Graph Sorting by Edge Count

Jonathan Stokes

December 14, 2025

Introduction

I wanted an algorithm to enumerate all undirected simple labeled graphs on n nodes.The number of these graphs is straightforward to compute. Given a graph of n nodes there are $n(n-1)/2$ possible edges. Each of these edges may either exist or not meaning that there are $2^{n(n-1)/2}$ possible graphs on n nodes. There are probably many ways of enumerating these graphs but I think the following mapping to the natural numbers is cute.

Algorithm

Given a graph of n nodes there are $n(n-1)/2$ edge locations. These edge locations can be represented as a binary string of length $n(n-1)/2$, there are $m = 2^{n(n-1)/2}$ permutations of this string each of which can represent both a graph and a natural number. Therefore to enumerate all simple undirected labeled graphs on n nodes and sort them by edge count simply collect the natural numbers by the number of 1s in their binary representation.

To state this algorithm formally it helps to introduce some notation. Let the set of simple undirected graphs on n nodes be $\mathcal{G}^n$ and the set of these graphs with e edges be denoted as $\mathcal{G}_e^n$ such that $\mathcal{G}^n := \{\mathcal{G}_e^n | e \in [0 \dots m]\}$. If x is a base 10 number let x_2 be its binary representation and the function $f(x) = x_2$ transform x from its base 10 representation to its base 2 representation. Additionally let $x_{2,i}$ be the i_{th} digit in binary representation x_2 where indexing starts at 0 and let $|x_2|$ be the number of digits in x_2.

Algorithm 1 Sort Graphs by Edge Count

Require: $n \geq 0$
Require: $\mathcal{G}^n = \emptyset$
1: $m = 2^{n(n-1)/2}$
2: **for** $g \in [0 \dots m-1]$ **do**
3: $\quad g_2 = f(g)$
4: $\quad e = \sum_{i=0}^{|g_2|-1} \mathbb{1}\{g_{2,i} = 1\}$
5: $\quad \mathcal{G}_e^n = \mathcal{G}_e^n \cup g$
6: **end for**
7: **return** $\mathcal{G}^n$

Results

I wrote this algorithm in Python as it is my language of choice for research. However, Python is not a particularly fast or memory efficient language and if $n = 8$ the number of graphs with n nodes is $m = 268,435,456$. So I decided this algorithm might be a good opportunity to see how Python threading performs without the global interpreter lock or GIL. The GIL is a memory safety feature in Python that

many argue hurts the performance of Python threads.

Before testing the performance of Python threads vs single and multi processes I wanted to find the optimal number of workers for the different compute methods. In Fig. 1 we see that the optimal number of workers for threads is 1 and the optimal number of workers for threads with no GIL or multiprocessing is 16. Also note that here threading with no GIL appears slightly slower than multiprocessing, as we will see in the following results this is not generally true and may be an artifact of looping over the graph enumeration algorithm to create the Fig. 1.

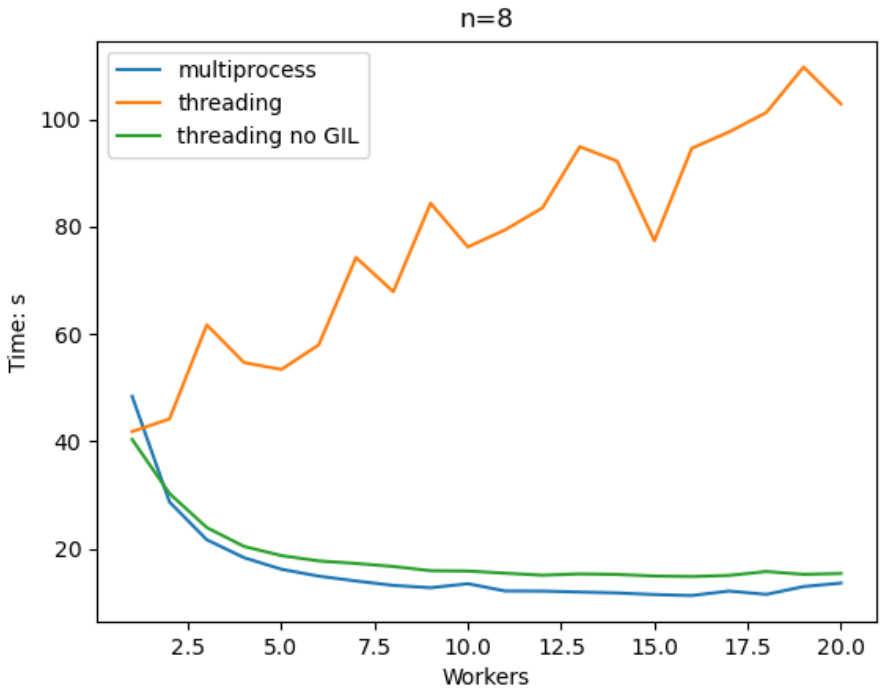

Figure 1

Using the optimal number of workers I then found the time and memory used by each compute method in Fig. 2 and Fig. 3.

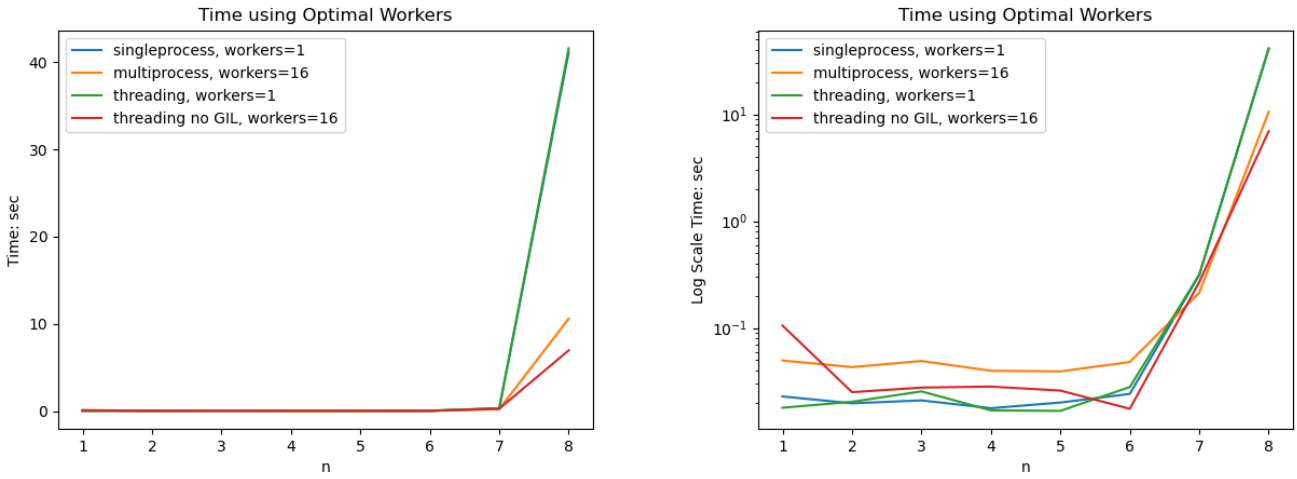

Figure 2: Left: Linear Y, Right: Log Y

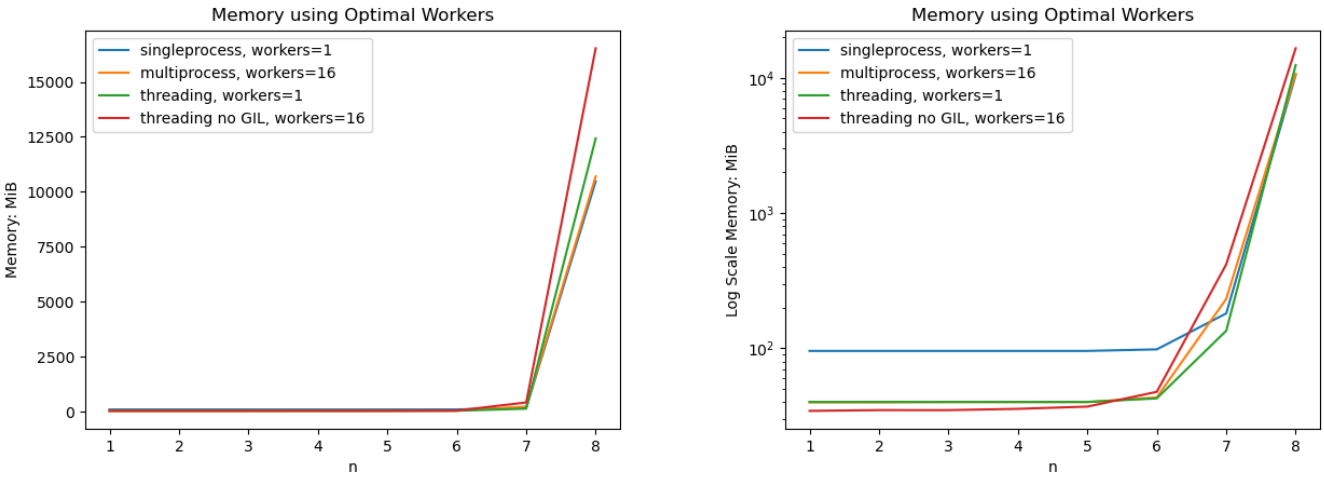

Figure 3: Left: Linear Y, Right: Log Y

In summary, for this graph enumeration algorithm threading with no GIL is slightly faster than multiprocessing, however it also uses slightly more memory. Therefore given that turning GIL on and off in Python currently requires switching between Python builds, I will probably stick with multiprocessing over threading without GIL for now.