Data Structures and Graph Algorithms

Gael Medina Matheus Moreira
Mentor: Carlos Alvarado

Abstract

Data Structures and Algorithms are integral parts of our day to day lives. From the GPS in our cars to the routers connecting our computers, algorithms are fundamental. This paper goes over several types of data structures and how they connect to the Dijkstra shortest path algorithm. This paper will also go over how different data structures are connected to one another and how they compare using time complexity.

Contents

1 Introduction

We will be going over various different graph algorithms, how they compare to one another, and how we interact with them in the real world. We will also be covering how these algorithms store information and how different ways of storing information are more or less efficient depending on the information.

We interact with algorithms everyday without even realizing it, and one such example is the GPS. A GPS needs an algorithm so it knows what directions to give, and in this paper we will be going over the base algorithm that a GPS might run on, Dijkstra's algorithm. We will also be explaining the importance of data structures and how they relate to algorithms, and more specifically, the importance of a Fibonacci heap when using Dijkstra's algorithm.

2 Algorithms and Efficiency

2.1 What is an Algorithm?

An algorithm is a set of instructions or rules that detail or explain how to do something. It's like a recipe in the way that it lays a specific set of steps or rules that must be abided by to solve some problem. An algorithm is comparable to a cake recipe. The recipe provides a series of instructions that include things like how to combine the ingredients, how long the cake should bake at a specific temperature, etc.

Definition 2.1 (Algorithm). *Any well-defined computational procedure that takes an input value or set of values and produces an output value or set of values is known as an algorithm. Thus, a sequence of computational steps that convert the input into the output is an algorithm.* [CLRS01].

Another example is how many people multiply. When learning multiplication, many of us learn to use the procedure / algorithm known as long multiplication. We get our two numbers we are multiplying and put them above each other. We then start from the rightmost digit of the bottom number and multiply it by all the digits of the top number and place them below both of our numbers. We then move on to the next right most digit, but this time we add a zero in front and redo our first step. To complete the multiplication using this method, we must follow a clear set of steps that lead to our outcome, which is exactly what an algorithm does. It takes in some input, this case being two numbers, and follows a clear and well-defined set of steps that leads us to some outcome.

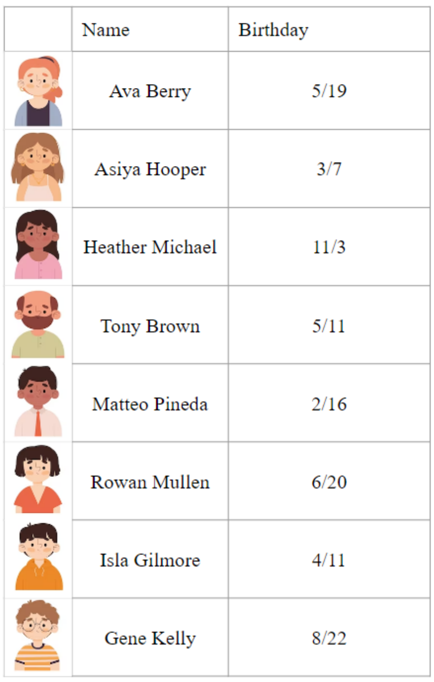
2.2 Algorithm Efficiency

Not all algorithms are as fast as each other, and a good way to compare the quality or speed of two separate algorithms is by checking their efficiency, more importantly how long it takes for the algorithm to run in the worst case scenario. The time it takes for a computer program to complete an algorithm is called the run time. The run time for a computer program can be affected by many different things, such as the size of the data your algorithm is working with and the overall efficiency of your algorithm. When dealing with large amount of information, it is important to figure out what algorithms would work best. Some algorithms might work better than others with less input, but worse with larger input, so having a way to see what algorithms work better over large amounts of information is paramount to choosing whats best for the problem we are solving.

Definition 2.2 (Big O Notation)**.** *Let f and h be functions from the positive integers to the nonnegative real numbers. We say that $h(n)$ is $O(f(n))$ if there exists a positive constant B such that $h(n) \leq Bf(n)$for all sufficiently large n. In this case we say that h grows no faster than f or, equivalently, that f grows at least as fast as h.* [BW13]

When describing an algorithm's worst-case run time, big O notation can be very useful. It gives us a way to group different algorithms based on their efficiency regardless of how large a data set is given. The notation is based on how the time required to solve a problem grows as the input data size grows. The run time of an algorithm can be represented as a function of input size using the big O notation, which gives an upper bound. If we had an algorithm in which the run time grew linearly in respect to the size of the input data, then the algorithm would have a time complexity of O(n). The algorithm's run time would at most double if the size of the input data also doubled. Some of the common run times we encounter O(1) for constant time complexity and $O(logn)$ for logarithmic time complexity. In general, if the worst-case run time of the algorithm is $O(n^k)$ where k is an integer, computer scientists will deem the problem as tractable and the algorithm as "good."

2.3 Explicit Examples



	Name	Birthday
	Ava Berry	5/19
	Asiya Hooper	3/7
	Heather Michael	11/3
	Tony Brown	5/11
	Matteo Pineda	2/16
	Rowan Mullen	6/20
	Isla Gilmore	4/11
	Gene Kelly	8/22

Above is a table with the names of various different people and their birthday days in months. For this example assume they were all born in the same year. Say that you were playing a board game with these people and you had to order them from oldest to youngest to get the proper playing order. How would you do it? There are various algorithms that you or a computer can use to figure this out.

2.3.1 Insertion Sort

One way the computer could sort these would be by going through each person and comparing them with everyone one else on the list, and when it finds someone that has a birthday before the chosen person, the computer would switch their places on the table. This method, while simple, would be very slow. If the computer started with Ava and went down, it would immediately switch Ava with Asiya. Then the computer would compare Asiya with everyone else and it would keep switching places until the table was sorted. In the worst case, the computer would have to compare every person with every other person, meaning that this algorithm runs in $O(n^2)$.

Algorithm 1 Insertion Sort

```
Require: $L$ the list holding our names and birthdays.
  n = length(L)
  for i between 0 and n-1 do
      j = i
      while j > 0 do
          if L[j-1] > L[j] then
              Swap the two
              j = j-1
          end if
      end while
  end for
```

2.3.2 Merge Sort

While insertion sort is definitely a feasible way to sort through a set of data, it isn't really as efficient as it could be. Merge sort is a clever way to sort data by splitting the data into half's. The algorithm starts by splitting the total data into two half's. It continues splitting the data into half's until the program is left with multiple sub lists that only contain 1 element. Since these lists only have 1 element, they are sorted lists. The program then starts grouping these sub lists in groups of 2 and starts comparing the elements within and placing the sorted elements into a new sorted list. Comparing 2 sorted lists runs in $O(n)$ since only one comparison must be made between each element in one sorted list to another. The total number of comparisons to sort the list is proportional to the number of times the list can be divided in half until each sub list has 1 element, which is given by $log_2(n)$. By combining the time it takes to split the data into sub lists and the time it takes to compare elements, we find that Merge Sort runs in $O(nlog_2(n))$.

2.3.3 Counting Sort

A clever way to make a faster algorithm would be to use another list. Instead of just having our initials list with all the people and their birthdays, we could have the computer create another list that held 365 empty spaces for each birthday. The computer would then place the name of the person in their corresponding box. This would allow us to now have an ordered list of all the people and their birthdays. Since the program only has to move each item once, this runs in $O(n)$.

Algorithm 2 Counting Sort

```
Require: L is a list that holds the name of each individual and their birthday
  F = [0]*365        ▷ This means we make a list of length 365 where every entry is 0.
  n = length(L)
  L' = []            ▷ This is an empty list that will hold our order later
  for x between 0 and n-1 do
      i = L[x][1]    ▷ This will hold the birthday of the person
      if F[i] equals 0 then
          F[i] receives L[x][0]
      end if
      Else:    ▷ If two people have the same birthday, then their names will be added together
      F[i] receives F[i]+", "+L[x][0]
      EndElse
  end for
  for j between 0 and the length of F do
      If F[j] is not empty:
      add F[j] to L'
  end for
```

3 Data Structures

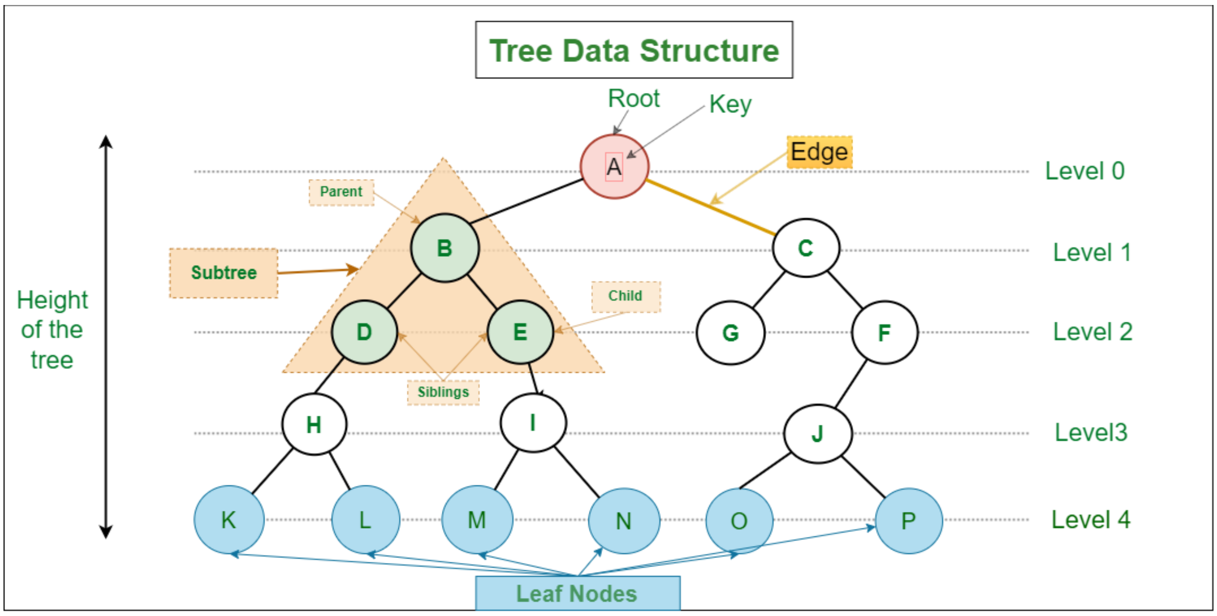
3.1 What are Data Structures?

Definition 3.1. *A data structure is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them.* [CLRS01]

Data structures are ways of organizing information that allow the data stored to be easily accessed and used. Some data structures are easier to make, while others are more difficult but are faster. There are various different types of data structures ranging from arrays to trees to heaps, and they all vary in how to make them and also how they work.

3.2 Why use them?

Data structures have a wide range of uses and are extremely helpful throughout our day to day. Think about your various social media platforms. In almost all of them you have your personal profile and then you have a list of friends you have added or something along those lines like followers. Something needs to keep track of this information, and that is where a data structure like a list or a graph would be helpful. Lists and graphs would store all of that information neatly, and a graph or a tree would allow one to see connections between information, like weather or not someone is following you.



3.3 Useful Structures

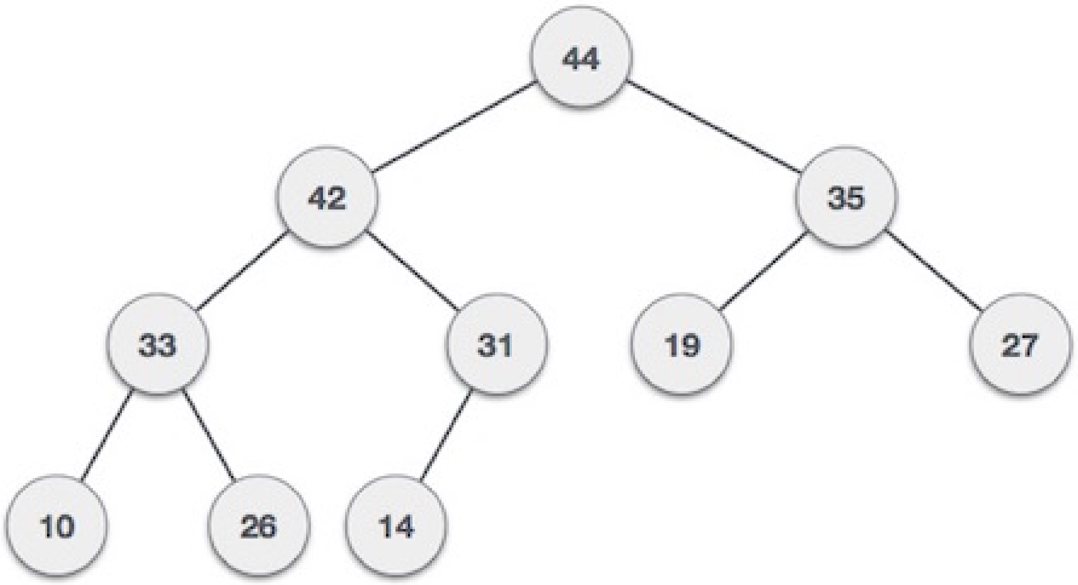
3.3.1 Data Trees

Definition 3.2 (Data Tree)**.** *A data tree is a type of data structure that represents a hierarchical connection between different bits of information. Data trees are very much like family trees, where each member is represented as a node(the circles in the image above) and each node is connected to each other by lines know as edges.*[Gee23]

The topmost node in a data tree is known as the root, and each node can have from 0 to an infinite amount of child nodes. Child nodes are nodes that branch from another node. Nodes that have the same parent node are called siblings, and siblings always lie at the same height of the tree. Each node in the tree can have only 1 parent, with the exception of the root that has no parents.

Data trees have a variety of important uses, and can help a lot when working with large data sets. A cool way to use a data tree would be to use one to solve a Rubik's cube. Your 'root' for this tree would be the starting position of your cube. Each child node would be a possible move you code make to the Rubik's cube from the starting position. Each and every move could be represented on this tree, and you could look down a series of branches to the leaves of the tree to figure out a way to solve a Rubik's cube.

3.3.2 Max Heap



Definition 3.3. *A max heap is a type of tree where the root of the tree is always the maximum number in the data set, and each parent node is always larger than the children nodes. These heaps can be extremely useful when you have to sort through a lot of data and you are trying to find the maximum value in a given data set.*

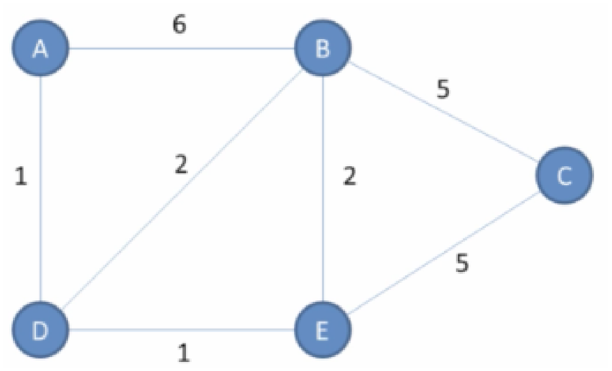
Say you were given the task to sort through a list of purchases your company made throughout the year, and you had to figure out what the most expensive purchase was. A max heap would work great here as using a computer to build it would mean all that information could be sorted through quickly and you would have a new set of arranged data that you could look through to see what the most expensive purchase was.

4 Graph Algorithms

4.1 Dijkstra's Algorithm

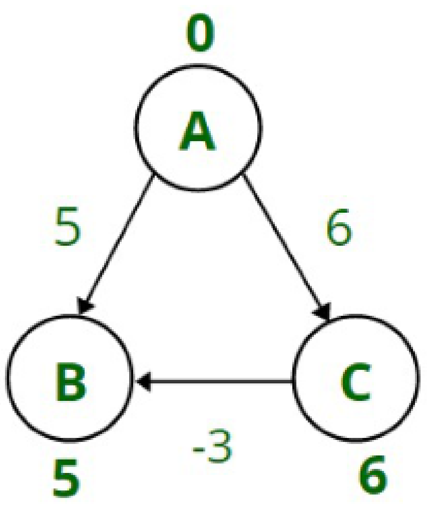
Dijkstra's algorithm is a graphical algorithm that finds the shortest path problem for a weighted, directed graph. In other words, it finds the shortest path from a starting vertex to all other vertices in the graph. The algorithm works by maintaining a set of vertices whose shortest distance from the starting vertex is already known. Initially, the starting vertex is added to this set with a distance of zero, and all other vertices are added with a distance of infinity. Then, the algorithm repeatedly selects the vertex with the smallest distance from the starting vertex that has not yet been added to the set, adds it to the set, and updates the distances of its neighboring vertices. (with non-negative weights). if there are negative edge weights, the algorithm may not find the shortest path or may not even terminate at all. If there is a negative loop, Dijkstra's algorithm will continue going through that loop infinitely in which case it will never terminate. Because of this, the algorithm won't consider negative weights which in some cases may prevent it from finding the shortest path. In such cases, other algorithms such as Bellman-Ford would be appropriate.

4.2 Examples



Say the computer started at point A and wanted to use Dijkstra's to find the shortest path to point C. It would start by traversing to point D and storing the information that the shortest path to point D is 1. It would then do a similar procedure from A to B and store that. Continue this process from D to E, then D to B. However, because the algorithm has found a new shortest path to get to B, it will replace the number it had previously stored (in this case that being 6) with the new number 3. Note that it won't substitute the stored values if the new "shortest" path isn't actually shorter; in this instance $3 < 6$ but that won't always be the case. Eventually, the system will conclude that the shortest path from A to C would be of length 7, that being from A to D to E to C.

In the second case (the figure in green), Dijkstra's algorithm wouldn't be able to find the shortest path from A to B. Even though to the human eye it's obvious that $6 - 3 < 5$, so the shortest path would be 3, the algorithm wouldn't detect this. It would see that the shortest path from A to C is 6, and because it assumes that there are no negative weights for edges it would make the conclusion that there exists no weight from C to B to make the total path length less than that of directly from A to B.



4.3 A formal proof of Dijkstra's algorithm

Let S be the set of visited nodes and d(v) be the current shortest distance from the starting node to node v. Initially, S contains only the starting node, and d(start) = 0. We want to prove that at any iteration of the algorithm, the node selected as the current node has the shortest distance among all unvisited nodes. Assume that there exists an unvisited node v such that d(v) < d(u) for all other unvisited nodes u. Let w be the node that connected v to the visited nodes in S, i.e., the node with the smallest weight edge (v, w). Then we have: d(w) <= d(v) + weight(v, w) (by the triangle inequality) d(w) <= d(v) + l(v, w) (since all weights are non-negative) Therefore, we have: d(w) < d(u) + l(u, w) (since d(v) < d(u) for all unvisited nodes u). Since d(w) is the smallest possible distance to node w, we must have w already visited before. But this contradicts the assumption that v is an unvisited node. Thus, the node with the smallest distance to the starting node among all unvisited nodes is always selected as the current node, and its distance is the shortest possible distance. Therefore, once the destination node is marked as visited, its distance is the shortest possible distance from the starting node.

4.4 Bellman Ford's Algorithm

Bellman-Ford's algorithm also solves the shortest path to each vertice, but it's able to detect negative cycles. Because of this, it doesn't assume all weights are non-negative and therefore has more versatile applications. Bellman-Ford's algorithm keeps track of the iterations and detects a negative cycle when there have been V iterations (where V is the number of vertices in the graph). This is because once there have been V iterations, a vertice must have been passed through twice because the starting node is given to have a path length of 0 and doesn't require an iteration. Because Dijkstra's algorithm assumes all weights are non-negative, it's more efficient than Bellman-Ford's algorithm for graphs with non-negative edge weights. Bellman-Ford's algorithm is necessary for graphs with negative edge weights or when detecting negative weight cycles is required.

Acknowledgements

We would like to thank the MIT Math Department for this opportunity, our mentor Carlos for his teachings, and Marisa Gaetz and Mary Stelow for coordinating and managing the program.

References

[BW13] E.A. BENDER and S.G. Williamson, *Mathematics for algorithm and systems analysis*, Dover Books on Mathematics, Dover Publications, Incorporated, 2013.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, 2nd ed., The MIT Press, 2001.

[Gee23] GeeksforGeeks, *Introduction to tree – data structure and algorithm tutorials*, 2023.