

Contents

1	Basic Test Results	2
2	ex11.py	5

1 Basic Test Results

```
1 Tue Jan 12 16:08:12 IST 2021
2 Tue Jan 12 16:08:12 IST 2021
3 Archive: /tmp/bodek.KRNTq/intro2cs1/ex11/tsviel/final/submission
4   inflating: src/ex11.py
5   inflating: src/_MACOSX/._ex11.py
6 10 passed tests out of 10 in test set named 'presubmit'.
7 result_code   presubmit    10    1
8 16 passed tests out of 16 in test set named 'diagnose1'.
9 result_code   diagnose1    16    1
10 16 passed tests out of 16 in test set named 'diagnose2'.
11 result_code   diagnose2    16    1
12 16 passed tests out of 16 in test set named 'diagnose3'.
13 result_code   diagnose3    16    1
14 16 passed tests out of 16 in test set named 'diagnose4'.
15 result_code   diagnose4    16    1
16 4 passed tests out of 4 in test set named 'calcsuccess'.
17 result_code   calcsuccess    4    1
18 4 passed tests out of 4 in test set named 'allillnesses'.
19 result_code   allillnesses    4    1
20 --> BEGIN TEST INFORMATION
21 Test name: pathsillness_11
22 Module tested: ex11
23 Function call: paths_to_illness('take-coat')
24 Expected return value: ([[False, True], [True]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
25 --> END TEST INFORMATION
26 *****
27 ***** There is a problem:
28 ***** The test named 'pathsillness_11' failed.
29 *****
30 Wrong result, input: ['take-coat']:
31 expected: ([[False, True], [True]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
32 actual:    ([], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
33 result_code   pathsillness_11   wrong    1
34 --> BEGIN TEST INFORMATION
35 Test name: pathsillness_12
36 Module tested: ex11
37 Function call: paths_to_illness('no-coat')
38 Expected return value: ([[False, False]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
39 --> END TEST INFORMATION
40 *****
41 ***** There is a problem:
42 ***** The test named 'pathsillness_12' failed.
43 *****
44 Wrong result, input: ['no-coat']:
45 expected: ([[False, False]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
46 actual:    ([], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
47 result_code   pathsillness_12   wrong    1
48 9 passed tests out of 11 in test set named 'pathsillness'.
49 result_code   pathsillness     9    1
50 3 passed tests out of 3 in test set named 'buildtree'.
51 result_code   buildtree        3    1
52 10 passed tests out of 10 in test set named 'optimaltree'.
53 result_code   optimaltree      10    1
54 --> BEGIN TEST INFORMATION
55 Test name: minimize_f0
56 Module tested: ex11
57 Function call: minimize(False)
58 Expected return value: (None, ('b', ('d', None)))
59 --> END TEST INFORMATION
```

```

60 *****
61 ***** There is a problem:
62 ***** The test named 'minimize_f0' failed.
63 *****
64 Wrong result, input: [False]:
65 expected: (None, ('b', ('d', None)))
66 actual: (None, ('a', (('b', ('d', None)), ('b', ('d', None)))))
67 result_code minimize_f0 wrong 1
68 --> BEGIN TEST INFORMATION
69 Test name: minimize_f1
70 Module tested: ex11
71 Function call: minimize(False)
72 Expected return value: (None, ('b', ('d', 'e')))
73 --> END TEST INFORMATION
74 *****
75 ***** There is a problem:
76 ***** The test named 'minimize_f1' failed.
77 *****
78 Wrong result, input: [False]:
79 expected: (None, ('b', ('d', 'e')))
80 actual: (None, ('a', (('b', ('d', 'e')), ('b', ('d', 'e')))))
81 result_code minimize_f1 wrong 1
82 --> BEGIN TEST INFORMATION
83 Test name: minimize_f4
84 Module tested: ex11
85 Function call: minimize(False)
86 Expected return value: (None, ('a', (('b', ('d', None)), None)))
87 --> END TEST INFORMATION
88 *****
89 ***** There is a problem:
90 ***** The test named 'minimize_f4' failed.
91 *****
92 Wrong result, input: [False]:
93 expected: (None, ('a', (('b', ('d', None)), None)))
94 actual: (None, ('a', (('b', ('d', None)), ('b', (None, None)))))
95 result_code minimize_f4 wrong 1
96 --> BEGIN TEST INFORMATION
97 Test name: minimize_f5
98 Module tested: ex11
99 Function call: minimize(False)
100 Expected return value: (None, ('a', (('b', ('d', None)), 'd')))
101 --> END TEST INFORMATION
102 *****
103 ***** There is a problem:
104 ***** The test named 'minimize_f5' failed.
105 *****
106 Wrong result, input: [False]:
107 expected: (None, ('a', (('b', ('d', None)), 'd')))
108 actual: (None, ('a', (('b', ('d', None)), ('b', ('d', 'd')))))
109 result_code minimize_f5 wrong 1
110 --> BEGIN TEST INFORMATION
111 Test name: minimize_t0
112 Module tested: ex11
113 Function call: minimize(True)
114 Expected return value: (None, 'd')
115 --> END TEST INFORMATION
116 *****
117 ***** There is a problem:
118 ***** The test named 'minimize_t0' failed.
119 *****
120 return value could not be expanded
121 result_code minimize_t0 reconvertree 1
122 --> BEGIN TEST INFORMATION
123 Test name: minimize_t1
124 Module tested: ex11
125 Function call: minimize(True)
126 Expected return value: (None, ('b', ('d', 'e')))
127 --> END TEST INFORMATION

```

```

128 *****
129 *****          There is a problem:
130 *****          The test named 'minimize_t1' failed.
131 *****
132 Wrong result, input: [True]:
133 expected: (None, ('b', ('d', 'e')))
134 actual:   (None, ('a', (('b', ('d', 'e')), ('b', ('d', 'e')))))
135 result_code minimize_t1 wrong 1
136 --> BEGIN TEST INFORMATION
137 Test name: minimize_t3
138 Module tested: ex11
139 Function call: minimize(True)
140 Expected return value: (None, 'd')
141 --> END TEST INFORMATION
142 *****
143 *****          There is a problem:
144 *****          The test named 'minimize_t3' failed.
145 *****
146 return value could not be expanded
147 result_code minimize_t3 recovertree 1
148 --> BEGIN TEST INFORMATION
149 Test name: minimize_t4
150 Module tested: ex11
151 Function call: minimize(True)
152 Expected return value: (None, 'd')
153 --> END TEST INFORMATION
154 *****
155 *****          There is a problem:
156 *****          The test named 'minimize_t4' failed.
157 *****
158 return value could not be expanded
159 result_code minimize_t4 recovertree 1
160 --> BEGIN TEST INFORMATION
161 Test name: minimize_t5
162 Module tested: ex11
163 Function call: minimize(True)
164 Expected return value: (None, 'd')
165 --> END TEST INFORMATION
166 *****
167 *****          There is a problem:
168 *****          The test named 'minimize_t5' failed.
169 *****
170 return value could not be expanded
171 result_code minimize_t5 recovertree 1
172 3 passed tests out of 12 in test set named 'minimize'.
173 result_code minimize 3 1
174 TESTING COMPLETED

```

2 ex11.py

```
1 #####
2 # FILE : ex11.py
3 # WRITER : TSVIEL ZAIKMAN , tsviel , 208241133
4 # EXERCISE : intro2cs1 ex11 2021
5 # DESCRIPTION: Traverse on medical records tree Graphs
6 #####
7 from collections import Counter
8 from itertools import combinations
9
10 EMPTY = 0 # LENGTH IS 0
11 MAX = 0 # MAXIMAL VALUE
12 TYPE_ERROR_MESSAGE = "Either your symptom or record is invalid" \
13     "- Raised TypeError"
14 VALUE_ERROR_MESSAGE = "A Value Error has been raised"
15
16
17 class Node:
18     def __init__(self, data, positive_child=None, negative_child=None):
19         self.data = data
20         self.positive_child = positive_child
21         self.negative_child = negative_child
22
23
24 class Record:
25     def __init__(self, illness, symptoms):
26         self.illness = illness
27         self.symptoms = symptoms
28
29
30 def parse_data(filepath):
31     with open(filepath) as data_file:
32         records = []
33         for line in data_file:
34             words = line.strip().split()
35             records.append(Record(words[0], words[1:]))
36         return records
37
38
39 def is_leaf(tree_node):
40     """Return True if node is leaf, false if not None"""
41     if tree_node.positive_child is None and tree_node.negative_child is None:
42         return True
43     return False
44
45
46 def empty(tree_node):
47     """Return True if node is None, false if not None"""
48     if tree_node.data is None:
49         return True
50     else:
51         return False
52
53
54 class Diagnoser:
55     def __init__(self, root: Node):
56         self.root = root
57
58     def diagnose(self, symptoms):
59         return self._diagnose_helper(symptoms, self.root)
```

```

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

```

```

128     """Recursive Helper of path to illness """
129     if is_leaf(node):
130         # Stop condition - We hit leaf
131         if illness is node.data: # If we reach our illness
132             return [[]]
133         return [] # If its not our illness
134         # Run recursion on the left branch
135
136     negative_path = self.__paths_to_illness(node.negative_child, illness)
137     # Run recursion on the right branch
138     positive_path = self.__paths_to_illness(node.positive_child, illness)
139     paths = [] # Create new list for routes
140     for route in negative_path: # append left direction
141         paths.append([False] + route)
142     for route in positive_path: # Append right direction
143         paths.append([True] + route)
144     return paths # Return all routes of tree
145
146 def remove_half_nodes(self, tree_node):
147     """Remove all half nodes from tree"""
148     if tree_node is None:
149         return None
150     if tree_node.positive_child is not None \
151         and tree_node.negative_child is not None:
152         tree_node = self.__remove_child_duplications(tree_node)
153     tree_node.negative_child = self.remove_half_nodes(
154         tree_node.negative_child) # Recur to left tree
155     tree_node.positive_child = self.remove_half_nodes(
156         tree_node.positive_child) # Recur to right tree
157     if is_leaf(tree_node): # We hit leaf
158         return tree_node
159     if tree_node.negative_child is None:
160         next_root = tree_node.positive_child
161         prev_root, tree_node = tree_node, None
162         del prev_root
163         return next_root
164     if tree_node.positive_child is None:
165         next_root = tree_node.negative_child
166         prev_root, tree_node = tree_node, None
167         del prev_root
168         return next_root
169     return tree_node
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
178     return node
179
180 def __delete_leaves(self, tree_node, val):
181     """
182     :param tree_node: a given tree object
183     :param val: a value we wish to remove
184     :return: remove the leaves with the given value
185     """
186     if tree_node is None:
187         return
188     tree_node.negative_child = self.__delete_leaves(
189         tree_node.negative_child, val)
190     tree_node.positive_child = self.__delete_leaves(
191         tree_node.positive_child, val)
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):
198         """The function minimize the tree and removes unnesseary path"""
199         if remove_empty:
200             self.__delete_leaves(self.root, None)
201             self.remove_half_nodes(self.root)
202             return
203
204
205     def _build_tree_helper(records):
206         """helper of build_tree for counting"""
207         if len(records) is EMPTY:
208             return None
209         illnesses = []
210         for i in range(len(records)):
211             illnesses.append(records[i].illness)
212         counts = Counter(illnesses)
213         # return sorted list by prevalence
214         return sorted(counts, key=counts.get, reverse=True)[0]
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
222         :return: Diagnoser Object for the tree we build
223         """
224         if len(symptoms) == EMPTY:
225             return Node(_build_tree_helper(records))
226
227         first_symptom, rest_symptoms = symptoms[0], symptoms[1:]
228
229         positive_path = _build_tree([record for record in records if
230                                     first_symptom
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
238         return final_tree
239
240
241     def build_tree(records, symptoms):
242         """
243         :param records: a list of records(objects)
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]
249             records_are_record = [type(record) == Record for record in records]
250             if not all(symptoms_are_strings) and all(records_are_record):
251                 # Logic of statement is based on de-morgan law
252                 raise TypeError
253             return Diagnoser(_build_tree(records, symptoms))
254         except TypeError:
255             return TYPE_ERROR_MESSAGE
256
257
258     def _optimal_tree(records, symptoms, depth):
259         """
260         Returns the optimal tree for different subsets of symptoms with size of
261         depth. The function iterates on all of the subsets of symptoms in size
262         and depth for each time we create a tree Node.
263         The function gets the rates for success, update the maximal value(tuple)

```



```

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

```