Dijkstra's Algorithm on a Graph of Wikipedia Pages

Daniel Leichus, Jason Shen, Jane Chen

Our Awesome Demo Video!

Overview

Dijkstra's algorithm calculates the shortest path between two nodes in a graph. We implemented Dijkstra's algorithm on a graph of Wikipedia pages in order to find the smallest amount of clicks needed to get from one Wikipedia page to another chosen page. We created a graph representation of all the Wikipedia pages where each node was a Wikipedia page, and each directed edge between two nodes represented a link that went from one node to another. Dijkstra's algorithm then runs through this graph and returns the shortest distance between two given pages, as well as the path as a list of the names of pages one would need to click.

Planning

We planned out our project by breaking down the project and thinking about the different things we would need to make this project successful. Here are our initial draft specification and final draft specification. Our initial phase of the planning consisted of some of the more general and basic things like what exactly we wanted the project to do, what language we would use, and what the final product should look like, as noted in the draft specification. The other part of our planning included more of the details about the project. The initial meeting with Joshua really helped us with this. He helped us understand the different parts the project would require, the different data structures we could or should use, how they all connected and how we should go about implementing them. He helped us to realize that the project was composed of four main parts: the graph, heap/queue, file i/o, and Dijkstra's algorithm. Since we had three people in our group, it made sense for each of us to each take one of the first three parts and then work together on Dijkstra's. We planned out our time accordingly, but due to Dan's concussion and other complications, the schedule didn't hold.

Design and Implementation

As mentioned above, our project consisted of four main parts. Here's a bit about the design and implementation of each of those parts.

For our graphs, we wanted to implement them in a way that it would keep track and represent the Wiki pages in addition to the links to other pages. We decided to do so by implementing the graph as a dictionary of keys and values, where the keys are the Wiki pages of type node and the values are lists of (node * int) tuples in which the nodes are the linked pages and the ints are the weights of the edge that is the link. To implement the graph, we at first thought of using the Moogle problem set as a template. But we quickly realized that there were a lot of nuances in the Moogle code that we didn't understand and that reimplementing it to fit our needs would raise some difficulties as well as provide some unnecessary work along the way. With the help of a suggestion by Joshua, we ended up using the Map.Make functor in the OCaml Core.std library by creating a Mapkey that we called StringKey.t (basically a wrapper of a string). By using the functions that already existed in this library, we were able to more easily create the graph than when we tried using Moogle as a template.

We also realized later on that we needed something to keep track the current guesses of the previous node on the minimum path, length of that minimum path as well as whether or not the guess page has been visited by Dijkstra's algorithm. We decided that this something would be a table, which we named LookupTable, (similar to a dictionary but with only one value per key) where each key is a page and each value contains the things listed above as a triple: (int * StringKey.t * bool). We placed the module for the LookupTable in the same file as the graph.

To get the actual pages with which we would construct the graph, we got database dumps of Simple English Wikipedia, which is a Wiki about $1/50^{th}$ the size of Wikipedia written using Simple English words. However, there are still around 100,000 articles and 5.4 million links so this is no small project. We used SQL queries to clean and format the database into a form usable by Ocaml and then the Core.Std module In_channel to read in the .txt file into our Graph.

Dijkstra's algorithm also requires a priority queue, something that would tell the algorithm which node would be checked next. We implemented this using an

imperative binary minheap. This minheap used a splay tree which means recently accessed elements are easy to access again. The code for this was inspired by code from Simon Cruanes on Github found at the link here: https://github.com/c-cube/ocaml-containers/blob/master/heap.ml. For our purposes, we added a compare (cmp) function that deconstructs the (node * int tuple) and extract the int so that we only have to compare the ints to determine their place in the queue.

We implemented Dijkstra's algorithm in a pretty standard way except for one slight change. The standard approach involves setting the distance of the source node from itself to 0, placing it in the queue, then taking it out of the queue if the end node is designated to be that same source node. For example, if, using this approach, we wanted to find the least amount of clicks to get from "English" back to itself, Dijkstra's would return a distance of 0. However we changed it so that we initially placed the source node's neighbors in the queue. In addition, we did so prior to entering the Dijkstra's loop. Doing so allowed us to make room for the possibility of the source and destination node being the same. Under this approach, the same example above would be returned with a path of "English, English_language, United_Nations, Elie_Wiesel, The_Gates_of_the_Forest, English" and a distance of 5. We feel that this was a neat and reasonable adaptation to the normal approach of this algorithm.

Reflection

We are extremely satisfied with how our project turned out. We were even more so satisfied with some important "lightbulb" moments that we had.

One such moment was when we decided to make the values in the LookupTable triples instead of tuples. We were at first going to include another separate table to keep track of whether or not nodes were visited by the algorithm, but then we realized we could avoid this extra work by simply adding an extra boolean value to the existing tuple to keep track of this.

Another was when we created the add_edge function. We implemented it in such a way that if the nodes that we wanted to add the edge between didn't already exist, the function would add them, and then add the edge. However, adding a new node

would mean recreating the graph. Instead of creating a completely new graph, and then applying the add_edge function to that graph, we made use of the add_node function we had made to say something like:
add_edge (add_node g ...) … This was a more elegant way to do the same thing and cuts down on the amount of code.

While we were very proud of things like these, we realize that there were things we could do to make improvements.

We could make it so that the graph is saved so the same graph can be used in multiple calls to Dijkstra's algorithm. Currently the exact same graph is being constructed each time dijkstras.native is being run. By getting rid of this overhead, perhaps we could run our algorithm on a graph of the full English Wikipedia in a short amount of time, after the full English Wikipedia is read into a graph, of course.

Also, because we were pressed for time, the testing of our code only involved checking if the outputs of a function in all possible cases matched what we would expect. We didn't take the time to write separate test functions, which would be something we could add to improve on our project.

Advice for Future Students

-Don't wait. The sooner you start working on it, the better your project will be. Also, when you wait, you give time for possible complications to arise, which could really hinder the progress of your project.
-When tackling a large project, find smart, efficient, reasonable ways to divide up the workload.
-Really utilize your TF. There's a reason why you have one. If you come across a part you're really stuck on, the first person you should go to is your TF. Our TF was very helpful with fixing our bugs, bouncing ideas off of, and offering very helpful suggestions. This project would definitely not be possible without him.
-Discuss the project with your partner(s) as often as you can. You should never feel like you don't know the progress or direction that your project is headed towards.

-Really utilize the multiple checkpoints made for you. These are not there only for your TF to keep tabs on you, it’s also for yourself to keep yourself on track to finishing the project by the deadline. Also, take the checkpoints seriously, and don’t complete them just for the sake of completion. The less effort you put in these means more effort you’ll have to put in later.