futoshiki

Project 2 for CS-UY 4613 - Artificial Intelligence - https://github.com/Chloe-323/futoshiki

Overview

Futoshiki (Japanese for Inequality Puzzle) is a puzzle game similar to Sudoku. The game is played on a 5x5 grid, with some of the squares already filled in. The player must fill in the remaining squares with the numbers 1 through 5 such that no two numbers in the same row or column are the same, and no inequality constraint is violated. Inequalities are indicated by a greater-than or less-than symbol between two squares. For example, in the puzzle below, the numbers 5, 2, and 1 are already filled in. The inequality constraints are indicated by the greater-than and less-than symbols. The player must fill in the remaining squares with the numbers 1 through 5 such that no two numbers in the same row or column are the same, and no inequality constraint is violated. In this case, the only valid solution is the one shown below.

```
_   _ > _   _   _    |     2   3 > 1   4   5
^   ^                |     ^   ^
_   _   _   _   _    |     3   5   2   1   4
                     |
5   _   _ > _   _    |     5   1   4 > 3   2
                     |
_   _   _   2   _    |     1   4   5   2   3
    v                |     v
_   _   _ < _   1    |     4   2   3 < 5   1
```

Running the Program

Installing Python

The program is written in Python 3. To run the program, you must have Python 3 installed. If you do not have Python 3 installed, you can download it from the Python website.

Installing Dependencies

You must install the following dependencies to run the program:

- pynput
- rich

Running the Program

To run the program, simply run the following command:

```
python3 futoshiki.py -i <input_file> [-o <output_file>] [-s]
```

Note: Certain versions of Windows may not support command line arguments. If none are given, the program will revert to an interactive prompt to get the information.

Input File Format

The input file must be a text file with the following format:

```
0 0 0 0 0
0 0 0 0 0
5 0 0 0 0
0 0 0 2 0
0 0 0 0 1

0 > 0 0
0 0 0 0
0 0 > 0
0 0 0 0
0 0 < 0

^ ^ 0 0 0
0 0 0 0 0
0 0 0 0 0
0 v 0 0 0
```

The first five lines define the initial numbers on the board. A zero represents an empty space, and any nonzero number represents a number in the initial state.

The next five lines define the horizontal inequality constraints. A zero represents no constraint, and `<` or `>` represent a constraint between two tiles.

The final four lines define the vertical inequality constraints. A zero represents no constraint, and `^` and `v` represent a constraint.

Output File Format

If an output file is specified, the program will write the solved board to the output file. Note that it will not show the constraints in the output file. The output file will have the following format:

```
2 3 1 4 5
3 5 2 1 4
5 1 4 3 2
1 4 5 2 3
4 2 3 5 1
```

Code Overview

The program defines a single class with five methods:

`__init__(self, input_file, output_file, solve)`

The `__init__` method reads the input file and initializes the board, row_constraints, and col_constraints variables based on the contents of the file. It also initializes the constraints variable based on the horizontal and vertical inequality constraints in the input file.

It will initialize the following instance variables:

- state: a namedtuple representing the current state of the board and containing the following variables:
  - board: A list of lists representing the numbers in the grid. The value at board[i][j] is the number in the cell at row i and column j. If the cell is blank, the value is 0.
  - constraints: A dictionary representing the inequality constraints in the puzzle. The keys are tuples of the form (row, column) and the values are dictionaries representing the constraints for that cell. For example, `constraints[(i, j)][(i, j+1)]` represents the constraint between the cell at `(i, j)` and the cell at `(i, j+1)`. A constraint can be either `>` or `<`. If there is no constraint between the two cells, then the pair is not in the dictionary.
  - row_constraints: A list of sets representing the numbers that have already been used in each row.
  - col_constraints: A list of sets representing the numbers that have already been used in each column.
- output_file: The name of the output file. If it is None, then the program will not write to an output file.

It then reads from the input file and initializes the state variable based on the contents of the file. The file format is defined above.

`__str__(self)`

The `__str__` method returns a string representation of the current state of the board. It will print the board with the inequality constraints in between the cells.

In order to display an aesthetically pleasing board, there is more space in between the tiles, such that all of the inequality constraints can fit betweem them without causing any offset or excessively warping the shape of the board.

There are four characters between each horizontal tile and 48 characters between each vertical tile. That is, given an index in the string, adding 4 to that index will give the index of the next tile in the same row, and adding 48 will give the index of the next tile in the same column.

`print(self, selected=None, err=False)`

The `print` method prints the current state of the board to the console. It will print the board with the inequality constraints in between the cells. It will also highlight the selected cell in red, if it is given. If `err` is true, then the selected cell will have an X in it. This is used in the `play` method to show the user that they have made an invalid move.

In order to print an aesthetically pleasing board, the rich library is used.

`play(self)`

The `play` method allows the user to play the game. It will print the board to the console and wait for the user to take an action. The user can take the following actions:

- `↑` - Move the cursor up
- `→` - Move the cursor right
- `↓` - Move the cursor down
- `←` - Move the cursor left
- `1`-`5` - Place the number in the selected cell
- `s` - Solve the puzzle

Once the user has selected an action, the program will update the board and print the new board to the console. If the user has selected a number, then the program will check if the move is valid. If it is not valid, then the program will print the board with an X in the selected cell. If it is, then that number will be added to the board at the location of the cursor. If the user has selected the solve action, then the program will solve the puzzle and print the solved board to the console.

This method defines an `immovable` set which will contain all the cells that cannot be changed. If a user puts a number down and then changes their mind, they can move the cursor to the cell and choose a different number to replace it. If they try to change a cell that was defined at the start of the puzzle, then they will receive an error. This is to prevent the user from changing the initial state of the puzzle.

`solve(self, state=None)`

The `solve` method solves the puzzle. It will return a state which represents the solved puzzle. If the puzzle is unsolvable, then it will return `None`. If the state variable is not None, then the method will solve the puzzle starting from that state. Otherwise, it will solve the puzzle starting from the current state.

The `solve` method uses a backtracking search algorithm to find a solution to the puzzle. The algorithm works by considering the unassigned tiles with the most constraints first, and trying the possible values for those tiles in the order of the number of constraints they will create. If a tile has no legal values, the algorithm backtracks to the previous tile and tries the next legal value for that tile. If there are no more legal values to try, the algorithm continues backtracking until it finds a tile with untried legal values. If the algorithm reaches a point where there are no more tiles to assign, it has found a solution to the puzzle.

The solve method contains three nested functions: `get_legal_moves`, `select_next_assignment`, and `update_constraints`:

- The `get_legal_moves` function takes a state and the indices row and col of a tile and returns a list of legal values for that tile. A value is legal if it has not already been used in the same row or column as the tile, and if it satisfies all the inequality constraints involving the tile.
- The `select_next_assignment` function takes a state and returns the indices of the next tile to assign a value to. It first finds the tiles with the most constraints, and if there is more than one such tile, it picks the one with the highest degree (defined as the number of unassigned tiles with constraints involving the tile). If there is still a tie, then it will pick one arbitrarily, depending on the order that Python stores items in the set.
- The `update_constraints` function takes a state, the indices row and col of a tile, and the value assigned to that tile. It updates the `row_constraints` and `col_constraints` variables to reflect the

new value, and it also updates the number of constraints for each tile.

The solve method starts by initializing the state variable to the provided state argument or the state instance variable of the Puzzle object. It then enters a loop where it selects the next tile to assign a value to using the `select_next_assignment` function, gets the legal values for that tile using the `get_legal_moves` function, and tries each legal value in turn. For each value it tries, it creates a copy of the current state, assigns the value to the tile, and updates the constraints using the `update_constraints` function. It then calls itself recursively with the new state to continue the search. If the recursive call returns a solution, the solve method writes the solution to the output file and returns it. If the recursive call returns `None`, indicating that the value was not a part of a valid solution, the solve method continues to the next legal value for the tile. If it has tried all the legal values for the tile and none of them led to a valid solution, it returns `None` to indicate that the current state is not part of a valid solution and the search should backtrack. This process continues until the puzzle has either been solved, or has been found to be unsolvable.

Assignment Files

Input1.txt

```
3 0 0 0 0
0 0 0 2 0
0 0 0 0 3
0 0 0 0 0
0 0 0 0 0

0 0 0 0
0 0 < 0
0 0 < 0
0 0 0 0
0 < 0 0

0 0 0 0 v
0 0 0 0 0
0 0 0 0 0
0 0 v 0 0
```

Output1.txt

```
3 2 4 1 5
5 3 1 2 4
1 5 2 4 3
2 4 5 3 1
4 1 3 5 2
```

Input2.txt

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 3 0
0 0 0 0 0

0 0 0 0
0 0 0 0
0 0 > 0
0 0 0 >
< < 0 >

^ 0 0 ^ 0
0 0 0 0 0
v 0 0 0 0
0 ^ 0 0 0
```

Output2.txt

```
1 5 3 2 4
3 2 1 4 5
5 4 2 1 3
4 1 5 3 2
2 3 4 5 1
```

Input3.txt

```
3 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
4 0 0 3 0

0 0 0 0
0 0 0 0
< < 0 0
0 0 < 0
> > 0 0

0 0 0 v 0
^ 0 0 0 0
0 0 0 0 0
0 ^ 0 0 0
```

Output3.txt

```
3 4 2 5 1
1 5 4 2 3
2 3 5 1 4
5 1 3 4 2
4 2 1 3 5
```

Futoshiki.py

```
from pynput import keyboard
from collections import deque, namedtuple
from os import sys
import os
import copy
import time
from rich.console import Console

board_size = 5
State = namedtuple('State', 'board constraints row_constraints col_constraints')
console = Console()

#Non blocking keyboard input gets stored in a queue
keypresses = deque()
listener = keyboard.Listener(on_press=keypresses.append)
listener.start()

class Puzzle:
    def __init__(self, input_file, output_file):

        board = None
        constraints = {}
        row_constraints = [set() for _ in range(board_size)]
        col_constraints = [set() for _ in range(board_size)]

        hconstraints = None
        vconstraints = None
        with open(input_file, 'r') as f:
            board = [[int(i) for i in f.readline().strip().split(' ')] for j in 
range(board_size)]
            f.readline()
            hconstraints = [[i for i in f.readline().strip().split(' ')] for j in 
range(board_size)]
            f.readline()
            vconstraints = [[i for i in f.readline().strip().split(' ')] for j in 
range(board_size)]
        self.output_file = output_file

        #Initialize all individual constraints to empty and add row and column 
constraints
        for row in range(board_size):
            for col in range(board_size):
                constraints[(row, col)] = {}
                if board[row][col] != 0:
                    row_constraints[row].add(board[row][col])
                    col_constraints[col].add(board[row][col])


        #Add horizontal inequality constraints
        for row in range(board_size):
            for col in range(board_size - 1):
                if hconstraints[row][col] != '0':
                    left = (row, col)
                    right = (row, col + 1)

                    constraint = hconstraints[row][col]
                    inv_constraint = '<' if constraint == '>' else '>'

                    constraints[left][right] = constraint

                    constraints[right][left] = inv_constraint

        #Add vertical inequality constraints
        for row in range(board_size - 1):
            for col in range(board_size):
                if vconstraints[row][col] != '0':
                    left = (row, col)
                    right = (row + 1, col)

                    constraint = '<' if vconstraints[row][col] == '^' else '>'
                    inv_constraint = '<' if constraint == '>' else '>'

                    constraints[left][right] = constraint
                    constraints[right][left] = inv_constraint

        self.state = State(board, constraints, row_constraints, col_constraints)

    def __str__(self):
        string = []
        for row in range(board_size):
            v_string = []
            for col in range(board_size):
                nxt = (row, col + 1)
                below = (row + 1, col)
                if self.state.board[row][col] == 0:
                    string.append('_')
                else:
                    string.append(str(self.state.board[row][col]))
                string.append(' ')
                if (row, col) in self.state.constraints and nxt in
self.state.constraints[(row, col)]:
                    string.append(self.state.constraints[(row, col)][nxt])
                else:
                    string.append(' ')
                if (row, col) in self.state.constraints and below in
self.state.constraints[(row, col)]:
```

```
                    v_string.append('^' if self.state.constraints[(row, col)]
[below] == '<' else 'v')
                else:
                    v_string.append(' ')
                v_string.append('   ')
                string.append(' ')
            string.append('\n')
            string.extend(v_string)
            string.append('\n')
        return ''.join(string)

    def print(self, selected = None, err = False):
        #Clear screen if windows:
        if sys.platform == 'win32':
            os.system('cls')
            #print('\033[2J', end = '')
        elif sys.platform == 'linux':
            print('\033[2J', end = '')
            pass
        if selected == None:
            console.print(self)
        else:
            repr_string = str(self)
            #calculate offset of selected tile in string
            #X coordinate:
            x_offset = 4 * selected[1]
            #Y coordinate:
            y_offset = 42 * selected[0]
            red_print = f'[bold red]{repr_string[x_offset + y_offset]}[/bold red]'
if not err else f'[bold red]X[/bold red]'
            console.print(repr_string[:x_offset + y_offset] + red_print +
repr_string[x_offset + y_offset + 1:])

    def play(self):
        initial_state = copy.deepcopy(self.state)
        immovable = set()
        for row in range(board_size):
            for col in range(board_size):
                if self.state.board[row][col] != 0:
                    immovable.add((row, col))
        selected = [0, 0]
        while True:
            self.print(selected = selected)
            input_key = None
            while input_key == None:

                if len(keypresses) > 0:
                    key = keypresses.popleft()
                    if key == keyboard.Key.esc:
                        sys.exit(0)
                    elif key == keyboard.KeyCode.from_char('s'):
                        solved_state = self.solve(initial_state)
                        if solved_state == None:
                            print('No solution found')
                            exit(1)
                        self.state = solved_state
                        self.print()
                        print('Solved!')
                        exit(0)
                    elif key == keyboard.Key.up:
                        selected[0] = max(0, selected[0] - 1)
                    elif key == keyboard.Key.down:
                        selected[0] = min(board_size - 1, selected[0] + 1)
                    elif key == keyboard.Key.left:
                        selected[1] = max(0, selected[1] - 1)
                    elif key == keyboard.Key.right:
                        selected[1] = min(board_size - 1, selected[1] + 1)
                    elif key == keyboard.KeyCode.from_char('1'):
                        input_key = 1
                        break
                    elif key == keyboard.KeyCode.from_char('2'):
                        input_key = 2
                        break
                    elif key == keyboard.KeyCode.from_char('3'):
                        input_key = 3
                        break
                    elif key == keyboard.KeyCode.from_char('4'):
                        input_key = 4
                        break
                    elif key == keyboard.KeyCode.from_char('5'):
                        input_key = 5
                        break
                    print('p')
                    self.print(selected = selected)

            #Draw board
            #Get player inputs
            is_valid = True
            is_overwrite = False
            if input_key in self.state.row_constraints[selected[0]] or input_key
in self.state.col_constraints[selected[1]]:
                is_valid = False
                self.print(selected = selected, err = True)
                time.sleep(0.2)
                continue
            if  (selected[0], selected[1]) in immovable:
                is_valid = False
                self.print(selected = selected, err = True)
                time.sleep(0.2)
            elif self.state.board[selected[0]][selected[1]] != 0:
                is_overwrite = True
            if (selected[0], selected[1]) in self.state.constraints:
                for other_tile in self.state.constraints[(selected[0],
selected[1])]:
                    if self.state.board[other_tile[0]][other_tile[1]] != 0:
                        if self.state.constraints[(selected[0], selected[1])]
[other_tile] == '<':
                            if self.state.board[other_tile[0]][other_tile[1]] <
```

```
input_key:
                                is_valid = False
                                self.print(selected = selected, err = True)
                                time.sleep(0.2)
                                break
                        else:
                            if self.state.board[other_tile[0]][other_tile[1]] >
input_key:
                                is_valid = False
                                self.print(selected = selected, err = True)
                                time.sleep(0.2)
                                break
            if is_valid:
                row = selected[0]
                col = selected[1]
                new_board = copy.deepcopy(self.state.board)
                new_board[row][col] = input_key
                new_row_constraints = copy.deepcopy(self.state.row_constraints)
                new_col_constraints = copy.deepcopy(self.state.col_constraints)
                if is_overwrite:
                    new_row_constraints[row].remove(self.state.board[row][col])
                    new_col_constraints[col].remove(self.state.board[row][col])
                new_row_constraints[row].add(input_key)
                new_col_constraints[col].add(input_key)

                new_state = State(new_board, self.state.constraints,
new_row_constraints, new_col_constraints)
                self.state = new_state

            #Check if move is valid
            #If valid, update board
            #Else, flash the X
            #Check if board is solved
        pass

    def solve(self, state = None):

        if state is None:
            state = self.state
        #So for this, we do backtracking search
        #The current state of the board is stored in a 2D array, and the
individual constraints will not change.
        #So we do best first where the heuristic is number of constraints, and
degree is the tie breaker
        #Each iteration, we pick the unassigned tile with the most constraints. If
there are multiple, we pick the one with the highest degree
        #We then assign it a legal value, and continue with this new state
        #If we reach a dead end, we backtrack until there are other legal moves
that we didn't try, then we try those.

        #We need a few helper functions:
        # select_next_assignment() - returns the next assignment to try
            #I think this should keep track of the number of constraints for each
tile, and the degree of each tile. Maybe some way to initialize it? Make it a
```

```
generator?
            #We should also consider that the number of constraints will change as
the tiles around the number get filled. So we should update the number of
constraints for each tile as we go
            #This should return the next tile to try.
        # get_legal_moves(row, col) - returns a list of legal moves for the given
tile
            #This needs to be in the order that we want to try the moves. So we
should try the moves in order of the number of constraints they will create
        # update_constraints(row, col) - updates the number of constraints for
each tile

        def get_legal_moves(state, row, col):
            all_moves = {1, 2, 3, 4, 5}
            moves = set()
            for move in all_moves:
                if move in state.row_constraints[row] or move in
state.col_constraints[col]:
                    continue
                is_valid = True
                for constraint in state.constraints[(row, col)]:
                    if state.board[constraint[0]][constraint[1]] == 0:
                        continue
                    if state.constraints[(row, col)][constraint] == '>':
                        if move <= state.board[constraint[0]][constraint[1]]:
                            is_valid = False
                            break
                    else:
                        if move >= state.board[constraint[0]][constraint[1]]:
                            is_valid = False
                            break
                if is_valid:
                    moves.add(move)
            return moves

        def select_next_assignment(state):
            max_constraints = 0
            max_constraints_tiles = set()
            #Find the tiles with the most constraints
            for row in range(board_size):
                for col in range(board_size):
                    if state.board[row][col] != 0:
                        continue

                    num_constraints = 5 - len(get_legal_moves(state, row, col))

                    if num_constraints > max_constraints:
                        max_constraints = num_constraints
                        max_constraints_tiles = {(row, col)}
                    elif num_constraints == max_constraints:
                        max_constraints_tiles.add((row, col))
            if len(max_constraints_tiles) == 0:
                return None #Need to backtrack
            elif len(max_constraints_tiles) == 1:
                return max_constraints_tiles.pop()

            #Find tile with highest degree
            #define degree as highest number of constraints on other tiless
            max_degree = 0
            max_degree_tiles = set()
            for tile in max_constraints_tiles:
                degree = 0
                for constraint in state.constraints[tile]:
                    if state.board[constraint[0]][constraint[1]] == 0:
                        degree += 1
                if degree > max_degree:
                    max_degree = degree
                    max_degree_tiles = {tile}
                elif degree == max_degree:
                    max_degree_tiles.add(tile)

            #Once we've finished, return any tile that works.
            return max_degree_tiles.pop()

        def generate_new_state(state, row, col, move):
            #return a new state with the next assignment, and with update row and
column constraints
            new_board = copy.deepcopy(state.board)
            new_board[row][col] = move
            new_row_constraints = copy.deepcopy(state.row_constraints)
            new_col_constraints = copy.deepcopy(state.col_constraints)
            new_row_constraints[row].add(move)
            new_col_constraints[col].add(move)
            new_state = State(new_board, state.constraints, new_row_constraints,
new_col_constraints)
            return new_state
            pass

        def is_finished(state):
            for row in range(board_size):
                for col in range(board_size):
                    if state.board[row][col] == 0:
                        return False
            return True
            #return true if the board is filled in and legal
            pass


        if is_finished(state):
            if self.output_file:
                with open(self.output_file, 'w') as f:
                    for row in state.board:
                        for col in row:
                            f.write(str(col) + " ")
                        f.write("\n")
            return state
        next_assignment = select_next_assignment(state)
        for move in get_legal_moves(state, next_assignment[0],
next_assignment[1]):
            new_state = generate_new_state(state, next_assignment[0],
next_assignment[1], move)
            result = self.solve(new_state)
            if result is not None:
                return result
        return None

input_file = None
output_file = None
auto_solve = False
for index, argument in enumerate(sys.argv):
    print(argument)
    if argument == "-i":
        input_file = sys.argv[index + 1]
    elif argument == "-o":
        output_file = sys.argv[index + 1]
    elif argument == '-s':
        auto_solve = True
    if input_file is None:
        print("Usage: python3 futoshiki.py -i <input_file> [-o <output_file>] [-
s]")
        print("Note: The default python installation on Windows does not allow for
command line arguments. You can enter them manually below.")
        input_file = input("Enter input file: ")
        output_file = input("Enter output file: ")
        if output_file == "":
            output_file = None
        auto_solve = input("Auto solve? (y/n): ") == 'y'
        keypresses.clear() #Clear this because if not it will read the keypresses
from the input prompt
    p = Puzzle(input_file, output_file)
    if auto_solve:
        s = p.solve()
        if s is not None:
            p.state = s
            p.print()
            print("Solved!")
        else:
            print("No solution found")
    else:
        p.play()
```