# MatrixBFS
# (or, more accurately, ParallelBFS)

Blake Appleby (aba2176)
Jonathan Hall (jah2328)
Ryan Wee (rw2800)

December 20 2023

**Note:** All source code can be found at `https://github.com/ryanwee1001/six-degrees-of-wikipedia`.

**Note:** Our project title is *MatrixBFS*, because our original plan was to parallelize BFS using matrix multiplication. However, we no longer use matrix multiplication. Hence, a more accurate name would be *ParallelBFS*.

## 1. Introduction

Our project aims to answer the question: Starting at some arbitrary wikipedia page $A$, how many clicks would you need to reach some other arbitrary wikipedia page $B$? (Assume of course that you can only click on links to other wikipedia pages, and can't just do a google search for page $B$.)[1]

It is quickly apparent that this is a shortest-path problem, on a directed graph with constant-weight edges. Each wikipedia page is a node, and each hyperlink on page $A$ to page $B$ is an edge from $A$ to $B$. Since edge weights are constant, we can solve this using BFS. Our goal is to parallelize this BFS.

[1] There is in fact a site dedicated solely to answering this question: `https://www.sixdegreesofwikipedia.com/`. It appears that "Columbia University" is four clicks away from "Haskell".

# 2. Design and Implementation

## 2.1 Parallelizing Matrix Multiplication

Our original approach was to approach BFS as a matrix multiplication problem. In particular, multiplying the adjacency matrix by a vector representing the current frontier yields the next frontier.[2] Parallelizing matrix multiplication thus allows us to parallelize BFS. In particular, each column of the adjacency matrix can be allocated to a different core.

However, we found that this approach was relatively ineffective. In particular, the graph of wikipedia pages is a sparse graph, i.e. $|E| = O(|V|)$. Naive BFS has a complexity of $O(|E|+|V|)$, which on a sparse graph is essentially $O(|V|)$. On the other hand, matrix multiplication is typically $O(|V|^3)$.[3] Assuming perfect parallelization on $N$ cores, the runtime of parallel matrix BFS would still be $O(\frac{|V|^3}{N})$. Since $N$ on a typical computer is nowhere near $|V|^2$, parallel matrix BFS would be significantly slower than naive BFS.[4]

## 2.2 Parallelizing frontier exploration

We can think of BFS as an iterative algorithm, where in the $n$th iteration we consider nodes that have a distance of $n$ from the starting node. To get from iteration $n$ to iteration $n+1$, we need to find the direct neighbors of all nodes in the $n$th frontier. This process of frontier exploration opens up two opportunities for parallelization:

- Say there are $k$ nodes in the $n$th frontier. Then we can get the neighbors for each of these $k$ nodes in parallel.
- The above step gives us $k$ sets of nodes. However, some node $N$ may be the neighbor of two or more nodes in the $n$th frontier, i.e. it may be a member of two or more of these sets. Hence, we need to get the union of these $k$ sets to form the next frontier. This multi-set union can be done in parallel.

[2] See our proposal for more details.

[3] The fastest matrix multiplication algorithm is around $O(|V|^{2.37})$.

[4] Our computers had around 8 cores. By comparison, there are more than 6e6 wikipedia articles, so $|V|^2$ would be something like 3.6e13.

In particular, we implement our recursive BFS function as follows. Note that we stop our BFS after a depth of 6. This is because we could easily end up choosing a start node and end node that are in two distinct connected components of the graph. If the connected component that the start node belongs to is especially large, searching through the entire component could take a very long time.

```
-- Does union on a list of sets in parallel
parallelUnion :: Ord a => [Set.Set a] -> Par (Set.Set a)
parallelUnion [] = do
    return Set.empty
parallelUnion [s] = do
    return s
parallelUnion sets = do
    let (half1, half2) = splitAt (length sets `div` 2) sets
    mergedHalf1 <- parallelUnion half1
    mergedHalf2 <- parallelUnion half2
    return $ Set.union mergedHalf1 mergedHalf2

-- Recursive function for BFS that is executed in parallel.
parallelBFS :: DirectedGraph -> [Node] -> Set.Set Node ->
        Int -> Node -> Par Int
parallelBFS _ _ _ 6 _ = do
    return (-1)
parallelBFS graph frontier visited dist target = do
    let neighbors = (parMap rpar (getNeighbors graph) frontier)
    tmpFrontier <- parallelUnion neighbors
    let nextFrontierSet = Set.difference tmpFrontier visited
    if null nextFrontierSet then
        return (-1)
    else do
        if Set.member target nextFrontierSet then
            return (dist + 1)
        else do
            let newVisited = Set.union visited nextFrontierSet
            let newFrontier = Set.toList nextFrontierSet
            parallelBFS graph newFrontier newVisited (dist + 1)
                    target
```

### 2.3 Parallelizing query handling

Another opportunity for parallelization is handling multiple *(start node, end node)* queries in parallel. We implement this as follows.

```
runQueries :: DirectedGraph -> [Query] -> Par [Int]
runQueries graph queries = do
    parMapM (parallelBFSDriver graph) queries
```

## 3. Evaluation

### 3.1 Setup

The data needed to build this graph is freely available for download at `https://en.wikipedia.org/wiki/Wikipedia:Database_download`. In particular, we extract two files from this dataset:

- *wikigraph.nodes*, containing a mapping from wikipedia page names to integer IDs. This file is formatted as a JSON file. To reduce memory usage, we only include mappings for pages used in our queries.
- *wikigraph.edges*, containing the edges between pages. Each node is represented using its integer ID instead of its string-based name. This file is formatted as a binary file. The neighbors of each page are represented as a sequential array of *int32*s, with a 0x0000 entry used as a delimiter between pages.

We also create a third file, *wikigraph.queries*. This file contains a set of queries formatted as csv lines. For instance, the query from "Columbia University" to "Haskell" is formatted as *Columbia University,Haskell*. During our evaluation, we observed that many wikipedia pages are auto-generated stubs with only one or two links. Hence, we only use 'good pages' that have $\geq 50$ links as the start node or end node of our queries. Pages that do not fulfil this criterion are still included in our graph; they just are not used as the start node or end node of our queries.

Samples of all three files can be found in the *data/* folder of the associated tarball. Note that we heavily truncate our *wikigraph.edges* file, because the original file is around 186MB. Hence, our code cannot be executed directly on the sample data provided. Feel free to contact us for the actual data files we used.

## 3.2 Methodology

Since the graph is very large, loading it into memory takes a non-significant amount of time.[5] Hence, we exclude the time taken for this I/O from the timing results. We do so by using 'seq' to force the maps and lists representing our graph to be fully evaluated before we commence our BFS.

```
orceEvaluationOfMap :: Map.Map k v -> ()
forceEvaluationOfMap m = Map.foldlWithKey
        (\_ k v -> k ‘seq‘ v ‘seq‘ ()) () m

forceEvaluationOfList :: [a] -> ()
forceEvaluationOfList l = foldl (\_ x -> x ‘seq‘ ()) () l

main :: IO ()
main = do
    edges <- readBinaryFileToGraph edgesFile
    nodes <- readJSONFileToNodes nodesFile
    queries <- readCSVFileToQueries queriesFile nodes

    -- Force the full evaluation of edges / nodes / queries,
    -- so we don’t include I/O in our timing results.
    let resE = forceEvaluationOfMap edges
    let resN = forceEvaluationOfMap nodes
    let resQ = forceEvaluationOfList queries
    putStrLn $ "Edges loaded: " ++ show (resE == ())
    putStrLn $ "Nodes loaded: " ++ show (resN == ())
    putStrLn $ "Queries loaded: " ++ show (resQ == ())

    start <- getCPUTime
    putStrLn $ "Results: " ++
        (show $ runPar $ runQueries edges queries)
    end <- getCPUTime
```

[5]This is also because the functions we use to parse the files are not very well-optimized.

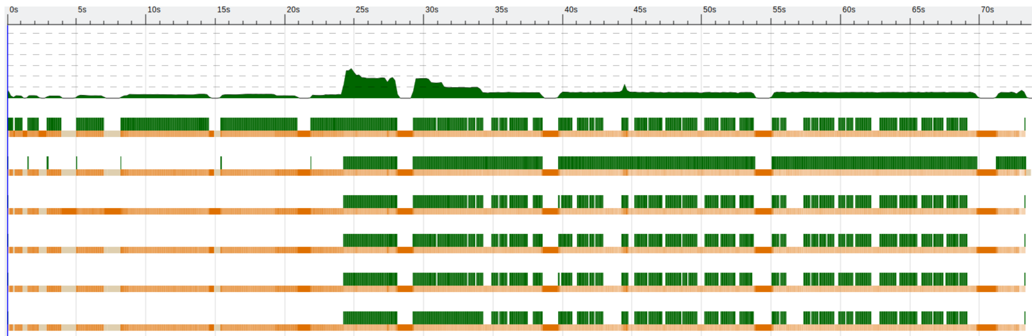


Figure 1: Threadscope results

## 3.3 Results

We ran a set of five random queries with varying results (in particular, shortest paths of 5, 6, 4, 5, 4). Figure 1 shows our threadscope.

The threadscope shows a good degree of parallelization. However, in terms of actual runtime, we saw a slowdown when using more cores instead of a speedup. In particular, the runtime (not including I/O) over the number of cores was as follows:

| Number of cores | Runtime (microseconds) |
|---|---|
| 1 | 65678413 |
| 2 | 73050117 |
| 3 | 79131775 |
| 4 | 78536043 |
| 5 | 88129798 |
| 6 | 92790432 |

Running on 6 cores with the '-s' flag gave the following information on sparks:

```
SPARKS: 3896628 (
    276198 converted,
    1741526 overflowed,
    0 dud,
    1616020 GC'd,
    205540 fizzled
)
```

# 4. Future Work

There are many parameters we could tune, for instance:

- The depth limit of our BFS (currently, we stop at 6)
- The types of queries used (do we use queries that have an answer of 1, or queries that have an answer of 6?)
- The number of queries used
- The data structure used for processing edges

Another area of future work would be bidirectional BFS, i.e. doing a BFS from both the start node and the end node to reduce the search space.