

Search Sort: usando algoritmos de busca para ordenar vetores

Lucas Paulo Martins Mariz

¹Universidade Federal de Minas Gerais

²Departamento de Ciência da Computação – UFMG

lucaspaulom@hotmail.com

Abstract. *We use ordering functions on a daily basis, either to sort small vectors or to define a certain priority for the execution of tasks. Algorithms such as bubble sort, quick sort, among others, are widely used in these approaches, however, there are some other methods to perform such procedures. Therefore, we will discuss the application of state space search algorithms, such as breadth-first search and uniform cost search, in order to explore and analyze their versatility and efficiencies.*

Resumo. *Diariamente utilizamos funções de ordenação, seja para ordenar pequenos vetores ou para definir uma determinada prioridade para a execução de tarefas. Algoritmos como bubble sort, quick sort, entre outros, são amplamente utilizados nessas abordagens, entretanto, existem alguns outros métodos para realizar tais procedimentos. Sendo assim, discutiremos a aplicação de algoritmos de busca em espaço de estados, como busca em largura e busca de custo uniforme, com o intuito de explorar e analisar suas versatilidades e eficiências*

1. Introdução

O problema da ordenação é amplamente conhecido na matemática e computação. Em sua abordagem mais geral, pode-se dizer que ele tem como principal função a organização dos dados em uma determinada sequência previamente definida.

Os algoritmos mais performáticos para esse tipo de problema não encontram grande dificuldade quando aplicados em conjuntos grandes de dados. Por outro lado, ao mudar a para a metodologia de busca em espaço de estados, encontramos diversas barreiras que, na maioria dos casos, tornam-se limitações.

Assim sendo, serão discutidas as diferentes formas de gerar e explorar o espaço gerado em cinco algoritmos diferentes: busca em largura, aprofundamento iterativo, busca de custo uniforme, busca gulosa e A*. Para esses 2 últimos uma heurística admissível foi implementada, a qual será apresentada posteriormente.

2. Procedimentos e Métodos

Os algoritmos a serem implementados foram apresentados no decorrer da disciplina. Sendo assim, a ideia principal consistiu em modificar as estruturas definidas com o intuito de satisfazer o problema proposto.

Dado a facilidade e poder providos pela linguagem python, no que tange à análise e manipulação de dados, ela foi escolhida como o principal recurso de implementação.

Com ela, bibliotecas robustas puderam ser exploradas, fator que contribuiu positivamente para as análises seguintes. Todo o código desenvolvido pode ser encontrado no repositório do Github (<https://github.com/LucasPMM/search-sort>).

Além disso, para cada conjunto de vetores, foram realizadas 10 execuções para se obter uma métrica de tempo consistente. Os tempos de execução variaram devido a gargalos do meu computador, fator que, caso não fosse contornado, poderia influenciar significativamente na qualidade dos resultados. Por outro lado, diversas variações de entradas foram avaliadas, pois, dessa forma, pode-se classificar a performance do algoritmo em diferentes situações. Assim, com o intuito de obter uma média melhor e evitar resultados extrapolados, tais medidas foram tomadas.

3. Implementação

A implementação foi subdividida em algumas etapas para facilitar a compreensão de todo o processo. Serão detalhados a seguir as classes e funções utilizadas.

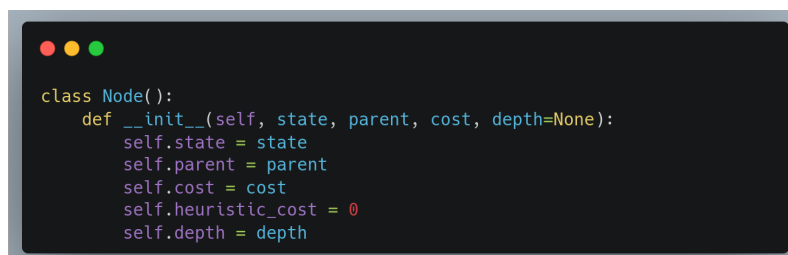
3.1. Custos, heurísticas e branching factor

Como determinado na instrução do trabalho prático, a troca entre elementos adjacentes possui custo 2 e, caso contrário, o custo seria 4. Inicialmente, desenvolvi toda estrutura utilizando o `heapq` do python, entretanto, o fator inconstante da ordenação dessa abordagem fez com que filas fossem implementadas em seu lugar.

Além disso, sabe-se que o branching factor é um fator de suma importância no tempo de execução do código. De forma a otimizar tal métrica, foi instituído um controlador que permite uma troca entre os elementos i e j (i menor que j) somente se o i -ésimo elemento for maior do que o j -ésimo elemento. Tal mudança, aparentemente sublime, chegou a reduzir pela metade o número de nós expandidos em alguns exemplos.

Por fim, algumas heurísticas foram testadas e foi escolhida a que performou melhor: a quantidade de elementos fora da posição correta em relação ao valor objetivo (que é o vetor ordenado). Essa heurística é claramente admissível, pois com n pares de elementos fora de posição, teríamos um custo resultante de pelo menos $2n$ para ordenar.

3.2. Definição da classe Node



```
class Node():
    def __init__(self, state, parent, cost, depth=None):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic_cost = 0
        self.depth = depth
```

Figura 1. Implementação da classe Node.

Uma classe principal nomeada `Node` foi desenvolvida com o intuito de armazenar algumas informações cruciais. Cada nó possui um nó "pai", o qual gerou tal "filho" e o custo associado a sua geração. Além disso, são armazenados o custo determinado pela heurística, que será utilizado nos algoritmos A* e guloso, e a profundidade do nó em questão, que será utilizada no aprofundamento iterativo.

3.3. Definição da classe Search

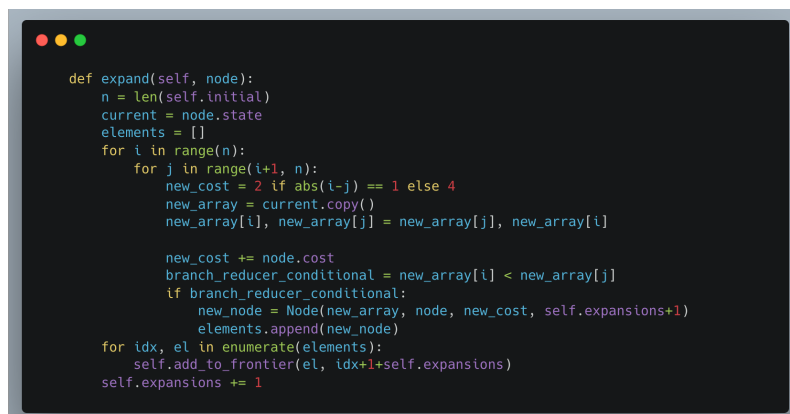
Um outra classe nomeada Search foi definida com o intuito de definir uma base para todos os algoritmos a serem executados bem como reaproveitar a maior quantidade de código possível. Em sua estrutura principal são armazenados o nó inicial, o objetivo, o algoritmo a ser utilizado, os nós já explorados, a fronteira a ser explorada e a quantidade de expansões.



```
class Search():
    def __init__(self, initial, algorithm):
        self.initial = initial
        self.goal = sorted(initial)
        self.algorithm = algorithm
        self.explored = []
        self.frontier = None
        self.expansions = 0
```

Figura 2. Implementação da classe Search.

Também definida nessa classe, a função de expandir o um nó basicamente passa por todos os elementos do array montando as possíveis combinações e calculando o custo correspondente. A adição do elemento gerado na fronteira só ocorre se respeitar a condição de um elemento ser maior do que o outro perante seus indexes, anteriormente discutida, e caso respeite as condições necessárias de cada algoritmo, que serão apresentadas nos próximos itens (Figura 3).



```
def expand(self, node):
    n = len(self.initial)
    current = node.state
    elements = []
    for i in range(n):
        for j in range(i+1, n):
            new_cost = 2 if abs(i-j) == 1 else 4
            new_array = current.copy()
            new_array[i], new_array[j] = new_array[j], new_array[i]

            new_cost += node.cost
            branch_reducer_conditional = new_array[i] < new_array[j]
            if branch_reducer_conditional:
                new_node = Node(new_array, node, new_cost, self.expansions+1)
                elements.append(new_node)
    for idx, el in enumerate(elements):
        self.add_to_frontier(el, idx+1+self.expansions)
    self.expansions += 1
```

Figura 3. Implementação da função de expansão.

Por último, temos a função de iniciar o processo de busca, a qual continua a expandir os elementos até que a fronteira esteja vazia. Há uma condição especial para os algoritmos A* e custo uniforme, evitando que elementos já explorados sejam novamente expandidos (Figura 4). Tal condição aparenta ser redundante, contudo, ela evita certos cenários já que foram desenvolvidas uma abordagem usando fila e outra usando heap.

No caso do A*, considerando uma heurística admissível, quando um nó é expandido o menor caminho até ele foi encontrado e, assim sendo, não é necessário expandir novamente. No caso do custo uniforme, foi tomada uma decisão para otimizar o tempo de execução. Ao encontrar o elemento de melhor custo, seria necessário substituí-lo na fronteira caso estivesse presente nela. No entanto, tal processo de remover um elemento e reordenar a fila de prioridades se mostrou bastante custoso. Sendo assim, os elementos são adicionados na fronteira e, conseqüentemente, o de menor custo será explorado primeiro e, para evitar a duplicidade da exploração, evitamos que elementos repetidos sejam explorados novamente. Uma consequência seria uma fila maior, ou seja, o loop vai rodar mais vezes, mas, como temos uma condicional que pula o elemento logo no início, esse novo problema é menos custoso que o cenário anterior.

Três algoritmos foram feitos de 2 formas, usando heapq e usando queue, são eles: UCS, greedy, A*. Usando heapq foi encontrado um problema descrito na documentação: as funções do heap não mantêm a fila ordenada, apenas garantem que a propriedade do heap é mantida. Dessa forma, nem sempre os elementos são expandidos na ordem correta e, conseqüentemente, os algoritmos levam mais tempo para finalizar. Novamente, foi escolhida a abordagem usando filas e a comparação entre as duas metodologias será realizada na seção 4.

Dito isso, a implementação usando heap é bem similar à implementação usando filas e seus detalhes serão discutidos nas próximas seções.

```
def start(self):
    self.init_frontier()
    while not self.empty_frontier():
        node = self.next_node()
        if node.state == self.goal:
            return node.cost, self.expansions, node.solution_path()

    if self.algorithm in ['U', 'A'] and node.state in self.explored:
        # If node was already explored (with better cost) on these algorithms, skip
        # If the heuristic is admissible the best path to the node has already been found when it is explored (A*)
        # If the best cost node was already explored, there is no need to explore it again (UCS)
        continue

    self.set_explored(node)
    self.expand(node)
```

Figura 4. Implementação da função de busca.

3.4. Busca em Largura

Começando pela implementação mais simples, a busca em largura, definimos apenas uma função de adicionar um nó na fronteira, a qual é responsável por verificar se este já foi explorado ou se já é pertencente a fronteira.

```
def add_to_frontier(self, node, _):
    if node.state in self.frontier or node.state in self.explored:
        return
    self.frontier.append(node)
```

Figura 5. Adicionar elementos na fronteira (BFS).

3.5. Aprofundamento Iterativo

Por sua vez, o aprofundamento iterativo também possui uma metodologia muito semelhante a busca em largura para adicionar elementos na fronteira (Figura 5). A sua maior diferença está na função de busca (Figura 6).

Em linhas gerais, definimos um loop infinito que será executado até o que o algoritmo retorne o valor desejado. A cada ciclo, a variável que armazena a profundidade máxima a ser explorada é incrementada, fazendo com que o algoritmo faça sucessivos recomeços.



```
def start(self):
    depth_limit = 0
    while True:
        self.reset()
        self.init_frontier()
        while not self.empty_frontier():
            node = self.next_node()
            if node.depth > depth_limit:
                continue

            if node.state == self.goal:
                return node.cost, self.stored_expansions + self.expansions, node.solution_path()

            self.set_explored(node)
            self.expand(node)

        depth_limit += 1
```

Figura 6. Função de busca no aprofundamento iterativo.

3.6. Busca de Custo Uniforme

A busca de custo uniforme possui uma função de controle de fronteira semelhante a BFS (Figura 5), retirando apenas a condição de checar se o elemento está na fronteira antes de adicionar. Como citado anteriormente, há uma condicional na função de busca principal que cobre o caso de evitar duplicações na exploração desse tipo de algoritmo. Fazendo dessa forma, evitamos também a necessidade de substituir um elemento na fronteira, processo que se mostrou bem custoso, caso o mesmo tenha sido encontrado com um custo melhor.



```
def add_to_frontier(self, node, _):
    if node.state in self.explored:
        return
    if tuple(node.state) in self.frontier_control:
        # Check costs
        index = list(map(lambda x: x[1], self.frontier)).index(node)
        cost = self.frontier[index][1].cost
        better_cost = node.cost < cost
        if not better_cost:
            return
        self.frontier.pop(index)
        heapify(self.frontier)
    heappush(self.frontier, (node.cost, node))
    self.frontier_control.add(tuple(node.state))
```

Figura 7. Adicionar elementos na fronteira (UCS) usando heap

Dito isso, o algoritmo usando heap também foi implementado para fins de comparação. Ao inserir um elemento na fronteira checamos se o mesmo já está presente com um custo maior e, caso positivo, o substituímos (Figura 7).

3.7. Busca Gulosa

A busca gulosa se diferencia das outras por considerar o valor da heurística como guia de expansão. Assim, uma fila de prioridades é montada com o intuito de expandir o elemento que melhora mais a distância do objetivo de maneira imediata (Figura 8).

A implementação usando heap é bem similar: apenas trocamos o custo real da Figura 7 pelo custo da heurística.




```
def add_to_frontier(self, node, iterator):  
    if node.state in self.explored:  
        return  
    self.frontier.put((self.hamming_distance(node), iterator, node))
```

Figura 8. Adicionar elementos na fronteira (Greedy).

3.8. Algoritmo A*

Por fim, o algoritmo A* funciona de uma forma semelhante ao algoritmo guloso, entretanto, ele leva em consideração a soma entre o custo real e o custo da heurística para organizar a fila de prioridades (Figura 9). Sendo assim, ele se torna uma combinação entre a busca de custo uniforme (Dijkstra) e busca gulosa, sendo necessária uma heurística admissível para que seu verdadeiro potencial seja extraído.

A implementação usando heap é bem similar: apenas trocamos o custo real da Figura 7 pelo custo da heurística mais o custo real.



```
def add_to_frontier(self, node, iterator):  
    if node.state in self.explored:  
        return  
    self.frontier.put((node.cost + self.hamming_distance(node), iterator, node))
```

Figura 9. Adicionar elementos na fronteira (A*).

4. Experimentos

Uma das etapas mais importantes de todo esse projeto consiste na experimentação do algoritmo desenvolvido bem como na análise dos resultados obtidos. Sendo assim, diferentes entradas com tamanhos variados foram testadas e tabelas com os dados a serem analisados foram construídas.

4.1. Resultados obtidos

Com a execução do algoritmo em diversos tipos de entradas, mais de 20 tabelas foram montadas (disponíveis no Github: <https://github.com/LucasPMM/search-sort/tree/main/tables>). A partir delas, diversas conclusões podem ser tomadas. Vamos pegar alguns dos dados obtidos para análise.

Algoritmos de busca em espaço de estados são capazes de realizar a tarefa proposta, entretanto, estão longe de serem ideais. Para fins de comparação, enquanto

instâncias com 20 elementos levam milionésimos de segundos para executar com uma função de ordenação trivial, essa mesma instância pode levar dias para rodar em alguns dos algoritmos propostos. Sendo assim, não faz sentido comparar tais implementações com outras funções próprias para ordenações. Desse modo, seguiremos as análises seguintes discutindo os algoritmos entre eles.

BFS				IDS				UCS			
N	Custo	Expansões	Tempo (s)	N	Custo	Expansões	Tempo (s)	N	Custo	Expansões	Tempo (s)
1	0	0	0.000009	1	0	0	0.000010	1	0	1	0.000025
2	2	1	0.000016	2	2	2	0.000022	2	2	2	0.000052
3	4	2	0.000025	3	4	3	0.000034	3	4	4	0.000114
4	6	15	0.000132	4	14	40	0.000330	4	6	15	0.000212
5	8	33	0.000837	5	32	328	0.006057	5	8	60	0.001306
6	10	273	0.041138	6	50	1108	0.049908	6	10	299	0.014475
7	12	694	0.619046	7	72	3056	0.345815	7	12	1516	0.279619
8	14	7403	72.974005	8	98	7270	1.850486	8	14	9514	14.978700
9	16	20561	1270.301665	9	128	15486	8.210359	9	16	59035	1253.012248

GREEDY				A*			
N	Custo	Expansões	Tempo (s)	N	Custo	Expansões	Tempo (s)
1	0	1	0.000026	1	0	1	0.000026
2	2	2	0.000037	2	2	2	0.000038
3	4	2	0.000044	3	4	2	0.000045
4	6	3	0.000064	4	6	5	0.000106
5	8	3	0.000085	5	8	9	0.000269
6	10	4	0.000121	6	10	27	0.001161
7	12	4	0.000162	7	12	93	0.006085
8	14	5	0.000462	8	14	324	0.044251
9	16	5	0.000444	9	16	1227	0.460286

Figura 10. Execuções com arrays em ordem decrescente

4.2. Otimalidade

Em um primeiro momento, pode-se discutir a questão da otimalidade. Apenas dois dos cinco algoritmos implementados encontra o menor custo possível: o A* e a busca de custo uniforme. A maioria dos algoritmos falha em encontrar o ótimo pois os custos podem ser diferentes (2 para trocas adjacentes e 4 caso contrário). Logo, devemos descartá-los e apenas considerar os que retornam resultados ótimos? Não necessariamente! Um exemplo prático: a busca por Aprofundamento Iterativo é completa e linear no espaço. Outras opções se apresentam como exponenciais. Nesse caso, se a prioridade for a otimização de espaço, a aplicação do IDS deve ser considerada.

Uma observação a ser feita: o algoritmo não ser ótimo não significa que ele não pode encontrar o ótimo. Para determinadas instâncias, todos os algoritmos foram capazes de encontrar o caminho de melhor custo, entretanto, tais instâncias são excessões.

Entre as opções ótimas, o A* é aquele que executa em menor tempo. Para entradas de tamanho 9, por exemplo, o A* chega a ser quase 2800 vezes mais rápido que a busca de custo uniforme. Ele também é eficientemente ótimo (expande o menor número de elementos possível dentre os algoritmos ótimos).

4.3. Tempo

De acordo com os dados da tabela (Figura 10), podemos analisar que, à medida que o número de elementos do vetor cresce, o tempo de execução também cresce. Isso

se mostrou como uma regra nesse primeiro teste pois, levando em consideração um array ordenado de forma decrescente, temos um cenário ruim para o algoritmo: um grande branching factor. De acordo com as regras definidas, um elemento i pode ser trocado com um elemento j , i menor que j , se, e somente se, o i -ésimo elemento for maior que o j -ésimo elemento. Nesse cenário, o primeiro elemento poderá fazer trocas com todos os elementos seguintes, fator que, em grandes instâncias, torna o algoritmo lento.

Ao observar a tabela (Figura 10), é perceptível que os algoritmos detentores de alguma heurística (nesse caso a quantidade de elementos fora da posição) se mostraram mais rápidos que os demais a partir de um determinado tamanho de entrada. Havendo algum conhecimento maior sobre o problema que possa ser utilizado, a busca com informação pode ser muito mais eficiente que as outras, como comprovado pelos testes.

4.4. Número de expansões

O número de expansões também é proporcional ao tamanho da entrada e se mostrou significativamente maior em aplicações sem heurística. Uma observação interessante seria que quanto maior o número de expansões não necessariamente leva a um maior tempo de execução. O tamanho da fronteira a ser analisada a cada iteração de cada algoritmo pode ser diferente, levando a tempos diferentes.

Outro ponto a ser analisado é que um algoritmo com poucas expansões pode não levar ao ótimo (o contrário também é válido), como é o caso da aplicação gulosa. Ela se mostrou bem rápida e com poucas expansões na maioria dos testes, entretanto não foi capaz de atingir a otimalidade na maioria dos casos. Sendo assim, caso a prioridade fosse velocidade em sacrifício da perfeição, seria um bom candidato a ser escolhido.

4.5. Tamanho da instância vs Tempo

Outros testes com entradas aleatórias foram realizados (disponíveis no Github: <https://github.com/LucasPMM/search-sort/tree/main/tables>) e é possível perceber que determinadas entradas possuem um tempo de execução menor do que outras, mesmo sendo maiores. Isso ocorre devido ao branching factor que pode ser entendido como: "quanto mais desordenado a entrada, mais custoso será para ordená-la". Ou seja, quanto mais possibilidades de troca estiverem disponíveis, mais o algoritmo tende a demorar. Tal constatação pode ser vista na tabela da Figura 11.

N = 16: [1 3 2 5 4 7 6 10 8 9 12 13 15 18 17 14 11 16]				N = 19: [1 3 2 5 4 7 6 10 8 9 12 13 15 11 17 14 19 18 16]				N = 20: [1 3 2 5 4 7 6 10 8 9 12 13 15 11 17 16 14 20 19 18]			
Algoritmo	Custo	Expansões	Tempo (s)	Algoritmo	Custo	Expansões	Tempo (s)	Algoritmo	Custo	Expansões	Tempo (s)
B	34	27440	106.069731	B	34	12262	15.794709	B	32	18121	38.557624
I	50	1659	0.190443	I	36	927	0.085309	I	38	1138	0.118293
U	28	27620	193.607536	U	26	12244	31.057561	U	26	18349	76.174505
G	34	12	0.001106	G	34	13	0.001083	G	32	12	0.001040
A	28	12336	31.644904	A	26	4056	2.748947	A	26	4816	3.862979

Figura 11. Execuções com arrays aleatórios

4.6. Heapq vs Queue

Como dito anteriormente, duas abordagens foram utilizadas nos algoritmos UCS, greedy e A*: heap e queue. Ao comparar as tabelas das Figuras 11 e 12, é perceptível

que a abordagem utilizando heap gasta mais tempo para encontrar o valor ótimo. Isso acontece pois, de acordo com a documentação e comprovado com testes, o heap não se reordena a cada mudança, fazendo com que a fronteira seja explorada de forma ligeiramente diferente. Sendo assim, considerando a mesma entrada de tamanho 20 por exemplo, o algoritmo A* leva cerca de 6 vezes mais tempo para completar do que usando filas. Por outro lado, não se pode retirar conclusões acerca do número de expansões. Em alguns casos ele difere para cima, em outros para baixo e em algumas exceções não se altera. O valor objetivo não se altera nem no custo uniforme e nem no A*, já que são ótimos, entretanto, chegou a variar na aplicação gulosa.

Dessa forma, considerando o tempo de execução como uma métrica importante, a abordagem utilizando queue se apresentou com um melhor custo benefício.

N = 18: [1 3 2 5 4 7 6 10 8 9 12 13 15 18 17 14 11 16]				N = 19: [1 3 2 5 4 7 6 10 8 9 12 13 15 11 17 14 19 18 16]				N = 20: [1 3 2 5 4 7 6 10 8 9 12 13 15 11 17 16 14 20 19 18]			
Algoritmo	Custo	Expansões	Tempo (s)	Algoritmo	Custo	Expansões	Tempo (s)	Algoritmo	Custo	Expansões	Tempo (s)
U	28	27422	247.084818	U	26	12049	39.369484	U	26	18227	103.585439
G	34	12	0.000971	G	34	13	0.001014	G	32	12	0.001002
A	28	12336	116.105917	A	26	4064	10.667697	A	26	4952	19.826272

Figura 12. Execuções com arrays aleatórios usando heapq

Por fim, a execução fica inviável para a maioria das entradas de tamanho maior do que 10. O número de possibilidades de expansões pode crescer de forma acentuada, fazendo com que tais instâncias levem dias até serem completadas, principalmente nas buscas sem informação. Em muitos casos, o tempo não é o maior vilão, e sim o espaço. Em alguns testes que deixei rodando de um dia para o outro encontrei meu computador completamente travado. Isso ocorre pois a fronteira é mantida na memória com custo exponencial de espaço e, em instâncias grandes, tal fator pode acarretar na lentidão ou até no esgotamento completo do processamento da máquina.

5. Conclusão

A execução desse trabalho se apresentou como uma ótima oportunidade de aplicar diversos conceitos vistos em sala de aula de forma prática, além de instigar a curiosidade sobre os fatores que levam a um determinado resultado: saber os motivos de uma métrica divergir do valor esperado é fundamental.

Com o desenvolvimento de toda a análise apresentada na seção anterior, pode-se perceber que, apesar de questões de eficiência, é possível aplicar algoritmos de busca para realização de tarefas não usuais. Alguns apresentam melhores resultados do que outros em questão de tempo de execução, custo ótimo e custo benefício. Entretanto, nem sempre o que executa mais rápido retorna o valor ótimo e nem sempre o valor ótimo deve ser perseguido perante o custo computacional a ser gasto.

Assim, a coerência da metodologia, ordem de grandeza dos dados, número de instâncias, entre outros, são alguns dos fatores que, em conjunto, determinam a qualidade do resultado e, na medida de sua importância, devem ser investigados mais a fundo com o intuito de obter conclusões mais significativas.

Referências

- [1] Slides disponibilizados pelo professor para a ministração da disciplina