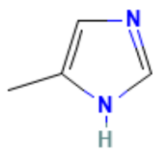
Recap: After task 1, 2, 3, We have successfully mapped most molecules to its corresponding Martini 3 beads structure

TASK 4:

Goal: Fix the algorithm to add more rules to be able to map more molecules to its corresponding martini 3 beads



1. 4-Methylimidazole

Issue: double bonds in 5-atoms ring are generally uninterrupted by lone atoms, but in this case we need to merge the lone C atom from the non-ring section into one of the double bond pair

```
(A) Build array0:
    For every atom that has:
      - element (index 1) is 'C' (or 'c')
      - final[atom[0]] is ""
      - AND it has at least one inner neighbor (from its inner connections at index 4)
        such that the neighbor's element is 'C' or 'N' AND the bond equals 2,
    add the pair (atom[0], neighbor[0]) as a tuple to array0.
For each tuple in array0:
    - If both atoms are C: assign bead "TC5" + random string.
    - If one atom is N: assign bead "TN6a" + random string.
```

here is the current code for the map_nonbenzene_5_ring_section function. We will copy most of the codes from the benzene section for this.

Here is what we need to change:

instead of - If both atoms are C: assign bead "TC5" + random string.

we do - - If both atoms are C:

if both atoms each have a foreign neighbor with the foreign sections of size 1: return an error saying 5-ring section is too difficult to be mapped and add a todo comment for later

if one atom has a foreign neighbor with the foreign section of size 1:

// Do something similar to benzene mapping as follow:

assign a bead based on the foreign atom type (using the first outer connection):

O: "SN6", N: "SN6d", S: "SC6", Cl: "SX3", I: "X1", C: "SC4", Br: SX2

assign it + random string to all 3 atoms

assign bead "TC5" + random string.

2. 4-BROMOANISOLE

Issue: A new bead type was found

Fix: Add new bead type to the dictionary: TX2, this is a alternative of Br in benzene, which was SX2. Following this logic, let's add more beads:

SX3: CC + Cl

SC6: CC + S

SN6d: CC + N

X1: CC + I

so, now we will have: TX3, TC6, TN6d. TX1 as C + Cl/S/N/I respectively

3. CHLORPROPHAM

A few issues here. First is the mapping of the Benzene ring section is not working

Issue: we need to treat atoms with no foreign and atoms with foreign of size 3+ or size 2 and the two atoms are not C and O respectively to be the same.

This issue has already been fixed by some additional checks

Issue 2: when the number of atoms in the array 1 is an odd numbers, we need to make sure the pairing can leave out the atom that is connected to the rest of the unmapped atoms in the benzene ring

Fix: from this line of instruction from

3. Build array1: atoms with no outer connections or outer connections of size 3+ or 2 that aren't CO that are unmapped; pair them (leftovers → array2) and assign beads ("TC5" if both C, "TN6a" if one N).

into:

3. Build array1: atoms with no outer connections or outer connections of size 3+ or 2 that aren't CO that are unmapped;

if the amount of atoms in this array is odd:

first find any atom that is not connected to any other atoms in this array and remove it. If found then go straight to the pairing

if not found, find any atom that is only connecting to exactly 1 other atom in this array and remove it, if found then go to the pairing

if not found, throw an error saying benzene ring odd parity is bugged

pair them (leftovers → array2) and assign beads ("TC5" if both C, "TN6a" if one N).

Issue 3: currently the code does not have a solution for if there are no candidate edges to choose from, and it will result to a dummy default mode, which does not lead to any solution.

We need to change a lot to make this work

currently that section of code is :

```
if len(candidate_edges) > 2:
    from collections import deque
    graph = {i: [tup[0] for tup in atom[4]] for i, atom in
enumerate(section)}
    def bfs_local(start: int, end: int) -> int:
        visited = set()
        queue = deque([(start, 0)])
        while queue:
            current, dist = queue.popleft()
            if current == end:
                return dist
          visited.add(current)
            for nbr in graph.get(current, []):
                if nbr not in visited:
                    queue.append((nbr, dist + 1))
        return float('inf')
    to_remove = set()
    for i in range(len(candidate_edges)):
        for j in range(i+1, len(candidate_edges)):
            if bfs_local(candidate_edges[i], candidate_edges[j]) <= 4:
                to_remove.add(candidate_edges[i])
                to_remove.add(candidate_edges[j])
    candidate_edges = [i for i in candidate_edges if i not in
to_remove]
    if not candidate_edges:
        candidate_edges = [0, len(section)-1]

# Use the first candidate as seed.
seed = candidate_edges[0]
def dfs_trace(local_idx: int, visited: set) -> List[int]:
    trace = [local_idx]
    visited.add(local_idx)
    for (nbr, bond) in section[local_idx][4]:
        if nbr not in visited and len(section[nbr][4]) < 3:
            trace.extend(dfs_trace(nbr, visited))
            break
    return trace
visited_trace = set()
trace_nodes = dfs_trace(seed, visited_trace)
# NEW CONDITION 1: If the trace is long enough and the fourth node has
≥ 3 inner connections, limit trace to 3 nodes.
if len(trace_nodes) >= 4 and len(section[trace_nodes[3]][4]) >= 3:
    trace_nodes = trace_nodes[:3]
if len(trace_nodes) > 4:
    trace_nodes = trace_nodes[:4]
# NEW CONDITION 2: Ensure remainder has at least 2 atoms.
```

```
while len(section) - len(trace_nodes) < 2 and len(trace_nodes) > 2:
    trace_nodes = trace_nodes[:-1]
if len(section) - len(trace_nodes) < 2:
    raise ValueError("Subdivision not possible: remainder too small
(normal method).")
sub_old_indices = sorted(trace_nodes)
rem_old_indices = [i for i in range(len(section)) if i not in
sub_old_indices]
sub_section = [section[i] for i in sub_old_indices]
remainder = [section[i] for i in rem_old_indices]
return sub_section, remainder, sub_old_indices, rem_old_indices
```

the part we care about here is: `if not candidate_edges:`

how can we fix this case? This case is not in the Building block table so we never had to worry about it. But now that it is, let's do the following:

first, we will only do the tracing if candidate edge is not empty

second, before we even remove too-close candidate edges, we will make a copy of the array (make sure the copy does not change as we change the main array)

if not candidate_edges:

create a new function that does the following:

reset candidate edges back to its copy

create an empty array

use bfs again and iterate through each edge and find the shortest distance between that edge to any other edges

if this array is empty or the distance is the same as the distance already in the array

save it to said array as an array of the form [edge1, distance, edge2]

if the distance is less than the distance already in the array, empty the array and add this new [edge1, distance, edge2] into the array if the distance is more or there is already an element in the array with the same distance but have the edges just swapped place: continues

now this array should have pairs of same distance edges tracing

now comes the hard part, for each element in this array, we have a pair of edges, we will try the following for each element:

1. treat the path to these edges as 1 part and if the other part are the rest of the atoms.
2. If the other part is all connected it will be its own part.
3. If not, we know that this tracing path had cut the rest of the atoms into 2 smaller sections. For this case we check the length of these 2 cut sections.
4. if they are the same, we will try the following for each section:

   merge the section to the main section that we made from connecting the edges and make it 1 big section and leave the other section as its own section

5. now that we guaranteed we have made 2 unique section that fits with the rules of martini, we run the rest of the steps to see if these divisions of sections is mappable.
6. If they are, keep these divisions and return them
7. if not, go back and choose the other way of merging to see if they are mappable
8. if all fails, return an error message saying non-ring section division leads to unassignable sections

What the f*ck it just works ?????

4. CYCLOPROPANE

Issue: 3-ring?

Fix: add more sections and structures to dictionary

5. CYCLOPENTENE

Issue: mapping miss the case when there aren't any foreign atoms

Fix: add the case. The new instruction will look like this:
If found: If the foreign neighbor (from the first outer connection) has element 'C' and the size of its section is 1,

assign "SC3" + random string to that atom, the foreign neighbor, and its first inner neighbor.

Else if the foreign neighbor has element 'O' and its section size is 1,

assign "SN6" + random string similarly.

Else if the foreign neighbor's section size is > 1, assign "TC3" + random string to that atom and its first inner neighbor.

If not found:

take any connecting 2 atoms and assign TC3 + random string to those two atoms

6. N-BOC-2-AMINOPHENOL

Issue: new bead type found and added: SC1: Structure: C(C)(C)(C)

Issue 2: Very complex structure for non-ring section.

currently the SUCCESS is just as long as it work it is a success but when i run the whole code i will get this
ValueError: Non-ring section cannot be mapped: no candidate bead found for 2-edge mapping (path: O=CN).

Fix: I wont set success to true unless I made sure that when i "try" the other function and everything and it gives me no errors and a final mapping, then I set it to true. Otherwise it will stay as false.

Issue 3: need to try out many different ways of mapping to find the right one

fix: trial and error:

try: dummy_final = final[:] # Test option1 # Automatically find the section index in full_mapping. section_index = full_mapping.index(section) dummy_full_mapping = full_mapping[:]

# Remove the section being subdivided dummy_full_mapping.pop(section_index) # Insert the candidate subdivisions in its place. dummy_full_mapping.insert(section_index, [section[i] for i in option1_sub]) dummy_full_mapping.insert(section_index + 1, [section[i] for i in option1_rem]) # --- New reindexing step: reindexed_sub = reindex_section([section[i] for i in option1_sub], option1_sub) reindexed_rem = reindex_section([section[i] for i in option1_rem], option1_rem) # Now update the boundary mapping on the reindexed candidate subdivisions. updated_sub_section, updated_remainder = update_divided_section_mapping(reindexed_sub, reindexed_rem)

```
dummy_final = map_non_ring_section_1bead(updated_sub_section, dummy_final,
martini_dict, dummy_full_mapping) dummy_final =
map_non_ring_section_1bead(updated_remainder, dummy_final, martini_dict,
dummy_full_mapping) candidate_sub_old_indices = option1_sub
candidate_rem_old_indices = option1_rem except Exception: try: dummy_final = final[:] #
Test option2 section_index = full_mapping.index(section) dummy_full_mapping =
full_mapping[:] dummy_full_mapping.pop(section_index)
dummy_full_mapping.insert(section_index, [section[i] for i in option2_sub])
dummy_full_mapping.insert(section_index + 1, [section[i] for i in option2_rem]) # --- New
reindexing step: reindexed_sub = reindex_section([section[i] for i in option2_sub],
option2_sub) reindexed_rem = reindex_section([section[i] for i in option2_rem],
option2_rem) updated_sub_section, updated_remainder =
update_divided_section_mapping( reindexed_sub, reindexed_rem ) dummy_final =
map_non_ring_section_1bead(updated_sub_section, dummy_final, martini_dict,
dummy_full_mapping) dummy_final = map_non_ring_section_1bead(updated_remainder,
dummy_final, martini_dict, dummy_full_mapping) candidate_sub_old_indices =
option2_sub candidate_rem_old_indices = option2_rem except Exception: continue else: #
When the two components have different sizes, choose the smaller one. if
len(components[0]) > len(components[1]): merged_sub =
sorted(candidate_sub_old_indices + components[1]) else: merged_sub =
sorted(candidate_sub_old_indices + components[0]) rem_after_merge = [i for i in
range(len(section)) if i not in merged_sub] if not is_connected(rem_after_merge, section):
continue candidate_sub_old_indices = merged_sub candidate_rem_old_indices =
rem_after_merge else: continue # *** New test: update boundary mapping and try
mapping the candidate parts. try: dummy_final = final[:] # Make a copy of current final
mapping. section_index = full_mapping.index(section) dummy_full_mapping =
full_mapping[:] # Copy the full mapping. dummy_full_mapping.pop(section_index)
dummy_full_mapping.insert(section_index, [section[i] for i in candidate_sub_old_indices])
dummy_full_mapping.insert(section_index + 1, [section[i] for i in
candidate_rem_old_indices]) # --- New reindexing step: reindexed_sub =
reindex_section([section[i] for i in candidate_sub_old_indices],
candidate_sub_old_indices) reindexed_rem = reindex_section([section[i] for i in
candidate_rem_old_indices], candidate_rem_old_indices) updated_sub_section,
updated_remainder = update_divided_section_mapping( reindexed_sub, reindexed_rem )
dummy_final = map_non_ring_section_1bead(updated_sub_section, dummy_final,
martini_dict, dummy_full_mapping) dummy_final =
map_non_ring_section_1bead(updated_remainder, dummy_final, martini_dict,
dummy_full_mapping)
```

new code section for the trial and error

7. PYRROLIDINE

issue: NH case is missing a case

Fix: add an extra case for it

8. 1,2,4-TRICHLOROBENZENE

issue: direction of mapping is important

Fix: add directional mapping

9. 1-CHLORO-NAPHTHALENE

```
pairs1 = []
used1 = set()
for a in array1:
    if a not in used1:
        for atom in section:
            if atom[0] == a:
                for tup in atom[4]:
                    nbr_local = tup[0]
                    if 0 <= nbr_local < len(section):
                        neighbor = section[nbr_local]
                        if neighbor[0] in array1 and neighbor[0] not in used1:
                            pairs1.append((a, neighbor[0]))
                            used1.add(a)
                            used1.add(neighbor[0])
```

        break

    break

okay so lets fix this pairing:
if the size is 2 then just match them two
if the size is 4: find an atom that is connected to only 1 other atom in array1 and match those 2 and then match the rest
if the size is 6: take any atom and match it with any atoms in array1 that it connects to and delete both of them from array1, then treat the rest as the size 4 case

10. 1-METHYL-NAPHTHALENE: same as 9
11. INDAZOLE

Fix: else: # If both have or both do not have a foreign neighbor, remove one arbitrarily (here, the first one). candidate_indices = [ci for ci in candidate_indices if not (ci[2]=='n' and ci[1] == candidate_n[0][1])] if both do not have a foreign neighbor, remove one that is connecting to only 1 other atom in this array

12. 3-Methyl-1H-indole-2-carboxylic acid

Fix:

```
elif len(candidate_n) == 1 and len(candidate_indices) % 2 == 1:
    # If there is only 1 'n' candidate and 0 2 or 4 'c' candidates
(odd), remove the single 'n'.
    candidate_indices = [ci for ci in candidate_indices if ci[2] !=
'n']
```

Everything except the dmbis should be accounted for now