IART 2020/21
Relatório Final do 1ºProjeto
Aquarium Puzzle - One Player Solitaire Games

Elementos da Turma 4 Grupo 11:
José Rodrigues - up201708806
Marina Dias - up201806787
Pedro Vale - up201806083

Abril 2021

Especificação do projeto

O jogo cuja solução vamos explorar chama-se Aquarium, e consiste num puzzle NxN, dividido em blocos (aquários), em que o objetivo é preenchê-los, tendo em consideração as regras específicas do jogo, que se baseiam na gravidade (da água), e cumprindo o número de células preenchidas em cada coluna/linha.

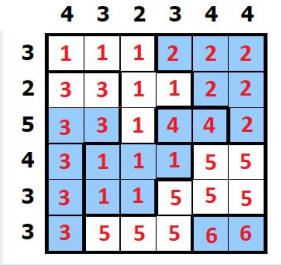

Figura 1

Formulação do problema

Representação do estado: Para representar o estado, temos uma classe **Board** que tem vários atributos, mas aqueles essenciais à representação do estado são 4, o *rowsRules* (uma lista de números com as restrições das linhas), o *colsRules* (uma lista de números com as restrições das colunas), o *pools* (uma lista de listas, que é uma lista de colunas, percorrendo o puzzle da esquerda para a direita, de cima para baixo, que a cada espaço atribui o número da piscina a que pertence), e o *currentState* (uma lista como a das *pools* mas que atribui a cada espaço 0 se tiver vazio ou 1 se tiver preenchido).

Exemplo:

rowsRules: [4,3,2,3,4,4]

colsRules: [3,2,5,4,3,3]

pools: [1, 1, 1, 2, 2, 2, 3, 3, 1, 1, 2, 2, 3, 3, 1, 4, 4, 2, 3, 1, 1, 1, 5, 5, 3, 1, 1, 5, 5, 5, 3, 5, 5, 5, 6, 6]

currentState: [0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1 ,1 ,1 ,1 , 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1]] (ver figura 1)

Estado inicial: O estado inicial de um puzzle é um estado em que a última lista está preenchida unicamente com zeros.

Teste Objetivo: Verificação de se o valor pré definido de posições preenchidas por cada linha e por cada coluna, está cumprido. Também é obrigatório que **dentro de cada aquário**, o nível da água seja igual e estar cheio até à base do aquário, ou seja, dentro de um mesmo bloco (aquário) não existem posições vazias debaixo ou ao lado de posições preenchidas, mas essa condição é automaticamente aplicada a cada jogada.

Formulação do problema

Operações:

- Preencher uma posição;

Pré-Condições: O jogo não impede qualquer jogada, pelo que não existem pré-condições de jogada.

Efeitos: Ao preencher uma posição, o jogo vai efetuar essa mudança no estado de jogo mas também vai verificar se alguma restrição sobre as posições preenchidas por linha e/ou coluna está a ser quebrada, e se sim, avisa o jogador através da mudança de cor da restrição quebrada.

Custo: Cada operação de preencher uma posição tem custo 1. Para efeitos de pesquisa são apenas contabilizados o número de jogadas.

Heurística: Nos métodos de pesquisa informada, no método *Greedy* utilizamos como função de avaliação, a soma entre a profundidade do nó e o número de restrições do *board* que ainda faltam cumprir. No método *A**, utilizamos como função de avaliação, a soma entre a profundidade do nó, o número de restrições do *board* e o número de blocos preenchidos pelo nó em avaliação.

Implementação

O nosso projeto está implementado em Java, e faz uso da biblioteca Lanterna para produzir uma user interface que permite e facilita a interação do utilizador com o mesmo.

Para nos organizarmos e trabalharmos em conjunto no projeto, usamos um repositório GitHub.

Dividimos o projeto em várias classes, em que as mais importantes são:

- Game - Esta classe é onde o jogo em si é controlado, inicializando e sendo a linha de comunicação entre todas as partes componentes do jogo, como a Search, o Board e o UI.
- Search - Onde se encontram os algoritmos de pesquisa, usados para resolver os puzzles.
- Board - Onde de encontram os atributos que constituem cada puzzle, funções que o alteram e funções que tratam do seu display.
- UI - Classe onde é definida o UI que inicia o jogo, pedindo ao utilizador as características do jogo que quer jogar.

Algoritmos de Pesquisa Implementados

Métodos de Pesquisa Cega:

- Pesquisa em Largura
- Pesquisa em Profundidade
- Pesquisa em Profundidade Iterativa

Métodos de Pesquisa Informada:

- Algoritmo A*
- Algoritmo Guloso

Abordagem aplicada

De um modo geral a abordagem entre os métodos é parecida. Começamos por, tendo um estado de jogo, obter os nós possíveis de serem explorados a seguir, esses nós são a linha mais inferior ainda por preencher de cada piscina.

A seguir o programa percorre essa lista, criando um estado de jogo que advém da aplicação de cada um desses nó, e avaliando se está a quebrar alguma restrição ou se o nó já foi visitado. Se sim, ignora o nó, senão, vai explorar esse nó, colocando o numa *Stack*, repetindo o ciclo.

O que varia entre os métodos é o tipo de lista em que os nós se encontram (o BFS utiliza uma *Queue*, os outros utilizam uma *ArrayList*), e a ordem em que os nós se apresentam dentro desta mesma lista.

Nos métodos de pesquisa informada, no método *Greedy* utilizamos como função de avaliação, a soma entre a profundidade do nó e o número de restrições do *board* que ainda faltam cumprir. No método *A**, utilizamos como função de avaliação, a soma entre a profundidade do nó, o número de restrições do *board* e o número de blocos preenchidos pelo nó em avaliação.

Resultados Experimentais

Puzzle	Algoritmos de pesquisa					Performance
	A*	Greedy	Iterative Deepening	Breadth first	Depth first	
6x6 #1	0.143	0.149	0.5	0.240	0.32	Tempo médio (s)
	219	237	168	1271	26	Steps
6x6 #2	0.85	0.113	0.88	0.729	0.46	Tempo médio (s)
	15	22	319	3729	40	Steps
6x6 #3	0.106	0.117	0.845	1.364	0.534	Tempo médio (s)
	58	59	18219	14733	7984	Steps
6x6 #4	0.621	0.641	0.459	1.271	0.248	Tempo médio (s)
	1129	1362	4849	9076	1466	Steps
6x6 #5	0.7	0.71	0.139	0.375	0.92	Tempo médio (s)
	10	11	1345	2023	406	Steps
10x10 #1	0.157	0.151	8.348	16.7	3.203	Tempo médio (s)
	18	30	274601	199053	96387	Steps
10x10 #2	0.218	0.242	Memória Insuficiente	Memória Insuficiente	Memória Insuficiente	Tempo médio (s)
	39	47				Steps
10x10 #3	0.292	0.268	9.244	Memória Insuficiente	2.384	Tempo médio (s)
	173	201	276170		70968	Steps
10x10 #4	13.331	13.338	Memória Insuficiente	Memória Insuficiente	Memória Insuficiente	Tempo médio (s)
	60562	61641				Steps
10x10 #5	0.133	0.140	1.611	40.741	0.74	Tempo médio (s)
	14	20	36090	482725	11342	Steps

Obtivemos estes resultados utilizando os variados algoritmos de pesquisa em 5 puzzles 6x6 e 5 puzzles 10x10, correndo o algoritmo 3 vezes por puzzle e fazendo a média dos tempos obtidos.

Nos casos em que o algoritmo ultrapassa os 2 milhões de passos o programa arrisca se a ficar sem memória, pelo que interrompemos o algoritmo quando isso acontece, estando esses casos registados na tabela como “Memória Insuficiente”.

Conclusão

- Após obter estes resultados podemos rapidamente concluir que os métodos de pesquisa informada são superiores em termos de performance aos métodos de pesquisa às cegas.
- De todos os algoritmos o A* foi o melhor a nível de performance, ou seja, tendo em conta tanto os passos necessários (memória gasta) como o tempo utilizado. No entanto o algoritmo de Busca em Profundidade destaca-se dentro dos de Pesquisa Cega pelo seu número de passos ser bastante menor em relação aos outros algoritmos.
- Tivemos alguma dificuldade em adaptar a resolução do puzzle aos algoritmos.
- O projeto ajudou a aprofundar e a pôr em prática o conhecimento dos conteúdos discutidos durante as aulas.

Referências

Para nos auxiliar neste projeto, fizemos uma pesquisa de forma a percebermos melhor o funcionamento do jogo e possíveis implementações de soluções que nos permitissem obter algumas ideias para a implementação das nossas próprias estratégias.
Essas referências são mencionadas abaixo:

- Aquarium Tutorial https://youtu.be/NaIMr3_sw4 / https://youtu.be/dm0ZqE1U6Qw / https://youtu.be/H4wzIH07R4U
- Resolução do Aquarium utilizando Métodos de Pesquisa escritos em Perl https://github.polettix.it/ETOOBUSY/2020/03/30/aquarium-parse-puzzle/
- Slides das Aulas de IART 2020/2021
- Iterative Deepening Search https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/
- A* search algorithm https://en.wikipedia.org/wiki/A*_search_algorithm