Lists `mylist = [1, 2, 3]` → create a list `mylist[0]` → first element `mylist[1:3]` → slice elements 1–2 `mylist[::-1]` → reverse list `copy = mylist[:]` → copy list
List Comprehensions `[x**2 for x in range(5)]` → squares 0–4 `[1, 3, 5, ..., 99]` = —[x for x in range(1, 100, 2)]— = —[2 * x + 1 for x in range(0, 50)]— `[x for x in lst if x % 2 == 0]` → keep evens
Dictionaries `d = {'a':1, 'b':2}` → create dictionary `d['a']` → access value `d.keys()` → all keys `d.values()` → all values `{x:x**2 for x in range(5)}` → dict comprehension
Sets `s = {1,2,3}` → create set `s1  s2`— → union `s1 & s2` → intersection
Tuples `t = (1,2,3)` → immutable sequence
Loops `for i in range(5): ...` → repeat 5 times `while condition: ...` → loop until false
Enumerate + Zip `enumerate(lst)` → index + value `zip(list1, list2)` → pair items
Conditionals `if ... elif ... else ...` → branching
Functions `def f(x,y=2): return x+y` → define function `lambda x: x**2` → quick anonymous function
Generators `yield` → produce values lazily
List Operations `append` → add item `extend` → add multiple `insert` → add at position `pop` → remove last `remove` → delete by value `sum, min, max, sorted` → aggregate utilities
Slicing Patterns `seq[::2]` → every other `seq[:-1]` → all but last `seq[::-1]` → reversed
Before an agent can start searching, a **well-defined problem** must be formulated. • Rational agents choose actions to maximize expected utility. Agent Function: simple reflex: percept only, model-based: percept + state, goal-based: model + goal, utility-based: model + goal + utility. • PEAS: Performance measure, Environment, Actuators, Sensors. • Search algorithms are judged on **completeness**, **optimality**, **time**, and **space** complexity.
Uninformed search uses only the problem definition: **Completeness** is considered complete when it guarantees to find a solution if one exists. **Optimality** means it finds the least-cost solution if one exists. **Admissible** means the heuristic never overestimates the true cost to reach the goal from node n, i.e., $h(n) \leq h^*(n)$. **Triange Inequality** $h(n)$ estimated cost from h -¿ g. $h(n')$ estimated cost from n' to g. $c(n, n')$ actual cost from n to n'. $h(n) \leq c(n, n') + h(n')$. **Consistent** estimated cost – **Best-first search**: expands nodes using an evaluation function. – **Breadth-first search**: expands shallowest nodes first; **complete** if finite branching factor, **optimal** if unit costs, but **exponential space**. – **Uniform-cost search**: expands lowest path cost $g(n)$; **optimal** for general costs. – **Depth-first search**: expands deepest nodes; **not complete**, **not optimal**, but **linear space**. – **Depth-limited search**: DFS with depth bound. – **Iterative deepening search**: runs DFS with increasing depth; **complete**, **optimal** if unit costs, **linear space**. – **Bidirectional search**: expands from initial and goal until they meet; **very efficient** in some problems.
Informed search uses a heuristic $h(n)$: – **Greedy best-first search**: expands lowest $h(n)$; **Not complete not optimal**, often efficient. – **A* search**: expands lowest $f(n) = g(n) + h(n)$; **complete and optimal** if h is **admissible**; space-heavy. – **Bidirectional A***: sometimes more efficient than A*. – **IDA***: iterative deepening A*; addresses **space issues**, still **optimal with admissible** h. – **RBFS** (recursive best-first search) and **SMA*** (simplified memory-bounded A*): **optimal**, memory-bounded. – **Beam search**: keeps only k best nodes; **incomplete**, **suboptimal**, but fast. – **Weighted A***: expands fewer nodes using $f(n) = g(n) + w \cdot h(n)$; **faster**, but **not optimal**.
Depth-First Search (DFS) • Uses a LIFO stack (explicit or recursion). • Not complete (can get stuck in infinite path). • Not optimal (returns first found solution). • Time: $O(b^m)$ • Space: O(m)

```
def DFS(problem):
    stack = [(problem.start, [])]
    visited = set()
    while stack:
        state, path = stack.pop()
        if goal(state): return path
        if state not in visited:
            visited.add(state)
            for a,s2 in successors(state):
                stack.append((s2, path+[a]))
```

Breadth-First Search (BFS) • Complete if branching factor finite. • Optimal if all step costs = 1. • Time: $O(b^d)$ • Space: $O(b^d)$

```
def BFS(problem):
    queue = deque([(problem.start, [])])
    visited = set()
    while queue:
        state, path = queue.popleft()
        if goal(state): return path
        if state not in visited:
            visited.add(state)
            for a,s2 in successors(state):
                queue.append((s2, path+[a]))
```

Uniform Cost Search (UCS) • Expands node with lowest total path cost $f(n) = g(n)$. • Uses priority queue (min-heap). • Equivalent to Dijkstra's algorithm. • Complete if step costs $\geq \epsilon$. • Optimal (finds least-cost solution). • Time/Space: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

```
def UCS(problem):
    frontier = [(0, problem.start, [])]
    visited = {}  # state: best_cost
    while frontier:
        cost, state, path = heappop(frontier)
        if goal(state): return path
        if state not in visited or cost < visited[state]:
            visited[state] = cost
            for a,(s2,c) in successors(state):
                heappush(frontier, (cost+c, s2, path+[a]))
```

Greedy Best-First Search • Expands node with lowest heuristic $f(n) = h(n)$. • Ignores cost so far, can get stuck in loops. • Complete: only in finite state spaces. • Optimal: No (may find suboptimal paths). • Time/Space: $O(b^m)$.

```
def Greedy(problem,h):
    frontier = [(h(problem.start), problem.start, [])]
    visited = set()
    while frontier:
        _, state, path = heappop(frontier)
        if goal(state): return path
        if state not in visited:
            visited.add(state)
            for a,s2 in successors(state):
                heappush(frontier,(h(s2),s2,path+[a]))
```

A* Search • Evaluation function: $f(n) = g(n) + h(n)$. • $g(n)$ = path cost so far, $h(n)$ = heuristic. • Complete if step costs $\geq \epsilon$. • Admissible h guarantees optimality. • Expands nodes in order of lowest $f(n)$. • Optimal if h is admissible. • Time/Space: Exponential in worst case.

```
def find_solution_a_star(self):
        tiebreaker = 0
        path = []
        frontier = PriorityQueue(), state = self.copy()
        frontier.put((f_n, tiebreaker, g_n, path, state))
        visited = set()
        while frontier.qsize() > 0:
            f, tie, g, path, state = frontier.get()
            if state in visited:
                continue
            visited.add(state)
            if state.is_solved():
                return path
            for move, succ_state in state.successors():
                h_n = succ_state.manhattan()
                g_n = g + 1
                f_n = g_n + h_n
                tiebr = tie + 1
                if succ_state in visited:
                    continue
                frontier.put((f_n, tiebr, g_n, path + [move],
    succ_state))
```

Admissibility and Consistency • Admissible: never overestimates, therefore the highest value would be closest to the true estimate. $h(n) \leq h^*(n)$ (never overestimates true cost). • Consistent: $h(n) \leq c(n, a, n') + h(n')$.

Iterative Deepening A* (IDA*) • Uses $f(n) = g(n)+h(n)$ threshold, increases iteratively. • Memory-bounded (like DFS). • Complete and optimal with admissible h.

Weighted A* Search • $f(n) = g(n) + W \cdot h(n)$, with $W > 1$. • Faster but not guaranteed optimal. Useful for satisficing.

```
def __eq__(self, other):
    return self.board == other.board

def as_immutable(self):
    return tuple(tuple(row) for row in self.board)

def __hash__(self):
    return hash(self.as_immutable())
self.board = [[False for _ in range(self.cols)]
                      for _ in range(self.rows)]
```

Heuristics – Greedy = expand lowest h (fast but not optimal). – A* = expand lowest $g + h$ (optimal if admissible h). – Manhattan distance = Tile Puzzle. – Euclidean distance = Grid Navigation. – IDA* = saves memory, optimal with admissible h. – Weighted A* = faster, near-optimal.

game is defined by: **initial state**, **legal actions**, **result function**, **terminal test**, and a **utility function**. • In **two-player, deterministic, zero-sum, perfect-information games**, **minimax** selects **optimal moves** by exhaustive depth-first search. • **Alpha–beta pruning** computes the **same optimal move** as minimax but **prunes irrelevant subtrees** for efficiency, ALPHA is min (-inf), BETA is (inf). • Full game trees are often infeasible; we use **cutoff depths** and an **evaluation function** to approximate utility at non-terminal nodes. **Complete** search means finding a solution in the game tree.

How to evaluate alpha–beta minimax (step-by-step) 1. **Initialize**: Start at root with $\alpha = -\infty$, $\beta = +\infty$. 2. **Max player's turn**: For each child: – Compute its minimax value (recursively). – Update $\alpha = \max(\alpha, value)$. – If $\alpha \geq \beta$: **prune** (stop exploring siblings). 3. **Min player's turn**: For each child: – Compute its minimax value (recursively). – Update $\beta = \min(\beta, value)$. – If $\beta \leq \alpha$: **prune** (stop exploring siblings). 4. **Terminal/cutoff**: If at terminal or depth limit, return **utility or evaluation**. 5. **Backtrack**: Values propagate upward. Root's chosen action = child with best minimax value. **Example**: Alpha= -inf, Beta= +inf - Left MIN: values [3,12,8] → returns 3; Alpha=3 - Middle MIN: [8,2,...] → returns 2; Beta ≤ Alpha → prune - Right MIN: [14,5,2] → drops below Alpha=3 → prune Final choice: Left branch with value 3 **Alpha-Beta Pruning**

Constraint Satisfaction Problems (CSPs) • Variables $X_1...X_n$ with domains $D_1...D_n$ and constraints $C_1...C_m$. • Goal: find a **complete and consistent** assignment satisfying all constraints. • Represented by a **constraint graph**: nodes = variables, edges = binary constraints. • Constraint types: unary, binary, higher-order.

Examples • Map coloring (adjacent regions differ in color). • Sudoku (rows, columns, boxes contain unique numbers).

CSP as Search • Initial state: {} (empty assignment). • Successor: assign a value consistent with constraints. • Goal test: all variables assigned. • Path cost: constant (depth = n). → Depth-first variant = **Backtracking Search**.

Backtracking Search • Standard DFS that assigns one variable at a time. • Backtrack when no legal values remain.

Heuristics • **MRV (Most Constrained Variable)** – fewest legal values. • **Most Constraining Variable** – affects the most others. • **Least Constraining Value** – rules out fewest options. • **Forward Checking** – prune domains of unassigned vars; fail early if domain empty.

Constraint Propagation / Arc Consistency • Propagates constraint effects to reduce domains. • X_i arc-consistent w.r.t. X_j if $\forall v \in D_i, \exists w \in D_j$ satisfying constraint.

AC-3 Algorithm

```
def AC3(csp):
    queue = all_arcs(csp)
    while queue:
        (Xi,Xj) = queue.pop(0)
        if remove_inconsistent(Xi,Xj):
            for Xk in neighbors(Xi)-{Xj}:
                queue.append((Xk,Xi))

def remove_inconsistent(Xi,Xj):
    removed = False
    for x in domain[Xi]:
        if not any(satisfies(x,y) for y in domain[Xj]):
            domain[Xi].remove(x)
            removed = True
    return removed
```

• Complexity: $O(n^2d^3)$ worst case.

Local Search for CSPs • Operates on complete states; minimizes constraint violations. • **Min-Conflicts heuristic:** choose conflicted variable, assign value violating fewest constraints. • Effective for large CSPs (e.g., N-Queens).

Tree-Structured CSPs • If graph is a tree → solvable in $O(nd^2)$ by topological ordering. • **Cycle Cutset:** remove c vars to make tree → $O(d^{c+2}(n - c))$.

Backjumping / Conflict Sets • Instead of chronological backtracking, jump to the cause of conflict. • Track conflict sets for efficiency.

Beyond Binary Constraints • **Path consistency:** ensures triples (X_i, X_j, X_k) consistent. • **Global constraints:** apply to many vars (e.g., AllDifferent).

Key Takeaways • CSPs identify satisfying assignments, not paths. • Declarative representation + general algorithms = scalable reasoning framework. **Logical Agents** • Logic = representation of knowledge + inference. • **Agents** derive new conclusions via reasoning instead of direct perception. • A **knowledge base (KB)** = set of sentences in a formal language. • An **inference algorithm** derives new sentences: KB $\vdash \alpha$. • If KB $\models \alpha$, α is **entailed** (true in all worlds where KB is true).

Model Theory • A **model** assigns truth values to each propositional symbol. • KB $\models \alpha$ if every model that satisfies KB also satisfies α. • Inference algorithm is **sound** if KB $\vdash \alpha$ KB $\models \alpha$. • It is **complete** if KB $\models \alpha$ KB $\vdash \alpha$.

Propositional Logic • Symbols: P, Q, R, . . . (atomic sentences). • Connectives: $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies), $\Leftrightarrow$ (biconditional). • Example: $(P \wedge Q) \Rightarrow R$. • Sentences built recursively from symbols and connectives.

Truth Tables • Used to determine entailment by enumeration. • **Procedure:** list all models, mark where KB true; if α true in all such models → KB $\models \alpha$. • Exponential in number of variables ($O(2^n)$).

Inference Rules (Sound) • **Forward Chaining:** from facts + rules infer new facts until goal found. • **Backward Chaining:** start from goal, prove premises recursively.

Converting to CNF (Conjunctive Normal Form) Eliminate $\Leftrightarrow$ and $\Rightarrow$. Move $\neg$ inward via De Morgan's laws. Distribute $\vee$ over $\wedge$ (CNF). Split conjunctions into separate clauses. Used for **Resolution** and SAT solvers.

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	commutativity of $\wedge$
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	commutativity of $\vee$
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associativity of $\wedge$
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associativity of $\vee$
$\neg(\neg\alpha) \equiv \alpha$	double-negation elimination
$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	De Morgan
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	dist. of $\wedge$ over $\vee$
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	dist. of $\vee$ over $\wedge$