# Copyright 2019 D-Wave Systems Inc.

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

from \_\_future\_\_ import print\_function

import dwavebinarycsp

import re

def get\_maze\_bqm(n\_rows, n\_cols, start, end, walls, penalty\_per\_tile=0.5):

    """Returns a BQM that corresponds to a valid path through a maze. This maze is described by the parameters.

    Specifically, it uses the parameters to build a maze constraint satisfaction problem (CSP). This maze CSP is then

    converted into the returned BQM.

    Note: If penalty\_per\_tile is too large, the path will be too heavily penalized and the optimal solution might

    produce no path at all.

    Args:

        n\_rows: Integer. The number of rows in the maze.

        n\_cols: Integer. The number of cols in the maze.

        start: String. The location of the starting point of the maze. String follows the format of get\_label(..).

        end: String. The location of the end point of the maze. String follows the format of get\_label(..).

        walls: List of Strings. The list of inner wall locations. Locations follow the format of get\_label(..).

        penalty\_per\_tile: A number. Penalty for each tile that is included in the path; encourages shorter paths.

    Returns:

        A dimod.BinaryQuadraticModel

    """

    maze = Maze(n\_rows, n\_cols, start, end, walls)

    return maze.get\_bqm(penalty\_per\_tile)

def get\_label(row, col, direction):

    """Provides a string that follows a standard format for naming constraint variables in Maze.

    Namely, "<row\_index>,<column\_index><north\_or\_west\_direction>".

    Args:

        row: Integer. Index of the row.

        col: Integer. Index of the column.

        direction: String in the set {'n', 'w'}. 'n' indicates north and 'w' indicates west.

    """

    return "{row},{col}{direction}".format(\*\*locals())

def assert\_label\_format\_valid(label):

    """Checks that label conforms with the standard format for naming constraint variables in Maze.

    Namely, "<row\_index>,<column\_index><north\_or\_west\_direction>".

    Args:

        label: String.

    """

    is\_valid = bool(re.match(r'^(\d+),(\d+)[nw]$', label))

    assert is\_valid, ("{label} is in the incorrect format. Format is <row\_index>,<column\_index><north\_or\_west>. "

                      "Example: '4,3w'").format(\*\*locals())

def sum\_to\_two\_or\_zero(\*args):

    """Checks to see if the args sum to either 0 or 2.

    """

    sum\_value = sum(args)

    return sum\_value in [0, 2]

class Maze:

    """An object that stores all the attributes necessary to represent a maze as a constraint satisfaction problem.

    Args:

        n\_rows: Integer. The number of rows in the maze.

        n\_cols: Integer. The number of cols in the maze.

        start: String. The location of the starting point of the maze. String follows the format of get\_label(..).

        end: String. The location of the end point of the maze. String follows the format of get\_label(..).

        walls: List of Strings. The list of inner wall locations. Locations follow the format of get\_label(..).

    """

    def \_\_init\_\_(self, n\_rows, n\_cols, start, end, walls):

        assert isinstance(n\_rows, int) and n\_rows > 0, "'n\_rows' is not a positive integer".format(n\_rows)

        assert isinstance(n\_cols, int) and n\_cols > 0, "'n\_cols' is not a positive integer".format(n\_cols)

        assert start != end, "'start' cannot be the same as 'end'"

        # Check label format

        assert\_label\_format\_valid(start)

        assert\_label\_format\_valid(end)

        for wall in walls:

            assert\_label\_format\_valid(wall)

        # Instantiate

        self.n\_rows = n\_rows

        self.n\_cols = n\_cols

        self.start = start

        self.end = end

        self.walls = walls

        self.csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)

    def \_apply\_valid\_move\_constraint(self):

        """Applies a sum to either 0 or 2 constraint on each tile of the maze.

        Note: This constraint ensures that a tile is either not entered at all (0), or is entered and exited (2).

        """

        # Grab the four directions of each maze tile and apply two-or-zero constraint

        for i in range(self.n\_rows):

            for j in range(self.n\_cols):

                directions = {get\_label(i, j, 'n'), get\_label(i, j, 'w'), get\_label(i+1, j, 'n'),

                              get\_label(i, j+1, 'w')}

                self.csp.add\_constraint(sum\_to\_two\_or\_zero, directions)

    def \_set\_start\_and\_end(self):

        """Sets the values of the start and end locations of the maze.

        """

        self.csp.fix\_variable(self.start, 1)  # start location

        self.csp.fix\_variable(self.end, 1)  # end location

    def \_set\_borders(self):

        """Sets the values of the outer border of the maze; prevents a path from forming over the border.

        """

        for j in range(self.n\_cols):

            top\_border = get\_label(0, j, 'n')

            bottom\_border = get\_label(self.n\_rows, j, 'n')

            try:

                self.csp.fix\_variable(top\_border, 0)

            except ValueError:

                if not top\_border in [self.start, self.end]:

                    raise ValueError

            try:

                self.csp.fix\_variable(bottom\_border, 0)

            except ValueError:

                if not bottom\_border in [self.start, self.end]:

                    raise ValueError

        for i in range(self.n\_rows):

            left\_border = get\_label(i, 0, 'w')

            right\_border = get\_label(i, self.n\_cols, 'w')

            try:

                self.csp.fix\_variable(left\_border, 0)

            except ValueError:

                if not left\_border in [self.start, self.end]:

                    raise ValueError

            try:

                self.csp.fix\_variable(right\_border, 0)

            except ValueError:

                if not right\_border in [self.start, self.end]:

                    raise ValueError

    def \_set\_inner\_walls(self):

        """Sets the values of the inner walls of the maze; prevents a path from forming over an inner wall.

        """

        for wall in self.walls:

            self.csp.fix\_variable(wall, 0)

    def get\_bqm(self, penalty\_per\_tile=0.5):

        """Applies the constraints necessary to form a maze and returns a BQM that would correspond to a valid path

        through said maze.

        Note: If penalty\_per\_tile is too large, the path will be too heavily penalized and the optimal solution might

          no path at all.

        Args:

            penalty\_per\_tile: A number. Penalty for each tile that is included in the path; encourages shorter paths.

        Returns:

            A dimod.BinaryQuadraticModel

        """

        # Apply constraints onto self.csp

        self.\_apply\_valid\_move\_constraint()

        self.\_set\_start\_and\_end()

        self.\_set\_borders()

        self.\_set\_inner\_walls()

        # Grab bqm constrained for valid solutions

        bqm = dwavebinarycsp.stitch(self.csp)

        # Edit bqm to favour optimal solutions

        for v in bqm.variables:

            # Ignore auxiliary variables

            if isinstance(v, str) and re.match(r'^aux\d+$', v):

                continue

            # Add a penalty to every tile of the path

            bqm.add\_variable(v, penalty\_per\_tile)

        return bqm

    def visualize(self, solution=None):

        def get\_visual\_coords(coords):

            coord\_pattern = "^(\d+),(\d+)([nw])$"

            row, col, dir = re.findall(coord\_pattern, coords)[0]

            new\_row, new\_col = map(lambda x: int(x) \* 2 + 1, [row, col])

            new\_row, new\_col = (new\_row-1, new\_col) if dir == "n" else (new\_row, new\_col-1)

            return new\_row, new\_col, dir

        # Constants for maze symbols

        WALL = "#"      # maze wall

        NS = "|"        # path going in north-south direction

        EW = "\_"        # path going in east-west direction

        POS = "."       # coordinate position

        EMPTY = " "     # whitespace; indicates no path drawn

        # Check parameters

        if solution is None:

            solution = []

        # Construct empty maze visual

        # Note: the maze visual is (2 \* original-maze-dimension + 1) because

        #   each position has an associated north-edge and an associated

        #   west-edge. This requires two rows and two columns to draw,

        #   respectively. Thus, the "2 \* original-maze-dimension" is needed.

        #      |      <-- north edge

        #     \_.      <-- west edge and position

        #   To get a south-edge or an east-edge, the north-edge from the row

        #   below or the west-edge from the column on the right can be used

        #   respectively. This trick, however, cannot be used for the last row

        #   nor for the rightmost column, hence the "+ 1" in the equation.

        width = 2\*self.n\_cols + 1       # maze visual's width

        height = 2\*self.n\_rows + 1      # maze visual's height

        empty\_row = [EMPTY] \* (width-2)

        empty\_row = [WALL] + empty\_row + [WALL]   # add left and right borders

        visual = [list(empty\_row) for \_ in range(height)]

        visual[0] = [WALL] \* width      # top border

        visual[-1] = [WALL] \* width     # bottom border

        # Add coordinate positions in maze visual

        # Note: the symbol POS appears at every other position because there

        #   could potentially be a path segment sitting between the two

        #   positions.

        for position\_row in visual[1::2]:

            position\_row[1::2] = [POS] \* self.n\_cols

        # Add maze start and end to visual

        start\_row, start\_col, start\_dir = get\_visual\_coords(self.start)

        end\_row, end\_col, end\_dir = get\_visual\_coords(self.end)

        visual[start\_row][start\_col] = NS if start\_dir=="n" else EW

        visual[end\_row][end\_col] = NS if end\_dir=="n" else EW

        # Add interior walls to visual

        for w in self.walls:

            row, col, \_ = get\_visual\_coords(w)

            visual[row][col] = WALL

        # Add solution path to visual

        for s in solution:

            row, col, dir = get\_visual\_coords(s)

            visual[row][col] = NS if dir=="n" else EW

        # Print solution

        for s in visual:

            print("".join(s))