CSC242: Intro to AI

Lecture 13

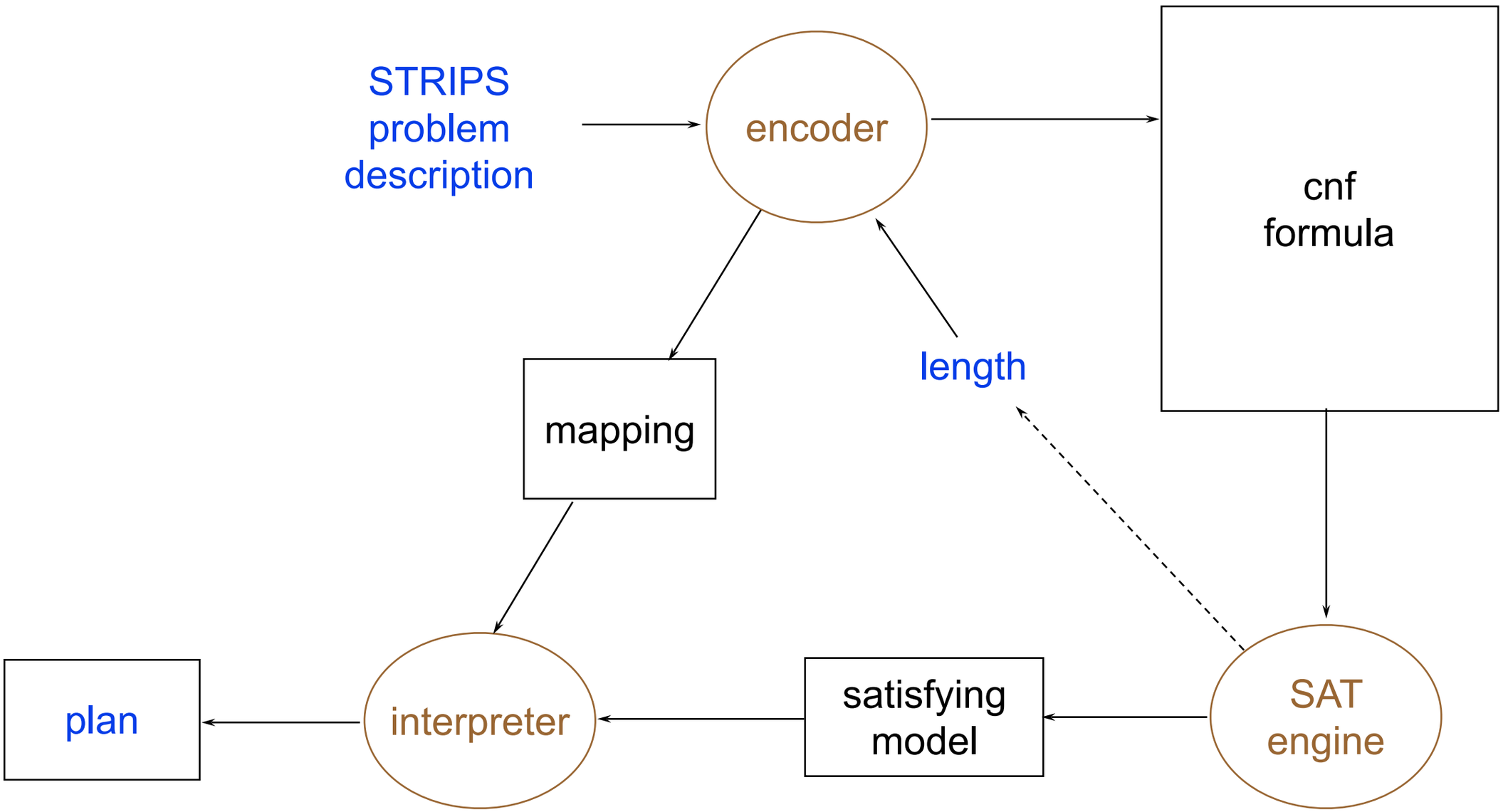
Planning as Satisfiability

Efficient Satisfiability Algorithms


Skillful Manipulation Based on
High-speed Sensory Motor Fusion

Planning as Satisfiability

SATPLAN


STRIPS
problem
description
encoder
cnf
formula
length
mapping
plan
interpreter
satisfying
model
SAT
engine

Translating STRIPS

- Ground action = a STRIPS operator with constants assigned to all of its parameters
- Ground fluent = a precondition or effect of a ground action

operator: Fly(a,b)
precondition: At(a), Fueled
effect: At(b), ~At(a), ~Fueled
constants: NY, Boston, Seattle

Ground actions:

Ground fluents:

Translating STRIPS

- Ground action = a STRIPS operator with constants assigned to all of its parameters
- Ground fluent = a precondition or effect of a ground action

operator: Fly(a,b)
precondition: At(a), Fueled
effect: At(b), ~At(a), ~Fueled
constants: NY, Boston, Seattle

Ground actions: Fly(NY,Boston), Fly(NY,Seattle), Fly(Boston,NY), Fly(Boston,Seattle), Fly(Seattle,NY), Fly(Seattle,Boston)

Ground fluents: Fueled, At(NY), At(Boston), At(Seattle)

Clause Schemas

$$\forall x \in \{A, B, C\} P(x)$$

represents

$$P(A) \wedge P(B) \wedge P(C)$$

This is not the same as FOL quantification, because we are quantifying over a set of symbols (constants), not over the elements of a domain of an interpretation (model).

Existential Quantification

$\exists x \in \{A, B, C\} P(x)$

represents

$P(A) \vee P(B) \vee P(C)$

This is not the same as FOL quantification, because we are quantifying over a set of symbols (constants), not over the elements of a domain of an interpretation (model).

Named Sets

It is often convenient to give a name to a set of constants:

$T = \{A, B, C\}$

$\forall x \in T \ . \ P(x)$

$\exists y \in T \ . \ Q(y)$

means the same thing as:

$\forall x \in \{A, B, C\} P(x)$

$\exists x \in \{A, B, C\} Q(x)$

In full FOL, we could define sets using predicate that is true just for members of the set:

$T(A) \wedge T(B) \wedge T(C) \wedge$

$\forall y . T(y) \supset (y = A \vee y = B \vee y = C)$

Conditions on Quantifiers

Expressions that can be evaluated to "true" or "false" can be added as conditions on the quantifiers in schemas.

Equality is interpeted as "the same symbol".

$$\forall x,y \in \{A,B,C\} : x \neq y \ . \ P(x)$$

expands to the ground formula

$$P(A,B) \wedge P(A,C) \wedge P(B,A) \wedge P(B,C) \wedge P(C,A) \wedge P(C,B)$$

Similarly,

$$\exists x,y \in \{A,B,C\} : x \neq y \ . \ P(x)$$

expands to the ground formula

$$P(A,B) \vee P(A,C) \vee P(B,A) \vee P(B,C) \vee P(C,A) \vee P(C,B)$$

SAT Encoding

- Time is sequential and discrete
  - Represented by integers
  - Actions occur instantaneously at a time point
  - Each fluent is true or false at each time point
- If an action occurs at time i, then its preconditions must hold at time i
- If an action occurs at time i, then its effects must hold at time i+1
- If a fluent changes its truth value from time i to time i+1, one of the actions with the new value as an effect must have occurred at time i
- Two conflicting actions cannot occur at the same time
- The initial state holds at time 0, and the goals hold at a given final state K

SAT Encoding

- If an action occurs at time i, then its preconditions must hold at time i

operator: Fly(a,b)
precondition: At(a), Fueled
effect: At(b), ~At(a), ~Fueled
cities: NY, Boston, Seattle

$$\forall i \in \{1,2,\ldots,K\}$$

$$\forall a \in \{\text{NY},\text{Boston},\text{Seattle}\}$$

$$\forall b \in \{\text{NY},\text{Boston},\text{Seattle}\}$$

$$\text{fly(a,b,i)} \supset (\text{at(a,i)} \wedge \text{fuel(i)})$$

SAT Encoding

- If an action occurs at time i, then its preconditions must hold at time i

operator: Fly(a,b)
precondition: At(a), Fueled
effect: At(b), ~At(a), ~Fueled
cities: NY, Boston, Seattle

$$\forall i \in Times$$

$$\forall a \in Cities$$

$$\forall b \in Cities$$

$$fly(a,b,i) \supset (at(a,i) \wedge fuel(i))$$

SAT Encoding

- If an action occurs at time i, then its effects must hold at time i+1

operator: Fly(a,b)
    precondition: At(a), Fueled
    effect: At(b), ~At(a), ~Fueled
constants: NY, Boston, Seattle

SAT Encoding

- If an action occurs at time i, then its effects must hold at time i+1

operator: Fly(a,b)
precondition: At(a), Fueled
effect: At(b), ~At(a), ~Fueled
constants: NY, Boston, Seattle

$$\forall i \in Times$$

$$\forall a \in Cities$$

$$\forall b \in Cities$$

$$fly(a,b,i) \supset (at(b,i+1)) \wedge \neg at(a,i+1) \wedge \neg fuel(i+1))$$

SAT Encoding

- If a fluent changes its truth value from time i to time i+1, one of the actions with the new value as an effect must have occurred at time i

operator: Fly(a,b)
precondition: At(a), Fueled
effect: At(b), ~At(a), ~Fueled
cities: NY, Boston, Seattle

$$\forall i \in Times$$

$$\forall b \in Cities$$

$$(\neg \text{at(b,i)} \wedge \text{at(b,i+1)}) \supset$$

$$\exists a \in Cities \ . \ \text{fly(a,b,i)}$$

SAT Encoding

- If a fluent changes its truth value from time i to time i+1, one of the actions with the new value as an effect must have occurred at time i

operator: Fly(a,b)
precondition: At(a), Fueled
effect: At(b), ~At(a), ~Fueled
cities: NY, Boston, Seattle

$$\forall i \in Times$$

$$\forall b \in Cities$$

$$(\neg at(b,i) \wedge at(b,i+1)) \supset$$

$$(fly(NY,b,i) \vee fly(Boston,b,i) \vee fly(Seattle,b,i))$$

SAT Encoding

- If a fluent changes its truth value from time i to time i+1, one of the actions with the new value as an effect must have occurred at time i
- Change from true to false:

operator: Fly(a,b)
precondition: At(a), Fueled
effect: At(b), ~At(a), ~Fueled
cities: NY, Boston, Seattle

$$\forall i \in Times$$

$$\forall a \in Cities$$

$$(at(a,i) \wedge \neg at(a,i+1)) \supset$$

$$\exists b \in Cities \ . \ fly(a,b,i)$$

Action Mutual Exclusion

- Two conflicting actions cannot occur at the same time
  - Actions conflict if one modifies a precondition or effect of another

operator: Fly(a,b)
precondition: At(a), Fueled
effect: At(b), ~At(a), ~Fueled
cities: NY, Boston, Seattle

Action Mutual Exclusion

- Two conflicting actions cannot occur at the same time
  - Actions conflict if one modifies a precondition or effect of another

operator: Fly(a,b)
precondition: At(a), Fueled
effect: At(b), ~At(a), ~Fueled
cities: NY, Boston, Seattle

$$\forall i \in Times$$

$$\forall a,b,c,d \in Cities : a \neq b \vee c \neq d$$

$$\neg \text{fly(a,b,i)} \vee \neg \text{fly(c,d,i)}$$

Satplan Demo (blackbox)

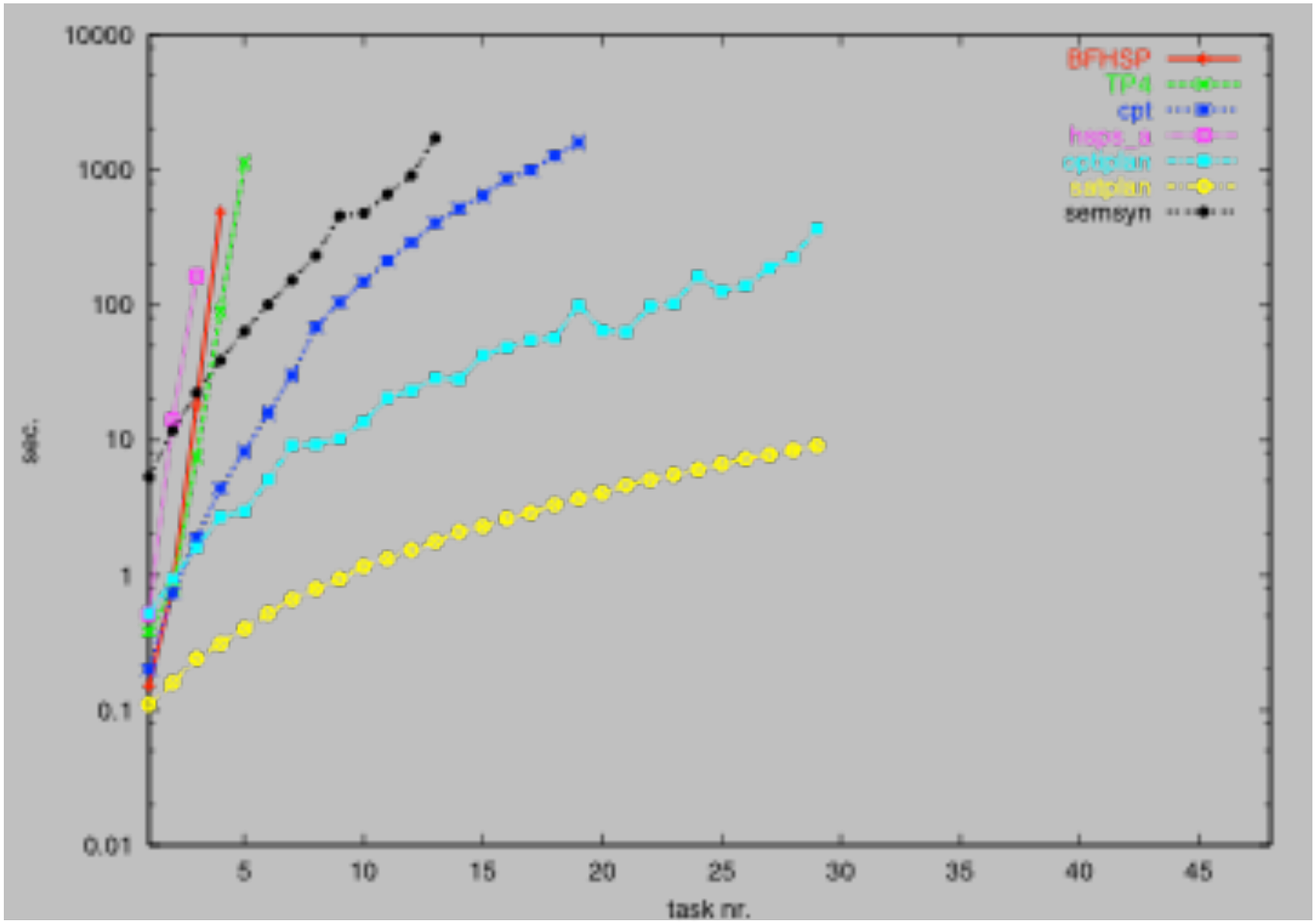
The IPC-4 Domains

- Airport: control the ground traffic [Hoffmann & Trüg]
- Pipesworld: control oil product flow in a pipeline network [Liporace & Hoffmann]
- Promela: find deadlocks in communication protocols [Edelkamp]
- PSR: resupply lines in a faulty electricity network [Thiebaux & Hoffmann]
- Satellite & Settlers [Fox & Long], additional Satellite versions with time windows for sending data [Hoffmann]
- UMTS: set up applications for mobile terminals [Edelkamp & Englert]

The Competitors: Optimal planners

optimal								
team repres.	login	planner	adl	fl	da	:m	til	dp
M) V. Vidal	cpt	16) cpt	-	-	+	-	-	?
N) P.Haslum	patrick	17) TP4	-	lim	+	-	-	-
N) P.Haslum	patrick	18) hsps_a	-	lim	+	-	-	-
O) H.Kautz	satplan	19) satplan	-	-	-	-	-	?
P) M.Briel	optiplan	20) optiplan	-	-	-	-	-	-
Q) E.Parker	eriqueparquer	21) semsyn	+	-	-	-	-	-
R) R.Zhou	zhou	22) BFHSP	-	-	-	-	-	-

Dining Philosophers



IPC 2004

International Planning Competition

Hosted at
International Conference on Automated Planning and Scheduling

Performance Award:
1st Prize, Optimal Track

Henry Kautz, David Roznyai, Farhad Teydaye-Saheli, Shane Neth and Michael Lindmark

$ 200

"SATPLAN04"

Whistler, June 6, 2004

Stefan Edelkamp *Jörg Hoffmann*
IPC-4 Co-Chairs Classical Part

Blackbox Demo

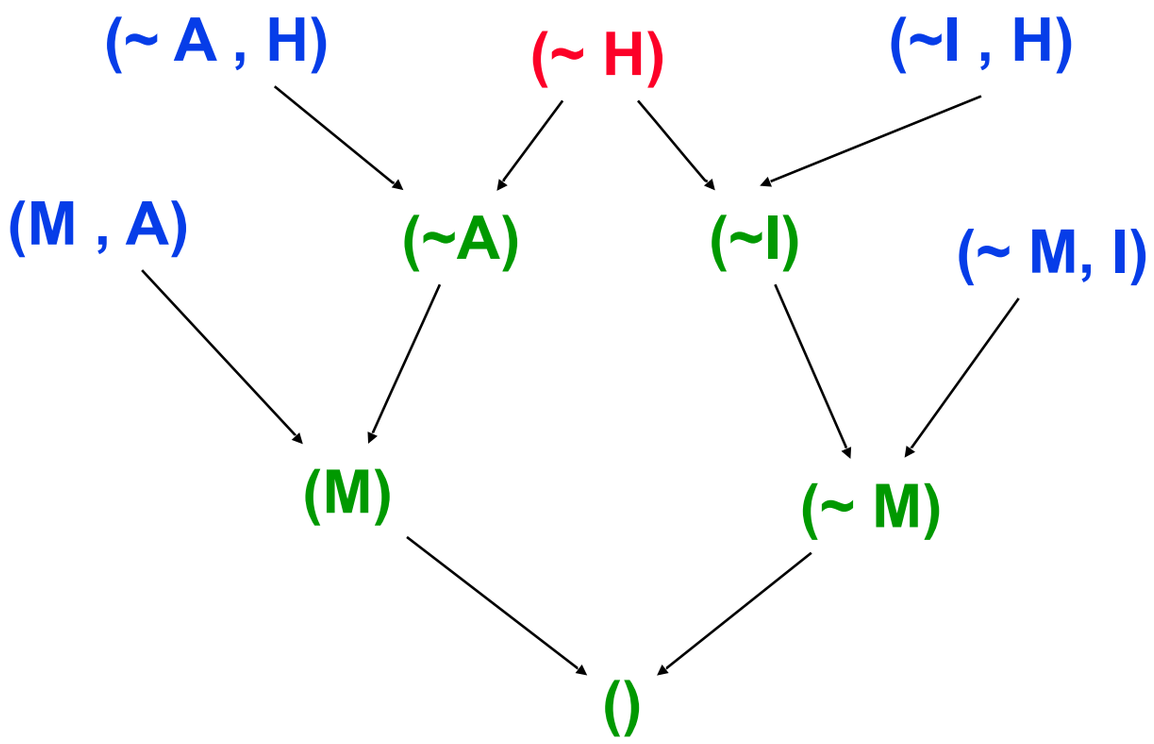
SAT Algorithms

Resolution Refutation Proof

DAG, where leaves are input clauses

Internal nodes are resolvants

Root is false (empty clause)



KB:

- If the unicorn is mythical, then it is immortal,
- if it is not mythical, it is an animal
- If the unicorn is either immortal or an animal, then it is horned.

Prove: the unicorn is horned.

Efficient Backtrack Search for Satisfiability Testing

Basic Backtrack Search for a Satisfying Model

```
Solve( F ): return Search(F, { });

Search( F, assigned ):
   if all variables in F are in assigned then
        if assigned |= F then return assigned;
        else return FALSE;
   choose unassigned variable x;
   return  Search(F, assigned U {x=0}) ||
           Search(F, assigned U {x=1});
end;
```

Is this algorithm complete?

What is its running time?

Basic Backtrack Search for a Satisfying Model

```
Solve( F ): return Search(F, { });

Search( F, assigned ):
   if all variables in F are in assigned then
        if assigned |= F then return assigned;
        else return FALSE;
   choose unassigned variable x;
   return  Search(F, assigned U {x=0}) ||
           Search(F, assigned U {x=1});
end;
```

Is this algorithm complete? YES

What is its running time?

Basic Backtrack Search for a Satisfying Model

```
Solve( F ): return Search(F, { });

Search( F, assigned ):
   if all variables in F are in assigned then
        if assigned |= F then return assigned;
        else return FALSE;
   choose unassigned variable x;
   return  Search(F, assigned U {x=0}) ||
           Search(F, assigned U {x=1});
end;
```

Is this algorithm complete? YES

What is its running time? $O(2^n)$ and $o(2^n)$

Propagating Constraints

- Suppose formula contains

  (A v B v ~C)

  and we set A=0.
- What is the resulting constraint on the remaining variables B and C?

  (B v ~C)
- Suppose instead we set A=1. What is the resulting constraint on B and C?

  No constraint

Empty Clauses and Formulas

- Suppose a clause in F is shortened until it become empty. What does this mean about F and the partial assignment?

  F cannot be satisfied by any way of completing the assignment; must backtrack

- Suppose all the clauses in F disappear. What does this mean?

  F is satisfied by any completion of the partial assignment

Better Backtrack Search

```
Search( F, assigned ):
   if F is empty then return assigned;
   if F contains [ ] then return FALSE;
   choose an unassigned variable c
   return Search(F • c, assigned U {c})   ||
          Search(F • ~c , assigned U {~c});
end
```

F • L = remove clauses from F that contain literal L, and
shorten clauses in F that contain ~L

Unit Propagation

- Suppose a clause in F is shortened to contain a single literal, such as

  (L)

  What should you do?

  Immediately add the literal to assigned.

  This may shorten some clauses and erase other clauses.

  Repeat if another single-literal clause appears.

Even Better Backtrack Search

```
Search( F, assigned ):
   if F is empty then return assigned;
   if F contains [ ] then return FALSE;
   if F contains a unit clause [L] then
         return Search(F • L, assigned U {L})
    else
         choose an unassigned variable c
        return Search(F • c, assigned U {c})   ||
               Search(F • ~c , assigned U {~c});
end
```

F• L = remove clauses from F that contain literal L, and
shorten clauses in F that contain ~L

Pure Literal Rule

- Suppose a literal L appears in F, but the negation of L never appears. What should you do?

  Immediately add the literal to assigned.
  This will erase some clauses, but not shorten any.

Davis-Putnam-Logemann-Loveland Procedure (DPLL)

```
DPLL( F, assigned ):
   if F is empty then return assigned;
   if F contains [ ] then return FALSE;
   if F contains a unit clause [L] or a pure literal L then
         return Search(F • L, assigned U {L})
    else
         choose an unassigned variable c
        return Search(F • c, assigned U {c})   ||
               Search(F • ~c , assigned U {~c});
end
```

F• L = remove clauses from F that contain literal L, and
shorten clauses in F that contain ~L

DPLL on the Unicorn

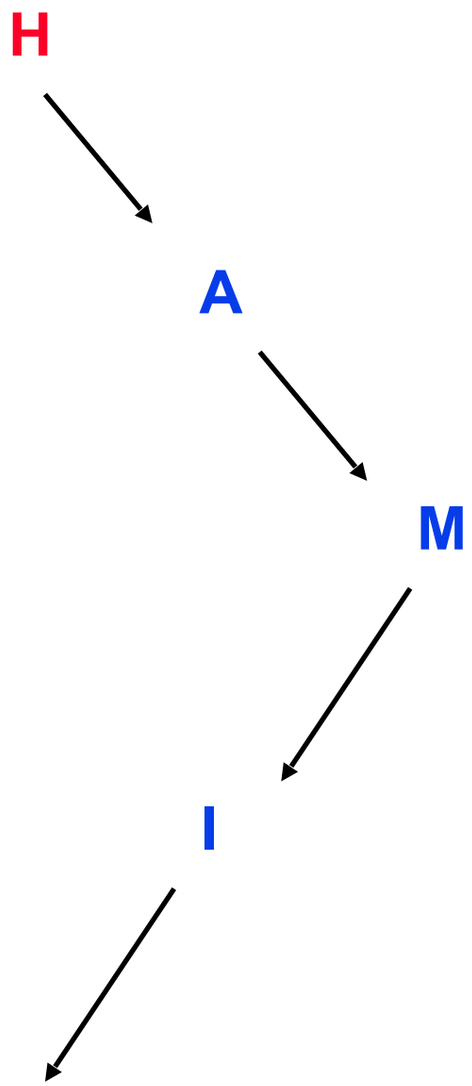
(~ M, I) (I)

(M , A) (M)

(~I , H) (~I) ( )

(~ A , H) (~A)

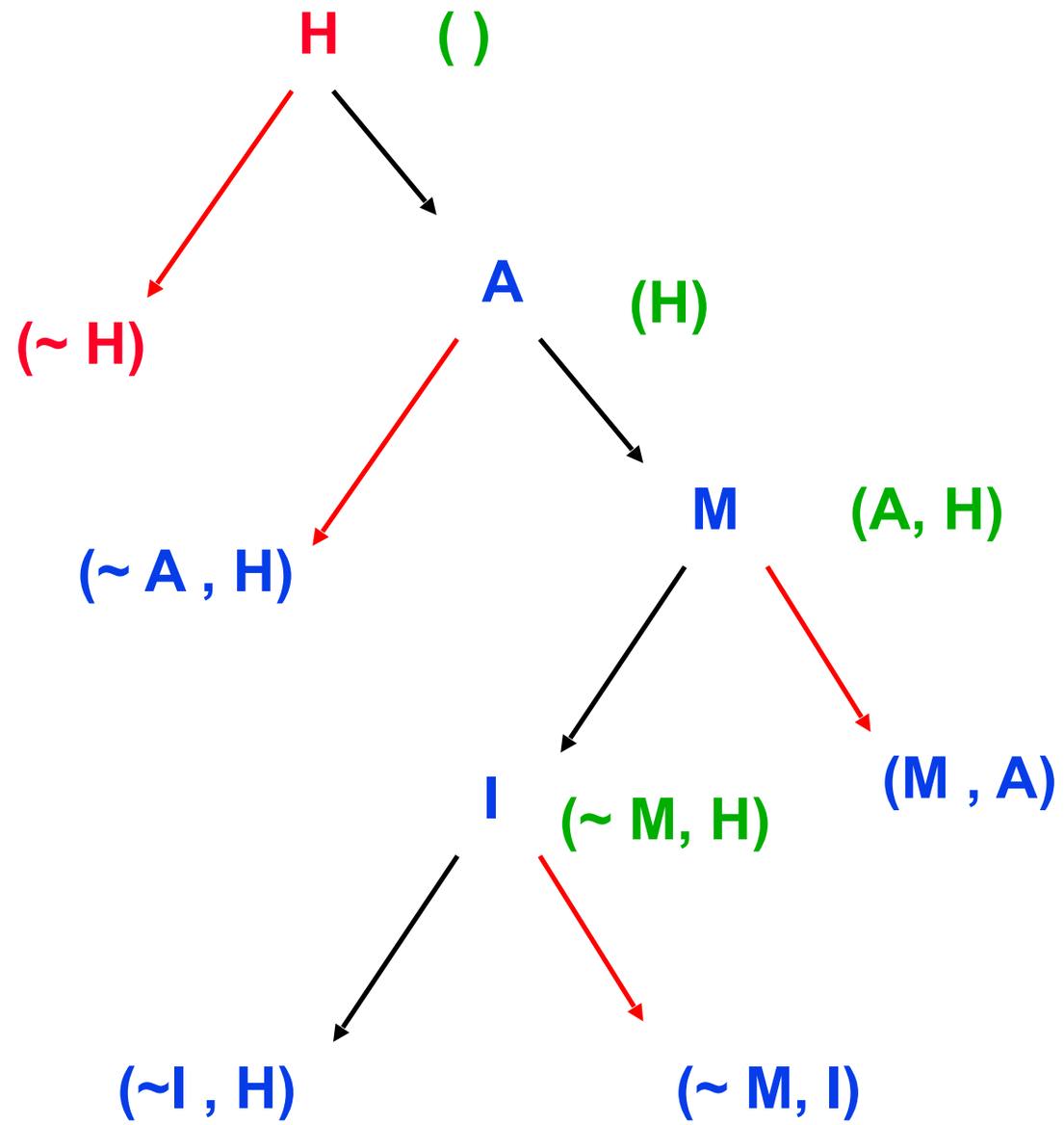
(~ H)



NO SEARCH!

Converting DPLL Tree to a Resolution Proof

- Add missing branches
- Attach clauses to leafs
- Label interior nodes with resolution of children



DPLL and Resolution

- DPLL is thus computational equivalent to creating a tree-shaped resolution proof
- In theory, since resolution is not restricted to tree-shaped proofs, it should be "better"
- In practice, the overhead of resolution makes it much worse

Scaling Up

- For decades, DPLL was considered only useful for "toy" problems
- Starting around 1996, researchers improved DPLL using
  - Good heuristics for choosing variable for branching
  - Caching
  - Clever Data Structures
- Today, modern versions of DPLL are used to solve big industrial problems in hardware and software verification, automated planning and scheduling, cryptography, and many other areas

What is BIG?

Consider a real world Boolean Satisfiability (SAT) problem

The instance bmc-ibm-6.cnf, IBM LSU 1997:

```
p cnf 5
-1 7 0
-1 6 0
-1 5 0
-1 -4 0
-1 3 0
-1 2 0
-1 -8 0
-9 15 0
-9 14 0
-9 13 0
-9 -12 0
-9 11 0
-9 10 0
-9 -16 0
-17 23 0
-17 22 0
```

I.e., ((not x_1) or x_7)
((not x_1) or x_6)
etc.

x_1, x_2, x_3, etc. our Boolean variables (set to True or False)

Set x_1 to False ??

10 pages later:

```
    1
185 −9 0
185 −1 0
177 169 161 153 145 137 129 121 113 105 97
 89 81 73 65 57 49 41
 33 25 17 9 1 −185 0
186 −187 0
186 −188 0
```

…

I.e., (x_177 or x_169 or x_161 or x_153 …
x_33 or x_25 or x_17 or x_9 or x_1 or (not x_185))

clauses / constraints are getting more interesting…

Note x_1 …

4000 pages later:

10236 −10050 0
10236 −10051 0
10236 −10235 0
10008 10009 10010 10011 10012 10013 10014
10015 10016 10017 10018 10019 10020 10021
10022 10023 10024 10025 10026 10027 10028
10029 10030 10031 10032 10033 10034 10035
10036 10037 10086 10087 10088 10089 10090
10091 10092 10093 10094 10095 10096 10097
10098 10099 10100 10101 10102 10103 10104
10105 10106 10107 10108 −55 −54 53 −52 −51 50
10047 10048 10049 10050 10051 10235 −10236 0
10237 −10008 0
10237 −10009 0
10237 −10010 0

...

Finally, 15,000 pages later:

−7 260 0
7 −260 0
1072 1070 0
−15 −14 −13 −12 −11 −10 0
−15 −14 −13 −12 −11 10 0

CSC 244 Logical Foundations of Artificial Intelligence

−7 −6 −5 −4 3 −2 0
−7 −6 −5 −4 3 2 0
185 0

Search space of truth assignments:

HOW?

$$2^{50000} \approx 3.160699437 \cdot 10^{15051}$$

Current SAT solvers solve this instance in approx. 1 minute!

Local Search Strategies

Greedy Local Search

```
state = choose_start_state();
while ! GoalTest(state) do
      state := arg min { h(s) | s in Neighbors(state) }
end
return state;
```

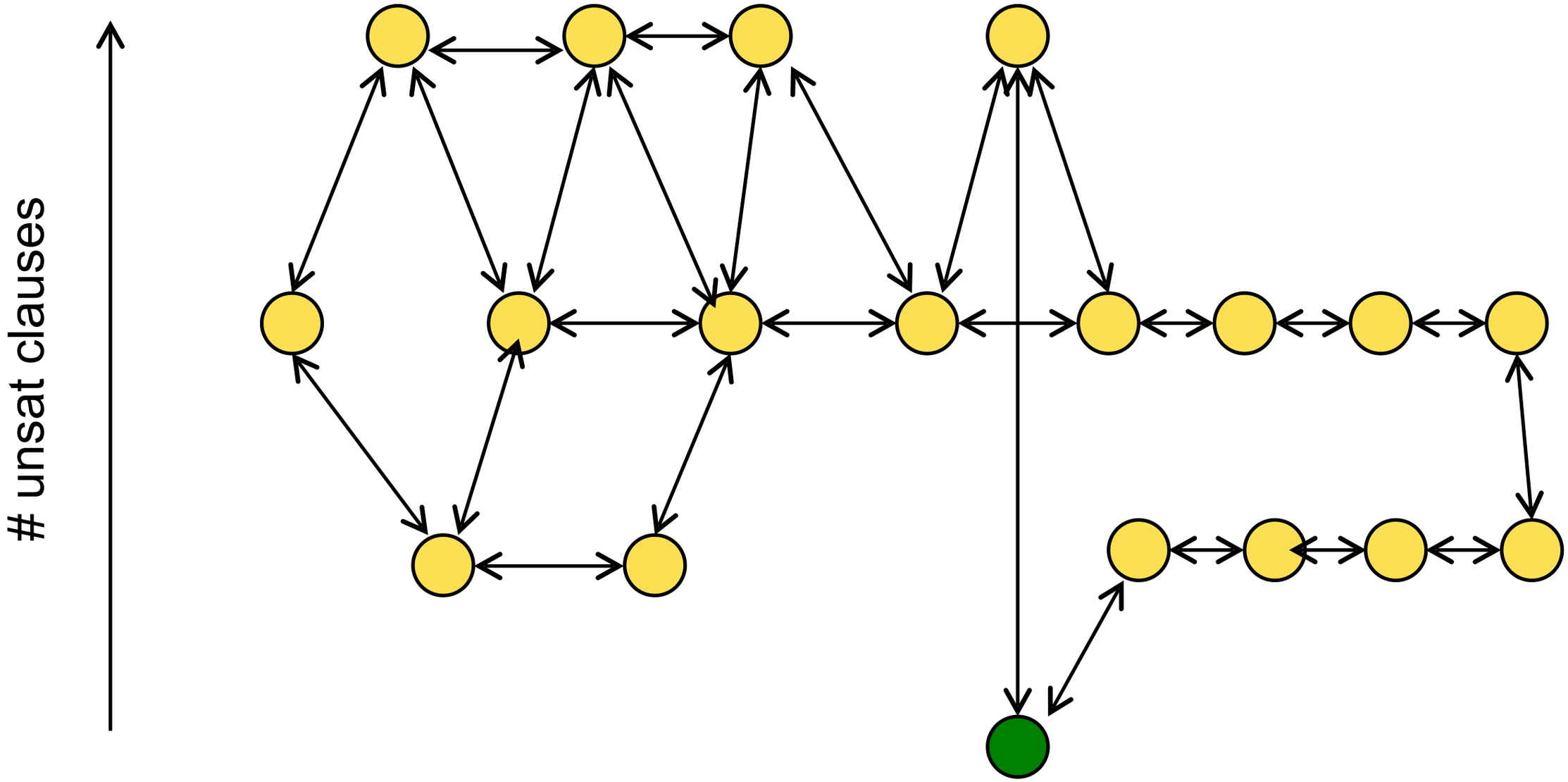
- Terminology:
  - “neighbors” instead of “children”
  - heuristic h(s) is the “objective function”, no need to be admissible
- No guarantee of finding a solution
  - sometimes: probabilistic guarantee
- Best goal-finding, not path-finding
- Many variations

Greedy Local Search for SAT

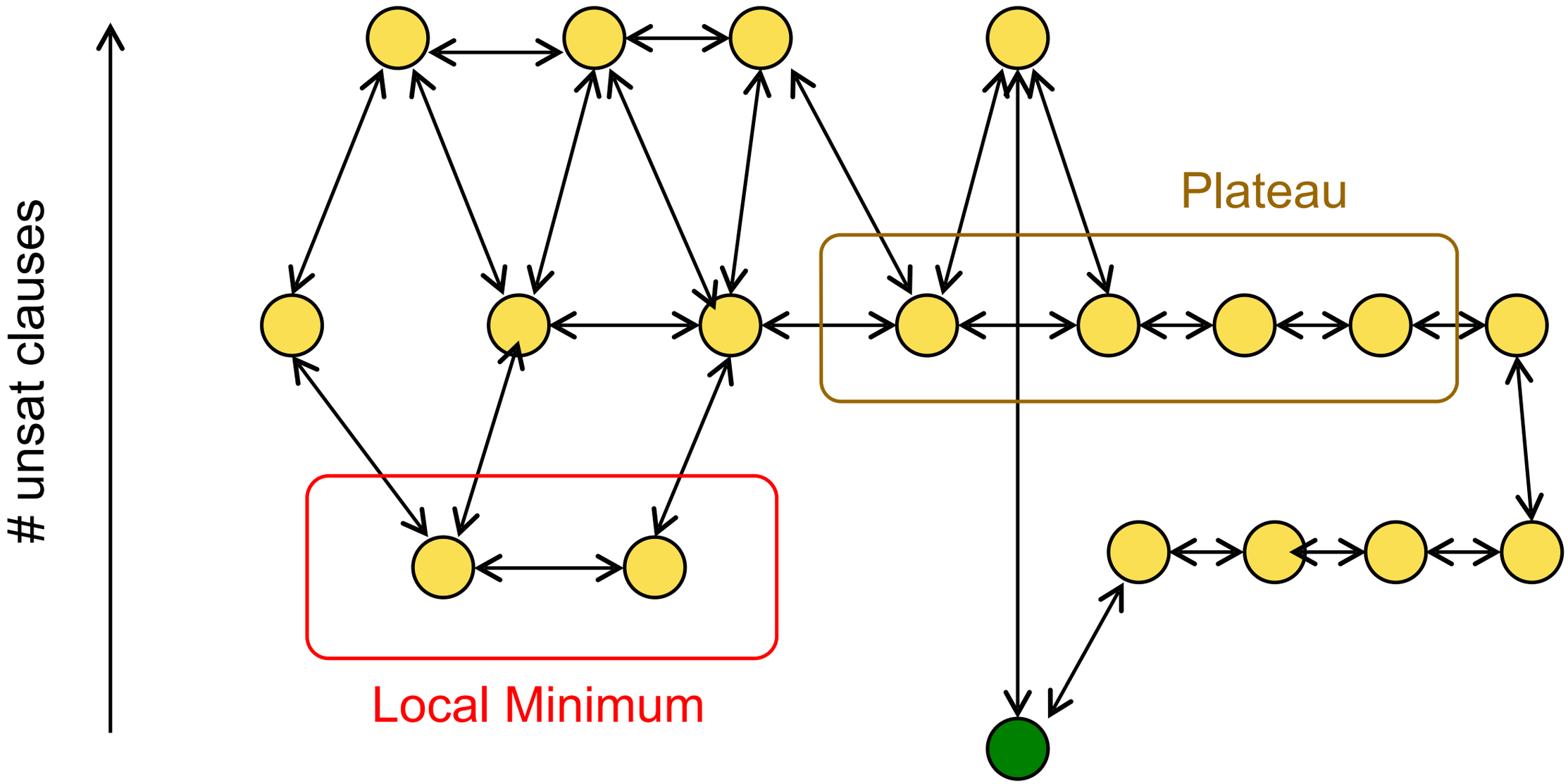
```
state = choose_start_state();
while ! GoalTest(state) do
      state := arg min { h(s) | s in Neighbors(state) }
end
return state;
```

- start = random truth assignment
- GoalTest = formula is satisfied
- h = number of unsatisfied clauses
- neighbors = flip one variable

Local Search Landscape



Local Search Landscape



Variations of Greedy Search

- Where to start?
  - RANDOM STATE
  - PRETTY GOOD STATE
- What to do when a local minimum is reached?
  - STOP
  - KEEP GOING
- Which neighbor to move to?
  - (Any) BEST neighbor
  - (Any) BETTER neighbor
- How to make greedy search more robust?

Restarts

```
for run = 1 to max_runs do
   state = choose_start_state();
   flip = 0;
   while ! GoalTest(state) && flip++ < max_flips do
        state := arg min { h(s) | s in Neighbors(state) }
   end
   if GoalTest(state) return state;
end
return FAIL
```

Uphill Moves: Random Noise

```
state = choose_start_state();
while ! GoalTest(state) do
    with probability noise do
            state = random member Neighbors(state)
    else
            state := arg min { h(s) | s in Neighbors(state) }
    end
end
return state;
```

Random Walk for SAT

- Observation: if a clause is unsatisfied, at least one variable in the clause must be different in any global solution

(A v ~B v C)

- Suppose you randomly pick a variable from an unsatisfied clause to flip. What is the probability this was a good choice?

Random Walk for SAT

- Observation: if a clause is unsatisfied, at least one variable in the clause must be different in any global solution

(A v ~B v C)

- Suppose you randomly pick a variable from an unsatisfied clause to flip. What is the probability this was a good choice?

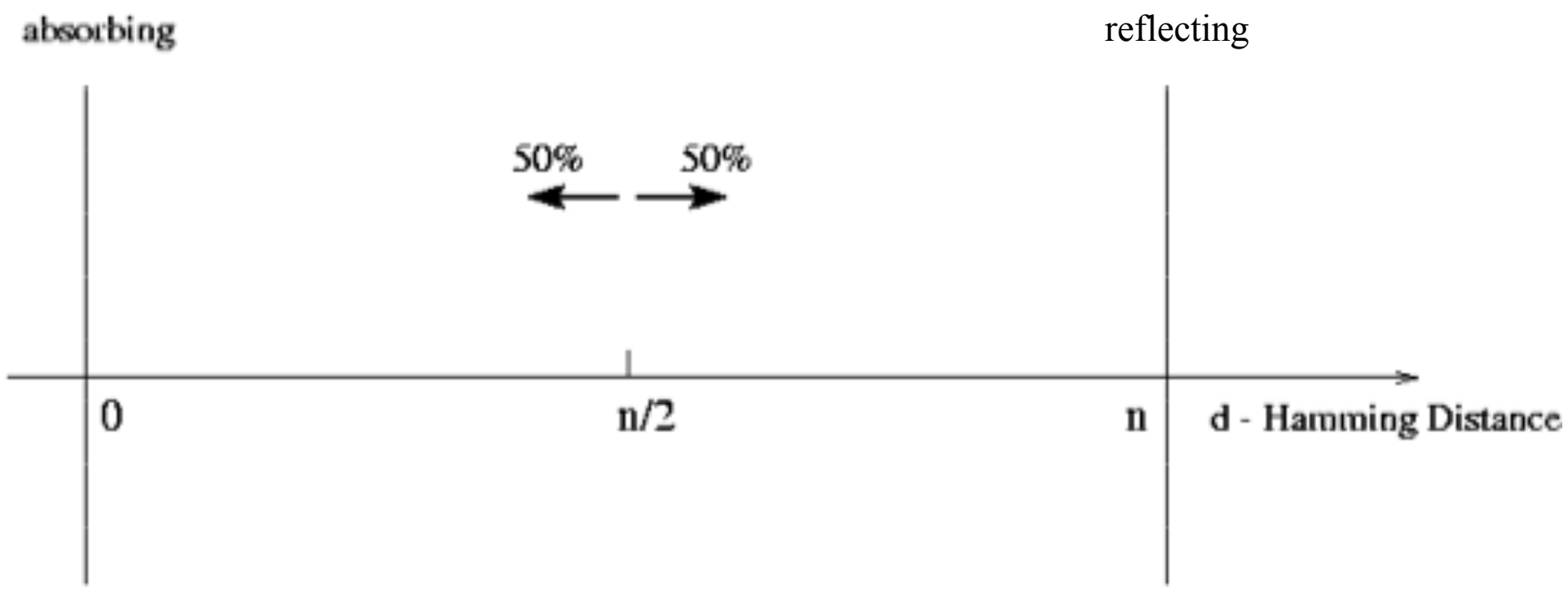
$$\Pr(\text{good choice}) \geq \frac{1}{\text{clause length}}$$

Random Walk Local Search

```
state = choose_start_state();
while ! GoalTest(state) do
    clause := random member { C | C is a clause of F and
                              C is false in state }
    var := random member { x | x is a variable in clause }
    state[var] := 1 – state[var];
end
return state;
```

Properties of Random Walk

- If clause length = 2:
  - 50% chance of moving in the right direction
  - Converges to optimal with high probability in $O(n^2)$ time



Greedy Random Walk

```
state = choose_start_state();
while ! GoalTest(state) do
    clause := random member { C | C is a clause of F and
                                  C is false in state };
    with probability noise do
        var := random member { x | x is a variable in clause };
    else
        var := arg_min(x) { #unsat(s) | x is a variable in clause,
                                s and state differ only on x};
    end
    state[var] := 1 – state[var];
end
return state;
```

Coming Up

- Pick up solution to Homework 3
- Right after break: Exam 2 Logic
  - based on Homework 3
- If you did not get your Exam 1 back, pick up a solution sheet
- Phase II Othello players due after break!