UNIVERSIDADE DO ALGARVE
Faculdade de Ciências e Tecnologia
Departamento de Engenharia Electrónica e Informática

Inteligência Artificial ***(Ano lectivo 2021/2022 – 1º Semestre)***
Licenciatura em Engenharia Informática

José Valente de Oliveira *jvo@ualg.pt*

Lab tutorial 1: Small Instances of the 8-puzzle
How to use the Strategy design pattern to implement state-space search strategies

Version 5.0 – 28 Set. 2021

Goal
Test-driven development of state-space strategies in Java using the Strategy design pattern, and assessment of code correctness and efficiency via Mooshak.

Appendices:
A: Problem description
B: JUnit assertions

Lab tutorial 1: Small Instances of the 8-puzzle, or
How to use the Strategy design pattern to implement state-space search strategies

Submit:
- Your group source code to mooshak http://deei-mooshak.ualg.pt/~jvo/ up to October 25, 2021

Part I: Getting started with Mooshak

1. Registo no Mooshak

1.1. Antes de mais nada, inscreva-se num turno e num grupo. Vá a http://deei.fct.ualg.pt/IA/Entregas/

1.2. Depois, siga para http://deei-mooshak.ualg.pt/~jvo/

1.3. Para obter User/Password clique em Register [for On-line Contest]

1.3.1. Em Contest, seleccione IA 2021/22

1.3.2. Em Group, selecione o seu turno prático

1.3.3. Em Name, escreva IA2122P<i>G<j> onde <i> é o seu turno prático e <j> é o seu número de grupo, usando os dados fornecidos em http://deei.fct.ualg.pt/IA/Entregas/

1.3.4. Em Email, escreva o endereço de correio eletrónico de um dos elementos do grupo para onde será enviada a password.

1.4. Depois de efectuado o login, pode escolher, entre outras coisas, o seguinte:

1.4.1. Problem – Escolha A – Lab Tutorial 1

1.4.2. View – lista o enunciado do problema selecionado

1.4.3. Ask – coloque aqui as suas questões sobre o problema selecionado

1.4.4. Procurar … Submit – permite fazer o upload do ficheiro com o código java. Neste momento, vamos submeter sempre um único ficheiro de código.

1.4.5. Depois de devidamente testado, faça o upload do seu ficheiro .java

1.4.6. Aguarde pela resposta do Mooshak e veja no Help o seu significado, se necessário.

2. Entradas/Saídas

(Extraídos de http://ctp.di.fct.unl.pt/~amd/cpn/2007tiup/etapa5/praticos.html)

<quote>

Dados de entrada

Os dados de entrada, usados para testar as soluções, são lidos da entrada padrão. Consistem em texto cuidadosamente formatado para ser simples de processar:

- Normalmente, nas primeiras linhas dos dados de entrada surgem alguns números inteiros que anunciam o tamanho das diversas partes do texto que se segue. Isso evita a necessidade de testar a condição de "fim-de-ficheiro", durante a leitura dos dados.
- A última linha do ficheiro está sempre devidamente terminada por uma mudança de linha.

- Espaços em branco, quando usados, são sempre considerados como **separadores**. Os espaços em branco nunca ocorrem em grupos. Uma linha nunca começa com um espaço em branco. Uma linha nunca acaba com um espaço em branco.

Note que as linhas com números inteiros que ocorrem no início dos dados de entrada devem ser consumidas bem até ao fim para evitar desalinhamentos na leitura dos dados subsequentes. Eis como isso se faz em Java:

```
import java.util.* ;

Scanner sc = new Scanner(System.in);
  /* Aviso: nunca crie mais do que um Scanner sobre o input. */
int n = sc.nextInt() ;
sc.nextLine() ;     /* Salta a mudança de linha. */
```

- Supondo que os dados se iniciam por uma linha contendo **dois inteiros separados por um espaço em branco**:

```
import java.util.* ;
Scanner sc = new Scanner(System.in);
int n = sc.nextInt() ;
int m = sc.nextInt() ;
sc.nextLine();            /* Salta a mudança de linha. */
```

Dados de saída
Os dados de saída, produzidos pelos programas submetidos aos mooshak, são escritos na saída padrão. É necessário respeitar rigorosamente o formato exigido no enunciado. Qualquer desacerto, mesmo ligeiro, é suficiente para que um programa seja classificado como "Presentation error".

Note que não é possível detectar visualmente certas anomalias nos dados de saída. Por exemplo: um espaço em branco no final duma linha, uma linha em branco no final dos dados, a omissão da mudança de linha na última linha dos dados. Todas estas situações são inaceitáveis e provocam um "Presentation error".

</quote>

3. Erros frequentes

Apresenta-se aqui alguns dos erros mais frequentes reportados pelo Mooshak e sua resolução.

3.1. Presentation error

Tipicamente, o programador esquece-se de colocar o “\n” no final da saída.
Solução: Usar System.out.println em vez de System.out.print

3.2. Compile time error

Muitas vezes o compile time error deve-se apenas à utilização de caracteres portugueses nos comentários. Nem o javac nem o eclipse se queixam mas o mooshak assiná-lo-os com um warning.
Solução: Remover do ficheiro a submeter todos os acentos, cedilhas, etc.

3.3. Run time error
Uma fonte de problemas é a leitura incorrecta dos dados de entrada. Erros típicos são: i) leitura incorrrecta do formato dos dados, ii) utilização de vários objectos Scanner sobre a entrada padrão.

Avisos do Compilador
Alguns compiladores geram avisos (warnings) quando encontram caracteres que não conseguem identificar, ***mesmo nos comentários***. Isto sucede com caracteres acentuados como os portugueses (por vezes pode ser apenas uma questão de *codepage* diferente entre o sistema onde foi escrito o código e o sistema onde é compilado).

O Mooshak pode estar configurado para abortar a compilação não só quando o compilador encontra erros no código mas também quando o compilador gera avisos, sendo indicado **"Compilation error"**.

Por isso recomenda-se que se evite a utilização no código fonte de caracteres acentuados mesmo nos comentários (no caso geral caracteres cujo código ASCII é superior a 127).

4. Submissão
Para submissão ao Mooshak temos duas hipóteses ou submetemos um único ficheiro .java ou um ficheiro compactado .zip ou .gz

Hipótese ficheiro .java: o programa completo é *integralmente* incluído num único ficheiro de extensão **.java**; é esse ficheiro que é submetido ao mooshak.

O conteúdo possível de desses ficheiros é:

0. opcionalmente, uma diretiva **package**;
1. zero ou mais diretivas **import**;
2. uma ou mais definições de classes, sendo que uma e só será pública e terá o mesmo nome do ficheiro.

Hipótese ficheiro compactado: Compacte o diretório raiz do código fonte *src* (e todos os sub-diretorios) num arquivo *.zip* ou *.gz;* é esse ficheiro que é submetido ao mooshak. Por defeito, o diretório raiz está em Workspace, num subdirectório com o nome do projeto:

…/<Workspace>/<Nome projeto>/src

Part II: To Do

In this tutorial we are going to develop a program for solving small instances of a classical toy problem, the 8-puzzle, cf. problem description at the appendix below. There is little of Artificial Intelligence in this lab tutorial. Our main goal here is to remind ourselves of the Strategy design pattern, and how to use it to implement state-space search algorithms. We will recall also Unit Tests on the fly.

The work developed in this tutorial will be *fully* reusable in future lab works.

0. Thoroughly read the problem description given in the appendix. Make sure you fully understand the problem. Should you have any doubt about it, do not hesitate to ask.

1. We will solve this problem using a best-first state-space search algorithm. Later we will see more efficient algorithms to solve this type of problem. For now, the algorithm given within the box below will serve our current goal.

Algoritmo de procura melhor primeiro (config initial, config goal)

```
Criar estado initial a partir da config initial.
Criar a fila com prioridade abertos para estados ainda não testados;
Criar lista fechados para estados já testados;
Inserir estado inicial em abertos;
Repetir até que uma solução seja encontrada (e enquanto houver memória):
        Se (lista abertos vazia) sair com fracasso
        estado actual := primeiro de abertos;
        remover actual de abertos;
        Se (config do estado actual for igual à config goal)
                Mostrar sequência desde inicial até goal;
        Senão
                sucessores := todos os filhos do estado actual;
                inserir actual na lista fechados;
                Para cada sucessor em sucessores
                    Se (sucessor não existir em fechados)
                            inserir sucessor em abertos;
```

2. Let's see how this algorithm works for our problem.

3. The algorithm is called with two arguments. The first (config *initial*) is the initial configuration of the board while the second denotes the goal configuration (config *goal*). From problem description, in sample input 1, config *initial* is 023145678 while config *goal* is 123405678.

4. Based on config *initial*, the algorithm creates the **state** *inicial.* A state represents a node in a search graph. This node can be seen as a data structure containing a config, its current cost, and a reference to the father of the node, the father of *inicial* state being null.

5. Next, a priority queue (named *abertos*) is created for holding those states not yet tested, and a list (named *fechados*) is created for holding those states already tested. A state is tested whenever its config is compared to the *goal* config.

6. The priority queue will maintain its elements ascending sorted by the state sorting cost.

7. The state *initial* is now inserted into the *abertos* priority queue.

8. At this point, the algorithm enters a loop that will end if one the following cases occur:

```
Best-first search (config initial, config goal)

    Create the state inicial (initial in English) from config initial;
    Create a priority queue named abertos (open in English) for those states not tested yet;
    Create a list named fechados (closed in English) for those states already tested;
    Insert the state inicial into abertos;
    Repeat until a solution is found:
        If (list abertos is empty) exit with failure;
        Let actual be the first from abertos;
        remove actual from abertos;
        If (config of the actual state equals config goal)
            A sequence from config initial to config goal is shown.
        else
            Let sucessores hold all the children of the state actual;
            Insert actual into fechados;
            For each sucessor in sucessores;
                If (sucessor not presented in fechados)
                    Insert sucessor into abertos;
```

a. A solution is found
b. No solutions are found (the *abertos* priority queue is empty)
c. The algorithm runs out of memory

9. Once inside the loop the first thing to do is to get (and remove) the first element from *abertos*, i.e., the element of the least current cost – keep in mind that *abertos* is a priorty queue. We named this state, *actual*.

10. Now, the config inside the state *actual* is tested for equality with the input config *goal*. If it is equal, a solution is found, returned, and the execution ends.

11. Else, *actual* will be expanded. That is, we find all configurations which are reachable from *actual*, when the valid rules of our problem are applied. In our case, this mean if the actual is 023145678, the reachable configurations are 123045678, and 203145768.

12. At this time, we have only configurations. We need to build states from these configurations. That is, for each configuration we need to create a state, compute its current cost, and assign its father (the state *actual*). We name each one of these new states *successor*.

13. The cost of a given state is the cost of its father plus the cost of the expansion. For our problem, this cost is simply 1.

14. Before inserting each *successor* into *abertos*, we need to checks whether this particular *successor* was (nor not) already processed by the algorithm. In other words, we insert *successor* into *abertos* whenever it does not exists in *fechados*. The equality test used here should compare the configs of the states. This completes the algorithm.

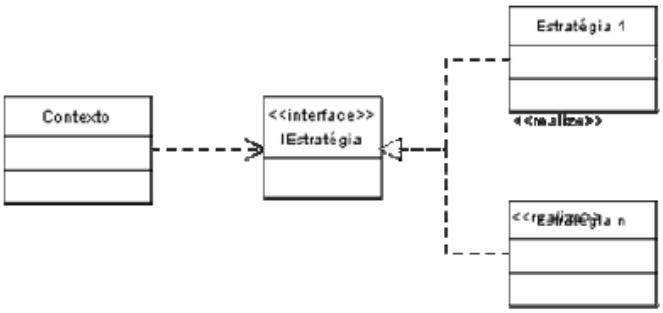
15. Make sure you fully understand the algorithm. Should you have any questions do not hesitate to ask.

16. As you can image, this algorithm is not specific for this problem. As we will learn in due time, we can find solutions for almost any type of problem using a state-space formulation and, for some of them this best-first search can be used.

17. We shall prepare our implementation of the algorithm to be easily applicable to any problem we may wish to solve with. In order to doing this, we need to clearly separated what is invariant from what can change.

18. In this case, one of the best ways of doing this is to adopt the design pattern Strategy. The basic idea is to implement our state-space algorithm (the context in the Strategy design pattern) dependent on an interface (the strategy). If we want to solve a concrete problem with the algorithm all we need to do is to define a class which implements the interface. If you want to solve another concrete problem, well, define another class that provides a problem specific implementation for the interface.

19. The UML class diagram for the Strategy design pattern can be given by the following figure.



20. In our case, an interface based on which our algorithm will be developed is the following:

```
interface Ilayout {
        /**
        @return the children of the receiver.
        */
        List<Ilayout> children();

        /**
        @return true if the receiver equals the argument l; return false otherwise.
        */
        boolean isGoal(Ilayout l);

        /**
        @return the cost for moving from the input config to the receiver.
        */
        double getG();
}
```

21. This interface specifies the behaviour that may changes from concrete problem to concrete problem, in our implementation. A class representing a concrete problem should implement at least this behaviour.

22. Having said this, we can now start developing our program. Open Eclipse and create a new java project.

23. We will now prepare the project for test-driven development. Add to the project a new JUnit Test Case (File -> New – Junit Test Case).

24. At the top select **New JUnit 5 test** and name it **PuzzleUnitTests**, them press Finnish.

25. A dialog may open asking this: ***JUnit 5 is not on the build path. Do you want to add it?*** If this appears press OK.

26. Now, within the file **PuzzleUnitTests.java** and the definition of class, we should have:

```
import org.junit.Test;
import static org.junit.Assert.*;

public class PuzzleUnitTests {
   // Tests will go here

}
```

27. The first tests we will develop will test the behaviour of the class constructor that implements the Interface Ilayout, above. We shall call this class Board.

```
public class PuzzleUnitTests {
    @Test
    public void testConstructor() {
          Board b = new Board("023145678");
        StringWriter writer = new StringWriter();
        PrintWriter pw = new PrintWriter ( writer ) ;
        pw.println(" 23");
        pw.println("145");
        pw.println("678");
        assertEquals(b.toString(), writer.toString());
        pw.close();
    }
    @Test
    public void testConstructor2() {
          Board b = new Board("123485670");
          StringWriter writer = new StringWriter();
        PrintWriter pw = new PrintWriter (writer) ;
        pw.println("123");
        pw.println("485");
        pw.println("67 ");
        assertEquals(b.toString(), writer.toString());
        pw.close();
    }
}
```

28. Now is time to write the simplest code that passes this test. That is, implement the constructor of *Board*, and implement its toString() method. Add a new class to the project, copy/past, and complete the code bellow:

```
interface Ilayout {
    /**
    @return the children of the receiver.
    */
    List<Ilayout> children();

    /**
```

```
    @return true if the receiver equals the argument l;
              return false otherwise.
    */
    boolean isGoal(Ilayout l);

    /**
    @return the cost for moving from the input config to the receiver.
    */
    double getG();
}


class Board implements Ilayout, Cloneable {
    private static final int dim=3;
    private int board[][];

public Board(){ board = new int[dim][dim];}


public Board(String str) throws IllegalStateException {
    if (str.length() != dim*dim) throw new
          IllegalStateException("Invalid arg in Board constructor");
    board = new int[dim][dim];
    int si=0;
     for(int i=0; i<dim; i++)
       for(int j=0; j<dim; j++)
            board[i][j] = Character.getNumericValue(str.charAt(si++));
}


public String toString() {
    // TO BE COMPLETED
}
```

29. Now, test and develop the functions specified in the interface Ilayout above, namely **children(), isGoal(Ilayout l), and getG();**

30. Our client code reads as:

```
public class Main {
      public static void main (String [] args) throws Exception {
          Scanner sc = new Scanner(System.in);

          BestFirst s = new BestFirst();
          Iterator<BestFirst.State> it = s.solve(new Board(sc.next()),
                                                 new Board(sc.next()));
          if (it==null) System.out.println("no solution was found");
          else {
                 while(it.hasNext()) {
                       BestFirst.State i = it.next();
                       System.out.println(i);
                       if (!it.hasNext()) System.out.println(i.getG());
                 }
          }
          sc.close();
       }
}
```

31. It's clear that the method `solve` returns an `iterator`, which iterates over the algorithm states that are in the solution path.

32. The time has come to implement our algorithm. Understand the code below and complete it as indicated.

```
class BestFirst {
   static class State {
         private Ilayout layout;
         private State father;
         private double g;
         public State(Ilayout l, State n) {
               layout = l;
               father = n;
               if (father!=null)
                     g = father.g + l.getG();
               else g = 0.0;
         }
         public String toString() { return  layout.toString(); }
         public double getG() {return g;}
   }
   protected Queue<State> abertos;
   private List<State> fechados;
   private State actual;
   private Ilayout objective;

   final private List<State> sucessores(State n) {
         List<State> sucs = new ArrayList<>();
         List<Ilayout> children = n.layout.children();
         for(Ilayout e: children) {
               if (n.father == null || !e.equals(n.father.layout)){
                     State nn = new State(e, n);
                     sucs.add(nn);
               }
         }
         return sucs;
   }


   final public Iterator<State> solve(Ilayout s, Ilayout goal) {
         objective = goal;
         Queue<State> abertos = new PriorityQueue<>(10,
                        (s1, s2) -> (int) Math.signum(s1.getG()-s2.getG()));
         List<State> fechados = new ArrayList<>();
         abertos.add(new State(s, null));
         List<State> sucs;

         // TO BE COMPLETED

   }
}
```

33. And we are almost done. Test your program on appropriated test cases.

34. Submit your program to mooshak (http://deei-mooshak.ualg.pt/~jvo/) – Contest: IA 2021/22, Problem A.

Appendix: Instâncias pequenas do quebra-cabeças de 8 peças

// For the English version see below

O quebra-cabeças baseia-se num tabuleiro com 3x3 posições e 8 peças deslizantes, numeradas de 1 a 8. A figura seguinte apresenta uma configuração possível para o tabuleiro.

1	2	3
4		5
6	7	8

Qualquer peça pode ser movimentada para uma posição adjacente, desde que essa posição esteja livre. O objetivo é encontrar as sucessivas configurações pelas quais o tabuleiro passa desde a configuração inicial até à configuração final, ambas dadas, pelo caminho mais curto.

Por exemplo, para passar de

```
  2 3
1 4 5
6 7 8
```

a

```
1 2 3
4   5
6 7 8
```

o caminho mais curto tem comprimento 2.

Input

A entrada é constituída por duas linhas. A primeira representa a configuração inicial, representado a segunda a configuração final desejada para o tabuleiro. Em ambas as linhas os tabuleiros são representados por sequências de dígitos numerados de 1 a 8 representando a posição das peças, linha após a linha. O zero representa o espaço em branco. Por exemplo, o tabuleiro acima é representado pela sequência: 123405678.

Todas as instâncias dadas terão solução única de comprimento pequeno, ie., que não excede 12

Output

A sequência de configurações desde a configuração inicial à configuração final (ambas inclusive), cada uma delas separadas por uma linha em branco. No fim deverá ser também apresentado um inteiro representando o tamanho do caminho encontrado (o mais curto).

Sample Input 1

```
023145678
123405678
```

Sample output 1

```
 23
145
678

123
 45
678

123
4 5
678

2
```

Sample input 2

```
123456780
436718520
```

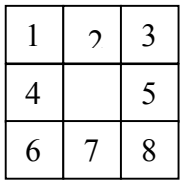
Sample output 2

```
123
456
78

123
456
7 8

123
4 6
758

1 3
426
758

 13
426
758

413
 26
758

413
726
 58

413
726
5 8

413
7 6
528

4 3
716
528

43
```

```
716
528

436
71
528

436
718
52

12
```

Appendix: Small instances of the 8-puzzle
// Leia acima a versão Portuguesa

The 8-puzzle is based on a board with 3x3 positions and 8 sliding tiles numbered from 1 to 8. The following figure shows a possible configuration for the board.



Any tile can be moved to an adjacent position as long as this destination position is empty. The program should return the successive configurations required for achieving the final configuration starting from the initial one, in the best possible case, i.e., in the shortest path.

For example, for transforming the config

```
  2 3
1 4 5
6 7 8
```

in

```
1 2 3
4   5
6 7 8
```

the length of the shortest path is 2.

Input
The input has two lines. The first line represents the initial configuration of the board while the second line denotes the goal configuration. In both lines, the board is represented by a sequence of digits from 1 to 8, representing tiles, row after row. A zero stands for the empty position. For example, the above board is represented by the sequence 123405678.

All given instances will have a unique solution, whose length will not exceed 12,

Output
The sequence of configurations from the initial configuration to the final one (both inclusive), with a blanc line separating each one of them. At the end of this sequence, a positive integer with the length of the shortest path found should be presented.

Sample Input 1
```
023145678
123405678
```

Sample output 1

```
 23
145
678

123
 45
678

123
4 5
678

2
```

Sample input 2

```
123456780
436718520
```

Sample output 2

```
123
456
78

123
456
7 8

123
4 6
758

1 3
426
758

 13
426
758

413
 26
758

413
726
 58

413
726
5 8

413
7 6
528

4 3
716
528

43
```

```
716
528

436
71
528

436
718
52

12
```

Appendix B: JUnit Assertions

0. Here is a list of the available assertions in JUnit 4.0. Possible arguments are given within parenthesis.

assertTrue
(boolean) Reports an error if boolean is false
(String, boolean) Adds error String to output

assertFalse
(bolean) Reports an error if boolean is true
(String, boolean) Adds error String to output

assertNull
(Object) Reports an error if object is not null
(String, Object) Adds error String to output

assertNotNull
(Object) Reports an error if object is null
(String, Object) *A*dds error String to output

assertSame
(Object, Object) Reports error if two objects are not identical
(String, Object, Object) Adds error String to output

assertNotSame
(Object, Object) Reports error if two objects are identical
(Styring, Object, Object) Adds error String to output

assertEquals
(Object, Object) Reports error if two objects are not equal
(String, Object, Object) Adds error String to output
(String, String) Reports delta between two strings if the two strings are not equal
(String, String, String) Adds error String to output
(boolean, boolean) Reports error if the two booleans are not equal
(String, boolean, boolean) Adds error String to output
(byte,byte) Reports error if the two bytes are not equal
(String, byte, byte) Adds error String to output
(char, char)Reports error if two chars are not equal
(String, char, char) Adds error String to output
(short, short) Reports error if two shorts are not equal
(String, short, short) Adds error String to output
(int, int) Reports error if two ints are not equal
(String, int, int) Adds error String to output
(long, long) Reports error if two longs are not equal
(String, long, long) Adds error String to output
(float, float, float) Reports error if the first two floats are not within range specified by third float
(String, float, float, float) Adds error String to output
(double, double, double) Reports error if the first two doubles are not within range specified by third double
(String, double, double, double) Adds error String to output
(object[], object[]); Reports error if either the legnth or element of each array are not equal. *New to JUnit 4*

(String, object[], object[]) Adds error String to output.
.

1. Besides that, you can use the arguments of annotation @Test for testing important behaviour of your code, such as Exception throwing, or code running time.

2. For example, you can write a test for checking whether the appropriated Exception is actually being thrown each time a method is invoked. Such a test can be given by:

```
@Test (expected=CloneNotSupportedException.class)
public void testCloneNotSupportedException() throws CloneNotSupportedException {
        Integer I = new Integer(3151);
        Integer J = (Integer) I.clone();
}
```

3. Also, we can make sure that your code does not take more than a certain amount of time to execute. For instance, you can make sure that solving a given instance of our problem does not take more than 1s:

```
@Test (timeout=1000)
public void testEficientSolve() {
    BestFirst s = new BestFirst();
    Iterator<BestFirst.State> it = s.solve(new Board("023145678"), new Board("123405678");
}
```

4. Learn more on JUnit at http://www.junit.org