420-406-AB Data Structures
Winter 2019

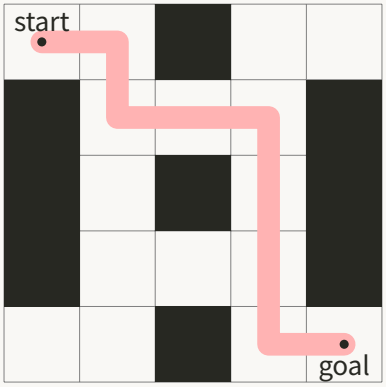
February 25th
Worth 5%

Assignment III: Search

(Due on Check on Léa)

Overview

You must solve the following problem, called *search*. Given a terrain, a grid of cells with free spaces and obstacles (walls), is it possible to move from a start position to a goal position. If it is possible, what is a *path*, or the sequence of moves, from the start to the goal?



1 Search

Search algorithms use a systematic method of visiting cells, by proceeding according to the rules of movement. Here, we can move one cell at a time in the UP, DOWN, LEFT and RIGHT directions. Search will proceed until the goal is found, or there is no more cells that can be reached from the start using these moves. If search finds the goal, we say that it's *reachable* and the algorithm outlines a path, or sequence of moves, to reach the goal.

We will used two search algorithms: depth-first search and breadth-first search.

To assist in searching, the terrain supports cell "coloring", which means assigning a color to a cell indicating it's status in the search. A cell marked WHITE is unvisited, a cell marked BLACK is visited and a cell marked GREY is seen, but not yet visited (breadth-first search only).

1.1 Depth-first Search (DFS)

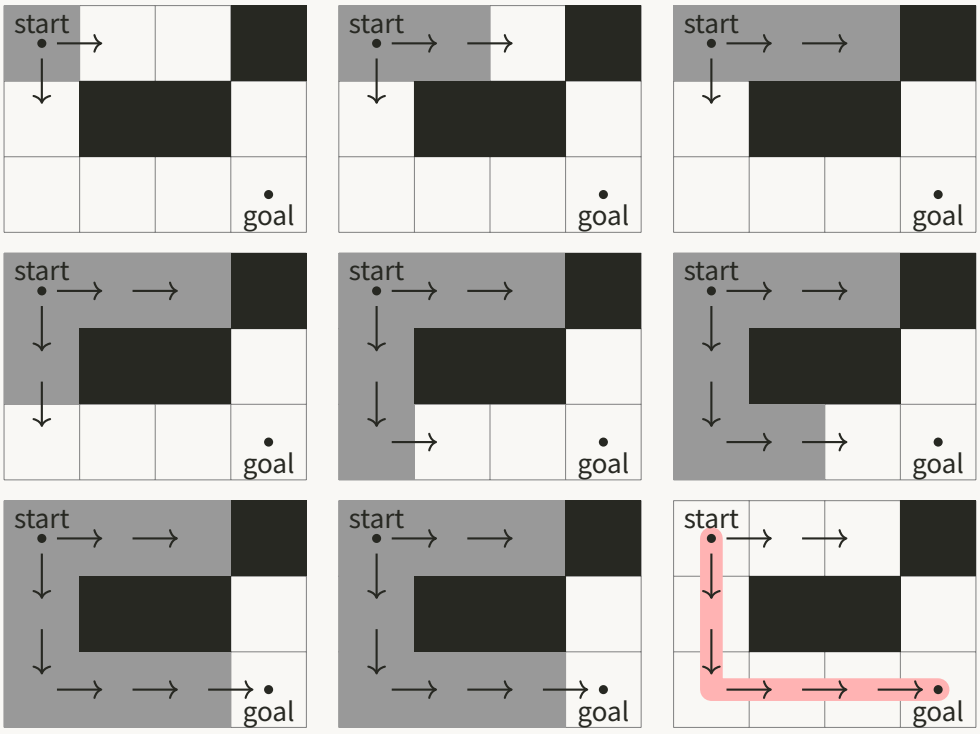
Starting from the **start** cell, the first strategy is to visit each of the cells in the following way, called *depth first*. This search strategy continues in one direction until there is nowhere else to go. It then goes back the way it came, called *backtracking*, and tries the next available direction with the same strategy. It's behaviour can be modelled with a `Stack`.

Depth-first visit. Perform the following:

1. Color the current cell BLACK (visited).
2. From the current cell, start in the UP direction.
3. If the cell in this direction is either:
   - off the terrain,
   - a wall, or
   - colored BLACK,

   then **skip** this direction.
4. Otherwise, **visit** this cell by moving to it and immediately *depth-first visit it*.
5. Repeat this action of skipping or visiting until you have tried all directions, proceeding clockwise, from the current cell.
6. **Backtrack** by returning back way you originally moved into the current position.

The following sequence of terrains show a DFS step-by-step:


start
goal
start
goal
start
goal
start
goal
start
goal
start
goal
start
goal
start
goal
start
goal

1.2 Breadth-first Search

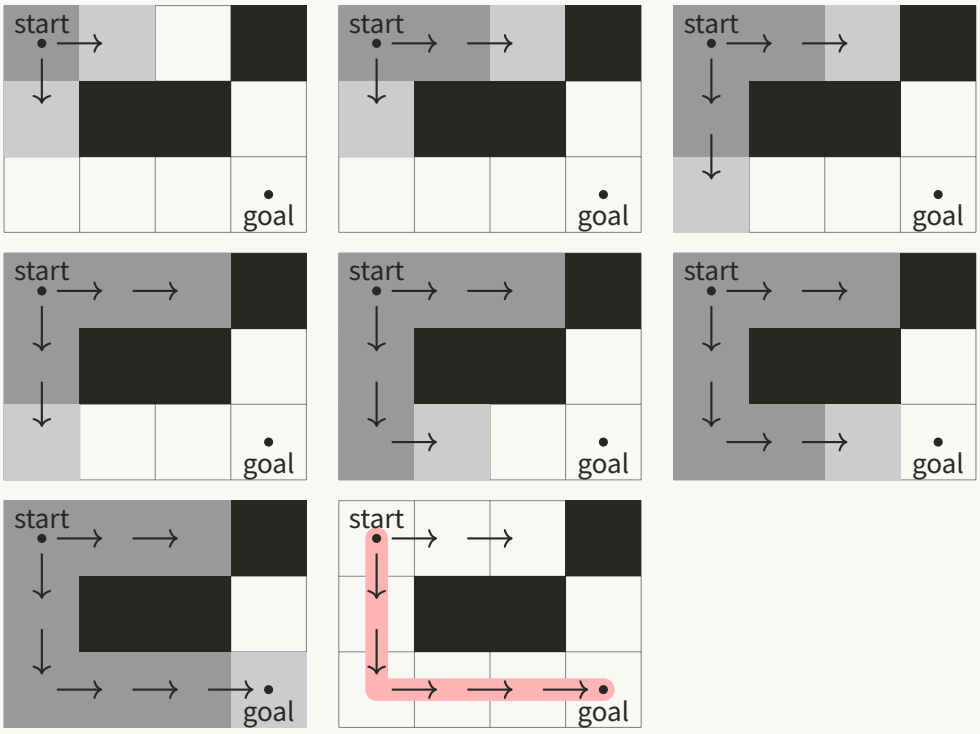
Starting from the **start** cell, the second strategy is to visit each of the cells in the following way, called a *breadth first*. Cell is visited in the order they are discovered, so the search strategy expands outwards from the starting cell in layers, like an onion. For example, the search will visit all cells 2 moves from the start before visiting cells that are 3 moves from the start. It's behaviour can be modelled with a `Queue`.

Breadth-first visit. Perform the following

1. Color the current cell BLACK (visited).
2. From the current cell, start in the UP direction.
3. If the cell in this direction is either:
   - off the terrain,
   - a wall, or
   - colored BLACK or GREY,

   then **skip** this direction.
4. Otherwise, **move** in this direction, color it GREY (seen), and *immediately backtrack to the current cell*.
5. Repeat this until for all directions, proceeding clockwise.
6. Go back, in turn, to each cell you have seen, but not visited (i.e.: GREY), since the beginning of the search. Do this in the order you discovered them, and breadth-first **visit** them.

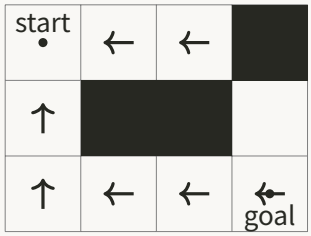
The following sequence of terrains show a BFS step-by-step:


start
goal
start
goal
start
goal
start
goal
start
goal
start
goal
start
goal
start
goal

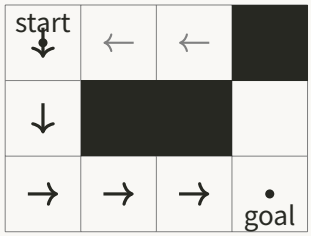
2 Paths

If a search successfully finds the goal cell from the start cell, we want to demonstrate the path from the start the goal. A convenient algorithm to accomplish this is the following.

1. During your search, record, for each visited cell, the direction of the cell that lead to it during search, called its **from** direction. For example, in the above example using BFS/DFS this would be:



2. Starting at the at the *goal*, work *backwards* towards the start by following the **from** directions.
3. At each cell, record the **to** direction as the opposite of the **from** direction.



At the end of the process, the path can be followed from the start to the goal by following the **to** directions.

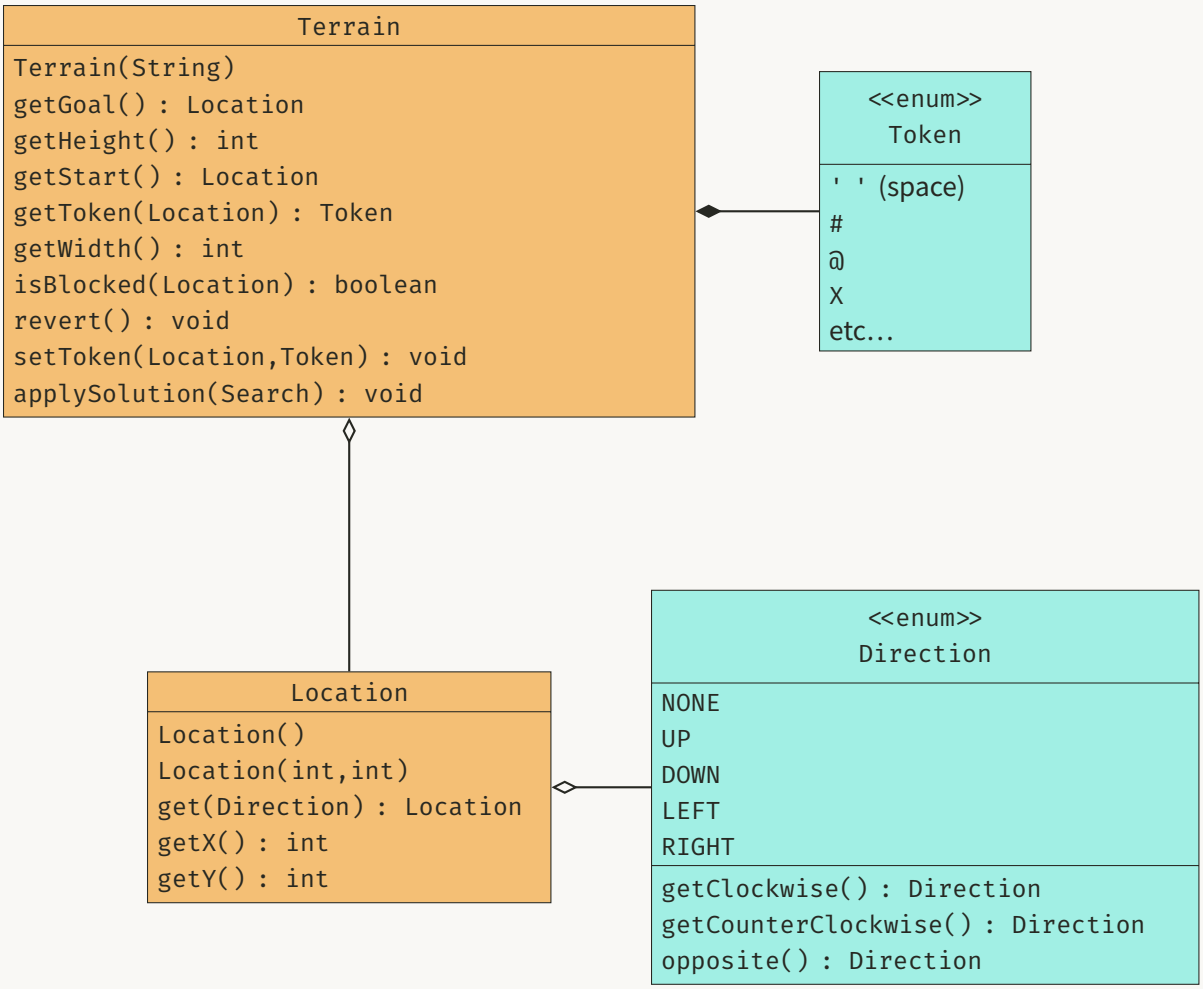
3 Design Overview

Implement the two search algorithm described in the previous sections, using the provided classes. The API documentation for these classes can be found in the `doc` directory of the starter. JavaDoc was used to document the API and generate these HTML pages[1].

Package `terrain`

The `terrain` package contains classes to represent a search problem. The primary class is the `Terrain`, used to load and represent the 2D grid with its walls, its starting point and its goal.

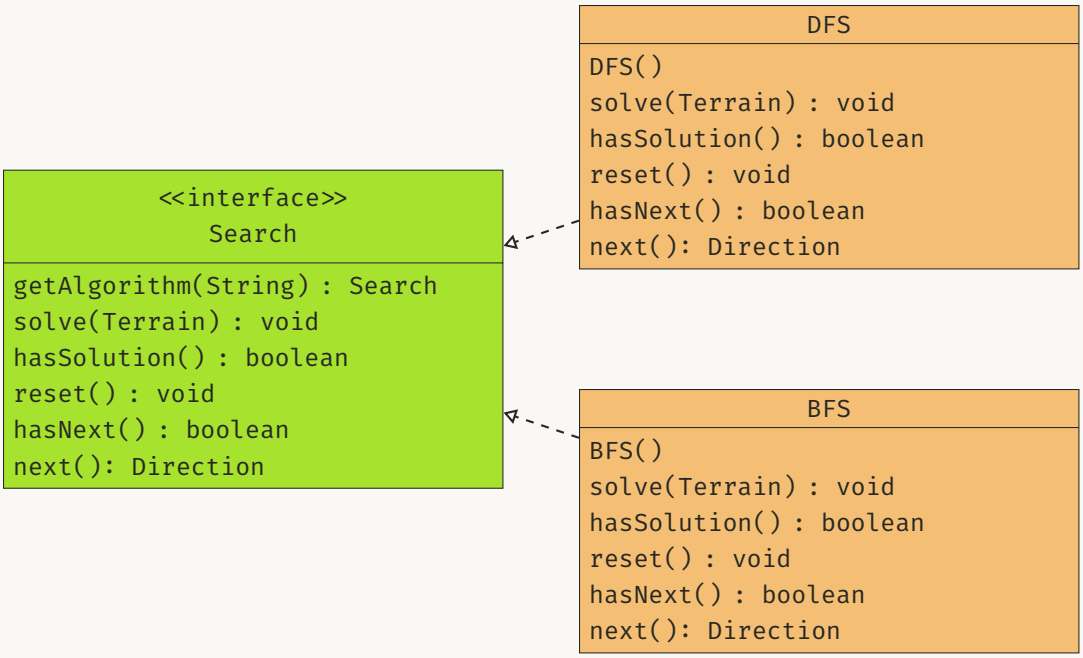
[1] `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`



Note that all getters and setters of the `Terrain` class are called with specific `Location`s in the terrain.

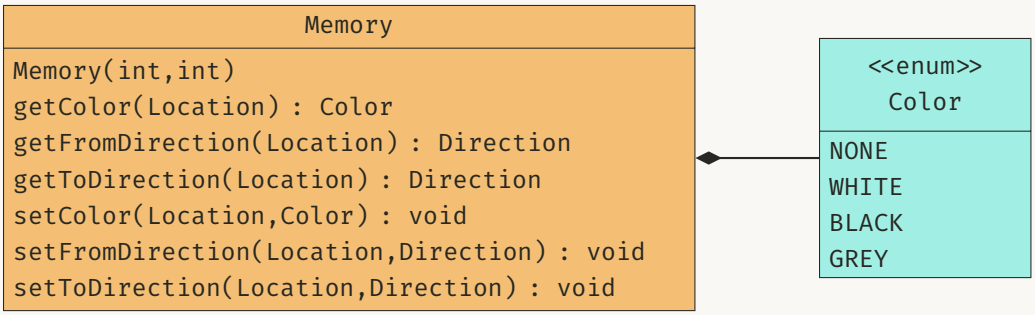
Package search

Each of the two algorithms will be implemented in their own class, `DFS` and `BFS`, that implements the interface `Search`. The `Search` defines methods to initiate the search algorithm and traverse the solution path.



The class `Memory` is used to store information during the search algorithm. It consists of "nodes" that store:

- A color (WHITE, GREY, BLACK).
- A direction "from" used to indicate how a node is discovered.
- A direction "to" to store the solution path.



Package `collections`

Data structures `Stack<T>` and `Queue<T>` from the course, along with associated exception classes and interfaces.

Package `animation` [optional]

Animate the solution path.

4 Requirements

Your program *must* meet the following requirements:

1. Your program should be clear and well commented. It must follow the "420-406 Style Guidelines" (on Léa).
2. Meet all pre-conditions of an operation *before* it is performed on a data structure. You can assume that the terrain files are well-formed.
3. Both `DFS` and `BFS` must implement the interface `Search`.
4. Don't use recursive methods in your solution.
5. Indicate which cells were seen (GREY in BFS) and which cells were visited (BLACK) during search. Demonstrate this by printing the `Memory` of your solution to STDOUT.
6. For each search algorithm, add it to the static method

   ```
   public static Search getAlgorithm(String algorithm) { ... }
   ```

   in `Search` as is shown in the starter `Search.java`. In your `Main`, update the constant to specify the search algorithm.
7. Submit using Git by following the instructions (on Léa).