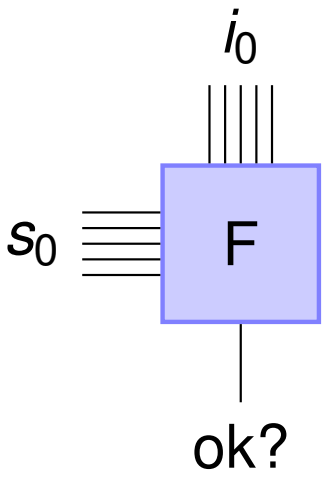
Temporal Induction and SAT-Solving

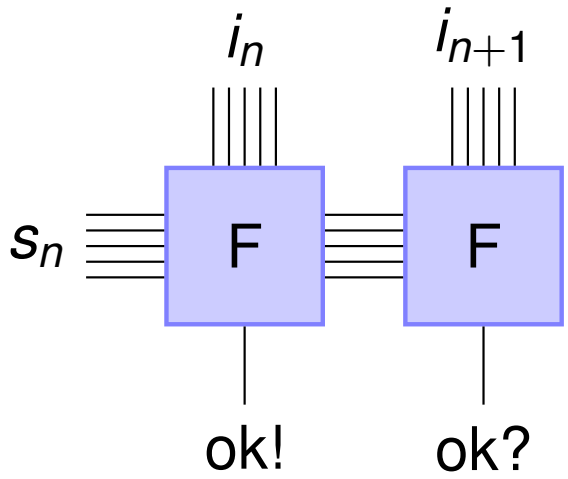
Niklas Sörensson

April 21, 2010

Simple Induction

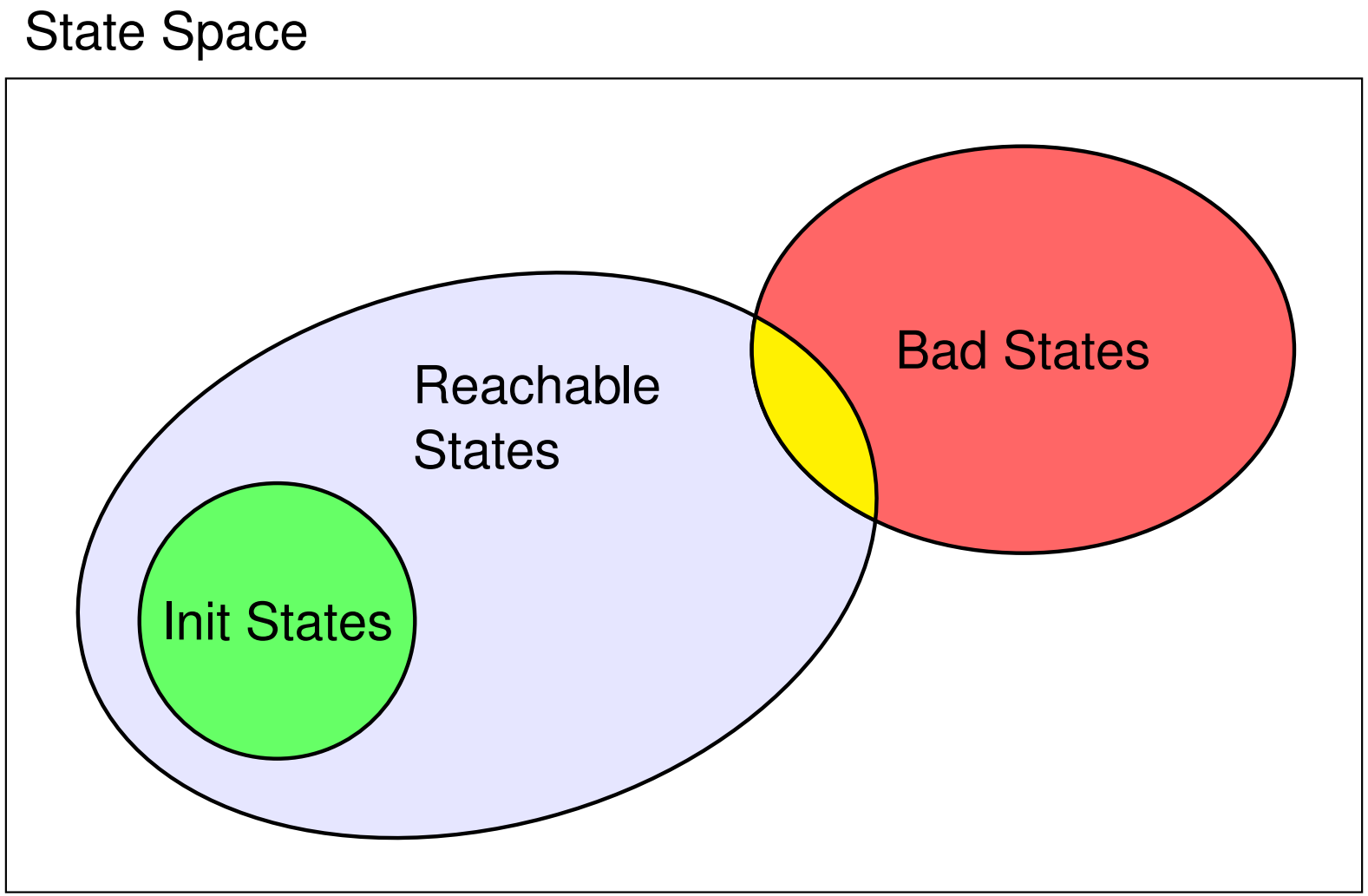


(Base-case)

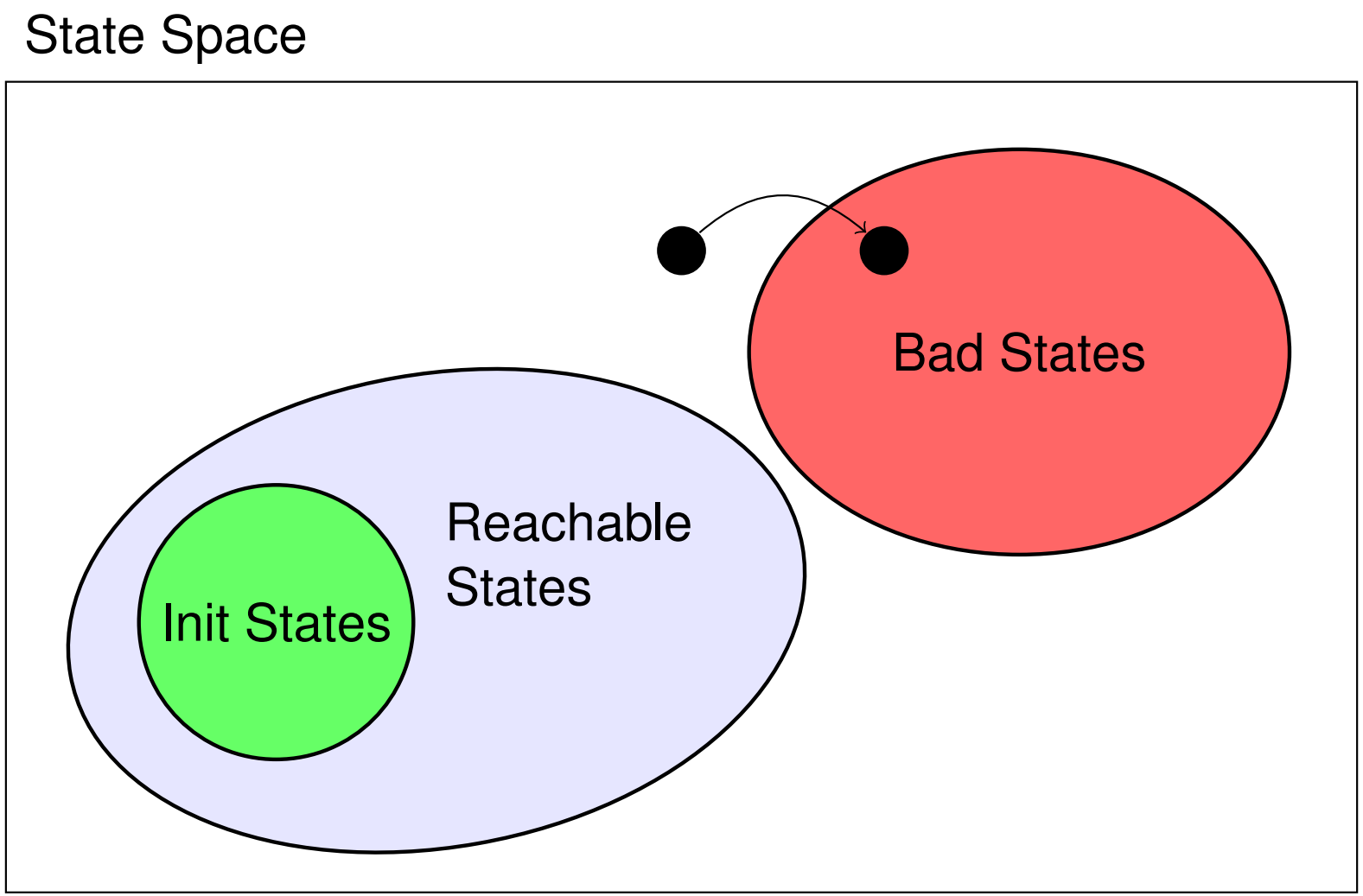


(Step-case)

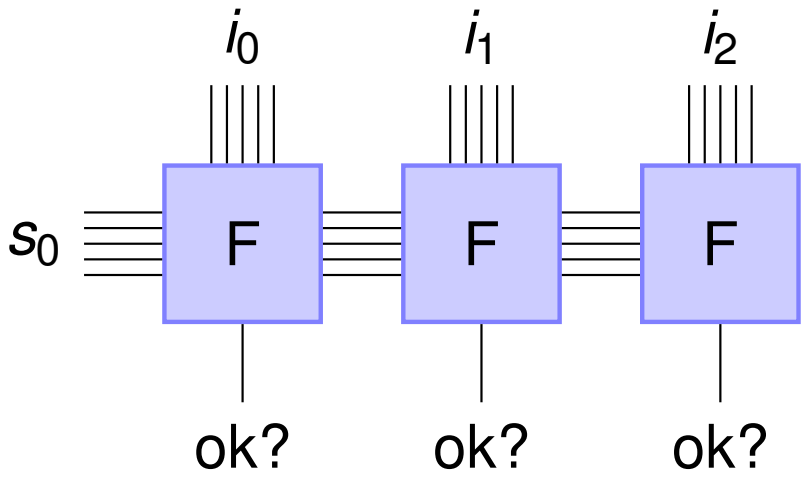
Reachability



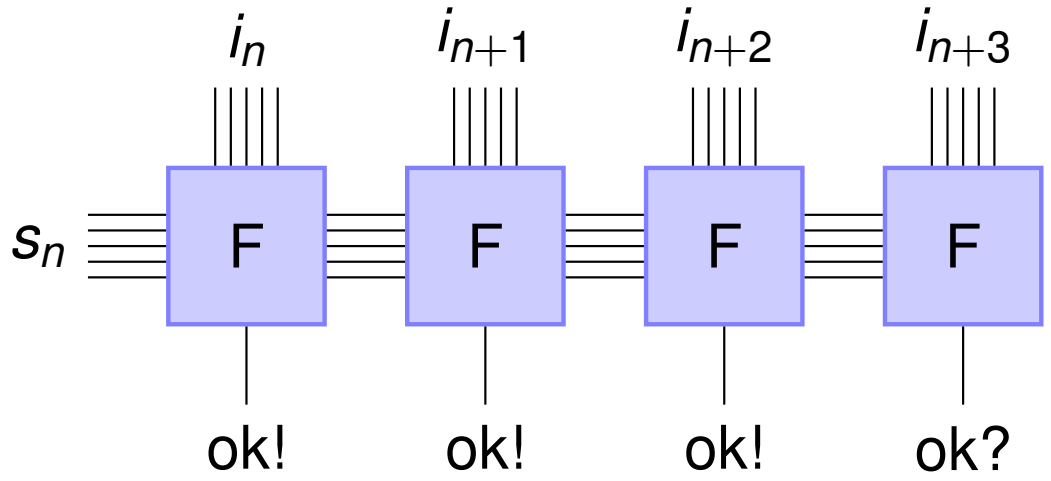
Example: Simple Induction Fails



Induction with Depth (k=3)

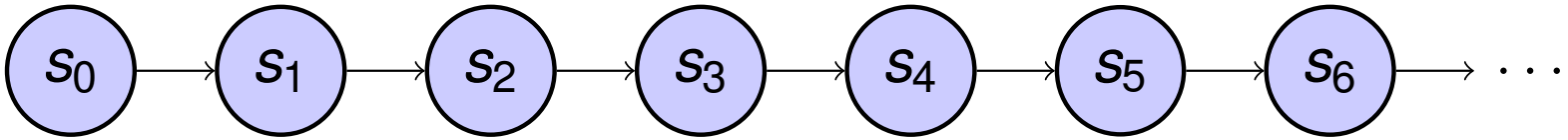


(Base-case)



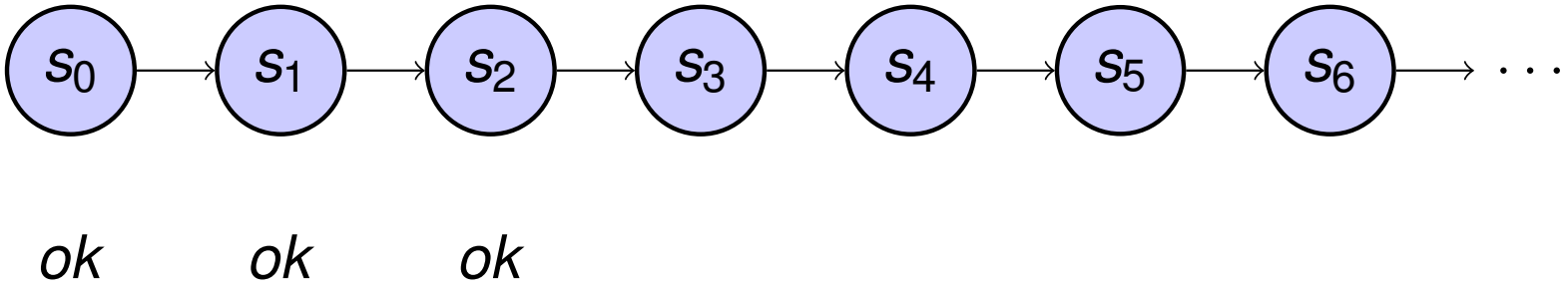
(Step-case)

Correctness of Induction with Depth (k=3)



For any trace:

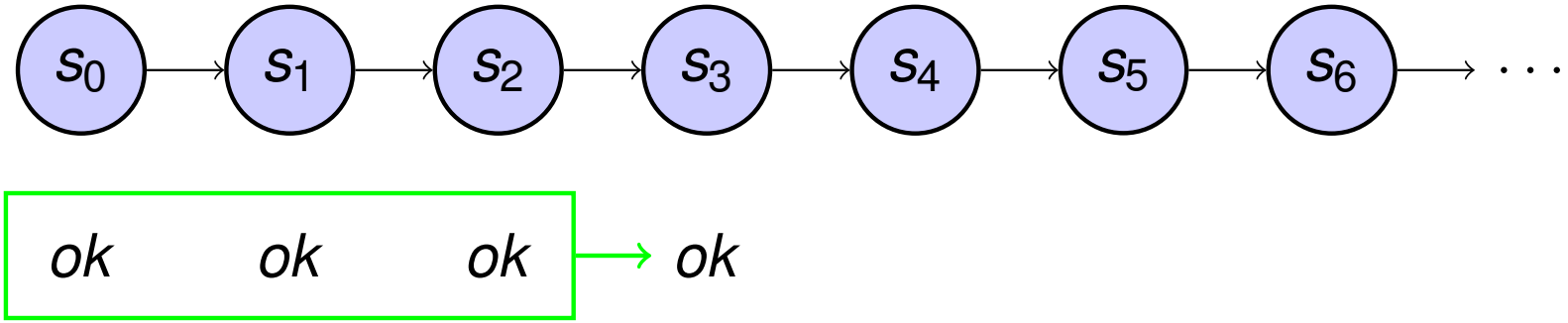
Correctness of Induction with Depth (k=3)



For any trace:

- First 3 states are ok

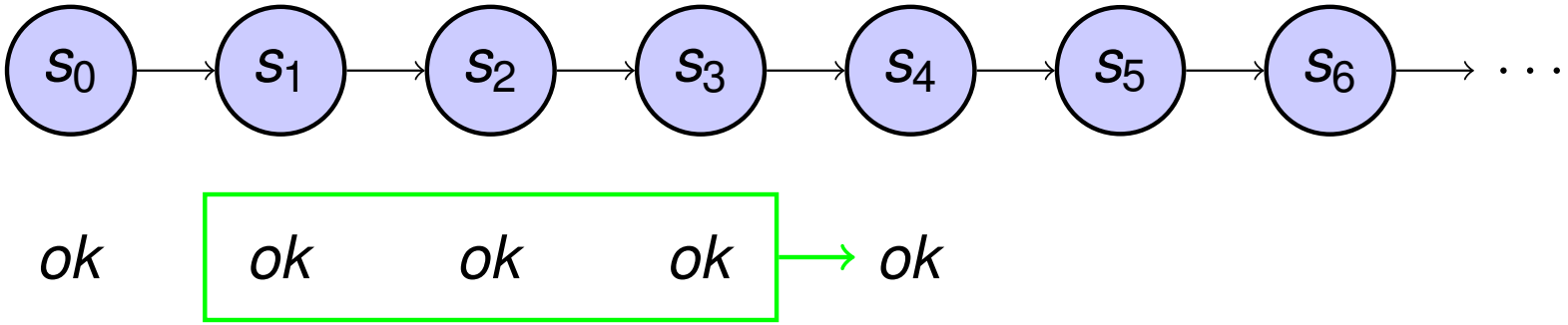
Correctness of Induction with Depth (k=3)



For any trace:

- First 3 states are ok
- 3 consecutive ok states must be succeeded by an ok state

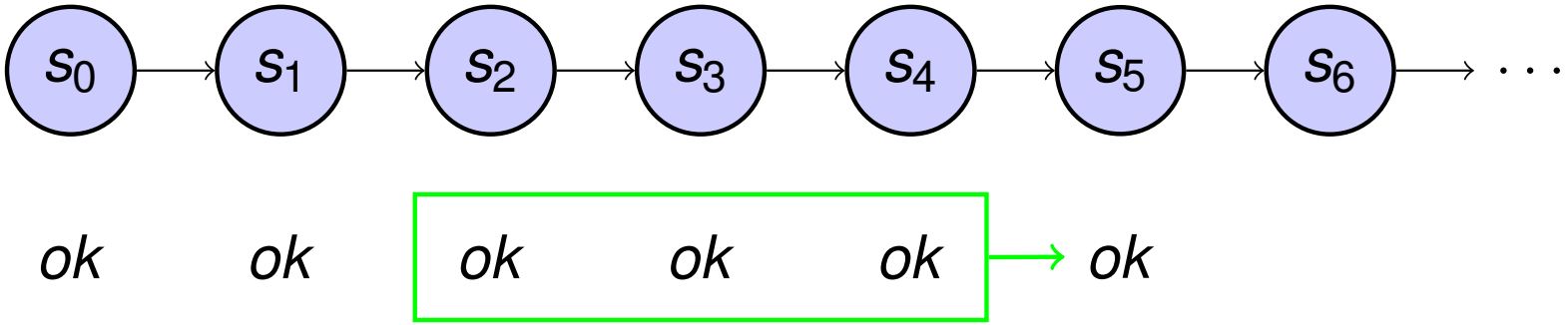
Correctness of Induction with Depth (k=3)



For any trace:

- First 3 states are ok
- 3 consecutive ok states must be succeeded by an ok state
- And so on

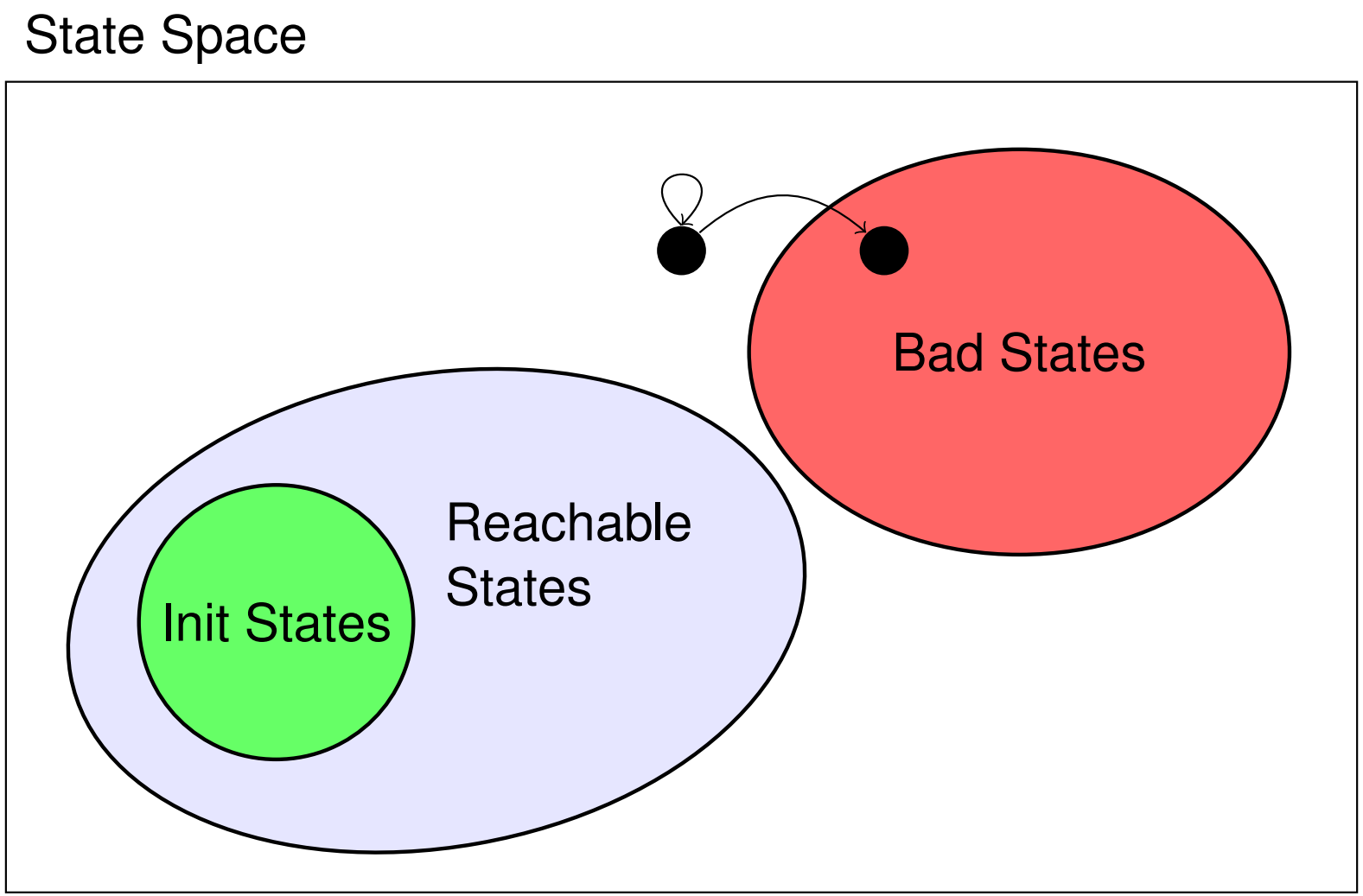
Correctness of Induction with Depth (k=3)



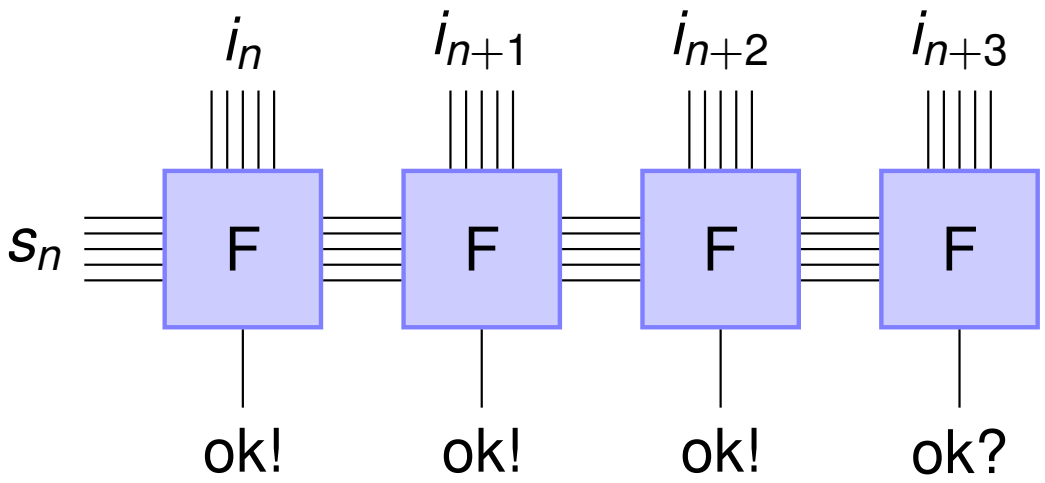
For any trace:

- First 3 states are ok
- 3 consecutive ok states must be succeeded by an ok state
- And so on

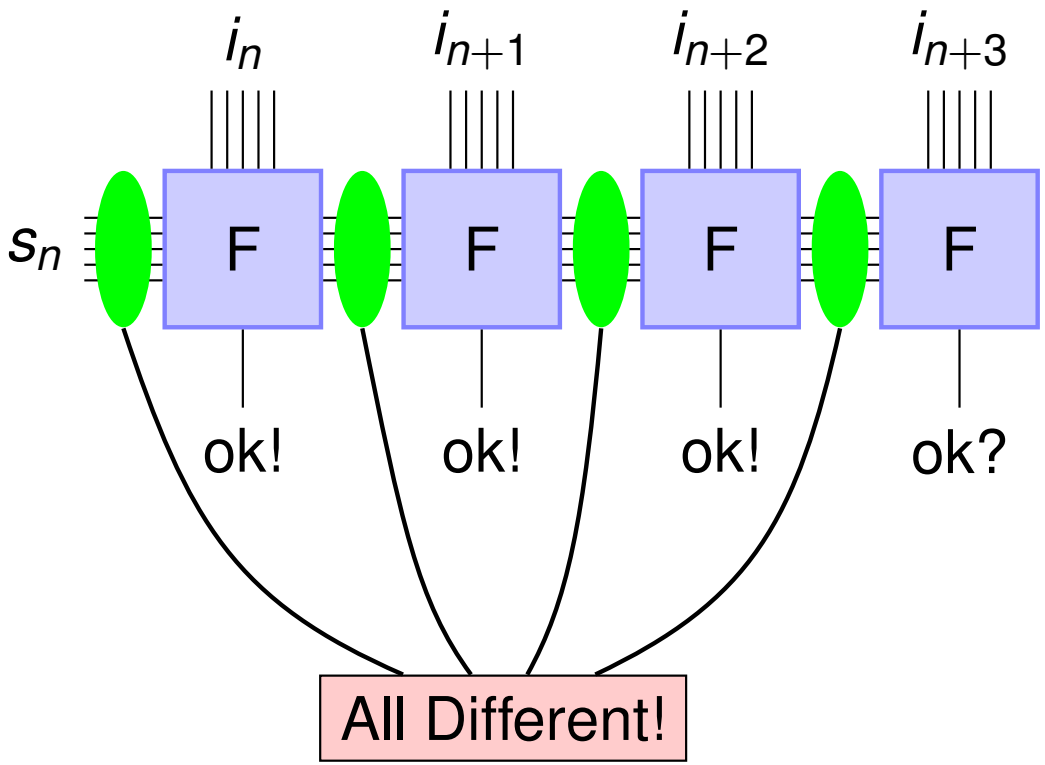
Example: Induction with Depth Fails



Unique State Induction (k=3)



Unique State Induction (k=3)



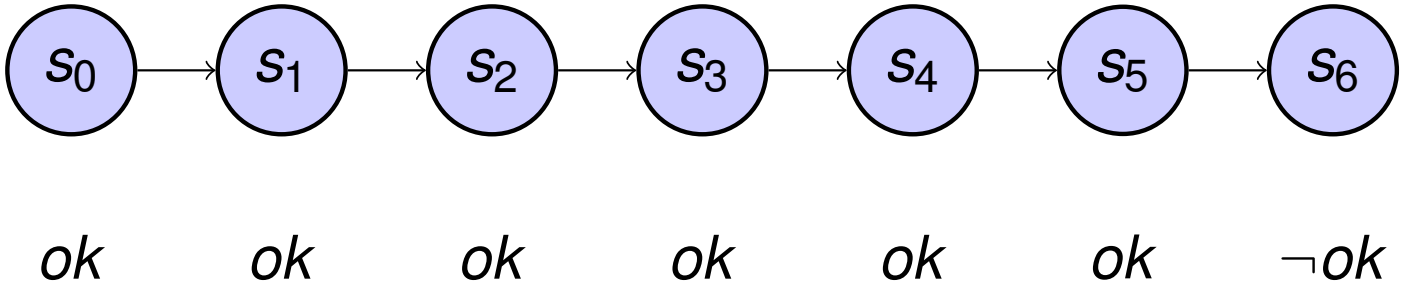
Correctness of Unique State Induction

- Unique state induction proves that *all loop-free paths are ok*
- Why can paths with loops be ignored?

Correctness of Unique State Induction

- Unique state induction proves that *all loop-free paths are ok*
- Why can paths with loops be ignored?

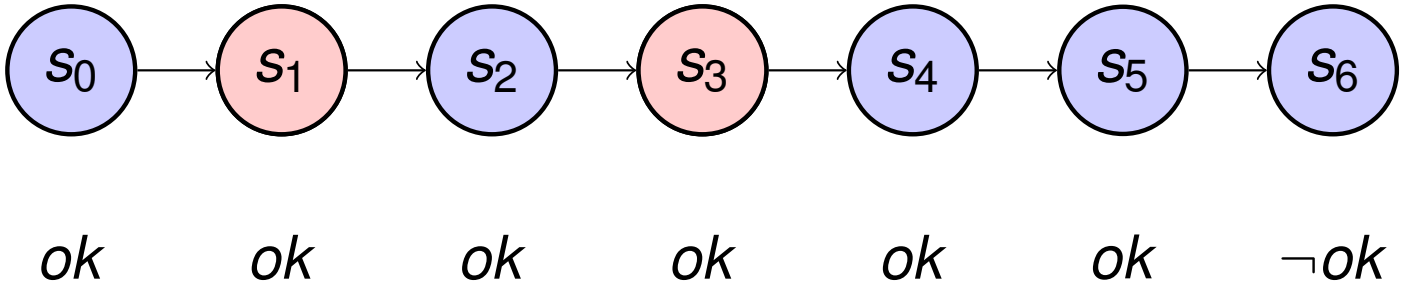
Example trace: $s_1 = s_3$, $s_4 = s_5$



Correctness of Unique State Induction

- Unique state induction proves that *all loop-free paths are ok*
- Why can paths with loops be ignored?

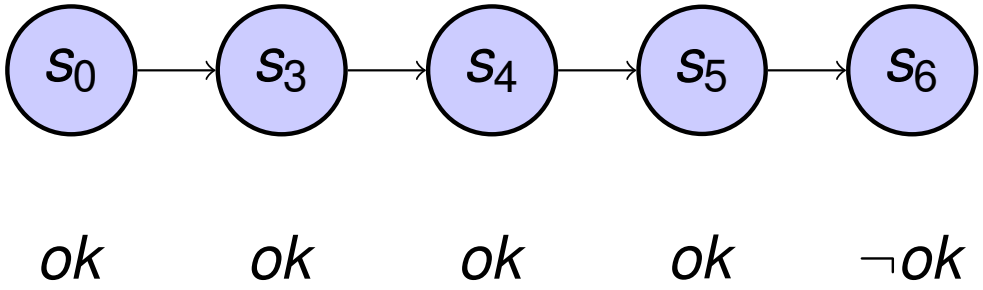
Example trace: $s_1 = s_3$, $s_4 = s_5$



Correctness of Unique State Induction

- Unique state induction proves that *all loop-free paths are ok*
- Why can paths with loops be ignored?

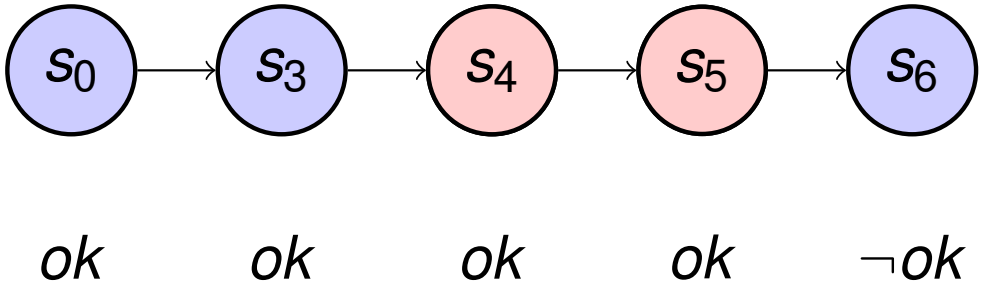
Example trace: $s_1 = s_3$, $s_4 = s_5$



Correctness of Unique State Induction

- Unique state induction proves that *all loop-free paths are ok*
- Why can paths with loops be ignored?

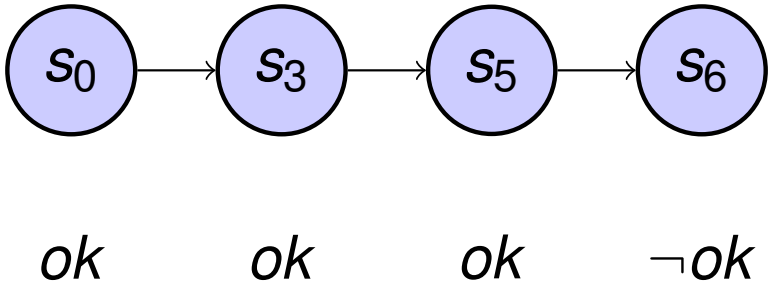
Example trace: $s_1 = s_3$, $s_4 = s_5$



Correctness of Unique State Induction

- Unique state induction proves that *all loop-free paths are ok*
- Why can paths with loops be ignored?

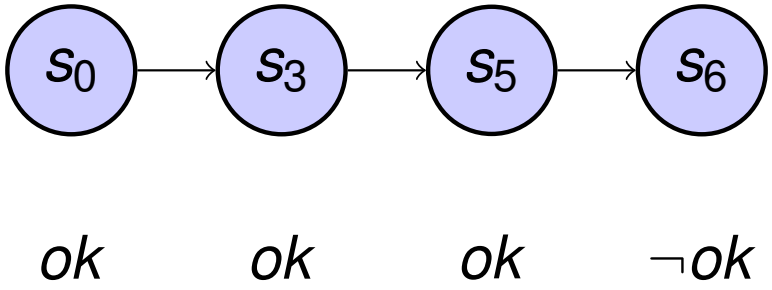
Example trace: $s_1 = s_3$, $s_4 = s_5$



Correctness of Unique State Induction

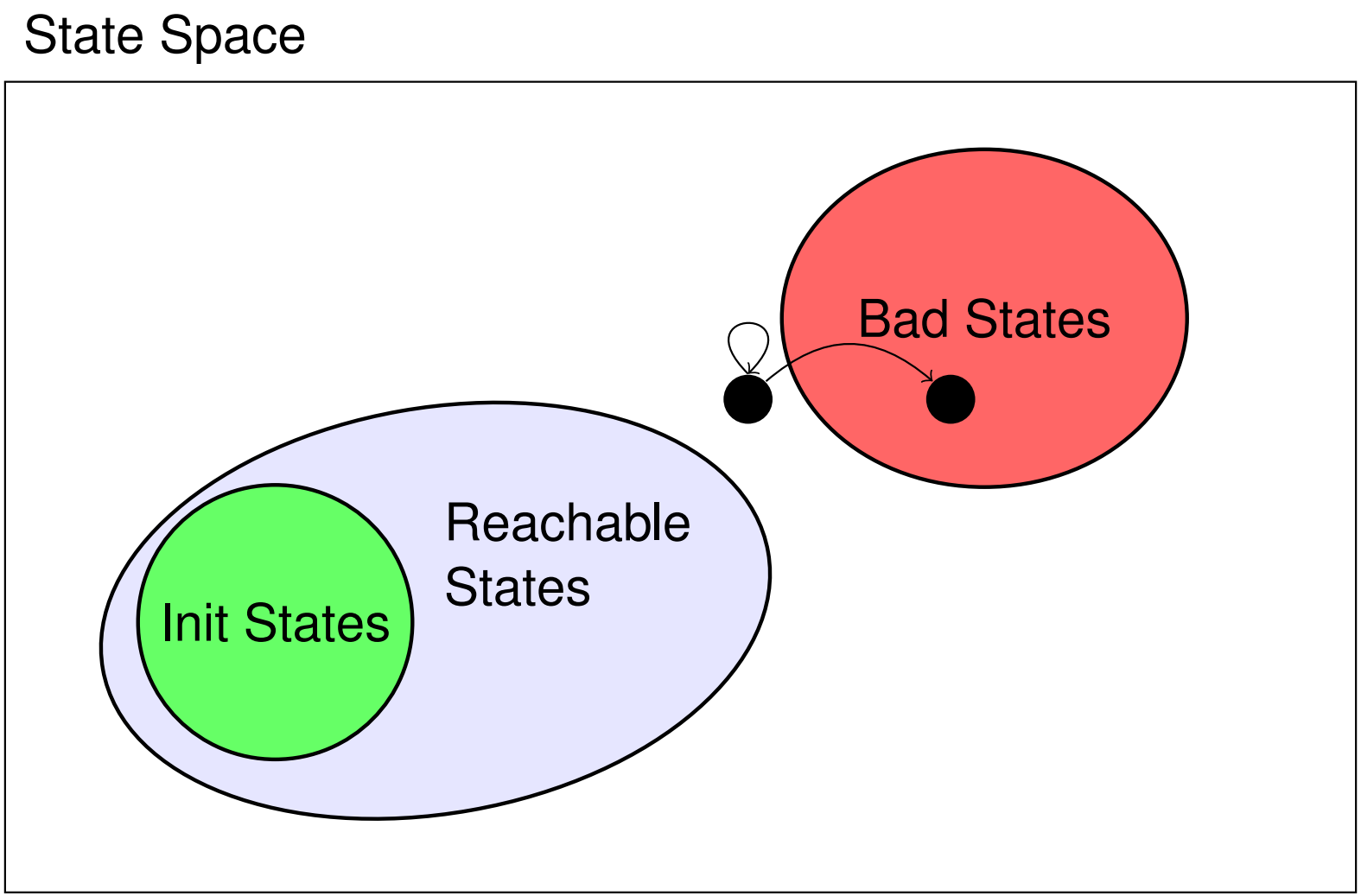
- Unique state induction proves that *all loop-free paths are ok*
- Why can paths with loops be ignored?

Example trace: $s_1 = s_3$, $s_4 = s_5$

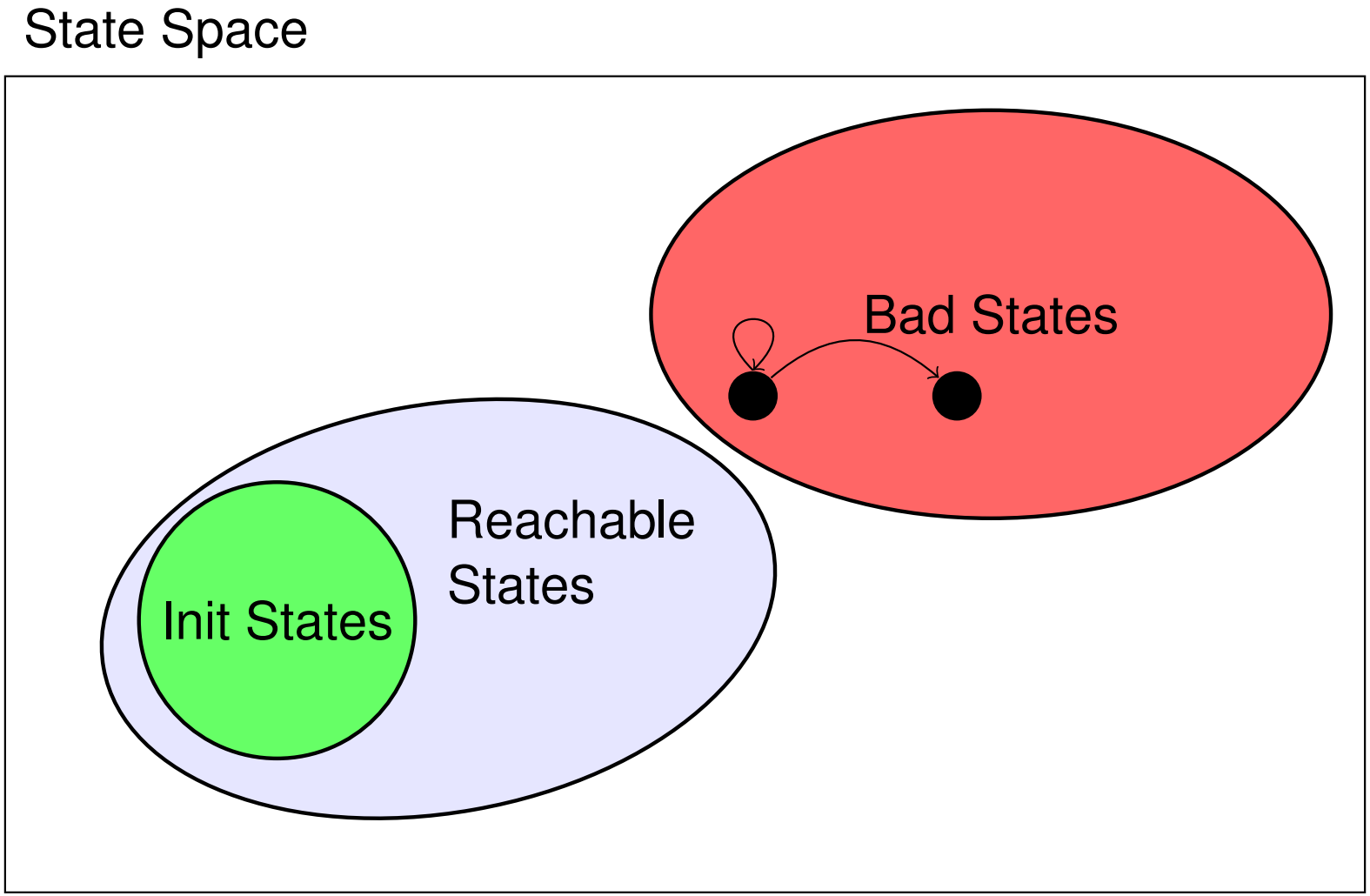


Counter-examples with loops can always be made loop-free!

Property Generalization



Property Generalization



Induction in Practice

- Complete

Induction in Practice

- Complete
- Increase depth until proof goes through (or counter example is found)

Induction in Practice

- Complete
- Increase depth until proof goes through (or counter example is found)
- Necessary depth can be very large (exponential)

Induction in Practice

- Complete
- Increase depth until proof goes through (or counter example is found)
- Necessary depth can be very large (exponential)
- Induction may be strengthened by:

Induction in Practice

- Complete
- Increase depth until proof goes through (or counter example is found)
- Necessary depth can be very large (exponential)
- Induction may be strengthened by:
  - Multiple properties

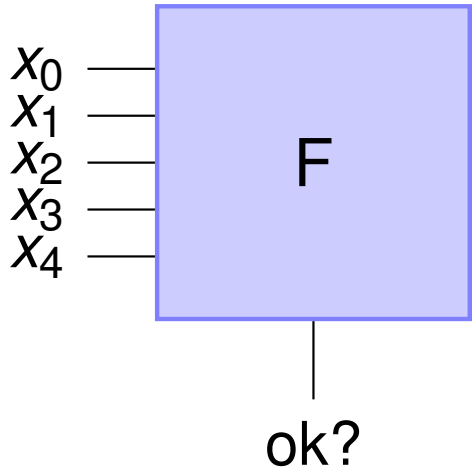
Induction in Practice

- Complete
- Increase depth until proof goes through (or counter example is found)
- Necessary depth can be very large (exponential)
- Induction may be strengthened by:
  - Multiple properties
  - Manual generalization

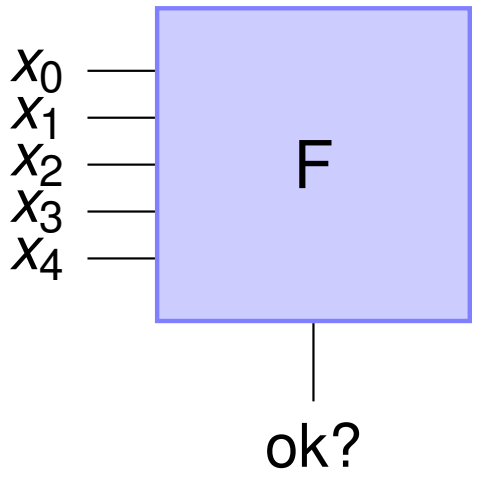
Induction in Practice

- Complete
- Increase depth until proof goes through (or counter example is found)
- Necessary depth can be very large (exponential)
- Induction may be strengthened by:
  - Multiple properties
  - Manual generalization
  - Automatic generalization

Satisfiability (SAT) in BMC and Induction

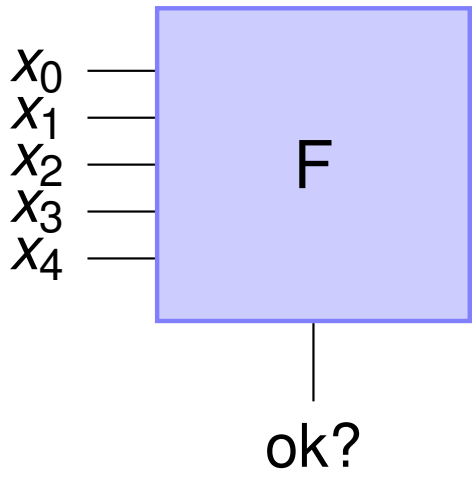


Satisfiability (SAT) in BMC and Induction



- Find values for inputs $x_0 \ldots x_n$ such that *ok* becomes false

Satisfiability (SAT) in BMC and Induction



- Find values for inputs $x_0 \ldots x_n$ such that *ok* becomes false
- Or, prove that no such set of values exists

Why is it hard?

x_0	x_1	x_2	x_3	ok
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

- Enumeration does not work
- Given k inputs there are 2^k combinations

Conjunctive Normal Form

Variables: $a, b, c, \ldots$
Literals: $a, \neg a, b, \neg b, c, \neg c, \ldots$

Clauses: *Disjunctions of literals*

Ex: $(\neg a \vee b, \neg c)$

CNF: *Conjunction of clauses*

Ex: $(\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg b \vee c)$

SAT-Problem Definition

- A CNF is *satisfiable* if there is a variable assignment that evaluates all clauses to true
- (i.e. each clause contains at least one true literal)
- If no satisfying assignment exists, the CNF is *unsatisfiable*

SAT-Problem Definition

- A CNF is *satisfiable* if there is a variable assignment that evaluates all clauses to true
- (i.e. each clause contains at least one true literal)
- If no satisfying assignment exists, the CNF is *unsatisfiable*

Example

$(a \vee b)$
$(\neg a \vee \neg b)$

SAT-Problem Definition

- A CNF is *satisfiable* if there is a variable assignment that evaluates all clauses to true
- (i.e. each clause contains at least one true literal)
- If no satisfying assignment exists, the CNF is *unsatisfiable*

Example

$(a \vee b)$
$(\neg a \vee \neg b)$

- $a = 1, b = 1$ is not a satisfying assignment

SAT-Problem Definition

- A CNF is *satisfiable* if there is a variable assignment that evaluates all clauses to true
- (i.e. each clause contains at least one true literal)
- If no satisfying assignment exists, the CNF is *unsatisfiable*

Example

$$(a \vee b)$$
$$(\neg a \vee \neg b)$$

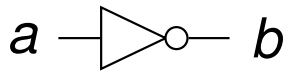
- $a = 1, b = 1$ is not a satisfying assignment
- $a = 1, b = 0$ is a satisfying assignment

Translating from Circuits into CNF

- Give fresh (SAT) variables for each input/gate
- Define clauses that establish input/output relationship of each gate

Translating from Circuits into CNF

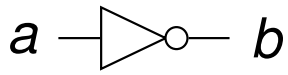
- Give fresh (SAT) variables for each input/gate
- Define clauses that establish input/output relationship of each gate



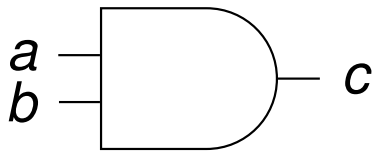
$$(\neg b \vee \neg a)$$
$$(a \vee b)$$

Translating from Circuits into CNF

- Give fresh (SAT) variables for each input/gate
- Define clauses that establish input/output relationship of each gate



$$(\neg b \vee \neg a)$$
$$(a \vee b)$$



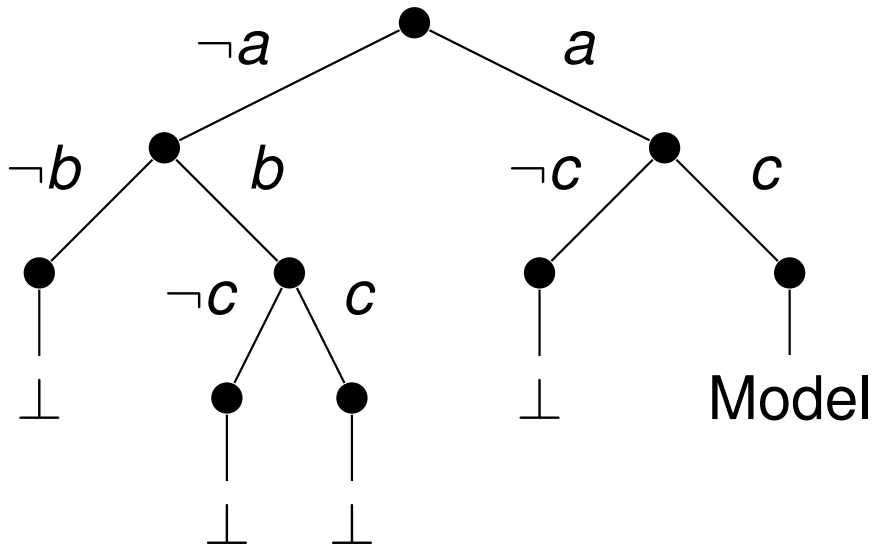
$$(\neg c \vee a)$$
$$(\neg c \vee b)$$
$$(\neg a \vee \neg b \vee c)$$

DPLL Algorithm: Backtracking Search

- Branch on some variable
- Solve each branch recursively
- Backtrack when some clause is falsified

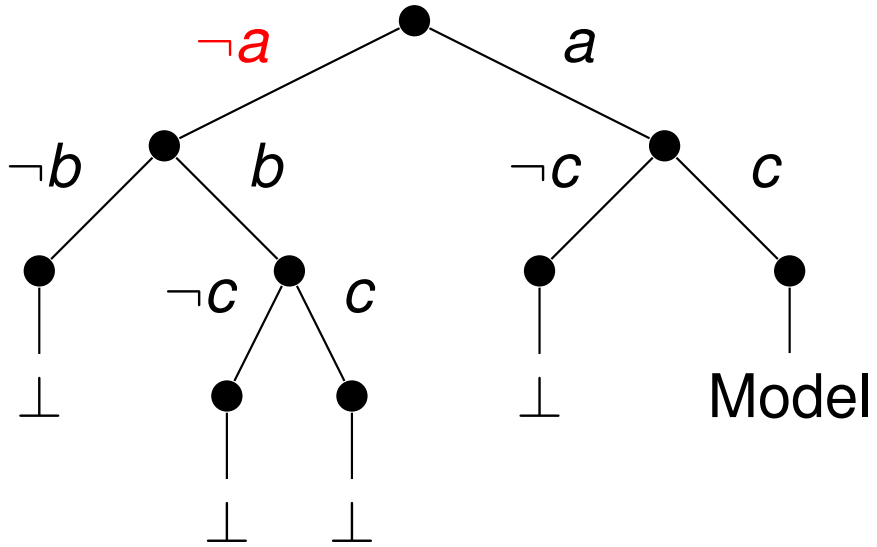
DPLL Algorithm: Backtracking Search

- Branch on some variable
- Solve each branch recursively
- Backtrack when some clause is falsified



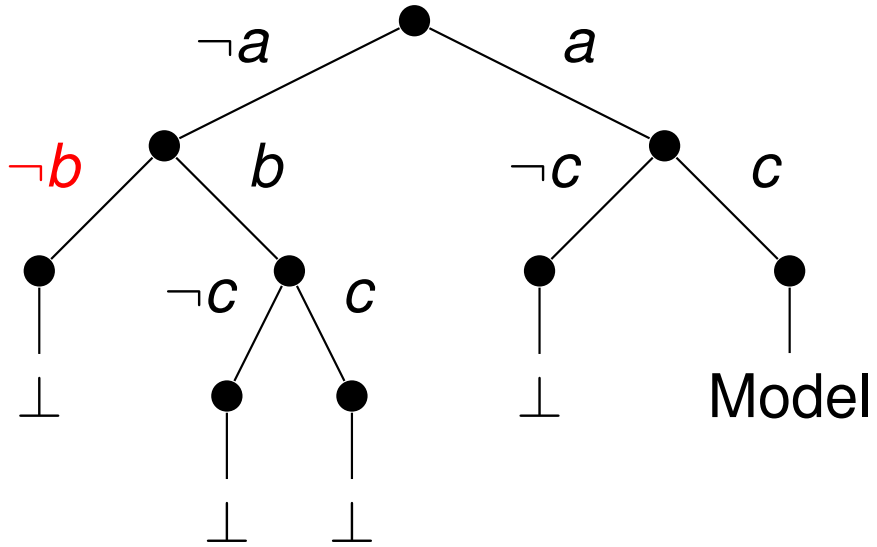
DPLL Algorithm: Backtracking Search

- Branch on some variable
- Solve each branch recursively
- Backtrack when some clause is falsified



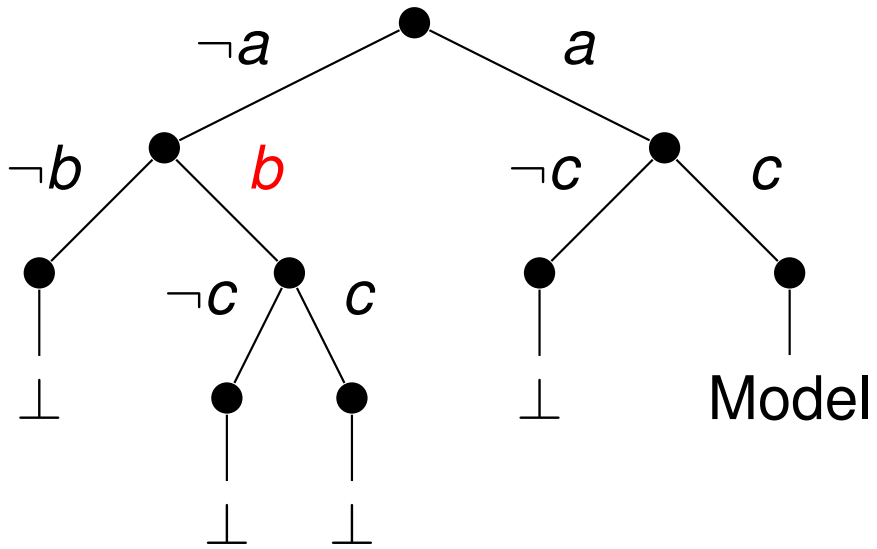
DPLL Algorithm: Backtracking Search

- Branch on some variable
- Solve each branch recursively
- Backtrack when some clause is falsified



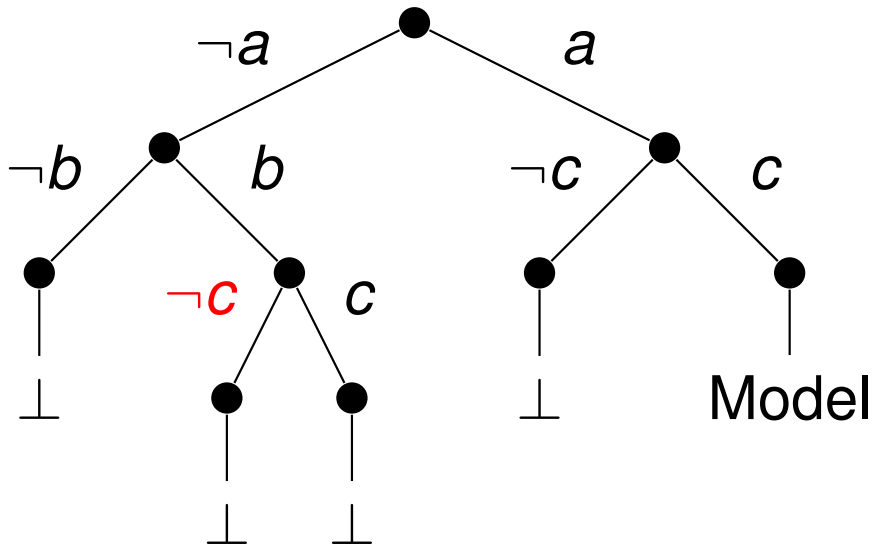
DPLL Algorithm: Backtracking Search

- Branch on some variable
- Solve each branch recursively
- Backtrack when some clause is falsified



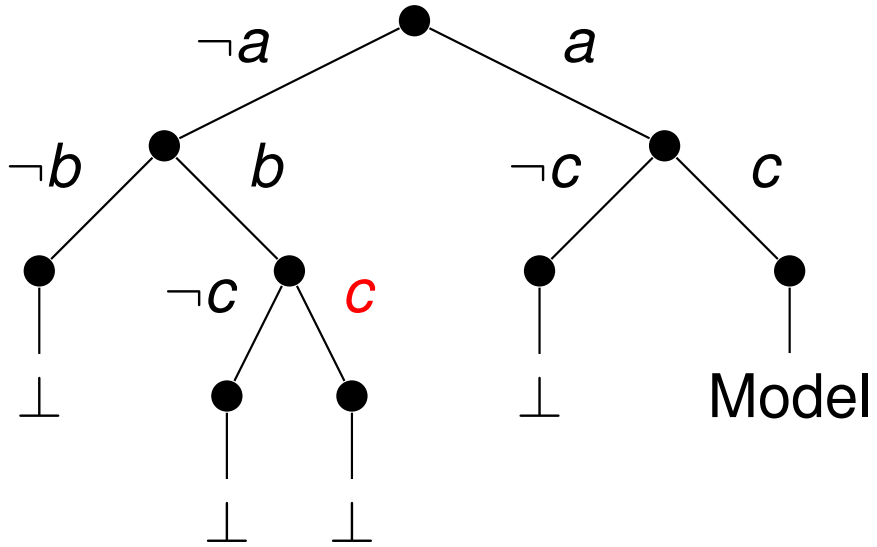
DPLL Algorithm: Backtracking Search

- Branch on some variable
- Solve each branch recursively
- Backtrack when some clause is falsified



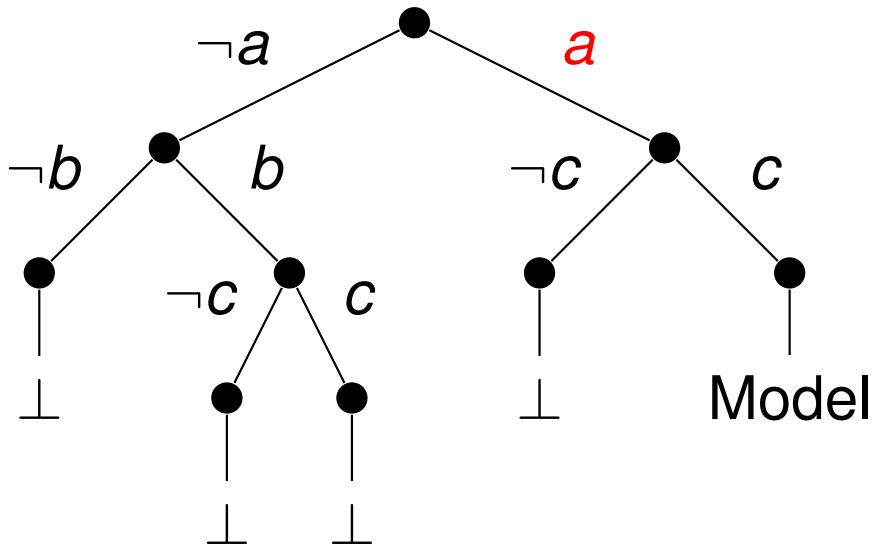
DPLL Algorithm: Backtracking Search

- Branch on some variable
- Solve each branch recursively
- Backtrack when some clause is falsified



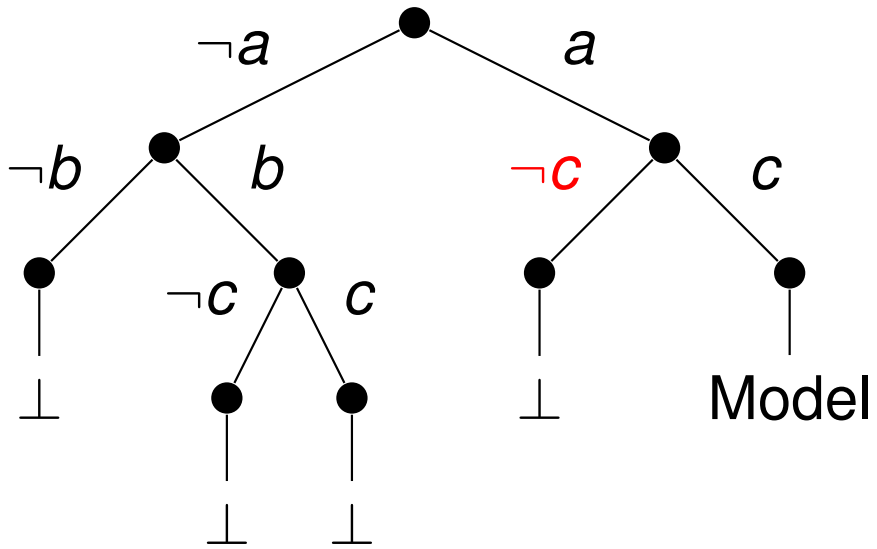
DPLL Algorithm: Backtracking Search

- Branch on some variable
- Solve each branch recursively
- Backtrack when some clause is falsified



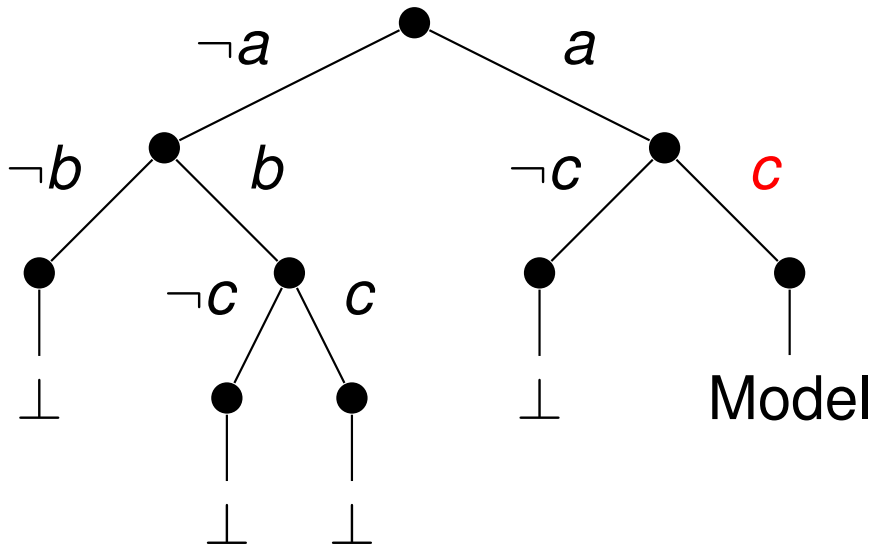
DPLL Algorithm: Backtracking Search

- Branch on some variable
- Solve each branch recursively
- Backtrack when some clause is falsified



DPLL Algorithm: Backtracking Search

- Branch on some variable
- Solve each branch recursively
- Backtrack when some clause is falsified



DPLL Algorithm: Unit Propagation

- Extend partial assignment with *necessary* implied assignments
- Avoids branching and reduces search space

DPLL Algorithm: Unit Propagation

- Extend partial assignment with *necessary* implied assignments
- Avoids branching and reduces search space

Example

$(\neg a \vee b)$
$(\neg a \vee c)$
$(\neg b \vee \neg c \vee d)$

DPLL Algorithm: Unit Propagation

- Extend partial assignment with *necessary* implied assignments
- Avoids branching and reduces search space

Example

$(\neg a \vee b)$
$(\neg a \vee c)$
$(\neg b \vee \neg c \vee d)$

$a = 1$

DPLL Algorithm: Unit Propagation

- Extend partial assignment with *necessary* implied assignments
- Avoids branching and reduces search space

Example

$(\neg a \vee b)$
$(\neg a \vee c)$
$(\neg b \vee \neg c \vee d)$

$a = 1$
$b = 1$

DPLL Algorithm: Unit Propagation

- Extend partial assignment with *necessary* implied assignments
- Avoids branching and reduces search space

Example

$(\neg a \vee b)$
$(\neg a \vee c)$
$(\neg b \vee \neg c \vee d)$

$a = 1$
$b = 1$
$c = 1$

DPLL Algorithm: Unit Propagation

- Extend partial assignment with *necessary* implied assignments
- Avoids branching and reduces search space

Example

$(\neg a \vee b)$
$(\neg a \vee c)$
$(\neg b \vee \neg c \vee d)$

$a = 1$
$b = 1$
$c = 1$
$d = 1$

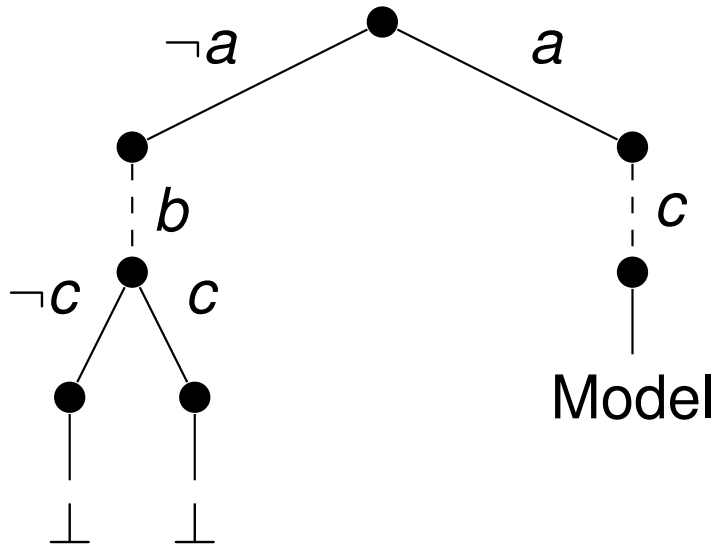
DPLL Algorithm: Unit Propagation

- Extend partial assignment with *necessary* implied assignments
- Avoids branching and reduces search space

Example

$(\neg a \vee b)$
$(\neg a \vee c)$
$(\neg b \vee \neg c \vee d)$

$a = 1$
$b = 1$
$c = 1$
$d = 1$



DPLL Algorithm: Further Topics

- Pure literal rule (not used nowadays)

DPLL Algorithm: Further Topics

- Pure literal rule (not used nowadays)
- Efficient Unit Propagation (Watched Literals)

DPLL Algorithm: Further Topics

- Pure literal rule (not used nowadays)
- Efficient Unit Propagation (Watched Literals)
- Variable Order Heuristics

DPLL Algorithm: Further Topics

- Pure literal rule (not used nowadays)
- Efficient Unit Propagation (Watched Literals)
- Variable Order Heuristics
- Non-chronological Backtracking

DPLL Algorithm: Further Topics

- Pure literal rule (not used nowadays)
- Efficient Unit Propagation (Watched Literals)
- Variable Order Heuristics
- Non-chronological Backtracking
- Learning

DPLL Algorithm: Further Topics

- Pure literal rule (not used nowadays)
- Efficient Unit Propagation (Watched Literals)
- Variable Order Heuristics
- Non-chronological Backtracking
- Learning
- Restarts

SAT Research

- SAT algorithms

SAT Research

- SAT algorithms
  - Heuristics & Data-structures

SAT Research

- SAT algorithms
  - Heuristics & Data-structures
  - Some would say more engineering than science

SAT Research

- SAT algorithms
  - Heuristics & Data-structures
  - Some would say more engineering than science
  - Stimulated by annual SAT-competition

SAT Research

- SAT algorithms
  - Heuristics & Data-structures
  - Some would say more engineering than science
  - Stimulated by annual SAT-competition
- Applications of SAT

SAT Research

- SAT algorithms
  - Heuristics & Data-structures
  - Some would say more engineering than science
  - Stimulated by annual SAT-competition
- Applications of SAT
  - Planning

SAT Research

- SAT algorithms
  - Heuristics & Data-structures
  - Some would say more engineering than science
  - Stimulated by annual SAT-competition
- Applications of SAT
  - Planning
  - FPGA routing

SAT Research

- SAT algorithms
  - Heuristics & Data-structures
  - Some would say more engineering than science
  - Stimulated by annual SAT-competition
- Applications of SAT
  - Planning
  - FPGA routing
  - Puzzles (Sudoku etc)

SAT Research

- SAT algorithms
  - Heuristics & Data-structures
  - Some would say more engineering than science
  - Stimulated by annual SAT-competition
- Applications of SAT
  - Planning
  - FPGA routing
  - Puzzles (Sudoku etc)
- Extensions to SAT

SAT Research

- SAT algorithms
  - Heuristics & Data-structures
  - Some would say more engineering than science
  - Stimulated by annual SAT-competition
- Applications of SAT
  - Planning
  - FPGA routing
  - Puzzles (Sudoku etc)
- Extensions to SAT
  - 0-1 Integer Linear Programming

SAT Research

- SAT algorithms
  - Heuristics & Data-structures
  - Some would say more engineering than science
  - Stimulated by annual SAT-competition
- Applications of SAT
  - Planning
  - FPGA routing
  - Puzzles (Sudoku etc)
- Extensions to SAT
  - 0-1 Integer Linear Programming
  - Satisfiability Modulo Theories (SMT)

SAT Research

- SAT algorithms
  - Heuristics & Data-structures
  - Some would say more engineering than science
  - Stimulated by annual SAT-competition
- Applications of SAT
  - Planning
  - FPGA routing
  - Puzzles (Sudoku etc)
- Extensions to SAT
  - 0-1 Integer Linear Programming
  - Satisfiability Modulo Theories (SMT)
  - Model Finding

MiniSat

Developed during my PhD studies together with Niklas Een

MiniSat

Developed during my PhD studies together with Niklas Een

- Simple
- Efficient
- Incremental
- Successful in SAT-competitions
- Widespread use in academia

MiniSat

Developed during my PhD studies together with Niklas Een

- Simple
- Efficient
- Incremental
- Successful in SAT-competitions
- Widespread use in academia

Download

`http://minisat.se`