Here's a structured approach to implement advanced heuristics in your chess engine:

---

1. Positional Bonuses

Goal: Reward pieces for occupying strong squares (e.g., central control for knights).
Implementation:

- Create **piece-square tables** to assign positional scores for each piece type.
- Example for knights (higher values for central squares):

```
KNIGHT_TABLE = [
    [-5, -4, -3, -3, -3, -3, -4, -5],
    [-4, -2,  0,  0,  0,  0, -2, -4],
    [-3,  0,  1,  1,  1,  1,  0, -3],
    [-3,  0,  1,  2,  2,  1,  0, -3],
    [-3,  0,  1,  2,  2,  1,  0, -3],
    [-3,  0,  1,  1,  1,  1,  0, -3],
    [-4, -2,  0,  0,  0,  0, -2, -4],
    [-5, -4, -3, -3, -3, -3, -4, -5]
]
```

- Add positional scores to the evaluation:

```
def evaluate(board):
    score = material_score(board)  # Existing material calculation
    for square in chess.SQUARES:
        piece = board.piece_at(square)
        if not piece:
            continue
        # Get positional bonus based on piece type and square
        if piece.piece_type == chess.KNIGHT:
            rank = chess.square_rank(square)
            file = chess.square_file(square)
            score += KNIGHT_TABLE[rank][file] * (1 if piece.color
== chess.WHITE else -1)
    return score
```

PROF

---

2. Pawn Structure Analysis

Goal: Penalize weak pawn structures (doubled, isolated pawns) and reward strong ones (passed pawns).
Implementation:

- **Doubled Pawns:** Two pawns of the same color on the same file.
- **Isolated Pawns:** Pawns with no friendly pawns on adjacent files.
- **Passed Pawns:** Pawns with no opposing pawns ahead on the same or adjacent files.

```python
def evaluate_pawn_structure(board, color):
    pawns = board.pieces(chess.PAWN, color)
    doubled = isolated = passed = 0

    # Check for doubled pawns
    files = [chess.square_file(p) for p in pawns]
    for file in files:
        if files.count(file) > 1:
            doubled += 1

    # Check for isolated pawns
    for pawn in pawns:
        adjacent_files = [chess.square_file(pawn) - 1,
chess.square_file(pawn) + 1]
        if not any(f in files for f in adjacent_files if 0 <= f <
8):
            isolated += 1

    # Check for passed pawns
    for pawn in pawns:
        is_passed = True
        opposing_pawns = board.pieces(chess.PAWN, not color)
        for opp_pawn in opposing_pawns:
            if (chess.square_rank(opp_pawn) <
chess.square_rank(pawn) and  # For white
                abs(chess.square_file(opp_pawn) -
chess.square_file(pawn)) <= 1):
                is_passed = False
        if is_passed:
            passed += 1

    return (-50 * doubled) + (-30 * isolated) + (20 * passed)
```

PROF

3. King Safety

Goal: Penalize exposed kings and reward castling/pawn shields.
Implementation:

- Check if the king is castled.
- Count pawns around the king for protection.

```python
def evaluate_king_safety(board, color):
    king_square = board.king(color)
    safety = 0

    # Bonus for castling
    if board.has_castling_rights(color):
        safety += 25
    # Penalize exposed kings (no pawn shield)
    pawn_shield = 0
    for delta in [-1, 0, 1]:
        file = chess.square_file(king_square) + delta
        if 0 <= file < 8:
            square = chess.square(file,
chess.square_rank(king_square) + (1 if color == chess.WHITE else
-1))
            if board.piece_at(square) == chess.Piece(chess.PAWN,
color):
                pawn_shield += 1
    safety += 15 * pawn_shield

    return safety
```

---

4. Mobility

Goal: Reward players with more legal moves.
Implementation:

- Calculate the number of legal moves for each player.

```
def evaluate_mobility(board):
    if board.turn == chess.WHITE:
        return len(list(board.legal_moves)) * 0.1  # Weight = 0.1
    else:
        return -len(list(board.legal_moves)) * 0.1
```

---

5. Combined Evaluation Function

PROF

Integrate all heuristics with weights:

```
def evaluate(board):
    # Material score (existing)
    material = material_score(board)

    # Positional bonuses
    positional = positional_score(board)

    # Pawn structure
    white_pawns = evaluate_pawn_structure(board, chess.WHITE)
    black_pawns = evaluate_pawn_structure(board, chess.BLACK)
    pawn_score = white_pawns - black_pawns

    # King safety
    white_king = evaluate_king_safety(board, chess.WHITE)
    black_king = evaluate_king_safety(board, chess.BLACK)
    king_score = white_king - black_king

    # Mobility
    mobility = evaluate_mobility(board)

    # Weighted sum
    total = (
        material * 1.0 +
        positional * 0.5 +
        pawn_score * 0.3 +
        king_score * 0.2 +
        mobility
    )
    return total
```

---

6. Optimization Tips

1. **Precompute Data:** Cache pawn structure or king safety calculations during board updates.
2. **Use Bitboards:** Leverage `python-chess`'s bitboard operations for faster piece checks.
3. **Iterative Testing:** Validate heuristics by testing against positions with known outcomes.
4. **Adjust Weights:** Experiment with weights (e.g., positional bonuses vs. material).

Example Test Case

Position: White has a knight on e5 (centralized) vs. Black's knight on a8 (edge).
Expected Behavior: The engine should prefer the centralized knight due to positional bonuses.

By implementing these heuristics, your engine will make more human-like strategic decisions! ♟