Trabajo TP6-1. Curso 2019-2020

Búsqueda local y Propagación de restricciones.

1. Objetivo de la práctica

El trabajo consta de tres tareas. ***La primera tarea*** tiene por objeto familiarizarse con el código en java para resolver problemas de búsqueda local y problemas de optimización. Utilizaremos el código del paquete `aima.core.search.local`. Tu trabajo consistirá en familiarizarte con los algoritmos **Hill-Climbing**, **SimulatedAnnealing** y **GeneticAlgorithm** utilizando el problema de las 8-reinas. Puedes partir del código en la clase `NQueensDemo` del paquete `aima.gui.demo.search` en el que se muestra, entre otros, la resolución del problema de las 8-reinas mediante algoritmos de Enfriamiento Simulado (**SimulatedAnnealing)**, Escalada (**Hill-Climbing**), y Algoritmos genéticos (**GeneticAlgorithm)**. La demo del algoritmo de búsqueda local **Hill-Climbing** no está correctamente implementada y deberás corregirla atendiendo a las notas de este guion.

La ***segunda tarea*** consistirá en la resolución de **sudokus** mediante la **propagación 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*** combina la idea de búsqueda local con la propagación de restricciones en el algoritmo `min-conflicts`. La idea es aplicar los algoritmos de búsqueda local, en los que se parte de una configuración completa del estado, y aplicar la propagación de restricciones utilizando una la heurística más obvia, que consiste en seleccionar el valor que tiene menor número de conflictos con otras variables. En esta tarea definiremos el

problema de las 8 –reinas como un problema CSP, y lo resolveremos con el algoritmo `min-conflicts`.

2. Primera Tarea (3/10)

`HillClimbingSearch / Escalada`

1. Implementa el método `nQueensHillClimbingSearch_Statistics(int numExperiments)` que realice `numExperimets veces la búsqueda Hill-Climbing` y muestre el porcentaje de éxitos, fallos, media de pasos al fallar y media de pasos en éxito. Realiza 10000 experimentos a partir de estados iniciales aleatorios no repetidos de las 8-reinas con la búsqueda **HillClimbingSearch.**

   Ejemplo de ejecución:

   ```
   NQueens HillClimbing con 10000 estados iniciales diferentes -->
   Fallos:**.**
   Coste medio fallos:*.**
   Exitos:**.**
   Coste medio Exitos:*.**
   ```

2. Implementa el método `nQueensRandomRestartHillClimbing()` que reinicia el estado inicial hasta que se obtiene el éxito. Muestra número de reintentos, solución y estadísticas.

   Ejemplo de ejecución:

   ```
   Search Outcome=SOLUTION_FOUND
   Final State=
   -----Q--
   -Q------
   ------Q-
   Q-------
   ---Q----
   -------Q
   ----Q---
   --Q-----

   Numero de intentos:*
   Fallos:*
   Coste medio fallos:*
   Coste éxito:*
   Coste medio éxito:*
   ```

   Nota 1: En la clase `NQueensDemo` se muestra un ejemplo de uso de la clase `HillClimbingSearch`. Pero ten cuidado porque usan las funciones que de forma incremental añaden una reina en cada columna a partir de un tablero vacío : *`NQueensFunctionFactory.getIActionsFunction()`*. Para hacer una búsqueda con `HillClimbingSearch` debemos partir de un estado completo y cambiar la posición de las reinas. Deberás utilizar las funciones en `NQueensFunctionFactory.`*`getCActionsFunction`*`()`.

Nota 2: Ten cuidado con como representa los tableros en el programa. A continuación, tienes ejemplos de matrices que representan los tableros correctamente y como los visualiza.

```
        /**
         * X--> increases left to right with zero based index
         * Y--> increases top to bottom with zero based index | |
         * V [colum,left to right x][row, top-down y]
         */
          static NQueensBoard EjemploSolucion = new NQueensBoard(
                                  // f1 f2 f3 f4 f5 f6 f7 f8
                      new int[][] {{0, 0, 0, 0, 1, 0, 0, 0},   //c1
                                   {0, 0, 1, 0, 0, 0, 0, 0},   //c2
                                   {1, 0, 0, 0, 0, 0, 0, 0},   //c3
                                   {0, 0, 0, 0, 0, 0, 1, 0},   //c4
                                   {0, 1, 0, 0, 0, 0, 0, 0},   //c5
                                   {0, 0, 0, 0, 0, 0, 0, 1},   //c6
                                   {0, 0, 0, 0, 0, 1, 0, 0},   //c7
                                   {0, 0, 0, 1, 0, 0, 0, 0}}); //c8
        /* ejemplo solución representado
         *
        --Q-----  3
        ----Q---  5
        -Q------  2
        -------Q  8
        Q-------  1
        ------Q-  7
        ---Q----  4
        -----Q--  6
        */
        static NQueensBoard EjemploEstadoCompleto = new NQueensBoard(
                    //f1 f2 f3 f4 f5 f6 f7 f8
        new int[][] {{1, 0, 0, 0, 0, 0, 0, 0},   //c1
                     {0, 0, 1, 0, 0, 0, 0, 0},   //c2
                     {1, 0, 0, 0, 0, 0, 0, 0},   //c3
                     {0, 0, 0, 0, 0, 0, 1, 0},   //c4
                     {0, 1, 0, 0, 0, 0, 0, 0},   //c5
                     {0, 0, 0, 0, 0, 0, 0, 1},   //c6
                     {0, 0, 0, 0, 0, 1, 0, 0},   //c7
                     {0, 0, 0, 1, 0, 0, 0, 0}}); //c8
/* EjemploEstadoCompleto
Q-Q-----
----Q---
-Q------
-------Q
--------
------Q-
---Q----
-----Q--
*/
```

SimulatedAnnealing / Enfriamiento Simulado

El algoritmo Simulated annealing (SA) genera un estado aleatorio sucesor: (i) si el coste del nuevo estado es mejor, el cambio se acepta; (ii) por el contrario, si se produce un incremento en la función de evaluación, el cambio será aceptado con una cierta probabilidad (Ver el pseudo-código del algoritmo). El SA acepta con mayor probabilidad estados peores al principio, pero según se va "enfriando", la probabilidad de aceptar estados peores es menor. Esta forma de aceptar con alta probabilidad estados peores al principio y con menos probabilidad al final permite evitar óptimos locales. El nombre e inspiración viene del proceso de templado ("*annealing*" en inglés) en metalurgia, una técnica que consiste en calentar y luego enfriar controladamente un metal para aumentar el tamaño de sus cristales y reducir sus defectos. El calor causa que los átomos se salgan de sus posiciones iniciales (se encuentran en un mínimo local de energía) y se muevan aleatoriamente; el enfriamiento lento les da mayores probabilidades de encontrar configuraciones con menor energía que la inicial.

Pseudo-código del Algoritmo Simulated Annealing

```
function Simulated_Annealing(T0, k, Tfinal)
T <- T0
Sactual <- Solución inicial
for T=T0 to Tfinal do
    if T = 0 then return Sactual
    else Snuevo <- Nueva solución aleatoria
    Inc(C) <- Coste(Snuevo)-Coste(Sactual)
    if (Inc(C) > 0) then Sactual <- Snuevo
    else Sactual <- Snuevo con probabilidad Exp(Inc(C) /F(T))
                        donde F(T) = k Exp(-delta*T)
    T <- Evolucion(T)
```

donde, Inc(C) = [Coste(nuevo estado) − Coste(estado anterior)]

- Cuando Inc(C) < 0:la probabilidad de cambio tiene una P [aceptación] = 1
- Cuando Inc(C) > 0: la probabilidad de cambio tiene una P[aceptación] = $\exp\left(\frac{Inc(c)}{F(T)}\right)$

  donde F (T) = k · exp(−δ T)

Los **parámetros** del algoritmo significan lo siguiente: (i) los valores que toma la **variable T** se corresponden con el paso de iteración en la ejecución, (ii) el **parámetro k** determina la velocidad que tarda en comenzar a decrecer la temperatura, y (iii) el **parámetro δ** mide lo rápido que desciende. En las Figura 1 y Figura 2 se puede ver el comportamiento del algoritmo para Inc(C) = 1, k= {10, 50, 500, 5000, 50000} manteniendo δ = 0,01, y δ= {0.0005, 0.001, 0.01, 0.1} manteniendo k = 10.

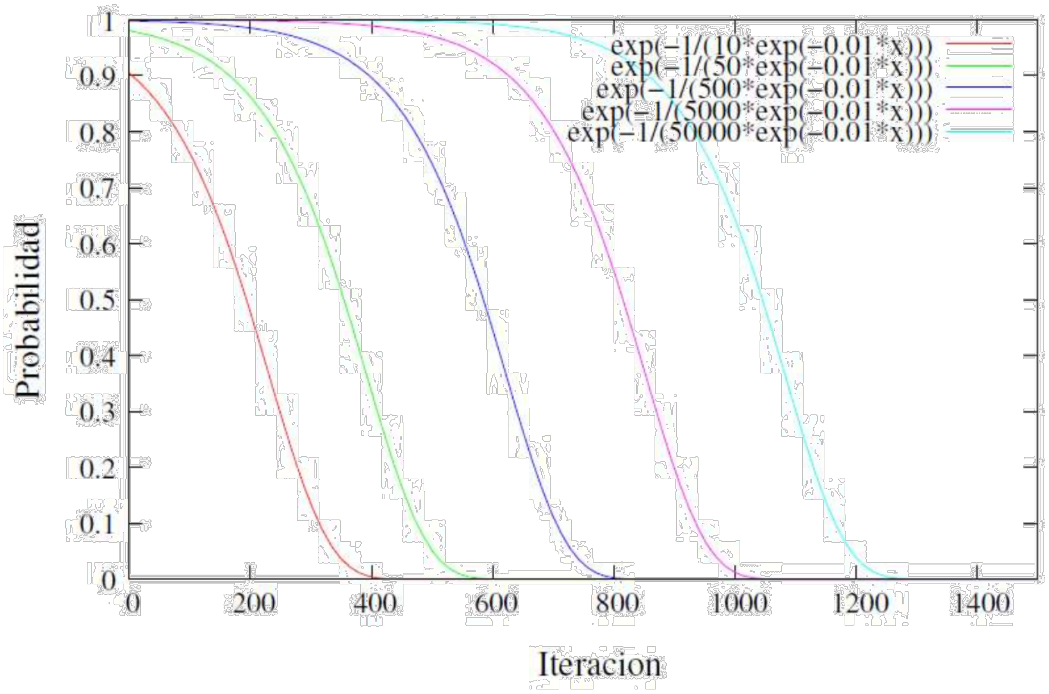

Figura 1: *Simulated annealing*: variación en función de k

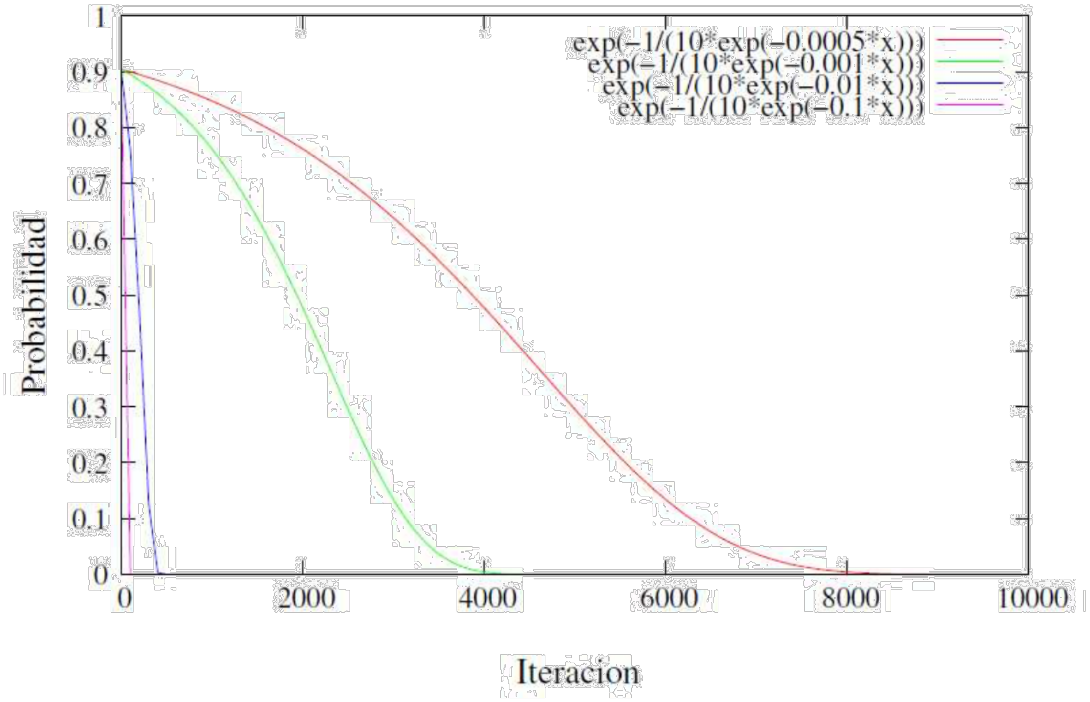

Figura 2: *Simulated annealing*: variación en función de δ

La mayor dificultad del uso del algoritmo SA es el ajuste de los parámetros ya que la única indicación que tenemos son los resultados que obtenemos experimentalmente. Esto nos obliga a hacer diversas pruebas para ver cómo se comporta el algoritmo con el problema dado. En general, es importante guiarnos por lo siguiente:

- Al aumentar el valor de k se aleja el punto en el que la P [aceptación] se anula, y al disminuir el valor de δ se alarga el tiempo necesario para converger
- Fijados unos valores de (k, δ), el número de iteraciones mínimo que necesitamos para T, será el de aquella iteración en que la P [aceptación] se anule. A partir de esta iteración, el algoritmo sólo aceptará mejores soluciones: "cómo de grande debe ser T es una cuestión a la que sólo se puede responder experimentalmente.

3. Implementa `nQueensSimulatedAnnealing_Statistics (int numExperiments)` que realice `numExperimets` y muestre el porcentaje de éxitos, fallos, media de pasos al fallar y media de pasos en éxito. Realiza 1000 experimentos a partir de estados iniciales aleatorios no repetidos de las 8-reinas con la búsqueda `SimulatedAnnealingSearch`**.** Utiliza los parámetros que creas son más adecuado y muéstralos. Ejemplo de ejecución:

```
 NQueensDemo Simulated Annealing con 1000 estados iniciales diferentes -->
Parámetros Scheduler:  Scheduler (*,*,*);

Fallos:**.**
Coste medio fallos:*.**
Exitos:**.**
Coste medio Exitos:*.**
```

4. Implementa el método `nQueensHillSimulatedAnnealingRestart()` que reinicia el estado inicial hasta que se obtiene el éxito con la búsqueda `SimulatedAnnealingSearch`. Muestra número de reintentos, solución y estadísticas. Ejemplo de ejecución:

```
Search Outcome=SOLUTION_FOUND
Final State=
--Q-----
-----Q--
-------Q
Q-------
----Q---
------Q-
-Q------
---Q----

Numero de intentos:1.0
Fallos:0.0
Coste  Éxito:36.0
```

GeneticAlgorithm / Algoritmos Genéticos

5. Realiza experimentos con la búsqueda `GeneticAlgorithm` y ajusta los parámetros de población inicial y probabilidad de mutación hasta que consideres dan los mejores resultados. Implementa el método `nQueensGeneticAlgorithmSearch()` con los parámetros elegidos . Ejemplo de ejecución:

```
GeneticAlgorithm
Parámetros iniciales:    Población: *, Probabilidad mutación: *
 Mejor individuo=
 -----Q--
 ---Q----
 ------Q-
 Q-------
 -------Q
 -Q------
 ----Q---
 --Q-----

 Tamaño tablero      = 8
 Fitness             = 28.0
 Es objetivo         = true
 Tamaño de población = *
 Iteraciones         = *
 Tiempo              = *ms.
```

Entrega una clase `NQueensLocal` **con los métodos solicitados implementados** y la ejecución de estos algoritmos en el main**.** En la **memoria** que debes entregar, muestra los resultados y comentarios de los experimentos realizados en esta primera parte del trabajo. Puedes incluir cualquier clase o método adicional que hayas implementado para la realización del trabajo. Opcionalmente puedes realizar graficas que muestren los algoritmos utilizando distintos parámetros.

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 × 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.

```
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()`.

Problema del Sudoku

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 - Propagación de restricciones y búsqueda local. (3/10)

La última tarea será resolver el problema de las 8-reinas con el algoritmo **min-conflicts**. Tendrás que definir el problema de las 8-reinas como un problema CSP. Debes definir en el paquete `aima.gui.nqueens.csp` las siguientes clases: `NQueensAssignment`, `NQueensConstraint`, `NQueensVariable`, `NQueensProblem`, y `NQueensMinConflictApp` que utilizará la estrategia `MinConflictsStrategy`.

Explica si el algoritmo encuentra siempre la solución.

Ejemplo de ejecución:

```
{Reina en columna [0]=5, Reina en columna [1]=2, Reina en columna [2]=6,
Reina en columna [3]=1, Reina en columna [4]=3, Reina en columna [5]=7,
Reina en columna [6]=0, Reina en columna [7]=4}

Time to solve = 0.005segundos

SOLUCION:
------Q-
---Q----
-Q------
----Q---
-------Q
Q-------
 --Q-----
 -----Q--
```

Debes generar una carpeta con nombre tu NIP, dentro habrá tres carpetas tarea1, tarea2 y tarea3 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).

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: 19 noviembre 2019 a las 23:59.