UFMG

Trabalho Prático 1

Ordenação usando Busca em Espaço de Estados

Arthur Pontes Nader - 2019022294

1) Introdução

Este trabalho tem como base a aplicação prática da inteligência artificial em uma importante área da computação: ordenação de vetores. A abordagem a ser usada é a busca em espaço de estados, utilizando as estratégias de busca Breadth-first search, Iterative deepening search, Uniform-cost search, A* search e Greedy best-first search.

2) Implementação

a) Tipos de dados

- **classe Estado:** essa classe representa um estado de um problema de busca. Ela possui os seguintes atributos:
  - configuracao: uma lista de inteiros referente a configuração do estado.
  - pai: o estado anterior na árvore de busca.
  - custo_total_avaliacao: custo a ser considerado na hora de avaliar o estado.
  - custo_real: o custo real para se atingir o estado.
  - representacao: uma string que representa a configuração do estado.

  Os seguintes métodos são possíveis:

  - representar_configuracao(): usa do atributo “configuracao” para gerar o atributo “representacao” do estado
  - expandir_estado(modo): gera e retorna uma lista de novos estados possíveis a partir do estado atual, utilizando um determinado modo passado como parâmetro para gerar o atributo “custo_total_avaliacao” dos novos estados. Há três modos possíveis: “real”, “a_estrela” e “guloso”. Os estados filhos são gerados calculando-se todas as permutações válidas a partir do estado atual. Essas permutações são geradas avaliando se o elemento mais à esquerda pode ser trocado por algum dos seguintes. Ao terminar, avalia-se se o próximo elemento com o restante do vetor e assim sucessivamente.

- __lt__(outro_estado): compara o custo total de avaliação do estado atual com o custo total de avaliação de outro estado, retornando True se o custo total de avaliação do estado atual for menor.
- calcular_heuristica(): calcula e retorna uma heurística para o estado atual.
- eh_solucao(): verifica se o estado atual é uma solução para o problema de busca. Retorna True se a configuração atual estiver ordenada em ordem crescente, caso contrário retorna False.

b) Funções

- **bfs(configuracao_inicial):** o BFS implementado apresenta o comportamento de Early Goal Test. Assim, ao explorar um nó, cada um de seus filhos é avaliado como solução do problema, sendo que caso um deles seja, retorna-se a solução imediatamente.
  A função sucessora deste algoritmo é uma fila. Ao explorar um estado, os nós filhos são colocados no final da fila, o que garante que o estado irmão seja explorado antes dos estados filhos, implementando assim o comportamento de explorar em largura.

- **ids(configuracao_inicial):** já o IDS não apresenta o comportamento de Early Goal Test. Outro fato sobre ele é que, mesmo que um estado já tenha sido explorado em uma iteração anterior, o contador global de estados explorados será incrementado se o estado vier a ser explorado de novo na iteração atual.
  A função sucessora consiste numa pilha. Ao explorar um estado, os estados filhos são colocados no topo da pilha, o que garante a exploração em profundidade antes de largura.

- **busca_geral(configuracao_inicial, modo):** essa função recebe um argumento chamado modo, que define qual será a forma de avaliação do custo para definir o próximo estado a ser explorado. Como dito anteriormente, há 3 formas possíveis:
  - “real”: o custo da avaliação será o custo real, ou seja, o algoritmo utilizado é o Uniform-cost search.
  - “a_estrela”: custo da avaliação é a soma do custo real mais o valor da heurística. Como o próprio nome revela, essa é a implementação do A* search.
  - “guloso”: utiliza apenas da heurística para gerar o custo de avaliação, implementando assim o comportamento do Greedy best-first search.

  A função sucessora nesse caso é uma fila de prioridades. Assim, o estado de menor custo de avaliação será o próximo a ser explorado, sendo que os estados sucessivos são colocados na mesma fila de prioridades.

3) Diferenças entre os algoritmos

O Breadth-first search (BFS) é um algoritmo que explora todos os nós em uma camada antes de passar para a próxima camada. Possui alta complexidade de tempo e espaço e não é garantidamente ótimo.

Já o algoritmo Iterative deepening search (IDS) executa uma busca em profundidade com limite cada vez maior até encontrar a solução. A sua principal vantagem é encontrar a solução com complexidade de espaço mais baixa em relação ao algoritmo BFS.

Uniform-cost search (UCS), também conhecido como algoritmo de Dijkstra, é um algoritmo de busca que explora o espaço de estados levando em consideração o custo real de cada ação realizada, escolhendo entre as diversas opções sempre a de menor custo para ser a próxima. Esse algoritmo encontra a solução com o menor custo, desde que o custo das ações seja positivo.

A* search é um algoritmo de busca que utiliza heurísticas para guiar a exploração do espaço de estados, combinando o custo real com uma estimativa do custo do caminho restante até a solução.

Por fim, o Greedy best-first search é um algoritmo de busca heurística que sempre escolhe o próximo nó que parece estar mais próximo da solução, sem levar em consideração o custo de chegar até esse nó. Esse algoritmo pode não encontrar uma solução ótima.

4) Heurística

A heurística implementada é baseada na diferença entre a posição atual de cada elemento no vetor de configuração e sua posição correta no estado objetivo. Quanto maior a diferença, maior será o valor atribuído à heurística, representando uma estimativa maior da distância entre o estado atual e o objetivo.

A função inicia com a inicialização da variável ‘heur’ como 0, que será usada para armazenar o valor da heurística calculada.

Em seguida, um loop for é usado para iterar sobre os elementos do vetor de configuração. Para cada elemento, é verificado se a diferença absoluta entre a posição atual do elemento e sua posição correta é igual a 1. Se for, é adicionado o valor 2 à heurística, indicando que o elemento está deslocado em uma posição adjacente à sua posição correta.

Se a diferença for maior do que 1, significa que o elemento está mais distante de sua posição correta. Nesse caso, é adicionado o valor 4 à heurística, o que reflete um deslocamento maior.

Ao final do loop, a função retorna o valor da heurística calculada.

A heurística calculada por essa função é utilizada em algoritmos de busca informada, como o A* e o Greedy best-first search, para guiar a exploração do espaço de estados de forma mais eficiente, dando prioridade aos estados que têm uma menor estimativa de custo para atingir o objetivo.

Essa heurística não é admissível, o que pode ser exemplificado pela seguinte configuração:

1, 4, 3, 2, 5

Nesse caso, a heurística calcula o valor 8, já que o número 4 e o número 2 estão fora de posições e não são vizinhos de suas posições corretas. Entretanto, o custo real para esse problema é 4, já que basta trocar o 4 e o 2 de posições para se obter a solução.

5) Exemplos

As soluções encontradas a seguir se referem a seguinte configuração inicial:

1 3 5 8 4 7 2 6

BFS: encontrou uma solução com custo 24 em 256 estados expandidos. O caminho da solução foi:

1 3 5 8 4 7 2 6
1 2 5 8 4 7 3 6
1 2 4 8 5 7 3 6
1 2 3 8 5 7 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

IDS: já esse algoritmo encontrou uma solução com custo 24 em 799 estados expandidos. O caminho da solução foi o mesmo da BFS.

UCS: encontrou uma solução com custo 14 em 278 estados expandidos. O seguinte caminho foi obtido:

1 3 5 8 4 7 2 6
1 3 5 2 4 7 8 6
1 3 5 2 4 7 6 8
1 3 2 5 4 7 6 8
1 3 2 4 5 7 6 8
1 3 2 4 5 6 7 8
1 2 3 4 5 6 7 8

A*: expandiu 8 estados para encontrar uma solução com custo 16. A seguir, tem-se o caminho da solução obtida:

1 3 5 8 4 7 2 6
1 3 5 6 4 7 2 8
1 3 5 4 6 7 2 8
1 3 5 4 6 2 7 8
1 3 2 4 6 5 7 8
1 2 3 4 6 5 7 8
1 2 3 4 5 6 7 8

Greedy: por fim, o algoritmo guloso encontrou uma solução com custo 20 e precisou de expandir 7 estados. O caminho da solução foi:

1 3 5 8 4 7 2 6
1 3 4 8 5 7 2 6
1 3 4 6 5 7 2 8
1 2 4 6 5 7 3 8
1 2 4 3 5 7 6 8
1 2 3 4 5 7 6 8
1 2 3 4 5 6 7 8

6) Resultados

Seja “n” igual ao tamanho do vetor a ser ordenado. Os seguintes resultados foram obtidos, sendo que o tempo foi medido em segundos:

Algoritmo	Tempo para n = 4	Tempo para n = 8	Tempo para n = 12	Tempo para n = 16
BFS	0.00041	0.0370	21	97
IDS	0.00039	0.0528	69	390
UCS	0.00056	0.0323	49	251
A*	0.00017	0.00079	0.0081	0.0068
Greedy	0.00017	0.00065	0.0046	0.0048

Já a tabela a seguir mostra o número de estados expandidos para os diversos tamanhos de vetores:

Algoritmo	Número de expansões para n = 4	Número de expansões para n = 8	Número de expansões para n = 12	Número de expansões para n = 16
BFS	3	256	64217	351472
IDS	13	799	401372	2850752
UCS	9	278	108656	458197
A*	3	8	19	14
Greedy	3	7	10	12

7) Discussão

Em termos de tempo de execução, podemos observar que o algoritmo A* e o Greedy foram os mais eficientes, com tempos muito baixos em comparação com os demais mesmo para tamanhos maiores de vetores. Já os algoritmos BFS, IDS e UCS tiveram tempos consideravelmente mais altos, especialmente quando o tamanho do vetor aumentou.

A análise do número de estados expandidos mostra que a BFS, IDS e UCS tiveram um comportamento similar, com um crescimento exponencial no número de estados expandidos à medida que o tamanho do vetor aumentou. Por outro lado, os algoritmos A* e Greedy mostraram um desempenho muito melhor em termos de número de expansões, com valores muito menores mesmo para vetores maiores.

Como a heurística não é admissível, o único algoritmo ótimo implementado foi o UCS. Portanto, para o problema de ordenação de vetores, os algoritmos A* e Greedy implementados se destacam como opções eficientes, enquanto UCS se destaca como opção que garante a otimalidade. BFS e IDS podem ser menos adequadas para tamanhos maiores de vetores devido ao crescimento exponencial do número de expansões.

A implementação da solução para o problema de ordenação proposto foi uma boa maneira de se fixar os conceitos de busca abordados durante a disciplina.