CCES – College of Computer Engineering and Sciences

Prince Sattam Bin Abdulaziz University

CS600: Advanced AI

Searching State Spaces

Chapter 2 — Uninformed, Informed, and Local Search

Dr. Khalil Chebil

Artificial Intelligence Course

Chapter Outline | What we will cover

Search Problems & State Spaces
Uninformed Search
Informed Search
Local Search
Genetic Algorithms

Learning Objectives

- Represent problems as **state-space graphs**
- Master **uninformed** search: BFS, DFS, UCS, bidirectional
- Master **informed** search: greedy best-first and A*
- Design **admissible** heuristics; apply **local search** and **genetic algorithms**

Section 1

Search Problems & State Spaces

What is Search? | Finding a path from start to goal

Many AI problems reduce to finding a **sequence of actions** transforming an **initial state** into a **goal state**.

Two scenarios

- **Goal-oriented**: find a path (maze, puzzle)
- **Optimization**: find a best state (8-queens, scheduling)

Why search matters

- No closed-form solution
- Huge solution spaces
- Needs systematic exploration
- Trades time, memory, quality

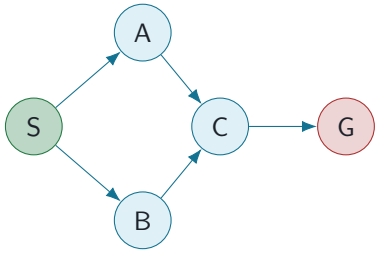
Applications: route planning ▪ puzzles ▪ game playing ▪ scheduling ▪ robot navigation

State Space as a Graph | Nodes, edges, paths

- **Node** — a state
- **Edge** — an action / transition
- **Path** — sequence from start to goal

State-space size explodes

8-Puzzle	$9! = 362{,}880$
8-Queens	$\binom{64}{8} \approx 4.4 \times 10^8$
Rubik's Cube	$\approx 4.3 \times 10^{19}$
Chess	$> 10^{43}$



Large spaces can't be stored — explore states dynamically.

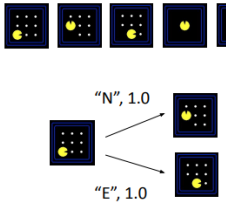
State-Space Graph vs. Search Tree | The tree can be far larger

Aspect	State-Space Graph	Search Tree
Definition	All states & transitions	Paths explored by the algorithm
Duplicates	Each state once	Same state may repeat
Size	Fixed by problem	Can be infinitely larger
Cycles	May contain cycles	Unrolled into branches

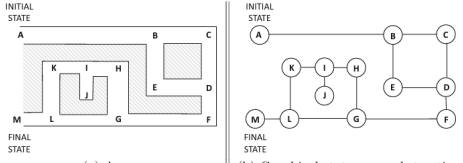
Consequence

A cycle ($a \leftrightarrow b$) makes the tree infinite. Search algorithms must track **visited states** to avoid infinite loops.

Classic Search Problems | Modeling states and actions

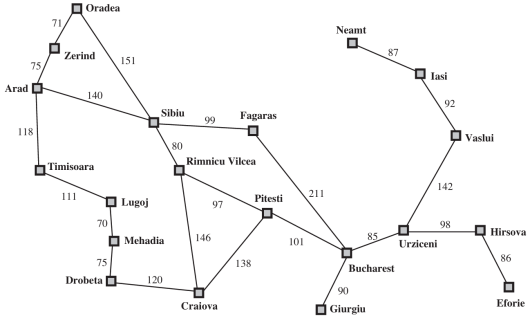


Pacman: position + food; move N/E/S/W



(a) A maze (b) Graphical state-space abstraction

Maze: position; move in 4 directions



Romania: current city; roads with distances; start Arad → goal Bucharest

Search Problem: 8-Queens | Formalizing state, actions, and goal

The State Space

Each state is a configuration of the board with a certain number of queens placed on it.

Successor Function

Generates every valid configuration reached by adding **one more queen**; e.g. with 3 queens already placed, it tries every column of the next row to produce all valid 4-queen configurations.

Start State & Goal Test

Start: an empty chessboard. **Goal**: all 8 queens placed with no two attacking each other (same row, column, or diagonal).

Dead end: a partial placement (say, 6 queens) may admit *no* valid 7-queen extension — the successor function returns nothing, and that branch is abandoned.

Section 2

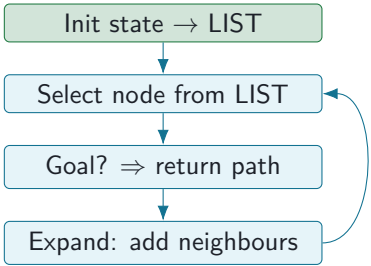
Uninformed Search

Generic Search Framework | One template, many strategies

Components

- **LIST** — frontier (nodes to explore)
- **VISIT** — already explored
- **pred** — predecessors for path recovery
- **Selection strategy** — defines the algorithm

FIFO $\Rightarrow$ BFS LIFO $\Rightarrow$ DFS priority $\Rightarrow$ UCS / A*



Generic Search Algorithm | The pseudocode behind BFS, DFS, UCS, ...

```
Algorithm GenericSearch(Initial State: s, Goal Condition: G)
begin
  LIST = { s };
  repeat
    Select current node i from LIST based on pre-defined strategy;
    Delete node i from LIST;  Add node i to hash table VISIT;
    for all nodes j in A(i) directly connected to i via transition do
      if (j is not in VISIT) add j to LIST;
      pred(j) = i;
  until LIST is empty or current node i satisfies G;
  if current node satisfies G return success;
  else return failure;
  { pred(.) is used to trace back the path from i to s }
end
```

FIFO $\Rightarrow$ BFS **LIFO** $\Rightarrow$ DFS **priority queue** $\Rightarrow$ UCS / Greedy / A*

Breadth-First Search (BFS) | Level-by-level, FIFO queue

Explores all nodes at depth d before depth $d+1$.

- Insert at **end**, remove from **front** (FIFO)
- Finds the **shallowest** goal

Good for

Shallow goals; optimal when all step costs are equal.

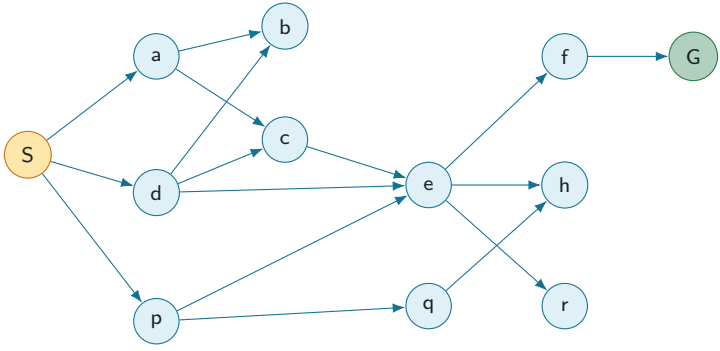
Cost

Memory heavy — stores the whole frontier.

Property	Value
Complete	Yes
Optimal	Yes* (equal costs)
Time	$\mathcal{O}(b^d)$
Space	$\mathcal{O}(b^d)$

b: branching factor d: depth of goal

Breadth-First Search — Worked Example | The same graph, explored level by level



Dequeue order (successors generated left-to-right, e.g. $A(S) = \{a, d, p\}$): S, a, d, p, b, c, e, q, f, h, r, G — every shallower node is fully expanded before the depth-4 goal G is reached.

Depth-First Search (DFS) | Go deep, LIFO stack

Explores as deep as possible, then backtracks.

- Insert / remove at **end** (LIFO)
- Expands most recent node first

Good for

Deep solutions, memory-limited settings.

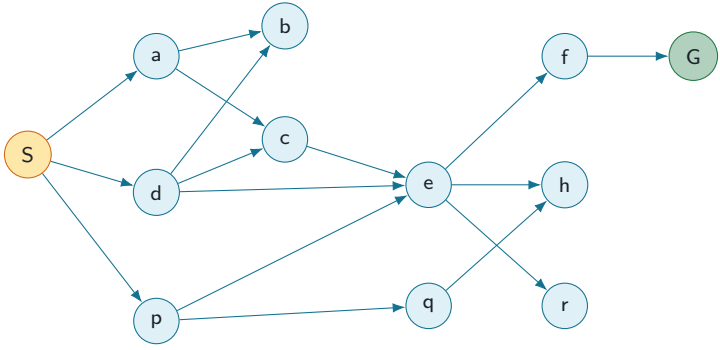
Risk

Can loop forever (needs cycle detection); not optimal.

Property	Value
Complete	No (with cycles)
Optimal	No
Time	$\mathcal{O}(b^m)$
Space	$\mathcal{O}(bm)$

m: maximum depth of the tree

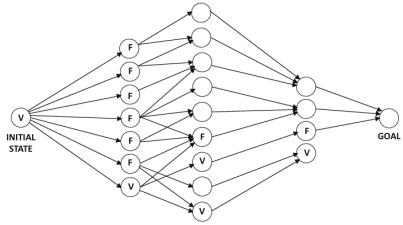
Depth-First Search — Worked Example | The same graph, explored via a LIFO stack



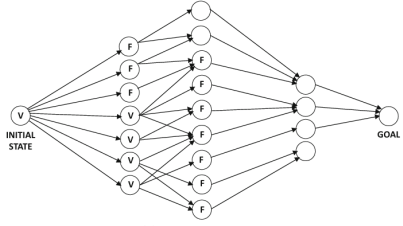
Pop order (pushing S's successors as a, d, p leaves p on top of the stack): S, p, e, r, h, f, G — the goal is reached in just 7 expansions, entirely bypassing a, b, c, d, and q.

Caveat: a different push order could send DFS down the a→c→e branch instead — the path found is not guaranteed to be cheapest.

BFS vs. DFS | Frontier shape trade-off



(a) Depth-first search



(b) Breadth-first search

Uniform-Cost Search (UCS) | BFS for weighted graphs

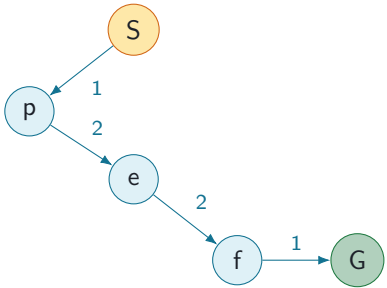
Expand the node with the **lowest path cost** $g(n)$.

- Priority queue ordered by $g(n)$
- $g(n)$ = sum of edge costs from start
- Optimal for any non-negative costs

Complexity

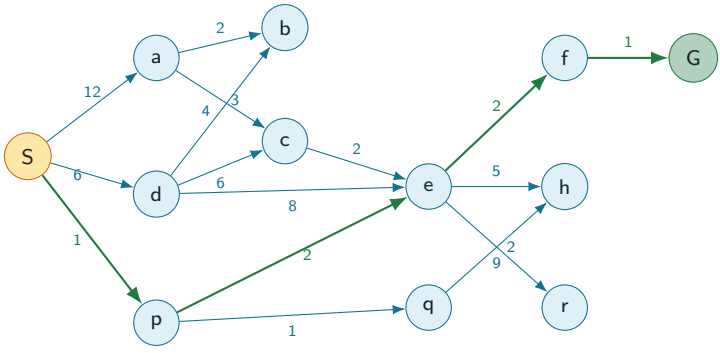
$\mathcal{O}\left(b^{C^*/\varepsilon}\right)$ time and space, where C^* is the optimal cost and ε the minimum edge cost.

UCS $\equiv$ BFS when all edge costs are equal.



Cheapest-first expansion order

Uniform-Cost Search — Worked Example | The same graph, now with edge costs



Cheapest path: S→p→e→f→G, cost 1+2+2+1 = 6 — cheaper than S→d→e→f→G (cost 17) or S→a→c→e→f→G (cost 20).

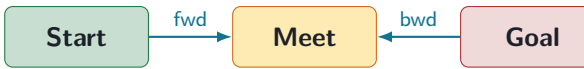
Bidirectional Search | Meet in the middle

Run two searches: **forward** from start and **backward** from goal; stop when they meet.

Speed-up

$\mathcal{O}(b^d) \rightarrow \mathcal{O}(b^{d/2})$

b=10, d=6 : $10^6 \rightarrow 2 \times 10^3$

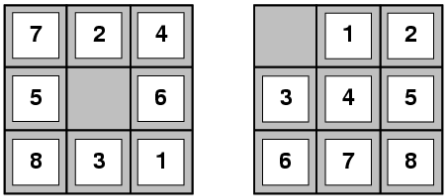


Needs predecessors and a meeting test; goal must be explicit.

Bidirectional Search | Pseudocode: two frontiers, two visited sets

```
Algorithm BiSearch(Initial State: s, Goal State: g)
begin
  FLIST = { s };  BLIST = { g };
  repeat
    Select if from FLIST, ib from BLIST (pre-defined strategy);
    Delete if from FLIST, ib from BLIST;
    Add if to FVISIT, ib to BVISIT;
    for each j in A(if) not in FVISIT: add j to FLIST; pred(j)=if;
    for each j in B(ib) not in BVISIT: add j to BLIST; succ(j)=ib;
  until FLIST or BLIST empty, or (FLIST intersect BLIST) != {};
  if (FLIST or BLIST empty) return failure; else return success;
  { Reconstruct the path via any node in FLIST intersect BLIST,
    tracing pred(.) back toward s and succ(.) forward toward g }
end
```

Case Study: The Eight-Puzzle | A 3×3 sliding-tile benchmark



A start and a goal configuration

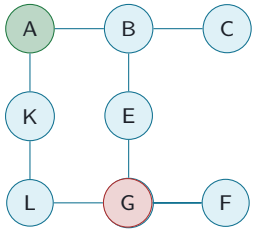
- **State space**: each arrangement of the 8 tiles + blank
- **Successor function**: slide a tile adjacent to the blank
- **Start state**: the given initial arrangement
- **Goal test**: tiles match the target arrangement

Why it matters

$9! = 362{,}880$ reachable states — small enough to benchmark BFS/DFS/UCS/A*, large enough to reveal their differences.

Case Study: Online Maze Search | Abstracting a maze into a graph

- **State space**: the agent's current cell in the maze
- **Successor function**: legal moves (up/down/left/right) allowed by the walls
- **Start state**: the agent's initial cell
- **Goal test**: the agent reaches the target cell



Maze walls $\Rightarrow$ missing edges between adjacent cells

Uninformed Search: Summary | Choose by structure and resources

Algorithm	Compl.	Opt.	Time	Space
BFS	Yes	Yes*	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$
DFS	No	No	$\mathcal{O}(b^m)$	$\mathcal{O}(bm)$
UCS	Yes	Yes	$\mathcal{O}(b^{C^*/\varepsilon})$	$\mathcal{O}(b^{C^*/\varepsilon})$
Bidirectional	Yes	Yes*	$\mathcal{O}(b^{d/2})$	$\mathcal{O}(b^{d/2})$

Key takeaway: uninformed search is inefficient for large spaces $\Rightarrow$ use **informed search**.

Section 3

Informed Search

A **heuristic** $h(n)$ estimates the cost from n to the nearest goal.

Evaluation functions

- $g(n)$ — actual cost start $\to n$
- $h(n)$ — estimated cost $n \to$ goal
- $f(n)$ — estimated total through n

Key properties

- **Admissible**: $h(n) \leq h^*(n)$ (never overestimates)
- **Consistent**: $h(n) \leq c(n, n') + h(n')$

Consistency $\Rightarrow$ admissibility (not vice versa).

8-puzzle: misplaced tiles, or sum of Manhattan distances.

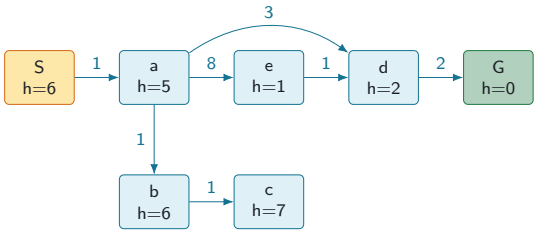
Greedy Best-First Search | $f(n) = h(n)$

Expand the node that *appears* closest to the goal.

- Priority queue ordered by $h(n)$
- Ignores the cost already paid, $g(n)$
- Fast, but **not optimal / not complete**

Typical heuristics

straight-line distance ▪ Manhattan distance ▪ misplaced tiles



Two ways to reach d: direct edge (cost 3) or via e (cost 8+1=9). Greedy compares only h: $h(e)=1 < h(d)=2$, so it follows S→a→e→d→G (total cost 12) — **missing** the cheaper S→a→d→G (total cost 6).

A* Search | $f(n) = g(n) + h(n)$

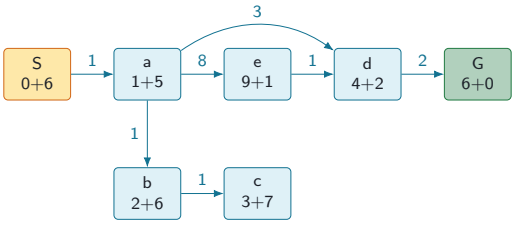
The most widely used informed search: balances cost **paid** and cost **remaining**.

$$f(n) = \underbrace{g(n)}_{\text{start}\to n} + \underbrace{h(n)}_{n\to\text{goal}}$$

- Priority queue ordered by $f(n)$
- Expands the most promising *total-cost* node

Optimally efficient

With a given admissible h, no optimal algorithm expands fewer nodes.



Same graph as Greedy: $f(d)=4+2=6 < f(e)=9+1=10$, so A* expands d directly and returns S→a→d→G (cost 6) — the shortcut that greedy **missed**.

A* Search | Combining Uniform-Cost and Greedy Best-First

```
... choose the node j according to f(j):
  f(j) = min{ c(j) }          // Uniform-Cost Search
  f(j) = min{ h(j) }          // Greedy Best-First Search
  f(j) = min{ c(j) + h(j) }   // A* Search
```

Same `GenericSearch` skeleton (LIST, VISIT, pred) — only the priority function $f(j)$ changes between UCS, Greedy, and A*.

Why A* is Optimal | Admissibility guarantees optimality

Proof sketch

Suppose A* returned suboptimal G_2 (cost C_2), while optimal G has cost $C^* < C_2$. Some node n on the optimal path is unexpanded. Since h is admissible:

$$f(n) = g(n)+h(n) \leq g(n)+h^*(n) = C^* < C_2 = f(G_2).$$

A* expands by increasing f, so n precedes G_2 — contradiction. ■

Terminate correctly

Stop when the goal is **selected for expansion**, not when first generated — otherwise the path may be suboptimal.

Condition

Optimal *iff* h is **admissible** (never overestimates the true cost).

8-Puzzle heuristics

- h_1 = number of **misplaced tiles**
- $h_2 = \sum_{\text{tiles}} |\Delta x| + |\Delta y|$ (**Manhattan distance**)

Both admissible; h_2 **dominates** h_1.

Techniques

Relaxed problems ▪ pattern databases ▪ learning

Heuristic	Avg nodes
None (UCS)	∼170,000
Misplaced h_1	∼500
Manhattan h_2	∼50

Good h: admissible, consistent, accurate, fast.

Section 4

Local Search

Explore the **neighborhood** of the current state; keep no path.

- State-centric loss functions
- Incremental refinement
- Low memory

Ideal for

8-queens, TSP, scheduling, VLSI layout, vehicle routing — the *result* matters, not the route to it.

Hill Climbing | Greedy local improvement

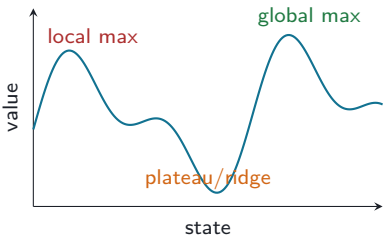
Repeatedly move to the best neighbor until none improves.

- 8-queens: move a queen to cut attacking pairs
- TSP: swap two cities to shorten the tour

Gets stuck at

local maxima ▪ plateaus ▪ ridges

Fixes: random restart, stochastic / first-choice moves.



Hill Climbing | Pseudocode

```
Algorithm HillClimb(Initial State: s, Loss function: L)
begin
  CURRENT = { s };
  repeat
    Scan neighbors of CURRENT until all scanned, or a state
      NEXT is found with lower loss than CURRENT;
    if NEXT found with lower loss than CURRENT:
      CURRENT = NEXT;
  until no improvement in previous iteration;
  return CURRENT;
end
```

Simulated Annealing | Accept some bad moves to escape optima

Inspired by metallurgical annealing: allow occasional **uphill** moves.

$$P(\text{accept}) = \begin{cases} 1 & \Delta L < 0 \\ e^{-\Delta L / T} & \Delta L \geq 0 \end{cases}$$

- High T: explore widely (accept bad moves often)
- Low T: exploit; $T \to 0$ becomes hill climbing
- Keep the **best** solution seen

Cooling schedules

- Linear: $T_0 - \alpha t$
- Exponential: $T_0 \, \gamma^t$
- Logarithmic: $T_0 / \log t$

✓ escapes local optima × needs schedule tuning

Simulated Annealing | Pseudocode

```
Algorithm SimulatedAnnealing(Initial State: s, Loss function: L)
begin
  CURRENT = { s };  BEST = { s };  t = 1;
  T = T0;                      // ~ typical loss difference between neighbors
  repeat
    NEXT = a random neighbor of CURRENT;
    dL = L(NEXT) - L(CURRENT);
    CURRENT = NEXT with probability min(1, exp(-dL / T));
    if L(CURRENT) < L(BEST): BEST = CURRENT;
    t = t + 1;  T = T0 / log(t + 1);
  until no improvement in BEST for N iterations;
  return BEST;
end
```

Tabu Search | Memory to avoid cycling

Enhance hill climbing with a **tabu list** of recently visited moves/states that are temporarily forbidden.

- **Tabu list**: fixed-size recent-move memory
- **Aspiration**: allow a tabu move if it beats the best-ever
- **Intensify** promising regions; **diversify** to new ones

Strengths

Escapes local optima deterministically; strong for TSP, scheduling, routing.

Needs tuning of the tabu-list size.

Tabu Search | Pseudocode

```
Algorithm TabuSearch(Initial State: s, Loss function: L)
begin
  CURRENT = { s };  VISIT = {};   // VISIT is the tabu list
  repeat
    Add CURRENT to VISIT;
    if all neighbors of CURRENT are in VISIT: RANDOM = null;
    else RANDOM = a random neighbor not in VISIT;
    Scan neighbors of CURRENT for a NOT-in-VISIT state NEXT
      with lower loss than CURRENT;
    if such NEXT found: CURRENT = NEXT;
    else if RANDOM != null: CURRENT = RANDOM;
  until RANDOM == null;
  return CURRENT;
end
```

Tabu Search — Practical Enhancements | Beyond the basic version

- Prefer the **best** unvisited improving neighbor, not just any improving one
- **Tabu-list size**: cap recent moves; drop the oldest entry once full
- **Aspiration criteria**: allow a tabu move if it beats the best solution found so far
- **Intensification** vs. **diversification**: focus on promising regions, or push into unexplored ones

Good fit

Combinatorial optimization: TSP, job scheduling, vehicle routing.

Local Search: Comparison

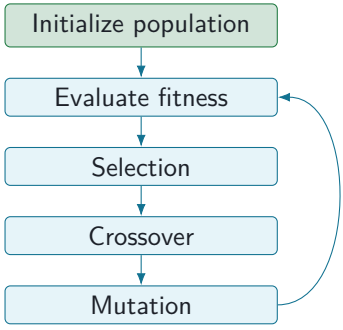
Method	Escapes optima?	Memory	Best for
Hill Climbing	No	None	Simple, quick
Random Restart	Yes	None	Many local optima
Simulated Annealing	Yes (prob.)	None	Large spaces
Tabu Search	Yes (determ.)	Tabu list	Avoiding cycles
Genetic Algorithm	Yes	Population	Complex optimization

Section 5

Genetic Algorithms

Evolve a **population** of solutions over generations.

- **Chromosome**: encoded solution
- **Fitness**: solution quality
- **Selection**: fitter parents reproduce
- **Crossover**: recombine parents
- **Mutation**: random diversity



TSP example: chromosome = city permutation; fitness = −tour length; order crossover; swap mutation.

Genetic Algorithm | Pseudocode: select, crossover, mutate, repeat

```
Algorithm GeneticAlgorithm(Initial Population: P, Loss function: L)
begin
  t = 1;
  repeat
    P = Select(P);       // fitter chromosomes survive, using L
    P = Crossover(P);     // some pairs stay unchanged
    P = Mutate(P);        // controlled by a mutation rate
    t = t + 1;
  until convergence condition is met;
  return P;
end
```

Selection Techniques | Choosing parents for the next generation

Method	How it works
Roulette-wheel	$P(\text{pick } i) = f_i / \sum_j f_j$ — each individual gets a slice proportional to its fitness
Rank selection	Sort the population by fitness; selection probability depends on **rank**, not raw value — resists domination by one outlier
Tournament	Draw k random individuals, keep the fittest; repeat to fill the mating pool
Elitism	Copy the top e individuals unchanged into the next generation — best fitness never decreases

Trade-off: strong selection pressure $\Rightarrow$ fast convergence, but risks **premature convergence** to a local optimum.

Crossover Techniques | Recombining parent chromosomes

For bit / value strings

- **Single-point**: cut at one locus, swap the tails
- **Two-point**: swap the middle segment between two cuts
- **Uniform**: each gene copied from either parent with prob. 0.5

For permutations (e.g. TSP tours)

- **Order Crossover (OX)**: copy a slice from parent 1, fill the rest in parent-2 order, skipping duplicates
- **Partially Mapped (PMX)**: swap a slice, repair conflicts via a mapping
- **Cycle Crossover (CX)**: preserves each city's absolute position via position cycles

Why not plain single-point on a tour? Naively swapping segments creates **invalid tours** with repeated / missing cities — permutation-safe operators (OX, PMX, CX) are required.

Mutation Techniques | Injecting diversity, escaping premature convergence

Method	How it works
Bit-flip	Flip a random $0 \leftrightarrow 1$ bit (binary-encoded chromosomes)
Swap	Exchange the genes at two randomly chosen positions — safe for permutations
Scramble	Randomly shuffle a contiguous sub-segment of the chromosome
Inversion	Reverse the order of a random sub-segment — a common TSP mutation

Mutation rate is usually small ($< 5\%$ per gene) — too high turns the search into random sampling; too low loses diversity and invites premature convergence.

Case Study: Genetic Algorithm for TSP | Encoding, fitness, and operator choice

Encoding

Chromosome = a permutation of city indices, e.g. $[3, 1, 4, 2, 5]$ represents the tour $3 \to 1 \to 4 \to 2 \to 5 \to 3$ (return to the start).

Fitness

fitness $= 1/$tour length — shorter tours score higher, so the fittest chromosomes are the shortest routes.

Operators used

Selection: tournament **Crossover**: order crossover (OX) **Mutation**: swap or inversion — all three respect the permutation constraint.

TSP: Order Crossover (OX) — Worked Example | Producing a valid child tour

```
Parent A: 1 2 | 3 4 5 | 6 7 8     (keep the boxed segment unchanged)
Parent B: 3 7 | 5 1 6 | 8 2 4

Child:    . . | 3 4 5 | . . .

Fill the remaining slots in Parent B's order, skipping 3, 4, 5:
  Parent B from the cut point: 8 2 4 3 7 5 1 6  ->  8 2 4 7 1 6

Child:    8 2 | 3 4 5 | 7 1 6     (every city appears exactly once)
```

A later **swap mutation** might exchange positions 1 and 6: 8 2 3 4 5 7 1 6 $\rightarrow$ 1 2 3 4 5 7 8 6

Section 6

Summary

Chapter Summary | Key takeaways

Uninformed

BFS (optimal, memory-heavy), DFS (light, not optimal), UCS (cost-optimal), bidirectional ($b^{d/2}$).

Informed

Greedy (h) is fast but not optimal; A* ($g+h$) is optimal with an admissible heuristic.

Local & evolutionary

Hill climbing, simulated annealing, tabu search, and genetic algorithms optimize the *final state*.

Looking ahead

Ch. 3: adversarial search and games (minimax, α–β).

Thank you!

Questions & Discussion

Next: Chapter 3 — Multiagent Search