CMPSCI 187 / Fall 2015
Search

Due on December 2, 2015 4PM EST

Mark Corner and Gerome Miklau

CMPSCI 187

Contents

Overview

In this assignment, you will implement depth- and breadth-first search algorithms. Given a problem, your implementation will not only find a goal, but find a complete solution — a *path* to a goal. You will also implement a solution validator. You'll develop these implementations on a search problem we provide you (a maze), but you'll then model a simple puzzle (the 8-puzzle) as a search problem yourself.

Learning Goals

- Show understanding of search and search problems in a context with only local information.
- Show understanding of a solution validator for a general search framework.
- Demonstrate understanding of iterative versions of depth- and breadth-first search with only local information.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then communicate to the course staff immediately.

Start this assignment as soon as possible. Do not wait until 5pm the night before the assignment is due to tell us you don't understand something, as our ability to help you will be minimal.

Reminder: Copying partial or whole solutions, obtained from other students or elsewhere, is academic dishonesty. Do not share your code with your classmates, and do not use your classmates' code. If you are confused about what constitutes academic dishonesty you should re-read the course syllabus and policies. We assume you have read the course information in detail and by submitting this assignment you have provided your virtual signature in agreement with these policies.

You are responsible for submitting project assignments that compile and are configured correctly. If your project submission does not follow these policies exactly you may receive a grade of zero for this assignment.

Policies

- For some assignments, it will be useful for you to write additional java files. Any java file you write that is used by your solution MUST be in the provided `src` directory you export.
- The course staff are here to help you figure out errors (not solve them for you), but we won't do so for you after you submit your solution. When you submit your solution, **be sure to remove all compilation errors from your project**. Any compilation errors in your project will cause the autograder to fail, and you will receive a zero for your submission. **No Exceptions!**

Test Files

In the `test` directory, we provide several JUnit test cases that will help you keep on track while completing the assignment. We recommend you run the tests often and use them to help create a checklist of things to do next. *But you should be aware that we deliberately do not provide you the full test suite we use when grading*.

We recommend that you think about possible cases and add new `@Test` cases to these files as part of your programming discipline. Simple tests to add will consider questions such as:

- Do your methods handle edge cases such as integer arguments that may be positive, negative, or zero. Many methods only accept arguments that are in a particular range.
- Does your code handle unusual cases, such as empty or maximally-sized data structures?

More complex tests will be assignment-specific. To build good test cases, think about ways to exercise methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case. Note that we will not be looking at your test cases (unless otherwise specified by the assignment documentation), they are just for your use.

Before submitting, make sure that your program compiles with and passes all of the original tests. If you have errors in these files, it means the structure of the files found in the `src` directory have been altered in a way that will cause your submission to lose some (or all) points.

Problem 1

Import Project into Eclipse

Begin by downloading the starter project and importing it into your workspace. It is very important that you **do not rename** this project as its name is used during the autograding process. If the project is renamed, your assignment will not be graded, and you will receive a zero.

The imported project may have some errors, but these should not prevent you from getting started. Specifically, we may provide JUnit tests for classes that do not yet exist in your code. You can still run the other JUnit tests.

The project should normally contain the following root items:

`src` This is the source folder where all code you are submitting must go. You can change anything you want in this folder (unless otherwise specified in the problem description and in the code we provide), you can add new files, etc.

`support` This folder contains support code that we encourage you to use (and must be used to pass certain tests). You must not change or add anything in this folder. To help ensure that, we suggest that you set the support folder to be read-only. You can do this by right-clicking on it in the package explorer, choosing Properties from the menu, choosing Resource from the list on the left of the pop-up Properties window, unchecking the Permissions check-box for Owner-Write, and clicking the OK button. A dialog box will show with the title "Confirm recursive changes", and you should click on the "Yes" button.

`test` The test folder where all of the public unit tests are available.

`JUnit 4` A library that is used to run the test programs.

`JRE System Library` This is what allows Java to run; it is the location of the Java System Libraries.

If you are missing any of the above or if errors are present in the project (other than as specifically described below), seek help immediately so you can get started on the project right away.

On Search

Search is a powerful and general technique for problem solving. Problems in many domains yield to search: scheduling, routing, constraint satisfaction, optimization, optimal game play, navigation, and planning, to name a few.

In lecture, we've examined the problem of search in a few contexts. Earlier in the semester, we showed how search could be used to mark "blobs" of adjacent squares in a grid. More recently, we showed how to implement search on the general *graph* abstraction.

For this assignment, you will be implementing search on a slightly different abstraction. Most search problems can be represented as graphs, where states in the problem correspond to vertices, and edges exist between vertices if one state is a successor of the other.

For example, in the game of chess (https://en.wikipedia.org/wiki/Chess), each possible arrangement of pieces is a state. As in most search problems, there is a single starting state. There are 20 successors to this state, each corresponding to one possible move by the first player (two for each pawn and two for each knight).

But there is a practical problem: Chess has an estimated 10^{123} states that can be reached through legal play. Building the entire graph to represent such a large *state space* is infeasible. Chess is a simple game and causes us this difficulty; many more complex search problems likewise have too large of a state space to fully translate to a graph.

But, we needn't necessarily create the entire graph. Instead, we can start at just the initial state, and look at its successors, and theirs (and so on), stopping when we reach a goal state. That is, we can simplify a search problem to three components:

- an *initial state*: the initial configuration of the problem we are trying to solve
- a *goal test*: given a state, this test determines if it is what we are searching for (for example, in chess, a board where we are victorious)
- a method to find a *list of successors* for a given state: given a state, enumerates the valid successors for this state (for example, in chess, the boards that result from each possible move from the current board)

In this assignment, you will adapt the graph-based breadth- and depth-first search covered in lecture and the book to this framework.

Examining the code

Take a look at the code in the project file. We'll describe the important bits here.

The `graphs` package is (a subset of) exactly the code we covered in class. It's used to to build mazes, one of the two search problems you'll be solving in this assignment.

The `mazes` package contains code to build random mazes (in `MazeGenerator`) and to represent them (in `Maze`). A maze consists of `Cell`s, which represent `x, y` coordinates in the maze (where the upper left is `0, 0` and the lower right is `width - 1, height - 1`).

The `Maze` class has a `toString` method which you may find helpful. Here is an example output of `toString` on a `Maze` of width and height three:

```
#0#1#2#
0  S  0
# # # #
1     1
# ### #
2  G  2
#0#1#2#
```

The starting cell (`1, 0`) is marked with an `S`; the goal cell (`1, 2`) is marked with a `G`. Cells that are adjacent (that is, are successors of one another) have empty space between them, and cells that are not have a wall, represented as `#`, between them. The borders of the maze contain the x coordinate (modulo 10) along the top and bottom, and the y coordinate along the left and right.

In this maze, one possible solution is (`1, 0`); (`0, 0`); (`0, 1`); (`0, 2`); (`1, 2`). This path represents the starting cell, a move left, a move down, a move down, and a move right, to the goal cell.

The `search` package contains classes related to the general implementation of search. The `SearchProblem` interface describes a search problem and the type of its associated state; `Maze` is a complete example of a search problem. `Searcher` is an abstract class describing the general functionality that will be required by breadth- and depth-first search implementations that operate on a `SearchProblem`. `RecursiveDepthFirstSearcher`, `StackBasedDepthFirstSearcher`, and `QueueBasedBreadthFirstSearcher` are subclasses of `Searcher` that do (or will) contain corresponding implementations. Notice that subclasses of `Searcher` don't just report a goal state was found: they find and return an explicit `List` of states, from the initial state to a goal state. Finally, `Solver` is a utility class that allows you to instantiate a single object and use it to solve a `SearchProblem` using an algorithm of your choice.

Finally, the `puzzle` package contains a stub class `EightPuzzle`, that you will use to build an implementation representing a new search problem, representing the 8-puzzle (a simplified version of the 15-puzzle; see https://en.wikipedia.org/wiki/15_puzzle).

What to do

There are a few small tests already, which you will likely want to add to. You may also want to write an interactive driver — the `main` method in `MazeDriver` should give you an idea of how to use the various classes together.

Any `Searcher` should be able to validate that the solution it found was valid. Start by implementing the `isValidSolution` method. **Do not** change any other public method in `Searcher`.

Once you have that working, the `main` method of the `MazeDriver` will run to completion, and show you the results of a recursive depth-first search on a random maze. You can vary the maze by changing the `width`, `height`, or `seed`.

Next, you'll need to finish the implementations of both `StackBasedDepthFirstSearcher` and `QueueBasedBreadthFirstSearcher`. But before you start, take a look at `RecursiveDepthFirstSearcher`.

Problem 1 continued on next page...

You'll see that it maintains a list of states it has already explored; this is analogous to the `GraphMarker` we presented in class, and is necessary to prevent a search algorithm from entering an infinite loop in some cases (do you know why?).

You'll also see that `RecursiveDepthFirstSearcher` constructs the path that it finds by passing the path along the call stack. Unfortunately, this approach won't work when you implement an iterative version of either depth- or breadth-first search. Instead, one possible approach is to maintain a list of known *predecessors* of each state as it's found by the search algorithm. We've shown an example of this in the (unused) `findSolutionWithExplicitPredecessors` method of `RecursiveDepthFirstSearcher`. You might choose to adapt this approach in your implementations of `StackBasedDepthFirstSearcher` and `QueueBasedBreadthFirstSearcher`. Alternatively, you can explicitly keep track of the path to each node as it is visited, though this will require your stack or queue to maintain not just vertex objects, but an object representing vertex / path pairs.

Once you have your various searchers working, it's time to implement a new search problem. Turn your attention to `EightPuzzle`. You should fill out each of the stub methods here. `getInitialState` and `isGoal` should be trivial, depending upon how you choose to represent a game state within `EightPuzzle`. `getSuccessors` will require that you understand the rules of the game, and return the complete set of successors of a given state, as a `List<List<Integer>>`.

Other notes

The default timeout (500 ms) set in the tests we've provided you suffices for those tests. But if you write tests involving larger search spaces, it may require that you change the timeout to be larger.

`RecursiveDepthFirstSearcher` can blow the stack (throw a `StackOverflowException`) if the call stack grows too deep. This is not an implementation error per se; some solvable instances of the `EightPuzzle` might cause this to happen.

Half of all possible `EightPuzzle` states will never lead to a solution. When testing, you might want to start with hand-crafted instances you know are solvable, rather than randomly generating them.

We will test your solvers on problems with no valid solution (that is, no path from the initial state to a goal); make sure you return an empty list (not **`null`**) in these cases.

Your implementations of `StackBasedDepthFirstSearcher` and `QueueBasedBreadthFirstSearcher` must perform the correct type of search iteratively; if submit the same code for each (or try to submit the code of `RecursiveDepthFirstSearcher` for either) **your submission will receive a zero**.

As usual, you can add new **`private`** methods to the classes in `src`, and you should complete the methods marked `TODO`, but you must not modify the method signatures of public methods, or change files in `support`.

Export and Submit

When you have completed the changes to your code, you should export an archive file containing the entire Java project. To do this, click on the `search-student` project in the package explorer. Then choose "File → Export" from the menu. In the window that appears, under "General" choose "Archive File". Then choose "Next" and enter a

destination for the output file. Be sure that the project is named **`search-student`**. Save the exported file with the `zip` extension (any name is fine).

Once you have the zip file ready, you can submit it to the online autograder. Go to autograder.markdcorner.com create an account with the correct email address and student id. Select the right project from the drop down and submit. It shouldn't take more than a minute to grade your project, but if there is a backlog of projects then it may take longer. If you encounter any errors that look like the autograder failed and not your project, please let the instructors know.