busca garante encontrá-la; e caso exista mais de uma, o algoritmo retorna à primeira encontrada, ou seja, a solução mais “rasa”.

A **busca em profundidade** também não leva em consideração o custo de um nó a outro, mas se diferencia na maneira em que eles são percorridos. Em uma estrutura de árvore, os nós filhos têm preferência se comparados com irmãos. Ou seja, dado um nó inicial, serão mapeadas todas as ligações existentes com seus respectivos nós filhos e estes serão visitados antes de qualquer outro, até que se atinja um nó sem conexão, ou folha. Apenas então a busca retorna à camada anterior

na estrutura, consumindo os nós irmãos, e assim sucessivamente, enquanto não chegar no objetivo. Vale destacar que esta busca é uma boa abordagem para problemas com múltiplas soluções, já que possui uma grande chance em achar a solução ao explorar pequena porção do espaço completo de busca.

A Figura 4 compara e ilustra o passo a passo do processo da busca em profundidade e em largura. Pode-se observar que ambos encontram uma solução: a busca em largura levou quatro iterações para isso, enquanto a busca em profundidade levou apenas duas, reforçando que ela é mais eficiente em casos com múltiplas soluções.

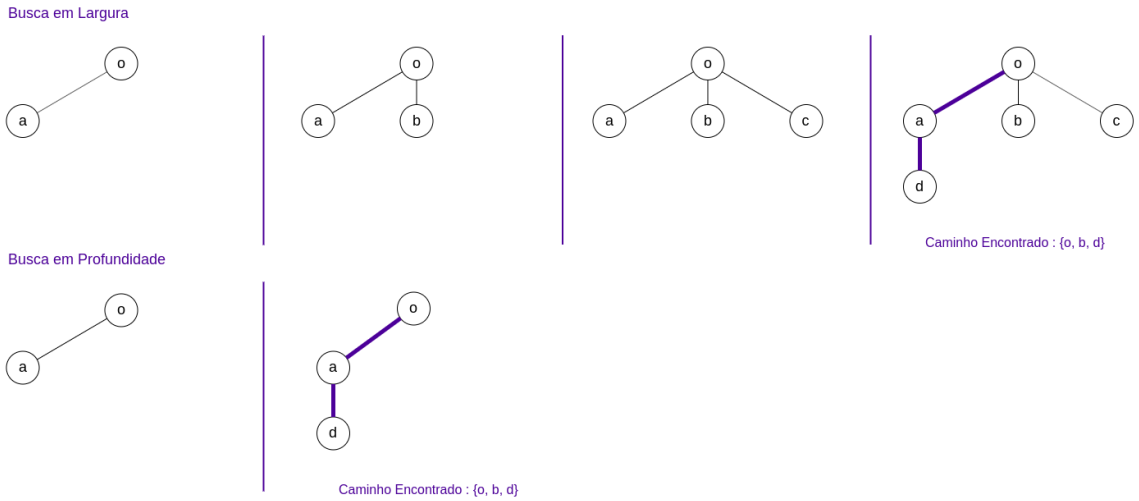


Figura 4. Ilustração do processo das buscas em largura e em profundidade, respectivamente, baseado no exemplo da Figura 3

2.2.2. Dijkstra e A*

Dentro do mundo dos jogos, **Dijkstra** e principalmente o **A*** (ou suas variações) são os algoritmos de referência quando se refere à busca de caminhos, de acordo com Botea et al. (1998)[8]. Ambos empregam uma abordagem direcionada, mas utilizam diferentes estratégias em seu processo para atingir o objetivo.

Dijkstra é direcionado ao custo e emprega a estratégia de busca de custo uniforme para solucionar o problema do caminho mais curto em um grafo com arestas de peso não negativo. O algoritmo **Dijkstra** funciona, basicamente, da seguinte forma:

[8] BOTEA, Adi; et al. Pathfinding in Computer Games. **IBM Research**, Dublin, 1998.

1. Atribuir d[s] = 0, que é a distância do nó inicial *s* a ele mesmo;
2. Para todo $v \in V[G]$, faça:
    a. d[v] = ∞;
3. A partir de um conjunto Q = V[G] e enquanto não estiver vazio, faça:
    a. Extrai um vértice *u* com menor d[u], que é a distância de *s* até *u*
    b. Para cada vértice *v* adjacente a *u*, verifica se d[v] > d[u] + d(u, v), onde d(u, v) é a distância de *u* até *v*
        i. Caso positivo, atribuir d[v] = d[u] + d(u, v)
        ii. E identificar a origem da conexão fazendo π[v] = u, de maneira a formar um caminho mínimo

No final do algoritmo, teremos o menor caminho entre *s* e qualquer outro vértice de G.

O **A*** (pronunciado *a-estrela*) é um dos algoritmos mais eficientes conhecidos em termos de busca por caminhos de acordo com Cui & Shi (2011) e é justamente por isso que normalmente ele ou alguma de suas variações é utilizado para estruturar o processo de busca de caminho ou pathfinding em jogos. Inserido na abordagem direcionada, ele combina as estratégias de busca de caminho uniforme com o uso de heurística, tendo o intuito de minimizar o custo total e chegar na solução ótima em um tempo relativamente rápido. A estrutura do **A*** básico se apoia em três principais atributos *g*, *h* e *f*, sendo:

- ***g***: custo do caminho percorrido do nó inicial até o nó corrente, ou seja, a soma de todos os custos entre os nós até o atual;
- ***h***: é a heurística, que representa o custo do nó corrente até o nó objetivo. No A* puro, essa é a distância em "L" entre os nós. Então, sendo u = (ux, uy) e v = (vx, vy) dois vértices, a distância é medida por
    - h = |(vy - uy)| + |(vx - ux)|;
- ***f***: soma total dos custos *g* e *h*, que é a melhor estimativa do custo total passando pelo nó corrente e é atribuído como a pontuação/fitness de um nó.

O objetivo dos atributos é ter uma forma de seguir pelo caminho mais promissor a partir de uma boa escolha entre os nós a cada iteração. Nesse caso, quanto menor

a pontuação relativa a um nó, maior a chance dele fazer parte do caminho que leva à solução. Intuitivamente, a cada passo do algoritmo são mapeados os nós conectados ao vértice atual e também calculados os valores de *g*, *h* e *f* para cada um deles. Desta forma, a escolha do próximo nó a ser visitado será pelo que tiver menor pontuação em *f*. Assim, o procedimento é repetido enquanto não se alcança o nó objetivo. Sendo assim, o algoritmo **A*** é executado da seguinte forma:

1. Adiciona o nó inicial *i* à uma lista “aberta” A, que contém os nós a serem explorados;
2. Calcular g, h e f para todos os nós adjacentes e adicioná-los em A;
3. Mover *i* para lista “fechada” F, que contém os nós completamente explorados;
4. Enquanto A ≠ {}, faça:
    a. Procurar pelo nó *n* com menor valor *f* em A. Este será o nó corrente;
    b. Para cada nó alcançável *a* a partir de *n*:
        i. Se *a* ∈ F, ignore-o;
        ii. Se *a* ∉ A, adicione-o e faça *n* pai de *a*. Calcular *g(a)*, *h(a)* e *f(a)*;
        iii. Se *a* ∈ A, checar se existe melhor caminho (nó com menor valor de *f*). Se sim, atualizar o pai dele para *a* e recalcular *g* e *f*;
    c. Parar quando:
        i. Adicionar o nó destino a F ou
        ii. Não encontrar o nó destino e A vazio.
5. Traçar o caminho contrário partindo do nó destino até *i*. Esta é a solução.

Baseado no exemplo da Figura 3, a Figura 5 contém pesos não negativos atribuídos às arestas a fim de comparar o **Dijkstra** com o **A***, ilustrado na Figura 6.

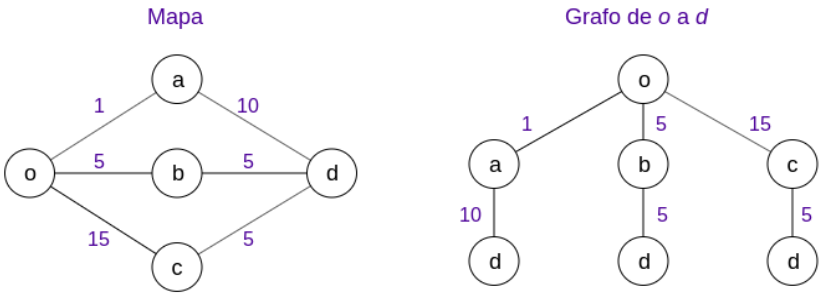


Figura 5. Exemplo de grafo com peso nas arestas representado em forma de mapa e estrutura de árvore

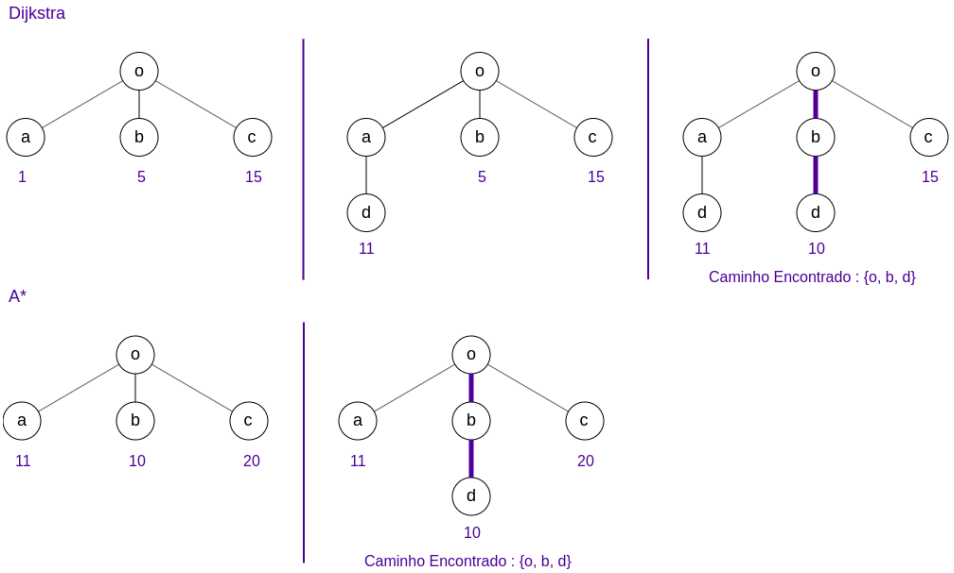


Figura 6. Comparação entre a execução dos algoritmos Dijkstra e A*, respectivamente, baseada no exemplo da Figura 5

A Figura 6 ilustra a execução dos algoritmos Dijkstra e A*. Conforme ilustrado pela figura, enquanto o Dijkstra leva três iterações para chegar na solução, o A* levou apenas duas. Não deixando de levar em consideração que em cada passo foi preciso calcular os custos e/ou heurísticas, ainda assim estes se mostram melhores opções se comparados com as buscas em largura e em profundidade, dado que retornam a solução ótima. No entanto, todos os algoritmos de busca apresentados possuem limitações. Em todos os casos, é necessário o conhecimento prévio do terreno, dificultando a dinamicidade do jogo.

Nas buscas em largura e em profundidade, o caminho pode não ser eficiente, dado que a busca não é guiada em função da posição do nó destino (*d*), diferente dos algoritmos Dijkstra e A*, onde as escolhas são, de certa forma, inteligentes. Apesar de ser bastante utilizado na indústria de jogos pela sua eficiência, o A* requer e utiliza uma grande quantidade de recursos de CPU, como citam Graham *et al.* (2003)[9], e em casos de mapas muito grandes, a busca por caminhos diferentes para personagens distintos pode fazer com que o jogo “congele” por alguns segundos devido ao alto processamento, desestimulando os jogadores. E, por esses problemas, os designers de jogos ajustam os algoritmos a partir de adaptações e modificações que julgam necessárias a fim de evitá-los.

[9] GRAHAM, Ross; McCABE, Hugh;SHERIDAN, Stephen. Pathfinding in Games. **The ITB Journal**, v. 4, n. 2, 2003.

2.3. Considerações Finais

Abaixo, a Tabela 1 resume e compara os algoritmos da literatura de acordo com algumas características.

Tabela 1. Comparação entre os principais algoritmos de Pathfinding

	Busca em Largura	**Busca em Profundidade**	**Dijkstra**	**A***
Performance	$O(b^d)$	$O(b^d)$	$O(b^{c/m})$	$O(b^d)$
Completude	sim	sim	sim	sim
Solução ótima	sim*	não	sim	sim
Solução para múltiplos destinos	sim	sim	sim	sim
Conhecimento prévio do terreno	sim	sim	sim	sim
Tipo de busca	não direcionada	não direcionada	direcionada	direcionada

Fonte: Autoria própria, 2019.

Em termos de complexidade, a busca em largura leva tempo proporcional à quantidade de nós do grafo, dado que cada um é visitado em tempo constante e é uma função em termos do fator de ramificação *b* e da profundidade da solução *d*. Dado que o número de nós no nível *d* é b^d, no pior caso, a busca levará $b + b^2 + b^3 + \ldots + b^d$ de tempo para visitá-los, que é $O(b^d)$. Uma vez que a busca em profundidade gera a mesma quantidade de nós que a busca em largura apenas em ordem diferente, a complexidade de tempo será a mesma $O(b^d)$. O Dijkstra possui complexidade determinada pelo tipo de estrutura utilizada em sua implementação. No pior caso, o algoritmo é $O(b^{c/m})$, onde *c* é o custo da solução ótima e *m* é o custo mínimo de uma aresta.

Já a complexidade de tempo do A* depende diretamente da precisão da função heurística. Por exemplo, se a função de avaliação heurística *h(n)* for estimado em 0

(zero), então o A* roda como o Dijkstra, com complexidade exponencial. Em geral, o tempo gasto pelo A* é uma função exponencial do erro da função heurística. Se o erro relativo for constante, que significa que ele é uma porcentagem fixa da quantidade sendo estimada, então o A* roda em tempo exponencial. No entanto, a base do expoente é menor do que as buscas guiadas por força bruta, reduzindo sua complexidade.

Em relação a encontrar uma solução ótima, a busca em largura a encontra apenas nos casos onde o custo das arestas é uniforme. Neles, sua execução será similar à do Dijkstra.

Porém, como descrito anteriormente, para funcionar corretamente, os algoritmos de Pathfinding necessitam do terreno previamente mapeado. Isso implica em certas limitações:

1. Nem sempre é possível ou desejamos que o personagem conheça todo o mapa onde está sitiado
2. Terrenos que possuem muitas mudanças em tempo de execução, precisam ser remapeados com frequência
3. Necessita-se saber a localização exata do destino
4. Dependendo do terreno e do algoritmo utilizado, o processo de mapeamento pode ser bem demorado

A fim de ilustrar os problemas citados, algumas situações de jogos fictícios foram imaginadas:

1. Um guarda que trabalha numa fábrica está fazendo sua ronda padrão. Ele tem total conhecimento da sua área de vigia, mas desconhece algumas áreas contempladas pela fábrica. O mesmo guarda avista um intruso e começa a perseguí-lo. A perseguição o leva a uma área desconhecida, e lá perde o intruso de vista. A partir desse momento, o vigia deverá vasculhar a área e, caso ouça algum barulho, tentará chegar até sua fonte, ou o mais próximo possível.

Nos algoritmos padrões, o agente traça uma rota direta entre sua posição e a fonte do barulho. Mas este não deveria ser o comportamento esperado numa situação

real, já que não há conhecimento do local de origem do barulho, acabando por desumanizar tal ação.

2. Um labirinto que, de tempo em tempo, rotaciona seus corredores.

O problema acarreta no retrabalho em mapear todo o terreno pelas inúmeras vezes que os corredores mudassem suas posições, gerando um processamento extra. Como solução, pode-se detectar as áreas que mudam com frequência para aplicar o algoritmo de mapeamento apenas nelas. Outra alternativa é guardar todos os estados do terreno em memória e utilizar o estado correspondente dado um tempo específico. Porém, nessas situações, uma pessoa iria atualizar o mapa do labirinto sob demanda, ou seja, conforme for explorando o labirinto e ir atualizando as modificações, invalidando o mapeamento pré-modificação.

3. Um jogo onde o jogador deve disputar com a IA para encontrar itens escondidos pelo mapa.

O jogo só possui sentido se a IA também não souber a localização dos itens, tendo que procurá-los pelo cenário, como o jogador faria. Atualmente existem técnicas para contornar a situação, como por exemplo combinar uma IA macro, que observa a localização de todos os itens escondidos e uma IA micro, que controla o personagem que disputará com o jogador real. Ou seja, em situações como essa, o importante é que o personagem não trapaceie: apesar de conhecer a posição dos objetos, a IA deverá descobri-los por si só, simulando o processo de exploração do cenário.

4. Um jogo que possui o terreno muito irregular, com diversas deformidades.

A complexidade do terreno influencia diretamente na complexidade do algoritmo de mapeamento. Aplicar o algoritmo em mapas densos, com muitos obstáculos e irregularidades pode resultar em lentidão durante o desenvolvimento do jogo (GRAHAM *et al.,* 2003).

Com tais limitações em vista, pensamos no que pode ser feito para contorná-las. Como estão ligadas principalmente ao mapeamento, chegamos ao seguinte questionamento: “*E se não precisarmos mapear previamente o terreno?*”. A resposta

se deu no desenvolvimento do algoritmo **Path Discovery**, que será apresentado no próximo capítulo.

CAPÍTULO 3 - O PATH DISCOVERY

Ao analisarmos as soluções existentes para algoritmos de Pathfinding, notamos que suas limitações estão associadas a necessidade de se ter o terreno todo mapeado para iniciar a busca por um caminho. Para contornar este problema, pensamos em um algoritmo que, ao invés de mapear todo o terreno previamente, mapeie dinamicamente os arredores do agente/personagem, conforme ele se locomove pelo cenário.

A proposta do algoritmo é implementar uma IA capaz de se adaptar a ambientes desconhecidos e tomar decisões inteligentes quanto a qual caminho seguir mesmo quando não se possui a localização exata do objetivo. Para isso, foi desenvolvido um algoritmo de Pathfinding baseado no A*.

Características importantes da IA desenvolvida:

- **Mapeamento em tempo real:** o terreno é mapeado conforme o agente se locomove;
- **Escolha inteligente de caminho:** o agente se preocupa em não escolher caminhos repetidos nem aqueles que levarão a becos sem saída. Utiliza nós e conexões já mapeados para tentar descobrir qual é a melhor escolha;

O capítulo está organizado de maneira que na seção 3.1 a lógica do Path Discovery será apresentada em função das etapas de mapeamento, escolha do destino, manipulação de caminho e movimentação; em seguida, na seção 3.2 será abordada a implementação do código da IA assim como o ambiente selecionado para seu desenvolvimento. Por fim, na seção 3.3 apontaremos algumas considerações.

3.1. O Algoritmo

O algoritmo pode ser divididos em 4 tópicos principais, que serão melhor detalhados à frente:

- **Mapeamento:** detecção dos arredores do agente, mapeando nós e conexões, adicionando-os ao grafo;
- **Escolha de destino:** como se dará a escolha do próximo nó a ser visitado;

- **Manipulação de caminho:** definição do caminho até o próximo destino e iterações em caminhos já definidos;
- **Movimentação:** deslocamento até o próximo ponto de destino.

A abordagem proposta é composta de 5 passos básicos que permitem um agente navegar por uma região ainda não explorada, como indicado abaixo e ilustrado na Figura 7.

1. Mapear os arredores
2. Determinar próximo nó a ser visitado
3. Rotacionar o agente, caso necessário
4. Mover para a frente
5. Verificar se alcançou o objetivo

Cada um desses passos está englobado em um ou mais dos 4 conceitos apresentados anteriormente. No primeiro passo, a nossa abordagem mapeia os arredores de um personagem através de sensores. Este tópico faz parte da etapa de mapeamento, que será descrita em mais detalhes na seção 3.1.1. No segundo passo, é feita a escolha do próximo nó a ser visitado. Esta decisão é responsabilidade das etapas de escolha do destino e manipulação do caminho, apresentadas na seção 3.1.2 e 3.1.3, respectivamente. O terceiro, quarto e quinto passos estão inclusos na etapa de movimentação, detalhada na seção 3.1.4.

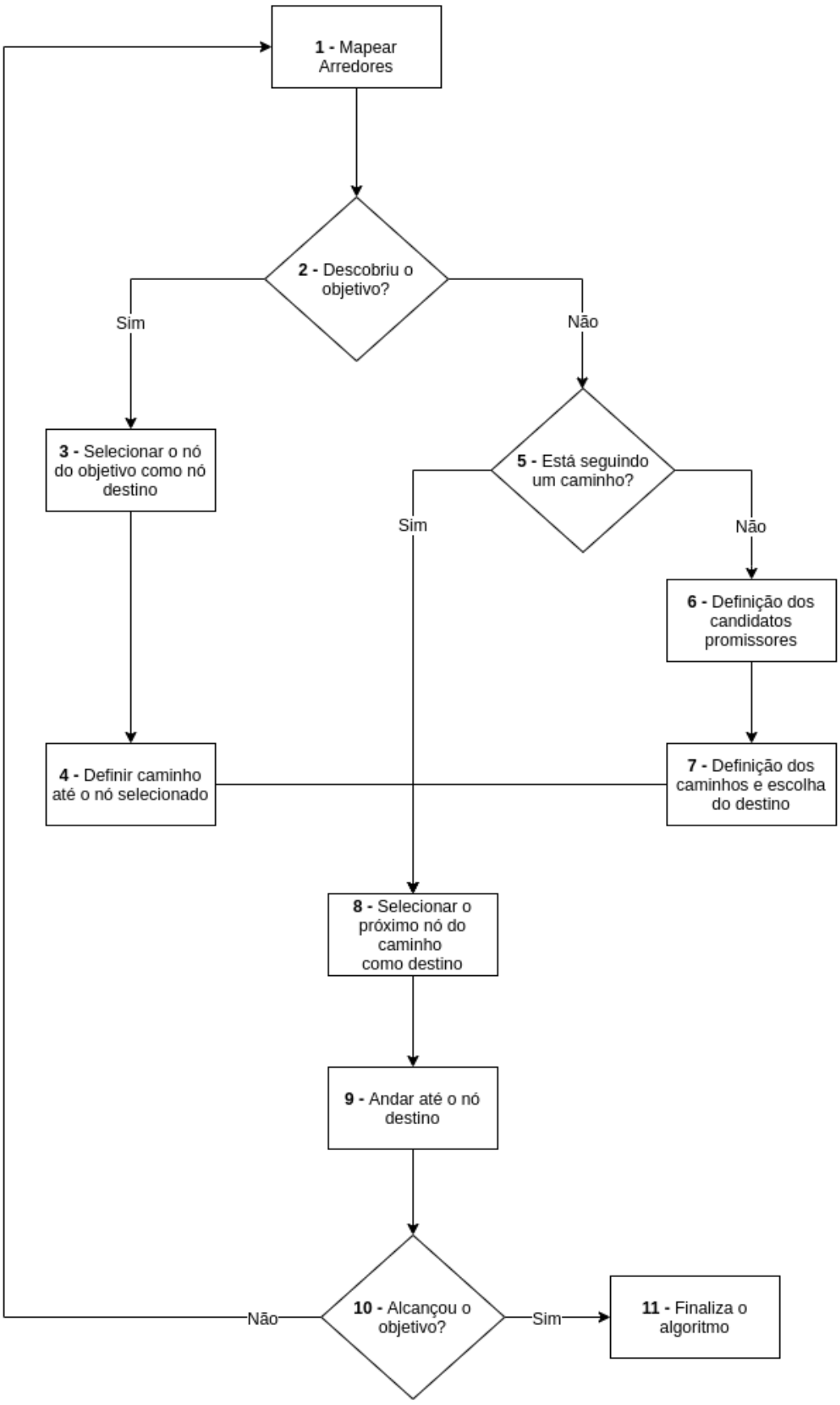


Figura 7. Fluxo resumido e geral do comportamento da IA

3.1.1. Mapeamento

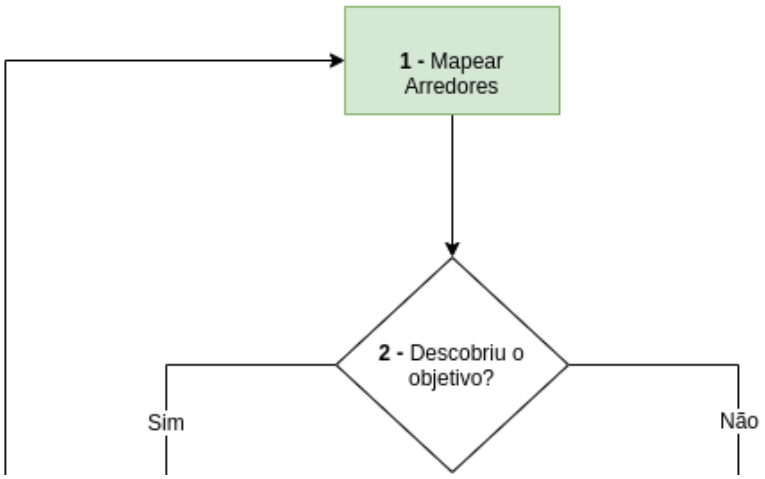


Figura 8. Etapa do mapeamento do terreno destacada no fluxo

A etapa de mapeamento é o primeiro passo do algoritmo, como ilustra a Figura 8, que é responsável por escanear os arredores do agente e inserir os dados coletados em um grafo. Isto é possível devido a sensores posicionados em volta dele que, quando ativados, enviam a informação de caminho livre ou bloqueado para a estrutura do grafo.

Como detalhado no capítulo 2, seção 2.1, os terrenos podem ser abstraídos em diferentes tipos de grids. No desenvolvimento, optamos por estruturar o terreno em um grid quadrangular com elementos uniformes, onde cada nó representa um quadrado 2x2cm do mapa, como mostra a Figura 9. Apesar de termos escolhido esse tipo grid, nossa abordagem também pode ser aplicada a qualquer outro, desde que seja regular.

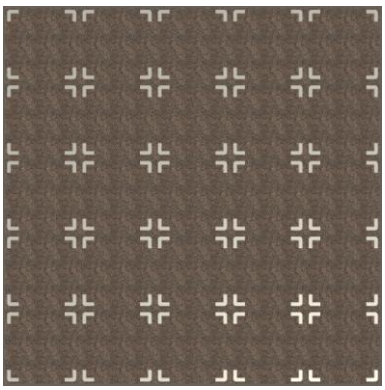


Figura 9. Representação do terreno na Unity em grid quadrangular

Cada nó possui 4 conexões: com o nó de cima, de baixo, da esquerda e da direita. A estrutura funciona como uma lista duplamente encadeada, significando que um nó conhece suas conexões e também os nós ligados a ele e vice-versa.

Com o intuito de simular um comportamento mais humanizado, determinamos que o agente possui uma profundidade de campo, ou seja, a distância que consegue “enxergar”, de três nós. Ao traçar um paralelo com o mundo real, idealizamos cada elemento do grid medindo 2x2 metros, sendo assim, a profundidade de campo do personagem é de 6 metros, valor que julgamos razoável para a IA proposta. Também definimos que são detectados, em ordem, os nós a frente, a esquerda e à direita do agente, de acordo com sua orientação.

Quando um nó é mapeado, a conexão que leva até ele também é adicionada. Uma observação relevante é que as únicas conexões mapeadas são as que estão em uma linha reta partindo da posição do agente até o novo nó. Ou seja, apenas as conexões com nós que estão na mesma linha (ou coluna) do nó de origem são detectadas, como mostra a Figura 10.

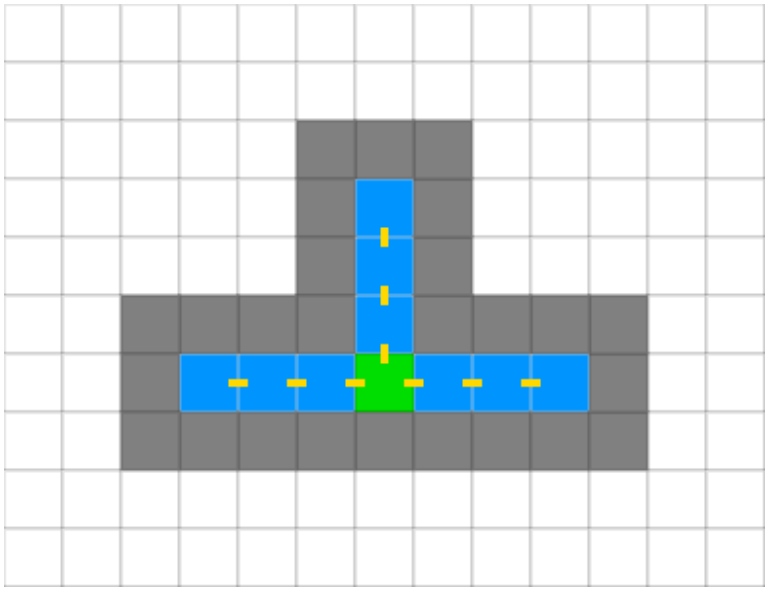


Figura 10. Representação do agente em verde, seu campo de visão, em azul e as conexões mapeadas em amarelo

Importante ressaltar que o agente mapeia inclusive os nós que não consegue alcançar, desde que estejam dentro do limite de sua profundidade de campo. Isso significa que, se há uma parede entre a IA e um nó, o nó será mapeado e esta conexão

será marcada como *bloqueada*. Dessa forma, é possível descobrir os nós que são becos sem saída e excluí-los da lista de nós promissores antes mesmo de visitá-los.

Quando o sensor detecta uma conexão com um nó que ainda não foi mapeado, este novo nó é adicionado na lista de nós não visitados. Quando é detectada uma conexão *bloqueada*, o nó é incluído na lista de nós não alcançáveis. Quando uma conexão com um nó já mapeado é identificada, o nó tem esta conexão atualizada. Em último caso, quando um nó não alcançável tem uma conexão válida mapeada, este nó é movido para a lista de nós não visitados.

Para identificação dos nós no grafo utilizamos os índices como um par ordenado de *linha* e *coluna*, onde o primeiro nó do grafo é o nó inicial do personagem e possui índice (0,0). É válido destacar que também admitimos valores negativos para os índices. Por exemplo, o nó imediatamente abaixo da posição inicial possui o valor (-1,0).

3.1.2. Escolha do Destino

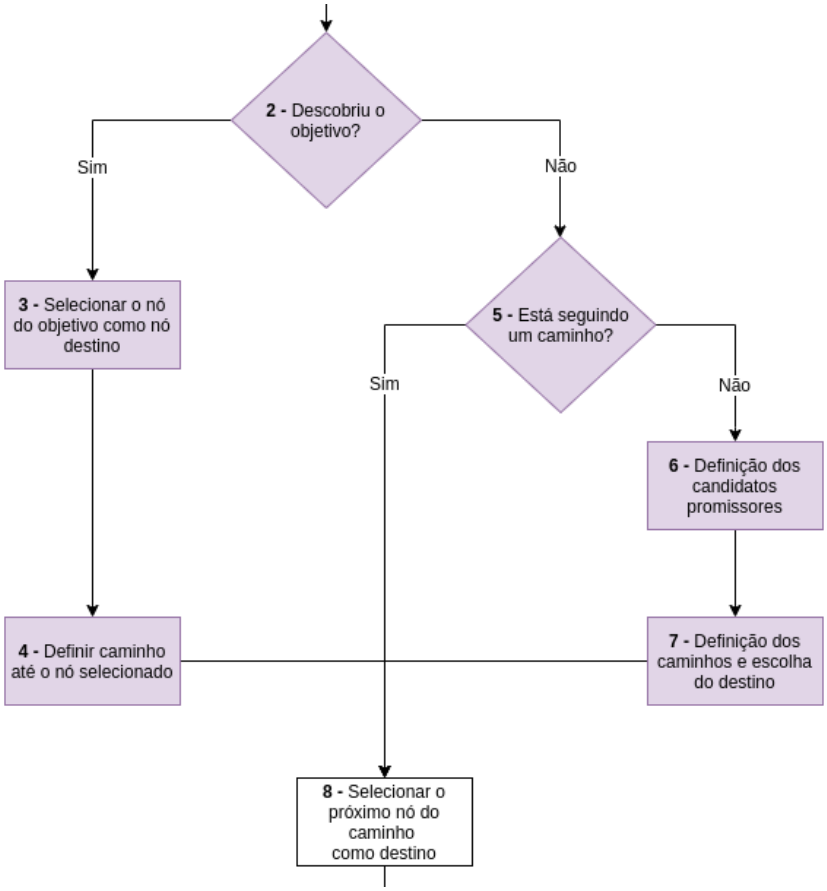


Figura 11. Etapa da escolha de destino destacada no fluxo

O próximo passo é a escolha do destino, que consiste na seleção de um novo nó para se alcançar e também em traçar uma rota até ele. Com o objetivo de tornar o comportamento do personagem mais humano e inteligente, definimos que seu principal critério para escolha do próximo destino é o custo necessário para alcançá-lo. Em caso de empate, a IA calcula qual destino possui maior possibilidade de levar a caminhos ainda não percorridos. A motivação desta escolha foi

A Figura 11 expõe que, para isso, é verificado se já foi descoberta a localização exata do objetivo final, visto na **etapa 2** do fluxo. Em caso afirmativo, ele é automaticamente escolhido como nó destino, de acordo com a **etapa 3**. Caso contrário, como mostra a **etapa 5**, é verificado se o agente está seguindo um caminho para um nó destino escolhido previamente. Se sim, o algoritmo segue o curso desta rota pela **etapa 8**, que veremos na seção 3.1.3. Do contrário, a **etapa 6** ilustra a definição de uma lista de nós candidatos, que limita o espaço de busca.

A lista de candidatos é inicialmente composta pelos elementos que foram mapeados e são alcançáveis, mas não visitados. Para cada um dos componentes da lista é verificado seu *valor promissor*. Este valor é calculado a partir de uma heurística, que indica quão interessante um determinado nó é para o agente. Dado um nó candidato, deseja-se saber o máximo de conexões que os nós conectados a ele podem possuir. Em outras palavras, este valor representa o máximo de possibilidades de descobertas de novos nós ao visitar este candidato. Esta heurística é composta pelo somatório das avaliações das possíveis conexões de cada nó. Realizada para cada uma de suas 4 conexões, a função de avaliação pode ser abstraída da seguinte forma:

1. Se a conexão ainda não foi mapeada
    a. Se o nó a quem ele possa estar conectado já foi visitado
        i. Retorna valor zero
    b. Se o nó a quem ele possa estar conectado já foi mapeado
        i. Valor a ser retornado inicia como -1 (é descontado o valor da conexão para alcançar este nó)
        ii. Para cada conexão não descoberta
            1. Adiciona 1 ao valor
        iii. Retorna valor
    c. Se o nó não foi mapeado
        i. Retorna valor 3
2. Senão
    a. Retorna valor zero

Caso o *valor promissor* retornado seja igual a zero, o elemento é retirado da lista de candidatos. Ao final desse processo, a lista estará definida apenas com os nós promissores.

Quando um nó é identificado como não promissor, ou seja, se ele foi retirado da lista de candidatos, ele também é retirado da lista de não visitados. Desta forma, evitamos que futuras buscas verifiquem um nó atestado anteriormente como improdutivo.

Atingindo a **etapa 7** do fluxo, a lista de candidatos está composta apenas pelos nós promissores. É preciso, então, escolher um de seus elementos como nó de destino. Definimos como critério de escolha a quantidade de passos necessários para alcançá-lo, partindo do nó corrente. Com esse objetivo, aplicamos o algoritmo **A*** para cada elemento da lista de candidatos, descobrindo um caminho do nó atual até ele. Após a obtenção de todos os resultados, ou seja, de todos os caminhos retornados pelo **A***, escolhemos o candidato que pode ser alcançado pela menor quantidade de passos. Se houver empate entre dois ou mais caminhos, é escolhido o nó que possuir maior *valor promissor*. No final das contas, o uso desta heurística fará com que o agente opte por um caminho com mais possibilidades. O segundo critério de desempate é pela ordem de inserção na lista durante a etapa de mapeamento dos nós (frente, esquerda e direita). Então, definiremos o trajeto atual do agente como o caminho para alcançar este candidato.

Continuando o fluxo a partir da **etapa 3**, uma vez que o destino foi definido, é utilizado o **A*** para traçar uma rota partindo da posição atual do agente até ele. A **etapa 4** ilustra esse processo de definição do caminho até o destino.

3.1.3. Manipulação de Caminho

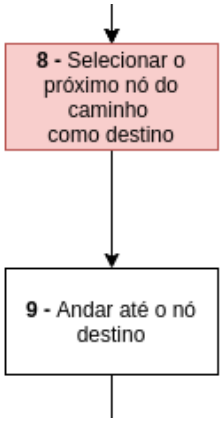


Figura 12. Etapa da manipulação de caminho destacada no fluxo

Como definido em 3.1.2, a escolha de um destino resulta na criação de um caminho de um nó de origem até o destino definido. A **etapa 8** trata da manipulação e iteração dos caminhos.

O caminho é formado por uma lista contendo todos os nós pertencentes a ele, onde o primeiro elemento é a origem e o último, o destino do trajeto. Portanto, a cada iteração no caminho, o nó corrente é atualizado.

3.1.4. Movimentação

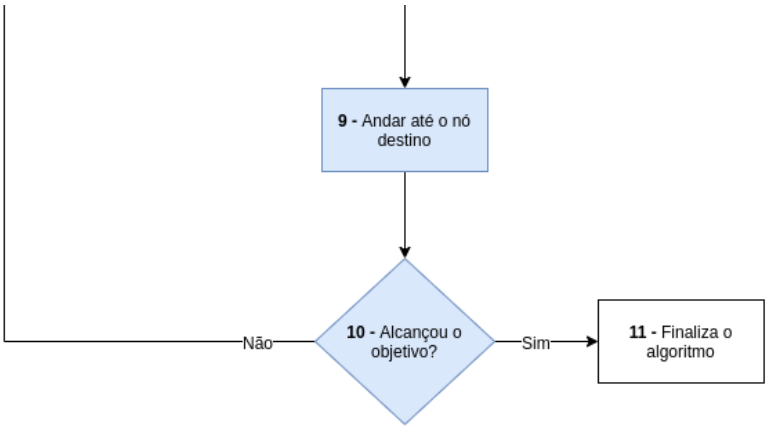


Figura 13. Etapa da movimentação destacada no fluxo

A Figura 13 possui destacada as etapas do fluxo que fazem parte do processo de movimentação, que é o quarto passo da abordagem proposta. A **etapa 9** é responsável pela rotação e movimentação do agente. Já a **etapa 10** é referente ao quinto passo da abordagem e verifica se o personagem alcançou o objetivo final. Abaixo segue o pseudocódigo destas operações:

1. Se o personagem não está encarando o nó destino
    a. Rotaciona a IA
2. Anda para frente
3. Se alcançou o objetivo final
    a. Termina o algoritmo
4. Senão
    a. Volta para a etapa de mapeamento

Importante salientar que a IA sempre se movimenta de acordo com sua orientação. Sendo assim, se ela está “olhando” para a direção norte e seu próximo destino está à direita, é preciso rotacioná-la 90º em sentido horário.

A partir da localização do próximo nó a ser visitado, o personagem é rotacionado, caso necessário. Após o agente “encarar” o nó destino, ele é movido em sua direção. Ao final do movimento, o agente estará posicionado no centro do nó de destino e seu nó atual será atualizado para este novo valor. Então, é feita uma verificação para descobrir se este nó representa o objetivo final. Em caso positivo, o algoritmo é finalizado (**etapa 11**); senão, retorna para a **etapa 1,** de mapeamento.

3.2. Implementação

Nesta seção mostraremos como se deu a implementação do algoritmo cuja lógica foi detalhada em 3.1. Inicialmente apresentaremos o ambiente de desenvolvimento e, em seguida, detalharemos partes de código da IA que julgamos importantes. O algoritmo implementado pode ser encontrado em[10]

[10] Disponível em <https://gitlab.com/gfribeiro/TCC-Labirinto.git.>.

3.2.1 O Ambiente

Como ambiente de desenvolvimento, escolhemos o motor de jogo Unity. A Unity é uma engine multiplataforma orientada a scripts escritos na linguagem de programação C#. Atualmente, é o motor de jogo mais utilizado do mundo[11]: cerca de 50% dos jogos elaborados nos dias de hoje são produzidos através da ferramenta[12].

Dada a popularidade entre os desenvolvedores de jogos e a facilidade, tanto em relação a linguagem de programação como pela estrutura de desenvolvimento da engine, decidimos pela Unity como nosso ambiente de desenvolvimento.

3.2.2. Controlador

AIBehaviour é o principal script da IA implementada, responsável por controlar os comportamentos fundamentais do agente citados na seção 3.1: mapeamento, escolha do destino, manipulação de caminho e movimentação.

A principal função deste script é a **MainBeahaviour()**, um método recursivo responsável por invocar todos os comportamentos citados acima. Após todos retornarem seus valores, a função chama a si mesma para completar o ciclo da Figura 7.

3.2.3. Mapeamento

A classe **NodeTable** é responsável por guardar e manipular o grafo criado a partir do grid citado em 3.1.1. Nela temos os seguintes atributos:

- **nodesList:** lista contendo todos os nós já descobertos;
- **nonVisitedNodes:** lista com todos os nós já descobertos e alcançáveis, mas ainda não visitados;
- **currentNodeIndex:** identificador do nó atual;

---

[11] PECKHAM, Eric. **How Unity built the world's most popular game engine**. Disponível em <https://techcrunch.com/2019/10/17/how-unity-built-the-worlds-most-popular-game-engine/>. Acesso em: ago. 2019.

[12] DILLET, Romain. **Unity CEO says half of all games are built on Unity**. Disponível em <https://techcrunch.com/2018/09/05/unity-ceo-says-half-of-all-games-are-built-on-unity/>. Acesso em: ago. 2019.

Para representar o nó, criamos a classe **Node**, que possui atributos dos tipos:

- **NodeIndex:** estrutura possuindo os componentes **linha** e **coluna** do nó. Serve para identificação e localização do mesmo no grafo.
- **NodeType:** enum que indica o tipo do nó. Auxilia na busca e na adição de um novo nó. Possui quatro possíveis estados:
    - *Discovered*: nó mapeado, mas não alcançável;
    - *Reachable*: nó mapeado e alcançável;
    - *Walked*: nó percorrido;
    - *Objective*: nó representando o destino final/objetivo.
- **frontConnection, backConnection, leftConnection e rightConnection:** do tipo **NodeConnection**, que representa a conexão com o nó da frente, de trás, da esquerda e da direita, respectivamente;

NodeConnection é a estrutura que auxilia na manipulação dos nós e suas conexões. Possui os seguintes atributos:

- **connectionStatus:** enum que informa o tipo da conexão, com 3 estados possíveis:
    - *NonDiscovered*: valor padrão, caso a conexão ainda não tenha sido verificada;
    - *Blocked*: indica conexões que estão bloqueadas por paredes ou objetos;
    - *Connected*: significa que dois nós estão conectados.
- **node:** nó que faz parte da conexão;

O método **AddNodeNoConnection()** é invocado quando se descobre um nó que não pode ser alcançado. Ele é responsável por adicionar novos nós à **NodeTable** e também por revisar os já existentes, adicionando as novas conexões, caso existam. A função recebe como parâmetro o **nodeIndex** do nó descoberto, o do nó “anterior” e a direção da conexão. O método funciona da seguinte forma:

1. Se o nó não existe
    a. Cria um novo nó, do tipo *Discovered*, e o adiciona na **nodesList**
2. Adiciona a nova conexão, de acordo com sua direção, como sendo do tipo *Blocked*

Similar ao método anterior, o **AddNodeConnection()** é uma função responsável por adicionar nós que podem ser alcançados. Ela também recebe como parâmetro o **nodeIndex** do nó descoberto, o do nó “anterior” e a direção da conexão. Segue seu passo a passo:

1. Se o nó existe
    a. Se o nó está marcado como “Discovered”
        i. Altera seu tipo para “Reachable”
        ii. Adiciona o nó na lista **nonVisitedNodes**
2. Senão
    a. Cria um novo nó, do tipo “Reachable”, e o adiciona na **nodesList**
    b. Adiciona o nó na lista **nonVisitedNodes**
3. Adiciona a nova conexão, de acordo com sua direção, como sendo do tipo “Connected”

A classe **MappingBehaviour** é responsável por ativar os sensores, detectando os nós e conexões disponíveis na posição atual do personagem e, então, repassá-los para a **NodeTable**. A ordem de ativação dos sensores foi definida como frente, esquerda e direita. Influenciando a ordem em que os nós descobertos são inseridos na lista.

Os sensores são controlados pela classe **SensorHitCheck**. O método **GetIsPathFree()** retorna um vetor de booleanas que informa se os nós adjacentes àquele sensor são alcançáveis ou não. O tamanho deste vetor varia de acordo com a profundidade do campo de visão. Como definimos este valor para 3, a posição 0 representa o nó mais próximo e a 2, o mais distante.

Abaixo, segue a estrutura do método **MapSurroundings()**, principal função da **MappingBehaviour**:

1. Aplica o **SensorHitCheck.GetIsPathFree()** para todos os sensores
2. Para cada resultado dos sensores
    a. Percorrer o vetor de resultados do sensor
        i. Se o resultado for verdadeiro
            1. Adiciona o nó a **NodeTable** pelo método **AddNodeConnection()**
        ii. Senão
            1. Adiciona o nó a **NodeTable** pelo método **AddNodeNoConnection()**

3.2.4. Escolha do Destino

Criamos uma estrutura, chamada de **Path**, a fim de auxiliar a escolha e a manipulação dos caminhos. Ela apoia tanto a execução do **A*** como a movimentação do personagem. A classe possui uma *lista ordenada*, onde o primeiro elemento é o nó de partida e o último, o nó destino de um caminho.

O **A*** foi introduzido no fluxo do Path Discovery com o objetivo de calcular, através da heurística aplicada por ele, o melhor caminho a ser percorrido dentre os destinos promissores. Como ressaltado na seção 2.2.2, é importante destacar que a função heurística leva em consideração as distâncias tanto do nó atual da iteração do **A*** até o nó inicial/partida como a distância até o nó destino, sendo adaptado para se encaixar à IA desenvolvida, como descrito:

- ***g***: calcula o custo da distância já percorrida desde o nó de partida até o corrente pela quantidade de elementos da *lista ordenada*;
- ***h***: utiliza a heurística h = |(vy - uy)| + |(vx - ux)|, adaptando o nó objetivo final para um nó parcial, ou seja, o nó destino utilizado se refere a um destino parcial, que são os nós promissores. O método que a calcula é a **Path.GetRawDistance()**, que recebe dois **NodeIndex**es como parâmetro e aplica a função heurística sobre eles;
- ***f***: calcula o custo *g* + *h*.

A escolha do destino se dá pelo candidato que possuir o **Path** com o menor número de elementos em sua lista de nós.

3.2.5. Manipulação de caminho

Quando se deseja obter o próximo nó de um caminho, utiliza-se o método **Path.GetNextDestination()**. Ele retorna o **NodeIndex** do próximo nó a ser visitado e incrementa o índice atual.

3.3. Considerações Finais

O algoritmo de Pathfinding **Path Discovery** foi desenvolvido visando suprir a necessidade de um algoritmo de busca não precisar conhecer o terreno e tê-lo completamente mapeado a priori. Essa característica se torna bastante importante quando se deseja elaborar jogos com certa dinamicidade. Para tanto, o Path Discovery possui uma adaptação de um dos algoritmos mais conhecidos e utilizados pela indústria de jogos, que é o **A***. Alguns aspectos como a tomada de decisão por um destino e um caminho assim como sua manipulação fizeram diferença para conseguir atingir o resultado desejado. Tais aspectos serão analisados e comparados aos outros algoritmos de busca citados durante este trabalho no capítulo 4.

CAPÍTULO 4 - ANÁLISE DOS RESULTADOS

Neste capítulo serão analisados os resultados do **Path Discovery** alcançados a partir de sua execução em três mapas/labirintos distintos, elaborados exclusivamente para os testes no ambiente de desenvolvimento Unity. Através de uma ferramenta online intitulada PathFinding[13], que simula uma animação da execução dos algoritmos de pathfinding abordados anteriormente (com exceção da **busca em profundidade**), extraímos os resultados gerados por ela com o objetivo de usá-las em uma análise comparativa.

A fim de uma comparação assertiva, os algoritmos foram executados nos mesmos terrenos base de teste, construídos na mesma ferramenta. Em todos os casos, levaremos em consideração para a análise a quantidade de passos que cada algoritmo levou para chegar no destino a partir de uma origem, assim como o caminho escolhido e percorrido para a solução em determinado mapa.

A fim de auxiliar a avaliação, foi desenvolvido um módulo a parte para visualização dos nós mapeados e também da tomada de decisão da IA. Utilizaremos este visualizador para realizar as comparações citadas anteriormente. Executado em paralelo com o **Path Discovery**, ele mostra todo o grafo mapeado até então a cada iteração do algoritmo de busca. Cada nó é composto por seus índices *linha* e *coluna* e também uma cor, onde cada uma representa:

- *Amarelo*: nó atual;
- *Verde*: nó já visitado;
- *Azul*: nó alcançável, mas não visitado;
- *Branco*: nó não alcançável.

As conexões dos nós, representadas pelas ligações entre eles, também possuem cores, que refletem:

- *Verde:* nó conectado (conexão do tipo *connected*);
- *Vermelho*: nó não conectado (conexão do tipo *blocked*);
- *Inexistente*: quando a conexão ainda não foi mapeada.

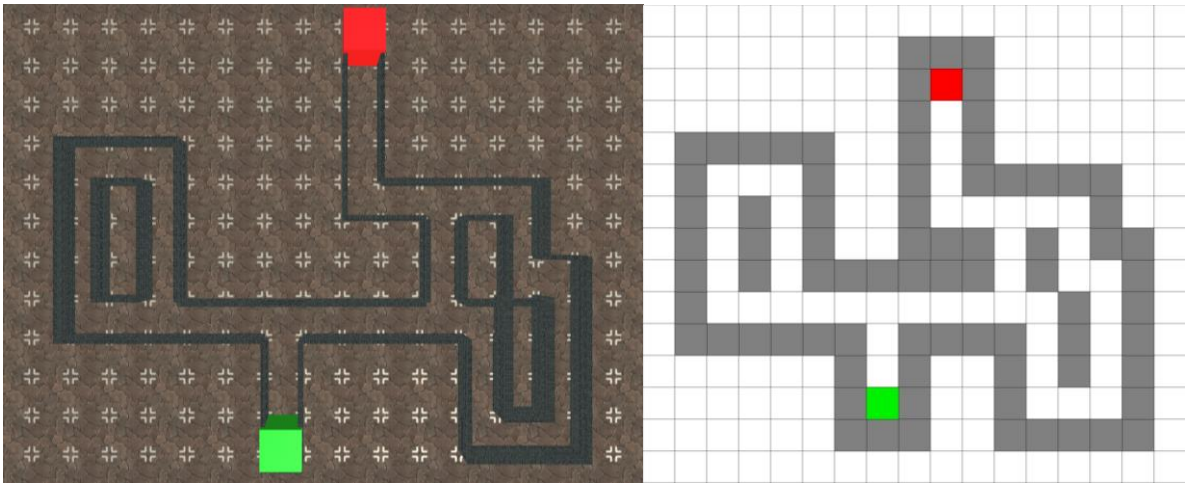
[13] BOTEA, Adi; et al. PathFinding. Disponível em <https://qiao.github.io/PathFinding.js/visual/>. Acesso em: ago. 2019.

Neste capítulo abordaremos três casos de teste, elaborados com certos objetivos, aplicados ao **Path Discovery** e também à **busca em largura**, **Dijkstra** e **A***, apoiados pelo uso da ferramenta online PathFinding. Para cada labirinto desenvolvido na engine Unity a fim de uma análise do comportamento do **Path Discovery**, o mesmo cenário foi replicado no PathFinding para análise dos outros algoritmos de busca, exceto a busca em profundidade, por limitação da ferramenta. Devemos destacar que cada grid no mapa Unity equivale a um grid da ferramenta. Desta forma, conseguiremos comparar a execução dos algoritmos de busca.

O capítulo está organizado da seguinte maneira: Seções 4.1, 4.2 e 4.3 mostram os casos de testes usados no experimento. Cada caso de teste possui uma seção contendo uma análise sobre a aplicação e andamento do **Path Discovery** apoiado pelo módulo de visualização e também uma segunda seção que compara os resultados da nossa abordagem aos dos algoritmos de busca conhecidos. Por fim, a seção 4.4 fala das considerações finais, abordando os pontos avaliados neste capítulo.

4.1. Caso de teste 1

Este mapa foi desenvolvido com o objetivo de verificar o comportamento da IA frente a múltiplas opções de caminho. O mapa foi criado em um grid 17x15 e cada célula é definida pelas coordenadas (X, Y). A coordenada (0,0) é o ponto inicial do agente. É importante observar e entender o porquê de cada decisão na escolha dos caminhos. E também verificar a capacidade de recuperação do agente, ou seja, encontrar outra rota levando até o objetivo após atestar que o caminho sendo seguido é improdutivo. A Figura 14 ilustra os mapas elaborados para testar os pontos levantados ressaltando os nós de partida do agente e o de objetivo, onde a Figura 14.a mostra o labirinto construído no Unity e a Figura 14.b na ferramenta online PathFinding.



(a) (b)

Figura 14. Representação do labirinto, onde o nó verde representa a posição do agente e o nó vermelho, o objetivo, (a) no Unity (b) no PathFinding

4.1.1. Path Discovery

A Figura 15 mostra alguns pontos da evolução, capturados pelo visualizador, durante a execução do **Path Discovery** baseado no mapa da Figura 14.a. O agente inicia em um corredor e até atingir o nó (3,0) possui apenas um caminho (seguir em frente). Ao alcançar este nó, como mostra a Figura 15, ele se depara com sua primeira escolha entre caminhos, uma vez que foram mapeados nós alcançáveis a sua esquerda e direita. Neste ponto, o agente possui dois caminhos possíveis de mesmo custo. Como mencionado em 3.1.2, a escolha de caminho em caso de empate é definida pelo maior *valor promissor*. O segundo critério de desempate é pela ordem de inserção dos nós na lista durante a etapa de mapeamento (frente, esquerda e direita), fazendo com que o agente opte andar para a esquerda seguindo o caminho dos nós não mapeados até alcançar o nó (5,-3), como ilustra a Figura 16.

Estando no vértice (4,-3), a IA escolherá o nó (3,1) como seu próximo destino dado que não existe nenhum outro nó não visitado alcançável por ele. Na Figura 17, quando a IA atinge o nó (6,4), o agente tem duas possibilidades, mas segue para o nó (6,3), que possui maior *valor promissor* dentre os candidatos. Assim, ele mantém o caminho até alcançar o objetivo em (10,2), como mostra a Figura 18.

Analisando o trajeto percorrido pela IA neste caso de teste, o **Path Discovery** se mostrou completo, mas não ótimo. Foram percorridos 43 nós para o agente alcançar o objetivo. Porém, o mesmo conseguiu alcançar o objetivo ao navegar por um terreno completamente desconhecido sem possuir qualquer informação sobre a

localização do objetivo. Nesse caso em particular, o labirinto acabou sendo todo mapeado, mesmo que o agente não tenha visitado todos os nós ((4,4) e (5,4) foram mapeados, mas não foi necessário visitá-los).

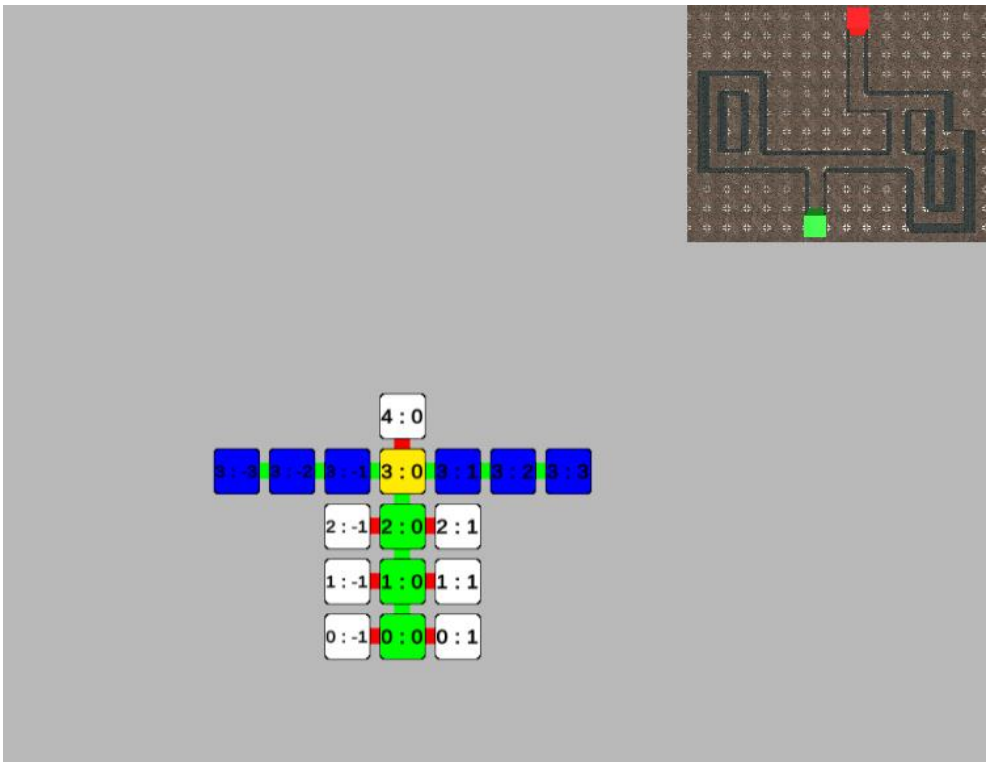


Figura 15. Caminho inicial do agente

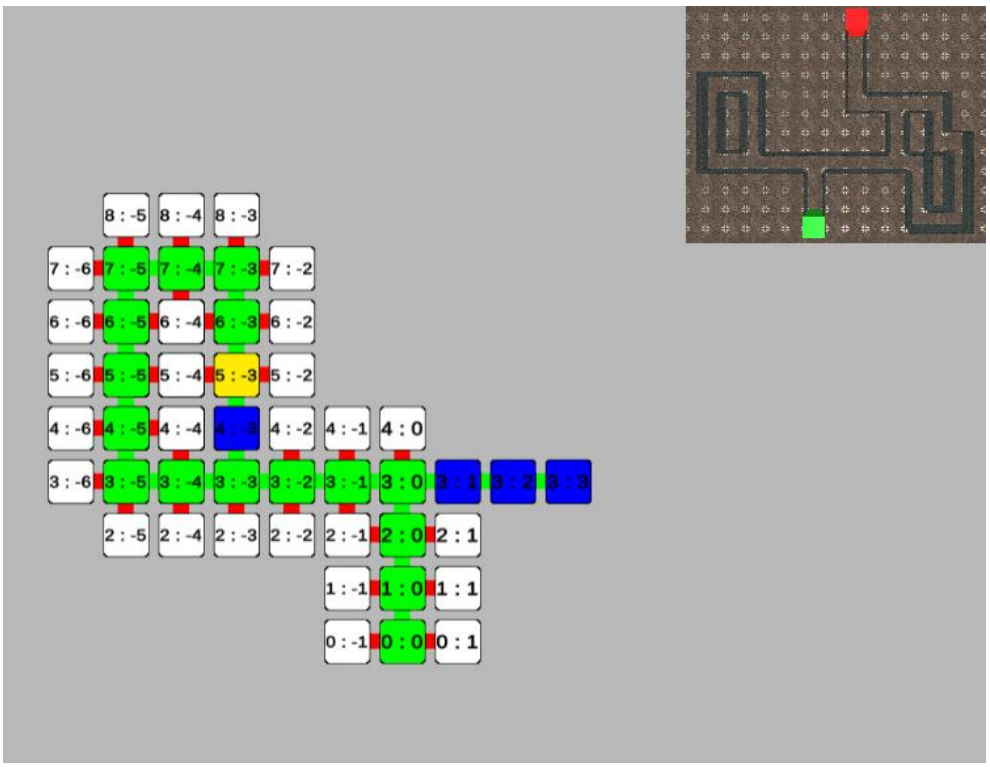


Figura 16. Retomada de decisão da IA ao se deparar com um caminho já visitado

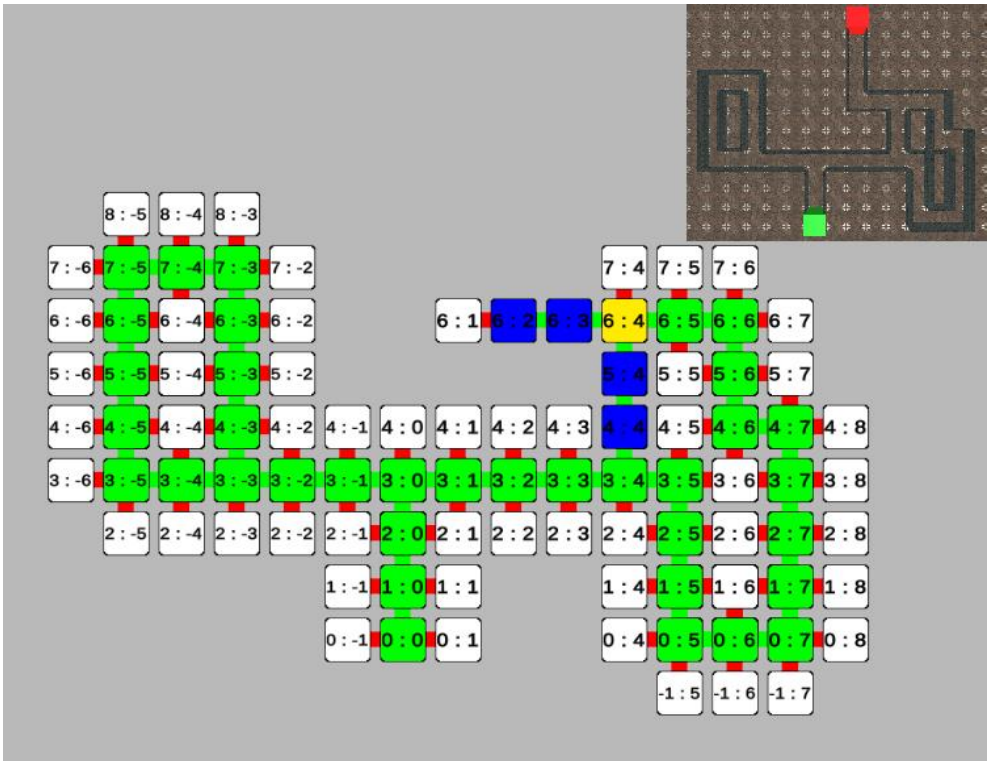


Figura 17. Momento em que a IA se decide por um caminho mais promissor, seguindo nó (6,3)

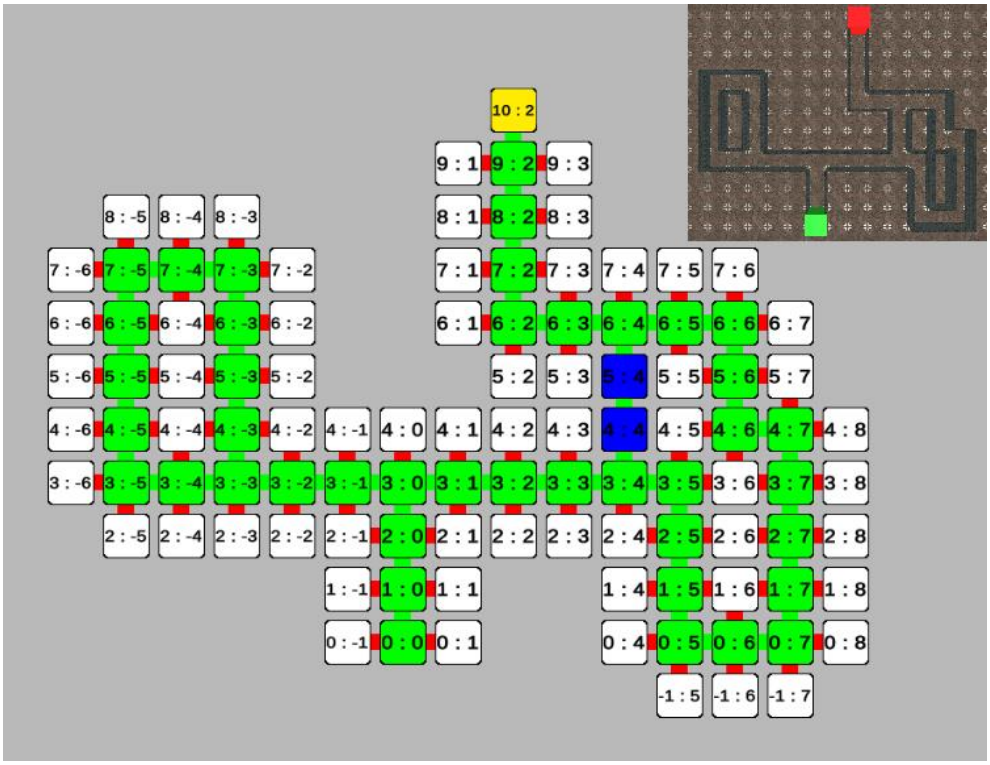


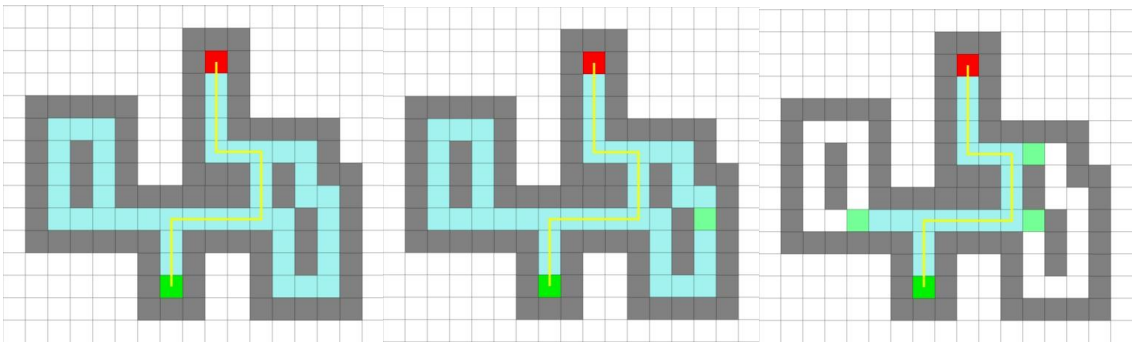
Figura 18. Finalização da IA com o agente encontrando o nó destino em (10,2)

4.1.2. Comparação

A Figura 19 mostra os resultados da execução da **busca em largura**, **Dijkstra** e **A***, respectivamente. Nela e nas imagens subsequentes, as cores dos nós indicam:

- *Verde*: ponto de partida;
- *Verde claro*: nós que marcam caminhos não promissores;
- *Azul*: nós testados como parte da solução;
- *Azul traçado por linha amarela*: nós que fazem parte da solução.

É possível observar na figura abaixo que os três algoritmos foram completos e retornaram a solução ótima. No entanto, enquanto a **busca em largura** e o **Dijkstra** visitaram 45 nós para chegar na solução, o **A*** visitou apenas 22.



(a) (b) (c)

Figura 19. Execução dos algoritmos baseada no mapa da Figura 14.b, onde (a) representa a busca em largura, (b) representa o algoritmo Dijkstra e (c) representa o A*

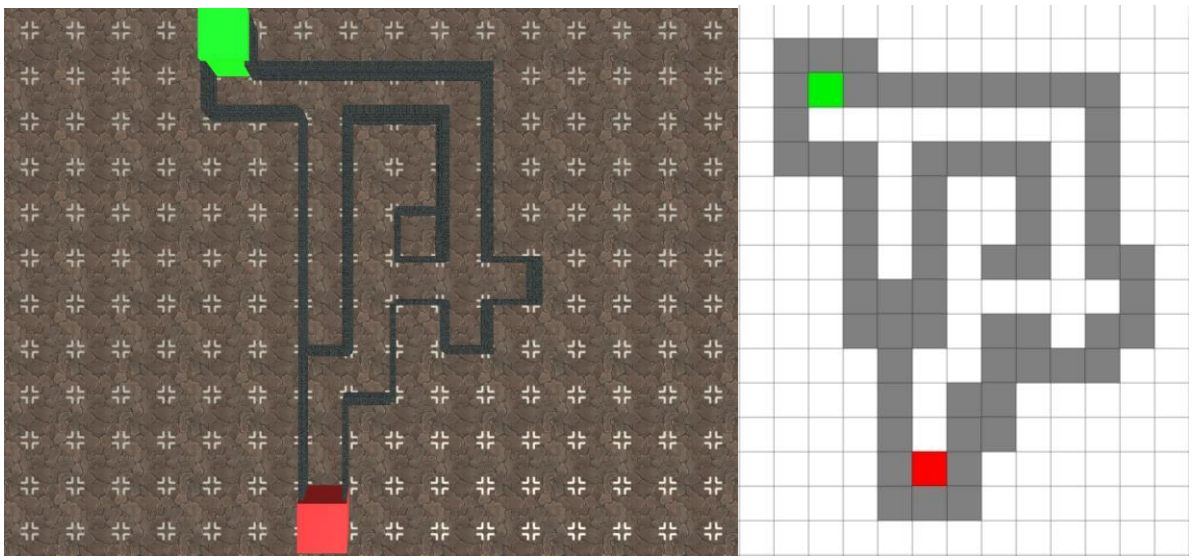
Comparando a execução dos algoritmos com o **Path Discovery**, é percebido que a quantidade de nós visitados pela nossa abordagem é inferior (43 nós visitados) a do **Dijkstra** (45 nós) e da **busca em largura** (45 nós), mas superior a do **A***, como era esperado. A **busca em largura** e o **Dijkstra** encontram a solução ótima, mas utilizam uma abordagem exaustiva. Mesmo conhecendo o nó destino, não são guiadas por ele em suas tomadas de decisão. Já o **A***, por saber a localização do objetivo, se comporta muito bem porque é orientado por ele, evitando caminhos que irão distanciá-lo do destino. O **Path Discovery** se mostrou eficiente, dado que o agente, por não possuir conhecimento do objetivo, explorou o mapa da melhor forma

possível, evitando um corredor (representado pelos nós em azul (5,4) e (4,4)) que levaria a um lugar já conhecido.

4.2. Caso de teste 2

Este ambiente foi elaborado pensando em verificar a escolha inteligente de caminhos do **Path Discovery**. Como mostra a Figura 20, no centro do labirinto há um espaço que não leva a lugar nenhum, similar a um beco sem saída, justamente para verificação de um comportamento mais humanizado. Ele foi propositalmente colocado nesta posição para explorarmos um recurso importante da nossa abordagem. Ao percorrer o labirinto, o agente mapeia as laterais do beco, indicando posteriormente que aquele espaço não leva ao objetivo.

Ou seja, isso seria similar a um humano fazer um esboço do labirinto durante a sua navegação. Como nessa situação específica foram mapeados os contornos da região não explorada, uma pessoa assumiria que tentar um caminho por ali resultaria em uma possibilidade menor de encontrar novos nós. Importante salientar que por conta de limitações da ferramenta online, não conseguimos replicar exatamente o mapa criado no Unity.



(a) (b)

Figura 20. Representação do labirinto, onde o nó verde representa a posição do agente e o nó vermelho, o objetivo, (a) no Unity (b) no PathFinding

4.2.1. Path Discovery

A Figura 21 ilustra um momento de tomada de decisão de caminhos. Neste caso, havendo empate entre os nós candidatos que possuem mesmo valor promissor, a IA se decide pelo nó a frente, como relembrado na seção 4.1.1. Na Figura 22 fica evidente o fato de que a IA não enxerga lateralmente. Em outras palavras, apesar de saber que não será possível seguir em frente partindo do nó (-6,5), ela precisa visitá-lo a fim de confirmar que não há novos caminhos para a esquerda e direita daquele nó.

Ao atingir a entrada do beco no nó (-5,3), como mostra a Figura 23, a IA se decide pelo nó que possui maior *valor promisso*r entre os nós (-4,3) e (-6,3). Vale lembrar que o *valor promissor* é o somatório de toda nova conexão passível de ser descoberta a partir de um nó. Ao analisarmos o *valor promissor de* (-4,3) chegamos ao valor 3 a partir de: 2 proveniente do nó (-4,2) + 1 vindo do nó (-4,4). Já o nó (-6,3) possui *valor promissor* de 4, sendo: 3 vindo do nó (-6,2) + 1 proveniente de (-6,4).

Para garantir que o critério de escolha, neste caso, foi decidido pelo *valor promissor* e não pela ordem de inserção na lista, alteramos a etapa de mapeamento de forma que os nós a direita do agente fossem detectados antes dos nós a esquerda. Mesmo assim a IA optou por seguir para (-6,3), o que comprovou que sua escolha foi guiada pelo nó mais promissor.

Importante salientar que o agente não descartou o nó (-4,3), apenas o classificou como menos promissor. Caso o caminho escolhido se revele um beco sem saída, a IA retornará e irá explorar o nó (-4,4).

Por fim, a imagem na Figura 24 ilustra a IA atingindo o nó objetivo.

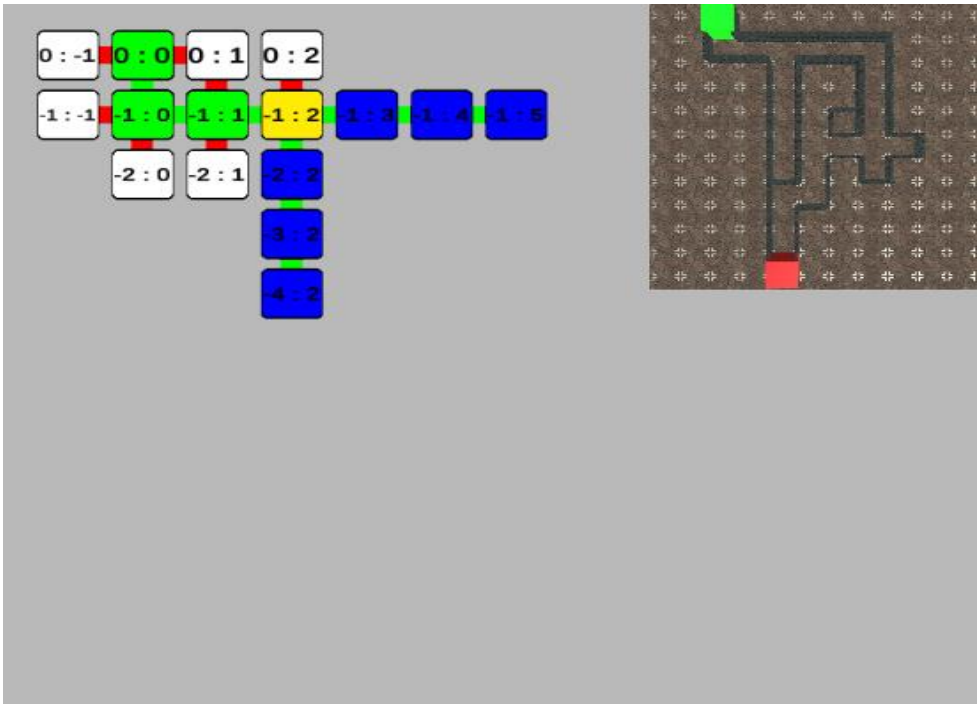


Figura 21. Início da execução em um momento de tomada de decisão de caminhos

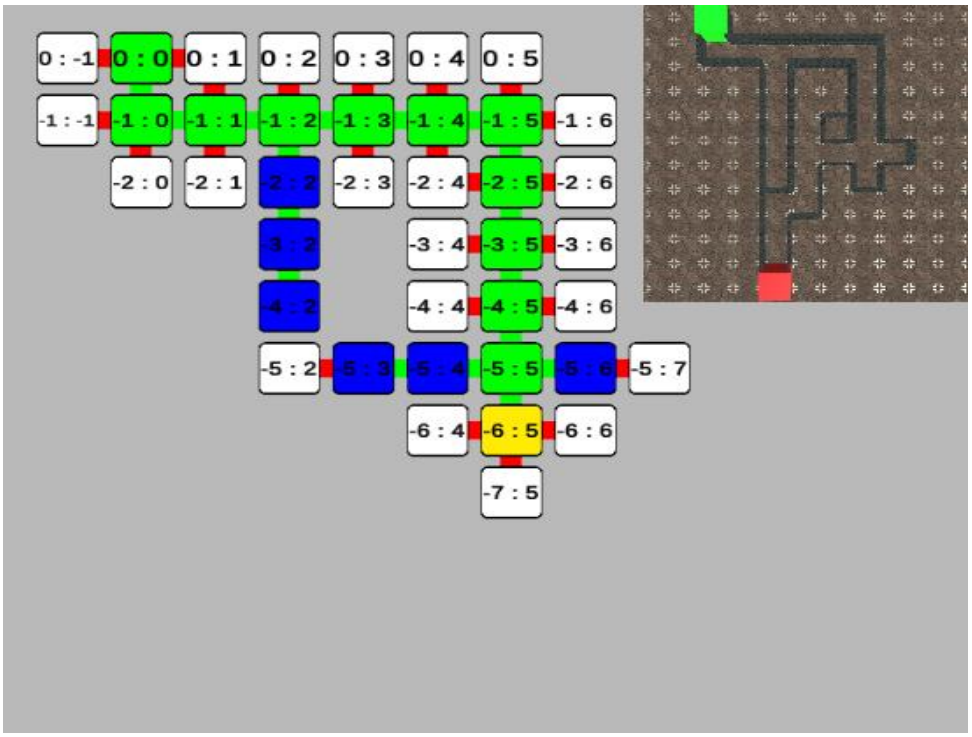


Figura 22. Situação de retomada de decisão

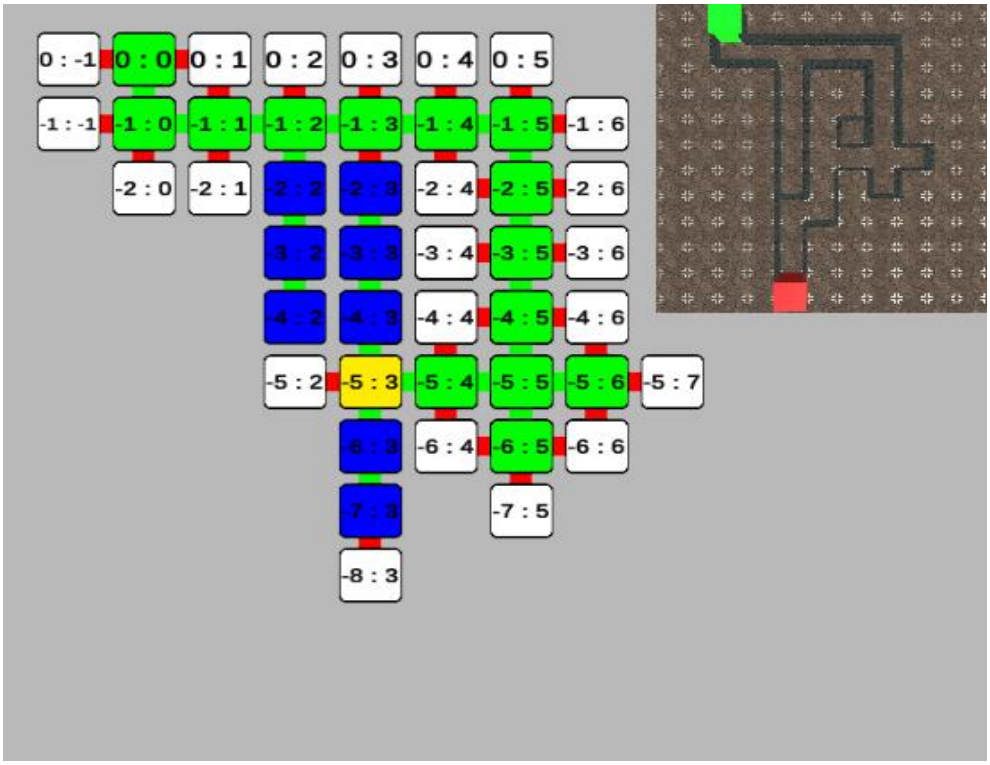


Figura 23. Tomada de decisão inteligente por parte da IA

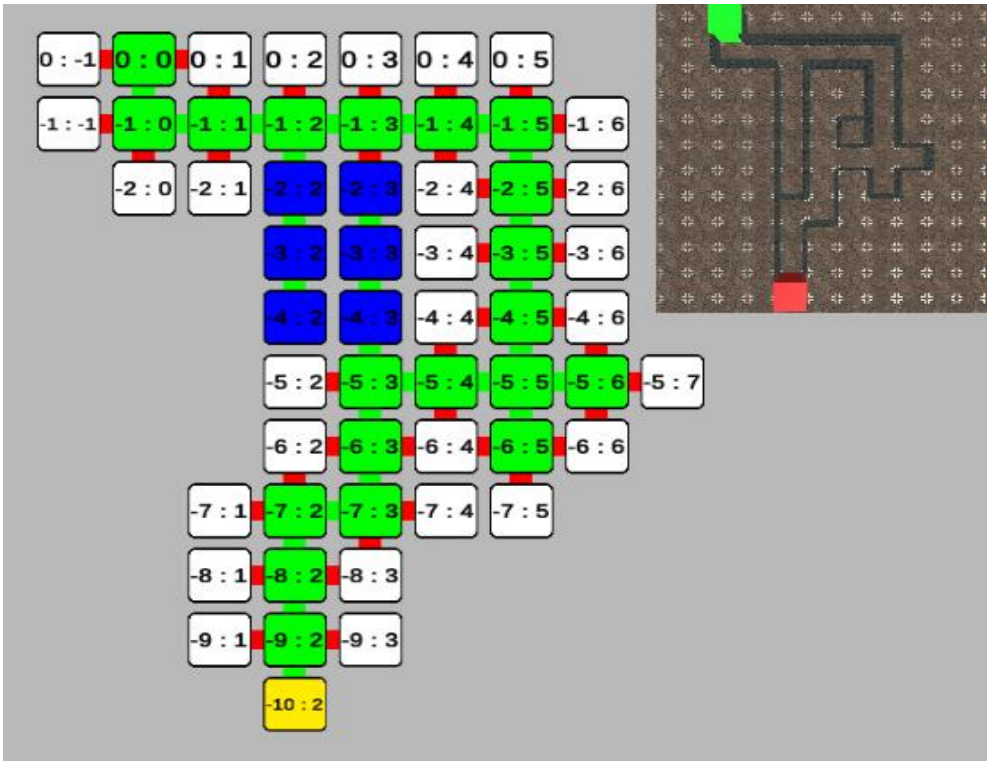
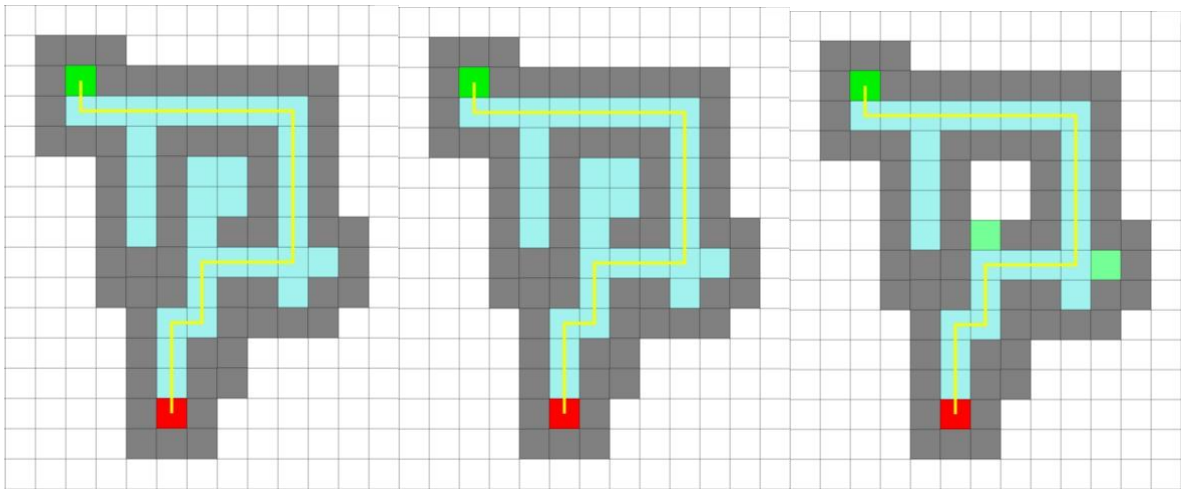


Figura 24. Finalização da IA com o agente encontrando o nó destino em (10,2)

4.2.2. Comparação



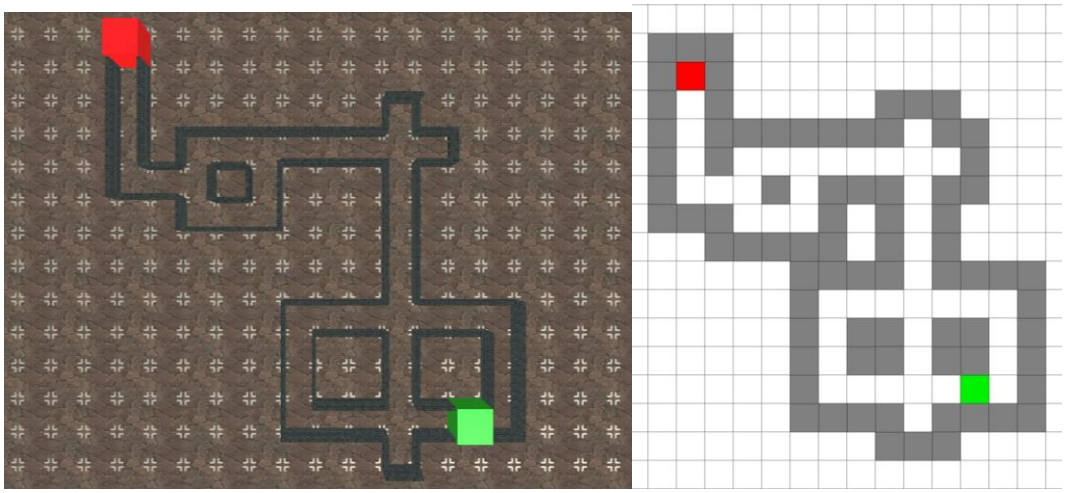
(a) (b) (c)

Figura 25. Execução dos algoritmos baseada no mapa da Figura 20.b, onde (a) representa a busca em largura, (b) representa o algoritmo Dijkstra e (c) representa o A*

A Figura 25 ilustra que a **busca em largura** e o **Dijkstra** completaram o percurso visitando todos os nós possíveis. Diferentemente deles, como demonstrado pela Figura 25.c, o **A*** completou a busca sendo inteligente e não visitando o espaço que simula o beco sem saída. No entanto, como a busca é orientada pela função heurística envolvendo o custo do nó atual até o nó objetivo, na primeira bifurcação encontrada, o **A*** optou por seguir o caminho para baixo que, aparentemente, levaria ao destino (de acordo com a heurística). Comparando à execução do **Path Discovery**, este se mostrou bastante eficiente justamente por não visitar caminhos menos promissores, como detalhado na seção 4.2.1, mesmo não sabendo onde está localizado o destino.

4.3. Caso de teste 3

O labirinto neste caso de teste foi elaborado visando analisar a retomada de caminhos pela IA a partir da inclusão de elementos que obrigam o agente a retornar por nós já visitados a fim de encontrar novos caminhos promissores. Também, foram adicionados elementos que poderiam levar a caminhos cíclicos infinitos com o objetivo de analisar o comportamento da IA nesses casos. A Figura 26 demonstra o mapa representado na Unity e também na ferramenta PathFinding.



(a) (b)

Figura 26. Representação do labirinto, onde o nó verde representa a posição do agente e o nó vermelho, o objetivo, (a) no Unity (b) no PathFinding

4.3.1. Path Discovery

A sequência de figuras a seguir apresenta a evolução do algoritmo até atingir o nó objetivo. Como mostra a Figura 27, o agente inicia o processo em (0,0) e opta por seguir em frente devido ao critério de desempate mencionado anteriormente, que é pela ordem de inserção quando os nós possuírem o mesmo *valor promissor*. A Figura 28 ilustra que a IA não ficou presa no ciclo e, ao perceber que não consegue seguir em frente a partir do nó (-1,2), acaba retornando para o nó (1,-2) a fim de continuar a busca.

Similar ao caso anterior, a Figura 29, que mostra a IA visitando o nó (9,-2) e, não podendo seguir em frente, retoma a busca optando visitar o nó (8,-3), que leva a um caminho com mais possibilidades, de acordo com seu *valor promissor*. A Figura 30 mostra a IA seguindo o padrão de escolha de caminho descrito anteriormente. Ela seguirá dando a volta até ter percorrido todos os nós daquele círculo até atingir novamente o nó (7,-8), se decidindo pelo caminho que levará ao objetivo, que no caso é o nó (7,-9), ilustrado na Figura 31.

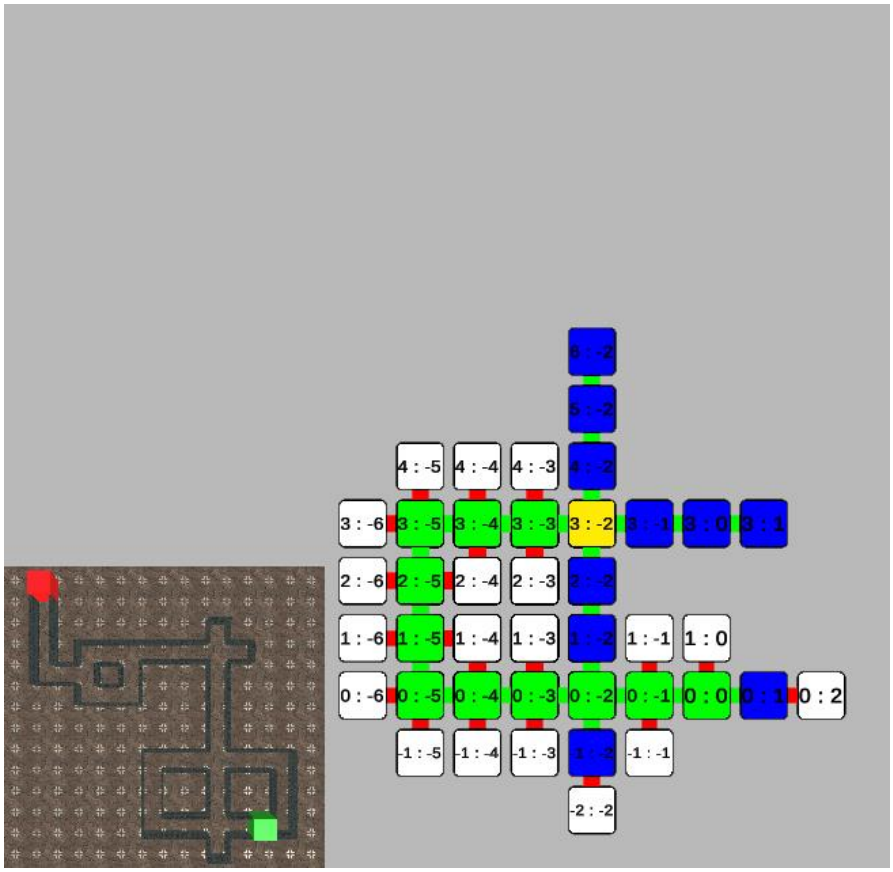


Figura 27. Escolha de caminho da IA

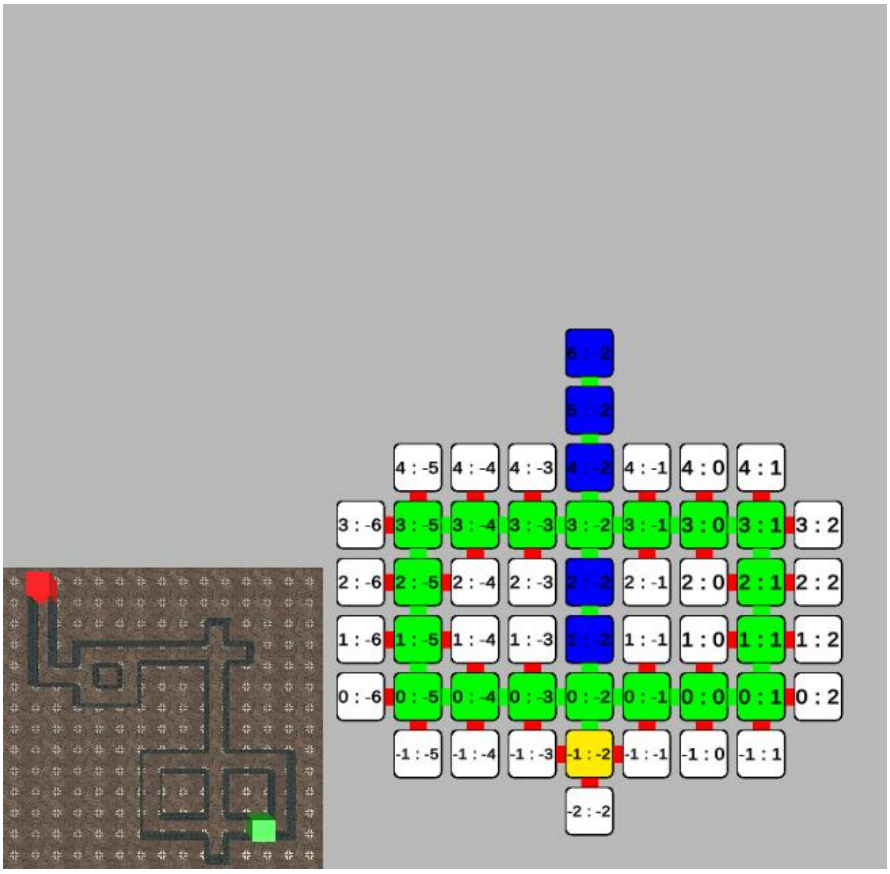


Figura 28. Momento em que a IA não entra em um ciclo, se recuperando

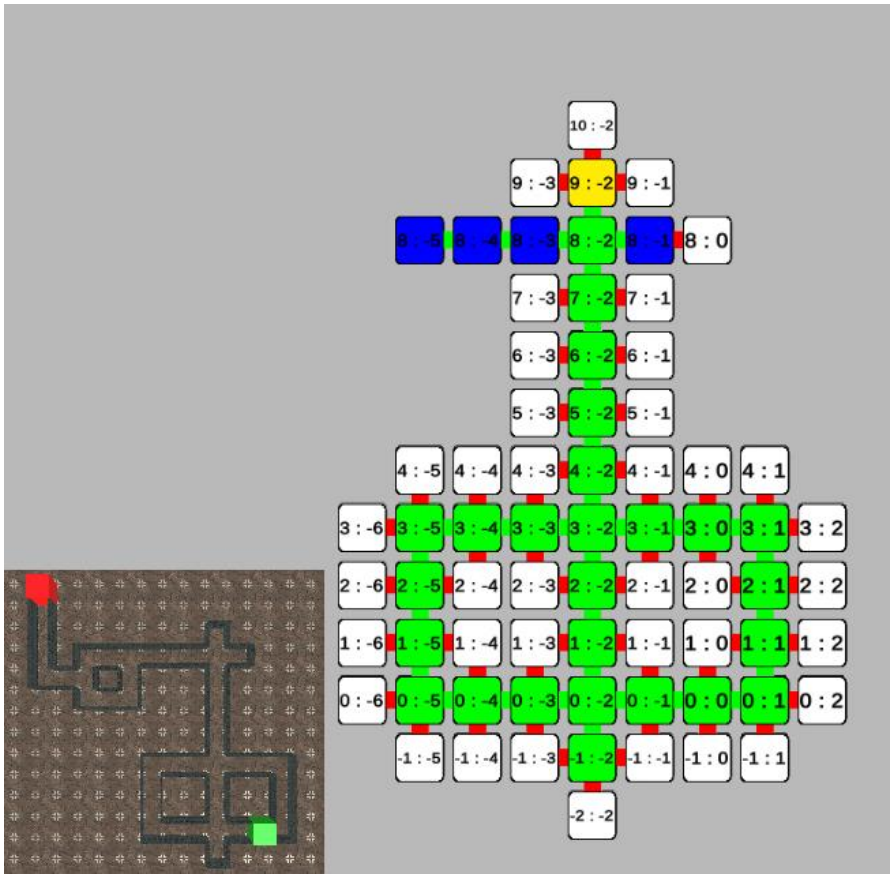


Figura 29. Retomada de decisão da IA

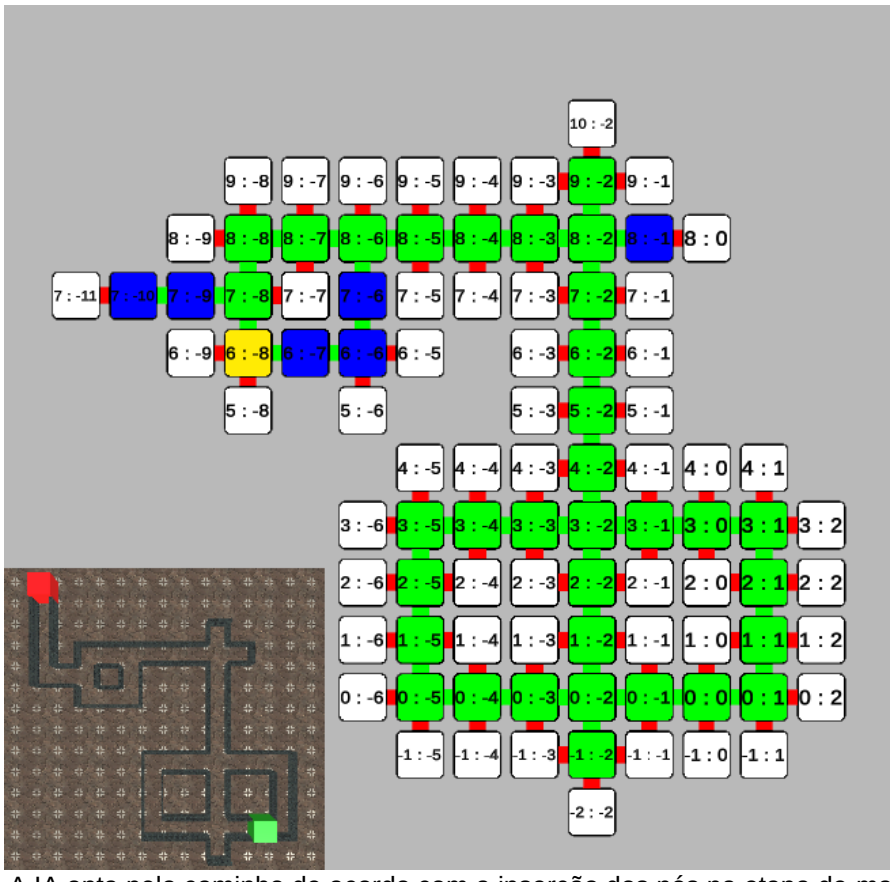


Figura 30. A IA opta pelo caminho de acordo com a inserção dos nós na etapa de mapeamento

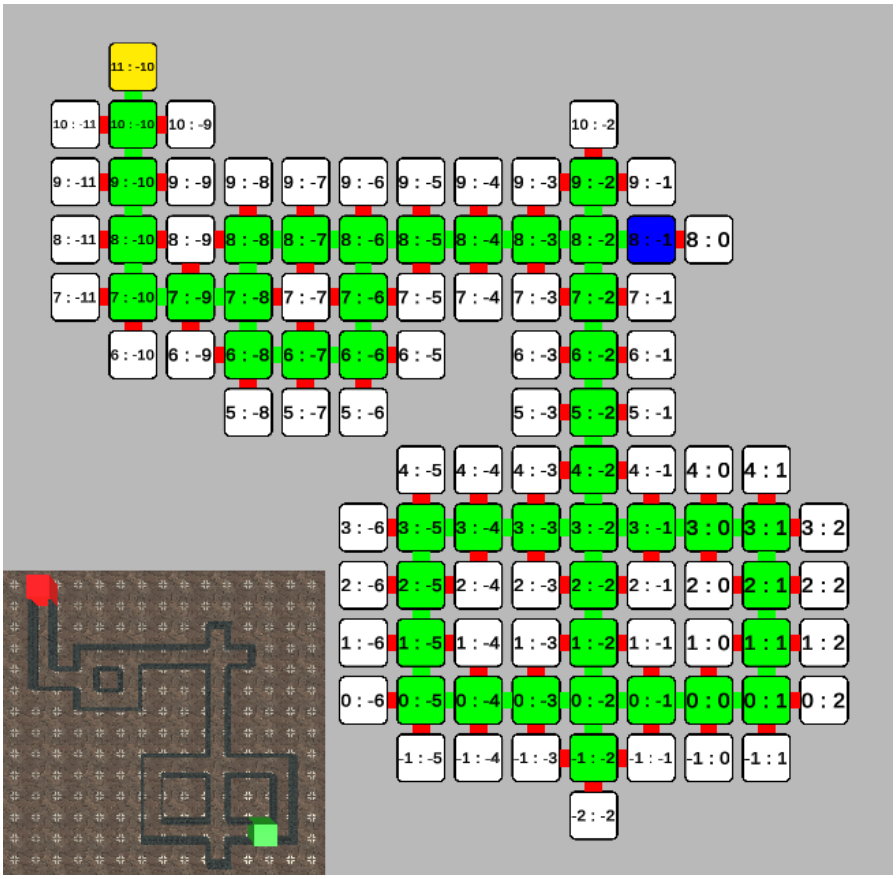
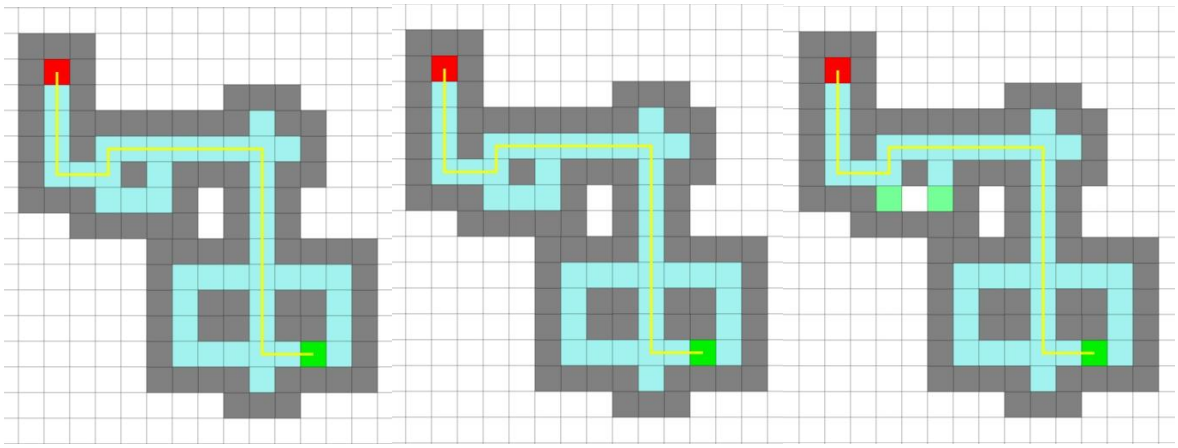


Figura 31. Finalização da IA com o agente encontrando o nó destino em (11,-10)

4.3.2. Comparação



(a) (b) (c)

Figura 32. Execução dos algoritmos baseada no mapa da Figura 26.b, onde (a) representa a busca em largura, (b) representa o algoritmo Dijkstra e (c) representa o A*

No exemplo da Figura 32, todos os algoritmos obtiveram resultados parecidos em relação à quantidade de passos dados e encontraram a solução ótima. A **busca**

em largura e o **Dijkstra** percorreram todos os nós do mapa, totalizando 45 passos, enquanto o **A*** visitou 44 nós até encontrar a solução.

Em relação à execução do **Path Discovery**, que percorreu 44 nós diferentes para atingir o destino, observamos que alguns nós foram visitados mais de uma vez devido à retomada de decisão por parte da IA, mas não comprometeu a busca a ponto de fazer com que ela ficasse presa em ciclos. Vale destacar que o agente completou ciclos com o intuito de voltar a um ponto novo e seguir um caminho ainda não explorado. Em todos os casos, os algoritmos se mostraram bastante similares.

4.4. Considerações Finais

Durante o desenvolvimento do **Path Discovery** e analisando os casos de teste e resultados, pudemos observar e constatar os seguintes pontos:

- **Mapeamento funcionou de forma correta:** o algoritmo mapeou precisamente todos os nós e suas conexões detectadas pelos sensores;
- **Priorização de caminhos funcionou corretamente:** nos momentos em que houve dúvida quanto a qual caminho seguir, o *valor promissor* foi utilizado para ajudar o agente a definir seu novo destino. Por exemplo, como mostrado em 4.2.1, o agente preferiu um caminho, que posteriormente levou ao objetivo, a outro que levava a um beco sem saída;
- **Não ficou preso em caminhos cíclicos:** a IA foi perspicaz a ponto de evitar caminhos que levariam a ciclos infinitos, como demonstrado em 4.3.1;
- **Sempre foi capaz de encontrar o objetivo mesmo não sabendo sua localização:** em todos os casos, o algoritmo atingiu o destino final. Desde que exista um caminho até o nó objetivo, o algoritmo irá encontrá-lo;
- **A ordem de mapeamento muitas vezes influenciou na escolha dos caminhos:** em todos os casos onde o *valor promissor* dos nós candidatos era igual, a ordem de inserção dos nós na lista (durante a etapa de mapeamento) foi utilizada como critério final de desempate;
- **Coerência:** todas as execuções do algoritmo, em um mesmo caso de teste, retornaram o mesmo resultado;

- **Não garante a solução ótima:** o algoritmo proposto não é capaz de retornar a solução ótima em todos os casos. Mas vale ressaltar que sem o conhecimento do terreno nem da posição do objetivo esta tarefa é impossível;
- **Performance:** o algoritmo apresentado necessita de mais processamento se comparado aos demais conhecidos. Além do custo extra da etapa de mapeamento, a escolha do próximo nó destino é muito custosa por executar o **A*** para todos os nós candidatos;
- **Complexidade:** Calculamos sua complexidade como $O(D*(n + p*b^d))$, onde:
    - *D*: número total de passos para alcançar a solução;
    - *n*: média de nós mapeados entre todas as iterações;
    - *p*: número médio de nós promissores por iteração;
    - *b*: fator ramificação;
    - *d*: profundidade média da solução dos candidatos;
- **Solução para múltiplos caminhos**:** a IA foi implementada de maneira que buscasse apenas um objetivo. No entanto, uma vez que a busca não é orientada pelo destino final, podemos facilmente adaptá-la para continuar buscando por nós objetivos;
- **Tipo de busca:** direcionada pelo custo e pela função heurística que retorna o *valor promissor*.

CAPÍTULO 5 - CONCLUSÃO

Diante de um contexto onde os algoritmos de busca conhecidos precisam necessariamente conhecer de antemão a localização exata do objetivo, nasceu a motivação para o planejamento e desenvolvimento do **Path Discovery**. A IA implementada se difere das demais a partir do momento em que não precisa ter conhecimento da posição do destino dentro do mapa. Para isso, o mapeamento do terreno é feito em tempo real, possibilitando maior dinamicidade nos jogos e também tomadas de decisão mais humanizadas.

Cabe destacar que o objetivo do trabalho não foi voltado a encontrar uma alternativa mais eficiente e otimizada para pathfinding, e sim desenvolver um algoritmo que independe do mapeamento e da localização do objetivo. Temos total conhecimento da deficiência em performance do **Path Discovery** quando comparada ao **A*** ou até mesmo a **busca em largura**. Mas é importante ressaltar que, sem o mapeamento prévio, estes algoritmos não funcionam.

Uma das preocupações no desenvolvimento desta IA foi a do processo de escolha de caminho. Um dos objetivos foi fazer com que o agente avaliasse os nós por onde passou e usasse estas informações para determinar o melhor caminho a ser seguido, aproximando-se de um raciocínio humano. Avaliamos a heurística que calcula o *valor promissor* como um dos grandes diferenciais do algoritmo pois mostrou resultados satisfatórios quando colocada em xeque.

Sendo assim, as contribuições deste trabalho são a construção de um algoritmo de busca que difere dos demais principalmente por independer do conhecimento da localização do nó destino e também por possibilitar a abertura de novas portas para estudos na área de Pathfinding, permitindo que esta solução seja aprimorada ou outras sejam encontradas a partir dela.

A Tabela 2 resume e completa as comparações entre o **Path Discovery** e os algoritmos de busca conhecidos, destacando as características da nossa abordagem.

Tabela 2. Resumo comparativo entre os algoritmos conhecidos e o Path Discovery

	Busca em Largura	**Busca em Profundidade**	**Dijkstra**	**A***	**Path Discovery**
Performance	$O(b^d)$	$O(b^d)$	$O(b^{c/m})$	$O(b^d)$	$O(D*(n + p*b^d))$
Completude	sim	sim	sim	sim	sim
Solução ótima	sim*	não	sim	sim	não
Solução para múltiplos destinos	sim	sim	sim	sim	sim**
Conhecimento prévio d*o terreno	sim	sim	sim	sim	**não**
Tipo de busca	não direcionada	não direcionada	direcionada	direcionada	direcionada

Fonte: Autoria própria, 2019.
**Inicialmente o algoritmo não resolve múltiplos destinos, mas uma pequena alteração é suficiente para isso.

5.1. Limitações

Principalmente pela otimização não ter sido um ponto priorizado durante o desenvolvimento do algoritmo, o alto processamento durante sua execução se torna uma limitação quando se trata de mapas muito grandes. Uma outra deficiência é relacionada ao mapeamento dos nós: haver sensores que mapeiam apenas a frente, esquerda e direita do nó limita o campo de visão do agente, fazendo com que ele não enxergue conexões em suas “laterais”. Também ressaltamos que atualmente o algoritmo só é garantido de funcionar em ambientes que podem ser abstraídos em grids quadrangulares. Por último, a heurística utilizada, embora contribua para o comportamento esperado da IA, é insuficiente e deveria ser aprimorada com o objetivo de fazer a IA escolher o melhor nó candidato possível.

5.2. Trabalhos futuros

Apesar do **Path Discovery** representar um grande avanço na área, ainda há melhorias a serem discutidas e pensadas. Algumas delas são referentes à

implementação, como: melhoria da função heurística que retorna o *valor promissor*; combinação do custo e *valor promissor* para escolha de um nó candidato, deixando de ser apenas um critério de desempate; otimização em termos de processamento da IA a partir da memorização de caminhos; implementação de sensores nas direções diagonais do agente a fim de aumentar seu campo de visão; também, deve-se considerar aplicar aleatoriedade como critério final de desempate, dado que atualmente a IA segue a ordem de inserção dos nós na lista durante a etapa de mapeamento (frente, esquerda e direita); e, por último, seria interessante implementar a resolução para casos de múltiplos objetivos (como, por exemplo, possuir vários objetivos a serem atingidos em uma determinada ordem ou então um caso onde é necessário apenas alcançar um deles para que o jogo finalize).

Por outro lado, alguns aperfeiçoamentos referentes a estudos e testes podem ser feitos, como: analisar a execução do algoritmo utilizando outros tipos de grid e, principalmente, a elaboração de novos casos de teste em outros ambientes, diferentes de labirintos. Também, enriqueceria o trabalho caso fossem realizados testes com humanos, com o objetivo de analisar e comparar os comportamentos da IA com pessoas de verdade.

Mesmo necessitando de melhorias, estamos orgulhosos e satisfeitos com os resultados alcançados pelo algoritmo **Path Discovery**.

REFERÊNCIAS BIBLIOGRÁFICAS

ALGFOOR, Zeyad Abd; SUNAR, Mohd Shahrizal; KOLIVAND, Honshang. A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. **International Journal of Computer Games Technology**, v. 2015.

BOTEA, Adi; et al. Pathfinding in Computer Games. **IBM Research**, Dublin, 1998.

_____. **PathFinding**. Disponível em <https://qiao.github.io/PathFinding.js/visual/>. Acesso em: ago. 2019.

CUI, Xiao; SHI, Hao. An Overview of Pathfinding in Navigation Mesh. **International Journal of Computer Science and Network Security**, Australia, v. 12, n. 12, dez. 2012.

_____. A*-based Pathfinding in Modern Computer Games. **International Journal of Computer Science and Network Security**, Australia, v. 11, n. 1, jan. 2011.

DILLET, Romain. **Unity CEO says half of all games are built on Unity**. Disponível em <https://techcrunch.com/2018/09/05/unity-ceo-says-half-of-all-games-are-built-on-unity/>. Acesso em: ago. 2019.

GRAHAM, Ross; McCABE, Hugh;SHERIDAN, Stephen. Pathfinding in Games. **The ITB Journal**, v. 4, n. 2, 2003.

KORF, Richard E. **Artificial Intelligence Search Algorithms**. Los Angeles: Universidade da Califórnia, jul. 1996.

PECKHAM, Eric. **How Unity built the world's most popular game engine**. Disponível em <https://techcrunch.com/2019/10/17/how-unity-built-the-worlds-most-popular-game-engine/>. Acesso em: ago. 2019.

SILVER, David. **Cooperative Pathfinding**. Edmonton: Universidade de Alberta, 2005.

ANEXO 01

ANÁLISE DA COMPLEXIDADE DO PATH DISCOVERY

Cada iteração do algoritmo consiste em:

1. Mapear o terreno;
2. Definir um próximo nó destino.

Em (1) é utilizado um FOR para percorrer a lista de nós e atualizá-la com as novas conexões descobertas. Com isso, deduzimos que sua complexidade é O(n), onde *n* é o número de nós já mapeados.

Em (2), temos 2 etapas:

A. É definida uma lista de nós candidatos, a partir de um FOR que percorre os nós já mapeados;
B. É aplicado o A* para cada um dos candidatos.

A complexidade da etapa (A) é O(n), já a de (B) é dita pela complexidade do algoritmo A* vezes o número de candidatos. Sendo assim, a complexidade de (B) é $O(p*b^d)$, onde *b* é o fator ramificação, *d* é a profundidade da solução e *p* o valor médio de candidatos. Por fim, definimos a complexidade de (2) como $O(n + p*b^d)$.

Já a complexidade total do algoritmo é dada pela complexidade de cada iteração (A + B) * número de iterações. Portanto, temos que a complexidade final do algoritmo é de $O(D*(n + p*b^d))$, onde:

- *D*: número total de passos para alcançar a solução;
- *n*: média de nós mapeados entre todas as iterações;
- *p*: número médio de nós promissores por iteração;
- *b*: fator ramificação;
- *d*: profundidade média da solução dos candidatos;