TFIP-AI - Machine Learning

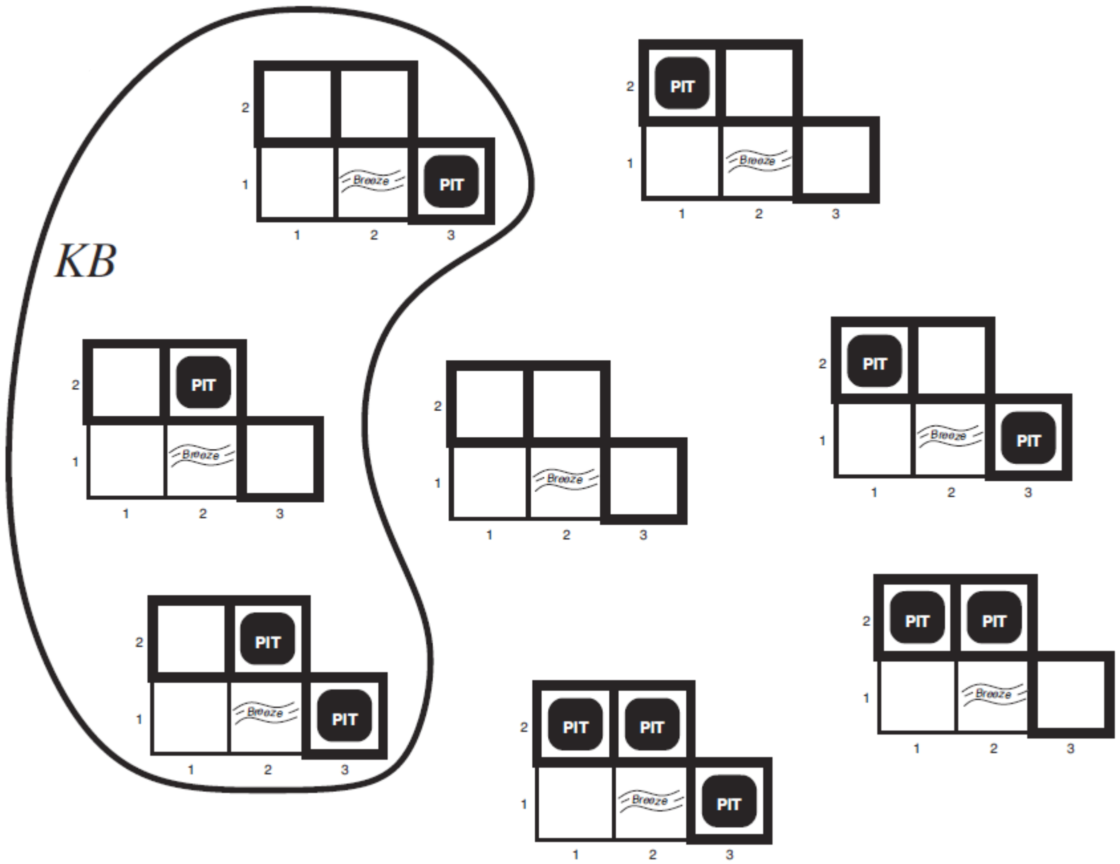
Unit 3 Logical and Reasoning Systems

Part 2 Logical Agents

Entailment

Does the knowledge base entail my query?

- Query 1: $\neg\, P[1,2]$
- Query 2: $\neg\, P[2,2]$



Logical Agent Vocab

Model

- Complete assignment of symbols to True/False

Sentence

- Logical statement
- Composition of logic symbols and operators

KB

- Collection of sentences representing facts and rules we know about the world

Query

- Sentence we want to know if it is *probably* True, *provably* False, or *unsure*.

Logical Agent Vocab

Entailment

- Input: sentence1, sentence2
- Each model that satisfies sentence1 must also satisfy sentence2
- "If I know 1 holds, then I know 2 holds"
- (ASK), TT-ENTAILS, FC-ENTAILS

Satisfy

- Input: model, sentence
- Is this sentence true in this model?
- Does this model satisfy this sentence
- "Does this particular state of the world work?’
- PL-TRUE

Logical Agent Vocab

Satisfiable

- Input: sentence
- Can find at least one model that satisfies this sentence
  - (We often want to know what that model is)
- "Is it possible to make this sentence true?"
- DPLL

Valid

- Input: sentence
- sentence is true in all possible models

Propositional Logical Vocab

Literal

- Atomic sentence: True, False, Symbol, ¬Symbol

Clause

- Disjunction of literals: $A \vee B \vee \neg C$

Definite clause

- Disjunction of literals, *exactly one* is positive
- $\neg A \vee B \vee \neg C$

Horn clause

- Disjunction of literals, *at most one* is positive
- All definite clauses are Horn clauses

Entailment

How do we implement a logical agent that proves entailment?

- Logic language
  - Propositional logic
  - First order logic

- Inference algorithms
  - Theorem proving
  - Model checking

Propositional Logic

Check if sentence is true in given model

In other words, does the model *satisfy* the sentence?

function PL-TRUE?(α,model) returns true or false

if α is a symbol then return Lookup(α, model)

if Op(α) = $\neg$ then return not(PL-TRUE?(Arg1(α),model))

if Op(α) = $\wedge$ then return and(PL-TRUE?(Arg1(α),model),
PL-TRUE?(Arg2(α),model))

etc.

(Sometimes called "recursion over syntax")

Simple Model Checking

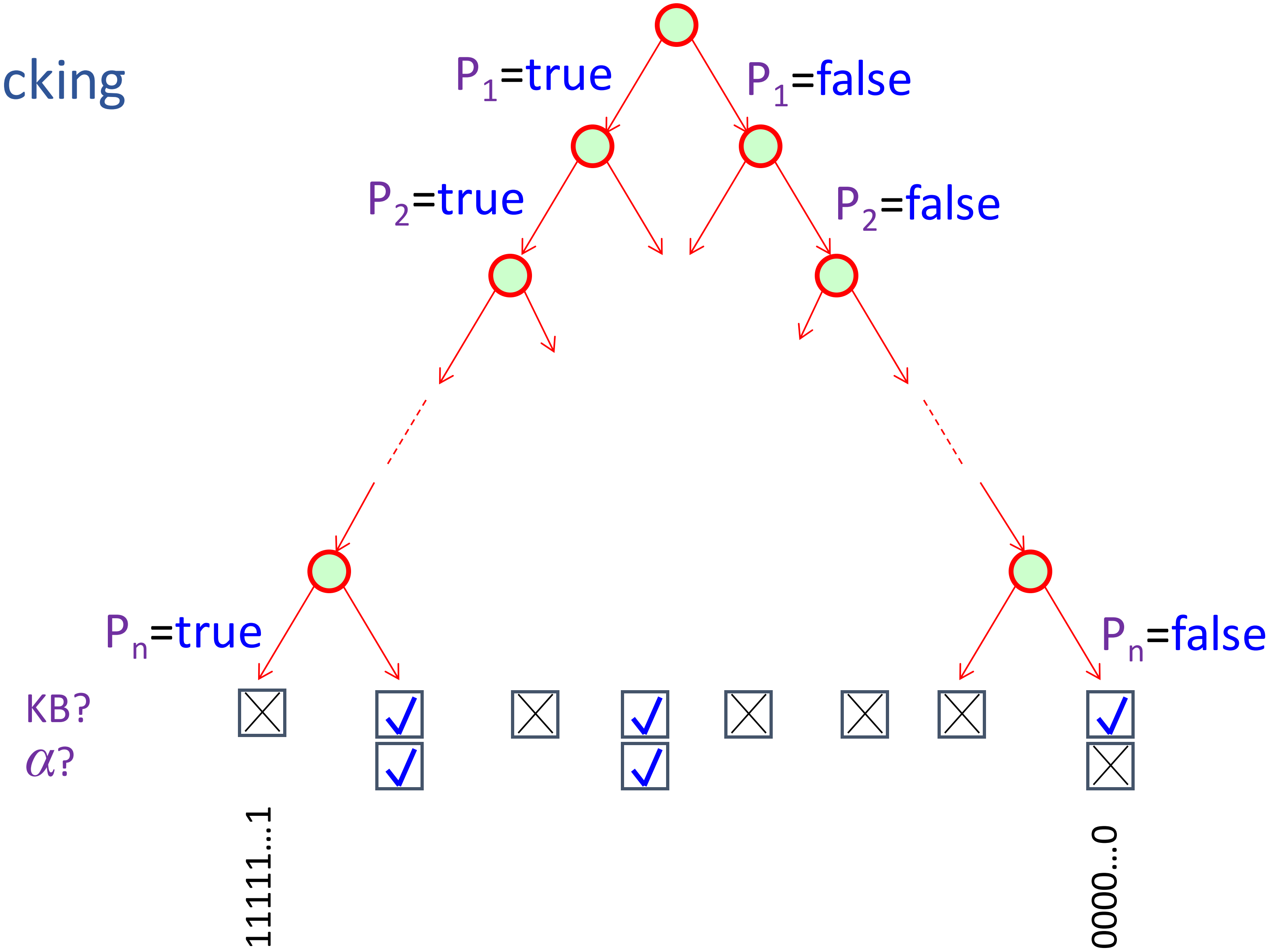
function TT-ENTAILS?(KB, α) returns true or false

Simple Model Checking, contd.

Same recursion as backtracking

$O(2^n)$ time, linear space

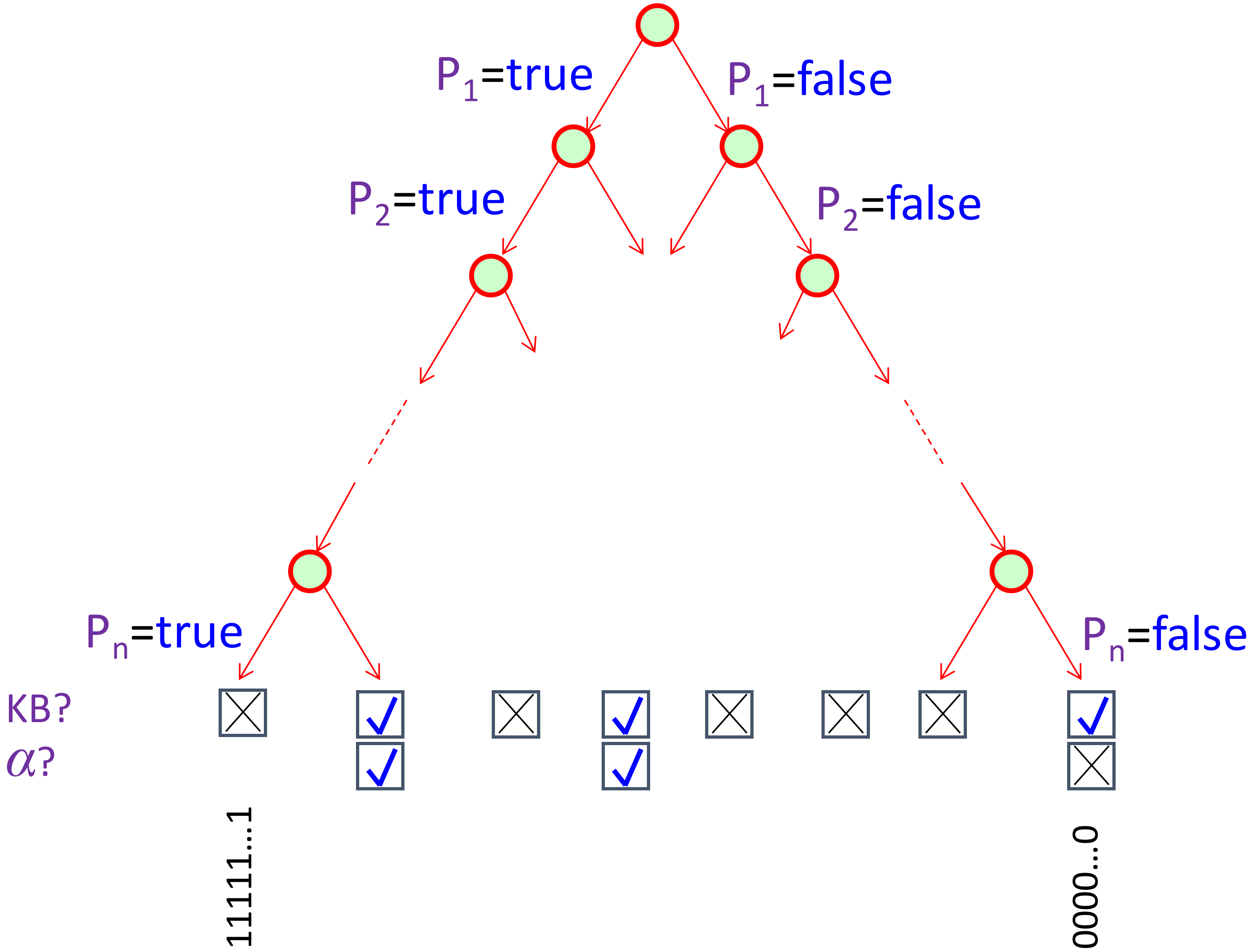
We can do much better!



What is the answer for this?

Which would you choose?

- DFS
- BFS



Simple Model Checking

```
function TT-ENTAILS?(KB, α) returns true or false
    return TT-CHECK-ALL(KB, α, symbols(KB) U symbols(α),{})

function TT-CHECK-ALL(KB, α, symbols,model) returns true or false
    if empty?(symbols) then
        if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
        else return true
    else
        P ← first(symbols)
        rest ← rest(symbols)
        return  and (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
                     TT-CHECK-ALL(KB, α, rest, model ∪ {P = false }))
```

Inference: Proofs

A proof is a *demonstration* of entailment between α and β

Method 1: *model-checking*

- For every possible world, if α is true make sure that is β true too
- OK for propositional logic (finitely many worlds); not easy for first-order logic

Method 2: *theorem-proving*

- Search for a sequence of proof steps (applications of *inference rules*) leading from α to β
- E.g., from $P \wedge (P \Rightarrow Q)$, infer Q by *Modus Ponens*

Properties

- *Sound* algorithm: everything it claims to prove is in fact entailed
- *Complete* algorithm: every sentence that is entailed can be proved

Simple Theorem Proving: Forward Chaining

Forward chaining applies Modus Ponens to generate new facts:

- Given $X_1 \wedge X_2 \wedge \ldots X_n \Rightarrow Y$ and $X_1, X_2, \ldots, X_n$
- Infer Y

Forward chaining keeps applying this rule, adding new facts, until nothing more can be added

Requires KB to contain only *definite clauses*:

- (Conjunction of symbols) $\Rightarrow$ symbol; or
- A single symbol (note that X is equivalent to $True \Rightarrow X$)

Forward Chaining Algorithm

function PL-FC-ENTAILS?(KB, q) returns true or false

count ← a table, where count[c] is the number of symbols in c's premise

inferred ← a table, where inferred[s] is initially false for all s

agenda ← a queue of symbols, initially symbols known to be true in KB

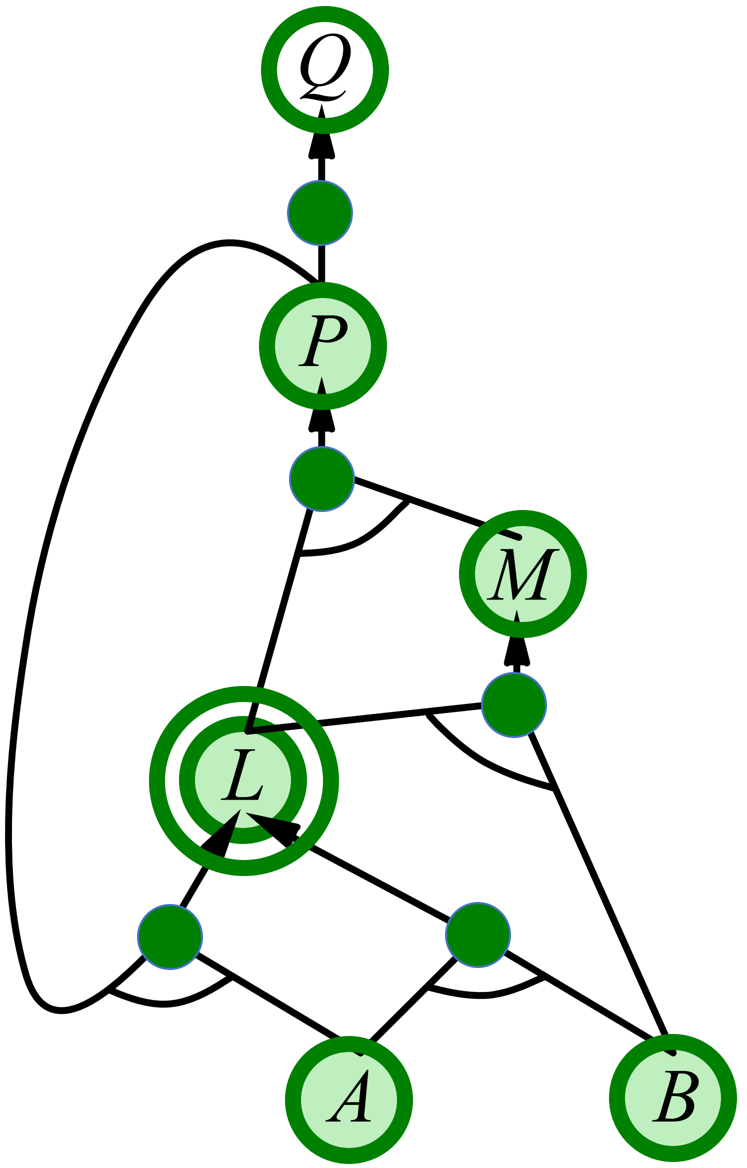
CLAUSES	COUNT	INFERRED	AGENDA
$P \Rightarrow Q$	1	A false	
$L \wedge M \Rightarrow P$	2	B false	
$B \wedge L \Rightarrow M$	2	L false	
$A \wedge P \Rightarrow L$	2	M false	
$A \wedge B \Rightarrow L$	2	P false	
A	0	Q false	
B	0		

Forward Chaining Example: Proving Q

CLAUSES	COUNT	INFERRED
$P \Rightarrow Q$	~~1~~ 0	A ~~false~~ true
$L \wedge M \Rightarrow P$	~~2~~ ~~1~~ 0	B ~~false~~ true
$B \wedge L \Rightarrow M$	~~2~~ ~~1~~ 0	L ~~false~~ true
$A \wedge P \Rightarrow L$	~~2~~ ~~1~~ 0	M ~~false~~ true
$A \wedge B \Rightarrow L$	~~2~~ ~~1~~ 0	P ~~false~~ true
A	0	Q ~~false~~ true
B	0	

AGENDA

~~A~~ ~~B~~ ~~L~~ ~~M~~ ~~P~~ ~~L~~ ~~Q~~



Forward Chaining Algorithm

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  count ← a table, where count[c] is the number of symbols in c's premise
  inferred ← a table, where inferred[s] is initially false for all s
  agenda ← a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
      p ← Pop(agenda)
      if p = q then return true
      if inferred[p] = false then
          inferred[p]←true
          for each clause c in KB where p is in c.premise do
              decrement count[c]
              if count[c] = 0 then add c.conclusion to agenda
  return false
```

Properties of forward chaining

Theorem: FC is sound and complete for definite-clause KBs

Soundness: follows from soundness of Modus Ponens (easy to check)

Completeness proof:

1. FC reaches a fixed point where no new atomic sentences are derived
2. Consider the final *inferred* table as a model ***m***, assigning true/false to symbols
3. Every clause in the original KB is true in ***m***

   Proof: Suppose a clause $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ is false in ***m***
   Then $a_1 \wedge \ldots \wedge a_k$ is true in ***m*** and b is false in ***m***
   Therefore the algorithm has not reached a fixed point!
4. Hence ***m*** is a model of KB
5. If KB |= q, q is true in every model of KB, including ***m***

A	false	xxxx	true
B	false	xxxx	true
L	false	xxxx	true
M	false	xxxx	true
P	false	xxxx	true
Q	false	xxxx	true

Satisfiability and Entailment

A sentence is *satisfiable* if it is true in at least one world (cf CSPs!)

Suppose we have a hyper-efficient SAT solver; how can we use it to test entailment?

- Suppose $\alpha \models \beta$
- Then $\alpha \Rightarrow \beta$ is true in all worlds
- Hence $\neg(\alpha \Rightarrow \beta)$ is false in all worlds
- Hence $\alpha \wedge \neg\beta$ is false in all worlds, i.e., unsatisfiable

So, add the negated conclusion to what you know, test for (un)satisfiability; also known as `reductio ad absurdum`

Efficient SAT solvers operate on ***conjunctive normal form***

Conjunctive Normal Form (CNF)

Every sentence can be expressed

Each clause is a disjunction of literal

Each literal is a symbol or a neg … d sym

Conversion to CNF by a se quence … andard transform … s:

Replace biconditional by two implications

Replace $\alpha \Rightarrow \beta$ by $\neg\alpha \vee \beta$

Distribute v over $\wedge$

- At_1,1_0 $\Rightarrow$ (Wall_0,1 $\Leftrightarrow$ Blocked_W_0)
- At_1,1_0 $\Rightarrow$ ((Wall_0,1 $\Rightarrow$ Blocked_W_0) $\wedge$ (Blocked_W_0 $\Rightarrow$ Wall_0,1))
- $\neg$At_1,1_0 v (($\neg$Wall_0,1 v Blocked_W_0) $\wedge$ ($\neg$Blocked_W_0 v Wall_0,1))
- ($\neg$At_1,1_0 v $\neg$Wall_0,1 v Blocked_W_0) $\wedge$ ($\neg$At_1,1_0 v $\neg$Blocked_W_0 v Wall_0,1)

Efficient SAT solvers

DPLL (Davis-Putnam-Logemann-Loveland) is the core of modern solvers.

The DPLL algorithm is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form, i.e. for solving the CNF-SAT problem.

Efficient SAT solvers

DPLL:

Essentially a backtracking search over models with some extras:

- *Early termination*: stop if
  - all clauses are satisfied; e.g., $(A \vee B) \wedge (A \vee \neg C)$ is satisfied by {A=true}
  - any clause is falsified; e.g., $(A \vee B) \wedge (A \vee \neg C)$ is satisfied by {A=false, B=false}
- *Pure literals*: if all occurrences of a symbol in as-yet-unsatisfied clauses have the same sign, then give the symbol that value
  - E.g., A is pure and positive in $(A \vee B) \wedge (A \vee \neg C) \wedge (C \vee \neg B)$ so set it to true
- *Unit clauses*: if a clause is left with a single literal, set symbol to satisfy clause
  - E.g., if A=false, $(A \vee B) \wedge (A \vee \neg C)$ becomes $(\text{false} \vee B) \wedge (\text{false} \vee \neg C)$, i.e. $(B) \wedge (\neg C)$
  - Satisfying the unit clauses often leads to further propagation, new unit clauses, etc.

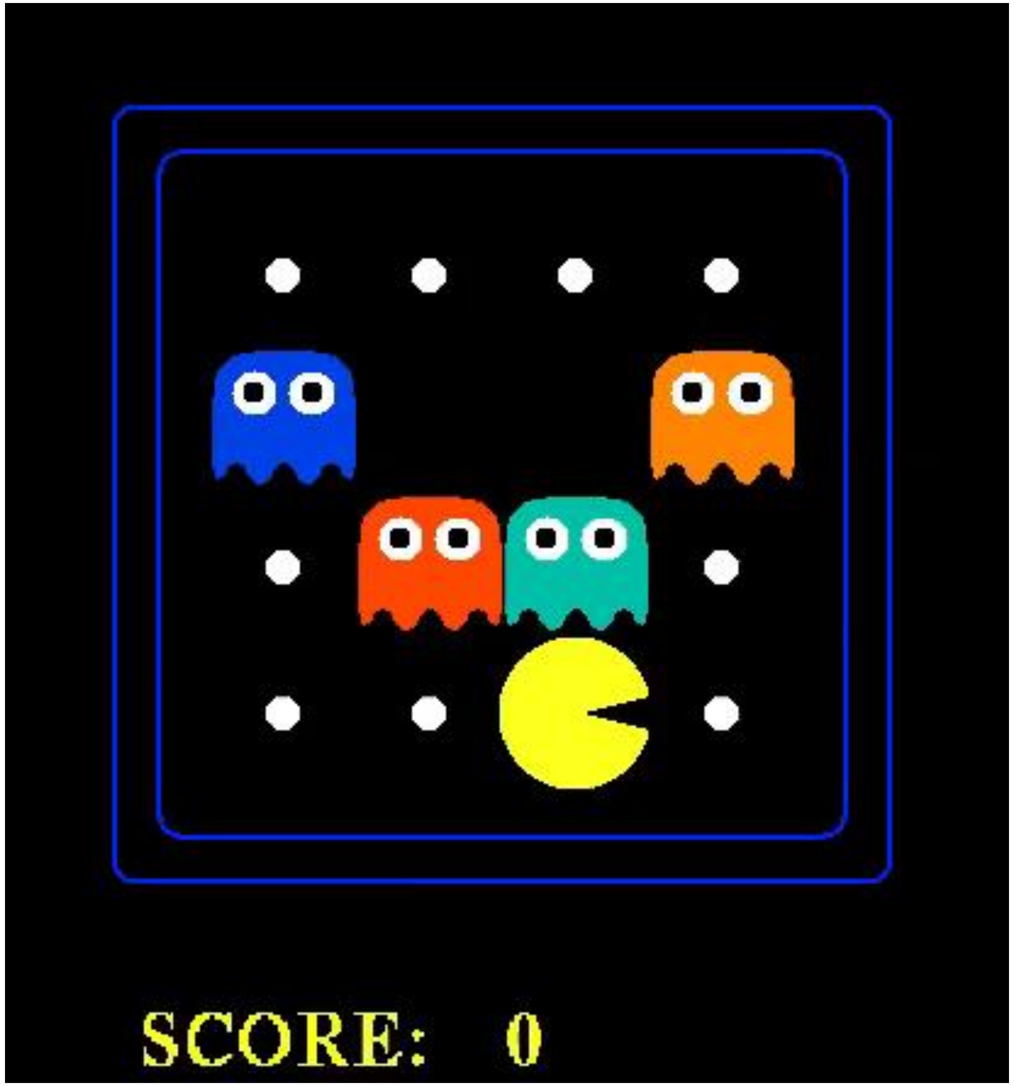
DPLL algorithm

```
function DPLL(clauses, symbols, model) returns true or false
   if every clause in clauses is true in model then return true
   if some clause in clauses is false in model then return false

   P, value ←FIND-PURE-SYMBOL(symbols, clauses, model)
   if P is non-null then return DPLL(clauses, symbols–P, model∪{P=value})

   P, value ←FIND-UNIT-CLAUSE(clauses, model)
   if P is non-null then return DPLL(clauses, symbols–P, model∪{P=value})

   P ← First(symbols)
   rest ← Rest(symbols)

   return or(DPLL(clauses, rest, model∪{P=true}),
             DPLL(clauses, rest, model∪{P=false}))
```

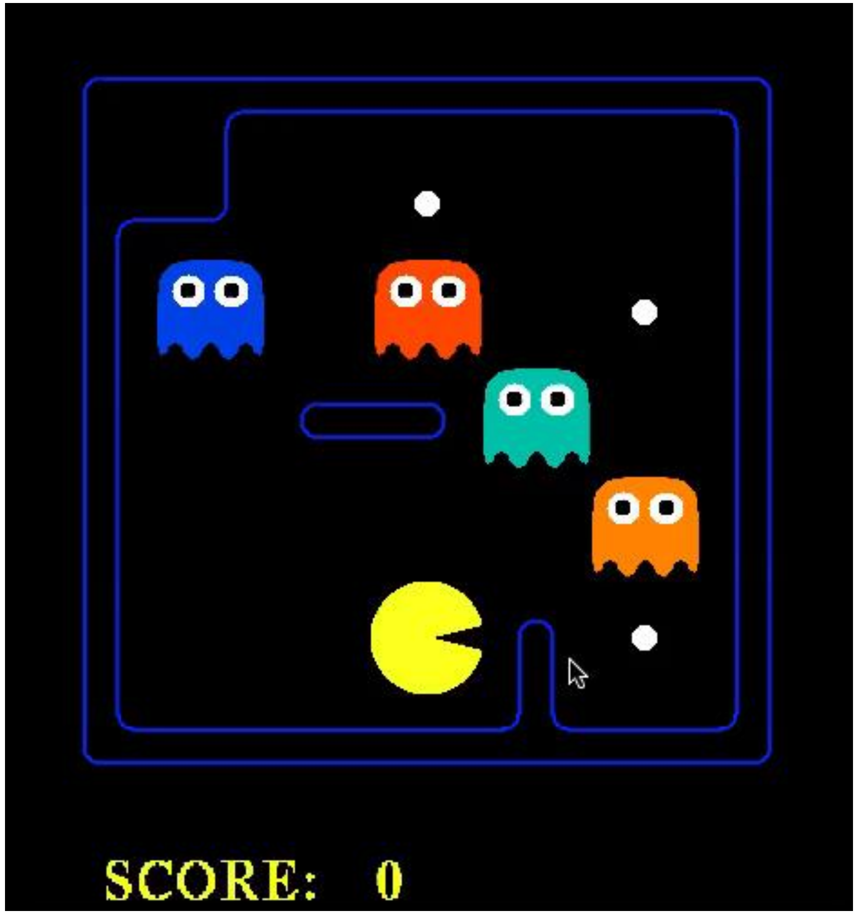
Planning as Satisfiability

Given a hyper-efficient SAT solver, can we use it to make plans?

Yes, for fully observable, deterministic case: planning problem is solvable iff there is some satisfying assignment for actions etc.


SCORE: 0


SCORE: 0


SCORE: 0

Planning as Satisfiability

Given a hyper-efficient SAT solver, can we use it to make plans?

Yes, for fully observable, deterministic case: planning problem is solvable iff there is some satisfying assignment for actions etc.

For T = 1 to infinity, set up the KB as follows and run SAT solver:

- Initial state, domain constraints
- Transition model sentences up to time T
- Goal is true at time T
- *Precondition axioms*: $At\_1,1\_0 \wedge N\_0 \Rightarrow \neg Wall\_1,2$ etc.
- *Action exclusion axioms*: $\neg(N\_0 \wedge W\_0) \wedge \neg(N\_0 \wedge S\_0) \wedge ..$ etc.

Initial State

The agent may know its initial location:

- At_1,1_0

Or, it may not:

- At_1,1_0 v At_1,2_0 v At_1,3_0 v … v At_3,3_0

We also need a *domain constraint* – cannot be in two places at once!

- $\neg$(At_1,1_0 $\wedge$ At_1,2_0) $\wedge$ $\neg$(At_1,1_0 $\wedge$ At_1,3_0) $\wedge$ …
- $\neg$(At_1,1_1 $\wedge$ At_1,2_1) $\wedge$ $\neg$(At_1,1_1 $\wedge$ At_1,3_1) $\wedge$ …
- …

Transition Model

How does each *state variable* or *fluent* at each time gets its value?

State variables for PL Pacman are At_x,y_t , e.g., At_3,3_17

A state variable gets its value according to a *successor-state axiom*

- $X_t \Leftrightarrow [X_{t-1} \wedge \neg(\text{some action}_{t-1} \text{ made it false})] \vee$
  $[\neg X_{t-1} \wedge (\text{some action}_{t-1} \text{ made it true})]$

For Pacman location:

- At_3,3_17 $\Leftrightarrow$ [At_3,3_16 $\wedge$ $\neg$(($\neg$Wall_3,4 $\wedge$ N_16) v ($\neg$Wall_4,3 $\wedge$ E_16) v …)]
  v [$\neg$At_3,3_16 $\wedge$ ((At_3,2_16 $\wedge$ $\neg$Wall_3,3 $\wedge$ N_16) v
  (At_2,3_16 $\wedge$ $\neg$Wall_3,3 $\wedge$ N_16) v …)]