

CS51 Assignment 4: Asymptotic Performance

Due: Friday, March 6th, 2009, 5:00PM

Total Points: 45

In this assignment, you will compare and analyze the running times of various graph-traversal algorithms. We've provided modules with code for breadth first search and depth first search in an update to `cs51.plt`, which you should be prompted to install when you restart DrScheme. We've also provide various utility functions for timing and plotting your results. The timing and plotting code is in `ps4.ss`. That should be the only file you modify and hand in. You will not have to write much code for this assignment—instead, you'll have to read our code, evaluate how well it works, and describe what you found.

1 Evaluating BFS, DFS

(44 points total)

As suggested in lecture, there are performance trade-offs between DFS and BFS for various applications. Depending on the type of graph and the type of search being performed, one may be far better than the other. In this assignment, we'll be looking at the runtime for searching (in order) for each node in the graph.

To get you started, we've written the module `graph-search`, which provides the functions (`bfs some-graph root found?`) and (`dfs some-graph root found?`). `some-graph` is a `graph` object, which you saw in the Babblor project. `root` is the starting node of the search, and `found?` is a predicate that takes a node as input and returns whether or not the node is the one you're looking for. For example, if you were searching for the node '3,' your `found?` function would be (`lambda (some-node) (equal? 3 some-node)`).

`bfs` and `dfs` return the desired node if it is found, and `#f` otherwise.

Exercise 1.

3 points

Using the functions provided by the `graph` module, make two small graphs. With both `dfs` and `bfs`, search each graph for a value other than the starting node, and ensure that they return the proper value.

1.1 Generating Test Cases

On small graphs, such as those you might construct by manually adding nodes and edges, even the least efficient search algorithm will complete in a few milliseconds. What we are really interested in is the *asymptotic* performance of these algorithms: the increase in running time as the number of nodes approaches infinity.

To generate these large graphs, we've written the module `random-graph`. It provides the function `(make-random-graph n gen-vertex gen-edges)`. `n` is the number of nodes in the graph to generate. `gen-vertex` is a function that takes an integer representing the node index and returns the desired node value. `gen-edges` is a function that returns the number of edges to generate for a particular node. The return value of `make-random-graph` is a graph.

Exercise 2.

2 points

To get familiar with `make-random-graph`, take a look at the examples provided. Write a call to `make-random-graph` to generate a graph with 10 nodes. The value of each node should be a string representing its node ID, and each node should have *exactly two* neighbors. Run BFS or DFS on your graph using the `visitor` function as the `found?` parameter. You may find the `get-a-node` function useful for getting a root to start your traversal from.

1.2 Measuring Running Time

To measure the running time of the provided BFS and DFS implementation, we have provided the function `(time-function f lst)`, which measures the running time of function `f` on each value in `lst`. The return value is a list of vectors. A vector is a fixed-length list, similar to an array in C. We're using vectors because the plotting module requires vector inputs. In this case, each vector contains three elements: `n`, `cpu time`, and `1`. In each vector, `n` is the value passed to `func`, `cpu time` is the amount of CPU time `func` took on input `n`, and `1` represents the "weight" for our curve fitting function (discussed below).

Exercise 3.

2 points

As an example, we've provided a function to recursively compute the fibonacci sequence, which exhibits interesting timing behavior. Uncomment the `time-function` call above to measure this. What do you notice about the running times?

Exercise 4.

7 points

(a) [2 points] Practice using `time-function` by measuring the running time of graph generation. We've provided the functions `make-sparse-graph` and `make-dense-graph`, both of which take a single argument, `n`, and return a graph with `n` nodes.

Time how long it takes to generate *sparse* graphs with 25, 50, 100, 200, 300, and 400 edges.

(b) [2 points]

Time how long it takes to generate *dense* graphs with 25, 50, 100, 200, 300, and 400 edges.

(c) [3 points] What do you notice about the running time of each function as the number of nodes increases? (“It gets bigger” is not a sufficient answer :)

1.3 Visualizing Your Output

For large data sets, it would be nice to have a better method of visualizing running times than simply a long list of numbers. Luckily, PLT Scheme includes a `plot` module. However, the `plot` takes lots of parameters, and uses keyword arguments, which we haven’t seen yet. As such, we’ve written a helper function `plot-times`, which takes a list of vectors like those produced by `time-function` and plots them on a graph with appropriately scaled axes.

Optionally, `plot-times` also takes a *fit function* and a graph title. A fit function is an equation for matching a line to a series of points. We’ve defined the fit functions `fit-dummy`, `fit-linear`, `fit-quadratic`, and `fit-exponential`. `fit-dummy` simply plots a line out of sight at $y=-1$. `fit-linear` describes a linear relationship, attempting to match the data to an equation of the form $y = an + b$. `fit-quadratic` describes a quadratic relationship of the form $y = a * n^2 + b * n + c$. `fit-exponential` describes an equation of the form $y = a * b^n$.

Exercise 5.

2 points

Define a cubic fitting function `fit-cubic` that fits data with a third degree polynomial. This may come in useful in later sections.

Exercise 6.

6 points

(a) [3 points] Using the code provided, analyze the running time of `fib`. Which of the fit functions best describes the asymptotic performance of `fib`? What are the approximate values of the constants?

(b) [3 points] Explain, in two sentences or less, what about the code makes it have the observed performance.

(c) [0 points] A small challenge (not for credit), if you’re interested: write a new `fib2` function which computes the same thing as `fib`, but is much much faster.

Exercise 7.

8 points

(a) [2 points] Use the helper functions defined above to plot the times to generate sparse and dense graphs.

(b) [2 points] What do you notice about the comparative running times of generating sparse vs. dense graphs?

(c) [4 points] How would you describe the two algorithms in terms of Big-O notation? Is one asymptotically faster than the other?

Exercise 8.*14 points*

(a) [4 points] We've defined the helper function (`run-search search graphs`), which returns a *function* that takes a single value `n`, looks up a graph of size `n` in `graphs`, and performs the specified `search` for every node in the graph. We've defined two lists of graphs, `sparse-graphs` and `dense-graphs`, for you to use.

Uncomment them, and then plot the running times of `run-search` for `bfs` and `dfs` on sparse and dense graphs (in other words, you should have four graphs).

(b) [3 points] Which of the provided fit functions best describes the asymptotic performance of these algorithms?

(c) [3 points] What are the approximate values of the constants? (You can either modify the code to print out the constants it estimates, or estimate them from the graphs)

(d) [4 points] Is one algorithm faster than the other? Why do you think that is? Under what circumstances might the results change?

Exercise 9.*0 points*

A CHALLENGE: look over our implementation of graphs and the search algorithms and try to think of ways they can improved. Where are there inefficiencies? Would fixing them result in a constant factor improvement, or asymptotic gains? For an extra hacker points, actually rewrite the implementation so it is faster. Describe what you changed, and include a graph of your improved version's performance. Make sure it still works correctly!

2 Finishing Up

*(1 point total)***Exercise 10.***1 point*

Please tell us how much time you spent on this problem set! You may use this space to add any thoughts or questions you want to send to your TF as well, if you so desire.