Theory of Algorithms: Homework 5
Project Code

Brae Troutman

April 7, 2023

1. Problem 1: $O(nW)$ 0/1 Knapsack Traceback

   The implementation of my answer for this problem can be found in the `q1` directory of the github repository linked at the top of the document in the file `src/knapsack01.jl` and the program can be run for each problem size using `make {small,medium,large}` to run a specific instance or simply `make` to run on all instances. Solutions to each instance are output in `out/{small,medium,large}.txt`.

   I realize that I've been really inconsistent with my programming language choices these past few homeworks, and I'm sorry for any inconvenience that causes with having to setup new compilers or anything along those lines

   I realized that my python implementations of the 0/1 knapsack in my last homework were woefully inefficient because of all the pointer chasing overhead and lack of locality that comes with using python's heterogeneous lists– I originally tried using numpy, but I'm still pretty unfamiliar with it and was having trouble vectorizing my functions to work on efficient NP arrays, so I switched to Julia which uses efficient column-wise storage by default for any concrete types of constant size.

2. Problem 2: $O(W)$ Optimal Split

   To solve this problem, I drew my code from Hirschberg's Algorithm for solving the Edit Distance problem with linear memory and still return the actual alignments of two strings. His divide and conquer strategy maps really closely to the example pseudocode given in the homework prompt, except some the subproblem dependencies of edit distance are a little different from 0/1 Knapsack.

   To find the optimal split capacity k of a given instance of the knapsack, we divide the list of values in half and solve the problem once with the full capacity of the knapsack on the first half of the items and return the list of optimal values for each capacity from 0 to W, and we also calculate the same with the last half of the items. Let's call these two solutions x_1 and x_2 respectively. By pairwise adding x_1 and the reverse of x_2, we are creating a vector where each entry represents the optimal value achieved when allowing the first half of the items to take up i of the capacity, and the second half to take up $W - i$ of the capacity. That means that the optimal split capacity will be the index of whatever the maximum value in this array is!

   My implementation essentially solves the problem twice, once for each half of the items, meaning that the total time complexity is $2nW = O(nW)$, and the space complexity also just adds a constant factor of two to the previous memory efficient solution.

   My answer to this problem is implemented in `q2/src/knapsack01.jl` in the functions `OptimalSubsolution` and `Knapsack`

3. Problem 3: Knapsack Divide and Conquer Traceback

   This answer is implemented along with the previous question in the `q2` directory.

   At the base case of this algorithm, we have a list of 1 item. If that item has a weight less than the capacity, we return a singleton list containing that item. Otherwise we return an empty list.

   In our recursive case, we find some value k representing the capacity needed to hold items in the optimal choice of items from the first half of the list of items. Given this optimal split k, solving the knapsack recursively with a capacity of k on the first half of the items and with a capacity of $W - k$ on the second half yields two lists that when concatenated represent the optimal choice of items from the full list.

   This is because the optimal split k allows us to divide our problem into two subproblems– we know that each subproblem solution is optimal because we know that our `OptimalSubsolution` function is optimal, and we know that the total weight of the items chosen will not exceed the total capacity because the individual subproblems will not exceed their share of the capacity k and $W - k$.

   As for the running time of this algorithm, we have the worst case recurrence:

$$T(nW) = 2T(n/2W) + O(W)$$

   Because we recursively call for two equal subproblems and use $O(W)$ work at each level to find the argmax of our two optimal subsolutions. For our two recursive calls, the total capacity may not be evenly split between the two, but the total capacity across the calls at each level of the tree must equal W.

   Since our n divides by 2 at every level, we have $2Wn = O(nW)$ time complexity.

   As for memory complexity, the optimal subsolution function allocates two arrays of size $2W$, meaning that each call has a $O(W)$ memory complexity. At each level of the tree, the total capacity sums to W, so the memory complexity is $O(W)$ across execution– ideally. Since my implementation allocates fresh vectors for each call of `OptimalSubsolution`, we technically pay for $O(\log nW)$ space for local copies.

4. Problem 4: Sparse Knapsack

   To demo this solution, run `make` in the `q4` directory.

   For the large problem, this implementation runs significantly faster than the $O(nW)$ and $O(W)$ memory complexity versions. For example, on the large problem the $O(nW)$ solution takes 8.14 seconds, the $O(W)$ solution takes 3.36 seconds, and the add-merge-kill implementation takes 0.59 seconds.

   This significant difference in running time is because the add-merge-kill algorithm does less work on average than the dense algorithms. Worst case scenario, the length of each sparse row in the sparse DP matrix doubles each time until it reaches the capacity W meaning that for the first $\log W$ rows the number of cells processed is less than the amount of work done at every row in the dense algorithm.

   In practice though, the rows will not grow that fast– for the large problem, the longest sparse row contains 3000 elements:

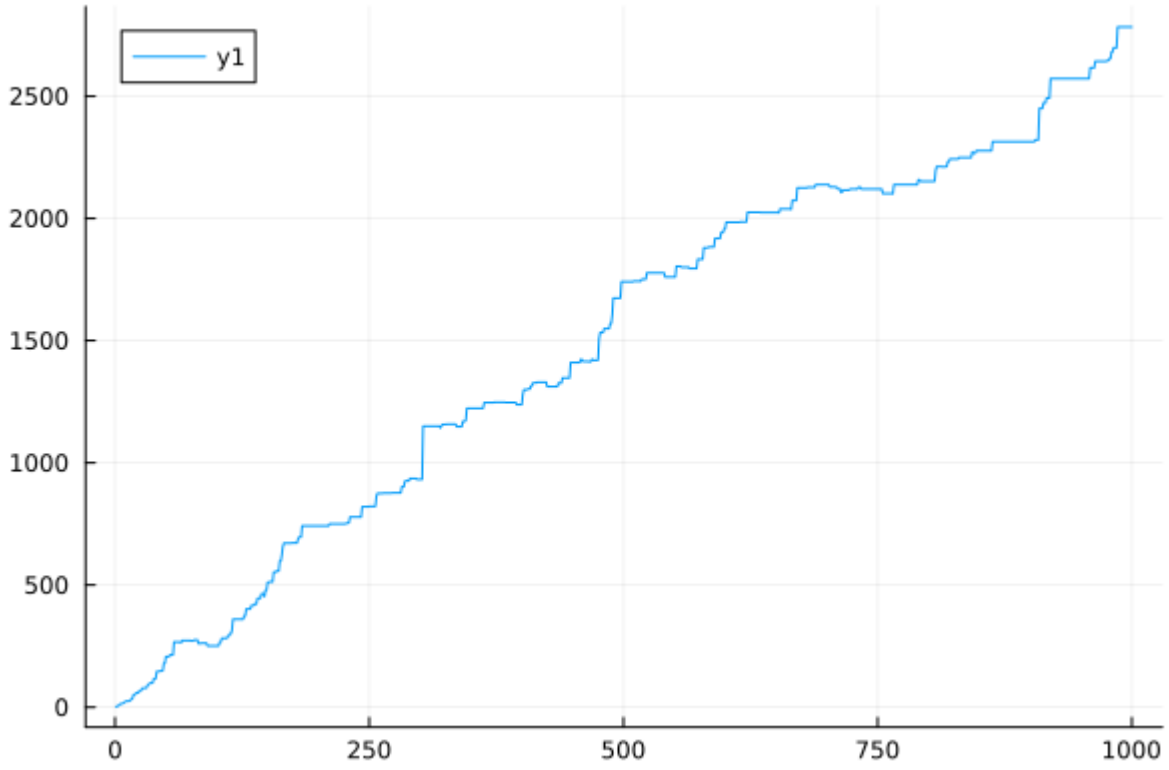

Figure 1: Sparse Row Length per Iteration

This means that, the add-merge-kill algorithm will very often do significantly less work per for of the table than the dense algorithm, which will always do $O(W)$ work for each row. Hence, the must better performance here.

5. Problem 5: Parallel Fast Fourier Transform

Algorithm 1 Parallel Fast Fourier Transform

function PFFT(a, ω)
  if length(a) $= 1$ **then**
    return a
  end if
  split a into e and o
  spawn $x = \text{PFFT}(e, \omega^2)$
  $y = \text{PFFT}(o, \omega^2)$
  sync
  for parallel $j = 0$ to $n/2 - 1$ **do**
    $v_j = x_j + \omega^j y_j$
    $v_{j+n/2} = x_j - \omega^j y_j$
  end for
end function

Since the FFT is a divide-and-conquer algorithm, it makes it pretty simple to parallelize, using spawns and syncs and a parallel for-loop. Since each sub-call to the FFT function is on an

independent half of the list, we can calculate the recursive call on each half in parallel, and the since each iteration of the for loop is independent of every other we can parallize that as well.

The work for the algorithm follows the recurrence $T_1(n) = 2T_1(n/2) + O(n)$, which by the master theorem yields $O(n \log n)$.

The span of the algorithm calculates the two recursive calls in parallel, and parallelizes the for loop, meaning that we only have to pay for one recursive call at each level and one parallel for loop. This yields the new recurrence $T_\infty(n) = T_\infty(n/2) + O(\log n)$, which we can't solve using the master theorem. However, from the master theorem corollary discussed in class, we know that $T(n) = T(n/2) + O(\log^k n)$ resolves to $T(n) = \Theta(\log^{k+1} n)$. That means that our span is $T_\infty(n) = O(log^2 n)$

Our parallelism would then be $\bar{p} = \frac{n \log n}{\log^2 n} = \frac{n}{\log n}$, which is great since n grows much faster than *logn* and we'll achieve linear speedup for a large number of processors with an algorithm that is still just as work efficient as the sequential.

6. Problem 6: Parallel Floyd-Warshall

Algorithm 2 Parallel Floyd-Warshall

```
function PFW(V, E)
    for parallel i = 1 to N do
        for parallel j = 1 to N do
            D(i, j) = w(i, j)
            if w(i, j) < ∞ then
                P(i, j) = i
            else
                P(i, j) = ∞
            end if
        end for
    end for
    for k = 1 to N do
        for parallel i = 1 to N do
            for parallel j = 1 to N do
                if D(i, j) > D(i, k) + D(k, j) then
                    D(i, j) = D(i, k) + D(k, j)
                    P(i, j) = P(k, j)
                end if
            end for
        end for
    end for
end function
```

The Floyd Warshall algorithms for All-Pairs Shortest Paths calculates the shortest paths between every vertex in a graph in $O(n^3)$ time and $O(n^2)$ memory (when implemented discarding previous layers of the solution tensor). This is accomplished through three nested for loops, where the outer loop iterates over $n \times n$ layers of the dynamic programming tensor and the two inner loops iterate over the individual elements of each tensor.

Since the two inner for loops depend only on the elements of the previous layer of the DP hypertable, they can be trivially parallelized. The same goes for the two initialization for loops at the top.

The span of the first two parallel for loops is $O(\log n + \log(n)) = O(\log n)$, because of the overhead of spawning threads and the constant time work done in each iteration.

The span of the inner two for loops of the triply nested loops is the same, and they must be repeated independently n times to build each layer of the tensor. Thus, the span for the triply nested loops is $O(n \log n)$, yielding and overall span for the algorithm of $O(n \log n + \log n) = O(n \log n)$

So the parallelism for the algorithm would be $\frac{T_1(n)}{T_\infty n} = \frac{n^3}{n \log n} = \frac{n^2}{\log n}$ which once again means we'll achieve meaningful speedup for a large number of processors on large inputs.