# ROB 502 FA24: Homework 6

*See Canvas for due dates.*

**Rules**:

1. All HW must be done individually, but you are encouraged to post questions on Piazza and discuss during OHs (recall: do not email assignment questions to teaching staff!)
2. Late work adheres to the late grading policy (see syllabus, or use late-day tokens)
3. Submit your code on Autograder.io
4. Remember that copying-and-pasting code from other sources is not allowed

**Code Download**:
The code is available on the ROB 502 FA24 Git repo in the appropriate HW folder.

We prefer that you use the `‘fetch’/‘pull’` Git methods for the HWs, but we know that Git can be hard/confusing; if necessary, you can always just **download** the code from the repo link. However, learning to use Git will be to your benefit for multiple reasons:

1. You can get some easy extra credit if you properly pull and version-control (and comment) your work for the HW assignments!
2. ROB 550 will require you to use Git in a much more complex capacity, and without much learning time (large code databases, lots of teamwork with multiple branches, etc.).
3. If you ever need to setup a new VM or use your code on another device (new computer, VM crashes, etc.) you can easily restore your code progress with Git.
4. Many robotics-related industries and research labs rely on Git for robust version-controlling of their work… if mastered, you can put this on your resumé!

Open the directory `hw6` in VSCode. You can do this by `cd`-ing to the `hw6` directory and running `code .` . Do not run VSCode from subdirectories of `hw6` .

**Instructions**:

See previous homework assignments for notes about using Autograder.io, but you should be an expert now!

Each problem will give you a file with some template code, and you need to fill in the rest. Make sure to only put your code in the areas that start with `// --- Your code here` and end with `// ---`. **Do not edit code outside these blocks!**

Sample input and output are given as `input.txt` and `output.txt` respectively. Getting your program to successfully compile and correctly reproduce the sample output will get you some points, while the remaining points will be from hidden inputs that test some edge cases. Think about valid inputs that could break your logic. **Note that the templates given will read from `input.txt` and output to `output.txt`, so it might be a good idea to make a copy of the sample output to avoid overriding it.**

1. **Hill climbing** [22 points]:

   Start by opening the `hw6/climber` folder in VSCode. In this problem you will implement a variant of hill-climbing search. Imagine a mountain climber that is always trying to climb upward. The climber starts at a coordinate, considers the neighboring coordinates, and moves to the coordinates that is highest above the current coordinates (but not too high, more on this below). This process iterates until the climber is at a local maximum.

   You are given the starting coordinates (first line) and a grid of integers representing a height map in `input.txt`. The height map can be of any size but will always contains at least one coordinate. Heights will always be integers. Coordinates start at 0. An example input.txt is shown below:

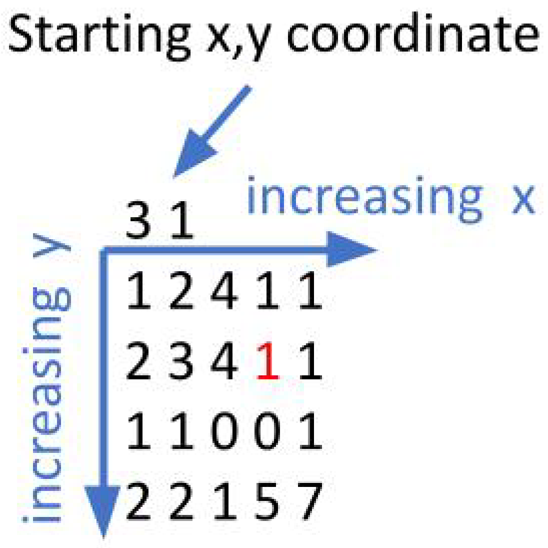


*Figure 1 - Example input.txt. The starting coordinate is show in red.*

   Your task is to implement the hill-climbing algorithm to find a local maximum when starting from the given starting coordinates. For hill-climbing you will need to generate the neighbors for the current coordinates. Let the current coordinates be $(x, y)$, then the neighbors should be:

   a. $(x + 1, y)$,
   b. $(x - 1, y)$,
   c. $(x, y + 1)$, and
   d. $(x, y - 1)$.

   Usually hill-climbing search will pick the highest-valued neighbor that is larger than the current height, but in this problem we will change how the neighbor is picked. The mountain climber will move to the **highest-valued neighbor** as long as that neighbor's height is **climbable**. To be climbable the neighbor's height must be 1) higher than the current height; and 2) no higher than the current height + 2.

For example, if the climber is currently at height 3 and the neighbors have heights 1, 6, 4, and 5, the climber will choose the neighbor with height 5. If two neighbors tie for the highest climbable height, the climber will pick among the highest climbable neighbors in the order of neighbors shown above (a-d). For example, let's say the neighbors $(x-1, y)$ and $(x, y-1)$ are tied for the highest climbable height, then we would pick the neighbor at $(x-1, y)$.

As you run your hill-climbing search, print each coordinate the climber visits until a local maximum is reach in `output.txt`. Each line in `output.txt` should be the coordinate followed by the height at that coordinate, e.g., `4 1 0` means the coordinate `x=4, y=1` with height `0`.

**You have three tasks in this problem:**

a. Fill out the `CMakeLists.txt` file and add the necessary `#include` statements to `climber.cpp`, `print_map.cpp`, and `run_climber.cpp`. You should compile a library called `climber` using `climber.cpp`, then you should compile two executables, `print_map` and `run_climber`, that link to the `climber` library. Once you've added the necessary `#include` statements and filed out the `CMakeLists.txt`, after this everything should compile without any other code changes.

   **NOTE**: In order for the autograder to work, you are given a "symlink" called `climber_CMakeLists.txt` which is essentially a "shortcut" to the real CMakeLists.txt. You should edit `CMakeLists.txt` but upload the symlink to the autograder.

b. Now you should implement the `printmap` function in `print_map.cpp`. Recall the ranged-for and auto syntax we learned for iterating over maps, covered in Lab 17. To run `printmap` in VSCode, use `ctrl+shift+p` then `Debug` . You can also use `Cmake: Set Debug Target` if you want to change which executable you're Debugging. When you run `printmap` it should print each entry in the map on a separate line using the format "{x},{y} val: {value}" where {x} means print the value of the variable x.

c. Finally, you should fill in the core loop of the algorithm in `climber.cpp`. This is the section of code where you decide where to move based on the current coordinates, current height, and the neighboring coordinates and heights. You are given one example `input.txt` and `output.txt`, but make sure you test your code with different inputs.

2. **Uninformed Maze Search (BFS, DFS)** [22 points]:

Start by opening the `hw6/maze` folder in VSCode. In this problem, you will implement uninformed search algorithms (BFS and DFS) to navigate a 2D maze. There are several files, some of which you should modify and others which you should use as given:

- `hw6/maze/maze_main.cpp` *don't modify* - reads input, calls your solution, then prints solution
- `hw6/maze/maze.h` *don't modify* - declares the problem class and base tree classes
- `hw6/maze/bfs.h` **modify** - declares the BFS class; you need to decide what data structure to use and define any helper functions
- `hw6/maze/dfs.h` **modify** - declares the DFS class; you need to decide what data structure to use and define any helper functions
- `hw6/maze/maze_impl.cpp` *don't modify* - implements some of the methods declared in `maze.h`, `dfs.h` and `bfs.h`
- `hw6/maze/maze_impl_student.cpp` **modify** - you need to implement the remaining methods that were not defined in `maze_impl.cpp`. This includes `ProblemDefinition::validStates` which returns the successors of a state, dependent on if you allow for diagonal traversal, `TreeSearch::extractPath` which extracts the actual start to goal path from a solved goal node into `path_` (solved using BFS or DFS), and `BFS::solve`, `DFS::solve`, `BFS::addNode`, and `DFS::addNode` (inside each `solve`, `extractPath` is called for you, so you need to find the goal node)
- `hw6/maze/CMakeLists.txt` **modify** - defines the project for CMake to link together (needed because the implementation is spread across multiple `.cpp` files)

The input is given in `input.txt` for example:

```
0 0 0
6 4
0 0 0 0 0 0 0
X 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 X 0
0 0 0 0 0 0 0
0 0 0 0 0 0 X
X 0 X 0 0 X X
0 0 X 0 0 X X
```

The first line is `start_x start_y allow_diagonal` where `allow_diagonal` controls whether the robot can take diagonal actions (which you will need to implement in `ProblemDefinition::validStates`; 1 being allowed diagonal, and 0 being not allowed). The second line is the x and y coordinates of the goal cell. The following lines define the maze (0 = free space; X = collision).

The output is written to `output.txt` for example:

```
BFS
0 0
1 0
... more lines of numbers ...
6 6
6 7

DFS
0 0
1 0
... more lines of numbers ...
6 6
6 7
```

Which illustrate the coordinates of the navigated state as we traverse from the start to goal state. Note that there may be multiple possible paths, any path that reaches the goal is acceptable.

**Complete this problem in the following steps:**

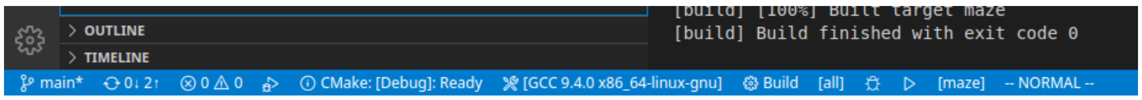
a. Read through the code we gave you to try to understand which functions have already been written, and what member variables you will need to read/write. This will make the rest of the problem much quicker and prevent you from accidentally re-implementing things we've already written. The ability to read through someone else's code and get a general understanding of what it does/should do is an essential skill! If you get confused, try making a diagram on paper that has every class (along with member functions) and the inheritance relationship between classes, then go through the code and see which file contains which function.
b. Fill out the `CMakeLists.txt` file to compile all the `.cpp` files listed above into one executable named `maze`. The code should compile as-is without you implementing anything yet.
c. Fill in `ProblemDefinition::validStates` to return the vector of possible states you can go to from a given state. You'll want to use the function `isStateValid` when implementing `validStates`.
d. Fill in `dfs.h` and `bfs.h` by adding the correct data-structure for the nodes. See the lecture slides for which data-structure to use.
e. Fill in the `solve` methods for both DFS and BFS in `hw6/maze/maze_impl_student.cpp`. This should include the main loop of the search algorithm, which is also described in the lecture slides on Tree Search. For this you will want to make use of functions like `addNode`, which you should implement, as well as `isGoal` and `validStates`
f. Implement `extractPath` , which should fill in the member variable `path_` with the sequence of states from the start to the given node. In the slides this step is referred to as "back tracking".

**To check your solutions and run the code:**

After making your modifications to the code as outlined above, the process of running the solver and then displaying the results will be as follows:

1. Create a `/build` folder within the `hw6/maze/` directory, and `cd` into that folder. Run CMake from there as `cmake ..` to create the makefile using the CMakeLists.txt file from the main directory.
2. "Make" (`make`) the MakeFile that you just created within the `/build` directory.
3. Copy any input text files for the mazes you'd like into the build directory, then run `./maze [input map name]` to solve that map with both BFS and DFS.
4. Run `python3 ../visualize_maze.py [input map name] output.txt` to visualize the solutions you just created.

*Or, if you would like, it can be convenient to setup CMake inside your IDE, as detailed in the setup guide. Follow that guide to configure and build the project inside the IDE; you should see a build button at the bottom bar. You can then run in debug mode by pressing the bug icon to the right of* `[all]` *in the same bar.*



```
[build] [100%] Built target maze
[build] Build finished with exit code 0
```

You can use `hw6/maze/visualize_maze.py` to visualize the maze and the output solutions; for example, on the sample input and output, with the BFS trajectory in blue, DFS trajectory in red, and obstacles represented as black patches:

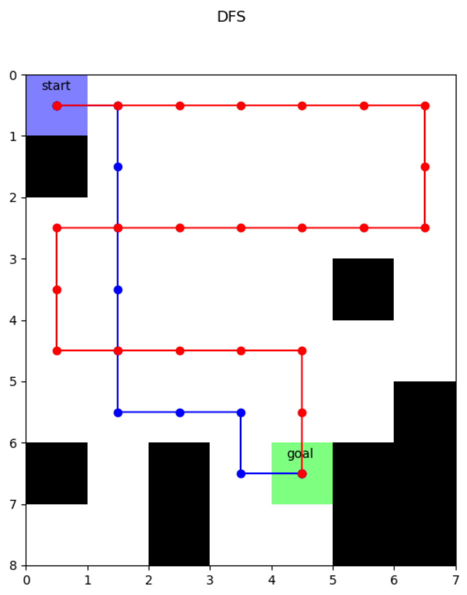


*Figure 2 - An example when diagonal movements are NOT allowed*

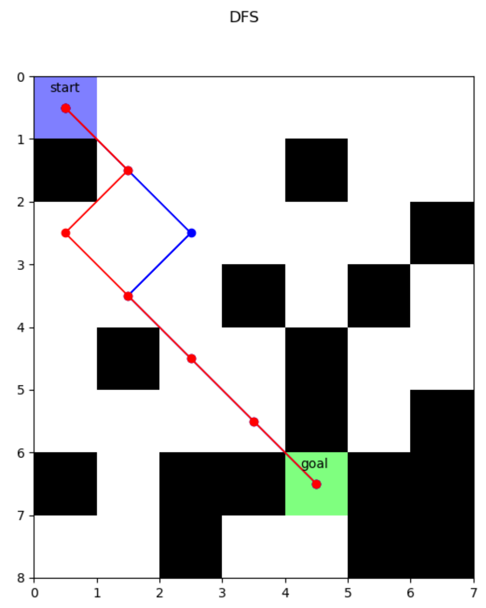


*Figure 3 - An example when diagonal movements are allowed*