Winter Semester 2017/18

Online and Approximation Algorithms

http://www14.in.tum.de/lehre/2017WS/oa/index.html.en

Susanne Albers

Department of Informatics

TU München

0. Organizational matters

Lectures:	4 SWS	
	Mon	08:00–10:00, MI 00.13.009A
	Wed	08:00–10:00, MI 00.13.009A
Exercises:	2 SWS	
	Wed	10:00–12:00, 00.08.036

Teaching assistant: Jens Quedenfeld (jens.quedenfeld@in.tum.de)

Bonus: If at least 50% of the maximum number of points of the homework assignments are attained and student presents the solutions of at least two problems in the exercise sessions, then the grade of the final exam will be improved by 0.3 (or 0.4).

0. Organizational matters

Valuation: 8 ECTS (4 + 2 SWS)

Office hours: by appointment (albers@in.tum.de)

0. Organizational matters

Problem sets: Made available on Monday by 10:00
on the course webpage.
Must be turned in one week later before the lecture.

Exam: Written exam; no auxiliary means are permitted, except for one hand-written sheet of paper.

Prerequisites: Grundlagen: Algorithmen und Datenstrukturen (GAD)
Diskrete Wahrscheinlichkeitstheorie (DWT)

Effiziente Algorithmen und Datenstrukturen
(advantageous but not required)

0. Literature

- [BY] A. Borodin und R. El-Yaniv. Online Computation and Competitive Analysis. Cambridge University Press, Cambridge, 1998. ISBN 0-521-56392-5

- [V] V.V. Vazirani. Approximation Algorithms. Springer Verlag, Berlin, 2001. ISBN 3-540-65367-8

0. Content

Online and approximation algorithms

Optimization problems for which the computation of an optimal solution is hard or impossible.

Have to resort to approximations:
Design algorithms with a provably good performance.

0. Content

Online algorithms

- Scheduling
- Paging
- List update
- Randomization
- Data compression
- Robotics
- Matching

0. Content

Approximation algorithms

- Traveling Saleman Problem
- Knapsack Problem
- Scheduling (makespan minimization)
- SAT (Satisfiability)
- Set Cover
- Hitting Set
- Shortest Superstring

1. Introduction

Online and approximation algorithms

Optimization problems for which the computation of an optimal solution is hard or impossible.

Have to resort to approximations:
Design algorithms with a provably good performance.

1.1 Online problems

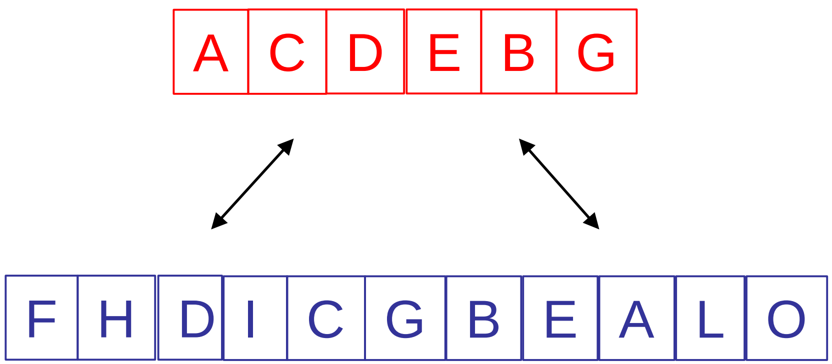
Relevant input arrives incrementally over time. Online algorithm has to make decisions without knowledge of any future input.

1. Ski rental problem: Student wishes to pick up the sport of skiing.
   Renting equipment: 10$ per season
   Buying equipment: 100$
   Do not know how long (how many seasons) the student will enjoy skiing.

2. Currency conversion: Wish to convert 1000$ into Yen over a certain time horizon.

1.1 Online problems

3. Paging/caching: Two-level memory system



small fast memory

large slow memory

σ = A C B E D A F …

Request: Access to page in memory system

Page fault: requested page not in fast memory; must be loaded into fast memory

Goal: Minimize the number of page faults

1.1 Online problems

4. Data structures: List update problem

Unsorted linear list

L: B $\rightarrow$ C $\rightarrow$ E $\rightarrow$ A $\rightarrow$ D

σ = A A C B E D A …

Request: Access to item in the list

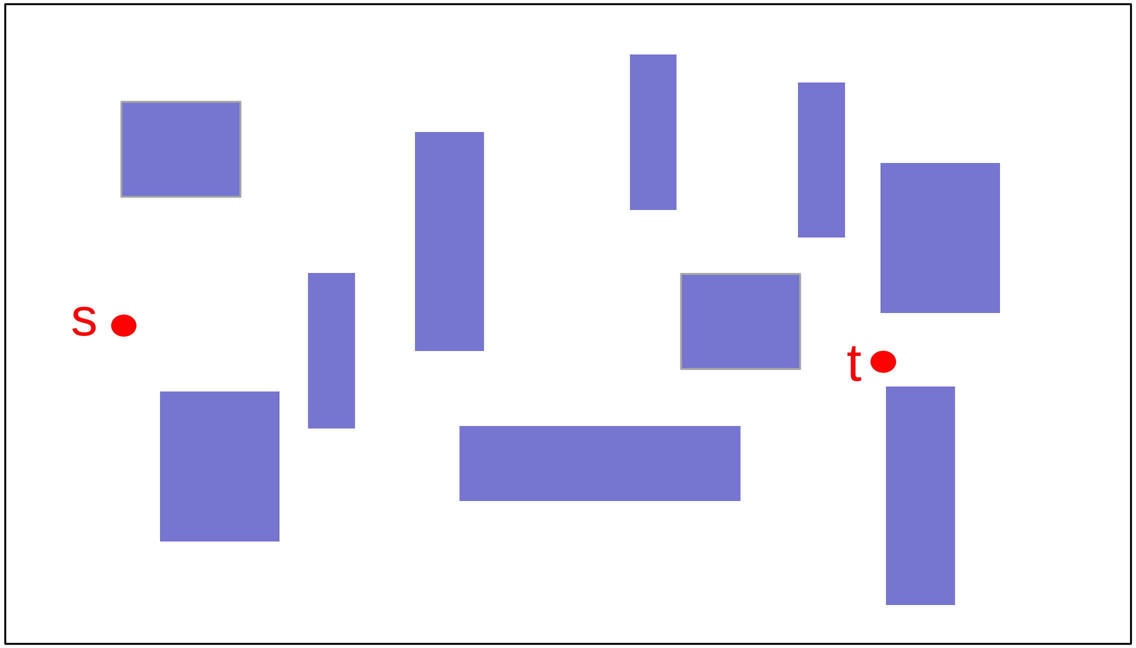
Cost: Accessing the i-th item in the list incurs a cost of i.

Rearrangements: After an access, requested item may be moved at no extra cost to any position closer to the front of the list (free exchanges). At any time two adjacent items may be exchanged at a cost of 1 (paid exchange).

Goal: Minimize cost paid in serving σ.

1.1 Online problems

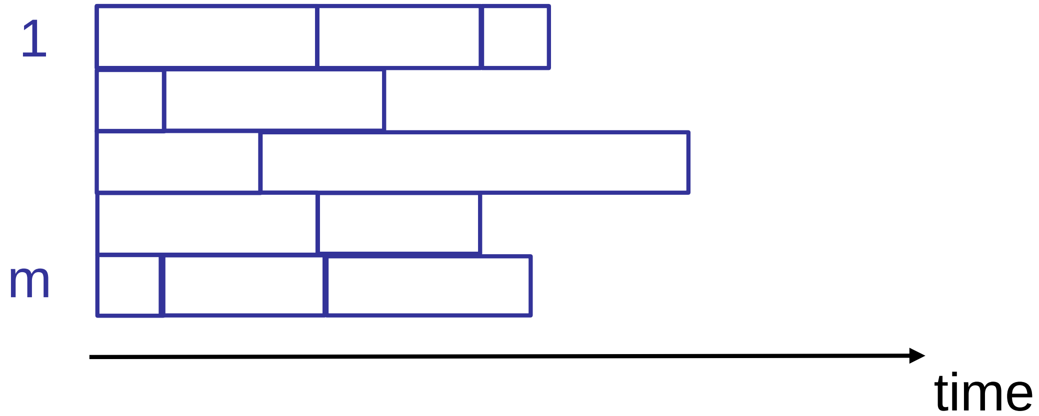
5. Robotics: Navigation



Unknown scene: Robot has to find a short path from s to t.

1.1 Online problems

6. Scheduling: Makespan minimization



m identical parallel machines

Input portion: Job J_i with individual processing time p_i

Goal: Minimize the completion time of the last job in the schedule.

1.2 NP-hard optimization problems

Assuming P ≠ NP, NP-hard optimization problems cannot be solved optimally in polynomial time.

1. Scheduling: Makespan minimization (see above)

   Entire job sequence is known in advance. Famous optimization problem studied by Ronald Graham in 1966.

2. Traveling Salesman Problem: n cities, $c(i,j)$ = cost/distance to travel from city i to city j, $1 \leq i,j \leq n$.

   Goal: Find tour that visits each city exactly once and minimizes the total cost.

1.2 NP-hard optimization problems

3. Knapsack Problem: n items with individual weights $w_1, \ldots, w_n \in \mathbb{N}$ and values $a_1, \ldots, a_n \in \mathbb{N}$. Knapsack of total weight (capacity) W.

   Goal: Find a feasible packing, i.e. a subset of the items whose total weight does not exceed W, that maximizes the value obtained.

4. Max SAT: n Boolean variables $\{x_1, \ldots, x_n\}$ with associated literals $\{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$ and clauses $C_1, \ldots, C_m$. Each clause is a disjunction of literals.

   Goal: Find an assignment of the variables that maximizes the number of satisfied clauses.

1.2 NP-hard optimization problems

5. Shortest Superstring: Finite alphabet Σ, n strings $\{s_1,\ldots, s_n\} \subseteq \Sigma^+$.
Goal: Find shortest string that contains all s_i as substring.

2. Online algorithms

Formal model:

A

Online algorithm has to serve σ.

$$\sigma = \sigma(1)\ \sigma(2)\ \sigma(3)\ \ldots.\ \sigma(t)\ \sigma(t+1)\ \ldots$$

Each request σ(t) has to be served without knowledge of any future requests.

Goal: Optimize a desired objective, typically the total cost incurred in serving σ.

2. Competitive analysis

Online algorithm A is compared to an optimal offline algorithm OPT that knows the entire input σ in advance and can serve it optimally, with minimum cost.

$$A \longleftrightarrow OPT$$

Cost: $A(\sigma)$ $OPT(\sigma)$

Online algorithm A is called c-competitive if there exists a constant a, which is independent of σ, such that

$$A(\sigma) \leq c \cdot OPT(\sigma) + a$$

holds for all σ.

2.1 Scheduling

Makespan minimization: m identical parallel machines.

n jobs $J_1,\ldots, J_n$. p_t = processing time of J_t , $1 \leq t \leq n$

Goal: Minimize the makespan

Algorithm Greedy: Schedule each job on the machine currently having the smallest load.

Algorithm is also referred to as *List Scheduling*.

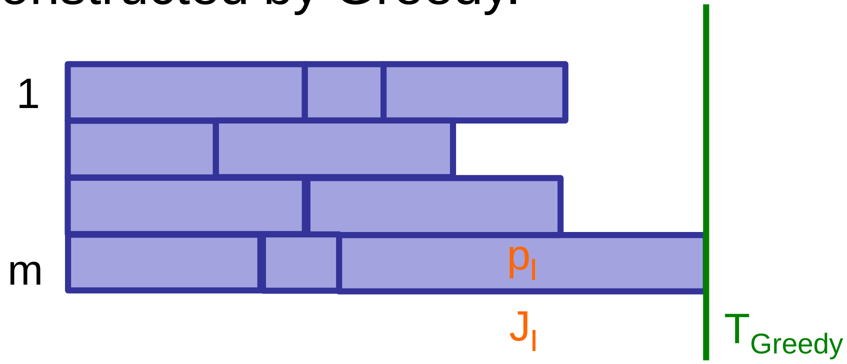
Theorem: Greedy is (2-1/m)-competitive.

Theorem: The competitive ratio of Greedy is not smaller than 2-1/m.

2.1 Scheduling

Theorem: Greedy is (2-1/m)-competitive.

Proof: Given an arbitrary job sequence $\sigma = J_1,\ldots,J_n$, consider the schedule constructed by Greedy.



T_{Greedy}: makespan Greedy

T_{OPT} : makespan OPT

Let J_l be the job that finishes last. At the time of assignment J_l was placed on a least loaded machine. This implies that the idle time on any machine is upper bounded by p_l.

$$mT_{Greedy} \leq \sum_{1\leq i\leq n} p_i + (m-1)\,p_l \leq \sum_{1\leq i\leq n} p_i + (m-1)\max_{1\leq i\leq n} p_i$$

2.1 Scheduling

It follows

$$T_{Greedy} \le \frac{1}{m}\sum_{1\le i\le n} p_i + \left(1-\frac{1}{m}\right)\max_{1\le i\le n} p_i \le \left(2-\frac{1}{m}\right)T_{OPT}.$$

The last inequality holds because of the following facts.

- $1/m \cdot \sum_{1\le i\le n} p_i \le T_{OPT}$: Even if OPT can distribute all jobs evenly among the machines, its makespan cannot be smaller that the average machine load.
- $\max_{1\le i\le n} p_i \le T_{OPT}$: The largest job must be placed (as a whole) on one of the machines.

Theorem: The competitive ratio of Greedy is not smaller than 2-1/m.

Proof: Consider the following job sequence.

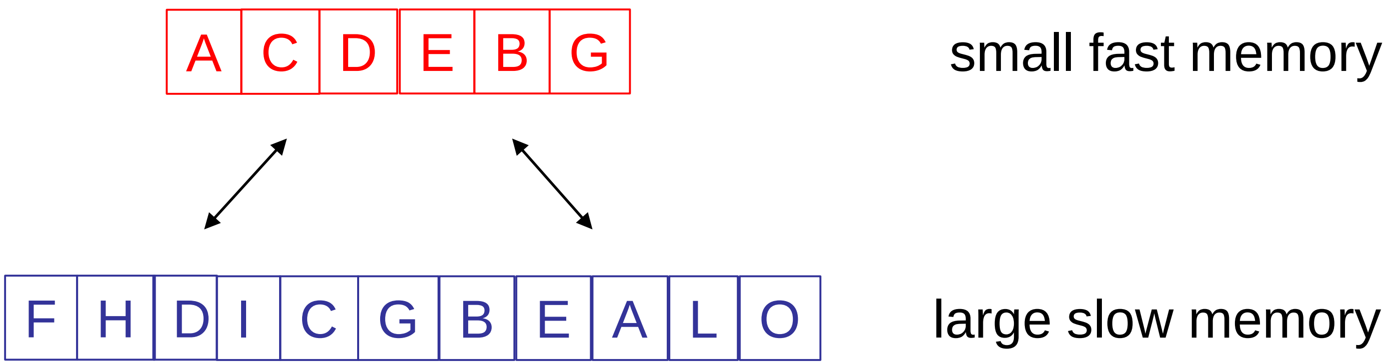
σ = m(m-1) jobs of processing time 1, followed by one job of processing time m

T_{Greedy} = m-1+ m = 2m-1

T_{OPT} = m

2.2 Paging

Two-level memory system



$\sigma =$ A C B E D A F …

Request: Access to page in memory system

Page fault: requested page not in fast memory; must be loaded into fast memory

Goal: Minimize the number of page faults

2.2 Paging

Popular online algorithms

- LRU (Least Recently Used): On a page fault evict the page from fast memory that has been requested least recently.
- FIFO (First-In First-Out): Evict the page that has been in fast memory longest.

Let k be the number of pages that can simultaneously reside in fast memory.

Theorem: LRU and FIFO are k-competitive.

Theorem: Let A be a deterministic online paging algorithm. If A is c-competitive, then $c \geq k$.

2.2 Paging

Theorem: LRU is k-competitive.

Proof: W.l.o.g. LRU and OPT start with the same configuration in fast memory.

Let $\sigma = \sigma(1),\dots,\sigma(m)$ be an arbitrary request sequence.

We will show $LRU(\sigma) \le k \cdot OPT(\sigma)$.

Partition σ into phases $P(1)$, $P(2)$, $P(3)$, … such that LRU generates

- at most k page faults in $P(1)$
- exactly k page faults in each $P(r)$, $r \ge 2$.

Such a partitioning can be obtained by traversing σ backwards. Whenever k faults by LRU have been encountered, a phase is cut off.

We will show that in each phase OPT has at least one page fault. This establishes $LRU(\sigma) \le k \cdot OPT(\sigma)$.

2.2 Paging

First consider P(1). The first page fault by LRU is also a fault for OPT because both algorithms start with the same set of pages in fast memory.

In the remainder we concentrate of any P(r), where $r \geq 2$.

Let $p_1, \ldots, p_k$ denote the k pages/requests where LRU has a fault.

Let q be the page referenced last in the preceding phase P(r-1).

P(r) : $q \mid p_1 \; \ldots \; p_2 \; \ldots \; p_k \mid$

Case 1: $p_i \neq p_j$, for all $i \neq j$, and $p_i \neq q$, for all i

At the end of P(r-1) page q is in OPT‘s fast memory. At that time the k distinct pages $p_1, \ldots, p_k$, which are different from q, cannot all reside in OPT’s fast memory so that OPT must incur at least one fault in P(r).

2.2 Paging

Case 2: $p_i = p_j$, for a pair i, j with $i \neq j$

P(r) : $q \mid \ \dots\ p_i\ \dots\ p_l\ \dots\ p_i\ \dots\ \mid$

p_i is evicted

LRU faults twice on p_i during P(r). Hence page p_i must be evicted from LRU's fast memory on a request to some page p_l. At that time p_i is the least recently requested page in fast memory. Thus, since the last reference to p_i exactly k-1 distinct pages were requested. These pages are different from p_i and p_l. We conclude that P(r) contains requests to k+1 distinct pages so that OPT must incur at least one fault.

2.2 Paging

Case 3: $p_i \neq p_j$, for all $i \neq j$, but $p_i = q$, for some i

P(r) : q | … p_l … q … |

q is evicted

LRU incurs a fault on q during P(r). Hence q must be evicted from LRU's fast memory on a request to some page p_l. At that time q is the least recently requested page in fast memory. Hence, since the beginning of the phase exactly k-1 distinct pages different from q and p_l were requested. Therefore P(r) contains requests to k+1 distinct pages, and OPT must incur at least one fault.

2.2 Paging

Theorem: Let A be a deterministic online paging algorithm. If A is c-competitive, then $c \geq k$.

Proof: Let $S=\{p_1,\ldots,p_{k+1}\}$ be a set of k+1 pages.

At any time exactly one page does not reside in fast memory.

Adversary: Always requests the page not available in A's fast memory.

$A(\sigma) = m$ $\quad$ m = length of σ

OPT can serve σ so that it incurs at most one fault on any k consecutive requests: Whenever OPT has a fault on a reference $\sigma(t)=p^*$, all pages of $S\setminus\{p^*\}$ are in fast memory. OPT can evict a page not needed during the next k-1 references.

2.2 Paging

Marking algorithms: Serve a request sequence in phases. First phase starts with the first request. Any other phase starts with the first request following the end of the previous phase.

At the beginning of a phase all pages are unmarked. Whenever a page is requested, it is marked. On a fault evict an arbitrary unmarked page in fast memory. If no such page is available, the phase ends and all marks are erased.

Flush-When-Full: If there is a page fault and there is no empty slot in fast memory, evict all pages.

2.2 Paging

Offline algorithm

- MIN: On a page fault evict the page whose next request is farthest in the future.

Theorem: MIN is an optimal offline algorithm for the paging problem, i.e. it achieves the smallest number of page faults/page replacements.

2.2 Paging

An algorithm is a *demand paging* algorithm if it only replaces a page in fast memory when there is a page fault.

Fact: Any paging algorithm can be turned into a demand paging algorithm such that, for any request sequence, the number of memory replacements does not increase.

2.2 Paging

Theorem: MIN is an optimal offline algorithm for the paging problem, i.e. it achieves the smallest number of page faults/page replacements.

Proof: Let σ be an arbitrary request sequence of length m.

Let A be an algorithm that serves σ with the minimum number of faults/page replacements. W.l.o.g. A is a demand paging algorithm.

Claim: Suppose that A and MIN serve the first i-1 requests identically but the i-th request differently, $1 \leq i \leq m$. We can transform A into an algorithm A' such that

- A', MIN serve the first i requests identically
- $A'(\sigma) \leq A(\sigma)$ and A' is again a demand paging algorithm.

2.2 Paging

Theorem follows by repeatedly applying the claim. Specifically, let A^i be the algorithm obtained from A after i steps of the transformation, i.e. A^i and MIN serve the first i requests identically. The claim ensures

$$A(\sigma) \geq A^1(\sigma) \geq A^2(\sigma) \geq \ldots \geq A^m(\sigma) = MIN(\sigma).$$

It remains to prove the claim. Consider the i-th request. Since A and MIN are demand paging algorithms, there is a fault on the i-th request. Let x be the referenced page.

Suppose that A evicts u while MIN evicts v.

Definition of A': It serves the first i-1 requests as A and MIN. On the i-th request it evicts v. Then A' simulates A until one of the following two events occurs.

2.2 Paging

1. A evicts v on a fault to page y. In this event A' evicts u.

2. Page u is requested an A evicts z. In this case A' loads v.

In each of the two events, the fast memories of A and A' are identical. Thereafter, A' works the same way as A.

By the MIN policy, Event 2 occurs before v is requested.

A' performs the same number of memory replacements as A. Finally, A' can be transformed into a demand paging algorithm without increasing the number of memory replacements (see Exercises).

2.3 Amortized analysis

General concept to analyze the cost of a sequence of operations executed, for instance, on a data structure.

Wish to show: An individual operation can be expensive, but the average cost of an operation is small.

Amortization: Distribute cost of a sequence of operations properly among the operations.

Example: Binary counter with increment operation. Cost of an operation is equal to the number of bit flips.

2.3 Amortized analysis, binary counter

Operation	Counter value	Cost
	00000	
1	00001	1
2	00010	2
3	00011	1
4	00100	3
5	00101	1
6	00110	2
7	00111	
8	01000	
9	01001	
10	01010	
11	01011	
12	01100	
13	01101	

2.3 Amortized analysis

Potential function technique

$$\Phi : \text{Config } D \to \mathbb{R}$$

It will be convenient if $\Phi(t) \geq 0$ and $\Phi(0) = 0$

Actual cost of operation t: $a(t)$

Amortized cost of operation t: $a(t) + \Phi(t) - \Phi(t-1)$

The goal is to show that for all t:

$$a(t) + \Phi(t) - \Phi(t-1) \leq c$$

2.3 Amortized competitive analysis

Given σ, wish to show $A(\sigma) \leq c \cdot OPT(\sigma)$

Potential: Φ : (Config A, Config OPT) $\rightarrow \mathbb{R}$

Again, it will be convenient if $\Phi(t) \geq 0$ and $\Phi(0) = 0$

A's actual cost of operation t: $A(t)$

A's amortized cost of operation t: $A(t) + \Phi(t) - \Phi(t-1)$

OPT's actual cost of operation t: $OPT(t)$

The goal is to show that for all t:

$$A(t) + \Phi(t) - \Phi(t-1) \leq c \cdot OPT(t)$$

2.3 Amortized competitive analysis

Summing the last inequality over all t, for a request sequence σ of length m, we obtain

$$\Sigma_{1\le t\le m} (A(t) + \Phi(t) - \Phi(t-1)) \le c \cdot \Sigma_{1\le t\le m} OPT(t),$$

which is equivalent to

$$A(\sigma) + \Phi(m) - \Phi(0) \le c \cdot OPT(\sigma)$$

and

$$A(\sigma) \le c \cdot OPT(\sigma) - \Phi(m) + \Phi(0).$$

If the potential is non-negative and initially zero, we obtain as desired

$$A(\sigma) \le c \cdot OPT(\sigma).$$

2.4 List update problem

Unsorted, linear linked list of items

L: B → C → E → A → D

$\sigma =$ A A C B E D A ...

Request: Access to item in the list

Cost: Accessing the i-th item in the list incurs a cost of i.

Rearrangements: After an access, requested item may be moved at no extra cost to any position closer to the front of the list (free exchanges). At any time two adjacent items may be exchanged at a cost of 1 (paid exchange).

Goal: Minimize cost paid in serving σ.

2.4 List update problem

Online algorithms

- Move-To-Front (MTF): Move requested item to the front of the list.
- Transpose: Exchange requested item with immediate predecessor in the list.
- Frequency Count: Store a frequency counter for each item in the list. Whenever an item is requested, increase its counter by one. Always maintain the items of the list in order of non-increasing counter values.

Theorem: MTF is 2-competitive.

Theorem: Transpose and Frequency Count are not c-competitive, for any constant c.

Theorem: Let A be a deterministic list update algorithm. If A is c-competitive, for all list lengths, then $c \geq 2$.

2.4 List update problem

Theorem: MTF is 2-competitive.

Proof: Φ = # inversions

inversion: ordered pair (x,y) of items such that

x before y in OPT's list

x behind y in MTF's list

$\Phi \geq 0$ $\Phi(0) = 0$ if initial lists of OPT and MTF are identical

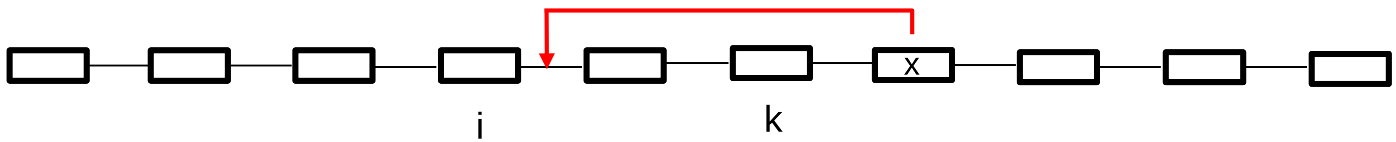
Consider an arbitrary request sequence $\sigma = \sigma(1),\ldots,\sigma(m)$.

Let $\sigma(t)$ be an arbitrary request, and let x denote the requested item.
We will analyze MTF's amortized cost on $\sigma(t)$ and prove

$$MTF(\sigma(t)) + \Phi(t) - \Phi(t-1) \leq 2\, OPT(\sigma(t)).$$

2.4 List update problem

1. Analysis of OPT‘s moves



Assume that in OPT‘s list, item x is stored at position k+1. The cost incurred in accessing the item is k+1.

Suppose that after the access, OPT inserts x behind the i-th item in the list, where i≤k. For each item y that is passed, an inversion (x,y) can be created. Since k-i items are being passed, the potential can increase by at most k-i during these item swaps.

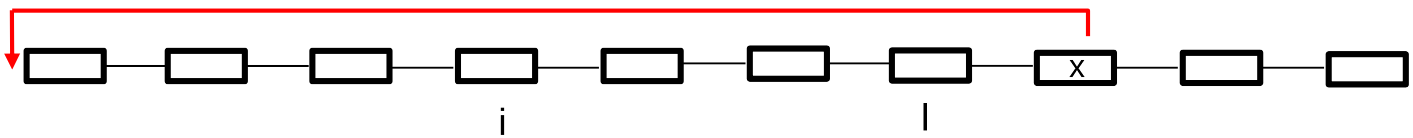
Finally, OPT may perform a number of, say, p(t) paid exchanges during the service of σ(t). Again, for each item swap, an inversion can be created such that the potential may increase by at most p(t).

Actual cost of OPT on σ(t) = k+1+p(t)

ΔΦ due to OPT‘s moves ≤ k-i+p(t)

2.4 List update problem

2. Analyis of MTF‘s moves



Assume that in MTF‘s list, item x is stored at position $l+1$. The cost incurred in accessing the item is $l+1$.

Case 1: $l \geq i$

Since $l \geq i$ there must exist at least $l-i$ items y_j that are stored before x in MTF‘s list but behind x in OPT‘s list. Hence there exist at least $l-i$ inversions of the form (x,y_j). When MTF moves x to the front of the list, each of these inversions is destroyed (potential drop by at least $l-i$). At the same time, for each of the first i items in OPT‘s list, an inversion can be created (potential increase by at most i).

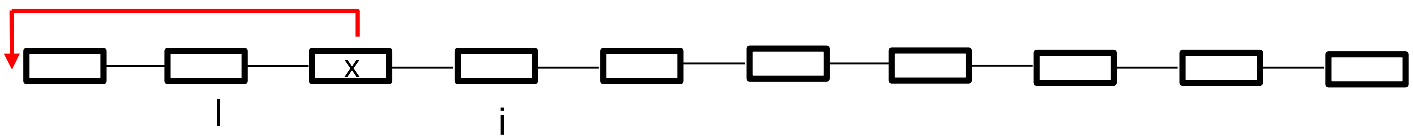
Actual cost of MTF on $\sigma(t) = l+1$

$\Delta\Phi$ due to MTF‘s moves $\leq -(l-i)+i$

$MTF(\sigma(t))+\Delta\Phi \leq l+1+(k-i)+p(t)-(l-i)+i = k+i+1+p(t) \leq 2(k+1+p(t)) = 2 \cdot OPT(\sigma(t))$

2.4 List update problem

2. Analyis of MTF‘s moves



Case 2: $l < i$

When MTF moves x to the front of the list, for each of the l items being passed, an inversion can be created (potential increase by at most l).

Actual cost of MTF on $\sigma(t)$ $= l+1$

$\Delta\Phi$ due to MTF‘s moves $\leq l$

$MTF(\sigma(t))+\Delta\Phi \leq l+1+(k-i)+p(t)+l \leq l+1+k+p(t) \leq 2(k+1+p(t)) = 2 \cdot OPT(\sigma(t))$

The second inequality holds because $l<i$; the third one holds since $l<i\leq k$.

2.4 List update problem

Theorem: Let A be a deterministic list update algorithm. If A is c-competitive, for all list lengths, then $c \geq 2$.

Proof: Let n be the number of items in the list.

Adversary: Always requests last item in A‘s list.

$A(\sigma) = m \cdot n$ m = length of the constructed σ

Let m be an integer multiple of n.

OPT: In order to serve σ, OPT maintains a static list of the items, sorted in order of non-increasing request frequencies. At most $\binom{n}{2}$ paid exchanges are needed to bring the initial list into this fixed static ordering. Let m_i denote the number of requests to the i-th item in the list. There holds $m_1 \geq m_2 \geq \ldots \geq m_n$.

$OPT(\sigma) \leq STAT(\sigma) \leq \Sigma_{1\leq i\leq n}\, i\cdot m_i + \binom{n}{2} \leq \Sigma_{1\leq i\leq n}\, i\cdot(m/n) + \binom{n}{2}$

2.4 List update problem

In order to verify the last inequality, observe that one can balance the request frequencies without decreasing the cost: While there exists an $m_i > m/n$ and an $m_j < m/n$, where $i < j$, we can decrease m_i by 1 and increase m_j by 1. This strictly increases the service cost. Thus

$$\mathrm{OPT}(\sigma) \le \Sigma_{1\le i\le n}\, i\cdot(m/n) + \binom{n}{2} = (n+1)m/2 + n(n-1)/2.$$

The cost ratio $c = A(\sigma) / \mathrm{OPT}(\sigma)$ satisfies

$$c \ge \frac{mn}{m(n+1)/2 + n(n-1)/2} = \frac{2n}{n+1+n(n-1)/m}$$

and the latter ratio tends to $2n/(n+1) = 2-2/(n+1)$ as m goes to infinity.

2.4 List update problem

Theorem: Transpose and Frequency Count are not c-competitive, for any constant c.

Proof: Transpose:

Always request the last item in Transpose's list. Only two items are referenced in turn. Let m be the length of the generated request sequence σ and assume that m is even.

Transpose(σ) = mn n = list length

OPT will move the two items ever referenced to the front of the list when they are first requested. They remain a positions 1 and 2, respectively.

$OPT(\sigma) \leq 2n + 1\cdot(m-2)/2 + 2\cdot(m-2)/2 = 2n + (m-2)\cdot 3/2$

The cost ratio tends to 2n/3 as m goes to infinity.

2.4 List update problem

Frequency Count (FC):

Let $x_1, x_2, \ldots, x_n$ be the order of the items in FC's initial list. Let k>n.

σ consists of k+1-i requests to x_i, for i=1,…,n.

$$FC(\sigma) = \sum_{i=1}^{n} i(k+1-i) = \frac{kn(n+1)}{2} + \frac{n(n+1)}{2} - \frac{n(n+1)(2n+1)}{6}$$

$$= \frac{kn(n+1)}{2} + \frac{n(n+1)(1-n)}{3} = \frac{kn(n+1)}{2} + \frac{n(1-n^2)}{3}$$

OPT can serve σ using the MTF algorithm. In this case the first request to an item x_i costs at most n, while the remaining requests to x_i can be served at a cost of 1 each.

Hence MTF(σ) ≤ $\Sigma_{1 \le i \le n}$ (n+k-i) =n(n+k) – n(n+1)/2. We obtain

$$\frac{FC(\sigma)}{OPT(\sigma)} \geq \frac{k(n+1)/2 + (1-n^2)/3}{(n+k)-(n+1)/2}$$

and the latter ratio tends to (n+1)/2 as k tends to infinity.

2.5 Randomized online algorithms

A = randomized online algorithm

A(σ) random variable, for any σ

Competitive ratio of A defined w.r.t. an adversary ADV who

- generates σ
- also serves σ

ADV knows the description of A

Critical question: Does ADV know the outcome of the random choices made by A?

2.5 Randomized online algorithms

Oblivious adversary:

$$\boxed{\text{A}}$$

$$\downarrow$$

$$\sigma = \underbrace{\sigma(1)\ \sigma(2)\ \sigma(3)\ \ldots.\ \sigma(m)}_{\boxed{\text{ADV}}}$$

Does not know the outcome of the random choices made by A.
Generates the entire σ in advance.

2.5 Randomized online algorithms

Adaptive adversary:

A

$\sigma = \sigma(1)\ \sigma(2)\ \sigma(3)\ \ldots.\ \sigma(t\text{-}1)$

$\sigma(t)$

ADV

Does know the outcome of the random choices made by A on the first t-1 requests when generating $\sigma(t)$.

Adaptive online adversary: Serves σ online.

Adaptive offline adversary: Serves σ offline.

2.5 Three types of adversaries

Oblivious adversary: Does not know the outcome of A‘s random choices; serves σ offline. A is c-competitive against oblivious adversaries, if there exists a constant a such that

$$E[A(\sigma)] \leq c \cdot ADV(\sigma) + a$$

holds for all σ generated by oblivious adversaries. Constant a must be independent of input σ.

Adaptive online adversary: Knows the outcome of A‘s random choices on first t-1 requests when generating σ(t); serves σ online. A is c-competitive against adaptive online adversaries, if there exists a constant a such that

$$E[A(\sigma)] \leq c \cdot E[ADV(\sigma)] + a$$

holds for all σ generated by adaptive online adversaries. Constant a must be independent of input σ.

2.5 Three types of adversaries

Adaptive offline adversary: Knows the outcome of A's random choices on first t-1 requests when generating σ(t); serves σ offline. A is c-competitive against adaptive offline adversaries, if there exists a constant a such that

$$E[A(\sigma)] \leq c \cdot E[OPT(\sigma)] + a$$

holds for all σ generated by adaptive offline adversaries. Constant a must independent of input σ.

2.5 Relating the adversaries

Theorem: If there exists a randomized online algorithm that is c-competitive against adaptive offline adversaries, then there also exists a c-competitive deterministic online algorithm.

Theorem: If A is c-competitive against adaptive online adversaries and there exists a d-competitive algorithm against oblivious adversaries, then there exists a cd-competitive algorithm against adpative offline adversaries.

Corollary: If A is c-competitive against adaptive online adversaries, then there exists a c^2-competitive deterministic algorithm.

2.6 Randomized paging

Algorithm RMARK: Serve σ in phases.

- At the beginning of a phase all pages are unmarked.
- Whenever a page is requested, it is marked.
- On a page fault, choose a page uniformly at random from among the unmarked pages in fast memory and evict it.

A phase ends when there is a page fault and the fast memory only contains marked pages. Then all marks are erased and a new phase is started (first request is the missing page that generated the fault).

Theorem: RMARK is $2H_k$-competitive against oblivious adversaries.

Here $H_k = \sum_{i=1}^{k} 1/i$ is the k-th Harmonic number.

There holds $\ln(k+1) \leq H_k \leq \ln k + 1$

2.6 Randomized paging

Theorem: RMARK is $2H_k$-competitive against oblivious adversaries.

Proof: Consider an arbitrary request sequence σ and assume that the initial fast memory is empty.

Suppose that RMARK generates phases P(1),…,P(l).

For each P(i) the following two properties hold.

- The phase contains requests to k distinct pages.
- The first page in P(i) generates a fault and hence is different from all pages in P(i-1) if i ≥ 2.

A page is called new with respect to P(i), where i ≥ 2, if it is referenced in P(i) but not in P(i-1). In P(1) every page is new.

n_i = # new pages in P(i)

2.6 Randomized paging

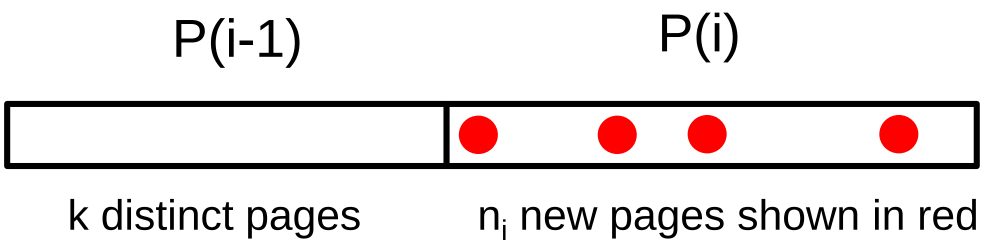
In the following the term cost refers to #page faults incurred.

We will show that in each P(i)

- amortized cost OPT ≥ $n_i/2$
- expected cost RMARK ≤ $n_i H_k$

Analysis OPT

Consider the subsequence consisting of P(i-1) and P(i), i ≥ 2. Exactly $k+n_i$ distinct pages are referenced so that OPT must incur at least n_i faults when serving the subsequence. In P(1) at least n_1 faults are incurred.



2.6 Randomized paging

By combining pairs of (a) odd and even numbered phases and (b) even and odd numbered phases, we obtain:

(a) $OPT(\sigma) \geq n_2 + n_4 + n_6 + \dots$

(b) $OPT(\sigma) \geq n_1 + n_3 + n_5 + \dots$

Summing (a) and (b) and dividing by 2, we get $OPT(\sigma) \geq \Sigma_{1\leq i\leq l}\, n_i/2$ so that a cost of $n_i/2$ can be charged to P(i).

Cost RMARK

Fix any P(i). RMARK incurs a cost of n_i for serving requests to new pages.

During the service of P(i) a page is called old if it is unmarked but was requested in P(i-1). Note: When an old page is referenced, it ceases to be old.

2.6 Randomized paging

o_i = # requests to old pages in P(i)

Analyze expected cost of j-th request to an old page, $1 \leq j \leq o_i$.

$n_i(j)$ = # new pages requested before j-th request to an old page

Immediately before the j-th request to an old page, there exist k-(j-1) old pages, $n_i(j)$ of which do not reside in fast memory. The probability of absence is the same for all the old pages. This holds true because RMARK evicts unmarked pages uniformly at random.

Hence the expected cost of the j-th request to an old page is

$$\frac{n_i(j)}{k-(j-1)} \leq \frac{n_i}{k-(j-1)}.$$

2.6 Randomized paging

There holds $o_i < k$ and $n_i + o_i = k$.

Hence the expected cost for serving requests to old pages is

$$\sum_{j=1}^{o_i} \frac{n_i}{k-(j-1)} \leq n_i(1/k + \ldots + 1/2) = n_i(\mathrm{H}_k - 1).$$

RMARK's total expected cost in P(i) is at most $n_i + n_i(H_k-1) = n_i H_k$.

2.6 Randomized paging

Theorem: Let A be a randomized online paging algorithm. If A is c-competitive against oblivious adversaries, then $c \geq H_k$.

Proof: $S=\{p_1,\ldots,p_{k+1}\}$ set of k+1 pages

ADV: oblivious adversary

At any time while constructing a request sequence σ, ADV maintains a probability vector $Q=(q_1,\ldots,q_{k+1})$.

q_i = probability that p_i is not in A‘s fast memory

There holds $\Sigma_{1\leq i\leq k+1}\, q_i = 1$ because at any time exactly one page does not reside in fast memory.

Initially, $p_1,\ldots,p_k$ are in fast memory; an arbitrary one gets labeled.

ADV constructs σ in phases.

2.6 Randomized paging

Each phase consists of k subphases.

Construction of subphase j, $1 \le j \le k$.

Invariant: At the beginning of the subphase there are j labeled pages and $u=k+1-j$ unlabeled pages. The labels guide ADV which pages to request. ADV will enforce an expected cost of at least $1/u = 1/(k+1-j)$ to algorithm A. Moreover, one additional page will get labeled.

At the end of the k-th subphase, the last page that got labeled remains labeled. This maintains the invariant for the first subphase of the following phase.

Over all the k subphases ADV enforces an expected cost of

$\Sigma_{1 \le i \le k} 1/(k+1-j) = H_k$.

2.6 Randomized paging

At any time let L = {indices of labeled pages} and $\lambda = \Sigma_{i \in L}\, q_i$.

Request generation in subphase j.

1. $\lambda = 0$ at the beginning of the subphase: There exists an unlabeled page p_i with $q_i \geq 1/u$. ADV requests p_i and labels it. Subphase ends.
2. $\lambda > 0$ at the beginning of the subphase: There exists a labeled page p_i with $q_i = \varepsilon > 0$.

   ADV requests p_i.

   while $\lambda > \varepsilon$ and A's expected cost in subphase is $< 1/u$ **do**

   ADV requests labeled page with largest q-value

   endwhile

   ADV requests unlabeled page with highest q-value and labels it.

2.6 Randomized paging

If in Case 2 the while-loop ends with $\lambda \leq \varepsilon$, then there exists an unlabeled page with a q-value of at least $(1-\varepsilon)/u$. The requests issued before and after the while-loop then yield an expected cost of at least $\varepsilon+(1-\varepsilon)/u \geq 1/u$.

ADV can serve the request sequence so that it incurs a fault only on the pages that are labeled/requested last in the phases.

2.6 Randomized paging

Will develop an alternative proof for the lower bound based on Yao's minimax principle. The latter is based on von Neumann's minimax theorem.

Informally: Performance of best randomized algorithm is equal to the performance of the best deterministic algorithm on a worst-case input distribution.

Let P be a probability distribution on possible inputs (request sequences).

Let A be any deterministic online algorithm. The competitive ratio c_A^P of A given P is the infimum of all c such that

$$E[A(\sigma)] \leq c \cdot E[OPT(\sigma)] + a$$

where σ is generated according to P.

2.6 Randomized paging

Theorem: Yao's Minimax Principle

$$\inf_R c_R = \sup_P \inf_A c_A^P$$

where c_R is the competitive ratio of randomized algorithm R.

Other performance measure is running time:

$$\inf_R T_R = \sup_P \inf_A T_A^P$$

T_R = expected worst-case running time of randomized algorithm R

T_A^P = expected running time of deterministic algorithm A if input is generated according to P.

Proof technique

General approach to establish a lower bound using Yao's principle

Task of algorithm designer/analyzer:

Construct a specific probability distribution P_0 for generating input.

Evaluate the expected costs of

- every deterministic online algorithm A and
- OPT

so as to obtain lower bound on $c_A^{P_0}$, for every A. This gives a lower bound on $\inf_A c_A^{P_0}$ and hence on $\sup_P \inf_A c_A^P$. By Yao's principle, one obtains a lower bound on $\inf_R c_R$.

2.6 Randomized paging

Theorem: Let A be a randomized online paging algorithm. If A is c-competitive against oblivious adversaries, then $c \geq H_k$.

Proof: Alternative proof using Yao's principle.

$S=\{p_1,\ldots,p_{k+1}\}$ set of k+1 pages

Initially, $p_1,\ldots,p_k$ are in fast memory.

Probability distribution for generating request sequences.

$\sigma(1) = p_{k+1}$

Every $\sigma(t)$, $t \geq 2$, requests a page chosen uniformly at random from $S \setminus \{\sigma(t-1)\}$.

Consider any deterministic online paging algorithm A.

A has a cost of 1 on $\sigma(1)$ and an expected cost of 1/k on every $\sigma(t)$, $t \geq 2$.

2.6 Randomized paging

In order to analyze expected cost, we partition a request sequence into phases like a MARKING algorithm: The first phase P(1) starts with σ(1). Phase P(i), i ≥1, ends when k distinct pages have been requested in P(i) and a request to the (k+1)-st distinct page occurs. This request forms the first request of P(i+1).

We analyze and compare expected cost in any phase P(i).

Analysis of OPT

OPT can serve the request sequence so that it incurs a page fault only on the first request of each phase. More precisely, when OPT incurs a page fault on a page p, all pages of S\{p} are in fast memory, and OPT can evict the page not referenced in the current phase.

Hence OPT has a cost of 1 per phase.

2.6 Randomized paging

Analysis of algorithm A

The expected cost of A in any P(i) is 1/k · expected length of P(i).

We analyze the expected phase length.

This is a Coupon's Collector Problem.

Subphase j, $1 \leq j \leq k$: Starts with the j-th distinct page requested (collected) in P(i). Ends just before the (j+1)-st distinct page is referenced. When $\sigma(t)$ is generated, a page is chosen uniformly at random from $S \setminus \{\sigma(t-1)\}$. Among these k pages, k-(j-1) will terminate the subphase. Success probability that subphase j ends is (k-(j-1))/k.

The expected length of subphase j is k/(k+1-j).

Summung over all j, the expected length of a phase is $k \cdot H_k$.

In summary, the expected cost of A in P(i) is H_k.

2.6 Randomized paging

Remark

The above process of generating a request sequence can be viewed as a random walk on a complete graph K_{k+1} consisting of k+1 vertices. The random walk always resides on one of the vertices. In each time step it moves to one of the k neighboring vertices chosen uniformly at random. A request sequence corresponds to the sequence of vertices visited.

Cover time of a random walk: Expected number of steps to visit all vertices, starting from an arbitrary vertex. For K_{k+1} this is again a Coupon‘s Collector Problem.

Expected length of a phase, as defined above, is equal to the cover time.

2.6 Randomized paging

Online algorithm

- Random: On a fault evict a page chosen uniformly at random from among the pages in fast memory.

Theorem: Random is k-competitive against adaptive online adversaries.

Theorem: Let A be a randomized online paging algorithm. If A is c-competitive against adaptive online adversaries, then $c \geq k$.

2.6 Randomized paging

Theorem: Random is k-competitive against adaptive online adversaries.

Proof: Let $\sigma = \sigma(1), \ldots, \sigma(m)$ be an arbitrary request sequence.

S_R = set of pages in Random's fast memory

S_{ADV} = set of pages in ADV's fast memory

$$\Phi = k \mid S_R \setminus S_{ADV} \mid$$

Let $R(\sigma(t))$ denote the cost incurred by Random on $\sigma(t)$.

We will show that, for any t with $1 \le t \le m$, there holds

$$R(\sigma(t)) + E[\Phi(t) - \Phi(t-1)] \le k \cdot ADV(\sigma(t)).$$

Hence $R(\sigma(t)) + E[\Phi(t)] - E[\Phi(t-1)] \le k \cdot ADV(\sigma(t))$ and by summing over all t we obtain

$$R(\sigma) \le k \cdot ADV(\sigma) - E[\Phi(m)] + E[\Phi(0)].$$

Assume that initially the fast memory contains k arbitrary pages.

2.6 Randomized paging

Consider any time t and assume that ADV generates request $\sigma(t)) = x$.

1. $x \in S_R$ and $x \in S_{ADV}$

   $R(\sigma(t)) + E[\Delta\Phi] = 0 + 0 = k \cdot ADV(\sigma(t))$

2. $x \in S_R$ and $x \notin S_{ADV}$

   In order to serve the request / page fault, ADV may evict a page contained in S_R, in which case the potential increases by k.

   $R(\sigma(t)) + E[\Delta\Phi] \leq 0 + k = k \cdot ADV(\sigma(t))$

3. $x \notin S_R$ and $x \in S_{ADV}$

   Since $x \in S_{ADV} \setminus S_R$, there must exist a page $y \in S_R \setminus S_{ADV}$. With probability 1/k, Random evicts y, in which case the potential drops by k.

   $R(\sigma(t)) + E[\Delta\Phi] \leq 1 - 1/k \cdot k = 0 = k \cdot ADV(\sigma(t))$

2.6 Randomized paging

4. $x \notin S_R$ and $x \notin S_{ADV}$

(Combination of Cases 2 and 3)

When serving the page fault, ADV may evict a page contained in S_R, in which case the potential increases by k.

Then $x \in S_{ADV} \setminus S_R$, and there must exist a page $y \in S_R \setminus S_{ADV}$. With probability 1/k, Random evicts y, in which case the potential drops by k.

$R(\sigma(t)) + E[\Delta\Phi] \leq 1 + k - 1/k \cdot k = k = k \cdot ADV(\sigma(t))$

2.6 Randomized paging

Theorem: Let A be a randomized online paging algorithm. If A is c-competitive against adaptive online adversaries, then $c \geq k$.

Proof: $S=\{p_1,\ldots,p_{k+1}\}$ set of k+1 pages

Initially A has $p_1,\ldots,p_k$ in fast memory.

Generation of σ: ADV always requests the page not available in A's fast memory. Hence A has a fault on every request and $A(\sigma) = m$, where m is the length of σ.

We will define k algorithms $B_1,\ldots,B_k$ such that $\Sigma_{1\leq i\leq k} B_i(\sigma) = m$. The adversary ADV chooses one of the algorithms uniformly at random so that $E[ADV(\sigma)] = 1/k \cdot \Sigma_{1\leq i\leq k} B_i(\sigma) = m/k$.

2.6 Randomized paging

Definition of B_i, $1 \le i \le k$: Initially pages $S \setminus \{p_i\}$ are in fast memory. If B_i has a fault on $\sigma(t)$, it evicts the page requested by $\sigma(t-1)$.

We will show that $B_1,\ldots,B_k$ always have different configurations, i.e. for any two B_i, B_j the page not in fast memory is different. This implies that on every request only one of the k algorithms has a page fault.

Claim: B_i, B_j, with $i \neq j$, always have different configurations.

Proof: Induction on the number of requests served. Statement of the claim holds initially. Suppose that it holds before the service of a request $\sigma(t)$, referencing page p.

- Page p in fast memories of B_i, B_j: Configurations do not change.
- Page p not in fast memory of one of the algorithms, say B_i: Then B_i evicts the page referenced by $\sigma(t-1)$, which still remains in the fast memory of B_j.

2.7 Refinements of competitive paging

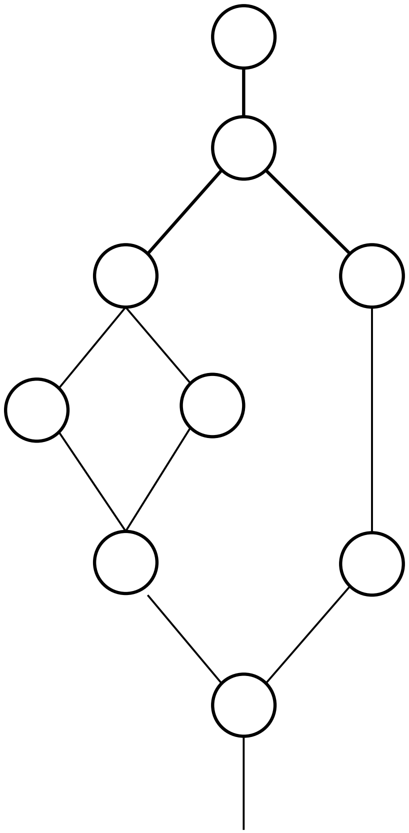
Deficiencies of the basic results:

- Competitive ratio of LRU/FIFO higher than the ratios observed in practice (typically in the range [1,2]).
- In practice LRU much better than FIFO

Reason: Request sequences in practice exhibit locality of reference, i.e. (short) subsequences reference few distinct pages.

2.7 Refined models

1. Access graph model: G(V,E) undirected graph. Each node represents a memory page. Page p can be referenced after q if p and q are adjacent in the access graph.



Competitive factors depend on G.

$\forall\, G : c_{LRU}(\mathrm{G}) \le c_{FIFO}(\mathrm{G})$

$\forall\, T : c_{LRU}(\mathrm{T})$ smallest possible ratio

Problem: quantify $c_A(\mathrm{G})$ for arbitrary G

2.7 Refined models

2. Markov paging: n pages

q_{ij} = probability that request to page i is followed by request to page j

$$Q = \begin{pmatrix} q_{11} & \cdots & q_{1n} \\ \vdots & \ddots & \vdots \\ q_{n1} & \cdots & q_{nn} \end{pmatrix}$$

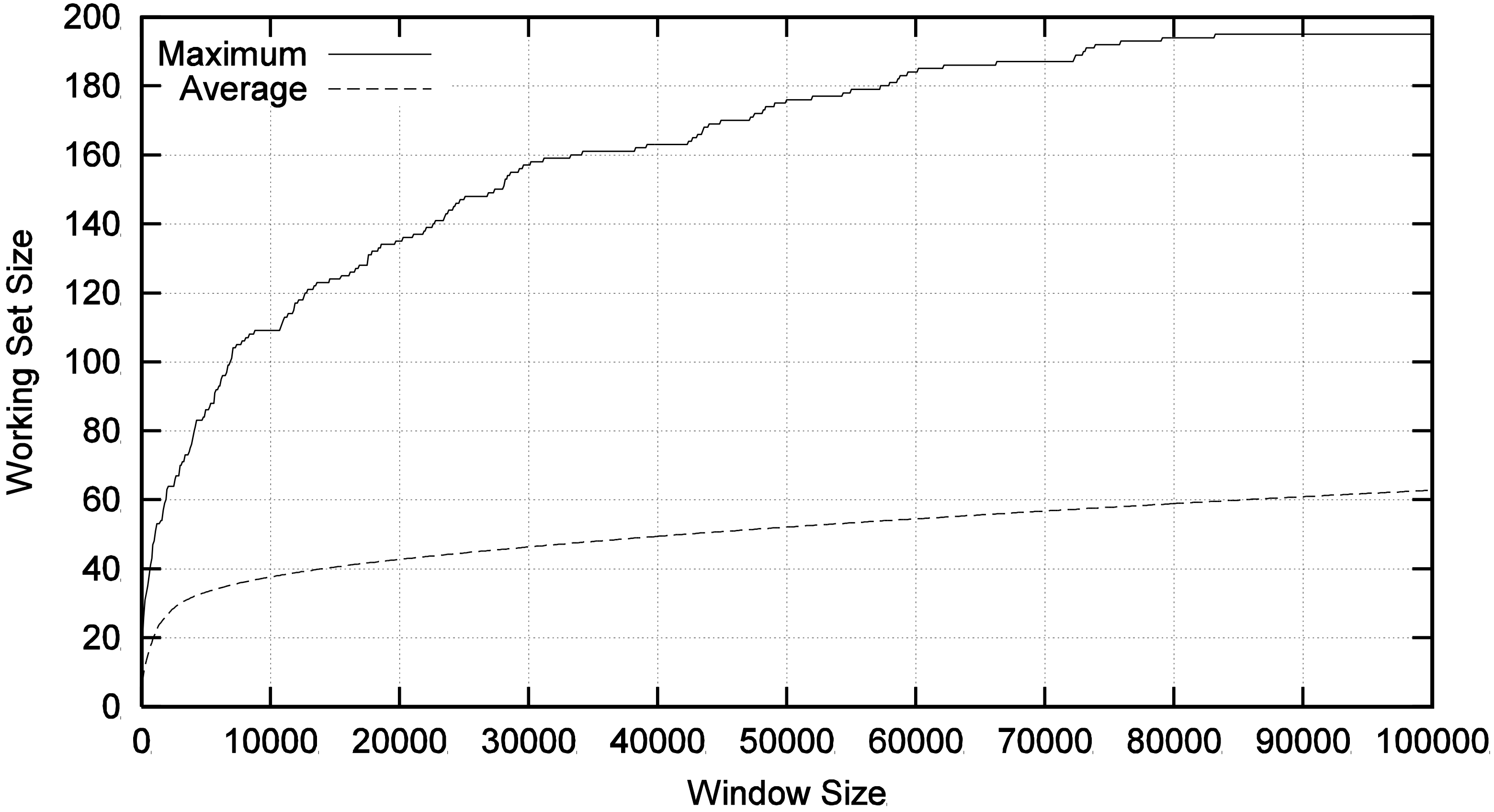
Page fault rate of A on σ = # page faults incurred by A on σ / |σ|

2.7 Refined models

3. Denning's working set model: n pages

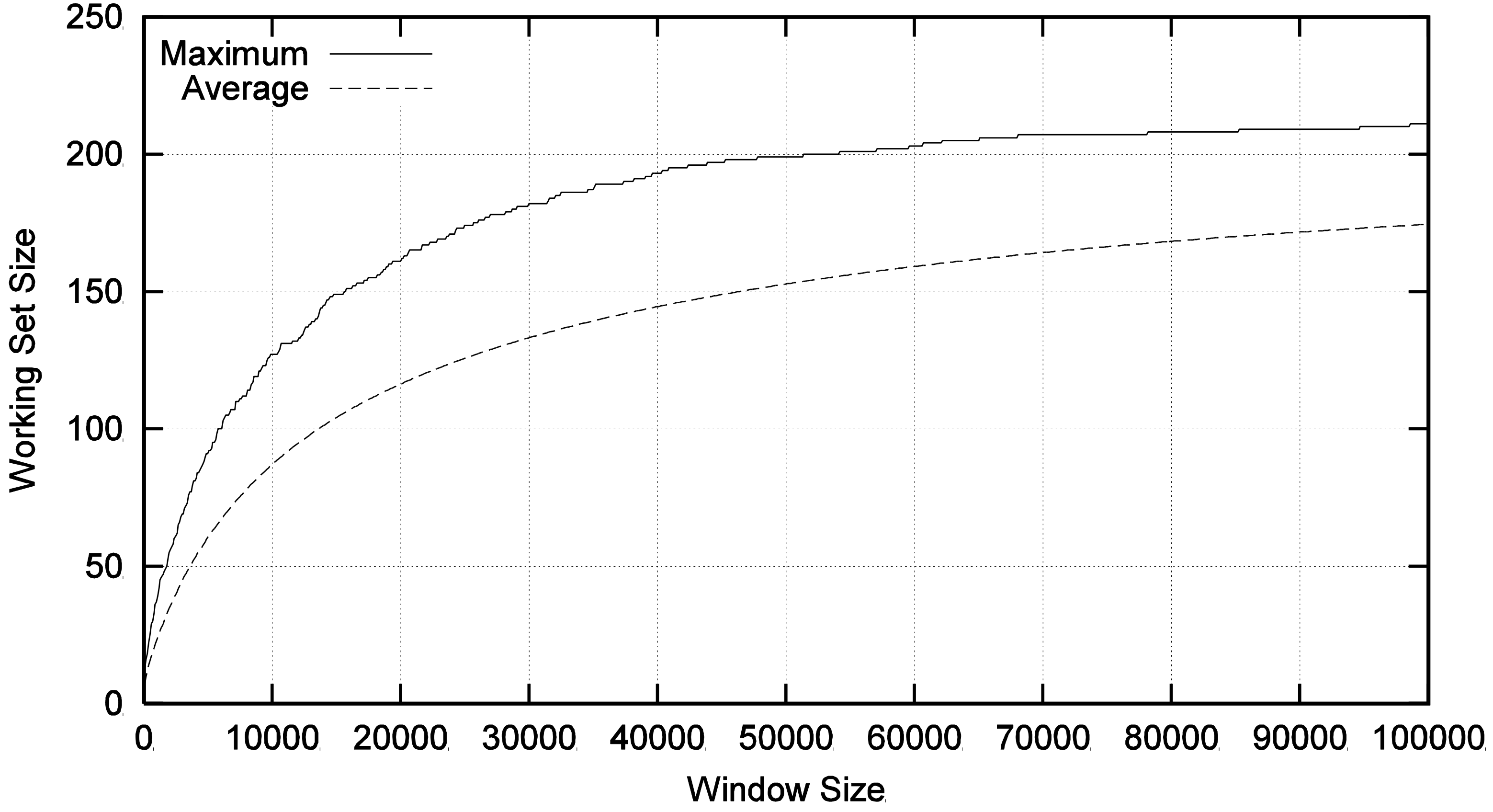
Concave function

2.7 Refined models



SPARC, GCC, 196 pages

2.7 Refined models



SPARC, COMPRESS, 229 pages

2.7 Refined models

Program executed on CPU characterized by concave function f.
It generates σ that are consistent with f.

Max-Model: σ consistent with f if, for all $n \in \mathbb{N}$, the number of distinct pages referenced in any window of length n is at most f(n).

Average-Model: σ consistent with f if, for all $n \in \mathbb{N}$, the average number of distinct pages referenced in windows of length n is at most f(n).

2.7 Refined models

4. Inter-request distances

$\sigma =$ B A C B D A A D B A C D B A C B C A B D A B C

$\sigma = \sigma(1), \ldots, \sigma(m)$

$\sigma(t)$ is a distance-l request if

$$\overset{A}{\sigma(t')}\ \underbrace{\ldots\ \ldots\ \ldots\ \ldots}_{l \text{ distinct pages}}\ \overset{A}{\sigma(t)}$$

2.7 Refined models

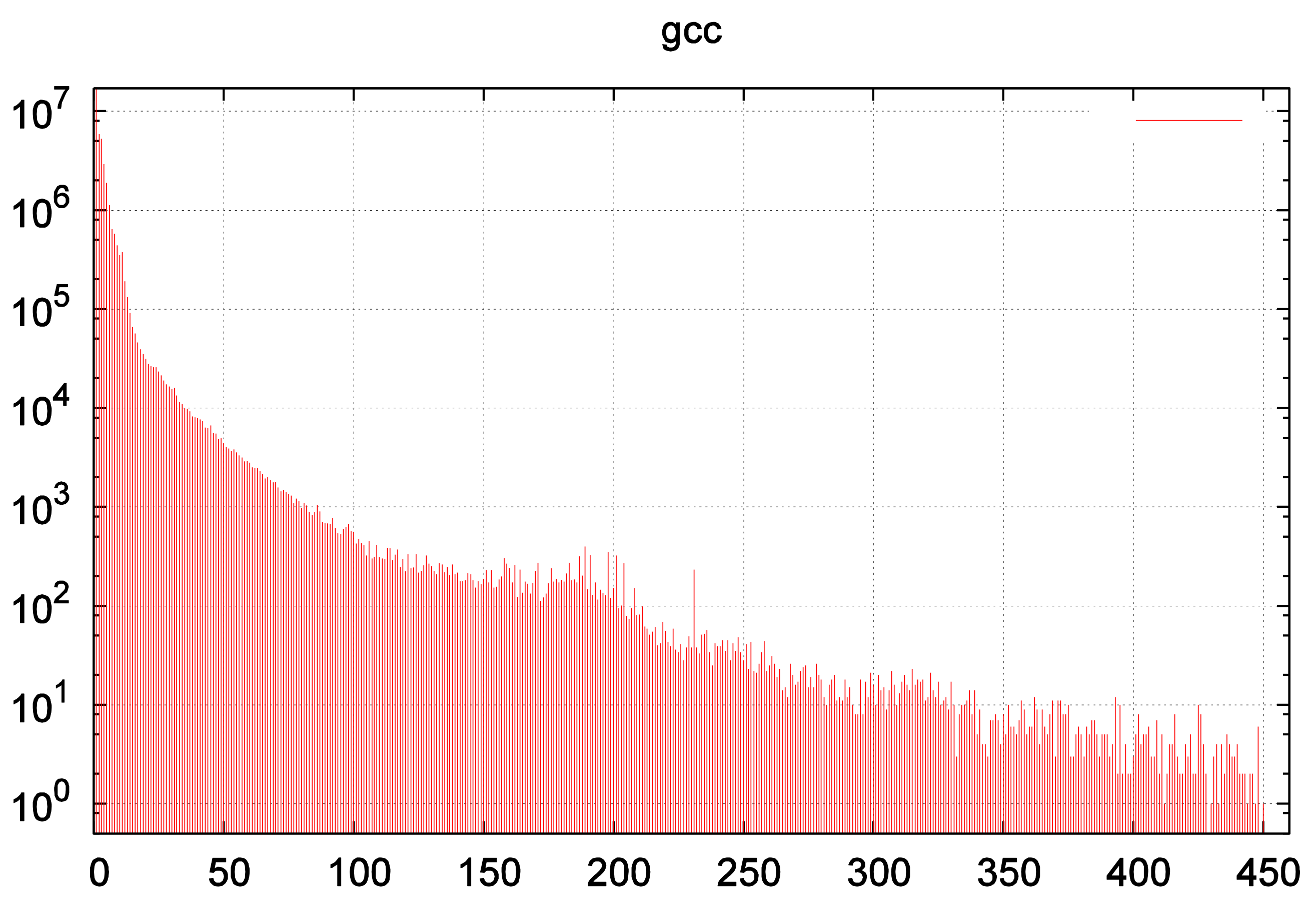
Characteristic vector

$C = (c_0, \ldots, c_{n-1})$ n = #pages

In σ characterized by C there are c_l distance-l requests, $0 \le l \le n-1$

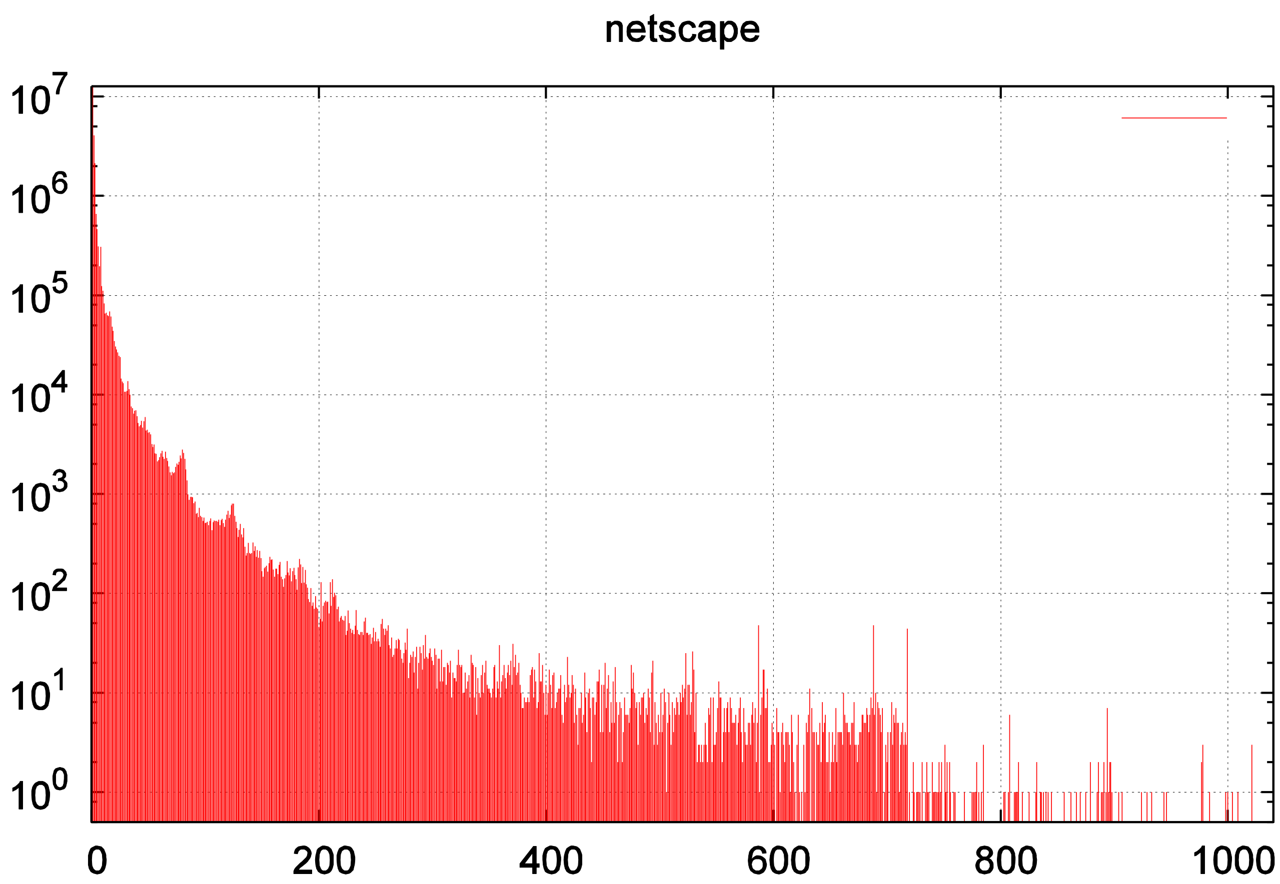
$$R_{ALG}(C) = \max_\sigma \; ALG(\sigma) / OPT(\sigma)$$

2.7 Refined models



Characteristic vector, gcc, 37524334, 468 pages

2.7 Refined models



Characteristic vector, netscape, 22077106, 1037 pages

2.7 Refined models

Characteristic vector

LRU(σ) = $\sum_{l=k}^{n-1} c_l$

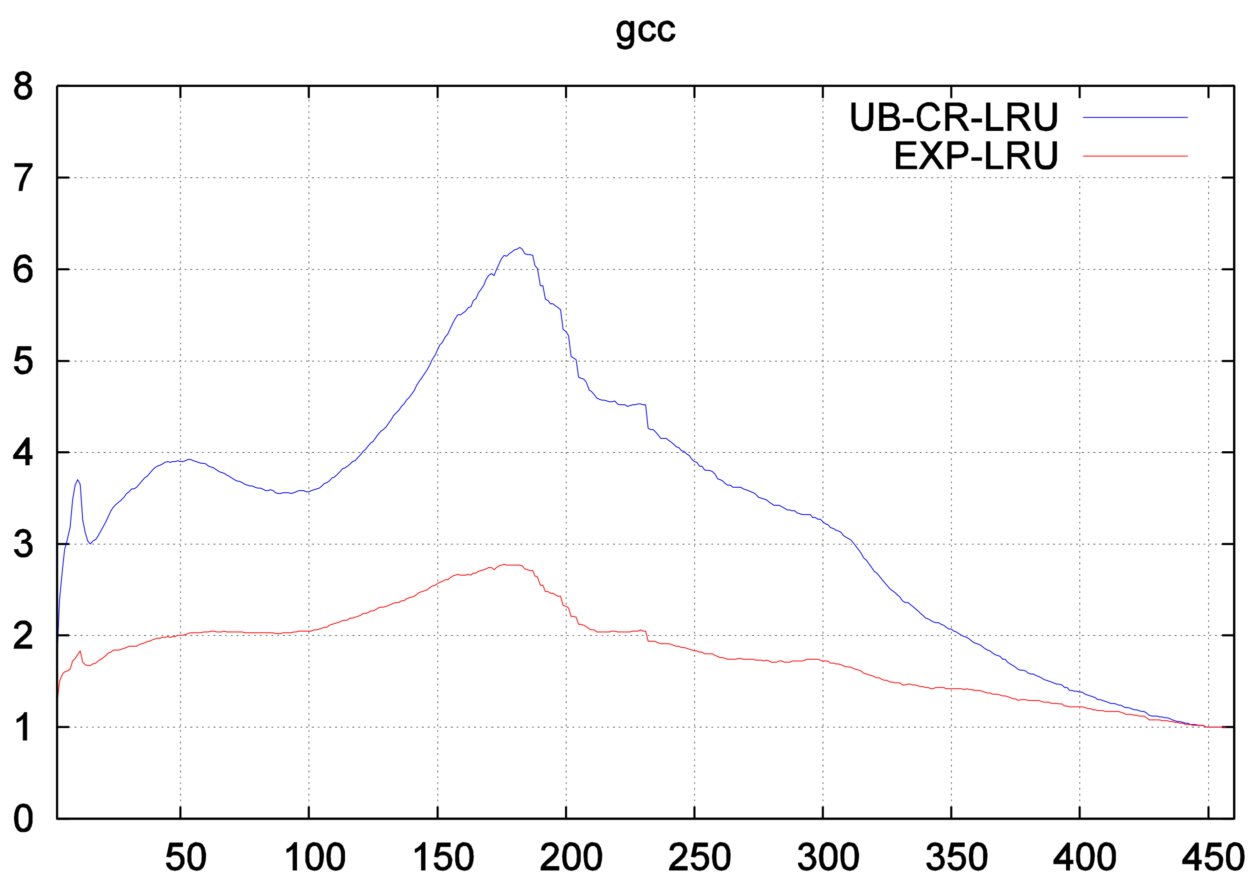
OPT(σ) ≥ max {k + $\sum_{l=k}^{\lambda} c_l(l-k+1)/(k-1)$ + $c'_\lambda(\lambda-k+1)/(k-1)$, n}

f(λ, c'_λ) = g(λ, c'_λ)

f(j,γ) = k + $\sum_{l=k}^{j-1} c_l(l-k+1)/(k-1)$ + $\gamma(j-k+1)/(k-1)$

g(j,γ) = n + (c_j-γ) + $\sum_{l=j+1}^{n-1} c_l$

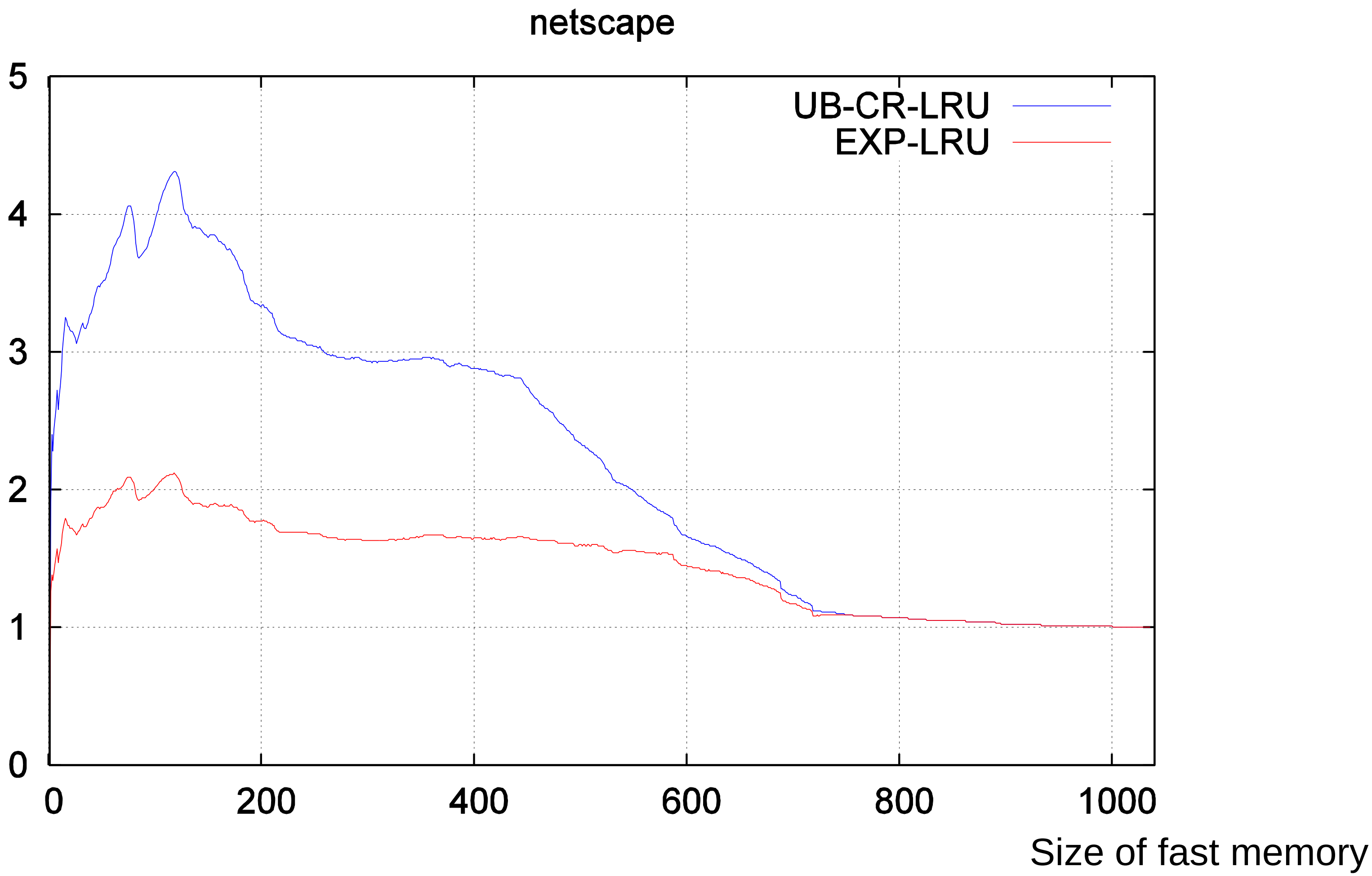
2.7 Refined models



Size of fast memory

Competitiveness gcc

2.7 Refined models



Competitiveness netscape

2.8 Randomized list update

List update problem:

Unsorted linear list

L: B → C → E → A → D

$\sigma =$ A A C B E D A …

Request: Access to item in the list

Cost: Accessing the i-th item in the list incurs a cost of i.

Rearrangements: After an access, requested item may be moved at no extra cost to any position closer to the front of the list (free exchanges). At any time two adjacent items may be exchanged at a cost of 1.

Goal: Minimize cost paid in serving σ.

2.8 Randomized list update

Algorithm Random Move-To-Front (RMTF): With probability ½, move requested item to the front of the list.

Theorem: The competitive ratio of RMTF is not smaller than 2, for a general list length n.

2.8 Randomized list update

Theorem: The competitive ratio of RMTF is not smaller than 2, for a general list length n.

Proof: Assume that the elements in the initial list are in the order $x_1, x_2, \ldots, x_n$.

The request sequence σ consists of phases, where each phase is of the form

$(x_n)^k (x_{n-1})^k \ldots (x_1)^k$ for some k>1.

We first analyze RMTF's expected cost to serve $(x_i)^k$, for any $1 \leq i \leq n$, assuming that x_i is stored at the last position of the list. If x_i is moved to the front of the list on the j-th request of $(x_i)^k$, which happens with probability $(1/2)^j$, then the service cost is jn+k-j. Thus the expected service cost is

$$\sum_{j=1}^{k} (1/2)^j \, (jn + k - j) \geq \sum_{j=1}^{k} (1/2)^j \, jn = 2n\left(1 - k(1/2)^{k+1} - (1/2)^k\right).$$

2.8 Randomized list update

The last equation hold because, for any z,

$$\sum_{j=0}^{k} jz^{j-1} = \frac{kz^{k+1}-(k+1)z^{k}+1}{(z-1)^2}.$$

We argue that with probability at least $1-n/2^k$, x_i is at the last position of the list when $(x_i)^k$ is requested. If x_i is not at the last position when $(x_i)^k$ is referenced, then there exists an item x_j, j≠i, that was not moved to the front of the list when $(x_j)^k$ was served. This happens with probability $1/2^k$. Since there exist n-1 items x_j with j≠i, by the Union Bound, the probability that x_i is not at the last position is upper bounded by $(n-1)/2^k < n/2^k$.

It follows that RMTF's expected cost on a phase is at least

$$2n^2\left(1-k(1/2)^{k+1}-(1/2)^k\right)(1-n/2^{\,k}).$$

OPT can use MTF to serve σ, which incurs a cost of n(n+k-1)≤n(n+k) per phase.

2.8 Randomized list update

We conclude that the ratio c = E[RMTF(σ)]/OPT(σ) satisfies

$$c \geq \frac{2(1-k(1/2)^{k+1}-(1/2)^{k})(1-n/2^{k})}{(1+k/n)}.$$

Setting $k = 2\lceil \log n \rceil$ we obtain

$$c \geq \frac{2(1-\lceil \log n \rceil(1/n^{2})-(1/n^{2}))(1-1/n)}{(1+2\lceil \log n \rceil/n)}.$$

and this ratio tends to 2 as n goes to infinity.

2.8 Randomized list update

Unsorted, linear linked list of items

$$\begin{array}{lccccccccc} & 1 & & 0 & & 1 & & 1 & & 0 \\ L: & \boxed{Y} & \rightarrow & \boxed{Z} & \rightarrow & \boxed{V} & \rightarrow & \boxed{X} & \rightarrow & \boxed{U} \end{array}$$

$\sigma =$ X X Z Y V U X …

Algorithm BIT: Maintain bit b(x), for each item x in the list. Bits are initialized independently and uniformly at random to 0/1. Whenever an item is requested, its bit is complemented. If value changes to 1, item is moved to the front of the list.

Theorem: BIT is 1.75-competitive against oblivious adversaries.

See [BY], pages 24-26.

2.8 Randomized list update

Theorem: BIT is 1.75-competitive against oblivious adversaries.

Proof: We extend the analysis of MTF.

Inversion: ordered pair (x,y) of items such that

x before y in OPT's list

x behind y in BIT's list

Inversion (x,y) has type 1 if $b(x)=0$

has type 2 if $b(x)=1$

$$\Phi = 2 \cdot \#\ \text{type-2 inversions} + \#\ \text{type-1 inversions}$$

Intuition: A type-2 inversion is harder to resolve. It requires two requests to x before BIT can break up a type-2 inversion (x,y).

$\Phi \geq 0$ $\quad$ $\Phi(0) = 0$ if initial lists of OPT and BIT are identical

2.8 Randomized list update

Fact: At any time and for each item x, b(x) is equal to 0 (respectively 1) with probability ½.

To verify the fact, observe that at any time

$$b(x) = (\text{initial bit value of } x + \#\text{requests to } x \text{ so far}) \bmod 2$$

and this expression only depends on the initial bit value.

Consider an arbitrary request sequence $\sigma = \sigma(1),\ldots,\sigma(m)$. We will show that for any t, $1 \leq t \leq m$,

$$BIT(\sigma(t)) + E[\Phi(t)-\Phi(t-1)] \leq 1.75\ OPT(\sigma(t)).$$

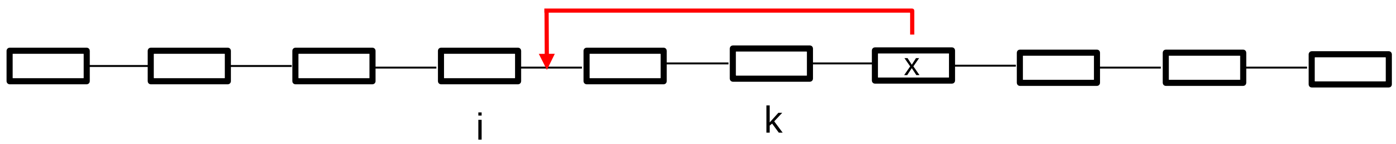
This implies

$$BIT(\sigma) \leq 1.75\ OPT(\sigma) - E[\Phi(m)] + E[\Phi(0)]\ .$$

Let $\sigma(t)$ be an arbitrary request, and let x denote the requested item.

2.8 Randomized list update

1. Analysis of OPT‘s moves



Assume that in OPT‘s list, x is at position k+1. The access cost is k+1.

Suppose that after the access, OPT inserts x behind the i-th item, where i≤k. For each item y that is passed, an inversion (x,y) can be created. With equal probabiliy ½ the bit value b(x) is 0 or 1. Thus an inversion has type-1 or type-2 with equal probability ½, causing an expected potential increase of ½·1+ ½·2 = 1.5. Since k-i items are being passed, the expected potential increase is at most 1.5(k-i).

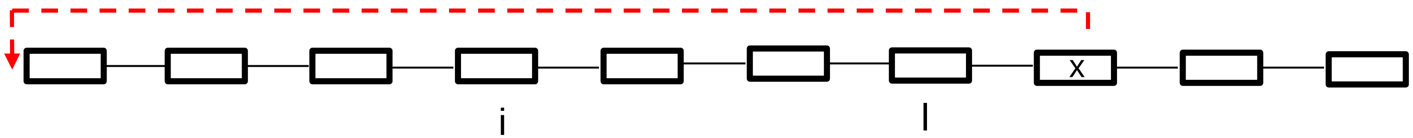
Additionally, OPT may perform, say, p(t) paid exchanges. For each item swap, an inversion can be created, causing an expected potential increase of 1.5.

Actual cost of OPT on σ(t) = k+1+p(t)

E[ΔΦ] due to OPT‘s moves ≤ 1.5(k-i+p(t))

2.8 Randomized list update

2. Analyis of BIT‘s moves



Assume that in BIT‘s list, x is at position l+1. The access cost is l+1.

Case 1: $l \geq i$

Since $l \geq i$ there must exist at least l-i items y_j that are stored before x in BIT‘s list but behind x in OPT‘s list. Hence there exist at least l-i inversions of the form (x,y_j).

b(x) = 0 before the request: The bit b(x) flips to 1. BIT moves x to the front of the list and destroys the inversions (x,y_j) (potential drop of at least l-i). For each of the first i items in OPT‘s list, an inversion can be created (expected potential increase of at most 1.5i). $E[\Delta\Phi] \leq -(l-i) + 1.5i$.

b(x) = 1 before the request: Item x does not move but b(x) changes to 0 so that each inversion (x,y_j) changes type, from type-2 to type-1. Thus $\Delta\Phi \leq -(l-i)$.

2.8 Randomized list update

The two above cases, $b(x) = 0$ and $b(x) = 1$, occur with equal probability of ½. Therefore:

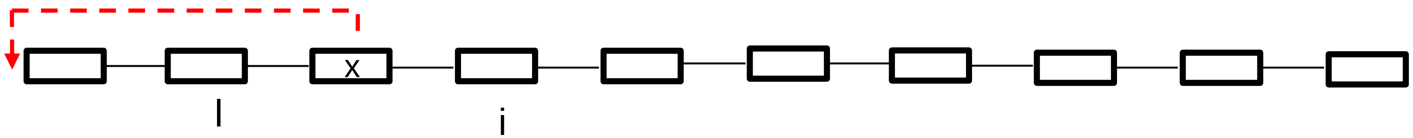
Actual cost of BIT on $\sigma(t)$ $= l+1$

$E[\Delta\Phi]$ due to BIT's moves $\leq -(l-i) + \frac{1}{2} \cdot 1.5i = -(l-i) + 0.75i$

$$
\begin{aligned}
BIT(\sigma(t)) + E[\Delta\Phi] &\leq l+1 + 1.5(k-i+p(t)) - (l-i) + 0.75i \\
&\leq 1.75(k-i+p(t)) + 1.75\, i + 1 \\
&< 1.75(k+1+p(t)) \\
&= 1.75 \cdot OPT(\sigma(t))
\end{aligned}
$$

2.8 Randomized list update

2. Analyis of BIT‘s moves



Case 2: $l < i$

b(x) = 0 before the request: BIT moves x to the front of the list and may create l inversions, each of which increases the potential by an expected value of 1.5. Hence $E[\Delta\Phi] \leq 1.5l$.

b(x) = 1 before the request: Item x does not move and b(x) changes to 0 so that each inversion (x,y_j) changes type, from type-2 to type-1. Thus $\Delta\Phi \leq 0$.

Again the two above cases occur with equal probability.

Actual cost of BIT on $\sigma(t) = l+1$

$E[\Delta\Phi]$ due to BIT‘s moves $\leq 0.75l$

2.8 Randomized list update

We conclude

$$
\begin{aligned}
BIT(\sigma(t))+E[\Delta\Phi] &\le l+1 + 1.5(k-i+p(t)) + 0.75l \\
&< 1.75(k-i+p(t)) + 1.75\, i + 1 \\
&< 1.75(k+1+p(t)) \\
&= 1.75\cdot OPT(\sigma(t)),
\end{aligned}
$$

where the second inequality holds because $l<i$.

2.8 Randomized list update

L: W → V → U → Y → X → Z

σ = … X U Y V V W W X …

Algorithm TIMESTAMP(p): Let $0 \le p \le 1$. Serve a request to item x as follows.

With probability p move x to the front of the list.

With probability 1-p insert x in front of the first item in the list that has been referenced at most once since the last request to x.

Theorem: TIMESTAMP(p), with $p = (3-\sqrt{5})/2$, achieves a competitive ratio of $(1+\sqrt{5})/2 \approx 1.62$ against oblivious adversaries.

2.8 Randomized list update

Algorithm Combination: With probability 4/5 serve a request sequence using BIT and with probability 1/5 serve it using TIMESTAMP(0).

Theorem: Combination is 1.6-competitive against oblivious adversaries.

Theorem: Let A be a randomized online algorithm for list update. If A is c-competitive against adaptive online adversaries, for a general list length, then $c \geq 2$.

2.8 Randomized list update

Theorem: Let A be a randomized online algorithm for list update. If A is c-competitive against adaptive online adversaries, for a general list length, then $c \geq 2$.

Proof: Let n denote the list length.

Request generation: ADV always requests last item in A's list.

$A(\sigma) = m \cdot n$ $\quad$ m = length of the constructed σ

ADV: In order to serve σ, ADV chooses one of the n! possible list configurations uniformly at random and serves σ with this static list.

Consider an arbitrary request $\sigma(t)=x$. With probability $(n-1)!/n! = 1/n$ item x is stored at position i, for any $1 \leq i \leq n$. Therefore, ADV's expected service cost for the request is $\Sigma_{1\leq i\leq n}\, i \cdot 1/n = (n+1)/2$.

At most $\binom{n}{2} \leq n(n-1)/2$ paid exchanges are required to bring the initial list into the selected static ordering.

2.8 Randomized list update

Hence the cost ratio c = A(σ)/ADV(σ) satisfies

$$c \geq \frac{mn}{m(n+1)/2 + n(n-1)/2} = \frac{2n}{n+1+n(n-1)/m}$$

and the latter ratio tends to 2-2/(n+1) as m goes to infinity.

2.9 Data compression

String S to be represented in a more compact way using fewer bits.
Symbols of S are elements of an alphabet Σ, e.g. $\Sigma = \{x_1, \ldots, x_n\}$.

Encoding: Convert string S of symbols into string I of integers.
Encoder maintains a linear list L of all the elements of Σ. It reads the symbols of S sequentially. Whenever symbol x_i has to be encoded, encoder looks up the current position of x_i in L, outputs this position and updates the list using a given algorithm.

$$S = \ldots \quad x_1 \; x_1 \; x_2 \; x_1 \; x_2 \; x_3 \; x_3 \; x_2 \quad \ldots$$

$$L: \quad \boxed{x_3} \rightarrow \boxed{x_4} \rightarrow \boxed{x_2} \rightarrow \boxed{x_5} \rightarrow \boxed{x_1} \rightarrow \boxed{x_6}$$

$$I = \ldots \quad 5 \; 1 \; 4 \; 2 \quad \ldots$$

Generates compression because frequently occuring symbols are stored near the front of the list and can be encoded using small integers/ few bits.

2.9 Data compression

Decoding: Decoder also maintains a linear list L of all the elements of Σ. It reads the integers of I sequentially. Whenever integer j has to be decoded, it looks up the symbol currently stored at position j in L, outputs this symbol and updates the list using the same algorithm as the encoder.

I = … 5 1 4 2 …

L: $x_3 \to x_4 \to x_2 \to x_5 \to x_1 \to x_6$

S = … x_1 x_1 x_2 x_1 x_2 x_3 x_3 x_2 …

2.9 Data compression

Integers of I have to be encoded using a variable-length prefix code.

A prefix code satisfies the „prefix property“:

No code word is the prefix of another code word.

Possible encoding of j : $2\lfloor \log j \rfloor + 1$ bits suffice

- $\lfloor \log j \rfloor \;\; 0's$ followed by
- binary representation of j, which requires $\lfloor \log j \rfloor + 1$ bits

2.9 Data compression

Two schemes

- Byte-based compression: Each byte in the input string represents a symbol.
- Word-based compresion: Each „natural language“ word represents a symbol.

The following tables report on experiments done using the Calgary corpus (benchmark library for data compression).

2.9 Byte-based compression

File	TS		MTF		
	Bytes	% Orig.	Bytes	% Orig.	Size in Bytes
bib	99121	89.09	106478	95.70	111261
book1	581758	75.67	644423	83.83	768771
book2	473734	77.55	515257	84.35	610856
geo	92770	90.60	107437	104.92	102400
news	310003	82.21	333737	88.50	377109
obj1	18210	84.68	19366	90.06	21504
obj2	229284	92.90	250994	101.69	246814
paper1	42719	80.36	46143	86.80	53161
paper2	63654	77.44	69441	84.48	82199
pic	113001	22.02	119168	23.22	513216
progc	33123	83.62	35156	88.75	39611
progl	52490	73.26	55183	77.02	71646
progp	37266	75.47	40044	81.10	49379
trans	79258	84.59	82058	87.58	93695

2.9 Word-based compression

File	TS		MTF		
	Bytes	% Orig.	Bytes	% Orig.	Size in Bytes
bib	34117	30.66	35407	31.82	111261
book1	286691	37.29	296172	38.53	768771
book2	260602	42.66	267257	43.75	610856
news	116782	30.97	117876	31.26	377109
paper1	15195	28.58	15429	29.02	53161
paper2	24862	30.25	25577	31.12	82199
progc	10160	25.65	10338	26.10	39611
progl	14931	20.84	14754	20.59	71646
progp	7395	14.98	7409	15.00	49379

2.9 Burrows-Wheeler transformation

Transformation: Given S, compute all cyclic shifts and sort them lexicographically.

In the resulting matrix M, extract last column and encode it using MTF encoding. Add index I of row containing original string.

Example:

0	a a b r a c	S = a b r a c a
1	a b r a c a	
2	a c a a b r	
3	b r a c a a	
4	c a a b r a	
5	r a c a a b	(c a r a a b, I=1)

2.9 Burrows-Wheeler transformation

Back-transformation: Sort characters lexicographically, gives first and last columns of M.

Fill remaining columns by repeatedly shifting last column in front of the first one and sorting lexicographically.

0	a	c
1	a	a
2	a	r
3	b	a
4	c	a
5	r	b

(c a r a a b, I=1)

2.9 Burrows-Wheeler transformation

Back-transformation using linear space:

- M‘= matrix M in which columns are cyclically rotated by one position to the right.
- Compute vector T that indicates how rows of M and M‘ correspond, i.e. row j of M‘ is row T[j] in M. Example: T = [4 , 0 , 5 , 1 , 2 , 3]

	M	M‘
0	a a b r a c	c a a b r a
1	a b r a c a	a a b r a c
2	a c a a b r	r a c a a b
3	b r a c a a	a b r a c a
4	c a a b r a	a c a a b r
5	r a c a a b	b r a c a a

2.9 Burrows-Wheeler transformation

Back-transformation using linear space:

- L : vector, last column of M = first column of M‘
- L[T[j]] is cyclic predecessor of L[j]

For i=0, … , N-1, there holds S[N-1-i] = L[T^i [I]]

2.9 Burrows-Wheeler transformation

File	Bytes	% Orig.	bits/char	Size in Bytes
bib	28740	25.83	2.07	111261
book1	238989	31.08	2.49	768771
book2	162612	26.62	2.13	610856
geo	56974	55.63	4.45	102400
news	122175	32.39	2.59	377109
obj1	10694	49.73	3.89	21504
obj2	81337	32.95	2.64	246814
paper1	16965	31.91	2.55	53161
paper2	25832	31.24	2.51	82199
pic	53562	10.43	0.83	513216
progc	12786	32.27	2.58	39611
progl	16131	22.51	1.80	71646
progp	11043	22.36	1.79	49379
trans	18383	19.62	1.57	93695

2.9 Burrows-Wheeler transformation

Program	mean bits per character
compress	3.36
gzip	2.71
BW-Trans	2.43

Comparison with Lempel-Ziv-based tools (LZW and LZ77)

2.9 Data compression

Assume that S is generated by a memoryless source $P = (p_1, \ldots, p_n)$.

In a string generated according to P, each symbol is equal to x_i with probability p_i.

The entropy of P is defined as

$$H(P) = \sum_{i=1}^{n} p_i \log (1/p_i)$$

It is a lower bound on the expected number of bits needed to encoded one symbol in a string generated according to P. (Shannon's Source Coding Theorem)

2.9 Huffman code

Constructs optimal prefix codes.

Code tree constructed using greedy approach.

Maintain forest of code trees.

- Initially, each symbol x_i represents a tree consisting of one node with accumulated probability p_i.
- While there exist at least two trees, choose T_1, T_2 having the smallest accumulated probabilies and merge them by adding a new root. New accumulated probability is the sum of those of T_1, T_2.

2.9 Data compression

E_{MTF} (P) = expected number of bits needed to encode one symbol using MTF encoding

Assume that an integer j is encoded using $2\lfloor \log j \rfloor + 1$ bits:

- $\lfloor \log j \rfloor \;\; 0's$ followed by
- binary representation of j, which requires $\lfloor \log j \rfloor + 1$ bits

Theorem: For each memoryless source P, there holds

$$E_{MTF}(P) \le 1 + 2\,H(P).$$

See: J.L. Bentley, D.D. Sleator, R.E. Tarjan, V.K. Wei. A locally adaptive data compression scheme. CACM 29(4), 320-330.

2.9 Data compression

Theorem: For each memoryless source P, there holds

$$E_{MTF}(P) \leq 1 + 2\,H(P).$$

Proof: Let f(j) = 1 + 2 log j.

Let $e(x_i)$ be the asymptotic expected position of x_i in MTF‘s list.

There holds $E_{MTF}(P) \leq \sum_{i=1}^{n} p_i f\big(e(x_i)\big)$.

$e(x_i)$ = 1 + expected number of items preceding x_i in the list

Item x_j precedes x_i in MTF‘s list if and only if after the last request to x_i item x_j is referenced again. In this case there exists a k ≥ 0 such that the last request to x_i is followed by k requests that are neither to x_i nor to x_j and a request to x_j.

$$\text{Prob}[x_j \text{ precedes } x_i] = \Sigma_{k\geq 0} (1-p_i-p_j)^k\, p_j = p_j/(p_i+p_j)$$

We obtain

$$e(x_i) = 1 + \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{p_j}{p_i + p_j} = \frac{1}{p_i}\left(p_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{p_i p_j}{p_i + p_j} \right) \leq \frac{1}{p_i}\left(p_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} p_j \right) = \frac{1}{p_i}.$$

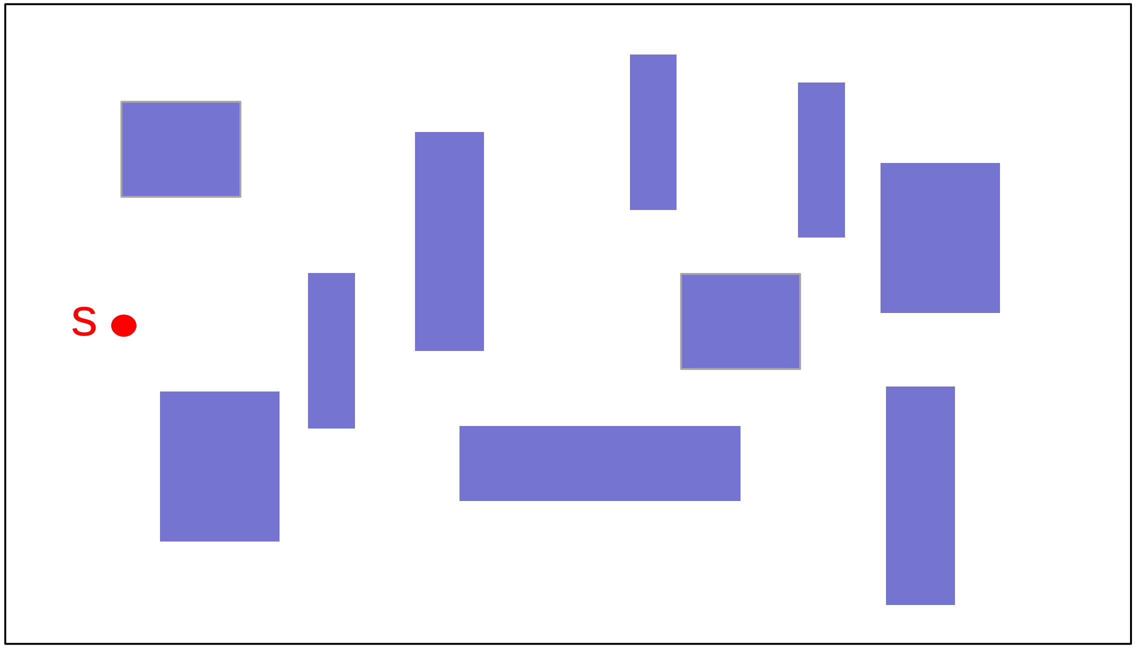
The inequality holds because $p_i/(p_i+p_j) \leq 1$, for any i.

We conclude

$E_{MTF}(P) \leq \sum_{i=1}^{n} p_i f(1/p_i) = \sum_{i=1}^{n} p_i (1 + 2\log(1/p_i)) = 1 + 2\,H(P)$.

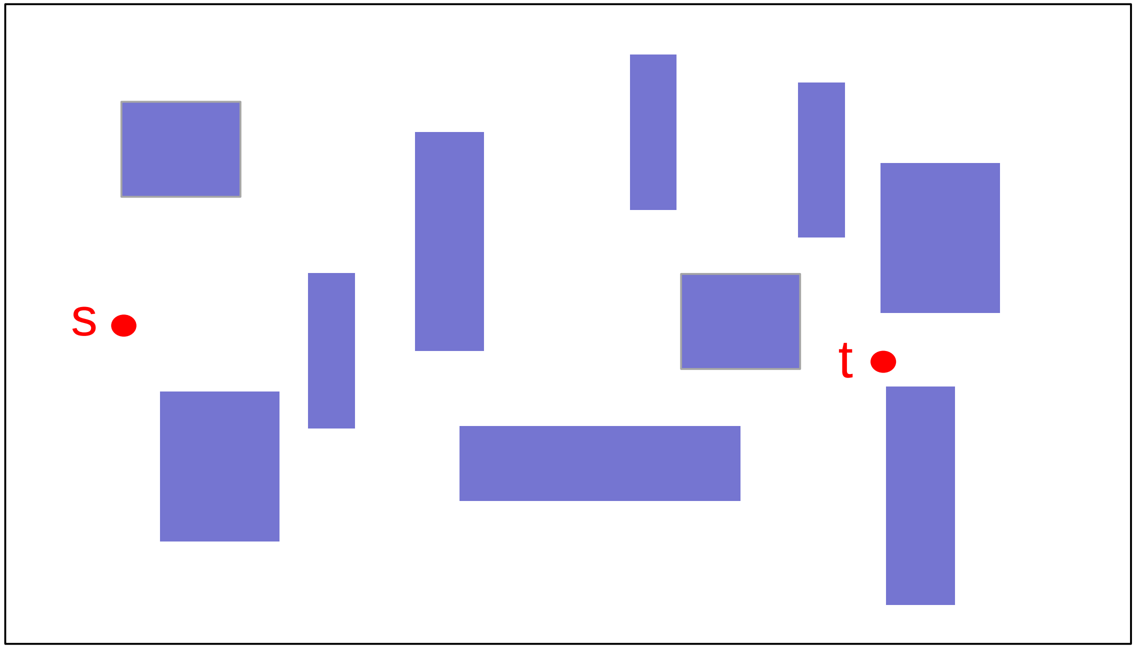
2.10 Robotics

Three problems: Navigation, Exploration, Localization



2.10 Robotics

Navigation: Find a short path from s to t.



Robot always knows its current position and the position of t.
Does not know in advance the position/extent of the obstacles.
Tactile robot: Can touch/sense the obstacles.

2.10 Robot navigation

The material on navigation is taken from the following two papers.

- A. Blum, P. Raghavan, B. Schieber. Navigating in unfamiliar geometric terrain. SIAM J. Comput. 26(1):110-137, 1997.

- R.A. Baeza-Yates, J.C. Culberson, G.J.E. Rawlins. Searching in the plane. Inf. Comput. 106(2):234-252, 1993.

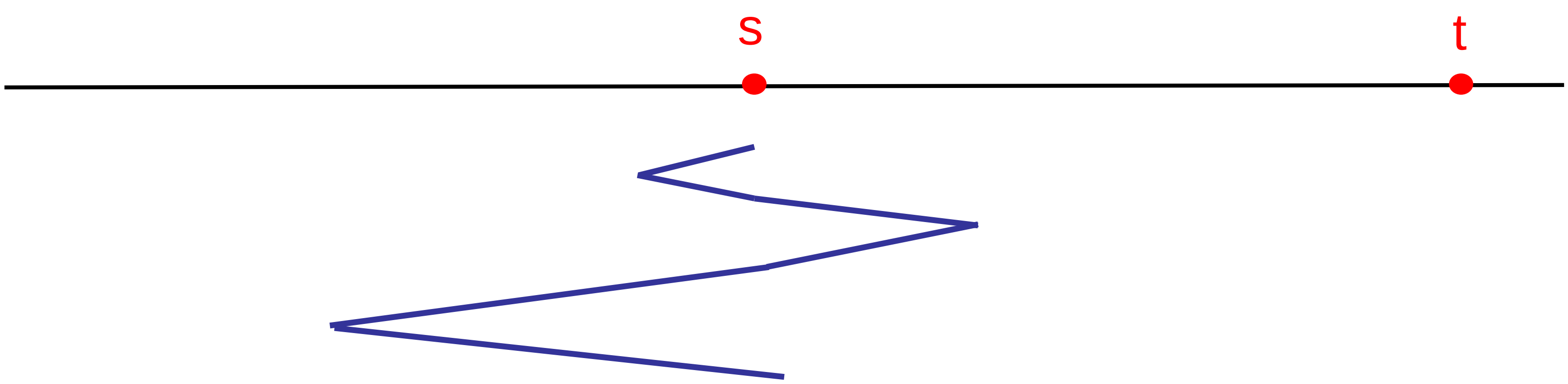
2.10 Navigation on the line

Tactile robot has to find a target t on a line. The position of t is not known in advance.

A Doubling strategy, described on the next page, is 9-competitive.

2.10 Navigation on the line

Doubling strategy: Oscillate around the origin s, with steps to the left and to the right. In iteration i, i ≥ 1, step a distance of 2^{i-1} to the left/right and back to s.



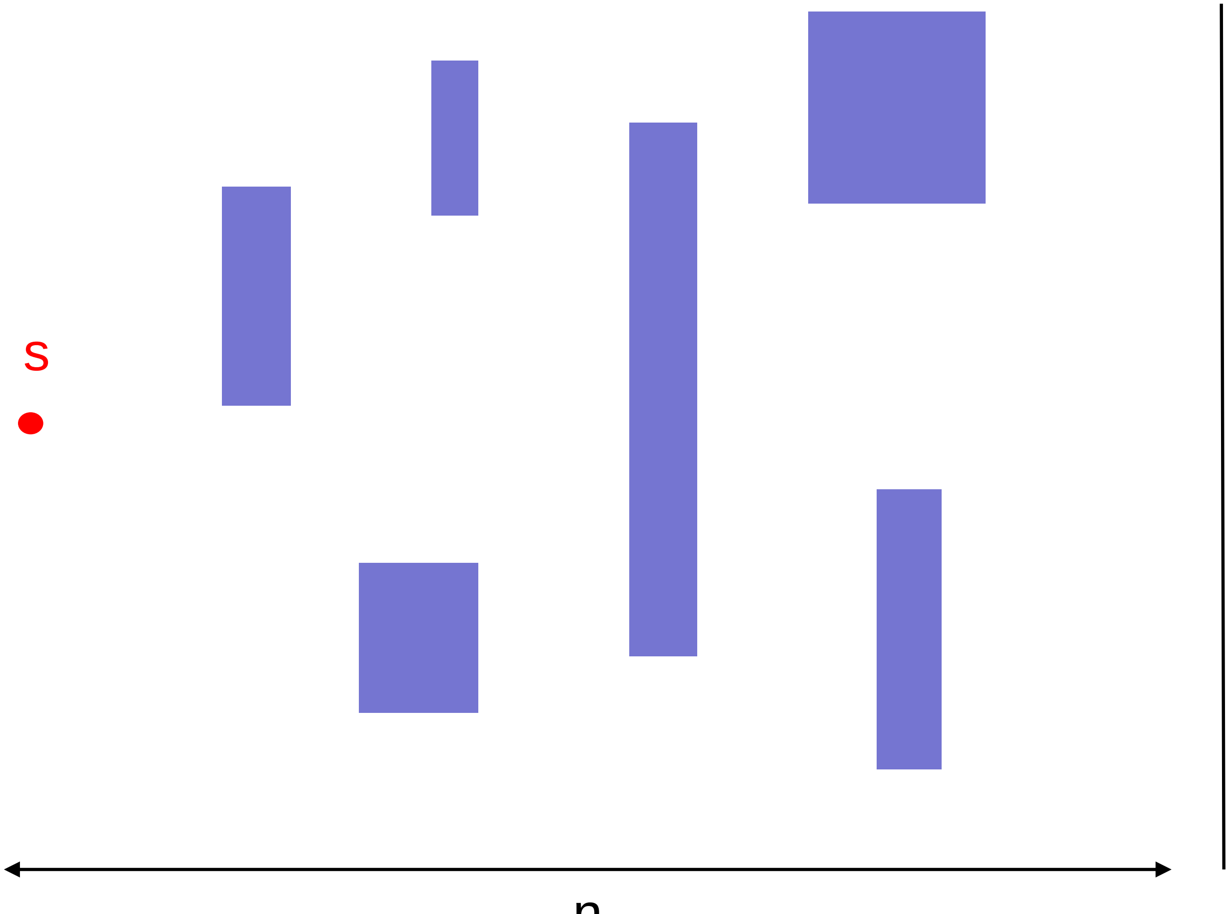
Let n be the distance of t from s. In iteration $\lceil \log n \rceil$ the length of the oscillation is sufficient to reach t. However, the oscillation might be done in the “wrong” direction, opposite of t. Therefore, the total distance traversed is upper bounded by

$$2\sum_{i=0}^{\lceil \log n \rceil} 2^i + n = 2\left(2^{\lceil \log n \rceil + 1} - 1\right) + n < 2 \cdot 2^{\log n + 2} + n = 9n.$$

2.10 Wall problem

Reach some point on a vertical wall that is a distance of n away.

Assumption: Obstacles have a width of at least 1 and are aligned with the axes.



2.10 Wall problem

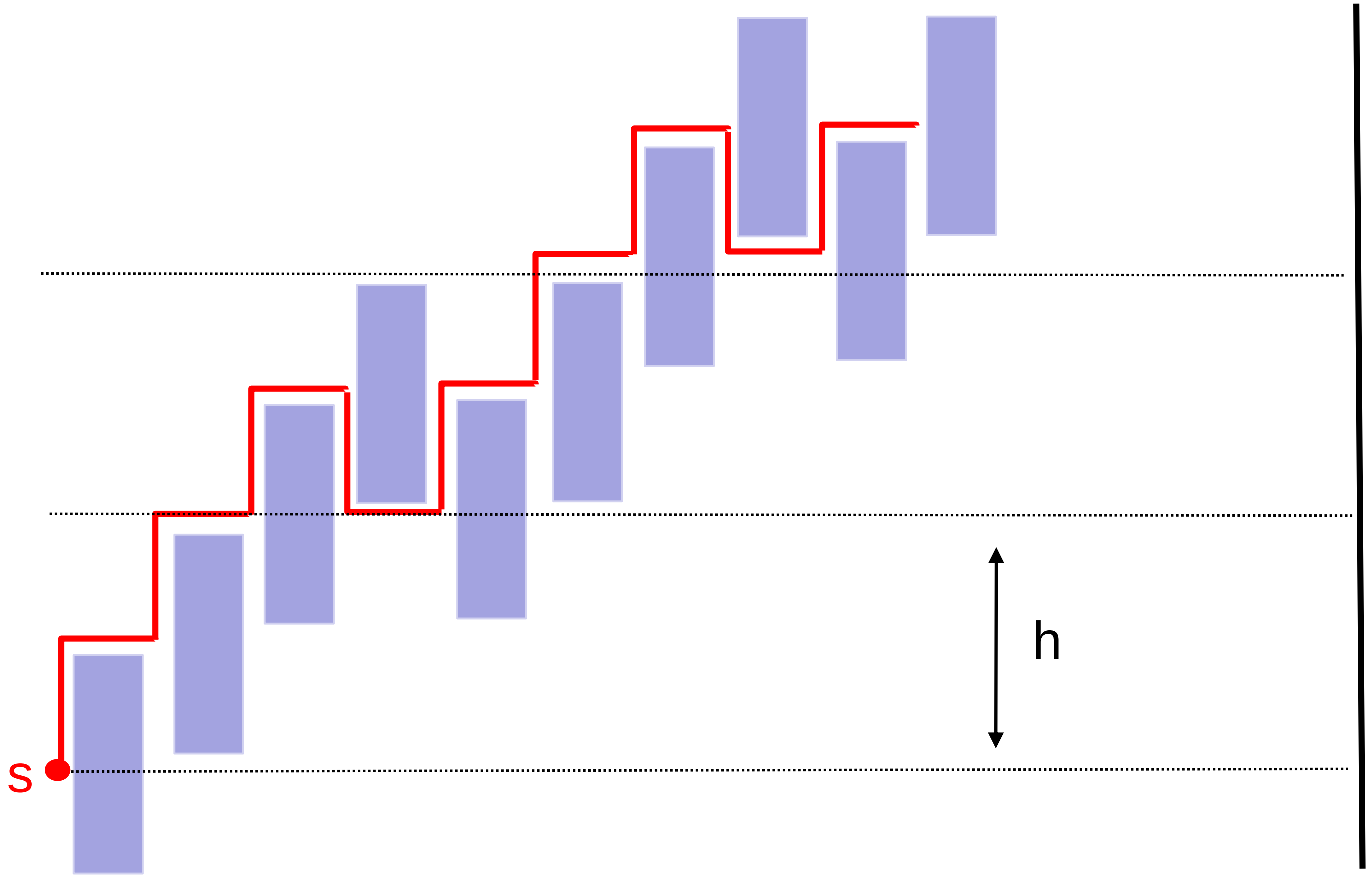
Theorem: Every deterministic online algorithm has a competitive ratio of $\Omega(\sqrt{n})$.

Upper bound: Will design an algorithm with competitive ratio of $O(\sqrt{n})$.

2.10 Wall problem

Theorem: Every deterministic online algorithm has a competitive ratio of $\Omega(\sqrt{n})$.

Proof:



2.10 Wall problem

Starting at s, the adversary places obstacles of height h and width 1 right in front of the robot, whenever it makes a progress of 1 in x-direction. The horizontal distance between neighboring obstacles is ε, where ε>0 is an arbitrarily small value. This way, n-1 obstacles are placed.

L_R = length of the path traversed by the robot
L_{OPT} = length of the optimum path

There holds $L_R \geq (n-1)h/2 > nh/4$, assuming that n >1.

For the analysis of L_{OPT}, partition the scene into corridors of height h starting at s. For each corridor, consider the obstacles that are fully contained in it. One of the next $\lceil \sqrt{n} \rceil$ corridors, starting from s, contains at most $\sqrt{n}$ full obstacles: If each of the next $\lceil \sqrt{n} \rceil$ corridors contained more than $\sqrt{n}$ obstacles, then there would be more than $\lceil \sqrt{n} \rceil \sqrt{n} \geq$ n obstacles in the scene.

2.10 Wall problem

Consider the path that walks to the middle line of this sparse corridor and then moves in x-direction, walking around the at most $\sqrt{n}$ obstacles it hits. This implies $L_{OPT} \leq \lceil \sqrt{n} \rceil h + \sqrt{n} h + n$. The last term accounts for the movement in x-direction.

Setting $h = \sqrt{n}$, there holds

$L_R > n\sqrt{n}/4$ and $L_{OPT} \leq n + \sqrt{n} + n + n \leq 4n$

so that the ratio L_R/L_{OPT} is in $\Omega(\sqrt{n})$.

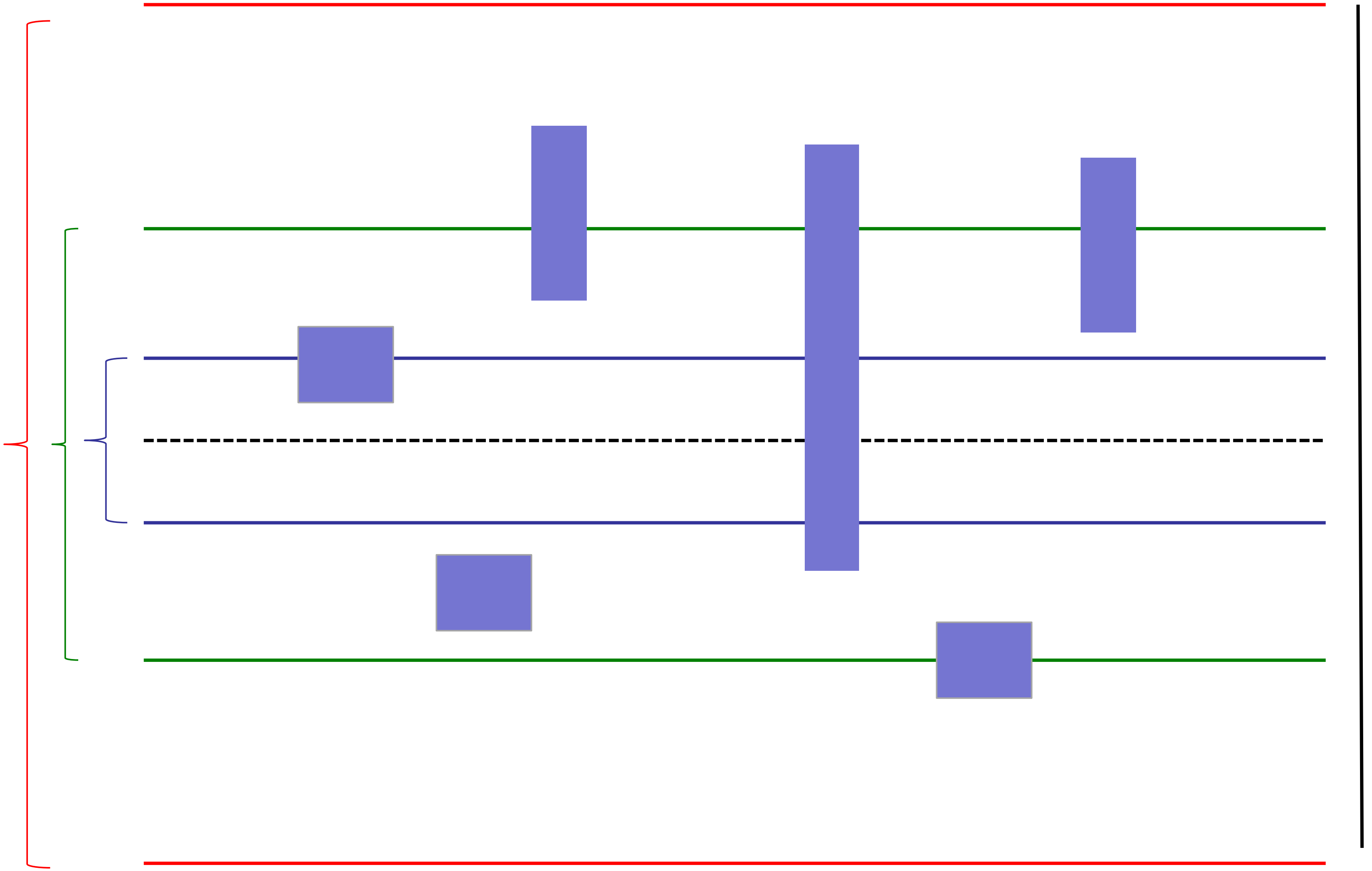
2.10 Wall problem

Upper bound: Design an algorithm with competitive ratio of $O(\sqrt{n})$.

Idea: Try to reach wall within a small window around the origin.

Double the window size whenever the optimal offline algorithm OPT would also have a high cost within the window, i.e. if OPT has a cost of W within the window of size W.

2.10 Wall problem



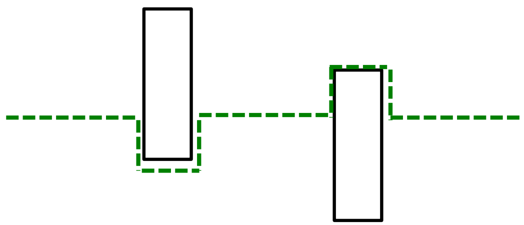
2.10 Wall problem

Window of size W: $W_0 = n$ (boundaries y = +W/2 y = -W/2)
τ := W/$\sqrt{n}$

Sweep direction = north/south
Sweep counter (initially 0)

Always walk in +x direction until obstacle is reached.

Rule 1: Distance to next corner ≤ τ
Walk around obstacle and back to original y-coordinate.



2.10 Wall problem

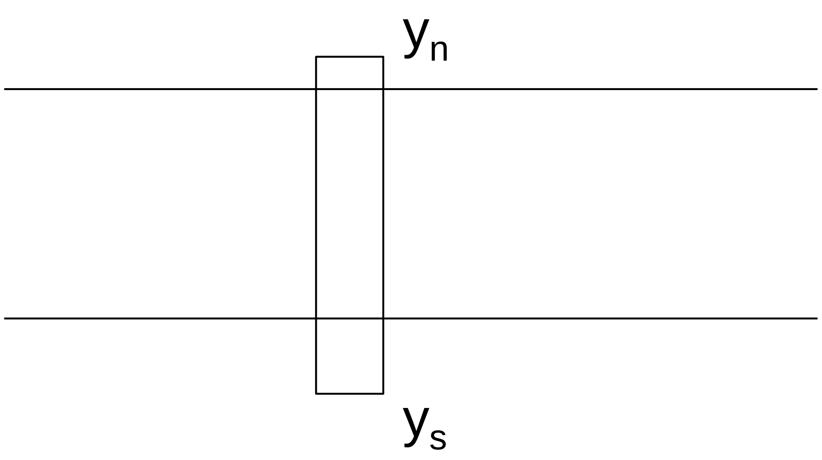
Rule 2: $y_n > W/2$ and $y_s < -W/2$ (y_n and y_s are y-coordinates of northern and southern corners of the obstacle)

$W := 4 \min \{y_n, |y_s|\}$

Walk to next corner within the window.

Sweep counter := 0

Sweep direction := north if at y_s, and south y_n

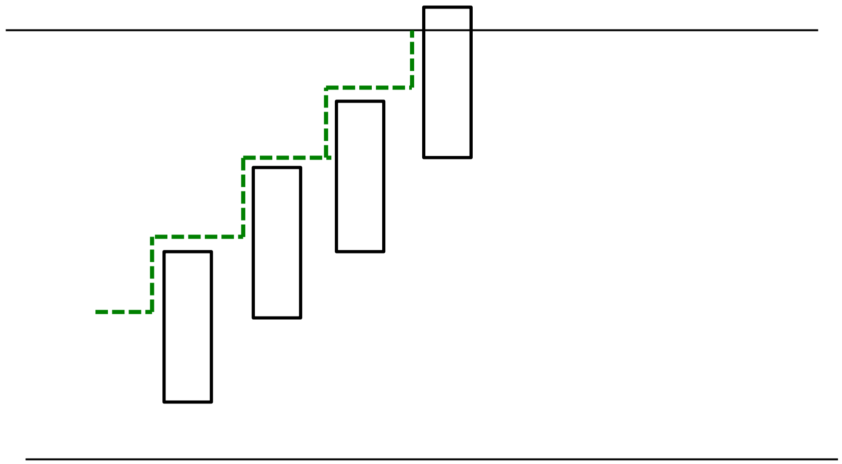


2.10 Wall problem

Rule 3: Distance to nearest corner > τ but $y_n \leq W/2$ or $y_s \geq -W/2$

Walk in sweep direction and then around obstacle.

If window boundary is reached, increase sweep counter by 1 and change sweep direction. If sweep counter reaches $\sqrt{n}$, double window size and set sweep counter to 0.



2.10 Wall problem

Analysis: W_f = last window size

Lemma: Robot walks a total distance of $O(\sqrt{n}\,W_f)$.

Lemma: Length of shortest path is $\Omega(W_f)$.

2.10 Wall problem

Lemma: Robot walks a total distance of $O(\sqrt{n}\,W_f)$.

Proof: The horizontal distance traversed by the robot is n. Hence it suffices to analyze the vertical distance.

Cost of Rule 1: Over all windows, the distance traversed due to Rule 1 is at most $2\cdot \tau_f \cdot n$ because each obstacle has a width of at least 1. Hence the distance is at most $2\cdot W_f / \sqrt{n} \cdot n = 2\sqrt{n}\, W_f$.

Cost of Rules 2 and 3: Consider any fixed window of size W. The distance traversed due to Rule 2 is at most W. As for Rule 3, one sweep costs W so that all the sweeps in the window cost $\sqrt{n}\,W$. Hence for a fixed window, the cost for Rules 2 and 3 is at most $2\sqrt{n}\,W$.

Whenever the window size increases, it is raised by a factor of at least 2. Therefore, over all windows, the total cost of Rules 2 and 3 is at most $2\sqrt{n}(W_f + W_f/2 + W_f/4 + \ldots) \le 4\sqrt{n}\,W_f$.

2.10 Wall problem

Lemma: Length of shortest path is $\Omega(W_f)$.

Proof: If W_f=n, there is nothing to show because the horizontal distance traversed by a shortest path is at least n.

Consider $W_f > n$.

W = largest window in which online robot has executed $\sqrt{n}$ full sweeps

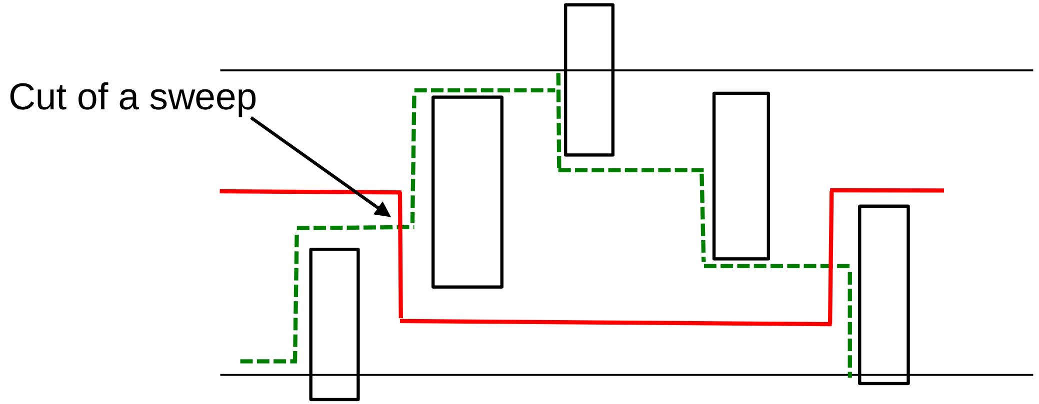
1. No such window exists or $W_f > 2W$.
   In this case W_f was determined according to Rule 2 and thus the length of a shortest path satisfies $L_{OPT} \geq W_f/4$.

2.10 Wall problem

W = largest window in which online robot has executed $\sqrt{n}$ full sweeps

2. $W_f \leq 2W$

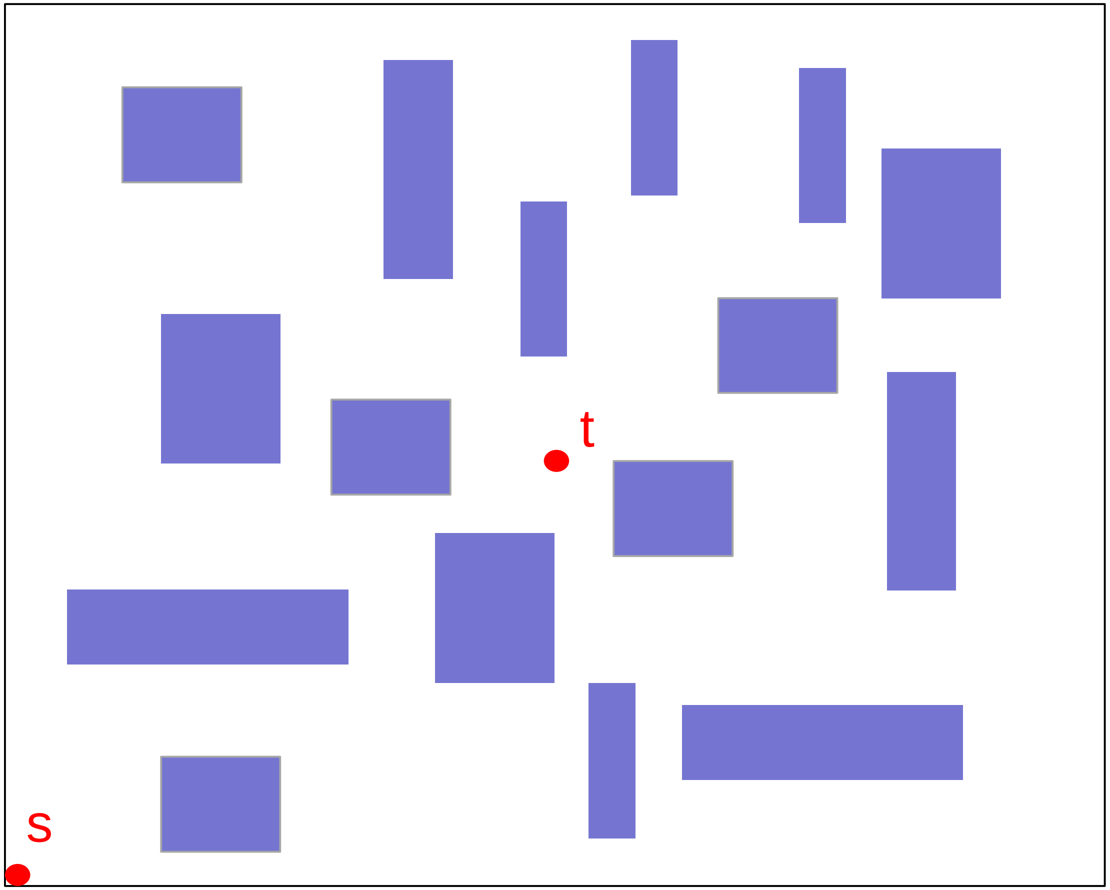
   If a shortest path reaches one of the window boundaries of W, then $L_{OPT} \geq W/2 \geq W_f/4$. If a shortest path does not reach the window boundaries, it has to cut all the $\sqrt{n}$ sweeps. In order to cut one sweep (see figure below), a vertical distance greater than $\tau \geq \tau_f/2$ has to be traversed because the nearest corner of the obstacle hit is at a distance greater than τ. Hence to cut all the sweeps, the total distance traversed is at least $\sqrt{n}\,\tau_f/2 = W_f/2$.



2.10 Room problem

Square room s = lower left corner t = (n,n) center of room

Rectangular obstacles aligned with axes; unit circle can be inscribed into any of them. No obstacle touches a wall.



2.10 Paths

Greedy <+x,+y>: Walk due east, if possible, and due north otherwise.
Paths <+x,-y>, <-x,+y> and <-x,-y> are defined analogously.

Brute-force <+x>: Walk due east. When hitting an obstacle, walk to nearest corner, then around obstacle. Return to original y-coordinate. Path <+y> defined accordingly.

Monotone path from (x_1,y_1) to (x_2,y_2): x- and y-coordinates do not change their monotonicity along the path.

2.10 Algorithm for room problem

Invariant: Robot always knows a monotone path from (x_0,n) to (n,y_0) that touches no obstacle. Initially $x_0 = y_0 = 0$.

In each iteration x_0 or y_0 increases by at least $\sqrt{n}$.

1. Walk to $t' = (x_0+\sqrt{n}, y_0+\sqrt{n})$
   Specifically, walk along monotone path to y-coordinate $y_0+\sqrt{n}$, then brute-force <+x>. If t' is below the monotone path, then walk to point with y-coordinate $y_0+\sqrt{n}$ on the monotone path. If t' is in an obstacle, take its north-east corner or point with y-coordinate equal to n at eastern obstacle boundary.
2. Walk Greedy <+x,+y> until x- or y-coordinate is n. Assume that point $(\hat{x}, \mathrm{n})$ is reached.
3. Walk Greedy <+x,-y> until a point $(n,\hat{y})$ or old monotone path is reached. Gives new monotone path. Set $(x_0,y_0) := (\hat{x},\hat{y})$.

2.10 Algorithm for room problem

4. If $x_0 < n - \sqrt{n}$ and $y_0 < n - \sqrt{n}$, then goto Step 1.
   If $y_0 \geq n - \sqrt{n}$, walk to (x_0,n) and then brute-force <+x>.
   If $x_0 \geq n - \sqrt{n}$, walk to (n,y_0) and then brute-force <+y>.

Theorem: The above algorithm is $O(\sqrt{n})$-competitive.

The algorithm can be generalized to rooms of dimension 2N x 2n, where $N \geq n$ and $t = (N,n)$.

In Step 1, set $t' = (x_0+\sqrt{n}r, y_0+\sqrt{n})$ where $r=N/n$. In Step 4 an x-threshold of $n - \sqrt{n}r$ is considered.

2.10 Algorithm for room problem

Theorem: The above algorithm is O($\sqrt{n}$)-competitive.

Proof: In the analysis of Step 1 we first evaluate the length of the path connecting (a) the point with y-coordinate $y_0+\sqrt{n}$ on the monotone path and (b) t‘.

The robot has to walk around at most $\sqrt{n}$ obstacles because the width of each obstacle is at least 1. When an obstacle is hit, the nearest corner is at a distance of at most $\sqrt{n}$ because no obstacle intersects the monotone path. Thus the length of the path connecting (a) and (b) is upper bounded by $2\sqrt{n}\sqrt{n}+\sqrt{n} \leq 3n$.

All other movements in Step 1 as well as in Steps 2 and 3 are along the monotone path and Greedy paths, each having a length of at most 2n.

Hence each iteration of Steps 1-3 traverses a distance of O(n). The algorithm executes at most $2\sqrt{n}$ iterations. Therefore, the total cost of Steps 1-3 is O($\sqrt{n}$n).

2.10 Algorithm for room problem

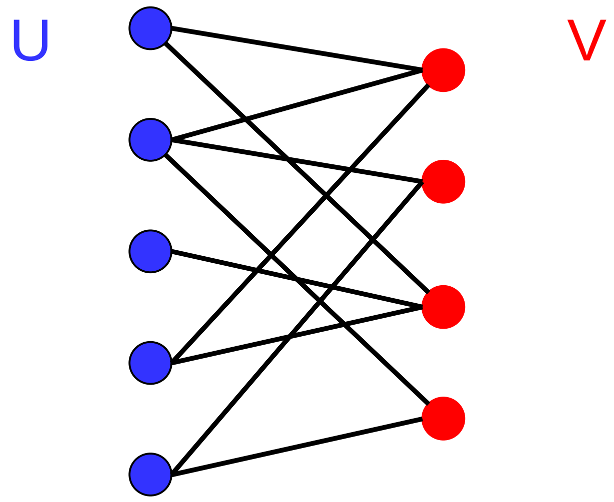
Step 4 is executed only once. The robot has to walk around at most n obstacles. For each obstacle, the nearest corner is at a distancee of at most $\sqrt{n}$. This results in a total cost of at most O($\sqrt{n}$ n).

Since the length of a shortest path is at least n, the theorem follows.

2.11 Bipartite matching

Input: $G = (U \cup V, E)$ undirected bipartite graph.
There holds $U \cap V = \emptyset$ and $E \subseteq U \times V$.

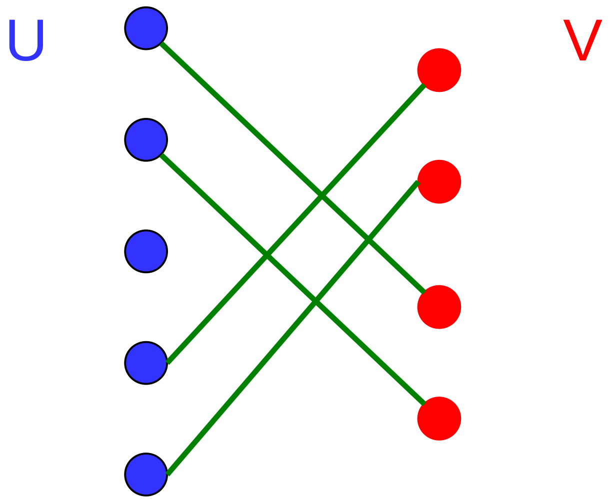
Output: Matching M of maximum cardinality.
$M \subseteq E$ is a matching if no vertex is incident to two edges of M.



2.11 Bipartite matching

Input: G = (U ∪ V, E)

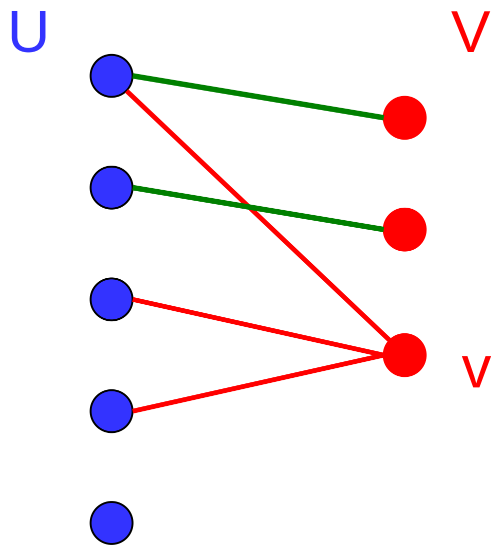
Output: Matching M of maximum cardinality



2.11 Online bipartite matching

U given initially $v \in V$ arrive one by one

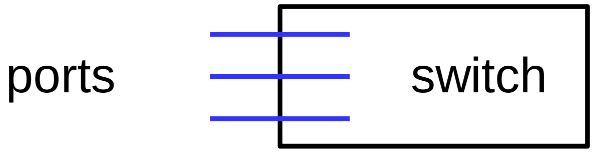
$v \in V$ arrives: neighbors in U are known;
v has to be matched immediately



R.M. Karp, U.V. Vazirani, V.V. Vazirani: An optimal algorithm for on-line bipartite matching. STOC 1990: 352-358.

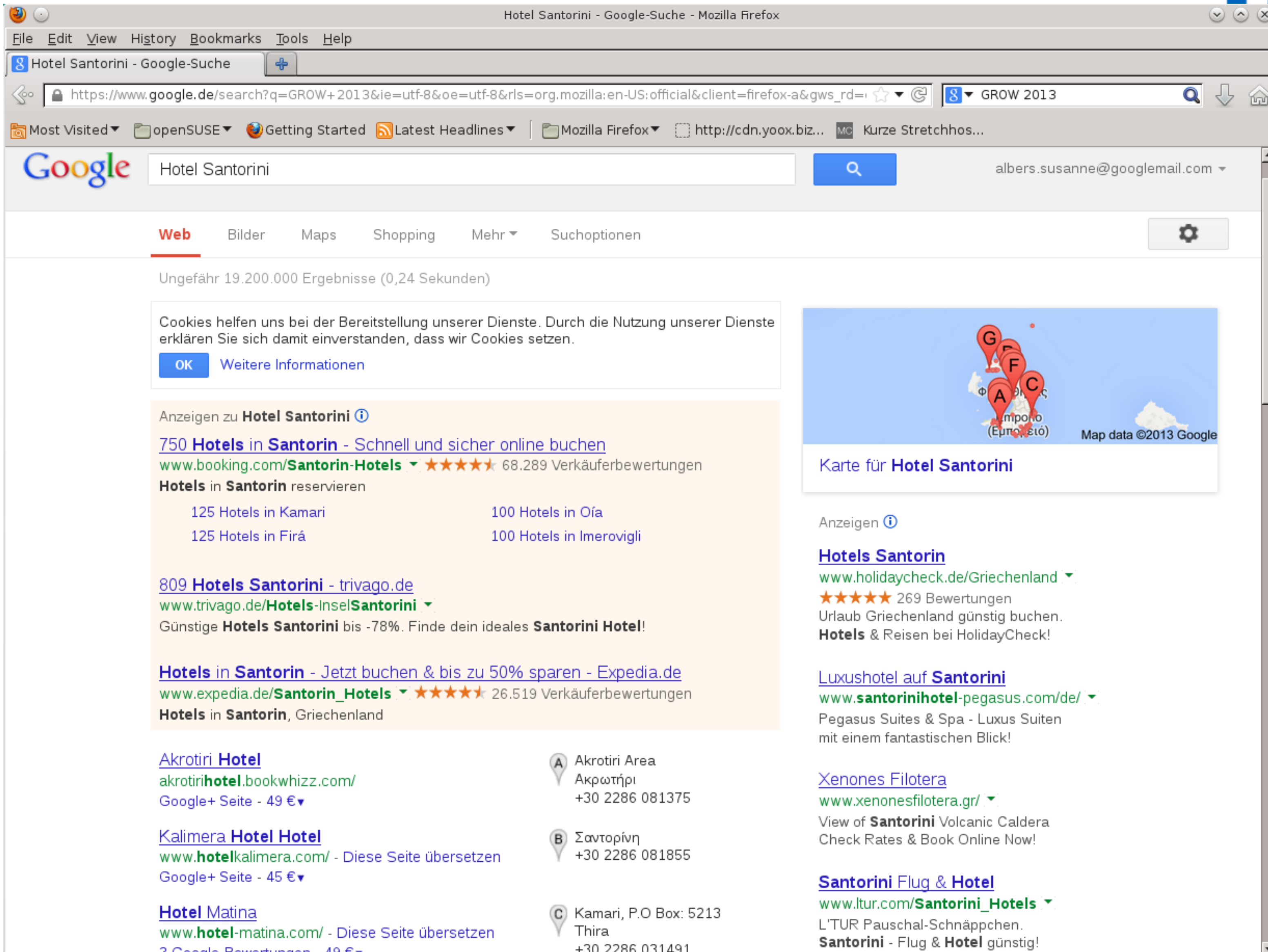
2.11 Applications

- Switch routing: U = set of ports   V = data packets

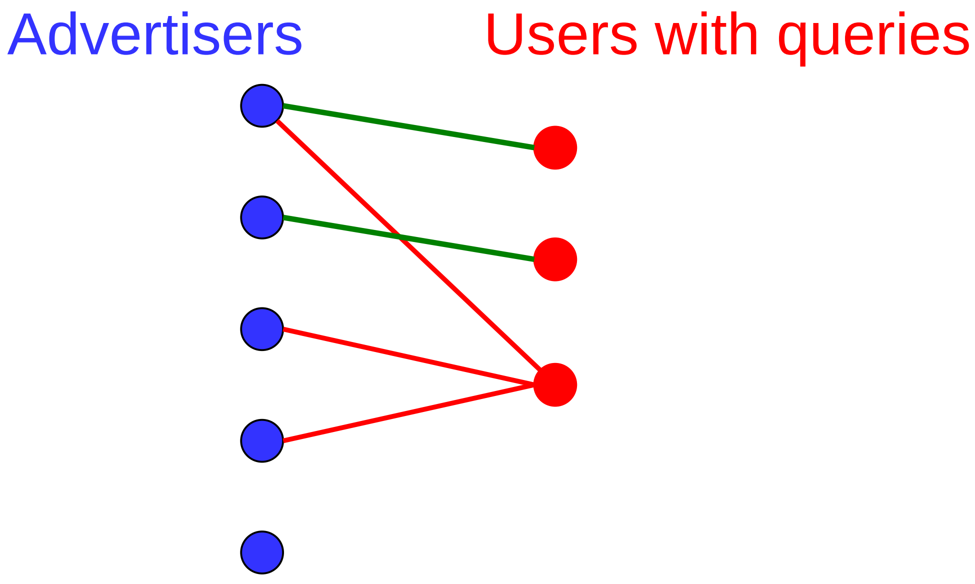


- Market clearing: U = set of sellers   V = set of buyers

- Online advertising: U = advertiser   V= users


Hotel Santorini - Google-Suche - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Hotel Santorini - Google-Suche
https://www.google.de/search?q=GROW+2013&ie=utf-8&oe=utf-8&rls=org.mozilla:en-US:official&client=firefox-a&gws_rd=
GROW 2013
Most Visited openSUSE Getting Started Latest Headlines Mozilla Firefox http://cdn.yoox.biz... Kurze Stretchhos...
Google
Hotel Santorini
albers.susanne@googlemail.com
Web Bilder Maps Shopping Mehr Suchoptionen
Ungefähr 19.200.000 Ergebnisse (0,24 Sekunden)
Cookies helfen uns bei der Bereitstellung unserer Dienste. Durch die Nutzung unserer Dienste erklären Sie sich damit einverstanden, dass wir Cookies setzen.
OK Weitere Informationen
Anzeigen zu Hotel Santorini
750 Hotels in Santorin - Schnell und sicher online buchen
www.booking.com/Santorin-Hotels ★★★★★ 68.289 Verkäuferbewertungen
Hotels in Santorin reservieren
125 Hotels in Kamari
100 Hotels in Oía
125 Hotels in Firá
100 Hotels in Imerovigli
809 Hotels Santorini - trivago.de
www.trivago.de/Hotels-InselSantorini
Günstige Hotels Santorini bis -78%. Finde dein ideales Santorini Hotel!
Hotels in Santorin - Jetzt buchen & bis zu 50% sparen - Expedia.de
www.expedia.de/Santorin_Hotels ★★★★★ 26.519 Verkäuferbewertungen
Hotels in Santorin, Griechenland
Akrotiri Hotel
akrotirihotel.bookwhizz.com/
Google+ Seite - 49 €
A Akrotiri Area
Ακρωτήρι
+30 2286 081375
Kalimera Hotel Hotel
www.hotelkalimera.com/ - Diese Seite übersetzen
Google+ Seite - 45 €
B Σαντορίνη
+30 2286 081855
Hotel Matina
www.hotel-matina.com/ - Diese Seite übersetzen
3 Google-Bewertungen - 49 €
C Kamari, P.O Box: 5213
Thira
+30 2286 031491
Map data ©2013 Google
Karte für Hotel Santorini
Anzeigen
Hotels Santorin
www.holidaycheck.de/Griechenland
★★★★★ 269 Bewertungen
Urlaub Griechenland günstig buchen.
Hotels & Reisen bei HolidayCheck!
Luxushotel auf Santorini
www.santorinihotel-pegasus.com/de/
Pegasus Suites & Spa - Luxus Suiten
mit einem fantastischen Blick!
Xenones Filotera
www.xenonesfilotera.gr/
View of Santorini Volcanic Caldera
Check Rates & Book Online Now!
Santorini Flug & Hotel
www.ltur.com/Santorini_Hotels
L'TUR Pauschal-Schnäppchen.
Santorini - Flug & Hotel günstig!

2.11 Adwords problem



2.11 Adwords problem

- U = set of advertisers B_u = daily budget of advertiser u
- V = sequence of queries v
- r_{uv} = revenue obtained from u when ad is shown to v

Goal: Maximize total revenue, while respecting the budgets.

Unit budgets, unit cost $\Rightarrow$ Online bipartite matching

2.11 Competitive analysis

Maximization problem

	A	⟷	OPT
Cost:	$A(\sigma)$		$OPT(\sigma)$

Online algorithm A is called c-competitive if there exists a constant a, which is independent of σ, such that

$$A(\sigma) \geq c \cdot OPT(\sigma) + a$$

holds for all σ.

2.11 Greedy algorithms

An algorithm has the greedy property if an arriving vertex $v \in V$ is matched if there is an unmatched adjacent vertex $u \in U$ available.

Theorem: Let A be a greedy algorithm. Then its competitive ratio is at least ½.

Proof: $G = (U \cup V, E)$

M_{OPT} = optimal matching

$2|M_{OPT}|$ = number of matched vertices in M_{OPT}

Let $(u,v) \in M_{OPT}$ be arbitrary.

In A's matching M_A at least one of the two vertices is matched.

Hence the number of vertices in A's matching at least $|M_{OPT}|$.

We conclude $|M_A| \geq \lceil ½ \cdot |M_{OPT}| \rceil \geq ½ \cdot |M_{OPT}|$.

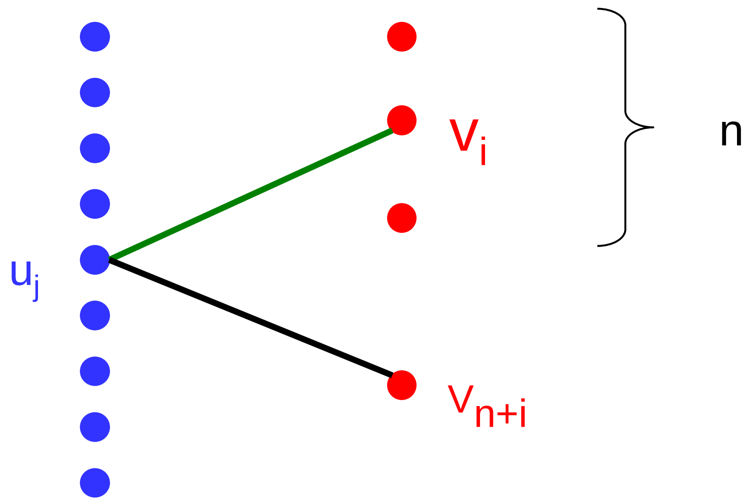
2.11 Deterministic online algorithms

Theorem: Let A be any deterministic algorithm. If A is c-competitive, then $c \leq \frac{1}{2}$.

Proof: $G = (U \cup V, E)$ $|U| = |V| = 2n$ even

$v_1, \ldots, v_n$ adjacent to all $u \in U$

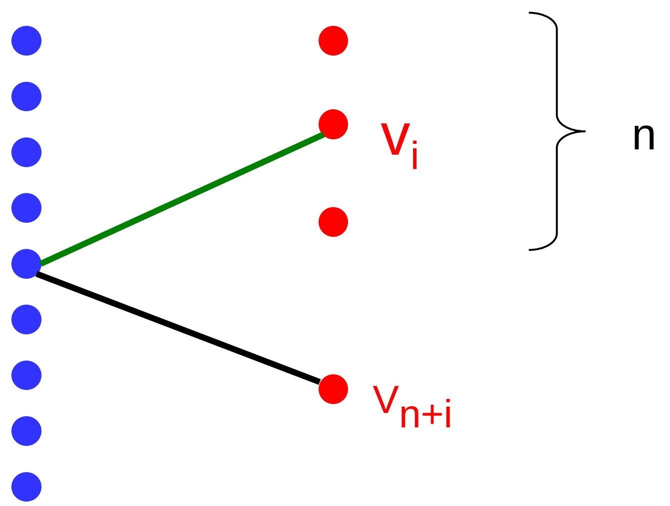
v_{n+i} : If v_i is matched by A to u_j, then v_{n+i} is adjacent to u_j only; otherwise to all $u \in U$



2.11 Deterministic online algorithms

A : $|M_A| \leq n$ — Among v_i and v_{n+i} only one can be matched.

OPT : $|M_{OPT}| = 2n$ — $v_{n+1}, \ldots, v_{2n}$ with 1 neighbor are matched to them.
All other v can be matched arbitrarily.

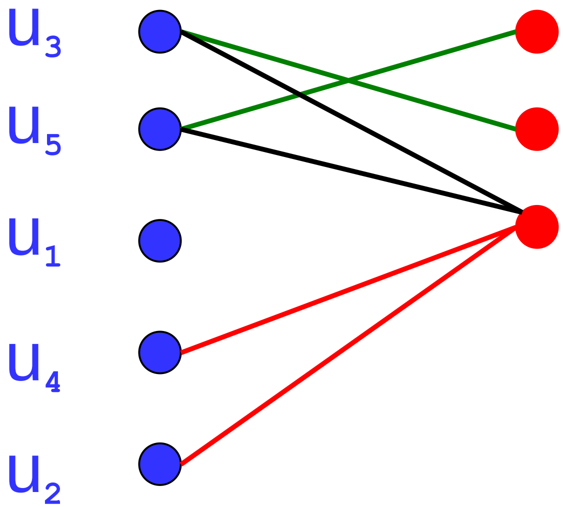


2.11 Ranking algorithm

Init: Choose permutation π of U uniformly at random.

Arrival of $v \in V$: N(v) = set of unmatched neighbors of v

If N(v) ≠ Ø, match v with $u \in N(v)$ of smallest rank, i.e. smallest π(u)-value.



2.11 Analysis of Ranking

Theorem: Ranking achieves a competitive ratio of 1-1/e ≈ 0.632 against oblivious adversaries.

Outline of the analysis:

1. It suffices to consider graphs $G = (U \cup V, E)$ having a perfect matching (each vertex is matched).

2. Analyze Ranking on graphs G with a perfect matching.

2.11 Reduction to G with perfect matching

$G = (U \cup V, E)$ π = permutation of U $w \in U \cup V$

$H = G \setminus \{w\}$

$$\pi_H = \begin{cases} w \in U \rightarrow \text{permutation obtained from } \pi \text{ by deleting } w \\ w \in V \rightarrow \pi \end{cases}$$

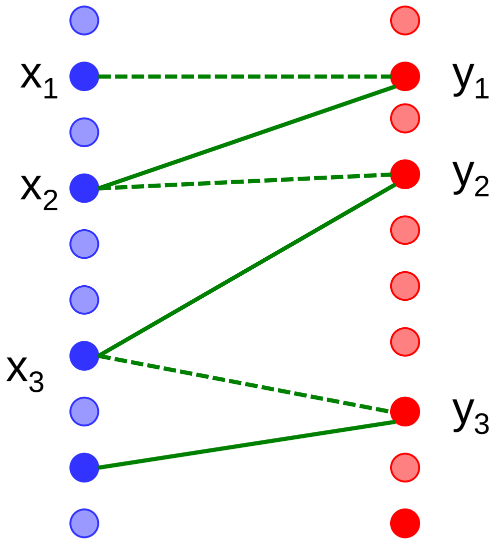
$M = \text{Ranking}(G, \pi)$ $M_H = \text{Ranking}(H, \pi_H)$

Lemma: There holds $|M| \geq |M_H|$.

2.11 Lemma: $|M| \geq |M_H|$

Case 1: $w \in U$

$w = x_1$



y_i matched with x_i in Ranking (G,π)

x_{i+1} matched with y_i in Ranking (H, π_H)

Process stops with

x_k not matched in Ranking (G, π)

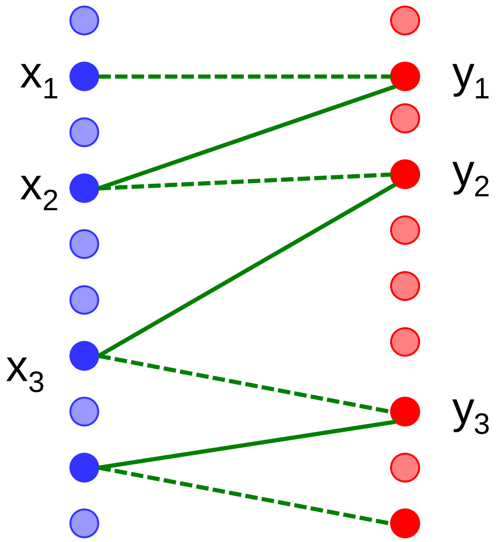
$\rightarrow |M_H| = |M|$

y_k not matched in Ranking (H, π_H)

$\rightarrow |M_H| = |M| - 1$

2.11 Lemma: $|M| \geq |M_H|$

Case 1: $w \in U$ $\quad w = x_1$



y_i matched with x_i in Ranking (G,π)

x_{i+1} matched with y_i in Ranking (H, π_H)

Process stops with

x_k not matched in Ranking (G, π)

$\rightarrow |M_H| = |M|$

y_k not matched in Ranking (H, π_H)

$\rightarrow |M_H| = |M| - 1$

Case 2: $w \in V$ analogous

2.11 Reduction to G with perfect matching

Corollary: The competitive ratio of Ranking is assumed on graphs G having a perfect matching.

Proof: $G = (U \cup V, E)$ arbitrary

M_{OPT} = optimum matching for G

H = obtained from G by deleting all vertices not in M_{OPT}

$\forall \pi \qquad |Ranking(G, \pi)| \geq |Ranking(H, \pi_H)|$

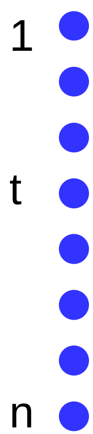
$$E[|Ranking(G)|] \geq E[|Ranking(H)|]$$

M_{OPT} is an optimum matching for both G and H.

2.11 Analysis on G with perfect matching

$|U| = |V| = n$ $\qquad$ $t \in \{1, \ldots n\}$

p_t = probability (over all π) that vertex of rank t in U is matched by Ranking



$E[|\text{Ranking}(G)|] = \sum_{1 \le t \le n} p_t$

Main Lemma: $1 - p_t \le 1/n \cdot \sum_{1 \le s \le t} p_s$

2.11 Main theorem

Theorem: Ranking achieves competitive ratio of 1-1/e.

Proof: $E[|\text{Ranking}(G)|] / |\text{OPT}(G)| = 1/n \cdot \sum_{1 \leq t \leq n} p_t$

Determine the infimum of $1/n \cdot \sum_{1 \leq t \leq n} p_t$

Main Lemma implies $1 + S_{t-1} \leq S_t (1 + 1/n)$ $\quad S_t = \sum_{1 \leq s \leq t} p_s$

$S_t = \sum_{1 \leq s \leq t} (1-1/(n+1))^s$ solves inequality with equality

Main Lemma: $1 - p_t \leq 1/n \cdot \sum_{1 \leq s \leq t} p_s$

2.11 Verifying solution to recurrence relation

$$S_t\left(1+\frac{1}{n}\right)=\sum_{1\le s\le t}\left(1-\frac{1}{n+1}\right)^s\left(1+\frac{1}{n}\right)$$

$$=\sum_{1\le s\le t-1}\left(1-\frac{1}{n+1}\right)^s+\left(1-\frac{1}{n+1}\right)^t$$

$$+\left(1-\left(1-\frac{1}{n+1}\right)^t\right)\left(1+\frac{1}{n}\right)-\frac{1}{n}+\frac{1}{n}\left(1-\frac{1}{n+1}\right)^t$$

$$=S_{t-1}+1$$

2.11 Calculating the competitive ratio

$$\frac{1}{n} S_n = \frac{1}{n} \sum_{1 \le s \le n} \left(1 - \frac{1}{n+1}\right)^s$$

$$= \left(1 - \left(1 - \frac{1}{n+1}\right)^n\right)\left(1 + \frac{1}{n}\right) - \frac{1}{n} + \frac{1}{n}\left(1 - \frac{1}{n+1}\right)^n$$

$$= 1 - \left(1 - 1/(n+1)\right)^n \xrightarrow{n \to \infty} 1 - 1/e$$

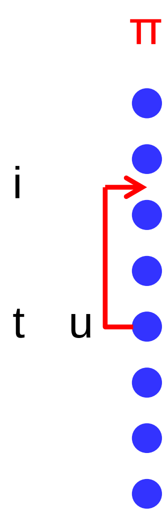
2.11 Establishing the Main Lemma

$G = (U \cup V, E)$ $|U| = |V| = n$

M^* = perfect matching

Fix π and $(u,v) \in M^*$ such that u has rank t in π.

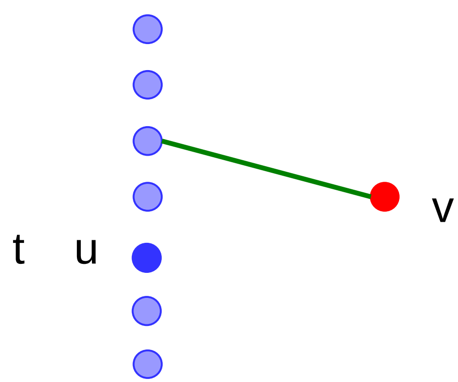
π_i = permutation in which u is reinserted so that its rank is i $1 \leq i \leq n$



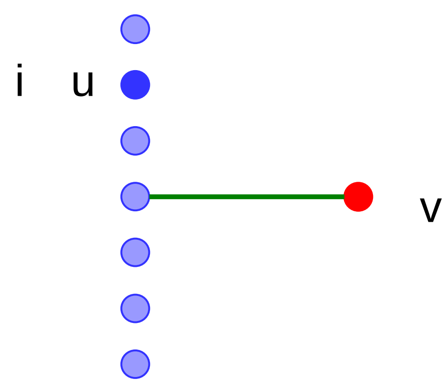
2.11 Claim

Claim: If u is not matched in Ranking (π), then for i = 1,…, n,

v is matched to a vertex of rank at most t in π_i in Ranking (π_i).



Ranking(π)



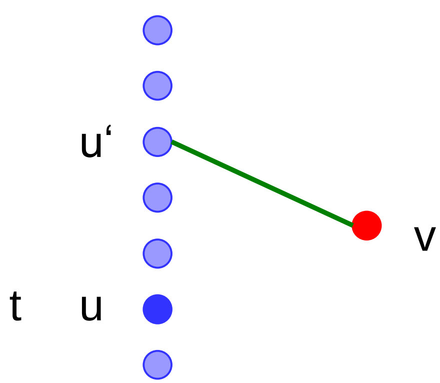
Ranking(π_i)

2.11 Proof of the Claim

X = { unmatched vertices with rank < t in π when Ranking executed with π }

X_i = { unmatched vertices with rank < t in π when Ranking executed with π_i }

Invariant: $X \subseteq X_i$ holds at any time before the arrival of v.



u' = partner of v in Ranking(π), has rank < t in π

Invariant → when v arrives in Ranking(π_i), u' $\in X_i$

and hence u' is available for a matching with v

u' has rank ≤ t in π_i

2.11 Proof of the invariant

Invariant: $X \subseteq X_i$ holds at any time before the arrival of v.

Proof: By induction on the vertex arrivals.

Assume that $X \subseteq X_i$ holds before the arrival of $y \in V$.

Invariant can only be violated if y matched in Ranking(π_i) to some $x_i \in X_i \cap X$.

In this case y also gets matched in Ranking(π) to some $x \in X$.

Suppose that $x \neq x_i$. This implies:

x_i has smaller rank than x in π_i.

x has smaller rank than x_i in π.

Observe that all vertices, different from that of rank t in π, occur in the same relative order in both π and π_i. Hence the vertices contained X_i and X occur in the same relative order in both π and π_i. Therefore we obtain a contradiction.

2.11 Establishing the Main Lemma

Main Lemma: $1 - p_t \leq 1/n \cdot \sum_{1\leq s \leq t} p_s$

Proof: For each π construct a set S_π .

u = vertex of rank t in π v = vertex matched u in M*

$S_\pi = \{ (\pi_i, v) \mid 1 \leq i \leq n \}$

S_π gets labeled if, for i = 1,…, n, v is matched to a vertex of rank at most t in π_i when Ranking (π_i) is executed.

Claim $\Rightarrow$ If u is not matched in Ranking(π), then S_π gets labeled.

Claim: If u not matched in Ranking (π), then for i = 1,…, n, v is matched in Ranking (π_i) to u_i of rank at most t in π_i.

2.11 Establishing the Main Lemma

This implies

$$1 - p_t \leq \#\text{ labeled sets } S_\pi \,/\, n! = \sum_{\pi\in P} |S_\pi| / (n \cdot n!)$$

where $P = \{\pi \mid S_\pi \text{ is labeled }\}$.

Proposition: Elements in all the sets S_π with $\pi \in P$ are distinct.

Using the above proposition we obtain:

$$1 - p_t \leq \sum_{\pi\in P} |S_\pi| / (n \cdot n!) = |\bigcup_{\pi\in P} S_\pi| / (n \cdot n!)$$

2.11 Establishing the Main Lemma

For any π', count occurrences of π' in $|\cup_{\pi\in P} S_\pi|$: (π',v_1) (π',v_2) (π',v_3) …

#occurrences of π' in $|\cup_{\pi\in P} S_\pi| \leq$ #v being matched to vertex of rank ≤ t in π'

= |R (π')|

R (π') = { vertices of rank ≤ t in U being matched in Ranking(π') }

2.11 Establishing the Main Lemma

$R(\pi') = \{$ vertices of rank $\leq t$ in U being matched in Ranking(π') $\}$

We conclude:

$$1 - p_t \leq |\mathrm{U}_{\pi \in P} S_\pi| / (n \cdot n!) \leq \sum_{\pi'} |R(\pi')| / (n \cdot n!)$$

$$= 1/n \cdot \sum_{\pi} |R(\pi)| / n!$$

$$= 1/n \cdot \sum_{1 \leq s \leq t} p_s$$

The last inequality holds because $\sum_\pi |R(\pi)| / n!$ is the expected number of vertices of rank $\leq t$ matched in Ranking, and this quantity is exactly $\Sigma_{1 \leq s \leq t}\, p_s$.

2.11 Proof of the Proposition

Proposition: Elements in all the sets S_π with $\pi \in P$ are distinct.

Proof: For a fixed π, elements of $S_\pi = \{ (\pi_i ,v) \mid 1 \le i \le n \}$ are distinct because they differ in the first component.

Suppose that $(\pi_i ,v) = (\pi'_j ,v)$ where $(\pi_i ,v) \in S_\pi$ $(\pi'_j ,v) \in S_{\pi'}$.

Let u be the vertex matched to v in M*.

Removing u in π_i and π'_j and reinserting it at position t, we obtain identical permutation, i.e. $\pi = \pi'$.

2.12 Financial Games

Online search: Find maximum/minimum in a sequence of prices that are revealed sequentially.

Period i: Price p_i is revealed. If p_i is accepted, then the reward is p_i; otherwise the game continues.

Application: job search, selling of a house.

One-way trading: An initial wealth of D_0, given in one currency has to be traded to some other asset or currency.

Period i: Price/exchange rate p_i is revealed. Trader must decide on the fraction of the remaining initial wealth to be exchanged.

2.12 Financial Games

Portfolio selection: s securities (assets) such as stocks, bonds, foreign currencies or commodities

Period i: price vector $\vec{p_i}$= (p_{i1}, …, p_{is})

p_{ij} = # units of the j-th asset that can be bought for 1$

vector of price changes $\vec{x_i}$= (x_{i1}, …, x_{is})

x_{ij}= $p_{ij}/p_{i+1,j}$

Portfolio: specifies a distribution of the wealth on the s assets just before period i

$\vec{b_i}$= (b_{i1}, …, b_{is}) and Σb_{ij}=1

At the end of first period the wealth per initial 1$ is $\sum_{j=1}^{s} b_{1j}\, x_{1j}$

2.12 Relation between search and trading

Any deterministic (randomized) one-way trading algorithm, that may trade the initial wealth in parts, can be viewed as a randomized search algorithm, and vice versa.

Theorem: a) Let A_1 be a randomized algorithm for one-way trading. Then there exists a deterministic algorithm A_2 for one-way trading such that $A_2(\sigma) = E[A_1(\sigma)]$, for all price sequences σ.

b) Let A_2 be a deterministic algorithm for one-way trading. Then there exists a randomized search algorithm A_3 such that $E[A_3(\sigma)] = A_2(\sigma)$, for all σ.

2.12 Relation between search and trading

Theorem: a) Let A_1 be a randomized algorithm for one-way trading. Then there exists a deterministic algorithm A_2 for one-way trading such that $A_2(\sigma) = E[A_1(\sigma)]$, for all price sequences σ.

b) Let A_2 be a deterministic algorithm for one-way trading. Then there exists a randomized search algorithm A_3 such that $E[A_3(\sigma)] = A_2(\sigma)$, for all σ.

Proof Idea: a) Any one-way trading algorithm is equivalent, in term of expected return, to a randomized one-way trading algorithm that trades the initial wealth at one randomly chosen period.

Any randomized one-way trading algorithm that trades at once is equivalent to a deterministic trading algorithm that trades the initial wealth in parts.

2.12 Search problems

Will concentrate on search problems.

Prices in $[m,M]$ $0 < m \le M$ $\varphi := M/m$

Discrete time, finite time horizon, n periods; both m and M are known to player.

Online algorithm is c-competitive if there exists a constant a such that

$$c\,A(\sigma) + a \ge OPT(\sigma)$$

for all price sequences.

2.12 Algorithms

Algorithm Reservation Price Policy (RPP): Accept first price of value at least $p^* := \sqrt{Mm}$. Here p^* is called the reservation price.

Theorem: RPP is $\sqrt{\varphi}$-competitive.

Algorithm EXPO: Let $\varphi = 2^k$ for some positive integer k.

RPP_i = deterministic RPP with price $m\,2^i$.

With probability 1/k, choose RPP_i for i=1, …, k.

Theorem: EXPO is $c(\varphi)\log \varphi$-competitive, where $c(\varphi)$ tends to 1 as $\varphi \to \infty$.

2.12 Algorithm RPP

Theorem: RPP is $\sqrt{\varphi}$-competitive.

Proof: Consider any price sequence σ and let p_{max} be the maximum price revealed.

- $p_{max} \geq p^*$: $c = OPT(\sigma) / RPP(\sigma) \leq M / p^* = \sqrt{M/m}$
- $p_{max} < p^*$: $c = OPT(\sigma) / RPP(\sigma) \leq p_{max} / m < p^* / m = \sqrt{M/m}$

2.12 Algorithm EXPO

Theorem: EXPO is $c(\varphi)\log \varphi$-competitive, where $c(\varphi)$ tends to 1 as $\varphi \to \infty$.

Proof: Let σ be any price sequence and p_{max} be the maximum price revealed.

We first focus on the case that $p_{max} < M$.

Let $j \in \{0,\ldots k-1\}$ be the integer such that $m2^j \leq p_{max} < m2^{j+1}$.

We modify σ so that the ratio OPT's return / EXPO's return can only increase.

1. Immediately before p_{max} price $m2^j$ is revealed.
   If EXPO chooses a reservation price of at most $m2^j$, then its return can only decrease. Otherwise the addition of $m2^j$ has no effect.
2. $p_{max} := m2^{j+1} - \varepsilon$, for arbitrarily small $\varepsilon > 0$
   This increases OPT's return while EXPO will not get p_{max}, given modification 1.
3. Every price $p < p_{max}$ with $m2^i \leq p < m2^{i+1}$ is reduced to $m2^i$.

2.12 Algorithm EXPO

Let $m2^i$, $1 \le i \le k$, be the reservation price selected by EXPO.

If $i \le j$, then EXPO's return is at least $m2^i$.
If $i > j$, then EXPO's return is at least m.
Hence the expected return of EXPO is at least

$$m(k-j)/k + \Sigma_{1 \le i \le j}\, m2^i/k = m(2^{j+1}+k-j-2)/k.$$

Since $p_{max} < m2^{j+1}$, the ratio OPT's return / EXPO's return is upper bounded by

$$k \frac{2^{j+1}}{2^{j+1}+k-j-2}.$$

This expression is maximized for $j^* = k - 2 + 1 / \ln 2$ and, for this setting, approaches $k = \log \varphi$ as φ grows.

Finally, assume that $p_{max} = M$, and recall that $M = m2^k$. In this case a worst-case price sequence consists of all the reservation prices revealed in increasing order. The expected return of EXPO is $\Sigma_{1 \le i \le k}\, m2^i/k = m(2^{k+1}-2)/k$. The ratio OPT's return / EXPO's return is upper bounded by $k2^k/(2^{k+1}-2) \le k$.

2.13 k-server problem

Metric space M; k mobile servers; request sequence σ.

Request: $x \in M$; one of the k servers must be moved to x, if the point is not already covered. Moving a server from y to x incurs a cost of dist(y,x).

Goal: Minimize total distance traveled by all the servers in processing σ.

Special cases: Paging; caching fonts in printers; vehicle routing.

Results: General metric spaces:

Deterministic: $k \leq c \leq 2k-1$

Randomized: $\Omega(\log k) \leq c \leq \tilde{O}(\log^2 k \log^3 n)$, where n is size of M.

Special metric spaces:

Competitive ratio of k for lines, trees, spaces of size N=k+1 and resistive spaces (modeling electrical networks).

2.13 k-server problem

Theorem: Let M be a metric space consisting of at least k+1 points and let A be a deterministic online algorithm. If A is c-competitive, then $c \geq k$.

Trees: Will restrict ourselves on metric spaces that are trees.

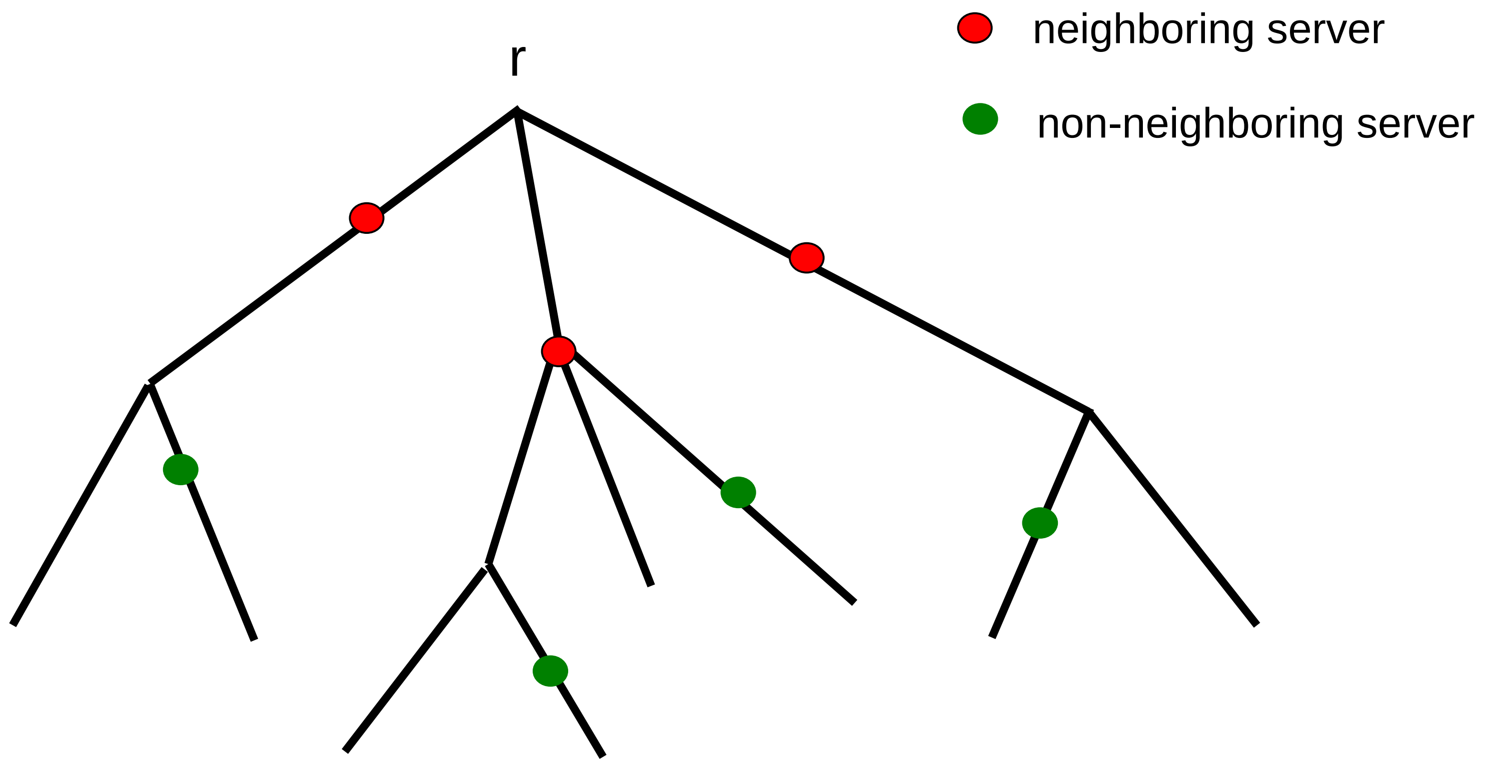
Consider a request at point r. Server s_i is a neighbor if no other server is located between s_i and r.

Algorithm Coverage: In response to a request at r, move all neighboring servers with equal speed in the direction of r until one server reaches r.

Theorem: Coverage is k-competitive.

2.13 Algorithm Coverage

Metric space can be laid out such that a request occurs at the root of the tree.



2.13 Analysis Coverage

Theorem: Coverage is k-competitive.

Proof: Potential function $\Phi = k\, M_{min} + D$

M_{min} = value of min-cost matching between Coverage's servers and OPT's servers

$D = \Sigma_{i<j}\, \text{dist}\,(s_i,s_j)$ $s_1,\ldots,s_k$ Coverage's servers

Given any σ, we analyze the amortized cost of a request σ(t) at point r.

- OPT moves servers at a total distance of d:
  Actual cost: d $\Delta\Phi \leq k\cdot d$

- Coverage moves m servers a distance of d' each.
  Actual cost: $m\cdot d'$

  $\Delta\Phi$ in M_{min}: W.l.o.g. one neighboring server is matched to OPT's server at r. For this server pair, the matching distance decreases by d', while for the other m-1 server pairs the distance may increase by d' each.

$$\Delta\Phi \leq -k\cdot d' + (m-1)\cdot k\cdot d' = (m-2)\cdot k\cdot d'$$

2.13 Analysis Coverage

$\Delta\Phi$ in D: Consider a non-neighboring server.

One neighboring server moves away, m-1 come closer.

For all the k-m non-neighboring servers:

$$\Delta\Phi \le (d' - (m-1)d')\,(k-m) = -(m-2)d'(k-m) = -(m-2)\,k\,d' + (m-2)\,m\,d'$$

Consider the $\binom{m}{2}$ pairs of neighboring servers.
For each pair, the distance decreases by 2d'.

$$\Delta\Phi = -2d'\,m(m-1)/2 = -d'\,m\,(m-1)$$

Let C(σ(t)) denote the actual cost incurred by Coverage. Then

$$C(\sigma(t)) + \Delta\Phi \le m\cdot d' + k\cdot d + (m-2)\cdot k\cdot d' - (m-2)\,k\,d' + (m-2)\,m\,d' - d'\,m\,(m-1)$$

$$= k\cdot d = k\cdot OPT(\sigma(t)).$$

2.14 Metrical task systems

$(\mathcal{M}, \mathcal{R})$ $\mathcal{M}$ = (M,dist) metric space $\mathcal{R}$ = set of allowed tasks

M: set of states in which an algorithm can reside |M| = N

dist(i,j) = cost of moving from state i to state j

$r \in \mathcal{R}$: r = (r(1), …, r(N))

r(i) $\in \mathbb{R}_0^+ \cup \{\infty\}$ cost of serving task in state i

Algorithm A: Initial state 0.

Sequence of requests/tasks: $\sigma = r_1, \ldots, r_n$.

Upon the arrival of r_i, A may first change state and then has to serve r_i.

A[i] : state in which r_i is served.

$$A(\sigma) = \sum_{i=1}^{n} dist(A[i-1], A[i]) + \sum_{i=1}^{n} r_i(A[i])$$

2.14 Example: paging

Pages $p_1, \ldots, p_n$; fast memory of size k.

Sets $S_1, \ldots S_l$, where $l = \binom{n}{k}$. Each set is a subset of $\{p_1, \ldots, p_n\}$ having size k.

For each set S_i, there is a state s_i, $i = 1, \ldots, \binom{n}{k}$

$\mathrm{dist}(s_i, s_j) = |S_j \setminus S_i|$

Request r = p

$$r(s_i) = \begin{cases} 0 & \text{if } p \in S_i \\ \infty & \text{otherwise} \end{cases}$$

2.14 Example: list update

List consisting of n items.

n! states s_i, where $1 \leq i \leq n!$, for each possible permutation of the n items.

dist(s_i,s_j) = number of paid exchanges needed to transform the two lists
(We may assume w.l.o.g. that algorithm only works with paid exchanges.)

Request r = x
$r(s_i)$ = position of item x in list s_i.

2.14 Results for metrical task systems

Deterministic: $c = 2N - 1$

Randomized: $\Omega(\log N / \log\log N) \leq c \leq O(\log^2 N \log\log N)$

Approximation Algorithms

3.1 Basics

NP-hard optimization problems: Computation of approximate solutions

Example: Job scheduling. m identical parallel machines.

n jobs with processing times $p_1,\ldots, p_n$. Assign the jobs to machines so that the makespan is as small as possible.

List scheduling: Assign each job to a least loaded machine. (2-1/m)-approximation.

General setting: Optimization problem Π, P = set of problem instances

For problem instance $I \in P$, let $F(I)$ denote the set of feasible solutions.

For solution $s \in F(I)$, let $w(s)$ denote its value (objective function value).

Goal: Find $s \in F(I)$ such that $w(s)$ is minimal if Π is a minimization problem (and maximal if Π is a maximization problem).

3.1 Basics

An approximation algorithm A for Π is an algorithm that, given an $I \in P$, outputs an $A(I) = s \in F(I)$ and has a running time that is polynomial in the encoding length of I.

Algorithm A achieves an approximation ratio of c if

$w(A(I)) \leq c \cdot OPT(I)$ (Π is a minimization problem)

$w(A(I)) \geq c \cdot OPT(I)$ (Π is a maximization problem)

for all $I \in P$. Here OPT(I) denotes the value of an optimal solution.

Sometimes an additive constant of b is allowed in the above inequalities. This constant b must be independent of the input. In this case c is referred to as an asymptotic approximation ratio.

3.1 Basics

Problem Max Cut: Undirected graph G=(V,E), where V is the set of vertices and E is the set of edges. Find a partition (S, V\S) of V such that the number of edges between S and V\S is maximal.

S is called a cut. Edges between S and V\S are called cut edges.

Symmetric difference: $S \Delta \{v\}$

$$S \Delta \{v\} = \begin{cases} S \cup \{v\} & \text{if } v \notin S \\ S \setminus \{v\} & \text{if } v \in S \end{cases}$$

Algorithm Local Improvement (LI):

S:=∅;

while $\exists\, v \in V$ such that $w(S \Delta \{v\}) > w(S)$ **do** $S := S \Delta \{v\}$ **endwhile;**

output S;

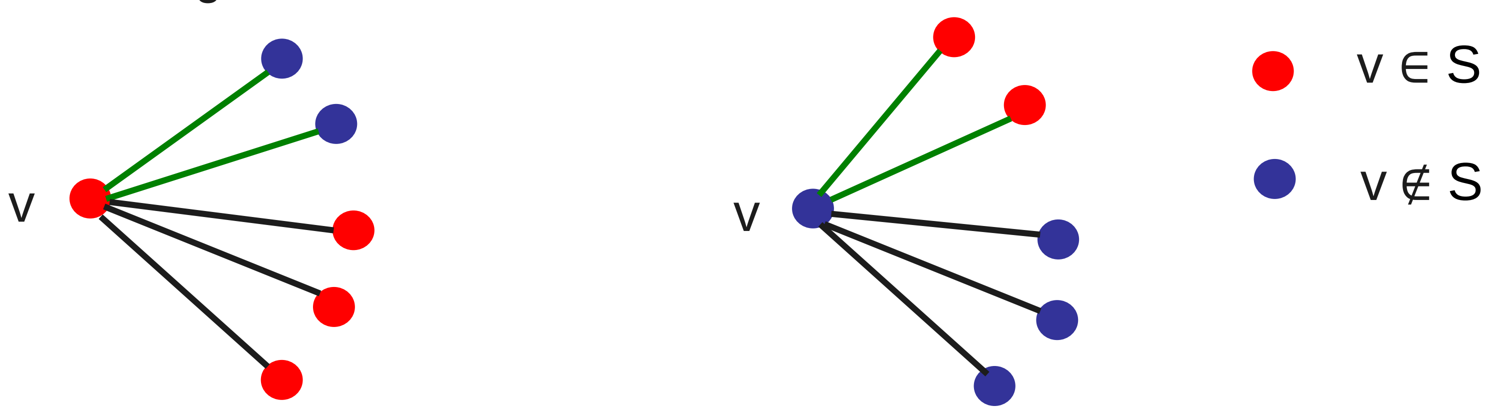
Theorem: LI achieves an approximation ratio of 1/2.

3.1 Basics

Theorem: LI achieves an approximation ratio of 1/2.

Proof: The while-loop of LI is executed at most |E| times because in each iteration the number of cut edges increases by at least 1. In every iteration the value of each of the |V| symmetric differences can be computed in O(|E|) time. Hence LI runs in polynomial time.

When LI terminates, for every v ∈ V, the number of adjacent cut edges is at least as large as the number of adjacent non-cut edges: If there were a vertex v not satisfying this property (see figure below), then S Δ {v} would be a cut with more cut edges.



3.1 Basics

By considering all $v \in V$, we obtain

$$\sum_{v \in V} (\#\text{cut edges adjacent to } v) \geq \sum_{v \in V} (\#\text{non}-\text{cut edges adjacent to } v)\ .$$

In the above inequality the left-hand side expression is twice the number of cut edges, i.e. $2|LI(G)|$. The right-hand side expression is twice the number of non-cut edges, i.e. $2(|E| - |LI(G)|)$.

We conclude

$$2|LI(G)| \geq 2(|E| - |LI(G)|) \text{ and } |LI(G)| \geq ½\cdot|E| \geq ½\cdot OPT(G).$$

3.2 Traveling Salesman Problem

Traveling Salesman Problem (TSP): Weighted graph $G=(V,E)$ with $V=\{v_1,\ldots,v_n\}$ and a function $w: E \rightarrow \mathbb{R}_0^+$ that assigns a length/weight to each edge. Find a tour that visits each vertex exactly once and has minimum length.

Formally, a tour is a Hamiltonian cycle.

A tour can be encoded as a permutation π on $\{1,\ldots, n\}$ having the property that

$\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E$ and $\{v_{\pi(n)}, v_{\pi(1)}\} \in E$.

Length/weight: $\sum_{i=1}^{n-1} w(\{v_{\pi(i)}, v_{\pi(i+1)}\}) + w(\{v_{\pi(n)}, v_{\pi(1)}\})$

Euclidean Traveling Salesman Problem (ETSP): n cities $s_1,\ldots, s_n$ in $\mathbb{R}^2$.

dist(s_i,s_j) = Euclidean distance between s_i and s_j. Find a tour that visits each city exactly once and has minimum length.

Will design algorithms with approximation ratios of 2 and 1.5.

TSP and ETSP are NP-hard

3.2 Traveling Salesman Problem

Minimum spanning tree: Weighted graph G=(V,E) with $w: E \to \mathbb{R}$. A minimum spanning tree T is a tree such that each $v \in V$ is a vertex of T and $\sum_{e \in T} w(e)$ is minimum.

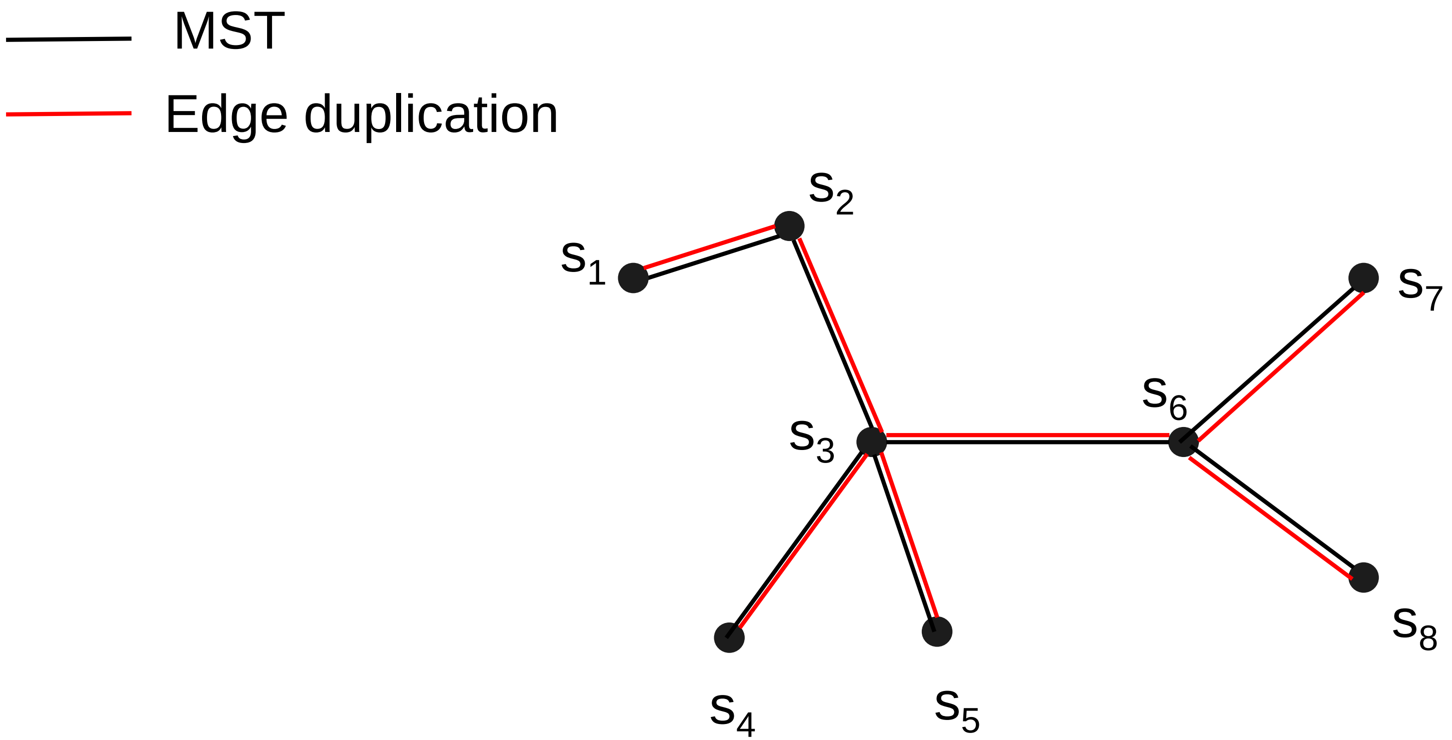
The following algorithm works with a multigraph, i.e. several copies of an edge may be contained in E.

Algorithms MST:

1. Compute a minimum spanning tree T for G=(V,E) with $V=\{s_1,\ldots,s_n\}$ and $w(s_i,s_j)$= Euclidian distance between s_i and s_j.
2. Construct graph H in which all edges of T are duplicated.
3. Compute an Eulerian cycle C in H (each edge is traversed exactly once).
4. Determine the order $s_{\pi(1)}, \ldots, s_{\pi(n)}$ of the first occurrences of $s_1,\ldots,s_n$ in C and output this sequence $s_{\pi(1)}, \ldots, s_{\pi(n)}$.

Theorem: Algorithm MST achieves an approximation ratio of 2.

3.2 Example MST algorithm



Possible tour starting at s_4: $s_4\ s_3\ s_5\ s_6\ s_8\ s_7\ s_2\ s_1$

3.2 Traveling Salesman Problem

Theorem: Algorithm MST achieves an approximation ratio of 2.

Proof: Let C_{OPT} be a tour of minimum length/weight OPT(I). Let T be a minimum spanning tree of weight w(T).

There holds $w(T) \leq OPT(I)$ because the removal of one edge from C_{OPT} yields a spanning tree for G.

The cycle C, computed in Step 3 of the algorithm, has a length of at most $2 \cdot OPT(I)$ because graph H is obtained from T by edge duplication.

The tour, derived in Step 4, is obtained from C by shortcuts that satisfy the triangle inequality.

The purpose of the edge duplication is to ensure that each vertex has even degree.

Proposition: In any tree T the number of vertices having odd degree is even.

Minimum perfect matching: Weighted graph G=(V,E) with w: E $\rightarrow \mathbb{R}_0^+$. A perfect matching is a subset F ⊆ E such that each vertex v ∈ V is incident to exactly one edge of F. Precondition: |V| is even. A perfect matching of minimum total weight is called a minimum perfect matching. There exist polynomial time algorithms for computing it.

3.2 Traveling Salesman Problem

Proposition: In any tree T the number of vertices having odd degree is even.

Proof: The total degree $D = \sum_{v \in T} \deg(v)$ = 2 #edges of T is an even number.

We split the sum along vertices with even and odd degree.

$$D = \sum_{\substack{v \in T \\ \text{with even degree}}} \deg(v) + \sum_{\substack{v \in T \\ \text{with odd degree}}} \deg(v)$$

Since the first sum gives an even value, so does the second sum. Therefore, in the second sum, the summation is over an even number of vertices.

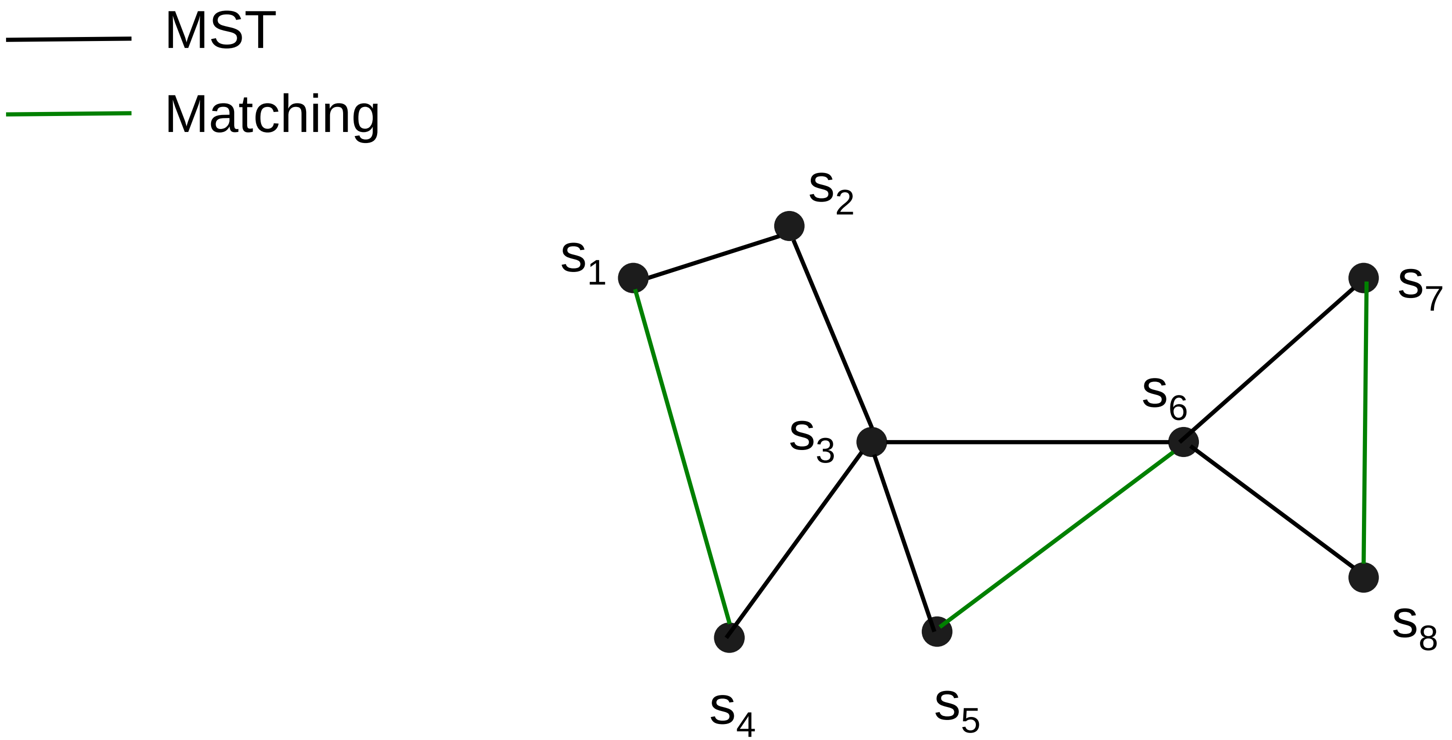
Algorithm Christiofides:

1. Compute a minimum spanning tree T for $s_1,\ldots,s_n$.
2. In T determine the set V' of vertices having odd degree and compute a minimum perfect matching F for V'.
3. Add F to T and compute an Eulerian cycle C.
4. Determine the order $s_{\pi(1)}, \ldots, s_{\pi(n)}$ of the first occurrences of $s_1,\ldots,s_n$ on C and output this sequence $s_{\pi(1)},\ldots, s_{\pi(n)}$.

Theorem: Algorithm Christofides achieves an approximation factor of 1.5

Theorem: The approximation ratio of the Christofides algorithm is not smaller than 1.5.

3.2 Example algorithm Christofides



Possible tour starting at s_4: $s_4\ s_3\ s_6\ s_8\ s_7\ s_5\ s_2\ s_1$

3.2 Traveling Salesman Problem

Theorem: Algorithm Christofides achieves an approximation factor of 1.5

Proof: As in the analysis of the MST algorithm there holds w(T) ≤ OPT(I).

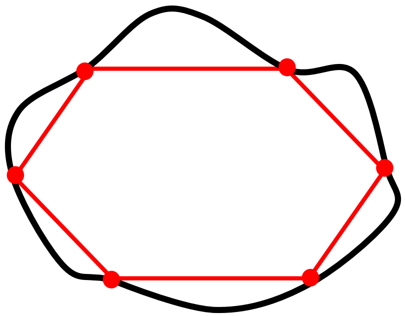
V' = set of vertices having odd degree in T

C_{OPT} = TSP tour of minimum length/weight for V

C'_{OPT} = TSP tour of minimum length/weight for V'

We first argue that $w(C'_{OPT}) \leq w(C_{OPT})$. To this end consider the vertices of V' on the tour C_{OPT}. Connect them by taking shortcuts, satisfying the triangle inequality, along C_{OPT} (cf. figure below). For the resulting cycle C' visiting V', there holds $w(C'_{OPT}) \leq w(C') \leq w(C_{OPT})$.

● vertices of V'



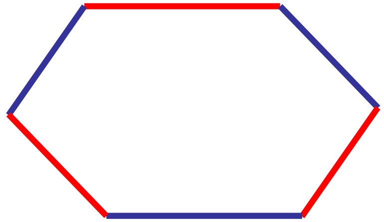
3.2 Traveling Salesman Problem

Cycle C'_{OPT} has an even number of vertices/edges and hence can be decomposed into two perfect matchings (cf. figure below). At least one of these matchings has a weight of at most $w(C'_{OPT})/2 \leq w(C_{OPT})/2$ = OPT(I)/2.

It follows that the cycle C computed in Step 3 of Christofides' algorithm has a weight of at most 3/2·OPT(I).

Finally, in Step 4, shortcuts are taken that satisfy again the triangle inequality.

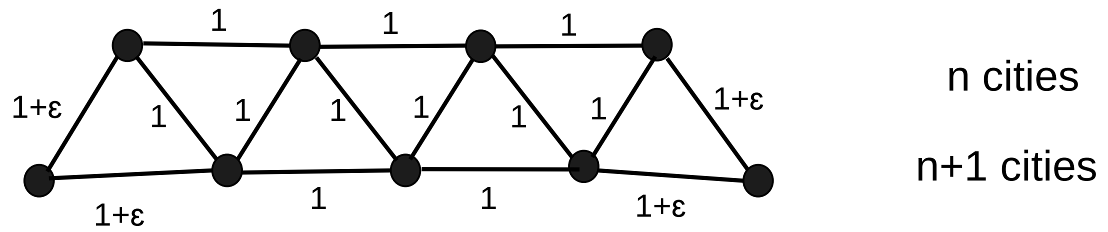
C'_{OPT}



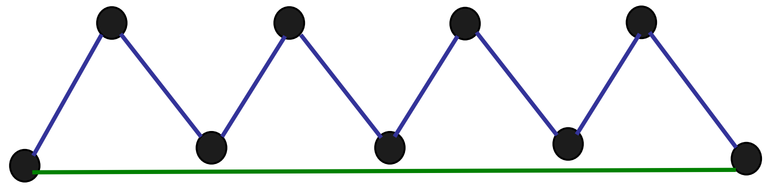
Theorem: The approximation ratio of the Christofides algorithm is not smaller than 1.5.

Proof: Consider the following problem instance with 2n+1 cities.

Length OPT: 2n+1+4ε



Christofides: Blue edges represent MST; the green edge is the added matching.



Total length ≥ 2n+n = 3n

3.2 Traveling Salesman Problem

Problem Hamiltonian Cycle (HC): $G=(V,E)$ unweighted graph. Does G have a Hamiltonian cycle, i.e. a cycle that visits each vertex exactly once?

NP-complete

Theorem: Let $c>1$. If $P \neq NP$, then general TSP does not have an approximation algorithm that achieves a performance factor of c.

In above theorem $c = c(n)$ can be an arbitrary function computable in polynomial time.

3.2 Traveling Salesman Problem

Theorem: Let $c>1$. If $P \neq NP$, then general TSP does not have an approximation algorithm that achieves a performance factor of c.

Proof: Suppose that there exists an approximation algorithm A for TSP that achieves an approximation factor of c. We show that in this case HC can be solved in polynomial time, i.e. P=NP.

Algorithm for HC: Let $G=(V,E)$ be the input for HC.

1. Construct a weighted graph $G'=(V',E')$ with $V'=V$, $E' = V \times V$ and

$$w(\{i,j\}) = \begin{cases} 1 & \{i,j\} \in E \\ c|V| & \{i,j\} \notin E \end{cases}$$

2. Apply algorithm A to G' and w' to obtain a TSP tour C_A.

3. **if** $w(C_A) \leq c|V|$ **then** output "G has a Hamiltonian cycle"

   else output "G does not have a Hamiltonian cycle"

3.2 Traveling Salesman Problem

We argue that the algorithm's output is correct.

If G has a Hamiltonian cycle, then G' has a TSP tour of length |V|. Since A is a c-approximation algorithm, it finds a tour C_A with $w(C_A) \leq c|V|$.

If G does not have a Hamiltonian cycle, then every TSP tour in G' must contain at least one edge of weight c|V|. Hence $w(C_A) > c|V|$.

3.3 Job scheduling

Makespan minimization: Schedule n jobs with processing times $p_1,\ldots, p_n$ to m identical parallel machines so as to minimize the makespan, i.e. the completion time of the last job that finishes in the schedule.

Algorithm Sorted List Scheduling (SLS):

1. Sort the n jobs in order of non-increasing processing times $p_1 \geq \ldots \geq p_n$.
2. Schedule the job sequence using List Scheduling (Greedy).

Theorem: SLS achieves an approximation factor of 4/3.

3.3 Job scheduling

Theorem: SLS achieves an approximation factor of 4/3.

Proof: It suffices to analyze job sequences $\sigma=J_1,\ldots,J_n$ with $p_1 \geq \ldots \geq p_n$ such that J_n finishes last in SLS's schedule and hence defines the makespan.

For, if J_l with $l<n$ finishes last, consider $\sigma'=J_1,\ldots,J_l$ and show $SLS(\sigma') \leq 4/3 \cdot OPT(\sigma')$. This implies $SLS(\sigma) = SLS(\sigma') \leq 4/3 \cdot OPT(\sigma') \leq 4/3 \cdot OPT(\sigma)$.

Let $OPT = OPT(\sigma)$

Case 1: $p_n \leq OPT/3$

Job J_n is assigned to a least loaded machine so that the idle time on any machine in SLS's final schedule is upper bounded by p_n. Therefore, $m \cdot SLS(\sigma) \leq \Sigma_{1\leq i\leq n}\, p_i + (m-1)p_n$, which implies

$$SLS(\sigma) \leq 1/m \cdot \Sigma_{1\leq i\leq n}\, p_i + (1-1/m)p_n \leq OPT + OPT/3 = 4/3 \cdot OPT.$$

3.3 Job scheduling

Case 2: p_n > OPT/3

All jobs $J_1,\ldots,J_n$ have a processing time greater than OPT/3.

Lemma: If p_n > OPT/3, then SLS constructs an optimal schedule.

Proof: By contradiction. Suppose that J_k is the first job in σ that SLS cannot assign to the current schedule so that a makespan of OPT is maintained.

Consider SLS's schedule immediately before the assignment. Each machine contains either one or two jobs.

Let $M_1,\ldots,M_i$ be the machines containing one job. Call these jobs large.

Let $M_{i+1},\ldots,M_m$ be the machines containing two jobs. Call these jobs small.

J_k is also called small.

3.3 Job scheduling

By assumption J_k cannot be placed on a least loaded machine so that a makespan of OPT is maintained. Hence J_k cannot be placed on a machine containing a large job so that a makespan of OPT is maintained. Observe that J_k is the smallest job in the job sequence considered so far.

We conclude that in an optimal schedule a large job cannot be combined with any other job.

Hence in an optimal schedule the i large jobs are located on i separate machines. The remaining 2(m-i)+1 small jobs must be executed on the other m-i machines. This implies that one machine contains three jobs, which is a contradiction to the fact that OPT is the optimum makespan.

3.3 Approximation schemes

An approximation scheme for an optimization problem is a set $\{A(\varepsilon) \mid \varepsilon > 0\}$ of approximation algorithms for the problem such that $A(\varepsilon)$ achieves an approximation factor of $1+\varepsilon$, in case of a minimization problem, and $1-\varepsilon$ in case of a maximization problem.

PTAS = Polynomial Time Approximation Scheme

3.3 PTAS for Knapsack

Problem Knapsack: n objects with weights $w_1,\ldots, w_n \in \mathbb{N}$ and values $v_1,\ldots, v_n \in \mathbb{N}$. Knapsack with weight bound b. Find a subset $I \subseteq \{1, \ldots, n\}$ with $\sum_{i\in I} w_i \leq b$ such that $\sum_{i\in I} v_i$ is maximal.

Problem is NP-hard.

For j=1,…,n and any non-negative integer i, let

$F_j(i)$ = minimum weight of a subset of $\{1,\ldots, j\}$ whose total value is at least i. If no such subset exists, set $F_j(i) := \propto$.

Observation: Let OPT be the value of an optimal solution.

Then $OPT = \max\{i \mid F_n(i) \leq b \}$

Lemma: a) $F_j(i) = 0$ for $i \leq 0$ and $j \in \{1,\ldots,n\}$

b) $F_0(0) = 0$ and $F_0(i) = \propto$ for $i > 0$

c) $F_j(i) = \min \{F_{j-1}(i), w_j + F_{j-1}(i-v_j) \}$ for $i,j > 0$

3.3 PTAS for Knapsack

Algorithm Exact Knapsack

$F_j(i)$ for j=0 and i ≤ 0 are known.

1. i:=0;
2. **repeat**
3. i:= i+1;
4. **for** j := 1 **to** n **do**
5. $F_j(i) = \min \{ F_{j-1}(i),\ w_j + F_{j-1}(i-v_j) \}$;
6. **endfor;**
7. **until** $F_n(i) > b$;
8. output i-1;

Theorem: Exact Knapsack has a running time of O(n OPT).

3.3 PTAS for Knapsack

Algorithm Scaled Knapsack(ε) $\varepsilon > 0$

1. $v_{max} := \max \{v_j \mid 1 \le j \le n \}$;
2. $k := \max \{1, \lfloor \varepsilon\, v_{max} / n \rfloor\}$
3. **for** j := 1 **to** n **do** $v_j(k) = \lfloor v_j / k \rfloor$ **endfor;**
4. Using algorithm Exact Knapsack, compute OPT(k) and S(k), i.e. the value and the subset of objects of an optimal solution for the Knapsack Problem with values $v_j(k)$ and unchanged weights w_j and b.
5. output $OPT^* = \sum_{j \in S(k)} v_j$.

Theorem: Scaled Knapsack(ε) achieves an approximation factor of 1- ε.

Theorem: Scaled Knapsack(ε) has a running time of $O(n^3/\varepsilon)$.

3.3 PTAS for Knapsack

Theorem: Scaled Knapsack(ε) achieves an approximation factor of 1- ε.

Proof: S = set of objects of an optimal solution

OPT = value of optimal solution

$$OPT^* = \sum_{j \in S(k)} v_j \geq k \sum_{j \in S(k)} \lfloor v_j / k \rfloor \geq k \sum_{j \in S} \lfloor v_j / k \rfloor$$

The last inequality holds because S(k) is the set of an optimal solution for the scaled values.

$$OPT^* \geq k \sum_{j \in S} \lfloor v_j / k \rfloor \geq k \sum_{j \in S} (v_j / k - 1) = \sum_{j \in S} v_j - k \,|\, S \,|$$
$$= OPT(1 - k \,|\, S \,|\, / \, OPT) \geq OPT(1 - kn / v_{\max})$$

The last inequality holds because |S| ≤ n and OPT ≥ v_{max} (each object alone can be packed feasibly into the knapsack).

3.3 PTAS for Knapsack

k=1: In this case no scaling has occurred and hence OPT* = OPT.

k>1: In this case $k \leq \varepsilon\, v_{max} / n$ and, equivalently, $kn/ v_{max} \leq \varepsilon$. It follows that $OPT^* \geq (1-\varepsilon)OPT$.

3.3 PTAS for Knapsack

Theorem: Scaled Knapsack(ε) has a running time of $O(n^3/\varepsilon)$.

Proof: Critical is the call to Exact Knapsack. All other instructions take $O(n)$ time.

The running time of Exact Knapsack is $O(n\ OPT^*) = O(n^2\ v_{max}/k)$ because up to n objects of value at most v_{max}/k can be packed into the knapsack.

k=1: In this case $\varepsilon\ v_{max}/n < 2$ and hence $v_{max}/k = v_{max} \leq 2n/\varepsilon$.

k>1: In this case $k = \lfloor \varepsilon\ v_{max} / n \rfloor$. This implies $k \geq \varepsilon\ v_{max}/n - 1$ and, equivalently,

$n(1+1/k)/\varepsilon \geq v_{max}/k$. We conclude $v_{max}/k \leq 2n/\varepsilon$.

3.3 PTAS for Makespan Minimization

m identical parallel machines, n jobs with processing times $p_1, \ldots, p_n$.

Algorithm SLS(k)

1. Sort $J_1, \ldots, J_n$ in order of non-increasing processing times such that $p_1 \geq \ldots \geq p_n$.
2. Compute an optimal schedule for the first k jobs.
3. Schedule the remaining jobs using List Scheduling (Greedy).

Theorem: For constant m and $k = \lceil (m-1)/\varepsilon \rceil$, algorithm SLS(k) is a PTAS.

3.3 PTAS for Makespan Minimization

Theorem: For constant m and $k = \lceil (m-1)/\varepsilon \rceil$, algorithm SLS(k) is a PTAS.

Proof: Let $\sigma = J_1,\dots,J_n$ be an arbitrary job sequence with $p_1 \geq \dots \geq p_n$. Let J_l be the job that finishes last in SLS(k)'s schedule and defines the makespan.

C = makespan SLS(k)   OPT = optimum makespan

Case 1: $l \leq k$

$C = OPT(J_1,\dots,J_k) \leq OPT(J_1,\dots,J_n) = OPT$ and thus $C = OPT$.

Case 2: $l > k$

Since J_l is placed on a least loaded machine, the idle time on any machine is upper bounded by p_l. Hence

$mC \leq \Sigma_{1 \leq i \leq n}\, p_i + (m-1)p_l$ which implies

$C \leq 1/m \cdot \Sigma_{1 \leq i \leq n}\, p_i + (m-1)/m \cdot p_l \leq OPT + (m-1)/m \cdot p_l$.

Moreover, $OPT \geq 1/m \cdot \Sigma_{1 \leq i \leq n}\, p_i \geq 1/m \cdot \Sigma_{1 \leq i \leq k}\, p_i \geq 1/m \cdot \Sigma_{1 \leq i \leq k}\, p_l = k/m \cdot p_l$ and thus

$p_l \leq m/k \cdot OPT$.

3.3 PTAS for Makespan Minimization

We conclude $C \leq OPT + (m\text{-}1)/k \cdot OPT$.

Setting $k = \lceil (m-1)/\varepsilon \rceil$, we obtain $C \leq (1+ \varepsilon)OPT$.

As for the running time, an optimal schedule for $J_1,\ldots,J_k$ can be computed by full enumeration, which takes $O(m^k) = O(m^{(m-1)/\varepsilon})$ time. The last expression is $m^{O(m/\varepsilon)}$.

3.3 PTAS for Makespan Minimization

Will construct a PTAS for an arbitrary/variable number of machines.

Problem Bin Packing: n elements $a_1, \dots, a_n \in [0,1]$. Bins of capacity 1. Pack the n elements into bins, without exceeding their capacity, so that the number of used bins is as small as possible.

Observation: There exists a schedule with makespan t if and only if $p_1, \dots, p_n$ can be packed into m bins of capacity t.

Notation: $I = \{p_1, \dots, p_n\}$

bins(I,t) = minimum number of bins of capacity t needed to pack I

OPT = min {t | bins(I,t) ≤ m}

LB ≤ OPT ≤ 2 LB

$\mathrm{LB} = \max \{\frac{1}{m}\sum_{i=1}^{n} \mathrm{p_i}, \max_{1\le i\le n} \mathrm{p_i}\}$

Execute binary search on [LB, 2LB] and solve a bin packing problem for each guess.

3.3 PTAS for Makespan Minimization

Bin packing for a constant number of element sizes.

k = number of element sizes

t = capacity of bins

Problem instance $(n_1, \ldots, n_k)$ with $\sum_{j=1}^{k} n_j = n$

Subproblem specified by $(i_1, \ldots, i_k)$ where i_j is the number of elements of element size j.

$\text{bins}(i_1, \ldots, i_k)$ = minimum number of bins to pack $(i_1, \ldots, i_k)$

3.3 PTAS for Makespan Minimization

Compute $Q = \{ (q_1,\ldots,q_k) \mid bins(q_1, \ldots, q_k) = 1, 0 \le q_i \le n_i \text{ for } i=1, \ldots, k\}$

Q contains $O(n^k)$ elements

Compute k-dimensional table with entries $bins(i_1, \ldots, i_k)$,

where $(i_1, \ldots, i_k) \in \{0,\ldots, n_1\} \times \ldots \times \{0,\ldots, n_k\}$

Initialize bins(q)=1 for all $q \in Q$ and

compute $bins(i_1, \ldots, i_k) = 1 + \min_{q \in Q} bins(i_1-q_1,\ldots, i_k-q_k)$

Takes $O(n^{2k})$ time.

Reduction from scheduling to bin packing: Two types of errors occur.

- Round the element sizes to a bounded number of sizes.

- Stop the binary search to ensure polynomial running time.

3.3 PTAS for Makespan Minimization

Basic algorithm: ε = error parameter $t \in [LB, 2LB]$

1. Ignore jobs of processing time smaller than εt.

2. Round down the remaining processing times.

   $p_i \in [t\varepsilon (1+\varepsilon)^i, t\varepsilon(1+\varepsilon)^{i+1})$ $i \geq 0$ is rounded to $t\varepsilon (1+\varepsilon)^i$

   $t\varepsilon(1+\varepsilon)^{i+1} < t$ implies $i+1 < \log_{1+\varepsilon} 1/\varepsilon$ and $k = \lceil \log_{1+\varepsilon} 1/\varepsilon \rceil$ job classes suffice

3. Compute optimal solution to this problem with bin capacity t.
   Makespan for original job sizes is at most $t(1+\varepsilon)$.

4. Remaining jobs ignored so far are first assigned to the available capacity in the open bins. Then new bins of capacity $t(1+\varepsilon)$ are used.

Let $\alpha(I,t,\varepsilon)$ denote the number of used bins.

3.3 PTAS for Makespan Minimization

Lemma: $\alpha(I,t,\varepsilon) \leq bins(I,t)$

Proof: Obvious if no new bins are opened to assign the small, initially ignored elements. Each time a new bin is opened, all the open ones are filled to an extent of at least t.

Corollary: $\min \{t \mid \alpha(I,t,\varepsilon) \leq m\} \leq OPT$.

Execute binary search on [LB,2LB] until the length of the search interval is at most εLB.

$(1/2)^i \, LB \leq \varepsilon LB$ implies $i = \lceil \log_2 1/\varepsilon \rceil$

Let T be the right interval boundary when the search terminates.

3.3 PTAS for Makespan Minimization

Lemma: $T \leq (1+\varepsilon)$ OPT

Proof: min $\{t \mid \alpha(I,t,\varepsilon) \leq m\}$ in the interval $[T-\varepsilon LB, T]$.

Hence $T \leq \min\{t \mid \alpha(I,t,\varepsilon) \leq m\} + \varepsilon LB \leq (1+\varepsilon)$ OPT.

Basic algorithm with $t = T$ produces a makespan of at most $(1+\varepsilon)T$

Theorem: The entire algorithm produces a solution with a makespan of at most $(1+\varepsilon)^2 T \leq (1+3\varepsilon)$ OPT.

The running time is $O(n^{2k} \lceil \log_2 1/\varepsilon \rceil)$ where $k = \lceil \log_{1+\varepsilon} 1/\varepsilon \rceil$.

3.4 Max-SAT and randomization

Problem Max-≥kSAT: Clauses $C_1,\dots,C_m$ over Boolean variables $x_1,\dots,x_n$.

$C_i = l_{i,1} \vee \dots \vee l_{i,k(i)}$ where $k(i) \geq k$ and

literals $l_{i,j} \in \{ x_1, \bar{x}_1, \dots, x_n, \bar{x}_n \}$ for $j=1,\dots,k(i)$

Find an assignment to the variables that maximizes the number of satisfied clauses.

Example: $C_1 = x_1 \vee \bar{x}_2 \vee x_3$ $\quad C_2 = x_1 \vee \bar{x}_3$ $\quad C_3 = x_2 \vee \bar{x}_3$

Max-≥kSAT is NP-hard

3.4 Max-SAT and randomization

Definition: A randomized approximation algorithm is an approximation algorithm that is allowed to make random choices. In polynomial time a random number in the range $\{1,\dots,n\}$, $n \in \mathbb{N}$, is chosen, where the coding length of n is polynomial in the coding length of the input. The bits of this number serve as a random source

Algorithm A achieves an approximation factor of c if

$E[w(A(I))] \leq c \cdot OPT(I)$ (in case of a minimization problem)

$E[w(A(I))] \geq c \cdot OPT(I)$ (in case of a maximization problem)

for all $I \in P$.

3.4 Max-SAT and randomization

Algorithm RandomSAT:

for i:=1 to n **do**

Choose a bit $b \in \{0,1\}$ uniformly at random;

if b=0 **then** $x_i := 0$ **else** $x_i := 1$ **endif;**

endfor;

Output the assignment of the variables $x_1,\ldots,x_n$;

Theorem: The expected number of satisfied clauses achieved by RandomSAT is at least $(1-1/2^k)m$.

3.4 Max-SAT and randomization

Theorem: The expected number of satisfied clauses achieved by RandomSAT is at least $(1-1/2^k)m$.

Proof: For j=1,…,m, define

$$X_j = \begin{cases} 1 & C_j \text{ satisfied} \\ 0 & \text{otherwise} \end{cases}.$$

Then $X = \Sigma_{1\le j\le m} X_j$ is the number of satisfied clauses.
A clause C_j is not satisfied if each of its k(j) literals gives 0. This happens with probability $1/2^{k(j)}$ because each Boolean variable is equal to 0 (or 1) with probability ½. Thus $\text{Prob}[X_j=0] = 1/2^{k(j)}$ and

$$E[X] = \Sigma_{1\le j\le m} E[X_j] = \Sigma_{1\le j\le m} \text{Prob}[X_j=1] = \Sigma_{1\le j\le m} (1-\text{Prob}[X_j=0])$$

$$= \Sigma_{1\le j\le m} (1- 1/2^{k(j)}) \ge \Sigma_{1\le j\le m} (1- 1/2^k) = (1- 1/2^k)\, m.$$

3.4 Max-SAT and randomization

Derandomization

$E[X|B]$ = expected value of X if event B holds

Algorithm DetSAT:

for i:=1 to n do

Compute $E_0 = E[\, X \mid x_j = b_j$ for $j=1,\dots,i-1$ and x_i = false];

Compute $E_1 = E[\, X \mid x_j = b_j$ for $j=1,\dots,i-1$ and x_i = true];

if $E_0 \geq E_1$ then $b_i := 0$ else $b_i := 1$; endif;

endfor;

Output $b_1,\dots,b_n$;

Theorem: DetSAT satisfies at least $E[X] = (1-1/2^k)m$ clauses.

Algorithm achieves the best possible performance. If P ≠ NP, no approximation factor greater than $1-1/2^k + \varepsilon$, for $\varepsilon > 0$, can be achieved.

3.4 Max-SAT and randomization

Theorem: DetSAT satisfies at least $E[X] = (1-1/2^k)m$ clauses.

Proof: For i=0,…,n, let $E^i = E[\, X \mid x_j = b_j \text{ for } j=1,\dots,i]$.

$E^0 = E[X]$ = expected number of satisfied clauses in RandomSAT

E^n = number of satisfied clauses in DetSAT

We will show $E^i \geq E^{i-1}$, for i=1,…,n. This implies $E^n \geq E^0 = E[X]$.

$E^{i-1} = E[\, X \mid x_j = b_j \text{ for } j=1,\dots,i-1]$

$= ½\cdot E[\, X \mid x_j = b_j \text{ for } j=1,\dots,i-1 \text{ and } x_i = \text{false}]$

$+ ½\cdot E[\, X \mid x_j = b_j \text{ for } j=1,\dots,i-1 \text{ and } x_i = \text{true}]$

$= ½\cdot(E_0 + E_1)$ (E_0, E_1 are the expected values defined in the i-th iteration)

$\leq \max\{E_0,E_1\}$

$= E[\, X \mid x_j = b_j \text{ for } j=1,\dots,i] = E^i$.

3.4 Max-SAT and randomization

LP relaxations

Example: max x+y

s.t. $x + 2y \leq 10$

$3x - y \leq 9$

$x,y \geq 0$

Consider Max-SAT, which corresponds to Max-≥1SAT

Formula φ with clauses $C_1,\ldots,C_m$ over Boolean variables $x_1,\ldots,x_n$.

For each clause C_j define

$V_{j,+}$ = set of unnegated variables in C_j

$V_{j,-}$ = set of negated variables in C_j

Formulation as integer linear program

For each x_i introduce variable y_i. For each clause C_j introduce variable z_j.

$$y_i = \begin{cases} 1 & x_i = \text{true} \\ 0 & x_i = \text{false} \end{cases} \qquad z_j = \begin{cases} 1 & C_j \text{ satisfied} \\ 0 & C_j \text{ not satisfied} \end{cases}$$

$$\begin{aligned} \max\ & \textstyle\sum_{j=1}^{m} z_j \\ \text{s.t.}\ & \sum_{i:x_i \in V_{j,+}} y_i + \sum_{i:x_i \in V_{j,-}} (1 - y_i) \geq z_j \quad j=1,\ldots,m \\ & y_i, z_j \in \{0,1\} \qquad i=1,\ldots,n \quad j=1,\ldots,m \end{aligned}$$

Integer linear programming (ILP) is NP-hard.

Theorem: (Khachyian 1980) LP is in P.

3.4 Max-SAT and randomization

Relaxed linear program for MaxSAT

$$\max \; \sum_{j=1}^{m} z_j$$

$$\text{s.t.} \quad \sum_{i:x_i \in V_{j,+}} y_i + \sum_{i:x_i \in V_{j,-}} (1 - y_i) \geq z_j \quad j=1,\ldots,m$$

$$y_i, z_j \in [0,1] \quad i=1,\ldots,n \quad j=1,\ldots,m$$

Algorithm RRMaxSAT (RandomizedRounding MaxSAT)

Find optimal solution $(\hat{y}_1, \ldots, \hat{y}_n)$ $(\hat{z}_1, \ldots, \hat{z}_m)$ to the relaxed LP for MaxSAT;
for i:=1 to n **do**

Choose a bit b ∈ {0,1} such that $b = \begin{cases} 1 & \text{with probability } \hat{y}_i \\ 0 & \text{with probability } 1 - \hat{y}_i \end{cases}$

if b=1 **then** x_i := 1 **else** x_i := 0 **endif;**

endfor;

Output the assignment of the variables $x_1,\ldots,x_n$;

3.4 Max-SAT and randomization

Theorem: RRMaxSAT achieves an approximation factor of $1-1/e \approx 0.632$.

Theorem: Given a formular φ, apply both RandomSAT and RRMaxSAT and select the better of the two solutions. Then the resulting algorithm achieves an approximation factor of ¾.

3.4 Max-SAT and randomization

Theorem: RRMaxSAT achieves an approximation factor of 1-1/e ≈ 0.632.

Proof: Let OPT(φ) be the maximum number of clauses that can be satisfied in φ. There holds OPT(φ) ≤ $\Sigma_{1\le j\le m}\, \hat{z}_j$.

For j=1,…,m, let

$$X_j = \begin{cases} 1 & C_j \text{ satisfied in solution of RRMaxSAT} \\ 0 & \text{otherwise} \end{cases}$$

and let X = $\Sigma_{1\le j\le m}\, X_j$ be the total number of satisfied clauses.

Obviously, E[X] = $\Sigma_{1\le j\le m}$ E[X_j] = $\Sigma_{1\le j\le m}$ Prob[X_j=1]. There holds

$$\text{Prob}[X_j = 0] = \prod_{i:x_i \in V_{j,+}} (1-\hat{y}_i) \cdot \prod_{i:x_i \in V_{j,-}} (1-(1-\hat{y}_i)),$$

where

$$\sum_{i:x_i \in V_{j,+}} \hat{y}_i + \sum_{i:x_i \in V_{j,-}} (1-\hat{y}_i) \ge \hat{z}_j.$$

3.4 Max-SAT and randomization

Lemma 1: Let $y_1,\dots,y_k$ be real numbers with $0 \le y_i \le 1$, for $i=1,\dots,k$, and $\Sigma_{1\le i\le k}\, y_i \ge y$. Then $\prod_{i=1}^{k}(1-y_i) \le (1-y/k)^k$.

Lemma 2: For all $k \in \mathbb{N}$ and $x \in [0,1]$, there holds $1-(1-x/k)^k \ge (1-(1-1/k)^k)x$.

By Lemma 1, $\text{Prob}[X_j = 0] \le (1-\hat{z}_j / k(j))^{k(j)}$.

Hence, using Lemma 2,

$$\begin{aligned}\text{Prob}[X_j = 1] &\ge 1-(1-\hat{z}_j / k(j))^{k(j)} \\ &\ge (1-(1-1/k(j))^{k(j)})\hat{z}_j \\ &\ge (1-1/e)\hat{z}_j.\end{aligned}$$

We conclude $E[X] \ge (1-1/e)\sum_{1\le j\le m} \hat{z}_j \ge (1-1/e)\text{OPT}(\varphi)$.

3.4 Max-SAT and randomization

Proof of Lemma 1: The inequality of arithmetic and geometric means states that, for non-negative real numbers $a_1,\ldots,a_k$, there holds

$$1/k \cdot \sum_{i=1}^{k} a_i \geq \left(\prod_{i=1}^{k} a_i\right)^{1/k}.$$

Set $a_i = 1-y_i$. We obtain

$$\prod_{i=1}^{k}(1-y_i) \leq \left(1-1/k \cdot \sum_{i=1}^{k} y_i\right)^k \leq (1-y/k)^k.$$

Proof of Lemma 2: For any $x \in [0,1]$, let $f(x) = 1-(1-x/k)^k$ and $g(x) = (1-(1-1/k)^k)x$.

Function f is concave in (0,1) since the second derivative is equal to $-(k-1)/k \cdot (1-x/k)^{k-2}$ and hence non-positive. Function g is linear.

Observe that $f(0)=g(0)$ and $f(1)=g(1)$. It follows that $f(x) \geq g(x)$, for all $x \in [0,1]$.

3.4 Max-SAT and randomization

Theorem: Given a formular φ, apply both RandomSAT and RRMaxSAT and select the better of the two solutions. Then the resulting algorithm achieves an approximation factor of ¾.

Proof: Let

m_1= expected number of satisfied clauses of RandomSAT

m_2= expected number of satisfied clauses of RRMaxSAT

We will show that $\max\{m_1,m_2\} \geq ¾\cdot OPT(\varphi)$.

Let k(j) = # literals in C_j.

$$m_1 \geq \sum_{j=1}^{m} (1-1/2^{k(j)}) \geq \sum_{j=1}^{m} (1-1/2^{k(j)})\hat{z}_j$$

The last inequality holds because $0 \leq \hat{z}_j \leq 1$.

$$m_2 \geq \sum_{j=1}^{m} (1-(1-1/k(j))^{k(j)})\hat{z}_j$$

3.4 Max-SAT and randomization

$$\max\{m_1 + m_2\} \geq (m_1 + m_2)/2$$
$$\geq 1/2 \cdot \sum_{j=1}^{m} ((1 - 1/2^{k(j)}) + (1 - (1 - 1/k(j))^{k(j)})\hat{z}_j$$
$$\geq 3/4 \cdot \mathrm{OPT}(\varphi).$$

The last inequality follows from the next lemma.

Lemma: For all $k \in \mathbb{N}$, there holds $1-1/2^k + 1-(1-1/k)^k \geq 3/2$.

Proof: The statement of the lemma is equivalent to $(1-1/k)^k \leq 1/2 - 1/2^k$.

We have $(1-1/k)^k = \sum_{i=0}^{k} \binom{k}{i} (-1/k)^i \leq 1 - \binom{k}{1}(1/k) + \binom{k}{2}(1/k)^2$.

The inequality holds because, for any i ≥ 0, $-\binom{k}{i}(1/k)^i + \binom{k}{i+1}(1/k)^{i+1} \leq 0$.

We conclude $(1-1/k)^k \leq 1 - 1 + k(k-1)/(2k^2) = 1/2 - 1/(2k) \leq 1/2 - 1/2^k$.

3.5 Probabilistic approximation algorithms

Definition: A probabilistic approximation algorithm for an optimization problem is an approximation algorithm that outputs a feasible solution with probability at least ½.

Problem Hitting Set: Ground set $V = \{v_1,\ldots,v_n\}$ and subsets $S_1,\ldots, S_m \subseteq V$. Find the smallest set $H \subseteq V$ with $H \cap S_l \neq \emptyset$ for $l=1,\ldots,m$.

H is called a hitting set.

Formulation as ILP: Variables $x_1, \ldots, x_n$

$$x_j = \begin{cases} 1 & \text{if } v_j \in H_{OPT} \\ 0 & \text{if } v_j \notin H_{OPT} \end{cases}$$

$$\min \quad \sum_{j=1}^{n} x_j$$

$$\text{s.t.} \quad \sum_{j: v_j \in S_l} x_j \geq 1 \qquad l = 1, \ldots, m$$

$$x_j \in \{0,1\} \qquad j = 1, \ldots, n \qquad \text{relaxed to} \quad x_j \in [0,1]$$

3.5 Probabilistic approximation algorithms

Algorithm RRHS (RandomizedRounding HittingSet)

Find optimal solution $(\hat{x}_1, \ldots, \hat{x}_n)$ to the relaxed LP for HittingSet;

H := Ø;
for i:=1 to $\lceil \ln(2\mathrm{m}) \rceil$ **do**
  for j:=1 to n **do**
    Choose a bit b ∈ {0,1} such that $\mathrm{b} = \begin{cases} 1 & \text{with probability } \hat{x}_\mathrm{j} \\ 0 & \text{with probability } 1 - \hat{x}_j \end{cases}$
    if b=1 **then** H := H ∪ $\{v_j\}$ **endif;**
  endfor;
endfor;
Output H;

Theorem: For each instance of HittingSet there holds:

(1) RRHS finds a feasible solution with probability at least ½.

(2) E[|RRHS(I)|] ≤ $\lceil \ln(2\mathrm{m}) \rceil$ OPT(I).

3.5 Probabilistic approximation algorithms

Theorem: For each instance of HittingSet there holds:
(1) RRHS finds a feasible solution with probability at least ½.
(2) E[|RRHS(I)|] ≤ $\lceil \ln(2m) \rceil$ OPT(I).

Proof: There holds OPT(I) ≥ $\sum_{j=1}^{n} \hat{x}_j$.
Let H_i be the set of elements added to H in the i-th iteration of the outer for-loop, counting elements already contained in H.

We first prove part (2). There holds E[|H_i|] = $\sum_{j=1}^{n} \hat{x}_j$ ≤ OPT(I) and

$$E[|H|] \leq \sum_{i=1}^{\lceil \ln(2m) \rceil} E[|H_i|] \leq \lceil \ln(2m) \rceil OPT(I).$$

3.5 Probabilistic approximation algorithms

For the proof of part (1) we first focus on any set $S_l, 1 \leq l \leq m$, and evaluate Prob[$H \cap S_l = \emptyset$]. Consider any H_i. There holds

$$\mathrm{Prob}[\mathrm{H_i} \cap \mathrm{S_l} = \emptyset\,] = \prod_{j: v_j \in S_l} (1 - \hat{x}_j) \leq (1 - 1/k(l))^{k(l)} \leq 1/e,$$

where k(l) is the number of elements in S_l. The first inequality follows from Lemma 1, used in the analysis of RRMaxSAT. Here we take into account that $\sum_{j: v_j \in S_l} \hat{x}_j \geq 1$. The second inequality holds because $1-x \leq e^{-x}$, for any $x \in [0,1]$.

It follows that

$$\mathrm{Prob}[\mathrm{H} \cap \mathrm{S_l} = \emptyset\,] \leq (1/e)^{\lceil \ln(2m) \rceil} \leq 1/(2m)$$

because $H \cap S_l = \emptyset$ if and only if $H_i \cap S_l = \emptyset$, for $i = 1, \ldots, \lceil \ln(2m) \rceil$. By the Union Bound (Boole's inequality) we conclude

$$\mathrm{Prob}[\exists S_l \text{ such that } \mathrm{H} \cap \mathrm{S_l} = \emptyset\,] \leq m/(2m) = 1/2.$$

3.5 Probabilistic approximation algorithms

Theorem: Let p be a fixed polynomial and A be a polynomial time algorithm that, for each instance I of an optimization problem, computes a feasible solution with probability at least $1/p(|I|)$. Then, for each $\varepsilon>0$, there exists a polynomial time algorithm A_ε, that outputs a feasible solution with probability $1-\varepsilon$.

Proof: Algorithm $A_\varepsilon(I)$

for i:= 1 **to** $\lceil p(|I|) \ln(1/\epsilon) \rceil$ **do**

Compute solution S using A;

if S is feasible **then** output S and break **endif;**

endfor;

Set k := $\lceil p(|I|) \ln(1/\epsilon) \rceil$. Then

$$\text{Prob}[A_\varepsilon(I) \text{ does not produce feasible output}]$$
$$\leq (1 - 1/p(|I|))^k \leq (e^{-1/p(|I|)})^k = e^{-\ln(1/\varepsilon)} = \varepsilon,$$

where the second inequality uses the fact that $1-x \leq e^{-x}$, for $x \in [0,1]$.

3.5 Probabilistic approximation algorithms

Theorem: Let A be a randomized approximation algorithm with approximation factor c for a minimization problems. Then, for any $\varepsilon>0$ and $p<1$ there exists an approximation algorithm $A_{\varepsilon,p}$ that, for each input instance I and probability at least p, computes a solution of value at most $(1+\varepsilon)\cdot c\cdot OPT(I)$.

Proof: Assume w.l.o.g. that A always computes a feasible solution.

X : random variable for w(A(I))

By Markov's inequality, $Prob[\, X \geq (1+\varepsilon)E[X]\,] \leq 1/(1+\varepsilon)$.

Choose $k:=k(p,\varepsilon)$ such that $\left(\frac{1}{1+\varepsilon}\right)^k \leq 1-p$.

Algorithm $A_{\varepsilon,p}$(I)

for i:= 1 **to** k **do**

$w_i := w(A(I))$;

endfor;

Output $\min_{1\leq i\leq k} w_i$;

3.5 Probabilistic approximation algorithms

For any i, $1 \le i \le k$, there holds Prob[$w_i \ge (1+\varepsilon)E[X]$] $\le 1/(1+\varepsilon)$ and thus Prob[$w \ge (1+\varepsilon)E[X]$] $\le 1/(1+\varepsilon)^k \le 1-p$.

Since A is a c-approximation algorithm we conclude that, with probability at least p,

$$w = A_{\varepsilon,p}(I) \le (1+\varepsilon)\cdot E[X] \le (1+\varepsilon)\cdot c\cdot OPT(I).$$

3.6 Set Cover

Problem: Universe $U = \{u_1,\ldots,u_n\}$. Sets $S_1,\ldots,S_m \subseteq U$ with associated non-negative costs $c(S_1),\ldots,c(S_m)$. Find $J \subseteq \{1,\ldots,m\}$ such that $\bigcup_{j\in J} S_j = U$ and $\sum_{j\in J} c(S_j)$ minimal.

Greedy approach: Repeatedly choose the most cost-effective set. At any time let C be the set of covered elements. Cost-effectiveness of S is $c(S) / |S-C|$.

Algorithm Greedy:

1. $C := \emptyset$;
2. **while** $C \neq U$ **do**
3. Determine the set S having the smallest ratio $\alpha = c(S) / |S-C|$;
4. Choose S and set $\text{price}(e) := \alpha$, for all $e \in S-C$;
5. $C := C \cup S$;
6. **endwhile;**
7. Output the selected sets;

3.6 Set Cover

Theorem: Greedy achieves an approximation factor of $H_n = \sum_{k=1}^{n} 1/k$.

Theorem: The approximation factor of Greedy is not smaller than H_n.

3.6 Set Cover

Theorem: Greedy achieves an approximation factor of $H_n = \sum_{k=1}^{n} 1/k$.

Proof: Number the elements $e_1,\ldots,e_n$ in the order in which they are covered by Greedy. The cost of the selected sets is charged to the newly covered elements, according to price(e); cf. line 4 of the algorithm.

Let OPT be the cost of an optimal solution.

Lemma: For k=1,…,n, there holds price(e_k) ≤ OPT/(n-k+1).
Proof: Consider the iteration in which e_k gets covered by Greedy. Assume that Greedy has already selected sets $S_1^G, \ldots, S_t^G$ and will choose S_{t+1}^G in the current iteration covering e_k. Let $C_t = S_1^G \cup \ldots \cup S_t^G$ be the current partial cover.

On the other hand, consider the set selection of an optimal solution. This selection may choose some of the sets $S_1^G, \ldots, S_t^G$ and, additionally, consists of sets $S_1^*, \ldots, S_l^*$ to form a full cover.

3.6 Set Cover

For i=1,…,l, let $n_i^* = |S_i^* - C_t|$ be the number of elements newly covered by S_i^* w.r.t. C_t. Observe that $n_1^* + \cdots + n_l^* \geq |U-C_t|$.

There holds:

$$\mathrm{OPT} \geq c(S_1^*) + \ldots + c(S_l^*) = \frac{c(S_1^*)}{n_1^*} \cdot n_1^* + \ldots + \frac{c(S_l^*)}{n_l^*} \cdot n_l^*$$

In the last sum, the ratios represent the cost-effectiveness of the respective sets. One of these sets must have a cost-effectiveness of at most $\mathrm{OPT}/|U-C_t|$ since otherwise

$$\mathrm{OPT} > \frac{\mathrm{OPT}}{|U-C_t|}(n_1^* + \ldots + n_l^*) \geq \mathrm{OPT}.$$

3.6 Set Cover

It follows that S_{t+1}^{G} has a cost-effectiveness of at most $OPT/|U-C_t|$ because Greedy always chooses a set with the best cost-effectiveness.

Immediately before selecting S_{t+1}^{G}, Greedy has covered at most k-1 elements so that $|U-C_t| \geq n-(k-1)$.

We conclude that e_k is charged a cost of at most $OPT/(n-k+1)$. □

In order to finish the proof of the theorem, we observe that the cost of Greedy is $\Sigma_{1 \leq k \leq n} \text{price}(e_k) \leq \Sigma_{1 \leq k \leq n} OPT/(n-k+1) = H_n \cdot OPT$.

3.6 Set Cover

Theorem: The approximation factor of Greedy is not smaller than H_n.

Proof: Consider an input instance with universe $U = \{u_1,\ldots,u_n\}$ and sets $S_1,\ldots,S_{n+1}$. For $i=1,\ldots,n$, set S_i contains element u_i and has a cost $c(S_i)=1/i$.

Set S_{n+1} contains all the elements $u_1,\ldots,u_n$ and has a cost of $c(S_{n+1})=1+\varepsilon$.

Greedy will iteratively choose the sets $S_n,\ldots,S_1$, incurring a cost of H_n.

An optimal solution picks S_{n+1}, paying a cost of $1+\varepsilon$ only.

3.7 Duality in linear programming

Motivation: Estimate the optimum objective function value of a given (primal) LP without actually solving the LP. Thereby construct an associated dual LP.

Example

$$\begin{aligned}\min\quad & 7x_1 + x_2 + 5x_3 \\ \text{s.t.}\quad & x_1 - x_2 + 3x_3 \geq 10 \\ & 5x_1 + 2x_2 - x_3 \geq 6 \\ & x_1, x_2, x_3 \geq 0\end{aligned}$$

Standard form for minimization problem:

In the contraints, inequalities are „≥“

All variable are non-negative

Feasible solution: vector $(x_1,\ldots,x_n)$ satisfying all the constraints

z^*: value of optimal solution

3.7 Duality in linear programming

Upper bound: $z^* \le 30$? Certificate (2,1,3)

Upper bound: $z^* \ge 10$? Seems harder to verify.

First constraint is a certificate.

Better: Add both contraints.

$$7x_1 + x_2 + 5x_3 \ge (x_1 - x_2 + 3x_3) + (5x_1 + 2x_2 - x_3) \ge 10 + 6$$

Systematically: Find multiplies for the constraints such that, for each variable, the weighted sum of the coefficients is a lower bound on the coefficient in the objective function.

Dual LP: max $10y_1 + 6y_2$

$$\begin{aligned} \text{s.t.} \quad y_1 + 5y_2 &\le 7 \\ -y_1 + 2y_2 &\le 1 \\ 3y_1 - y_2 &\le 5 \\ y_1, y_2 &\ge 0 \end{aligned}$$

3.7 Duality in linear programming

General programs:

Primal Program

min $\sum_{j=1}^{n} c_j x_j$

s.t. $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ $i=1,\ldots,m$

$x_j \geq 0$ $j=1,\ldots,n$

Dual Program

max $\sum_{i=1}^{m} b_i y_i$

s.t. $\sum_{i=1}^{m} a_{ij} y_i \leq c_j$ $j=1,\ldots,n$

$y_i \geq 0$ $i=1,\ldots,m$

3.7 Duality in linear programming

Dual of the dual program gives again the primal program.

By the construction of dual programs:

- Any feasible solution to the dual program is a lower bound for the primal program.
- Any feasible solution to the primal program is an upper bound for the dual program.

Let $(x_1, \ldots, x_n)$ be a feasible solution to the primal program.
Let $(y_1, \ldots, y_m)$ be a feasible solution to the dual program.
If they lead to the same objective function value, then they are optimal.

3.7 Duality in linear programming

Theorem: Weak Duality

Let $(x_1,\dots, x_n)$ and $(y_1,\dots, y_m)$ be feasible solutions to the primal and dual programs, respectively. Then $\sum_{j=1}^{n} c_j x_j \geq \sum_{i=1}^{m} b_i y_i$.

Proof: There holds

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i \geq \sum_{i=1}^{m} b_i y_i .$$

The first inequality holds because $(y_1,\dots,y_m)$ satisfies the constraints of the dual program and the $x_1,\dots, x_n$ are non-negative.

The second inequality holds because $(x_1,\dots, x_n)$ satisfies the constraints of the primal program and the $y_1,\dots, y_m$ are non-negative.

3.7 Duality in linear programming

Theorem: LP-Duality

The primal program has a finite optimum if and only if the dual program has a finite optimum.

Vectors x*= $(x_1^*, \dots, x_n^*)$ and y*= $(y_1^*, \dots, y_m^*)$ are optimal solutions if and only if $\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*$.

Theorem: Complementary Slackness

Let x=$(x_1, \dots, x_n)$ and y= $(y_1, \dots, y_m)$ be feasible solutions to the primal and dual programs, respectively. The solutions x, y are optimal if and only if the following conditions hold.

Primal slackness conditions: For each j = 1,…,n there holds

$$x_j = 0 \quad \text{or} \quad \sum_{i=1}^{m} a_{ij} y_i = c_j$$

Dual slackness conditions: For each i = 1,…,m there holds

$$y_i = 0 \quad \text{or} \quad \sum_{j=1}^{n} a_{ij} x_j = b_i$$

3.7 Dual fitting technique

Consider a minimization problem (analogously maximization problem)

1. (P) Primal program; val(x*) = value of an optimal solution x*.
   (D) Dual program
2. Compute solution x for (P) and vector y for (D), which may be infeasible, such that val(x) ≤ val'(y), where val'(y) is the objective function value of (D).
3. Divide y by α such that y' = y / α is feasible for (D). Then val'(y') ≤ val(x*).
4. Technique achieves an approximation factor of α because

$$\mathrm{val}(x) \le \mathrm{val}'(y) = \mathrm{val}'(\alpha\, y') = \alpha\, \mathrm{val}'(y') \le \alpha\, \mathrm{val}(x^*).$$

3.7 Set Cover and LP

Problem: Universe $U = \{u_1,\dots,u_n\}$. Sets $S_1,\dots,S_m \subseteq U$ with associated non-negative costs $c(S_1),\dots,c(S_m)$. Find $J \subseteq \{1,\dots,m\}$ such that $\bigcup_{j\in J} S_j = U$ and $\sum_{j\in J} c(S_j)$ minimal.

Formulation as LP: Set system $\Sigma = \{S_1,\dots,S_m\}$

(P) min $\Sigma_{S\in\Sigma}\ c(S)\ x_S$

s.t. $\Sigma_{S:\ e\in S}\, x_S \geq 1$ $\quad e \in U$

$x_S \in \{0,1\}$ $\quad S \in \Sigma$ $\quad$ relaxed to $x_S \in [0,1]$

(D) max $\Sigma_{e\in U}\ y_e$

s.t. $\Sigma_{e\in S}\, y_e \leq c(S)$ $\quad S \in \Sigma$

$y_e \geq 0$ $\quad e \in U$

Intuitively: Want to pack elements into sets s.t. cost of the sets is observed.

3.7 Set Cover and LP

Greedy approach: Repeatedly choose the most cost-effective set. At any time let C be the set of covered elements. Cost-effectiveness of S is $c(S) / |S-C|$.

Algorithm Greedy:

1. C:= Ø;
2. **while** $C \neq U$ **do**
3. Determine the set S having the smallest ratio $\alpha = c(S) / |S-C|$;
4. Choose S and set price(e) := α, for all $e \in S-C$;
5. $C := C \cup S$;
6. **endwhile;**
7. Output the selected sets;

Theorem: Greedy achieves an approximation factor of H_n.

3.7 Set Cover and LP

Theorem: Greedy achieves an approximation factor of H_n.

Proof: We verify the property specified in Step 2 of the dual fitting approach. Given the solution computed by Greedy, define $x_S = 1$ for each set S that is selected by Greedy. For all other sets $S \in \Sigma$, define $x_S = 0$.

For each $e \in U$, define $y_e = \text{price}(e)$ as specified by Greedy.

There holds

$$\text{val}(x) = \Sigma_{S \in \Sigma}\, c(S)\, x_S = \Sigma_{e \in U}\, \text{price}(e) = \Sigma_{e \in U}\, y_e = \text{val}(y),$$

i.e. the required inequality is satisfied with equality.

It remains to take care of Step 3.

3.7 Set Cover and LP

Lemma: For each $S \in \Sigma$, there holds $\Sigma_{e \in S}$ price(e) $\leq H_n \cdot c(S)$.

Proof: Consider any set S and let k be the number of elements in S. Number the elements $e_1,\ldots,e_k$ in the order in which they get covered by Greedy.

Consider the point in time when e_i gets covered, $1 \leq i \leq k$. At that time at most k-(i-1) elements of S are covered so that the cost-effectiveness of S is upper bounded by $c(S)/(k-i+1)$. If Greedy does not pick S, it selects a set whose cost-effectiveness is at most $c(S)/(k-i+1)$ and thus price(e_i) $\leq c(S)/(k-i+1)$.

We conclude that $\Sigma_{e \in S}$ price(e) $\leq \Sigma_{1 \leq i \leq k}\, c(S)/(k-i+1) = H_k \cdot c(S) \leq H_n \cdot c(S)$. □

Set $y'_e = y_e/H_n$. Then by the above lemma,

$$\Sigma_{e \in S}\, y'_e = \Sigma_{e \in S}\, y_e/H_n = \Sigma_{e \in S}\, \text{price}(e)/H_n \leq c(S).$$

3.7 Primal-dual algorithms

Repeatedly modify the primal and dual solutions until relaxed complementary slackness conditions hold.

(P) min $\sum_{j=1}^{n} c_j \, x_j$

s.t. $\sum_{j=1}^{n} a_{ij} \, x_j \geq b_i$ i=1,…,m

$x_j \geq 0$ j=1,…,n

(D) max $\sum_{i=1}^{m} b_i \, y_i$

s.t. $\sum_{i=1}^{m} a_{ij} \, y_i \leq c_j$ j=1,…,n

$y_i \geq 0$ i=1,…,m

Relaxed primal slackness conditions: Let α≥1. For each j = 1,…,n, there holds

$$x_j = 0 \quad \text{or} \quad c_j/\alpha \leq \sum_{i=1}^{m} a_{ij} y_i \leq c_j$$

Relaxed dual slackness conditions: Let ß≥1. For each i = 1,…,m, there holds

$$y_i = 0 \quad \text{or} \quad b_i \leq \sum_{j=1}^{n} a_{ij} x_j \leq ß \, b_i$$

3.7 Primal-dual algorithms

Theorem: Let x,y be feasible primal and dual solutions satisfying the relaxed complementary slackness conditions. Then val(x) ≤ αß val'(y).
Hence val(x) ≤ αß val(x*).

Proof: There holds

val(x) = $\sum_{j=1}^{n} c_j x_j \leq \alpha \sum_{j=1}^{n}\left(\sum_{i=1}^{m} a_{ij} y_i\right) x_j = \alpha \sum_{i=1}^{m}\left(\sum_{j=1}^{n} a_{ij} x_j\right) y_i \leq \alpha\beta \sum_{i=1}^{m} b_i y_i$

= αß val'(y).

3.7 Primal-dual algorithms

General scheme:

- Many algorithms work with $\alpha = 1$ or $\beta = 1$.
- Algorithm starts with non-feasible primal „solution" and feasible dual solution, e.g. $x=0$ and $y=0$.
- In each iteration one improves the feasibility of the primal solution and the optimality of the dual solution until the primal solution is feasible and the relaxed complementary slackness conditions hold.
- Primal solution is always modified such that it remains integral. Modifications of the primal and dual solutions are done in a synchronized way.

3.7 Primal-dual algorithm for Set Cover

(P) min val(x) = $\Sigma_{S\in\Sigma}$ c(S) x_S

s.t. $\Sigma_{S:\, e\in S}\, x_S \geq 1 \qquad e \in U$

$x_S \in [0,1] \qquad S \in \Sigma$

(D) max $\Sigma_{e\in U}\, y_e$

s.t. $\Sigma_{e\in S}\, y_e \leq c(S) \qquad S \in \Sigma$

$y_e \geq 0 \qquad e \in U$

Choose α = 1 and ß = f   f = frequency of the element occurring most often in any set

Set is called dense if $\Sigma_{e\in S}\, y_e = c(S)$

Relaxed primal slackness conditions: For S $\in \Sigma$, x_S=0 or $\Sigma_{e\in S}\, y_e = c(S)$

Intuitively, cover contains only dense sets.

Relaxed dual slackness conditions: For e $\in$ U, $y_e = 0$ or $1 \leq \Sigma_{S:\, e\in S}\, x_S \leq f$

Intuitively, each element is covered at most f times.

3.7 Primal-dual algorithm for Set Cover

Algorithm:

1. Set $x=0$ and $y=0$. No element is covered.
2. **while** there exists an uncovered element e **do**

    (a) Increase y_e until a set S is dense;

    (b) Add all dense sets S to the cover and set $x_S=1$;

    (c) Elements of all sets of (b) are covered;

   endwhile;
3. Output x;

Theorem: The above algorithm achieves an approximation factor of f.

Theorem: The approximation factor of the above algorithm is not smaller than f.

3.7 Primal-dual algorithm for Set Cover

Theorem: The above algorithm achieves an approximation factor of f.

Proof: When the algorithm terminates, all elements are covered.

No constraint of (D) is violated since dense sets are added to the cover and, for the corresponding elements e, the dual variable y_e is not increased any further. Hence vectors x and y are feasible.

Since dense sets are added to the cover, the relaxed primal slackness conditions hold. Each element occurs in at most f sets so that the relaxed primal slackness conditions hold as well.

3.7 Primal-dual algorithm for Set Cover

Theorem: The approximation factor of the above algorithm is not smaller than f.

Proof: Let $U=\{e_1,\ldots,e_{n+1}\}$.

There are n sets.

For $i=1,\ldots n-1$, set $S_i = \{e_i,e_n\}$ and $c(S_i)=1$.

Moreover, $S_n=\{e_1,\ldots,e_{n+1}\}$ and $c(S_n)=1+\varepsilon$, for arbitrary $\varepsilon>0$.

There holds f=n.

Iteration 1: Algorithm raises y_{e_n} to 1 and adds $S_1,\ldots,S_{n-1}$ to the cover.
Iteration 2: Algorithm raises $y_{e_{n+1}}$ to ε and adds S_n to the cover.

The algorithm incurs a total cost of $n+\varepsilon$, while the optimum solution picks S_{n+1} and has a cost of $1+\varepsilon$ only.

3.8 Shortest Superstring

Problem: Σ finite alphabet, n strings $S = \{s_1, \ldots, s_n\}$. Find shortest string s such that all s_i of S are substring of s. W.l.o.g. no s_i is substring of any s_j, where $i \neq j$.

Example: S = {ate, half, lethal, alpha, alfalfa} s = lethalalphalfalfate

3.8 Shortest Superstring

Reduction to Set Cover: Let s_i, s_j be strings such that the last k characters of s_i are equal to the first k characters of s_j.

σ_{ijk} = composition of s_i and s_j, with an overlap of k characters, where $k \geq 1$

M = set of all σ_{ijk}, for all feasible combinations of i,j and $k \geq 1$

$U = \{s_1, \ldots, s_n\}$

Sets: set(π) for all $\pi \in M \cup U$ where

$\text{set}(\pi) = \{s_i \in U \mid s_i \text{ is substring of } \pi\}$

cost of set(π) is equal to $|\pi|$

3.8 Shortest Superstring

Algorithm (Shortest Superstring via Set Cover):

1. Apply the Greedy algorithm for Set Cover to the above Set Cover instance. Let $set(\pi_1), \ldots, set(\pi_k)$ be the selected sets.
2. Concatenate $\pi_1, \ldots, \pi_k$ in an arbitrary order and output the resulting string.

Lemma: The above algorithm outputs a feasible solution.

Lemma: There holds $OPT_{SC} \leq 2\ OPT$, where OPT is the length of the shortest superstring and OPT_{SC} is the optimum solution to the Set Cover instance.

Corollary: The above algorithm achieves an approximation factor of $2H_n$.

3.8 Shortest Superstring

Lemma: The above algorithm outputs a feasible solution.

Proof: Each string s_i, $1 \leq i \leq n$, is contained in at least one of the sets set(π), where $\pi \in M \cup U$. In Step 1 of the algorithm a cover of U is computed. Hence each s_i is contained in at least one of the sets set(π_1), … , set(π_k) determined in this step. It follows that each s_i, $1 \leq i \leq n$, is substring of at least one of the strings $\pi_1, \ldots, \pi_k$.

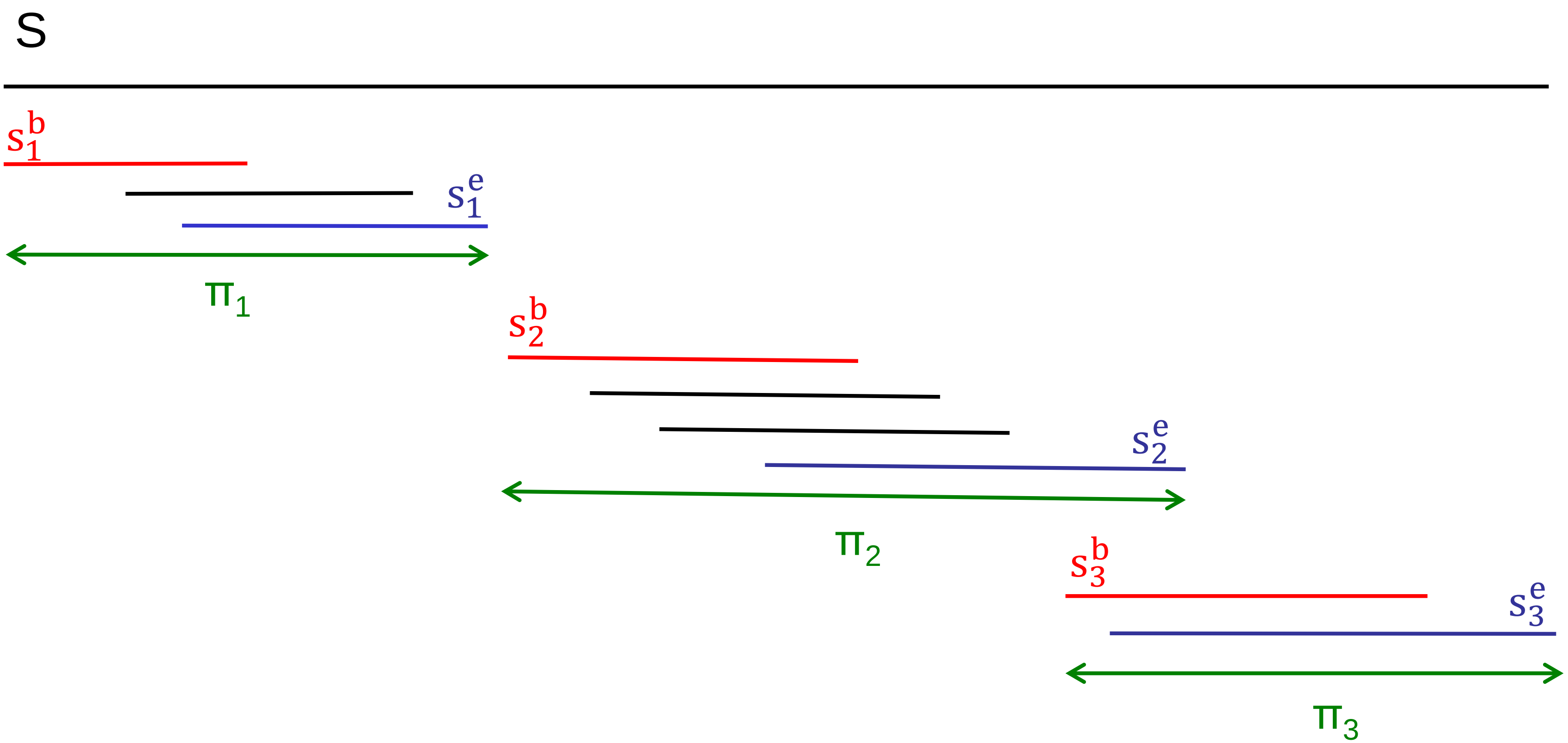
3.8 Shortest Superstring

Lemma: There holds $OPT_{SC} \leq 2 \cdot OPT$, where OPT is the length of the shortest superstring and OPT_{SC} is the cost optimum solution to the Set Cover instance.

Proof: Let S be a shortest superstring for $s_1,\ldots,s_n$. We will show that there exists a solution to the Set Cover instance whose cost is upper bounded by $2|S|$.

In S mark the first occurrence of each of the strings $s_1,\ldots,s_n$. Recall that no string is substring of another string. Therefore, the startpoints of these strings are distinct. Similarly, the endpoints are distinct.

3.8 Shortest Superstring



3.8 Shortest Superstring

We next group the strings according to their occurrence in S; cf. the figure on previous page.

s_1^b = first string occurring in S $\qquad$ s_1^e = last string overlapping with s_1^b

For any i ≥ 2:

s_i^b = first string occurring after s_{i-1}^e $\qquad$ s_i^e = last string overlapping with s_i^b

For any i ≥ 1: π_i = range between s_i^b and s_i^e

Each of the strings $s_1,\ldots,s_n$ is substring some π_i, i ≥1. Therefore the sets set(π_i), i ≥1, form a solution to the Set Cover instance.

Observe that, for any i ≥ 1, string π_{i+2} does not overlap with π_i: The first string contained in π_{i+2} does not overlap with the first string in π_{i+1} and hence with no string in π_i.

We conclude $\Sigma_{i \text{ odd}} |\pi_i| \leq |S|$ and $\Sigma_{i \text{ even}} |\pi_i| \leq |S|$.

3.9 Minimum-Degree Spanning Tree

Minimum-Degree Spanning Tree: Given an undirected unweighted graph $G=(V,E)$, find a spanning tree T of G so as to minimize the maximum degree of vertices in T.

Notation:

$n = |V|$

$d_T(u)$ = degree of vertex u in tree T

$\Delta(T) = \max_{u \in V} d_T(u)$

T* = spanning tree that minimizes the maximum degree

$OPT = \Delta(T^*)$

3.9 Minimum-Degree Spanning Tree

Improving Pair: [u; (v,w)]

T ∪ (v,w) creates a cycle C containing u.

$\max \{d_T(v), d_T(w)\} \leq d_T(u) - 2$

l = $\lceil \log_2 n \rceil$

Algorithm Local Improvement:

1. T := arbitrary spanning tree;
2. **while** there exists an improving pair [u; (v,w)] with $d_T(u) \geq \Delta(T) - l$ **do**
3. Add (v,w) to T;
4. Delete an edge incident to u on the cycle created;
5. **endwhile;**
6. Output the resulting locally optimal tree;

3.9 Minimum-Degree Spanning Tree

Theorem: Let T be a locally optimal tree. Then $\Delta(T) \leq 2\ \mathrm{OPT} + l$.

Theorem: The algorithm finds a locally optimal tree in polynomial time.

3.9 Minimum-Degree Spanning Tree

Theorem: Let T be a locally optimal tree. Then $\Delta(T) \leq 2$ OPT + l.

Proof: We develop a lower bound on OPT that relates to $\Delta(T)$.

General approach: Identify and delete k specific edges in T. This breaks T into k+1 components. Moreover, identify a set S of vertices such that every edge of G connecting different components is incident to a least one vertex of S.

Tree T* contains at least k edges with endpoints in different components because T* connects all vertices of V. By the choice of S, each such edge is incident to at least one vertex of S. Therefore, the average degree of vertices of S in T* is at least $k/|S|$. Thus OPT $\geq k/|S|$.

In order to apply this general approach, we make use of the following claim.

3.9 Minimum-Degree Spanning Tree

S_i = set of vertices of degree at least i in T.

Claim: (a) For each S_i, where $i \geq \Delta(T) - l$, there exist at least $(i-1)|S_i|+1$ distinct edges of T incident on vertices in S_i. After removing these edges, each edge of G that connects distinct components is incident to at least one vertex in S_{i-1}.

(b) There exists an $i \geq \Delta(T) - l+1$ such that $\frac{1}{2}\cdot|S_{i-1}| \leq |S_i|$.

Let i be an integer satisfying part (b) of the claim. Delete all the edges of T incident to vertices in S_i. Part (a) implies that $k = (i-1)|S_i|+1$ distinct edges are deleted and that every edge of G connecting different components is incident to a vertex in $S = S_{i-1}$. Using again part (b) we obtain

$$OPT \geq k/|S| = ((i-1))|S_i|+1) / |S_{i-1}| > (i-1)|S_i| / |S_{i-1}| \geq (i-1)|S_{i-1}| / (2|S_{i-1}|) = (i-1)/2$$
$$\geq (\Delta(T) - l)/2,$$

which is equivalent to $\Delta(T) \leq 2\ OPT + l$.

3.9 Supplement: Proof of the Claim

Part (a): In T the total degree of vertices in S_i is at least $i|S_i|$. There exist at most $|S_i|-1$ edges in T having both endpoints in $|S_i|$ because T does not contain a cycle. (Observe that by adding m edges to set of m vertices, one creates a cycle.) Hence the total number of distinct edges incident to vertices is S_i is at least $i|S_i| - (|S_i|-1) = (i-1)|S_i|+1$.

In T remove the edges incident to vertices in S_i. Consider any edge e =(v,w) connecting different components. There are two cases.

- Edge e belongs to T: In this case e is one of the edges removed. At least one endpoint is in S_i and hence in S_{i-1}.
- In T edge e closes a cycle: This cycle must contain a vertex u of S_i because e connects different components, which are linked in T using the removed edges. Since [u; (v,w)] is not an improving pair, there holds $d_T(v) \geq d_T(u) - 1$ or $d_T(w) \geq d_T(w) - 1$. Thus $v \in S_{i-1}$ or $w \in S_{i-1}$.

3.9 Supplement: Proof of the Claim

Part (b): Suppose that $\frac{1}{2} \cdot |S_{i-1}| > |S_i|$ holds for $i = \Delta(T) - l+1,\ldots, \Delta(T)$. Then

$$|S_{\Delta(T) - l}| > 2\, |S_{\Delta(T) - l+1}| > 2^2\, |S_{\Delta(T) - l+2}| > \ldots > 2^l\, |S_{\Delta(T)}| \geq n.$$

The last inequality holds because $l = \lceil \log_2 n \rceil$ and $|S_{\Delta(T)}| \geq 1$. We obtain $|S_{\Delta(T) - l}| > n$, which is a contradiction.

3.9 Supplement: Analysis running time

Theorem: The algorithm finds a locally optimal tree in polynomial time.

Proof: Define a potential function. For any T, let $\Phi(T) = \sum_{v \in V} 3^{d_T(v)}$.

The initial potential is upper bounded by $n3^n$. The lowest potential is attained for a path, having a potential of $2 \cdot 3+(n-2)3^2$. The latter value is greater than n, for $n \geq 2$.

We will show that, for any improving move changing a tree T into T', there holds $\Phi(T') \leq (1 - 2/(27n^3))\,\Phi(T)$.

The number of steps / improving moves by the algorithm is then upper bounded by $k = 27n^4 \ln(3) / 2$ because

$$n3^n \cdot (1 - 2/(27n^3))^k \leq n3^n \cdot e^{-2k/(27n^3)} = n3^n \cdot e^{-n \ln 3} = n.$$

The inequality holds because $1-x \leq e^{-x}$, for $x \in [0,1]$.

3.9 Supplement: Analysis running time

Consider an improving move [u; (v,w)], changing the current tree T into T'.

At u the degree decreases from, say, i to i-1. The resulting potential change is $-3^i+3^{i-1} = -2\cdot3^{i-1}$.

Consider any of the two vertices v and w. The degree of the vertex increases from, say, j to j+1, resulting in a potential increase of $3^{j+1}-3^j = 2\cdot3^j \leq 2\cdot3^{i-2}$. The last inequality holds because the degree of both v and w is by at least 2 smaller than the degree of u. For the two vertices v and w, the total increase in potential is upper bounded by $4\cdot3^{i-2}$.

Thus $\Phi(T') - \Phi(T) \leq -2\cdot3^{i-1} + 4\cdot3^{i-2} = -2\cdot3^{i-2} \leq -2\cdot3^{\Delta(T)-l-2}$. There holds

$$3^l \leq 3 \cdot 3^{\log_2 n} \leq 3 \cdot 4^{\log_2 n} \leq 3 \cdot 2^{2\log_2 n} = 3n^2.$$

We conclude $\Phi(T') - \Phi(T) \leq -2/(27n^2)\cdot3^{\Delta(T)} = -2/(27n^3)\cdot n\cdot3^{\Delta(T)} \leq -2/(27n^3)\cdot\Phi(T)$ because $\Phi(T) \leq n\cdot3^{\Delta(T)}$.