Unit 2: Problem Solving and Search

Artificial Intelligence

Kiran Bagale

November 2025

Outline

Problem Solving and Search

Overview

- Duration: 9 hours
- Focus: Fundamental search algorithms and optimization techniques
- Applications: Pathfinding, game playing, constraint satisfaction

2.1: What is a Problem?

A well defined problem in AI consists of five components:

- **Initial state**: Where the agent starts
- **Actions**: Set of possible actions available to the agent
- **Transition model**: Result of taking an action in a state
- **Goal test**: Determines if a state is a goal state
- **Path cost**: Numerical cost of a path (sum of step costs)

A **solution** is a sequence of actions that leads from the initial state to the goal state.

Problem Solving

Problem solving is fundamental to many AI-based applications.

Two types of problem.

- The Problems like the computation of the sine of an angle or the square root of a value can be solved through the use of a deterministic procedure, and success is guaranteed.
- In the real world, very few problems lend themselves to straightforward solutions.

Most real world problems can be solved only by searching for a solution.
AI is concerned with these type of problems solving.

Problem solving is a process of generating solutions from observed data.

- a problem is characterized by a set of goals,
- a set of objects, and
- a set of operations.

These could be ill-defined and may evolve during problem solving.

Problem Solving Con...

Problem space

It is an abstract space.

- A problem space encompasses all valid states that can be generated by the application of any combination of operators on any combination of objects.
- The problem space may contain one or more solutions.

Solution is a combination of operations and objects that achieve the goals.
Search refers to the search for a solution in a problem space.

- Search proceeds with different types of search control strategies.
- The depth-first search and breadth-first search are the two common search strategies.

Problem Solving Con...

To build a system to solve a particular problem

- **Define the problem precisely** - find input situations as well as final situations for acceptable solution to the problem.
- **Analyze the problem** - find few important features that may have impact on the appropriateness of various possible techniques for solving the problem.
- Isolate and represent task knowledge necessary to solve the problem.
- Choose the best problem solving technique(s) and apply to the particular problem.

Well-Defined Problems

A problem is **well-defined** if:

1. The initial state is clearly specified
2. Actions available in each state are clearly defined
3. The transition model is deterministic and known
4. The goal test is clearly defined
5. The path cost function is clearly defined

Example: 8-puzzle, route finding, vacuum world

Search

The problem can then be solved by using the rules, in combination with an appropriate control strategy, to move through the problem space until a path from an initial state to a goal state is found.

States, Actions, and Transitions

Problem Definition

A **problem** is defined by its *elements* and their *relations*.

State Space

The set of all states reachable from the initial state by any sequence of actions, including some impossible ones.

Specify one or more states, that describe possible situations, from which the problem-solving process may start called *initial states*.

Actions

Given state s, ACTIONS(s) returns the set of actions that can be executed in s.

Transition Model

RESULT(s, a) returns the state that results from doing action a in state s.

State Space Search

Used in problem solving
It is a process in AI, where successive configurations or states of an instance are considered within finding a Goal state with desired property.

Problems are modeled as **State Space** - set of states in which a problem can be.
Representations:
$S : (S, A, Actions(S), Results(S, a), Costs(S, A))$

where,
S - set of all possible states,
A - set of all possible actions,
Actions(S) - Function[which action is possible for current state],
Results(S,a) - function[state reached by performing actions 'a' on state 'S'],
Costs(S,A) - total cost on achieving Goal

Example: 8-Puzzle

Problem Components

- **States**: 3×3 grid configurations
- **Actions**: {Up, Down, Left, Right}
- **Initial State**: Random configuration
- **Goal State**: Ordered tiles

Initial State:

7	2	4
5		6
8	3	1

Goal State:

	1	2
3	4	5
6	7	8

Actions: Move blank Up, Down, Left, Right
Path cost: Number of moves

Problem Space Representation

Represented by

- **Directed graph**, where nodes represent search state and paths represent the operators applied to change the state.
- **A tree**, to simplify the algorithms.

A tree usually decreases the complexity of a search at a cost. Here, the cost is due to duplicating some nodes on the tree that were linked numerous times in the graph; e.g., node **B** and node **D** shown in example below.

Tree

A tree is a graph in which any two vertices are connected by exactly one path. Alternatively, any connected graph with no cycles is a tree.

Tree vs Graph

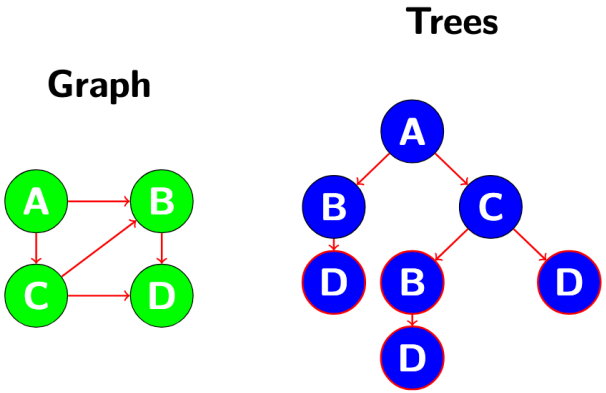


Figure: Paths in Graph and trees

Problem Solution

Solution Path

Definition: A solution is a path from the initial state to a goal state in the state space.

Cost Function: Assigns numeric cost to each path and operators

Solution Quality: Measured by path cost function

Optimal Solution: Has the lowest path cost among all solutions

Solution Types: Any, optimal, or all solutions

The importance of cost depends on the problem type and solution requirements.

Problem Formulation Components

State Space

- Current world state
- Initial state (start)
- Goal state (desired)

Operators

- Transform states
- Preconditions
- Instructions

Problem Solving

- Find operator sequence
- Start $\rightarrow$ Goal state
- Minimize steps or cost

Solution Quality

- Shortest sequence
- Least expensive path
- Any quick solution

Key Principle: State space should include everything needed to solve the problem—no more, no less.

Single-State vs Multi-state Problem Formulation

Single State

Exact prediction is possible

State - exactly known after any action sequence.

Accessibility - information through Sensors.

Consequences - action known to agents.

Goal - unique fora each known initial state

Simple but restricted eg: Vacuum world - can't deal with incomplete accessibility, incomplete knowledge about consequences can alter the world

Multiple State

Semi-exact prediction is possible

State - not exactly known but limited to set of possible states.

Accessibility - not all information through Sensors. [Reasoning]

Consequences - not always known to agents. [Randomness]

Goal - no fixed action sequence that leads to Goal

Less restricted, more complex eg: Vacuum world - but no sensors, cannot deal with changing world during execution.

2.2: Constraint Satisfaction Problems (CSPs)

CSP Definition

A CSP consists of:

- Variables: $X = \{X_1, X_2, \ldots, X_n\}$
- Domains: $D = \{D_1, D_2, \ldots, D_n\}$
- Constraints: $C = \{C_1, C_2, \ldots, C_m\}$

Goal: Find an assignment of values to variables that satisfies all constraints. **Examples**: Map coloring, N-Queens, Sudoku, scheduling

Types of Consistency

- **Node Consistency**: Each variable satisfies unary constraints
- **Arc Consistency**: For every value in X_i, there exists a consistent value in X_j
- **Path Consistency**: For any pair of values, there exists a consistent assignment for intermediate variables

Node Consistency

Definition

A variable is **node-consistent** if all values in its domain satisfy the variable's unary constraints.

Example: If variable X has domain $\{1, 2, 3, 4\}$ and constraint $X < 3$, then after node consistency: $D_X = \{1, 2\}$

Node consistency is easy to achieve: simply remove values that violate unary constraints.

Arc Consistency

Definition

A variable X_i is **arc-consistent** with respect to another variable X_j if for every value in the domain of X_i, there exists some value in the domain of X_j that satisfies the binary constraint between them.

Example: Map coloring

- Variables: WA, NT, SA (Western Australia, Northern Territory, South Australia)
- Domains: $\{\text{red}, \text{green}, \text{blue}\}$
- Constraint: Adjacent regions must have different colors

If we assign WA = red, then NT must be arc-consistent by removing red from its domain.

Path Consistency

Definition

A pair of variables $\{X_i, X_j\}$ is **path-consistent** with respect to a third variable X_k if for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there exists an assignment to X_k that satisfies the constraints on $\{X_i, X_k\}$ and $\{X_j, X_k\}$.

Path consistency is stronger than arc consistency but more expensive to enforce.

Backtracking Algorithm: The basic uninformed algorithm for CSPs

```
1: function BACKTRACK(assignment, csp)
2:     if assignment is complete then
3:         return assignment
4:     end if
5:     var ← SELECTUNASSIGNEDVARIABLE(csp)
6:     for each value in ORDERDOMAINVALUES(var, assignment, csp) do
7:         if value is consistent with assignment then
8:             add {var = value} to assignment
9:             result ← BACKTRACK(assignment, csp)
10:            if result ≠ failure then
11:                return result
12:            end if
13:            remove {var = value} from assignment
14:        end if
15:    end for
16:    return failure
17: end function
```

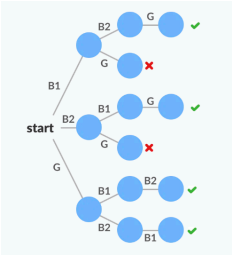
Example

Problem

You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.
Constraint: Girl should not be on the middle.

Solution: There are a total of 3! = 6 possibilities. We will try all the possibilities and get the possible solutions. We recursively try all the possibilities.
All the possibilities are:

Figure: All possible states



Infrastructure for search algorithms

- n.STATE: the state in the state space to which the node corresponds; n.PARENT: the node in the search tree that generated this node; n.ACTION: the action that was applied to the parent to generate the node; n.PATH-COST: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

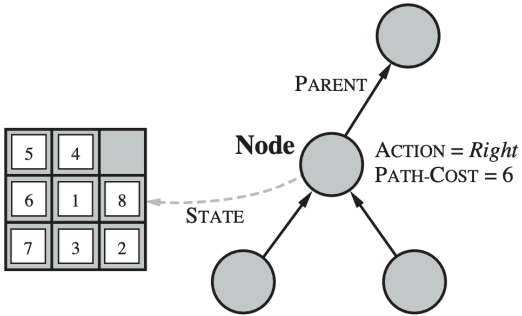


Figure: Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Queue

Operation in Queue

The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a queue. The operations on a queue are as follows:

- **EMPTY?(queue)** returns true only if there are no more elements in the queue.
- **POP(queue)** removes the first element of the queue and returns it.
- **INSERT(element**, queue) inserts an element and returns the resulting queue.

Queues are characterized by the order in which they store the inserted nodes. Three common variants are the **first-in, first-out or FIFO queue**, which pops the oldest element of the queue; the **last-in, first-out or LIFO queue** (also known as a **stack**), which pops the newest element of the queue; and the **priority queue**, which pops the element of the queue with the highest priority according to some ordering function.

Measuring problem-solving performance

- **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
- **Optimality:** Does the strategy find the optimal solution?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

Time and space complexity are always considered with respect to some measure of the problem difficulty.
In theoretical computer science, the typical measure is the size of the state space graph, $|V| + |E|$, where V is the set of vertices (nodes) of the graph and E is the set of edges (links).
search cost — typically depends on the **time complexity**
total cost - combines the **search cost** + the **path cost** of the solution found.

2.3 Search Strategies Overview

Uninformed (Blind) Search Strategies Key Idea: No domain knowledge; rely only on state space structure.

Algorithms

- **Breadth-First Search (BFS)** – explores level by level
- **Depth-First Search (DFS)** – explores deep paths first
- **Depth-Limited Search (DLS)** – DFS with cut-off
- **Iterative Deepening Search (IDS)** – repeated DLS with increasing depth
- **Uniform Cost Search (UCS)** – expands lowest-cost path

Properties:

- Completeness: Always finds solution if one exists
- Optimality: Finds least-cost solution, BFS (unit cost), UCS
- Time complexity: Number of nodes generated, $O(b^d)$ or higher
- Space complexity: Maximum nodes in memory, BFS high, DFS low

Informed (Heuristic) Search Strategies

Idea: Use heuristic function $h(n)$ to estimate cost to goal.

Algorithms

- **Greedy Best-First Search** – expand lowest $h(n)$
- **A Search** – $f(n) = g(n) + h(n)$
- **IDA** – depth-first + A thresholds
- **Beam Search** – keep best k nodes only

Properties:

- Uses domain knowledge
- Fast, efficient for large spaces
- Optimality depends on heuristic

Local Search Algorithms

Idea: Work with a single current state; move to a neighboring state.

Algorithms

- **Hill Climbing** – move to best neighbor
- **Simulated Annealing** – probabilistic acceptance
- **Genetic Algorithm** – population + crossover + mutation
- **Local Beam Search** – track k best states

Usage:

- Optimization problems
- Large or continuous state spaces

Adversarial Search

Used for: Two-player perfect information games.

Key Algorithms

- **Minimax Algorithm** – MAX vs MIN optimal play
- **Alpha-Beta Pruning** – eliminates irrelevant branches
- **Expectimax** – for stochastic environments

Properties:

- Optimal vs. optimal opponent
- Time: $O(b^m)$ (exponential)
- Alpha-Beta improves to $O(b^{m/2})$ in best case

Evaluation of Search Strategies

Criteria

- **Completeness:** Finds solution if one exists
- **Optimality:** Guarantees best (lowest cost) solution
- **Time Complexity:** Usually $O(b^d)$ or $O(b^m)$
- **Space Complexity:** Memory used during search

Key Variables

- b: branching factor
- d: depth of optimal solution
- m: maximum depth of search tree

Search Algorithms: Summary Table

Algorithm	**Complete**	**Optimal**	**Time**	**Space**
BFS	Yes	Yes (unit cost)	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	$O(m)$
UCS	Yes	Yes	Exp.	Exp.
IDS	Yes	Yes	$O(b^d)$	$O(d)$
Greedy	No	No	varies	varies
A	Yes	Yes (adm. h)	Exp.	Exp.
Hill Climbing	No	No	Low	Low
GA	No	No	varies	med.
Minimax	Yes	Yes	$O(b^m)$	$O(bm)$
α-β	Yes	Yes	$O(b^{m/2})$	$O(bm)$

2.4 Uninformed Search Algorithms

Algorithm	Complete	Optimal	Time	Space
BFS	Yes	Yes*	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	$O(bm)$
Iterative Deepening	Yes	Yes*	$O(b^d)$	$O(bd)$
Uniform Cost	Yes	Yes	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$

Table: *Optimal if step costs are equal

- b: branching factor
- d: depth of shallowest(Optimal) solution
- m: maximum depth of search tree

Breadth-First Search (BFS)

A fundamental graph traversal algorithm widely used in Artificial Intelligence (AI) and computer science.

It systematically explores the vertices of a graph layer by layer, ensuring that all nodes at the current depth are visited before moving to the next level. This approach makes BFS particularly effective in scenarios requiring the shortest path or exhaustive exploration of possibilities.

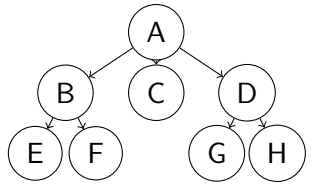
Strategy: Expand shallowest unexpanded node

Implementation: Queue (FIFO)

Properties:

- Complete: Yes
- Optimal: Yes (if step cost = 1)
- Time: $O(b^d)$
- Space: $O(b^d)$

where b = branching factor, d = depth of solution



Order: A, B, C, D, E, F, G, H

BFS Cont...

BFS Pseudocode

1. Initialize an empty queue.
2. Enqueue the starting node and mark it as visited.
3. While the queue is not empty:

a. Dequeue a node from the queue.
b. Process the node (e.g., print or store it).
c. Enqueue all unvisited adjacent nodes and mark them as visited.

Real-World AI Applications

Network Broadcasting - broadcasting messages in computer networks in the shortest possible time.

Peer-to-Peer Networks - In distributed systems like peer-to-peer networks, BFS is utilized to locate resources or peers efficiently.

Medical Diagnosis Systems - In AI-driven healthcare systems, BFS is applied to trace the spread of diseases through contact networks.

Depth-First Search (DFS)

Better for exploring deeper nodes first, making it suitable for scenarios where depth is prioritized over breadth.

DFS generally uses less memory as it does not store all neighboring nodes at once, unlike BFS.

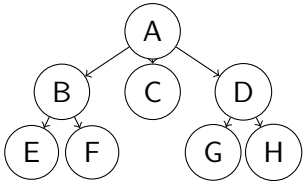
Strategy: Expand deepest unexpanded node

Implementation: Stack (LIFO)

Properties:

- Complete: No (can get stuck in loops)
- Optimal: No
- Time: $O(b^m)$
- Space: $O(bm)$

where m = maximum depth



Order: A, B, E, F, C, D, G, H

Use Cases: Puzzle solving (e.g., Sudoku), topological sorting, and finding connected components in graphs.

Iterative Deepening Search (IDS)

Idea: Combine benefits of BFS and DFS

Strategy:

- Perform DFS with depth limit 0
- If no solution, increase limit to 1
- Repeat, increasing depth limit each iteration

Properties:

- Complete: Yes
- Optimal: Yes (if step cost $= 1$)
- Time: $O(b^d)$
- Space: $O(bd)$

Advantage: Memory efficiency of DFS with optimality of BFS!

2.5 Informed Search: Best First Search

Idea: Use an **evaluation function**

$$f(n) = g(n) + h(n)$$

to select which node to expand
Implementation: Priority queue ordered by $f(n)$

Special cases:

- **Greedy Best-First**: $f(n) = h(n)$ Uses only $h(n)$ to select next node
- **A* Search**: $f(n) = g(n) + h(n)$

where:

- $g(n)$ = cost from start to node n
- $h(n)$ = estimated cost from n to goal (heuristic)

Disadvantages:

- Fast but not optimal
- Can get stuck in local minima

Greedy Best-First Search

Evaluation function: $f(n) = h(n)$

Expands node that *appears* to be closest to goal.

Example: Route finding with straight-line distance heuristic

Properties:

- Complete: No (can get stuck in loops)
- Optimal: No
- Time: $O(b^m)$ (worst case)
- Space: $O(b^m)$

Issue: Ignores actual path cost $g(n)$!

A* Search

Evaluation function: $f(n) = g(n) + h(n)$

- $g(n)$ = actual cost from start to n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n

Optimality Condition: A* is optimal if $h(n)$ is **admissible**

Admissible Heuristic

A heuristic $h(n)$ is admissible if it never overestimates the cost to reach the goal:

$$h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost from n to goal.

A* Search Properties

Properties:

- Complete: Yes (with finite branching factor)
- Optimal: Yes (if $h(n)$ is admissible)
- Time: Exponential in $[h^*(n) - h(n)]$
- Space: Keeps all nodes in memory: $O(b^d)$

Optimally Efficient: Among all optimal algorithms using the same heuristic, A* expands the fewest nodes.

Main Problem: Space complexity! (see Memory-Bounded Search)

A* Algorithm

```
1: frontier ← priority queue initialized with start node
2: explored ← empty set
3: while frontier is not empty do
4:     node ← PopLowestF(frontier)                    ▷ node with lowest f-value
5:     if node is goal then
6:         return solution
7:     end if
8:     add node to explored
9:     for each child of node do
10:        if child ∉ explored and child ∉ frontier then
11:            add child to frontier
12:        else if child ∈ frontier and f(child) is lower then
13:            update child in frontier with the lower f-value
14:        end if
15:    end for
16: end while
17: return failure
```

2.6: Adversarial Search

Context: Multi-agent environments where agents have conflicting goals

Game Playing Scenario

- Two or more agents with conflicting goals
- Turn-based, deterministic, perfect information
- Examples: Chess, Checkers, Go

Game Properties:

- Two-player
- Zero-sum: One player's gain is other's loss
- Turn-taking
- Deterministic
- Perfect information: Both players see entire state

Goal: Find optimal strategy (move sequence)

Key Algorithm: Minimax with Alpha-Beta pruning

Minimax Algorithm

Idea:

- MAX player tries to maximize score
- MIN player tries to minimize score
- Assume opponent plays optimally

Minimax value of node n:

$$\text{MINIMAX}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is terminal} \\ \max_{s \in \text{succ}(n)} \text{MINIMAX}(s) & \text{if } n \text{ is MAX node} \\ \min_{s \in \text{succ}(n)} \text{MINIMAX}(s) & \text{if } n \text{ is MIN node} \end{cases}$$

Properties:

- Complete: Yes
- Optimal: Yes (against optimal opponent)
- Time: $O(b^m)$
- Space: $O(bm)$

Alpha-Beta Pruning

Idea: Prune branches that cannot influence final decision

Optimization of Minimax

- α = best value for MAX along path to root
- β = best value for MIN along path to root

Pruning conditions:

- At MIN node: if value $\leq \alpha$, prune remaining children
- At MAX node: if value $\geq \beta$, prune remaining children

Effectiveness:

- Best case: $O(b^{m/2})$ (perfect ordering)
- Can effectively double solvable depth
- Same result as Minimax, just faster

2.7 Local Search and Optimization

Idea: Keep only current state, move to neighbors

Advantages:

- Very memory efficient: $O(1)$ space
- Can find reasonable solutions in large/infinite spaces
- Useful for optimization problems

Applications:

- Don't care about path, only solution state
- Integrated circuits layout
- Job shop scheduling
- Protein folding

Hill Climbing

Strategy: Continually move to neighbor with highest value

- Iteratively move to neighbor with better value
- Problems: Local maxima, plateaus, ridges
- Variants: Stochastic, First-choice, Random-restart

Algorithm 1 Hill Climbing

```
1: current ← initial state
2: loop
3:     neighbor ← BestSuccessor(current)
4:     if Value(neighbor) ≤ Value(current) then
5:         return current
6:     end if
7:     current ← neighbor
8: end loop
```

Simulated Annealing

Idea: Allow "bad" moves to escape local maxima

Probability of accepting worse move:

$$P(\text{accept}) = e^{\Delta E / T}$$

where:

- ΔE = change in evaluation function (negative if worse)
- T = "temperature" parameter

Temperature schedule:

- Start with high T (more random exploration)
- Gradually decrease T (more greedy exploitation)
- At $T \to 0$, becomes hill climbing

Properties: Complete and optimal with proper schedule

2.8: Genetic Algorithms

Inspired by: Natural selection and evolution

Components:

- **Population**: Set of candidate solutions (individuals)
- **Fitness function**: Evaluates quality of each individual
- **Selection**: Choose individuals for reproduction
- **Crossover**: Combine two parents to create offspring
- **Mutation**: Random changes to individuals

Representation: Typically bit strings or real-valued vectors

Genetic Algorithm Process

Algorithm 2 Genetic Algorithm

```
1: Initialize population randomly
2: repeat
3:     Evaluate fitness of each individual
4:     Select parents based on fitness
5:     Apply crossover to produce offspring
6:     Apply mutation to offspring
7:     Replace population with offspring
8: until termination condition is met
9: return best individual
```

Advantages:

- Parallel search of solution space
- Can handle discontinuous, noisy fitness functions
- Effective for complex optimization problems

Summary: Search Strategy Comparison

Algorithm	Complete	Optimal	Time	Space
BFS	Yes	Yes*	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	$O(bm)$
IDS	Yes	Yes*	$O(b^d)$	$O(bd)$
Greedy	No	No	$O(b^m)$	$O(b^m)$
A*	Yes	Yes**	Exponential	$O(b^d)$
Hill Climb	No	No	-	$O(1)$
Sim. Anneal	Yes***	Yes***	-	$O(1)$

* If step costs are equal
** If heuristic is admissible
*** With proper temperature schedule

Key Takeaways

- **Problem formulation** is crucial: states, actions, goals, costs
- **Uninformed search**: No domain knowledge (BFS, DFS, IDS)
- **Informed search**: Use heuristics (Greedy, A*)
- **A*** is optimal with admissible heuristics
- **CSPs**: Special structure allows constraint propagation
- **Adversarial search**: Minimax with alpha-beta pruning
- **Local search**: Memory efficient, good for optimization
- **Genetic algorithms**: Population-based evolutionary approach

References

- Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
- Rich, E., Knight, K., Nair, S. B. (2009). *Artificial intelligence.* (McGraw-Hill.)

Thank You!

Questions?