

CS51 Project 1d: $10^{100-\epsilon}$

Due: Friday, 13 March 2009 at 11:59 PM

Total Points: 43 (including 5 style points)

This is the fourth and final part of Project 1. Having worked with ADTs, graphs and random processes, we will now put them together to explore solutions to a compelling problem: finding “important” nodes in graphs like the Internet, such as the World Wide Web.

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.¹ These relationships can be represented nicely by a graph structure, which is what we’ll be using here.

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, `nodescore`, to hold such associations. `nodescore` has the following interface:

```
[nodescore? (-> any/c boolean?)]
```

¹For more about the history of search engines, you can check out this page: http://en.wikipedia.org/wiki/Search_engine.

```

[ns-new nodescore?]
[ns-empty? (-> nodescore? boolean?)]
[ns->string (-> nodescore? string?)]
[ns-scale (-> number? nodescore? nodescore?)]
[ns-normalize (-> nodescore? nodescore?)]
[ns-nodes (-> nodescore? list?)]
[ns-get-score (-> any/c (and/c nodescore? (not/c ns-empty?)) number?)]
[ns-set-score (-> any/c number? nodescore? nodescore?)]
[ns-add-score (-> any/c number? nodescore? nodescore?)]

```

The `nodescore` structure makes it easy to create, modify, normalize (to sum to 1), and display NodeScores. More detailed documentation can be found in the implementation in `nodescore.ss`.

2 Testing

Testing these algorithms is hard, because most of them involve a lot of randomness. We aren't giving any explicit points for testing on this problem set. However, you will doubtless want to test your code to make sure that it works! For the deterministic algorithm (`in-degree-nodescore`), you can do a small example by hand and then check that your algorithm gives the correct result. However, for the rest of them, you can't do much better than running it a bunch of times for a large number of iterations and checking that it always converges towards something sensible. The graph `tst-graph` in particular is good for testing, since it's fairly clear what the nodescore should be.

We've defined many test graphs for you at the top of the `.scm` file (not reproduced here). Feel free to use any of these for your testing, or to define any other example graphs you'd like to use.

3 NodeScore Algorithms *(22 points total)*

In this section, we'll write a series of NodeScore algorithms: that is, functions that take a graph and return a `nodescore` on it. As an example, we've implemented a trivial NodeScore algorithm (`uniform-nodescore`) that gives all nodes equal score (normalized to sum to 1, of course).

3.1 In-Degree NodeScore

A somewhat less trivial algorithm is the following: a node's score is proportional to the number of nodes linking to it (not including itself). If we apply this to our understanding

of the internet as a graph, this means that a page becomes more relevant the more pages that link to it. This is an improvement over the uniform-nodescore algorithm because the nodescore of a webpage is determined by the value that other webpages place on it by linking to it.

Exercise 1.

22 points

(a) [3 points] Implement `in-degree-nodescore`, which takes a graph and returns a nodescore on the graph giving each node a score proportional to the number of other nodes linking to it (not including itself). The nodescore should be normalized, but may be the zero nodescore if the graph has no non-self edges. You will likely want to follow the template of our example above; think carefully about each of the arguments you pass to the function(s) you call.

This approach works well for many graphs. But suppose some scurvy spammer sets out to subvert the system: all he has to do is set up a bunch of dummy pages to link to the page that he wants to become popular. If that page would typically be found by searching a particularly competitive search term, such as "cars" or "hotels", now the spammer's page, which has many other pages linking to it, would suddenly seem extraordinarily relevant and shoot up in the rankings.

What we'd really like is a way to conclude that those dummy pages really are dummy pages, and don't signify that a lot of people value the spammer's page. That seems easy, since no one links to the dummy pages. So what if we don't count an edge if its source node itself is not linked to?

(b) [1 point] Explain how the spammer still wins. Arrr!

3.2 Sisyphean NodeScore

We need a better way of saying that nodes are popular or unpopular, considering global properties of the graph, and not just edges adjacent to or near the nodes in question. For example, there could be an extremely relevant webpage that is several nodes removed from the node we start at. That node would 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 (choosing according to their weights).² Let's say Sisyphus can take k steps after starting from a random node - how could we design a system to determine nodescores that is based off how likely it is that he

²We assume the Web is strongly connected and that every page has an outgoing link, unrealistic assumptions that we will return to address later.

will reach a certain page? Hint: by asking, where will Sisyphus spend most of his time?

(c) [8 points] Implement `sisyphus-nodescore`, which takes a graph and a number k , 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 k steps, starting from a random node. Your function may error if some node has no outgoing edges.

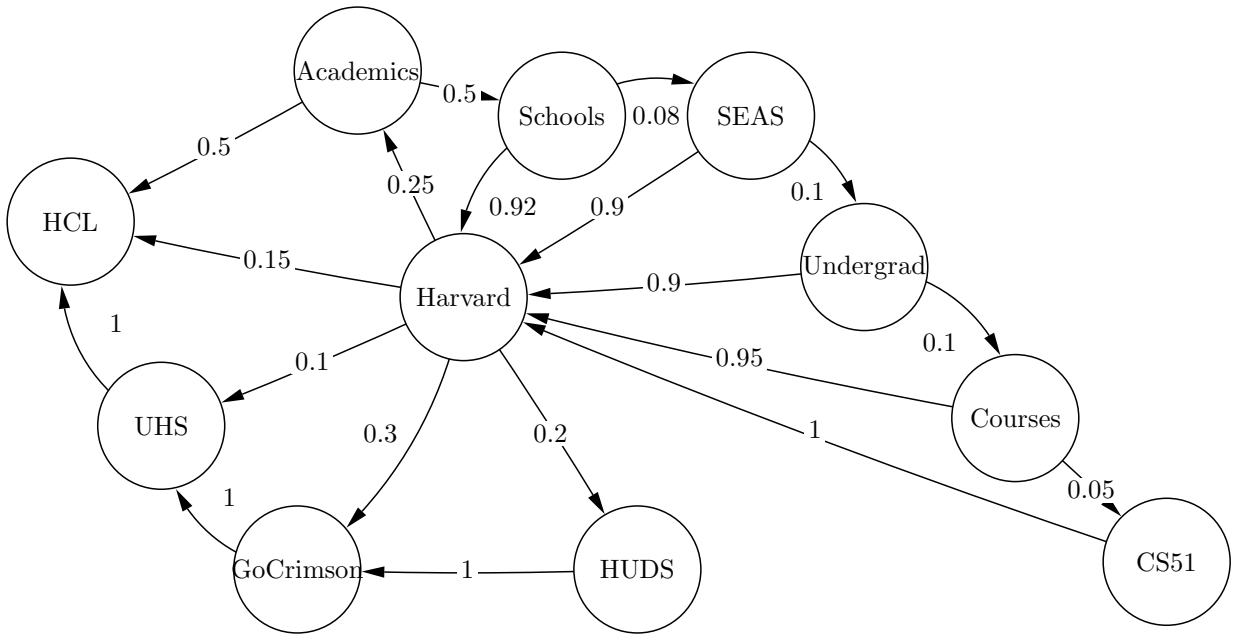
You will find the code you wrote for your babbler to pick a random outgoing edge from a node to be useful. You will have to modify it slightly to work with edges with fractional weights. The `random` function, which takes no parameters and returns a random number in the range $(0, 1)$, will be useful. (Note that `(random n)` will return a random *integer* between 0 and n).

3.3 Some Improvements

Our Sisyphean NodeScore 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's left to do but go back to the beginning?

(d) [5 points] Write `sisyphus-end-jump-nodescore`, which is like `sisyphus-nodescore` except that instead of erroring at the end of the Internet, it has Sisyphus jump back to the first node ever visited.

Not crashing on a realistic Web graph is certainly an improvement. But what if we have a graph like this:



In this case, we may never get to the CS51 page on the right, since it's only accessible via those low probability links, and it's very improbable that Sisyphus will traverse the whole length without jumping back to the main part of the graph on the left. So we might give those nodes zero score, when in fact they should have some small but nonzero score. To fix this, instead of jumping back to the beginning upon reaching a node with no outgoing edges, let's jump to a random node.

(e) [2 points] Write `sisyphus-end-rand-nodescore`, which is like `sisyphus-nodescore` except that instead of erroring at the end of the Internet, it has Sisyphus jump back to a random node.

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...

(f) [3 points] Write `sisyphus-jumpy-nodescore`, which is like `sisyphus-end-rand-nodescore` except that with some small probability `alpha`, Sisyphus will jump randomly to any node in the graph uniformly, regardless of whether the current node has outgoing edges. (Note that if the current node has no outgoing edges, Sisyphus should still jump to a random node in the graph, as before.)

4 SuperNodeScore

(16 points total)

Our algorithm so far works pretty well, but on a huge graph it would take a long time to explore all of the small-probability nodes.³

So we'll need to adjust our algorithm somewhat. In particular, let's suppose Sisyphus is bitten by a radioactive eigenvalue,⁴ 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 among all outgoing edges of that node according to the weight of each edge.

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, with weights 0.2, 0.3, and 0.5. Then $0.2q$ of Sisyphus will propagate outwards from that node along the edge of weight 0.2, $0.3q$ along the edge of weight 0.3, and $q/2$ along the edge of weight 0.5. 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 pass a parameter α , similarly to what we did at the end of exercise 1. At each step, each node propagates a fraction $(1-\alpha)$ of its value to its neighbors as described above, but also a fraction α 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 α times its value to all nodes at each timestep, then at each timestep each node accrues (α/n) times the overall value in this manner. Thus, we can model this effect by having the base NodeScore at each timestep give (α/n) to every node.

For all complexity calculations on the remainder of this assignment, you may make the (unrealistic) assumption that `graph-outgoing-edges` takes constant time, or $O(1)$.

Exercise 2.

16 points

(a) [2 points] Write a function `fixed-nodescore` that takes a graph and a value, and returns a NodeScore where the value of each node in the graph is the value given.

³Especially when new nodes are being added all the time...

⁴Represented by λ . Coincidence?

Now we have 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, based on the edge weights.

(b) [6 points] Write a function `propagate-weight` that takes a node, a graph, a value of alpha, 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. You might find code from your babblor useful in doing the appropriate weighting. Remember that only $(1-\alpha)$ of the NodeScore gets distributed among the neighbors (the other alpha was distributed evenly to all nodes in the base case). Your function should take time linear in the number of edges outgoing from the given node in the given graph (making the assumption stated above, that `graph-outgoing-edges` is constant-time).

(c) [5 points] Now we have all the components we need. Write a function `quantum-nodescore` that takes a graph, a number of timesteps to simulate, and a value of alpha to use as the distributing parameter. It should return the NodeScore resulting from applying the updating procedures described above. Your function must run in $O(k*e)$, where k is the number of timesteps to run and e is the number of edges in the input graph. (Hint: As explained above, your `propagate-weight` function should be linear in the number of edges leaving the input node. Thus, your `quantum-nodescore` function should probably do a fold over what list, in order to make the overall complexity per timestep linear in the total number of edges?)

(d) [3 points] Prove that your `quantum-nodescore` function runs in $O(k*e)$. You may need to prove the Big-O runtime of your `propagate-weight` function in order to do this.