The AIMA java classes

- It is a java class package that allows to define and solve search problems
- It implements some of the algorithms explained in the course
  - Uninformed search: Breadth First search, Deep First search, Iterative Deepening Search
  - Heuristic search: A^*, IDA^*
  - Local Search: Hill Climbing, Simulated Annealing
- The implementation uses genericity to separate the representation of the problem from the search algorithms

Defining the problem

To define a problem you must define and instantiate some java classes

- The definition of the implementation of the state and the search operators defined in the problem
- The definition of the successor function that gives all the states accessible from a given one (`aima.search.framework.SuccessorFunction`)
- The definitions of a function that returns if a state is the goal state (`aima.search.framework.GoalTest`)
- The definition of the heuristic function (`aima.search.framework.HeuristicFunction`)

The state class

- Its a class independent from the AIMA classes
- It must have one or more constructor functions that generate the object representing the initial state
- It must implement the search operators of the problem as functions able to transform a state in another
- Its advisable to implement also test functions that return if a search operator can be applied to a state.
- Some other auxiliary functions could be implemented in order to help to the implementation of the other classes

The successor function class

- This class has to implement the class `aima.search.framework.SuccessorFunction`
- It contains only the function `public List getSuccessors(Object aState)`
- This functions generates a list with all the accessible states from the state received as parameter
- This list contains pairs of elements, the first one is a string that describes the applied operator, the other is the resulting state.
- The strings in this pairs are the ones that the search algorithm uses to describe the path of the solution
- This class uses the functions implemented in the class that defines the state.

The goal state class

- This class has to implement `aima.search.framework.GoalTest`
- It contains only the function `public boolean isGoalState(Object aState)`
- This function returns true when the state received as parameter is the goal state

The heuristic function class

- This class has to implement
  `aima.search.framework.HeuristicFunction`
- It contains only the function
  `public int getHeuristicValue(Object n)`
- This functions has to return the value of the heuristic function (h')
- Obviously this functions depends on the problem

Example: the 8 puzzle

- Defined inside the package `aima.search.eightpuzzle`
- You can find the 4 classes needed to solve the problem:
  - `EightPuzzleBoard`, represents the board (an array with 9 positions, numbers from 0 to 8, 0 represents the blank tile)
  - `ManhattanHeuristicFunction`, implement an heuristic function (sum of Manhattan distance of tiles positions)
  - `EightPuzzleSuccessorFuncion`, implements the function generating all the accessible states from a given one (all possible movements of the blank tile)
  - `EightPuzzleGoalTest`, defines the function that test for the goal state
- The class `aima.search.demos.EightPuzzleDemo` implements some functions that solve the problems using different search algorithms

Example: Problem 15th

- Defined in the package `IA.probIA15`
- You can find the 4 classes needed to solve the problem:
  - `ProbIA15Board`, implementation of the board (an array of 5 positions with a given configuration)
  - `ProbIA15HeuristicFunction`, implements the heuristic function (number of white tiles)
  - `ProbIA15SuccessorFunction`, implements the function generating all the accessible states from a given one (jump and shift)
  - `probIA15GoalTest`, defines the function that test for the goal state
- The class `IA.probIA15.ProbIA15Demo` implements some functions that solve the problems using different search algorithms

IA.probIA15.ProbIA15Board

```
1   public class ProbIA15Board {
2    /* Strings used in the trace */
3    public static String DESP_DERECHA = "Desplazar Derecha";
4    ...
5    private char [] board = {'N','N','B','B','O'};
6
7   /* Constructor */
8    public ProbIA15Board(char[] b) {
9     for(int i=0;i<5;i++) board[i]=b[i];
10   }
11
12   /* Auxiliary functions */
13
14   public char [] getConfiguration(){
15     return board;
16   }
17
18  /* Get the tile in position i */
19   private char getPos(int i){
20    return(board[i]);
21   }
22
23  /* Position of the blank tile */
24   public int getGap(){
25    int v=0;
26
27    for (int i=0;i<5;i++) if (board[i]=='O') v=i;
28    return v;
29   }
30   ...
```

IA.probIA15.ProbIA15Board

```
31   /* Functions that test the conditions of the operators */
32    public boolean puedeDesplazarDerecha(int i) {
33     if (i==4) return(false);
34     else return(board[i+1]=='O');
35    }
36
37    ...
38
39   /* Functions implementing the operators*/
40    public void desplazarDerecha(int i){
41     board[i+1]=board[i];
42     board[i]='O';
43    }
44
45    ...
46
47   /* Function that tests for the goal state */
48    public boolean isGoal(){
49     boolean noblanco=true;
50
51     for(int i=0;i<5;i++) noblanco=noblanco && (board[i]!='B');
52     return noblanco;
53    }
```

IA.probIA15.ProbIA15HeuristicFunction

```
 1  package IA.probIA15;
 2
 3  import java.util.Comparator;
 4  import java.util.ArrayList;
 5  import IA.probIA15.ProbIA15Board;
 6  import aima.search.framework.HeuristicFunction;
 7
 8  public class ProbIA15HeuristicFunction implements HeuristicFunction{
 9
10    public int getHeuristicValue(Object n) {
11     ProbIA15Board board=(ProbIA15Board)n;
12     char [] conf;
13     int sum=0;
14
15     conf=board.getConfiguration();
16     for(int i=0;i<5;i++) if (conf[i]=='B') sum++;
17
18     return (sum);
19   }
20  }
```

IA.probIA15.ProbIA15SuccessorFunction

```
1   package IA.probIA15;
2
3   import aima.search.framework.Successor;
4   import aima.search.framework.SuccessorFunction;
5   import IA.probTSP.ProbTSPHeuristicFunction;
6
7   public class ProbIA15SuccessorFunction implements SuccessorFunction {
8
9    public List getSuccessors(Object aState) {
10    ArrayList retVal= new ArrayList();
11    ProbIA15Board board=(ProbIA15Board) aState;
12
13    for(int i=0;i<5;i++){
14     if (board.puedeDesplazarDerecha(i)){
15      ProbIA15Board newBoard= new ProbIA15Board(board.getConfiguration());
16      newBoard.desplazarDerecha(i);
17      retVal.add(new Successor(new String(ProbIA15Board.DESP_DERECHA+" "+
18                  newBoard.toString()),newBoard));
19     }
20     ...
21    }
22    return (retVal);
23   }
24  }
```

IA.probIA15.ProbIA15GoalTest

```
1  package IA.probIA15;
2
3  import java.util.ArrayList;
4  import aima.search.framework.GoalTest;
5
6  public class ProbIA15GoalTest implements GoalTest{
7
8      public boolean isGoalState(Object aState) {
9          boolean goal;
10         ProbIA15Board board= (ProbIA15Board) aState;
11
12         return board.isGoal();
13     }
```

How to solve a problem with a search algorithm

You can see the procedure from the examples, you have to:

- Define an object of type `Problem` that receives as a parameter a set of objects representing the initial state, the generator of successor states function, the goal state function and, if an informed search algorithm is used, the heuristic function
- Define an object of type `Search` as an instance of the class of the algorithm we are going to use
- Define an object of type `SearchAgent` that receives the objects `Problem` and `Search`
- The functions `printActions` and `printInstrumentation` prints the solution search path and some statistical information depending on the algorithm

IA.probIA15.ProbIA15Demo

```
1   private static void IAP15BreadthFirstSearch(ProbIA15Board IAP15) {
2
3      Problem problem =  new Problem(IAP15,
4                                     new ProbIA15SuccessorFunction(),
5                                     new ProbIA15GoalTest());
6      Search search =  new BreadthFirstSearch(new TreeSearch());
7      SearchAgent agent = new SearchAgent(problem,search);
8   ...
9
10  }
11
12  private static void IAP15AStarSearchH1(ProbIA15Board TSPB) {
13    Problem problem =  new Problem(TSPB,
14                                   new ProbIA15SuccessorFunction(),
15                                   new ProbIA15GoalTest(),
16                                   new ProbIA15HeuristicFunction());
17    Search search =  new AStarSearch(new GraphSearch());
18    SearchAgent agent = new SearchAgent(problem,search);
19    ...
20  }
```

Running the examples - 8 puzzle

You can find this demo inside the package `aima.search.demos`

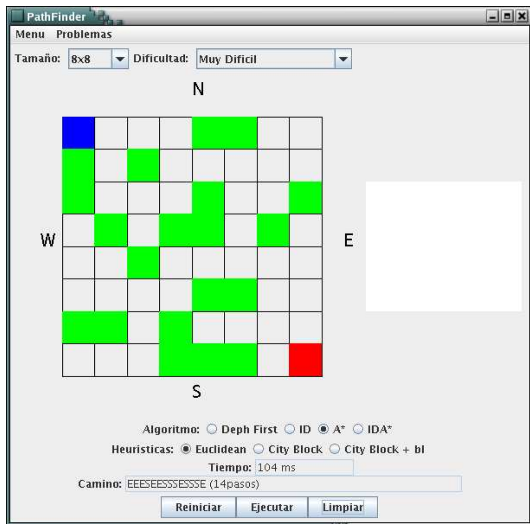
- If you run the class `aima.search.demos.EightPuzzleDemo` the problem is solved using the following algorithms:
  - Limited depth first search
  - Iterative deepening search
  - Best first search (2 heuristics)
  - A* search (2 heuristics)

Running the examples - Problem 15th and PathFinder

You can find this demos inside the packages `IA.probIA15` and `IA.probPathFinder`

- `IA.probIA15.ProbIA15Demo`, breadth, limited depth, iterative deepening, A^* and IDA^* with two heuristic functions
- `IA.probPathFinder.ProbPathFinderJFrame`, shows you a GUI that allows to choose different problems, search algorithms and heuristic functions
    - The problems is to find a path from the blue square to the red square

Running the examples - PathFinder



Running the examples - PathFinder

Notice that:

- The uninformed algorithms fail when the size of the problem increases
- The heuristic functions has a crucial influence on the performance of the informed algorithms
- IDA* beats A* because uses less memory, A* fails when an specific size of problem is reached
- Some configurations of the problem (look at the menus) can not be solve by any algorithm in a reasonable time (specific problem knowledge is needed)

Example: The Travelling Salesman problem

- Defined inside the package `IA.probTSP`
- You can find 4 classes defining the problem:
  - `ProbTSPBoard`, implementation of the problem (an array of size n representing the path that visits the n cities)
  - `ProbTSPHeuristicFunction`, implement an heuristic function (length of the path)
  - `ProbTSPSuccessorFunction`, implements the function generating all the accessible states from a given one (All possible two cities interchange)
  - `probTSPGoalTest`, this functions always returns false (this is because the goal state is unknown)
- The class `ProbTSPJFrame`, the main program, it shows a GUI that allows to run the hill climbing and simulated annealing algorithms with a random TSP problem

Examples: The Travelling Salesman problem

- You can change the number of cities
- Each panel shows the solution of a TSP problem using the hill climbing and simulated annealing algorithms
- Notice that frequently the simulated annealing algorithm finds a better solution
- You can adjust the parameters of the simulated annealing algorithm using the GUI

Examples: The Travelling Salesman problem

Travelling Salesman Problem

Menu

Num Ciudades: 10 15 20 25 30 35 40

Hill Climbing

```
Intercambio 2 7 Coste(275) ---> |0|1|7|3|4|5|6|
Intercambio 4 9 Coste(237) ---> |0|1|7|3|9|5|6|
Intercambio 0 5 Coste(213) ---> |5|1|7|3|9|0|6|
Intercambio 5 9 Coste(197) ---> |5|1|7|3|9|4|6|
```

Simulated Annealing

```
Intercambio 2 0 Coste(213) ---> |0|6|2|8|4|5|1|7
Intercambio 4 0 Coste(197) ---> |4|6|2|8|0|5|1|7
Intercambio 1 3 Coste(197) ---> |4|8|2|6|0|5|1|7
Intercambio 2 0 Coste(211) ---> |2|8|4|6|0|5|1|7
Intercambio 5 3 Coste(208) ---> |2|8|4|5|0|6|1|7
Intercambio 2 0 Coste(206) ---> |4|8|2|5|0|6|1|7
Intercambio 4 3 Coste(206) ---> |4|8|2|0|5|6|1|7
Intercambio 0 2 Coste(217) ---> |2|8|4|0|5|6|1|7
Intercambio 4 2 Coste(230) ---> |2|8|5|0|4|6|1|7
Intercambio 4 5 Coste(231) ---> |2|8|5|0|6|4|1|7
Intercambio 6 5 Coste(228) ---> |2|8|5|0|6|1|4|7
Intercambio 6 5 Coste(231) ---> |2|8|5|0|6|4|1|7
Intercambio 4 1 Coste(236) ---> |2|6|5|0|8|4|1|7
Intercambio 9 8 Coste(233) ---> |2|6|5|0|8|4|1|7
Intercambio 6 7 Coste(215) ---> |2|6|5|0|8|4|7|1
Intercambio 7 5 Coste(214) ---> |2|6|5|0|8|1|7|4
Intercambio 2 1 Coste(227) ---> |2|5|6|0|8|1|7|4
Intercambio 5 0 Coste(208) ---> |1|5|6|0|8|2|7|4
Intercambio 1 0 Coste(206) ---> |5|1|6|0|8|2|7|4
Intercambio 1 0 Coste(208) ---> |1|5|6|0|8|2|7|4
Intercambio 0 1 Coste(206) ---> |5|1|6|0|8|2|7|4
Intercambio 8 9 Coste(233) ---> |5|1|6|0|8|2|7|4
Intercambio 7 5 Coste(217) ---> |5|1|6|0|8|4|7|2
Intercambio 2 0 Coste(202) ---> |6|1|5|0|8|4|7|2
Intercambio 4 2 Coste(218) ---> |6|1|8|0|5|4|7|2
Intercambio 1 7 Coste(194) ---> |6|2|8|0|5|4|7|1
Intercambio 8 0 Coste(174) ---> |3|2|8|0|5|4|7|1

Final state = |3|2|8|0|5|4|7|1|6|9| Coste= 174
```

Parametros Annealing

Num It: 10000 9000 8000 7000 6000 5000 4000 3000 2000 1000

K: 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

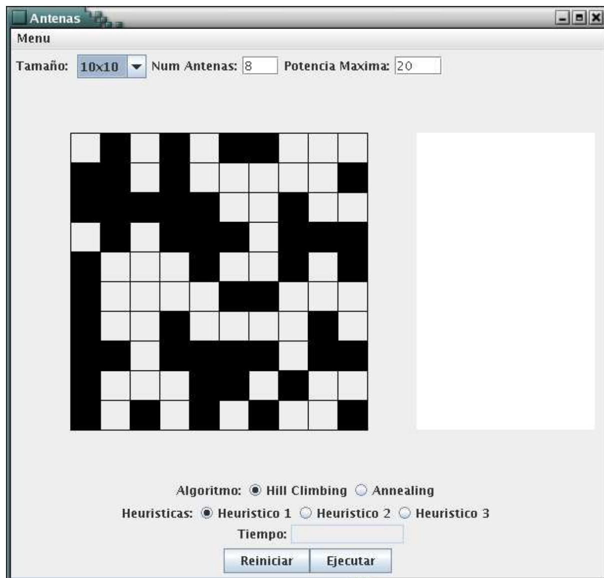
Lambda: 0,01

Ejecutar Prob Aleatorio | Ejecutar Prob Específico | Semilla: 1

Examples: Mobile phone antennas problem

- Defined inside the package `IA.probAntenas`
- You can find 4 classes defining the problem:
  - `ProbAntenasBoard`, implementation of the problem (a matrix of size N×N representing a city map and an array with the information of the antennas)
  - `ProbAntenasHeuristicFunction`, `ProbAntenasHeuristicFunction2`, `ProbAntenasHeuristicFunction3`, implement some heuristic functions
  - `ProbAntenasSuccessorFunction`, implements the function generating all the accessible states from a given one (Move an antenna, increase and decrease the power of an antenna)
  - `probAntenasGoalTest`, this functions always returns false
- The class `ProbAntenasJFrame`, the main program, it shows a GUI that allows to run the hill climbing and simulated annealing algorithms with a random problem

Examples: Mobile phone antennas problem



Examples: Mobile phone antennas problem

Notice that:

- If you change the heuristic function, you change the type of the solution the algorithm looks for (priority to cover maximum area, maximize the number of antennas used, penalize the overlapping between antennas, ...)
- The Simulated Annealing algorithm is more tolerant to poor heuristic functions and bad initial states (heuristic 2)
- The heuristic 2 shows also that the Simulated Annealing solutions are more stable that ones from the Hill Climbing algorithm, this means that the former has less probability to fall into local minima

The classes of the search algorithms (uninformed)

- `aima.search.uninformed`
  - `BreadthFirstSearch`, Breadth First Search, it has a parameter of type `TreeSeach`
  - `DepthLimitedSearch`, Depth Limited Search, it has an integer parameter meaning the bound for the depth of the search
  - `IterativeDeepeningSearch`, Iterative Deepening Search, it has no parameters

The classes of the search algorithms (informed)

- `aima.search.informed`
  - `AStarSearch`, A* search, it has a parameter of type `GraphSearch`
  - `IterativeDeepeningAStarSearch`, IDA* search, it has no parameters
  - `HillClimbingSearch`, Hill Climbing search, it has no parameters
  - `SimulatedAnnealingSearch`, Simulated Annealing search, it has 4 parameters:
    - Maximum number of iterations,
    - Number of iterations for each temperature decreasing step
    - The parameters k and λ that determine the behaviour of the temperature function

The classes of the search algorithms (Simulated Annealing)

- The temperature function is $\mathcal{F}(T) = k \cdot e^{-\lambda \cdot T}$
- The greater is the value of k the greater the temperature is
- The greater is the value of λ the faster is the decrease of the value of the function when the temperature decreases
- The value of T is calculated as a function of the number of iterations and the number of iterations for each temperature step
- The function that calculates the non acceptance probability of a state is:

$$\text{non acceptance probability} = \frac{1}{1 + e^{\left(\frac{\Delta E}{\mathcal{F}(T)}\right)}}$$

Where ΔE is the difference of energy between the actual state and the successor state and T is the current temperature