CS51 PROBLEM SET 5:
AskShiebs

This problem set is due March 29, 2017 at 5:00pm EST.

You may work with a partner on this problem set. If you choose to work with a partner, you can find instructions for configuring GitHub Classroom here.

0. Introduction

0.1. **Description.** The goal of this problem set is to build a small web search engine called AskShiebs. *You will be able (and encouraged) to work with a partner on this problem set.*

We've provided a working skeleton of the search engine web server. Your job is to implement some key components of AskShiebs in three stages:

(1) First, you will implement a *web crawler*, which retrieves web pages by following links from page to page, building up an *index* of the pages as it crawls. The index stores a mapping from words appearing on the pages to the pages that they appear on.
(2) The index makes use of the naive default implementation of dictionaries that we've provided. But this implementation doesn't scale very well, so you'll re-implement sets using more efficient abstractions.
(3) In order to assess the benefit of the new implementation, you will design, develop, test, and deploy a performance testing system for timing the performance of the search engine, both in its crawling and query evaluation. Unlike in previous problem sets (and in previous parts of this problem set), we provide no skeleton code to carry out this part of the problem set. Part of the point of this problem set is to allow you the freedom to experiment with *ab initio* design. The deliverable for this part of the problem set, in addition to any code that you write, is a report on the relative performance of the two implementations.

As a further completely optional challenge, you can follow our walkthroughs on how to implement two versions of Google's PageRank algorithm which the server can use to sort the links returned for a query so that more "important" links are returned first.

Date: March 13, 2017.

0.2. **Testing explanation.** Testing your code is required. You will be using a similar method for testing as you did on the previous problem set. All the functors you write will have a `run_tests : unit -> bool` function. See the examples of tests in `dict.ml` (the tests for `remove`), and see how to run them by looking at the very bottom of `dict.ml`.

For testing the crawler, we have provided three sample dumps of webpages.

- `simple-html`: a directory containing a small list (8 pages) of pages to test your crawler.
- `html`: a directory containing a medium-sized list (20 pages) of pages (the OCaml manual).
- `wiki`: a directory containing a large list (224 pages) of pages to test your crawler, once you have finished implementing your sets and dictionaries. These pages have been scraped from Wikipedia, using `http://en.wikipedia.org/wiki/Teenage_Mutant_Ninja_Turtles` as the start point. **Note: Even with optimized dictionaries and sets, indexing all the pages (224) in the `wiki` directory may take several minutes.**

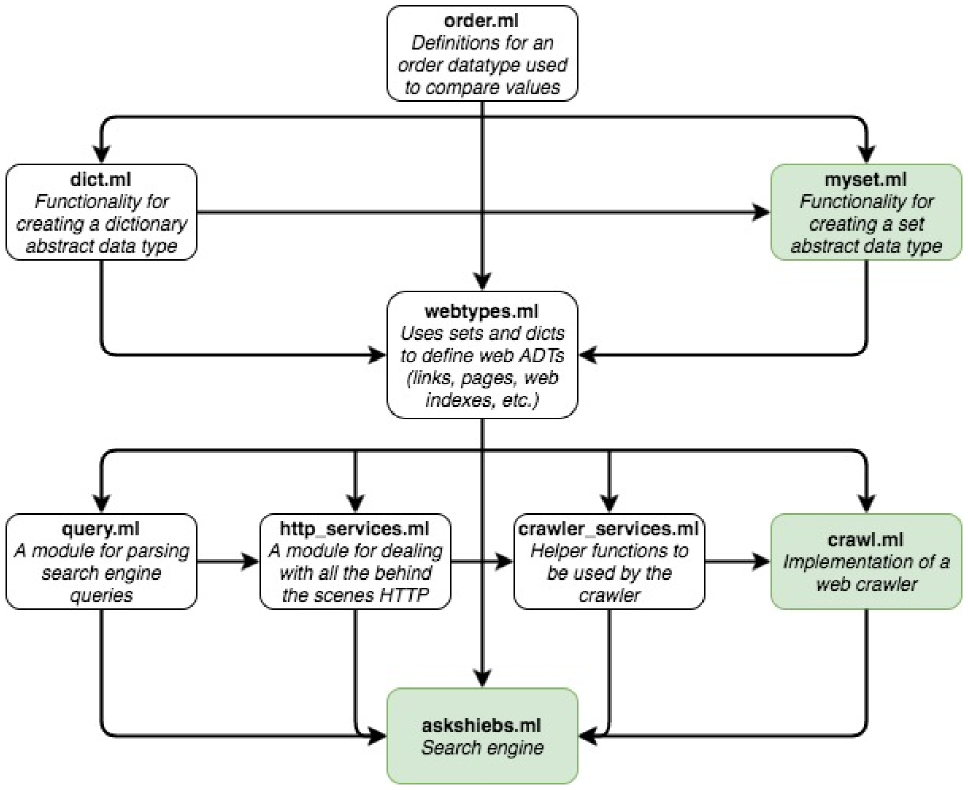
0.3. **Words of warning.** Please heed these three important words of warning.

(1) This project may present new difficulties as compared to previous problem sets because you need to read and understand all of the code that we have given you, as well as write new code. Consequently, we highly recommend you start as early as possible. This is not the kind of problem set you can expect to complete at the last minute.

(2) Once you've completed the assignment, you'll have implemented a web crawler, which you might be inclined to let loose on the open web. However, you shouldn't just point AskShiebs at an arbitrary web site and start crawling it unless you understand the robots.txt protocol. This protocol lets web sites tell crawlers (like Google's or Yahoo's or Bing's) which subdirectories they are allowed to index, at what rate they can connect to the server, how frequently they can index the material, etc. If you don't follow this protocol, then you are likely to have some very angry people on your doorsteps. So to be on the safe side, you should restrict your crawling to your local disk.

(3) Setting up and configuring a real web server demands a lot of knowledge about security. In particular, we do not recommend that you poke holes in your machine's firewall so that you can talk to AskShiebs from a remote destination. Rather, you should restrict your interaction to a local machine.

0.4. **Getting started.** After downloading the problem set, you will find a set of `.ml` files that make up the project source. You will also find three directories of web pages that you can use for testing: `simple-html`, `html`, and `wiki`. Below is a brief description of the contents of each file and how they relate. You should only have to make changes to the files marked in green, though you may well need to refer to the others for important functionality for you to use.


order.ml
Definitions for an order datatype used to compare values
dict.ml
Functionality for creating a dictionary abstract data type
myset.ml
Functionality for creating a set abstract data type
webtypes.ml
Uses sets and dicts to define web ADTs (links, pages, web indexes, etc.)
query.ml
A module for parsing search engine queries
http_services.ml
A module for dealing with all the behind the scenes HTTP
crawler_services.ml
Helper functions to be used by the crawler
crawl.ml
Implementation of a web crawler
askshiebs.ml
Search engine

0.5. **Build AskShiebs.** In its initial state, the code that we provide implements a functioning web server. However, it doesn't yet crawl over a set of files and index them, so searching never generates any results. It is nevertheless useful to try it to make sure that everything is working.

Compile AskShiebs via the command line and start it up from a terminal or shell by executing the following commands:

```
$ ocamlbuild askshiebs.byte

$ ./askshiebs.byte 8080 42 simple-html/index.html
```

The first command line argument (8080) represents the *port* that your AskShiebs server listens to. Unless you know what you are doing, you should leave it as 8080.

The second command line argument (42) represents the *number of pages to index*. AskShiebs will index no more than that many pages.

The final command line argument (simple-html/index.html) indicates the *page from which your crawler should start.* AskShiebs will only index pages that are on your local file system (inside the `simple-html`, `html`, or `wiki` directories).

You should see that the server starts and then prints some debugging information ending with the lines:

```
Starting AskShiebs on port 8080.
Press Ctrl-c to terminate AskShiebs.
```

Now try to connect to AskShiebs with your web browser. Open up a browser and type in the following URL:

```
$ http://localhost:8080
```

You may need to try a different port (e.g., 8081, 8082, etc.) to find a free port.

Once you connect to the server, you should see a web page with a search bar and the AskShiebs logo. You can try to type in a query and if you do, you'll see an empty response. There are known problems with trying to view AskShiebs from Safari. If you experience these problems, try another web browser.

The AskShiebs page lets you enter a search query. The query is either a word (e.g., "functor"), a conjunctive query (e.g., "functor AND queue"), or a disjunctive query (e.g., "functor OR queue"). By default, queries are treated as conjunctive so if you write "functor queue", you should get back all of the pages that have both the word "functor" and the word "queue".

The query is parsed and sent back to AskShiebs as an http request. At this point, AskShiebs computes the set of URLs (drawn from those pages it has indexed) whose web pages satisfy the query. For instance, if we have a query "functor OR queue", then AskShiebs would compute the set of URLs of all pages that contain either "functor" or "queue".

After computing the set of URLs that satisfy the user's query, AskShiebs builds an html page response and sends the page back to the user to display in their web-browser.

1. Implementing the crawler

Your first task is to implement the web crawler and build the link index. The link index is a *dictionary*, a finite mapping that maps each word found on the crawled pages to the set of pages on which it occurs. The dummy `crawl` function in the file `crawl.ml` that we provide just returns an empty dictionary. You will need to replace it with a proper implementation that crawls all of the pages and builds an appropriate `Webtypes.LinkIndex.dict` dictionary mapping the words on the pages to the sets of links.

You will find the definitions in the `Crawler_services` module (in the file `crawler_services.mli` and `.ml`) useful. For instance, you should use the function `Crawler_services.get_page` to fetch a page given a link. The data returned, of type `Webtypes.page option`, contains the URL for the page, a list of links that occur on that page, and a list of words that occur on that page. You need to update your `Webtypes.LinkIndex.dict` so that it maps each word on the page to a set that includes the page's URL. Then you need to continue crawling the other links on the page recursively. Of course, you need to figure out how to avoid an infinite loop when one page links to another and vice versa, and for efficiency, you should only visit a page at most once.

Notice how the variable `num_pages_to_search` specifies how many unique pages you should crawl. So you'll want to stop crawling after you've seen that number of pages (or if you run out of links to process first).

The module `Webtypes.LinkIndex` provides operations for building and manipulating dictionaries mapping words (strings) to sets of links. It is implemented in `webtypes.ml` by calling a functor and passing it an argument where keys are defined to be strings, and values are defined to be of type `LinkSet.set`. The interface for `Webtypes.LinkIndex` can be found in `dict.ml`.

The module `Webtypes.LinkSet` is also defined in `webtypes.ml` and like `Webtypes.WordDict`, it is built using a set functor, where the element type is specified to be a `link`. The interface for `LinkSet` can be found in the `myset.ml` file.

Running the crawler in the top-level loop won't really be possible, so to test and debug your crawler, you will want to compile via command line, and add code that prints things out. See the OCaml documentation for the `Printf.printf` functions for how to do this, and note that all of our abstract types (sets, dictionaries, etc.) provide operations for converting values to strings for easy printing.

Once you are confident your crawler works, run it on the small html directory:

```
$ ./askshiebs.byte 8080 7 simple-html/index.html
```

The `simple-html` corpus contains several very small html files that you can inspect yourself, and you should compare that against the output of your crawler.

If you attempt to run your crawler on the larger sets of pages, you may notice that your crawler takes a very long time to build up your index. The dummy list implementations of dictionaries and sets that we provide do not scale very well.

Note: Unless you do a tiny bit of extra work, your index will be case sensitive. This is fine, but may be confusing when testing, so keep it in mind.

2. Sets as dictionaries

A *set* is a data structure that stores *keys* where keys cannot be duplicated (unlike a *list* which may store duplicates). AskShiebs uses lots of sets. For example, the link index maps words to sets of URLs, and the query evaluator manipulates sets of URLs. It would be useful to have sets be faster than the naive list-based implementation we have provided in `myset.ml`. To improve on that implementation, you will build sets of *ordered keys*, keys on which there is provided a total ordering.

As lazy computer scientists, you will want you to use what you already have for dictionaries to build sets. You will write a functor which, when given a `COMPARABLE` module, produces a `SET` module by building and adapting an appropriate `DICT` module (provided in `dict.ml`).

Part of the problem is to figure out how you can implement sets in terms of dictionaries. As a hint, you may want to think about a dictionary, abstractly, as a *set* of (key, value) pairs.

For this part, you will need to uncomment the `DictSet` functor in the file `myset.ml`. The first step is to build a suitable dictionary `D` by calling `Dict.Make`. The key question to figure out is what to pass in for the type definitions of keys and values. Then you need to build the set definitions in terms of the operations provided by `D : DICT`.

You should make sure to write a set of unit tests that exercise your implementation, following the testing explanation in Section 0.2. You must test *all* your functions (except for the string functions).

2.1. **Try your crawler.** Once you have the `DictSet` functor implemented, tested, and working, you should be able to easily change the `Make` functor at the bottom of `myset.ml` so that it uses the `DictSet` functor instead of `ListSet`. Make the change and try out your new crawler on the three corpora. Which implementation is faster? Try the crawler on the larger test set in the `wiki` directory in addition to the `html` and `simple-html` directories with:

```
$ ./askshiebs.byte 8080 42 wiki/Teenage_Mutant_Ninja_Turtles
```

If you are confident everything is working, try changing 42 to 224. (This may take several minutes to crawl and index.) You're done with the second part of the problem set!

3. Testing the performance of the implementations

You now have two possible implementations of sets (`ListSet` and `DictSet`) that can be used throughout the implementation of the crawler and search engine. Write some OCaml (we provide no skeleton or code of any kind; you're on your own) to time the crawling of the three provided corpora with each of the set implementations, as well as testing query performance. There is already a useful function called `time_crawler` in `askshiebs_tests.ml`, and the timing functions in the CS51 module may be helpful too. You may also find the `time` and `gettimeofday` functions in the `Sys` and `Unix` modules useful for this purpose. (In general, we automatically reject submitted code that uses the `Sys` and `Unix` modules but we make an exception for direct calls to these particular functions when explicitly specified with the appropriate module prefix – no global or local opens.)

We recommend that to the extent possible any code that you write for performing your testing not change the structure of code we provide, so that our unit tests still will work. Ideally, all the code for your testing should be in new files, not the ones we provided.

Write up a short report outlining what you discover about the impact of the two implementations on the performance of web crawling and query response.

This portion of the problem set is purposefully underspecified. There is no "right answer" that we are looking for. It is up to you to determine what makes sense to evaluate the performance of the two set implementations. It is up to you to decide how to present your results in clear, well-structured, cogent prose.

You should submit the writeup with the rest of your code as a text file named `writeup.txt` or `writeup.md`. We recommend that you use Markdown format for the writeup. (See here for why.) There are many tools for writing and rendering Markdown files. Some of our favorites are: ByWord, Marked, MultiMarkdown Composer, Sublime Text, and the Swiss army knife of file format conversion tools, pandoc. If you use Markdown, you can optionally include the rendered version as a PDF file in your submission.

4. Submission

Before submitting, please estimate how much time you and your partner each spent on each section of the problem set by editing the line in each file that looks like

```
let minutes_spent_on_part () : int = failwith "not provided" ;;
```

to replace the value of the function with an approximate estimate of how long (in minutes) the part took you to complete. If you and your partner spent different amounts of time on a part, provide the average of the two. You'll find *all* of these lines for all three parts of the problem set in `askshiebs.ml`.

Make sure your code still compiles. Then, to submit the problem set, follow the Gradescope instructions found here. Don't forget to submit the writeup as well.

If you find yourself with extra time on your hands, you may want to try the challenge problem in the next section and resubmit after. Otherwise, that's it for Problem Set 5!

5. Challenge Problem: PageRank

Only do this part if you know for sure the rest of your problem set is working and you have done an exemplary job on the performance writeup. You may do one or both of these rankers for fun!

We will now apply our knowledge of ADTs and graphs to explore solutions to a compelling problem: finding "important" nodes in graphs like the Internet, or the set of pages that you're crawling with AskShiebs.

The concept of assigning a measure of importance to nodes is very useful in designing search algorithms, such as those that many popular search engines rely on. Early search engines often ranked the relevance of pages based on the number of times that search terms appeared in the pages. However, it was easy for spammers to game this system by including popular search terms many times, propelling their results to the top of the list.

When you enter a search query, you really want the important pages: the ones with valuable information, a property often reflected in the quantity and quality of other pages linking to them. Better algorithms were eventually developed that took into account the relationships between web pages, as determined by links. (For more about the history of search engines, you can check out this page.) These relationships can be represented nicely by a graph structure, which is what we'll be using here.

5.0.1. *NodeScore ADT.* Throughout the assignment, we'll want to maintain associations of graph nodes to their importance, or "NodeScore": a value between 0 (completely unimportant) and 1 (the only important node in the graph).

In order to assign NodeScores to the nodes in a graph, we've provided a module with an implementation of an ADT, `NODE_SCORE`, to hold such associations. The module makes it easy to create, modify, normalize (to sum to 1), and display NodeScores. You can find the module signature and implementation in `nodescore.ml`.

5.0.2. *NodeScore Algorithms.* In this section, you'll implement a series of NodeScore algorithms in different modules: that is, functions `rank` that take a graph and return a `node_score_map` on it. As an example, we've implemented a trivial NodeScore algorithm in the `IndegreeRanker` module that gives all nodes a score equal to the number of incoming edges.

5.0.3. *Random Walk Ranker, or Sisyphus walks the web.* In this section, we will walk you through creating a robust ranking algorithm based on random walks. The writeup goes through several steps, and we encourage you to do your implementation in that order. However, you will only submit the final version.

5.1. **Challenge 1.** You may realize that we need a better way of saying that nodes are popular or unpopular. In particular, we need a method that considers global properties of the graph and not just edges adjacent to or near the nodes being ranked. For example, there could be an extremely relevant webpage that is several nodes removed from the node we start at. That node might normally fare pretty low on our ranking system, but perhaps it should be higher based on there being a high probability that the node could be reached when browsing around on the internet.

So consider Sisyphus, doomed to crawl the Web for eternity: or more specifically, doomed to start at some arbitrary page, and follow links randomly. (We assume for the moment that the Web is strongly connected and that every page has at least one outgoing link, unrealistic assumptions that we will return to address soon.)

Let's say Sisyphus can take `k` steps after starting from a random node. We design a system to determine nodescores based off how likely Sisyphus reaches a certain page. In other words, we ask: where will Sisyphus spend most of his time?

5.1.1. *Step 1.* Implement `rank` in the `RandomWalkRanker` functor in pagerank.ml, which takes a graph, and returns a normalized nodescore on the graph where the score for each node is proportional to the number of times Sisyphus visits the node in `num_steps` steps, starting from a random node. For now, your function may raise an exception if some node has no outgoing edges. Note that `num_steps` is passed in as part of the functor parameter `P`.

You may find the library function `Random.int` useful, and may want to write some helper functions to pick random elements from lists (don't forget about the empty list case).

5.1.2. *Step 2.* Our Sisyphean ranking algorithm does better at identifying important nodes according to their global popularity, rather than being fooled by local properties of the graph. But what about the error condition we mentioned above: that a node might not have any outgoing edges? In this case, Sisyphus has reached the end of the Internet. What should we do? A good solution is to jump to some random page and surf on from there.

Modify your rank function so that that instead of raising an error at the end of the Internet, it has Sisyphus jump to a random node.

5.1.3. *Step 3.* This should work better. But what if there are very few pages that have no outgoing edges - or worse, what if there is a set of vertices with no outgoing edges to the rest of the graph, but there are still edges between vertices in the set? Sisyphus would be stuck there forever, and that set would win the node popularity game hands down, just by having no outside links. What we'd really like to model is the fact that Sisyphus doesn't have an unlimited attention span, and starts to get jumpy every now and then. . .

Modify your rank function so that if the module parameter `P.do_random_jumps` is `Some alpha`, then with probability `alpha` at every step, Sisyphus will jump to a random node in the graph, regardless of whether the current node has outgoing edges. (If the current node has no outgoing edges, Sisyphus should still jump to a random node in the graph, as before.)

Don't forget to add some testing code. As stated in "Testing" section below, possible approaches include just running it by hand a lot of times and verifying that the results seem reasonable, or writing an approx-equal function to compare the result of a many-step run with what you'd expect to find.

Submit your final version of `RandomWalkRanker`.

5.1.4. *QuantumRanker.* Our algorithm so far works pretty well, but on a huge graph it would take a long time to find all of the hard-to-get-to nodes. (Especially when new nodes are being added all the time. . .)

5.2. **Challenge 2.** We'll need to adjust our algorithm somewhat. In particular, let's suppose Sisyphus is bitten by a radioactive eigenvalue, giving him the power to subdivide himself arbitrarily and send parts of himself off to multiple different nodes at once. We have him start evenly spread out among all the nodes. Then, from each of these nodes, the pieces of Sisyphus that start there will propagate outwards along the graph, dividing themselves evenly among all outgoing edges of that node.

So, let's say that at the start of a step, we have some fraction q of Sisyphus at a node, and that node has 3 outgoing edges. Then q/3 of Sisyphus will propagate outwards from that node along each edge. This will mean that nodes with a lot of value will make their neighbors significantly more important at each timestep, and also that in order to be important, a node must have a large number of incoming edges continually feeding it importance.

Thus, our basic algorithm takes an existing graph and NodeScore, and updates the NodeScore by propagating all of the value at each node to all of its neighbors. However, there's one wrinkle: we want to include some mechanism to simulate random jumping. The way that we do this is to use a parameter `alpha`, similarly to what we did in exercise 2. At each step, each node propagates a fraction `1-alpha` of its value to its neighbors as described above, but also a fraction `alpha` of its value to all nodes in the graph. This will ensure that every node in the graph is always getting some small amount of value, so that we never completely abandon nodes that are hard to reach.

We can model this fairly simply. If each node distributes alpha times its value to all nodes at each timestep, then at each timestep each node accrues (alpha/n) times the overall value in this manner. Thus, we can model this effect by having the base NodeScore at each timestep give (alpha/n) to every node.

That gives us our base case for each timestep of our quantized Sisyphus algorithm. What else do we need to do at each timestep? Well, as explained above, we need to distribute the value at each node to all of its neighbors.

5.2.1. *Step 1.* Write a function `propagate_weight` that takes a node, a graph, a NodeScore that it's building up, and the NodeScore from the previous timestep, and returns the new NodeScore resulting from distributing the weight that the given node had in the old NodeScore to all of its neighbors. Remember that only (1-alpha) of the NodeScore gets distributed among the neighbors (`alpha` of the weight was distributed evenly to all nodes in the base case). A value for alpha is passed in to the functor as part of the P parameter. **Note:** To handle the case of nodes that have no outgoing edges, assume that each node has an implicit edge to itself.

5.2.2. *Step 2.* Now we have all the components we need. Implement the `rank` function in the `QuantumRanker` module. It should return the NodeScore resulting from applying the updating procedures described above. The number of timesteps to simulate is passed in to the functor as part of the P parameter.

Don't forget to add some tests!

5.3. **Using your page rankers.** At the bottom of `pagerank.ml` is a definition of an `EngineRanker` module that's used to order the search results. Replace the call to the `IndegreeRanker` functor with calls to the other rankers you wrote, and experiment with how it changes the results.

5.3.1. *Fun fact.* You know by now that PageRank assigns an importance value to every page on the web in order to rank it. But did you know the naming of the algorithm has nothing to do with pages, and it's actually named after Larry Page?