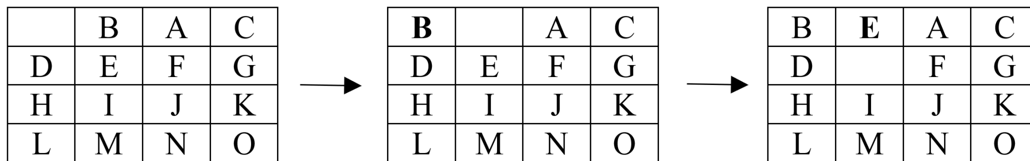
COMP 550.001 - Fall 2017
Assignment 6

This assignment includes an optional Part C. Earning half of the points will be worth half of a late day (only integral late days may be used to turn in homework late, but a partial late day can count as partial extra credit at the end of the semester), and earning at least 80% of the points will be worth a full late day. You should submit your code in a .zip or .tar.gz file to Sakai, and the analysis in a physical copy.

[Optional] Part C: Due Wednesday, November 29, 2017

Graph Search of Puzzle States

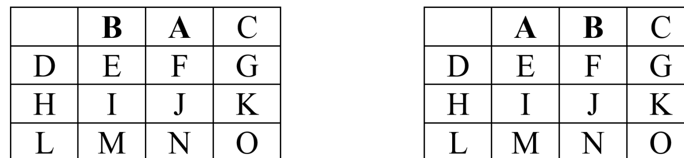
In the 15-puzzle, there are 15 lettered tiles and a blank square arranged in a 4×4 grid. Any lettered tile adjacent to the blank square can be slid into the blank. For example, a sequence of two moves is shown below:



In the leftmost configuration above, the A and B tiles are out of order. Using only legal moves, it is not possible to swap the A and B, while leaving all the other tiles in their original positions and the blank in the top left corner.

In this problem, you will take steps to prove that this is the case, and implement a graph search to find the shortest sequence of moves to the solved state if it is reachable.

Theorem. *No sequence of moves transforms the board below on the left into the board below on the right.*



a) We define the "order" of the tiles in a board to be the sequence of tiles on the board reading from the top row to the bottom row and from left to right within a row. For example, in the left board depicted in the above theorem, the order of the tiles is B, A, C, D, E, etc.

Can a row move change the order of the tiles? Prove your answer.

b) How many pairs of tiles will have their relative order changed by a column move? More formally, for how many pairs of letters L_1 and L_2 will L_1 appear earlier in the order of the tiles than L_2 before the column move and later in the order after the column move? Prove your answer correct.

c) We define an *inversion* to be a pair of letters L_1 and L_2 for which L_1 precedes L_2 in the alphabet, but L_1 appears after L_2 in the order of the tiles. For example, consider the following configuration:

A		B	D
C	G	F	E
H	I	J	K
L	M	N	O

There are exactly four inversions in the above configuration: D and C, G and F, G and E, and F and E.

What effect does a row move have on the parity of the number of inversions? Prove your answer.

d) What effect does a column move have on the parity of the number of inversions? Prove your answer.

The previous problem part implies that we must make an *odd* number of column moves in order to exchange just one pair of tiles (A and B, say). But this is problematic, because each column move also knocks the blank square up or down one row. So after an *odd* number of column moves, the blank can not possibly be back in the first row, where it belongs! Now we can bundle up all these observations and state an *invariant*, a property of the puzzle that never changes, no matter how you slide the tiles around.

Lemma. *In every configuration reachable from the position shown below, the parity of the number of inversions is the same as the parity of the row containing the blank square.*

row 0		A	B	C
row 1	D	E	F	G
row 2	H	I	J	K
row 3	L	M	N	O

You do not have to prove this lemma. Note that you could use this lemma to prove the theorem we originally set out to prove.

e) You will now use the properties proven above to write a function that determines whether a puzzle is solvable. Fill in the implementations of `getInversionCount` and `isSolvable` in the class `NumberPuzzle`.

f) Given what you now know about this puzzle, you can implement a BFS search to find the sequence of moves to get to a solved state. However, the state space of all possible puzzle states is very large. As a result, you must build the graph as you perform the search.

Each state of the puzzle represents a node in the BFS search. This is represented by the `BfsNode` class, which takes a `NumberPuzzle` as the state in its constructor. This class overrides `equals` and

`hashCode`; you can use a `HashSet` to track whether you have seen a given node before, rather than coloring it white/gray/black.

Fill in the function `getBestMoves` in `Main.java`. Your implementation should return a sequence of puzzle states from the starting state to the solved state.

g) As you might have noticed, your implementation is very slow. How long does it take to solve the 18-move puzzle state given in `main`?

h) You can speed up the BFS search substantially by searching in two directions at once. For every iteration of BFS, you can extend one node in the forward direction (starting at the initial puzzle state), and one node in the backward direction (starting at the solved puzzle state).

Implement this bi-directional search in `getBestMovesBidirectional`. As before, it should return a sequence of puzzle states from the starting state to the solved state.

i) How long does it take to solve the 18-move puzzle state using the bi-directional search?

j) Finally, fill in the table below of times in milliseconds for each of the puzzle states from 2 moves to 18 moves.

Moves	`getBestMoves` (ms)	`getBestMovesBidirectional` (ms)
2		
4		
6		
8		
10		
12		
14		
16		
18		

Part C proofs are derived from MIT's 6.042 course, Fall 2010. The overall problem is inspired by the pocket cube problem from MIT's 6.006, Fall 2011.

- Tom Leighton and Marten van Dijk, 6.042J/18.062J Mathematics for Computer Science, Fall 2010. (MIT OpenCourseWare: Massachusetts Institute of Technology), https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-042j-mathematics-for-computer-science-fall-2010/index.htm (Accessed November 15, 2017). License: Creative commons BY-NC-SA

- Erik Demaine and Srinivas Devadas, 6.006 Introduction to Algorithms, Fall 2011. (MIT OpenCourseWare: Massachusetts Institute of Technology): https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/index.htm (Accessed November 15, 2017). License: Creative commons BY-NC-SA