```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>

#define N 3
#define MAX_STATES 362880 // 9!
```

- **`#include` Statements**: These include standard C libraries (`stdio.h`, `stdlib.h`, `stdbool.h`) for input/output, memory allocation, boolean types, and constants (`limits.h`) for integer limits.
- **`#define` Directives**: Defines `N` as 3 (size of the puzzle board, which is 3x3) and `MAX_STATES` as 362880 (the factorial of 9, representing the maximum number of possible states in the 8-puzzle).

```
c
Copy code
typedef struct {
   int board[N][N];
   int blank_row;
   int blank_col;
   int cost;
} State;
```

- **`State` Structure**: Defines a structure `State` which holds:
  - `board[N][N]`: 2D array to represent the puzzle board.
  - `blank_row`, `blank_col`: Coordinates of the blank tile.
  - `cost`: Cost associated with reaching this state.

```
c
Copy code
typedef struct {
   State* state;
   int priority;
} PQNode;
```

- **`PQNode` Structure**: Defines a structure `PQNode` which holds:
  - `state`: Pointer to a `State` structure.
  - `priority`: Priority value associated with this state.

```
c
Copy code
typedef struct {
   PQNode* nodes[MAX_STATES];
   int size;
} PriorityQueue;
```

- **`PriorityQueue` Structure**: Defines a structure `PriorityQueue` which holds:
  - `nodes[MAX_STATES]`: Array of pointers to `PQNode` structures (acts as the priority queue).
  - `size`: Current number of elements in the priority queue.

```
c
Copy code
State* initializeState(int initial[N][N]) {
```

```
    State* state = (State*)malloc(sizeof(State));
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            state->board[i][j] = initial[i][j];
            if (initial[i][j] == 0) {
                state->blank_row = i;
                state->blank_col = j;
            }
        }
    }
    state->cost = 0;
    return state;
}
```

- **`initializeState` Function**: Initializes a `State` structure with an initial configuration of the puzzle board:
  - Allocates memory for a new `State`.
  - Copies the initial board configuration.
  - Determines the position of the blank tile (`0` in the board configuration).
  - Sets initial cost to `0`.
  - Returns a pointer to the initialized `State`.

```
c
Copy code
bool isGoalState(State* state) {
    int count = 1;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (state->board[i][j] != count % (N*N)) {
                return false;
            }
            count++;
        }
    }
    return true;
}
```

- **`isGoalState` Function**: Checks if the current state matches the goal configuration:
  - Initializes a `count` variable to track expected tile values.
  - Iterates through the board and checks if each tile matches the expected value (`count % (N*N)`).
  - Returns `true` if the board matches the goal state, otherwise `false`.

```
c
Copy code
int calculateHammingPriority(State* state) {
    int priority = 0;
    int count = 1;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (state->board[i][j] != count % (N*N) && state->board[i][j] !=
0) {
                priority++;
            }
            count++;
        }
    }
```

```
    return priority;
}
```

- **`calculateHammingPriority` Function**: Computes the Hamming priority (number of misplaced tiles) for a given state:
    - Initializes `priority` to `0`.
    - Uses `count` to track expected tile values.
    - Counts mismatches (`priority++`) between the current board state and the expected value for each tile.
    - Returns the computed Hamming priority.

```
c
Copy code
int calculateManhattanDistance(int value, int row, int col) {
    int goal_row = (value - 1) / N;
    int goal_col = (value - 1) % N;
    return abs(goal_row - row) + abs(goal_col - col);
}
```

- **`calculateManhattanDistance` Function**: Computes the Manhattan distance (sum of horizontal and vertical distances from the current position to the goal position) for a specific tile:
    - Calculates the goal position (`goal_row`, `goal_col`) for the given `value`.
    - Computes and returns the Manhattan distance between the current position (`row`, `col`) and the goal position.

```
c
Copy code
int calculateManhattanPriority(State* state) {
    int priority = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (state->board[i][j] != 0) {
                priority += calculateManhattanDistance(state->board[i][j],
i, j);
            }
        }
    }
    return priority;
}
```

- **`calculateManhattanPriority` Function**: Computes the Manhattan priority (sum of Manhattan distances of all tiles from their goal positions) for a given state:
    - Initializes `priority` to `0`.
    - Iterates through the board and computes the Manhattan distance for each non-blank tile.
    - Accumulates the total priority and returns it.

```
c
Copy code
void printBoard(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", board[i][j]);
        }
```

```
        printf("\n");
    }
    printf("\n");
}
```

- **`printBoard` Function**: Prints the current state of the board:
    - Iterates through the board and prints each element followed by a space.
    - Prints a newline after each row and an additional newline at the end to separate boards.

```
c
Copy code
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

- **`swap` Function**: Swaps two integers:
    - Takes pointers to two integers (`a` and `b`).
    - Swaps their values using a temporary variable `temp`.

```
c
Copy code
PriorityQueue* createPriorityQueue() {
    PriorityQueue* pq = (PriorityQueue*)malloc(sizeof(PriorityQueue));
    pq->size = 0;
    return pq;
}
```

- **`createPriorityQueue` Function**: Creates and initializes a priority queue:
    - Allocates memory for a new `PriorityQueue`.
    - Initializes `size` to `0`.
    - Returns a pointer to the created `PriorityQueue`.

```
c
Copy code
void push(PriorityQueue* pq, State* state, int priority) {
    PQNode* newNode = (PQNode*)malloc(sizeof(PQNode));
    newNode->state = state;
    newNode->priority = priority;
    pq->nodes[pq->size++] = newNode;
}
```

- **`push` Function**: Adds a state to the priority queue with a specified priority:
    - Allocates memory for a new `PQNode`.
    - Sets `state` and `priority` for the new node.
    - Adds the new node to `pq->nodes` array.
    - Increments the `size` of the priority queue.

```
c
Copy code
State* pop(PriorityQueue* pq) {
    int highestPriority = INT_MAX;
    int idx = -1;
```

```
    for (int i = 0; i < pq->size; i++) {
        if (pq->nodes[i]->priority < highestPriority) {
            highestPriority = pq->nodes[i]->priority;
            idx = i;
        }
    }
    State* state = pq->nodes[idx]->state;
    free(pq->nodes[idx]);
    for (int i = idx; i < pq->size - 1; i++) {
        pq->nodes[i] = pq->nodes[i + 1];
    }
    pq->size--;
    return state;
}
```

- **`pop` Function**: Removes and returns the state with the highest priority from the priority queue:
    - Initializes `highestPriority` to `INT_MAX` (maximum integer value).
    - Finds the index (`idx`) of the node with the highest priority.
    - Retrieves the `state` from `pq->nodes[idx]`.
    - Frees the memory allocated for the node.
    - Adjusts the priority queue array (`nodes`) and decreases `size`.
    - Returns the popped `state`.

```
c
Copy code
bool isEmpty(PriorityQueue* pq) {
    return pq->size == 0;
}
```

- **`isEmpty` Function**: Checks if the priority queue is empty:
    - Returns `true` if the `size` of `pq` is `0`, indicating an empty queue; otherwise, returns `false`.

```
c
Copy code
void aStarSearch(State* initialState) {
    PriorityQueue* openSet = createPriorityQueue();
    push(openSet, initialState, initialState
```