CS51 PROBLEM SET 5: THE SEARCH FOR INTELLIGENT SOLUTIONS

STUART M. SHIEBER

Contents

You may work with a partner on this problem set. If you choose to work with a partner, you can find instructions for configuring GitHub Classroom in the CS51 Reference or CSCI E-51 Reference.

1. Overview

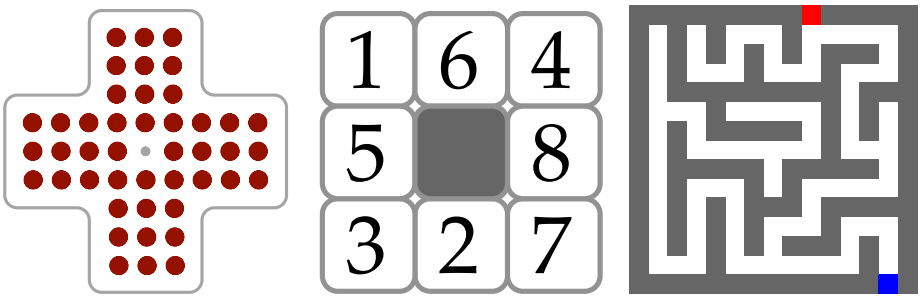
In this assignment, you will apply your knowledge of OCaml modules and functors to complete the implementation of a program for solving search problems, a core problem in the field of artificial intelligence. In the course of working on this assignment, you'll implement a more efficient *queue module*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

Questions to consider: The "questions to consider" in this writeup are not graded (though the numbered problems are!). They are provided primarily to help you assess your understanding of the material. You aren't expected to include answers to them in your submission, but the concepts they address could arise on exams. Answers to most of the questions are in the comments of the problem set.

Testing: Unit testing is required, as always. Please see `tests.ml`, where we've provided some tests of the game solver using the tile and maze puzzles.

Partners: You are allowed (and encouraged) to work with a partner on this assignment. You are also allowed to work alone, if you prefer. If you are an extension student, you are free

Date: March 23, 2018.



(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.

to pair up with another extension or on-campus student. For more detailed instructions, look here.

Compilation and Submitting: As usual, make sure to commit and push early and often. By sharing your remote repository with your partner, you can more effectively collaborate on maintaining your shared code base. When submitting to Gradescope, you must add your partner on your submission *every time you submit*–if you submit without adding your partner, your most recent submission will not include them.

Downloading: To download the provided materials for this problem set, first create your repository in GitHub Classroom for this homework by following this link. Then, follow the GitHub Classroom instructions found here. Note that, now that you are working with a partner, you should endeavor to keep up-to-date with your partner's code. Run `git pull` to fetch any changes, and `git push` to upload your changes to the repository. It's generally a good practice to `pull` before you `push`, so you can resolve any conflicts between your and your partner's code. More information about Git and its use can be found in Eddie Kohler's guide to Git and in Brian Yu's Git video.

2. 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.

STATE
INITIAL STATE
GOAL STATE
MOVE

A good example is the 8 puzzle, depicted in Figure 2. (You may know it better as the 15 puzzle, the 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?

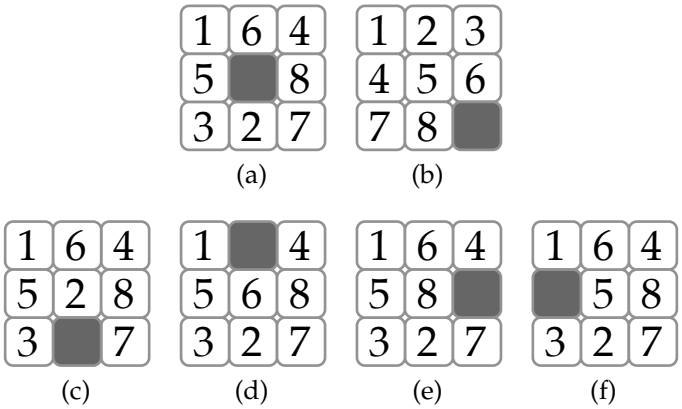


FIGURE 2. The 8 puzzle: (a) an initial state, (b) the goal state, (c-f) the states resulting from moving up, down, left, and right from the initial state, respectively.

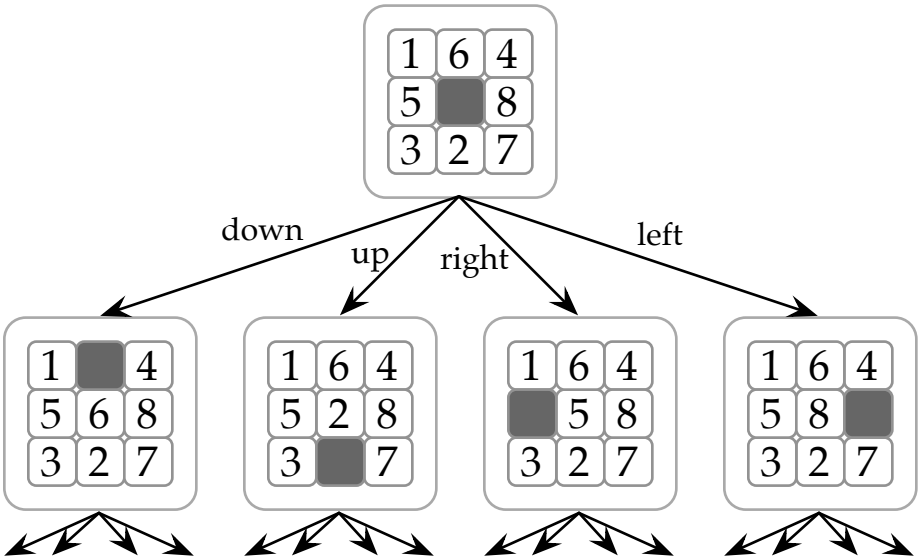


FIGURE 3. A snippet from the search tree for the 8 puzzle.

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 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.

To solve a puzzle of this sort, you maintain a collection of states to be searched. This collection is initialized with just the initial state. You can then take a state from the 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 collection (or at least those that have not been visited before) and the search continues. (This requires that you

also keep track of a set of states that have already been visited, 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 which order 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.

DEPTH-FIRST SEARCH
BREADTH-FIRST SEARCH
BEST_FIRST SEARCH
GREEDY SEARCH

3. 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*
- `gamedescription.ml`
  - `GAMEDESCRIPTION` – A signature for specifications of games, providing information about the states and moves, initial and goal states, the neighbors structure of the game, etc.
- `mazes.ml` – A specification of maze puzzles satisfying `GAMEDESCRIPTION`
- `tiles.ml` – A specification of tile puzzles satisfying `GAMEDESCRIPTION`
- `gamesolve.ml`
  - `GAMESOLVER` – A signature for game solvers
  - `MakeGameSolver` – *A higher-order functor for generating different game solvers*

3.1. 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 set of states reachable from the initial state that have yet to be explored, as the game 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 lecture 10. (However, you should be able to provide an implementation more elegant and more purely functional than the examples from the lecture. 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 4, you will experiment with that hypothesis and provide a report on your findings as part of the problem set.

3.2. **Game description.** A simple module signature of a game description, called `GAMEDESCRIPTION`, 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 game. You will want to read and understand it to familiarize yourself with the functionality a `GAMEDESCRIPTION` 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 `GAME_DESCRIPTION` signature, one for tile games 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.

3.3. **Game solving.** In file `gamesolve.ml`, we define a `GAMESOLVER` signature, which provides a `solve` function to solve a game described as a `GAMEDESCRIPTION`. The type of the `solve` function is `unit -> move list * state list`. When called (by applying it to unit), it returns a solution to the game, a list of moves that when executed on the initial state reaches a goal state, along with a list of all the states that were ever examined in the search process in looking for a goal state.

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

- A collection of states that are *pending* examination (initialized with just the initial state);
- A set of all states that have been *visited* (that is, that have ever been added to the pending collection), again initialized with just the initial state;
- A list of all states that have ever been *expanded* (that is, examined for determination if they are a goal state), initially empty.

The search proceeds by taking a state from the pending collection (call it the *current state*) and examining the current state to see if it is a goal state (and adding it to the expanded list). If it is a goal state, appropriate information can be returned from the search. If not, the neighbors of the current state are generated. Those neighbors that are previously unvisited are added to the pending collection, and they are added to the visited set as well.

You will implement a functor `MakeGameSolver` that maps a collection implementation (implemented as a functor that generates modules satisfying the signature `COLLECTION`) and a game description (a module satisfying the `GAMEDESCRIPTION` signature) to a module implementing the `GAMESOLVER` signature. The `solve` function in the generated `GAMESOLVER` module solves the game described in the `GAMEDESCRIPTION` using a search that uses the provided collection regime.

The type signature of `MakeGameSolver` is

```
(functor(sig type t end -> COLLECTION)) -> GAMEDESCRIPTION -> GAMESOLVER
```

.

Notice that `MakeGameSolver` 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 argument functor can be used to provide a collection of elements of any type. We recommend that you use it to store a collection of elements of type `state * (move list)`. Each such element is a state reachable by executing the associated list of moves starting from the initial state.

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

Problem 2. *Implement the `MakeGameSolver` functor. Again, you'll want to test it fully.* □

4. Testing, metering, and writeup

We've provided two sample games, an 8-puzzle game and a maze puzzle, in files `tiles.ml` and `mazes.ml`, respectively. Once you've implemented the solver (Section 3.3) 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. 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 `CS51` *module (*`cS51.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 why. 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.

5. Submission

Before submitting, please estimate how much time you and your partner each spent on the problem set by editing the lines in `collections.ml` and `gamesolve.ml` that look like:

```
let minutes_spent_collections () : int = failwith "not provided" ;;
let minutes_spent_gamesolve () : int = failwith "not provided" ;;
```

to replace the values of the functions with an approximate estimate of how long (in minutes) each section of the problem set took you to complete. If you and your partner spent different amounts of time on a part, provide the average of the two.

Make sure your code still compiles. Then, to submit the problem set, follow the Gradescope instructions found here. Don't forget to submit the writeup as well.