Algoritmos e Estruturas de Dados II — Leonardo Heredia
Bernardo Müller e Santiago Firpo

Resumo

Com o uso de mapas e otimização de caminhos, podemos encontrar a rota mais eficiente para chegar ao nosso destino, economizando tempo e recursos. A proposta deste trabalho é acharmos o percurso mais otimizado em um percurso com a ajuda de algoritmos com grafos. A solução para um algoritmo mais otimizado foi feita com a utilização do A*, um algoritmo que combina a distância percorrida até o momento com uma heurística que estima a distância restante até o destino. Alguns pontos importantes com a complexidade do algoritmo, detalhes da sua implementação e seus resultados também serão discutidos neste trabalho.

Introdução

Os mapas e algoritmos com grafos são ferramentas essenciais em diversas áreas, desde navegação por GPS até planejamento de rotas em logística e otimização. Para nossa implementação, foi escolhido o algoritmo *A**, amplamente utilizado em problemas de *pathfinding* em sistemas de mapeamento e *games*.

O algoritmo A* é utilizado para encontrar o caminho mais curto em um grafo ponderado. Ele combina a distância percorrida até o momento com uma heurística que estima a distância restante até o destino. Utilizando uma fila de prioridade, o A* explora os caminhos mais promissores primeiro, priorizando eficiência. Ele é aplicado em diversas áreas, como navegação GPS e logística, e permite otimizar o planejamento de rotas.

Problema

Com um veículo que se move apenas em Norte, Sul, Leste e Oeste. Montar um algoritmo que caminhe entre os portos (que são possíveis de alcançar) da

maneira mais eficiente. Visto que um ponto no mapa gasta 1 uma unidade de gasolina, a maneira mais eficiente deve ser a que passa em menos pontos para alcançar seu objetivo.

Este é um clássico problema de caminho mais curto em grade, muito comum em jogos 2D.

Modelagem do problema

A primeira estrutura implementada para resolver o problema foi o *Grid*, uma estrutura genérica que representa uma grade bidimensional de elementos de qualquer tipo, que permite acesso a qualquer elemento por meio de suas coordenadas. Internamente, o *grid* guarda todos os seus elementos em uma *array* reta alocada na *heap*.

Foi escolhida essa representação ao invés de uma matriz para armazenar todos os elementos de forma contígua porque essa disposição contígua é mais amigável para o cache e reduz a quantidade de misses, visto que a localidade dos dados é respeitada. O *grid* é usado tanto para o carregamento do mapa quanto para a resolução do caminho por meio do A*.

Para armazenar os portos de destino, foi utilizado um *Hash Map* para garantir operações de tempo constante.

Sobre as estruturas utilizadas no A*, foi decidido que cada nodo possui os dados de custo de caminhamento. Foi utilizada uma fila de prioridade para armazenar os nodos a pesquisar (lista aberta) e garantir acesso constante ao nodo ótimo, assim como inserção e remoção em tempo logarítmico. Para a lista fechada, que representa os nodos já pesquisados, foi utilizado um *hash set* para garantir a adição e consulta em tempo constante.

A implementação usa orientação a dados para tornar a representação dos dados o mais amigável possível para o *cache* e a *CPU*. Este é um paradigma que indica algumas diretrizes, como, por exemplo: reduzir o uso da heap ao máximo, não utilizar referências, usar tipos localizados na stack (como structs), reduzir o número de alocações dinâmicas ao máximo, entre outros. Esse mesmo paradigma é utilizado em jogos de enorme escala com objetivos de desempenho extremos, como *Sea of Thieves*, para utilizar o *hardware* em sua potência máxima.

Metodologia

Para solucionar o problema de encontrar o caminho de maneira otimizada e eficiente, foi escolhido o algoritmo de busca de caminho A*. Este algoritmo é utilizado para encontrar o caminho mais curto entre dois pontos de um grafo ponderado, considerando tanto a distância percorrida até o momento quanto uma heurística que estima a distância restante até o destino. Essa combinação garante que o *A** faça escolhas informadas sobre quais caminhos explorar primeiro, meljorando a eficiência do algoritmo. Por isso, pode-se dizer que o A* é uma evolução do algoritmo de Dijkstra, com uma busca consideravelmente mais otimizada em grafos onde a função heurística é aplicável.

Para utilizar o *A** em um mapa, é necessário representar o mapa como um grafo, onde os nodos representam locais e as arestas representam as conexões entre esses locais. Cada aresta possui um peso ou custo associado, que representa a distância ou o esforço necessário para atravessá-la. O grafo também deve conter informações sobre a localização do ponto de partida e do destino.

Pode-se também representar o mapa como um *grid* simples. Nessa representação, não é necessário guardar informações sobre os vértices, pois todas as distâncias entre nodos são iguais. Cada nodo só precisa saber todos os seus vizinhos e se este é um nodo caminhável ou não.

Solução

O algoritmo A* consiste em:

- Adicionar o nodo inicial na lista aberta.
- Enquanto a lista aberta conter elementos, o algoritmo remove o nodo com o menor custo total (custo total é igual a custo de caminhamento mais o custo heurístico, a distância estimada até o nodo final).
- Se o nodo atual removido for o nodo de destino, o *loop* é quebrado.
- Todos os nodos vizinhos do nodo atual pesquisado são verificados. Caso o custo de caminhamento para o vizinho a partir do nodo atual seja menor que o custo mínimo de caminhamento atual do vizinho, o custo do vizinho é atualizado e o seu nodo de origem agora é o nodo atual e o vizinho é adicionado para a lista aberta.

- Quando não há mais nodos para serem pesquisados e o *loop* for quebrado, existem duas alternativas:
    - Caso o nodo de destino tenha um nodo de origem, isso significa que existe um caminho possível. O algoritmo então traça o caminho utilizando o nodo de origem em cada nodo, criando uma espécie de lista encadeada invertida.
    - Caso o nodo final não tenha nodo de origem, isso significa que o nodo de destino é inacessível a partir do nodo de início, e não existe caminho possível entre os dois nodos.

Para este algoritmo, está sendo utilizada a distância Manhattan para a função heurística, por ser uma heurística rápida, eficiente e confiável.

A complexidade do algoritmo A* é $O(b^d)$, onde b é o fator de ramificação, ou o número médio de opções de arestas por nodo. No nosso cenário, esse número está entre 2 e 4, mas se aproxima de 4. Já d é o número de nodos no caminho mínimo desejado. Isso significa que a complexidade temporal do algoritmo é exponencial. Teoricamente, o algoritmo é extremamente custoso e complexo, visto que a classe de complexidade exponencial é uma das mais complexas. No entanto, na prática, a complexidade temporal é extremamente dependente da eficiência da função heurística. Uma função heurística perfeita é aquela que torna o fator de ramificação 1, ou seja, para cada nodo, só há uma opção para ser pesquisada. Funções heurísticas boas reduzem o fator de ramificação efetivo, deixando ele entre o fator de ramificação geral e 1.

Já a complexidade espacial é sempre $O(b^d)$, onde b é o fator de ramificação geral (a média de escolhas possíveis de arestas para cada nodo). Já d é o número de nodos no caminho mínimo desejado. Nesse caso, a complexidade possui sempre o mesmo valor, pois todos os nodos precisam ser armazenados, já que todos podem ser, em teoria, pesquisados.

Exemplo

A simulação abaixo é uma visualização do algoritmo A*, considerando que as diagonais de um nodo também são seus vizinhos:

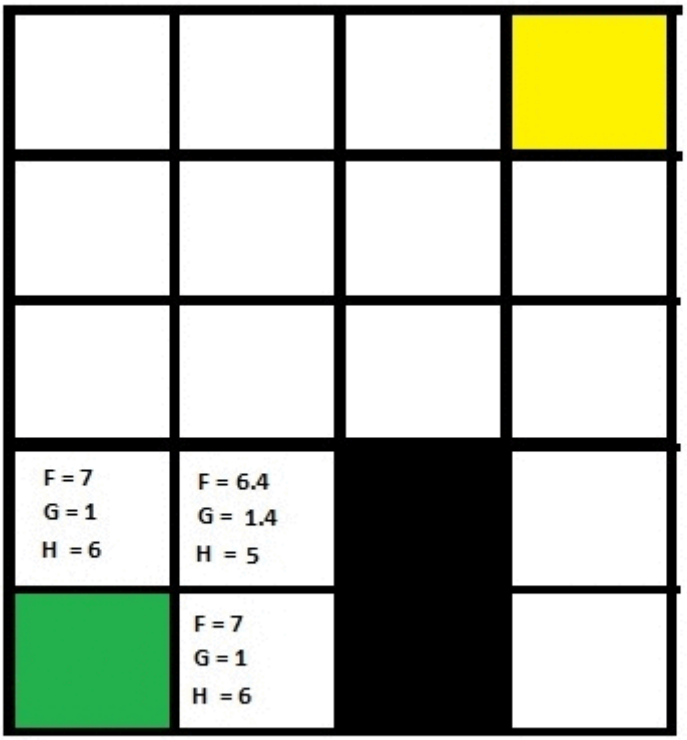


Figura 1: visualização do algoritmo A*. Fonte: *Brilliant*. Disponível em https://brilliant.org/wiki/a-star-search/

Resultados

Mapa 0:

```
Please insert the path to the map file:
/Users/bernardomuller/ItineraryMapSolver/maps/mapa0.txt
..1.......
.....*....
..........
*.........
........2*
Found a path of length 10 between harbor 1 at position IntVector { X = 2, Y = 3 } and harbor 2 at position IntVector { X = 8, Y = 0 }:
..1.......
..@@@*....
....@@....
*....@....
.....@@@@*
```

```
Found a path of length 10 between harbor 2 at position IntVector { X = 7, Y = 0 } and harbor 1 at position IntVector { X = 2, Y = 4 }:
..@.......
..@..*....
..@@@@@...
*.....@...
......@@2*
Complete itinerary path:
..@.......
..@@@*....
..@@@@@...
*....@@...
.....@@@@*
Total walking cost of itinerary (1 -> 2 -> 1) is 20
```

Total walking cost of itinerary (1 -> 2 -> 1) is 20

Mapa 1:

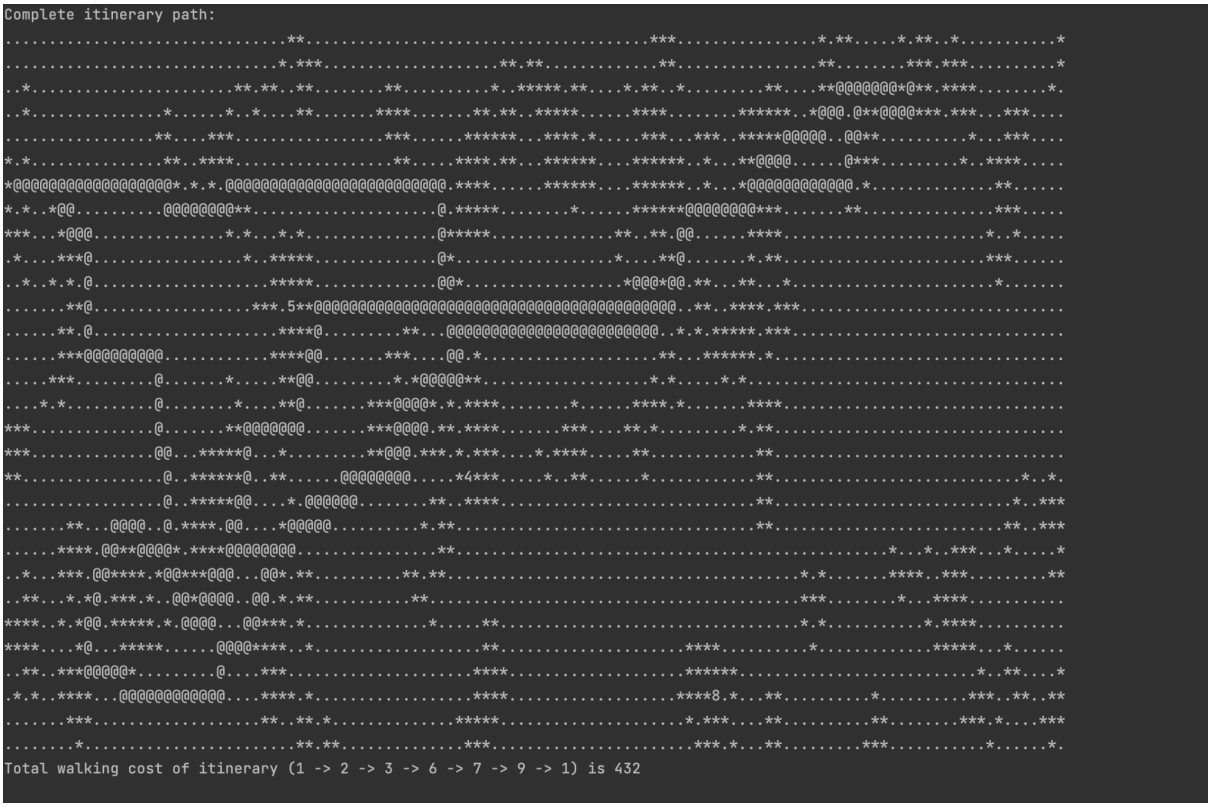
```
/Users/bernardomuller/ItineraryMapSolver/maps/mapa1.txt
***...**.................*.........********.......................*.******....*
***.....**.............*.**.**........******......*..................**2****5..*.
...*....****............***............***.....***...................*.*..*....*
...*********..*.*.....................*......1.**..*.................**.**.....
....**........***.*.............................*.****....*...........***.**....
...*.........****7.*...................*.........*.***..*.**............**.......
..**.............**.....................**...........*...****.............***.....
..***..........***.....................***...............****........*.*.*9......
..***.........***.......................................*****.........******.....
..*....*......**.*.....................................*...........*.****.....
........*.......*.............................8****....*........*....*.*......
.......***.....................****...........*.4.**..........*.**......*......
.......***...................**...******.............3***.......**.*..............
.............................*..******.................6**...******..............
.................................****..................******...****..............
No path exists between harbor 1 at IntVector { X = 46, Y = 11 } and harbor 2 at IntVector { X = 70, Y = 13 }. Ignoring..
```

```
Found a path of length 14 between harbor 1 at position IntVector { X = 46, Y = 10 } and harbor 3 at position IntVector { X = 51, Y = 2 }:
***...**.................*.........********.......................*.******....*
***.....**.............*.**.**........******......*..................**2****5..*.
...*....****............***............***.....***...................*.*..*....*
...*********..*.*.....................*......1.**..*.................**.**.....
....**........***.*.....................@@*.****....*...........***.**....
...*.........****7.*...................*.........@*.***..*.**............**.......
..**.............**.....................**.......@....*...****.............***.....
..***..........***.....................***.......@........****........*.*.*9......
..***..........***.............................@@.......*****.........******.....
..*....*......**.*.............................@........*...........*.****.....
........*.......*.............................@****....*........*....*.*......
.......***.....................****...........*@@@**..........*.**......*......
.......***...................**...******.............@@***.....**.*..............
.............................*..******.................6**...******..............
.................................****..................******...****..............
Found a path of length 3 between harbor 3 at position IntVector { X = 50, Y = 2 } and harbor 4 at position IntVector { X = 49, Y = 3 }:
```

```
Complete itinerary path:
***...**.................*.........********.......................*.******....*
***.....**.............*.**.**........******......*..................**2****@@.*.
...*....****............***............***.....***...................*.*..*@@..*
...*********..*.*@@@@@@@@@@@@@@@@@@@@@@@@*.....@.**..*.................**.**@@...
....**........***@*@@.................@@.....@@*.****....*...........***.**@...
...*.........****@.*@@.................*@@@....@@*.***..*.**............**..@@...
..**.............**..@.................**@....@@@@@@*...****............***@@...
..***..........***...@@@@@@@@@.........***@@....@...@....****.........*.*.*@@@@...
..***.........***.............@@@@@@@@@@...@@@@@@@..@@..*****.........******@....
..*....*......**.*.....................@@@@.....@@@@@@@@@@*@@@@@@@@@@@@*.****@....
........*.......*.........................@@@@@@@@@****@@@@*@@@@@@@@@*@@@@*.*@@....
.......***.....................****...........*@@@**@@@@@@@...*.**@@@@@@@*@.....
.......***...................**...******.............@@@@***.....**.*..@@@@@@@@.....
.............................*..******.................@@@@@@@**...******..............
.................................****..................******...****..............
Total walking cost of itinerary (1 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 1) is 304
```

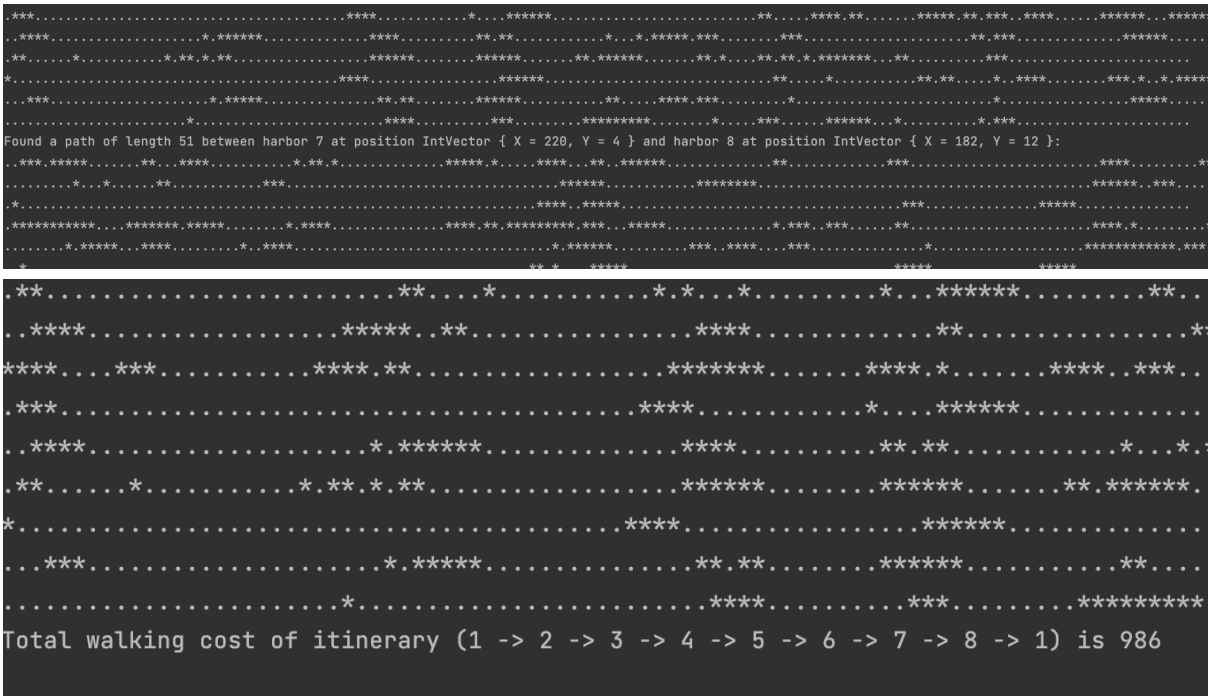
Total walking cost of itinerary (1 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 1) is 304

Mapa 2



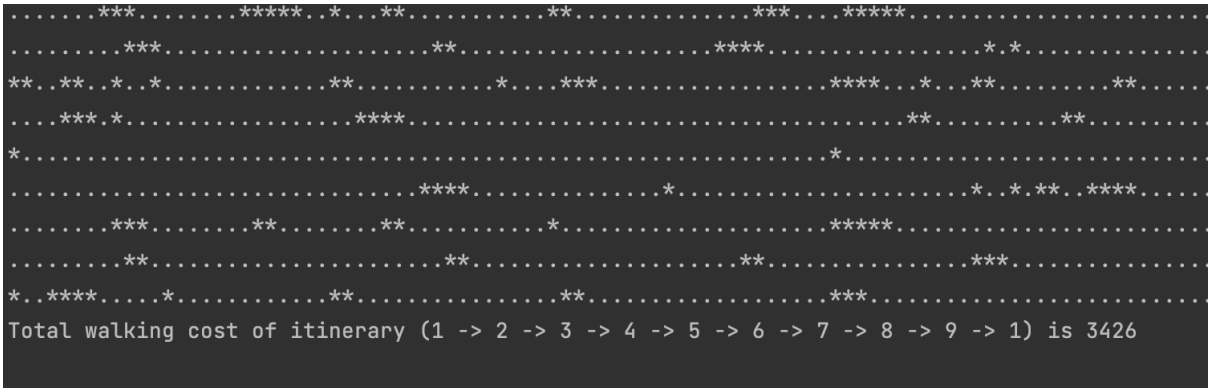
Total walking cost of itinerary (1 -> 2 -> 3 -> 6 -> 7 -> 9 -> 1) is 432

Mapa 3



Total walking cost of itinerary (1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 1) is 986

Mapa 4 (Bônus)



Total walking cost of itinerary (1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 1) is 3426

Conclusão

O algoritmo A* se mostrou uma excelente solução para problemas de *pathfinding* em ambientes com restrições de desempenho. A partir da investigação, foi possível atestar que o algoritmo não é aplicável em todos os problemas de menor caminho, apenas em cenários em que existem funções heurísticas simples e eficientes. Pode-se pensar na heurística como uma função de distância diferente do menor caminho.

Foi comprovada também a eficiência do paradigma de orientação a dados. Por meio da afinidade dos dados com o hardware que os representa, foi possível extrair ainda mais desempenho do algoritmo. Esta eficiência teria se perdido com outros paradigmas que adicionam indireção à execução do programa e afastam a representação dos dados de sua contraparte física. Consequentemente, aumentando o número de *cache misses*. Paradigmas como a orientação a objetos usam mecanismos como funções virtuais e dados estritamente localizados na *heap*.

Ambos abstraiem detalhes da arquitetura do software, embora também contribuam para a fragmentação de memória e custos de desempenho relacionados a alocações dinâmicas.

REFERÊNCIAS

A* Pathfinding in Unity. [*S. l.*: *s. n.*], 2019. Disponível em: https://www.youtube.com/watch?v=alU04hvz6L4&t=4s. Acesso em: 14 maio 2023.

A* Pathfinding (E01: algorithm explanation). [*S. l.*: *s. n.*], 2014. Disponível em: https://www.youtube.com/watch?v=-L-WgKMFuhE. Acesso em: 14 maio 2023.

COX, Graham. **A* Pathfinding Algorithm**. Disponível em https://www.baeldung.com/cs/a-star-algorithm. Acesso em: 20 junho 2023.