Closest Pair of Points

CS 320, Fall 2017

Dr. Geri Georg, Instructor
georg@colostate.edu
Closest Pair of Points

Set of \( n \geq 2 \) points in set \( Q \); coincident points have distance of 0

CLOSE defined by Euclidean distance

Brute force: \( \binom{n}{2} = \Theta(n^2) \)

Divide and conquer: \( O(n \lg n) \)

Base case: \( |P| \leq 3 \)
\[ \mathbf{X} = [5, 6, 12, 8, 10, 3, 11, 9, 4, 7, 1, 2] \]
\[ \mathbf{Y} = [5, 2, 9, 12, 10, 4, 6, 7, 11, 3, 8, 1] \]

The line \( l \) bisects the set of points \( \mathbf{P} \).
$X_L = [5, 6, 12, 8, 10, 3] \quad X_R = [11, 9, 4, 7, 1, 2]$

$Y_L = [5, 12, 10, 6, 3, 8] \quad Y_R = [2, 9, 4, 7, 11, 1]$  

$l$ (bisects set of points $P$)  

$\delta = \min(\delta_L, \delta_R)$
\[ X_L = [5, 6, 12, 8, 10, 3] \quad X_R = [11, 9, 4, 7, 1, 2] \]
\[ Y_L = [5, 12, 10, 6, 3, 8] \quad Y_R = [2, 9, 4, 7, 11, 1] \]
\[ \delta_L = [11, 9, 4, 7, 1, 2] \quad \delta_R = [2, 9, 4, 7, 11, 1] \]

\[ \delta = \min(\delta_L, \delta_R) \]

The line \( l \) bisects the set of points \( \mathcal{P} \).

But maybe
If points are less than $\delta$ apart, then they must lie within $\delta$ of either side of the line $l$ and within $\delta$ of each other vertically.
All points in $P_L$ are at least $\delta$ apart and all points in $P_R$ are at least $\delta$ apart since $\delta = \min(\delta_L, \delta_R)$.

8 possible points around line $l$:
- must be corners of $\delta \times \delta$ square,
- 4 on the line if coincident
From $Y$, create $Y'$ with just the points in the $2\delta$–wide strip; already sorted vertically
For each point $p$ in $Y'$, calculate the distance from $p$ to the next 7 points in $Y'$.

**Answer:** closest pair from $Y'$, $P_L$, or $P_R$ as applicable.
Recap Closest Pair of Points

<table>
<thead>
<tr>
<th>Pt</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>-2</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

1. If base case, return brute force
2. Create X, Y arrays, find l, create L, R versions of arrays
3. Recurse Left, then Right
4. Check strip around line l (Y’ array)
5. Return closest pair
Expectations -1

DELIVERABLES

• A zip file of your Python3 program (and any additional files it uses) and your PDF complexity analysis discussion file. Due date TBD.

PAIR PROGRAMMING

• Both of you work with a single computer; taking turns and talking with each other so that both understand everything you are doing.

USE GIT HUB!

• Create a new private repository called cp_yourInitials_partnerInitials (e.g. cp_ae_jx) in the CS320 organization, and add your partner as a collaborator.
• Push your work to this repository multiple times.
Expectations -2

COMPLEXITY ANALYSIS

• Proper X and Y choices. Independent variable on X and dependent on Y. What is a choice for independent variable describing size that will make sense to a person not familiar with the problem?
• Description of theoretical complexity, and rationale.
• Description of your implementation, why it should meet theoretical expectations, justification of why it does or does not, and if not, why you think this is the case.
TopoSort Timing

<table>
<thead>
<tr>
<th>E</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.00001</td>
</tr>
<tr>
<td>652</td>
<td>0.00019</td>
</tr>
<tr>
<td>2183</td>
<td>0.00061</td>
</tr>
<tr>
<td>9780</td>
<td>0.00245</td>
</tr>
<tr>
<td>60653</td>
<td>0.02180</td>
</tr>
<tr>
<td>249103</td>
<td>0.10778</td>
</tr>
<tr>
<td>1563737</td>
<td>0.83010</td>
</tr>
<tr>
<td>6257067</td>
<td>3.75096</td>
</tr>
</tbody>
</table>

I would like to point out here that |E| is being used as the wc -l count of each DAG generated file. As per Piazza, |E| will suffice for |V| + |E|

---

Compute each nodes’ inDegree and store into a dictionary. (O(# of edges))
Finding which nodes have an inDegree of 0. (O(# of nodes))
For each node to be popped and appended to tps (O(# of nodes))
For reducing each inDegree of the correct node (O(# of edges))
Total O(# nodes + # edges)

Linear time!

Initially I chose to implement this topological sort using the delete_node which was supplied to us in the src.py file. The delete_node method was useful in removing the specific value from a node key which needed to have its inDegree decremented. However I recorded 129 seconds to process a file of size 5000. I realized that it wasn’t necessarily important to remove them this way, if I could access the integers which represent the #of InDegrees on a specific node. Upon beginning to optimize the program, I re-adjusted my logic so I can decrement the integer number of inDegrees (from compute_inDegrees) of a node key rather than removing the specific value from that node key list. This improved the efficiency of the code tremendously.

I believe my program meets the theoretical complexity bounds for the Topological sort based on the input size pretty well. If you look at the graph, you can see it is linear time!
<table>
<thead>
<tr>
<th>#Nodes</th>
<th>#Edges</th>
<th>V+E</th>
<th>Time</th>
<th>(V+E)/Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>8</td>
<td>0.00002 secs</td>
<td>400,000</td>
</tr>
<tr>
<td>10</td>
<td>27</td>
<td>37</td>
<td>0.00003 secs</td>
<td>1,233,333</td>
</tr>
<tr>
<td>15</td>
<td>29</td>
<td>44</td>
<td>0.00005 secs</td>
<td>8,800,000</td>
</tr>
<tr>
<td>50</td>
<td>688</td>
<td>738</td>
<td>0.0013 secs</td>
<td>5,676,923</td>
</tr>
<tr>
<td>100</td>
<td>2,602</td>
<td>2702</td>
<td>0.0035 secs</td>
<td>7,720,000</td>
</tr>
<tr>
<td>200</td>
<td>8,744</td>
<td>8944</td>
<td>0.0138 secs</td>
<td>6,481,159</td>
</tr>
<tr>
<td>500</td>
<td>58,996</td>
<td>59496</td>
<td>0.0981 secs</td>
<td>6,064,831</td>
</tr>
<tr>
<td>1000</td>
<td>253,793</td>
<td>254793</td>
<td>0.5422 secs</td>
<td>4,699,243</td>
</tr>
<tr>
<td>2500</td>
<td>1,549,946</td>
<td>1552446</td>
<td>0.38764 secs</td>
<td>4,004,865</td>
</tr>
<tr>
<td>5000</td>
<td>6,287,394</td>
<td>6292394</td>
<td>1.85477 secs</td>
<td>3,392,546</td>
</tr>
</tbody>
</table>

*units are measured in \((\text{abs}(\text{Vertices} + \text{Edges})))\)

Analysis:

Comparing the run time to the \(|V| + |E|\), my code runs around what is expected (linear growth). The last column is the only concrete number to compare everything to, although the number provided is virtually meaningless. In short, this number represents at an average rate of 4,847,290 units* a second.

If you plot the numbers \((V+E, \text{Time})\) into a graph, you will get a linear regression model that is increasing as the input converges to infinity. If the time was plotted to more decimal places, I would imagine it would be tighter in terms of distance from the line, \(t = 48247290(x) + 0; \ x \text{ being } (|V| + |E|).\)

My original algorithm contained a double for loop, \(n^2\) runtime, but then I modified the colors to not search inside the list of “nodes”, and instead made it a dictionary of \{key: Boolean\}s to represent when the node was visited. And since we weren’t checking for loops, we wouldn’t need to check if node.color == gray, leaving us with 2 colors; white or black/0 or 1/T or F.
Time Complexity Analysis of Topological Sort

The expected time complexity to run a topological sort via depth first search on a given graph is linear \( V + E \), where \( V \) is number of vertices in the graph, and \( E \) is number of edges. This is because in order to figure out where nodes go in a topological sort, it is necessary to process every single edge and vertex, but it is never necessary to process a vertex or edge more than once. To prove that my program satisfies this expectation, I generated random graphs of sizes given for the assignment. I then plotted those graphs’ \( V+E \) against my program’s running time on them.

![Graph](image)

This fits a linear regression so well that the regression (dotted line) is hard to see. This linear regression gives an \( R^2 \) value of 1 (according to Excel, which is usually accurate to about 3 decimal places), an exceptional fit.

It is also interesting to note that \text{time().time()} appears to be more accurate than \text{time().process_time()}, because using \text{time().process_time()} results in a 0.000 time for the first several sizes tested, whereas \text{time().time()} reports linearly increasing times as expected. Because of this, I opted to use \text{time().time()} in the time complexity analysis of my program.

As a proof of that, the program accuracy_test shows \text{time().time()} being accurate to 1/1024 seconds (approximately) while \text{time().process_time()} is only accurate to exactly 1/64 seconds.
The complexity of a topological sort should be $\Theta(V + E)$. Since the maximum number of edges a graph can have is the square of its nodes, this means that the worst case-complexity with $n$ nodes would be $\Theta(n + n^2)$, which can more simply be written as $\Theta(n^2)$.

Taking the data I got from the tests given in the assignment, I plotted the eight points in a coordinate plane on Desmos and connected them, making the green graph shown above. For comparison, I added two quadratic graphs with slightly different coefficients, represented by the blue and black lines.

Seeing how the data fit in with the two quadratic lines, I suspect that my implementation is quadratic as well. Since the worst-case complexity of the topological sort should be $\Theta(n^3)$, I conclude that my implementation is roughly in line with the complexity bounds.
In order to analyze the time complexity, running time was recorded for random graphs of sizes [5, 50, 100, 200, 500, 1000, 2500, 5000]. The experiment was repeated several times and the maximum value was selected for each graph size.

Current algorithm complexity depends on the two values directly affected by the size of the graph: number of vertices and number of edges. While the number of vertices is equal to \( n \), the maximum number of edges in DAG can be calculated with the formula: \( (n-1)n/2 \).

The results can only be considered successful if the plotted graph is close to \( C^*(V+E) \) where \( C \) is a constant. Given the \( V=n \) and \( E=(n-1)n/2 \) we can transform the control function in \( C^*(n+(n-1)n/2) \), and, therefore, \( C^* \theta(n^2) \). Therefore, \( \theta(n^2) \) should be growing at the same rate as the results from an experiment.

The running time for \( n=1000 \) was equal to 0.09375. Therefore, constant \( C \) can be roughly estimated as \( f(1000)/0.09375 \). The results will be plotted for the following functions:

\[
\begin{align*}
\theta &= 10^{-7}n^2 \\
O &= n^3/10666666666 \\
\Omega &= n^*\log(n)/32000
\end{align*}
\]

As it can be seen from the plot, Running Time and \( C^*n^2 \) were close to equal for all points. Their lines in the graph overlap, what points out that their values are very close. Meanwhile, even though \( n^3 \) and \( n^*\log(n) \) were asymptotically tightened to the \( n=100 \), their growth behavior is significantly different from the observed complexity.

Since the current results are very close to anticipated values, I find the algorithm implementation satisfactory.
Achieving Best Complexity

**Goal:** $T(n) = 2T(n/2) + O(n)$ for a set $Q$ of size $n$ points

**Issue:** Each recursion needs $X_L$, $X_R$, $Y_L$, $Y_R$, and $Y'$ SORTED!

With your partner, discuss how to do this without having to re-sort with each recursion.
Level of Abstraction for Pseudo-Code?

```
FIND-MAXIMUM-SUBARRAY (A, low, high)
1   if high == low
2      return (low, high, A[low])  // base case: only one element
3   else mid = ⌊(low + high)/2⌋
4      (left-low, left-high, left-sum) =
5         FIND-MAXIMUM-SUBARRAY (A, low, mid)
6      (right-low, right-high, right-sum) =
7         FIND-MAXIMUM-SUBARRAY (A, mid + 1, high)
8      (cross-low, cross-high, cross-sum) =
9         FIND-MAX-CROSSING-SUBARRAY (A, low, mid, high)
10     if left-sum ≥ right-sum and left-sum ≥ cross-sum
11        return (left-low, left-high, left-sum)
12     elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
13        return (right-low, right-high, right-sum)
14     else return (cross-low, cross-high, cross-sum)
```

```
FIND-MAX-CROSSING-SUBARRAY (A, low, mid, high)
1   left-sum = −∞
2   sum = 0
3   for i = mid downto low
4      sum = sum + A[i]
5      if sum > left-sum
6         left-sum = sum
7         max-left = i
8   right-sum = −∞
9   sum = 0
10  for j = mid + 1 to high
11     sum = sum + A[j]
12     if sum > right-sum
13        right-sum = sum
14        max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```
Algorithm

Define a complete pseudo-code for the closest pair algorithm, both the setup and the recursive procedure.

Work with your partner, using the worksheets we’ve supplied plus additional paper as needed.

To get credit for this, you need to make sure one of us checks your work by the end of class today.
Data Structures, etc.
You must use the **numpy** library.

References:


[https://docs.scipy.org/doc/numpy-1.13.0/reference/index.html](https://docs.scipy.org/doc/numpy-1.13.0/reference/index.html)

Data Structures, etc.

The input file will be a file containing lines like this:

```
pointId x-coord y-coord
```

Where `pointId` is an integer > 0 and the coordinates can be real numbers.

Use the `np.genfromtxt` function to read in the point data.

For distance, create an numpy array of the differences in your points, then use `np.linalg.norm` to get the L2 or Frobenius norm, which is the distance:

```
|x|_2 = \sqrt{x_1^2 + x_2^2}
```

https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.linalg.norm.html#numpy.linalg.norm
More etc.

• Your Python3 program must be called `close.py`

• The input file will be an argument to the program. We will run your program like this:
  ```
  python3 close.py <test_file.txt>
  ```

• You must write an output file using the same base name as the input file and **prepending** “output_” to that name. The output file must contain the ids of the closest points.

Ali’s solution has 6 helper functions (ranging from 2 lines to 16 lines of code), a 25-line `CLOSEST` function, and a 7-line “main”.
Helper Functions

With your partner, decide what helper functions you need, and write pseudo-code for them. After we’ve checked these you can program them in Python3.

To get credit for this task, you need to make sure one of us checks your work.
Additional Tips

Sorting:

```python
nps = [['FortCollins', 'Colorado', 'RMNP'],
['Mariposa', 'California', 'Yosemite'],
['BarHarbor', 'Maine', 'Acadia'],
['ThreeRivers', 'California', 'Sequoia'],
['ThreeRivers', 'California', 'KingsCanyon””']]

X = sorted(nps)
Y = sorted(nps, key=lambda row: row[1])
Z = sorted(nps, key=lambda row: row[2])
```
Grading
The in-class work for this assignment is worth 70 points: 50 points for the pseudo-code and 20 points for the helper functions list/pseudo-code.

The additional 30 points for the assignment are as follows:

• 15 points for passing tests
• 7 points for proper use of GitHub
• 8 points for good complexity analysis discussion (including graph of performance – use appropriate independent and dependent variables and their axes)
Submission

You must submit the following items in a zip file:

• close.py
• any additional files it imports
• a .pdf file that shows the results of timing your program on different sized inputs and a brief discussion of how well you think your implementation meets the theoretical complexity bounds for the algorithm based on input size.

BE SURE TO UNZIP YOUR SUBMISSION ON THE SCHOOL MACHINES AND RUN YOUR PROGRAM. If your program fails to run on the school machines with our tests, you will receive a zero.

Your complexity analysis file must be a .pdf file. We will not be grading any other type of file.

You must also use the class GitHub site.
Image Credits

voronoiClosePts.png: http://sanet.csis.u-tokyo.ac.jp/sub_en/about_sanet.html