Solving Hashiwokakero in Prolog

Pascal Chatterjee and Joakim Carselind
{joacar, pascalc}@kth.se

October 9, 2012

Introduction

Hashiwokakero is a Japanese puzzle that involves connecting islands with bridges under strict constraints:

1. An island can only have bridges to horizontally/vertically adjacent islands.
2. There can be at most two bridges between islands.
3. The arity of an island must exactly equal its number (one to eight).
4. Bridges cannot cross each other.
5. The solution must form a connected component.

Hashiwokakero is similar to Sudoko and Sudoko can be reduced to graph coloring which is NP-complete so Hashiwokakero probably is NP-complete too. As Sudoko, Hashiwokakero can be formulated as a constraint satisfaction problem.

Constraint satisfaction problem

We need to encode the constraints one to five above to a readable form and readable ought to be viewed from Prologs perspective. Prolog is developed from first-order logic and is a declarative programming paradigm built on rules, facts and a powerful inference engine.

The problem

The input is a map with size $m \times n$ containing a number of islands and potential bridge locations that naturally can be viewed as water. Let (x, y) be the coordinate and definie $(1, 1)$ to be the upper left corner. x increases to the right and y increases down. Below is an valid input map where the digits is the arity of the island at that location

```
5 5
2 . 1 . .
. . . . .
4 . 3 . 1
. . . . .
3 . . . 2
```

The solution is the same map but with a legal configuartion of bridges connecting the islands. Below is an example (there is a second, can you spot it?)

```
5 5
2 - 1 . .
| . . . .
4 = 3 - 1
| . . . .
3 = = = 2
```

Technical description

We wrote two python scripts to faciliate the input/output handling since this is not Prologs strong suits. The first script (input.py) takes the input map and outputs a prolog knowledge database that represents the data needed to solve the problem.

```
rows(5).
columns(5).
grid([
    [[1, 1], 2],
    [[1, 3], 1],
    [[3, 1], 4],
    [[3, 3], 3],
    [[3, 5], 1],
    [[5, 1], 3],
    [[5, 5], 2]
```

```
]).
```

Prolog outputs the solution like below

```
[
  [[0,0], [2,0], 1],
  [[0,0], [0,2], 2],
  [[0,2], [2,2], 2],
  [[0,2], [0,4], 1],
  [[2,2], [2,4], 1],
  [[0,4], [4,4], 2]
]
```

and this is prettyfied by output.py into a more human-friendly form.

A first attempt

A first attempt, a naïve approach if one so like, to solve the problem is to generate all possible configurations and test each one of them until a configuration satisifying the constraints is found and accepted as the solution. The combinatorial nature of the problem results in a giant search space where lots of compution time is spent on "stupid" possible solutions. Clearly, we could do better.

A intelligent attempt

Taking a functional programming view, the problem could be viewed as reducing a list of islands with a well constructed function and accumulate the result to produce a solution list of the type above. In this function, our aim is to continously narrow the search space by using a technique in CSP called forward-checking. In Prolog one would say that we "push the tester into the generator". The closer into the generator we can put the tester, the search space will be smaller since we do not validate incorrect solutions (in as large extent as in the naïve approach) in the tester. By cutting away uninteresting search branches we can reduce the time solving the puzzle dramatically.

Here is a strategy to perform forward checking on each constraint.

1. If we add the exact number of bridges that an island is required to have when considering it in the reduction, this constraint is also fulfilled. If the number of islands already bridging to this island is too many, or if we cannot make the number of new bridges we need, we can fail right away and backtrack.

2. We can restrict the number of bridges between islands to the domain $\{0, 1, 2\}$ when considering the bridges to add to an island.

3. In our reducing relation, if we only consider building bridges to islands that are horizontally or vertically adjacent to the current one, we will only produce solutions consistent with this constraint.

4. When trying to place a new bridge, we can take care to make sure it does not cross one of the bridges already placed.

5. If we traverse the list of islands so that each successive island is reachable from the previous one, and place only those bridges that extend the accumulated component, then the final solution will also be connected.

Conclusion

One important observation was that predicates that we thought was determinate was not. Those predicates polluted the search space with redundant configurations. In interest of effeciency we used the non-relational cut operator which mean that the order predicates are definied does matter. In our solution we did not make use of any SICStus library that allows for our program to be executed by any other prolog system. (This was great since our SICStus license expired while writing this report and it runs perfectly fine under SWI).

We are well aware the SICStus provides both a graph library and more importantly a CSP library which we refrained from using as it would made this report not worth writing.

The complete solution can be found in the Git repository `https://github.com/joacar/logik12_project.git`.