Contents

1	Basic Test Results	2
2	puzzle solver.py	3

1 Basic Test Results

```
Starting tests...
1
    Tue Nov 30 11:36:18 IST 2021
    97a0f83f46dfdb6e300a64c1d6a31cfd1e635555 -
4
    Archive: /tmp/bodek.Z_cuBu/intro2cs1/ex8/eyalmutzary/presubmission/submission
     inflating: src/puzzle_solver.py
9
10
   Running presubmit code tests...
    --> BEGIN TEST INFORMATION
11
   Test name: presubmit_generate
12
   Module tested: puzzle_solver
   Function call: generate_puzzle([[1]])
14
15
   Expected return value: {(0, 0, 1)}
   More test options: {}
   --> END TEST INFORMATION
17
   *************************
   ******
19
                       There is a problem:
   *******
                        The test named 'presubmit_generate' failed.
20
21
    **********************
   Wrong result, input: [[[1]]]:
22
23
   expected: {(0, 0, 1)}
   actual: None
24
   result_code presubmit_generate
25
                                   wrong
   5 passed tests out of 6 in test set named 'presubmit'.
   result_code presubmit 5 1
27
   Done running presubmit code tests
28
29
   Finished running the presubmit tests
30
31
32
    Additional notes:
33
   The presubmit tests check only for the existence of the correct function names.
34
35
    Make sure to thoroughly test your code.
```

2 puzzle solver.py

```
from typing import List, Tuple, Set, Optional
1
3
    # We define the types of a partial picture and a constraint (for type checking).
4
    Picture = List[List[int]]
    Constraint = Tuple[int, int, int]
8
    # ----- Prolog -----
9
10
11
    def create_default_picture(n: int, m: int) -> Picture:
12
        picture = []
        for i in range(n):
14
15
            picture.append([])
16
            for _ in range(m):
                picture[i].append(-1)
17
18
        return picture
19
20
21
    def append_constraints_set(picture: Picture, constraint_set: Set[Constraint]) -> List[List[int]]:
        if constraint_set == set():
22
23
            return picture
        for constraint in constraint_set:
24
          picture[constraint[0]][constraint[1]] = constraint[2]
25
26
        return picture
27
28
29
    def print_picture(picture: List[List[int]]) -> None:
        for i in range(len(picture)):
30
            for j in range(len(picture[i])):
31
                if picture[i][j] == -1:
                    print(" ? " , end="")
33
                \# elif picture[i][j] == 0:
34
                # print(" ", end="")
35
                # elif picture[i][j] == 1:
36
37
                # print(" " , end="")
38
                    print(" " + str(picture[i][j]) + " " , end="")
39
            print()
40
41
42
    # ----- Part 1 -----
43
44
45
    def _should_break(num: int, is_max: bool) -> bool:
46
47
        if is_max and num == 0:
            return True
48
        elif not is_max and num <= 0:</pre>
49
50
           return True
        return False
51
52
53
    def _seen_row(row: List[int], col: int, is_max: bool):
54
        count: int = 0
55
        for i in range(col, len(row), 1):
            if _should_break(row[i], is_max):
57
                break
            count += 1
```

```
60
         for i in range(col-1, -1, -1):
 61
             if _should_break(row[i], is_max):
 62
                  break
              count += 1
 63
          if row[col] != 0:
 64
             count -= 1
 65
          return count
 66
 67
 68
     def _seen_col(picture: Picture, row: int, col: int, is_max: bool):
 69
 70
          count: int = 0
 71
          for i in range(row, len(picture), 1):
 72
             if _should_break(picture[i][col], is_max):
 73
                 break
 74
             count += 1
         for i in range(row, -1, -1):
 75
             if _should_break(picture[i][col], is_max):
 76
 77
                  break
             count += 1
 78
 79
         if picture[row][col] != 0:
 80
             count -= 1
 81
         return count
 82
 83
 84
 85
     def max_seen_cells(picture: Picture, row: int, col: int) -> int:
 86
 87
          if picture[row][col] == 0:
             return 0
 88
 89
          else:
 90
             return _seen_row(picture[row], col, True) +\
                     _seen_col(picture, row, col, True)
 91
 92
 93
     # tested
 94
 95
     def min_seen_cells(picture: Picture, row: int, col: int) -> int:
          if picture[row][col] <= 0:</pre>
 96
 97
             return 0
 98
             return _seen_row(picture[row], col, False) + \
 99
100
                  _seen_col(picture, row, col, False)
101
102
103
     # tested
     def check_constraints(picture: Picture, constraints_set: Set[Constraint]) -> int:
104
105
         status = 1
106
         if constraints_set == set():
             return 1
107
108
109
         for const in constraints_set:
             min_seen = min_seen_cells(picture, const[0], const[1])
110
111
              max_seen = max_seen_cells(picture, const[0], const[1])
112
             if const[2] < min_seen or const[2] > max_seen:
                  return 0
113
              elif const[2] == min_seen and const[2] == max_seen:
114
                  continue
115
              elif min_seen <= const[2] <= max_seen:</pre>
116
117
                 status = 2
         return status
118
119
120
121
     # def copy_picture(picture: Picture):
122
           new_picture = []
           for i in range(len(picture)):
123
     #
124
               new_picture.append([])
125
     #
               for j in range(len(picture[i])):
                   new_picture[i].append(picture[i][j])
126
    #
127
           return new_picture
```

```
128
129
     def formal_solution(picture):
130
         new_picture = []
131
         for i in range(len(picture)):
132
133
             new_picture.append([])
              for j in range(len(picture[i])):
134
                  if picture[i][j] == 0:
135
136
                      new_picture[i].append(0)
137
                  else:
                      new_picture[i].append(1)
138
139
          return new_picture
140
141
142
     def _solve_puzzle_helper(picture: Picture,
                               ind: int.
143
144
                               constraints_set: Set[Constraint],
                               sol: List[Picture]) -> Optional[Picture]:
145
          check = check_constraints(picture, constraints_set)
146
147
          if ind == len(picture) * len(picture[0]):
              if check == 1 and len(sol) == 0:
148
149
                  sol.append(formal_solution(picture))
150
              return picture
151
152
         row, col = ind // len(picture[0]), ind % len(picture[0])
153
         if picture[row][col] != -1:
154
155
              _solve_puzzle_helper(picture, ind + 1, constraints_set, sol)
156
157
158
         for value in (0, 1):
             if len(sol) == 1 or check == 0:
159
160
                  {\tt return}
161
             picture[row][col] = value
              _solve_puzzle_helper(picture, ind + 1, constraints_set, sol)
162
163
         picture[row][col] = -1
164
165
     def solve_puzzle(constraints_set: Set[Constraint], n: int, m: int) -> Optional[Picture]:
166
         picture = create_default_picture(n, m)
167
168
          append_constraints_set(picture, constraints_set)
          sol = []
169
          _solve_puzzle_helper(picture, 0, constraints_set, sol)
170
171
          if len(sol) > 0:
             return sol[0]
172
173
          else:
             return None
174
175
176
177
     def _count_solutions(picture: Picture,
178
179
                           ind: int,
180
                           constraints_set: Set[Constraint],
181
                           counter: List[int]) -> None:
          check = check_constraints(picture, constraints_set)
182
          if ind == len(picture) * len(picture[0]):
183
              if check == 1:
184
                  counter[0] += 1
185
             return
186
187
         row, col = ind // len(picture[0]), ind % len(picture[0])
188
189
190
          if picture[row][col] != -1:
              _count_solutions(picture, ind + 1, constraints_set, counter)
191
192
              return
193
         for value in (0, 1):
194
195
             if check == 0:
```

```
196
                 return
197
             picture[row][col] = value
             _count_solutions(picture, ind + 1, constraints_set, counter)
198
199
         picture[row][col] = -1
200
201
202
     def how_many_solutions(constraints_set: Set[Constraint], n: int, m: int) -> int:
203
204
         picture = create_default_picture(n, m)
         append_constraints_set(picture, constraints_set)
205
         counter = [0]
206
         _count_solutions(picture, 0, constraints_set, counter)
207
         return counter[0]
208
209
210
211
212
213
         - switch all 1 to -1
         - loop on the picture:
214
             - if value is 0 or -2 -> continue
215
216
             - else:
                 - go for min scan.
217
218
                 - set the value to the min score
                 - set -2 to every square around
219
^{220}
221
     def generate_puzzle(picture: Picture) -> Set[Constraint]:
222
223
```