Scientific Computation Midterm Test Solution
Autumn 2024

There are 4 questions worth a total of 40 points. The questions are **not** equally weighted. You are allowed to use a single A4-sized sheet of paper with handwritten notes.

1. (7 points) Apply the *merge* algorithm to efficiently combine the lists `A = [5,7,8,10]` and `B = [1,2,4,6]` into a single list sorted in increasing order. The algorithm requires comparisons between elements in `A` and `B`. Provide a list of such comparisons in the order they occur. In your answer, (a, b) should indicate a comparison between an element in `A` with value a and an element in `B` with value b. Provide a 1-2 sentence explanation of how you decide which elements to compare.
   Solution: Each iteration, we compare the smallest elements in `A` and `B` that haven't already been placed in the final merged list. The comparisons are: $(5, 1), (5, 2), (5, 4), (5, 6), (7, 6)$. (Comparisons stop after $(7, 6)$ as all of the elements in `B` will have been placed in the final list at that point.)

2. (8 points) The code below generates an n-element list, `L`, and then calls `method3` m times. Analyze the average-case asymptotic time complexity of lines 20-27. You should provide an estimate of the big-O cost along with an explanation of how you obtained this estimate. A line-by-line operation count is not required though relatively important parts of the code should be discussed. You **do not** need to analyze the cost of using `np.random.randint` in line 21.

```python
1  def method3(L,x,flag=True):
2      if flag:
3          X = {}
4          for i,l in enumerate(L):
5              X[l] = i
6          if x in X:
7              return X[x],X
8          else:
9              return -1000,X
10     else: #flag=False
11         if x in L:
12             return L[x]
13         else:
14             return -1000
15     return -1000
16
17 n = 200; m = 40
18 L = list(np.random.randint(0,10**5,n)) #create list of n integers between 0 and 10**5
19 results = []
20 for i in range(m):
21     x = np.random.randint(0,1000) #generate random integer between 0 and 1000
22     if i==0:
23         ind,X = method3(L,x,flag=True)
24         results.append(ind)
```

```python
    else:
        ind = method3(X,x,flag=False)
        results.append(ind)
```

Solution: During the first iteration an n-element dictionary is created ($\mathcal{O}(n)$ cost). During each of the m iterations, there are at most 2 dictionary lookups ($\mathcal{O}(1)$ cost), there is an append to a list ($\mathcal{O}(1)$), and there are a few other miscellaneous $O(1)$ operations (two comparisons, `i` is incremented, `ind` is assigned...). Aside from the first iteration, the cost per iteration does not depend on n, so the total cost is $\mathcal{O}(n+m)$.

3. The function `func3` below looks for paths between node s and other nodes in a weighted undirected NetworkX graph and returns `dmin`, a "distance" between nodes s and x, if at least one path between s and x exists.

```python
def func3(G,s,x):
    """Input:
    G: weighted NetworkX graph with n nodes (numbered as 0,1,...,n-1)
    s: an integer corresponding to a node in G
    x: an integer corresponding to a node in G
    """
    n = len(G)
    dinit = np.inf
    Fdict = {}
    Mdict = {}
    Mdict[s]=1

    while len(Mdict)>0:
        dmin = dinit
        for n,delta in Mdict.items():
            if delta<dmin:
                dmin=delta
                nmin=n
        if nmin == x:
            return dmin
        Fdict[nmin] = Mdict.pop(nmin)
        for m,en,wn in G.edges(nmin,data='weight'):
            if en in Fdict:
                pass
            elif en in Mdict:
                dcomp = dmin*wn
                if dcomp<Mdict[en]:
                    Mdict[en]=dcomp
            else:
                dcomp = dmin*wn
                Mdict[en] = dcomp

    return -1000 #no path from s to x
```

(a) (6 points) Provide a 1-2 sentence description of the general strategy this function is using, and explain how this code defines the distance between 2 nodes.
Solution: The code is using Dijkstra's method where the priority queue is managed using a dictionary (3 points). The length of a path between two nodes is the product

of the edge weights on the path. The distance is the length of a shortest path between the nodes. (This can be inferred from line 30 where the initial provisional distance for node `en` is the product of the distance for node `n` and the weight of the edge connecting `n` and `en`.) (3 points)

(b) (7 points) A constraint must be applied to the edge weights for the function to generally return valid results. Provide a clear and concise explanation of what this constraint should be. Assuming that this constraint is satisfied, explain the correctness of the part of the underlying algorithm associated with line 21 during one iteration of the algorithm. Assume that distances have been correctly assigned during all previous iterations. A rigorous proof is not needed, but the key ideas required to construct such a proof should be discussed.

Solution: The edge weights must be greater than or equal to one. Otherwise, when we finalize a node, there could be a shorter path via an un-finalized node. Line 21 corresponds to the part of Dijkstra's algorithm where a node with the smallest provisional distance is removed from the priority queue and finalized. To establish the correctness of this step, we can essentially repeat the argument presented in lecture taking care to modify them to account for the change in the definition of the distance and the edge-weight constraint. Let node x be the node being finalized and let δ_x be its provisional distance. Based on lines 14-18 this must be the minimum provisional distance in the priority queue.

Part 1: Say that x has k neighbors in F, the set of finalized nodes, with $k \geq 1$, and label these nodes as $i_1, i_2, ..., i_k$ with the subscript indicating the order in which the nodes were finalized. Dijkstra's algorithm ensures that $\delta_x = \min(d_{si_1} w_{i_1,x}, d_{si_2} w_{i_2,x}, ..., d_{si_k} w_{i_k,x})$. Each of the quantities on the RHS correspond to the lengths of paths from x to s where all nodes except x have been finalized. There can be no such path with length shorter than δ_x as this would imply that one or more of the finalized distances are incorrect.

Part 2: There can be no direct links between finalized nodes and unexplored nodes. Whenever a node is finalized, all of its neighbors are explored (lines 22-31).

Part 3: The only other possible shortest paths start at s, follow one link from a finalized node to a node in the priority queue other than x and then continues on to x. Say this link connects nodes j and y where j has been finalized. The shortest such path will have length, $l = d_{sj} w_{jy} d_{yx}$. The algorithm ensures that $\delta_y \leq d_{sj} w_{jy}$, so $l \geq \delta_y d_{yx}$. We require x to have a smallest provisional distance, so $\delta_x \leq \delta_y$, and $l \geq \delta_x d_{yx}$. Finally, our edge-weight constraint guarantees that $d_{yx} \geq 1$, so $l \geq \delta x$. The results of parts 1-3 taken together tells us there can be no paths from s to x with length smaller than δ_x and δ_x is the length of a path between the nodes, so $d_{sx} = \delta_x$.

4. Carefully read the function `func4` below along with the code underneath the function which calls it twice.

```
1  def func4(delta,Nt,tf,c):
2      """Input:
3      delta: positive real number
4      Nt: positive integer
5      tf: positive real number
6      c: positive real number
7      """
8      t = np.linspace(0,tf,Nt+1)
9      Dt = t[1]-t[0]
10
11     x = np.zeros((Nt+1))
12     y = x.copy()
13     x[0] = (1/c)*np.pi/2 + delta
14     y[0] = np.pi + delta
15
16     for i in range(Nt):
17         x[i+1] = x[i] +  Dt*(-np.sin(y[i]))
18         y[i+1] = y[i] + Dt*(np.cos(c*x[i]))
19
20     return t,x,y
21 #set parameters
22 delta = 1e-4
23 Nt = 10000
24 tf = 10
25 c = 2*np.pi
26 #call func4 twice
27 tA,xA,yA = func4(delta,Nt,tf,c)
28 tB,xB,yB = func4(delta,Nt*4,tf,c)
29 #further calculations
30 eA = np.abs(xA[-1]-xAexact[-1])
31 eB = np.abs(xB[-1]-xBexact[-1])
32 print(eB/eA) #question 4(b)
33
34 DtA = tA[1]-tA[0] #DtA = 0.001 for the parameters set above
35 k = (1/c)*np.pi/2
36 fA = (xA[Nt]-k)/(xA[Nt-ishiftA]-k)
37 print('fA=',fA) #question 4(c)
```

(a) (4 points) What is the system of ODEs that `func4` is computing a numerical solution to?
Solution: From lines 16-18, we see that this code is using the explicit Euler method to solve:

$$dx/dt = -\sin(y)$$
$$dy/dt = \cos(cx)$$

(b) (4 points) Assume that `xAexact` and `xBexact` contain exact solutions that can be compared directly to `xA` and `xB`, respectively. Provide an estimate for what will be printed by line 32. Also provide a clear and concise explanation of how you obtained your estimate. It is fine to state results from lecture or elsewhere provided the statements are clear, correct, and relevant.

Solution: The error of the explicit Euler method scales linearly with Δt (and $\to 0$ when $\Delta t \to 0$). `xB` is computed with a time step that is four times smaller than that used for `xA`, so the error should also be four times smaller. Consequently, we expect `eB/eA` $\approx 1/4$.

(c) (4 points) The variable `fA` is set in line 36. Assume that `xA` has been computed accurately, and explain how to set `ishiftA` so that `fA` ≈ 1. Note that `ishiftA` should be a positive integer that is much larger than 10.

Solution: The initial conditions are close to the following fixed point: $(\bar{x}, \bar{y}) = (\frac{1}{c}\frac{\pi}{2}, \pi)$, and since $\delta \ll 1$, we should consider the behavior of small perturbations to this fixed point. Let $x = \bar{x} + \epsilon x_1 + \mathcal{O}(\epsilon^2)$, and $y = \bar{y} + \epsilon y_1 + \mathcal{O}(\epsilon^2)$. Substituting these expansions into the ODEs gives, after some simplifications,

$$\frac{dx_1}{dt} = -\cos(\bar{y})y_1 + \mathcal{O}(\epsilon)$$
$$\frac{dy_1}{dt} = -c\sin(c\bar{x})x_1 + \mathcal{O}(\epsilon)$$

Letting $\epsilon \to 0$ then gives us a system of two constant coefficient ODEs which we can write as,

$$\frac{d\mathbf{z}}{dt} = \mathbf{A}\mathbf{z},$$

with $\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -c & 0 \end{bmatrix}$. We can look for solutions of the form $e^{\lambda t}$ and find the eigenvalues, $\lambda = \pm i\sqrt{c}$. So we expect sinusoidally oscillating behavior with period $= 2\pi/\sqrt{c}$. This means that we need `ishift`$\Delta t = \frac{2\pi}{\sqrt{c}}$, or `ishift` $= \frac{2\pi}{\Delta t\sqrt{c}}$ (it is fine to leave the solution in this form). From the code, we have `c` $= 2\pi$, and $\Delta t =$ `Dt` $= 0.001$, so `ishift` $= 1000\sqrt{2\pi}$. (3 points for analysis, 1 point for figuring out `ishift`).