15 Puzzle Solver in C Using Literate Programming

Keno Goertz

March 26, 2021

Contents

1 Introduction

The *15 puzzle* is a game where 15 numbered square tiles are arranged on a 4×4 grid. They can be moved to fill the empty square. The objective of the game is to arrange the tiles so that they are in order, as shown in Fig. 1.

This program takes a given board constellation as its input and returns the moves required to solve the puzzle in the least number of moves. For this, the possible board constellations are represented as nodes in a directed graph. Two nodes are connected by two edges in the graph if you can get from one node to the other with a legal move in the game—there are always two edges, since you can invariably go back to

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 1: The solved 15 puzzle

the previous node by moving the tile in the direction opposite to that in which it has just been moved. The edges have weights of either 0 or 1, depending on whether the tile being moved is getting closer to its final position, or getting further away from it. Combined with a few properties to be described in later sections, this will be enough to find the solution with the least number of moves.

Overview The following listing shows the basic structure of our program. First, we include the needed libraries. Then, we implement data structures and functions for playing the 15 puzzle game. After that, we implement functions which abstract board constellations into the previously mentioned graph representation. Finally, we tie things together by searching through the paths on the graph until finding the solution.

2a ⟨*15_puzzle_solver.c* 2a⟩≡
  ⟨*includes* 2b⟩
  ⟨*implementing the game* 2c⟩
  ⟨*graph representation* 7b⟩
  ⟨*searching the graph* 9a⟩
  ⟨*example main function* 18⟩

Includes We only use standard libraries.

2b ⟨*includes* 2b⟩≡ (2a)
```
#include <stdio.h>
#include <stdlib.h>
```

2 Implementing the game

Before we start writing a solver, we need to implement data structures to hold game information and functions to play it.

2c ⟨*implementing the game* 2c⟩≡ (2a)
  ⟨*constants of the game implementation* 2d⟩
  ⟨*data structures of the game* 3a⟩
  ⟨*functions for playing the game* 4c⟩

2.1 Constants of the game implementation

Although our primary focus is on the classical 4×4 game, we want to allow alternate board sizes. This can be achieved by simply adjusting the constants here.

2d ⟨*constants of the game implementation* 2d⟩≡ (2c)
```
#define BOARD_ROWS 4
#define BOARD_COLUMNS 4
```

2.2 Data structures of the game

We will need a data structure to describe the four possible moves. We also need a data structure holding the board constellation. Finally, we write a function to initialize the board to its solved state.

3a ⟨*data structures of the game* 3a⟩≡ (2c)
```
⟨data structure for moves 3b⟩
⟨data structure for the board 3c⟩
⟨malloc helper function 3d⟩
⟨initializing the board 4b⟩
```

Data structure for moves We use an enum to refer to the possible moves. The direction of the move describes in which direction a tile moves to fill up the previous location of the *empty tile*. We refer to the positions on the board as *squares*, and to the movable numbered tiles as *tiles*. We also use the concept of an *empty tile*, which is not really a tile at all, but rather finds itself at the square where no real tile is present.

3b ⟨*data structure for moves* 3b⟩≡ (3a)
```
enum move
  {
    UP, DOWN, LEFT, RIGHT, NONE
  };
```

Data structure for the board The data structure for the board consists of a two-dimensional array of unsigned integers. This array contains the tiles at the indices of the squares of the board. The empty tile gets assigned the number 0. The data structure also contains the row and column index of the empty tile, because it is inefficient to search through the array for it every time we want to make a move.

3c ⟨*data structure for the board* 3c⟩≡ (3a)
```
struct board
{
  unsigned tiles[BOARD_ROWS][BOARD_COLUMNS];
  size_t empty_tile_row;
  size_t empty_tile_column;
};
```

`malloc` helper function If `malloc` ever fails, we just want to print an error message about this and exit the program. We are going to let a function call `malloc` and do the check by itself so that we don't have to put the checks everywhere in our code.

3d ⟨*malloc helper function* 3d⟩≡ (3a) 4a ▻
```
void *safe_malloc (size_t size)
{
  void *ptr = malloc (size);
  if (!ptr)
    {
      fprintf (stderr, "malloc failed\n");
      exit (EXIT_FAILURE);
    }
  return ptr;
}
```

We implement the same for `realloc`.

4a ⟨*malloc helper function* 3d⟩+≡ (3a) ◁ 3d
```
void *safe_realloc (void *ptr, size_t size)
{
  ptr = realloc (ptr, size);
  if (!ptr)
    {
      fprintf (stderr, "realloc failed\n");
      exit (EXIT_FAILURE);
    }
  return ptr;
}
```

Initializing the board We also write a function to initialize our board struct. It returns a pointer to a board that has the tiles arranged in the solution of the puzzle.

4b ⟨*initializing the board* 4b⟩≡ (3a)
```
struct board *board_init ()
{
  struct board *board = safe_malloc (sizeof (struct board));
  for (size_t row = 0; row < BOARD_ROWS; row++)
    {
      for (size_t column = 0; column < BOARD_COLUMNS; column++)
        {
          board->tiles[row][column] = (row * BOARD_COLUMNS) + column + 1;
        }
    }
  board->tiles[BOARD_ROWS - 1][BOARD_COLUMNS - 1] = 0;
  board->empty_tile_row = BOARD_ROWS - 1;
  board->empty_tile_column = BOARD_COLUMNS - 1;
  return board;
}
```

2.3 Functions for playing the game

We need to implement the following functions to play the game on our board struct.

4c ⟨*functions for playing the game* 4c⟩≡ (2c)

⟨*checking if a move is allowed* 5a⟩
⟨*getting the position of the tile that should be moved* 5b⟩
⟨*making moves on the board* 6a⟩
⟨*printing the board* 7a⟩

2.3.1 Checking if a move is allowed

We need to check whether a certain move is allowed given the position of the empty tile. For example, we can't move down if the empty tile is in the top row.

5a ⟨*checking if a move is allowed* 5a⟩≡ (4c)

```
int board_check_move (struct board const *board, enum move move)
{
  size_t row = board->empty_tile_row;
  size_t column = board->empty_tile_column;
  if ((move == UP && row >= BOARD_ROWS - 1)
      || (move == DOWN && row < 1)
      || (move == LEFT && column >= BOARD_COLUMNS - 1)
      || (move == RIGHT && column < 1))
    {
      return 0;
    }
  return 1;
}
```

2.3.2 Getting the position of the tile that should be moved

We define a function to obtain the row and column of the tile that should be moved in a specified move.

5b ⟨*getting the position of the tile that should be moved* 5b⟩≡ (4c)

```
void board_get_moving_tile_pos (struct board const *b, enum move move,
                                size_t *row, size_t *column)
{
  *row = b->empty_tile_row;
  *column = b->empty_tile_column;
  switch (move)
    {
    case UP:
      *row += 1;
      break;
    case DOWN:
      *row -= 1;
      break;
    case LEFT:
      *column += 1;
      break;
    case RIGHT:
      *column -= 1;
      break;
    }
}
```

2.3.3 Making moves on the board

This function takes a board and a move as its input and returns the board after making the move by creating a new board struct. The original board struct remains unchanged. This is needed so that we can go back when our search along our graph has led us into hopeless territory—which happens in the majority of cases, since there is just one solution among the approximately 10^{13} board constellations. If something goes wrong (for example, the move is not allowed), a NULL pointer is returned.

6a ⟨*making moves on the board* 6a⟩≡ (4c)

```
struct board *board_make_move (struct board const *b, enum move move)
{
  ⟨do not make an unallowed move 6b⟩
  ⟨allocate and initialize the new board 6c⟩
  ⟨set the tiles of the new board and return it 6d⟩
}
```

Do not make an unallowed move If the specified move is not allowed, we abort the function and return NULL.

6b ⟨*do not make an unallowed move* 6b⟩≡ (6a)

```
if (!board_check_move (b, move))
  {
    return NULL;
  }
```

Allocate and initialize the new board We need to allocate the struct for the new board into memory and initially set it to be equal to the current board.

6c ⟨*allocate and initialize the new board* 6c⟩≡ (6a)

```
struct board *newb = safe_malloc (sizeof (struct board));
*newb = *b;
```

Set the tiles of the new board and return it Finally, we need to change the `tiles` array and the empty tile position of the new board to reflect on the move.

6d ⟨*set the tiles of the new board and return it* 6d⟩≡ (6a)

```
// Position of tile before the move
size_t prev_row, prev_column;
board_get_moving_tile_pos (b, move, &prev_row, &prev_column);

// Position of tile after the move
size_t row = b->empty_tile_row;
size_t column = b->empty_tile_column;

newb->tiles[row][column] = newb->tiles[prev_row][prev_column];
newb->tiles[prev_row][prev_column] = 0;
newb->empty_tile_row = prev_row;
newb->empty_tile_column = prev_column;

return newb;
```

2.3.4 Printing the board

Especially for debugging purposes, it is useful to be able to print the board.

7a ⟨*printing the board* 7a⟩≡ (4c)
```
void board_print (struct board const *board)
{
  printf ("[");
  for (size_t row = 0; row < BOARD_ROWS; row++)
    {
      printf ("[");
      for (size_t column = 0; column < BOARD_COLUMNS; column++)
        {
          printf ("%u, ", board->tiles[row][column]);
        }
      printf ("\b\b], ");
    }
  printf ("\b\b]\n");
}
```

3 Graph representation

We treat a board constellation as a node on a directed graph. The possible moves are represented by edges. The edges are labeled with a 0 if the moving tile is getting closer to its final position, and with a 1 if it is getting further away. We call these labels the *cost* of the move.

We have already defined a function to check whether a move is allowed. Now we need functions to get the cost of a move and to check whether a board is solved.

7b ⟨*graph representation* 7b⟩≡ (2a)
```
⟨getting the cost of a move 7c⟩
⟨checking whether a board is solved 8c⟩
```

3.1 Getting the cost of a move

This function returns −1 if the specified move is not allowed. If it is allowed, the cost of the move (either 0 or 1) is returned.

7c ⟨*getting the cost of a move* 7c⟩≡ (7b)
```
int board_get_move_cost (struct board const *board, enum move move)
{
  ⟨handle unallowed moves when getting the cost 7d⟩
  ⟨calculate various positions of the tile 8a⟩
  ⟨calculate the cost of moving the tile 8b⟩
}
```

Handle unallowed moves when getting the cost We return a cost of −1 (representing infinity) for unallowed moves.

7d ⟨*handle unallowed moves when getting the cost* 7d⟩≡ (7c)
```
if (!board_check_move (board, move))
  {
    return -1;
  }
```

Calculate various positions of the tile We first get the position of the tile that is to be moved. Then, we calculate the position that this tile would have in the solution of the puzzle. Finally, we calculate the position that the tile would have after the move.

8a ⟨*calculate various positions of the tile* 8a⟩≡ (7c)
```
size_t row, column;
board_get_moving_tile_pos (board, move, &row, &column);
unsigned tile = board->tiles[row][column];
size_t solved_row = (tile - 1) / BOARD_COLUMNS;
size_t solved_column = (tile - 1) % BOARD_COLUMNS;
size_t moved_row = board->empty_tile_row;
size_t moved_column = board->empty_tile_column;
```

Calculate the cost of moving the tile At last, we return a cost of 0 if the tile moves closer to its solved position, and a cost of 1 if it moves away from it.

8b ⟨*calculate the cost of moving the tile* 8b⟩≡ (7c)
```
if (abs (moved_row - solved_row) < abs (row - solved_row)
    || abs (moved_column - solved_column) < abs (column - solved_column))
  {
    return 0;
  }
return 1;
```

3.2 Checking whether a board is solved

This function simply returns 1 if the board is solved, i.e., every tile is at the correct position, and 0 if this is not the case.

8c ⟨*checking whether a board is solved* 8c⟩≡ (7b)
```
int board_is_solved (struct board const *board)
{
  for (size_t row = 0; row < BOARD_ROWS; row++)
    {
      for (size_t column = 0; column < BOARD_COLUMNS; column++)
        {
          unsigned solved_tile = ((row * BOARD_COLUMNS + column + 1)
                                  % (BOARD_ROWS * BOARD_COLUMNS));
          unsigned tile = board->tiles[row][column];
          if (tile != solved_tile)
            {
              return 0;
            }
        }
    }
  return 1;
}
```

4 Searching the graph

We will use the following properties to perform the graph search:

- The optimal solution is evidently the one that minimizes the number of moves with a cost of 1.
- If the board can be solved with only zero-cost moves, the numbers of moves required will be the sum of the *Manhattan distances* of all tiles to their positions in the solution. We will refer to this sum simply as the Manhattan distance d of the board.
- If n moves of cost 1 are needed to solve the board, the total number of moves will be $d + 2n$, since moving a tile away from its solved position requires an additional move to bring it closer again.

Thus, we first search all paths of zero cost with length d and see if one of them leads to the solution. If none of them do, we check all paths with cost 1 and length $d + 2$. If that fails, we check all paths with cost 2 and length $d + 4$, and so on.

9a ⟨*searching the graph* 9a⟩≡ (2a)
  ⟨*data structures of the search* 9b⟩
  ⟨*calculating the manhattan distance* 12a⟩
  ⟨*performing the search* 12b⟩

4.1 Data structures of the search

We will perform the graph search using a recursive function. This function will have several lists as its arguments, for example the moves performed so far and the added up cost alongside each move. For this, we need to implement a generic list data type.

9b ⟨*data structures of the search* 9b⟩≡ (9a)
  ⟨*list data type* 9c⟩
  ⟨*initializing the list* 10a⟩
  ⟨*destroying the list* 10b⟩
  ⟨*setting an element of the list* 10c⟩
  ⟨*getting an element of the list* 11a⟩
  ⟨*getting the last element of the list* 11b⟩
  ⟨*pushing back to the list* 11c⟩

List data type The following struct defines our generic list. We also use a constant for the initial size of our lists.

9c ⟨*list data type* 9c⟩≡ (9b)

```
#define LIST_INIT_SIZE 64

struct list
{
  void **elements;
  size_t length;
  size_t size;
  size_t sizeof_element;
};
```

Initializing the list This allocates a list and all of its void pointers into memory. We pass the size of a list element as the argument. For example, if we want a list of `int`, we pass `sizeof(int)` as the argument.

10a ⟨*initializing the list* 10a⟩≡ (9b)

```
struct list *list_init (size_t sizeof_element)
{
  struct list *list = safe_malloc (sizeof (struct list));
  list->length = 0;
  list->size = LIST_INIT_SIZE;
  list->elements = safe_malloc (sizeof (void*) * LIST_INIT_SIZE);
  list->sizeof_element = sizeof_element;
  for (size_t i = 0; i < LIST_INIT_SIZE; i++)
    {
      list->elements[i] = safe_malloc (sizeof_element);
    }
  return list;
}
```

Destroying the list This frees all the memory of a list.

10b ⟨*destroying the list* 10b⟩≡ (9b)

```
void list_destroy (struct list *list)
{
  for (size_t i = 0; i < list->size; i++)
    {
      free (list->elements[i]);
    }
  free (list->elements);
  free (list);
}
```

Setting an element of the list With this function, we can set an element of our list to the given value. We use `char` pointers to fill up the elements because a `char` has a size of one byte. If our data type is larger than one byte, we can hence use pointer arithmetic on the `char` pointers to fill up all bytes of the element.

10c ⟨*setting an element of the list* 10c⟩≡ (9b)

```
void list_set (struct list *list, size_t pos, void *value)
{
  if (pos < 0 || pos >= list->length)
    {
      fprintf (stderr, "Tried to set list element out of bounds\n");
      exit (EXIT_FAILURE);
    }
  for (size_t i = 0; i < list->sizeof_element; i++)
    {
      *((char*) list->elements[pos] + i) = *((char*) value + i);
    }
}
```

Getting an element of the list The following function enables us to get the value of a list element.

11a ⟨*getting an element of the list* 11a⟩≡ (9b)
```
void *list_get (struct list *list, size_t pos)
{
  if (pos < 0 || pos >= list->length)
    {
      fprintf (stderr, "Tried to get list element out of bounds\n");
      exit (EXIT_FAILURE);
    }
  return list->elements[pos];
}
```

Getting the last element of the list We will often want to get the last element of the list, so we define a function for this.

11b ⟨*getting the last element of the list* 11b⟩≡ (9b)
```
void *list_get_last (struct list *list)
{
  return list_get (list, list->length - 1);
}
```

Pushing back to the list This resizes the list if necessary and appends the value to its end.

11c ⟨*pushing back to the list* 11c⟩≡ (9b)
```
void list_push_back (struct list *list, void *value)
{
  size_t pos = list->length;
  if (pos >= list->size)
    {
      list->size *= 2;
      list->elements = safe_realloc (list->elements,
                                     list->size * sizeof (void*));
      for (size_t i = pos; i < list->size; i++)
        {
          list->elements[i] = safe_malloc (list->sizeof_element);
        }
    }
  list->length += 1;
  list_set (list, pos, value);
}
```

4.2 Calculating the Manhattan distance

For our algorithm to work as described in Sec. 4, we need to calculate the Manhattan distance d of the board.

12a ⟨*calculating the manhattan distance* 12a⟩≡ (9a)

```
unsigned manhattan_distance (struct board const *board)
{
  unsigned manhattan_distance = 0;
  for (size_t row = 0; row < BOARD_ROWS; row++)
    {
      for (size_t column = 0; column < BOARD_COLUMNS; column++)
        {
          unsigned tile = board->tiles[row][column];
          if (tile == 0)
            {
              continue;
            }
          size_t solved_row = (tile - 1) / BOARD_COLUMNS;
          size_t solved_column = (tile - 1) % BOARD_COLUMNS;
          manhattan_distance += abs (row - solved_row);
          manhattan_distance += abs (column - solved_column);
        }
    }
  return manhattan_distance;
}
```

4.3 Performing the search

To perform the search, we will use a recursive function that searches all moves for which the total cost stays below a certain maximum. Our main function will continuously increase this maximum cost until the recursive function is able to find a solution. We will also set up utility functions to be used by the recursive search.

12b ⟨*performing the search* 12b⟩≡ (9a)

```
⟨functions for the recursive search 17a⟩
⟨search recursively with maximum cost 14c⟩
⟨main search function 13a⟩
```

4.3.1 Main search function

The solution that minimizes the number of moves is also the one that minimizes the combined cost of all moves. Thus, we start searching for a solution with a cost of zero. Such a solution must have exactly *d* moves, where *d* is the Manhattan distance. If we don't find a solution, we continuously increase the total allowed cost until a solution is found. The number of moves of the solution will be exactly $d + 2n$ where n is the number of moves with cost 1 that are needed to solve the puzzle. Our recursive search function returns NULL if no solution can be found within these constraints for the cost, otherwise it returns a list of the moves that solve the puzzle.

13a ⟨*main search function* 13a⟩≡ (12b)
```
struct list *search_solution (struct board const *board)
{
  ⟨set up arguments of the recursive search 13b⟩
  unsigned max_cost = 0;
  while (1)
    {
      struct list *solution =
        search_solution_recursive (boards, moves, costs, max_cost,
                                   distance);
      if (solution)
        {
          ⟨free memory of recursive search arguments 14a⟩
          return solution;
        }
      ⟨reset variables to start search from scratch 14b⟩
      max_cost += 1;
    }
}
```

Set up arguments of the recursive search These variables are needed by the recursive search. They are three lists, corresponding to the moves made so far. The first is the list of the board constellations made by the moves so far, starting with the inital board constellation (the one to be solved). The second is simply the list of moves made so far. The third is a list of the accumulated cost at each given move (`costs[i]` is the sum of the costs of all moves up to `i`).

13b ⟨*set up arguments of the recursive search* 13b⟩≡ (13a)
```
struct list *boards = list_init (sizeof (struct board*));
struct list *moves = list_init (sizeof (enum move));
struct list *costs = list_init (sizeof (unsigned));

unsigned distance = manhattan_distance (board);
unsigned cost = 0;
enum move move = NONE;

list_push_back (boards, &board);
list_push_back (costs, &cost);
list_push_back (moves, &move);
```

Free memory of recursive search arguments When we find our solution, we need to free the memory that was allocated for the lists which are the arguments to our recursive search (except the list of moves, which we return).

14a ⟨*free memory of recursive search arguments* 14a⟩≡ (13a)
```
for (size_t i = 1; i < boards->length; i++)
  {
    struct board *board = *((struct board**) list_get (boards, i));
    free (board);
  }
list_destroy (boards);
list_destroy (costs);
```

Reset variables to start search from scratch Before we increase the allowed cost and start the next iteration of our search function's loop, we reset the arguments of the recursive search so that it starts from scratch.

14b ⟨*reset variables to start search from scratch* 14b⟩≡ (13a)
```
for (size_t i = 1; i < boards->length; i++)
  {
    struct board *board = *((struct board**) list_get (boards, i));
    free (board);
  }
boards->length = 1;
moves->length = 1;
costs->length = 1;
```

4.3.2 Search recursively with maximum cost

We search through all possible moves recursively and exit recursion when the cost has exceeded the allowed cost or when our number of moves is greater than $d + 2n$, where n is the allowed number of moves with cost 1, or when we find our solution. Finally, if we can not find a solution, we return NULL.

14c ⟨*search recursively with maximum cost* 14c⟩≡ (12b)
```
struct list *search_solution_recursive (struct list *boards,
                                        struct list *moves,
                                        struct list *costs,
                                        unsigned max_cost,
                                        unsigned distance)
{
  ⟨variables inside the recursive search 14d⟩
  ⟨conditions to exit recursion 15⟩
  ⟨recursively try all possible moves 16⟩
  return NULL;
}
```

Variables inside the recursive search The variables used inside the function are just the last elements of the lists which are the function arguments.

14d ⟨*variables inside the recursive search* 14d⟩≡ (14c)
```
struct board *board = *((struct board**) list_get_last (boards));
unsigned cost = *((unsigned*) list_get_last (costs));
enum move move = *((enum move*) list_get_last (moves));
```

Conditions to exit recursion We stop searching further along a path when our cost or the number of moves have exceeded their allowed maximum values. In this case we return NULL. We of course also stop when we have found the solution, in which case we return the list of moves that solve the puzzle.

15 ⟨*conditions to exit recursion* 15⟩≡ (14c)

```
if (cost > max_cost
    || boards->length - 1 > distance + 2 * max_cost)
  {
    return NULL;
  }

if (board_is_solved (board))
  {
    return moves;
  }
```

Recursively try all moves We just try one move after another, going into the next level of recursion. Our exit conditions defined above will take care of the rest.

16 ⟨*recursively try all possible moves* 16⟩≡ (14c)

```
struct list *result;if (board_check_move (board, UP) && move != DOWN)
  {
    search_solution_make_move (boards, moves, costs, UP);
    result = search_solution_recursive (boards, moves, costs,
                                        max_cost, distance);
    if (result)
      return result;
    search_solution_undo (boards, moves, costs);
  }
if (board_check_move (board, DOWN) && move != UP)
  {
    search_solution_make_move (boards, moves, costs, DOWN);
    result = search_solution_recursive (boards, moves, costs,
                                        max_cost, distance);
    if (result)
      return result;
    search_solution_undo (boards, moves, costs);
  }
if (board_check_move (board, LEFT) && move != RIGHT)
  {
    search_solution_make_move (boards, moves, costs, LEFT);
    result = search_solution_recursive (boards, moves, costs,
                                        max_cost, distance);
    if (result)
      return result;
    search_solution_undo (boards, moves, costs);
  }
if (board_check_move (board, RIGHT) && move != LEFT)
  {
    search_solution_make_move (boards, moves, costs, RIGHT);
    result = search_solution_recursive (boards, moves, costs,
                                        max_cost, distance);
    if (result)
      return result;
    search_solution_undo (boards, moves, costs);
  }
```

4.3.3 Functions for the recursive search

The first utility function used in the recursive search is simply responsible for making a move and pushing the appropriate values to the back of the lists.

17a ⟨*functions for the recursive search* 17a⟩≡ (12b) 17b ▻

```
void search_solution_make_move (struct list *boards,
                                struct list *moves,
                                struct list *costs,
                                enum move move)
{
  unsigned cost = *((unsigned*) list_get_last (costs));
  struct board *board = *((struct board**) list_get_last (boards));
  cost += board_get_move_cost (board, move);
  struct board *board2 = board_make_move (board, move);
  list_push_back (boards, &board2);
  list_push_back (moves, &move);
  list_push_back (costs, &cost);
}
```

If we have made a move that lead to one of the exit conditions of our recursion, we need to be able to undo it.

17b ⟨*functions for the recursive search* 17a⟩+≡ (12b) ◅ 17a

```
void search_solution_undo (struct list *boards,
                           struct list *moves,
                           struct list *costs)
{
  struct board *board = *((struct board**) list_get_last (boards));
  boards->length -= 1;
  moves->length -= 1;
  costs->length -= 1;
  free (board);
}
```

5 Example main function

We are done! The following main function shows how to use the above code to find the solution to a given board. Of course, one could also define a function to construct the initial board from user input, but this is left as an exercise to the reader.

The example board we use looks as shown in Fig. 2. Its optimal solution is found by this code in a couple of seconds and takes 52 moves.

15	14	1	6
9	11	4	12
	10	7	3
13	8	5	2

Figure 2: The example board used in the main function.

18 ⟨*example main function* 18⟩≡ (2a)

```
int main ()
{
  struct board board = {
    .tiles = {{15,14,1,6},
              {9,11,4,12},
              {0,10,7,3},
              {13,8,5,2}},
    .empty_tile_row = 2,
    .empty_tile_column = 0,
  };
  struct list *moves = search_solution (&board);
  ⟨print the solution 19⟩
  list_destroy (moves);
  return 0;
}
```

Print the solution This just iterates through the list of moves in the solution and prints out corresponding letters.

19 ⟨*print the solution* 19⟩≡ (18)

```
  for (size_t i = 1; i < moves->length; i++)
    {
      enum move move = *((enum move*) list_get (moves, i));
      switch (move)
        {
        case UP:
          printf ("U");
          break;
        case DOWN:
          printf ("D");
          break;
        case LEFT:
          printf ("L");
          break;
        case RIGHT:
          printf ("R");
          break;
        default:
          printf ("X");
        }
    }
  printf ("\n");
```