# Problem 1

**(a)** Firstly, the indicator $f_{ij}^k$ is confined as an integer of either 0 or 1. Then, on each vertex, there is flow conservation law, forcing $\sum_j f_{ij}^k - f_{ji}^k = 0$, if $i$ is not the source $s_k$ nor the sink $t_k$. Next, when $i$ is source or sink, we make the previous summation 1 or $-1$, indicating an outgoing or incoming path with flow value 1. Lastly, we need to ensure that on each edge, there is only a single flow passing through, hence $\sum_k f_{ij}^k - f_{ji}^k \leq 1$.

In all, we can write the following integer linear program

$$f_{ij}^k \in \{0,1\}, \quad \forall i,j,k$$

$$\begin{aligned}
\sum_j f_{ij}^k - f_{ji}^k &= 1, && \text{if } i = s_k \text{ for some } k,\\
\sum_j f_{ij}^k - f_{ji}^k &= -1, && \text{if } i = t_k \text{ for some } k,\\
\sum_j f_{ij}^k - f_{ji}^k &= 0, && \text{otherwise.}
\end{aligned}$$

$$\sum_k f_{ij}^k + f_{ji}^k \leq 1, \quad \forall i,j,k$$

**(b)** If we relax the ILP, then the first constraint will be modified to

$$f_{ij}^k \in [0,1], \quad \forall i,j,k$$

This means each flow $k$ can take a fractional value on each path from $s_k$ to $t_k$. The total value of the flow is still 1 as we do not modify the second and third constraints.

The path can be read out form the solution in a water-filling manner. Specifically, we can find[1] an edge $e(i,j)$ which has some positive $f_{ij}^k$ through the edge. Then we find the smallest flow $\min_k f_{ij}^k$ on that edge. Next, traverse the graph until the source $s_k$ and the sink $t_k$, and along the way we subtract the flow corresponding to the flow $k$'s portion from the edge. Repeat the process until no edge has positive flow left.

To show a polynomial running time, we provide a (possibly quite loose) upper bound. For each edge, search for the flow with minimum $f_{ij}^k$ takes $O(n)$ time, because a flow can be at most spited into path connected to all other nodes. Traversing the

[1] We could create a linked list storing edges that has positive flow through, for constant time lookup (return head) and delete (as we traverse the graph and subtract flow on edge to empty, we can go directly to the linked list and remove an item, if the edge to linked list pointer is also saved at each edge).

graph to the source and sink would take $O(m)$ time, where $m$ is the number of edges. In total, there can be at most $n$ chooses $2 = O(n^2)$ source and each split $n$ flows, hence a running time of $O(n^3(m+n))$, which is a polynomial time for read out the solution.

**(c)** Notice that the sum of the path values outgoing from the source is 1. We can have a independent indicator variable for each of the path. Then we assign the indicator variable to 1 based on the probability translated from the solution of the relaxation of the problem. Specifically, for an edge, we let $X_i$ be the indicator variable that source-sink park $i$ uses the full capacity on this edge.

Now consider the capacity constraint, we have

$$\mathbb{E}\left[\sum_i X_i\right] = \sum_i p_i \leq 1$$

From Chernoff bound we will have

$$\Pr\left[\sum X_i > O(log\, n)\right] < 1/n^{10}$$

This means on any edge, the probability for it to carry more than $O(logn)$ paths is less than $1/n^{10}$. There can be $O(n^2)$ edges in the graph, from union bound we know, the total probability for any edge carrying more than $O(log\, n)$ paths is bounded by $n^2 \times 1/n^{10} = 1/n^8$. Notice that this is is a small number for large $n$ and it will take constant trials (as the probability otherwise is expoentially decayed) and each trial polynomial time as in (b) to get a solution such that every edge carries O(log n) paths.

# Problem 2

We assume the cache-oblivious scan can be finished in $O(N/B)$ time.

**(a)** Suppose the content in the memory is the pointer value to the next element. Then We create a copy of the list and sort the copy using cache-oblivious sorting, which takes $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$. Next, we scan the two lists together. Notice that during this list we can use the sorted copy to mark the *predecessor* of each element, as the element in the sorted copy points to the current element in the original list. We then do another scan, storing the successor to the element's predecessor, which at the end stores the successor's successor for each element.

The operations in this process involve a sorting on successor's pointer value and 3 times of scanning, therefore the total memory transfer is $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$.

**(b)** Like in (a) we have the original list and a *copy* of the list sorted by the successor pointer value. Then as we scan through two lists, we flip coins and associate the coin flip result with each element. We label the node *selected* if the element itself has coin flip head and its successor has coin flip tail[2]. The rest node is labeled *unselected*.

This ensures the property that no two selected nodes are adjacent in the list because the adjacent nodes would not *both* have its coin head and successor tail, as one node is the other's successor.

The probability for this pair coin flip having a coin head and a coin tail is $1/4$. Since coin flips are independent event, we know at the end we would select $N/4$ nodes. Therefore the $\alpha$ we choose is $1/4$.

The operations in this process involve one sorting and constant times of scanning, therefore the total memory transfer is $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$.

**(c)** We can scan through the list and create a list of the nodes that is unselected (this is to exclude the selected nodes). Then we need to connect these nodes in the new list. This can be done by utilizing (a), the field of successor's successor. Specifically, when doing the scan, we also scan (a)'s sorted list based on successor's pointer value. On scanning the sorted list, if the node is *selected* (current node can't be selected because of (b) no adjacent nodes can be both selected), we update the successor on the unselected node in the new list to be the current node's successor's successor.

The operations in this process involve one sorting in (a) and constant times of scanning, therefore the total memory transfer is $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$.

[2]Notice that the other list is a copy, when the successor has coin flipped associated already, we don't do the coin flip again when scan to its own location in the original list and vice versa.

**(d)** Suppose now we already solved the list-ranking problem on the unselected nodes. Together with (a), (c) each element on the list is associated with the pointer to the successor (before selection and after selection), successor's successor (before selection and after selection), label of selected or unselected, and the rank if the element is unselected.

Now we can sort the unselected list (it is created in (c)) based on its rank. Then we scan the sorted list, in ascending order. Each time, we modify the rank of current element to the rank of its successor +1 (This is needed because we add back the selected element). Additionally, if its successor *before* selection is *selected*, it means the current successor is redirected to the unselected element. In this case, we (1) modify the successor pointing back to the selected element[3] (2) let the rank of the selected element to be the rank of current successor's successor (or the selected element's successor)'s rank + 1 (3) increment current element's rank by 1.

The operations in this process involve one sorting on the rank, and constant times of scanning, therefore the total memory transfer is $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$.

**(e)** From (d) we know given the unselected list we can solve the entire problem in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$. For the unselected list, we can recursively go down the hill to solve the problems within. Each time unroll back it takes the memory transfer of sorting. Notice the $\alpha = 3/4 < 1$ factor in (b), the problem can be decomposed into a sorting merging part and a recursive part of a problem with 3/4 the size. Therefore, the total memory transfers can be calculated as

$$
\begin{aligned}
&O(\frac{N}{B}\log_{M/B}\frac{N}{B}) + O(\left(\frac{3}{4}\right)\frac{3N/4}{B}\log_{M/B}\frac{3N/4}{B}) + O(\left(\frac{3}{4}\right)^2\frac{(3/4)^2N}{B}\log_{M/B}\frac{(3/4)^2N}{B}) + \dots \\
&\le O(\frac{N}{B}\log_{M/B}\frac{N}{B}) + O(\left(\frac{3}{4}\right)\frac{N}{B}\log_{M/B}\frac{N}{B}) + O(\left(\frac{3}{4}\right)^2\frac{N}{B}\log_{M/B}\frac{N}{B}) + \dots \\
&= O\left((1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + ...)\frac{N}{B}\log_{M/B}\frac{N}{B}\right) \\
&= O\left(4 \times \frac{N}{B}\log_{M/B}\frac{N}{B}\right) \\
&= O(\frac{N}{B}\log_{M/B}\frac{N}{B}).
\end{aligned}
$$

[3]The selected element still has pointer to its successor, which is current successor's successor

# Problem 3

For LRU, notice that the content of the cache with smaller size $k$ is a subset of the cache with larger size $k'$. This is because only the least recently used page is deported, we can sort (in our mind) all the pages in the cache and the most recently used are kept in the stack. After sorting, the most recently used ones with size $k$ will the same for two caches with different size and these are the ones in both caches. Therefore, cache with size $k$ is a subset of cache with size $k'$.

This being said, when page fault happens, it can never be the case that a content appears in cache with size $k$ but not in the cache with larger size $k'$, as the prior one is a subset of the lateral one. Therefore, LRU will never incur Beladys anomaly. □

For FIFO, we provide a concrete example. Consider a system with cache size 3 and another with cache size 4. The page requests come in sequence of

$$1,\ 2,\ 3,\ 4,\ 1,\ 2,\ 5,\ 1,\ 2,\ 3,\ 4,\ 5$$

Then for cache size 3, it will have 9 page fault as in

$$\overset{\downarrow}{1}, \times, \times \to \overset{\downarrow}{2}, 1, \times \to \overset{\downarrow}{3}, 2, 1 \to \overset{\downarrow}{4}, 3, 2 \to \overset{\downarrow}{1}, 4, 3 \to \overset{\downarrow}{2}, 1, 4$$
$$\to \overset{\downarrow}{5}, 2, 1 \to 5, 2, 1 \to 5, 2, 1 \to 5, 2, 1 \to \overset{\downarrow}{3}, 5, 2 \to \overset{\downarrow}{4}, 3, 5 \to 4, 3, 5$$

But for cache size 4, it will have 10 page fault as in

$$\overset{\downarrow}{1}, \times, \times, \times \to \overset{\downarrow}{2}, 1, \times, \times \to \overset{\downarrow}{3}, 2, 1, \times \to \overset{\downarrow}{4}, 3, 2, 1 \to 4, 3, 2, 1 \to 4, 3, 2, 1$$
$$\to 4, 3, 2, 1 \to \overset{\downarrow}{5}, 4, 3, 2 \to \overset{\downarrow}{1}, 5, 4, 3 \to \overset{\downarrow}{2}, 1, 5, 4 \to \overset{\downarrow}{3}, 2, 1, 5 \to \overset{\downarrow}{4}, 3, 2, 1 \to \overset{\downarrow}{5}, 4, 3, 2$$

Therefore it shows an example of Beladys anomaly for FIFO. □

# Problem 4

**(a)** The potential is the same as the one being used in the line case

$$\Phi = kM_{min} + \sum_{i<j} d^{DC}(s_i, s_j)$$

Now when OPT moves server by distance $d$, it increases the matching $M_{min}$ by at most $d$. Therefore, the potential change is at most $kd$, since $d^{DC}$ is not affected here in OPT moving.

Next, to show $k$-competitive, we want to show when DC moves, it decreases the potential by $d$. Notice that one of the servers in the neighbors is matched with OPT at the request (similarly in the line case, otherwise there can be crossing and this is not needed). Suppose this server moves distance $d'$. Because some servers move might be 'blocked', we assume $m$ servers move this distance. Now the optimal matching grows by at most

$$-d' + (m-1)d' = (m-2)d'$$

For the servers that move, the distance among them decrease by[4]

$$m(m-1)d'$$

And for the distance to *each* of the rest of the servers, the distance changes is similar to the analysis for optimal matching, as $m-1$ servers move closer and one moves further away.

Then the total change in the potential becomes

$$\Delta\Phi = k(m-2)d' - m(m-1)d' - (k-m)(m-2)d' = -md'$$

This means when DC moves *in total* $d$, each server moves $d' = d/n$ and the decrease becomes $-n \times d/n = -d$ if all servers move. In other words, every $d$ move in OPT would allow at most $kd$ move in DC, therefore DC is k-competitive.

**(b)** We set the tree to be a single root connecting to $n$ leaves. Each leaf stores a page and each of the $k$ servers represents a memory entry. The cost on the edge is $1/2$ so that a memory swap would cost 1 (traversing 2 edges). The memory swap happens when a server moves to a tree leaf, and it swap out the page it holds and swap in the page in the leaf.

We also assume servers sit on the leaves initially[5], in order to get the exact cost. The optimal solution here can be used for page replacement, following the server movement on the leaves. The cost to this $k$-server problem would be the same as the cost in paging problem.

[4] This is twice $m$ chooses 2 because each server in the pair needs to be account for.

[5] More because in the marking algorithm initially all pages are marked (technically). If they are not on the leaves, the cost will still be the same in asymptotic sense.

**(c)** It corresponds to the Marking algorithm. The servers on the leaves correspond to the pages being marked. When moving around, some may be hold at the root node, in which case the page is unmarked.

The k-server algorithm moves server around and some may hold the root node, in which case the page is unmarked. In Marking algorithm, the "if all marked, unmark all" denotes the case everyone moves to the root node. When evict, a random server from the root node will take over the leaf at the page request, and it will be correspondingly marked (all others hold in the root node, remain unmarked). This becomes essentially the marking algorithm for paging.

# Problem 5

**(a)** Consider a sequence of actions (following wall-time)

$$\begin{aligned} & P_1 : \text{WRITE}(l = 1, v = 1) \to P_1 : \text{READ}(l = 1) \\ \to\ & P_2 : \text{WRITE}(l = 1, v = 2) \to P_2 : \text{READ}(l = 1) \\ \to\ & P_1 : \text{WRITE}(l = 1, v = 1) \to P_1 : \text{READ}(l = 1) \\ \to\ & P_2 : \text{WRITE}(l = 1, v = 2) \to P_2 : \text{READ}(l = 1) \\ \to\ & \dots \dots \end{aligned}$$

Using SHARE_ALL, upon each WRITE we need to invoke $update(l, v)$, and each time it costs a bus communication for the sync up. However, in the optimal setting, the above sequence has only has *local* read after every write, there is actually no need for synchronization. Since the sequence can repeat itself infinitely more times, the SHARE_ALL cost is unbounded compared with the optimal.

**(b)** Consider now for $n$ processor we have the following sequence of actions

$$\begin{aligned} & P_1 : \text{WRITE}(l = 1, v = 1) \\ \to\ & P_2 : \text{READ}(l = 1) \\ \to\ & P_3 : \text{READ}(l = 1) \\ \to\ & \dots \\ \to\ & P_n : \text{READ}(l = 1) \\ \to\ & P_1 : \text{WRITE}(l = 1, v = 2) \\ \to\ & P_2 : \text{READ}(l = 1) \\ \to\ & P_3 : \text{READ}(l = 1) \\ \to\ & \dots \\ \to\ & P_n : \text{READ}(l = 1) \\ \to\ & \dots \dots \end{aligned}$$

Using SHARE_NONE, the copy of the content at $l = 1$ will only be valid at the current read, because it *invalid* after request the entry each time. This requires two broadcast for each read operation (request and invalid). However, the optimal action can be just write once, broadcast to everyone's local entry and let each processor read locally. Since the sequence can repeat infinitely long, the SHARE_NONE cost is unbounded compared with the optimal.

**(c)**

**1.** Consider the WRITE case first.

- If the memory location starts as $(S, S)$, then $A$ would use *invalidation* to make the memory location unique, namely $(U, \cdot)$. If OPT chooses to make its shared state unique, then the state becomes $(U, U)$. Here $C_A(t)$ would take $1 + n$ operations for write local location and invalidation. OPT can just update the local copy and take 1 operation. But the potential grows by $n$ as this memory location in each processor now becomes $(U, U)$, which implies $C_A(t) = 1 + n \leq 2 \times 1 + n = 2C_{OPT}(t) + \Delta\Phi$. On the other hand, if OPT decides to propagate the copy, then it becomes $(U, S)$ for each location. OPT takes $1 + n$ work, but potential drop by $n$ as there is $n$ more $(U, S)$ appearing, therefore $C_A(t) = 1 + n \leq 2 \times (1 + n) - n = 2C_{OPT}(t) + \Delta\Phi$.

- If the memory location starts as $(S, U)$, then $A$ would use *invalidation* to make the memory location unique, namely $(U, \cdot)$. If OPT chooses to make its shared state unique, then the state becomes $(U, U)$. This would be similar to the first sub-case above, where $C_A(t) = 1 + n \leq 2 \times 1 + n = 2C_{OPT}(t) + \Delta\Phi$. If OPT chooses to propagate the copy, then it becomes $(U, S)$ for each location. Again similarly, OPT takes $1 + n$ work, but potential drop by $n$ as there is $n$ more $(U, S)$ appearing, therefore $C_A(t) = 1 + n \leq 2 \times (1 + n) - n = 2C_{OPT}(t) + \Delta\Phi$.

- If the memory location starts as $(U, S)$, then $A$ would directly update the content in memory location locally resulting in $(U, \cdot)$. If OPT chooses to make its shared state unique, then the state becomes $(U, U)$. Since A takes 1 work and OPT takes 1 and potential grows $n + n$ since $(U, S)$ is replaced by $(U, U)$, we have $C_A(t) = 1 \leq 2 \times 1 + 2n = 2C_{OPT}(t) + \Delta\Phi$. If OPT chooses to propagate the copy, then it becomes $(U, S)$ for each location. Now OPT takes $1 + n$ work and the potential remains unchanged, therefore $C_A(t) = 1 \leq 2 \times (1 + n) = 2C_{OPT}(t) + \Delta\Phi$.

- if the memory location starts as $(U, U)$, then $A$ would directly update the content in memory location locally resulting in $(U, \cdot)$. If OPT chooses to make its shared state unique, then the state becomes $(U, U)$. Since A takes 1 work and OPT takes 1 and the potential remains the same, so we have $C_A(t) = 1 \leq 2 \times 1 = 2C_{OPT}(t) + \Delta\Phi$. If OPT chooses to propagate the copy, then it becomes $(U, S)$ for each location. In this case, OPT takes $1 + n$ work, but potential drop by $2n$ since $(U, U)$ is replaced by $(U, S)$, therefore $C_A(t) = 1 \leq 2 \times (1 + n) - 2n = 2C_{OPT}(t) + \Delta\Phi$.

Then we consider the READ case.

- If the memory location starts as $(U, U)$. (1) If the unique content is in the local processor that A query for, there is no need for *request*. Then it results in $(U, \cdot)$. If OPT chooses to propagate, the state becomes $(U, S)$ and it costs OPT $1+n$ operation. Therefore $C_A(t) = 1 \leq 2 \times (1+n) - 2n = 2C_{OPT}(t) + \Delta\Phi$. Else if OPT chooses to also read locally, we have $C_A(t) = 1 \leq 2 \times 1 = 2C_{OPT}(t) + \Delta\Phi$. (2) If the unique content is not in the current processor (it is invalid), A takes $1+n$ work. OPT in this case would be forced to use *request*, otherwise it can't return a correct result. The potential change in this case is $-n$ because $(U, U)$ is replaced by $(S, S)$. Therefore $C_A(t) = 1+n \leq 2 \times (1+n) - n = 2C_{OPT}(t) + \Delta\Phi$.

- If the memory location starts as $(S, U)$. (1) If the unique content is in the local processor that A query for, there is no need for *request*. Then it results in $(S, \cdot)$. If OPT chooses to propagate, the state becomes $(S, S)$ and it costs OPT $1+n$ operation. Therefore $C_A(t) = 1 \leq 2 \times (1+n) = 2C_{OPT}(t) + \Delta\Phi$. Else if OPT chooses to also read locally, we have $C_A(t) = 1 \leq 2 \times 1 = 2C_{OPT}(t) + \Delta\Phi$. (2) If the unique content is not in the current processor (it is invalid), A takes $1+n$ work. OPT in this case would be forced to use *request*, otherwise it can't return a correct result. The state becomes $(S, S)$. Therefore $C_A(t) = 1+n \leq 2 \times (1+n) = 2C_{OPT}(t) + \Delta\Phi$.

- If the memory location starts as $(U, S)$. (1) If the unique content is in the local processor that A query for, there is no need for *request*. Then it remains the state in $(U, S)$. If OPT chooses to propagate, it will cost OPT $1+n$ operation. Therefore $C_A(t) = 1 \leq 2 \times (1+n) = 2C_{OPT}(t) + \Delta\Phi$. Else if OPT chooses to also read locally, we have $C_A(t) = 1 \leq 2 \times 1 = 2C_{OPT}(t) + \Delta\Phi$. (2) If the unique content is not in the current processor (it is invalid), A takes $1+n$ work. OPT in this case would be forced to use *request*, otherwise it can't return a correct result. The potential change in this case is $n$ because $(U, S)$ is replaced by $(S, S)$. Therefore $C_A(t) = 1+n \leq 2 \times (1+n) + n = 2C_{OPT}(t) + \Delta\Phi$.

- If the memory location starts as $(S, S)$. (1) If the unique content is in the local processor that A query for, there is no need for *request*. Then it remains the state in $(S, S)$. If OPT chooses to propagate, it will cost OPT $1+n$ operation. Therefore $C_A(t) = 1 \leq 2 \times (1+n) = 2C_{OPT}(t) + \Delta\Phi$. Else if OPT chooses to also read locally, we have $C_A(t) = 1 \leq 2 \times 1 = 2C_{OPT}(t) + \Delta\Phi$. (2) If the unique content is not in the current processor (it is invalid), A takes $1+n$ work. OPT in this case would be forced to use *request*, otherwise it can't return a correct result. Therefore $C_A(t) = 1+n \leq 2 \times (1+n) = 2C_{OPT}(t) + \Delta\Phi$.

Since in each of the WRITE/READ case and in each of the state transition $C_A(t) \leq 2C_{OPT}(t) + \Delta\Phi$ is hold, we know the inequality holds in general.

**2.** Summing over all time we have $C_{A/OPT}(t)$ aggregates to $C_{A/OPT}$ and the potential difference telescopes,

$$\sum_{t=0}^{T} C_A(t) \leq \sum_{t=0}^{T} \Big[ 2\, C_{OPT}(t) + \Phi_t \Big]$$
$$C_A \leq 2\, C_{OPT} + (\Phi_T - \Phi_{T-1}) + (\Phi_{T-1} - \Phi_{T-2}) + \cdots + (\Phi_1 - \Phi_0)$$
$$C_A \leq 2\, C_{OPT} + \Phi_T - \Phi_0.$$

This means the cost of A is bounded by twice the cost of OPT with a potential difference. Notice from the definition of the potential that $\Phi$ is upper bounded by total number of memory entries $\times$ total number of processors and lower bounded by 0, therefore this potential difference term can be viewed as a constant in asymptotic sense when we compare the cost. Hence, SHARE_SOME algorithm is 2-competitive.

## (d)

**1.** We can follow the conditions. For the first two cases, READ will results in a request from the other processor. Then the state will become shared. At this moment, the adversary will send a WRITE to the processor that just called READ. Notice that algorithm A is deterministic, if after the write to the local, it decides to propagate the content to the other processor, adversary will then ask another WRITE command. Otherwise, the adversary will call a READ from the other processor. On the WRITE command on the shared entry, the algorithm needs to also call either ab *invalidate* or *update*, as the cache cannot just communicate nothing. Therefore, for each step, there is at least one extra communication over the bus. Hence $T$ steps would cost $T$ bus communication[6].

**2.** Now for this *fixed* sequence, *SEEK* is possible to look ahead and decide to take a different action in a priori. Specifically, we can group the sequence of operations into pairs, and SEEK decides to take the specific action in advance. For READ-READ, at the first READ, we can call a request, so that the second one only needs to do a local call. For WRITE-READ, we can do an update at the WRITE, so that the READ is local. For READ-WRITE, we can do request-invalidate to save the communication in the WRITE. For WRITE-WRITE, we do a invalidate in the first WRITE. This way, SEEK only has 1 bus communication cost for each pair, while the algorithm A needs to do bus communication each time. Therefore algorithm A can not do better than 2 competitive.

[6]We assume the local $O(T)$ computation is cheap.