Title: Experimenting with Pathfinding Algorithms
Date: 06/16/2023
By: Ethan Corgatelli

1.0 Table of Contents

2.0 Abstract

This paper describes the five different search strategies for pathfinding that I tested. I used Rust for this project. I'm new to Rust, but I've been finding it to be very enjoyable to use.

- In the Methods section, I provide a brief description of how my program works.
- The Results section contains figures generated by each pathfinding algorithm.
- In the Conclusions section I discuss the results, comparing each algorithm's pathfinding choices.
- In the Appendix I have included one full output of an algorithm running, which shows the map at every iteration of the search.

My code and full results can be found here: github.com/ETBCOR/cs470/tree/master/proj1

3.0 Methods

This section contains a brief high level overview of how my code works. Each subheading contains details relevant to each of the five algorithms.

To start this project I wrote the ***Terrain*** enum which takes one of the following values: *Road*, *Field*, *Forest*, *Hills*, *River*, *Mountains*, or *Water*. I wrote two functions in *Terrain*'s implementation block: *from()* which takes a character as input and returns an *Option<Terrain>* (which is either *Some(Terrain)* or *None* in the case that the character provided doesn't represent a *Terrain*), and *cost()* which is an associated function[1] that returns an integer representing the cost of moving into the *Terrain* it was called on.

There's an enum ***Status*** which takes one of the following values: *Untraversed*, *Path*, *Up(bool)*, *Down(bool)*, *Left(bool)*, or *Right(bool)*. Each algorithm uses a 2D array of these *Status*es to store the current state of exploration on the map. The directional variants represent where that tile came from during exploration, and the boolean value stored represents whether that tile is in the open set or not (this is useful for visualizing the results). *Status* has one associated function called *deactivate()* which changes the value to *false* for the directional variants and does nothing to the other variants.

There's a struct ***Visit*** with three fields: *step* (an integer), *loc* (a Vec2; two integers), and *cost* (an integer). Each algorithm uses some kind of queue of these *Visit*s to keep track of what positions on the map will be explored next.

Finally, there's a struct ***Map*** which has the following fields: 1) ***map***: a 2D map array, 2) ***costs***: a 2D cost (integers) array, 3) ***display_costs***: a boolean flag dictating whether or not to display the costs while printing the map, 4) ***dim***: the dimensions of the map, 5) ***start***: the start position, and 6) ***goal***: the goal position. There are many functions related to *Map*, as it was the main data structure I used during this project. I'll leave out details on all of them, but here's an overview of some of them:

- ***from_file_path()***: Takes a file path (as string) as input and returns a *Map* object constructed from the data in that file (crashes if file parsing fails).
- ***map_text()***: An associated function that returns a string representing the current state of the map.
- ***go_up() / go_down() /…*** : Associated functions that take a Vec2 representing a location to travel from that all return *Option<Vec2>* (which is either *Some(Vec2)* if the resulting position is both valid and *Untraversed*, or *None* if not).
- ***backtrack()***: Takes a reference to a file as input (so that the results of backtracking can be recorded) and returns a tuple of integers representing the distance and cost of the path that was traced while backtracking from the goal position to the start position.

The functions for each algorithm all start with a few variable declarations in common: ***f*** (a *File* that is created and will be written to), ***done*** (a boolean that keeps track of whether a valid path has been found), ***map*** (a *Map* object which is a deep clone of the map object referenced by the function's argument), ***q*** (some implementation of a queue that holds *Visit* objects), ***start*** & ***goal*** (*Vec2*s), and ***pops*** (an integer that counts the number of iterations).

[1] In Rust an associated function for some type is a function whose first parameter is a reference to an object of that type, and is called like so: *object.function()*.

3.1 Breadth First Algorithm

The ***breadth_first()*** algorithm is the simplest of all the algorithms I tested, and it's where I started this project. Its queue implementation is Rust's *VecDeque* ("A double-ended queue implemented with a growable ring buffer" – Rust docs). It does not use the *costs* field of the *Map* reference it was passed.

During initialization it marks the start position on the map as *Path*, then pushes it onto the queue.

In each iteration it first pops the front *Visit* off the queue. Then it deactivates the location stored in the *Visit* in *map* (removing it from the "open set"). Then for each valid neighbor, the neighbor's direction is updated (it came from the current cell), and its location is checked against the goal location. If the goal has been reached, *done* is set to true, the queue is emptied (deactivating any currently "open" spots), and the main loop is broken.

Finally, outside of the loop, if *done* is true, *backtrack()* is called, otherwise an error message is printed (this part is the same for all the algorithms and won't be mentioned again).

3.2 Lowest Cost Path Algorithm

This and the subsequent descriptions will be relative to the description above, as all the algorithms work in quite a similar way.

The ***lowest_cost_path()*** algorithm uses a *PriorityQueue* (from this crate) for its queue. To push items to a *PriorityQueue*, an item and a priority (an integer) must be provided. Popping items off the queue always returns the item in the queue with the highest priority.

This algorithm initializes all positions of the *costs* 2D array to the *MAX* possible value of the integer type, then sets *display_costs* to true, then sets the start position to *Path* on the map, just like in breadth first initialization. Then it also updates the cost of the start position based on the appropriate *Terrain* from the map. Finally, it pushes the start position to the queue with a priority of 0 (because it will be the only item in the queue during the first iteration, it will be popped first regardless of priority).

In each iteration it pops the highest priority *Visit* off the queue. It first deactivates the current position. Then for each valid neighbor, if the the cost of the current cell plus the cost of traveling into the neighbor cell (the ***maybe_cost***) is less than the neighbor's currently recorded cost, then the neighboring cell's cost is set to *maybe_cost*, it's arrow is set to point at the current cell, and it is added to the queue with a priority of ***MAX — maybe_cost*** (so that cells with a higher cost get a lower priority and evaluated later). This algorithm is guaranteed to find a path with the shortest possible distance if any exist.

3.3 Greedy Best First Algorithm

Algorithms 3-5 have the same initialization process as algorithm 2.

The ***greedy_best_first()*** algorithm explores all unvisited neighbors at each step, updating their cost and direction.

While adding neighbors to the queue it simply uses ***MAX* — [taxi-cab dist from goal]** as the priority heuristic. It essentially makes a sprint for the goal.

3.4 A* Algorithm (Taxicab Heuristic)

The ***a_star_taxicab()*** algorithm (like in the *lowest_cost_path()* algorithm) only updates neighbors and pushes them to the queue when it would have been better to come from the current cell. However it uses ***MAX* — (*maybe_cost* + [taxi-cab dist from goal])** as the priority heuristic, which is essentially the heuristics from algorithms 2 and 3 combined.

3.5 A* Algorithm (Euclidean Heuristic)

The ***a_star_euclidean()*** algorithm is just like *a_star_taxicab()*, but instead uses ***MAX* — *(maybe_cost* + [Euclidean dist from goal])** as the priority heuristic.

4.0 Results

Explanation: the letter on the left side of each cell represents the terrain variant at that location on the map. The start and goal positions have an “S” / “G” printed on the right side of their cells. Cells with an arrow have been explored; the arrow points to the adjacent cell it came from. Full block characters represent the final path (but they were once arrows).

4.1 Breadth First Results

M→	M→	M→	h→	h→	f→	f→	f S	f←	f←	f←	f←	f←	f←	f←
M→	M→	M→	M→	M→	h→	h→	f	f←	f←	f←	f←	f←	f←	f←
h→	M→	M→	M→	h→	h→	h→	f	f←	F←	F←	F←	f←	f←	f←
f→	h→	M→	h→	f→	f→	F→	F	F←	F←	F←	F←	F←	f←	f←
f→	h→	h→	h→	f→	f→	F→	F	F←	F←	F←	F←	F←	F←	F←
f→	f→	f→	f→	F→	F→	F→	F	F←	F←	F←	F←	f←	f←	f←
r→	r→	r→	r→	f→	F→	F→	F	F←	F←	F←	f←	f←	f←	f←
f→	f→	f→	r→	r→	f→	f→	F	F←	F←	f←	f←	f←	f←	f←
R→	R→	f→	f→	r→	r→	r→	f	F←	F←	F←	F←	f←	f←	f←
f→	R→	f→	f→	f	f	r	F	F←	F←	F←	F←	F←	f←	f←
f→	R→	f→	f	f	W	W	W	W	W	F↑	F←	F←	F←	F←
f→	R→	f→	f	W	W	W	W	W	W	W	W	F↑	F←	F←
f→	R→	R→	f	f←	f←	W	W	W	W	W	W	W	r↑	r←
f→	f→	R→	R	R←	R←	f←	f←	f←	f←	W	W	f→	f↑	f←
f→	f→	f→	f	f←	R←	R←	R←	f←	f←	f→	f→	f→	f↑	f←
f→	f→	f→	f	f←	f←	f←	R←	f←	f←	f→	f→	f→	f↑	f←
h→	f→	f→	f	f←	f←	f←	R←	R←	R	R→	R→	R→	R↑	R←
M→	h→	h→	f	f←	f←	f←	f←	f	f	f	f→	f→	f↑	f←
M→	h→	h→	f	f	f	f	f G	f	f	f	f	f→	f↑	f←
M→	M→	h→	h↑	h←	f←	f←	f	f	f	f	f	f	f↑	f

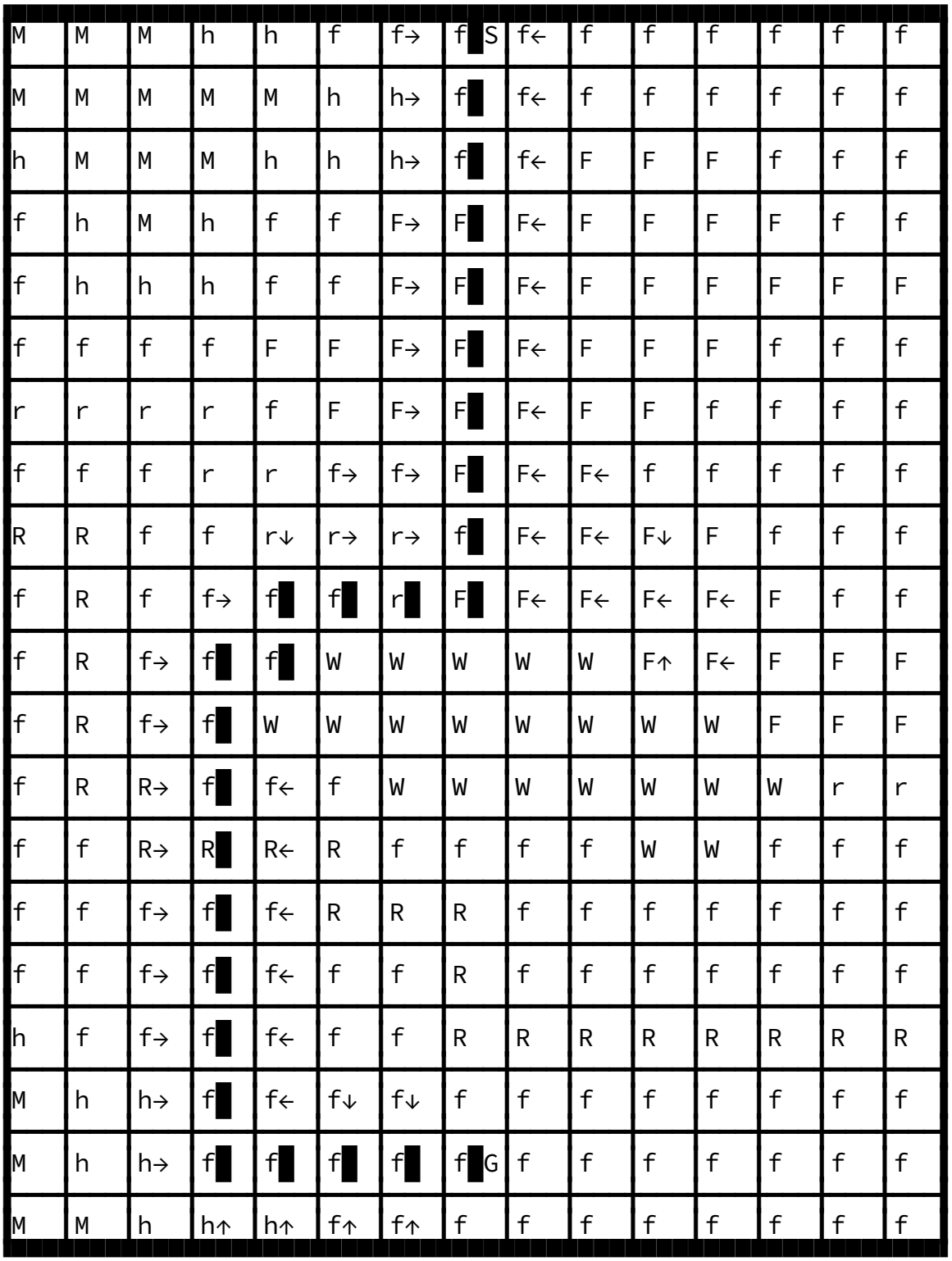
Path found (dist: 27 cost: 70 iterations: 263) by breadth first alg

4.2 Lowest Cost Path Results

M→	M→	M→	h→	h→	f→	f→	f S	f←	f←	f←	f←	f←	f←	f←
M↓	M→	M↑	M↑	M↑	h↑	h→	f	f←	f↑	f↑	f←	f↑	f←	f←
h↓	M↓	M→	M→	h→	h→	h→	f	f↑	F↑	F↑	F↑	f↑	f↑	f↑
f↓	h→	M→	h→	f	f	F	F	F↑	F←	F↑	F↑	F↑	f↑	f↑
f↓	h→	h→	h→	f	f↑	F↑	F↑	F↑	F←	F↑	F↑	F↑	F↑	F↑
f→	f→	f	f	F	F↑	F↑	F↑	F↑	F↑	F↑	F↑	f↑	f↑	f←
r↑	r↑	r	r↑	f↑	F↑	F→	F↑	F↑	F↑	F↑	f↑	f↑	f↑	f↑
f→	f↑	f	r↑	r↑	f↑	f→	F↑	F↑	F←	f↑	f↑	f↑	f↑	f←
R→	R	f	f←	r↑	r↑	r→	f↑	F←	F↑	F↑	F↑	f↑	f↑	f←
f↑	R	f↑	f→	f→	f↑	r→	F↑	F↑	F←	F↑	F→	F↑	f↑	f←
f→	R	f↑	f→	f↑	W	W	W	W	W	F↑	F→	F↑	F↑	F↑
f→	R	f↑	f↑	W	W	W	W	W	W	W	W	F↑	F↑	F↑
f→	R	R	f↑	f←	f←	W	W	W	W	W	W	W	r↑	r↑
f→	f↑	R	R	R	R	f←	f←	f←	f←	W	W	f→	f↑	f↑
f↑	f↑	f↑	f↑	f↑	R	R	R	f←	f←	f←	f→	f↑	f↑	f←
f↑	f↑	f↑	f↑	f↑	f↑	f↑	R	f←	f←	f	f	f↑	f↑	f↑
h↑	f↑	f↑	f↑	f↑	f↑	f↑	R	R←	R←	R←	R	R→	R↑	R←
M	h↑	h↑	f↑	f↑	f↑	f↑	f	f↑	f↑	f	f	f	f↑	f
M	h	h	f↑	f↑	f↑	f	f G	f	f	f	f	f	f	f
M	M	h	h	h	f	f	f	f	f	f	f	f	f	f

Path found (dist: 31 cost: 58 iterations: 244) by lowest cost alg

4.3 Greedy Best First Results



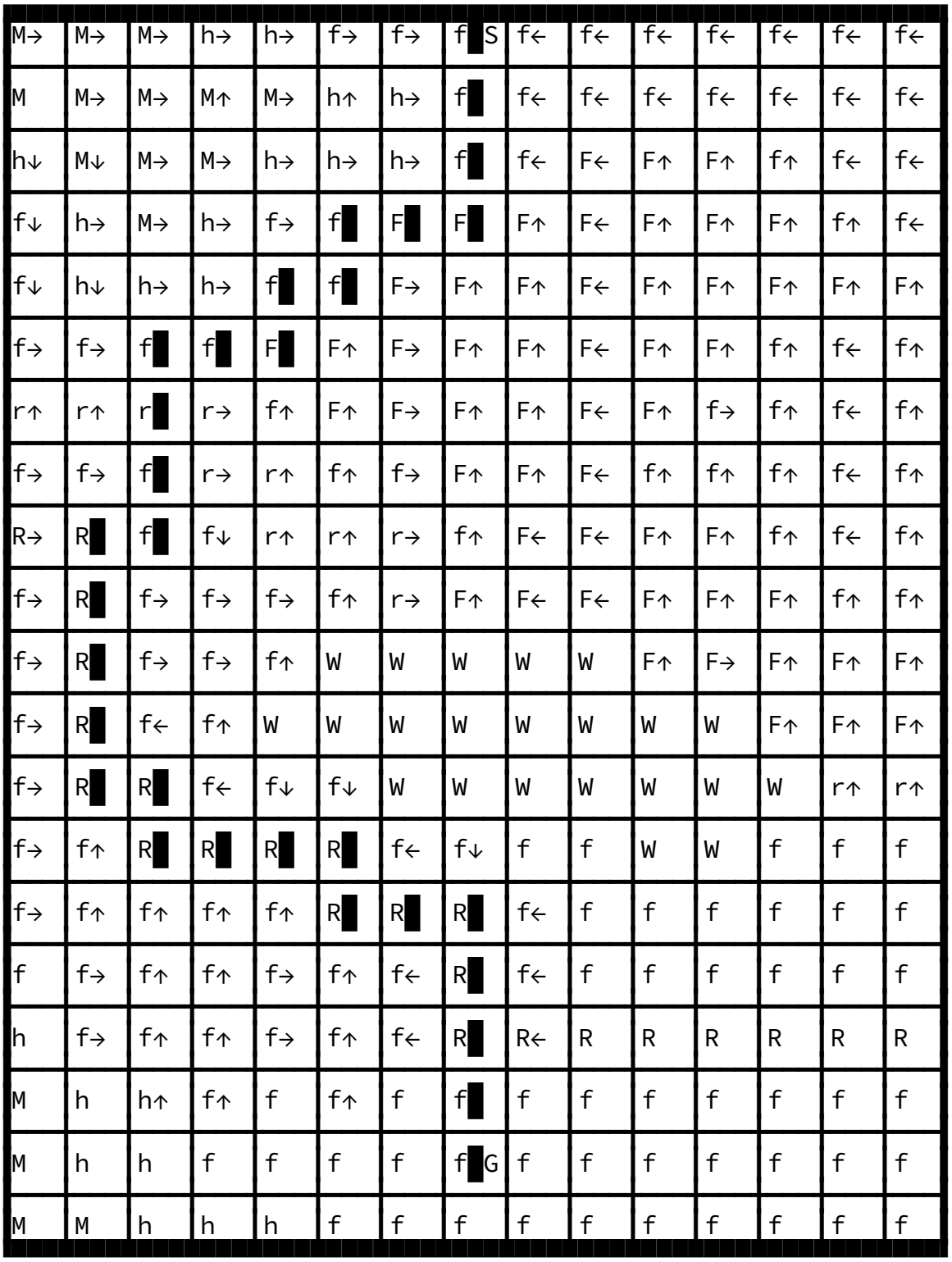
Path found (dist: 27 cost: 70 iterations: 79) by greedy best first alg

4.4 A* (Taxicab Heuristic) Results

M	M→	M→	h→	h→	f→	f→	f S	f←	f←	f←	f←	f←	f←	f←
M	M	M→	M↑	M→	h↑	h→	f	f←	f←	f←	f←	f←	f←	f←
h	M	M→	M→	h→	h→	h→	f	f←	F←	F←	F↑	f←	f↑	f←
f↓	h→	M→	h→	f	f	F	F	F←	F↑	F←	F↑	F↑	f↑	f←
f↓	h↓	h→	h→	f	f→	F→	F↑	F←	F←	F←	F↑	F←	F↑	F←
f→	f→	f	f	F	F↑	F→	F↑	F←	F←	F←	F↑	f→	f↑	f←
r↑	r↑	r	r→	f↑	F→	F→	F↑	F←	F←	F←	f↑	f←	f↑	f←
f→	f→	f	r→	r↑	f→	f→	F↑	F←	F←	f←	f↑	f←	f↑	f←
R→	R→	f	f→	r↑	r↑	r→	f↑	F←	F←	F←	F↑	f→	f↑	f←
f→	R→	f	f→	f→	f↑	r→	F↑	F←	F←	F←	F↑	F→	f↑	f←
f	R→	f	f→	f↑	W	W	W	W	W	F→	F↑	F→	F↑	F←
f	R→	f	f↑	W	W	W	W	W	W	W	W	F→	F↑	F←
f	R→	R	f←	f↓	f↓	W	W	W	W	W	W	W	r↑	r
f	f→	R	R	R	R	f←	f↓	f	f	W	W	f	f	f
f	f	f↑	f↑	f↑	R	R	R	f←	f	f	f	f	f	f
f	f	f	f↑	f↑	f↑	f↑	R	f←	f	f	f	f	f	f
h	f	f	f	f	f	f→	R	R←	R	R	R	R	R	R
M	h	h	f	f	f	f	f	f	f	f	f	f	f	f
M	h	h	f	f	f	f	f G	f	f	f	f	f	f	f
M	M	h	h	h	f	f	f	f	f	f	f	f	f	f

Path found (dist: 29 cost: 59 iterations: 191) by A* (taxicab) alg

4.5 A* (Euclidean Heuristic) Results



Path found (dist: 31 cost: 58 iterations: 212) by A* (euclid) alg

5.0 Conclusions

I'm happy to say that after enough bug fixing, all of the algorithms I tested seem to be working as expected. Here are the final statistics, with optimal values highlighted in green:

Algorithm	Distance	Cost	Iterations
Breadth First	27	70	263
Lowest Cost Path	31	58	244
Greedy Best First	27	70	79
A* (taxicab)	29	59	191
A* (euclid)	31	58	212

Note that distances and costs were calculated inclusively (counting the start / goal positions).

As we can see in the table above, both the breadth first search and greedy best first search get to the goal using the shortest possible path of 27 (going around the East side of the lake is a minimum of 28), as expected.

Both the lowest cost path search and the Euclidean A* search found the path with the lowest cost, taking full advantage of the *Road* cells. Interestingly, the taxicab A* search takes a small detour off the *Road* through some *Field*, opting to stay closer to getting to the goal despite the slightly higher cost of that path, whereas the Euclidean A* search is more accurate, which matches my expectation.

The greedy best first search found a path in far fewer iterations than the other algorithms, but I'm not fully confident that that would *always* be the case for every possible map[2], so I didn't shade it green.

[2] I can imagine trying to design a map for which greedy best first is not the optimal search algorithm in regard to number of iterations, but I imagine it would be difficult. Even after having failed to prove by example that greedy best first search isn't iterations-optimal, I'd hesitate to claim that it's optimal without some kind of proof – which is obviously beyond the scope of this assignment.

Appendix: More Results

This section contains the output of running one of the pathfinding algorithms (A* Euclidean) on a smaller map I created for testing purposes. The output is in columns read left to right. The final printout of the board is the one used in each figure in the Results section. All the other printouts, however, show the costs of explored cells throughout the progression of the algorithm. Big arrows represent the open set. Small arrows represent the closed set.

```
Running A* search (heuristic:
euclidean dist)
```

04S	F	M	F	F G
f	f	W	F	F
f	f	f	f	f

04S	08◂	M	F	F G
06▴	f	W	F	F
f	f	f	f	f

04S	08◂	M	F	F G
06↑	08◂	W	F	F
08▴	f	f	f	f

04S	08←	18◂	F	F G
06↑	08◂	W	F	F
08▴	f	f	f	f

04S	08←	18◂	F	F G
06↑	08←	W	F	F
08▴	10▴	f	f	f

04S	08←	18◂	F	F G
06↑	08←	W	F	F
08↑	10▴	f	f	f

04S	08←	18◂	F	F G
06↑	08←	W	F	F
08↑	10↑	12◂	f	f

04S	08←	18◂	F	F G
06↑	08←	W	F	F
08↑	10↑	12←	14◂	f

04S	08←	18◂	F	F G
06↑	08←	W	18▾	F
08↑	10↑	12←	14←	16◂

04S	08←	18◂	F	F G
06↑	08←	W	18▾	20▾
08↑	10↑	12←	14←	16←

04S	08←	18◂	22▾	F G
06↑	08←	W	18↓	20▾
08↑	10↑	12←	14←	16←

04S	08←	18←	22▾	F G
06↑	08←	W	18↓	20▾
08↑	10↑	12←	14←	16←

04S	08←	18←	22↓	24↓
06↑	08←	W	18↓	20↓
08↑	10↑	12←	14←	16←

```
Doing backtracking
```

04S	08←	18←	22↓	24
06↑	08←	W	18↓	20↓
08↑	10↑	12←	14←	16←

04S	08←	18←	22↓	24
06↑	08←	W	18↓	20
08↑	10↑	12←	14←	16←

04S	08←	18←	22↓	24
06↑	08←	W	18↓	20
08↑	10↑	12←	14	16

04S	08←	18←	22↓	24
06↑	08←	W	18↓	20
08↑	10↑	12	14	16

04S	08←	18←	22↓	24
06↑	08←	W	18↓	20
08↑	10	12	14	16

04S	08←	18←	22↓	24
06↑	08	W	18↓	20
08↑	10	12	14	16

04S	08←	18←	22↓	24
06	08	W	18↓	20
08↑	10	12	14	16

S	F←	M←	F↓	F G
f	f	W	F↓	F
f↑	f	f	f	f

```
Path found (dist: 9 cost: 24
iterations: 14) by A* (euclid)
alg
```