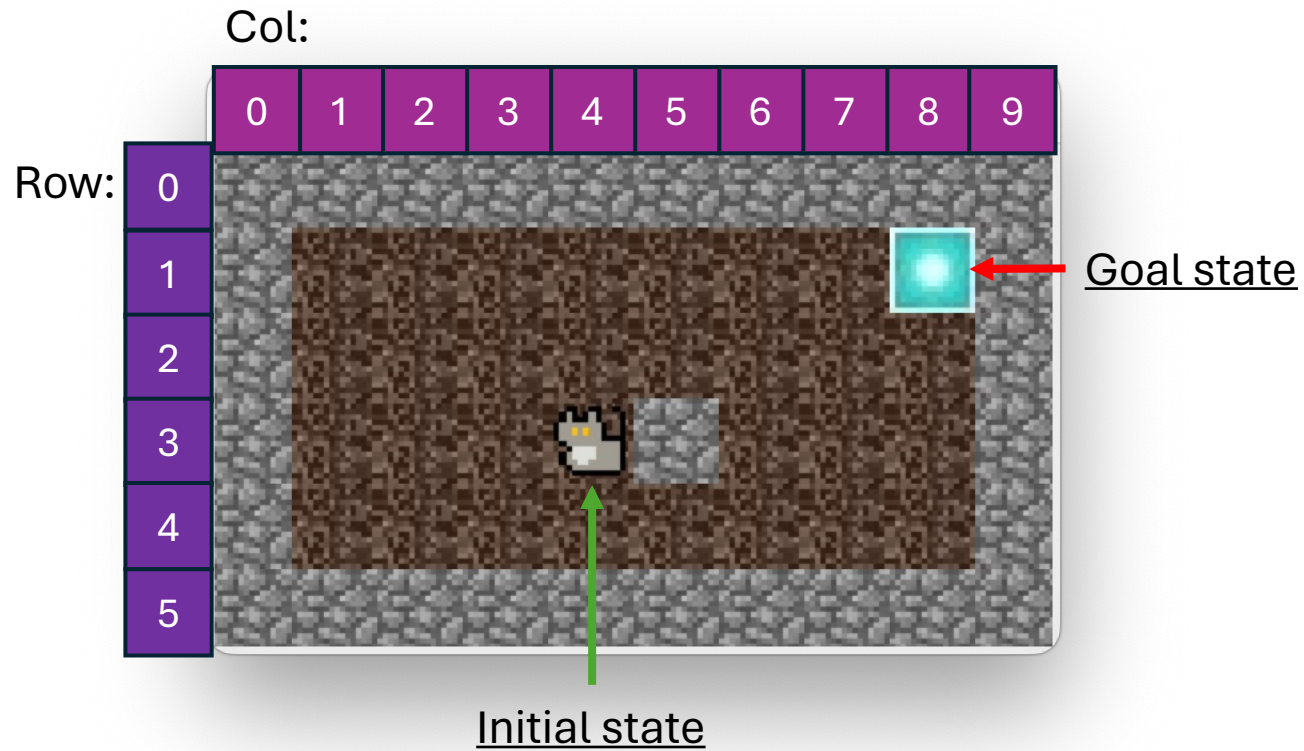


## 2-D Maze Example



Key:



Expanded node



Node on the frontier

## DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9	
0											
1											
2											
3											
4											
5											

## DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

# DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## DFS

Col:






Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1		f								
2		e	f	f	f					
3		e	e	e	e					
4		e	f	f	f					
5										

## DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										



## DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## DFS

Col:

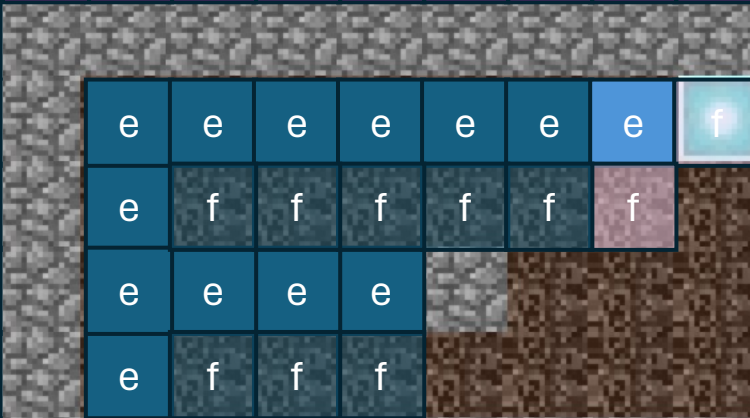
Row:

[illegible]

## DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## DFS

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## DFS

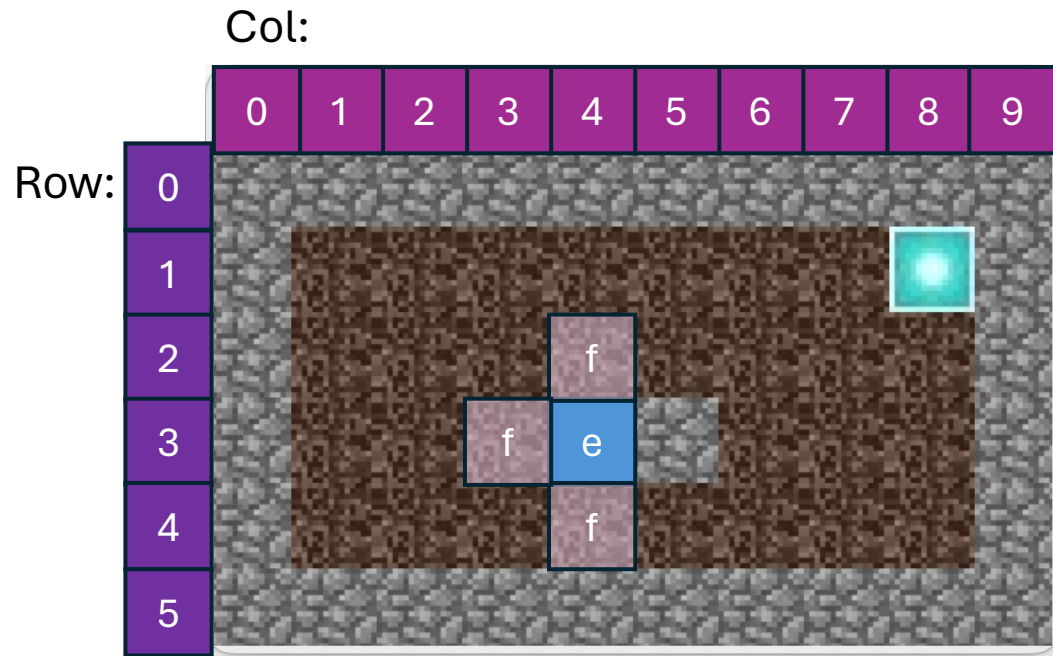
Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1		e	e	e	e	e	e	e	g	
2		e								
3		e	e	e	e					
4		e								
5										



# BFS/UCS with 1 cost per action



# BFS/UCS with 1 cost per action

Col:

Row:

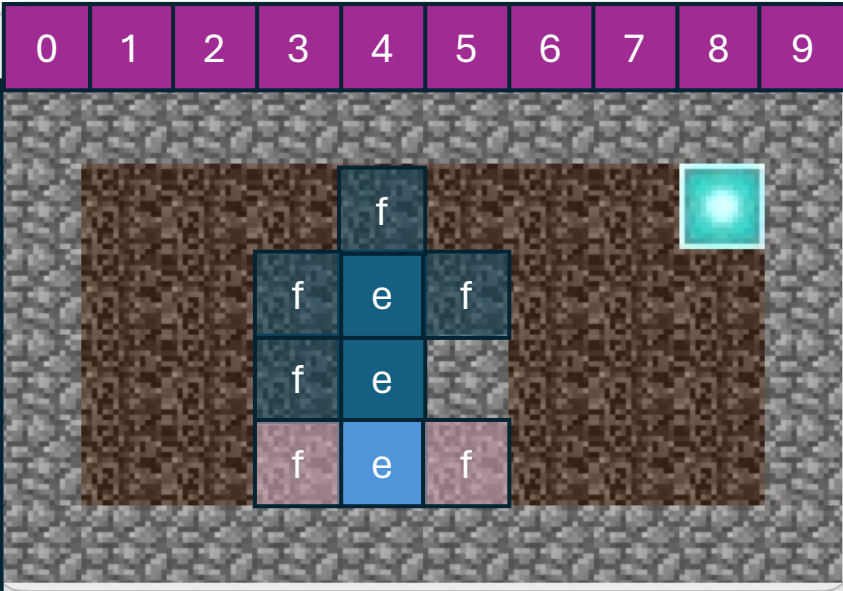
	0	1	2	3	4	5	6	7	8	9
0										
1					f					
2				f	e	f				
3				f	e					
4					f					
5										

# BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1					f					
2				f	e	f				
3				f	e					
4				f	e	f				
5										



## BFS/UCS with 1 cost per action

Col:

Row:

[illegible]

## BFS/UCS with 1 cost per action

Col:

Row:

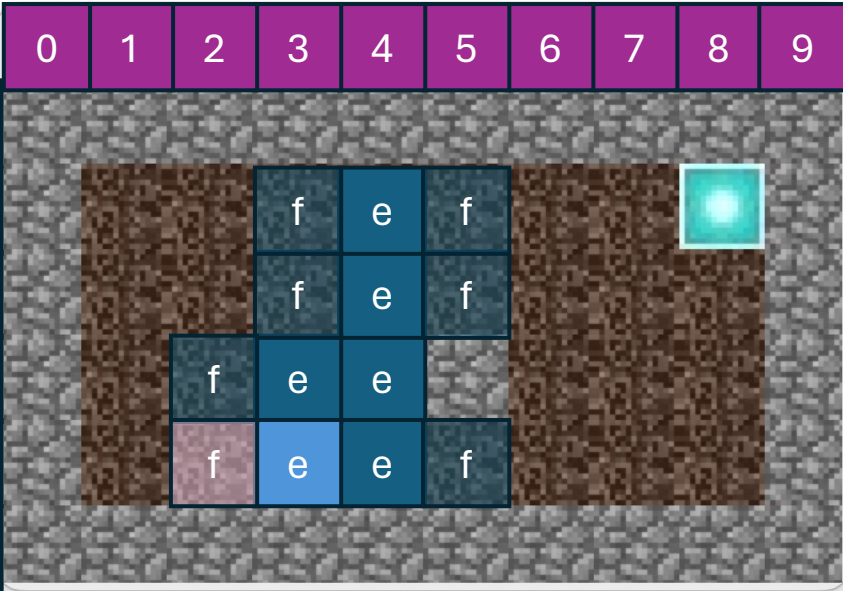
	0	1	2	3	4	5	6	7	8	9	
0											
1											
2											
3											
4											
5											

# BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1				f	e	f				
2				f	e	f				
3			f	e	e					
4			f	e	e	f				
5										



## BFS/UCS with 1 cost per action

Col:

Row:

[illegible]

## BFS/UCS with 1 cost per action

Col:

Row:

[illegible]



## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

Row:

[illegible]

## BFS/UCS with 1 cost per action

Col:

Row:

[illegible]

## BFS/UCS with 1 cost per action

Col:

Row:

[illegible]

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:


Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1			f	e	e	e	f			
2		f	e	e	e	e	e	f		
3		f	e	e	e		f			
4		f	e	e	e	e	e	f		
5										



## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1		f	e	e	e	e	f			
2		f	e	e	e	e	e	f		
3		e	e	e	e		f			
4		f	e	e	e	e	e	f		
5										

## BFS/UCS with 1 cost per action

Col:

Row:

[illegible]

## BFS/UCS with 1 cost per action

Col:

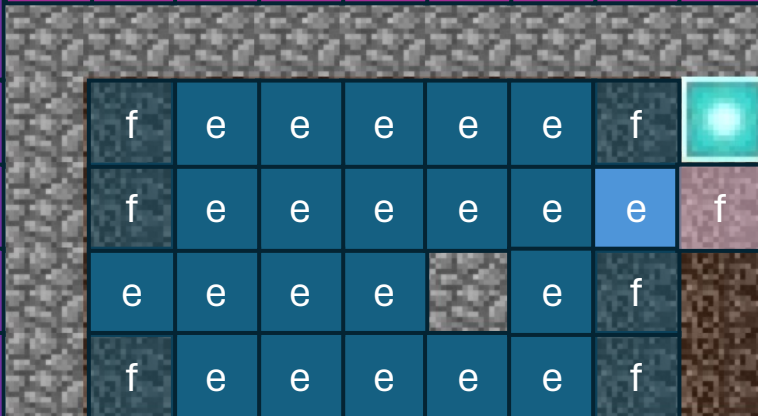
Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:



Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

Row:


	0	1	2	3	4	5	6	7	8	9
0										
1		f	e	e	e	e	e	f		
2		e	e	e	e	e	e	e	f	
3		e	e	e	e		e	f		
4		e	e	e	e	e	e	e	f	
5										



## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

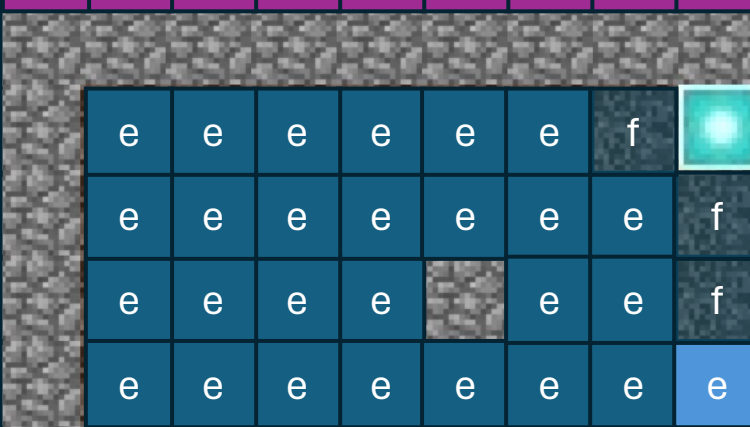
Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1		e	e	e	e	e	e	e	f	
2		e	e	e	e	e	e	e	f	
3		e	e	e	e		e	e	f	
4		e	e	e	e	e	e	e	e	
5										

In **BFS**, the goal check is performed on node **generation**, so BFS will find the goal at this step. **UCS** performs the goal check on node **expansion** (i.e. when the node is selected from the frontier), so continues searching.

```

function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
  
```

```

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
  
```

```

function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
  
```

**Figure 3.7** The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for **yield**.

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1		e	e	e	e	e	e	e	g	
2		e	e	e	e	e	e	e	e	
3		e	e	e	e		e	e	e	
4		e	e	e	e	e	e	e	e	
5										

Since **UCS** performs the goal check on node expansion (i.e. when the node is selected from the frontier), it only detects the goal now.

```

function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
  
```

```

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
  
```

```

function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure


function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
  
```

**Figure 3.7** The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for **yield**.

## BFS/UCS with 1 cost per action

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										



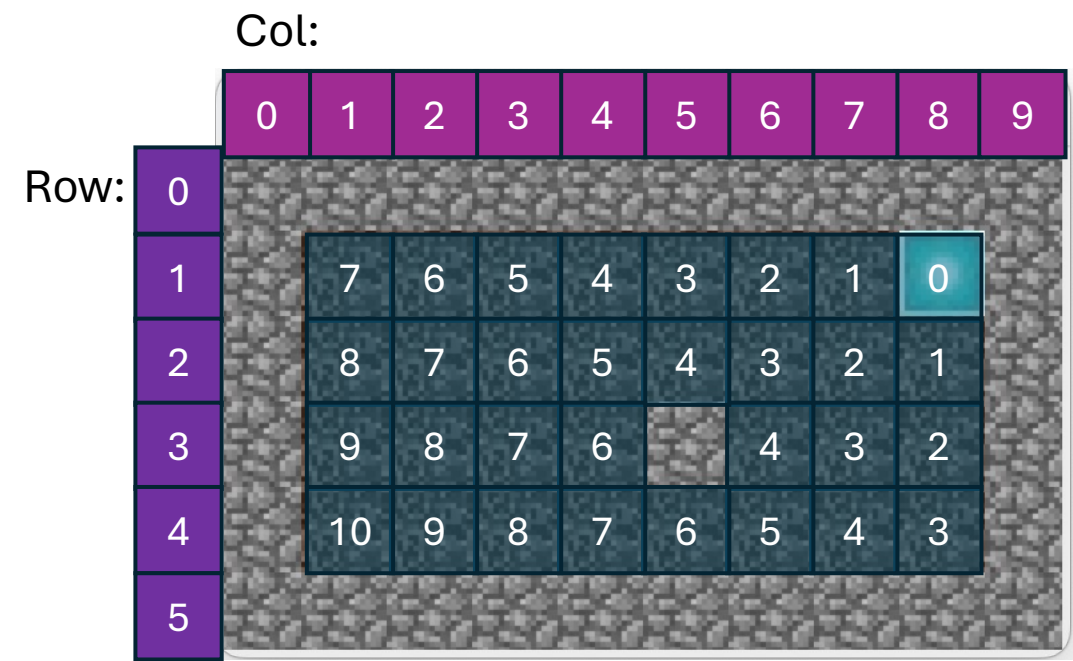
Heuristic  $\rightarrow$  Manhattan distance to the goal

Col:

Row:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

GBFS with heuristic,  $h(n)$  = Manhattan distance to the goal



GBFS with heuristic,  $h(n)$  = Manhattan distance to the goal

Col:

[illegible]

GBFS with heuristic,  $h(n)$  = Manhattan distance to the goal

Col:

	0	1	2	3	4	5	6	7	8	9
Row: 0										
1										
2										
3										
4										
5										

	7	6	5	4	3	2	1	0
8	7	6	5	e	4	3	2	1
9	8	7	e			4	3	2
10	9	8	7	6	5	4	3	

GBFS with heuristic,  $h(n)$  = Manhattan distance to the goal

Col:

	0	1	2	3	4	5	6	7	8	9
Row: 0										
1		7	6	5	e	3	2	1	0	
2		8	7	6	e	4	3	2	1	
3		9	8	7	e		4	3	2	
4		10	9	8	7	6	5	4	3	
5										

GBFS with heuristic,  $h(n)$  = Manhattan distance to the goal

Col:

[illegible]

GBFS with heuristic,  $h(n)$  = Manhattan distance to the goal

Col:

[illegible]

GBFS with heuristic,  $h(n)$  = Manhattan distance to the goal

Col:

[illegible]



GBFS with heuristic,  $h(n)$  = Manhattan distance to the goal

Col:

[illegible]

GBFS with heuristic,  $h(n)$  = Manhattan distance to the goal

