Knowledge Representation and Reasoning

Exercise Session 4

Teaching assistant: Robbe Van den Eede (robbe.vandeneede@kuleuven.be)

The solutions can be found here.

1 4-Colouring of a graph

For the ontology, we can use a typed vocabulary, where Node and Colour are the types.

The vocabulary also contains:

- a relation $\mathit{edge} \subseteq \mathit{Node} \times \mathit{Node}$, containing the pairs (n_1, n_2) for which the nodes n_1 and n_2 are neighbours in the graph.
- a total function $\mathit{colouring} : \mathit{Node} \to \mathit{Colour}$ mapping each node to its assigned colour.

Then the theory consists of a single assertion:

$$\forall n_1, n_2 \in \mathit{Node}\ : \mathit{edge}(n_1, n_2) \implies \mathit{colouring}(n_1) \neq \mathit{colouring}(n_2).$$

Note that, in this case, the structure does not have to enumerate the edges both ways ($\mathit{edge}(A, B)$ and $\mathit{edge}(B, A)$), because the quantification over all pairs will ensure that the colors of adjacent nodes are distinct.

2 Sudoku

For the ontology, we can use a typed vocabulary, where Num and Block are the types. For a standard sudoku, Num ranges from 1 to 9 and Block from 1 to 3.

The vocabulary also contains:

- a function $\mathit{square} : \mathit{Num} \times \mathit{Num} \to \mathit{Num}$ mapping a square (=cell in the matrix) to the value on the square,
- a relation $\mathit{input\_square} : \mathit{Num} \times \mathit{Num} \times \mathit{Num} \to \mathit{Bool}$ expressing the input to the sudoku problem, containing the values that are already present. We cannot use a total function here, as the input grid is not fully filled (but we could use a partial function as well).
- a function $\mathit{map\_to}$: $\mathit{Num} \to \mathit{Block}$ mapping a row or column to the block it belongs to.

Remember: a grid is best represented by a function from a pair of coordinates (row and column) to a description of its content (here, a number).

The theory must contain the following restrictions:

- The solution must extend the input:

$$\forall r, c, n \in \mathit{Num} : \mathit{input\_square}(r, c, n) \implies \mathit{square}(r, c) = n$$

- Two different cells in the same row must contain different numbers:

$$\forall r, c_1, c_2 \in \mathit{Num} : c_1 \neq c_2 \implies \mathit{square}(r, c_1) \neq \mathit{square}(r, c_2)$$

- Two different cells in the same column must contain different numbers:

$$\forall r_1, r_2, c \in \mathit{Num} : r_1 \neq r_2 \implies \mathit{square}(r_1, c) \neq \mathit{square}(r_2, c)$$

- Two different cells in the same square must contain different numbers:

$$\begin{aligned}&\forall\, r_1, r_2, c_1, c_2 \in \mathit{Num} : (r_1 \neq r_2 \vee c_1 \neq c_2)\\ &\quad \wedge \mathit{map\_to}(r_1) = \mathit{map\_to}(r_2) \wedge \mathit{map\_to}(c_1) = \mathit{map\_to}(c_2)\\ &\quad \implies \mathit{square}(r_1, c_1) \neq \mathit{square}(r_2, c_2)\end{aligned}$$

Alternatively, the final three constraints can be reformulated in the following equivalent way:

$$\forall r, c_1, c_2 \in \mathit{Num} : \mathit{square}(r, c_1) = \mathit{square}(r, c_2) \implies c_1 = c_2$$

$$\forall r_1, r_2, c \in \mathit{Num} : \mathit{square}(r_1, c) = \mathit{square}(r_2, c) \implies r_1 = r_2$$

$$\begin{aligned}&\forall\, r_1, r_2, c_1, c_2 \in \mathit{Num} : \mathit{map\_to}(r_1) = \mathit{map\_to}(r_2) \wedge \mathit{map\_to}(c_1) = \mathit{map\_to}(c_2)\\ &\quad \wedge \mathit{square}(r_1, c_1) = \mathit{square}(r_2, c_2)\\ &\quad \implies r_1 = r_2 \wedge c_1 = c_2\end{aligned}$$

Part of the input is:

$$\mathit{map\_to} = \{1 \to 1, 2 \to 1, 3 \to 1, 4 \to 2, 5 \to 2, 6 \to 2, 7 \to 3, 8 \to 3, 9 \to 3\}$$

An alternative solution uses cardinality constraints:

```
// the given input
! r, c in Num, n : InputSquare(r,c,n) => Square(r,c) = n.

// each number occurs once in a row
! r,y in Num: #{c: Square(r, c)=y} = 1.

// each number occurs once in a column
! c,y in Num: #{r: Square(r, c)=y} = 1.

// each number occurs once in a block
! b1, b2 in Block, y in Num:
   #{r,c in Num: Square(r,c)=y & MapTo(r)=b1 & MapTo(c)=b2} = 1.
```

The bigger sudoku can be solved using the same theory, by adapting the values for the types Num and Block and for the functions $\mathit{map\_to}$ and $\mathit{input\_square}$.

3 Travelling salesman problem

For the ontology, we can use a typed vocabulary with one type Vertex.

The vocabulary contains:

- a predicate $edge : Vertex \times Vertex \to Bool$ indicating that there is a road from one city to another (a two-way road is represented by two facts: one for each direction),
- a constant $start : () \to Vertex$ representing the starting city of the tour,
- a function $next : Vertex \to Vertex$ mapping a city to the next city in the tour,
- a predicate $reachable : Vertex \to Bool$ containing all cities that are reachable from the starting city when following the tour.

The theory consists of the following formulas:

$$\forall v \in Vertex : edge(v, next(v))$$

$$\forall v : reachable(v)$$

$$\left\{ \begin{array}{l} reachable(start()) \\ \forall v \in Vertex : reachable(next(v)) \leftarrow reachable(v) \end{array} \right\}$$

$$\exists v : next(v) = start()$$

Note that the use of a function $next/1$ implies that from every city we go to exactly one other city. Together with the constraint that all cities are reached by our tour ($\forall x : \ Reachable(x)$), this entails that each city is visited exactly once on our tour (as desired).
If we would use a binary predicate $next/2$ instead of the function $next/1$, we would have to specify explicitly that each city is visited only once on the tour. (Without such constraint, we could otherwise find solutions in which some cities are visited twice on the tour, or more.)

4 N-Queens

One possible modelling consists of the following vocabulary, with:

- types Row and Col, containing the possible rows as columns respectively,
- a predicate $queen : Row \times Col \to Bool$ containing the positions (r, c) on which a queen is placed.

A corresponding theory is the following:

$\forall r \in Row : \exists c \in Col : queen(r, c)$

Twosquares(r1,c1)and(r2,c2)areonthesamediagonalifandonlyif abs(r1-r2)==abs(c1-c2)

$\forall c \in Col : \exists r \in Row : queen(r, c)$

$\forall r \in Row : \forall c_1, c_2 \in Col : queen(r, c_1) \land queen(r, c_2) \implies c_1 = c_2$

$\forall r_1, r_2 \in Row : \forall c \in Col : queen(r_1, c) \land queen(r_2, c) \implies r_1 = r_2$

$\forall r_1, r_2 \in Row : \forall c_1, c_2 \in Col : queen(r_1, c_1) \land queen(r_2, c_2) \land r_1 \neq r_2 \implies abs(r_1 - r_2) \neq abs(c_1 - c_2)$

The last formula says that 2 different queens cannot on the same (upward or downward) diagonal.

Alternative solutions (there are many):

- You may have used the contraposition of an implication.
- One other way is to create a predicate $queen\_placed : Row \times Col \times Queen \to Bool$, with an added type $Queen$. Although it might be more straightforward to encode it like this, generating its models will be much slower now because we are 'identifying' queens (by introducing a type $Queen$, and N constants of this type), while this is unnecessary (because all N queens are completely interchangeable, so there is no reason why we need to talk about having *a particular* queen on some square as opposed to only talking about having *some* queen at that square, as in the above solution).

- Another approach is to use a function $queen : Row \rightarrow Col$. This enforces the rule that there is one queen per row. The theory becomes much simpler:

$$\begin{aligned}
&\forall r_1, r_2 \in Row : r_1 \neq r_2 \implies queen(r_1) \neq queen(r_2)\\
&\forall r_1, r_2 \in Row : r_1 \neq r_2 \implies abs(r_2 - r_1) \neq abs(queen(r_2) - queen(r_1))
\end{aligned}$$

  A possible downside to this variant is that it extends less easily to a wider theory of "queen problems" in chess.

- Yet another approach is to use cardinality constraints:

```
vocabulary V{
    type Row
    type Col
    queen: Row*Col→ Bool
}
theory T: V{
    ! r in Row: #{c in Col: queen(r,c)} = 1.
    ! c in Col: #{r in Row: queen(r,c)} = 1.
    ! (r1, c1) in queen:
       #{(r2, c2) in queen: abs(r1-r2) = abs(c1-c2)} = 1.
}
structure S:V{
    Row := {1..8}.
    Col := {1..8}.
}
procedure main() {
    pretty_print(model_expand(T, S, max=100))
}
```

A pure Prolog program would not contain explicit quantifications nor types, and would consist of a list of rules, which have operational semantics: the semantics are defined by how Prolog operates and how the program is structured. The above theory instead is independent of the inference used.