U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Inteligência Artificial

Primeiro Trabalho: Pesquisas

24/03/2022

Trabalho realizado por:

João Marrucho - 201804960

Bruno Dias - 201907828

Pedro Leite - 201906697

Índice

1. Introdução

Um problema de busca/procura é definido por um conjunto de estados, um estado inicial, um estado final, uma função sucessor (mapeia um estado num conjunto de novos estados), uma representação do espaço de estados e cada nó é uma estrutura com pelo menos 5 componentes: estado, nó pai, jogada/regra aplicada para gerar o nó, número de nós no caminho para este nó (profundidade do nó na árvore) e custo do caminho desde o nó raiz. A resolução desses problemas baseiam-se em algoritmos de pesquisa que exploram caminhos de forma incremental a partir dos nós no início. Como produto da pesquisa, o algoritmo expande-se para os nós inexplorados até que um nó final seja encontrado. A maneira em que esse algoritmo se expande é definida pela sua estratégia de pesquisa.

2. Jogo dos 15

O jogo dos 15 é um quebra-cabeças composto por 15 peças e um espaço vazio para que se possa movimentar as peças, num tabuleiro 4x4. As peças devem ser ordenadas numa determinada ordem, deslocando-se as peças ocupando o espaço vazio, só as peças adjacentes ao espaço vazio é que podem ocupar o seu espaço (seja para cima, para baixo, para esquerda ou para a direita, não se pode fazer movimentações diagonais). Este será o jogo testado no trabalho.

3. Procura não guiada

Uma estratégia de pesquisa é dita "cega" se ela não leva em conta informações específicas sobre o problema a ser resolvido. Existem diversas estratégias cegas para a construção e pesquisa numa árvore, entre as quais a Pesquisa em Profundidade e a Pesquisa em Largura.

Pesquisa em Profundidade procura explorar completamente o ramo mais à esquerda da árvore antes de tentar o ramo vizinho. Dependendo do problema e da maneira como é implementada, pode necessitar de pouca memória e ser eficiente para problemas com muitas soluções. Em alguns casos, é definida uma profundidade limite, que vai aumentando gradualmente, chamando-se **Pesquisa Iterativa Limitada em Profundidade**. O algoritmo de pesquisa em

profundidade não encontra necessariamente a solução mais próxima, mas pode ser mais eficiente se o problema possuir um grande número de soluções ou se a maioria dos caminhos levar a uma solução. A complexidade temporal é O(B^M) e a complexidade espacial é O(B*M), onde B é o número médio de sucessores de cada nível e M a profundidade máxima da árvore.

Pesquisa em Largura consiste em construir uma árvore de estados a partir do estado inicial, aplicando a cada momento, todas as regras possíveis aos estados do nível mais baixo, gerando todos os estados sucessores de cada um destes estados. Assim, cada nível da árvore é completamente construído antes que qualquer nó do próximo nível seja adicionado à árvore. Todos os nós de menor profundidade são expandidos primeiro. Possui uma pesquisa muito sistemática e normalmente demora muito tempo e ocupa muito espaço. A principal vantagem do algoritmo de pesquisa em largura é que este encontra o menor caminho do nó inicial até ao nó final mais próximo. A complexidade temporal e espacial é O(B^D), onde B é o número médio de sucessores de cada nível e D a profundidade da solução.

4. Procura guiada

Procura que utiliza características próprias do problema particular para ajudar no processo de pesquisa. Uma heurística será então uma estimação adequada do custo do passo desde um estado até um objetivo. Uma **heurística** será então uma estimação adequada do custo do passo desde um estado até um objetivo.

Pesquisa Gulosa com Heurística procura tentar minimizar o custo estimado para chegar à solução, ao seguir o caminho imediato que de acordo com a heurística tem menor custo.

Pesquisa A* procura tentar minimizar o custo total do caminho, uma pesquisa de custo uniforme para minimizar o custo do caminho da raiz até o nó corrente. Isto é, tem em conta tanto os custos dos nós anteriores, com o suposto custo para chegar ao resultado final. Ao contrário da gulosa que só vê o custo mais próximo para chegar ao resultado do nó em que se encontra, está pode voltar

para nós com nível de profundidade inferior. Permite chegar sempre à solução ótima.

5. Implementação

A linguagem utilizada no nosso trabalho foi **Python** devido à nossa familiarização com essa linguagem e a sua adequação à nossa estratégia de programação. Python possui uma grande variedade de bibliotecas e funções que acabam por nos auxiliar neste trabalho tal como a sua facilidade de escrita.

Escolhemos **vetores** para representar as configurações fornecidas, pois possuem uma boa portabilidade, baixo custo de memória e são fáceis de manipular.

Foi necessário a criação de uma classe "node", para inserir os vetores com as configurações para depois fazer cada pesquisa. Esta classe contém o vetor, a profundidade na árvore, o caminho até esse nó, uma função que retorna o seu nome, para nas funções de pesquisa sabermos que nós já foram visitados, e outra função que imprime o nó e as suas características.

Optamos por separar o código em diferentes ficheiros: um ficheiro para designar as funções auxiliares, um ficheiro para designar os nós, um ficheiro *main* e os restantes ficheiros para designar os diferentes tipos de pesquisa.

6. Funções Auxiliares

Para determinar se é possível chegar da matriz inicial à matriz final, definimos uma função "solvable". Que para ambas as matrizes, conta o número de inversões (uma inversão é o número de células à frente da célula atual que são maiores que a atual) e a paridade da fila onde está o "0" (de baixo para cima, as filas estão enumeradas de 1 a 4). É possível chegar à solução se satisfazer a condição:
((inv1%2==0)==(branco1%2==1))==((inv2%2==0)==(branco2%2==1)).

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

Configuração Inicial 1

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

Configuração Inicial 2

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

Configuração Final

Na configuração inicial 1, o “0” está numa fila par e possui um número ímpar de inversões. Sendo a paridade da Configuração Inicial diferente tal como na Configuração. Final, a função “solvable” determina que é possível resolver.

Na configuração inicial 2, o “0” está numa fila par e possui um número par de inversões, sendo a paridade da Config. Inicial igual e na Configuração Final diferente, a função “solvable” determina que não é possível resolver.

Temos uma função “can_move”, que consoante a posição do 0 retorna uma string com as diferentes movimentações que o 0 pode fazer (“C” = cima, “B” = baixo, “E” = esquerda e “D” = direita). E outra função chamada “sucessores”, que ao receber o vetor do nó a ser explorado no momento e um caracter com o movimento que se pretende realizar, proveniente da string recebida pela função "can_move", retorna um vetor com esse movimento feito para depois ser acrescentado a um nó.

7. Implementação da Pesquisa em Profundidade

Para a implementação da Pesquisa em Profundidade (função “dfs”), decidimos utilizar um vetor de forma a simular o funcionamento de uma stack (pilha). Para evitar visitar um nó mais que uma vez, temos um set de strings que representam os vetores dos nós já visitados. Começamos com um nó na pilha, nó inicial com a matriz de input, e entramos num ciclo. Nesse ciclo retira-se sempre o último elemento colocado na pilha (pop) e verifica-se se é solução. Caso seja solução, o vetor igual à configuração final devolve o percurso percorrido até atingir esse nível. Caso contrário continua o ciclo onde é adicionada a string ao set dos "visited", e com o uso das funções "can_move" e "sucessores" descobre-se as características dos nós sucessores ao atual que irão ser adicionados ao final da pilha (push). No final disto volta-se a repetir o ciclo até a pilha estar vazia, ou,

devido a este algoritmo ser muito ineficiente para saber a solubilidade deste jogo, chegar a um limite de profundidade 20 onde deixa de ser possível desenvolver nós que passem para além desse limite (limite adequado para os casos testados que são nós que não atingem grandes profundidades).

8. Implementação da Pesquisa em Profundidade Iterativa

A implementação da Pesquisa em Profundidade Iterativa (função "idfs") é semelhante à Pesquisa em Profundidade, porém tem um ciclo que vai aumentando gradualmente o limite de profundidade máximo até onde os nós se podem desenvolver. Este limite começa em 0, apenas com a configuração inicial como nó, e vai aumentando um nível de cada vez que a pilha fica vazia. Ao contrário da Pesquisa em Profundidade, não foi imposto um limite máximo para a profundidade visto que o algoritmo verifica sempre se há solução num certo nível, antes de recomeçar com um nível de profundidade máxima superior, de modo a não ficar demasiado tempo a correr para casos de teste pequenos.

9. Implementação da Pesquisa em Largura

Para a implementação da Pesquisa em Largura (função "bfs"), decidimos fazer com que um vetor simulasse o comportamento de queues (filas). A base do algoritmo que foi aplicado é a mesma que na Pesquisa em Profundidade, mudando o comportamento de forma a quando se retirar um elemento, retirar do inicio da fila e não do fim que é o caso de uma stack. Ao contrário da Pesquisa em Profundidade, este algoritmo percorre todos os nós de uma profundidade antes de aumentar o valor desta, não sendo por isso necessário limitá-la.

10. Implementação da Pesquisa Gulosa com Heurística e da Pesquisa A*

Para a implementação da Gulosa com Heurística e do A*, utilizamos duas heurísticas. O somatório do número de peças fora do lugar, na função "outofrorder". E a Manhattan Distance, que consiste no somatório das distâncias de cada peça à configuração final, essa distância é a soma entre as

diferenças das coordenadas (x1, y1) e (x2, y2), ou seja, somatório de |x1-x2| + |y1-y2|, neste caso (x1, x2), são as coordenadas da célula do nó atual e (y1,y2) as a célula do nó destino, e o valor em (x1, x2) e (y1, y2), esta heurística está implementada na função "manhattan".

Para a **implementação da Gulosa com Heurística** (função "gbfs"), temos em conta o custo, consoante cada heurística. Este custo vai definir a nossa escolha, já que escolhemos sempre o nó sucessor com menor custo. Vamos utilizar heaps, onde o custo do pai vai ser sempre maior que a dos filhos. Para evitar visitar um nó mais que uma vez, temos um set de strings que representam os vetores dos nós já visitados. Começamos com um nó, nó inicial com a matriz de input, e entramos num ciclo. Nesse ciclo, caso o nó seja solução, o vetor igual à configuração final devolve o percurso percorrido até atingir esse nível. Caso contrário, continua o ciclo e colocamos o nós sucessores do nó atual na heap (heappush) e retiramos da heap o menor nó (heappop), que vai ser utilizado para o próximo ciclo. A **implementação da A*** (função "astar") é semelhante à Gulosa com Heurística, mas neste caso temos em conta o custo do caminho até ao nó atual (prevcost) e a heurística (cost), o nosso custo total é a soma desses dois elementos (prevcost + cost), sendo preciso ir guardando os caminhos, logo nunca limpamos a heap.

11. Resultados

Estratégia	Tempo (segundos)	Espaço	Encontrou a solução?	Profundidade/ Custo
DFS*	2.536	129477	Sim*	20*
BFS	1.247	55829	Sim	12
IDFS	0.767	4693	Sim	12
Gulosa (Manhattan)	0.409	53	Sim	12
Gulosa (Out of order)	0.424	1530	Sim	70
A* (Manhattan)	0.610	849	Sim	12
Gulosa (Out of order)	0.463	1404	Sim	12

Tabela de comparação do tempo, quantidade de espaço gasto, complete e otimalidade de cada algoritmo utilizado.

*limitado à profundidade máxima de 20

12. Conclusão

Dentro dos algoritmos de pesquisa não informada, de acordo com o dados da tabela, a Pesquisa em Profundidade acabou por ser o pior dos algoritmos de pesquisa não informada, não sendo uma opção viável para resolver este problema. Verificamos que caso não colocássemos um limite máximo de profundidade o programa iria correr durante muito mais tempo, sem encontrar solução num espaço de tempo aceitável, devido ao facto de a ordem dos movimentos possíveis não ser aleatória e dá prioridade a movimentos para a esquerda e direita quando possível. Ao contrário da Pesquisa em Largura e da Pesquisa em Profundidade Iterativa, a Pesquisa em Profundidade acabou por não encontrar uma solução ótima tendo 20 de profundidade (nível máximo que permitimos no nosso programa),

Comparando os algoritmos de pesquisa informada e nas informada, é possível verificar que os algoritmos de pesquisa informada no geral têm melhor desempenho em termos de memória e tempo que demoram a chegar à solução.

Dentro dos algoritmos de pesquisa informada temos o A* e o Guloso com Heurística. No caso de teste usado, a Pesquisa Gulosa com Heurística acabou por ter um comportamento melhor que o A*, mas após realizarmos mais testes, verificamos que com casos maiores o A* acaba por conseguir chegar sempre a uma solução ótima o que pode não acontece com o Guloso com Heurística. Para além disso, existem duas variantes de cada algoritmo, nas quais usamos heurísticas diferentes. Numa usamos a de Manhattan e na outra o Out of Order, das duas a de Manhattan foi a que se distinguiu pela positiva, conseguindo obter um custo em memória muito inferior do que usando a Out of Order. Além disso usando a pesquisa gulosa, acabou por chegar a uma solução longe de ser ótima. Com isto dito, justificamos por tanto que a Heurística de Manhattan é a que escolhemos para para ambos os algoritmos de pesquisa. Concluímos que a pesquisa mais apropriada para este jogo, é a Pesquisa A* com a Heurística Manhattan, já que acaba por chegar sempre a uma solução ótima , mesmo que o algoritmo aumente muito em tamanho de nós, mantém um baixo custo de memória e chega rapidamente à solução.

13. Referências Bibliográficas

https://www.gsigma.ufsc.br/~popov/aulas/ia/modulo3/index.html

https://en.wikipedia.org/wiki/Heuristic_(computer_science)

S. Russell, P. Norvig; Artificial Intelligence: A Modern Approach, 3rd ed, Prentice Hall, 2009

Slides da unidade curricular