iFUB

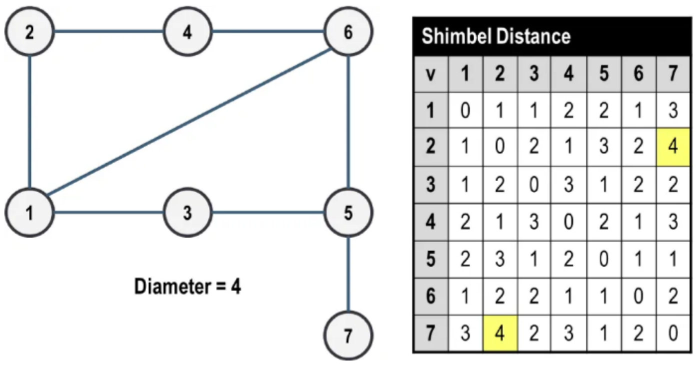
Aakash Aanegola - 2019101009
Dhruv Kapur - 2019101038

Finding the diameter of a graph

The diameter of a graph is defined as the longest shortest path between any two nodes for the entire graph. The diameter of a graph is an important property as it informs us about the "connectivity" of the graph, i.e. an upper bound on how many hops are required to travel from one node to another.

$$\delta_G = \max_{u,v} s(u,v)$$
$$s(u,v) = \min_{v} d(u,v)$$

Till date there have been no algorithms that have breached the $O(VE)$ or $O(V^3)$ barrier, but some parallelized and pruning algorithms have been suggested.



v	1	2	3	4	5	6	7
1	0	1	1	2	2	1	3
2	1	0	2	1	3	2	4
3	1	2	0	3	1	2	2
4	2	1	3	0	2	1	3
5	2	3	1	2	0	1	1
6	1	2	2	1	1	0	2
7	3	4	2	3	1	2	0

Diameter of a Graph

Notation

δ_G - Diameter of a graph G

$d(u,v)$ - Distance between u and v

V - Number of vertices in a graph

E - Number of edges in a graph

$ecc(u)$ - Eccentricity of a node, i.e the distance of the furthest node from that node

$F_i(u)$ - i^{th} fringe set of u, the set of all nodes that are at a distance of i from u

The algorithms

Naive diameter computation (APSP)

The diameter of a graph can be computed using the BFS algorithm in the following way:

- For a node u the max depth of the BFS tree is equal to $ecc(u)$
- The diameter of the graph is equal to $\max_u ecc(u)$
- From the above two statements, we can create an algorithm to compute the diameter of a tree

```
Naive-BFS-Diameter(G):
  diameter = 0
  for u in G.nodes:
    diameter = max(diameter, ecc(u))
  return diameter
```

In the above pseudo-code, $ecc(u)$ performs a BFS from u and returns the max depth.

Parallelized Naive diameter computation

The above algorithm can be parallelized by recognizing that computation of the eccentricity of a node u, given by the function `ecc(u)` is independent of the same for the other nodes. As a result, we can run the eccentricity calculation for each node in parallel by invoking `ecc(u)` on separate computing resources (thread for instance), which would speed up the process by an order of c, the number of parallel processes.

iFUB
The iFUB algorithm is a pruning algorithm that decreases the number of BFS calls required. In the naive case, we perform $O(V)$ BFS's which may not be required in every case. The algorithm accomplishes the pruning due to the following theorem:

For any $1 \leq i < ecc(u)$ *and* $1 \leq k < i$ *and for any* $x \in F_{i-k}(u)$ *such that* $ecc(x) > 2(i-1)$*, there exists a complementary* $y_x \in F_j(u)$ *such that* $d(x, y_x) = ecc(x)$ *with* $j >= i$

Without getting into the proof, this essentially translates to "*if there is a node v at a certain depth of the tree i, and it has an eccentricity of more than $2(i-1)$ then the eccentricities of all the nodes at depths less than i are less than $ecc(v)$.*"

Using this theorem we can construct a termination criterion for the search for the diameter in the following way

```
iFUB(G):
  i = ecc(u)
  lb = ecc(u)
  ub = 2*ecc(u)
  while ub-lb > 0:
    B_i(u) = max(ecc(v) for v in F_i(u))
    lb = max(lb, B_i(u))

    if lb > 2(i-1):
      break
    ub = 2(i-1)
    i = i-1
  return lb
```

Choosing the start node u can be done randomly or using another algorithm known as *four-sweep* that selects a node that has a low eccentricity by choosing a central node.

```
four-sweep(G):
  r1 <- highest degree node
  a1 <- furthest node from r1
  b1 <- furthest node from a1
  r2 <- midpoint of a1-b1 path
  a2 <- furthest node from r2
  b2 <- furthest node from a2
  u  <- midpoint of a2-b2 path
  return u
```

Parallel iFUB

Again, we can parallelize the above algorithm by calling each `ecc(u)` call on a different thread.

Our methods

Graph generation
Apart from the graphs (sparse matrix collection) available online, we also decided to generate graphs given the number of nodes and an upper bound on the number of edges. To generate a graph with n vertices we first generate a Prufer sequence of length $n-2$ which maps to a tree with n vertices.

A Prufer sequence is a sequence of length $n-2$ where each of its elements corresponds to a node u in the tree, i.e. the elements belong to $\{1, 2, ..., n\}$. Prufer sequences are converted to trees in the following way:

```
Convert-Prüfer-to-Tree(pf)
  n ← length[pf]
  T ← a graph with n + 2 isolated nodes, numbered 1 to n + 2
  degree ← an array of integers
  for each node i in T do
    degree[i] ← 1
  for each value i in pf do
    degree[i] ← degree[i] + 1

  for each value i in a do
    for each node j in T do
      if degree[j] = 1 then
        Insert edge[i, j] into T
          degree[i] ← degree[i] - 1
          degree[j] ← degree[j] - 1
          break
  u ← v ← 0
  for each node i in T
    if degree[i] = 1 then
      if u = 0 then
        u ← i
      else
        v ← i
        break
  Insert edge[u, v] into T
  degree[u] ← degree[u] - 1
  degree[v] ← degree[v] - 1
  return T
```

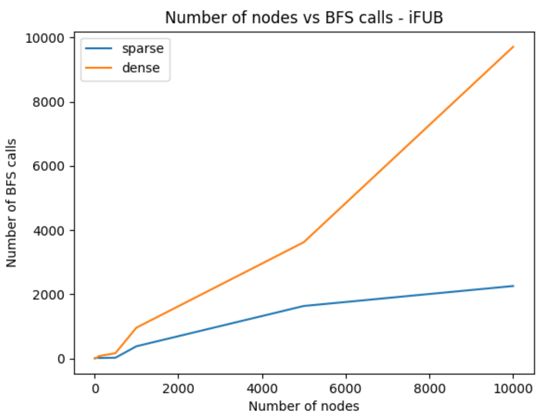
After we have a tree, we try to generate $m - n + 1$ edges (since this is a random process, we are not guaranteed that the edges will be created) randomly and add them to the graph as well. We can specify the upper bound on the number of edges and were able to generate graphs with $50,000$ nodes and $2,000,000$ edges in under 5 seconds.

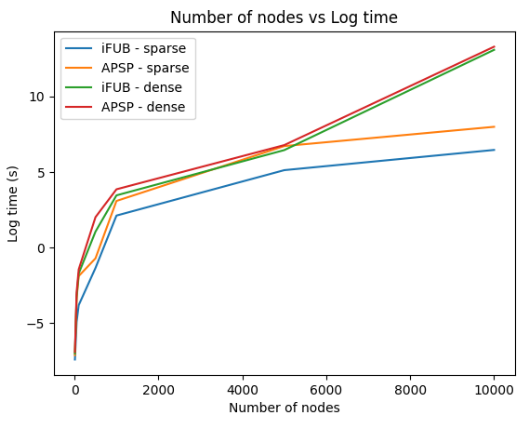
Results

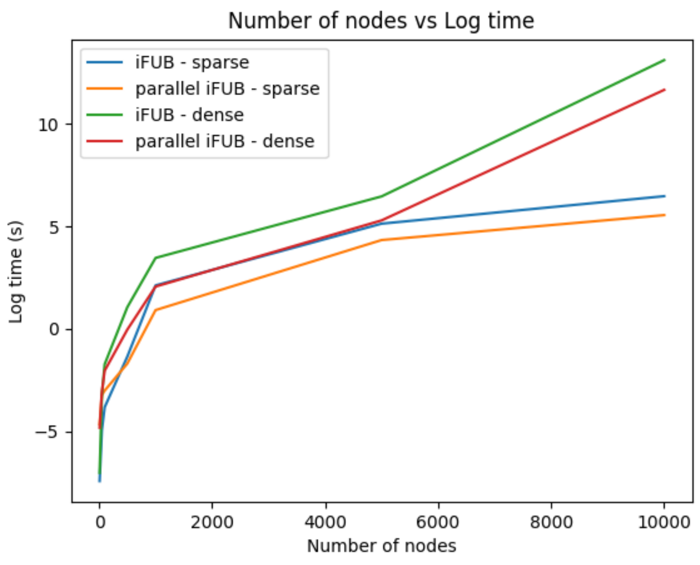
We compare the results of **iFUB**, **parallel-iFUB and** the naive diameter computation (APSP) algorithms. (all the times listed are in seconds)

Here we use both time and the number of BFS calls as a rubric.

Number of Nodes	Number of Edges	Diameter	Naive BFS count	iFUB BFS count	Naive BFS time	iFUB time	Parallel iFUB time
10	12	5	10	6	.0008	.0006	0.0096
10	17	4	10	6	.001	.0009	0.0081
50	99	6	50	20	.0342	.0072	0.0397
50	186	4	50	12	.0508	.0392	0.0464
100	198	8	100	14	.1508	.0220	0.0494
100	1637	2	100	72	.2348	.1791	0.1306
500	999	10	500	22	.4891	.2585	0.1840
500	4888	4	500	164	7.9432	2.8725	0.9630
1000	1996	10	1000	379	21.9210	8.3238	2.4962
1000	19595	3	1000	958	47.4501	31.7270	7.8582
5000	9994	12	5000	1635	831.0641	168.5402	75.8602
5000	99602	4	5000	3623	886.5420	636.7969	197.7632
10000	19999	14	10000	2257	2975.9291	643.4839	255.8250
10000	1477582	3	10000	9710	-	487567.4652	114452.3569







From the above table and graphs it is clear that the *iFUB* algorithm offers a reduction in the number of BFS calls required, and the effect is amplified for sparse graphs (graphs with fewer edges, and likely a larger diameter).

The iFUB algorithm offers a significant speedup (especially for sparse graphs) over the naive APSP based solution. Parallelizing the iFUB algorithm (using the 8 cores we had available on our local machines) offers a 3-5x speedup for all graphs (ignoring smaller graphs where thread creation is more expensive than the computation happening on each thread).