Análise comparativa de seis métodos clássicos de Busca em Espaço de Estados para a solução do problema 8-Puzzle

André Luiz Moreira Dutra - 2019006345
Introdução a Inteligência Artificial

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

cienciandre@ufmg.br

1. Introdução

O problema das oito peças (8-puzzle) é um *toy problem* no qual oito peças quadradas, numeradas de 1 a 8, são posicionadas aleatoriamente em um tabuleiro 3x3, e seu objetivo é atingir, com o mínimo de movimentos possível, uma configuração específica movendo as peças dado que o movimento de uma peça só pode ser feito a uma posição vazia adjacente. Culturalmente, a eficiência de agentes racionais é testada por meio de problemas como esse. Nesse contexto, este trabalho objetiva implementar, na linguagem Python 3.8, seis algoritmos de busca em espaço de estados, BFS, IDS, UCS, A*, Busca Gulosa e Hill Climbing, e realizar uma análise comparativa do desempenho de cada um na resolução do 8-puzzle sob a perspectiva de múltiplas métricas.

1	2	3
4	5	6
7	8	

Estado solução do 8-puzzle.

2. Estruturas de Dados

Embora os algoritmos já sejam de conhecimento amplo, eles requerem estruturas de dados que permitam sua implementação. Primeiramente, como os algoritmos operam realizando uma busca na árvore de estados do problema, era necessário primeiramente definir o que é um estado, e como transitar entre um estado e outro, essencialmente definindo o caminhamento pelos nós da árvore. Além disso, cada algoritmo requer estruturas de dados

específicas para a fronteira, onde os nós a serem acessados são guardados, então cada fronteira também foi implementada como uma estrutura à parte.

2.1. Nós da Árvore

Os nós da árvore de busca representam um estado do 8-puzzle, e o momento em que ele foi acessado pelo algoritmo. Assim, eles foram representados pela classe No, que contém três atributos: estado, custo e pai. Além disso, a classe também possui funções que obtêm dados importantes para as implementações, como a função de transição, e funções para identificar se o nó é o goal ou se tem ciclos.

Como variáveis da classe, temos o estado, o custo e o pai. O estado de um nó é definido pela configuração das peças naquele momento. Na classe, o estado foi representado como uma string de 9 posições, onde as peças do tabuleiro eram listadas de cima para baixo e da esquerda para a direita, com 0 indicando o espaço em branco. O custo de um nó equivale à altura do nó na árvore, e representa essencialmente quantos movimentos foram necessários para alcançar aquele nó a partir da raiz. Na classe, o custo foi representado como um inteiro. Além disso, é necessário guardar também o nó pai do nó acessado, para que a solução possa ser obtida novamente ao final de cada algoritmo, e por isso na classe uma referência ao nó pai é guardada na variável pai.

Como métodos da classe, temos o construtor, goal, tem_ciclos, print e vizinhos. O construtor da classe constrói o objeto, dados os parâmetros dados. Por default, um nó construído sem pai definido define o pai como None. A função goal verifica se o estado do nó corresponde ao objetivo do jogo, que é basicamente a configuração “123456780”, e retorna um booleano indicando se a igualdade é verdadeira. A função tem_ciclos define se o nó forma um ciclo com algum de seus ancestrais, iterando pelos pais de seus pais à procura de um nó que tenha o mesmo estado. Se sim, ele encontra um ciclo, e se alcançar um nó cujo pai é None ele não encontra, retornando o booleano correspondente. A função print apenas imprime na saída padrão o estado do nó, formatado com um número em cada posição do tabuleiro 3x3, com uma posição vazia no lugar do 0.

Por fim, a função vizinhos retorna os vizinhos do nó que a chama, ou seja, determina a função de transição do problema. Para isso, dado o estado do pai, ele determina as coordenadas i_0 e j_0 do espaço em branco. Em seguida, ele gera quatro coordenadas, incrementando ou decrementando apenas uma das coordenadas do espaço em branco, que são as coordenadas das peças adjacentes ao branco, que são as

únicas que podem se mover. Por fim, ele seleciona entre elas as coordenadas que se encontram dentro do tabuleiro e, para cada uma, gera um novo nó cujo estado tem a peça e o espaço em branco trocados, indicando um dos possíveis movimentos no 8-puzzle. Cada nó filho tem o custo igual ao do pai incrementado de um, representando o movimento realizado, e a referência pai igual ao nó que chamou a função. A função retorna, por fim, a lista contendo os nós vizinhos do nó que a chama.

2.2. Fronteira

A fronteira, nos algoritmos de busca, é a estrutura de nós na qual os nós a serem acessados são armazenados. Como para grande parte dos algoritmos avaliados a única diferença entre eles é a política que define a ordem na qual os nós serão retirados da fronteira, foram implementadas três estruturas diferentes para a fronteira, porém com a mesma assinatura de interface. Assim, um algoritmo base, que recebe a fronteira e apenas coloca e retira nós da estrutura, sem conhecer sua política, pôde ser usado para implementar múltiplos algoritmos, evitando repetição de código.

As três estruturas implementadas foram FilteredPriorityQueue, Queue e Stack, e se encontram no pacote strucs. Todas elas possuem os mesmos métodos: get, que obtém o próximo elemento, put, que posiciona um elemento na estrutura, empty, que retorna um booleano indicando se a estrutura está vazia, e update, que atualiza um elemento na estrutura, se necessário.

As estruturas Stack e Queue são, respectivamente, uma pilha e uma fila comuns, com política de retirada de nós LIFO e FIFO. Sua implementação foi feita usando as listas nativas da linguagem, o que tornou a implementação simples dada a flexibilidade oferecida pelos objetos. No entanto, possivelmente a implementação da fila não foi a mais eficiente, considerando que não foi usada uma fila circular, apenas a lista comum. Em ambos, o método update não faz nada, já que não tem sentido para a implementação de seus respectivos algoritmos, e só está presente para que a assinatura seja a mesma e o algoritmo base funcione em qualquer uma delas.

A estrutura FilteredPriorityQueue funciona de uma forma um pouco diferente. Em alguns dos algoritmos, o nó escolhido para ser retirado é aquele que possui o menor valor para determinada função aplicada sobre ele. Nesse sentido, a FilteredPriorityQueue faz justamente isso. Ela recebe como parâmetro do construtor uma função e a guarda na variável filter. A função filter deve receber o nó e retornar um valor numérico, e será o filtro aplicado sobre o nó para determinar sua prioridade na fila. A fila é implementada como uma lista, e os objetos são adicionados e retirados

usando as funções heappush e heappop do módulo heapq. Para ordenar pelo filtro, os nós, numa chamada a put, são guardados na lista como tuplas (valor da função, nó), e desse modo a comparação é feita pelo valor da função primeiro. Numa chamada a get, a função obtém a tupla e retorna apenas o nó. Por fim, a função update procura na lista um nó de estado igual ao do nó fornecido e, se o filtro aplicado ao nó fornecido for menor que o valor do nó na fila, a tupla do nó da fila é substituída pela do nó fornecido. A lista ao fim é repriorizada usando o método heapify do módulo heapq.

3. Algoritmos e Heurísticas

O objetivo do problema é encontrar a solução para o problema com o menor número de movimentos possível, ou seja, alcançar o nó objetivo (goal) com o menor custo possível. Como nesse caso o custo cresce de um a cada transição, o custo é igual à profundidade da árvore. Portanto, queremos encontrar o nó de menor profundidade na árvore de estados.

Conforme mencionado, os algoritmos implementados foram o BFS, IDS, UCS, A*, Busca Gulosa e Hill Climbing. Além disso, um algoritmo base foi implementado para ajudar na implementação dos algoritmos BFS, UCS, Busca Gulosa e A*, enquanto o algoritmo DFS Limitada foi implementado para ajudar na implementação do IDS.

3.1. Heurísticas utilizadas

Os algoritmos A*, Busca Gulosa e Hill Climbing precisam de informações extras do problema para tomar a decisão sobre que nó escolher, e por isso requerem que heurísticas do problema sejam formadas. Uma heurística é uma função que estima o custo restante de um estado até que chegue no goal. Para o 8-puzzle, foram usadas duas heurísticas, geradas relaxando algumas das restrições do problema.

Relaxando a restrição de que peças não podem se sobrepor, cada peça pode se mover até sua casa com um número de movimentos igual à distância Manhattan até a casa. Esta é a primeira heurística, que soma as distâncias Manhattan de cada peça até sua posição original, e está definida na função H.

Relaxando a condição anterior mais a condição de que uma peça só pode se mover para posições vizinhas, cada peça pode se mover até sua casa com apenas um movimento, saltando da posição errada para a correta. Esta é a segunda heurística, que soma o número de peças fora das suas posições corretas, e ela está definida na função

F, que é a função de avaliação do algoritmo A*, somada ao custo do nó (o que será discutido em mais detalhe na descrição do algoritmo.

Para que uma heurística seja admissível, e portanto possa ser utilizada nos algoritmos, ela deve ser menor ou igual ao custo real entre o estado e o goal. Observe que, como ambas vêm de relaxações do problema, ambas são admissíveis por definição. No caso da primeira, adicionando a restrição de que peças não podem se sobrepor, mais movimentos serão necessários para abrir caminho para movimentar cada peça até sua posição. E no caso da segunda, evidentemente, para mover uma peça até sua posição correta será necessário pelo menos um movimento. Portanto ambas as heurísticas são admissíveis, e por isso podem ser usadas nos algoritmos.

3.2. Algoritmo Base

Conforme mencionado anteriormente, vários dos algoritmos são idênticos, diferindo apenas na política de retirada de nós da fronteira. Por isso, foi implementado um algoritmo base, que recebe o estado inicial e a fronteira, e retorna o estado final alcançado, o tamanho máximo atingido pela fronteira e uma lista com os nós acessados. Todos os algoritmos mantêm uma lista com os nós acessados, mesmo quando não é necessário para o algoritmo.

O algoritmo cria um nó raiz, com o estado inicial e custo 0, o insere na fronteira e cria uma lista vazia de nós acessados. Em seguida, ele entra em um loop infinito. A cada iteração ele primeiro checa se a fronteira está vazia, retornando o nó de erro se for o caso. Em seguida ele tira um nó da fronteira, o adiciona à lista de acessados e checa se ele é o goal, retornando-o se for o caso. Se não, para cada um de seus vizinhos, se ele não estiver na lista de acessados, ele checa se o vizinho está na fronteira. Se não, ele adiciona o vizinho na fronteira, e se sim, ele atualiza o valor do vizinho na fronteira com a função update.

Note que o algoritmo em si não é específico a nenhum dos algoritmos mencionados. Tudo que ele faz é, essencialmente, retirar um nó da fronteira, verificar se é o goal e adicionar os vizinhos ainda não acessados na fronteira, até que o goal seja encontrado ou a fila fique vazia. O que vai transformá-lo em cada um dos algoritmos é a escolha da fronteira.

3.3. BFS, UCS, Busca Gulosa e A*

A busca em largura, ou BFS, visa acessar os nós em ordem crescente de profundidade na árvore. Desse modo, ela acessa a raiz, depois todos os seus filhos, os

filhos dos seus filhos, e assim por diante. Como para esse problema queremos encontrar o nó goal de menor profundidade, o BFS é ótimo e completo, pois acessa cada nível de cada vez e no menor nível em que a solução estiver ela será encontrada. Nesse caso, a estrutura ideal para a fronteira da BFS é uma fila FIFO, pois os nós de um próximo nível entram na fila e só são acessados quando o nível anterior já foi atingido. Portanto, o BFS foi implementado passando para o algoritmo base o estado inicial e a fronteira, como um objeto do tipo Queue vazio.

Já a busca por custo uniforme, ou UCS, visa acessar os nós escolhendo sempre o de menor custo em relação à raiz. Observe que, nesse caso, o custo é igual à profundidade, então o algoritmo funciona essencialmente da mesma forma que o anterior. Desse modo, ele também é completo e ótimo. Para manter a implementação fiel à original, ao invés da fila comum foi usada uma fila de prioridade, do tipo FilteredPriorityQueue. Como queremos que a prioridade dos nós seja dada pelo custo, foi definida uma função G que recebe um nó e retorna seu custo, e ela foi passada como filtro da fila de prioridade. Portanto, o UCS foi implementado passando para o algoritmo base o estado inicial e a fronteira, como um objeto do tipo FilteredPriorityQueue, com filtro G, vazio.

A Busca Gulosa, por sua vez, tem como objetivo acessar os nós escolhendo sempre o de menor heurística em relação à solução, na esperança de que, escolhendo nós cada vez mais próximos da solução, ela será encontrada com baixo custo. É importante ressaltar, no entanto, que este algoritmo não é nem completo nem ótimo, pois não há garantia de que a solução encontrada é feita no mínimo de passos, e é possível que ele entre em um loop infinito e nunca chegue na solução. A Busca Gulosa, nesse caso, foi implementada passando para o algoritmo base o estado inicial e a fronteira, como um objeto do tipo FilteredPriorityQueue vazio com filtro H, a função da heurística mencionada anteriormente.

O A*, por fim, atravessa a árvore escolhendo sempre o de menor soma do custo do nó e de sua heurística. Desse modo, o A* seleciona os nós em ordem da estimativa de seu custo total, e como a heurística é admissível, a estimativa é sempre menor que o valor real, então a solução ótima é sempre encontrada. Desse modo, o A* é completo e ótimo, e é o de melhor desempenho entre os algoritmos testados. Para estimar o custo total, utilizamos a função F, mencionada nas heurísticas, que calcula uma heurística diferente da anterior e a soma ao custo do nó. Desse modo, o A* foi implementado passando para o algoritmo base o estado inicial e a fronteira, como um objeto do tipo FilteredPriorityQueue vazio com filtro F.

3.4. DFS Limitada e IDS

O algoritmo de busca IDS, a busca por descida iterativa, realiza várias iterações do algoritmo da DFS limitada para diversos limites diferentes. Por esse motivos, para implementá-lo, era necessário implementar primeiro a DFS limitada.

O algoritmo da DFS limitada realiza uma DFS, uma busca por profundidade, com um limite máximo de profundidade definido. Assim, ele acessa um nó, em seguida acessa cada filho e seus respectivos filhos sucessivamente até que encontre a solução ou o limite seja atingido. Desse modo, ele não entra em loop infinito, e portanto é completo se a solução estiver dentro do limite.

Observe, no entanto, que nesse caso o algoritmo base não pode ser usado, pois um nó pode ser acessado mais de uma vez. Suponha, por exemplo, que a solução está dentro do limite, mas que um caminho diferente acessa um dos nós pertencentes ao caminho da solução antes de ele ser atravessado. Nesse caso ocorreria uma falha se cada nó fosse acessado apenas uma vez. Portanto, o acesso a um nó mais de uma vez é permitido, contanto que um ciclo não seja formado, e isso pode ser verificado pela função tem_ciclo do nó.

A implementação da DFS limitada foi feita, portanto, de forma muito semelhante ao algoritmo base, porém sem as verificações se os vizinhos pertencem à lista de acessados ou à fronteira. Ao invés disso, é checado apenas se há ciclos. O algoritmo portanto cria a fronteira, com um objeto Stack, padrão para o caminhamento em DFS, cria um nó raiz com o estado inicial e o adiciona na pilha. A lista de nós acessados ainda é criada e mantida para que possa ser retornada ao fim da execução, caso seja solicitada a impressão. Em seguida, ele entra em um loop até que a pilha esteja vazia, onde a cada iteração ele retira um nó da pilha, adiciona na lista de acessados, verifica se é o goal, nesse caso retornando o nó, se não for, verifica se o custo do nó superou o limite dado, nesse caso continuando e não adicionando os vizinhos na pilha e, se o limite não tiver sido superado, ele adiciona cada um de seus vizinhos à pilha se eles não formarem ciclos. Se o loop terminar, indicando que a fila está vazia, ele retorna o nó de erro.

O IDS, por sua vez, funciona realizando múltiplas DFS limitadas. Começando com o limite -1, ele entra em loop realizando a DFS limitada para o estado inicial dado. Se o estado retornado for o goal, ele retorna o estado. Se não, ele incrementa o limite e realiza uma nova DFS limitada. Note que, como ele acessa um nível mais profundo a cada iteração, e como o algoritmo de DFS limitada é completo, o primeiro

nível em que ele encontra a solução é o menor nível possível, e portanto ele é completo e ótimo. Além disso, como a fronteira no DFS atinge tamanho máximo linear, ele é melhor que o BFS e UCS em complexidade de espaço.

3.5. Hill Climbing

O Hill Climbing é um algoritmo de busca local, ou seja, seu objetivo é apenas encontrar o goal, sem necessariamente se preocupar com o número de estados acessados no processo. Nesse sentido, seu funcionamento é bem simples, mas ele não é nem completo nem ótimo. A partir de uma métrica aplicada sobre cada nó, seu objetivo em sua versão original é, partindo do nó raiz, com o estado inicial, escolher sempre o vizinho de menor valor, caso o valor dele seja menor que o do nó atual, e repetir esse processo até que se chegue em um nó com valor menor que todos os vizinhos, o chamado mínimo local, que é o nó retornado pelo algoritmo.

Na implementação usada, a métrica aplicada sobre o nó foi a heurística da distância de Manhattan, definida na função H, uma vez que, quanto menor o seu valor, mais próximo da solução um estado está. No entanto, diferentemente da versão original, a implementação usada permite que um nó salte para o menor vizinho não só se ele tiver valor menor, como também se seu valor for igual ao do nó de saída. Com isso, é possível atingir algum dos múltiplos nós de custo igual que tenha um vizinho menor, caso exista, e encontrar um mínimo local ainda mais baixo. Porém, também é possível entrar em loop infinito acessando os mesmos vizinhos de mesmo valor. Por esse motivo, quando o algoritmo seleciona um vizinho de valor igual, ele verifica se o vizinho ainda não foi acessado. Há também um limite de iterações para o loop, para que ele não entre em loop acessando múltiplos estados de mesmo custo, mas com a restrição de não retornar a estados acessados o limite sequer foi necessário.

4. Análise dos Resultados

Entre as diferenças de cada algoritmo, as que mais se destacam são a otimalidade, completude, complexidade de tempo e de espaço. Um algoritmo é completo quando sempre encontra a solução, e é ótimo quando a solução encontrada é a de menor custo. Além disso, considerando implementações otimizadas, a complexidade de tempo é proporcional ao número de nós expandidos, ou seja, o número de iterações de cada loop, enquanto a complexidade de espaço é proporcional ao tamanho máximo da estrutura de dados da fronteira.

Desse modo, para realizar uma comparação prática dos algoritmos, eles foram testados com sete instâncias diferentes do 8-puzzle, cada uma com custo mínimo de movimentos igual a 3, 6, 9, 12, 15, 18 e 21, respectivamente. Para cada instância e cada algoritmo, foram coletados o custo do estado final encontrado, se ele é a solução, o número de nós expandidos, o tempo de execução absoluto e o tamanho máximo atingido pela fronteira.

1		3
4	2	5
7	8	6

Solução: 3

1	5	2
4	8	3
7	6	

Solução: 6

1		2
8	5	3
4	7	6

Solução: 9

5	8	2
1	7	3
	4	6

Solução: 12

8		2
5	7	3
1	4	6

Solução: 15

8	7	2
5	4	3
1	6	

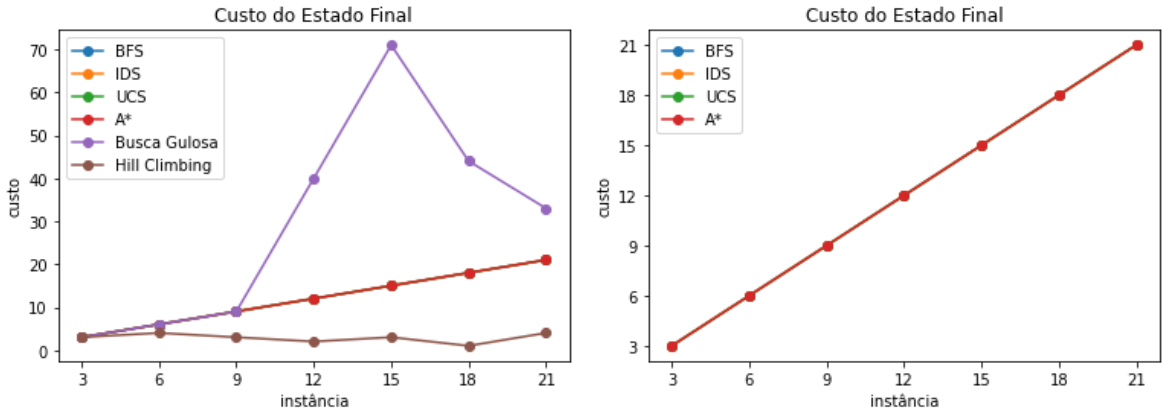
Solução: 18

8		7
5	4	2
1	6	3

Solução: 21

4.1. Solução Obtida

Para cada uma das instâncias, o estado final obtido foi coletado. Nos gráficos a seguir, podemos ver os resultados obtidos para cada algoritmo em cada instância:



No primeiro gráfico, podemos ver o custo do estado final encontrado por cada algoritmo, e os estados finais que não são a solução estão marcados com um "x". No segundo gráfico, os mesmos dados foram agrupados apenas para os algoritmos completos e ótimos. Dentre os algoritmos selecionados, os únicos que não são completos e ótimos são o Hill Climbing e a Busca Gulosa, e isso fica evidente pelos resultados. Enquanto o segundo gráfico apresenta uma correspondência exata entre o custo mínimo da instância e o custo da solução encontrada por cada algoritmo, no primeiro gráfico o Hill Climbing e a Busca Gulosa são os únicos que se dispersam.

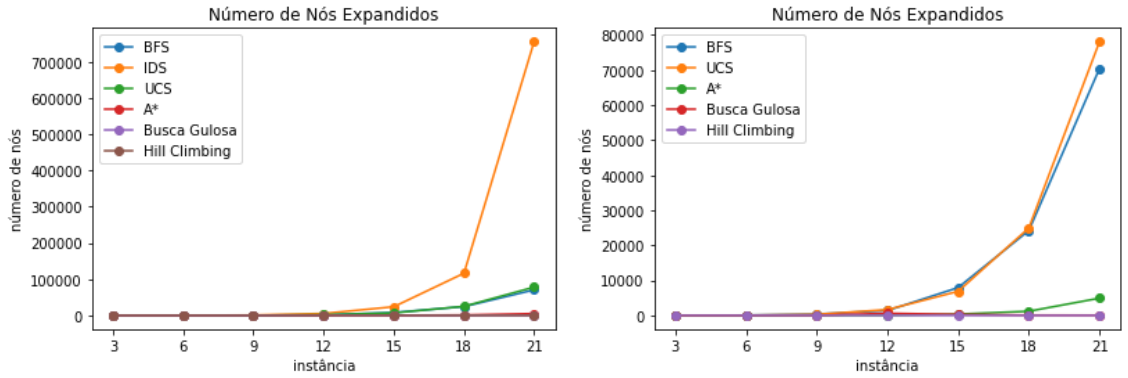
A Busca Gulosa, embora tenha encontrado a solução em todas as instâncias, o custo obtido só foi ótimo para as três primeiras instâncias. Nesse caso, a restrição de não revisitar estados possivelmente influenciou para que ele obtivesse a solução correta em todas as instâncias, embora isso não possa confirmar a completude do algoritmo.

O Hill Climbing, por sua vez, só conseguiu encontrar a solução na instância de custo mínimo 3, parando em outros mínimos locais nas instâncias de tamanho maior. Como o Hill Climbing também estava restrito a não revisitar estados, isso, em conjunto a outros fatores do problema, fez com que ele parasse em mínimos locais após muito poucos movimentos. Cada estado tinha no máximo três vizinhos para escolher, já que o pai não poderia ser revisitado, e a heurística utilizada, a distância Manhattan, só era afetada pelas distâncias Manhattan das peças trocadas, no caso a peça escolhida e o espaço em branco. Como o espaço em branco fica no canto inferior direito, qualquer movimento que não fosse para baixo ou para a direita aumentaria sua

distância em 1, gerando um estado de heurística total no mínimo igual à do anterior. O que acontece no algoritmo, então, é que uma sequência de movimentos é feita gradualmente até que o espaço em branco chegue no canto inferior direito, ou próximo a ele. Nesse caso, ele só terá dois vizinhos que terão custo no mínimo igual ao próprio, e ele caminha por um número limitado de vizinhos iguais até parar. Como o tabuleiro tem tamanho 3x3, o número de movimentos acaba sendo menor que 10, conforme observado nos gráficos.

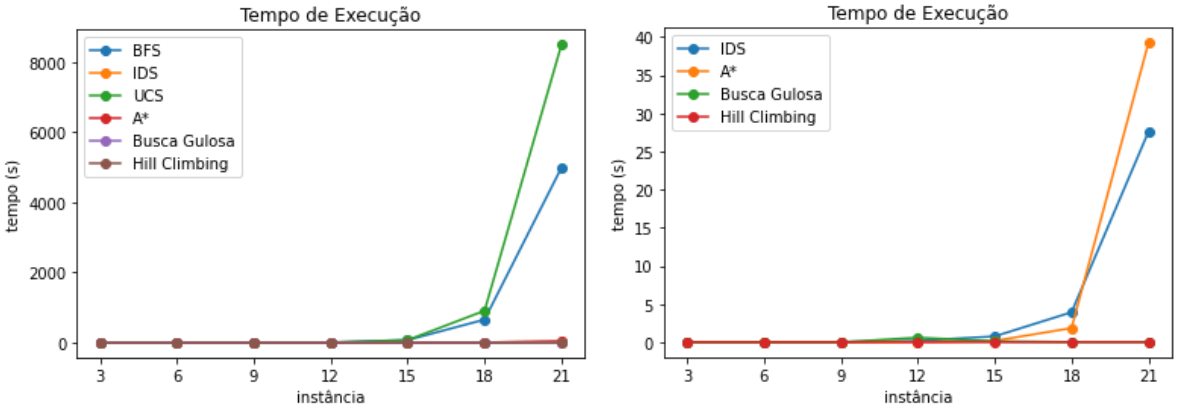
4.2. Número de Nós Expandidos e Tempo de Execução

Conforme observado anteriormente, o tempo de execução, em tese, seria proporcional ao número de nós expandidos usando uma implementação ótima. No entanto, outros fatores podem influenciar o tempo absoluto, como implementações específicas de cada estrutura, ou mesmo a própria estrutura utilizada. Por isso, tanto o número total de nós expandidos quanto o tempo total de execução de cada algoritmo foi coletado. O total de nós expandidos em cada algoritmo pode ser observado nos seguintes gráficos:



Como podemos ver, o IDS foi o algoritmo com maior número de nós expandidos. Esse número é condizente com a implementação, pois, além de ele fazer várias DFS limitadas, revisitando os nós de cada camada inferior novamente a cada camada nova, a própria DFS limitada permitia revisitar nós, contanto que um ciclo não fosse criado, gerando um número de acessos muito superior aos demais. Abaixo dele, e muito próximos entre si, estão os algoritmos BFS e UCS. Conforme mencionado nas especificações de cada algoritmo, no caso deste problema, no qual o custo é equivalente à profundidade da árvore, os dois algoritmos essencialmente operam da mesma forma, e por isso é esperado que o número de nós expandido seja próximo. É notável que o número de nós expandidos de ambos é muito superior que o dos algoritmos restantes, já que ambos acessam todos os nós de cada custo inferior ao

custo da solução antes de a encontrarem. Por fim, os dados também deixam evidente o quanto o A* é ótimo em custo de tempo em relação aos demais algoritmos de solução ótima, já que o número de nós expandidos nele é muito abaixo de todos os outros, estando acima apenas da Busca Gulosa e do Hill Climbing, que não são completos nem ótimos. Observe agora o tempo de execução absoluto de cada algoritmo, medido em segundos, nos seguintes gráficos:

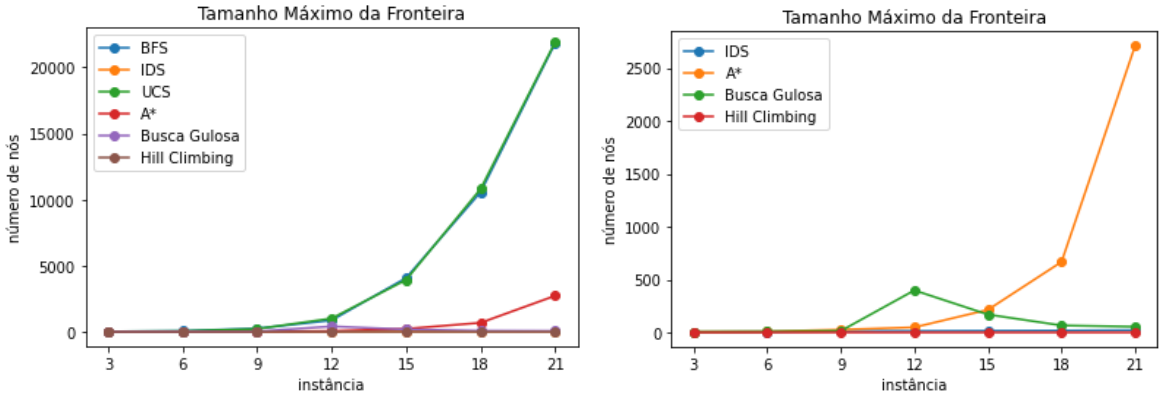


Ao contrário dos resultados anteriores, podemos ver que o BFS e o UCS lideram em tempo absoluto, chegando ao nível de horas em instâncias maiores. Possivelmente, isso se deve a dois fatores: a implementação da fronteira e a checagem se o nó já foi acessado. A fronteira, no caso do BFS, foi implementada com o list de python, e o método get foi implementado apenas retirando o primeiro elemento da lista e movendo os demais elementos para frente. Assim, embora os demais métodos tenham custo escalar, o get tem custo linear para o BFS. Já no caso do UCS, tanto o get quanto o put têm custo logarítmico, já que a fronteira é um heap, e o método update, no caso dele, tem custo n log n, pois reestrutura o heap da fronteira. Assim, em termos de tempo, ele consegue ser ainda mais custoso que o BFS. Para além disso, todos os algoritmos que usam o algoritmo base verificam se o nó já foi acessado, o que é uma operação linear no número de nós acessados, que é exponencial em relação à altura da árvore. Desse modo, é coerente que os algoritmos originados do algoritmo base tenham tempo absoluto maior que o IDS, que tem uma implementação de fronteira de custo escalar em todos os métodos, e cuja verificação de ciclos é proporcional à altura da árvore, muito menor que o número de nós acessados. Assim, o BFS e o UCS, que lideram em número de nós acessados entre os baseados no algoritmo base, têm maior custo de tempo, seguidos do A*, e só então do IDS. A Busca Gulosa, embora também se baseie no algoritmo base, acessa tão poucos nós que não consegue superar os demais algoritmos, enquanto o Hill Climbing, que acessa menos de 10 nós antes de alcançar o mínimo local, é o de menor tempo absoluto,

conforme esperado. Como objetivos para futuras implementações, temos o de implementar a lista de nós acessados como uma tabela hash, para diminuir o tempo de procura a cada nó.

4.3. Tamanho Máximo da Fronteira

Como a estrutura principal de armazenamento de dados em cada algoritmo é a fronteira, o custo espacial é proporcional ao tamanho máximo que ela atinge. Ao contrário do caso anterior, em que tanto o número de nós acessados quanto o tempo absoluto foi medido, para realizar a comparação, o mesmo não poderia ser feito com a memória utilizada, uma vez que todos os algoritmos guardaram uma lista de nós acessados, sendo necessária ou não, para que pudessem ser impressos posteriormente caso solicitado. Desse modo, apenas o tamanho máximo de cada fronteira, em nós, foi acessado, como pode ser visto nos seguintes gráficos:



Novamente, o BFS e o UCS lideram o topo, com um número de nós quase idêntico em todas as instâncias. Em ambos os casos, no pior caso a fronteira guarda todos os nós de uma camada, que pode ter número de nós potencialmente exponencial, o que justifica a superioridade dos valores observados. Embora sejam seguidos pelo A*, seu custo é bem abaixo do BFS e do UCS, o que também confirma a melhoria na complexidade de espaço proporcionada pelo A* na complexidade de tempo e espaço da solução. No entanto, entre os algoritmos completos e ótimos, o que lidera é o IDS, que é inferior inclusive à Busca Gulosa, se comparando ao Hill Climbing, cuja fronteira tem custo constante igual a 1, já que ele não mantém múltiplos estados a serem acessados em uma memória. Este valor é esperado, considerando que a pilha do IDS alcança tamanho máximo proporcional à altura da árvore, que é muito menor que o número de nós acessados. Fica evidente, portanto, que o IDS é superior em complexidade de espaço em relação aos demais algoritmos ótimos.

5. Conclusão

Este trabalho permitiu a consolidação do aprendizado sobre os diversos tipos de algoritmos de busca em espaço de estados, por meio tanto da implementação prática de cada um quanto da comparação do desempenho deles para diferentes instâncias. Por meio dele, foi possível observar, na prática, os benefícios de cada algoritmo em cada aspecto, por meio de dados concretos que permitissem compará-los. Também foi possível ver como implementações diferentes para um mesmo algoritmo podem influenciar seu desempenho a ponto de torná-lo melhor ou pior que outros, apesar dos resultados teóricos. Por meio dos dados, podemos concluir, por fim, que fazendo uma implementação otimizada o melhor algoritmo, em termos de tempo, otimalidade e completude, é o A*, enquanto o melhor em termos de espaço, otimalidade e completude é o IDS. Dependendo da implementação, o IDS pode superar o A* inclusive em tempo, que foi o caso, indo em contrapartida ao seu alto número de nós acessados.

Bibliografia

Chaimowicz, Luiz. *Slides da disciplina Introdução à Inteligência Artifical. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. 2022.*

heapq — Heap queue algorithm — Python 3.10.4 documentation. Disponível em: <https://docs.python.org/3/library/heapq.html>. Acesso em: 28 maio. 2022.