# HW8: A* Search

**Due** Nov 23, 2021 by 10:59am **Points** 100 **Submitting** a file upload **File Types** zip **Available** Nov 15, 2021 at 10pm - Nov 23, 2021 at 10:59am

This assignment was locked Nov 23, 2021 at 10:59am.

**Update Nov 19**: Max queue length output will not be checked when grading. The provided student testing script is updated to reflect this.

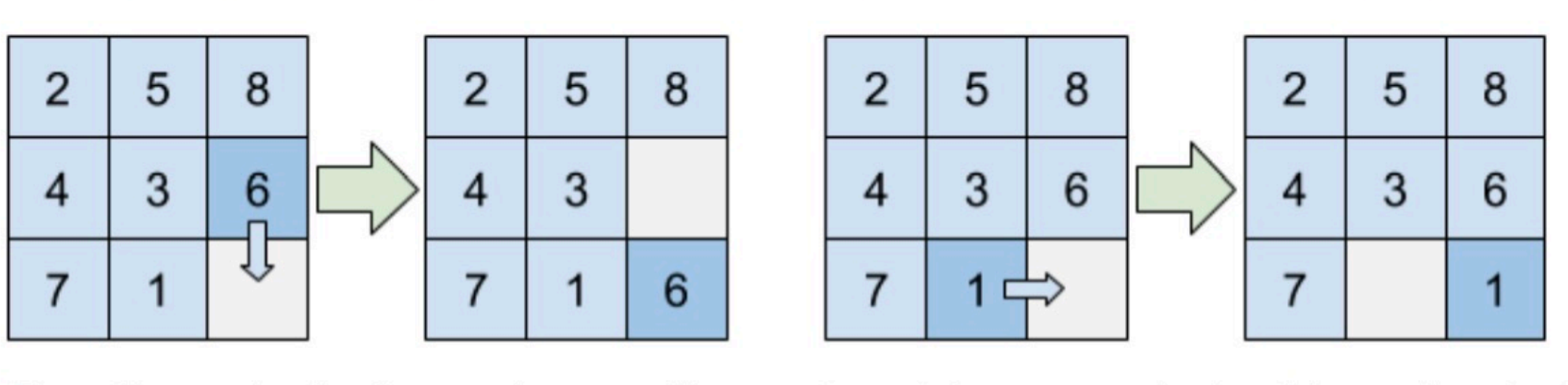
## Assignment Goals

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

## Summary

This assignment is about solving the 8-tile puzzle we have discussed in class. The 8-tile puzzle was invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3x3 grid with 8 tiles labeled 1 through 8 and an empty grid. The goal is to rearrange the tiles so that they are in order.

You solve the puzzle by moving the tiles around. For each step, you can only move one of the neighbor tiles (left, right, top, bottom) into an empty grid. And all tiles must stay in the 3x3 grid. An example is shown in the picture below. In this example, there are only 2 valid moves, i.e., either moving 6 down or moving 1 right.



Given these rules for the puzzle, you will generate a state space and solve this puzzle using the **A* search algorithm**. Note that not all 8-tile puzzles are solvable. For this assignment, it is safe to assume that the input puzzle is always solvable.

## Program Specification

The code for this program should be written in Python, in a file called **funny_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]`.

In this assignment, you will need to implement a priority queue. We highly recommend to use package `heapq` for the implementation. You should refer to heapq if you are not familiar with it.

Any code that you do not personally write **should be cited** asyou 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:

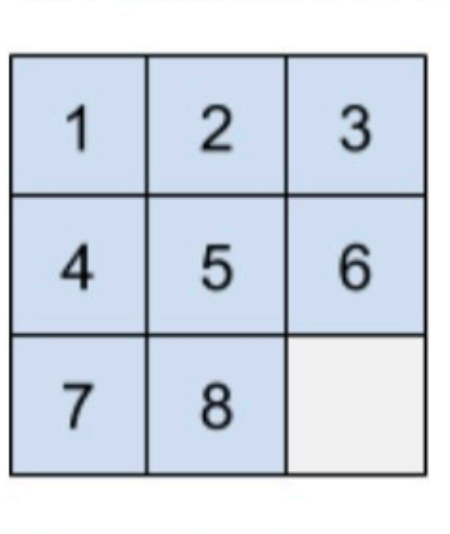
```
''' Original author: ecto
    Source: https://codepen.io/ectophage/pen/dZwaPW

    The following function was translated from the original Javascript for the current program
'''
def addRandomTile():
```

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:



### Heuristic

Since we are using the A* search algorithm, we need a heuristic function. Recall the Manhattan distance mentioned in lecture (the l1-norm). **We will use the sum of Manhattan distance of each tile to its goal position as our heuristic function.**

The Manhattan distance in this case is the sum of the absolute difference of their x co-ordinates and y co-ordinates.

In our first example puzzle (`[2, 5, 8, 4, 3, 6, 7, 1, 0]`) , the h() value for the original state is 10. This is computed by calculating the Manhattan distance of each title and summing them. Specifically, tiles 4/6/7 are already in place, thus they have 0 distances.

Tile 1 has a Manhattan distance of 3 (`manhattan([2,1], [0,0]) = abs(2-0) + abs(1-0) = 3`), tiles 2/3/5/8 have distances of 1/2/1/3, respectively.

*Caution*: do not count the distance of tile '0' since it is actually not a tile but an empty grid.

### 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.

#### Print Successors

This function should print out the successor states of the initial state, as well as their heuristic value according to the function described above. The number of successor states depends on the current state. There could be 2 (empty grid is at corner), 3 (empty grid is at middle of a boundary), or 4 (empty grid is at the center of a boundary) successors.

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

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

#### Priority Queue

Now is a good time to implement the priority queue in your code.

We recommend you use the python library **heapq** to create your priority queue. Here is a quick example:

```
>>> import heapq
>>> pq = []
>>> heapq.heappush(pq,(5, [1, 2, 3, 4, 5, 0, 6, 7, 8], (0, 5, -1)))
>>> print(pq)
[(5, [1, 2, 3, 4, 5, 0, 6, 7, 8], (0, 5, -1))]
```

The code will push an item ([1, 2, 3, 4, 5, 0, 6, 7, 8], (0, 5, -1)) with priority 5 into the queue. This format follows **(g+h, state, (g, h, parent_index))** representing both the cost, state and the parent index in A* search (a parent index of -1 denotes the initial state, without any parent). For more details, please refer to the documentation of heapq
.

To get the final path, for each element in the priority queue we need to remember its parent state. Remember to store the state when you pop it from the priority queue, so you could refer to it later when you generate the final path.

Here is how you can pop from a priority queue,

```
>>> b = heapq.heappop(pq)
>>> print(b)
(5, [1, 2, 3, 4, 5, 0, 6, 7, 8], (0, 5, -1))
>>> print(pq)
[]
```

The priority queue is stored in a list and you can print its items (with the associated priority) by using

```
>>> # assume that you have generated and enqueued the successors
>>> print(*pq, sep='\n')
(7, [1, 2, 0, 4, 5, 3, 6, 7, 8], (1, 6, 0))
(7, [1, 2, 3, 4, 0, 5, 6, 7, 8], (1, 6, 0))
(7, [1, 2, 3, 4, 5, 8, 6, 7, 0], (1, 6, 0))
```

Note that the heappush maintains the priority in ascending order, i.e., heappop will always pop the element with the smallest priority.

We require that the states **with the same cost (priority)** to be popped in a specific order: if you consider the state to be a nine-digit integer, the states should be sorted in *ascending* order - just like we mentioned above. If you follow the format in pq as shown above, heapq will automatically take care of this and you do not need more work.

#### 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=10 moves: 0
[4, 3, 8, 5, 1, 0, 7, 2, 6] h=11 moves: 1
[4, 3, 0, 5, 1, 8, 7, 2, 6] h=10 moves: 2
[4, 0, 3, 5, 1, 8, 7, 2, 6] h=9 moves: 3
[4, 1, 3, 5, 0, 8, 7, 2, 6] h=8 moves: 4
[4, 1, 3, 5, 8, 0, 7, 2, 6] h=7 moves: 5
[4, 1, 3, 5, 8, 6, 7, 2, 0] h=6 moves: 6
[4, 1, 3, 5, 8, 6, 7, 0, 2] h=7 moves: 7
[4, 1, 3, 5, 0, 6, 7, 8, 2] h=6 moves: 8
[4, 1, 3, 0, 5, 6, 7, 8, 2] h=5 moves: 9
[0, 1, 3, 4, 5, 6, 7, 8, 2] h=4 moves: 10
[1, 0, 3, 4, 5, 6, 7, 8, 2] h=3 moves: 11
[1, 3, 0, 4, 5, 6, 7, 8, 2] h=4 moves: 12
[1, 3, 6, 4, 5, 0, 7, 8, 2] h=5 moves: 13
[1, 3, 6, 4, 5, 2, 7, 8, 0] h=4 moves: 14
[1, 3, 6, 4, 5, 2, 7, 0, 8] h=5 moves: 15
[1, 3, 6, 4, 0, 2, 7, 5, 8] h=6 moves: 16
[1, 3, 6, 4, 2, 0, 7, 5, 8] h=5 moves: 17
[1, 3, 0, 4, 2, 6, 7, 5, 8] h=4 moves: 18
[1, 0, 3, 4, 2, 6, 7, 5, 8] h=3 moves: 19
[1, 2, 3, 4, 0, 6, 7, 5, 8] h=2 moves: 20
[1, 2, 3, 4, 5, 6, 7, 0, 8] h=1 moves: 21
[1, 2, 3, 4, 5, 6, 7, 8, 0] h=0 moves: 22
Max queue length: 394
```

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, 6, 5, 4, 0, 2, 1, 3] h=17
[8, 7, 6, 5, 4, 3, 2, 0, 1] h=17
```

For your solution function:

```
>>> 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: 3
```

## Additional hints

For some complicated cases, your solve function could take a very long time to get an answer, or maybe even the entire program breaks. It is recommended that you don't explore any state that has been visited before (justified because the Manhattan distance heuristic here is consistent ). To do this, you can use a set to track which states have been visited already.

## Submission Notes

Please submit your files in a zip file named **hw8_<netid>.zip** , where you replace <netid> with your netID (your wisc.edu login). Inside your zip file, there should be **only** one file named: **funny_puzzle.py** . Do not submit a Jupyter notebook .ipynb file. Failure to adhere to this will be **penalized (10 pts)**.

Be sure to **remove all testing/debugging output** before submission; your functions should run silently except required I/O. Failure to adhere to this will be **penalized (10pts)** .

**We provide a** testing script for you to check the output format. Unzip the file and put your funny_puzzle.py in the same directory as test.py; then run "python3 test.py". If you pass all the tests, you will get 15 points. Note that the actual grading test cases are different and this is only intended for a sanity check.

Make sure to test your submission on the **CS Linux** machines!

**Regrading:** Any regrading request needs to be submitted within 72 hours from the grade release. The regrading requests should be about more than 10 point deduction due to output format mismatch that can be fixed by modifying one or two lines in the code, and 10 points will be deducted for such output format mismatch. Those requests due to other mistakes (e.g., logic flow of the code) are not accepted. If a regrading request isn't justifiable (the initial grade is correct and clear, subject to the instructors' judgment), the request for regrading will be penalized (10 pts).

This assignment is due on **Nov 23 at 10:59 AM** . It is preferable to first submit a version well before the deadline (at least one hour before) and check the content/format of the submission to make sure it's the right version. Then, later update the submission until the deadline if needed.

**HW8 Rubric (1)**

| Criteria | Ratings | | Pts |
|---|---|---|---|
| print_succ 1 | 5 pts Full Marks | 0 pts No Marks | 5 pts |
| print_succ 2 | 5 pts Full Marks | 0 pts No Marks | 5 pts |
| print_succ 3 | 5 pts Full Marks | 0 pts No Marks | 5 pts |
| print_succ 4 | 5 pts Full Marks | 0 pts No Marks | 5 pts |
| solve 1 | 10 pts Full Marks | 0 pts No Marks | 10 pts |
| solve 2 | 10 pts Full Marks | 0 pts No Marks | 10 pts |
| solve 3 | 10 pts Full Marks | 0 pts No Marks | 10 pts |
| solve 4 | 10 pts Full Marks | 0 pts No Marks | 10 pts |
| solve complicated 1 | 20 pts Full Marks | 0 pts No Marks | 20 pts |
| solve complicated 2 | 20 pts Full Marks | 0 pts No Marks | 20 pts |
| | | | Total Points: 100 |