Course: Advanced Algorithms by Dr. Zarei

Homework: HW1

Name: Mohammad Mohammadi

Student ID: 402208592

Question 1

Consider the following problem in the streaming model. Median: Given a stream S = <a1,…,am> of m distinct items over the universe [n], compute a median of S. Prove that for m < n/2 any deterministic streaming algorithm that solves Median exactly must use Ω(m log(n/m)) bits in the worst case.

Answer 1

1. Combination Argument:
   - Firstly, consider the number of possible streams S S that can be created with m distinct items out of n. This is given by $\binom{n}{m}$, the number of ways to choose m items from n.
2. Lower Bound:
   - To distinguish between these different streams using a streaming algorithm, you would need at least $\log\binom{n}{m}$ bits. Using Stirling's approximation and simplifying, this value is in the order of mlog(n/m).
3. Median Determination:
   - Now, for m < n/2, determining the median becomes more challenging. This is because the median could lie in a large range, especially as m becomes much smaller than n. For many different streams with different sequences of numbers, the median might be the same. As a result, the algorithm needs to remember more about the stream to distinguish between these different cases and to compute the median accurately.
4. Conclusion:
   - Given the nature of the median problem and the lower bound, any deterministic streaming algorithm that wishes to solve the Median problem exactly must use at least Ω(mlog(n/m)) bits in the worst case.

For the combinatorial nature of the problem and the principles of information theory, to provide a lower bound on the space complexity for deterministic streaming algorithms solving the Median problem, we need at least Ω(mlog(n/m)) bits in the worst case.

Question 2

Consider the following problem in the streaming model. Element Uniqueness: Given a stream S = <a1,...,am> over the universe [n], with m <= n, decide if all items in S are distinct. Either prove that any deterministic streaming algorithm that solves Element Uniqueness exactly must use Ω(m log(2n/m)) bits in the worst case, or give a deterministic streaming algorithm that solves Element Uniqueness exactly using a sub-linear number of bits. If you give an algorithm, you should also prove its correctness and analyze the number of bits of storage it uses.

Answer 2

We have to prove that any deterministic streaming algorithm requires Ω(mlog(2n/m)) bits in the worst case.

Let's consider the scenario where we are trying to determine the space complexity of the streaming algorithm for this problem.

One way to think about it is in terms of the number of possible distinct streams we can have. For the worst-case scenario, we consider the case when all elements are distinct, i.e., m distinct numbers are chosen out of n. The number of such streams is:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

In the streaming model, we want to distinguish between all these different streams, i.e., every distinct stream must map to a unique memory configuration (because if two different streams led to the same memory configuration, our algorithm wouldn't be able to distinguish between them, and it would be incorrect).

If we have B bits of memory, there are 2^B possible memory configurations. To distinguish between all distinct streams, we need:

$$2^B \geq \binom{n}{m}$$

Taking the logarithm of both sides:

$$B \geq \log_2 \binom{n}{m}$$

Using Stirling's approximation in the worst-case scenario ,which is when m=n/2 (because $\binom{n}{m}$ is maximized when m is closest to n/2) and some manipulations:

$$\log_2(\binom{n}{m}) = \Omega(m \log(2n/m))$$

Note that when m = n/2, mlog(2n/m) is very close to nlog(n) in terms of order of growth. Thus, it hinted strongly at a growth rate that is related to nlog(n) in the worst case, which is consistent with the lower bound for the streaming problem. Further manipulations with tighter bounds and considerations may lead to the more exact form, but the intuition remains the same.

This gives us a lower bound on the number of bits needed.

Question 3

Consider the following problem in the streaming model. Two Missing Items: Given a stream S = <a1,...,an-2> over the universe [n] in which all items in S are different, compute the items j1,j2 in [n] that are missing from S. Note that only streams of length n - 2 are considered and that all items in the stream are distinct, which implies there are exactly two missing items. Either prove that any deterministic streaming algorithm that solves Two Missing Items exactly must use Ω(n) (n) bits in the worst case, or give a deterministic streaming algorithm that solves Two Missing Items exactly using a sub-linear number of bits. If you give an algorithm, you should also prove its correctness and analyze the number of bits of storage it uses.

Answer 3

1. As numbers stream in, keep a running total of their sum and their squares.
2. When the stream ends, calculate the expected sum and sum of squares for all numbers from 1 to n.
3. The difference between the expected sum and our running sum will be the sum of the two missing numbers.
4. Similarly, the difference between the expected sum of squares and our running sum of squares will give us the sum of the squares of the two missing numbers.

Using these two differences, we can deduce the two missing numbers.

Given that i1 and i2 are the missing numbers, we know:

- i1 + i2 = (Actual Sum) - (Calculated Sum)
- i1^2 + i2^2 = (Actual Sum of Squares) - (Calculated Sum of Squares)

From the above two equations, we can derive i1 and i2. The quadratic nature of these equations will give us two potential values, which are exactly the two missing numbers.

Space Analysis:

For the sum, it can take up to sum of 1 to n which takes O(logn) bits of space to represent. And also for sum of squres we will require up to O(log(n^2)) bits to represent which is of O(2logn) which is again of O(logn). Thus, the total space required is O(logn) + O(logn) = O(logn). This is a sub-linear number of bits!

Correctness:

The correctness comes from the algebraic properties of the summation of natural numbers and their squares. We're leveraging these properties to find the two missing numbers based on the differences between the expected and obtained sums and sum of squares.

In conclusion, we have successfully designed a deterministic streaming algorithm that solves the Two Missing Items problem using O(logn) bits, which is sub-linear in space.

Example:

For a simple illustration, consider the universe [5] and the stream is 1, 3, 5.

- The expected sum is 1+2+3+4+5=15.
- Our stream's sum is 1+3+5=9.
- So, the sum of the missing numbers is 15−9=6.

Using a similar logic with squares, we can deduce the two missing numbers as 2 and 4.

In conclusion, we can solve the problem using sub-linear space O(logn) bits.

Question 4

Present a parallel algorithm for removing the duplicate items that appear in a sequence. The input to the algorithm is a sequence, and the output is a new sequence containing exactly one copy of every item that appears in the input sequence. It is assumed that the order of the items in the output sequence does not matter.

Answer 4

Based on the pdf file on Parallel Algorithms page 28, we will use hashing for this matter as below:

Hashing Approach:

1. A hash table is created with a prime number of entries. The size is approximately double the number of items in the set V. The reason for using a prime size is that it makes the design of a good hash function easier and reduces the chance of collisions.
2. A hash value is computed for each item in the set V, and there's an attempt to write the index of the value into the hash table at the computed hash value's position.
3. If multiple values hash to the same memory location, only one of the values is written (based on an arbitrary concurrent write model).
4. If a value is successfully written into the hash table, its index is termed a "winner". Winners represent unique values.
5. The values that could not be written into the hash table because they had a collision with a value of the same kind are termed as "losers".
6. Among the losers, a distinction is made:
   - Some items are defeated by another item with the same value. These are duplicates.
   - Others are defeated by items with a different value. This means the hash function caused a collision.
7. Items that collided with a duplicate are set aside since they are duplicates. Items that collided because of the hash function are retained, and the process is repeated with a different hash function.

To achieve a duplicate-free sequence:

1. Winners are added to the sequence as they represent unique values.
2. Among the losers, true duplicates are set aside, while others are hashed again using a different hash function until all items are processed.

The algorithm:

```
ALGORITHM: REMOVE_DUPLICATES(V)
1.  m := NEXT_PRIME(2 ∗ |V|)
2.  TABLE := distribute(−1, m)
3.  i := 0
4.  RESULT := {}
5.  while |V| > 0
6.      TABLE := TABLE ← {(hash(V[j], m, i), j) : j ∈ [0..|V|]}
7.      WINNERS := {V[j] : j ∈ [0..|V|] | TABLE[hash(V[j], m, i)] = j}
8.      RESULT := RESULT ++ WINNERS
9.      TABLE := TABLE ← {(hash(k, m, i), k) : k ∈ WINNERS}
10.     V := {k ∈ V | TABLE[hash(k, m, i)] ≠ k}
11.     i := i + 1
12. return RESULT
```

The remove_duplicates(v) algorithm is designed to eliminate duplicate elements from a given list v using a parallel approach. Initially, the algorithm creates a TABLE of a size determined by the next prime after double the length of v. This table is populated with default values.

The main logic revolves around a hashing function, which, for each element of v, computes a unique index. The algorithm then examines the TABLE for conflicts using these hash values. If a conflict (i.e., a duplicate) is detected, only the first occurrence (the "winner") is retained. The remaining duplicates are then removed in subsequent iterations.

As the process iterates over the list v, the results accumulate in the RESULT list. This is done in parallel, with each element being processed simultaneously, thus speeding up the duplicate removal process.

By the end of the algorithm, RESULT contains the input list v without any duplicates.

Question 5

Consider the problem of labeling the connected components of an undirected graph. The problem is to label all the vertices in a graph G such that two vertices u and v have the same label if and only if there is a path between the two vertices. Propose a parallel algorithm for this problem.

Answer 5

Based on the pdf file on Parallel Algorithms, page 37, section 4.3.2, we will use Deterministic

Graph Contraction, also said to be Tree-based Graph Contraction, using a concurrent write model for this matter as below:

1. Base Case:
   - If there are no edges (|E| = 0), then each vertex is its own connected component, and you simply return the existing labels.
2. Hooks Definition:
   - For each edge (u, v) ∈ E, if u > v, then it forms a hook. This step ensures that each edge has a unique direction (from a smaller to a larger vertex, based on some ordering or vertex ID).
3. Updating Labels:
   - The labels of the vertices are updated based on the hooks. Essentially, a vertex v will now "point" to u if (u, v) is a hook. This helps in setting up a tree structure, which is used for contraction.
4. Point to Root:
   - Each vertex traces its way up its chain of hooks to the root of its tree. The POINT_TO_ROOT function facilitates this. Once the root is found, all nodes in that tree (or component) will point directly to the root. This function may be implemented using a series of parallel steps where each vertex repeatedly updates its label to point to its parent's label.
5. Define New Edge Set E:
   - Edges in E are formed based on the new labels. An edge (u, v) exists in E if and only if the labels of u and v are different. This step essentially moves to the next level of contraction by defining inter-component edges.
6. Recursion:
   - The algorithm then recurses on the newly defined graph with updated labels and edge set E.

The algorithm:

```
ALGORITHM: CC_TREE_CONTRACT(LABELS, E)
1. if (|E| = 0)
2.     then return LABELS
3. else
4.     HOOKS := {(u, v) ∈ E | u > v}
5.     LABELS := LABELS ← HOOKS
6.     LABELS := POINT_TO_ROOT(LABELS)
```

```
    E' := {(LABELS[u], LABELS[v]): (u, v) ∈ E | LABELS[u] ≠ LABELS[v]}
    return CC_TREE_CONTRACT(LABELS, E')
```

In the context of the concurrent write model for the goal of parallelism:

Concurrent Writes:

- Multiple threads or processes can update the labels concurrently. The deterministic nature of the hooks (based on the condition u > v) ensures that there's no race condition when updating the labels.

Parallelism:

- Most of the steps in the algorithm (hook formation, label update, and edge set formation for E) can be executed in parallel for each vertex or edge.

Synchronization:

- After each major step (like updating labels or defining E), there may need to be a barrier synchronization to ensure that all processes have completed their updates before the next step starts.

The complexity of the CC_TREE_CONTRACT algorithm is a function of both its depth (how many times it recursively contracts) and the work done at each level of contraction.

1. Depth of Recursion: Each contraction ideally reduces the size of the graph. In the best case, the algorithm can halve the size of the graph with each contraction. So, the depth of the recursion would be O(log n), where n is the number of vertices. However, it's worth noting that in the worst-case scenario, the graph might not contract as efficiently, and the depth could be larger.
2. Work at Each Level: In each recursion:
    - Creating hooks requires O(m) operations, where m is the number of edges.
    - The POINT_TO_ROOT operation, in a parallel setting, can be accomplished in O(log n) time using a parallel prefix operation (O(nlogn) for none parallel prefix operation).
    - Forming the new edge set E takes O(m) time.

Given that the depth is O(log n) and work at each level is dominated by O(m), the overall time complexity in a parallel setting is O(m log n).

But attentioning to the example provided in Fig. 11, on page 39 of the book, in the worst case, the total work can be O((m+nlogn)logn) and depth O(log^2 (n)).

Overall, this deterministic tree contraction algorithm effectively leverages the power of parallelism in the concurrent write model to find connected components efficiently. It reduces the graph's size at each recursive step, ensuring rapid convergence towards identifying all components.

Question 6

Discrete Fourier Transform (DFT) has a long history of parallel algorithms. Propose a method to parallelize the Fast Fourier Transform (FFT) algorithm for solving the DFT.

Answer 6

Based on the pdf file on Parallel Algorithms, page 54, section 7.2, we will use a algorithm described as below:

Parallel FFT Algorithm:

This algorithm is a version of the FFT designed to be executed in parallel (reducing the time taken by reducing the depth in comparison to work):

1. First, it checks the size (or length) of the input array A. This length is n.
2. If the size of the array is just 1, then the function returns that single element. This is the base case for the recursion.
3. If not, the function breaks down the task into two parts:
    - Compute the FFT of the even-indexed elements.
    - Compute the FFT of the odd-indexed elements.
4. These two tasks are done simultaneously ("in parallel"), meaning they are computed at the same time to speed up the process.
5. After computing the FFT for the even and odd elements, the results are combined together in a specific way that involves some complex number math (the e^(2πij/n) part).
6. The combined result is then returned.

Performance:

The parallel FFT does the same amount of overall work as the regular FFT, which is O(n log n). This notation means that the time it takes grows in proportion to n times the logarithm of n. However, because it's executed in parallel, its "depth" (or the longest time you'd have to wait for any single chain of operations to complete) is reduced to O(log n).

In simple terms, this parallel version of FFT can be much faster than the standard version, especially when n (the number of data points) is large.

```
ALGORITHM FFT(A)
1. n := length of A
2. if n is 1 then
     return A
3. else
4.     in parallel do:
5.         EVEN := FFT(elements of A at even indices)
6.         ODD := FFT(elements of A at odd indices)
7.     for j from 0 to n/2 do:
         Combine EVEN[j] and ODD[j] using the formula:
         {EVEN[j] + e^(2πij/n) * ODD[j]}
        end for
8.     return the combined result
end
```

The algorithm recursively divides the input into even and odd indexed elements and computes the FFT on those subsets. After computing the FFT for even and odd elements, the results are combined in a specific manner involving complex number calculations.

Question 7

Suppose we have joined an online shopping site, and that there are n cars that we are rather interested in. We would like to buy the best one. (We are assuming that cars are comparable, and that there is a consistent way, after checking two cars, to decide which one is better.) We could rent all of them, one after the other, for one day and then pick the best, but the rule is that we cannot rent simultaneously and after one day of rent either we choose the car or ignore it forever! How can we maximize the probability of ending up buying the best car? Propose a competitive algorithm and analyze its ratio.

Answer 7

Based on the pdf file on Optimization from Stanford University, this is like The Secretary Problem, section 2 of the file. It is also called The Marriage Problem, which is a classic problem in probability theory and optimal stopping theory. The aim is to select the best option from a sequence of options when the options are presented one at a time, and once an option is passed, it cannot be revisited.

To design a competitive algorithm for the online car shopping problem we define the algorithm as below:

1. Observation Phase: For the first r of the n cars (where r is a value we'll determine), merely observe them and determine their quality. Don't choose any of them. This period is used to establish a reference standard.
2. Selection Phase: After observing the first r cars, pick the next car that is better than all the r cars observed so far. If no such car appears in the remaining (n-r) cars, then the strategy fails, and we don't get the best car.

The question now is: What is the optimal value of r?

The optimal value of r that maximizes the probability of choosing the best car turns out to be approximately n/e, where e is the base of the natural logarithm (approximately 2.71828). [This is chosen based on the descriptions the pdf file has provided on the matter!]

The competitive ratio (i.e., the probability of successfully choosing the best car using this strategy) is roughly 1/e or about 37%. This means that by following this strategy, you have about a 37% chance of choosing the very best car out of the n cars.

Proof:

Consider the case where the best car is the (r+1)th car. The probability that this happens is 1/n. In this case, the strategy will succeed if and only if the second-best car is among the first r cars. The probability of that happening is r/n. Therefore, the probability of both events happening is r/n^2.

Now, if the best car is the (r+2)th car, then the strategy will succeed if both the best and the second-best cars from the first (r+2) cars are among the first r cars. This probability is (r(r-1))/(n(n-1)).

You can continue this logic for all positions after r, and then sum these probabilities to get the overall success probability. The value of r that maximizes this sum is roughly n/e.

To make this decision practically for a real-world shopping site:

1. Look at roughly n/e cars just to get a sense of the market (where n is the total number of cars you're interested in).
2. After that, go for the first car you find that's better than all the cars you've observed so far in the observation phase.

This strategy only gives you the highest probability of getting the best car, but it doesn’t guarantee it! Therefore, we will choose the last car that we see, even if it is not the best we have seen so far.

Competitive Ratio:

To find the competitive ratio, we should compare this algorithm's performance to the offline optimal. The offline optimal, given that it knows all car qualities beforehand, will always choose the best car with probability 1.

So, the competitive ratio R is:

```
R(r)= (Expected performance of online algorithm with parameter r) /
(Performance of Offline Optimal)
```

The choice of r that maximizes R(r) will be the optimal parameter for the online algorithm to ensure the best competitive ratio. Which in our scenario is (n/e).

Given our choice of the observation phase as n/e, the competitive ratio will not approach 1, meaning the algorithm will not be optimal. However, it strikes a balance in terms of not observing too many cars and still having a decent chance at picking the best car.

However, it's worth noting that this approach won't yield a particularly high competitive ratio.