Knight Board

Problem 1

In general, a knight on 8x8 board

	0	1	2	3	4	5	6	7
0	*	*	*	*	*	*	*	*
1	*	*	*	*	*	*	*	*
2	*	*	K	*	*	*	*	*
3	*	*	*	*	*	*	*	*
4	*	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

can move in 8 possible ways –

	0	1	2	3	4	5	6	7
0	*	K	*	K	*	*	*	*
1	K	*	*	*	K	*	*	*
2	*	*	K	*	*	*	*	*
3	K	*	*	*	K	*	*	*
4	*	K	*	K	*	*	*	*
5	*	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

The actual set of allowed moves depends on the edge cases.

The common pattern here is that the knight successive valid knight moves differ exactly by 1 in one direction and exactly by 2 in the other direction. This with the edge cases serves as sufficient condition for a valid move

i.e.

$$|dx| = 2, |dy| = 1 \; or \; |dx| = 1, |dy| = 2,$$

this can be combined to

$$|dx| + |dy| = 3 \; and,$$

$$|dx| * |dy| = 2$$

where,
$|y|$: modulo operator; returns y if y>=0 else –y

The *isValidMove* and *isValidPosition* in the code enforce these conditions

Problem 2

To find one of the path from start to end node, I have used the standard BFS routine. BFS reaches all sqaures in the order from closest to farthest from the starting square. Hence, it not only gives a path if it exists, it also returns the shortest one. This solves for problem 3.

Problem 3

Same as Problem 2

Problem 4

Now, each move has different cost incurred depending on the square we land, we can't simply use BFS to find shortest path.

Dijkstra's algorithm is designed for exactly such a situation. Here, I have implemented the greedy implementation of algorithm which adds chessboard squares from closest to farther from the starting square. This procedure is repeated till the destination square is arrived at.

The R[ock] and B[arrier] squares are handled by the *is_valid_32* method which checks if landing square or one of the squares in the path are restricted as the case maybe.

W[ater] and L[ava] squares are handled in the *move_cost* method which returns the cost of the move 2 or 5 respectively and 1 otherwise.

Finally T[eleport] squares are handled in the while loop of the *solution4* method. This is because these square incur a zero cost and result in a move not defined a standard knight's move.