// CS570 Artifical Intelligence

// Spring 2013

// Project 1a

// Heyan Huang

// Feb 1, 2013

Part I: Algorithms

For this small project, the breadth first search algorithm was used.

This project was implemented using C++ language, single linked list Queue and double linked list multi-branches tree (at most 4 branches for one tree node) data structures are used to enforce this bread first search algorithm and help record visited cells and the optimal path (if the optimal path exist). The steps of this project are described as listed:

read the data from a map file (map.txt), which includes the width and height of the map, the col column and row coordinate of the start point and the goal, togehter with the whole map, which is a two-dimensional character array composed of different character letters marking different restrictions (and different costs).

while reading the 2-d array data, to make the store and print array steps easier, a new 2-d static character array with 100x100 width \* height was created to store the original map array (with water restrictions), and help record visited map cells and print optimal path using this algorithm (if the optimal solution exists).

Searching details:

For data structure objectives, Queue's "Fist-in, First-out" policy enforce the breadth first search algorithm while stack's "Last-in, First-out" may works on depth first search when push in elements are ordered according to objective. Queue is not used to record the path, so single linked list works well already. But instead, tree was used to record/memorize all the "breadth" of all previous steps/paths, and I would need to use the tree to trace back from Goal opposite back to Start to visiualize the optimal path solutions, sofor tree I have to used double linked list.

For the searching, each of the left, right, up and down side of current cell (first node in the queue, initially started from the 'S' start position) was checked to see if it is the goal. If it is, then game is over. we find the goal and we can use a tree member function to find the path from goal trace back to the start point, and successfully print out the path, restriction water area togehter with marked visited cells.

If the check goal answer is no, we have more other conditions need to be checked, including if the cell is restriction water cell (from implementation point, I checked if the cell is within array boundary first before water), and if the cell is visited already (by checking the static array which stores this piece of information). If any of these answer is no, the cell is kicked outside consideration. When all checking conditions are satisfied, the cell is pushed into queue and insert the tree (left, right, up and down, checking order is not important, the important part is insert the cell into correct/corresponding linked list/position of the tree. for example, if this is left-side of currect cell, then current cell was inserted into tree parent's left pointer).

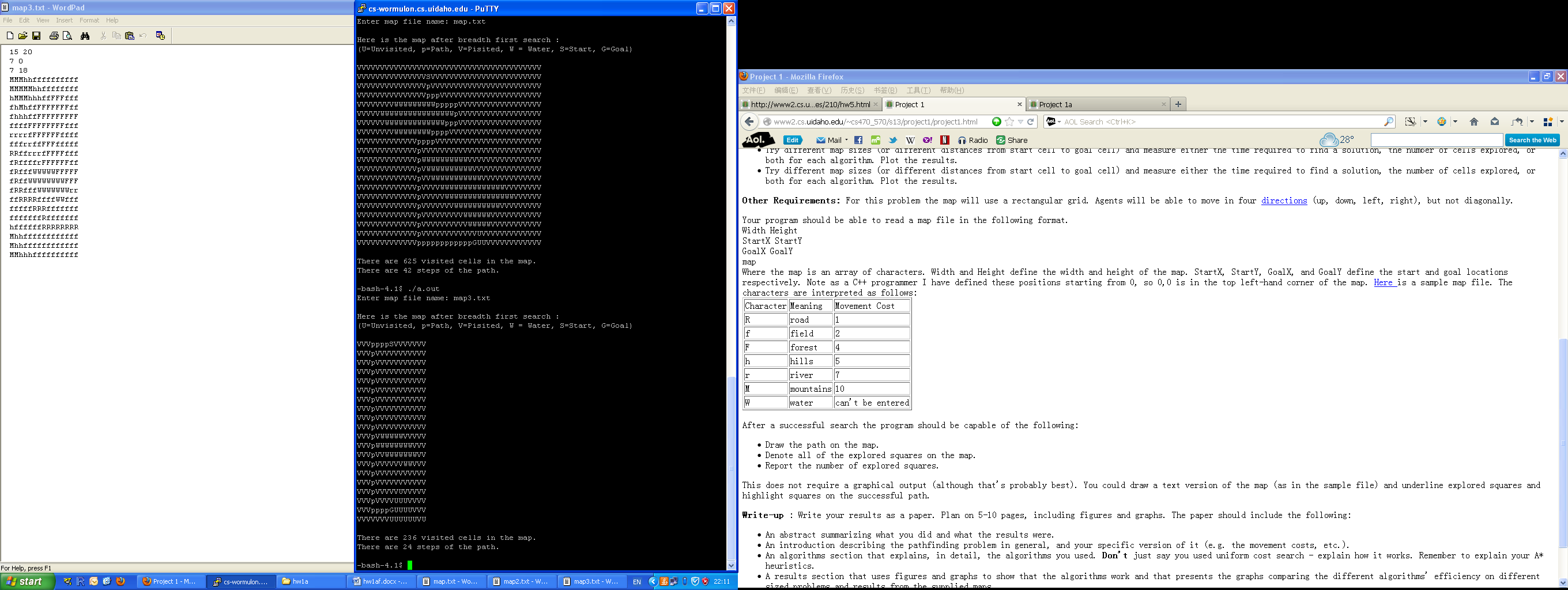
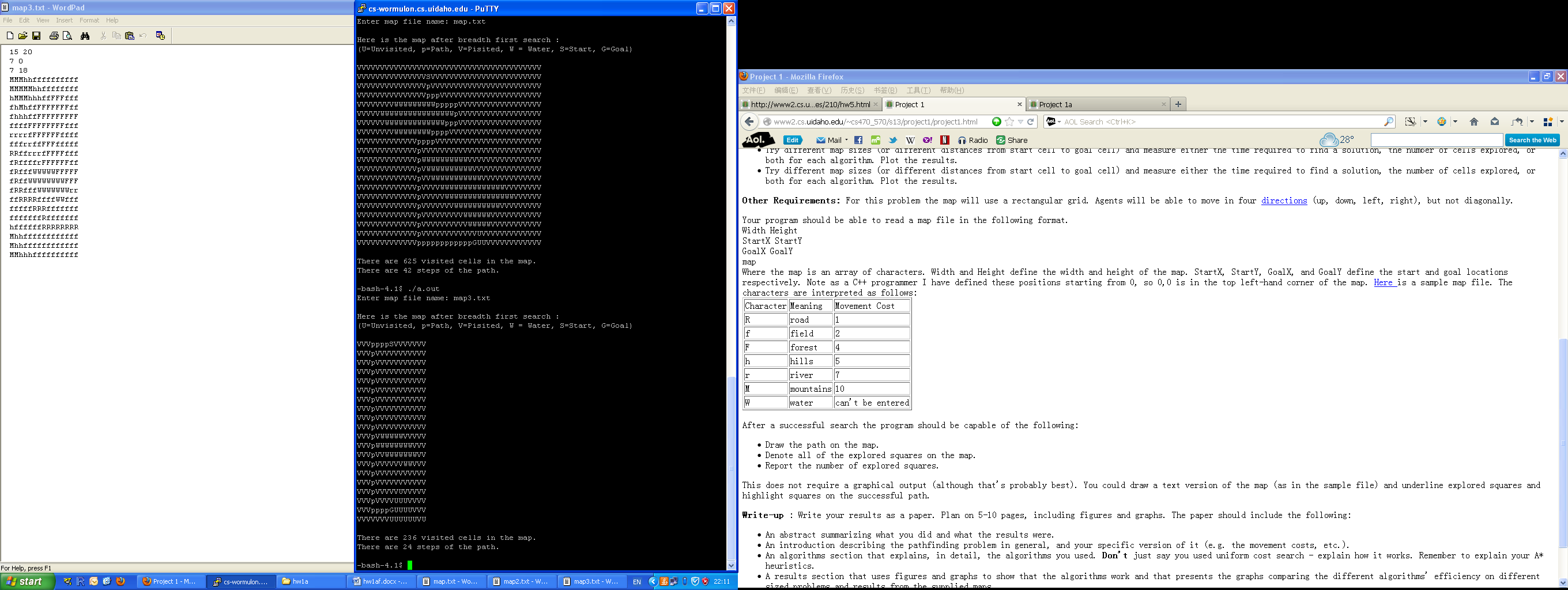
After one cycle, we need to checked if the queue is empty. the game would be over if queue is empty already. Otherwise, we dequeue and repeatedly start another cycle until we find the goal. Recursion was used here for repeating cycles. At the same time static array stores all the changed informaiton, for example, cell changed to be visited, or final step some visited changed to be path.

Using static storage 2d array and tree member functions, after search, the map status can be printed out including unvisited cells (U), visited cells (V), water (W) , path (p), start point(S) and the goal (G).

Part II: Results

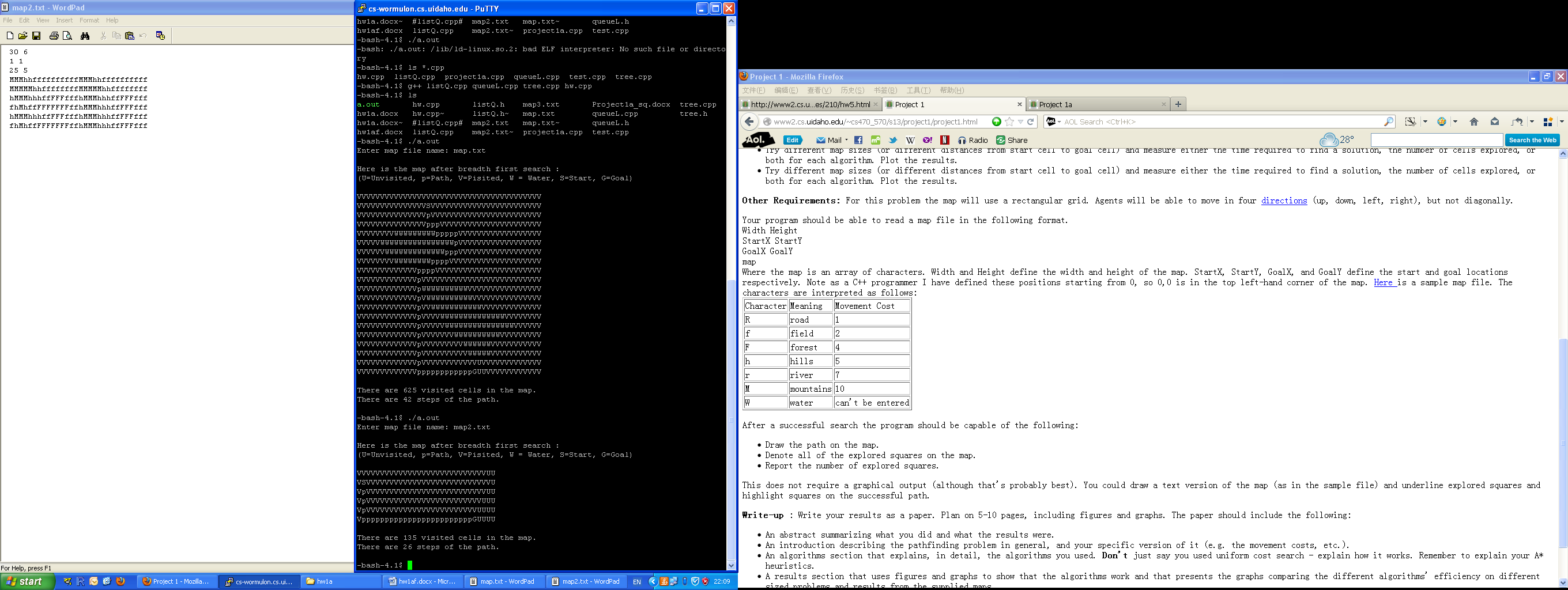
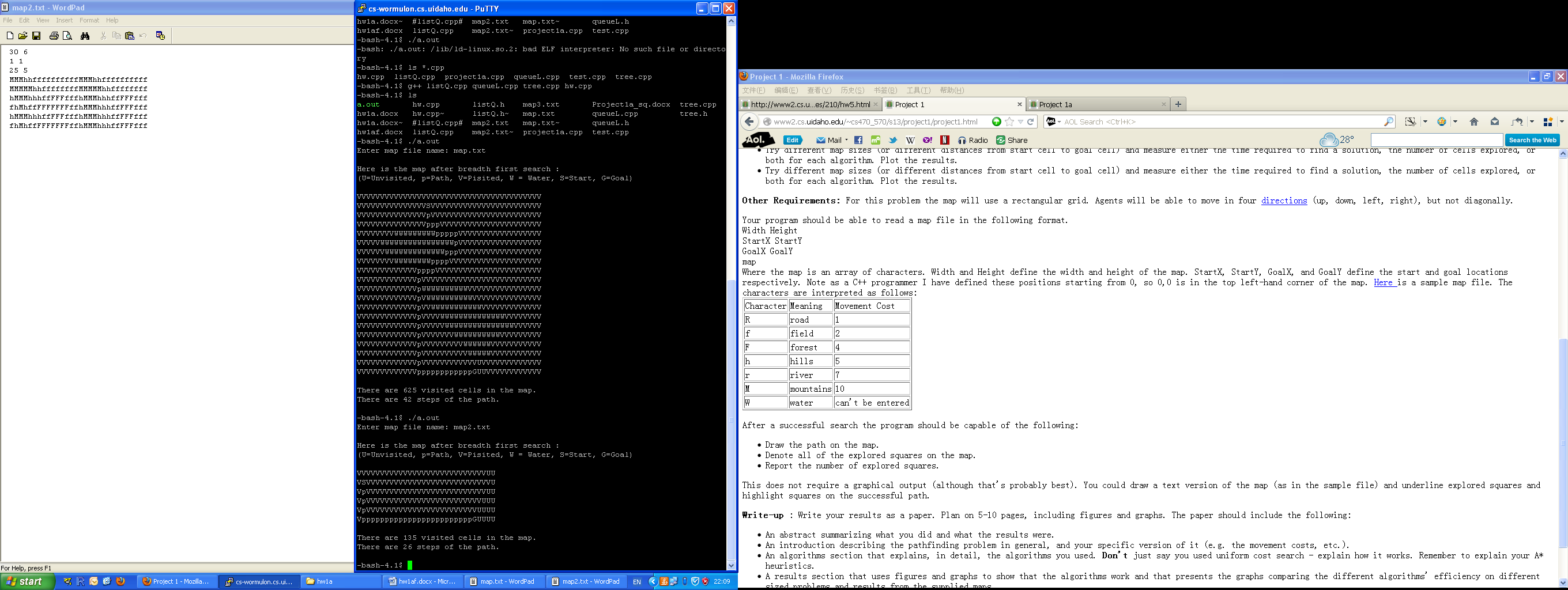
Breadth first search is able to find the optimal results(shortest path/steps) if such a path exists. But at the same time, it explores a large amount of cells.

I used Dr. Soule's sample map file first, followed are the original and final maps:

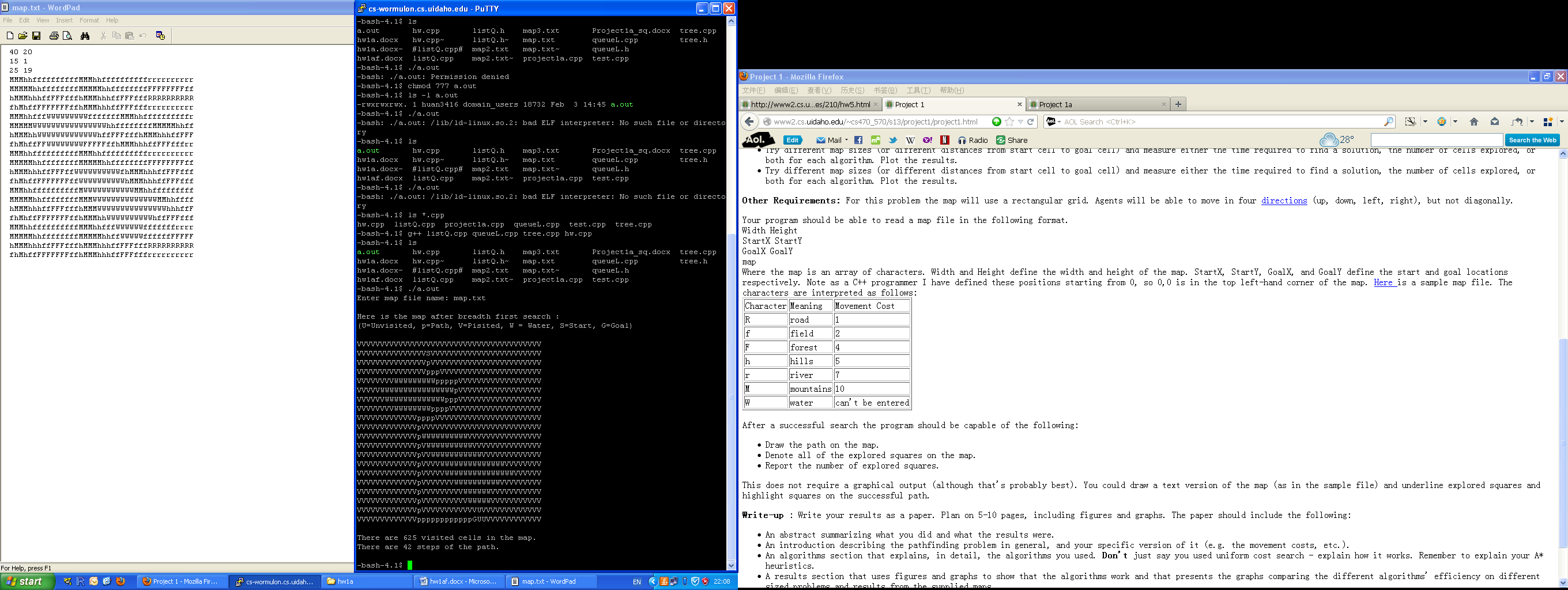
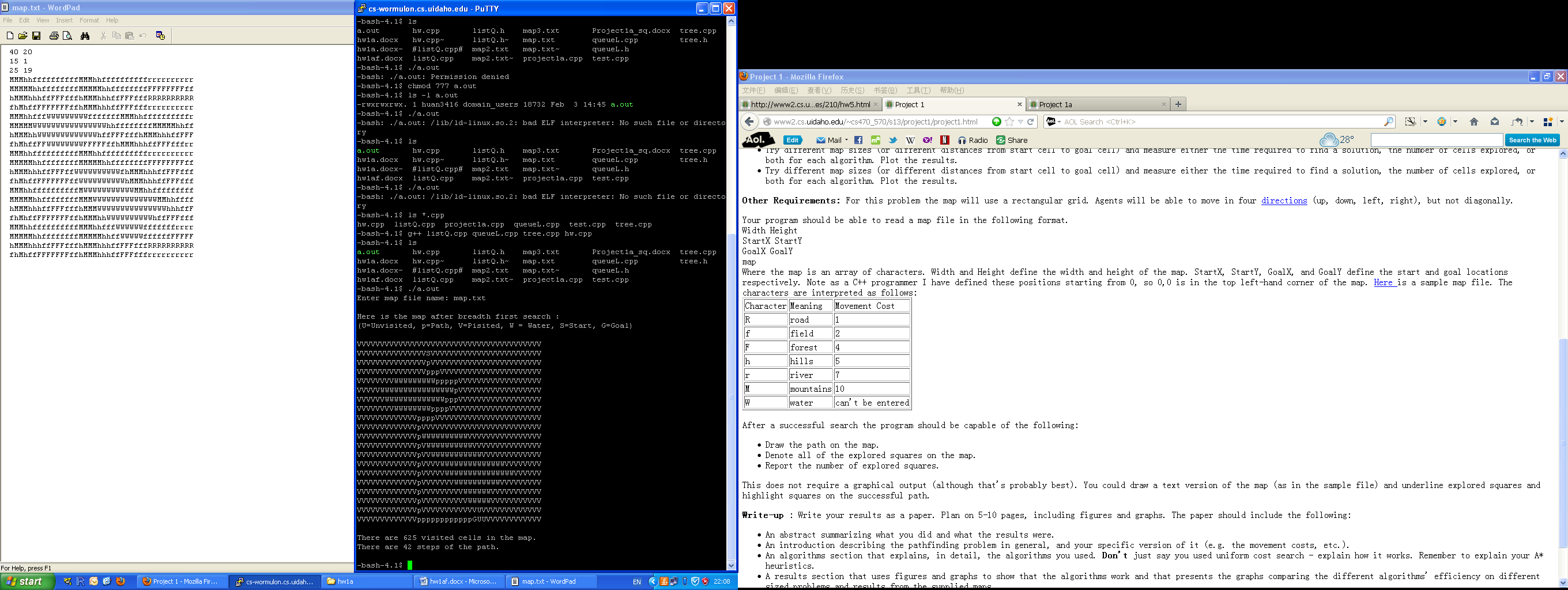
 

This is Dr. Soule listed sample map for us. The bread first search visited 236 cells and took 24 path steps from the start point to reach the goal.

236 24

This is a small map manually made by myself. The algorithm visited 135 cells and took 26 path steps from the start point to reach the goal.

This is the map I aimed to separate start and goal apart and in the middle blocked the path by too large pools of water. Tha algorithm visited 625 cells and took 42 path steps to reach the goal. And from the right map we can see the algorithm visited all possible cells to find the solution.

Part III: Discussion

If the start and the goal are both in the map, and the water restriction did n't block all the possible path, in other words, if there is a possible/potential path, the breadth first search algorithm can always find the solution, so it is complete.

Without considering the cost, only choose the minimum number of steps for the path as the goal, breadth first search algorithm can always find the shortest path, which proves that it always finds the optimal solution.

However, if we take the time and space complexity into consideration, this method has its own restrictions.

In this project, I created a tree to store all the existing information and potential changes of cells that are visited. The tree needs to remember not only the cells been visited, but also all the previous "paths". As the visited cells/steps increase, the tree expands and gets bigger and bigger, need very large memory space to store all these information.

As shown from the several different maps, breadth first search explores almost all of the area of the map if the start point and goal are separated far from each other, which means it also takes time to visit all those cells.