Verifying the DPLL Algorithm in Dafny

Cezar-Constantin Andrici Ștefan Ciobâcă

Alexandru Ioan Cuza University, Iași, Romania
stefan.ciobaca@gmail.com

September 5th, 2019
Working Formal Methods Symposium 2019 (FROM 2019)
Timișoara

This work was supported by a grant of the Alexandru Ioan Cuza University of Iași, within the Research Grants program UAIC Grant, code GI-UAIC-2018-07.

Outline

The SAT Problem

Input: a propositional formula ϕ (usually in CNF)
Output: is ϕ satisfiable? ($+$ witness)
Example input: $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3)$
Output: yes ($x_1 = 1, x_3 = 0, x_2 = ?$)

The SAT Problem

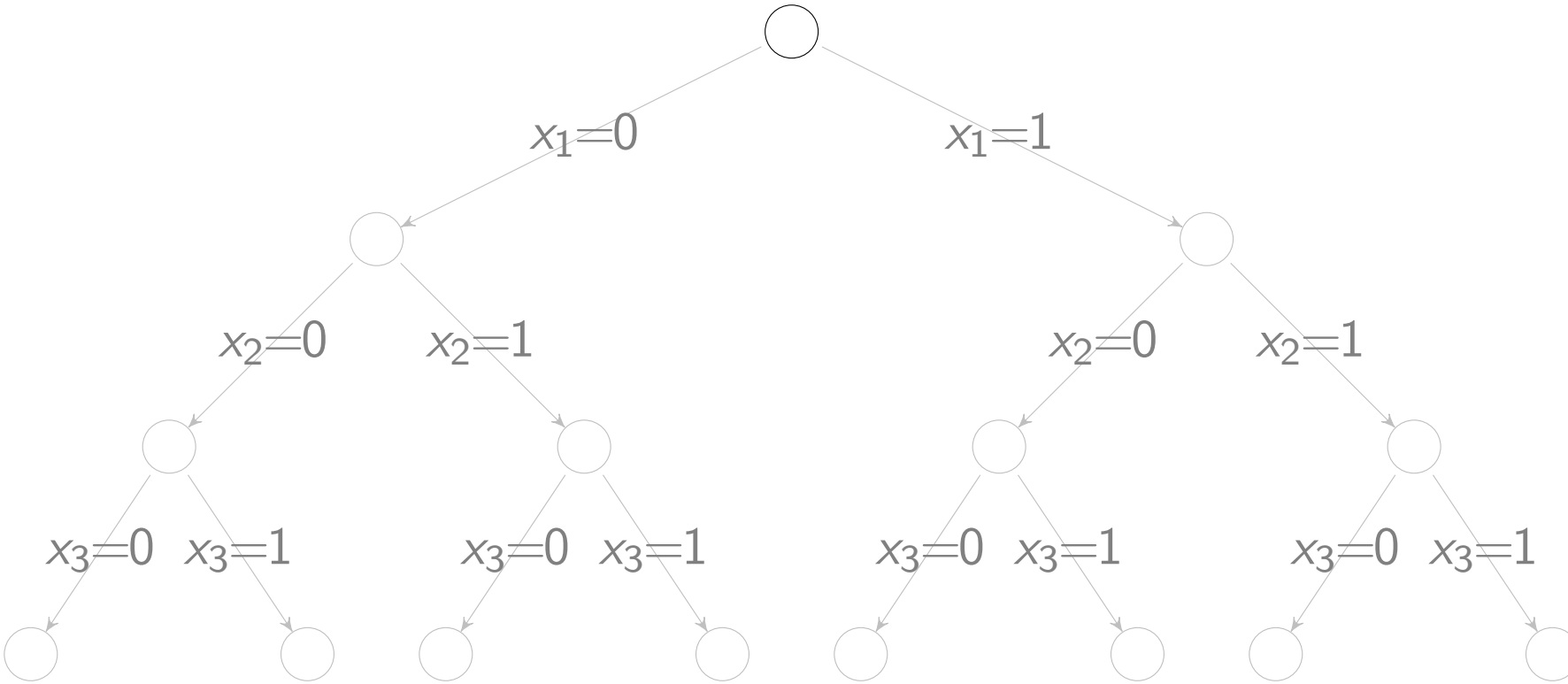
- canonical NP-complete problem;
- used in software and hardware verification:
  - bounded model checking,
  - functional equivalence,
  - SMT solving,
  - many others;
- combinatorial optimization, encoding of other (NP-complete) problems.

Relatively recently: very fast SAT solvers capable of solving in reasonable time instances with millions of variables/clauses.

Also: yearly SAT competition with many competitors and several tracks.
Fact: All state-of-the-art SAT solvers are based on CDCL (which is itself a refinement of DPLL, which is a refinement of backtracking).

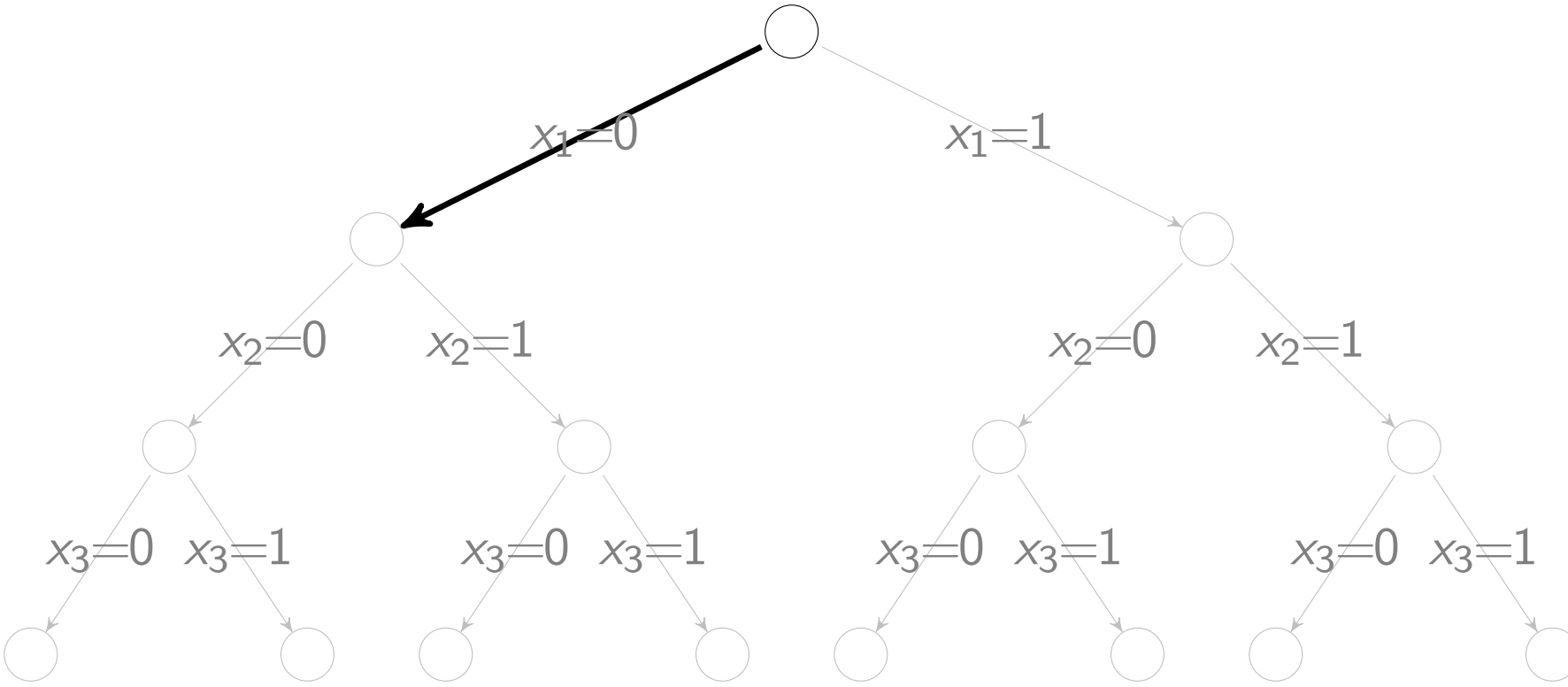
$x_1 = ?$ $x_2 = ?$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



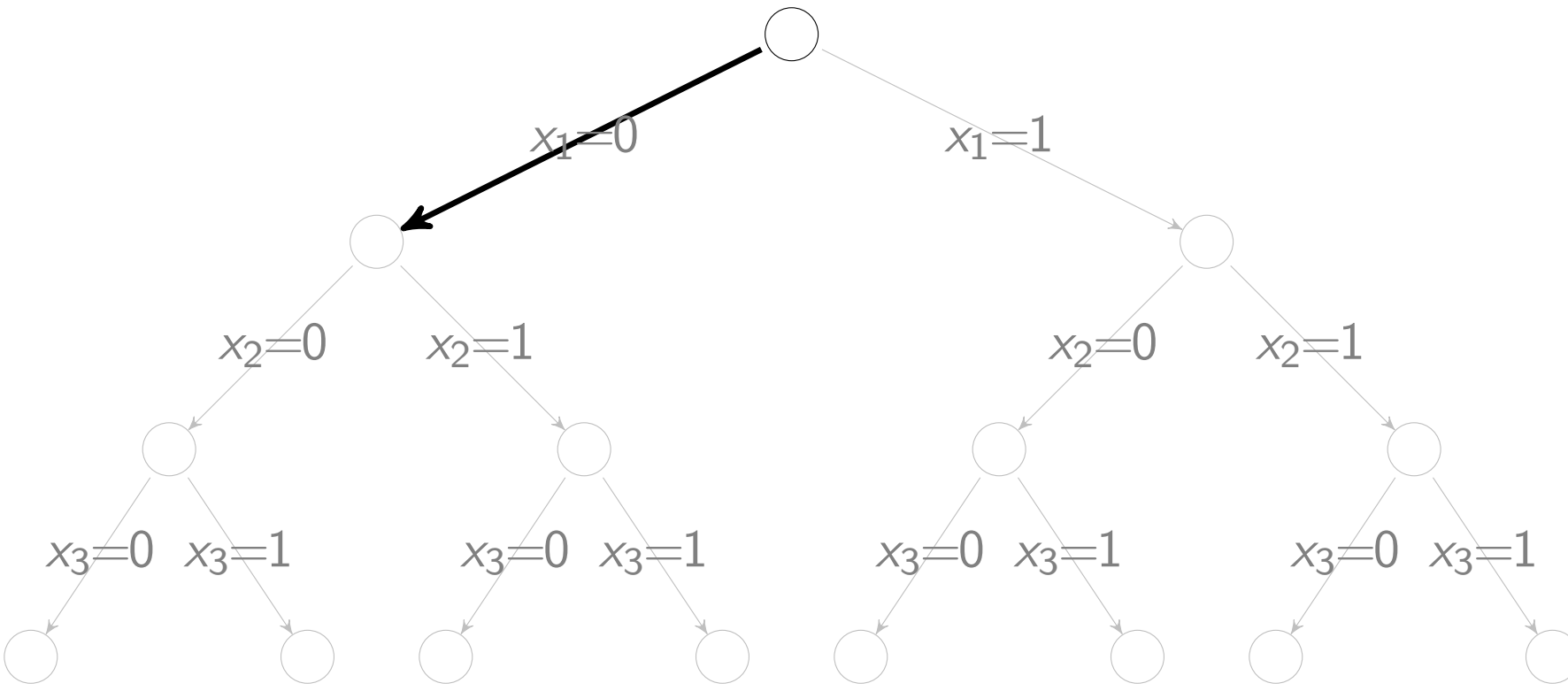
$x_1 = ?$ $x_2 = ?$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



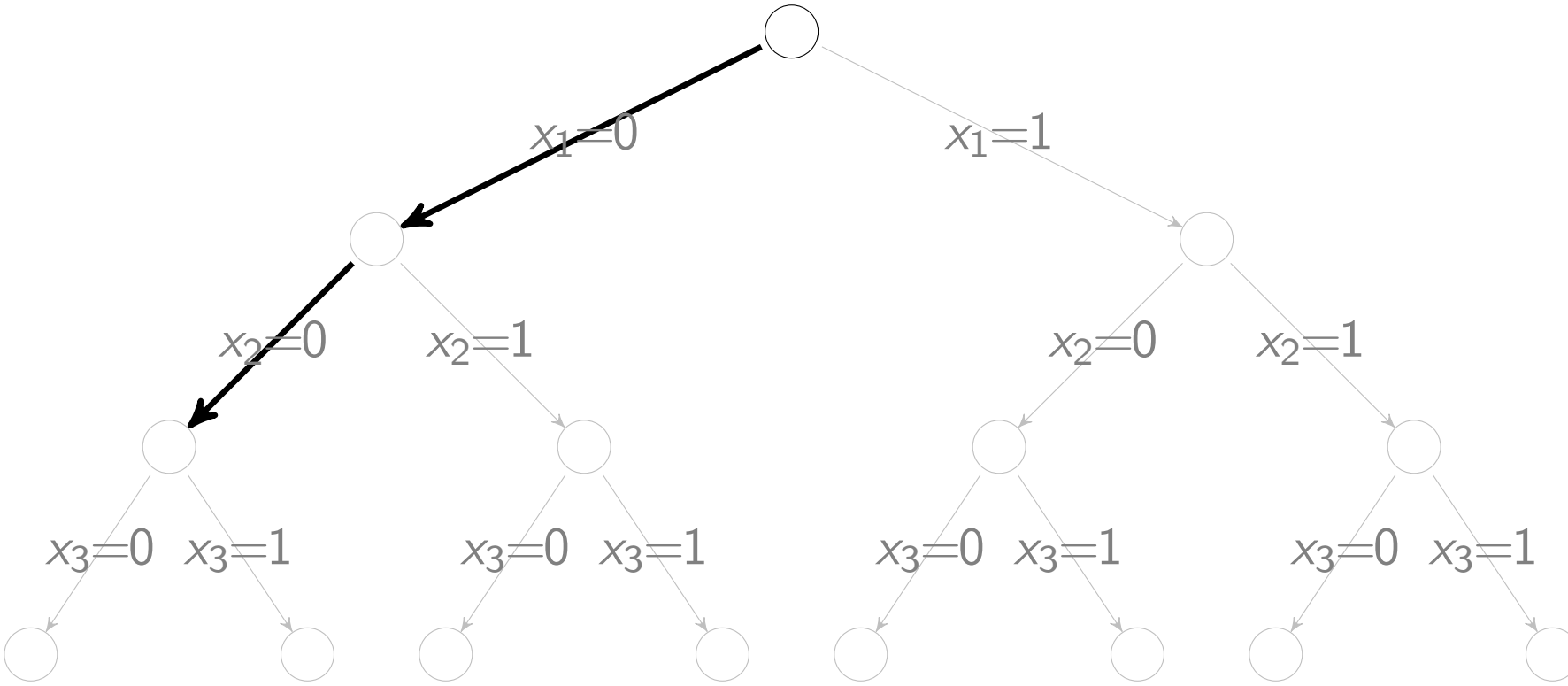
$x_1 = 0$ $x_2 = ?$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



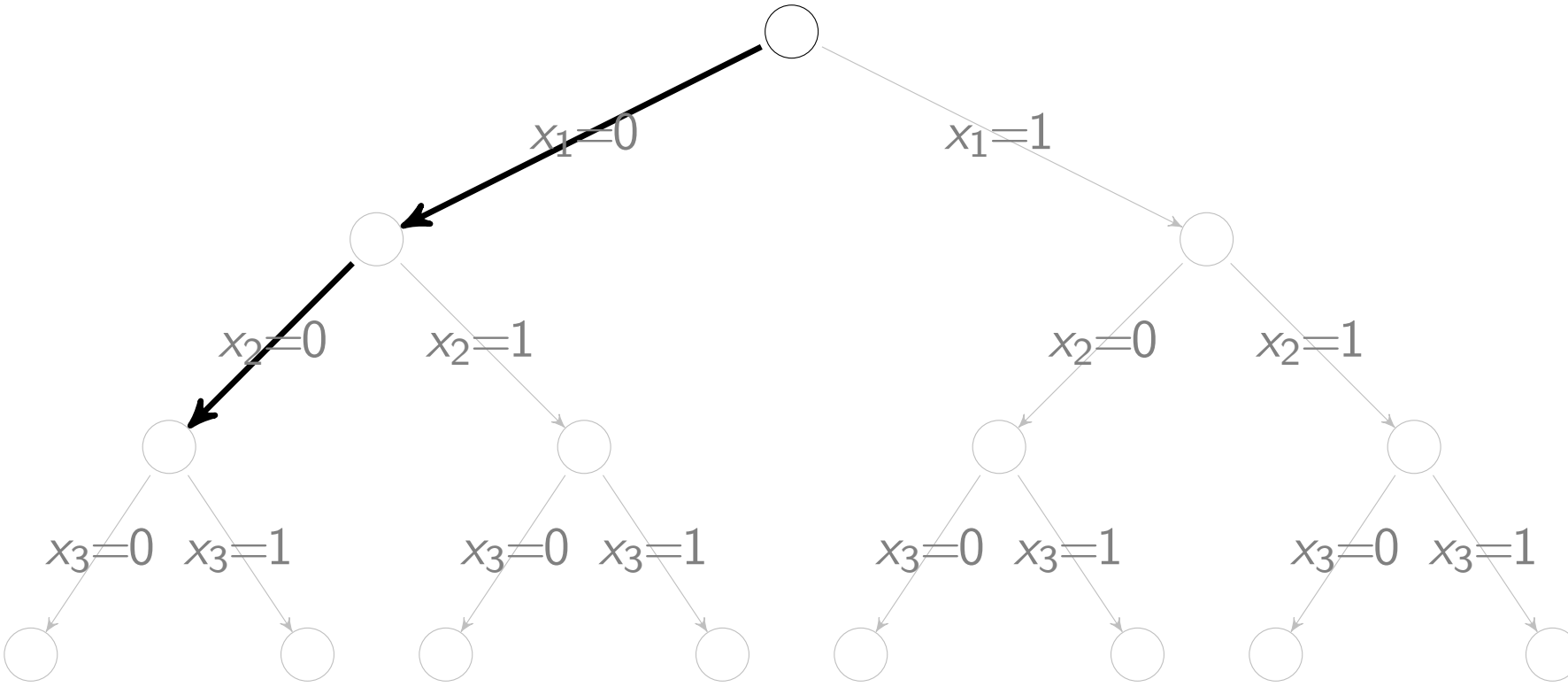
$x_1 = 0$ $x_2 = ?$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



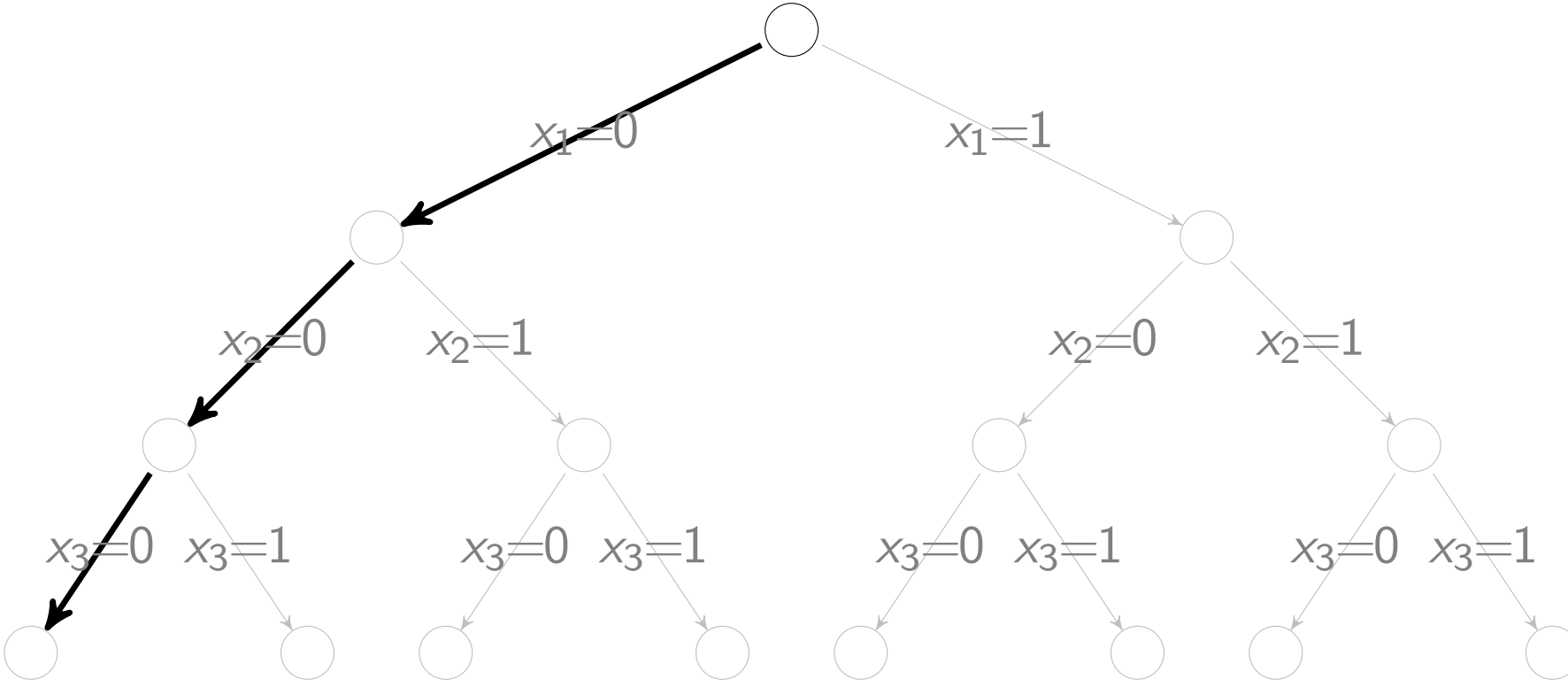
$x_1 = 0$ $x_2 = 0$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg \boldsymbol{x_2}) \bigwedge (x_1 \vee \neg x_3)$$



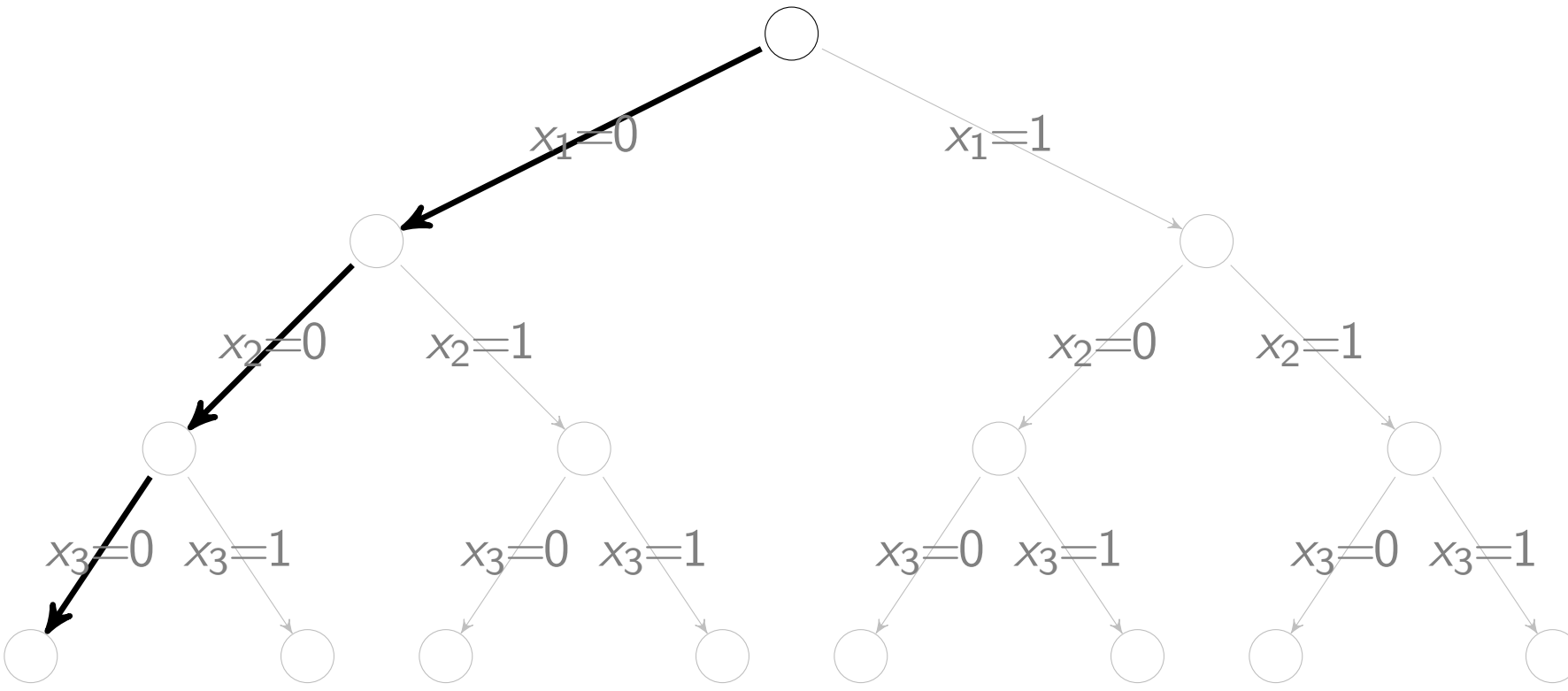
$x_1 = 0$ $x_2 = 0$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg \boldsymbol{x_2}) \bigwedge (x_1 \vee \neg x_3)$$



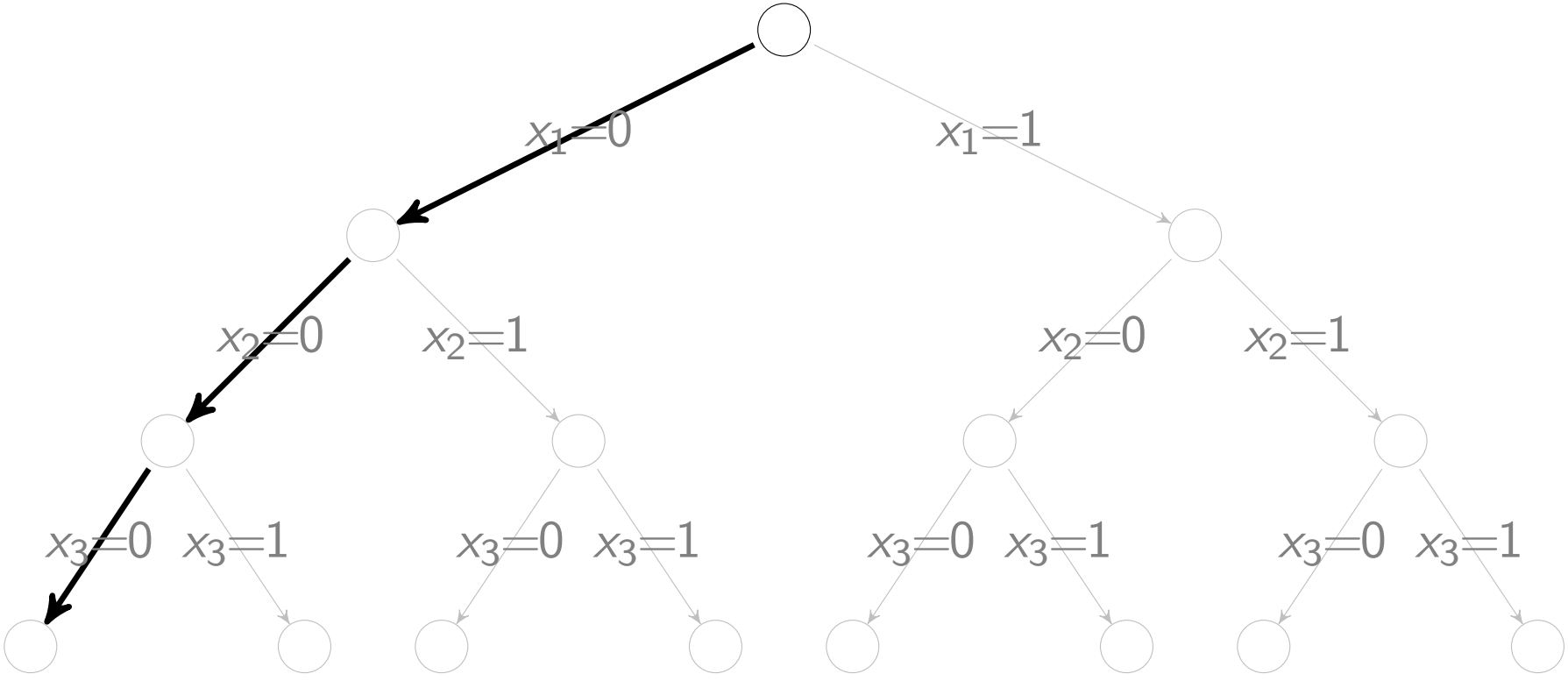
$x_1 = 0$ $x_2 = 0$ $x_3 = 0$

$(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee \neg \boldsymbol{x_2}) \wedge (x_1 \vee \neg \boldsymbol{x_3})$



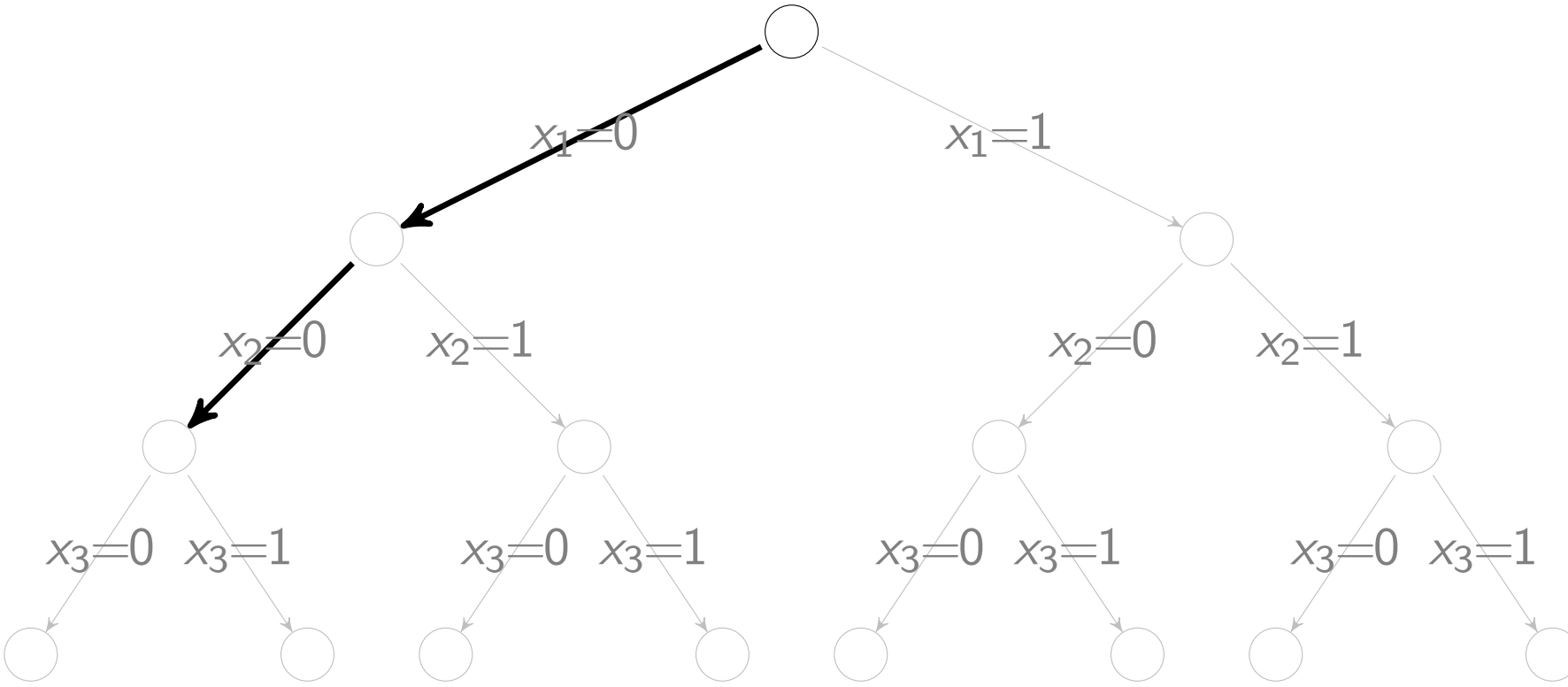
$x_1 = 0$ $x_2 = 0$ $x_3 = 0$

$(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee \neg x_2) \wedge (x_1 \vee \neg x_3)$



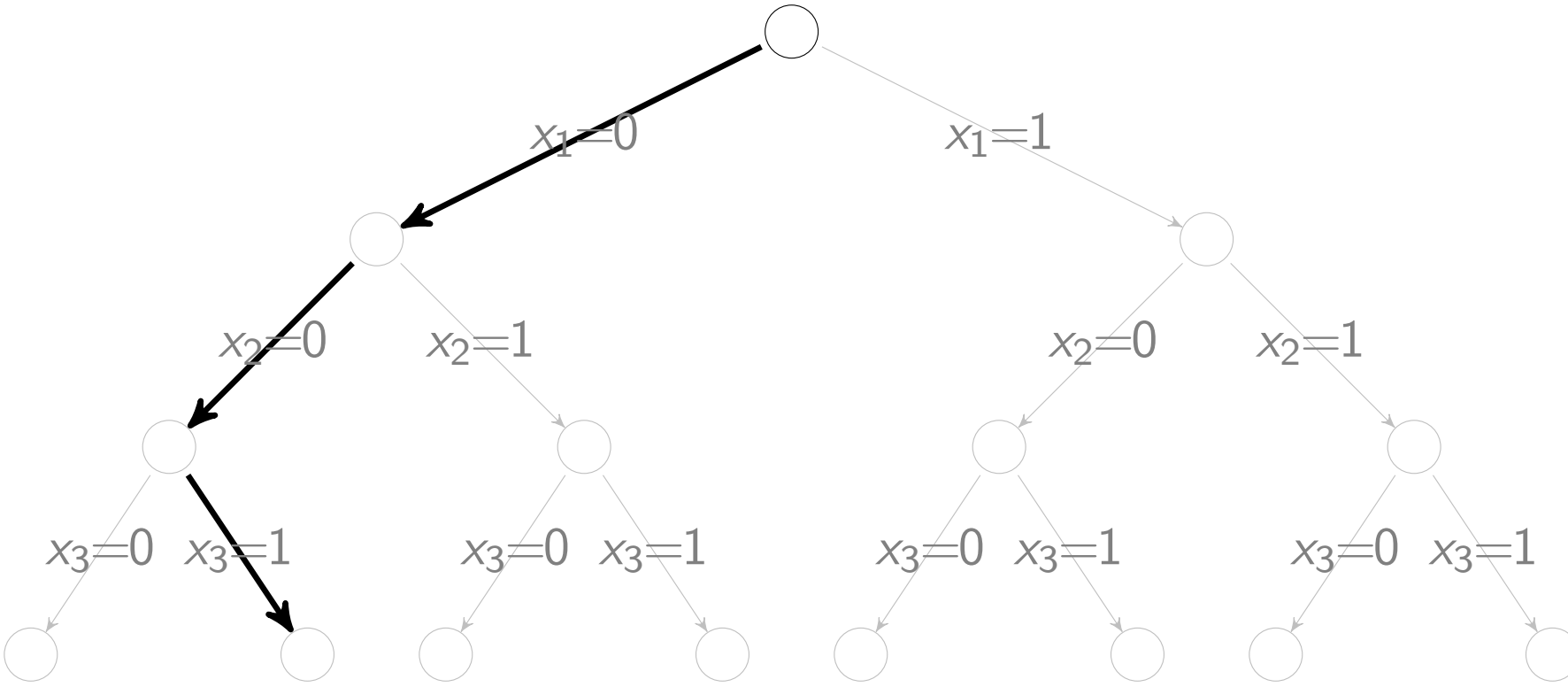
$x_1 = 0$ $x_2 = 0$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg \boldsymbol{x_2}) \bigwedge (x_1 \vee \neg x_3)$$



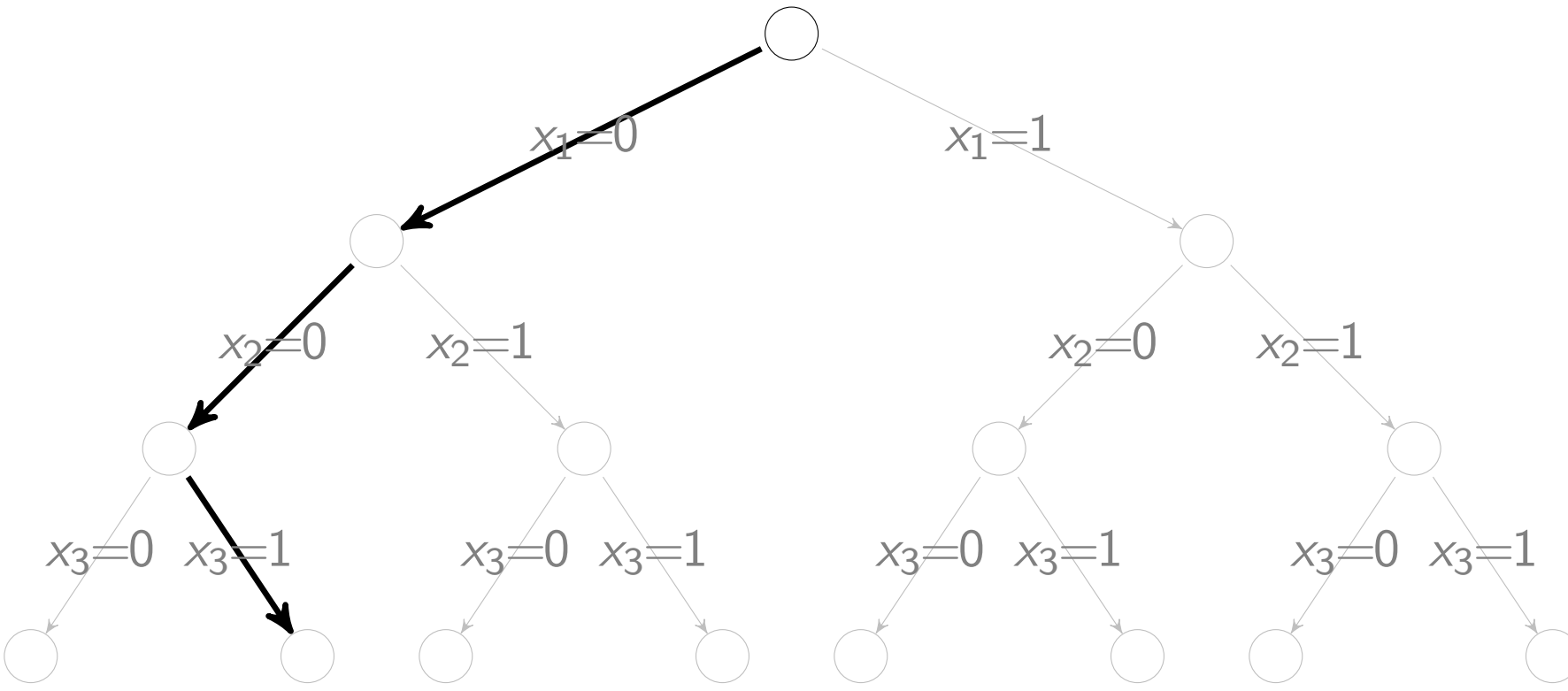
$x_1 = 0$ $x_2 = 0$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg \boldsymbol{x_2}) \bigwedge (x_1 \vee \neg x_3)$$

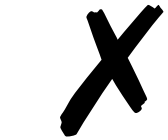


$x_1 = 0$ $x_2 = 0$ $x_3 = 1$

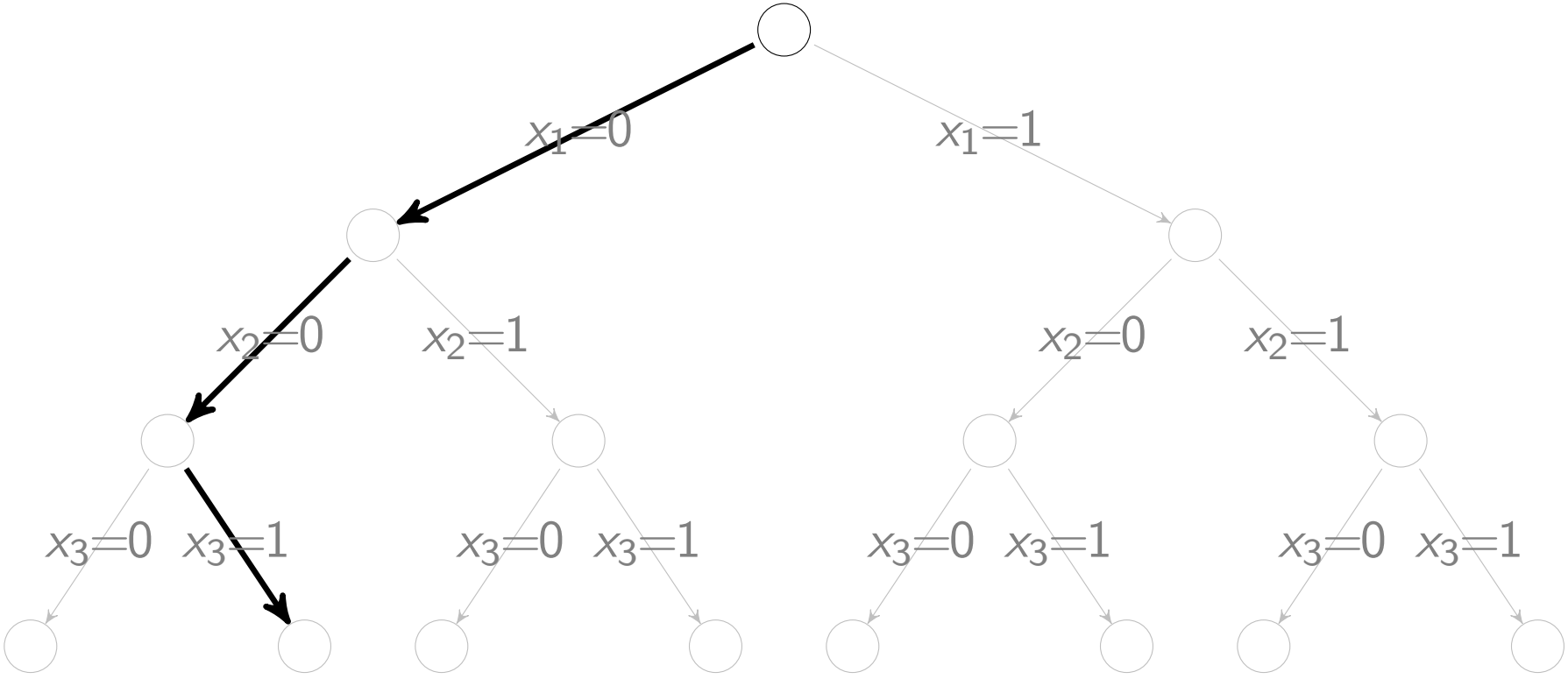
$(x_1 \vee x_2 \vee \boldsymbol{x_3}) \bigwedge (\boldsymbol{x_3} \vee \neg \boldsymbol{x_2}) \bigwedge (x_1 \vee \neg x_3)$



$x_1 = 0$ $x_2 = 0$ $x_3 = 1$

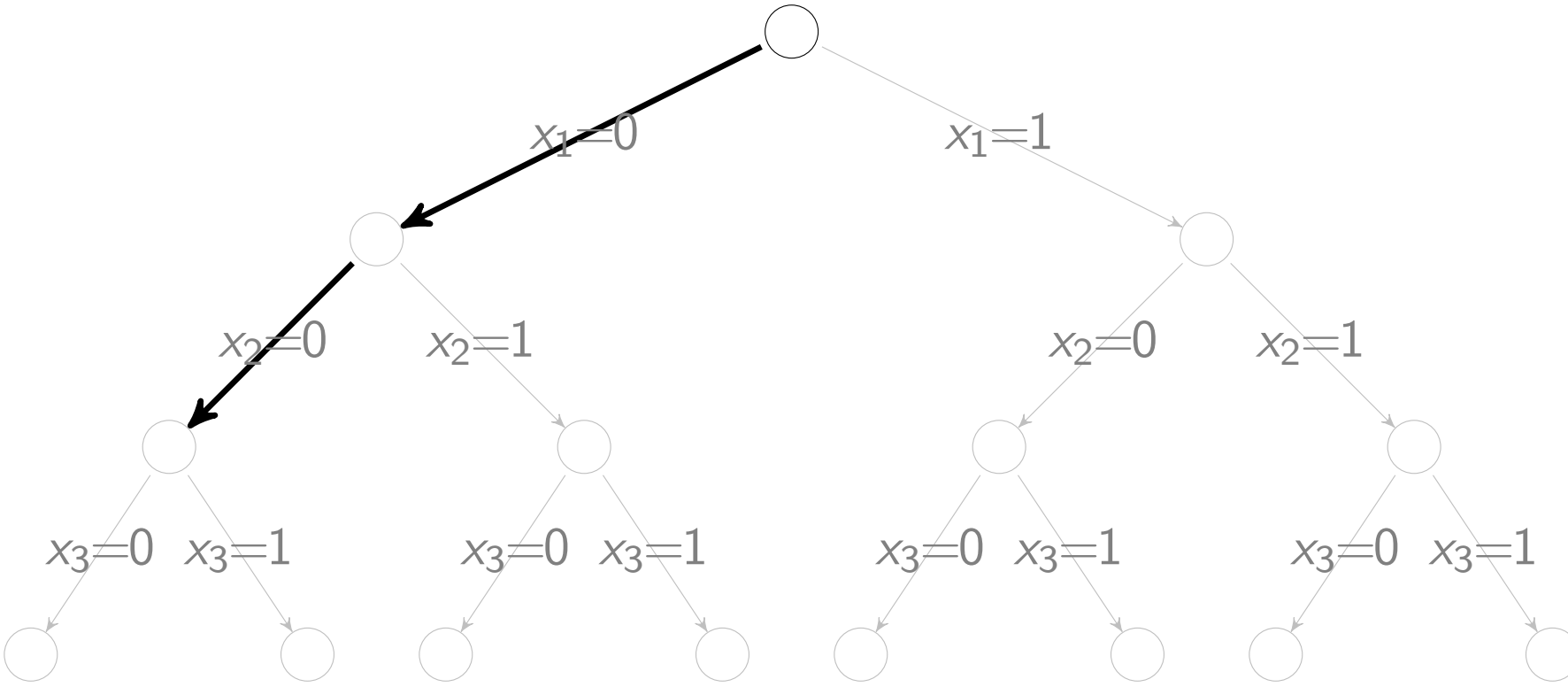


$(x_1 \vee x_2 \vee \boldsymbol{x_3}) \bigwedge (\boldsymbol{x_3} \vee \neg \boldsymbol{x_2}) \bigwedge (x_1 \vee \neg x_3)$



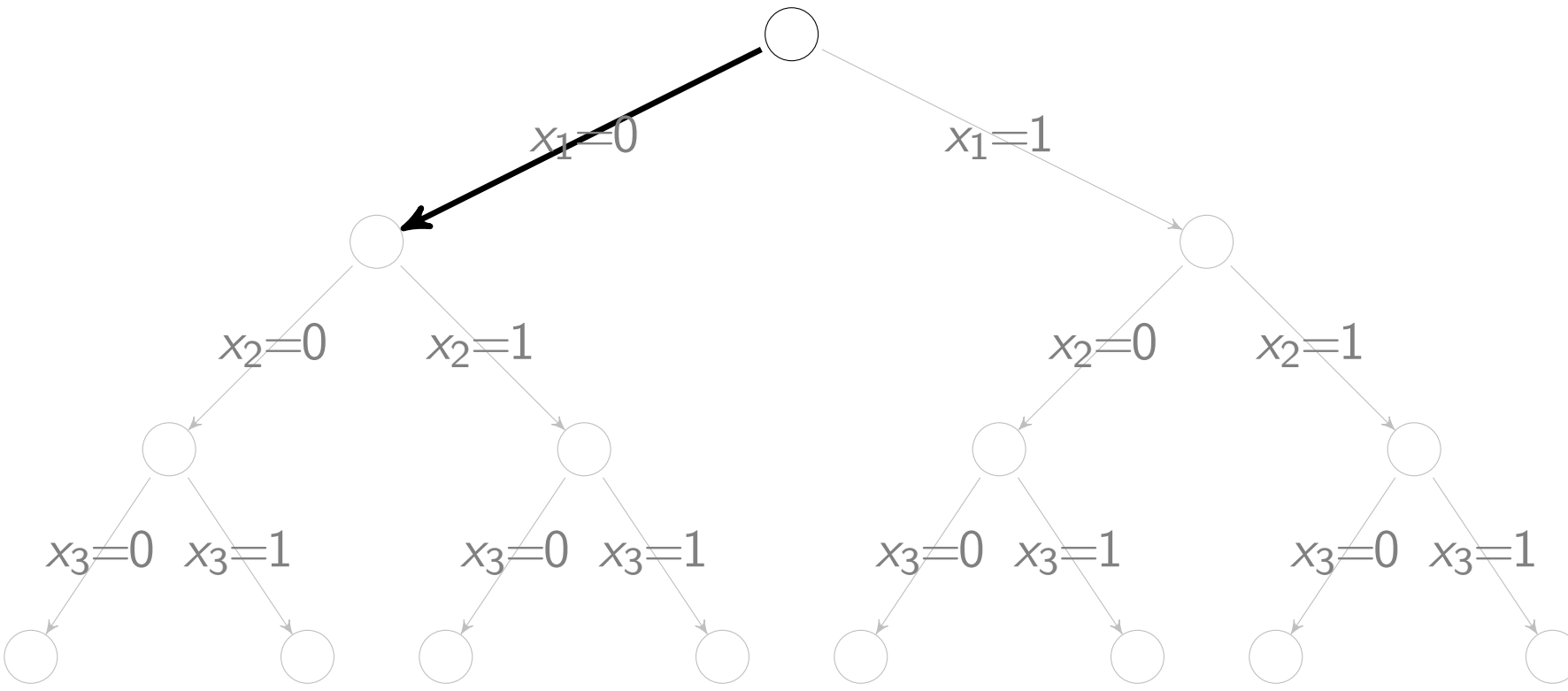
$x_1 = 0$ $x_2 = 0$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg \boldsymbol{x_2}) \bigwedge (x_1 \vee \neg x_3)$$



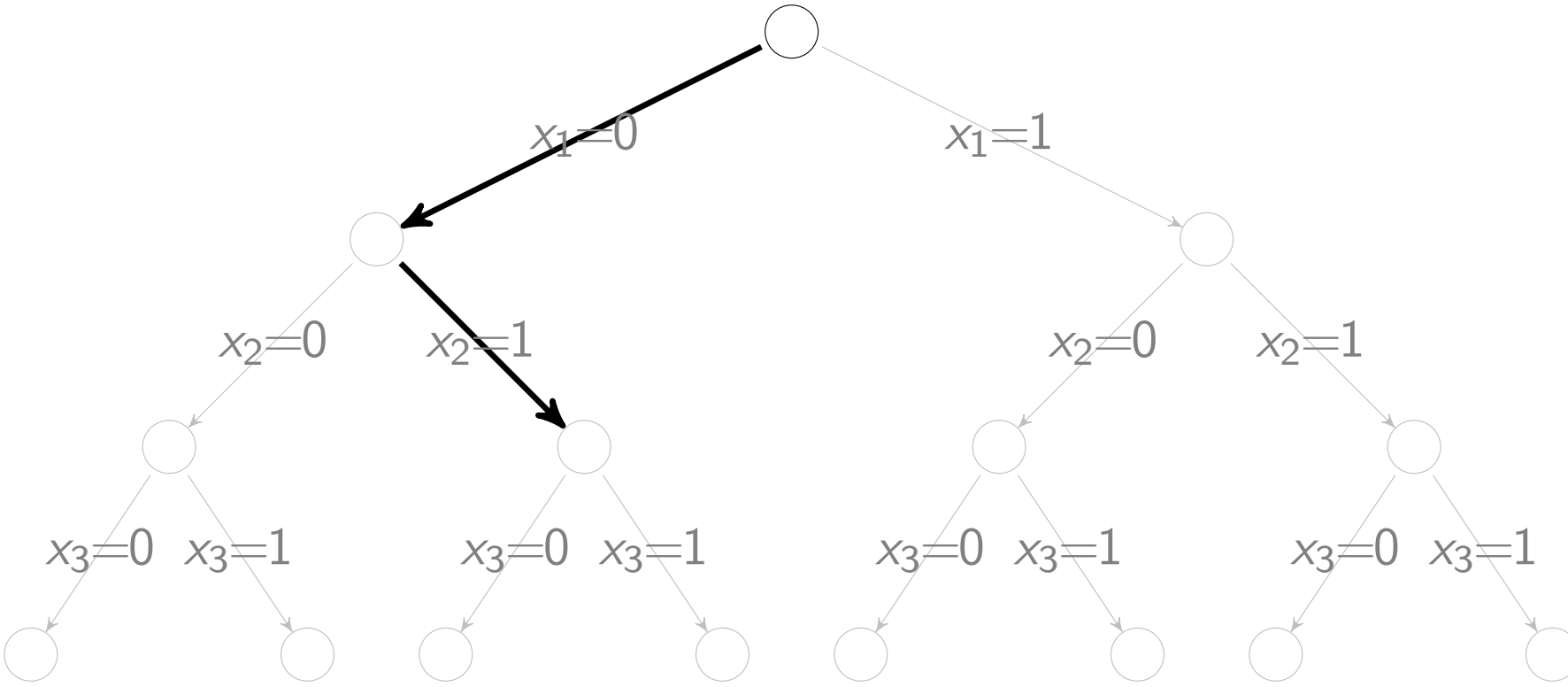
$x_1 = 0$ $x_2 = ?$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



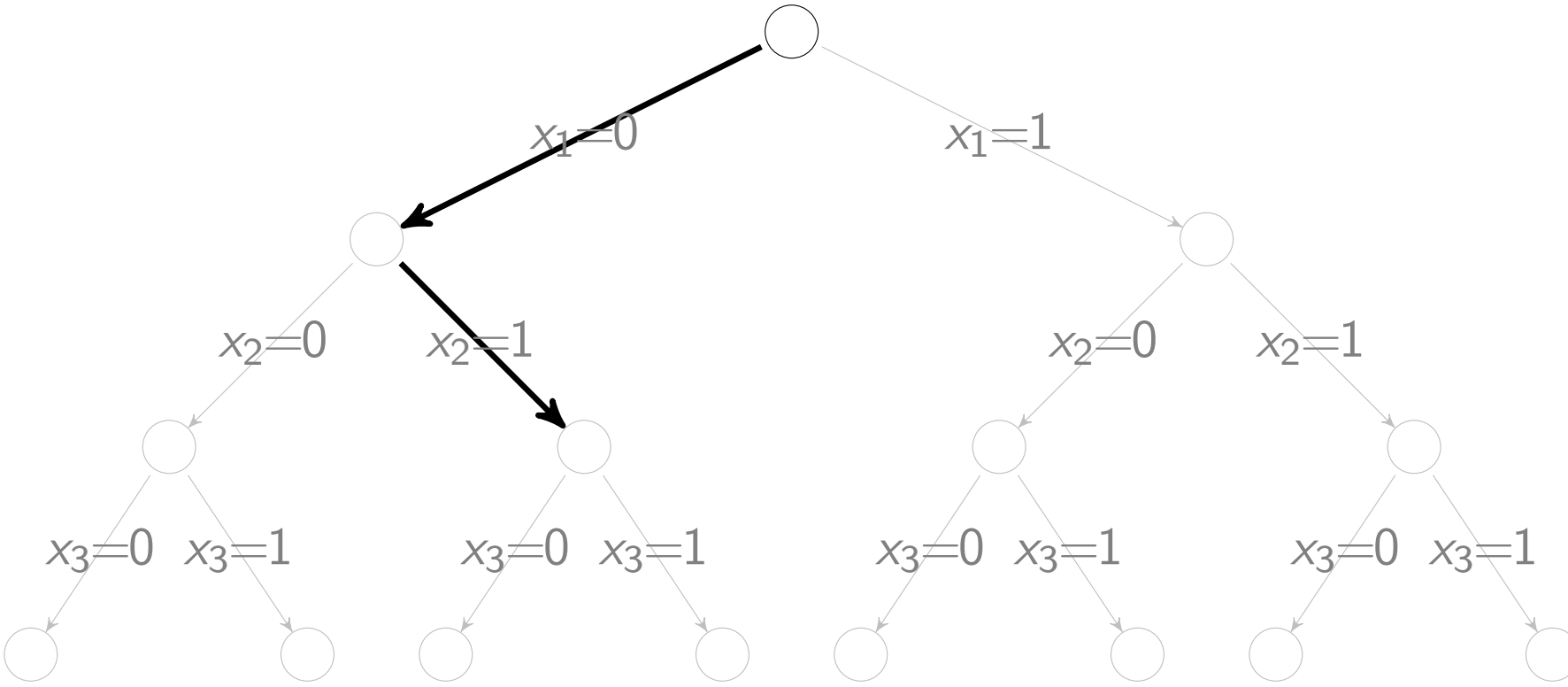
$x_1 = 0$ $x_2 = ?$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee \neg x_2) \wedge (x_1 \vee \neg x_3)$$



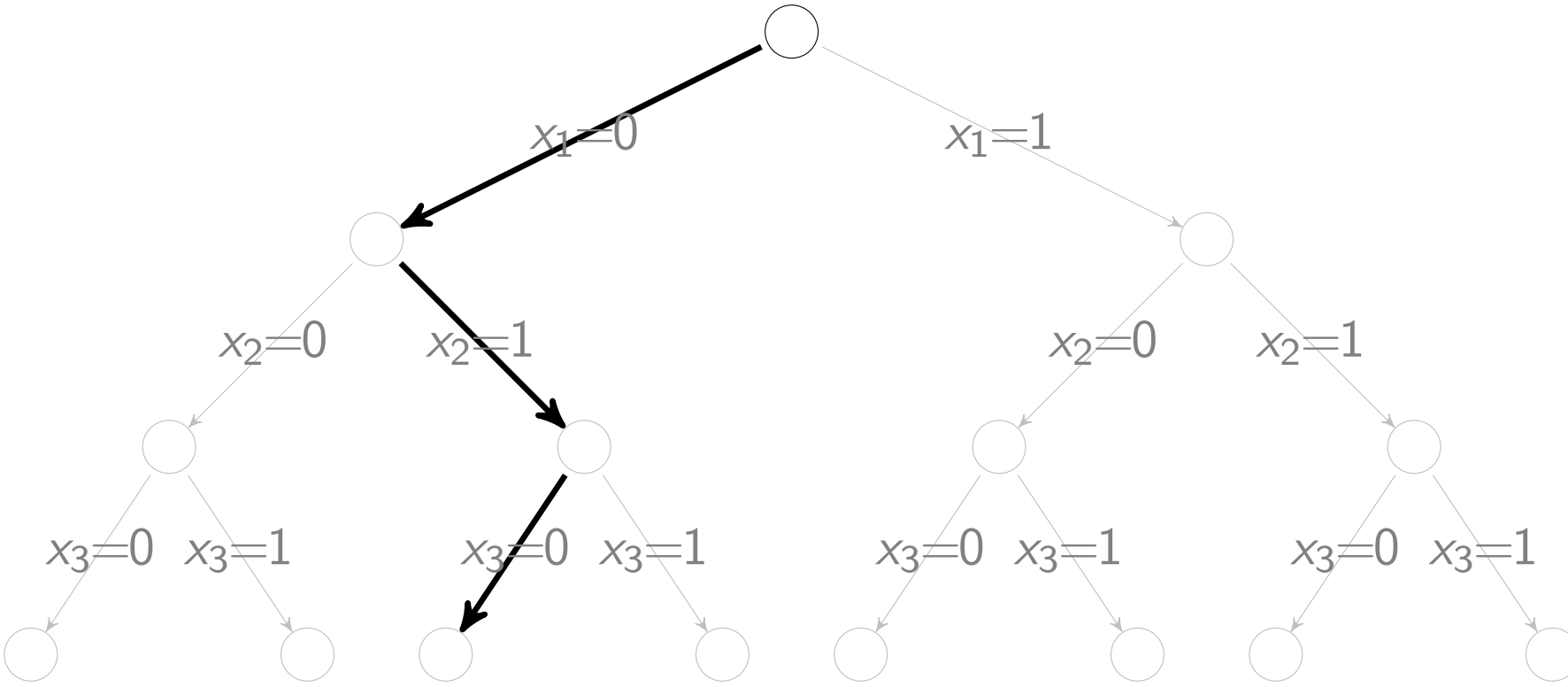
$x_1 = 0$ $x_2 = 1$ $x_3 = ?$

$$(x_1 \vee \boldsymbol{x_2} \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



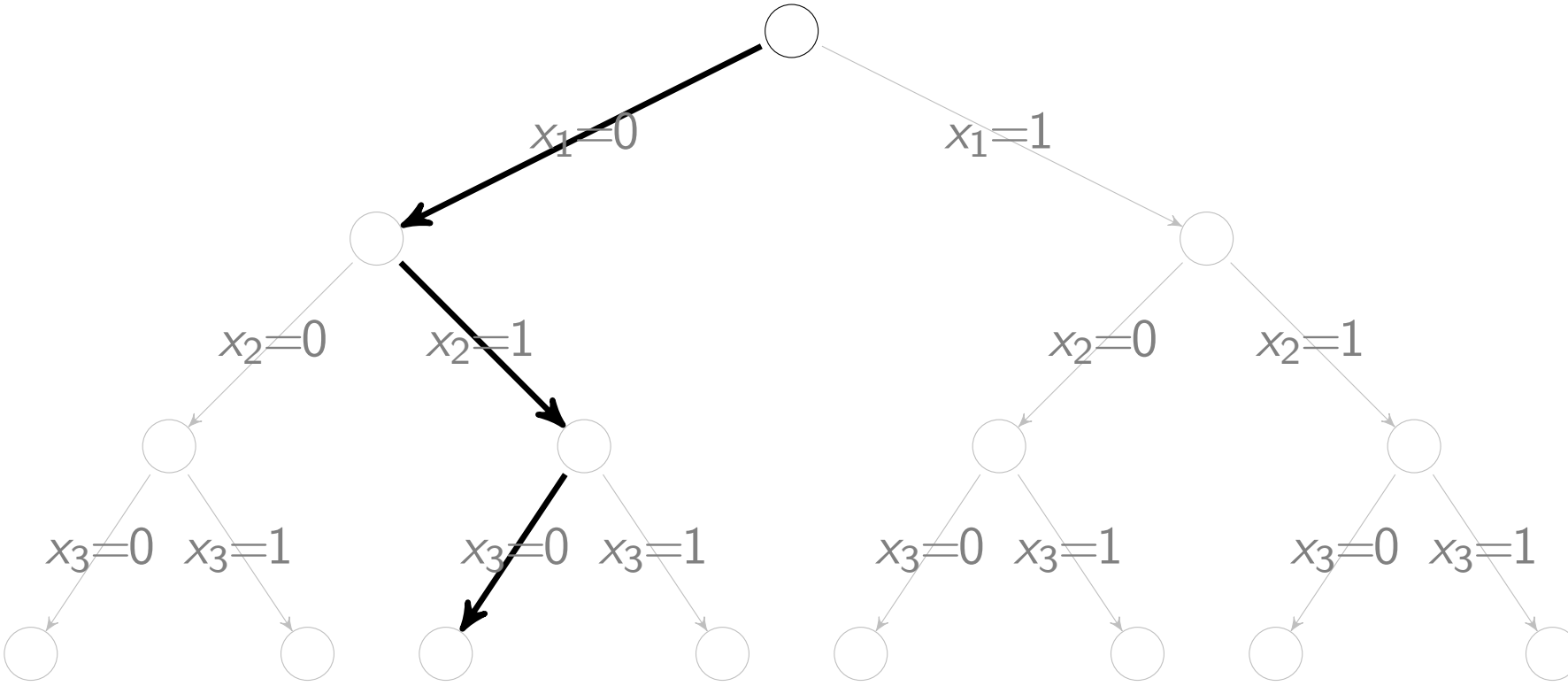
$x_1 = 0$ $x_2 = 1$ $x_3 = ?$

$$(x_1 \vee \boldsymbol{x_2} \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



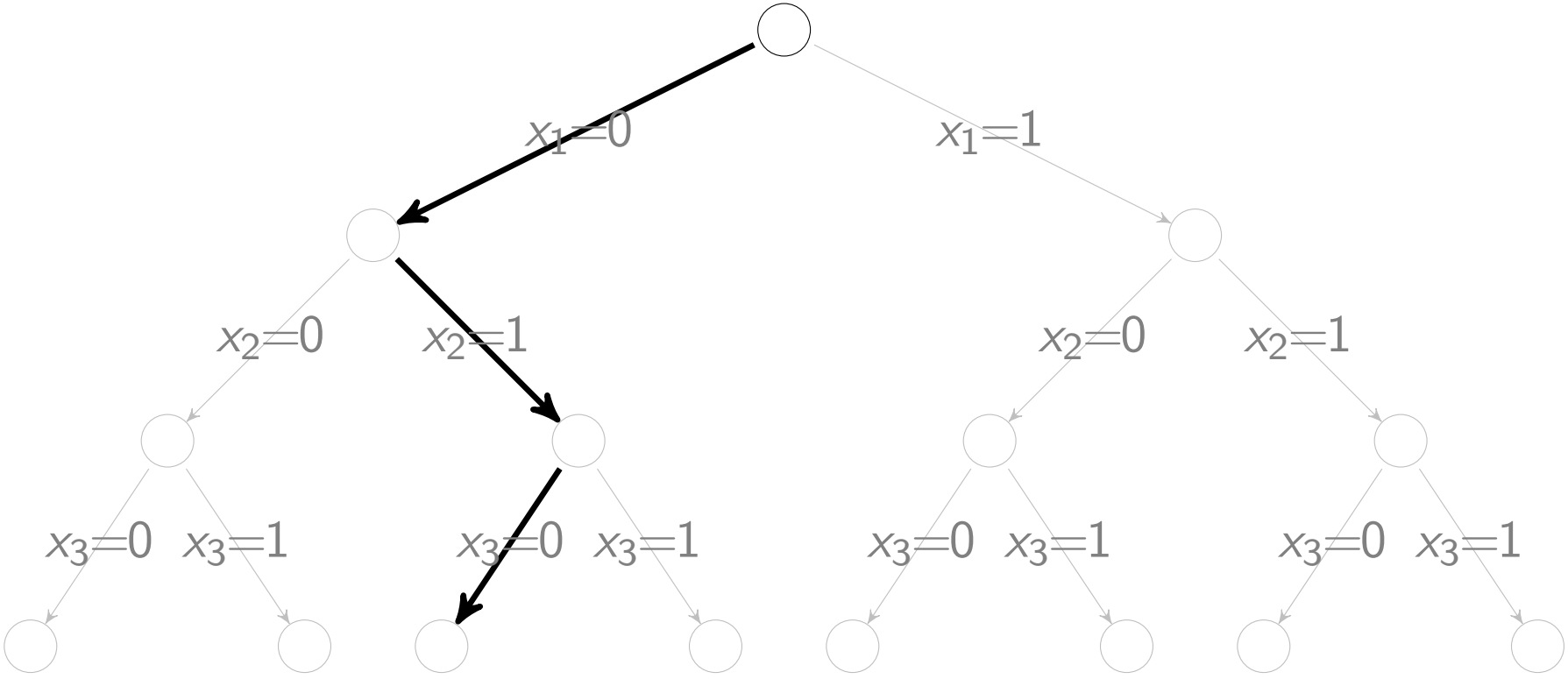
$x_1 = 0$ $x_2 = 1$ $x_3 = 0$

$(x_1 \vee \boldsymbol{x_2} \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg \boldsymbol{x_3})$



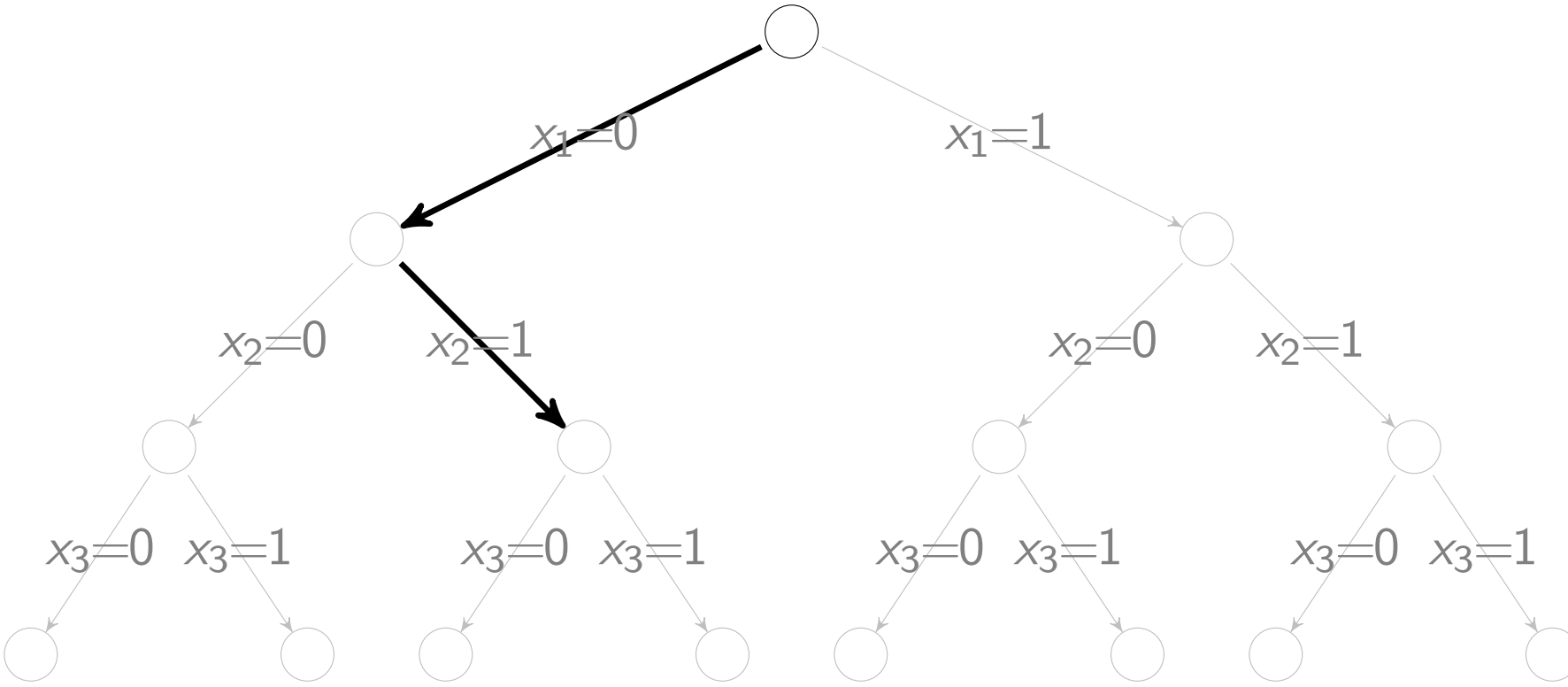
$x_1 = 0$ $x_2 = 1$ $x_3 = 0$

$(x_1 \vee \boldsymbol{x_2} \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg \boldsymbol{x_3})$



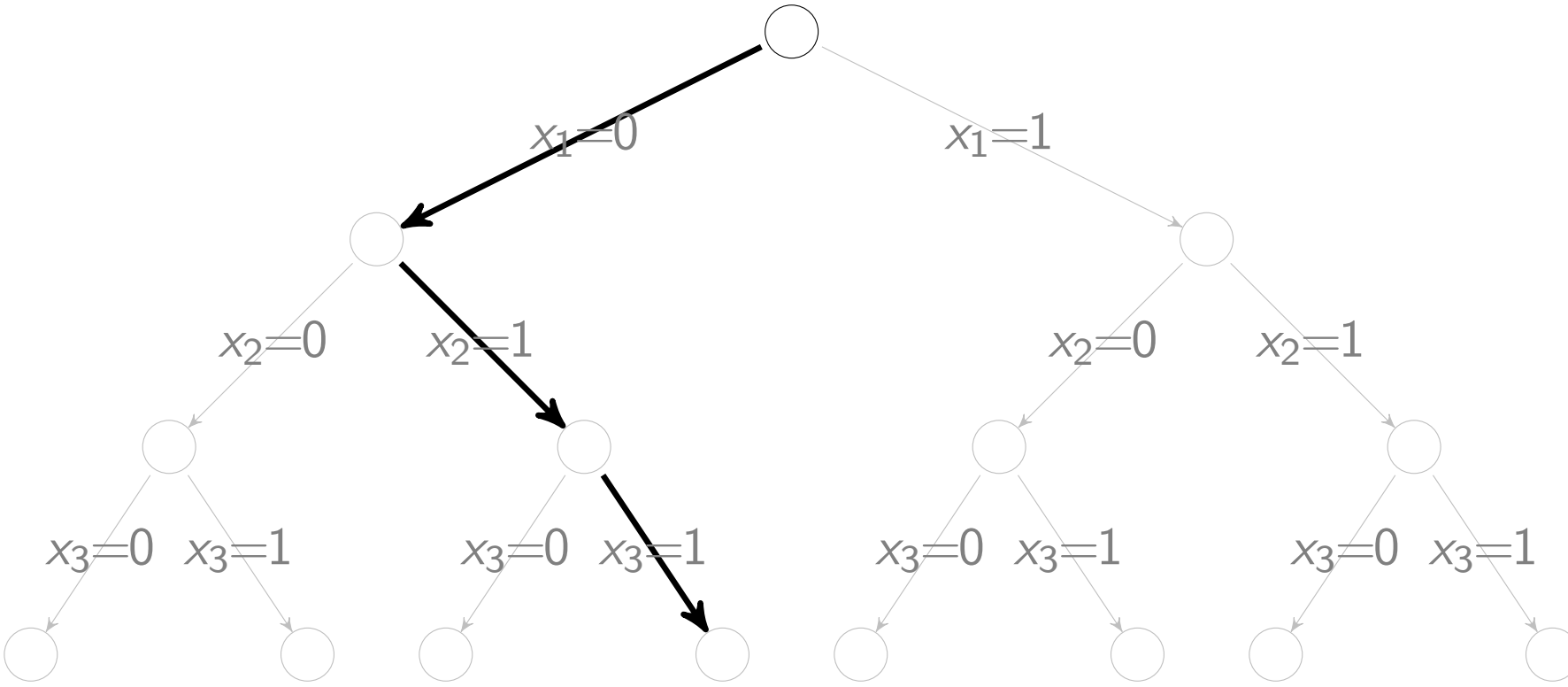
$x_1 = 0$ $x_2 = 1$ $x_3 = ?$

$$(x_1 \vee \boldsymbol{x_2} \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



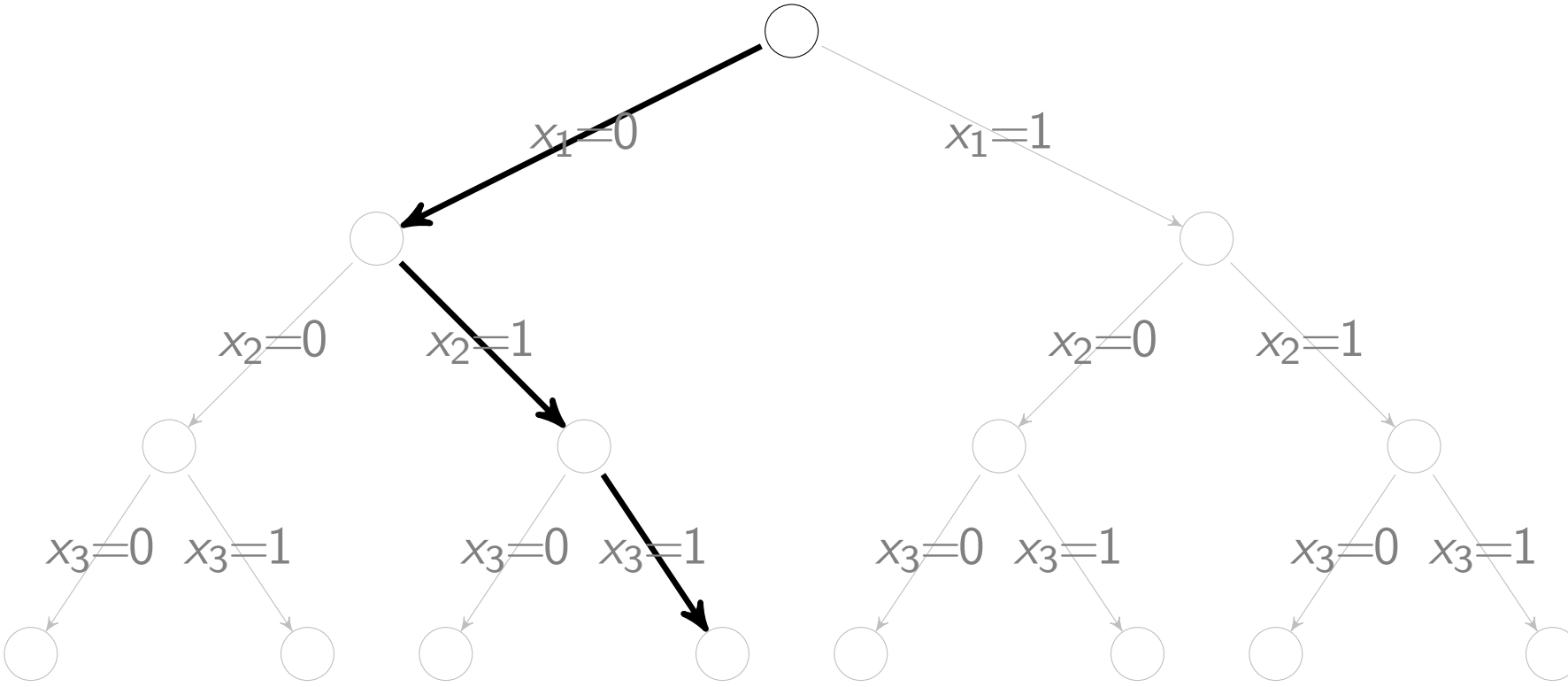
$x_1 = 0$ $x_2 = 1$ $x_3 = ?$

$$(x_1 \vee \boldsymbol{x_2} \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



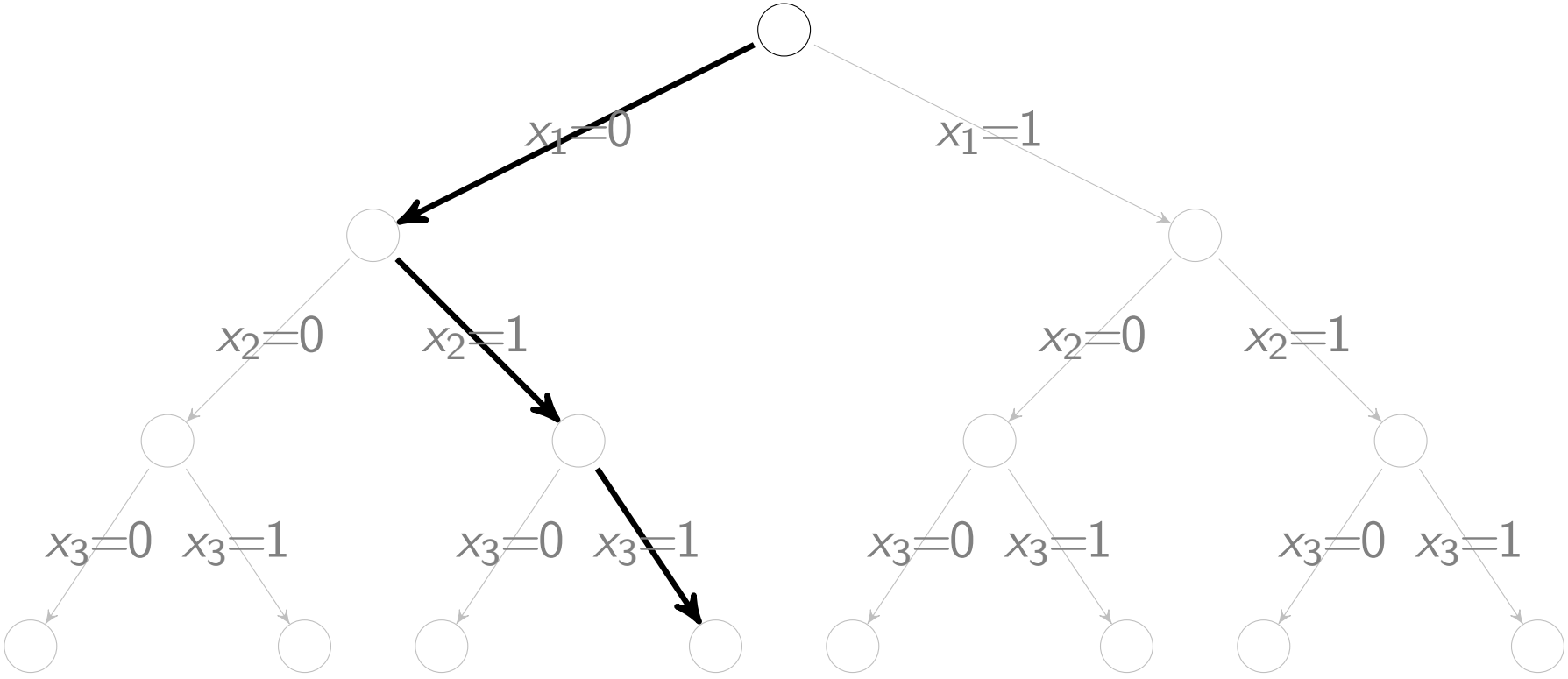
$x_1 = 0$ $x_2 = 1$ $x_3 = 1$

$(x_1 \vee \boldsymbol{x_2} \vee \boldsymbol{x_3}) \bigwedge (\boldsymbol{x_3} \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$



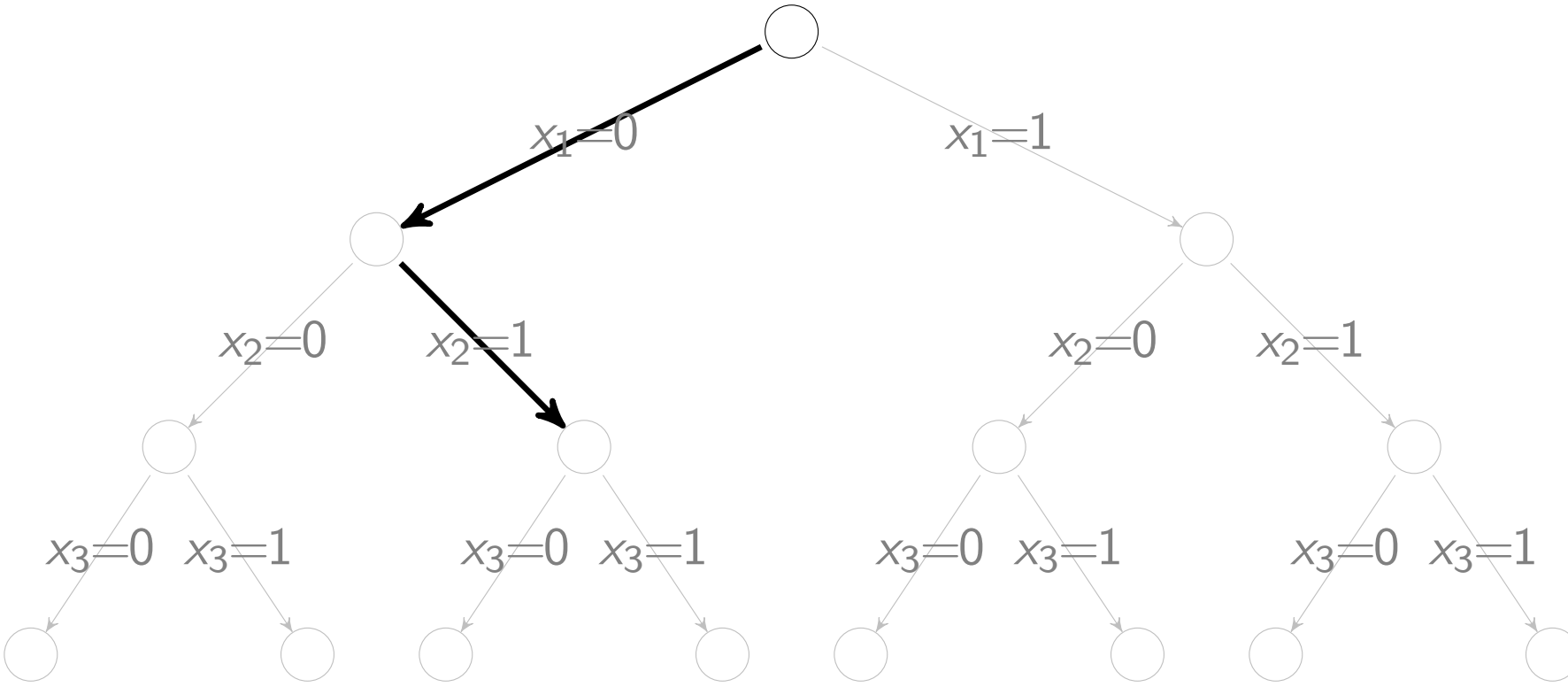
$x_1 = 0$ $x_2 = 1$ $x_3 = 1$

$(x_1 \vee \boldsymbol{x_2} \vee \boldsymbol{x_3}) \bigwedge (\boldsymbol{x_3} \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$



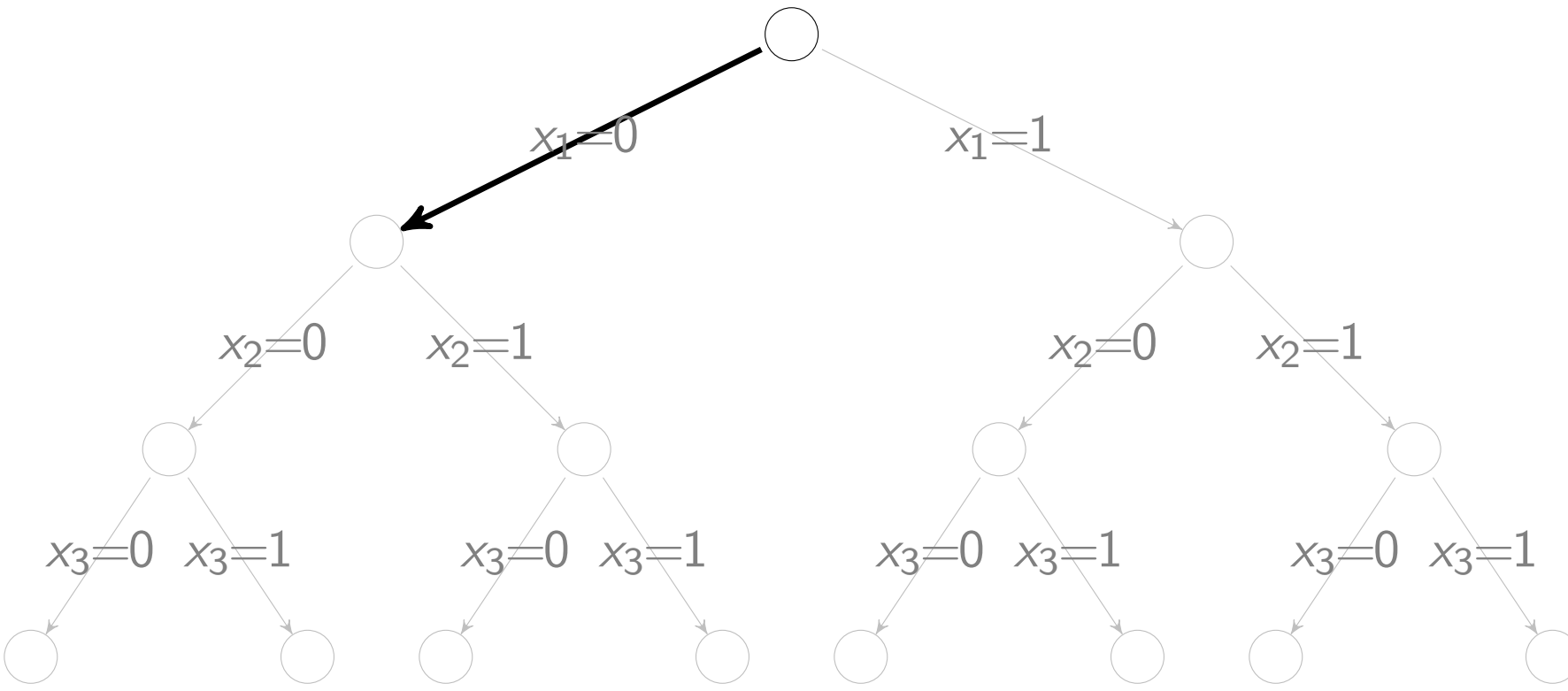
$x_1 = 0$ $x_2 = 1$ $x_3 = ?$

$$(x_1 \vee \boldsymbol{x_2} \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



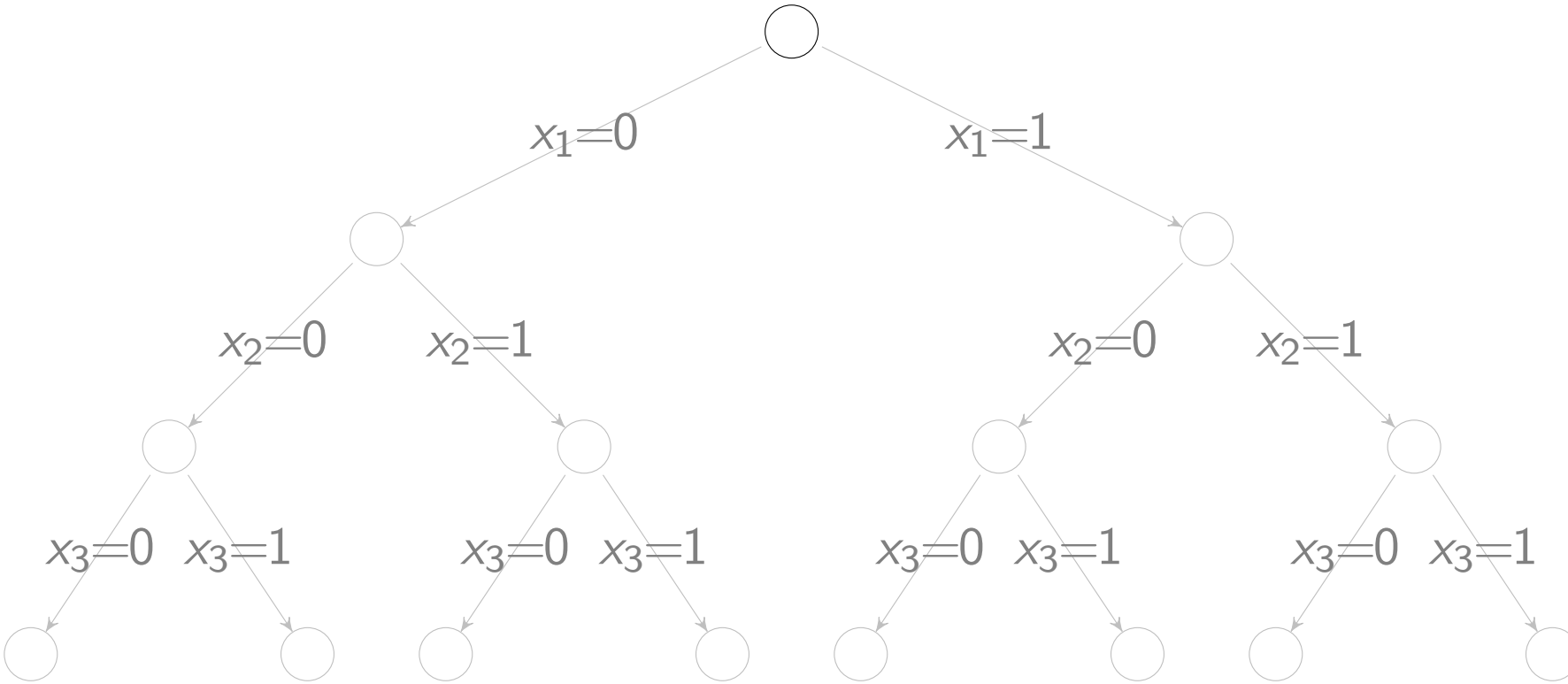
$x_1 = 0$ $x_2 = ?$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



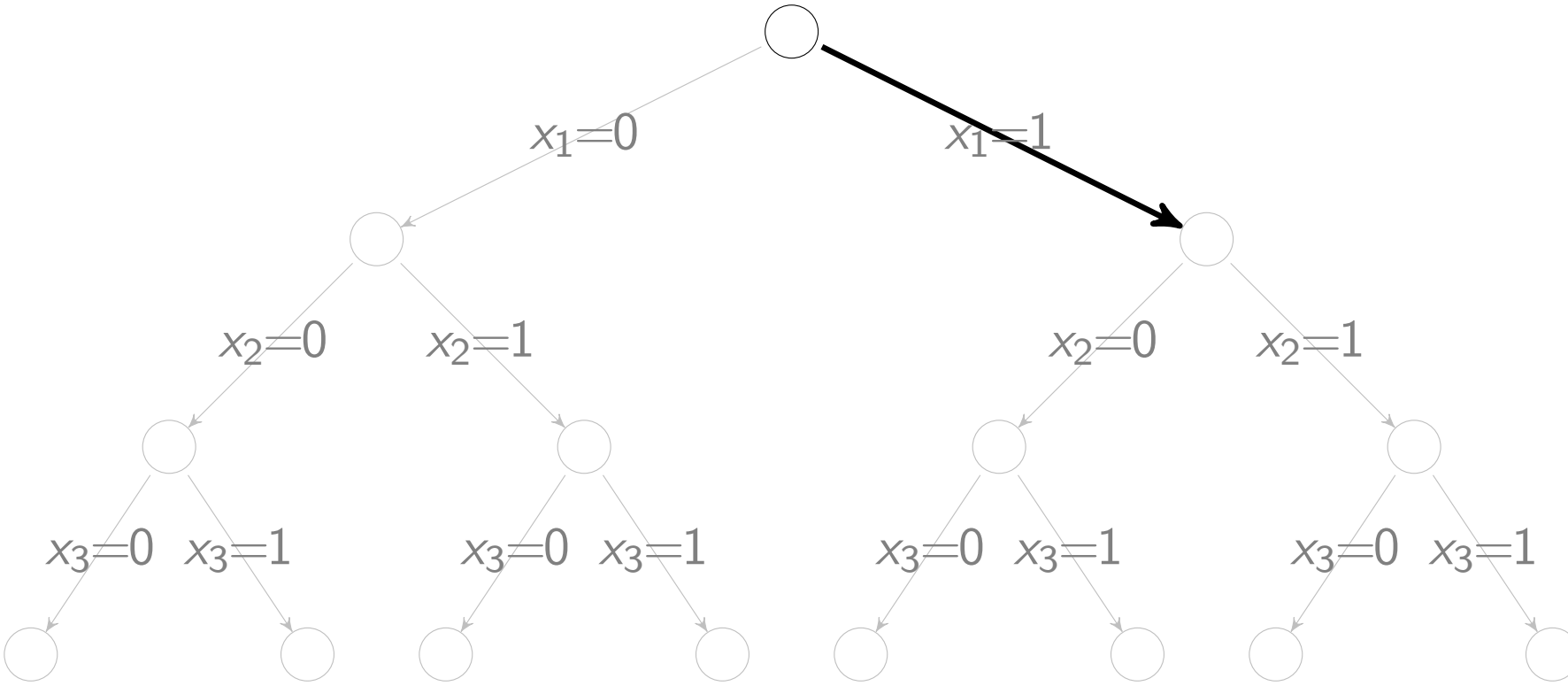
$x_1 = ?$ $x_2 = ?$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



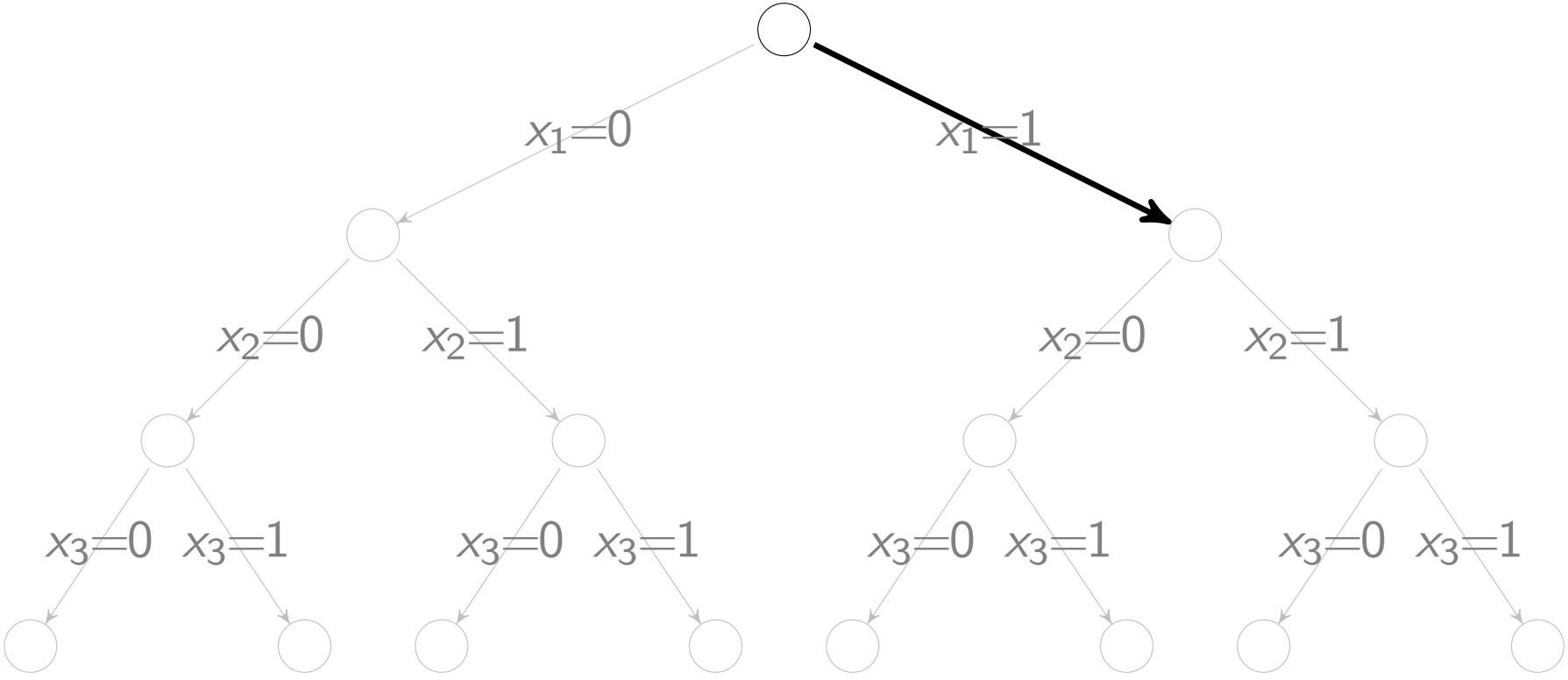
$x_1 = ?$ $x_2 = ?$ $x_3 = ?$

$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



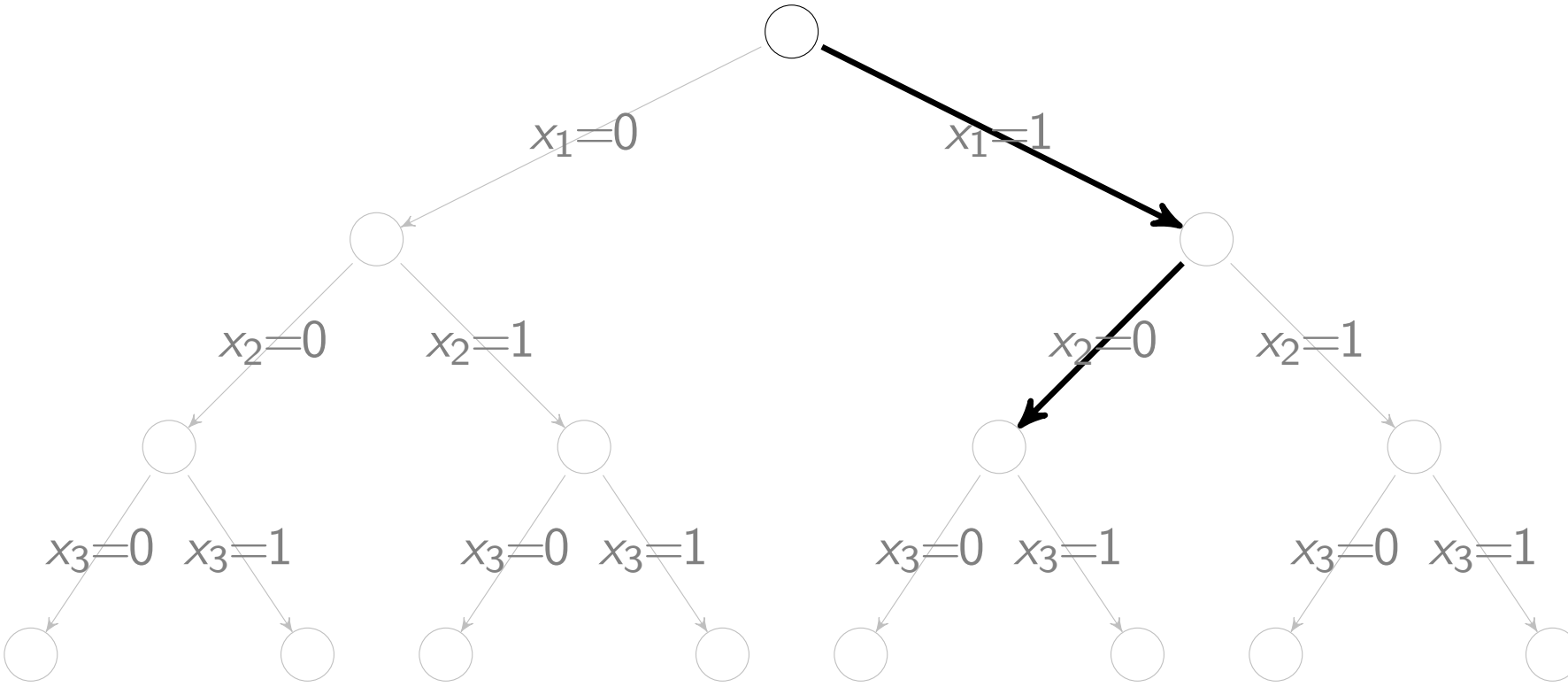
$x_1 = 1$ $\quad$ $x_2 = ?$ $\quad$ $x_3 = ?$

$$(\boldsymbol{x_1} \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (\boldsymbol{x_1} \vee \neg x_3)$$



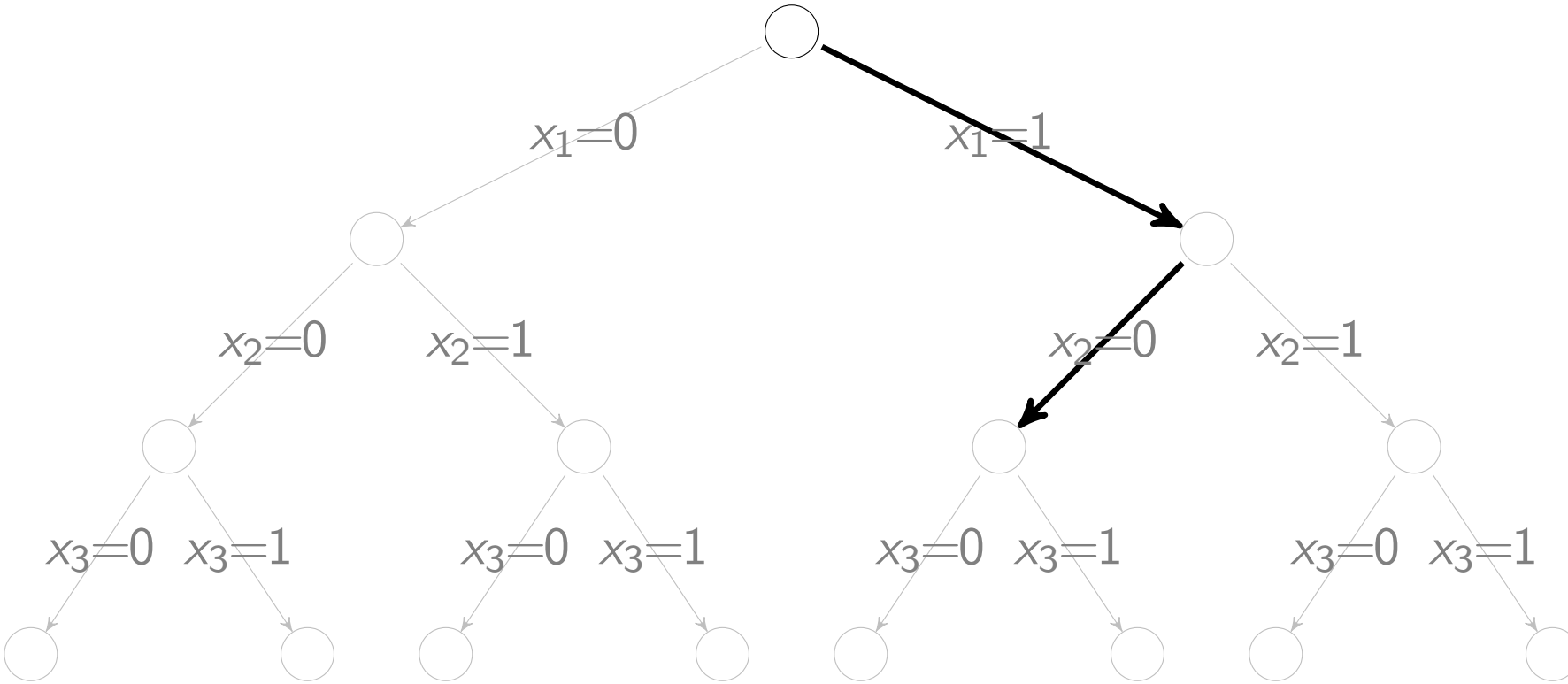
$x_1 = 1$ $x_2 = ?$ $x_3 = ?$

$$(\boldsymbol{x_1} \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (\boldsymbol{x_1} \vee \neg x_3)$$



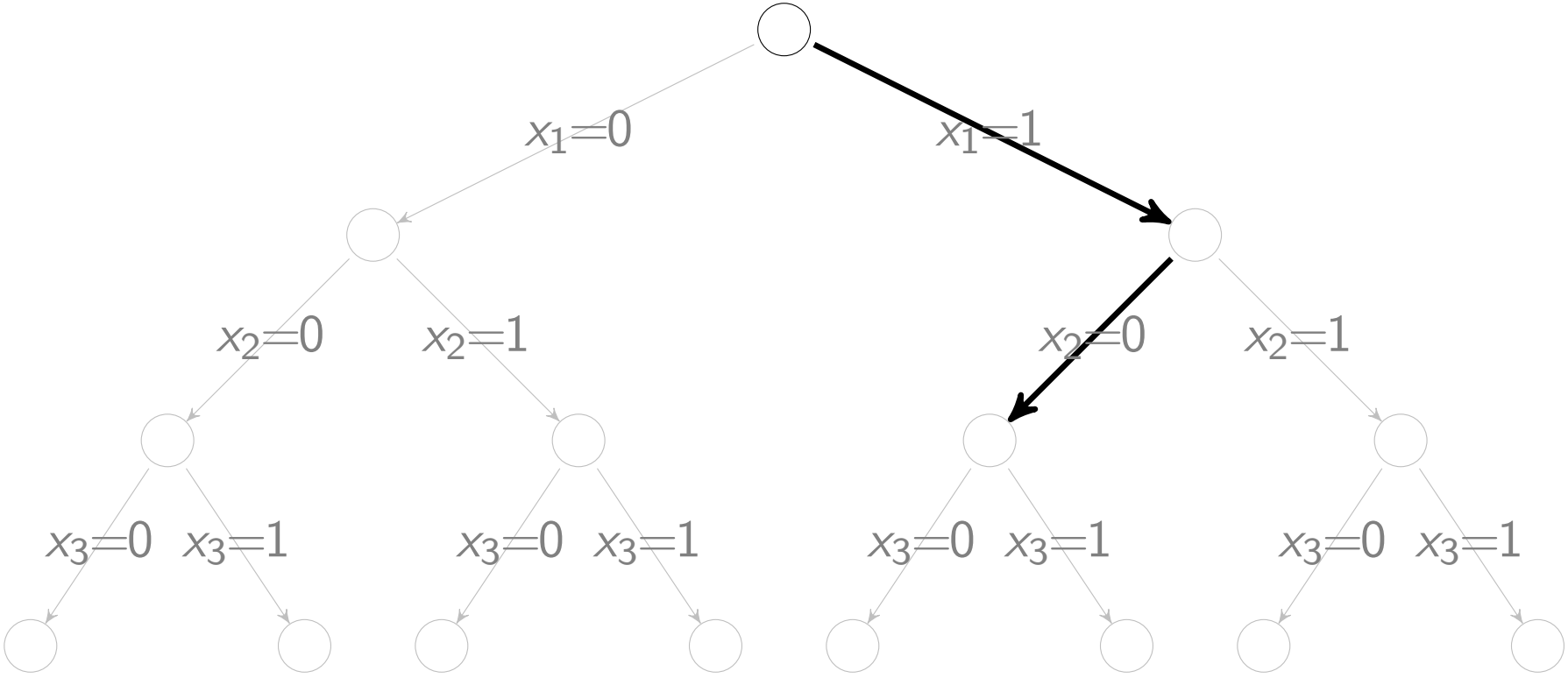
$x_1 = 1$ $x_2 = 0$ $x_3 = ?$

$$(\boldsymbol{x_1} \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg \boldsymbol{x_2}) \bigwedge (\boldsymbol{x_1} \vee \neg x_3)$$



$x_1 = 1$ $x_2 = 0$ $x_3 = ?$

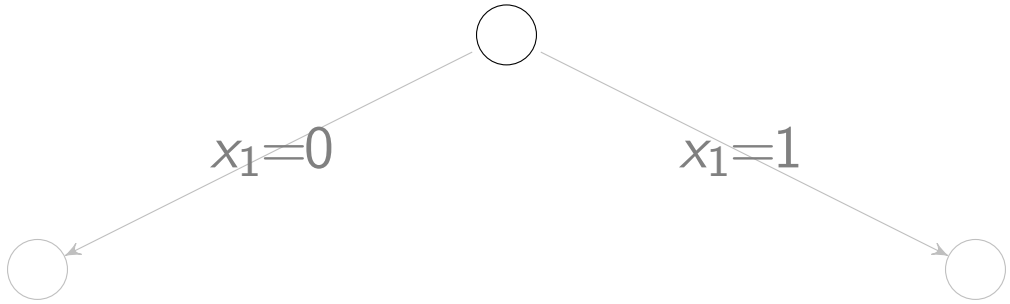
$$(\boldsymbol{x_1} \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg \boldsymbol{x_2}) \bigwedge (\boldsymbol{x_1} \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = ?$ $\quad$ $x_2 = ?$ $\quad$ $x_3 = ?$

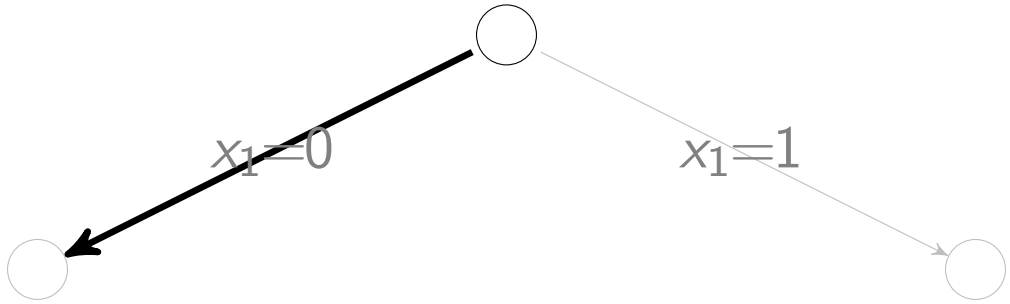
$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = ?$ $\quad$ $x_2 = ?$ $\quad$ $x_3 = ?$

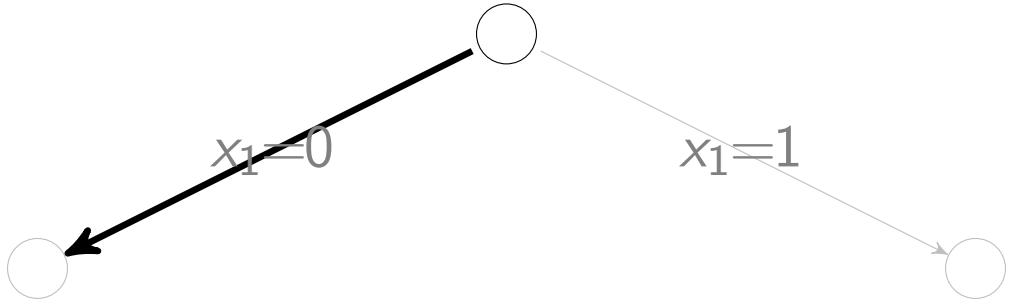
$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 0$ $\qquad$ $x_2 = ?$ $\qquad$ $x_3 = ?$

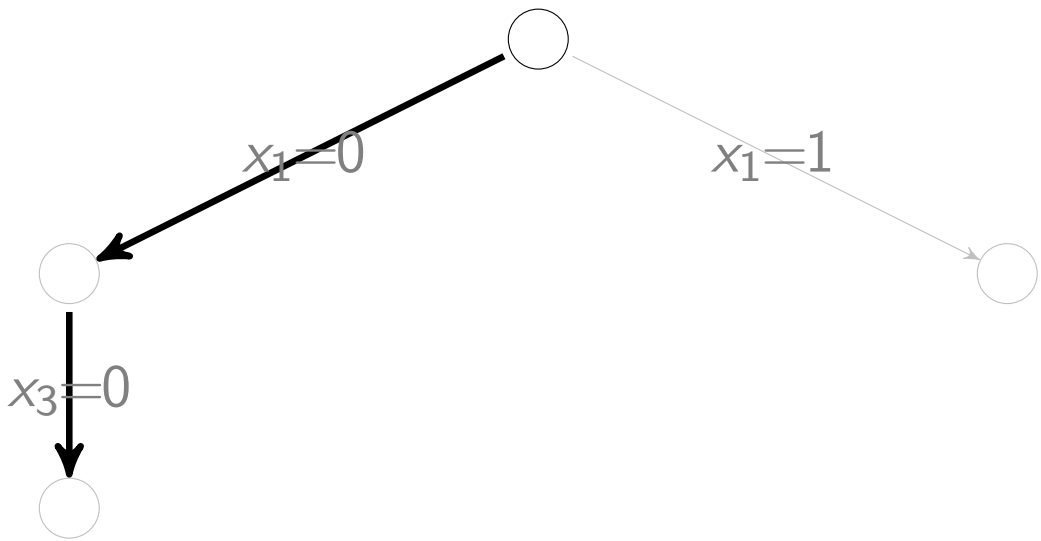
$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 0$ $\qquad$ $x_2 = ?$ $\qquad$ $x_3 = ?$

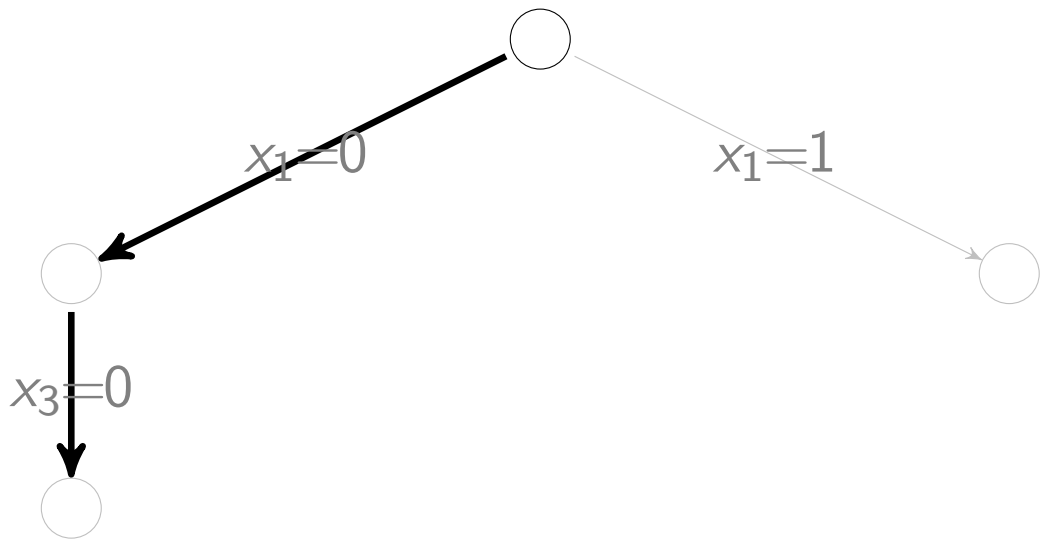
$$(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee \neg x_2) \wedge (x_1 \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 0$ $\qquad$ $x_2 = ?$ $\qquad$ $x_3 = 0$

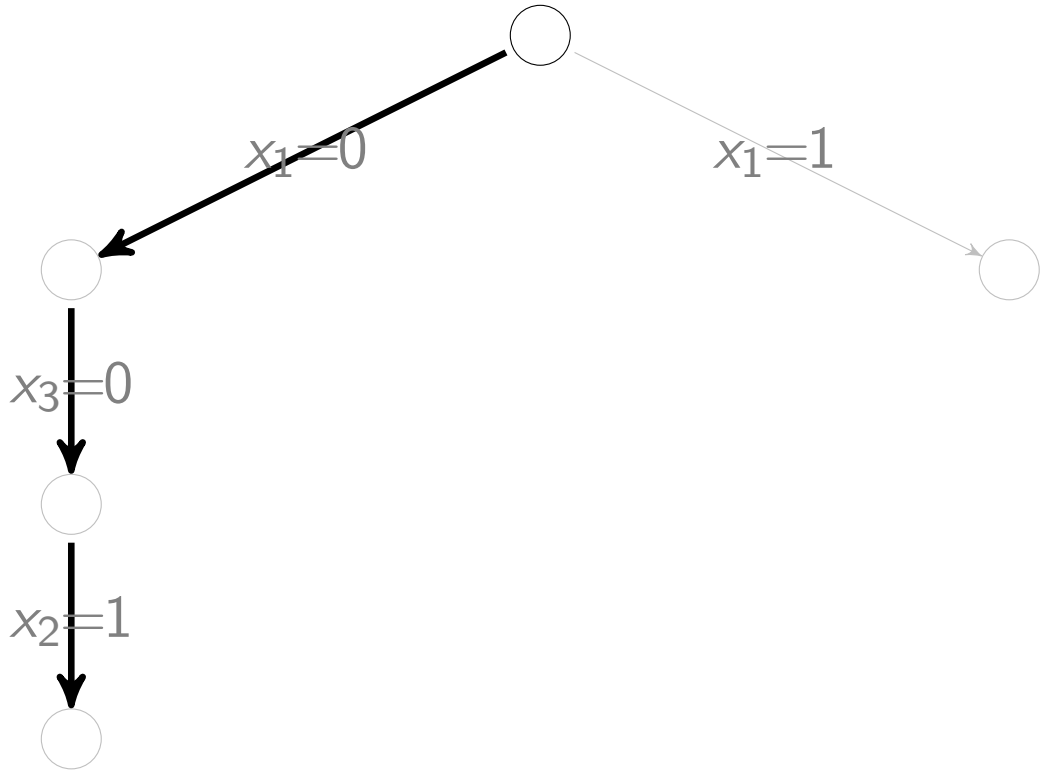
$$(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee \neg x_2) \wedge (x_1 \vee \neg \boldsymbol{x_3})$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 0$ $\quad$ $x_2 = ?$ $\quad$ $x_3 = 0$

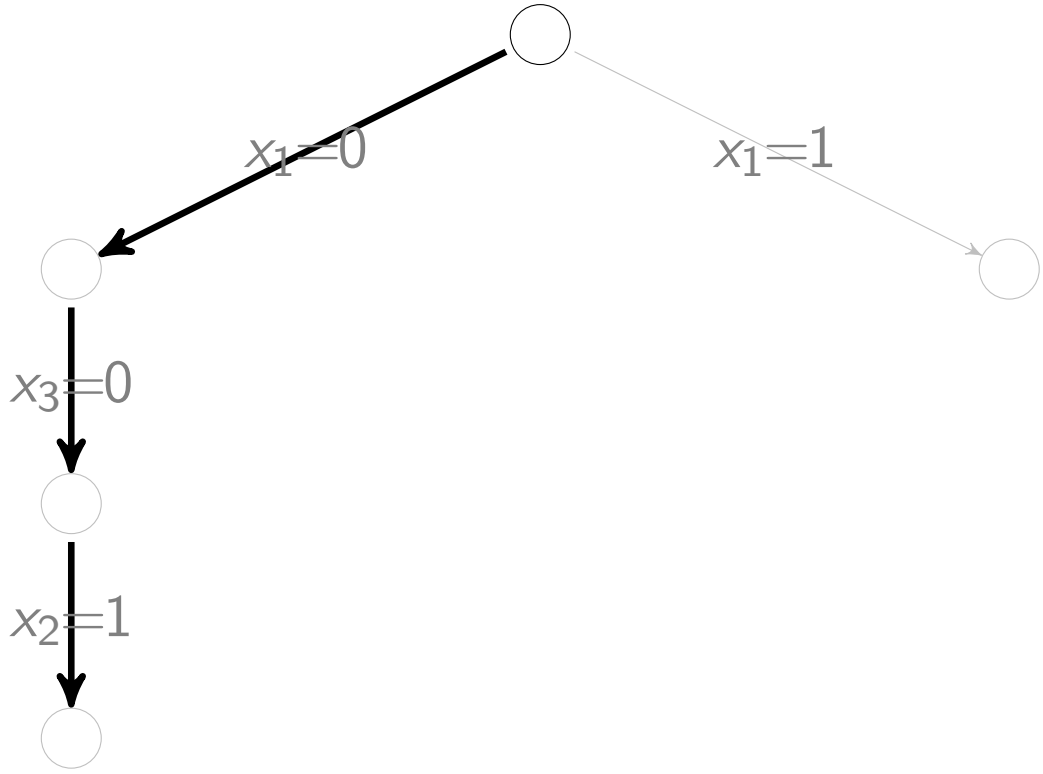
$$(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee \neg x_2) \wedge (x_1 \vee \neg \boldsymbol{x_3})$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 0$ $\quad$ $x_2 = 1$ $\quad$ $x_3 = 0$

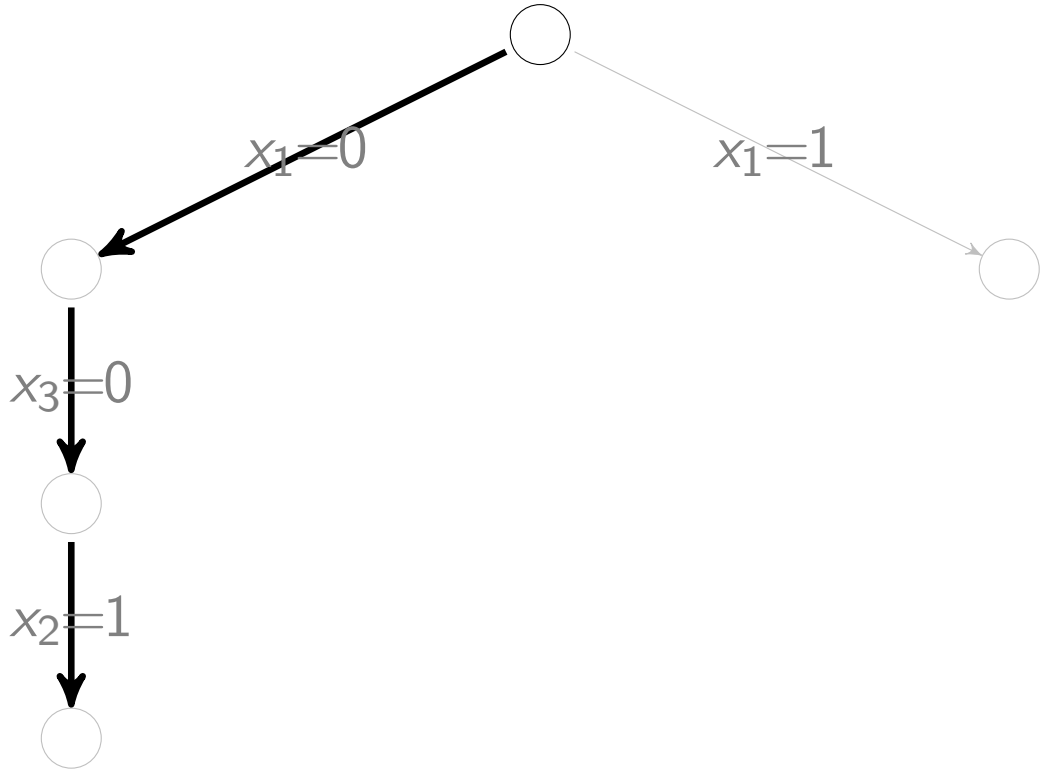
$$(x_1 \vee \boldsymbol{x_2} \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg \boldsymbol{x_3})$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 0$ $x_2 = 1$ $x_3 = 0$

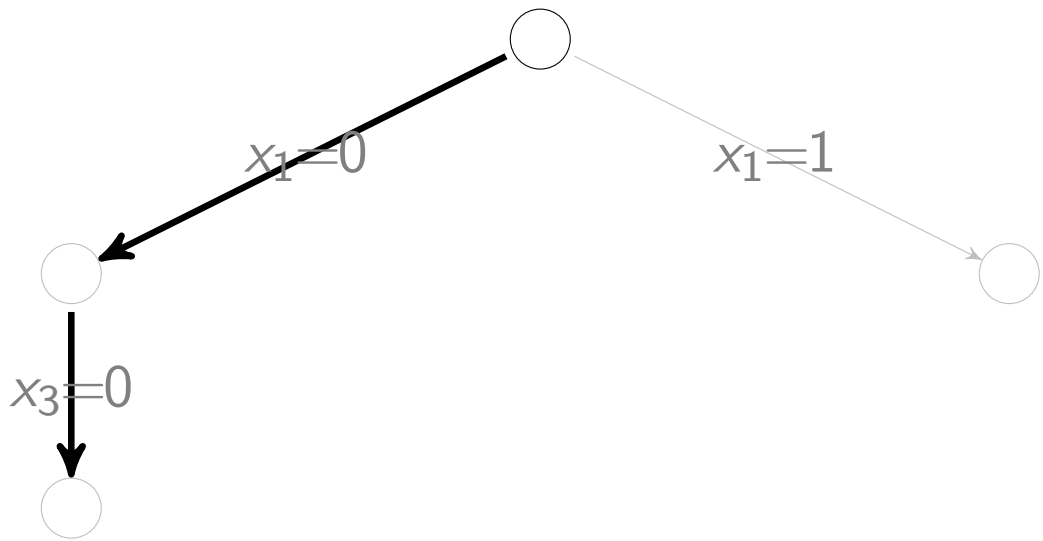
$(x_1 \vee \boldsymbol{x_2} \vee x_3) \wedge (x_3 \vee \neg x_2) \wedge (x_1 \vee \neg \boldsymbol{x_3})$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 0$ $\qquad$ $x_2 = ?$ $\qquad$ $x_3 = 0$

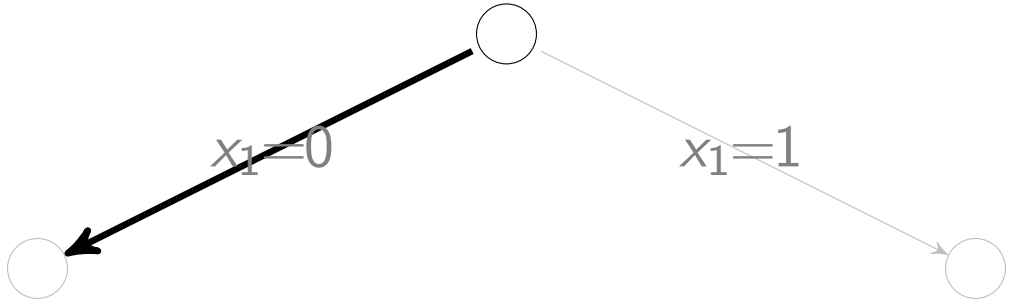
$$(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee \neg x_2) \wedge (x_1 \vee \neg \boldsymbol{x_3})$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 0$ $\qquad$ $x_2 = ?$ $\qquad$ $x_3 = ?$

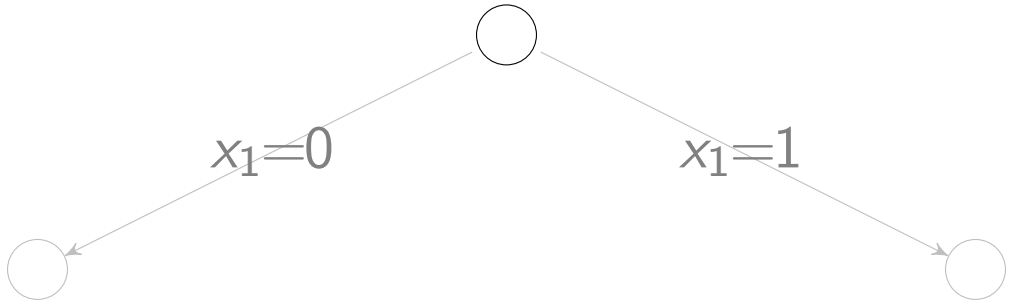
$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = ?$ $\quad$ $x_2 = ?$ $\quad$ $x_3 = ?$

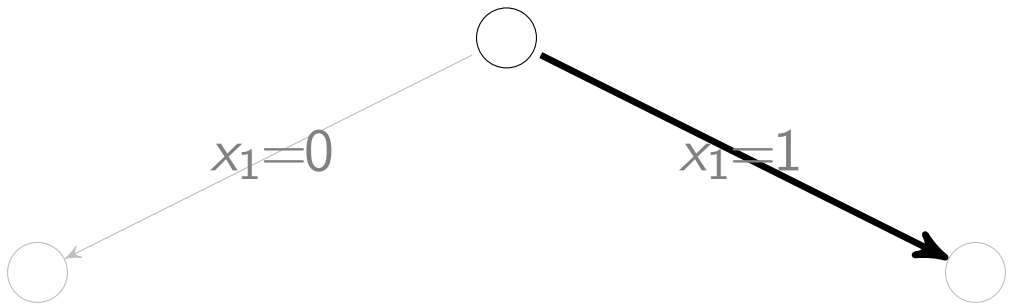
$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = ?$ $\quad$ $x_2 = ?$ $\quad$ $x_3 = ?$

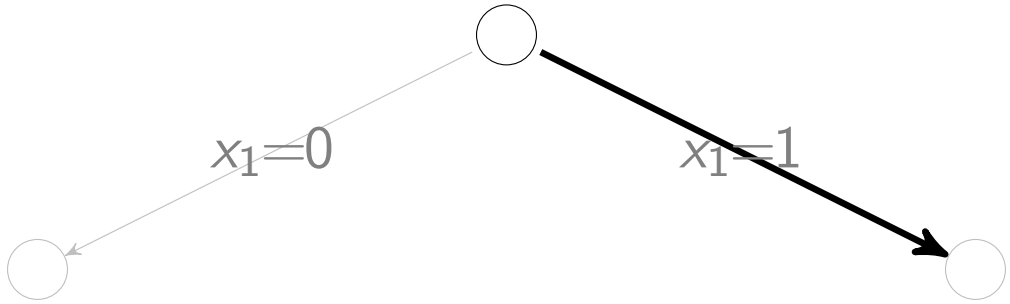
$$(x_1 \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (x_1 \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 1$ $\quad$ $x_2 = ?$ $\quad$ $x_3 = ?$

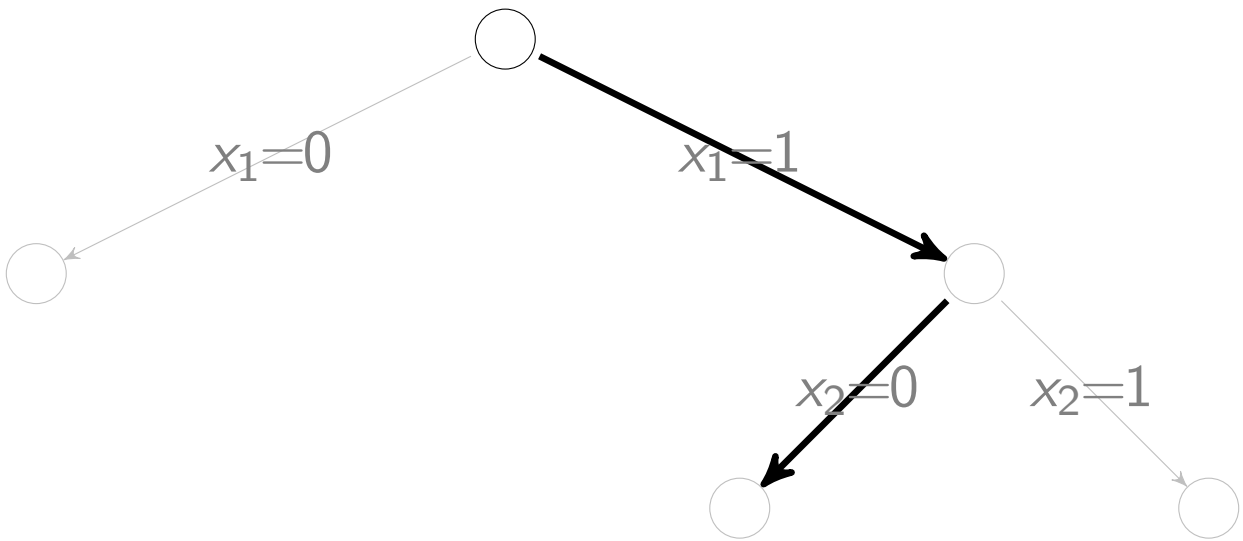
$$(\boldsymbol{x_1} \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (\boldsymbol{x_1} \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 1$ $\qquad$ $x_2 = ?$ $\qquad$ $x_3 = ?$

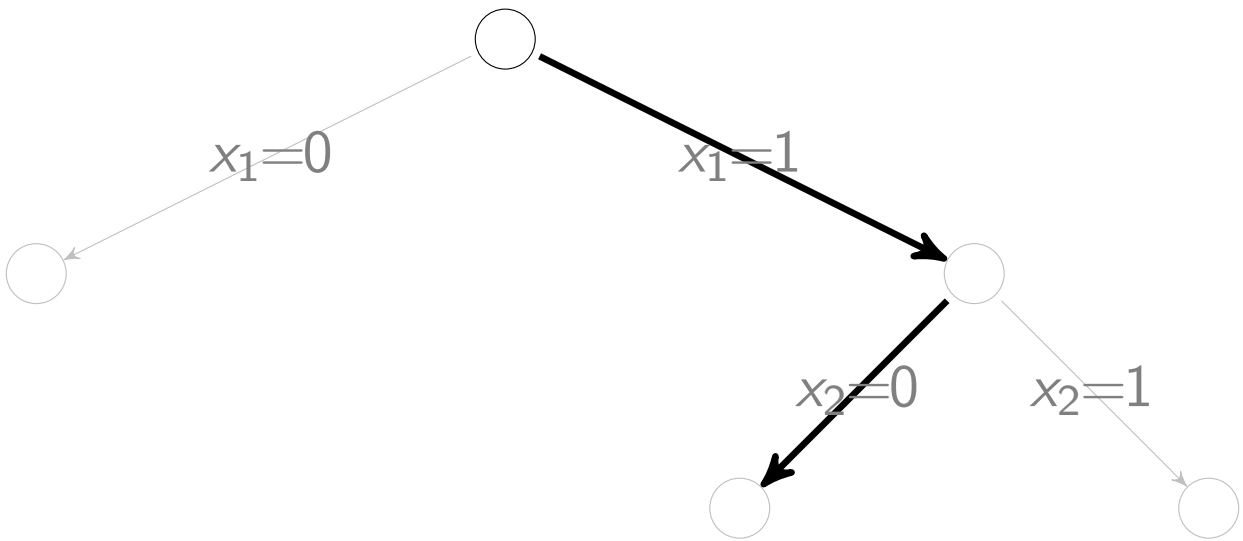
$$(\boldsymbol{x_1} \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg x_2) \bigwedge (\boldsymbol{x_1} \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 1$ $\qquad$ $x_2 = 0$ $\qquad$ $x_3 = ?$

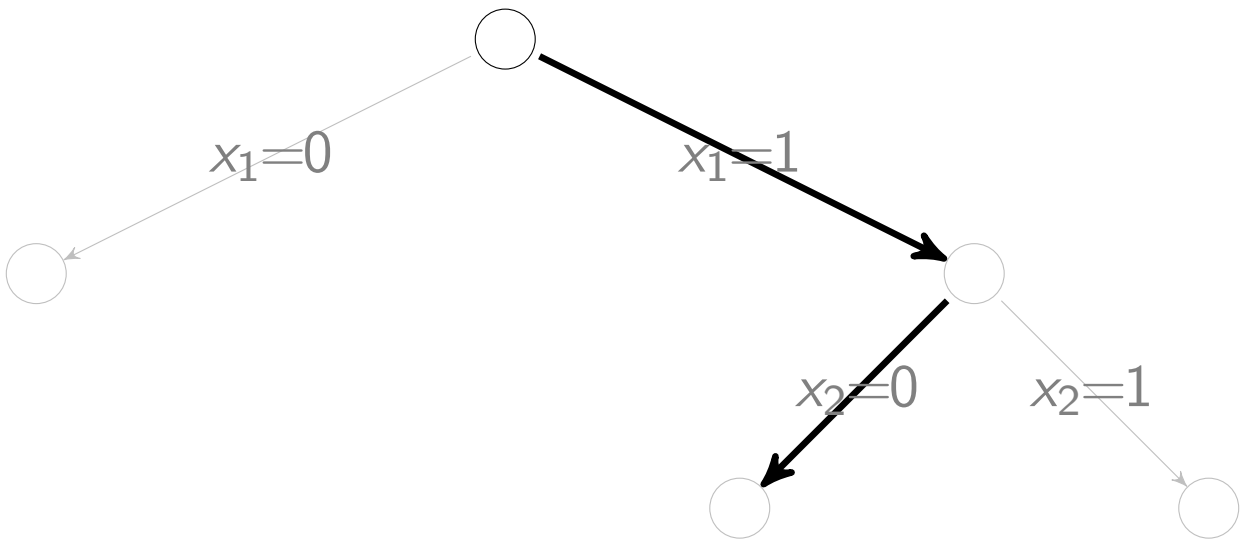
$$(\boldsymbol{x_1} \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg \boldsymbol{x_2}) \bigwedge (\boldsymbol{x_1} \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

$x_1 = 1$ $\qquad$ $x_2 = 0$ $\qquad$ $x_3 = ?$

$$(\boldsymbol{x_1} \vee x_2 \vee x_3) \bigwedge (x_3 \vee \neg \boldsymbol{x_2}) \bigwedge (\boldsymbol{x_1} \vee \neg x_3)$$



DPLL $\simeq$ backtracking + unit propagation

- Key implementation issue: fast data structures to identify conflicts/unit clauses;
  - for each variable: in which clauses does it appear
    - positively;
    - negatively;
  - for each clause: how many
    - true literals;
    - false literals;
  - history of decision literals (together with unit propagations);
- Make sure updates preserve invariants.

CDCL = DPLL +

- variable ordering;
- backjumping;
- clause learning (and forgetting);
  - high-performance solvers use even trickier data structures (two-watched literals);
- random restarts.

Key takeaway: the data structures are complicated. Even DPLL can be tricky to get right!

Possible Issues

- SAT solvers go wrong!
  - See Brummayer et al. (SAT 2010).
  - MiniSAT has > 5.000 lines of C++ code.
- The SAT Competition now requires UNSAT certificates;
  - exponential in the worst case.
- Is there a better way to ensure a higher degree of confidence?

Idea: develop an efficient, provably correct SAT solver.

Dafny

- An object-oriented programming language;
- Developed by Microsoft Research;
- Code can be annotated with preconditions, postconditions and loop invariants;
- Code will fail to compile unless postconditions are provable;
- Uses Boogie and Z3 in the backend;
- Can emit C# code.

Dafny

```
method binarySearch(a: array<int>, key : int) returns (r : int)
   requires ∀ j, k • 0 ≤ j < k < a.Length ⟹ a[j] ≤ a[k];
   ensures r ≥ 0 ⟹ 0 ≤ r < a.Length ∧ a[r] = key;
   ensures r < 0 ⟹ ∀ k • 0 ≤ k < a.Length - 1 ⟹ a[k] ≠ key;
{
   var left : int := 0;
   var right : int := a.Length - 1;
   while (left ≤ right)
      invariant 0 ≤ left ≤ a.Length;
      invariant -1 ≤ right < a.Length;
      invariant ∀ k • 0 ≤ k < left ⟹ a[k] < key;
      invariant ∀ k • right < k < a.Length ⟹ a[k] > key;
   {
      var mid : int := (left + right) / 2;
      if (key < a[mid]) {
         right := mid - 1;
      } else if (key > a[mid]) {
         left := mid + 1;
      } else {
         return mid;
      }
   }
   return -1;
}
```

Dafny

```
method binarySearch(a: array<int>, key : int) returns (r : int)
   requires ∀ j • 0 ≤ j < a.Length - 1 ⟹ a[j] ≤ a[j + 1];
   ensures r ≥ 0 ⟹ 0 ≤ r < a.Length ∧ a[r] = key;
   ensures r < 0 ⟹ ∀ k • 0 ≤ k < a.Length - 1 ⟹ a[k] ≠ key;
{
   var left : int := 0;
   var right : int := a.Length - 1;
   while (left ≤ right)
      invariant 0 ≤ left ≤ a.Length;
      invariant -1 ≤ right < a.Length;
      invariant ∀ k • 0 ≤ k < left ⟹ a[k] < key;
      invariant ∀ k • right < k < a.Length ⟹ a[k] > key;
   {
      var mid : int := (left + right) / 2;
      if (key < a[mid]) {
         right := mid - 1;
      } else if (key > a[mid]) {
         left := mid + 1;
      } else {
         return mid;
      }
   }
   return -1;
}
```

Dafny

```
lemma plusOne(a : array<int>)
   requires ∀ j • 0 ≤ j < a.Length - 1 ⟹ a[j] ≤ a[j + 1];
   ensures ∀ j, k • 0 ≤ j < k < a.Length ⟹ a[j] ≤ a[k];
{
   ∀ j
      | 0 ≤ j < a.Length
      ensures ∀ k • j < k < a.Length ⟹ a[j] ≤ a[k];
      {
         var kp := j + 1;
         while (kp < a.Length)
            invariant j < kp ≤ a.Length;
            invariant ∀ kpp • j < kpp < kp ⟹ a[j] ≤ a[kpp];
         {
            assert a[j] ≤ a[kp - 1];
            kp := kp + 1;
         }
      }
}
method binarySearch(a: array<int>, key : int) returns (r : int)
   requires ∀ j • 0 ≤ j < a.Length - 1 ⟹ a[j] ≤ a[j + 1];
   ensures r ≥ 0 ⟹ 0 ≤ r < a.Length ∧ a[r] = key;
   ensures r < 0 ⟹ ∀ k • 0 ≤ k < a.Length - 1 ⟹ a[k] ≠ key;
{
   plusOne(a);
   var left : int := 0;
```

DPLL in Dafny

- Around 3.200 lines of Dafny code (has gotten shorter), 22 methods, 29 helper lemmas, 324 preconditions, 149 postconditions, 136 invariants, 38 *reads* annotations;
- Organized as five files:
  - `data_structures.dfy` - Core data structures;
  - `stack.dfy` - History of decision points;
  - `formula.dfy` - Init data structures, create decision points;
  - `solver.dfy` - Entry point;
  - `utils.dfy` - Various helper functions.

Function / Method / Lemma	Verification Time (s)
Formula.setLiteral(ℓ, *value*)	630.59
SATSolver.solve()	332.01
Formula.setVariable(v, *value*)	294.54
Formula.undoLayerOnStack()	249.16
SATSolver.step()	233.25
Formula.constructor(vC, *clauses*)	91.14
Others	< 3

What We Prove

```
datatype SAT_UNSAT = SAT(tau: seq<int>) | UNSAT
...
  method solve() returns (result : SAT_UNSAT)
    requires formula.valid();
...
    ensures result.SAT? ⟹
      formula.isSatisfiableExtend(formula.truthAssignment[..]);
    ensures result.UNSAT? ⟹
      ¬formula.isSatisfiableExtend(formula.truthAssignment[..]);
...
  {
...
  }
```

Core Data Structures

```
trait DataStructures {
  var variablesCount : int;
  var clauses : seq<seq<int> >;

  var stack : Stack;
  var truthAssignment : array<int>; // from 0 to variablesCount - 1,

  var trueLiteralsCount : array<int>; // from 0 to |clauses| - 1
  var falseLiteralsCount : array<int>; // from 0 to |clauses| - 1

  var positiveLiteralsToClauses : array<seq<int> >; // from 0 to var
  var negativeLiteralsToClauses : array<seq<int> >; // frm 0 to vari
...
```

Core Data Structures Invariant

```
trait DataStructures {
...
  predicate validNegativeLiteralsToClause(variable : int, s :
seq<int>)
    reads this; requires validVariablesCount();
    requires validClauses(); requires 0 ≤ variable < variablesCount;
  {
    SeqPredicates.valuesBoundedBy(s, 0, |clauses|) ∧
    SeqPredicates.orderedAsc(s) ∧
    (∀ clauseIndex • clauseIndex in s ⟹ -variable-1 in clauses[cl
    (∀ clauseIndex • 0 ≤ clauseIndex < |clauses| ∧ clauseIndex ∉
s ⟹ -variable-1 ∉ clauses[clauseIndex])
  }
...
  predicate valid()
    reads this, stack, stack.stack, truthAssignment, trueLiteralsCou
      falseLiteralsCount, positiveLiteralsToClauses, negativeLiteral
  {
    validReferences() ∧ validVariablesCount() ∧
    validClauses() ∧ validStack() ∧ validTruthAssignment() ∧
    validTrueLiteralsCount(truthAssignment[..]) ∧
    validFalseLiteralsCount(truthAssignment[..]) ∧
    validPositiveLiteralsToClauses() ∧
    validNegativeLiteralsToClauses()
  }
```

Main Unit Propagation Method

```
  method setLiteral(literal : int, value : bool)
    requires valid();
...
    ensures (
      ghost var tau := old(truthAssignment[..]);
      ghost var (variable, val) := convertLVtoVI(literal, value);
      ghost var tau' := tau[variable := val];

      isSatisfiableExtend(tau') ⟺
              isSatisfiableExtend(truthAssignment[..])
    );
...
  {
...
    while (k < |negativelyImpactedClauses|)
...
    {
...
      if (trueLiteralsCount[clauseIndex] = 0 ∧
        falseLiteralsCount[clauseIndex] + 1 = |clauses[clauseIndex]|)
...
        setLiteral(literal, true);
...
        k := k + 1;
    }
```

Final Remarks

- Written by the first author in the course of one year (working part-time; also learned Dafny during this time).
- Some issues found (and fixed) during verification for edge cases (e.g., when the same literal occurs twice in a clause).
- Guarantees:
  - Soundness,
  - Completeness and
  - Termination.
- Performance:

Test (vars, clauses)	Our solver	MiniSat
Hole6 (42, 133)	UNSAT / 3.21s	UNSAT / 0.02s
Zebra (155, 1135)	SAT / 1.09s	SAT / 0.00s
Hanoi4 (718, 4934)	timed out	SAT / 0.03s
Queens16 (256, 6336)	SAT / 4.64s	SAT / 0.00s

Conclusion and Future Work

- It is possible to validate a SAT solver using the Dafny system;
- The resulting code is (currently) far from efficient, but there is good hope..
- Large verification time is the main issue and difficulty:
  - Split large functions into smaller functions;
  - Verify only a single function/lemma at a time;
  - Imperative features
    - make verification annoying and time consuming: e.g., must convince Dafny that certain data structures do not change;
    - make the solver **fast**;
- Future work:
  - improve verification and runtime performance;
  - implement CDCL;
  - apply know-how to C code (e.g., with Frama-C).

Thank you!