

Contents

1	Basic Test Results	2
2	puzzle solver.py	3

1 Basic Test Results

```
1 Starting tests...
2 Tue Nov 30 11:36:18 IST 2021
3 97a0f83f46dfdb6e300a64c1d6a31cfd1e635555 -
4
5
6 Archive: /tmp/bodek.Z_cuBu/intro2cs1/ex8/eyalmutzary/presubmission/submission
7 inflating: src/puzzle_solver.py
8
9
10 Running presubmit code tests...
11 --> BEGIN TEST INFORMATION
12 Test name: presubmit_generate
13 Module tested: puzzle_solver
14 Function call: generate_puzzle([[1]])
15 Expected return value: {(0, 0, 1)}
16 More test options: {}
17 --> END TEST INFORMATION
18 *****
19 ***** There is a problem:
20 ***** The test named 'presubmit_generate' failed.
21 *****
22 Wrong result, input: [[[1]]]:
23 expected: {(0, 0, 1)}
24 actual: None
25 result_code presubmit_generate wrong 1
26 5 passed tests out of 6 in test set named 'presubmit'.
27 result_code presubmit 5 1
28 Done running presubmit code tests
29
30 Finished running the presubmit tests
31
32 Additional notes:
33
34 The presubmit tests check only for the existence of the correct function names.
35 Make sure to thoroughly test your code.
36
```

2 puzzle solver.py

```
1  from typing import List, Tuple, Set, Optional
2
3
4  # We define the types of a partial picture and a constraint (for type checking).
5  Picture = List[List[int]]
6  Constraint = Tuple[int, int, int]
7
8
9  # ----- Prolog -----
10
11
12  def create_default_picture(n: int, m: int) -> Picture:
13      picture = []
14      for i in range(n):
15          picture.append([])
16          for _ in range(m):
17              picture[i].append(-1)
18      return picture
19
20
21  def append_constraints_set(picture: Picture, constraint_set: Set[Constraint]) -> List[List[int]]:
22      if constraint_set == set():
23          return picture
24      for constraint in constraint_set:
25          picture[constraint[0]][constraint[1]] = constraint[2]
26      return picture
27
28
29  def print_picture(picture: List[List[int]]) -> None:
30      for i in range(len(picture)):
31          for j in range(len(picture[i])):
32              if picture[i][j] == -1:
33                  print(" ? ", end="")
34              # elif picture[i][j] == 0:
35              #     print(" ", end="")
36              # elif picture[i][j] == 1:
37              #     print(" ", end="")
38              else:
39                  print(" " + str(picture[i][j]) + " ", end="")
40      print()
41
42
43  # ----- Part 1 -----
44
45
46  def _should_break(num: int, is_max: bool) -> bool:
47      if is_max and num == 0:
48          return True
49      elif not is_max and num <= 0:
50          return True
51      return False
52
53
54  def _seen_row(row: List[int], col: int, is_max: bool):
55      count: int = 0
56      for i in range(col, len(row), 1):
57          if _should_break(row[i], is_max):
58              break
59      count += 1
```

```

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

```

```

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

```

```

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

```