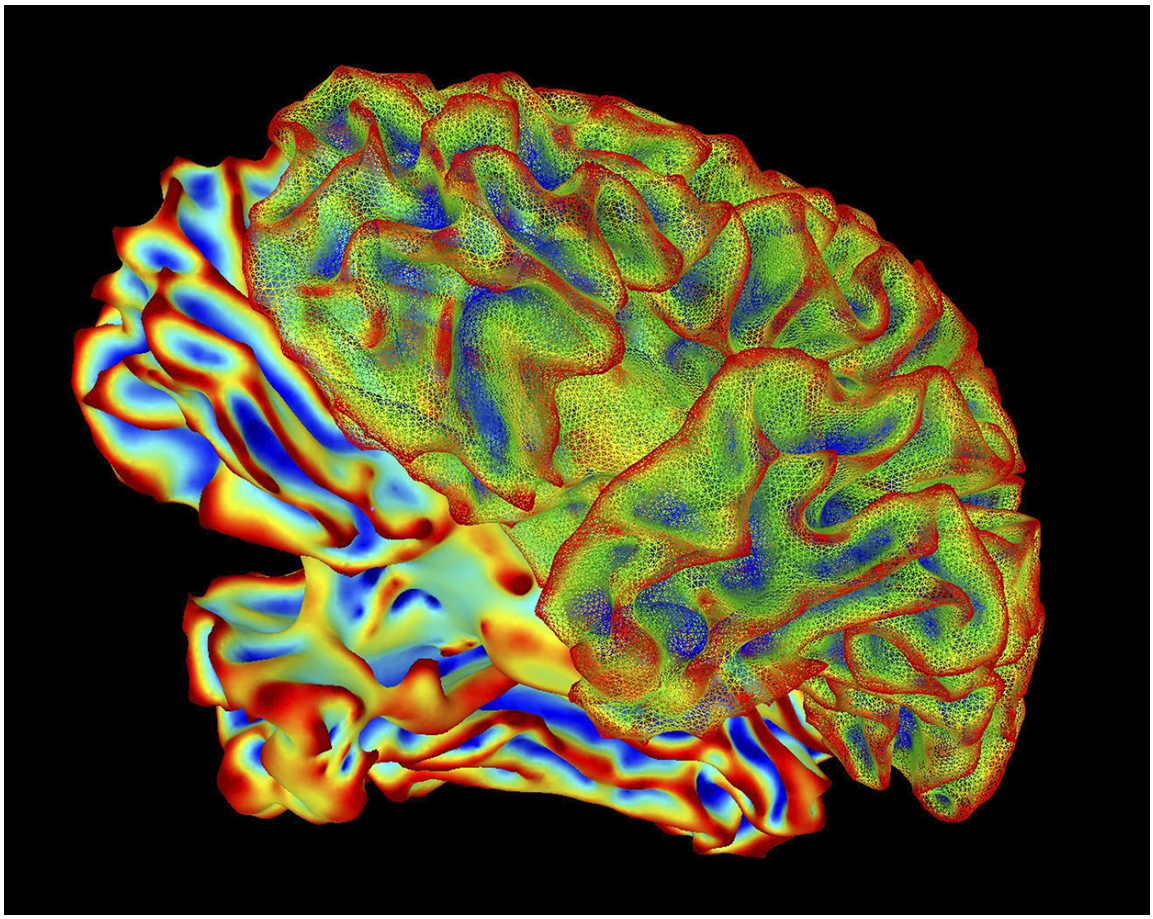
Monte Carlo Tree Search

CSI 4106 Introduction to Artificial Intelligence

Marcel Turcotte
Version: Nov 22, 2026 13:11

Preamble

Message of the Day



‘Mind-captioning’ AI decodes brain activity to turn thoughts into text
Nature News, 2025-11-05.

Brief summary (Generated by gtp-5-mini on 2025-11-21):

- What it is: “Mind-captioning” is a technique that decodes brain activity to generate descriptive sentences of what a person is seeing or imagining.

- How it works: researchers used a deep-language model to convert captions from >2,000 videos into numerical “meaning signatures,” then trained a separate model to map six participants’ brain-scan patterns (while watching or recalling videos) to those signatures.
- Key findings: the method can recover descriptive content from both perception and memory, offering clues about how the brain represents meaning before language is produced.
- Potential uses: could help people with language impairments communicate and advance understanding of neural representations of thought.
- Risks and caveats: prior methods sometimes confounded model-generated language with brain content; this approach aims to link brain patterns to pre-existing meaning signatures. The work raises serious mental-privacy concerns (surveillance, manipulation, discrimination) as decoding thought content becomes more accurate.

Learning objectives

- **Explain** the concept and key steps of Monte Carlo Tree Search (MCTS).
- **Compare** MCTS with other search algorithms like BFS, DFS, A*, Simulated Annealing, and Genetic Algorithms.
- **Analyze** how MCTS balances exploration and exploitation using the UCB1 formula.
- **Implement** MCTS in practical applications such as Tic-Tac-Toe.

Introduction

Monte Carlo Tree Search (MCTS)

In the introductory lecture on **state space search**, I used **Monte Carlo Tree Search (MCTS)**, a key component of **AlphaGo**, to exemplify the role of search algorithms in reasoning.

Today, we conclude this series by examining the implementation details of this algorithm.

Applications

- De novo **drug design**
- Electronic **circuit** routing
- Load monitoring in **smart grids**
- **Lane keeping** and overtaking tasks
- Motion planning in **autonomous driving**
- Even solving the **travelling salesman problem**

See: Kemmerling et al. (2024)

The paper by Kemmerling and colleagues (Kemmerling et al. 2024) illustrates the wide range of applications for MCTS when combined with deep neural networks.

Applications (continued)

See also Besta et al. (2025) on the role of MTCS in Reasoning Language Models (RLMs).

Historical Notes

- **2008**: the algorithm is introduced in the context of **AI game** (Chaslot et al. 2008)
- **2016**: the algorithm is combined with **deep neural networks** to create **AlphaGo** (Silver et al. 2016)

Definition

A **Monte Carlo algorithm** is a computational method that uses **random sampling to obtain numerical results**, often used for optimization, numerical integration, and probability distribution estimation.

It is characterized by its ability to handle **complex problems** with probabilistic solutions, trading **exactness** for **efficiency** and **scalability**.

Have you previously encountered the term "**Monte Carlo algorithm**"?

Have you previously encountered the term "Monte Carlo algorithm"? If so, please provide representative examples of Monte Carlo methods.

Monte Carlo integration, approximating integrals via random sampling, is a canonical example of Monte Carlo methods.

The method draws $x \sim \mathrm{Uniform}(a, b)$, then uses

$$\int_a^b f(x), dx \approx (b-a), \frac{1}{n} \sum_{i=1}^{n} f(X_i).$$

See Section 6.1 for examples of source code. ## Algorithm For a specified number of iterations (**simulations**): 1. **Selection** (guided tree descent) 2. Node **expansion** 3. **Rollout** (simulation) 4. **Back-propagation** A Monte Carlo algorithm uses random sampling (**simulations**) for probability distribution estimation. Note that Step 4, labeled "back-propagation," is not the same as the gradient-based backpropagation algorithm used to train neural networks. MTCS does not compute gradients or perform any gradient-based optimization. - **MCTS is not a single fixed algorithm**, many variants exist in the literature and in practice. - The four canonical steps (**Selection, Expansion, Simulation, Backpropagation**) admit **multiple design choices**: - **Expansion**

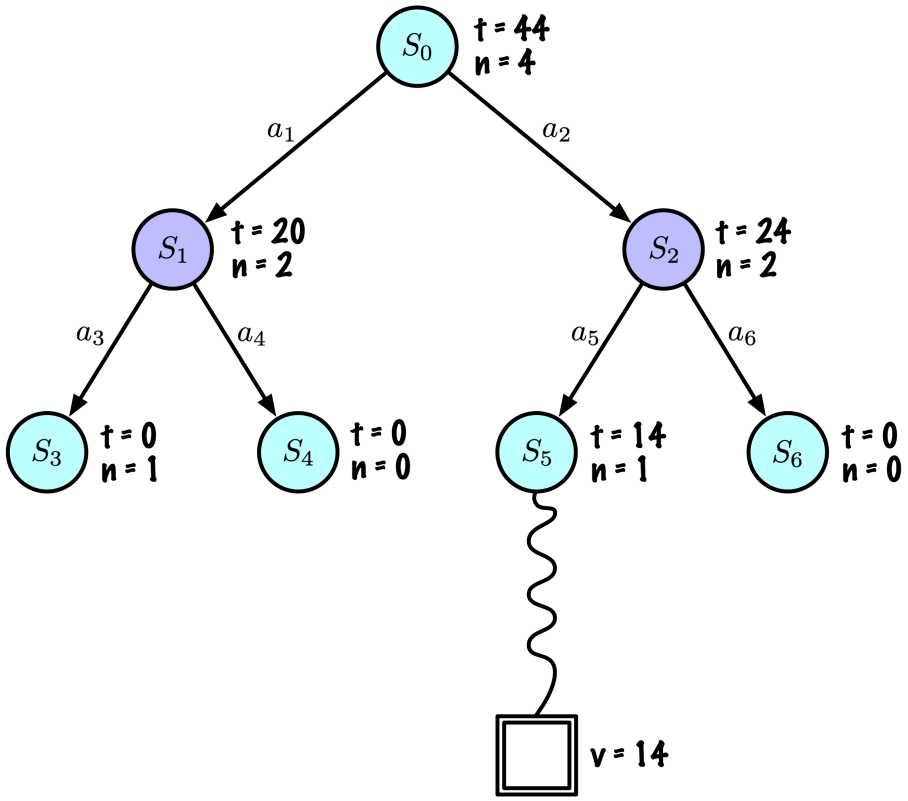
strategies vary widely: - *Incremental expansion*: add **one** child per visit (common in classical UCT). - *Full expansion*: add **all legal children** at once (common in some game engines). - **Simulation policies** range from pure random playouts to heuristic-guided rollouts. - **Backpropagation updates** may use win counts, average values, prior probabilities, or value-network estimates. - **Tree persistence** may reset every move, persist within a game, or persist **across** games. **Implication:** MCTS should be understood as a *family* of algorithms—different implementations may behave very differently depending on expansion, rollout, and backup rules. ## Algorithm

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

Attribution: Russell and Norvig (2020), Figure 5.11 ## Any-Time Algorithm **MCTS** is a textbook example of an *any-time algorithm*: - It can be interrupted at any moment. - **More time ⇒ more simulations ⇒ better action estimates**. - It returns *the best current move* given whatever number of iterations have been completed. This is exactly how it is used in Go, chess, Atari, MuZero, etc.: run until the time budget expires, then act. ## Discussion Like other algorithms previously discussed, such as **BFS**, **DFS**, and $A^\star$, Monte Carlo Tree Search (MCTS) maintains a **frontier** of unexpanded nodes. Examining the relationship between MCTS and previous search algorithms offers valuable insights into their similarities and differences, providing an excellent opportunity to synthesize key concepts. ## Discussion Similar to $A^\star$, Monte Carlo Tree Search (MCTS) employs a heuristic, referred to as a **policy**, to determine the **next node for expansion**. However, in $A^\star$, the heuristic is typically a **static** function estimating cost to a goal, whereas in MCTS, the "policy" involves **dynamic** evaluation. By "static evaluation," we mean a function that yields the same result for a given state, irrespective of the moment the function is called within the program's execution. ## Discussion Similar to **Simulated Annealing** and **Genetic Algorithms**, Monte Carlo Tree Search (MCTS) incorporates a mechanism to **balance exploration** and **exploitation**. ## Discussion - MCTS **leverages all visited nodes** in its **decision-making process**, unlike $A^\star$, which primarily focuses on the current frontier. - Additionally, **MCTS iteratively updates the value of its nodes based on simulations**, whereas $A^\star$ typically uses a **static** heuristic. The efficacy of Monte Carlo Tree Search (MCTS) in identifying promising nodes improves as the execution time is extended. The comparison between simulated annealing and MCTS will be revisited shortly. ## Discussion In contrast to previous algorithms with **implicit** search trees, MCTS constructs an **explicit tree structure** during execution. As we will explore, MCTS maintains **both explicit** and **implicit** representations of the search tree. While previous algorithms often imply a search tree structure without explicitly constructing it, MCTS explicitly builds and maintains a tree structure during execution.

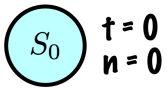
This explicit tree is used to record the outcomes of simulations and guide decision-making. The explicit tree tracks visited states and their evaluations, while the implicit aspect refers to the expansion of the tree during simulations. ## Walk-through

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



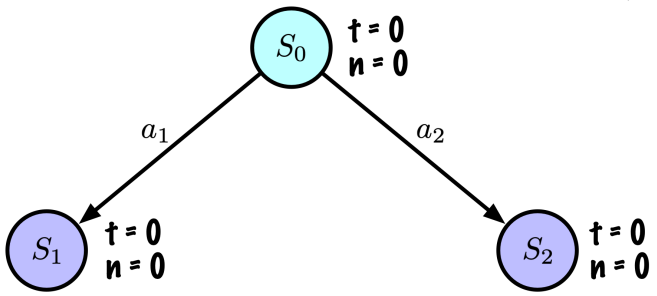
Adapted from: Monte Carlo Tree Search by John Levine posted on YouTube on 2017-03-06. We begin by providing an overview of the algorithm's execution, followed by an in-depth analysis of its individual components. The accompanying diagram illustrates the algorithm's operation after four iterations. Before delving into the specifics of each iteration, it is essential to elucidate the fundamental concepts. In the diagram, the blue and purple nodes depict the explicit search tree constructed by Monte Carlo Tree Search (MCTS) during each iteration. Each node maintains a record of the number of visits and its cumulative score. The edges between nodes are annotated with actions. For example, at the algorithm's initiation, two actions are available. Upon selecting action a_1, two additional actions become accessible. The algorithm employs the UCB1 (Upper Confidence Bound 1) formula to select the subsequent node to visit, thereby guiding the descent process. Upon reaching a leaf node, the algorithm conducts a simulation, known as a rollout, to estimate the node's potential utility. ## Walk-through

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



Each node keeps track of the number of visits (n) and a total score (t). S_0 is the initial state. ## Walk-through

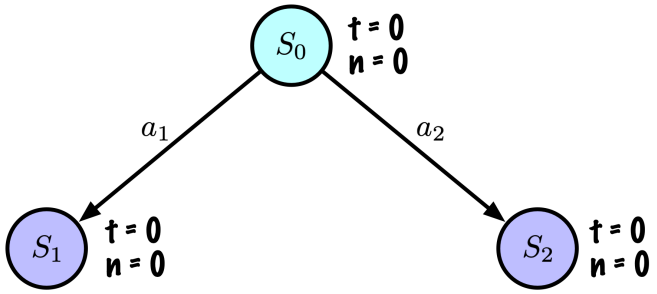
$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



Adding the available actions, a_1 and a_2, as well as the corresponding states, S_1 and S_2.

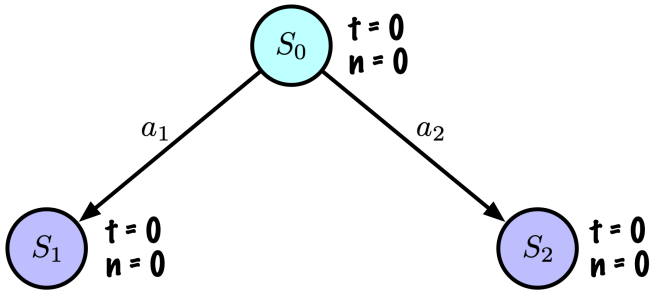
Walk-through (1.1)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



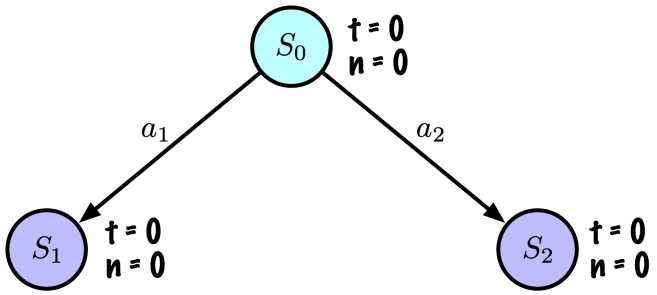
1.1 Start of the first iteration: Selection Step. ## Walk-through (1.1)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



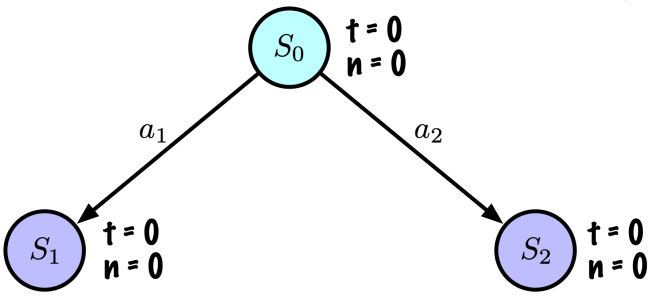
1.1 The UCB1 score of S_1 and S_2 are both ∞ since $n_1 = n_2 = 0$. We can select the either node. ## Walk-through (1.1)

$$\text{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



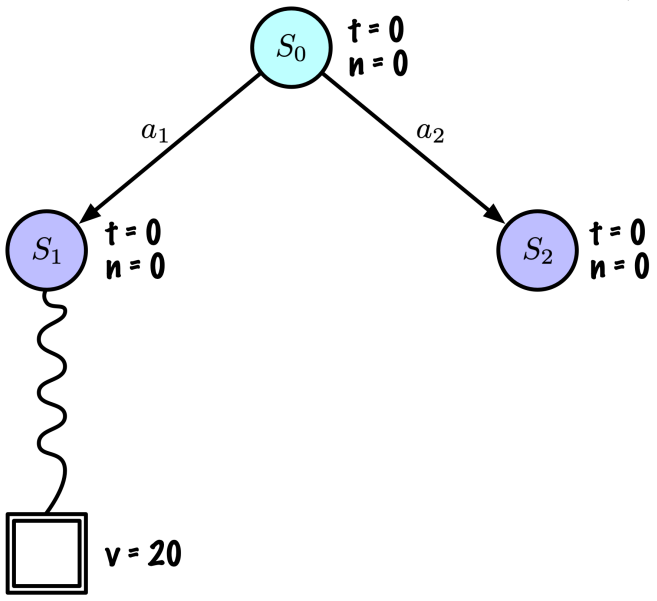
1.1 We reached a leaf node, S_1. ## Walk-through (1.2)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



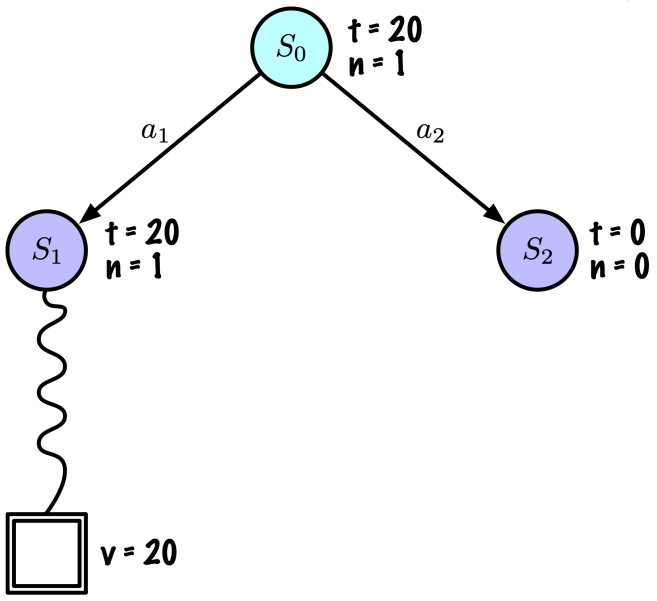
1.2 Node expansion. This node has not been visited yet. Therefore, no expansion. The node's unvisited status is indicated by $n = 0$. Prior to expanding the node, it is essential to evaluate its utility, which is accomplished through a rollout in the subsequent step. ## Walk-through (1.3)

$$\text{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



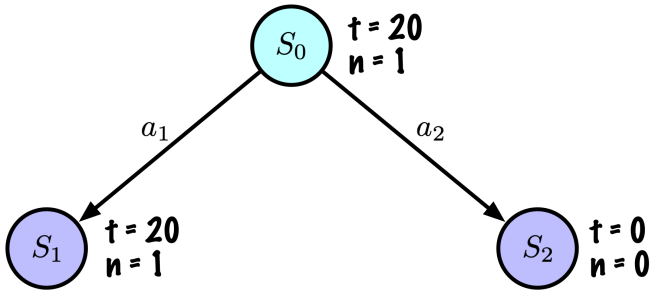
1.3 A rollout (simulation) is simply randomly selecting actions until a terminal node is found. In a rollout simulation, the algorithm iteratively selects subsequent actions until it reaches a terminal node. At this stage, the utility of the resultant state is assessed (`game.evaluate(state, player)`). Notably, the trajectory from the selected node to the terminal state is not tracked, as indicated by the wiggly line in the diagram. Rollout simulations proceed from the selected node either until a terminal state is encountered or a predetermined depth limit is achieved. If the latter occurs, a heuristic or value function is employed to evaluate the state. ## Walk-through (1.4)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



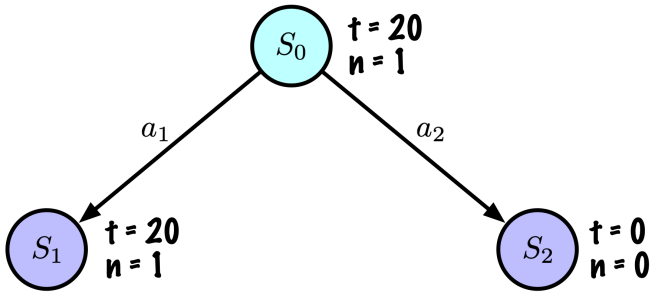
1.4 Back-propagation. ## Walk-through (1.End)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



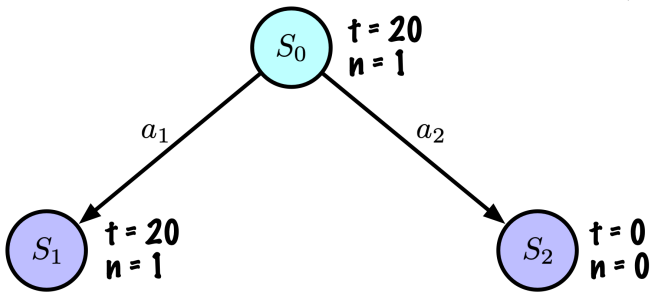
End of iteration 1. ## Walk-through (2.1)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



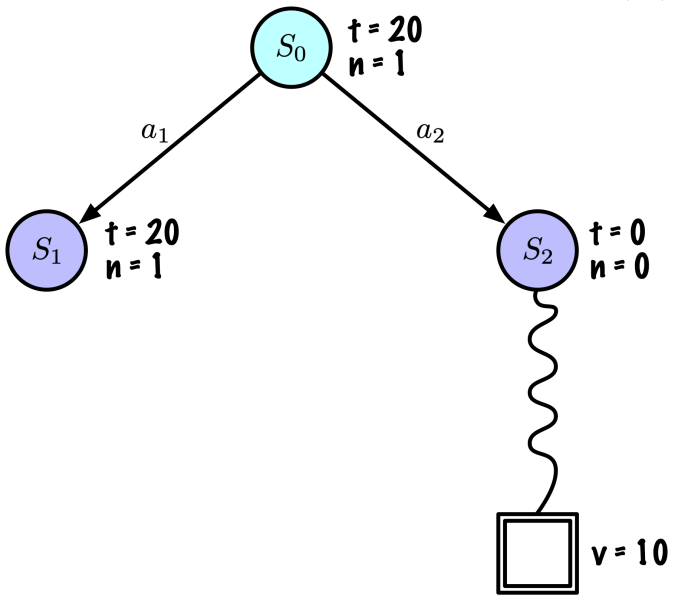
2.1 Selection. Computing the UCB1 value of $S_1 = 20 + 2\sqrt{\frac{\ln(1)}{1}}$ and $S_2 = \infty$. We select S_2. To determine the next action, the algorithm must first ascertain the utility values of all immediate child nodes. ## Walk-through (2.2)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



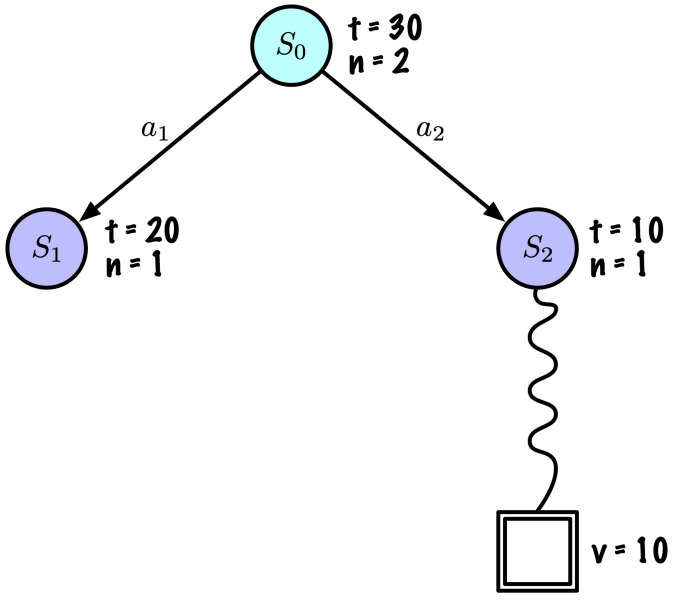
2.2 Expansion. This node has not been visited yet. Therefore, no expansion. ## Walk-through (2.3)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



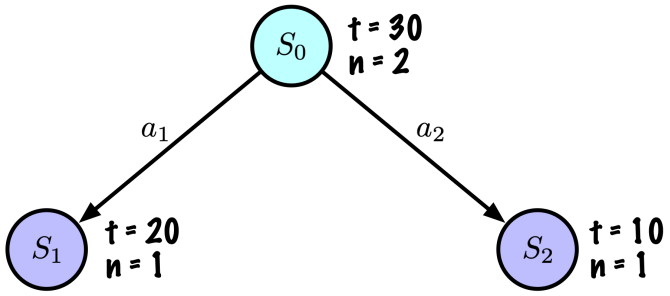
2.3 Rollout. ## Walk-through (2.4)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



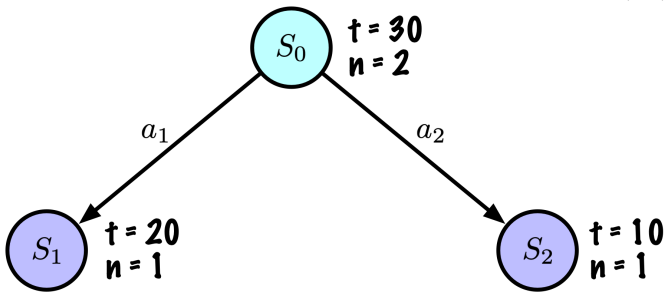
2.4 Back-propagation. ## Walk-through (2.End)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



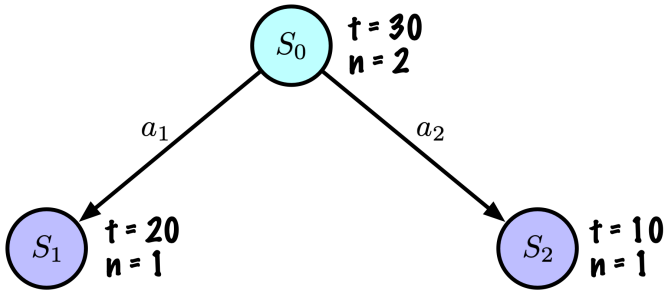
End of iteration 2. ## Walk-through (3.1)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



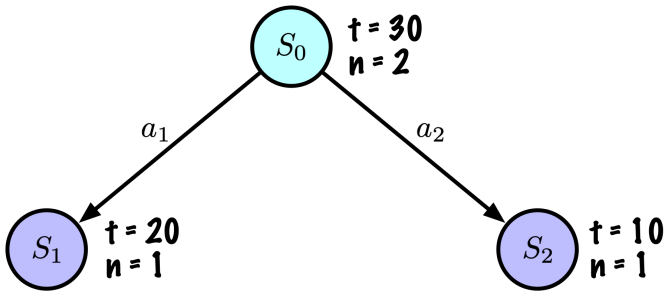
3.1 Selection. Calculating UCB1 values. ## Walk-through (3.1)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



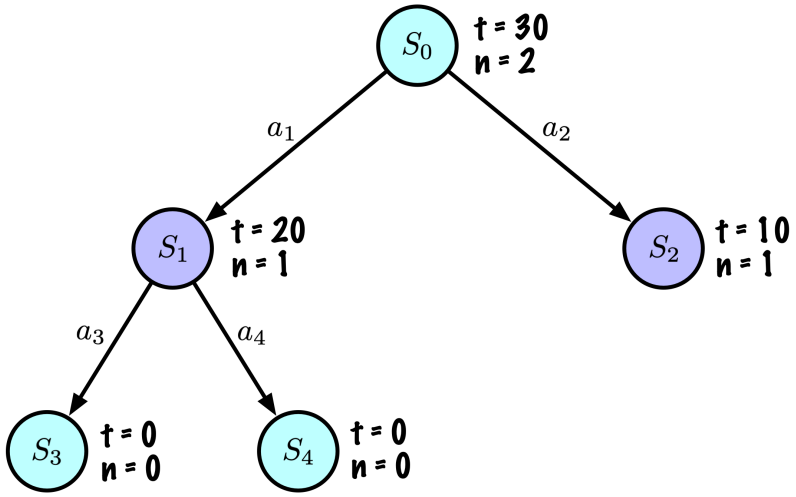
3.1 Selection. Computing the UCB1 value of $S_1 = 20 + 2\sqrt{\frac{\ln(2)}{1}} = 21.67$ and $S_2 = 10 + 2\sqrt{\frac{\ln(2)}{1}} = 11.67$. Selecting S_1 ## Walk-through (3.2)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



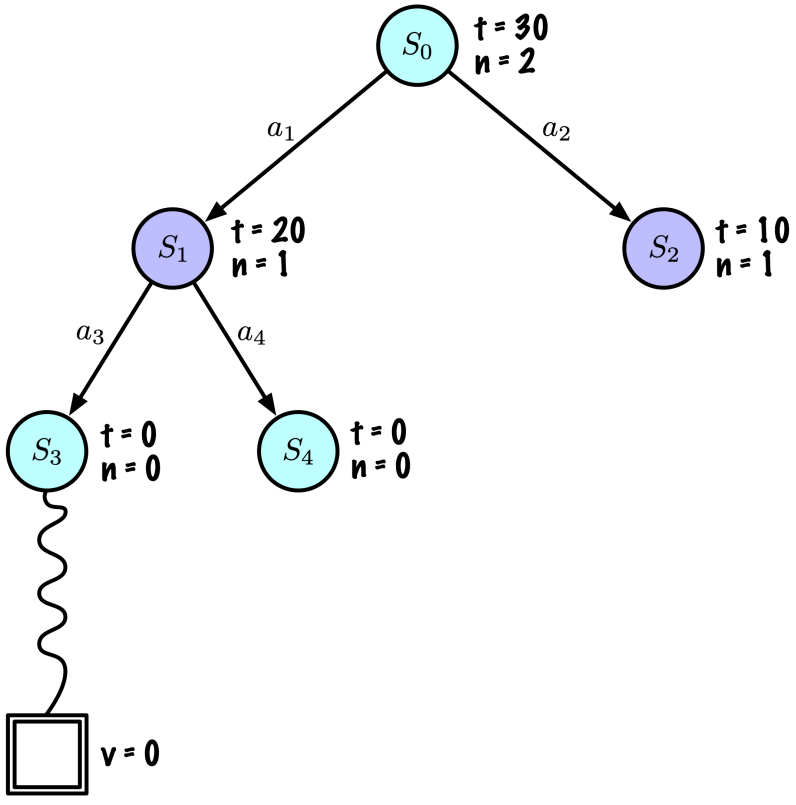
3.2 Node expansion. ## Walk-through (3.2)

$$\text{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



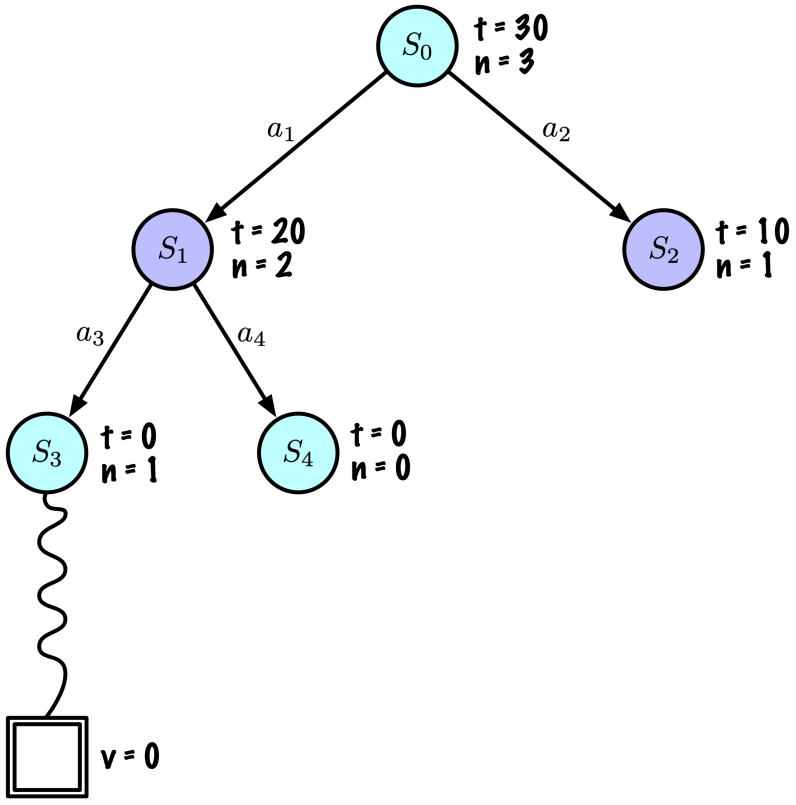
3.2 Node expansion. Since $n_1 > 0$, the node is expanded. ## Walk-through (3.3)

$$\text{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



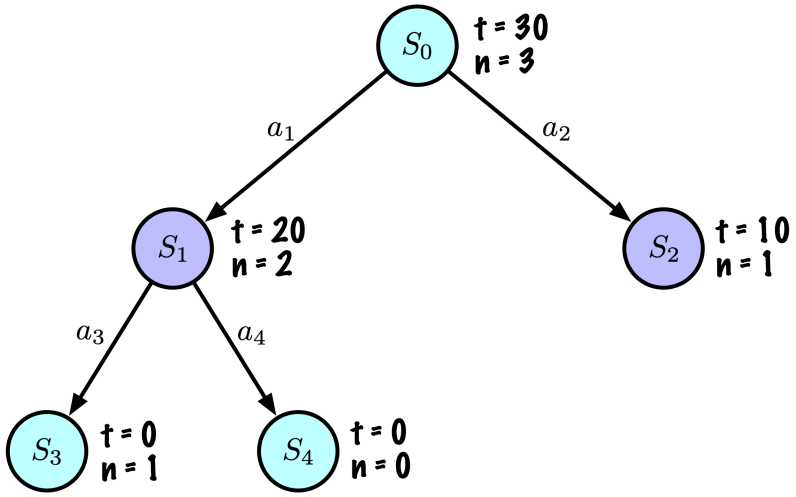
3.3 Rollout. ## Walk-through (3.4)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



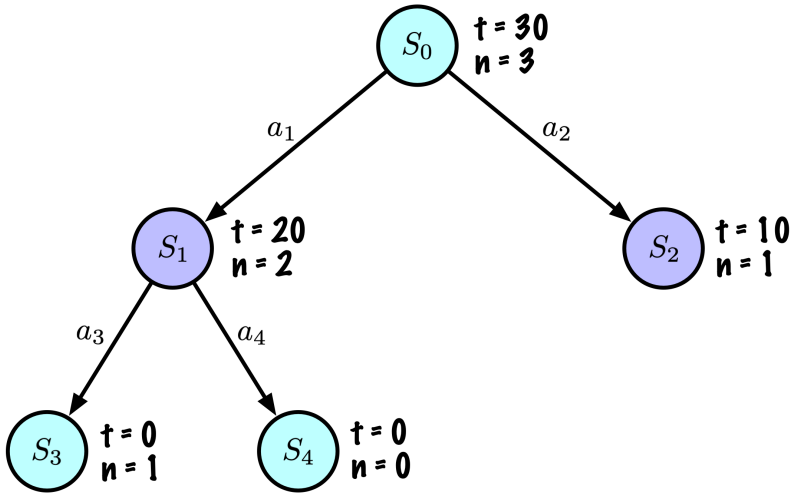
3.4 Back-propagation. ## Walk-through (3.End)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



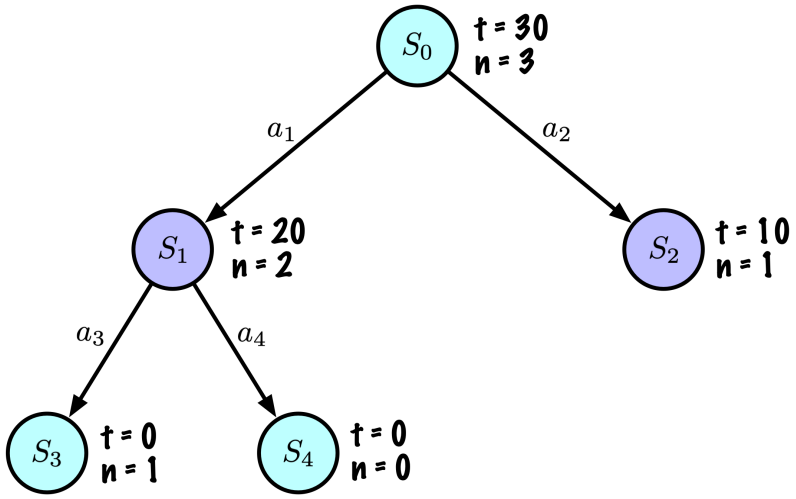
End of iteration 3. ## Walk-through (4.1)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



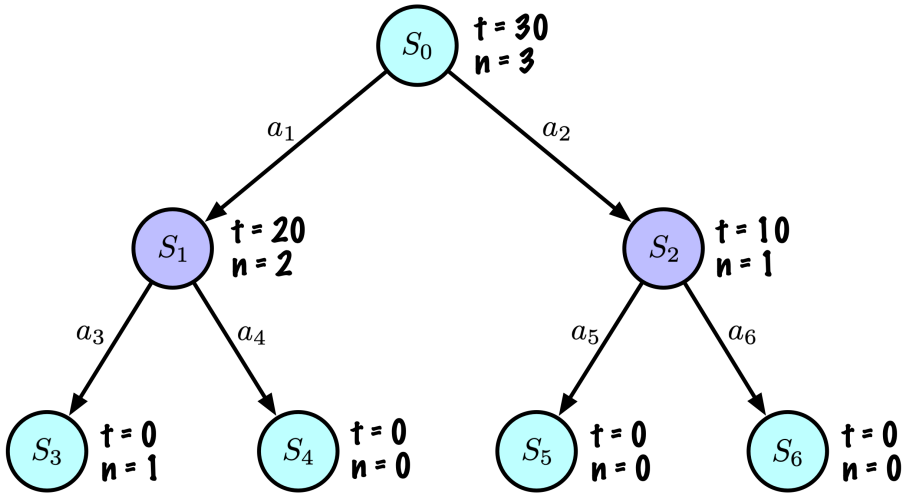
4.1 Selection. Calculating UCB1 values. ## Walk-through (4.1)

$$\text{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



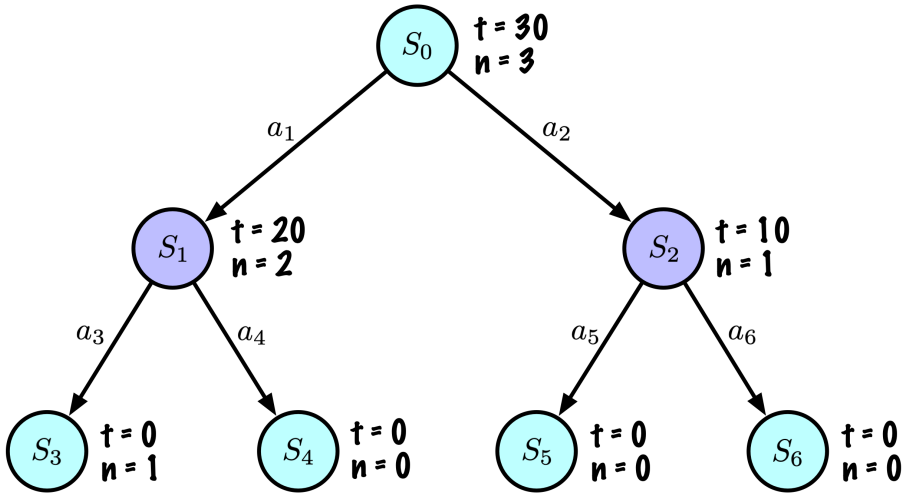
Computing the UCB1 value of $S_1 = \frac{20}{2} + 2\sqrt{\frac{\ln(3)}{2}} = 10 + 2\sqrt{\frac{\ln(3)}{2}} = 11.48$ and $S_2 == 10 + 2\sqrt{\frac{\ln(3)}{1}} = 12.10$. Selecting S_2 ## Walk-through (4.2)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



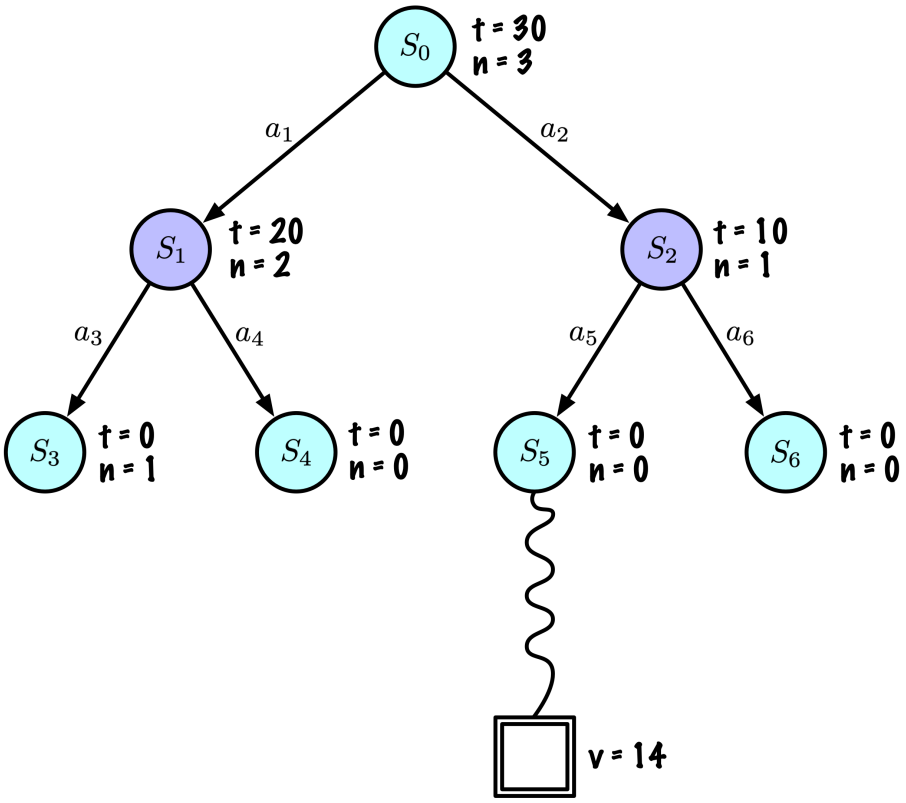
4.2 Expansion. ## Walk-through (4.2)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



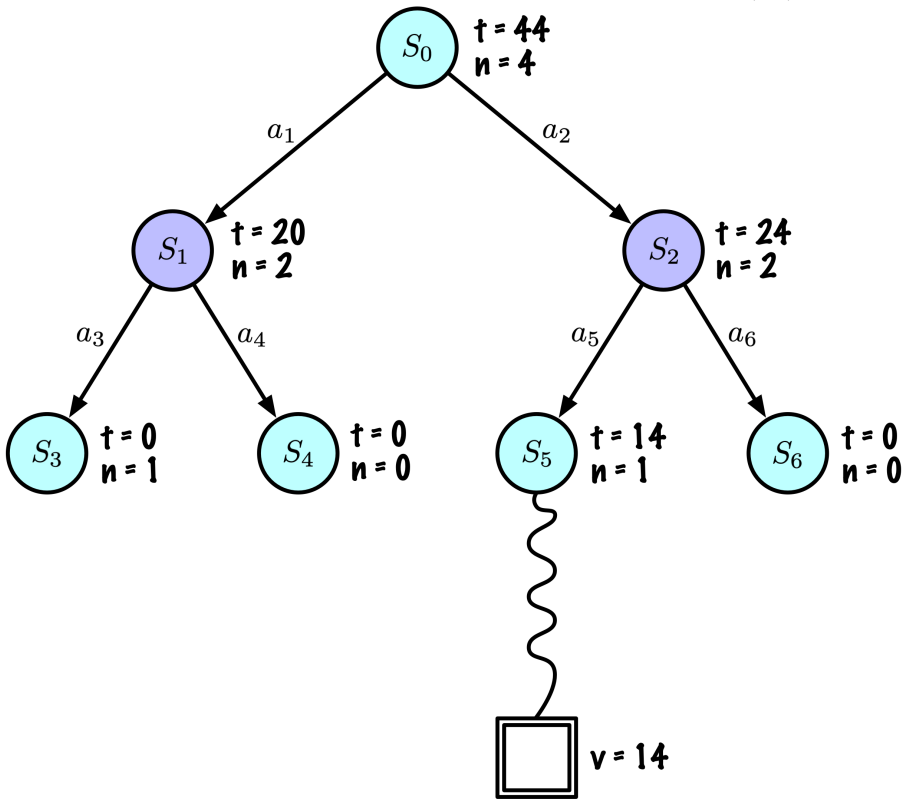
4.2 Expansion. Both nodes have the same USCB1 value, ∞. Selecting S_6. ## Walk-through (4.3)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



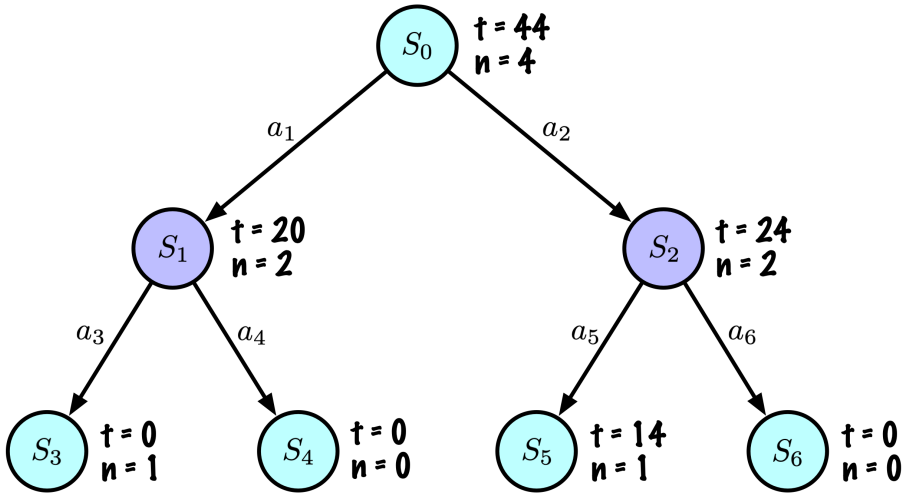
4.3 Rollout. ## Walk-through (4.4)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



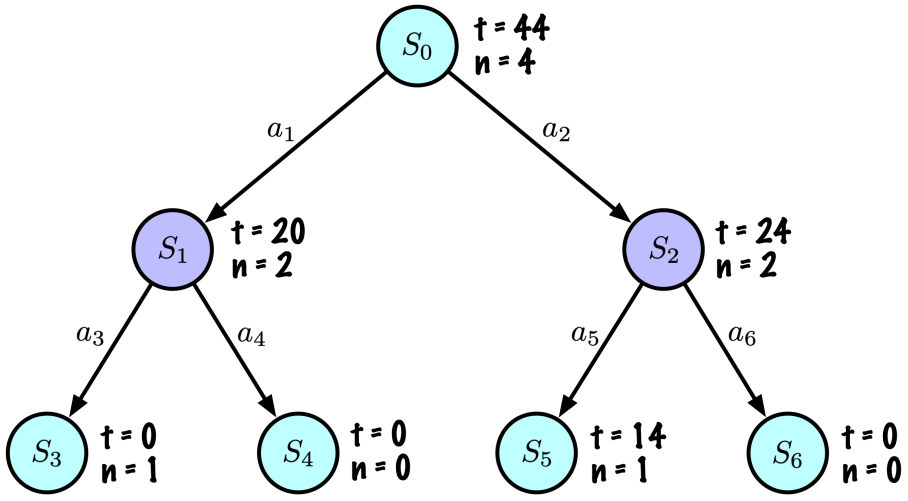
4.4 Back-propagation. ## Walk-through (4.End)

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



End of iteration 4. ## Walk-through

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$



If the algorithm halts at this stage, it will recommend a_2 as the optimal move, given that S_2 possesses the highest average score. In practical applications, various strategies are employed to determine the optimal action. One common approach is to select the node with the highest number of visits, as this method may offer greater robustness compared to choosing the node based solely on the highest average score. In applications such as chess, Go, or Atari games, MCTS conducts 1000 to 2000 simulations per move. This seemingly low count is attributed to the use of deep learning algorithms, which direct the search through tree and default policies. Each set of iterations, for instance, 1000, is utilized solely to determine the subsequent optimal move. During the initial iterations, MCTS operates with limited information for selecting the next best move. As iterations increase, the estimates become more refined. The number of nodes in a search tree after 1000 iterations of MCTS depends on several factors, including the branching factor at each node and the specific policy used for node expansion. Generally, each iteration of MCTS consists of four main steps: selection, expansion, simulation, and backpropagation. During the expansion phase, a new node is added to the tree. In a typical MCTS setup: 1. **Selection**: Descent the existing tree from the root to a leaf node using a tree policy, often based on Upper Confidence Bounds for Trees (UCT). 2. **Expansion**: Add one or more child nodes to the selected node if it is not fully expanded. 3. **Simulation**: Perform a simulation from the new node to a terminal state. 4. **Backpropagation**: Update the value estimates of the path nodes based on the simulation result. Assuming that each iteration expands exactly one new node, the

search tree will have approximately 1000 additional nodes after 1000 iterations. However, the actual number can vary if multiple nodes are expanded per iteration or due to the tree's initial setup and other variations in the algorithm's implementation. ## Russell and Norvig

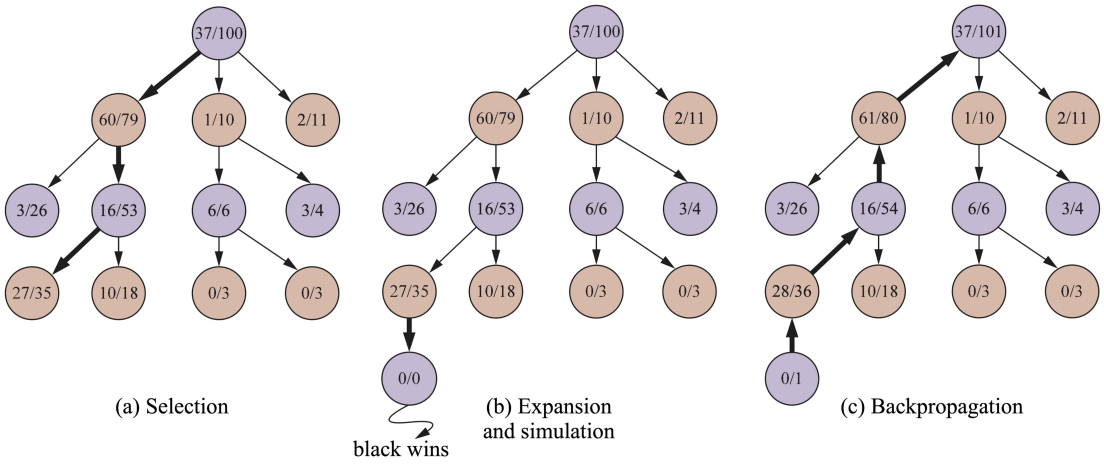


Figure 5.10 One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked 27/35 (for 27 wins for black out of 35 playouts). In (b) we expand the selected node and do a simulation (playout), which ends in a win for black. In (c), the results of the simulation are back-propagated up the tree.

Attribution: Russell and Norvig (2020), Figure 5.10 This example includes a significantly larger number of nodes, which may aid in comprehending the selection step more effectively. The algorithm seeks to navigate toward the most promising area of the tree, characterized by the highest average score, and subsequently expands this section of the search tree. Observe that the backpropagation step updates all nodes along the path from the selected node to the root. This update can influence the path chosen in the subsequent iteration. Because the algorithm operates in an adversarial setting, we count only victories attributed to black nodes. ## Summary: tree building Initially, the tree has one node, it is S_0. We add its descendants and we are ready to start. The Monte Carlo Tree Search slowly builds its search tree. ## Summary: 4 steps With each iteration, the following steps occur: 1. **Selection**: Identify the "best" node by descending a single path in the tree, guided by UCB1. 2. **Expansion**: Expand the node if it is a leaf in the MCTS Tree and $n > 0$. 3. **Rollout**: Simulate a game from the current state to a terminal state by randomly selecting actions. 4. **Backpropagation**: Use the obtained information to update the current node and all parent nodes up to the root. ## Summary: nodes Each node records its **total score** and **visit count**. This information is used to calculate a **value that guides tree descent**, balancing **exploration** and **exploitation**. ## Summary: exploration vs exploitation

$$\mathrm{UCB1}(S_i) = \overline{V_i} + C\sqrt{\frac{\ln(N)}{n_i}}$$

The usual value for C is $\sqrt{2}$. **Exploration** essentially occurs when **two nodes** have approximately the **same average score**, then MCTS favours nodes with **fewer visits** (dividing by n). For $n < \ln(N)$, the value of the ratio is greater than 1, whereas for $n > \ln(N)$, the ratio becomes less than 1. So there is a small fraction of the time where exploration kicks in. But even then, the contribution of the ratio is quite tame, we're taking the square root of that ratio, multiplied by $\sqrt{2} \sim 1.414213562$. ## Summary: exploration vs exploitation In **simulated annealing**, the initial temperature and the scale of the objective function are linked. Acceptance rule for a candidate move with score change $\Delta E = E_{\text{new}} - E_{\text{old}}$: - If $\Delta E \leq 0$: always accept (better or equal solution). - If $\Delta E > 0$: accept with probability

$$p = \exp(-\Delta E/T).$$

Summary: exploration vs exploitation In **Simulated Annealing**: - T defines how "big" a bad move has to be before it is unlikely to be accepted. - If T is large compared to typical ΔE: - Even sizeable worsening moves have reasonable probability. - Very exploratory. - If T is small: - Only very small worsening moves are accepted. - Mostly exploitative / hill-climbing. That's why you often pick initial T using the *distribution* of ΔE on random states: e.g., "set T_0 so that a typical ΔE has, say, 60-80% acceptance." It's explicitly **tied to the scale of the scoring function**. ## Summary: C as an exploration scale In UCT (UCB1) we're using

$$\text{score}(i) = V_i + C\sqrt{\frac{\ln N}{n_i}},$$

where: - V_i: average playout value of child i (exploitation term), - N: total visits to the parent node, - n_i: visits to child i, - C: exploration constant. ## Summary: C as an exploration scale At a given node: - The child with largest $\text{score}(i)$ is selected. - The second term

$$C\sqrt{\frac{\ln N}{n_i}}$$

is **pure exploration**: large when n_i is small, shrinking as you visit that child. ## Summary: C as an exploration scale Consider two children, 1 and 2. You choose 2 instead of 1 when:

$$V_2 + C\sqrt{\frac{\ln N}{n_2}} > V_1 + C\sqrt{\frac{\ln N}{n_1}}.$$

Rearrange:

$$V_2 - V_1 > C\left(\sqrt{\frac{\ln N}{n_1}} - \sqrt{\frac{\ln N}{n_2}}\right).$$

Summary: C as an exploration scale

- The **difference in average playout values** that can be "overruled" by exploration is proportional to C.
- **Larger** C → **exploration** term dominates more → you're willing to try a child whose V_i is significantly worse, just because it's under-explored.
- **Smaller** C → you stick more to the **currently best-looking** V_i.

1. Extreme cases

- $C = 0$ (no exploration term)
  - UCT score = purely the average value.
  - The algorithm becomes **greedy**:
    - It keeps going down the move that *currently* looks best.
    - Other moves may get very few (or zero) visits.
  - Consequences:
    - Can lock into a **wrong move** if early rollouts were unlucky.
    - Tree is very **narrow and deep**.
    - With noisy rollouts and few simulations, play can be surprisingly bad.
- C **very large**
  - Exploration bonus dominates:
    - Even if a move's average value looks worse, if it has fewer visits it still gets chosen a lot.
  - Consequences:
    - Search becomes **very exploratory**, almost like "systematic dithering".
    - Tree is **wide and shallow**.
    - Values at the root converge slowly; decisions are noisy.
    - With limited simulations, move choice can look close to random.

2. Moderate C: what actually changes as you vary it

For a sensible interval (say C in $[0.5, 2]$ for tic-tac-toe with rewards in $[-1, 1]$):

- **Smaller** C **(e.g. 0.3–0.7)**:
  - Faster **commitment** to the currently best-looking move.
  - Tree shape: **few branches highly explored**, others barely touched.
  - Good when:

    - Rollouts are relatively **low-noise**,
    - You have **limited simulation budget**,
    - The "best" branch is easy to identify.
  - Risk: can **miss** strong but initially unlucky moves.

- **Larger C (e.g. 1.0–2.0)**:
  - More **balanced coverage** of children:
    - Even if one move looks slightly better, others still get substantial visits.
  - Tree shape: **wider** near the root, more even visit counts across moves.
  - Good when:
    - Rollouts are **noisy**,
    - You want the algorithm to be more "open-minded" about alternatives.
  - Risk: spends **too much** time on clearly bad branches if budget is small.

In aggregate metrics (e.g. win rate vs Random for fixed number of simulations):

- As you **increase C from 0**:
  - Performance usually **improves** at first (you stop being myopic).
- Past some point:
  - Performance then **degrades** (you explore so much you don't refine the best line enough).

So you typically see a **U-shaped curve** in performance as a function of (C): too little exploration is bad, too much is bad, there's a "sweet spot" that depends on the game, rollout noise, and simulation budget.

3. What to look for in tic-tac-toe

If you vary C and watch:

- **Tree statistics**:
  - Distribution of visit counts per child at the root.
  - Depth vs breadth of the explored tree.
- **Game outcomes** vs a fixed opponent (e.g. Random or Minimax):
  - With small C, you may see odd blunders due to "early over-commitment".
  - With large C, you may see "too much experimentation", especially with few simulations.
  - For a mid-range C, the solver stabilizes into strong, consistent play.

That's the concrete, observable effect of changing C: it reshapes the trade-off between **digging deep into what looks good now** versus **giving other moves a fair chance**.

Summary: C as an exploration scale

In the **classical UCB1 theory**, rewards are assumed to be in $[0, 1]$, and there's a specific recommended constant (e.g. $\sqrt{2}$). If your reward scale is different (say in $[-1, 1]$ or large magnitude), you essentially **rescale** that constant; in practice people tune C empirically.

Summary: C as an exploration scale

Analogy:

- Simulated annealing's T and MCTS's C both **balance exploration vs exploitation**.
- In both cases, their *effective* meaning depends on the **scale of the objective / rewards**.
- In SA: "how bad can a move be and still often be accepted?"
- In MCTS: "how much worse can a child's current V_i be and still get chosen for exploration?"

Summary: C as an exploration scale

Key differences:

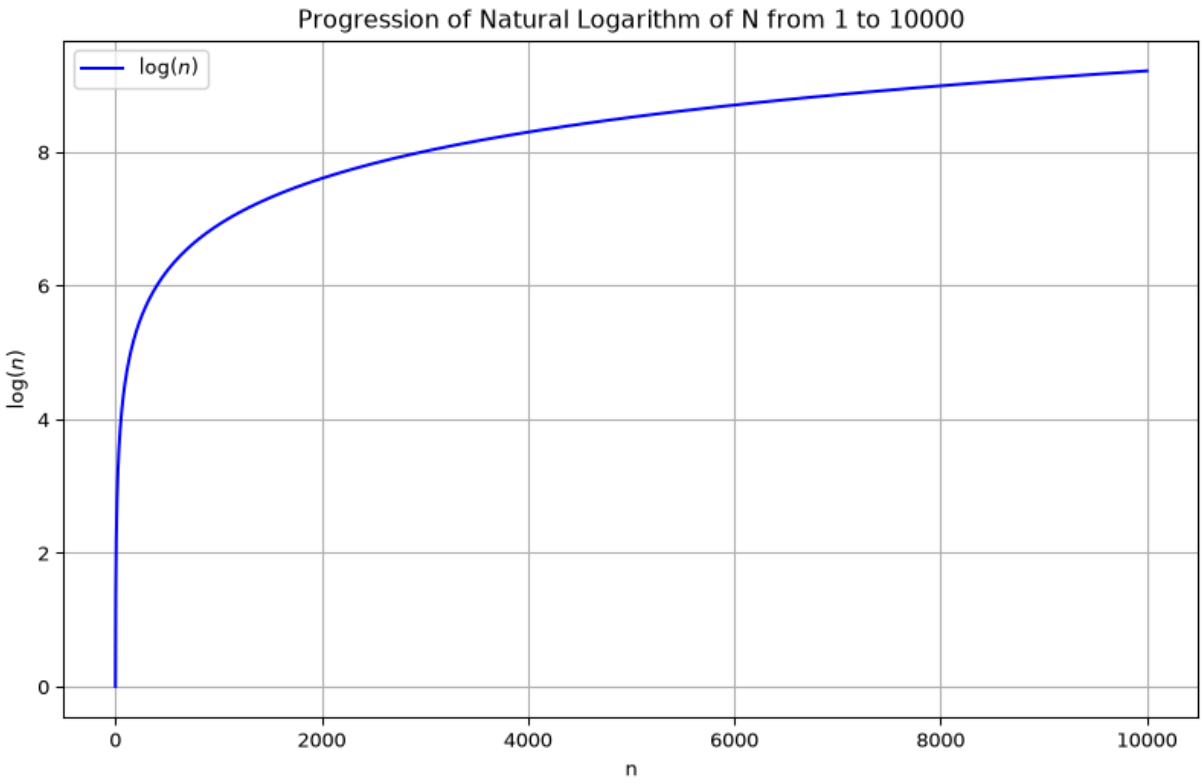
- **Simulated Annealing**:
  - Single trajectory.
  - T is **explicitly scheduled** (high to low) over time.
  - Balances local moves in a single search path.

Summary: C as an exploration scale

- **MCTS (UCT)**:
  - Tree of many paths.
  - C is constant, but exploration **decays automatically** via $\sqrt{\ln N / n_i}$:
    - **Early**: n_i small → high exploration.
    - **Late**: n_i big → exploration term shrinks, behavior gets more greedy.

Summary

In [1]:

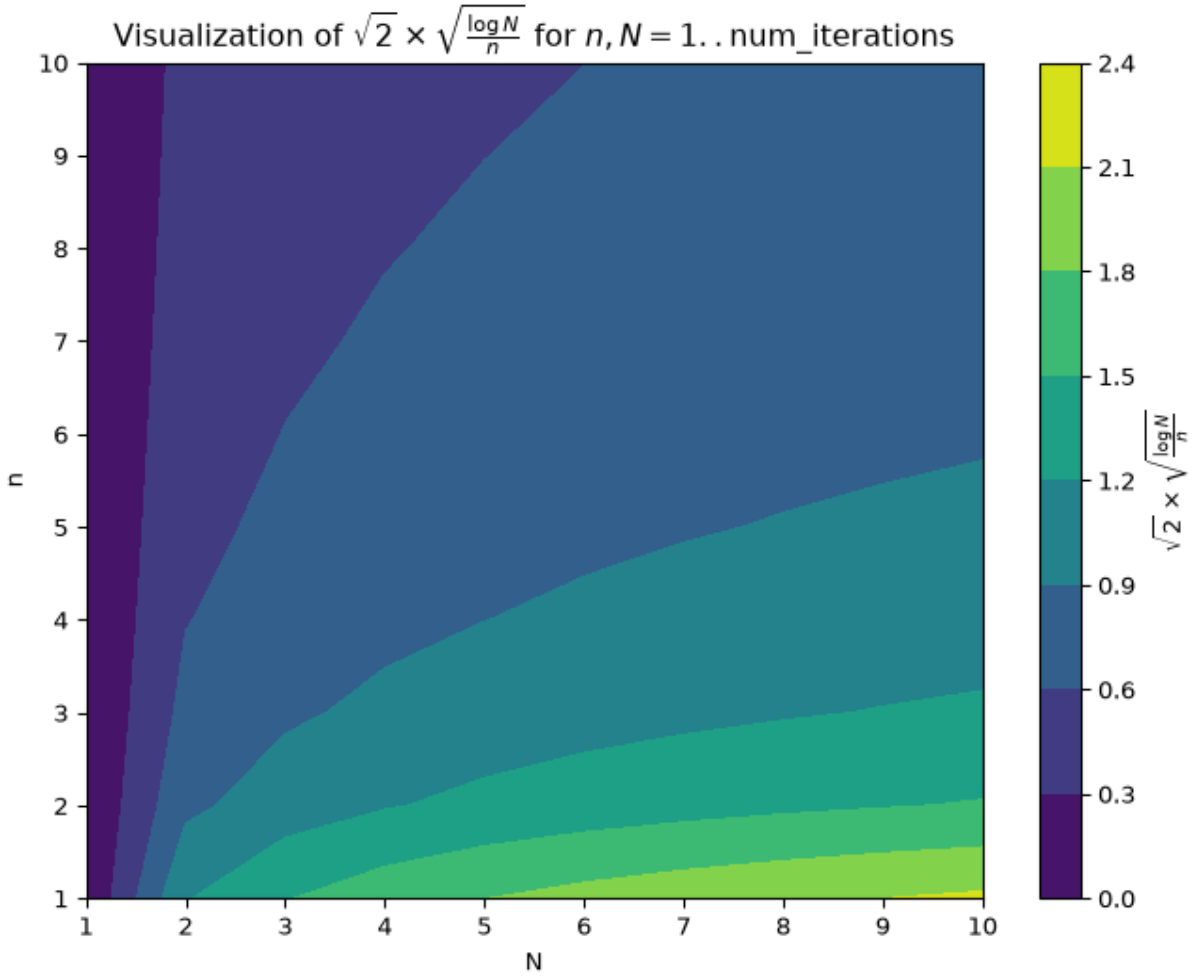



```
[4.60517019 6.90775528 9.21034037]
```

Summary

In [2]:

```python
import numpy as np
import matplotlib.pyplot as plt

num_iterations = 10

# Define the range for n and N
n_values = np.arange(1, num_iterations + 1)
N_values = np.arange(1, num_iterations + 1)

# Prepare a meshgrid for n and N
N, n = np.meshgrid(N_values, n_values)

# Compute the expression for each pair (n, N)
Z = np.sqrt(2) * np.sqrt(np.log(N) / n)

# Plotting
plt.figure(figsize=(8, 6))
plt.contourf(N, n, Z, cmap='viridis')
plt.colorbar(label=r'$\sqrt{2} \times \sqrt{\frac{\log{N}}{n}}$')
plt.xlabel('N')
plt.ylabel('n')
plt.title(r'Visualization of $\sqrt{2} \times \sqrt{\frac{\log{N}}{n}}$ for
plt.show()
```

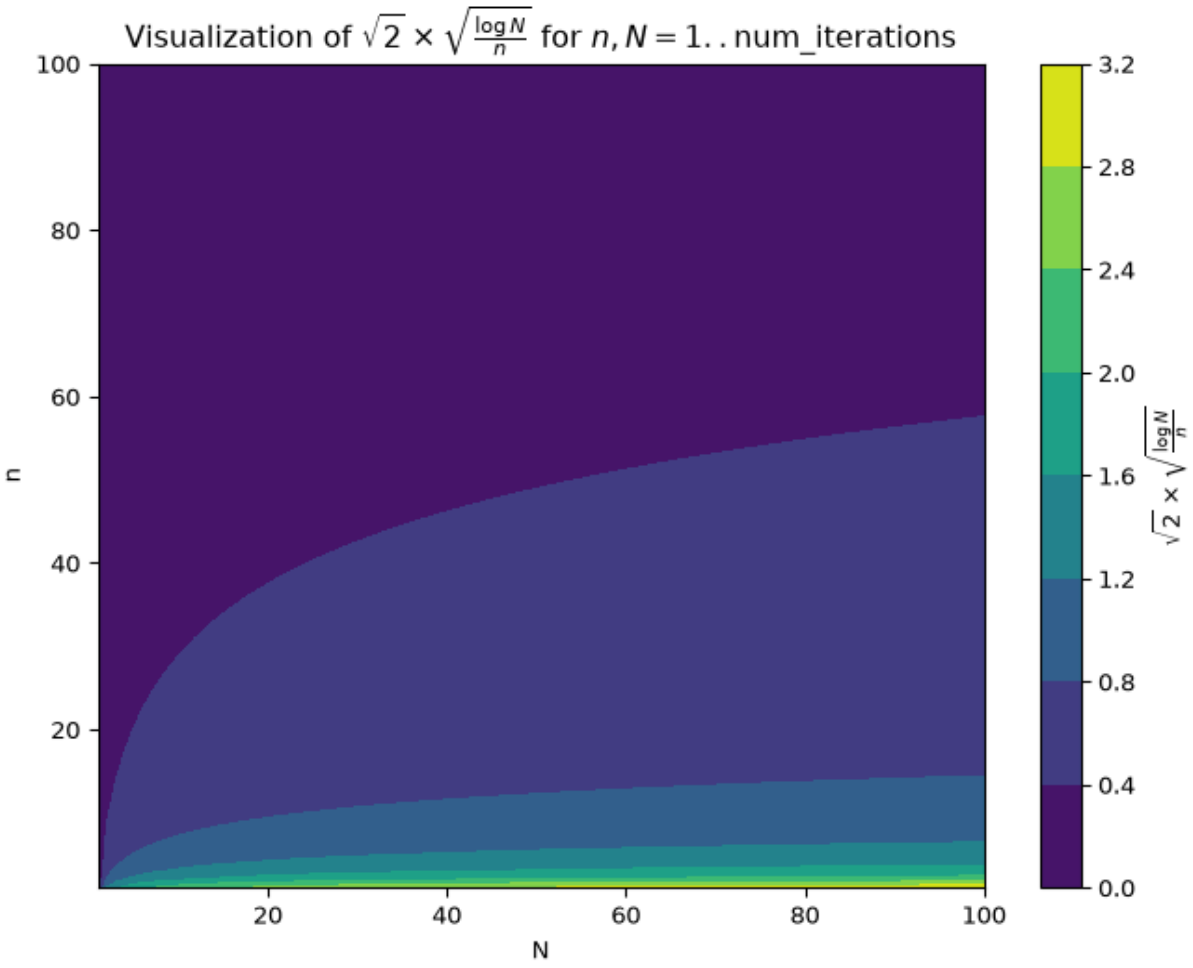


Summary

In [3]:

```python
import numpy as np
import matplotlib.pyplot as plt

num_iterations = 100

# Define the range for n and N
n_values = np.arange(1, num_iterations + 1)
N_values = np.arange(1, num_iterations + 1)

# Prepare a meshgrid for n and N
N, n = np.meshgrid(N_values, n_values)

# Compute the expression for each pair (n, N)
Z = np.sqrt(2) * np.sqrt(np.log(N) / n)

# Plotting
plt.figure(figsize=(8, 6))
plt.contourf(N, n, Z, cmap='viridis')
plt.colorbar(label=r'$\sqrt{2} \times \sqrt{\frac{\log{N}}{n}}$')
plt.xlabel('N')
plt.ylabel('n')
```

```
plt.title(r'Visualization of $\sqrt{2} \times \sqrt{\frac{\log{N}}{n}}$ for
plt.show()
```



Origin of UCB1 in MCTS:

The **Upper Confidence Bound 1 (UCB1)** formula originates from the **multi-armed bandit problem**, a classic problem in reinforcement learning and decision theory. In this problem, a gambler must decide which arm of multiple slot machines to pull to maximize their total reward, balancing the exploration of less-known machines and the exploitation of machines known to provide high rewards.

The UCB1 algorithm was developed to address this exploration-exploitation dilemma by providing a statistical upper bound on the expected reward of each action (or arm). In the context of **Monte Carlo Tree Search (MCTS)**, UCB1 is adapted to guide the selection of nodes during the **Selection** phase, helping the algorithm decide which node to explore next.

Understanding the UCB1 Formula:

The UCB1 formula used in MCTS is:

$$\text{UCB1}(i) = \overline{V}_i + C\sqrt{\frac{\ln N}{n_i}}$$

- $\overline{V}_i$: The average reward (value) of node i (exploitation term).
- C: A constant parameter that balances exploration and exploitation (commonly set to $\sqrt{2}$).
- N: The total number of simulations or visits to the parent node.
- n_i: The number of times node i has been visited.

Components Explained:

1. **Exploitation Term ($\overline{V}_i$)**: Represents the average reward obtained from node i, encouraging the selection of nodes with higher known rewards.

2. **Exploration Term ($C\sqrt{\frac{\ln N}{n_i}}$)**: Provides a bonus to nodes that have been visited less frequently, encouraging the exploration of less-visited nodes.

Balancing Exploration and Exploitation:

- **Exploitation**: Favors nodes with high average rewards.
- **Exploration**: Favors nodes that have been visited less, to gather more information.

The **exploration term decreases as n_i increases**, meaning that as a node is visited more often, the incentive to explore it further diminishes. Conversely, nodes with fewer visits receive a higher exploration bonus.

Why Exploration Occurs When Average Scores Are Similar:

Here's why:

- **Similar Average Rewards ($\overline{V}_i$)**: When nodes have comparable exploitation values, the exploration term becomes the deciding factor in the UCB1 value.

- **Influence of the Exploration Term**:

  - **Less-Visited Nodes**: Have a higher exploration term due to smaller n_i, increasing their UCB1 value.
  - **Well-Visited Nodes**: Have a lower exploration term, as n_i is larger.
- **Result**: The algorithm is more likely to select less-explored nodes when the average rewards are similar, promoting exploration to potentially discover better outcomes.

Mathematical Insight:

- When $\overline{V}_i$ **values are equal**, the UCB1 formula simplifies to comparing the exploration terms.

- The node with the **smaller** n_i (less visited) will have a larger exploration term due to the $\frac{1}{\sqrt{n_i}}$ relationship.
- As N increases (more total simulations), the exploration bonus diminishes logarithmically, ensuring that the algorithm eventually favors exploitation.

Visualizing the Exploration Term:

To further understand how exploration is encouraged:

- **Exploration Term Behavior**:
  - **Early Stages (n_i is small)**: Exploration term is significant, promoting the exploration of all nodes.
  - **Later Stages (n_i increases)**: Exploration term decreases, and nodes with higher average rewards are preferred.
- **Logarithmic Growth**:
  - The $\ln N$ term grows slowly, meaning that the exploration bonus reduces over time unless n_i remains low.

Conclusion:

The UCB1 formula in MCTS effectively balances exploration and exploitation by:

- **Using the average reward** to exploit known good nodes.
- **Incorporating the exploration term** to ensure that less-visited nodes are explored, especially when their average rewards are similar to others.
- **Adapting over time**, so the algorithm initially explores widely but gradually focuses on the most promising nodes as more information is gathered.

In summary, the UCB1 formula originates from the need to solve the exploration-exploitation dilemma in the multi-armed bandit problem and is integral to MCTS's ability to efficiently search large decision spaces. The exploration primarily occurs when nodes have similar average scores because the exploration term then plays a crucial role in differentiating between them, guiding the algorithm to potentially unexplored but promising areas of the search space.

Additional Resources:

- **Research Papers**:
  - Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). *Finite-time Analysis of the Multiarmed Bandit Problem*. Machine Learning.
  - Kocsis, L., & Szepesvári, C. (2006). *Bandit Based Monte-Carlo Planning*. ECML.
- **Further Reading**:
  - Monte Carlo Tree Search Tutorial
  - Understanding UCB1 and MCTS

Core Game Framework

Game

In [4]:
```python
import math
import random
import numpy as np
import matplotlib.pyplot as plt

class Game:

    """
    Abstract interface for a deterministic, 2-player, zero-sum,
    turn-taking game.

    Conventions (used by Tic-Tac-Toe and the solvers below):
    - Players are identified by strings "X" and "O".
    - evaluate(state) returns:
        > 0  if the position is good for "X"
        < 0  if the position is good for "O"
        == 0 for a draw or non-terminal equal position
    """

    def initial_state(self):

        """Return an object representing the starting position of the game."

        raise NotImplementedError

    def get_valid_moves(self, state):

        """
        Given a state, return an iterable of legal moves.
        The type of 'move' is game-dependent (e.g., (row, col) for Tic-Tac-T
        """

        raise NotImplementedError

    def make_move(self, state, move, player):

        """
        Return the successor state obtained by applying 'move' for 'player'
        to 'state'. The original state should not be modified in-place.
        """

        raise NotImplementedError

    def get_opponent(self, player):

        """Return the opponent of 'player'."""

        raise NotImplementedError
```

```
    def is_terminal(self, state):

        """
        Return True if 'state' is a terminal position (win, loss, or draw),
        False otherwise.
        """

        raise NotImplementedError

    def evaluate(self, state):

        """
        Return a scalar evaluation of 'state':
            +1 for X win, -1 for O win, 0 otherwise (for Tic-Tac-Toe).
        For other games this may be generalized, but here we keep it simple.
        """

        raise NotImplementedError

    def display(self, state):

        """Print a human-readable representation of 'state' (for debugging).

        raise NotImplementedError
```

TicTacToe

In [5]:
```
class TicTacToe(Game):

    """
    Classic 3x3 Tic-Tac-Toe implementation using a NumPy array of strings.
    Empty squares are represented by " ".
    Player "X" is assumed to be the maximizing player.
    """

    def __init__(self):
        self.size = 3

    def initial_state(self):

        """Return an empty 3x3 board."""

        return np.full((self.size, self.size), " ")

    def get_valid_moves(self, state):

        """All (i, j) pairs where the board cell is empty."""

        return [
            (i, j)
            for i in range(self.size)
            for j in range(self.size)
            if state[i, j] == " "
        ]
```

```
    def make_move(self, state, move, player):

        """
        Return a new board with 'player' placed at 'move' (row, col).
        The original state is not modified.
        """

        new_state = state.copy()
        new_state[move] = player
        return new_state

    def get_opponent(self, player):

        """Swap player labels between 'X' and 'O'."""

        return "O" if player == "X" else "X"

    def is_terminal(self, state):

        """
        A state is terminal if:
        - Either player has a 3-in-a-row (evaluate != 0), or
        - There are no empty squares left (draw).
        """

        if self.evaluate(state) != 0:
            return True
        return " " not in state

    def evaluate(self, state):

        """
        Return +1 if X has three in a row, -1 if O has three in a row,
        and 0 otherwise (including non-terminal states and draws).

        This is a "game-theoretic" evaluation at terminal states; for
        non-terminal positions we simply return 0.
        """

        lines = []

        # Rows and columns
        for i in range(self.size):
            lines.append(state[i, :])   # row i
            lines.append(state[:, i])   # column i

        # Main diagonals
        lines.append(np.diag(state))
        lines.append(np.diag(np.fliplr(state)))

        # Check each line for a win
        for line in lines:
            if np.all(line == "X"):
                return 1
            if np.all(line == "O"):
```

```
                return -1
        return 0

    def display(self, state):

        """
        Visualize a Tic-Tac-Toe board using matplotlib.

        Parameters
        ----------
        state : np.ndarray of shape (size, size)
            Board containing ' ', 'X', or 'O'.
        """

        size = self.size

        fig, ax = plt.subplots()
        ax.set_aspect('equal')
        ax.set_xlim(0, size)
        ax.set_ylim(0, size)

        # Draw grid lines
        for i in range(1, size):
            ax.axhline(i, color='black')
            ax.axvline(i, color='black')

        # Hide axes completely
        ax.axis('off')

        # Draw X and O symbols
        for i in range(size):
            for j in range(size):
                cx = j + 0.5
                cy = size - i - 0.5     # invert y-axis for correct row orie

                symbol = state[i, j]

                if symbol == "X":
                    ax.plot(cx, cy, marker='x',
                            markersize=40 * (3/size),
                            color='blue',
                            markeredgewidth=3)
                elif symbol == "O":
                    circle = plt.Circle((cx, cy),
                                        radius=0.30 * (3/size),
                                        fill=False,
                                        color='red',
                                        linewidth=3)
                    ax.add_patch(circle)

        plt.show()
```

Solver

In [6]:
```python
class Solver:

    """
    Base class for all solvers (Random, Minimax, AlphaBeta, MCTS, etc.).

    Solvers must implement:
        - select_move(game, state, player)

    Solvers may optionally implement:
        - reset()           : called at the start of each game
        - opponent_played() : used by persistent solvers (e.g., MCTS)

    Notes
    -----
    • Solvers may keep internal state that persists across moves.
    • GameRunner may call reset() automatically before every match.
    """

    def select_move(self, game, state, player):

        """
        Must be implemented by subclasses.
        Returns a legal move for the given player.
        """

        raise NotImplementedError

    def get_name(self):

        """
        Return the solver's name for reporting, logging, or tournament resul

        The default returns the class name, but solvers may override
        to include parameters (e.g., "MCTS(num_simulations=500)").
        """

        return self.__class__.__name__

    def opponent_played(self, move):
        """
        Optional. Called after the opponent moves.
        Useful for stateful solvers like MCTS.
        Stateless solvers can ignore it.
        """
        pass

    def reset(self):

        """
        Optional. Called once at the beginning of each game.
        Override only if the solver maintains internal state
        (e.g., MCTS tree, cached analysis, heuristic tables).
        """

        pass
```

RandomSolver

In [7]:
```python
class RandomSolver(Solver):

    """
    A simple baseline solver:
    - At each move, chooses uniformly at random among all legal moves.
    - Does not maintain any internal state (no learning).
    """

    def __init__(self, seed=None):
        self.rng = random.Random(seed)

    def select_move(self, game, state, player):

        """Return a random legal move for the current player."""

        moves = game.get_valid_moves(state)

        return self.rng.choice(moves)

    def opponent_played(self, move):

        """Random solver has no internal state to update."""

        pass
```

GameRunner

In [8]:
```python
class GameRunner:

    """
    Utility to run a single game between two solvers on a given Game.

    This class is deliberately simple: it alternates moves between "X" and "
    until a terminal state is reached.
    """

    def __init__(self, game, verbose=False):
        self.game = game
        self.verbose = verbose

    def play_game(self, solver_X, solver_O):

        """
        Play one full game:
        - solver_X controls player "X"
        - solver_O controls player "O"

        Returns
        -------
```

```
        result : int
            +1 if X wins, -1 if O wins, 0 for a draw.
        """

        state = self.game.initial_state()
        player = "X"
        solvers = {"X": solver_X, "O": solver_O}

        # Play until terminal position
        while not self.game.is_terminal(state):
            # Current player selects a move
            move = solvers[player].select_move(self.game, state, player)

            # Apply the move
            state = self.game.make_move(state, move, player)

            if self.verbose:
                self.game.display(state)

            # Notify the opponent (for persistent solvers like MCTS)
            opp = self.game.get_opponent(player)
            solvers[opp].opponent_played(move)

            # Switch active player
            player = opp

        if self.verbose:
            print(self.game.evaluate(state), "\n")

        # Final evaluation from X's perspective
        return self.game.evaluate(state)
```

evaluate_solvers

In [9]:

```
def evaluate_solvers(game, solver_X, solver_O, num_games, verbose=False):

    """
    Evaluate two solvers head-to-head on a given game.

    Parameters
    ----------
    game      : Game
        An instance of a Game (e.g., TicTacToe).
    solver_X  : Solver
        Solver controlling player "X" (the maximizing player).
    solver_O  : Solver
        Solver controlling player "O" (the minimizing player).
    num_games : int
        Number of games to play with these fixed roles.

    Notes
    -----
    - The same solver instances are reused across games.
      This allows *persistent* solvers (e.g., MCTS) to accumulate
```

```
      experience across games.
    - Outcomes are interpreted from X's perspective:
        +1 -> X wins
        -1 -> O wins
         0 -> draw
    """

    runner = GameRunner(game)

    # Aggregate statistics over all games
    results = {
        "X_wins": 0,
        "O_wins": 0,
        "draws": 0,
    }

    for i in range(num_games):
        # Play one game with solver_X as "X" and solver_O as "O"
        outcome = runner.play_game(solver_X, solver_O)

        # Update counters based on outcome (+1, -1, or 0)
        if outcome == 1:
            results["X_wins"] += 1
            if verbose:
                print(f"Game {i + 1}: X wins")
        elif outcome == -1:
            results["O_wins"] += 1
            if verbose:
                print(f"Game {i + 1}: O wins")
        else:
            results["draws"] += 1
            if verbose:
                print(f"Game {i + 1}: Draw")

    # Print final summary
    if verbose:
        print(f"\nAfter {num_games} games:")
        print(f"  X ({solver_X.get_name()}) wins: {results['X_wins']}")
        print(f"  O ({solver_O.get_name()}) wins: {results['O_wins']}")
        print(f"  Draws: {results['draws']}")

    return results
```

MinimaxSolver

In [10]:
```
from functools import lru_cache

def canonical(state):

    """
    Convert a NumPy array board into a hashable, immutable representation
    (tuple of tuples). This allows us to use it as a key in dicts or
    as an argument to lru_cache. MCTS can also reuse this representation.
    """
```

```
    return tuple(map(tuple, state))

class MinimaxSolver(Solver):

    """
    A classic, exact Minimax solver for Tic-Tac-Toe.

    - Assumes "X" is the maximizing player.
    - Uses memoization (lru_cache) to avoid recomputing values for
      identical positions.
    """

    def select_move(self, game, state, player):

        """
        Public interface: choose the best move for 'player' using Minimax.
        For Tic-Tac-Toe we can safely search the full game tree.
        """

        # Store game on self so _minimax can use it
        self.game = game

        # From X's perspective: X is maximizing, O is minimizing
        maximizing = (player == "X")

        # For Tic-Tac-Toe, depth=9 is enough to cover all remaining moves.
        _, move = self._minimax(canonical(state), player, maximizing, 9)
        return move

    @lru_cache(maxsize=None)
    def _minimax(self, state_key, player, maximizing, depth):

        """
        Internal recursive minimax.

        Parameters
        ----------
        state_key : hashable representation of the board (tuple of tuples)
        player    : player to move at this node ("X" or "O")
        maximizing: True if this node is a 'max' node (X to move),
                    False if this is a 'min' node (O to move)
        depth     : remaining search depth (not used for cutoffs in this
                    full-search Tic-Tac-Toe implementation, but kept for
                    didactic purposes and easy extension).
        """

        # Recover the NumPy board from the canonical state_key
        state = np.array(state_key)

        # Terminal test: win, loss, or draw
        if self.game.is_terminal(state):
            # Evaluation is always from X's perspective: +1, -1, or 0
            return self.game.evaluate(state), None

        moves = self.game.get_valid_moves(state)
```

```
        best_move = None

        if maximizing:
            # X to move: maximize the evaluation
            best_val = -math.inf
            for move in moves:
                st2 = self.game.make_move(state, move, player)
                val, _ = self._minimax(
                    canonical(st2),
                    self.game.get_opponent(player),
                    False,
                    depth - 1
                )
                if val > best_val:
                    best_val = val
                    best_move = move
            return best_val, best_move

        else:
            # O to move: minimize the evaluation (since evaluation is for X)
            best_val = math.inf
            for move in moves:
                st2 = self.game.make_move(state, move, player)
                val, _ = self._minimax(
                    canonical(st2),
                    self.game.get_opponent(player),
                    True,
                    depth - 1
                )
                if val < best_val:
                    best_val = val
                    best_move = move
            return best_val, best_move
```

MinimaxAlphaBetaSolver

In [11]:

```
class MinimaxAlphaBetaSolver(Solver):

    """
    A classical Minimax solver enhanced with Alpha-Beta pruning.

    - Assumes "X" is the maximizing player.
    - Uses memoization (lru_cache) to avoid recomputing states.
    - Performs a *full* search of Tic-Tac-Toe (depth=9).
    - Returns the optimal move for the current player.
    """

    # ------------------------------------------------------------
    # Solver interface
    # ------------------------------------------------------------

    def select_move(self, game, state, player):

        """
```

```
        Public interface required by Solver.
        Runs Alpha-Beta search from the current state.
        """

        self.game = game

        maximizing = (player == "X")   # X maximizes, O minimizes

        # Reset cache between games to avoid storing millions of keys
        self._alphabeta.cache_clear()

        value, move = self._alphabeta(
            canonical(state),
            player,
            maximizing,
            9,               # full-depth search
            -math.inf,       # alpha
            math.inf         # beta
        )
        return move

    # ------------------------------------------------------------
    # Internal alpha-beta with memoization
    # ------------------------------------------------------------

    @lru_cache(maxsize=None)
    def _alphabeta(self, state_key, player, maximizing, depth, alpha, beta):

        """
        Parameters
        ----------
        state_key : tuple-of-tuples board
        player    : player whose turn it is ('X' or 'O')
        maximizing: True if this node is a maximizing node for X
        depth     : remaining depth
        alpha     : best guaranteed value for maximizer so far
        beta      : best guaranteed value for minimizer so far
        """

        state = np.array(state_key)

        # Terminal or horizon case
        if self.game.is_terminal(state) or depth == 0:
            return self.game.evaluate(state), None

        moves = self.game.get_valid_moves(state)
        best_move = None

        # ------------------------------------------------------------
        # MAX (X)
        # ------------------------------------------------------------

        if maximizing:
            value = -math.inf

            for move in moves:
```

```
                st2 = self.game.make_move(state, move, player)

                child_val, _ = self._alphabeta(
                    canonical(st2),
                    self.game.get_opponent(player),
                    False,              # now minimizing
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val > value:
                    value = child_val
                    best_move = move

                alpha = max(alpha, value)
                if beta <= alpha:
                    break  # β-cutoff

            return value, best_move

        # ------------------------------------------------------
        # MIN (O)
        # ------------------------------------------------------

        else:
            value = math.inf

            for move in moves:
                st2 = self.game.make_move(state, move, player)

                child_val, _ = self._alphabeta(
                    canonical(st2),
                    self.game.get_opponent(player),
                    True,               # now maximizing
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val < value:
                    value = child_val
                    best_move = move

                beta = min(beta, value)
                if beta <= alpha:
                    break  # α-cutoff

            return value, best_move
```

Sanity Check

In [12]:
```
game = TicTacToe()
```

```
a = RandomSolver(7)
b = MinimaxSolver()

results = evaluate_solvers(game, a, b, num_games=100)

results
```

```
{'X_wins': 0, 'O_wins': 82, 'draws': 18}
```

Sanity Check

In [13]:
```
game = TicTacToe()

a = RandomSolver(7)
b = MinimaxAlphaBetaSolver()

results = evaluate_solvers(game, a, b, num_games=100)

results
```

```
{'X_wins': 0, 'O_wins': 82, 'draws': 18}
```

Implementation

MCTSClassicSolver

In [14]:
```
class MCTSClassicSolver(Solver):

    """
    A textbook, first-contact implementation of Monte Carlo Tree Search (MCT
    for deterministic 2-player zero-sum games (e.g., Tic-Tac-Toe).

    Key ideas:
      - For each decision, we build a tree rooted at the current position.
      - Each node stores:
          * state: board position
          * player: player to move in this state ("X" or "O")
          * N: visit count
          * W: total reward from this node player's perspective
          * children: move -> child Node
          * untried_moves: list of legal moves not yet expanded
          * parent: link to the parent node (for backpropagation)
      - One MCTS *simulation* = selection → expansion → simulation (rollout)
      - We throw away the tree after returning a move (no learning).
    """

    class Node:

        """A single node in the MCTS search tree."""

        def __init__(self, state, player, parent=None, moves=None):
            self.state = state            # board position (NumPy array)
```

```
            self.player = player          # player to move in this state
            self.parent = parent          # parent Node (None for root)
            self.children = {}            # move -> child Node
            self.untried_moves = list(moves) if moves is not None else []
            self.N = 0                    # visit count
            self.W = 0.0                  # total reward (this player's pers

    def __init__(self, num_simulations=500, exploration_c=math.sqrt(2), seed

        """
        Parameters
        ----------
        num_simulations : int
            Number of simulations (playouts) to run per move.
        exploration_c : float
            Exploration constant C in the UCT formula.
        seed : int or None
            Optional random seed for reproducibility.
        """

        self.num_simulations = num_simulations
        self.exploration_c = exploration_c
        self.rng = random.Random(seed)

        self.game = None
        self.root = None  # root Node for the current search

    # ------------------------------------------------------------------
    # Public Solver interface
    # ------------------------------------------------------------------

    def select_move(self, game, state, player):

        """
        Choose a move for 'player' in 'state' using classic MCTS.

        A new tree is built from scratch for this call. The tree is not
        reused for later moves or games.
        """

        self.game = game

        self.root = None  # root Node for the current search

        # Create the root node for the current position.
        root_state = state.copy()
        root_moves = self.game.get_valid_moves(root_state)
        self.root = self.Node(root_state, player, parent=None, moves=root_mo

        # Run multiple simulations starting from the root.
        for _ in range(self.num_simulations):
            self._run_simulation()

        # After simulations, choose the child with the largest visit count.
        if not self.root.children:
            # No children: no legal moves (terminal). Fall back to random if
```

```
            moves = self.game.get_valid_moves(self.root.state)
            return self.rng.choice(moves) if moves else None

        best_move = None
        best_visits = -1
        for move, child in self.root.children.items():
            if child.N > best_visits:
                best_visits = child.N
                best_move = move

        return best_move

    def opponent_played(self, move):

        """
        Classic MCTS here is stateless across moves and games:
        we rebuild the tree for every decision.

        So we do not need to track the opponent's move.
        """

        pass

    # ------------------------------------------------------------
    # Internal MCTS steps
    # ------------------------------------------------------------

    def _run_simulation(self):

        """
        Perform one MCTS simulation from the root.

        1. Selection: descend the tree using UCT until we reach a node
           that is terminal or has untried moves.
        2. Expansion: if the node is non-terminal and has untried moves,
           expand one child.
        3. Simulation (rollout): from the new child, play random moves
           to the end of the game.
        4. Backpropagation: update N and W along the path with the outcome.
        """

        node = self.root

        # 1. SELECTION: descend while fully expanded and non-terminal.
        while True:

            if self.game.is_terminal(node.state):
                # Terminal position: evaluate immediately.
                outcome = self.game.evaluate(node.state)  # from X's perspec
                self._backpropagate(node, outcome)
                return

            if node.untried_moves:

                # 2. EXPANSION: choose one untried move and create a child n
                move = self.rng.choice(node.untried_moves)
```

```
                node.untried_moves.remove(move)

                next_state = self.game.make_move(node.state, move, node.play
                next_player = self.game.get_opponent(node.player)
                next_moves = self.game.get_valid_moves(next_state)

                child = self.Node(next_state, next_player, parent=node, move
                node.children[move] = child

                # 3. SIMULATION: rollout from the newly created child.
                outcome = self._rollout(child.state, child.player)

                # 4. BACKPROPAGATION: update all nodes on the path from chil
                self._backpropagate(child, outcome)

                return

            # Node is fully expanded and non-terminal → choose a child by UC
            node = self._select_child(node)

    def _select_child(self, node):

        """
        UCT selection: for each child

            V_parent(child) = - (child.W / child.N)
            UCT = V_parent(child) + C * sqrt( ln(N_parent + 1) / N_child )

        We store W and N from the child's own perspective, so we negate
        child.W / child.N to get the parent's perspective.
        """

        parent_visits = node.N
        best_score = -math.inf
        best_child = None

        for move, child in node.children.items():
            if child.N == 0:
                score = math.inf  # always explore unvisited children at lea
            else:
                # Average reward from the child's own perspective.
                avg_child = child.W / child.N

                # Parent and child players alternate; reward from parent per
                # is the negative of the child's perspective.
                reward_parent = -avg_child

                exploration = self.exploration_c * math.sqrt(
                    math.log(parent_visits + 1) / child.N
                )

                score = reward_parent + exploration

            if score > best_score:
                best_score = score
                best_child = child
```

```
        return best_child

    def _rollout(self, state, player_to_move):

        """
        Random playout from 'state' until the game ends.

        Returns the final result from X's perspective:
          +1 if X wins, -1 if O wins, 0 for draw.
        """

        current_state = state.copy()
        current_player = player_to_move

        while not self.game.is_terminal(current_state):
            moves = self.game.get_valid_moves(current_state)
            move = self.rng.choice(moves)
            current_state = self.game.make_move(current_state, move, current
            current_player = self.game.get_opponent(current_player)

        return self.game.evaluate(current_state)

    def _backpropagate(self, node, outcome):

        """
        Backpropagate the simulation outcome up the tree.

        outcome is always from X's perspective: +1, -1, or 0.

        For each node on the path from 'node' up to the root:
          - Convert outcome to that node's player's perspective:
              reward = outcome   if node.player == "X"
                     = -outcome  if node.player == "O"
          - Update:
              node.N += 1
              node.W += reward
        """

        current = node
        while current is not None:
            if current.player == "X":
                reward = outcome
            else:
                reward = -outcome

            current.N += 1
            current.W += reward

            current = current.parent
```

Node

In [15]:
```python
class MCTSClassicSolver(Solver):

    class Node:

        def __init__(self, state, player, parent=None, moves=None):
            self.state = state            # board position (NumPy array)
            self.player = player          # player to move in this state
            self.parent = parent          # parent Node (None for root)
            self.children = {}            # move -> child Node
            self.untried_moves = list(moves) if moves is not None else []
            self.N = 0                    # visit count
            self.W = 0.0                  # total reward (this player's pers
```

`MCTSClassicSolver` is_a `Solver`.

Uses `Node` to explicitely build its search tree.

`__init__`

In [16]:
```python
    def __init__(self, num_simulations=500, exploration_c=math.sqrt(2), seed

        self.num_simulations = num_simulations
        self.exploration_c = exploration_c
        self.rng = random.Random(seed)

        self.game = None
        self.root = None
```

Public Solver interface

In [17]:
```python
    def select_move(self, game, state, player):

        self.game = game
        self.root = None  # building a new tree for each call

        # Create the root node for the current position.
        root_state = state.copy()
        root_moves = self.game.get_valid_moves(root_state)
        self.root = self.Node(root_state, player, parent=None, moves=root_mo

        # Run multiple simulations starting from the root.
        for _ in range(self.num_simulations):
            self._run_simulation()

        best_move = None
        best_visits = -1
        for move, child in self.root.children.items():
            if child.N > best_visits:
                best_visits = child.N
                best_move = move
```

```
        return best_move
```

Public Solver interface

In [18]:
```
    def opponent_played(self, move):
        pass
```

`_run_simulation`

In [19]:
```
    def _run_simulation(self):

        node = self.root

        # 1. SELECTION: descend while fully expanded and non-terminal.
        while True:

            if self.game.is_terminal(node.state):
                # Terminal position: evaluate immediately.
                outcome = self.game.evaluate(node.state)  # from X's perspec
                self._backpropagate(node, outcome)
                return

            if node.untried_moves:

                # 2. EXPANSION: choose one untried move and create a child n
                move = self.rng.choice(node.untried_moves)
                node.untried_moves.remove(move)

                next_state = self.game.make_move(node.state, move, node.play
                next_player = self.game.get_opponent(node.player)
                next_moves = self.game.get_valid_moves(next_state)

                child = self.Node(next_state, next_player, parent=node, move
                node.children[move] = child

                # 3. SIMULATION: rollout from the newly created child.
                outcome = self._rollout(child.state, child.player)

                # 4. BACKPROPAGATION: update all nodes on the path from chil
                self._backpropagate(child, outcome)

                return

            # Node is fully expanded and non-terminal → choose a child by UC
            node = self._select_child(node)
```

`_select_child`

In [20]:
```
    def _select_child(self, node):

        parent_visits = node.N
        best_score = -math.inf
        best_child = None

        for move, child in node.children.items():
            if child.N == 0:
                score = math.inf  # always explore unvisited children at lea
            else:
                # Average reward from the child's own perspective.
                avg_child = child.W / child.N

                # Parent and child players alternate; reward from parent per
                # is the negative of the child's perspective.
                reward_parent = -avg_child

                exploration = self.exploration_c * math.sqrt(
                    math.log(parent_visits + 1) / child.N
                )

                score = reward_parent + exploration

            if score > best_score:
                best_score = score
                best_child = child

        return best_child
```

_rollout

In [21]:
```
    def _rollout(self, state, player_to_move):

        current_state = state.copy()
        current_player = player_to_move

        while not self.game.is_terminal(current_state):
            moves = self.game.get_valid_moves(current_state)
            move = self.rng.choice(moves)
            current_state = self.game.make_move(current_state, move, current
            current_player = self.game.get_opponent(current_player)

        return self.game.evaluate(current_state)
```

_backpropagate

In [22]:
```
    def _backpropagate(self, node, outcome):

        current = node
        while current is not None:
            if current.player == "X":
```

```
                reward = outcome
            else:
                reward = -outcome

            current.N += 1
            current.W += reward

            current = current.parent
```

visualize_tree

In [25]:

```
from graphviz import Digraph

def visualize_tree(root, max_depth=3, show_mcts_stats=True, show_edge_labels

    """
    Visualize a game tree rooted at `root` using Graphviz.

    Assumes:
      - `root` is a Node with attributes:
          state, player, children: dict[move -> Node], N, W.
      - This matches the Node used in MCTSClassicSolver.

    Parameters
    ----------
    root : Node
        Root of the (sub)tree to visualize.
    max_depth : int
        Maximum depth to recurse (root at depth 0).
    show_mcts_stats : bool
        If True, include N and V for each node (compact vertical layout).
    show_edge_labels : bool
        If True, label edges with the move (e.g., (row, col)).
    """

    dot = Digraph(format="png")

    dot.edge_attr.update(
        fontsize="8",
        fontname="Helvetica"
    )

    # Make the tree compact

    dot.graph_attr.update(
        rankdir="TB",   # top-to-bottom
        nodesep="0.15", # horizontal spacing
        ranksep="0.50", # vertical spacing
    )
    dot.node_attr.update(
        shape="box",
        fontsize="9",
        fontname="Helvetica",
        margin="0.02,0.02",
```

```
    )

    def add_node(node, node_id, depth):
        if depth > max_depth:
            return

        # Build compact label
        if show_mcts_stats and node.N > 0:
            V = node.W / node.N
            # player on top, then N, then V (vertical)
            label = f"{node.player}\\nN={node.N}\\nV={V:.2f}"
        else:
            label = f"{node.player}"

        dot.node(node_id, label=label)

        # Recurse on children
        if depth == max_depth:
            return

        for move, child in node.children.items():
            child_id = f"{id(child)}"
            if show_edge_labels:
                dot.edge(node_id, child_id, label=str(move))
            else:
                dot.edge(node_id, child_id)
            add_node(child, child_id, depth + 1)

    add_node(root, "root", depth=0)

    return dot
```

Search Tree (`num_simulations=10`)

In [26]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"

solver = MCTSClassicSolver(num_simulations=10, seed=4)

move = solver.select_move(game, state, player)

dot = visualize_tree(solver.root, 9, True)
print(move)
dot
```

```
(1, 0)
```


No description has been provided for this image

Search Tree (`num_simulations=500`)

In [27]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"

solver = MCTSClassicSolver(num_simulations=500, seed=4)

move = solver.select_move(game, state, player)

dot = visualize_tree(solver.root, 9, True)
print(move)
dot
```

```
(1, 1)
```



Search Tree (`exploration_c=math.sqrt(2)/2`)

In [28]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"

solver = MCTSClassicSolver(num_simulations=500, exploration_c=math.sqrt(2)/2

move = solver.select_move(game, state, player)

dot = visualize_tree(solver.root, 9, True)
print(move)
dot
```

```
(1, 1)
```



Search Tree (`exploration_c=0.1`)

In [29]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"

solver = MCTSClassicSolver(num_simulations=500, exploration_c=1e-09, seed=4)

move = solver.select_move(game, state, player)

dot = visualize_tree(solver.root, 9, True)
print(move)
dot
```

```
(2, 2)
```



Search Tree (`nexploration_c=1000`)

In [30]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"

solver = MCTSClassicSolver(num_simulations=500, exploration_c=1000, seed=4)

move = solver.select_move(game, state, player)

dot = visualize_tree(solver.root, 9, True)
print(move)
dot
```

```
(2, 2)
```



Search Tree (`num_simulations=500`)

In [31]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"

solver = MCTSClassicSolver(num_simulations=500, seed=4)

move = solver.select_move(game, state, player)

dot = visualize_tree(solver.root, 2, True)
print(move)
dot
```

```
(1, 1)
```

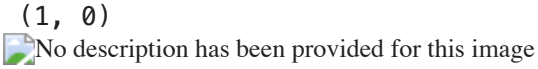


Visualizing only two layers of the tree (depth = 2).

Search Tree (`num_simulations=10`)

In [32]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"

solver = MCTSClassicSolver(num_simulations=10, seed=4)
move = solver.select_move(game, state, player)
dot = visualize_tree(solver.root, 1, True)
print(move)
dot
```

```
(1, 0)
```



Visualizing only one layer of the tree (depth = 1).

As the number of simulations increases, both the number of nodes and the tree's depth proportionally expand. However the tree complexity, action selection is based only on the immediate descendants of the root node.

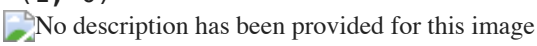
Why increasing the number of simulations then?

Augmenting the number of simulations enhances our confidence in the decision-making process.

Search Tree (`num_simulations=50`)

In [33]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"

solver = MCTSClassicSolver(num_simulations=50, seed=4)
move = solver.select_move(game, state, player)
dot = visualize_tree(solver.root, 1, True)
print(move)
dot
```

```
(1, 0)
```



Search Tree (`num_simulations=250`)

In [34]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"

solver = MCTSClassicSolver(num_simulations=250, seed=4)
move = solver.select_move(game, state, player)
dot = visualize_tree(solver.root, 1, True)
print(move)
dot
```

```
(2, 2)
```



Search Tree (`num_simulations=500`)

In [35]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"

solver = MCTSClassicSolver(num_simulations=500, seed=4)
move = solver.select_move(game, state, player)
```

```
dot = visualize_tree(solver.root, 1, True)
print(move)
dot
```

```
(1, 1)
```



Search Tree (`num_sims=1000`)

In [36]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"

solver = MCTSClassicSolver(num_simulations=1000, seed=4)
move = solver.select_move(game, state, player)
dot = visualize_tree(solver.root, 1, True)
print(move)
dot
```

```
(1, 1)
```



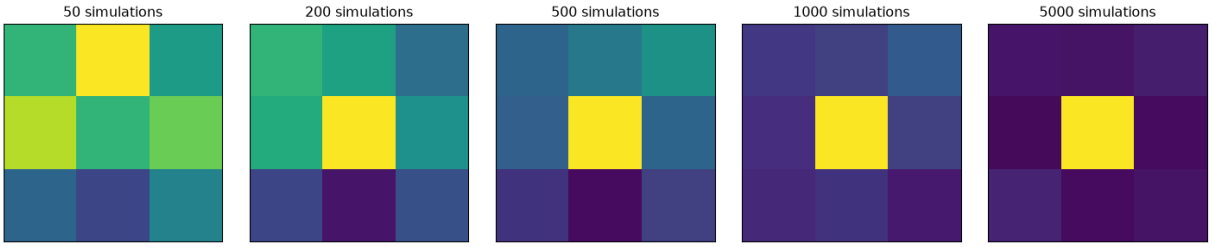
Move Preference vs `num_simulations`

In [37]:
```
def mcts_heatmaps(game, solver_class, simulations_list, player="X", seed=0):

    """
    Display heatmaps showing square visit frequencies
    for different MCTS simulation counts.

    Parameters
    ----------
    game : TicTacToe instance
    solver_class : a class such as MCTSClassicSolver
    simulations_list : list of ints (e.g. [50, 100, 200, 500, 1000])
    player : "X" or "O"
    """

    initial = game.initial_state()

    num_plots = len(simulations_list)
    fig, axes = plt.subplots(1, num_plots, figsize=(3 * num_plots, 3))

    if num_plots == 1:
        axes = [axes]  # normalize indexing

    for ax, sims in zip(axes, simulations_list):

        # ------------------------------------------------------
        # Run MCTS
        # ------------------------------------------------------

        solver = solver_class(num_simulations=sims, seed=seed)
```

```
        solver.reset()
        solver.select_move(game, initial, player)   # builds tree
        root = solver.root

        # ------------------------------------------------------------
        # Build a 3×3 matrix of visits
        # ------------------------------------------------------------

        visit_matrix = np.zeros((3, 3), dtype=float)

        for move, child in root.children.items():
            i, j = move
            visit_matrix[i, j] = child.N

        # ------------------------------------------------------------
        # Normalize (avoid division by zero)
        # ------------------------------------------------------------

        vmax = visit_matrix.max()
        if vmax > 0:
            heat = visit_matrix / vmax
        else:
            heat = visit_matrix

        # ------------------------------------------------------------
        # Plot heatmap
        # ------------------------------------------------------------

        im = ax.imshow(heat, cmap="viridis", vmin=0, vmax=1)
        ax.set_title(f"{sims} simulations")
        ax.set_xticks([])
        ax.set_yticks([])

    plt.tight_layout()
    plt.show()
    plt.close(fig)
```

In [38]:

```
game = TicTacToe()

mcts_heatmaps(
    game,
    solver_class=MCTSClassicSolver,
    simulations_list=[50, 200, 500, 1000, 5000],
    player="X",
)
```



`num_simulations` ∈ 50, 200, 500, 1000, 5000.

Tallies by X's first move

In [39]:

```
def tally_scores(game):
    """
    Enumerate all complete games of Tic-Tac-Toe from the initial position
    (X to move) and tally how many end in:

        - X win
        - draw
        - O win

    Returns
    -------
    overall : dict
        {'X': total_X_wins, 'draw': total_draws, 'O': total_O_wins}

    table : list[list[dict]]
        A 3x3 list of dicts. For each cell (i, j),
        table[i][j] = {'X': ..., 'draw': ..., 'O': ...}
        counts games where X's *first move* was at (i, j).
    """

    size = game.size  # should be 3 for standard Tic-Tac-Toe

    # Overall tallies for all games
    overall = {'X': 0, 'draw': 0, 'O': 0}

    # Per-first-move tallies as a 3x3 grid
    table = [
        [ {'X': 0, 'draw': 0, 'O': 0} for _ in range(size) ]
        for _ in range(size)
    ]

    def recurse(state, player, first_move):

        """
        Depth-first enumeration of all complete games.

        Parameters
        ----------
        state : board position (NumPy array)
        player : 'X' or 'O' (player to move)
        first_move : None, or (row, col) of X's very first move
        """

        # Base case: terminal state → classify outcome

        if game.is_terminal(state):
            v = game.evaluate(state)  # +1 (X win), -1 (O win), 0 (draw)

            if v > 0:
                outcome = 'X'
            elif v < 0:
                outcome = 'O'
```

```
            else:
                outcome = 'draw'

            # Update overall tally
            overall[outcome] += 1

            # If we know X's first move, update that cell's tally too
            if first_move is not None:
                i, j = first_move
                table[i][j][outcome] += 1

            return

        # Recursive case: expand all legal moves
        for move in game.get_valid_moves(state):
            next_state = game.make_move(state, move, player)
            next_player = game.get_opponent(player)

            # Record X's very first move
            if first_move is None and player == "X":
                fm = move  # this becomes the first_move for the rest of thi
            else:
                fm = first_move

            recurse(next_state, next_player, fm)

    # Start from the empty board, X to move, and no first_move yet
    initial_state = game.initial_state()
    recurse(initial_state, player="X", first_move=None)

    return overall, table

def print_tally_table(table):

    """
    Print a 3x3 table of tallies.

    Each cell shows: X:<wins> D:<draws> O:<wins>
    where counts are restricted to games where X's first move
    was played in that cell.
    """

    size = len(table)
    for i in range(size):
        row_cells = []
        for j in range(size):
            stats = table[i][j]
            cell_str = f"X:{stats['X']} D:{stats['draw']} O:{stats['O']}"
            row_cells.append(cell_str)
        print(" | ".join(row_cells))
    print()

def print_tally_table_percentages(table):

    """
    Print a 3x3 table of tallies.
```

```
    Each cell shows: X:<wins> D:<draws> O:<wins>
    where counts are restricted to games where X's first move
    was played in that cell.
    """

    size = len(table)
    for i in range(size):
        row_cells = []
        for j in range(size):
            stats = table[i][j]
            cell_str = f"X:{stats['X']/255168:.2%} D:{stats['draw']/255168:.
            row_cells.append(cell_str)
        print(" | ".join(row_cells))
    print()


game = TicTacToe()

overall, table = tally_scores(game)

print("Overall tally:")
print(overall)  # {'X': ..., 'draw': ..., 'O': ...}

print("\nTallies by X's first move (3x3 grid):")
print_tally_table(table)

print("\nTallies by X's first move (3x3 grid) as percentages:")
print_tally_table_percentages(table)
```

Overall tally:

{‘X’: 131184, ‘draw’: 46080, ‘O’: 77904}

{‘X’: 51.41%, ‘draw’: 18.06%, ‘O’: 30.53%}

Tallies by X’s first move (3x3 grid, X/draw/O):

14652/5184/7896	14232/5184/10176	14652/5184/7896
14232/5184/10176	15648/4608/5616	14232/5184/10176
14652/5184/7896	14232/5184/10176	14652/5184/7896

Tallies by X’s first move (3x3 grid, X/draw/O) as percentages:

5.74% / 2.03% / 3.09%	5.58% / 2.03% / 3.99%	5.74% / 2.03% / 3.09%
5.58% / 2.03% / 3.99%	6.13% / 1.81% / 2.20%	5.58% / 2.03% / 3.99%
5.74% / 2.03% / 3.09%	5.58% / 2.03% / 3.99%	5.74% / 2.03% / 3.09%

evaluate_solvers_with_plot

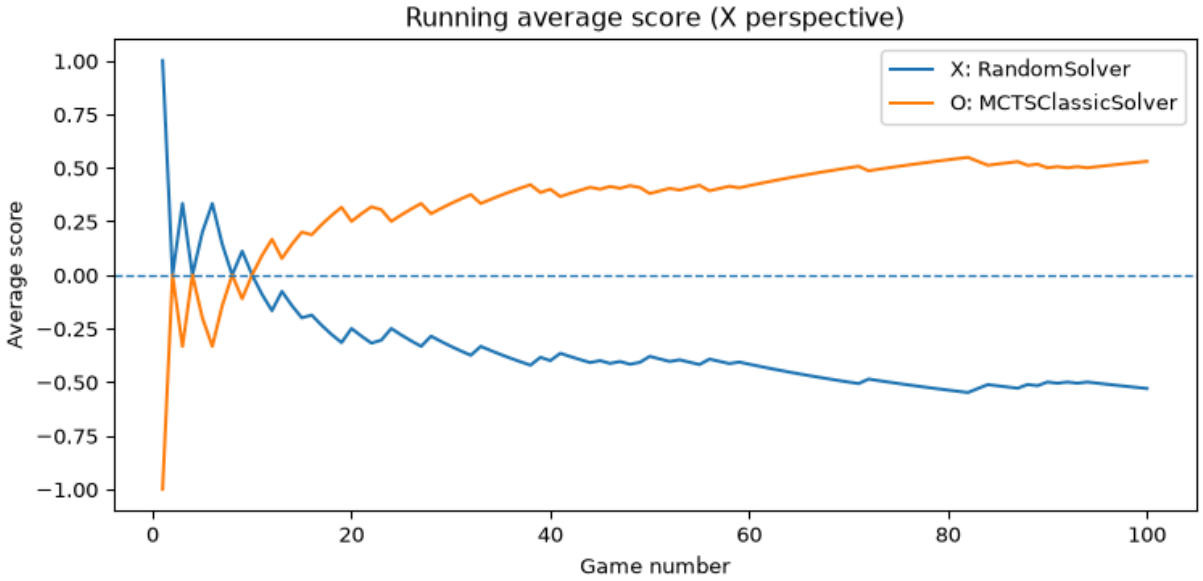
In [40]:

```python
def evaluate_solvers_with_plot(game, solver_X, solver_O, num_games):
    """
    Play 'num_games' games between solver_X (as 'X') and solver_O (as 'O'),
    track cumulative performance, and plot running average scores.

    Scoring is from X's perspective:
        outcome = +1  if X wins
        outcome = -1  if O wins
        outcome =  0  if draw

    The running average score for O is simply the negative of X's
    running average (zero-sum).
    """

    runner = GameRunner(game)

    # Counters for final summary
    results = {
        "X_wins": 0,
        "O_wins": 0,
        "draws": 0,
    }

    # For plotting: running average score as a function of game index
    avg_scores_X = []
    avg_scores_O = []

    cumulative_score_X = 0.0

    for i in range(num_games):
        outcome = runner.play_game(solver_X, solver_O)
        # Update win/draw counters
        if outcome == 1:
            results["X_wins"] += 1
        elif outcome == -1:
            results["O_wins"] += 1
        else:
            results["draws"] += 1

        # Update cumulative score (from X's perspective)
        cumulative_score_X += outcome
        avg_X = cumulative_score_X / (i + 1)
        avg_O = -avg_X  # zero-sum

        avg_scores_X.append(avg_X)
        avg_scores_O.append(avg_O)

    # Plot running average scores
    games = range(1, num_games + 1)
    plt.figure(figsize=(8, 4))
    plt.plot(games, avg_scores_X, label=f"X: {solver_X.get_name()}")
    plt.plot(games, avg_scores_O, label=f"O: {solver_O.get_name()}")
    plt.axhline(0.0, linestyle="--", linewidth=1)
```

```
    plt.xlabel("Game number")
    plt.ylabel("Average score")
    plt.title("Running average score (X perspective)")
    plt.legend()
    plt.tight_layout()
    plt.show()

    return results, avg_scores_X, avg_scores_O
```

Random vs MCTS

In [41]:
```
rand = RandomSolver(seed=0)
mcts = MCTSClassicSolver(num_simulations=10, seed=1)

results, avg_X, avg_O = evaluate_solvers_with_plot(game, rand,  mcts, num_ga
results
```

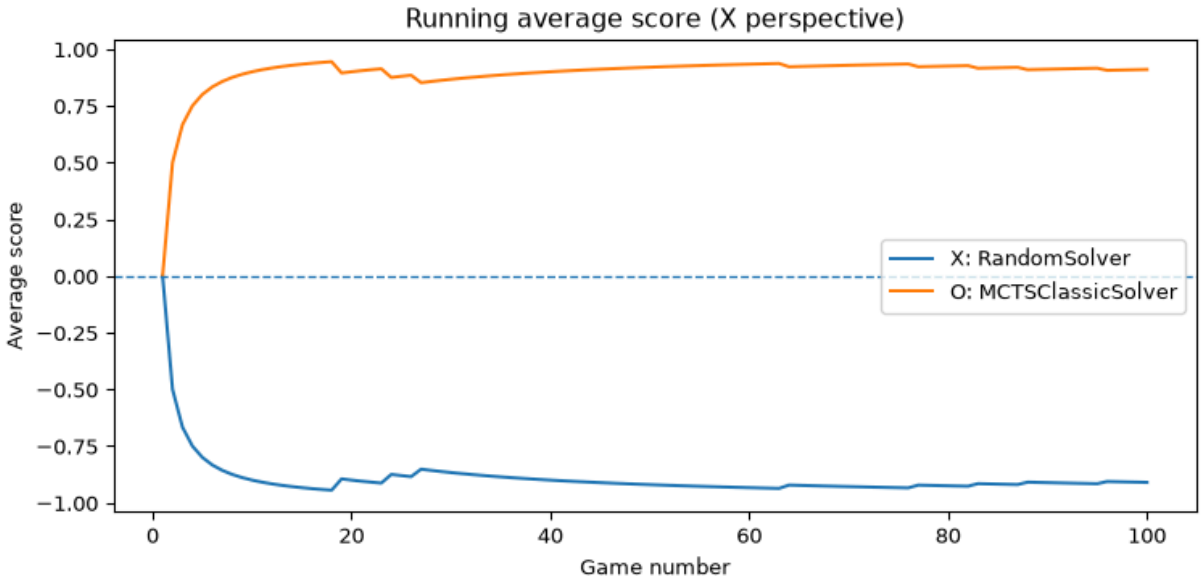



```
{'X_wins': 19, 'O_wins': 72, 'draws': 9}
```

`num_simulations=10`

Random vs MCTS

In [42]:
```
rand = RandomSolver(seed=0)
mcts = MCTSClassicSolver(num_simulations=100, seed=1)

results, avg_X, avg_O = evaluate_solvers_with_plot(game, rand,  mcts, num_ga
results
```

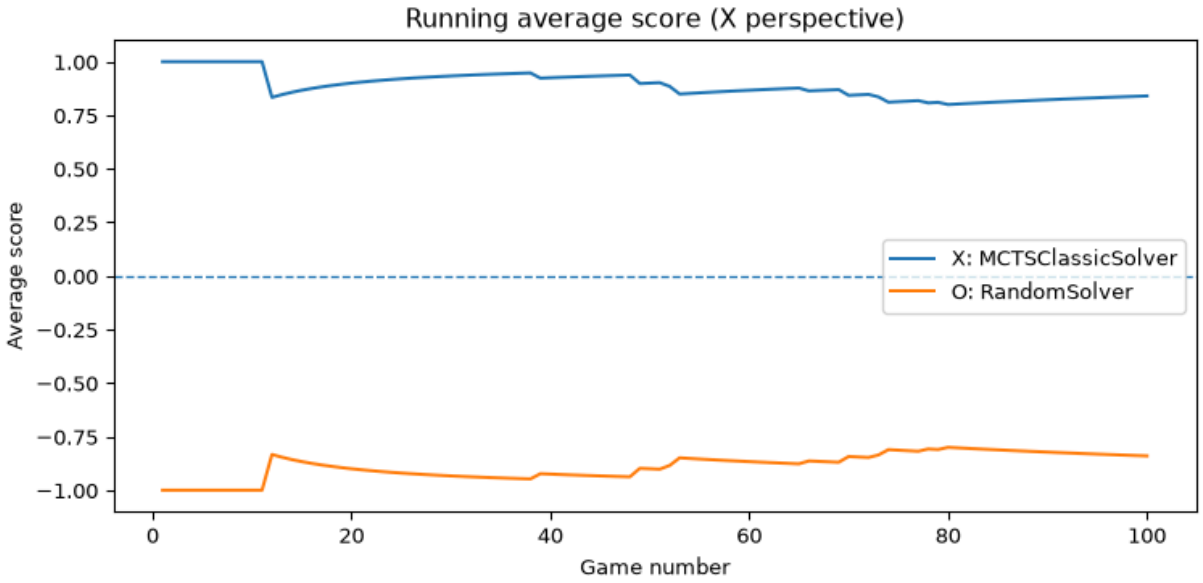



```
{'X_wins': 0, 'O_wins': 91, 'draws': 9}
```

`num_simulations=100`

MCTS vs Random

In [43]:
```
mcts = MCTSClassicSolver(num_simulations=10, seed=0)
rand = RandomSolver(seed=0)

results, avg_X, avg_O = evaluate_solvers_with_plot(game, mcts, rand, num_gam
results
```

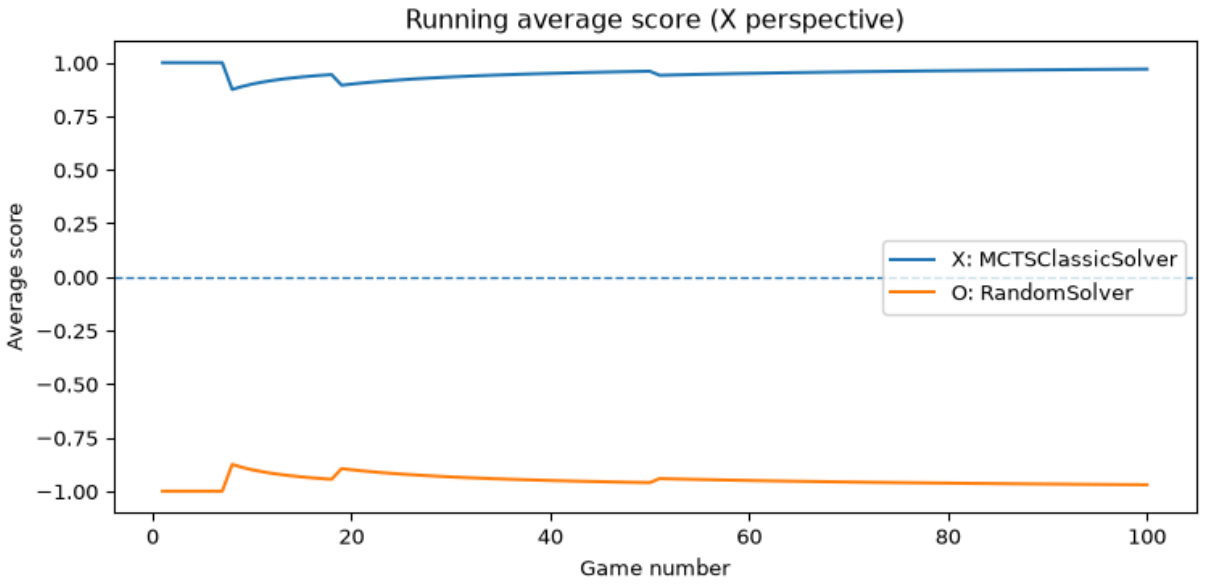



```
{'X_wins': 89, 'O_wins': 5, 'draws': 6}
```

`num_simulations=10`

MCTS vs Random

In [44]:
```
mcts = MCTSClassicSolver(num_simulations=100, seed=0)
rand = RandomSolver(seed=0)

results, avg_X, avg_O = evaluate_solvers_with_plot(game, mcts, rand, num_gam
results
```



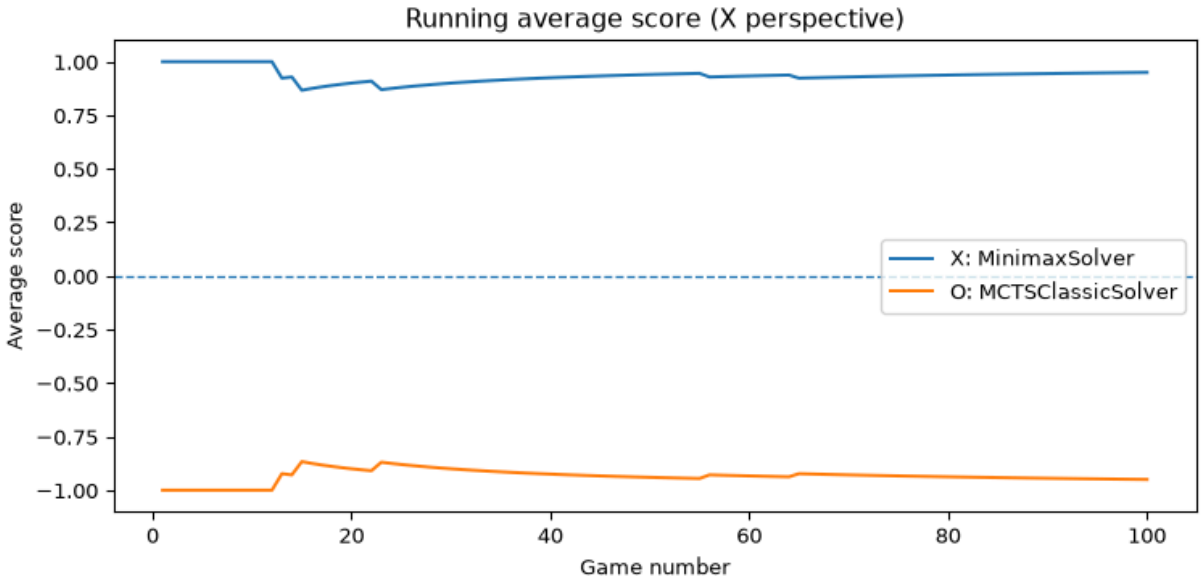

```
{'X_wins': 97, 'O_wins': 0, 'draws': 3}
```

`num_simulations=100`

As an exercise, modify the value of the `exploration` parameter. What happens when `exploration_c` is set to `math.sqrt(2)/2`?

Minimax vs MCTS

In [45]:
```
minimax = MinimaxSolver()
mcts = MCTSClassicSolver(num_simulations=10, seed=2)

results, avg_X, avg_O = evaluate_solvers_with_plot(game, minimax, mcts, num_
results
```

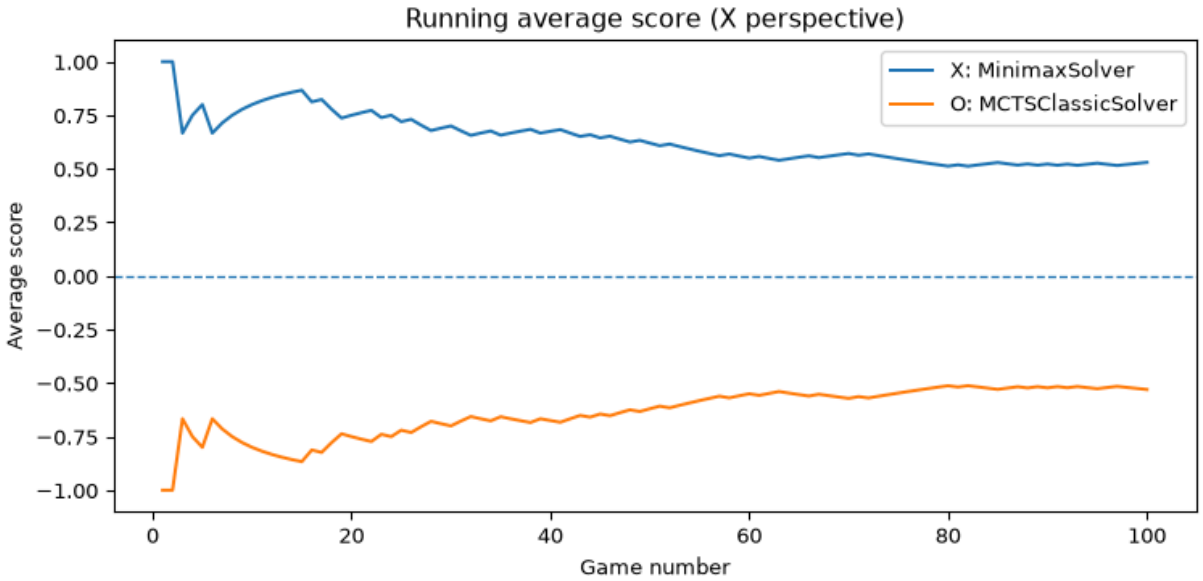



```
{'X_wins': 95, 'O_wins': 0, 'draws': 5}
```

`num_simulations=10`

Minimax vs MCTS

In [46]:
```
minimax = MinimaxSolver()
mcts = MCTSClassicSolver(num_simulations=100, seed=2)

results, avg_X, avg_O = evaluate_solvers_with_plot(game, minimax, mcts, num_
results
```

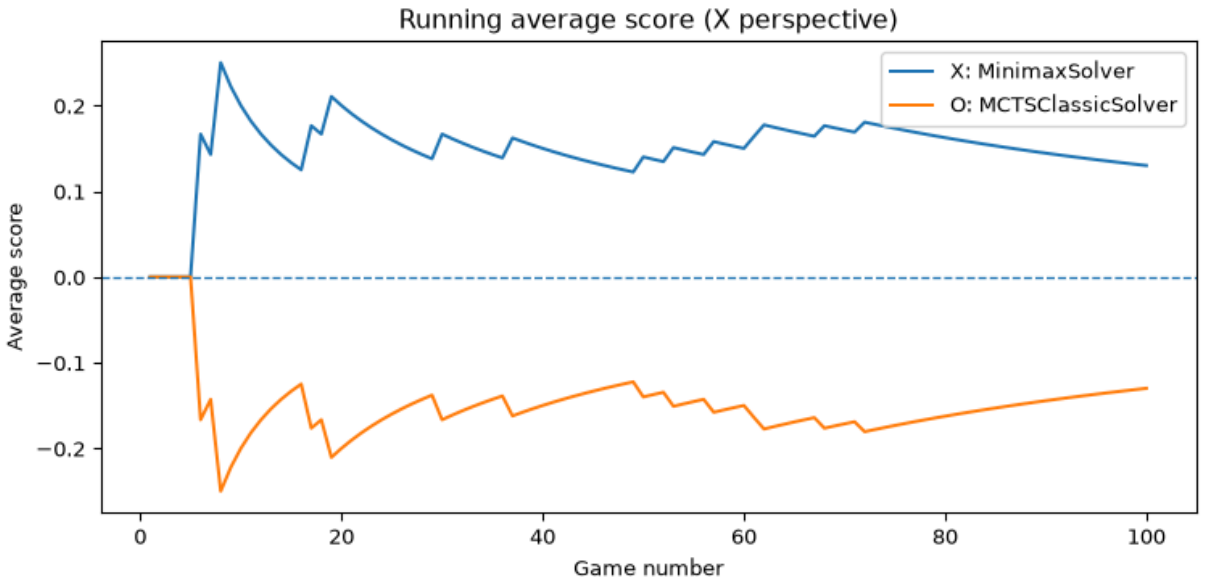



```
{'X_wins': 53, 'O_wins': 0, 'draws': 47}
```

`num_simulations=100`

Minimax vs MCTS

In [47]:
```
minimax = MinimaxSolver()
mcts = MCTSClassicSolver(num_simulations=500, seed=2)

results, avg_X, avg_O = evaluate_solvers_with_plot(game, minimax, mcts, num_
results
```

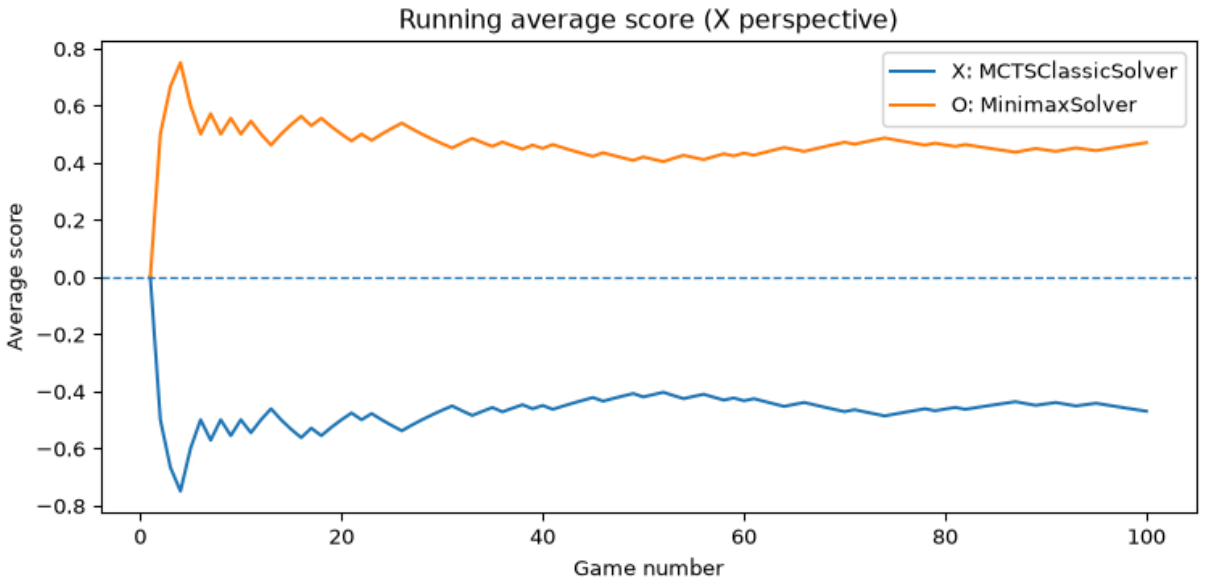



```
{'X_wins': 13, 'O_wins': 0, 'draws': 87}
```

`num_simulations=500`

MCTS vs Minimax

In [48]:
```
mcts = MCTSClassicSolver(num_simulations=10, seed=2)
minimax = MinimaxSolver()

results, avg_X, avg_O = evaluate_solvers_with_plot(game, mcts, minimax, num_
results
```

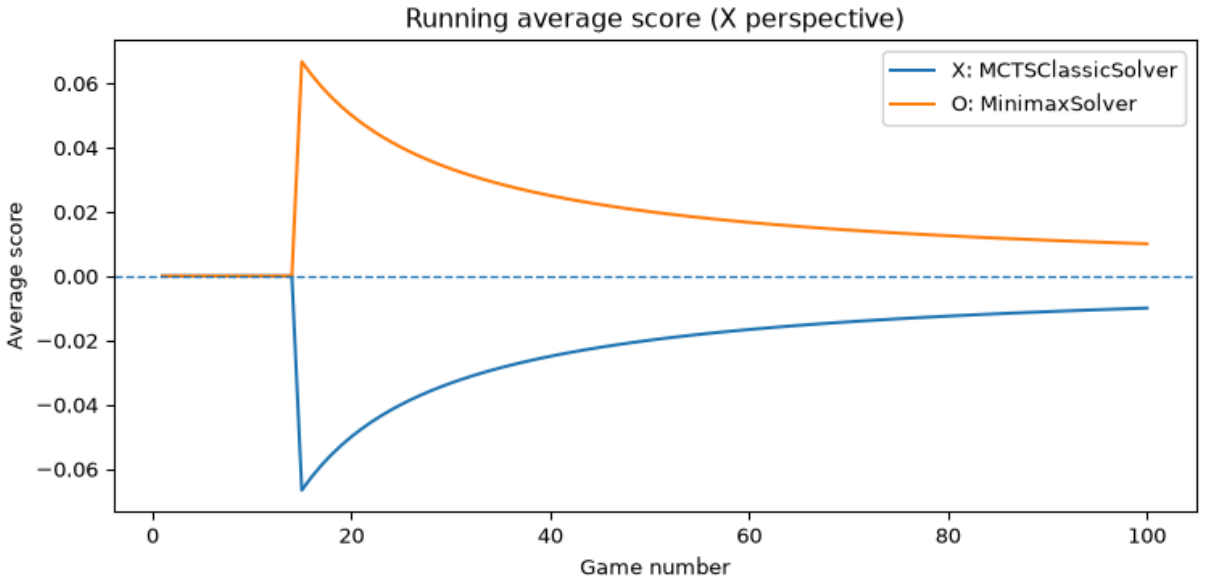


```
{'X_wins': 0, 'O_wins': 47, 'draws': 53}
```

`num_simulations=10`

MCTS vs Minimax

In [49]:
```
mcts = MCTSClassicSolver(num_simulations=100, seed=2)
minimax = MinimaxSolver()

results, avg_X, avg_O = evaluate_solvers_with_plot(game, mcts, minimax, num_
results
```

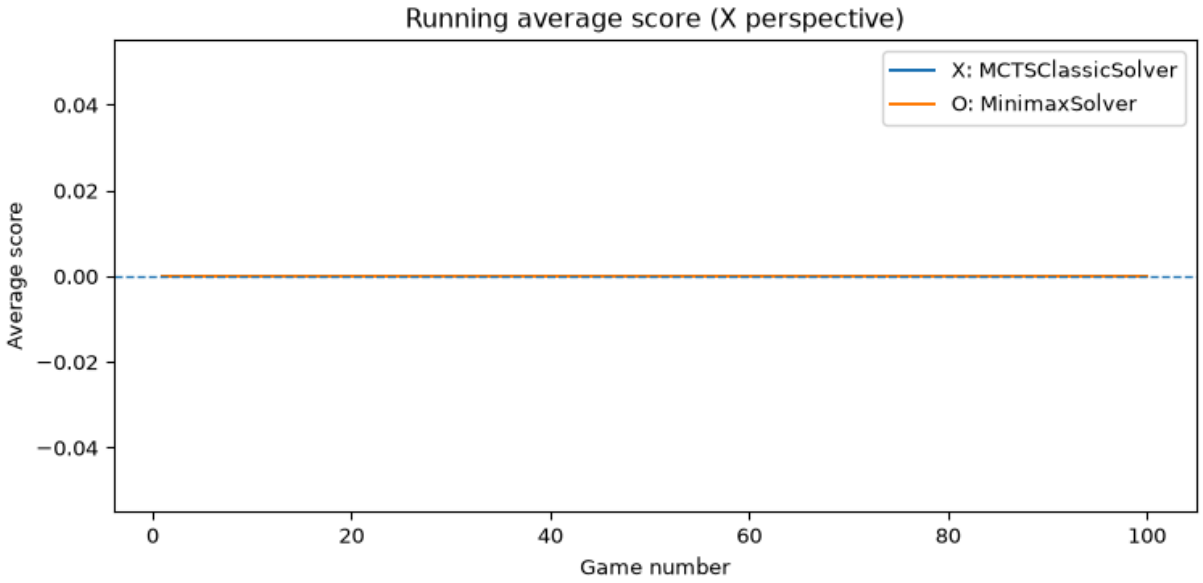



```
{'X_wins': 0, 'O_wins': 1, 'draws': 99}
```

`num_simulations=100`

MCTS vs Minimax

In [50]:
```
mcts = MCTSClassicSolver(num_simulations=500, seed=2)
minimax = MinimaxSolver()

results, avg_X, avg_O = evaluate_solvers_with_plot(game, mcts, minimax, num_
results
```

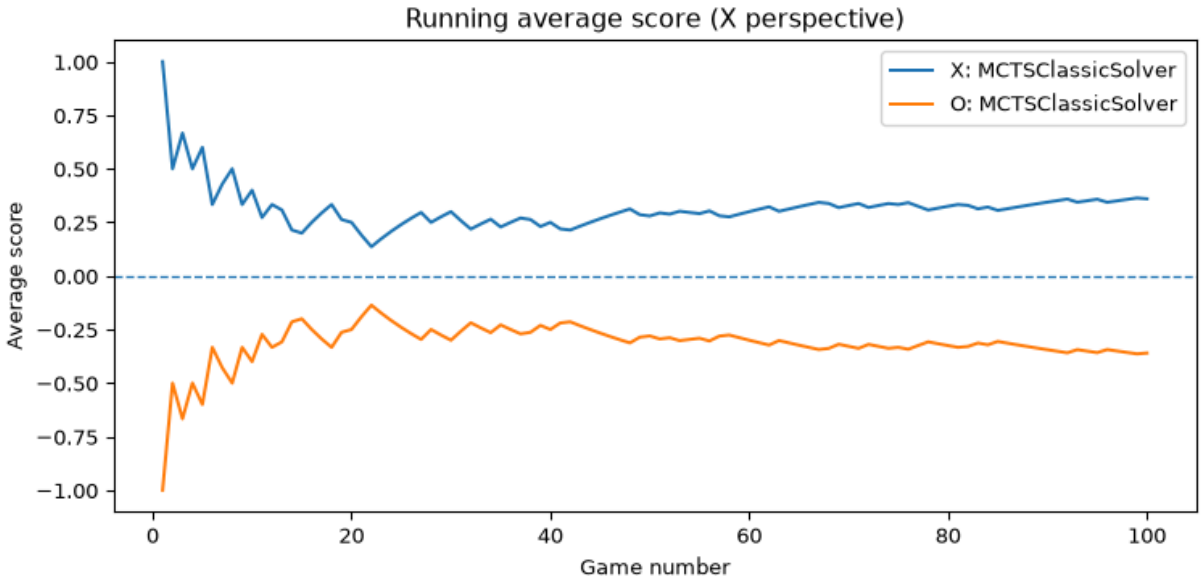



```
{'X_wins': 0, 'O_wins': 0, 'draws': 100}
```

`num_simulations=500`

MCTS (few sims) vs MCTS (few sims)

In [51]:
```
mcts_a = MCTSClassicSolver(num_simulations=10, seed=3)
mcts_b = MCTSClassicSolver(num_simulations=10, seed=4)

results, avg_X, avg_O = evaluate_solvers_with_plot(game, mcts_a, mcts_b, num
results
```

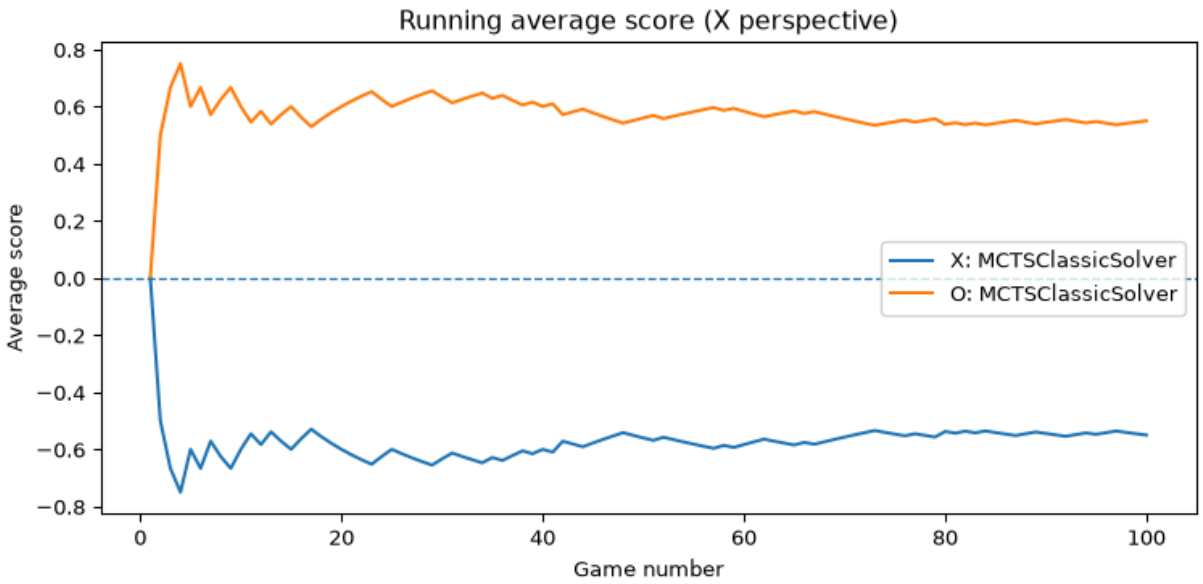



```
{'X_wins': 60, 'O_wins': 24, 'draws': 16}
```

MCTS (few sims) vs MCTS (many sims)

In [52]:
```
mcts_a = MCTSClassicSolver(num_simulations=10, seed=3)
mcts_b = MCTSClassicSolver(num_simulations=500, seed=4)
```

```
results, avg_X, avg_O = evaluate_solvers_with_plot(game, mcts_a, mcts_b, num
results
```




```
{'X_wins': 2, 'O_wins': 57, 'draws': 41}
```

Learning Across Moves and Games

In [53]:
```
class MCTSSolver(Solver):

    """
    Monte Carlo Tree Search solver for deterministic, zero-sum,
    two-player games like Tic-Tac-Toe.

    Key ideas:
    - The solver maintains a search tree keyed by canonical(state).
    - Each node stores:
        * N: visit count
        * W: total reward from the perspective of the player to move at
             that node (positive is good for that player)
        * children: mapping move -> child_state_key
        * untried_moves: moves that have not been expanded yet
        * player: the player to move at this node ("X" or "O")
    - select_move():
        * Ensures the current state is in the tree.
        * Runs a fixed number of simulations from the current root.
        * Returns the move leading to the most visited child.
    - opponent_played(move):
        * Advances the internal root along the actual move played
          (if that move has been explored).
        * This allows the solver to reuse search statistics across moves
          and across games.
    """

    def __init__(self, num_simulations=500, exploration_c=math.sqrt(2), seed

        """
```

```
        Parameters
        ----------
        num_simulations : int
            Number of MCTS simulations to run per move.
        exploration_c : float
            Exploration constant 'c' in the UCT formula.
        seed : int or None
            Optional random seed for reproducibility.
        """
        self.num_simulations = num_simulations
        self.exploration_c = exploration_c
        self.rng = random.Random(seed)

        # The search tree: state_key -> node dictionary
        self.tree = {}

        # Current root in the tree
        self.root_key = None   # canonical(state)
        self.root_player = None  # player to move at the root ("X" or "O")

        # Game reference (set on first select_move)
        self.game = None

    # ------------------------------
    # Public API
    # ------------------------------
    def select_move(self, game, state, player):
        """
        Choose a move for 'player' from 'state' using Monte Carlo Tree Searc

        This method:
        1. Synchronizes the internal root with the provided state.
        2. Runs a fixed number of MCTS simulations from the root.
        3. Returns the move leading to the child with the largest visit cour
        """
        self.game = game

        state_key = canonical(state)

        # Ensure the root in the tree corresponds to the current state.
        # If this state has been seen before, we reuse its node and statisti
        self.root_key = state_key
        self.root_player = player
        self._get_or_create_node(state_key, player)

        # Run MCTS simulations starting from the current root
        for _ in range(self.num_simulations):
            self._run_simulation()

        # After simulations, pick the child with the highest visit count.
        root_node = self.tree[self.root_key]

        if not root_node["children"]:
            # No children: must be a terminal state or no legal moves.
            # Fall back to any valid move (or raise error); here we pick at
            moves = self.game.get_valid_moves(np.array(self.root_key))
```

```
            return self.rng.choice(moves)

        best_move = None
        best_visits = -1

        for move, child_key in root_node["children"].items():
            child = self.tree[child_key]
            if child["N"] > best_visits:
                best_visits = child["N"]
                best_move = move

        return best_move

    def opponent_played(self, move):
        """
        Update the internal root based on the opponent's move.

        This is called by GameRunner after the other player has made a move.

        We try to move the root to the corresponding child node:
        - If the move was explored, we reuse that subtree.
        - Otherwise, we create a fresh node for the resulting state.
        """
        # If we do not yet have a root or a game reference, nothing to do.
        if self.root_key is None or self.game is None:
            return

        root_node = self.tree.get(self.root_key)
        if root_node is None:
            # Should not happen, but be robust.
            self.root_key = None
            self.root_player = None
            return

        # If we already explored this move from the root, just move down the
        if move in root_node["children"]:
            child_key = root_node["children"][move]
            self.root_key = child_key
            self.root_player = self.tree[child_key]["player"]
            return

        # Otherwise, we need to apply the move on the board and create a new
        state = np.array(self.root_key)
        player_who_played = root_node["player"]
        next_state = self.game.make_move(state, move, player_who_played)
        next_key = canonical(next_state)
        next_player = self.game.get_opponent(player_who_played)

        self.root_key = next_key
        self.root_player = next_player
        self._get_or_create_node(next_key, next_player)

    # ----------------------------
    # Internal helpers
    # ----------------------------
```

```
    def _get_or_create_node(self, state_key, player_to_move):
        """
        Ensure that a node for 'state_key' exists in the tree.

        If not present, create it with:
        - N = 0, W = 0
        - untried_moves = all valid moves from this state
        - children = {}
        - player = player_to_move
        """
        if state_key not in self.tree:
            state = np.array(state_key)
            self.tree[state_key] = {
                "N": 0,  # visit count
                "W": 0.0,  # total reward from this node's player's perspect
                "children": {},  # move -> child_state_key
                "untried_moves": self.game.get_valid_moves(state),
                "player": player_to_move,
            }
        return self.tree[state_key]

    def _run_simulation(self):
        """
        Perform one MCTS simulation from the current root:

        1. SELECTION:
           Follow the tree using UCT until we reach a node with untried_move
           or a terminal state.
        2. EXPANSION:
           If the node has untried_moves and is non-terminal, expand one mov
        3. ROLLOUT:
           From the new leaf, play random moves to a terminal state.
        4. BACKPROPAGATION:
           Propagate the final outcome back along the visited path.
        """
        if self.root_key is None:
            return  # nothing to do

        state_key = self.root_key
        state = np.array(state_key)

        path = []  # list of state_keys visited along this simulation

        # -------------------------
        # 1–2. Selection & Expansion
        # -------------------------
        while True:
            path.append(state_key)
            node = self.tree[state_key]

            # If this is a terminal state, stop and evaluate directly.
            if self.game.is_terminal(state):
                outcome = self.game.evaluate(state)  # from X's perspective
                break

            # If there are untried moves, expand one of them.
```

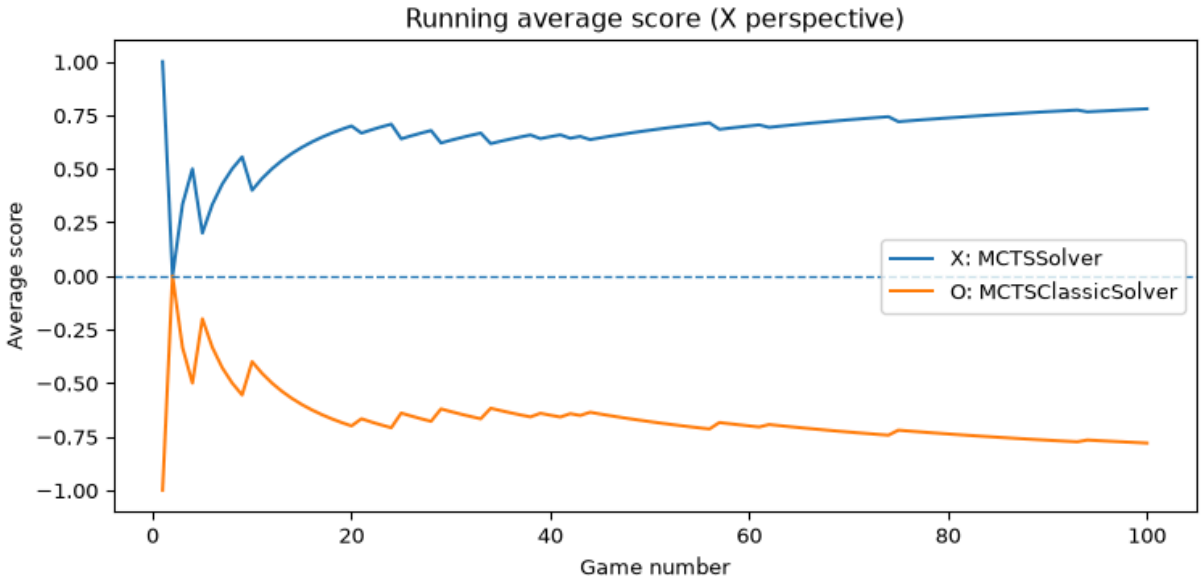
```
            if node["untried_moves"]:
                move = node["untried_moves"].pop()
                next_state = self.game.make_move(state, move, node["player"]
                next_key = canonical(next_state)
                next_player = self.game.get_opponent(node["player"])

                # Create the child node if it does not yet exist.
                self._get_or_create_node(next_key, next_player)

                # Link child in the tree
                node["children"][move] = next_key

                # The rollout starts from this new leaf node.
                state_key = next_key
                state = next_state
                path.append(state_key)

                outcome = self._rollout(state, next_player)
                break

            # Otherwise, the node is fully expanded: select a child using UC
            move, child_key = self._select_child(node)
            state_key = child_key
            state = np.array(state_key)

        # ------------------------
        # 4. Backpropagation
        # ------------------------
        self._backpropagate(path, outcome)

    def _select_child(self, node):
        """
        Select a child of 'node' using the UCT (Upper Confidence Bound) rule

        UCT score from the perspective of the player at 'node':

            score(child) = mean_reward_from_node_perspective
                           + c * sqrt( ln(N_parent + 1) / N_child )

        Note:
        - Each child stores W and N from its own player's perspective.
        - We convert the child's value to the parent's perspective by flippi
          the sign, because the child player is always the opponent of the
          parent player in a two-player alternating game.
        """
        parent_visits = node["N"]
        parent_player = node["player"]

        best_move = None
        best_child_key = None
        best_score = -math.inf

        for move, child_key in node["children"].items():
            child = self.tree[child_key]

            if child["N"] == 0:
```

```
                # Encourage exploring unvisited children at least once.
                uct_score = math.inf
            else:
                # Average reward from the child's player perspective.
                avg_child_reward = child["W"] / child["N"]

                # Convert to the parent's perspective.
                # Parent and child players are always opponents here.
                reward_from_parent_perspective = -avg_child_reward

                uct_score = (
                    reward_from_parent_perspective
                    + self.exploration_c
                    * math.sqrt(math.log(parent_visits + 1) / child["N"])
                )

            if uct_score > best_score:
                best_score = uct_score
                best_move = move
                best_child_key = child_key

        return best_move, best_child_key

    def _rollout(self, state, player_to_move):
        """
        Perform a random playout (simulation) from 'state' until a terminal

        Parameters
        ----------
        state : NumPy array
            Current board position.
        player_to_move : str
            Player to move ("X" or "O") at this rollout start.

        Returns
        -------
        outcome : int
            Final game result from X's perspective:
            +1 (X wins), -1 (O wins), or 0 (draw).
        """
        current_state = state.copy()
        current_player = player_to_move

        # Play random moves until the game ends.
        while not self.game.is_terminal(current_state):
            moves = self.game.get_valid_moves(current_state)
            move = self.rng.choice(moves)
            current_state = self.game.make_move(current_state, move, current
            current_player = self.game.get_opponent(current_player)

        return self.game.evaluate(current_state)  # +1, -1, or 0 from X's pe

    def _backpropagate(self, path, outcome):
        """
        Backpropagate the final outcome along the simulation path.
```

```
        Parameters
        ----------
        path : list of state_keys
            The sequence of states visited from root to leaf.
        outcome : int
            Final result from X's perspective: +1, -1, or 0.

        For each node on the path:
        - We convert 'outcome' to that node's player's perspective:
              reward = outcome   if player == "X"
                     = -outcome  if player == "O"
        - Then update:
              node.N += 1
              node.W += reward
        """
        for state_key in path:
            node = self.tree[state_key]
            player = node["player"]

            # Convert outcome from X's perspective to this node's perspectiv
            if player == "X":
                reward = outcome
            else:
                reward = -outcome

            node["N"] += 1
            node["W"] += reward
```

Same Computational Budget

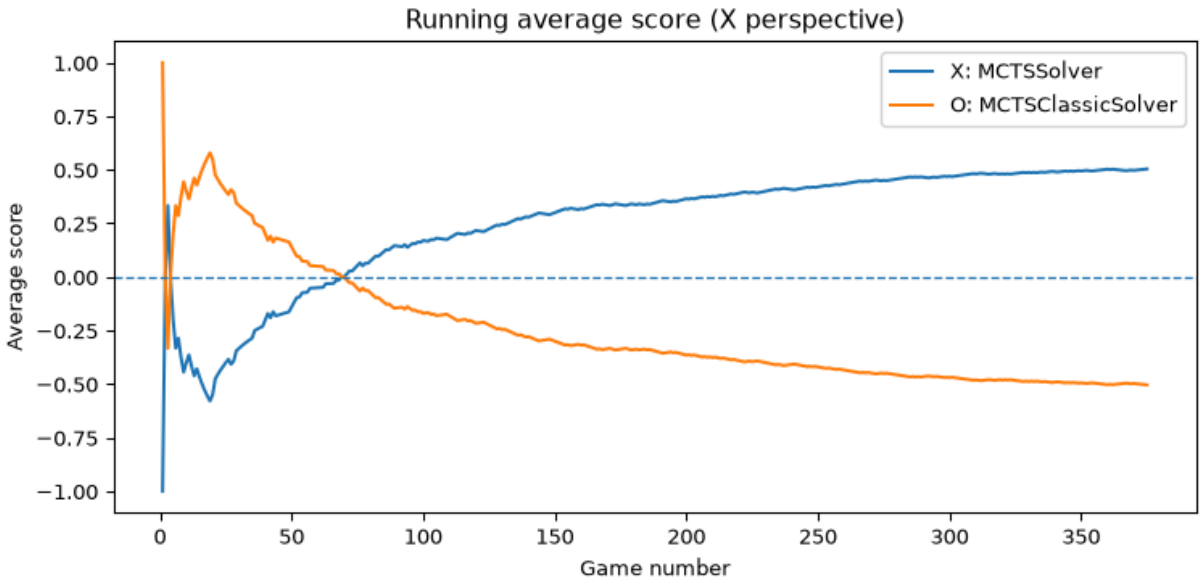
In [54]:
```
mcts_a = MCTSSolver(num_simulations=10, seed=3)
mcts_c = MCTSClassicSolver(num_simulations=10, seed=4)
results, _, _ = evaluate_solvers_with_plot(game, mcts_a, mcts_c, num_games=1
results
```




```
{'X_wins': 86, 'O_wins': 8, 'draws': 6}
```

Learner vs Thinker

In [55]:
```
a = MCTSSolver(num_simulations=10, seed=3)
b = MCTSClassicSolver(num_simulations=40, seed=4)

results, _, _ = evaluate_solvers_with_plot(game, a, b, num_games=375)
results
```

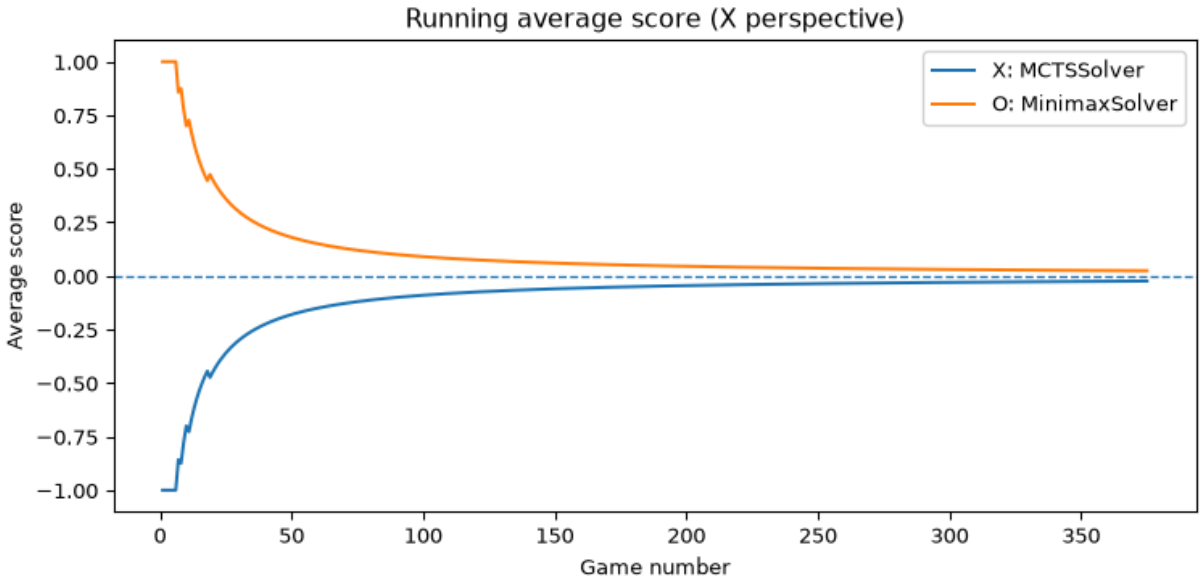



```
{'X_wins': 207, 'O_wins': 18, 'draws': 150}
```

In this example, the `MCTSSolver` executes a limited number of simulations, yet it accumulates learning over successive moves and games. In contrast, the `MCTSClassicSolver` is allocated a larger computational budget, enabling it to perform a greater number of simulations. Initially, the `MCTSClassicSolver` demonstrates superior performance for approximately the first 75 games. However, as the `MCTSSolver` accrues experience and its simulation count approaches that of the `MCTSClassicSolver`, its performance steadily improves.

Learner vs Minimax

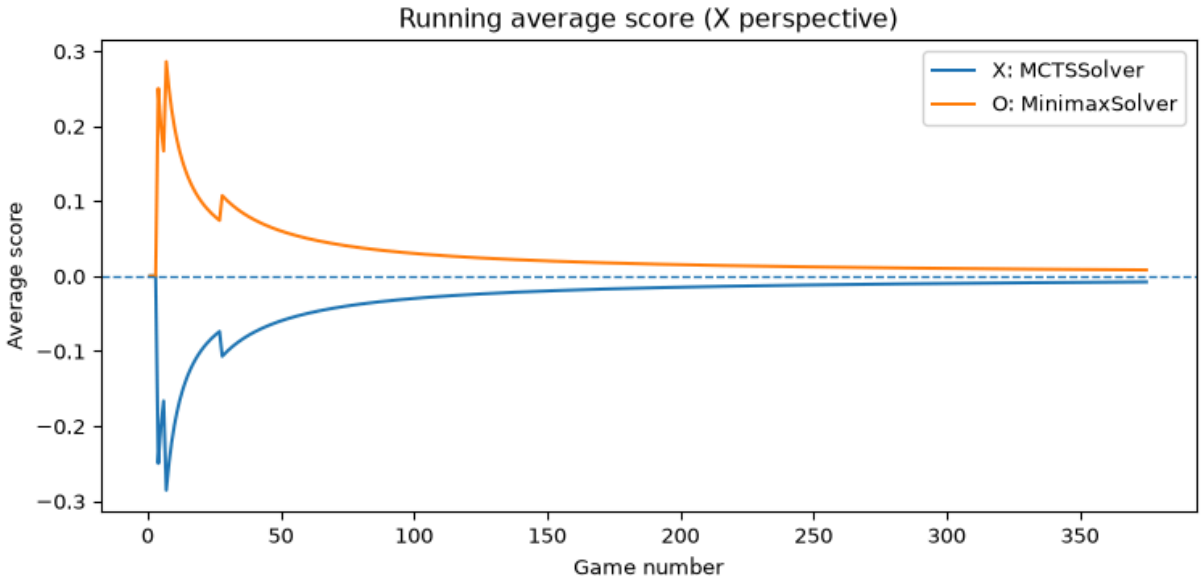
In [56]:
```
a = MCTSSolver(num_simulations=10, seed=3)
b = MinimaxSolver()

results, _, _ = evaluate_solvers_with_plot(game, a, b, num_games=375)
results
```




```
{'X_wins': 0, 'O_wins': 9, 'draws': 366}
```

Learner vs Minimax

In [57]:
```
a = MCTSSolver(num_simulations=20, seed=3)
b = MinimaxSolver()

results, _, _ = evaluate_solvers_with_plot(game, a, b, num_games=375)
results
```

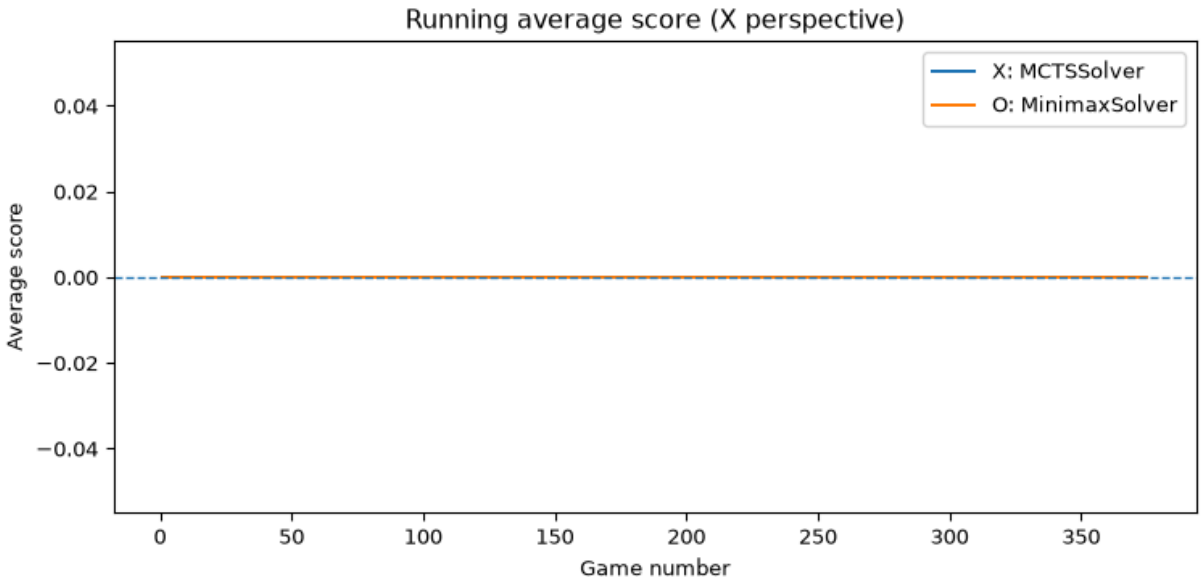



```
{'X_wins': 0, 'O_wins': 3, 'draws': 372}
```

Learner vs Minimax

In [58]:
```
a = MCTSSolver(num_simulations=50, seed=3)
b = MinimaxSolver()
```

```
results, _, _ = evaluate_solvers_with_plot(game, a, b, num_games=375)
results
```




```
{'X_wins': 0, 'O_wins': 0, 'draws': 375}
```

Exploration

- Incorporate heuristics to detect when a **winning move is achievable in a single step**.
- Experiment with varying the number of iterations and the **constant** C.

Monte Carlo Tree Search

Where we are now: plain MCTS

- We already know:
  - A **search tree** of Tic-Tac-Toe positions
  - Nodes store:
    - Number of times visited
    - Average game outcome from that position
  - MCTS uses this tree to choose moves

Monte Carlo Tree Search

- In our current code:
  - **Selection**: follow the tree (UCT) to promising nodes
  - **Expansion**: add a new child node
  - **Rollout**: play **random moves** to the end of the game
  - **Backpropagation**: send the final result back up the tree

Monte Carlo Tree Search

- This works very well for Tic-Tac-Toe, but:
  - Random rollouts can be slow and noisy in larger games
  - The tree does not “know” anything **before** search starts

Add a Policy Network

Goal: give MCTS a **better idea of which moves to explore first**.

New component: policy network

- Input: a board position
- Output: a **probability for each legal move**
  - “In this position, move A looks 40%, move B 30%, move C 10%, …”

Add a Policy Network

Usage inside MCTS:

- At a **new node** (when we expand a state):
  1. Call the **policy network** on the board
  2. Store the move probabilities as **priors** for this node
- During **selection**:
  - MCTS still uses visit counts from the tree
  - But now it also uses the policy priors to prefer moves that look good according to the network

Add a Value Network

Goal: avoid long random rollouts and get a **direct estimate** of how good a position is.

New component: value network

- Input: a board position
- Output: a single number:
  - Close to +1 if X is likely to win
  - Close to -1 if O is likely to win
  - Around 0 for a likely draw

Add a Value Network

Usage inside MCTS:

- At a **leaf node** (frontier of the tree):
  - Instead of doing a random rollout:
    1. Call the **value network** on the board
    2. Use its output as the leaf value
    3. Backpropagate this value up the tree

AlphaTicTacToe

Putting it together (AlphaGo-style "AlphaTicTacToe"):

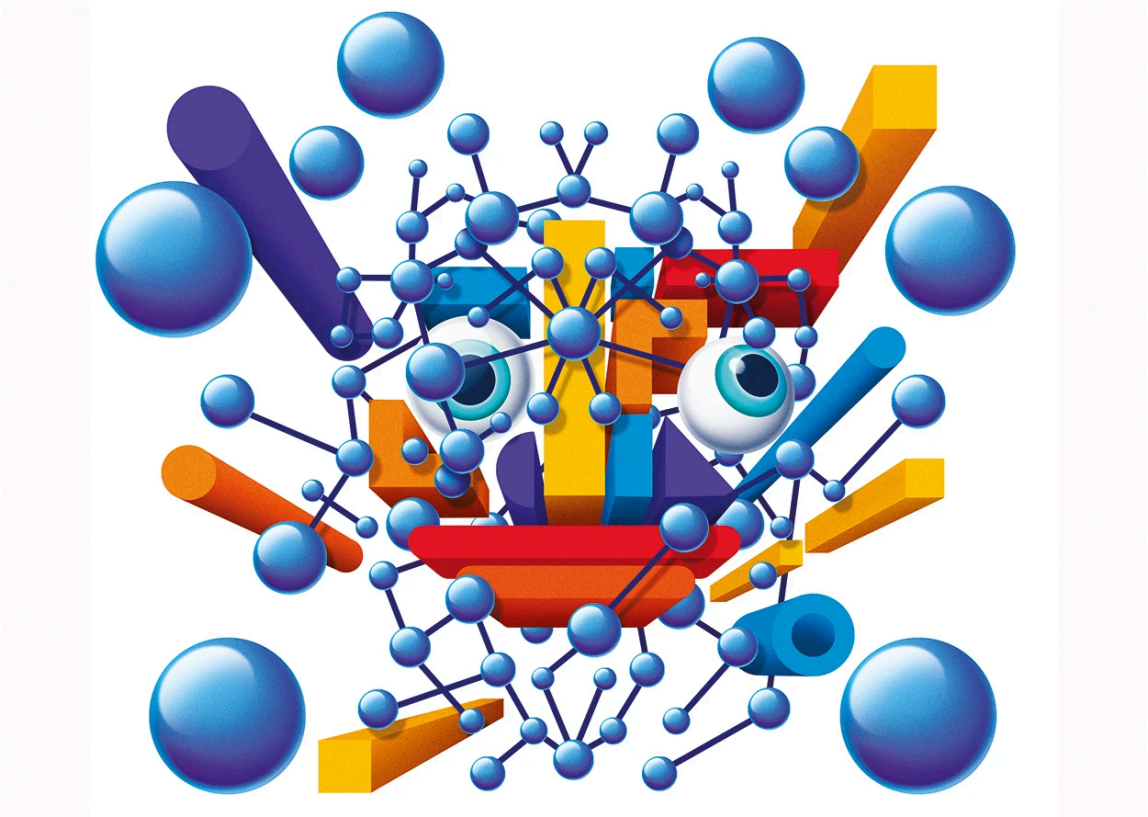
- MCTS + **policy network**:
  - Guides which moves to explore
- MCTS + **value network**:
  - Evaluates positions without random playouts
- Over time, both networks can be trained from example games (e.g., self-play):
  - Policy network learns "good moves"
  - Value network learns "good positions"

Learning reinforcement learning comfortably takes about a semester. However, in practice, systems like AlphaGo/AlphaZero combine **Monte Carlo Tree Search (MCTS)** with **deep neural networks** that output a policy (promising moves) and a value (how good a position is). These networks are trained from **self-play games**, using the game outcomes as a reward signal. That loop, *act, observe outcomes, update the policy/value to do better next time*, is exactly what we call **reinforcement learning**.

In later work, AlphaGo Zero and AlphaZero use **one** neural network with **two outputs** (policy and value). This simplifies the architecture.

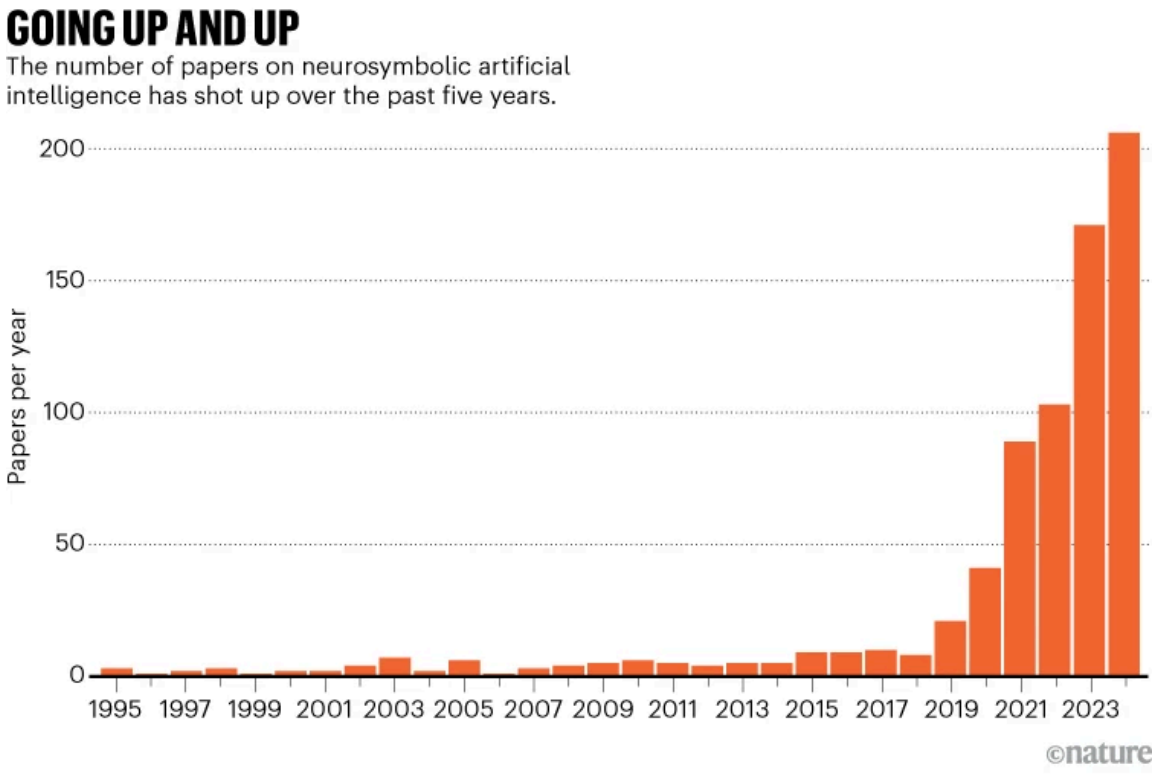
Prologue

Blending Logic and Learning



This AI combo could unlock human-level intelligence, Nature News, 2025-11-25.

I initially intended to develop one or two lectures centered on the integration of formal reasoning and deep learning; however, this plan did not materialize. Instead, I invite you to read this concise article, which explores why numerous experts argue that incorporating symbolic methods into neural networks may be the most effective, or possibly the sole, approach to embedding logical reasoning capabilities within artificial intelligence systems.



The Last Assignment

Transforming the Canadian economy with AI

Write a 1–2 page proposal on an **economically** relevant domain and answer:

- Which **algorithm(s)** from CSI 4106 is appropriate? Why?
- What makes the problem **computationally hard**?
- What **data** would be required?
- How would you evaluate **success**?

To be clear, this assignment is optional and not for grades. However, if the topic inspires you, feel free to send me your proposal. Exceptional entries will be posted in a Hall of Fame section of the course, with or without your name, depending on your preference.

Summary

- **Monte Carlo Tree Search (MCTS)** is a search algorithm used for **decision-making** in complex games.
- MCTS operates through four main steps: **Selection**, **Expansion**, **Rollout (Simulation)**, and **Backpropagation**.
- It balances **exploration** and **exploitation** using the **UCB1 formula**, which guides node selection based on visit counts and scores.

- MCTS maintains an **explicit search tree**, updating node values iteratively based on simulations.
- The algorithm has **wide-ranging applications**, including AI gaming, drug design, circuit routing, and autonomous driving.
- Introduced in 2008, MCTS gained prominence with its use in AlphaGo in 2016.
- Unlike traditional algorithms like $A^\star$, MCTS uses **dynamic policies** and leverages all visited nodes for decision-making.
- Implementing MCTS involves tracking node statistics and applying the **UCB1** formula to **guide search**.
- A practical example of MCTS is demonstrated through implementing **Tic-Tac-Toe**.
- Further exploration includes integrating **MCTS** with **deep learning models** like AlphaZero and MuZero.

Further exploration

- JAX-native environment for simulations
- AlphaZero
- MuZero
- Gumbel
- MCTS Tic-Tac-Toe with Visualization

The End

- Consult the course website for information on the final examination.

References

Besta, Maciej, Julia Barth, Eric Schreiber, et al. 2025. *Reasoning Language Models: A Blueprint*. https://arxiv.org/abs/2501.11223.

Browne, Cameron B., Edward Powley, Daniel Whitehouse, et al. 2012. "A Survey of Monte Carlo Tree Search Methods." *IEEE Transactions on Computational Intelligence and AI in Games* 4 (1): 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810.

Chaslot, Guillaume, Sander Bakkes, Istvan Szita, and Pieter Spronck. 2008. "Monte-Carlo Tree Search: a new framework for game AI." *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Stanford, California), AIIDE'08, 216–17.

Kemmerling, Marco, Daniel Lütticke, and Robert H. Schmitt. 2024. "Beyond Games: A Systematic Review of Neural Monte Carlo Tree Search Applications." *Applied Intelligence* 54 (1): 1020–46. https://doi.org/10.1007/s10489-023-05240-w.

Russell, Stuart, and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson. http://aima.cs.berkeley.edu/.

Silver, David, Aja Huang, Chris J. Maddison, et al. 2016. “Mastering the game of Go with deep neural networks and tree search.” *Nature* 529 (7587): 484–89. https://doi.org/10.1038/nature16961.

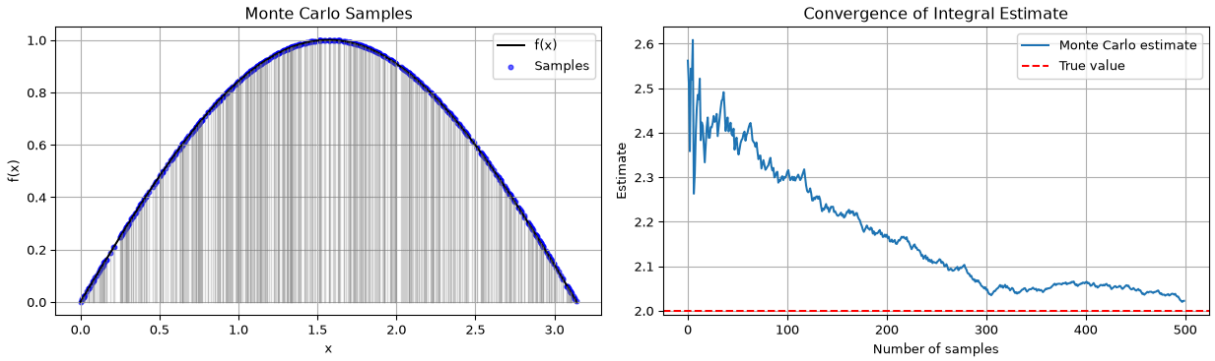
Świechowski, Maciej, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. 2023. “Monte Carlo Tree Search: a review of recent modifications and applications.” *Artificial Intelligence Review* 56 (3): 2497–562. https://doi.org/10.1007/s10462-022-10228-y.

Appendix

Numerical Integration

In [59]:
```
import random
import math
import numpy as np
import matplotlib.pyplot as plt

def monte_carlo_integrate_visual_with_sticks(f, a, b, n_samples, seed=None):

    """
    Monte Carlo integration visualization.
    Shows the function curve, sampled points, and vertical lines ("sticks").
    Also plots convergence of the Monte Carlo estimate.
    """

    if seed is not None:
        np.random.seed(seed)

    xs = np.random.uniform(a, b, size=n_samples)
    ys = f(xs)

    cumulative_avg = np.cumsum(ys) / np.arange(1, n_samples + 1)
    estimates = (b - a) * cumulative_avg

    fig, ax = plt.subplots(1, 2, figsize=(13, 4))

    # ----- Left panel: samples + vertical lines -----
    X = np.linspace(a, b, 400)
    ax[0].plot(X, f(X), color="black", label="f(x)")

    # Vertical lines
    for x_i, y_i in zip(xs, ys):
        ax[0].plot([x_i, x_i], [0, y_i], color="gray", alpha=0.3, linewidth=

    # Sampled points
    ax[0].scatter(xs, ys, s=12, color="blue", alpha=0.6, label="Samples")
```

```
    ax[0].set_title("Monte Carlo Samples")
    ax[0].set_xlabel("x")
    ax[0].set_ylabel("f(x)")
    ax[0].grid(True)
    ax[0].legend()

    # ----- Right panel: convergence -----
    true_value = 2.0  # ∫₀^π sin(x) dx
    ax[1].plot(estimates, label="Monte Carlo estimate")
    ax[1].axhline(true_value, linestyle="--", color="red", label="True value

    ax[1].set_title("Convergence of Integral Estimate")
    ax[1].set_xlabel("Number of samples")
    ax[1].set_ylabel("Estimate")
    ax[1].grid(True)
    ax[1].legend()

    plt.tight_layout()
    plt.show()

    return estimates[-1]

def main():
    f = np.sin
    a, b = 0.0, math.pi
    n_samples = 500

    estimate = monte_carlo_integrate_visual_with_sticks(f, a, b, n_samples,
    print(f"Final estimate ≈ {estimate:.6f} (true = 2.0)")

main()
```




```
Final estimate ≈ 2.022106 (true = 2.0)
```

Numerical Integration

In [60]:

```
def monte_carlo_integrate(f, a, b, n_samples, seed=None):

    """
    Estimate ∫_a^b f(x) dx using simple Monte Carlo integration.

    Parameters
    ----------
    f : callable
```

```
        Function to integrate.
    a, b : float
        Integration bounds (a < b).
    n_samples : int
        Number of random samples to draw.
    seed : int or None
        Optional seed for reproducibility.

    Returns
    -------
    estimate : float
        Monte Carlo estimate of the integral.
    """

    if seed is not None:
        random.seed(seed)

    total = 0.0
    for _ in range(n_samples):
        x = random.uniform(a, b)
        total += f(x)

    return (b - a) * total / n_samples

def main():
    # Example: integrate f(x) = sin(x) on [0, π]
    f = math.sin
    a, b = 0.0, math.pi

    for n in [100, 1_000, 10_000, 100_000]:
        estimate = monte_carlo_integrate(f, a, b, n, seed=0)
        print(f"n = {n:6d} → estimate ≈ {estimate:.6f} (delta = {abs(2.0 - e

main()
```

```
n =    100 → estimate ≈ 2.080957 (delta = 0.080957)
n =   1000 → estimate ≈ 1.992136 (delta = 0.007864)
n =  10000 → estimate ≈ 2.007041 (delta = 0.007041)
n = 100000 → estimate ≈ 1.996149 (delta = 0.003851)
```

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EE**CS**)

University of Ottawa