Greedy algorithms 2

Abhiram Ranade

February 2, 2016

Minimizing maximum lateness

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n
Output: Starting times s[1..n] and finishing times f[1..n] s.t.

- At any time only one job is being processed.
- Maximum lateness l[i] = f[i] - d[i] of any job i is minimized
  l[i] could be negative; does not hurt.

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n
Output: Starting times s[1..n] and finishing times f[1..n] s.t.

- At any time only one job is being processed.
- Maximum lateness l[i] = f[i] - d[i] of any job i is minimized
  l[i] could be negative; does not hurt.

Is there a greedy algorithm for this?

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n
Output: Starting times s[1..n] and finishing times f[1..n] s.t.

- At any time only one job is being processed.
- Maximum lateness l[i] = f[i] - d[i] of any job i is minimized
  l[i] could be negative; does not hurt.

Is there a greedy algorithm for this?
Let your imagination loose!

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n
Output: Starting times s[1..n] and finishing times f[1..n] s.t.

- At any time only one job is being processed.
- Maximum lateness l[i] = f[i] - d[i] of any job i is minimized
  l[i] could be negative; does not hurt.

Is there a greedy algorithm for this?
Let your imagination loose!

- A job with a small deadline is more likely to get delayed.

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n
Output: Starting times s[1..n] and finishing times f[1..n] s.t.

- At any time only one job is being processed.
- Maximum lateness l[i] = f[i] - d[i] of any job i is minimized
  l[i] could be negative; does not hurt.

Is there a greedy algorithm for this?
Let your imagination loose!

- A job with a small deadline is more likely to get delayed.
  greedy choice = job with min d[i]?

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n
Output: Starting times s[1..n] and finishing times f[1..n] s.t.

- At any time only one job is being processed.
- Maximum lateness l[i] = f[i] - d[i] of any job i is minimized
  l[i] could be negative; does not hurt.

Is there a greedy algorithm for this?
Let your imagination loose!

- A job with a small deadline is more likely to get delayed.
  greedy choice = job with min d[i]?
- A job with a large processing time is likely to get delayed.

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n
Output: Starting times s[1..n] and finishing times f[1..n] s.t.

- At any time only one job is being processed.
- Maximum lateness l[i] = f[i] - d[i] of any job i is minimized
  l[i] could be negative; does not hurt.

Is there a greedy algorithm for this?
Let your imagination loose!

- A job with a small deadline is more likely to get delayed.
  greedy choice = job with min d[i]?
- A job with a large processing time is likely to get delayed.
  greedy choice = job with max p[i]?

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n
Output: Starting times s[1..n] and finishing times f[1..n] s.t.

- At any time only one job is being processed.
- Maximum lateness l[i] = f[i] - d[i] of any job i is minimized
  l[i] could be negative; does not hurt.

Is there a greedy algorithm for this?
Let your imagination loose!

- A job with a small deadline is more likely to get delayed.
  greedy choice = job with min d[i]?
- A job with a large processing time is likely to get delayed.
  greedy choice = job with max p[i]?
- Or a bit of both..

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n
Output: Starting times s[1..n] and finishing times f[1..n] s.t.

- At any time only one job is being processed.
- Maximum lateness l[i] = f[i] - d[i] of any job i is minimized
  l[i] could be negative; does not hurt.

Is there a greedy algorithm for this?
Let your imagination loose!

- A job with a small deadline is more likely to get delayed.
  greedy choice = job with min d[i]?
- A job with a large processing time is likely to get delayed.
  greedy choice = job with max p[i]?
- Or a bit of both..
  greedy choice = job with min d[i] - p[i]?

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n
Output: Starting times s[1..n] and finishing times f[1..n] s.t.

- At any time only one job is being processed.
- Maximum lateness l[i] = f[i] - d[i] of any job i is minimized
  l[i] could be negative; does not hurt.

Is there a greedy algorithm for this?
Let your imagination loose!

- A job with a small deadline is more likely to get delayed.
  greedy choice = job with min d[i]?
- A job with a large processing time is likely to get delayed.
  greedy choice = job with max p[i]?
- Or a bit of both..
  greedy choice = job with min d[i] - p[i]?

Which do we pick?

Minimizing maximum lateness

Input: p[1..n], d[1..n] processing times and deadlines of jobs 1..n
Output: Starting times s[1..n] and finishing times f[1..n] s.t.

- At any time only one job is being processed.
- Maximum lateness l[i] = f[i] - d[i] of any job i is minimized
  l[i] could be negative; does not hurt.

Is there a greedy algorithm for this?
Let your imagination loose!

- A job with a small deadline is more likely to get delayed.
  greedy choice = job with min d[i]?
- A job with a large processing time is likely to get delayed.
  greedy choice = job with max p[i]?
- Or a bit of both..
  greedy choice = job with min d[i] - p[i]?

Which do we pick? Try to prove the correctness of each. Pick the one for which you can get a proof!

Least deadline job must be scheduled first

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.
Proof: Suppose job j is scheduled just before job i in opt.

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.
Proof: Suppose job j is scheduled just before job i in opt.
Suppose we exchange j, i.

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.
Proof: Suppose job j is scheduled just before job i in opt.
Suppose we exchange j, i.
Lateness of i has reduced.

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.
Proof: Suppose job j is scheduled just before job i in opt.
Suppose we exchange j, i.
Lateness of i has reduced.
New lateness of j is $f[i] - d[j]$

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.

Proof: Suppose job j is scheduled just before job i in opt.

Suppose we exchange j, i.

Lateness of i has reduced.

New lateness of j is $f[i] - d[j] \leq f[i] - d[i]$

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.
Proof: Suppose job j is scheduled just before job i in opt.
Suppose we exchange j, i.
Lateness of i has reduced.
New lateness of j is $f[i] - d[j] \leq f[i] - d[i]$
$=$ Old lateness of i.

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.
Proof: Suppose job j is scheduled just before job i in opt.
Suppose we exchange j, i.
Lateness of i has reduced.
New lateness of j is $f[i] - d[j] \leq f[i] - d[i]$
$=$ Old lateness of i.
Lateness of other jobs does not change.

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.
Proof: Suppose job j is scheduled just before job i in opt.
Suppose we exchange j, i.
Lateness of i has reduced.
New lateness of j is $f[i] - d[j] \leq f[i] - d[i]$
$=$ Old lateness of i.
Lateness of other jobs does not change.
Thus max lateness cannot increase.

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.
Proof: Suppose job j is scheduled just before job i in opt.
Suppose we exchange j, i.
Lateness of i has reduced.
New lateness of j is $f[i] - d[j] \leq f[i] - d[i]$
$=$ Old lateness of i.
Lateness of other jobs does not change.
Thus max lateness cannot increase.

We can repeat this step until i moves to the beginning.

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.
Proof: Suppose job j is scheduled just before job i in opt.
Suppose we exchange j, i.
Lateness of i has reduced.
New lateness of j is $f[i] - d[j] \leq f[i] - d[i]$
$=$ Old lateness of i.
Lateness of other jobs does not change.
Thus max lateness cannot increase.

We can repeat this step until i moves to the beginning.

Greedy choice: Schedule job with least deadline.

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.

Proof: Suppose job j is scheduled just before job i in opt.
Suppose we exchange j, i.
Lateness of i has reduced.
New lateness of j is $f[i] - d[j] \leq f[i] - d[i]$
$=$ Old lateness of i.
Lateness of other jobs does not change.
Thus max lateness cannot increase.

We can repeat this step until i moves to the beginning.

Greedy choice: Schedule job with least deadline.

Optimal substructure: Obvious.

Least deadline job must be scheduled first

Thm: There exists an optimal schedule in which job i having minimum deadline is scheduled first.
Proof: Suppose job j is scheduled just before job i in opt.
Suppose we exchange j, i.
Lateness of i has reduced.
New lateness of j is $f[i] - d[j] \leq f[i] - d[i]$
$=$ Old lateness of i.
Lateness of other jobs does not change.
Thus max lateness cannot increase.

We can repeat this step until i moves to the beginning.

Greedy choice: Schedule job with least deadline.

Optimal substructure: Obvious.

Algorithm: Schedule jobs in order of increasing deadline.

Offline Caching

Offline Caching

Configuration: Computer with large main memory, and a *cache* with that can store C items.

Offline Caching

Configuration: Computer with large main memory, and a *cache* with that can store C items.

- CPU can access an item only if it is in cache.

Offline Caching

Configuration: Computer with large main memory, and a *cache* with that can store C items.

- CPU can access an item only if it is in cache.
- If requested item is in cache, it is called a *cache hit*.

Offline Caching

Configuration: Computer with large main memory, and a *cache* with that can store C items.

- CPU can access an item only if it is in cache.
- If requested item is in cache, it is called a *cache hit*.
- If requested item is not in cache, it is called a *cache miss*.

Offline Caching

Configuration: Computer with large main memory, and a *cache* with that can store C items.

- CPU can access an item only if it is in cache.
- If requested item is in cache, it is called a *cache hit*.
- If requested item is not in cache, it is called a *cache miss*.
- When there is a cache miss, the item must be *fetched* from memory into the cache.

Offline Caching

Configuration: Computer with large main memory, and a *cache* with that can store C items.

- CPU can access an item only if it is in cache.
- If requested item is in cache, it is called a *cache hit*.
- If requested item is not in cache, it is called a *cache miss*.
- When there is a cache miss, the item must be *fetched* from memory into the cache.
- When a new item is to be stored in the cache, some item present must be thrown out or *evicted*.

Offline Caching

Configuration: Computer with large main memory, and a *cache* with that can store C items.

- CPU can access an item only if it is in cache.
- If requested item is in cache, it is called a *cache hit*.
- If requested item is not in cache, it is called a *cache miss*.
- When there is a cache miss, the item must be *fetched* from memory into the cache.
- When a new item is to be stored in the cache, some item present must be thrown out or *evicted*.

Input: Sequence $r_1, \ldots, r_n$ of item requests from CPU.

Offline Caching

Configuration: Computer with large main memory, and a *cache* with that can store C items.

- CPU can access an item only if it is in cache.
- If requested item is in cache, it is called a *cache hit*.
- If requested item is not in cache, it is called a *cache miss*.
- When there is a cache miss, the item must be *fetched* from memory into the cache.
- When a new item is to be stored in the cache, some item present must be thrown out or *evicted*.

Input: Sequence $r_1, \ldots, r_n$ of item requests from CPU.
Output: What item to evict, if any, to accommodate each r_i, so as to minimize total number of fetches.

Offline Caching

Configuration: Computer with large main memory, and a *cache* with that can store C items.

- CPU can access an item only if it is in cache.
- If requested item is in cache, it is called a *cache hit*.
- If requested item is not in cache, it is called a *cache miss*.
- When there is a cache miss, the item must be *fetched* from memory into the cache.
- When a new item is to be stored in the cache, some item present must be thrown out or *evicted*.

Input: Sequence $r_1, \ldots, r_n$ of item requests from CPU.
Output: What item to evict, if any, to accommodate each r_i, so as to minimize total number of fetches.

Example
$C = 2$, cache initially holds items `x, y`
Request sequence: `a, b, a, b, c, b, c, a, a`

Offline Caching

Configuration: Computer with large main memory, and a *cache* with that can store C items.

- CPU can access an item only if it is in cache.
- If requested item is in cache, it is called a *cache hit*.
- If requested item is not in cache, it is called a *cache miss*.
- When there is a cache miss, the item must be *fetched* from memory into the cache.
- When a new item is to be stored in the cache, some item present must be thrown out or *evicted*.

Input: Sequence $r_1, \ldots, r_n$ of item requests from CPU.
Output: What item to evict, if any, to accommodate each r_i, so as to minimize total number of fetches.

Example
$C = 2$, cache initially holds items `x, y`

Request sequence:	`a, b, a, b, c, b, c, a, a`
`Eviction sequence:`	`x, y, -, -, a, -, -, c, -`

What item should we evict?

What item should we evict?

Greedy Intuition: An item which will be needed soon should not be evicted.

What item should we evict?

Greedy Intuition: An item which will be needed soon should not be evicted.

FF: farthest in future:

What item should we evict?

Greedy Intuition: An item which will be needed soon should not be evicted.

FF: farthest in future:

- If next requested item is in cache, evict nothing.

What item should we evict?

Greedy Intuition: An item which will be needed soon should not be evicted.

FF: farthest in future:

- If next requested item is in cache, evict nothing.
- If next requested item is not in cache, evict that item whose next access time is largest.

What item should we evict?

Greedy Intuition: An item which will be needed soon should not be evicted.

FF: farthest in future:

- If next requested item is in cache, evict nothing.
- If next requested item is not in cache, evict that item whose next access time is largest.

Assume: Cache contains useless items at the beginning.

What item should we evict?

Greedy Intuition: An item which will be needed soon should not be evicted.

FF: farthest in future:

- If next requested item is in cache, evict nothing.
- If next requested item is not in cache, evict that item whose next access time is largest.

Assume: Cache contains useless items at the beginning.
Assume: Item not accessed at all: next access time $= \infty$.

What item should we evict?

Greedy Intuition: An item which will be needed soon should not be evicted.

FF: farthest in future:

- If next requested item is in cache, evict nothing.
- If next requested item is not in cache, evict that item whose next access time is largest.

Assume: Cache contains useless items at the beginning.
Assume: Item not accessed at all: next access time $= \infty$.

Example:
Cache contains `a,b,c,d`. Future requests: p, c, b, c, d, p, a, d

What item should we evict?

Greedy Intuition: An item which will be needed soon should not be evicted.

FF: farthest in future:

- If next requested item is in cache, evict nothing.
- If next requested item is not in cache, evict that item whose next access time is largest.

Assume: Cache contains useless items at the beginning.
Assume: Item not accessed at all: next access time $= \infty$.

Example:
Cache contains `a,b,c,d`. Future requests: p, c, b, c, d, p, a, d
Decision: Evict a.

What item should we evict?

Greedy Intuition: An item which will be needed soon should not be evicted.

FF: farthest in future:

- If next requested item is in cache, evict nothing.
- If next requested item is not in cache, evict that item whose next access time is largest.

Assume: Cache contains useless items at the beginning.
Assume: Item not accessed at all: next access time $= \infty$.

Example:
Cache contains `a,b,c,d`. Future requests: p, c, b, c, d, p, a, d
Decision: Evict a.

Fetch only on miss: Item x fetched to memory in step t only if $r[t] = x$ and x is not in cache at step t.

What item should we evict?

Greedy Intuition: An item which will be needed soon should not be evicted.

FF: farthest in future:

- If next requested item is in cache, evict nothing.
- If next requested item is not in cache, evict that item whose next access time is largest.

Assume: Cache contains useless items at the beginning.
Assume: Item not accessed at all: next access time $= \infty$.

Example:
Cache contains `a,b,c,d`. Future requests: p, c, b, c, d, p, a, d
Decision: Evict a.

Fetch only on miss: Item x fetched to memory in step t only if $r[t] = x$ and x is not in cache at step t.

Can show that fetch without miss does not help.

Optimality of FF

Optimality of FF

Instance: $R = (r[1..n]), I$. Request sequence + initial cache content

Optimality of FF

Instance: $R = (r[1..n]), I$. Request sequence + initial cache content
Solution: $E[1..n]$. Eviction sequence. "$-$" denotes no eviction.

Optimality of FF

Instance: $R = (r[1..n]), I$. Request sequence + initial cache content
Solution: $E[1..n]$. Eviction sequence. "$-$" denotes no eviction.
Cost of solution: Number of evictions.

Optimality of FF

Instance: $R = (r[1..n]), I$. Request sequence + initial cache content
Solution: $E[1..n]$. Eviction sequence. "$-$" denotes no eviction.
Cost of solution: Number of evictions.

Theorem: FF is optimal, i.e. has minimum number of evictions.

Optimality of FF

Instance: $R = (r[1..n]), I$. Request sequence + initial cache content
Solution: $E[1..n]$. Eviction sequence. "$-$" denotes no eviction.
Cost of solution: Number of evictions.

Theorem: FF is optimal, i.e. has minimum number of evictions.

"Greedy choice" Lemma: There exists an optimal solution in which FF is used in step 1.

Optimality of FF

Instance: $R = (r[1..n]), I$. Request sequence + initial cache content
Solution: $E[1..n]$. Eviction sequence. "$-$" denotes no eviction.
Cost of solution: Number of evictions.

Theorem: FF is optimal, i.e. has minimum number of evictions.

"Greedy choice" Lemma: There exists an optimal solution in which FF is used in step 1.

To be proved soon.

Optimality of FF

Instance: $R = (r[1..n]), I$. Request sequence + initial cache content
Solution: $E[1..n]$. Eviction sequence. "$-$" denotes no eviction.
Cost of solution: Number of evictions.

Theorem: FF is optimal, i.e. has minimum number of evictions.

"Greedy choice" Lemma: There exists an optimal solution in which FF is used in step 1.

To be proved soon.

"Optimal substructure" Lemma: FF can be used for the remaining steps.

Optimality of FF

Instance: $R = (r[1..n]), I$. Request sequence + initial cache content
Solution: $E[1..n]$. Eviction sequence. "$-$" denotes no eviction.
Cost of solution: Number of evictions.

Theorem: FF is optimal, i.e. has minimum number of evictions.

"Greedy choice" Lemma: There exists an optimal solution in which FF is used in step 1.

To be proved soon.

"Optimal substructure" Lemma: FF can be used for the remaining steps.

Obvious.

Optimality of FF

Instance: $R = (r[1..n]), I$. Request sequence + initial cache content
Solution: $E[1..n]$. Eviction sequence. "$-$" denotes no eviction.
Cost of solution: Number of evictions.

Theorem: FF is optimal, i.e. has minimum number of evictions.

"Greedy choice" Lemma: There exists an optimal solution in which FF is used in step 1.

To be proved soon.

"Optimal substructure" Lemma: FF can be used for the remaining steps.

Obvious.

Implication: There exists an optimal solution in which FF is used at all steps.

Proof of Greedy choice Lemma

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n]$ = eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n] =$ eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything.

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n]$ = eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything. So $E = E^*$ is the required eviction sequence.

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n]$ = eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything. So $E = E^*$ is the required eviction sequence.

Case $r[1] \notin I$: Let x = item in I whose next access time is largest.

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n] =$ eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything. So $E = E^*$ is the required eviction sequence.

Case $r[1] \notin I$: Let $x =$ item in I whose next access time is largest. Then $E[1] = x$.

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n]$ = eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything. So $E = E^*$ is the required eviction sequence.

Case $r[1] \notin I$: Let x = item in I whose next access time is largest. Then $E[1] = x$. If $E^*[1] = x$, we are done.

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n]$ = eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything. So $E = E^*$ is the required eviction sequence.

Case $r[1] \notin I$: Let x = item in I whose next access time is largest. Then $E[1] = x$. If $E^*[1] = x$, we are done. So assume $E^*[1] \neq x$.

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n] =$ eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything. So $E = E^*$ is the required eviction sequence.

Case $r[1] \notin I$: Let $x =$ item in I whose next access time is largest. Then $E[1] = x$. If $E^*[1] = x$, we are done. So assume $E^*[1] \neq x$.

After step 1, cache content is different for OPT and FF1.

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n]$ = eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything. So $E = E^*$ is the required eviction sequence.

Case $r[1] \notin I$: Let x = item in I whose next access time is largest. Then $E[1] = x$. If $E^*[1] = x$, we are done. So assume $E^*[1] \neq x$.

After step 1, cache content is different for OPT and FF1.
Let $C^*[t]$ = cache content after step t when OPT is used.

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n] =$ eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything. So $E = E^*$ is the required eviction sequence.

Case $r[1] \notin I$: Let $x =$ item in I whose next access time is largest. Then $E[1] = x$. If $E^*[1] = x$, we are done. So assume $E^*[1] \neq x$.

After step 1, cache content is different for OPT and FF1.
Let $C^*[t] =$ cache content after step t when OPT is used.
Let $C[t] =$ cache content after step t when FF1 is used.

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n] =$ eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything. So $E = E^*$ is the required eviction sequence.

Case $r[1] \notin I$: Let $x =$ item in I whose next access time is largest. Then $E[1] = x$. If $E^*[1] = x$, we are done. So assume $E^*[1] \neq x$.

After step 1, cache content is different for OPT and FF1.
Let $C^*[t] =$ cache content after step t when OPT is used.
Let $C[t] =$ cache content after step t when FF1 is used.

We next show how to construct $E[2..T]$ such that $E[2..T], E^*[2..T]$ cause the same number of evictions,

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n]$ = eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything. So $E = E^*$ is the required eviction sequence.

Case $r[1] \notin I$: Let x = item in I whose next access time is largest. Then $E[1] = x$. If $E^*[1] = x$, we are done. So assume $E^*[1] \neq x$.

After step 1, cache content is different for OPT and FF1.
Let $C^*[t]$ = cache content after step t when OPT is used.
Let $C[t]$ = cache content after step t when FF1 is used.

We next show how to construct $E[2..T]$ such that
$E[2..T], E^*[2..T]$ cause the same number of evictions,
$C[T] = C^*[T]$

Proof of Greedy choice Lemma

Lemma: For any instance $r[1..n], I$, there exists an optimal eviction sequence E[1..n] in which FF is used in step 1. "FF1"

Proof Let $E^*[1..n]$ = eviction sequence produced by an optimal algorithm, with FF not necessarily used in step 1.

Case $r[1] \in I$: Clearly, neither OPT nor FF will evict anything. So $E = E^*$ is the required eviction sequence.

Case $r[1] \notin I$: Let x = item in I whose next access time is largest. Then $E[1] = x$. If $E^*[1] = x$, we are done. So assume $E^*[1] \neq x$.

After step 1, cache content is different for OPT and FF1.
Let $C^*[t]$ = cache content after step t when OPT is used.
Let $C[t]$ = cache content after step t when FF1 is used.

We next show how to construct $E[2..T]$ such that
$E[2..T], E^*[2..T]$ cause the same number of evictions,
$C[T] = C^*[T]$
Then $E[1..t]||E^*[t+1..n]$ will be the desired sequence.

Remarks

Remarks

T will be fixed soon.

Remarks

T will be fixed soon.

In step 1 OPT throws out $E^*[1]$, FF1 throws out $E[1]$

Remarks

T will be fixed soon.

In step 1 OPT throws out $E^*[1]$, FF1 throws out $E[1]$

Rest of the cache is same.

Remarks

T will be fixed soon.

In step 1 OPT throws out $E^*[1]$, FF1 throws out $E[1]$

Rest of the cache is same.

We write this as: $C[1] - C^*[1] = E^*[1] - E[1]$.

Remarks

T will be fixed soon.

In step 1 OPT throws out $E^*[1]$, FF1 throws out $E[1]$

Rest of the cache is same.

We write this as: $C[1] - C^*[1] = E^*[1] - E[1]$.
Will ensure invariant $C[t] - C^*[t] = E^*[1] - E[1]$ for $t = 2..T-1$.

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

Construction of $E[2..t]$

T = smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Construction of $E[2..t]$

T = smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$:

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?

Construction of $E[2..t]$

T = smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?
$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?
$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)
$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches.

Construction of $E[2..t]$

T = smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?
$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)
$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Construction of $E[2..t]$

T = smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?

$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)

$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?

$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)

$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.

Construction of $E[2..t]$

T = smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?
$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)
$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.
$r[T] \notin C^*[T-1]$

Construction of $E[2..t]$

T = smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?
$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)
$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.
$r[T] \notin C^*[T-1] = C[T-1] + E[1] - E^*[1]$

Construction of $E[2..t]$

T = smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?
$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)
$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.
$r[T] \notin C^*[T-1] = C[T-1] + E[1] - E^*[1]$
$r[T] \neq E^*[1] \Rightarrow$ Eviction needed also with FF1.

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?
$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)
$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.
$r[T] \notin C^*[T-1] = C[T-1] + E[1] - E^*[1]$
$r[T] \neq E^*[1] \Rightarrow$ Eviction needed also with FF1.
FF1 evicts $E^*[1]$: $E[T] = E^*[1]$

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?
$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)
$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.
$r[T] \notin C^*[T-1] = C[T-1] + E[1] - E^*[1]$
$r[T] \neq E^*[1] \Rightarrow$ Eviction needed also with FF1.
FF1 evicts $E^*[1]$: $E[T] = E^*[1]$ Caches now same!

Construction of $E[2..t]$

T = smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?
$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)
$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.
$r[T] \notin C^*[T-1] = C[T-1] + E[1] - E^*[1]$
$r[T] \neq E^*[1] \Rightarrow$ Eviction needed also with FF1.
FF1 evicts $E^*[1]$: $E[T] = E^*[1]$ Caches now same!

Event 2 happens at time T: OPT fetches $E^*[1]$

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?
$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)
$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.
$r[T] \notin C^*[T-1] = C[T-1] + E[1] - E^*[1]$
$r[T] \neq E^*[1] \Rightarrow$ Eviction needed also with FF1.
FF1 evicts $E^*[1]$: $E[T] = E^*[1]$

Caches now same!

Event 2 happens at time T: OPT fetches $E^*[1]$
$E^*[T] = E[1] \Rightarrow$

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?
$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)
$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.
$r[T] \notin C^*[T-1] = C[T-1] + E[1] - E^*[1]$
$r[T] \neq E^*[1] \Rightarrow$ Eviction needed also with FF1.
FF1 evicts $E^*[1]$: $E[T] = E^*[1]$

Caches now same!

Event 2 happens at time T: OPT fetches $E^*[1]$
$E^*[T] = E[1] \Rightarrow E[T] =$"-", FF1 fetches nothing.

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?

$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)

$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.

$r[T] \notin C^*[T-1] = C[T-1] + E[1] - E^*[1]$

$r[T] \neq E^*[1] \Rightarrow$ Eviction needed also with FF1.

FF1 evicts $E^*[1]$: $E[T] = E^*[1]$ Caches now same!

Event 2 happens at time T: OPT fetches $E^*[1]$

$E^*[T] = E[1] \Rightarrow E[T] =$"-", FF1 fetches nothing. Caches same!

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?

$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)

$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.

$r[T] \notin C^*[T-1] = C[T-1] + E[1] - E^*[1]$

$r[T] \neq E^*[1] \Rightarrow$ Eviction needed also with FF1.

FF1 evicts $E^*[1]$: $E[T] = E^*[1]$ Caches now same!

Event 2 happens at time T: OPT fetches $E^*[1]$

$E^*[T] = E[1] \Rightarrow E[T] =$"-", FF1 fetches nothing. Caches same!

$E^*[T] \neq E[1] \Rightarrow$

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?

$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)

$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.

$r[T] \notin C^*[T-1] = C[T-1] + E[1] - E^*[1]$

$r[T] \neq E^*[1] \Rightarrow$ Eviction needed also with FF1.

FF1 evicts $E^*[1]$: $E[T] = E^*[1]$ Caches now same!

Event 2 happens at time T: OPT fetches $E^*[1]$

$E^*[T] = E[1] \Rightarrow E[T] =$"-", FF1 fetches nothing. Caches same!

$E^*[T] \neq E[1] \Rightarrow E[T] = E^*[T]$, FF1 fetches E[1]

Construction of $E[2..t]$

$T =$ smallest time when one of the following events happen:

1. $r[T] \neq E[1], E^*[1]$, and $E^*[T] = E[1]$
2. $r[T] = E^*[1]$

Action of FF1 for $t = 2..T-1$: What can happen at t?

$r[T] = E[1]$ cannot happen before $r[T] = E^*[1]$ (FF property)

$r[t] \neq E[1], E^*[1], E^*[t] \neq E[1]$: OPT evicted something that was in both caches. We make FF1 do the same. $E[t] = E^*[t]$.

Invariant maintained

Event 1 happens at time T: $\Rightarrow$ Eviction for OPT.

$r[T] \notin C^*[T-1] = C[T-1] + E[1] - E^*[1]$

$r[T] \neq E^*[1] \Rightarrow$ Eviction needed also with FF1.

FF1 evicts $E^*[1]$: $E[T] = E^*[1]$ Caches now same!

Event 2 happens at time T: OPT fetches $E^*[1]$

$E^*[T] = E[1] \Rightarrow E[T] =$"-", FF1 fetches nothing. Caches same!

$E^*[T] \neq E[1] \Rightarrow E[T] = E^*[T]$, FF1 fetches E[1] Caches same!