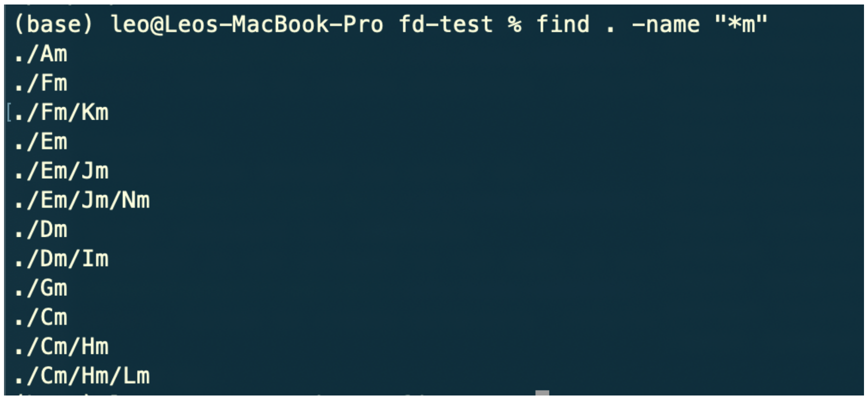
findw

Introduction

The *find* command in Unix-based operating systems is used to recursively navigate file directory hierarchies according to a given text or regular expression pattern. For example, one can recursively navigate and find a list of all files or directories starting with some name in a folder, where the path of each file from start to end is shown.



```
(base) leo@Leos-MacBook-Pro fd-test % find . -name "*m"
./Am
./Fm
./Fm/Km
./Em
./Em/Jm
./Em/Jm/Nm
./Dm
./Dm/Im
./Gm
./Cm
./Cm/Hm
./Cm/Hm/Lm
```

An example of output from the find command is shown above, where it recursively lists all files or directories whose name matches the pattern “*m” (e.g Am,Bm,Cm…). find also supports a maxdepth argument to specify the maximum number of levels (subdirectories) to search until.

The aim of my independent project, findw, was to investigate the possibility of building a parallel and recursive adaptation of the find command that could work for web pages. In this case, given the URL for an initial page, one could possibly find a list of all pages containing some text pattern in their title that are accessible through the initial page.

A tool such as this could be useful in cases such as search engine optimization analysis, where an analyst may want to get information about the way a page is connected to other pages on the internet.

As compared to file directories, the web poses an additional challenge in that links can be cyclic while directories are usually directed acyclic graphs[1], and there is no way to know when to stop searching since links can be followed ad infinitum or until every link has already been seen. Hence, an argument similar to find’s maxdepth would be required instead of optional.

[1] In the case when they do contain cycles with symbolic links, the find command reports an error: https://serverfault.com/questions/265598/how-do-i-find-circular-symbolic-links. On the web, we would want to ignore this instead since cycles are to be expected.

In summary, the tool I wanted would have the following requirements:

- Can search webpages, the way the find command does for files / directories, in a recursive manner with a specifided
- Be relatively simple to use where even a non-technical person interested in the command line could use it
- Take advantage of parallelism and/or concurrency since paths (a series of page clicks) are independent of each other

However, I felt that the existing solutions I could find did not meet these requirements.

Some popular existing solutions such as Scrapy (https://scrapy.org/), Heritrix (https://github.com/internetarchive/heritrix3), and StormCrawler (https://github.com/DigitalPebble/storm-crawler) are frameworks meant for running or implementing web crawlers in various languages (Java, Python).

One could use these to implement a web scraper - to gather information about existing pages instead of finding new pages on the Internet - but they are not simple tools that are relatively compact and easy to use. Rather, they are meant as frameworks to support building tools such as findw or more advanced crawling programs, so programming knowledge is required to use them.

I did find a tool called *webgrep* (https://github.com/dhondta/webgrep) that similarly aims to port the Unix grep command (which searches instead the contents of text files) for web pages. However, it is not recursive. Additionally, there is *recursive_scraper* (https://crates.io/crates/recursive_scraper) which is closest to what findw does, but it is meant more for scraping an entire site and does not have a way to specify a maximum depth.

Choice of Rust

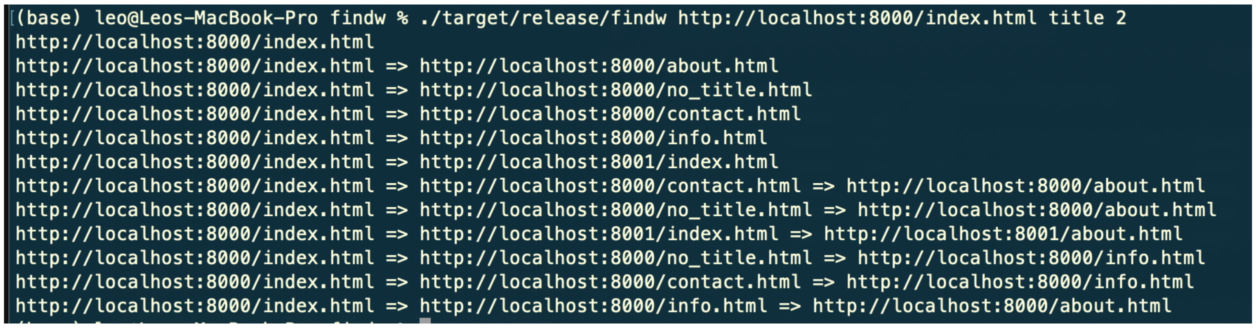
I chose to write findw in Rust as it is known for its emphasis on memory safety and speed, and has good support for concurrency and parallelism with the widely used tokio library.

Additionally, I had experience with Rust from a previous personal project which made it easier to use for me than more common language choices for command-line tools such as C or C++.

Inputs and outputs

findw takes as input, in order, three required arguments: the URL to start searching from, a text pattern to check against page titles, and the maximum depth limit as an integer >= 0.

For example, the input **http://localhost:8000/index.html title 2** means to start searching from the page at the URL specified for all pages containing the text ‘title’ in their page title, but only up to 2 clicks away.



```
(base) leo@Leos-MacBook-Pro findw % ./target/release/findw http://localhost:8000/index.html title 2
http://localhost:8000/index.html
http://localhost:8000/index.html => http://localhost:8000/about.html
http://localhost:8000/index.html => http://localhost:8000/no_title.html
http://localhost:8000/index.html => http://localhost:8000/contact.html
http://localhost:8000/index.html => http://localhost:8000/info.html
http://localhost:8000/index.html => http://localhost:8001/index.html
http://localhost:8000/index.html => http://localhost:8000/contact.html => http://localhost:8000/about.html
http://localhost:8000/index.html => http://localhost:8000/no_title.html => http://localhost:8000/about.html
http://localhost:8000/index.html => http://localhost:8001/index.html => http://localhost:8001/about.html
http://localhost:8000/index.html => http://localhost:8000/no_title.html => http://localhost:8000/info.html
http://localhost:8000/index.html => http://localhost:8000/contact.html => http://localhost:8000/info.html
http://localhost:8000/index.html => http://localhost:8000/info.html => http://localhost:8000/about.html
```

Example output (can also be found in **src/output/2.out**) is shown above. Each line of output is a string representing a path that can be followed from the first page, delimited by “=>”, where each arrow indicates another click. Paths are printed in no particular order: e.g a path 2 clicks away could be printed before a path 1 click away if the network request times were faster.

Every path printed indicates a series of clicks from the first page where the title for the last page in the path contains the text pattern specified. For instance, the line:

http://localhost:8000/index.html => http://localhost:8000/about.html

indicates that about.html can be reached in 1 click from the first page index.html, and the page title for about.html contains ‘title’ (without the quotes) as a substring.

Since the maximum depth specified is 2, we only see paths for pages up to 2 clicks away.

findw also supports an optional flag, **-t**, to use titles of pages in path output instead of the full URLs. However, URLs are printed by default as they are less ambiguous since pages can often have empty titles - in this case the -t option simply replaces those with a placeholder.

Implementation

GitHub link: https://github.com/leonidas1712/findw/tree/main

The code in **src/main.rs**, the entry point to the program, simply parses the inputs as per the previous page and passes them to the search function in **src/search.rs** - this is where the crux of the implementation is.

The search follows a modified version of the breadth-first search algorithm (BFS) [1, p.82]. In this case, the BFS queue is done implicitly through the use of a tokio::mpsc channel, where the main thread handles processing nodes from the queue in the while loop of search.rs.

Every node for the search represents some point along a path of clicked URLs from the start. This is represented by the Path struct (**src/search_helpers.rs**). Path contains the current depth, a vector (contents_array) for the list of URLs encountered so far on this path, a HashSet (path_vis) of the URLs encountered, and the latest url on this path.

The contents_array is used when printing the paths to output, and path_vis is needed as we need a way to avoid cycles while traversing a path. The contents_array contains the URL strings from path_vis, but this is a separate variable because HashSet does not preserve insertion order and we need to print the path according to insertion order. latest_url is simply a helper variable for use in the main logic.

The main loop of the search function tries to leverage parallelism by creating a separate tokio task (representing a new task or process that can possibly be scheduled in parallel) to process each new path node. Starting from the initial path node with a depth of 0, the following steps are taken for each node in the queue:

1. Get the page title and child URLs (URLs directly accessible from this page) from the page at the latest_url of the path node - this involves a network request
2. If the title matches the input pattern, print the full path for this node
3. If the current depth is less than the max depth limit, process child URLs that are not already visited along the path into new path nodes to add into the queue.
    - If the same URL appears twice on a page, it is processed only once into a new node to avoid unnecessary duplicate work

This continues until there is no more work to process either because we have reached the depth limit, or because all new URLs have already been seen along their paths.

In terms of parallelism, the main loop in search.rs is sequential as it can only process one path node at a time (the ‘single’ in multi-producer, single-consumer or mpsc) but ideally the generated tasks can be scheduled in parallel.

Non-trivial implementation details

The choice of BFS for the search algorithm instead of, say, depth first search (DFS), was because of its optimal nature where it will find the shortest path from source to destination [1].

In practice for findw this may not necessarily be the case if we stop at the first match found due to the additional overhead of network requests. Additionally, DFS is known to be inherently sequential, so a parallel implementation using DFS would have been more tricky [2].

When new path nodes are created for child nodes, the previous node's contents_array and path_vis are cloned entirely and updated.

This may seem prohibitively expensive, but the most obvious alternative would have been to maintain pointers to previous nodes' URLs (where each node just keeps its current URL and the previous) then traverse a series of pointers backwards each time we want to print a path.

For small depths (e.g < 10), the cloning strategy may not necessarily be much worse since traversing pointers could involve more overhead from cache misses than if we are simply traversing a contiguous array. However, I admit I did not get a chance to experiment with this due to time constraints.

Another non-trivial detail here is the issue of how to stop the search. Ideally, we would rely on the default behaviour of tokio's mpsc where when all transmitters have gone out of scope, the receiver in the main loop is also closed[2]. However, this did not seem possible here since an initial transmitter (first_tx) is created before the loop and is needed so we can clone transmitters for each new tokio task.

To solve this problem, I used a mutex (**sync**) containing the count of all nodes created so far[3]. This mutex is incremented when new child nodes are added to the queue (step 3 on the previous page) and decremented whenever a task is finished and exits. Once the count reaches 0, the last task sends a "Close" message to the receiver to signal that it can now end the main loop.

[2] https://docs.rs/tokio/latest/tokio/sync/mpsc/index.html

[3] As suggested by Prof Cristina

Testing and performance scripts

The following table summarizes the relevant files and folders in the repository. Some files and folders are omitted for brevity.

Name	**Description**
input/*.in	Files for input test cases
output/*.out	Files with correct output corresponding to cases in input/
perf/*.perf	Performance metrics (runtime) for various input cases
results/* (this is ignored)	Stores intermediate results from runs. This is under .gitignore but is used by various scripts
src/*	Contains implementation in Rust **main.rs -** Entry point to program **search.rs -** Core search logic **search_helpers.rs** - Helper structs for search logic **url_helpers.rs** - Helper structs and functions for network requests
test_site1/ test_site2/	Test sites to run on localhost - used for some test cases (see site.sh)
test.sh	Given test case name(s), runs case from input/<case>.in and compares with reference output at output/<case>.in
gen_out.sh	Given a test case name corresponding to a file in input/, generates reference output and stores it in output/<name>.out
perf.sh	Given test case name and no.of runs, generates performance metrics file at perf/<case>.perf with hyperfine
site.sh	Starts the local test sites from test_site1, test_site2 folders
validation.py	Given test case name(s), validates the corresponding output/<case>.out files to check for correctness - e.g no duplicate paths, no cycles along a path.
run.sh	Helper script to run an input case

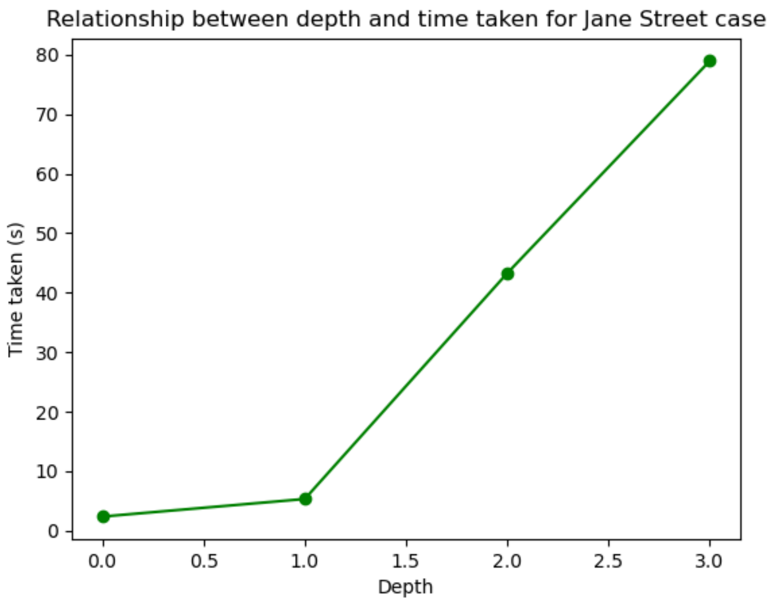
Examples of how to use the scripts can be seen at the README for the project (https://github.com/leonidas1712/findw/blob/main/README.md)

Performance

Since we are using BFS, the expected time complexity of the algorithm is O(b^d) where b is the branching factor (i.e maximum number of child URLs per page) and d is the depth to which we are searching [1].

For a real world case to try, I used this blog article link with various depths and the search pattern ‘Jane’: https://blog.janestreet.com/what-the-interns-have-wrought-2023/

(Relevant cases and logs can be seen in jane0, jane, jane2, jane3.under input/, perf/)



Measurements were taken on a Macbook Pro with an M1 Max processor with 10 cores, each with a max clock speed of 3.2GHz and 32GB RAM.

I tried a depth of 4 as well but found that the time taken did not increase significantly after depth = 3, possibly because there were not many more links due to cycle avoidance or reaching external links at that point.

We can see that there is something similar to an exponential increase in time taken as expected from theory, though it does not map cleanly to an exponential curve due to the differences in branching factors across pages once we account for duplication and cycle avoidance.

Future work

Some future work to expand on this project could include the following:

- **Realtime BFS order:** The current implementation prints paths without any particular order. However, it could be useful to print paths first at 1 click away, then all at 2 clicks away, and so on. This could help if the user does not know what max depth they really want and instead wants to know what the pages of 1,2,3…etc. clicks away look like for analysis purposes.
- **First match:** This feature would involve stopping once the first match (i.e shortest path) to any page with the given pattern is found - an optimal solution. However, this is challenging as the theoretical first match may not be the same as that found by the program due to network requests - e.g a page may be 1 click away but found only later than something 2 clicks away because it took longer for the network request.
- **Regex support:** The current implementation simply does a substring check for the pattern - we could support regular expressions to make the pattern feature more powerful
- **A* search using text similarity:** An alternative to BFS is to use A* search, which employs a heuristic for the number of steps left to reach a goal state [1, p.93]. This is challenging as we would need to employ text similarity and convert that into a suitable heuristic.

Conclusion

In conclusion, findw tries to address the need for a parallel and recursive adaptation of the Unix find command tailored for web pages that is relatively simple to use.

The utility of the traditional find command, which navigates file directory hierarchies, is extended to the web, allowing users to discover pages containing specific text patterns in their titles from a source page.

The Rust implementation works on some engineered (test_site1,2) and real test cases, and the repository includes a variety of scripts to aid testing. In the process of building this tool, I learned a lot more about parallel computing, networking, and software engineering practices when developing scripts and strategies to check for correctness.

I hope to extend this implementation with the future work outlined earlier to make it more robust and useful - with the aim of making it a publicly usable ‘crate’ or Rust tool / library in the future.

References

[1] S. J. Russell and P. Norvig, “Solving Problems by Searching,” in Artificial Intelligence: A modern approach, Harlow: Pearson, 2016, pp. 82–93

[2] J. H. Reif, “Depth-first search is inherently sequential,” Information Processing Letters, vol. 20, no. 5, pp. 229–234, 1985. doi:10.1016/0020-0190(85)90024-9