Ordenação usando busca de Espaço de Estados

Documentação

Samuel Lipovetsky

May 15, 2023

1 Estrutura de dados utilizadas

1.1 Construção do grafo de estados

Inicialmente antes dos algoritmos serem executados, é gerada o grafo que representa os estados e os pesos entres esses estados. Para chegar nesse objetivo , é utilizado um dicionário no python em que cada chave desse dicionário representa um nó da arvore e essa chave tem como valor um outro dicionario em que cada chave secundária representa um filho do nó original. Ou seja, nesse dicionario dict[x][y] representa o valor da função de transição de x para y, sendo que esse valor é 2 se o swap for em dígitos adjacentes ou 4 caso contrario.

Já para facilitar esse processo de construção do grafo a partir das permutações do nó start , é gerado também uma lista de permutações como formato [(1,2,2),(1,3,4)....] sendo que no exemplo (a,b,c) a e b representam os digitos a serem trocados dentro de um estado e c representa o custo dessa troca.

As funcionalidades descritas até agora foram implementadas em um classe chamada adjancecyDict() que tem como métodos generatePermutationList , que gera a lista de permutação, populateAdjList , que gera o grafo a partir da lista de permutação e swap, que é uma função auxilias utilizada na troca dos dígitos.

Dessa forma, após executarmos esses métodos temos o grafo que representa os estados possíveis já com os pesos entre cada nó.

2 Detalhamento dos Algoritmos

2.1 Breadht First Search (BFS)

O BFS é o algoritmo mais simples entre os implementados, é simplesmente uma busca em largura sem nenhuma otimização. Foi utilizada uma fila no python a partir da estrutura de dados deque. Inicialmente, é inserido o nó inicial nessa fila e enquanto a fila não estiver vazia o primeiro elemento da fila é removido e todos os nós filhos desse elemento que já não foram inseridos na fila são inseridos. Para lembrar qual elemento já foi ou não inserido na fila existe um dicionario de nós visitados , de forma que, se exste uma chave nesse dicionario que é um nó do grafo, esse grafo ja foi visitado. Ainda, esse dicionario de nós visitados também é utilizado

para construir o caminho que o algoritmo fez para chegar no resultado, uma vez que , nesse dicionario a querry dict[x]=y representa que y é o antecessor de x no caminho feito pelo algoritmo.

2.2 Iterative Deepening Search (IDS)

O IDS tenta combinar as vantagens do BFS e do DFS. São feitas N iterações , em que N representa a altura da arvore de estados, e em cada iteração é executado uma busca em profundidade porém com a profundidade limitada. A implementação é similar ao do BFS, porém em vez de ser uma fila é utilizada a lógica de uma pilha. Esse algoritmo também usa a estrutura de dados deque do python. Similarmente ao BFS, existe um dicionário de itens visitados, porém , nesse dicionario existe uma informação a mais que é a profundidade daquele nó. Dessa forma, o algoritmo sabe quando a profundidade máxima daquela iteração ocorreu.

2.3 Uniform Cost Search (UCS)

O UCS funciona como um BFS porém em vez de usar uma fila é utilizada um heap que sempre tem como primeiro elemento aquele com menor valor, sendo que esse valor é a distancia do nó inicial até esse nó.Cada elemento da heap é composto por uma tupla (custo, caminho) que representa o menor custo para chegar naquele elemento a partir do no inicial e caminho representa os passos feitos do nó inicial até o elemento.

2.4 A* e Busca Gulosa

Ambos os algoritmos foram implementados igualmente o UCS ,mudando somente a função que avalia qual o menor caminho entre um nó e outro a partir da soma da heurística no caso do A* ou o uso exclusivo da heurística na busca gulosa.

3 A heurística utilizada

A heurística escolhida foi o número de inversões em um estado. É dito que existe uma inversão em uma lista se um par de elementos (a, b) tal que $a < b$ e $v[a] > v[b]$. Assim, a ideia dessa heurística é que estados mais organizados serão menos penalizados e ficariam na frente de estados menos organizados na heap do A* e da busca gulosa.

3.1 Consistência e admissibilidade da heurística

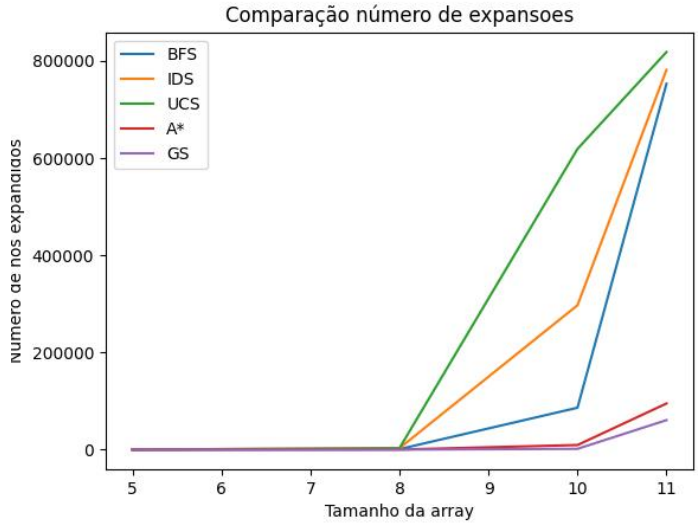
Para uma heurística ser consistente é necessário que tal heurística nunca superestime o custo real. Como a menor operação que é a de trocar dígitos adjacentes tem peso dois e o número de inversão representa a menor quantidade de trocas possíveis para ordernar um vetor, a heurística que utiliza justamente o número de inversões sempre será menor que o custo real, que será sempre pelo menos duas vezes o número de inversões.

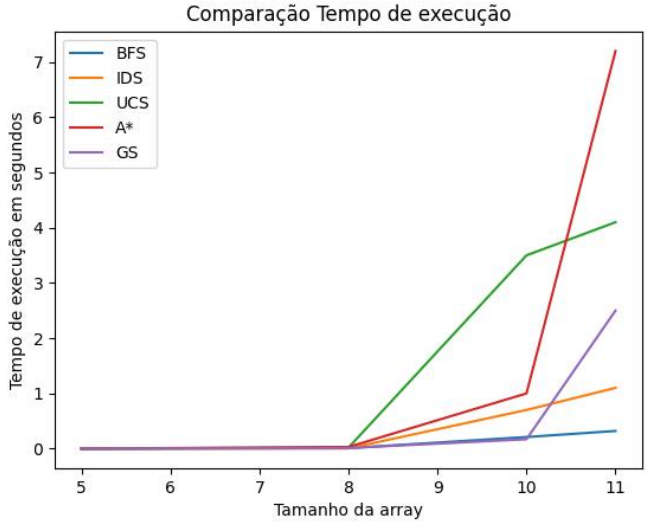
Além disso, para uma heurística ser consistente é preciso que o valor da heurística de qualquer nó nunca seja menor que o custo real de alcançar um nó filho mais o valor da heurística desse nó filho. Nesse caso, como trocamos somente dois elementos

de posição de um nó pai para um nó filho , o número de inversões de um nó filho sempre sera (Número de inversões do no pai - 1) , dessa forma, como o menor custo de pai para filho é 2 , essa heurística empregada também é consistente.

4 Comparação

Para compara a eficiência dos algoritmos foram feitas duas comparações em relação ao número de expansões e em relação ao tempo de execução





podemos perceber que , apesar de A* e GS expandirem uma quantidade menor de nós , o tempo de execução desses algoritmos crescem muito rápido. Dessa forma, para essa instancia do problema existe um trade-off claro entre uso de memória e velocidade. Porém, os algoritmos que usam a heurística foram mais lentos devido ao fato de que a heurística é calculada durante a execução dos algoritmos, enquanto a a criação dos estados é feito previamente, assim do modo que os algoritmos foram

implementados expandir nós é muito mais rápido do que calcular a heurística, uma vez que, o tempo de criar o grafo inicial não foi levado em consideração.

5 Ambiente de execução

A execução dos algoritmos estão conforme com as especificações do Trabalho pratico. Como o algoritmos está feito a partir de um script, deve ser chamada pelo console "python3 ./TP1 ...." no diretório em que TP1 está.