Propositional formula: true or false. **Strict def**: need brackets (outer; not…). Removing some => abbrev. of genuine formulas. Meaningless: not strictly well-formed (def). e.g., p or q and r

i. Any propositional atom—p, q, r, etc.—is a formula.
ii. $\top$ and $\perp$ are formulas.
iii. If A is a formula then so is $(\neg A)$.
iv. If A, B are formulas, so are $(A \wedge B)$, $(A \vee B)$, $(A \to B)$, $(A \leftrightarrow B)$.
v. Nothing is a formula unless built by (i)–(v).

binary connectives: right-associative: p->q->r is p->(q->r). Conj/disj: associative

$\neg(A \to B) \equiv A \wedge \neg B$

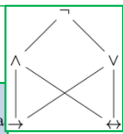
$A \to B \equiv \neg(A \wedge \neg B)$

$A \to B \equiv \neg A \vee B$ *

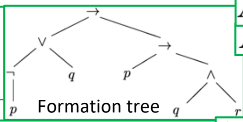
De Morgan Laws

$\neg(A \wedge B) \equiv \neg A \vee \neg B$

$\neg(A \vee B) \equiv \neg A \wedge \neg B$



Subformulas (!= substrings): subtrees but no repeated occurrences



Formation tree

- A formula that is either atomic or negated-atomic is a *literal*.
- A *clause* is a formula $L_1 \vee \cdots \vee L_n$ $(n \geqslant 1)$, where each L_i is a literal.

Verum/Falsum are atomic formulas but not atoms. An atom p is an atomic formula. Not p, Not falsum, verum etc. are literals and clauses

$(p \vee q) \vee r$ is *not* technically a clause. It would have to be of the form $L_1 \vee L_2 \vee L_3$, which by the right-associativity of '$\vee$', is $L_1 \vee (L_2 \vee L_3)$. So, the brackets are in the wrong place. But $(p \vee q) \vee r$ is logically equivalent to $p \vee (q \vee r)$, which *is* a clause.

$A \leftrightarrow B \equiv (A \to B) \wedge (B \to A)$

$A \leftrightarrow B \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$

$A \leftrightarrow B \equiv \neg A \leftrightarrow \neg B$

$\neg(A \leftrightarrow B) \equiv A \leftrightarrow \neg B$

$\neg(A \leftrightarrow B) \equiv \neg A \leftrightarrow B$

$\neg(A \leftrightarrow B) \equiv (A \wedge \neg B) \vee (\neg A \wedge B)$

$h_v(A \to B) = \mathbf{t}$ iff $h_v(A) = \mathbf{f}$ or $h_v(B) = \mathbf{t}$

$h_v(A \leftrightarrow B) = \mathbf{t}$ iff $h_v(A) = h_v(B)$

Given $A_1, \ldots, A_n, B \in fmlas_{\mathcal{A}}$, an argument

'$A_1, \ldots, A_n$, therefore B'

is *propositionally valid* if, for any atomic evaluation function v:

if $h_v(A_1) = \mathbf{t}, \ldots, h_v(A_n) = \mathbf{t}$, then $h_v(B) = \mathbf{t}$.

In this case, we write $A_1, \ldots, A_n \models B$.

Modus **ponens**: $A, A \to B \models B$

Modus **tollens**: $A \to B, \neg B \models \neg A$

XOR: $(A \vee B) \wedge \neg(A \wedge B)$

Set-notation for CNF: $\emptyset$ is falsum if "empty clause", and verum if "empty conjunction"

CNF / DNF:

A formula of propositional logic is in *conjunctive normal form* if it is a conjunction of clauses—e.g., $(p \vee \neg q) \wedge (r \vee \neg p)$.

A formula is in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions—e.g., $(p \wedge \neg q) \vee (r \wedge \neg p)$.

Special cases of CNF/DNF:
1) CNF: conjunction of literals; conjunction with a single clause; not(p or q) is not a clause so not a conjunction;
2) DNF: disjunction with a single conjunction (of clauses, or of one clause);
Note you should NOT use log. equivalent clauses to determine if in CNF/DNF.
A tautology can appear as a clause in a CNF: e.g., "p or not p"

Resolution = inference rule taking 2 clauses -> produce the 3rd clause implied

argument	validity	satisfiability	equivalence
$A_1, \ldots, A_n \models B$	$(\bigwedge A_i) \to B$ valid	$(\bigwedge A_i) \wedge \neg B$ unsat.	$((\bigwedge A_i) \to B) \equiv \top$
$\top \models A$	A valid	$\neg A$ unsat.	$A \equiv \top$
$A \not\models \perp$	$\neg A$ not valid	A satisfiable	$\neg A \not\equiv \top$
$A \models B$ and $B \models A$	$A \leftrightarrow B$ valid	$A \leftrightarrow \neg B$ unsat.	$A \equiv B$

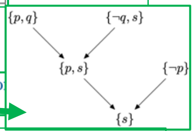
Use set-notation in resolution steps

Let C_1 and C_2 be clauses such that, for some atom p, then $p \in C_1$ and $\neg p \in C_2$. The *resolvent of* C_1 *and* C_2 *on* p is the clause

$$(C_1 \setminus \{p\}) \cup (C_2 \setminus \{\neg p\}).$$

We write this as $resolvent(C_1, C_2, p)$.

Single steps of resolution are chained together to form a resolution derivation, which we represent as a tree.



Resolution is sound: If $C = resolvent(C_1, C_2, p)$, then $\{C_1, C_2\} \models C$
(SLD in def.log programs too)

Or we write: $S \vdash_{\mathrm{res(PL)}} C$, where S is the CNF with clauses C1 and C2.

Resolution and SAT: **PL Resolution**

$S \vdash_{\mathrm{res(PL)}} \emptyset$ iff $S \models \perp$ (refutation-soundness + -completeness)

We know S is satisfiable iff $S \not\models \perp$

Algorithm 2.1 DP(S : a set of clauses)

Davis-Putnam Algo (BFS)

```
while ⊤:
    if ∅ in S:
        return UNSATISFIABLE
    S = {C | C ∈ S, ¬tautology(C)}
    if there is a clause {l} in S:
        S = {C \ {l̄} | C ∈ S, ¬(l ∈ C)}
        continue
    elif there is a pure literal l:
        S = {C | C ∈ S, ¬(l ∈ C)}
        continue
    elif there is p ∈ A s.t. p ∈ C1 and ¬p ∈ C2 for C1, C2 ∈ S:
        R = {resolvent(C1, C2, p) | C1 ∈ S, C2 ∈ S}
        S = {C | C ∈ S, p ∉ C, ¬p ∉ C} ∪ R
        continue
```

PURE RULE: remove clauses containing pure literals. **UNIT PROP**: remove unit clauses and remove the compl. literal *from* remaining clauses

Algorithm 2.2 DLL(S : a set of clauses)

Davis-Logemann-Loveland Algo (DLL or DPLL) (DFS)

```
while ⊤:
    if ∅ in S:
        return UNSATISFIABLE
    S = {C | C ∈ S, ¬tautology(C)}
    if there is a clause {l} in S:
        S = {C \ {l̄} | C ∈ S, ¬(l ∈ C)}
        continue
    elif there is a pure literal l:
        S = {C | C ∈ S, ¬(l ∈ C)}
        continue
    elif there is p ∈ A s.t. p ∈ C1 and ¬p ∈ C2 for C1, C2 ∈ S:
        if DLL({C \ {¬p} | C ∈ S, ¬(p ∈ C)}) is SATISFIABLE:
            return SATISFIABLE
        else
            return DLL({C \ {p} | C ∈ S, ¬(¬p ∈ C)})
return SATISFIABLE
```

In **FOL**, **Constant**: name for an object. **Predicate**: properties/rel. between objects. **FS**: map objects to other objects. Can use "=" in FOL formulas.

Ground term: contains no variable (!= unground term). Terms have NO truth

FOL syntactic errors (not formulas): NO predicates or connectives inside predicates/FS; be clear when predicate/FS, as predicate can have truth value, but FS (term) cannot.

A *signature for first-order logic* is a tuple $(\mathcal{C}, \{\mathcal{P}_i\}_{i \in \mathbb{N}}, \{\mathcal{F}_i\}_{i \in \mathbb{N}^+})$: signature L
- $\mathcal{C}$ is a countable set of *constants*.
- Each $\mathcal{P}_i$ is a countable set of *predicate symbols* of arity i. (The members of $\mathcal{P}_0$ are also called *(propositional) atoms*.)
- Each $\mathcal{F}_i$ is a countable set of *function symbols* of arity i.

Let $\mathcal{L}$ be a signature. The $\mathcal{L}$-terms include only: L-terms
i. Any constant in $\mathcal{L}$.
ii. Any variable. NO predicates
iii. For any function-symbol $f \in \mathcal{F}_n$, and any $\mathcal{L}$-terms $t_1, \ldots, t_n$, the expression $f(t_1, \ldots, t_n)$.

An occurrence of a variable X is **bound** if it lies under a quantifier in the formation tree (otherwise **free**). **Sentence:** formula with no free variable + has truth in a structure.

Meaning phi of a predicate p is set of objects/tuples in D that p applies to (if 0-arity, phi(p)=$\emptyset$)

Let $\mathcal{L}$ be a signature. The $\mathcal{L}$-formulas include only: L-formulas
i. For all $P \in \mathcal{P}_n$, and $\mathcal{L}$-terms $t_1, \ldots, t_n$, the expression $P(t_1, \ldots, t_n)$. (i)-(iii) = atomic
ii. For all $\mathcal{L}$-terms t_1 and t_2, the expression $(t_1 = t_2)$.
iii. $\top$ and $\perp$.
iv. For all $\mathcal{L}$-formulas A and B: the expressions $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \to B)$, and $(A \leftrightarrow B)$.
v. For all $\mathcal{L}$-formulas A and variables X, the expressions $(\forall X A)$ and $(\exists X A)$. Quantifiers bind as strongly as "not" in PL

Let $\mathcal{L}$ be a signature of FOL. An $\mathcal{L}$-structure is a pair $M = (D, \varphi)$: L-structure
- D is a non-empty set.
- For each constant $c \in \mathcal{C}$, $\varphi(c) \in D$.
- For each predicate $p \in \mathcal{P}_n$, $\varphi(p) \subseteq D^n$. $(D^0 = \{\emptyset\}, D^1 = D.)$
- For each $f \in \mathcal{F}_n$, $\varphi(f)$ is a function from D^n to D.

dom_M is D, and φ_M is φ.

If X does not occur free (or at all) in B, then: $\forall X(A \to B) \equiv \exists X A \to B$

If X does not occur free (or at all) in B, then: $\exists X(A \to B) \equiv \forall X A \to B$

Variable assignment under M with sigma function (V: D) to give **truth** value to FOL formulas with **free** variables.
X-variant to give **truth** value to FOL formula with **bound** X

$M, \sigma \models \exists X\, A$ iff $M, \sigma' \models A$ for some X-variant of σ, σ';

$M, \sigma \models \forall X\, A$ iff $M, \sigma' \models A$ for every X-variant of σ, σ'.

An $\mathcal{L}$-formula A is (logically) valid if for every $\mathcal{L}$-structure M and assignment σ into M, we have $M, \sigma \models A$. We write this: $\models A$

$\forall X(novel(X) \wedge wrote(hj, X) \to likes(aj, X))$,
$wrote(hj, amb)$, $novel(amb)$,
$\models likes(aj, amb)$

Take any $M = (D, \varphi)$, and assume, for arbitrary σ, that M and σ make all the premises true. Let σ' be that X-variant of σ which assigns to X the object $\varphi(amb)$. We must have $M, \sigma' \models novel(X) \wedge wrote(hj, X) \to likes(aj, X)$ by the semantics of $\forall X$ (Def. 3.9).

Since $M, \sigma \models wrote(hj, amb)$ and $M, \sigma \models novel(amb)$, then $\varphi(amb) \in \varphi(novel)$ and $(\varphi(hj), \varphi(amb)) \in \varphi(wrote)$. So $M, \sigma' \models novel(X) \wedge wrote(hj, X)$, by the semantics of $\wedge$. By the semantics of $\to$, then $M, \sigma' \models likes(aj, X)$.

From this, since $\sigma'(X) = \varphi(amb)$, it must be that $M, \sigma \models likes(aj, amb)$. Direct reasoning

So our argument is valid.

Let σ_1 and σ_2 be assignments and X a variable. Then σ_2 is an X-variant of σ_1 if $\sigma_1(Y) = \sigma_2(Y)$ for all variables Y other than X. We write this as $\sigma_1 =_X \sigma_2$.

Note: it's possible that σ_1 and σ_2 agree on X too! (So any assignment σ is a X-variant of itself, for any X.)

Note: $\sigma_1 =_X \sigma_2$ does not imply $\sigma_1 = \sigma_2$.

The use of resolution in SLD means that we are checking whether $\mathsf{P} \cup \{\neg Q\} \models \perp$. This is the case when $\mathsf{P} \cup \{\neg Q\}$ has no model. *

Let $\mathcal{L}$ be a signature. An *atom* is an expression $p(t_1, \ldots, t_n)$ for terms $t_1, \ldots, t_n$ and $p \in \mathcal{P}_n$. (Members of $\mathcal{P}_0$ are atoms.)

A *literal* is an atom or a negated atom. Verum/falsum: NOT lit.

A *clause* is an $\mathcal{L}$-formula

$$\forall X_1 \cdots \forall X_m (C_1 \vee \cdots \vee C_n)$$

where $C_1, \ldots, C_n$ are literals and $X_1, \ldots, X_m$ are all the (free) variables in $C_1, \ldots, C_n$.

atom, lit., clause in Definite Logic Program

Substitution and unification for SLD (FOL)

Let F_1 and F_2 be formulas and let θ be a unifier for them. θ is a *most general unifier* (mgu) for F_1 and F_2 if, for all unifers θ' of F_1 and F_2, $F_1\theta'$ is an instance of $F_1\theta$.

Let F_1 and F_2 be formulas. A *unifier of* F_1 *and* F_2 is any substitution θ such that $F_1\theta = F_2\theta$.

If there is a unifier of F_1 and F_2, then they are *unifiable*.

The *application* of θ to an $\mathcal{L}$-formula F is the result of substituting each *free* occurrence of the variable U_i in F by the term t_i. We denote this as $F\theta$, and $F\theta$ is known as an *instance* of F.

θ is *idempotent* if, for all formulas F, $(F\theta)\theta = F\theta$.

FOL resolution: unification by substitution. Start with a query, replace one of the negated atoms in the query by the body of a rule in P whose head is that atom (% unification). Resolvent of one of P clause w/ query is always a new **query**! derived empty cl. => **refutation of goal <-Q** (not Q), so **P|=Q. OR, use M(P)**

Definite clause (DC): clause w/ exactly 1 positive literal. DC with no negative lit. (fact), otherwise (rule). **Logic program. notation**
Def.Log.Program P = set of DCs

$$A \leftarrow B_1, \ldots, B_n$$

$$\forall X_1 \cdots \forall X_m (A \vee \neg B_1 \vee \cdots \vee \neg B_n)$$

Computed answer substitution (CAS): Mgus theta_1 **o** theta_m (root to success), **theta***. **Computed answer**: original query w/ variables substituted using the CAS

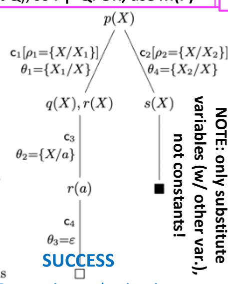
Consider the program P:

$\mathsf{c}_1 : p(X) \leftarrow q(X), r(X).$
$\mathsf{c}_2 : p(X) \leftarrow s(X).$
$\mathsf{c}_3 : q(a).$
$\mathsf{c}_4 : r(a).$

The *resolution search tree* for a query $\leftarrow p(X)$ is shown. $(\mathsf{P} \models \exists X p(X)?)$

The empty clause (written $\square$) is derived with c_1, c_3 and c_4. (ε is the empty substitution, i.e., $\emptyset$.)

Resolving the query $p(X)$ using c_2 produces $s(X)$. This cannot be resolved further; this failure is marked as $\blacksquare$.



NOTE: only substitute variables (w/ other var.), not constants!

Renaming substitution p

A *query* is a clause in which all literals are negated atoms.

Queries therefore have the form $\forall X_1 \cdots \forall X_m(\neg Q_1 \vee \cdots \vee \neg Q_n)$ for atomic Q_i.

Herbrand Interprt (HI) = FOL structure where constants and terms made with FS name *themselves* (phi function), + D is the set of ground L-terms. *

A *Herbrand model of* a first-order sentence F is any Herbrand interpretation M such that $M \models F$.

Use of HI = alternative to FOL SLD: **just check if Q in M(P)**

Let S be a set of sentences in Skolem normal form ($\forall F$, where F is quantifier-free). S has a model iff S has a Herbrand model.

We have simplified the task: we don't need to check truth in *all models* of P, just in *all Herbrand models* of P.

H models represented as bases

Let $\mathcal{L}$ be a signature. The *Herbrand base* for $\mathcal{L}$ is the set of ground $\mathcal{L}$-atoms. (We write this as $\mathrm{B}_{\mathcal{L}}$.)
L-atoms = predicates applied to objects
(Note that $\mathrm{B}_{\mathcal{L}}$ is infinite iff $\mathcal{C}$ is non-empty and there is at least one function-symbol in $\mathcal{L}$.)

A Herbrand model M is *minimal* if no proper subset of M is a model.

If a minimal Herbrand model M is unique, we call it the *least* Herbrand model.

Then $M \models \mathsf{P}$ iff for all $\mathsf{c} \in ground(\mathsf{P})$, if $body(\mathsf{c}) \subseteq M^H$ then $head(\mathsf{c}) \in M^H$. c = def. clause M = *a* Herbrand model of P

$\mathbf{M}(\mathsf{P}) = \bigcap\{M \mid M \text{ is a Herbrand model of } \mathsf{P}\}$

$\mathbf{M}(\mathsf{P})$ is the *least Herbrand model* of P.

Use of M(P):

$\mathbf{M}(\mathsf{P}) = \{q \in \mathrm{B}_{\mathcal{L}} \mid \mathsf{P} \models_{\mathrm{FOL}} q\}$

IF Q is a ground atom q

$I \subseteq \mathrm{B}_{\mathcal{L}}$

$T_{\mathsf{P}}(I) = \{head(\mathsf{c}) \mid \mathsf{c} \in ground(\mathsf{P}), body(\mathsf{c}) \subseteq I\}$

The Herbrand base is always a model of P (every P has at least one model) pathcost: root->n; h(n): n->goal (estimated)

1

```
function TREE-SEARCH(problem) returns solution or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure      /* a solution has not been returned earlier! */
    else choose a leaf node and remove it from the frontier
    if the node contains a goal state then return solution
    else expand the chosen node, add the resulting nodes to frontier
```

2

```
function GRAPH-SEARCH(problem) returns solution or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    else choose a leaf node and remove it from the frontier
    if the node contains a goal state then return solution
    else add the node to the explored set
         expand the chosen node, add the resulting nodes to frontier
         only if not in the frontier or explored set
```

7

```
function ITERATIVE-DEEPENING SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then
      return result
    end if
  end for
```

6

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then          /* checks if the initial state is a goal state */
    return SOLUTION(node)
  else if limit = 0 then
    return cutoff
  else
    cutoff_occurred? ← false
    for all action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)   /* generates children nodes */
      result ← RECURSIVE-DLS(child, problem, limit - 1)   /* recursive call on the children */
      if result = cutoff then                     /* checks if a cutoff has been returned */
        cutoff_occurred? ← true
      else if result ≠ cutoff then                /* if not, the result is returned */
        return result
      end if
    end for
    if cutoff_occurred? then                      /* if a cutoff occurred, then it returns a cutoff */
      return cutoff
    else                                          /* otherwise, the search failed */
      return failure
    end if
  end if
```

The *cutoff* value indicates no solution within the depth limit.

4

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop
    if EMPTY?(frontier) then
      return failure
    end if
    node ← POP(frontier)               /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then
      return SOLUTION(node)            /* checks if goal node */
    end if
    add node.STATE to explored
    for all action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child    /* checks for lower-cost path */
      end if
    end for
  end loop
```

Greedy BestFS: $f(n) = h(n)$

A*: $f(n) = \text{pathcost} + h(n)$

3

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then
    return SOLUTION(node)              /* checks if the initial state is a goal state */
  end if
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop
    if EMPTY?(frontier) then
      return failure                   /* checks if frontier is empty, in case returns failure */
    end if
    node ← POP(frontier)               /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for all action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)   /* generates children nodes */
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then
          return SOLUTION(CHILD)       /* checks children nodes */
        end if
        frontier ← INSERT(child, frontier)        /* add children nodes to frontier */
      end if
    end for
  end loop
```

Key issue: how to measure/compare performance of problem-solving algorithms?

- **Completeness**: does it find a solution whenever one exists?
- **Optimality**: does it find an optimal (i.e., cost-minimizing) solution?
- **Time complexity**: how much time is needed to find a solution?
- **Space complexity**: how much memory is needed to find a solution?

d = depth of optimal sol

Time and space complexity are expressed in terms of

- **branching factor** b: minimum number of child nodes (We assume $b \geq 2$) QUESTION Why?
- **depth** d: minimum number of steps to a solution from the initial state (i.e., distance of shallowest goal state)
- **maximum length** m of any path in the state space.

Time: number of nodes generated during search.

Space: maximum number of nodes stored in memory.

5

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then
    return SOLUTION(node)
  end if
  frontier ← a LIFO queue (aka stack) with node as the only element
  explored ← an empty set
  loop
    if EMPTY?(frontier) then
      return failure
    end if
    node ← POP(frontier)               /* now chooses the deepest node in frontier */
    add node.STATE to explored
    for all action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then
          return SOLUTION(CHILD)
        end if
        frontier ← INSERT(child, frontier)
      end if
    end for
  end loop
```

Search	Complete?	Time	Space	Optimal?
Breadth-first	Yes[1]	$O(b^d)$	$O(b^d)$	Yes[4]
Uniform-cost	Yes[1,2]	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	Yes
Depth-first	No[3]	$O(b^m)$	$O(bm)$	No
Depth-limited	No	$O(b^\ell)$	$O(b\ell)$	No
Iterative-deepening	Yes[1]	$O(b^d)$	$O(bd)$	Yes[4]
Bidirectional (when applicable)	Yes[1,5]	$O(b^{d/2})$	$O(b^{d/2})$	Yes[4,5]

- b is the branching factor
- d is the depth of the shallowest solution
- m is the maximum depth of the search tree
- ℓ is the depth limit

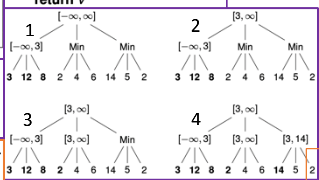
[1] If b is finite
[2] If step costs $\geq \epsilon$ for positive ϵ.
[3] Complete for graph-search version.
[4] If step costs are all identical.
[5] If both directions use breadth-first search.

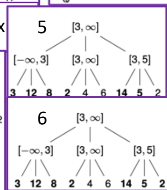
IDEA Maximize the worst-case outcome of MAX.

```
function ALPHA-BETA-SEARCH (state) returns action
  v ← MAX-VALUE(state, −∞, ∞)
  return action in ACTIONS(state) with value v

function MAX-VALUE (state, α, β) returns utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE (state, α, β) returns utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Minimax

- **Completeness**: Yes
- **Time complexity**: $O(b^m)$ exponential
  - it performs a DFS of the game tree.
  - unfortunately, most games have high branching factor.
- **Space complexity**: $O(bm)$ (the same as DFS)
  - $O(m)$ if we generate one successor at a time.
  - At each stage only one successor is generated, whose value is compared with the current value.





TimeC: GreedyBFS O(b^m); A* O(b^d)
SpaceC: GreedyBFS O(b^m); A* O(b^d)
Complete (tree/graph): GreedyBFS N/Y; A* Y/Y (w/ consistent h)
Optimal (tree/graph): GreedyBFS N/N; A* Y/Y (**IFF admissible h**) + optimally efficient

Belief states represent the agent's belief about the possible states it may be in.

A **solution** is a belief state containing only goal states.

PO. belief state b of physical state s <=> the set of s' s.t. PERCEPT(s) = PERCEPT(s'), s~s' <=> equivalence class [s]. b satisfies GOAL-TEST if all s in b satisfy GOAL-TEST. RESULT(b,a) = set of belief states bo

$\{b_o \mid b_o = \text{UPDATE}(\text{PREDICT}(b, a), o)$ and $o \in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))\}$

Solve sensorless problem: any (un)informed SA

To solve PO problem

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan or failure
  OR-SEARCH(problem.INITIAL-STATE, problem,[])

function OR-SEARCH(state, problem, path) returns a conditional plan or failure
  if problem.GOAL-TEST(state) return the empty plan
  if state is on path return failure
  for all action in problem.ACTIONS(state) do
    plan ← AND-SEARCH(RESULT(state, action), problem, [state | path])
    if plan ≠ failure return [action | plan]
  end for
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan or failure
  for all s_i in states do
    plan_i ← OR-SEARCH(s_i, problem, path)
    if plan_i = failure return failure
  end for
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Recursive DFS type

Cond. plan reaches a goal state in all circum.

Planning problems in PDDL => sol is a plan

State: conjunction of fluents (ground, functionless atoms). **Closed-world assumption**: fluents not mentioned false. **Unique names assumption**. **Action schema**: action name, list of var., pre- and post-conditions (conj. of literals); post-condition EFFECT only contains things that changed. **Planning domain**: set of action schemas. **Planning problem**: planning domain + initial state (conj. of ground atoms) + goal (conj. of +/- literals that may contain univ.quant. variables). **Plan**: sequence of ground actions a0 to al. **Ground action**: name(a)(x1/t1, …, xk/tk), where a = action schema. **Induced state sequence (ISS)**: s0, s1, …, sl+1. **Executable** (ISS exists) and **valid** (executable and sl+1 entails goal(P)) plans.

The **result** of executing action a in state s is defined as state s' obtained from s by

- removing the **delete list** DEL(a) of fluents that appear as negative literals in the action's post-condition;
- adding the **add list** ADD(a) of fluents that are positive literals in the post-condition.

$$\text{RESULT}(s, a) \quad = \quad (s \setminus \text{DEL}(a)) \cup \text{ADD}(a) \qquad (1)$$

Solve planning problem: any un(informed) SA. Search **forward** or **backward (relevant-states search)**.

Backward search by regressing actions: ground goal g, ground action a, then regression from g over a gives state g'

$$(2)\ g' \quad = \quad (g \setminus \text{ADD}(a)) \cup \text{PRE}(a)$$

DEL(a) does not appear in (2): we don't know whether the fluents in DEL(a) were true before ⇒ g' is the **weakest precondition** for RESULT(g', a) = g

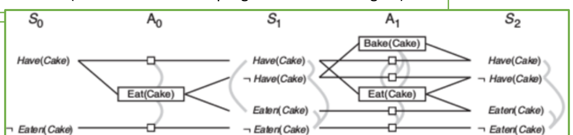
Relevant action = action that could be last step in plan leading up to current goal state

Assume a goal g containing a goal literal $g_i(\vec{t})$, and an action schema $A(\vec{z})$.

If $A(\vec{z})$ has an effect literal $e_j(\vec{y})$ such that $g_i(\vec{t}) = e_j[\vec{y}/\theta]$, then let $a = A[\vec{z}/\theta]$.

If there is no effect in a that is the negation of a literal in g, then a is **relevant** towards g.

Automated heuristics finding from relaxed problems. Relax problem by adding more transitions (more paths). How? By relaxing actions: drop all or certain preconditions from all actions + drop all effects that are not literals in goal state + ignore delete lists (to make monotonic progress towards the goal)



Planning graphs: data structure for giving better heuristic estimates + searching for sol (GRAPHPLAN). Polynomial-size approx. (constructed quickly). Gives estimate of num steps to reach goal (admissible h). Estimate always correct when unreachable goal reported. Directed graph organised into levels. Level i is an admissible heuristic/estimate of how difficult it is to achieve a literal /perform an action, from initial state S0. **Persistence actions** for all fluents, **mutex links** between fluents or actions, **termination** iff 2 same cons. S levels. **Time/SpaceC**: if a actions, l literals/fluents, n levels, then O(n(a+l^2)).

```
function GRAPHPLAN(problem) returns solution or failure
  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← an empty hash table
  for t = 0 to ∞ do
    if goals all non-mutex in S_t of graph then
      solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
      if solution ≠ failure then
        return solution
      end if
    end if
    if graph and nogoods have both leveled off then
      return failure
    end if
    graph ← EXPAND-GRAPH(graph, problem)
  end for
```

To extract a plan directly

Extract by Backward search, w/ intermediate goal states at each level before last one

Cost of a goal literal gi: level at which it **FIRST** appears in planning graph (**level cost**) => admissible estimate for each goal literal. **Cost of a conj. of goals**:

max-level heuristic: maximum level cost of any of the goals (admissible).

level sum heuristic: sum of the level costs of goals (might be not admissible, but works well for decomposable problems).

set-level heuristic: level at which all the literals in the goal appear without any pair of them being mutually exclusive.

A **mutex link** holds between *two literals* at a given level if any of the following two conditions holds:

- One is the negation of the other.
- *Inconsistent support*: each possible pair of actions that could achieve the two literals is mutually exclusive (e.g., *Eaten*(*Cake*) and *Have*(*Cake*) are mutex in S_1, but not in S_2).

A **mutex link** holds between *two actions* at a given level if any of the following three holds:

- *Inconsistent effects*: one action negates an effect of the other.
- *Competing needs*: one of the preconditions of one action is mutex with a precondition of the other.
- *Interference*: one of the effects of one action is the negation of a precondition of the other.

Admissible heuristic: never overestimates the estimated cost from state s to goal (obtain from relaxed problems).