Estratégias de Busca no Jogo dos 15

*Trabalho desenvolvido por:*

Ekaterina Aksenova 202007202  
João Reis 202005824  
Tomás Fontes 202107382

1. Introdução

Um problema de busca é caracterizado por:

* um conjunto de estados
* um estado inicial
* um estado final/objetivo
* função sucessor que mapeia um estado a um conjunto de novos estados
* uma árvore de procura para representar os estados
* nós da árvore, definidos por:
  + estado atual
  + nó pai
  + regra aplicada para gerar o nó
  + profundidade (número de nós atravessados para chegar a este)
  + custo do caminho desde o nó raiz

Métodos utilizados para resolver problemas de procura:

* Não Informadas
  + DFS (Busca em profundidade)
  + BFS (Busca em largura)
  + IDFS (Busca iterativa limitada em profundidade)
  + BCU (Busca custa uniforme)
  + BB (Busca bidirecional)
* Informadas
  + Greedy (Gulosa)
  + A\*
  + IDA\* (busca A\* em profundidade iterativa)
  + RBFS (Recursive-bounded best-first search)

1. Sobre o Jogo dos 15

O jogo dos 15 consiste em, dada uma configuração de números inicial, chegar a uma determinada configuração final. Neste caso, existe um tabuleiro virtual que é composto por 15 células numeradas e uma em branco (numa matriz 4x4, o espaço em branco é representado por 0). Pelas regras do jogo, só são permitidos movimentos da célula em branco para direita, esquerda, para cima ou para baixo. Dado que não é possível voltar para trás após fazer um movimento, em alguns casos, só são possíveis 3, 2, ou 1 movimento, dependendo da posição da célula do 0.

Existem pares de configurações inicial/final para quais não existe solução.

1. Estratégias de Procura

No desenvolvimento do trabalho, foram utilizadas e testadas várias estratégias de procura para encontrar a solução ótima, tendo cada estratégia as suas particularidades e trade-offs.

* 1. Procura não informada
* Busca em profundidade (DFS ou Depth-First Search)

A busca em profundidade expande sempre primeiro o nó mais profundo no nível atual da árvore, ou seja, cada nó é expandido até não ter sucessores. Depois de serem expandidos, esses nós são excluídos da busca, e o algoritmo procura o nó seguinte que seja mais profundo e que ainda tenha sucessores não explorados. Assim, o DFS explora sempre em profundidade até não conseguir gerar mais nós ou chegar a um limite de profundidade dado pelo utilizador, o que garante *backtracking* para os nós mais acima.

As aplicações deste tipo de procura incluem a deteção de ciclos num grafo e encontrar um caminho entre dois nós.

A complexidade temporal do DFS O(|V|+|E|) onde **|V|** é o número de vértices/nós no grafo, e **|E|** o número de arestas. A complexidade espacial é O(h), onde **h** é profundidade máxima da árvore.

* Busca em largura (BFS ou Breadth-First Search)

Neste tipo de busca, o objetivo do algoritmo é: expandir primeiro o nó de raiz, e depois expandir todos os seus sucessores, e assim sucessivamente. Geralmente, os nós filhos são expandidos da esquerda para direita em cada nível de profundidade, antes de atravessar para o nível mais abaixo. (1)

Este algoritmo pode ser usado para determinar a profundidade de cada nó numa dada árvore, encontrar a distância mais curta entre dois nós (quando é garantido que cada aresta tem custo 0 ou 1), entre outras.

A complexidade temporal do BFS O(|V|+|E|) onde **|V|** é o número de vértices/nós no grafo, e **|E|** o número de arestas. A complexidade espacial é O(|V|) visto que, no pior caso, é preciso armazenar todos os nós numa queue.

* Busca iterativa limitada em profundidade (IDFS)

Neste caso, a estratégia de procura é consiste em utilizar DFS em cada nível começando do valor inicial, continuando até ao nível mais profundo da árvore, mas o limite do nível máximo de procura é incrementado com cada iteração, sendo que o algoritmo é restrito de explorar nós para além desta profundidade especificada automaticamente em cada ciclo. O último nível é visitado uma vez, e cada nível acima é visitado mais uma vez mais o nível anterior.

A busca IDFS é utilizada quando não é conhecida a profundidade da solução, e é preciso percorrer um espaço de procura bastante grande. A imposição do limite de profundidade contribui para economizar os recursos de memoria ou espaço.

A complexidade temporal do IDFS é O(bd) onde **b** é o fator de ramificação e **d** é a altura do nó menos profundo. A complexidade espacial é O(d).

* 1. Procura informada com heurística
* O que é uma heurística?

Heurística é designada como uma função que classifica cada decisão de um dado algoritmo com base nas informações disponíveis para determinar qual é a melhor ramificação para seguir. No jogo dos 15 em específico, as heurísticas são estimativas do número de jogadas que falta fazer para chegar de um determinado estado ao estado objetivo.

* Busca Gulosa (ou Greedy)

Busca gulosa tenta resolver o problema fazendo a escolha localmente ótima em cada fase com a esperança de encontrar um ótimo global. Esta abordagem considera apenas o valor heurístico de cada nó, e no jogo dos 15 em particular escolhe sempre o nó com o mínimo valor independentemente de contexto.

* Busca A\*

A\* procura o caminho em um grafo de um vértice inicial até um vértice final. Este algoritmo é muito semelhante à busca gulosa, mas considera tanto a heurística como a profundidade do nó quando faz a escolha. Consiste numa combinação de aproximações heurísticas como do algoritmo Guloso e o Algoritmo de Dijkstra.

1. Descrição de implementação

Para este trabalho escolhemos C como a linguagem a ser utilizada para a implementação das estratégias de busca/procura, porque é uma linguagem adequada com a qual todos os elementos do grupo já trabalharam, o C também proporciona a capacidade de gerir a memória diretamente, o que é benéfico para estes problemas com complexidades espaciais grandes.

Apesar de ter umas particularidades no que diz respeito a erros de segmentação e de atribuição de espaço nos vetores, o C apresenta uma certa simplicidade na criação de funções e na implementação de interatividade.

A representação do tabuleiro no jogo dos 15 vai ser guardada como matrizes 4x4, porque esta estrutura de dados torna a abstração do tabuleiro real mais fácil para nós como programadores. Deste modo, a implementação dos operadores para os movimentos das peças torna-se mais fácil, é uma questão de visitar as posições adjacentes (variando entre 1 e -1 nos valores de i e j).

Primeiro, começámos por implementar uma função para determinar a resolubilidade de um par de estados/tabuleiros, ou seja, um método que retorne 1 se for possível chegar de um estado inicial a um estado final seguindo as regras do jogo dos 15.

Para tal criámos três funções auxiliares: paridade\_zero que, dado uma matriz, retorna 1 se o zero estiver numa linha par e 0 se não; inversões que retorna o número de inversões de um array que é tirado da matriz (convertendo a matriz num array com 16 elementos na mesma ordem), e paridade\_matriz que retorna 1 se a paridade da linha do zero e a paridade do número de inversões forem opostas e 0 se não. Finalmente a função solubilidade retorna 1 se as matrizes tiverem a mesma paridade e 0 se não.

Para este problema precisámos de implementar estruturas de dados apropriadas. Implementámos nodes que funcionam como estruturas que guardam os estados do tabuleiro do jogo, juntamente com a profundidade do nó na árvore abstrata, o valor heurístico do tabuleiro, um apontador para o node de qual foi expandido e o movimento que gerou o tabuleiro deste node. Também foi necessário criar implementações simples de estruturas de dados: queue (FIFO), stack (LIFO) e minHeap.

Definimos as heurísticas h1 e h2 como descritas nas aulas, em que h1 retorna o número de peças fora do sítio e h2 retorna o somatório das distâncias de Manhattan de todas as peças. Ambas estas heurísticas não contam o espaço em branco como uma peça, se o fizessem, não seriam admissíveis.

Fizemos as buscas não informadas BFS e DFS, que são idênticas em tudo exceto na estrutura de dados em que são guardados os nós a expandir, o algoritmo BFS usa uma queue e o DFS usa uma stack. O DFS também precisa de um campo extra para limitar a profundidade até qual explora, se não a probabilidade de ficar sem espaço na pilha antes de chegar a uma solução é muito alta.

O IDFS é um algoritmo que parte do DFS e faz a procura para todos os valores de profundidade máxima até encontrar uma solução. No final de cada iteração sobre o valor de profundidade, este algoritmo apaga completamente o conteúdo da stack, para voltar a ser utilizada.

As estratégias informadas são semelhantes entre si, nós implementámos quatro funções diferentes. A abordagem gulosa utiliza uma minHeap para ordenar a exploração dos nodes pelo seu valor heurístico, implementámos dois algoritmos gulosos, um para cada heurística. A abordagem A\* funciona de uma maneira muito semelhante à gulosa, mas em vez de atribuir o valor heurístico de cada node somente a partir da função heurística, soma-lhe a profundidade desse mesmo node, também implementámos um algoritmo A\* para cada heurística.

Com os algoritmos de busca implementados, era necessário criar uma função para apresentar a solução, chamámos-lhe de pathToSolution. Este método recebe um node v e imprime a sua profundidade (número de jogadas desde o tabuleiro inicial) e a lista de jogadas necessárias para lá chegar. Este método é chamado por todos os algoritmos de busca só quando v é o estado final, ou seja, quando se encontrou a solução.

Finalmente, para ajudar com a comparação dos métodos de busca, criámos um contador de ticks do relógio que é convertido em segundos para ser convertido em segundos no final da execução da busca, um contador de nodes explorados e um sistema de menus que permite testar todos os algoritmos para a mesma configuração sem ter de a reescrever.

1. Resultados

C1

1 2 3 4 9 5 7 8 13 6 10 12 0 14 11 15

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Estratégia | Tempo (segundos) | Nós visitados | Encontrou a solução? | Profundidade/Custo |
| BFS | 0.006000 | 228 | Sim | 7 |
| DFS(7) | 0.002000 | 378 | Sim | 7 |
| DFS(18) | 0.604000 | 357795 | Sim | 17 |
| IDFS | 0.006000 | 745 | Sim | 7 |
| Gulosa h1 | 0.004000 | 8 | Sim | 7 |
| Gulosa h2 | 0.002000 | 10 | Sim | 7 |
| A\* h1 | 0.006000 | 8 | Sim | 7 |
| A\* h2 | 0.005000 | 10 | Sim | 7 |

C2

9 12 13 7 0 14 5 2 6 1 4 8 10 15 3 11

9 5 12 7 14 13 0 8 1 3 2 4 6 10 15 11

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Estratégia | Tempo (segundos) | Nós visitados | Encontrou a solução? | Profundidade/Custo |
| BFS | 1.778000 | 458198 | Sim | 16 |
| DFS(16) | 0.017000 | 4825 | Sim | 16 |
| DFS(18) | 0.041000 | 21950 | Sim | 16 |
| IDFS | 0.824000 | 447160 | Sim | 16 |
| Gulosa h1 | 0.009000 | 29 | Sim | 28 |
| Gulosa h2 | 0.000000 | 50 | Sim | 16 |
| A\* h1 | 0.003000 | 114 | Sim | 16 |
| A\* h2 | 0.021000 | 50 | Sim | 16 |

C3

6 12 5 9 14 2 4 11 0 7 8 13 3 10 1 15

14 6 12 9 7 2 5 11 8 4 13 15 3 10 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Estratégia | Tempo (segundos) | Nós visitados | Encontrou a solução? | Profundidade/Custo |
| BFS | 0.080000 | 22036 | Sim | 12 |
| DFS(12) | 0.012000 | 4202 | Sim | 12 |
| DFS(18) | 0.109000 | 39974 | Sim | 16 |
| IDFS | 0.056000 | 25665 | Sim | 12 |
| Gulosa h1 | 0.039000 | 37 | Sim | 12 |
| Gulosa h2 | 0.000000 | 18 | Sim | 12 |
| A\* h1 | 0.006000 | 22 | Sim | 12 |
| A\* h2 | 0.004000 | 18 | Sim | 12 |

Nota: 1) Os valores entre parênteses correspondem aos limites de profundidade utilizados para os testes do DFS

6. Comentários Finais e Conclusões

Em suma os algoritmos BFS e DFS são algoritmos simples, por um lado o BFS é útil em problemas com fator de ramificação muito grandes ou em casos com soluções pouco profundas. Por outro lado, o DFS torna-se relevante para casos com soluções profundas ou quando se conhece a profundidade da solução ótima. Sendo assim o BFS é completo e ótimo e o DFS não é ótimo e só é completo se for limitado a uma profundidade igual ou superior a profundidade ótima, porém este fica mais eficiente quanto mais próximo o limite se encontra da profundidade da solução ótima.

O contador dos nós visitados do IDFS é cumulativo, isto é, conta todos os nós avaliados em todos os ciclos, mas a memória real que necessita é igual à do DFS com limite ótimo, logo este algoritmo consegue conservar recursos da máquina. O IDFS é completo e ótimo como o BFS mas muito mais eficiente em termos de espaço de procura.

Em relação às abordagens informadas utilizámos duas heurísticas distintas, das quais a h2 é melhor visto que estima o número de jogadas necessárias para chegar ao estado final com melhor precisão, especialmente para casos com maior espaço de procura.

No que diz respeito a gulosa esta não é completa nem ótima, visto que entra em ciclos por não considerar a profundidade real, mas sim o valor heurístico, em contrapartida A\* é um algoritmo completo e ótimo.

Sendo assim, as abordagens informadas são a melhor estratégia para a resolução deste problema, destacando-se a A\* por garantir uma solução ótima.

7. Referências Bibliográficas

<https://www.javatpoint.com/search-algorithms-in-ai>

<https://zoo.cs.yale.edu/classes/cs470/materials/aima2010.pdf>

<https://horatiuvlad.com/unitbv/inteligenta_artificiala/2015/TilesSolvability.html>

<https://personal.math.ubc.ca/~cass/courses/m308-02b/projects/grant/fifteen.html>

<https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/>

<https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/>

<https://www.educative.io/edpresso/what-is-iterative-deepening-search>