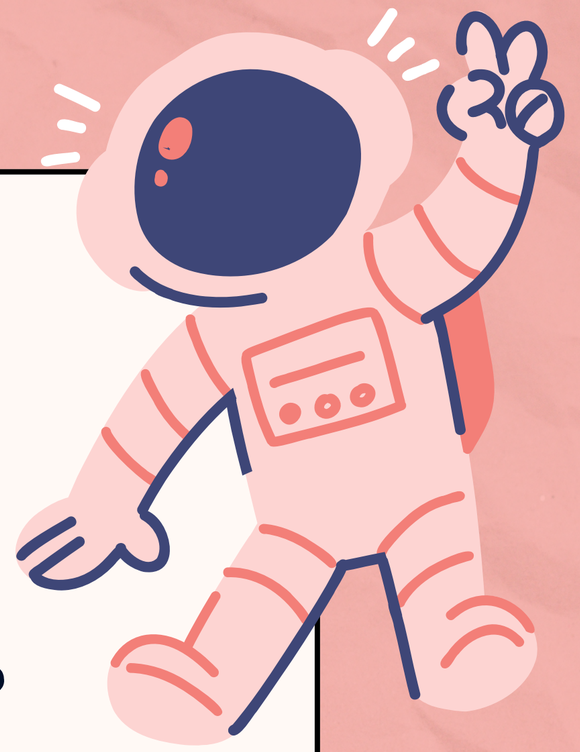
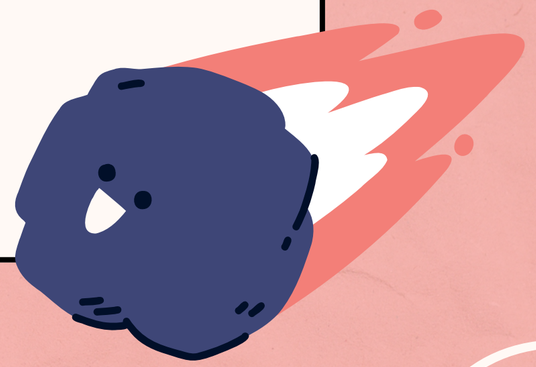
CSCI 111

8-PUZZLE

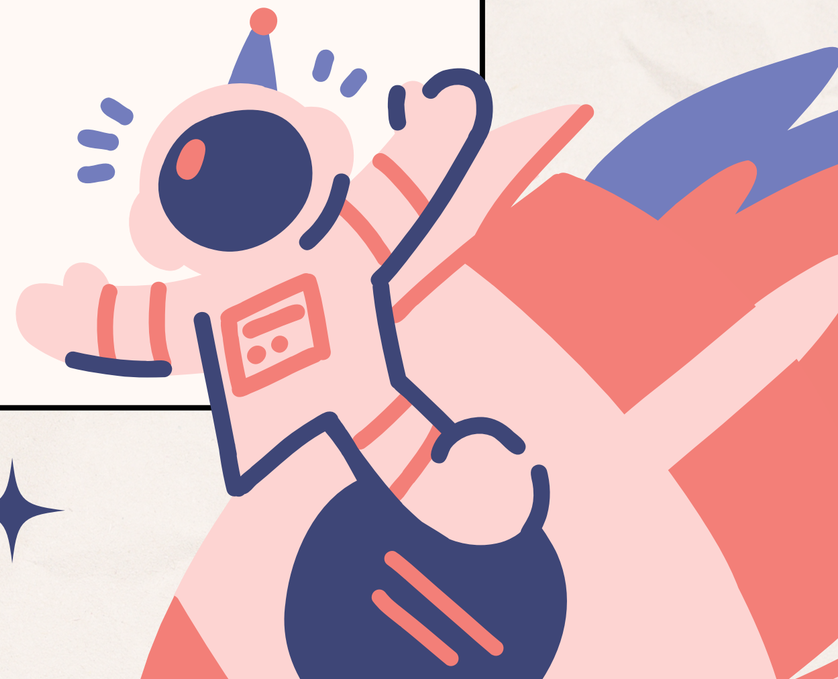
BENJAMIN,
ELLYN, JEREMY





WHAT IS 8-PUZZLE

Let's Start!

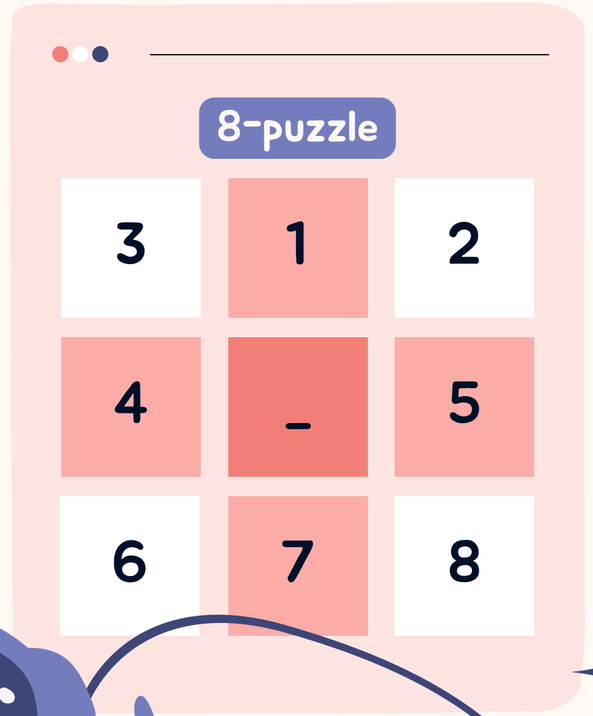


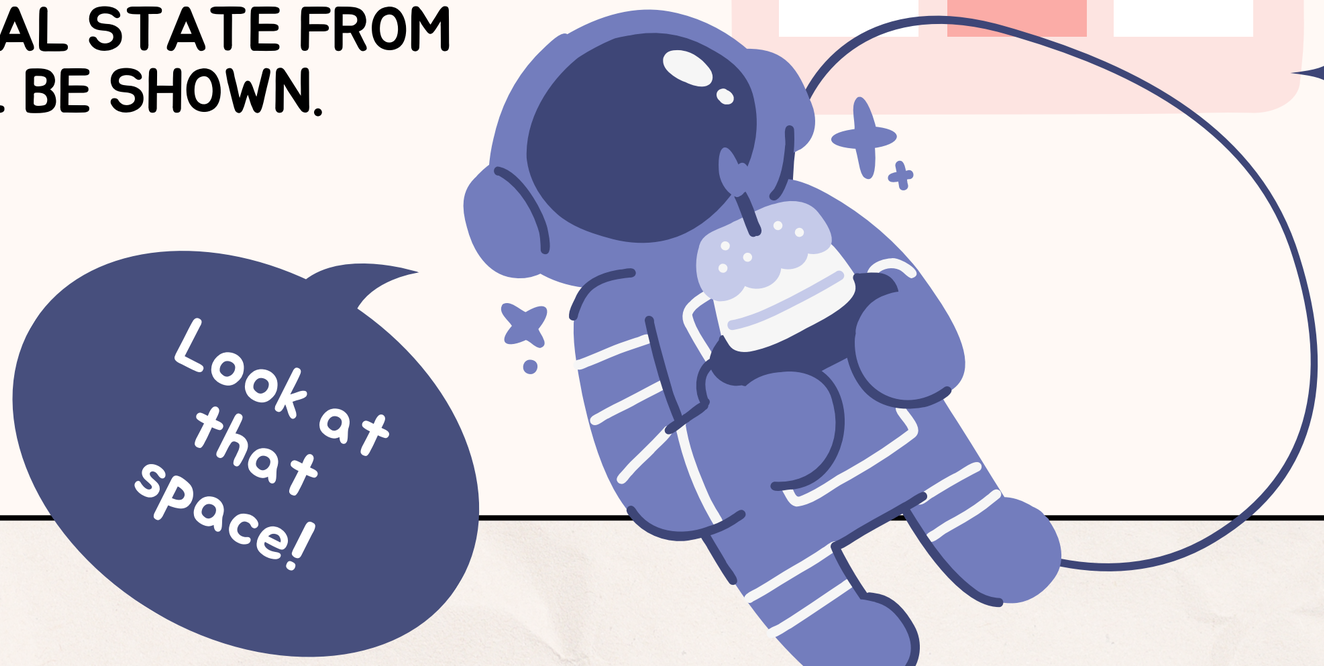
8-puzzle

WE HAVE A 3X3 GRID WITH 8 TILES, NUMBERED 1 TO 8, AND A BLANK SPACE.

WITH THIS, WE ARE ONLY ABLE TO MOVE ADJACENT TILES INTO THE BLANK SPACE.

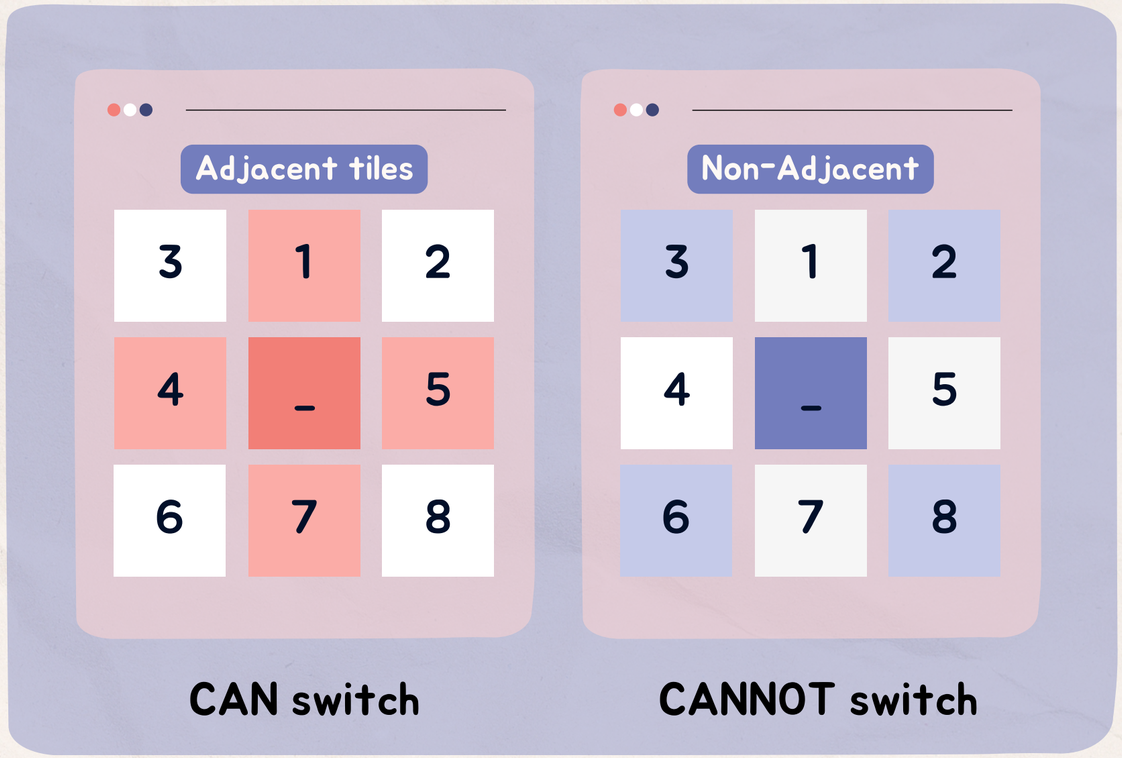
OUR GOAL IS TO REACH THE GOAL STATE FROM THE INITIAL STATE WHICH WILL BE SHOWN.





Rules

- CAN ONLY SWITCH THE BLANK TILE AND ITS ADJACENT TILE
- CANNOT SKIP TILES
- CANNOT MOVE DIAGONALLY



Example 1

_	1	2
3	4	5
6	7	8

Example 1

_	1	2
3	4	5
6	7	8

Example 1

_	1	2
3	4	5
6	7	8

Example 2

3	1	2
_	4	5
6	7	8

Example 3

3	1	2
4	_	5
6	7	8

THE GOAL STATE

Goal 1

_	1	2
3	4	5
6	7	8

CAN BE SET BY USING THE GOAL STATE PART OF THE CODE. THIS IS THE EXAMPLE THAT WE ARE USING.

THE GOAL STATE

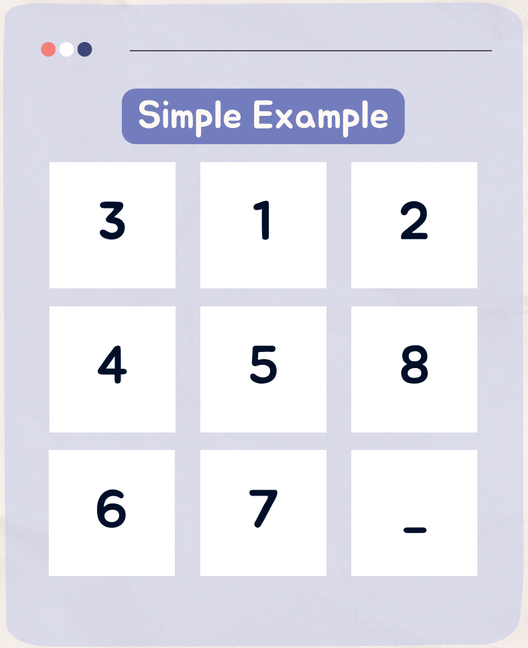

```
goal_state = np.array([["_", 1,2], [3,4,5], [6,7,8]])
goal_state
```

```
array([['_', '1', '2'],
       ['3', '4', '5'],
       ['6', '7', '8']], dtype='<U21')
```

CAN BE SET BY USING THE GOAL STATE PART OF THE CODE. THIS IS THE EXAMPLE THAT WE ARE USING.

THE INITIAL STATE

Simple Example



```
initial_list = np.array([[3,1,2], [4, 5,8], [6,7,"_"]])
def print_board(initial_list):
  for row in initial_list:
    print(f"{row[0]} | {row[1]} | {row[2]}")
  print("\n")
print_board(initial_list)
```

```
3 | 1 | 2
4 | 5 | 8
6 | 7 | _
```

WE DEFINE THE INITIAL STATE OF THE BOARD USING A NUMPY ARRAY

THE INITIAL STATE

Hard Example

Hard example

1	8	2
_	4	3
7	6	5

WE DEFINE THE INITIAL STATE OF THE BOARD USING A NUMPY ARRAY

```
initial_list = np.array([[1,8,2], ["_", 4,3], [7,6,5]])
def print_board(initial_list):
  for row in initial_list:
    print(f"{row[0]} | {row[1]} | {row[2]}")
  print("\n")
print_board(initial_list)
```

```
1 | 8 | 2
_ | 4 | 3
7 | 6 | 5
```

Types of SEARCHES

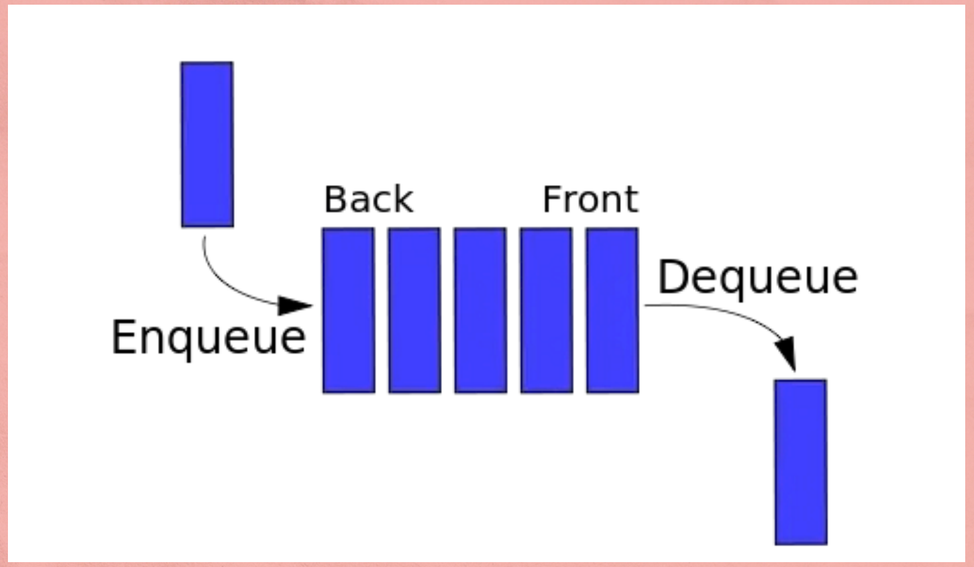
algorithms

SEARCH ALGORITHMS

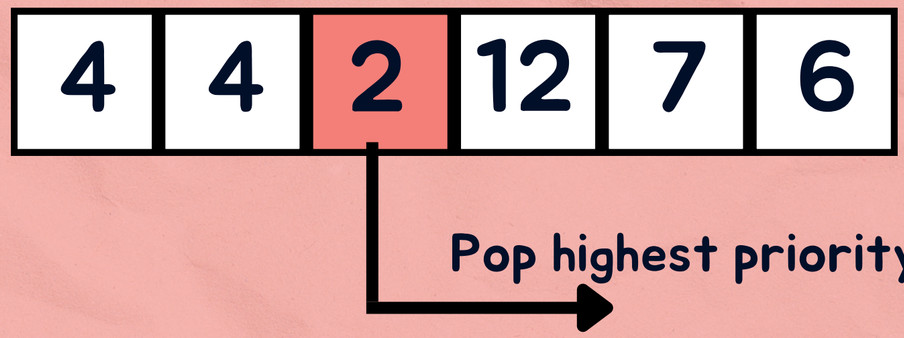
1. Breadth First Search
2. Greedy Best-First Search
3. A* Search

DATA STRUCTURES

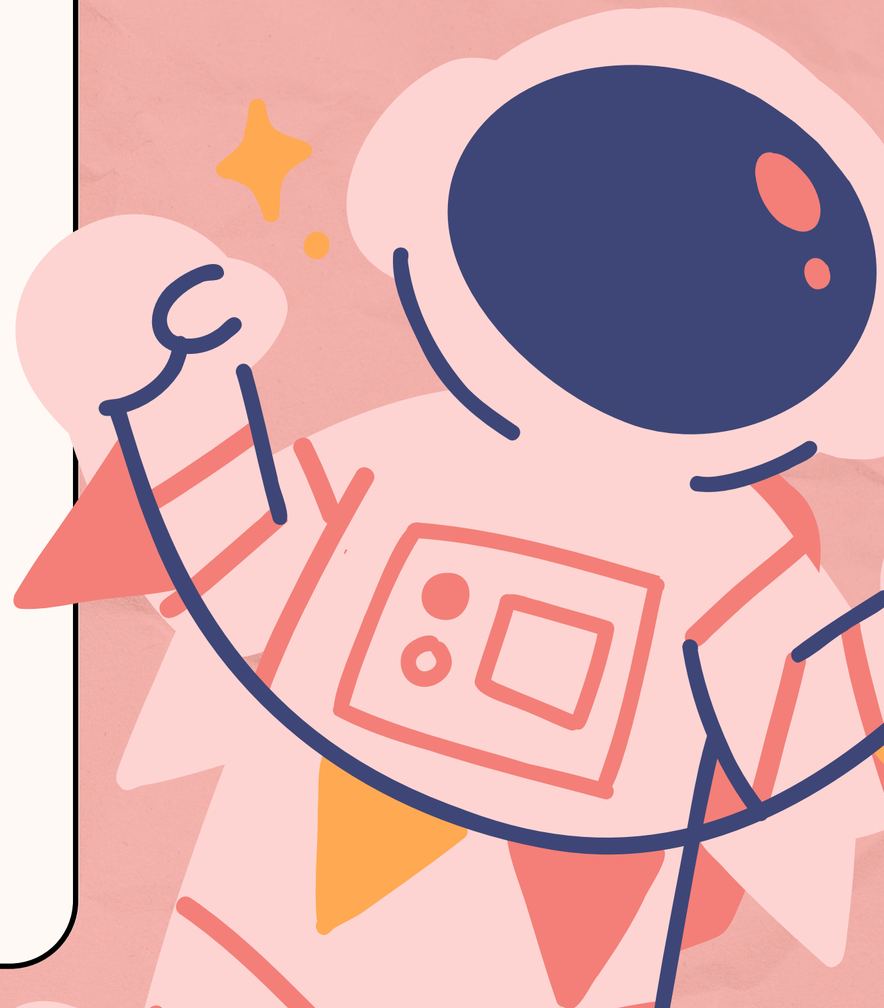
First In First Out (FIFO) queue



Priority Queue



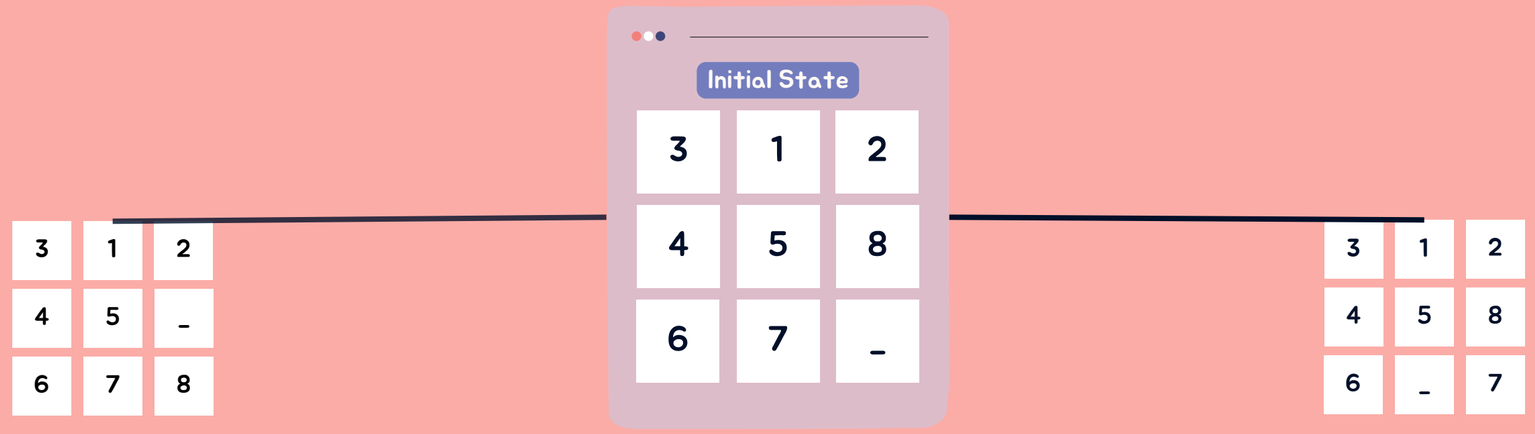
BREADTH FIRST SEARCH



The first type
BFS

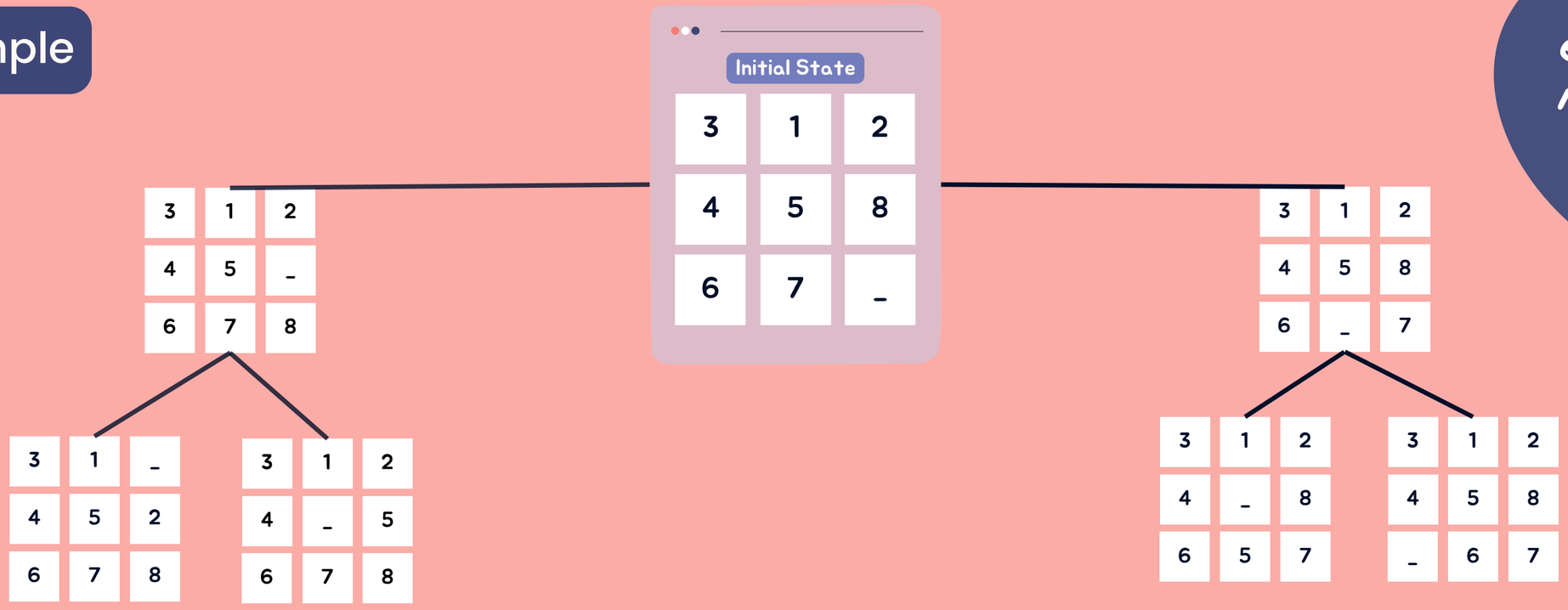
- AN UNINFORMED SEARCH STRATEGY WHERE ALL CHILDREN NODES ARE EXAMINED AT EACH LEVEL FROM THE INITIAL STATE OF THE BOARD.
- FOR THE 8-PUZZLE PROBLEM, WE HAVE A BRANCHING FACTOR EQUAL TO THE NUMBER OF TILES ADJACENT TO THE EMPTY TILE.
- IF THE TILE IS IN THE MIDDLE, WE HAVE 4 CHILDREN NODES. IF ITS IN THE EDGE, 3 CHILDREN NODES. IF ITS IN THE CORNER, WE HAVE 2 CHILDREN NODES.
- THE BFS SEARCH ALGORITHM WILL TAKE NOTE OF PAST ACTIONS TO AVOID REPETITIVE STATES.
- THE SEARCH ENDS AS SOON AS THE GOAL STATE APPEARS IN THE QUEUE.

Simple Example



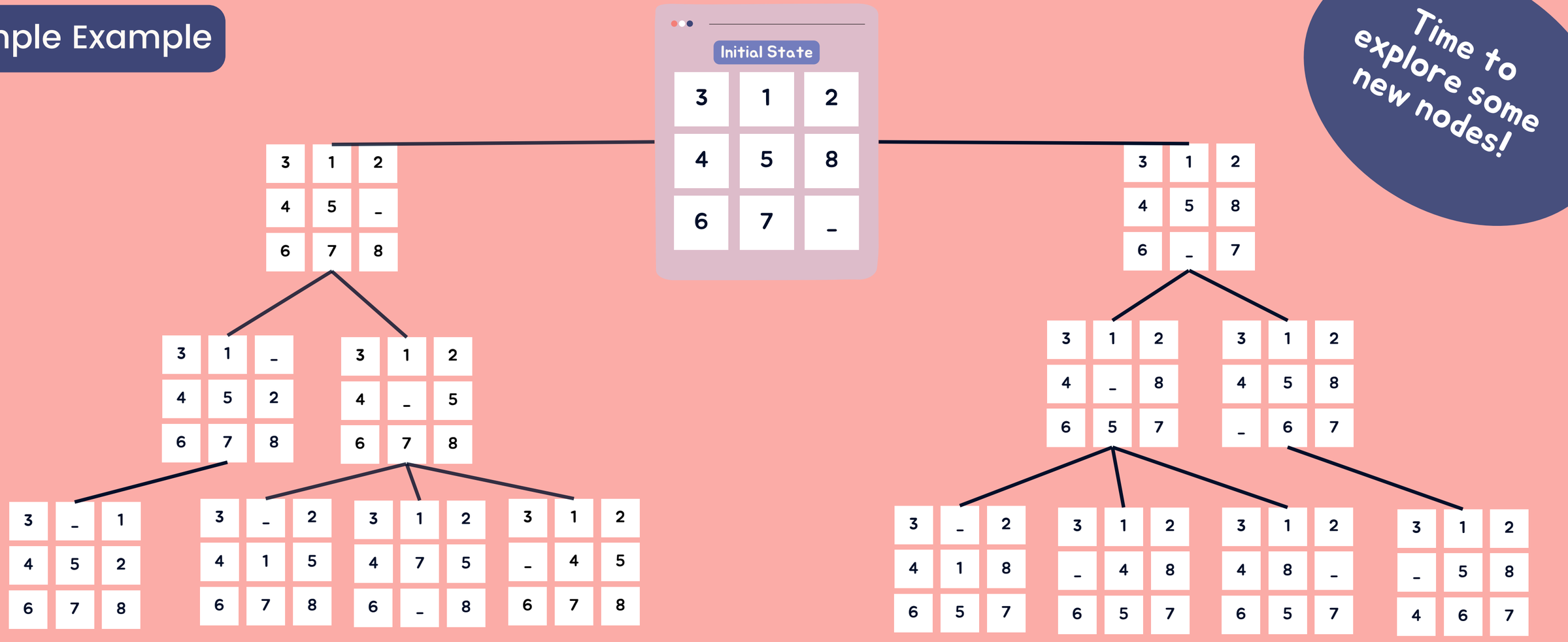
Time to explore some new nodes!

Simple Example

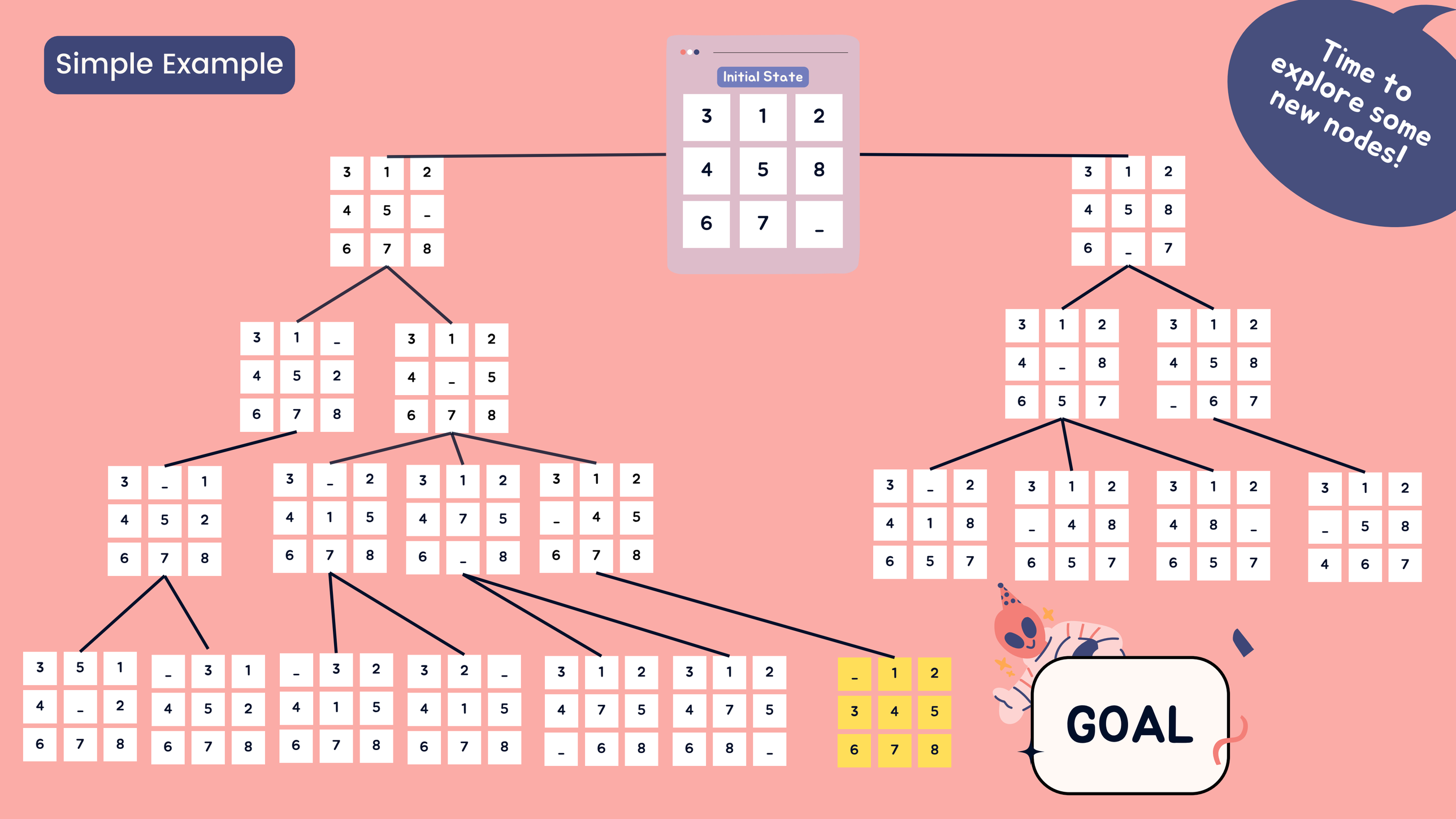


Time to explore some new nodes!

Simple Example



Simple Example

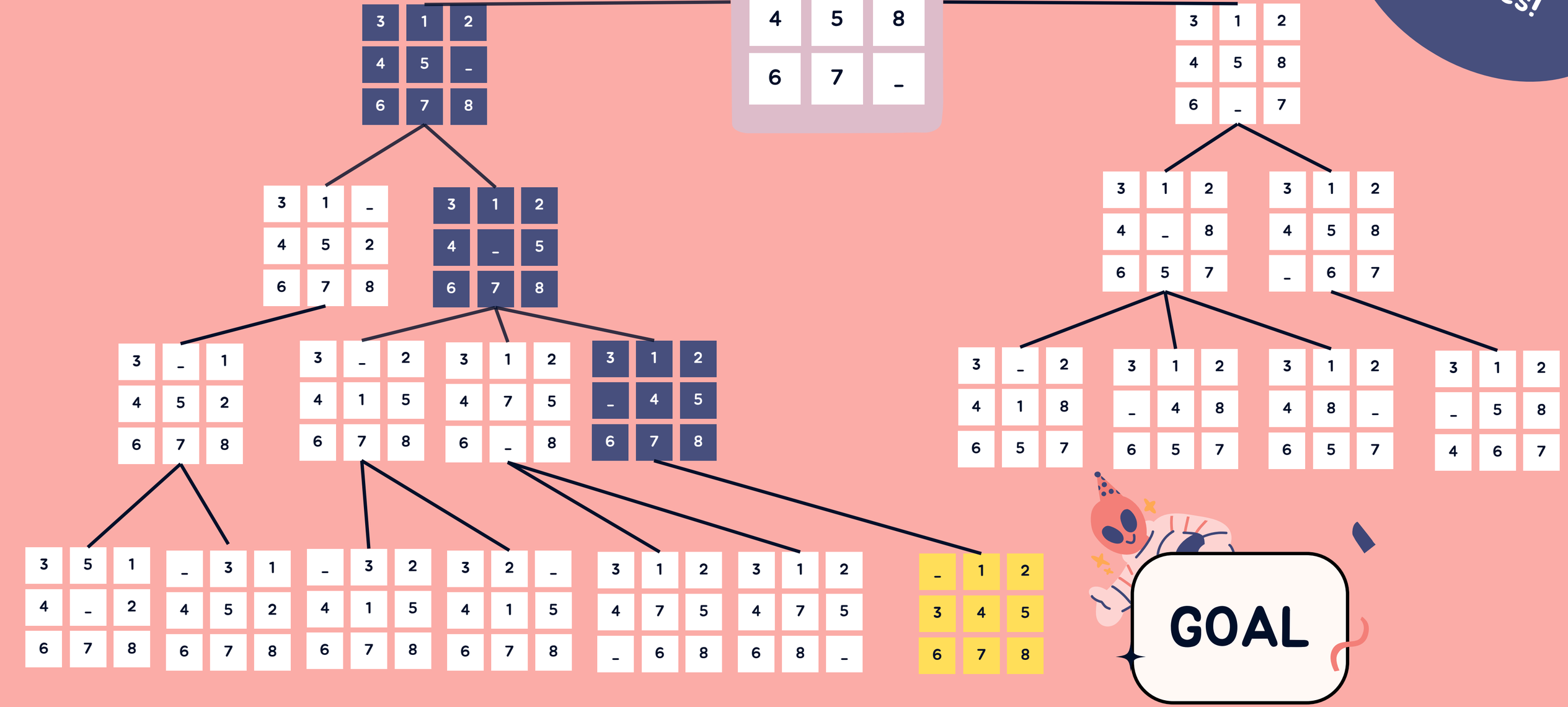


Simple Example

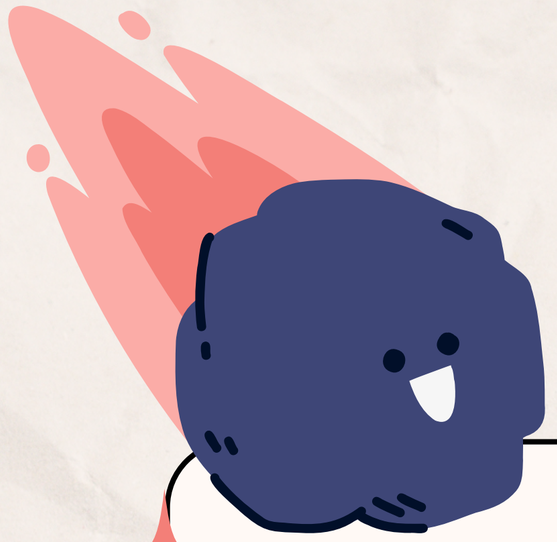
Initial State

3	1	2
4	5	8
6	7	_

Time to explore some new nodes!



GOAL



Initial Setup + Breadth First Search Codes

INITIAL SETUPS

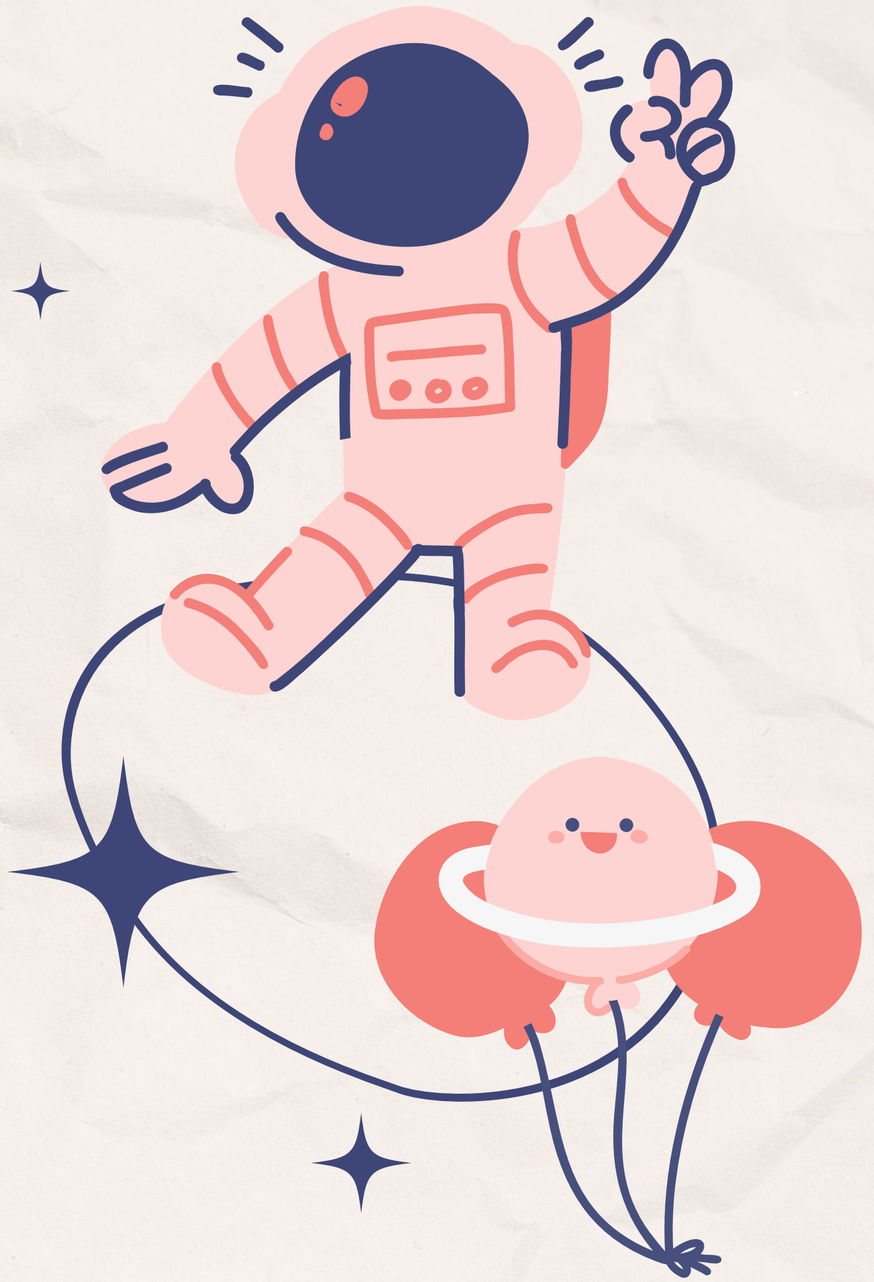
```
[ ]  import numpy as np
```

```
[ ]  initial_list = np.array([[3,1,2], [4, 5,8], [6,7,"_"]])
     def print_board(initial_list):
       for row in initial_list:
         print(f"{row[0]} | {row[1]} | {row[2]}")
       print("\n")
     print_board(initial_list)
```

```
3 | 1 | 2
4 | 5 | 8
6 | 7 | _
```

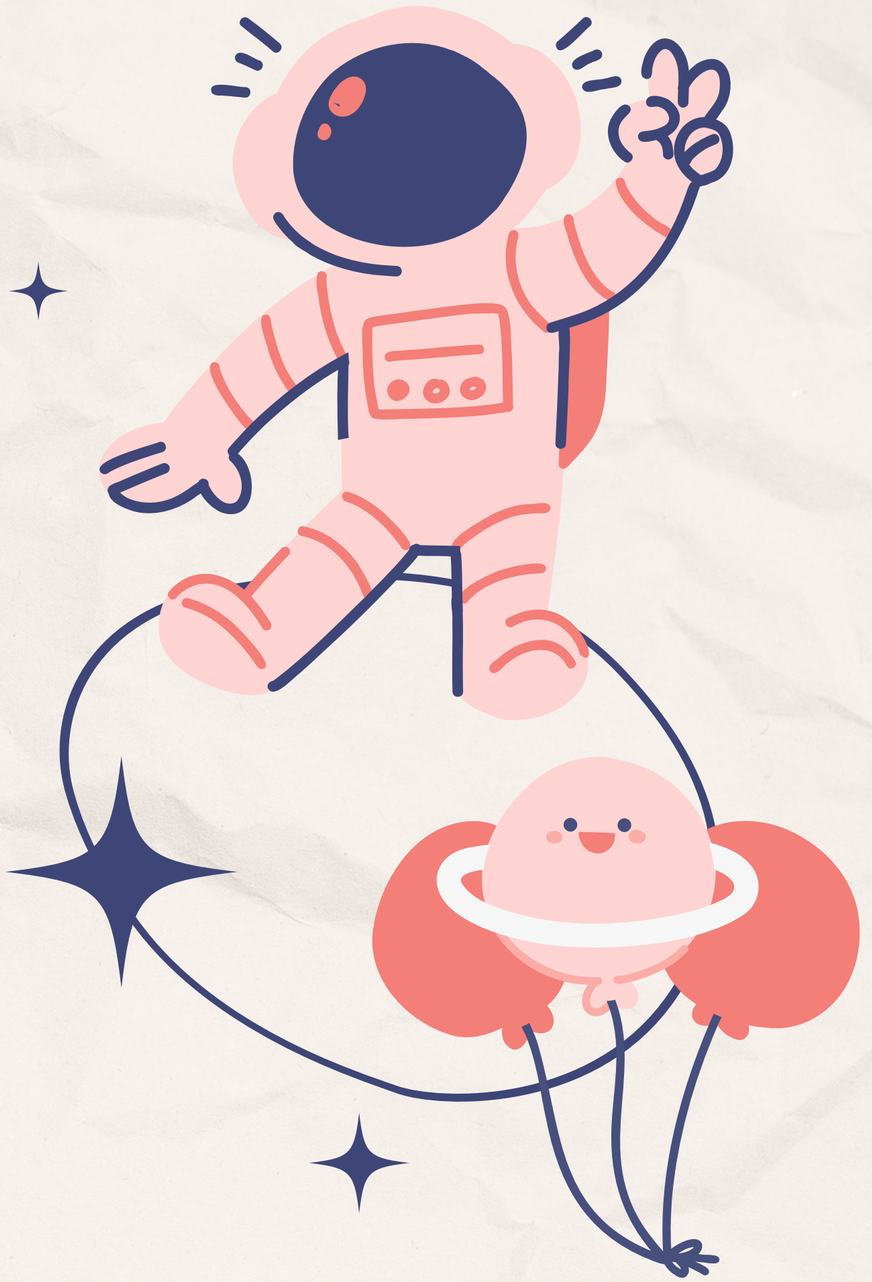
```
[ ]  goal_state = np.array([["_", 1,2], [3,4,5], [6,7,8]])
     goal_state
```

```
array([['_', '1', '2'],
       ['3', '4', '5'],
       ['6', '7', '8']], dtype='<U21')
```



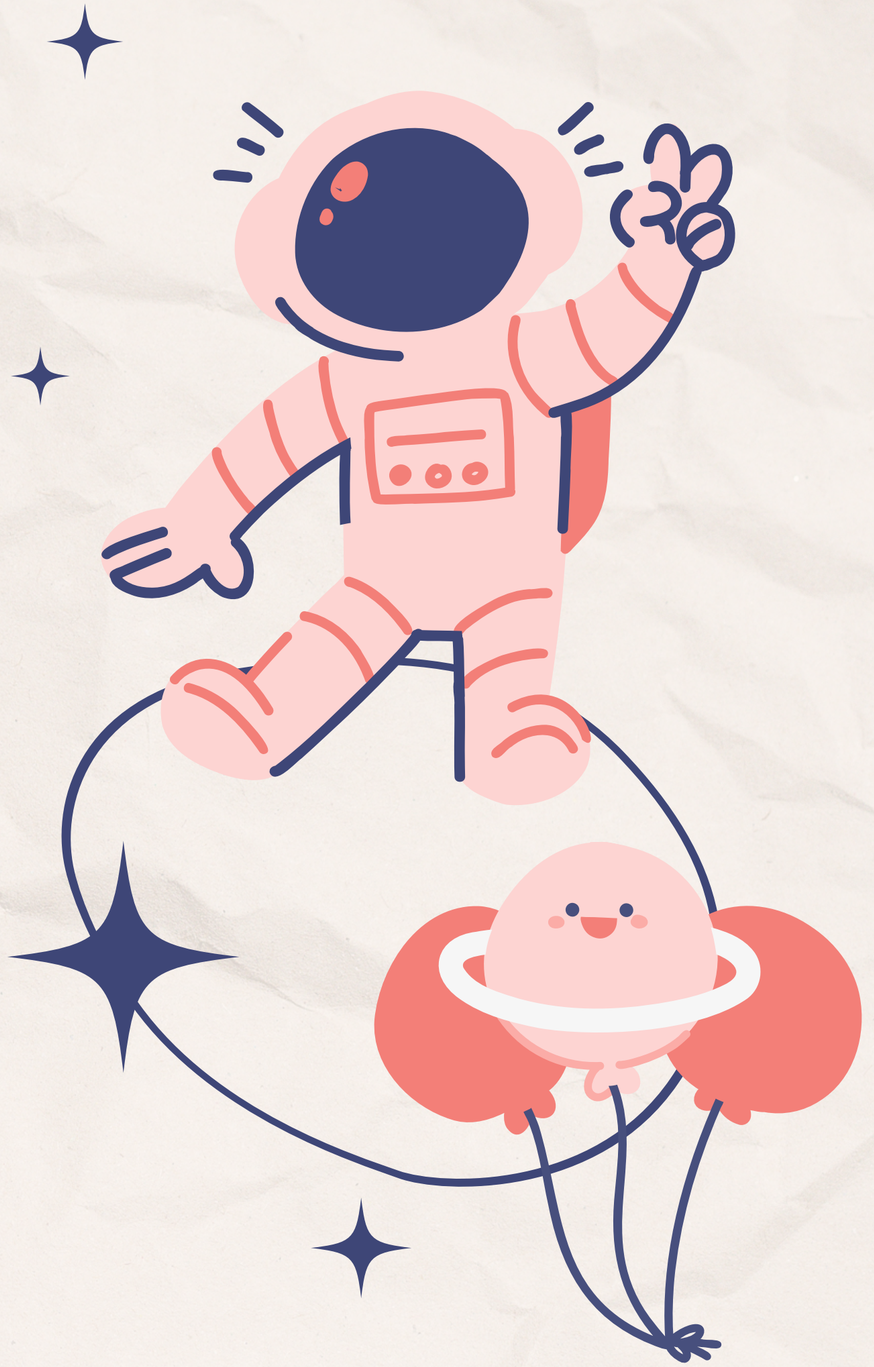
FUNCTION TO GET THE INDEX OF THE EMPTY TILE.

```
def getting_index_of_empty(initial_list):
  location = np.where(initial_list == "_" )
  coordinates = []
  for coordinate in location:
    coordinate = coordinate.tolist()
    for x in coordinate:
      coordinates.append(x)
  return coordinates # row column (index starting with 0)
```



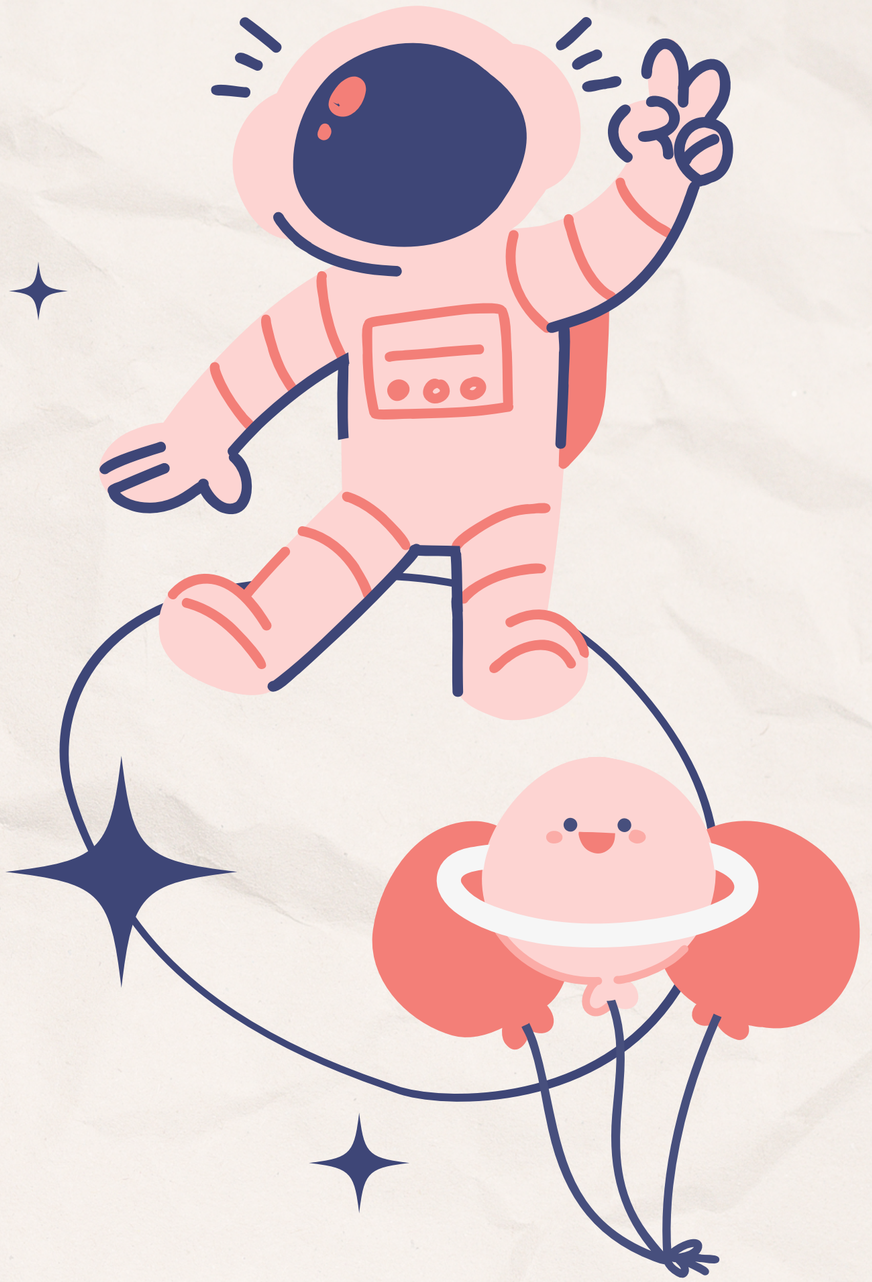
FUNCTION TO GET VALUES OF THE TILES ADJACENT TO THE EMPTY TILE

```
def getting_adjacent_tiles(initial_list):
  neighboring_array = []
  coordinates = getting_index_of_empty(initial_list)
  up = [coordinates[0]-1, coordinates[1]]
  down = [coordinates[0]+1, coordinates[1]]
  left = [coordinates[0], coordinates[1]-1]
  right = [coordinates[0], coordinates[1]+1]
  neighboring_array.extend([up, down, left, right])
  filters = [3, -1]
  final_l = [i for i in neighboring_array if not any(b in filters for b in i)]
  neighbors = []
  for num in final_l:
    x = num[0]
    y = num[1]
    a = initial_list[x,y]
    neighbors.append(a)
  return final_l, neighbors
```



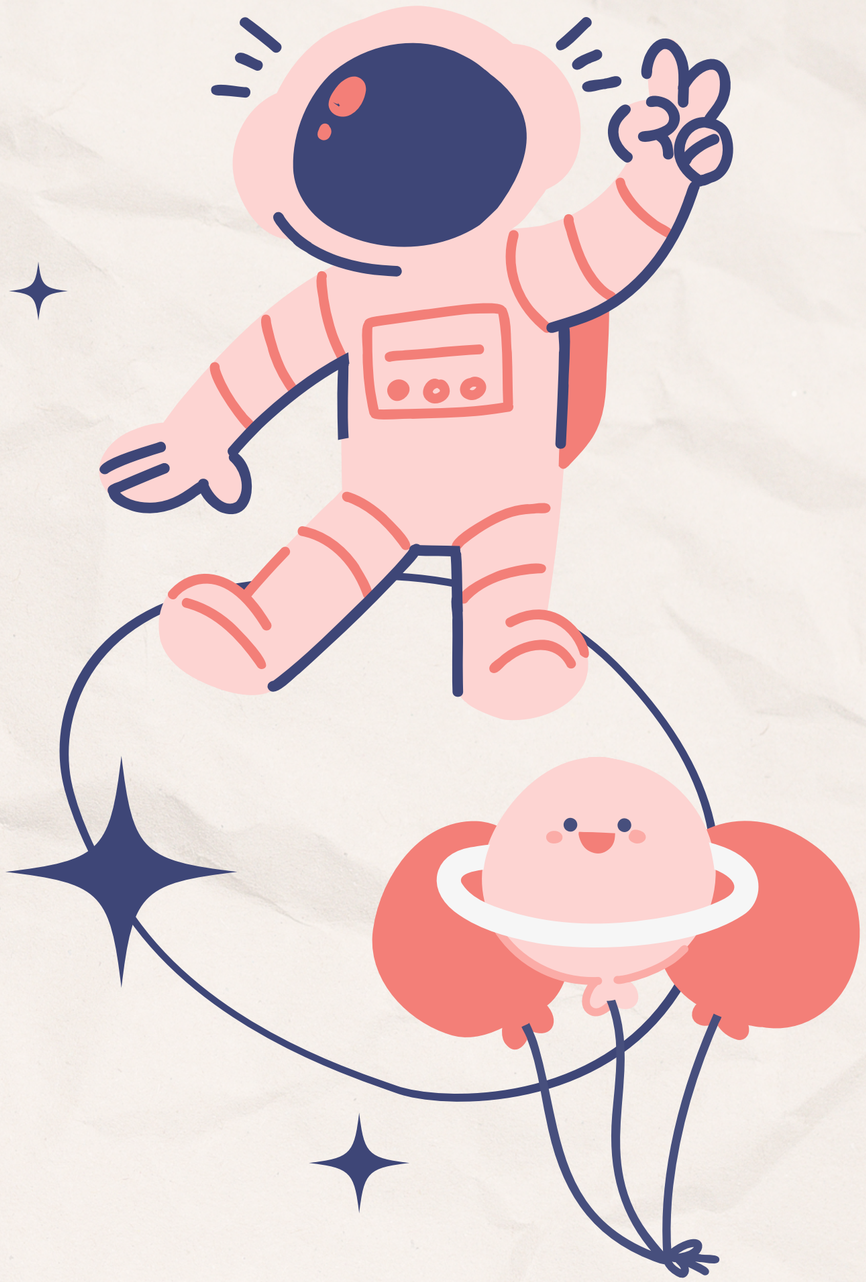
FUNCTION TO SWAP TILES

```
def switch_board(initial_list):
  boards = []
  boards.append(initial_list)
  empty_tile_coordinates = getting_index_of_empty(initial_list)
  neighboring_coordinates, neighbors = getting_adjacent_tiles(initial_list)
  for count, coordinate in enumerate(neighboring_coordinates):
    swapped_list = np.copy(initial_list)
    x = coordinate[0]
    y = coordinate[1]
    swapped_list[x,y] = "_"
    empty_tile_coordinate_x = empty_tile_coordinates[0]
    empty_tile_coordinate_y = empty_tile_coordinates[1]
    swapped_list[empty_tile_coordinate_x, empty_tile_coordinate_y] = neighbors[count]
    boards.append(swapped_list)
  return boards[1:]
```



FUNCTION TO CHECK WHETHER THE GOAL STATE IS IN THE QUEUE

```
def inList(test, goal):
    for element in test:
        if np.array_equal(element, goal):
            return True
    return False
```



BFS Algorithm

```
def BFS(initial_list, goal_state, flag):
  flag = flag # We only print the search tree once
  if flag == 0:
    print(initial_list)
  queue = [initial_list]
  checker = True
  for i in range(20000): # to prevent infinite loops
    if checker == False:
      break
    if flag == 0:
      print(i)
    if inList(queue, goal_state):
      print(queue[i])
      break
    temp_queue = switch_board(queue[i])
    for node in temp_queue:
      if (node==goal_state).all():
        if flag == 0:
          print("We have reached the goal node!")
        #x = i #for backtracking, x is the i at which we found the goal node!
        checker = False
      if inList(queue, node) == False: # We don't revisit children nodes we have
        if flag == 0:
          print(node)
        queue.append(node)
  return queue
```

1

We created a loop which iterates over the ith node in the queue

2

We used previously defined functions to generate the children nodes

BFS Algorithm

Function that returns the correct path obtained from the BFS

```
def path(goal, initial):
  flag = 1
  #returns the i
  parent_node = goal
  steps = [parent_node]
  while True:
    if (parent_node == initial).all():
      break
    x, queue = BFS(initial_list, parent_node, flag)
    parent_node = queue[x]
    steps.append(parent_node)
  inverted_steps = steps[::-1]
  for array in inverted_steps:
    print_board(array)
```

Uses recursion of the BFS algorithm to determine the parent nodes.

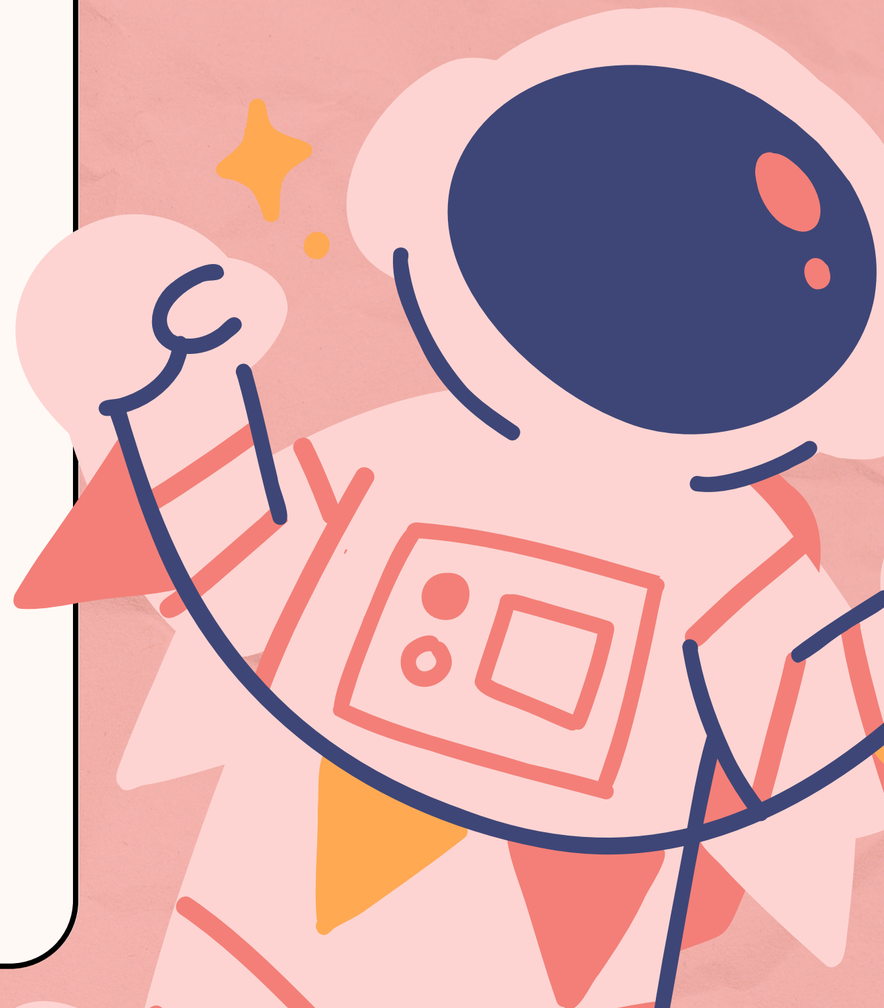
Sample Path

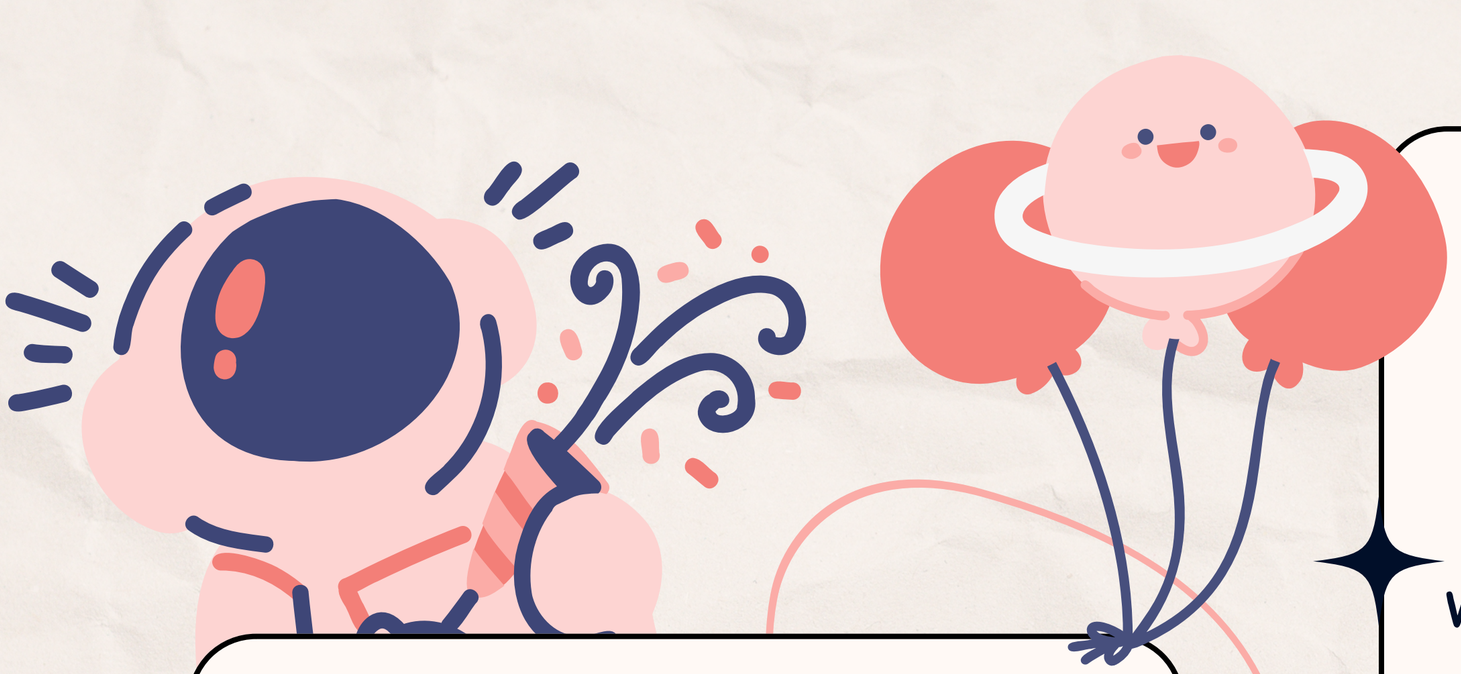
```
3 | 1 | 2
4 | 5 | 8
6 | 7 | _


3 | 1 | 2
4 | 5 | _
6 | 7 | 8


3 | 1 | 2
4 | _ | 5
6 | 7 | 8


3 | 1 | 2
_ | 4 | 5
6 | 7 | 8


_ | 1 | 2
3 | 4 | 5
6 | 7 | 8
```

Initial State

Goal State

GREEDY BEST-FIRST SEARCH





The second type
GREEDY BEST-FIRST

- INFORMED SEARCH ALGORITHM BASED ON A DISTANCE HEURISTIC, H(N)
- OUR EVALUATION FUNCTION IS PICKING THE NODE WITH THE LEAST H SCORE

WE USED TWO HEURISTICS:

1. MANHATTAN DISTANCE
- RIGHT ANGLE DISTANCE
- GET THE SUM OF THE ABSOLUTE VALUE OF THE DIFFERENCES IN COORDINATES OF EACH TILE

2. HAMMING DISTANCE
- NUMBER OF INCORRECTLY PLACED TILES ACCORDING TO THE GOAL STATE

Manhattan distance

TILE	DISTANCE
1	0
2	0
3	1
4	1
5	1
6	0
7	0
8	1
_	4
Total	7

Current State

3	1	2
4	5	8
6	7	_

Goal State

_	1	2
3	4	5
6	7	8

Heuristic: 7

Hamming distance

TILE	MISPLACED?
1	NO
2	NO
3	YES
4	YES
5	YES
6	NO
7	NO
8	YES
_	YES
Total	5

Current State

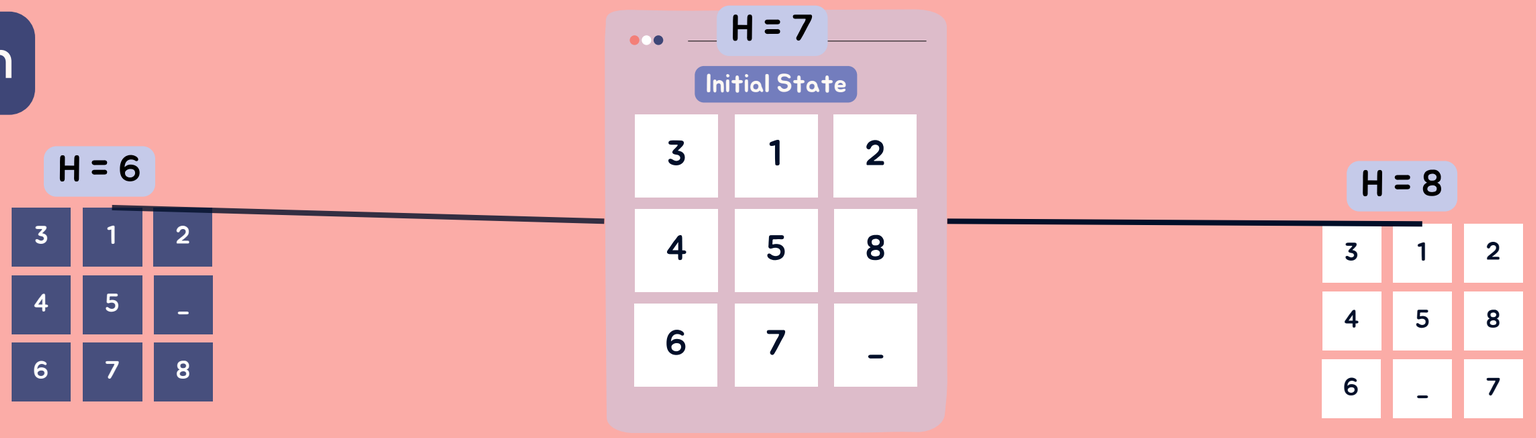
3	1	2
4	5	8
6	7	_

Goal State

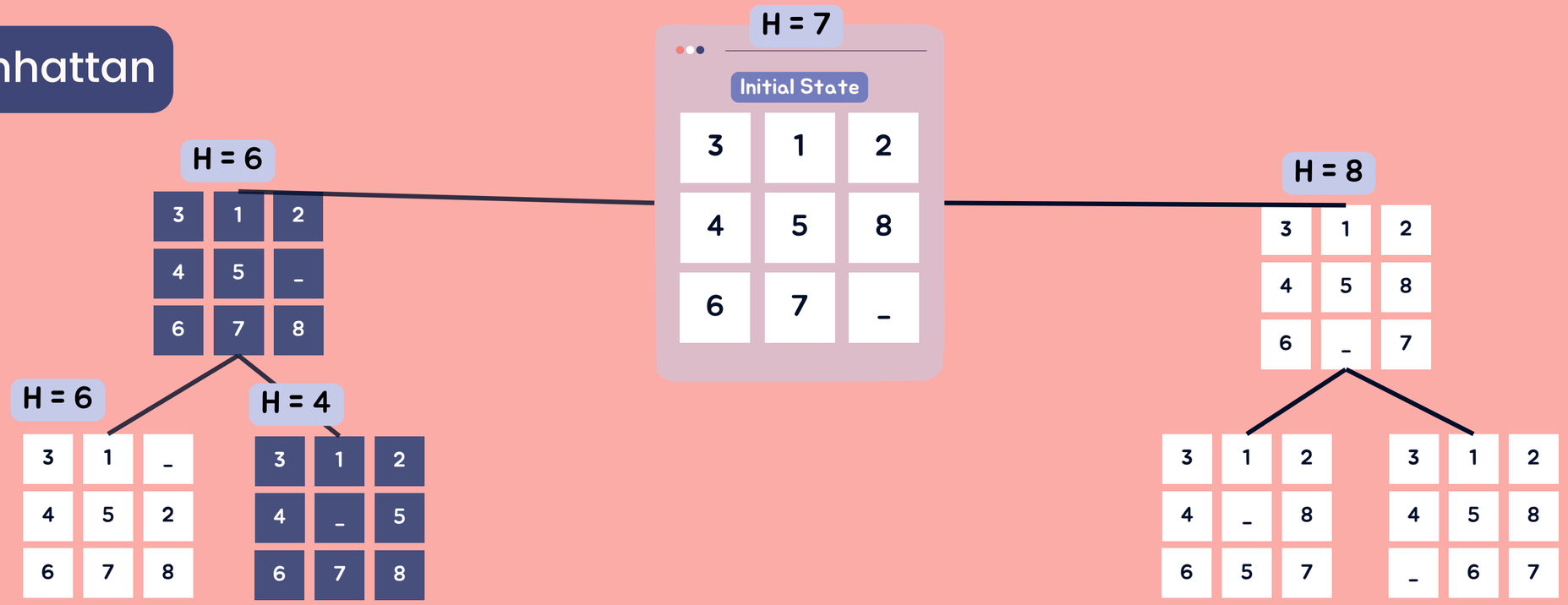
_	1	2
3	4	5
6	7	8

Heuristic: 5

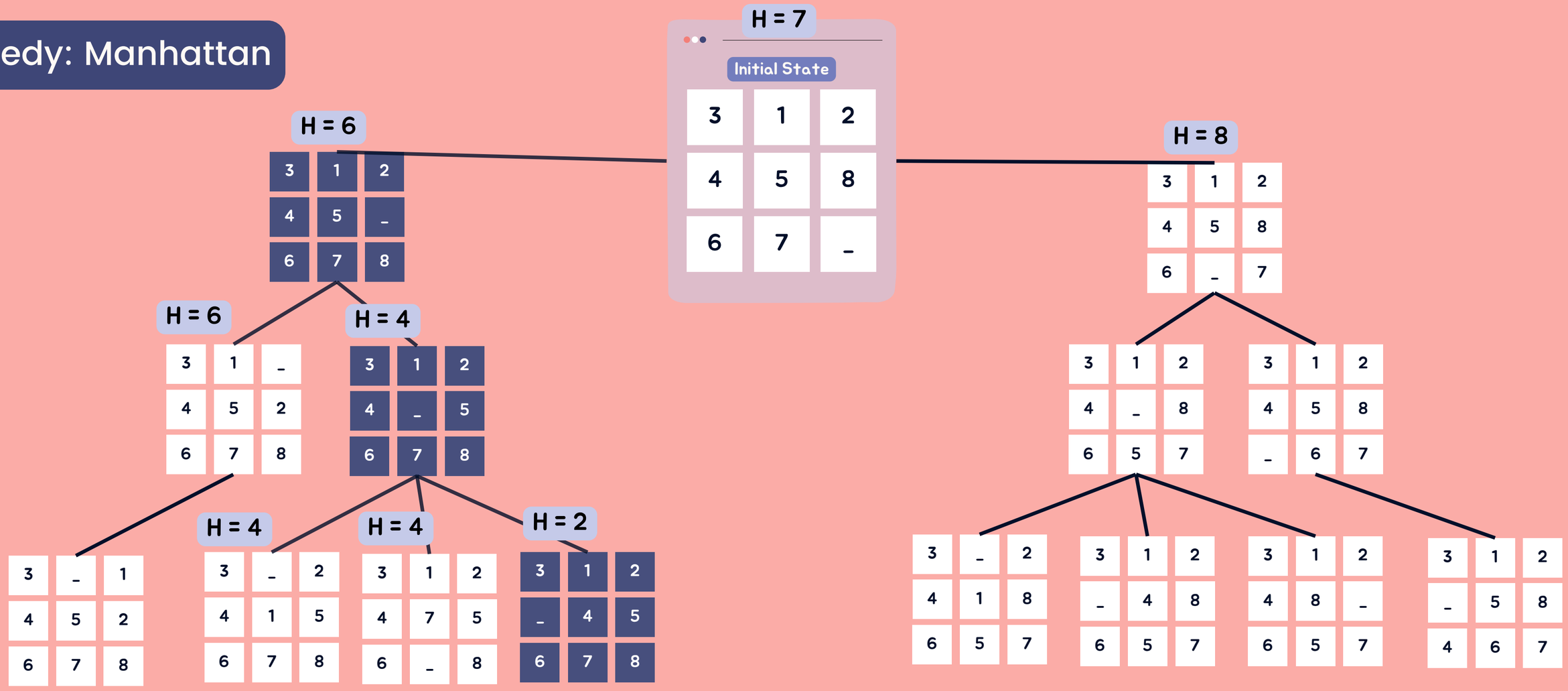
Greedy: Manhattan



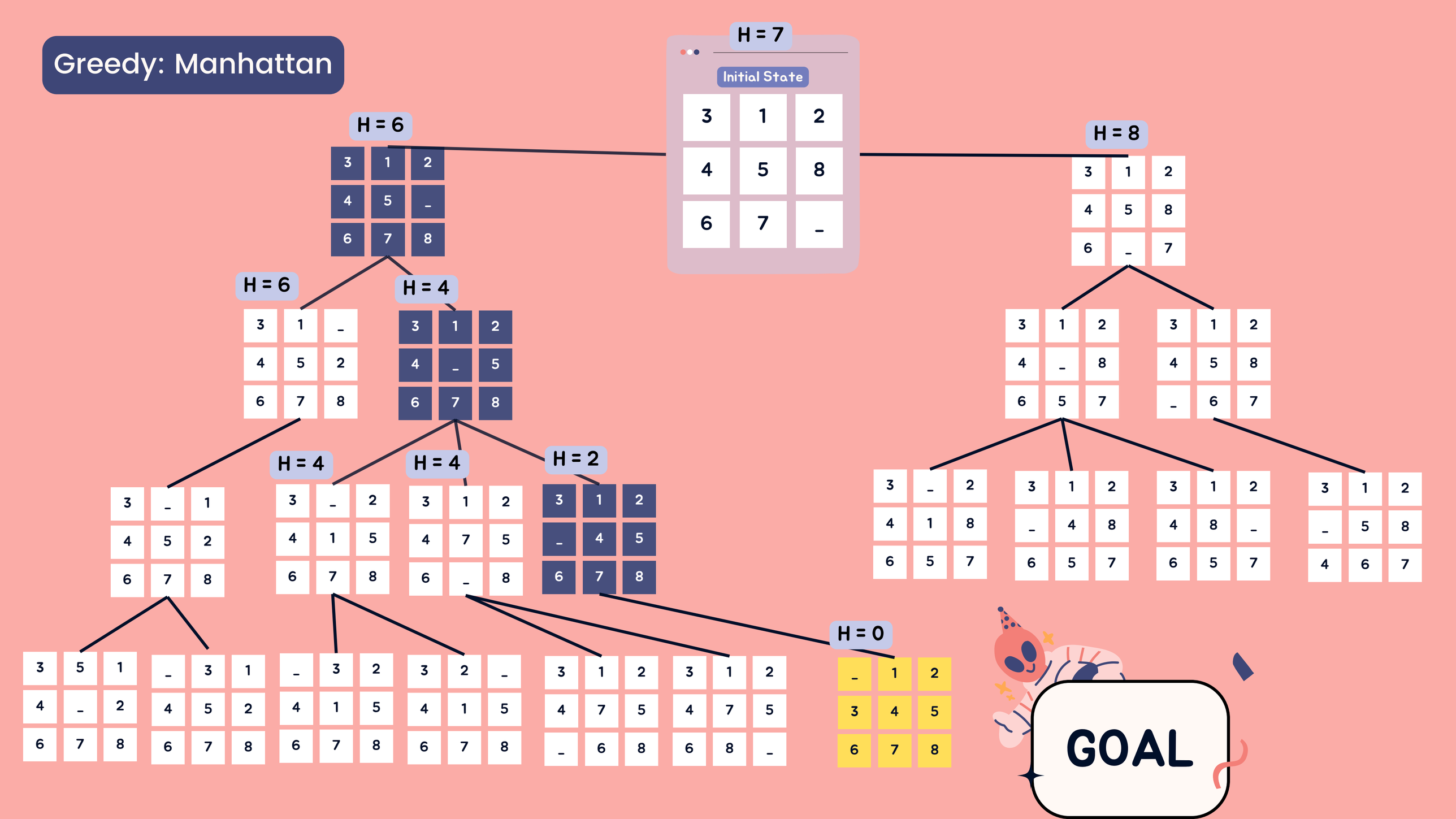
Greedy: Manhattan



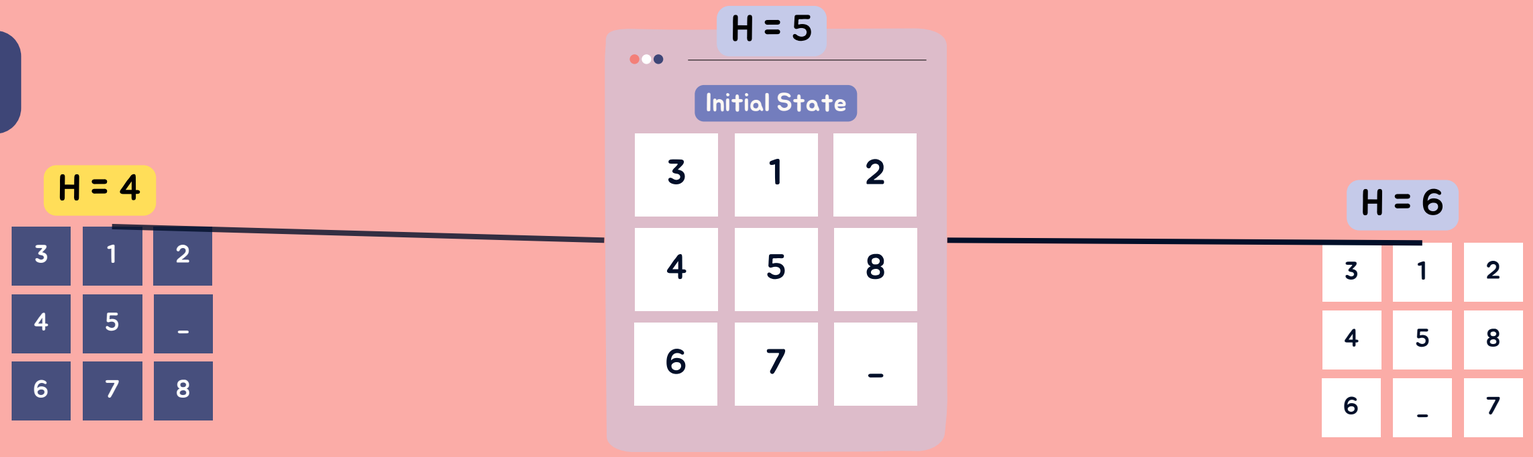
Greedy: Manhattan



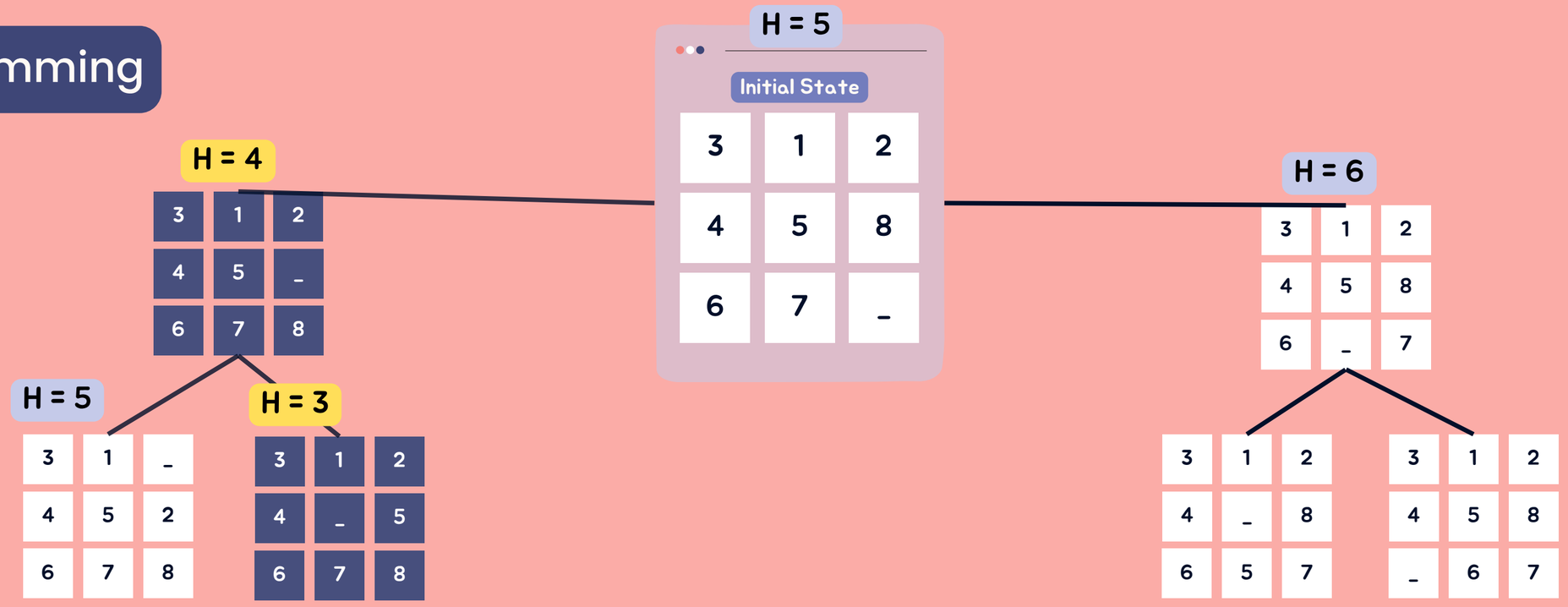
Greedy: Manhattan



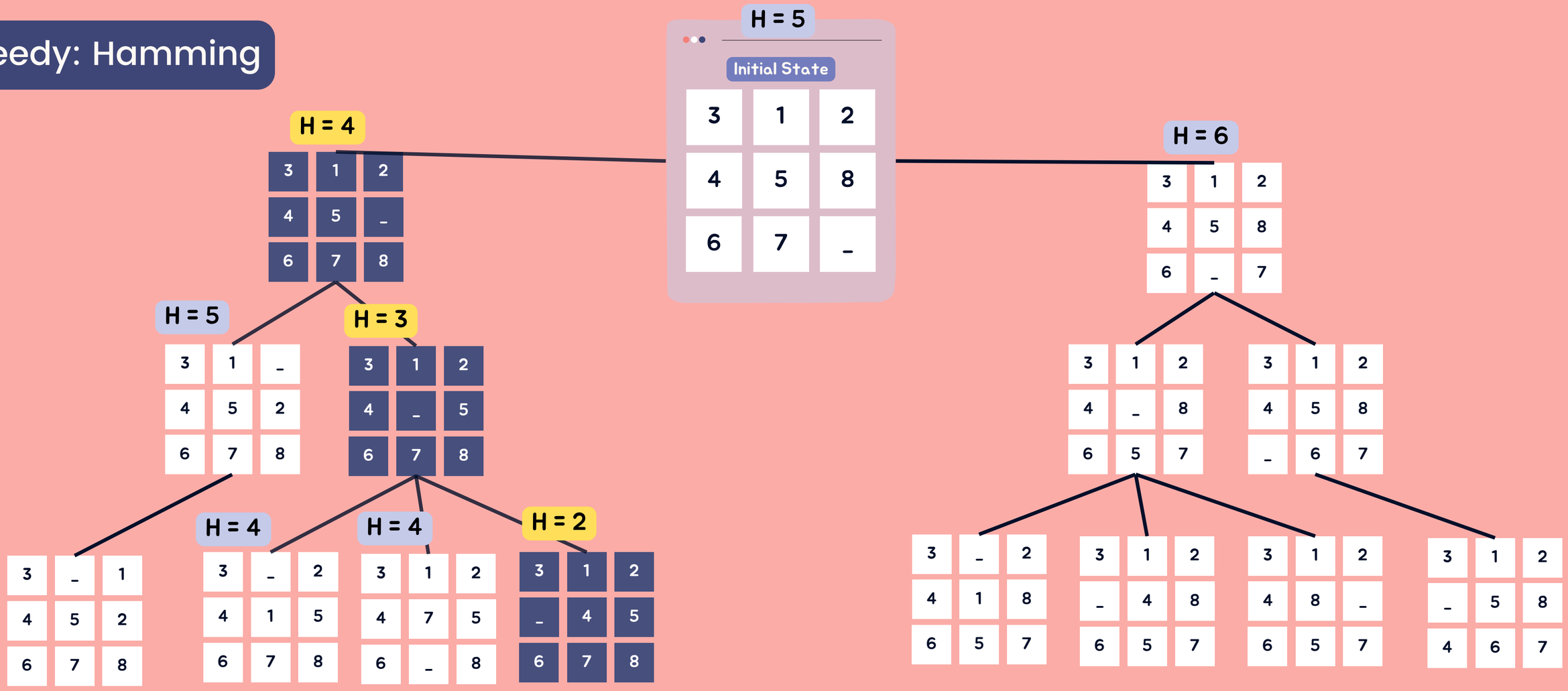
Greedy: Hamming



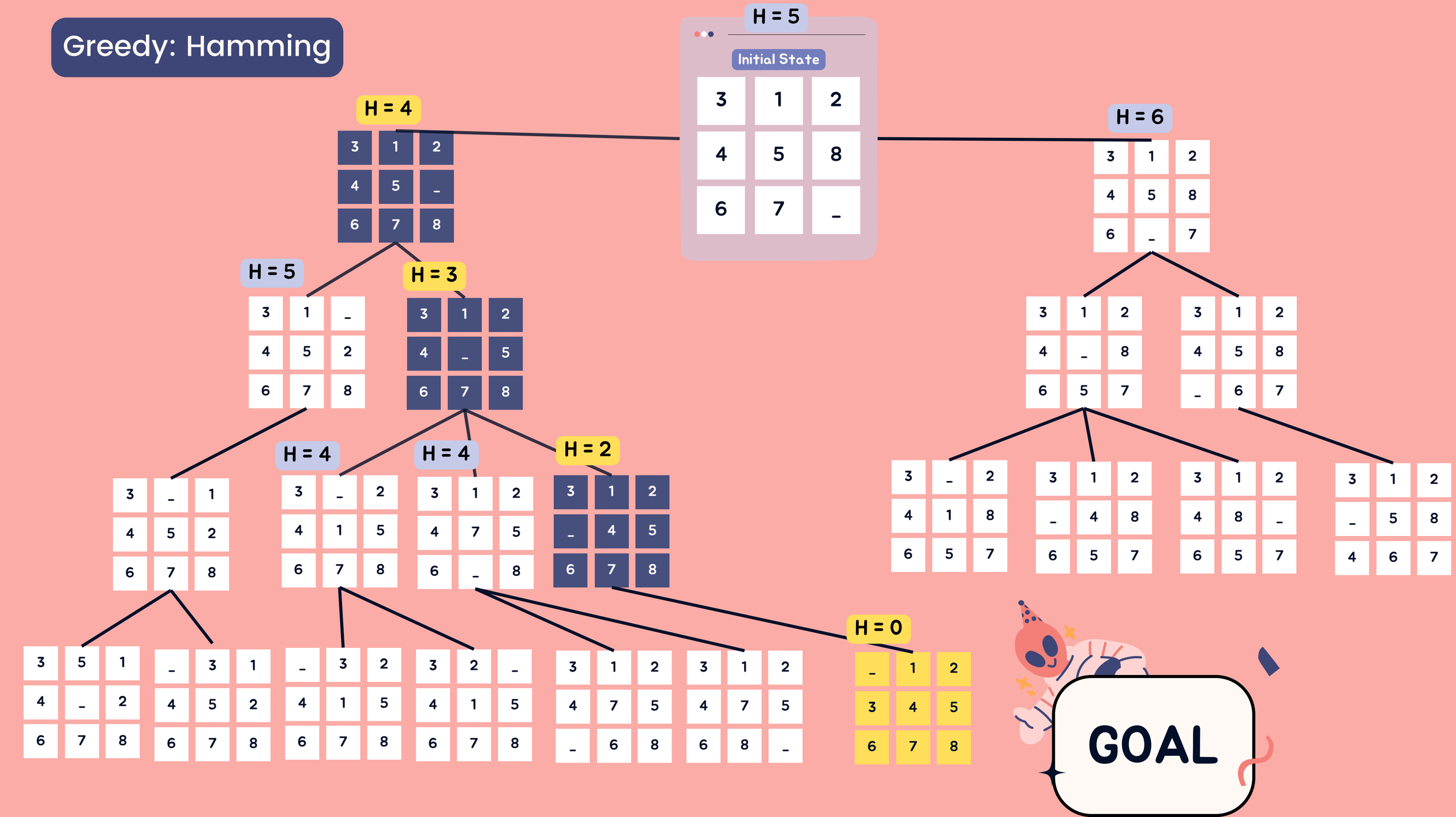
Greedy: Hamming



Greedy: Hamming



Greedy: Hamming



Greedy Best-First Search Codes

Defining Functions

```
def getting_index_of_key(current_state, key):
  location = np.where(current_state == key)
  coordinates = []
  for coordinate in location:
    coordinate = coordinate.tolist()
    for x in coordinate:
      coordinates.append(x)
  return coordinates # row column (index starting with 0)
```

WE USE NP.WHERE TO OUTPUT THE INDEX (ROW-COLUMN FORM) OF A CERTAIN TILE IN THE GRID.

THE OUTPUT, COORDINATES, IS A LIST [ROW NO., COL. NO].

Heuristic Functions

```python
from math import sqrt

#create function to calculate Manhattan distance
def manhattan(a, b):
    sum = 0
    for val1, val2 in zip(a,b):
        for a, b in zip(val1,val2):
          sum+=abs(a-b)
    return sum

def hamming(a,b): # number of misplaced tiles
  sum = 0
  for val1, val2 in zip(a,b):
    if val1 != val2:
      sum += 1
  return sum
```

WE CAN CALCULATE OUR HEURISTIC FOR THE A* AND GREEDY BEST FIRST SEARCH USING THESE 2 FUNCTIONS, WHICH TAKE IN TWO STATES (CURRENT STATE/GOAL STATE) AS INPUT.

Greedy Best First Search

```
def get_coordinates_of_all(current_state):
  number_dict = {"_": getting_index_of_key(current_state, "_")}
  for i in range (1,9):
    i_str = str(i)
    current_key =  getting_index_of_key(current_state, i_str)
    number_dict[i] = current_key
  return number_dict

from math import sqrt
```

THIS FUNCTION USES THE PREVIOUS FUNCTION TO STORE THE COORDINATES OF ALL THE TILES IN A SINGLE DICTIONARY.

Greedy Best First Search Algorithm

```
# Manhattan or Hamming
hard_example = np.array([[1,8,2], ["_",4, 3], [7,6,5]])
def best_first(initial_list, goal_state, flag, heuristic):
  flag = flag # We only print the search tree once
  heuristic = heuristic
  # Our priority queue will contain the initial state and the h score of the board
  priority_queue = {}
  print(hard_example)
  # We will also have a list containing all the states we have explored.
  states = [initial_list]
  if heuristic == "Manhattan":
    h_score = manhattan(get_coordinates_of_all(initial_list).values(), (get_coordinates_of_all(goal_state).values()))
  elif heuristic == "Hamming":
    h_score = hamming(get_coordinates_of_all(initial_list).values(), (get_coordinates_of_all(goal_state).values()))
```

- WE DEFINE A PRIORITY QUEUE DICTIONARY WITH KEY = TUPLE(NUMPY ARRAY), AND VALUE = HEURISTIC (MANHATTAN OR HAMMING)
- WE USE THE AFOREMENTIONED MANHATTAN() AND HAMMING() FUNCTIONS TO GET THE VALUE OF THE HEURISTIC FOR THE EACH NODE.

Greedy Best First Search Algorithm

```
for i in range(5000): # to prevent infinite loops
  if checker == False:
    break
  if flag == 0:
    print(i)
  lowest = min(priority_queue.items(), key=lambda x: x[1]) # get th
  state = lowest[0]
  priority_queue.pop(state)
  state = np.asarray(state) # convert tuple back to a numpy array
  min_h_score = lowest[1]
```

- IN EACH ITERATION OF THE LOOP, WE SELECT THE KEY-VALUE PAIR OF THE DICTIONARY WITH THE LOWEST VALUE USING A LAMBDA FUNCTION.
- WE THEN CONVERT THE KEY BACK INTO A NUMPY ARRAY.
- WE POP OR REMOVE THE STATE FROM OUR QUEUE.

Greedy Best First Search Algorithm

```
temp_queue = switch_board(state)
for node in temp_queue:
  if (node==goal_state).all():
    if flag == 0:
      print("We have reached the goal node!")
    #x = i #for backtracking, x is the i at which we found the goal node!
    checker = False
  if inList(states, node) == False: # We only explore the children nodes we haven't explored before
    states.append(node)
    if heuristic == "Manhattan":
      h_score = manhattan(get_coordinates_of_all(node).values(), (get_coordinates_of_all(goal_state).values()))
    elif heuristic == "Hamming":
      h_score = hamming(get_coordinates_of_all(node).values(), (get_coordinates_of_all(goal_state).values()))
    new_h_score = h_score
    if flag == 0:
      print(f'h_score = {h_score}')
      print(node)
    node = tuple(map(tuple, node))
    priority_queue[node] = new_h_score
```

WE THEN EXPLORE THE CHILDREN NODES OF THE CURRENT STATE. WE ADD THE CHILDREN NODE TO THE PRIORITY QUEUE IF IT HASN'T BEEN EXPLORED BEFORE. WE GET THE H-SCORE OF EACH OF THE CHILDREN NODES.

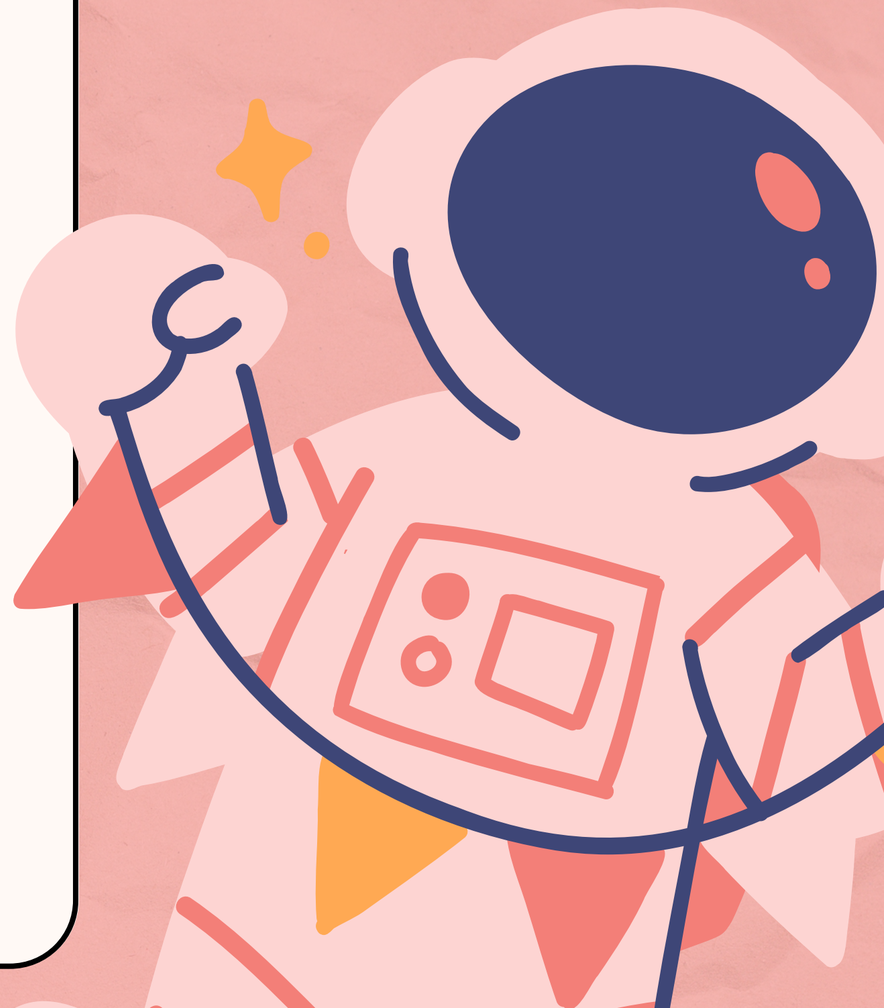
Sample Function Output

The ith node expanded

The children nodes and their respective h scores are printed.

```
[['3' '1' '2']
 ['4' '5' '8']
 ['6' '7' '_']]
0
h_score = 6
[['3' '1' '2']
 ['4' '5' '_']
 ['6' '7' '8']]
h_score = 8
[['3' '1' '2']
 ['4' '5' '8']
 ['6' '_' '7']]
1
h_score = 6
[['3' '1' '_']
 ['4' '5' '2']
 ['6' '7' '8']]
h_score = 4
[['3' '1' '2']
 ['4' '_' '5']
 ['6' '7' '8']]
```

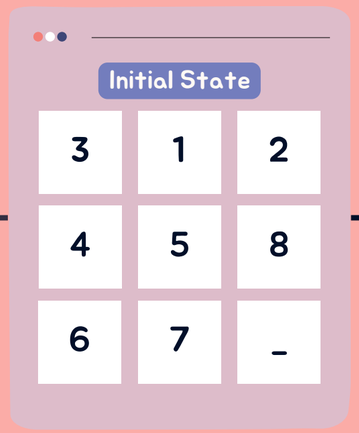
A*
SEARCH



The third type
A* SEARCH

- INFORMED SEARCH ALGORITHM BASED ON A COST AND DISTANCE HEURISTIC.
- OUR EVALUATION FUNCTION IS F(N) = G(N) + H(N).
- G(N) IS THE COST TO REACH THE NODE. H(N) IS THE MANHATTAN (RIGHT ANGLE) DISTANCE FROM NODE TO GOAL OR HAMMING DISTANCE (NUMBER OF MISPLACED TILES).
- OUR ALGORITHM PICKS THE NODE WITH THE LEAST F SCORE AND EXPANDS IT.
- WE END THE SEARCH ONCE ONE OF THE CHILDREN NODES IS EQUAL TO THE GOAL NODE

A* Search: Manhattan



G = 1, H = 6

3	1	2
4	5	_
6	7	8

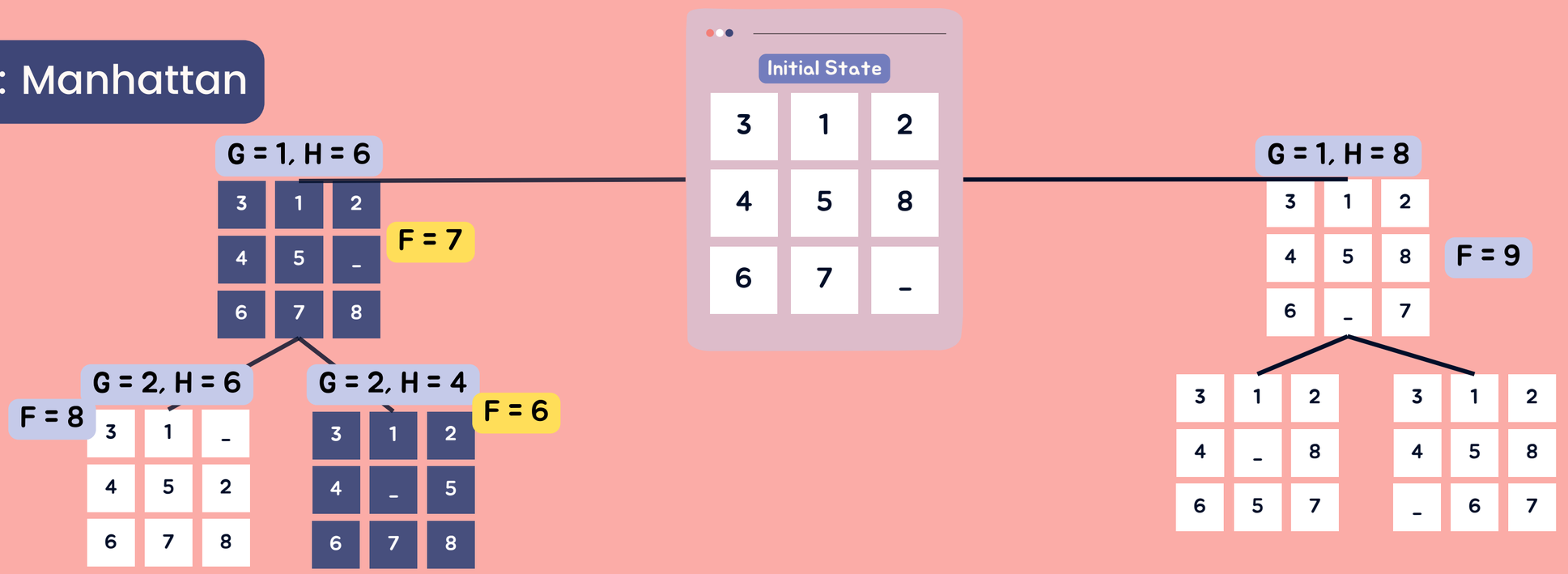
F = 7

G = 1, H = 8

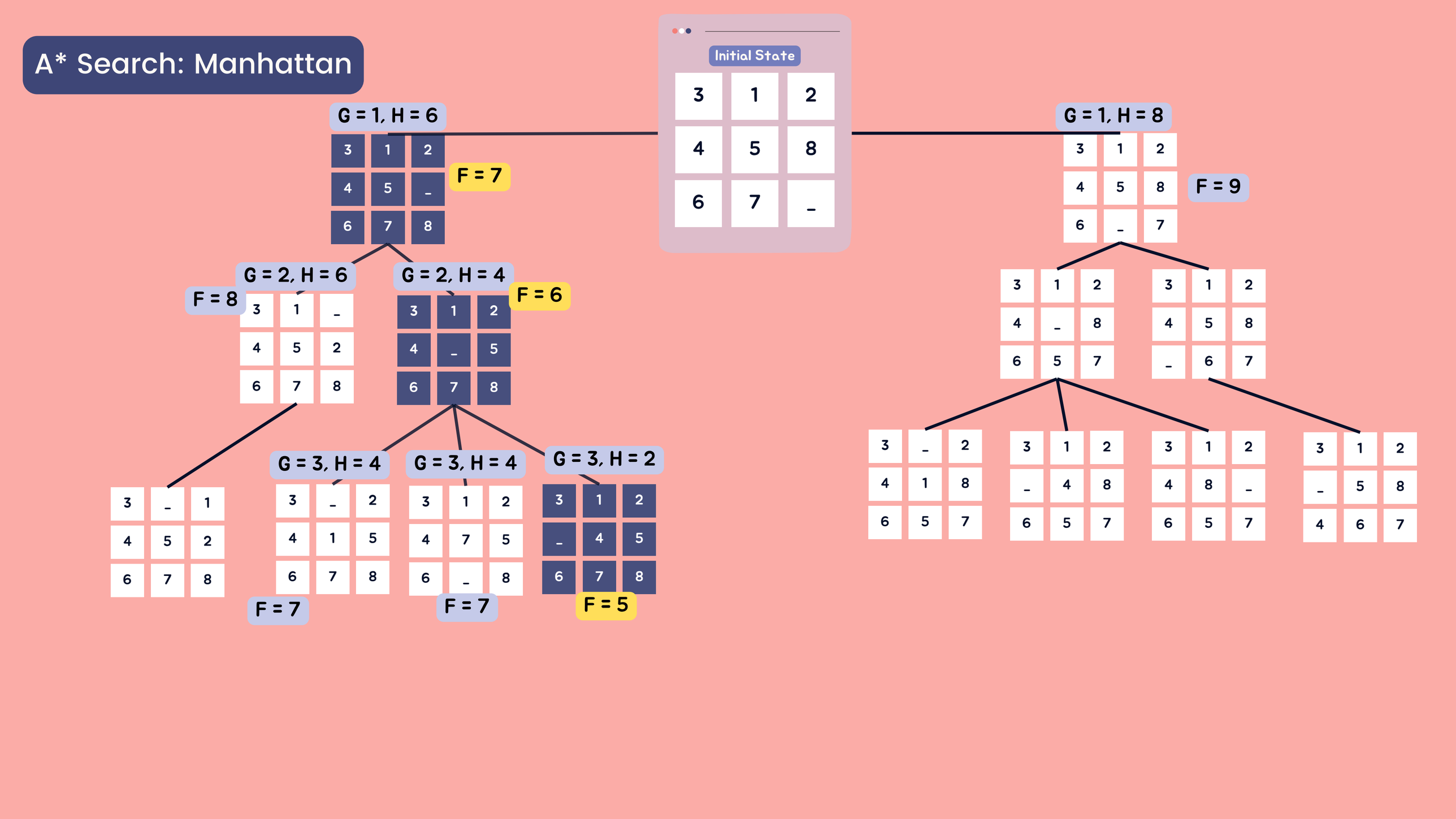
3	1	2
4	5	8
6	_	7

F = 9

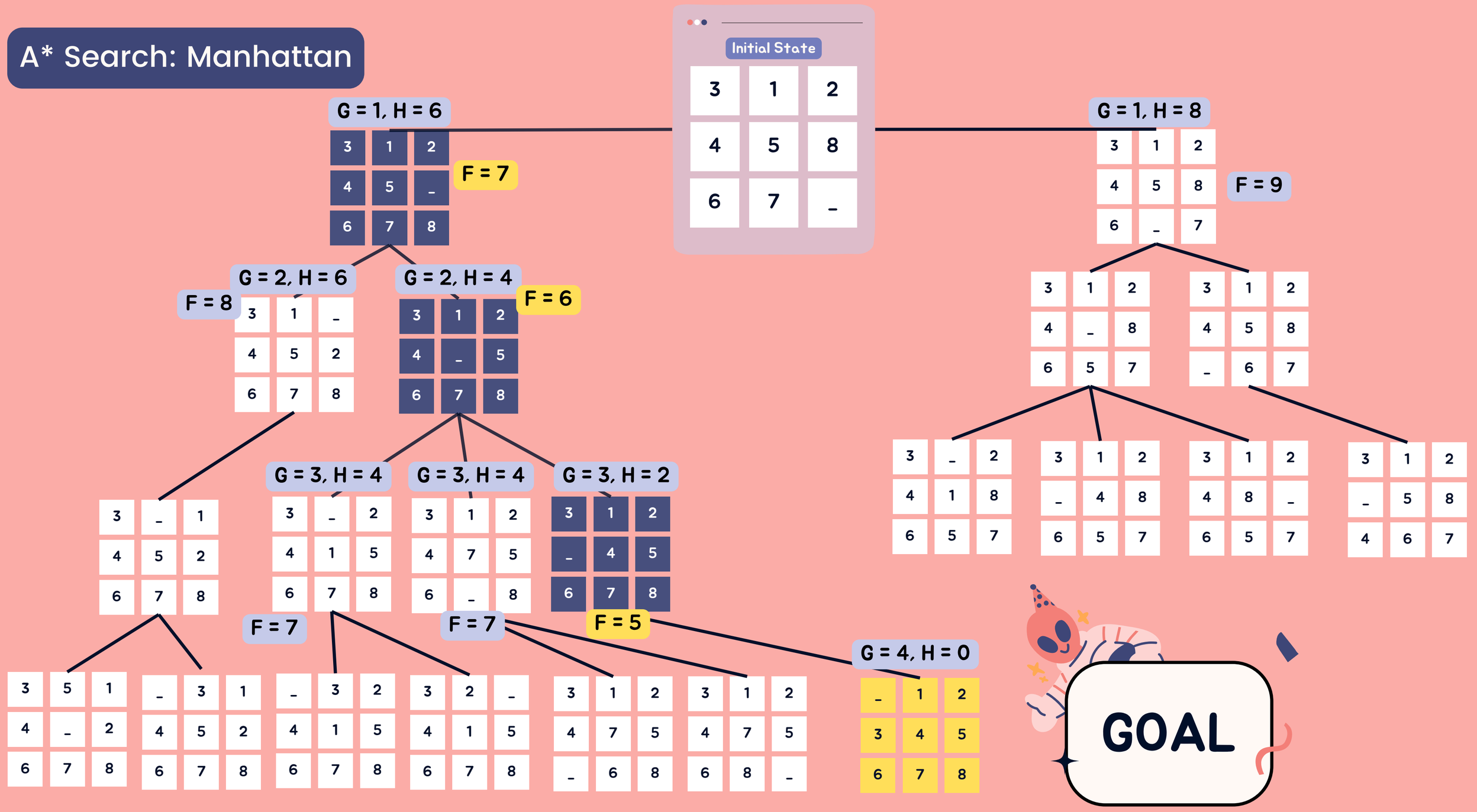
A* Search: Manhattan



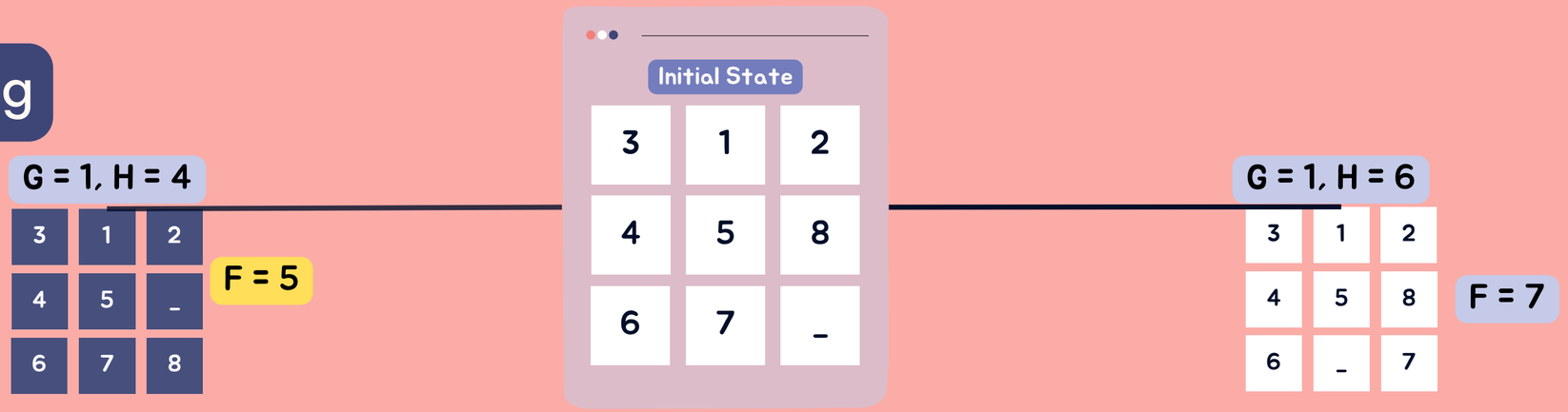
A* Search: Manhattan



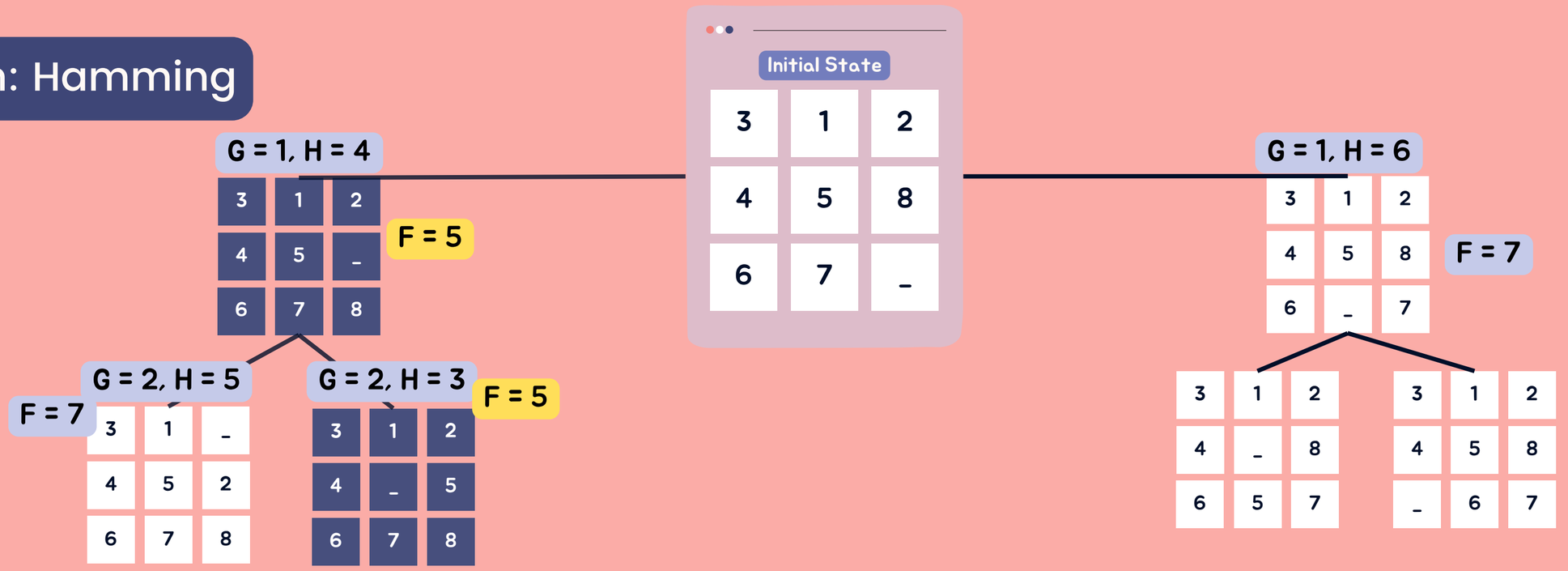
A* Search: Manhattan



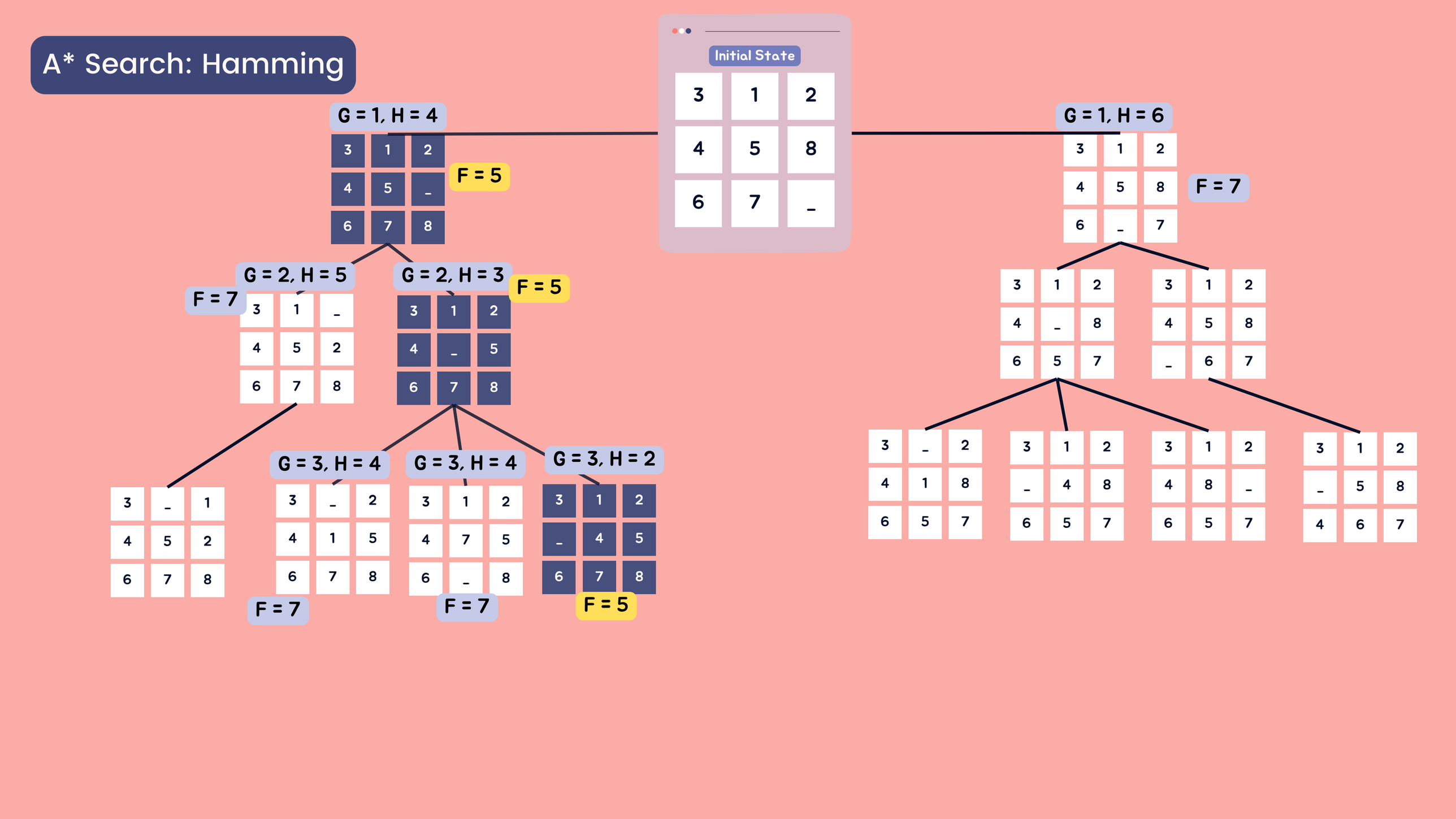
A* Search: Hamming



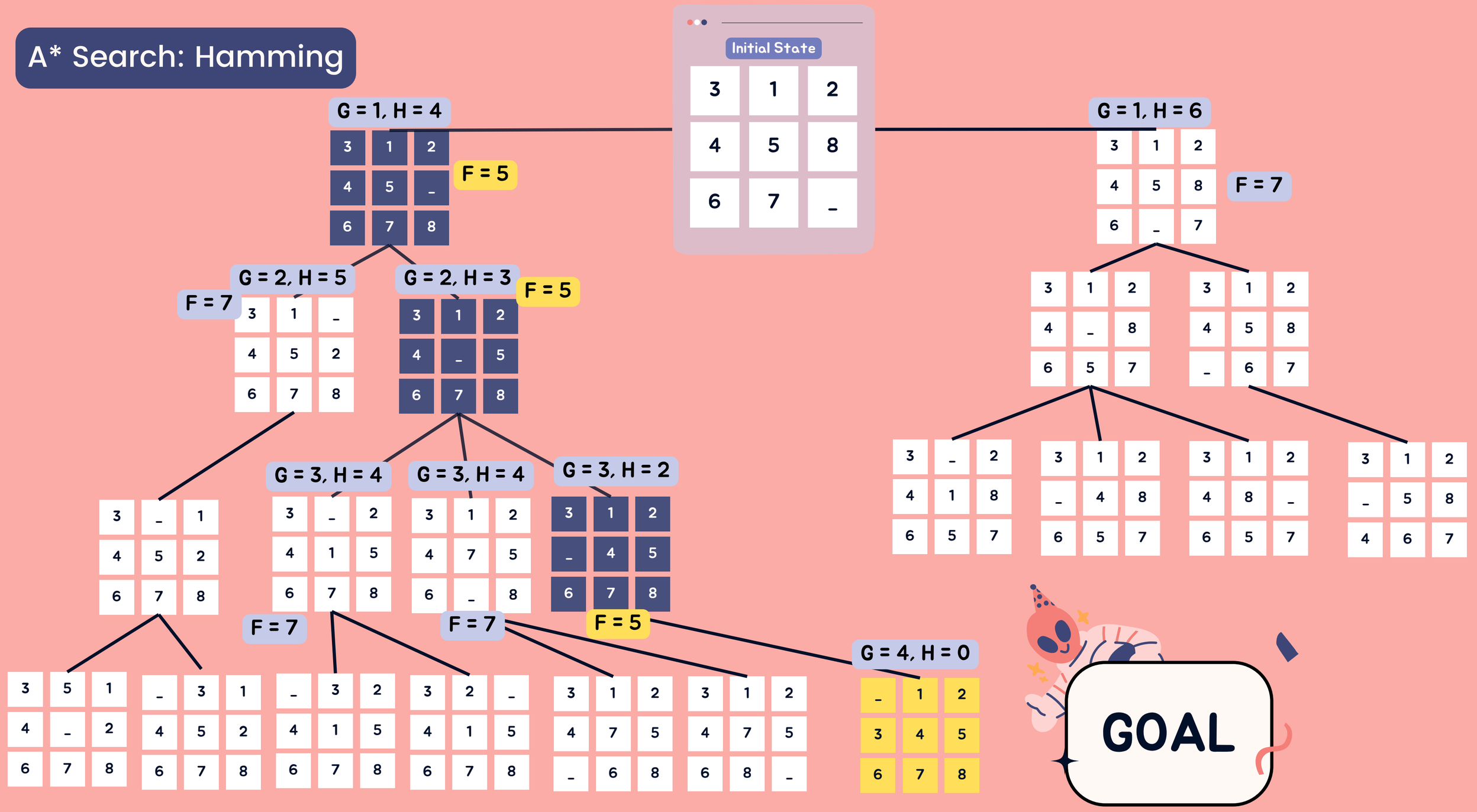
A* Search: Hamming



A* Search: Hamming



A* Search: Hamming



A* Search Codes

Differences between A* and Best First Search

```
g_score = 0 # depth of initial node is 0
if heuristic == "Manhattan":
  h_score = manhattan(get_coordinates_of_all(initial_list).values(), (get_coordinates_of_all(goal_state).values()))
elif heuristic == "Hamming":
  h_score = hamming(get_coordinates_of_all(initial_list).values(), (get_coordinates_of_all(goal_state).values()))
f_score = g_score + h_score
# convert numpy array to a tuple because the key of a python dictionary must be unhashable
initial_list = tuple(map(tuple, initial_list))
priority_queue[initial_list] = f_score
```

WE ADD A G_SCORE EQUIVALENT TO THE DEPTH OF THE NODE TO OUR H SCORE.

Differences between A* and Best First Search

```
states.append(node)
if heuristic == "Manhattan":
  h_score = manhattan(get_coordinates_of_all(node).values(), (get_coordinates_of_all(goal_state).values()))
  parent_g_score = min_f_score - manhattan(get_coordinates_of_all(state).values(), (get_coordinates_of_all(goal_state).values()))
elif heuristic == "Hamming":
  h_score = hamming(get_coordinates_of_all(node).values(), (get_coordinates_of_all(goal_state).values()))
  parent_g_score = min_f_score - hamming(get_coordinates_of_all(state).values(), (get_coordinates_of_all(goal_state).values()))
child_g_score = parent_g_score + 1
new_f_score = child_g_score + h_score
```

FOR EACH CHILDREN NODE, WE FIND THE G-SCORE OF THE PARENT THEN ADD ONE.

Differences between A* and Best First Search

```
61 a_star(initial_list, goal_state, 0, "Hamming")
```

The ith node expanded

3 children nodes and their corresponding g and h scores

Goal Node has been reached

```
2
g-score = 3 --- h_score = 4
[['3' '_' '2']
 ['4' '1' '5']
 ['6' '7' '8']]
g-score = 3 --- h_score = 4
[['3' '1' '2']
 ['4' '7' '5']
 ['6' '_' '8']]
g-score = 3 --- h_score = 2
[['3' '1' '2']
 ['_' '4' '5']
 ['6' '7' '8']]
3
We have reached the goal node!
g-score = 4 --- h_score = 0
[['_' '1' '2']
 ['3' '4' '5']
 ['6' '7' '8']]
```