Algorithm 1 Breadth First Search

```
F(Frontier)← Queue(u)
E(Explored)← {u}
while F is not empty do
    u ← F.pop()
    for all children v of u do
        if GoalTest(v) then
            return path(v)
        if v not in E then
            E.add(v)
            F.push(v)
return Failure
```

Algorithm 5 Depth First Search(DFS)

```
F(Frontier)← Stack(u)
while F is not empty do
    u ← F.pop()
    if GoalTest(u) then
        return path(u)
    if HasUnvisitedChildren(u) ther
        for all children v of u do
            F.push(v)
return Failure
```

Algorithm 4 Uniform Cost Search(UCS): Findl

```
F(Frontier)← PriorityQueue(u)
E(Explored)← {u}
ĝ[u] ← 0
while F is not empty do
    u ← F.pop()
    if GoalTest(u) then
        return path(u)
    E.add(u)
    for all children v of u do
        if v not in E then
            if v in F then
                ĝ[v] = min(ĝ[v], ĝ[u] + c(u, v))
            else
                F.push(v)
                ĝ[v] = ĝ[u] + c(u, v)
return Failure
```

Algorithm 6 Iterative Deepening Search

```
cutoff ← initval
while true do
    ret ←FindPathToGoal(u, cutoff)
    if ret == Failure then
        IncreaseCutoff(cutoff)
    else
        return ret
```

Algorithm 5 Depth Limited Search

```
F(Frontier)← Stack(u)
while F is not empty do
    u ← F.pop()
    if GoalTest(u) then
        return path(u)
    if depth(u) ≤ cutoff then
        for all children v of u do
            F.push(v)
return Failure
```

Algorithm A* Algorithm: FindPathToGoal

```
F(Frontier)← PriorityQueue(u)
E(Explored)← {u}
Initialize ĝ   ĝ[u] ← 0;  ĝ[v] = ∞. ∀v ≠ u
while F is not empty do
    u ← F.pop()
    if GoalTest(u) then
        return path(u)
    E.add(u)
    for all successors v of u do
        if v not in E then
            F.push(v)
            ĝ[v] = ĝ[u] + c(u, v)
            f̂[v] = h[v] + ĝ[v]
return Failure
```

Property	BFS	UCS	DFS	IDS
Complete	Yes[1]	Yes[2]	No	Yes
Optimal	No[3]	Yes	No	No
Time	$\mathcal{O}(b^{d+1})$	$\mathcal{O}\left(b^{1+\left\lfloor \frac{C^*}{\varepsilon} \right\rfloor}\right)$	$\mathcal{O}(b^{m+1})$	$\mathcal{O}(b^{d+1})$
Space	$\mathcal{O}(b^{d+1})$	$\mathcal{O}\left(b^{1+\left\lfloor \frac{C^*}{\varepsilon} \right\rfloor}\right)$	$\mathcal{O}(bm)$	$\mathcal{O}(bd)$

1. if b is finite.
2. If b is finite and step cost $\geq \varepsilon$

Problem defined by: **states** S and **initial state**, **actions** A, **state transition model** $T(s_1 \in S, a \in A) \mapsto s_2 \in S$, **goal test**, **cost function** $C(a)\colon A \mapsto \mathbb{R}$.

Consistency: $\forall v_i, v_j, h(v_i) \leq h(v_j) + c(v_i, v_j)$ where where v_j is a successor of v_i and $c(v_i, v_j)$ is the cost of moving from v_i to v_j, i.e. for each step we move backwards, the value of the heuristic at the further node $\leq$ the value at the closer node plus the cost of the step.

Admissibility: $\forall v_i, h(v_i) \leq C^*(v_i) = g(w) - g(v_i)$ where $C^*(v)$ is the true optimal cost to reach the goal from v, i.e. the value of the heuristic is always less than or equal to the true cost of reaching the goal.

A^* is optimal for consistent h, or admissible h with tree search. Consistency and $h(w) = 0 \implies$ admissibility. Admissible heuristics can be created by relaxing problem constraints.

For cycle checking, we keep track of all previously seen nodes (expanded or in frontier) *and* their cost. When adding a new node to the frontier, if it matches a previously seen node and has higher (or equal) cost, don't add it. If it matches but has lower cost, add it and prune the higher cost node from the frontier. Note for A^* the cost is taken as $g(n)$ and not $f(n)$.

```
BacktrackingSearch(prob, assign)
if AllVarsAssigned(prob, assign) then
    if isConsistent(assign) then
        return assign
    else
        return failure
var ← PickUnassignedVar(prob, assign)
for value ∈ OrderDomainValue(var, prob, assign) do
    assign ← assign ∪ (var = value)
    result ← BacktrackingSearch(prob, assign)
    if result! = failure then return result
    assign ← assign \ (var = value)
return failure
```

Algorithm 3 BacktrackingSearch_with_Inference(*prob*, *assign*)

```
if AllVariablesAssigned(prob, assign) then return assign
var ← PickUnassignedVar(prob, assign)
for value in OrderDomainValue(var, prob, assign) do
    if ValIsConsistentWithAssignment(value, assign) then
        assign ← assign ∪ {(var = value)}
        inference ← Infer(prob, var, assign)
        assign ← assign ∪ inference
        if inference!=failure then
            result ← BacktrackingSearch.(prob, assign)
            if result!=failure then return result
            assign ← assign \ inference
        assign ← assign \ {(var = value)}
return failure
```

```
Infer(prob, var, assign)
inference ← ∅
varQueue ← [var]
while varQueue is not empty do
    y ← varQueue.pop()
    for each constraint C in prob where y ∈ Vars(C) do
        for all x ∈ Vars(C) \ y do
            S ← ComputeDomain(x, assign, inference)
            for each value v in S do
                if no valid value exists for all var ∈ Var(C) \ x s.t. C[x ⊢ v] is satisfied then
                    inference ← inference ∪ (x ∉ {v})
            T ← ComputeDomain(x, assign, inference)
            if T = ∅ then return failure
            if S ≠ T then
                varQueue.add(x)
return inference
```

Algorithm 2 BacktrackingSearch(*prob*, *assign*)

```
if AllVarAssigned(prob, assign) then return assign
var ← PickUnassignedVar(prob, assign)
for value in OrderDomainValue(var, prob, assign) do
    if ValIsConsistentWithAssignment(value, assign) then
        assign ← assign ∪ {(var = value)}
        result ← BacktrackingSearch.(prob, assign)
        if result!=failure then return result
        assign ← assign \ {(var = value)}
return failure
```

Problem defined by: **variables** $X = \{ x_1, x_2, \ldots, x_n \}$, **domains** $D = \{ D_{x_1}, D_{x_2}, \ldots, D_{x_3} \}$, **constraints** C relating to domains. NP-Hard.

Default inference is expensive. Methods to limit depth: **forward checking:** only check the effect of assigning the current variable (don't add x to queue); or add to queue only if the variable has fewer than N valid values left, $|T| \leq N$.

For Infer, only operate on the variables that haven't been assigned. Heuristics: **Minimum Remaining Value:** PickUnassignedVar choose the unassigned variable with the smallest effective domain size (allows quick backtracking); **Least Constraining Value:** OrderDomainValue choose the value that rules out the fewest domain values for other variables (so we maximize the chances of success).

Variants: boolean satisfiability (B-SAT), where $D_{x_i} = \{ 0, 1 \}$; binary CSP, where every constraint $C(x_i, x_j)$ is over 2 variables. 2-SAT: combination of the two, is the only one with polynomial time solutions.

Type	Characteristics	Analogy
Simple Reflex	Action depends only on percept	**Infant** • If Hungry then Cry • If Happy then Sleep
Model-based Reflex	Action depends on internal state (based on percept history), model of the world, and percept	**Kid** If Want candy & Didn't Receive candy then Ask for candy
Goal-based	Action depends on current state, percepts, model of the world Action plan to achieve desired goal	**Teenager** Goal: Want to get into UofT Plan: Study and Sports Activites based on the Goal.
Utility-based	Useful for multiple (possible) conflicting goals A weighted combination of goals	**Adult** α*Job+ β*Partner+ γ*Health

Fully observable (vs. partially observable):
- sensors provide access to the complete state of the environment at each point in time.

Deterministic (vs. stochastic)
- The next state of the environment is completely determined by the current state and the action executed by the agent.

Static (vs. dynamic)
- The environment is unchanged while an agent is deliberating.

Discrete (vs. continuous)
- A finite number of distinct states, percepts, and actions.

Single agent (vs. multi-agent)
- An agent operating by itself in an environment.

$$\text{MiniMax}(s) = \begin{cases} \text{Utility}(s) & s \text{ is a terminal state} \\ \max_{a \in A} \text{MiniMax}(\text{Result}(s,a)) & \text{MAX player} \\ \min_{a \in A} \text{MiniMax}(\text{Result}(s,a)) & \text{MIN player} \end{cases}$$

DFS: $\mathcal{O}(b^d)$ time, $\mathcal{O}(bd)$ space (best and worst).
Searches entire tree; needs finite search space.
Estimate utils for non-terminal states if too deep.

Algorithm 6 α-β-MaxValue(s,α, β)

```
if IsTerminal(s)=True then return UTILITY(s)
v ← −∞
for all action a in Actions(s) do
    v ←max(v, α-β-MinValue(Result(s, a),α,β))
    if v ≥ β then return v
    α ←max(α, v)
return v
```

Algorithm 5 α-β-MinValue(s,α, β)

```
if IsTerminal(s)=True then return UTILITY(S)
v ← ∞
for all action a in Actions(s) do
    v ←min(v, α-β-MaxValue(Result(s, a),α, β))
    if v ≤ α then return v
    β ←min(v, β)
return v
```

Algorithm 4 α-β-MinimaxDecision(S)

```
bestAction ← null
max ← −∞
for all action a in Actions(s) do
    util ← α-β-MinValue(Result(s, a),max, ∞)
    if util > max then
        bestAction ← a
        max ← util
return bestAction
```

α: Best value for MAX along path to root (init $-\infty$). β: Best value for MIN along path to root (init ∞).
If $\alpha \geq \beta$, prune remaining children. $\mathcal{O}(b^{\frac{d}{2}})$ time (best), $\mathcal{O}(b^{\frac{3}{4}d})$ (average), $\mathcal{O}(b^d)$ (worst).

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) \qquad P(A|B) = \sum_{C_i} P(A|B, C_i)P(C_i|B) \qquad P(X_1, \ldots, X_N) = \prod_{i=1}^{N} P(X_i | \text{ parents}(X_i))$$

Independence: $P(A|B) = P(A) \iff P(B|A) = P(B) \iff P(A, B) = P(A)P(B)$
Conditional Independence: $P(A|B, C) = P(A|C) \iff P(B|A, C) = P(B|C) \iff P(A, B|C) = P(A|C)P(B|C)$

Bayesian Network Law: Given all parents, the probability of any node is independent of its non-descendants.

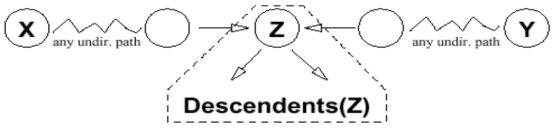
(1)

If Z in evidence, the path between X and Y blocked

(2)

If Z in evidence, the path between X and Y blocked

(3)



If Z is ***not*** in evidence and***no*** descendent of Z is in evidence, then the path between X and Y is blocked

$\mathcal{E}$ d-separates X and Y if it blocks every undirected path between X and Y
$\mathcal{P}$ is any undirected path from X to Y in the network; $\mathcal{E}$ blocks $\mathcal{P}$ iff $\exists Z \in \mathcal{P}$ s.t. one of:

1. $Z \in \mathcal{E}$ and one arc on $\mathcal{P}$ enters Z and another leaves Z
2. $Z \in \mathcal{E}$ and both arcs on $\mathcal{P}$ leave Z
3. Both arcs on $\mathcal{P}$ enter Z, and $Z, \text{descendants}(Z) \notin \mathcal{P}$

Restrict: $f_{E=e}$ takes only rows that have value e for E; E is no longer a variable in the factor.
Multiply: $f_1 * f_2$ combines tables, multiplying rows with the same values for shared variables.
Sum out: $\sum_{Z_j} f$ sums rows with the same values for all other variables; Z_j is no longer a variable.

1. Replace each factor f∈F that mentions a variable(s) in **E** with its restriction $f_{\mathbf{E}=\mathbf{e}}$ (this might yield a factor over no variables, a constant)

2. For each Z_j—in the order given—eliminate $Z_j \in \mathbf{Z}$ as follows:
   (a) Compute new factor $g_j = \sum_{Z_j} f_1 * f_2 * \ldots * f_k$,
   where the f_i are the factors in F that include Z_j
   (b) Remove the factors f_i that mention Z_j from F and add new factor g_j to F

3. The remaining factors refer only to the query variable Q.
   Take their product and normalize to produce Pr(Q|**e**)

Complexity: Exponential in time and space to **elimination width** k.
Polytree: single path between any pair of nodes; there exists a VE order that never increases factor size.
Min-fill heuristic: always eliminate the variable that creates the smallest factor.
Guarantees linear time for polytrees in the network size (not # of vars).

Commutative law	$p \wedge q \equiv q \wedge p$	$p \vee q \equiv q \vee p$
Associative law	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
Distributive law	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
Identity law	$p \wedge \textbf{true} \equiv p$	$p \vee \textbf{false} \equiv p$
Universal bound law	$p \vee \textbf{true} \equiv \textbf{true}$	$p \wedge \textbf{false} \equiv \textbf{false}$
Idempotent law	$p \wedge p \equiv p$	$p \vee p \equiv p$
Negation law	$p \vee \neg p \equiv \textbf{true}$	$p \wedge \neg p \equiv \textbf{false}$
Double negation law	$\neg(\neg p) \equiv p$	
de Morgan's law	$\neg(p \wedge q) \equiv \neg p \vee \neg q$	$\neg(p \vee q) \equiv \neg p \wedge \neg q$
Absorption law	$p \vee (p \wedge q) \equiv p$	$p \wedge (p \vee q) \equiv p$
Implication law	$p \rightarrow q \equiv \neg p \vee q$	

Propositional Logic

Operators: $\neg A$ (negation), $A \wedge B$ (conjunction/and), $A \vee B$ (disjunction/or), $A \to B$ (implication; $\neg A \vee B$), $A \leftrightarrow B$ (bi-implication).

τ **satisfies** A iff $\bar{\tau}(A)$ is true. τ satisfies Φ iff τ satisfies all formulas in Φ.

A is a **logical consequence** of Φ ($\Phi \models A$) iff for all τ, if τ satisfies Φ, then it satisfies A.

Limited by boolean variables (cannot cross reference between individuals in a statement) and the lack of quantifiers (having to list all members to specify a property).

First-Order Logic

Defined by: Variables V, functions F, predicates P. **Terms**: variables or functions of terms. 0-ary functions are constant terms (cannot be quantified). **Vocabulary**: $\mathcal{L}$, a set of function and predicate symbols. **Formula**: *atomic* ($P(t_1, \ldots, t_n)$ where t_i are terms), or formulas combined with propositional operators, or $\exists$ and $\forall$ quantifiers.

Converting from English: things become constants, types/properties become unary predicates, relationships become binary (or more) predicates, associations become functions.

Structure: an $\mathcal{L}$-structure $\mathcal{M}$ contains the **universe** $M \neq \varnothing$, function extensions $f^{\mathcal{M}} \colon M^n \mapsto M$ for each $f \in \mathcal{L}$ (specified as individual mappings), predicate extensions $P^{\mathcal{M}} \subseteq M^n$ for each $P \in \mathcal{L}$ (specified as sets of n-tuples $\langle A, B, \ldots \rangle$ for which the predicate is true). **Object assignment**: an object assignment σ for $\mathcal{M}$ is a mapping from a set of variables to the universe of M. Note $\sigma(m/x)$ is an assignment mapping x to $m \in M$ (for quantifiers).

Satisfaction: $\mathcal{M}$ is a model of C under σ (denoted $\mathcal{M} \models C[\sigma]$) if C is true, under the definitions and variable mappings of $\mathcal{M}$ and σ.

x is **bounded** in A if it only exists in A under a quantifier; otherwise it is **free**. If σ and σ' have the same assignment for all free variables of A, then $\mathcal{M} \models A[\sigma] \iff \mathcal{M} \models A[\sigma']$. A is a **sentence** if it is **closed** (no free variables). For sentences, σ is irrelevant and can be dropped. $\mathcal{M}$ is a **model** of Φ ($\mathcal{M} \models \Phi$) if it satisfies all sentences in Φ. Φ is **satisfiable** if there exists $\mathcal{M}$ that models Φ.

A is a **logical consequence** of Φ ($\Phi \models A$) iff for every $\mathcal{M}$, $\mathcal{M} \models \Phi \implies \mathcal{M} \models A$. If $\Phi \models A$, then $\nexists \mathcal{M}$ s.t. $\mathcal{M} \models \Phi \cup \{\neg A\}$.

Knowledge base: a collection of sentences representing the agent's beliefs, can be used for inference about implicit knowledge through proof procedures. Procedures are **sound** if it only produces logical consequences of the KB, and **complete** if it can produce all logical consequences of the KB.

Resolution by refutation: a sound and complete proof procedure; first assume $\neg A$ (refutation), then convert $\neg A$ and KB to a clausal theory C, and resolve clauses in C until the empty clause is reached, where we conclude A is true.

Clausal theory: a set (conjunction) of **clauses** that must all be true; each clause is a disjunction of **literals** (at least one is true); each literal is either an atomic formula or a negated atomic formula.

Resolution: $\dfrac{a_1 \vee \cdots \vee a_n \vee c \qquad b_1 \vee \cdots \vee b_m \vee \neg c}{a_1 \vee \cdots \vee a_n \vee b_1 \vee \cdots \vee b_m}$

Conversion to clausal form:

1. Convert implications: $A \to B$ to $\neg A \vee B$.
2. Move negations inward and simplify double negations: $\neg(A \wedge B) \iff \neg A \vee \neg B$, $\neg(A \vee B) \iff \neg A \wedge \neg B$, $\neg \forall x A \iff \exists x \neg A$, $\neg \exists x A \iff \forall x \neg A$.
3. Variable standardization: rename so that each quantified variable is unique.
4. Skolemization: remove $\exists$ by replacing the variable quantified with a new unique constant $\exists x P(x) \iff P(\boldsymbol{a})$, or unique function which mentions every variable that scopes the existential $\forall x \exists y P(x, y) \iff \forall x P(x, g(x))$.
5. Prenex form: take all quantifiers (only $\forall$ at this point) to the front: $\forall x P \wedge Q \iff \forall x (P \wedge Q)$, $\forall x P \vee Q \iff \forall x (P \vee Q)$.
6. Distribute $\vee$ over $\wedge$: $A \vee (B \wedge C) \iff (A \vee B) \wedge (A \vee C)$.
7. Convert to clauses: remove all $\forall$ quantifiers (implicit) and break apart $\wedge$.

Resolution is **refutation complete**: it can eventually prove that a clausal theory is unsatisfiable, but may not terminate when it is satisfiable. In general first-order unsatisfiability is semi-decidable (there exists an algorithm that correctly gives positive answers), but not decidable.