Trabalho Prático 1
Ordenação de vetores usando busca

Isis Ferreira Carvalho
Matrícula: 2020006663

Universidade Federal de Minas Gerais (UFMG)

1. Introdução

O trabalho proposto envolve a aplicação de algoritmos de busca de espaço de estados para encontrar a versão ordenada de um vetor dado como entrada. Basicamente, partimos do vetor de entrada, explorando os vetores filhos (vetores que podem ser gerados a partir do original após uma troca de elementos), o que acaba por gerar um espaço de estados, no qual navegaremos a partir de diferentes algoritmos, até encontrar a solução desejada (o vetor ordenado).

Esse trabalho está organizado da seguinte forma: a seção 2 apresentará as estruturas de dados implementadas e a modelagem dos componentes auxiliares da busca, a seção 3 tratará das diferenças entre os algoritmos implementados, a seção 4 especificará a heurística utilizada, a seção 5 mostrará exemplos de soluções encontradas pelos algoritmos, e a seção 6 fará uma análise comparativa entre a performance dos mesmos, concluindo o relatório com uma discussão dos resultados obtidos.

2. Estruturas de dados e componentes da busca

Para expressar os estados presentes no nosso problema, sobre os quais são efetuadas as buscas, foram implementadas algumas estruturas de dados para representar os nós de busca. Pelo caráter distinto dos algoritmos implementados e dos dados que eles precisam, cada um deles usa uma estrutura diferente especializada apenas para esse algoritmo. Essas estruturas (NodeBFS, NodeDijkstra, NodeIDS, NodeGreedy, NodeAStar) são implementadas como classes, porém todas possuem alguns atributos em comum.

Como NodeAStar é a estrutura mais qualificada e completa, prestaremos atenção especial a ela. Ela é implementada como uma classe com um método de inicialização, o qual declara 7 atributos e recebe 4 parâmetros, e um método auxiliar **invalid_node**, que invalida a exploração de um nó. Os atributos do objeto são:

- **array**: array *numpy* que guarda o vetor em questão (recebe valor do parâmetro *array*);
- **str_rep**: representação em forma de string do array, gerado pela função auxiliar *concatenate_array*, utilizada para fazer busca nos conjuntos de nós explorados e nós da fronteira;

- **real_cost**: custo real de caminhada até o nó em questão (recebe valor do parâmetro *step_cost*);
- **heuristic**: custo estimado pela heurística (recebe valor do parâmetro *heuristic_cost*, ausente nas estruturas relativas aos algoritmos de busca sem informação);
- **parent**: nó que gerou esse nó, utilizado para gerar o caminho entre o nó raiz e o nó final, com o vetor ordenado (recebe valor do parâmetro *parent*);
- **total_cost**: soma do custo real e do custo da heurística, usado pelo algoritmo A* para escolher o próximo nó a ser explorado (presente apenas em NodeAStar);
- **valid**: inteiro que representa se um nó deve ser explorado ou não (recebe 1 em sua inicialização, porém recebe 0, pela função **invalid_node**, se foi encontrado um nó que representa o mesmo vetor porém com menor custo: é um atributo que ajuda a otimizar número de nós explorados).

Além dessas estruturas implementadas, temos a função **concatenate_array(array, sep)**, que gera uma representação *hashable* de um vetor (para fazer a busca nos dicionários e lista na execução dos algoritmos). Temos a função **get_path(node)**, que parte do nó final encontrado na busca e gera o caminho percorrido do vetor inicial até a solução, imprimindo o caminho ao final. Por último, temos a função **explore_node(node)**, que implementa a exploração dos vetores filhos a partir de um nó inicial passado como parâmetro. Essa função retorna uma lista de filhos válidos de um nó, de acordo com o critério detalhado no fórum de Avisos de apenas explorar um vetor se a troca que o gera é vantajosa e aproxima o vetor da solução desejada.

3. Principais diferenças entre os algoritmos

Como abordado em aula, temos que os algoritmos BFS, UCS e IDS são busca sem informação, enquanto a busca gulosa e o A* são buscas com informação. Todos os algoritmos acima são completos, ou seja, eles encontram a solução se ela existir. A seguir, entraremos em breves detalhes de implementação de cada um dos algoritmos, mas no geral, me baseei bastante nos pseudocódigos dados em aula para fazer a implementação de todos.

3.1. BFS

Como visto em aula, o BFS explora o nó que está mais tempo na fila, o que torna possível a exploração por camadas do grafo: isso garante que ele encontre a solução mais rasa existente, porém com custo exponencial de tempo e espaço.

Como o BFS explora nós baseado numa política FIFO, utilizei um objeto *OrderedDict*, que é um dicionário que mantém a ordem de inserção dos objetos, para garantir que os nós fossem explorados na ordem certa e que eu pudesse fazer buscas eficientes na fronteira para checar se um determinado nó já se encontra nela. Com o intuito de fazer buscas eficientes também no conjunto de nós explorados, armazenei esses nós em um dicionário

comum (isso vale para todos os algoritmos que armazenam nós explorados). Sempre que quero checar se um dado nó já está na fronteira ou no conjunto de explorados, essa checagem é feita utilizando o atributo *hashable* representado por *str_rep*.

Como descrito no pseudocódigo, basicamente o BFS:

- Pega o primeiro elemento da fronteira (enquanto ela não for vazia);
- Gera todos os filhos deste elemento;
- Para cada filho, checa se ele não está nem na fronteira nem no conjunto de explorados:
  - Se não estiver, faz o *early goal test* e retorna a solução se o filho for a solução;
  - Se filho não for solução, insere ele na fronteira.

3.2. Dijkstra (UCS)

O UCS explora o nó com menor custo do caminho percorrido, garantindo a optimalidade da solução. Ele faz isso usando uma fila de prioridade, removendo o nó de menor custo presente na fila e explorando esse nó. Para implementar essa fila de prioridade, utilizei o módulo ***heapq*** do Python, que transforma uma lista de tuplas (valor, chave) em um *heap* (fila de prioridade). O componente valor de cada tupla é o custo da solução, e a chave será a representação em *string* do vetor.

Para armazenar os nós de fato (já que isso não é possível com a implementação de *heap* que temos), foi implementado um dicionário para conter os nós da fronteira, além do dicionário que representa o conjunto de nós já explorados. Assim, o funcionamento do algoritmo se dá da seguinte forma:

- Pega o nó de menor custo da fronteira (enquanto ela não for vazia), e checa se ele não foi explorado e se é válido (*valid = 1*);
- Faz o *goal test* e retorna se for a solução;
- Senão, adiciona o nó ao conjunto de explorados, gera seus filhos;
- Para cada filho, checa se não está na fronteira nem nos explorados:
  - Se for o caso, insere na fronteira;
  - Se estiver na fronteira, checa se nó que representa o mesmo vetor tem custo maior: se sim, insere novo nó de custo menor na fronteira e invalida nó antigo.

3.3. IDS

O IDS faz repetidas buscas por profundidade limitada, aumentando o limite a cada iteração, o que garante que ele encontre a solução mais rasa existente (como no BFS), porém sem gastar espaço exponencial com o armazenamento dos nós explorados (o que, por consequência, faz com que vários estados sejam explorados de forma repetida). O tempo continua exponencial.

Para conseguir implementar a busca por profundidade, precisamos fazer a retirada de nós da fronteira usando a política LIFO. Para tal, foi usada uma lista, visto que o método *pop*

de listas em Python remove o seu último elemento e o retorna. O algoritmo de busca em profundidade limitada foi implementado da seguinte forma:

- Pega o último nó da fronteira (enquanto ela não for vazia), faz o *goal test*;
- Se não for a solução, checa se o limite de profundidade foi atingido com o nó atual:
  - Se for o caso, volta ao primeiro passo;
  - Se não for o caso, gera os filhos do nó e os insere na fronteira.

O IDS apenas itera por diferentes profundidades, fazendo buscas por profundidade limitada até encontrar uma solução (e temos certeza de que isso ocorrerá, pois o vetor solução é atingível a partir do vetor original).

3.4. Greedy e A*

Vamos passar pelas implementações desses 2 algoritmos em conjunto, pois elas são bastante similares. Como sabemos, a busca gulosa explora sempre o nó com menor heurística, chegando a uma solução sub-ótima de maneira bem rápida, que é a vantagem desse algoritmo. Já o A* garante solução ótima, por explorar os nós com menor soma do custo real com a heurística. Assim como o Dijkstra, ele atualiza um nó que já esteja na fronteira, caso ele encontre um nó que represente o mesmo vetor porém com custo menor.

Na verdade, a implementação é exatamente a mesma do Dijkstra, porém levando em consideração a função de avaliação de cada um deles. Um detalhe importante de diferença entre os 2 é que, quando a busca gulosa encontra um nó cujo vetor já está na fronteira, ela não checa se o atual possui custo menor, visto que, como a função de avaliação é a própria heurística, esse custo será o mesmo e essa situação nunca ocorrerá.

4. Especificação da heurística

A heurística implementada basicamente faz a contagem de quantas posições do vetor se encontram fora do lugar. Por exemplo, um vetor [1, 3, 5, 4, 2] tem heurística = 3, pois apenas o 1 e o 4 estão em seus devidos lugares. Podemos até enxergar a heurística como sendo como que um limite inferior para a solução de uma versão relaxada do problema: se qualquer troca tivesse custo 2, a heurística seria ainda menor que o custo real da solução dessa nova versão do problema.

A heurística proposta é admissível, visto que temos, no pior caso, um vetor completamente desordenado (em que todas as posições estão incorretas), com heurística = n. Para ordenar esse vetor, precisaríamos de, no mínimo, `ceil(n/2)` trocas. O custo mínimo que essas trocas podem representar é de `2*ceil(n/2)`, que definitivamente é ≥ a n. Assim, a heurística é sim admissível, pois o custo que ela representa será sempre ≤ ao custo real do caminho até a solução.

5. Exemplos de soluções encontradas

Para exemplificar e comparar soluções encontradas pelos diferentes algoritmos, rodaremos todos eles nos baseando em uma só entrada: o vetor [5, 3, 7, 2, 8, 1, 4, 6]. Para cada algoritmo, repetiremos a execução 5 vezes, de forma a obter uma média do tempo gasto. Obviamente as execuções de um mesmo algoritmo retornarão a mesma solução, pois todos eles são determinísticos. Na próxima seção, exploraremos as execuções de forma mais analítica.

Na tabela abaixo, se encontram o custo, a quantidade de nós explorados e o caminho encontrado até a solução para cada algoritmo a partir da entrada indicada acima, para ilustrar as diferenças entre os algoritmos.

	BFS	**Dijkstra**	**IDS**	**Greedy**	**A***
Média dos tempos (5 ex)	0.085119s	0.122527s	1.490332s	**0.001206s**	0.056448s
Custo	20	**18**	**18**	20	**18**
Nº de estados explorados	1.611	2.186	10.013	**6**	441
Caminho até a solução	5 3 7 2 8 1 4 6 1 3 7 2 8 5 4 6 1 2 7 3 8 5 4 6 1 2 3 7 8 5 4 6 1 2 3 4 8 5 7 6 1 2 3 4 5 8 7 6 1 2 3 4 5 6 7 8	5 3 7 2 8 1 4 6 5 3 2 7 8 1 4 6 5 2 3 7 8 1 4 6 5 2 3 7 1 8 4 6 1 2 3 7 5 8 4 6 1 2 3 4 5 8 7 6 1 2 3 4 5 6 7 8	5 3 7 2 8 1 4 6 5 3 7 2 6 1 4 8 5 3 7 2 1 6 4 8 5 3 4 2 1 6 7 8 5 3 2 4 1 6 7 8 5 2 3 4 1 6 7 8 1 2 3 4 5 6 7 8	5 3 7 2 8 1 4 6 1 3 7 2 8 5 4 6 1 2 7 3 8 5 4 6 1 2 3 7 8 5 4 6 1 2 3 4 8 5 7 6 1 2 3 4 5 8 7 6 1 2 3 4 5 6 7 8	5 3 7 2 8 1 4 6 5 3 2 7 8 1 4 6 5 2 3 7 8 1 4 6 5 2 3 7 1 8 4 6 1 2 3 7 5 8 4 6 1 2 3 4 5 8 7 6 1 2 3 4 5 6 7 8

6. Análise quantitativa e discussão dos resultados

A seguir, faremos experimentos com tamanhos crescentes de entrada, de 5 até 11. Para que seja possível comparar o desempenho dos algoritmos, para cada tamanho de

entrada, rodamos cada algoritmo com o mesmo vetor, fazendo 5 simulações por vetor para obter um tempo médio por entrada mais confiável.

Na tabela a seguir, se encontram os resultados desses experimentos, com gráficos ilustrando alguns pontos de comparação entre os algoritmos mais adiante. Percebe-se, de cara, que o tempo de execução escalaria muito para entradas de tamanho 12, se mantidaxpara a maior parte dos algoritmos (apenas o BFS e a busca gulosa retornariam num tempo aceitável). Por conta disso, exploramos apenas até vetores de tamanho 11.

Vetor usado nos experimentos	Algoritmo	Tempo médio	# nós explorados	Custo da solução
3 4 2 5 1	B	0.000574	22	14
	U	0.001012	33	10
	I	0.001756	6	12
	G	0.000319	4	12
	A	0.001041	15	10
5 6 2 4 3 1	B	0.003092	74	16
	U	0.014946	261	12
	I	0.007808	309	12
	G	0.000579	4	12
	A	0.005263	48	12
4 6 2 5 7 1 3	B	0.019432	575	24
	U	0.034018	727	16
	I	0.429925	589	18
	G	0.000954	6	18
	A	0.016918	207	16
7 6 8 2 4 5 1 3	B	0.193517	3226	16
	U	0.556484	5996	16
	I	2.863339	382988	20
	G	0.000926	5	16
	A	0.040946	282	16
8 3 9 7 2 4 6 1 5	B	1.260283	18710	22
	U	3.271193	36996	20
	I	91.463288	10644297	22
	G	0.0011	6	22

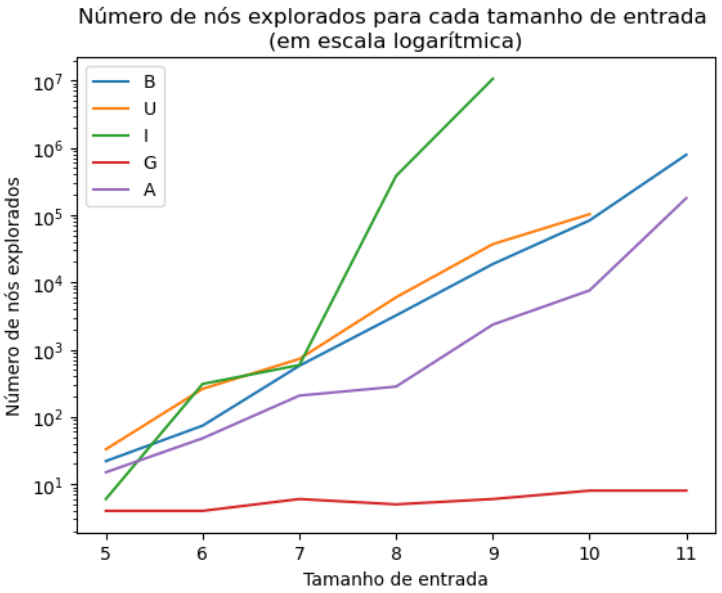
	A	0.335404	2355	20
7 3 9 8 2 6 5 1 10 4	B	12.422783	83370	26
	U	10.991297	103036	22
	I	-	-	-
	G	0.001776	8	28
	A	1.220062	7594	22
3 8 11 9 5 2 10 6 7 1 4	B	125.017582	789929	30
	U	334.877594	2372974	28
	I	-	-	-
	G	0.002393	8	32
	A	35.811419	178761	28

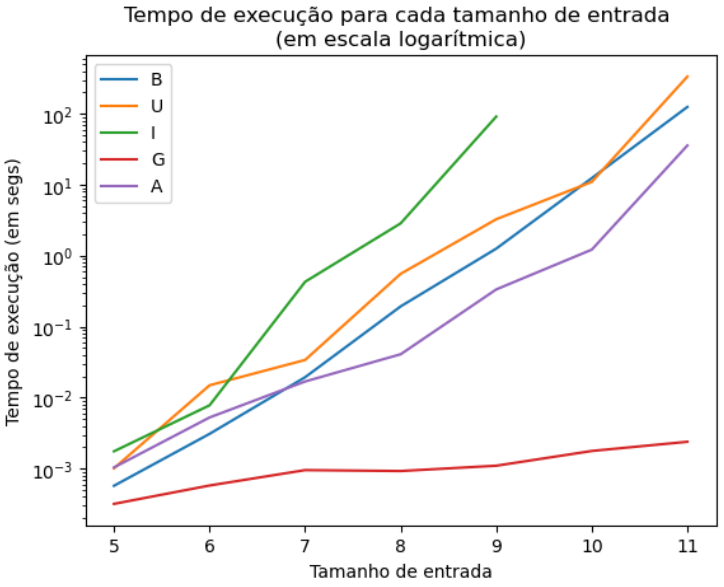
Como discutido na seção 3, os algoritmos Dijkstra e A* realmente sempre encontram uma solução com o menor custo possível, porém o A* em geral é bem mais rápido e explora uma quantidade muito menor de nós para chegar à solução. Isso faz sentido, pois o A* é basicamente o Dijkstra, porém com o uso de informação adicional para guiar sua busca: a heurística proposta na seção 4. Fica evidente a superioridade do A* em relação aos demais, pois é o algoritmo que retorna solução ótima e possui o 2º melhor tempo de acordo com os experimentos feitos.

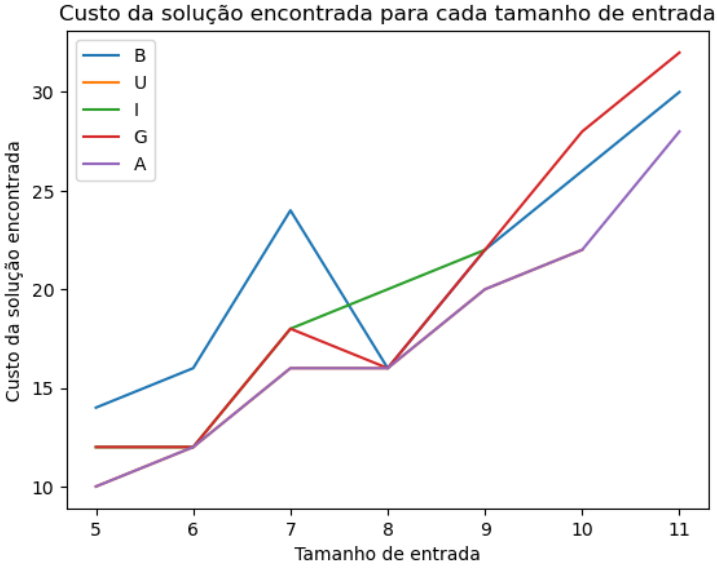
O IDS, como visto na maior parte dos testes feitos, é o algoritmo mais lento: ele explora a maior quantidade de nós entre todos os algoritmos propostos, por não armazenar os nós explorados anteriormente (ele repete a busca já feita em explorações anteriores). Idealmente isso não ocorreria, pois no pseudocódigo mostrado em aula, é para existir uma checagem de ciclos que impediria essa exploração repetida: porém, só é possível fazer isso guardando os nós já explorados, o que vai contra a “proposta” do algoritmo original, que é encontrar a solução mais rasa com uma complexidade de espaço de ordem linear. Por conta dessa decisão de implementação, o algoritmo não rodou para entradas de tamanho maiores que 9.

Para entradas de tamanho 12, foi feita uma tentativa de rodar os algoritmos (sem incluir o IDS). Contudo, meu computador não aguentou a carga de processamento e acabou matando o processo por si só.

Por último, a busca gulosa é o melhor algoritmo em relação à escalabilidade: ele retorna uma solução sub-ótima com muita rapidez até para entradas bem grandes. Entretanto, como podemos ver nos gráficos a seguir, para entradas maiores, ele realmente retorna soluções cada vez piores. A seguir, temos alguns gráficos só para facilitar a visualização dos dados presentes na tabela.


Número de nós explorados para cada tamanho de entrada
(em escala logarítmica)
Número de nós explorados
Tamanho de entrada
B
U
I
G
A


Tempo de execução para cada tamanho de entrada
(em escala logarítmica)
Tempo de execução (em segs)
Tamanho de entrada
B
U
I
G
A


Custo da solução encontrada para cada tamanho de entrada
Custo da solução encontrada
Tamanho de entrada
B
U
I
G
A