Chess as a Search Problem

A breadth-first search companion to Lecture 16

Marcel Turcotte

Learning objectives

This self-study companion applies the search terminology and breadth-first search (BFS) from Lecture 16 to a constrained puzzle. It uses chess movement rules without playing a complete game of chess. BFS provides a baseline for investigating more advanced search methods in later lectures.

By the end of the example, we will be able to:

- formulate a puzzle as states, actions, transitions, a constraint, a goal test, and a path cost;
- explain how the safety constraint determines which actions are available;
- explain why repeated-state detection matters; and
- use BFS to find a minimum-move solution and report what it explored.

The puzzle

This puzzle was inspired by an exercise in Lichess Learn.

White moves repeatedly; Black never moves. White must capture all Black pieces in as few moves as possible. After each White move, every White piece attacked by a Black piece must be defended by another White piece.

The initial position is allowed to violate this safety rule. In fact, it is deliberately unsafe: White's first move must repair the problem.

Component	Definition in this puzzle

Search formulation

Component	Definition in this puzzle
State	The locations and types of all pieces
Initial state	The position encoded below in FEN notation
Actions	White's pseudo-legal chess moves whose resulting states satisfy the safety constraint
Transition model	Move one White piece; remove a captured Black piece; promote a pawn that reaches the last rank
Validity constraint	Every attacked White piece is defended
Goal test	No Black pieces remain
Action cost	One unit per White move

The **path cost** is the number of White moves. All actions have the same cost, so BFS finds a minimum-move solution. The implementation first enumerates *candidate actions* from the piece movement rules, then retains only those whose resulting states satisfy the safety constraint.

Setup and initial board

To use your own Python notebook environment, install the dependencies once and select that environment's kernel:

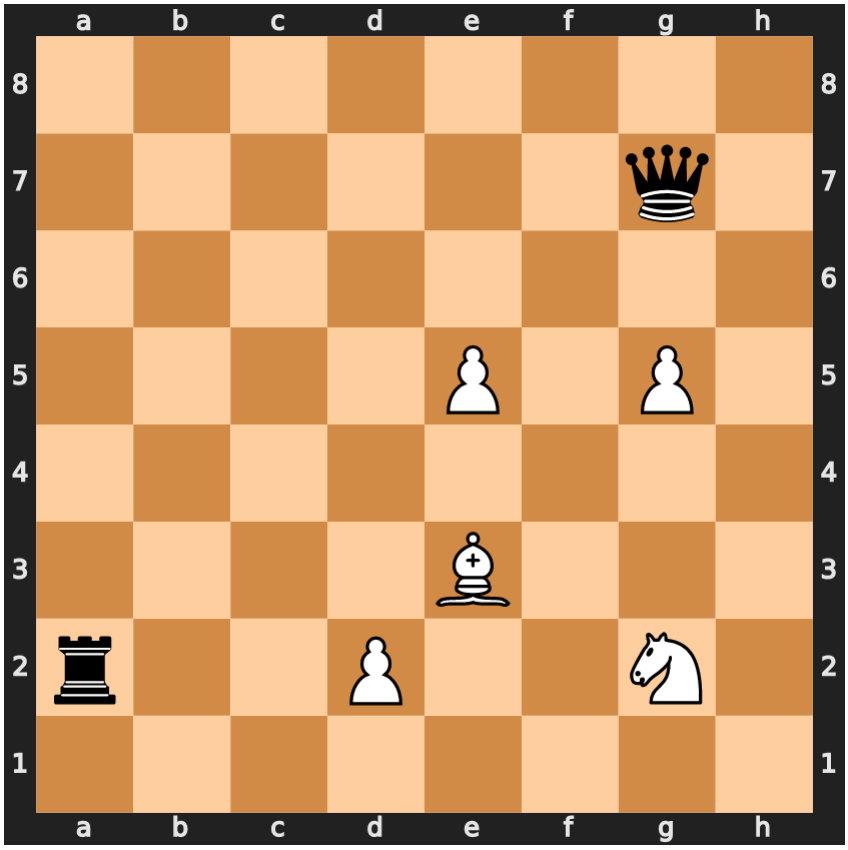
```
python -m pip install chess resvg-py ipykernel
```

```
from collections import deque
from dataclasses import dataclass
from typing import Optional

import chess
import chess.svg
import resvg_py
from IPython.display import Image, Markdown, display
```

```
def board_image(
    position,
    *,
    display_size=420,
    scale=2,
    lastmove=None,
    fill=None,
):
    """Return a high-resolution PNG chessboard for notebook display."""
    svg_options = {
        "board": position,
        "size": display_size * scale,
    }

    if lastmove is not None:
        svg_options["lastmove"] = lastmove
    if fill is not None:
        svg_options["fill"] = fill

    svg = chess.svg.board(**svg_options)
    png = resvg_py.svg_to_bytes(svg_string=svg)

    return Image(data=png, width=display_size)
```

FEN records the board rank by rank, from rank 8 down to rank 1. Uppercase letters are White pieces; lowercase letters are Black pieces. The `w` says that White moves next.

```
START_FEN = "8/6q1/8/4P1P1/8/4B3/r2P2N1/8 w - - 0 1"

board = chess.Board(START_FEN)
display(board_image(board))
```



The position contains a Black rook on a2 and queen on g7. White has pawns on d2, e5, and g5, a bishop on e3, and a knight on g2. Kings are unnecessary for this puzzle, so we use *pseudo-legal* movement rather than the rules for check. Ordinary pawn promotion is allowed; castling and en passant are absent.

Goal test

A goal state contains no Black pieces. Expressing the goal as a small, independent predicate makes it easy to test and reuse.

```
def is_goal(position):
    """Return True exactly when no Black pieces remain."""
    return not any(
        piece.color == chess.BLACK
        for piece in position.piece_map().values()
    )
```

```
assert not is_goal(board)
```

The safety constraint

Let $T(p, s)$ mean that a Black piece attacks White piece p in state s, and let $D(p, s)$ mean that another White piece defends it. A valid post-move state satisfies

$$\forall p \in W(s), \qquad T(p, s) \Rightarrow D(p, s).$$

An unattacked White piece does not need a defender. Safety is a property of the *whole resulting position*, not only of the piece that moved.

```
def threatened_white_squares(position):
    """Return the squares of White pieces attacked by Black."""
    return {
        square
        for square, piece in position.piece_map().items()
        if piece.color == chess.WHITE
        and position.attackers(chess.BLACK, square)
    }


def undefended_white_squares(position):
    """Return attacked White-piece squares that have no White defender."""
    return {
        square
        for square in threatened_white_squares(position)
        if not position.attackers(chess.WHITE, square)
    }


def is_safe(position):
    """
    Determine whether a position satisfies the puzzle's safety rule.

    A position is safe if every White piece attacked by at least one Black
    piece is defended by at least one other White piece. White pieces that are
    not under attack do not need to be defended.
    """
    return not undefended_white_squares(position)
```

```
initial_problems = sorted(
    chess.square_name(square) for square in undefended_white_squares(board)
)
print("Unsafe White pieces in the initial state:", initial_problems)
assert not is_safe(board)
assert initial_problems == ["e5"]
```

```
Unsafe White pieces in the initial state: ['e5']
```

The output confirms the intended reading of the puzzle: the pawn on e5 is attacked by the queen and has no defender.

Sanity checks: allowed and forbidden moves

Before searching, we should test the transition rule against examples whose answers we already know. The helper below applies one White move and resets the turn to White, because Black never takes a turn in this puzzle.

Moves use **UCI notation**: `e5e6` moves a piece from e5 to e6. A final letter specifies a promotion: `e7e8q` moves a pawn to e8 and promotes it to a queen.

```
def after(position, uci):
    """Return the state after one pseudo-legal White move in UCI notation."""
    parent = position.copy(stack=False)
    parent.turn = chess.WHITE
    move = chess.Move.from_uci(uci)

    if move not in parent.pseudo_legal_moves:
        raise ValueError(f"{uci} is not a pseudo-legal White move")

    parent.push(move)
    parent.turn = chess.WHITE
    return parent


expected = {
    "e3f4": True,   # The bishop now defends e5.
    "e5e6": True,   # The pawn leaves the queen's attack.
    "g2h4": False,  # The original problem on e5 remains.
```

```
    "d2d3": False,  # The rook now attacks the undefended knight on g2.
}

for uci, should_be_safe in expected.items():
    child = after(board, uci)
    problems = sorted(
        chess.square_name(square)
        for square in undefended_white_squares(child)
    )
    result = is_safe(child)
    print(
        f"{uci}: {'allowed' if result else 'forbidden':9s} "
        f"(undefended: {problems or 'none'})"
    )
    assert result == should_be_safe
```

```
e3f4: allowed   (undefended: none)
e5e6: allowed   (undefended: none)
g2h4: forbidden (undefended: ['e5'])
d2d3: forbidden (undefended: ['e5', 'g2'])
```

These examples reveal why validity must be checked *after* applying an action. For example, `g2h4` is a normal knight move, but it is forbidden because the resulting state still leaves e5 attacked and undefended.

Generating valid successors

Successor generation has three conceptual steps:

$$\begin{aligned}\text{candidate action} &\longrightarrow \text{apply transition}\\ &\longrightarrow \text{keep the state only if it is safe.}\end{aligned}$$

```
def expand(position):
    """Yield `(move, child)` for every safe state one White move away."""
    parent = position.copy(stack=False)
    parent.turn = chess.WHITE

    for move in parent.pseudo_legal_moves:
        child = parent.copy(stack=False)
        child.push(move)
```

```
        child.turn = chess.WHITE

        if is_safe(child):
            yield move, child


first_moves = [move.uci() for move, _ in expand(board)]
print(f"The initial state has {len(first_moves)} valid successors:")
print(first_moves)

assert "e3f4" in first_moves
assert "e5e6" in first_moves
assert "g2h4" not in first_moves
assert "d2d3" not in first_moves
```

```
The initial state has 2 valid successors:
['e3f4', 'e5e6']
```

Repeated states and the state key

The same arrangement can be reached by more than one path. Pieces can also move away and later return, creating cycles. Without repeated-state detection, BFS would waste work rediscovering those positions.

Only the piece placement affects future decisions in this puzzle: White always moves, there is no castling or en passant, and move counters do not matter. `board_fen()` therefore gives an appropriate state key without retaining irrelevant bookkeeping from the full FEN.

```
def state_key(position):
    """Return the part of the state that matters for future decisions."""
    return position.board_fen()


assert state_key(board) == START_FEN.split()[0]
```

This is a general modeling lesson: a search state should retain everything that can affect the future, but it need not retain everything a library happens to store.

Breadth-first search

The frontier stores search nodes as pairs `(state, path)`. Here, `path` is a sequence of **actions** (chess moves); the Lecture 16 implementation instead records the sequence of **states** along a path. Both describe how to reach the node's state from the initial state.

The `reached` set includes states on the frontier and states already expanded. It prevents duplicate work. The counters expose the search effort that would otherwise be hidden behind the final answer.

```
@dataclass(frozen=True)
class SearchStats:
    solution_depth: Optional[int]
    states_expanded: int
    states_generated: int
    duplicate_states: int
    max_frontier_size: int
    reached_states: int


def bfs(start):
    """Find a minimum-move solution with breadth-first graph search."""
    root = start.copy(stack=False)
    root.turn = chess.WHITE

    frontier = deque([(root, [])])
    reached = {state_key(root)}

    expanded = 0
    generated = 0
    duplicates = 0
    max_frontier = len(frontier)

    while frontier:
        position, path = frontier.popleft()

        # BFS tests states in nondecreasing path length.
        if is_goal(position):
            stats = SearchStats(
                solution_depth=len(path),
                states_expanded=expanded,
                states_generated=generated,
                duplicate_states=duplicates,
```

```
                max_frontier_size=max_frontier,
                reached_states=len(reached),
            )
            return path, stats

        expanded += 1

        for move, child in expand(position):
            generated += 1
            key = state_key(child)

            if key in reached:
                duplicates += 1
                continue

            reached.add(key)
            frontier.append((child, path + [move]))
            max_frontier = max(max_frontier, len(frontier))

    stats = SearchStats(
        solution_depth=None,
        states_expanded=expanded,
        states_generated=generated,
        duplicate_states=duplicates,
        max_frontier_size=max_frontier,
        reached_states=len(reached),
    )
    return None, stats
```

The code uses **graph search**: `reached` ensures that each distinct piece placement enters the frontier at most once. Here, "generated" counts valid successors before duplicate removal, while "expanded" counts non-goal states removed from the frontier and examined.

Run and inspect the search

```
solution, stats = bfs(board)

if solution is None:
    print("No solution found.")
else:
```

```
    print(f"Minimum solution length: {len(solution)} moves")
    print("Moves:", " ".join(move.uci() for move in solution))

print()
print(f"States expanded:       {stats.states_expanded:,}")
print(f"States generated:      {stats.states_generated:,}")
print(f"Duplicate states:      {stats.duplicate_states:,}")
print(f"Maximum frontier size: {stats.max_frontier_size:,}")
print(f"Distinct states reached: {stats.reached_states:,}")

assert solution is not None
assert len(solution) == stats.solution_depth
```

```
Minimum solution length: 8 moves
Moves: e5e6 g2h4 h4f5 f5g7 e6e7 e7e8q e8g8 g8a2

States expanded:       4,101
States generated:      50,202
Duplicate states:      34,959
Maximum frontier size: 11,143
Distinct states reached: 15,244
```

The solution is shortest because every edge costs one move and BFS examines shallower states before deeper ones. The statistics also show that the queue is only part of the algorithm: state representation, constraint filtering, and duplicate detection all materially shape the search.

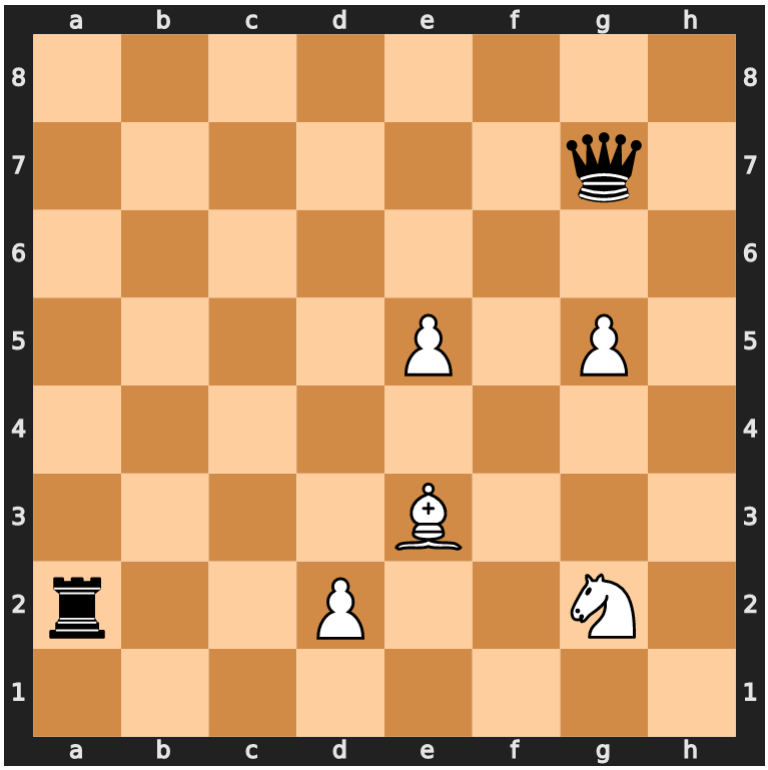
Visualize the solution

The display below begins with the initial state and then shows every move in the minimum solution. The most recent move is highlighted by `lastmove`. Gold squares mark White pieces currently attacked by Black; because all post-move states are valid, each such piece also has a White defender.

```
THREATENED_COLOR = "#f2c14e99"


def display_solution(start, moves, size=380):
    """Display the initial state and each annotated position on a path."""
    current = start.copy(stack=False)
    current.turn = chess.WHITE
```

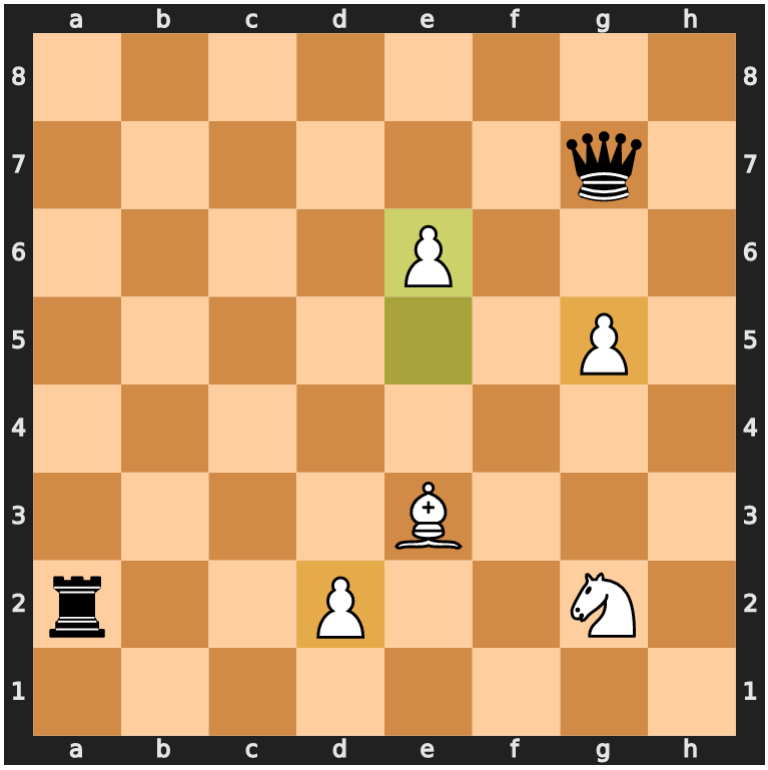
```
    display(Markdown("### Initial position"))
    display(board_image(current, display_size=size))

    for number, move in enumerate(moves, start=1):
        assert move in current.pseudo_legal_moves
        current.push(move)
        current.turn = chess.WHITE

        threatened = threatened_white_squares(current)
        names = sorted(chess.square_name(square) for square in threatened)
        annotation = ", ".join(names) if names else "none"
        fill = {square: THREATENED_COLOR for square in threatened}

        display(
            Markdown(
                f"\n### Move {number}: `{move.uci()}`\n\n"
                f"Threatened and defended White pieces: {annotation}.\n"
            )
        )
        display(
            board_image(
                current,
                display_size=size,
                lastmove=move,
                fill=fill,
            )
        )

        # Every displayed post-move state should satisfy the constraint.
        assert is_safe(current)

    assert is_goal(current)


display_solution(board, solution)
```

Initial position



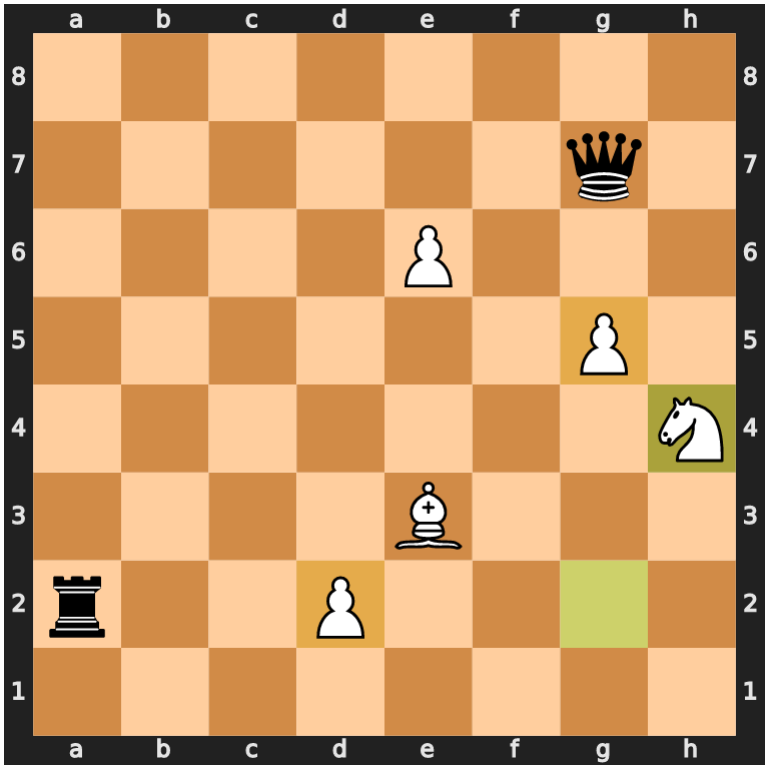
Move 1: `e5e6`

Threatened and defended White pieces: d2, g5.



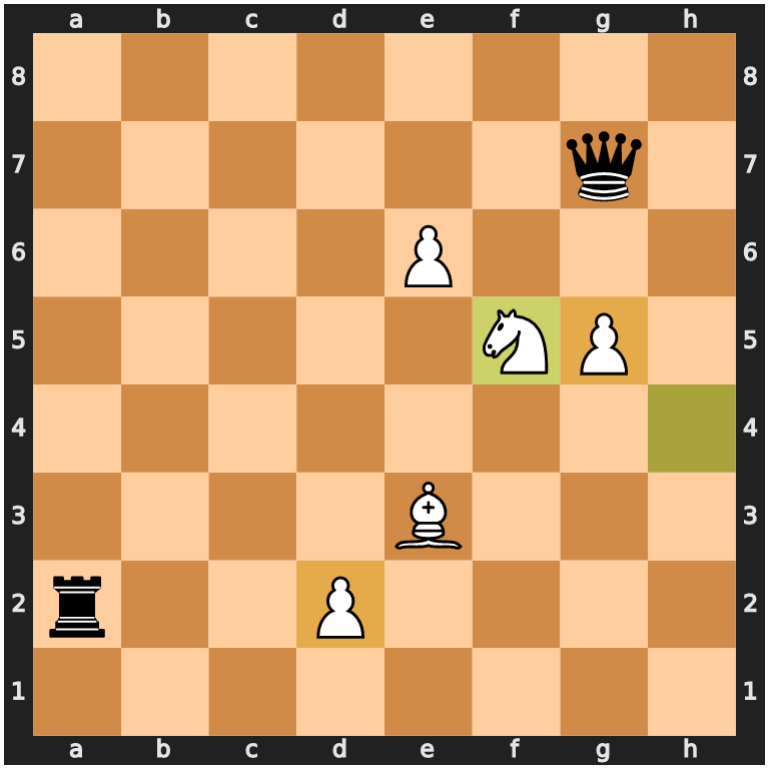
Move 2: `g2h4`

Threatened and defended White pieces: d2, g5.



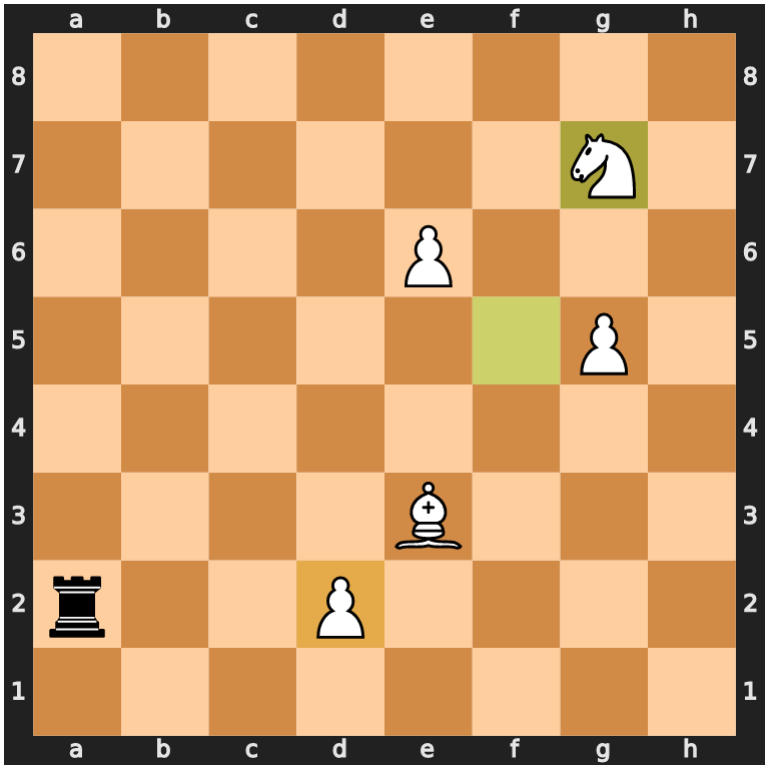
Move 3: `h4f5`

Threatened and defended White pieces: d2, g5.



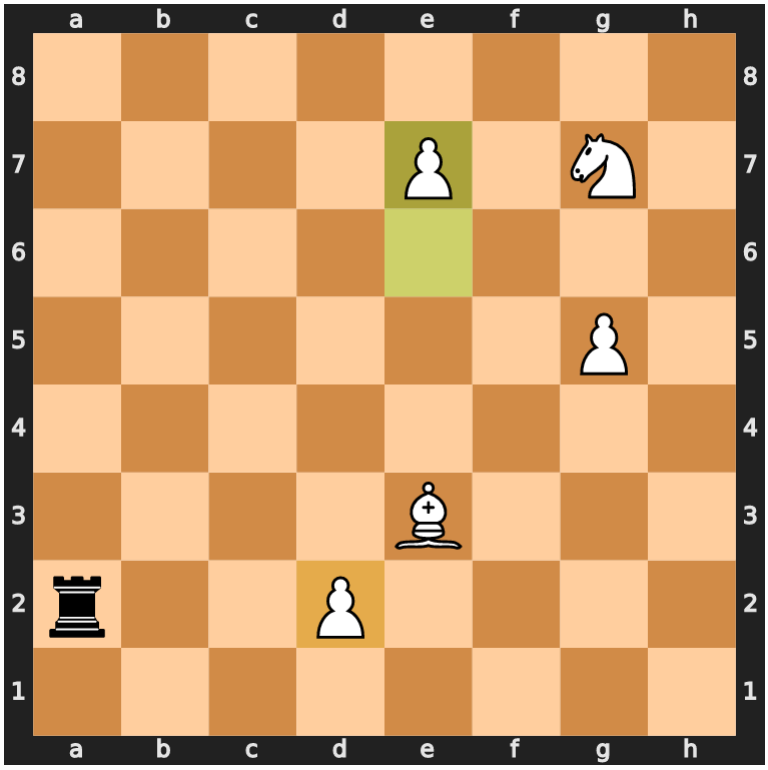
Move 4: `f5g7`

Threatened and defended White pieces: d2.



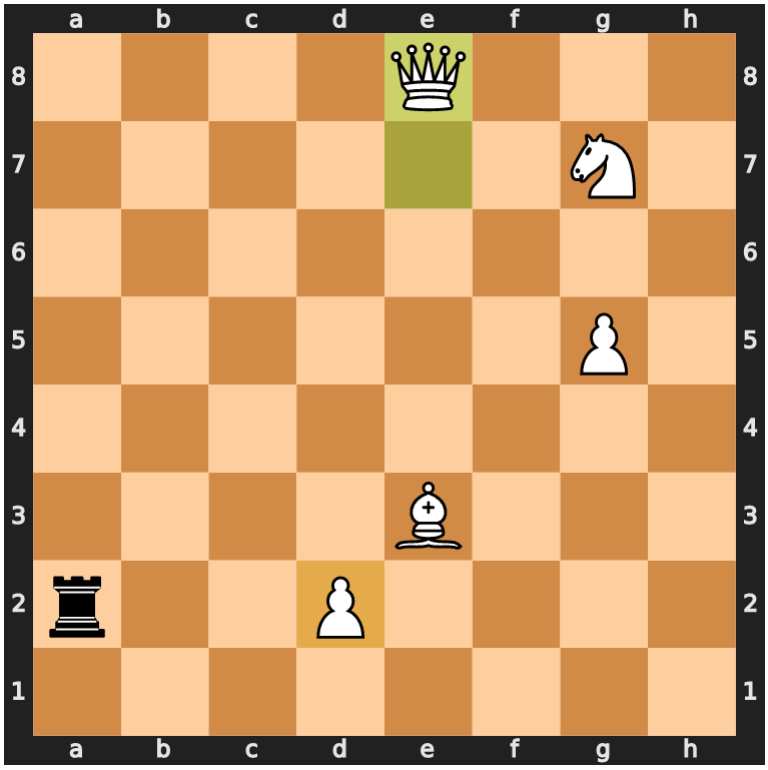
Move 5: e6e7

Threatened and defended White pieces: d2.



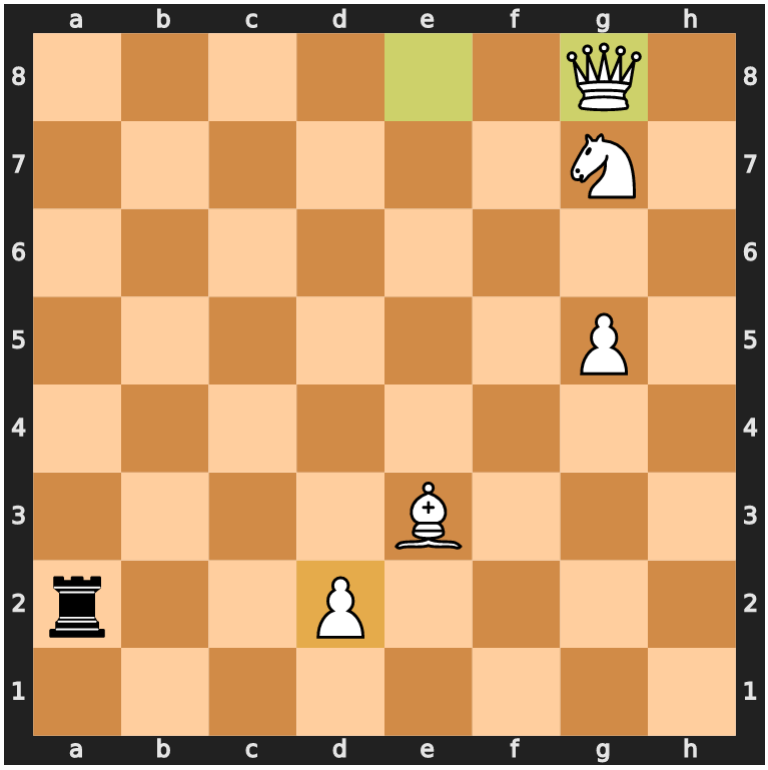
Move 6: `e7e8q`

Threatened and defended White pieces: d2.



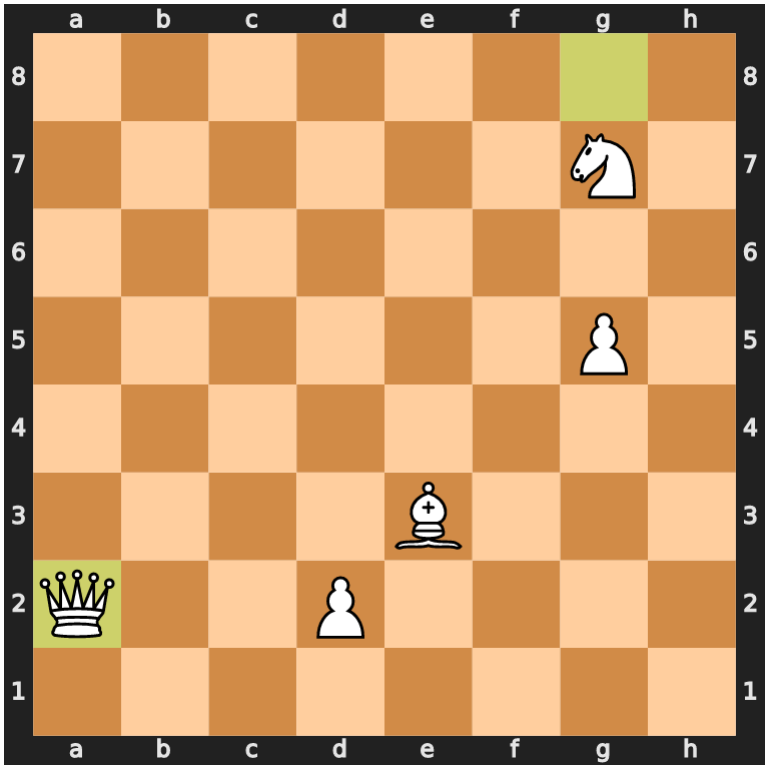
Move 7: e8g8

Threatened and defended White pieces: d2.



Move 8: `g8a2`

Threatened and defended White pieces: none.



Discussion prompts

1. What would happen without `reached` if a knight moved from one square to a second square and later returned?
2. Why does BFS minimize the number of moves for this puzzle?

Extensions for later lectures

- **Lecture 17 — Informed search:** How would the algorithm change if moves had different costs? What board information might make a useful heuristic for greedy best-first search or A*?
- **Lecture 22 — Adversarial search:** How would the problem change if Black were allowed to choose a reply?

The last question provides a bridge from single-agent search to adversarial search: once Black chooses moves too, the natural next models are minimax and alpha-beta pruning rather than BFS over a passive environment.