Navigation and AI

Introduction

In this project I created a pathfinder that uses the A* algorithm and it is implemented in Java. My focus have been to create a fast A* pathfinding algorithm. The main optimization that can be done for an A* algorithm is to use a fast priority queue for the open set. I have also implemented a cool map generator to verify that the algorithm always finds a path, at least if there is a solution. The pathfinder searches in a two-dimensional grid world where enemies are unable to walk diagonally. The enemies (in blue) tries to reach the hero (in red) which moves randomly around while ignoring the rules in the map. Enemies gets stuck when they enter a swamp (a dark green tile) for several turns and they cannot cross a wall. They are of course aware of the cost of landing on swamp and tries to avoid them unless it pays off in the end.

Map generation

In this section I will present and explain the map generator. The map is a large two-dimensional array. The method `generateMap(int width, int height)` which body is presented below generates a map of the specified size. In the Game class I can set the size of each tile when it is drawn to the screen. This gives me the possibility to test maps that have a number of pixels equal to the screen size. I have a two percentage variables that determines the amount of walls and swamp. It is possible that the hero or an enemy becomes blocked, but I have tried to clear out the area around afterwards.

```
int value = 0;
// Border?
if (y == 0 || y == height - 1 || x == 0 || x == width - 1){
	value = 1;
} else {
	Random random = new Random();
	// Wall?
	if(random.nextInt(100) > wallPercentage){
		value = 1;
	// Swamp?
	} else if(random.nextInt(100) > swampPercentage){
		value = 2;
	}
}
arr[y][x] = value;
```

The body of the loop in the generateMap method that assigns tile values to each tile in the map randomly.

A* and priority queue

The criteria for a fast priority queue is good run times, $O(\log_n)$ or better, for the poll, add, remove and contains operations. I have looked at three contestants from the Java API.

Collection	Poll*	Add	Remove	Contains
PriorityQueue[1]	$O(\log_n)$	$O(\log_n)$	$O(\log_n)$	O(n)
HashSet[2]	O(n)	O(1)	O(1)	O(1)
TreeSet[3]	O(1)	$O(\log_n)$	$O(\log_n)$	$O(\log_n)$

* or similar operation(s)

As i can be seen in the table there is no single solution that outperforms the others on all operations, but the TreeSet does meet our criteria. I have, however, combined a HashSet and a TreeSet to gain a better performance but on the cost of memory.
I have created a class called HashTreeSet that implements the Set interface. It consists of both a HashSet and a TreeSet and stores the elements in both sets. The implemented methods of the Set interface are implemented so it only calls the fastest of its sets, unless it is an add or remove call then it needs to update both sets. My HashTreeSet has the following running times.

Collection	Poll	Add	Remove	Contains
HashTreeSet	O(1)	$O(\log_n)$	$O(\log_n)$	O(1)

Benchmarks

I have run my A* algorithm up against the PriorityQueue to see if my HashTreeSet really beats it on performance. The time measured are average running times in the first 30 seconds of the test.

Map size	HashTreeSet	PriorityQueue
512x720	~720 ms	~6052 ms
128x128	~15 ms	~45 ms

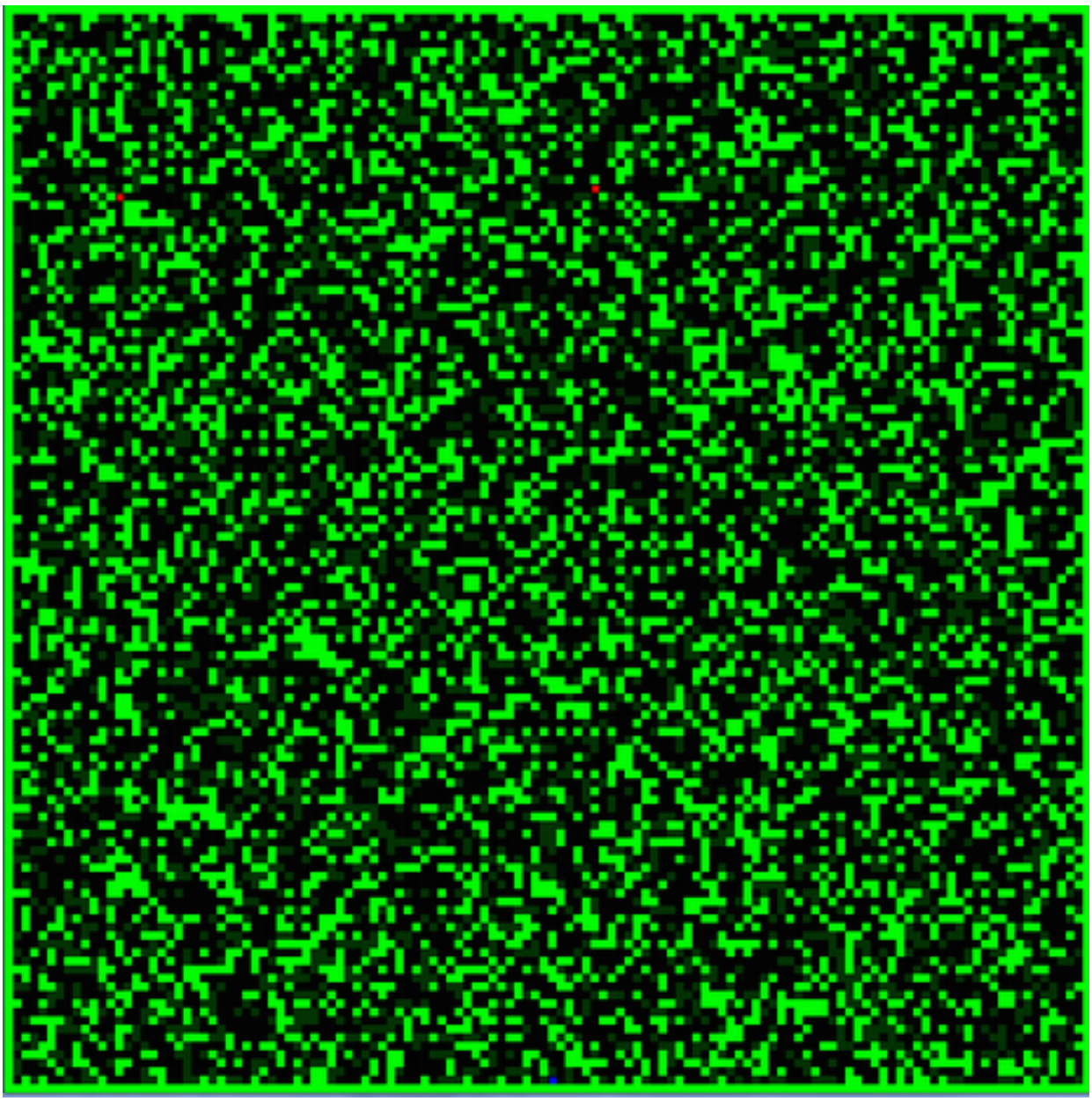
And it does significantly!

---

[1] http://docs.oracle.com/javase/1.5.0/docs/api/java/util/PriorityQueue.html

[2] http://docs.oracle.com/javase/1.4.2/docs/api/java/util/HashSet.html

[3] http://docs.oracle.com/javase/1.4.2/docs/api/java/util/TreeSet.html



A screenshot of the test with 128x128 tiles.