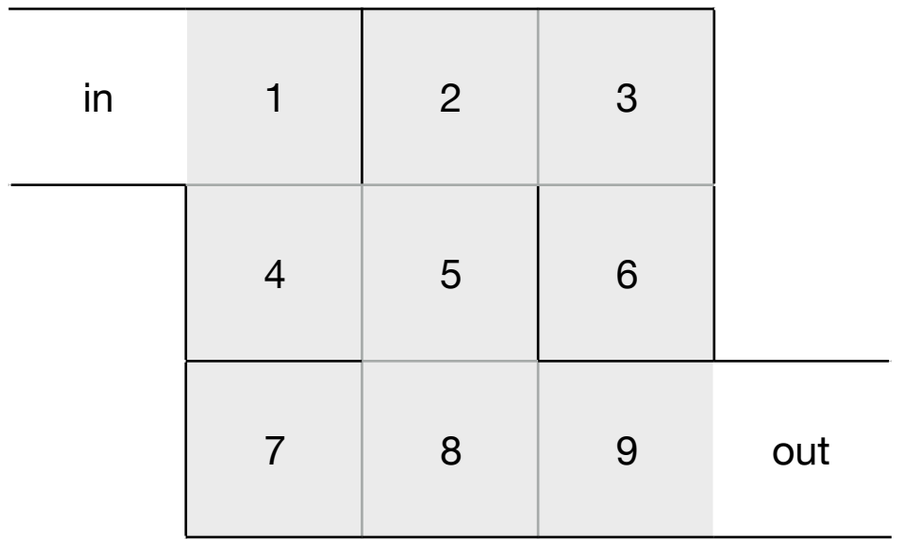
AI Games course

Certificate 1, session 1

The maze example

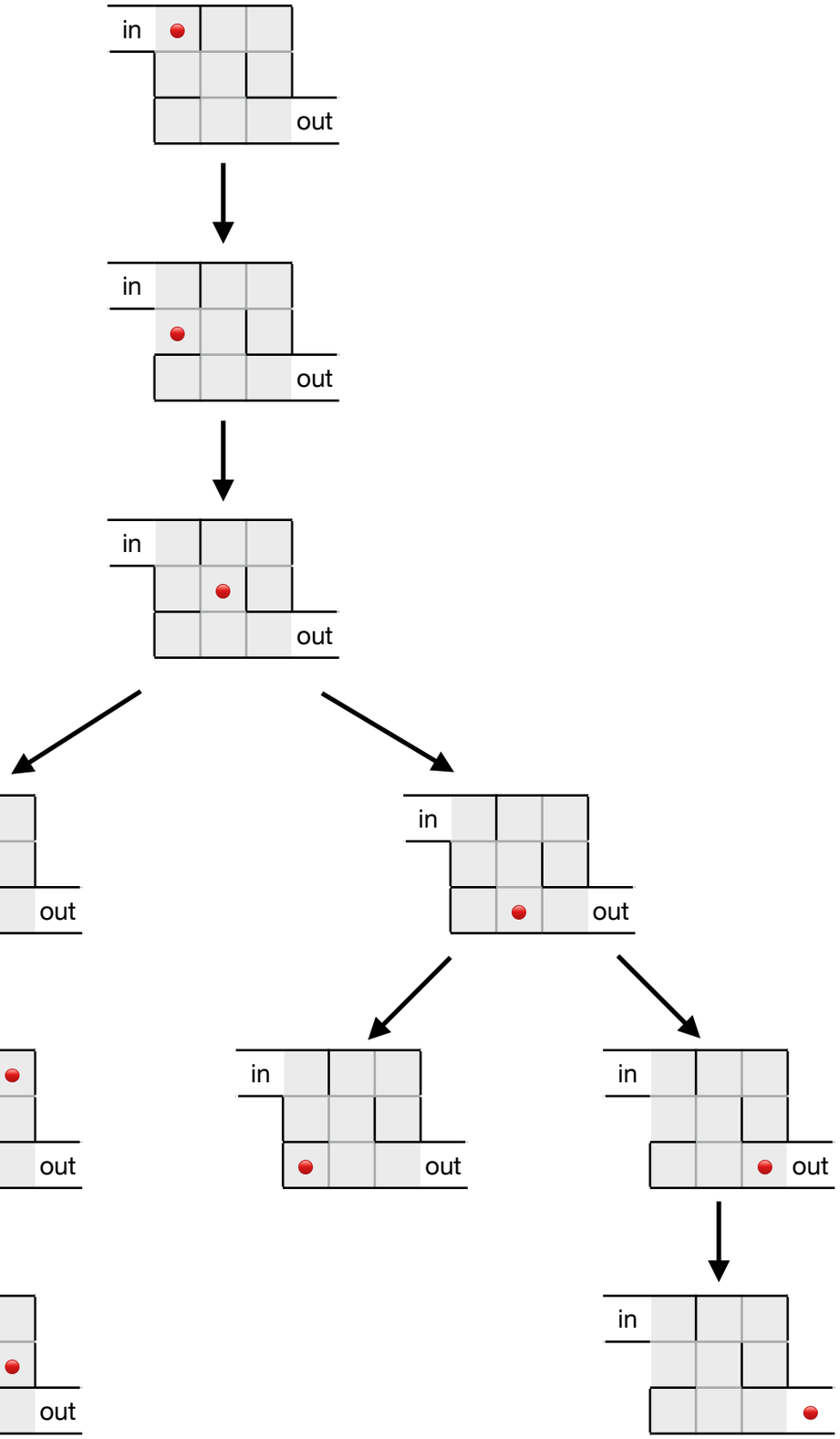
in	1	2	3
	4	5	6
	7	8	9
			out

The maze example: graph representation

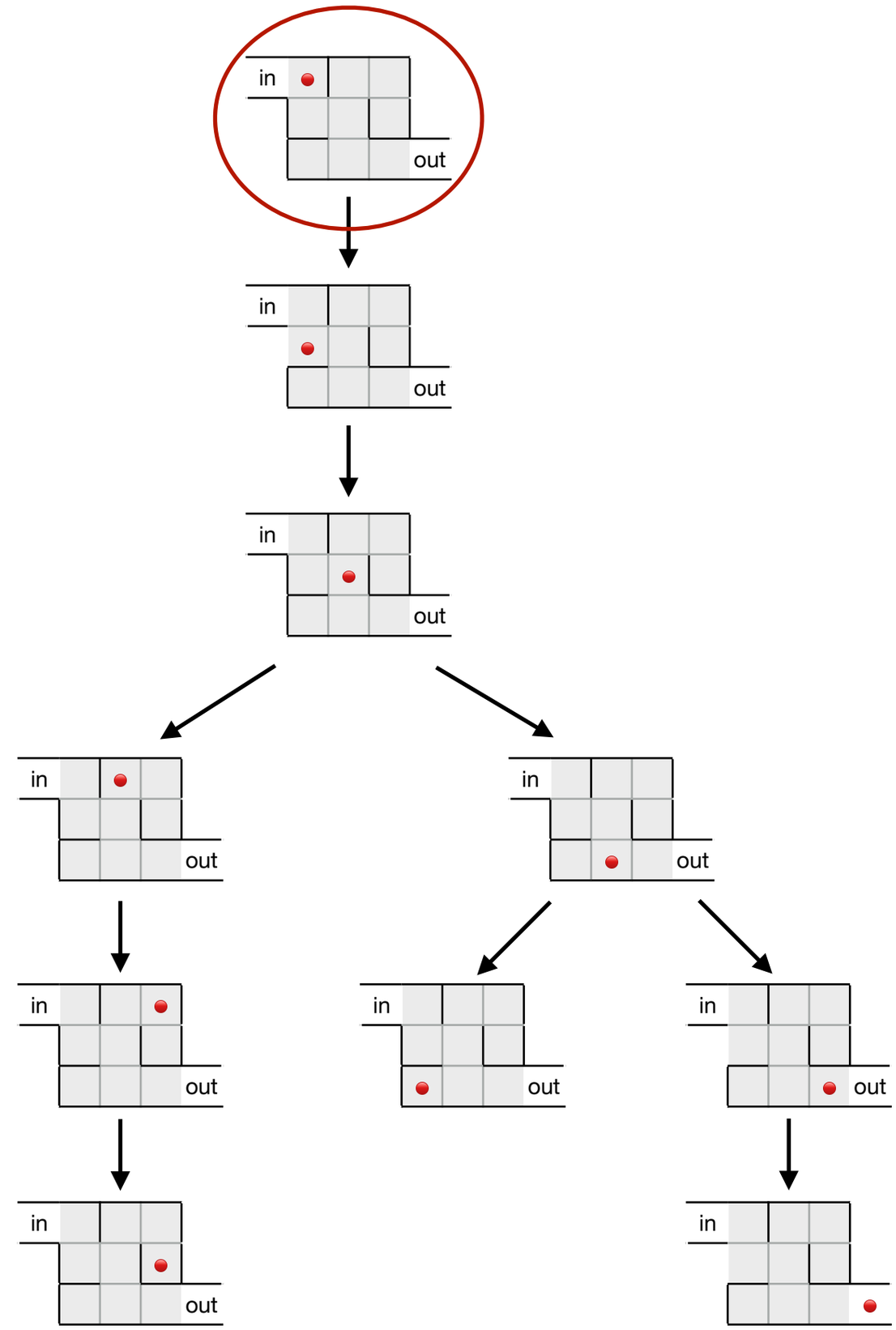


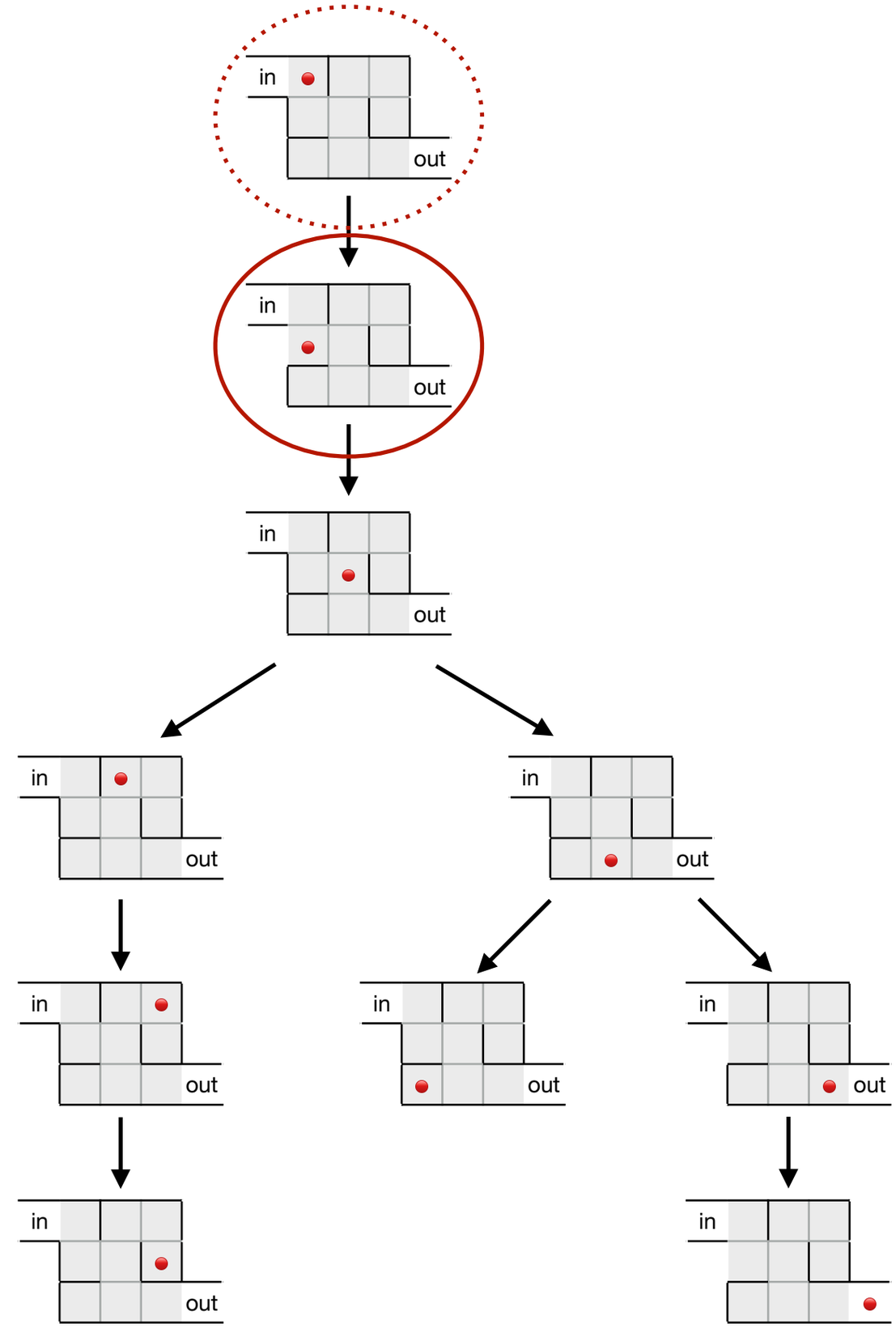

```
maze = {'in': {1},
         1: {'in',4},
         2: {3,5},
         3: {2,6},
         4: {1,5},
         5: {2,4,8},
         6: {3},
         7: {8},
         8: {5,7,9},
         9: {8,'out'},
       'out': {9}}
```

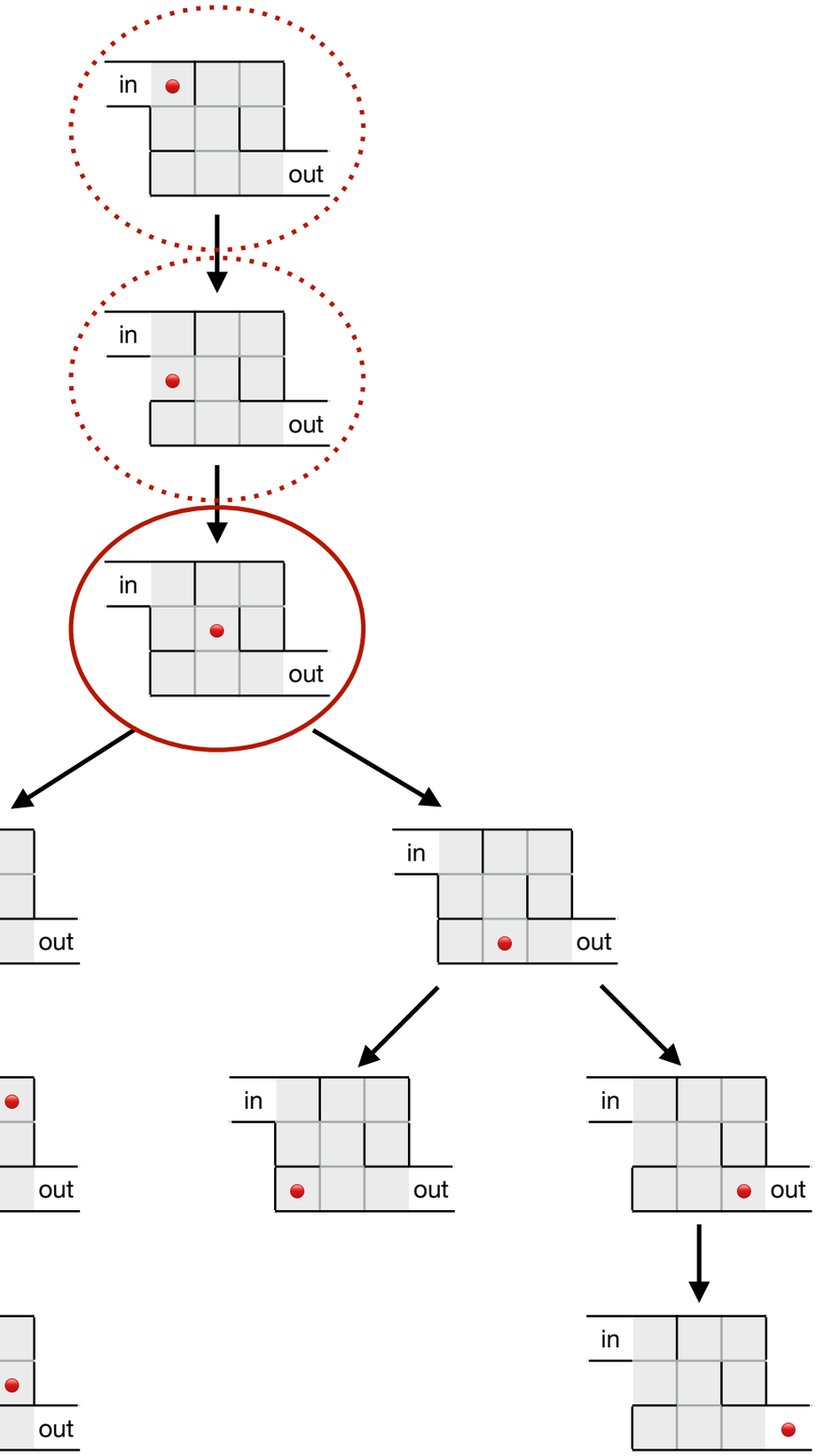
Maze
“space tree”

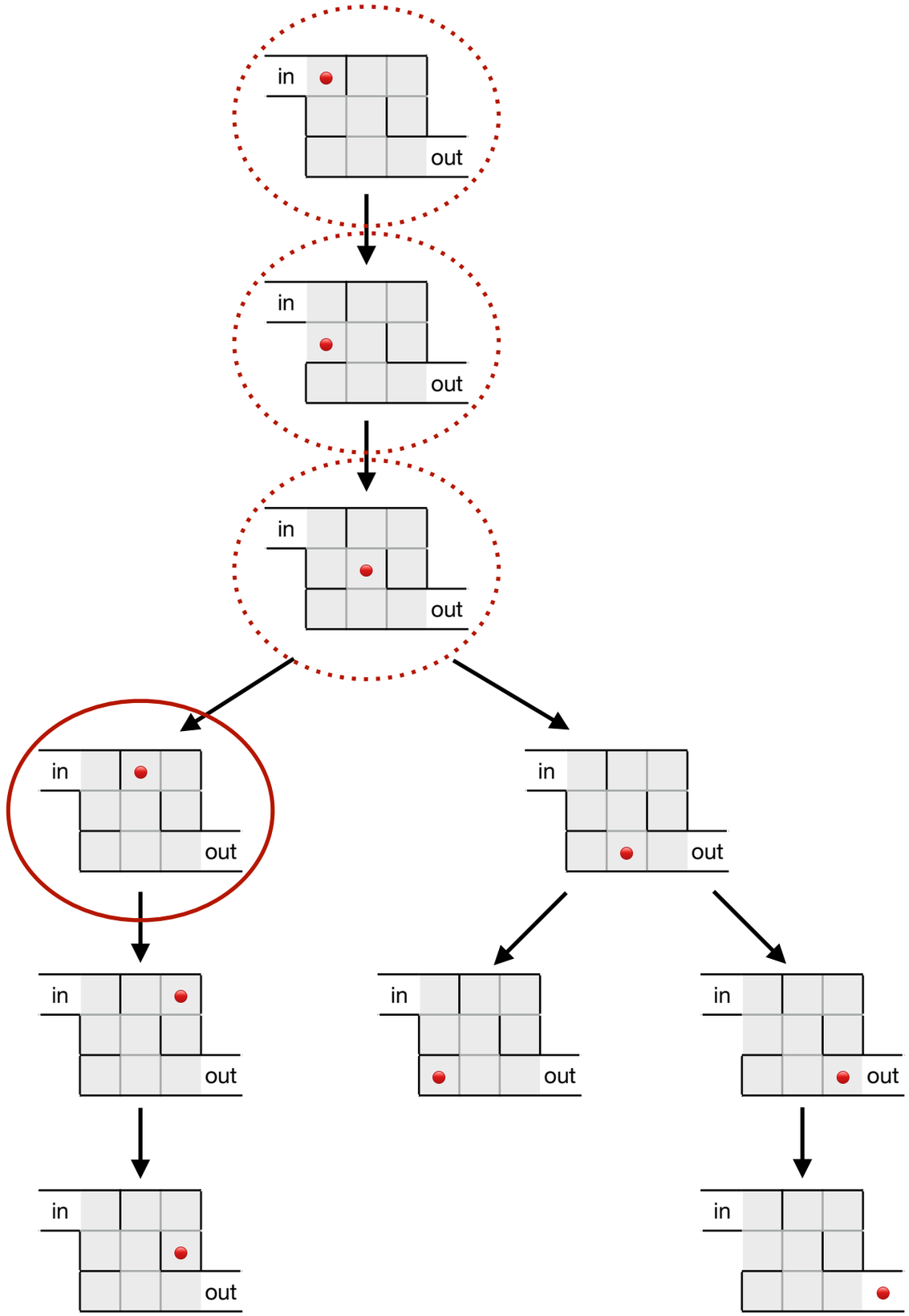


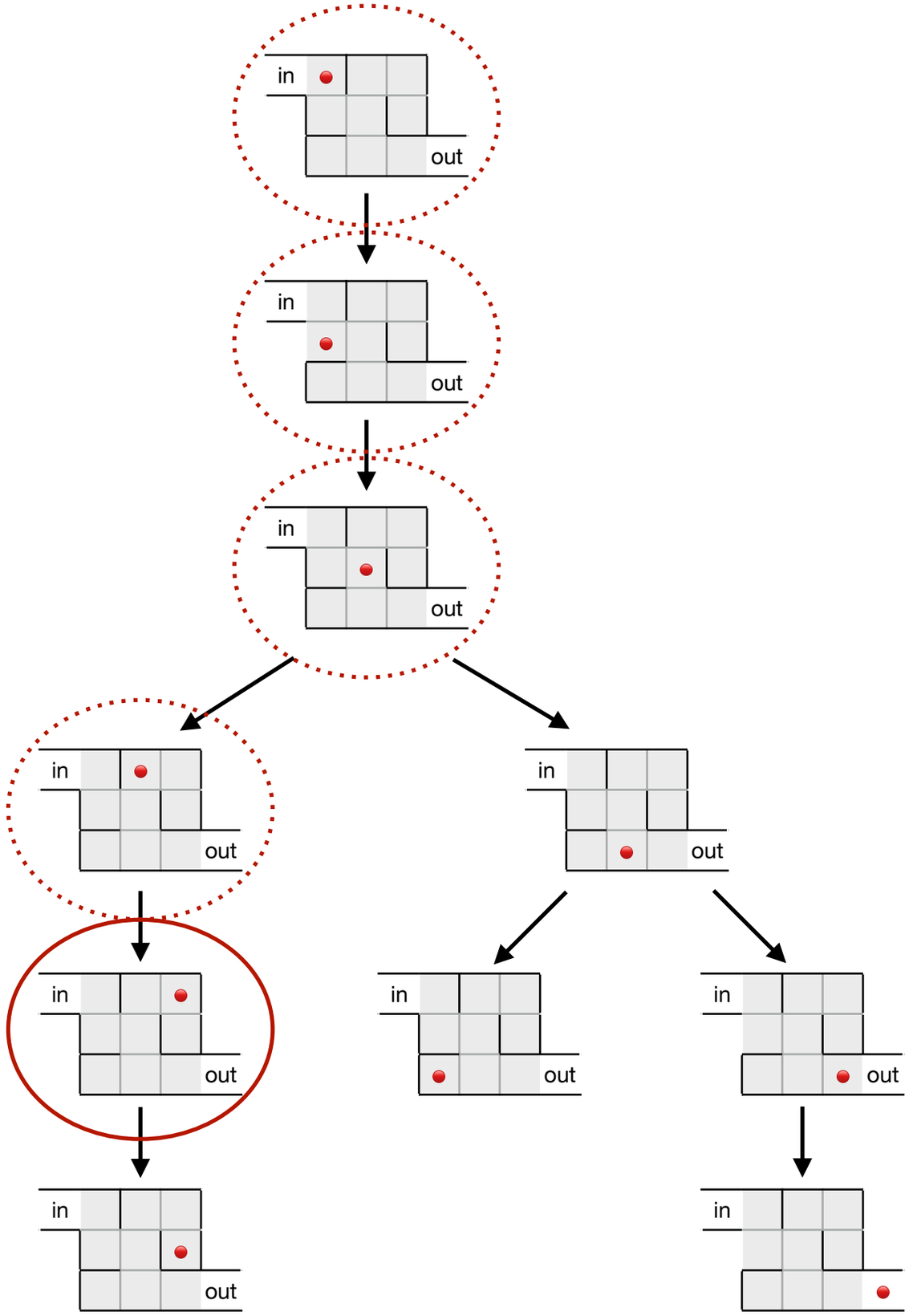
Depth-first search

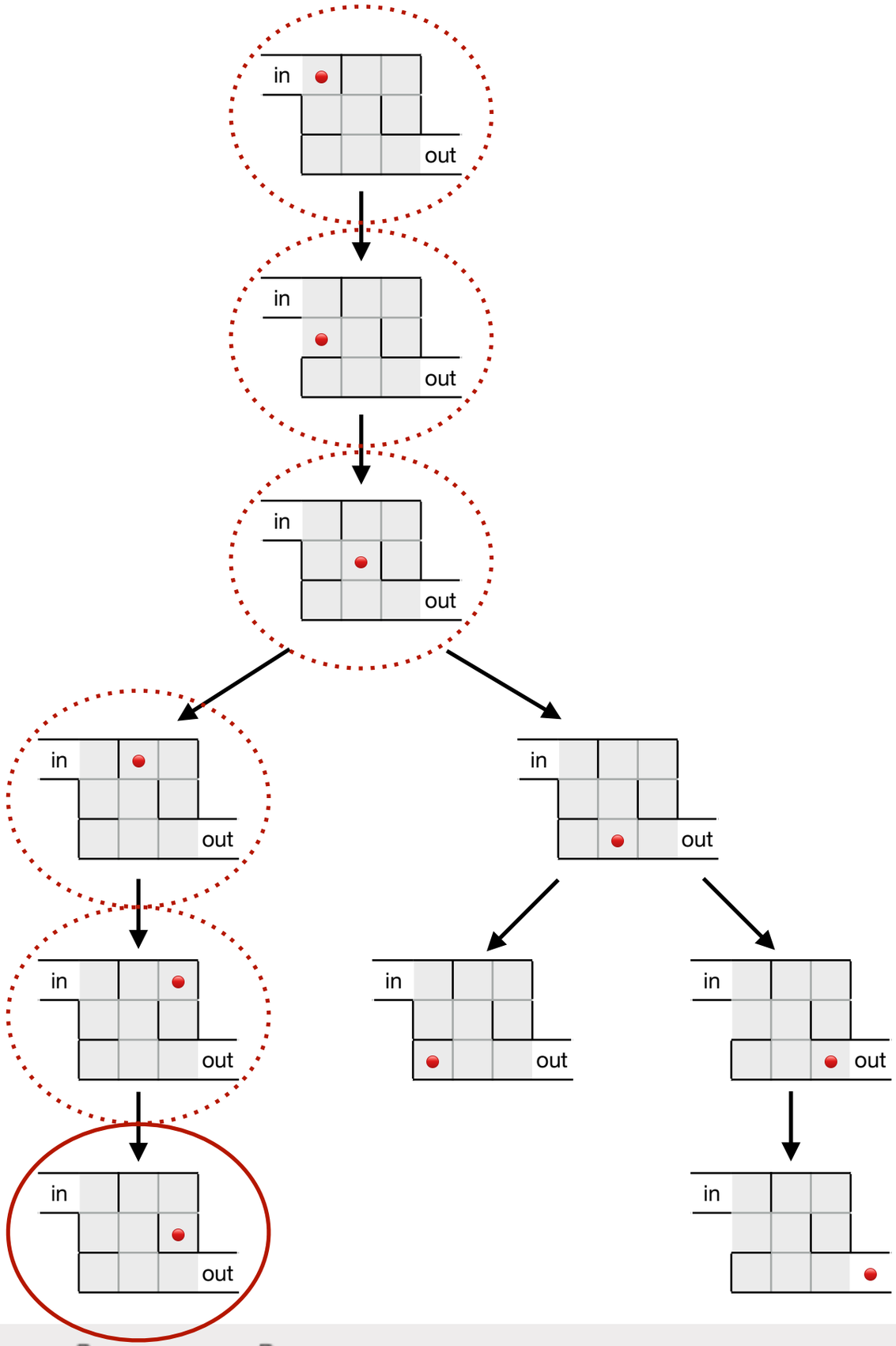


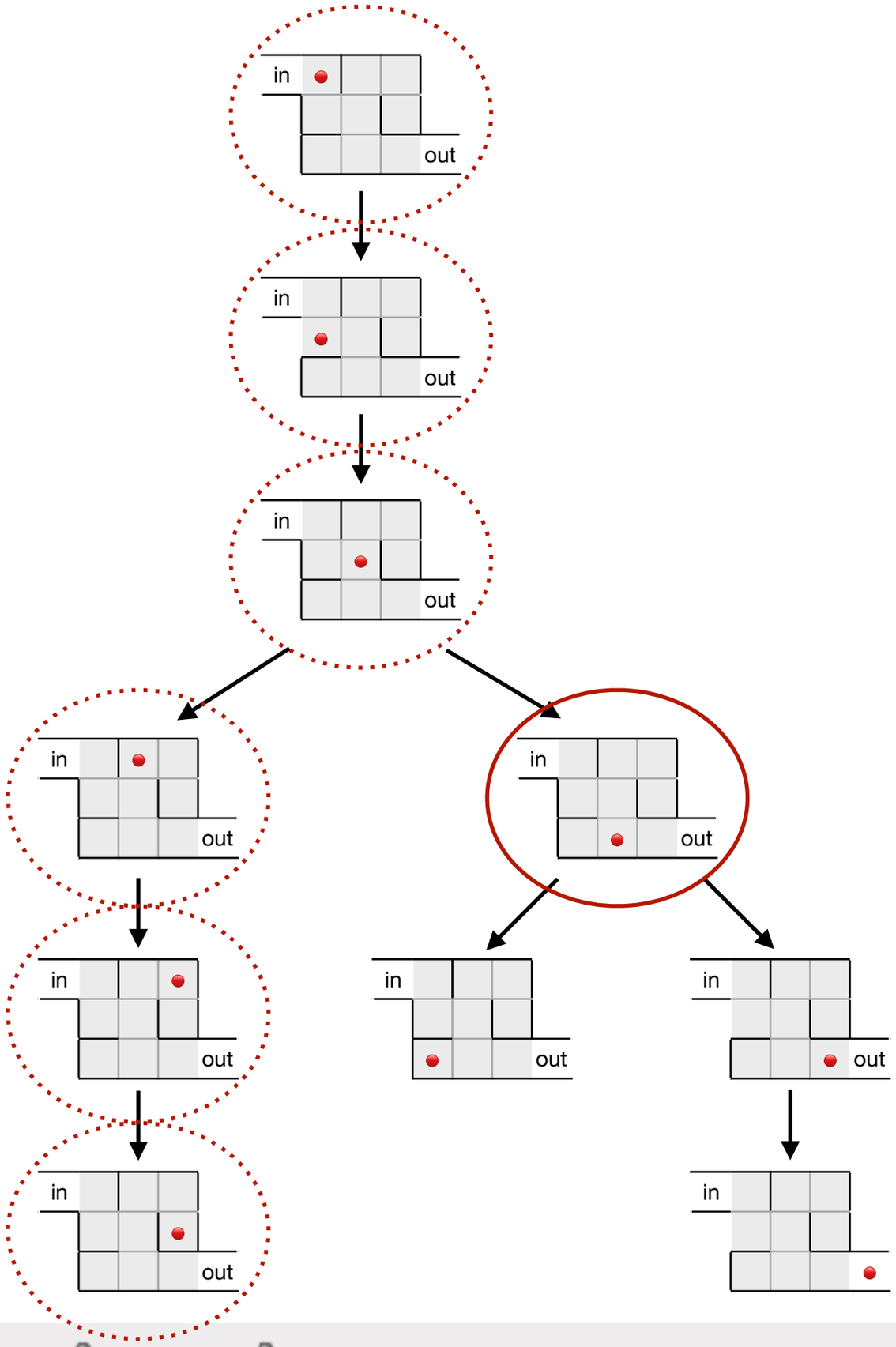


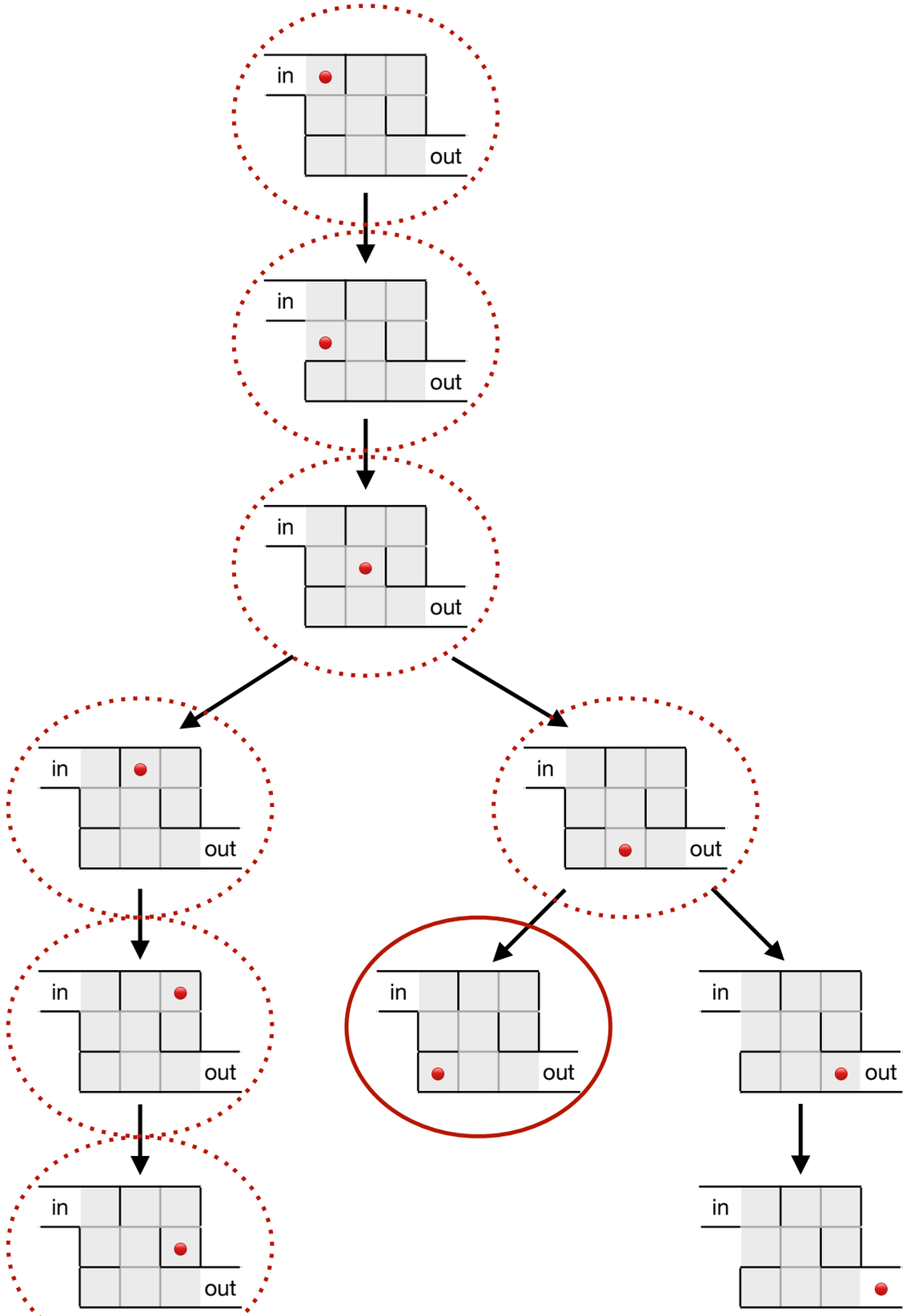


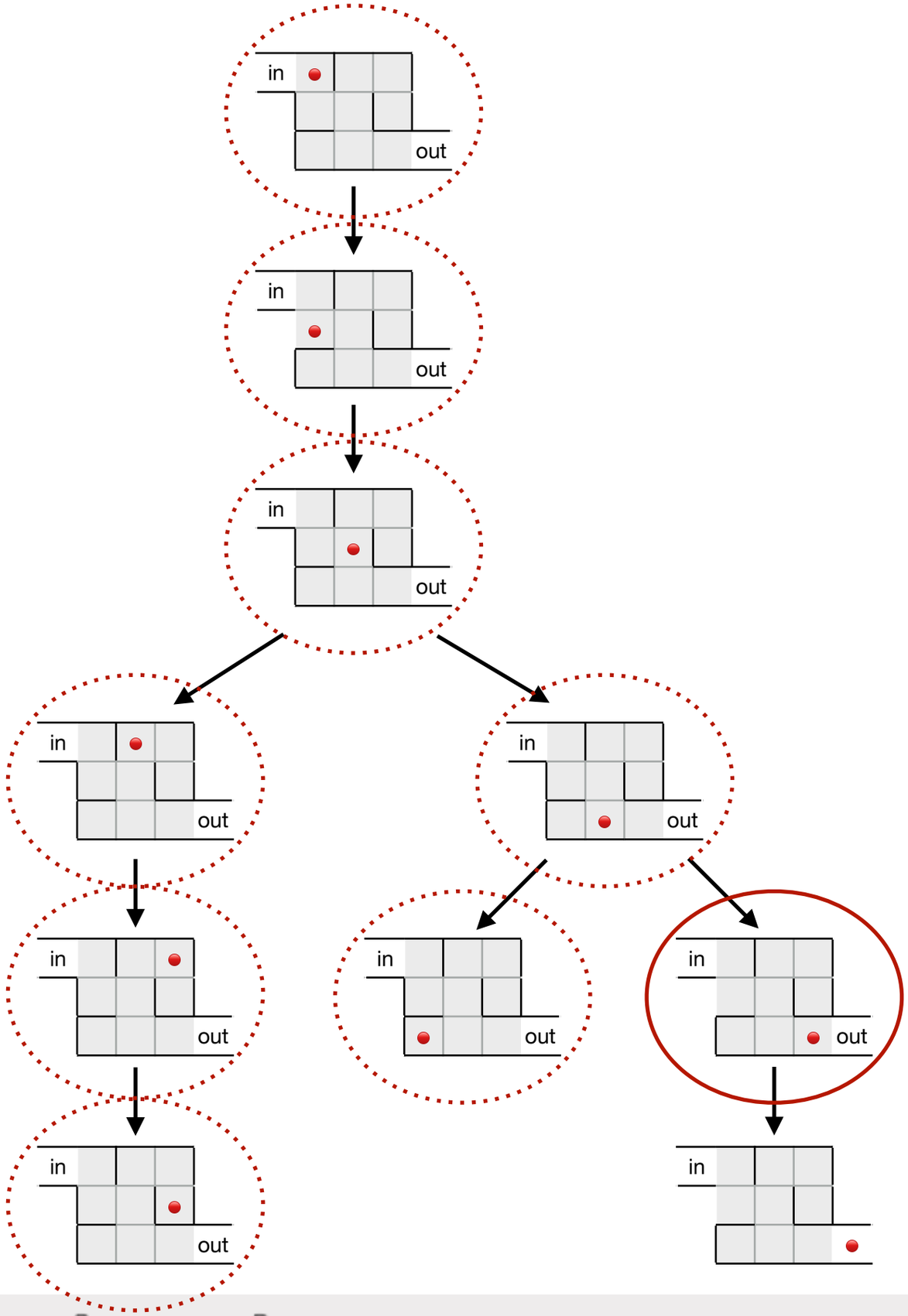


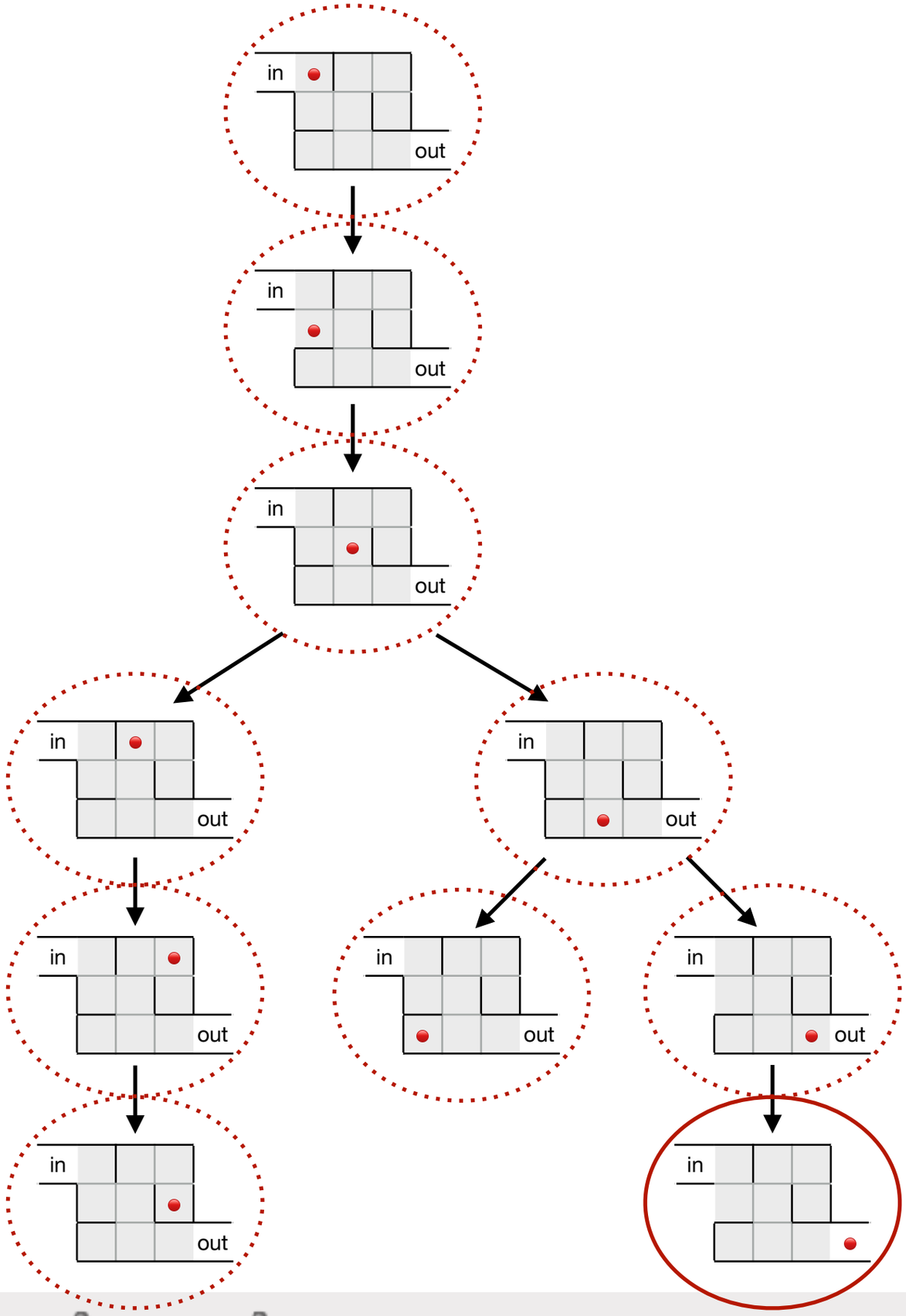












DFS algorithm

DFS algorithm

DFS: given a graph **G** and a starting node **n**

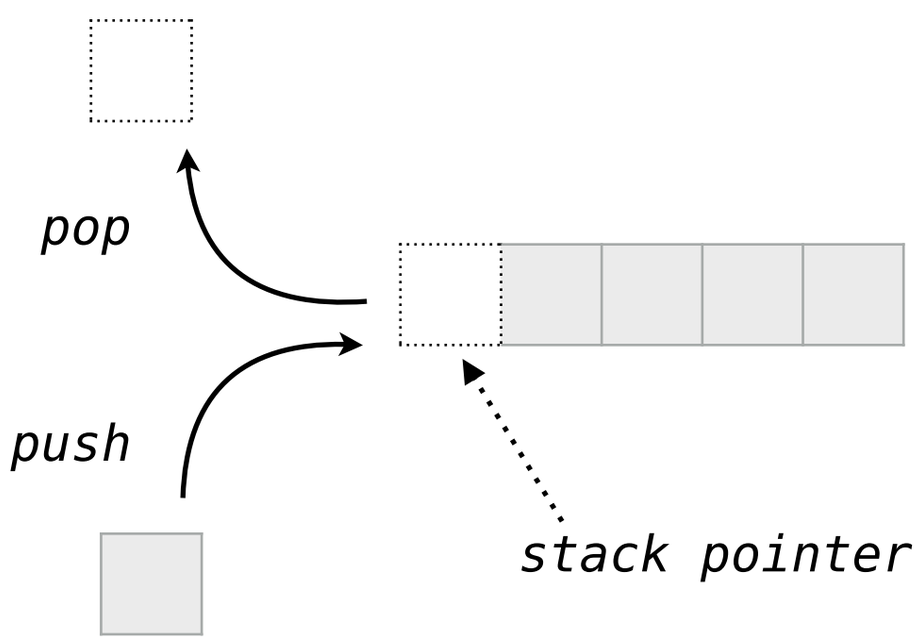
0. put **n** into a *stack*
1. *pop top node* from the stack and mark it as *visited*
2. put into the *stack* all the nodes reachable from the top node and not yet visited
3. while the stack is not empty, recursively apply steps 1–3

DFS algorithm

DFS: given a graph **G** and a starting node **n**

0. put **n** into a *stack*
1. *pop top node* from the stack and mark it as *visited*
2. put into the *stack* all the nodes reachable from the top node and not yet visited
3. while the stack is not empty, recursively apply steps 1–3

Stack:



LIFO:
last in,
first out

DFS algorithm

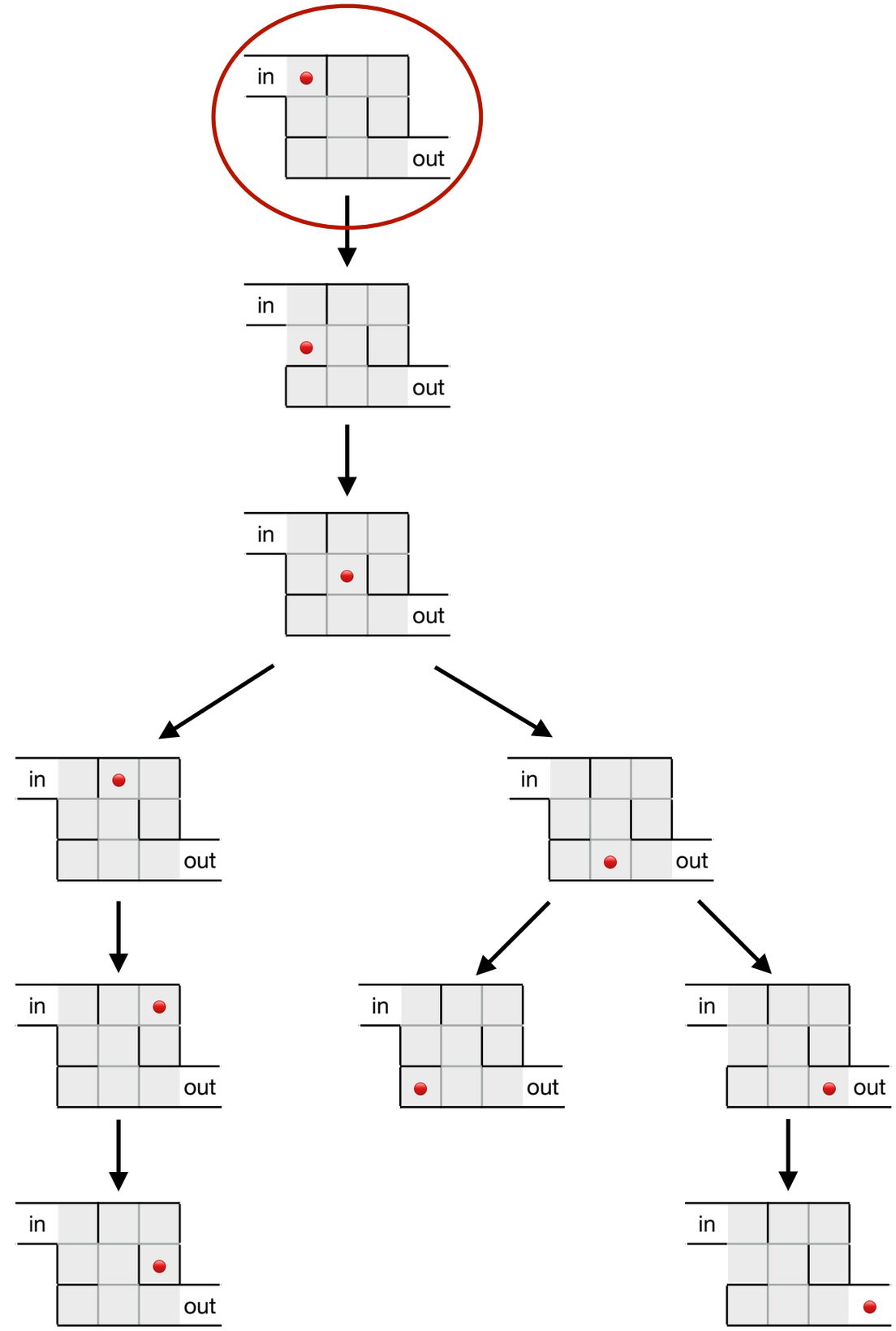
```
1   def dfs(graph, start):
2       visited, stack = set(), [start]
3       while stack:
4           vertex = stack.pop()
5           if vertex not in visited:
6               visited.add(vertex)
7
8           print(vertex)
```

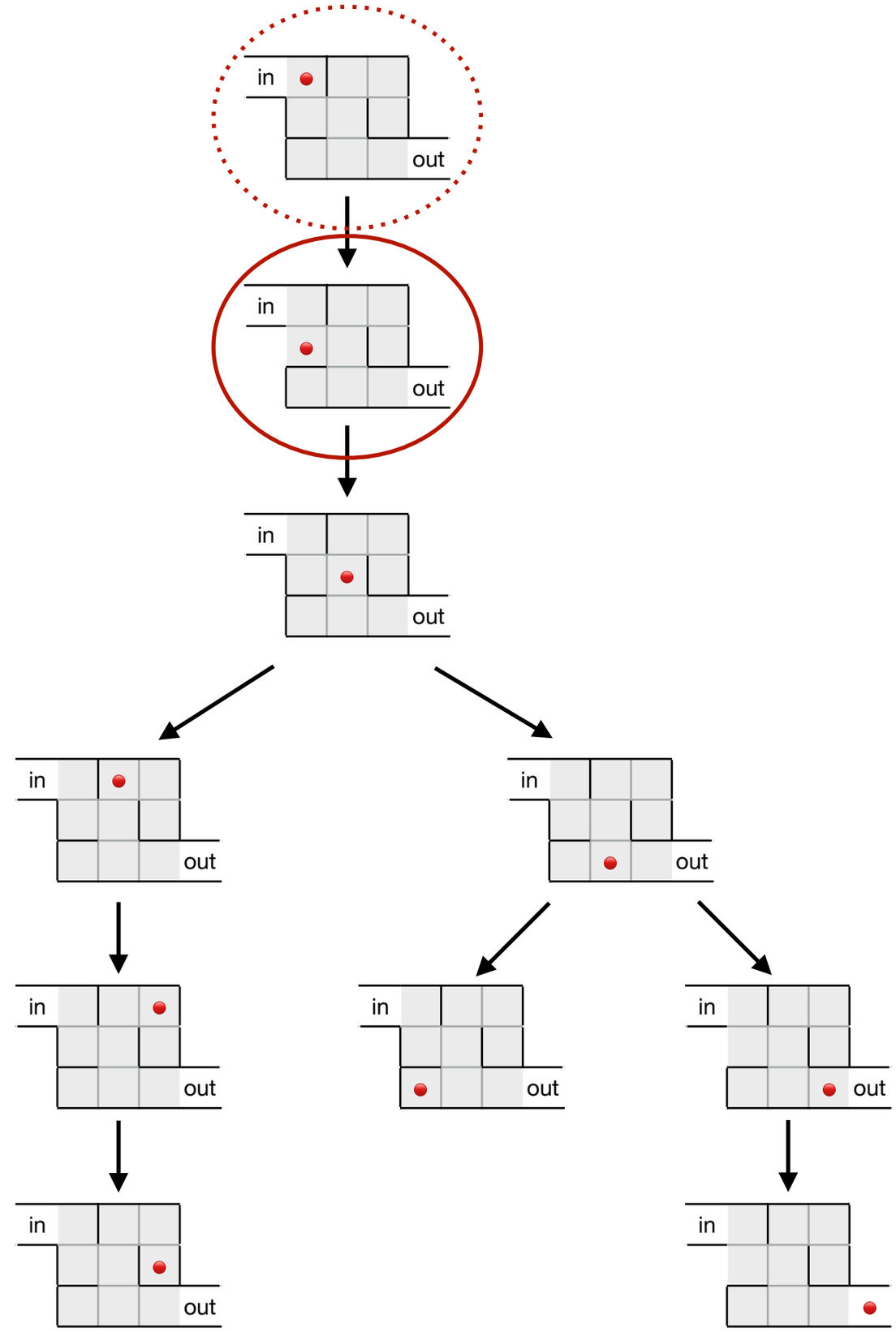
DFS algorithm

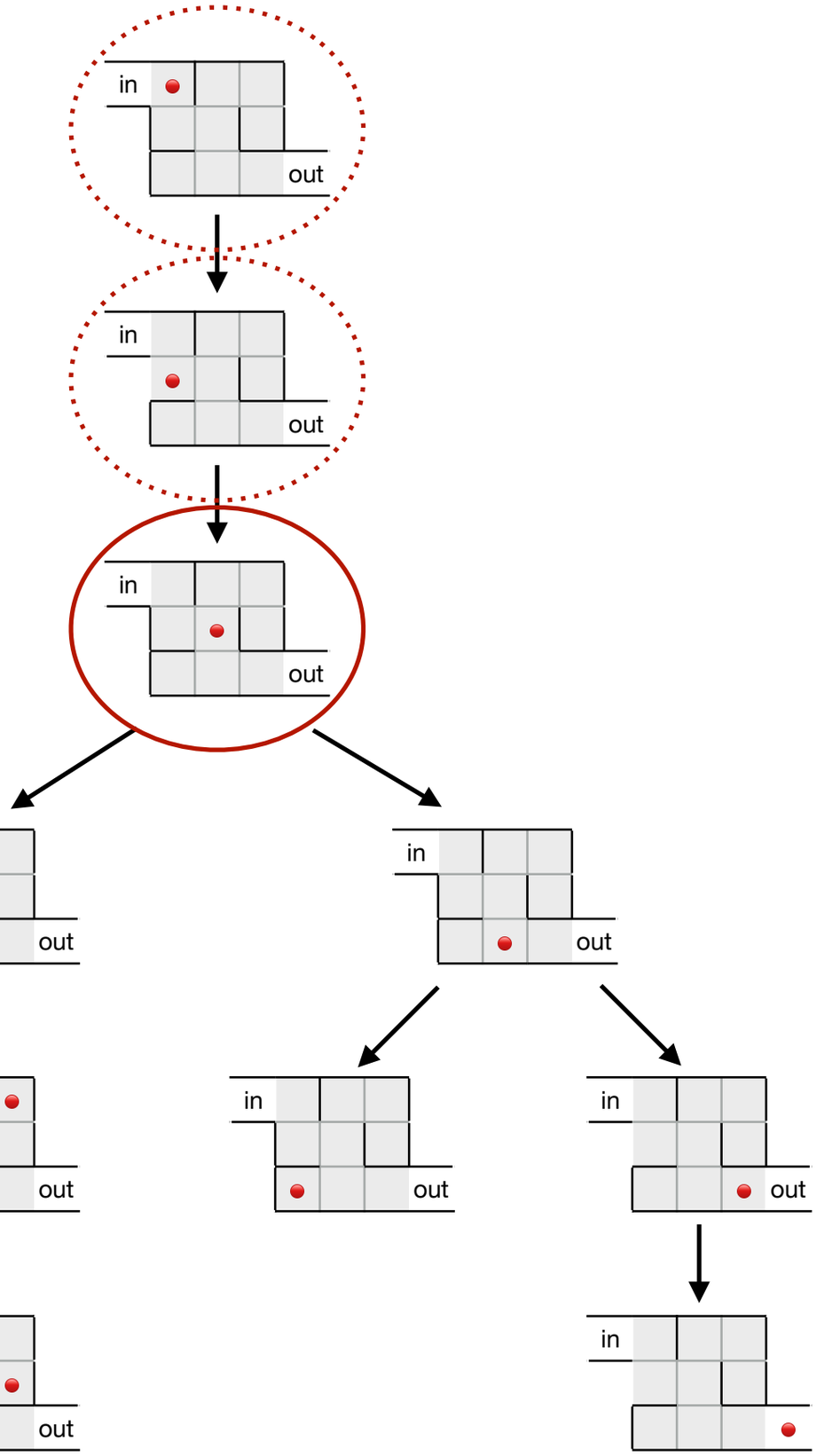
```
1   def dfs(graph, start):
2       visited, stack = set(), [start]
3       while stack:
4           vertex = stack.pop()
5           if vertex not in visited:
6               visited.add(vertex)
7               stack.extend(graph[vertex] - visited)
8               print(vertex)
```

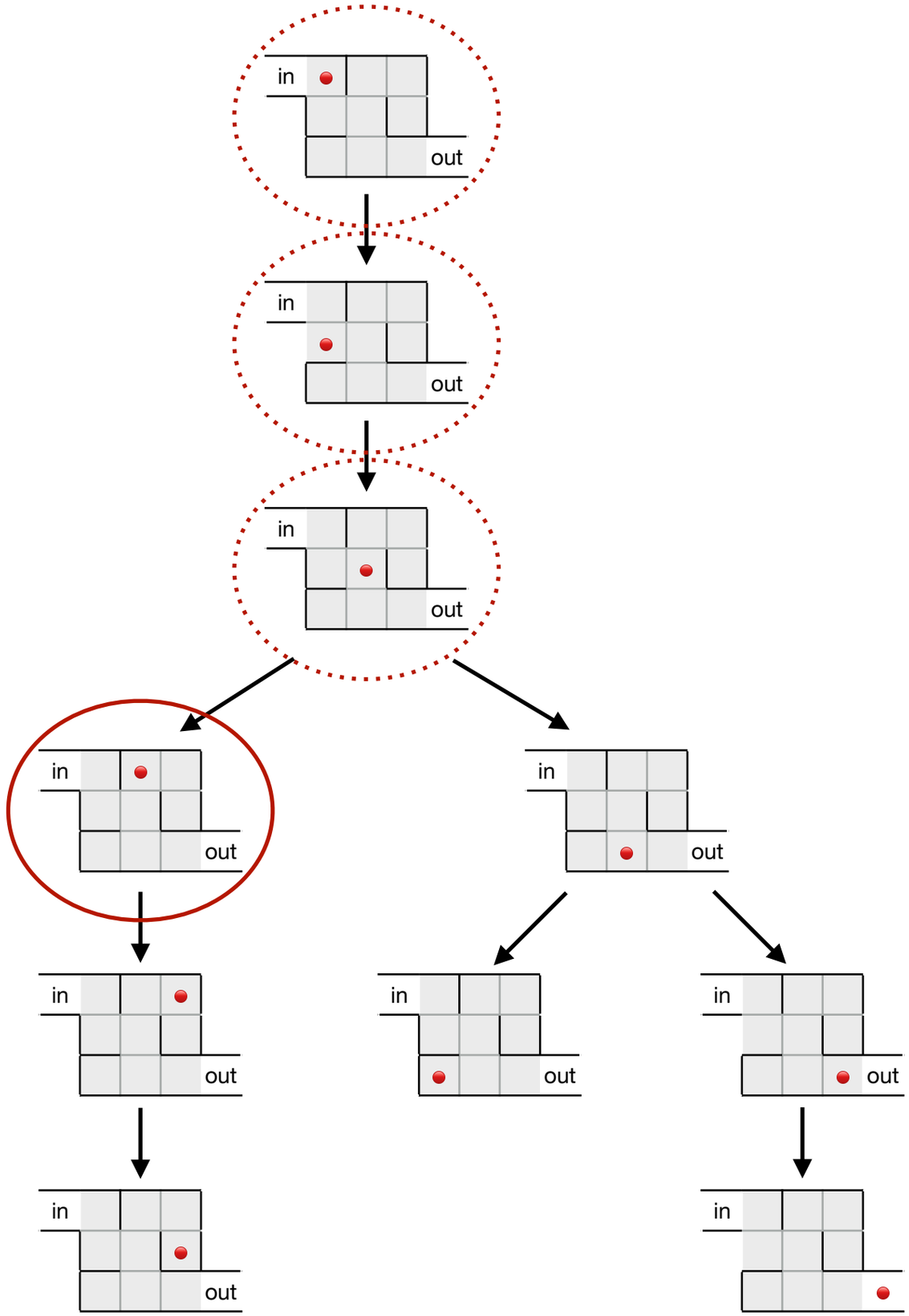
```
>>> dfs(maze,'in')
```

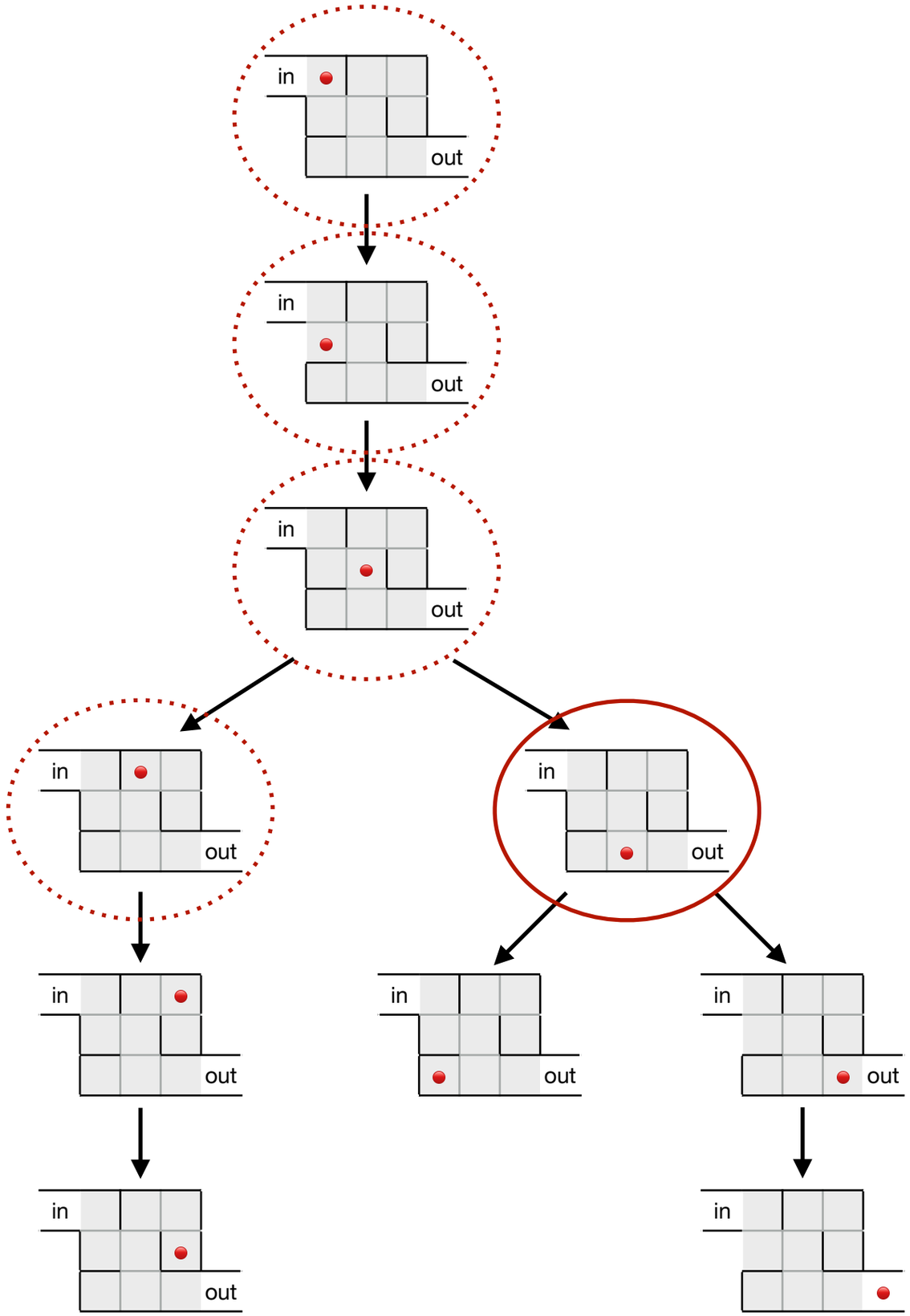

Breadth-first search

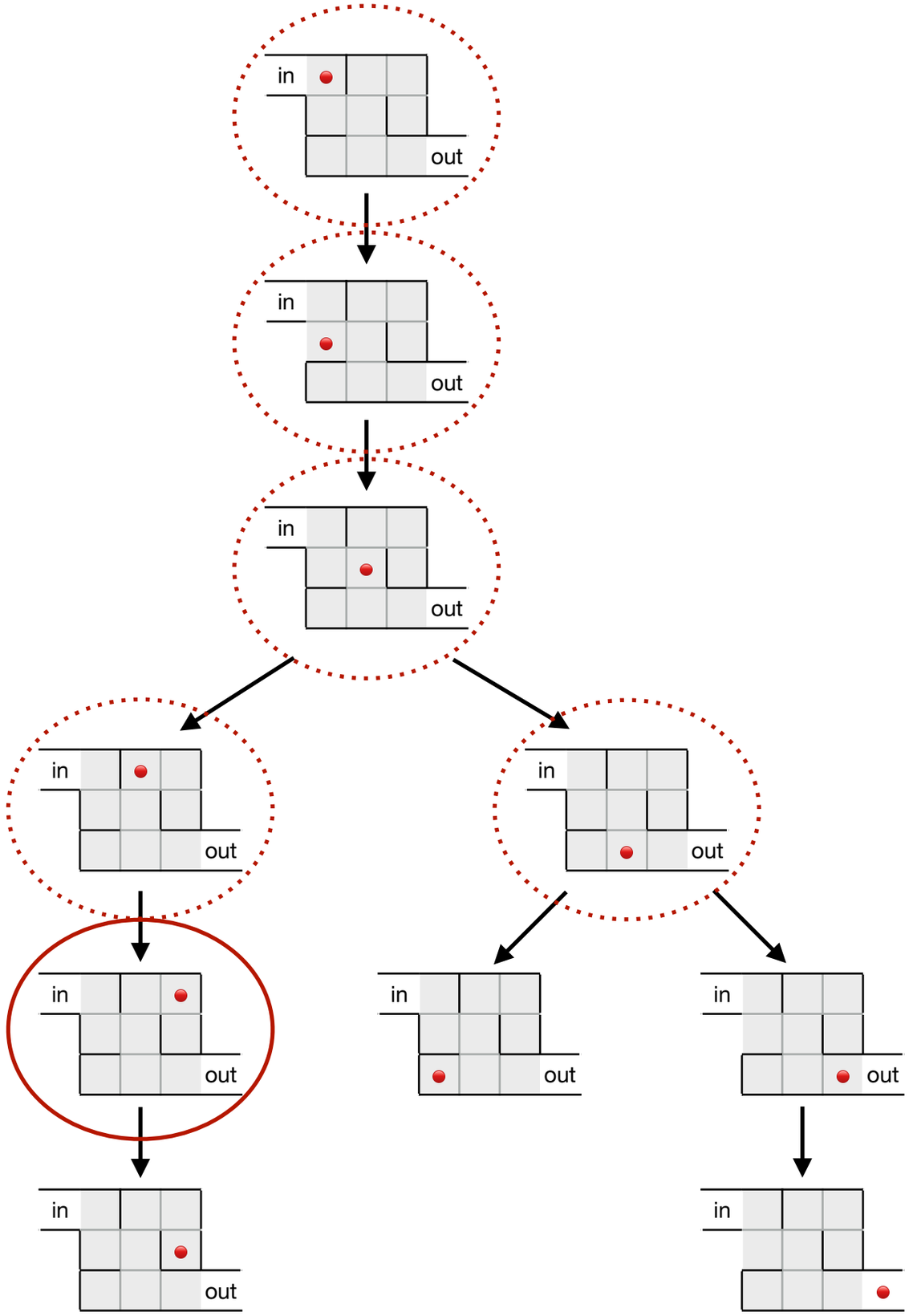


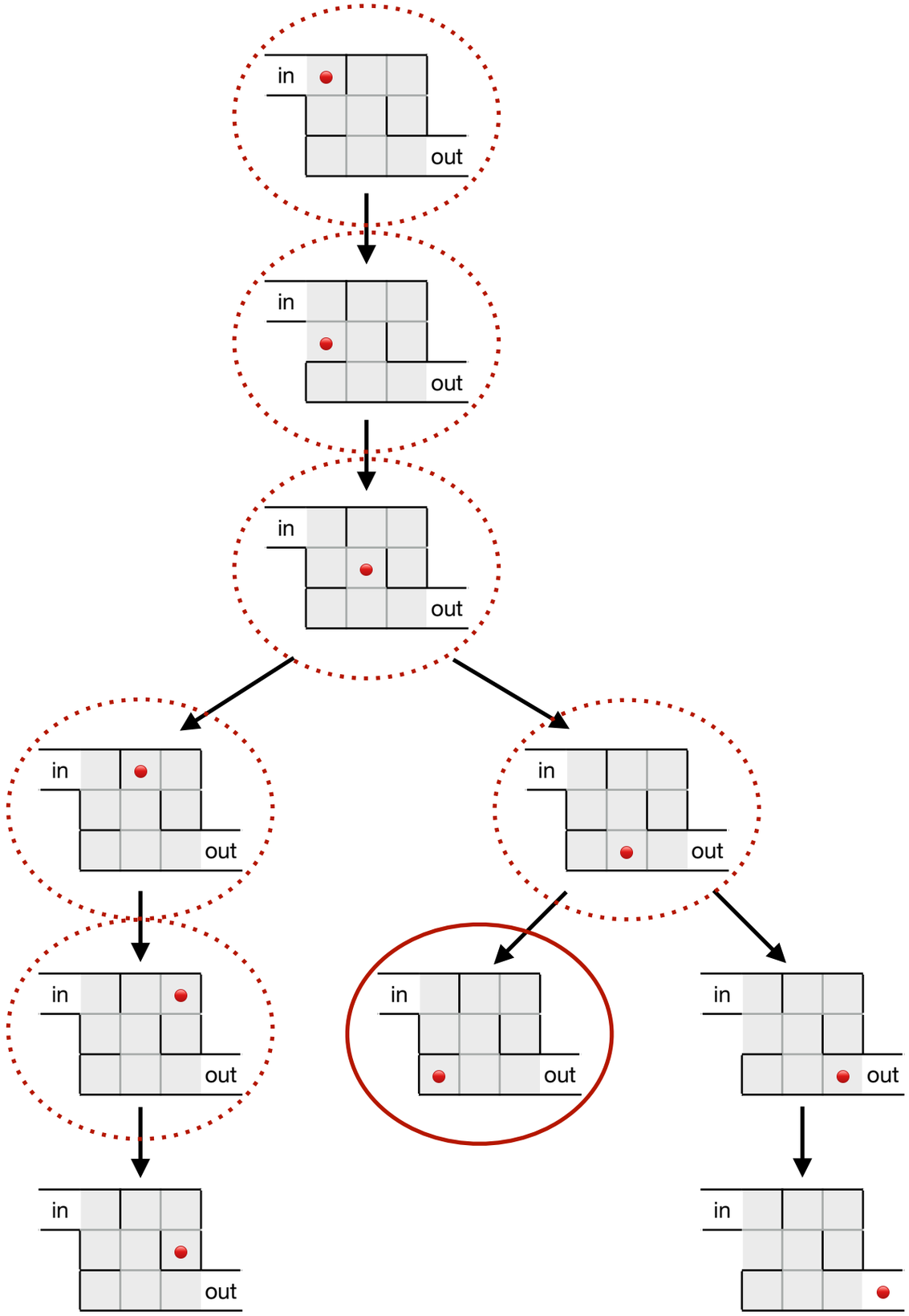


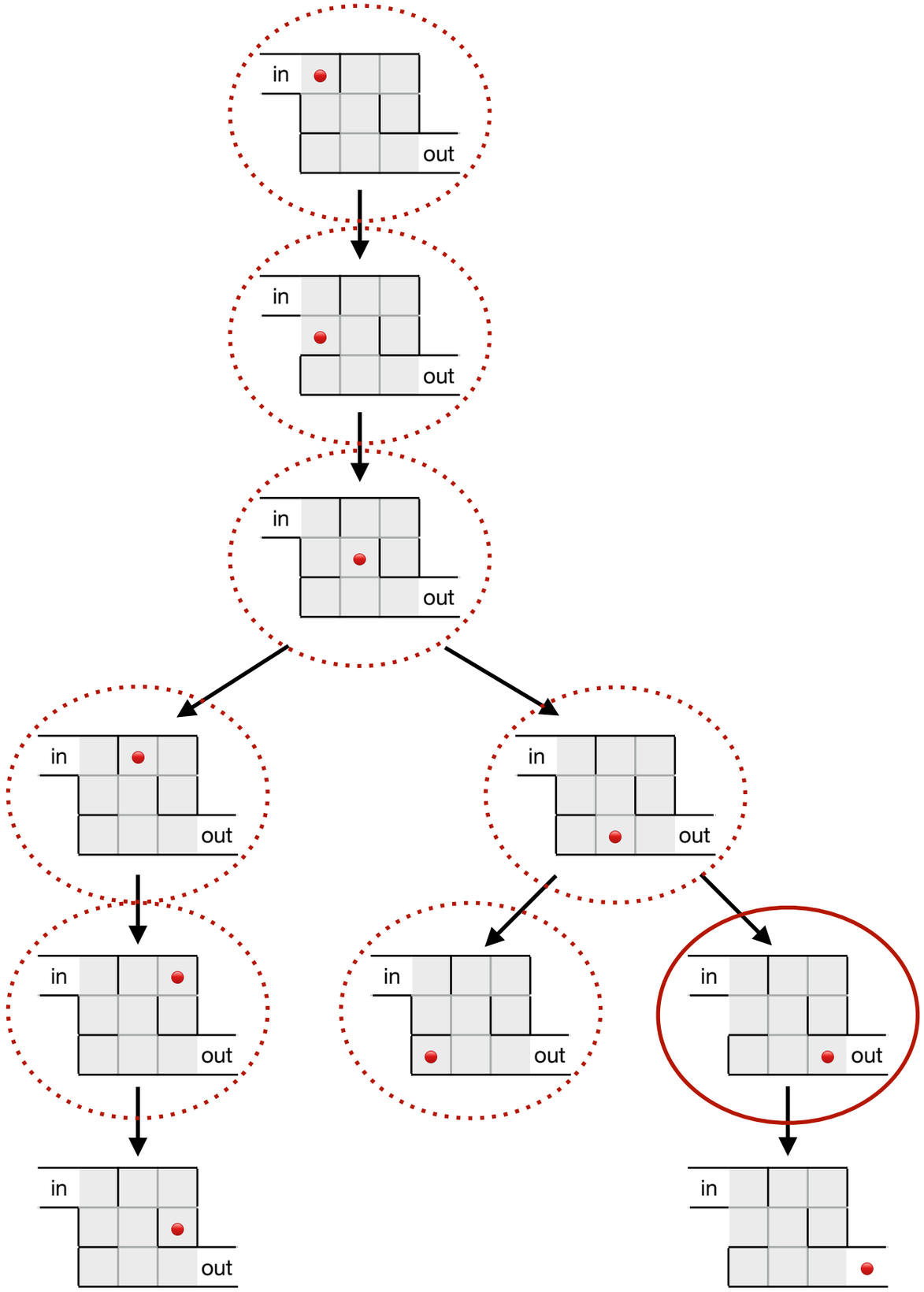


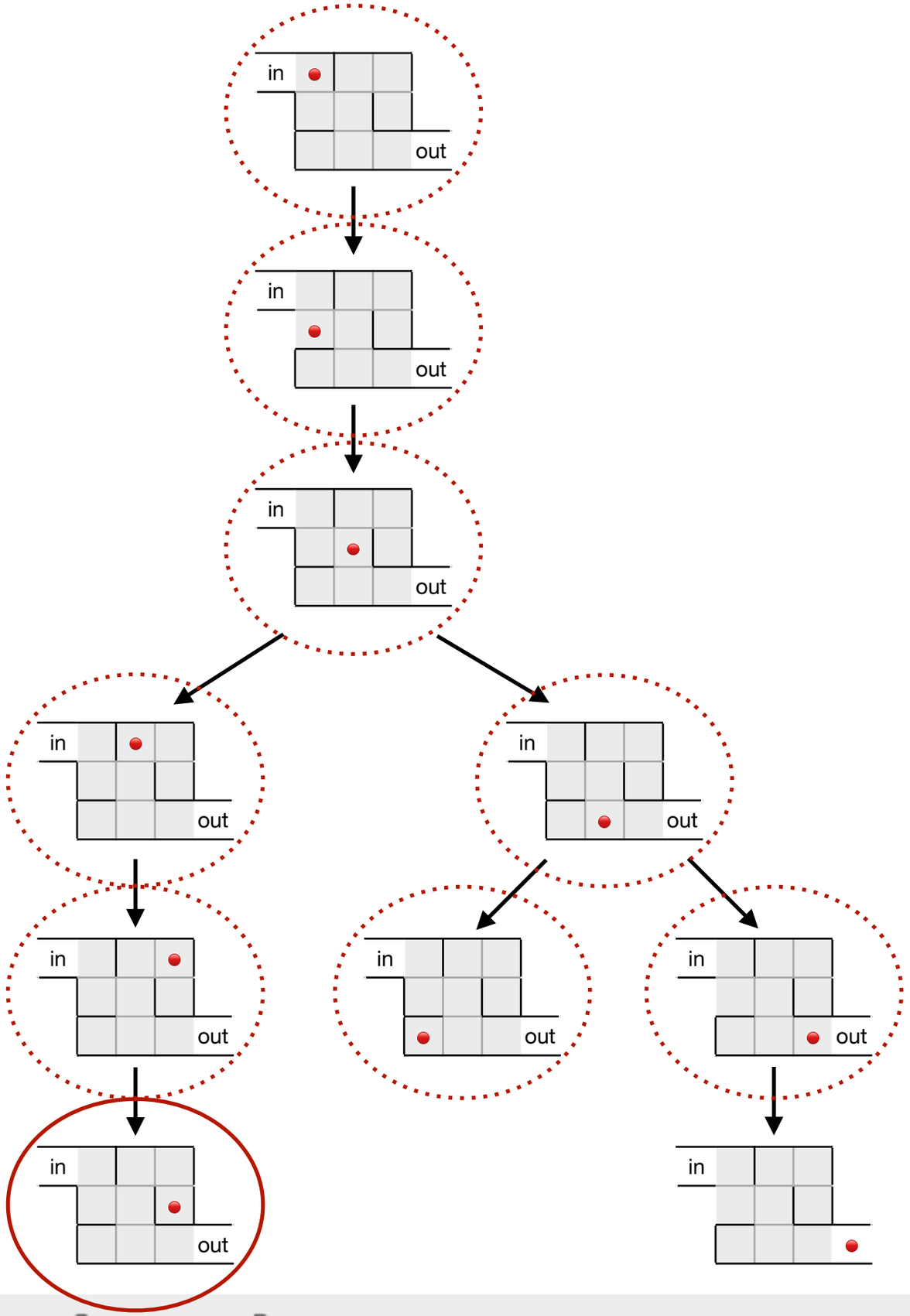


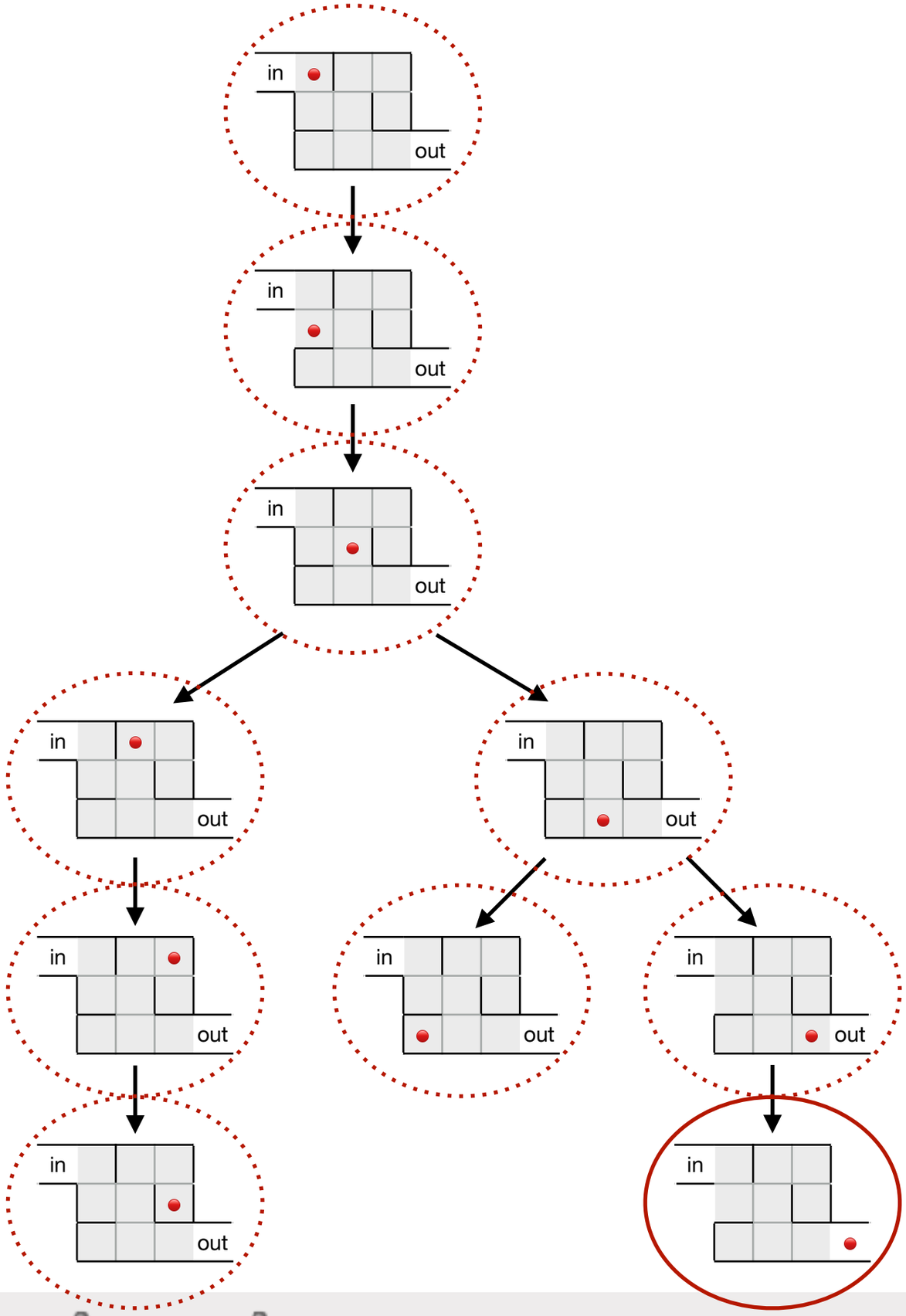












BFS algorithm

BFS algorithm

BFS: given a graph **G** and a starting node **n**

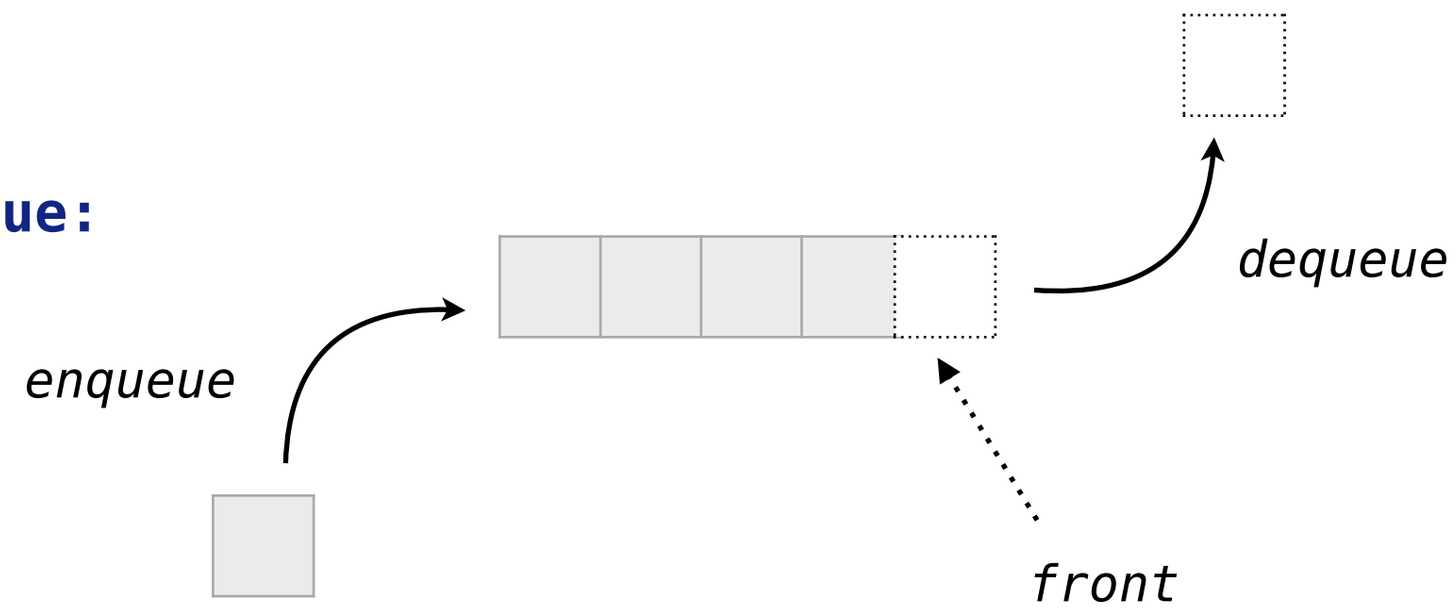
0. put **n** into a *queue*
1. *dequeue first node* from the queue and mark it as *visited*
2. put all nodes reachable from the first node in the *queue*
3. while the queue is not empty, recursively apply steps 1–3

BFS algorithm

BFS: given a graph **G** and a starting node **n**

0. put **n** into a *queue*
1. *dequeue first node* from the queue and mark it as *visited*
2. put all nodes reachable from the first node in the *queue*
3. while the queue is not empty, recursively apply steps 1–3

Queue:



FIFO:
first in,
first out

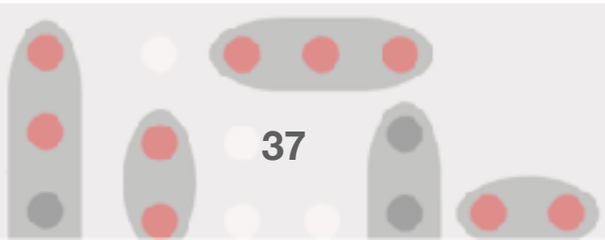
BFS algorithm

```
1   def bfs(graph, start):
2       visited, queue = set(), [start]
3       while queue:
4           vertex = queue.pop(0)
5           if vertex not in visited:
6               visited.add(vertex)
7
8           print(vertex)
```


BFS algorithm

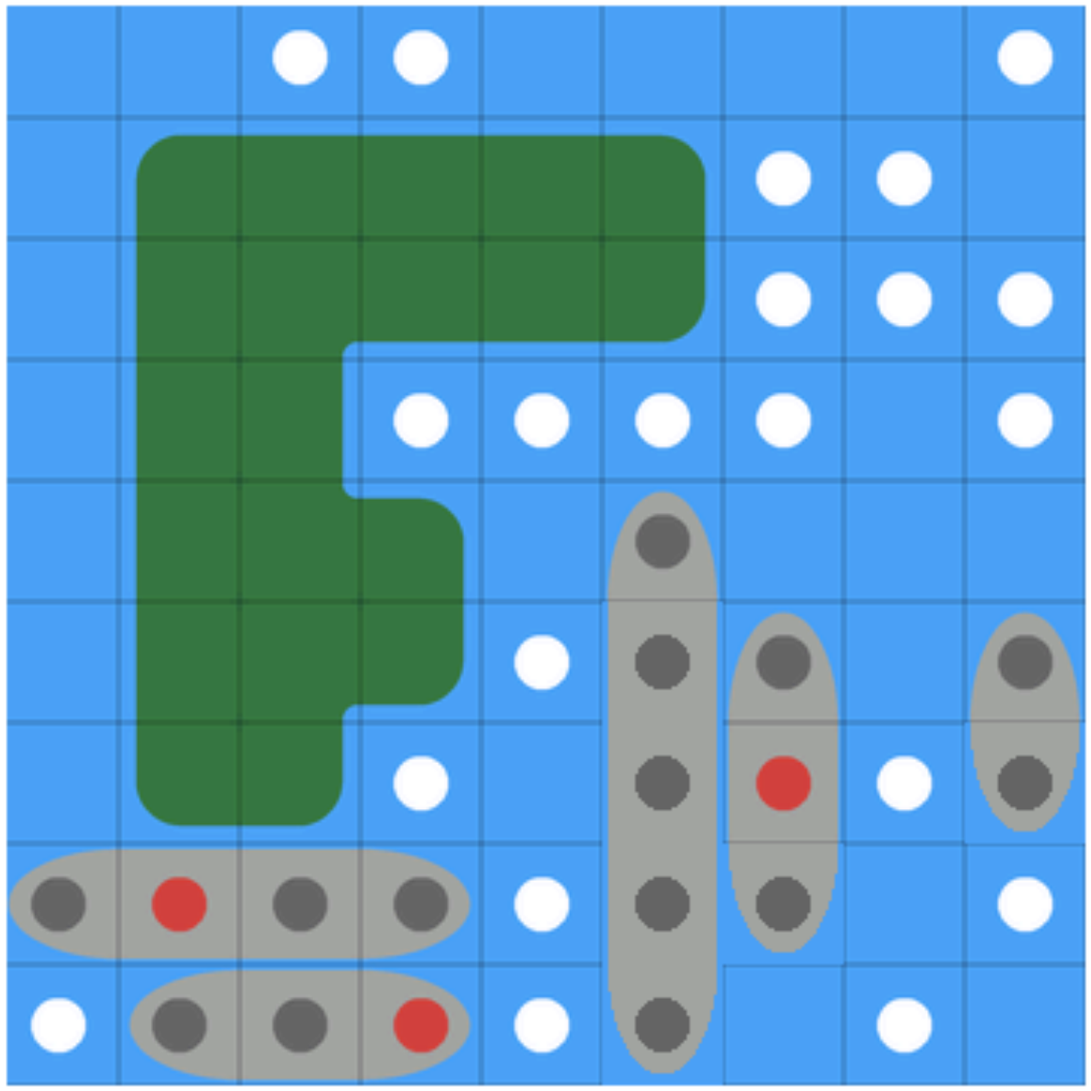
```
1   def bfs(graph, start):
2       visited, queue = set(), [start]
3       while queue:
4           vertex = queue.pop(0)
5           if vertex not in visited:
6               visited.add(vertex)
7               queue.extend(sorted(graph[vertex] - visited))
8           print(vertex)
```

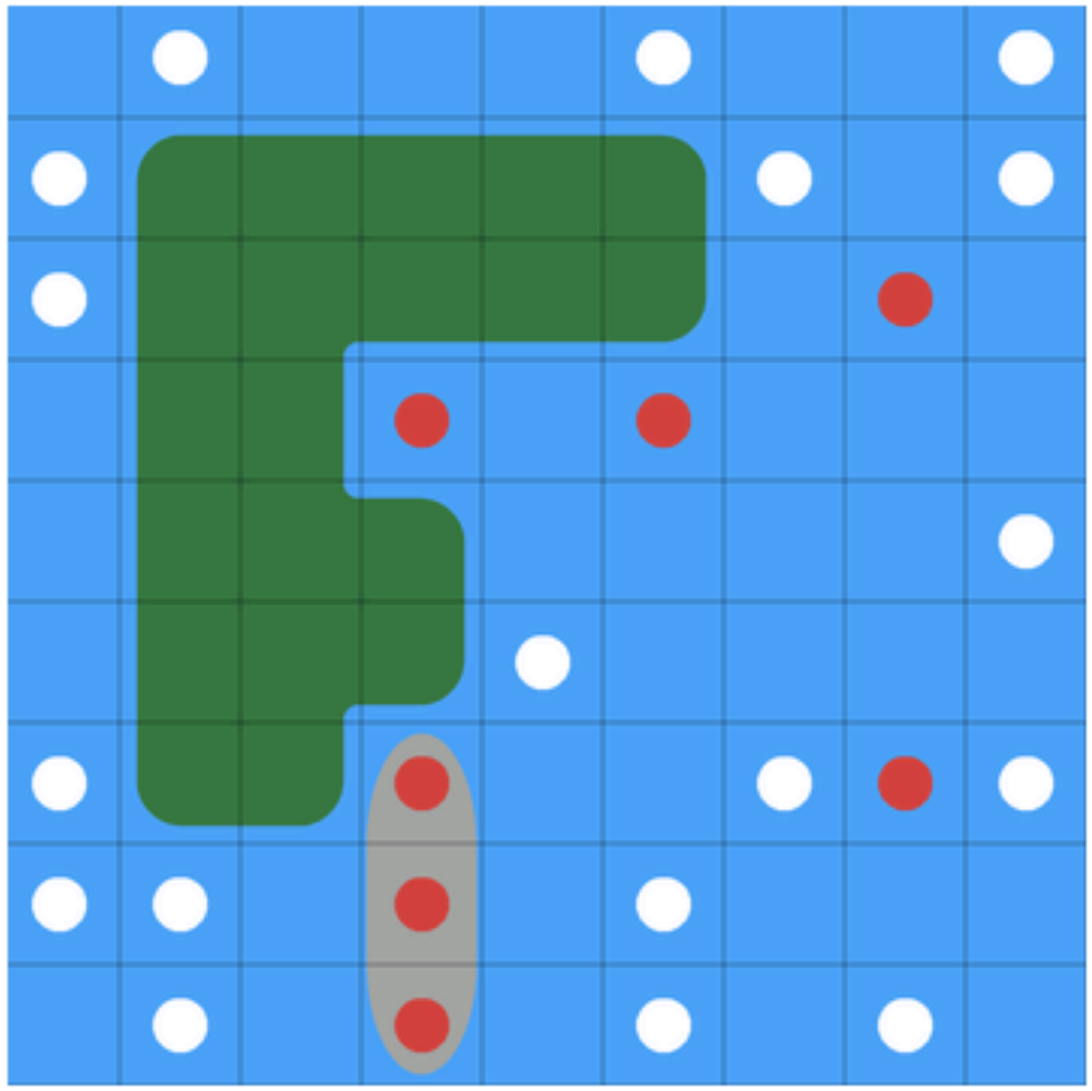
Search in Battleships



BATTLESHIPS

Battleships is an excellent, mid level introduction to developing game playing bots. More complicated than Noughts and Crosses, but less of a challenge than Texas Hold 'Em, Battleships strategy is more easily defined, and, ultimately, easier to implement.





Battleships

```
29 def calculateMove(gameState):
30     if "handCount" not in persistentData:
31         persistentData["handCount"] = 0
32     if gameState["Round"] == 0:
33         #move = exampleShipPlacement()  # Does not take land into account
34         move = deployRandomly(gameState)
35     else:
36         persistentData["handCount"] += 1
37         move = chooseRandomValidTarget(gameState)
38     print(str(persistentData["handCount"]) + '. MOVE: ' + str(move))
39     return move
```

>>> print(‘Game state: ’ + str(gameState))

Target vs Hunt mode

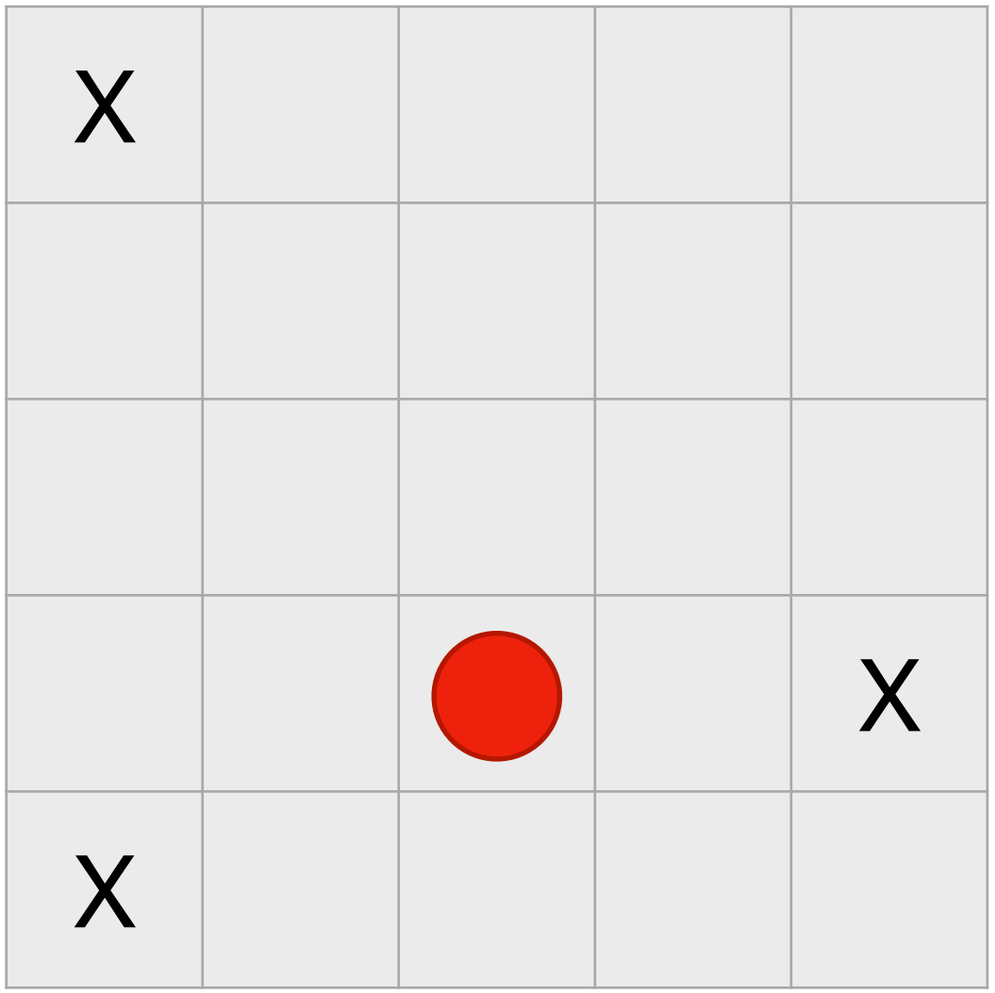
- in the *Hunt* mode, we randomly search for ships on the board:

  chooseRandomValidTarget(gameState)

- in the *Target* mode, when a ship is hit, we try to sink it by searching through its neighbourhood cells!

  which type of search should we choose?

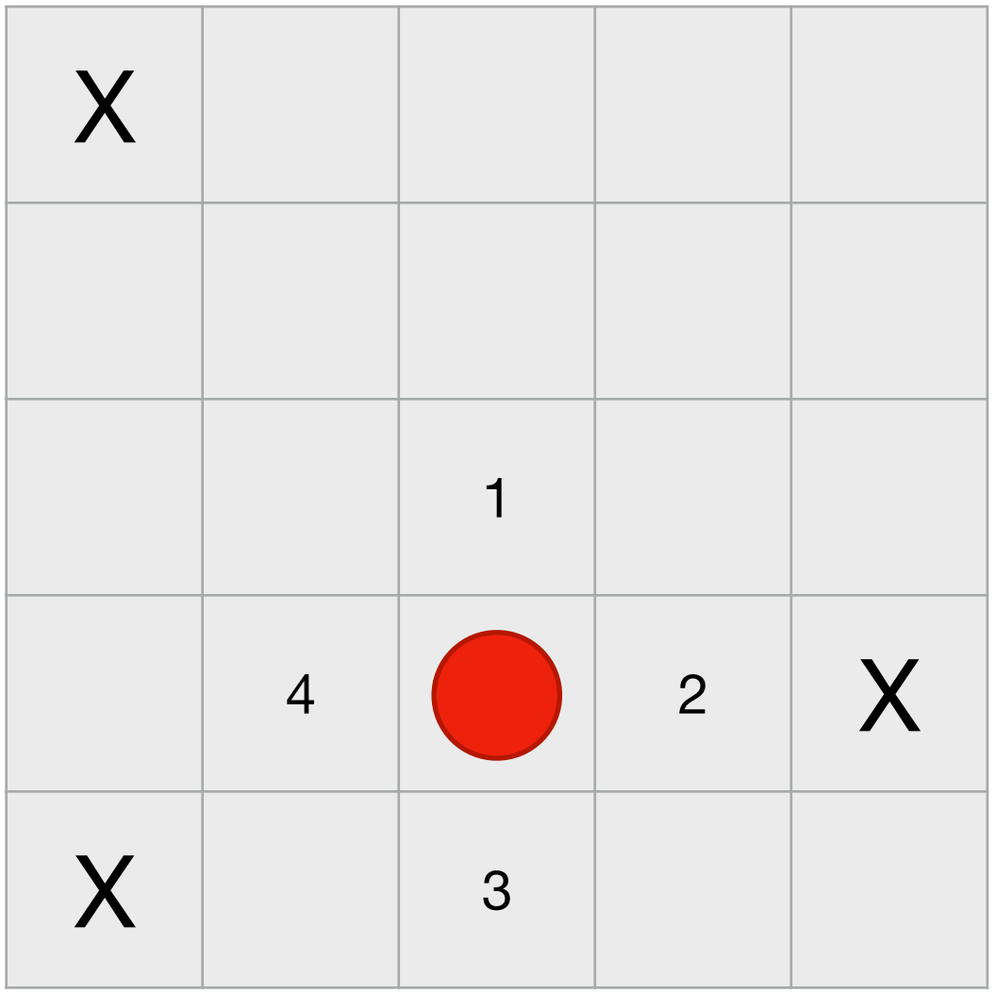
Search in Target mode



X is a missed shot,

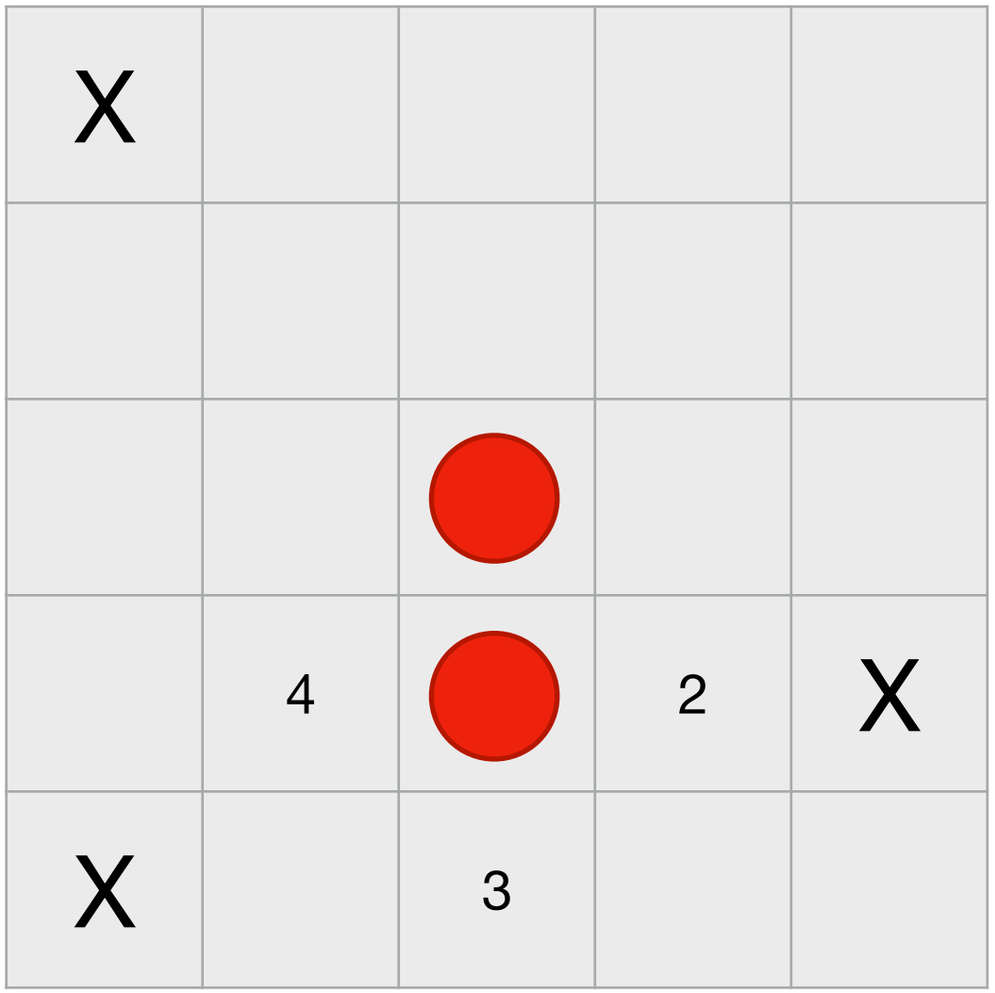
● is a hit ship.

Search in Target mode



BFS will first consider all neighbours of the start node, even if the next hit is discovered.

Search in Target mode



BFS will first consider all neighbours of the start node, even if the next hit is discovered.

Search in Target mode

X		3	4	5
		2		
		1		
				X
X				

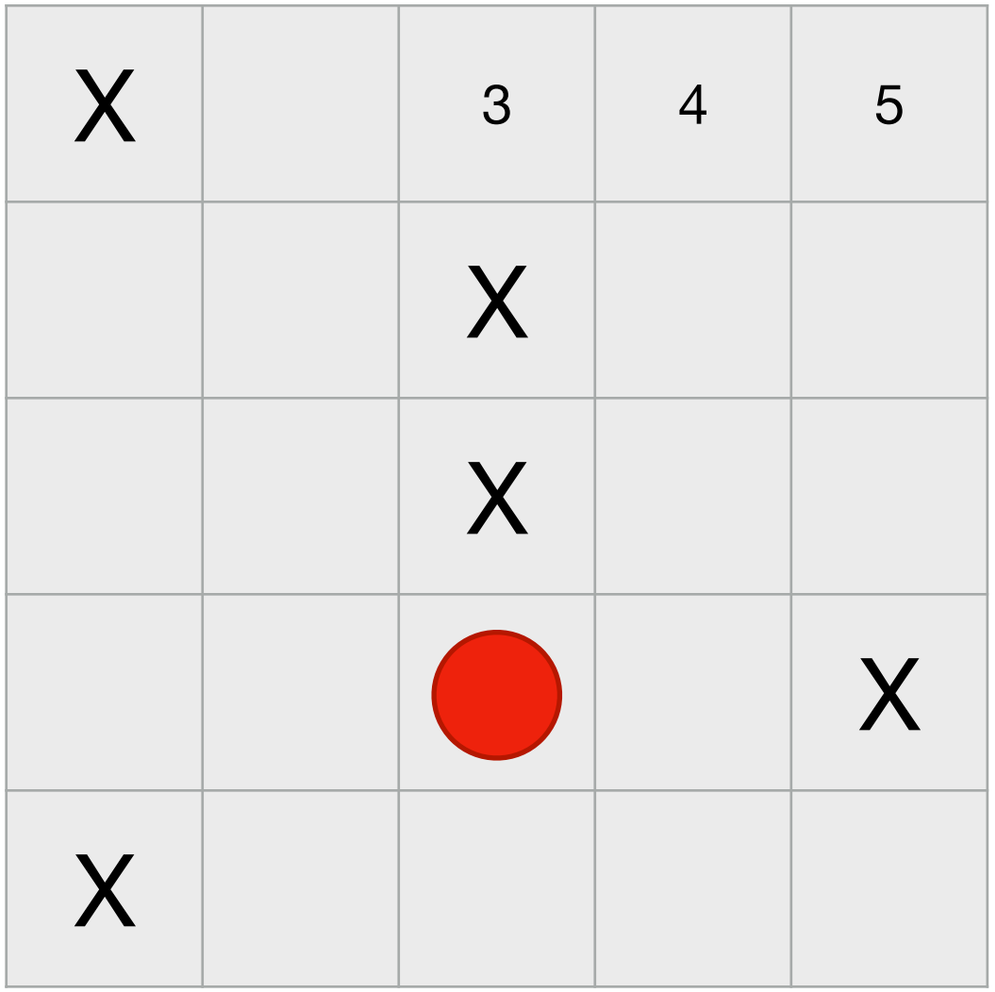
DFS, on the other hand, will follow a path from the start node, even if the next hit is not discovered.

Search in Target mode

X		3	4	5
		2		
		X		
				X
X				

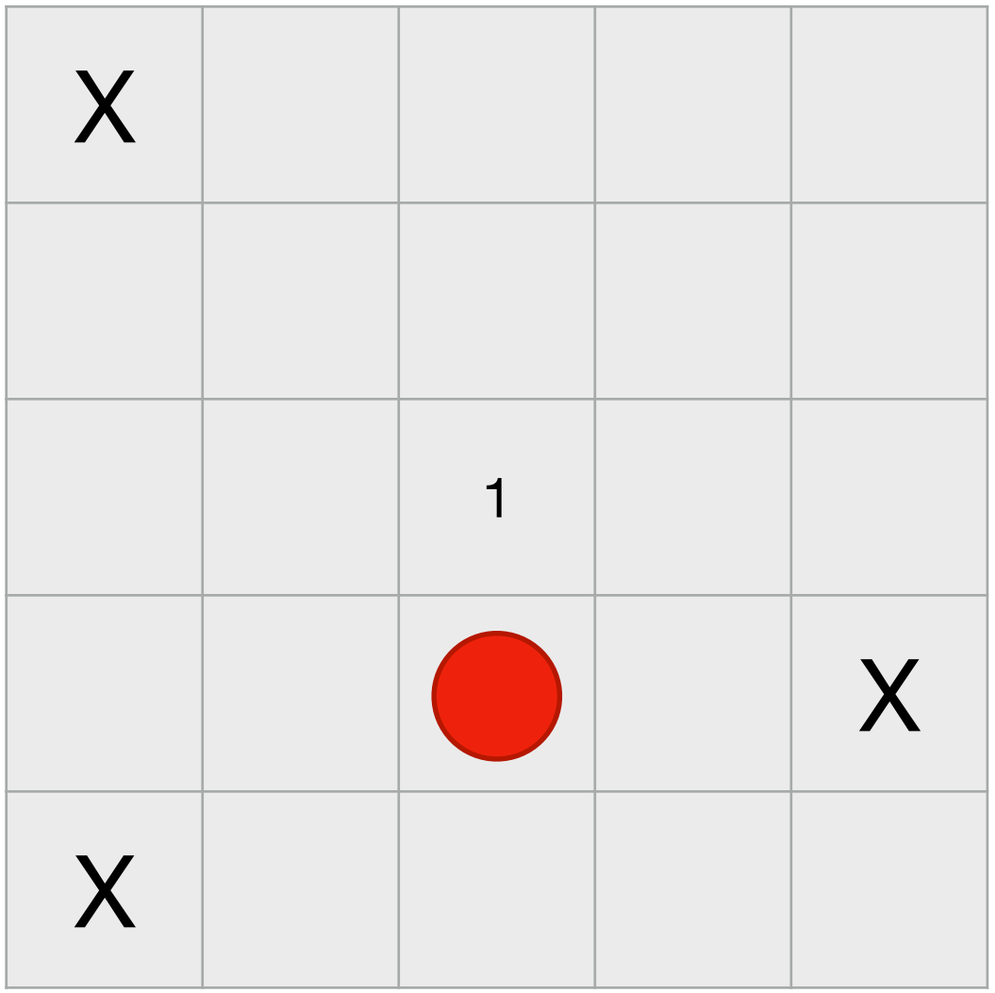
DFS, on the other hand, will follow a path from the start node, even if the next hit is not discovered.

Search in Target mode



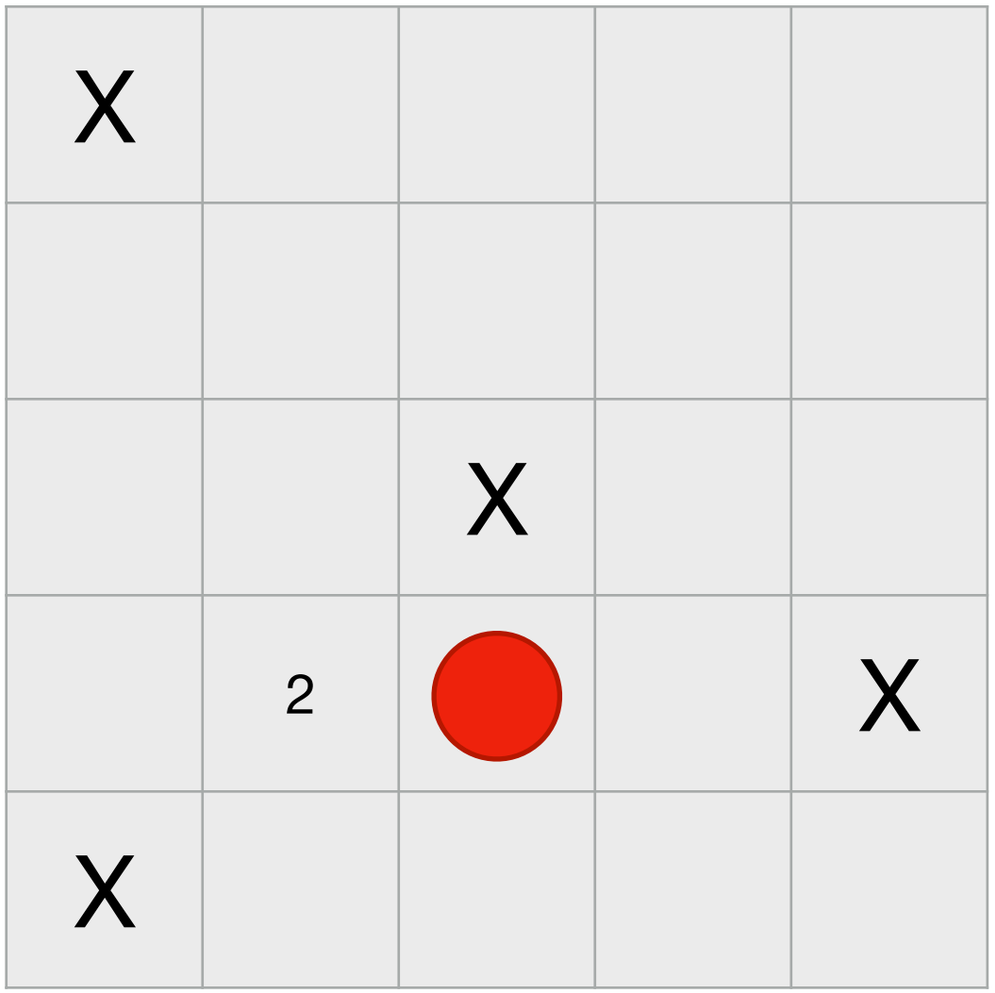
DFS, on the other hand, will follow a path from the start node, even if the next hit is not discovered.

Search in Target mode



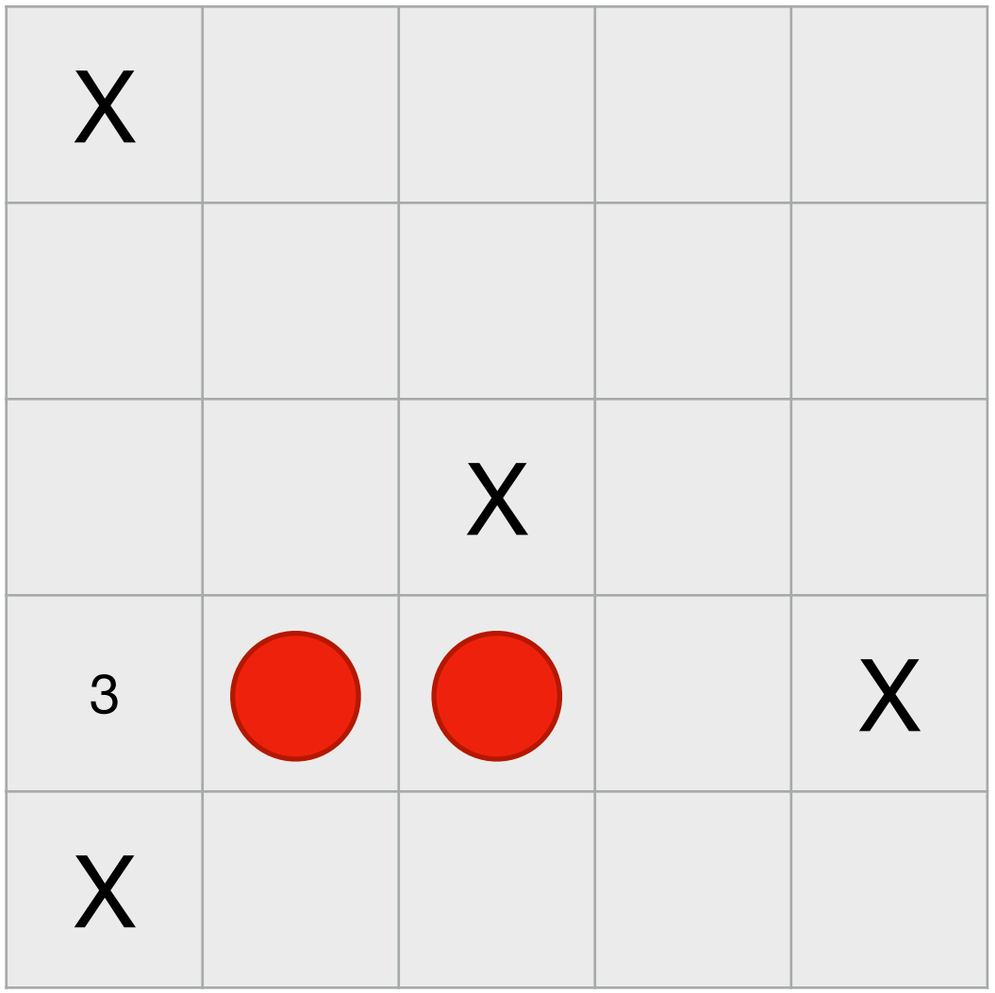
Solution: DFS with pruning
whenever the top node returns a miss, we do not add its neighbours to the stack.

Search in Target mode



Solution: DFS with pruning
whenever the top node returns a miss, we do not add its neighbours to the stack.

Search in Target mode



Solution: DFS with pruning
whenever the top node returns a miss, we do not add its neighbours to the stack.

Coding search strategies

The important part

- function *calculateMove*

```
4
5   def calculateMove(gamestate):
6       if gamestate["Round"] == 0:  # If we are in the ship placement round
7           # move = exampleShipPlacement()  # Does not take land into account
8           move = deployRandomly(gamestate)  # Randomly place your ships
9       else:  # If we are in the ship hunting round
10          move = chooseRandomValidTarget(gamestate)  # Randomly fire at valid sea targets
11      return move
```

Here you can modify your moves strategy and *search*!

Homework: integrate DFS with pruning strategy into the battleships code