DM 503 Programming B

Spring 2012 Re-Exam Project

Department of Mathematics and Computer Science
University of Southern Denmark

May 14, 2012

Introduction

The purpose of the project for DM503 is to try in practice the use of recursion and object-oriented program design.

Please make sure to read this entire note before starting your work on the project. Pay close attention to the sections on deadlines, deliverables, and exam rules.

Exam Rules

Thisproject is an exam. Thus, the project must be done individually, and no cooperation is allowed beyond what is explicitly stated in this document.

Deliverables

- A short project report in PDF format (at least 5-6 pages without front page and appendix) has to be delivered. This report has to contain the following 7 sections:
  - front page
  - specification
  - design
  - implementation
  - testing
  - conclusion
  - appendix
- All source code.

The deliverables have to be delivered using Blackboard's Assignment Hand-In functionality. Delivering by e-mail or to the teacher is only considered acceptable in case Blackboard is down before the deadline.

Deadline

June 18, 2012, 12:00

The Problem

Your task is to implement strategies for heroes and monsters in a simplified version of the game Rogue[1].

More precisely, you have to implement the classes Rogue, Monster, and SpeedyRogue as described in more detail below.

The Game

Rogue is supposedly the first "graphical" adventure game ever. It was very popular with Unix users at universities at the beginning of the 1980s. The goal of Rogue is "to descend into the Dungeons of Doom, defeat monsters and find treasure and come back with the amulet of Yendor using its levitation capabilities". The dungeon features monsters, which will attack the player character when entering a room. For the player character to be in optimal health when facing these monsters, a good strategy is to avoid fighting the monsters for as long as possible. This gives rise to a very interesting search problems over a graph representation of the dungeon. These problems are the background for the tasks of this project.

In our simplified version, the game is played on a $N \times N$ playing field, where the hero is represented by the sign @ and monsters are represented by uppercase letters (A–Z). There is only exactly one hero and one monster on each playing field. If the monster catches the hero, it defeats him and the game is ended.

The hero and the monster move by turns. In each turn, first the monster moves, then the hero. Each player (hero / monster) can stay on his current position or move to one of the neighboring cells. There are three types of cells: rooms, hallways, and walls. They are represented by the characters '.' (dot), '+' (plus), and ' ' (space), respectively.

If a player stands in a room cell, he can move to up to eight neighbor fields (north, northeast, east, southeast, south, southwest, west, northwest - where north is up and east is right). If a player stands in a hallway cell, he can move to up to four neighbor fields (north, east, south, west) but not diagonally. It is not possible to move onto or through walls.
The home page of the course contains a number of possible playing fields to test your strategies.

[1]`http://en.wikipedia.org/wiki/Rogue_(computer_game)`

```
          + + + + +         . . . . . . . . A .
          +       +         . . . . . . . . . .
. . . . . . . .   +         . . . . . . . . . .
. . . . . @ . .   +         . . . . . . . . . .
. . I . . . . .   +         . . . . @ . . . . .
. . . . . . . .   +         . . . . . . . . . .
. . . . . . . .   +         . . . . . . . . . .
          +       +         . . . . . . . . . .
          +       +         . . . . . . . . . .
          + + + + +         . . . . . . . . . .
```

Above there are two 10x10 playing fields. In case of the left field, the hero can escape from the monster forever by running into the hallway (and if the monster follows across the room to get to the other end of the hallway). In case of the right field, the monster can always catch the hero by moving towards the hero and trapping him in a corner. Note that a playing field can easily be modelled as a graph where each node represents a cell of the playing field and each edge between two nodes represents that it is allowed to move from one of these nodes to the other one in one step.

Strategy of the Monster The goal of the monster is, of course, to catch and eat the hero. A natural strategy of the monster is to compute for each turn the shortest paths towards the hero and take one step along one of these paths. Note that there can be more than one shortest path. In that case, the monster should select the new position with the least Euclidian distance to the hero. That is not necessarily an optimal strategy. This can for example be seen in the left playing field below where the monster B can be sure to catch the hero by moving northeast (NE) while moving east (E) will allow the monster to flee into the hallway. On the right playing field , the monster can only be sure to catch the hero by moving E, not by moving NE or southeast (SE).

```
    + + + + + + + +                   + + + + +
    +             +                   +       +
. . . . . . .     +         . . . . . . . .   +
. . . . . . .     +         . . . . . . . .   +
. . . . @ . .     +         . . C . . @ . .   +
. . B . . . .     +         . . . . . . . .   +
. . . . . . .     +         . . . . . . . .   +
. . . . . . . + + +                   +       +
. . . . . . .                         +       +
. . . . . . .                         + + + + +
```

Your first task is to implement the strategy described above for the monster (see description of the class Monster below).

Strategy of the Hero The strategy of the hero is to avoid the monster as long as possible. A natural strategy of the hero is to identify the neighbor cells where the hero can move in the next step. For each of these neighbor cells, the length of the shortest path to the monster can be computed. The hero then moves to a cell which has maximal distance to the monster. Note that there can be more than one cell with a maximal shortest path to the monster. In that case, the hero should select one of those neighbor cells which offers the most possible moves in the next turn, i.e., which has the maximal number of neighbors among all cells with maximal distance from the monster. It is easy to see that also this strategy is not optimal.

```
. . . . . . .
. . . . . . .
. . . . . @ .
. . . . . . . + + +
. . . . . . .     +
. . . . . . . + + +
. . . . . . J
. . . . . . .
. . . . . . .
. . . . . . .
```

In case of the above playing field, the hero can only avoid the monster forever if he moves SE instead of flying from the monster. Then in the second turn, he can disappear into the corridor.

Your second task is to implement the strategy described above for the hero (see description of the class Rogue below).

Players on Steroids In the full game Rogue, there is a rich variety of things to pick up. Gold, food, weapons, armor, and, of course, magical potions. In our simplified game, there is only one type of potion – called Potion of Speed and denoted by the character ‘s’. When a player moves onto a cell with a potion, he immediately consumes the potion. In all following turns, he can now move twice instead of once. If a player consume more than one potion, for each potion, one move per turn is added. Thus, if a player were to consume three potions, he could move three steps in each turn. The existence of potions changes what is a good strategy and what not.

```
. . . . . . . . A .
. . . . . . . . . .
. . . . . . . . . .
. . . . . s . . . .
. . . . @ . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

In case of the above playing field, the hero should move towards the monster in the first move to consume the potion. Afterwards, the hero can easily avoid the monster forever.

Your third task is to implement a second strategy for the (see description of the class SpeedyRogue below) that tries to reach a speed potion if it can do so before the monster gets there.

Input-format Each file contains a playing field, where the first line denotes N (the size of the NxN playing field contained in the file) and each following line denotes all playing fields by 2N characters as defined above. The characters denoting a cell are separated by a space while the line is ended by a newline.

```
10
    + + + + + + + +
    +             +
. . . . . . .     +
. . . . . . .     +
. . . . @ . .     +
. . B . . . .     +
. . . . . . .     +
. . . . . . . + + +
. . . . . . .
. . . . . . .
```

A room is a connected block of room cells. Because of ministerial regulations, rooms may not be directly next to each other. Rooms are connected by at least one hallway cell. There is exactly one monster and one hero per playing field and they both start in a room (not necessarily the same).

The Task Together with this description, the course home page contains the following four template files:

Position.java This class represents a position on the playing field. The most important methods are `getX()` (returning the horizontal coordinate of the position), `getY()` (returning the vertical component of the position), and `equals(...)` to compare the current position with a given other position.

Dungeon.java This class represents a playing field. It is possible to obtain its size (`size()`), get the hero's position (`getRoguePosition()`), get the monster's position (`getMonsterPosition()`), and a (possibly empty) set of the positions of potions (`getPotionPositions()`).

Furthermore, it is possible to find out wheter it is allowed to place a player at a certain position (i.e., whether we have a room or a hallway cell), whether a given position is a room cell, whether a given position is a hallway cell, and whether it is allowed to move from one position to another in accordance to the rules of the game.

AbstractCharacter.java This is an abstract class that represents a player in the game, i.e., a hero or a monster. A player knows the playing field and his position and through the playing field also the position of the opponent. Furthermore, every player has to implement a method `move(int remainingMoves)`, that is called each time it is this player's turn to move. The method should return the new position the player wants to move to. The argument denotes the number of moves this player can take before it is the opponents turn. Unless the player has consumed at least one potion, this is always 0.

GameEngine.java This file handles runs the game, i.e., it reads the playing field from a file, constructs the two players, and controls when which player can move. It stops when the hero is caught. This class contains the only main method of the project.

Note that positions that the positions mentioned above have their origin in a starting point at the top-left of the playing field and grow to the right and down. The top-left cell has the position (0,0).

It is not necessary (and also not possible with the contents of the course) to understand all details in the file `GameEngine.java`. Especially the loading of the players uses Java's reflection framework which has not been handled in class. The good news is that it is not necessary to understand this class to be able to complete the project.

It is in general *NOT ALLOWED* to *CHANGE* the template files!

Your task is to write three classes `Monster`, `Rogue`, and `SpeedyRogue` that both inherit from `AbstractCharacter` and implement the strategies described above. To this end, you should use the methods of the playing field (`Dungeon`), in particular the method `isLegalMove`, to build a graph over the playing field (as described above) and implement breadth-first search (BFS) to compute the length of the shortest path between to given cells of the playing field.

A good idea is to let `SpeedyRogue` inherit from `Rogue` in order to reuse the move method for the case that one does not want to (or cannot) reach a potion. In this case, the method `move(int remainingMoves)` of `Rogue` can be called by `super.move(remainingMoves)`.

It is a part of your task to use classes and source code that you have not written yourself (nor that you necessarily completely understand). This is a very relevant setting for the application of object-oriented programming.

When you have implemented the strategies described above for both the monster and for the hero, the game can be started as follows:

```
java GameEngine dungeonA.txt Rogue Monster
```

where `dungeonA.txt` can be replaced by another playing field and where you want to test your strategies for the hero i `Rouge.java` and the monster `Monster.java`. If you want to use the more advanced strategy for the hero, use `SpeedyRouge` the second argument to `GameEngine`.