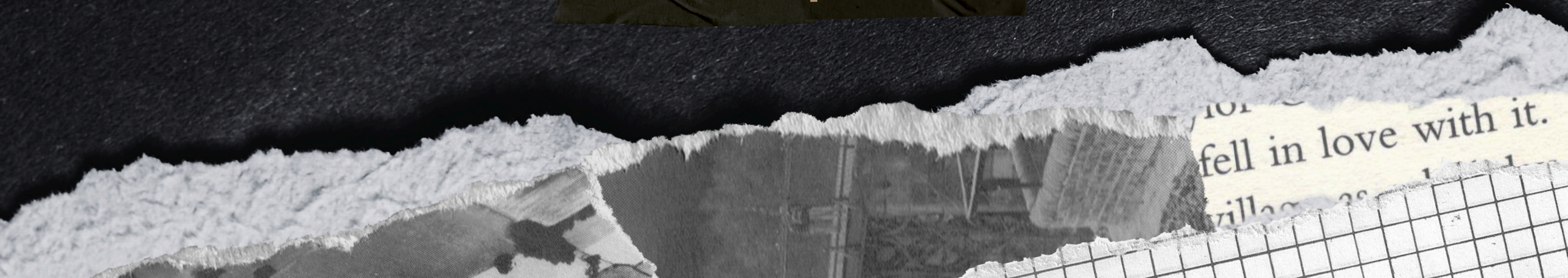
Hunt the Wumpus

Group 5



GROUP MEMBERS

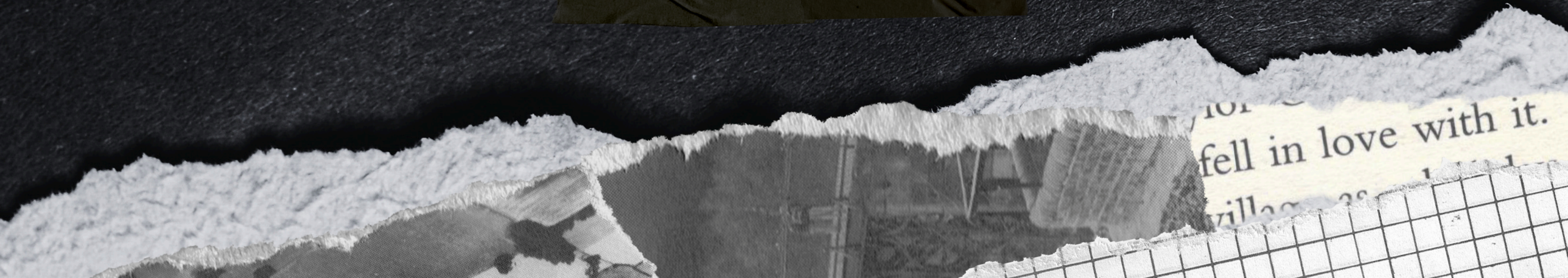
ANGELICA SUTI WHIHARTO
(001202300195)

INTAN KUMALA PASYA
(001202300063)

MUH. FAKHRI HISYAM AKBAR
(001202300093)

TABLE OF CONTENT

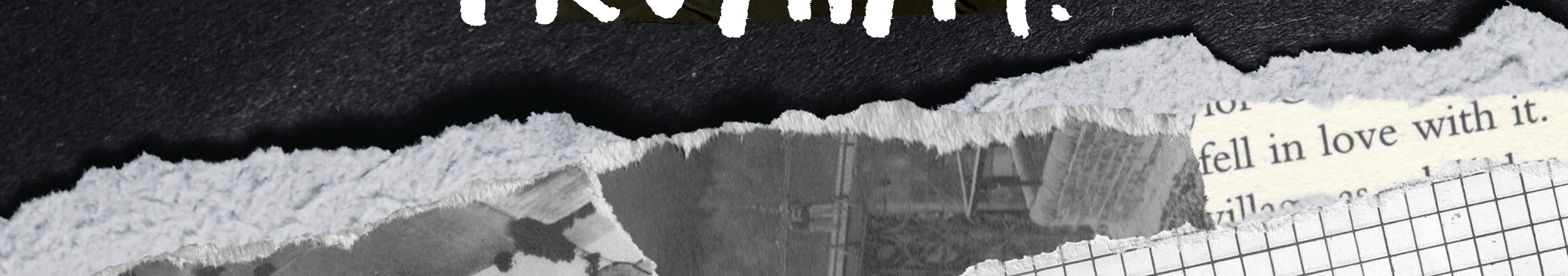
ABOUT THE GAME



ABOUT THE GAME

Hunt the Wumpus is a classic text-based adventure game from the early 1970s, often seen as an early example of procedural generation and strategic gameplay. The game was created by Gregory Yob in 1972. The goal is to explore a series of connected rooms or caves to hunt a mythical creature called the "Wumpus" while avoiding various hazards. It's often considered one of the early inspirations for AI in games due to the strategic elements players need to consider.

WHAT IS ALPHA-BETA PRUNING?



WHAT IS ALPHA - BETA PRUNING

Alpha - beta pruning is an optimization technique for the minimax algorithm. It reduces the number of nodes evaluated in the game tree by eliminating branches that cannot influence the final decision. This is achieved by maintaining two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively.

- Alpha: The best (highest) value that the maximizer can guarantee given the current state.
- Beta: The best (lowest) value that the minimizer can guarantee given the current state.

ALPHA-BETA PRUNING

```python
def alpha_beta_pruning(state: GameState, depth: int, alpha: float, beta: float, maximizing: bool) -> Tuple[int, Optional[Tuple[int, int]]]:
    if depth == 0 or state.game_over:
        return evaluate_state(state, depth), None

    valid_moves = get_valid_moves(state)

    if maximizing:
        max_eval = float('-inf')
        best_move = None
        for move in valid_moves:
            new_state = deepcopy(state)
            new_state.player_pos = list(move)
            new_state.score -= 1  # Cost for moving
            new_state.visited.add(tuple(move))

            # Check what's in the new position
            x, y = move
            if new_state.grid[x][y] == 'W' and not new_state.wumpus_killed:
                new_state.score += 500
                new_state.wumpus_killed = True
                new_state.player_armed = True  # Player becomes armed after killing Wumpus
                new_state.grid[x][y] = ' '  # Remove Wumpus from the grid
            elif new_state.grid[x][y] == 'G' and not new_state.gold_collected:
                new_state.score += 1000
                new_state.gold_collected = True
                new_state.player_armed = False  # Player becomes unarmed after collecting Gold
                new_state.grid[x][y] = ' '  # Remove Gold from the grid
            elif new_state.grid[x][y] == 'P':
                new_state.score -= 1000
                new_state.game_over = True
                new_state.won = False
```

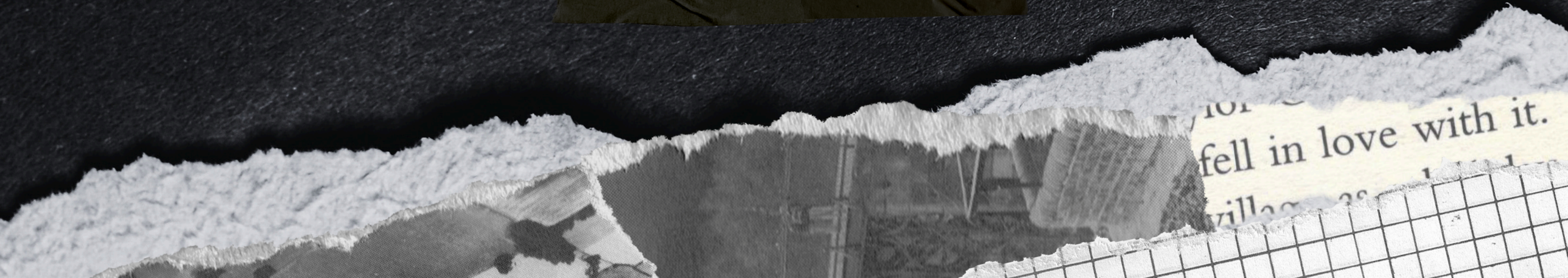
ALPHA-BETA PRUNING

```
            # Check win condition
            if new_state.wumpus_killed and new_state.gold_collected and new_state.player_pos == new_state.start_pos:
                new_state.game_over = True
                new_state.won = True
                new_state.score += 2000  # Bonus for completing all objectives

            eval_score, _ = alpha_beta_pruning(new_state, depth - 1, alpha, beta, False)
            if eval_score > max_eval:
                max_eval = eval_score
                best_move = move
            alpha = max(alpha, eval_score)
            if beta <= alpha:
                break
        return max_eval, best_move
    else:
        min_eval = float('inf')
        best_move = None
        for move in valid_moves:
            new_state = deepcopy(state)
            new_state.player_pos = list(move)
            new_state.score -= 1
            new_state.visited.add(tuple(move))

            eval_score, _ = alpha_beta_pruning(new_state, depth - 1, alpha, beta, True)
            if eval_score < min_eval:
                min_eval = eval_score
                best_move = move
            beta = min(beta, eval_score)
            if beta <= alpha:
                break
        return min_eval, best_move
```

EXPLANATION



EXPLANATION

```python
def alpha_beta_pruning(state: GameState, depth: int, alpha: float, beta: float, maximizing: bool) -> Tuple[int, Optional[Tuple[int, int]]]:
    if depth == 0 or state.game_over:
        return evaluate_state(state, depth), None

    valid_moves = get_valid_moves(state)
```

This part is Function signature and Base case.

- State : this is for holds information player, score, grid, etc.
- Depth: This is used for how deep of search tree, limiting how far algorithm works
- Alpha & Beta : This is used for the best score that maximizing and minimizing player can guarantee so far
- Maximizing:Bool : Indicates if the current player is trying to maximize or minimize the score
- Tuple : This used for best evaluation score and best move (If found)

Base Case :
If the search depth reaches zero or the game has ended (state.game_over), the algorithm halts recursion. It then assesses the game state using evaluate_state() and returns the score, along with None for the move, since no further moves are available.

Valid Moves :
If the search depth is zero or the game is over (state.game_over), the algorithm stops, evaluates the state with evaluate_state(), and returns the score with None for the move. Otherwise, it retrieves all valid moves from the current position using get_valid_moves(state).

EXPLANATION

```python
if maximizing:
    max_eval = float('-inf')
    best_move = None
    for move in valid_moves:
        new_state = deepcopy(state)
        new_state.player_pos = list(move)
        new_state.score -= 1  # Cost for moving
        new_state.visited.add(tuple(move))
```

Maximizing Player's Turn & Loop Over Valid Moves

- If it is the maximizing player's turn, the function initializes max_eval to the lowest possible value (-inf) to track the highest score and best_move to None.

Loop Over Valid Moves

- For each valid move, a deep copy of the current state (new_state) is created to simulate the move. The player's position (new_state.player_pos) is updated, 1 point is subtracted from the score for making the move, and the new position is marked as visited by adding it to the visited set.

EXPLANATION

```python
# Check what's in the new position
x, y = move
if new_state.grid[x][y] == 'W' and not new_state.wumpus_killed:
    new_state.score += 500
    new_state.wumpus_killed = True
    new_state.player_armed = True  # Player becomes armed after killing Wumpus
    new_state.grid[x][y] = ' '  # Remove Wumpus from the grid
elif new_state.grid[x][y] == 'G' and not new_state.gold_collected:
    new_state.score += 1000
    new_state.gold_collected = True
    new_state.player_armed = False  # Player becomes unarmed after collecting Gold
    new_state.grid[x][y] = ' '  # Remove Gold from the grid
elif new_state.grid[x][y] == 'P':
    new_state.score -= 1000
    new_state.game_over = True
    new_state.won = False
```

Handling Special Cases (Wumpus, Gold, Pits)

- Wumpus (W): If the player encounters the Wumpus and hasn't killed it yet, the player defeats the Wumpus, earning 500 points, becoming armed, and the Wumpus is removed from the grid.
- Gold (G): Upon discovering uncollected gold, the player earns 1000 points, collects the gold, and becomes unarmed. The gold is then removed from the grid.
- Pit (P): If the player falls into a pit, they lose 1000 points, and the game ends with the player losing (new_state.game_over = True).

EXPLANATION

```python
# Check win condition
if new_state.wumpus_killed and new_state.gold_collected and new_state.player_pos == new_state.start_pos:
    new_state.game_over = True
    new_state.won = True
    new_state.score += 2000  # Bonus for completing all objectives
```

Check for Win Condition

- If the player has killed the Wumpus, collected the gold, and returned to the starting position, they win the game. The game ends, and the player receives a 2000-point bonus for completing all objectives.

EXPLANATION

```
eval_score, _ = alpha_beta_pruning(new_state, depth - 1, alpha, beta, False)
if eval_score > max_eval:
    max_eval = eval_score
    best_move = move
alpha = max(alpha, eval_score)
if beta <= alpha:
    break
```

Recursion with Alpha-Beta Pruning

- The algorithm recursively calls alpha_beta_pruning with the updated game state (new_state), reduced depth (depth - 1), and for the minimizing player (maximizing=False). If the evaluation score of the current move (eval_score) exceeds max_eval, it updates max_eval and records the move as best_move. Alpha is then updated to the maximum of alpha and eval_score. If beta is less than or equal to alpha, the algorithm prunes this branch to stop further exploration, as better moves are unlikely.

EXPLANATION

```
else:
    min_eval = float('inf')
    best_move = None
    for move in valid_moves:
        new_state = deepcopy(state)
        new_state.player_pos = list(move)
        new_state.score -= 1
        new_state.visited.add(tuple(move))

        eval_score, _ = alpha_beta_pruning(new_state, depth - 1, alpha, beta, True)
        if eval_score < min_eval:
            min_eval = eval_score
            best_move = move
        beta = min(beta, eval_score)
        if beta <= alpha:
            break
```

Minimizing Player's Turn

- If it's the minimizing player's turn (e.g., simulating an opponent or trying to minimize bad outcomes), min_eval is initialized to the highest possible value (inf), and best_move is set to None.

Loop For Minimizing Player

- This process works similarly to the maximizing player but focuses on minimizing the evaluation score. For each valid move, the algorithm deep-copies the current state, updates the player's position, deducts 1 point from the score, and marks the position as visited. It then recursively calls alpha_beta_pruning with maximizing=True, switching back to the maximizing player. If a smaller evaluation score is found, min_eval is updated, and best_move is set to the current move. Beta is updated to the minimum of beta and eval_score. If beta <= alpha, the algorithm prunes further branches.

EXPLANATION

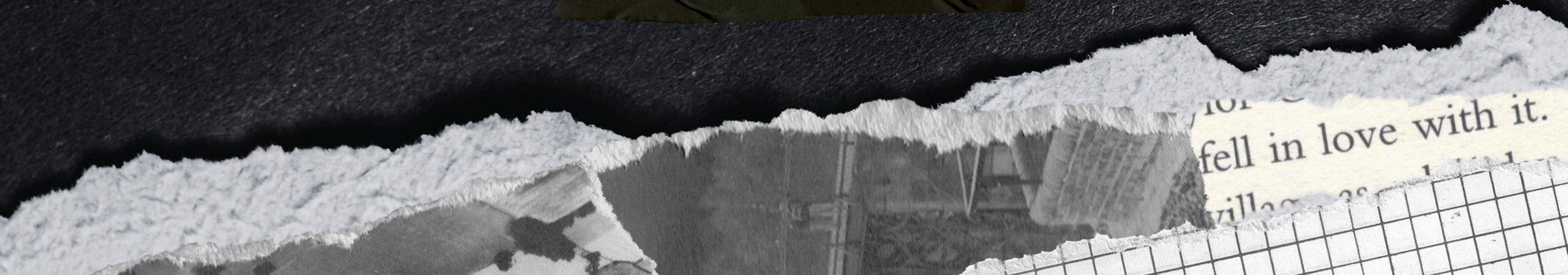
```
return max_eval, best_move
```

```
return min_eval, best_move
```

Return Best Move

- After evaluating all valid moves, the function returns the best evaluation score, which is either `min_eval` for the minimizing player or `max_eval` for the maximizing player, along with the corresponding `best_move`.

OUTPUT



OUTPUT

Hunt The Wumpus with Alpha Beta Pruning

Score: 978

	S				
S					
		B			
	B		B		
		B			

Wumpus Killed: No | Gold Collected: Yes

Hunt The Wumpus with Alpha Beta Pruning

Score: -4

S		B			
	S		B		
S		B			
					
				B	
					B

Wumpus Killed: No | Gold Collected: No

Hunt The Wumpus with Alpha Beta Pruning

Score: 3480

S		B			
	S		B		
S		B			
		You Won!			
				B	
					B

Wumpus Killed: Yes | Gold Collected: Yes

THANK YOU