INSTITUTO FEDERAL DO ESPÍRITO SANTO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA – PPCOMP

CARLOS EDUARDO GOMES REDDO ALVES

ALGORITMOS DE BUSCAS APLICADOS EM LABIRINTOS E MAPAS E ALGORITMOS DE OTIMIZAÇÃO

Serra
2023

CARLOS EDUARDO GOMES REDDO ALVES

ALGORITMOS DE BUSCAS APLICADOS EM LABIRINTOS E MAPAS E ALGORITMOS DE OTIMIZAÇÃO

Relatório Técnico do Programa de Pós-Graduação em Computação Aplicada – PPCOMP do Instituto Federal do Espírito Santo, projeto da Disciplina de Inteligência Artificial.

Serra
2023

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Um labirinto é uma estrutura intricada e complexa composta por caminhos entrelaçados e interligados, projetada para desafiar e confundir aqueles que tentam navegar por ela. Geralmente, é construído com paredes, corredores estreitos e interseções múltiplas, criando uma série de opções e direções possíveis. O objetivo é criar um desafio mental e físico, exigindo raciocínio estratégico, orientação espacial e perseverança para encontrar o caminho correto para o centro ou para a saída. Labirintos podem ser encontrados em diferentes contextos, como em jardins, parques temáticos ou em jogos virtuais, proporcionando uma experiência intrigante e divertida para aqueles que se aventuram em seu interior.

Há várias técnicas para um computador resolver o caminho de um labirinto, dentre elas estão os algoritmos de busca, que buscam achar o melhor caminho para um objetivo, sendo a saída no caso do labirinto. Para evitar a exploração de todos os caminhos possíveis, foram desenvolvidos alguns algoritmos que buscam mover-se o mínimo possível de forma a ganhar mais performance. Alguns desses algoritmos de busca são: Busca em Profundidade (*DFS*), *Dijkstra*, Caminho uniforme e A Estrela.

Um labirinto pode ser visto como um tipo específico de grafo, onde os corredores e interseções representam os vértices, e as paredes entre eles são as arestas. Nesse contexto, encontrar uma rota em um labirinto se traduz em descobrir o caminho correto para alcançar a saída. Os algoritmos utilizados para resolver o problema de encontrar uma rota em um grafo podem ser aplicados para solucionar o desafio de navegar por um labirinto.

Outro problema também resolvido com algoritmos de buscas é encontrar uma rota em um mapa, o interpretando como um grafo e determinando um caminho ou trajeto entre dois vértices específicos em uma estrutura. Esse desafio pode se tornar complexo dependendo da natureza do grafo, sua conectividade e as restrições impostas. O problema geralmente envolve a busca por um caminho que satisfaça certas condições, como a menor distância, o menor número de arestas percorridas ou a otimização de algum critério específico. A complexidade aumenta ainda mais quando o grafo é direcionado, ponderado ou contém ciclos.

Um ponto bônus trabalhado é o problema das 8 rainhas. Um desafio clássico de posicionamento de peças de xadrez em um tabuleiro de 8x8, no qual o objetivo é colocar oito rainhas no tabuleiro de forma que nenhuma delas possa atacar outra. Isso significa que nenhuma rainha deve compartilhar a mesma linha, coluna ou diagonal com outra rainha. O problema envolve encontrar todas as configurações possíveis que satisfaçam essa restrição, ou seja, todas as maneiras de posicionar as oito peças no tabuleiro sem que elas se ataquem mutuamente. O desafio é encontrar uma solução válida ou determinar que não há solução possível. Esse problema é um exemplo clássico de um problema de

colocação de peças e tem aplicações em áreas como a ciência da computação, teoria dos jogos e algoritmos de busca e otimização.

1.1 Algoritmos

Para a exploração dos problemas serão utilizamos alguns algoritmos como: *BFS, DFS, UCS* e *A**. Na tabela 1 é possível visualizar a diferença de complexidade entre os algoritmos.

Tabela 1 – Comparação de complexidade dos algoritmos

Algoritmo	Custo
BFS	$O(V + E)$
DFS	$O(V + E)$
UCS	$O((V + E)logV)$
A*	$O(b^d)$ a $O(V^2)$

Fonte: Desenvolvido pelo autor

1.1.1 Breadth First Search

O algoritmo *BFS* explora todos os vértices em um nível antes de prosseguir para o próximo nível. Isso garante que o caminho encontrado seja o mais curto em termos de número de arestas. A complexidade de tempo do algoritmo é $O(V + E)$, onde V representa o número de vértices e E o número de arestas do grafo. Ele visita todos os vértices e todas as arestas uma vez.

1.1.2 Depth First Search

O algoritmo *DFS* explora um ramo do grafo o máximo possível antes de retroceder. Ele pode encontrar rapidamente um caminho, mas não necessariamente o mais curto. A complexidade de tempo do é $O(V + E)$, assim como o *BFS*. No entanto, a ordem em que os vértices são visitados pode afetar o desempenho do algoritmo em termos de encontrar o caminho desejado.

1.1.3 Uniform Cost

O algoritmo Custo Uniforme, *UCS*, atribui custos às arestas e seleciona o caminho com o menor custo acumulado até o momento. Ele explora todas as possibilidades, mas de forma ordenada em termos de custo.

A complexidade de tempo do *UCS* depende do número de vértices e arestas, bem como dos custos associados a cada aresta. Em geral, a complexidade é maior do que o *BFS* e o *DFS*, podendo chegar a $O((V + E)logV)$ em casos em que a busca de menor custo é necessária.

1.1.4 A Star

O algoritmo A* também atribui custos às arestas, mas inclui uma heurística que estima o custo restante para o destino. Ele combina a busca em largura com a heurística, o que ajuda a direcionar a busca para áreas promissoras do grafo.

Sua complexidade de tempo depende da qualidade da heurística escolhida. Em geral, a complexidade pode variar de $O(b^d)$ a $O(V^2)$, onde b é o fator de ramificação médio, d é a profundidade da solução e V é o número de vértices.

1.2 Execução

Todos os algoritmos executados neste relatório foram executados em um computador com a seguintes configurações:

- Processador: Intel i7 13700K@5.4GHz
- RAM: 32GB DDR4@3200MT/s
- SO: Windows 11 22H2
- Python: 3.10.9

2 Labirinto

2.0.1 Definição do problema

Um labirinto aleatório de tamanho 300x300, tendo 50% do seu espaço bloqueado por paredes, deve-se determinar um ponto de início e fim e um algoritmo de busca deverá percorrer o caminho até encontrar a saída. Para cada algoritmo serão gravados os seguintes dados: tempo de execução (em segundos), espaços visitados e caminho mínimo até a solução.

Será feita uma execução de todos os algoritmos, a fim de comparação, utilizando a *seed* 42, de forma que todos os labirintos serão iguais, alterando apenas o método que será percorrido.

A tabela 2 ilustra a ordem dos vizinhos a ser gerada para um determinado nó, onde o nó atual é demarcado por n enquanto os seus vizinhos ao redor são numerados em ordem crescente, seguindo a ordem a ser retornada pelo algoritmo.

Tabela 2 – Ordem dos vizinhos dado um nó N

7	6	5
8	N	4
2	1	3

Fonte: Desenvolvido pelo autor

2.0.2 Algoritmos aplicados

Para encontra a saída do labirinto foram escolhidos 5 algoritmos: *breadth-first, depth-first, uniform cost, A Star* e *iterative deepening depth-first*. A execução de cada um seguirá as regras estabelecidas previamente.

2.0.3 Resultados

Tabela 3 – Resultado dos algoritmos no problema do labirinto

Método	Passos	Expandidos	Iterações	Tempo (s)	Custo
Breadth First	360	54582	54583	1.9555053711	471
Depth First	652	857	858	0.0744957924	724
Uniform Cost	370	44707	44710	3.7965686321	473
A Star	361	42820	42823	3.6994338036	468
Interactive Deepening Depth First	360	54579	54582	4.1128177643	469
A Star Non Cumulative	486	550	552	0.052646637	555

Fonte: Desenvolvido pelo autor

A tabela 3 exibe o resultado da execução dos algoritmos, exibindo o número de passos do caminho encontrado, o número de nós expandidos, o custo total do caminho e o tempo de processamento, em segundos.

Após a execução dos algoritmos, é visível que *breadth-first* e *interactive deepening depth first* conseguiram encontrar o menor caminho no labirinto, sendo necessário apenas 360 movimentações para se chegar ao destino final, a saída. Apesar do ótimo desempenho do caminho escolhido, foram necessários analisar 54582 e 54579 blocos, respectivamente, levando 1.9 segundo de execução para o primeiro algoritmo e 4.11 segundos para o segundo.

Já o método *depth-first* mostrou possuir um bom desempenho, tanto na quantidade de blocos explorados quanto no tempo de execução: 857 blocos, necessitando de apenas 76 milissegundos. Apesar disso, o caminho escolhido foi um pouco maior que os demais, utilizando de 652 passos para se chegar a saída.

Os algoritmos *uniform cost* e *A Star* sofreram com um grande problema: a distância de um nó para todos os seus vizinhos sempre possuía o mesmo valor. Como essas técnicas se baseiam na distância acumulada percorrida, $g(n)$, seu funcionamento tornou-se semelhante ao *breadth-first* pois todos os nós anteriores sempre possuem a mesma distância. A distância entre os nós foi calculada da seguinte forma: nós adjacentes possuem uma distância de 1 e nós a diagonal valor 2.

Apesar do cálculo do *A Star* também incluir uma heurística $h(n)$, definida pela distância euclidiana do nó até a saída do labirinto, o mesmo problema citado acima acontecia, fazendo-o explorar praticamente todo o labirinto até encontrar a solução. O *uniform cost* necessitou percorrer 44707 nós para encontrar um caminho para a saída composto por 370 nós em 3.79 segundos. Já o *A Star* necessitou percorrer 42820 nós, encontrando um caminho de 361 nós em 3.49 segundos.

Como forma de contornar esse problema, foi feita uma alteração exploratória no algoritmo *A Start*, de forma que a sua função $g(n)$ passe a ser a distância atual para nó anterior, ao invés do valor acumulado. A função $h(n)$ permaneceu inalterada. Com essa pequena alteração, o algoritmo precisou percorrer apenas 550 nós para encontrar um caminho composto por 486 nós em apenas 52 milissegundos. Apesar de não ser o melhor caminho, o número de nós percorridos é muito inferior a versão cumulativa, fazendo ser executado cerca de 70 vezes mais rápido.

3 Mapa

3.0.1 Definição do problema

O problema de encontrar um caminho em um mapa utilizando grafos é um desafio clássico da teoria dos grafos e da ciência da computação. Ele envolve determinar um caminho ótimo ou viável entre dois pontos específicos em um mapa representado como um grafo.

Figura 1 – Parte do mapa da Romênia transformado em grafo

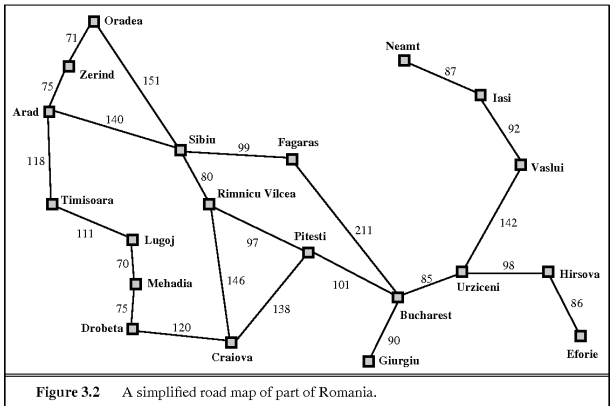

Figure 3.2 A simplified road map of part of Romania.

Fonte: Russell, Norvig e Davis (2010)

Na figura 1, o mapa é modelado como um grafo, onde os locais ou pontos de interesse são representados pelos vértices do grafo e as conexões entre esses locais são representadas pelas arestas. Cada aresta pode ter um peso associado, que representa a distância, o tempo de percurso ou algum outro critério relevante para determinar a qualidade do caminho.

Neste problema, será utilizados alguns algoritmos de buscas para se determinar um caminho entre um ponto de início e um ponto de destino.

Figura 2 – Valores de $h(n)$ para as cidades do mapa

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

Fonte: Russell, Norvig e Davis (2010)

3.0.2 Algoritmos aplicados

Como algoritmos para realizar a busca do caminho, serão utilizados o *uniform cost* e *A Star*.

3.0.3 Resultados

Tabela 4 – Resultado dos algoritmos executados

Algoritmo	Cidades visitadas	Distância	Tempo (ms)
Uniform Cost	13	418	0.49
A Star	12	450	0.49

Fonte: Desenvolvido pelo autor

É possível observar na tabela 4 que o tempo de execução é o mesmo para todos os algoritmos, devido ao mapa não possuir muitas cidades.

Também é possível visualizar que o $A*$ precisou visitar uma cidade a menos que o *Uniform Cost*, que conseguiu realizar o caminho com uma distância de 418 contra 450 do método anterior. Isso se deve ao fato de o A Estrela precisar utilizar uma heurística para calcular o custo entre as cidades, de forma que um caminho considerado curto para o custo uniforme pode não ter o mesmo valor para o A Estrela. Esses valores podem ser observados na figura 2.

4 8 Rainhas

4.0.1 Definição do problema

O desafio das 8 rainhas é um quebra-cabeça clássico de tabuleiro de xadrez, onde o objetivo é posicionar oito rainhas em um tabuleiro de xadrez de 8x8 sem que elas se ataquem mutuamente. Isso significa que nenhuma rainha pode compartilhar a mesma linha, coluna ou diagonal com outra rainha. Como cada rainha pode se mover horizontalmente, verticalmente e diagonalmente, o desafio consiste em encontrar uma configuração na qual todas as oito rainhas possam coexistir pacificamente no tabuleiro.

O desafio é um problema conhecido de programação e é frequentemente usado para demonstrar habilidades de resolução de problemas, algoritmos de busca e técnicas de otimização. Existem várias abordagens para resolver esse problema, incluindo algoritmos de força bruta, algoritmos baseados em busca em profundidade ou em largura.

Como forma de aumentar a complexidade de execução do algoritmo, será utilizado um tabuleiro 128x128 contendo 128 rainhas.

4.0.2 Algoritmos aplicados

Para a solução do desafio foi utilizado o algoritmo *Hill Climbing* e mais três variações: *Stochastic, First Choice* e *Random Restart*. A posição inicial das rainhas se da de forma aleatória, interagindo com apenas uma única rainha por vez e realizando uma iteração de troca de posição até que nenhuma rainha esteja atacando outra.

Uma iteração consiste na interação do algoritmo com todas as rainhas, não necessariamente precisando move-las (caso a melhor solução seja a posição atual). A ordem de interação das rainhas é aleatória.

O algoritmo *Random Restart* está definido para conseguir realizar um máximo de 10 iterações e 10 reinicializações.

4.0.3 Resultados

Tabela 5 – Resultado da execução dos algoritmos

Algoritmo	Iterações	Tempo (s)
Classic	11	4.6976726055
Stochastic	166	26.8597397804
First Choice	6	3.6026864052
Random Restart	7	9.7111568451

Fonte: Desenvolvido pelo autor

A tabela 5 ilustra os resultados obtidos de cada algoritmo. É observável que o método *Stochastic* apresentou o pior resultado, tanto por realizar 166 iterações quanto pelo tempo

de execução de 26 segundos.

Os algoritmos *First Choice* e *Random Restart* conseguiram um resultado muito próximo: eles foram capaz de resolverem o problema em apenas 6 e 7 iterações, respectivamente. Apesar disso, o *Random Restart* precisou do tripo do tempo comparado ao *First Choice*.

Já a execução do algoritmo em sua forma clássica precisou de 4 segundos para resolver o problema com 11 iterações.

5 Conclusão

Observando os resultados gerados pelos problemas descritos, é possível ver que os algoritmos de busca *Uniform Cost* e *A Star* conseguem encontrar um bom caminho até o objetivo quando os nós não possuem a mesma distância para todos os vizinhos, como no caso do mapa.

No desafio do labirinto, algoritmos que tentem a se espalhar menos, como o *Depth First*, possuem um ganho de desempenho ao visitarem menos nós, o que leva a economizar tempo de execução.

Já nos algoritmos de otimização como o *Hill Climbing*, aplicados ao desafio das rainhas, vimos um resultado bem próximo nas abordagens, com exceção do *Stochastic*, que apresentou um resultado bem mais demorado que os demais.

REFERÊNCIAS

RUSSELL, S.J.; NORVIG, P.; DAVIS, E. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010. (Prentice Hall series in artificial intelligence). ISBN 9780136042594. Disponível em: <https://books.google.com.br/books?id=8jZBksh-bUMC>.