Problem G: Spot the Ships

Programação Orientada a Objetos

Aluno:
Davi Barrel Santos
Número:
62838
Curso:
Biologia
Ano:
3º
Professor:
Helder Daniel (hdaniel@ualg.pt)

Introdução:

O projeto consistiu na criação de um programa que busca resolver um jogo de batalha naval, estruturado em um tabuleiro e navios. Diferentes algorítmos para resolver o jogo foram desenvolvidos, e, seguindo o Design Pattern Strategy, o algoritmo usado para resolver o jogo seria determinado em tempo de execução.

O input consiste em uma string referente ao tipo de estratégia que deve ser usada para resolver o jogo; dados a respeito do número de linhas e colunas do tabuleiro; o número de navios que serão inseridos no tabuleiro; e coordenadas e tamanho de cada navio que será inserido no tabuleiro. O output informa o numero de ‘shots’ necessários para resolver o jogo, e um diagrama visual do jogo após resolução.

Design Pattern Strategy

O design pattern usado para a estruturação do código foi o **Strategy Pattern**, que consiste em uma **Interface ou Classe Abstrata** e **Classes concretas**. Estas classes concretas implementam a interface e montam estratégias diferentes para a resolução de um problema comum. Dessa maneira, pode-se, em tempo de execução, decidir qual a melhor maneira de se solucionar um problema comum de acordo com dados do input, por exemplo.

No problema SpotTheShips foi usada uma Interface, denominada ScanningStrategies, contendo os métodos getCounter() e scan(Board currBoard), que possui um tabuleiro como argumento. A interface ScanningStrategies foi implementada pelas classes concretas Linear, Smart e Probabilistic. Nestas foram desenvolvidos algorítmos ou estratégias diferentes para escanear o tabuleiro, mas todas tem a mesma finalidade, encontrar os navios.

Resumo:

Por implementar o Strategy Pattern, o código basea-se em um interface e classes concretas no modulo “client”. Em cada uma dessas classes concretas são implementadas estrátegias diferentes (Linear, Smart, Probabilistic). De maneira geral, essas classes interagem com as classes de produção (Position, Ship, Board, BoardScanner, PositionSorter, LikelihoodMatrix) de modo a implementar as respectivas estratégias.

A classe Position é a unidade básica do tabuleiro, que guarda o estado (scanned/containsShip) e coordenadas (row, col). Usando esta classe, a classe Ships, cria navios com posições definidas. Por meio de navios, células no tabuleiro criado pela classe Board, são marcadas como contendo navio ou não, com auxilio da classe BoardScanner marca se já foram ou não escaneadas. A BoardScanner gera um novo scanner, que pode marcar celulas no tabuleiro como já escaneadas, remover vizinhos proíbidos, caso encontre um navio, e tambem mover celulas para a prioridade de escaneamento, de acordo com a estratégia utilizada. Também usando a classe BoardScanner, a classe LikelihoodMatrix gera uma matriz de verosimilhança, calculando a posibilidade de cada

celula conter navios, de acordo com o tamanho destes e com as células já escaneadas. Uma vez que a classe LikelihoodMatrix estrutura-se usando a estrutura de dados de TreeMap<Position, Integer>, é necessário que as posições estejam ordenadas, o que requer um Comparator<Position>. Isso é suprido pela classe PositionSorter que usando a estrutura do tabuleiro (numero de colunas) e as coordenadas de Position, compara e define qual está mais longe de (0,0) em termos oridinais.

Classes:

Módulo de produção:

- Position:

É a unidade básica de formação do tabuleiro, posteriormente criado pela classe Board. Os atributos da classe permitem localizar a 'célula' no tabuleiro (int row, int col), assim como guardar os estados de scanned e containsShip, em que salva-se se a célula já foi escaneada pelo scanner posteriormente aplicado, e se a célula contém um navio, respectivamente.

Métodos importantes:

→ **hashCode():** retorna um codigo hash usado para inserir Positions em um HashMap posteriormente;

- Ship:

Classe responsável por associar navios à posições no tabuleiro. São criadas novas positions, que posteriormente servirão como referencia para localizar células no tabuleiro e marcá-las como containsShip = true. Navios são criados a partir de uma celula inicial, seguindo uma orientação sul ou leste, e alongam-se em um tamanho específico.

Metodos importantes:

→ **generateThisShipCoordinates()**: cria Positions que serivrão como referencia para marcar posições do tabuleiro como contendo navios.

- Board:

Classe concreta que “cria” um tabuleiro usando uma List<Position> e atributos auxiliares como numero de colunas e linhas que compõe o tabuleiro, número de céĺulas ocupadas por navios.

Métodos importantes:

→ **getPosInListFromCoordinates(int row, int col)**: retorna um objeto Position referente às coordenadas passadas como argumento;

→ **isValidCoord (int row, int col)**: retorna um booleano referente a valiade das coordenadas passadas segundo os limites do tabuleiro;

→ **insertShipsOnBoard (Ship currShip)**: usa as coordenadas de Position que compõe o currShip para marcar no tabuleiro as coordenadas referentes como contendo navio. Incrmenta o contador de células ocupadas pro navios;

→ **getAPositionNeighborhood (Position pos):** retorna posições vizinhas a pos, em um HashMap<String, Position> para facilmente accessar vizinhos recorrentemente usados. O HashMap retornado é composto por strings referente aos pontos cardeais (N, NW, NE, S, SE, SW, W, E).

- BoardScanner:

Classe que orquestra a interação entre as classes do Strategy Pattern (Linear, Smart, Probabilistic) e o tabuleiro (Classe Board). Possui atributos que permitem o manejo dos objetos Position do tabuleiro, e métodos que permitem marcá-las como escaneadas, contar navios encontrados e outros para dar suporte às classes do Strategy Pattern.

Métodos importantes:

→ **setCellAsScannedOnBoard (Position currScanCell)** : marca no tabuleiro a celula passada como argumento como escaneada.

→ **removeForbiddenNeighborsFromScan(Position position, HashMap<String,Position> validNeighbors):** permite facilmente por meio do HashMap remover as celulas vizinhas (NW, NE, SW, SE) da lista de escaneamento, por não poderem conter navios, como definido nas constrains do projeto.

→ **movePriorityCells (HashMap<String, Position> validNeighbors)** : move a celula ao sul da celula que esta a ser escaneada para uma posição de prioridade na lista de escaneamento. Isso ocorre de acordo com o estado das celulas vizinhas, podendo a celula sul ocupar primeira ou segunda posição na prioridade de escaneamento.

→ **addScannedCell (Position pos)**: adiciona uma celula à lista de já escaneadas e atualiza o contador de celulas escaneadas;

- PositionSorter implements Comparator<Position>:

Classe auxiliar que permite ordenar as posições do tabuleiro de acordo com suas coordenadas, e criar posteriormente um TreeMap<Position, Integer> para a matriz de verosimilhança com as posições ordenadas.

→ **compare (Position po1, Position po2)** : retorna um inteiro referente à comparação da posição de pos1 e pos2 no tabuleiro.

- LikelihoodMatrix:

Classe fundamental para a abordagem probabilística de resolução. Gera uma matriz de verosimilhança baseada em uma pontuação referente a possibilidade de uma célula conter um navio, considerando navios de tamanhos de 1 a 5 células.

Métodos importantes:

→ **resetMatrixScores():** recalcula a matriz marcando como -1 as celulas já escaneadas e como 0 as celulas ainda não escaneadas;

→ **calculateMatrixScoreForShipSize (int shipSize)**: calcula, para o tamanho de navio passado como argumento, o score em cada célula. Este é adicionado ao score que a célula já continha. Fundamenta-se no TreeMap para facilmente acessar as células.

→ **recomputeMatrix():** usando outros métodos da classe, recomputa a matriz inteira permitindo posteriormente escanear a celula mais bem pontuada.

→ **computeMaxProbableCell():** retorna a Position mais provável de conter um navio.

→ **addNeighborsToPriorityList():** adiciona celulas vizinhas à uma lista de celulas com prioridade para escaneamento;

Módulo client

- ScanningStrategies (interface):

Interface que contém os métodos:

→ getCounter(): retorna o contador do BoardScanner em uso, de acordo com a estrategia da classe concreta, com o inteiro referente ao número de ‘tiros’ executados para a resolução do tabuleiro;

→ scan(Board currBoard): executa a ação de escanear o tabuleiro de acordo com o algorítmo desenvolvido nas classes concretas;

- Linear (classe concreta):

Classe que implementa ScanningStrategies e seu algoritmo é desenvolvido dentro do método scan(Board currBoard), possui por contrato o getCounter(). Escanea o tabuleiro linearmente da esquerda para direita e de cima para baixo.

- Smart (classe concreta):

Classe que implementa ScanningStrategies e o algorítmo ‘smart’ é desenvolvido dentro do método scan(Board currBoard). O metodo getCounter() retorna o numero de escaneamentos realizados.

- Probabilistic (classe concreta):

Classe que implementa ScanningStrategies e o algoritmo de escaneamento por verossimilhança é desenvolvido dentro do método scan(Board currBoard). O metodo getCounter() retorna o numero de escaneamentos realizados.

Diagrama UML: