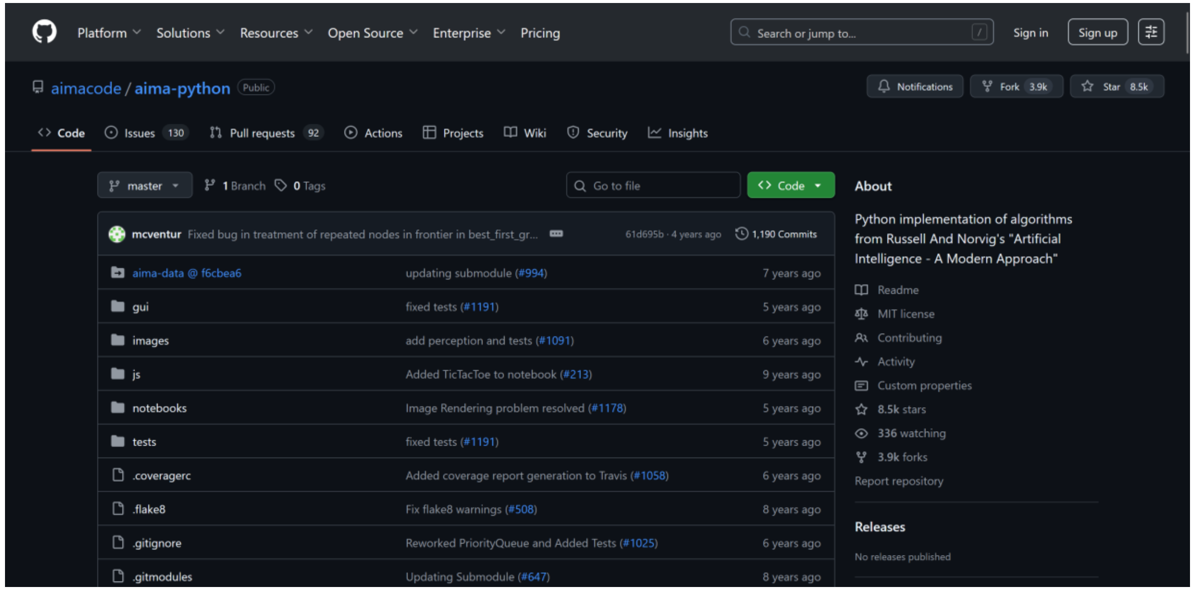
AIMA – The Official Repo for the Textbook

- https://github.com/aimacode/aima-python
- You're free to use any/all code from it to help develop your experimental platform for Project 1
- Many useful search functions can be found in: https://github.com/aimacode/aima-python/blob/master/search.py



Getting Started Guide

1. Installing AIMA
2. Defining a Problem
3. Applying A*
4. Setting up your TSP

Installing AIMA

- First, make sure you have a place to run python code from the command line, where you can also pull from git and install packages.
- Then just follow the Installation Guide at: https://github.com/aimacode/aima-python
- Don't worry if a few of the tests fail, one did for me

Installation Guide

To download the repository:

```
git clone https://github.com/aimacode/aima-python.git
```

Then you need to install the basic dependencies to run the project on your system:

```
cd aima-python
pip install -r requirements.txt
```

You also need to fetch the datasets from the `aima-data` repository:

```
git submodule init
git submodule update
```

Wait for the datasets to download, it may take a while. Once they are downloaded, you need to install `pytest`, so that you can run the test suite:

```
pip install pytest
```

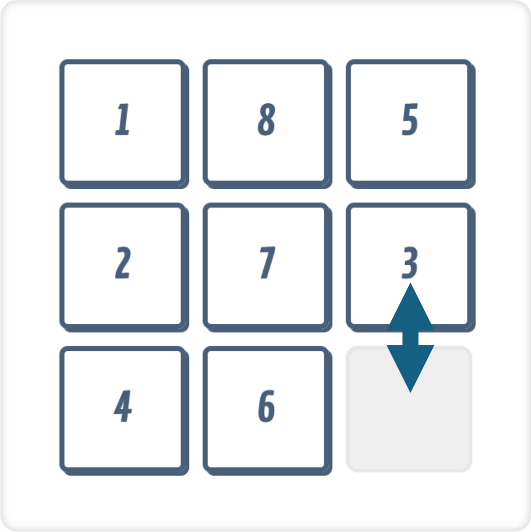
Then to run the tests:

```
py.test
```

And you are good to go!

Defining a Problem

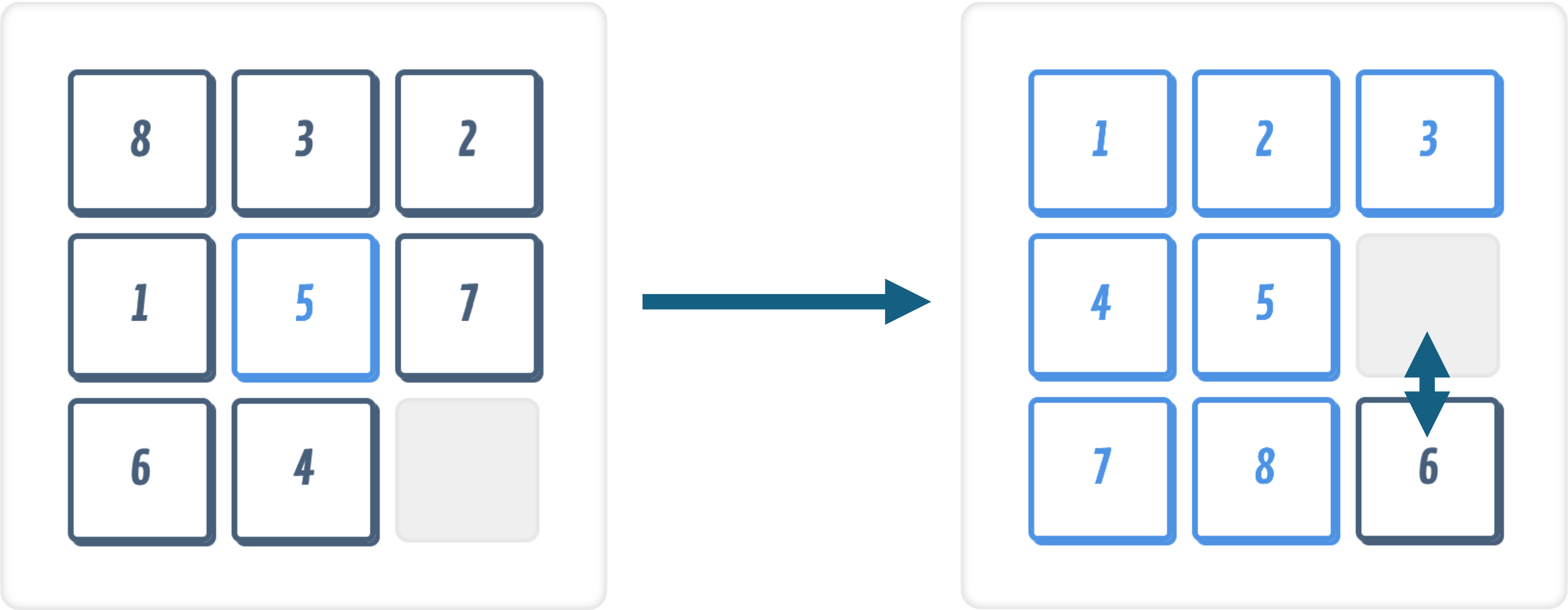
- See search.py Line 15 for the abstract “Problem” class.
- Define simple functions to describe the states, actions, successors, etc.
- Once your Problem is defined, you can run several search algorithms with it right out of the box!
- See the “EightPuzzle” on Line 426 for an example Problem.




```python
425
426 class EightPuzzle(Problem):
427     """ The problem of sliding tiles numbered from 1 to 8 on a 3x3 board, where one of the
428     squares is a blank. A state is represented as a tuple of length 9, where  element at
429     index i represents the tile number  at index i (0 if it's an empty square) """
430
431     def __init__(self, initial, goal=(1, 2, 3, 4, 5, 6, 7, 8, 0)):
432         """ Define goal state and initialize a problem """
433         super().__init__(initial, goal)
434
435     def find_blank_square(self, state):
436         """Return the index of the blank square in a given state"""
437
438         return state.index(0)
439
440     def actions(self, state):
441         """ Return the actions that can be executed in the given state.
442         The result would be a list, since there are only four possible actions
443         in any given state of the environment """
444
445         possible_actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
446         index_blank_square = self.find_blank_square(state)
447
448         if index_blank_square % 3 == 0:
449             possible_actions.remove('LEFT')
450         if index_blank_square < 3:
```

The Eight Puzzle

- 3x3 grid of number tiles, with one open slot where you can slide neighboring tiles into
- Swap tiles until they are all in order (goal state)
- Example: https://sliding.toys/mystic-square/8-puzzle/



Eight Puzzle Problem

- __init__
  - Receives the initial state and goal state (if applicable)
  - This problem has one distinct goal state (is this true for TSP?)
  - States are Tuples here
  - Don’t use Lists as States because they are not hashable!
- A helper function
- actions
  - Given a state, return a list of the possible actions
  - Describe the actions however you see fit
  - These actions are directions that can swap with the blank spot

```python
class EightPuzzle(Problem):
    """ The problem of sliding tiles numbered from 1 to 8 on a 3x3 board, where one of the
    squares is a blank. A state is represented as a tuple of length 9, where  element at
    index i represents the tile number  at index i (0 if it's an empty square) """

    def __init__(self, initial, goal=(1, 2, 3, 4, 5, 6, 7, 8, 0)):
        """ Define goal state and initialize a problem """
        super().__init__(initial, goal)

    def find_blank_square(self, state):
        """Return the index of the blank square in a given state"""

        return state.index(0)

    def actions(self, state):
        """ Return the actions that can be executed in the given state.
        The result would be a list, since there are only four possible actions
        in any given state of the environment """

        possible_actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
        index_blank_square = self.find_blank_square(state)

        if index_blank_square % 3 == 0:
            possible_actions.remove('LEFT')
        if index_blank_square < 3:
            possible_actions.remove('UP')
        if index_blank_square % 3 == 2:
            possible_actions.remove('RIGHT')
        if index_blank_square > 5:
            possible_actions.remove('DOWN')

        return possible_actions
```

Eight Puzzle Problem

- result
  - Apply the action and return the resulting state
  - Again: Tuples, not Lists
  - Swap the blank tile in the specified direction
- goal_test
  - Simple for this problem
  - What would it look like for TSP?
- Another helper function
  - Feel free to define any extra functions that might help
- h (heuristic function)
  - Compute the heuristic for a given **node**
  - Just call “node.state” to get the state back out
  - Required for A*

```python
    def result(self, state, action):
        """ Given state and action, return a new state that is the result of the action.
        Action is assumed to be a valid action in the state """

        # blank is the index of the blank square
        blank = self.find_blank_square(state)
        new_state = list(state)

        delta = {'UP': -3, 'DOWN': 3, 'LEFT': -1, 'RIGHT': 1}
        neighbor = blank + delta[action]
        new_state[blank], new_state[neighbor] = new_state[neighbor], new_state[blank]

        return tuple(new_state)

    def goal_test(self, state):
        """ Given a state, return True if state is a goal state or False, otherwise """

        return state == self.goal

    def check_solvability(self, state):
        """ Checks if the given state is solvable """

        inversion = 0
        for i in range(len(state)):
            for j in range(i + 1, len(state)):
                if (state[i] > state[j]) and state[i] != 0 and state[j] != 0:
                    inversion += 1

        return inversion % 2 == 0

    def h(self, node):
        """ Return the heuristic value for a given state. Default heuristic function used is
        h(n) = number of misplaced tiles """

        return sum(s != g for (s, g) in zip(node.state, self.goal))
```

Applying A*

- Once your Problem (and heuristic) are defined, you can run AIMA's A* implementation with just a few lines of code:

```
from time import time
from search import EightPuzzle, astar_search

INITIAL_STATE = (8, 3, 2, 4, 5, 7, 1, 6, 0)
EP = EightPuzzle(INITIAL_STATE)
t0 = time()
astar_search(EP, display=True)
print('Solved in %f seconds'%(time()-t0))
```

- See **Piazza** for a demo script
- Don't be surprised if takes a few minutes to solve!
- Bonus tip: add "print(node.state)" to line 280 of search.py to show the states as A* explores them