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 `[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
Strings `'hello'.upper()` → uppercase `'hi there'.split()` → split words `' '.join(list)` → join with spaces `s.strip()` → trim whitespace `s.replace('a','b')` → replace text
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. • A problem consists of **five parts**: the initial state, actions, transition model, goal states, and action cost function. • The environment is represented by a **state space graph**. A **path** from initial to goal is a **solution**. • Search algorithms are judged on **completeness**, **optimality**, **time**, and **space** complexity.

Uninformed search uses only the problem definition: – **Best-first search**: expands nodes using an evaluation function. – **Breadth-first search**: expands shallowest nodes first; **complete**, **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 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 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 $h(n)$. • 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 AStar(problem,h):
    frontier = [(h(problem.start),0,problem.start,[])]
    visited = {}
    while frontier:
        f,g,state,path = heappop(frontier)
        if goal(state): return path
        if state not in visited or g < visited[state]:
            visited[state]=g
            for a,(s2,c) in successors(state):
                g2=g+c; f2=g2+h(s2)
                heappush(frontier,(f2,g2,s2,path+[a]))
```

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.

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.

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**

```
def alphabeta(state, depth, alpha, beta, maximizingPlayer):
    if depth == 0 or terminal(state):
        return evaluate(state)
    if maximizingPlayer:
        value = -inf
        for child in successors(state):
            value = max(value,
                        alphabeta(child, depth-1,
                                  alpha, beta, False))
            alpha = max(alpha, value)
            if alpha >= beta:
                break  # prune
        return value
    else:
        value = +inf
        for child in successors(state):
            value = min(value,
                        alphabeta(child, depth-1,
                                  alpha, beta, True))
            beta = min(beta, value)
            if beta <= alpha:
                break  # prune
        return value
```