- Apresentação das estruturas usadas e da modelagem dos componentes da busca (estado, função sucessora, etc);

• Breve descrição das principais diferenças entre os algoritmos;

• Especificação da heurística utilizada. Ela é admissível? Por quê?;

• Exemplos de soluções encontradas pelos algoritmos;

• Análise quantitativa comparando os algoritmos com relação ao número de estados expandidos e tempo de

execução para diferentes tamanhos de entrada (por exemplo, vetores de tamanho 5, 10, 15 e 20). **Apresente**

**tabelas e/ou gráficos comparativos**;

• Discussão dos resultados obtidos.

Documentação contendo uma **descrição sucinta** das estruturas de dados, heurísticas e algoritmos empregados

para modelar e resolver o problema, como também uma **discussão dos resultados obtidos**.

**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 uma análise comparativa entre a performance dos mesmos, e a seção 6 concluirá o relatório com uma discussão dos resultados obtidos.

**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, útil 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 em Node);
* **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 (ausente em Node);
* **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.

**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. Os algoritmos implementados diferem em alguns quesitos bastante pontuais, como por exemplo a ordem de exploração dos nós:

* 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.
* O UCS explora o nó com menor custo do caminho percorrido, garantindo a optimalidade da solução.
* 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 tempo continua exponencial.
* 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.
* O A\* também garante solução ótima, por explorar os nós com menos função de avaliação. Assim como o UCS, ele atualiza um nó que já esteja na fronteira, caso ele encontre um nó que represente o mesmo vetor porém com custo menor.

**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.

**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, 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.

**Análise quantitativa e discussão dos resultados**

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 | 0.153655s | 1.490332s | 0.000739s | 0.04183s |
| 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 |

**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, rodaremos cada algoritmo com o mesmo vetor, fazendo 5 simulações por vetor para obter um tempo médio por entrada mais confiável (exceto em casos com processamento muito demorado, que serão rodados apenas 1 vez, e indicados com um asterisco). Na tabela a seguir, se encontram os resultados desses experimentos, com gráficos ilustrando alguns pontos de comparação entre os algoritmos. Os algoritmos que não rodaram em menos de 10 horas serão indicados com um traço.