ICS 2020 Spring

Final Exam

Note: Please download the exam materials from NYU Classes. You have 2 hours to complete the whole exam. Please download the exam materials from Final exam under Assignments on NYU Classes which include the following items:

- answers.txt
- Q1_graph_student.py
- contacts.txt
- Q2_e_greedy_student.py
- Q3_help_indiana_student.py

You're allowed to use any material. Please have your answers to non-programming questions in **answers.txt**. When finished, please submit all files to NYU Classes. You are given 15 minutes to submit your answers.
(**Note:** If your submission is received after 10:15 pm (Shanghai time), you will get a 10-point-off punishment.**)**

READ EACH QUESTION CAREFULLY BEFORE YOU PROCEED WITH SOLVING IT.

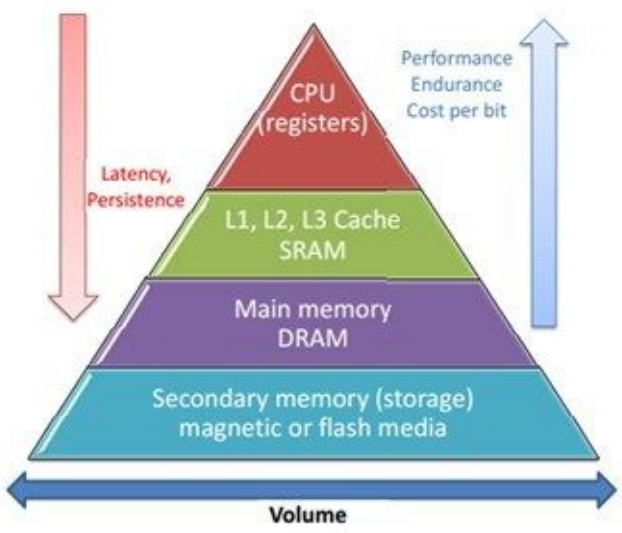
- **The Programming part has multiple questions with increasing difficulties.**
- **You do NOT have to finish one segment before moving on to the next; choose your best strategy.**

Good luck!

Part 1: Short Lecture Questions

Note: save your answers in **answers.txt**

1. The following figure shows the memory hierarchy.



In order to make use of the **temporal locality** when a computer runs a program, into which part of the memory should a recent accessed variable be put? (5 points)

A. Main memory
B. Secondary memory
C. Cache or registers
D. Main memory or secondary memory

2. What is the complexity of running `bar_chart(lst)` when `lst` is a `list` of `n` numbers (in big-O notation)? (5 points)

```
def value_count(l, n):
    """ l is a list of numbers and n is a number """
    c = 0
    for e in l:
        if e == n:
            c += 1
    return c

def bar_chart(l):
    d = {}
    copy = set(l)
    for e in copy:
        d[e] = value_count(l, e)
    return d
```

3. Walmart maintains a dataset of customer information which describe the customer's geographic attributes and purchase history. Now, based on these datasets, the company wants to apply a machine learning algorithm to identify which customer will repeat the purchase in the current month. Which method should they choose? (5 points)

A. Regression
B. Classification
C. Clustering
D. Convolutional neural network

4. Which of the following is true? (Select all that apply.) (5 points)

A. Standard deviation is a measure of the asymmetry of the probability distribution.
B. In linear regression models we can use the gradient descent algorithm to estimate the model parameters.
C. K means clustering is a supervised learning algorithm.
D. When we use K nearest neighbor classifier to predict the label a new data point, different choices of k could result in different predictions.

5. 60% of your code can be perfectly parallelized and executed on different machines (i.e., f = 0.4).

(a) How much speedup do you expect with 6 machines?(5 points)
(b) How much speedup do you expect with an infinite number of machines?(5 points)

Part 2: Programming

- The Programming part has multiple questions with increasing difficulties.
- You do **NOT** have to finish one segment before moving on to the next; choose your best strategy.

Q1. Graph (50 points)

A graph in computer science is a structure consisting of vertices and edges. Figure. 1 shows an example of a graph, where the nodes “a”, “b”, “c”, “d”, “e”, “f” are vertices and the lines connect the nodes are the edges, written as {a, b}, {a,c}, {a, d}, {b,d}, {c, d} and {e,f}.

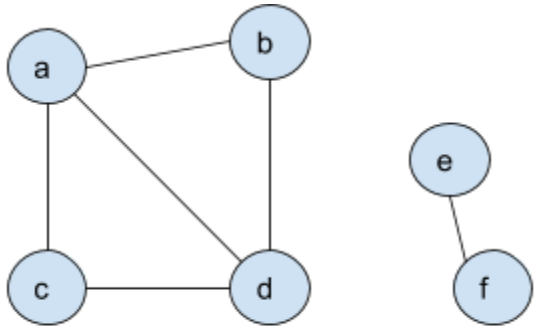


Fig. 1 An undirected graph which has 6 nodes and 6 edges.

Graphs are widely used in modeling. For example, a social network can be represented by a graph. Each person is represented by a vertice, and there is an edge between two vertices if the two corresponding persons know each other.

We want to create a class `Graph` which represents an undirected graph. In general, this class has two parts: the information about a graph, which is stored in the form of a Python dictionary, and some operations on the graph such as adding vertex or edges. Let’s take Fig.1 as an example. In the `Graph` class, the graph in Fig.1 is stored in a dictionary, like

```
fig1 = {"a":["b", "c", "d"],
        "b":["a", "d"],
        "c":["a", "d"],
        "d":["a", "b","c"],
        "e":["f"],
        "f":["e"]}
```

The keys are the nodes of the graph, and the values are lists of nodes that are connected to the keys by one edge. You need to complete the `graph_student.py`. (Note: the examples given are continued, which means the second example is based on the first one, and so on.)

(a) The Graph class contains the following methods. (5 points per method)

__init__()	The code is given. It initializes an instance of `Graph`, assigning the input graph to `self._graph_dict`. Note: you are not allowed to modify this method. `In [59]: graph_fig1 = Graph(fig1)` `In [60]: graph_fig1._graph_dict` `Out[60]:` `{'a': ['b', 'c', 'd'],` `'b': ['a', 'd'],` `'c': ['a', 'd'],` `'d': ['a', 'b', 'c', 'e'],` `'e': ['f', 'd'],` `'f': ['e']}`
vertices()	You need to implement this method. It returns a list of all vertices. `In [61]: graph_fig1.vertices()` `Out[61]: ['a', 'b', 'c', 'd', 'e', 'f']`
neighbors()	You need to implement this method. It takes a vertex as the argument and returns a list of all the vertices that connect to it by one edge. If the vertex is not in the graph, it prints "the vertex is not in the graph." and returns nothing. (see the following example). `In [11]: graph_fig1.neighbors("a")` `Out[11]: ['b', 'c', 'd']` `In [12]: graph_fig1.neighbors("z")` `z is not in the graph.`
edges()	You need to implement this method. It returns a list of all edges; each edge is a **set** of two vertices. Note: {'a', 'b'} is equal to {'b', 'a'}, and in your output, the edges may have different order to the following example, which is fine. `In [83]: graph_fig1.edges()` `Out[83]: [{'a', 'b'}, {'a', 'c'}, {'a', 'd'},` `{'b', 'd'}, {'c', 'd'}, {'e', 'f'}]`
add_vertex()	You need to implement this method. It takes a vertex as the argument and adds it into `self._graph_dict` if it is not in the graph. If the vertex has been in the graph already, it prints "the vertex has already been in the graph.".(see the following example).

	`` In [69]: graph_fig1.add_vertex("g") In [70]: graph_fig1.vertices() Out[70]: ['a', 'b', 'c', 'd', 'e', 'f', 'g'] In [71]: graph_fig1.add_vertex("a") a has already been in the graph. ```
add_edge()	You need to implement this method. It takes two vertices as arguments and adds them in to `self._graph_dict` as values to the corresponding keys. If any of the vertices is not in the graph, it will first add the vertex into the graph and then add the edge. ``` In [84]: graph_fig1.add_edge("d", "e") In [85]: graph_fig1.edges() Out[85]: [{'a', 'b'}, {'a', 'c'}, {'a', 'd'}, {'b', 'd'}, {'c', 'd'}, {'d', 'e'}, {'e', 'f'}] In [6]: # When "g" is not in the graph, In [7]: graph_fig1.add_edge("f", "g") In [8]: graph_fig1.edges() Out[8]: [{'a', 'b'}, {'a', 'c'}, {'a', 'd'}, {'b', 'd'}, {'c', 'd'}, {'d', 'e'}, {'e', 'f'}, {'f', 'g'}] ```
remove_edge()	You need to implement this method. It takes two vertices as arguments and removes the edge between them. If the edge does not exist, then it does nothing.

	`` In [22]: graph_fig1.remove_edge("f", "g") In [23]: graph_fig1.edges() Out[23]: [{'a', 'b'}, {'a', 'c'}, {'a', 'd'}, {'b', 'd'}, {'c', 'd'}, {'d', 'e'}, {'e', 'f'}] In [92]: graph_fig1.remove_edge("d", "e") In [93]: graph_fig1.edges() Out[93]: [{'a', 'b'}, {'a', 'c'}, {'a', 'd'}, {'b', 'd'}, {'c', 'd'}, {'e', 'f'}] In [31]: #When there is no edge between two vertices In [32]: graph_fig1.remove_edge("d", "g") In [33]: graph_fig1.edges() Out[33]: [{'a', 'b'}, {'a', 'c'}, {'a', 'd'}, {'b', 'd'}, {'c', 'd'}, {'e', 'f'}] ```
remove_vertex()	You need to implement this method. It takes a vertex as the argument and removes it from the graph. Note: it also removes edges containing the vertex. If the vertex is not in the graph, then it does nothing. ``` In [36]: #"g"is an isolated vertex In [37]: graph_fig1.remove_vertex('g') In [38]: graph_fig1.vertices() Out[38]: ['a', 'b', 'c', 'd', 'e', 'f'] In [39]: graph_fig1.edges() Out[39]: [{'a', 'b'}, {'a', 'c'}, {'a', 'd'}, {'b', 'd'}, {'c', 'd'}, {'e', 'f'}] ```

	`` In [40]: #"a" is connected to other vertices In [41]: graph_fig1.remove_vertex('a') In [42]: graph_fig1.vertices() Out[42]: ['b', 'c', 'd', 'e', 'f'] In [43]: graph_fig1.edges() Out[43]: [{'b', 'd'}, {'c', 'd'}, {'e', 'f'}] In [51]: #When 'z' is not in the graph In [52]: graph_fig1.remove_vertex('z') In [53]: graph_fig1.vertices() Out[53]: ['b', 'c', 'd', 'e', 'f'] ```
__str__()	The code is given. It overrides the inherited __str__() method, printing the nodes and their neighbors.

(b) Contact tracing is an effective public health measure for the control of COVID-19. Graph models are often used in this task. The file `contacts.txt` contains the records of the contacts of a local community. We want to build an undirected graph model for tracing contacts. In the graph, each member is represented as a vertice, and if two members have contacts, then, there is an edge between them. You need to write **two functions**.

(i) The first one is `load_graph()` which loads the records in `contacts.txt` and returns an instance of `Graph` class. Note: In `contacts.txt`, each line represents the contacts of a person. The first character is the person's ID, and the following characters are the IDs of persons whom he/she contacted with. (5 points)

(ii) The second one is `trace_contact()`. Given a graph **g** and vertex **v**, this function returns all vertices in **g** that are connected with **v** by edges (including **v** itself). Let's take Fig.2 as an example. Given a vertex `"b"`, the function returns `["a", "b", "c", "d"]`, and given a vertex `"e"`, it returns `["e", "f", "g", "h"]`. Such a set of vertices is also called a component. (10 points)

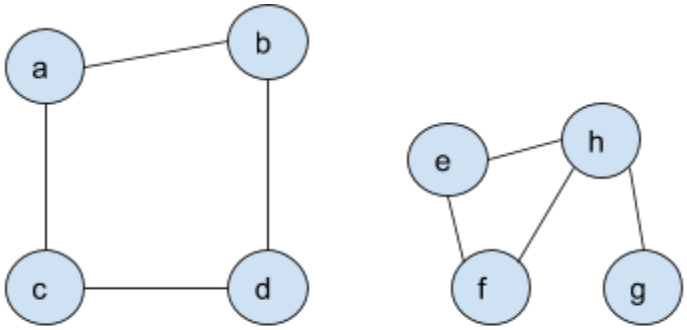


Fig. 2 An undirected graph which has 8 nodes and 8 edges.

We can trace the contacts of a member “x” by the following steps:
Step 1. Put “x” into the component.
Step 2. Find the neighbors of “x”;
Step 3. Trace the contacts of every neighbor: If a neighbor “y” is not in the component, then go to Step 1, replace “x” by “y” and go on with Step 2 to find the neighbors of “y”, and so on. Otherwise, pick another neighbor and check if it is in the component or not.
Step 4. Stop if every neighbor of “x” is traced, and now, the component is filled by all the members that connect to “x” via one (or several) edge(s).

load_graph()	You need to implement this function. It takes a filename as the argument and returns an instance of Graph class. `In [10]: graph_fig2 = load_graph("contacts.txt")` `In [11]: print(graph_fig2)` `{'a': ['b', 'c'], 'b': ['a', 'd'], 'c': ['a', 'd'], 'd': ['b', 'c'], 'e': ['f', 'h'], 'f': ['e', 'h'], 'h': ['e', 'f', 'g'], 'g': ['h']}`
trace_contact()	You need to implement this function. It takes one graph, one vertex, and an empty list (named as `component` in the starting code) as the arguments and returns the `component` which should contain all the vertices that connect to the vertex.

```
In [9]: component = []

In [10]: trace_contact(graph_fig2, "a", component)
Out[10]: ['a', 'b', 'd', 'c']

In [11]: graph_fig2.add_edge("d", "e")

In [12]: component = []

In [13]: trace_contact(graph_fig2, "a", component)
Out[13]: ['a', 'b', 'd', 'c', 'e', 'f', 'h', 'g']
```

Q2. Epsilon-greedy Ads Drop-off (5 points)

Among the most useful strategies to deal with uncertainty, one is to combine ***exploration*** and ***exploitation*** strategically. This is the core of the ε-greedy algorithm; ε is a positive value smaller than 1:

With probability ε we pick randomly, otherwise we pick the current best choice

(The reinforcement Q learning in-class exercise actually uses this strategy.)

Suppose you owned a supermarket near a community of size 5, and you want to drop off advertisement cards (Ads) of recent promotions to each house. Whether each neighbor will pick up your Ads is determined by a probability, the “degrees of interest”, which is hidden from you. In the warm-up phase that we implemented in the code, we use a 0.5 “degrees of interest” for every neighbor as the *initial but wrong* estimation.

```
+++++++ warm-up phase
0 {'pick-up': 5.0, 'drop-off': 10}
1 {'pick-up': 5.0, 'drop-off': 10}
2 {'pick-up': 5.0, 'drop-off': 10}
3 {'pick-up': 5.0, 'drop-off': 10}
4 {'pick-up': 5.0, 'drop-off': 10}
------- true degree of interests:
0 0.8444218515250481
1 0.7579544029403025
2 0.420571580830845
3 0.25891675029296335
4 0.5112747213686085
```

```
++++++++ after simulation  100000
0 {'pick-up': 77581, 'drop-off': 92008}
1 {'pick-up': 1493, 'drop-off': 1970}
2 {'pick-up': 830, 'drop-off': 2060}
3 {'pick-up': 502, 'drop-off': 2033}
4 {'pick-up': 1021, 'drop-off': 1979}
-------- estimated degree of interests:
0 0.8431984175289106
1 0.7578680203045686
2 0.4029126213592233
3 0.2469257255287752
4 0.5159171298635674
```

During each iteration, you will drop off a promotion card to ***only one*** neighbour, based on the neighbours behavior, update your data set for total picked-up Ads and total dropped-off Ads. The degree of interest is estimated as, total picked-up Ads divided by total dropped-off Ads

As a smart businessman, you not only want to estimate the degree of interest, you'd also want to deliver the Ads to neighbors with a high degree of interest. So you'll apply the ε-greedy algorithm. To do that, for every Ads card to drop off, you throw a dice and decide whether to
1) randomly deliver it to a neighbor, or
2) deliver it to the neighbor with the highest estimated degree of interest.

Complete the code in `e_greedy_student.py`. Here is the output we expect:(note: The order of the key-value pairs on your screen may **not** be the same as the following **but it is fine**. Because Python uses complex algorithms to determine where the key-value pairs are stored in a dictionary. For our purposes we can think of these orderings as unpredictable.)

Q3. Helping Indiana Jones (15 points)

Once upon a time, Indiana Jones came to a lost temple. Fortunately, he found 6 different treasures whose weights and values are listed in the following table.

Treasure	Golden Idol	Peacock's eye	Lost Ark	Holy Grail	Crystal skull	Truncheon
Weight(kg)	5	3	7	2	3	4
Value	3	6	5	4	3	4

Unfortunately, he only had one bag that could take 15 kg weights maximum (we don't consider the size of the bag) Now, you need implement functions in `help_indiana_student.py` to help Indiana to

(a) calculate the maximum value that he can take away with his bag. Please implement `calculate_max_value()` for this task. (5 points)
(b) find what treasures he should take if he wants to obtain the maximum value. Please implement `pick_items()` for this task. This function returns two values: one is the maximum value, another is a list of items he should pick. (10 points) [Hint: you may modify the function you write in (a), so that it can return the items picked as well.]

Note:

1. In the starting code, the weigh-value table is stored in a Python dictionary as follows,

```
treasures_found = {
   "Golden Idol":{"weight": 5, "value": 3},
   "Peacock's eye":{"weight": 3, "value": 6},
```

```
"Lost Ark":{"weight": 7, "value": 5},
"Holy Grail":{"weight": 2, "value": 4},
"Crystal skull":{"weight": 3, "value": 3},
"Truncheon":{"weight": 4, "value": 4}
}
```

Since Python dictionary type is mutable, please use **copy.deepcopy()** if you need to deep copy a Python dictionary. (The copy package has been imported in the starting code already.)

By the way, there are two tests in the starting code based on **different** weight-value tables. To get full points, your code should pass both of them as follows,

```
---This is test 1---
The maximum value is 18
The maximum value and items to pick are
 (18, ['Crystal skull', 'Holy Grail', 'Lost Ark', "Peacock's eye"])

---This is test 2---
The maximum value is 17
The maximum value and items to pick are
 (17, ['Truncheon', 'Holy Grail', "Peacock's eye", 'Golden Idol'])
```

2. You can write anything inside `calculate_max_value()` and `pick_items()`, even define some other functions inside them if needed. But you need to make sure that the name of the returned variable(s) is the same as that of the given starting code so that the tests can run properly.