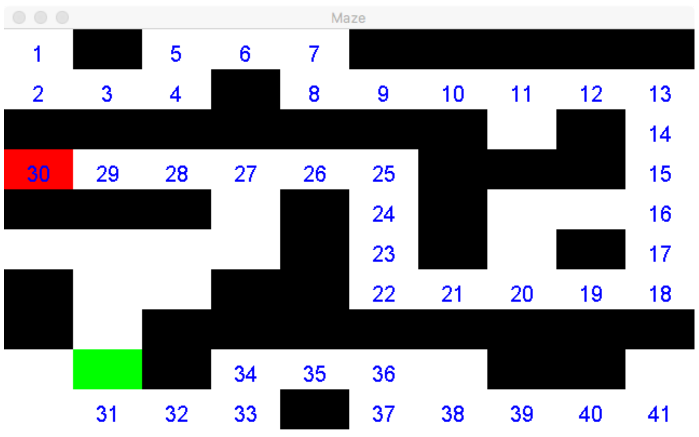
[4]



Compute the shortest path in a form of a list of `Position` from the start cell (0,0) to the end (`SIZE-1`,`SIZE-1`) and visualise it. If no such path is possible, no path should be displayed. A possible path for the map map1.csv is shown above, with each step of the path numbered. Note that the shortest path uses a teleporter to shorten the walk to the exit. Example of a maze where no path exists is provided in map2_deadend.csv .

[20]

How to find a shortest path

This is called a breadth-first search. Consider a map `Map<Position,Integer> distance` from positions to the length of a shortest path from an initial state (0,0). When you begin, this only contains a single entry for the initial state with a length of 0. You then look at all valid cells reachable from the initial cell, their distance from the initial cell is 1. From those cells, you consider subsequent valid cells, with a distance increased by 1. It is possible that at some point you will find that there is a shorter path to a particular cell than the one you have seen before – this is the case where there are multiple paths leading to the same cell. In this case, you replace the previous distance by the new (smaller) distance.

The algorithm hence maintains

```
    Queue<Position> posQueue
```

that holds positions that need to be considered and then you have a loop that *removes* a position from the queue to consider it. Call this position `pos`. Consideration involves looking at all possible directions around that position. You can use a copy constructor to make a copy of `pos` and then use the `move` method to compute the new position `newPos`. Valid cells are those with `newPos` containing coordinates 0..`SIZE-1` and then you explore what `distance` map returns for `newPos`,

1. if this is null, you have not explored that cell before, or
2. if it is a larger value than the distance from the initial state for the current path (1+what `distance` map has for `pos`), you have found a shorter alternative path.

In both of these cases you have to consider all directions from `newPos` cells and therefore *offer* `newPos` to the `posQueue`. The algorithm terminates when `posQueue` becomes empty.

Every time you modify `distance`, you could record the path to the initial state that got you to that state. This can be done by maintaining a current path from the initial state that is kept on a separate queue `pathQueue`. When you add something to `posQueue`, you add the modified path to `pathQueue` and when you pop from `posQueue`, you also pop from `pathQueue`.

An alternative is to have a map `Map<Position,Position> prevElement` from a position of a cell to the position you have used to enter that cell. In this way, when you find a new shortest path and hence add an entry to `distance`, you also add `pos` as a value to `prevElement` for key `newPos`.