CSCI5240 - Combinatorial Search and Optimization with Constraints Fall 2016

Teacher: Prof. Jimmy Lee

Assignment 10

Due: 11 Dec, 2016

1. *The Knight's Tour*

   A Knight's tour is a sequence of moves where a Knight visits every cell of an $n \times n$ chessboard exactly once in n^2 moves. If the last cell that the Knight visits is reachable from the starting cell by one Knight's move, such a tour is called a *closed* one. Otherwise, it is an *open* one. Given a particular cell on an $n \times n$ chessboard, we aim at finding a closed Knight's tour that starts from the cell.

   A possible CSP model of the problem is as follows. We label each cell using numbers from 1 to n^2, and we have n^2 variables x_0, x_1, ... x_{n^2-1} with the identical domain $[1..n^2]$, where x_i indicates the number of the cell the Knight visits in the i-th move. Obviously, all variables should take different values, and the values of two consecutive variables should form a legal Knight's move. To model that, we define a new constraint $move(x, y, n)$, which returns true if x and y forms a legal Knight's move in an $n \times n$ chessboard and false otherwise.

   Figure 1 shows a partially completed Knight's tour in a 6×6 chessboard starting from the top left corner, where the number i appearing in a cell means the Knight will reach the cell at the i-th move.

   Design and implement the *move* constraint as a user-defined constraint by implementing a propagator that ensure the *move* constraint is *arc consistent* after propagation. Implement the model using *only* the *all-different* constraint and the *move* constraint. Find all possible Knight's tours that coincide with the configuration in Figure 1.

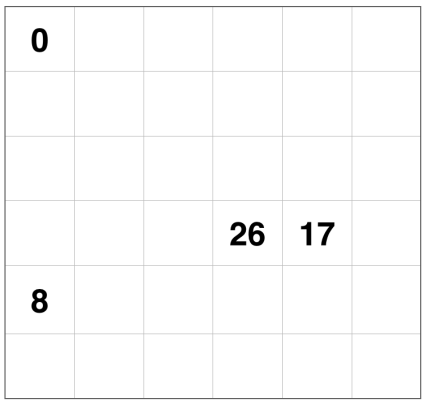

Figure 1: An instance of *the Knight's Tour Problem*

2. *The n-Queens Problem*

   The n-Queens Problem is a well known and classic problem in Constraint Programming. Solving the n-Queens Problem behaves dramatically differently under the use of different search heuristics. One commonly used search heuristic is the first-fail, minimum-domain-value heuristic: select the variable with the minimum domain size (ties are broken by lexicographic order) and select the minimum current domain value from the variable during search. It can be posted in Gecode by calling branch(*this, x, `INT_VAR_SIZE_MIN()`, `INT_VAL_MIN()`), where x is a variable array. Another commonly used heuristic, first-fail, median-domain-value heuristic, differs in the value selection: it selects the median of current domain values from the chosen variable. In the case of an even number of domain values (say m), it selects the value ranking $m/2$ in increasing order of values.

(a) Implement the first-fail, median-domain-value heuristic in Gecode. You are not allowed to use the built-in value selection strategy `INT_VAL_MED()` directly. Compare the runtime and the number of failures of first-fail, median-domain-value heuristic with those of first-fail, minimum-domain-value heuristic by testing on different sizes of the problem. Start the test from $n = 10$, and increase by 10 each time until it reaches 200 (or any larger number you want). Summarise your result in two tables (each for one metric) in the following form (for example, runtime in milliseconds):

n	first-fail, minimum-domain-value	first-fail, median-domain-value
10	0.826	0.802
20	1.980	1.632
30	3.003	3.971
...	...	...

(b) Similarly, search heuristics in MiniZinc can be specified using search annotations. Check Chapter 6.2 of MiniZinc tutorial for instructions about how to specify the two heuristics just mentioned. Compare the runtime and the number of failures of these two search heuristics in MiniZinc and summarise the result in tables.