Part 2

Combinatorial optimization problems involve searching through a **discrete but vast solution space** to find the best combination of decisions (maximize or minimize an objective). Classic examples include the **Travelling Salesman Problem (TSP)**, **Knapsack**, **Scheduling**, and **Boolean Satisfiability (SAT)**. For any abstract problem (problem *class*), a **problem instance** provides specific input data; solving it means finding an optimal solution for that instance. Any potential solution (even if incomplete or suboptimal) is a **candidate solution**, whereas a **solution** typically refers to an *optimal/feasible* solution that meets all problem requirements. Exhaustively exploring all combinations is often infeasible, so AI relies on clever search strategies.

Search paradigms split into two broad categories: **constructive vs. perturbative** and **systematic vs. local**. *Constructive* methods build solutions incrementally from scratch (adding components stepwise), whereas *perturbative* methods start with a complete candidate and **perturb** (modify) parts of it to improvestudocu.com. *Systematic* search (e.g. DFS, BFS, A*) explores the state space in a structured way, guaranteeing to find an optimal solution or prove none exists (i.e. **complete** algorithms)studocu.com. In contrast, *local* search begins with an initial state and then **iteratively improves** it by moving to a neighboring state based on local informationstudocu.com. Local search trades guarantees for efficiency, often finding “good” solutions faster in huge spaces but sometimes getting **stuck in local optima**.

Local search algorithms like **hill-climbing** are inspired by climbing a hill: continually take a step to a better (higher) neighboring state until no improvement is possible. They are simple and quick but prone to getting trapped on **local maxima/minima** – points that are better than nearby states but not the global optimumstudocu.comstudocu.com. Flat regions (**plateaus**) with equal neighboring values also pose problems, as the algorithm has no direction for improvementstudocu.com. Strategies like **random restarts**, backtracking, or more sophisticated heuristics are needed to escape these traps. Two important **metaheuristic** extensions address these issues: **Simulated Annealing** and **Tabu Search**. Simulated annealing behaves like a more flexible hill-climber that occasionally *accepts worse solutions* (like a downhill step) with a probability that decreases over time, analogous to slowly cooling metal so it settles into a low-energy (optimal) statestudocu.comstudocu.com. This helps it overcome local optima. Tabu search, on the other hand, augments hill-climbing with a short-term **memory of visited solutions**: it keeps a *tabu list* of recently visited states or

moves to avoid immediate cycles and encourage exploring new regionsstudocu.comalgorithmafternoon.com. Both methods are **incomplete heuristics** – they don't guarantee the absolute optimal solution, but they often find near-optimal solutions in reasonable time for hard problems.

In the realm of **game playing** (two-player adversarial games), AI uses specialized search techniques on **game trees**. A game state (e.g. board configuration) is the node, and moves generate successor states forming a tree. For **zero-sum games** (one player's gain is the other's loss), the objective is to find a strategy that maximizes one's chances of winning while minimizing the opponent'sstudocu.com. The classic algorithm is **Minimax**, which assumes both players play optimally. It recursively explores game states to a certain depth, assigning **utility values** (e.g. win = 1, draw = 0, loss = –1) at terminal states or via an evaluation heuristicstudocu.com. Players alternate layers in the search tree: at **MAX nodes** we choose the move with highest utility (best for the current player), at **MIN nodes** we choose the lowest utility (best response for the opponent)studocu.comstudocu.com. Minimax guarantees an optimal move in theory but is computationally expensive as the game tree grows exponentially with depth. **Alpha-Beta pruning** is an optimization that dramatically cuts down the number of nodes evaluated by **pruning** branches that cannot possibly influence the final decisionstudocu.commygreatlearning.com. By keeping track of two bounds – alpha (best value achievable so far for MAX) and beta (best value for MIN) – the algorithm **skips evaluating subtrees** that would only yield worse outcomes than an already found alternativestudocu.comstudocu.com. This yields the same optimal move as standard minimax but much faster, enabling deeper lookahead within the same time. Overall, Chapter 4 provides a comprehensive toolkit: from general combinatorial optimization strategies to specialized adversarial search techniques, equipping us to tackle NP-hard problems and complex games in a principled yet practical manner.

Combinatorial Problems: Definitions and Examples

Combinatorial problems are those that involve **finding an optimal object or arrangement from a finite set of possibilities**. Often these are phrased as puzzles of *"counting, arranging, or choosing items in specific ways"* under certain constraints. In **combinatorial optimization** (a key focus in AI), we have an objective function to **minimize or maximize** (e.g. total cost, distance, time, profit) and a vast space of candidate solutions. The challenge is that the number of possible combinations can grow exponentially with problem size, making brute-force search impractical. Practical importance of these problems is huge – they appear in domains ranging from scheduling and logistics to cryptography and bioinformatics.

Examples of combinatorial optimization problems include:

- **Travelling Salesman Problem (TSP)** – Find the shortest possible route that visits each city exactly once and returns to the start. (Applications: route planning, circuit board drilling)
- **Knapsack Problem** – Select a subset of items with given weights and values to maximize total value without exceeding a weight capacity. (Applications: resource allocation, budgeting)
- **Job Scheduling Problem** – Determine an optimal schedule for a set of tasks under resource and precedence constraints (e.g. minimize total completion time or lateness).
- **Facility Location Problem** – Choose locations for facilities (e.g. warehouses) to minimize cost of serving clients. (Applications: operations research, network design)
- **Boolean Satisfiability (SAT)** – Given a Boolean formula, determine if there is an assignment of true/false to variables that makes the formula true (or find such an assignment). SAT is a canonical NP-complete problem and many combinatorial problems can be formulated as SAT.

Each problem has an **abstract formulation** (problem *class*) and numerous **instances**.
The **abstract problem** defines the general question (e.g. “find the shortest tour for any set of cities”), whereas a **problem instance** provides specific input data (e.g. a particular set of 20 cities with distances). Solving the instance means producing a concrete answer (e.g. the optimal route for those cities). It’s important to distinguish the *solution of an instance* from the *solution of the abstract problem*: the latter would be an **algorithm** that can solve *any* instance of that problem.

Within any given instance, we consider **candidate solutions** and **optimal solutions**.
A *candidate solution* is any potential solution encountered during the search – it may be fully specified but not necessarily valid or optimal. For example, in TSP a candidate solution could be *any* round-trip tour (even a very long one). A *feasible* solution is a candidate that satisfies all problem constraints (in TSP, a tour that visits every city exactly once). Among feasible solutions, one (or several) will have the optimal objective value (shortest distance for TSP) – those are the **optimal solutions** for that instance. In common usage, we often say “the solution” meaning an optimal solution. The notes emphasize that candidates *might not satisfy all conditions* of the problem, especially in heuristic searches where partial or imperfect solutions are considered. A trivial example: any arrangement of cities is a candidate tour; only those that visit all cities exactly once are *valid* TSP tours, and only the shortest of those is optimal.

Why are combinatorial problems hard? The number of possible candidates typically grows combinatorially (factorial, exponential) with input size. TSP has $(n-1)!/2$ possible tours for n

cities; SAT has 2^N assignments for N variables. These huge search spaces make brute force enumeration infeasible beyond small sizes (this is the hallmark of NP-hard problems). Therefore, solving combinatorial problems efficiently requires **clever search algorithms** – exact methods (like branch-and-bound, dynamic programming) for smaller instances, and **heuristics/metaheuristics** for larger ones. The next sections introduce key search paradigms to tackle these challenges.

Search Paradigms: Perturbative vs. Constructive, Systematic vs. Local

To navigate large search spaces, AI employs different **search paradigms** that can be combined based on problem characteristics.

Perturbative vs. Constructive Search: These terms describe how candidate solutions are generated:

- **Perturbative search** starts with one or more complete candidate solutions and then **modifies (perturbs) one or more components** of a solution to get a new candidatestudocu.com. Each solution can be thought of as composed of parts (e.g. an assignment of values to variables in SAT, or an ordering of cities in TSP). By tweaking some parts (flip a variable's truth value, swap two cities in a tour, etc.), we move to a neighboring solution. Perturbative methods *directly* explore the space of full solutions. For example, a perturbative approach to TSP might begin with a random tour and repeatedly perform edge swaps to try to shorten it.
- **Constructive search** (or constructive heuristics) builds solutions gradually from scratch, typically by adding components one at a timestudocu.com. These algorithms incorporate *partial solutions* into the search space. They start from an empty or trivial state and **iteratively extend** it until a complete solution is formed. For instance, a constructive heuristic for TSP is the **Nearest Neighbor** algorithm, which starts at one city then repeatedly extends the tour by choosing the nearest unvisited city. In SAT, a constructive approach might assign truth values to variables one by one, backtracking upon conflict (as in DPLL). Constructive methods are naturally suited to building feasible solutions when it's hard to start with one. They often use greedy choices or rules to decide which component to add next.

Both approaches can be effective. Perturbative search is powerful when you have a decent initial solution to improve or when moves (perturbations) can efficiently explore the space of solutions. Constructive methods are useful when no obvious complete solution is at hand or when partial solutions can be evaluated to guide search. In practice, hybrid strategies exist

(e.g. **Iterated Local Search** uses perturbative local search on a solution, then occasionally "restarts" by constructing a new solution or heavily perturbing the current one).

Systematic vs. Local Search: This classification concerns **completeness** and **search strategy**:

- **Systematic search** algorithms explore the state space in a deliberate, **exhaustive** manner designed to find guaranteed solutions. They systematically consider states (often using a search tree) and will eventually find an optimal solution or conclude none exists, provided enough time/memory. They are **complete** (and optimal if properly designed with heuristics like A*). Classic examples: Depth-First Search, Breadth-First Search, and A* (for pathfinding). *Systematic search is characterized by backtracking and bookkeeping* – it remembers which states have been visited or pruned, ensuring every possibility is covered without infinite looping. As a result, systematic methods can prove optimality but often suffer from combinatorial explosion in big search spacesstudocu.com.
- **Local search** algorithms abandon the idea of exhaustive enumeration. Instead, they **start from an arbitrary initial state and iteratively move to a neighbor state** according to some heuristic, aiming to improve step by stepstudocu.com. Importantly, a local searcher typically **does not keep track of all visited states** or explore systematically – it focuses on the current state and its local neighborhood. This yields drastically lower memory usage and often much faster progress to a good solution. However, local search is generally *incomplete*: it might get trapped in a region and never find the global optimum, without any proof that one doesn't exist elsewhere. It cannot guarantee optimality or even finding a solution if one exists (for example, a naive hill-climber might wander endlessly if the goal is not directly "uphill" of the start). On the upside, local methods can be run indefinitely and tend to find reasonably good solutions quickly, making them ideal when approximate solutions are acceptable.

In summary, **systematic vs local** is a trade-off between **guarantee vs. efficiency**. Systematic search is like an exhaustive detective – it will find the answer eventually (if it exists) by methodically checking possibilities. Local search is like a heuristic hunt – it picks a direction that seems promising and pursues it, which is faster but may miss the optimal solution if the heuristic misleads it. Many modern algorithms blend these ideas (for instance, local search with random restarts gains a form of completeness in a probabilistic sense, and systematic search may incorporate heuristic pruning to skip unpromising branches, as in branch-and-bound).

For **combinatorial optimization**, local and metaheuristic searches (described next) are indispensable. Systematic search is often infeasible due to exponential growth, except on

smaller problem instances or using clever pruning. Practically, one might use a systematic algorithm to find optimal solutions up to a certain size, then switch to local heuristic methods for larger sizes where an exact solution is out of reach.

Local Search and Hill-Climbing

Local search is a heuristic approach that operates on the principle of **iterative improvement**. The algorithm maintains a single current state (a candidate solution) and tries to improve it by local changes. Formally, a local search algorithm has three main ingredients:

1. A **current candidate solution** (start with an initial guess, which could be random or constructed by a heuristic).
2. A definition of the **neighborhood** of a solution – i.e. the set of candidate solutions that are considered "close" to the current one by a small changestudocu.com. This depends on the problem (e.g. for TSP, neighbors might be all tours obtainable by swapping two cities in the route).
3. An **evaluation function** (or cost/loss function to minimize, or utility to maximize) that tells us how good a candidate is.

The basic **local search loop** is straightforwardstudocu.com:

- Start with an initial solution s.
- **Repeat**: look at neighbors of s (all or some). If any neighbor s' is better (lower cost/higher value) than s under the evaluation function, move to the best such neighbor (set $s := s'$).
- Stop when no neighbor improves on s (or when some other stopping criterion is met, like a time/iteration limit).

This procedure is often called **hill climbing** (for maximization; for minimization it's like descending a valley). It's akin to standing on a landscape and always stepping to the adjacent higher ground until no step is higher. The algorithm is an **anytime algorithm** – it can return the current best solution if stopped early, and it typically converges when it reaches a peak (or plateau). There are variants:

- *Simple (greedy) hill climbing:* examine neighbors one by one and take the first improvement found.
- *Steepest ascent hill climbing:* evaluate *all* neighbors and move to the best among them (the steepest upward step).
- *Stochastic hill climbing:* randomly choose among better moves, or sample neighbors at random and move if improvement occurs.
- *First-choice hill climbing:* a variant where neighbors are sampled in random order until an improvement is found (which can be more efficient if there are many neighbors).

Advantages: Hill climbing is very memory-efficient (no frontier or explored set needed as in systematic search) and can quickly find good solutions for large problems when the landscape is "nice" (e.g. smooth unimodal, or with few local optima). In fact, if the state-space landscape has a single peak (is unimodal/convex), hill climbing will find the global optimumen.wikipedia.orgen.wikipedia.org. It's commonly used in scheduling, routing, and other optimization tasks where generating and tweaking a full solution is easy.

Challenges – Local Maxima and Plateaus: The biggest issue is getting **stuck**:

- A **local maximum** (or minimum) is a solution better than all its immediate neighbors but worse than the global optimum. Once at a local maximum, hill climbing has no neighboring move that improves it, so it terminates – failing to reach the global beststudocu.com. For example, in an 8-Queens puzzle (place 8 queens with none attacking), a greedy algorithm might place queens to reduce conflicts and get stuck in a configuration with, say, 1 conflict that it can't resolve by any single move – that's a local minimum (with respect to conflict count) short of the goal.
- A **plateau** is a flat region of the state space where many neighboring states have the same valuestudocu.com. On a plateau, the algorithm has no clue which direction leads to improvement (all moves appear neutral). It might wander randomly until it finds an edge of the plateau or, in the worst case, loop indefinitely among equivalent states. A plateau could also hide a higher peak beyond it (like a mesa with another hill rising later), but a greedy climb can't ascend if all immediate steps are flat.

Another tricky terrain feature is a **ridge** – an incline that requires a sequence of moves in different directions (not simply uphill in one direction). A greedy hill-climber that only considers immediate neighbors might fail to follow a ridge if each single-step isn't directly upward.

Because of these difficulties, basic hill climbing often *fails to find the global optimum* on hard problems. For instance, in almost any non-trivial NP-hard problem like TSP or boolean SAT, local optima aboundstudocu.com. Hill-climbing on such landscapes will typically converge to a suboptimal solution unless additional strategies are used.

Techniques to overcome local optima:

- **Backtracking and memory of explored peaks:** One approach is to allow the algorithm to backtrack when stuck. For example, *hill climbing with backtracking* can remember the best states encountered and, upon hitting a dead end, return to a prior promising state and try a

different pathstudocu.com. This simulates a bit of systematic search within the local search, at the cost of more memory.

- **Random Restart:** Run the hill-climbing algorithm multiple times from different random initial statesen.wikipedia.org. The idea is that even if each run gets stuck in a local optimum, a sufficiently large number of independent runs might sample a trajectory that reaches the global optimum. This is an effective and simple strategy if evaluating solutions is fast.
- **Allow sideways moves:** Permit moves that *do not improve* the solution (or even worsen it slightly) to escape plateaus or shallow local maxima. However, too many sideways moves can lead to wandering. Some implementations allow a fixed number of sideways steps before giving up.
- **Heuristic modifications:** For example, modify the evaluation function dynamically to encourage exploration (known as "simulated annealing" approach, described next) or employ techniques like **tabu lists** to avoid cycling (described later).

Despite its shortcomings, hill climbing is often a component of more sophisticated algorithms. Its speed and simplicity make it a good baseline or a subroutine once other mechanisms (like randomization or memory) handle the pitfalls. In fact, *iterative improvement* via local changes is at the heart of many **metaheuristics** (simulated annealing, tabu search, evolutionary algorithms' mutation operators, etc.). Next, we discuss two such metaheuristics that enhance local search to better navigate complex landscapes.

Simulated Annealing

Simulated annealing (SA) is a **probabilistic local search** algorithm that addresses the problem of local optima by occasionally allowing **downhill moves** (worse solutions) in a controlled way. It is inspired by the physical process of **annealing in metallurgy**, where a metal is heated and then slowly cooled to remove defects, reaching a low-energy crystalline structurestudocu.com. By analogy, SA "heats" the search process to allow exploration (including worsening moves) and then "cools" it over time to focus and settle on a good solution.

Key idea: *"Always move uphill if possible; sometimes move downhill to escape traps, with a probability that decreases over time."* In hill-climbing terms, it's like a climber who occasionally takes a step in a bad direction in hope of finding a higher hill later, and as time goes on, becomes more conservative about taking such risks.

Simulated annealing augments the local search loop with a **temperature** parameter T that controls the randomness:

- Start with an initial solution s (random or constructed) and an initial temperature T_0.
- **Repeat (iteration $t = 1, 2, \ldots$):**
  1. Pick a random neighbor s' of the current solution $s$studocu.com.
  2. **Evaluate** the change in cost: $\Delta = \text{Cost}(s') - \text{Cost}(s)$ (for a minimization problem; for maximization Δ would be the negative change in utility).
  3. If $\Delta < 0$ (the neighbor is better), accept s' as the new current solution ($s := s'$).
  4. If $\Delta \geq 0$ (the neighbor is worse or equal), *maybe* accept it with probability $P(\text{accept}) = \exp(-\Delta/T)$studocu.com. This is the Metropolis criterion: even a worse move can be accepted, especially if Δ is small or T is high (notice $\exp(-\Delta/T)$ is close to 1 for small Δ or large T, and close to 0 for large Δ or low T).
  5. Update the temperature T according to a **cooling schedule** as t increases (e.g. $T := \alpha T$ for some $0 < \alpha < 1$ each iteration, or a slower schedule like $T = T_0 / \log(1 + t)$ studocu.com).
  6. Optional: keep track of the **best solution found so far** in a separate variable, because SA, by design, can wander uphill and might end at a state worse than a previously seen onestudocu.com. At the end, you return the best seen.

Initially, at high temperature, the algorithm is almost purely random – it will accept even significantly worse moves with decent probability. This "high-energy" exploration helps it roam widely and jump out of local basins. As T lowers, the acceptance of bad moves becomes rarer (the algorithm behaves more like greedy hill-climbing when cold). Ideally, if cooled slowly enough, the search will **converge to a near-optimal (or optimal) solution** once the "thermal energy" is too low to accept anything but improvementsstudocu.comstudocu.com.

Analogy (Physics): Think of a ball bouncing on a hilly landscape. If you shake the landscape violently (high T), the ball can jump out of local dips. As you gradually reduce the shaking, the ball settles into a deep valley. If cooled infinitely slowly, it's theoretically guaranteed to reach a global minimum. In practice, we use a finite cooling schedule, so it's not a guarantee but greatly improved odds of escaping local minima compared to plain hill-climbing.

Cooling Schedule: The choice of how T decreases over time is crucial. Common schedules include:

- **Exponential decay:** $T := \alpha T$ per iteration or per fixed number of moves (with α slightly below 1, e.g. 0.99). Simple but needs tuning of α and initial T_0.
- **Linear decay:** $T := T_0 - \beta t$ (decrease by a fixed amount each step). Usually not ideal for long runs because T might become zero too fast.

- **Logarithmic or slower decay:** e.g. $T = \frac{T_0}{\log(t+1)}$ or $T_0/(1+\beta\ln(1+t))$studocu.com. These decay more slowly, ensuring a long "tail" of low but nonzero temperature for fine-tuning.
- **Adaptive or automated schedules:** Adjust T based on the search progress (e.g. lower T when the system appears to be near equilibrium or if acceptance rate of worse moves falls below some threshold).

A well-chosen schedule balances **exploration vs. exploitation**: too rapid cooling can freeze the search in a suboptimal state (like quenching metal too fast leads to defects), whereas too slow cooling wastes time exploring unnecessarily. In practice, much experimentation or even self-tuning is used to get good results.

Advantages: Simulated annealing is conceptually simple yet surprisingly powerful. It can find excellent solutions to very hard problems given enough time. It's one of the few algorithms with a theoretical guarantee (if cooled infinitely slowly, it converges in probability to a global optimum). SA's stochastic nature often finds better solutions than deterministic greedy search, and it's robust to getting out of small local minima. It's broadly applicable – you only need a way to generate neighbors and a way to evaluate cost.

Disadvantages: The performance **hinges on tuning parameters** – initial temperature, cooling rate, iteration count, etc. If T_0 is too low, the algorithm behaves like greedy hill-climbing and may immediately get stuckstudocu.com. If it cools too fast, it "freezes" prematurely (also getting stuck). If it stays hot too long, it wastes time random-walking. Finding the right balance often requires trial and error or domain knowledgestudocu.com. SA can also be **computationally expensive** for large problems, since it might require many iterations to slowly anneal and thoroughly explorestudocu.com. Each iteration involves evaluating neighbors, which could be costly. Moreover, like any heuristic, it doesn't *guarantee* the optimal solution – it might return a good solution that's not the absolute best (though in practice it's often near-optimal). In summary, simulated annealing trades certainty for flexibility: it's not foolproof, but it significantly **reduces the chance of getting trapped in poor solutions** compared to naive local searchstudocu.comstudocu.com.

Insight: An interesting perspective is to view SA as injecting "controlled chaos" into the search. Early on, the algorithm behaves almost randomly (high entropy), which is analogous to melting the constraints of the problem – it's willing to try outrageous solutions. As it cools, structure and order emerge (like a crystal forming), and it becomes more conservative, honing in on minima. This process has parallels in **thermodynamics** and even in **biology** (e.g. biological systems or

evolutionary algorithms occasionally introduce random mutations – a high-temperature phase – followed by selection and stabilization).

Exam Tip (application): If asked to **design a simulated annealing algorithm** for a specific problem, outline these components: (1) representation of a state (solution), (2) method to generate random neighbor states (small mutation of current solution), (3) an objective (or loss) function to evaluate quality, (4) a schedule for temperature: initial T_0 and how it decreases, (5) stopping criterion (e.g. no improvement in X iterations or minimum temperature reached). For example, *"To apply SA to the knapsack problem, we could represent a state as a binary vector of item selections. A neighbor could be generated by flipping the inclusion/exclusion of one item (or swapping an included and excluded item). The cost function might be negative profit plus a heavy penalty for violating the weight limit (to handle infeasible picks). We'd start at a high temperature T_0 that allows occasional acceptance of lower-profit (or overweight) solutions, and gradually reduce T. This would let the search sometimes take out high-value heavy items (a seemingly bad move) to explore if it allows inclusion of multiple lighter items yielding better total value later."* By giving such an outline, you demonstrate understanding of how SA components map to a concrete problem.

Tabu Search

Tabu Search is a **metaheuristic that extends local search with memory**. It was developed by Fred Glover (1986) and has proven very effective for many combinatorial problems. The core idea is to allow local search to continue beyond local optima by **forbidding or penalizing moves that would immediately undo recent progress**. In simple terms, it's hill-climbing that can go downhill, but it avoids cycling by keeping a short-term memory of recently visited solutions or moves – those are marked "taboo" (tabu) for a certain number of stepsstudocu.com.

Mechanism: Tabu search uses a **tabu list** to store either:

- Recent solutions visited, or
- Attributes of recent moves (e.g. recently swapped elements in a solution).

If a new candidate is on the tabu list (meaning it was visited too recently or it results from reversing a recent move), the algorithm will generally **forbid** moving to that candidate, even if it looks better, to prevent fruitless back-and-forth oscillationstudocu.com. This enables the search to **"climb out" of local optima** by taking a sequence of moves that may temporarily worsen the solution, without immediately undoing those moves.

A basic **Tabu Search algorithm**:

1. Start with an initial solution s (and maybe an initial best solution $s_{\text{best}} = s$).
2. Initialize an empty tabu list L with a fixed maximum size (tabu tenure z = number of recent states or moves to remember)studocu.com.
3. **Repeat** until stopping condition:
   - Generate one or more neighbor solutions $N(s)$ of the current solution s.
   - Among the neighbors, pick the **best** neighbor s' that is *not tabu* (not in L) **or** that satisfies an aspiration criterion (explained shortly).
   - Move to s' (set $s := s'$).
   - Update s_{best} if s' is better than the best-found so far.
   - Update the tabu list: record the move or solution just made to L. If L exceeds size z, drop the oldest entry (FIFO queue discipline)studocu.com.
4. End when some criterion met (e.g. a time/iteration limit or no improvement for a while). Return s_{best} found.

Unlike simple hill climbing, **tabu search will take the best available neighbor even if it's worse than the current solution**, as long as it's not tabu. This means it doesn't always improve each step – it performs a kind of guided walk through the solution space, allowing occasional degradation to escape local maximastudocu.com. The tabu list is what guides it away from immediate cycles or backtracking into the local optimum it just left. Essentially, the algorithm says: *"I know that move looked attractive 2 steps ago, but I'm forbidding it for now so that we explore something new."* This yields a broader search of the space than naive local search.

Tabu list details: The list can store either full solutions or specific moves/attributes. Storing full solutions could consume a lot of memory, so often we store something like "the move that swapped city X and Y in the TSP tour" or "the variable x_i was flipped from 0 to 1". A move can be declared tabu for the next z iterations (tabu tenure). After z new moves, it's removed and becomes allowed again. The tenure z is a key parameter: too short and you may revisit cycles; too long and you may forbid too many states, possibly excluding good regions (**diversification** vs **intensification** trade-off). Sometimes z is varied dynamically during the search (adaptive tabu).

Aspiration Criterion: This is an important concept – a rule that allows overriding the tabu status of a move if certain conditions are met. A common criterion is: *if a tabu move leads to a*

*solution better than the best seen so far, then ignore the tabu and accept the move*algorithmafternoon.com. This prevents the tabu list from disallowing genuinely excellent solutions. Essentially, “great improvements are always allowed,” taboo or not. Other aspiration criteria could be based on reaching a certain objective threshold, etc.

Variations and Advanced Features: Tabu search is more of a framework; many refinements existalgorithmafternoon.com:

- **Intensification**: periodically focus the search around high-quality solutions (e.g. restart from the best-known solution or reduce tabu tenure to fine-tune around it)algorithmafternoon.com.
- **Diversification**: if the search stagnates or keeps cycling through a region, force it to explore a different area (e.g. jump to a radically different solution or increase tabu tenure to escape patterns)algorithmafternoon.com.
- **Long-term memories**: aside from the short-term tabu list, some implementations keep frequency counts of how often certain attributes (like particular elements being in the solution) appear, and use that to guide occasional diversification moves (e.g. penalize overused components to encourage something new).

Advantages: Tabu search is powerful at **exploring a search space without getting stuck** in one region. By preventing immediate reversal, it encourages the algorithm to “try something different” when at a local optimum, often leading to discovering better solutions after a few detours. It can escape local minima more reliably than simulated annealing in many cases because it systematically avoids turning back. Tabu search has been successfully applied to very large scheduling, routing, and assignment problems, often producing state-of-the-art results. It’s also quite **flexible** – easy to tailor the neighborhood moves and tabu conditions to specific problems, and it can incorporate domain-specific heuristics. Another benefit: it remembers a small part of the search history (the tabu list), which is a form of exploiting past knowledge to influence future search (a bit like how humans avoid repeating recent mistakes)algorithmafternoon.com.

Disadvantages: The performance depends on various parameters that may require tuning: the length of the tabu list, the neighborhood selection strategy, and any aspiration criteriaalgorithmafternoon.com. If the tabu list is too short, the search might still cycle; too long, and it may restrict the search excessively or reject many feasible moves, possibly missing good solutions. Tabu search can also be computationally heavy: in each iteration, if one evaluates many neighbors to pick the best non-tabu move, that could be costly (though often you use heuristics to limit neighbors). It’s still a heuristic – no guarantee of optimality and it may require

many iterations to converge to a high-quality solutionalgorithmafternoon.com. In highly constrained problems, designing an effective neighborhood that yields feasible solutions can be tricky, and tabu search might struggle if almost every neighbor is infeasible (though one can also tabu *infeasible moves* to search among feasible space).

Analogy and Perspective: Tabu search introduces a notion of **short-term memory** to local search, analogous to how a chess player might remember the last few moves and avoid repeating a sequence that didn't work. Biologically, one could compare it to an animal foraging: it avoids the exact spots it recently checked (tabu list), thus forcing itself to try new locations for food. In a maze, tabu search is like leaving breadcrumbs on recent wrong turns so you don't immediately go back and forth – you explore further down other paths.

Exam Strategy: When explaining tabu search, it's good to mention the tabu list concept clearly and possibly illustrate with a small example. For instance, *"consider a 8-puzzle (sliding tile) solved by tabu search: the algorithm might move tiles around and reach a configuration that is nearly solved but stuck. A purely greedy approach might oscillate two tiles back and forth; tabu search would mark that swap as tabu, so it tries a different swap leading to a messier board but from which eventually a new path to solution emerges."* Also, be prepared to discuss choices of tabu tenure and aspiration: *"a tabu tenure of 7 means once a move is made, it cannot be undone for the next 7 movesalgorithmafternoon.com. If a move produces a board configuration better than any seen before, an aspiration criterion could allow it even if ordinarily tabu."* These specifics show a deeper understanding.

Game Playing: States, Game Trees, and Utility in Adversarial Search

Game playing in AI deals with making optimal decisions in competitive environments. We focus on **two-player, zero-sum games**, like chess, checkers, tic-tac-toe, etc., where one player's win is the other's lossstudocu.com. Key concepts include how to represent game **states**, how to evaluate outcomes (utility values), and how to efficiently explore the **game tree** of possible moves.

State Representation: A game state must contain all information needed to decide the next moves and determine if the game is over. For board games, a state typically includes the positions of all pieces and whose turn it is. For example, in chess a state could be represented by an 8×8 matrix of pieces plus a flag for the side to move, plus contextual info like castling

rights (for full rules). In tic-tac-toe, a 3×3 grid with X's and O's and the next player to move suffices. The representation should allow generating all **legal moves** from that state.

Game Tree: Starting from the initial state (root), each possible move by the current player leads to a new state (child node). Then the opponent's possible responses lead to further states, and so on, forming a tree of alternating moves. A path down the tree corresponds to a sequence of moves by both players. **Terminal states** are end-of-game positions (win, loss, or draw). In a game tree, levels alternate between players: we often label them MAX (the player whose turn it is from the root perspective) and MIN (the opponent) alternately at each plystudocu.com. Because both players aim to win, their objectives conflict – this is captured by the zero-sum assumption (one's utility = – (other's utility)).

Utility Values: We assign a numeric value to terminal states to indicate the outcome from the perspective of the MAX player. A win for MAX might be +1 (or +∞ for an outright win in theory), a loss –1 (or –∞), and a draw 0. These are **utility values** or payoffsstudocu.com. For non-terminal states that we can't fully analyze (because the game is complex), we use an **evaluation function** (heuristic) to estimate the utility – e.g. in chess, a common heuristic is material balance (e.g. assign points to pieces and sum up difference)studocu.com. This gives a score representing how favorable the position is for MAX. The aim is for MAX to maximize this utility and MIN to minimize it.

Zero-Sum and Adversarial Assumption: "Zero-sum" means we can think in terms of a single utility value from one player's perspective; whatever one gains, the other loses equally. We assume *optimal play* – each player is rational and will try to maximize their own advantage (MAX maximizes utility, MIN minimizes it)studocu.com. This is a strong assumption (real human opponents may make suboptimal moves), but it provides a solid baseline strategy: if you plan for the worst the opponent can do, you guarantee at least a draw in games that have a forced draw with perfect play.

Game Tree Complexity: The number of states grows exponentially with the number of moves (the **branching factor** is the average number of legal moves per state, and game length ~ depth). For example, in chess the branching factor is about 35 and games can easily go 40 moves each side, so naive game trees have 35^{80} nodes (astronomical). Even tic-tac-toe has a manageable tree (branching up to 9, depth up to 9, so about $9! = 362,880$ leaf nodes). If a game's state space is **small enough**, we can *exhaustively search* the entire tree and assign

values to all possible end states (e.g. tic-tac-toe is easily solved, connect-4 has been solved, checkers was weakly solved, chess and Go are not feasible to fully solve with brute force)studocu.comstudocu.com. When full enumeration is impossible, AI uses **heuristic search**: look ahead a limited number of moves and use evaluation functions at the frontier (these are essentially informed guesses of utility). This introduces uncertainties (the horizon effect, described later), but with good heuristics and techniques like pruning and iterative deepening, game-playing AIs perform remarkably well.

The rest of this section delves into the two key algorithms: **Minimax** for choosing optimal moves assuming perfect play, and **Alpha-Beta pruning** to make this search efficient.

Minimax Search Algorithm

Minimax is the fundamental algorithm for optimal decision-making in zero-sum games. It embodies the principle: *"Maximize my minimum payoff"*, meaning MAX will choose a move that maximizes the worst-case outcome assuming the opponent (MIN) responds in the most punishing way. Equivalently, it can be seen as *MIN will minimize the maximum payoff MAX could get from any move*. This strategy results in the **Nash equilibrium** for deterministic zero-sum games – it's optimal if both play optimally.

Assumptions: Both players are perfectly rational and play optimally to win (or force a draw)studocu.com. They have common knowledge of the game rules and each other's optimality. Under these assumptions, minimax will find the move that guarantees the best possible outcome for MAX. If the opponent deviates (is suboptimal), MAX's choice will still be fine (it might even win more easily).

Algorithm Overview: Minimax performs a **depth-first search** of the game tree, exploring possible moves and countermoves, and uses recursion to determine the value of each state:

- At a **MAX node** (our turn), we take the maximum value of its children (because we can choose the move that leads to the state with the highest utility for us).
- At a **MIN node** (opponent's turn), we take the minimum value of its children (because the opponent will choose the move that is worst for us, i.e. minimizes our utility)studocu.com.
- **Terminal nodes** (game over or depth limit reached) have a utility given by the game outcome or an evaluation function.

In pseudo-code form, a simple minimax could be written as a function `Minimax(state, depth, player)` :

plaintext

Copy

```
function Minimax(state, depth, player): if state is terminal or depth == 0: return utility(state) // evaluate terminal or at cutoff if player == MAX: bestVal = -∞ for each action in actions(state): val = Minimax(result(state, action), depth-1, MIN) bestVal = max(bestVal, val) return bestVal else (player == MIN): bestVal = +∞ for each action in actions(state): val = Minimax(result(state, action), depth-1, MAX) bestVal = min(bestVal, val) return bestVal
```

The initial call is `Minimax(root_state, d, MAX)` for some search depth d. The result is the minimax value of the root. To pick a move, we would choose the action leading to the child state that has that minimax value.

In the notes, the algorithm is described in steps: (1) If at a leaf (or depth limit), return its static value; (2) If at a MIN node, recursively get values of children and return **min**; (3) If at a MAX node, return **max**studocu.com. This repeats until the whole relevant game tree (to the given depth or to end) is traversed.

Interpreting the results: The minimax value of the root (assuming perfect play) tells you how good the position is for MAX. For instance, in tic-tac-toe from the starting empty board, the minimax value is 0 (a forced draw with optimal play). In a won position, the minimax value might be +1 (win for MAX in a finite number of moves). The actual algorithm will also typically keep track of *which move* achieves that value (through something called the “principle variation”).

Properties: If the game tree is fully searched to terminal states, minimax will find a *provably optimal strategy*. In finite deterministic games with no element of chance, if both players use minimax decisions, they will reach an equilibrium outcome (which could be win, loss, or draw for MAX depending on the game). Minimax search always finds the optimal path in a finite tree given the assumptionsstudocu.comstudocu.com. However, as the branching factor or depth grows, the number of nodes expands exponentially (complexity $O(b^d)$, where b is branching factor and d depth). This is why in practice we use depth limits and heuristics rather than search the entire game (except trivial games).

Example: Consider a simple game tree: MAX at root can choose moves leading to states A or B; then MIN responds. If the leaf utilities (from MAX's view) are, say:

- After A: MIN can force either 3 or 5 (so MIN will choose 3, worst for MAX).
- After B: MIN can force either 2 or 4 (so will choose 2).
  Minimax will assign value 3 to A (because MIN gives 3) and 2 to B. MAX will choose A with value 3, as that's the higher outcome. Thus, MAX's minimax decision is move A yielding a payoff of 3. This example shows how MAX foresees MIN's best counter (taking 3 instead of allowing 5)studocu.com.

Strategic Insight: Minimax embodies a **conservative strategy** – it secures the best guaranteed outcome. It can be seen as each player assuming the opponent will exploit any mistake, so it never leaves an opening that would lead to a disastrous outcome if avoidable. This can lead to strategies that might seem counter-intuitive to humans (who might take calculated risks hoping the opponent errs). For AI, minimax provides a baseline of *perfect play*. In practice, real players are not perfect; nonetheless, minimax (with depth limits) combined with good heuristics has historically done well even against humans (e.g. in chess engines). One should be aware of the **horizon effect**: if the depth cutoff is such that a catastrophic outcome lies just beyond the search horizon, the algorithm might misjudge a move as safe. For instance, a chess engine might not see a sequence of moves leading to inevitable loss if it's beyond its depth d, so it might make a move that looks good within d moves but actually is losing after $d+2$ moves. This is called the *horizon effect* – the algorithm fails to see beyond the "horizon" and is blindsided by a trap or a long-term consequencestudocu.comstudocu.com. Exam questions might ask about this as a limitation of minimax with fixed depth.

In summary, **minimax provides the formal foundation for adversarial search**, guaranteeing an optimal play strategy in theory. However, due to its exponential complexity, we usually employ enhancements like alpha-beta pruning and heuristic evaluations for practicality.

Alpha-Beta Pruning

Alpha-Beta pruning is an optimization technique for the minimax algorithm that **cuts off unnecessary branches** in the game tree, without affecting the final result. It greatly reduces the number of nodes evaluated, often enabling the algorithm to search much deeper in the same amount of timemygreatlearning.com. The basic idea: if you have found a guarantee that one move is at least as good as X for MAX, and you find another move that leads to a situation that is worse than X for MAX (assuming optimal opponent play), then you can stop exploring that second move's further consequences because MAX would never choose that move.

Alpha-beta introduces two values, **α (alpha)** and **β (beta)**, which represent bounds on potential outcomes during searchstudocu.com:

- **α (alpha)**: the *best (highest) value found so far for MAX* along the current path. In other words, the minimum score MAX is assured of given the moves already explored. Initially, α = –∞ (no assurance yet). Only MAX nodes update α (taking a max).
- **β (beta)**: the *best (lowest) value found so far for MIN* along the current path (or equivalently, the lowest value that MIN can force given current knowledge). Initially, β = +∞. Only MIN nodes update β (taking a min).

As the search goes depth-first, these α and β bounds get passed down the tree and updated. Whenever the algorithm finds **α ≥ β**, it triggers a **prune**: this means the current node (and its subtree) cannot produce a better outcome than what has already been found on an alternative branch, so further exploration is pointlessstudocu.com.

To illustrate the logic:

- **Alpha-cut (prune)**: Occurs at a MIN node when it finds a value ≤ some α value that MAX can get elsewhere. For example, say at a MAX node, we have an option A that leads to value 5 for MAX. Now we're evaluating another option B; during exploring B's subtree (MIN's turn), we find a child state that gives a value (for MAX) of, say, 3 for B. The moment MIN sees a value that is ≤5, MIN knows MAX already has a guaranteed 5 via A, so MAX would never allow B to be chosen if all outcomes of B are ≤5. Thus, MIN (trying to minimize) will ensure B yields at most 3, and MAX won't pick B because A yields 5 > 3. Therefore, **we can stop evaluating other children of that MIN node** because MIN, being optimal, would choose the worst for MAX anyway, and we already see it's worse than an alternative (A) MAX hasstudocu.comstudocu.com. We cut off B's remaining unexplored moves — that's an alpha-cut.
- **Beta-cut (prune)**: Occurs at a MAX node similarly. Suppose at a MIN node we have an option that leads to value 7 (bad for MIN, good for MAX). Now evaluating another sibling (still under the same MIN parent) via a MAX node, if MAX finds a move that yields value, say, 9 (for MAX) in that subtree, then at the MIN parent, β (the best for MIN so far) becomes 7 (because MIN can enforce 7 from the first child). When a different child branch starts yielding values ≥7, actually this scenario is inverted (I must be careful with example): Typically, a beta-cut occurs at a MAX node when α gets high enough. For instance, at some MAX node, if it's found a value ≥ β (which is a value MIN can assure elsewhere), then MAX's branch is going beyond what MIN will allow. Concretely, if at a MIN node, one child returns value 3 (so β=3 for that MIN), and we are exploring another child which is a MAX

node. If that MAX node finds a move with value 5 (≥ β=3), then MIN, having β=3 as an alternative, will never let MAX actually get 5 because MIN would choose the branch giving 3 instead. So as soon as the MAX child's value ≥ 3, we know this branch (child of MIN) won't be chosen by MIN (because MIN has a better way to hold MAX to 3). Therefore, further moves in this MAX branch can be pruned – that's a beta-cut.

In code terms, alpha-beta is usually implemented in the minimax recursion:

plaintext

Copy

```
function AlphaBeta(node, α, β, maximizingPlayer): if node is terminal: return utility(node) if maximizingPlayer: for each child of node: α = max(α, AlphaBeta(child, α, β, false)) if α ≥ β: break // β cut return α else: for each child of node: β = min(β, AlphaBeta(child, α, β, true)) if α ≥ β: break // α cut return β
```

At a MAX node, we prune when we find a value ≥ β (opponent already has an option for a lower value, so opponent won't allow this branch). At a MIN node, prune when value ≤ α (MAX has a higher alternative already, so MAX wouldn't go down this branch).

Result: Alpha-beta yields the exact same minimax value and optimal moves as a full minimax search, but often examines far fewer nodesgeeksforgeeks.org. In the *best-case scenario* (with optimal move ordering – meaning you always explore the best moves first), alpha-beta can reduce the complexity from $O(b^d)$ to about $O(b^{d/2})$mygreatlearning.com. This effectively doubles the depth you can search for the same effort – a huge gain in games like chess. For example, instead of exploring 100 million nodes, maybe you only explore 10 thousand with perfect pruning (extreme hypothetical). *Worst-case*, if the moves are encountered in a particularly bad order (always exploring the worst moves first), alpha-beta does no pruning at all and ends up examining b^d nodes just like minimax. But with even moderate move ordering (e.g. using heuristic suggestions or ordering captures first in chess), you get substantial cuts.

Pruning Example: The notes describe a scenariostudocu.com: Suppose at node A (MAX) we have two children B and C (both MIN nodes). While exploring B's subtree, we found B's minimax value = 3 (after checking B's children which yielded 3 and 5, MIN would choose 3)studocu.com. This sets α (at A) to 3 (since A as MAX currently can guarantee at least 3 via B). Now we explore C. Imagine C's first child (call it F, a MAX node's result at C's level) comes out with value -5 for MAX (i.e. very bad for MAX). This sets β (at C) to -5 (C can hold MAX to -5 so

far). Now before exploring C's other child (say G), notice A's α = 3 and C's β = -5. At A (MAX), we see that C (MIN) can at most give us -5 (if that held); but A already has α=3 from B. Since α (3) ≥ β (-5) in the context of considering C, we realize MAX would never choose C because B offers 3 which is > -5. Therefore, **we do not need to explore G at all**studocu.comstudocu.com. This was an alpha-cut from the perspective of A's search (pruning C's remaining branch). In general, these cuts save time by not exploring moves that rational players would avoid.

Alpha-beta does not require additional memory beyond some variables, and it's usually implemented via simple modifications to the DFS recursion of minimax. Modern game engines incorporate even more pruning techniques (like null-move heuristic, transposition tables to avoid re-searching identical states, etc.), but alpha-beta is the cornerstone.

Insight: One way to think of alpha-beta is as an analogy to **branch-and-bound** in optimization – if you already found a pretty good solution (bound) and you're exploring another branch that can't possibly beat that bound, you cut it off early. In game trees, because two players have opposite goals, the bound logic is two-sided (alpha for one side, beta for the other).

A tip for understanding: α is the highest guarantee for MAX so far (Max's floor) and β is the lowest guarantee for MIN (Max's ceiling, since Min tries to push the value down). When α meets or exceeds β, it means the "ceiling" and "floor" have crossed – the range of possible values narrows to nothing, so further search is futile.

Why doesn't pruning affect the result? Because whenever we prune, we *know* that branch would never influence the final decision given optimal play. We are effectively relying on logical dominance: "Given what's already explored, this branch is suboptimal no matter what happens down there." Thus, we never discard potentially optimal moves – only ones proven to be worse than an already found alternative.

Exam Pointers: Be prepared to perform alpha-beta by hand on a small game tree. Typically, you'd be given a tree with leaf values and asked to show the order of evaluation and where cuts occur, and what nodes are pruned (often marking an "X" on branches that get pruned). Remember: maximize updates α, minimize updates β, and check for α ≥ β to prune. One non-obvious detail: the **order of exploring children** matters a lot – if you explore in the optimal

order, you prune more. On an exam, if not specified, you can assume some order is given or choose a logical one (like left-to-right) and do alpha-beta accordingly.

For instance, one might present a tree and ask: “Using alpha-beta, which nodes are not evaluated?” You’d simulate the search:

- Traverse down leftmost branch first, accumulate α/β, then use those to cut off other branches when possible.
- Mark the cutoff when you hit α ≥ β.

Alpha-beta essentially **optimizes minimax to practical usability**, enabling modern AI to search many moves ahead in complex games. With alpha-beta, chess programs routinely search 10-12 plies (half-moves) or much deeper with enhancements, which would be impossible with raw minimax.

Exam Wildcards and Non-Obvious Details

- **Abstract Problem vs. Instance vs. Solution:** Remember that *solving an abstract problem* means providing an algorithm or general method. A *problem instance* is a specific dataset, and a *solution* to that instance is a specific optimal result. Students often confuse “the solution” as any feasible answer versus the optimal answer – in optimization context, *solution* usually implies **optimal** solution, while any feasible answer is a **candidate solution**. Clarify this in answers.
- **Candidate Solutions vs. Optimality:** A candidate may violate constraints or just be suboptimal. Always distinguish feasible candidates from the optimal solution. In proofs or algorithm design, mention if your method examines partial candidates (like constructive search does) or only full solutions (perturbative).
- **Local Search Completeness:** Emphasize that **local search is incomplete** – it can fail even if solutions exist. It’s a heuristic approach. In contrast, systematic search (with heuristics like A*) is complete (given finite space and admissible heuristics). This distinction is a common exam point.
- **Local Maxima, Plateaus, Ridges:** These specific failure modes of hill climbing are often asked. Students should not just say “hill climbing can get stuck” but identify *how*: a local maximum (no better neighbors)studocu.com, a plateau (flat neighbors)studocu.com, or a ridge (a narrow path of improvements that greedy moves might miss). Recognizing these in given scenarios (e.g. “why did the algorithm stall at this state?”) is key.
- **Sideways Moves & Random Restarts:** These simple modifications to hill climbing are easy to state but often overlooked. Mention that allowing some sideways or downward moves is essentially what simulated annealing formalizes.

- **Simulated Annealing Parameters:** The performance of SA **heavily depends on the cooling schedule and initial temperature**studocu.com. On exams, if asked "why might SA fail to find the optimal?", a top answer is "cooled too quickly or started at too low temperature → search got stuck in local optimum" or "inadequate iterations at low temperature to refine solution". Also note SA is an example of an **anytime algorithm** – you can stop it and have a solution (best seen so far) at any time.
- **Tabu Tenure Choice:** In Tabu Search, the length of the tabu list is criticalalgorithmafternoon.com. A non-obvious detail: sometimes the tabu list stores **moves** instead of states – e.g. in TSP, store recently swapped city pairs so you don't swap them back immediately. Also, **aspiration criteria** are a nuance: a tabu move can be allowed if it yields a solution better than any seen so far, ensuring tabu search doesn't miss a global optimum just because it was once visited or involves a recent move.
- **Strategic Oscillation (Tabu):** Advanced tabu search uses the idea of oscillating around feasibility or around optima by allowing temporary rule violations or going back and forth across a boundary (this might be beyond our scope, but worth noting if known).
- **Game State Explosion:** Always keep in mind the **branching factor** when discussing game tree algorithms. Many answers lose points by not acknowledging why we need pruning. Minimax without pruning is $O(b^d)$ which is infeasible for large b, d. Alpha-beta doesn't change worst-case complexity but *on average* drastically prunes if move ordering is reasonably good.
- **Horizon Effect:** This is the phenomenon in depth-limited search where the algorithm cannot see beyond a certain depth, so it might make a move that appears good short-term but is actually bad (because a disaster lies just beyond the horizon)studocu.com. Good to mention when talking about limitations of minimax.
- **Alpha-Beta Pruning Doesn't Affect Outcome:** Some students mistakenly think alpha-beta is an approximation. Make it clear: alpha-beta yields exactly the same decision as plain minimaxgeeksforgeeks.org. It's just more efficient. It prunes *only* branches that cannot affect the final choice. So one should never say "alpha-beta finds a nearly optimal move" – it finds the optimal move, just faster.
- **Move Ordering for Alpha-Beta:** The effectiveness of pruning depends on examining the best moves first. In an exam, they might ask "how can you maximize alpha-beta cut-offs?" Answer: *"By exploring the most promising moves first (for Max, those likely with higher heuristic value; for Min, lower). In practice, one might sort moves by an evaluation heuristic before searching."* If ordering is worst-case (always choose the poorest move first), alpha-beta does no pruning.
- **Minimax in Non-Zero-Sum or Stochastic Games:** A subtle point – minimax applies to deterministic, zero-sum games. If a game is not zero-sum (both can gain, or there are different preferences), then minimax is not directly applicable – you'd need more general game theory (like Nash equilibrium search). If the game has randomness (dice, card

draws), you need an extension like **expectiminimax** (with chance nodes). These might be outside COS314's scope, but it's good to recall boundaries of applicability.

- **Utility Function Design:** When you can't search to end, a good evaluation (utility) function is crucial. A wild card detail: in designing a game AI, often the evaluation function is weighted sum of features (in chess, material, mobility, king safety, etc.). It's not purely search – knowledge matters.
- **Cutting Off Search & Iterative Deepening:** Practical game engines use **iterative deepening** (depth 1,2,... etc. until time runs out) and alpha-beta. Mention that if time-limited, they return the best found at whatever depth was completed – another example of an anytime algorithm.
- **Metaheuristic vs. Systematic Approaches:** Sometimes exam questions ask "why use a metaheuristic like SA/Tabu instead of an exact algorithm?" Key points: metaheuristics find good solutions in reasonable time for NP-hard problems, whereas exact algorithms might take exponential time. However, metaheuristics do not guarantee optimality or even finding a feasible solution in some cases. They are typically used when *approximate solutions* are acceptable due to time constraints.