CS 630 – Fall 2024 – Lab 2
Sep 18, 2024

1. Longest Simple Paths in Disconnected Graphs

The longest path problem presented in lecture asks for the longest simple path (no repeating vertices) as measured by the sum of the edge weights. The longest path problem appears to be expensive in the worst case and often takes exponential time in practice. Suppose that you have access to a cloud API that will solve instances of longest path problem for a fee, but only for connected graphs.

1. If you are given an arbitrary graph G of n vertices and m edges, how can you solve the longest path problem using the cloud API? Beware that G is not necessarily connected.
2. How much time does your algorithm need not counting the calls to the cloud API? Express your answer in $O()$ notation.
3. What is the maximum number of calls to the cloud API used by your algorithm?

Solution:

1. First, you should partition the graph into connected components so you can use the cloud API. This can be done using any traversal such as depth first search. You just need to make sure that your implementation restarts on new nodes until all nodes have been visited – you cannot assume you have a single root to start from. Each of these restarts will start a new connected component.

   After your algorithm has divided the graph into connected components, it should call the cloud API on each of them and return the highest value returned by an individual API call. These values may not be combined since they are from disconnected components.
2. The graph traversal to find connected components should take $O(n+m)$ time. Additional bookkeeping to prepare the API calls should also take $O(n+m)$ as the sum of the numbers of vertices and the sum of the numbers of edges across API calls will add up to the original graph sizes.
3. The traversal may find up to n connected components in the worst case where none of the nodes are connected. So you may make n calls to the cloud API.

2. Linear Range Counting

Suppose we have a data set of numbers $X = x_1, x_2, \ldots, x_n$ and we need to answer range queries such as "How many x_i are between 30 and 50?".

1. If you just received a data set X of n numbers and are given a range query with bounds $x_{\min}$ and $x_{\max}$, how much time is required to answer the range query? Express your answer with $O()$ notation.
2. If you just received a sorted data set X of n numbers and are given a range query with bounds $x_{\min}$ and $x_{\max}$, how much time is required? Express your answer in $O()$ notation.
3. What subroutine did you use to answer (2) more quickly than (1)? How many times did you need to invoke that subroutine?

Solution:

1. Since the data set is in arbitrary order, it will take $O(n)$ time to scan the entire data set once amd check each number to see if it should be counted.
2. Since the data set is now sorted, the numbers matched by the query will form a contiguous range, and you can use binary search to find the beginning and end of that range. This will take $O(\log n)$ time for the two binary searches.
3. The subroutine was binary search and it was invoked twice.

3. All Pairs Reachability

A common problem with graphs is all pairs reachability. We will consider this problem with the graph's adjacency matrix $\mathbf{A}$ as input. The adjacency matrix $\mathbf{A}$ for a graph is defined as

$$A[i,j] = \begin{cases} 1 \text{ if there is an edge from node } i \text{ to node } j \\ 0 \text{ otherwise} \end{cases}$$

and the all pairs reachability matrix $\mathbf{R}$ is defined as

$$R[i,j] = \begin{cases} 1 \text{ if there is a path from node } i \text{ to node } j \\ 0 \text{ otherwise} \end{cases}$$

We can similarly define $\mathbf{R_{=k}}$ which only counts paths of length exactly k, and $\mathbf{R_{\leq k}}$ which counts path of length at most k. For a graph of n nodes, $\mathbf{R} = \mathbf{R_{\leq n}}$. We can compute $\mathbf{R}$ many ways. One way is to build it up inductively starting from $\mathbf{R_0}$ and $\mathbf{R_1}$. Both of those have simple formulas if the adjacency matrix is available.

$$\begin{aligned} R_{=0}[i,j] &= \mathbf{I} = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ otherwise} \end{cases} \\ R_{=1}[i,j] &= A[i,j] \end{aligned}$$

so $R_{=0} = I$ and $R_{=1} = A$.

We can see deeper connections between reachability and the adjacent matrix through matrix multiplication. Recall that if we multiply two $n \times n$ matrices, $\mathbf{A}$ and $\mathbf{B}$, the cells in the result $\mathbf{AB}$ are defined with the following formula.

$$\mathbf{AB}[i,j] \quad = \quad \sum_{k=1}^{n} A[i,k]B[k,j]$$

1. If we multiply the adjacency matrix $\mathbf{A}$ by itself, computing $\mathbf{A^2}$, how should we interpret each value $A^2[i,j]$?
2. How can we compute $\mathbf{R_{=2}}$ from $\mathbf{A^2}$?
3. How can we compute $\mathbf{R_{=k}}$ using matrix multiplication? How many matrix multiplications are required? Express your answer in $O()$ notation.
4. How can we adapt this computation to compute $\mathbf{R_{\leq k}}$ instead? How many matrix multiplications are required? Express your answer in $O()$ notation.
5. Bonus question: Can you additionally bound the number of matrix multiplications used for (4) independently of k, no matter how large k is?

Solution:

1. $A^2[i][j]$ is the number of paths of exact length 2 between nodes i and j. This comes from the summation

   $$A^2[i,j] \quad = \quad \sum_{k=1}^{n} A[i,k]A[k,j]$$

   where each $A[i,k]A[k,j]$ is 1 if there is a path from node i to node k to j and 0 otherwise.
2. $R_{=2}[i,j] = \min(A^2[i,j], 1)$ because $A^2[i,j] >= 1$ if such a path exists and zero if no such path exists.
3. For any $k_1, k_2 \geq 0$ such that $k_1 + k_2 = k$, then $R_{=k}[i,j] = \min((R_{=k_1}R_{=k_2})[i,j], 1)$. This follows from the arguments used for (1) and (2).

   The number of matrix multiplications required is $O(\log k)$. The basic idea is to use the repeated squaring algorithm – compute $\mathbf{R_{=1}}, \mathbf{R_{=2}}, \mathbf{R_{=4}}, \ldots$ until the next step would overshoot k. Each step uses one matrix multiplication, so there are $O(\log k)$ "squaring"

steps. Then use the binary representation of k to pick the subset to combine to get exactly $\mathbf{R_{=k}}$. This takes another $O(\log k)$ matrix multiplications, so the total number of matrix multiplications is still $O(\log k)$.

4. For $\mathbf{R_{\leq k}}$, the same argument applies for the recurrences. For any $k_1, k_2 \geq 0$ such that $k_1 + k_2 = k$, then $R_{\leq k}[i,j] = \min((R_{\leq k_1} R_{\leq k_2})[i,j], 1)$.

   However, we are missing $R_{\leq 1}$ to start the repeated squaring process.

$$R_{\leq 1}[i,j] \quad = \quad \max(R_{=0}[i,j], R_{=1}[i,j])$$

   which is sufficient to get started.

   Thus, calculating $\mathbf{R_{\leq k}}$ takes $O(\log k)$ matrix multiplications.

5. For $\mathbf{R_{\leq k}}$, another bound on the number of matrix multiplications required is $O(\log n)$. This is because the maximum length of a simple path is n, the number of vertices, so once $k > n$, no more pairs will become reachable.