CS 4, Winter 2023: final exam

Michael Vanier

Mar 11, 2023

CONTENTS:

CHAPTER

ONE

PRELIMINARIES

1.1 Due date

This exam is due on Friday, March 17th, at 6 PM.

1.2 What to hand in

There are three (3) separate CodePost assignments for the final exam: `Final_partA`, `Final_partB`, and `Final_partC`. Submit one file to each of them as follows:

- For `Final_partA`, submit the file `boardrep.ml`.
- For `Final_partB`, submit the file `boardmetric.ml`.
- For `Final_partC`, submit the file `astar.ml`.

1.3 Requirements

There is *no time limit* on this exam, except that it must be handed in on time.

This is an *open-book* exam. You may consult either or both of the textbooks (SICP and Real World OCaml). You may also consult the OCaml manual.

This is an *open reference* exam, with this exception: You may not search for or use specific solutions of the exam problems you find online or in a book, in a previous year's exam written by another student, or anywhere else, however unlikely it may be that you would find such a thing.

This is an *open interpreter/compiler* exam. Specifically, you are allowed to use OCaml to develop and debug your code (and we **very strongly recommend** that you do this). Note that you will need to pass the test scripts to get full credit.

This is a *no-collaboration* exam. Do not discuss the exam with anyone (verbally or in writing) before the due date, even if you both have turned in your exam already. Honor code rules apply. The only exception is to request a clarification of some point in the exam; you are allowed to email the course instructor and/or the course TAs for this purpose only.

1.4 Formatting

Here is how your exam should be formatted. Note that **we will deduct a significant proportion of your final grade (up to 1/4 of the total grade) if you violate any of these rules.** If you have any questions about these rules, contact us before starting the exam and we'll clarify them.

- Make sure that your submitted files are plain text files. Check this by typing the following at a Unix prompt:

```
$ more boardrep.ml
$ more boardmetric.ml
$ more astar.ml
```

  If it prints out your files in a readable manner, it's plain text.

- Write the files as regular OCaml code. Please comment your code appropriately.
- *Make sure* that all of the lines in your final exam file have **no more than 80 characters in a line**. Most text editors will show you what column you're on as you're editing, which will help you keep the line lengths to a maximum of 80 characters.

1.5 Grading

The exam will receive a floating-point score between `0.0` and `40.0`, based on your performance across all of the problems. **There is no rework.**

1.6 Clarifications

If you need a clarification of anything on the exam after you've started writing it, you may email Mike or one of the TAs or make a private post on the course Piazza page.

CHAPTER

TWO

PREAMBLE

The exam as a whole is worth 40.0 marks. It consists of three sections, which are worth the following:

- part A: 20 marks
- part B: 10 marks
- part C: 10 marks

Specific point values for individual problems are listed with the problem descriptions below.

The exam is one big miniproject. The goal of the project is to write a program which can solve a generalized version of the 15 puzzle. The generalization is that the board size doesn't have to be only 4x4; it can be any NxN board. We thus refer to the puzzle more generically as the "N-puzzle". We will only be working seriously with boards of size 3x3 and 4x4; larger boards can also be solved by the program but the time required increases greatly. (Solving the N-puzzle optimally is NP-hard.)

You will do this by first completing the implementation of two low-level modules for managing the puzzle state and assessing the "goodness" (nearness to the solved position) of a puzzle state. Then you will implement a higher-level module which intelligently searches through the state space to find an optimal solution from any starting position. If you do this right, most puzzle positions (for 3x3 and 4x4 puzzles at least) will be solvable in a reasonable amount of time, though positions that are far from the solved position may take a long time to solve. Some positions are unsolvable, and your search function will detect that too.

The description of the puzzles and the algorithms is quite long, but **please read it through carefully before beginning!** Failure to do so will *drastically* increase the time required to finish the exam successfully. Also, **failure to do things exactly as we describe below will result in lost marks!** There is no time limit, so take your time. Our solution (not including the supplied code, of which there is a lot) is around 250 lines long, so if yours is much longer, you may be doing something wrong (or at least something that could be done with much less code). We hope you enjoy this exam!

2.1 Code base

We are providing you with a number of files of code which you should use as-is (*i.e.* don't edit these files), as well as template files for the submitted final exam files (`boardrep.ml`, `boardmetric.ml` and `astar.ml`) which have some of the code pre-written for you and which you will need to edit. All these files are collected in a single zip file called `final.zip` which can be downloaded from the course Canvas site.

Each of the files you need to submit will have places where the following code: `failwith "TODO"` is found. You should replace this code with your own code.

Note: We use `failwith "TODO"` so that the code will compile as-is, but it won't do anything useful. Any attempt to use it will result in the `Failure` exception being raised with the error message `"TODO"`.

Also, if you compile the code without filling in the "TODO" parts, you will see a number of warnings from OCaml about unused variables and other things. Once you complete the code, these warnings should go away.

Note that you are allowed to add more functions (*e.g.* helper functions) in addition to the ones already in the file, and we encourage you to do so if you find it helpful.

The files you need to edit and submit are:

- `boardrep.ml` (for part A)
- `boardmetric.ml` (for part B)
- `astar.ml` (for part C)

The supporting files are:

- `Makefile`
- `boardrep.mli`
- `boardmetric.mli`
- `board.ml`
- `board.mli`
- `npuzzle.ml`
- `npuzzle.mli`
- `pqueue.ml`
- `pqueue.mli`
- `astar.mli`
- `main.ml`
- `tests_final_partA.ml`
- `tests_final_partB.ml`
- `tests_final_partC.ml`

Please do not edit any of the supporting files! You will not be submitting them, so we wouldn't see your edits anyway.

2.2 The `Makefile`

One of the supplied files is a `Makefile` which you should use to compile and test your code. There are several `Makefile` targets:

- Typing `make` by itself compile a program called `npuzzle` that we will discuss below. This program will find N-puzzle solutions for boards of a user-specified size, and a number of other options are available as well. (For instance, you can also solve the puzzle interactively instead of getting the computer to solve it.}
- Typing `make test` will compile and run all the test scripts (`tests_partA.ml`, `tests_partB.ml` and `tests_partC.ml`) and will report all test successes and failures.

  For more specific tests, type:

  - `make test_partA` to run only the part A tests;
  - `make test_partB` to run only the part B tests;
  - `make test_partC` to run only the part C tests.

- Typing `make all` is equivalent to typing `make` followed by `make test`.
- Typing `make clean` will remove all compilation targets (object code, executables, generated test files *etc.*) but leave all source code alone.

All programs will be compiled using the OCaml native-code compiler by default (for speed), not the byte-code compiler. However, in the event that you have a problem with the native-code compiler, you can switch to the byte-code compiler by changing one line in the `Makefile`; just change

```
COMP = native
```

to:

```
COMP = byte
```

and recompile, and the new version will use the byte-code compiler. It'll also run very slowly, so we encourage you to use the native-code compiler if at all possible. Alternatively, you can force the `Makefile` to use the byte-code compiler by invoking it like this:

```
$ make COMP=byte
```

If you do this, you don't need to edit the `Makefile`.

CHAPTER

THREE

OCAML NOTES

3.1 Libraries to install

We will be using a priority queue library which is not part of the OCaml standard library. To get this, you need to install the `batteries` package using `opam` as follows:

```
$ opam install batteries
```

3.2 Testing

Please run the test scripts while you are writing the code! We are supplying you with a test script for each section, so just run it (by typing `make test_partA` for part A, `make test_partB` for part B, or `make test_partC` for part C) while you are working on the code. We've also made it easy to comment out the tests for code you haven't written yet. (See the bottom of the test scripts; you only have to comment out a few lines.) **Don't write hundreds of lines of code before starting the testing process!** If you do this, you may find that you have made a fundamental mistake early on, and you'll have to rewrite most of your code. Instead, write a little code, then test a little code, and repeat until you're done.

For the first part of the exam, compiling the `npuzzle` program and running it in interactive mode (see the next section) is also an excellent way to see if your code works.

3.3 Design guidelines

The main guideline we want you to follow is that all of the code you will write should be *purely functional* unless *specifically stated otherwise*. You will not need to use references (`ref` values) in any of your code, **and you should not!** Some of the code deliberately uses arrays, and so that code is necessarily imperative, but even there, the imperative aspects are internal to the modules. (We'll have more to say about this later.) When in doubt, do things functionally!

One consequence of this is the following: **do not use `for` loops, `while` loops, or `ref` values anywhere in the code**. (Not even for the array code; you can always write a recursive function to simulate a loop.)

3.4 Functional records

Much of the code will use OCaml records for representing data types. Although OCaml supports records with mutable (imperative) fields, we will not use them here. Instead, many functions will receive as input a record of a particular type (say, type `t`), and will output the same record type, but where some fields will have different values. These are called "functional records" and we've seen them in the assignments. Note that "field punning" (see assignment 4) can make using functional records considerably more pleasant.

Be aware of the following shortcut: if you have a record of type `t`, for instance:

```
type t = {
  foo : int;
  bar : float;
  baz : string
}
```

and you want to change only some of its fields, you can use this syntax:

```
(* Return a copy of record r, with the baz field changed. *)
# let say_hello r = { r with baz = "hello!" } ;;
val say_hello : t -> t = <fun>

# say_hello { foo = 10; bar = 1.2; baz = "baz" } ;;
- : t = {foo = 10; bar = 1.2; baz = "hello!"}
```

Typically, the way you will use this is to compute all the new values of fields that have changed and then create and return a new record containing these values plus whatever values didn't change, using the syntax above.

If you need to change multiple fields, the syntax is similar:

```
(* Return a copy of record r, with the bar and baz fields changed. *)
# let change_it r = { r with bar = 1.23; baz = "hello!" } ;;
val change_it : t -> t = <fun>

# change_it { foo = 10; bar = 1.2; baz = "baz" } ;;
- : t = {foo = 10; bar = 1.23; baz = "hello!"}
```

3.5 Arrays

Be aware that if you try to access an element of an array beyond its bounds (with an index less than zero or >= the size of the array), OCaml will raise an `Invalid_argument` exception. You can catch this exception if you want to do something special when this happens. This will in fact be the case in some of the code you will write.

3.6 Useful libraries and library functions

Here are some functions we found useful in our solution. You don't have to use them, and you can use other functions from the libraries indicated.

- The Map.S module from the OCaml standard libraries. This gives all the functions you can call on OCaml `Map` modules (those created with the `Map.Make` functor).
- The `add` and `find_opt` functions from `Map.S`.
- The List module from the OCaml standard libraries.
- The `length`, `mem` and `fold_left` functions from `List`.
- The Array module from the OCaml standard libraries.
- The `make`, `copy`, and `of_list` functions from `Array`.

CHAPTER

FOUR

THE 15-PUZZLE: BACKGROUND

4.1 Puzzle description

The "15-puzzle" is a very simple sliding block puzzle that you have almost certainly played before. The "board" is a 4x4 grid with square pieces on all but one of the grid spaces. Each piece is labelled with one of the integers between 1 and 15 (inclusive). The "solved" configuration looks like this:

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

A move consists of sliding a single piece into the "hole", which is the grid location where there is no piece. In the diagram, this is the location at the lower right corner of the grid *i.e.* at row 3, column 3 (using 0 indexing). You can only move a piece which is horizontally or vertically adjacent to the hole into the hole. In the diagram, only the pieces with the labels 12 and 15 can be moved. Moving a piece also moves the hole, so if the 12 was moved into the hole the configuration would become:

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

Now the hole has moved to row 2, column 3 since the 12 piece had to vacate that location to move into where the hole used to be.

Moves can't wrap around the grid, so the piece with the label 9 can't be moved into the hole from the above position, for instance. No matter where the hole is, there will always be between 2 and 4 legal moves to make.

To start the puzzle, starting from the solved configuration a large number of random moves are made to scramble the configuration of the puzzle.[1] Moves are made until the board is back to the solved configuration, which is the criteria for "winning" or "solving" the puzzle.

Here is an online version of the game, which you should probably play once or twice to get familiar with the game if you haven't played it before. Solving the 15 puzzle is not rocket science, but it's also not trivial. Solving it optimally (*i.e.* with the minimum number of moves) is very difficult for humans.

Here is a typical scrambled configuration:

```
 3  8  4 12
10  1 11 15
```

(continues on next page)

[1] You can't just place the pieces into random locations on the grid, since half of the configurations you would get that way will be unsolvable.

(continued from previous page)

```
 5  9     2
 6 13  7 14
```

To make it clearer where the hole is, in our diagrams we will use the number 0 to represent the hole. In that case, the previous diagram would look like this:

```
 3  8  4 12
10  1 11 15
 5  9  0  2
 6 13  7 14
```

As you can see, the hole location is in row 2, column 2 (again, using zero-indexing for the row, column indices).

We can also have puzzles that have grid sizes other than 4x4. For this exam, we will only use NxN puzzles where N is either 3 or 4 (a few of the tests also use 5x5 boards). Here is a scrambled configuration on a 3x3 grid:

```
2 7 3
5 0 6
8 4 1
```

We will use 3x3 grids extensively for testing, since the 3x3 puzzle is much easier to solve. (We can call this version of the puzzle the "8-puzzle"; similarly, a 5x5 grid would be the "24-puzzle" and an NxN grid would be the "(N^2-1)-puzzle".

The solver you will write will not only be able to solve boards starting from a randomized configuration, but it will also produce optimal solutions. It will be able to solve NxN boards of any size, but boards 5x5 and longer will take a long time to solve.

Note: The problem of computing optimal solutions to the N-puzzle problem is NP-hard.

4.2 Outline of the solution algorithm

The algorithm we will use to solve the 15 puzzle is the A* algorithm, also known as "best-first search". In some ways it resembles the depth-first search and breadth-first search algorithms you implemented in assignment 7. Like those algorithms, every unsolved board configuration examined generates a collection of new board configurations which get added to a data structure to be examined later (unless they have already been examined, in which case they may or may not be discarded; see below). However, instead of pushing the new configurations onto a stack (like depth-first search) or a queue (like breadth-first search) the new configurations get added to a *priority queue*. (We've already implemented a priority queue in assignment 6, but don't worry if you had problems with that; here, we supply the priority queue implementation for you.) The important point is that we will compute the "goodness" of every board configuration and we will use the priority queue to ensure that we look at the "best" configurations first, before looking at any of the less good ones. The goodness of a board configuration depends both on how different it is from the solved configuration and how many moves it took from the starting position to get to that configuration. Ideally we want a configuration that matches the solved configuration exactly and took the fewest number of moves to get to that point.

There are different ways to compute "goodness" of a configuration; we will refer to these as *metrics* because they compute the "distance" from the configuration to the solved configuration. We will use two different metrics, which we will describe below. Both will work, though one will result in a much more efficient search.

4.3 Running the `npuzzle` program

The program you will write is called `npuzzle`. Running it with the `-help` argument gives this usage message:

```
$ ./npuzzle -help
usage: npuzzle [-s size] [-l num] [-r nrandom] [-a] [-h] [-i] [-help]
    -s size: side length of board (default 3)
      (e.g. 3 for 8-puzzle, 4 for 15-puzzle, etc.)
    -l num: load starting board at index `num` (>= 0)
    -r num: randomize the board by making `num` (>= 0) random moves
      starting from the solved state
    -a: use the array board representation instead of the map representation
    -h: use the Hamming metric instead of the Manhattan metric
    -i: don't solve the board; allow user to solve it interactively
    -help: print this help message
  default is equivalent to: -s 3 -l 0
```

There is a lot going on here. We'll describe the command-line options briefly here and in more detail below.

- The `-s` option (followed by a positive integer >= 3) specifies the "size" of the board (the N in NxN).
- The `-l` option (followed by a non-negative integer) loads a stored initial board configuration (like the predefined boards in assignment 7).
- The `-r` option (followed by a non-negative integer) makes a number of random moves starting from the solved configuration to generate the initial configuration.
- The `-a` option specifies that the board should be represented as an array instead of as a map (which is the default).
- The `-h` option specifies that the solver should use the Hamming metric instead of the Manhattan metric (which is the default).
- The `-i` option indicates that the solver should be turned off and the user should solve the puzzle interactively.
- The `-help` option prints the help message.

If no command-line arguments are specified, it's equivalent to `-s 3 -l 0`, which means to use a 3x3 board and load the 0th board configuration (which happens to be the solved configuration).

4.3.1 Sample runs

Here's a solution of a 3x3 configuration:

```
$ ./npuzzle -s 3 -r 100
Configuration: [4; 1; 3; 2; 0; 7; 6; 8; 5]
Solution found in 18 moves.
```

The solution (if found) is put into a file called `npuzzle.out`. In that file, the boards are displayed starting from the original board and ending with the solved board. The number of moves is also printed to the file.

This particular starting configuration was arrived at by making 100 random moves starting from the solved configuration. The program prints out the configuration as a list, but it's really this configuration:

```
4 1 3
2 0 7
6 8 5
```

Here is the solution:

```
4  1  3
2  .  7
6  8  5

4  1  3
2  8  7
6  .  5

4  1  3
2  8  7
.  6  5

4  1  3
.  8  7
2  6  5

4  1  3
8  .  7
2  6  5

4  1  3
8  7  .
2  6  5

4  1  3
8  7  5
2  6  .

4  1  3
8  7  5
2  .  6

4  1  3
8  7  5
.  2  6

4  1  3
.  7  5
8  2  6

4  1  3
7  .  5
8  2  6

4  1  3
7  2  5
8  .  6

4  1  3
7  2  5
.  8  6

4  1  3
```

(continues on next page)

(continued from previous page)

```
 .  2  5
 7  8  6

 .  1  3
 4  2  5
 7  8  6

 1  .  3
 4  2  5
 7  8  6

 1  2  3
 4  .  5
 7  8  6

 1  2  3
 4  5  .
 7  8  6

 1  2  3
 4  5  6
 7  8  .

18
```

And here is an unsolveable configuration:

```
$ ./npuzzle -s 3 -l 8
Configuration: [1; 2; 3; 4; 5; 6; 8; 7; 0]
No solution found.
```

This is stored configuration #8 for 3x3 boards. (The stored configurations are all in the `main.ml` file.) As you can see, it's the same as the solved configuration except two adjacent pieces have been swapped. That's enough to make it unsolvable. The program helpfully tells us that.

Of course, we could do the same runs with 4x4 boards (that would require the `-s 4` argument).

4.3.2 [OPTIONAL] Interactive solution

If we use the `-i` argument we can solve a puzzle interactively. When doing this, we are presented with a board configuration and asked to input a move. Acceptable inputs include:

- `q` to quit the program
- `u` to move a piece [U]p into the hole
- `d` to move a piece [D]own into the hole
- `l` to move a piece [L]eft into the hole
- `r` to move a piece [R]ight into the hole

These are the only allowed inputs. Note that an "up" move (say) means something different to the computer solver than it does to the interactive solver. The computer solver interprets an "up" move as moving the hole one position upwards in the grid (so the row index decreases by 1). The interactive solver interprets an "up" move as moving an adjacent piece up into where the hole was. This turns out to be more intuitive when solving a puzzle interactively.

Here is a sample interactive solve of a 3x3 board:

```
$ ./npuzzle -s 3 -r 100 -i

  4  1  2
  6  .  3
  8  7  5

Enter move (u,d,l,r,q) > r

  4  1  2
  .  6  3
  8  7  5

Enter move (u,d,l,r,q) > d

  .  1  2
  4  6  3
  8  7  5

Enter move (u,d,l,r,q) > l

  1  .  2
  4  6  3
  8  7  5

Enter move (u,d,l,r,q) > l

  1  2  .
  4  6  3
  8  7  5

Enter move (u,d,l,r,q) > u

  1  2  3
  4  6  .
  8  7  5

Enter move (u,d,l,r,q) > u

  1  2  3
  4  6  5
  8  7  .

Enter move (u,d,l,r,q) > r

  1  2  3
  4  6  5
  8  .  7

Enter move (u,d,l,r,q) > d

  1  2  3
```

(continues on next page)

(continued from previous page)

```
  4  .  5
  8  6  7

Enter move (u,d,l,r,q) > r

  1  2  3
  .  4  5
  8  6  7

Enter move (u,d,l,r,q) > u

  1  2  3
  8  4  5
  .  6  7

Enter move (u,d,l,r,q) > l

  1  2  3
  8  4  5
  6  .  7

Enter move (u,d,l,r,q) > d

  1  2  3
  8  .  5
  6  4  7

Enter move (u,d,l,r,q) > r

  1  2  3
  .  8  5
  6  4  7

Enter move (u,d,l,r,q) > u

  1  2  3
  6  8  5
  .  4  7

Enter move (u,d,l,r,q) > l

  1  2  3
  6  8  5
  4  .  7

Enter move (u,d,l,r,q) > l

  1  2  3
  6  8  5
  4  7  .

Enter move (u,d,l,r,q) > d
```

(continues on next page)

(continued from previous page)

```
  1  2  3
  6  8  .
  4  7  5

Enter move (u,d,l,r,q) > r

  1  2  3
  6  .  8
  4  7  5

Enter move (u,d,l,r,q) > r

  1  2  3
  .  6  8
  4  7  5

Enter move (u,d,l,r,q) > u

  1  2  3
  4  6  8
  .  7  5

Enter move (u,d,l,r,q) > l

  1  2  3
  4  6  8
  7  .  5

Enter move (u,d,l,r,q) > l

  1  2  3
  4  6  8
  7  5  .

Enter move (u,d,l,r,q) > d

  1  2  3
  4  6  .
  7  5  8

Enter move (u,d,l,r,q) > r

  1  2  3
  4  .  6
  7  5  8

Enter move (u,d,l,r,q) > u

  1  2  3
  4  5  6
  7  .  8
```

(continues on next page)

(continued from previous page)

```
Enter move (u,d,l,r,q) > l

  1  2  3
  4  5  6
  7  8  .

SOLVED!
```

You don't have to use the interactive solver at all. However, it's quite useful for debugging part A.

CHAPTER

FIVE

PART A: THE BOARDREP MODULE

This section is worth 20 marks total (10 marks for each representation). The code for this section will be written in the file `boardrep.ml`.

The code in this file defines the low-level board representation, as well as functions for manipulating the representation. The key abstraction is the `BoardRep` module type, which has this definition:

```
type loc = int * int
type move = Up | Down | Left | Right

module type BoardRep =
  sig
    type t

    exception Invalid_move
    exception Invalid_location

    val init      : int -> t
    val load      : int -> int list -> t
    val get_size  : t -> int
    val get_hole  : t -> loc
    val get       : t -> loc -> int
    val make_move : t -> move -> t
    val show      : t -> unit
  end
```

Here is a brief description of the contents of this module type.

- The type `t` is the board representation type.
- The `init` function takes a positive integer $n >= 2$ and creates an $n \times n$ board representation in the solved configuration.
- The `load` function takes a positive integer $n >= 2$ and a list of integers (all of which should be in the range $[0, n^2 - 1]$), and creates a board representation in the configuration specified by the list. The list represents the contents of the board rows concatenated with each other; if it was the solved configuration for a 3x3 board, for instance, it would be `[1; 2; 3; 4; 5; 6; 7; 8; 0]`.
- The `get_size` function takes a board representation and returns the "size" of the board (the n in $n \times n$).
- The `get_hole` function takes a board representation and returns the location of the hole as a (row, column) pair.
- The `get` function takes a board representation and a (row, column) location and returns the contents of that location as an integer. The hole is represented by the integer 0. If the (row, column) location is off the board, the `Invalid_location` exception is raised.

- The `make_move` function takes a board representation and a move and attempts to move the hole in the direction specified by the move. If it succeeds, it returns the updated board representation. If it fails, it raises the `Invalid_move` exception.
- The `show` function prints out the board representation in a human-readable form. (This function is primarily for debugging.)

You will be implementing two different modules that instantiate this module type: `ArrayRep` and `MapRep`. The type `t` for each representation has been provided to you, as well as the implementations of `get_size`, `get_hole`, `get`, and `show`. Your job is to implement the functions `init`, `load`, and `make_move` for each representation.

5.1 The `ArrayRep` board representation

The `ArrayRep` board representation uses a one-dimensional array of integers as the underlying data representation.[1] Specifically, the representation is:

```
type t =
  {
    acontents : int array;
    size : int;
    hole : loc
  }
```

We keep track of the size, the location of the hole, and the contents of the board as a one-dimensional array of integers. In order to use a one-dimensional array as if it were a two-dimensional grid, we have to convert between the (row, column) indices of a grid location to an index into the array. This is easy to do:

$$array\ index = row \times size + column$$

So if the board is a 4x4 board and you want to access location (row 2, column 3), this would have index $2 \times 4 + 3 = 11$.

One peculiarity of this representation is that when updating the board (which happens after calling `make_move`) the original board should not be altered. This means that you have to create a **copy** of the original board (using the `Array.copy` function) and then mutate it instead of mutating the original board. If you don't do this, your code won't work and the tests won't pass!

Note: You are allowed to use imperative coding idioms (array access, array update, *etc.*) in all the `ArrayRep` code in part A (*i.e.* in the `boardrep.ml` file, `ArrayRep` parts only). In some cases it isn't necessary, but in others it is.

5.2 The `MapRep` representation

The `MapRep` board representation uses a map between locations and integers (which is called a `LocMap` in the code) as the underlying data representation. Specifically, the representation is:

```
type t =
  {
    mcontents : int LocMap.t;
    size : int;
```

(continues on next page)

[1] You might wonder why we didn't choose to use a two-dimensional array of integers *i.e.* an array of arrays. The main reason is to avoid problems with aliasing that can easily come up when dealing with arrays of arrays.

(continued from previous page)

```
    hole : loc
  }
```

This is a standard purely functional representation of two-dimensional data. As with `ArrayRep`, we keep track of the board size and the location of the hole.

5.3 Functions to write

For each of the two representations, you need to implement these functions. Each representation is worth 10 marks, so the entire section is worth 20 marks.

5.3.1 `init`

[3 marks]

This function takes in a positive integer >= 2 and returns a board representation in the solved configuration. If the argument is < 2 then a `Failure` exception is raised (most conveniently by using `failwith`). Make sure you set the `size` and `hole` fields in the board record before returning it.

Examples

```
# let b_bad = ArrayRep.init 0 ;;
Exception: Failure "ERROR: init: size must be at least 2".

# let b1 = ArrayRep.init 3 ;;
val b1 : ArrayRep.t = <abstr>
# ArrayRep.get_size b1 ;;
- : int = 3
# ArrayRep.get_hole b1 ;;
- : loc = (2, 2)
# ArrayRep.get b1 (0, 0) ;;
- : int = 1
# ArrayRep.get b1 (2, 2) ;;
- : int = 0
# ArrayRep.show b1 ;;

  1  2  3
  4  5  6
  7  8  0

- : unit = ()

# let b_bad = MapRep.init 0 ;;
Exception: Failure "ERROR: init: size must be at least 2".

# let b2 = MapRep.init 4 ;;
val b2 : MapRep.t = <abstr>
# MapRep.get_size b2 ;;
- : int = 4
# MapRep.get_hole b2 ;;
```

(continues on next page)

(continued from previous page)

```
- : loc = (3, 3)
# MapRep.get b2 (0, 0) ;;
- : int = 1
# MapRep.get b2 (3, 3) ;;
- : int = 0
# MapRep.show b2 ;;

  1  2  3  4
  5  6  7  8
  9 10 11 12
 13 14 15  0

- : unit = ()
```

5.3.2 `load`

[3 marks]

This function takes in a positive integer >= 2 (representing the board size as in the `init` function) as well as a list of integers (representing the board contents) and returns a board representation where the contents of the board are taken from the list. The elements in the list represent the labels of the pieces in the board rows, all concatenated together. So, for example, an input list of `[5;3;1;2;0;6;4;8;7]` corresponds to this 3x3 board:

```
5 3 1
2 0 6
4 8 7
```

As usual, 0 represents the hole. Again, make sure you set the `size` and `hole` fields in the board record before returning it.

This function should fail (by raising a `Failure` exception) in the following situations:

- The `size` argument is < 2.
- The length of the input list isn't `size * size`.
- The contents of the input list doesn't contain all integers between 0 and `size * size - 1`, inclusive.

Examples

```
(* Check error handling. *)

# let b1 = ArrayRep.load 3 [] ;;   (* too few elements *)
Exception: Failure "invalid list length".

# let b1 = ArrayRep.load 3 [1;2;3;4;5;6;7;8;9;0] ;;  (* too many elements *)
Exception: Failure "invalid list length".

# let b1 = ArrayRep.load 3 [1;2;3;4;5;6;7;8;9] ;;    (* no hole *)
Exception: Failure "invalid list contents".

# let b1 = ArrayRep.load 3 [1;2;3;4;5;6;7;1;0] ;;    (* repeated 1s *)
```

(continues on next page)

(continued from previous page)

```
Exception: Failure "invalid list contents".

# let b1 = ArrayRep.load 3 [1;3;5;7;9;11;13;15;0] ;; (* missing elements *)
Exception: Failure "invalid list contents".

(* With correct inputs. *)

# let b1 = ArrayRep.load 3 [1;2;3;4;5;6;7;8;0];;
val b1 : ArrayRep.t = <abstr>

# ArrayRep.show b1 ;;

  1  2  3
  4  5  6
  7  8  0

- : unit = ()

# ArrayRep.get_size b1 ;;
- : int = 3

# ArrayRep.get_hole b1 ;;
- : loc = (2, 2)

# let b2 = ArrayRep.load 3 [5;1;3;2;8;7;4;0;6] ;;
val b2 : ArrayRep.t = <abstr>

# ArrayRep.get_size b2 ;;
- : int = 3

# ArrayRep.get_hole b2 ;;
- : loc = (2, 1)

# ArrayRep.show b2 ;;

  5  1  3
  2  8  7
  4  0  6

- : unit = ()

(* Using MapRep gives the same results. *)
```

5.3.3 make_move

[4 marks]

This function takes in a board and a direction (one of `Up`, `Down`, `Left` or `Right`) and attempts to move the hole one space in that direction on the board. The previous contents in the hole's new location are put into the location vacated by the hole. Note that `Up` means in the direction of lower row indices while `Down` means in the direction of higher row indices; similarly, `Left` means in the direction of lower column indices while `Right` means in the direction of higher column indices. If the hole would move off the board, the function raises an `Invalid_move` exception.

Making a move returns a new board; the board passed as an argument to the function is unchanged, even if the board uses the array representation.

Examples

```
# let b1 = ArrayRep.load 3 [6; 1; 8; 3; 7; 2; 0; 4; 5] ;;
val b1 : ArrayRep.t = <abstr>

# ArrayRep.show b1 ;;

  6  1  8
  3  7  2
  0  4  5

- : unit = ()

# let b2 = ArrayRep.make_move b1 Up ;;  (* move the hole up *)
val b2 : ArrayRep.t = <abstr>

# ArrayRep.show b2 ;;

  6  1  8
  0  7  2
  3  4  5

- : unit = ()

(* Check that making a move doesn't alter the original board. *)

# ArrayRep.show b1 ;;

  6  1  8
  3  7  2
  0  4  5

- : unit = ()

# let b3 = ArrayRep.make_move b1 Right ;;  (* move the hole to the right *)
val b3 : ArrayRep.t = <abstr>

# ArrayRep.show b3 ;;

  6  1  8
  3  7  2
```

(continues on next page)

(continued from previous page)

```
  4  0  5

- : unit = ()

(* Attempting to move the hole off-board raises an Invalid_move exception. *)

# let b3 = ArrayRep.make_move b1 Down ;;
Exception: ArrayRep.Invalid_move.

# let b3 = ArrayRep.make_move b1 Left ;;
Exception: ArrayRep.Invalid_move.

(* Using MapRep gives the same results. *)
```

Note: You should catch any `Invalid_location` exceptions and re-raise them as `Invalid_move` exceptions. This function should never raise an `Invalid_location` exception.

CHAPTER

SIX

PART B: THE BOARDMETRIC MODULE

This section is worth 10 marks total. The code for this section will be written in the file `boardmetric.ml`.

The code in this file defines the modules and functions which compute the similarity of a board representation relative to another one. We refer to this as a "metric" because it computes a "distance" between two boards. The key abstraction is the `BoardMetric` module type, which has this definition:

```
module type BoardMetric =
  sig
    type t

    val distance : t -> t -> int
  end
```

This module type has only one function: `distance`. That function takes in two values of type `t` (which will be board representations) and returns a non-negative integer measure of how close the two boards' states are from each other. By convention, the first board will be the solved state. If the function returns 0, the two boards are identical. If the two boards have different sizes, the function signals an error by raising a `Failure` exception.

Note: You are **not** allowed to assume that the first board is a solved board. If you ignore the first argument of the `distance` function and just assume that it's the solved board, you will lose marks. Don't do this.

We use this module type to define two functors, with these partial definitions:

```
module Hamming(B : BoardRep) : BoardMetric with type t = B.t =
  struct
    type t = B.t

    let distance b1 b2 =
      failwith "TODO"
  end

module Manhattan(B : BoardRep) : BoardMetric with type t = B.t =
  struct
    type t = B.t

    let distance b1 b2 =
      failwith "TODO"
  end
```

Each functor takes a module with the `BoardRep` module type and creates a `BoardMetric` module. The two functors define different distance metrics which will be used in the board solver described in the next section. Both metrics are

what is referred to as "admissible", which we will discuss more below; it basically means that they can be used by the solver to find optimal solutions to the N-puzzle.

Note that whether we use an `ArrayRep` or a `MapRep` board representation makes no difference to the `BoardMetric` code; the same code works for all representations because the code only uses the functions in the `BoardRep` module type to access the board representation *i.e.* the interface functions.

6.1 The Hamming metric

[5 marks]

The Hamming metric is one of the simplest metrics imaginable. Basically, you iterate through all the board locations and add 1 for each location that is different. One slight alteration to this rule is that you don't count the hole in the second board (which is the board you are actually comparing to the first board, which by convention is a solved board); you only count the locations that contain pieces *i.e.* nonzero values.

In order to test this module interactively, you should compile the code using the bytecode compiler instead of the native-code compiler. One way to do this is as follows:

```
$ make clean
$ make COMP=byte
```

In part C, you will want to recompile using the native-code compiler (the default) to make the code run as fast as possible.

Examples

```
# #load "boardrep.cmo" ;;
# open Boardrep ;;
# #use "boardmetric.ml" ;;
module type BoardMetric = sig type t val distance : t -> t -> int end
module Hamming :
  functor (B : Boardrep.BoardRep) ->
    sig type t = B.t val distance : t -> t -> int end
module Manhattan :
  functor (B : Boardrep.BoardRep) ->
    sig type t = B.t val distance : t -> t -> int end

(* Define specialized Hamming modules for each board representation.
   Note that the same definition of distance can be used for both board
   representations.  The module HA is used for ArrayRep boards
   and HM is used for MapRep boards. *)

# module HA = Hamming(ArrayRep) ;;
module HA : sig type t = ArrayRep.t val distance : t -> t -> int end
# module HM = Hamming(MapRep) ;;
module HM : sig type t = MapRep.t val distance : t -> t -> int end

(* Now we can define boards and compute their distances from each other. *)

# let b1 = ArrayRep.init 3 ;;
val b1 : HA.t = <abstr>
# ArrayRep.show b1 ;;
```

(continues on next page)

(continued from previous page)

```
  1  2  3
  4  5  6
  7  8  0

- : unit = ()
# let b1a = ArrayRep.init 4 ;;
val b1a : HA.t = <abstr>

(* You can't compute distances between boards of different sizes. *)
# HA.distance b1 b1a ;;
Exception: Failure "incompatible board sizes".

# let b2 = ArrayRep.load 3 [1; 2; 3; 4; 5; 6; 7; 0; 8] ;;
val b2 : HA.t = <abstr>
# ArrayRep.show b2 ;;

  1  2  3
  4  5  6
  7  0  8

- : unit = ()

(* The distance from a board to itself is 0. *)
# HA.distance b1 b1 ;;
- : int = 0

# HA.distance b1 b2 ;;
- : int = 1

(* This distance is small because the board b2 is one move away from being solved. *)

# let b3 = MapRep.init 4 ;;
val b3 : HM.t = <abstr>
# MapRep.show b3 ;;

  1  2  3  4
  5  6  7  8
  9 10 11 12
 13 14 15  0

- : unit = ()

# let b4 = MapRep.load 4 [9; 5; 6; 4; 13; 3; 1; 11; 0; 10; 12; 7; 14; 2; 15; 8] ;;
val b4 : HM.t = <abstr>
# MapRep.show b4 ;;

  9  5  6  4
 13  3  1 11
  0 10 12  7
 14  2 15  8
```

(continues on next page)

(continued from previous page)

```
- : unit = ()

# HM.distance b3 b4 ;;
- : int = 12

(* This distance is large because the board b4 is nowhere near solved. *)
```

As you can see, the distance is just the sum of all the locations where the second board's contents are different from the first board's contents, skipping the 0 in the second board.

6.2 The Manhattan metric

[5 marks]

Although the Hamming metric is easy to compute, it's not very "accurate" in the sense that all it tells you is which locations have the wrong values. It would be nice if, for a given wrong value, the metric took into account how far that value was from the place it was supposed to be. For instance, both of these boards have the same Hamming distance (1) from the solved board:

```
(* Board 1: *)

1  2  3
4  5  6
7  0  8

(* Board 2: *)

1  0  3
4  5  6
7  8  2
```

However, it's clear that board 1 is much closer to being solved than board 2 is. It would be nice if the board metric would take that into account. One metric which does is the so-called Manhattan metric (also called the "taxicab metric"). For this metric, each piece location in the second board is compared with the location of the same piece in the first board and the (row, column) distances are added up. For a given piece, this (row, column) distance is the sum of the row distance and the column distance. The row distance is the absolute value of the difference in row indices, and similarly for the column distance. We add the piece distances for all pieces (skipping the 0 piece in the second board as we did for the Hamming metric) to get the total distance between the two boards.

In the example above, board 1 has only one "piece" (nonzero number) which is not in the solved position: the 8. The 8 is at row 2, column 2 but it should be at row 2, column 1, so its distance from its true position is `(2 - 2) + (2 - 1) = 1`. Board 2 also has only one piece which is not in the solved position: the 2. The 2 is at row 2, column 2 but it should be at row 0, column 1. Its distance is therefore `(2 - 0) + (2 - 1) = 3`. So board 2 would be considered much further from solved than board 1, which is in fact the case.

Hint: This can be done very inefficiently if you're not careful, which will make your overall solver really slow! An efficient way to compute the Manhattan metric would be to record the location of each number (except 0) in a board in a data structure, and then compare two such data structures (solved vs not solved) element-by-element. The data structure could be a list of locations or an array of locations. The element-by-element comparison would simply be computing the (Manhattan) distance between two locations. Then add up all the distances to get the overall distance

between two boards.

Examples

```
# #load "boardrep.cmo" ;;
# open Boardrep ;;
# #use "boardmetric.ml" ;;
module type BoardMetric = sig type t val distance : t -> t -> int end
module Hamming :
  functor (B : Boardrep.BoardRep) ->
    sig type t = B.t val distance : t -> t -> int end
module Manhattan :
  functor (B : Boardrep.BoardRep) ->
    sig type t = B.t val distance : t -> t -> int end

# module MA = Manhattan(ArrayRep) ;;
module MA : sig type t = ArrayRep.t val distance : t -> t -> int end
# module MM = Manhattan(MapRep) ;;
module MM : sig type t = MapRep.t val distance : t -> t -> int end

# let b1 = ArrayRep.init 3 ;;
val b1 : MA.t = <abstr>
# let b1a = ArrayRep.init 4 ;;
val b1a : MA.t = <abstr>

# MA.distance b1 b1a ;;
Exception: Failure "incompatible board sizes".

# let b2 = ArrayRep.load 3 [1; 2; 3; 4; 5; 6; 7; 0; 8] ;;
val b2 : MA.t = <abstr>
# ArrayRep.show b2 ;;

  1  2  3
  4  5  6
  7  0  8

- : unit = ()

# MA.distance b1 b1 ;;
- : int = 0

# MA.distance b1 b2 ;;
- : int = 1

(* This distance is small because the board b2 is one move away from being solved. *)

# let b3 = MapRep.init 4 ;;
val b3 : MM.t = <abstr>
# MapRep.show b3 ;;

  1  2  3  4
  5  6  7  8
```

(continues on next page)

(continued from previous page)

```
  9 10 11 12
 13 14 15  0

- : unit = ()

# let b4 = MapRep.load 4 [9; 5; 6; 4; 13; 3; 1; 11; 0; 10; 12; 7; 14; 2; 15; 8] ;;
val b4 : MM.t = <abstr>
# MapRep.show b4 ;;

  9  5  6  4
 13  3  1 11
  0 10 12  7
 14  2 15  8

- : unit = ()

# MM.distance b3 b4 ;;
- : int = 24

(* This distance is large because the board b4 is nowhere near solved. *)
```

Note: You are allowed to use certain imperative coding idioms (array access, array update, *etc.*, but not `ref`s, `while` or `for` loops) in all the Manhattan distance code in this section, but not in the Hamming distance code. It's not absolutely necessary but it may make the code more efficient.

CHAPTER

SEVEN

PART C: THE ASTAR MODULE

[**10 marks**]

This section is worth 10 marks total. The code for this section will be written in the file `astar.ml`.

In this section you'll be implementing the board solver. Actually, the code you'll write in this section is more general than that; it implements the "A*" algorithm, which (like depth-first search or breadth-first search) is a general search algorithm for problems that can be described in a particular way.

The file `astar.mli` includes the following signatures of the `Task` module type and the `AStar` functor:

```
module type Task =
  sig
    type t

    val compare : t -> t -> int
    val eval    : t -> t -> int
    val next    : t -> t list
    val display : t -> string
  end

module AStar (T : Task) :
  sig
    exception No_solution

    val solve : T.t -> T.t -> T.t list
  end
```

Notice that there is no mention of `BoardRep`s or `BoardMetric`s in the code. Instead, the `AStar` algorithm can operate on any problem which conforms to the module type `Task`, which has a very simple interface. A `Task` consists of some type `t` (in our case, representing an N-puzzle board), as well as several functions on that type.

Note: In what follows, we will use the term "state" to mean "task state" *i.e.* a value that has type `t` in a module whose signature conforms to the `Task` module type. You can think of a task state as the state of an N-puzzle board.

The `Task` functions work as follows.

- `compare` takes two states and returns an integer (0 if equal, -1 or 1 if one is "less than" or "greater than" the other). This is only needed to define a `Map` module using `Task` states as the keys.
- `eval` compares two states for similarity; the first task will be the "solved" state, so `eval` returns an integer which describes how close the second state is from being solved. When `eval` returns 0, the state is solved.

- `next` takes a state and returns a list of states that can be derived from the first state by making a single move. (Note that the details of what a move is makes no difference to the solver.)
- `display` takes a state and returns a string representation of the state. (This is to help you with debugging; the solution code doesn't actually use this function.)

The files `board.ml` and `board.mli` (described in the next section) implement the `Board` module type, which is a superset of the `Task` module type. This allows N-puzzle boards to be used as task states, which in turn allows them to be solved using the A* algorithm.

The `AStar` functor can generate a solver module given a `Task` module. As far as external code is concerned, the solver module only contains two things:

- the `No_solution` exception, which is raised when the solver fails to find a solution
- the `solve` function, which takes a goal state and a starting state and either returns a solution (as a list of states, starting from the starting state and ending in the goal state) or raises the `No_solution` exception if a solution can't be found.

Of course, internally the solver module can have helper functions, other types, other exceptions, internal modules, *etc.*

We've supplied you with a partial implementation of the `AStar` functor, which is here (with comments removed):

```
module AStar (T : Task) =
  struct
    exception No_solution
    exception Solved of T.t list

    type qstate = {
      tstate  : T.t;
      history : T.t list;
      nmoves  : int;
      fitness : int
    }

    module Tqueue = MakePriorityQueue(struct
        type t = qstate
        let compare s1 s2 = Stdlib.compare s1.fitness s2.fitness
      end)

    module Tmap = Map.Make(T)

    type t = {
      queue : Tqueue.t;
      best  : int Tmap.t
    }

    let solve goal init =
      let init_eval  = T.eval goal init in
      let init_state =
        {
           tstate  = init;
           history = [];
           nmoves  = 0;
           fitness = init_eval;
        }
      in
```

(continues on next page)

(continued from previous page)

```
    let init_queue  = Tqueue.insert (Tqueue.empty) init_state in
    let init_best   = Tmap.empty in
    let init_solver = { queue = init_queue; best = init_best } in

    let rec iter { queue; best }  =
      failwith "TODO"
    in
      if init_eval = 0 then
        [init]
      else
        try
          iter init_solver
        with
          | Tqueue.Empty -> raise No_solution
          | Solved tlist -> tlist
end
```

Your job is to complete the code (by replacing the `failwith "TODO"` line with your code). Our solution was about 40 additional lines of code. In the rest of this section, we'll be going through the code and the algorithm in some detail. We won't give examples of the use of the `solve` function, because (a) the test script already does that, and (b) you can run the `npuzzle` program itself to test the solver as described above.

Note: You are not required to use all the code we've given you in the `solve` function; you can write `solve` any way you want as long as it has the correct type and works correctly. You *are* required to use the rest of the code in `solve.ml`. In particular, we want you to use the priority queue `Tqueue`, the map `Tmap`, and the `qstate` type in an appropriate way.

7.1 Internal types, modules, and exceptions

Before writing the `solve` function, you need to know about several types, modules and exceptions that the function uses.

7.1.1 Exceptions

In addition to the `No_solution` exception discussed above, there is an internal exception called `Solved` which contains a state list (which should be a solution). It's raised when a solution is found, but it's caught before the `solve` function returns. The only purpose of this is to make it easy to break out of a deep computation as soon as you find a solution. (If you don't want to use this exception, you don't have to.)

7.1.2 The `Tqueue` priority queue module and the `qstate` type

The `Tqueue` module contains the type of the priority queue used in the search, along with functions on that type. (See the `pqueue.ml` and `pqueue.mli` files for a description of these functions, though you should be familiar with priority queues from assignment 6.) The priority queue doesn't just store task states; it stores `qstate`s. Here's the definition of the `qstate` type:

```
type qstate = {
  tstate  : T.t;
```

(continues on next page)

(continued from previous page)

```
  history : T.t list;
  nmoves  : int;
  fitness : int
}
```

A `qstate` is a task state augmented with a fitness (how "good" the state is, which depends on how close it is to the solved state and how many moves it took to get to this state from the initial state), the number of moves taken to get to that state, and the history of the state (the list of states that preceded that state). Note that the history is kept in reverse order, with the most recent states at the front (this is more efficient).

When we define the `Tqueue` module, we compare states on the queue using the `fitness` of a state. A `fitness` is the sum of the evaluation of the state (using the `eval` function on states) and the number of moves needed to get to that state from the initial state. (The `eval` function will ultimately be derived from one of the `BoardMetric distance` functions.) The smaller the fitness, the better, so states with lower fitness values will be located closer to the "top" of the priority queue (because better states are searched first).

The priority queue defines a function called `pop_min` which pops the state with the lowest fitness value off the queue and also returns the updated queue.

7.1.3 The `Tmap` module

When doing a search, you don't want to repeatedly do the same search, especially if you are guaranteed that the repeated search will not give you better results. In our case, if we have a particular state which has already been searched over, you might think that there is no point in searching that state again. However, that isn't always true. What matters is the `fitness` of the queue state. Even if the same states always evaluate to the same distance from the solved state, they may have taken a different number of moves to get there, which will affect the fitness. Therefore, we create a module called `Tmap` which maps states to the number of moves taken to get to that state, and we create a map called `best` from this module. This map keeps track of the smallest number of moves we have needed to get to each state we have examined in the search.

When we are examining a state, we know the number of moves that have occurred between the initial state and the current state. We check the `best` map to see if the state is in the map. If not, we add it as a (key, value) pair with the key being the state and the value being the move count; this returns a new `best` map. If the state is already in the `best` map, we check the number of moves for that state in the map; if the count stored in the map is greater than the new count, we update the map with the new best count for that state. Otherwise, we leave the map unchanged.

We use the `best` map to decide whether or not it's worth adding a state to the priority queue (more on this below).

7.1.4 The solver type `t`

We've mentioned the priority queue and the `best` map above; it shouldn't surprise you, then, that the definition of the solver type `t` is:

```
type t = {
  queue : Tqueue.t;
  best  : int Tmap.t
}
```

In other words, a solver is a priority queue of `qstate`s along with the `best` map between states and move counts.

7.2 Algorithm walkthrough

OK, now that we have all the elements, we are ready to describe the A* algorithm itself. (This is also described in comments in the `astar.ml` file, for convenience.)

The invariant is that the task states in the `qstate` values on the queue represent unsolved states. (If they are solved, they shouldn't have been put on the queue.) We repeat the following steps:

1. Pop a `qstate` value off the queue. If the queue is empty, there is no solution. In that case, raise a `No_solution` exception.
2. Check if the state in the popped queue state is a key in the `best` map.
    - If it isn't, add it to the `best` map as a key with the number of moves leading to the state state (the `nmoves` field of the queue state) as the value. Go to the next step.
    - If it is, compare it to the move count in the map. If the new number of moves is smaller than the one in the map, replace the binding in the map and go to the next step. Otherwise, this state has already been searched for with a smaller (or at least no larger) number of moves, so there is no point in searching it again; discard it and restart the loop (step 1).
3. Assuming we got this far, compute the next `qstate`s from the current `qstate`. Check to see if any of them contains a state which is a solution; if so, compute the final list of states (which should have the solved state last, not first) and raise a `Solved` exception to break out of the loop. Otherwise, add them back into the queue and continue.

To start off the search, we compute an initial queue state and add it to an empty priority queue, and we create an initial empty `best` map. We use these to create the initial solver state. To maintain the invariant, we also check for the case where the initial state is already solved; if so, we simply return a singleton list with the initial state. Otherwise, we start the loop. (This initialization code is provided for you.) In the template code, the loop is the `iter` helper function, which is what you need to implement.

7.3 Testing

As we mentioned above, use the test scripts to check that your algorithm implementation works as expected. In addition, running the `npuzzle` program can be used to check the algorithm, in particular with some number of random moves to scramble the initial setup (the `-r` command-line option). You should try to use at least 100 random moves to get a good scramble. Very large numbers of random moves (say, more than 500) can produce boards which take a very long time to solve for sizes 4x4 and up. Nevertheless, all boards that are generated using random initial moves should be solvable, so if one isn't, you have a bug. In contrast, some of the pre-loaded boards in `main.ml` (which you load using the `-l` command-line option) are intentionally unsolvable; your program should definitely indicate as much if you load those board configurations.

7.4 Functions we used

Despite the complexity of the algorithm, you don't have to use many functions from `Tmap` (derived from the Map.S module) or `Tqueue` (from the `Pqueue` module in `pqueue.ml`). We only used these:

- `Tmap` : `find_opt`, `add` and the `empty` value
- `Tqueue` : `insert`, `pop_min` and the `Empty` exception

(This includes the supplied code.)

CHAPTER

EIGHT

THE REST OF THE CODE

You are required to complete the implementation of only three files in the program. You might wonder what's in the rest of the files. Those files include:

- `boardrep.mli`, `boardmetric.mli`, and `astar.mli` (these contain module types and the signatures of functors that we've discussed above)
- `pqueue.ml` and `pqueue.mli`
- `board.ml` and `board.mli`
- `npuzzle.ml` and `npuzzle.mli`
- `main.ml`

For completeness, and to satisfy your curiosity, we'll briefly describe here the files we haven't previously discussed.

The `pqueue.ml` and `pqueue.mli` files implement a priority queue much like the one you implemented in assignment 6. This uses the `BatHeap` module from the `batteries` package you installed above. If you're curious, the documentation of this module can be found here.

The `board.ml` and `board.mli` files define the `Board` module type and the `Make` functor, which makes a `Board` module given a `BoardRep` module and a `BoardMetric` module. Modules created using `Make` implement game boards with much more functionality than the `BoardRep` modules. For instance, there are functions to

- randomize the board
- compare boards for ordering purposes (which just uses `Stdlib.compare`, so it's nothing fancy like it was in assignment 7)
- generate all possible boards that can be derived from a board by making a single move (needed for the search process of a solver)
- interactively solve a board with user input
- evaluate a board for how close it is to the solution
- print the state of the board
- check that a board is in a valid state (for testing)

The `Make` functor takes both a `BoardRep` module and a `BoardMetric` module as inputs (which is why it can evaluate the board). Most importantly, the `Board` module type is a superset of the `Task` module type defined in `astar.mli`, so we can use a `Board` module as the input to the `AStar` functor to generate a board solver.

The `npuzzle.ml` and `npuzzle.mli` files contain a module type `Solver` and the `NPuzzle` functor which generates board solvers given a `BoardRep` and a `BoardMetric`. Internally, the module uses the `AStar` module to do the solving. So this module "wires up" all the components you've defined to create a working solver. (This code is actually quite routine and uninteresting, which is why we didn't make you implement it yourselves.) These files also include code to print solutions and to check that solutions are valid (for testing).

Finally, `main.ml` does the command-line argument processing and includes the definition of a number of sample boards.

[END OF EXAM]