Implementation FAQs (courtesy Yas, TA in Spring 2017)

Q. My search algorithm seems correct but is too slow. How can I reduce its running time?

A. Search algorithm is perhaps one of the best learning materials for computational complexity and Python's idiosyncrasies. There are four dos and don'ts:

1. Don't store possibly large data member such as solution `path` in search tree node class. Instead, rethink what operation should be fast.

Explanation: Storing a path from the root node in each node class achieves O(1) lookup time at the expense of O(n) creation time. For example, if the current state is visited after 60,000 intermediate states, the current state has to allocate a list of 60,000 elements, and each of the children states have to allocate a list of 60,001 elements. This would soon use up physical memory, and typically your machine's Operating System kills the search process.

A key observation is that **in the case of search algorithm, path lookup operation is executed just once after search finishes. Thus, the lookup operation is fine to be slow**. You might consider another data structure having O(n) lookup time for solution path but requiring O(1) operations during search.

2. Don't be satisfied by just using `list` as frontier. Instead, design your `Frontier` class which works faster.

Explanation: one major bottleneck of `list`, `dequeue`, or `queue` class in Python is that their **membership testing operation** is O(n). The membership testing speed is critical for search algorithm because that operation is executed for every child state. Coming up with using such list-like data structures is a good first step, but for using it with reasonable execution time, you might need one more trick.

Note that pseudocode in lecture slides does not necessarily reflect implementation details (i.e. time and space). Rather, it conceptually shows the algorithm's inputs, processing orders, and outputs. One of your missions in this assignment is to make the "frontier" thing into a reality by using Python's "low-level" primitives.

3. Don't use O(n) operation when you have another faster way to do the same thing.

Explanation: Roughly speaking, **if you set one O(n) operation under your `for neighbor in neighbors` loop, your code will be highly likely to exceed grading time limit**. In other words, it happens that your code executes drastically faster when you fix just one line of your code.

For example, merging two **set**s and checking an element is in the merged set is an expensive operation, while checking an element is in one of the two sets are O(1) operation.

4. Don't use `copy.deepcopy()` for list. Instead, use `list1 = [5,6]; list2 = list(list1)` or `list2 = list1[:]`.

Explanation: `copy.deepcopy()` handles very rare recursive edge cases and is slow. When simply copying a list or other data structures, you can construct a new list by `list()` constructor or **:** operator. Some people avoid the second notation due to readability, but it's frequently used in the real world.

Q. How can I locate the slow code block?

A. The simplest way is to execute **python -m cProfile -s tottime driver.py <puzzle initial state>**.

- -m : specifies module name. cProfile is a built-in profiling Python module.
- -s : sorting by the tottime (total execution time) statistic.
- For meaning of each statistics, see this official documentation.
- The profiling result will be shown even if you stop your driver.py by Ctrl + C command.

If you would like to see the results not on stdout but on browser, cprofilev module might help. There is a well-written tutorial here.

Having said that, sometimes cProfile does not show which part of code makes search slow. Typical case is the first top entry when sorted by tottime is your entire search function (e.g. A-STAR-SEARCH, which is called only one time), and it takes 99% of your execution time.

One approach is to make a wrapper function for an operation which you think might cause the slowness. For example, if you're suspecting **`set1 | set2`**(merging two sets) operation is slow, you can make the following wrapper function:

```
def merge_two_sets(set1, set2): # O(1)? O(n)? O(len(set1) +
len(set2))?
      return set1 | set2
```

This simple trick triggers cProfile module to show how long this one-operation function takes. In my test script, 95% of execution time is actually caused by this operation. Recall that cProfile will return profiling results even if you pause your program by Ctrl + C command.

Q. Do I need to optimize my search algorithm as much as possible?

A. You don't need to squeeze your code's performance by fancy optimization techniques such as bit shifting or reducing the number of function calls (i.e. putting every operation in one function for reducing overhead of function calls). Except copy.deepcopy(), most of your design choices are about choosing best data structures in terms of time/space complexity.

Q. Is there any other test cases?

A. The following two test cases might help for your stat validation. Note that long path_to_goals are truncated and running_time/max_ram_usage are removed:

python driver.py dfs 6,1,8,4,0,2,7,3,5

path_to_goal: ['Up', 'Left', 'Down', ... , 'Up', 'Left', 'Up', 'Left']
cost_of_path: 46142
nodes_expanded: 51015
search_depth: 46142
max_search_depth: 46142

python driver.py bfs 6,1,8,4,0,2,7,3,5

path_to_goal: ['Down', 'Right', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down', 'Left', 'Up', 'Left', 'Up', 'Right', 'Right', 'Down', 'Down', 'Left', 'Left', 'Up', 'Up']
cost_of_path: 20
nodes_expanded: 54094
search_depth: 20
max_search_depth: 21

python driver.py ast 6,1,8,4,0,2,7,3,5

path_to_goal: ['Down', 'Right', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down', 'Left', 'Up', 'Left', 'Up', 'Right', 'Right', 'Down', 'Down', 'Left', 'Left', 'Up', 'Up']
cost_of_path: 20
nodes_expanded: 696
search_depth: 20
max_search_depth: 20

python driver.py dfs 8,6,4,2,1,3,5,7,0

path_to_goal: ['Up', 'Up', 'Left', ..., , 'Up', 'Up', 'Left']
cost_of_path: 9612
nodes_expanded: 9869
search_depth: 9612
max_search_depth: 9612

python driver.py bfs 8,6,4,2,1,3,5,7,0

path_to_goal: ['Left', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down', 'Left', 'Up', 'Right', 'Right', 'Up', 'Left', 'Left', 'Down', 'Right', 'Right', 'Up', 'Left', 'Down', 'Down', 'Right', 'Up', 'Left', 'Up', 'Left']
cost_of_path: 26
nodes_expanded: 166786
search_depth: 26
max_search_depth: 27

python driver.py ast 8,6,4,2,1,3,5,7,0

path_to_goal: ['Left', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down', 'Left', 'Up', 'Right', 'Right', 'Up', 'Left', 'Left', 'Down', 'Right', 'Right', 'Up', 'Left', 'Down', 'Down', 'Right', 'Up', 'Left', 'Up', 'Left']
cost_of_path: 26
nodes_expanded: 1585
search_depth: 26
max_search_depth: 26

Note that these two test cases are different from ones used for grading. Coming up with tricky test cases also help you understand search algorithm behaviors deeply.

Q. (Windows Users) Is there "resource" module in Windows?

If you use Python in Cygwin, resource module is available. If otherwise, one possible workaround is to use third-party module only if the machine is Windows:

```
import sys
if sys.platform == "win32":
    import psutil
    print("psutil", psutil.Process().memory_info().rss)
else:
    # Note: if you execute Python from cygwin,
    # the sys.platform is "cygwin"
    # the grading system's sys.platform is "linux2"
```

```
import resource
print("resource", resource.getrusage(resource.RUSAGE_SELF).ru_maxrss)
```

Note that **the values of max_ram_usage and running_time are not graded** as stated in project instruction page. These stats are only for helping your study on search algorithm's time/space metrics.

Q. Why does the example of python driver.py dfs 1,2,5,3,4,0,6,7,8 return ['Up', 'Left', 'Left'] instead of ['Up', 'Left', 'Down', ...] (a solution path with 31 moves)? We are using UDLR (Up, Down, Left, Right) order, and Down move should be executed before Left move. Isn't the ['Up', 'Left', 'Left'] solution resulted from optimization forbidden in project instruction?

No, ['Up', 'Left', 'Left'] solution does not use the forbidden optimization and is a correct answer. Compared to the simplicity of the 3-move solution path, you would notice that "nodes_expanded" statistic is extremely large (181437 states). In fact, the total reachable states in (solvable) 8-puzzle is 9!/2 = 181440 states, so this statistic suggests depth first search **constantly overlooks the goal state and expands more than 99.9% of possible states in the search space**.

Why is this happening?
Please think about the reason for one minute before reading the following explanations.

There are two facts we can read from dfs pseudocode in class slides:

Fact 1. goalTest() function is only called for states which are just popped out from frontier. In other words, goalTest() is not applied to **neighbor** states **even if the neighbor is actually the solution state** (because we prohibited such an optimization).
Fact 2. A child state (neighbor) is only pushed into frontier when it's not already in frontier (and explored). More specifically, **the goal state** is only pushed into frontier when it's not already in frontier.

Combining those two facts reaches to the conclusion: **since the goal state is already pushed into frontier (i.e. the state corresponding to ['Up', 'Left', 'Left'] move), any subsequent encounter to the solution state cannot execute frontier.push() or goalTest() and is effectively meaningless.** Thus, the dfs algorithm extensively searches through numerous states in 8-puzzle (without putting the solution state into frontier again), gradually goes back to the previously pushed states, and finally find the solution state which is pushed at the very beginning of the search.

This question would arise when you remove/forget neighbor in frontier checking in your pseudocode. And if you add the checking, you will face the slowness of membership checking. In that case, please see the first question of this page ("**Q. My search algorithm seems correct but is too slow. How can I reduce its running time?**").

Q. Why the max_search_depth of python driver.py bfs 1,2,5,3,4,0,6,7,8 is 4 even though the goal state is at depth 3?

The following figure would be useful (please ignore g and h values):