

Optimal Splitters for Temporal and Multi-version Databases

Wangchao Le¹ Feifei Li¹ Yufei Tao^{2,3} Robert Christensen¹



¹University of Utah



香港中文大學
THE CHINESE UNIVERSITY OF HONG KONG

²Chinese University of Hong Kong

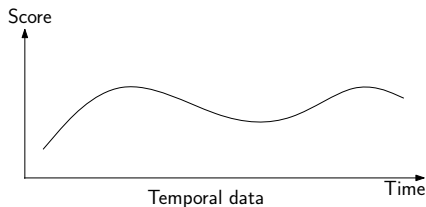


³Korea Adv. Inst. Sci & Tech

July 11, 2013

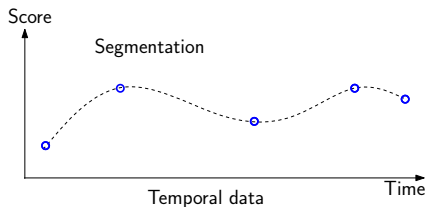
Temporal and Multi-version Data

- Temporal and multi-version data are important in:
 - financial market
 - scientific application
 - data warehousing



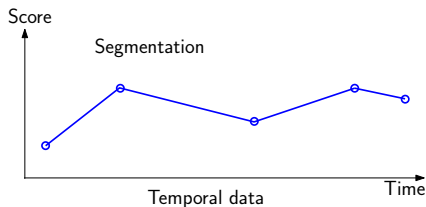
Temporal and Multi-version Data

- Temporal and multi-version data are important in:
 - financial market
 - scientific application
 - data warehousing



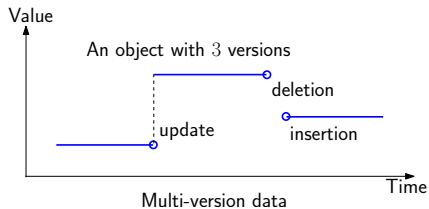
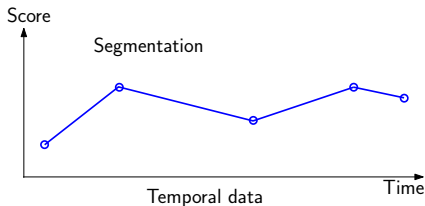
Temporal and Multi-version Data

- Temporal and multi-version data are important in:
 - financial market
 - scientific application
 - data warehousing



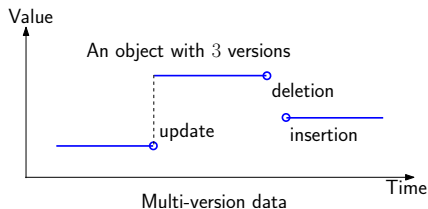
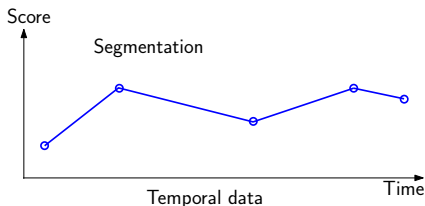
Temporal and Multi-version Data

- Temporal and multi-version data are important in:
 - financial market
 - scientific application
 - data warehousing



Temporal and Multi-version Data

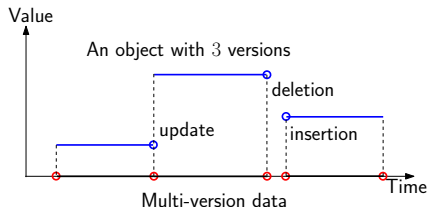
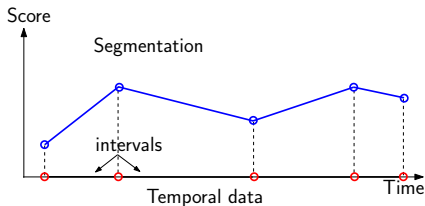
- Temporal and multi-version data are important in:
 - financial market
 - scientific application
 - data warehousing



- User applications:
 - collect and query data in a **long-running history**
 - an object $o \rightarrow$ disjoint temporal intervals

Temporal and Multi-version Data

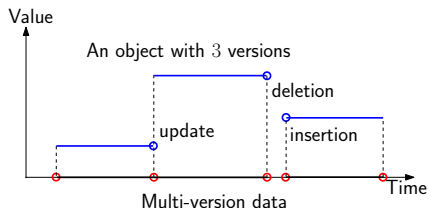
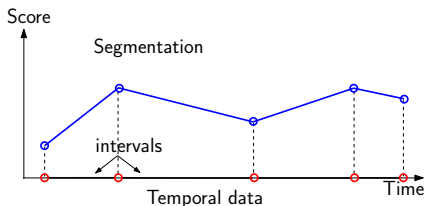
- Temporal and multi-version data are important in:
 - financial market
 - scientific application
 - data warehousing



- User applications:
 - collect and query data in a **long-running history**
 - an object $o \rightarrow$ disjoint temporal intervals

Temporal and Multi-version Data

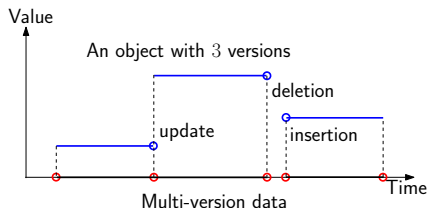
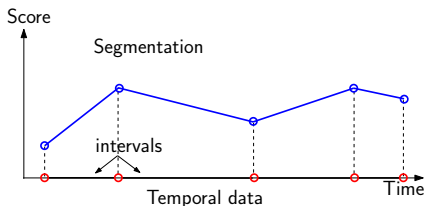
- Temporal and multi-version data are important in:
 - financial market
 - scientific application
 - data warehousing



- User applications:
 - collect and query data in a **long-running history**
 - an object $o \rightarrow$ disjoint temporal intervals
 - scale out by storing data in a **distributed** and **parallel** framework

Temporal and Multi-version Data

- Temporal and multi-version data are important in:
 - financial market
 - scientific application
 - data warehousing



- User applications:
 - collect and query data in a **long-running history**
 - an object $o \rightarrow$ disjoint temporal intervals
 - scale out by storing data in a **distributed** and **parallel** framework

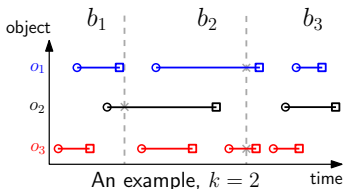
Have to deal with data partitioning

Problem Formulation

- Partition interval data into buckets based on time
 - process queries w.r.t a given time with selected node(s)/core(s)

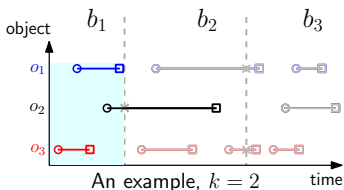
Problem Formulation

- Partition interval data into buckets based on time
 - process queries w.r.t a given time with selected node(s)/core(s)
- A size- k partition P over a set of intervals \mathcal{I} , denoted as $P(\mathcal{I}, k)$:
 - has k distinct vertical splitters and $k + 1$ buckets



Problem Formulation

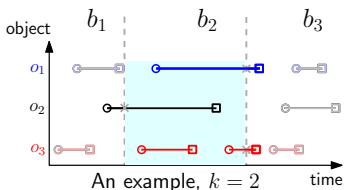
- Partition interval data into buckets based on time
 - process queries w.r.t a given time with selected node(s)/core(s)
- A size- k partition P over a set of intervals \mathcal{I} , denoted as $P(\mathcal{I}, k)$:
 - ④ has k distinct vertical splitters and $k + 1$ buckets



- ② an interval $[s, e] \in b_i$ if it intersects b_i (b_i is a set of intervals)

Problem Formulation

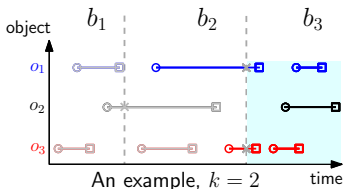
- Partition interval data into buckets based on time
 - process queries w.r.t a given time with selected node(s)/core(s)
- A size- k partition P over a set of intervals \mathcal{I} , denoted as $P(\mathcal{I}, k)$:
 - ④ has k distinct vertical splitters and $k + 1$ buckets



- ② an interval $[s, e] \in b_i$ if it intersects b_i (b_i is a set of intervals)

Problem Formulation

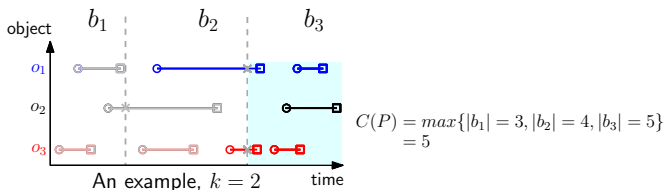
- Partition interval data into buckets based on time
 - process queries w.r.t a given time with selected node(s)/core(s)
- A size- k partition P over a set of intervals \mathcal{I} , denoted as $P(\mathcal{I}, k)$:
 - ④ has k distinct vertical splitters and $k + 1$ buckets



- ② an interval $[s, e] \in b_i$ if it intersects b_i (b_i is a set of intervals)
- ③ **Cost** of a partition: $c(P) = \max\{|b_1| \dots |b_{k+1}|\}$

Problem Formulation

- Partition interval data into buckets based on time
 - process queries w.r.t a given time with selected node(s)/core(s)
- A size- k partition P over a set of intervals \mathcal{I} , denoted as $P(\mathcal{I}, k)$:
 - 1 has k distinct vertical splitters and $k + 1$ buckets



- 2 an interval $[s, e] \in b_i$ if it intersects b_i (b_i is a set of intervals)
- 3 **Cost** of a partition: $c(P) = \max\{|b_1| \dots |b_{k+1}|\}$

Problem Formulation

- **Load-balancing** is important in a distributed setting

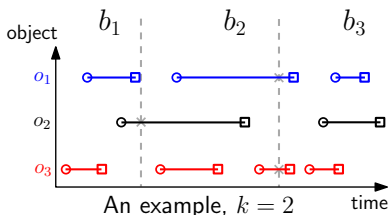
Problem Formulation

- **Load-balancing** is important in a distributed setting
- Objective: minimize the maximum load on a single node

Definition

An **optimal partition** of size- k is a partition $P^*(\mathcal{I}, k)$ with the smallest cost, *i.e.*

$$P^*(\mathcal{I}, k) = \operatorname{argmin}(c(P))$$



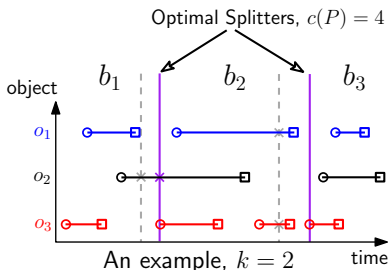
Problem Formulation

- **Load-balancing** is important in a distributed setting
- Objective: minimize the maximum load on a single node

Definition

An **optimal partition** of size- k is a partition $P^*(\mathcal{I}, k)$ with the smallest cost, *i.e.*

$$P^*(\mathcal{I}, k) = \operatorname{argmin}(c(P))$$



Problem Formulation

- **Load-balancing** is important in a distributed setting
- Objective: minimize the maximum load on a single node

Definition

An **optimal partition** of size- k is a partition $P^*(\mathcal{I}, k)$ with the smallest cost, *i.e.*

$$P^*(\mathcal{I}, k) = \operatorname{argmin}(c(P))$$

- In this talk, our objective:
Find P^* and $c(P^*)$ for \mathcal{I} and a fixed budget k

- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

Strategy to Place Splitters

- Where to place splitters?

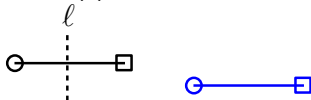
Strategy to Place Splitters

- Where to place splitters?
 - let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.

Strategy to Place Splitters

- Where to place splitters?

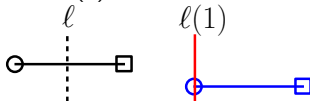
- let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
- for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$



Strategy to Place Splitters

- Where to place splitters?

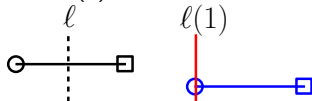
- let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
- for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$



Strategy to Place Splitters

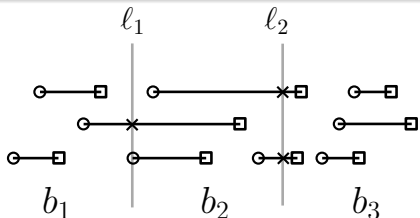
- Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
- for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$



Observation

For any partition P with distinct splitters $l_1 < \dots < l_k$. Let l_i be the **largest splitter that does not in \mathbf{S}** . Define P' from P by replacing l_i with $l_i(1)$. Then, $c(P') \leq c(P)$.

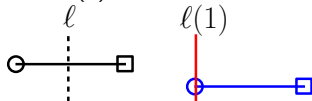


$$c(P) = 5$$

Strategy to Place Splitters

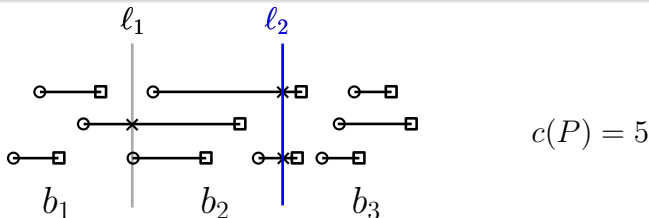
- Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
- for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$



Observation

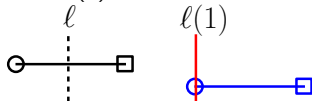
For any partition P with distinct splitters $l_1 < \dots < l_k$. Let l_i be the largest splitter that does not in \mathbf{S} . Define P' from P by replacing l_i with $l_i(1)$. Then, $c(P') \leq c(P)$.



Strategy to Place Splitters

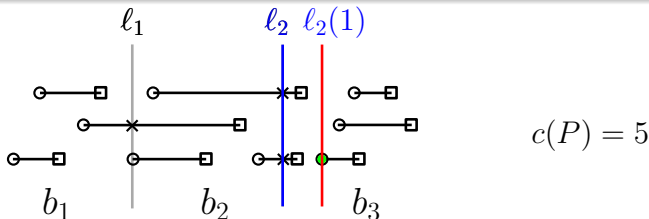
- Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
- for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$



Observation

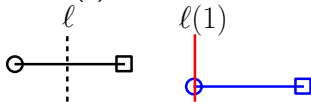
For any partition P with distinct splitters $l_1 < \dots < l_k$. Let l_i be the largest splitter that does not in \mathbf{S} . Define P' from P by replacing l_i with $l_i(1)$. Then, $c(P') \leq c(P)$.



Strategy to Place Splitters

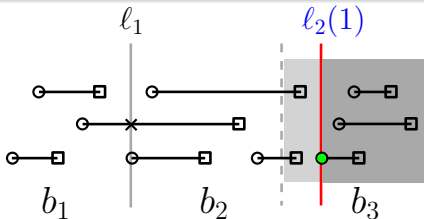
- Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
- for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$



Observation

For any partition P with distinct splitters $\ell_1 < \dots < \ell_k$. Let ℓ_i be the largest splitter that does not in \mathbf{S} . Define P' from P by replacing ℓ_i with $\ell_i(1)$. Then, $c(P') \leq c(P)$.



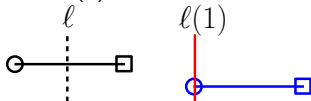
- no effect on b_2
- shrink b_3

$$c(P') = 4 \leq c(P) = 5$$

Strategy to Place Splitters

- Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
- for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$



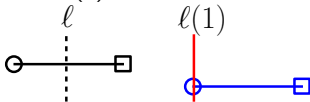
Observation

For any partition P with distinct splitters $\ell_1 < \dots < \ell_k$. Let ℓ_i be the largest splitter that does not in \mathbf{S} . Define P' from P by replacing ℓ_i with $\ell_i(1)$. Then, $c(P') \leq c(P)$.

Strategy to Place Splitters

- Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
- for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$



Observation

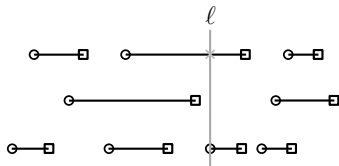
For any partition P with distinct splitters $\ell_1 < \dots < \ell_k$. Let ℓ_i be the largest splitter that does not in \mathbf{S} . Define P' from P by replacing ℓ_i with $\ell_i(1)$. Then, $c(P') \leq c(P)$.

Should always try to split on **S** !

- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

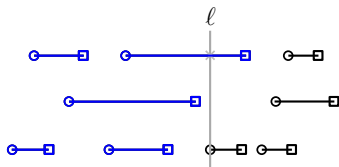
Dynamic Programming Approach

- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



Dynamic Programming Approach

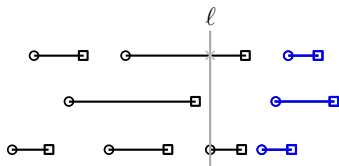
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in \mathcal{I} \mid s_i < \ell\}$

Dynamic Programming Approach

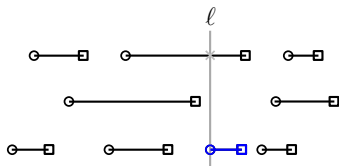
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in \mathcal{I} \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in \mathcal{I} \mid s_i > \ell\}$

Dynamic Programming Approach

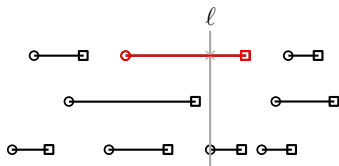
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$

Dynamic Programming Approach

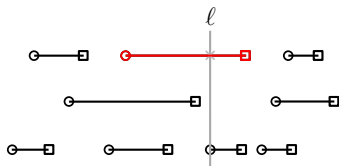
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

Dynamic Programming Approach

- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array

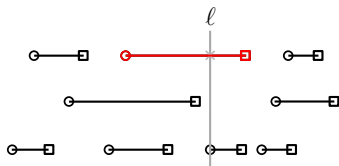


- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

Dynamic Programming Approach

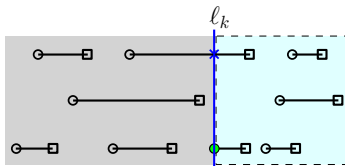
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in \mathcal{I} \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in \mathcal{I} \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in \mathcal{I} \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in \mathcal{I} \mid s_i < \ell < e_i\}$

- Dynamic programming

$c(P^*(\mathcal{I}, k))$



LastBucket = $|\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$

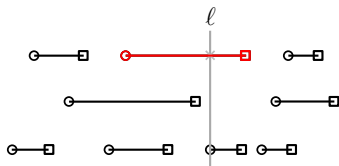


A sub-problem: $c(P^*(\mathcal{I}^-(\ell_k), k - 1))$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(\mathcal{I})$

Dynamic Programming Approach

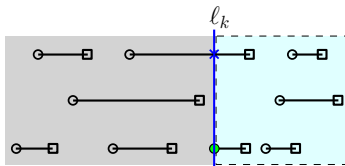
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\}$$



$$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$$

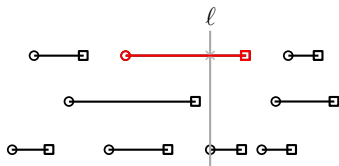


$$\text{A sub-problem: } c(P^*(\mathcal{I}^-(\ell_k), k-1))$$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(I)$

Dynamic Programming Approach

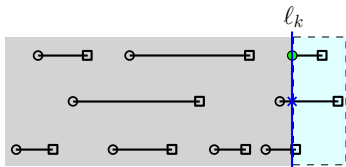
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\}$$



$$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$$

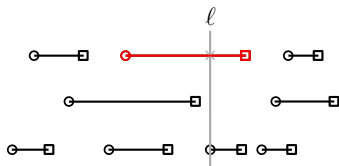


$$\text{A sub-problem: } c(P^*(\mathcal{I}^-(\ell_k), k-1))$$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(I)$

Dynamic Programming Approach

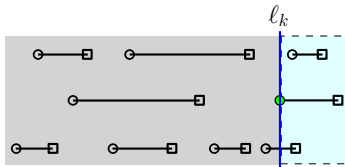
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\}$$



$$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$$

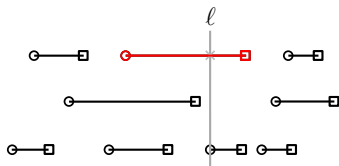


$$\text{A sub-problem: } c(P^*(\mathcal{I}^-(\ell_k), k-1))$$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(I)$

Dynamic Programming Approach

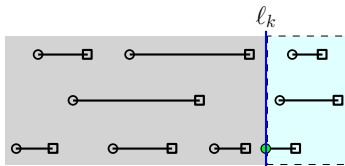
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\}$$



$$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$$

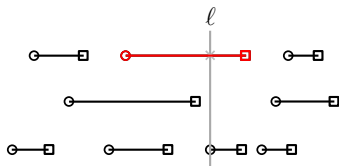


$$\text{A sub-problem: } c(P^*(\mathcal{I}^-(\ell_k), k-1))$$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(I)$

Dynamic Programming Approach

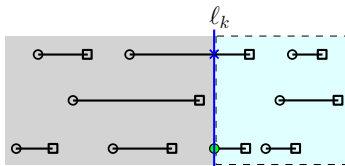
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\}$$



$$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$$

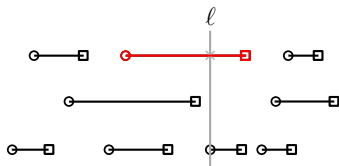


$$\text{A sub-problem: } c(P^*(\mathcal{I}^-(\ell_k), k-1))$$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(I)$

Dynamic Programming Approach

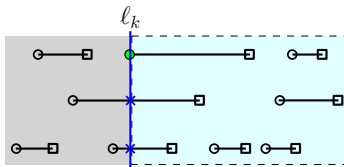
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\}$$



$$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$$

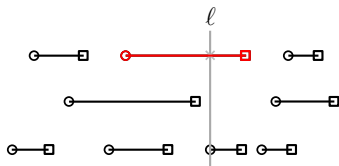


$$\text{A sub-problem: } c(P^*(\mathcal{I}^-(\ell_k), k-1))$$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(I)$

Dynamic Programming Approach

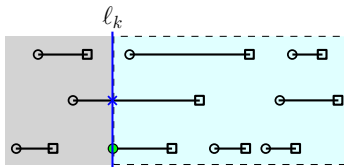
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\}$$



$$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$$

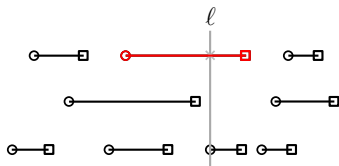


$$\text{A sub-problem: } c(P^*(\mathcal{I}^-(\ell_k), k-1))$$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(I)$

Dynamic Programming Approach

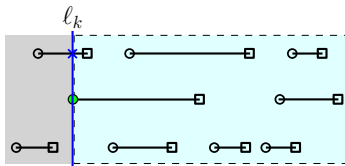
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\}$$



$$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$$

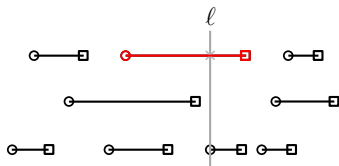


$$\text{A sub-problem: } c(P^*(\mathcal{I}^-(\ell_k), k-1))$$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(I)$

Dynamic Programming Approach

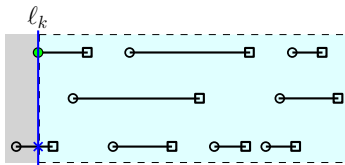
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\}$$



$$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$$

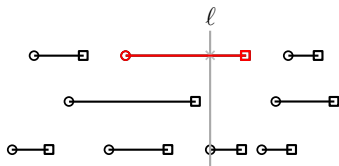


$$\text{A sub-problem: } c(P^*(\mathcal{I}^-(\ell_k), k-1))$$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(I)$

Dynamic Programming Approach

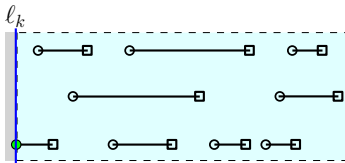
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\}$$



$$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$$

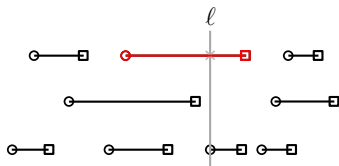


$$\text{A sub-problem: } c(P^*(\mathcal{I}^-(\ell_k), k-1))$$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(I)$

Dynamic Programming Approach

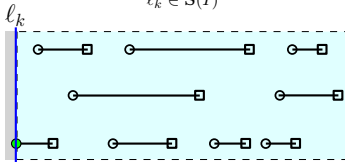
- Given a splitter ℓ and a set of intervals \mathcal{I} stored in an array



- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell\}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I \mid s_i > \ell\}$
- $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I \mid s_i = \ell\}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \min_{\ell_k \in \mathbf{S}(I)} \{ \max \{ c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket}) \} \}$$



$$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$$



$$\text{A sub-problem: } c(P^*(\mathcal{I}^-(\ell_k), k-1))$$

- How many ways to place ℓ_k ? $\ell_k \in \mathbf{S}(I)$

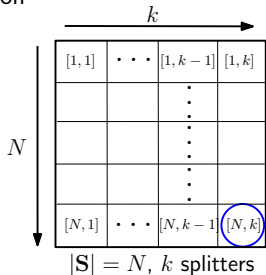
- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

Cost Analysis

- A common sub-problem may appear more than one time

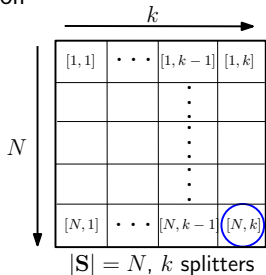
Cost Analysis

- A common sub-problem may appear more than one time
 - Memoization



Cost Analysis

- A common sub-problem may appear more than one time
 - Memoization

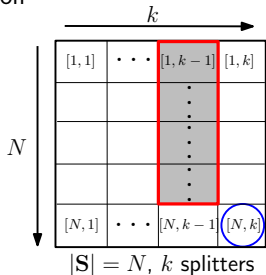


- Cost of the DP approach

$$c(P^*(\mathcal{I}, k)) = \min_{\ell_k \in S} \{ \max \{ c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket}) \} \}$$

Cost Analysis

- A common sub-problem may appear more than one time
 - Memoization



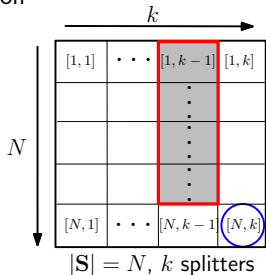
- Cost of the DP approach

$$c(P^*(\mathcal{I}, k)) = \min_{\ell_k \in S} \{ \max \{ c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket}) \} \}$$

- to fill in Cell $[i, j]$, need to check $i-1$ preceding rows

Cost Analysis

- A common sub-problem may appear more than one time
 - Memoization



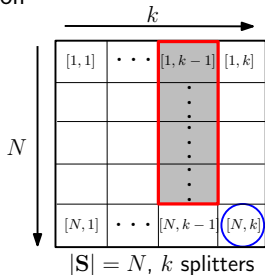
- Cost of the DP approach

$$c(P^*(\mathcal{I}, k)) = \min_{\ell_k \in \mathbf{S}} \{ \max \{ c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket}) \} \}$$

- 1 to fill in $\text{Cell}[i, j]$, need to check $i-1$ preceding rows
- 2 **O(1)** cost to obtain **LastBucket** ($|\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$)

Cost Analysis

- A common sub-problem may appear more than one time
 - Memoization



- Cost of the DP approach

$$c(P^*(\mathcal{I}, k)) = \min_{\ell_k \in \mathbf{S}} \{ \max \{ c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket}) \} \}$$

- 1 to fill in Cell $[i, j]$, need to check $i-1$ preceding rows
- 2 **O(1)** cost to obtain LastBucket ($|\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$)
- 3 $O(kN^2)$ for DP

- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

Cost- t Splitter Problem

A decision version of our problem:

Definition (**Cost- t splitters problem**)

Determine whether there is a size- k partition P with $c(P) \leq t$

Cost- t Splitter Problem

A decision version of our problem:

Definition (**Cost- t splitters problem**)

Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, t is *feasible*
 - **Output:** $\bar{t} \in [1, t]$ s.t. $\exists P \in \mathcal{P}(I, k), c(P) = \bar{t}$
- 2 otherwise, t is *infeasible*
 - **Output:** $\bar{t} = 0$

Cost- t Splitter Problem

A decision version of our problem:

Definition (**Cost- t splitters problem**)

Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, t is *feasible*
 - **Output:** $\bar{t} \in [1, t]$ s.t. $\exists P \in \mathcal{P}(I, k), c(P) = \bar{t}$
- 2 otherwise, t is *infeasible*
 - **Output:** $\bar{t} = 0$

Lemma

If t is *infeasible*, then any $t' < t$ is also *infeasible*

Cost- t Splitter Problem

A decision version of our problem:

Definition (**Cost- t splitters problem**)

Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, t is *feasible*
 - **Output:** $\bar{t} \in [1, t]$ s.t. $\exists P \in \mathcal{P}(I, k), c(P) = \bar{t}$
- 2 otherwise, t is *infeasible*
 - **Output:** $\bar{t} = 0$

Lemma

If t is *infeasible*, then any $t' < t$ is also *infeasible*

Sketch of the Algorithm:

Cost- t Splitter Problem

A decision version of our problem:

Definition (**Cost- t splitters problem**)

Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, t is *feasible*
 - **Output:** $\bar{t} \in [1, t]$ s.t. $\exists P \in \mathcal{P}(I, k), c(P) = \bar{t}$
- 2 otherwise, t is *infeasible*
 - **Output:** $\bar{t} = 0$

Lemma

If t is *infeasible*, then any $t' < t$ is also *infeasible*

Sketch of the Algorithm:

- 1 The optimal cost t^* is in the range of $\mathbf{R} = [1, N]$

Cost- t Splitter Problem

A decision version of our problem:

Definition (**Cost- t splitters problem**)

Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, t is *feasible*
 - **Output:** $\bar{t} \in [1, t]$ s.t. $\exists P \in \mathcal{P}(I, k), c(P) = \bar{t}$
- 2 otherwise, t is *infeasible*
 - **Output:** $\bar{t} = 0$

Lemma

If t is *infeasible*, then any $t' < t$ is also *infeasible*

Sketch of the Algorithm:

- 1 The optimal cost t^* is in the range of $\mathbf{R} = [1, N]$
- 2 Binary search on \mathbf{R}
- 3 Solve $O(\log N)$ instances of Cost- t splitters problem

Cost- t Splitter Problem

A decision version of our problem:

Definition (**Cost- t splitters problem**)

Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, t is *feasible*
 - **Output:** $\bar{t} \in [1, t]$ s.t. $\exists P \in \mathcal{P}(I, k), c(P) = \bar{t}$
- 2 otherwise, t is *infeasible*
 - **Output:** $\bar{t} = 0$

Lemma

If t is *infeasible*, then any $t' < t$ is also *infeasible*

Sketch of the Algorithm:

- 1 The optimal cost t^* is in the range of $\mathbf{R} = [1, N]$
- 2 Binary search on \mathbf{R}
- 3 Solve $O(\log N)$ instances of Cost- t splitters problem
- 4 Report t^* , when t^* is *feasible* but $t^* - 1$ is *infeasible*

Cost- t Splitter Problem

A decision version of our problem:

Definition (**Cost- t splitters problem**)

Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, t is *feasible*
 - **Output:** $\bar{t} \in [1, t]$ s.t. $\exists P \in \mathcal{P}(I, k), c(P) = \bar{t}$
- 2 otherwise, t is *infeasible*
 - **Output:** $\bar{t} = 0$

Lemma

If t is *infeasible*, then any $t' < t$ is also *infeasible*

Sketch of the Algorithm:

- 1 The optimal cost t^* is in the range of $\mathbf{R} = [1, N]$
- 2 Binary search on \mathbf{R}
- 3 **Solve** $O(\log N)$ instances of **Cost- t splitters problem**
- 4 Report t^* , when t^* is *feasible* but $t^* - 1$ is *infeasible*

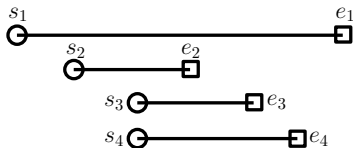
- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$
 - by non-descending order of s_i 's
 - break ties by non-descending order of e_i 's

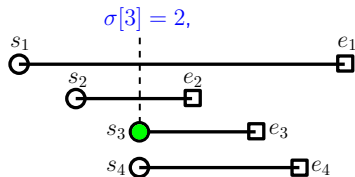
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$
 - by non-descending order of s_i 's
 - break ties by non-descending order of e_i 's
- The stabbing-count array for \mathcal{I}
 - $\forall s_i \in \mathcal{I}$, maintain two counts σ, δ
 - ▷ $\sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting s_i
 - ▷ $\delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than i



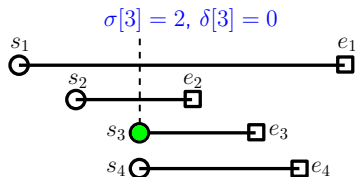
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$
 - by non-descending order of s_i 's
 - break ties by non-descending order of e_i 's
- The stabbing-count array for \mathcal{I}
 - $\forall s_i \in \mathcal{I}$, maintain two counts σ, δ
 - ▷ $\sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting s_i
 - ▷ $\delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than i



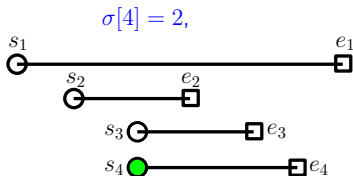
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$
 - by non-descending order of s_i 's
 - break ties by non-descending order of e_i 's
- The stabbing-count array for \mathcal{I}
 - $\forall s_i \in \mathcal{I}$, maintain two counts σ, δ
 - ▷ $\sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting s_i
 - ▷ $\delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than i



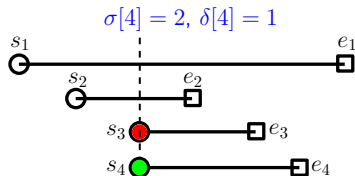
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$
 - by non-descending order of s_i 's
 - break ties by non-descending order of e_i 's
- The stabbing-count array for \mathcal{I}
 - $\forall s_i \in \mathcal{I}$, maintain two counts σ, δ
 - ▷ $\sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting s_i
 - ▷ $\delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than i



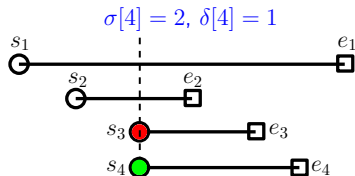
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$
 - by non-descending order of s_i 's
 - break ties by non-descending order of e_i 's
- The stabbing-count array for \mathcal{I}
 - $\forall s_i \in \mathcal{I}$, maintain two counts σ, δ
 - ▷ $\sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting s_i
 - ▷ $\delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than i



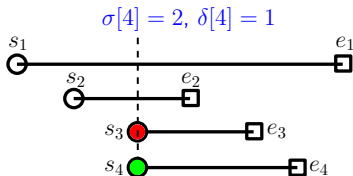
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$ [$O(N \log N)$ time]
 - by non-descending order of s_i 's
 - break ties by non-descending order of e_i 's
- The stabbing-count array for \mathcal{I} [$O(N)$ time]
 - $\forall s_i \in \mathcal{I}$, maintain two counts σ, δ
 - $\triangleright \sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting s_i
 - $\triangleright \delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than i



Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$ [$O(N \log N)$ time]
 - by non-descending order of s_i 's
 - break ties by non-descending order of e_i 's
- The stabbing-count array for \mathcal{I} [$O(N)$ time]
 - $\forall s_i \in \mathcal{I}$, maintain two counts σ, δ
 - $\triangleright \sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting s_i
 - $\triangleright \delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than i



Lemma

The stabbing-count array can be built in $O(N \log N)$ time

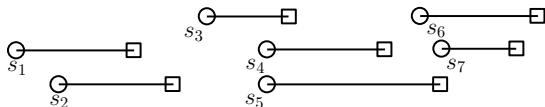
- t -jump method

- t -jump method
 - ① solves an instance of the Cost- t splitters problem
 - ② if **feasible**, output the feasible P and $c(P)$

- t -jump method
 - ① solves an instance of the Cost- t splitters problem
 - ② if **feasible**, output the feasible P and $c(P)$
 - ③ a greedy algorithm

- t -jump method

- 1 solves an instance of the Cost- t splitters problem
- 2 if **feasible**, output the feasible P and $c(P)$
- 3 a greedy algorithm

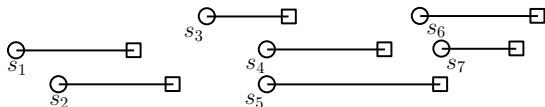


Intuition

t -jump method

- t -jump method

- 1 solves an instance of the Cost- t splitters problem
- 2 if **feasible**, output the feasible P and $c(P)$
- 3 a greedy algorithm

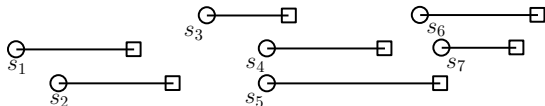


Intuition

- 1 place splitters in ascending order

- t -jump method

- 1 solves an instance of the Cost- t splitters problem
- 2 if **feasible**, output the feasible P and $c(P)$
- 3 a greedy algorithm

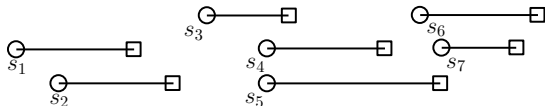


Intuition

- 1 place splitters in ascending order
- 2 ℓ_{i+1} is pushed as far as possible from ℓ_i , let each new b_i have size t

- t -jump method

- ① solves an instance of the Cost- t splitters problem
- ② if **feasible**, output the feasible P and $c(P)$
- ③ a greedy algorithm

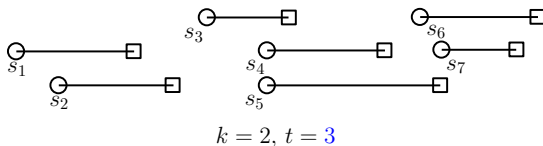


Intuition

- ① place splitters in ascending order
- ② l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- ③ if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method
 - 1 solves an instance of the Cost- t splitters problem
 - 2 if **feasible**, output the feasible P and $c(P)$
 - 3 a greedy algorithm

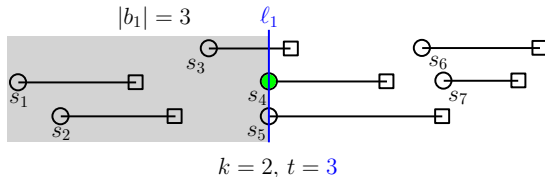


Intuition

- 1 place splitters in ascending order
- 2 ℓ_{i+1} is pushed as far as possible from ℓ_i , let each new b_i have size t
- 3 if not achievable, move ℓ_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method
 - 1 solves an instance of the Cost- t splitters problem
 - 2 if **feasible**, output the feasible P and $c(P)$
 - 3 a greedy algorithm



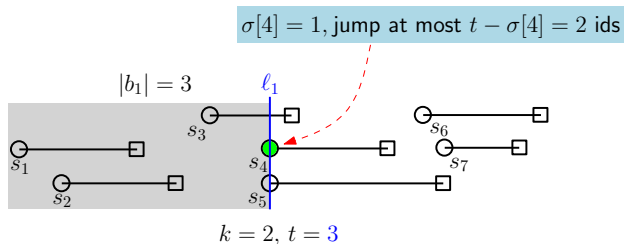
Intuition

- 1 place splitters in ascending order
- 2 l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- 3 if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method

- 1 solves an instance of the Cost- t splitters problem
- 2 if **feasible**, output the feasible P and $c(P)$
- 3 a greedy algorithm

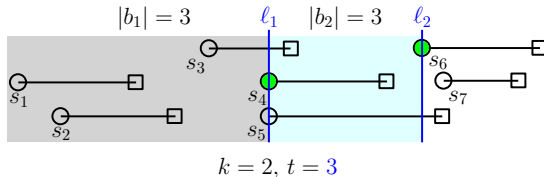


Intuition

- 1 place splitters in ascending order
- 2 l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- 3 if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method
 - 1 solves an instance of the Cost- t splitters problem
 - 2 if **feasible**, output the feasible P and $c(P)$
 - 3 a greedy algorithm



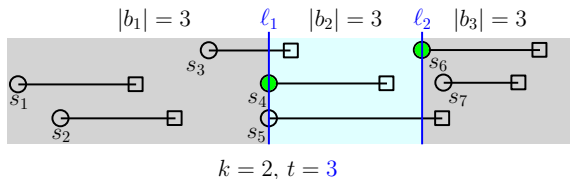
Intuition

- 1 place splitters in ascending order
- 2 l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- 3 if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method

- ① solves an instance of the Cost- t splitters problem
- ② if **feasible**, output the feasible P and $c(P)$
- ③ a greedy algorithm



$t=3$ is feasible

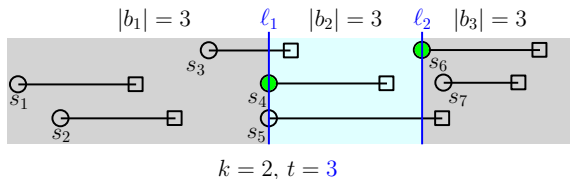
Intuition

- ① place splitters in ascending order
- ② l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- ③ if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method

- 1 solves an instance of the Cost- t splitters problem
- 2 if **feasible**, output the feasible P and $c(P)$
- 3 a greedy algorithm



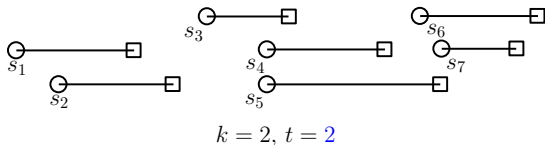
$t = 3$ is feasible
cost $O(k)$ time

Intuition

- 1 place splitters in ascending order
- 2 l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- 3 if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method
 - 1 solves an instance of the Cost- t splitters problem
 - 2 if **feasible**, output the feasible P and $c(P)$
 - 3 a greedy algorithm



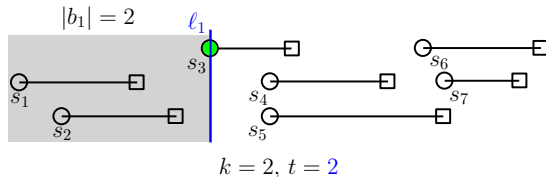
Intuition

- 1 place splitters in ascending order
- 2 ℓ_{i+1} is pushed as far as possible from ℓ_i , let each new b_i have size t
- 3 if not achievable, move ℓ_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method

- 1 solves an instance of the Cost- t splitters problem
- 2 if **feasible**, output the feasible P and $c(P)$
- 3 a greedy algorithm



Intuition

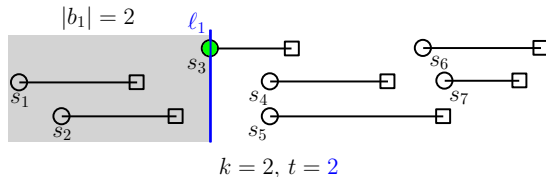
- 1 place splitters in ascending order
- 2 l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- 3 if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method

- 1 solves an instance of the Cost- t splitters problem
- 2 if **feasible**, output the feasible P and $c(P)$
- 3 a greedy algorithm

jump $t = 2$ ids



Intuition

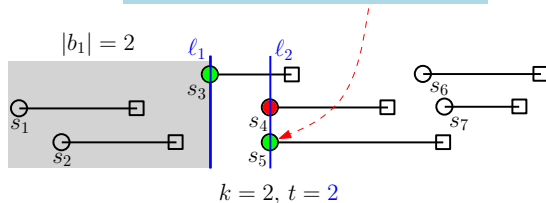
- 1 place splitters in ascending order
- 2 l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- 3 if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method

- ① solves an instance of the Cost- t splitters problem
- ② if **feasible**, output the feasible P and $c(P)$
- ③ a greedy algorithm

jump $t = 2$ ids, move back $\delta[5] = 1$



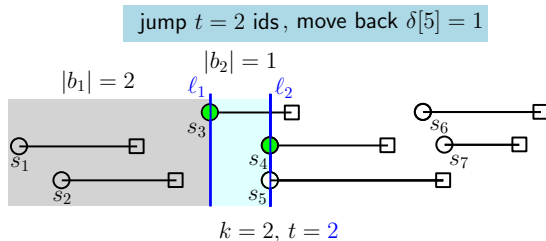
Intuition

- ① place splitters in ascending order
- ② l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- ③ if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method

- 1 solves an instance of the Cost- t splitters problem
- 2 if **feasible**, output the feasible P and $c(P)$
- 3 a greedy algorithm



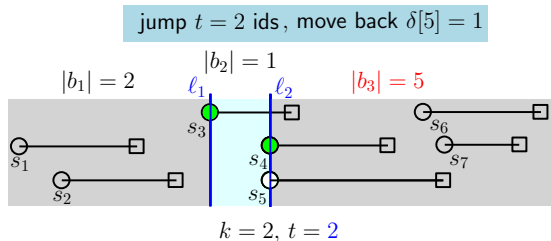
Intuition

- 1 place splitters in ascending order
- 2 l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- 3 if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method

- 1 solves an instance of the Cost- t splitters problem
- 2 if **feasible**, output the feasible P and $c(P)$
- 3 a greedy algorithm



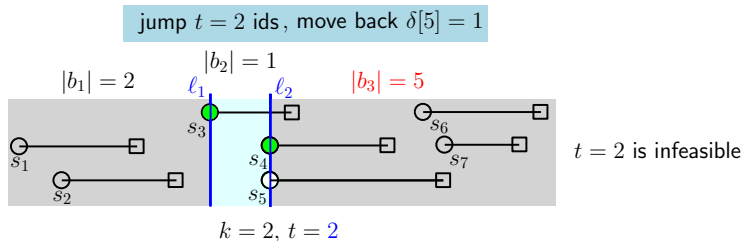
Intuition

- 1 place splitters in ascending order
- 2 l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- 3 if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method

- 1 solves an instance of the Cost- t splitters problem
- 2 if **feasible**, output the feasible P and $c(P)$
- 3 a greedy algorithm



Intuition

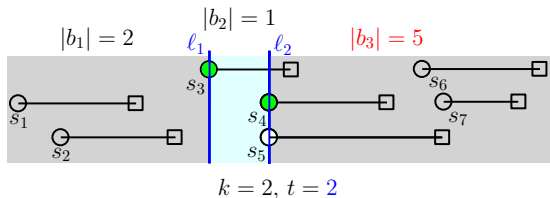
- 1 place splitters in ascending order
- 2 l_{i+1} is pushed as far as possible from l_i , let each new b_i have size t
- 3 if not achievable, move l_{i+1} backward just enough to form the new b_i

t -jump method

- t -jump method

- 1 solves an instance of the Cost- t splitters problem
- 2 if **feasible**, output the feasible P and $c(P)$
- 3 a greedy algorithm

jump $t = 2$ ids, move back $\delta[5] = 1$



$t = 2$ is infeasible

Lemma (**Correctness of t -jump**)

If t -jump returns feasible, then the splitters output constitute a partition with cost $\bar{t} \leq t$. Otherwise, t must be infeasible.

- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

Cost Analysis

- 1 $O(\log N)$ instances of Cost- t splitters problems in a binary search
- 2 Cost- t splitters problem can be answered in $O(k)$ (k is # splitters), $O(k \log N)$ in total ($k \ll N$)

Cost Analysis

- 1 $O(\log N)$ instances of Cost- t splitters problems in a binary search
- 2 Cost- t splitters problem can be answered in $O(k)$ (k is # splitters), $O(k \log N)$ in total ($k \ll N$)
- 3 Sorting intervals and constructing the stabbing-count array take $O(N \log N)$ time

- 1 $O(\log N)$ instances of Cost- t splitters problems in a binary search
- 2 Cost- t splitters problem can be answered in $O(k)$ (k is # splitters), $O(k \log N)$ in total ($k \ll N$)
- 3 Sorting intervals and constructing the stabbing-count array take $O(N \log N)$ time

Theorem

The problem of finding optimal splitters can be solved in $O(N \log N)$ time in internal memory.

- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

External Memory Method

- \mathcal{I} stored in a disk-resident array using $O(N/B)$ blocks

External Memory Method

- \mathcal{I} stored in a disk-resident array using $\mathbf{O(N/B)}$ blocks
- Define the cost of external sorting as

$$SORT(N) = (N/B) \log_{M/B}(N/B)$$

External Memory Method

- \mathcal{I} stored in a disk-resident array using $O(N/B)$ blocks
- Define the cost of external sorting as

$$SORT(N) = (N/B) \log_{M/B}(N/B)$$

Theorem

The problem of finding optimal splitters can be solved using $O(SORT(N))$ I/Os in external memory

External Memory Method

- \mathcal{I} stored in a disk-resident array using $O(N/B)$ blocks
- Define the cost of external sorting as

$$SORT(N) = (N/B) \log_{M/B}(N/B)$$

Theorem

The problem of finding optimal splitters can be solved using $O(SORT(N))$ I/Os in external memory

Adapting the main-memory algorithm ?

- 1 sorting takes $SORT(N)$ I/Os
- 2 solving a cost- t splitters problem takes $O(\min(k, N/B))$ I/Os
- 3 $O(SORT(N) + \min(k, N/B) \log N)$ I/Os in total

External Memory Method

- \mathcal{I} stored in a disk-resident array using $O(N/B)$ blocks
- Define the cost of external sorting as

$$SORT(N) = (N/B) \log_{M/B}(N/B)$$

Theorem

The problem of finding optimal splitters can be solved using $O(SORT(N))$ I/Os in external memory

Adapting the main-memory algorithm ?

- 1 sorting takes $SORT(N)$ I/Os
- 2 solving a cost- t splitters problem takes $O(\min(k, N/B))$ I/Os
- 3 $O(SORT(N) + \min(k, N/B) \log N)$ I/Os in total

Problems ▶ not a clean bound when $k \in [1, N]$ ▶ may require excessive I/Os

- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

Concurrent t -jump method

Definition (**Cost- t testing**)

Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, output **Yes**
- 2 otherwise, output **No**

Concurrent t -jump method

Definition (**Cost- t testing**)

Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, output **Yes**
- 2 otherwise, output **No**

Cost- t Testing vs. **Cost- t Splitters Problem**

- ▶ avoid storing the feasible splitters ($O(k/B)$ space)
- ▶ lead to the concurrent extension of cost- t testing

Concurrent t -jump method

Definition (**Cost- t testing**)

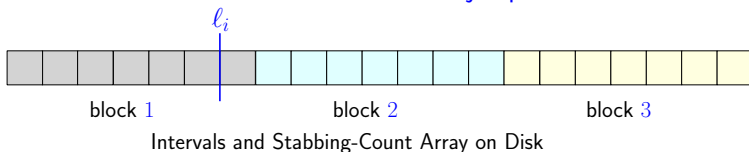
Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, output **Yes**
- 2 otherwise, output **No**

Cost- t Testing vs. **Cost- t Splitters Problem**

- ▶ avoid storing the feasible splitters ($O(k/B)$ space)
- ▶ lead to the concurrent extension of cost- t testing

Intuition of concurrent t -jump



Concurrent t -jump method

Definition (**Cost- t testing**)

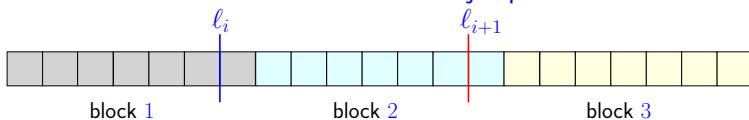
Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, output **Yes**
- 2 otherwise, output **No**

Cost- t Testing vs. Cost- t Splitters Problem

- ▶ avoid storing the feasible splitters ($O(k/B)$ space)
- ▶ lead to the concurrent extension of cost- t testing

Intuition of concurrent t -jump



Intervals and Stabbing-Count Array on Disk

- t -jump scans *forwardly*, next block to be read is *uniquely defined*

Concurrent t -jump method

Definition (**Cost- t testing**)

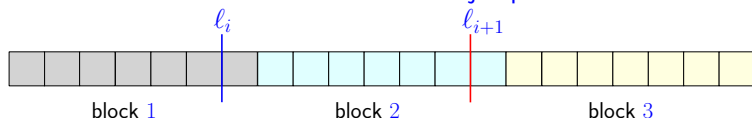
Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, output **Yes**
- 2 otherwise, output **No**

Cost- t Testing vs. Cost- t Splitters Problem

- ▶ avoid storing the feasible splitters ($O(k/B)$ space)
- ▶ lead to the concurrent extension of cost- t testing

Intuition of concurrent t -jump



Intervals and Stabbing-Count Array on Disk

- t -jump scans *forwardly*, next block to be read is *uniquely defined*
- one execution requires $O(1)$ space

Concurrent t -jump method

Definition (**Cost- t testing**)

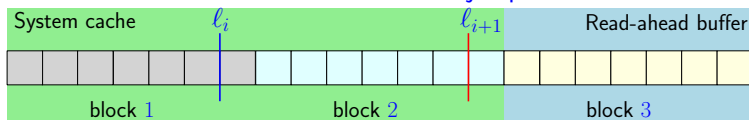
Determine whether there is a size- k partition P with $c(P) \leq t$

- 1 if such P exists, output **Yes**
- 2 otherwise, output **No**

Cost- t Testing vs. Cost- t Splitters Problem

- ▶ avoid storing the feasible splitters ($O(k/B)$ space)
- ▶ lead to the concurrent extension of cost- t testing

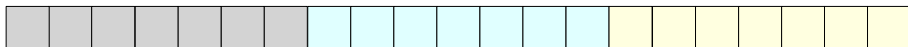
Intuition of concurrent t -jump



Intervals and Stabbing-Count Array on Disk

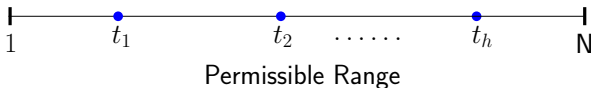
- t -jump scans *forwardly*, next block to be read is *uniquely defined*
- one execution requires $O(1)$ space

Concurrent t -jump method

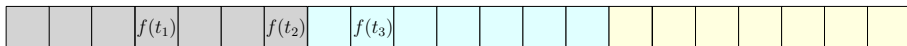


Intervals and Stabbing-Count Array

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$

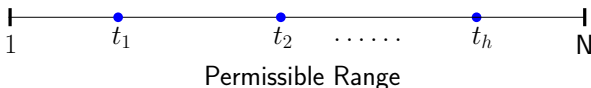


Concurrent t -jump method

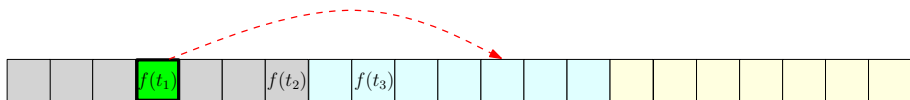


Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$

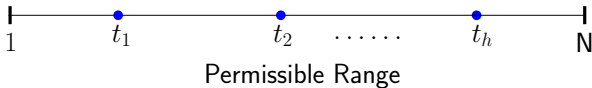


Concurrent t -jump method

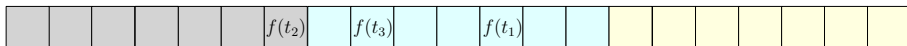


Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$

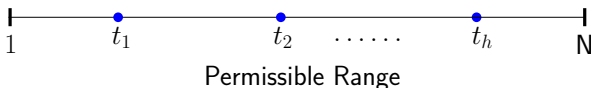


Concurrent t -jump method

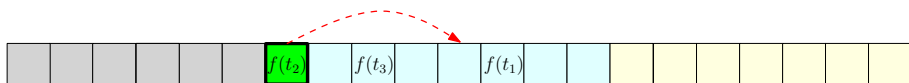


Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$

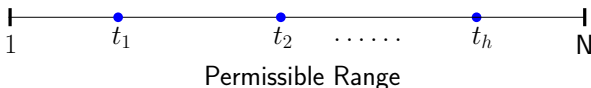


Concurrent t -jump method

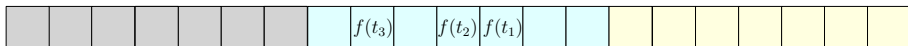


Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$

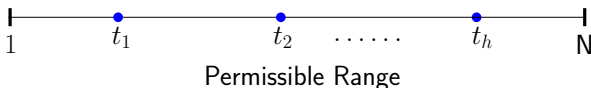


Concurrent t -jump method

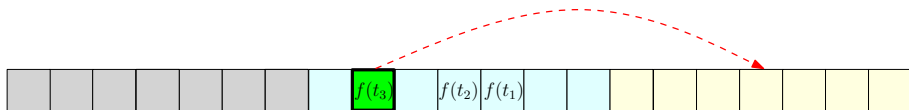


Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$

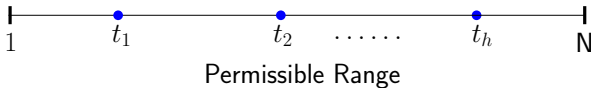


Concurrent t -jump method

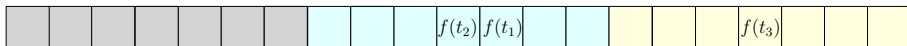


Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$

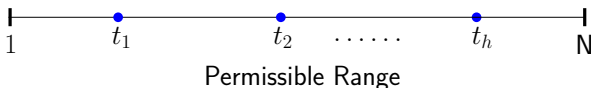


Concurrent t -jump method



Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$

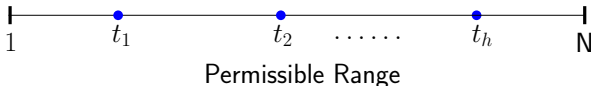


Concurrent t -jump method

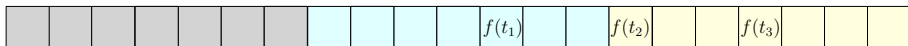


Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$

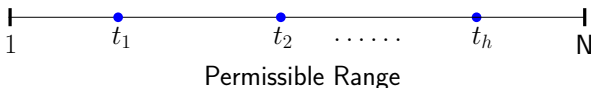


Concurrent t -jump method

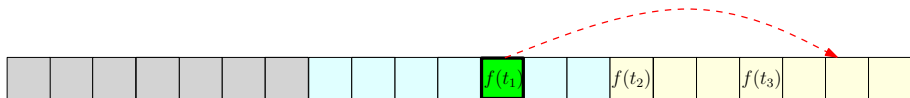


Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$

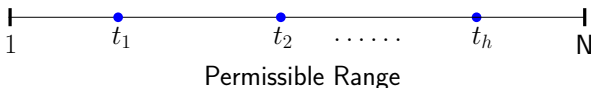


Concurrent t -jump method

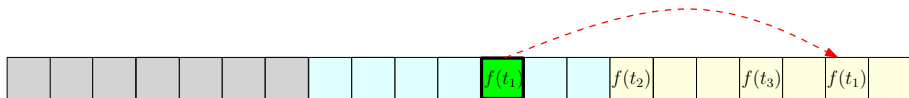


Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$

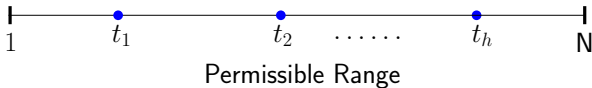


Concurrent t -jump method



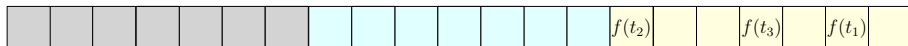
Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$



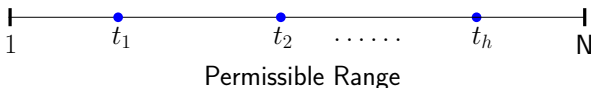
Concurrent t -jump method

✗ cost- t_1 infeasible ✓ cost- t_2 feasible ✓ cost- t_3 feasible



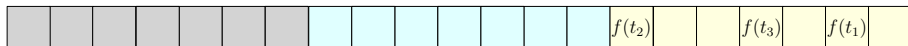
Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$



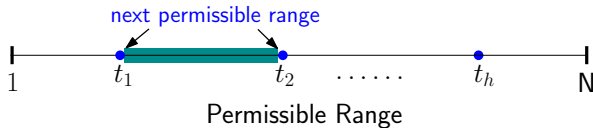
Concurrent t -jump method

✗ cost- t_1 infeasible ✓ cost- t_2 feasible ✓ cost- t_3 feasible



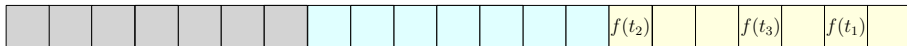
Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$



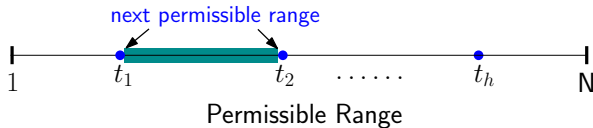
Concurrent t -jump method

✗ cost- t_1 infeasible ✓ cost- t_2 feasible ✓ cost- t_3 feasible



Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- ▶ initialize h threads of cost- t testings, $1 \leq t_1 < t_2 < \dots < t_h \leq N$
- ▶ $f(t_i)$ the frontier of cost- t_i testing
- ▶ at any time activate the thread with $\min(f(t_i))$
- ▶ when t^* is found, one more scan to locate the splitters



- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

- Construct the stabbing-count array: $O(\text{SORT}(N))$ I/Os

Cost Analysis

- Construct the stabbing-count array: $O(\text{SORT}(N))$ I/Os
- One round of Concurrent Cost- t testings: $O(N/B)$ I/Os at most

Cost Analysis

- Construct the stabbing-count array: $O(\text{SORT}(N))$ I/Os
- One round of Concurrent Cost- t testings: $O(N/B)$ I/Os at most
- # rounds of Concurrent Cost- t testings: $O(\log_M N) \leq O(\log_{M/B} N/B)$

Cost Analysis

- Construct the stabbing-count array: $O(\text{SORT}(N))$ I/Os
- One round of Concurrent Cost- t testings: $O(N/B)$ I/Os at most
- # rounds of Concurrent Cost- t testings: $O(\log_M N) \leq O(\log_{M/B} N/B)$
- Cost to find t^* : $\text{SORT}(N)$ at most

Cost Analysis

- Construct the stabbing-count array: $O(\text{SORT}(N))$ I/Os
- One round of Concurrent Cost- t testings: $O(N/B)$ I/Os at most
- # rounds of Concurrent Cost- t testings: $O(\log_M N) \leq O(\log_{M/B} N/B)$
- Cost to find t^* : $\text{SORT}(N)$ at most
- Retrieve the optimal splitters: $O(\min(k, N/B))$ I/Os

- Construct the stabbing-count array: $O(\text{SORT}(N))$ I/Os
- One round of Concurrent Cost- t testings: $O(N/B)$ I/Os at most
- # rounds of Concurrent Cost- t testings: $O(\log_M N) \leq O(\log_{M/B} N/B)$
- Cost to find t^* : $\text{SORT}(N)$ at most
- Retrieve the optimal splitters: $O(\min(k, N/B))$ I/Os

Concurrent t -jump method is as efficient as external sorting!

- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

Experiments: Setup

- Internal: DP, t -jump

Experiments: Setup

- Internal: DP, *t*-jump
- External: *t*-jump, *ct*-jump, *sc*-tree (use *Segment B-tree*)

Experiments: Setup

- Internal: DP, t -jump
- External: t -jump, ct -jump, sc -tree (use *Segment B-tree*)
- Implementation in C++
 - I/O efficient methods are implemented with TPIE

Experiments: Setup

- Internal: DP, *t*-jump
- External: *t*-jump, *ct*-jump, *sc*-tree (use *Segment B-tree*)
- Implementation in C++
 - I/O efficient methods are implemented with TPIE
- Experiments on a Linux machine with 4GB of Mem

Experiments: Setup

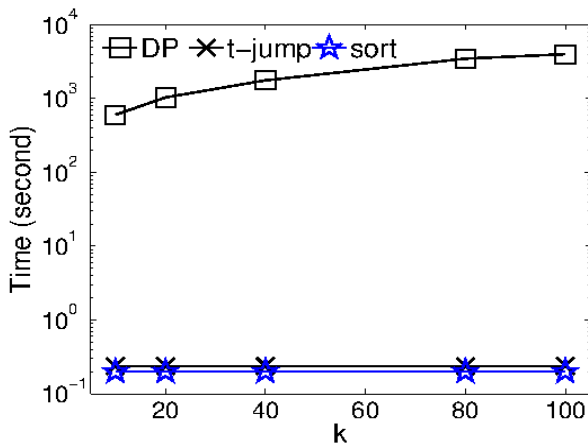
- Internal: DP, *t*-jump
- External: *t*-jump, *ct*-jump, *sc-tree* (use *Segment B-tree*)
- Implementation in C++
 - I/O efficient methods are implemented with TPIE
- Experiments on a Linux machine with 4GB of Mem
- Two large real datasets:
 - *Temp* is a temperature dataset from the *MesoWest*
 - contains measurements from Jan 1997 to Oct 2011
 - *Meme* is obtained from the *Memetracker* Project
 - tracks the frequency of popular quotes over time

Experiments: Setup

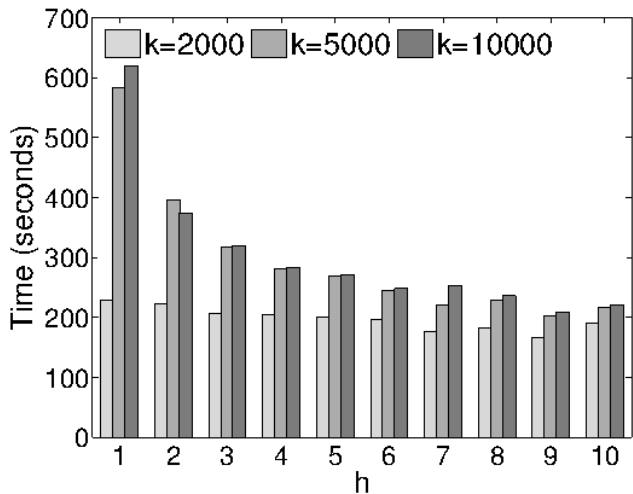
- Internal: DP, t -jump
- External: t -jump, ct -jump, **sc-tree** (use *Segment B-tree*)
- Implementation in C++
 - I/O efficient methods are implemented with TPIE
- Experiments on a Linux machine with 4GB of Mem
- Two large real datasets:
 - **Temp** is a temperature dataset from the *MesoWest*
 - contains measurements from Jan 1997 to Oct 2011
 - **Meme** is obtained from the *Memetracker* Project
 - tracks the frequency of popular quotes over time

	Internal	External
Dataset	a subset of <i>Meme</i>	a subset of <i>Temp</i>
Size	~ 21 MB	~ 5 GB
N	~ 1 million	~ 200 million
k	40	5000
h	not applicable	5

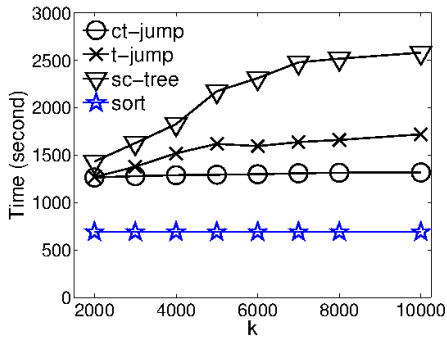
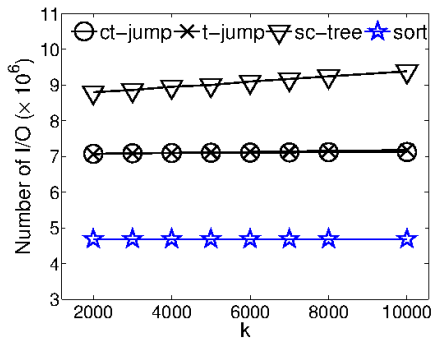
Experiments: Vary k Internal Memory Methods



Experiments: Vary h External Memory Methods



Experiments: Vary k External Memory Methods



- 1 Motivation and Problem Formulation
- 2 A Baseline Method
 - Strategy to Place Splitters
 - Dynamic Programming Approach
 - Cost Analysis
- 3 Internal Memory Method
 - Cost- t Splitter Problem
 - Stabbing-count Array and t -jump method
 - Cost Analysis
- 4 External Memory Method
 - Concurrent t -jump method
 - Cost Analysis
- 5 Experiments
- 6 Conclusion

Conclusion

- We studied the optimal splitters problem for large interval data, which is essential in a distributed and parallel setting

- We studied the optimal splitters problem for large interval data, which is essential in a distributed and parallel setting
- Our best solutions t -jump and ct -jump are more efficient than the baseline solutions
 - both are as efficient as sorting algorithms

- We studied the optimal splitters problem for large interval data, which is essential in a distributed and parallel setting
- Our best solutions t -jump and ct -jump are more efficient than the baseline solutions
 - both are as efficient as sorting algorithms
- Future work includes extending our studies to higher dimensions

Thank You

Q and A

Strategy to Place Splitters

- Where to place splitters?

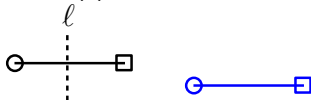
Strategy to Place Splitters

- Where to place splitters?
 - let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.

Strategy to Place Splitters

- Where to place splitters?

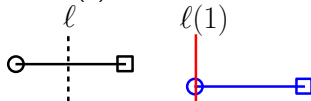
- let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
- for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$



Strategy to Place Splitters

- Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
- for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$

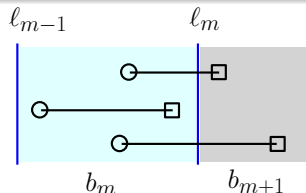


Strategy to Place Splitters

- Where to place splitters?
 - let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
 - for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is undefined. Let P' be a partition with splitters $\ell_1, \dots, \ell_{m-1}$, then $c(P') = c(P)$.



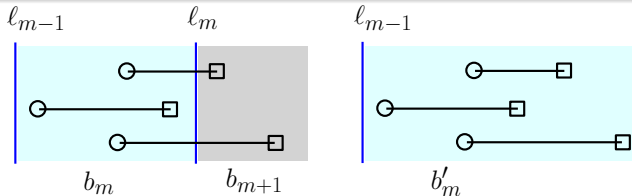
Strategy to Place Splitters

- Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
- for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is undefined. Let P' be a partition with splitters $\ell_1, \dots, \ell_{m-1}$, then $c(P') = c(P)$.



- $b_{m+1} \subseteq b_m = b'_m$
- $c(P') = c(P) \geq |b_m| = |b'_m| \geq |b_{m+1}|$

Strategy to Place Splitters

- Where to place splitters?
 - let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
 - for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is undefined. Let P' be a partition with splitters $\ell_1, \dots, \ell_{m-1}$, then $c(P') = c(P)$.

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is **defined**. Let ℓ_i be the **largest splitter that does not in \mathbf{S}** . Define P' from P by (i) deleting ℓ_i , if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing ℓ_i with $\ell_i(1)$. Then, $c(P') \leq c(P)$.

Strategy to Place Splitters

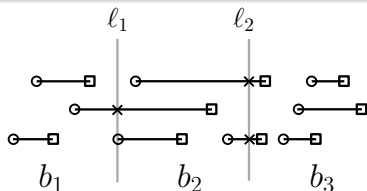
- Where to place splitters?
 - let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
 - for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is undefined. Let P' be a partition with splitters $\ell_1, \dots, \ell_{m-1}$, then $c(P') = c(P)$.

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is defined. Let ℓ_i be the largest splitter that does not in \mathbf{S} . Define P' from P by (i) deleting ℓ_i , if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing ℓ_i with $\ell_i(1)$. Then, $c(P') \leq c(P)$.



$$c(P) = 5$$

Strategy to Place Splitters

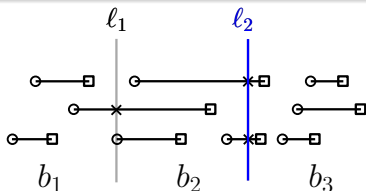
- Where to place splitters?
 - let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
 - for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is undefined. Let P' be a partition with splitters $\ell_1, \dots, \ell_{m-1}$, then $c(P') = c(P)$.

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is defined. Let ℓ_i be the largest splitter that does not in \mathbf{S} . Define P' from P by (i) deleting ℓ_i , if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing ℓ_i with $\ell_i(1)$. Then, $c(P') \leq c(P)$.



$$c(P) = 5$$

Strategy to Place Splitters

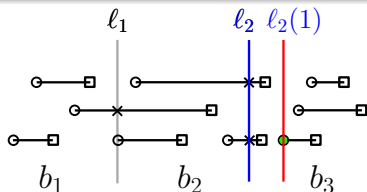
- Where to place splitters?
 - let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
 - for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is undefined. Let P' be a partition with splitters $\ell_1, \dots, \ell_{m-1}$, then $c(P') = c(P)$.

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is defined. Let ℓ_i be the largest splitter that does not in \mathbf{S} . Define P' from P by (i) deleting ℓ_i , if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing ℓ_i with $\ell_i(1)$. Then, $c(P') \leq c(P)$.



$$c(P) = 5$$

Strategy to Place Splitters

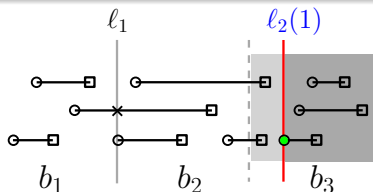
- Where to place splitters?
 - let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
 - for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is undefined. Let P' be a partition with splitters $\ell_1, \dots, \ell_{m-1}$, then $c(P') = c(P)$.

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is defined. Let ℓ_i be the largest splitter that does not in \mathbf{S} . Define P' from P by (i) deleting ℓ_i , if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing ℓ_i with $\ell_i(1)$. Then, $c(P') \leq c(P)$.



- no effect on b_2
- shrink b_3

$$c(P') = 4 \leq c(P) = 5$$

Strategy to Place Splitters

- Where to place splitters?
 - let $\mathcal{I} = \{[s_1, e_1] \dots [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1 \dots s_N\}$ in ascending order.
 - for any splitter ℓ , let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is undefined. Let P' be a partition with splitters $\ell_1, \dots, \ell_{m-1}$, then $c(P') = c(P)$.

Lemma

For any partition P with distinct splitters $\ell_1 < \dots < \ell_m$ and $\ell_m(1)$ is defined. Let ℓ_i be the largest splitter that does not in \mathbf{S} . Define P' from P by (i) deleting ℓ_i , if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing ℓ_i with $\ell_i(1)$. Then, $c(P') \leq c(P)$.

Should always try to split on **S** !