Problem Set: The search for intelligent solutions

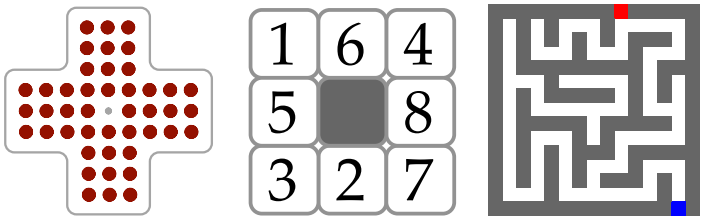
Stuart M. Shieber

March 24, 2023

In this assignment, you will apply your knowledge of OCaml modules and functors to complete the implementation of a program for solving search problems, a core problem in the field of artificial intelligence. In the course of working on this assignment, you'll implement a more efficient *queue module* using two stacks; create a *higher-order functor* that abstracts away details of search algorithms and puzzle implementations; and compare, visualize, and analyze the *performance* of various search algorithms on different puzzles.

1 Search problems

The field of ARTIFICIAL INTELLIGENCE pursues the computational emulation of behaviors that in humans are indicative of intelligence. A hallmark of intelligent behavior is the ability to figure out how to achieve some desired goal. Let's consider an idealized version of this behavior – puzzle solving. A puzzle can be in any of a variety of STATES. The puzzle starts in a specially designated INITIAL STATE, and we desire to reach a GOAL STATE by finding a sequence of MOVES that, when executed starting in the initial state, reach the goal state. Figure 1 provides some examples of this sort of puzzle – peg solitaire, the 8-puzzle, and a maze puzzle.



(a) (b) (c)

Figure 1: Some puzzles based on search for a goal state. (a) the peg solitaire puzzle; (b) the sliding-tile 8 puzzle; (c) a maze puzzle.

A good example is the 8 puzzle, depicted in Figure 2. (You may know it better as the 15 puzzle, its larger 4 by 4 version.) A 3 by 3 grid of numbered tiles, with one tile missing, allows sliding of a tile adjacent to the empty space. The goal state is to be reached by repeated moves of this sort. But which moves should you make?

Solving goal-directed problems of this sort requires a SEARCH among all the possible move sequences for one that achieves the goal. You can think of this search process as a walk of a SEARCH TREE, where the nodes in the tree are the possible states of the puzzle and the

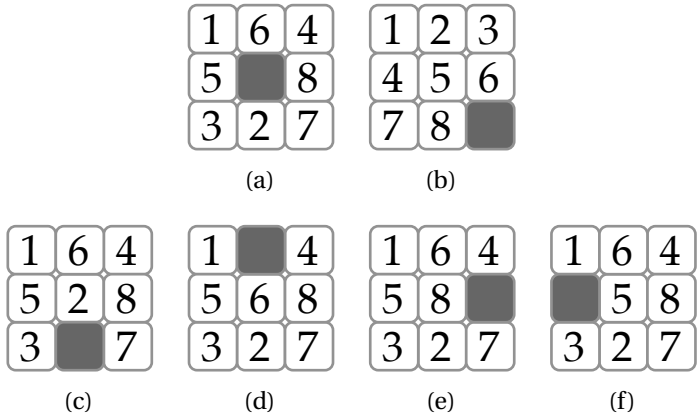


Figure 2: The 8 puzzle: (a) an initial state, (b) the goal state, (c-f) the states resulting from moving up, down, left, and right from the initial state, respectively.

directed edges correspond to moves that change the state from one to another. Figure 3 depicts a small piece of the tree corresponding to the 8 puzzle.

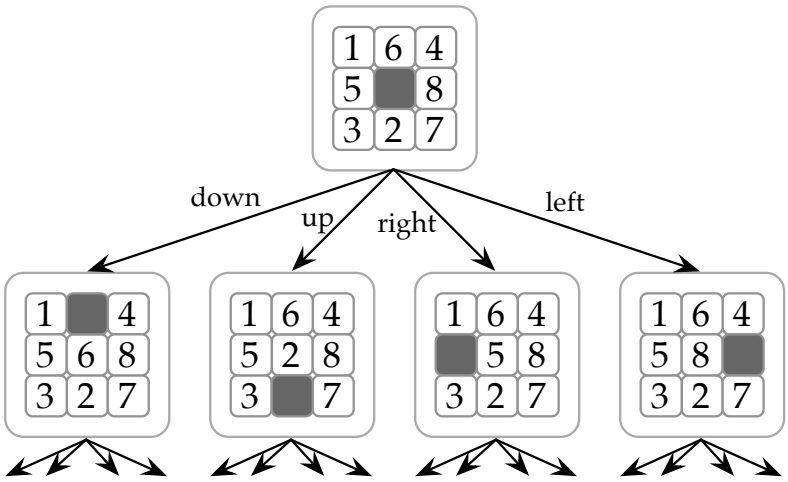


Figure 3: A snippet from the search tree for the 8 puzzle.

To solve a puzzle of this sort, you maintain a collection of states to be searched, which we will call the *pending* collection. The pending collection is initialized with just the initial state. You can then take a state from the pending collection and test it to see if it is a goal state. If so, the puzzle has been solved. But if not, this state's NEIGHBOR states – states that are reachable in one move from the current state – are added to the pending collection (or at least those that have not been visited before) and the search continues.

To avoid adding states that have already been visited before, you'll need to keep track of a set of states that have already been visited, which we'll call the *visited* set, so you don't revisit one that has already been visited. For instance, in the 8 puzzle, after a down move, you don't want to then perform an up move, which would just take you back to

where you started. (The standard OCaml `Set` library will be useful here to keep track of the set of visited states.)

Of course, much of the effectiveness of this process depends on the order in which states are taken from the collection of pending states as the search proceeds. If the states taken from the collection are those most recently added to the collection (last-in, first-out, that is, as a stack), the tree is being explored in a DEPTH-FIRST manner. If the states taken from the collection are those least recently added (first-in, first-out, as a queue), the exploration is BREADTH-FIRST. Other orders are possible, for instance, the states might be taken from the collection in order of how closely they match the goal state (using some metric of closeness). This regime corresponds to BEST-FIRST or GREEDY SEARCH.

2 *Structure of the provided code*

We have provided code to start off the process of building a general puzzle-solving system that allows for different search regimes applied to different puzzle types. You'll want to familiarize yourself with the provided code.

You will work with the following signatures, modules, and functors. Some of the important components are listed by the file they are found in. Components that you will write are shown *in italics*.

- `collections.ml`
  - `COLLECTION` – A signature for collections of elements allowing adding and taking elements
  - `MakeStackList` – A functor for generating a stack Collection implemented using lists
  - `MakeQueueList` – A functor for generating a queue Collection implemented using lists
  - `MakeQueueStack` – *A functor for generating a queue Collection, where the Queue is implemented using two stacks*
- `puzzledescription.ml`
  - `PUZZLEDESCRIPTION` – A signature for specifications of puzzles, providing information about the states and moves, initial and goal states, the neighbors structure of the puzzle, etc.
- `mazes.ml` – A specification of maze puzzles satisfying `PUZZLEDESCRIPTION`
- `tiles.ml` – A specification of tile puzzles satisfying `PUZZLEDESCRIPTION`

- `puzzlesolve.ml`
  - `PUZZLESOLVER` – A signature for puzzle solvers
  - `MakePuzzleSolver` – *A higher-order functor for generating different puzzle solvers*

2.1 Collections

The file `collections.ml` provides a signature for `COLLECTION` modules. A `COLLECTION` module, which is very similar to the `OrderedCollection` module in Problem Set 4, is a data structure with types `elt` and `collection`, and functions `take` and `add`. The `take` function removes an `elt` from a `collection`, while `add` inserts an `elt` into a `collection`. Collections are a generalization over a variety of data structures including stacks, queues, and priority queues.

Collections will be used to store the states reachable from the initial state that have yet to be explored, as the puzzle solver searches the space of states in order to find a goal state. By using different collections that implement different orderings in how elements are taken from the collection, different search regimes are manifested and different efficiencies of the search process can result.

We have provided two functors for especially simple implementations of `COLLECTION`, one that implements a stack represented internally as a list, and one that implements a queue represented internally with a list. You will implement a third functor `MakeQueueStack`, that implements a queue represented internally with *two stacks*, similar to the approach introduced in Section 15.5.2. (However, you should be able to provide an implementation more elegant and more purely functional than the examples there. There's no reason to introduce imperative programming techniques for `MakeQueueStack`.)

Problem 1

Implement the `MakeQueueStack` functor. You'll want to test it fully as well.

In theory, the implementation in `MakeQueueStack` should be more efficient than that of `MakeQueueList`. In Section 3, you will experiment with that hypothesis and provide a report on your findings as part of the problem set.

2.2 Puzzle description

A module signature of a puzzle description, called `PUZZLEDESCRIPTION`, specifies an interface that includes data types for the states and moves; an initial state; a predicate for goal states; and functions for generating neighbors, for executing move sequences, and for visualizing the puzzle. You will want to read and understand it to

familiarize yourself with the functionality that a `PUZZLEDESCRIPTION` provides.

To understand the notion of moves and states, consider a simple maze. In a maze, the state would be represented as the current position in the maze, and the moves would allow you to change your position by walking up, down, left, or right. The neighbors of a state would be the positions that you could move to in one step using those moves, keeping in mind that not all four move types are possible from a given state. Similarly, in the 8 puzzle, a state is a configuration of the tiles and a move is the swapping of the empty tile with one of its four adjacent tiles, the adjacent tile moving up, down, left, or right.

We've provided two implementations of the `PUZZLE_DESCRIPTION` signature, one for tile puzzles like the 8 puzzle, and one for maze puzzles. Examining the files `tiles.ml` and `mazes.ml` should provide further understanding of how these puzzles, and the search problems they engender, are being represented.

2.3 Puzzle solving

In file `puzzlesolve.ml`, we define a `PUZZLESOLVER` signature, which provides a `solve` function to solve a puzzle described as a `PUZZLEDESCRIPTION`. The type of the `solve` function is `unit -> move list * state list`. When called (by applying it to unit), it returns a pair. The first element of the pair is a solution to the puzzle, a list of moves that when executed on the initial state reaches a goal state. The second element is a list of all the states that were ever visited in the search process in looking for a goal state, in any order. (These are just the elements of the visited set; the `Set.S.elements` function may come in handy for generating this from the visited set.)

The search process will thus need to maintain several data structures:

- A collection of states[1] that are *pending* examination (initialized with just the initial state); and
- A set of all states that have been *visited* (that is, that have ever been removed from the pending collection for examination), initialized with the empty set.

The search proceeds by taking a state from the pending collection (call it the *current state*). If the current state has already been visited (that is, it's in the visited set), we can move on to the next pending state. But if it has never been visited, we add it to the visited set and examine it further. If it is a goal state, the search is over and appropriate information can be returned from the search. If not, the neighbors of the

[1] A subtlety about the pending collection. Instead of storing as its elements just the states, it is helpful to store both a state and the list of moves it took to get to that state. Thus, the elements will be of type `state * move list`. That way, when a goal state is found, you'll know the path to get there as well.

current state are generated and added to the pending collection, and the search continues.

You will implement a functor `MakePuzzleSolver` that maps a collection implementation (a functor that generates modules satisfying the signature `COLLECTION`) and a puzzle description (a module satisfying the `PUZZLEDESCRIPTION` signature) to a module implementing the `PUZZLESOLVER` signature. The `solve` function in the generated `PUZZLESOLVER` module solves the puzzle described in the `PUZZLEDESCRIPTION` using a search that uses the provided collection regime.

The type signature of `MakePuzzleSolver` is

```
(functor (sig type t end -> COLLECTION))
  -> PUZZLEDESCRIPTION
  -> PUZZLESOLVER
```

Notice that `MakePuzzleSolver` is a functor that takes a functor as its argument. It is a higher-order functor! (You may not have realized that was even possible in OCaml.) The idea is that the functor `MakeCollection` is used for generating the collection for storing pending states that have yet to be searched. Using different collection regimes – stacks (`MakeStackList`), queues (`MakeQueueList`, `MakeQueueStack`), etc. – leads to different search regimes – depth-first, breadth-first, etc. The argument functor can be used to provide a collection of elements of any type. We recommend that you use it to store the collection of pending elements of type `state * (move list)`. (See footnote 1.) Each such element pairs a state remaining to be searched with the list of moves that was used to reach it starting from the initial state.

The `PUZZLESOLVER` signature provides for an exception `CantReachGoal` for the solver to raise if it can't find a solution to the puzzle.

Problem 2

Implement the `MakePuzzleSolver` functor. Again, you'll want to test it fully.

3 Testing, metering, and writeup

We've provided two sample puzzles, an 8-puzzle and a maze puzzle, in files `tiles.ml` and `mazes.ml`, respectively. Once you've implemented the solver (Section 2.3) you should be able to solve simple puzzles of these sorts. The file `tests.ml` has code to generate some maze and tile puzzles and solve them with various solvers. Once you've completed the first two problems in the problem set, you should be able to build `tests.byte` and see the puzzle solvers in action.

Running the tests uses OCaml's graphics to display the puzzles and even animates the search process. After each display, press any key to

move on to the next display. Figure 4 shows the display of the maze solver as it animates the solution of a maze.

This code should provide some inspiration for your own systematic testing of the solver on these kinds of puzzles. You can even implement your own puzzles if you're of a mind too. (Extra karma!)

You can try out both depth-first and breadth-first search for solving the puzzles by using different collection functors in the solver. With your two-stack implementation of queues, you can experiment with the relative efficiency of the two queue implementations.

Problem 3

In order to assess the benefit of the additional implementation of collections, you should design, develop, test, and deploy a performance testing system for timing the performance of the solver using different search regimes and implementations. Unlike in previous problem sets, we provide no skeleton code to carry out this part of the problem set (though `tests.ml` may be useful to look at). You should add a new file `experiments.ml` that carries out your experiments. The design of this code is completely up to you. This part of the problem set allows you the freedom to experiment with *ab initio* design. The deliverable for this part of the problem set, in addition to any code that you write, is a report on the relative performance that you find.

You'll want to write some OCaml code to time the solving process on appropriate examples with several collection implementations. There are some timing functions that may be useful in the `Absbook` module (`absbook.ml`), which we've used in `tests.ml`. You may also find the `time` and `gettimeofday` functions in the `Sys` and `Unix` modules useful for this purpose.

We recommend that to the extent possible any code that you write for performing your testing not change the structure of code we provide, so that our unit tests will still work. Ideally, all the code for your experimentation should be in new files, not the ones we provided.

You should generate your writeup with the rest of your code as a raw text, Markdown, or LaTeX file named `writeup.txt`, `writeup.md`, or `writeup.tex`. (We recommend these markup-based non-wysiwyg formats for reasons described here. For Markdown in particular, there are many tools for writing and rendering Markdown files. Some of our favorites are: ByWord, Marked, MultiMarkdown Composer, Sublime Text, and the Swiss army knife of file format conversion tools, pandoc.) If you use Markdown or LaTeX, you should include the rendered version as a PDF file `writeup.pdf` in your submission as well.

This portion of the problem set is purposefully underspecified. There is no "right answer" that we are looking for. It is up to you to determine what makes sense to evaluate the performance of the code. It is up to you to decide how to present your results in clear, well-structured, cogent prose and appropriate visualizations of the experimental results.

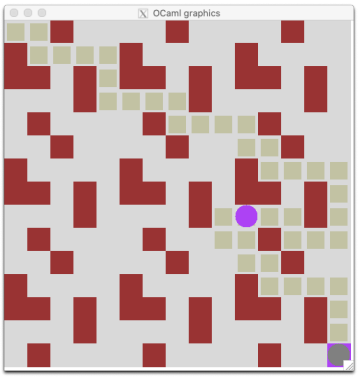


Figure 4: A frame from an animation of the maze solver. The initial position of the maze is in the upper left, and the goal in the lower right, marked with a grey circle on purple background. The current position is marked with the purple circle slightly southeast of the center. The maze walls are brick red, and the visited positions are marked with small squares.

Version information: Commit 13e4d5e from 2023-03-24 by enscma2. CI build of I_TAG ().