Assignment 1 - Homework Exercises on Approximation Algorithms

Duy Pham - 0980384
Maciej Wikowski - 0927420
Pattarawat Chormai - 0978675

October 6, 2015

IO.I-1

(i)

In this problem, we will show that when $m = M/B + 1$, if we have the optimal replacement policy, then it performs only $O(n/B + \sqrt(n))$ I/Os.

Indeed, when $M > B$ (which is obvious) and $m > M$, the array can be separated into blocks as being shown in figure 1.

Basically, we move in the column order, then each step in the first column requires an I/O. In each column, after filling the memory, there is still one more item left. This is when we need the replacement policy. The optimal replacement method (sometimes called MIN) removes the block that has the longest time to be reused again in the future. In this particular setting, the MIN will remove the block which the last item has the smallest index. E.g. the 3^{rd} selected block in figure 1.

If B does not divides m, then each column in the array has $(m-1)/B$ blocks which the last item index is that column index, which are also the blocks that have the longest time to be reused in the future. Moreover, these blocks are indeed never used again in the same row. Therefore, the number of I/Os that MIN performs is equal to the number of I/Os that the row order performs, plus an extra number of I/Os for the remaining column of size $s < B$ (because B does not divides m).

We have m rows, and $m/B + 1$ I/Os for each row. Thus the number of I/Os is:

$$(\frac{m}{B} + 1)m = O(\frac{n}{B} + \sqrt{n})$$

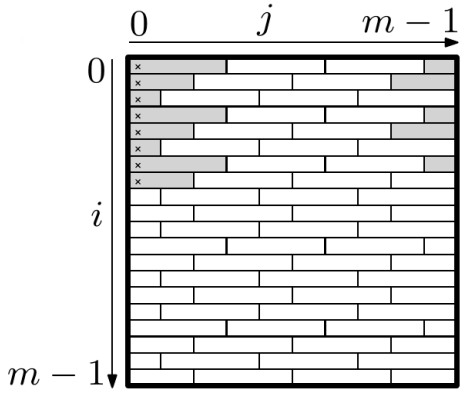

Figure 1: Blocks loaded into an 2-dimensional array

If B divides m, then we do not have the distribution as being shown in figure 1. Instead, all loaded blocks in a columns have the same ending index. Then the optimal replacement here is the Most Recently Used, that is, the policy evicts the latest used block. The array is now divided into m/B columns of size B. Each such column requires m I/Os for the first "real" column, and 1 I/O for each of the remaining columns. In general, the number of I/Os is:

$$\frac{m}{B}(m + B - 1) = \frac{m^2}{B} + \frac{m(B-1)}{B} = O(\frac{n}{B} + \sqrt{n})$$

(ii)

In case $m > 2M/B$ and m is significantly larger than M, then there are more than $2M/B$ blocks for each column.

If we traverse the array in column-major order and the OS uses the optimal replacement strategy, then at the end of each column, all elements in the first M/B blocks have been removed to get the space for the latest elements in the column. Therefore, whatever replacement method we use (even the optimal one), we cannot reuse any block when we move to the next column. Therefore, we need to load a new block every time we want to fetch a new item. Thus, the I/O complexity is $\Omega(n)$.

If the OS uses the Most Recently Used replacement policy, then we can keep $m/2$ items for the first column, but we have to load-and-clear every item in the remaining $m/2$ items. Therefore, the I/O complexity is still a factor of m^2, which is $\Omega(n)$.

IO.I-2

(i)

Let $T(n)$ denote I/O's of the problem and T_x, T_y and T_z denote the number of I/O's of matrices X, Y and Z respectively. For each cell in Z, we need

$$\begin{aligned} T_x &\leq \frac{\sqrt{n} + 2(B-1)}{B} \\ T_y &\leq \sqrt{n} \\ T_z &= 1 \end{aligned}$$

Then, the total number of I/O's we need for computing a cell of Z is :

$$\frac{\sqrt{n} + 2(B-1)}{B} + \sqrt{n} + 1$$

Therefore, the total number of I/O's that we need for computing the product $Z = XY$:

$$\begin{aligned} T(n) &\leq (\frac{\sqrt{n} + 2(B-1)}{B} + \sqrt{n} + 1)n \\ &\leq \frac{n\sqrt{n} + 2n(B-1)}{B} + n\sqrt{n} + n \\ &= O(n\sqrt{n}) \end{aligned}$$

(ii)

If Y is stored in $column - major$ order. For each cell, we will use

$$T_y = \frac{\sqrt{n} + 2(B-1)}{B}$$

. Therefore, the total number of I/O's is :

$$\begin{aligned} T(n) &\leq (2\frac{\sqrt{n} + 2(B-1)}{B} + 1)n \\ &\leq \frac{2n\sqrt{n} + 4n(B-1)}{B} + n \\ &= O(\frac{n\sqrt{n}}{B}) \end{aligned}$$

(iii)

Assume matrices X, Y and Z are small enough which can be loaded into the main memory. As a result, we can compute the product of $Z = XY$ without using any I/O's while computing. However, before doing that, we still need to perform some I/O's for loading data and writing the result back.

We know that the number of I/O's for loading or writing a matrix into the main memory is :

$$\sqrt{n}(\sqrt{n}/B + 2)$$

Since in the base case, we need 2 matrices, namely X and Y, in main memory and after finishing computing we need to write the result, Z, back to external memory. Hence, the I/O's complexity of the base case is:

$$3\sqrt{n}(\sqrt{n}/B + 2)$$

Because our problem is divided into 8 subproblems with size 4 time smaller and need to write the result back after finishing. Then, we can formulate the recurrence of I/O's complexity of this algorithm.

$$T(n) = \begin{cases} 3(\frac{n}{B} + 2\sqrt{n}) & \text{if } n \leq \frac{M}{3} \\ 8T(\frac{n}{4}) + O(\frac{n}{B}) & \text{otherwise} \end{cases}$$

Next, we then solve this recurrence function.

$$\begin{aligned} T(n) &= 8T(\frac{n}{4}) + O(\frac{n}{B}) \\ &= 8(8T(\frac{n}{4^2}) + O(\frac{n}{4B})) + O(\frac{n}{B}) \\ &= 8^2T(\frac{n}{4^2}) + 3O(\frac{n}{B}) \\ &= 8^2(8T(\frac{n}{4^3}) + O(\frac{n}{4^2B})) + 3O(\frac{n}{B}) \\ &= 8^3T(\frac{n}{4^3}) + 7O(\frac{n}{B}) \\ &= ... \\ &= 8^\alpha T(\frac{n}{4^\alpha}) + (2^\alpha - 1)O(\frac{n}{B}) \end{aligned}$$

We reach the base case when $\alpha = \log_2 \sqrt{\frac{3n}{M}}$. Then we can derive

$$\begin{aligned}
T(n) &= 8^{\alpha} T\Big(\frac{n}{4^{\alpha}}\Big) + (2^{\alpha} - 1) O\Big(\frac{n}{B}\Big) \\
&= 2^{3\log_2 \sqrt{\frac{3n}{M}}} T\Big(\frac{M}{3}\Big) + (\sqrt{\frac{3n}{M}} - 1) O\Big(\frac{n}{B}\Big) \\
&= \frac{3n\sqrt{3n}}{M\sqrt{M}} T\Big(\frac{M}{3}\Big) + O\Big(\frac{n\sqrt{3n}}{B\sqrt{M}}\Big) - O\Big(\frac{n}{B}\Big) \\
&= \frac{3n\sqrt{3n}}{M\sqrt{M}} \Big(3\Big(\frac{M}{3B} + 2\sqrt{\frac{M}{3}}\Big)\Big) + O\Big(\frac{n\sqrt{3n}}{B\sqrt{M}}\Big) - O\Big(\frac{n}{B}\Big) \\
&= \frac{3n\sqrt{3n}}{M\sqrt{M}} \Big(\frac{M}{B} + 6\sqrt{\frac{M}{3}}\Big) + O\Big(\frac{n\sqrt{3n}}{B\sqrt{M}}\Big) - O\Big(\frac{n}{B}\Big) \\
&= \frac{3n\sqrt{3n}}{B\sqrt{M}} + \frac{18n\sqrt{n}}{M} + O\Big(\frac{n\sqrt{3n}}{B\sqrt{M}}\Big) - O\Big(\frac{n}{B}\Big) \\
&= O\Big(\frac{4n\sqrt{3n}}{B\sqrt{M}}\Big) + \frac{18n\sqrt{n}}{M} - O\Big(\frac{n}{B}\Big)
\end{aligned}$$

According to $TallCacheAssumption$, we know that $M = \Omega(B^2)$. Therefore, the I/O's complexity of the algorithm is $O(\frac{n\sqrt{n}}{B\sqrt{M}})$.

(iv)

Claim. The algorithm from (iii) has better spatial locality.

To prove the claim, we know that the algorithm from (iii) does not perform any I/O's until it reaches the base case. When it is at the base case, a subproblem fits into the internal memory, and no blocks are evicted until the subproblem is complete. Each loaded block contains more than 1 useful variable for the problem, while the algorithm from (i) uses only 1 variable from each block in matrix Y before that block is evicted.

Claim : The algorithm from (iii) has better temporal locality.

To prove the claim, assume t is the time that an element of Y is used by both algorithm. The algorithm from (i) will use the variable again after $t + m - 1$ iterations. While the algorithm from (iii) will use again, roughly less than $t + m/2$, because its problems are divided into 4 pieces and each subproblem is finished before moving to a bigger subproblem.

IO.1-3

(i)

The recurrence formula of I/O's complexity of a binary search is as follow:

$$T(n) = T(n/2)$$

The base case is $T(n) = 1$ where $n \leq B$.

The worst case of a binary search is when each step of comparison, we need to do an I/O to load the next center until the 2 consecutive centers are in the same block, which means the remaining array has to fit in 1 block. Suppose that after k steps of binary searching, we reach that state, then both the left and right subtrees are inside 1 block:

$$2\frac{n}{2^k} \leq B \iff \frac{n}{2^k} \leq \frac{B}{2}$$

Then,

$$k \geq \log_2 \frac{2n}{B} = \log_2 \frac{n}{B} + 1 = O(\log_2 \frac{n}{B})$$

This means we have to do $O(\log_2(n/B))$ I/Os before everything is inside the memory.

(ii)

To avoid the case that we perform an I/O's every time we make a comparison, we can organize the blocks differently. Starting from the root node of the binary search tree, we continue adding the child nodes to the block until it is full, then we repeat the process again for the remaining nodes. In general, each block is a subtree of size B, starting from the root node.

We know that the height of the parent tree (also the length of a search path) is $\log(n)$, and the height of each subtree is $\log(B)$. Each time we perform a search, the number of I/Os that need to be performed is the number of subtrees that each path contains. which is:

$$\#I/O's = \frac{\log(n)}{\log(B)} = \log_B(n)$$

The new grouping strategy is significantly better than the previous I/O's complexity $O(\log(n/B))$.

(iii)

Our proposed solution significantly increase the spatial locality. In the original approach, we load a block to read only 1 center. Meanwhile, in our solution, at least $\log(B)$ items in a block are useful for the algorithm.

The temporal locality of the 2 approaches is the same. It is because we never reuse the block again after reading it, in both methods.

IO.II-1

Let $A = n\sqrt{n}/(MB)$, then the optimal policy performs cA I/Os.

Suppose we perform the algorithm on a machine with $M' = M/2$ memory. Then the optimal policy definitely performs $n\sqrt{n}/(\frac{M}{2}B) = 2cA$ I/Os.

According to theorem 5.3 in the course notes, the LRU will perform the following number of I/Os when it is given a memory of M, while the optimal is given a memory $M' = M/2 < M$:

$$LRU(M) \leq \frac{M}{M - M' + B} MIN(M')$$

$$= \frac{M}{\frac{M}{2} + B} MIN(M')$$

$$= \frac{2M}{M + 2B} 2c \frac{n\sqrt{n}}{MB}$$

$$= 4c \frac{n\sqrt{n}}{(M + 2B)B}$$

$$\leq 4c \frac{n\sqrt{n}}{MB} = 4cA$$

Let $c' = 4c$, then the LRU need at most $c'A$ I/Os.

IO.II-2

(i)

Let n be the total number of items to be sorted, and $k = M/B - 1$ be the number of subarrays.

The complexity of the EM-MergeSort is the cost of the recurrence plus the cost of merging k subarrays of size n/k into an array of size n.

For each element, we have to perform $k - 1$ comparisons to put it to the proper position in the final array. We have n element in total, so the cost of merging is $O(n(k-1)) = O(nk)$.

We can write the recurrence formula of the EM-MergeSort as follows:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/k) + O(nk) & \text{otherwise} \end{cases}$$

The base case is when $n = 1$, then $T(n) = 1$.

Analyzing this recurrence formula, we have:

$$\begin{aligned} T(n) &= k^2 T(n/(k^2)) + 2O(nk) \\ &= k^3 T(n/(k^3)) + 3O(nk) \\ &= \cdots \\ &= k^\alpha T(n/(k^\alpha)) + \alpha O(nk) \end{aligned}$$

$T(n)$ reaches the base case when $n/(k^\alpha) = 1 \iff \alpha = \log_k(n)$, so the final formula is:

$$\begin{aligned} T(n) &= k^{log_k(n)}(n/(k^{log_k(n)})) + \log_k(n)O(nk) \\ &= n + \log_k(n)O(nk) \\ &= O(nk \log_k(n)) \end{aligned}$$

(ii)

We propose another merging strategy as shown in algorithm 1.

Now we analyze the complexity of the new merging strategy, and then the complexity of the EM-MergeSort using this as the main merging routine.

For each step, we reduce the number of pairs by 2. So it takes $O(\log(k))$ steps to converge to 1 array.

Algorithm 1 Merging k arrays

Require: List of subarrays $S = [A_1, A_2, \cdots, A_k]$
Ensure: Merged Array A
  if $(k \mod 2) > 0$ **then**
    $A_{last} = Ak$
    $k = k - 1$
  end if
  while $SizeOf(S) > 1$ **do**
    Construct $k/2$ pairs of subarrays $\{A_1, A_2\}, \cdots, \{A_{k-1}, A_k\}$
    S := [ ]
    for Each pair $\{A_i, A_j\}$ **do**
      Merge A_i and A_j into A_{ij}
      Add A_{ij} to S
    end for
  end while
  if A_{last} NOT NULL **then**
    Merge S[0] with A_{last}
  end if
    return S[0]

For each pair of size n/k, it takes $O(2n/k)$ comparisons to merge them into 1. In each step of the merge, we perform $O(n)$ comparisons.

In general, the complexity of the merge step is $O(n \log(k))$. So the recurrence formula of the K-way MergeSort is:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/k) + O(n \log k) & \text{otherwise} \end{cases}$$

We have:

$$\begin{aligned} T(n) &= k^2 T(n/(k^2)) + 2O(n \log(k)) \\ &= k^3 T(n/(k^3)) + 3O(n \log(k)) \\ &= \cdots \\ &= k^\alpha T(n/(k^\alpha)) + \alpha O(n \log(k)) \end{aligned}$$

$T(n)$ reaches the base case when the problem size reaches 1. It takes $\log_k n$ steps to do so, then:

$$
\begin{aligned}
T(n) &= n + \log_k(n) O(n \log(k)) \\
&= n + O(n \frac{log(n) \log(k)}{\log(k)}) \\
&= O(n \log(n))
\end{aligned}
$$

So our new merging strategy satisfies the condition.

IO.III-1

(i)

When the subarray fits into 1 block, then one of the subtree does not need any more I/Os and the other one needs one more I/O (because the next center is in a consecutive block).

The number of steps needed to reach such case is $\log(n/B)$. Then the error of the number of I/Os is $O(1)$. We have:

$$\#I/Os = \log(n/B) \pm O(1)$$

(ii)

Instead of storing blocks as in-order traversal, we will separate the tree in to small subtrees of size B and store it into a block as shown in figure 2.

We know that the depth of the tree is $\log n$ and the depth of subtree is $\log B$. Therefore IO's complexity of root-to-leaf traversal is :

$$T(n) = \frac{\log n}{\log B} = \log_B n$$

As shown in figure 2, this blocking strategy make the tree look similar to B-tree in a sense that its nodes are grouped into subtrees and each subtree is comparable to a node in B-tree that can have more than 1 element.

IO.III-2

Let P denote the sequence of operations $op_0, \cdots, op_{n-1}$ and $op_i = (i, type_i, x_i)$ where i is timestamp, $type_i \in \{Insert, Delete, Search\}$ and x_i is the value of op_i . Suppose all $op_i \in P$ are stored in external memory already. In order to report $Search$ operations in O($SORT(n)$), we have to do following steps below.

1. Sort P by x_i and i in ascending order using EM-MergeSort
2. Traverse through all $op_i \in P$

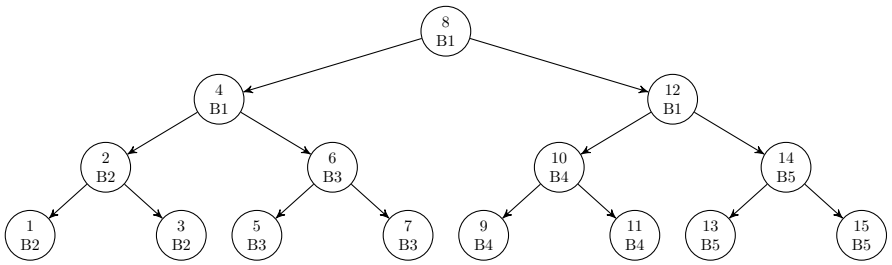

Figure 2: An illustration of subtree blocking strategy when B=3

- if $type_i \in \{Insert, Delete\}$ then $pv = op_i$
- if $type_i \in \{Search\}$ and x_i is equal to x_{pv} and $type_{pv}$ = Insert then report op_i = Found
- otherwise report op_i = Not Found

Theorem. This steps performs $O(SORT(n))$ and report $Search$ operations correctly.

Proof. We first look at the number of I/O's that we need to perform such steps. We know that $EM-MergeSort$ takes $O(SORT(n))$ I/O's and traversing sorted array takes $O(n/B)$ I/O's. The total number of I/O's complexity is still $O(SORT(n))$. Secondly, this steps will always report correct result since we sort P by x_i and i in ascending order that means it will report as Found only when there is a $Insert$ operation coming before $Search$ operation. □

IO.III-3

Theorem. The algorithm performs $O(SORT(|V| + |E|))$ and returns valid solution.

Proof. Let denote N_{in} a set of in-neighbor of $v_i \in V$. We observe that each iteration the algorithm performs

$$2O(SORT(1 + |N_{in}(v_i)|))$$

One from $Extract - Min$ operation and the other from $Insertions$.
Thus, the total I/O's complexity is :

$$\begin{aligned} T(G(V,E)) &= \sum_{i=0}^{n-1} O(1 + |N_{in}|)(v_i) \\ &= O(SORT(|V| + |E|)) \end{aligned}$$

To prove that the algorithm returns valid solution. We first argue that the

Algorithm 2 Coloring Undirected Graph

Input: $G(V, E)$
Output: Color of $v_i \in V$
Operation:
Convert G into a DAG graph G^* by directing every edge (v_i, v_j) where $i < j$
Initialise an array F[0..$n-1$], a min-priority queue Q
Initialise a set of colour C[0..d_{max}]
for $i := 0$ to $n-1$ **do**
Performs $Extract - Min$ operations on Q until getting a pair whose priority $i' > i$
Insert the extra extracted pair back to Q
$f(v_i)$:= Choose one colour from C \ { extracted pairs }
F[i] := $f(v_i)$
for each j in $V[i].OutNeighbors$ **do**
Insert a pair ($f(v_i)$, j) into Q
end for
end for
return F

algorithm does not use more than $d_{max} + 1$ colours . Secondly, there are no 2 adjacent nodes having same colour because each time it picks a colour the list of colours is subtracted by colours used for its in-neighbors.

□