CAL POLY

101 | 202 | 480

Instructor Info

Projects

Tile Driver I

Tile Driver II

Biogimmickry

Mine Shafted

Board Stupid

Course Details

Description

Assessment

Resources

Calendar

Policies

Enrollment

Code Style

Student Conduct

Accommodation

Guides

Project I: Tile Driver

Due Fri, Jul 2 by 9PM Pacific Time

To begin, use the following file: http://users.csc.calpoly.edu/~dkauffma/480/tiledriver.py

- To download locally, right-click link and choose "Save link as...".
- To download and unzip on a CSL server run:

```
wget http://users.csc.calpoly.edu/~dkauffma/480/tiledriver.py
unzip tiledriver.py
```

Description

Sliding Tile Puzzles (or N-Puzzles) are single-user games that require one to move tiles around a grid until the tiles are in a particular order. Since the tiles cannot be lifted, their movement is made possible by having one tile missing from the grid. Each move thus involves moving a tile adjacent to this empty space.

3	7	1
4		2
6	8	5

Empty Tile in Center

3	7	1
4	2	
6	8	5

Tile 2 Slides Left

3	7	1
4	2	5
6	8	

Tile 5 Slides Up

As simple as this puzzle appears, it turns out that finding an optimal (shortest-path) solution - from an arbitrary mixed tile state to the final solved state - becomes extremely difficult as the value of N, the number of tiles, increases. In fact, this problem is classified as NP-hard, which means it is at least as difficult as other known computation problems that cannot be solved in polynomial (n^x) time. For this reason, this project will only focus on small values of N.

Path Encoding and A* Search

Casey Submission System

Getting Started

Secure Shell (SSH)

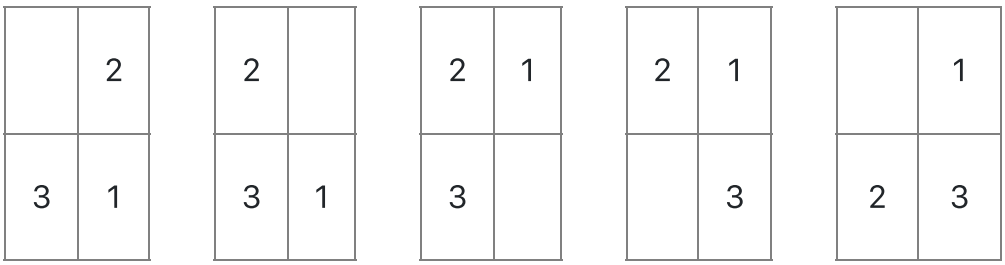
Vim Editor

While each state of a sliding tile puzzle can be represented as a sequence of integers (see implementation below), this value does not provide any information about how the user arrived there. Furthermore, a puzzle solver should display to the user the sequence of moves required to get from the initial state to the final state. We will represent such a sequence of moves as a string, where each character in the string is a single move.

To represent tile movement, we will use the same navigation keys as in Vim, specifically:

- `"H"` : Left
- `"J"` : Down
- `"K"` : Up
- `"L"` : Right

In the following sequence of diagrams, a 2x2 puzzle is solved with the path of moves that is generated under each state.



The path "HKLJ" is an optimal (shortest possible) path from the initial state to the final state. There are, of course, infinite non-optimal paths, such as "HLHLHLHLHLHKLJ" where the first move is repeatedly undone by going backward. Note that finding the optimal path using brute force is impractical; if the optimal path for a 4x4 puzzle is 20 moves long and there are an average of two valid, non-opposing moves that can be made on each turn, that would require up to 2^{20} (over one million) paths to check!

Thus, to efficiently find an optimal path, you will implement the A* algorithm, a famous search technique in the field of artificial intelligence. The algorithm works by evaluating the cost of each path using the following sum:

Path Cost = Length of Current Path + Estimated Distance to Final State

For example, the cost of the path "HKL" in the fourth diagram above is the sum of its length, 3, and the Manhattan distance to the final state, 1, for a

path cost of 4.

The algorithm begins by finding all frontier puzzle states available from the initial state. On each iteration of a loop, the frontier state with the lowest cost is explored, which requires generating all possible new puzzle states from it. Once these new states are created, the state that created them is removed from the list of frontier states. This process of exploring lowest-cost states and generating new states continues until the lowest-cost state has a distance of zero, indicating that it is the final state and an optimal path was found.

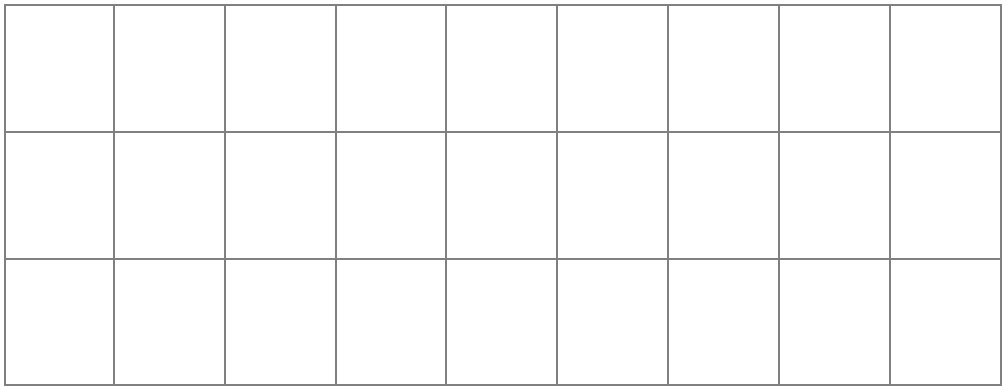
Search Heuristics

In order for A* Search to be optimal, all heuristics used must be admissible and consistent when combined. Thus, it is important that the heuristics do not count any of the same moves when computing their estimated distances.

Note that the following heuristics methods are already implemented for you in the starter file. The descriptions below serve as an example of the potential complexities in computing the heuristic for an informed search.

Manhattan Distance

Unlike Euclidean distance, which measures the distance between two points as a straight line, Manhattan distance does so by counting the number of tiles that must be traversed between two tiles in a grid. This approach is similar to how one thinks about distance when driving in a grid-like city or moving between squares on a game board. Importantly, when calculating Manhattan distance you cannot traverse diagonally.



In the grid above, the green and blue squares have a distance of 4, the blue and red squares have a distance of 6, and the green and red squares have a distance of 10.

Manhattan distance can be used to determine how far a particular state of a puzzle is from its final solved state. To do so, sum up the Manhattan distance of each tile (not including the blank tile) from its current position to its final position.

3	7	1
4		2
6	8	5

Random Puzzle State

	1	2
3	4	5
6	7	8

Final Puzzle State

In the example above, the Manhattan distance is 8 (verify this result yourself).

Linear Conflicts

The Manhattan distance used as a heuristic for solving sliding tile puzzles can be augmented by observing when two or more tiles are in linear conflict. For two tiles to be in linear conflict, they must be in their correct row or column but be in incorrect order. Since one tile will need to move out of the row or column (to get out of the other tile's way) and then move back again, each linear conflict adds 2 to the total estimated distance.

*	7	*
*	4	*
*	1	*

1, 4, and 7 are in linear conflict

7	*	*
*	4	*
*	*	1

1 and 7 can exit goal column to go around 4

*	*	1
*	4	*
7	*	*

1 and 7 must eventually re-enter goal column

A conflicting tile only needs to move out of the way of other tiles once. Furthermore, only the **minimum** number of tiles that must move in a row

or column should be counted as conflicts to ensure that conflicts are not overcounted. This heuristic may be calculated as follows:

1. Find the tiles that are in conflict for each row or column.
2. For each row or column with conflicts:
    a. Find the tile that has the highest number of conflicts relative to the other tiles. Remove this tile from consideration and increment the number of conflicts found.
    b. Repeat the step above until the remaining tiles under consideration have no conflicts with one another.

Implementation

Allowed Modules: queue

Tiles will be represented as a tuple of integers, with each integer a single tile in the puzzle in row-major order. The blank tile will be represented with the integer 0.

For example, the left-most puzzle in the first diagram above would be represented as:

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

Tiles may only be moved in a given direction if the blank tile is there. For example, a tile may only be moved up if the blank tile is above it.

Use the `get` static method in the provided `Heuristic` class to compute the combined Manhattan distance and linear conflicts heuristic for a given set of tiles.

```
Heuristic.get((0, 1, 2, 3)) -> 0
Heuristic.get((3, 2, 1, 0)) -> 6
```

Required Functions

You only need to implement the function(s) below, which will serve as the interface for the submission system's test driver. However, your program should demonstrate good decomposition with additional functions or classes.

```
solve_puzzle(tiles: Tuple[int, ...]) -> str
```

Find a path containing the optimal number of moves to the solution and return it as a string. A state is considered a solution if its Manhattan distance to an ordered tuple of integers equals zero.

Any state added to the frontier should represent a move that is both **valid** (a tile can move in that direction) and **non-opposing** (does not undo the previous move). There is at least one and at most three valid, non-opposing moves that can be made from an arbitrary state (except the first move, which may have up to four).

Your puzzle states representations (whether they be objects, dictionaries, etc.) should, in addition to the tiles, contain at least the path string used to get to the state and its estimated distance to the goal. For efficiency, the heuristic should only be computed once per state. To be under the time limit, you may need to use a priority queue (see queue.PriorityQueue) for the frontier states; if inserting objects into the queue, you will need to implement a `__lt__` method for its class so that the objects may be compared for prioritiy.

Sample Puzzles

The following 3x3 puzzles have an optimal solution of 30 moves and may be useful for testing.

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

Scoring Rubric

The score you receive on this assignment will be based on which achievements the program satisfies. The achievements are listed in the order recommended to complete them.

	Achievement	Credit
1	Solve Any 2x2 Puzzle	5%
2	Optimally Solve Any 2x2 Puzzle	20%
3	Solve 20-Length 3x3 Puzzles	10%
4	Optimally Solve 20-Length 3x3 Puzzles	20%

	Achievement	Credit
5	Solve 30-Length 3x3 Puzzles	15%
6	Optimally Solve 30-Length 3x3 Puzzles	30%

Submission

On a CSL server with `tiledriver.py` in your current directory:

Instructor	Command
Daniel Kauffman	`/home/dkauffma/casey 480 tiledriver1`