

Programming 1

[Submit Assignment](#)

Due Monday by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip and py
Available until Sep 29 at 11:59pm

COMS W4701 - Artificial Intelligence - Programming Homework 1

Heuristic Search and the n -Puzzle

Due: 11:59pm on Monday, September 25th

Total Points: 100

You are welcome to discuss the problems with other students but you must turn in your own work. Please review the academic honesty policy for this course (at the end of the syllabus page).

Create a .zip or .tgz archive containing any of the files you submit. Upload that single file to Courseworks.

The file you submit should use the followign naming convention:



YOURUNI_programming1.[zip | tgz]. For example, my uni is db2711, so my file should be named db2711_written1.zip or db2711_written1.tgz.

As a reminder, any assignments submitted late will incur a 20 point penalty. No submissions will be accepted later than 4 days after the submission deadline.

The instructions below are fairly specific and it is okay to deviate from implementation details. **However: You will be graded based on the functionality of each function. Make sure the function signatures (function names, parameter and return types/data structures) match exactly the description in this assignment.**

Introduction

In this assignment you will implement and compare different search strategies for solving the n -Puzzle, which is a generalization of the 8 and 15 puzzle to squares of arbitrary size.

First, download the file [npuzzle.py](#)  . You will modify this file by filling in various pre-defined functions step-by-step. We have chosen a "plain" Python implementation that does not make use of classes and instead uses standard built-in Python data structures. Where necessary, details about these data structures are described below.

When you run the program, you should see the following output:

```
$ python npuzzle.py
1 4 2
0 5 8
3 6 7

====BFS====
No solution found.
Total states expanded: 0.
Max frontier size: 0.
Total time: 0.000s
```

The first part of the output is a representation of an initial puzzle state. Note that the "hole" is represented by the number 0.

The remaining part of the output illustrates how the program formats information about the performance of each search algorithm on the problem. Your job is to implement various search strategies: DFS, BFS, Greedy best-first search, and A*. For the informed/heuristic search methods you will implement a simple heuristic that simply counts the number of misplaced tiles.

State Representation

Each state is specified using a tuple of tuples. The inner tuple represents each row of the puzzle. For example, the puzzle configuration above would be represented as `((1, 4, 2), (0, 5, 8), (3, 6, 7))`. You will find the following line in the program:

```
state = ((1, 4, 2),
        (0, 5, 8),
        (3, 6, 7))
```

We choose tuples instead of lists. The reason is that when implementing search algorithms we need to keep track of states that have already been explored. The most efficient way to do this is to use hashing. As in Java, Python only allows immutable objects to be hashed. Since lists are mutable, we use tuples instead, which are immutable. Another benefit of using tuples is that we have to make sure actions actually result in *new* states, rather than modifying existing state data structures. By using tuples we make sure that new states are different objects since we would not be able to modify the existing objects.

Part 1: Goal Test (10 pts)

Complete the function `goal_test(state)`, such that it returns True if the state passed to it is a goal state, and False otherwise. The state is in the format described above. For the 8-puzzle, the only goal state is `((0,1,2),(3,4,5),(6,7,8))`. You can test your function by asking Python to go to the interpreter mode (`-i` parameter) after running your program and then calling the function.

```
$ python -i npuzzle.py
....
>>> state = ((0,1,2),(3,4,5),(6,7,8))
>>> goal_test(state)
True
```

Alternatively you can modify the "main" part of the program to test the function.

Part 2: Transition Model (15 pts)

Complete the function `get_successors(state)` which takes a state as a parameter and should return a list of permitted actions and successor states

For example, if the input state is

```
((0, 4, 2),
 (1, 5, 8),
 (3, 6, 7))
```

The function should return

```
[("Left",((4, 0, 2),(1, 5, 8),(3, 6, 7))),
 ("Up", ((1, 4, 2),(0, 5, 8),(3, 6, 7)))]
```

Your function should attempt actions in the following order: "Left", "Right", "Up", "Down".

A good approach is to find the location of the "hole" first, then determine which actions can apply, and then compute the new state resulting from each action. To compute these states, you can use the `swap_cells(state, i1,j1, i2,j2)` method, which creates a new state that's a copy of `state` but with the values at positions `(i1,j1)` and `(i2,j2)` swapped.

Part 3: Breadth First Search (20 pts)

Complete the function `bfs(state)` by implementing breadth first search as described in class. The return value for the function is described at the bottom of this section.

Node Information: In class, we discussed the Node data structure. A Node in the search graph contains: The current state, the parent state, the action that led from the parent to the current state, and the cost of getting to the state. Instead of representing this information in a Node data structure for each state, the search algorithms will use three different dictionaries (Python's built-in hashmap). There is already code to initialize these hash maps in the function.

Algorithm review: As a reminder, all graph search algorithms use a frontier and an *explored* set. The frontier contains possible nodes that can be expanded in the next time step. The *explored* set contains nodes that have already been expanded. Because we are not using a Node data structure, our frontier and *explored* set contain states. All information about these states is stored in dictionaries (see above).

The algorithm initially puts the initial state in the frontier. It then repeatedly takes one of the states off the frontier. If the state passes the goal test, the algorithm recovers the solution (see below). Otherwise it adds the states to the *explored* list and expands it, putting new states into the frontier.

One small complication is that we should not allow states to be added to the frontier set multiple times. In addition to the *explored* set, it is useful to also maintain a *seen* set, that keeps track of states that have been added to the frontier list (but may not have been explored yet).

Queues and Sets in Python: For BFS, the frontier should be a queue (FIFO). In Python, you can use a list as a queue, by using the `append(x)` method to add at the end and the `pop(0)` method to pop an element from the front. For example:

```
>>> li = [1,2,3]
>>> li.append(4)
>>> li
[1, 2, 3, 4]
>>> li.pop(0)
1
>>> li
[2, 3, 4]
```

The *explored* and *seen* sets should be Python sets (which are hash sets). Here is how to use them:

```
>>> s = set()
>>> s.add(1)
>>> s.add(2)
>>> s.add(2)
>>> s
{1, 2}
>>> 1 in s
True
>>> 3 in s
False
```

Bookkeeping: In addition to exploring states, the search algorithm must also do some bookkeeping. When the successors are generated and before they are put on the agenda, make sure that the dictionaries for the parent state, and action are updated (the cost dictionary is not needed for BFS).

Recovering the Solution: Once a solution is found, your program needs to use the information in the parent and action dictionary to recover the sequence of actions leading from the initial state to the goal. We recommend that you write a separate function to recover the solution, so that you can re-use it for the other search algorithms.

Return Value: The `bfs` function should return three piece of information (technically it will return a tuple, but you can think of it as three different values).

1. A list of actions, this is the actual solution. For example:

```
['Up', 'Left', 'Left', 'Down', 'Right', 'Down', 'Right']
```

1. The total number of explored states including the initial and the goal state.
2. The maximum size of the frontier.

When all these aspects are implemented correctly, running the program will print out a solution to the sample 8-puzzle.

```
$ python npuzzle.py
1 4 2
0 5 8
3 6 7

====BFS====
Solution has 7 actions.
Total states expanded: 153.
Max frontier size: 90.
['Up', 'Left', 'Left', 'Down', 'Right', 'Down', 'Right']
Total time: 0.000s
```

Part 3: Depth First Search (10 pts)

Complete the function `dfs(state)` by implementing depth first search. The change from your BFS implementation should be minimal (i.e. use a stack instead of a queue). To retrieve the item at the end of a list you can simply use `.pop()` instead of `.pop(0)`.

Uncomment the code to run dfs in the main section of the Python program. Also try running the program with the more difficult test case puzzle (commented out).

Part 4: Heuristic Function (10 pts)

Complete the function `misplaced_heuristic(state)` that returns the number of misplaced tiles in the state.

Stretch goal (not required for full credit on this assignment): Also implement the `manhattan_heuristic` function. For each misplaced tile, compute the manhattan distance between the current position and the goal position. Then sum all distances.

Part 5: Greedy Best-First Search (20 pts)

Complete the function `best_first(state, heuristic)` by implementing greedy best-first search. In Python, functions are objects that can be passed to parameters, so we can pass any heuristic function. The implementation of `best_first` is almost identical to BFS, but instead of visiting states on the frontier in a FIFO order, we select the state with the lowest estimated cost in each step. We need to make the following changes:

- Instead of just states, the frontier should contain (cost, state) pairs.
- The cost should be computed for each state before it is placed into the frontier by calling the `heuristic(state)` function.
- We need to use the list as a priority queue. Python contains the `heapq` module, which provides the functions `heappush(li, item)` and `heappop(li)` where `li` is a list. Essentially, these functions treat the list `li` as a binary min heap and make sure items end up in the right location. `heappop(li)` returns and removes the item with the lowest value. For example:

```
>>> heappush(li, 3)
>>> heappush(li, 1)
>>> heappush(li, 2)
>>> heappop(li)
1
>>> heappop(li)
2
>>> heappop(li)
```

3

This will work with the (cost, state) pairs we keep on the frontier because pairs are compared to each other according to their first element.

Uncomment the code to run best first in the main section of the Python program. Try running the program with both puzzle configurations (easy and hard).

Part 6: A* Search (15)

Complete the function `astar(state, heuristic)` by implementing A* search. The implementation is very similar to greedy best-first search, but instead of just storing the heuristic cost with each state in the frontier, we use the combined cost: $f = \text{cost}(\text{state}) + \text{heuristic}(\text{state})$. We already know how to compute the heuristic. To get the cost, we need to do some more bookkeeping:

- Add the initial state to the cost dictionary with a cost of 0 (already done).
- Before adding each child state to the frontier, take the cost of the parent from the dictionary and add 1. This is the cost for the state. Add this cost to the cost dictionary for this state, for future reference.

Uncomment the code to run A* in the main section of the Python program. Try running the program with both puzzle configurations (easy and hard).

That's it!