Parameterized Complexity of Small Decision Tree Learning.

Ordyniak, S., & Szeider, S. (2021).
Proceedings of the AAAI Conference on Artificial Intelligence, 35 (7), 6454-6462.

let’s break it down!

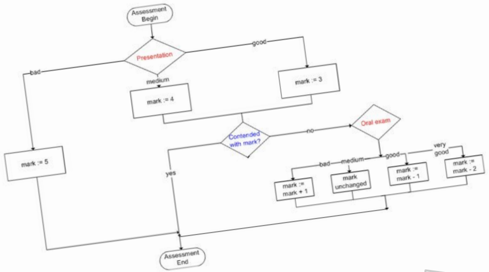
Parameterized Complexity of Small Decision Tree Learning.

Ordyniak, S., & Szeider, S. (2021).
Proceedings of the AAAI Conference on Artificial Intelligence, 35 (7), 6454-6462.

Parameterized Complexity of Small Decision Tree Learning.

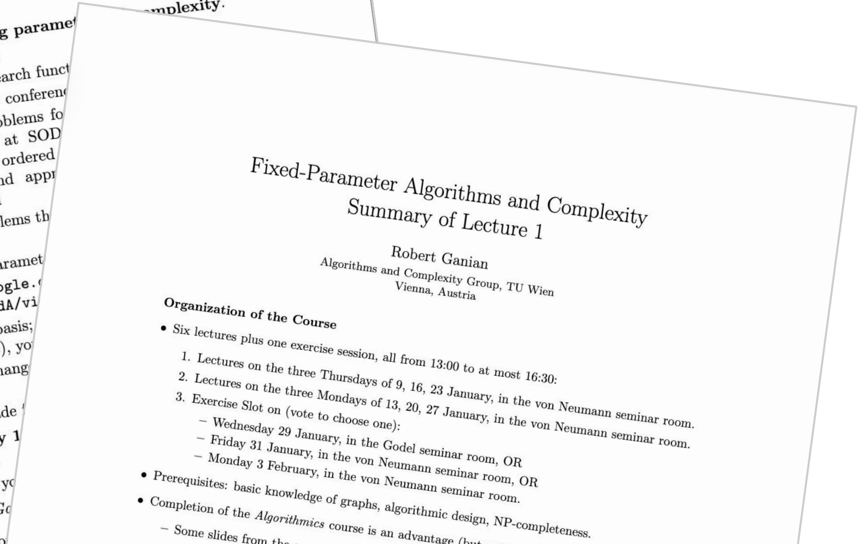
Ordyniak, S., & Szeider, S. (2021).
Proceedings of the AAAI Conference on Artificial Intelligence, 35 (7), 6454-6462.

grading schema



mplexity.

- More information about the presentations:
  - ∗ Each student must select a published paper **featuring paramet**
  - ∗ **Finding a paper is part of the task**. Some hints:
    - · Use DBLP (https://dblp.uni-trier.de/) and its search funct
    - · The paper can either be a journal publication or conferenc
    - · If you want a paper covering fundamental problems fo ory, check out recent parameterized papers at SOD MFCS, ISAAC, IPEC (these venues are ordered to "less competitive", in a very rough and appr https://dblp.org/db/conf/icalp/icalp2022.html
    - · If you want a paper covering fundamental problems th try papers at AAAI or IJCAI
    - · Tip: look for titles with "parameterized", "paramet
  - ∗ Enter your proposed paper via `https://docs.google.c` `WGys2PdkRumiKB8fDjLeyA6MXsA-OfRckhrddlYDdA/vi`
  - ∗ Papers are assigned on a first-come-first-serve basis; available (or if it is not well-suited for the course), yo
  - ∗ You can send me any questions, comments or chang `com` or `rganian@ac.tuwien.ac.at`)
  - ∗ Once you select a paper, you will receive a grade
- Paper selection must be completed by **Wednesday 1** to complete it by **Tuesday 14 January, 23:59**).
- It is not required to understand all the details in yo
  choose any paper co-authored by Robert G

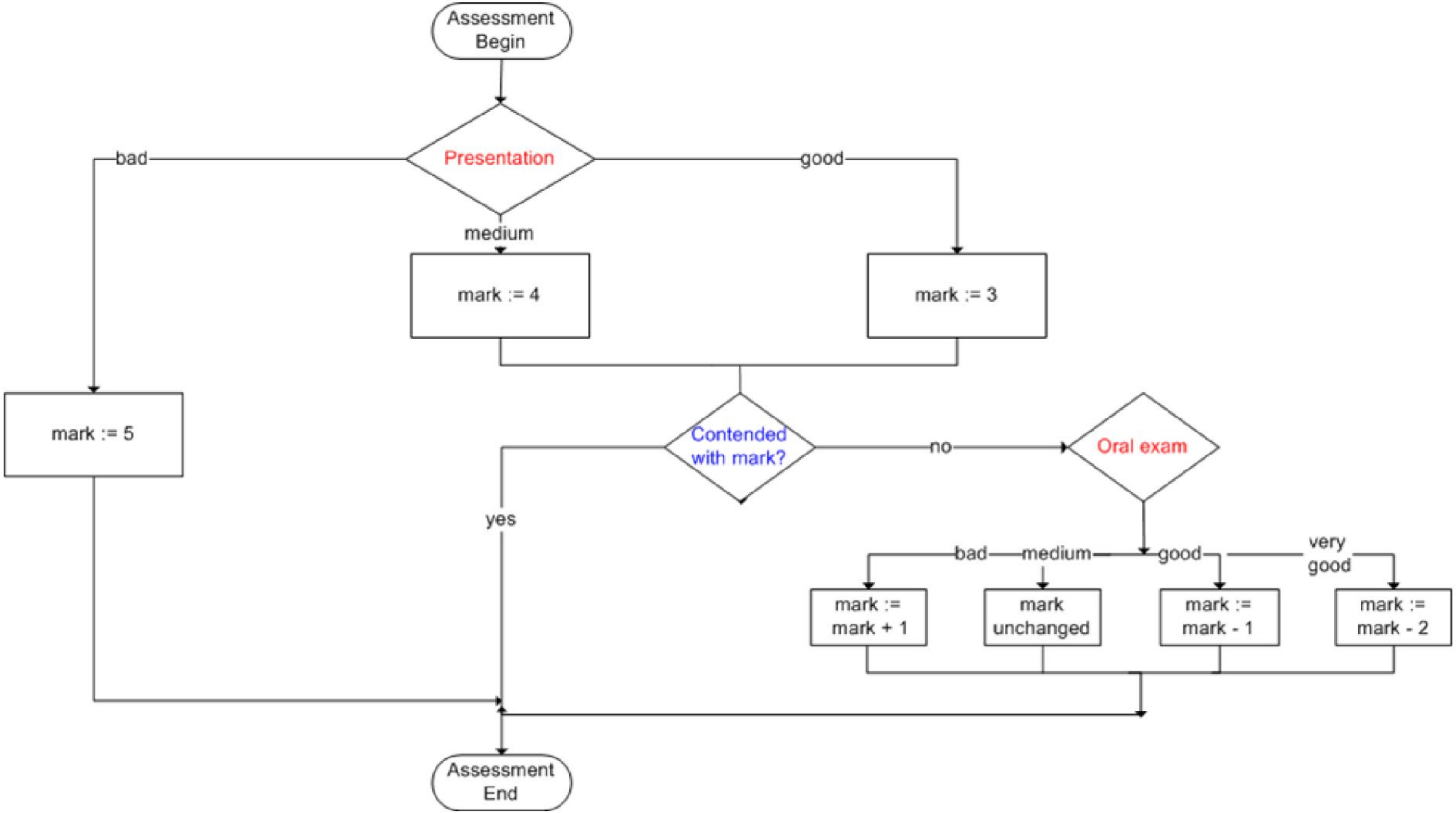


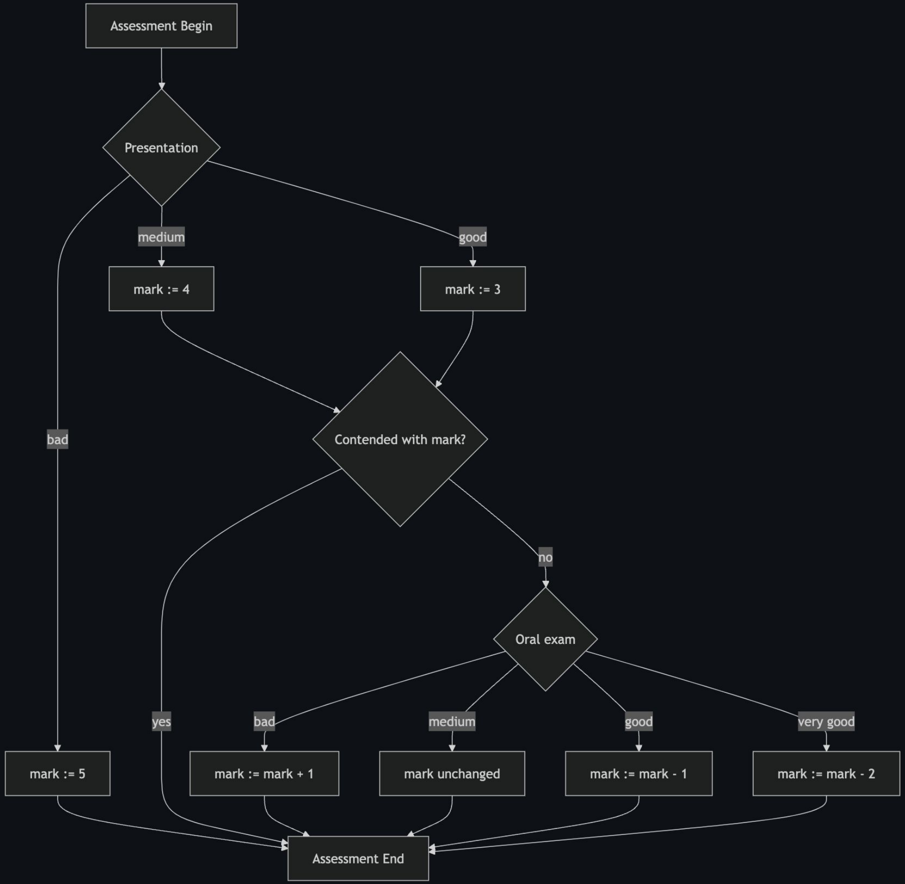
Fixed-Parameter Algorithms and Complexity
Summary of Lecture 1

Robert Ganian
Algorithms and Complexity Group, TU Wien
Vienna, Austria

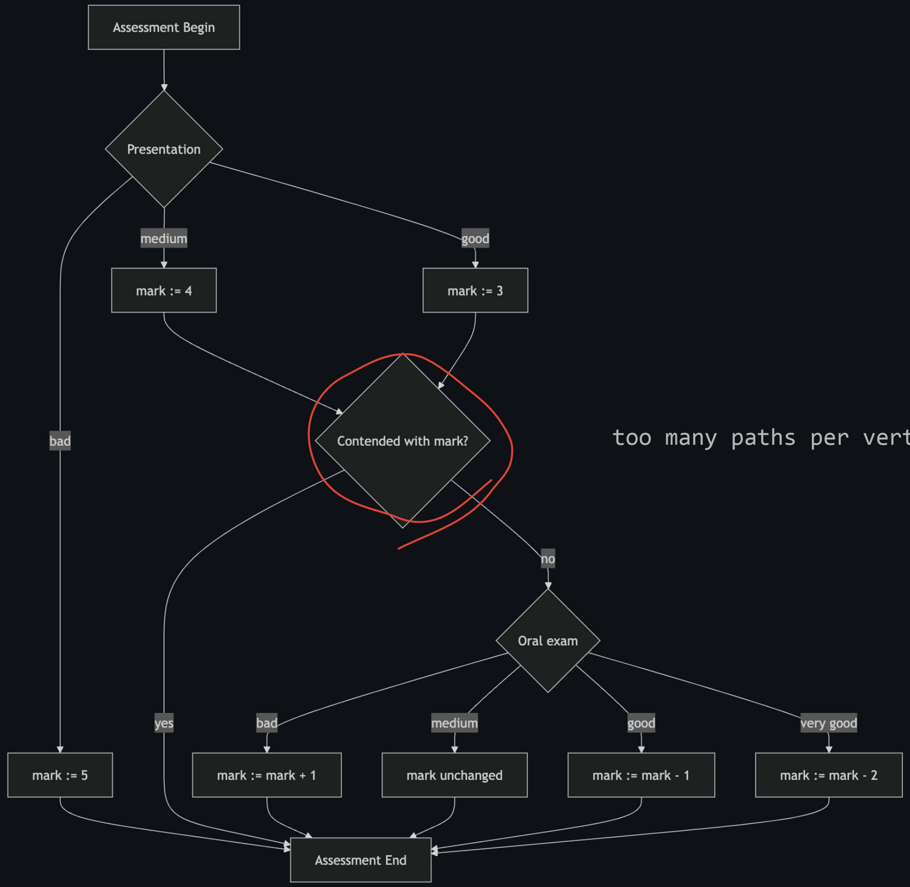
Organization of the Course

- Six lectures plus one exercise session, all from 13:00 to at most 16:30:
  1. Lectures on the three Thursdays of 9, 16, 23 January, in the von Neumann seminar room.
  2. Lectures on the three Mondays of 13, 20, 27 January, in the von Neumann seminar room.
  3. Exercise Slot on (vote to choose one):
     - Wednesday 29 January, in the Godel seminar room, OR
     - Friday 31 January, in the von Neumann seminar room, OR
     - Monday 3 February, in the von Neumann seminar room.
- Prerequisites: basic knowledge of graphs, algorithmic design, NP-completeness.
- Completion of the *Algorithmics* course is an advantage (but
  - Some slides from the


Assessment Begin
Presentation
bad
medium
good
mark := 4
mark := 3
mark := 5
Contended with mark?
yes
no
Oral exam
bad
medium
good
very good
mark := mark + 1
mark unchanged
mark := mark - 1
mark := mark - 2
Assessment End

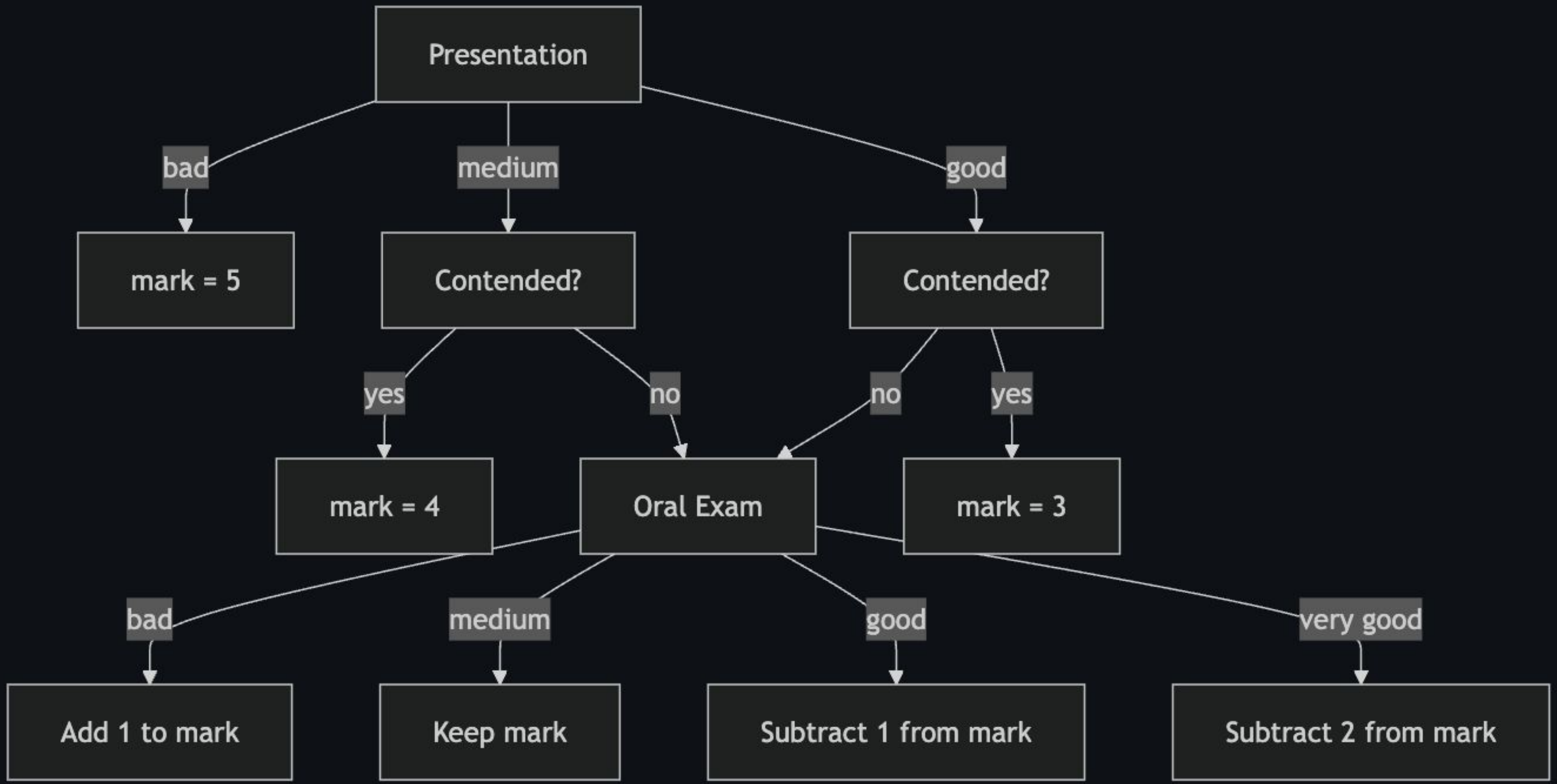


code: https://github.com/sueszli/optimal-tree-solver/blob/main/docs/grading.md



too many paths per vertex

code: https://github.com/sueszli/optimal-tree-solver/blob/main/docs/grading.md



Presentation

bad

medium

good

mark = 5

Contended?

Contended?

yes

no

no

yes

mark = 4

Oral Exam

mark = 3

bad

medium

good

very good

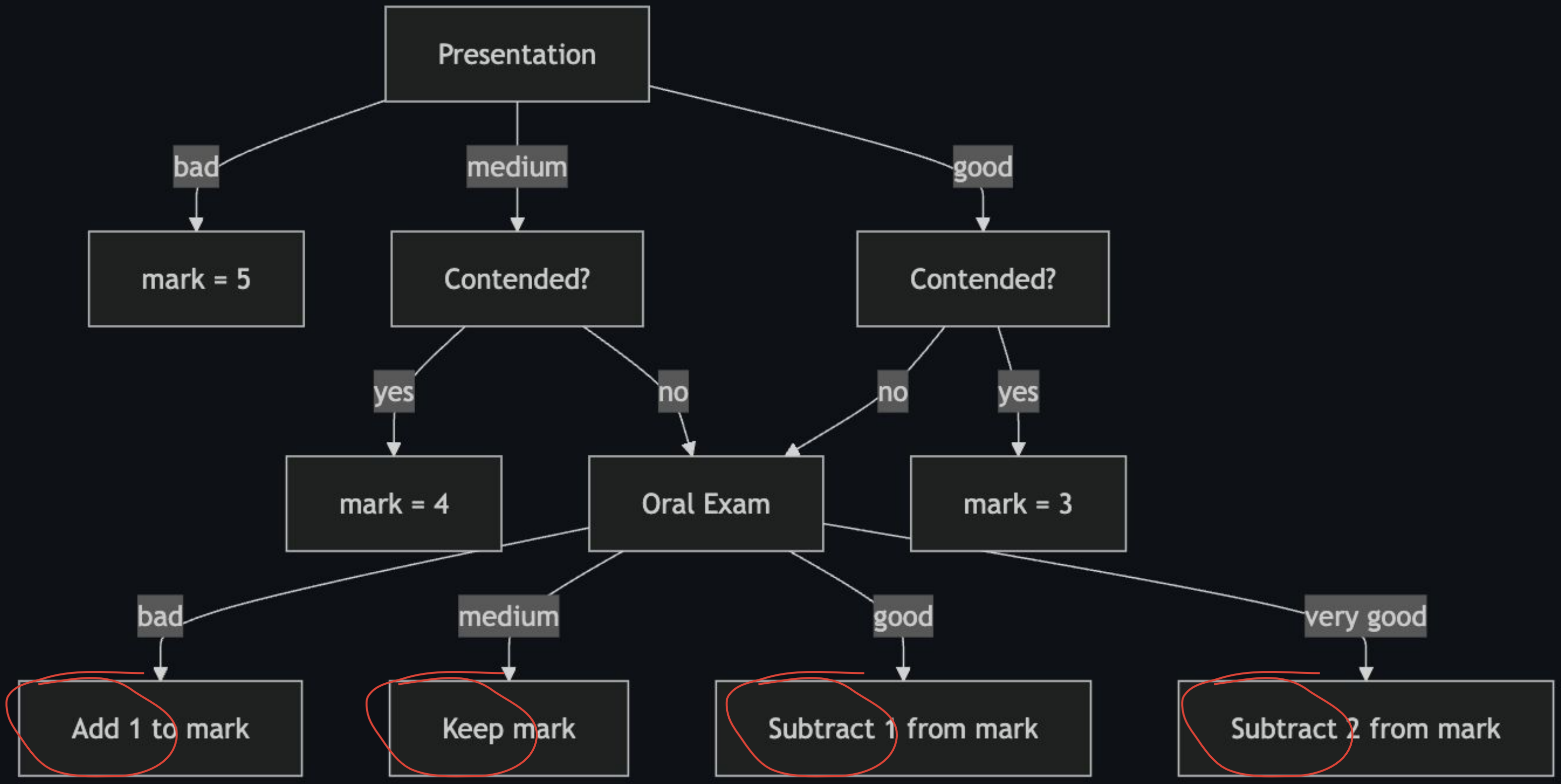
Add 1 to mark

Keep mark

Subtract 1 from mark

Subtract 2 from mark

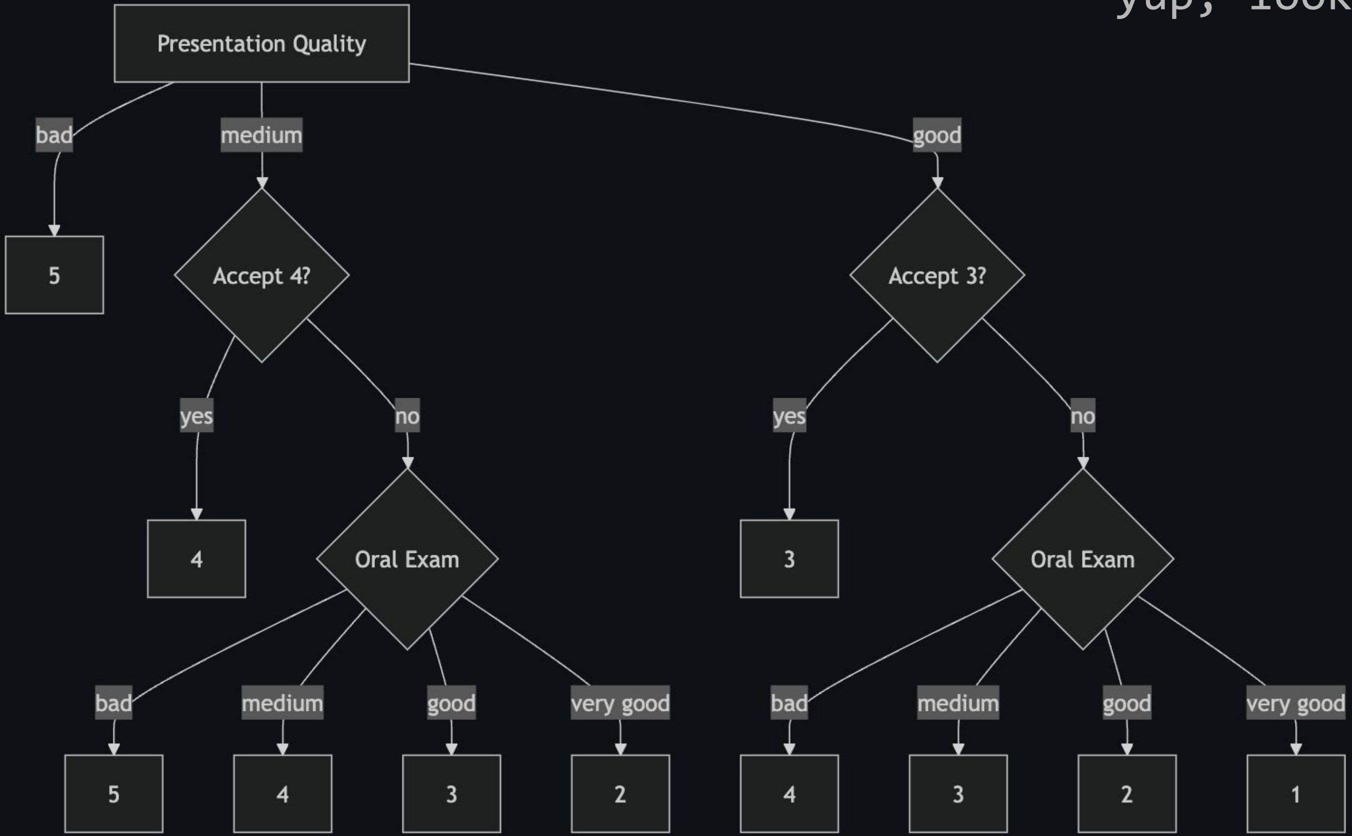
code: https://github.com/sueszli/optimal-tree-solver/blob/main/docs/grading.md



labels need arithmetic

code: https://github.com/sueszli/optimal-tree-solver/blob/main/docs/grading.md

yup, looks right!



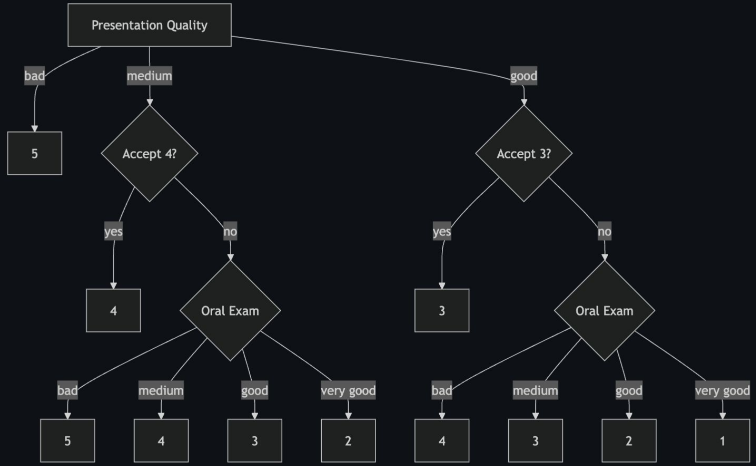
code: https://github.com/sueszli/optimal-tree-solver/blob/main/docs/grading.md

data schema: [presentation grade, accept grade 3/4, oral grade]

labels: [1, 2, 3, 4, 5]
but *boolean* in paper

split: we try to find a perfect decision boundary, full acc.
no entropy metrics, just brute forcing combinations.

tree depth: 3 (longest root-to-leaf path)
tree size: 8 (count of non-leaf nodes)



we prefer decision trees to be small in depth/size:

- easier to interpret
- use fewer, more robust features

problems:

- minimum decision tree size (DTS)
- minimum decision tree depth (DTD)

decision problems.

but our algorithms also provide the solution.

Parameterized Complexity of Small Decision Tree Learning.

Ordyniak, S., & Szeider, S. (2021).
Proceedings of the AAAI Conference on Artificial Intelligence, 35 (7), 6454-6462.

traditional complexity theory:

finding minimal (size/depth) decision trees is NP-hard (Hyafil & Rivast, 1976).

- no known polynomial-time algorithm exists for all instances unless P=NP.
- finding solution is computationally intractable for large inputs.

parameterized complexity theory:

contribution of this paper.
problem is fixed-parameter tractable.

= problem is feasible for small problem parameter values, despite NP-hardness.

= runtime is polynomial in the input size, but exponential in some problem parameters.

traditional complexity theory:

finding minimal (size/depth) decision trees is NP-hard (Hyafil & Rivast, 1976).

- no known polynomial-time algorithm exists for all instances unless P=NP.
- finding solution is computationally intractable for large inputs.

parameterized complexity theory:

contribution of this paper.
problem is fixed-parameter tractable.

= problem is feasible for small problem parameter values, despite NP-hardness.

= runtime is polynomial in the input size, but exponential in some problem parameters.

how do you find these?

problem becomes *fixed-parameter tractable* when parametrized by:

- solution size/depth
- maximum domain size (max value range of any feature)
- maximum hamming distance (max num of features that differ between any 2 examples.)

might be redundant

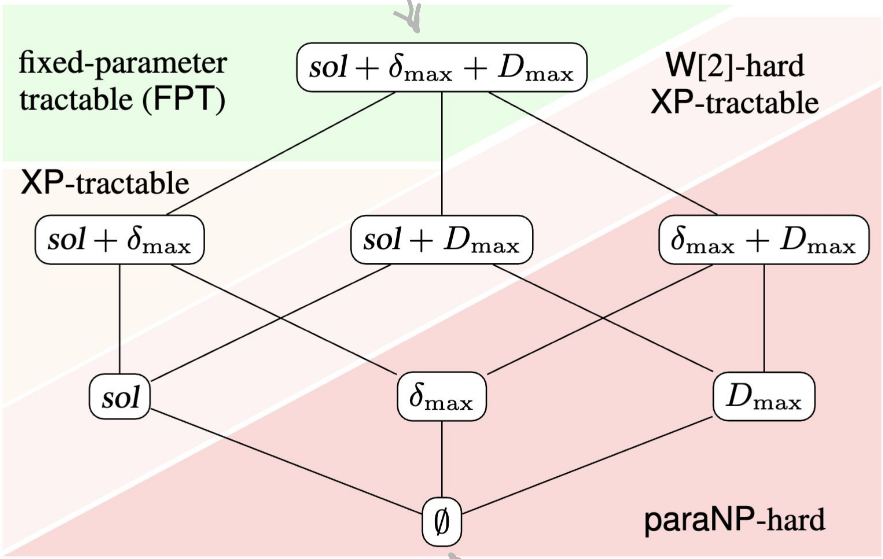
$$O\left(f(sol, D_{\max}, \delta_{\max}) \cdot \mathrm{poly}(n)\right)$$

some exponential function

polynomial in the input size

how did the authors find the right problem parameters?

to here?



complexity landscape

how to get from here

how did the authors find the right problem parameters?

deconstruction of intractability:

find naturally occurring problem parameters that strongly influence the problem’s runtime (practical solvability).

- mostly trial and error.
- reductions to other problems.
- analyzing practical usage.

how did the authors find the right problem parameters?

solution size:
- from problem definition

max domain size:
- determines search space for decision boundaries

max hamming dist:
- is the size of largest set in “*hitting set*” reduction
- very small in practice

The optimal decision tree algorithm.

way harder

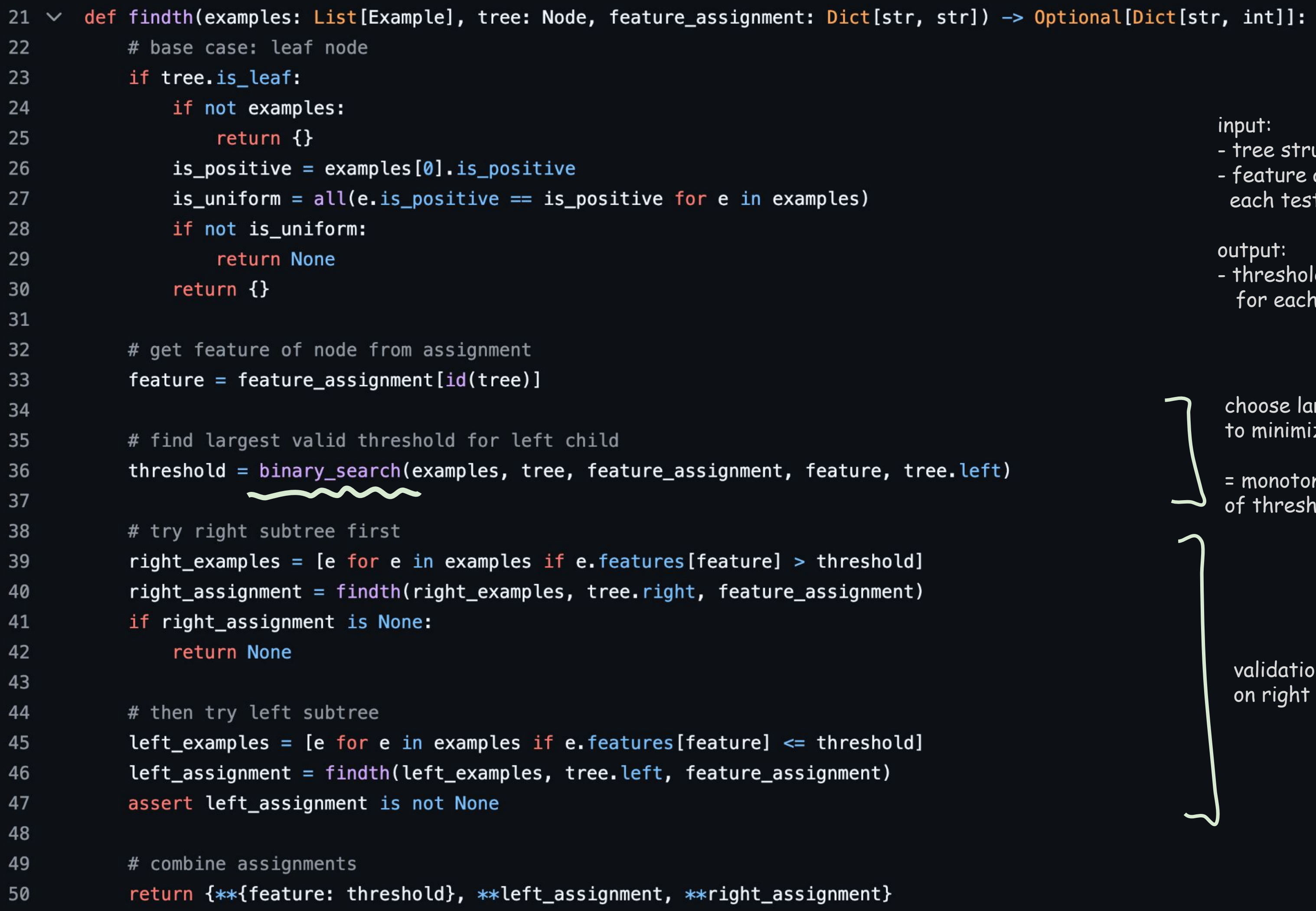
1) feature selection

2) tree construction

code: https://github.com/sueszli/optimal-tree-solver/blob/main/src/mindt.py

2) tree construction

- input: examples, tree size + set of features
- output: minimal tree (or null)



```python
def findth(examples: List[Example], tree: Node, feature_assignment: Dict[str, str]) -> Optional[Dict[str, int]]:
    # base case: leaf node
    if tree.is_leaf:
        if not examples:
            return {}
        is_positive = examples[0].is_positive
        is_uniform = all(e.is_positive == is_positive for e in examples)
        if not is_uniform:
            return None
        return {}

    # get feature of node from assignment
    feature = feature_assignment[id(tree)]

    # find largest valid threshold for left child
    threshold = binary_search(examples, tree, feature_assignment, feature, tree.left)

    # try right subtree first
    right_examples = [e for e in examples if e.features[feature] > threshold]
    right_assignment = findth(right_examples, tree.right, feature_assignment)
    if right_assignment is None:
        return None

    # then try left subtree
    left_examples = [e for e in examples if e.features[feature] <= threshold]
    left_assignment = findth(left_examples, tree.left, feature_assignment)
    assert left_assignment is not None

    # combine assignments
    return {**{feature: threshold}, **left_assignment, **right_assignment}
```

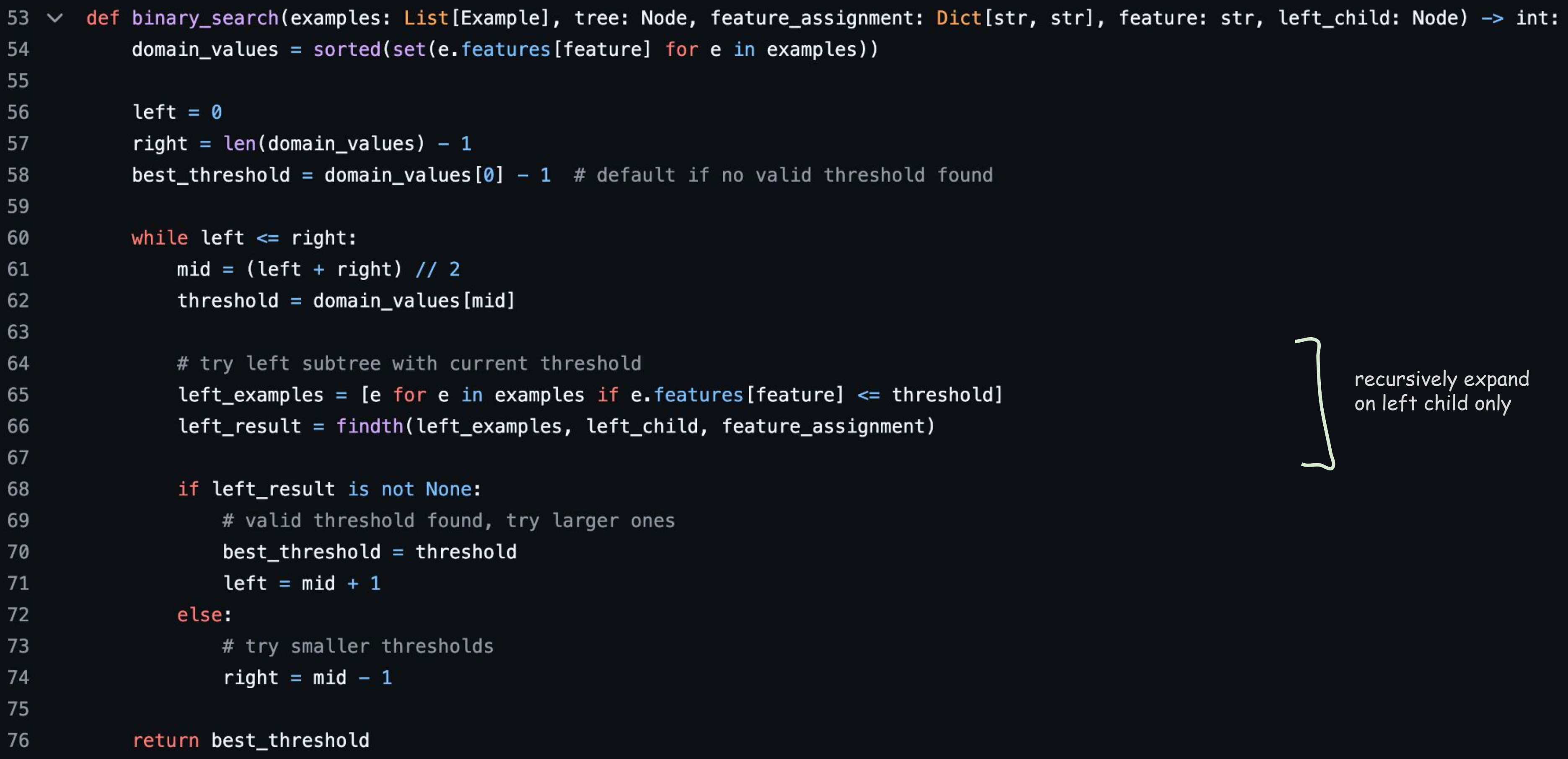
input:
- tree structure
- feature assignment for
  each test node

output:
- threshold assignment
  for each test node

choose largest threshold
to minimize entropy

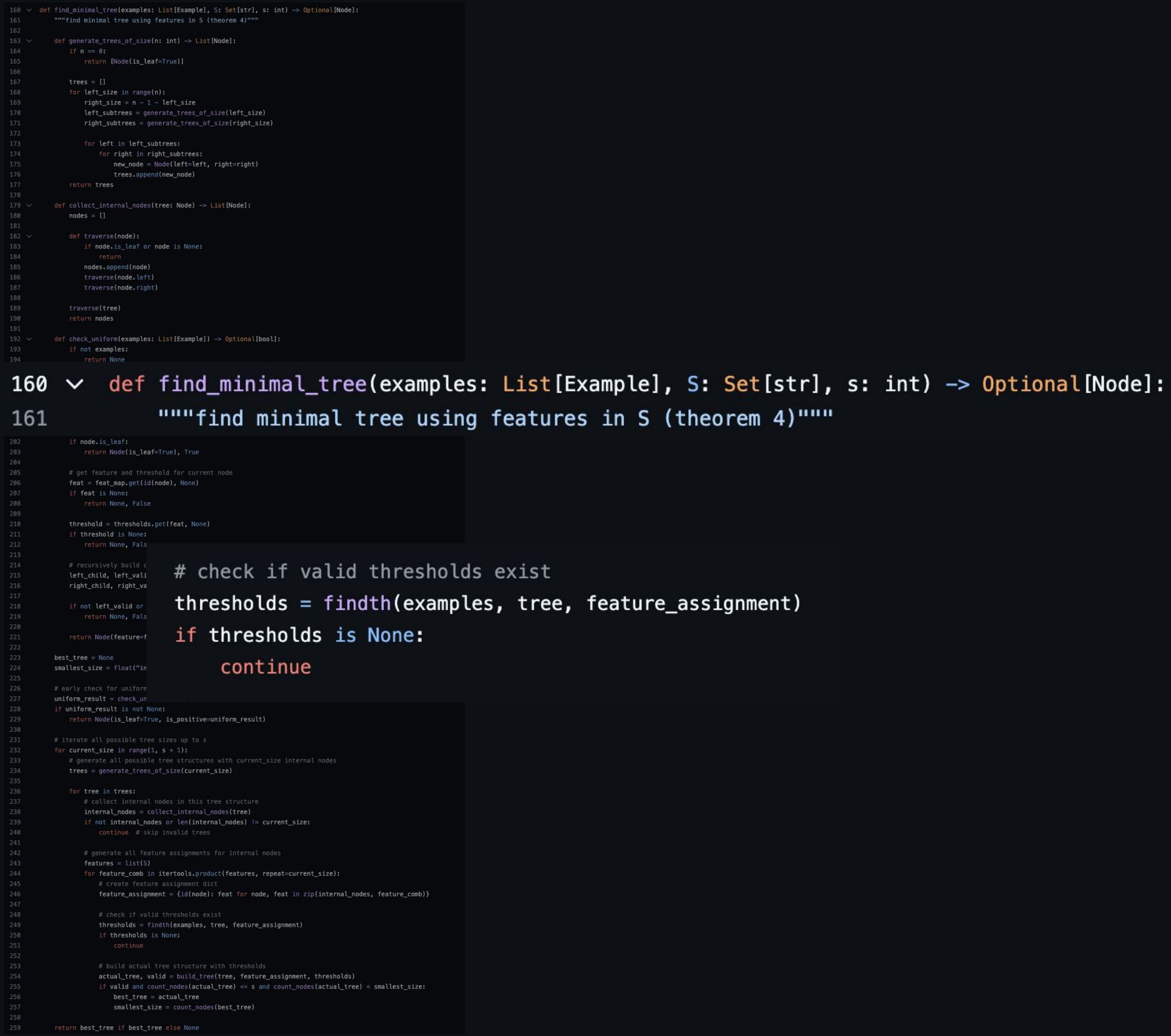
= monotonicity property
of thresholds

validation
on right child



```python
def binary_search(examples: List[Example], tree: Node, feature_assignment: Dict[str, str], feature: str, left_child: Node) -> int:
    domain_values = sorted(set(e.features[feature] for e in examples))

    left = 0
    right = len(domain_values) - 1
    best_threshold = domain_values[0] - 1  # default if no valid threshold found

    while left <= right:
        mid = (left + right) // 2
        threshold = domain_values[mid]

        # try left subtree with current threshold
        left_examples = [e for e in examples if e.features[feature] <= threshold]
        left_result = findth(left_examples, left_child, feature_assignment)

        if left_result is not None:
            # valid threshold found, try larger ones
            best_threshold = threshold
            left = mid + 1
        else:
            # try smaller thresholds
            right = mid - 1

    return best_threshold
```

recursively expand
on left child only



```python
def find_minimal_tree(examples: List[Example], S: Set[str], s: int) -> Optional[Node]:
    """find minimal tree using features in S (theorem 4)"""

    def generate_trees_of_size(n: int) -> List[Node]:
        if n == 0:
            return [Node(is_leaf=True)]

        trees = []
        for left_size in range(n):
            right_size = n - 1 - left_size
            left_subtrees = generate_trees_of_size(left_size)
            right_subtrees = generate_trees_of_size(right_size)

            for left in left_subtrees:
                for right in right_subtrees:
                    new_node = Node(left=left, right=right)
                    trees.append(new_node)
        return trees

    def collect_internal_nodes(tree: Node) -> List[Node]:
        nodes = []

        def traverse(node):
            if node.is_leaf or node is None:
                return
            nodes.append(node)
            traverse(node.left)
            traverse(node.right)

        traverse(tree)
        return nodes

    def check_uniform(examples: List[Example]) -> Optional[bool]:
        if not examples:
            return None
```

```python
def find_minimal_tree(examples: List[Example], S: Set[str], s: int) -> Optional[Node]:
    """find minimal tree using features in S (theorem 4)"""
```

```python
        if node.is_leaf:
            return Node(is_leaf=True), True

        # get feature and threshold for current node
        feat = feat_map.get(id(node), None)
        if feat is None:
            return None, False

        threshold = thresholds.get(feat, None)
        if threshold is None:
            return None, Fals

        # recursively build c
        left_child, left_vali
        right_child, right_va

        if not left_valid or
            return None, Fals

        return Node(feature=f

    best_tree = None
    smallest_size = float("in

    # early check for uniform
    uniform_result = check_un
    if uniform_result is not None:
        return Node(is_leaf=True, is_positive=uniform_result)

    # iterate all possible tree sizes up to s
    for current_size in range(1, s + 1):
        # generate all possible tree structures with current_size internal nodes
        trees = generate_trees_of_size(current_size)

        for tree in trees:
            # collect internal nodes in this tree structure
            internal_nodes = collect_internal_nodes(tree)
            if not internal_nodes or len(internal_nodes) != current_size:
                continue  # skip invalid trees

            # generate all feature assignments for internal nodes
            features = list(S)
            for feature_comb in itertools.product(features, repeat=current_size):
                # create feature assignment dict
                feature_assignment = {id(node): feat for node, feat in zip(internal_nodes, feature_comb)}

                # check if valid thresholds exist
                thresholds = findth(examples, tree, feature_assignment)
                if thresholds is None:
                    continue

                # build actual tree structure with thresholds
                actual_tree, valid = build_tree(tree, feature_assignment, thresholds)
                if valid and count_nodes(actual_tree) <= s and count_nodes(actual_tree) < smallest_size:
                    best_tree = actual_tree
                    smallest_size = count_nodes(best_tree)

    return best_tree if best_tree else None
```

```python
# check if valid thresholds exist
thresholds = findth(examples, tree, feature_assignment)
if thresholds is None:
    continue
```

enumerate all possible:

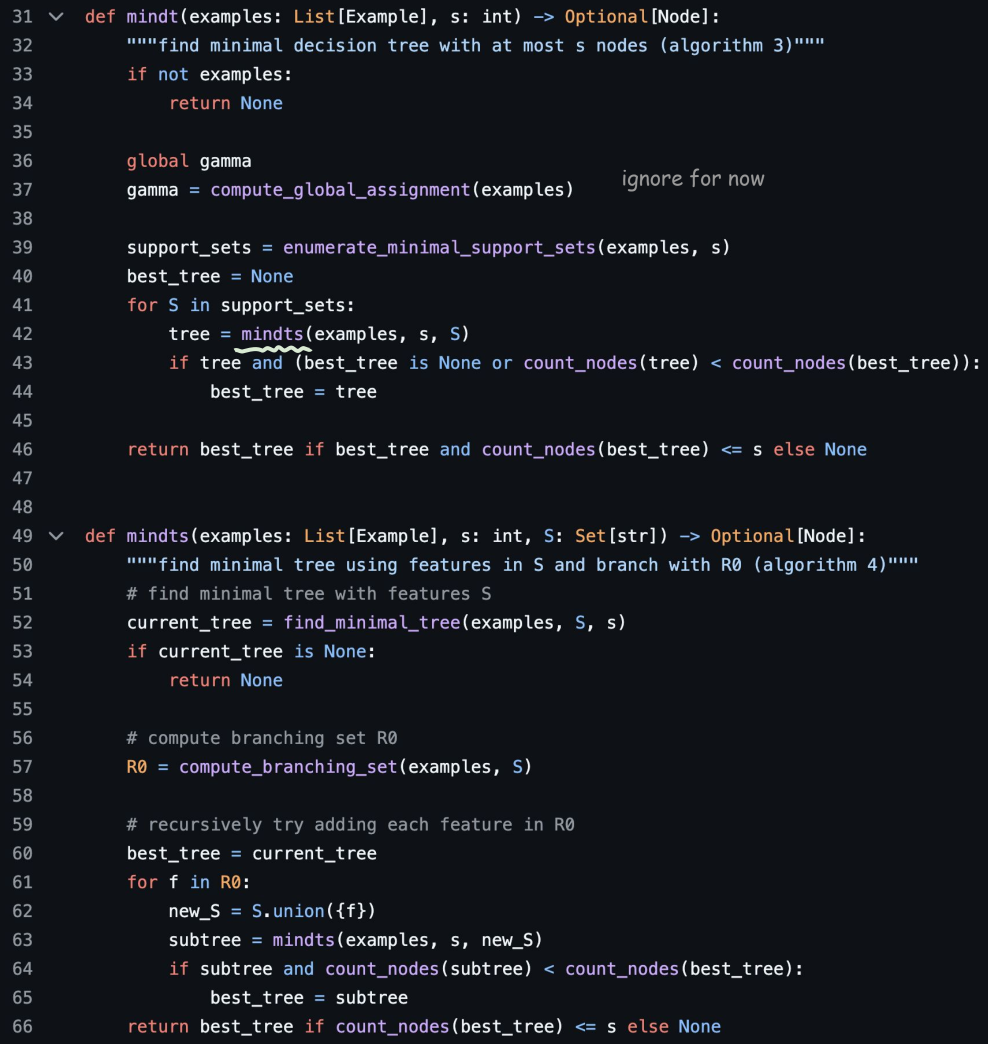
- tree structures
- node feature assignments

given a set of features.

validate each combination.

1) feature selection

- input: examples, tree size
- output: minimal tree (or null)



```python
def mindt(examples: List[Example], s: int) -> Optional[Node]:
    """find minimal decision tree with at most s nodes (algorithm 3)"""
    if not examples:
        return None

    global gamma
    gamma = compute_global_assignment(examples)

    support_sets = enumerate_minimal_support_sets(examples, s)
    best_tree = None
    for S in support_sets:
        tree = mindts(examples, s, S)
        if tree and (best_tree is None or count_nodes(tree) < count_nodes(best_tree)):
            best_tree = tree

    return best_tree if best_tree and count_nodes(best_tree) <= s else None


def mindts(examples: List[Example], s: int, S: Set[str]) -> Optional[Node]:
    """find minimal tree using features in S and branch with R0 (algorithm 4)"""
    # find minimal tree with features S
    current_tree = find_minimal_tree(examples, S, s)
    if current_tree is None:
        return None

    # compute branching set R0
    R0 = compute_branching_set(examples, S)

    # recursively try adding each feature in R0
    best_tree = current_tree
    for f in R0:
        new_S = S.union({f})
        subtree = mindts(examples, s, new_S)
        if subtree and count_nodes(subtree) < count_nodes(best_tree):
            best_tree = subtree
    return best_tree if count_nodes(best_tree) <= s else None
```

ignore for now

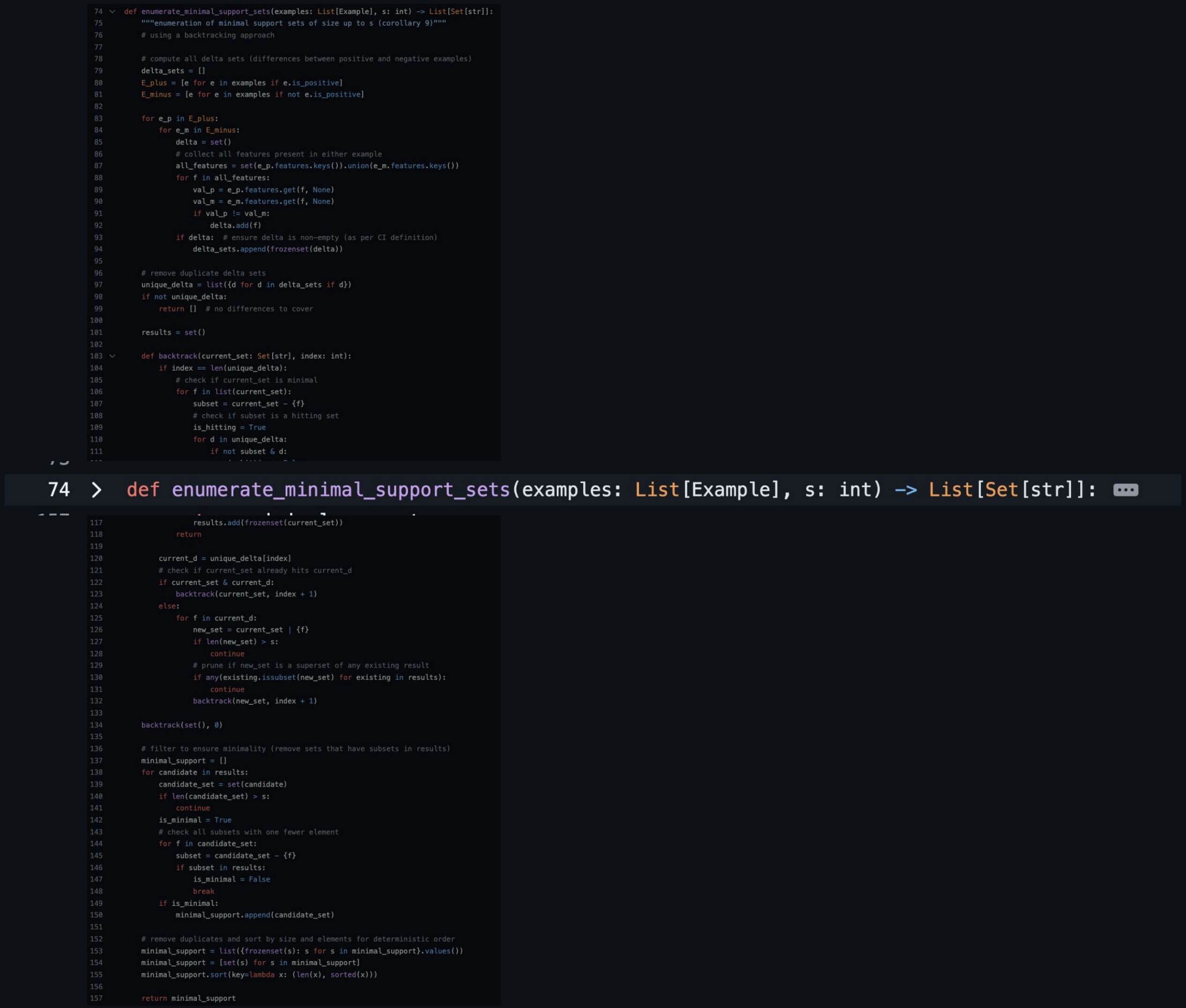
all possible feature combinations
that can distinguish *most* examples

check if tree exists (see first stage)

add "additional features" that help distinguish
all examples.

find the "greatest common denominator"
among all "additional features",

called "branching set / R0"

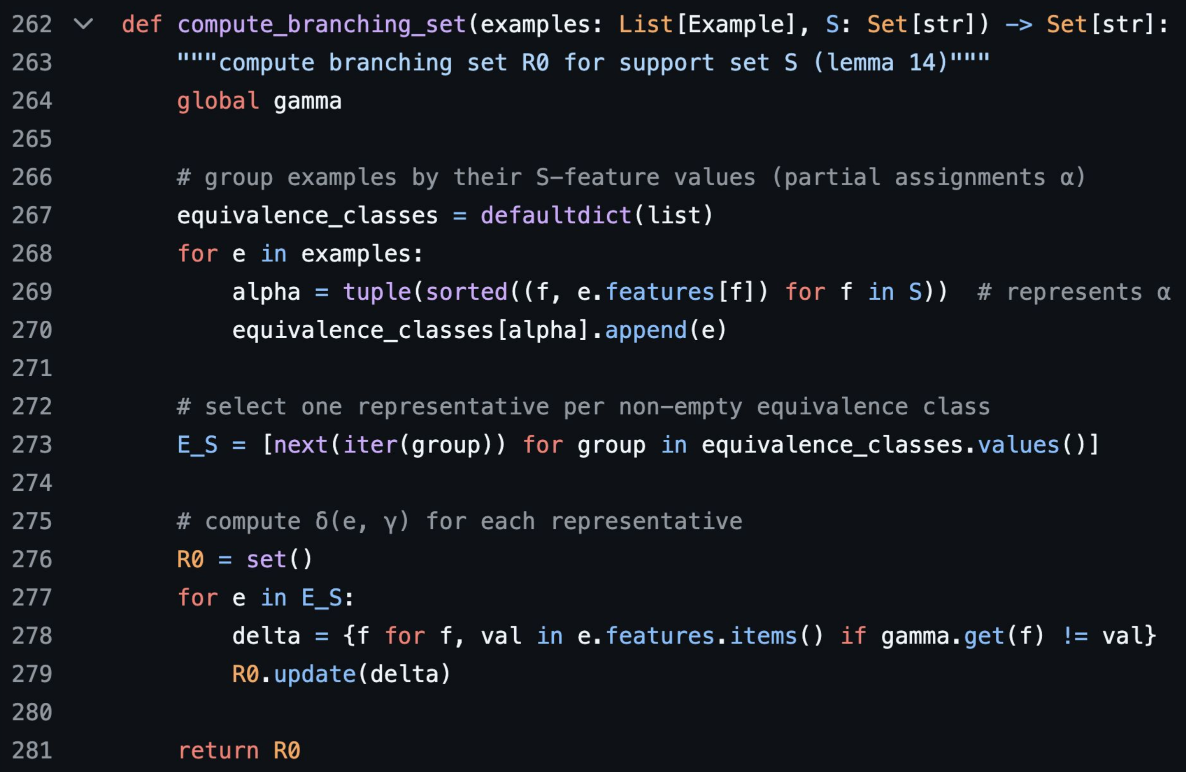


```python
def enumerate_minimal_support_sets(examples: List[Example], s: int) -> List[Set[str]]:
    """enumeration of minimal support sets of size up to s (corollary 9)"""
    # using a backtracking approach

    # compute all delta sets (differences between positive and negative examples)
    delta_sets = []
    E_plus = [e for e in examples if e.is_positive]
    E_minus = [e for e in examples if not e.is_positive]

    for e_p in E_plus:
        for e_m in E_minus:
            delta = set()
            # collect all features present in either example
            all_features = set(e_p.features.keys()).union(e_m.features.keys())
            for f in all_features:
                val_p = e_p.features.get(f, None)
                val_m = e_m.features.get(f, None)
                if val_p != val_m:
                    delta.add(f)
            if delta:  # ensure delta is non-empty (as per CI definition)
                delta_sets.append(frozenset(delta))

    # remove duplicate delta sets
    unique_delta = list({d for d in delta_sets if d})
    if not unique_delta:
        return []  # no differences to cover

    results = set()

    def backtrack(current_set: Set[str], index: int):
        if index == len(unique_delta):
            # check if current_set is minimal
            for f in list(current_set):
                subset = current_set - {f}
                # check if subset is a hitting set
                is_hitting = True
                for d in unique_delta:
                    if not subset & d:
```

```python
def enumerate_minimal_support_sets(examples: List[Example], s: int) -> List[Set[str]]: •••
```

```python
                results.add(frozenset(current_set))
            return

        current_d = unique_delta[index]
        # check if current_set already hits current_d
        if current_set & current_d:
            backtrack(current_set, index + 1)
        else:
            for f in current_d:
                new_set = current_set | {f}
                if len(new_set) > s:
                    continue
                # prune if new_set is a superset of any existing result
                if any(existing.issubset(new_set) for existing in results):
                    continue
                backtrack(new_set, index + 1)

    backtrack(set(), 0)

    # filter to ensure minimality (remove sets that have subsets in results)
    minimal_support = []
    for candidate in results:
        candidate_set = set(candidate)
        if len(candidate_set) > s:
            continue
        is_minimal = True
        # check all subsets with one fewer element
        for f in candidate_set:
            subset = candidate_set - {f}
            if subset in results:
                is_minimal = False
                break
        if is_minimal:
            minimal_support.append(candidate_set)

    # remove duplicates and sort by size and elements for deterministic order
    minimal_support = list({frozenset(s): s for s in minimal_support}.values())
    minimal_support = [set(s) for s in minimal_support]
    minimal_support.sort(key=lambda x: (len(x), sorted(x)))

    return minimal_support
```

enumerating all possible
mostly distinguishing features



```python
def compute_branching_set(examples: List[Example], S: Set[str]) -> Set[str]:
    """compute branching set R0 for support set S (lemma 14)"""
    global gamma

    # group examples by their S-feature values (partial assignments α)
    equivalence_classes = defaultdict(list)
    for e in examples:
        alpha = tuple(sorted((f, e.features[f]) for f in S))  # represents α
        equivalence_classes[alpha].append(e)

    # select one representative per non-empty equivalence class
    E_S = [next(iter(group)) for group in equivalence_classes.values()]

    # compute δ(e, γ) for each representative
    R0 = set()
    for e in E_S:
        delta = {f for f, val in e.features.items() if gamma.get(f) != val}
        R0.update(delta)

    return R0
```

a little hack
to find the “branching set”.

thanks for listening!