Contents

1	Basic Test Results	2
2	ex11.py	5

1 Basic Test Results

```
Tue Jan 12 16:08:12 IST 2021
   Tue Jan 12 16:08:12 IST 2021
   Archive: /tmp/bodek.KRNToq/intro2cs1/ex11/tsviel/final/submission
     inflating: src/ex11.py
     inflating: src/__MACOSX/._ex11.py
   10 passed tests out of 10 in test set named 'presubmit'.
   result_code
               presubmit
                            10
   16 passed tests out of 16 in test set named 'diagnose1'.
   result_code diagnose1
                           16
   16 passed tests out of 16 in test set named 'diagnose2'.
   result_code diagnose2
11
                           16
   16 passed tests out of 16 in test set named 'diagnose3'.
12
   result_code diagnose3
                           16
   16 passed tests out of 16 in test set named 'diagnose4'.
14
15
   result_code diagnose4
                            16
   4 passed tests out of 4 in test set named 'calcsuccess'.
   result_code calcsuccess 4 1
17
   4 passed tests out of 4 in test set named 'allillnesses'.
   result_code allillnesses 4
19
   --> BEGIN TEST INFORMATION
20
21
   Test name: pathsillness_11
   Module tested: ex11
22
23
   Function call: paths_to_illness('take-coat')
   Expected return value: ([[False, True], [True]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
   --> END TEST INFORMATION
25
   ****************************
26
   ******
                       There is a problem:
27
                        The test named 'pathsillness_11' failed.
   ******
28
   *************************
   Wrong result, input: ['take-coat']:
30
   expected: ([[False, True], [True]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
31
   actual: ([], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
   result code
                pathsillness_11
                                wrong
33
34
    --> BEGIN TEST INFORMATION
   Test name: pathsillness_12
35
   Module tested: ex11
36
   Function call: paths_to_illness('no-coat')
37
   Expected return value: ([[False, False]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
38
39
   --> END TEST INFORMATION
    *************************
                       There is a problem:
   ******
41
   *******
                        The test named 'pathsillness_12' failed.
42
    *************************
43
   Wrong result, input: ['no-coat']:
44
   expected: ([[False, False]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
   actual: ([], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
46
47
   result_code
                pathsillness_12 wrong
   9 passed tests out of 11 in test set named 'pathsillness'.
   result_code pathsillness 9 1
49
   3 passed tests out of 3 in test set named 'buildtree'.
50
   result_code buildtree 3 1
51
   10 passed tests out of 10 in test set named 'optimaltree'.
52
   result_code optimaltree
                              10
    --> BEGIN TEST INFORMATION
54
55
   Test name: minimize_f0
   Module tested: ex11
57 Function call: minimize(False)
58 Expected return value: (None, ('b', ('d', None)))
   --> END TEST INFORMATION
```

```
*************************
60
    ******
61
                        There is a problem:
                        The test named 'minimize_f0' failed.
62
    ******
    ********************
    Wrong result, input: [False]:
64
65
    expected: (None, ('b', ('d', None)))
    actual: (None, ('a', (('b', ('d', None)), ('b', ('d', None)))))
    result code minimize f0
                            wrong
67
68
    --> BEGIN TEST INFORMATION
    Test name: minimize_f1
69
70
    Module tested: ex11
    Function call: minimize(False)
71
    Expected return value: (None, ('b', ('d', 'e')))
72
    --> END TEST INFORMATION
73
    ***********************
                      There is a problem:
    ******
75
76
    ******
                       The test named 'minimize_f1' failed.
    **************************
77
    Wrong result, input: [False]:
78
    expected: (None, ('b', ('d', 'e')))
    actual: (None, ('a', (('b', ('d', 'e')), ('b', ('d', 'e')))))
80
    result_code minimize_f1
81
                            wrong
    --> BEGIN TEST INFORMATION
83
    Test name: minimize f4
84
    Module tested: ex11
    Function call: minimize(False)
85
    Expected return value: (None, ('a', (('b', ('d', None)), None)))
86
87
    --> END TEST INFORMATION
    *************************
88
89
    ******
                        There is a problem:
90
    ******
                        The test named 'minimize_f4' failed.
    *************************************
91
92
    Wrong result, input: [False]:
    expected: (None, ('a', (('b', ('d', None)), None)))
actual: (None, ('a', (('b', ('d', None)), ('b', (None, None)))))
93
94
    result_code minimize_f4
95
                            wrong
                                   1
96
    --> BEGIN TEST INFORMATION
    Test name: minimize_f5
97
    Module tested: ex11
    Function call: minimize(False)
99
    Expected return value: (None, ('a', (('b', ('d', None)), 'd')))
100
    --> END TEST INFORMATION
101
    *************************
102
103
    ****** There is a problem:
    ******
                       The test named 'minimize_f5' failed.
104
    *************************
105
    Wrong result, input: [False]:
    expected: (None, ('a', (('b', ('d', None)), 'd')))
107
108
    actual: (None, ('a', (('b', ('d', None)), ('b', ('d', 'd')))))
109
    result_code minimize_f5
                            wrong
    --> BEGIN TEST INFORMATION
110
    Test name: minimize_t0
111
112
    Module tested: ex11
113
    Function call: minimize(True)
    Expected return value: (None, 'd')
114
    --> END TEST INFORMATION
115
116
    ***********************************
    ******
                       There is a problem:
117
    ******
                       The test named 'minimize_t0' failed.
118
    *************************
119
120
    return value could not be expanded
               minimize_t0
121
    result code
                            recovertree 1
    --> BEGIN TEST INFORMATION
    Test name: minimize_t1
123
124
   Module tested: ex11
    Function call: minimize(True)
Expected return value: (None, ('b', ('d', 'e')))
127 --> END TEST INFORMATION
```

```
128
   ***********************
129
    ******
                     There is a problem:
                      The test named 'minimize_t1' failed.
130
    ******
    ************************
    Wrong result, input: [True]:
132
    expected: (None, ('b', ('d', 'e')))
133
    actual: (None, ('a', (('b', ('d', 'e')), ('b', ('d', 'e')))))
   result_code minimize_t1
                         wrong 1
135
136
    --> BEGIN TEST INFORMATION
    Test name: minimize_t3
137
   Module tested: ex11
138
139
    Function call: minimize(True)
   Expected return value: (None, 'd')
140
   --> END TEST INFORMATION
141
    ***********************
    ****** There is a problem:
143
                     The test named 'minimize_t3' failed.
144
    ******
    *******************
145
   return value could not be expanded
146
   result_code
              minimize_t3
                         recovertree
147
    --> BEGIN TEST INFORMATION
148
149
   Test name: minimize_t4
   Module tested: ex11
150
   Function call: minimize(True)
151
   Expected return value: (None, 'd')
152
    --> END TEST INFORMATION
153
    *************************
154
155
    ******* There is a problem:
    ******
                     The test named 'minimize_t4' failed.
156
157
    **********************
158
   return value could not be expanded
   result code minimize t4
159
                         recovertree
   --> BEGIN TEST INFORMATION
160
161
    Test name: minimize_t5
   Module tested: ex11
162
   Function call: minimize(True)
163
    Expected return value: (None, 'd')
164
    --> END TEST INFORMATION
165
    ************************
    ******
                    There is a problem:
The test named 'minimize_t5' failed.
167
168
    ******
169
    *************************
   return value could not be expanded
170
171
    result_code minimize_t5 recovertree
   3 passed tests out of 12 in test set named 'minimize'.
172
   result_code minimize 3
173
                          1
   TESTING COMPLETED
```

2 ex11.py

```
# FILE : ex11.py
   # WRITER : TSVIEL ZAIKMAN , tsviel , 208241133
   # EXERCISE : intro2cs1 ex11 2021
    # DESCRIPTION: Traverse on medical records tree Graphs
   from collections import Counter
   from itertools import combinations
10
   EMPTY = 0 # LENGTH IS 0
    MAX = 0 # MAXIMAL VALUE
11
   TYPE_ERROR_MESSAGE = "Either your symptom or record is invalid" \
12
                       "- Raised TypeError"
   VALUE_ERROR_MESSAGE = "A Value Error has been raised"
14
15
16
    class Node:
17
18
       def __init__(self, data, positive_child=None, negative_child=None):
19
           self.data = data
           self.positive_child = positive_child
20
           self.negative_child = negative_child
21
22
23
    class Record:
24
       def __init__(self, illness, symptoms):
25
26
           self.illness = illness
27
           self.symptoms = symptoms
28
29
    def parse_data(filepath):
30
       with open(filepath) as data_file:
31
           records = []
           for line in data_file:
33
               words = line.strip().split()
34
               records.append(Record(words[0], words[1:]))
35
           return records
36
37
38
    def is_leaf(tree_node):
39
40
        """Return True if node is leaf, false if not None"""
       if tree_node.positive_child is None and tree_node.negative_child is None:
41
42
           return True
       return False
43
44
45
    def empty(tree_node):
46
        """Return True if node is None, false if not None"""
47
        if tree_node.data is None:
           return True
49
50
       else:
          return False
51
52
53
    class Diagnoser:
54
55
       def __init__(self, root: Node):
           self.root = root
57
58
       def diagnose(self, symptoms):
           return self._diagnose_helper(symptoms, self.root)
```

```
60
         def _diagnose_helper(self, symptoms, tree_root):
61
62
              :param symptoms: a list of symptoms (strings)
63
             :param tree root (object)
64
             :return: name of illness located on leaf
65
66
             if is_leaf(tree_root):
67
68
                 return tree_root.data
             if tree_root.positive_child is not None and tree_root.data in symptoms:
69
                 return self._diagnose_helper(symptoms, tree_root.positive_child)
70
71
             if tree_root.negative_child is not None:
                 return self._diagnose_helper(symptoms, tree_root.negative_child)
72
73
74
         def calculate_success_rate(self, records):
75
76
              :param self: root of choice tree
              :param records: list of records objects
77
             :return: the ratio between amount of success on records to amount of
78
             records in total
79
80
             records_length = len(records)
81
82
             try:
83
                 if records_length == EMPTY:
84
                      raise ValueError
                  success = 0 # Counter
85
                 for record in records:
86
87
                      illness, symptoms = record.illness, record.symptoms
                      if self.diagnose(symptoms) == record.illness:
88
89
                          success += 1
90
                  return success / records_length
             except ValueError:
91
                 return "Your records list is empty. " + VALUE_ERROR_MESSAGE
92
93
         def all illnesses(self):
94
95
              """returns all illnesses"""
96
             res = self.__sort_all_illnesses_list(
97
                 self.__all_illnesses(self.root, []))
             return res
99
100
         def __all_illnesses(self, tree_node, illnesses_list):
              """recursive helper to all_illnesses"""
101
             if tree_node.positive_child is not None:
102
103
                  self.__all_illnesses(tree_node.positive_child, illnesses_list)
             if tree_node.negative_child is not None:
104
                 self.__all_illnesses(tree_node.negative_child, illnesses_list)
105
106
              if is_leaf(tree_node) and not empty(tree_node):
                  # if Leaf and not None
107
108
                  illnesses_list.append(tree_node.data)
109
             return illnesses_list
110
         def __sort_all_illnesses_list(self, illnesses_to_sort):
111
112
               ""Sort by occurrences and remove duplicates from illnesses list
             using dictionary's properties of orderd keys"""
113
             if len(illnesses_to_sort) <= 1:</pre>
114
                 return illnesses_to_sort
115
116
117
             sorted_ilnesses = sorted(illnesses_to_sort,
                                       key=illnesses_to_sort.count,
118
119
                                       reverse=True)
             sorted_and_unique_illnesses_lst = list(dict.fromkeys(sorted_ilnesses))
120
121
             return sorted_and_unique_illnesses_lst
122
         def paths_to_illness(self, illness):
123
               ""Return all path to illness (list of bools)"""
124
             return self.__paths_to_illness(self.root, illness)
125
126
127
         def __paths_to_illness(self, node, illness):
```

```
128
              """Recursive Helper of path to illness """
             if is_leaf(node):
129
                  # Stop condition - We hit leaf
130
                  if illness is node.data: # If we reach our illness
131
132
                      return [[]]
                  return [] # If its not our illness
133
                  # Run recursion on the left branch
134
135
136
             negative_path = self.__paths_to_illness(node.negative_child, illness)
              # Run recursion on the right branch
137
             positive_path = self.__paths_to_illness(node.positive_child, illness)
138
139
             paths = [] # Create new list for routes
             for route in negative_path: # append left direction
140
141
                  paths.append([False] + route)
142
             for route in positive_path: # Append right direction
                 paths.append([True] + route)
143
144
             return paths # Return all routes of tree
145
         def remove_half_nodes(self, tree_node):
146
              """Remove all half nodes from tree"""
147
             if tree_node is None:
148
149
                 return None
150
             if tree_node.positive_child is not None \
                      and tree_node.negative_child is not None:
151
152
                  tree_node = self.__remove_child_duplications(tree_node)
153
             tree_node.negative_child = self.remove_half_nodes(
                  tree_node.negative_child) # Recur to left tree
154
155
              tree_node.positive_child = self.remove_half_nodes(
                  tree_node.positive_child) # Recur to right tree
156
157
             if is_leaf(tree_node): # We hit leaf
158
                  return tree_node
             if tree_node.negative_child is None:
159
160
                  next_root = tree_node.positive_child
                  prev_root, tree_node = tree_node, None
161
162
                  del prev_root
                  return next_root
163
164
             if tree_node.positive_child is None:
165
                  next_root = tree_node.negative_child
166
                  prev_root, tree_node = tree_node, None
                  del prev_root
167
168
                  return next_root
             return tree_node
169
170
171
         def __remove_child_duplications(self, node):
172
173
              :param node: a give node object which has 2 children
174
              :return: a node only with the positive child
175
176
             if node.positive_child.data == node.negative_child.data:
177
                 node = node.positive_child
             return node
178
179
180
         def __delete_leaves(self, tree_node, val):
181
              :param tree_node: a given tree object
182
              :param val: a value we wish to remove
183
184
              :return: remove the leaves with the given value
              n n n
185
             if tree_node is None:
186
187
188
             tree_node.negative_child = self.__delete_leaves(
                  tree_node.negative_child, val)
189
             tree_node.positive_child = self.__delete_leaves(
190
                  tree_node.positive_child, val)
191
192
193
             if tree_node.data == val and is_leaf(tree_node):
194
                 return
195
             return tree_node
```

```
196
197
         def minimize(self, remove_empty=False):
              """The function minimize the tree and removes unnesseary path"""
198
199
              if remove_empty:
                  self.__delete_leaves(self.root, None)
200
201
              self.remove_half_nodes(self.root)
202
203
204
     def _build_tree_helper(records):
205
          """helper of build_tree for counting"""
206
207
          if len(records) is EMPTY:
208
             return None
209
          illnesses = []
210
         for i in range(len(records)):
             illnesses.append(records[i].illness)
211
212
          counts = Counter(illnesses)
213
          # return sorted list by prevalence
         return sorted(counts, key=counts.get, reverse=True)[0]
214
215
216
217
     def _build_tree(records, symptoms):
218
219
          recursive function to build a tree
220
          :param records: list of record objects
221
          :param symptoms: list of strings representing symptoms of illness
          :return: Diagnoser Object for the tree we build
222
223
         if len(symptoms) == EMPTY:
224
225
             return Node(_build_tree_helper(records))
226
         first_symptom, rest_symptoms = symptoms[0], symptoms[1:]
227
228
229
         positive_path = _build_tree([record for record in records if
                                       first_symptom
230
231
                                        in record.symptoms], rest_symptoms)
232
         negative_path = _build_tree([record for record in records if
233
                                       first_symptom
234
                                       not in record.symptoms], rest_symptoms)
235
236
          final_tree = Node(first_symptom, positive_path, negative_path)
237
         return final tree
238
239
240
     def build_tree(records, symptoms):
241
242
          :param records: a list of records(objects)
243
244
          :param symptoms: a list of symptoms(strings)
245
          :return: Diagnoser object based on the built_tree
246
247
          try:
248
              symptoms_are_strings = [type(item) == str for item in symptoms]
              records_are_record = [type(record) == Record for record in records]
249
              if not all(symptoms_are_strings) and all(records_are_record):
250
251
                  # Logic of statement is based on de-morgan law
252
                  raise TypeError
253
             return Diagnoser(_build_tree(records, symptoms))
          except TypeError:
254
255
              return TYPE_ERROR_MESSAGE
256
257
258
     def _optimal_tree(records, symptoms, depth):
259
          Returns the optimal tree for different subsets of symptoms with size of
260
          depth. The function iterates on all of the subsets of symptoms in size
261
          and depth for each time we create a tree Node.
262
263
          The function gets the rates for success, update the maximal value(tuple)
```

```
264
          : param\ records \colon\ list\ of\ Records
265
          :param symptoms: list of symptoms
          :param depth: the size of the subset of symptoms
266
267
          :return: an optimal tree node object
268
         max_val = None, MAX
269
          combs = combinations(symptoms, depth) # all combinations of symptoms
270
          # and depth using itertools lib
271
272
         for comb in combs:
             diagnostic = build_tree(records, comb) # Build subset tree of combs
273
              \# Checks the success rates for records
274
275
              success_rates = diagnostic.calculate_success_rate(records)
276
              if success_rates >= max_val[1]:
277
                  # case the success rates of this tree are higher than the
278
                  \# maximal\_value
                  max_val = diagnostic.root, success_rates # update it.
279
280
281
          if type(max_val[0]) is not Node:
              return Node("") # Case we fail to produce node
282
283
284
         return max_val[0]
285
286
287
     def optimal_tree(records, symptoms, depth):
288
289
          :param records: a list of objects of Record type
          : param\ symptoms:\ a\ list\ of\ strings\ representing\ symptoms
290
          :param depth: the depth of the tree required
291
          :return: a Diagnostic based on the optimal tree
292
293
294
          try:
              optimal_tree_node = _optimal_tree(records, symptoms, depth)
295
              depth_property = (len(symptoms) >= depth >= 0)
296
297
              contains_duplicates = any(
                  symptoms.count(element) > 1 for element in symptoms)
298
299
300
              if not depth_property or contains_duplicates:
                  raise ValueError
301
302
              symptoms_are_strings = [type(item) == str for item in symptoms]
303
              records_are_record = [type(record) == Record for record in records]
304
305
              if not all(symptoms_are_strings) and all(records_are_record):
306
307
                  raise TypeError
308
              return Diagnoser(optimal_tree_node)
309
310
          except ValueError:
             return "Either The symptoms doesn't meet the depth " \
311
                                    "property or it contains duplicates. " + \
312
                                    VALUE_ERROR_MESSAGE
313
          except TypeError:
314
315
              return TYPE_ERROR_MESSAGE
316
317
     if __name__ == "__main__":
318
319
         pass
```