```
from puzzle import Puzzle
import copy


class GridPegSolitairePuzzle(Puzzle):
    """
    Snapshot of peg solitaire on a rectangular grid. May be solved,
    unsolved, or even unsolvable.
    """

    def __init__(self, marker, marker_set):
        """
        Create a new GridPegSolitairePuzzle self with
        marker indicating pegs, spaces, and unused
        and marker_set indicating allowed markers.

        @type marker: list[list[str]]
        @type marker_set: set[str]
                          "#" for unused, "*" for peg, "." for empty
        """
        assert isinstance(marker, list)
        assert len(marker) > 0
        assert all([len(x) == len(marker[0]) for x in marker[1:]])
        assert all([all(x in marker_set for x in row) for row in marker])
        assert all([x == "*" or x == "." or x == "#" for x in
marker_set])
        self._marker, self._marker_set = marker, marker_set

    def __eq__(self, other):
        """
        Return whether GridPegSolitairePuzzle self is equivalent to
other.
        @type self: GridPegSolitairePuzzle
        @type other: GridPegSolitairePuzzle
        @rtype: bool
        >>> grid1 = [["*", "*", "*", "*", "*"],["*", "*", "*", "*",
"*"],\
        ["*", "*", "*", "*", "*"],["*", "*", ".", "*", "*"],\
        ["*", "*", "*", "*", "*"]]
        >>> g1 = GridPegSolitairePuzzle(grid1, {"*", ".", "#"})
        >>> grid2 = [["*", "*", "*", "*", "*"],["*", "*", "*", "*",
"*"],\
        ["*", "*", "*", "*", "*"],["*", "*", ".", "*", "*"],\
        ["*", "*", "*", "*", "*"]]
        >>> g2 = GridPegSolitairePuzzle(grid2, {"*", ".", "#"})
        >>> g1.__eq__(g2)
        True
        >>> grid3 = [["*", ".", "#", "*", "*"],["*", "*", "*", "*",
"*"],\
        ["*", "*", "*", "*", "*"],["*", "*", ".", "*", "*"],\
        ["*", "*", "*", "*", "*"]]
        >>> g3 = GridPegSolitairePuzzle(grid3, {"*", ".", "#"})
        >>> g1.__eq__(g3)
        False
```

```
        """
        return (type(self) == type(other) and
                self._marker == other._marker and
                self._marker_set == other._marker_set)

    def __str__(self):
        """
        Return a human-readable string representation of
GridPegSolitairePuzzle
        self
        @type self: GridPegSolitairePuzzle
        @rtype: str

        >>> grid = [["*", "*", "*", "*", "*"],\
         ["*", "*", "*", "*", "*"]]
        >>> grid.append(["*", "*", "*", "*", "*"])
        >>> grid.append(["*", "*", ".", "*", "*"])
        >>> grid.append(["*", "*", "*", "*", "*"])
        >>> a = GridPegSolitairePuzzle(grid, {"*", ".", "#"})
        >>> print(a)
        |*|*|*|*|*|
        |*|*|*|*|*|
        |*|*|*|*|*|
        |*|*|.|*|*|
        |*|*|*|*|*|
        """

        def row_pickets(row):
            """
            Return string of a row.

            @type row: list[str]
            @rtype: str
            """
            string = ''
            for i in range(len(self._marker[0])):
                string += '|' + row[i]
            return string + '|'

        rows = [row_pickets(lst) for lst in self._marker]
        return "\n".join(rows)

    def extensions(self):
        """
        Return list of extensions of GridPegSolitairePuzzle self

        @type self: GridPegSolitairePuzzle
        @rtype: list[GridPegSolitairePuzzle]

        >>> grid = [["*", "*", "*", "*", "*"], ["*", "*", "*", "*",
"*"],\
        ["*", "*", "*", "*", "*"], ["*", "*", ".", "*", "*"],\
        ["*", "*", "*", "*", "*"], ["*", "*", "*", "*", "*"]]
        >>> a = GridPegSolitairePuzzle(grid, {"*", ".", "#"})
```

```
        >>> grid1 = [["*", "*", "*", "*", "*"], ["*", "*", ".", "*",
"*"],\
        ["*", "*", ".", "*", "*"], ["*", "*", "*", "*", "*"],\
        ["*", "*", "*", "*", "*"], ["*", "*", "*", "*", "*"]]
        >>> a1 = GridPegSolitairePuzzle(grid1, {"*", ".", "#"})
        >>> grid2 = [["*", "*", "*", "*", "*"], ["*", "*", "*", "*",
"*"],\
        ["*", "*", "*", "*", "*"], [".", ".", "*", "*", "*"],\
        ["*", "*", "*", "*", "*"], ["*", "*", "*", "*", "*"]]
        >>> a2 = GridPegSolitairePuzzle(grid2, {"*", ".", "#"})
        >>> grid3 = [["*", "*", "*", "*", "*"], ["*", "*", "*", "*",
"*"],\
        ["*", "*", "*", "*", "*"], ["*", "*", "*", ".", "."],\
        ["*", "*", "*", "*", "*"], ["*", "*", "*", "*", "*"]]
        >>> a3 = GridPegSolitairePuzzle(grid3, {"*", ".", "#"})
        >>> grid4 = [["*", "*", "*", "*", "*"], ["*", "*", "*", "*",
"*"],\
        ["*", "*", "*", "*", "*"], ["*", "*", "*", "*", "*"],\
        ["*", "*", ".", "*", "*"], ["*", "*", ".", "*", "*"]]
        >>> a4 = GridPegSolitairePuzzle(grid4, {"*", ".", "#"})
        >>> L1 = list(a.extensions())
        >>> L2 = [a1, a2, a3, a4]
        >>> len(L1) == len(L2)
        True
        >>> all([s in L2 for s in L1])
        True
        >>> all([s in L1 for s in L2])
        True
        """

        # b is the index of "."
        # b[0] is it's row index and b[1] is it's column index
        def check_left(b):
            # if the empty space is in the left two row,
            # impossible to jump left
            if b[1] < 2:
                return False
            # if the left two strings are "*", then has left extension
            elif self._marker[b[0]][b[1] - 1] == "*" \
                    and self._marker[b[0]][b[1] - 2] == "*":
                return True
            else:
                return False

        def check_right(b):
            # if the empty space is in the right two row,
            # impossible to jump right
            a = len(self._marker[b[0]]) - 1
            if a - b[1] < 2:
                return False
            elif self._marker[b[0]][b[1] + 1] == "*" \
                    and self._marker[b[0]][b[1] + 2] == "*":
                return True
            else:
```

```
                return False

        def check_up(b):
            # if the empty space is in the top two row,
            # impossible to jump up
            if b[0] < 2:
                return False
            elif self._marker[b[0] - 1][b[1]] == "*" \
                    and self._marker[b[0] - 2][b[1]] == "*":
                return True
            else:
                return False

        def check_down(b):
            # if the empty space is in the bottom two row,
            # impossible to jump down
            a = len(self._marker) - 1
            if a - b[0] < 2:
                return False
            elif self._marker[b[0] + 1][b[1]] == "*" \
                    and self._marker[b[0] + 2][b[1]] == "*":
                return True
            else:
                return False

        # Find all empty spaces
        cpk = copy.deepcopy(self._marker)
        extensions = []
        for i in range(len(cpk)):
            for j in range(len(cpk[i])):
                if cpk[i][j] == ".":
                    pt = [i, j]
                    cp1 = copy.deepcopy(cpk)
                    if check_left(pt):
                        cpk[pt[0]][pt[1] - 1] = "."
                        cpk[pt[0]][pt[1] - 2] = "."
                        cpk[pt[0]][pt[1]] = "*"
                        grid1 = cpk
                        a1 = GridPegSolitairePuzzle(grid1, {"*", ".",
"#"})
                        extensions.append(a1)
                    cp2 = copy.deepcopy(cp1)
                    if check_right(pt):
                        cp1[pt[0]][pt[1] + 1] = "."
                        cp1[pt[0]][pt[1] + 2] = "."
                        cp1[pt[0]][pt[1]] = "*"
                        grid2 = cp1
                        a2 = GridPegSolitairePuzzle(grid2, {"*", ".",
"#"})
                        extensions.append(a2)
                    cp3 = copy.deepcopy(cp2)
                    if check_up(pt):
                        cp2[pt[0] - 1][pt[1]] = "."
                        cp2[pt[0] - 2][pt[1]] = "."
```

```
                        cp2[pt[0]][pt[1]] = "*"
                        grid3 = cp2
                        a3 = GridPegSolitairePuzzle(grid3, {"*", ".",
"#"})
                        extensions.append(a3)
                    if check_down(pt):
                        cp3[pt[0] + 1][pt[1]] = "."
                        cp3[pt[0] + 2][pt[1]] = "."
                        cp3[pt[0]][pt[1]] = "*"
                        grid4 = cp3
                        a4 = GridPegSolitairePuzzle(grid4, {"*", ".",
"#"})
                        extensions.append(a4)
        return extensions

    def is_solved(self):
        """
        Check if the GridPegSolitairePuzzle self is solved.
        @type self: GridPegSolitairePuzzle
        @rtype: bool

        >>> grid = [["*", "*", "*", "*", "*"], ["*", "*", "*", "*", "*"],
\
        ["*", "*", "*", "*", "*"], ["*", "*", ".", "*", "*"],\
         ["*", "*", "*", "*", "*"]]
        >>> a = GridPegSolitairePuzzle(grid, {"*", ".", "#"})
        >>> a.is_solved()
        False
        >>> grid = [[".", ".", ".", "*", "."], [".", ".", ".", ".", "."],
\
        [".", ".", ".", ".", "."], [".", ".", ".", ".", "."]]
        >>> a = GridPegSolitairePuzzle(grid, {"*", ".", "#"})
        >>> a.is_solved()
        True
        """
        count = sum(i.count("*") for i in self._marker)
        return count == 1


if __name__ == "__main__":
    import doctest

    doctest.testmod()
    from puzzle_tools import depth_first_solve

    grid = [["*", "*", "*", "*", "*"],
            ["*", "*", "*", "*", "*"],
            ["*", "*", "*", "*", "*"],
            ["*", "*", ".", "*", "*"],
            ["*", "*", "*", "*", "*"]]
    gpsp = GridPegSolitairePuzzle(grid, {"*", ".", "#"})
    import time

    start = time.time()
```

```
solution = depth_first_solve(gpsp)
end = time.time()
print("Solved 5x5 peg solitaire in {} seconds.".format(end - start))
print("Using depth-first: \n{}".format(solution))
```

```python
from puzzle import Puzzle


class MNPuzzle(Puzzle):
    """
    An nxm puzzle, like the 15-puzzle, which may be solved, unsolved,
    or even unsolvable.
    """

    def __init__(self, from_grid, to_grid):
        """
        MNPuzzle in state from_grid, working towards
        state to_grid

        @param MNPuzzle self: this MNPuzzle
        @param tuple[tuple[str]] from_grid: current configuration
        @param tuple[tuple[str]] to_grid: solution configuration
        @rtype: None
        """
        # represent grid symbols with letters or numerals
        # represent the empty space with a "*"
        assert len(from_grid) > 0
        assert all([len(r) == len(from_grid[0]) for r in from_grid])
        assert all([len(r) == len(to_grid[0]) for r in to_grid])
        self.n, self.m = len(from_grid), len(from_grid[0])
        self.from_grid, self.to_grid = from_grid, to_grid

    def __eq__(self, other):
        """
        Return whether MNPuzzle self is equivalent to other.

        @type self: MNPuzzle
        @type other: MNPuzzle | Any
        @rtype: bool

        >>> from_grid1 = (("*", "2", "3"), ("1", "4", "5"))
        >>> to_grid1 = (("1", "2", "3"), ("4", "5", "*"))
        >>> mnp1 = MNPuzzle(from_grid1, to_grid1)
        >>> from_grid2 = (("*", "2", "3"), ("1", "4", "5"))
        >>> to_grid2 = (("1", "2", "3"), ("4", "5", "*"))
        >>> mnp2 = MNPuzzle(from_grid2, to_grid2)
        >>> mnp1 == mnp2
        True
        >>> from_grid3 = (("*", "B", "C"), ("A", "D", "E"))
        >>> to_grid3 = (("A", "B", "C"), ("D", "E", "*"))
        >>> mnp3 = MNPuzzle(from_grid3, to_grid3)
        >>> mnp1 == mnp3
        False
        """
        return (type(other) == type(self) and
                self.n == other.n and self.m == other.m and
                self.from_grid == other.from_grid and
                self.to_grid == self.to_grid)
```

```
    def __str__(self):
        """
        Return a human-readable string representation of MNPuzzle self.

        >>> from_grid1 = (("*", "2", "3"), ("1", "4", "5"))
        >>> to_grid1 = (("1", "2", "3"), ("4", "5", "*"))
        >>> mnp1 = MNPuzzle(from_grid1, to_grid1)
        >>> print(mnp1)
        |*|2|3|
        |1|4|5|
        -------
        |1|2|3|
        |4|5|*|
        """

        def row_pickets(row):
            """
            Return string of a row.

            @type row: tuple[str]
            @rtype: str
            """
            string = ''
            for i in range(self.m):
                string += '|' + row[i]
            string += '|'
            return string

        m = self.m
        divider = ["-" * (m * 2 + 1)]
        rows = [row_pickets(self.from_grid[j]) for j in range(self.n)]
        rows += divider
        rows += [row_pickets(self.to_grid[j]) for j in range(self.n)]
        return "\n".join(rows)

    def extensions(self):
        """
        Return list of extensions of MNPuzzle self

        @return: MNPuzzle
        @rtype: list[MNPuzzle]

        >>> from_grid1 = (("*", "2", "3"), ("1", "4", "5"))
        >>> to_grid1 = (("1", "2", "3"), ("4", "5", "*"))
        >>> mnp1 = MNPuzzle(from_grid1, to_grid1)
        >>> L1 = list(mnp1.extensions())
        >>> L2 = [MNPuzzle((('1', '2', '3'), ('*', '4', '5')), \
        (('1', '2', '3'), ('4', '5', '*'))), \
        MNPuzzle((('2', '*', '3'), ('1', '4', '5')),\
         (('1', '2', '3'), ('4', '5', '*')))]
        >>> len(L1) == len(L2)
        True
        >>> all([s in L2 for s in L1])
        True
```

```
        >>> all([s in L1 for s in L2])
        True
        >>> from_grid3 = (("2", "*", "3"), ("1", "4", "5"),('6', '7',
'8'))
        >>> to_grid3 = (("1", "2", "3"), ("4", "5", "6"), ('7', '8',
'*'))
        >>> mnp3 = MNPuzzle(from_grid3, to_grid3)
        >>> L3 = mnp3.extensions()
        >>> L4 = [MNPuzzle((('2', '4', '3'), ('1', '*', '5'), ('6', '7',
'8')),\
         (("1", "2", "3"), ("4", "5", "6"), ('7', '8', '*'))),\
          MNPuzzle((('*', '2', '3'), ('1', '4', '5'), ('6', '7', '8')), \
          (("1", "2", "3"), ("4", "5", "6"), ('7', '8', '*'))), \
          MNPuzzle((('2', '3', '*'), ('1', '4', '5'), ('6', '7', '8')),\
           (("1", "2", "3"), ("4", "5", "6"), ('7', '8', '*')))]
        >>> len(L3) == len(L4)
        True
        >>> all([s in L4 for s in L3])
        True
        >>> all([s in L3 for s in L4])
        True
        """

        global row_

        # swap_list is the list form of from_grid
        # row_i and col_i are the index of the empty space
        def swap_above(swap_list, row_i, col_i):
            s = swap_list
            if row_i == 0:
                return []
            else:
                s[row_i][col_i], s[row_i - 1][col_i] = \
                    s[row_i - 1][col_i], s[row_i][col_i]
                return s

        def swap_below(swap_list, row_i, col_i):
            s = swap_list
            if row_i == (len(s) - 1):
                return []
            else:
                s[row_i][col_i], s[row_i + 1][col_i] = \
                    s[row_i + 1][col_i], s[row_i][col_i]
                return s

        def swap_left(swap_list, row_i, col_i):
            s = swap_list
            if col_i == 0:
                return []
            else:
                s[row_i][col_i], s[row_i][col_i - 1] = \
                    s[row_i][col_i - 1], s[row_i][col_i]
                return s
```

```
        def swap_right(swap_list, row_i, col_i):
            s = swap_list
            if col_i == len(s[0]) - 1:
                return []
            else:
                s[row_i][col_i], s[row_i][col_i + 1] = \
                    s[row_i][col_i + 1], s[row_i][col_i]
                return s

        # find the index to "*"
        for i in range(len(self.from_grid)):
            if '*' in self.from_grid[i]:
                row_ = i
        col_ = self.from_grid[row_].index("*")
        # change from_grid to list form
        sl = [list(self.from_grid[k]) for k in range(self.n)]
        l = tuple(map(tuple, swap_left(sl, row_, col_)))
        sl = [list(self.from_grid[k]) for k in range(self.n)]
        r = tuple(map(tuple, swap_right(sl, row_, col_)))
        sl = [list(self.from_grid[k]) for k in range(self.n)]
        a = tuple(map(tuple, swap_above(sl, row_, col_)))
        sl = [list(self.from_grid[k]) for k in range(self.n)]
        b = tuple(map(tuple, swap_below(sl, row_, col_)))

        allowed_extension = []
        for element in [a, b, l, r]:
            if element != ():
                allowed_extension.append(MNPuzzle(element, self.to_grid))
            else:
                pass
        return allowed_extension

    def is_solved(self):
        """
        Return whether Puzzle self is solved.

        @type self: MNPuzzle
        @rtype: bool

        >>> from_grid1 = (("*", "2", "3"), ("1", "4", "5"))
        >>> to_grid1 = (("1", "2", "3"), ("4", "5", "*"))
        >>> mnp1 = MNPuzzle(from_grid1, to_grid1)
        >>> mnp1.is_solved()
        False
        >>> from_grid2 = (("1", "2", "3"), ("4", "5", "*"))
        >>> to_grid2 = (("1", "2", "3"), ("4", "5", "*"))
        >>> mnp2 = MNPuzzle(from_grid2, to_grid2)
        >>> mnp2.is_solved()
        True
        """
        return self.from_grid == self.to_grid


if __name__ == "__main__":
```

```python
import doctest

doctest.testmod()
target_grid = (("1", "2", "3"), ("4", "5", "*"))
start_grid = (("*", "2", "3"), ("1", "4", "5"))
from puzzle_tools import breadth_first_solve, depth_first_solve
from time import time

start = time()
solution = breadth_first_solve(MNPuzzle(start_grid, target_grid))
end = time()
print("BFS solved: \n\n{} \n\nin {} seconds".format(
    solution, end - start))
start = time()
solution = depth_first_solve((MNPuzzle(start_grid, target_grid)))
end = time()
print("DFS solved: \n\n{} \n\nin {} seconds".format(
    solution, end - start))
```

```
class Puzzle:
    """"
    Snapshot of a full-information puzzle, which may be solved, unsolved,
    or even unsolvable.
    """

    def fail_fast(self):
        """
        Return True if Puzzle self can never be extended to a solution.

        Override this in a subclass where you can determine early that
        this Puzzle cann't be solved.

        @type self: Puzzle
        @rtype: bool
        """
        return False

    def is_solved(self):
        """
        Return True iff Puzzle self is solved.

        This is an abstract method that must be implemented
        in a subclass.

        @type self: Puzzle
        @rtype: bool
        """
        raise NotImplementedError

    def extensions(self):
        """
        Return list of legal extensions of Puzzle self.

        This is an abstract method that must be implemented
        in a subclass.

        @type self: Puzzle
        @rtype: generator[Puzzle]
        """
        raise NotImplementedError
```

```python
from collections import deque
from puzzle import Puzzle
# set higher recursion limit
# which is needed in PuzzleNode.__str__
# you may uncomment the next lines on a unix system such as CDF
# import resource
# resource.setrlimit(resource.RLIMIT_STACK, (2**29, -1))
import sys

sys.setrecursionlimit(10 ** 6)
visited = set()


def depth_first_solve(puzzle):
    """
    Return a path from PuzzleNode(puzzle) to a PuzzleNode containing
    a solution, with each child containing an extension of the puzzle
    in its parent.  Return None if this is not possible.

    @type puzzle: Puzzle
    @rtype: PuzzleNode
    """
    solution = PuzzleNode(puzzle)
    if solution.puzzle.is_solved():
        return solution
    elif solution.puzzle.fail_fast():
        return None
    else:
        # else, check it's extensions
        for item in solution.puzzle.extensions():
            if item.__str__() not in visited:
                visited.add(item.__str__())
                # call recursion on the item
                result = depth_first_solve(item)
                # if we can find the solution
                if result is not None:
                    return PuzzleNode(puzzle, [result])
    return None

# reference https://www.youtube.com/watch?v=zLZhSSXAwxI
# reference https://en.wikipedia.org/wiki/Depth-first_search


def breadth_first_solve(puzzle):
    """
    Return a path from PuzzleNode(puzzle) to a PuzzleNode containing
    a solution, with each child PuzzleNode containing an extension
    of the puzzle in its parent.  Return None if this is not possible.

    @type puzzle: Puzzle
    @rtype: PuzzleNode
    """
    seen = set()
    store = deque()
```

```
    store.append(PuzzleNode(puzzle))
    while store:
        r = store.popleft()
        if not r.puzzle.__str__() in seen:
            seen.add(r.puzzle.__str__())
            # found the solution node
            if r.puzzle.is_solved():
                myself = r
                # build a path back to the root
                while myself.parent is not None:
                    myself.parent.children = [myself]
                    myself = myself.parent
                return myself
            elif not r.puzzle.fail_fast():
                for i in r.puzzle.extensions():
                    # indicate it's parent since we want to find path
back
                    # after we find the solution node.
                    store.append(PuzzleNode(i, parent=r))

# reference: https://en.wikipedia.org/wiki/Breadth-first_search


# Class PuzzleNode helps build trees of PuzzleNodes that have
# an arbitrary number of children, and a parent.
class PuzzleNode:
    """
    A Puzzle configuration that refers to other configurations that it
    can be extended to.
    """

    def __init__(self, puzzle=None, children=None, parent=None):
        """
        Create a new puzzle node self with configuration puzzle.

        @type self: PuzzleNode
        @type puzzle: Puzzle | None
        @type children: list[PuzzleNode]
        @type parent: PuzzleNode | None
        @rtype: None
        """
        self.puzzle, self.parent = puzzle, parent
        if children is None:
            self.children = []
        else:
            self.children = children[:]

    def __eq__(self, other):
        """
        Return whether PuzzleNode self is equivalent to other

        @type self: PuzzleNode
        @type other: PuzzleNode | Any
```

```
        @rtype: bool

        >>> from word_ladder_puzzle import WordLadderPuzzle
        >>> pn1 = PuzzleNode(WordLadderPuzzle("on", "no", {"on", "no",
"oo"}))
        >>> pn2 = PuzzleNode(WordLadderPuzzle("on", "no", {"on", "oo",
"no"}))
        >>> pn3 = PuzzleNode(WordLadderPuzzle("no", "on", {"on", "no",
"oo"}))
        >>> pn1.__eq__(pn2)
        True
        >>> pn1.__eq__(pn3)
        False
        """
        return (type(self) == type(other) and
                self.puzzle == other.puzzle and
                all([x in self.children for x in other.children]) and
                all([x in other.children for x in self.children]))

    def __str__(self):
        """
        Return a human-readable string representing PuzzleNode self.

        # doctest not feasible.
        """
        return "{}\n\n{}".format(self.puzzle,
                                 "\n".join([str(x) for x in
self.children]))
```

```python
from puzzle import Puzzle


class SudokuPuzzle(Puzzle):
    """
    A sudoku puzzle that may be solved, unsolved, or even unsolvable.
    """

    def __init__(self, n, symbols, symbol_set):
        """
        Create a new nxn SudokuPuzzle self with symbols
        from symbol_set already selected.

        @type self: SudokuPuzzle
        @type n: int
        @type symbols: list[str]
        @type symbol_set: set[str]
        """
        assert n > 0
        assert round(n ** (1 / 2)) * round(n ** (1 / 2)) == n
        assert all([d in (symbol_set | {"*"}) for d in symbols])
        assert len(symbol_set) == n
        assert len(symbols) == n ** 2
        self._n, self._symbols, self._symbol_set = n, symbols, symbol_set

    def __eq__(self, other):
        """
        Return whether SudokuPuzzle self is equivalent to other.

        @type self: SudokuPuzzle
        @type other: SudokuPuzzle | Any
        @rtype: bool

        >>> grid1 = ["A", "B", "C", "D"]
        >>> grid1 += ["D", "C", "B", "A"]
        >>> grid1 += ["*", "D", "*", "*"]
        >>> grid1 += ["*", "*", "*", "*"]
        >>> s1 = SudokuPuzzle(4, grid1, {"A", "B", "C", "D"})
        >>> grid2 = ["A", "B", "C", "D"]
        >>> grid2 += ["D", "C", "B", "A"]
        >>> grid2 += ["*", "D", "*", "*"]
        >>> grid2 += ["*", "*", "*", "*"]
        >>> s2 = SudokuPuzzle(4, grid2, {"A", "B", "C", "D"})
        >>> s1.__eq__(s2)
        True
        >>> grid3 = ["A", "B", "C", "D"]
        >>> grid3 += ["D", "C", "B", "A"]
        >>> grid3 += ["*", "D", "*", "*"]
        >>> grid3 += ["*", "A", "*", "*"]
        >>> s3 = SudokuPuzzle(4, grid3, {"A", "B", "C", "D"})
        >>> s1.__eq__(s3)
        False
        """
        return (type(other) == type(self) and
```

```
                self._n == other._n and self._symbols == other._symbols
and
                self._symbol_set == other._symbol_set)

    def __str__(self):
        """
        Return a human-readable string representation of SudokuPuzzle
self.

        >>> grid = ["A", "B", "C", "D"]
        >>> grid += ["D", "C", "B", "A"]
        >>> grid += ["*", "D", "*", "*"]
        >>> grid += ["*", "*", "*", "*"]
        >>> s = SudokuPuzzle(4, grid, {"A", "B", "C", "D"})
        >>> print(s)
        AB|CD
        DC|BA
        -----
        *D|**
        **|**
        """

        def row_pickets(row):
            """
            Return string of characters in row with | divider
            between groups of sqrt(n)

            @type row: list[str]
            @rtype: str
            """
            string_list = []
            r = round(self._n ** (1 / 2))
            for i in range(self._n):
                if i > 0 and i % r == 0:
                    string_list.append("|")
                string_list.append(row[i])
            return "".join(string_list)

        def table_dividers(table):
            """
            Return rows of strings in table with
            "-----" dividers between groups of sqrt(n) rows.

            @type table: list[str]
            @rtype: list[str]
            """
            r = round(self._n ** (1 / 2))
            t, divider = [], "-" * (self._n + r - 1)
            for i in range(self._n):
                if i > 0 and i % r == 0:
                    t.append(divider)
                t.append(table[i])
            return t
```

```
        rows = [row_pickets([self._symbols[r * self._n + c]
                             for c in range(self._n)])
                for r in range(self._n)]
        rows = table_dividers(rows)
        return "\n".join(rows)

    def is_solved(self):
        """
        Return whether Puzzle self is solved.

        @type self: Puzzle
        @rtype: bool

        >>> grid = ["A", "B", "C", "D"]
        >>> grid += ["C", "D", "A", "B"]
        >>> grid += ["B", "A", "D", "C"]
        >>> grid += ["D", "C", "B", "A"]
        >>> s = SudokuPuzzle(4, grid, {"A", "B", "C", "D"})
        >>> s.is_solved()
        True
        >>> grid[9] = "D"
        >>> grid[10] = "A"
        >>> s = SudokuPuzzle(4, grid, {"A", "B", "C", "D"})
        >>> s.is_solved()
        False
        """
        # convenient names
        n, symbols = self._n, self._symbols
        # no "*" left and all rows, column, subsquares have correct
symbols
        return ("*" not in symbols and
                all([(self._row_set(i) == self._symbol_set and
                      self._column_set(i) == self._symbol_set and
                      self._subsquare_set(i) ==
                      self._symbol_set) for i in range(n ** 2)]))

    def extensions(self):
        """
        Return list of extensions of SudokuPuzzle self.

        @type self: Puzzle
        @rtype: list[Puzzle]

        >>> grid = ["A", "B", "C", "D"]
        >>> grid += ["C", "D", "A", "B"]
        >>> grid += ["B", "A", "D", "C"]
        >>> grid += ["D", "C", "B", "*"]
        >>> s = SudokuPuzzle(4, grid, {"A", "B", "C", "D"})
        >>> L1 = list(s.extensions())
        >>> grid[-1] = "A"
        >>> L2 = [SudokuPuzzle(4, grid, {"A", "B", "C", "D"})]
        >>> len(L1) == len(L2)
        True
        >>> all([s in L2 for s in L1])
```

```
        True
        >>> all([s in L1 for s in L2])
        True
        """
        # convenient names
        symbols, symbol_set, n = self._symbols, self._symbol_set, self._n
        if "*" not in symbols:
            # return an empty generator
            return [_ for _ in []]
        else:
            # position of first empty position
            i = symbols.index("*")
            # allowed symbols at position i
            # A | B == A.union(B)
            allowed_symbols = (self._symbol_set -
                               (self._row_set(i) |
                                self._column_set(i) |
                                self._subsquare_set(i)))
            # list of SudokuPuzzles with each legal digit at position i
            return (
                [SudokuPuzzle(n,
                 symbols[:i] + [d] + symbols[i + 1:], symbol_set)
                 for d in allowed_symbols])

    def fail_fast(self):
        """
        return True if Puzzle self can never be extended to a solution,
hence
        abandoning, and false otherwise.

        @type self: Puzzle
        @rtype: bool

        >>> grid = ["A", "B", "C", "D"]
        >>> grid += ["C", "D", "A", "B"]
        >>> grid += ["B", "A", "D", "C"]
        >>> grid += ["D", "C", "B", "*"]
        >>> s = SudokuPuzzle(4, grid, {"A", "B", "C", "D"})
        >>> s.fail_fast()
        False
        >>> grid = ["A", "B", "C", "D"]
        >>> grid += ["C", "*", "*", "B"]
        >>> grid += ["B", "*", "D", "A"]
        >>> grid += ["D", "C", "B", "*"]
        >>> s = SudokuPuzzle(4, grid, {"A", "B", "C", "D"})
        >>> s.fail_fast()
        True
        >>> grid = ["*", "*", "*", "*"]
        >>> grid += ["*", "*", "*", "*"]
        >>> grid += ["*", "*", "*", "*"]
        >>> grid += ["*", "*", "*", "*"]
        >>> s = SudokuPuzzle(4, grid, {"A", "B", "C", "D"})
        >>> s.fail_fast()
        False
```

```
        >>> grid = ["A", "B", "C", "D"]
        >>> grid += ["*", "*", "*", "B"]
        >>> grid += ["C", "*", "D", "A"]
        >>> grid += ["D", "*", "*", "*"]
        >>> s = SudokuPuzzle(4, grid, {"A", "B", "C", "D"})
        >>> s.fail_fast()
        True
        """

        list_index = []
        for i in range(self._n ** 2):
            if self._symbols[i] == "*":
                list_index.append(i)
        for ind in list_index:
            available_sym = self._symbol_set - (self._row_set(ind) |
                                                self._column_set(ind) |
                                                self._row_set(ind))
            if available_sym == set():
                return True
        return False

    # some helper methods
    def _row_set(self, m):
        #
        # Return set of symbols in row of SudokuPuzzle self's symbols
        # where position m occurs.
        #
        # @type self: SudokuPuzzle
        # @type m: int
        assert 0 <= m < self._n ** 2
        # convenient names
        n, symbols = self._n, self._symbols
        # first position in m's row
        r = (m // n) * n
        # set of elements from symbols[r] .. symbols[r+n-1]
        return set([symbols[r + i] for i in range(n)])

    def _column_set(self, m):
        # Return set of symbols in column of SudokuPuzzle self's symbols
        # where position m occurs.
        #
        # @type self: SudokuPuzzle
        # @type m: int
        assert 0 <= m <= self._n ** 2
        # convenient names
        symbols, n = self._symbols, self._n
        # first position in m's column
        c = m % n
        # set of elements from symbols[c], symbols[c + n],
        # ... symbols[c + (n * (n-1))]
        return set([symbols[c + (i * n)] for i in range(n)])

    def _subsquare_set(self, m):
```

```
        # Return set of symbols in subsquare of SudokuPuzzle self's
symbols
        # where position m occurs.
        #
        # @type self: Sudoku Puzzle
        # @type m: int
        assert 0 <= m < self._n ** 2
        # convenient names
        n, symbols = self._n, self._symbols
        # row, column where m occur
        row, col = m // n, m % n
        # length of subsquares
        ss = round(n ** (1 / 2))
        # upper-left position of m's subsquare
        ul = (((row // ss) * ss) * n) + ((col // ss) * ss)
        # return set of symbols from subsquare starting at ul
        return set(
            [symbols[ul + i + n * j] for i in range(ss) for j in
range(ss)])


if __name__ == "__main__":
    import doctest

    doctest.testmod()
    s = SudokuPuzzle(9,
                     ["*", "*", "*", "7", "*", "8", "*", "1", "*",
                      "*", "*", "7", "*", "9", "*", "*", "*", "6",
                      "9", "*", "3", "1", "*", "*", "*", "*", "*",
                      "3", "5", "*", "8", "*", "*", "6", "*", "1",
                      "*", "*", "*", "*", "*", "*", "*", "*", "*",
                      "1", "*", "6", "*", "*", "9", "*", "4", "8",
                      "*", "*", "*", "*", "*", "1", "2", "*", "7",
                      "8", "*", "*", "*", "7", "*", "4", "*", "*",
                      "*", "6", "*", "3", "*", "2", "*", "*", "*"],
                     {"1", "2", "3", "4", "5", "6", "7", "8", "9"})

    from time import time

    print("solving sudoku from July 9 2015 Star... \n\n{}\n\n".format(s))
    from puzzle_tools import depth_first_solve

    start = time()
    sol = depth_first_solve(s)
    print(sol)
    while sol.children:
        sol = sol.children[0]
    end = time()
    print("time to solve 9x9 using depth_first: "
          "{} seconds\n".format(end - start))
    print(sol)

    s = SudokuPuzzle(9,
                     ["*", "*", "*", "9", "*", "2", "*", "*", "*",
```

```
                  "*", "9", "1", "*", "*", "*", "6", "3", "*",
                  "*", "3", "*", "*", "7", "*", "*", "8", "*",
                  "3", "*", "*", "*", "*", "*", "*", "*", "8",
                  "*", "*", "9", "*", "*", "*", "2", "*", "*",
                  "5", "*", "*", "*", "*", "*", "*", "*", "7",
                  "*", "7", "*", "*", "8", "*", "*", "4", "*",
                  "*", "4", "5", "*", "*", "*", "8", "1", "*",
                  "*", "*", "*", "3", "*", "6", "*", "*", "*"],
                 {"1", "2", "3", "4", "5", "6", "7", "8", "9"})

print("solving 3-star sudoku from \"That's Puzzling\","
      "November 14th 2015\n\n{}\n\n".format(s))
start = time()
sol = depth_first_solve(s)
while sol.children:
    sol = sol.children[0]
end = time()
print("time to solve 9x9 using depth_first: {} seconds\n".format(
    end - start))
print(sol)

s = SudokuPuzzle(9,
                 ["5", "6", "*", "*", "*", "7", "*", "*", "9",
                  "*", "7", "*", "*", "4", "8", "*", "3", "1",
                  "*", "*", "*", "*", "*", "*", "*", "*", "*",
                  "4", "3", "*", "*", "*", "*", "*", "*", "*",
                  "*", "8", "*", "*", "*", "*", "*", "9", "*",
                  "*", "*", "*", "*", "*", "*", "*", "2", "6",
                  "*", "*", "*", "*", "*", "*", "*", "*", "*",
                  "1", "9", "*", "3", "6", "*", "*", "7", "*",
                  "7", "*", "*", "1", "*", "*", "*", "4", "2"],
                 {"1", "2", "3", "4", "5", "6", "7", "8", "9"})

print(
    "solving 4-star sudoku from \"That's Puzzling\", "
    "November 14th 2015\n\n{}\n\n".format(
        s))
start = time()
sol = depth_first_solve(s)
while sol.children:
    sol = sol.children[0]
end = time()
print("time to solve 9x9 using depth_first: {} seconds\n".format(
    end - start))
print(sol)
```

```
from puzzle import Puzzle


class WordLadderPuzzle(Puzzle):
    """
    A word-ladder puzzle that may be solved, unsolved, or even
unsolvable.
    """

    def __init__(self, from_word, to_word, ws):
        """
        Create a new word-ladder puzzle with the aim of stepping
        from from_word to to_word using words in ws, changing one
        character at each step.

        @type from_word: str
        @type to_word: str
        @type ws: set[str]
        @rtype: None
        """
        (self._from_word, self._to_word, self._word_set) = (from_word,
                                                             to_word, ws)
        # set of characters to use for 1-character changes
        self._chars = "abcdefghijklmnopqrstuvwxyz"

    def __eq__(self, other):
        """
        Return whether WordLadderPuzzle self is equivalent to other.

        @type self: WordLadderPuzzle
        @type other: WordLadderPuzzle | Any
        @rtype: bool

        >>> wlp1 = WordLadderPuzzle ("cost", "save", \
        {"cost", "cast", "cave","case","save"})
        >>> wlp2 = WordLadderPuzzle ("cost", "save", \
        {"cost", "cast", "cave", "save", "case"})
        >>> wlp1.__eq__(wlp2)
        True
        >>> wlp3 = WordLadderPuzzle ("cast", "save", \
        {"cost", "cast", "cave", "save", "case"})
        >>> wlp1.__eq__(wlp3)
        False
        """
        return (type(self) == type(other) and
                self._from_word == other._from_word and
                self._to_word == other._to_word and
                self._word_set == other._word_set)

    def __str__(self):
        """
        Return a human-readable string representation of
WordLadderPuzzles self
```

```
        @type self: WordLadderPuzzles
        @rtype: str

        >>> wlp1 = WordLadderPuzzle("cost", "save", \
        {"cost", "cast", "cave","case","save"})
        >>> print (wlp1)
        cost -> save
        """
        return "{0} -> {1}".format(self._from_word, self._to_word)

    def extensions(self):
        """
        Return list of extensions of WordLadderPuzzle self

        @type self: WordLadderPuzzle
        @rtype: list[WordLadderPuzzle]

        >>> wps1 = WordLadderPuzzle("cost", "save", \
        {"cost", "cast", "cave","case","save","cosy"})
        >>> L1 = list(wps1.extensions())
        >>> L2 = [WordLadderPuzzle("cast", "save", \
        {"cost", "cast", "cave","case","save", "cosy"}),\
        WordLadderPuzzle("cosy", "save", \
        {"cost", "cast", "cave","case","save", "cosy"})]
        >>> len(L1) == len(L2)
        True
        >>> all([s in L2 for s in L1])
        True
        >>> all([s in L1 for s in L2])
        True
        """
        # list of all the possible words by changing each char to other
25 chars
        list_maybe = []
        # list of possible words that are in word_set
        list_allow = []
        for i in range(len(self._from_word)):
            list_word = list(self._from_word)
            # change the char to other 25 chars, so replace the char with
""
            for char in self._chars.replace(list_word[i], ""):
                list_word[i] = char
                str_word = "".join(list_word)
                list_maybe.append(str_word)
        for word in list_maybe:
            if word in self._word_set:
                list_allow.append(word)
            else:
                pass
        return ([WordLadderPuzzle(i, self._to_word, self._word_set) for i
in
                  list_allow])

    def is_solved(self):
```

```
        """
        Return whether Puzzle self is solved.

        @type self: WordLadderPuzzle
        @rtype: bool

        >>> wps1 = WordLadderPuzzle("cast", "save", \
        {"cost", "cast", "cave","case","save"})
        >>> wps1.is_solved()
        False
        >>> wps2 = WordLadderPuzzle("save", "save", \
        {"cost", "cast", "cave","case","save"})
        >>> wps2.is_solved()
        True
        """
        return self._from_word == self._to_word


if __name__ == '__main__':
    import doctest

    doctest.testmod()
    from puzzle_tools import breadth_first_solve, depth_first_solve
    from time import time

    with open("words.txt", "r") as words:
        word_set = set(words.read().split())
    w = WordLadderPuzzle("same", "cost", word_set)
    start = time()
    sol = breadth_first_solve(w)
    end = time()
    print("Solving word ladder from same->cost")
    print("...using breadth-first-search")
    print("Solutions: {} took {} seconds.".format(sol, end - start))
    start = time()
    sol = depth_first_solve(w)
    end = time()
    print("Solving word ladder from same->cost")
    print("...using depth-first-search")
    print("Solutions: {} took {} seconds.".format(sol, end - start))
```