Relatório Trabalho Prático 1 - Inteligência Artifical
Algoritmos de Busca no Pac-Man

Daniel Souza de Campos[1] **- 2018054664**

[1]Departamento Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

danielcampos@dcc.ufmg.br

1. Introdução

Um agente inteligente pode ser visto como um ser que, dado o ambiente no qual foi inserido e munido de objetivos, consegue aventurar-se e influenciar esse ambiente. Para desbravar o ambiente, o agente capta e analisa sinais via seus sensores e consegue atuar, podendo influenciar o mundo, por meio dos seus atuadores. Uma situação comum nesses sistemas é a de o agente querer chegar em um estado, que é uma combinação de aspectos do ambiente, que seja mais favorável do que o estado atual conforme o seu propósito nesse ambiente. Dessa forma, surgem algumas questões a serem tratadas.

O problema geral é o de buscar um caminho até a posição objetiva a partir da posição atual do agente. A posição objetiva pode ser conhecida ou não para o agente, o que influencia na abordagem de procura de caminho e o seu custo. Esse problema possui diversas aplicações como em um carro autônomo, na qual o ambiente é parcialmente observável, ou um personagem em um jogo, que pode conhecer completamente o mundo ao seu redor, tentando chegar ao seu destino. Neste trabalho, a situação levada em consideração é o agente PacMan em um tabuleiro quadriculado a procura de uma, ou mais, posição que contém uma recompensa.

Apesar de específica, a situação descrita pode ser vista como um modelo para outras situações ao englobar questões como o custo de se tomar uma ação (movimento para uma posição vizinha), barreiras (paredes que impedem movimento entre posições) e saber a posição do alvo de antemão ou não.

2. Metodologia e Análise Experimental

2.1. Ferramentas Utilizadas

O trabalho foi desenvolvido com a linguagem de programação Python no editor de códigos *Visual Studio Code*. A máquina de desenvolvimento é um notebook Dell com processador Intel core i5 de sétima geração, 8GB RAM com sistema operacional Windows.

Para repositório e controle de versão de código, foi utilizado o GitHub e o trabalho estará disponível publicamente em https://github.com/Pendulun/PacManSearchAlgorithms após a data de limite de entrega do trabalho 17/12/2021.

2.2. Descrição do problema

Como dito anteriormente, o problema é caracterizado por um agente, PacMan, que procura alcançar uma posição objetiva com uma recompensa em um tabuleiro quadriculado. Além do agente e do objetivo, o tabuleiro contém barreiras/paredes que impedem a movimentação do agente entre posições e dificultam a procura do objetivo pelo agente.

No tabuleiro, o agente pode se mover para a posição imediatamente acima, abaixo, a direita ou esquerda executando as ações, respectivamente, *North*, *South*, *East* ou *West*. Além disso, a ação de movimento pode ter custo variado ou não, dependendo do mapa e situação escolhida de execução. O caminho em si da solução é uma sequência formada por essas ações que o agente deverá seguir partindo do seu ponto inicial até chegar no objetivo. Esse caminho deverá conter apenas ações válidas dado o estado momentâneo, ou seja, não poderá tentar passar por cima de uma barreira ou sair do mapa por exemplo. Naturalmente, uma sequência que chega corretamente no objetivo obedece essa regra.

Uma forma conveniente de modelar o problema dada as suas regras é via um grafo. Um grafo é composto por um conjunto de nós que são ligados entre si por um conjunto de arestas. Um aresta liga dois nós/vértices e pode ser direcionada ou não. Assim, pode-se visualizar uma posição no tabuleiro como sendo um vértice que está ligado a outros vértices que representam seus vizinhos alcançáveis. A aresta que liga dois vértices pode ter um peso associado representando o custo da ação de sair do vértice A para o vértice B.

2.3. Descrição geral dos algoritmos

Seguindo a necessidade de encontrar um caminho do agente para o objetivo, foram implementados 5 algoritmos de busca. Todos eles realizam o que é chamado de expansão de estados. Um estado é simplesmente uma posição no tabuleiro, representado por um nó/vértice em um grafo, e sua expansão é o ato de alcançar suas posições vizinhas que poderão ou não ser expandidas no futuro. Alcançar um estado pela primeira vez significa que ele foi descoberto.

A possibilidade de expansão se baseia em duas coisas: 1) O estado/posição já não foi expandido anteriormente e 2) O estado tem maior prioridade para ser expandido. Idealmente, um estado não deve ser expandido mais de uma vez para melhorar o custo computacional do algoritmo. Já a prioridade é representada por uma função $f(x)$, também conhecida como função de avaliação, que retorna um número que, aqui, quanto menor for, maior será a prioridade de expandir o seu estado.

Uma questão importante é a decisão de quando testar se um estado alcançado é o estado objetivo. Existem dois momentos intuitivos de checar essa condição: 1) Quando o estado é alcançado ou 2) Quando o estado é escolhido para expansão. O primeiro representa um *Early Goal Test*, já que, assim que alcançamos o estado objetivo, já o identificamos e retornamos uma solução. O segundo representa um *Late Goal Test* pois, apesar de já termos alcançado o estado objetivo, apenas retornamos uma solução quando ele for priorizado para expansão, fazendo com que o algoritmo leve mais iterações do que um que faz *Early Goal Test*. Entretanto, checar um estado pelo objetivo de forma tardia tem a vantagem de permitir com que um outro caminho melhor seja encontrado para o objetivo nesse meio tempo, pois outros nós/estados serão levados em consideração antes de retornar a solução.

Os algoritmos implementados no trabalho se diferenciam, basicamente, pela lógica de prioridade calculada para a expansão de um nó, influenciando, portanto, na ordem de descobrimento, número de posições levadas em consideração para fazerem parte do caminho e, possivelmente, no traçado do caminho até o objetivo. Assim, alguns algoritmos podem encontrar o caminho ótimo para a recompensa e outros não.

De forma geral, pode-se construir um grafo que representa o processo de busca do caminho para o objetivo fazendo com que um nó sendo expandido possua uma aresta direcionada para todos os nós descobertos na sua expansão. Esses nós são conhecidos como nós filhos e o nó expandido é conhecido como nó pai. Dependendo do algoritmo, é possível que seja alterado o pai de um nó em tempo de execução.

Ao anotar qual o nó pai e a ação tomada para se chegar ao nó atual, ao final dos algoritmos, basta realizar um caminhamento contrário partindo do nó objetivo até o nó inicial, que não possui nó pai, para conhecer todo o caminho da solução. É assim que todos os algoritmos implementados geram a sua solução.

2.4. Busca sem Informação

A primeira parte do trabalho se trata em desenvolver algoritmos de busca para encontrar a recompensa sem saber de antemão a sua posição. Essa situação caracteriza uma Busca sem Informação ou Busca Cega. Dessa forma, o agente deve ser capaz de explorar o tabuleiro seguindo alguma regra para encontrar o objetivo e traçar um caminho até ele sem levar em consideração a posição do objetivo, já que ela não é conhecida.

A seguir, serão descritos os algoritmos implementados que caem na classe de Busca sem Informação.

2.4.1. Busca em Profundidade (DFS)

Uma escolha da lógica de prioridade para expansão é escolher aquele nó que foi descoberto mais recentemente. Isso acontece ao expandir o nó descoberto mais recente entre todos os nós descobertos mas não expandidos. Em um grafo, isso consiste em priorizar o nó com maior profundidade no momento e que não tenha sido expandido. Dessa forma, a função de prioridade de expansão de um nó $f(x)$ será igual à profundidade do nó. Na implementação do algoritmo, usou-se uma Pilha (lista LIFO) que simula esse comportamento, já que o nó adicionado mais recentemente à pilha de nós a serem explorados será o primeiro a ser expandido.

O algoritmo implementado realiza *Late Goal Test* e o seu efeito é uma busca que simula uma minhoca que anda debaixo da terra e só volta uma casa quando é impedida de passar por algum motivo ou quando chega em uma posição pela qual já passou.

A Figura 1 exemplifica os estados expandidos para encontrar um caminho em uma fase de tamanho médio usando DFS. Para efeitos de futura comparação com os outros algoritmos, a Figura 2 mostra o DFS em uma fase de tamanho grande.

Esse algoritmo não é ótimo, ou seja, pode não encontrar o melhor caminho disponível, entretanto, pode encontrar um caminho mais rapidamente como será discutido na seção de comparação de algoritmos. Além disso, sendo *b* o *branching factor* e *m* a profundidade máxima, ele possui complexidade de tempo $O(b^m)$ e de espaço $O(bm)$.

2.4.2. Busca em Largura (BFS)

O segundo algoritmo define outra função $f(n)$ para servir de prioridade. Ao contrário do DFS, a maior prioridade de expansão será concedida ao nó disponível com menor

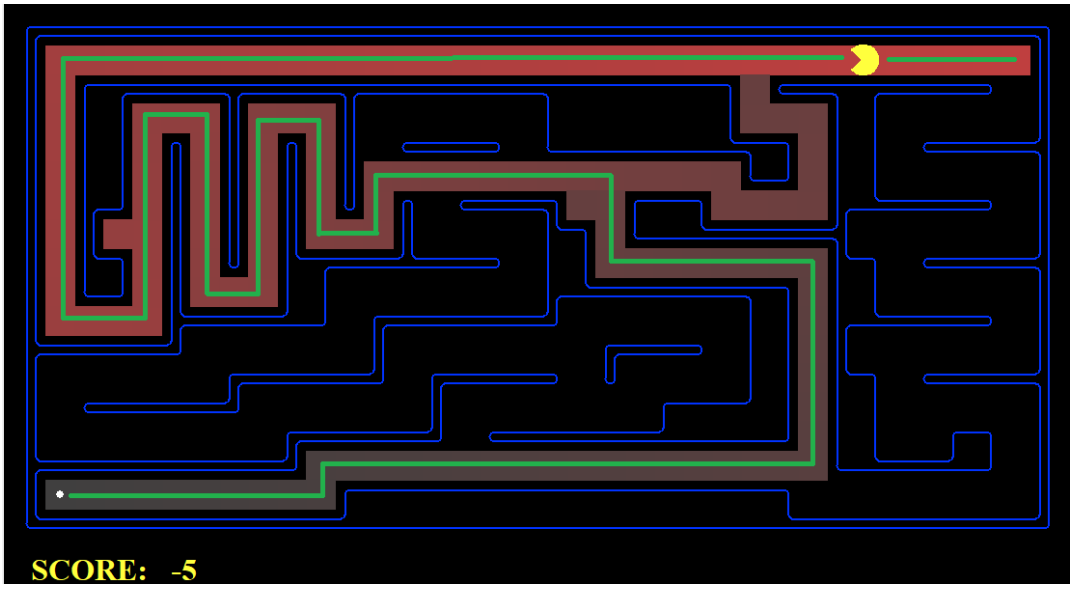

Figure 1. Execução do algoritmo DFS em um tabuleiro de tamanho médio. Em verde, o caminho que o PacMan percorre para chegar ao destino

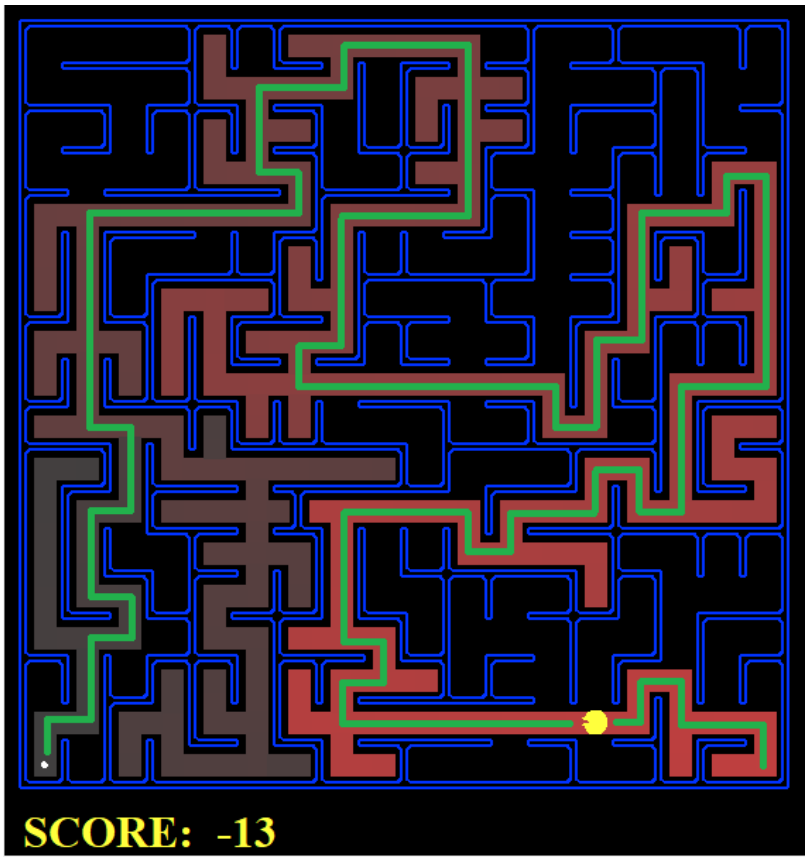

Figure 2. Execução do algoritmo DFS em um tabuleiro de tamanho grande e seus nós expandidos. Em verde, o caminho encontrado até o objetivo.

profundidade. Isso pode ser implementado utilizando uma Fila (lista FIFO), na qual o nó mais antigo a ter sido adicionado terá a maior prioridade para expansão.

O algoritmo também realiza *Late Goal Test* e o seu efeito é uma busca que parece uma onda do mar se espalhando por um terreno, expandindo sempre os nós logo a frente.

A Figura 3 mostra os nós expandidos para a mesma fase grande da Figura 2. Pode-se perceber que o BFS expande mais nós e, nesse caso, encontra o mesmo caminho. Na Figura 4 também é possível ver que esse algoritmo expandiu mais nós, entretanto, conseguiu encontrar um caminho melhor.

Esse algoritmo tem complexidade de tempo e espaço igual a $O(b^d)$, sendo b o branching factor e d a profundidade da solução. Dessa forma, possui maior complexidade de espaço do que o DFS e é isso que pesa mais nesse algoritmo.

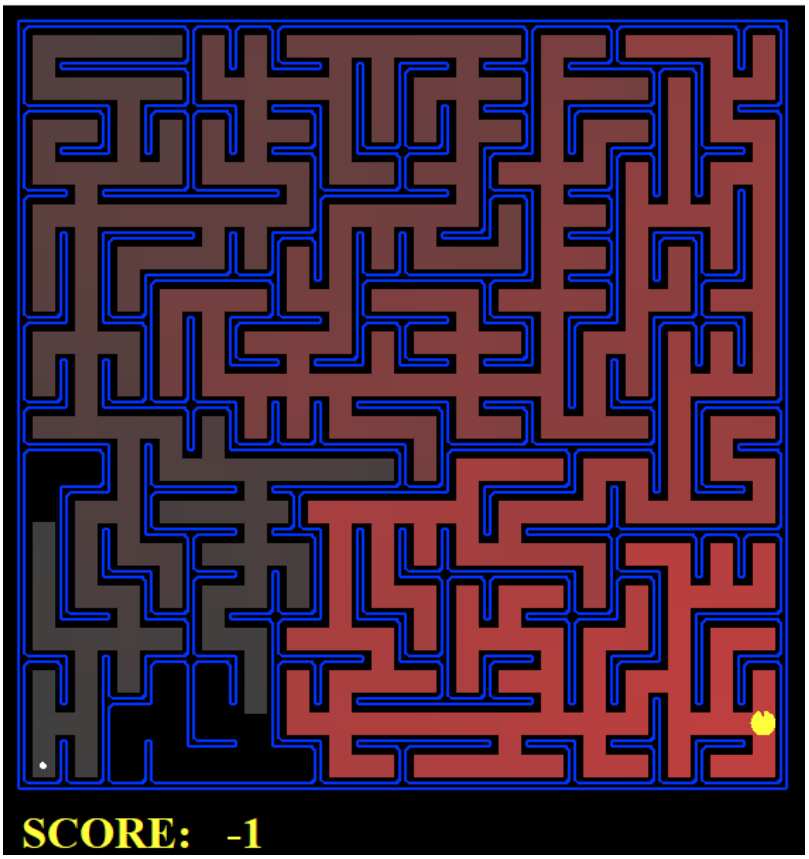

Figure 3. Execução do algoritmo BFS em um tabuleiro de tamanho grande e seus nós expandidos. O caminho encontrado foi idêntico ao da Figura 2

2.4.3. Busca com Custo Uniforme (UCS)

Esse algoritmo possui uma função de prioridade $f(n)$ diferente dos outros dois algoritmos. Ele dá prioridade para expandir o nó com menor custo acumulado para chegar nele. Dessa forma, esse algoritmo leva em consideração os custos das ações a serem realizadas por um estado. Isso pode ser implementado ao usar uma Fila de Prioridades.

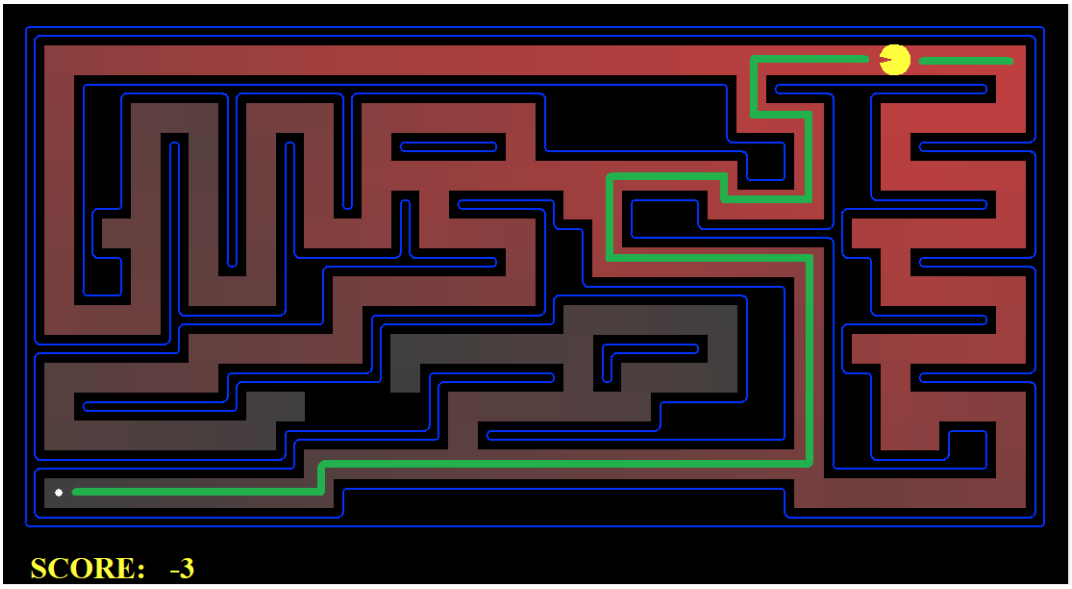

Figure 4. Execução do algoritmo BFS em um tabuleiro de tamanho médio e seus nós expandidos. O caminho encontrado foi menor do que da Figura 1

Além disso, o algoritmo também é capaz de atualizar o caminho até um nó e a sua prioridade na fila de prioridades de nós a serem expandidos. Funciona como o seguinte: Digamos que um nó B é descoberto por um nó A e colocado na fila de prioridades, lembrando que a sua prioridade é o custo para se chegar ao nó dado o caminho atual. Em algum momento, pode ser que outro nó C alcance esse nó B de novo. Se o nó B ainda não tiver sido expandido e o custo de encontrar B via C for menor do que o custo de encontrar B via A, atualizamos a prioridade de B para esse novo custo e também atualizamos o nó pai de B para C. Isso permite ao algoritmo encontrar caminhos próximos do ótimo, senão ótimos.

Essa é uma boa abordagem em situações na qual o movimento do agente possui custo variado dependendo da posição de origem ou de destino. Além disso, mostra que maior quantidade de ações não necessariamente resulta em maior custo para se chegar ao alvo nesses casos.

A Figura 5 mostra a execução desse algoritmo em uma fase de tamanho médio. Pode-se perceber que tanto os nós expandidos quanto o caminho encontrado foram iguais ao do BFS. Isso se deve pois, nessa fase, todo movimento tem custo igual, demonstrando uma propriedade de que esse algoritmo performa igual ao BFS quando o custo das ações não varia.

2.5. Busca com Informação

A segunda parte do trabalho se trata em desenvolver algoritmos de busca que levam em consideração a posição do objetivo no tabuleiro, caracterizando uma Busca com Informação. Ao ter o conhecimento da posição do objetivo, o algoritmo pode construir uma lógica de prioridade que facilite a busca do caminho até o objetivo agindo como uma indicação do caminho a ser seguido para o agente. Essa lógica específica é conhecida como heurística. Dessa forma, a função $f(n)$ leva em consideração a heurística $h(n)$. Algumas heurísticas conhecidas são a Distância Euclidiana e a Distância Manhattan.

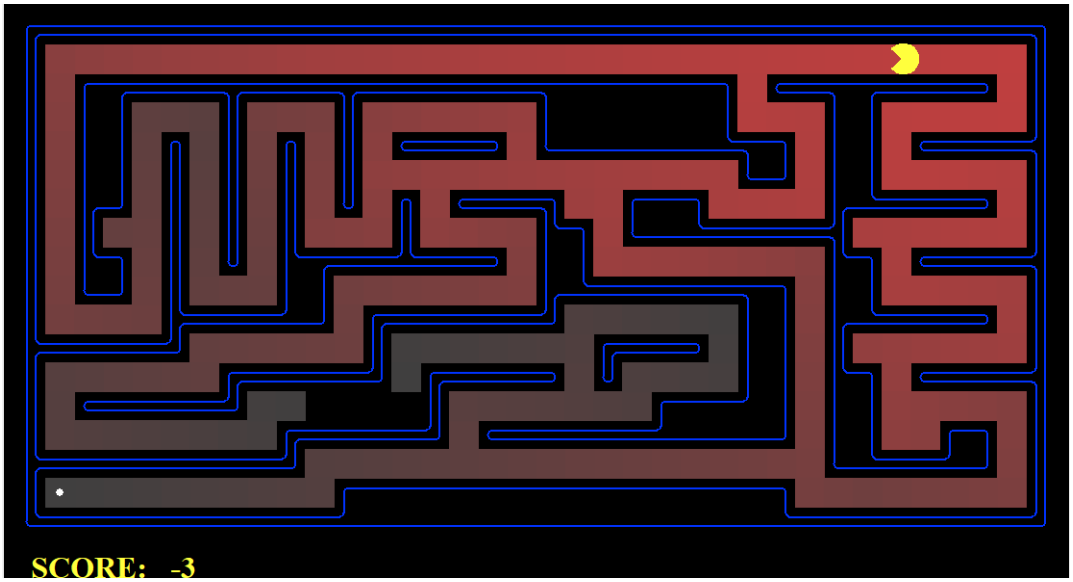

Figure 5. Execução do algoritmo UCS em um tabuleiro de tamanho médio e seus nós expandidos. O caminho encontrado foi igual ao da Figura 4 e foi omitido.

A heurística deve satisfazer duas propriedades: 1) Deve ser admissível e 2) Deve ser consistente. A primeira propriedade significa que a heurística deve ser otimista com relação ao custo de se atingir o objetivo. Dessa forma, ela deve ser menor ou igual ao custo real. A segunda propriedade significa que, para um nó pai *n*, a heurística $h(n)$ para se atingir um objetivo *G* não pode ser maior do que o custo *C* de ir para um nó filho *n'* somado à heurística de sair de *n'* e chegar ao mesmo objetivo G, $h(n) \leq C + h(n')$. Uma heurística ser consistente implica em ela ser admissível mas não o contrário.

Nessa seção, são descritos os dois algoritmos implementados que fazem parte da classe de Busca com Informação.

2.5.1. Busca Gulosa (GS)

O primeiro algoritmo de Busca com Informação é o algoritmo de Busca Gulosa. Assim como os outros três algoritmos anteriores, a sua principal característica é a forma na qual é escolhido um nó para expansão. A sua função de prioridade é simplesmente igual à heurística adotada para o problema, $f(n) = h(n)$. Dessa forma, os nós com menor custo estimado/indicado pela heurística até o objetivo serão priorizados. Isso também pode ser implementado via uma Fila de Prioridades.

Além da heurística, esse algoritmo também realiza a atualização da prioridade de um nó de acordo com o menor custo encontrado até ele semelhante ao descrito na seção do UCS.

O efeito desse algoritmo junto com a heurística da Distância Manhattan também se assemelha a de uma minhoca rastejando para frente que procura o contorno mais rápido para passar por barreiras. A Figura 6 exemplifica esse comportamento.

A Figura 7 mostra a execução desse algoritmo em uma fase de tamanho médio. Pode-se perceber que ela expandiu poucos nós, ou seja, foi rápido, e encontrou um ca-

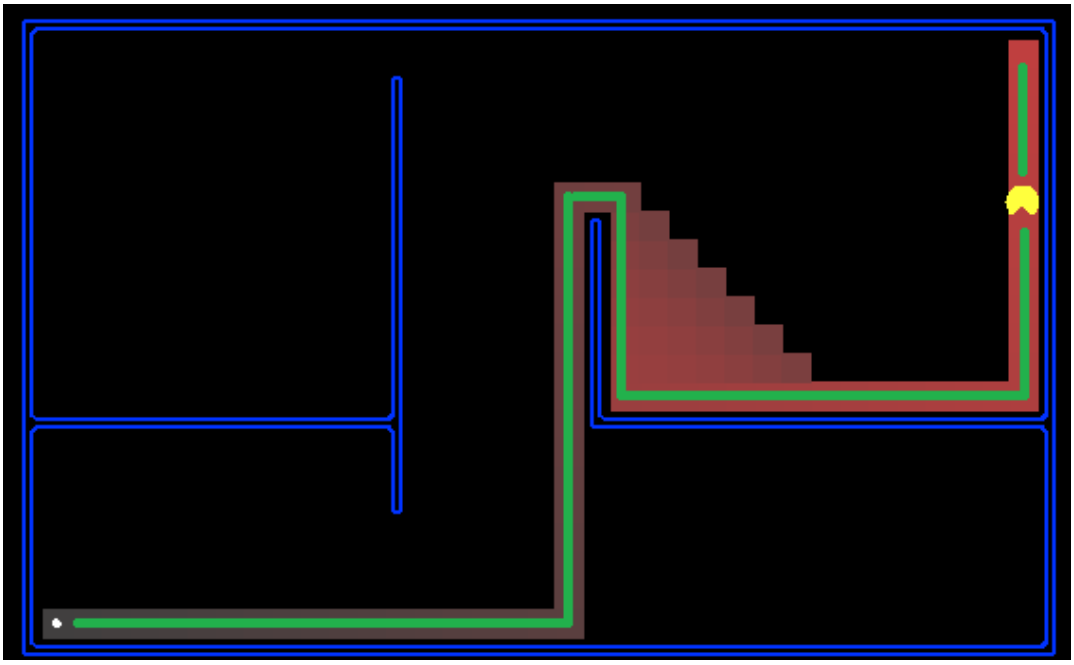

Figure 6. Execução do algoritmo GS com heurística de Distância Manhattan em um tabuleiro de tamanho médio aberto e seus nós expandidos.

minho de tamanho intermediário entre o BFS e o do DFS. A Figura 8 mostra a execução na fase de tabuleiro grande e pode-se perceber que ele expandiu menos nós do que o BFS/UCS mas mais nós do que o DFS.

2.5.2. Busca A* (A estrela)

O algoritmo de Busca A*, leva em consideração, além da heurística, o custo do caminho até o nó atual, dado por $g(n)$, assim como o UCS. Portanto, sua função de avaliação é $f(n) = g(n) + h(n)$. Para tal, também pode-se utilizar uma Fila de Prioridades. Além disso, ele também é capaz de atualizar a prioridade de um nó de acordo com seu custo do caminho sem computar a heurística.

O efeito de usar tanto o custo do caminho até o nó atual quanto a heurística é um meio termo entre o UCS e o GS. A Figura 9 mostra a execução desse algoritmo em uma fase de tamanho médio. Pode-se perceber que ele expande bastantes nós, sendo pior do que o DFS, mas ainda sim é melhor do que BFS e UCS. Além disso, ele foi capaz de encontrar o caminho ótimo, coisa que o DFS não conseguiu. A Figura 10 mostra o A* em um tabuleiro grande. Ele expandiu mais do que GS e DFS, mas menos do que BFS e UCS.

O A* é ótimo, completo e eficiente. Apesar de a complexidade de tempo e espaço serem exponenciais, pode-se colocar um peso positivo maior W na heurística para o cálculo da função de avaliação, $f(n) = g(n) + W \times h(n)$. Isso faz com que o algoritmo seja ainda mais direcionado para o objetivo, expandindo menos nós, mas pode não encontrar o caminho ótimo.

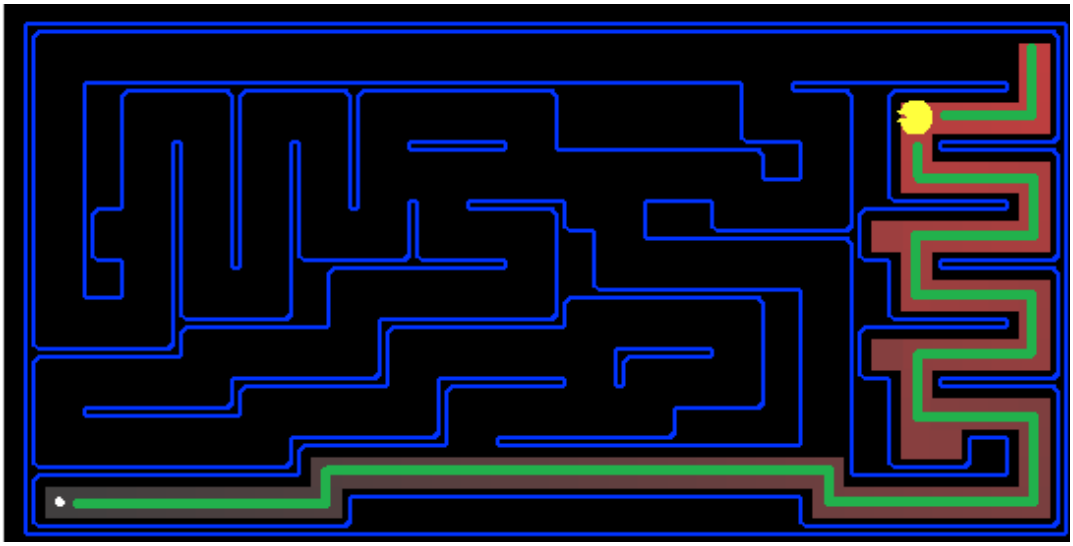

Figure 7. Execução do algoritmo GS com heurística de Distância Manhattan em um tabuleiro de tamanho médio e seus nós expandidos. O caminho encontrado foi diferente de todos os outros algoritmos e teve bom custo/benefício.

2.5.3. Nova Heurística Proposta

Como parte final do trabalho, foi necessário propor uma nova heurística para ser utilizada pelos algoritmos de Busca com Informação em situações onde existe mais de um objetivo no mapa, ou seja, mais de uma comida a ser alcançada pelo PacMan.

Desse modo a heurística proposta foi a **média das Distâncias Manhattan dos objetivos restantes ao agente**,

$$h(n) = \frac{sum_{Manhattan(a,Goal_i)}}{len(Goals)}$$

sendo *Goals* uma lista com as posições dos objetivos.

Essa heurística é admissível pois o seu resultado é menor do que a distância total necessária a se percorrer para pegar todos os objetivos, $media(n) < \sum n_i$. Além disso, ela é consistente pois o custo para expandir um nó compensa a possível diminuição que a média das distâncias terá ao expandir o nó. Isso acontece pois um nó expandido ou chega mais perto de todos os objetivos, diminuindo a média, ou ele chega perto de parte dos objetivos e se afasta dos outros, o que pode manter ou até mesmo aumentar a média das distancias.

2.6. Comparação entre os Algoritmos

Essa seção mostra e analisa os resultados obtidos com a execução de todos os algoritmos acima em diversas fases. As métricas levadas em consideração são: 1) Número de nós expandidos 2) Custo do caminho solução e 3) O custo benefício do algoritmo calculado por: C = *custo do caminho*, Q = *quantidade de nós expandidos* levando a *Custo Benefício* $= C/Q$.

As fases de execução e uma breve descrição delas:

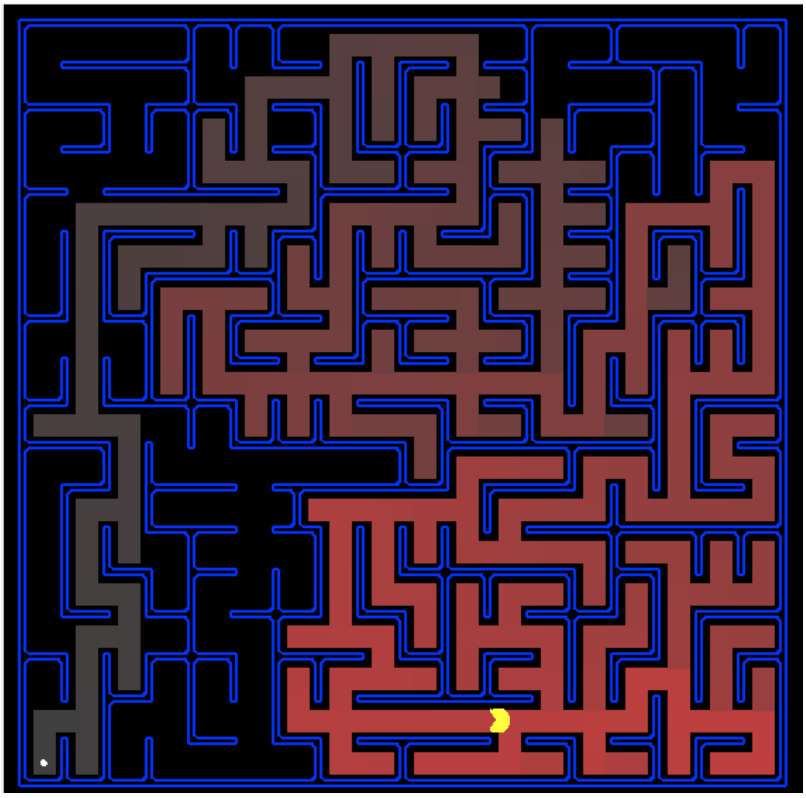

Figure 8. Execução do algoritmo GS com heurística de Distância Manhattan em um tabuleiro de tamanho grande e seus nós expandidos. O caminho encontrado foi igual ao de todos os outros algoritmos para essa fase.

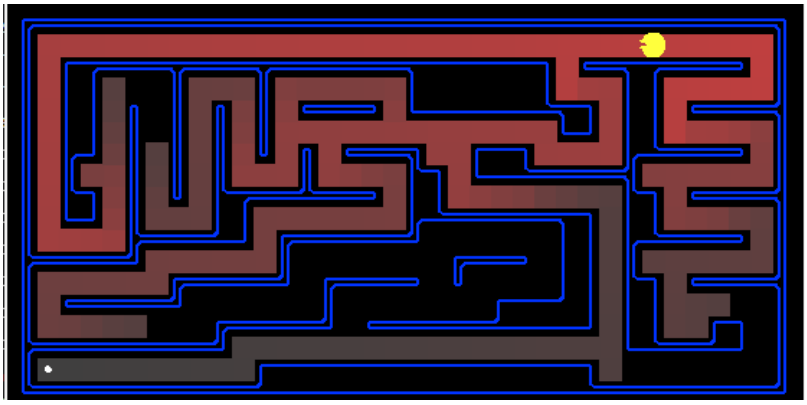

Figure 9. Execução do algoritmo A* com heurística de Distância Manhattan em um tabuleiro de tamanho médio e seus nós expandidos. O caminho encontrado foi igual ao do BFS e UCS.

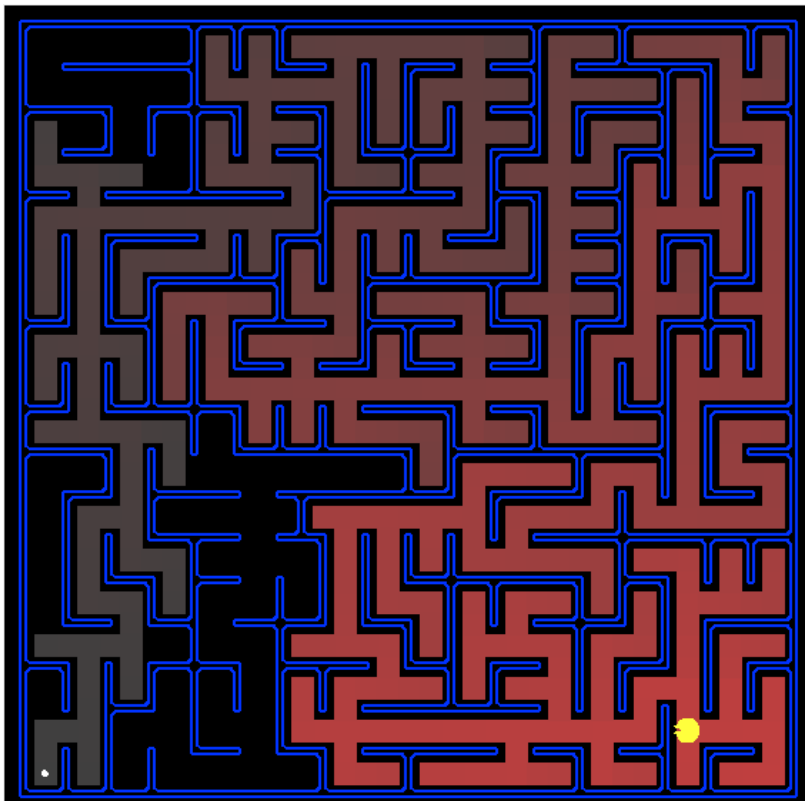

Figure 10. Execução do algoritmo A* com heurística de Distância Manhattan em um tabuleiro de tamanho grande e seus nós expandidos. O caminho encontrado também foi o ótimo.

- Fase mediumMaze: Um labirinto já mostrado em figuras passadas de tamanho médio. Possui bastantes barreiras entre a posição inicial e o objetivo e existem vários caminhos possíveis até o objetivo.
- Fase bigMaze: Um labirinto que também já foi mostrado em figuras anteriores de tamanho grande. Possui muitas barreiras entre a posição inicial e o objetivo, mas apenas um caminho de fato para se chegar ao objetivo.
- Fase openMaze: Uma fase mostrada na Figura 6. Possui poucas barreiras entre o estado inicial e o objetivo com grande área aberta para movimentação. Devido à sua área aberta, vários caminhos são possíveis, mas poucos com custo ótimo.
- Fase trickySearch: Uma fase que não foi mostrada em nenhuma figura. Utilizada para testar algoritmos de Busca com Informação, ou seja, aceitam heurística. Possui poucas barreiras ao longo do mapa mas muitos objetivos a serem alcançados. Efetivamente, foi nessa fase que a nova heurística proposta foi testada.

Dessa forma, foram executados todos os algoritmos propostos nas fases possíveis de serem executadas descritas acima. Para as fases *openMaze*, *mediumMaze* e *bigMaze*, os algoritmos de Busca com Informação usaram a heurística Distância de Manhattan. Já para o *trickySearch*, usaram a nova heurística proposta.

A Figura 11 mostra uma tabela com o número de nós expandidos por algoritmo em cada fase. Em azul, estão os menores números para cada fase e, em laranja, estão os maiores números. Pode-se perceber que algoritmos que tem como característica uma

Fase	Quantidade de nós expandidos por algoritmo				
	DFS	BFS	UCS	GS	ASTAR
mediumMaze	146	269	269	**78**	221
bigMaze	**390**	620	620	466	549
openMaze	806	682	682	**89**	535
trickySearch	-	-	-	**8870**	11254

Figure 11. Tabela de número de nós expandidos para cada algoritmo implementado em cada fase de teste.

Fase	Custo da solução por algoritmo				
	DFS	BFS	UCS	GS	ASTAR
mediumMaze	130	**68**	**68**	74	**68**
bigMaze	**210**	**210**	**210**	**210**	**210**
openMaze	298	**54**	**54**	68	**54**
trickySearch	-	-	-	187	**60**

Figure 12. Tabela de custo do caminho solução para cada algoritmo implementado em cada fase de teste.

busca mais profunda, como o GS e DFS, expandem menos nós em ambientes cheios de barreiras, como os labirintos, e também no ambiente mais aberto.

A Figura 12 mostra o custo total dos caminhos de solução dados por cada algoritmo em cada fase. Em azul, aparecem os menores custos por fase e, em laranja, os maiores. Comparando as duas tabelas, percebe-se que, apesar de expandirem menos nós, na maioria dos casos, os algoritmos com característica de busca profunda não encontraram os melhores caminhos, isto é, caminhos com custo mínimo. A única vez que encontraram tal solução ótima foi quando ela era a única que existia. Isso ocorreu na fase *bigMaze*. Uma outra visualização para comparação pode ser vista na Figura 13.

Essa observação evidencia uma propriedade dos algoritmos com certa busca profunda: são mais rápidos para encontrar um caminho e tem menor custo de memória devido a menor expansão de nós. Dessa forma, podem ser mais aconselhados quando o espaço de estados é muito grande e, consequentemente, o tempo de pesquisa de um caminho também.

Por outro lado, caso seja realmente necessário encontrar o melhor caminho em questão de custo, os outros algoritmos devem ser utilizados, escolhendo aquele com melhor custo benefício de expansão e tempo de busca. Para tal, foi construída uma tabela com base no custo benefício descrito anteriormente. Essa tabela pode ser vista na Figura 14 e tem uma versão em gráfico na Figura 15. Ambas demonstram que o GS foi melhor no geral, entretanto, isso provavelmente mudaria caso o custo das ações fosse

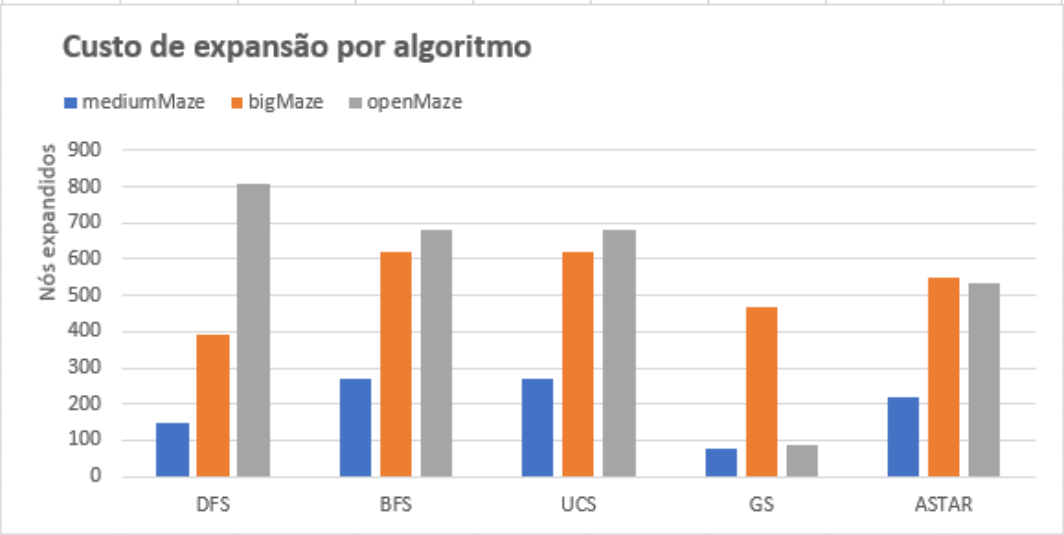

Figure 13. Gráfico do custo do caminho solução para cada algoritmo implementado em cada fase de teste.

Fase	Custo da solução/qt. nós expandidos por algoritmo				
	DFS	BFS	UCS	GS	ASTAR
mediumMaze	0.890411	0.252788	0.252788	**0.948718**	0.307692
bigMaze	**0.538462**	0.33871	0.33871	0.450644	0.382514
openMaze	0.369727	0.079179	0.079179	**0.764045**	0.100935
trickySearch	-	-	-	**0.021082**	0.005331

Figure 14. Tabela de custo benefício para cada algoritmo implementado em cada fase de teste.

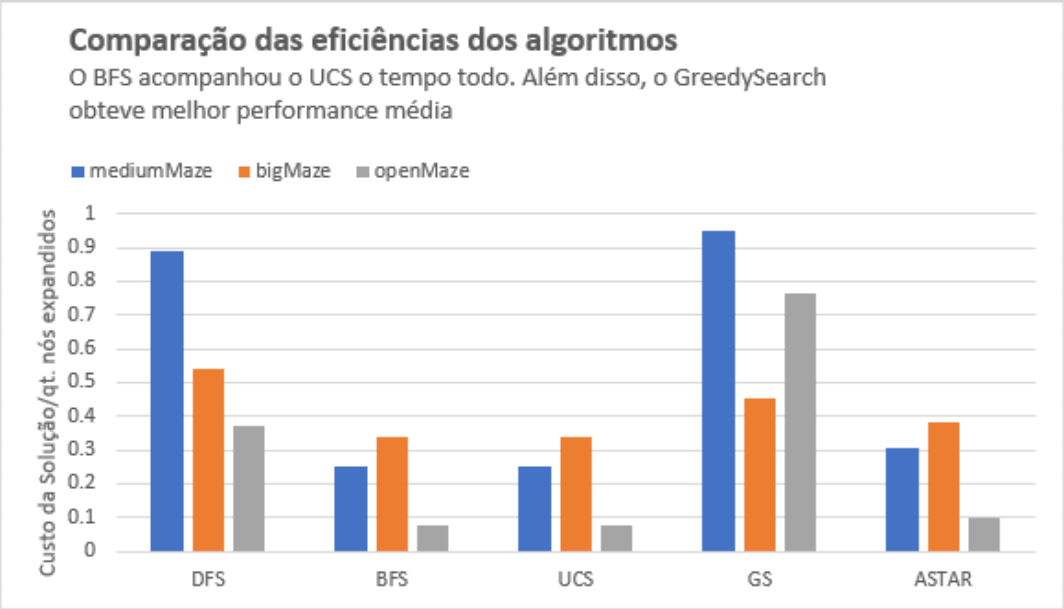

Figure 15. Gráfico de custo benefício para cada algoritmo implementado em cada fase de teste.

variável.

Para a heurística proposta, pode-se perceber que ela produziu um custo benefício muito baixo tanto com o GS quanto com o A*. Isso fica evidente pelo número de nós necessários a serem expandidos na casa de dez mil. Ao longo dos testes, foi notado que esse número varia de acordo com a quantidade de objetivos que se deseja alcançar no mapa. Quanto mais objetivos, mais nós precisam ser expandidos.

3. Conclusão

O objetivo do trabalho era desenvolver vários algoritmos de busca de estado dentro de ambientes tabulares e com barreiras com o agente PacMan. O problema geral possui diversas aplicações interessantes, como, por exemplo, a navegabilidade de um carro autônomo por uma cidade. Os algoritmos se diferenciam, principalmente, pela lógica de prioridade na hora de expansão de seus nós. De acordo com a prioridade, caminhos diferentes podem ser criados e nem sempre são ótimos.

De acordo com os testes realizados, algoritmos de Busca sem Informação, como UCS e BFS, podem ser mais indicados quando o espaço de busca é menor. Se não, o DFS pode ser utilizado com a negativa de que o melhor caminho possivelmente não será encontrado. Caso se faça necessário encontrar, com certeza, o melhor caminho entre o agente e o objetivo, entre esses três, o BFS é indicado desde que o custo das ações seja constante.

Se o custo das ações for variável, pode-se usar outro algoritmo de Busca sem Informação como o UCS ou o algoritmo de Busca com Informação A*. Esse último combina ao custo das ações uma heurística que serve como indicação para o agente de qual caminho seguir. Caso as ações tenham custo constante e seja possível levar em consideração a posição do objetivo, o GS pode ser bem aproveitado também.

A proposta de uma heurística de acordo com o entendimento do problema pode fazer com que a busca de caminhos seja mais eficiente dentro do ambiente. Entretanto, isso não significa que a formulação de uma boa nova heurística consistente seja fácil. Além do entendimento do problema, experiência e criatividade são fatores que contribuem para o desenvolvimento de uma boa heurística.

Por fim, com este trabalho, foi possível por em prática algoritmos e aprendizados das aulas que contribuirão em outras áreas de estudo no geral.

4. Bibliografia

References

Patel, A. Heuristics. On http://theory.stanford.edu/ amitp/GameProgramming/Heuristics.html. Last access on 13/12/2021.

Stuart Russel, P. N. (2020). *Artificial Intelligence A Modern Approach*. Pearson, 4th edition.