Práctica: El Problema Hashi (Puentes)

Vamos a ver cómo usar un SAT-solver moderno para resolver puzles Hashi.

1. Problema

Hashi (en japonés, significa 'puentes') es un rompecabezas lógico publicado por Nikoli. La versión decisional es un problema NP-completo. El puzle está formado por varias islas dispuestas en filas y columnas horizontales y verticales. El objetivo es conectar todas las islas mediante puentes cumpliendo las siguientes reglas:

- Los puentes deben ir en línea recta (horizontal o vertical)
- Los puentes no pueden cruzarse entre sí
- Los puentes no pueden pasar por encima de otras islas
- Cada par de islas puede estar conectado por un máximo de dos puentes
- Cada isla tiene un número que indica el número de puentes que deben conectarse a ella
- Todas las islas deben formar una única red de puentes conectada

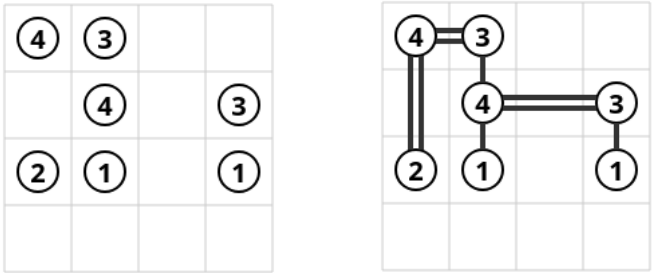
Formalización matemática:

Dado: un conjunto de n islas, cada isla $i \in [0..n-1]$ con coordenadas (x_i, y_i) y con un número r_i de puentes requeridos, encontrar: un conjunto de puentes E con las siguientes restricciones:

- Cada puente es horizontal o vertical
- Los puentes no pueden cruzarse
- Cada par de islas esté conectada, como máximo, a través de 2 puentes
- Cada isla debe tener exactamente r_i puentes
- El grafo formado debe ser conexo

Ejemplo:

El siguiente juego tiene 7 islas dispuestas en una cuadrícula de 4x4, cada una en las coordenadas (0,0), (0,2), (1,0), (1,1), (1,2), (3,1), y (3,2). El número en cada isla indica el número de puentes que deben conectarse a ella.



2. Explicación de la reducción (Hashi $\leq_p$ SAT)

Supongamos que tenemos un puzle Hashi con n islas. Queremos reducir el problema a SAT, es decir, construir una fórmula CNF, de forma que el puzle tenga solución si y solo si la fórmula sea satisfactible.

Variables

Para codificar el problema, necesitamos las siguientes variables:

- **Variables de puente:** $b1_{i,j}$ y $b2_{i,j}$, donde (i, j) representa un par de islas. La variable booleana $b1_{i,j}$ indica que hay 1 puente entre ellas, $b2_{i,j}$ indica que hay 2 puentes entre ellas. Por lo tanto, $(\neg b1_{i,j} \wedge \neg b2_{i,j}))$ significa que no hay puente entre las dos islas. Se crean todas estas variables para cada arista posible. Esto es, en aquellos casos donde las islas estén alineadas horizontal o verticalmente, y además no haya islas en medio.

$$E = \{(i, j) \mid i, j \in [0..n-1] \text{ con } i < j \text{ alineadas y sin islas en medio}\}$$

- **Variables de conexion:** $reach_i$, para garantizar la conectividad del grafo. La variable $reach_i$ indica que se puede alcanzar la isla i a través de los puentes.

Restricciones

Debemos construir una fórmula CNF cumpliendo las siguientes restricciones:

(a) **Valores válidos de puente:** Se debe asegurar que no existe un segundo puente si no se dispone del primero.

$$\forall (i, j) \in E : (b2_{i,j} \Rightarrow b1_{i,j})$$

(b) **Prohibición de cruces:** Dos aristas no pueden cruzarse. Si $(i, j) \cap (i', j')$ representa la intersección geométrica entre dos aristas.

$$\forall (i, j), (i', j') \text{ aristas en } E, \text{ con } (i, j) \cap (i', j') \neq \emptyset : (\neg b1_{i,j} \vee \neg b1_{i',j'})$$

(c) **Cumplimiento del número de puentes:** Cada isla debe tener exactamente el número de puentes requerido.

$$\forall i \in [0..n-1] : \sum_{j:(i,j)\in E} (b1_{i,j} + b2_{i,j}) + \sum_{j:(j,i)\in E} (b1_{j,i} + b2_{j,i}) = r_i$$

(d) **Garantía de conectividad:** Todas las islas deben estar conectadas. La formalización de esta situación debería incluir, entre otras, restricciones del tipo:

- La primera isla es siempre accesible ($reach_0$)
- Las demás islas deben estar también conectadas.

$$\forall i \in [1..n-1] : (reach_i \Leftrightarrow [(\bigvee_{j:(i,j)} (b1_{i,j} \wedge reach_j) \vee (\bigvee_{j:(j,i)} (b1_{j,i} \wedge reach_j)]$$

3. Métodos para usar el SAT-solver

Usaremos el SAT-solver Glucose3 disponible en Python. Para ello disponemos de varios métodos:

Módulos e inicialización

```
from pysat.solvers import Glucose3
from pysat.formula import CNF, IDPool
from pysat.card import CardEnc
variables = IDPool()
```

El módulo `pysat.formula` contiene métodos para manipular CNFs y un gestor de variables denominado `IDPool`.

El módulo `pysat.card` permite escribir restricciones numéricas sin necesidad de codificarlas con variables booleanas.

El método `IDPool()`: inicializa el gestor de variables y devuelve su identificador. En este caso, el gestor se llama `variables`. A partir de ahora `variables` será nuestro gestor.

Creación de variables

```
positive = variables.id(("b1", p_1, p_2, ...))
("b1", p_1, p_2, ...)) = variables.obj(positive)
```

El método `id()` del gestor de variables asigna un número positivo a un objeto. Si el objeto ya tenía un número previamente asignado, no genera uno nuevo, sino que devuelve el existente.

La operación contraria se realiza con el método `obj()`, el cual devuelve el objeto asociado a un número positivo. Si no hay ningún objeto asociado a ese número, devuelve `None`.

Manipulacion de CNFs

```
cnf = CNF()
new_cnf = cnf.append(list)
new_cnf = cnf.extend(cnf*)
```

El método `CNF()` inicializa una CNF y devuelve su identificador. En este caso, la CNF se llama `cnf`. Para visualizar una CNF creada con este método es necesario convertirla primero a una lista (`print(list(cnf))`).

El método `append()`, dada una lista de literales (enteros salvo el 0), la añade como una nueva cláusua a `cnf`.

El método `extend()`, dada una CNF (`cnf*`), añade todas sus cláusulas a `cnf`.

Restricciones numéricas

```
cnf = CardEnc.equals(lits, z, vpool=variables)
```

El método `equals()`, dada una lista de positivos que codifican variables (`lits`), un entero `z`, y un gestor de variables, devuelve una CNF que codifica la restricción:

$$\sum_{n \in lits} n = z$$

Al pasar el gestor de variables es necesario indicar que el parámetro `vpool` es nuestro gestor de variables.

Ejecución del SAT-solver

```
with Glucose3(bootstrap_with=cnf) as sat:
   if sat.solve():
     #tenemos una asignacion a las variables
     solution = sat.get_model()
   else:
     #sat.get:model() es None
```

La llamada al SAT-solver se hace pasando la `cnf` que hemos construido y dando un nombre al SAT-solver. En nuestro caso el SAT-solver se llama `sat`.

El método `solve()` busca una asignación que haga cierta a la `cnf`. Devuelve `True` si la encuentra o `False` si no.

El método `sat.get_model()` devuelve la asignación que hace cierta a la `cnf` o `None` si esa asignación no existe.

4. Tareas a Realizar

Para resolver correctamente el puzzle Hashi, con los módulos:

```
from pysat.solvers import Glucose3
from pysat.formula import CNF, IDPool
from pysat.card import CardEnc
```

El gestor de variables:

```
variables = IDPool()
```

debes programar la siguiente función:

(a) `solve_hashi_true_sat(dimensions, islands_data)`

Entrada:

`dimensions`: lista con las coordenadas de cada isla [anchura, altura]

`islands_data`: lista de islas, cada una con [x, y, puentes_requeridos]

Salida:

Un diccionario con la solución en el siguiente formato:

```
{
    "width": int,
    "height": int,
    "islands": lista de islas de la entrada
    "solution": [
        {
            "id1": int, "x1": int, "y1": int,
            "id2": int, "x2": int, "y2": int,
            "bridges": int (1 o 2)
        },
        ...
    ]
}
```

`None` si no hay salida

Pasos de Implementación: Con las listas:

```
nodes = [[isle[0], isle[1]] for isle in islands_data]
required_bridges = [isle[2] for isle in islands_data]
```

Crear la CNF: `cnf = CNF()`.

Identificar aristas: Determinar qué pares de islas pueden conectarse (en línea horizontal o vertical, sin otras islas en medio) y construir el conjunto de aristas `edges`.

```
edges = construct_edges(nodes)
```

(a) Añadir valores válidos de puente: Para cada arista (i, j) de `edges` añadir la restricción $(b2_{i,j} \Rightarrow b1_{i,j})$.

```
cnf = add_bridge_2_implies_bridge_1(edges, cnf)
```

(b) Añadir prohibición de cruces: Para cada par de aristas (i, j) e (i', j') de `edges`, comprobar si se cruzan geométricamente. Si es así añadir la restricción $(\neg b1_e \vee \neg b1'_e)$.

```
cnf = add_crossing_constraints(nodes, edges, cnf)
```

(c) Añadir el cumplimiento del número de puentes: Para cada isla i, añadir una restricción que garantice que la suma de todos sus puentes coincide con el número de puentes requerido r_i.

$$\forall j \in [0..n-1] : \sum_{j:(i,j)\in E} (b1_{i,j} + b2_{i,j}) + \sum_{j:(j,i)\in E} (b1_{j,i} + b2_{j,i}) = r_i$$

```
cnf = add_required_bridges_contraints(nodes, edges,
                                      cnf, required_bridges)
```

(d) Añadir restricciones de conectividad: Esta función no hace falta implementarla porque esta hecha en el fichero `implemented_functions`.

```
cnf = add_connectivity_constraints(nodes, edges, cnf,
                                   variables)
```

Formatear la Salida: Esta función no hace falta implementarla porque esta hecha en el fichero `implemented_functions`.

```
formated_sol(dimensions, nodes, edges, solution, variables)
```

5. **Casos de prueba**

Dispones de un test formado por 6 puzles pequeños. Además, ejecutando el programa `main.py`, podrás probar una serie de 100 puzles de 10x10 que se resuelven rápido. Están disponibles en la carpeta `mypuzzles`. Existe otra carpeta, `big_puzzles`, con un puzle de 33x33 y otro de 110x110. Este último necesita mucho tiempo para su resolución.

El programa `main.py` deja las soluciones en la carpaeta `solutions` en formato HTML para poder visualizarlas.

6. **Documentación sobre PySAT**

`https://pysathq.github.io/docs/html/`