Documentação do Trabalho Prático da disciplina de Introdução à Inteligência Artificial

Kaio Lucas de Sá – 2019006850

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil
desakaio4@gmail.com

1. Introdução

Esta documentação lida com o problema da implementação de diversos algoritmos de buscas de soluções em grafos para o problema 8-puzzle.

A documentação tem como objetivo dar uma descrição de estruturas de dados e algoritmos implementados para o trabalho, bem como das heurísticas escolhidas para os algoritmos A* e Greedy best-first Search. Também será discutido aqui uma comparação dos resultados obtidos por esses algoritmos.

Para a resolução do problema, foi seguida uma abordagem de programação orientada a objetos, onde foram projetadas três classes, necessárias para o funcionamento do código, mantendo a coerência e bom-senso na modularização da mesma. O projeto conta com três classes, sendo elas: Board, responsável por modelar o problema 8-puzzle em si e realizar operações em uma instância do problema, bem como comparar nós explorados com o objetivo, a classe Node, que implementa um nó de um grafo, e a classe Tree que implementa uma árvore qualquer.

A seção 2 traz uma breve descrição das principais estruturas de dados usadas no trabalho, já na seção 3 temos um pequeno resumo dos algoritmos implementados, na seção 4 pode-se encontrar um pequeno estudo de comparação de tempo de execução dos diferentes algoritmos. A seção 5 trata das conclusões do trabalho, e, por fim, a seção 6 apresenta a bibliografia utilizada.

2. Estruturas de Dados

Para a confecção do trabalho foram utilizadas 3 estruturas de dados principais: a estrutura de Matriz, uma Árvore e Listas encadeadas.

A matriz foi utilizada na modelagem do tabuleiro do problema, sendo o principal componente da classe Board, essa é uma solução óbvia para o problema que foi desenvolvido, visto que este se trata basicamente de um tabuleiro bidimensional.

A estrutura de árvore foi o principal componente utilizado pelos algoritmos de busca, como estes tratam de busca pela solução em um grafo a partir de um estado inicial, também fica claro de perceber o porquê dessa decisão, visto que, podemos gerar os níveis da árvore aos poucos, à medida que exploramos seus nós, até que encontremos a solução. A árvore também facilita o processo de impressão do caminho tomado até a solução se este for requisitado no momento de execução do programa.

Já as listas foram utilizadas para implementar as filas de exploração dos algoritmos, mudando seu comportamento de FIFO para FILO de acordo com o algoritmo que seja implementado simplesmente mudando o parâmetro passado para o método pop() da classe de listas já nativa do python. Essa estrutura também foi utilizada para armazenar os nós explorados pelos algoritmos naquela execução.

3. Algoritmos Implementados

3.1 Breadth-First Search

Esse foi o primeiro, e também o mais simples algoritmo implementado, a ideia dele é que basicamente, a partir de um estado inicial nós começamos a explorar os níveis da árvore até que encontremos a solução.

Por exemplo, suponha uma árvore composta pelos seguintes nós A, que possui os filhos B e C, onde, por sua vez B tem os filhos D e E, e C possui os filhos F, G e H. O algoritmo então explora os nós na seguinte sequência: A, B, C, D, E, F, G e, por último, H.

3.2 Uniform-Cost Search

O algoritmo UCS funciona da seguinte maneira: Ele realiza uma busca seguindo a ordem de menor custo, sendo esse custo algum parâmetro que seja consistente com o problema. Para o 8-puzzle o custo utilizado foi a profundidade do nó na árvore, uma vez que o objetivo é que encontremos a solução em menor número de passos possível.

Com esse parâmetro em mente, para esse caso o UCS age basicamente como uma busca em profundidade, já que, a escolha de que nó explorar é feita baseada naquele com menor custo na fila, e dessa forma, mesmo que um estado que já está na fila seja encontrado enquanto um outro nó é explorado, o nó que já está com ele na fila de exploração vai sempre ter uma profundidade menor ou igual daquele recém-explorado, portanto, a ordem da fila de prioridades nunca é alterada.

3.3 Iterative Deepening Search

Este algoritmo basicamente realiza sucessivas buscas em profundidade até um limite de profundidade que aumenta a cada iteração até que encontre o estado ótimo de solução. Note que, caso a solução não exista, o Iterative Deepening Search nunca termina sua execução, já que este não guarda os estados já explorados entre uma iteração e outra.

No caso da mesma árvore citada no BFS a ordem de exploração do IDS seria:

1ª Iteração - A
2ª Iteração - A, B, C
3ª Iteração - A, B, D, E, C, F, G e H

3.4 A*

O algoritmo A* trata de um algoritmo que utiliza de uma heurística durante a exploração da árvore com o objetivo de encontrar mais rapidamente a solução ótima para o problema. O que ele faz é explorar, dentre os nós da fronteira, aquele que tenha um menor valor da heurística utilizada até que ele encontre a solução.

Para esse trabalho a heurística utilizada para o algoritmo A* foi a de soma da distância de manhattan de todas as peças do tabuleiro até sua posição na solução. Dessa

forma, a cada iteração exploramos o nó com menor valor f(h) na fronteira, sendo f(h) definida neste caso como:

$f(h) = p(nó) + \Sigma\, manhattan\ dist(tabuleiro[i,j])$

para todo i e j >= 0 e < 3, onde p(nó) é a profundidade do nó explorado.

Essa heurística é admissível porque como escolhemos para expansão em cada passo aquele nó com a menor distância de Manhattan acumulada do objetivo sempre nos aproximamos do estado solução.

3.5 Greedy Best-First Search

O GBFS também é um algoritmo que necessita de uma heurística para sua execução, porém no caso dele utilizamos a heurística de número de peças no tabuleiro que estão fora do local correto delas. Dessa forma, nesse caso nosso f(h) é definido por:

$f(h) = p(nó) + num\ peças\ fora\ dist\ (tabuleiro)$

onde p(nó) é a profundidade do nó explorado.

Porém, diferentemente do A* o que o GBFS faz, se assemelha ao UCS, porém definimos o nó que será explorado a cada iteração baseado no menor valor de f (h), e também trocamos nós na fronteira pelo mesmo estado encontrado depois caso o mesmo estado encontrado depois tenha um f(h) menor do que o que está na fronteira.

A heurística é admissível pois sempre expandimos um nó de forma que tenhamos mais peças na posição correta comparando o tabuleiro com o estado solução.

3.6 Hill-Climbing com Passos Laterais

Outro algoritmo que teve sua implementação requisitada para o trabalho foi o Hill-Climbing com passos laterais, porém após várias tentativas de valores para o número de passos laterais e de implementação do algoritmo o autor deste trabalho não conseguiu com que ele fugisse de mínimos locais, retornando em todos os casos uma solução diferente do estado objetivo, e portanto, ele não será considerado para as análises descritas a seguir.

4. Análise do Desempenho dos Algoritmos

Podemos ver na tabela abaixo como foi o desempenho dos algoritmos para alguns casos de entrada teste.

Algoritmo	**Resposta Ótima**	**Tempo (em segundos)**
BFS	**1**	**0**
UCS	**1**	**0**
A*	**1**	**0.0009992122650146484**
GBFS	**1**	**0**
BFS	**2**	**0**

UCS	**2**	**0**
A*	**2**	**0**
GBFS	**2**	**0**
BFS	**3**	**0**
UCS	**3**	**0.0010001659393310547**
A*	**3**	**0**
GBFS	**3**	**0**
BFS	**10**	**0.21918511390686035**
UCS	**10**	**0.6942951679229736**
A*	**10**	**0.0019996166229248047**
GBFS	**10**	**0.0019986629486083984**
BFS	**15**	**35.00364303588867**
UCS	**15**	**120.69066572189331**
A*	**15**	**0.017109155654907227**
GBFS	**15**	**0.016017436981201172**
BFS	**20**	**1364.2334468364716**
UCS	**20**	**3292.563317298889**
A*	**20**	**0.1340928077697754**
GBFS	**20**	**0.13670587539672852**
A*	**30**	**903.10817527771**
GBFS	**30**	**741.8688478469849**

Para o caso em que a solução ótima exige 30 passos, a execução dos algoritmos de busca sem informação foi feita, porém, após mais de 4 horas executando e não encontrando a solução o programa foi interrompido, e portanto, não há dados concretos de quanto tempo estes algoritmos demoraram para esses casos.

5. Conclusões

Com a análise dos resultados colhidos pelo trabalho podemos chegar a conclusão de que, sempre que reduzimos o espaço de busca de alguma forma inteligente, de maneira que escolhemos o caminho que esteja mais perto de encontrar uma solução, nós reduzimos o tempo de busca por ela. Algoritmos de busca sem informação tendem a demorar mais para

executar, pois eles procuram entre todo o espaço de busca a solução a qual queremos, explorando todos nós até que encontremos ela.

Como para esse problema, existem estados que podem ter até 3 filhos que não foram explorados anteriormente, fica fácil ver o quanto o espaço de busca cresce à medida que precisamos de mais passos para que encontremos uma solução. A busca com informação é mais “esperta” explorando propriedades do problema que podem nos ajudar a encontrar mais rapidamente a solução, diminuindo consideravelmente o tempo de execução principalmente em casos onde o objetivo está longe do estado de origem.

6. Bibliografia

O livro texto da disciplina.
Chaimowicz, L. and Prates, R. (2020). Modelo de documentação de Trabalho Prático da disciplina de Estrutura de Dados.
Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.
Slides da disciplina