Las clases de AIMA

Javier Béjar

Inteligencia Artificial - 2025/2026 2Q

CS - FIB

Búsqueda con AIMA

- Clases java que permiten definir y resolver 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

La definición de un problema requiere la definición de cuatro 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`)

- Es una clase independiente de AIMA
- Su constructor genera un objeto que representa el 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

- Implementa `aima.search.framework.SuccessorFunction`
- Ha de implementar la función

```
public List getSuccessors(Object aState)
```

- Genera una lista con los estados accesibles a partir del que recibe como parámetro
- Contiene pares de elementos `<String, Estado>` (operación aplicada/estado sucesor)
- Los string de esos pares serán los que se escriban como resultado de la búsqueda (camino solución)
- Esta función usará las funciones declaradas en la clase que define el problema (representación del estado, operadores)

- Implementa la clase `aima.search.framework.GoalTest`
- Ha de implementar la función

```
public boolean isGoalState(Object aState)
```

- Esta función ha de retornar cierto cuando el estado que se reciba sea una solución

- Implementa `aima.search.framework.HeuristicFunction`
- Ha de implementar la función

```
public double getHeuristicValue(Object n)
```

- Esta función retorna el valor de la función heurística (la h)
- Las características de la función dependerán del problema.

Ejemplos

- Definido en el package `aima.search.eightpuzzle`
- Tenemos las 4 clases que representan el problema:
  - `EightPuzzleBoard`, representación del tablero (Vector de 9 posiciones, números del 0 al 8, 0 = 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` soluciona el problema con distintos algoritmos

- 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

```
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  ...
```

```
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  }
```

```java
package IA.probIA15;

import java.util.Comparator;
import java.util.ArrayList;
import aima.search.framework.HeuristicFunction;

public class ProbIA15HeuristicFunction implements HeuristicFunction{

  public double getHeuristicValue(Object n) {
   ProbIA15Board board=(ProbIA15Board)n;
   char [] conf;
   double sum=0;

   conf=board.getConfiguration();
   for(int i=0;i<5;i++) if (conf[i]=='B') sum++;

   return (sum);
 }
}
```

```
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 }
```

```java
package IA.probIA15;

import java.util.ArrayList;
import aima.search.framework.GoalTest;

public class ProbIA15GoalTest implements GoalTest{

    public boolean isGoalState(Object aState) {
        boolean goal;
        ProbIA15Board board= (ProbIA15Board) aState;

        return board.isGoal();
    }
```

Ejecución de la búsqueda

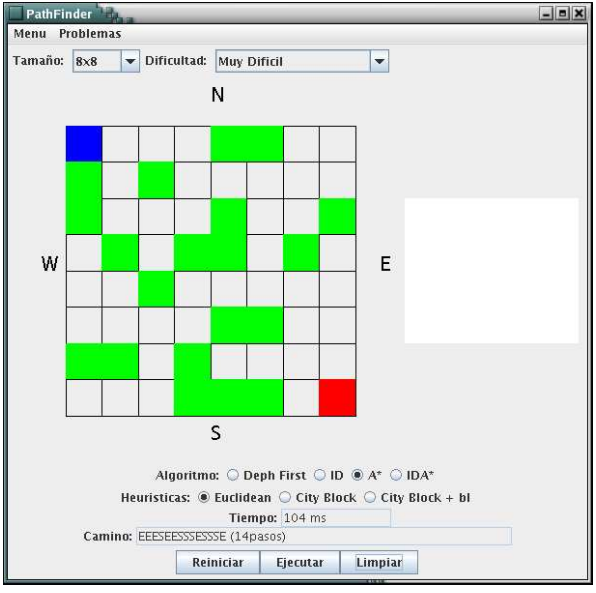
- Instanciar `Problem` usando el objeto que representa el estado inicial, la funcion generadora de sucesores, la funcion que prueba el estado final y, si se utiliza un algoritmo de busqueda informada, la función heurística
- Definir un objeto `Search` que sea una instancia del algoritmo que se va a usar
- 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

```
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 }
21
```

Demos

A través de la interfaz que se muestra cuando ejecutáis `java -jar aima.jar` podéis acceder a las demos de algoritmos de búsqueda heurística

- `EightPuzzle` (Profundidad limitada, iterativa, A* con dos heurísticos)
- `ProbIA15` (anchura, profundidad limitada, profundidad iterativa, A^* e IDA^* con dos heurísticas)
- `PathFinder` (permite elegir diferentes alogitmos y heuristicas)


PathFinder
Menu
Problemas
Tamaño:
8x8
Dificultad:
Muy Dificil
N
W
E
S
Algoritmo:
Deph First
ID
A*
IDA*
Heuristicas:
Euclidean
City Block
City Block + bl
Tiempo:
104 ms
Camino:
EEESEESSSESSSE (14pasos)
Reiniciar
Ejecutar
Limpiar

Aplicación práctica

- Tenemos un vector de monedas (5)
- Cada moneda presenta su cara o su cruz
- Partimos de una configuración cualquiera
- El objetivo es llegar a una configuración específica
- La restricción para que tenga solución es que la paridad del número de unos sea la misma en ambos estados
- El único operador es voltear a la vez dos monedas consecutivas (la del borde derecho se voltea con el izquierdo)
- La heurística es el número de diferencias entre el estado actual y la solución

- Código inicial en http://github.com/bejar/AIMAexample
- Podéis descargarlo como un `.zip` o desde terminal haciendo `git clone` + la direccion del fichero .git
- Para usar los fuentes tenéis dos opciones:
  - Crear un proyecto con Netbeans indicando que tenéis unos fuentes (siguiendo las indicaciones)
  - Usar un editor de textos y línea de comandos (compilar con `javac`, ejecutar con `java`)
- Si usáis Netbeans debéis indicarle al proyecto donde esta el fichero `AIMA.jar`, si usáis línea de comandos deberéis añadirlo a la variable `CLASSPATH`

- En el directorio `IA/ProbIA5` tenéis cuatro ficheros fuente:
  - `ProbIA5Board.java` (el estado)
  - `ProbIA5SuccesorFunction.java` (generadora de sucesores)
  - `ProbIA5HeuristicFunction.java` (heurística)
  - `ProbIA5GoalTest.java` (test de fin de búsqueda)
- En el directorio raíz tenéis un programa principal que permite ejecutar un problema

- Deberéis completar:
  - En `ProbIA5Board.java`: funciones del operador, función heurística, función estado final, función para hacer una copia del estado
  - `ProbIA5SuccesorFunction.java`: bucle que recorra todos los pares consecutivos y genere los estados sucesores (deberéis hacer copias del estado actual)
- Ejecutad el problema con A* e IDA*

Búsqueda local: TSP

- Definido en el package `IA.probTSP`
- Tenemos las 4 clases que representan el problema:
  - `ProbTSPBoard`: Vector de n posiciones que representa la secuencia de recorrido entre las n ciudades
  - `ProbTSPHeuristicFunction`: Suma del recorrido
  - `ProbTSPSuccessorFunction`: Genera los estados accesibles (intercambios de 2 ciudades)
  - `ProbTSPSuccessorFunctionSA`: Genera los estados accesibles optimizado para Simulated Annealing (un intercambio al azar)
  - `probTSPGoalTest`: Siempre retorna falso (en búsqueda local desconocemos el estado final)
- La demo usa hill climbing y el simulated annealing

- 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

```
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 }
```

```
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 }
```

Menu

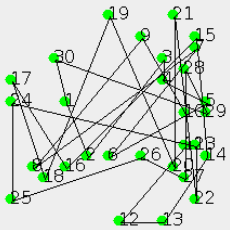
Ordenado

Animación

Num Ciudades:

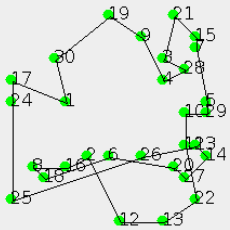
10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95100

Estado Inicial



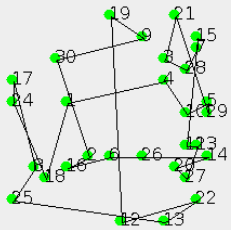
Coste=384

Hill Climbing



'asos= 19 Coste: 148 Tiempo: 64 ms

Simulated Annealing



'asos= 151 Coste: 196 Tiempo: 8 ms

Parametros Annealing

Num It: 1000

K: 5,

Lambda: 0,01

Ejecutar Prob Aleatorio

Ejecutar Prob Específico

Semilla: 1

En la documentación de la práctica (sec. 6.5 y 6.6) tenéis una explicación de experimentos realizados con este problema.

Podéis experimentar con el efecto de la solución inicial, el tamaño del problema y el algoritmo usado para solucionarlo.

La sección 6.6 explica los parámetros del annealing y su impacto en la forma que se realiza la búsqueda.

Podéis observar el efecto que tiene el usar diferentes parámetros en la demo y la evolución el valor de la función heurística haciendo doble click sobre la solución.

- `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