A Study on Computer Algorithms

Jessica Zhao

December 2020

1 Introduction

An algorithm, in its most basic form, is a set of step-by-step instructions to carry out a task. They are a fundamental idea in computer science, but their applications extend beyond into many other fields. They run in the background of almost every app, showing a user who enjoys sports more sports-related content.

Algorithms appear even outside the context of computer science. Take giving someone a set of directions for example. When someone asks you how to get from Point A to Point B, you give them a clear, concise set of directions: "Take this street down, turn left there, walk another block, turn right..." etc. Chances are you would give them the shortest route possible from Point A to Point B, despite there being many ways to get to the destination, both long and short. Giving someone (or a computer) a longer set of directions is not efficient. It would take them too much time. The goal of computer scientists designing algorithms is exactly this: have the computer get the task done in the most efficient way possible.

2 Runtime/Big O

To analyze an algorithm's efficiency, we must first define what exactly "efficient" means. The first thought that comes to mind is most likely time. Time is certainly factored into an algorithm's efficiency – the less time the computer takes, the more efficient it is. The number of instructions given can be another – the less instructions given, the better the algorithm. In this case, both time and the number of instructions on a computer are connected to how we define efficiency. Intuitively, it makes sense: The less instructions given, the less time it will take to complete a task. For a computer specifically, the amount of time it takes for a computer to execute a program is known as **runtime**.

There are a few ways to describe runtime: "Best case" runtime (what inputs the algorithm handles most efficiently), "worse case" (what inputs the algorithm handles least efficiently), and "average" runtime. For each of these cases, it's often easiest to have a bound on the runtime instead of figuring out the actual runtime. There are three types bounds: The lower bound, Ω, the upper bound, O, and tightly bound, Θ, where Ω, O, and Θ take as input functions of an input size n. Suppose the worst-case running time of an algorithm is $f(n)$. Then, $g(n)$ is the lower bound of the algorithm if for some constants C and N, $f(n) \geq C \cdot g(n)$ for $n > N$. We could also say that the worst case running time of this algorithm is $\Omega(g(n))$. Conversely, we can say that a function $h(n)$ is the upper bound for that same algorithm if for some constants K and M, $f(n) \leq K \cdot h(n)$ for $n > M$. The algorithm can then be said to have a worst case running time of $O(h(n))$. Tightly bound refers to when an algorithm is $\Omega(g(n))$ and also $O(g(n))$. We can also apply the same ideas to when $f(n)$ is the best case running time instead. Typically, any claim about running time is made using O, also known as Big-O Notation, as it's the easiest to prove and the one programmers care most about, given that algorithm design is all about reducing the worst case running time.

3 Algorithms

3.1 Gale-Shapley

The Gale-Shapley algorithm solves the Stable Marriage Problem. The problem is as follows: Given a set $M = \{m_1, m_2, ...m_n\}$ of n men, and a set $W = \{w_1, w_2, ...w_n\}$ of n women, is there a way to match each man with a woman such that each person only appears in one pair? This set of pairs is known as a perfect matching set. That is, every person is married to somebody and there is no polygamy.

It sounds like a simple problem, but suppose each person has a list of preferences for who they'd want to marry. Each man has a ranking of all the women, and each woman has a ranking of all the men, assuming no ties in preferences. Let S be a set of pairs consisting of one man and one woman each. Consider two pairs in the set, (m, w) and (m', w'). This set S is **stable** if m and w prefer each other and m' and w' prefer each other. The set is **unstable** if, for example, m prefers w' to w and w' prefers m to m'. In the second case, m and w' would abandon their partners to end up with each other, leaving w and m' on their own. Thus, the goal of the Stable Marriage is to create a set of pairs with no instabilities.

The idea of the algorithm is outlined as follows:

1. First, everyone is initially unmarried. An unmarried man will propose to the first woman in his list of rankings. That woman, however, cannot immediately reject him, for she is unsure if another man of higher ranking in her list will propose to her. We are also unsure if this first pair will be a stable matching. So the two enter an intermediate state of engagement.
2. Now, some men and women are free, i.e. not engaged, while others are engaged. We choose any free man m to propose to the woman of highest rank in his list to whom he has not yet proposed. If the woman is not already engaged, she and that man will become engaged. If she is, let us call that man m'. If m ranks higher than m' in the woman's list, then she will leave m' to be free and become engaged to m. If m ranks lower than m', she will stay engaged to her current partner. In either case, m crosses w off his list as he will not propose to the same woman twice.
3. The process will end when no one is free.

The result from this algorithm is that it always produces perfect, stable pairs. The running time for this algorithm is $O(n^2)$. For proofs and further details, see [2].

3.2 DFS/BFS

Another useful algorithm is one that searches for a solution. Consider Figure 1 below.

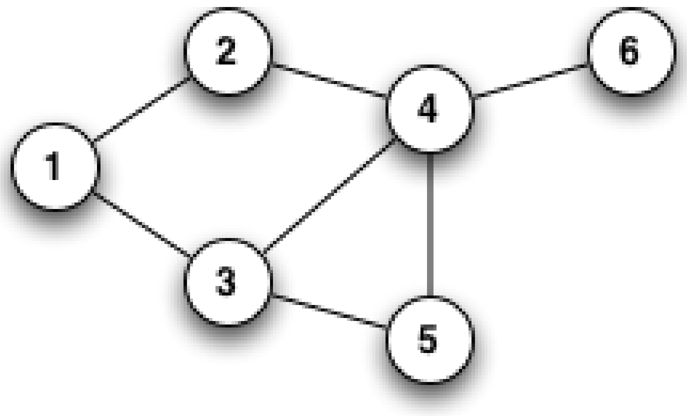

Figure 1: A graph with six nodes and seven edges, image courtesy of[1].

The numbered vertices, or nodes, combined with the edges between them form what we call a **graph**. There are a number of questions we can ask about this graph: Is there a connection between one node and another? Does the graph have a cycle? What's the shortest route between one node and another (assuming a route exists)? Many problems that involve graphs can be answered using one of two search algorithms: Breadth-first-search (BFS) and depth-first-search (DFS).

Breadth-first-search works by picking a starting node, then visiting the nodes that are one edge away from the starting node, then visiting all of the nodes that are one edge away from those, and so on.This process repeats until the objective is completed. Below is a rough outline of the steps for a BFS:

1. Pick a starting node.
2. Find all the nodes adjacent to the starting node and add them to a queue.
3. The next node chosen is the first in the queue. For each node that's visited, add its adjacent nodes that have not been visited to the queue, and repeat.

In our example from Figure 1, suppose 1 is the starting node. We add 2 and 3 to the queue, which now looks like [2, 3]. We then pick the first one in the queue, which is 2, and add the nodes adjacent to 2. Since 1 has already been visited, the only adjacent node is 4. That gets added to the queue, and 2 is removed, making the queue [3, 4]. The next node visited is 3, and we add the nodes adjacent to 3. Those nodes are 4 and 5, but because 4 is already in the queue, 5 is the only one added, making the queue [4, 5]. This process repeats so on and so forth.

Depth-first-search works in a similar way, but the order the nodes are visited in is different. There is a starting node, except this time, it goes all the way through one path of edges before it backtracks. The steps for a DFS algorithm are outlined as follows:

1. Pick a starting node. Mark it as visited.
2. Add all of its adjacent nodes to a stack.
3. The next node to visit is the next unvisited node that is at the "top", or the last node added, of the stack. Remove that node from the stack, mark it as visited, then add all of its adjacent nodes to the stack
4. Repeat the process until the search is complete.

In our example with 1 as the starting node, we add 2 and 3 to the stack. So now the stack becomes $[2, 3]$. Assume the "top" of the stack is the right-most element in the list. Then by our algorithm, 3 is the next node to visit. We add the nodes that are adjacent to 3 that have not yet been visited. Since 1 has already been visited, we add 4 and 5 to the stack, making the stack $[2, 4, 5]$. Now, 5 is the next node to visit. We have already visited 3, and 4 is already in the stack, so we're done with node 5 blue, and the stack is $[2, 4]$. We go to 4 next. Its adjacent nodes are 2, 3, 4, and 6. We've already visited 3 and 5, and 2 is already in the stack. The node 6 is the only unvisited node that is not yet in the stack, so we add it. The stack is now $[2, 6]$. We go to 6, but 4 has already been visited. So we go to 2, and find that all of its adjacent nodes have been visited. Now that the stack is empty, the loop terminates, and our traversal through the graph is complete.

Both algorithms have a running time of $O(N + E)$, where N is the number of nodes and E is the number of edges, and both can be used to answer whether or not there is a path between two nodes. However, when this problem becomes finding the shortest path between nodes, BFS works better than DFS. The problem with using DFS in this situation is that the path will go all the way through one path before making its way back up near the starting node, when the answer may have been only two nodes away. In our example graph, say we want to find the shortest path from 2 to 5. If we use DFS and pick 1 as the next node, our path would end up being 4 nodes long ([2, 1, 3, 5]), or possibly even longer if we choose to pick 4 after visiting 3. However, the shortest path would actually be [2, 3, 5], which is 3 nodes long.

A Code

Code for the Gale-Shapley algorithm and an example of a DFS traversal can be found at `https://github.com/zhaoj29/TwoplesF2020`.

References

[1] Techie Delight. *[Undirected graph with 6 nodes]*. 2016. URL: `https://www.techiedelight.com/terminology-and-representations-of-graphs/`.

[2] Jon Kleinberg and Eva Tardos. *Algorithm Design*. USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN: 0321295358.