FEUP - MIEIC

IART 2019/20

Trabalho realizado por:

Matheus Gonçalves (201405081)

Miguel Pires (201406989)

Ricardo Cardoso (201604686)

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Especificação do ZHED

O **ZHED** é um jogo do tipo solitário que consiste em resolver um puzzle. Os puzzles são quadrículas com casas cinzentas, umas numeradas e outras não, e uma casa objetivo, de cor branca. Para completar cada nível, cada casa numerada tem que ser expandida em uma das quatro direções (cima, direita, baixo, esquerda) e sobreposta para alcançar a casa objetivo.

Cada casa expande ***n*** casas na direção escolhida, diminuindo em um para cada casa vazia. Quando uma casa em expansão se sobrepõe a uma casa já preenchida, o número de casas a serem preenchidas na direção da expansão não diminui.

Problema de pesquisa

- Representação do estado:

ArrayList<ArrayList<char>>

. => espaço em branco

[1..9] => casas selecionáveis

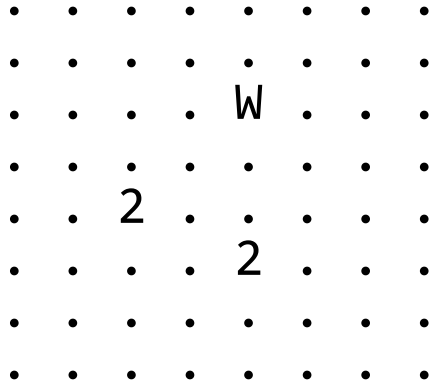
W => casa vencedora

=> casa ocupada (corresponde a uma ação realizada por uma peça)

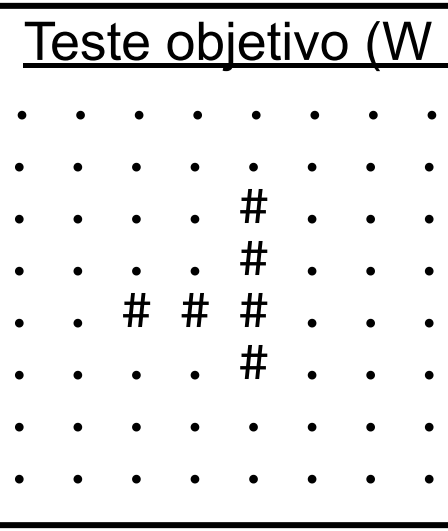
- Operadores:

Esquerda, Direita, Cima, Baixo.

- Estado inicial (exemplo):



- Teste objetivo (W = ●):



Sequencia de ações: Peça (3,5) → Direita, Peça (5,6) → Cima

Arquitetura do projeto

Para a realização do solver com recursos aos algoritmos propostos (BFS, DFS, A* etc…) implementamos um conjunto de classes que visam a encontrar a solução de um certo nível do jogo. As classes mais relevantes do nosso projeto são:

- Game.java - Inicia a interface com o qual o utilizador interage.
- Level.java - Contém a informação do nível a ser jogado bem como a lógica dos movimentos das peças.
- NewNode.java - Classe usada para criar um node inserido nas funções dos algoritmos, contém a seguinte informação:
  - NewNode dad (o nó pai)
  - Level state (o estado do puzzle)
  - Piece lastPiece (peça a ser expandida para originar no nó atual)
  - Direction lastOperator (operador usado para originar o nó atual)
  - int cost (custo até ao nó)
  - int priority (inteiro que define a prioridade na expansão do nó; onde são aplicadas as heurísticas)
- Piece.java - Informação das peças (posição e número de passos)
- SolveSearch.java - Lógica dos algoritmos de pesquisa implementados

Algoritmos de pesquisa implementados

Neste trabalho temos 3 algoritmos de pesquisa cega:

- Pesquisa Primeiro em Largura
- Pesquisa Primeiro em Profundidade
- Aprofundamento Progressivo

```
public NewNode breadthFirstSearch()
```

```
public NewNode depthFirstSearch()
```

```
public NewNode iterativeDeepeningSearch()
```

E 2 algoritmos de pesquisa informada, com diferentes funções heurísticas:

- Pesquisa Gananciosa
- Pesquisa A*

```
public NewNode greedySearch()
```

```
public NewNode AStarSearch()
```

Abordagem aplicada (1/2) - pesquisa cega

Nos algoritmos de pesquisa cega aplicámos a seguinte abordagem:

- Pesquisa em largura:
  É utilizada uma Queue para armazenar os nós uma vez que adota o sistema FIFO (First-in-First-out), ou seja, são expandidos primeiro os nós que estão à mais tempo na fila.

- Pesquisa em profundidade:
  É utilizada uma Stack para armazenar os nós uma vez que adota o sistema LIFO (Last-in-First-out), ou seja, são expandidos primeiro os nós que acabaram de chegar à fila.

- Pesquisa em profundidade iterativa:
  Semelhante à pesquisa em profundidade mas limitando o valor máximo de profundidade dos nós que são adicionados à Stack, de forma iterativa e começando em 1.

Em todos os casos o algoritmo verifica se o nó inicial é a solução do puzzle, caso não seja então explora cada uma das peças que faltam ser expandidas e cria um nó diferente para cada uma das 4 direções que podem ser escolhidas. Após a criação de cada um dos nós verifica se encontrou a solução. Se não tiver encontrado a solução analisa o próximo elemento da fila (Stack ou Queue).

Abordagem aplicada (2/2) - pesquisa informada

Nos algoritmos de pesquisa informada utilizamos uma PriorityQueue para armazenar os nós sendo que tanto no algoritmo de pesquisa gulosa bem como no A* usamos uma variável que define a prioridade de expansão de um nó (utilizada pela PriorityQueue) no decorrer do algoritmo utilizando duas heurísticas diferentes:

- Algoritmo Guloso: `int priority = numExpandedCells - lastPiece.numSteps;`
- Algoritmo A*: `int priority = (numExpandedCells - lastPiece.numSteps) * factor - cost;`

numExpandedCells: retorna o número de células expandidas na direção especificada no nó (>=numSteps);
lastPiece.numSteps: número que indica o valor da peça;
factor: Este fator diminui o peso do custo de um nó na sua prioridade;
cost: custo uniforme associado a expandir o nó, tem um valor igual à profundidade do nó, dando hipótese a nós de profundidades inferiores de serem igualmente expandidos.

Com estas heurísticas, se a expansão de uma peça numa direção for superior ao valor dessa mesma peça (ou seja, se já existirem casas ocupadas nessa direção) esse nó receberá um valor de prioridade maior, uma vez que está num bom caminho para chegar à solução.

Resultados experimentais

Tabela comparativa dos diferentes tempos médios de execução com os diferentes níveis, por algoritmo. Para estes testes corremos o programa 3 vezes para cada par algoritmo/nível, fazendo a média dos tempos de execução, e o nº de nós é sempre o mesmo.

NIVEL	ALGORITMO							PERFORMANCE
	A* com fator			Pesquisa gulosa	Profundidade iterativa	Primeiro em profundidade	Primeiro em largura / custo uniforme	
	3	4	5					
1 (4 peças)	1	1	1	5	25	7	38	Tempo medio (ms)
	195	195	195	1.315	3.655	1.687	3.595	Nr. de nos
2 (5 peças)	91	103	63	23	73	57	100	Tempo medio (ms)
	35.447	39.807	38.183	8.115	64.783	25.343	42.727	Nr. de nos
3 (6 peças)	69	53	38	330	4.054	3.402	2.105	Tempo medio (ms)
	15.728	15.676	12.680	234.492	4.484.656	3.537.976	1.031.392	Nr. de nos
4 (7 peças)	1.348	534	838	2.113	77.679	53.038	memoria insuficiente	Tempo medio (ms)
	492.771	190.391	334.291	1.271.763	74.026.707	47.519.499		Nr. de nos
5 (9 peças)	memoria insuficiente	memoria insuficiente	1.474	memoria insuficiente	mais de 10 min	mais de 10 min	memoria insuficiente	Tempo medio (ms)
			593.609					Nr. de nos

Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz

Conclusões

Após terminar este trabalho podemos concluir os seguintes aspetos:

- Os algoritmos de pesquisa informada são muito mais eficientes do que os algoritmos de pesquisa cega.
- Dentro dos algoritmos de pesquisa cega seria de esperar que o melhor fosse o algoritmo de aprofundamento progressivo. No entanto, como neste puzzle é quase sempre necessário chegar ao último nível da árvore de pesquisa para encontrar uma solução, o facto de fazermos um aprofundamento iterativo faz com que seja menos eficiente do que utilizar a pesquisa primeiro em profundidade. A pesquisa em largura precisa de muita memória em simultâneo enquanto a pesquisa em profundidade consegue libertar os ramos que já foram vistos, ainda que demore mais tempo.
- Os algoritmos de pesquisa informada foram muito mais eficientes, no entanto níveis com um número elevado de peças precisam de uma quantidade muito grande de memória para serem resolvidos mesmo utilizando estes algoritmos (A* e pesquisa gananciosa).
- A variação fator de redução de peso do custo do algoritmo A* cria uma grande variação da eficácia do algoritmo, tal como pode ser visto na tabela anterior, no entanto, após vários testes, consideramos que os melhores fatores variam entre 3 e 5. Um fator inferior a 3 dá demasiado peso ao custo do nó, reduzindo a eficácia do algoritmo; um fator superior a 5 torna-se praticamente igual ao algoritmo ganancioso.
- O único algoritmo que conseguiu resolver o nível 5 (em tempo útil) foi o A* com fator 5.

Referências consultadas

- ZHED Solver - https://www.wilgysef.com/articles/zhed-solver/
- GeeksforGeeks webpage - https://www.geeksforgeeks.org/search-algorithms-in-ai/
- Slides das aulas de Inteligência Artificial
- Russel, S., & Norvig, P. (2010) *Artificial Intelligence A Modern Approach*, New Jersey.

Software utilizado

- IntelliJ IDEA
- Repositório no GitHub
- Gradle Build Tool