COMP305 Problem Set 1

Exercise 1-1

3, 2, 5, 1, 4, 6, 7, 13, 8, 12, 15, 14, 9, 10, 11

Exercise 1-2

1. $\Theta(n \log n)$
2. $\Theta(n)$
3. $\Theta(n^{\log \sqrt{5}})$
4. $\Theta(n^2 \log n)$
5. $\Theta(n^{\log_5 4})$
6. $\Theta(n^{\log_5 4})$
7. $\Theta(\log^5 n)$
8. $\Theta(\log^2 n \log \log n)$
9. $\Theta(\log \log n)$
10. $\Theta(n \log n)$

Problem 1-1

Part (a)

Suppose the following graph of three nodes: $V = \{v_1,\ v_2,\ v_3\}$; $E = \{(v_1, v_2),\ (v_2, v_3)\}$; $P = \{40,\ 50,\ 40\}$. A greedy algorithm would choose v_2 and end up with a profit of 50, although the optimal set is $\{v_1,\ v_3\}$ with a total profit of 80.

Part (b)

This is a variation of the Maximum Weighted Independent Set (MWIS) problem. For general graphs, this problem is NP-hard, whence there is no known efficient algorithm; it's also hard to approximate. However for an acylic graph i.e. a tree, there exists a deterministic solution in polynomial time. [1]

Brute Force

A straightforward algorithm would be:

- For a graph G, find all possible sets in G.
- For all subsets in G, find an independent set that maximizes the profits.
  Since the task is about finding a maximum-weighted independent set, the number of such sets I in a subtree T_i rooted at i is reduced. Although this would still guarantee the correct solution for a general graph, time complexity for it must be similar to $\Omega(2^n)$, which is already not in P. [2]

Adaptation of MWIS for trees

Since we're concerning ourselves with a tree rather than a generic graph, the tree can be arbitrarily rooted, and the tree can be traversed bidirectionally, from the root to leaves and vice versa. A major consequence is that the task of finding a maximizing independent set I in a subtree T_i is reduced to that subtree only, enabling us to formulate the task recursively — divide and conquer. From here on out, suppose we treat the graph as a tree, say T. We can define the maximal profit associated with a subtree easily. [3]

$P_i(I) = \sum_{i \in I} p_i \mid I \subset T_i$
i.e. $\text{total profit associated with the set } I$

$P_i = max\ \{\ P_i(I)\ |\ I - \text{independent set in } T_i \text{ and } v_i \in I\ \}$
$\bar{P}_i = max\ \{\ P_i(I)\ |\ I - \text{independent set in } T_i \text{ and } v_i \notin I\ \}$

where $v_{root} - \text{root node in T}$, $v_i - \text{node } i \text{ in tree } T$, $T_i - \text{subtree in T rooted at } v_i$, and $p_i - \text{profit associated with a node } v_i$. And so, the maximal weight of the whole tree becomes $P_{solution} = max\ \{\ P_{root}\ ,\ \bar{P}_{root}\ \}$. And the set that yields the maximum weight is $I_{solution} = argmax(P_{solution})$.

Using this, we can define the solution recursively:
Base Case. For a leaf node v_i, $I = \{v_i\}$. Hence, $P_i = p_i$ and $\bar{P}_i = 0$.
Recursive Case. For a non-leaf node v_i,
$P_i = p_i + \sum_{j\ \in\ \text{children}} \bar{P}_j$
$\bar{P}_i = \sum_{j\ \in\ \text{children}} max\{\ P_j\ ,\ \bar{P}_j\ \}$
$P_{solution} = max\{\ P_{root}\ ,\ \bar{P}_{root}\ \}$

Algorithm

Suppose a node is represented as follows:

```
class Node:
    weight: int,
    parent: Node,
    children: List[Node]
```

Then, a straightforward recursive algorithm is as follows: [4]

```
def include(node, path):
    current_path = path.copy()
    current_path.add(node)
    current_total = node.weight
    for child in node.children:
        temp_total, temp_path = exclude(child, current_path)
        current_total += temp_total
        current_path = current_path.union(temp_path)
    return current_total, current_path

def exclude(node, path):
    current_path = clone(path)
    current_total = 0
    for child in node.children
        total1, path1 = include(child, current_path)
        total2, path2 = exclude(child, current_path)
        max_total = max(total1, total2)
        max_path = path1 if max_total == total1 else path2
        current_path = current_path.union(max_path)
        current_total += max_total
    return current_total, current_path

root = # decide root
include_total, include_path = include(root, set())
exclude_total, exclude_path = exclude(root, set())
max_total = max(include_total, exclude_total)
max_path = include_path if max_total == include_total else exclude_path
```

This algorithm, however, grows similar to the recursive fibonacci algorithm. Nodes in layer i (one-indexed) of the tree, get passed to the function `include` $f(i)$ times and to `exclude` $f(i+1)$ times, where $f(n)$ is the n^{th} number in the fibonacci sequence and $f(0) = 0, f(1) = 1$.

Since `include` makes a single call to `exclude`, and `exclude` makes one call to itself and another call to `include`. These calls, "pile up" through the layers of the tree. So that the complexity of the solution becomes proportional to the

sum the of fibonacci numbers, considering that all operations inside of the two functions can be reduced to $O(1)$.

$$T(h) = c_{include} * F(h) + c_{exclude} * F(h+1)$$

$h -$ height of the tree
$F(n) -$ sum of the first n fibonacci numbers

We can't pin a single number for the height of the tree, but we know for a fact that in the worst case, the tree can be a linear tree (a path graph) with a height of n. Using the fact that $F(n) = f(n+2) - f(2)$ and Binet's formula [5],

$$T(n) = c_{include} * (f(n+2) - 1) + c_{exclude} * (f(n+3) - 1)$$

$$T(n) \in O\left(\frac{(1+\sqrt{5})^{n+3} - (1-\sqrt{5})^{n+3}}{2^{n+3}\sqrt{5}}\right)$$

Correctness

The algorithm would start from the root comparing potential profit for each child node, arriving and terminating at a base case, solving for a set of nodes that maximizes potential profit. The two separate functions ensure that the solution is an independent set.

Memoization

However, most of these calls to `include` and `exclude` are unnecessary. Just like the recursive fibonacci, dynamic programming strategies can reduce the complexity significantly. In particular, a fast store and lookup array can reduce the **MWIS** problem to $O(n)$. [6] Specifically, `include` and `exclude` would be called once for each node leading to a complexity of $T(n) = (c_{include} + c_{exclude}) * n = C * n \in O(n)$.

Space Complexity

The solution path is found in a single pass while the algorithm is traversing the tree, allocating a new set at each recursive call, hence a space complexity of $O(n^2)$. This can also be reduced to $O(n)$, by finding the maximum weight independent set in a second-pass. Space complexity for the tree itself can also be reduced, as there are more efficient representations.

Part (c)

The solution to this can be a modified version of the previous algorithm. In fact, this is the **maximum independent set problem**, a special case of **MWIS**, where all weights are equal. As before, this problem is NP-hard for general graphs, but a solution in P exists for trees. Setting the weights to unit in the previous algorithm, and again, by utilizing memoization, we can achieve a complexity of $O(n)$ for this solution as well. [7]

Part (d)

For graphs that are not necessarily acylic, this problem is NP-hard, and approximation algorithms do not guarantee a correct solution. Hence, the brute-force solution described in part (a) is on around the same order of growth. This algorithm would guarantee a correct solution since it would examine every subset of $G = (V, E)$, a total of $2^{|V|}$ subsets, and find an independent set among them that maximizes the weights. In the worst case, deciding whether a set is independent requires going through each node, and checking all of its neighbors, hence $O(|V|^2)$. Calculating the total weight is a sum over all the node weights — $O(|V|)$. The complexity of the brute-force solution would be $T(G) = c_1 * |V|^2 * 2^{|V|} + c_2 * |V| * 2^{|V|} \in O(|V|^2 * 2^{|V|})$. Granted, optimizations can greatly help. However, escaping the exponential space is impossible.[2-1]

Problem 1-2

Part (a)

Maximum distance between two locations within or on the boundary of one such square can be $\sqrt{(1/2)^2 + (1/2)^2} = 1/\sqrt{2} < 1$.

Part (b)

This is a variation of the **Closest Pair of Points** problem. [8] Let's say all the requests (points) are in P, where $P \subset \mathbb{R} \times \mathbb{R}$ and for any two points in P, (x_1, y_1) and (x_2, y_2), $x_1 \neq x_2$ and $y_1 \neq y_2$. We constrain L_x and L_y as follows:

$L_x = \{(x_i, y_j) \in P \mid x_1 \leq x_i \leq x_n\}$
$L_y = \{(x_i, y_j) \in L_x \mid y_1 \leq y_j \leq y_n\}$

Iterative Solution

Since we have an ordering for all points with respect to their x and y coordinates, we can check the 4 closest points per point. So for a point u and one other point closest to it v, check if they are too close if $|u_x - v_x| \leq 1$ or $|u_y - v_y| \leq 1$.

Correctness

Since both of these lists are sorted, we can utilize binary search and we really only need to check the 4 closest requests around each of the requests and report a pair that is too close to each other. Suppose we iterate through each location in the sorted L_x, since there are no two requests with the same x or y coordinates, binary search can correctly find the corresponding index of the location in L_y.

Complexity

Since, the lists are sorted, binary search can find an element in $O(\log_2 N)$. Hence, total complexity is $O(N * \log_2 N)$.

Algorithm

The algorithm can be described with the following pseudocode: [9][4-1]

```
Location = Tuple[float, float]
Pair = Tuple[Location, Location] | None

def close(a: Location, b: Location) -> bool:
    return distance(a, b) <= 1

def close_pair_iterative(lx: List[Location], ly: List[Location], N: int) -> Pair:
    for i in 0..(N-1):
        # check two neighbors along the x-axis
        if close(px_prev := lx[i - 1], px := lx[i]):
            return px_prev, px
        if close(px := lx[i], px_next := lx[i + 1]):
            return px, px_next

        # check two neighbors along the y-axis
        j = search(ly, lx[i])
        if j > 0 and close(py_prev := ly[j - 1], py := ly[j]):
            return py_prev, py
        if j < N - 1 and close(py := ly[j], py_next := ly[j + 1]):
            return py, py_next
    # set valid
    return None
```

Recursive Solution

We can also formulate the problem recursively using divide and conquer. [10] At each step, divide L_x in half with respect to the middle request m to $L_{x_{left}}$ and $L_{y_{right}}$, enforcing the same constraints. If either subset has two requests p_1 and p_2, check if they are too close if $|p_{1_x} - p_{2_x}| \leq 1$. If either subset has one request, simply return. Otherwise, check nodes that are within a distance of 1 both to the left and to the right of m and continue dividing. Dividing the task and going through points around midpoint is $O(n)$ yielding the following recurrence relation: $T(n) = 2\,T(n/2) + O(n)$, which, according to the master theorem is on the order of $O(n \log n)$. [11]

Part (c)

Modifying the iterative solution would be more intricate in contrast to the recursive solution, where not a lot of fundamental changes are required. We can still recurse by dividing L_x in half, and still examine a number of points proportial to n, remaining on the same order of growth of $O(n \log n)$. Adapt the base case to a trio of points rather than a pair. If no such trio is found, look for a trio within the thin strip centered at midpoint. [12]

---

1. https://en.wikipedia.org/wiki/Independent_set_(graph_theory)↩
2. In fact, an exact algorithm runs in $O(1.1996^n)$.↩↩
3. https://doi.org/10.3390/math8020285↩
4. Its python equivalent was tested with python 3.11.↩↩
5. https://mathworld.wolfram.com/BinetsFormula.html↩
6. https://www.altcademy.com/blog/identify-the-maximum-weighted-independent-set-in-a-tree/↩
7. https://www.geeksforgeeks.org/largest-independent-set-problem-using-dynamic-programming/↩
8. https://en.wikipedia.org/wiki/Closest_pair_of_points_problem and Textbook, chapter 33.4↩
9. Arrays are zero-indexed just like in Python.↩
10. https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/↩
11. Textbook, chapter 4.↩
12. https://stackoverflow.com/questions/7539623/closest-group-of-3-points↩