UNIVERSITEIT GENT

Formele Logica Academiejaar 2014-2015

Project Alloy

Jasper D'haene <jasper.dhaene@ugent.be>
Florian Dejonckheere <florian@floriandejonckheere.be>

1 Ramsey-getallen

Ramsey's theorem dicteert dat in een voldoende grote set waarvan de bogen gekleurd zijn met een willekeurig aantal kleuren, monochromatisch gekleurde subsets te vinden zijn. Stel $m, n \in \mathbb{R}^+$ en twee kleuren $k_i, i \in [0, 1]$, dan definieert Ramsey's theorem $R(m, n, k)$ de ondergrens voor de complete graaf die een subset over tenminste m toppen met kleur k_0, of een subset over tenminste n toppen met kleur k_1 bevat.

Informeel is dit probleem (voor $k = 2$) ook wel bekend als het *party problem*: hoe groot is de minimale set van personen die uitgenodigd moeten worden voor een feestje, waarvoor geldt dat ofwel minstens m personen elkaar (mutueel) kennen, ofwel n personen elkaar (mutueel) niet kennen.
Het Ramseygetal wordt bepaald door het uitvoeren van één of meerdere instanties van het predikaat *Sub_Graph*. Dit predikaat controleer of er disjuncte subsets te vinden zijn van verschillende kleuren. Als er bij dit predikaat een instantie wordt gevonden, is het Ramseygetal gevonden. De parameters van het algoritme zijn in te stellen op de laatste regels. Stel het volgende Ramseygetal:

$$R(m, n; k) = v \tag{1}$$

Dan kunnen de programmaparameters als volgt geschreven worden:

```
...

run {
        Sub_Graph[m] // Of Sub_Graph[n]
} for k Colour, exactly v Node, exactly (v * (v - 1)) Edge
```

Wanneer het algoritme wordt uitgevoerd voor $R(3, 3) \equiv R(3, 3; 2)$ kan men door middel van iteratie het Ramseygetal bepalen. Dit wordt uiteindelijk vastgesteld als

$$R(3, 3) = 6 \tag{2}$$

Ditzelfde proces herhalen we voor $R(3, 3, 3)$.

$$R(3, 3, 3) = 17 \tag{3}$$

Listing 1: Ramseygetallen Alloy model

```
/**
 * ramsey_numbers.als - Calculating Ramsey numbers in Alloy 4
 *
 * Florian Dejonckheere <florian@floriandejonckheere.be>
 * Jasper D'haene <jasper.dhaene@gmail.com>
 *
 * */

/**
 * Om dit probleem wat interessanter en modelleerbaarder
 * te maken, stellen we dat de te calculeren Ramseyget∀en
 * gewoon het antwoord zijn van het 'party problem':
 *
 * R(m,n) = Hoeveel gasten moeten er uitgenodigd worden
 * zodat ≥ m elkaar kennen en ≥ n elkaar niet kennen.
 *
 * Hint: R(3, 3) = 6 [1]
 * Hint: R(3, 3) = R(3, 3, 2)
 * Hint: R(3) = R(3, 3, 3) = R(3, 3, 3; 3)
 * Hint: R(3, 3, 3) = 17 [2]
 *
 * [1] http://mathworld.wolfram.com/RamseyNumber.html
 * [2] http://en.wikibooks.org/wiki/Combinatorics/Bounds_for_Ramsey_numbers
 *
 * */

sig Colour {}
sig Node {}

sig Edge {
        connection: Node -> Node,
        colour: one Colour
}{
        // No self-referencing
        ∀ node:Node • (node -> node not in connection)
}

// Make sure that 'colouring' is the same as Edge->colour
fact {
        colour = ~(Graph.colouring)
}

// Make sure symmetric relations have the same colour
fact {
        ∀ e: Edge • some e': Edge • {
                e.connection = ~(e'.connection) ∧ e.colour = e'.colour
        }
}

one sig Graph{
        nodes: set Node,

        // Edges: Node -> Node
        edges: set Edge,
        colouring: Colour one -> some Edge
}{
        // All nodes in graph
        ∀ node: Node • node in nodes

        // All edges in graph
        ∀ edge:Edge • edge in edges

        // Edges relationship is symmetrical
        ∀ edge: Edge • some edge':Edge • edge.connection = ~(edge'.connection)

        // Every edge only connects 2 points
        ∀ edge: Edge • one edge.connection

        // Complete graph
        ∀ n, n' : Node • some e:Edge • n ≠ n' ⟹ n -> n' in e.connection
}
```

```
pred Sub_Graph [ X:  Int] {
        some col: Colour • some edges_set:  set col.(Graph.colouring) •
                #edges_set = X
                ∧ No_Symmetry[edges_set] // Geen A->B en B->A in edges_set.
                ∧ Mutual_Friends[edges_set] // Alle edges zijn op een willekeurige manier verbonden met elkaar
}

pred No_Symmetry[edges_set:set Edge]{
        ∀ edge: edges_set • ∀ edge':(edges_set-edge) • edge.connection ≠ ~(edge'.connection)
}

fun Nodes_In_Set[edges:set Edge]: set Node{
        {
                n: Node • some n':Node • {
                        n ->n' in edges.connection ∨ n'->n in edges.connection
                }
        }
}

pred Mutual_Friends[edges_set:set Edge]{
        ∀ node: Nodes_In_Set[edges_set] •  ∀ node': (Nodes_In_Set[edges_set] - node) •
        node->node' in edges_set.connection ∨ node'->node in edges_set.connection
}

/**
 * #Edge should be #Node*(#Node-1)
 */

// R(r,s) = R(5,2) = 5
run {
        // Input X = (r * (r-1))/2. Choose either r or s
        Sub_Graph[10]
}
for 2 Colour, exactly 5 Node, exactly 20 Edge
```

2 Cyclische toren van Hanoi

De cyclische toren van Hanoi is een variant op de bekende combinatorische puzzel. Buiten de drie hoofdregels geldt de volgende regels ook:

4. Schijven kunnen enkel cyclisch naar rechts opgeschoven worden. Voor drie palen A, B, C is dit dus $A \to B \to C \to A$

Als basis voor het algoritme werd de geïmplementeerde versie van Ilya Shlyakhter beschouwd. Deze versie wordt bijgeleverd als voorbeeld bij de Alloy Analyzer, maar implementeert de niet-cyclische variant.

Listing 2: Cyclische toren van Hanoi Alloy model

```
module examples/puzzles/hanoi

/**
 * Cyclic towers of Hanoi model
 *
 * Author of hanoi model: Ilya Shlyakhter
 * Modified by Jasper D'haene <jasper.dhaene@gmail.com>
 * */

open util/ordering[State] as states
open util/ordering[Stake] as stakes
open util/ordering[Disc] as discs

sig Stake { }
sig Disc { }

/**
 * sig State: the complete state of the system --
 * which disc is on which stake.  An solution is a
 * sequence of states.
 */
sig State {
        on: Disc -> one Stake  // _each_ disc is on _exactly one_ stake
        // note that we simply record the set of discs on each stake --
        // the implicit assumption is that on each stake the discs
        // on that stake are ordered by size with sm∀est disc on top
        // and largest on bottom, as the problem requires.
}

/**
 * compute the set of discs on the given stake in this state.
 * ~(this.on) map the stake to the set of discs on that stake.
 */
fun discsOnStake[st: State, stake: Stake]: set Disc {
  stake.~(st.on)
}

/**
 * compute the top disc on the given stake, or the empty set
 * if the stake is empty
 */
fun topDisc[st: State, stake: Stake]: lone Disc {
  { d: st.discsOnStake[stake] • st.discsOnStake[stake] in discs/nexts[d] + d }
}

/**
 * Describes the operation of moving the top disc from stake fromStake
 * to stake toStake.  This function is defined implicitly but is
 * nevertheless deterministic, i.e. the result state is completely
 * determined by the initial state and fromStake and toStake; hence
 * the "det" modifier above.  (It's important to use the "det" modifier
 * to tell the Alloy Analyzer that the function is in fact deterministic.)
 */
pred Move [st: State, fromStake, toStake: Stake, s': State] {
        //modified: Cyclic move property
        fromStake ≠ stakes/last ⟹
```

```
                fromStake.next = toStake
        else
                toStake = stakes/first

        let d = st.topDisc[fromStake] • {
                // ∀ discs on toStake must be larger than d,
                // so that we can put d on top of them
                st.discsOnStake[toStake] in discs/nexts[d]
                // after, the fromStake has the discs it had before, minus d
                s'.discsOnStake[fromStake] = st.discsOnStake[fromStake] - d
                // after, the toStake has the discs it had before, plus d
                s'.discsOnStake[toStake] = st.discsOnStake[toStake] + d
                // the remaining stake afterwards has exactly the discs it had before
                let otherStake = Stake - fromStake - toStake •
                        s'.discsOnStake[otherStake] = st.discsOnStake[otherStake]
        }
}

/**
 * there is a leftStake that has ∀ the discs at the beginning,
 * and a rightStake that has ∀ the discs at the end
 */
pred Game1 {
        Disc in states/first.discsOnStake[stakes/first]
        //modified end condition
        some finalState: State • Disc in finalState.discsOnStake[stakes/last] ∧ (finalState =
states/last)

        // each adjacent pair of states are related by a valid move of one disc
        ∀ preState: State - states/last •
                let postState = states/next[preState] •
                        some fromStake: Stake • {
                                // must have at least one disk on fromStake to be able to move
                                // a disc from fromStake to toStake
                                some preState.discsOnStake[fromStake]
                                // post- results from pre- by making one disc move
                                some toStake: Stake • preState.Move[fromStake, toStake, postState]
                        }
}

/**
 * There is a leftStake that has ∀ the discs at the beginning,
 * and a rightStake that has ∀ the discs at the end
 */
pred Game2 {
        Disc in states/first.discsOnStake[stakes/first]
        some finalState: State • Disc in finalState.discsOnStake[stakes/last]

        // Each adjacent pair of states are related by a valid move of one disc
        ∀ preState: State - states/last •
                let postState = states/next[preState] •
                        some fromStake: Stake • {
                                let d = preState.topDisc[fromStake] • {
                                        // Must have at least one disk on fromStake to be able to move
                                        // a disc from fromStake to toStake
                                        some preState.discsOnStake[fromStake]
                                        postState.discsOnStake[fromStake] = preState.discsOnStake[fromStake] - d
                                        some toStake: Stake • {
                                                // post- results from pre- by making one disc move
                                                preState.discsOnStake[toStake] in discs/nexts[d]
                                                postState.discsOnStake[toStake] = preState.discsOnStake[toStake] +
                                                // the remaining stake afterwards has exactly the discs it had bef
                                                let otherStake = Stake - fromStake - toStake •
                                                        postState.discsOnStake[otherStake] = preState.discsOnStake
                                        }
                                }
                        }
                }

// #state == 22 + 3X voor geldige instantie.
run Game1 for 1 but 3 Stake, 3 Disc, 22 State expect 1
run Game2 for 1 but 3 Stake, 3 Disc, 8 State expect 1
```