CS4613 Artificial Intelligence Project 2: Futoshiki

Seeger Zou, jz3714
Justin Cheok, jhc737
Fall 2022

Instructions:

1. Ensure that run.py is in the directory
2. Run the following command: `python run.py --infile arg1 --outfile arg2`, arg1 is the input filename and arg2 is the output filename. Example is shown below.

```
❯ python run.py --infile Input2.txt --outfile Output2.txt
```

3. The output file will be generated.

```
❯ cat 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
```

Source code:

puzzle.py

```
import numpy as np
import argparse

def listify(lst):
    '''
    turns the states into list of lists
    returnlst = list of lists (2D matrix for the 4x4 puzzle)
    '''
    returnlst = []
    for item in lst:
        nums = [int(x) if x.isnumeric() else x for x in item.split()]
        returnlst.append(nums)
    return returnlst

def printlst(lst, f):
    '''
    formatting the list to print out each item in the list
    '''
    for item in lst:
        for enter in item:
            f.write(str(enter))
            f.write(" ")
```

```
        f.write('\n')
    f.write('\n')

class BackTracker():
    class Board():
        def __init__(self, puzzle_arr, dom_arr, h_const, v_const, parent=None):
            self.puzzle = puzzle_arr # [[1,2,3],[3,2]]
            self.domains = dom_arr # list of list of sets [[{1,2,3,4,5},{},{}],[]]

            self.h_const = h_const
            self.v_const = v_const

            self.children = []
            self.parent = parent

            self.mrv_lst = []
            self.target = (0,0)
            self.target_vals = []
            self.target_index = 0

        def initialize(self):
            # initializes dom_arr to {1,2,3,4,5} for all cells
            self.domains = [[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]]

            for row in range(5):
                for col in range(5):
                    self.domains[row][col] = {1,2,3,4,5}

        def isValid(self):
            # checks for empty set:
            for row in range(5):
                for col in range(5):
                    if self.puzzle[row][col]==0 and len(self.domains[row][col])==0:
                        return False
            # row column checking
            for row in self.puzzle:
                ith_row = set(row).discard(0)
                if ith_row == None: # if it only contains 0, that's fine
                    continue
                elif len(ith_row) != np.count_nonzero(row):
                    return False
            for col in range(5):
                col_length = set(self.puzzle[:,col]).discard(0)
                if col_length == None: # if it only contains 0, that's fine
                    continue
                elif len(col_length) != np.count_nonzero(self.puzzle[:,col]):
                    return False

            # horizontal and vertical constraints checking
            for h_coord in self.h_const:
                right = (h_coord[0],h_coord[1]+1)
                left = h_coord
                ineq = self.h_const[h_coord]

                if self.puzzle[left[0]][left[1]]!=0 and self.puzzle[right[0]][right[1]]!=0:
                    if ineq == 1:
                        if self.puzzle[left[0]][left[1]] <= self.puzzle[right[0]][right[1]]:
```

```
                            return False
                    elif ineq == 0:
                        if self.puzzle[left[0]][left[1]] >= self.puzzle[right[0]][right[1]]:
                            return False

            for v_coord in self.v_const:
                up = v_coord
                down = (v_coord[0]+1,v_coord[1])
                ineq = self.v_const[v_coord]
                if self.puzzle[up[0]][up[1]]!=0 and self.puzzle[down[0]][down[1]]!=0:
                    if ineq == 1:
                        if self.puzzle[up[0]][up[1]] >= self.puzzle[down[0]][down[1]]:
                            return False
                    elif ineq == 0:
                        if self.puzzle[up[0]][up[1]] <= self.puzzle[down[0]][down[1]]:
                            return False
            return True

        def update(self):
            # updates domains
            # row column checking
            for row in range(5):
                for col in range(5):

                    curr = self.puzzle[row][col]
                    if curr!=0:
                        for i in range(5):
                            self.domains[row][i].discard(curr)
                        for j in range(5):
                            self.domains[j][col].discard(curr)

            # horizontal and vertical constraints checking
            for h_coord in self.h_const:
                right = (h_coord[0],h_coord[1]+1)
                left = h_coord
                ineq = self.h_const[h_coord]
                if self.puzzle[left[0]][left[1]]!=0 and self.puzzle[right[0]][right[1]]!=0: #
                    continue
                elif self.puzzle[left[0]][left[1]]!=0: # if left assigned
                    if ineq == 1:
                        self.domains[right[0]][right[1]] -= set([i for i in range(self.puzzle|
                    else:
                        self.domains[right[0]][right[1]] -= set([i for i in range(1,self.puzzl

                elif self.puzzle[right[0]][right[1]]!=0: # if right assigned
                    if ineq == 1:
                        self.domains[left[0]][left[1]] -= set([i for i in range(1, self.puzzle
                    else:
                        self.domains[left[0]][left[1]] -= set([i for i in range(self.puzzle[ri
            for v_coord in self.v_const:
                up = v_coord
                down = (v_coord[0]+1,v_coord[1])
                ineq = self.v_const[v_coord]
                if self.puzzle[up[0]][up[1]]!=0 and self.puzzle[down[0]][down[1]]!=0: # skip i
                    continue
                elif self.puzzle[up[0]][up[1]]!=0: # if up assigned
                    if ineq == 1:
```

```
                        self.domains[down[0]][down[1]] -= set([i for i in range(1,self.puzzle[
                    else:
                        self.domains[down[0]][down[1]] -= set([i for i in range(self.puzzle[up

                elif self.puzzle[down[0]][down[1]]!=0: # if down assigned
                    if ineq == 1:
                        self.domains[up[0]][up[1]] -= set([i for i in range(self.puzzle[down[0
                    else:
                        self.domains[up[0]][up[1]] -= set([i for i in range(1,self.puzzle[down


        def chooseTargetVal(self):
            mrv_lst = []
            least = 6 # keeps the shortest domains, i.e. the minimum remaining values
            for row in range(len(self.domains)):
                for col in range(len(self.domains[row])):

                    if self.puzzle[row][col] != 0: # if already assigned, skip
                        continue

                    curr_domain_len = len(self.domains[row][col])
                    if curr_domain_len < least:
                        least = curr_domain_len

                        mrv_lst = [(row,col)]
                    elif curr_domain_len == least:
                        mrv_lst.append((row,col))
            if len(mrv_lst) == 0: # shouldn't happen?
                print('MRV_lst has zero elements')
                pass

            elif len(mrv_lst) > 1:
                # degree heuristic
                degree_lst = [0]*len(mrv_lst)
                for i in range(len(mrv_lst)):
                    degree = 0
                    row,col = mrv_lst[i][0], mrv_lst[i][1]
                    # left
                    if col > 0:
                        if self.puzzle[row][col-1]==0:
                            degree+=1
                    # right
                    if col < 4:
                        if self.puzzle[row][col+1]==0:
                            degree+=1
                    # up
                    if row > 0:
                        if self.puzzle[row-1][col]==0:
                            degree+=1
                    # down
                    if row < 4:
                        if self.puzzle[row+1][col]==0:
                            degree+=1

                    degree_lst[i] = degree
                self.target = mrv_lst[degree_lst.index(max(degree_lst))]
            else:
```

```
191                 self.target = mrv_lst[0]
192             self.target_vals = list(self.domains[self.target[0]][self.target[1]])
193 
194 
195         def isComplete(self):
196             return not (0 in self.puzzle)
197 
198     def __init__(self, initial_arr, initial_dom_arr, h_const, v_const):
199         self.root = self.Board(initial_arr, initial_dom_arr, h_const, v_const)
200 
201 
202     def solve(self):
203         curr = self.root
204         curr.initialize()
205         curr.update()
206 
207         while True:
208             # find the target to select values
209             if curr.isComplete() and curr.isValid():
210                 return curr.puzzle
211 
212             # going down
213             elif not curr.isComplete() and curr.isValid():
214                 curr.chooseTargetVal()
215                 new_puzzle = copy.deepcopy(curr.puzzle)
216                 new_domains = copy.deepcopy(curr.domains)
217                 new_insert_pos = curr.target
218                 new_puzzle[new_insert_pos[0]][new_insert_pos[1]] = curr.target_vals[curr.targe
219 
220                 curr.children.append(self.Board(new_puzzle,new_domains,curr.h_const,curr.v_cor
221                 curr = curr.children[curr.target_index]
222                 curr.update()
223 
224             elif not curr.isValid():
225                 curr = curr.parent
226                 while curr.target_index+1 >= len(curr.target_vals):
227                     curr = curr.parent
228                 curr.target_index+=1
229 
230 
231 def gen_constraints(horiz, vert):
232     HConstraints = {}
233     VConstraints = {}
234     constraint_dict = {'^' : 1,'>' : 1, '<' : 0, 'v' : 0}
235     for line in range(len(horiz)):
236         for i in range(len(horiz[line])):
237             lin = horiz[line]
238             if lin[i] in constraint_dict.keys():
239                 HConstraints[(line, i)] = constraint_dict[lin[i]]
240     for line in range(len(vert)):
241         for i in range(len(vert[line])):
242             lin = vert[line]
243             if lin[i] in constraint_dict.keys():
244                 VConstraints[(line, i)] = constraint_dict[lin[i]]
245     return [HConstraints, VConstraints]
246 
```

```
247 def printlst(lst, f):
248     '''
249     formatting the list to print out each item in the list
250     '''
251     for item in lst:
252         for enter in item:
253             f.write(str(enter))
254             f.write(" ")
255         f.write('\n')
256     f.write('\n')
257 
258 def main():
259     parser = argparse.ArgumentParser(description='Futoshiki solver')
260     parser.add_argument('--infile',type=argparse.FileType('r'),help='input file')
261     parser.add_argument('--outfile',type=argparse.FileType('w'),help='output file')
262     args = parser.parse_args()
263     infile = args.infile.readlines()
264     outfile = args.outfile
265 
266     lines = listify(infile)
267     inpdata = lines[0:5]
268     horiz = lines[6:11]
269     vert = lines[12:17]

270 
271     test = np.array(inpdata)
272     constraints = gen_constraints(horiz, vert)
273     h_const = constraints[0]
274     v_const = constraints[1]
275 
276     solver = BackTracker(test,[],h_const,v_const)
277     solution = solver.solve()
278     printlst(solution, outfile)
279 main()
280 
```

Output Files

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

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

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