

```

1  # In file program.py.
2  from queues import BeamPQSet
3  from retrosynthetic_expansion import ParentRetrosteps
4
5  class SmilesGraph():
6      """A bipartite graph comprised of chemical and reaction nodes."""
7
8      def __init__(self, task_data):
9          """Creates an initial graph containing only the synthesis target."""
10
11      def add_spider(self, spider):
12          """Adds one "spider" (all retrosynthetic options for a given retron) to the
13             graph."""
14
15      def get_spider(self, smiles):
16          """Returns all reactions from the graph which produce molecule with the given
17             smiles."""
18
19      def is_expanded(self, smiles):
20          """Returns boolean information if a spider for chemical with given smiles was
21             already added."""
22
23      def is_terminal(self, smiles):
24          """Returns boolean information if the molecule does not need to be synthesized.
25
26             Equivalently, returns boolean information if the molecule can be used as a
27             starting
28             material for the synthesis. Whether a molecule is terminal is decided based on
29             some conditions specified by the user, like the molecule being
30             buyable (and optionally having some maximum price per gram) or being known
31             in the literature (optionally also with some minimum number of literature
32             occurrences). One can also specify such molecule's maximum molecular weight.
33             """
34
35      def get_terminal_cost(self, smiles):
36          """Returns the cost of a terminal molecule with a given smiles.
37
38             For buyable molecules, such cost is the price in dollars of one gram of
39             the molecule, while for molecules which are not buyable but known in the
40             literature the cost is predicted based on the molecule's number of
41             literature occurrences.
42             """
43
44  class SearchRunner():
45      """A class for exploring the chemical space."""
46
47      def __init__(self, task_data):
48          self.target = task_data["target"]
49          self.smiles_graph = SmilesGraph(task_data)
50          # The number of sets of reactions' nonterminal substrates per product to
51          # consider in the search.
52          self.num_nonterminal_subs_per_product = task_data["num_reactions_per_product"]
53          # Used in the BeamPQ queue (see file queues.py),
54          # influences search width.
55          self.beam_width = task_data["beam_width"]
56          # A reaction scoring function, used to assign costs to reactions.
57          self.rsfs = self.get_rsf(task_data["rsf_formula"])
58          # A list of chemical scoring functions.
59          self.csfs = [get_csf(csf_formula) for csf_formula in
60                      task_data["csf_formulas"]]
61          # A list of functions for scoring search nodes.
62          node_scoring_functions = [self.get_node_scoring_function(csf) for csf in
63                                   self.csfs]
64          # A queue using several beam-search-inspired subqueues, each
65          # utilising a different node-scoring function.
66          self.queue = BeamPQSet(node_scoring_functions, task_data["beam_width"])
67          # A generator of spiders corresponding to individual retrosynthetic steps.
68          self.retro_step_generator = ParentRetrosteps(task_data["retro_step_params"])
69
70      def get_rsf(self, rsf_formula):
71          """Returns reaction scoring function corresponding to rsf_formula."""
72
73      def get_csf(self, csf_formula):

```

```

68     """Returns chemical scoring function corresponding to csf_formula."""
69
70     def get_node_scoring_function(self, csf):
71         """Returns a function for scoring search nodes using csf."""
72         # node["cost_so_far"] is the cost of the reactions and terminal
73         # substrates in the found part of a pathway corresponding to a node, while
74         # node["nonterminal_substrates"] are the nonterminal substrates
75         # that still need to be synthesized.
76         return lambda node: node["cost_so_far"] + sum(csf(s) for s in
77             node["nonterminal_substrates"])
78
79     def get_best_rx_nonterminal_substrates_costs_so_far(self, smiles):
80         """Finds best nonterminal substrates of reactions producing smiles.
81
82         First, uses self.smiles_graph.get_spider(smiles) to get all reactions
83         producing the smiles. Then, computes a list of pairs
84         (rx_nonterminal_subs, rx_cost_so_far), where rx_nonterminal_subs are
85         sets of nonterminal substrates of such reactions and rx_cost_so_far are
86         the smallest sums of reaction scoring functions (computed using self.rsfs)
87         and costs of terminal substrates of reactions with the given
88         rx_nonterminal_subs (self.smiles_graph.is_terminal is used to
89         check which substrates are terminal and self.get_terminal_cost to
90         compute their cost). Finally, returns a list of up to
91         self.num_nonterminal_subs_per_product pairs
92         (rx_nonterminal_subs, rx_cost_so_far) with the lowest values of:
93         rx_cost_so_far + sum(self.csfs[0](s) for s in rx_nonterminal_subs).
94         """
95
96     def get_descendant_nodes(self, node, sub):
97         """Returns descendant nodes of node due to expansion of sub."""
98
99         descendant_nodes = []
100         # The expanded substrate is removed from the new set of nonterminal substrates.
101         new_nonterminal_subs = {s for s in node["nonterminal_substrates"] if s != sub}
102         # For each set of nonterminal substrates and "costs so far"
103         # of some reactions producing sub.
104         for rx_nonterminal_subs, rx_cost_so_far in
105             self.get_best_rx_nonterminal_substrates_costs_so_far(sub):
106             descendant_nodes.append({
107                 "cost_so_far": node["cost_so_far"] + rx_cost_so_far,
108                 "nonterminal_substrates": new_nonterminal_subs + rx_nonterminal_subs,
109                 "depth": node["depth"] + 1
110             })
111         return descendant_nodes
112
113     def run(self):
114         """Runs an algorithm for exploring chemical space."""
115         # We provide a simplified, sequential version of our algorithm.
116         # Search node corresponding to the target of the search.
117         target_node = {
118             "nonterminal_substrates": {self.target},
119             "cost_so_far": 0,
120             "depth": 0
121         }
122         self.queue.push(target_node)
123
124         try:
125             while True:
126                 node = self.queue.pop()
127                 for sub in node["nonterminal_substrates"]:
128                     if not self.smiles_graphs.is_expanded(sub):
129                         spider = self.retro_step_generator.get_retrosynthetic_step(sub)
130                         self.smiles_graph.add_spider(spider)
131                         send_expanded_spider_to_selection_process(spider)
132                     for descendant_node in self.get_descendant_nodes(node, sub):
133                         self.queue.push(descendant_node)
134
135         # The exception below is raised when pop was performed on an empty queue,
136         # i.e., the whole available chemical search space has been explored.
137         except QueueEmpty:
138             pass
139
140     class SelectionRunner():

```

```

138     """A class for selecting pathways from the explored chemical space."""
139     def __init__(self, task_data):
140         self.smiles_graph = SmilesGraph(task_data)
141
142     def select_best_pathways(self):
143         """Finds a given number of pathways in self.smiles_graph.
144
145         The algorithm tries to find both diverse and cost-efficient synthetic
146         pathways of the target. Detailed description and pseudocodes of this
147         algorithm were provided in Section S1 of the SI of
148         "Badowski, T., Molga, K. & Grzybowski, B. A. Selection of cost-effective
149         yet chemically diverse pathways from the networks of computer-generated
150         retrosynthetic plans. Chem. Sci. 10, 4640-4651 (2019)."
```

151 """

```

152
153     def run(self):
154         """Runs a pathway selection loop."""
155         while True:
156             # receives new spiders expanded in the search process.
157             new_spiders = get_expanded_spiders()
158             for spider in new_spiders:
159                 self.smiles_graph.add(spider)
160             best_pathways = self.select_best_pathways()
161             send_best_pathways_to_the_user(best_pathways)
162
163     def main():
164         """Runs retrosynthetic search and pathway selection processes."""
165         task_data = get_task_data_from_user()
166         @background
167         SearchRunner(task_data).run()
168         SelectionRunner(task_data).run()
169         # Waits for the search and selection processes to finish
170         # (which happens when the user stops the search or the whole
171         # available space has been explored by the search process).
172         join()
173
174     if __name__ == '__main__':
175         main()
176

```

```

1  # In file queues.py.
2  from collections import defaultdict
3
4  class BeamPQ:
5      """A beam-search-inspired modification of a priority queue.
6
7      It ensures that before the lowest-scored node is popped, a given number
8      of nodes with the lowest scores found so far at each lower search depth
9      must be popped. The queue deduplicates nodes with the same nonterminal
10     substrates, as they require the same pathway endings.
11     """
12
13     def __init__(self, beam_width):
14         # An internal priority queue storing nodes and their scores.
15         # We used a binomial queue.
16         self.q = PriorityQueue()
17         self.beam_width = beam_width
18         # A dictionary mapping search depths to dictionaries mapping
19         # at most beam_width nonterminal substrate tuples with the lowest
20         # minima m of scores of all nodes with such nonterminal substrates
21         # and depths, to such m.
22         self.depths_subs_best_scores = defaultdict(dict)
23         # A dictionary mapping search depths d to dictionaries mapping nonterminal
24         # substrate tuples t of still unpopped nodes n with such depths and
25         # with scores s in self.depths_subs_best_scores[d][t], to tuples (s, n).
26         self.depths_subs_scores_nodes = defaultdict(dict)
27         # A dictionary mapping nodes' nonterminal substrate tuples to the lowest
28         # scores of nodes with such nonterminal substrates pushed so far.
29         self.subs_best_scores = {}
30         # A set of pairs (score, nonterminal substrate tuple) for nodes which were
31         # already popped from self.depths_subs_scores_nodes.
32         self.popped_from_beam = set()
33
34     def update_beam(self, score, node):
35         """Updates information on beam_width records.
36
37         Uses self.depths_subs_best_scores to check if the score beats some of
38         the self.beam_width score records at the node's depth and for its
39         nonterminal nodes. If so, then updates self.depths_subs_best_scores and
40         self.depths_subs_scores_nodes.
41         """
42
43     def push(self, node, score):
44         """Pushes onto the queue."""
45         subs = tuple(sorted(node['nonterminal_substrates']))
46         # If the nonterminal substrates were not encountered yet or improve the
47         # score record.
48         if not subs in self.subs_best_scores or score < self.subs_best_scores[subs]:
49             self.subs_best_scores[subs] = score
50             self.q.push((score, node))
51             self.update_beam(score, node)
52
53     def pop_check_beam(self, score, node):
54         """Updates popped node using information on beam_width records.
55
56         If there are some nodes in self.depths_subs_scores_nodes at lower depths
57         than the popped node, then a node with the lowest score at the lowest
58         depth is popped from self.depths_subs_scores_nodes and returned with its
59         score, while the argument node and score are pushed back onto self.q.
60         Otherwise, the argument node and score are returned.
61         """
62
63     def node_to_skip(self, score, node):
64         """Returns if the node should be skipped.
65
66         The node should be skipped if its score is higher than for the node's
67         nonterminal substrates in self.subs_best_scores or if such substrates and
68         score are in self.popped_from_beam.
69         """
70
71     def pop(self):
72         """Pops from the queue."""

```

```

73     self.pop_stats = None
74     score, node = self.q.pop()
75     while self.node_to_skip(score, node):
76         score, node = self.q.pop()
77     return self.pop_check_beam(score, node)
78
79 class BeamPQSet:
80     """A queue using several BeamPQs with different scoring functions."""
81     def __init__(self, node_scoring_functions, beam_width):
82         # A list of functions computing scores of search nodes.
83         self.node_scoring_functions = node_scoring_functions
84         self.num_queues = len(self.node_scoring_functions)
85         self.queues = [BeamPQ(beam_width=beam_width)
86                         for _ in range(len(self.num_queues))]
87         # The index of the queue previously popped from, initialized to -1.
88         self.i = -1
89         # A set of pairs of form:
90         # (node's nonterminal substrates tuple, tuple of scores for the node)
91         # for all the already popped nodes.
92         self.popped_subs_scores = set()
93
94     def push(self, node):
95         """Pushes node and computed scores onto all the component queues."""
96         scores = tuple(scoring_fun(node) for scoring_fun in self.node_scoring_functions)
97         node["scores"] = scores
98         for q, score in zip(self.queues, scores):
99             q.push(node, score)
100
101     def pop_from_single_queue(self, q):
102         """Pops from q.
103
104         Pops from q until a node with nonterminal substrates and node["scores"]
105         not present in self.popped_subs_scores is found, in which case
106         self.popped_subs_scores is updated and the node is returned, or the queue
107         becomes empty, in which case a QueueEmpty exception is raised.
108         """
109
110     def pop(self):
111         """Pops and returns node from the next nonempty component queue.
112
113         If all the component queues are empty raises a QueueEmpty exception.
114         """
115         # The number of component queues found to be empty.
116         num_empty_queues = 0
117         # While there can be some nonempty component queue.
118         while num_empty_queues < self.num_queues:
119             # Computes the index of the next queue.
120             self.i = (self.i + 1) % self.num_queues
121             try:
122                 return self.pop_from_single_queue(self.queues[self.i])
123             # If the queue is empty.
124             except QueueEmpty:
125                 num_empty_queues += 1
126         # Now all component queues are empty.
127         raise QueueEmpty
128

```

```

1  # In file retrosynthetic_expansion.py.
2  # This is a simplified pseudocode for computing retrosynthetic analysis for a
3  # given target (i.e., a "spider" of retrosynthetic possibilities) as provided
4  # by the get_retrosynthetic_step method. For the sake of consistency and
5  # simplicity, we skip here e.g. code parallelization, we abstract also from
6  # efficiency (copying vs. sharing data, pre-loading results, server-client
7  # architecture for certain functionalities), and other implementation details.
8
9  class ParentRetrosteps(object):
10     def __init__(self, params):
11
12         self.params = params
13         # Other initializations.
14
15     def match_smiles_to_reaction(self, smiles, reaction):
16         """Checks if reaction is matching to smiles.
17
18         Inspects if smiles matches to main retron of the reaction's
19         transformation/rule.
20         """
21
22     def reaction_cores_overlap(self, r_prev, r_next):
23         """Checks whether reaction cores overlap.
24
25         Owing to appropriate tracking of atoms in a reaction (correspondance
26         between product and substrates atoms), we are able to check the condition
27         important for an FGI, i.e., overlap between consecutive
28         reactions' cores.
29         """
30
31     def compute_FGI(self, fgi, smiles):
32         """Computes Functional Group Interconversions (FGI) for a given target.
33
34         If required by the user (self.params) considers FGI composed of sequences
35         of two/three reaction steps, and operating within the same region of the
36         molecule, by applying them sequentially (computing simple
37         retrosynthesis steps).
38         """
39         res = self.compute_retrosynthesis(fgi.reactions[0], smiles)
40         while (len(fgi.reactions) > 0):
41             fgi.reactions = fgi.reactions[1:]
42             new_tmp_res = []
43             for r in res:
44                 matching_substrates = [s for s in r.substrates if
45                     self.match_smiles_to_reaction(s, fgi.reactions[0])]
46                 for sub in matching_substrates:
47                     res_tmp = self.compute_retrosynthesis(fgi.reactions[0], sub)
48                     res_tmp_accepted = [r_next for r_next in res_tmp if
49                         self.reaction_cores_overlap(r, r_next)]
50                     res_tmp_combined = self.combine_results(r, res_tmp_accepted)
51                     new_tmp_res.extend(res_tmp_combined)
52             res = new_tmp_res
53             # Removes results with forbidden motifs that are defined for a given FGI.
54             results = self.remove_forbidden_motifs(res, fgi)
55             # Prepares results s.t. they can be used within results of
56             # get_retrosynthetic_step.
57             return self.postprocess_results(results)
58
59     def run_rxn_as_long_as_possible(self, substrates, prev_res):
60         """Applies reaction prev_res to substrates iteratively as long as possible."""
61         already_found = set()
62         applied_rxis = [] # list, not set, as we count repetitive applications
63         tmp_res = []
64         substrates_lst = [(0, substrates, prev_res)]
65         while True:
66             results_changed = False
67             new_tmp_res = []
68             for (cnt, substrates, prev_res) in substrates_lst:
69                 chosen_subs_lst = [s for s in substrates if self.match_smiles_to_reaction(s,
70                     prev_res.reaction)]
71                 if len(chosen_subs_lst) != 0: # do not process further empty or ambiguous

```

```

70     matchings
71     continue
72     chosen_subs = chosen_subs_lst[0]
73     curr_res_lst_orig = self.compute_retrosynthesis(prev_res.reaction,
74     chosen_subs)
75     curr_res_lst = [curr_res for curr_res in curr_res_lst_orig if \
76                     self.result_fulfills_multirx_condition(curr_res)]
77     for r in curr_res_lst:
78         if r in already_found:
79             continue
80         # Updates parameters needed for, e.g., non-selectivity and
81         # protections/conflicts assignments for the whole sequence of reactions.
82         self.update_parameters(r)
83
84         already_found.add(r)
85         results_changed = True
86         new_tmp_res.append(r)
87     if not results_changed:
88         break
89     tmp_res = new_tmp_res
90     substrates_lst = [(cnt + 1, r.substrates, r) for r in tmp_res]
91
92     return tmp_res
93
94 def compute_simultaneous_reactions(self, sr, smiles):
95     """Computes sequences of simultaneous reactions.
96
97     If required by the user (self.params), looks for hard-coded sequences
98     of reactions and applies them sequentially (as single retrosynthetic
99     steps), each as many times as possible.
100     """
101     results = []
102     for rxn in sr.reactions:
103         if results == []:
104             prev_res = None
105             substrates = [smiles]
106         else:
107             prev_res = results[0]
108             substrates = results[0].substrates
109             res_run_for_rxn = self.run_rxn_as_long_as_possible(rxn, substrates, prev_res)
110             if res_run_for_rxn == []:
111                 continue
112             results = res_run_for_rxn
113         # For simultaneous reactions, protections and
114         # conflicts are computed after the whole sequence of reactions.
115         self.adjust_protections_and_incompatibilities(results)
116         # Prepares results s.t. they can be used within results of
117         # get_retrosynthetic_step.
118         return self.postprocess_results(results)
119
120 def post_filters_apply(self, r):
121     """Rejects results with conflicts etc.
122
123     E.g., for bypasses it is useful to compute also results with conflicts,
124     analyze them (e.g., while looking for candidates for bypasses), and then
125     reject instead of rejecting them on-the-fly.
126     """
127
128 def match_smiles_to_reactions(self, smiles):
129     """Returns subset of reactions matching to smiles.
130
131     Fast filtering feasible reactions for a given target. We skip here
132     implementation details.
133     """
134
135 def match_smiles_to_strategy(self, smiles, res):
136     """Returns set of reaction pairs (strategies) matching to smiles.
137
138     Returned strategies are matching to smiles if their first step is a
139     reaction used in res, and their second step's main retron matches to smiles.
140     """

```

```

140
141 def get_bypass_candidates(self, main_product, results_all):
142     """Selects candidates for by-passes.
143
144     Choses reactions that have conflicts and/or non-selectivities, but at the
145     same time offer:
146     (a) high (e.g., best 10% in results_all) decrease in csf (a chemical
147     scoring function, either default or user-defined); Reaction should also
148     have better csf for substrates than for main_product
149     OR
150     (b) increase in the number of stereocenters between substrates and
151     main_product.
152     """
153
154     def gain_in_stereo(res):
155         prod_stereo = sum(p.count_stereo() for p in res.products)
156         subs_stereo = sum(s.count_stereo() for s in res.substrates)
157         return (prod_stereo > subs_stereo)
158
159
160     t_csf = self.calc_csf(main_product)
161     csf_distr = [sum(self.calc_csf(s) for s in r.substrates) for r in results_all]
162
163     th_acc = self.get_percentile(csf_distr, 10)
164
165     accepted = []
166     for i, cand in enumerate(results_all):
167         if not (cand.has_nonselctivity or cand.has_conflict):
168             continue
169         s_func = csf_distr[i]
170         if ((s_func < t_csf) and (s_func <= th_acc)) or gain_in_stereo(cand):
171             accepted.append(cand)
172         else:
173             continue
174
175     return accepted
176
177 def combine_steps(self, r1, r2):
178     """Combines sequences of two reactions into one super-reaction.
179
180     For internal processing, sequence of two reactions is combined into a data
181     structure that can be further processed as an alternative to a single
182     reaction, within results of get_retrosynthetic_step function.
183     """
184
185 def submit_second_step_strategy(self, res1):
186     """Computes second step of strategy according to a predefined set of
187     strategies."""
188
189     results = []
190     for subs in res1.substrates:
191         candidate_reactions = self.match_smiles_to_strategy(subs, res1)
192         for cr in candidate_reactions:
193             res = self.compute_retrosynthesis(cr, subs)
194             for r in res:
195                 if not self.post_filters_apply(r):
196                     results.append(self.combine_steps(res1, r))
197     return results
198
199 def get_substrates_with_bypass_allowed(self, cand, substrates):
200     """Selects cand substrates for which bypass can be applied.
201
202     Selects substrates that match cand reaction and do not contain problematic
203     groups (causing conflicts or nonselectivities); also checks e.g., if its
204     "sibling" substrates do not have such problems (to avoid a situation when
205     a problematic group is separated by a reaction but remains in sibling
206     substrate, instead of being transformed by this reaction).
207     """
208
209     #implementation is ommited here
210
211 def submit_second_step_bypass(self, res1, bypass_candidates):

```



```

211     """Computes second step of strategy according to a predefined set of
212     strategies."""
213
214     results = []
215     already_applied_rxns = set()
216     for bypass_cand in bypass_candidates:
217         if bypass_cand.rxid in already_applied_rxns:
218             continue
219
220         synthons_idx_matching_rxn2 =
221         self.get_substrates_with_bypass_allowed(bypass_cand, res1.substrates)
222         for subs in synthons_idx_matching_rxn2:
223             res2 = self.compute_retrosynthesis(bypass_cand, subs)
224             for r in res2:
225                 if not self.post_filters_apply(r):
226                     results.append(self.combine(res1, r))
227             return results
228
229 def apply_reaction(self, rxn, smiles):
230     """Runs retrosynthetic rule for selected smiles.
231
232     This code allows for maintaining appropriate stereochemistry, bonds, etc.
233     for each transformation executed according to a retrosynthetic rule.
234     """
235
236 def check_cycloaddition(self, result):
237     """Checks if result is accepted by a classifier built on a particular training
238     set.
239
240     For example, a filter based on a Random Forest classifier is used to
241     evaluate substrates for Diels-Alder cycloaddition. For more details on the
242     use of phisically-relevant descriptors in such classifiers, please, refer to
243     Beker, W., Gajewska, E. P., Badowski, T. and Grzybowski, B. A. Prediction of
244     major regio-, site-, and diastereoisomers in Diels-Alder reactions by using
245     machine-learning: The importance of physically meaningful descriptors.
246     Angew. Chem. Int. Ed. 58, 4515-4519 (2019).
247     """
248
249 def check_forbidden_structures(self, result):
250     """Detects if a reaction from result contains forbidden structures.
251
252     Forbidden structures may be:
253     - highly-strained or chemically nonsensical motifs (expert-defined motifs
254       not allowed neither in main product nor in substrates)
255     - "suspicious motifs" allowed in substrates only if they are present
256       in main_product.
257
258     Returns True if forbidden structure is detected.
259     Returns False otherwise.
260     """
261
262     for fm in self.forbidden_motifs: # expert-defined motifs
263         if result.main_product.contains(fm):
264             return True
265         for sub in result.substrates:
266             if sub.contains(fm):
267                 return True
268     for wm in self.warm_motifs: # expert-defined motifs
269         if not result.main_product.contains(wm):
270             for sub in result.substrates:
271                 if sub.contains(wm):
272                     return True
273
274     return False
275
276 def check_cross_incompatibilities(self, result):
277     """Detects if target of any synthon contains cross-incompatibility
278
279     Cross-incompatibilities are sequences of chemical groups that should not
280     appear together with neither main
281     retron nor any of its synthons.
282     Returns True if cross-incompatibility detected,

```

```

280     Returns False otherwise.
281     """
282
283     for ci in self.cross_incompatibilities: # expert-defined sequences
284         if all(result.main_product.contains(c) for c in ci):
285             return True
286         for sub in result.substrates:
287             if all(sub.contains(c) for c in ci):
288                 return True
289     return False
290
291 def check_aromatic_substitution(self, result):
292     """Checks the applicability of an aromatic substitution to given reaction
293     substrates.
294
295     This filter involves application of, for instance, Hammett substituent
296     constants, extended-Hueckel method, proton-affinity calculations,
297     and several other heuristics. For detailed description of methods used here
298     please, refer to Klucznik, T. et al. Efficient syntheses of diverse,
299     medicinally relevant targets planned by computer and executed in the
300     laboratory. Chem 4, 522-532 (2018). Supplementary Information, Section S5
301     """
302
303 def apply_postfilters(self, result):
304     """Filters out reactions which do not satisfy post-filters.
305
306     Applies postfilters that, for instance:
307     - remove reactions with forbidden motifs present in the substrates
308       (see function self.check_forbidden_structures)
309     - remove reactions with cross-incompatibilities
310       (see self.check_cross_incompatibilities)
311     - remove results with disallowed aromatic substitution
312       (see self.check_aromatic_substitution)
313     - applies cycloaddition filter (see self.check_cycloaddition)
314     - runs additional checks for unexpected changes in stereochemistry
315     """
316
317 def compute_protection_and_conflict_information(self, result):
318     """Computes protections/incompatibilities information.
319
320     Assigns information about possible protections/incompatibilities based on
321     groups specified in expert-coded reactions. Detects if these groups are
322     present on the same atoms both in target smiles and in any substrate.
323     Also, considers predefined side-reactions and rearrangements.
324
325     For more details of protecting/conflicting groups assignments please refer
326     to Klucznik, T. et al. Efficient syntheses of diverse, medicinally relevant
327     targets planned by computer and executed in the laboratory.
328     Chem 4, 522-532 (2018). Supplementary Information, Fig. S9
329     """
330
331 def eliminate_forbidden_groups_combinations(self, result):
332     """Eliminates reactions with unexpected combinations of functional groups in any
333     synthon.
334
335     Details of implementation are omitted here."""
336
337 def assign_nonselectivity(self, result):
338     """Assigns non-selectivity variable that might be used in a reaction scoring
339     function.
340
341     Inspects if an inverted reaction (in the forward, not retrosynthetic
342     direction) run may produce an alternative product.
343     If so, the corresponding non-selectivity variable is set to one,
344     otherwise it is set to zero.
345     """
346
347 def detects_labile_groups(self, result):
348     """Detects if reaction from result contains labile groups.
349
350     Labile group is a group from expert-defined list. Reaction contains a labile
351     group if it is a substructure of (main) product and (any) substrate

```

```

349     """
350
351     for lg in self.labile_groups: # elf.labile_groups is a pre-defined list
352         if result.main_product.contains(lg):
353             if any(sub.contains(lg) for sub in result.substrates):
354                 self.assign_labile_variable_for_rsf(result)
355
356     def compute_chemical_parameters(self, result):
357         """Computes additional parameters for result's substrates.
358
359         These parameters may be used, e.g., in some scoring functions
360         (e.g., heuristic ones based on the number of rings, or adjusted
361         length of smiles representation) or as stop-points for the retrosynthesis
362         algorithm (e.g., molecular weight).
363         """
364
365     def compute_retrosynthesis(self, rxn, smiles):
366         """Generates all viable options within a single retrosynthetic step for smiles
367         with reaction rxn."""
368
369         results_1st = self.apply_reaction(rxn, smiles)
370         results = []
371         for result in results_1st:
372             # post-process result
373             if self.apply_postfilters(result):
374                 continue
375             self.compute_protection_and_conflict_information(result)
376             if self.eliminate_forbidden_groups_combinations(result):
377                 continue
378             self.assign_nonselectivity(result)
379             self.detects_labile_groups(result)
380             self.compute_chemical_parameters(result)
381             results.append(result)
382         return results
383
384     def get_retrosynthetic_step(self, smiles):
385         """Computes single retrosynthesis step.
386
387         Returns the whole "spider" of all retrosynthetic possibilities either in
388         individual steps, or by applying strategies, bypasses, FGIs,
389         simultaneous reactions, etc., to form causal reaction sequences.
390         """
391
392         candidate_reactions = self.match_smiles_to_reactions(smiles)
393         results_all = []
394         results_filtered = []
395         for cr in candidate_reactions:
396             res = self.compute_retrosynthesis(cr, smiles)
397             for fgi in self.fgi_dict[cr]: # expert-coded sequences of FGI starting with cr
398                 res.extend(self.compute_FGI(fgi, smiles))
399             for sr in self.simultaneous_reactions[cr]: # expert-coded sequences starting
400                 with cr
401                 res.extend(self.compute_simultaneous_reactions(sr, smiles))
402             results_all.extend(res)
403             for r in res:
404                 if not self.post_filters_apply(r):
405                     results_filtered.append(r)
406                     results_filtered.extend(self.submit_second_step_strategy(res))
407
408         res_filtered2 = results_filtered.copy()
409         for res in results_filtered:
410             bypass_candidates = self.get_bypass_candidates(smiles, results_all)
411             res_filtered2.extend(self.submit_second_step_bypass(res, bypass_candidates))
412         return res_filtered2

```