CS 380/7813 Python Programming, Fall 2012
Homework 3: Algorithmic Implementation, Cython, multiprocessing, and numpy

Professor: Liang Huang   TA: Kai Zhao

December 6, 2012

Instructions

1. This assignment is due on **Saturday 12/8** at 11:59 PM. Among the three HWs, you can only submit **one** HW late (by 48 hours) without penalty; the second late HW will receive a zero.
2. You may discuss this HW *in a high level* with another student, but you should write your partner's name in `debrief.txt`. High-level discussions include algorithmic design (but **not** how to implement them), input/output specifications, etc.
3. You are **not allowed** to use programs found on the Internet, although you are allowed to consult sites such as stackoverflow.com for specific questions and you'll have to report any help you got from the internet. Plagiarism will be caught.
4. There will be 10% for *Pythonic styles* and 20% *efficiency* in each assignment. Try to make your code more Pythonic and less like C++/Java. Also, try your best to write simple, efficient, short, self-evident, and beautiful code. If your code is better or prettier than ours, you get extra points.

1 Hashed Heap (`hheap.py`) 40%

For many famous algorithms involving priority queues (such as Dijkstra, Prim, A*, etc.), it is important to be able to change the priority of a particular key after its insertion into the priority queue. For example, Dijkstra uses a `Decrease_Key` operation when updating the estimate of a node. Unfortunately, most existing packages for Python (and other languages) do not provide this feature. Here you need to implement a hashed binary heap which efficiently handles priority changes.

Here is the skeleton:

```
class hheap(dict):

    @staticmethod
    def _parent(i): # please use bit operation (same below)!
        ...

    @staticmethod
    def _left(i):
        ...

    @staticmethod
    def _right(i):
        ...
    def __init__(self):
        self.heap = []
        dict.__init__(self)

    def __setitem__(self, key, value):
        """ either insert or change the value for a key """
        ...

    def __getitem__(self, key):
        ...

    def pop(self): # pop root (best)
        ...

    def update_if_better(self, key, newvalue):
        """update if newvalue is better than the current value for key
           or insert if key is not here yet."""
        ...

    def _swap(self, i, j):
        """swap the contents b/w indices i and j; update hash accordingly"""
        ...

    def __str__(self):
        """print like a dict, but in order of value"""
        ...
```

You should be able to test it like this:

```
>>> from hheap import hheap
>>> d=hheap()
>>> d["b"] = 2
>>> d["c"] = 1
>>> d["b"] = 0
>>> d.pop()
('b', 0)
>>> d["c"]
1
>>> d["b"]
KeyError!
>>> d["e"] = 4
>>> d.update_if_better("e", 0.5)
>>> d.update_if_better("c", 0.2)
>>> d.update_if_better("c", 0.3)
>>> d.update_if_better("f", 0.3)
>>> d
{'c': 0.2, 'f': 0.3, 'e': 0.5}
>>> d.pop()
('c', 0.2)
```

2 Dijkstra's Shortest Path (`dijkstra.py`) 30%

Let us build a high-speed railroad network in the United States, so that traveling from NYC to DC becomes only 2 hours! To achieve that *noble goal*, we have to plan a railroad network and calculate the time between cities, for which we need to solve the shortest path problem. To make our lives easier we only care about the distances from a home city to all other cities (called "single-source shortest path problem").

Dijkstra's single-source shortest path algorithm is one of the corner stones in computer science. Implement it for a given undirected graph using the hashed priority queue you implemented above. Make sure the time complexity of Dijkstra is $O((n+m)\log n)$ where n and m are the number of nodes and edges, respectively.

The input (`sys.stdin`) will start with a line of cities separated by white spaces; the first city on that line is considered **home city**. Each of the following lines is a triple A B d, where A and B are two cities and d is their distance (hours on planned high-speed rail). The distance is symmetric (i.e., undirected graph).

The output (**standard output**) contains n lines where n is the number of cities. Each line starts with the city name, followed by its shortest distance from the home city, and the neighoring city that gives this distance (e.g. `via Dallas`). If a city is unreachable from the source, print `unreachable`. Important: the order of lines in the output should follow an increasing order of distances (i.e., cities closer to the source should appear earlier), and whenever there is a tie, use lexicographical order.

Sample Input
NYC Chicago LA Boston DC SF Dallas Honolulu Anchorage
NYC Boston 2
NYC DC 2
NYC Chicago 7
Chicago Dallas 8
DC Dallas 10
Chicago SF 15
Chicago LA 15
SF LA 3
LA Dallas 10

Sample Output
NYC 0
Boston 2 (via NYC)
DC 2 (via NYC)
Chicago 7 (via NYC)
Dallas 12 (via DC)
LA 22 (**via Chicago**)
SF 22 (via Chicago)
Anchorage unreachable
Honolulu unreachable

3 Prim's Minimum Spanning Tree (`prim.py`) 15%

Prim's minimum spanning tree (MST) algorithm is another classic in the computer science repertoire. It is very similar to Dijkstra's algorithm above; in fact, you only need to change the concept of priority from the distance from the source (as in Dijkstra) to the distance between S and $V - S$.

The input format (`sys.stdin`) is almost identical to that of Dijkstra, except that you can now assume all cities are connected. The first city on the first line is also considered home city where you would grow your MST from (using home city as the root node). The output (**standard output**) starts with a line of the total length of the MST, followed by $n-1$ lines. Each of the following lines is a triple A B d where A and B are two cities and d is their distance. Important: the output should follow a level-order traversal of the MST, and within each level, use lexicographical order.

Sample Input
NYC Chicago LA Boston DC SF Dallas
NYC Boston 2
NYC DC 2
NYC Chicago 7
Chicago Dallas 8
DC Dallas 10
Chicago SF 15
Chicago LA 15
SF LA 3
LA Dallas 10

Sample Output
32
NYC Boston 2
NYC DC 2
NYC Chicago 7
Chicago Dallas 8
Dallas LA 10
LA SF 3

As you can see, the only differences between MST and the Dijkstra spanning tree are that MST uses edges Chicago–Dallas and LA–SF instead of DC–Dallas and Chicago–SF. This is because Chicago is closer to Dallas (preferred by MST), but it leads to a longer distance from NYC (preferred by Dijkstra), and LA is closer to SF but leads to a longer distance from NYC.

4 Cython (`rational2.pyx`, `rational3.pyx`, `results.txt`) 20%

Here is a simplified version of the Rational number class you wrote for HW2, which is a mutable type and only supports multiplication.

```
def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a%b)

class Rational(object):
    __slots__ = "nom", "denom"

    def __init__(self, nom, denom=1):
        self.nom, self.denom = nom, denom
        self.reduce()

    def __str__(self):
        if self.denom == 1:
    return str(self.nom)
        return "%s/%s" % (self.nom, self.denom)

    __repr__ = __str__

    def reduce(self):
        if self.denom == 0:
            raise ZeroDivisionError("Denominator is zero!")

        g = gcd(self.nom, self.denom)
        self.nom /= g
        self.denom /= g
        return self # so that you can chain...

    def __mul__(self, other):
        return Rational(self.nom * other.nom, self.denom * other.denom)

    def __imul__(self, other):
        self.nom *= other.nom
        self.denom *= other.denom
        return self.reduce()
```

Translate the above code into two Cython versions: `rational2.pyx` for Cython with C ints, and `rational3.pyx` for Cython with Python types.

Now you can benchmark this way:

```
import rational, rational2, rational3
import time

for x in [rational, rational3, rational2]: # slowest to fastest
```

```
t = time.time()

r = x.Rational(1,2)
for i in xrange(1, 10000):
    r *= x.Rational(i+3, i+4)
print r

print x.__name__, time.time() - t
```

Include in `results.txt` the following:

1. the benchmark results.
2. What is the relative speedups for rational2 and rational3 over rational?
3. Why are the speedups smaller than those of the Fibonacci example on the slides? Explain it by looking at `rational2.c` and/or `rational3.c`.
4. What will happen if you replace 10000 with 100000? Why this does not happen to Fibonacci?

Debriefing

Please answer these questions in `debrief.txt` and submit it along with the programs. **Note:** You get 5% off for not responding to this part.

1. Did you discuss this HW in a high level with a classmate? If so, what's his/her name?
2. What help did you get from online resources, if any?
3. How many hours did you spend on this assignment?
4. Would you rate it as easy, moderate, or difficult?
5. Are the lectures too fast, too slow, or just in the right pace?
6. Any other comments?