## Agent Model
Perceives its environment through sensors and acting upon that environment through actuators

- Percepts – Sensors
- Actions – Actuators
- Environment
- Performance Measure

## Rational Agent
Select actions that **maximises its (expected) utility**
Percepts, Env, Action Space → Action Selected
i.e., Specified by an agent function $f: P \rightarrow A$

## Rationality
What is rational at a given time depends on PEAS:
- **P**erformance measure
- Prior **E**nvironment knowledge
- **A**ctions/Actuators
- Percept sequence to date (**S**ensors)

Limitations:
- Percepts may not provide all the required info (Rationality ≠ omniscience)
- Actual outcome of actions may not be as expected (Rationality ≠ clairvoyant)

## Environment Types
- **Fully observable (vs. partially observable):** An agent's sensors give it 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.
  - If the environment is deterministic except for the actions of other agents, then the environment is **strategic**
- **Episodic (vs. sequential):** The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself.
- **Static (vs. dynamic):** The environment is unchanged while an agent is deliberating.
  - (The environment is **semi-dynamic** if the environment itself does not change with the passage of time, but the agent's performance score does)
- **Discrete (vs. continuous):** A limited number of distinct, clearly defined percepts and actions.
- **Single agent (vs. multiagent):** An agent operating by itself in an environment

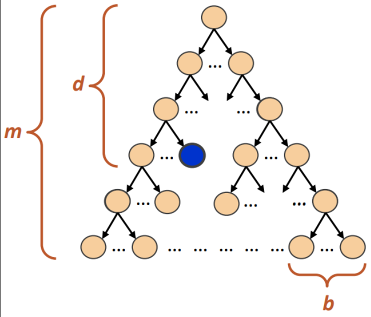
## Search Problem Formulation
- **State space**, e.g. At(Arad), At(Bucharest)
- **Initial state**, e.g. At(Arad)
- **Actions**, set of actions given a specific state
  - **Transition model** e.g., Result(At(Arad),Go(Zerind)) → At(Zerind)
  - **Path cost** (additive), e.g., sum of distances, number of actions, etc
- **Goal test**, can be
  - Explicit, e.g. At(Bucharest)
  - Implicit, e.g. checkmate(x)

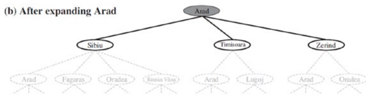
## Search Problem Solution
- A **solution** is a sequence of actions from the initial state to a goal state (E.g., Arad → Sibiu → Fagarus → Bucharest)
- An **optimal solution** is a solution with the lowest path cost

## General Search
- **Root** = Initial State, **Leaves** = Generated State
- **State** is a repr of a physical configuration
- **Node** is a data structure constituting **part of a search tree** (Comprises of state, parent, child, action path-cost, depth)
- **Expand** function creates new nodes
  - Uses Actions and Transition Model to create corresponding states

## Search Strategies
Defined by picking the order of node expansion
Evaluated through:
- **Completeness** - find a solution if one exists?
- **Optimality** - least-cost solution?
- **Time complexity** - number of nodes generated/expanded
- **Space complexity** - maximum number of nodes in memory

Time and space complexity measured in terms of:
- **b** – max branching factor
- **d** – depth of least-cost solution
- **m** – max depth of state space




## General Tree Search



```
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
```

Problem: Repeated states (Redundant paths can cause a tractable problem to become intractable)

## General Graph Search
```
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
```

## Types of Searches
- **Uninformed Search**
  No extra info about states beyond that in the problem definition
- **Informed Search**
  Uses problem-specific knowledge beyond the definition of the problem itself
- **Adversarial Search**
  Used in multi-agent environment where the agent needs to consider the actions of other agents and how they affect its performance

## Breadth-First Search
General idea: Expand shallowest unexpanded node
Implementation: Use First-In First-Out (FIFO) queue
- **Completeness:** Yes (if b is finite)
- **Optimality:** Yes (if cost=1 per step; Not optimal in general)
- **Time complexity:** $1 + b + \cdots + b^d = O(b^d)$
- **Space complexity:** $O(b^d)$, all node in memory

Problems: Memory Requirements & Execution Time

## Uniform Cost Search
General idea: Expand unexpanded node n with the lowest path cost g(n)
Implementation: Using a priority queue ordered by path cost g
- **Completeness:** Yes (if step cost > ε, some positive constant)
- **Optimality:** Yes
- **Time complexity:** $O(b^{C^*/\varepsilon})$, where C* = cost of optimal solution
- **Space complexity:** $O(b^{C^*/\varepsilon})$

Problems: Possible redundant searches

## Depth-First Search
General idea: Expand deepest unexpanded node
Implementation: Use Last-In First-Out (LIFO) queue
- **Completeness:** No (if m is infinite)
- **Optimality:** No
- **Time complexity:** $O(b^m)$, bad if m>d by a lot
- **Space complexity:** $O(bm)$, linear space

## Depth-Limited Search
General idea:
Depth-First Search with predetermined depth limit $l$ (Nodes at depth $l$ have no child nodes & Solves infinite-path problem)
- **Completeness:** No (if $l < d$)
- **Optimality:** No (if $l > d$)
- **Time complexity:** $O(b^l)$
- **Space complexity:** $O(bl)$

## Iterative Deepening Search
General idea: Use increasing Depth-Limited Search (DLS) to find the best depth limit l
- I.e., use DLS with depth limit 1. If no solution, then increase depth limit to 2. So on and so on, until solution is found

Best of both Breadth-First Search and Depth-First Search
- **Completeness:** Yes
- **Optimality:** Yes
- **Time complexity:** $O(b^d)$
- **Space complexity:** $O(bd)$

## BFS vs DFS
Use BFS when:
- Optimal solution is important
- m is much greater than d

Use DFS when:
- Space is important. DFS: $O(bm)$, BFS: $O(b^d)$

## Heuristics
- The heuristic function h(n) is an estimate of how close a state n is to the goal state
- Informed search algorithms use heuristics to solve the search problem

## Greedy Best-First Search
General idea: Expand the node n with the lowest heuristic h(n)
Implementation: Use a priority queue ordered by heuristic h(n)
- **Completeness:** No (Can get stuck in loops, unless we keep track of repeated nodes)
- **Optimality:** No
- **Time complexity:** $O(b^m)$
- **Space complexity:** $O(b^m)$, all nodes in mem

## UCS vs G-BFS
- UCS is complete and optimal but may waste search in the wrong direction
- Greedy search generally in the correct direction but not complete or optimal
- Combine UCS & G-BFS → A* Search

## A* Search
General idea: Expand the node n that has incurred the least cost and is nearest to the goal state
Implementation: Using a priority queue ordered by eval. func. f(n)
- Evaluation function f(n) = g(n) + h(n)
- Path cost g(n) = total path cost from start node to node n
- Heuristic h(n) = estimated distance from node n to goal state
- **Completeness:** Yes (if step cost > ε, some positive constant)
- **Optimality:** Yes (If heuristics are admissible/consistent)
- **Time complexity:** $O(b^{C^*/\varepsilon})$, where C* = cost of optimal solution
- **Space complexity:** $O\left(b^{\frac{C^*}{\varepsilon}}\right)$

Applications:
- Path finding problems
- Video games
- Resource planning problems
- Robot motion planning

## Heuristic Properties
**Admissibility**: A heuristic h(n) is admissible if $h(n) \leq h^*(n)$. For example:
- $h(n)$ = estimated distance from node n to goal state
- $h^*(n)$ = true cost from node n to goal state

**Consistent**: A heuristic h(n) is consistent if $h(n) \leq c(n, a, n') + h(n')$. For example:
- h(n) = estimated distance from node n to goal state G
- h(n') = estimated distance from node n' to goal state G
- c(n,a,n') = cost of getting from node n to n'

**Dominance**: A heuristic $h_2(n)$ dominates $h_1(n)$ if $h_2(n) \geq h_1(n)$, for all n.
- Only if both heuristics are admissible
- A more dominant heuristic will be better for search (Potentially explore less branches)

## Designing Heuristics
Admissible heuristics can be derived from the exact solution cost of a relaxed problem

## Constraint Satisfaction Problems
**State**
- Defined by variables $X_i$ that take on values from domain $D_i$

**Goal Test**
- A set of constraints $C_i$ specifying allowable combinations of values for subsets of variables

In contrast to standard search problems
- State is a "black box" - any old data structure that supports goal test, eval, successor

CSP comprises of:
- Finite set of variables $X = \{X_1, X_2, \ldots, X_n\}$
- Non-empty domain D of k possible values for each variable Di, where $D_i = \{v_1, \ldots, v_k\}$
- Finite set of constraints $C = \{C_1, C_2, \ldots, C_m\}$
- Each constraint $C_i$ limits the values that variables can take, e.g., $V_1 \neq V_2$

**Complete**: Every variable is assigned
**Consistent**: Does not violate any constraint
**CSP solution**: Complete & Consistent assignment

## Advantages of CSP
- **Formal representation language** that can be used to formalize many problem type
- Represent problem as a CSP and solve with general-purpose solver
- Can use **general-purpose solver**, which are more efficient than standard search
- Constraints allow us to focus the search to valid branches
- Branches that violate constraints are removed
- Non-trivial to do this for standard search (need manual selection of actions)

## Constraint Graph
Nodes = Variables, Edges = Constraints

## Variety of CSPs
Discrete Variables
- Finite domains: $O(d^n)$ complete assignments for n variables, domain size d
- Infinite domains: Integer, Strings, etc.

Continuous Variables
- Time, float, etc.

## Variety of Constraints
- **Unary**: Involve single variable
- **Binary**: Involve pairs of variables
- **Higher order**: Involve 3 or more variables
- **Preference** (Soft constraints)

## CSPs as Standard Search
- Can be easily formulated (Initial State, Actions, Path Cost, Goal State)
- Sequence of actions do not matter, only the goal state (i.e., solution at depth n, use DFS)
- However, there are potentially $n!\, d^n$ leaves

## Commutativity
- CSP variable assignments are commutative: i.e., Regardless of variable assignment order
- Only need to consider assignments to a single variable at each level/depth
- reduce from $n!\, d^n$ leaves to $d^n$ leaves

## Backtracking Search
- DFS for CSPs with single variable assignment
- Backtracking occurs when there are no legal values for a variable
- The basic uniformed algorithm for CSPs

## General Purpose Methods
Can give huge gains in speed:

**Minimum remaining values**
- Choose the variable with the fewest legal values
  (i.e., the most constrained variable)

**Degree Heuristic**
- When multiple variables have the same MRV
- Choose the variable with the most constraints on remaining variables

**Least constraining value**
- Given a variable, choose the least constraining value
- The one that rules out the fewest values in the remaining variables

**Forward Checking**
- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values
- propagates information from assigned to unassigned variables
- Does not provide early detection for all failures
- Need to enforce constraints locally

**Arc Consistency**
- Simplest form of propagation make each arc consistent
- $X \rightarrow Y$ is consistent iff. for every value X there is some allowed y
- Arc consistency detects failure earlier than forward checking
- Can be run as a pre-processor or after each assignment
- Ordering of arcs do not matter
- Complexity of $O(n^2 d^3)$:
  - $O(n^2)$: Need to check for all edges, potentially $n^2$ edges
  - $O(d^2)$: For each edge, comparing their two domains
  - $O(d)$: Each variable change repropagate to neighbours, max d times

```
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
      for each Xk in NEIGHBORS[Xi] do
        add (Xk, Xi) to queue
```

```
function REMOVE-INCONSISTENT-VALUES(Xi, Xj) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[Xi] do
    if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
      then delete x from DOMAIN[Xi]; removed ← true
  return removed
```

**Problem Structure**
- Suppose each subproblem has c variables out of n total
- Worst-case solution cost is $\binom{n}{c} \times d^c$

**Tree-Structured CSPs**
- if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time

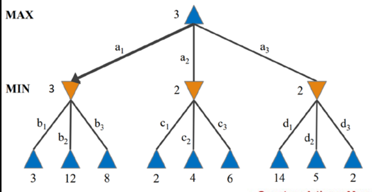
1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
2. For j from n down to 2, apply RemoveInconsistent(Parent$(X_j)$, $X_j$)
3. For j from 1 to n, assign $X_j$ consistently with Parent$(X_j)$

**Nearly Tree-Structured CSPs**
- Conditioning: instantiate a variable, prune its neighbours' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size c ⇒ runtime $O(d^c \cdot (n-c)d^2)$, fast for small c

## Representing Game as Search Problem
- Initial State
- Actions
- Terminal Test (Win/Lose/Draw)
- Utility Function (Numerical reward for the outcome):
  E.g., Chess → +1, 0, -1
  Poker → Cash win or lose
- Zero-sum: each player's utility for a state are equal and opposite

## Minimax
- Perfect play for deterministic, perfect-information (fully observable) games
- Idea: choose moves with highest minimax value
  - best achievable payoff wrt best play

- **Completeness:** Yes, if tree is finite
- **Optimality:** Yes, against optimal opponent
- **Time complexity:** $O(b^m)$
- **Space complexity:** $O(bm)$




Operator = Action or Move

```
function MINIMAX-DECISION(game) returns an operator
   for each op in OPERATORS[game] do
      VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
   end
   return the op with the highest VALUE[op]

function MINIMAX-VALUE(state, game) returns a utility value
   if TERMINAL-TEST[game](state) then
      return UTILITY[game](state)
   else if MAX is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
   else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

## Cutting Off Search

- MinimaxCutoff is identical to MinimaxValue except
- Terminal is replaced by Cutoff?
- Utility is replaced by Eval

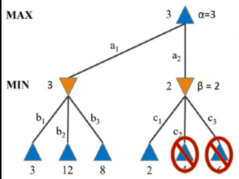
```
function MINIMAX-DECISION(game) returns an operator
   for each op in OPERATORS[game] do
      VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
   end
   return the op with the highest VALUE[op]

function MINIMAX-VALUE(state, game) returns a utility value
   if TERMINAL-TEST[game](state) then
      return UTILITY[game](state)
   else if MAX is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
   else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Replaced by Cutoff-Test

Replaced by Eval score

## α-β Pruning

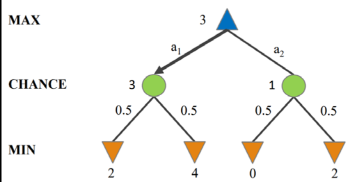
- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering", time complexity is O(bm/2)
    - doubles depth of search
    - can easily reach depth 8 and play good chess
- A simple example of the value of reasoning about which computations are relevant (a form of meta-reasoning)



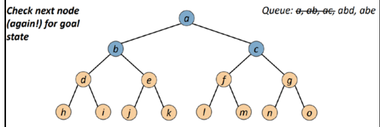

- α is the best value (to MAX) found so far off the current path
- β is the best value (to MIN) found so far off the current path
- Prune if α >= β

## ExpectiMiniMax (Non-deterministic Games)

- Accounts for chance nodes
- If state is a chance node, return weighted average expected values of child nodes

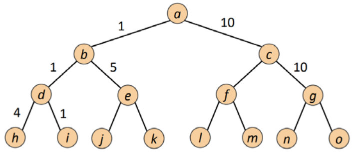



## Breadth-First Search



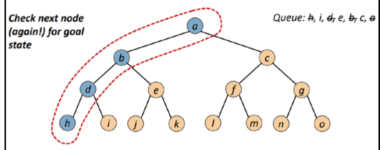

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
   node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   frontier ← a FIFO queue with node as the only element
   explored ← an empty set
   loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the shallowest node in frontier */
      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
            if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
            frontier ← INSERT(child, frontier)
```

## Uniform-Cost Search

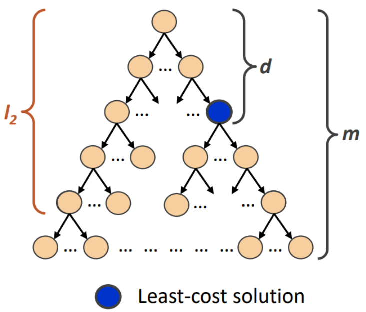



```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
   node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
   frontier ← a priority queue ordered by PATH-COST, with node as the only element
   explored ← an empty set
   loop do
      if EMPTY?(frontier) then return failure
      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
```

## Depth-First Search




## Depth-Limited Search




## Iterative Deepening Search