```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a 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
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     add the node to the explored set
     expand the chosen node, adding the resulting nodes to the frontier
       only if not in the frontier or explored set
```

```
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 do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each 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
```

```
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 return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
     cutoff_occurred? ← false
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit − 1)
        if result = cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
     if cutoff_occurred? then return cutoff else return failure
```

```
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
```

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a 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)))
  return v

function MIN-VALUE(state) returns a 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)))
  return v
```

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

function MAX-VALUE(state, α, β) returns a 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 a 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
```

Solving Problems by Searching

Define search problem

- State space (should be partial-valid)
- Initial state
- Goal state(s) / goal test
- Actions(s) returns actions available to agent at s
- Transition model Result(s, a) returns next state
- Action cost function Action-Cost(s, a, s')
- Path cost

Formulating problems: modelling the search problem through abstraction

Note: assume goal exists at finite depth

Tree and Graph Search Algo

- Start at initial state, keep searching till reaching a goal state
- Frontier: nodes that we have seen but haven't explored yet. At initialisation, frontier is just the source.
- At each iteration, choose a node from frontier, explore it, and add its neighbours to frontier
- Graph search: a node that's been explored once will not be revisited.
- A state: represents a physical config
- A node: a data structure constituting part of search tree. It includes state, parent node, action, and path cost g(n).
- 2 diff nodes can contain the same world state.

Search problem params

- b: branching factor
- d: depth of shallowest goal node
- m: maximum depth of search tree (may be inf)

Uninformed Search

Breadth-First Search (BFS)

- Expand shallowest unexpanded node
- Frontier is FIFO queue
- Goal test is applied when pushing nodes to frontier rather than during expansion
- # of nodes: $O(b) + O(b^2) + \cdots + O(b^d)$

Uniform-Cost Search (UCS)

- Expand least-path-cost unexpanded node
- Frontier is PQ ordered by path cost
- Equivalent to BFS if all step costs are equal

Depth-First Search (DFS)

- Expand least-deepest unexpanded node
- Frontier is LIFO stack

Depth-Limited Search (DLS)

- Run DFS with depth limit l

Iterative Deepening Search (IDS)

- Perform DLSs with increasing depth limit until goal node is found
- Better if state space is large and depth of solution is unknown
- # nodes: $(d+1)O(b^0) + dO(b^1) + (d-1)O(b^2) + \cdots + 2O(b^{d-1}) + O(b^d)$

Property	BFS	UCS	DFS	DLS	IDS
Complete	Yes*	Yes*	No	No	Yes*
Optimal	No	Yes	No	No	No
Time	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$\mathcal{O}(b^m)$	$\mathcal{O}(b^\ell)$	$\mathcal{O}(b^d)$
Space	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$\mathcal{O}(bm)$	$\mathcal{O}(b\ell)$	$\mathcal{O}(bd)$

1. BFS and IDS are complete if b is finite.
2. UCS is complete if b is finite and step cost $\geq \varepsilon$
3. BFS and IDS are optimal if step costs are identical.

Proof of UCS' optimality

Let $c(n)$ be the cost of the path to node n.

If n_2 is expanded after n_1, then $c(n_1) \leq c(n_2)$

- Case 1: n_2 is on the frontier when n_1 is expanded
- Case 2: n_2 was added to the frontier when n_1 was expanded

When n is expanded, every path with cost $< c(n)$ has already been expanded.

- Let $S_0, n_0, n_1, \dots, n_k$ be a path with cost $< c(n)$. Let n_i be the last node on this path that has been expanded.
- $n_i + 1$ is still on the frontier. And $c(n_i + 1) < c(n)$.
- UCS would have expanded $n_i + 1$, not n. So every node on this path must already be expanded.

The first time UCS expands a state, it has found the minimal cost path to it

- No cheaper path exists, else that path would have been expanded before.
- No cheaper path will be discovered later, as all those paths must be at least as expensive.

BFS	- Goal node is near root - Tree is deep but goals are rare
DFS	- Goal node is very deep or all goal nodes are at the same depth - Better space complexity than BFS
UCS	- If cost is known/non-uniform and optimality is a requirement - Equivalent to BFS if step costs are uniform
LDFS	- If we know at what depth the goal node is
IDS	- Like a fusion of BFS and DFS - Some overhead (1/(b-1))

Informed Search

Best-First Search

- Use evaluation function f(n) as a cost estimate
- Frontier: PQ ordered by non-decreasing cost f

Greedy Best-First Search

- f(n) = h(n)
- h(n): heuristic function, which estimates the cheapest path cost from n to goal
- Greedy best-first search expands the node that appears closest to goal
- Completeness: if b is finite, tree-based variant is incomplete, while graph-based variant is complete
- Not optimal
- Time & space $O(b^m)$

A* Search

- f(n) = g(n) + h(n)
- g(n): cost of reaching n from start node
- Complete if finite # of nodes and $f(n) \leq f(G)$
- Optimality depends on heuristics
- Time $O\left(b^{h^*(s_0)-h(s_0)}\right)$
- Space $O(b^m)$

Admissible Heuristic

- Admissible heuristic: $\forall n, h(n) \leq h^*(n)$, i.e. never overestimates cost to reach goal
- If h(n) is admissible, then A* using Tree-Search is optimal

Proof.

- If A* using admissible heuristic returns suboptimal goal t, then there exists a node n in frontier, on optimal path but not expanded.

- $f(t) = g(t) > g^*(t) = f^*(t) = g(n) + h^*(n) \geq g(n) + h(n) = f(n)$
- Admissible heuristic doesn't guarantee optimality for Graph-search. Graph-search discards new paths to a repeated state. If the heuristic is not consistent, the optimal path might be discarded

Dominance: If $h_2(n) \geq h_1(n)$ for all n, then h2 dominates h1. it follows that h2 incurs lower search cost than h1.

Deriving Admissible Heuristics:

- Relaxed problem: one with fewer restrictions on actions
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original proble

Consistent Heuristic

- Consistent heuristic: for every node n and successor n' of n generated by a, $h(n) \leq d(n,n') + h(n')$
- Equivalently, f(n) is non-decreasing along any path. $f(n') = g(n') + h(n') = g(n) + d(n,n') + h(n') \geq g(n) + h(n) = f(n)$
- Consistency implies admissibility (proof by induction)
- If h(n) is consistent, then A* using Graph-Search is optimal

Proof. When A* selects a node n for expansion, the shortest path to n has been found

- If A* returns a suboptimal path to n, there exists a node m in frontier, on optimal path but not expanded.
- However, A* with consistent heuristic explores nodes in a non-decreasing order of f value, hence m should have been explored before n.

Local Search

- The path to goal is irrelevant; the goal state itself is the solution
- State space: set of complete configs
- Find final configs satisfying constraints
- Local search algo: maintain single current best state and try to improve it
- Advantages: very little/constant memory, and can find reasonable solutions in large state space

Hill-climbing search

- if highest-valued successor if better than current, update current.
- Always terminate with a solution
- Problem: can get stuck in local maximum
- Non-guaranteed fixes: sideway moves, random restarts

Adversarial Search aka Games

Game: Problem Formulation

- Initial state
- States
- Players: Player(s) defines which player has the move in state s
- Actions: Actions(s) returns the set of legal moves in s
- Transition model: Result(s, a)
- Terminal test Terminal(s) == true iff game end
- Utility function Utility(s, p): final numeric value for a game that ends in terminal state s for player p

Winning Strategy

- Let Vmax be the set of nodes controlled by the MAX player and Vmin be the set of nodes controlled by the MIN player.
- A *strategy* for the MAX player is a mapping s1 : Vmax → V ; similarly, a strategy for the MIN player is a mapping s2 : Vmin → V.
- A strategy s1* for player 1 is called winning if for any strategy s2 by player 2, the game ends with player 1 as the winner.
- The leaves of the minimax tree are *payoff nodes*. There is a payoff a(v) ∈ R associated with each payoff node v. More formally, the utility of the MAX player from v is umax(v) = a(v) and the utility of the MIN player is umin(v) = −a(v). The utility of a player from a pair of strategies s1 ∈ S1,s2 ∈ S2 is simply the utility they receive by the leaf node reached when the strategy pair (s1 , s2) is played.

Optimal Strategy at Node - Minimax

- MAX chooses move to maximize the minimum payoff
- MIN chooses move to minimize the maximum payoff

$$Minimax(s) = \begin{cases} Utility(s) & \text{if TerminalTest}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if Player}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if Player}(s) = \text{MIN} \end{cases}$$

- Complete if game tree is finite
- Optimal
- Time $O(b^m)$
- Space $O(bm)$
- Returns a SPNE: best action at every choice node

Proof: by induction

- Assume MINIMAX computes SPNE for all subtrees at height h-1. WLOG, consider node v at height h and assume this is a MAX node.
- Let s_1^* be the strategy outputted by MINIMAX, and s_1 another strategy by player 1
- Suppose that v_1^*, v_1 are the nodes chosen by s_1^* and s_1
- $u_1(v, s_1^*, s_2^*) = u_1(v_1^*, s_1^*, s_2^*) \geq u_1(v_1, s_1^*, s_2^*) \geq u_1(v_1, s_1, s_2^*) = u_1(v, s_1, s_2^*)$

Alpha-beta pruning

- Maintain a lower bound alpha and upper bound beta of the values of MAX's and MIN's nodes seen thus far
- MAX node n: $\alpha(n)$ = highest observed value found on path from n; initially $\alpha(n) = -\infty$
- MIN node n: $\beta(n)$ = lowest observed value found on path from n; initially $\beta(n) = +\infty$
- Given a MIN node n, stop searching below n if there is some MAX ancestor i of n with $\alpha(i) \geq \beta(n)$
- Given a MAX node n, stop searching below n if there is some MIN ancestor i of n with $\beta(i) \leq \alpha(n)$
- Pruning never affects the final outcome i.e. it leaves at least one strategy played in a Nash Equilibrium; however, alpha-beta pruning cannot be used to find SPNE.
- Time: $O\left(b^{\frac{m}{2}}\right)$ for perfect ordering, $O\left(b^{\frac{3m}{4}}\right)$ for random ordering when b < 1000

Evaluation function and cut-off test

- Evaluation function: estimated expected utility of state
- Cut-off test: depth limit
- Heuristic minimax value: run minimax until depth d, then start evaluation function to choose nodes

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \boxed{\text{if CUTOFF-TEST}(s,d)} \\ \max_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$