Streaming Set Cover in Practice

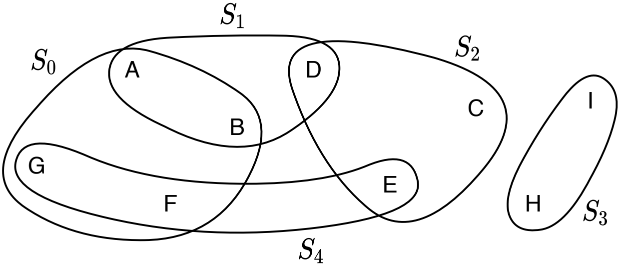
Michael Barlow[1] Christian Konrad[1] Charana Nandasena[2]

[1]Department of Computer Science,
University of Bristol, UK
{michael.barlow,christian.konrad}@bristol.ac.uk

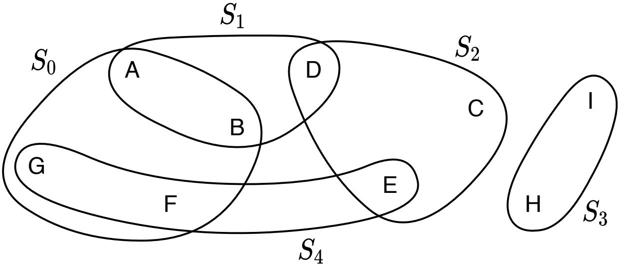
[2]Melbourne School of Engineering
University of Melbourne, Australia
anandasena@student.unimelb.edu.au

Symposium on Algorithm Engineering and Experiments
(ALENEX21), January 2021

Example Set Cover Instance

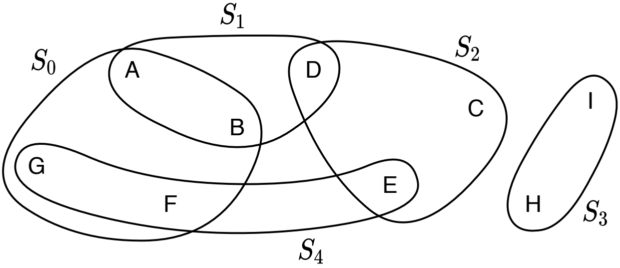


Example Set Cover Instance



$\mathcal{U}$	$\{A, B, C, D, E, F, G, H, I\}$
$\mathcal{S}$	$\{S_0, S_1, S_2, S_3, S_4\}$

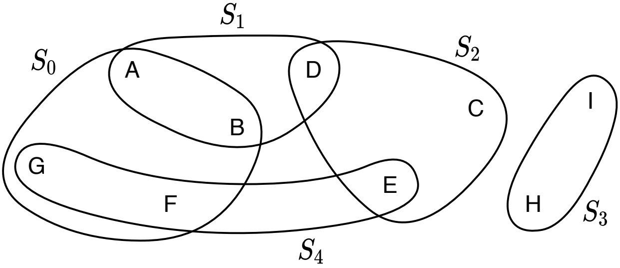
Example Set Cover Instance



$$\begin{array}{c|l} \mathcal{U} & \{A, B, C, D, E, F, G, H, I\} \\ \mathcal{S} & \{S_0, S_1, S_2, S_3, S_4\} \end{array}$$

$n = |\mathcal{U}|$,

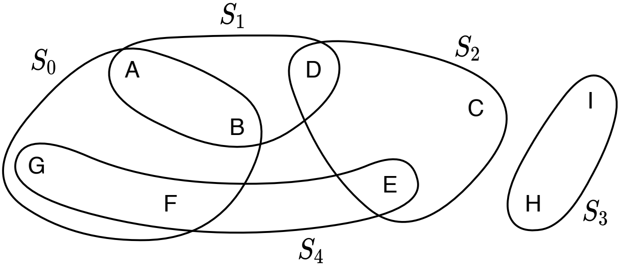
Example Set Cover Instance



$\mathcal{U}$	$\{A, B, C, D, E, F, G, H, I\}$
$\mathcal{S}$	$\{S_0, S_1, S_2, S_3, S_4\}$

$n = |\mathcal{U}|$, $m = |\mathcal{S}|$,

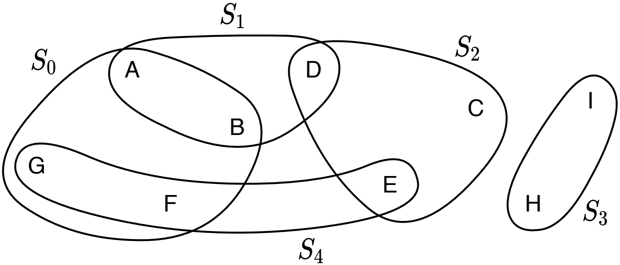
Example Set Cover Instance



$\mathcal{U}$	$\{A, B, C, D, E, F, G, H, I\}$
$\mathcal{S}$	$\{S_0, S_1, S_2, S_3, S_4\}$

$n = |\mathcal{U}|$, $m = |\mathcal{S}|$, $\Delta = \max(|S_i|)$

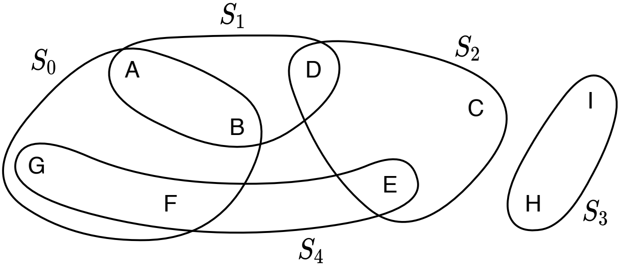
Example Set Cover Instance



$$\begin{array}{c|l} \mathcal{U} & \{A, B, C, D, E, F, G, H, I\} \\ \mathcal{S} & \{S_0, S_1, S_2, S_3, S_4\} \end{array}$$

$n = |\mathcal{U}|$, $m = |\mathcal{S}|$, $\Delta = \max(|S_i|)$ and $M = \sum_i |S_i|$

Example Set Cover Instance



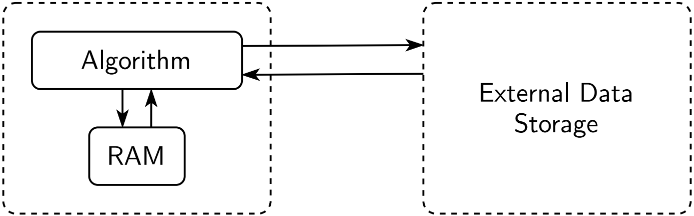
$\mathcal{U}$	$\{A, B, C, D, E, F, G, H, I\}$
$\mathcal{S}$	$\{S_0, S_1, S_2, S_3, S_4\}$

$n = |\mathcal{U}|$, $m = |\mathcal{S}|$, $\Delta = \max(|S_i|)$ and $M = \sum_i |S_i|$

Optimal cover: $\{S_0, S_2, S_3\}$

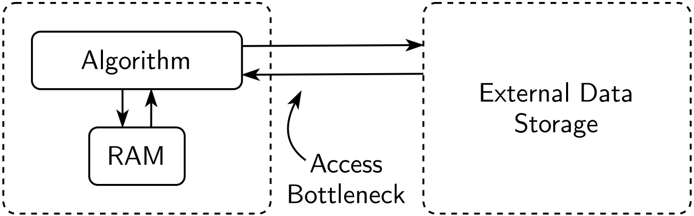
Memory Access

Direct access:



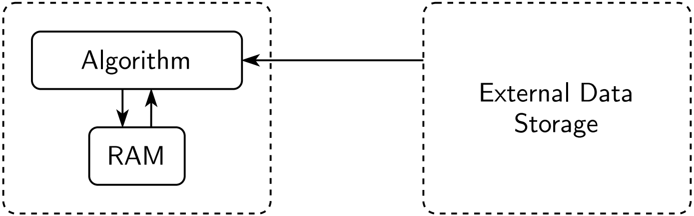
Memory Access

Direct access:

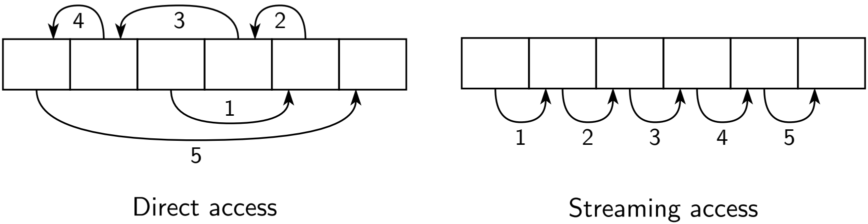


Memory Access

Streaming access:



Accessing Sets



GREEDY: Theory

GREEDY

Greedy: Theory

Greedy

1. Find the set with the most uncovered elements

Greedy: Theory

Greedy

1. Find the set with the most uncovered elements
2. Add this set to the solution

Greedy: Theory

Greedy

1. Find the set with the most uncovered elements
2. Add this set to the solution
3. Repeat until universe is covered

GREEDY: Theory

GREEDY

1. Find the set with the most uncovered elements
2. Add this set to the solution
3. Repeat until universe is covered

Approximation factor of $\mathcal{O}(\ln n)$, which is essentially optimal [5, 3]

GREEDY: Implementation

Finding the set with the highest *contribution*

Greedy: Implementation

Finding the set with the highest *contribution*

- Multiple passes over the input

GREEDY: Implementation

Finding the set with the highest *contribution*

- Multiple passes over the input
  - Sequential memory access

GREEDY: Implementation

Finding the set with the highest *contribution*

- Multiple passes over the input
  - Sequential memory access
  - Time complexity depends on solution size: $\mathcal{O}(M \cdot |\text{Solution}|)$

GREEDY: Implementation

Finding the set with the highest *contribution*

- Multiple passes over the input
  - Sequential memory access
  - Time complexity depends on solution size: $\mathcal{O}(M \cdot |\text{Solution}|)$
- Maintain a priority queue

Greedy: Implementation

Finding the set with the highest *contribution*

- Multiple passes over the input
  - Sequential memory access
  - Time complexity depends on solution size: $\mathcal{O}(M \cdot |\text{Solution}|)$
- Maintain a priority queue ...and an inverted index

Greedy: Implementation

Finding the set with the highest *contribution*

- Multiple passes over the input
  - Sequential memory access
  - Time complexity depends on solution size: $\mathcal{O}(M \cdot |\text{Solution}|)$
- Maintain a priority queue ...and an inverted index
  - Time complexity of $\mathcal{O}(M \log m)$

GREEDY: Implementation

Finding the set with the highest *contribution*

- Multiple passes over the input
  - Sequential memory access
  - Time complexity depends on solution size: $\mathcal{O}(M \cdot |\text{Solution}|)$
- Maintain a priority queue ...and an inverted index
  - Time complexity of $\mathcal{O}(M \log m)$
  - Complicated

GREEDY: Implementation

Finding the set with the highest *contribution*

- Multiple passes over the input
  - Sequential memory access
  - Time complexity depends on solution size: $\mathcal{O}(M \cdot |\text{Solution}|)$
- Maintain a priority queue ...and an inverted index
  - Time complexity of $\mathcal{O}(M \log m)$
  - Complicated
  - Precomputation step

GREEDY: Implementation

Finding the set with the highest *contribution*

- Multiple passes over the input
  - Sequential memory access
  - Time complexity depends on solution size: $\mathcal{O}(M \cdot |\text{Solution}|)$
- Maintain a priority queue ...and an inverted index
  - Time complexity of $\mathcal{O}(M \log m)$
  - Complicated
  - Precomputation step
  - Inverted index doubles footprint

GREEDY: Implementation

Finding the set with the highest *contribution*

- Multiple passes over the input
  - Sequential memory access
  - Time complexity depends on solution size: $\mathcal{O}(M \cdot |\text{Solution}|)$
- Maintain a priority queue ...and an inverted index
  - Time complexity of $\mathcal{O}(M \log m)$
  - Complicated
  - Precomputation step
  - Inverted index doubles footprint
  - Arbitrary memory access pattern

GREEDY: Implementation

Finding the set with the highest *contribution*

- Multiple passes over the input
  - Sequential memory access
  - Time complexity depends on solution size: $\mathcal{O}(M \cdot |\text{Solution}|)$
- Maintain a priority queue ...and an inverted index
  - Time complexity of $\mathcal{O}(M \log m)$
  - Complicated
  - Precomputation step
  - Inverted index doubles footprint
  - Arbitrary memory access pattern

Alternatives?

1: Disk Friendly Greedy (DFG) [2]

Aim: to access memory in large, contiguous chunks

1: Disk Friendly Greedy (DFG) [2]

Aim: to access memory in large, contiguous chunks

Disk Friendly Greedy [2]

1: Disk Friendly Greedy (DFG) [2]

Aim: to access memory in large, contiguous chunks

Disk Friendly Greedy [2]

1. (Preprocess) Write sets to files on disk such that the k^{th} file contains all sets with sizes in the range $[p^k, p^{k+1})$

1: Disk Friendly Greedy (DFG) [2]

Aim: to access memory in large, contiguous chunks

Disk Friendly Greedy [2]

1. (Preprocess) Write sets to files on disk such that the k^{th} file contains all sets with sizes in the range $[p^k, p^{k+1})$
2. For each file, starting with the one with the largest k:

1: Disk Friendly Greedy (DFG) [2]

Aim: to access memory in large, contiguous chunks

Disk Friendly Greedy [2]

1. (Preprocess) Write sets to files on disk such that the k^{th} file contains all sets with sizes in the range $[p^k, p^{k+1})$
2. For each file, starting with the one with the largest k:
   2.1 Add sets to the solution if their contribution is at least p^k

1: Disk Friendly Greedy (DFG) [2]

Aim: to access memory in large, contiguous chunks

Disk Friendly Greedy [2]

1. (Preprocess) Write sets to files on disk such that the k^{th} file contains all sets with sizes in the range $[p^k, p^{k+1})$
2. For each file, starting with the one with the largest k:
   - 2.1 Add sets to the solution if their contribution is at least p^k
   - 2.2 Remaining sets are reduced and reassigned to respective files

1: Disk Friendly Greedy (DFG) [2]

Aim: to access memory in large, contiguous chunks

Disk Friendly Greedy [2]

1. (Preprocess) Write sets to files on disk such that the k^{th} file contains all sets with sizes in the range $[p^k, p^{k+1})$
2. For each file, starting with the one with the largest k:
   2.1 Add sets to the solution if their contribution is at least p^k
   2.2 Remaining sets are reduced and reassigned to respective files
3. Sets in the final file are added if their contribution is non-zero

1: Disk Friendly Greedy (DFG) [2]

Aim: to access memory in large, contiguous chunks

Disk Friendly Greedy [2]

1. (Preprocess) Write sets to files on disk such that the k^{th} file contains all sets with sizes in the range $[p^k, p^{k+1})$
2. For each file, starting with the one with the largest k:
   2.1 Add sets to the solution if their contribution is at least p^k
   2.2 Remaining sets are reduced and reassigned to respective files
3. Sets in the final file are added if their contribution is non-zero

- Largely sequential access pattern

1: Disk Friendly Greedy (DFG) [2]

Aim: to access memory in large, contiguous chunks

Disk Friendly Greedy [2]

1. (Preprocess) Write sets to files on disk such that the k^{th} file contains all sets with sizes in the range $[p^k, p^{k+1})$
2. For each file, starting with the one with the largest k:
   2.1 Add sets to the solution if their contribution is at least p^k
   2.2 Remaining sets are reduced and reassigned to respective files
3. Sets in the final file are added if their contribution is non-zero

- Largely sequential access pattern
- Approximation factor of $1 + p \ln n$ (close to optimal)

1: Disk Friendly Greedy (DFG) [2]

Aim: to access memory in large, contiguous chunks

Disk Friendly Greedy [2]

1. (Preprocess) Write sets to files on disk such that the k^{th} file contains all sets with sizes in the range $[p^k, p^{k+1})$
2. For each file, starting with the one with the largest k:
   2.1 Add sets to the solution if their contribution is at least p^k
   2.2 Remaining sets are reduced and reassigned to respective files
3. Sets in the final file are added if their contribution is non-zero

- Largely sequential access pattern
- Approximation factor of $1 + p \ln n$ (close to optimal)
- Precomputation step

1: Disk Friendly Greedy (DFG) [2]

Aim: to access memory in large, contiguous chunks

Disk Friendly Greedy [2]

1. (Preprocess) Write sets to files on disk such that the k^{th} file contains all sets with sizes in the range $[p^k, p^{k+1})$
2. For each file, starting with the one with the largest k:
   2.1 Add sets to the solution if their contribution is at least p^k
   2.2 Remaining sets are reduced and reassigned to respective files
3. Sets in the final file are added if their contribution is non-zero

- Largely sequential access pattern
- Approximation factor of $1 + p \ln n$ (close to optimal)
- Precomputation step
- Footprint still linear

2: Streaming Set Cover: EMEK-ROSÉN [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \text{ polylog}(n, m))$ bits of working memory

2: Streaming Set Cover: Emek-Rosén [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \text{ polylog}(n, m))$ bits of working memory

Emek-Rosén [4]

2: Streaming Set Cover: Emek-Rosén [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \, \text{polylog}(n, m))$ bits of working memory

Emek-Rosén [4]

1. Read set from stream

2: Streaming Set Cover: Emek-Rosén [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \text{ polylog}(n, m))$ bits of working memory

Emek-Rosén [4]

1. Read set from stream
2. Find the *maximal effective subset* of this set

2: Streaming Set Cover: Emek-Rosén [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \text{ polylog}(n, m))$ bits of working memory

Emek-Rosén [4]

1. Read set from stream
2. Find the *maximal effective subset* of this set
3. For each element in this subset:

2: Streaming Set Cover: Emek-Rosén [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \operatorname{polylog}(n, m))$ bits of working memory

Emek-Rosén [4]

1. Read set from stream
2. Find the *maximal effective subset* of this set
3. For each element in this subset:
   - Set the effectiveness to the *level* of the subset
   - Set the effectiveness identifier to that of the set

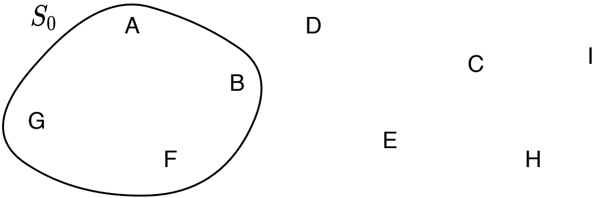
2: Streaming Set Cover: Emek-Rosén [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \text{ polylog}(n, m))$ bits of working memory

Emek-Rosén [4]

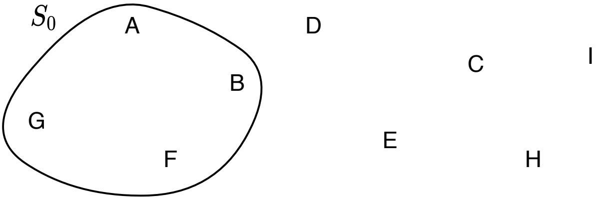
1. Read set from stream
2. Find the *maximal effective subset* of this set
3. For each element in this subset:
   - Set the effectiveness to the *level* of the subset
   - Set the effectiveness identifier to that of the set
4. Repeat until stream terminates

2: Streaming Set Cover: Emek-Rosén [4]



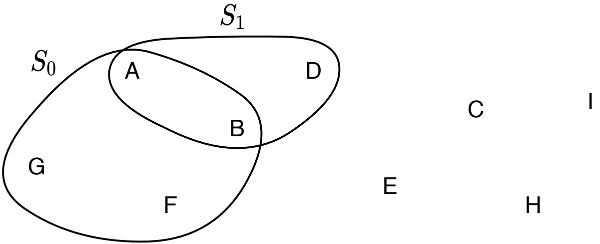
x	A	B	C	D	E	F	G	H	I
eff(x)	-1	-1	-1	-1	-1	-1	-1	-1	-1
eid(x)	-	-	-	-	-	-	-	-	-

2: Streaming Set Cover: Emek-Rosén [4]



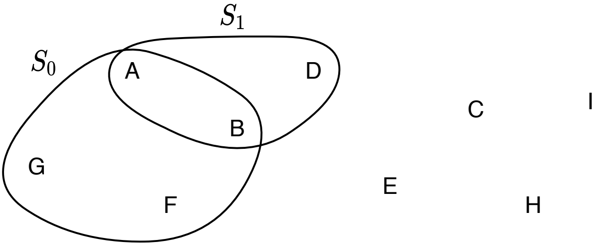
x	A	B	C	D	E	F	G	H	I
eff(x)	**2**	**2**	-1	-1	-1	**2**	**2**	-1	-1
eid(x)	**0**	**0**	-	-	-	**0**	**0**	-	-

2: Streaming Set Cover: EMEK-ROSÉN [4]



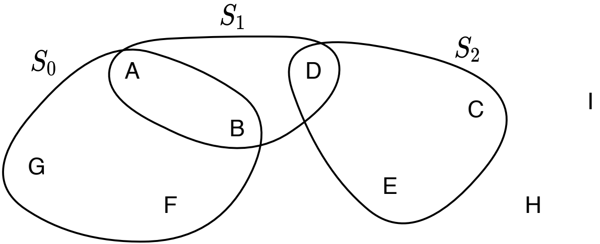
x	A	B	C	D	E	F	G	H	I
eff(x)	2	2	-1	-1	-1	2	2	-1	-1
eid(x)	0	0	-	-	-	0	0	-	-

2: Streaming Set Cover: EMEK-ROSÉN [4]



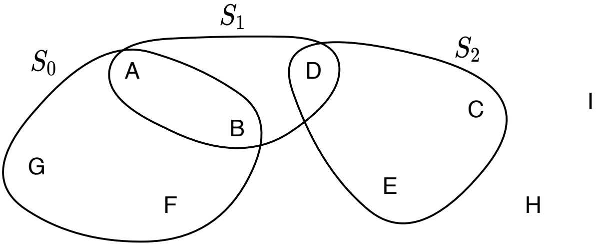
x	A	B	C	D	E	F	G	H	I
eff(x)	2	2	-1	**0**	-1	2	2	-1	-1
eid(x)	0	0	-	**1**	-	0	0	-	-

2: Streaming Set Cover: Emek-Rosén [4]



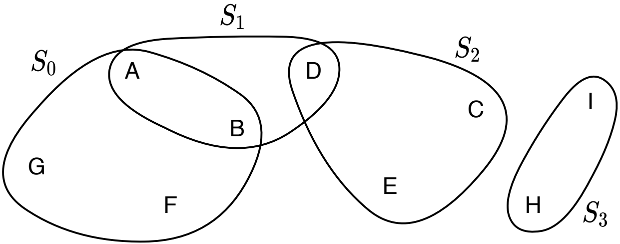
x	A	B	C	D	E	F	G	H	I
eff(x)	2	2	-1	0	-1	2	2	-1	-1
eid(x)	0	0	-	1	-	0	0	-	-

2: Streaming Set Cover: EMEK-ROSÉN [4]



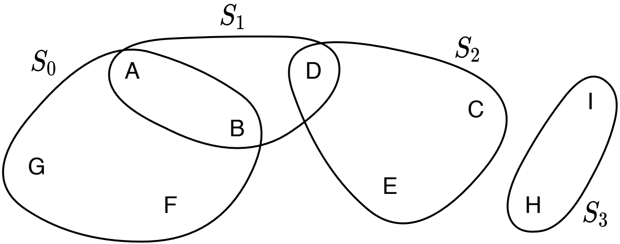
x	A	B	C	D	E	F	G	H	I
eff(x)	2	2	**2**	**2**	**2**	2	2	-1	-1
eid(x)	0	0	**2**	**2**	**2**	0	0	-	-

2: Streaming Set Cover: Emek-Rosén [4]



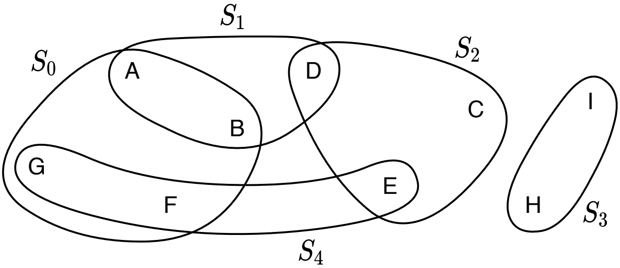
x	A	B	C	D	E	F	G	H	I
eff(x)	2	2	2	2	2	2	2	-1	-1
eid(x)	0	0	2	2	2	0	0	-	-

2: Streaming Set Cover: EMEK-ROSÉN [4]



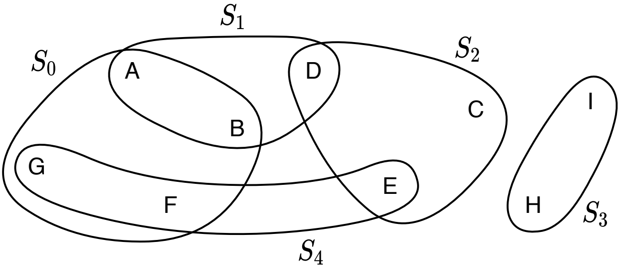
x	A	B	C	D	E	F	G	H	I
eff(x)	2	2	2	2	2	2	2	**1**	**1**
eid(x)	0	0	2	2	2	0	0	**3**	**3**

2: Streaming Set Cover: EMEK-ROSÉN [4]



x	A	B	C	D	E	F	G	H	I
eff(x)	2	2	2	2	2	2	2	1	1
eid(x)	0	0	2	2	2	0	0	3	3

2: Streaming Set Cover: Emek-Rosén [4]



x	A	B	C	D	E	F	G	H	I
eff(x)	2	2	2	2	2	2	2	1	1
eid(x)	0	0	2	2	2	0	0	3	3

2: Streaming Set Cover: Emek-Rosén [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \operatorname{polylog}(n, m))$ bits of working memory

Emek-Rosén [4]

1. Read set from stream
2. Find the *maximal effective subset* of this set
3. For each element in this subset:
   - Set the effectiveness to the *level* of the subset
   - Set the effectiveness identifier to that of the set
4. Repeat until stream terminates

2: Streaming Set Cover: EMEK-ROSÉN [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \text{ polylog}(n, m))$ bits of working memory

EMEK-ROSÉN [4]

1. Read set from stream
2. Find the *maximal effective subset* of this set
3. For each element in this subset:
   - Set the effectiveness to the *level* of the subset
   - Set the effectiveness identifier to that of the set
4. Repeat until stream terminates

- One-pass streaming algorithm

2: Streaming Set Cover: Emek-Rosén [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \text{ polylog}(n, m))$ bits of working memory

Emek-Rosén [4]

1. Read set from stream
2. Find the *maximal effective subset* of this set
3. For each element in this subset:
   - Set the effectiveness to the *level* of the subset
   - Set the effectiveness identifier to that of the set
4. Repeat until stream terminates

- One-pass streaming algorithm
- Runs in $\mathcal{O}(M)$ time

2: Streaming Set Cover: Emek-Rosén [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \text{ polylog}(n, m))$ bits of working memory

Emek-Rosén [4]

1. Read set from stream
2. Find the *maximal effective subset* of this set
3. For each element in this subset:
   - Set the effectiveness to the *level* of the subset
   - Set the effectiveness identifier to that of the set
4. Repeat until stream terminates

- One-pass streaming algorithm
- Runs in $\mathcal{O}(M)$ time
- Uses only $\widetilde{\mathcal{O}}(n)$ space

2: Streaming Set Cover: Emek-Rosén [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \text{ polylog}(n, m))$ bits of working memory

Emek-Rosén [4]

1. Read set from stream
2. Find the *maximal effective subset* of this set
3. For each element in this subset:
   - Set the effectiveness to the *level* of the subset
   - Set the effectiveness identifier to that of the set
4. Repeat until stream terminates

- One-pass streaming algorithm
- Runs in $\mathcal{O}(M)$ time
- Uses only $\widetilde{\mathcal{O}}(n)$ space
- Approximation factor of $\mathcal{O}(\sqrt{\Delta})$ (this is optimal [4])

2: Streaming Set Cover: Emek-Rosén [4]

Aim: use $\widetilde{\mathcal{O}}(n) := \mathcal{O}(n \text{ polylog}(n, m))$ bits of working memory

Emek-Rosén [4]

1. Read set from stream
2. Find the *maximal effective subset* of this set
3. For each element in this subset:
   - Set the effectiveness to the *level* of the subset
   - Set the effectiveness identifier to that of the set
4. Repeat until stream terminates

- One-pass streaming algorithm
- Runs in $\mathcal{O}(M)$ time
- Uses only $\widetilde{\mathcal{O}}(n)$ space
- Approximation factor of $\mathcal{O}(\sqrt{\Delta})$ (this is optimal [4])

How does Emek-Rosén perform in practice?

Emek-Rosén in Practice

Emek-Rosén in Practice

File name	DFG (RAM)	Emek-Rosén
accidents.dat	181	213
kosarak.dat	17 741	18 618
orkut-cmty.dat	149 244	158 439
webdocs.dat	406 338	413 819
twitter.dat	9 246 029	9 955 112
friendster.dat	-	13 310 036

Table: Cover size

Emek-Rosén in Practice

File name	DFG (RAM)	Emek-Rosén
accidents.dat	1.43	0.72
kosarak.dat	1.03	0.79
orkut-cmty.dat	15.44	12.35
webdocs.dat	18.39	15.51
twitter.dat	213.48	158.35
friendster.dat	-	367.52

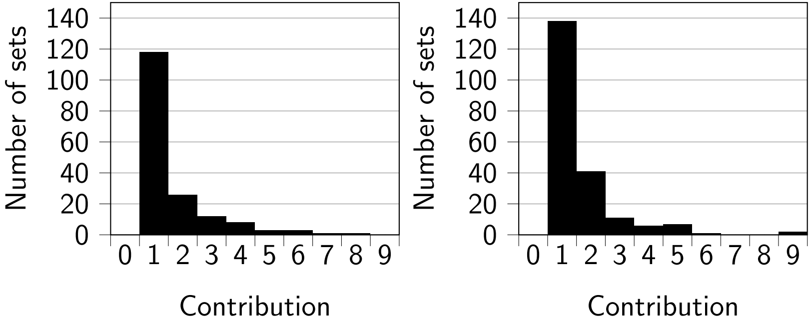
Table: Time (s)

Emek-Rosén in Practice

File name	DFG (RAM)	Emek-Rosén
accidents.dat	66.65	0.91
kosarak.dat	75.45	2.37
orkut-cmty.dat	1094.59	21.67
webdocs.dat	1401.55	56.04
twitter.dat	8044.29	797.70
friendster.dat	-	1183.56

Table: Peak RAM usage (MB)

Emek-Rosén in Practice



(a) DFG (b) Emek-Rosén

Generalising Emek-Rosén to Multiple Passes

Multi-Pass Emek-Rosén

Generalising Emek-Rosén to Multiple Passes

Multi-Pass Emek-Rosén

1. Do an Emek-Rosén pass

Generalising Emek-Rosén to Multiple Passes

Multi-Pass Emek-Rosén

1. Do an Emek-Rosén pass
2. Restrict the universe based on effectiveness

Generalising Emek-Rosén to Multiple Passes

Multi-Pass Emek-Rosén

1. Do an Emek-Rosén pass
2. Restrict the universe based on effectiveness
3. Repeat for p passes

Generalising Emek-Rosén to Multiple Passes

Multi-Pass Emek-Rosén

1. Do an Emek-Rosén pass
2. Restrict the universe based on effectiveness
3. Repeat for p passes

Generalising EMEK-ROSÉN to Multiple Passes

MULTI-PASS EMEK-ROSÉN

1. Do an EMEK-ROSÉN pass
2. Restrict the universe based on effectiveness
3. Repeat for p passes

Approximation factor of $\mathcal{O}(\Delta^{\frac{1}{p+1}})$

Multi-Pass Emek-Rosén in Practice

Multi-Pass Emek-Rosén in Practice

Progressive Greedy [1]

Multi-Pass Emek-Rosén in Practice

Progressive Greedy [1]

1. Add all sets whose contribution is above a threshold

Multi-Pass Emek-Rosén in Practice

Progressive Greedy [1]

1. Add all sets whose contribution is above a threshold
2. Reduce threshold

Multi-Pass Emek-Rosén in Practice

Progressive Greedy [1]

1. Add all sets whose contribution is above a threshold
2. Reduce threshold
3. Repeat for p passes

Multi-Pass Emek-Rosén in Practice

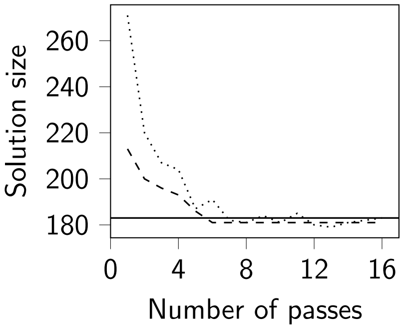
Progressive Greedy [1]

1. Add all sets whose contribution is above a threshold
2. Reduce threshold
3. Repeat for p passes

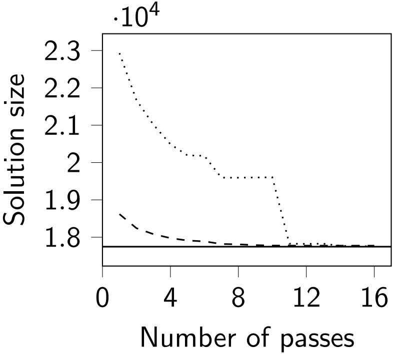
Approximation factor of $\mathcal{O}(\Delta^{\frac{1}{p+1}})$

Multi-Pass Emek-Rosén in Practice

Results (1/3)



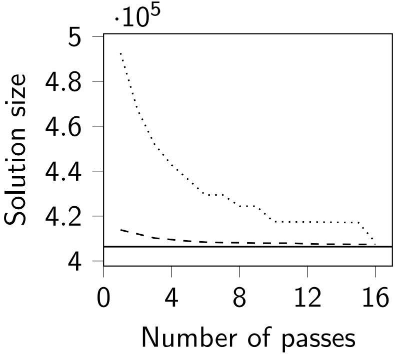
(a) accidents.dat



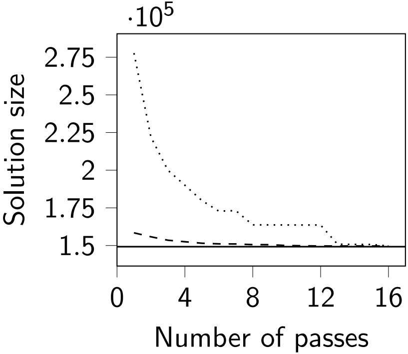
(b) kosarak.dat

Multi-Pass Emek-Rosén in Practice

Results (2/3)



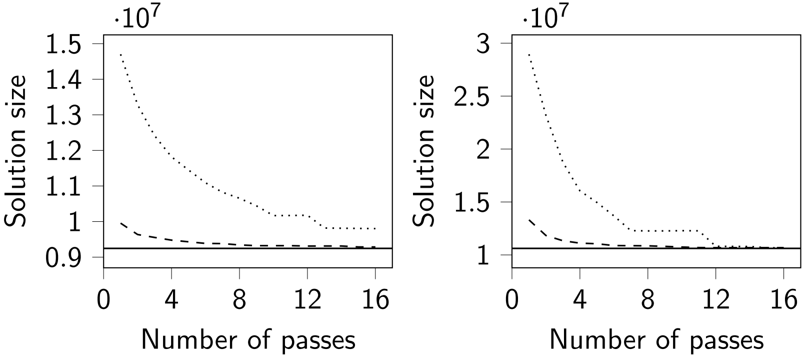
(a) webdocs.dat



(b) orkut-cmty.dat

Multi-Pass Emek-Rosén in Practice

Results (3/3)



(a) twitter.dat

(b) friendster.dat

Summary

- EMEK-ROSÉN works well in practice

Summary

- Emek-Rosén works well in practice
  - Slightly larger covers
  - Much smaller memory footprint
  - Faster

Summary

- Emek-Rosén works well in practice
  - Slightly larger covers
  - Much smaller memory footprint
  - Faster
- Solution quality can be improved with multiple passes

References I

Amit Chakrabarti and Anthony Wirth.
Incidence geometries and the pass complexity of semi-streaming set cover.
In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1365–1373. SIAM, 2016.

Graham Cormode, Howard Karloff, and Anthony Wirth.
Set cover algorithms for very large datasets.
In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 479–488, 2010.

Irit Dinur and David Steurer.
Analytical approach to parallel repetition.
In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 624–633, 2014.

References II

Yuval Emek and Adi Rosén.
Semi-streaming set cover - (extended abstract).
In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 453–464. Springer, 2014.

Uriel Feige.
A threshold of ln n for approximating set cover.
Journal of the ACM (JACM), 45(4):634–652, 1998.