TASK 1:

Given the smiles code with possible rings

input: Smiles tokens, adjacency matrix, properties matrix

input 1: smiles tokens : [‘c’, ‘1’, ‘c’, ‘c’, ‘c’, ‘c’, ‘c’, ‘1’]

input 2:

[[0 1 0 0 0 1]

[1 0 1 0 0 0]

[0 1 0 1 0 0]

[0 0 1 0 1 0]

[0 0 0 1 0 1]

[1 0 0 0 1 0]]

input 3: properties matrix:

(Element, isRing, isEdge):

[['C', True, False]

['C', True, False]

['C', True, False]

['C', True, False]

['C', True, False]

['C', True, False]]

output: breakdown of what the output should look like

an array(the whole molecule) of arrays(each section, e.g. a ring is a section) of arrays(each atoms) of elements

The most inner arrays is composed of the form:

[index, string element, int ring_status, array of outer connection, array of inner connection, bool isedge]

Explain of the algorithm:

index: This is strictly the index of this atom in the properties matrix (use will be explain later)

String element, isedge: use the property matrix and smiles tokens (subject to change)?

int ring_status: 0 for no ring, 1 for is ring, 2 for is benzene ring, this can be done by tracing the smiles string?

array of outer connection: this array is going to be of the form: [‘index of the section’, ‘index of the atom of that section’, ‘numbers of bond’]

array of inner connection: this array is going to be composed of 3 tuples of the form: (‘index of connected atom’, ‘numbers of bond’)

So how are we going to do this?

Thinking process:

easiest case is when we have either 5 or 6 arrays in the properties matrix that have the isRing = True. Then we know that all of that belongs to a ring and whatever is not that does not belong to a ring.

But what if there are more than 1 rings, chances are one atom is going to be a part of 2 rings. We can generalize this process as follow:

we will look for a few things here:

REFER TO TASK 1a HERE. Seems like this task need to be done before task 1 can be run

loop i from 1 to infinite, increment by 1 every loop. then change i to a string and inspect of that string is found in the token, break if the string is not found (no more ring left)

if the number is found, clearly, the token behind it is a part of the ring i

now, we have to find that token position in the adjacency matrix:

REFER TO TASK 1b HERE

we will now look into the connectivity matrix using the new index we found

REFER TO TASK 1c HERE

Using the function from TASK 1c, we now have an array of the indices that make up this ring.

now we will add the first ‘section’ of our input array. before we go to the function, we have to do 1 thing: determine if this is a benzene ring. To do this, we just check if that “token behind the number index” is ‘c’ or not. if it is ‘c’, it is part of the benzene ring

REFER TO TASK 1d HERE

1 thing to note here is that TASK 1d do not fix to the atom outer connection but rather just set it to [0, 0, 0] because to truly inspect all outer connection we need all sections be finished

after the i loop from 1 to infinite and breaks, we know now that there are no ring left. We also have the updated properties matrix.

Now that we have mapped out all of the ring parts, we will now try to map all the unmapped section (they are guaranteed to not be ring)

do a while loop without condition that does the following

trace through the properties matrix and find the first row with the first index that is not ‘X’

now that we find one element that is not X, we can proceed to map this element to its section. To do this, first we have to use BFS to find out which non ‘X’ element is still there and is connected to this node

REFER TO TASK 1f HERE

now that we have the index array, we can do a similar thing to task 1d but we will create another task so we can twist it a little bit

REFER TO TASK 1g HERE

if we can’t find any, break out of the while loop as we have successfully mapped every single atoms to a section

now that we have the section mapped out, we repeat this process untill all sections are successfully mapped out, then we can move on to our last section of this mapping: map the outer atom

REFER TO TASK 1h HERE

after, our total mapping scheme is complete and we will only return the mapping.

TASK 1a:

Create a function:

let’s try tracing the smiles code. we will try and map the ring nodes into our output matrix first. To do this:

First, the possiblities we are working with in the smiles are: [,] , (,) , C , c , Cl , O , N , NH, S, I, Br, =, #, 1, 2, more numbers if needed. We can ignore all of the []() and = # bonds because we can get them from the connectivity matrix. create a matrix of these Tokens except the ignored. Perks of this:

1. We can easily differentiate between Cl and C, as well as the other 2 characters atom can just now be 1 token
2. C and c are differentiable can be used as the key for telling which is benzene ring
3. N and NH are differentiated
4. the token 1, 2, etc. are used to determine the start of a ring

then we will know for sure the letter before it is the correct atom

input: Smiles string: c1c(CC)cccc1

output: Smiles tokens: [‘c’, ‘1’, ‘c’, ‘C’, ‘C’, ‘c’, ‘c’, ‘c’, ‘c’, ‘1’]

TASK 1b:

input: int index, array smiles tokens

ex: 2, [‘c’, ‘1’, ‘c’, ‘C’, ‘C’, ‘c’, ‘c’, ‘c’, ‘c’, ‘1’]

output: 1

we want to find the actual index, ignoring the indexes with value that is a number

this can be done by counting the index of the tokens in the token strings, skipping over numbers.

TASK 1c:

create a function called find_ring

input: int index, matrix of int

We will use breadth first search to find a path that go to exactly 4 or 5 different nodes before going back to the original node.

EX: index = 0, matrix =

[[0 1 0 0 1]

[1 0 1 0 0]

[0 1 0 1 0]

[0 0 1 0 1]

[1 0 0 1 0]]

Lets say we start with index 0 here, we can see a connection between index 0 and 1 (as long as the number is not 0, there is some connection). One possible path here is

0 -> 1 -> 2 -> 3 -> 4 -> 0

This path go over exactly 4 nodes which is perfect.

Note: We are guaranteed there will be exactly 1 path that goes through 4 or 5 different nodes before returning.

we will return all of these indices

return value: [0, 1, 2, 3, 4] (does not have to be in order)

TASK 1d:

Create a function call append_ring_section

input: index array, connectivity matrix, properties matrix, string tokens, isbenzene, solution array to task 1

Example: index array: [0, 1, 2, 3, 4, 5]

connectivity matrix:

[[0 1 0 0 0 1]

[1 0 1 0 0 0]

[0 1 0 1 0 0]

[0 0 1 0 1 0]

[0 0 0 1 0 1]

[1 0 0 0 1 0]]

property matrix

(Element, isRing, isEdge):

[['C', True, False]

['C', True, False]

['C', True, False]

['C', True, False]

['C', True, False]

['C', True, False]]

Smiles tokens: [‘c’, ‘1’, ‘c’, ‘C’, ‘C’, ‘c’, ‘c’, ‘c’, ‘c’, ‘1’]

isbenzene = true

Let’s review how the solution array should look like:

an array(the whole molecule) of arrays(each section, e.g. a ring is a section) of arrays(each atoms) of elements

The most inner arrays is composed of the form:

[index, string element, int ring_status, array of outer connection, array of inner connection, bool isedge]

Explain of the algorithm:

index: This is strictly the index of this atom in the properties matrix (use will be explain later)

String element, isedge: use the smiles token

int ring_status: 0 for no ring, 1 for is ring, 2 for is benzene ring

array of outer connection: this array is going to be of the form: [‘index of the section’, ‘index of the atom of that section’, ‘numbers of bond’]

array of inner connection: this array is going to be composed of n tuples of the form: (‘index of connected atom’, ‘numbers of bond’)

so we dont know how many sections are made already but we can always append a new section with n arrays based on the length of the index array given to us. for each of the inner array we will append everything we know about this atom:

we will append an empty array to solution array to task 1

for each index in the index array, we will do the following:

index = index

string element = DONE IN TASK 1e

int ring_status, 1 if isbenzene is false and 2 if isbenzene is true

array of outer connection: set it as [0, 0, 0] for now

array of inner connection: we will now check the connectivity row index index:

for every element that is not 0:

value = element

check the column index of that element

check if that index is found in the index array: if not continue, if it is found what index of that index value is it in the index array

add a tuple (index, value) to the array of inner connection

bool isedge = check the index 2 of the row index index in the property matrix

put this all into an array [index, string element, int ring_status, array of outer connection, array of inner connection, bool isedge]

append this into the empty section made above

now we will also have to create a way to mark if a node is already mapped into a section!

this is done be changing the property matrix and change all of the atom (column index 0) on all of the rows that indices are found in the index array (e.g. [0, 1, 2, 3, 4, 5]), lets change it to X to denote the node is already used

return the new solution array to task 1 with a whole section finished (except for the default value of (0,0,0) in the atom outer connection which we will fix later) and the updated properties matrix.

TASK 1e:

input index, smiles token arrays

example : index 2, Smiles tokens: [‘c’, ‘1’, ‘c’, ‘C’, ‘C’, ‘c’, ‘c’, ‘c’, ‘c’, ‘1’]

return the element index shown but skipping over the numbers

so the solution here is ‘C’, not ‘c’, because after skipping all the numbers. ‘C’ is the new index 2

TASK 1f:

input: index, properties matrix, connectivity matrix

output an array of index of node that is connected to the index, including the index, that is not ‘X’ in the property matrix

EX: index: 0

properties matrix:

['C', True, False]

['C', True, False]

['X', True, False]

['C', True, False]

['C', True, False]

['X', True, False]

connectivity matrix

[[0 1 0 0 0 1]

[1 0 1 0 0 0]

[0 1 0 1 0 0]

[0 0 1 0 1 0]

[0 0 0 1 0 1]

[1 0 0 0 1 0]]

output [0, 1]

what does this mean: we see that 0 is connected to 1 and 5 and 1 is connected to 0 and 2, but 5 and 2 are both X so even when 3 and 4 are not X, they can't be connected to 0 and 1. We can use some type of BFS here.

TASK 1g:

Create a function call append_non_ring_section

input: index array, connectivity matrix, properties matrix, string tokens, solution array to task 1

Example: index array: [0, 1, 2, 3, 4, 5]

connectivity matrix:

[[0 1 0 0 0 1]

[1 0 1 0 0 0]

[0 1 0 1 0 0]

[0 0 1 0 1 0]

[0 0 0 1 0 1]

[1 0 0 0 1 0]]

property matrix

(Element, isRing, isEdge):

[['C', True, False]

['C', True, False]

['C', True, False]

['C', True, False]

['C', True, False]

['C', True, False]]

Smiles tokens: [‘c’, ‘1’, ‘c’, ‘C’, ‘C’, ‘c’, ‘c’, ‘c’, ‘c’, ‘1’]

isbenzene = true

Let's review how the solution array should look like:

an array(the whole molecule) of arrays(each section, e.g. a ring is a section) of arrays(each atoms) of elements

The most inner arrays is composed of the form:

[index, string element, int ring_status, array of outer connection, array of inner connection, bool isedge]

Explain of the algorithm:

index: This is strictly the index of this atom in the properties matrix (use will be explain later)

String element, isedge: use the smiles token

int ring_status: 0 for no ring, 1 for is ring, 2 for is benzene ring

array of outer connection: this array is going to be of the form: [‘index of the section’, ‘index of the atom of that section’, ‘numbers of bond’]

array of inner connection: this array is going to be composed of n tuples of the form: (‘index of connected atom’, ‘numbers of bond’)

so we dont know how many sections are made already but we can always append a new section with n arrays based on the length of the index array given to us. for each of the inner array we will append everything we know about this atom:

we will append an empty array to solution array to task 1

for each index in the index array, we will do the following:

index = index

string element = DONE IN TASK 1e

int ring_status, 0

array of outer connection: set it as [0, 0, 0] for now

array of inner connection: we will now check the connectivity row index index:

for every element that is not 0:

value = element

check the column index of that element

check if that index is found in the index array: if not continue, if it is found what index of that index value is it in the index array

add a tuple (index, value) to the array of inner connection

bool isedge = check the index 2 of the row index index in the property matrix put this all into an array [index, string element, int ring_status, array of outer connection, array of inner connection, bool isedge]

append this into the empty section made above

now we will also have to create a way to mark if a node is already mapped into a section!

this is done be changing the property matrix and change all of the atom (column index 0) on all of the rows that indices are found in the index array (e.g. [0, 1, 2, 3, 4, 5]), lets change it to X to denote the node is already used

return the new solution array to task 1 with a whole section finished (except for the default value of (0,0,0) in the atom outer connection which we will fix later) and the updated properties matrix.

TASK 1h:

GOAL: assignment the foreign connection properly

input: connectivity matrix, solution array to task 1

output: updated solution array to task 1 with all of the array of outer connection tuples changed accordingly

how to do this:

for i in range length array

for j in range length array

we will use indexing because we will need to access the index

check the array[i][j][0] for the index of that atom:

find the row at the connectivity table

for each element in that row: if it is 0 continue

if it not 0, value = value, column = column index

trace the rest of that section to see if any of the array[i][*][0] contain this value, if it is then dont do anything. But no index inside the section has this index, we will find this index elsewhere

do the most efficient algorithm that find this index in any array[*][*][0]. if this index is not found throw and error as something is definitely wrong with the code: you can comment of that error as foreign index not found anywhere

if the index is found: change array[i][j][3] = (i, j, value)

return the updated solution array to task 1