Six Degrees of Wikipedia: Final Report

Implementation

Serial

For the serial implementation, I did a basic BFS algorithm. I used the Data.Sequence library as my container due to better performance characteristics when appending to the back than native lists. The basic implementation was to pull a path from the queue, get all outgoing links for the last node in that path, check if any of those were the target node (if so, return that path + target), otherwise append an additional path with each linked page to the end of the queue. I also added seen nodes to a Data.Set.

Parallel

The Parallel implementation expands on the previous BFS algorithm by adding workers pulling from the queue and writing to it. To adjust to this, the queue is a Channel, with the set of seen nodes stored in an MVar.

Performance Issues

These solutions unfortunately do not scale well whatsoever. The main issue is the database queries. Database queries on a database as large as Wikipedia take 3-5 seconds per query. Getting the outgoing links for each page individually, when each page has ~500 outgoing links, and those all need to be searched makes it impossible to complete shortest path search problems in a reasonable amount of time. This is not alleviated by parallel solutions at all. The issue is fundamentally at the point of I/O. The other problem with the parallel solution is that the database errors out while reading, possibly due to lock files or something else.

Possible Solutions

To alleviate these problems, there are a few possible solutions. I could somehow batch I/O queries together in order to avoid additional round trips around querying. Using a more scalable database that can support more readers at once would help some, but ultimately even with 100 concurrent threads running DB queries at once, it would still require ~15 seconds for a single page's referenced pages to all be processed. The best option would be to load the entire database into Haskell via a binary format of some sort, thus putting everything into memory and lowering I/O time to a minimum. Doing this reading upfront would most likely be faster in the end.

Project Takeaways

- Start projects earlier!
- I ran into many issues getting the database setup, and it would probably be best to avoid unfamiliar technologies as much as possible in order to reduce complexity.
- Think more carefully about where bottlenecks are. I attempted to implement a few improvements on the BFS algorithm, but ultimately nothing can overcome the I/O speed other than having everything loaded in memory.
- Haskell makes you formally define the specifications of your program. When adjusting quickly, it may be better to use Python or another less strict language in order to move through ideas quickly.