Introduction to Online Algorithms

Subir Kumar Ghosh[1]
Department of Computer Science
School of Mathematical Sciences
Ramakrishna Mission Vivekananda University
Belur Math, Howrah 711202
subir.ghosh@rkmvu.ac.in

(This lecture is delivered in the workshop "Expository Lectures on Graph and Geometric Algorithms" held at BITS Pilani, Hyderabad Campus during September 21-22, 2018.)

[1]Formerly at School of Technology & Computer Science, Tata Institute of Fundamental Research, Mumbai 400005, India

Overview

1. Competitive Analysis

Overview

1. Competitive Analysis
2. The Paging Problem
3. Machine Learning

Overview

1. Competitive Analysis
2. The Paging Problem
3. Machine Learning
4. Graph Colouring

Overview

1. Competitive Analysis
2. The Paging Problem
3. Machine Learning
4. Graph Colouring
5. Searching for a Target in an Unbounded Region

Competitive Analysis of Online Algorithms

- An online algorithm A is presented with a *request sequence* $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$.

Competitive Analysis of Online Algorithms

- An online algorithm A is presented with a *request sequence* $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$.
- The algorithm A has to serve each request online, i.e., without the knowledge of future requests.

Competitive Analysis of Online Algorithms

- An online algorithm A is presented with a *request sequence* $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$.
- The algorithm A has to serve each request online, i.e., without the knowledge of future requests.
- This means that when serving request $\sigma(t), 1 \leq t \leq m$, the algorithm A does not know any request $\sigma(t')$ for $t' > t$.

Competitive Analysis of Online Algorithms

- An online algorithm A is presented with a *request sequence* $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$.
- The algorithm A has to serve each request online, i.e., without the knowledge of future requests.
- This means that when serving request $\sigma(t), 1 \leq t \leq m$, the algorithm A does not know any request $\sigma(t')$ for $t' > t$.
- Serving requests incurs cost, and the goal is to serve the entire request sequence so that the total cost is as small as possible.

Competitive Analysis of Online Algorithms

- An online algorithm A is presented with a *request sequence* $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$.
- The algorithm A has to serve each request online, i.e., without the knowledge of future requests.
- This means that when serving request $\sigma(t), 1 \leq t \leq m$, the algorithm A does not know any request $\sigma(t')$ for $t' > t$.
- Serving requests incurs cost, and the goal is to serve the entire request sequence so that the total cost is as small as possible.
- This setting can also be regarded as a *request-answer game:* An adversary generates requests, and an online algorithm has to serve them one at a time.

1. S. Albers, *Competitive Online Algorithms*, Lecture notes, Max Plank Institute, Saarbrucken, 1996.

.

- Sleator and Tarjan suggested to evaluate the performance on an online algorithm using competitive analysis.

- Sleator and Tarjan suggested to evaluate the performance on an online algorithm using competitive analysis.
- In a competitive analysis, an online algorithm A is compared to an *optimal offline algorithm*.

- Sleator and Tarjan suggested to evaluate the performance on an online algorithm using competitive analysis.
- In a competitive analysis, an online algorithm A is compared to an *optimal offline algorithm*.
- An optimal offline algorithm knows the entire request sequence in advance and can serve it with minimum cost.

- Sleator and Tarjan suggested to evaluate the performance on an online algorithm using competitive analysis.
- In a competitive analysis, an online algorithm A is compared to an *optimal offline algorithm*.
- An optimal offline algorithm knows the entire request sequence in advance and can serve it with minimum cost.
- Given a request sequence σ, let $C_A(\sigma)$ denote the cost incurred by A and let $C_{OPT}(\sigma)$ denote the cost paid by an optimal offline algorithm OPT.

- Sleator and Tarjan suggested to evaluate the performance on an online algorithm using competitive analysis.
- In a competitive analysis, an online algorithm A is compared to an *optimal offline algorithm*.
- An optimal offline algorithm knows the entire request sequence in advance and can serve it with minimum cost.
- Given a request sequence σ, let $C_A(\sigma)$ denote the cost incurred by A and let $C_{OPT}(\sigma)$ denote the cost paid by an optimal offline algorithm OPT.
- The algorithm A is called c-competitive if there exists a constant c such that $C_A(\sigma) \leq c \times C_{OPT}(\sigma) + a$ for all request sequences σ.

- Sleator and Tarjan suggested to evaluate the performance on an online algorithm using competitive analysis.
- In a competitive analysis, an online algorithm A is compared to an *optimal offline algorithm*.
- An optimal offline algorithm knows the entire request sequence in advance and can serve it with minimum cost.
- Given a request sequence σ, let $C_A(\sigma)$ denote the cost incurred by A and let $C_{OPT}(\sigma)$ denote the cost paid by an optimal offline algorithm OPT.
- The algorithm A is called c-competitive if there exists a constant c such that $C_A(\sigma) \leq c \times C_{OPT}(\sigma) + a$ for all request sequences σ.
- Here we assume that A is a deterministic online algorithm, and the factor c is also called the *competitive ratio* of A.

- Sleator and Tarjan suggested to evaluate the performance on an online algorithm using competitive analysis.
- In a competitive analysis, an online algorithm A is compared to an *optimal offline algorithm*.
- An optimal offline algorithm knows the entire request sequence in advance and can serve it with minimum cost.
- Given a request sequence σ, let $C_A(\sigma)$ denote the cost incurred by A and let $C_{OPT}(\sigma)$ denote the cost paid by an optimal offline algorithm OPT.
- The algorithm A is called c-competitive if there exists a constant c such that $C_A(\sigma) \leq c \times C_{OPT}(\sigma) + a$ for all request sequences σ.
- Here we assume that A is a deterministic online algorithm, and the factor c is also called the *competitive ratio* of A.
- The makespan algorithm for online scheduling of jobs, where jobs are assigned to the least loaded machine in an online fashion, is an online algorithm and its competitive ratio is same as the approximation ratio 2 of the corresponding offline algorithm.

The Paging Problem

- Consider a two-level memory system that consists of a small fast memory and a large slow memory.

The Paging Problem

- Consider a two-level memory system that consists of a small fast memory and a large slow memory.
- Here, each request species a page in the memory system.

The Paging Problem

- Consider a two-level memory system that consists of a small fast memory and a large slow memory.
- Here, each request species a page in the memory system.
- A request is served if the corresponding page is in fast memory.

The Paging Problem

- Consider a two-level memory system that consists of a small fast memory and a large slow memory.
- Here, each request species a page in the memory system.
- A request is served if the corresponding page is in fast memory.
- If a requested page is not in fast memory, a *page fault* occurs.

The Paging Problem

- Consider a two-level memory system that consists of a small fast memory and a large slow memory.
- Here, each request species a page in the memory system.
- A request is served if the corresponding page is in fast memory.
- If a requested page is not in fast memory, a *page fault* occurs.
- Then a page must be moved from fast memory to slow memory so that the requested page can be loaded into the vacated location.

The Paging Problem

- Consider a two-level memory system that consists of a small fast memory and a large slow memory.
- Here, each request species a page in the memory system.
- A request is served if the corresponding page is in fast memory.
- If a requested page is not in fast memory, a *page fault* occurs.
- Then a page must be moved from fast memory to slow memory so that the requested page can be loaded into the vacated location.
- A paging algorithm specifies which page to evict on a fault.

The Paging Problem

- Consider a two-level memory system that consists of a small fast memory and a large slow memory.
- Here, each request species a page in the memory system.
- A request is served if the corresponding page is in fast memory.
- If a requested page is not in fast memory, a *page fault* occurs.
- Then a page must be moved from fast memory to slow memory so that the requested page can be loaded into the vacated location.
- A paging algorithm specifies which page to evict on a fault.
- For online paging algorithms, the decision which page to evict must be made without the knowledge of any future requests.

The Paging Problem

- Consider a two-level memory system that consists of a small fast memory and a large slow memory.
- Here, each request species a page in the memory system.
- A request is served if the corresponding page is in fast memory.
- If a requested page is not in fast memory, a *page fault* occurs.
- Then a page must be moved from fast memory to slow memory so that the requested page can be loaded into the vacated location.
- A paging algorithm specifies which page to evict on a fault.
- For online paging algorithms, the decision which page to evict must be made without the knowledge of any future requests.
- The cost to be minimized is the total number of page faults incurred on a request sequence.

Deterministic online Paging Algorithms

- We list four well-known deterministic online paging algorithms:
  - LRU (Least Recently Used): On a fault, evict the page in fast memory that was requested least recently.

Deterministic online Paging Algorithms

- We list four well-known deterministic online paging algorithms:
  - LRU (Least Recently Used): On a fault, evict the page in fast memory that was requested least recently.
  - FIFO (First-In First-Out): Evict the page that has been in fast memory longest.

Deterministic online Paging Algorithms

- We list four well-known deterministic online paging algorithms:
  - LRU (Least Recently Used): On a fault, evict the page in fast memory that was requested least recently.
  - FIFO (First-In First-Out): Evict the page that has been in fast memory longest.
  - LFU (Least Frequently Used): Evict the page that has been requested least frequently.

Deterministic online Paging Algorithms

- We list four well-known deterministic online paging algorithms:
  - LRU (Least Recently Used): On a fault, evict the page in fast memory that was requested least recently.
  - FIFO (First-In First-Out): Evict the page that has been in fast memory longest.
  - LFU (Least Frequently Used): Evict the page that has been requested least frequently.
  - LIFO (Last-in-First-Out): Evict the page that has been in the fast memory shortest.
- Before analyzing these algorithms, we remark that Belady exhibited an optimal offline algorithm (called MIN) for the paging problem.

Deterministic online Paging Algorithms

- We list four well-known deterministic online paging algorithms:
  - LRU (Least Recently Used): On a fault, evict the page in fast memory that was requested least recently.
  - FIFO (First-In First-Out): Evict the page that has been in fast memory longest.
  - LFU (Least Frequently Used): Evict the page that has been requested least frequently.
  - LIFO (Last-in-First-Out): Evict the page that has been in the fast memory shortest.
- Before analyzing these algorithms, we remark that Belady exhibited an optimal offline algorithm (called MIN) for the paging problem.
- On a fault, MIN evicts the page whose next request occurs furthest in the future.

Deterministic online Paging Algorithms

- We list four well-known deterministic online paging algorithms:
  - LRU (Least Recently Used): On a fault, evict the page in fast memory that was requested least recently.
  - FIFO (First-In First-Out): Evict the page that has been in fast memory longest.
  - LFU (Least Frequently Used): Evict the page that has been requested least frequently.
  - LIFO (Last-in-First-Out): Evict the page that has been in the fast memory shortest.
- Before analyzing these algorithms, we remark that Belady exhibited an optimal offline algorithm (called MIN) for the paging problem.
- On a fault, MIN evicts the page whose next request occurs furthest in the future.
- Belady showed that on any sequence of requests, MIN achieves the minimum number of page faults.

- Observe that the algorithm LFU and LIFO are not competitive as follows.

- Observe that the algorithm LFU and LIFO are not competitive as follows.
- Consider the sequence $\sigma = p_1, p_1, p_2, p_2, p_3, p_4, p_3, p_4, \ldots$.

- Observe that the algorithm LFU and LIFO are not competitive as follows.
- Consider the sequence $\sigma = p_1, p_1, p_2, p_2, p_3, p_4, p_3, p_4, \ldots$.
- In this sequence for the fast memory of size 3, LFU keeps on exchanging p_3 and p_4 ad infinitum, while the optimum can keep these two pages in the cache.

- Observe that the algorithm LFU and LIFO are not competitive as follows.
- Consider the sequence $\sigma = p_1, p_1, p_2, p_2, p_3, p_4, p_3, p_4, \ldots$.
- In this sequence for the fast memory of size 3, LFU keeps on exchanging p_3 and p_4 ad infinitum, while the optimum can keep these two pages in the cache.
- Consider the request sequence $\sigma = p_1, p_2, p_3, p_4, p_3, p_4, \ldots$.

- Observe that the algorithm LFU and LIFO are not competitive as follows.
- Consider the sequence $\sigma = p_1, p_1, p_2, p_2, p_3, p_4, p_3, p_4, \ldots$.
- In this sequence for the fast memory of size 3, LFU keeps on exchanging p_3 and p_4 ad infinitum, while the optimum can keep these two pages in the cache.
- Consider the request sequence $\sigma = p_1, p_2, p_3, p_4, p_3, p_4, \ldots$.
- In this sequence for the fast memory of size 3, LIFO keeps on exchanging p_3 and p_4 ad infinitum, while the optimum can keep these two pages in the cache. This is therefore not competitive.

- Let k denote the number of pages that can simultaneously reside in a fast memory.

- Let k denote the number of pages that can simultaneously reside in a fast memory.
- Sleator and Tarjan proved the algorithms LRU and FIFO are k-competitive as follows.

- Let k denote the number of pages that can simultaneously reside in a fast memory.
- Sleator and Tarjan proved the algorithms LRU and FIFO are k-competitive as follows.
- We show that LRU is k-competitive. The analysis for FIFO is very similar.
- Without loss of generality we assume that LRU and OPT initially start with the same fast memory.

- Let k denote the number of pages that can simultaneously reside in a fast memory.
- Sleator and Tarjan proved the algorithms LRU and FIFO are k-competitive as follows.
- We show that LRU is k-competitive. The analysis for FIFO is very similar.
- Without loss of generality we assume that LRU and OPT initially start with the same fast memory.
- Consider any request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$.

- Let k denote the number of pages that can simultaneously reside in a fast memory.
- Sleator and Tarjan proved the algorithms LRU and FIFO are k-competitive as follows.
- We show that LRU is k-competitive. The analysis for FIFO is very similar.
- Without loss of generality we assume that LRU and OPT initially start with the same fast memory.
- Consider any request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$.
- We prove that $C_{LRU}(\sigma) \leq k \times C_{OPT}(\sigma)$.

- Let k denote the number of pages that can simultaneously reside in a fast memory.
- Sleator and Tarjan proved the algorithms LRU and FIFO are k-competitive as follows.
- We show that LRU is k-competitive. The analysis for FIFO is very similar.
- Without loss of generality we assume that LRU and OPT initially start with the same fast memory.
- Consider any request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$.
- We prove that $C_{LRU}(\sigma) \leq k \times C_{OPT}(\sigma)$.
- We partition σ into phases $P(0), P(1), P(2), \ldots$ such that LRU has at most k faults on $P(0)$ and exactly k faults on $P(i)$, for every $i \geq 1$.

- Let k denote the number of pages that can simultaneously reside in a fast memory.
- Sleator and Tarjan proved the algorithms LRU and FIFO are k-competitive as follows.
- We show that LRU is k-competitive. The analysis for FIFO is very similar.
- Without loss of generality we assume that LRU and OPT initially start with the same fast memory.
- Consider any request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$.
- We prove that $C_{LRU}(\sigma) \leq k \times C_{OPT}(\sigma)$.
- We partition σ into phases $P(0), P(1), P(2), \ldots$ such that LRU has at most k faults on $P(0)$ and exactly k faults on $P(i)$, for every $i \geq 1$.
- Scan the request from the end, and whenever there are k faults made by LRU, start a new phase.

- To establish the desired bound, we have to show that OPT has at least one page fault during each phase.

- To establish the desired bound, we have to show that OPT has at least one page fault during each phase.
- For phase $P(0)$, OPT has a page fault on the first request on which LRU has a fault, as LRU and OPT have started with the same fast memory.

- To establish the desired bound, we have to show that OPT has at least one page fault during each phase.
- For phase $P(0)$, OPT has a page fault on the first request on which LRU has a fault, as LRU and OPT have started with the same fast memory.
- Consider an arbitrary phase $P(i), i \geq 1$. Let $\sigma(t_i)$ be the first request in $P(i)$ and let $\sigma(t_{i+1} - 1)$ be the last request in $P(i)$.

- To establish the desired bound, we have to show that OPT has at least one page fault during each phase.
- For phase $P(0)$, OPT has a page fault on the first request on which LRU has a fault, as LRU and OPT have started with the same fast memory.
- Consider an arbitrary phase $P(i), i \geq 1$. Let $\sigma(t_i)$ be the first request in $P(i)$ and let $\sigma(t_{i+1} - 1)$ be the last request in $P(i)$.
- Furthermore, let p be the page that is requested last in $P(i-1)$.

- To establish the desired bound, we have to show that OPT has at least one page fault during each phase.
- For phase $P(0)$, OPT has a page fault on the first request on which LRU has a fault, as LRU and OPT have started with the same fast memory.
- Consider an arbitrary phase $P(i), i \geq 1$. Let $\sigma(t_i)$ be the first request in $P(i)$ and let $\sigma(t_{i+1} - 1)$ be the last request in $P(i)$.
- Furthermore, let p be the page that is requested last in $P(i - 1)$.
- If $P(i)$ contains requests to k distinct pages that are different from p, then OPT must have a page fault in $P(i)$ as OPT has page p in its fast memory at the end of $P(i - 1)$ and thus cannot have all the other k pages request in $P(i)$ in its fast memory.

- To establish the desired bound, we have to show that OPT has at least one page fault during each phase.
- For phase $P(0)$, OPT has a page fault on the first request on which LRU has a fault, as LRU and OPT have started with the same fast memory.
- Consider an arbitrary phase $P(i), i \geq 1$. Let $\sigma(t_i)$ be the first request in $P(i)$ and let $\sigma(t_{i+1} - 1)$ be the last request in $P(i)$.
- Furthermore, let p be the page that is requested last in $P(i-1)$.
- If $P(i)$ contains requests to k distinct pages that are different from p, then OPT must have a page fault in $P(i)$ as OPT has page p in its fast memory at the end of $P(i-1)$ and thus cannot have all the other k pages request in $P(i)$ in its fast memory.
- So, if the k requests on which LRU has a fault are to k distinct pages and if these pages are also different from p, then the bound holds.

- So suppose that LRU faults twice on a page $q \in P(i)$.

- So suppose that LRU faults twice on a page $q \in P(i)$.
- Assume that LRU has a fault on $\sigma(s_1) = q$ and $\sigma(s_2) = q$, with $t_i < s_1 < s_2 < t_{i+1} - 1$.

- So suppose that LRU faults twice on a page $q \in P(i)$.
- Assume that LRU has a fault on $\sigma(s_1) = q$ and $\sigma(s_2) = q$, with $t_i < s_1 < s_2 < t_{i+1} - 1$.
- Page q is in LRU's fast memory immediately after $\sigma(s_1)$ is served and is evicted at some time t with $s_1 < t < s_2$.

- So suppose that LRU faults twice on a page $q \in P(i)$.
- Assume that LRU has a fault on $\sigma(s_1) = q$ and $\sigma(s_2) = q$, with $t_i < s_1 < s_2 < t_{i+1} - 1$.
- Page q is in LRU's fast memory immediately after $\sigma(s_1)$ is served and is evicted at some time t with $s_1 < t < s_2$.
- When q is evicted, it is the least recently requested page in fast memory.

- So suppose that LRU faults twice on a page $q \in P(i)$.
- Assume that LRU has a fault on $\sigma(s_1) = q$ and $\sigma(s_2) = q$, with $t_i < s_1 < s_2 < t_{i+1} - 1$.
- Page q is in LRU's fast memory immediately after $\sigma(s_1)$ is served and is evicted at some time t with $s_1 < t < s_2$.
- When q is evicted, it is the least recently requested page in fast memory.
- Thus the subsequence $\sigma(s_1), \ldots, \sigma(s_2)$ contains requests to $k + 1$ distinct pages, at least k of which must be different from q.

- So suppose that LRU faults twice on a page $q \in P(i)$.
- Assume that LRU has a fault on $\sigma(s_1) = q$ and $\sigma(s_2) = q$, with $t_i < s_1 < s_2 < t_{i+1} - 1$.
- Page q is in LRU's fast memory immediately after $\sigma(s_1)$ is served and is evicted at some time t with $s_1 < t < s_2$.
- When q is evicted, it is the least recently requested page in fast memory.
- Thus the subsequence $\sigma(s_1), \ldots, \sigma(s_2)$ contains requests to $k + 1$ distinct pages, at least k of which must be different from q.
- Finally suppose that within $P(i)$, LRU does not fault twice on a page but on one of the faults, page p is request.

- So suppose that LRU faults twice on a page $q \in P(i)$.
- Assume that LRU has a fault on $\sigma(s_1) = q$ and $\sigma(s_2) = q$, with $t_i < s_1 < s_2 < t_{i+1} - 1$.
- Page q is in LRU's fast memory immediately after $\sigma(s_1)$ is served and is evicted at some time t with $s_1 < t < s_2$.
- When q is evicted, it is the least recently requested page in fast memory.
- Thus the subsequence $\sigma(s_1), \ldots, \sigma(s_2)$ contains requests to $k+1$ distinct pages, at least k of which must be different from q.
- Finally suppose that within $P(i)$, LRU does not fault twice on a page but on one of the faults, page p is request.
- Let $t > t_i$ be the first time when p is evicted. Using the same arguments as above, we obtain that the subsequence $\sigma(t_i - 1), \sigma(t_i), \ldots, \sigma(t)$ must contain $k+1$ distinct pages.
- Hence, LRU is k-competitive.

- Sleator and Tarjan also showed that if a deterministic online paging algorithm A is $c-$competitive (using LRU or FIFO strategy), then $c \geq k$ as follows.

- Sleator and Tarjan also showed that if a deterministic online paging algorithm A is $c-$competitive (using LRU or FIFO strategy), then $c \geq k$ as follows.
- Let $S = \{p_1, p_2, \ldots, p_{k+1}\}$ be a set of $k+1$ arbitrary pages.

- Sleator and Tarjan also showed that if a deterministic online paging algorithm A is $c-$competitive (using LRU or FIFO strategy), then $c \geq k$ as follows.
- Let $S = \{p_1, p_2, \ldots, p_{k+1}\}$ be a set of $k+1$ arbitrary pages.
- Assume without loss of generality that A and OPT initially have $p_1, p_2, \ldots, p_k$ in their fast memories.

- Sleator and Tarjan also showed that if a deterministic online paging algorithm A is $c-$competitive (using LRU or FIFO strategy), then $c \geq k$ as follows.
- Let $S = \{p_1, p_2, \ldots, p_{k+1}\}$ be a set of $k + 1$ arbitrary pages.
- Assume without loss of generality that A and OPT initially have $p_1, p_2, \ldots, p_k$ in their fast memories.
- Consider the request sequence where each request is made to the page that is not in the fast memory of A.

- Sleator and Tarjan also showed that if a deterministic online paging algorithm A is $c-$competitive (using LRU or FIFO strategy), then $c \geq k$ as follows.
- Let $S = \{p_1, p_2, \ldots, p_{k+1}\}$ be a set of $k+1$ arbitrary pages.
- Assume without loss of generality that A and OPT initially have $p_1, p_2, \ldots, p_k$ in their fast memories.
- Consider the request sequence where each request is made to the page that is not in the fast memory of A.
- So, A has a page fault on every request.

- Sleator and Tarjan also showed that if a deterministic online paging algorithm A is $c-$competitive (using LRU or FIFO strategy), then $c \geq k$ as follows.
- Let $S = \{p_1, p_2, \ldots, p_{k+1}\}$ be a set of $k+1$ arbitrary pages.
- Assume without loss of generality that A and OPT initially have $p_1, p_2, \ldots, p_k$ in their fast memories.
- Consider the request sequence where each request is made to the page that is not in the fast memory of A.
- So, A has a page fault on every request.
- Suppose that OPT has a fault on some request $\sigma(t)$.

- Sleator and Tarjan also showed that if a deterministic online paging algorithm A is $c-$competitive (using LRU or FIFO strategy), then $c \geq k$ as follows.
- Let $S = \{p_1, p_2, \ldots, p_{k+1}\}$ be a set of $k+1$ arbitrary pages.
- Assume without loss of generality that A and OPT initially have $p_1, p_2, \ldots, p_k$ in their fast memories.
- Consider the request sequence where each request is made to the page that is not in the fast memory of A.
- So, A has a page fault on every request.
- Suppose that OPT has a fault on some request $\sigma(t)$.
- When serving $\sigma(t)$, OPT can evict a page that is not requested during the next $k-1$ requests $\sigma(t+1), \ldots, \sigma(t+k-1)$.

- Sleator and Tarjan also showed that if a deterministic online paging algorithm A is $c-$competitive (using LRU or FIFO strategy), then $c \geq k$ as follows.
- Let $S = \{p_1, p_2, \ldots, p_{k+1}\}$ be a set of $k+1$ arbitrary pages.
- Assume without loss of generality that A and OPT initially have $p_1, p_2, \ldots, p_k$ in their fast memories.
- Consider the request sequence where each request is made to the page that is not in the fast memory of A.
- So, A has a page fault on every request.
- Suppose that OPT has a fault on some request $\sigma(t)$.
- When serving $\sigma(t)$, OPT can evict a page that is not requested during the next $k-1$ requests $\sigma(t+1), \ldots, \sigma(t+k-1)$.
- Thus, on any k consecutive requests, OPT has at most one fault.

- Sleator and Tarjan also showed that if a deterministic online paging algorithm A is $c-$competitive (using LRU or FIFO strategy), then $c \geq k$ as follows.
- Let $S = \{p_1, p_2, \ldots, p_{k+1}\}$ be a set of $k+1$ arbitrary pages.
- Assume without loss of generality that A and OPT initially have $p_1, p_2, \ldots, p_k$ in their fast memories.
- Consider the request sequence where each request is made to the page that is not in the fast memory of A.
- So, A has a page fault on every request.
- Suppose that OPT has a fault on some request $\sigma(t)$.
- When serving $\sigma(t)$, OPT can evict a page that is not requested during the next $k-1$ requests $\sigma(t+1), \ldots, \sigma(t+k-1)$.
- Thus, on any k consecutive requests, OPT has at most one fault.
- Hence, A is at least k-competitive.

- The competitive ratios shown for deterministic paging algorithms are not very meaningful from a practical point of view as the performance ratios of LRU and FIFO become worse as the size of the fast memory increases.

- The competitive ratios shown for deterministic paging algorithms are not very meaningful from a practical point of view as the performance ratios of LRU and FIFO become worse as the size of the fast memory increases.
- However, in practice, these algorithms perform better the bigger the fast memory is.

- The competitive ratios shown for deterministic paging algorithms are not very meaningful from a practical point of view as the performance ratios of LRU and FIFO become worse as the size of the fast memory increases.
- However, in practice, these algorithms perform better the bigger the fast memory is.
- Furthermore, the competitive ratios of LRU and FIFO are the same, whereas in practice LRU performs much better.

- The competitive ratios shown for deterministic paging algorithms are not very meaningful from a practical point of view as the performance ratios of LRU and FIFO become worse as the size of the fast memory increases.
- However, in practice, these algorithms perform better the bigger the fast memory is.
- Furthermore, the competitive ratios of LRU and FIFO are the same, whereas in practice LRU performs much better.
- For these reasons, there has been a study of competitive paging algorithms with *access graphs*.

- The competitive ratios shown for deterministic paging algorithms are not very meaningful from a practical point of view as the performance ratios of LRU and FIFO become worse as the size of the fast memory increases.
- However, in practice, these algorithms perform better the bigger the fast memory is.
- Furthermore, the competitive ratios of LRU and FIFO are the same, whereas in practice LRU performs much better.
- For these reasons, there has been a study of competitive paging algorithms with *access graphs*.
- In an access graph, each node represents a page in the memory system. Whenever a page p is requested, the next request can only be to a page that is adjacent to p in the access graph.

- The competitive ratios shown for deterministic paging algorithms are not very meaningful from a practical point of view as the performance ratios of LRU and FIFO become worse as the size of the fast memory increases.
- However, in practice, these algorithms perform better the bigger the fast memory is.
- Furthermore, the competitive ratios of LRU and FIFO are the same, whereas in practice LRU performs much better.
- For these reasons, there has been a study of competitive paging algorithms with *access graphs*.
- In an access graph, each node represents a page in the memory system. Whenever a page p is requested, the next request can only be to a page that is adjacent to p in the access graph.
- Access graphs can model more realistic request sequences that exhibit locality of reference.

- The competitive ratios shown for deterministic paging algorithms are not very meaningful from a practical point of view as the performance ratios of LRU and FIFO become worse as the size of the fast memory increases.
- However, in practice, these algorithms perform better the bigger the fast memory is.
- Furthermore, the competitive ratios of LRU and FIFO are the same, whereas in practice LRU performs much better.
- For these reasons, there has been a study of competitive paging algorithms with *access graphs*.
- In an access graph, each node represents a page in the memory system. Whenever a page p is requested, the next request can only be to a page that is adjacent to p in the access graph.
- Access graphs can model more realistic request sequences that exhibit locality of reference.
- It has been shown that using access graphs, one can overcome some negative aspects of conventional competitive paging results.

On-line Algorithms in Machine Learning

- The areas of On-Line Algorithms and Machine Learning are both concerned with problems of making decisions about the present based only on knowledge of the past.

On-line Algorithms in Machine Learning

- The areas of On-Line Algorithms and Machine Learning are both concerned with problems of making decisions about the present based only on knowledge of the past.
- Although these areas differ in terms of their emphasis and the problems typically studied, there are a collection of results in Computational Learning Theory that fit nicely into the "on-line algorithms" framework.

On-line Algorithms in Machine Learning

- The areas of On-Line Algorithms and Machine Learning are both concerned with problems of making decisions about the present based only on knowledge of the past.
- Although these areas differ in terms of their emphasis and the problems typically studied, there are a collection of results in Computational Learning Theory that fit nicely into the "on-line algorithms" framework.
- Here, we discuss some models and algorithms from Computational Learning Theory that seem particularly interesting from the point of view of on-line algorithms.

On-line Algorithms in Machine Learning

- The areas of On-Line Algorithms and Machine Learning are both concerned with problems of making decisions about the present based only on knowledge of the past.
- Although these areas differ in terms of their emphasis and the problems typically studied, there are a collection of results in Computational Learning Theory that fit nicely into the "on-line algorithms" framework.
- Here, we discuss some models and algorithms from Computational Learning Theory that seem particularly interesting from the point of view of on-line algorithms.

1. A. Blum, *On-line algorithms in machine learning*, In Proceedings of the Workshop on Online Algorithms, DBLP, pp. 306-325, 1996.

- We begin by describing the problem of "predicting from expert advice," which has been studied extensively in the theoretical machine learning literature.

- We begin by describing the problem of "predicting from expert advice," which has been studied extensively in the theoretical machine learning literature.
- Suppose, a learning algorithm is given the task each day of predicting whether or not it will rain that day.

- We begin by describing the problem of "predicting from expert advice," which has been studied extensively in the theoretical machine learning literature.
- Suppose, a learning algorithm is given the task each day of predicting whether or not it will rain that day.
- In order to make this prediction, the algorithm is given as input the advice of n "experts".

- We begin by describing the problem of "predicting from expert advice," which has been studied extensively in the theoretical machine learning literature.
- Suppose, a learning algorithm is given the task each day of predicting whether or not it will rain that day.
- In order to make this prediction, the algorithm is given as input the advice of n "experts".
- Each day, each expert predicts yes or no, and then the learning algorithm uses this information in order to make its own prediction.

- We begin by describing the problem of "predicting from expert advice," which has been studied extensively in the theoretical machine learning literature.
- Suppose, a learning algorithm is given the task each day of predicting whether or not it will rain that day.
- In order to make this prediction, the algorithm is given as input the advice of n "experts".
- Each day, each expert predicts yes or no, and then the learning algorithm uses this information in order to make its own prediction.
- Note that the algorithm is given no other input besides the yes/no bits produced by the experts.

- We begin by describing the problem of "predicting from expert advice," which has been studied extensively in the theoretical machine learning literature.
- Suppose, a learning algorithm is given the task each day of predicting whether or not it will rain that day.
- In order to make this prediction, the algorithm is given as input the advice of n "experts".
- Each day, each expert predicts yes or no, and then the learning algorithm uses this information in order to make its own prediction.
- Note that the algorithm is given no other input besides the yes/no bits produced by the experts.
- After making its prediction, the algorithm is then told whether or not, in fact, it rained that day.

- We begin by describing the problem of "predicting from expert advice," which has been studied extensively in the theoretical machine learning literature.
- Suppose, a learning algorithm is given the task each day of predicting whether or not it will rain that day.
- In order to make this prediction, the algorithm is given as input the advice of n "experts".
- Each day, each expert predicts yes or no, and then the learning algorithm uses this information in order to make its own prediction.
- Note that the algorithm is given no other input besides the yes/no bits produced by the experts.
- After making its prediction, the algorithm is then told whether or not, in fact, it rained that day.
- Suppose, we make no assumptions about the quality or independence of the experts.

- We begin by describing the problem of "predicting from expert advice," which has been studied extensively in the theoretical machine learning literature.
- Suppose, a learning algorithm is given the task each day of predicting whether or not it will rain that day.
- In order to make this prediction, the algorithm is given as input the advice of n "experts".
- Each day, each expert predicts yes or no, and then the learning algorithm uses this information in order to make its own prediction.
- Note that the algorithm is given no other input besides the yes/no bits produced by the experts.
- After making its prediction, the algorithm is then told whether or not, in fact, it rained that day.
- Suppose, we make no assumptions about the quality or independence of the experts.
- So, we cannot hope to achieve any absolute level of quality in our predictions.

- In that case, a natural goal instead is to perform nearly as well as the best expert so far: that is, to guarantee that at any time, our algorithm has not performed much worse than whichever expert has made the fewest mistakes to date.

- In that case, a natural goal instead is to perform nearly as well as the best expert so far: that is, to guarantee that at any time, our algorithm has not performed much worse than whichever expert has made the fewest mistakes to date.
- In the language of competitive analysis, this is the goal of being competitive with respect to the best single expert.

- In that case, a natural goal instead is to perform nearly as well as the best expert so far: that is, to guarantee that at any time, our algorithm has not performed much worse than whichever expert has made the fewest mistakes to date.
- In the language of competitive analysis, this is the goal of being competitive with respect to the best single expert.
- We call the sequence of events in which the algorithm (1) receives the predictions of the experts, (2) makes its own prediction, and then (3) is told the correct answer, a *trial*.

- In that case, a natural goal instead is to perform nearly as well as the best expert so far: that is, to guarantee that at any time, our algorithm has not performed much worse than whichever expert has made the fewest mistakes to date.
- In the language of competitive analysis, this is the goal of being competitive with respect to the best single expert.
- We call the sequence of events in which the algorithm (1) receives the predictions of the experts, (2) makes its own prediction, and then (3) is told the correct answer, a *trial*.
- We assume that predictions belong to the set $\{0, 1\}$.

- In that case, a natural goal instead is to perform nearly as well as the best expert so far: that is, to guarantee that at any time, our algorithm has not performed much worse than whichever expert has made the fewest mistakes to date.
- In the language of competitive analysis, this is the goal of being competitive with respect to the best single expert.
- We call the sequence of events in which the algorithm (1) receives the predictions of the experts, (2) makes its own prediction, and then (3) is told the correct answer, a *trial*.
- We assume that predictions belong to the set $\{0, 1\}$.
- We now describe a simple algorithm called the Weighted Majority algorithm.

- In that case, a natural goal instead is to perform nearly as well as the best expert so far: that is, to guarantee that at any time, our algorithm has not performed much worse than whichever expert has made the fewest mistakes to date.
- In the language of competitive analysis, this is the goal of being competitive with respect to the best single expert.
- We call the sequence of events in which the algorithm (1) receives the predictions of the experts, (2) makes its own prediction, and then (3) is told the correct answer, a *trial*.
- We assume that predictions belong to the set $\{0, 1\}$.
- We now describe a simple algorithm called the Weighted Majority algorithm.
- This algorithm maintains a list of weights $w_1, w_2, \ldots, w_n$, one for each expert, and predicts based on a weighted majority vote of the expert opinions.

1. Initialize the weights $w_1, w_2, \ldots, w_n$ of all the experts to 1.

1. Initialize the weights $w_1, w_2, \ldots, w_n$ of all the experts to 1.
2. Given a set of predictions $\{x_1, \ldots, x_n\}$ by the experts, output the prediction with the highest total weight.

   That is, *if* $\sum_{i:x_i=1} w_i \geq \sum_{i:x_i=0} w_i$ *then* output 1 else output 0.

1. Initialize the weights $w_1, w_2, \ldots, w_n$ of all the experts to 1.
2. Given a set of predictions $\{x_1, \ldots, x_n\}$ by the experts, output the prediction with the highest total weight.

   That is, *if* $\sum_{i:x_i=1} w_i \geq \sum_{i:x_i=0} w_i$ *then* output 1 else output 0.
3. When the correct answer y is received, penalize each mistaken expert by multiplying its weight by $1/2$.

   That is, *if* $x_i \neq y$, *then* $w_i := w_i/2$ *else* w_i is not modified.

1. Initialize the weights $w_1, w_2, \ldots, w_n$ of all the experts to 1.
2. Given a set of predictions $\{x_1, \ldots, x_n\}$ by the experts, output the prediction with the highest total weight.

   That is, *if* $\sum_{i:x_i=1} w_i \geq \sum_{i:x_i=0} w_i$ *then* output 1 else output 0.
3. When the correct answer y is received, penalize each mistaken expert by multiplying its weight by $1/2$.

   That is, *if* $x_i \neq y$, *then* $w_i := w_i/2$ *else* w_i is not modified.
4. Goto 2.

1. Initialize the weights $w_1, w_2, \ldots, w_n$ of all the experts to 1.
2. Given a set of predictions $\{x_1, \ldots, x_n\}$ by the experts, output the prediction with the highest total weight.

   That is, *if* $\sum_{i:x_i=1} w_i \geq \sum_{i:x_i=0} w_i$ *then* output 1 else output 0.
3. When the correct answer y is received, penalize each mistaken expert by multiplying its weight by $1/2$.

   That is, *if* $x_i \neq y$, *then* $w_i := w_i/2$ *else* w_i is not modified.
4. Goto 2.

- **Theorem:** The number of mistakes M made by the Weighted Majority algorithm is at most $2.41(m + \lg n)$, where m is the number of mistakes made by the best expert so far.

1. Initialize the weights $w_1, w_2, \ldots, w_n$ of all the experts to 1.
2. Given a set of predictions $\{x_1, \ldots, x_n\}$ by the experts, output the prediction with the highest total weight.

   That is, *if* $\sum_{i:x_i=1} w_i \geq \sum_{i:x_i=0} w_i$ *then* output 1 else output 0.
3. When the correct answer y is received, penalize each mistaken expert by multiplying its weight by $1/2$.

   That is, *if* $x_i \neq y$, *then* $w_i := w_i/2$ *else* w_i is not modified.
4. Goto 2.

- **Theorem:** The number of mistakes M made by the Weighted Majority algorithm is at most $2.41(m + \lg n)$, where m is the number of mistakes made by the best expert so far.
- *Proof:* Let W denote the total weight of all the experts, so initially $W = n$.

1. Initialize the weights $w_1, w_2, \ldots, w_n$ of all the experts to 1.
2. Given a set of predictions $\{x_1, \ldots, x_n\}$ by the experts, output the prediction with the highest total weight.

   That is, *if* $\sum_{i:x_i=1} w_i \geq \sum_{i:x_i=0} w_i$ *then* output 1 else output 0.
3. When the correct answer y is received, penalize each mistaken expert by multiplying its weight by $1/2$.

   That is, *if* $x_i \neq y$, *then* $w_i := w_i/2$ *else* w_i is not modified.
4. Goto 2.

- **Theorem:** The number of mistakes M made by the Weighted Majority algorithm is at most $2.41(m + \lg n)$, where m is the number of mistakes made by the best expert so far.
- *Proof:* Let W denote the total weight of all the experts, so initially $W = n$.
- If the algorithm makes a mistake, this means that at least half of the total weight of experts predicted incorrectly.

- Therefore in Step 3, the total weight is reduced by at least a factor of $1/4$

- Therefore in Step 3, the total weight is reduced by at least a factor of $1/4$
- Thus if the algorithm makes M mistakes, then $W \leq n(3/4)^M$

- Therefore in Step 3, the total weight is reduced by at least a factor of $1/4$
- Thus if the algorithm makes M mistakes, then $W \leq n(3/4)^M$
- On the other hand, if the best expert has made m mistakes, then its weight is $1/2^m$ and therefore, $W \geq 1/2^m$.

- Therefore in Step 3, the total weight is reduced by at least a factor of $1/4$
- Thus if the algorithm makes M mistakes, then $W \leq n(3/4)^M$
- On the other hand, if the best expert has made m mistakes, then its weight is $1/2^m$ and therefore, $W \geq 1/2^m$.
- Therefore, $1/2^m \leq n(3/4)^M$ or, $-m \ln 2 \leq M \lg(3/4) + \lg n$

- Therefore in Step 3, the total weight is reduced by at least a factor of $1/4$
- Thus if the algorithm makes M mistakes, then $W \leq n(3/4)^M$
- On the other hand, if the best expert has made m mistakes, then its weight is $1/2^m$ and therefore, $W \geq 1/2^m$.
- Therefore, $1/2^m \leq n(3/4)^M$ or, $-m \ln 2 \leq M \lg(3/4) + \lg n$
- Since $0 < \ln 2 < 1$, $-m \leq M \lg(3/4) + \lg n$

- Therefore in Step 3, the total weight is reduced by at least a factor of $1/4$
- Thus if the algorithm makes M mistakes, then $W \leq n(3/4)^M$
- On the other hand, if the best expert has made m mistakes, then its weight is $1/2^m$ and therefore, $W \geq 1/2^m$.
- Therefore, $1/2^m \leq n(3/4)^M$ or, $-m \ln 2 \leq M \lg(3/4) + \lg n$
- Since $0 < \ln 2 < 1$, $-m \leq M \lg(3/4) + \lg n$
- or, $m \geq (-1)M \lg(3/4) - \lg n$, or, $m + \lg n \geq M \lg(4/3)$

- Therefore in Step 3, the total weight is reduced by at least a factor of $1/4$
- Thus if the algorithm makes M mistakes, then $W \leq n(3/4)^M$
- On the other hand, if the best expert has made m mistakes, then its weight is $1/2^m$ and therefore, $W \geq 1/2^m$.
- Therefore, $1/2^m \leq n(3/4)^M$ or, $-m \ln 2 \leq M \lg(3/4) + \lg n$
- Since $0 < \ln 2 < 1$, $-m \leq M \lg(3/4) + \lg n$
- or, $m \geq (-1) M \lg(3/4) - \lg n$, or, $m + \lg n \geq M \lg(4/3)$
- or, $M \leq \frac{1}{\lg(4/3)}(m + \lg n) \leq 2.41(m + \lg n)$.

Online Graph Colouring

- Let $G = (V, E)$ denote a simple graph G with vertex set V and edge set E, where $|V| = n$ and $|E| = m$.

Online Graph Colouring

- Let $G = (V, E)$ denote a simple graph G with vertex set V and edge set E, where $|V| = n$ and $|E| = m$.
- A *colouring* (or, *proper colouring*) of a graph G is a function $\varphi : V(G) \rightarrow \mathbb{N}$ such that whenever a vertex x is adjacent to another vertex y in G, then $\varphi(x) \neq \varphi(y)$.

Online Graph Colouring

- Let $G = (V, E)$ denote a simple graph G with vertex set V and edge set E, where $|V| = n$ and $|E| = m$.
- A *colouring* (or, *proper colouring*) of a graph G is a function $\varphi : V(G) \to \mathbb{N}$ such that whenever a vertex x is adjacent to another vertex y in G, then $\varphi(x) \neq \varphi(y)$.
- Note that all vertices of G with the same colour form an independent set in G.

Online Graph Colouring

- Let $G = (V, E)$ denote a simple graph G with vertex set V and edge set E, where $|V| = n$ and $|E| = m$.
- A *colouring* (or, *proper colouring*) of a graph G is a function $\varphi : V(G) \to \mathbb{N}$ such that whenever a vertex x is adjacent to another vertex y in G, then $\varphi(x) \neq \varphi(y)$.
- Note that all vertices of G with the same colour form an independent set in G.
- The chromatic number of a graph G is defined as the minimum number of colours required in a colouring of G and is denoted by $\chi(G)$.

Online Graph Colouring

- Let $G = (V, E)$ denote a simple graph G with vertex set V and edge set E, where $|V| = n$ and $|E| = m$.
- A *colouring* (or, *proper colouring*) of a graph G is a function $\varphi : V(G) \to \mathbb{N}$ such that whenever a vertex x is adjacent to another vertex y in G, then $\varphi(x) \neq \varphi(y)$.
- Note that all vertices of G with the same colour form an independent set in G.
- The chromatic number of a graph G is defined as the minimum number of colours required in a colouring of G and is denoted by $\chi(G)$.
- The problem of finding $\chi(G)$ for any graph G (MINIMUM GRAPH COLOURING) is NP-complete.

1. A. Gyárás and J. Lehel. *On-line and first-fit colorings of graphs*. Journal of Graph Theory, vol. 12, pp. 217-277, 1988.

- An online colouring of a graph G is an assignment of colours to vertices $V$of G in some order $v_1, \ldots, v_n$ such that (i) the colour of v_i, $1 \leq i \leq k$, is assigned by only looking at the subgraph of G induced by the set $\{v_1, \ldots, v_i\}$, and (ii) the assigned colour of v_i is never changed.

- An online colouring of a graph G is an assignment of colours to vertices $V$of G in some order $v_1, \ldots, v_n$ such that
  (i) the colour of v_i, $1 \leq i \leq k$, is assigned by only looking at the subgraph of G induced by the set $\{v_1, \ldots, v_i\}$, and
  (ii) the assigned colour of v_i is never changed.
- Minimum Online Colouring:
  Instance: $G = (V, E)$ and an ordering $v_1, \ldots, v_n$ of V .
  Solution: An online colouring of G using the given ordering.
  Measure: The number of colours used in the online colouring.
  Goal: Minimize the number of colours.

- An online colouring of a graph G is an assignment of colours to vertices $V$of G in some order $v_1, \ldots, v_n$ such that
  (i) the colour of v_i, $1 \leq i \leq k$, is assigned by only looking at the subgraph of G induced by the set $\{v_1, \ldots, v_i\}$, and
  (ii) the assigned colour of v_i is never changed.
- Minimum Online Colouring:
  Instance: $G = (V, E)$ and an ordering $v_1, \ldots, v_n$ of V .
  Solution: An online colouring of G using the given ordering.
  Measure: The number of colours used in the online colouring.
  Goal: Minimize the number of colours.
- We would like to have a way to analyze and compare algorithms that solve Minimum Online Colouring.

- Let $\mathbb{A}$ is an online colouring algorithm. Let $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$ be the colourings produced by $\mathbb{A}$ for each possible ordering of the vertices of G.

- Let $\mathbb{A}$ is an online colouring algorithm. Let $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$ be the colourings produced by $\mathbb{A}$ for each possible ordering of the vertices of G.
- Let $\chi_A(G)$ denote the maximum number of colours used among $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$, which measures the worst-case behaviour of $\mathbb{A}$ on G.

- Let $\mathbb{A}$ is an online colouring algorithm. Let $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$ be the colourings produced by $\mathbb{A}$ for each possible ordering of the vertices of G.
- Let $\chi_A(G)$ denote the maximum number of colours used among $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$, which measures the worst-case behaviour of $\mathbb{A}$ on G.
- The competitive ratio of an online graph colouring algorithm $\mathbb{A}$ for a class of graphs $\mathcal{C}$ is $max_{G \in \mathcal{C}} \{\chi_A(G)/\chi(G)\}$.

- Let $\mathbb{A}$ is an online colouring algorithm. Let $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$ be the colourings produced by $\mathbb{A}$ for each possible ordering of the vertices of G.
- Let $\chi_A(G)$ denote the maximum number of colours used among $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$, which measures the worst-case behaviour of $\mathbb{A}$ on G.
- The competitive ratio of an online graph colouring algorithm $\mathbb{A}$ for a class of graphs $\mathcal{C}$ is $max_{G \in \mathcal{C}}\{\chi_A(G)/\chi(G)\}$.
- Clearly, a small competitive ratio is preferred over a large one.

- Let $\mathbb{A}$ is an online colouring algorithm. Let $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$ be the colourings produced by $\mathbb{A}$ for each possible ordering of the vertices of G.
- Let $\chi_A(G)$ denote the maximum number of colours used among $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$, which measures the worst-case behaviour of $\mathbb{A}$ on G.
- The competitive ratio of an online graph colouring algorithm $\mathbb{A}$ for a class of graphs $\mathcal{C}$ is $max_{G \in \mathcal{C}}\{\chi_A(G)/\chi(G)\}$.
- Clearly, a small competitive ratio is preferred over a large one.
- To evaluate the efficiency of an online colouring algorithm, we find bounds on the competitive ratio of the algorithm.

- Let $\mathbb{A}$ is an online colouring algorithm. Let $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$ be the colourings produced by $\mathbb{A}$ for each possible ordering of the vertices of G.
- Let $\chi_A(G)$ denote the maximum number of colours used among $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$, which measures the worst-case behaviour of $\mathbb{A}$ on G.
- The competitive ratio of an online graph colouring algorithm $\mathbb{A}$ for a class of graphs $\mathcal{C}$ is $max_{G \in \mathcal{C}}\{\chi_A(G)/\chi(G)\}$.
- Clearly, a small competitive ratio is preferred over a large one.
- To evaluate the efficiency of an online colouring algorithm, we find bounds on the competitive ratio of the algorithm.
- On-line colouring can be viewed as a two-person game on a graph G.

- Let $\mathbb{A}$ is an online colouring algorithm. Let $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$ be the colourings produced by $\mathbb{A}$ for each possible ordering of the vertices of G.
- Let $\chi_A(G)$ denote the maximum number of colours used among $\{\mathbb{C}_1, \ldots, \mathbb{C}_k\}$, which measures the worst-case behaviour of $\mathbb{A}$ on G.
- The competitive ratio of an online graph colouring algorithm $\mathbb{A}$ for a class of graphs $\mathcal{C}$ is $max_{G \in \mathcal{C}}\{\chi_A(G)/\chi(G)\}$.
- Clearly, a small competitive ratio is preferred over a large one.
- To evaluate the efficiency of an online colouring algorithm, we find bounds on the competitive ratio of the algorithm.
- On-line colouring can be viewed as a two-person game on a graph G.
- In each step player I reveals the vertices of G and player II answers by immediately colouring the current vertex.

- The aim of II might be to use as few colors as possible and then the strategy of I against II consists in finding the "worst" order of vertices that forces as much color as possible.

- The aim of II might be to use as few colors as possible and then the strategy of I against II consists in finding the "worst" order of vertices that forces as much color as possible.
- Before we construct even a single online colouring algorithm, we can determine some upper and lower bounds on the competitive ratio of any online algorithm.

- The aim of II might be to use as few colors as possible and then the strategy of I against II consists in finding the "worst" order of vertices that forces as much color as possible.
- Before we construct even a single online colouring algorithm, we can determine some upper and lower bounds on the competitive ratio of any online algorithm.
- Trivially, we have an upper bound of n on the competitive ratio since any algorithm can assign at most one colour to each of the n vertices of the input graph.

- The aim of II might be to use as few colors as possible and then the strategy of I against II consists in finding the "worst" order of vertices that forces as much color as possible.
- Before we construct even a single online colouring algorithm, we can determine some upper and lower bounds on the competitive ratio of any online algorithm.
- Trivially, we have an upper bound of n on the competitive ratio since any algorithm can assign at most one colour to each of the n vertices of the input graph.
- Similarly, we can easily see that if the input graph is restricted to be from the class of complete graphs on n vertices, any online colouring algorithm yields an exact solution with competitive ratio of 1.

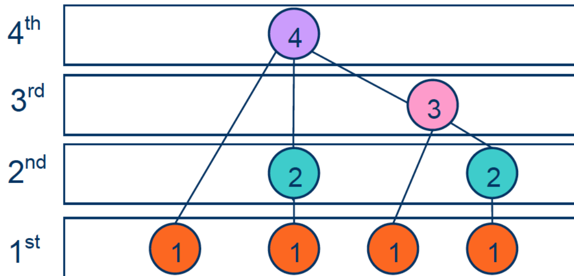
- The simplest on-line colouring is the *first fit algorithm* (denoted as FF), which is an online colouring that works by assigning the smallest possible integer as color to the current vertex of the graph.

- The simplest on-line colouring is the *first fit algorithm* (denoted as FF), which is an online colouring that works by assigning the smallest possible integer as color to the current vertex of the graph.
- Observe that if $\chi_{FF}(G) = k$, then FF produces a partition of $V = S_1 \cup S_2 \cup \ldots \cup S_k$, where S_i is a maximal nonempty independent set in the subgraph induced by $S_i \cup \ldots \cup S_k$, for every $i, 1 \leq i \leq k$.

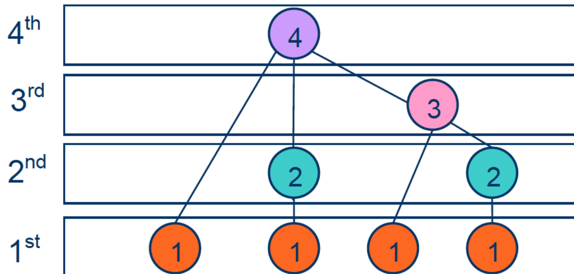
- The simplest on-line colouring is the *first fit algorithm* (denoted as FF), which is an online colouring that works by assigning the smallest possible integer as color to the current vertex of the graph.
- Observe that if $\chi_{FF}(G) = k$, then FF produces a partition of $V = S_1 \cup S_2 \cup \ldots \cup S_k$, where S_i is a maximal nonempty independent set in the subgraph induced by $S_i \cup \ldots \cup S_k$, for every $i, 1 \leq i \leq k$.
- Note that S_i represents i^{th}colour vertex set of V.

- The simplest on-line colouring is the *first fit algorithm* (denoted as FF), which is an online colouring that works by assigning the smallest possible integer as color to the current vertex of the graph.
- Observe that if $\chi_{FF}(G) = k$, then FF produces a partition of $V = S_1 \cup S_2 \cup \ldots \cup S_k$, where S_i is a maximal nonempty independent set in the subgraph induced by $S_i \cup \ldots \cup S_k$, for every $i, 1 \leq i \leq k$.
- Note that S_i represents i^{th}colour vertex set of V.
- It is interesting to note that the converse is also true: every maximal independent set partition of V can be reproduced by FF if an appropriate ordering of the vertices is taken.

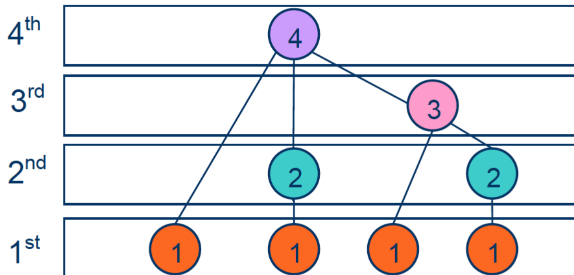
- The simplest on-line colouring is the *first fit algorithm* (denoted as FF), which is an online colouring that works by assigning the smallest possible integer as color to the current vertex of the graph.
- Observe that if $\chi_{FF}(G) = k$, then FF produces a partition of $V = S_1 \cup S_2 \cup \ldots \cup S_k$, where S_i is a maximal nonempty independent set in the subgraph induced by $S_i \cup \ldots \cup S_k$, for every $i, 1 \leq i \leq k$.
- Note that S_i represents i^{th}colour vertex set of V.
- It is interesting to note that the converse is also true: every maximal independent set partition of V can be reproduced by FF if an appropriate ordering of the vertices is taken.
- We first show that $\chi_{FF}(G)/\chi(G)$ is not bounded for trees. In fact, there is no bounded on-line algorithm for trees as shown in the following theorem.



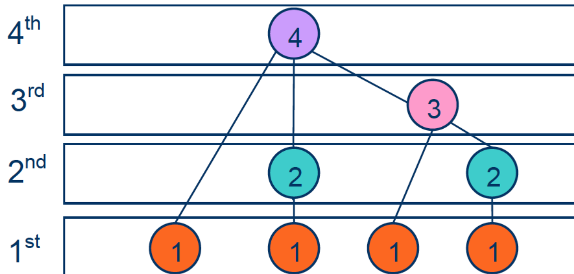
4th
3rd
2nd
1st
4
3
2
2
1
1
1
1



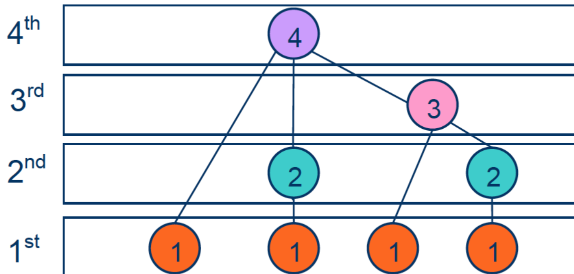
- **Theorem:** For every positive integer k, there exists a tree T_k of size $n = 2^{k-1}$ such that $\chi_{\mathbb{A}}(T_k) \geq k$ holds for every on-line algorithm $\mathbb{A}$.



- **Theorem:** For every positive integer k, there exists a tree T_k of size $n = 2^{k-1}$ such that $\chi_{\mathbb{A}}(T_k) \geq k$ holds for every on-line algorithm $\mathbb{A}$.
- In the proof, on-line colorings are viewed as a two-person game; the vertices of a graph are revealed by player I and player II colors the current vertex.



- **Theorem:** For every positive integer k, there exists a tree T_k of size $n = 2^{k-1}$ such that $\chi_{\mathbb{A}}(T_k) \geq k$ holds for every on-line algorithm $\mathbb{A}$.
- In the proof, on-line colorings are viewed as a two-person game; the vertices of a graph are revealed by player I and player II colors the current vertex.
- Suppose that I wins when II is forced to use at least k colors.



- **Theorem:** For every positive integer k, there exists a tree T_k of size $n = 2^{k-1}$ such that $\chi_{\mathbb{A}}(T_k) \geq k$ holds for every on-line algorithm $\mathbb{A}$.
- In the proof, on-line colorings are viewed as a two-person game; the vertices of a graph are revealed by player I and player II colors the current vertex.
- Suppose that I wins when II is forced to use at least k colors.
- We show a winning strategy for I by defining "winning" trees T_k for every $k = 1, 2, \ldots$.

- Assume that $T_1, T_2, \ldots, T_{k-1}$ have already been defined.

- Assume that $T_1, T_2, \ldots, T_{k-1}$ have already been defined.
- Let T'_j represents a copy of T_j for $1 \leq j \leq n-1$.

- Assume that $T_1, T_2, \ldots, T_{k-1}$ have already been defined.
- Let T_j' represents a copy of T_j for $1 \leq j \leq n-1$.
- Then T_k is formed as the union of $T_1', T_2', \ldots, T_{k-1}'$ plus a new vertex x joined to every root of $T_1', T_2', \ldots, T_{k-1}'$.

- Assume that $T_1, T_2, \ldots, T_{k-1}$ have already been defined.
- Let T'_j represents a copy of T_j for $1 \leq j \leq n-1$.
- Then T_k is formed as the union of $T'_1, T'_2, \ldots, T'_{k-1}$ plus a new vertex x joined to every root of $T'_1, T'_2, \ldots, T'_{k-1}$.
- Now we show how I has to play on T_k against II, who can apply an arbitrary on-line coloring algorithm $\mathbb{A}$.
- If I can obtain $k-1$ colours at the roots of $T'_1, T'_2, \ldots, T'_{k-1}$, then II is forced to use the kth colour at the root of T_k.

- Assume that $T_1, T_2, \ldots, T_{k-1}$ have already been defined.
- Let T'_j represents a copy of T_j for $1 \leq j \leq n-1$.
- Then T_k is formed as the union of $T'_1, T'_2, \ldots, T'_{k-1}$ plus a new vertex x joined to every root of $T'_1, T'_2, \ldots, T'_{k-1}$.
- Now we show how I has to play on T_k against II, who can apply an arbitrary on-line coloring algorithm $\mathbb{A}$.
- If I can obtain $k-1$ colours at the roots of $T'_1, T'_2, \ldots, T'_{k-1}$, then II is forced to use the kth colour at the root of T_k.
- By induction on j, I can obtain j colours at the roots of $T'_1, T'_2, \ldots, T'_j$ for $1 \leq j \leq k-1$.

- Assume that $T_1, T_2, \ldots, T_{k-1}$ have already been defined.
- Let T'_j represents a copy of T_j for $1 \leq j \leq n-1$.
- Then T_k is formed as the union of $T'_1, T'_2, \ldots, T'_{k-1}$ plus a new vertex x joined to every root of $T'_1, T'_2, \ldots, T'_{k-1}$.
- Now we show how I has to play on T_k against II, who can apply an arbitrary on-line coloring algorithm $\mathbb{A}$.
- If I can obtain $k-1$ colours at the roots of $T'_1, T'_2, \ldots, T'_{k-1}$, then II is forced to use the kth colour at the root of T_k.
- By induction on j, I can obtain j colours at the roots of $T'_1, T'_2, \ldots, T'_j$ for $1 \leq j \leq k-1$.
- Obtaining in this way $k-1$ distinct colors at the roots of $T'_1, T'_2, \ldots, T'_{k-1}$, I wins by revealing vertex x of T_k.

- Assume that $T_1, T_2, \ldots, T_{k-1}$ have already been defined.
- Let T'_j represents a copy of T_j for $1 \leq j \leq n-1$.
- Then T_k is formed as the union of $T'_1, T'_2, \ldots, T'_{k-1}$ plus a new vertex x joined to every root of $T'_1, T'_2, \ldots, T'_{k-1}$.
- Now we show how I has to play on T_k against II, who can apply an arbitrary on-line coloring algorithm $\mathbb{A}$.
- If I can obtain $k-1$ colours at the roots of $T'_1, T'_2, \ldots, T'_{k-1}$, then II is forced to use the kth colour at the root of T_k.
- By induction on j, I can obtain j colours at the roots of $T'_1, T'_2, \ldots, T'_j$ for $1 \leq j \leq k-1$.
- Obtaining in this way $k-1$ distinct colors at the roots of $T'_1, T'_2, \ldots, T'_{k-1}$, I wins by revealing vertex x of T_k.
- Therefore, $\chi_{\mathbb{A}}(T_k) \geq k$ holds for every on-line algorithm $\mathbb{A}$.

- Assume that $T_1, T_2, \ldots, T_{k-1}$ have already been defined.
- Let T'_j represents a copy of T_j for $1 \leq j \leq n-1$.
- Then T_k is formed as the union of $T'_1, T'_2, \ldots, T'_{k-1}$ plus a new vertex x joined to every root of $T'_1, T'_2, \ldots, T'_{k-1}$.
- Now we show how I has to play on T_k against II, who can apply an arbitrary on-line coloring algorithm $\mathbb{A}$.
- If I can obtain $k-1$ colours at the roots of $T'_1, T'_2, \ldots, T'_{k-1}$, then II is forced to use the kth colour at the root of T_k.
- By induction on j, I can obtain j colours at the roots of $T'_1, T'_2, \ldots, T'_j$ for $1 \leq j \leq k-1$.
- Obtaining in this way $k-1$ distinct colors at the roots of $T'_1, T'_2, \ldots, T'_{k-1}$, I wins by revealing vertex x of T_k.
- Therefore, $\chi_{\mathbb{A}}(T_k) \geq k$ holds for every on-line algorithm $\mathbb{A}$.
- Finally, since a tree can always be coloured by two colours, $\chi_{FF}(G)/\chi(G) = (\log n + 1)/2$ is not bounded for trees.

- The general story for FF is rather disappointing as stated in the following theorem.

- The general story for *FF* is rather disappointing as stated in the following theorem.
- **Theorem:** The competitive ratio of *FF* can be as bad as $n/4$.

- The general story for FF is rather disappointing as stated in the following theorem.
- **Theorem:** The competitive ratio of FF can be as bad as $n/4$.
- Consider a bipartite graph $G = (U, V, E)$ where $U = \{u_1, u_2, \ldots, u_t\}$, $V = \{v_1, v_2, \ldots, v_t\}$ and $E = \{u_i v_j : i \neq j$.

- The general story for FF is rather disappointing as stated in the following theorem.
- **Theorem:** The competitive ratio of FF can be as bad as $n/4$.
- Consider a bipartite graph $G = (U, V, E)$ where $U = \{u_1, u_2, \ldots, u_t\}$, $V = \{v_1, v_2, \ldots, v_t\}$ and $E = \{u_i v_j : i \neq j$.
- Let $\{u_1, v_1, u_2, v_2, \ldots, u_t, v_t\}$ be the given order of vertices to the first-fit algorithm.

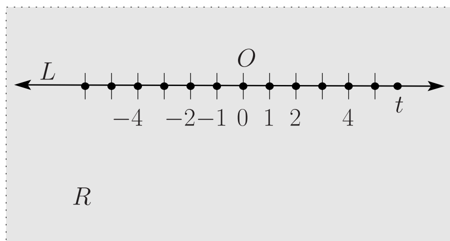
- The general story for FF is rather disappointing as stated in the following theorem.
- **Theorem:** The competitive ratio of FF can be as bad as $n/4$.
- Consider a bipartite graph $G = (U, V, E)$ where $U = \{u_1, u_2, \ldots, u_t\}$, $V = \{v_1, v_2, \ldots, v_t\}$ and $E = \{u_i v_j : i \neq j$.
- Let $\{u_1, v_1, u_2, v_2, \ldots, u_t, v_t\}$ be the given order of vertices to the first-fit algorithm.
- We show by induction on i that the colour i is assigned to both u_i and v_i, and thus FF uses $\{1, \ldots, t\}$ to colour G.

- The general story for FF is rather disappointing as stated in the following theorem.
- **Theorem:** The competitive ratio of FF can be as bad as $n/4$.
- Consider a bipartite graph $G = (U, V, E)$ where $U = \{u_1, u_2, \ldots, u_t\}$, $V = \{v_1, v_2, \ldots, v_t\}$ and $E = \{u_i v_j : i \neq j$.
- Let $\{u_1, v_1, u_2, v_2, \ldots, u_t, v_t\}$ be the given order of vertices to the first-fit algorithm.
- We show by induction on i that the colour i is assigned to both u_i and v_i, and thus FF uses $\{1, \ldots, t\}$ to colour G.
- FF assigns colour 1 to both u_1 and v_1 since they are not adjacent and 1 is the lowest colour available.

- The general story for FF is rather disappointing as stated in the following theorem.
- **Theorem:** The competitive ratio of FF can be as bad as $n/4$.
- Consider a bipartite graph $G = (U, V, E)$ where $U = \{u_1, u_2, \ldots, u_t\}$, $V = \{v_1, v_2, \ldots, v_t\}$ and $E = \{u_i v_j : i \neq j$.
- Let $\{u_1, v_1, u_2, v_2, \ldots, u_t, v_t\}$ be the given order of vertices to the first-fit algorithm.
- We show by induction on i that the colour i is assigned to both u_i and v_i, and thus FF uses $\{1, \ldots, t\}$ to colour G.
- FF assigns colour 1 to both u_1 and v_1 since they are not adjacent and 1 is the lowest colour available.

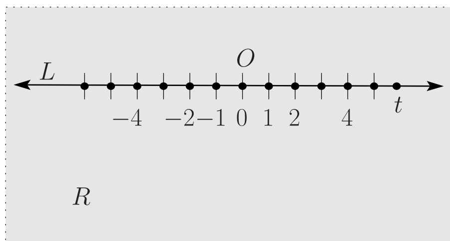
1. Avery Miller, *Online graph colouring*, Canadian Undergraduate Mathematics Conference, http://www.cumc.math.ca/2005/papers/miller.pdf, 2004.

Searching for a target in an unbounded region



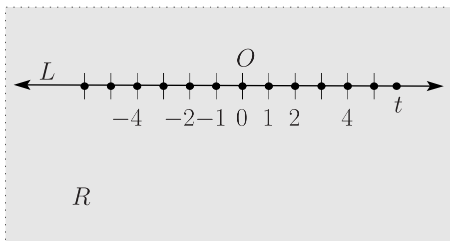
- Let L be a line in the plane.

Searching for a target in an unbounded region



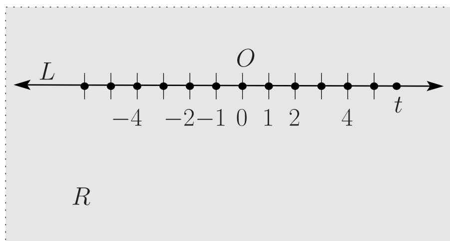
- Let L be a line in the plane.
- Assume that the target point t is placed in an unknown location on L.

Searching for a target in an unbounded region



- Let L be a line in the plane.
- Assume that the target point t is placed in an unknown location on L.
- Starting from a given position O on L, the problem is to design an online algorithm for a point robot for locating t.

Searching for a target in an unbounded region



- Let L be a line in the plane.
- Assume that the target point t is placed in an unknown location on L.
- Starting from a given position O on L, the problem is to design an online algorithm for a point robot for locating t.
- It is assumed that the robot can detect t if it stands on top of t or reaches t.

1. S. K. Ghosh and R. Klein, *Online algorithms for searching and exploration in the plane*, Computer Science Review, vol. 4, pp. 189-201, 2010.

- The problem may be viewed as an autonomous robot facing a very long wall and wishing to cross over to the other side of the wall through a door on the wall.

- The problem may be viewed as an autonomous robot facing a very long wall and wishing to cross over to the other side of the wall through a door on the wall.
- The wall has many doors spaced uniformly at equal distances, of which only one is open.

- The problem may be viewed as an autonomous robot facing a very long wall and wishing to cross over to the other side of the wall through a door on the wall.
- The wall has many doors spaced uniformly at equal distances, of which only one is open.
- This problem of searching a target point on a line is known as the *linear search problem*.

- The problem may be viewed as an autonomous robot facing a very long wall and wishing to cross over to the other side of the wall through a door on the wall.
- The wall has many doors spaced uniformly at equal distances, of which only one is open.
- This problem of searching a target point on a line is known as the *linear search problem*.
- The robot walks one unit to the right along L.

- The problem may be viewed as an autonomous robot facing a very long wall and wishing to cross over to the other side of the wall through a door on the wall.
- The wall has many doors spaced uniformly at equal distances, of which only one is open.
- This problem of searching a target point on a line is known as the *linear search problem*.
- The robot walks one unit to the right along L.
- If t is not found, then it returns to its starting point O.

- The problem may be viewed as an autonomous robot facing a very long wall and wishing to cross over to the other side of the wall through a door on the wall.
- The wall has many doors spaced uniformly at equal distances, of which only one is open.
- This problem of searching a target point on a line is known as the *linear search problem*.
- The robot walks one unit to the right along L.
- If t is not found, then it returns to its starting point O.
- In the next step, the robot walks two units to the left of O along L.

- The problem may be viewed as an autonomous robot facing a very long wall and wishing to cross over to the other side of the wall through a door on the wall.
- The wall has many doors spaced uniformly at equal distances, of which only one is open.
- This problem of searching a target point on a line is known as the *linear search problem*.
- The robot walks one unit to the right along L.
- If t is not found, then it returns to its starting point O.
- In the next step, the robot walks two units to the left of O along L.
- If t is not found again, the robot returns to O.

- The problem may be viewed as an autonomous robot facing a very long wall and wishing to cross over to the other side of the wall through a door on the wall.
- The wall has many doors spaced uniformly at equal distances, of which only one is open.
- This problem of searching a target point on a line is known as the *linear search problem*.
- The robot walks one unit to the right along L.
- If t is not found, then it returns to its starting point O.
- In the next step, the robot walks two units to the left of O along L.
- If t is not found again, the robot returns to O.
- In the next step, the robot walks four units to the right along L and if it is again unsuccessful in locating t, it returns to O.

- The problem may be viewed as an autonomous robot facing a very long wall and wishing to cross over to the other side of the wall through a door on the wall.
- The wall has many doors spaced uniformly at equal distances, of which only one is open.
- This problem of searching a target point on a line is known as the *linear search problem*.
- The robot walks one unit to the right along L.
- If t is not found, then it returns to its starting point O.
- In the next step, the robot walks two units to the left of O along L.
- If t is not found again, the robot returns to O.
- In the next step, the robot walks four units to the right along L and if it is again unsuccessful in locating t, it returns to O.
- Note that while walking four units to the right, it checks both locations at three and four for t, i.e., it does not skip any intermediate location for checking t.

- The process of doubling the length of the walk in each step and checking all the locations encountered along the walk is continued till t is located.

- The process of doubling the length of the walk in each step and checking all the locations encountered along the walk is continued till t is located.
- It has been shown that the robot walks at most 9 times the number of units between O and t on L, which gives 9 as the competitive ratio for their online algorithm.

- The process of doubling the length of the walk in each step and checking all the locations encountered along the walk is continued till t is located.
- It has been shown that the robot walks at most 9 times the number of units between O and t on L, which gives 9 as the competitive ratio for their online algorithm.
- Moreover, it has been proved that there is no competitive strategy with a ratio less than 9 proving that this online algorithm is optimal.

- The process of doubling the length of the walk in each step and checking all the locations encountered along the walk is continued till t is located.
- It has been shown that the robot walks at most 9 times the number of units between O and t on L, which gives 9 as the competitive ratio for their online algorithm.
- Moreover, it has been proved that there is no competitive strategy with a ratio less than 9 proving that this online algorithm is optimal.
- Let us explain how an alternate walk gives a competitive ratio of 9.

- The process of doubling the length of the walk in each step and checking all the locations encountered along the walk is continued till t is located.
- It has been shown that the robot walks at most 9 times the number of units between O and t on L, which gives 9 as the competitive ratio for their online algorithm.
- Moreover, it has been proved that there is no competitive strategy with a ratio less than 9 proving that this online algorithm is optimal.
- Let us explain how an alternate walk gives a competitive ratio of 9.
- Without loss of generality, assume that t is located at a distance d from the origin on the positive axis.

- The process of doubling the length of the walk in each step and checking all the locations encountered along the walk is continued till t is located.
- It has been shown that the robot walks at most 9 times the number of units between O and t on L, which gives 9 as the competitive ratio for their online algorithm.
- Moreover, it has been proved that there is no competitive strategy with a ratio less than 9 proving that this online algorithm is optimal.
- Let us explain how an alternate walk gives a competitive ratio of 9.
- Without loss of generality, assume that t is located at a distance d from the origin on the positive axis.
- Assume that $2^{k-1} < d \leq 2^k$ for some k.

- The process of doubling the length of the walk in each step and checking all the locations encountered along the walk is continued till t is located.
- It has been shown that the robot walks at most 9 times the number of units between O and t on L, which gives 9 as the competitive ratio for their online algorithm.
- Moreover, it has been proved that there is no competitive strategy with a ratio less than 9 proving that this online algorithm is optimal.
- Let us explain how an alternate walk gives a competitive ratio of 9.
- Without loss of generality, assume that t is located at a distance d from the origin on the positive axis.
- Assume that $2^{k-1} < d \leq 2^k$ for some k.
- So, the total distance traveled during the alternative walk is $(2.1 + 2.|-2| + 2.4 + 2.|-8| + \ldots + 2.2^{k-1} + 2.|-2^k| + d = 2.2^{k+1} + d)$.

- If the location of t is known a priori, then it is a straight walk of length d from the origin to t.

- If the location of t is known a priori, then it is a straight walk of length d from the origin to t.
- So, the competitive ratio of the alternate walk is $(2.2^{k+1} + d)/d = 1 + 2.2^{k+1}/d$ which is at most $1 + (2.2^{k+1}/2^{k-1}) = 9$.

- If the location of t is known a priori, then it is a straight walk of length d from the origin to t.
- So, the competitive ratio of the alternate walk is $(2.2^{k+1} + d)/d = 1 + 2.2^{k+1}/d$ which is at most $1 + (2.2^{k+1}/2^{k-1}) = 9$.
- Suppose the robot knows that t is located exactly at a distance d away from O.

- If the location of t is known a priori, then it is a straight walk of length d from the origin to t.
- So, the competitive ratio of the alternate walk is $(2.2^{k+1} + d)/d = 1 + 2.2^{k+1}/d$ which is at most $1 + (2.2^{k+1}/2^{k-1}) = 9$.
- Suppose the robot knows that t is located exactly at a distance d away from O.
- Then the robot first walks a distance d to the right.

- If the location of t is known a priori, then it is a straight walk of length d from the origin to t.
- So, the competitive ratio of the alternate walk is $(2.2^{k+1} + d)/d = 1 + 2.2^{k+1}/d$ which is at most $1 + (2.2^{k+1}/2^{k-1}) = 9$.
- Suppose the robot knows that t is located exactly at a distance d away from O.
- Then the robot first walks a distance d to the right.
- If t is not found, then the robot returns to O and then walks a distance d to the left.

- If the location of t is known a priori, then it is a straight walk of length d from the origin to t.
- So, the competitive ratio of the alternate walk is $(2.2^{k+1} + d)/d = 1 + 2.2^{k+1}/d$ which is at most $1 + (2.2^{k+1}/2^{k-1}) = 9$.
- Suppose the robot knows that t is located exactly at a distance d away from O.
- Then the robot first walks a distance d to the right.
- If t is not found, then the robot returns to O and then walks a distance d to the left.
- So, the competitive ratio of this straightforward on-line algorithm is 3.

- If the location of t is known a priori, then it is a straight walk of length d from the origin to t.
- So, the competitive ratio of the alternate walk is $(2.2^{k+1} + d)/d = 1 + 2.2^{k+1}/d$ which is at most $1 + (2.2^{k+1}/2^{k-1}) = 9$.
- Suppose the robot knows that t is located exactly at a distance d away from O.
- Then the robot first walks a distance d to the right.
- If t is not found, then the robot returns to O and then walks a distance d to the left.
- So, the competitive ratio of this straightforward on-line algorithm is 3.
- On the other hand, we know that the competitive ratio is 9 if d is not known apriori.

- If the location of t is known a priori, then it is a straight walk of length d from the origin to t.
- So, the competitive ratio of the alternate walk is $(2.2^{k+1} + d)/d = 1 + 2.2^{k+1}/d$ which is at most $1 + (2.2^{k+1}/2^{k-1}) = 9$.
- Suppose the robot knows that t is located exactly at a distance d away from O.
- Then the robot first walks a distance d to the right.
- If t is not found, then the robot returns to O and then walks a distance d to the left.
- So, the competitive ratio of this straightforward on-line algorithm is 3.
- On the other hand, we know that the competitive ratio is 9 if d is not known apriori.
- So, the competitive ratio of an online algorithm for this problem lies between 3 and 9 if robot knows that t is at least 1 unit and at most $d \leq r$ units away from O.