

Maze PA Explanation

Mark Redekopp

Maze Solver

- Consider this maze
 - S = Start
 - F = Finish
 - . = Free
 - # = Wall
- Find the shortest path

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

Maze Solver

- To find **a** (there might be many) shortest path we use a breadth-first search (BFS)
- BFS requires we visit all nearer squares before further squares
 - A simple way to meet this requirement is to make a square "get in line" (i.e. a queue) when we encounter it
 - We will pull squares from the front to explore and add new squares to the back of the queue

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
.	.	.	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#

Queue



[illegible]

Maze Solver

- We start by putting the starting location into the queue
- Then we enter a loop...while the queue is not empty
 - Extract the front location, call it "curr"
 - Visit each neighbor (N,W,S,E) one at a time
 - If the neighbor is the finish
 - Stop and trace backwards
 - Else if the neighbor is a valid location and not visited before
 - Then add it to the back of the queue
 - Mark it as visited so we don't add it to the queue again
 - Record its predecessor (the location [i.e. curr] that found this neighbor)

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

Queue

[illegible]

Visited

(0,0) 0	(0,1) 0	(0,2) 0	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 0	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 0	(3,1) 0	(3,2) 0	(3,3) 0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
-1,-1	-1,-1	-1,-1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
-1,-1	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
-1,-1	-1,-1	-1,-1	-1,-1

Maze Solver

- We start by putting the starting location into the queue
- Then we enter a loop...while the queue is not empty
 - Extract the front location, call it "curr"
 - Visit each neighbor (N,W,S,E) one at a time
 - If the neighbor is the finish
 - Stop and trace backwards
 - Else if the neighbor is a valid location and not visited before
 - Then add it to the back of the queue
 - Mark it as visited so we don't add it to the queue again
 - Record its predecessor (the location [i.e. curr] that found this neighbor)

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
.	.	.	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#

curr = 1,0

Queue

1,0	0,0	2,0																	
-----	-----	-----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Visited

(0,0)	(0,1)	(0,2)	(0,3)
1	0	0	0
(1,0)	(1,1)	(1,2)	(1,3)
1	0	0	0
(2,0)	(2,1)	(2,2)	(2,3)
1	0	0	0
(3,0)	(3,1)	(3,2)	(3,3)
0	0	0	0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
1,0	-1,-1	-1,-1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
1,0	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
-1,-1	-1,-1	-1,-1	-1,-1

Maze Solver

- We start by putting the starting location into the queue
- Then we enter a loop...while the queue is not empty
 - Extract the front location, call it "curr"
 - Visit each neighbor (N,W,S,E) one at a time
 - If the neighbor is the finish
 - Stop and trace backwards
 - Else if the neighbor is a valid location and not visited before
 - Then add it to the back of the queue
 - Mark it as visited so we don't add it to the queue again
 - Record its predecessor (the location [i.e. curr] that found this neighbor)

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
.	.	.	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#

curr = 0,0

Queue

1,0	0,0	2,0	0,1																
-----	-----	-----	-----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Visited

(0,0)	(0,1)	(0,2)	(0,3)
1	1	0	0
(1,0)	(1,1)	(1,2)	(1,3)
1	0	0	0
(2,0)	(2,1)	(2,2)	(2,3)
1	0	0	0
(3,0)	(3,1)	(3,2)	(3,3)
0	0	0	0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
1,0	0,0	-1,-1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
1,0	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
-1,-1	-1,-1	-1,-1	-1,-1

Maze Solver

- We start by putting the starting location into the queue
- Then we enter a loop...while the queue is not empty
 - Extract the front location, call it "curr"
 - Visit each neighbor (N,W,S,E) one at a time
 - If the neighbor is the finish
 - Stop and trace backwards
 - Else if the neighbor is a valid location and not visited before
 - Then add it to the back of the queue
 - Mark it as visited so we don't add it to the queue again
 - Record its predecessor (the location [i.e. curr] that found this neighbor)

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
.	.	.	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#

curr = 2,0

Queue

1,0	0,0	2,0	0,1	3,0															
-----	-----	-----	-----	-----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Visited

(0,0)	(0,1)	(0,2)	(0,3)
1	1	0	0
(1,0)	(1,1)	(1,2)	(1,3)
1	0	0	0
(2,0)	(2,1)	(2,2)	(2,3)
1	0	0	0
(3,0)	(3,1)	(3,2)	(3,3)
1	0	0	0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
1,0	0,0	-1,-1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
1,0	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
2,0	-1,-1	-1,-1	-1,-1

Maze Solver

- We start by putting the starting location into the queue
- Then we enter a loop...while the queue is not empty
 - Extract the front location, call it "curr"
 - Visit each neighbor (N,W,S,E) one at a time
 - If the neighbor is the finish
 - Stop and trace backwards
 - Else if the neighbor is a valid location and not visited before
 - Then add it to the back of the queue
 - Mark it as visited so we don't add it to the queue again
 - Record its predecessor (the location [i.e. curr] that found this neighbor)

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
.	.	.	.
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0)	(2,1) #	(2,2)	(2,3) #
(3,0)	(3,1)	(3,2)	(3,3) #

curr = 0,1

Queue

1,0	0,0	2,0	0,1	3,0	0,2														
-----	-----	-----	-----	-----	-----	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Visited

(0,0)	(0,1)	(0,2)	(0,3)
1	1	1	0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 1	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 1	(3,1) 0	(3,2) 0	(3,3) 0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
1,0	0,0	0,1	-1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) 1,0	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) 2,0	(3,1) -1,-1	(3,2) -1,-1	(3,3) -1,-1

Maze Solver

- We start by putting the starting location into the queue
- Then we enter a loop...while the queue is not empty
 - Extract the front location, call it "curr"
 - Visit each neighbor (N,W,S,E) one at a time
 - If the neighbor is the finish
 - Stop and trace backwards
 - Else if the neighbor is a valid location and not visited before
 - Then add it to the back of the queue
 - Mark it as visited so we don't add it to the queue again
 - Record its predecessor (the location [i.e. curr] that found this neighbor)

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
.	.	.	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#

curr = 3,0

Queue

1,0	0,0	2,0	0,1	3,0	0,2	3,1										
-----	-----	-----	-----	-----	-----	-----	--	--	--	--	--	--	--	--	--	--

Visited

(0,0)	(0,1)	(0,2)	(0,3)
1	1	1	0
(1,0)	(1,1)	(1,2)	(1,3)
1	0	0	0
(2,0)	(2,1)	(2,2)	(2,3)
1	0	0	0
(3,0)	(3,1)	(3,2)	(3,3)
1	1	0	0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
1,0	0,0	0,1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
1,0	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
2,0	3,0	-1,-1	-1,-1

Maze Solver

- We start by putting the starting location into the queue
- Then we enter a loop...while the queue is not empty
 - Extract the front location, call it "curr"
 - Visit each neighbor (N,W,S,E) one at a time
 - If the neighbor is the finish
 - Stop and trace backwards
 - Else if the neighbor is a valid location and not visited before
 - Then add it to the back of the queue
 - Mark it as visited so we don't add it to the queue again
 - Record its predecessor (the location [i.e. curr] that found this neighbor)

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
.	.	.	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#

curr = 0,2

Found the Finish at (1,2)

Queue

1,0	0,0	2,0	0,1	3,0	0,2	3,1										
-----	-----	-----	-----	-----	-----	-----	--	--	--	--	--	--	--	--	--	--

Visited

(0,0)	(0,1)	(0,2)	(0,3)
1	1	1	0
(1,0)	(1,1)	(1,2)	(1,3)
1	0	0	0
(2,0)	(2,1)	(2,2)	(2,3)
1	0	0	0
(3,0)	(3,1)	(3,2)	(3,3)
1	1	0	0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
1,0	0,0	0,1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
1,0	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
2,0	3,0	-1,-1	-1,-1

Maze Solver

- Now we need to backtrack and add '*'s to our shortest path
- We use the predecessor array to walk backwards from curr to the start
 - Set `maze[curr] = '*'`
 - Not real syntax (as 'curr' is a Location struct)
 - Change `curr = pred[curr]`

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
.	.	*	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#

curr = 0,2

curr = pred[curr]

Queue

1,0	0,0	2,0	0,1	3,0	0,2	3,1													
-----	-----	-----	-----	-----	-----	-----	--	--	--	--	--	--	--	--	--	--	--	--	--

Visited

(0,0)	(0,1)	(0,2)	(0,3)
1	1	1	0
(1,0)	(1,1)	(1,2)	(1,3)
1	0	0	0
(2,0)	(2,1)	(2,2)	(2,3)
1	0	0	0
(3,0)	(3,1)	(3,2)	(3,3)
1	1	0	0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
1,0	0,0	0,1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
1,0	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
2,0	3,0	-1,-1	-1,-1

Maze Solver

- Now we need to backtrack and add '*'s to our shortest path
- We use the predecessor array to walk backwards from curr to the start
 - Set $\text{maze}[\text{curr}] = '*'$
 - Not real syntax (as 'curr' is a Location struct)
 - Change $\text{curr} = \text{pred}[\text{curr}]$

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
.	*	*	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#

curr = 0,2

curr = pred[curr]

Queue

1,0	0,0	2,0	0,1	3,0	0,2	3,1													
-----	-----	-----	-----	-----	-----	-----	--	--	--	--	--	--	--	--	--	--	--	--	--

Visited

(0,0)	(0,1)	(0,2)	(0,3)
1	1	1	0
(1,0)	(1,1)	(1,2)	(1,3)
1	0	0	0
(2,0)	(2,1)	(2,2)	(2,3)
1	0	0	0
(3,0)	(3,1)	(3,2)	(3,3)
1	1	0	0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
1,0	0,0	0,1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
1,0	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
2,0	3,0	-1,-1	-1,-1

Maze Solver

- Now we need to backtrack and add '*'s to our shortest path
- We use the predecessor array to walk backwards from curr to the start
 - Set $\text{maze}[\text{curr}] = '*'$
 - Not real syntax (as 'curr' is a Location struct)
 - Change $\text{curr} = \text{pred}[\text{curr}]$

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
*	*	*	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#

curr = 0,2

curr = pred[curr]

Queue

1,0	0,0	2,0	0,1	3,0	0,2	3,1										
-----	-----	-----	-----	-----	-----	-----	--	--	--	--	--	--	--	--	--	--

Visited

(0,0)	(0,1)	(0,2)	(0,3)
1	1	1	0
(1,0)	(1,1)	(1,2)	(1,3)
1	0	0	0
(2,0)	(2,1)	(2,2)	(2,3)
1	0	0	0
(3,0)	(3,1)	(3,2)	(3,3)
1	1	0	0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
1,0	0,0	0,1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
1,0	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
2,0	3,0	-1,-1	-1,-1

Need to Do

- Queue class
 - Make internal array to be of size = max number of squares
 - Should it be dynamic?
 - We need to keep track of the “front” and “back” since only a portion of the array is used
 - Just use integer indexes to record where the front and back are

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
*	*	*	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#

curr = 0,2

curr = pred[curr]

Queue

1,0	0,0	2,0	0,1	3,0	0,2	3,1										
-----	-----	-----	-----	-----	-----	-----	--	--	--	--	--	--	--	--	--	--

Visited

(0,0)	(0,1)	(0,2)	(0,3)
1	1	1	0
(1,0)	(1,1)	(1,2)	(1,3)
1	0	0	0
(2,0)	(2,1)	(2,2)	(2,3)
1	0	0	0
(3,0)	(3,1)	(3,2)	(3,3)
1	1	0	0

Predecessor

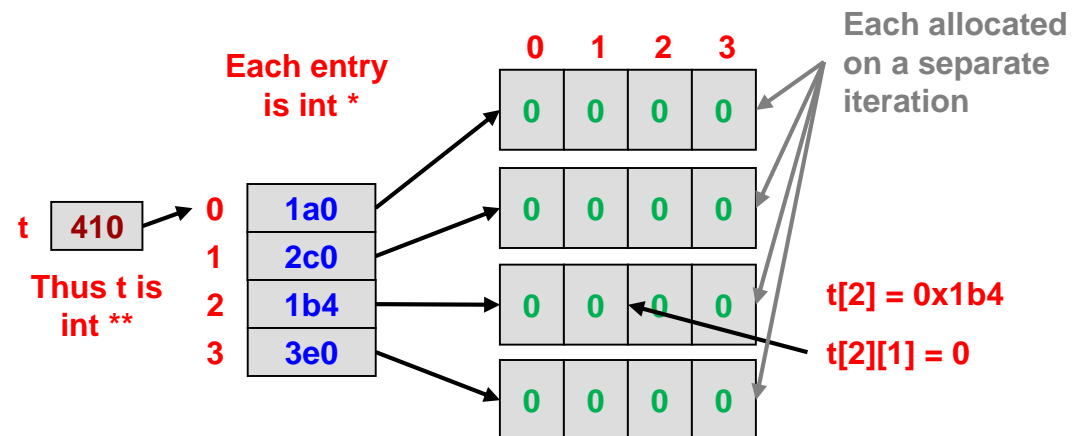
(0,0)	(0,1)	(0,2)	(0,3)
1,0	0,0	0,1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
1,0	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
2,0	3,0	-1,-1	-1,-1

Need to Do

- Allocate 2D arrays for maze, visited, and predecessors
 - Note: in C/C++ you cannot allocate a 2D array with variable size dimensions
 - BAD:** `new int[numrows][numcols];`
 - Solution:
 - Allocate 1 array of NUMROW pointers
 - Then loop through that array and allocate an array of NUMCOL items and put its start address into the i-th array you allocated above

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
*	*	*	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#



BACKUP

Maze Solver

- To find a (there might be many) shortest path we use a breadth-first search (BFS)
- BFS requires we visit all nearer squares before further squares
 - A simple way to meet this requirement is to make a square "get in line" (i.e. a queue) when we encounter it
 - We will pull squares from the front to explore and add new squares to the back of the queue

Maze array:

(0,0)	(0,1)	(0,2)	(0,3)
.	.	.	.
(1,0)	(1,1)	(1,2)	(1,3)
S	#	F	#
(2,0)	(2,1)	(2,2)	(2,3)
.	#	.	#
(3,0)	(3,1)	(3,2)	(3,3)
.	.	.	#

Queue

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Visited

(0,0)	(0,1)	(0,2)	(0,3)
0	0	0	0
(1,0)	(1,1)	(1,2)	(1,3)
1	0	0	0
(2,0)	(2,1)	(2,2)	(2,3)
0	0	0	0
(3,0)	(3,1)	(3,2)	(3,3)
0	0	0	0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
-1,-1	-1,-1	-1,-1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
-1,-1	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
-1,-1	-1,-1	-1,-1	-1,-1

Maze Solver

- To find ***a*** (there might be many) shortest path we use a breadth-first search (BFS)
- BFS requires we visit all nearer squares before further squares
 - A simple way to meet this requirement is to make a square "get in line" (i.e. a queue) when we encounter it
 - We will pull squares from the front to explore and add new squares to the back of the queue

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

curr = 1,0

Queue

[illegible]

Visited

(0,0) 0	(0,1) 0	(0,2) 0	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 0	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 0	(3,1) 0	(3,2) 0	(3,3) 0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
-1,-1	-1,-1	-1,-1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
-1,-1	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
-1,-1	-1,-1	-1,-1	-1,-1

Maze Solver

- To find ***a*** (there might be many) shortest path we use a breadth-first search (BFS)
- BFS requires we visit all nearer squares before further squares
 - A simple way to meet this requirement is to make a square "get in line" (i.e. a queue) when we encounter it
 - We will pull squares from the front to explore and add new squares to the back of the queue

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

Queue

[illegible]

Visited

(0,0) 0	(0,1) 0	(0,2) 0	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 0	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 0	(3,1) 0	(3,2) 0	(3,3) 0

Predecessor

(0,0)	(0,1)	(0,2)	(0,3)
-1,-1	-1,-1	-1,-1	-1,-1
(1,0)	(1,1)	(1,2)	(1,3)
-1,-1	-1,-1	-1,-1	-1,-1
(2,0)	(2,1)	(2,2)	(2,3)
-1,-1	-1,-1	-1,-1	-1,-1
(3,0)	(3,1)	(3,2)	(3,3)
-1,-1	-1,-1	-1,-1	-1,-1