Trabajo TP6-1. Curso 2021-2022

Lenguajes de Reglas, Propagación de restricciones, y juegos

1. Objetivo de la práctica

El trabajo consta de tres tareas. ***La primera tarea*** tiene por objeto familiarizarse con en el desarrollo de **programas en CLIPS** tanto escribiendo código como depurando programas. Para ello, deberás escribir programas CLIPS que resuelvan los problemas planteados con los algoritmos que se piden. Utilizaremos el modulo de control para implementar los distintos algoritmos (A*, CU). Para una adecuada separación del módulo de control (MAIN) del resto de módulos puedes consultar el ejemplo del 8-puzzle de la lección de control en sistemas de producción.

La ***segunda tarea*** consistirá en la resolución de **sudokus** mediante la **propagación de restricciones**. No se ha explicado en teoría este tema, por lo que este trabajo servirá para familiarizarte con los problemas de satisfacción de restricciones. En los **problemas de satisfacción de restricciones** se representa el estado mediante un conjunto de variables que pueden tener valores en dominios definidos, y se definen un conjunto de restricciones especificando combinaciones de valores para subconjuntos de variables. El objetivo es entender de forma intuitiva como la propagación de restricciones puede reducir el espacio de estados en los que realizar las búsquedas. Los algoritmos CSP (Constraint Satisfaction Problem, CSP) entrelazan búsqueda (***backtracking***), algoritmos de propagación de restricciones considerando diferente número de variables, como las restricciones sobre los valores de dos variables (***AC-3***, Arc *consistency version 3*), y **heurísticas de propósito general** como elegir primero las variables con menor número de valores posibles. Para un conocimiento detallado de cómo resolver problemas mediante la satisfacción de restricciones debes leer el capítulo 5 del libro “Artificial Intelligence” de Stuart Russell y Peter Norvig (http://aima.cs.berkeley.edu/newchap05.pdf) para tener una idea general. Para implementar el problema de resolución de sudokus utilizaras el código en el paquete `aima.core.search.cps`, y estudiarás el ejemplo de coloreado del mapa de Australia que puedes encontrar en el paquete `aima.gui.applications.search.csp`.

Finalmente, la ***tercera tarea*** se familiarizarás con los algoritmos de búsqueda con adversario utilizados en **juegos** realizando búsquedas en un árbol de juego, y a continuación aprenderás a modelar juegos y a utilizar los algoritmos MINIMAX y poda alfa-beta con el juego modelado.

2. Primera Tarea (3/10)

2.1 Problema 1 CLIPS (1 puntos)

La situación inicial es

```
+---+---+---+---+---+---+---+
| B | B | B |   | V | V | V |
+---+---+---+---+---+---+---+
```

La situación final es

```
+---+---+---+---+---+---+---+
| V | V | V |   | B | B | B |
+---+---+---+---+---+---+---+
```

Los movimientos permitidos consisten en desplazar una ficha al hueco, saltando como máximo, sobre otras dos. Puedes partir del programa Puzzle.clp presentado en las transparencias y modificar la representación del estado y los operadores. Puedes utilizar la siguiente representación:

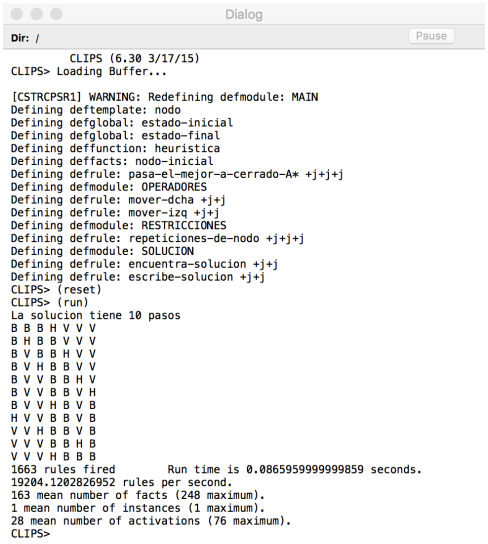
```
(deftemplate nodo
    (multislot estado)
    (multislot camino)
    (slot heuristica)
    (slot coste)
    (slot clase (default abierto)))

(defglobal MAIN
    ?*estado-inicial* = (create$ B B B H V V V))
```

Funciones que pueden ser útiles: implode$, explode$, create$, duplicate, loop-for-count ... En http://clipsrules.sourceforge.net/OnlineDocs.html encontrarás la documentación de CLIPS, aunque es suficiente con los ejemplos que encontrarás en las trasparencias de clase.

En esta versión utilizaremos módulos: MAIN, OPERACIONES, RESTRICCIONES que detecta nodos repetidos y SOLUCION que reconoce la solución y escribe los pasos. El módulo MAIN implementa la búsqueda con heurística. Debes implementar un A*. Puedes utilizar la **heurística** que cuenta el número de fichas descolocadas. Por ejemplo: La heurística del siguiente estado para h es 4.

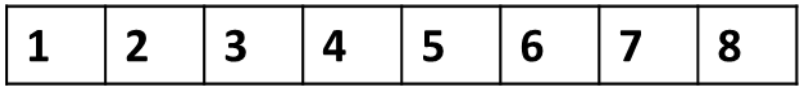
```
+---+---+---+---+---+---+---+
| B | V | B |   | V | V | B |
+---+---+---+---+---+---+---+
```

Debes entregar un único fichero `fichas.clp` con los módulos mencionados y listo para ser ejecutado como en el ejemplo que se muestra a continuación:


Dialog
Dir: /
Pause
CLIPS (6.30 3/17/15)
CLIPS> Loading Buffer...
[CSTRCPSR1] WARNING: Redefining defmodule: MAIN
Defining deftemplate: nodo
Defining defglobal: estado-inicial
Defining defglobal: estado-final
Defining deffunction: heuristica
Defining deffacts: nodo-inicial
Defining defrule: pasa-el-mejor-a-cerrado-A* +j+j+j
Defining defmodule: OPERADORES
Defining defrule: mover-dcha +j+j
Defining defrule: mover-izq +j+j
Defining defmodule: RESTRICCIONES
Defining defrule: repeticiones-de-nodo +j+j+j
Defining defmodule: SOLUCION
Defining defrule: encuentra-solucion +j+j
Defining defrule: escribe-solucion +j+j
CLIPS> (reset)
CLIPS> (run)
La solucion tiene 10 pasos
B B B H V V V
B H B B V V V
B V B B H V V
B V H B B V V
B V V B B H V
B V V B B V H
B V V H B V B
H V V B B V B
V V H B B V B
V V V B B H B
V V V H B B B
1663 rules fired Run time is 0.0865959999999859 seconds.
19204.1202826952 rules per second.
163 mean number of facts (248 maximum).
1 mean number of instances (1 maximum).
28 mean number of activations (76 maximum).
CLIPS>

2.2 Problema 2 CLIPS (1 puntos)

Dada la siguiente figura, encontrar un camino de desde la casilla inicial a la casilla final. Se comienza situado en el cuadrado más a la izquierda (casilla 1), y el objetivo es llegar exactamente a la última casilla (casilla 8).



En cada casilla se pueden realizar dos acciones diferentes:

- **Andar** desde la casilla S a la casilla S+1 con **coste 1**.
- **Saltar** de la casilla S a las 2*S con **coste 2**.

con la **restricción** de que no puede haber más acciones de saltar que de andar para llegar a un estado, y que no se permiten las acciones que mueven más allá de la casilla objetivo.

Se pide completar el siguiente programa CLIPS. Completa el código necesario deftemplate, reglas, etc. para implementar una **búsqueda de CU**. **Funciones útiles:** explode$, create$, para guardar el camino. create$ crea una lista de elementos, y explode$ genera una cadena con los elementos de una lista. Ejemplo de uso:

```
CLIPS> (implode$ (create$ 1 0 0))
"1 0 0"
```

Programa CLIPS a completar los recuadros:

```
; ------------------------------------------------------------
; MODULO MAIN (COMPLETAR)
;-------------------------------------------------------------
(defmodule MAIN
      (export deftemplate nodo))
```

```
(deftemplate MAIN::nodo
      ;;; Definición del estado.
      (multislot camino)
      (slot coste (default 0))
     (slot clase (default abierto)))
```

```
 (deffacts MAIN::estado-inicial
    ;;; DEFINE Nodo inicial
 )
```

```
 (defrule MAIN::pasa-el-mejor-a-cerrado-CU
  ;;; IMPLEMENTA CU
 )
```

```
;---------------------------------------------------------
; MODULO OPERADORES (COMPLETAR)
;---------------------------------------------------------
; Acciones andar y saltar con sus restricciones
(defmodule OPERADORES
   (import MAIN deftemplate nodo))
```

```
(defrule OPERADORES::Andar
;;; IMPLEMENTA
)
(defrule OPERADORES::Saltar
;;; IMPLEMENTA
)
```

```
;---------------------------------------------------------
; MODULO RESTRICCIONES (COMPLETAR)
;---------------------------------------------------------
; Nos quedamos con el nodo de menor coste
; La longitud del camino no es el coste

(defmodule RESTRICCIONES
   (import MAIN deftemplate nodo))
```

```
; eliminamos nodos repetidos
(defrule RESTRICCIONES::repeticiones-de-nodo
 ;;; IMPLEMENTA
)
```

```
 ;---------------------------------------------------------
; MODULO SOLUCION
;---------------------------------------------------------
;Definimos el modulo solución
(defmodule SOLUCION
   (import MAIN deftemplate nodo))

;Miramos si hemos encontrado la solucion
(defrule SOLUCION::encuentra-solucion
      (declare (auto-focus TRUE))
      ?nodo1 <- (nodo (casilla 8) (camino $?camino)
                      (clase cerrado))
   =>
      (retract ?nodo1)
      (assert (solucion ?camino)))

;Escribimos la solución por pantalla
(defrule SOLUCION::escribe-solucion
    (solucion $?movimientos)
   =>
      (printout t "La solucion tiene " (- (length ?movimientos) 1)
                  " pasos" crlf)
      (loop-for-count (?i 1 (length ?movimientos))
           (printout t "(" (nth ?i $?movimientos) ")" " "))
       (printout t crlf)
      (halt))
```

3. Segunda Tarea - resolución de sudokus mediante propagación de restricciones y búsqueda. (4/10)

Las reglas del Sudoku

Si no estás familiarizado con la resolución de Sudokus, en las siguientes páginas puedes encontrar estrategias en las que describen como resolverlos:

http://norvig.com/sudoku.html, https://sudokudragon.com,
http://www.sudokumania.com.ar/metodos

Un sudoku está resuelto si los cuadrados de cada unidad del sudoku se completan con una permutación de los dígitos 1 a 9. Esto queda más claro si entendemos la definición de cuadrado y unidad de un Sudoku (https://norvig.com/sudoku.html):

```
 A1 A2 A3| A4 A5 A6| A7 A8 A9
 B1 B2 B3| B4 B5 B6| B7 B8 B9
 C1 C2 C3| C4 C5 C6| C7 C8 C9
---------+---------+---------
 D1 D2 D3| D4 D5 D6| D7 D8 D9
 E1 E2 E3| E4 E5 E6| E7 E8 E9
 F1 F2 F3| F4 F5 F6| F7 F8 F9
---------+---------+---------
 G1 G2 G3| G4 G5 G6| G7 G8 G9
 H1 H2 H3| H4 H5 H6| H7 H8 H9
 I1 I2 I3| I4 I5 I6| I7 I8 I9
```

Dónde C2 es un cuadrado en la intersección de la tercera fila (denominada C) con la segunda fila (denominada 2). *Cada cuadrado tiene exactamente 3* ***unidades***. Por ejemplo, las unidades del cuadrado C2 son:

La unidad columna de C2

```
   A2   |         |
   B2   |         |
   C2   |         |
--------+---------+---------
   D2   |         |
   E2   |         |
   F2   |         |
--------+---------+---------
   G2   |         |
   H2   |         |
   I2   |         |
```

La unidad fila de C2

```
        |         |
        |         |
 C1 C2 C3| C4 C5 C6| C7 C8 C9
--------+---------+---------
        |         |
        |         |
        |         |
--------+---------+---------
        |         |
        |         |
        |         |
```

La unidad caja de C2

```
 A1 A2 A3|         |
 B1 B2 B3|         |
 C1 C2 C3|         |
---------+---------+---------
         |         |
         |         |
         |         |
---------+---------+---------
         |         |
         |         |
         |         |
```

Propagación de restricciones

En la resolución de sudokus hay dos estrategias importantes que pueden seguirse para ir rellenando los cuadrados:

1. Si un cuadrado tiene un único valor posible, eliminar el valor de todos los cuadrados que comparten la unidad.
2. Si una unidad tiene un único lugar posible para un valor, entonces se puede colocar en ese cuadrado.

Como ejemplo de la primera estrategia, si asignamos 7 a la casilla `A1` `{'A1': '7', 'A2':'123456789', ...}`, y si vemos que `A1` tiene un único valor, podemos sacar 7 de la celda `A2` (cualquier otra celda de la misma fila, columna, o caja), quedando `{'A1': '7', 'A2': '12345689', ...}`. Como ejemplo de la segunda estrategia, si ninguna casilla entre `A3` y `A9` tiene un 3 como valor posible, entonces el 3 debe estar en `A2`, y podemos actualizar a `{'A1': '7', 'A2':'3', ...}`. Estas actualizaciones de `A2` pueden causar otras actualizaciones de las casillas que comparten unidad, y estas actualizaciones a su vez pueden causar otras. Este proceso se denomina propagación de restricciones.

Una aproximación para resolver el problema sería definir las estrategias más habituales utilizadas para la resolución de sudokus, es decir representaremos el ***conocimiento de "expertos"*** en la resolución de sudokus para la propagación de restricciones. Además de las dos estrategias básicas presentadas, podemos implementar estrategias más sofisticadas. Por ejemplo, la ***estrategia de parejas/tríos desnudos*** (http://www.playsudoku.biz/parejas-trios-desnudos.aspx) busca dos celdas en la misma unidad que tengan los mismos valores posibles. Dado `{'A5': '26', 'A6':'26', ...}`, podemos determinar que el 2 y el 6 deben estar en las celdas `A5` y `A6` (aunque no sabemos cual va en que celda), y, por lo tanto, podemos eliminar el 2 y el 6 de los otros cuadrados de la fila `A` en que aparezcan.

Podríamos implementar estas estrategias específicas fácilmente con un lenguaje de reglas como **CLIPS**. Codificar estrategias para propagar restricciones de acuerdo a los expertos es un camino, pero podría requerir implementar muchas, y algunas son difíciles de implementar de forma que resulten patrones sencillos y eficientes. Además, nunca estaríamos seguros que serán suficientes para completar el sudoku. En lugar de eso, **vamos a comprobar como los algoritmos genéricos CPS, aplicando heurísticas genéricas pueden resolver de forma eficiente Sudokus** que se pueda plantear como un problema de propagación de restricciones.

Búsqueda

La propagación de restricciones se entrelaza con la búsqueda (backtracking). El riesgo que se corre en este caso es el número de posibilidades. Tomando como ejemplo el siguiente sudoku, en el que se muestran los valores asignado y los posible en las celdas:

```
  4      1679   12679  |  139     2369    269    |   8      1239     5
26789     3    1256789 | 14589   24569   245689  | 12679    1249   124679
 2689   15689   125689 |   7     234569  245689  | 12369   12349   123469
-----------------------+------------------------+-----------------------
 3789     2     15789  |  3459   34579    4579   | 13579     6     13789
 3679   15679   15679  |  359      8     25679   |   4     12359   12379
36789     4     56789  |  359      1     25679   | 23579   23589   23789
-----------------------+------------------------+-----------------------
 289      89     289   |   6      459      3     |  1259     7     12489
  5      6789     3    |   2      479      1     |   69     489     4689
  1      6789     4    |  589     579     5789   | 23569   23589   23689
```

Podemos ver que `A2` tiene 4 posibilidades (1679) y `A3` tiene 5 posibilidades (12679); juntas dan un total de 20, y si continuamos multiplicando obtendremos $4.62838344192 \times 10^{38}$ posibilidades. Para abordar la búsqueda en este espacio de estados utilizaremos una búsqueda en profundidad en la que cuando generemos un nodo, aplicaremos sobre este estado la propagación de restricciones para eliminar valores de los rangos en la misma fila/columna/caja que el asignado. CPS utiliza heurísticas que se pueden aplicar a cualquier problema como expandir el nodo que tenga menos posibilidades pendientes.

Modelado de un problema CSP con `aima.core.search.csp`

Para definir un problema CSP debemos heredar de la clase `CSP`, en la que se definen las variables, sus dominios y las restricciones.

```java
package aima.core.search.csp;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Hashtable;
import java.util.List;

public class CSP {
    private List<Variable> variables;
    private List<Domain> domains;
    private List<Constraint> constraints;

    private Hashtable<Variable, Integer> varIndexHash;
    private Hashtable<Variable, List<Constraint>> cnet;

    public CSP() {
        variables = new ArrayList<Variable>();
        domains = new ArrayList<Domain>();
        constraints = new ArrayList<Constraint>();
        varIndexHash = new Hashtable<Variable, Integer>();
        cnet = new Hashtable<Variable, List<Constraint>>();
    }

    public CSP(List<Variable> vars) {
        this();
        for (Variable v : vars)
            addVariable(v);
    }
```

```
protected void addVariable(Variable var) {
        if (!varIndexHash.containsKey(var)) {
           Domain emptyDomain = new Domain(Collections.emptyList());
            variables.add(var);
            domains.add(emptyDomain);
            varIndexHash.put(var, variables.size() - 1);
            cnet.put(var, new ArrayList<Constraint>());
        } else { throw new IllegalArgumentException(
                        "Variable with same name already exists."); }
    }
    public List<Variable> getVariables() {
        return Collections.unmodifiableList(variables);
    }
    public int indexOf(Variable var) {
        return varIndexHash.get(var);
    }
    public Domain getDomain(Variable var) {
        return domains.get(varIndexHash.get(var));
    }
    public void setDomain(Variable var, Domain domain) {
        domains.set(indexOf(var), domain);
    }
    public void removeValueFromDomain(Variable var, Object value) {
        Domain currDomain = getDomain(var);
        List<Object> values = new ArrayList<Object>(currDomain.size());
        for (Object v : currDomain)
            if (!v.equals(value))
                values.add(v);
        setDomain(var, new Domain(values));
    }
    public void addConstraint(Constraint constraint) {
        constraints.add(constraint);
        for (Variable var : constraint.getScope())
            cnet.get(var).add(constraint);
    }
    public List<Constraint> getConstraints() {
        return constraints;
    }
    public List<Constraint> getConstraints(Variable var) {
        return cnet.get(var);
    }
    public Variable getNeighbor(Variable var, Constraint constraint) {
        List<Variable> scope = constraint.getScope();
        if (scope.size() == 2) {
            if (var.equals(scope.get(0)))
                return scope.get(1);
            else if (var.equals(scope.get(1)))
                return scope.get(0);
        }
        return null;
    }
    public CSP copyDomains() {
        CSP result = new CSP();
        result.variables = variables;
        result.domains = new ArrayList<Domain>(domains.size());
        result.domains.addAll(domains);
        result.constraints = constraints;
        result.varIndexHash = varIndexHash;
        result.cnet = cnet;
       return result}
}
```

Las variables se definen en la clase `Variable`. Una variable tiene un nombre de tipo String. Si mis variables tienen más atributos, tendré que heredar de esta clase y añadir los campos necesarios en la nueva clase.

```
package aima.core.search.csp;

public class Variable {
    private String name;

    public Variable(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public String toString() {
        return name;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj == null)
            return false;
        if (obj.getClass() == getClass())
            return this.name.equals(((Variable) obj).name);
        return false;
    }
    @Override
    public int hashCode() {
        return name.hashCode();
    }
}
```

Para definir las restricciones debemos implementar el interface `Constraint`:

```
package aima.core.search.csp;

import java.util.List;
public interface Constraint {
    List<Variable> getScope();
    boolean isSatisfiedWith(Assignment assignment);
}
```

Una `Constraint` tiene dos métodos:

- `getScope()` devuelve el conjunto de variables el ámbito/scope de la restricción,
- `isSatisfiedWith()` devuelve cierto si la restricción está satisfecha por la asignación de valor dada.

Una asignación se representa mediante la clase `Assignment` que asigna valores a algunas o todas las variables del problema `CSP`. La clase permite comprobar si los valores asignados a las variables son consistentes con algunas restricciones, si es completa (todas las variables asignadas), o si es solución del problema.

```java
package aima.core.search.csp;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Hashtable;
import java.util.List;

public class Assignment {
    List<Variable> variables;
    Hashtable<Variable, Object> variableToValue;

    public Assignment() {
        variables = new ArrayList<Variable>();
        variableToValue = new Hashtable<Variable, Object>();
    }
    public List<Variable> getVariables() {
        return Collections.unmodifiableList(variables);
    }
    public Object getAssignment(Variable var) {
        return variableToValue.get(var);
    }
    public void setAssignment(Variable var, Object value) {
        if (!variableToValue.containsKey(var))
            variables.add(var);
        variableToValue.put(var, value);
    }
    public void removeAssignment(Variable var) {
        if (hasAssignmentFor(var)) {
            variables.remove(var);
            variableToValue.remove(var);
        }
    }
    public boolean hasAssignmentFor(Variable var) {
        return variableToValue.get(var) != null;
    }
    public boolean isConsistent(List<Constraint> constraints) {
        for (Constraint cons : constraints)
            if (!cons.isSatisfiedWith(this))
                return false;
        return true;
    }
   public boolean isComplete(List<Variable> vars) {
        for (Variable var : vars)
            if (!hasAssignmentFor(var))
                return false;
        return true;
   }
   public boolean isComplete(Variable[] vars) {
        for (Variable var : vars)
            if (!hasAssignmentFor(var))
                return false;
        return true;
    }
    public boolean isSolution(CSP csp) {
        return isConsistent(csp.getConstraints())
                && isComplete(csp.getVariables());
    }
    public Assignment copy() {
        Assignment copy = new Assignment();
       for (Variable var : variables)
           copy.setAssignment(var, variableToValue.get(var));
        return copy;
   }
```

```
    @Override
    public String toString() {
        boolean comma = false;
        StringBuffer result = new StringBuffer("{");
        for (Variable var : variables) {
            if (comma)
                result.append(", ");
            result.append(var + "=" + variableToValue.get(var));
            comma = true;
        }
        result.append("}");
        return result.toString();
    }
}
```

Las variables tienen dominios que definen los valores que pueden tomar. La definición de dominios se realiza implementado la clase `Domain`, que es una subclase de la clase `Iterable` (véase http://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html):

```
package aima.core.search.csp;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import fr.emse.ai.util.ArrayIterator;

public class Domain implements Iterable<Object> {
    private Object[] values;

    public Domain(List<?> values) {
        this.values = new Object[values.size()];
        for (int i = 0; i < values.size(); i++)
            this.values[i] = values.get(i);
    }
    public Domain(Object[] values) {
        this.values = new Object[values.length];
        for (int i = 0; i < values.length; i++)
            this.values[i] = values[i];
    }
    public int size() {
        return values.length;
    }
    public Object get(int index) {
        return values[index];
    }
    public boolean isEmpty() {
        return values.length == 0;
    }
    public boolean contains(Object value) {
        for (Object v : values)
           if (v.equals(value))
               return true;
        return false;
    }
```

```
    @Override
    public Iterator<Object> iterator() {
       return new ArrayIterator<Object>(values);}

     public List<Object> asList() {
        List<Object> result = new ArrayList<Object>();
        for (Object value : values)
            result.add(value);
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Domain) {
            Domain d = (Domain) obj;
            if (d.size() != values.length)
                return false;
            else
                for (int i = 0; i < values.length; i++)
                    if (!values[i].equals(d.values[i]))
                        return false;
        }
        return true;
   }
    @Override
    public int hashCode() {
        int hash = 9; // arbitrary seed value
        int multiplier = 13; // arbitrary multiplier value
        for (int i = 0; i < values.length; i++)
            hash = hash * multiplier + values[i].hashCode();
        return hash;
    }
    @Override
    public String toString() {
        StringBuffer result = new StringBuffer("{");
        boolean comma = false;
        for (Object value : values) {
            if (comma)
                result.append(", ");
            result.append(value.toString());
            comma = true;
        }
        result.append("}");
        return result.toString();
    }
}
```

Ejemplo de modelado de un problema con `aima.core.search.csp`

En concreto, el problema de colorear el mapa de Australia se modela como se muestra en el siguiente código:

```
package aima.core.search.cps;

import fr.emse.ai.csp.core.CSP;
import fr.emse.ai.csp.core.Domain;
import fr.emse.ai.csp.core.NotEqualConstraint;
import fr.emse.ai.csp.core.Variable;

public class MapCSP extends CSP {
    //nombre de las variables
    public static final Variable NSW = new Variable("NSW");
    public static final Variable NT = new Variable("NT");
    public static final Variable Q = new Variable("Q");
    public static final Variable SA = new Variable("SA");
    public static final Variable T = new Variable("T");
    public static final Variable V = new Variable("V");
    public static final Variable WA = new Variable("WA");
    public static final String RED = "RED";
    public static final String GREEN = "GREEN";
    public static final String BLUE = "BLUE";
    //constructor.
    public MapCSP() {
        // recopila variables
        addVariable(NSW);
        addVariable(WA);
        addVariable(NT);
        addVariable(Q);
        addVariable(SA);
        addVariable(V);
        addVariable(T);
       // define dominios
        Domain colors =
           new Domain(new Object[]{RED, GREEN, BLUE});
       // define dominio de variables
        for (Variable var : getVariables())
            setDomain(var, colors);
       // Añade las restricciones
        addConstraint(new NotEqualConstraint(WA, NT));
        addConstraint(new NotEqualConstraint(WA, SA));
        addConstraint(new NotEqualConstraint(NT, SA));
        addConstraint(new NotEqualConstraint(NT, Q));
        addConstraint(new NotEqualConstraint(SA, Q));
        addConstraint(new NotEqualConstraint(SA, NSW));
        addConstraint(new NotEqualConstraint(SA, V));
        addConstraint(new NotEqualConstraint(Q, NSW));
        addConstraint(new NotEqualConstraint(NSW, V));
    }
}
```

El problema de coloreado del mapa/grafo de Australia tiene una variable por estado, y restricciones entre los estados vecinos que no deben ser coloreados con el mismo color. Todas las variables tienen el mismo dominio (`red, green, blue`). Las restricciones se definen utilizando la clase `NotEqualConstraint` que implementa la inferface `Constraint`. La restricción será satisfecha si los valores de las variables son distintos:

```
package aima.core.search.cps;

import java.util.ArrayList;
import java.util.List;

public class NotEqualConstraint implements Constraint {
    private Variable var1;
    private Variable var2;
    private List<Variable> scope;

    public NotEqualConstraint(Variable var1, Variable var2) {
        this.var1 = var1;
        this.var2 = var2;
        scope = new ArrayList<Variable>(2);
        scope.add(var1);
        scope.add(var2);
    }
    @Override
    public List<Variable> getScope() {
        return scope;
    }
    @Override
    public boolean isSatisfiedWith(Assignment assignment) {
        Object value1 = assignment.getAssignment(var1);
        return value1 == null
              || !value1.equals(assignment.getAssignment(var2));
    }
}
```

Para resolver un problema CSP se utilizan los algoritmos presentados en el capítulo capítulo 5 “Costraint Satisfaction Problems” del libro “Artificial Intelligence” de Stuart Russell y Peter Norvig (http://aima.cs.berkeley.edu/newchap05.pdf). Para poder hacerlo, debemos heredar de la clase abstracta `SolutionStrategy`. Esta clase define un método `solve()` que devuelve una asignación de las variables (en un `Assigment`). Nos permite también seguir la traza paso a paso del proceso de resolución utilizando el interface `CPSStateListene` tal como se muestra en el código mostrado a continuación.

```
package aima.core.search.csp;

import java.util.ArrayList;
import java.util.List;

public abstract class SolutionStrategy {
    List<CSPStateListener> listeners = new
                                    ArrayList<CSPStateListener>();
    public void addCSPStateListener(CSPStateListener listener) {
        listeners.add(listener);
    }
    public void removeCSPStateListener(CSPStateListener listener) {
        listeners.remove(listener);
    }
    protected void fireStateChanged(CSP csp) {
       for (CSPStateListener listener : listeners)
           listener.stateChanged(csp.copyDomains());
   }
```

```
  protected void fireStateChanged(Assignment assignment, CSP csp) {
        for (CSPStateListener listener : listeners)
            listener.stateChanged(assignment.copy(),
                                  csp.copyDomains());
    }
    public abstract Assignment solve(CSP csp);
}
```

Para resolver un CSP hay que invocar el método `solve()`. Por ejemplo, el problema del coloreado del mapa se puede resolver de la siguiente forma:

```
MapCSP map = new MapCSP();
BacktrackingStrategy bts = new BacktrackingStrategy();
bts.addCSPStateListener(new CSPStateListener() {
    @Override
    public void stateChanged(Assignment assignment, CSP csp) {
        System.out.println("Assignment evolved : " + assignment);
    }
    @Override
    public void stateChanged(CSP csp) {
        System.out.println("CSP evolved : " + csp);
    }
});
double start = System.currentTimeMillis();
Assignment sol = bts.solve(map);
double end = System.currentTimeMillis();
System.out.println(sol);
System.out.println("Time to solve = " + (end - start));
```

Que mostrará por pantalla:

```
Assignment evolved : {NSW=RED}
Assignment evolved : {NSW=RED, WA=RED}
Assignment evolved : {NSW=RED, WA=RED, NT=RED}
Assignment evolved : {NSW=RED, WA=RED, NT=GREEN}
Assignment evolved : {NSW=RED, WA=RED, NT=GREEN, Q=RED}
Assignment evolved : {NSW=RED, WA=RED, NT=GREEN, Q=GREEN}
Assignment evolved : {NSW=RED, WA=RED, NT=GREEN, Q=BLUE}
Assignment evolved : {NSW=RED, WA=RED, NT=GREEN, Q=BLUE, SA=RED}
Assignment evolved : {NSW=RED, WA=RED, NT=GREEN, Q=BLUE, SA=GREEN}
Assignment evolved : {NSW=RED, WA=RED, NT=GREEN, Q=BLUE, SA=BLUE}
Assignment evolved : {NSW=RED, WA=RED, NT=BLUE}
Assignment evolved : {NSW=RED, WA=RED, NT=BLUE, Q=RED}
Assignment evolved : {NSW=RED, WA=RED, NT=BLUE, Q=GREEN}
Assignment evolved : {NSW=RED, WA=RED, NT=BLUE, Q=GREEN, SA=RED}
Assignment evolved : {NSW=RED, WA=RED, NT=BLUE, Q=GREEN, SA=GREEN}
Assignment evolved : {NSW=RED, WA=RED, NT=BLUE, Q=GREEN, SA=BLUE}
Assignment evolved : {NSW=RED, WA=RED, NT=BLUE, Q=BLUE}
Assignment evolved : {NSW=RED, WA=GREEN}
```

```
Assignment evolved : {NSW=RED, WA=GREEN, NT=RED}
Assignment evolved : {NSW=RED, WA=GREEN, NT=RED, Q=RED}
Assignment evolved : {NSW=RED, WA=GREEN, NT=RED, Q=GREEN}
Assignment evolved : {NSW=RED, WA=GREEN, NT=RED, Q=GREEN, SA=RED}
Assignment evolved : {NSW=RED, WA=GREEN, NT=RED, Q=GREEN, SA=GREEN}
Assignment evolved : {NSW=RED, WA=GREEN, NT=RED, Q=GREEN, SA=BLUE}
Assignment evolved : {NSW=RED, WA=GREEN, NT=RED, Q=GREEN, SA=BLUE, V=RED}
Assignment evolved : {NSW=RED, WA=GREEN, NT=RED, Q=GREEN, SA=BLUE, V=GREEN}
Assignment evolved : {NSW=RED, WA=GREEN, NT=RED, Q=GREEN, SA=BLUE, V=GREEN, T=RED}
{NSW=RED, WA=GREEN, NT=RED, Q=GREEN, SA=BLUE, V=GREEN, T=RED}
Time to solve = 5.0
```

Si no quieres que muestre por pantalla la traza, puede comentar las líneas que imprimen mensajes en los métodos `stateChange()`.

3.1 Problema del Sudoku (4 puntos)

Para facilitarte las cosas se te suministra código que te será útil. La clase `Sudoku` es ajena a la resolución de problemas mediante CSP, pero te servirá para leer sudokus y escribirlos por pantalla, además de comprobar si un sudoku leído o resuelto está incompleto, completo, y en este caso si está correctamente resuelto. El método `leerSudoku2()` te permitirá leer sudokus de los ficheros con el formato dado.

También se te suministran la clase `SudokuProblem` que define el problema a partir de las celdas con valor suministradas en la clase `AvailableCells`. En este código ya se hace la mayor parte del trabajo consistente en definir las restricciones entre variables del Sudoku. Tendrás que definir las clases `SudokuVariable`, `SudokuConstraint` , y `SudokuApp`. `SudokuVariable` hereda de Variable y añade el valor de la celda, y las coordenadas x e y, además del nombre de la variable. `SudokuConstraint` especifica que el valor de dos celdas debe ser distinto, y finalmente `SudokuApp`, lee y resuelve los sudokus especificados en los ficheros: `easy50.txt`, `top95.txt` y `hardest.txt`. Al final de la ejecución debes mostar el número total de sudokus tratados y cuantos han sido resueltos con éxito (Deberían ser todos resueltos).

Puedes crear una lista con todos los sudokus en los ficheros de la siguiente forma:

```
Sudoku [] lista = union (union(Sudoku.listaSudokus2("easy50.txt"),
                                Sudoku.listaSudokus2("top95.txt")),
                          Sudoku.listaSudokus2("hardest.txt"));
```

Puedes estudiar la clase `MapColoringApp` en `aima.gui.applications.search.csp` para decidir la mejor estrategia CSP a usar. Debes definir todas las clases del problema del sudoku en el paquete `aima.gui.sudoku.csp`.

Ejemplo de ejecución. Por brevedad sólo se muestra el último sudoku resuelto de los 156 y el resultado de sudokus resueltos correctamente.

…

```
---------
....7..2.
8.......6
.1.2.5...
9.54....8
.........
3....85.1
...3.2.8.
4.......9
.7..6....
SUDOKU INCOMPLETO - Resolviendo
{Cell at [0][7]=2, Cell at [8][1]=7, … Cell at [8][7]=3}
Time to solve = 0.07segundos
SOLUCION:
594876123
823914756
617235894
965421378
781653942
342798561
159342687
436587219
278169435
Sudoku solucionado correctamente
+++++++++
Numero sudokus solucionados:156
```

4. Tercera Tarea – Juegos con Adversario- Minimax y poda alfa-beta. (4/10)

El objetivo del trabajo es comprender como modelar juegos con adversario tal como se describe en el AIMA. Para familiarizarnos con el algoritmo MINIMAX, implementaremos una función que devuelva un valor de un árbol de juego.

Implementación de MINIMAX y poda Alfa-beta en un árbol de juego

Para reforzar la comprensión del algoritmo MINIMAX, implementarás una función que devolverá el valor minimax de un árbol. Utilizaremos una estructura sencilla `ArboldeJuego` para representar un árbol de juego:

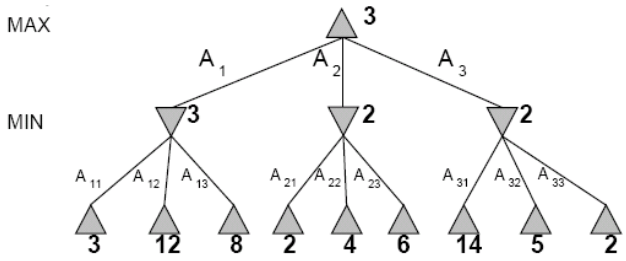
```
package aima.core.search.adversarial;

import java.util.ArrayList;
import java.util.List;

public class ArboldeJuego<V> {

    private V valor;                          //Tipo integer, Double, Float
    private boolean max;                      //nodo MAX o MIN
    private ArrayList<ArboldeJuego<V>> hijos;
    private boolean visitado=false;            // Marca de nodo visitado

    public ArboldeJuego(V valor, boolean max) {
     this.visitado=false;
        this.valor = valor;
        this.max = max;
        hijos = new ArrayList<ArboldeJuego<V>>();
    }

    public ArboldeJuego(V valor,
                        boolean max,
                        List<ArboldeJuego<V>> hijos) {
        this(valor, max);
        for (ArboldeJuego<V> hijo : hijos)
            this.hijos.add(hijo);
    }

    public boolean esTerminal() {
        return hijos.isEmpty();
    }

    public boolean esMax() {
        return max;
    }

    public void agnadeHijo(ArboldeJuego<V> hijo) {
        this.hijos.add(hijo);
    }

    public V getValor() {
        return this.valor;
    }

    public void setValor(V valor) {
```

```
        this.valor = valor;
    }

    public ArrayList<ArboldeJuego<V>> getHijos() {
        return hijos;
    }

    public void setHijos(ArrayList<ArboldeJuego<V>> hijos) {
        this.hijos = hijos;
    }

    public void setVisitado() {
        this.visitado = true;
    }

    public boolean getVisitado() {
        return this.visitado;
    }

    public String toString() {
        return printArbol(0);
      }


    public void printArbolExplorado() {
        System.out.println(printArbolExplorado(0));
        System.out.println();
      }

      private static final int indentacion = 4;

      private String printArbol(int increment) {
        String s = "";
        String inc = "";
        for (int i = 0; i < increment; ++i) {
          inc = inc + " ";
        }
        if (this.esTerminal()){
                String formato = "%"+increment+"s";
                s = inc+ String.format(formato,"["+ valor +"]");
                //s = inc +"["+ valor +"]";
        }
        else {
                String type;
                if (this.max) type ="-MAX";
                else type="-MIN";
                s = inc + valor+type;

        }
        for (ArboldeJuego<V> hijo : hijos) {
             s += "\n" + hijo.printArbol(increment + indentacion);
        }
        return s;
      }

      private String printArbolExplorado(int incremento) {
       String s = "";
          String inc = "";
          for (int i = 0; i < incremento; ++i) {
            inc = inc + " ";
          }
          if (this.esTerminal()){
                String formato = "%"+incremento+"s";
                String marca="";
                if (!this.visitado) marca="X";
```

```
                else marca = valor.toString();
                s = inc+ String.format(formato,"["+ marca +"]");
                      //s = inc +"["+ valor +"]";
          }
          else {
                String tipo;
                if (this.max) tipo ="-MAX";
                else tipo="-MIN";
                String marca="";
                if (!this.visitado) marca="X";
                else marca = valor.toString();
                s = inc + valor+tipo;

          }
          for (ArboldeJuego<V> hijo : hijos) {
             s += "\n" + hijo.printArbolExplorado(incremento + indentacion);
          }
          return s;
        }
      }
```

Se instanciará el árbol de juego presentado en las transparencias para ilustrar el algoritmo MINIMAX:

```
ArrayList<ArboldeJuego<Integer>> sublist1 =
                new ArrayList<ArboldeJuego<Integer>>();
                sublist1.add(new ArboldeJuego<Integer>(3,true));  //MAX, terminal 3
                sublist1.add(new ArboldeJuego<Integer>(12,true)); //MAX, terminal 12
                sublist1.add(new ArboldeJuego<Integer>(8,true));  //MAX, terminal 8
          ArboldeJuego<Integer> subtree1 =    // árbol MIN del que cuelga 3,12,8
                new ArboldeJuego<Integer>(Integer.MIN_VALUE,false,sublist1);

          ArrayList<ArboldeJuego<Integer>> sublist2 =
                new ArrayList<ArboldeJuego<Integer>>();
                sublist2.add(new ArboldeJuego<Integer>(2,true));
                sublist2.add(new ArboldeJuego<Integer>(4,true));
                sublist2.add(new ArboldeJuego<Integer>(6,true));
          ArboldeJuego<Integer> subtree2 =  // árbol MIN del que cuelga 2,4,6
                new ArboldeJuego<Integer>(Integer.MIN_VALUE,false,sublist2);

          ArrayList<ArboldeJuego<Integer>> sublist3 =
                new ArrayList<ArboldeJuego<Integer>>();
                sublist3.add(new ArboldeJuego<Integer>(14,true));
                sublist3.add(new ArboldeJuego<Integer>(5,true));
                sublist3.add(new ArboldeJuego<Integer>(2,true));
          ArboldeJuego<Integer> subtree3 =  // árbol MIN del que cuelga 14,5,2
                        new
ArboldeJuego<Integer>(Integer.MIN_VALUE,false,sublist3);

          ArrayList<ArboldeJuego<Integer>> topTree = //Top arrayList con subárboles
                      new ArrayList<ArboldeJuego<Integer>>();
                topTree.add(subtree1);
                topTree.add(subtree2);
                topTree.add(subtree3);
          ArboldeJuego<Integer> tree=
                        new ArboldeJuego<Integer>(Integer.MIN_VALUE,true,topTree);
```



4.1 Problema Búsquedas en árboles de juego (2 puntos)

1. Crea la clase `MinimaxArbolJuego` en el paquete `aima.core.search.adversarial` que implementa la función que devuelva el valor minimax de una instancia de `ArboldeJuego<Double>`. Puedes adaptar el algoritmo de la clase `MinimaxSearch`.
2. Crea la clase `AlfaBetaArbolJuego` que implementa el algoritmo alfa-beta que devuelve el valor minimax de una instancia de `ArboldeJuego<Double>` usando la poda alfa-beta. Puedes adaptar el algoritmo de la de la clase `AlphaBetaSearch` .
3. Actualiza las clases `MinimaxArbolJuego` y `AlfaBetaArbolJuego` para que muestren por pantalla los árboles, los nodos visitados y las podas realizadas como se muestra en los ejemplos (La clase Arbol de Juego ya está preparada). A continuación crea la clase `EjemplosArbolJuego`, y compara los resultados del algoritmo MINIMAX y la poda ALFA-BETA con el árbol propuesto y ejemplos que consideres oportuno. Ejemplos de ejecución en el principal `EjemplosArbolJuego`:

```
Valor con MINIMAX:3.0
3.0-MAX
    3.0-MIN
            [3.0]
          [12.0]
            [8.0]
    2.0-MIN
            [2.0]
            [4.0]
            [6.0]
    2.0-MIN
          [14.0]
            [5.0]
            [2.0]
Nodos visitados:12
------------------------------
Valor con poda Alfa Beta:3.0
3.0-MAX
    3.0-MIN
            [3.0]
          [12.0]
            [8.0]
    2.0-MIN
            [2.0]
              [X]
              [X]
    2.0-MIN
          [14.0]
            [5.0]
            [2.0]
Nodos visitados:10
------------------------------
Valor con poda Alfa Beta:4.0
4.0-MAX
    4.0-MIN
          [10.0]
            [4.0]
            [8.0]
    3.0-MIN
            [3.0]
              [X]
              [X]
    2.0-MIN
            [2.0]
              [X]
              [X]

Nodos visitados:8
```

Modelado del Juego

Para representar un juego partimos de la interface parametrizado GAME que recoge la estructura de un juego:

```
package aima.core.search.adversarial;

import java.util.List;
public interface Game<STATE, ACTION, PLAYER> {
        STATE getInitialState();
        PLAYER[] getPlayers();
        PLAYER getPlayer(STATE state);
        List<ACTION> getActions(STATE state);
        STATE getResult(STATE state, ACTION action);
        boolean isTerminal(STATE state);
        double getUtility(STATE state, PLAYER player);
}
```

La definición depende del tipo de los estados, las acciones y los jugadores:

- `getInitialState` devuelve el estado inicial del juego,
- `getPlayers` devuelve los jugadores(normalmente 2, como `0` y `1`, o `max` y `min`),
- `getPlayer` devuelve el jugador active de un estado,
- `getActions` devuelve la lista de posibles acciones para un estado,
- `getResult` devuelve el estado resultante de realizar una acción en un estado dado,
- `isTerminal` comprueba si un estado es terminal,
- `getUtility` devuelve la función de utilidad de un estado.

Nim: Ejemplo de modelado de juego

En lugar de utilizar los algoritmos sobre un árbol de juego, vamos a definir un juego sencillo, el juego de Nim del que hay muchas variantes. En concreto vamos a utilizar la variante subtration game.

```
package aima.gui.demo.juegos;
import aima.core.search.adversarial.Game;
import java.util.ArrayList;
import java.util.List;

public class NIMJuego implements Game<List<Integer>, Integer, Integer> {

    public final static Integer[] players = {0, 1}; // 0 Max, 1 MIN
    public final static List<Integer> initialState = new ArrayList<Integer>();

    public NIMJuego(int size) {
        initialState.add(0);    // Empieza MAX
        initialState.add(size);
    }
    @Override
    public List<Integer> getInitialState() {
```

```
            return initialState;
        }

        @Override
        public Integer[] getPlayers() {
            return players;
        }

        @Override
        public Integer getPlayer(List<Integer> state) {
            return state.get(0);
        }

        @Override
        public List<Integer> getActions(List<Integer> state) {
            ArrayList<Integer> actions = new ArrayList<Integer>();
            for (int i = 1; i <= Math.min(3, state.get(1)); i++)
                actions.add(i);
            return actions;
        }

        @Override
        public List<Integer> getResult(List<Integer> state, Integer action) {
            ArrayList<Integer> newState = new ArrayList<Integer>();
            newState.add(1 - state.get(0)); // cambia jugador
            newState.add(state.get(1) - action);
            return newState;
        }

        @Override
        public boolean isTerminal(List<Integer> state) {
            return state.get(1) == 0;
        }

        @Override
        public double getUtility(List<Integer> state, Integer player) {// max 0, min 1
            if (state.get(0) == 1 - player){
                if (state.get(1) == 1)
                    return Double.POSITIVE_INFINITY;
                else if (((state.get(1) - 1) % 4) == 0)
                    return 1;
                else
                    return -1;
            } else {
                if (state.get(1) == 1)
                    return Double.NEGATIVE_INFINITY;
                else if (((state.get(1) - 1) % 4) == 0)
                    return -1;
                else
                    return 1;
            }
        }
    }
```

En este juego, los dos jugadores (representado por valores `Integer` 0 y 1) cogen alternándose 1, 2 o 3 cerillas de una caja de cerillas. El jugador que saca la última cerilla de la caja pierde. El estado del juego se presenta mediante `List<Integer>` , que contiene en el índice cero el jugador y en el índice las cerillas que quedan en la caja. Las acciones se representan mediante valores `Integer` que indican el número de cerillas que coge el jugador.

Estudia el código de ejemplo. La lógica detrás de la función de utilidad. Averigua como calcula las cerillas que tiene que coger el computador.

Algoritmos de toma de decisiones en juegos

Una vez definido el juego tenemos que utilizar los algoritmos para la toma de decisiones. Para implementar los algoritmos de juego se utiliza el interfaz `AdversarialSearch` en el paquete `aima.core.search.adversarial`.

```
package aima.core.search.adversarial;

import aima.core.search.framework.Metrics;

public interface AdversarialSearch<STATE, ACTION> {
  ACTION makeDecision(STATE state);
  Metrics getMetrics();
}
```

Este interfaz se utiliza para implementar el juego para un tipo de estado y acción. Precisa de un método para tomar la decisión, que devuelve la acción dado un estado del juego. El método `getmetrics` se utilizará para comparar las métricas de los algoritmos. Encontrarás las implementaciones de las versiones de la búsqueda Minimax (`MinimaxSearch)` y alfa-beta `AlphaBetaSearch` que habrás utilizado para inspirarte en la realización de la tarea 4.1.

Jugando contra el computador

El agente ya puede tomar decisiones y jugar. A continuación, se muestra un programa que implementa el juego de Nim:

4.2 Tarea Juego Nim (1 punto)

```
package aima.gui.demo.juegos;

import aima.core.search.adversarial.AlphaBetaSearch;
import aima.core.search.adversarial.IterativeDeepeningAlphaBetaSearch;
import aima.core.search.adversarial.MinimaxSearch;

import java.util.List;
import java.util.Scanner;

public class NimJuegoApp {

    public static void main(String[] args) {
        NIMJuego juego = new NIMJuego(20);
        //MINIMAX
        MinimaxSearch<List<Integer>, Integer, Integer> minimaxSearch =
            MinimaxSearch.createFor(juego);
        //Alfa-beta
        AlphaBetaSearch<List<Integer>, Integer, Integer> alphabetaSearch =
```

```
            AlphaBetaSearch.createFor(juego);

        List<Integer> estado = juego.getInitialState();
        while (!juego.isTerminal(estado)) {
            System.out.println("======================");
            System.out.println(estado);
            int accion = -1;
            if (estado.get(0) == 0) {//humano
                List<Integer> acciones = juego.getActions(estado);
                Scanner teclado = new Scanner(System.in);
                while (!acciones.contains(accion)) {
                    System.out.println("Jugador Humano: "+
                                       "¿Cuál es tu acción?");
                    accion = teclado.nextInt();
                }
            } else {                    //computador
                System.out.println("Computador elige acción:");
                accion = minimaxSearch.makeDecision(estado);
                System.out.println("Metricas para Minimax : " +
                   minimaxSearch.getMetrics());
                alphabetaSearch.makeDecision(estado);
                System.out.println("Metricas para Alfa-Beta : " +
                   alphabetaSearch.getMetrics());
            }
            System.out.println("La acción elegida es " + accion);
            estado = juego.getResult(estado, accion);
        }
        System.out.print("GAME OVER: ");
        if (estado.get(0) == 0)
            System.out.println("¡Gana el humano!");
        else
            System.out.println("¡Gana el computador!");

    }
}
```

Modifica `NimJuegoApp` y `NimJuego` para que se pueda elegir el número de cerillas inicial en la caja, cuantas cerillas se pueden coger en cada turno y quien empieza. Ejemplo de traza de ejecución:

```
¿Total de cerillas en la caja?
20
¿Cuantas cerillas se pueden quitar en cada turno?
3
¿Empieza computador o humano? (O:Humano, 1:Computador)
1
======================
[1, 20]
Jugador Computador, eligo  acción.
Metricas para Minimax : {expandedNodes=266078}
Metricas para AlphaBeta : {expandedNodes=42330}
```

```
La acción elegida es 3
======================
[0, 17]
Jugador Humano: ¿Cuál es tu acción?
3
La acción elegida es 3
======================
[1, 14]
Jugador Computador, eligo  acción.
Metricas para Minimax : {expandedNodes=6871}
Metricas para AlphaBeta : {expandedNodes=2823}
La acción elegida es 1
======================
[0, 13]
Jugador Humano: ¿Cuál es tu acción?
3
La acción elegida es 3
======================
[1, 10]
Jugador Computador, eligo  acción.
Metricas para Minimax : {expandedNodes=599}
Metricas para AlphaBeta : {expandedNodes=397}
La acción elegida es 1
======================
[0, 9]
Jugador Humano: ¿Cuál es tu acción?
3
La acción elegida es 3
======================
[1, 6]
Jugador Computador, eligo  acción.
Metricas para Minimax : {expandedNodes=51}
Metricas para AlphaBeta : {expandedNodes=47}
La acción elegida es 1
======================
[0, 5]
Jugador Humano: ¿Cuál es tu acción?
3
La acción elegida es 3
======================
[1, 2]
Jugador Computador, eligo  acción.
Metricas para Minimax : {expandedNodes=3}
Metricas para AlphaBeta : {expandedNodes=3}
La acción elegida es 1
======================
[0, 1]
Jugador Humano: ¿Cuál es tu acción?
3
Jugador Humano: ¿Cuál es tu acción?
1
La acción elegida es 1
```

```
GAME OVER: ¡Gana el computador!
```

4.3 Tarea Juego Tic Tac Toe (1 punto)

Estudia las clases y `TitacToeState` y `TicTacToeGame` en el paquete `aima.core.environment.tictactoe`, e implementa la clase `TicTacToeApp`. Debes implementar el método `JuegaSoloMaquina` en los que sólo juega la máquina en todos los turnos. Si `metrics` es **`true`**, muestra las métricas, y el método `JuegaContraUsuario` que juega contra un humano, además del programa `main` :

```
public static void JuegaSoloMaquina(
   AdversarialSearch<TicTacToeState, XYLocation> search,
    TicTacToeGame game,
   boolean metrics,
   String Algoritmo)


public static void JuegaContraUsuario(
    AdversarialSearch<TicTacToeState, XYLocation> search,
     TicTacToeGame game)
```

```
Ejemplo de Traza:

MINI MAX con TIC TAC TOE Jugando solo maquina
Juega X
Metrics : {expandedNodes=549945}
Juega O
Metrics : {expandedNodes=59704}
Juega X
Metrics : {expandedNodes=7331}
Juega O
Metrics : {expandedNodes=934}
Juega X
Metrics : {expandedNodes=197}
Juega O
Metrics : {expandedNodes=46}
Juega X
Metrics : {expandedNodes=13}
Juega O
Metrics : {expandedNodes=4}
Juega X
Metrics : {expandedNodes=1}



ALFA-BETA con TIC TAC TOE Jugando solo maquina

Juega X
Metrics : {expandedNodes=30709}
Juega O
```

```
Metrics : {expandedNodes=4089}
Juega X
Metrics : {expandedNodes=1519}
Juega O
Metrics : {expandedNodes=220}
Juega X
Metrics : {expandedNodes=97}
Juega O
Metrics : {expandedNodes=32}
Juega X
Metrics : {expandedNodes=13}
Juega O
Metrics : {expandedNodes=4}
Juega X
Metrics : {expandedNodes=1}

MINI MAX DEMO con TIC TAC TOE Jugando contra humano

======================
- - -
- - -
- - -

Jugador Humano,¿Cual es tu acción?
Fila(0-2):0
Columna(0-2):0
======================
X - -
- - -
- - -

La máquina Juega, y elige:
Acción elegida: Coloco 0  ( 1 , 1 )
======================
X - -
- O -
- - -

Jugador Humano,¿Cual es tu acción?
Fila(0-2):
…
```

Debes generar una carpeta con nombre tu NIP, dentro habrá tres carpetas tarea21, tarea22, tarea31 y tarea41 tarea 42 tarea43 conteniendo los ficheros pedidos correspondientes y una memoria en pdf o Word. Comprime en un zip y sube a Moodle. Sólo se admiten ficheros zip (no tar o cualquier otro fichero de compresión).

Las carpetas contendrán todos los ficheros en un único paquete evaluación, de forma que solo con arrastrar los ficheros a ese paquete se puedan ejecutar los programas.

Recuerda que el trabajo es INDIVIDUAL, y que son trabajos **TUTORADOS**, por lo que es conveniente que acudas a tutorías para avanzar en el trabajo y que el esfuerzo dedicado sea el adecuado.

FECHA de ENTREGA: 28 de noviembre 2020 a las 23:59.