Las clases de AIMA

- Conjunto de clases en java que permiten definir problemas de búsqueda
- Implementan varios de los algoritmos vistos en clase
  - Búsqueda ciega: Anchura, Profundidad, Profundidad iterativa
  - Búsqueda heurística: A*, IDA*
  - Búsqueda local: Hill Climbing, Simulated Annealing
- Son un conjunto de clases genéricas que separan la representación del problema de los algoritmos de búsqueda

Definición de un problema

La definición de un problema requiere la instanciación de una serie de clases

- La definición de la representación del estado y los operadores del problema
- La definición de la función generadora de estados accesibles (`aima.search.framework.SuccessorFunction`)
- La definición de la función que determina si se ha llegado al estado final (`aima.search.framework.GoalTest`)
- La definición de la función heurística (`aima.search.framework.HeuristicFunction`)

La representación del estado

- Ha de ser una clase independiente
- Su constructor ha de poder generar la representación del estado
- Ha de tener funciones que transformen el estado según los posibles operadores a utilizar
- Es conveniente que haya funciones que indiquen si un operador se puede aplicar sobre un estado
- Se pueden incluir las funciones auxiliares que se crean necesarias para el resto de clases

La función generadora de estados accesibles

- Implementa la clase `aima.search.framework.SuccessorFunction`
- Ha de implementar la función `public List getSuccessors(Object aState)`
- Esta función genera la lista de los estados accesibles a partir del que recibe como parámetro.
- Esta lista contiene pares de elementos que consisten en un string que representa la operación que se ha aplicado y el estado sucesor resultante.
- Los string que se ponen en esos pares serán los que se escriban como resultado de la búsqueda
- Esta función necesitará usar las funciones declaradas en la clase que define el problema (representación del estado, operadores)

La función que identifica el estado final

- Implementa la clase `aima.search.framework.GoalTest`
- Ha de implementar la función `public boolean isGoalState(Object aState)`
- Esta función ha de retornar cierto cuando un estado sea final

La función heurística

- Implementa la clase `aima.search.framework.HeuristicFunction`
- Ha de implementar la función `public double getHeuristicValue(Object n)`
- Esta función ha de retornar el valor de la función heurística (la h)
- Las características de la función dependerán del problema.

Ejemplo: 8 puzzle

- Definido en el package `aima.search.eightpuzzle`
- Tenemos las 4 clases que representan el problema:
  - `EightPuzzleBoard`, representación del tablero (un vector de 9 posiciones, números del 0 al 8, el 0 es el blanco)
  - `ManhattanHeuristicFunction`, implementa la función heurística (suma de distancia manhattan de cada ficha)
  - `EightPuzzleSuccessorFuncion`, implementa la función que genera los estados accesibles desde uno dado (posibles movimientos del blanco)
  - `EightPuzzleGoalTest`, define la función que identifica el estado final
- La clase `aima.search.demos.EightPuzzleDemo` tiene las funciones que permiten solucionar el problema con distintos algoritmos

Ejemplo: Problema 15

- Definido en el package `IA.probIA15`
- Tenemos las 4 clases que representan el problema:
  - `ProbIA15Board`, representación del tablero (un vector de 5 posiciones con la configuración inicial de fichas)
  - `ProbIA15HeuristicFunction`, implementa la función heurística (número de piezas blancas)
  - `ProbIA15SuccessorFunction`, implementa la función que genera los estados accesibles desde uno dado (saltos y desplazamientos)
  - `probIA15GoalTest`, define la función que identifica el estado final.
- La clase `IA.probIA15.ProbIA15Demo` permite ejecutar los algoritmos de búsqueda ciega y heurística

IA.probIA15.ProbIA15Board

```
1  public class ProbIA15Board {
2   /* Strings para la traza */
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  /* Funciones auxiliares */
13
14  public char [] getConfiguration(){
15    return board;
16  }
17
18 /* Ficha en la posicion i */
19  private char getPos(int i){
20   return(board[i]);
21  }
22
23 /* Posicion del blanco */
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  /* Funciones para comprobar los movimientos */
32   public boolean puedeDesplazarDerecha(int i) {
33    if (i==4) return(false);
34    else return(board[i+1]=='O');
35   }
36   ...
37
38  /* Funciones que implementan los operadores*/
39   public void desplazarDerecha(int i){
40    board[i+1]=board[i];
41    board[i]='O';
42   }
43   ...
44
45  /* Funcion que comprueba si es el estado final */
46   public boolean isGoal(){
47    boolean noblanco=true;
48
49    for(int i=0;i<5;i++) noblanco=noblanco && (board[i]!='B');
50    return noblanco;
51   }
```

IA.probIA15.ProbIA15HeuristicFunction

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

IA.probIA15.ProbIA15SuccessorFunction

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

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     }
```

Como ejecutar los algoritmos para un problema

El funcionamiento lo podéis ver en los ejemplos, lo que se ha de hacer es:

- Definir un objeto `Problem` que recibe un objeto que representa el estado inicial, la funcion generadora de sucesores, la funcion que prueba el estado final y, si se va a utilizar un algoritmo de busqueda informada, la función heurística
- Definir un objeto `Search` que sea una instancia del algoritmo que se va a usar
- Definir un objeto `SearchAgent` que recibe los objetos `Problem` y `Search`
- Las funciones `printActions` y `printInstrumentation` permiten imprimir el camino de búsqueda y la información de la ejecución del algoritmo de búsqueda

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 }
```

Ejecución de los ejemplos - 8 puzzle

El programa se encuentra en `aima.search.demos`

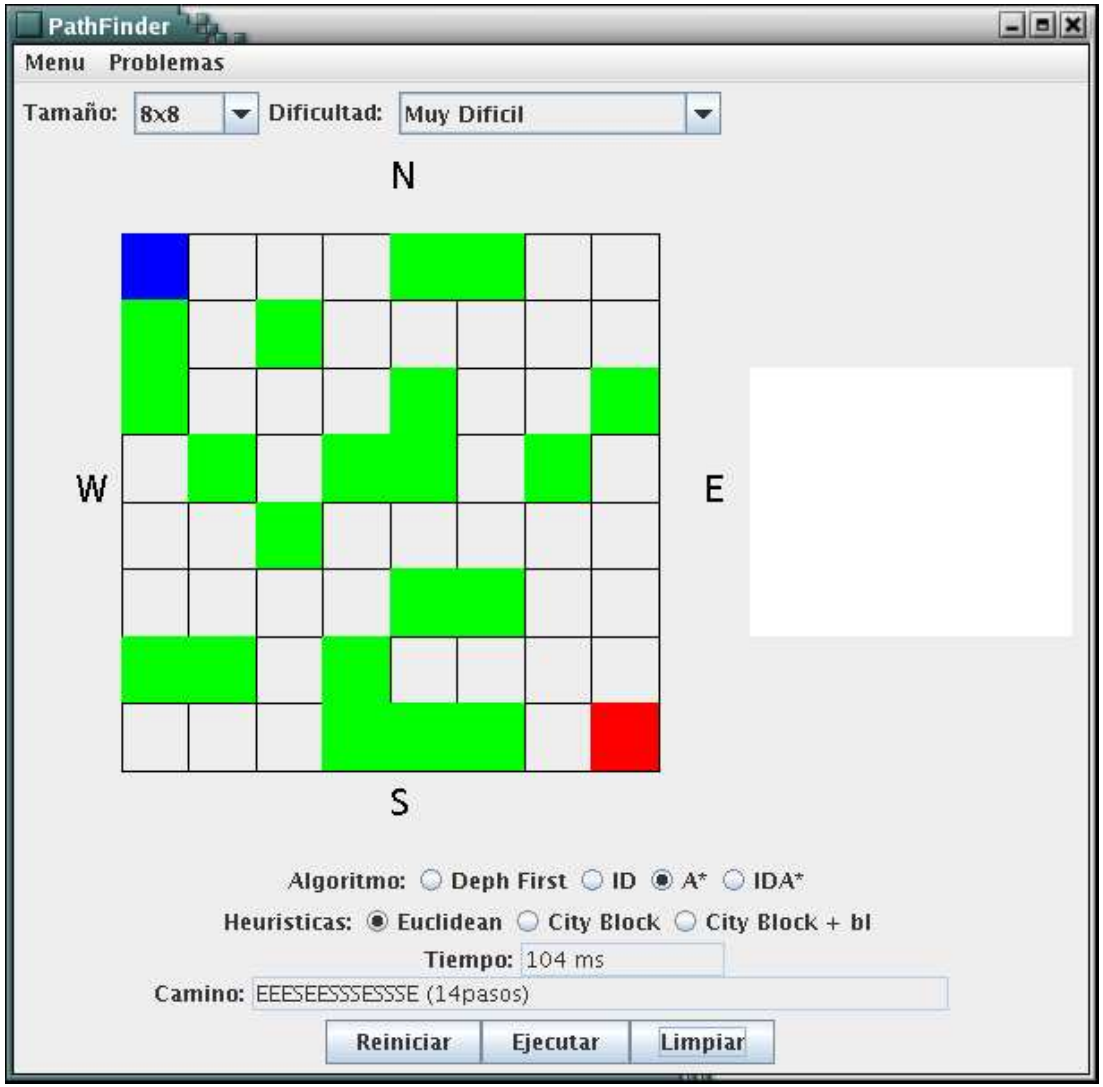
- Ejecutando la clase `aima.search.demos.EightPuzzleDemo` se resuelve mediante los algoritmos:
  - Profundidad limitada
  - Profundidad iterativa
  - Best first (2 heurísticos)
  - A* (2 heurísticos)

Ejecución de los ejemplos - Problema 15 y PathFinder

Los programas se encuentran en `IA.probIA15` y `IA.probPathFinder`

- `IA.probIA15.ProbIA15Demo`, anchura, profundidad limitada, profundidad iterativa, A* e IDA* con dos heurísticas
- `IA.probPathFinder.ProbPathFinderJFrame`, presenta un interfaz a partir del cual se puede seleccionar el problema, el algoritmo y varias heurísticas
  - El problema consiste en encontrar un camino entre la posición azul y la roja

Ejecución de los ejemplos - PathFinder



Ejecución de los ejemplos - PathFinder

Cosas a observar:

- Los algoritmos exhaustivos dejan de ser efectivos a partir de tamaños pequeños
- La heurística influye mucho en la eficiencia de los algoritmos informados
- IDA* gana a A* en tiempo dado que gasta menos memoria, a partir de cierto tamaño A* deja de ser competitivo
- Hay configuraciones de problemas (ver menús) que no se pueden resolver con ningún algoritmo en un tiempo razonable (hace falta un conocimiento mas especializado)

Ejemplo: el viajante de comercio

- Definido en el package `IA.probTSP`
- Tenemos las 4 clases que representan el problema:
  - `ProbTSPBoard`, representación del tablero (un vector de n posiciones que representan la secuencia de recorrido entre las n ciudades)
  - `ProbTSPHeuristicFunction`, implementa la función heurística (suma del recorrido)
  - `ProbTSPSuccessorFunction`, implementa la función que genera los estados accesibles desde uno dado (todos los posibles intercambios de 2 ciudades)
  - `ProbTSPSuccessorFunctionSA`, implementa la función que genera los estados accesibles desde uno dado optimizado para Simulated Annealing
  - `probTSPGoalTest`, define una función que siempre retorna falso como prueba de estado final (en este caso desconocemos el estado final)

Función de generación de sucesores

- En el caso de Hill Climbing la generación de sucesores necesita ver todos los sucesores para escoger el mejor
- En el caso de Simulated Annealing su estrategia no necesita ver todos los sucesores ya que se elige uno al azar.
- Desde el punto de vista de la eficiencia para Simulated Annealing es mejor generar un sucesor aplicando un operador seleccionado al azar que generar todos los sucesores y escoger uno al azar entre ellos

IA.probTSP.ProbTSPSuccessorFunction (HC)

```
1  public List getSuccessors(Object aState) {
2     ArrayList                retVal = new ArrayList();
3     ProbTSPBoard             board  = (ProbTSPBoard) aState;
4     ProbTSPHeuristicFunction TSPHF  = new ProbTSPHeuristicFunction();
5
6      for (int i = 0; i < board.getNCities(); i++) {
7         for (int j = i + 1; j < board.getNCities(); j++) {
8             ProbTSPBoard newBoard = new ProbTSPBoard(board.getNCities(),
9                                board.getPath(), board.getDists());
10
11            newBoard.swapCities(i, j);
12
13            int    v = TSPHF.getHeuristicValue(newBoard);
14            String S = ProbTSPBoard.INTERCAMBIO + " " + i + " " + j
15                     + " Coste(" + v + ") ---> " + newBoard.toString();
16
17            retVal.add(new Successor(S, newBoard));
18        }
19    }
20
21    return retVal;
22 }
```

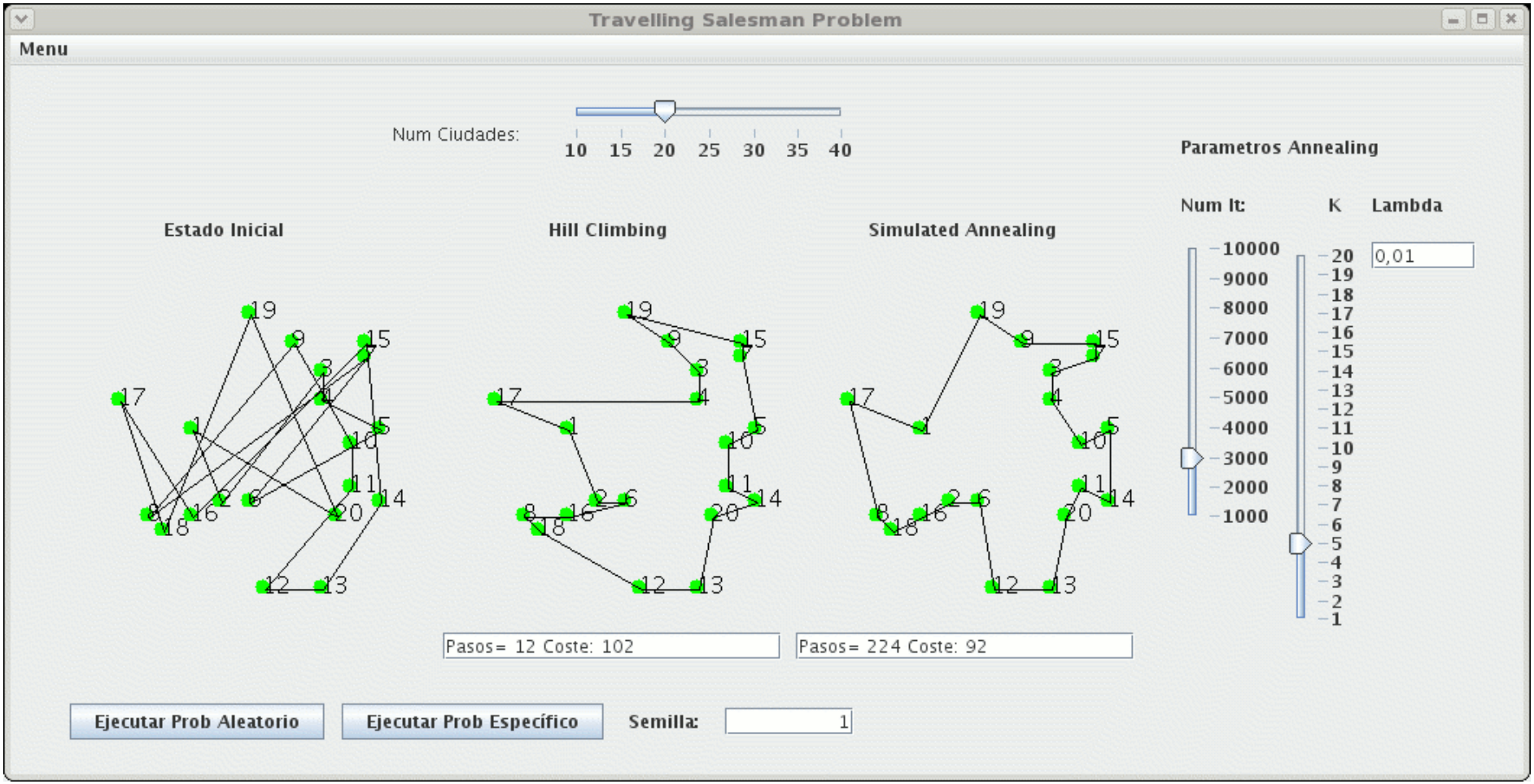
IA.probTSP.ProbTSPSuccessorFunctionSA (SA)

```
1  public List getSuccessors(Object aState) {
2     ArrayList                retVal = new ArrayList();
3     ProbTSPBoard             board  = (ProbTSPBoard) aState;
4     ProbTSPHeuristicFunction TSPHF  = new ProbTSPHeuristicFunction();
5     Random myRandom=new Random();
6     int i,j;
7
8    i=myRandom.nextInt(board.getNCities());
9
10   do{
11          j=myRandom.nextInt(board.getNCities());
12   } while (i==j);
13
14
15   ProbTSPBoard newBoard = new ProbTSPBoard(board.getNCities(), board.getPath(),
16                board.getDists());
17
18   newBoard.swapCities(i, j);
19
20   int    v = TSPHF.getHeuristicValue(newBoard);
21   String S = ProbTSPBoard.INTERCAMBIO + " " + i + " " + j
22                    + " Coste(" + v + ") ---> "+ newBoard.toString();
23
24   retVal.add(new Successor(S, newBoard));
25
26   return retVal;
27 }
```

Ejemplo: el viajante de comercio

- Podéis introducir el numero de ciudades
- En cada panel aparece la ejecución del problema del viajante de comercio con Hill Climbing y Simulated Annealing
- Podréis comprobar que frecuentemente la solución que da el Simulated Annealing es mejor
- Evidentemente para cada tamaño de problema habría que reajustar los parámetros del Simulated Annealing

Ejemplo: el viajante de comercio



Las clases de los algoritmos (informados)

- `aima.search.informed`
  - `AStarSearch`, búsqueda A*, recibe como parámetro un objeto `GraphSearch`
  - `IterativeDeepeningAStarSearch`, búsqueda IDA*, sin parámetros
  - `HillClimbingSearch`, búsqueda Hill Climbing, sin parámetros
  - `SimulatedAnnealingSearch`, búsqueda SimulatedAnnealing, recibe 4 parámetros:
    - El número máximo de iteraciones,
    - El número de iteraciones por cada paso de temperatura
    - Los parámetros k y λ que determinan el comportamiento de la función de temperatura

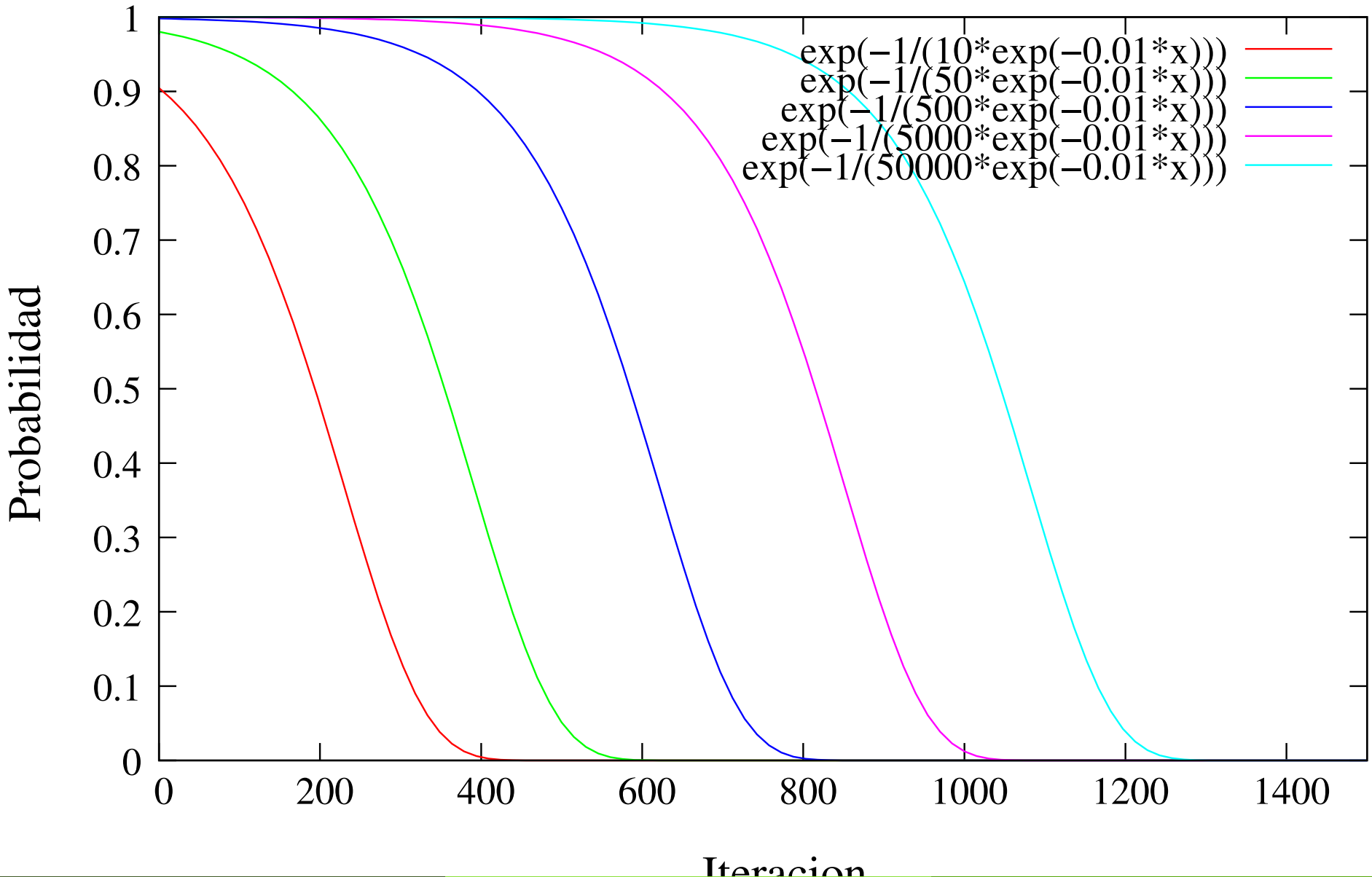
Las clases de los algoritmos (Simulated Annealing)

- La función de temperatura es $\mathcal{F}(T) = k \cdot e^{-\lambda \cdot T}$
- Cuanto mayor es k tarda en comenzar a decrecer la función
- Cuanto mayor es λ mas rápido desciende la función
- El valor de T se obtiene a partir del numero de iteraciones realizadas
- La función que determina la probabilidad de aceptación de un estado peor es:

$$\text{Probabilidad de aceptación} = e^{(\frac{\Delta E}{\mathcal{F}(T)})}$$

Donde ΔE es la diferencia de energía entre el estado actual y el siguiente y T es el numero de iteración actual

Influencia de los parámetros (K)



Influencia de los parámetros (λ)