1. Forward-checking repeatedly applies the constraint that no two adjacent nodes can have the same colour. E.g., 2 can no longer be R or C so it must be B. 3 can no longer be R or C so it must be B. 6 can then no longer be B or R so it must be C. Now, 7 is adjacent to each of R, B, and C, so the constraints have been violated and backtracking is required.

Yes, the AC-3 algorithm will detect the problem earlier.

**def** remove\_inconsistencies(D\_i, D\_j):

result = False

**for** idx, d **in** enumerate(D\_i):

**if** **not** any(d\_ != d **for** d\_ **in** D\_j):

**del** D\_i[idx]

result = True

**return** result

**def** ac3(arcs, options):

queue = arcs

**while** len(queue):

i, j = queue[0]

queue = queue[1:]

**if** remove\_inconsistencies(options[i], options[j]):

**for** (i\_, k) **in** arcs:

**if** i\_ != i **or** k == i **or** k == j:

**continue**

queue.append((k, i))

arcs = [

(1,2),

(1,3),

(1,4),

(2,4),

(2,6),

(3,4),

(3,7),

(4,5),

(5,6),

(5,7),

(6,7),

(6,8),

(7,8)

]

arcs += list((j, i) **for** (i, j) **in** arcs)

options = {

1: ["R"],

2: ["R", "B", "C"],

3: ["R", "B", "C"],

4: ["C"],

5: ["R"],

6: ["B"],

7: ["R", "B", "C"],

8: ["B"],

}

ac3(arcs, options)

**print**(options)

*# {1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: []}*

1. The Goal clause could only be true if and only if its input (actionList) results in the goal being reached. In the process of unifying this clause, if Goal is satisfiable, then actionList will be instantiated to a value which satisfies Goal. If required, prolog-style backtracking could be used to generate all such possible values of actionList.