CS 142: Program design and methodology
Spring Term, 2026

Lab 5[1]

In this lab, you will implement the idea of *backtracking*, a technique for problems that involve making a series of choices. We'll be using the `Stack` class from Java's standard libary, with its characteristic `push` and `pop` operations.

Exploring a cave!

A very important use of the stack data structure is in backtracking, a process by which one investigates an option for solving a problem and then abandons it for another option when it does not lead to a solution. An intuitive way to explain this process is with hiking. Suppose you are hiking in the woods where there are many paths to follow and you are trying to reach a specific campsite to meet with some friends. There are many paths to take where one or two can lead to the campsite, whereas others will take you around the mountain to some other part of the woods. At each intersection, therefore, you must make a choice which path to follow. For example, if the path forks, do you go to the left or right? If there is a four-way intersection, do you go left, right or straight? Suppose you picked a sequence of turns and end up at a cliff, where there is a great view, but it is not the campsite. What do you do? Well, one response is to go back to where you most recently had to choose a direction, and select a different path. If you have tried all the paths from a given intersection, then you go back even further to the second to last intersection and try those alternatives. If none of those alternatives work either, then you go back one more intersection and so on. If you end up back at your starting point, then you know that none of the paths work.

In order to implement this strategy, you would need to keep track of the intersections so that you can go back to revisit your previous choices. This is where a stack comes in handy; keeping the choices in a stack allows us to revisit our most recent choices since they are the ones on top.

In this lab, you'll implement backtracking in the setting of a cave containing treasure. You want to explore the cave to find treasures, retracing your steps whenever you run out of new options. (This happens both at dead ends and at places where you've already explored all the ways forward.)

Begin by downloading the given code from the course website, unzipping it, and then opening its directory (File → Open). This will open the project. The code is all in `TreasureMap.java` in the `src` ("source") subdirectory. If you run it, the program should print out the following maze, which it read from the file `maze1`:

```
WWWWWW
W....W
WS..TW
WWWWWW
```

Here, the character `W` represents a wall, `S` is the starting position, `T` is a treasure, and the periods mark open spaces we haven't explored yet. It also prints a message about not finding the treasure.

Spend a bit of time looking through the code. There is a class `Point` to store the 2D coordinates of a location in the maze, plus code to read the maze in from a file and conveniently access it. Your code will go into the method `search`, which is already started. This method should perform a search from `start`, which is a `Point` with the starting location, and return the number of treasures found. So far, it's got code to create the `Set<Point>`[2] `found`, which will contain the locations of the treasures that you find, and a `Stack<Point>` called `stack` to keep track of the path you're currently exploring.

[1] Based on a lab by Jan Pearce.

[2] We measured performance of `Set` last lab, but haven't really talked about it. It models an unordered collection of items (`Point` objects in this case). It's main operations are `add` and `contains`. It also supports iterators.

Begin by entering a first version of the code to do the exploration:

```
stack.push(start);        //build path starting with starting location
while (!stack.empty()) {
   Point curr = stack.peek();                    //current location
   Point next = new Point(curr.x + 1, curr.y);  //try +x direction (right)
   if (canGo(next)) {     //make sure next isn't a wall or already visited
      stack.push(next);   //let's try this direction
      markVisited(next);  //mark so we know not to go there again
   } else
      stack.pop();  //can't go farther; backtrack
}
```

This code tries to explore in the +x direction (i.e. to the right) from the initial starting location. When the search runs into a wall, it will keep backtracking all the way to the starting location since it is only exploring in one direction.

To see the effect of this change, move the line `map.print()` from near the bottom of `main` to outside the braces so that the program prints the map after the search every time and not only when treasure is found. Then when you run the program, you should see a path in the output made of spaces rather than periods. These locations are the ones visited during the search. (Periods turn into spaces. If a treasure location is visited, its `T` is changed to lowercase.)

From the display, you can see that the treasure location is visited, but the program still doesn't realize it. To fix this, you'll want to add a check in the loop of `search` that checks whether the current location (`curr`) is a treasure location (use the method `isTreasure`). If so, add `curr` to `found`, which is supposed to be the set of treasures we've found. This will automatically change the return value of `search` so that the program knows you found a treasure.

Once the program realizes when it finds a treasure, change the map being searched by changing the `maze1` in the `main` method to `maze2`. This is a map where the starting location and treasure are not in the same row. Run the program to see the new map and verify that our search routine doesn't find the treasure.

Searching in only one direction clearly isn't what we want. For the next step, expand `search` so that if the search cannot proceed in the +x direction, it makes a candidate point in the +y direction (i.e. down) and goes that way if possible. This code should be very similar to the existing if statement that checks the +x direction. Note that you only want to try the +y direction if the search cannot go in the +x direction and you only want to pop from the stack (i.e. backtrack) if the search cannot proceed in either direction.

Next, verify that your new search code manages to find the treasure in `map2`. Once you've done that, add the other two directions (-x and -y) to the search routine in a way similar to how you added the +y direction. Then you can try more complicated maps such as `map3` or create your own. To do this, create a new Text File (File menu) and put in the description. It begins with 2 integers, the first giving the number of columns and the second giving the number of rows. Then comes the map itself using `W`, `S`, `T`, and period as described above. (Feel free to examine the given map files for examples and to modify them as desired.)

A final touch is to mark the paths used to reach the treasures. To do this, add the line `paths.addAll(stack)` to the code that recognizes treasure. This adds the current contents of `stack` to a set of points `paths` meant to store the paths. The `print` method already prints these locations using asterisks so that the paths are drawn.