

# P2: Torus 8-Puzzle A\* Search

**Due** Feb 11 by 9:29am    **Points** 100    **Submitting** a file upload    **File Types** py  
**Available** Feb 4 at 11am - Feb 11 at 9:29am 7 days

This assignment was locked Feb 11 at 9:29am.

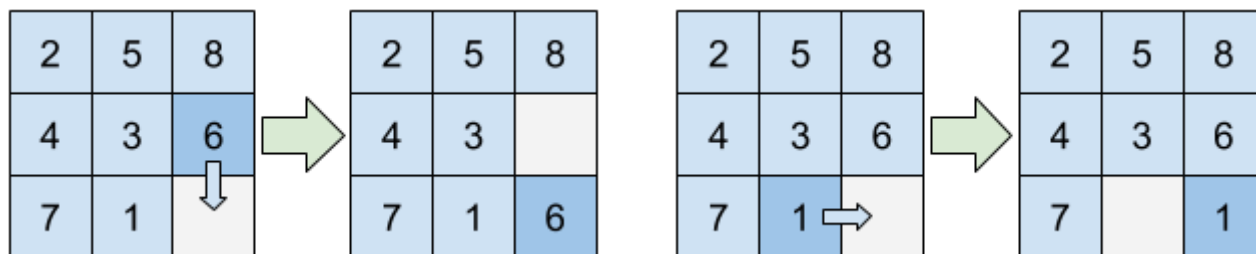
## Program Goals

- Deepen understanding of state space generation
- Practice implementation of an efficient search algorithm

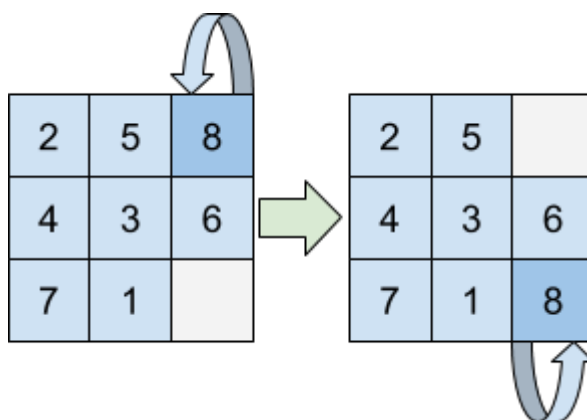
## Summary

As we hinted at in class, this assignment will concern the 8-tile puzzle—with a twist.

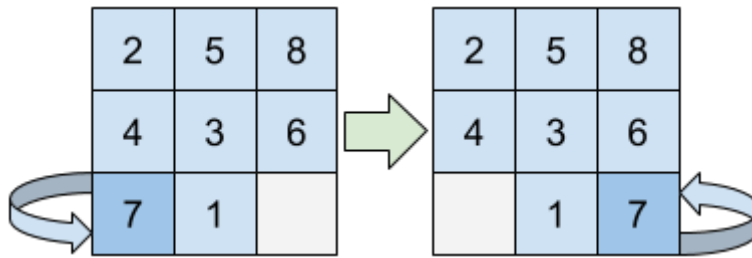
Recall the standard successors associated with a state in the typical 8-tile puzzle:



In addition to these standard moves, we will allow tiles to *wrap*—that is, tiles can slide out of the grid vertically:



or horizontally:



and back into the puzzle on the opposite side.

You can think of the rows and columns of the puzzle as rings, where a tile that goes out of one end automatically re-enters on the other end. In topology, we call a ring-shaped thing a "torus" (hence the name of the assignment).

Given these rules for the puzzle, you will generate a state space and solve this puzzle using the **A\* search algorithm**.

## Program Specification

The code for this program should be written in Python, in a file called **torus\_puzzle.py**.

You may represent your states internally however you wish; a two-dimensional list or numpy matrix may be useful. We will provide states in a one-dimensional list of integers, with the empty space represented as 0. We will represent the initial state above in our input as `[2,5,8,4,3,6,7,1,0]` and the successors pictured are, in order, `[2,5,8,4,3,0,7,1,6]`, `[2,5,8,4,3,6,7,0,1]`, `[2,5,0,4,3,6,7,1,8]`, and `[2,5,8,4,3,6,0,1,7]`.

You are not required to use any particular libraries or packages for this assignment, nor are any libraries or packages prohibited.

Any code that you do not personally write **should be cited** as you would cite a quotation in an essay; that is, with its author and source in as complete a format as possible. Code used without citation will be considered plagiarism and violates our policy of appropriate academic conduct. For example:

```
''' Original author: gottani
    Source: https://gottani.tumblr.com/post/181973941339 \_ \(https://gottani.tumblr.com/post/181973941339\)
    The following function was translated from the original Javascript for the current program
'''
def heuristic(state):
```

Even if you modify the code, you should still cite your original inspiration (and any assisting TAs or peer mentors!).

## Goal State

The goal state of the puzzle is `[1, 2, 3, 4, 5, 6, 7, 8, 0]`, or visually:

1	2	3
4	5	6
7	8	

## Heuristic

Since we are using the A\* search algorithm, we should agree on a heuristic. For simplicity, we will use the count of tiles which are not in their goal spaces.


In the images provided in the summary, the  $h()$  value for the second and third states is 5 (tiles 4, 6, and 7 are in their goal locations). The first standard example and the horizontal wrapping example each have an  $h()$  value of 6, since we moved tiles out of their goal locations.

## Functions

For this program you should write at least **two (2)** Python functions:

1. **print\_succ(state)** — given a state of the puzzle, represented as a single list of integers with a 0 in the empty space, print to the console all of the possible successor states
2. **solve(state)** — given a state of the puzzle, perform the A\* search algorithm and print the path from the current state to the goal state

You may, of course, add any other functions you see fit, but these two functions must be present and work as described here.

**You must also use** (and complete) the [PriorityQueue class](#) . We will use the methods provided there to test the intermediate steps of your implementation.

### Generate Successors

This function should print out the four successor states of the initial state, as well as their heuristic value according to the function described above.

```
>>> print_succ([1,2,3,4,5,0,6,7,8])
[1, 2, 0, 4, 5, 3, 6, 7, 8] h=4
[1, 2, 3, 0, 5, 4, 6, 7, 8] h=4
[1, 2, 3, 4, 0, 5, 6, 7, 8] h=4
[1, 2, 3, 4, 5, 8, 6, 7, 0] h=3
```

We **do** require that these be printed in a specific order: if you consider the state to be a nine-digit integer, the states should be sorted in *ascending* order. Conveniently, if you ask Python to sort one-dimensional arrays, it will adhere to this order by default; don't do more work than you have to:

```
>>> lists = [[1,2,3,4,5,8,6,7,0],
... [1,2,3,0,5,4,6,7,8],
... [1,2,3,4,0,5,6,7,8],
... [1,2,0,4,5,3,6,7,8]]
>>> sorted(lists)
=> [[1, 2, 0, 4, 5, 3, 6, 7, 8],
     [1, 2, 3, 0, 5, 4, 6, 7, 8],
     [1, 2, 3, 4, 0, 5, 6, 7, 8],
     [1, 2, 3, 4, 5, 8, 6, 7, 0]]
```

We strongly recommend that you create a separate helper function to create the list of successors of this function according to your internal representation, for use in the puzzle solver.

## Priority Queue

Now is a good time to add [our priority queue skeleton](#)  to your code and complete the enqueue method.

Everything we have provided so far should allow you to create a priority queue, add a dictionary to the queue, and see the contents of the queue:

```
>>> pq = PriorityQueue()
>>> pq.enqueue({'state': [1, 2, 3, 4, 5, 0, 6, 7, 8], 'h': 3, 'g': 0, 'parent': None, 'f': 3})
>>> print(pq)
{'state': [1, 2, 3, 4, 5, 0, 6, 7, 8], 'h': 3, 'g': 0, 'parent': None, 'f': 3}
```

If I pop from this priority queue, the following should happen:

```
>>> pq.is_empty()
=> True
```

And if I generate the successors of that state and add them to the priority queue, the following should happen (output will appear all on the same line; I've added line breaks here for readability):

```
>>> # assume i have generated and enqueued the successors
>>> print(pq)
{'state': [1, 2, 0, 4, 5, 3, 6, 7, 8], 'h': 4, 'g': 1, 'parent': {'state': [1, 2, 3, 4, 5, 0, 6, 7, 8], 'h': 3, 'g': 0, 'parent': None, 'f': 3}, 'f': 5}
{'state': [1, 2, 3, 0, 5, 4, 6, 7, 8], 'h': 4, 'g': 1, 'parent': {'state': [1, 2, 3, 4, 5, 0, 6, 7, 8], 'h': 3, 'g': 0, 'parent': None, 'f': 3}, 'f': 5}
{'state': [1, 2, 3, 4, 0, 5, 6, 7, 8], 'h': 4, 'g': 1, 'parent': {'state': [1, 2, 3, 4, 5, 0, 6, 7, 8], 'h': 3, 'g': 0, 'parent': None, 'f': 3}, 'f': 5}
{'state': [1, 2, 3, 4, 5, 8, 6, 7, 0], 'h': 3, 'g': 1, 'parent': {'state': [1, 2, 3, 4, 5, 0, 6, 7, 8], 'h': 3, 'g': 0, 'parent': None, 'f': 3}, 'f': 4}
```

What you will need to do is complete the `enqueue()` method such that adding a state to the priority queue that is already present in the queue will not increase the size of the queue but rather update the cost (in our example and the lecture slides, `g`) to the *lower* of the two options.

```
>>> pq.enqueue('' the first child, but with a g of 0 and an f of 4 '')
>>> print(pq)
{'state': [1, 2, 0, 4, 5, 3, 6, 7, 8], 'h': 4, 'g': 0, 'parent': {'state': [1, 2, 3, 4, 5, 0, 6, 7, 8], 'h': 3, 'g': 0, 'parent': None, 'f': 3}, 'f': 4}
{'state': [1, 2, 3, 0, 5, 4, 6, 7, 8], 'h': 4, 'g': 1, 'parent': {'state': [1, 2, 3, 4, 5, 0, 6, 7, 8], 'h': 3, 'g': 0, 'parent': None, 'f': 3}, 'f': 5}
{'state': [1, 2, 3, 4, 0, 5, 6, 7, 8], 'h': 4, 'g': 1, 'parent': {'state': [1, 2, 3, 4, 5, 0, 6, 7, 8], 'h': 3, 'g': 0, 'parent': None, 'f': 3}, 'f': 5}
{'state': [1, 2, 3, 4, 5, 8, 6, 7, 0], 'h': 3, 'g': 1, 'parent': {'state': [1, 2, 3, 4, 5, 0, 6, 7, 8], 'h': 3, 'g': 0, 'parent': None, 'f': 3}, 'f': 4}
```

## Solve the Puzzle

This function should print the solution path from the provided initial state to the goal state, along with the heuristic values of each intermediate state according to the function described above, and total moves taken to reach the state. Recall that our cost function  $g(n)$  is the total number of moves so far, and every valid successor has an additional cost of 1.

```
>>> solve([4,3,8,5,1,6,7,2,0])
[4, 3, 8, 5, 1, 6, 7, 2, 0] h=6 moves: 0
[4, 3, 0, 5, 1, 6, 7, 2, 8] h=6 moves: 1
[4, 0, 3, 5, 1, 6, 7, 2, 8] h=5 moves: 2
[4, 1, 3, 5, 0, 6, 7, 2, 8] h=5 moves: 3
[4, 1, 3, 0, 5, 6, 7, 2, 8] h=4 moves: 4
[0, 1, 3, 4, 5, 6, 7, 2, 8] h=3 moves: 5
[1, 0, 3, 4, 5, 6, 7, 2, 8] h=2 moves: 6
[1, 2, 3, 4, 5, 6, 7, 0, 8] h=1 moves: 7
[1, 2, 3, 4, 5, 6, 7, 8, 0] h=0 moves: 8
Max queue length: 46
```

The max queue length is tracked by our priority queue implementation (as the variable `max_len`), and displaying it may be useful for debugging purposes, but it is **not** required output.

Note: please do not use `exit()` or `quit()` to end your function when you find a path, as this will cause Python to close entirely.

## Additional Examples

For your successor function:

```
>>> print_succ([8,7,6,5,4,3,2,1,0])
[8, 7, 0, 5, 4, 3, 2, 1, 6] h=8
[8, 7, 6, 5, 4, 0, 2, 1, 3] h=8
[8, 7, 6, 5, 4, 3, 0, 1, 2] h=8
[8, 7, 6, 5, 4, 3, 2, 0, 1] h=8
```

For your solution function (be aware, that last one may take a while):

```
>>> solve([1,2,3,4,5,6,7,0,8])
[1, 2, 3, 4, 5, 6, 7, 0, 8]  h=1  moves: 0
[1, 2, 3, 4, 5, 6, 7, 8, 0]  h=0  moves: 1
Max queue length: 4
>>> solve([8,7,6,5,4,3,2,1,0])
[8, 7, 6, 5, 4, 3, 2, 1, 0]  h=8  moves: 0
[8, 7, 0, 5, 4, 3, 2, 1, 6]  h=8  moves: 1
[8, 7, 3, 5, 4, 0, 2, 1, 6]  h=7  moves: 2
[8, 7, 3, 0, 4, 5, 2, 1, 6]  h=7  moves: 3
[8, 7, 3, 4, 0, 5, 2, 1, 6]  h=6  moves: 4
[8, 0, 3, 4, 7, 5, 2, 1, 6]  h=6  moves: 5
[8, 1, 3, 4, 7, 5, 2, 0, 6]  h=6  moves: 6
[8, 1, 3, 4, 7, 5, 0, 2, 6]  h=6  moves: 7
[0, 1, 3, 4, 7, 5, 8, 2, 6]  h=6  moves: 8
[1, 0, 3, 4, 7, 5, 8, 2, 6]  h=5  moves: 9
[1, 2, 3, 4, 7, 5, 8, 0, 6]  h=4  moves: 10
[1, 2, 3, 4, 0, 5, 8, 7, 6]  h=4  moves: 11
[1, 2, 3, 4, 5, 0, 8, 7, 6]  h=3  moves: 12
[1, 2, 3, 4, 5, 6, 8, 7, 0]  h=2  moves: 13
[1, 2, 3, 4, 5, 6, 0, 7, 8]  h=2  moves: 14
[1, 2, 3, 4, 5, 6, 7, 0, 8]  h=1  moves: 15
[1, 2, 3, 4, 5, 6, 7, 8, 0]  h=0  moves: 16
Max queue length: 27944
```

## Torus Puzzle

Criteria	Ratings		Pts
print_succ() works for center gap	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
print_succ() works for vertical wraps	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
print_succ() works for horizontal wraps	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
Successors printed in correct order	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
Priority queue class present as specified	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
Priority queue updates correctly	25.0 pts Full Marks	0.0 pts No Marks	25.0 pts
solve() correctly terminates for a one-move solution	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
solve() correctly finds a two-move solution	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
solve() correctly finds a solution for more complicated initial states	20.0 pts Full Marks	0.0 pts No Marks	20.0 pts
Output of solve() correctly formatted	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
Total Points: 100.0			