Métodos de Pesquisa e Ordenação em Estruturas de Dados

UNIDADE 08

Estrutura de dados: Pesquisando coisas, parte II

Em cursos como Ciência da Computação e Engenharia da Computação é comum estudarmos a fundo algoritmos de pesquisa em grafos. Isso acontece porque esses algoritmos são muito relevantes em áreas que vão desde a navegação em redes de mapas até a otimização de rotas em redes de logística. Os grafos, sendo estruturas que consistem em nós (ou vértices) conectados por arestas, são uma ótima forma de representar uma variedade de problemas do mundo real com uma boa flexibilidade.

Algoritmos como o *Uniform Cost Search* (UCS), A* e Dijkstra desempenham um papel crucial nesse contexto. Por isso, ao aprender sobre esses algoritmos, você não só adquire ferramentas para resolver problemas práticos complexos, mas também desenvolve um entendimento sobre os princípios relacionados à busca e otimização em sistemas complexos.

Vamos lá?

E aquelas outras estruturas?

Por sua natureza, pilhas e filas não são estruturas otimizadas para busca, pois são construídas para gerenciar elementos em ordens específicas, não para realizar pesquisa eficiente entre seus elementos. No entanto, é possível realizar uma busca linear (também conhecida como busca sequencial) nessas estruturas, percorrendo cada elemento sequencialmente até encontrar o desejado.

Em ambos os casos, o custo de busca é O(n), pois você pode ter de olhar cada elemento uma vez. Adicionalmente, se preservar a estrutura original é necessário, você acaba com um custo adicional de tempo e espaço para reconstruir a pilha ou a fila original. É por isso que as pilhas e filas não são as estruturas mais adequadas para operações de busca quando comparadas com outras estruturas de dados, como tabelas de *hash* ou árvores de busca binária, que são projetadas para permitir buscas mais eficientes.

Procurando coisas em grafos - parte II

A busca em grafos pode ser realizada de várias maneiras, dependendo do objetivo. Podemos usar o BFS e o DFS que vimos na semana passada, mas também podemos usar outras técnicas especialmente feitas para grafos, como o UCS, A* (ou A-star) e o algoritmo de Djikstra (pronunciado como “*dáikstra*”).

UCS

O algoritmo *Uniform Cost Search* (UCS), também conhecido como algoritmo de custo uniforme ou busca de custo uniforme, é um algoritmo de busca em grafos que serve para encontrar o caminho de custo mínimo de um nó de partida para um nó de destino. Ele é semelhante à busca em largura, mas com uma diferença fundamental: UCS utiliza uma fila de prioridade para explorar o nó com o menor custo cumulativo primeiramente, em vez de explorar simplesmente o próximo nó na fila.

Aqui está a lógica básica do algoritmo UCS:

+ **Início**

  Coloque o nó inicial na fila de prioridade com um custo cumulativo de zero. Este custo é o custo para chegar ao nó a partir do nó inicial.

+ ***Loop* de busca**

- Enquanto a fila de prioridade não estiver vazia, faça o seguinte:
    - Remova o nó com o menor custo cumulativo da fila de prioridade.
    - Se o nó removido for o nó de destino, então termine a busca; o caminho encontrado é o caminho de menor custo.
    - Se o nó removido não for o nó de destino, expanda o nó (isto é, examine todos os seus vizinhos ou sucessores).
    - Para cada vizinho, calcule o custo cumulativo para alcançá-lo a partir do nó inicial, que é igual ao custo cumulativo do nó atual mais o custo para chegar ao vizinho a partir deste nó.
    - Se o vizinho ainda não foi visitado ou se um caminho mais barato para o vizinho for encontrado, adicione (ou atualize) o vizinho na fila de prioridade com o novo custo cumulativo.

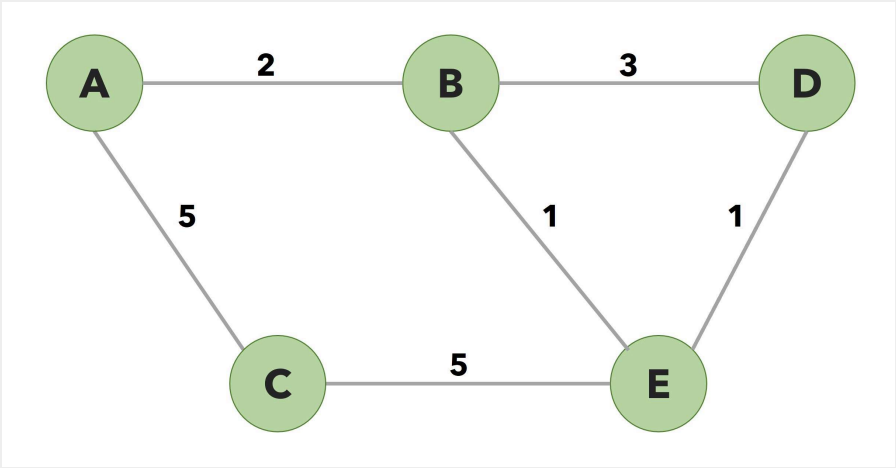
+ Término

A busca termina quando o nó de destino é removido da fila de prioridade, o que significa que o menor caminho foi encontrado, ou quando a fila de prioridade fica vazia, o que significa que não existe um caminho.

EXEMPLO

Suponha que você tem um grafo com 5 nós (A, B, C, D, E), em que A é o nó de partida e E o nó de destino. Veja na imagem a seguir os pesos entre as conexões entre os diferentes nós:

#ParaTodosVerem



Fonte: Os autores (2024).

Agora, como o UCS funcionaria nesse exemplo? Vejamos:

1. **Iniciar**: coloque A na fila de prioridade com custo 0. Afinal, ele é o ponto de partida e ainda não começamos a nos mover para os outros nós.
2. **Expandir A**: remova A da fila, adicione B com custo 2 e C com custo 5.
3. **Expandir B**: remova B da fila, adicione D com custo 2+3=5 e E com custo 2+1=3.
    1. Explicando o custo de D: é a distância para nos movimentarmos de A para B (2) e de B para D (3). Assim, 2 + 3 = 5.
    2. Explicando o custo de E: é a distância para nos movimentarmos de A para B (2) e de B para E (1). Assim, 2 + 1 = 3.
4. **Expandir E**: remova E com custo 3. E é o nó de destino, então o algoritmo termina aqui.

Então, o caminho mais barato de A para E usando UCS é A > B > E com um custo total de 3.

A*

O algoritmo A* é um algoritmo de busca em grafos que é particularmente eficaz para encontrar o caminho mais curto entre um nó inicial e um nó de destino. Ele estende o conceito do algoritmo de busca de custo uniforme (UCS) incorporando heurísticas que estimam o custo do caminho mais barato do nó atual até o nó de destino.

Isso permite que o A* descarte caminhos que, embora possam ser promissores quando olhamos a trajetória que fizemos no passado (g(n)), são caros em termos de custo estimado para alcançar o destino (h(n)). A função de avaliação do A* é f(n) = g(n) + h(n), em que:

- **g(n)** é o custo do caminho do nó inicial até o nó n.
- **h(n)** é a heurística estimada do custo de n para o nó de destino.
- **f(n)** é a soma de g(n) e h(n), e é o valor que o algoritmo A* tenta minimizar.

É provável que esta explicação tenha ficado um pouco complexa. Vamos explicar isso melhor com uma analogia. Vamos supor que você está planejando uma viagem de carro da sua casa até a praia. O seu objetivo é encontrar a rota que ofereça o melhor equilíbrio entre a distância já percorrida e a distância restante até o destino.

Aqui, a função **g(n)** é como o odômetro do seu carro, mostrando quantos quilômetros você já percorreu desde o início da viagem. Isso representa o "custo cumulativo passado".

A função **h(n)** é como um GPS que, em vez de mostrar a distância exata que resta, dá uma **estimativa** da distância até a praia com base na sua localização atual. Isso é análogo ao "custo estimado para alcançar o destino". O GPS pode considerar várias condições, como trânsito esperado ou estradas fechadas, para fazer essa estimativa. Contudo, ainda é uma estimativa e este número pode não necessariamente se tornar realidade.

Agora, **f(n)** é como um painel de controle que combina as leituras do odômetro (**g(n)**) e do GPS (**h(n)**) para dar a você uma noção de qual rota pode ser mais eficiente no total. A ideia é que você quer minimizar tanto a distância já percorrida quanto a distância restante.

Assim, enquanto você dirige, você pode se deparar com vários cruzamentos (nós no grafo). Em cada um deles, você verifica seu painel (calcula **f(n)**) para decidir se continua pelo mesmo caminho ou se pega um desvio que possa parecer mais longo, mas que, no final, promete ser mais rápido de acordo com o GPS (**h(n)**). Se o desvio for muito longo,

mesmo que você já tenha percorrido uma distância considerável (**g(n)** é alto), o A* lhe dirá para não seguir por ali, porque o custo estimado para chegar ao destino é alto (**h(n)** é alto).

Portanto, ao tomar cada decisão de qual caminho seguir, o algoritmo A* permite que você ignore as rotas que parecem boas até o momento, mas que levarão a longos desvios no futuro, otimizando para a rota mais rápida e eficiente com base nas informações atuais e nas estimativas do que está por vir.

Vejamos como funciona a lógica geral deste algoritmo:

+ Início

Coloque o nó inicial na fila de prioridade com um valor inicial para f(n), que é simplesmente h(n), pois g(n) é zero no início.

+ *Loop* de busca

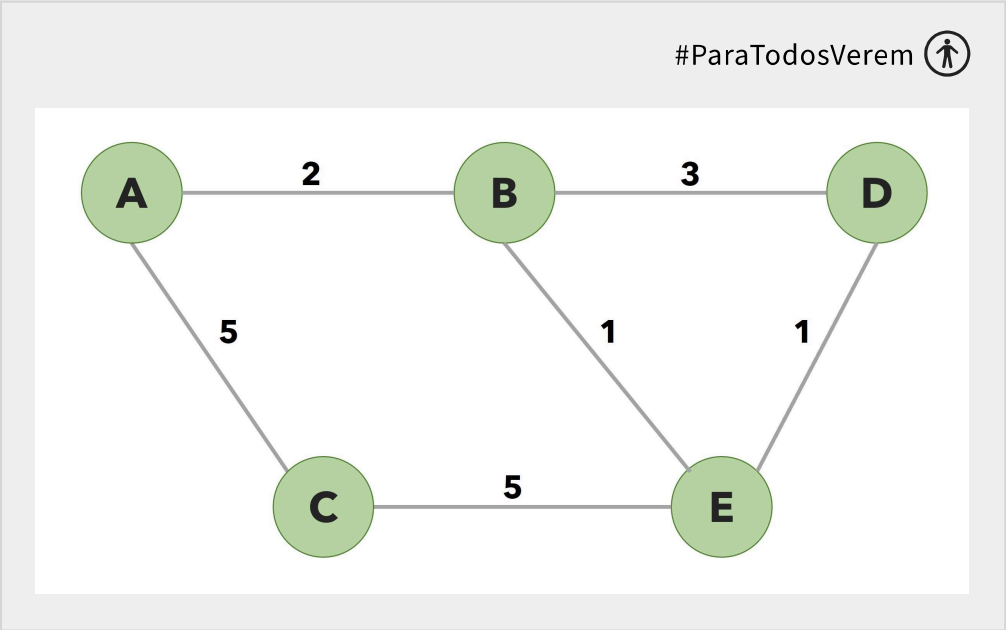
- Enquanto a fila de prioridade não estiver vazia, faça o seguinte:
    - Remova o nó com o menor valor de f(n) da fila de prioridade.
    - Se o nó removido for o nó de destino, reconstrua e retorne o caminho encontrado.
    - Expanda o nó removido e calcule g(n) e h(n) para cada sucessor. Use esses valores para calcular f(n) para cada sucessor.
    - Se um sucessor já está na fila com um valor mais alto para f(n), atualize o sucessor na fila com o novo valor mais baixo.
    - Se um sucessor está no conjunto de explorados, mas o novo caminho é melhor, mova-o de volta para a fila de

prioridade.

+ Conclusão

O algoritmo termina quando o destino é alcançado ou a fila de prioridade fica vazia, indicando que não há caminho possível.

O que acha de retomarmos aquele grafo que vimos anteriormente para o nosso teste? Relembremos o grafo na imagem a seguir:



Fonte: Os autores (2024).

Vamos considerar novamente sendo A o nó inicial e E o destino. A aplicação do A* seria assim:

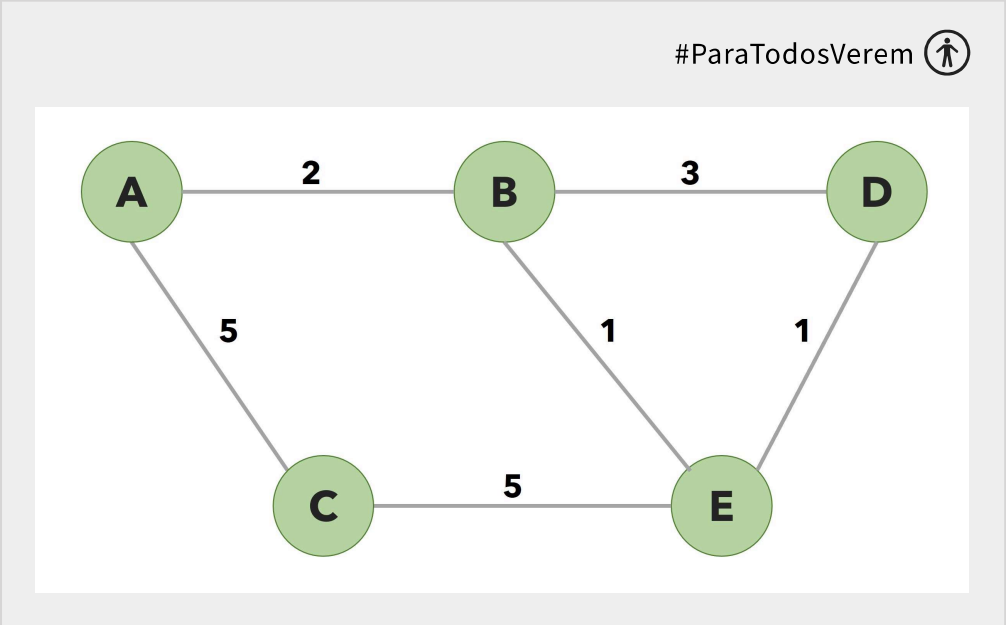
1. Primeiramente, precisamos ter uma estimativa de distância para se chegar ao destino. A estimativa é isso mesmo: um *valor estimado*. Pensando neste exemplo poderíamos pensar em valores como:
   - Entre A – B: h(n) = 3.
   - Entre A – C: h(n) = 1.
   - Entre B – D: h(n) = 1.
   - Entre B – E: h(n) = 0 (afinal, já chegamos ao destino E).
   - Entre B – E: h(n) = 0 (afinal, já chegamos ao destino E).

2. Começamos com A, f(A) = g(A) + h(A) = 0 (porque estamos no nó inicial) + estimativa para E (vamos supor que h(A) = 3, olhando o ponto anterior). Logo, f(A) = 0 + 3 = 3.
3. Expandimos A e adicionamos B e C à fila de prioridade.
    - f(B) = g(A-B) + h(B) = 2 + 3 = 5.
    - f(C) = g(A-C) + h(C) = 5 + 1 = 6.
4. Selecionamos o próximo nó com o menor f(n), que é B.
    - Expandimos B e descobrimos os seus sucessores, D e E.
    - f(D) = g(B-D) + h(D) = (g(A-B) + distância B-D) + h(D) = (2 + 3) + 1 = 6.
    - f(E) = g(B-E) + h(E) = (g(A-B) + distância B-E) + h(E) = (2 + 1) + 0 = 3.
5. D tem o menor f(n), então D é removido e o caminho A -> C -> D é retornado.

Djikstra

O algoritmo de Dijkstra é ensinado com frequência nos cursos Ciência e Engenharia de Computação. É simples e eficiente. Por isso, ele é usado em alguns mecanismos de busca ou em algoritmos de roteamento de redes.

Vamos relembrar o grafo que já usamos nos outros exemplos anteriores, mas agora com um contexto. Logo, vamos supor que você está em um parque de diversões, e cada atração é um nó em um mapa. As trilhas entre as atrações são os caminhos que você pode seguir, e os números nessas trilhas são o tempo que leva para caminhar de uma atração a outra. Você quer encontrar a forma mais rápida de chegar à melhor montanha-russa do parque (E), começando da entrada (A).



Fonte: Os autores (2024).

Os tempos são destacados em minutos entre estes pontos. Ou seja: demoraríamos 2 minutos para andar entre A e B, ou 5 minutos para andar entre C e E. Neste caso, vamos entender como funcionaria o algoritmo de Djikstra:

+ Iniciação

- Começamos marcando a distância ao nó de início, A, como 0 e todas as outras distâncias como infinito (afinal, ainda não exploramos estes outros nós).
- Definimos A como o nó atual.

+ Explorar vizinhos

- Atualizamos as distâncias dos vizinhos diretos de A (B e C) com os tempos de caminhada dados.
- Agora, temos: distância até B = 2 minutos e até C = 5 minutos. As outras permanecem infinitas.

+ Escolher o próximo nó

- Escolhemos o nó com a menor distância conhecida que ainda não foi visitado. Neste caso, B com 2 minutos.

+ Atualizar distâncias

- A partir de B, atualizamos as distâncias para D e E, se encontrarmos caminhos mais curtos por meio de B.
- A distância de B para E é de 1 minuto, mas já temos 2 minutos para nos movimentarmos de A até B. Logo, o tempo total que gastaríamos até o momento seria de 3 minutos se fôssemos a partir de B.
- Continuamos a verificar os caminhos a partir de C e D, atualizando as distâncias conforme necessário.

+ Repetir

- Repetimos os passos 3 e 4 até que todos os nós tenham sido visitados ou a menor distância entre os nós restantes seja infinita (o que significaria que esses nós não são acessíveis a partir do nó de início).

+ Conclusão

- Uma vez que todos os nós foram visitados, o caminho mais curto para cada nó terá sido estabelecido.
- Para a montanha-russa E, podemos rastrear o caminho de volta para encontrar a rota mais rápida.

Visualização do caminho mais curto:

Seguindo os passos apresentados, descobriríamos que o caminho mais rápido de A (entrada) para E (montanha-russa) é A -> B -> E, com um tempo total de caminhada de 2 + 1 = 3 minutos.

Este é o poder do algoritmo de Dijkstra: ele sistematicamente "visita" cada nó da forma mais eficiente, garantindo que, no final do processo, você terá encontrado o caminho mais rápido para o seu destino desejado. Logo, não necessariamente o primeiro caminho é sempre o melhor.

Prós e contras

Agora, o que acha de compararmos estas técnicas? Como já vimos anteriormente, o DFS e o BFS também se aplicam a grafos. Logo, vamos comparar estas duas técnicas com o UCS, o A*, e o algoritmo de Djikstra.

Critério	BFS	DFS	UCS	A*	Dijkstra
Ordem de visitação:	Nível por nível.	Ramo por ramo.	Baseado no custo mínimo até agora.	Baseado em custo + heurística.	Baseado no caminho de custo mínimo atual.
Utilidade:	Caminho mais curto em grafos sem pesos ou com pesos iguais.	Ciclos, componentes conectados, topologia.	Caminho de custo mínimo.	Caminho eficiente com heurística.	Caminho de custo mínimo sem pesos negativos.
Complexidade de tempo:	O(V + E).	O(V + E).	O(V + E log V).	O(V + E) no melhor caso, pode variar.	O((V + E) log V).
Complexidade de espaço:	O(V).	O(V).	O(V).	O(V).	O(V).
Estrutura de dados auxiliar:	Fila.	Pilha.	Fila de prioridades.	Fila de prioridades com heurística.	Fila de prioridades.
Comportamento:	Iterativo.	Recursivo ou iterativo.	Iterativo.	Iterativo.	Iterativo.

Garante solução ótima?	Sim, em grafos sem pesos ou com pesos iguais.	Não.	Sim.	Sim, com heurística admissível.	Sim, sem pesos negativos.
Completo?	Sim (se existe solução, vai encontrar).	Não (pode ficar preso em *loops*).	Sim (se custos são positivos).	Sim (com heurística admissível e custos positivos).	Sim (se custos são positivos).
Otimização:	Não.	Não.	Sim.	Sim.	Sim.
Melhor aplicação:	Grafos e árvores sem pesos ou com pesos iguais.	Exploração e *backtracking* em grafos e árvores.	Grafos ponderados onde custos são a principal consideração.	Grafos ponderados com uma heurística eficiente.	Grafos ponderados sem arestas de peso negativo.

Notas:

- **V** representa o número de vértices no grafo e **E** o número de arestas.
- **Complexidade de tempo e espaço** se refere ao pior caso.
- **Comportamento Iterativo** geralmente usa uma estrutura de dados auxiliar como fila ou pilha para manter o controle dos nós a serem visitados.
- **Garante solução ótima** significa que o método encontrará a melhor solução possível dentro dos critérios estabelecidos.
- **Completo** indica se o método de busca pode sempre encontrar uma solução, se ela existir.
- **Otimização** mostra se o algoritmo é capaz de garantir a solução mais eficiente em termos de custo ou outro critério.

Observações:

- Assim como a busca de custo uniforme, Dijkstra é um caso especial de A*, em que a função heurística é sempre zero.
- O algoritmo de Dijkstra não funciona corretamente se houver arestas com pesos negativos pois assume que uma vez que o nó mais curto é encontrado, não há caminho mais curto para ele. Isso acaba não sendo verdade se um caminho posterior levar a um peso negativo.

Conclusão

Nesta unidade, exploramos as técnicas de pesquisa em grafos. Mais especificamente, o UCS, A* e Djikstra. Esses algoritmos não são apenas ferramentas para resolver problemas computacionais, mas também janelas através das quais vislumbramos a elegância e a complexidade dos sistemas de dados. Aprendemos que, embora cada algoritmo tenha suas peculiaridades e áreas de aplicação ideais, todos compartilham o objetivo comum de encontrar o caminho mais eficiente e otimizado por meio de um labirinto de nós e arestas. Esta exploração nos levou a uma compreensão mais profunda de como os algoritmos podem ser adaptados e aplicados a uma variedade de contextos do mundo real, desde o planejamento de rotas no GPS até a otimização de redes em telecomunicações e logística.

Perceba que a análise desses algoritmos nos proporcionou insights valiosos sobre os princípios fundamentais da ciência da computação, como eficiência algorítmica, o balanceamento de custo-benefício, e a importância da heurística na tomada de decisões computacionais. À medida que avançamos em nossa jornada no campo da computação, levamos conosco não apenas um conjunto de ferramentas técnicas, mas também uma mentalidade analítica aprimorada, capaz de decompor e resolver complexos desafios computacionais. Lembre-se: a computação nada mais é do que um campo da matemática.

Referências

TENENBAUM, A. M. **Estrutura de dados usando C**. São Paulo: Makron Books, 1995.