Combinatorics in Algorithms

Joshua Randazzo
Mathematics Department
Nebraska Wesleyan University

August 21, 2014

Introduction

First it seems pertinent to define the idea of algorithms as they are to be discussed. An algorithm is a finite number of well defined procedures, that takes an appropriate specified type input, or initial state, through a series of states via an enumerable amount of procedures, until it reaches a final state, or output, at which time the algorithm halts. While thought about this definition are borrowed from Bóna, and exact defintion of what an algorithm is is notoriously hard to pin down [1]. With all different types of algorithms its important to know that not every possible input is appropriate for every algorithm. For example, their are algorithms designed to find maximum or minimum values of functions over specified ranges, and input of *PurpleDuck* would not qualify as an appropriate input. When calling a procedure well defined, we more clearly mean that each procedure is completely unambiguous, and exactly prescribes a specific action. Furthermore, a finite number of those well-defined procedures means that, regardless of input, the steps taken will not result in a infinite loop of states, and the algorithm will halt. Which raises the question; How can you tell if a set of steps will halt and produce an output from any appropriate input, or will result in a infinite loop. This paper will look at such, as well as a few different types of algorithms, and quite importantly their connection to combinatorics, as well as some of their real world application.

Halting Problem

Certainly some algorithms are easy to see that with acceptable input and will halt:

For positive integer input
$A_1 =$ input
(1) $n = 1$,
(2) Assign $A_{n+1} = A_n + 3$,
(3) Change $n = n + 1$,
(4) If $n = 5$ output A_n and goto '6',

(5) else goto '(2)',

(6) Terminate.

Clearly, A_n will increment from 1 to 5, adding 3 to the input in each step, and will achieve an output. Though not all procedure sets are as easy to determine as algorithms as this. Regrettably there isn't a definitive way to determine whether a procedure set is an algorithm or not.

Theorem 1. *There does not exist an algorithm that can determine whether a set of procedures will halt when given an input.*

Proof:

Let us assume there exists an algorithm that can determine if a set of procedure with halt, namely Halt(T,t) so that,

$$\textit{Halt(T,t)} = \begin{cases} \textit{'Yes' if T halts when given t as input,} \\ \textit{'No' if T goes into an infinite loop when given t as input.} \end{cases}$$

and also a program diagonal such that,

$$\textit{Diagonal(s)} = \begin{cases} \textit{Return 'Yes' and halt if Halt(s,s) is 'No',} \\ \textit{Goes into an infinite loop if Halt(s,s) is 'yes'.} \end{cases}$$

Assume Diagonal(Diagonal) halts. By definition of Diagonal, that means Halt(Diagonal, Diagonal) is 'No', but by definition of Halt, it does not halt, creating a contradiction. Assume Diagonal(Diagonal) goes into an infite loop. By definition of Problem, that means Halt(Diagonal, Diagonal) is 'Yes', but by definition of Halt, it does halt, again creating a contradiction. Therefore, our original premise that there exists an algorithm that can determine whether other sets of procedure halt is false. Therefore there is no such algorithm that can determine whether a set of procedures is in fact an algorithm. [1, pg 496-497]

Growth Rate Comparison

A major application of combinatorics to algorithms, is counting the number of procedures, or steps, required to complete the algorithm. While other things do come into play, such as space complexity, being able to complete a task in less steps generally speaking makes it a better algorithm then one that requires more steps for the same outcome. For sake of comparison the algorithms efficiencies are usually functionally approximated, meaning that their run times can generally be approximated by functionally approximated, and have different classifications based on best-case, average, and worst-case scenarios. There are 3 definitions that are widely used in approximation. The first of which is O (read 'Big O') which compares 2 functions, lets say f and g. [1, pg 494-495]

Definition 1.1. *Let f be a function from $\mathbb{Z}^+ \to \mathbb{R}$, and let g also be a function from $\mathbb{Z}^+ \to \mathbb{R}$. We can say $f(n) = O(g(n))$(read 'f is big O of g') if $\exists c \in \mathbb{R}^+$ such that*

$$f(n) \leq cg(n)$$

$\forall n \in \mathbb{Z}^+$. This means that f is no more than a constant larger than g for all n.

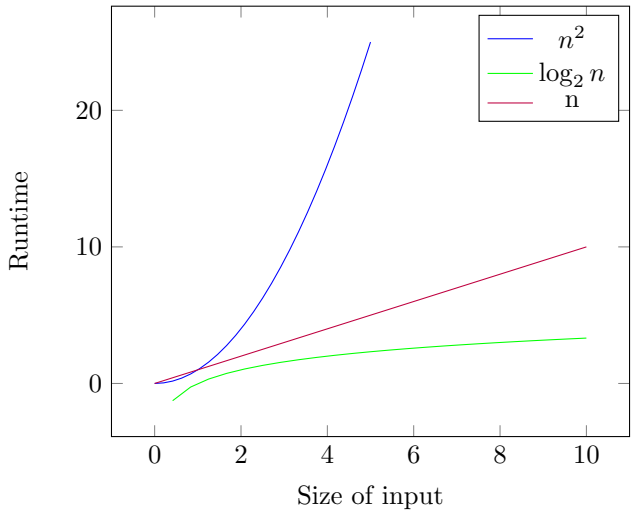
Definition 1.2. *Similarly with functions f and g both defined from $\mathbb{Z}^+ \to \mathbb{R}$ we can say $f(n) = \Omega(g(n))$(read 'f is big Omega of g') if $\exists c \in \mathbb{R}^+$ such that*

$$f(n) \geq cg(n)$$

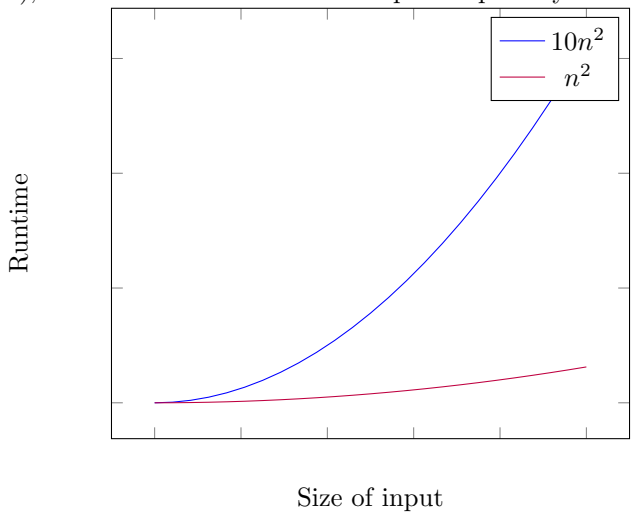
$\forall n \in \mathbb{Z}^+$.

Definition 1.3. *Finally if and only if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ are true, then we can also say $f(n) = \Theta(g(n))$(read 'f is Theta of g').*

Having such approximations are valuable in that being able to say that an algorithms run time on average is linear, or at worst quadratic, or whatever the case may be, give good comparison to others that perform the same task, and maybe in completely different efficiency classes, such as comparing something with linear, quadratic, or logarithmic runtimes.



Also its important to keep in mind, just because algorithms can be classified by the same $O(f(n))$, that they can run significantly faster or slower. Certainly algorithms that run at true n^2 pace and one at $10 \cdot n^2$ pace both fall under $O(n^2)$, but run as a rather different speed especially for small input.



Sort Algorithms

One of the most common types of algorithms used today are sorts. Ordering a set of object, by a specific classification makes it exceptionally easier to find things, as well as to compare them. Probably the most useful type sort real numbers, which is how a lot of things are classified, or how a certain attribute of a group of objects is described. Assuming a list is n items long, In theory best case scenario for a lower bound of for the number of comparisons for sorting them would be $\frac{n}{2}$ because that is the minimum number of comparison required so that we have some knowledge about each item in the list. Regrettably we can't sort lists in $\frac{n}{2}$ time, which begs the question, How fast is a good sort? And perhaps will the same sort always take the same amount of operations? Currently the best case scenario for a sort is $n-1$, which can be obtained in multiple manners, most easily by looking at an already ordered list and comparing adjacent elements $e_1 < e_2, e_2 < e_3, ...e_{n-1} < e_n$. Certainly some sorts are better than others, as well as some being better in different situations, whether we know the size of the list to be sorted, or if the listed is nearly sorted to begin with.

Cocktail Sort

First we'll look at the cocktail sort, also known as a bidirectional bubble sort. Assuming that the sort is to be sorted from smallest to largest, and the input is a list of real numbers, and currently for simplicity all items are distinct. First label the numbers $a_1, a_2, ..., a_{n-1}, a_n$. First compare $a_1 < a_2$ if that's true leave them be if false swap them, then $a_2 < a_3$ again if that true leave them and false swap them, continue this through $a_{n-1} < a_n$. Now we know that the largest item is in the correct position so on subsequent passes we will go one less position each time, that is next pass up will stop at $a_{n-2} < a_{n-1}$. Next we traverse the list in the opposite order, comparing $a_{n-2} < a_{n-1}$, if that's true then leave them and if not swap them, just as on the way up, continuing in the same way until $a_1 < a_2$. Now we know that the smallest item is in place, and as it was with the largest side we won't need to go all the way to the bottom again.

If through any pass up or down, no items are swapped then the algorithm halts, as the items are completely sorted. So if by chance the list was already sorted, this sort would perform $n - 1$ comparisons, and such would be a wondrous time algorithm, regrettably that's just its best case scenario. Regrettably both its worst and average case scenarios are $O(n^2)$, though when this sort comes acrossed list that is partially sorted its run time can get close to $O(n)$. If all of the items starting places are no more than k spaces from there final resting spots, the algorithm runs $O(k \cdot n)$. For small lists, or lists that are known to be nearly sorted it doesn't do a bad job, but in the world of sorting algorithms it definitely isn't king. [3]

Quicksort

Now we will discuss a significantly more efficient Sort. Quicksort was designed by Tony Haare in 1960, and has been since approved upon. The main idea of Quicksort is that and item in the list is chosen as a pivot point, and all the items in the list larger than the pivot are put to one side and all the items smaller are put on the other, thus leaving the pivot in its proper final position, and the above and below the pivot are sorted in the same fashion recursively calling the algorithm on the now shorter sub-lists. Next is how the choose the pivot points, at differing times, the first, or last or middle element has been used, but Robert Sedgewick(A respected computer scientist, and Professor at Princeton) offered an optimization that calls for the use of the median element of the left, right, and middle element of the array. This adds efficiency in that, if say the first k elements are already properly ordered and pivots are being selected from the first element, the first k iterations will do nothing. Like most quality sorts its best case scenario is $O(n)$, though very rare and unlikely its worst case run times are $O(n^2)$. Though on average it runs $O(n \log n)$ and typically runs faster the other $O(n \log n)$ algorithms. [2]

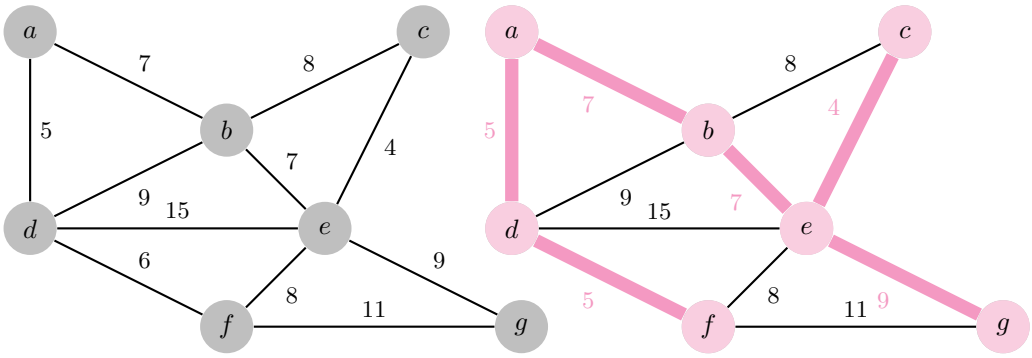
Algorithms on Graphs

Next we'll look at algorithms and how they can be used on graphs. One of the most common, which many of us use routinely are shortest path algorithms, that is how to get from point A to point$B$ the fastest. Every time someone opens Google maps, or uses a Garmin, or there cell phone for directions, some variation of a shortest path algorithm is put to work. Algorithms that construct minimum spanning trees, are also very important and readily applicable to modern life. Anytime electrical lines, or railway lines are constructed such algorithms are consulted in an effort of efficiency, not only for time but materials sake as well.

Kruskal's Algorithm

Kruskal's Algorithm is a a greedy type algorithm, used to find a minimum spanning trees. When greedy algorithms are effective they generally are the best option, and are more efficient. Though in most cases greedy algorithms fail, due to the fact that they look for the best option in the current local situation, and as a result miss whats best for the global goal of the algorithm. Kruskal's Algorithm takes input of a graph (edge's and vertices), with weighted edge's. first it selects the edge with the lowest value. Next and until all vertices are accounted for it selects the next lowest edge that doesn't create a cycle. In small graphs, and when performing this algorithm by hand, this seems rather straight forward, but when implementing this algorithm via computer, its a little more complicated. The primary problem is how do you determine that adding an edge will create a cycle or not. Certainly this has been approached in different manners, but the way that seems most effective is that when the first edge is selected, the two attached vertices, are put into a list. When subsequent edges are up for selection, if neither of the vertices of the edge are in an existing list, and new list is created, with those two vertices in the list, and the edge is added. If one vertex is in a list, the other vertex is added to the list, and the edge is added. If one of the vertices is in one list and the other is in another list, the lists are merged, and the edge is added. Finally if both vertices are in the same list, this means that a cycle would be created and the edge is rejected. Also, its interesting to note that if the input graph isn't connected the output graph

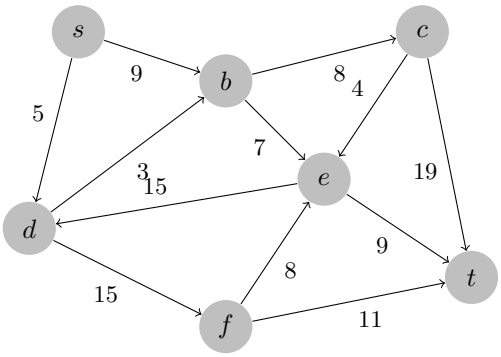
will be a set minimum spanning trees (Also known as a minimum spanning forest). When discussing the complexity of graphs its common to discuss such relative to the number of edges(E) and vertices(V). Kruskal's algorithm runs in O(Elog$_2$E). [1, pg 496,497] So here's an example of an input graph, and the output based on Kruskal's algorithm. All the edges selected follow the simplest greedy premise of taking the lowest weight edge, up until the point when $\bar{f}e$ should be taken, but a cycle would be created, and likewise with $\bar{b}c$ and $\bar{b}d$. And the the graph is finished with $\bar{e}g$.



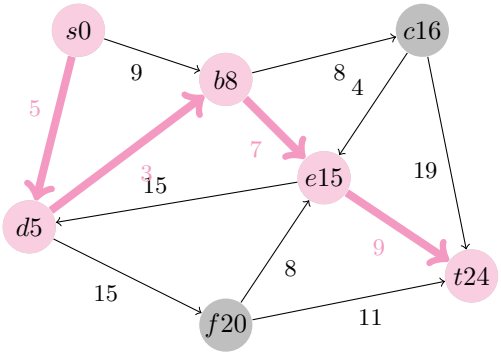
Djikstra's Algorithm

Before we can actually describe the manner in which Djikstra's algorithm solves its problems, which is the shortest distance from one vertex to another in directed graph, G, we must establish some definitions. First the we will denote d(e_i) as the length of e_i and $\delta(v_i)$ as the distance from the starting vertex, denoted s, to v_i. Additionally, we will call our final or terminal vertex t. Initially we will set $\delta(s) = 0$ and $\delta(t) = \infty$. First we will divide the set of vertices of G, $V(G)$, into 2 sets, S vertices that already have a path to s and T those that don't. Initially $S = \{s\}$ and $T = V(G) - s$. Next forall vertices, v, that are adjacent to s we set $\delta(v) = d(s, v)$, that is to say we set the starting shortest distance of these adjacent vertices to the length of the edge that connects them to the start. Now we will describe the basic steps of the algorithm. Pick a vertex v such that $v \in T$ and that $\delta(v)$ is minimal. Put v into set S and now continue to the adjacent vertices of v, that is for edge vr if $r \in T$ and $\delta(v)+d(v, r) \geq \delta(r)$,

then do nothing. Otherwise reassign $\delta(r) = \delta(v) + d(v, r)$, which indicates that we have found a shorter path to r. Continue this step, until all vertices are apart of set S, or when there are no more edges for S to T. [1, pg499-503] Since,



s is the starting vertex we will first assign δ values to its adjacent vertices d and b, $\delta(b) = 9$ and $\delta(d) = 5$ and add both to S, $S = \{s, b, d\}$. Next since $\delta(d)$ is minimal we looks at its adjacent edges f and b, $\delta(f) = 20$, and since $\delta(d) = 5$ and $d(d, b) = 3$ and $3 + 5 \leq 9$ we reassign $\delta(b) = 8$. Continuing this process we get the output graph, with the shortest path highlighted.



Dijkstra's algorithm runs in $O(n^2)$.Though there are refinements and optimizations for the algorithm, which can improve its runtime.

Closing Thoughts

As with most pieces of academia, there are real world practical, items that affect the reality of how things work. With Some of the sort algorithms, they are

remarkably effective for starting a sort, and as they recurse down into smaller lists other algorithms can finish the job in shorter time. Also its imperative for the users of these algorithms to know what types of lists, and the sizes of lists being sorted, as well as the storage space they have available to do so. Clearly some sorts fair better with larger or smaller, more sorted or more jumbled lists, as well some use significantly more storage. Furthermore, with some of the recursive sorts, the rise of parallel processors has changed which algorithms are better for which tasks, but that's a discussion for an other paper. Likewise, graph algorithms are affected by other real world phenomena as well. For example when Google maps decides on a shortest path, there are different setting for walking, driving, or biking. As well the speed allowed on a current roadway is taken into account. When building railway networks via minimum spanning tree, some times the physical distance from a to b can take a back seat to the fact say a mountain is in the way, or perhaps not disturbing a natural preserve. Our ability to automate problems through the use of algorithms is a wonderful advancement, but as long as technology and the world change the algorithms used and optimizations added will continue to change.

Bibliography

[1] Bóna, Miklós, A Walk Through Combinatorics, Third Edition, World Scientific Publishing CO., Hackensack, NJ, 2011

[2] Martin, David R., http://www.sorting-algorithms.com/quick-sort, 2007

[3] "Cocktail Sort." Wikipedia, The Free Encyclopedia. Wikimedia Foundation, Inc. 11 July 2014. Web. 21 Aug. 2014.