

## 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:

```
1 ...  
2  
3 run {  
4     Sub_Graph[m] or Sub_Graph[n]  
5 } for k Colour, exactly v Node, exactly (v * (v - 1)) Edge
```

Listing 1: Ramseygetallen Alloy model

```

1  /**
2   * ramsey_numbers.als - Calculating Ramsey numbers in Alloy 4
3   *
4   * Florian Dejonckheere <florian@floriandejonckheere.be>
5   * Jasper D'haene <jasper.dhaene@gmail.com>
6   *
7   * */
8
9  /**
10   * Om dit probleem wat interessanter en modelleerbaarder
11   * te maken, stellen we dat de te calculeren Ramseygetallen
12   * gewoon het antwoord zijn van het 'party problem':
13   *
14   *  $R(m,n)$  = Hoeveel gasten moeten er uitgenodigd worden
15   * zodat  $\geq m$  elkaar kennen en  $\geq n$  elkaar niet kennen.
16   *
17   * Hint:  $R(3, 3) = 6$  [1]
18   * Hint:  $R(3, 3) = R(3, 3, 2)$ 
19   * Hint:  $R(3) = R(3, 3, 3) = R(3, 3, 3; 3)$ 
20   * Hint:  $R(3, 3, 3) = 17$  [2]
21   *
22   * [1] http://mathworld.wolfram.com/RamseyNumber.html
23   * [2] http://en.wikibooks.org/wiki/Combinatorics/Bounds\_for\_Ramsey\_numbers
24   *
25   * */
26
27 sig Colour {}
28 sig Node {}
29
30 sig Edge {
31     connection: Node -> Node,
32     colour: one Colour
33 }{
34     // No self-referencing
35      $\forall$  node: Node • (node -> node not in connection)
36 }
37
38 // Make sure that 'colouring' is the same as Edge->colour
39 fact {
40     colour =  $\sim$ (Graph.colouring)
41 }
42
43 // Make sure symmetric relations have the same colour
44 fact {
45      $\forall$  e: Edge • some e': Edge • {
46         e.connection =  $\sim$ (e'.connection)  $\wedge$  e.colour = e'.colour
47     }
48 }
49
50 // Force a monochromely-coloured set with X nodes
51 pred Colours [col: Colour, X: Int] {
52     #( $\sim$ (Graph.colouring).col) = X
53 }
54
55 one sig Graph{
56     nodes: set Node,
57
58     // Edges: Node -> Node
59     edges: set Edge,
60     colouring: Colour one -> some Edge
61 }{
62     // All nodes in graph
63      $\forall$  node: Node • node in nodes
64
65     // All edges in graph
66      $\forall$  edge: Edge • edge in edges
67
68     // Edges relationship is symmetrical
69      $\forall$  edge: Edge • some edge': Edge • edge.connection =  $\sim$ (edge'.connection)
70
71     // Every edge only connects 2 points
72      $\forall$  edge: Edge • one edge.connection

```

```

73
74      // Complete graph
75       $\forall n, n' : \text{Node} \bullet \text{some } e : \text{Edge} \bullet n \neq n' \implies n \rightarrow n' \text{ in } e.\text{connection}$ 
76  }
77
78  /**
79   * Run the numbers
80   *
81   * #Edge should be  $\#(N) * \#(N-1)$ 
82   *  $N = 3 \implies E = 6$ 
83   *  $N = 4 \implies E = 12$ 
84   *  $N = 5 \implies E = 20$ 
85   *  $N = 6 \implies E = 30$ 
86   *
87   */
88  assert TwoColours {
89      some c, c': Colour  $\bullet c \neq c'$  and {
90          /**
91           * Colour conditions:
92           * Each line indicates a (disjoint) subset of edges of the same colour.
93           * Please note that you should double the number of cliques
94           * as input variable, because each edge consists of two 'Edge'
95           * objects (due to undirected symmetry of graph).
96           */
97          Colours[c, 6] or Colours[c', 2]
98      }
99  }
100
101  check TwoColours for exactly 3 Colour, exactly 3 Node, 6 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  $A \rightarrow B \rightarrow C \rightarrow A$

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

```

1 module examples/puzzles/hanoi
2
3 /*
4  * Cyclic towers of Hanoi model
5  *
6  * author of hanoi model: Ilya Shlyakhter
7  * modified by Jasper D'haene <jasper.dhaene@gmail.com>
8  */
9
10 open util/ordering[State] as states
11 open util/ordering[Stake] as stakes
12 open util/ordering[Disc] as discs
13
14 sig Stake { }
15
16 sig Disc { }
17
18 /**
19  * sig State: the complete state of the system --
20  * which disc is on which stake. An solution is a
21  * sequence of states.
22  */
23 sig State {
24   on: Disc -> one Stake // _each_ disc is on _exactly one_ stake
25   // note that we simply record the set of discs on each stake --
26   // the implicit assumption is that on each stake the discs
27   // on that stake are ordered by size with smVest disc on top
28   // and largest on bottom, as the problem requires.
29 }
30
31 /**
32  * compute the set of discs on the given stake in this state.
33  * ~(this.on) map the stake to the set of discs on that stake.
34  */
35 fun discsOnStake[st: State, stake: Stake]: set Disc {
36   stake.~(st.on)
37 }
38
39 /**
40  * compute the top disc on the given stake, or the empty set
41  * if the stake is empty
42  */
43 fun topDisc[st: State, stake: Stake]: lone Disc {
44   { d: st.discsOnStake[stake] • st.discsOnStake[stake] in discs/nexts[d] + d }
45 }
46
47 /**
48  * Describes the operation of moving the top disc from stake fromStake
49  * to stake toStake. This function is defined implicitly but is
50  * nevertheless deterministic, i.e. the result state is completely
51  * determined by the initial state and fromStake and toStake; hence
52  * the "det" modifier above. (It's important to use the "det" modifier
53  * to tell the Alloy Analyzer that the function is in fact deterministic.)
54  */
55 pred Move [st: State, fromStake, toStake: Stake, s': State] {
56   //modified: Cyclic move property

```

```

57   fromStake ≠ stakes/last ⇒
58     fromStake.next = toStake
59   else
60     toStake = stakes/first
61
62   let d = st.topDisc[fromStake] • {
63     // ∀ discs on toStake must be larger than d,
64     // so that we can put d on top of them
65     st.discsOnStake[toStake] in discs/nexts[d]
66     // after, the fromStake has the discs it had before, minus d
67     s'.discsOnStake[fromStake] = st.discsOnStake[fromStake] - d
68     // after, the toStake has the discs it had before, plus d
69     s'.discsOnStake[toStake] = st.discsOnStake[toStake] + d
70     // the remaining stake afterwards has exactly the discs it had before
71     let otherStake = Stake - fromStake - toStake •
72     s'.discsOnStake[otherStake] = st.discsOnStake[otherStake]
73   }
74 }
75
76 /**
77  * there is a leftStake that has ∀ the discs at the beginning,
78  * and a rightStake that has ∀ the discs at the end
79  */
80 pred Game1 {
81   Disc in states/first.discsOnStake[stakes/first]
82   //modified end condition
83   some finalState: State • Disc in finalState.discsOnStake[stakes/last] ∧ (finalState = states/last)
84
85   // each adjacent pair of states are related by a valid move of one disc
86   ∀ preState: State - states/last •
87     let postState = states/next[preState] •
88       some fromStake: Stake • {
89         // must have at least one disk on fromStake to be able to move
90         // a disc from fromStake to toStake
91         some preState.discsOnStake[fromStake]
92         // post- results from pre- by making one disc move
93         some toStake: Stake • preState.Move[fromStake, toStake, postState]
94       }
95 }
96
97 /**
98  * there is a leftStake that has ∀ the discs at the beginning,
99  * and a rightStake that has ∀ the discs at the end
100  */
101 pred Game2 {
102   Disc in states/first.discsOnStake[stakes/first]
103   some finalState: State • Disc in finalState.discsOnStake[stakes/last]
104
105   // each adjacent pair of states are related by a valid move of one disc
106   ∀ preState: State - states/last •
107     let postState = states/next[preState] •
108       some fromStake: Stake •
109         let d = preState.topDisc[fromStake] • {
110           // must have at least one disk on fromStake to be able to move
111           // a disc from fromStake to toStake
112           some preState.discsOnStake[fromStake]
113           postState.discsOnStake[fromStake] = preState.discsOnStake[fromStake] - d
114           some toStake: Stake • {
115             // post- results from pre- by making one disc move
116             preState.discsOnStake[toStake] in discs/nexts[d]
117             postState.discsOnStake[toStake] = preState.discsOnStake[toStake] + d
118             // the remaining stake afterwards has exactly the discs it had before
119             let otherStake = Stake - fromStake - toStake •
120             postState.discsOnStake[otherStake] = preState.discsOnStake[otherStake]
121           }
122         }
123     }
124 }
125
126 // #state = 22 + 3X voor geldige instantie.
127 run Game1 for 1 but 3 Stake, 3 Disc, 22 State expect 1
127 run Game2 for 1 but 3 Stake, 3 Disc, 8 State expect 1

```