Sam Beluch, Luke Schroeder

Professor Ames

Principles of Programming Languages

Friday, May 10th, 2019

Analysis of Dijkstras Algorithm in Haskell, Racket, C, and Python

For our final project, we proposed to implement a famous computer science algorithm in four different languages: Python, C, Haskell, and Racket (Scheme). Python and C are imperative languages, with the former being odynamic and the latter static. Haskell and Racket are both functional programming languages, with Haskell being static and Racket dynamic. In essence, we tested the implementation, timing, and complexity of the algorithm in the four different types of languages.

Both of us knew Dijkstra's algorithm well, as we have been through Data Structures, Design and Analysis of Computer Algorithms, and other classes that used or at least mentioned it. So, we made sure to code the algorithm ourselves, without looking up or using outside code. All four programs take in the files "*graph.txt"*, which must be in the same directory as the program being run. This is the data file, which specifies the amount of nodes, the names of them, and all edges with their weights in successive lines.

Haskell

<u>Design</u>

In Haskell we implemented Dijkstra's using the Data.Graph class. This class provides methods that allow users to generate a graph from a properly formatted list of vertices and edges. Our main function parses the input file for these properly formatted lists. We ensured that all the impure, IO monad based code was limited to our main function to adhere as much as possible to the purity of the language. In main we use several list comprehensions such as

```
[ (i - 2, ( words file ) !! i )  |  i <- [2 .. 1 + numberofnodes ] ]
```

which parses the input file for a list of tuples formatted as (<vertex index>, <vertex name>) and the more complex [(vertices !! i , Dijkstras i graph) | i <- [0 .. nvertices - 1]] . This generates a list of tuples where the first element is a vertex of the form denoted by the previous list constructor and the second element is a list of all the shortest paths of Dijkstra’s applied with that vertex as the source. Our Dijkstra’s function initializes the minimum distance to source as 0 and the rest of the distances as the maxBound on the Int type in Haskell it also initializes a list of all the vertices in the graph. It then calls the recursive Dijkstras function which recursively operates until the list of unchecked vertices is empty. With each call, the recursivedijkstras function finds the minimum distance to an unchecked node and removes this node from the unchecked list, parses the adjacent elements of the node and updates their distances using the update distance function. The update distance function takes the weight to the current node, a list of pairs of nodes associated with their current shortest path weights, and a list of neighboring vertices and the weights of their edges. It maps a single anonymous function onto the distance list that updates the shortest path edge weights. Here (a,b) is recursively stripped from the (neighboring vertex, edge weight) list:

```
(\(c,d) -> if (d == b && weight < maxBound && a < maxBound && a + weight < c)
            then (a + weight, d)
            else (c,d)).
```

Analysis

- Time/Space Complexity

        - Assume V vertices and E edges as our input size of recursedijktras which carries out the majority of the computation. We call this function recursively V times for each unchecked vertex. Within this function we call getminweightandindex which calls sort on lists of length V, V-1, .. of all the unchecked indexes. So we have O(Vlog V + V-1 log V-1 + ..) If we had implemented this as a min heap, which is difficult to do in a pure language that lacks mutability without employing outside libraries, we would have O(V) calls * O(1) searches for minimum + the complexities associated with maintaining a heap = O(V). However the function that dominates the time complexities of our program is update distance. For each vertex in the graph this function parses the list of edges connected to a given vertex and for each parses the entire list of vertices to update their distance. Overall, this function is called V times parsing a list of length V and accessing the edges 2E times. Thus our dijkstrasrecurse function overall has a complexity of O(V * V * E) which is extremely inefficient. This too does not take into account calling the function on every vertex as the source, adding a factor of V to any Dijkstras implementation. The space efficiency is O(V + E) in the graph representation provided. For both updatedistance and recursedjikstras we have with every recursive call a new list created of size V, V-1, … 0 which gives a space complexity of O(V^2). This is because Haskell is pure so it creates a new entity with each call.
- Ease of Implementation
    - Pros
        - Haskell is an extremely concise language. Every function is compact and readable and can accomplish many tasks in a single line. List

        comprehensions like the ones detailed in the analysis section are an extremely useful tool for generating precise and complicated arrays of data. For this specific task, graph algorithms lend themselves to recursion and the Data.Graph class is useful to investigate and implement. Functions like map and anonymous functions serve as powerful tools for applying operations across the scope of a graph.

- Cons
    - As a pure language, Haskell discourages mutability thus making a fundamental component of executing graph algorithms difficult. Lists of minimum distances and The Data.Graph class is very ill fitted for Dijkstra's in particular as it stores all information about a given graph in a collection of lists. This is reflected in the pathetic time complexity of the algorithm. The space complexity was worsened by the pure nature of the language creating new entities with every recursive call of certain functions. Even the file parsing required complicated list manipulations rather than a simple for loop.

C

Design

In C we implemented a simple 2-D matrix representation of a graph where each vertex name corresponded to a row and column of the graph. Each positive integer stored in graph[u][v] = graph[v][u] and represents the weight of the undirected edge between vertices indexed u and v. Pairs of vertices with no connecting edge have a value of 0 stored at their corresponding positions in the graph matrix. Our main method parses the input file and does some simple error checking to ensure correct input formulation. It mallocs space for the int graph, the shortest paths matrix that will contain the shortest paths for vertex v in row v, and a list of all the vertices in the graph it then calls populateallshortestpaths which surprisingly populates the contents of the shortest path matrix. Populateallshortestpaths calls dijkstras with each vertex in the graph as a source. Dijkstras then initializes the shortest distance list for the given source and passes this and a list of all unchecked vertices to Iterativedjikstras which populates the corresponding row of the shortestpaths matrix with the updated list of shortest paths from the source.

<u>Analysis</u>

- Time/Space Complexity
    - Assume V vertices and E edges in the graph. Then the graph representation as a matrix has a space complexity of O(V^2) since it has dimensions V*V. Since we implement the matrix representation of a graph, we have a time complexity of O(V^2) which in turn gets multiplied by a factor of V since we are calling Dijkstras on each vertex in the graph. Hence the overall time complexity is O(V^3).
- Ease of Implementation
    - Pros
        - Since C is a statically typed imperative language, all the information about a given variable or function declaration is easily visible. Dynamic allocation of memory is useful in creating data that is accessible throughout the entire scope of the program. Iterative functions allow natural parsing of 1D and 2D arrays. Additionally, I/O is facilitated by functions like fopen and printf that allow for easily customizable output and error checking. In particular, the fscanf function that parses a file, automatically parsing and storing data in properly typed variables, is very useful in stripping information about the nodes of a graph. Mutability is

key in saving space complexity in comparison to pure languages since you can alter and update existing structures rather than create fresh copies repeatedly. It is possible to optimize the space complexity even more by manipulating data down to the bit level. Overall the space and time complexities of the program were superior to those of the Haskell program.

- Cons
    - The C program is more than double the size of the Haskell file by line count. Memory declaration and management, although incredibly optimizable, is cumbersome when dealing with simple tasks like mallocing space for a 2-d array. It lacks the useful list functions present throughout declarative languages. Additionally bugs often result in a segmentation fault, a fatal error that provides no explanation. Overall, C trades bulkiness and complexity of code for customization and optimization.

Python (Dijkstra.py)

<u>Design</u>

Coding Dijkstra's algorithm in python was much easier in Python than in Racket. Reading from a file was straight forward, as we had semesters of experience in Python. As python is dynamically typed, it was extremely easy to store the data, as we could just add to the data as we read without having to explicitly allocate space or state how large the variable will be. Once read, the data was then stored in arrays, with an array for the nodes and a 2D array to represent the graph with the number at each index corresponding to the weight of the edge. This is done with the readFile function. From there, on each node, main calls the Dijkstra function on each node of the graph. This drives the algorithm, and calls findMin and update. FindMin selects the closest neighboring node adjacent to the current selected vertex, and update then goes through and updates the data. Overall, this is done for all vertices and the final data is printed in a V x V array, just as in the C program.

Analysis

- Time/Space Complexity
    - The time/space complexity is essentially identical to that of the C program. For each vertex, up to V other vertices are checked, with the vertex picked by finding the minimum of a V size array. Thus, for each vertex the complexity is O(V^2), for a total of O(V^3) for the whole graph. Space complexity remains at O(V^2), as our graph is represented that way.
- Ease of Implementation
    - Pros
        - Since Python is a dynamic language, reading the data in and storing the graph was extremely simple, even more so than in C. This is because memory did not have to be specifically allocated for each array, as we could just add on to them. Python also easily parses a file line by line into an array with a simple function call and loop. Also, Python is imperative, so through each function call it was easy to return multiple variables with data to use. For example, when returning from update, we could return both the distance to the closest vertex as well as the vertex itself. Overall, Python’s structure and syntax allows for very user-friendly coding, ignoring some efficiency for simplicity and ease of programming.

- Cons
    - Python being dynamic can cause problems. If there is an error in the way the data is stored or read, the function may return a variable that cannot be read correctly by a parent function. Ensuring correct data types is extremely important. Furthermore, Python gives up efficiency by not worrying about memory allocation, as in C. Functions can also become very complex, while in Haskell or Racket the functions stay shorter due to the fact variables are not stored the same way. Overall, for an algorithm like Dijkstra's, we feel as if Python's pros outweigh the cons, and when picking a language for the code Python would be an earlier choice.

Racket (D.rkt)

Design

As a functional language, Racket does not store variables in the same way as Python or C. Instead, one driving function, MainDriver, calls the algorithm on all the vertices read in from the file. From there, all of the sub-functions are called recursively using list iteration, which could be very effective being that we store most of our data in Lists in Python and C anyway.

To perform the algorithm, the program reads in the file into two structures: n, a list of vertices, and g, an undirected graph that uses Racket's graph interface. The graph is stored as two hash-lists, the first being edges and the second being booleans of whether a certain vertex can reach another one. This graph interface gave helpful functions such as has-edge? And edge-weight that could be used when traversing through data.

The mainDriver function iterates through the list of vertices, callin subDriver on each one. This creates a hash list for that node that stores the edge, its distance, and whether its been visited, which is then used in the driver function. This finds the minimum distance neighbor and uses that to update the hash of edges and distances. This is done by using the find and update functions, and continues until all reachable vertices are visited. Once this is done, the list for that node is returned to the mainDriver, which then continues to iterate through the rest before returning a global list of all data.

Side Note: Racket does have a dijkstra function that can be called on a graph implementation. However, we felt this was against the point of our project and made sure not to use it.

<u>Analysis</u>

- Time/Space Complexity
    - Assume V vertices and E edges in the graph. The time complexity in Racket is extremely hard to analyze, as it is very recursive and functional. Each time a vertex is iterated through, new hash-lists are created over and over again as variables are not mutable. Also, when parsing the file and the data, it is not possible to access an index of a list variably, so instead the lists are iterated through over and over. In all though, as the code follows Dijkstra's algorithm, the base does follow the O(V^ 2) time complexity for each vertex, as we did not implement a heap in Racket either. But, for each function, since hash-list is created many times instead of updating a current one, the time depends on backend racket time complexity for the data structure and the graph. And, each of this is done V times, leading to a program no more efficient than Python or C. Also, in terms of space, the space complexity is similar, as the lists are

essentially arrays. However, new lists and hashes are created over and over, leading to a lot of space and time wasted throughout run time.

- Ease of Implementation
    - Pros
        - The graph implementation stored a lot of data easily for us. Also, lists lead to easy ordering of the program, as we know how the lists are iterated through. Using the dijkstra function call would have been extremely easy, but we strayed from that. As a functional language, the code is concise like that of Haskell, as functions tend to be much shorter, with at most a few if statements in each. This leads to easier debugging function by function and step by step.
    - Cons
        - Coding Dijkstra's in Racket seemed unnecessarily complicated. Not being able to store variables led to many more functions to deal with the different data types. It was extremely difficult to trace or picture the data through many function, especially as new functional data types are creatd with every function call. Not being able to access a list by indices instead of iteration also leads to difficulties. Parsing the file was similar to in Haskell, as it had to be done through complicated list functions instead of a simple loop storing variables as in Python or C.

<u>Overall</u>

If we were to code Dijkstra's algorithm, we would pick Python or C. It is an algorithm well suited for imperative languages, as coding the algorithm in those languages took a fraction of the time as in the functional languages. Haskell, Racket, and other functional languages have their perks, and coding something so complex definitely greatly improved our skills and mindset for those languages.

How to Run

Python:

Call [python dijkstra.py] in the command line, with graph.txt in the same directory as the program. It is in python 2.7

Racket:

Run “D.rkt” in DrRacket, with graph.txt in the same directory as the code. The graph implementation in Racket is required as well.

Haskell:

Call ghci dijkstras.hs in the same directory as any graph file. Then call main and type the full name of the graph file you wish to run the program on.

C

Compile with gcc -o dijktras dijkstras.c. Then execute with ./dijkstras <graph filename> With the full graph filename as the sole command line argument.