Ciências ULisboa | Informática

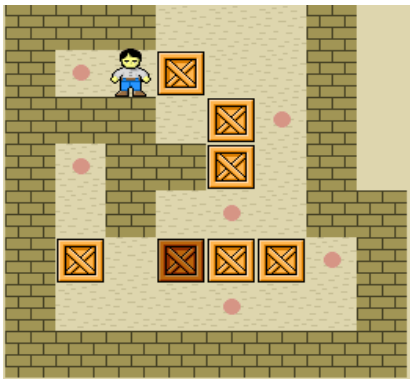
Introdução à Programação

Licenciatura em Engenharia Informática

Trabalho: 3ª parte

2021/2022

Sokoban



O trabalho de programação que vos é proposto em IP é em torno do jogo *Sokoban*. O jogo *Sokoban* é jogado numa quadrícula retangular como a mostrada na imagem, onde algumas posições têm paredes imóveis ( ). Distribuídas pelas restantes posições da quadrícula existem *caixas* móveis ( ) e igual número de *objetivos* ( ). O *jogador* ( ) tem de empurrar as caixas de modo que todas as caixas fiquem nas posições objetivo.

Cada posição da quadrícula é considerada *ocupada* se tem uma parede, uma caixa ou o jogador. Em cada jogada é apenas preciso indicar a direção (*cima*, *baixo*, *esquerda* ou *direita*) em que se quer deslocar o jogador. Se a posição adjacente nessa direção existir e não estiver ocupada, o jogador desloca-se para lá; se contiver uma caixa e a posição adjacente à caixa nessa direção existir e não estiver ocupada, a caixa é empurrada para essa posição e o jogador fica onde a caixa estava anteriormente. Em todas as outras situações nada acontece, pois, nem o jogador nem as caixas podem sair da quadrícula ou ocupar posições com paredes e o jogador não pode empurrar duas caixas simultaneamente. O jogador perde o jogo se as caixas ficarem presas num canto ou atrás de uma parede.

Na segunda fase do trabalho implementaram as funcionalidades que permitiam jogar um cenário de *Sokoban*. Nesta terceira fase vão desenvolver as classes que permitem construir uma aplicação para jogar *Sokoban*.

Em que consiste o trabalho, afinal?

A vossa tarefa é programar um tipo enumerado `Direction` cujos valores representem as quatro direções possíveis para as jogadas de *Sokoban* – UP, DOWN, LEFT, RIGHT – e duas classes Java:

- `SokobanMap` que é uma classe que define um tipo cujos objetos, imutáveis, representam mapas (isto é, cenários iniciais) válidos deste jogo.
- `SokobanGame` que é uma classe que define um tipo de objetos que representam partidas deste jogo. Uma partida desenrola-se em diferentes níveis. Para cada nível há um mapa inicial. Quando o jogador termina um nível, pode passar ao nível seguinte.

A API oferecida por cada uma destas classes é detalhada de seguida.

`SokobanMap`. Os objetos deste tipo são imutáveis e representam mapas válidos de *Sokoban*. Um mapa é constituído por uma matriz que identifica as posições ocupáveis, um conjunto de posições objetivo, um conjunto de posições iniciais para as caixas e posição inicial do jogador. A classe deve incluir:

- `public static boolean isValidMap(int rows, int columns, boolean[][] occupiableMap, int[][] goals, int[][] boxes, int[] playerPos)` que, considerando que `occupiableMap` define quais as posições do mapa que são *ocupáveis* (ou seja, *não* têm paredes), `goals` e `boxes` definem as posições dos objetivos e das caixas no mapa e `playerPos` indica a posição inicial do

jogador, verifica:

1. se `rows, columns>2`
2. se `occupiableMap!=null` e `occupiableMap` é uma *matriz* de dimensão `rows x columns`
3. se `goals` é um *vetor de posições válidas*, isto é:
   (a) `goals!=null` e `goals.length>0`
   (b) para cada `0≤i<goals.length,` o vetor `goals[i]` representa uma *posição válida no mapa*, ou seja: `goals[i]!=null`, `goals[i].length == 2`, `0≤goals[i][0]<rows` e `0≤goals[i][1]<columns`
   (c) `goals` não tem elementos repetidos (o vetor `goals[i]` representa uma posição diferente de `goals[j]` para cada `i≠j`)
4. se `boxes` também é um *vetor de posições válidas*
5. se `goals.length == boxes.length`
6. se todos os elementos de `goals` e `boxes` (`goals[i]` e `boxes[i],` para `0≤i<goals.length`) são posições *ocupáveis* de acordo com a matriz `occupiableMap`
7. se o vetor `playerPos` é uma *posição válida no mapa*, pode ser *ocupável* de acordo com a matriz `occupiableMap` e não está contida no vetor `boxes.`

- `public SokobanMap(int rows, int columns, boolean[][] occupiableMap, int[][] goals, int[][] boxes, int[] playerPos)` que, assumindo que `isValidMap(rows,columns, occupiableMap,goals,boxes,playerPos)` constrói um mapa com os dados fornecidos
- `public int getRows()/getColumns()` que indica o número de linhas/colunas do mapa
- `public int getNrBoxes()` que indica o número de caixas
- `public int[] getInitialPlayerPosition()` que devolve um vetor de tamanho 2 com a posição (linha e coluna) inicial do jogador
- `public int[][] getInitialPositionBoxes()/getInitialPositionGoals()` que devolve uma matriz de dimensão `getNrBoxes()x2` com a posição (linha e coluna) inicial de cada caixa/objetivo
- `public boolean isOccupiable(int i, int j)` que indica se a posição dada é *ocupável* (ou seja, não tem parede), assumindo que `0≤i<getRows()` e `0≤j<getColumns()`

`SokobanGame`. Os objetos deste tipo representam partidas de um jogo de *Sokoban*. O estado de uma partida deverá conter o nível atual, o mapa desse nível e a posição atual das caixas e do jogador nesse mapa. A classe deve incluir:

- `public SokobanGame()` que cria uma partida nova com o mapa e as posições iniciais dos objetivos, caixas e jogador no nível `1`, fornecidos pela classe `SokobanMapGenerator` (deverão consultar a documentação fornecida).
- `public int getRows()/getColumns()` que devolve o número de linhas/colunas do mapa
- `public int[] getPlayerPosition()` que devolve a posição (linha e coluna) atual do jogador
- `public Direction getDirection()` que devolve a direção do último movimento (se ainda não houve nenhum movimento, a direção retornada deverá ser DOWN).
- `public int getLevel()` que devolve o nível atual do jogo
- `public int getNrMoves()` que devolve o número de movimentos válidos já executados pelo jogador no nível atual (aqueles que resultam num movimento do jogador para outra posição)
- `public int[][] getPositionBoxes()/getPositionGoals()` que devolve uma matriz de dimensão `numCaixas()x2` com a posição (linha e coluna) atual de cada caixa/objetivo
- `public boolean isOccupiable(int i, int j)` que indica se a posição dada é *ocupável* (ou seja, não tem parede), assumindo que `0≤i<getRows()` e `0≤j<getColumns()`
- `public void move(Direction dir)` que executa a jogada (com tudo o que isso implica),

assumindo que `dir!=null` e `!levelCompleted()`.

- `public boolean levelCompleted()` que indica se o nível atual está terminado, isto é, se todas as caixas se encontram em posições objetivo (e apenas uma caixa por posição objetivo)
- `public boolean isTerminated()` que indica se o nível atual está terminado e é o último nível
- `public void loadNextLevel()` que carrega o mapa e as posições iniciais dos objetivos, caixas e jogador no próximo nível, fornecidos pela classe `SokobanMapGenerator`, assumindo que `!isTerminated() && levelCompleted()`
- `public void restartLevel()` que coloca a partida no estado inicial do nível atual
- `public String toString()` que devolve uma representação textual do estado da partida que mostre a distribuição das paredes e objetivos pela quadrícula e ainda a localização do jogador e das caixas, o número de movimentos e o nível atual, como se mostra no exemplo abaixo. As posições da quadrícula são representadas da seguinte forma: P representa o jogador, B representa uma caixa que não está numa posição objetivo, * representa uma caixa que está numa posição objetivo, G representa um objetivo que não está ocupado e – representa uma parede. O espaço em branco é usado para representar as posições que nem estão ocupadas nem têm um objetivo.

```
+-----------------+
 LEVEL: 4
+------ MAP ------+
| - - -       -   |
| - G P B     -   |
| - - -   B G -   |
| - G - - B   -   |
| -   -   G   - - |
| - B   * B B G - |
| -       G     - |
| - - - - - - - - |
+-----------------+
 MOVES: 0
+-----------------+
```

Tarefa Adicionar (Opcional)

A tarefa adicional, que é opcional e que permite aceder a um valor de bónus, consiste em fazer uma classe `SokobanTxtInterface` com uma versão textual da interface do jogo.

Como posso testar o meu código?

Uma alternativa é usar a classe `SokobanTest` fornecida, a qual exercita uma parte do seu código em alguns cenários típicos. A outra alternativa é exercitar o seu código através da execução de um programa com uma interface gráfica que é fornecida. Para tal, precisa apenas de colocar as suas classes no mesmo diretório das classes que são fornecidas e mandar executar a classe `SokobanGUI`. A interface gráfica permite jogar um jogo de *Sokoban*, como ilustrado abaixo. Para movimentar o jogador, deverão usar as setas do teclado *ou* as teclas R, L, U, D.

O que entrego?

Os ficheiros `Direction.java`, `SokobanMap.java` e `SokobanGame.java` e, no caso de ter escolhido fazer a tarefa adicional, `SokobanTxtInterface.java`. Não há relatório a entregar porque o vosso software é a vossa documentação. Assim, não se esqueçam de comentar condignamente a vossa classe. Devem incluir no início de cada uma das classes um cabeçalho *javadoc* com `@author` (nome e número dos alunos que compõem o grupo). Para cada procedimento/função definidos há que preparar um cabeçalho incluindo a sua descrição, e, se for caso disso, `@param`, `@requires`, `@ensures` e `@return`. Apresentem um texto "limpinho", que siga as normas de codificação em Java, bem alinhado e com um número de colunas adequado. Consultem, na página da disciplina, o *Guia de Estilo Java para IP*.

Como entrego o trabalho?

Um dos alunos do grupo entrega o trabalho através da ligação que, para o efeito, existe na página da disciplina no *moodle*. O prazo de entrega é dia 9 de Janeiro às 23h55.

Quanto vale o trabalho?

Esta 3ª parte do trabalho é cotada para 7.5 valores e a tarefa adicional é cotada para `1` valor. A nota final do trabalho será dada por `min(notaFase1 + notaFase2 + notaFase3 + notaTarefaAdc, 20)`.