Assignment 4

Due: February 18, 2020

Kanyid, Bradon
bradon.kanyid@pdx.edu

Reimer, Daniel
daniel.reimer@pdx.edu

Your solutions must be typed (preferably typeset in LaTeX) and submitted as a hard-copy at the beginning of class on the day its due.
When asked to provide an algorithm you need to give well formatted pseudocode, a description of how your code solves the problem, and a brief argument of its correctness.

Problem 1: Maximum Subarray Sum The Maximum Subarray Sum problem is the task of finding the contiguous subarray with largest sum in a given array of integers. Each number in the array could be positive, negative, or zero. For example: Given the array $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ the solution would be $[4, -1, 2, 1]$ with a sum of 6.

(a) [5 points] Give a brute force solution for this problem with complexity of $O(n^2)$.

Solution

Pseudocode

```
genSubArray(xs, start, end, subarray):
  if end = len(xs):
    return subarray

  if start > end:
    return genSubArray(xs, 0, end+1, subarray)
  else
    subarray.append(xs[start .. end+1])
    return genSubArray(arr, start+1, end, subarray)
```

```
maxSubarray(xs):
  subArrays = genSubArray(xs, 0, len(xs), [])

  return max (map (sum, subArrays))
```

Description

This code solves the problem by first recursively finding the subarrays of a given array. Then, it calculates the sum of each subarray and takes the max. This works because given every single possible subarray, we can brute force test all the subarrays by finding the sum and taking the highest value found.

Efficiency

Generating the subarrays is done in $O(n^2)$ since there is a nested recursive call where both calls go over the entire array. To find the max of the subarrays is done in $O(n^2)$ as the subarray list now contains n^2 number of elements and the sum is mapped to each subarray. The max is then found in linear time. Therefore the overall effeciency $O(n^2)$.

(b) [10 points] Give a divide and conquer solution for this problem with complexity $O(n \log n)$.

Solution

Pseudocode

```
max_interval (xs, l_index, r_index):
  if l_index = r_index:
    return xs[l_index]

  midpoint = (r_index + l_index) / 2
  l_max = max_interval(xs[l_index .. midpoint], l_index, midpoint)
  r_max = max_interval(xs[midpoint .. r_index], midpoint, r_index)

  curr_sum = 0
  left_sum = - inf
  for i in xs[midpoint .. l_index]:
    curr_sum += xs[i]
```

```
    if curr_sum > left_sum:
      left_sum = curr_sum

  curr_sum = 0
  right_sum = - inf
  for i in xs[midpoint .. r_index]:
    curr_sum += xs[i]
    if curr_sum > right_sum:
      right_sum = curr_sum

  mid_max = left_sum + right_sum

  return max(l_max, r_max, mid_max)
```

Description

This code works by splitting the array into two subarrays and recursing down the left and right half and finding the max of the subarrays under each branch. Following, find the max of the subarrays that overlap each of the left and right branches. The max of the left branch, right branch, and the max between the two branches is then returned.

Efficiency

The overall effeciency is O(nlogn) since to calculate the left and right branches is overall O(nlogn) since the operation is essientally depth first search. Calculating the midpoint is done in two steps each only traversing over half the list each and therefore O(n) for that step. Overall the efficiency is O(nlogn).

(c) [10 points] Give a dynamic programming solution for this problem with complexity $O(n)$.

Solution

Pseudocode

```
maxSubarray(xs):
  curr = xs[0]
  max = xs[0]

  for x in xs
```

```
    curr = max (x, curr + x)
    max  = max (curr, max)

  return max
```

Description

This code works by iterating over the list, continuously summing each element and taking the max of the all-time max with the current max. The current max finds the max by comparing the current element against the partial sum with that element added. If the current sum goes below the current element, it effectively makes that element the next possible start of the subarray, by setting the current sum to just that element. *max* tracks the largest subarray seen over the entire array.

Efficiency

The overall efficiency is $O(n)$ since there is a single iteration over the list and the operations inside the iteration is finding the max of two values and summing two values which are both constant time operations.

Problem 2: Restaurant Placement A new restaurant chain is opening and you have been given the task of selecting the restaurant locations with the goal of maximizing their total profit.

The street network is described as an undirected graph $G = (V, E)$, where the potential restaurant sites are the vertices of the graph. Each vertex v has a non-negative integer value p_v, which describes the potential *profit* of site v. Two restaurants cannot be built on adjacent vertices. You are supposed to design an algorithm that outputs the chosen set $U \subseteq V$ of sites that maximizes the total profit $\sum_{v \in V} p_v$.

For parts (a)-(c), suppose that the street network G is *acyclic*, i.e. a tree.

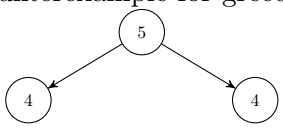
(a) [5 points] Consider the following *greedy* restaurant placement algorithm. Choose the highest profit vertex v_0 in the tree, breaking ties according to some ordering on vertex names, and put it into U. Remove v_0 and all of its neighbors from G. Repeat until no vertices remain. Give a coun-

terexample to show that this algorithm does not always give a restaurant placement with maximal profit.

Solution

This trivial counterexample shows that the greedy algorithm proposed would first take the parent node, worth a profit of 5, and discard both children worth 4. An optimal strategy for this tree would take the two leaves and discard the parent, for a total profit of 8.

Figure 1: Counterexample for greedy algorithm.



(b) [10 points] Give an efficient algorithm to determine a placement with maximum profit.

Solution

For each node, we need to evaluate the profit of keeping it (let's call this the K function) and the effect it has downstream (forcing others to be dropped), which we will call the D function.

For a leaf node, each are defined pretty straightforwardly:

$$K(v) = P_v$$
$$D(v) = 0$$

For parent nodes, the functions are more complex:

$$K(v) = P_v + \sum_{c \in v} D(c)$$
$$D(v) = 0 + \sum_{c \in v} \max(K(c), D(c))$$
where $c = v$'s children

Because the K and D functions are defined in terms of only its children, we can use dynamic programming to compute the children first, by doing a

post-order traversal, then using those results the next layer up. In order to do a post-order traversal, we first need to choose a root node, but since the graph is acyclic, it is safe to choose any node as the root.

Once you've calculated these results recursively from the leaves up to the root of the tree (via the aformentioned post-order traversal), the final evaluation of the root node's K and D functions will determine if there's more profit when we keep the root and drop it's children, or if there's more profit in dropping the root and keeping it's children. Take the max of K and D, and whichever is higher, determines the entire graph.

The main work being done is the post-order traversal, which has $O(n)$-time complexity.

(c) [5 points] Suppose that, in the absence of good data, the restaurant chain decides that all sites are equally good. The goal therefore is to simply find the placement with the maximum number of locations. Give a simple greedy algorithm for this case and argue its correctness.

Solution

Since we are maximizing the total number of restaurants, what we want to accomplish is to take all the leaves, removing parents as necessary. To do so, we would, similar to 2b, do a post-order traversal of the graph, creating a tree from it by picking any node as the root. Again, this is ok to do because of the acyclic stipulation on the graph.

Once we know we are at a leaf node, we mark it as a node to keep, and remove its parent, in order to meet the condition that there are no adjacent restaurants.

We can argue that this is correct because of the relationship between parent nodes and child nodes. Any child node has at most one parent. Any parent in the graph (now tree) can have N children, with a minimum of one. If we were to choose a parent node initially, we would be removing at least one child node to select that parent. In many cases, it would be more than one child. We are always in the position that we could substitute in one of those children in the parent node's place, and be in the same situation, without invalidating that child's siblings. That means choosing the child node is always at least as good as a parent node, and many times it is better.

Thus, this shows that starting by adding all the children to the solution set, and invalidating backwards, will bear the best result.

(d) [5 points] Suppose that the graph is arbitrary and not necessarily *acyclic*. Give the fastest correct algorithm you can for solving the problem.

Solution

The following algorithm is the fastest we could determine:

1. Calculate every subset of the graph
2. Remove subsets that contain restaurants on adjacent vertices
3. Compute total profit for each remaining subset
4. Take the max value from these subsets

Further analysis shows that this problem is a form of the Maximum Independent Set problem, and thus is NP-Hard. Thus, there is no known polynomial-time solution.