Using Dijkstra's Shortest Path Algorithm as a Basis for Solving the Maximum Sum Path Problem

Kelvin Schutz
scke1201@stcloudstate.edu

Abstract

The Maximum Sum Path is a problem that asks to find a path down a tree by starting at the top of the tree and moving to an adjacent number on the row below it. Brute Force approaches have difficulty with these trees, because they introduce 2^n-1 possible paths to calculate, where n is the number of levels to the tree. A recursive approach may also be taken, as well as utilizing memoization to avoid any unnecessary duplicate computations. Unfortunately, recursion has its own issues in relying on an adequately deep call stack. Dijkstra's Shortest Path algorithm can be modified to solve the Maximum Sum Path problem by using a greedy approach. Not only does outperform recursive solutions, it is also free from call stack depth issues.

Introduction

The Maximum Sum Path is a problem posed on projecteuler.net that states the following:

> By starting at the top of the triangle below and moving to an adjacent number on the row below, the maximum total from top to bottom is 23.
>
> **3**
> **7** 4
> 2 **4** 6
> 8 5 **9** 3
>
> That is, 3 + 7 + 4 + 9 = 23.
>
> Find the maximum total from top to bottom in triangle.txt, a 15K text file containing a triangle with one-hundred rows.

The problem also informs the reader that a Brute Force approach would take many times the length of the universe's existence to compute. We could certainly look at the problem in hopes of breaking it into smaller problems of the same form. Ultimately, we could use recursion to solve this for us, but we would also need to employ memoization, as the recursion stack would become too deep and overflow.

I propose a different solution. We reframe this problem in the hopes that other algorithms have a more comprehendible solution for us. The nature of recursion is that computers work wonderfully with them, with the use of a stack structure. Unfortunately, humans don't think recursively so well. Thankfully, algorithms come in step-by-step form, so that even a novice of computer science/mathematics can follow along.

In this paper, we look to Dijkstra's Shortest Path algorithm to overcome issues with the use of a call stack and come to understand the algorithm in ways that a novice may understand it for the Maximum Sum Path problem. After laying the foundation, we will compare the recursive + memoization solution[2], which we will call the Top-Down approach, against my proposed solution inspired by Dijkstra's Shortest Path algorithm, which we will call the Bottom-Up approach.

Methods

Dijkstra's Shortest Path algorithm attempts to find the shortest route between any two connected nodes[1]. Just as easily, we could also attempt to find the longest route between any two nodes with a change in how we do comparisons in the algorithm. In the Maximum Sum Path problem, we are seeking to find the longest route between a set of nodes located at the bottom of a tree and the root node at the top. The tree is represented in figure 1.

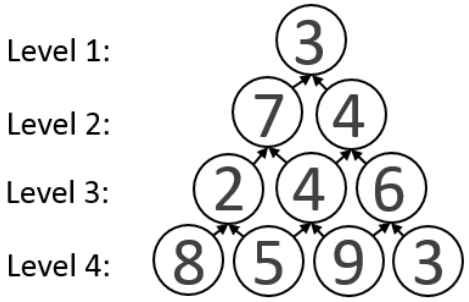

Figure 1: The number of levels are indicated to the left, starting at level 1. To find the number of nodes in a tree (n) with a number of levels (k): n = (k * (k + 1)) / 2

In a way, we could attempt to use Dijkstra's Shortest Path algorithm for each bottom node. If we were to choose the largest of the bottom nodes, and choose the largest of its two parents, we may find that we arrive at the wrong solution when using that as a sequence for checking every level on up. If we can say this about the lowest level, it must be true that the problem also arises at all other levels, since they are equally the same in all regards. Thus, we must check every node at every level in order to arrive at the optimal solution. We can go about this in several ways, yet still use the idea of a greedy algorithm to find the locally optimal solution at every step [4]. We cannot say, however, that any locally optimal solution will be part of the maximum sum path. That

is, we must carry over all candidates through each level; not doing so would very likely lead to a non-optimal solution. In this way, we can choose to see each level as a scope for finding the locally optimal solution. Once all of the candidates for one level have been found, we move onto the next level. We can do this for every level in the tree, until we arrive at the root (level 1). The root must contain our solution by this process. Figure 2 shows what would be happening during each iteration of a level on the tree.

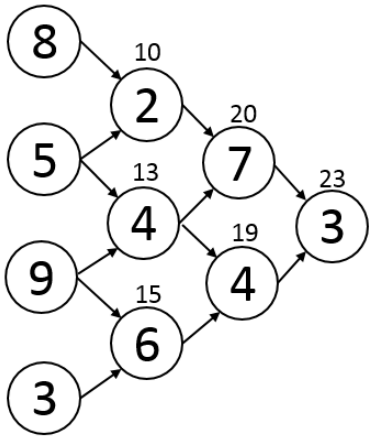

Figure 2: Starting on the far left, a node's value propagates to its parent only if it is greater than its sibling sharing the same parent. The propagation is noted by the value above a node. In the case of the propagated 10 value, 8 is greater than 5, so it is added to 2 to give the propagated value 10.

I chose Python to program this algorithm into. As is common with heaps and trees, I represented the tree in an array (list in Python). Figure 3 shows the array representation of the tree.

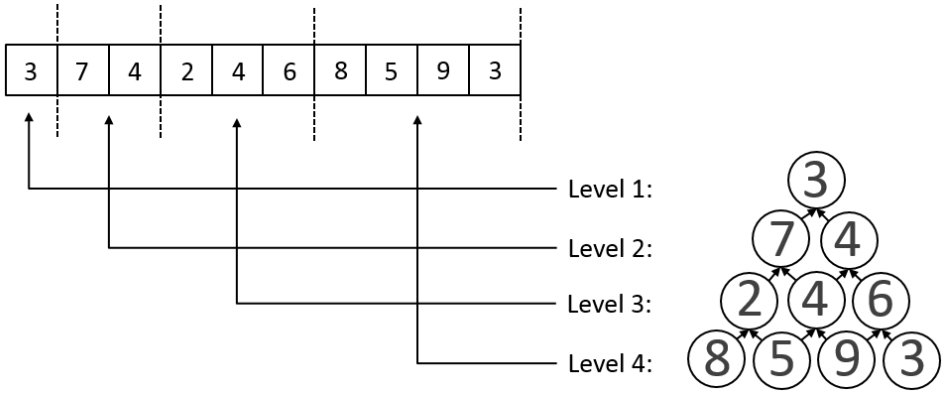

Figure 3: Each level is represented in the array, starting with the root at the head of the array.

In order to access specific nodes of the tree, specifically the children of a given node and all of the elements of a given level, we must find appropriate formulas. In the case of finding the child nodes, I implemented `getLeftChild` and `getRightChild` functions to handle retrieving those indices from the array.

```
defun getLeftChild(index, level):
      return index+level

defun getRightChild(index, level):
      return index+level+1
```

Figure 4: Find the index of left and right children

Since we will know the current index and level as we iterate through each level, we can simply pass the index and level as parameters to the function.

We will also need to know the range on which we will be iterating through for each level. This is very simple, since the number of elements in a level is equal to the level number itself. I created an appropriate function for getting the range of elements to iterate on for a given level.

```
defun getFirstElement(level):
      return (level*(level-1))/2

defun getLastElement(level):
      return ((level*(level-1))/2)+level-1
```

Figure 5: Get the first and last indices of a given level

And with this we can create the framework for our algorithm in a set of nested for loops. The algorithm will begin by skipping the very bottom level, since it contains no children. For each element in a level, we compare the children and add the larger of the two to its parent's value. In effect, the local best solutions are stored in the parents and then the algorithm jumps up to the next level, doing the same thing with new parents and children containing the previous locally optimal solutions. Figure 6 shows the algorithm taking place on its initial phase.

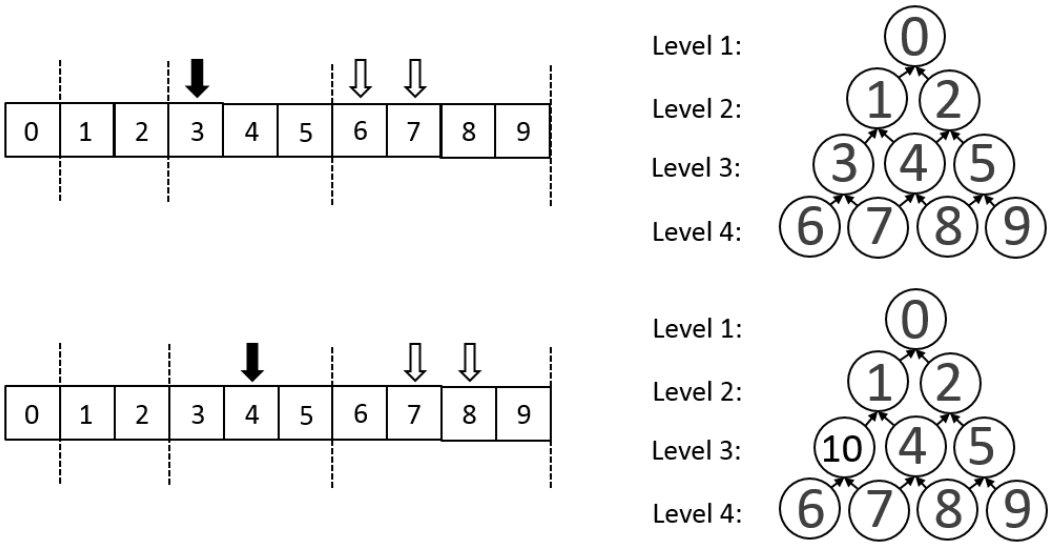

Figure 6: The algorithm begins by comparing the far left node of the second to last level in the tree to its children. In this case, the first node's index is 3, and per our access functions, its left and right children are located in index 6 and 7, respectively. The values of all nodes in this tree are initialized as their indices. Since the value of 7 is greater than 6, it is added to 3 to make 10. 10 is the optimal sum for that parent node.

This process can be realized in code now, since all the pieces are known.

```
def findMaxPath(tree):
  #Iterate through each level except the last
  for x in range(numOfLevels-1, 0, -1):
    #For every element in a level, compare its two children
    for y in range(getFirstElement(x), getLastElement(x)+1):
      #Add the larger of the two to the current element
      if(tree[getLeftChild(y,x)] > tree[getRightChild(y,x)]):
        tree[y] += tree[getLeftChild(y,x)]
      else:
        tree[y] += tree[getRightChild(y,x)]
  return tree[0]
```

Figure 7: Bottom-Up approach

The function will return the root node of the tree, which contains the optimal solution for the problem. With this algorithm implemented, we can test various data sets against an implementation of the recursive solution mentioned earlier [2]. I've provided the main algorithm code below for reference. It employs a list to record previously calculated paths as a memoization technique. As we will see, recursion runs into some issues as the data sets grow larger. In order to delay the inevitable stack overflow, I increased the programs recursion limit with `sys.setrecursionlimit(25000)`, so that it was Window's stack limits causing the crash instead of the Python interpreter.

```
def test(line, index):
    global cache
    if line > len(lines)-1:
        return 0
    key = str(line) + "|" + str(index)
    if not key in cache:
        cache[key] = lines[line][index]
                + max(test(line+1, index), test(line+1, index+1))
    return cache[key]
```

Figure 8: Top-Down approach

Results

Table 1 below contains the results for both algorithms running each test case. The results for the Brute Force approach aren't included, since the approach fails to complete even the lowest test case (100 levels) in any conceivable amount of time. All tests were run in Python v2.7.6 on the same machine using an Intel i7 4700MQ at 2.4GHz with 24GB of RAM under Windows 7.

	Number of Levels (n)						
Algorithm	**100**	**500**	**1000**	**2500**	**5000**	**10,000**	**20,000**
Bottom-Up	.005	.139	.548	3.582	14.657	56.194	228.251
Top-Down	.008	.219	.952	7.162	39.487	Overflow	Overflow

Table 1: The far left column displays the algorithm used. The top of each column shows the number of levels being tested. The results of each test are in seconds. Cases of stack overflow are noted for the Top-Down algorithm.

In all test cases, the Bottom-Up algorithm outperforms the Top-Down algorithm. As the number of levels increases in a given test case, the Bottom-Up algorithm continues to expand its difference against the Top-Down algorithm. As the number of levels (n) approaches 10,000 and above, the Top-Down algorithm expends Windows' stack limits resulting in overflow errors. Adding test cases beyond 20,000 is difficult, because of the time it takes to generate files of such a size. In fact, it took longer to generate the test files than the Bottom-Up algorithm took to solve them! A file with 20,000 levels contains over 400 million items and takes up 585MB of hard drive space.

Conclusion

The Maximum Sum Path problem requires more than a Brute Force approach to solve. As the number of levels approaches 100 in a tree, the time to compute the solution becomes astronomically long. Two common approaches involve solving the problem from the bottom-up and from the top-down. The Top-Down approach is commonly

solved using recursion plus memoization. Unfortunately, this technique introduces itself to stack overflow issues as the problem size increases. This, in itself, requires a different approach to solve problems of larger magnitudes. There are also efficiency and readability gains to be had by choosing an alternative, Bottom-Up approach.

The Bottom-Up approach resembles Dijkstra's Shortest Path algorithm in that it applies a greedy algorithm in search of the global maximum (minimum in the case of Dijkstra's algorithm). By taking this algorithm a step further, we can expand on its use from a single source node to a multitude of source nodes. Rather than immediately jumping to a recursive solution, Dijkstra's algorithm gives a sufficient basis for exploring its implementation in the Maximum Sum Path problem.

The results clearly show that recursive algorithms have a quickly-approached threshold, even when incorporating memoization techniques. Whether this is an issue, largely depends on the domain of the problem being solved. Regardless of the domain size, the consistently outperforming algorithm should be chosen for a given problem set. The Maximum Sum Path problem gives an appropriate context in which two different approaches can solve a problem, but with quite different consequences in how they go about finding the solution.

References

[1] Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". *Numerische Mathematik* 1: 269-271

[2] lassevk (2005, Sept 23) *Another top-bottom solution in Python.* Retrieved from https://projecteuler.net/thread=67

[3] Unknown Author (2004, April 9) *Maximum Sum Path II*. from https://projecteuler.net/problem=67

[4] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani (July 18, 2006) "Greedy Algorithms". *Algorithms* 1: 133-154. Retrieved from http://beust.com/algorithms.pdf