CISS450: Artificial Intelligence
Assignment 2

OBJECTIVES

1. Python object-oriented programming
2. Data structures

Q1. [Data structures: doubly linked list]

Write a doubly linked list class `DLList`. There must be an accompanying `DLNode` class – here's the skeleton which is almost completed:

```
class DLNode:

    def __init__(self, value=None,
                       next=None, prev=None,
                       is_sentinel=False):

        # To make things simple, I'm using sentinel nodes. Also,
        # I do not use a separate class for sentinel nodes. I simply
        # include a variable __is_sentinel to tell me if the node is
        # a sentinel node.

        self.__value = value
        self.__next = next
        self.__prev = prev
        self.__is_sentinel = is_sentinel

    def get_next(self):
        return self.__next

    def set_next(self, next):
        self.__next = next

    def get_prev(self):
        # ***** TO BE COMPLETED *****
        pass

    def set_prev(self, prev):
        # ***** TO BE COMPLETE *****
        pass

    def get_value(self):
        return self.__value

    def get_is_sentinel(self):
        return self.__is_sentinel

    # Add properties prev, next_, is_sentinel, value.
    # Use next_ to avoid confusion with the next keyword.

    def __repr__(self):
        return "<DLNode %s value:%s, prev:%s, next:%s>" % (id(self),
                                                           self.__value,
                                                           id(self.__prev),
                                                           id(self.__next))
```

```
    def __str__(self):
        return "%s" % self.__value

    def __eq__(self, node):
        # ***** TO BE COMPLETED *****
        # Returns true if self.__value and node.__value are the same
        pass
```

The `DLList` class must contain the following methods which must be implemented (except the those labeled OPTIONAL). Description is provided where necessary (most of them are obvious.).

- `__init__`. After `xs = DLList()`, `xs` will be an empty doubly linked list. After `xs = DLList([1, 2, 3])`, `xs` will be a doubly linked list with 3 values where `1` is the value at the head and `3` is the value at the tail.
- `__len__`. This returns the number of values in the list. The runtime must be $O(1)$.
- `__eq__`. Two `DLList` objects are the same if the values in the linked lists are the same in the same order (of course).
- `get_list`. Returns a Python list of values from the object's values.
- `__str__`. See the sample run output below.
- `__repr__`. See the sample run output below.
- `is_empty`. Returns `True` exactly when the list is empty, i.e. no values.
- `clear`. This will remove all the nodes from the linked list. Note that sentinels are of course still present if you want to use sentinels.
- `insert_head`. If `xs` is a double linked list object, calling `xs.insert_head(value)` creates a node with the given value and attach the node as the new head node of `xs`.
- `delete_head`. Removes the head node. The value in the node that is removed is returned. If the doubly linked list is empty, `None` is returned.
- `insert_tail`. Similar to `insert_head` but occurs at the tail.
- `delete_tail`. Similar to `delete_head` but occurs at the tail.
- `get_head`. Return the value at the head node. If the doubly linked list is empty, `None` is returned.
- `get_tail`. Return the value at the head node. If the doubly linked list is empty, `None` is returned.
- `find`: If `xs` is a double linked list, calling `xs.find(value)` will return a reference to the node containing the given value. The search goes from head to tail. `None` is returned if the value is not found.
- `insert_before`. Calling `xs.insert_before(key, node)` will insert key before the given node `node`.
- `insert_after`. Calling `xs.insert_after(key, node)` will insert key after the given node `node`.

You should also add the following properties:

- `head`: The value at the head node or `None` if the list is empty.
- `tail`: The value at the tail node or `None` if the list is empty.

Here's a skeleton:

```
class DLList:

    def __init__(self):
        head_sentinel = DLNode(is_sentinel=True)
        tail_sentinel = DLNode(is_sentinel=True)
        head_sentinel.set_next(tail_sentinel)
        tail_sentinel.set_prev(head_sentinel)
        self.__tail_sentinel = tail_sentinel
        self.__head_sentinel = head_sentinel
```

Here are some sample runs. The following

```
xs = DLList([1, 2, 3])
print(repr(xs))
```

produces this output:

```
[1, 2, 3]
<DLList 140171100698640 [<DLNode 140171100320400 value:1, prev:140171100698
576, next:140171100320336>, <DLNode 140171100320336 value:2, prev:140171100
698576, next:140171100079184>, <DLNode 140171100079184 value:3, prev:140171
100698576, next:140171100698704>]>
```

(The long numeric strings are addresses from `id`.)

The following

```
xs = DLList([1, 2, 3])
ys = xs.get_list()
print(ys, type(ys))
```

produces this output:

```
[1, 2, 3] <class 'list'>
```

The following

```
xs = DLList(); print(xs.head, xs, xs.tail, len(xs), xs.is_empty())
xs.insert_head(5); print(xs.head, xs, xs.tail, len(xs), xs.is_empty())
xs.insert_head(2); print(xs.head, xs, xs.tail, len(xs), xs.is_empty())
```

```
xs.insert_tail(6); print(xs.head, xs, len(xs), xs.is_empty())
xs.head = 1234; print(xs.head, xs, xs.tail, len(xs), xs.is_empty())
xs.tail = 5678; print(xs.head, xs, xs.tail, len(xs), xs.is_empty())
x = xs.delete_head()
print(x)
print(xs.head, xs, xs.tail, len(xs), xs.is_empty())
x = xs.delete_tail()
print(x)
print(xs.head, xs, xs.tail, len(xs), xs.is_empty())
```

produces the following output

```
None [] None 0 True
5 [5] 5 1 False
2 [2, 5] 5 2 False
2 [2, 5, 6] 6 3 False
1234 [1234, 5, 6] 6 3 False
1234 [1234, 5, 5678] 5678 3 False
1234
5 [5, 5678] 5678 2 False
5678
5 [5] 5 1 False
```

The following

```
xs = DLList([1, 2, 3])
xs.clear()
print(xs.head, xs, xs.tail, len(xs), xs.is_empty())
```

produces the following output

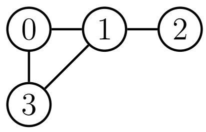
```
None [] None 0 True
```

The above sample runs are not meant to be a complete test of your code. You should test your code thoroughly with your own test cases.

(For self-study, you can also look at the documentation for `collections.deque` and implement all the methods in your `DLList` class.)

Q2. Data structures: Graph. Graph using sets.

The graph coloring problem is a very famous problem in CS (and math). The following is an undirected graph:



Mathematically, the above graph can be described by

$$
\begin{aligned}
V &= \{0, 1, 2, 3\} \\
E &= \{\{0, 1\}, \{0, 3\}, \{2, 1\}, \{3, 1\}\} \\
G &= (V, E)
\end{aligned}
$$

For the edge joining node 0 and node 1, we wrote $\{0, 1\}$, i.e., it's a set. This tells us that the edge is $\{0, 1\}$ which is the same as $\{1, 0\}$, i.e., "you can go from 0 to 1" and "you can go from 1 to 0".

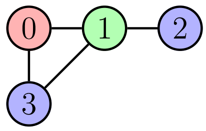
In Python, the above can be described by:

```
V = {0, 1, 2, 3}
E = {frozenset([0, 1]), frozenset([0, 3]), frozenset([2, 1]), frozenset([3, 1]}
```

and `G` is an object with instance variables `V` and `E` above as vertex set and edge set. That's why I mean when I say we will be implementing the graph with sets – python sets. (Question: Why did I use `frozenset` for $\{0, 1\}$ instead of the usual python `set`?) Note that in Python3 an example of a set of integers 1, 2, 3 is `set([1, 2, 3])`. Another way to do that in Python3 is `{1, 2, 3}`:

```
>>> X = set([1,2,3])
>>> Y = {1,2,3}
>>> print(X, type(X))
{1, 2, 3} <class 'set'>
>>> print(Y, type(Y))
{1, 2, 3} <class 'set'>
>>> print(X == Y)
True
```

A **coloring** of a graph is just an assignment of colors to the nodes of the graph such that adjacent nodes are colored differently. Here's a coloring that uses three colors:



Recall that python sets (`set` or `frozenset` are implemented using python's dictionaries which are hashtables). This means the membership check (i.e., $\in$) has a runtime of $\Theta(1)$.

Colorings are usually described as functions and in code, and functions are usually implemented using hashtables (i.e., dictionaries in the case of python). For instance, the above coloring is essentially the same as this function:

$$\begin{aligned}\{0,1,2,3\} &\to \{\textsc{Red}, \textsc{Green}, \textsc{Blue}\}\\ 0 &\mapsto \textsc{Red}\\ 1 &\mapsto \textsc{Green}\\ 2 &\mapsto \textsc{Blue}\\ 3 &\mapsto \textsc{Blue}\end{aligned}$$

However the following is not a valid coloring

$$\begin{aligned}\{0,1,2,3\} &\to \{\textsc{Red}, \textsc{Green}, \textsc{Blue}\}\\ 0 &\mapsto \textsc{Red}\\ 1 &\mapsto \textsc{Green}\\ 2 &\mapsto \textsc{Blue}\\ 3 &\mapsto \textsc{Green}\end{aligned}$$

because nodes 1 and 3 are adjacent but have been assigned the same color.

In generally, a coloring is valid exactly when adjacent nodes/vertices are colored differently.

The goal is to write a function

```
is_coloring(G, c)
```

that tests if the color assignment `c` is indeed a valid coloring for graph G. Here, the color assignment is a dictionary. For instance the following color assignment in mathematical notation

$$\begin{aligned}\{0,1,2,3\} &\to \{\textsc{Red}, \textsc{Green}, \textsc{Blue}\}\\ 0 &\mapsto \textsc{Red}\\ 1 &\mapsto \textsc{Green}\\ 2 &\mapsto \textsc{Blue}\\ 3 &\mapsto \textsc{Blue}\end{aligned}$$

is described in Python by

```
c = {0:'RED', 1:'GREEN', 2:'BLUE', 3:'BLUE'}
```

At this point, the graph coloring problem is a “very difficult problem”, i.e., the best runtime is exponential. To make this absolutely precise, one would use theory of automata and complexity theory to say that the graph coloring problem is an NP-complete problem, or that the graph coloring problem belongs to the NP-complete complexity class.

The graph coloring has many extremely important applications to all kinds of resource allocation problems. For instance if you have a collection of tasks and some of these tasks cannot share a resource, you can use recast the problem as a graph coloring problem. This occurs a lot. For instance: classroom assignment problem to classes (number of classes more than number of classrooms), CPU register allocations during compiler optimization (number of variables stored in memory more than the number of registers), radio frequency channels allocation to transmitter (number of transmitter more than available radio frequency channels), etc.

IMPLEMENTATION OF GRAPH CLASS USING SETS: VERTEX SET

First here's the `VertexSet` class. The vertex set is implemented as a set for fast searches. There are many ways to implement edge set. From CISS350, the adjacency data (i.e., whether a node is adjacent to another by an edge) can be represented by an adjacency matrix or an adjacency linked list. Also, if there are very few edges and the vertex set is huge, the adjacency matrix is sparse and therefore the edges can be as a hashtable, i.e., using a python dictionary.

```
# File: VertexSet.py

class VertexSet:
    def __init__(self, V):
        '''
        V      - list/tuple/set/frozenset of nodes
        self.V - set of nodes with values from V
        '''
        self.V = set(V)

    def __contains__(self, i):
        return i in self.V

    def __iter__(self):
        for _ in self.V:
            yield _

    def __str__(self):
        xs = list(self.V)
        xs.sort()
        s = ', '.join([str(x) for x in xs])
        return '{%s}' % s


if __name__ == '__main__':
    V = VertexSet([1,2,3])
    print(V)
    print(1 in V)    # 1 in V same as V.__contains__(1)
    print(4 in V)
    print("v0" in V)
    for _ in V:
        print(_, type(_))
```

Stare at that `__iter__` method. I'll come back to that later.

Next, we have to implement the adjacency information, i.e., the edges. For the first implementation, we'll following the mathematical definition of graphs, i.e., the edges form a set and we'll use Python's set, which is implemented using Python distoinaries (i.e., hashtables). Each edge $\{v, v'\}$ is also a set, i.e., $\{v, v'\} = \{v', v\}$. Again following the mathematical definition of an edge, we'll implement an edge as a set (of two values). Therefore the edges form a set of sets. For each edge, we will use a frozensets of two vertices. A frozenset is just a Python set except that the values in this set cannot be changed – it's the immutable version of Python sets.

IMPLEMENTATION OF GRAPH CLASS USING SETS: EDGE SET

Here's the base class for edges:

```
class Edges:

    def __init__(self):
        pass

    def __contains__(self, u, v):
        raise NotImplementedError

    def __iter__(self):
        raise NotImplementedError

    def __str__(self):
        raise NotImplementedError
```

Note that some methods simply throw the `NotImplementedError` exception. The purpose is to force subclasses to override these methods. For instance if you create an object `x` from class `Edges` and execute `print(x)`, you will get the `NotImplementedError`. However if `X` is a subclass of `Edges` and `X` implements `__str__` by returning a string, then if you call `print(x)`, you will get a string. Therefore having a method throw the `NotImplementedError` effectively makes that method pure in the sense of C++. Therefore the above `Edges` class is in some sense an abstract base class in the sense of C++. (But the two are not the same. What exactly is the difference between a python class with at least one method that throws a `NotImplementedError` and a C++abstract base class?)

Here's the implementation of a collection of edges using a set:

```
import Edges

class SetEdges(Edges.Edges):

    def __init__(self, E):
        Edges.Edges.__init__(self)
        self.E = set()
        for e in E:
            self.E.add(frozenset(e))

    def __contains__(self, u, v):
        return frozenset([u, v]) in self.E

    def __iter__(self):
```

```
        for _ in self.E:
            yield _

    def __str__(self):
        xs = [list(e) for e in self.E]
        xs.sort()
        return '{%s}' % (', '.join(['{%s, %s}' % tuple(e) for e in xs]))


if __name__ == '__main__':
    E = SetEdges([[3, 1], [2, 1]])
    print(E, type(E))
    print(str(E), type(E))
    for e in E:
        print(e, type(e))
```

Again, stare at that `__iter__` method, and also look at the `yield`.

Generators and iterables

This is a quick introduction on python generators and python iterables.

Try this:

```
for x in [5, 6, 7]:
    print(x)
```

No surprises. If that list `[5, 6, 7]` is huge, then considerable time is spent created the python list before you even begin the iteration. Here's an example:

```
for x in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]:
    print(x)
```

but say the list goes up to 999999.

Now try this:

```
def gen():
    i = 0
    while i < 1000000:
        yield i
        i += 1

for x in gen():
    print(x)
    input("press enter ... ")
```

The for-loop will give you an iteration of `x` running from `0` to `999999` without the overhead of creating a list with values from 0 to `999999`. Note that this explains the difference between python 2 of

```
for x in range(1000000):
    print(x)
```

where `range(1000000)` will create a list of values from 0 to 999999 and python 3 of

```
for x in range(1000000):
    print(x)
```

where `range(1000000)` is a generator.

An object `obj` can have a similar behavior. These are called **iterables**. The usage code looks like this:

```
for x in obj:
    ... do something with x ...
```

In the above, the `x` is called an **iterator**.

The following example will give you an iterable `obj` and an iterator `x` will obtain values `3, 4, 5` from `obj`:

```
class X:

    def __init__(self):
        self.it = None

    def __iter__(self):
        self.it = 3
        return self

    def __next__(self):
        if self.it == 6:
            self.it = None
            raise StopIteration
        else:
            x = self.it
            self.it += 1
            return x

obj = X()
for x in obj:
    print(x)
```

IMPLEMENTATION OF GRAPH CLASS USING SETS: GRAPHS

Now we can create a graph class that uses the `SetEdges` class. First the base graph class `Graph`:

```
# File: Graph.py

class Graph:

    def __init__(self):
        pass

    def is_node(self, i):
        raise NotImplementedError

    def is_edge(self, i, j):
        raise NotImplementedError

    def is_adj(self, i, j):
        raise NotImplementedError

    def __str__(self):
        raise NotImplementedError
```

Now for the graph class that uses `SetEdges`:

```
import Graph
import VertexSet
import SetEdges

class SetGraph(Graph.Graph):
    '''
    If G is a graph object, then
    G.E -- set of edges where each edge is a frozenset of two nodes
    '''
    def __init__(self, V, E):
        Graph.Graph.__init__(self)
        self.__V = VertexSet.VertexSet(V)
        self.__E = SetEdges.SetEdges(E)

    def get_V(self):
        return self.__V
    V = property(get_V, None)
```

```
    def get_E(self):
        return self.__E
    E = property(get_E, None)

    def is_node(self, i):
        return i in self.__V

    def is_edge(self, i, j):
        return frozenset([i,j]) in self.__E

    def is_adj(self, i, j):
        return self.is_edge(i, j)

    def __str__(self):
        return '<SetGraph V=%s, E=%s>' % (str(self.__V),
                                          str(self.__E))


if __name__ == '__main__':
    V = [1, 2, 3]
    E = [(1, 2), (1, 3)]
    G = SetGraph(V, E)
    print("G:", G)
    for v in G.V:
        print(v, type(v))
    for e in G.E:
        print(e, type(e))
```

And now we can do this if you have the `is_coloring` function:

```
import SetGraph
V = [0, 1, 2, 3]
E = [(0, 1), (0, 3), (2, 1), (3, 1)]
G = SetGraph.SetGraph(V, E)
c = {0:'RED', 1:'GREEN', 2:'BLUE', 3:'BLUE'}
print(is_coloring(G, c))
```

And finally, here's the `main.py` where you have to provide the `is_coloring` function:

```
# File: main.py

import SetGraph
```

```
# define is_coloring function here

if __name__ == '__main__':
    n = int(input("number of nodes: "))
    V = list(range(n))
    E = []
    while 1:
        i = int(input())
        if i == -1: # stop entering edges if input is -1
            break
        j = int(input())
        E.append((i, j))
    G = SetGraph.SetGraph(V, E)

    c = []
    for i in V:
        color = input('color for %s: ' % i)
        c.append((i, color))

    print(is_coloring(G, c))
```

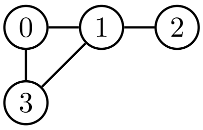
Study the above example carefully.

NOTE.

- A more interesting question is to find a valid coloring when given a list of colors. And of course you usually want to find a coloring with the least number of colors. But like I said the graph coloring problem is NP-complete. So it's a very difficult problem to solve efficiently. Of course it's trivial to solve it if you don't care about runtime – just iterate over all coloring and check which ones are valid. But that's brute force and the runtime is going to be extremely slow. (If there are V notes and c colors, what is the runtime?)

Q3. [Data structures: Graph. Graph using adjacency matrix]

This is the same as the previous question except that the new graph class here is implemented using an adjacency matrix. As an example, for the following graph,



the adjacency matrix is given by

	0	1	2	3
0	0	1	0	1
1	1	0	1	1
2	0	1	0	0
3	1	1	0	0

where the nodes i and j are adjacent exactly when the (i, j)–entry in the matrix is 1; otherwise it is 0.

Create an `AdjMatrixEdges` class in `AdjMatrixEdges.py`. (This is a subclass of the `Edges` class.) Next create an `AdjMatrixGraph` class in `AdjMatrixGraph.py`. (This is a subclass of the `Graph` class.)

Your `main.py` is similar to the `main.py` from the previous question except that you test the coloring function on a graph that is created using the `AdjMatrixGraph` class.

You must be able to iterate over all edges just like the `SetGraph`. In other words if `G` is a `AdjMatrixGraph` object, then

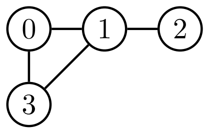
```
for e in G.E:
    i, j = e
    print(i, j)
```

You will need to study how to write an iterator.

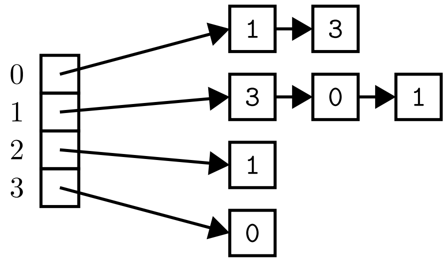
Q4. [Data structures: Graph. Graph using adjacency lists.]

This problem is similar to the previous except that the graph is implemented using adjacency lists.

Here again is the graph from the previous question:



Another way to describe a graph is to use an array of linked lists (usually singly linked lists, but doubly linked lists are OK too):



For instance you can see that at index 0, you have a linked list with values 1, 3. This represents that 0 is adjacent (i.e., joined to) to 1 and 3.

Implement the adjacency information using an array of linked list. You should use `collections.deque`.

Create a class `AdjListEdges` class in `AdjListEdges.py`. (This is a subclass of the `Edges` class.) Next create an `AdjListGraph` class in `AdjListGraph.py`. (This is a subclass of the `Graph` class.)

Your `main.py` is similar to the `main.py` from the previous question except that you test the coloring function on a graph that is created using the `AdjListGraph` class.

You must be able to iterate over all edges just like the `SetGraph`. In other words if `G` is a `AdjListGraph` object, then

```
for e in G.E:
    i, j = e
    print(i, j)
```

You will need to study how to write an iterator.

Q5. [Unique deque]

Python's deque class (`collections.deque`) is the usual double-ended queue class. Let us add one operator to it: we want to find a value in the deque. (This assume the value is unique.) For a standard deque, this would involve iterating through the whole deque – that has a runtime of $O(n)$. Make sure you study the `collections.deque` class in my notes. You might want to check python's documentation as well.

Create a class that has all the standard operations of the double linked list class (see Q1), but it supports fast search: if `dq` is a double-ended queue, `x = dq.find(val)` will return a reference (you can think of pointer) to the the node in the `dq` with value `val`. If `val` is not found, then `None` is returned.

You can modify your `DLList` class or you can use `collections.deque`. Using your own `DLList` will of course give you more control since you wrote the class yourself. Note that your doubly-linked list class `DLList` is similar to `collections.deque` in that finding a value in the container is slow. If you use `DLList`, just add a `find` method to the class. The `find` should return a reference to the node with the target value. Of course one way to verify that is to do `print(repr(dq.find(target)))` to verify you have the node.

Note again that we assume that the values in `dq` are *unique*. If you attempt to add a duplicate (using insert head or insert tail), a `DuplicateError` exception object is thrown.

The name of the class is `UniqueDeque`.

Here are some test cases for you to try out.

```
dq = UniqueDeque()
dq.insert_tail(1)
print(dq)                  # [1]
dq.insert_tail(5)
print(dq)                  # [1, 5]
dq.insert_head(6)
print(dq)                  # [6, 1, 5]
x = dq.find(1)
print(x)                   # 1
x = dq.find(42)
print(x)                   # None

try:
    dq.insert_tail(1)
except DuplicateError:
    print("duplicate")     # duplicate

print(len(dq))             # 3
x = dq.delete_tail()
print(x)                   # 5
print(dq)                  # [6, 1]
x = dq.delete_head()
print(x)                   # 6
print(dq)                  # [1]
dq.clear()
print(dq)                  # []
```

Note that you must implement the `find` so that the runtime is $O(1)$. You can easily check the runtime experimentally by adding a huge number of values into your `dq`

Note that that the find returns a reference (or pointer if you like) to the node.

This is a very important class. (As well as variations of it.) We will be using it later in AI search algorithms.

SPOILERS ON THE NEXT PAGE ...

SPOILER

The giveaway is that the search has $O(1)$ average runtime. This means right away that the search must be through a hashtable data structure. Which in the case of python means a dictionary is involved. Therefore each unique deque object must contain a Python deque object and a dictionary.