Project II: Tile Driver II

Due Mon, Jul 12 by 9PM Pacific Time

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

Description

This specification assumes the completion of Tile Driver: Part I.

Whereas the previous project involved solving puzzles, this project requires creating them. Searching for puzzles of maximum difficulty - that is, puzzles with the longest optimal path for their size - represents a very different challenge.

One strategy would be to start from a solved puzzle and move toward puzzle states with longer optimal paths; another would be to start from randomly generated puzzles, which would first need to be verified to be solvable. In either case, the length of the optimal path for any state in the search can often only be known by solving that puzzle, which is an expensive operation. To expedite the search, only select states should be solved, guiding the search through unsolved states using heuristics.

Puzzle Solvability

Some local search algorithms rely on creating many random states to cover more of the state space. Therefore, when searching for complex shuffled states, it becomes necessary to verify that a state generated randomly is solvable before using it as a starting point to search for more complex states.

Any move made from a solvable puzzle state is guaranteed to be solvable; conversely, any move made from an unsolvable puzzle state is likewise unsolvable. If the solved puzzle state is used to search for shuffled puzzle states, any resulting state will always be solvable.

The solvability of a puzzle state can be determined by counting the number of inversions in a one-dimensional list of integers representing the tiles. Inversions are defined by the number of occurrences of a pair of integers in incorrect order. For sliding tile puzzles, the blank tile is not counted toward the number of inversions.

For example, the following sequence has 10 inversions:

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

Given the width of the puzzle and the number of inversions, the solvability of any configuration of tiles can be found using the following approach.

- If the width of the puzzle is an odd number:
  - The puzzle is solvable only if the number of inversions in the puzzle is an even number
- If the width of the puzzle is an even number:
  - If the blank tile is in an even row (zero-indexed):
    - The puzzle is solvable only if the number of inversions in the puzzle is an even number
  - If the blank tile is in an odd-numbered row (zero-indexed):
    - The puzzle is solvable only if the number of inversions in the puzzle is an odd number

The logic for determining puzzle solvability is provided for you in the starter file.

Implementation

Allowed Modules: random

Design a local search algorithm for use in the required functions below. This algorithm should be based off elements from one or more of the following methods:

- Hill Climbing, which includes random restarts

- Simulated Annealing, which allows worse successors to be explored
- Beam Search, which uses multiple current states

Creativity in your design is encouraged and you may deviate from these traditional algorithms so long as the resulting method constitutes a local search; in other words, the number of active states throughout the search should remain constant. Methods that rely only on random chance and do not use search logic will not receive credit, even if successful.

The same algorithm need not be used for both search functions. Furthermore, consider the differences in search space topography between `conflict_tiles` and `shuffle_tiles`. Does one have smoother slopes? What about the frequency and range of plateaus? Not all local search methods discussed will perform equally well on both problems.

Required Functions

```
conflict_tiles(width: int, min_lc: int)
-> Tuple[int, ...]
```

Create a solvable shuffled puzzle of the given width with a minimum number of linear conflicts (ignoring Manhattan distance). Return a suitable puzzle **as soon as it is found** to satisfy the time constraint. Because the path cost of each state is irrelevant, no call to `tiledriver.solve_puzzle` will be necessary.

Use `tiledriver.Heuristic._get_linear_conflicts` from `tiledriver.py` to compute the number of linear conflicts in a state. Note that the maximum number of conflicts for a puzzle of width 3 or greater is $2 * ((width - 1)^2 + (width - 2))$.

Based on the puzzle width, your implementation will be required to produce puzzles with the following number of linear conflicts:

- 3x3: 10 conflicts
- 4x4: 14 conflicts
- 5x5: 18 conflicts

```
shuffle_tiles(width: int, min_len: int,

solve_puzzle: Callable[[Tuple[int, ...]],
str])

-> Tuple[int, ...]
```

Create a solvable shuffled puzzle of the given width with an optimal solution length equal to or greater than the given minimum length. Return a suitable puzzle **as soon as it is found** to satisfy the time constraint. Because solving the puzzle for every state in the search would be too expensive, you must search many states between solving them; determining a good frequency with which to solve states is part of the challenge.

The `solve_puzzle` argument is a callable that solves a puzzle and returns an optimal path. This callable will be provided by the instructor's test suite (primarily for students who were unable to complete the first part of the project). For those who have successfully implemented `tiledriver.solve_puzzle`, simply pass a reference to that function to `shuffle_tiles` when testing locally: `shuffle_tiles(2, 6, tiledriver.solve_puzzle)`.

Based on the puzzle width, your implementation will be required to produce puzzles with the following optimal path lengths:

- 2x2: 6 moves
- 3x3: 29 moves

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	Creates 10-Conflict 3x3 Puzzles	10%
2	Creates 14-Conflict 4x4 Puzzles	15%
3	Creates 18-Conflict 5x5 Puzzles	25%

	Achievement	Credit
4	Creates 6-Length 2x2 Puzzles	15%
5	Creates 29-Length 3x3 Puzzles	35%

Submission

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

Instructor	Command
Daniel Kauffman	`/home/dkauffma/casey 480 tiledriver2`