Artificial Intelligence

CSP: Backtracking and Inference

Jendrik Seipp

Linköping University

based on slides by Thomas Keller and Malte Helmert (University of Basel)

CSP Algorithms

we now consider algorithms for solving CSPs

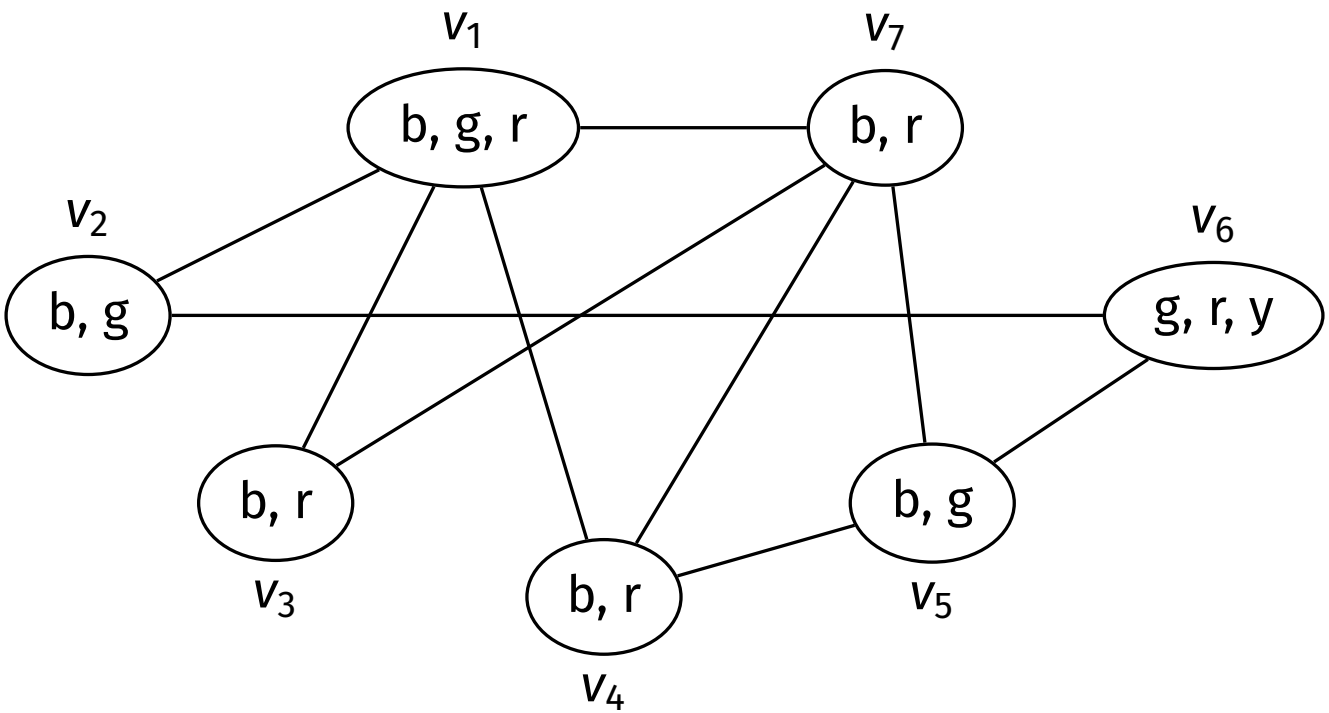
basic concepts:

- search: check partial assignments systematically
- backtracking: discard inconsistent partial assignments
- inference: derive equivalent, but tighter constraints to reduce the size of the search space

Backtracking Without Inference (= Naive Backtracking)

Naive Backtracking: Example

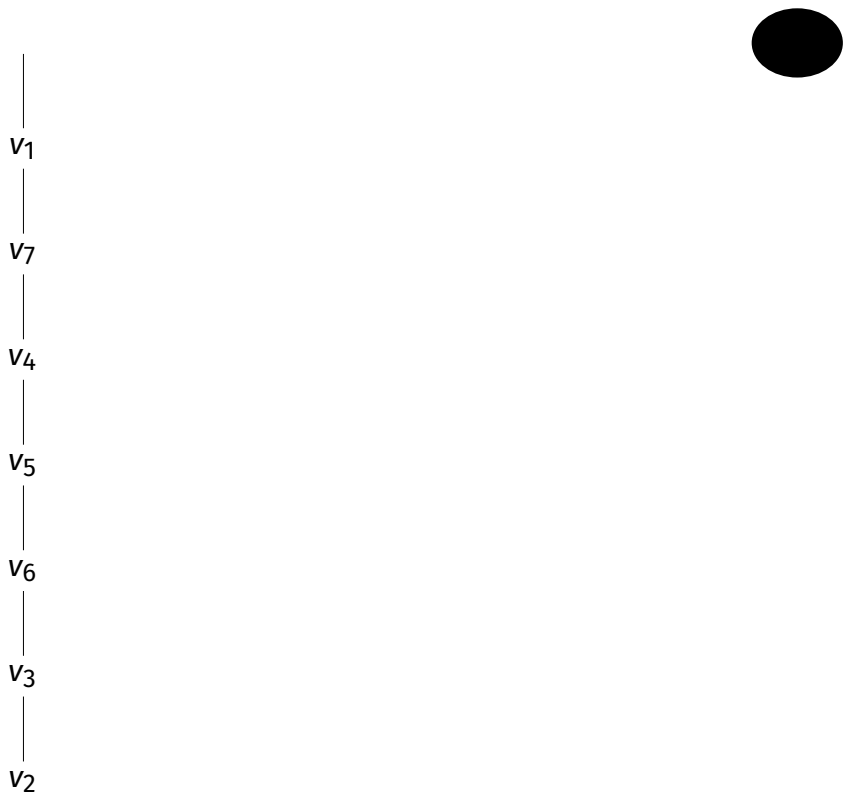
Consider the CSP for the following graph coloring instance:



Naive Backtracking: Example

search tree for naive backtracking with

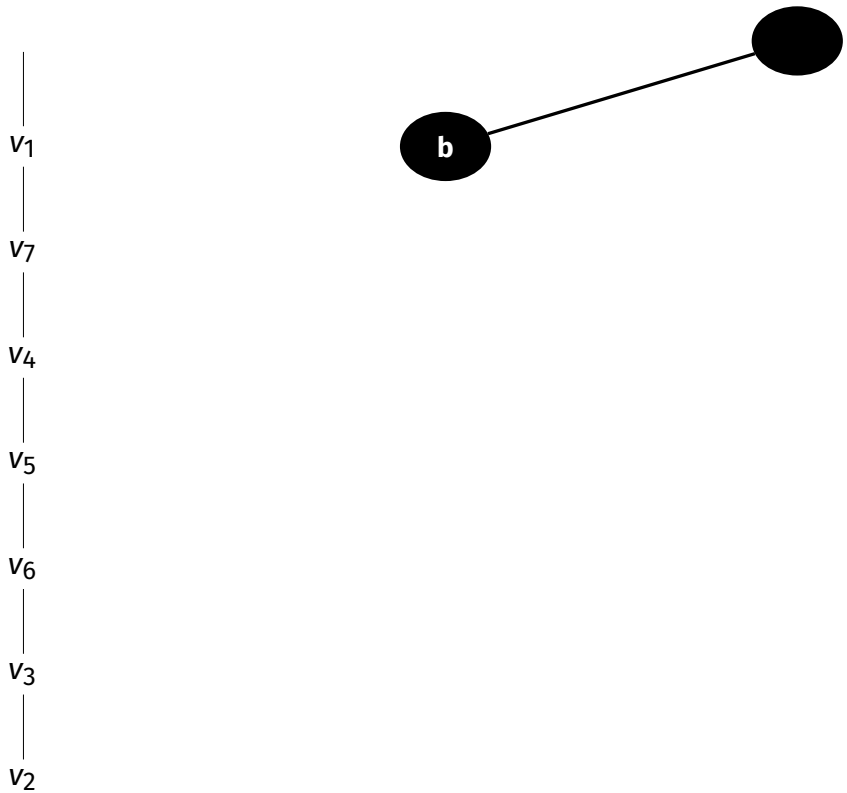
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

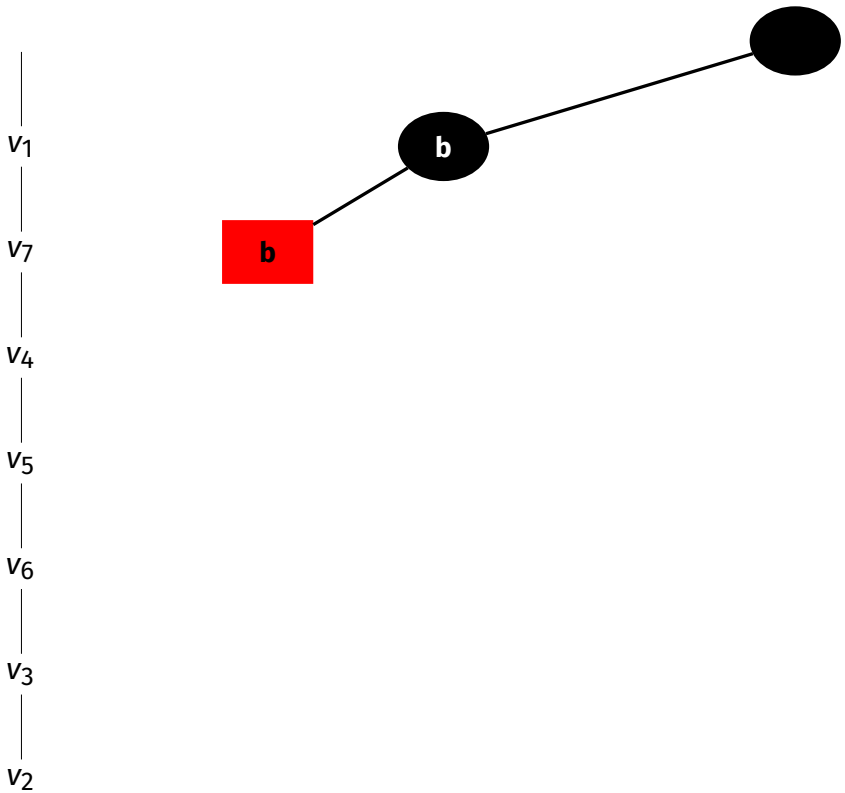
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

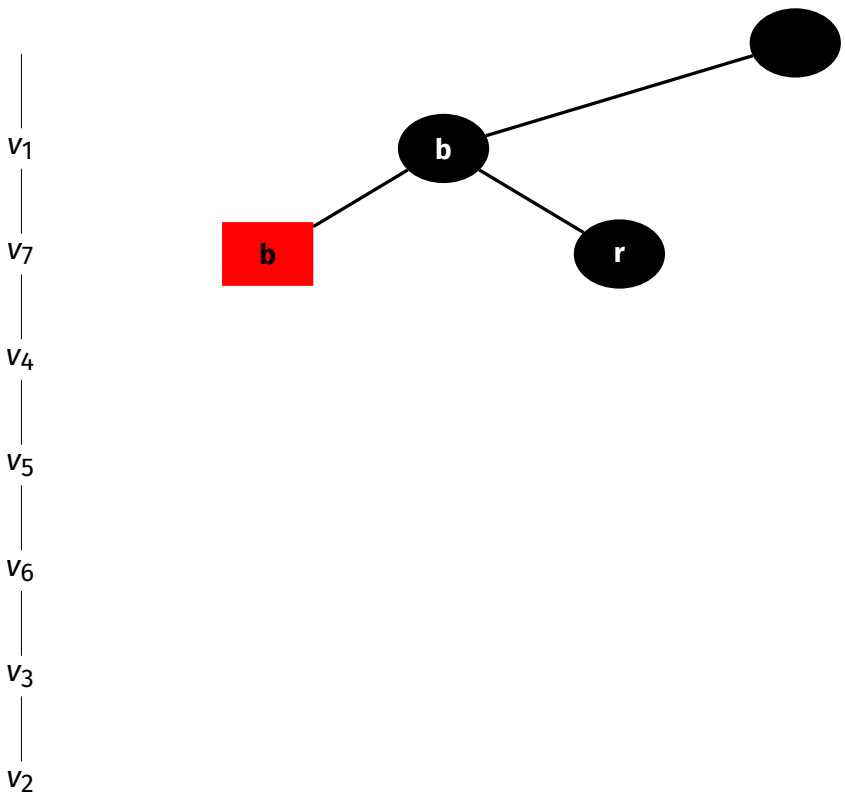
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

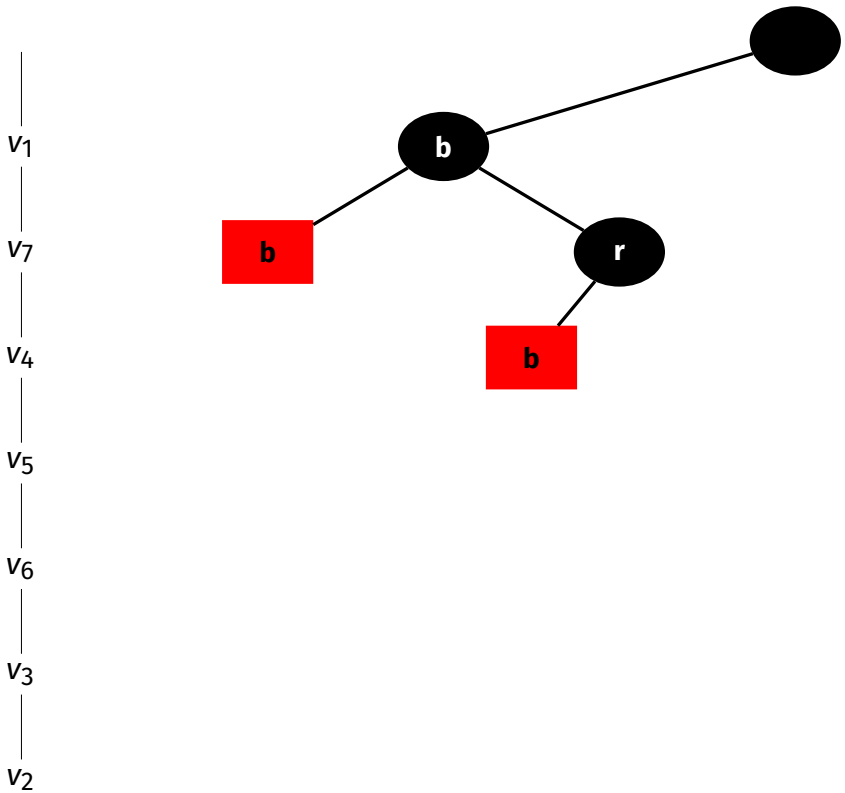
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

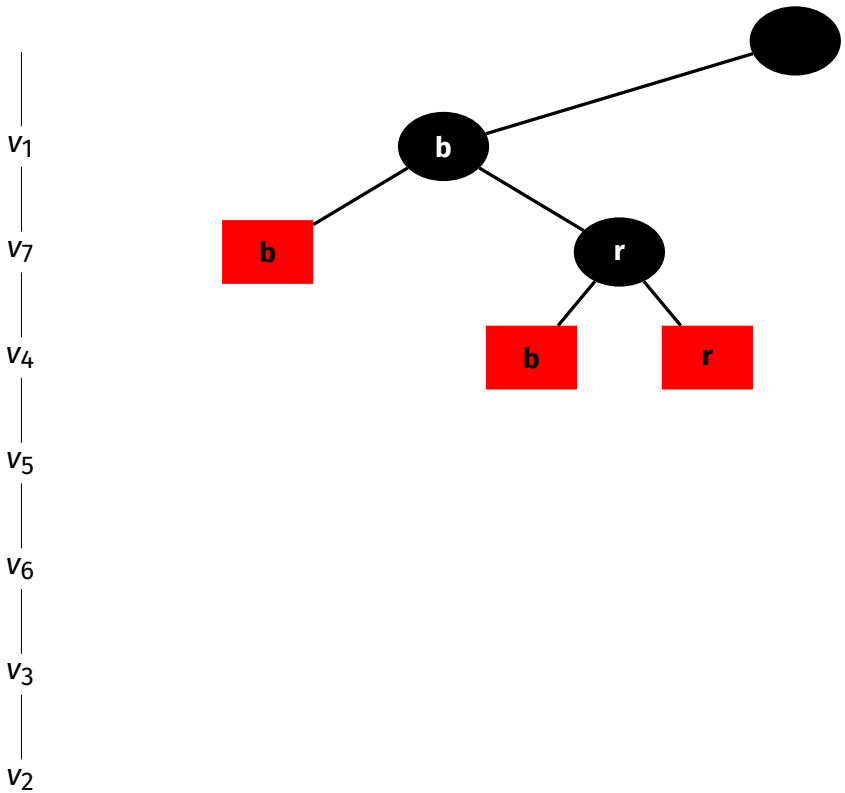
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

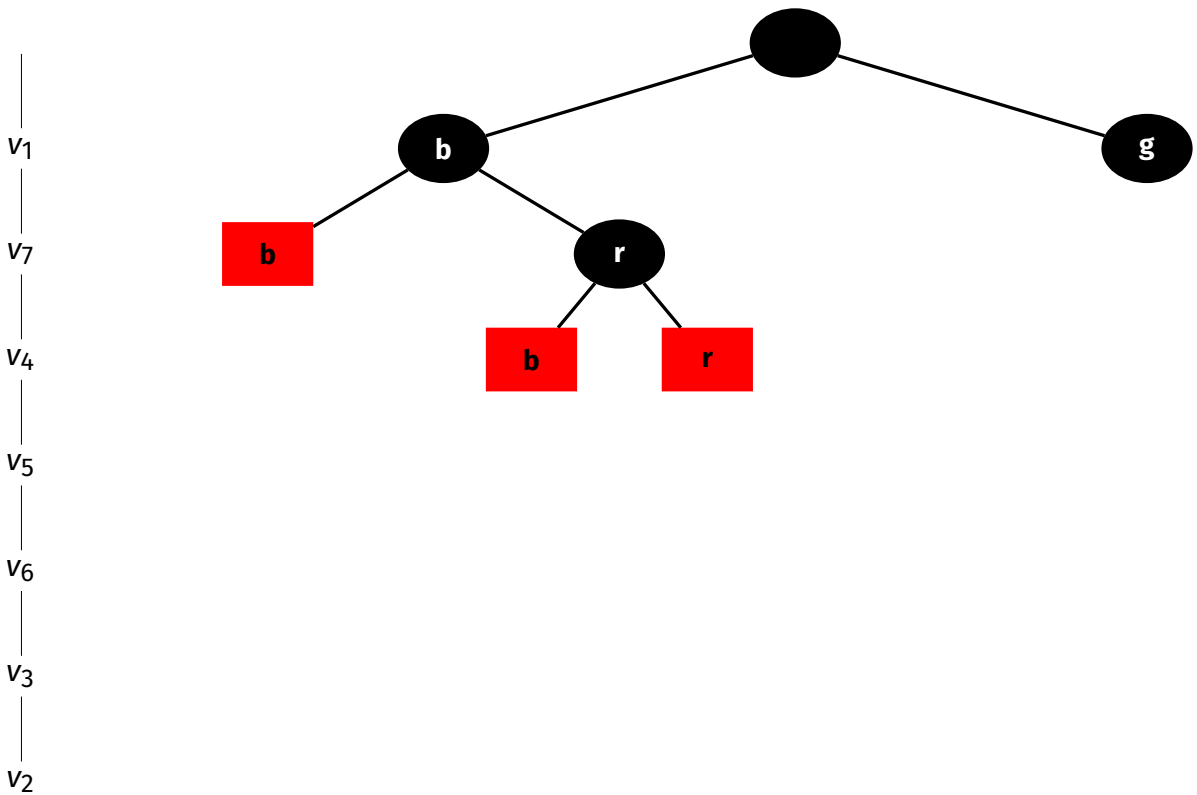
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

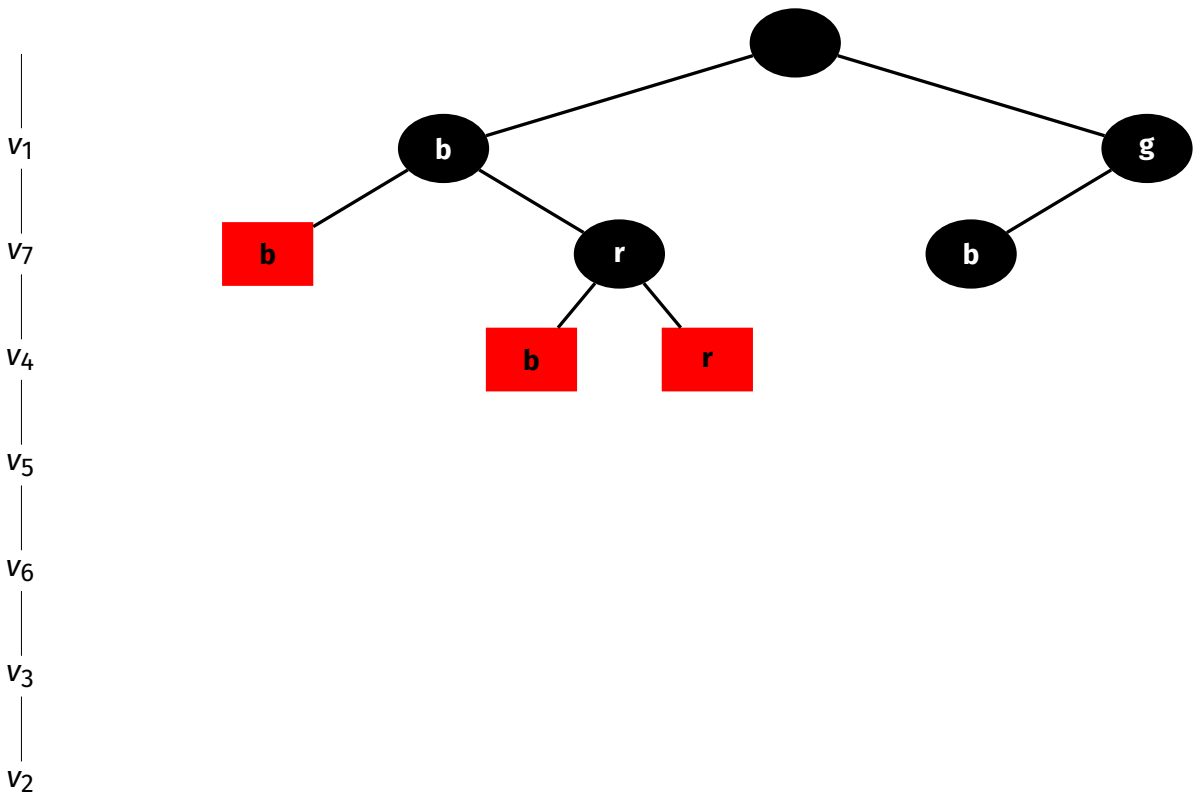
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

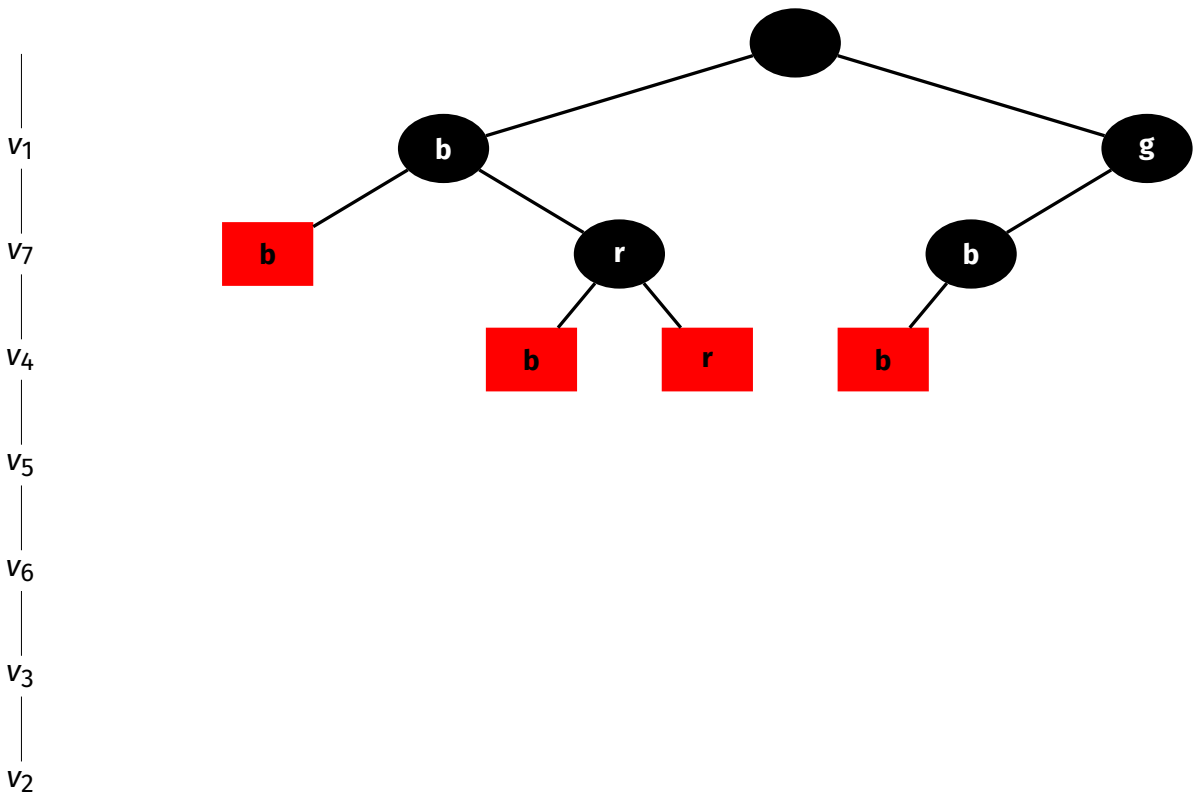
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

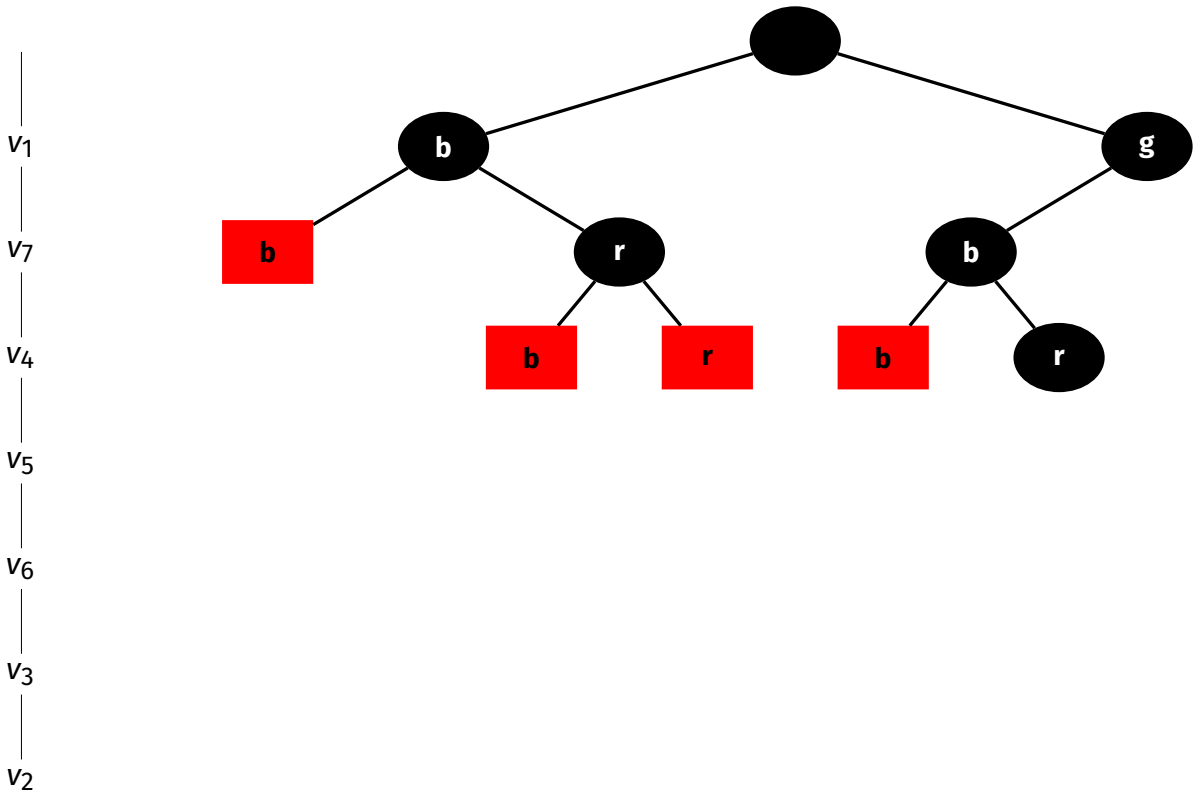
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

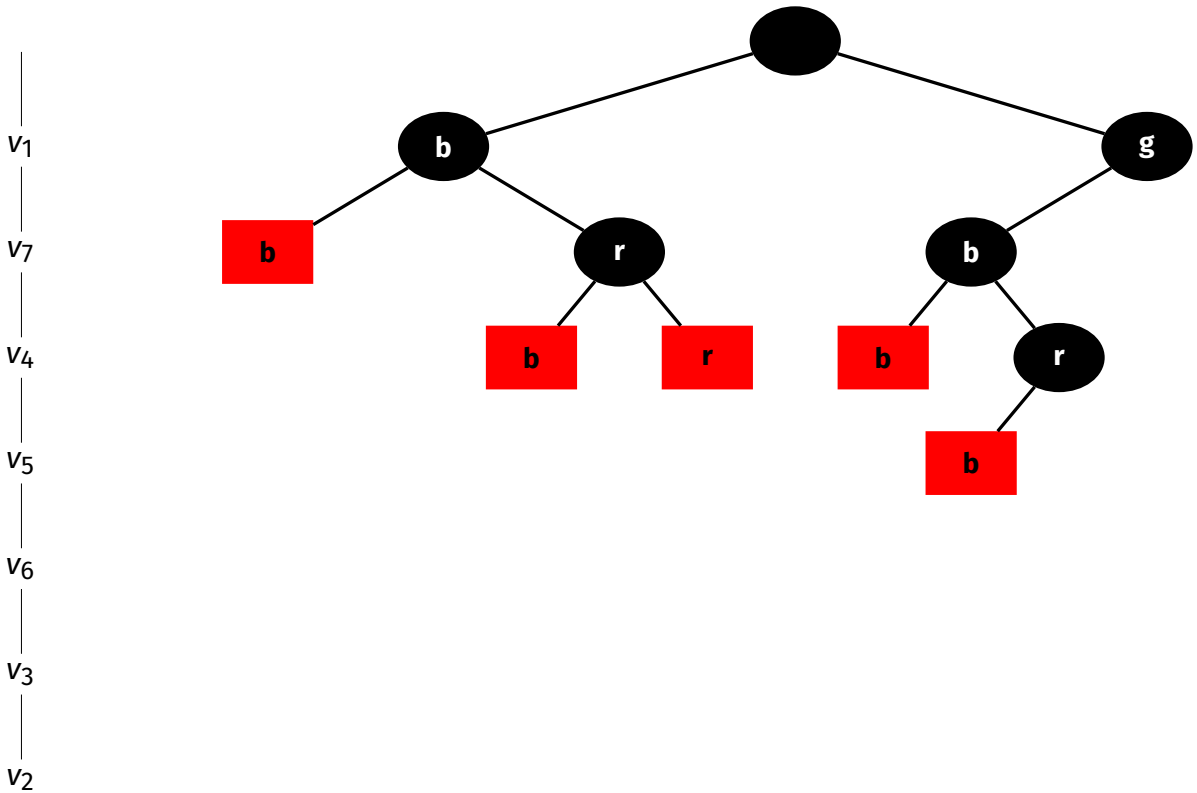
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

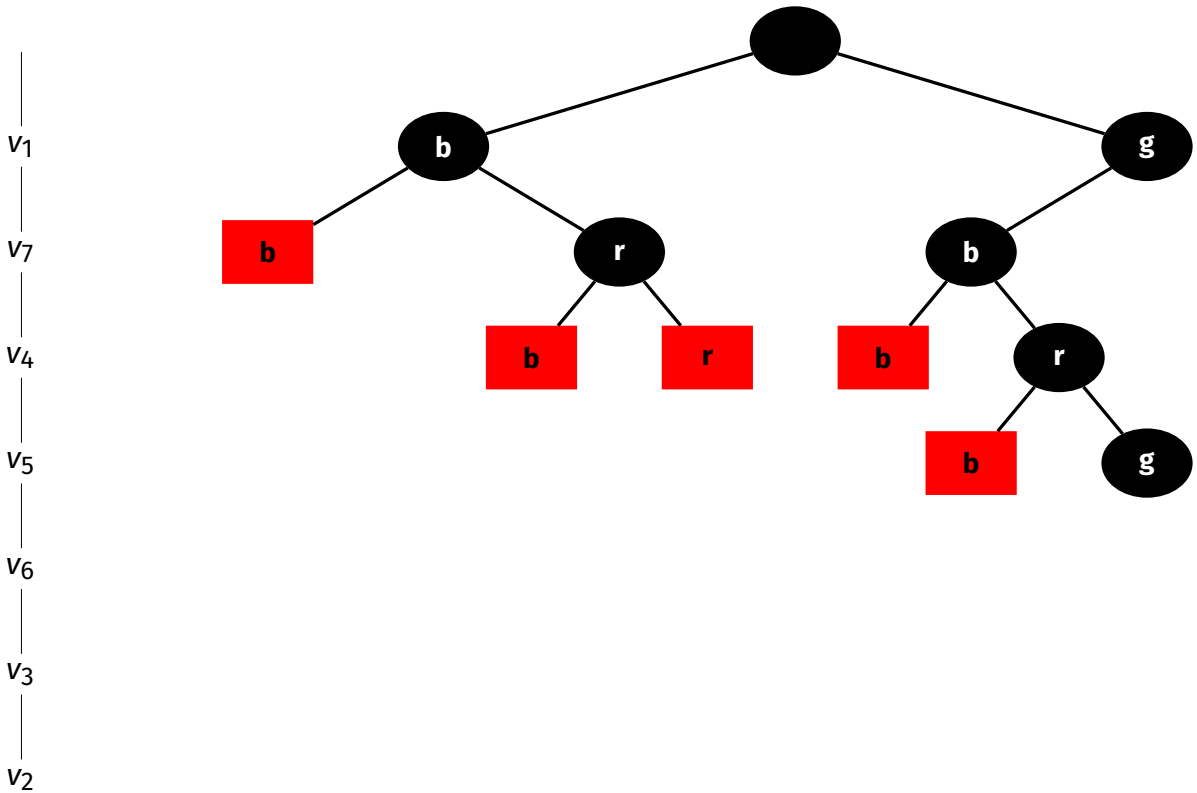
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

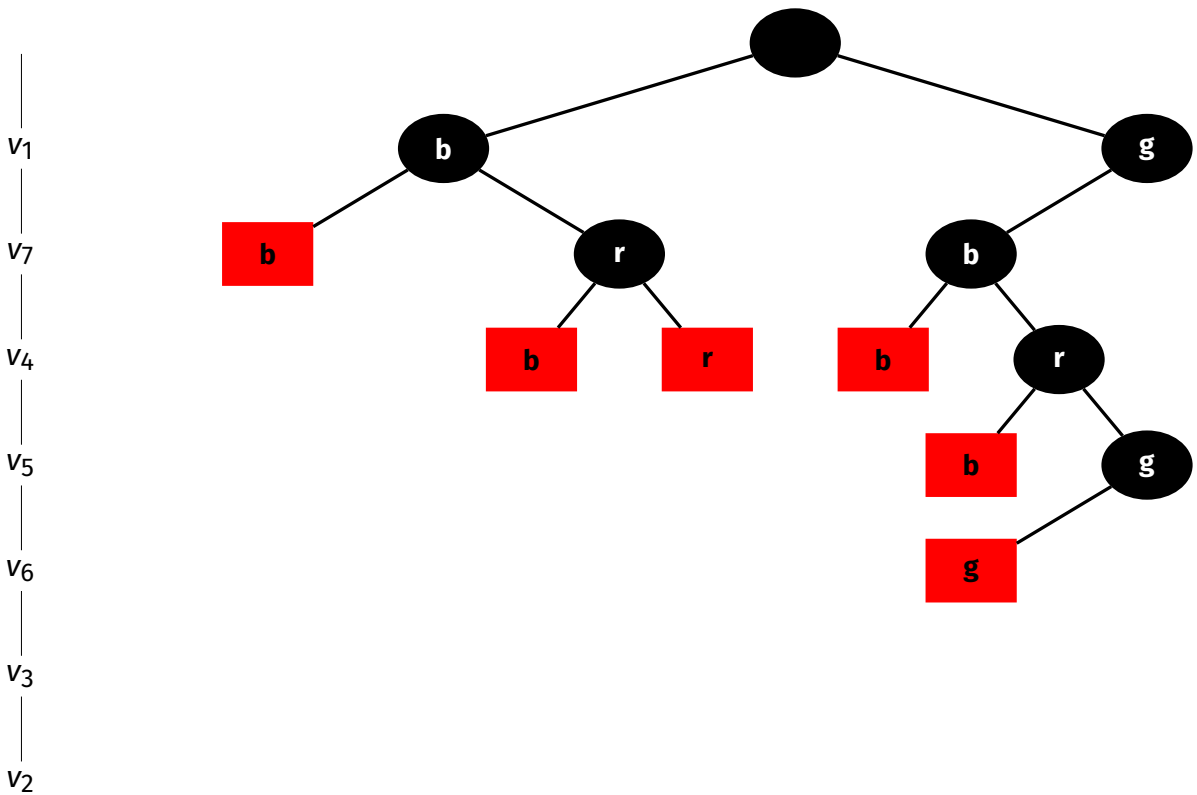
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

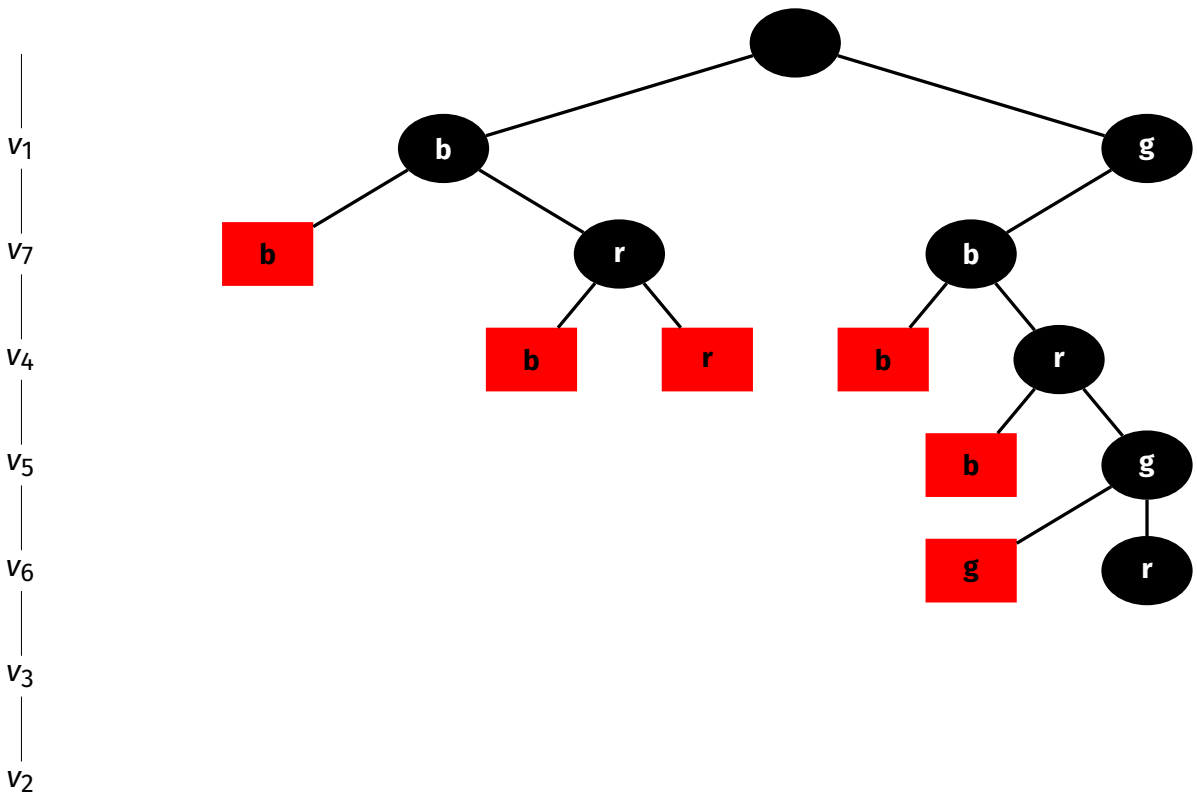
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

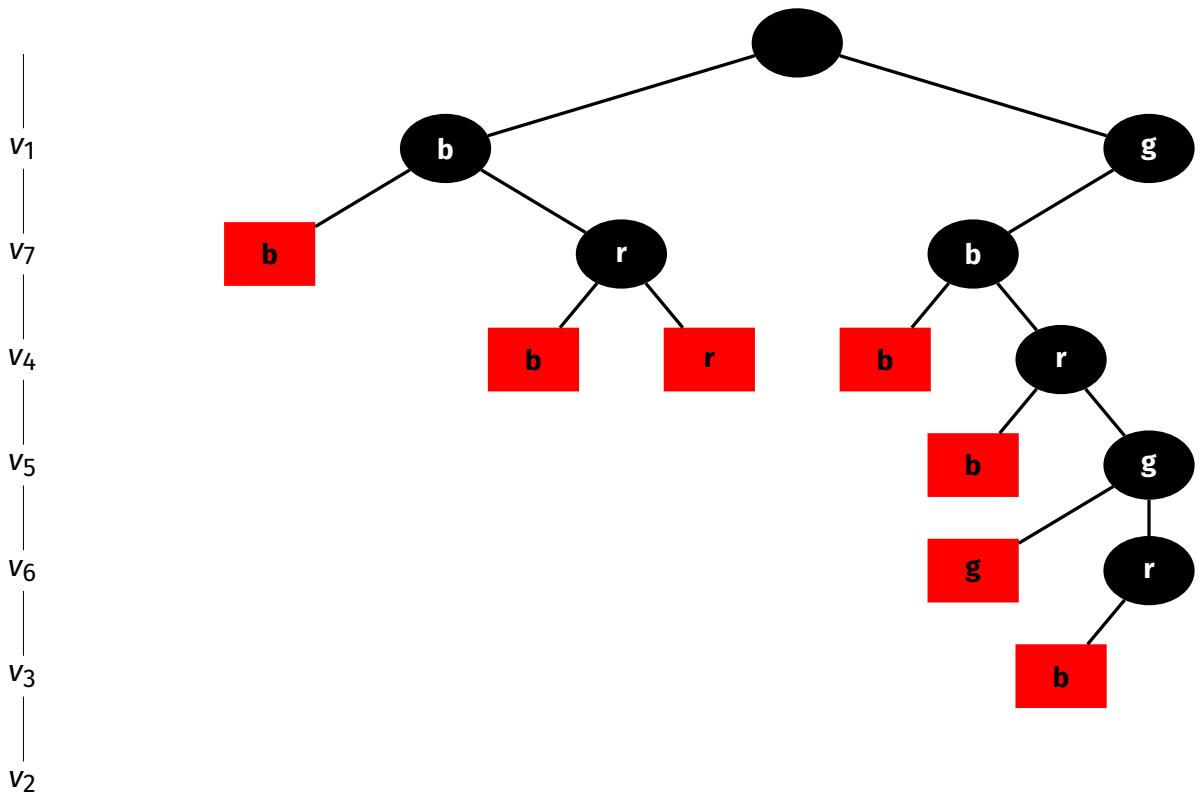
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

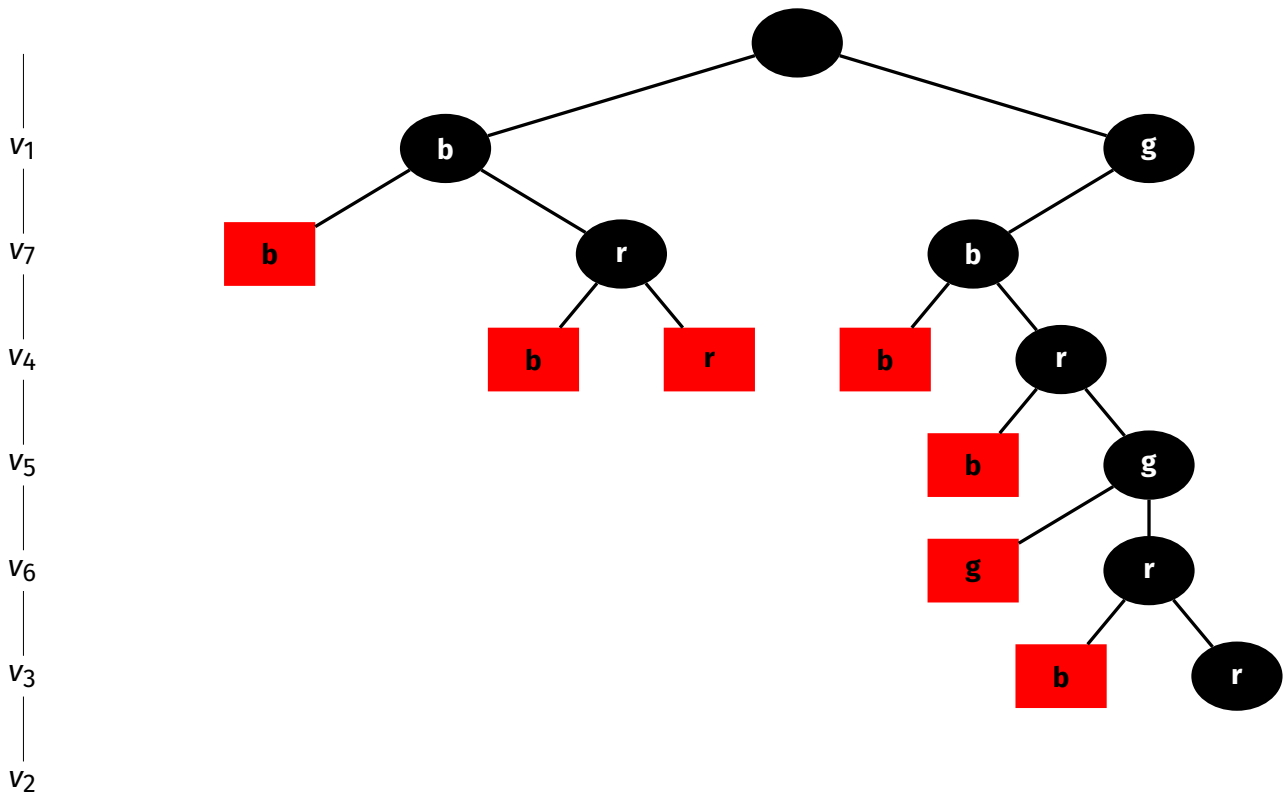
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

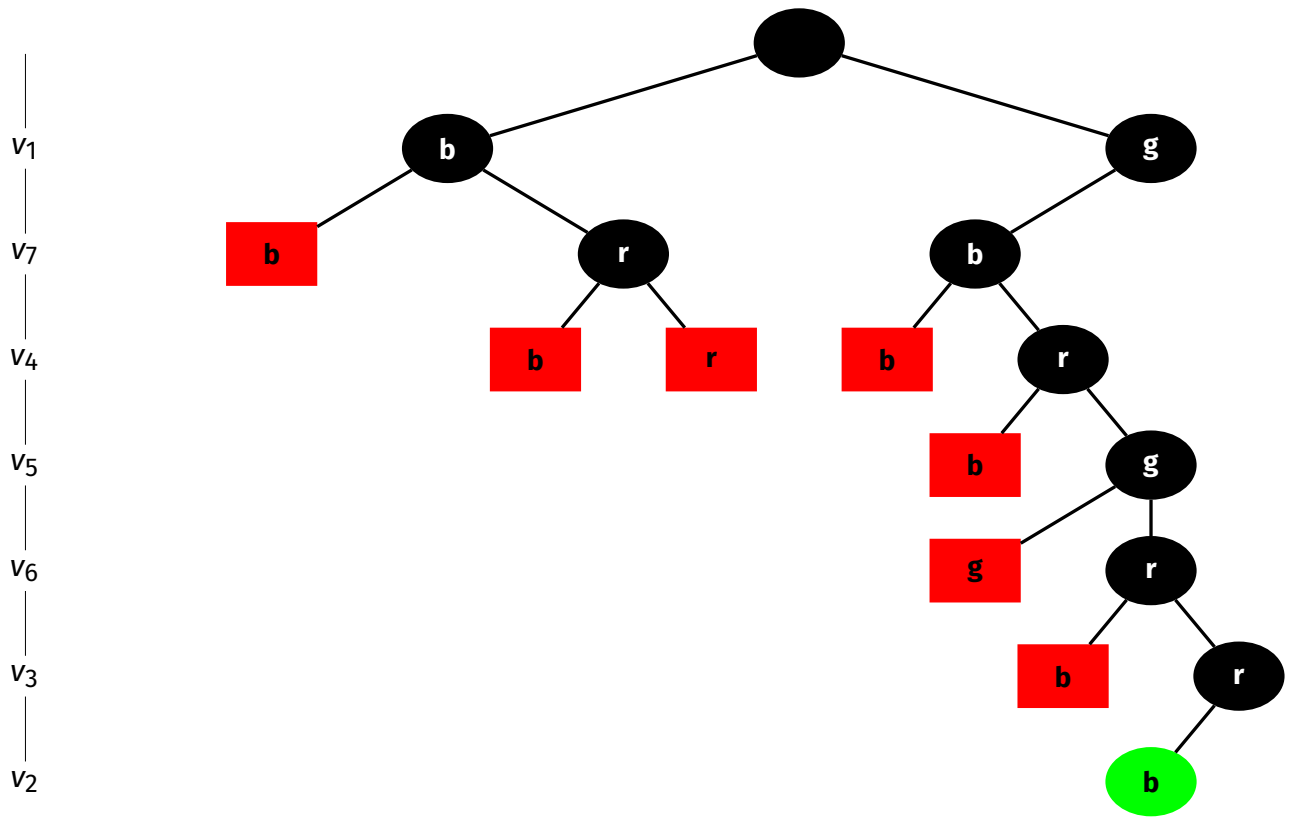
- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Naive Backtracking: Example

search tree for naive backtracking with

- fixed variable order $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetical order of the values



Is This a New Algorithm?

we have already seen this algorithm:
Backtracking corresponds to depth-first search
with the following state space:

- states: partial assignments
- initial state: empty assignment $\varnothing$
- goal states: consistent total assignments
- actions: $\texttt{assign}_{v,d}$ assigns value $d \in dom(v)$ to variable v
- action costs: all 0 (all solutions are of equal quality)
- transitions:
  - for each non-total consistent assignment α,
    choose variable $v =$ SELECT-UNASSIGNED-VARIABLE
  - transition $\alpha \xrightarrow{\texttt{assign}_{v,d}} \alpha \cup \{v \mapsto d\}$ for each $d \in dom(v)$

Why Depth-First Search?

depth-first search is particularly well-suited for CSPs:

- path length bounded (by the number of variables)
- solutions located at the same depth (lowest search layer)
- state space is directed tree, initial state is the root
  $\rightsquigarrow$ no duplicates

hence none of the problematic cases for depth-first search occurs

Naive Backtracking: Discussion

- naive backtracking often has to exhaustively explore similar search paths (i.e., partial assignments that are identical except for a few variables)
- "critical" variables are not recognized and hence considered for assignment (too) late
- decisions that necessarily lead to constraint violations are only recognized when all variables involved in the constraint have been assigned.

$\rightsquigarrow$ more intelligence by focusing on critical decisions and by inference of consequences of previous decisions

Variable and Value Orders

Variable Orders

- SELECT-UNASSIGNED-VARIABLE method in backtracking search allows to influence order in which variables are considered for assignment
- selected order can strongly influence the search space size and hence the search performance
- general aim: make critical decisions as early as possible

Variable Orders

two common variable ordering criteria:

- minimum remaining values:
  prefer variables that have small domains
  - intuition: few subtrees $\rightsquigarrow$ smaller tree
  - extreme case: only one value $\rightsquigarrow$ forced assignment
- most constraining variable:
  prefer variables contained in many nontrivial constraints
  - intuition: constraints tested early
    $\rightsquigarrow$ inconsistencies recognized early $\rightsquigarrow$ smaller tree

combination: use minimum remaining values criterion,
then most constraining variable criterion to break ties

Value Orders

- ORDER-DOMAIN-VALUES method in backtracking search allows to influence order in which values of the selected variable v are considered
- this is less important because it does not matter in subtrees without a solution
- in subtrees with a solution, ideally a value that leads to a solution should be chosen
- general aim: make most promising assignments first

Value Orders

Definition (conflict)

Let $C = \langle V, dom, C \rangle$ be a CSP.
For variables $v \neq v'$ and values $d \in dom(v)$, $d' \in dom(v')$,
the assignment $v \mapsto d$ is in conflict with $v' \mapsto d'$ if there is $c_{v,v'} \in C$ s.t. $(d, d') \notin \text{rel}(c_{v,v'})$.

value ordering criterion for partial assignment α
and selected variable v:

- minimum conflicts: prefer values $d \in dom(v)$ such that $v \mapsto d$ causes as few conflicts as possible with variables that are unassigned in α

Inference

Inference

Inference

Derive additional constraints
that are implied by the given constraints,
i.e., that are satisfied in all solutions.

example: CSP with variables v_1, v_2, v_3 with domain $\{1, 2, 3\}$
and constraints $v_1 < v_2$ and $v_2 < v_3$.

we can infer:

- v_2 cannot be equal to 3
- $\langle (v1, v2), \{(1, 2), (1, 3), (2, 3)\}$ can be tightened to $\langle (v1, v2), \{(1, 2)\}$
  (tighter binary constraint)
- $v_1 < v_3$
  ("new" binary constraint = trivial constraint tightened)

Trade-Off Search vs. Inference

Inference formally

Replace a given CSP C with an equivalent, but tighter CSP.

trade-off:

- the more complex the inference, and
- the more often inference is applied,
- the smaller the resulting state space, but
- the higher the complexity per search node.

When to Apply Inference?

different possibilities to apply inference:

- once as preprocessing before search
- combined with search: before recursive calls during backtracking procedure
  - already assigned variable $v \mapsto d$ corresponds to $dom(v) = \{d\} \rightsquigarrow$ more inferences possible
  - during backtracking, derived constraints have to be retracted because they were based on the given assignment

$\rightsquigarrow$ powerful, but possibly expensive

Backtracking with Inference: Discussion

- INFERENCE method in backtracking search allows to apply different inference methods
- inference methods can recognize unsolvability (given α)
- efficient implementations of inference are often incremental: the last assigned variable/value pair $v \mapsto d$ is taken into account to speed up the inference computation

Arc Consistency

Arc Consistency: Definition

Definition (Arc Consistent)

Let $C = \langle V, dom, C \rangle$ be a CSP.

(a) A variable $v \in V$ is arc consistent
with respect to another variable $v' \in V$,
if for every value $d \in dom(v)$
there exists a value $d' \in dom(v')$ with $\langle d, d' \rangle \in c_{v,v'}$.

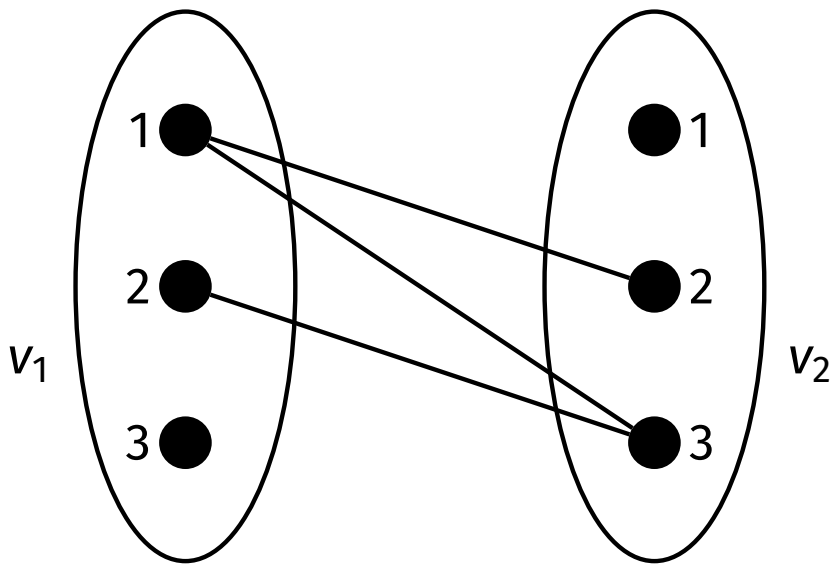
(b) The CSP C is arc consistent,
if every variable $v \in V$ is arc consistent
with respect to every other variable $v' \in V$.

remarks:

- definition for variable pair is not symmetrical
- v always arc consistent with respect to v'
  if the constraint between v and v' is trivial

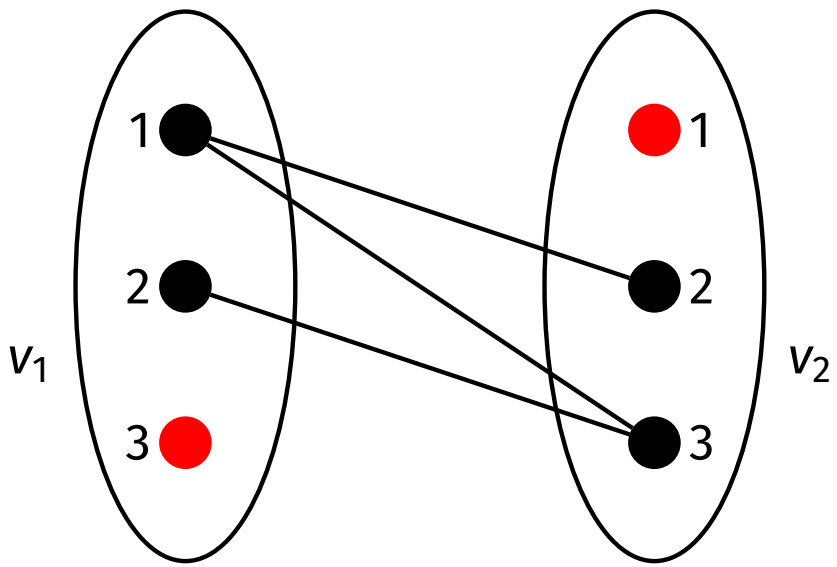
Arc Consistency: Example

Consider a CSP with variables v_1 and v_2,
domains $dom(v_1) = dom(v_2) = \{1, 2, 3\}$
and the constraint expressed by $v_1 < v_2$.



Arc Consistency: Example

Consider a CSP with variables v_1 and v_2,
domains $dom(v_1) = dom(v_2) = \{1, 2, 3\}$
and the constraint expressed by $v_1 < v_2$.



Arc consistency of v_1 with respect to v_2
and of v_2 with respect to v_1 are violated.

Enforcing Arc Consistency

- enforcing arc consistency, i.e., removing values from $dom(v)$ that violate the arc consistency of v with respect to v', is a correct inference method

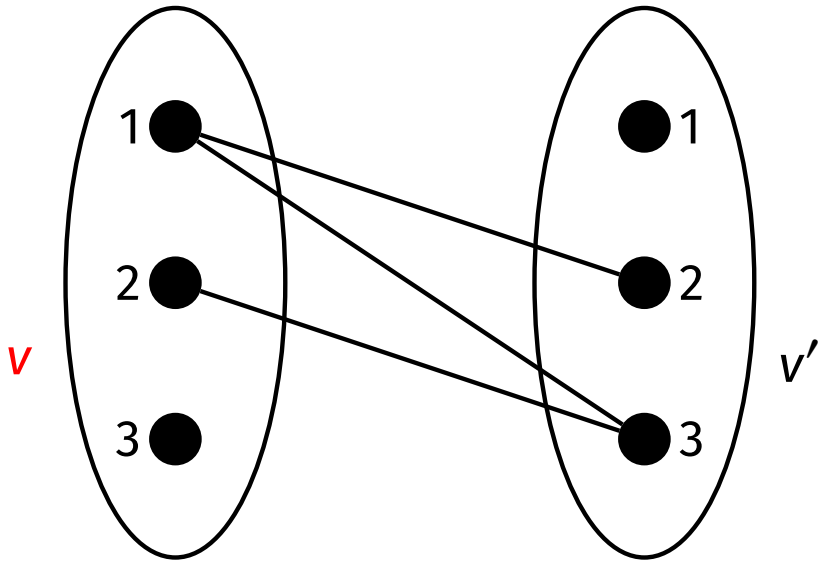
Processing Variable Pairs: Revise

function Revise($\langle V, dom, C\rangle, v, v'$):
    revised = **false**
    let $c = \langle (v, v'), \text{rel}\rangle \in C$
    for each $d \in dom(v)$:
        if there is no $d' \in dom(v')$ s.t. $(d, d') \in \text{rel}(c)$:
            remove d from $dom(v)$
            revised = **true**
    return *revised*

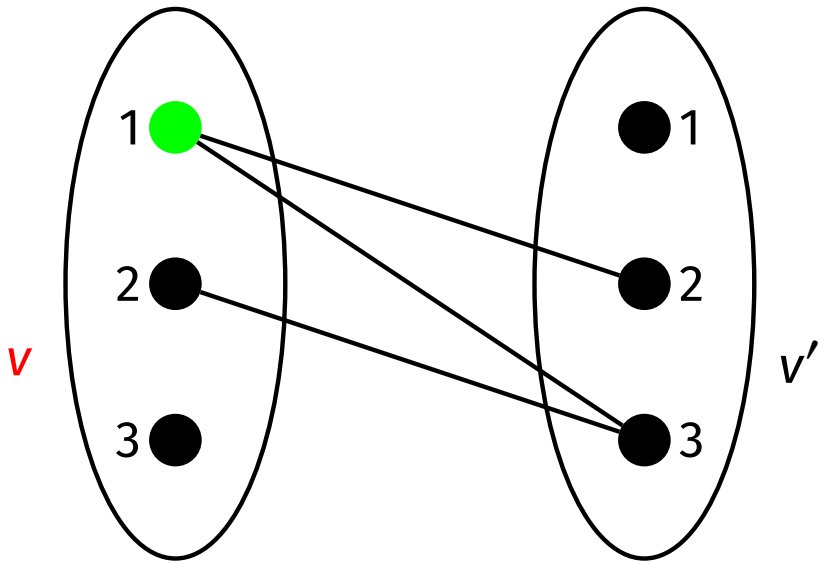
effect: v arc consistent with respect to v'.
All violating values in $dom(v)$ are removed.

time complexity: $O(k^2)$, where k is maximal domain size

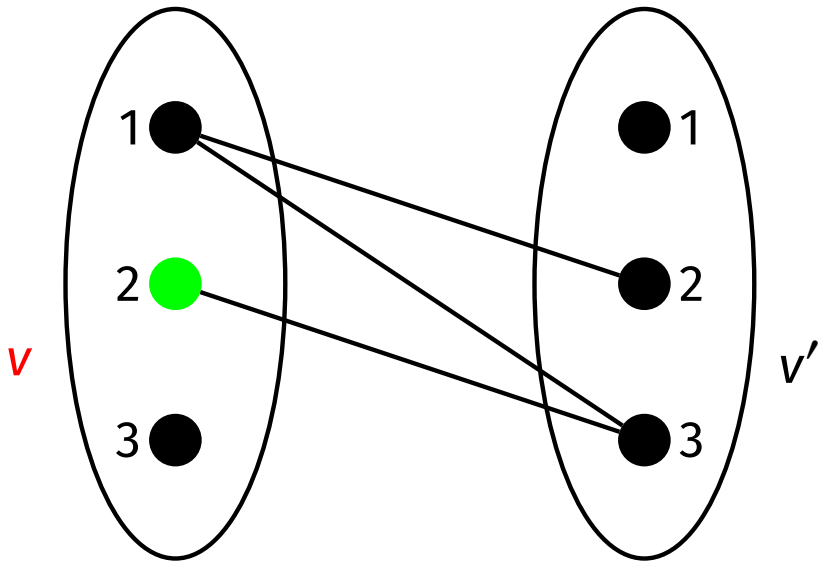
Example: revise



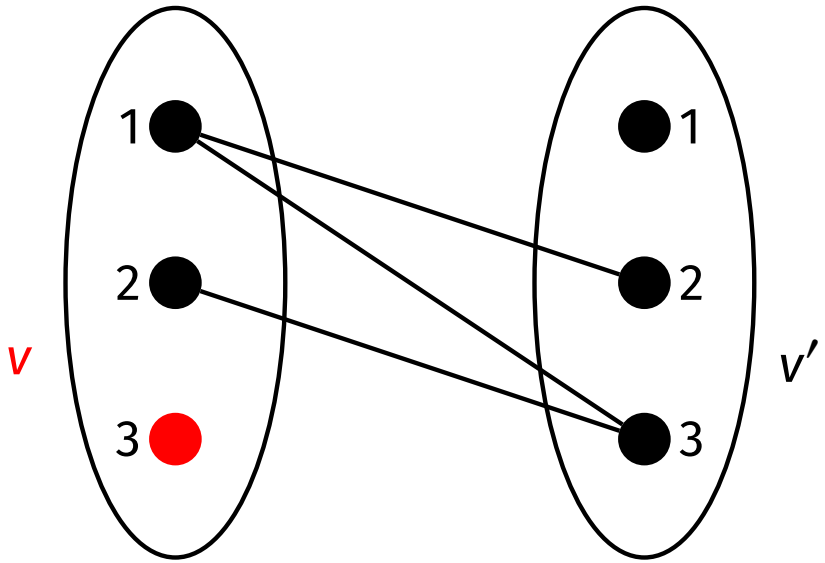
Example: revise



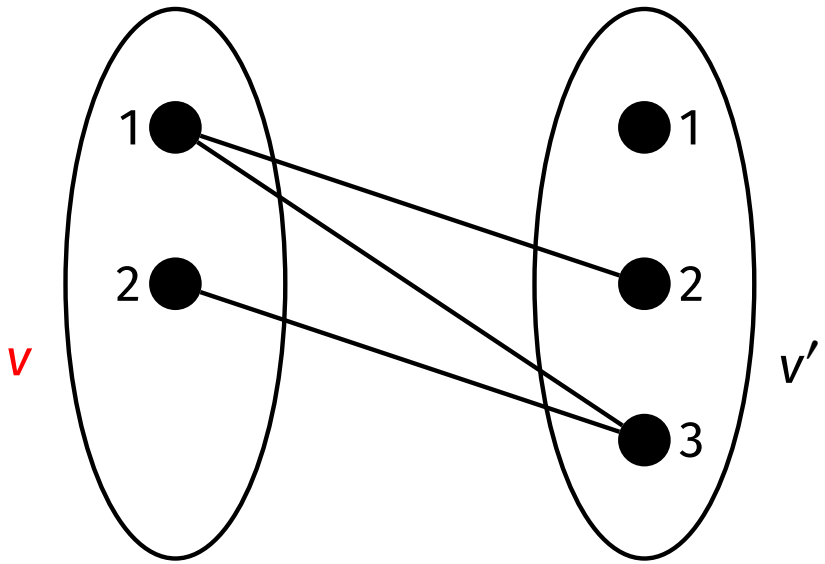
Example: revise



Example: revise



Example: revise



Enforcing Arc Consistency: AC-3

idea:

- transform C into equivalent arc consistent CSP
- store potentially inconsistent variable pairs in a queue

function AC-3(C):

$\langle V, dom, C \rangle := C$

$queue := \emptyset$

for each nontrivial constraint $c_{u,v}$:
    insert $\langle u, v \rangle$ into $queue$
    insert $\langle v, u \rangle$ into $queue$

while $queue \neq \emptyset$:
    remove an arbitrary element $\langle u, v \rangle$ from $queue$
    if Revise(C, u, v):
        for each $w \in V \setminus \{u, v\}$ where $c_{w,u}$ is nontrivial:
            insert $\langle w, u \rangle$ into $queue$

Path Consistency

Path Consistency

idea of arc consistency:

- for every assignment to a variable u
  there must be a suitable assignment to every other variable v
- If not: remove values of u for which
  no suitable "partner" assignment to v exists

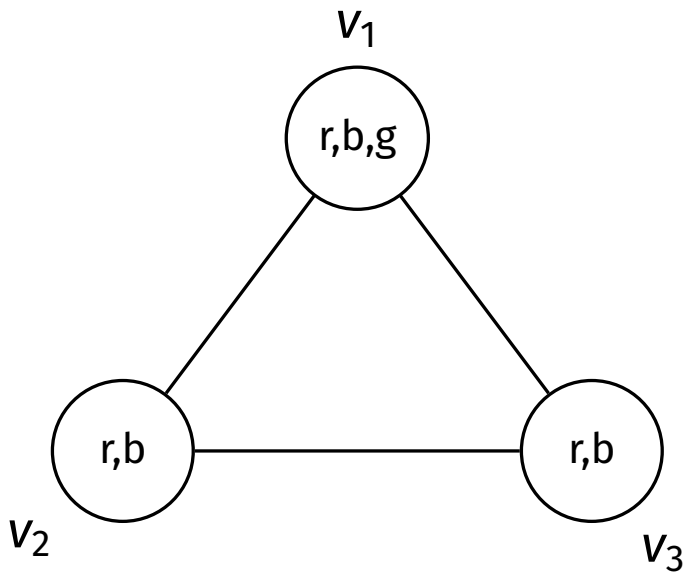
this idea can be extended to three variables (path consistency):

- for every joint assignment to variables u, v
  there must be a suitable assignment to every third variable w
- if not: remove pairs of values of u and v for which
  no suitable "partner" assignment to w exists.

$\rightsquigarrow$ tighter binary constraint on u and v

Path Consistency: Example

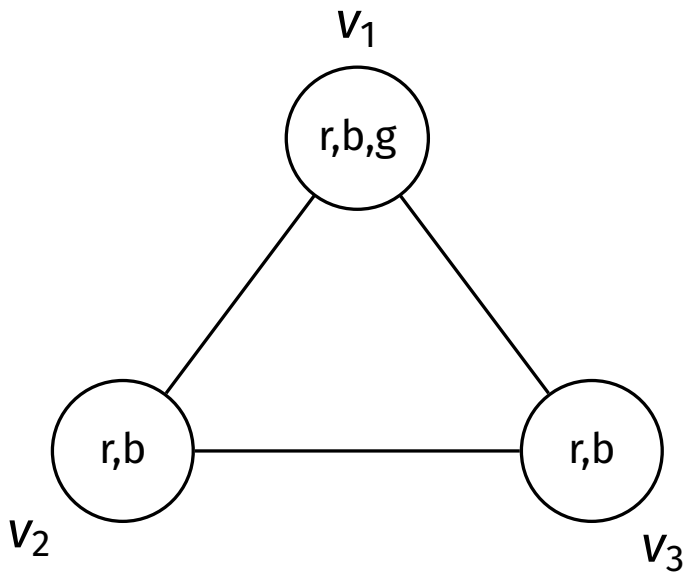
arc consistent, but not path consistent



$$c_{12} = \langle (v_1, v_2), \{(r, b), (b, r), (g, r), (g, b)\}$$
$$c_{13} = \langle (v_1, v_3), \{(r, b), (b, r), (g, r), (g, b)\}$$
$$c_{23} = \langle (v_2, v_3), \{(r, b), (b, r)\}$$

Path Consistency: Example

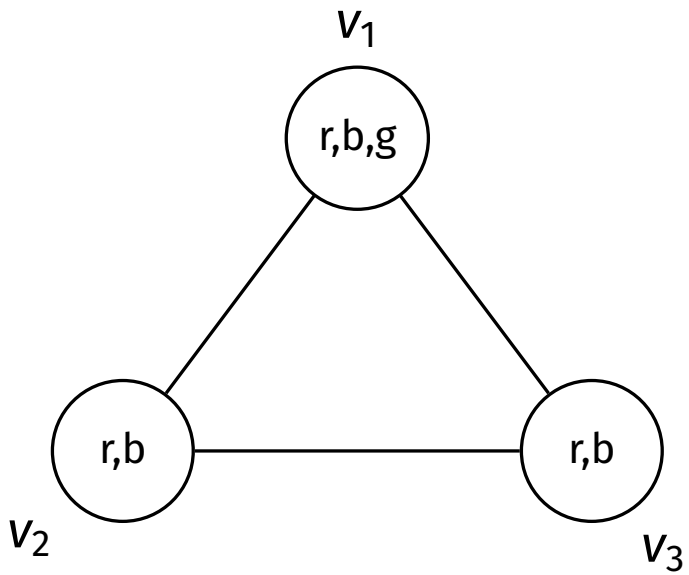
arc consistent, but not path consistent



$$c_{12} = \langle (v_1, v_2), \{(r, b), (b, r), (g, r), (g, b)\}$$
$$c_{13} = \langle (v_1, v_3), \{(r, b), (b, r), (g, r), (g, b)\}$$
$$c_{23} = \langle (v_2, v_3), \{(r, b), (b, r)\}$$

Path Consistency: Example

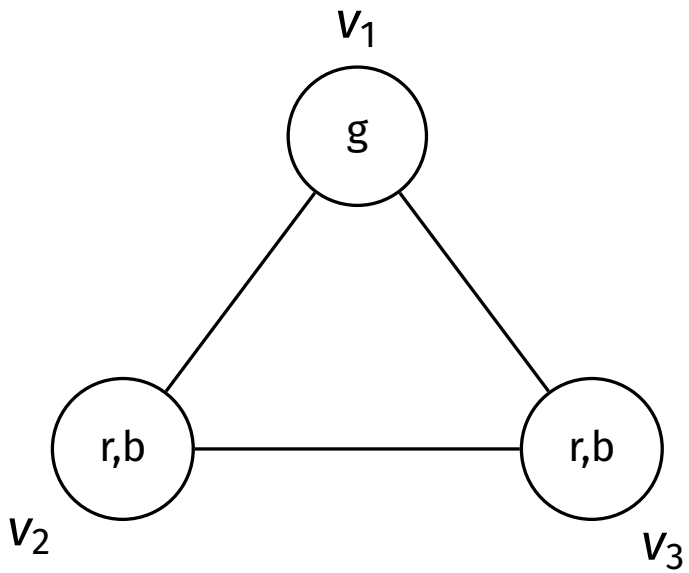
not arc consistent, but path consistent



$$c_{12} = \langle (v_1, v_2), \{(g, r), (g, b)\}$$
$$c_{13} = \langle (v_1, v_3), \{(g, r), (g, b)\}$$
$$c_{23} = \langle (v_2, v_3), \{(r, b), (b, r)\}$$

Path Consistency: Example

arc consistent and path consistent



$$c_{12} = \langle (v_1, v_2), \{(g, r), (g, b)\}$$
$$c_{13} = \langle (v_1, v_3), \{(g, r), (g, b)\}$$
$$c_{23} = \langle (v_2, v_3), \{(r, b), (b, r)\}$$