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 are several items left which total size is less than B. 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 OPT 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:

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

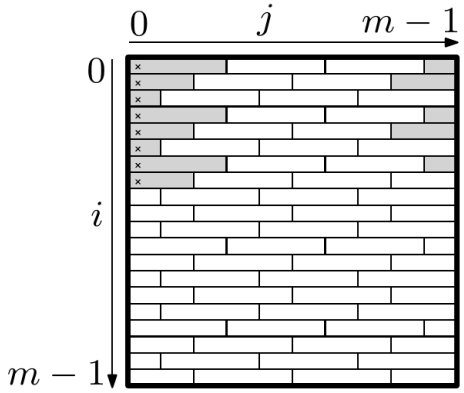

Figure 1: Blocks loaded into an 2-dimensional array

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 each column of the array can store at least 2 memory-size regions, plus several individual items.

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 $M - size$ part has 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 everytime 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 IO's of the problem and T_x, T_y and T_z denote IO's of matrix 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 IO's we need for computing a cell is :

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

Therefore, the total IO's that we need to compute 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 IO'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)

In order to compute sub-problem, all variables that we have in sub-problem should fit into main memory.

Let M_x, M_y and M_z denote memory space that is required by X, Y and Z when computing a sub-problem

For each sub problem, we need

$$M_x = t(2t + 2(B-1))$$
$$M_y = 2t(t + 2(B-1))$$
$$M_z = t(t + 2(B-1))$$

Let M denote the total memory we have. Then, we find IO's recursive base case which happens when all variables in sub-problem can fit into main memory.

$$M_x + M_y + M_z \leq M$$
$$5n^2 + 6(B-1)n \leq M$$

Thus, we can formulate recurrence IO's complexity function of this algorithm.

$$T(t) = \begin{cases} \frac{5t^2+6(B-1)t}{B}, & \text{if } 5t^2 + 6(B-1)t \leq M \\ 4T(\frac{t}{2}), & \text{otherwise} \end{cases}$$

Theorem. the IO's complexity of the algorithm is $O(t^2/B)$

Proof. We will proof this theorem by induction.

Assume, t' is the value that satisfies base case condition,

$$5t'^2 + 6(B-1)t' \leq M$$

Hence, for some t' the I/O's complexity is :

$$T(t') \leq O(t'^2/B)$$

Then, we can derive the I/O's complexity of the next case:

$$\begin{aligned} T(2t') &\leq 4T(t') \\ &\leq 4O(t'^2/B) \\ &\leq O(4t'^2/B) \\ &\leq O((2t')^2/B) \end{aligned}$$

Therefore, the IO's complexity of the algorithm is $O(t^2/B)$ as claimed. □

(iv)

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

To prove the claim, as we can see from (i), the algorithm does not use blocks of Y effectively. When it fetches a block of Y, it uses only one variable and move on. In contrast, the algorithm from (iii) can use all variables in a block.

Because the variables are kept in main memory and used during the execution of sub problem.

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

To prove the claim, assume B_0 is the first block of Y. In (i), the algorithm will need the block again when the algorithm finishes m iterations while the algorithm from (iii) will need the block after $m/2$ iterations which is 2 time shorter than (i).

IO.1-3

(i)

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 everytime 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 agafor 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:

$$\#I/Os = \frac{log(n)}{log(B)} = log_B(n)$$

So we have to perform $O(log_B(n))$ I/Os using the new grouping strategy, which is significantly better than the previous I/O 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:

$$T(n) = kT(n/k) + O(nk)$$

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

Analyzing this recurrence formula, we have:

$$T(n) = k^2T(n/(k^2)) + 2.O(nk) = k^3T(n/(k^3)) + 3.O(nk) = \cdots$$

$$T(n) = k^\alpha T(n/(k^\alpha)) + \alpha O(nk)$$

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

$$T(n) = k^{log_k(n)}(n/(k^{log_k(n)}))+log_k(n).O(nk) = n+log_k(n).O(nk) = O(nklog_k(n))$$

(ii)

We propose another merging strategy as shown in algorithm **??**.

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.

The merge step is a reverse operation of a binary search tree. For each step, we reduce the number of pairs by 2. So it takes $O(log(k))$ steps to converge to 1 array.

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.

Algorithm 1 Finding minimum row square cover

Require: List of subarrays $S = A_1, A_2, \cdots, A_k$
Ensure: Merged Array A
  if $k/2 > 0$ **then**
    $A_{last} = Ak$
    $k = k - 1$
  end if
  while $SizeOf(S) > 1$ **do**
    Construct $k/2$ pairs of subarrays $\{A_1, A_2\}, \cdots, \{A_{k-1}, A_k\}$
    Clear(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]

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

$$T(n) = kT(n/k) + O(nlog(k))$$

We have:

$$T(n) = k^2T(n/(k^2)) + 2.O(nlog(k)) = k^3T(n/(k^3)) + 3.O(nlog(k)) = \cdots =$$

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

$$T(n) = n + log_k(n)O(nlog(k)) = n + O(n\frac{log(n)log(k)}{log(k)}) = O(nlog(n))$$

So our new merging strategy satisfies the condition.

IO.III-1

(i)

(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 the figure.

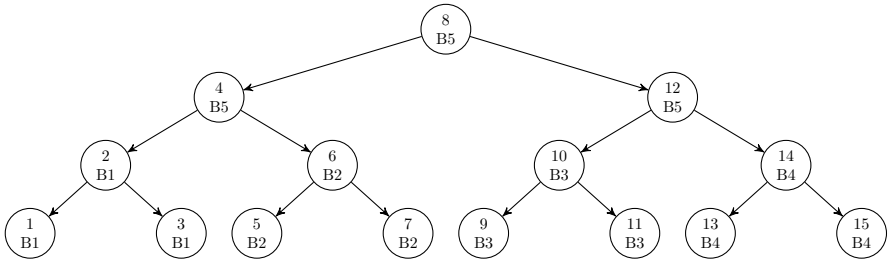

Figure 2: An illustration of sub-tree blocking strategy when B=3

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

$$\frac{\log n}{\log B} = \log_B n$$

IO.III-3

Algorithm 2 Coloring Undirected Graph

Input: $G(V, E)$
Output: Color of $v_i \in V$
Operation:
Convert G into a directed graph G^*
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 part ($f(v_i)$, j) into Q
end for
end for
return F

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

Proof. 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 :

$$\sum_{i=0}^{n-1} O(1 + |N_{in}(v_i) = O(SORT(|V| + |E|))$$

To prove that the algorithm returns valid solution. We first argue that the algorithm does not use more than $d_max + 1$ colours . Secondly, there are not 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.

□