Practical Dynamic de Bruijn Graphs

Victoria Crawford [1,*], **Alan Kuhnle** [1,*], Christina Boucher [1], Rayan Chikhi [2], and Travis Gagie [3]

[1]Computer and Information Science and Engineering, University of Florida, Gainesville, 32306
[2] CNRS, CRIStAL, University of Lille, Lille, France
[3] CeBiB and School of Computer Science and Engineering, Diego Portales University, Santiago, Chile
*These authors contributed equally to this work.

de Bruijn Graph

- Alphabet Σ of size σ

de Bruijn Graph

- Alphabet Σ of size σ
- Nodes correspond to a subset of k-mers

de Bruijn Graph

- Alphabet Σ of size σ
- Nodes correspond to a subset of k-mers
- Edges correspond to a subset of $(k+1)$-mers

de Bruijn Graph

- Alphabet Σ of size σ
- Nodes correspond to a subset of k-mers
- Edges correspond to a subset of $(k+1)$-mers
- An edge from node u to node v if and only if there is a $(k+1)$-mer with prefix u and suffix v

de Bruijn Graph

[1] Holley, G., Wittler, R., and Stoye, J. (2015). Bloom filter trie—a data structure for pan-genome storage. In Proc. of *WABI*, pages 217–230.

de Bruijn Graph

- compact representations of de Bruijn are needed

[1] Holley, G., Wittler, R., and Stoye, J. (2015). Bloom filter trie—a data structure for pan-genome storage. In Proc. of *WABI*, pages 217–230.

de Bruijn Graph

- compact representations of de Bruijn are needed
- if the graph changes, would be nice to update the data structure rather than recomputation

[1] Holley, G., Wittler, R., and Stoye, J. (2015). Bloom filter trie—a data structure for pan-genome storage. In Proc. of *WABI*, pages 217–230.

de Bruijn Graph

- compact representations of de Bruijn are needed
- if the graph changes, would be nice to update the data structure rather than recomputation
- for example, necessary when pruning spurious elements from graph

[1] Holley, G., Wittler, R., and Stoye, J. (2015). Bloom filter trie—a data structure for pan-genome storage. In Proc. of *WABI*, pages 217–230.

de Bruijn Graph

- compact representations of de Bruijn are needed
- if the graph changes, would be nice to update the data structure rather than recomputation
- for example, necessary when pruning spurious elements from graph
- BloomFilterTrie[1] supports k-mer insertions, but not deletions

[1] Holley, G., Wittler, R., and Stoye, J. (2015). Bloom filter trie—a data structure for pan-genome storage. In Proc. of *WABI*, pages 217–230.

Belazzougui et al. (2016)

Belazzougui et al. (2016)

- "Fully Dynamic de Bruijn Graphs" by Djamal Belazzougui, Travis Gagie, Veli Makinen, and Marco Previtali in SPIRE 2016

Belazzougui et al. (2016)

- "Fully Dynamic de Bruijn Graphs" by Djamal Belazzougui, Travis Gagie, Veli Makinen, and Marco Previtali in SPIRE 2016
  - Fully dynamic de Bruijn graph representation

Belazzougui et al. (2016)

- "Fully Dynamic de Bruijn Graphs" by Djamal Belazzougui, Travis Gagie, Veli Makinen, and Marco Previtali in SPIRE 2016
  - Fully dynamic de Bruijn graph representation
  - Exact membership queries

Contributions

Contributions

- Practical implementation of the fully dynamic de Bruijn Graph described by Belazzougui et al.

Contributions

- Practical implementation of the fully dynamic de Bruijn Graph described by Belazzougui et al.
  - Supports edge insertions and deletions

Contributions

- Practical implementation of the fully dynamic de Bruijn Graph described by Belazzougui et al.
  - Supports edge insertions and deletions
  - Supports limited number of node insertions and deletions

Contributions

- Practical implementation of the fully dynamic de Bruijn Graph described by Belazzougui et al.
  - Supports edge insertions and deletions
  - Supports limited number of node insertions and deletions
- Competitive in construction time / space, query time with alternative compact implementations of de Bruijn graphs

Overview of Data Structure

- Let G be de Bruijn graph with n nodes

Overview of Data Structure

- Let G be de Bruijn graph with n nodes
  1. Hash function that takes k-mers to $\{0, ..., n-1\}$

Overview of Data Structure

- Let G be de Bruijn graph with n nodes
  1. Hash function that takes k-mers to $\{0, ..., n-1\}$
  2. IN, OUT matrices contain edge information

Overview of Data Structure

- Let G be de Bruijn graph with n nodes
  1. Hash function that takes k-mers to $\{0, ..., n-1\}$
  2. IN, OUT matrices contain edge information
  3. Covering forest

Overview of Data Structure

- Let G be de Bruijn graph with n nodes
  1. Hash function that takes k-mers to $\{0, ..., n-1\}$
  2. IN, OUT matrices contain edge information
  3. Covering forest
     - Each tree contains at least $k \log \sigma$ nodes

Overview of Data Structure

- Let G be de Bruijn graph with n nodes
  1. Hash function that takes k-mers to $\{0, ..., n-1\}$
  2. IN, OUT matrices contain edge information
  3. Covering forest
     - Each tree contains at least $k \log \sigma$ nodes
     - Exactly one k-mer stored for each tree ($O(k \log \sigma)$ bits)

Overview of Data Structure

- Let G be de Bruijn graph with n nodes
  1. Hash function that takes k-mers to $\{0, ..., n-1\}$
  2. IN, OUT matrices contain edge information
  3. Covering forest
     - Each tree contains at least $k \log \sigma$ nodes
     - Exactly one k-mer stored for each tree ($O(k \log \sigma)$ bits)
- Supports exact membership queries

Overview of Data Structure

- Size: $O(n\sigma)$ bits if no small connected components

Overview of Data Structure

- Size: $O(n\sigma)$ bits if no small connected components
- Expected construction time: $O(kn + n\sigma)$

Overview of Data Structure

- Size: $O(n\sigma)$ bits if no small connected components
- Expected construction time: $O(kn + n\sigma)$
- Membership query time: $O(k \log \sigma)$

Overview of Data Structure

- Size: $O(n\sigma)$ bits if no small connected components
- Expected construction time: $O(kn + n\sigma)$
- Membership query time: $O(k \log \sigma)$
- Edge insertion / deletion time: $O(k \log \sigma)$

Hash function f

Hash function f

- f takes k-mers to $\{0, ..., n-1\}$

Hash function f

- f takes k-mers to $\{0, ..., n-1\}$
- f is a bijection on the set of k-mers N in the de Bruijn graph

Hash function f

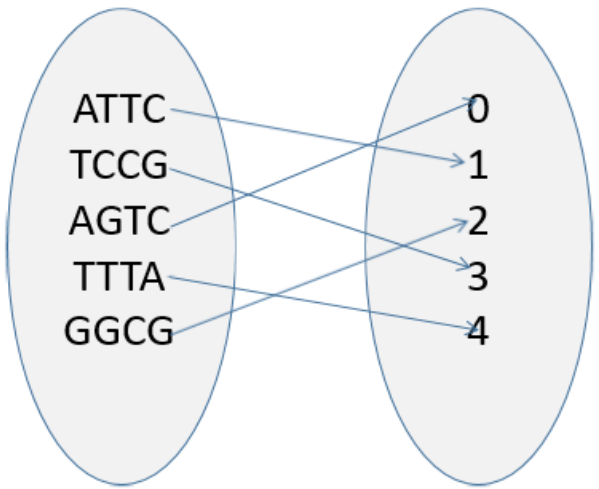
- f takes k-mers to $\{0, ..., n-1\}$
- f is a bijection on the set of k-mers N in the de Bruijn graph
- $f = h \circ g$ where

Hash function f

- f takes k-mers to $\{0, ..., n-1\}$
- f is a bijection on the set of k-mers N in the de Bruijn graph
- $f = h \circ g$ where
  - g is a Karp-Rabin hash function that is injective on N

Hash function f

- f takes k-mers to $\{0, ..., n-1\}$
- f is a bijection on the set of k-mers N in the de Bruijn graph
- $f = h \circ g$ where
  - g is a Karp-Rabin hash function that is injective on N
  - h is a minimum perfect hash function



IN and OUT

IN and OUT

- Edges stored in two matrices: IN and OUT

IN and OUT

- Edges stored in two matrices: IN and OUT
  - Each of size $n \times \sigma$

IN and OUT

- Edges stored in two matrices: IN and OUT
  - Each of size $n \times \sigma$

$$(u = ba_1 \dots a_{k-1}, v = a_1 a_2 \dots a_{k-1} c) \in E \iff OUT(f(u), c) = 1, IN(f(v), b) = 1.$$

Covering Forest

Covering Forest

- The de Bruijn graph is partitioned into a forest $\mathcal{F}$ with directed edges

Covering Forest

- The de Bruijn graph is partitioned into a forest $\mathcal{F}$ with directed edges
  - Edges are a subset of the de Bruijn graph edges

Covering Forest

- The de Bruijn graph is partitioned into a forest $\mathcal{F}$ with directed edges
  - Edges are a subset of the de Bruijn graph edges
  - Each tree $T \in \mathcal{F}$ has bounded height $\alpha \leq h(T) \leq 3\alpha$, $\alpha = k \log \sigma$

Covering Forest

- The de Bruijn graph is partitioned into a forest $\mathcal{F}$ with directed edges
  - Edges are a subset of the de Bruijn graph edges
  - Each tree $T \in \mathcal{F}$ has bounded height $\alpha \leq h(T) \leq 3\alpha$, $\alpha = k \log \sigma$
    - Unless a connected component is too small

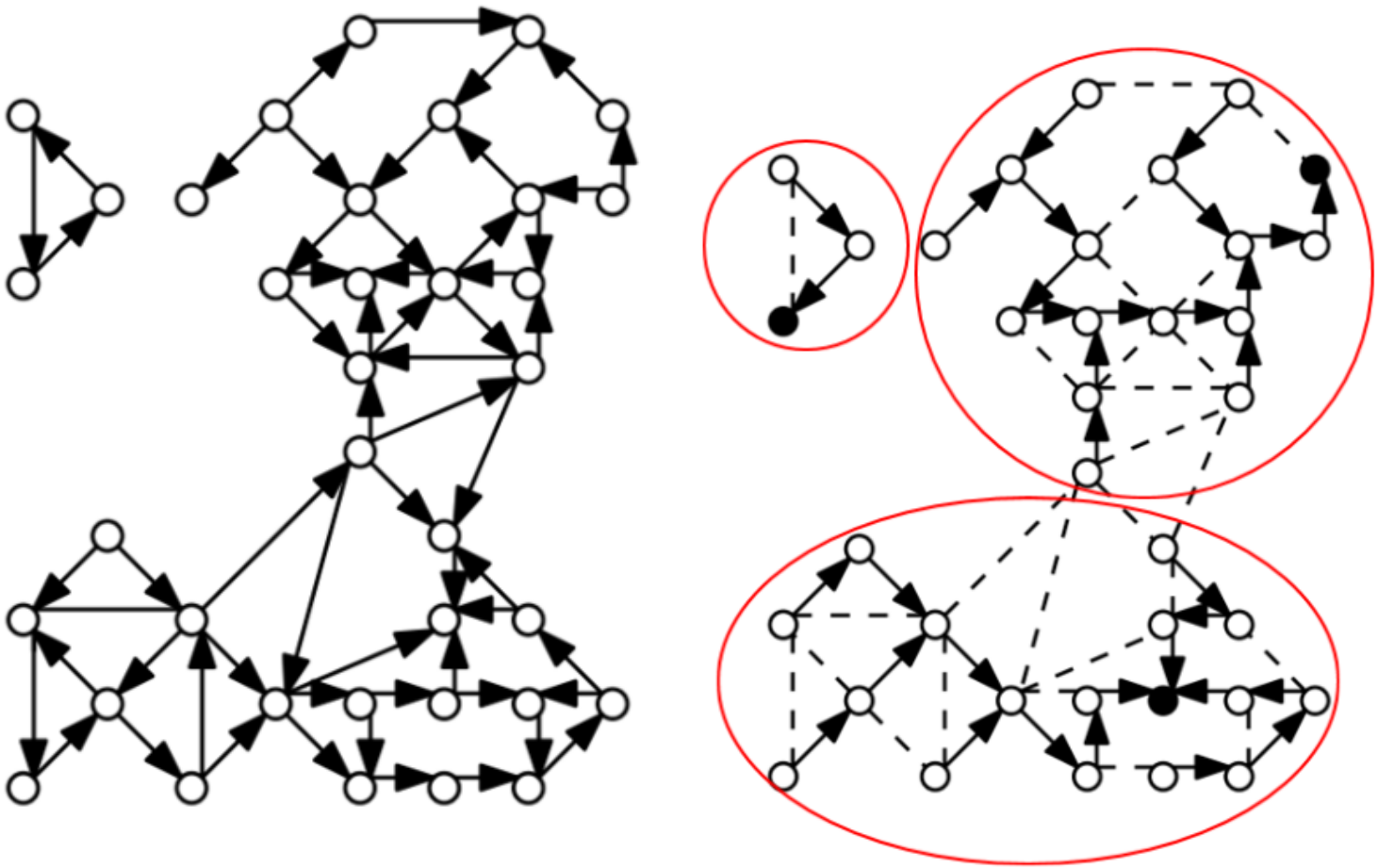
Covering Forest

- The de Bruijn graph is partitioned into a forest $\mathcal{F}$ with directed edges
  - Edges are a subset of the de Bruijn graph edges
  - Each tree $T \in \mathcal{F}$ has bounded height $\alpha \leq h(T) \leq 3\alpha$, $\alpha = k \log \sigma$
    - Unless a connected component is too small
  - k-mer of root for each tree is stored

Covering Forest

- The de Bruijn graph is partitioned into a forest $\mathcal{F}$ with directed edges
  - Edges are a subset of the de Bruijn graph edges
  - Each tree $T \in \mathcal{F}$ has bounded height $\alpha \leq h(T) \leq 3\alpha$, $\alpha = k \log \sigma$
    - Unless a connected component is too small
  - k-mer of root for each tree is stored
  - From every node there is a path to a root along the forest edges

Covering Forest



Overview of Implementation

- **Hash function implementation**
- IN, OUT, and forest implementation
- Forest construction procedure
- Membership query procedure
- Dynamic edges
- Limited dynamic nodes

Hash Function Implementation

[2]Limasset, A., Rizk, G., Chikhi, R., and Peterlongo, P. (2017). Fast and scalable minimal perfect hashing for massive key sets. In Proc. of *SEA* , pages 25:1–25:16.

Hash Function Implementation

- Hash f takes k-mers to $\{0, ..., n-1\}$

[2] Limasset, A., Rizk, G., Chikhi, R., and Peterlongo, P. (2017). Fast and scalable minimal perfect hashing for massive key sets. In Proc. of *SEA* , pages 25:1–25:16.

Hash Function Implementation

- Hash f takes k-mers to $\{0, ..., n-1\}$
- f is bijective when restricted to nodes N of de Bruijn graph

[2] Limasset, A., Rizk, G., Chikhi, R., and Peterlongo, P. (2017). Fast and scalable minimal perfect hashing for massive key sets. In Proc. of *SEA* , pages 25:1–25:16.

Hash Function Implementation

- Hash f takes k-mers to $\{0, ..., n-1\}$
- f is bijective when restricted to nodes N of de Bruijn graph
- $f = h \circ g$ where

[2] Limasset, A., Rizk, G., Chikhi, R., and Peterlongo, P. (2017). Fast and scalable minimal perfect hashing for massive key sets. In Proc. of *SEA* , pages 25:1–25:16.

Hash Function Implementation

- Hash f takes k-mers to $\{0, ..., n-1\}$
- f is bijective when restricted to nodes N of de Bruijn graph
- $f = h \circ g$ where
  - g is a Karp-Rabin hash function that is injective on N

[2] Limasset, A., Rizk, G., Chikhi, R., and Peterlongo, P. (2017). Fast and scalable minimal perfect hashing for massive key sets. In Proc. of *SEA* , pages 25:1–25:16.

Hash Function Implementation

- Hash f takes k-mers to $\{0, ..., n-1\}$
- f is bijective when restricted to nodes N of de Bruijn graph
- $f = h \circ g$ where
  - g is a Karp-Rabin hash function that is injective on N
  - h is a minimum perfect hash function (MPHF)

[2] Limasset, A., Rizk, G., Chikhi, R., and Peterlongo, P. (2017). Fast and scalable minimal perfect hashing for massive key sets. In Proc. of *SEA* , pages 25:1–25:16.

Hash Function Implementation

- Hash f takes k-mers to $\{0, ..., n-1\}$
- f is bijective when restricted to nodes N of de Bruijn graph
- $f = h \circ g$ where
  - g is a Karp-Rabin hash function that is injective on N
  - h is a minimum perfect hash function (MPHF)
- The `BBHash`[2] MPHF is used for h

[2] Limasset, A., Rizk, G., Chikhi, R., and Peterlongo, P. (2017). Fast and scalable minimal perfect hashing for massive key sets. In Proc. of *SEA* , pages 25:1–25:16.

Karp-Rabin f Implementation

Karp-Rabin f Implementation

Definition (Karp-Rabin Hash)

Given a prime P and base r, a Rabin-Karp hash function f is a function defined over the space of all strings of length k such that $f(x_1 ... x_k) = (\sum_{i=1}^{n} x_i r^i) \mathrm{mod} P$.

Karp-Rabin f Implementation

Definition (Karp-Rabin Hash)

Given a prime P and base r, a Rabin-Karp hash function f is a function defined over the space of all strings of length k such that $f(x_1 ... x_k) = (\sum_{i=1}^{n} x_i r^i) \mathrm{mod} P$.

- Generate a prime P and base r such that the KR hash function is injective on N, the set of k-mers in the de Bruijn graph

Karp-Rabin f Implementation

Definition (Karp-Rabin Hash)

Given a prime P and base r, a Rabin-Karp hash function f is a function defined over the space of all strings of length k such that $f(x_1 ... x_k) = (\sum_{i=1}^{n} x_i r^i) \mathrm{mod} P$.

- Generate a prime P and base r such that the KR hash function is injective on N, the set of k-mers in the de Bruijn graph
  - Set P to the smallest prime greater than $k|N|^2$

Karp-Rabin f Implementation

Definition (Karp-Rabin Hash)

Given a prime P and base r, a Rabin-Karp hash function f is a function defined over the space of all strings of length k such that $f(x_1...x_k) = (\sum_{i=1}^{n} x_i r^i) \mathrm{mod} P$.

- Generate a prime P and base r such that the KR hash function is injective on N, the set of k-mers in the de Bruijn graph
  - Set P to the smallest prime greater than $k|N|^2$
  - Pick a random number r in $\{1, ..., P\}$ for the base

Karp-Rabin f Implementation

Definition (Karp-Rabin Hash)

Given a prime P and base r, a Rabin-Karp hash function f is a function defined over the space of all strings of length k such that $f(x_1...x_k) = (\sum_{i=1}^{n} x_i r^i) \mathrm{mod} P$.

- Generate a prime P and base r such that the KR hash function is injective on N, the set of k-mers in the de Bruijn graph
  - Set P to the smallest prime greater than $k|N|^2$
  - Pick a random number r in $\{1, ..., P\}$ for the base
  - Test if this Karp-Rabin hash is injective on the set of k-mers

Karp-Rabin f Implementation

Definition (Karp-Rabin Hash)

Given a prime P and base r, a Rabin-Karp hash function f is a function defined over the space of all strings of length k such that $f(x_1...x_k) = (\sum_{i=1}^{n} x_i r^i) \mathrm{mod} P$.

- Generate a prime P and base r such that the KR hash function is injective on N, the set of k-mers in the de Bruijn graph
  - Set P to the smallest prime greater than $k|N|^2$
  - Pick a random number r in $\{1, ..., P\}$ for the base
  - Test if this Karp-Rabin hash is injective on the set of k-mers
  - If not injective, try a new base r.

Karp-Rabin f Implementation

Definition (Karp-Rabin Hash)

Given a prime P and base r, a Rabin-Karp hash function f is a function defined over the space of all strings of length k such that $f(x_1...x_k) = (\sum_{i=1}^{n} x_i r^i) \mathrm{mod} P$.

- Generate a prime P and base r such that the KR hash function is injective on N, the set of k-mers in the de Bruijn graph
  - Set P to the smallest prime greater than $k|N|^2$
  - Pick a random number r in $\{1, ..., P\}$ for the base
  - Test if this Karp-Rabin hash is injective on the set of k-mers
  - If not injective, try a new base r.
- Expected time to construct: $O(kn)$ (Belazzougui et al., 2016)

Karp-Rabin f Implementation

Definition

Karp-Rabin Hash Function Given a prime P and base r, a Rabin-Karp hash function f is a function defined over the space of all strings of length k such that $f(x_1 ... x_k) = (\sum_{i=1}^{n} x_i r^i) \mathrm{mod} P$.

- Powers of r: $r, r^2, \ldots, r^k \mod P$ are precomputed and stored for use in Karp-Rabin computations

Karp-Rabin f Implementation

Definition

Karp-Rabin Hash Function Given a prime P and base r, a Rabin-Karp hash function f is a function defined over the space of all strings of length k such that $f(x_1 ... x_k) = (\sum_{i=1}^{n} x_i r^i) \mathrm{mod} P$.

- Powers of r: $r, r^2, \ldots, r^k \mod P$ are precomputed and stored for use in Karp-Rabin computations
- $\rightarrow O(k)$ computation time for hash function

Updating the hash value

- Suppose we have a k-mer m, with hash $h \circ g(m)$

Updating the hash value

- Suppose we have a k-mer m, with hash $h \circ g(m)$
- want to know hash value of a neighbor of m

Updating the hash value

- Suppose we have a k-mer m, with hash $h \circ g(m)$
- want to know hash value of a neighbor of m
- Update the KR value $g(m)$ to $g(n)$ in $O(1)$

Updating the hash value

- Suppose we have a k-mer m, with hash $h \circ g(m)$
- want to know hash value of a neighbor of m
- Update the KR value $g(m)$ to $g(n)$ in $O(1)$
- For example, if n is an OUT-neighbor with letter *last*, and m starts with letter *first*:

$$g(n) = \frac{(g(m) - \mathit{first} \cdot r)}{r} + \mathit{last} \cdot r^K$$

Updating the hash value

- Suppose we have a k-mer m, with hash $h \circ g(m)$
- want to know hash value of a neighbor of m
- Update the KR value $g(m)$ to $g(n)$ in $O(1)$
- For example, if n is an OUT-neighbor with letter *last*, and m starts with letter *first*:

$$g(n) = \frac{(g(m) - \mathit{first} \cdot r)}{r} + \mathit{last} \cdot r^K$$

- To compute modulo P, precompute $r^{-1} \mod P$ with generalized Euclidean algorithm

Updating the hash value

- Suppose we have a k-mer m, with hash $h \circ g(m)$
- want to know hash value of a neighbor of m
- Update the KR value $g(m)$ to $g(n)$ in $O(1)$
- For example, if n is an OUT-neighbor with letter *last*, and m starts with letter *first*:

$$g(n) = \frac{(g(m) - \mathit{first} \cdot r)}{r} + \mathit{last} \cdot r^K$$

- To compute modulo P, precompute $r^{-1} \mod P$ with generalized Euclidean algorithm
- Finally, apply h to $g(n)$ to get hash value

Overview of Implementation

- Hash function implementation
- **IN, OUT, and forest implementation**
- Forest construction procedure
- Membership query procedure
- Dynamic edges
- Limited dynamic nodes

Building IN and OUT

- IN and OUT are binary matrices of size n (number of kmers) by σ (the size of our alphabet, 4) that store edge information.

Building IN and OUT

- IN and OUT are binary matrices of size n (number of kmers) by σ (the size of our alphabet, 4) that store edge information.
- The rows are hash values (representing each node) and the columns represent letters.

Building IN and OUT

- IN and OUT are binary matrices of size n (number of kmers) by σ (the size of our alphabet, 4) that store edge information.
- The rows are hash values (representing each node) and the columns represent letters.
- All entries stored in $n \cdot \sigma$ bits.

Building IN and OUT

- IN and OUT are binary matrices of size n (number of kmers) by σ (the size of our alphabet, 4) that store edge information.
- The rows are hash values (representing each node) and the columns represent letters.
- All entries stored in $n \cdot \sigma$ bits.
- To construct, simply read through edge k-mers, and apply the hash function to set IN,OUT

The Forest Data Structure

- The forest is stored in two parts:

The Forest Data Structure

- The forest is stored in two parts:
  - array of bits stores forest edge information

The Forest Data Structure

- The forest is stored in two parts:
  - array of bits stores forest edge information
  - a map taking hash values of roots to their k-mer value

The Forest Data Structure

- The forest is stored in two parts:
  - array of bits stores forest edge information
  - a map taking hash values of roots to their k-mer value
- Each node is assigned 4 bits:

The Forest Data Structure

- The forest is stored in two parts:
  - array of bits stores forest edge information
  - a map taking hash values of roots to their k-mer value
- Each node is assigned 4 bits:
  - first bit tells if the node is a root

The Forest Data Structure

- The forest is stored in two parts:
  - array of bits stores forest edge information
  - a map taking hash values of roots to their k-mer value
- Each node is assigned 4 bits:
  - first bit tells if the node is a root
  - second bit tells whether the parent in the forest is accessed via IN or OUT

The Forest Data Structure

- The forest is stored in two parts:
  - array of bits stores forest edge information
  - a map taking hash values of roots to their k-mer value
- Each node is assigned 4 bits:
  - first bit tells if the node is a root
  - second bit tells whether the parent in the forest is accessed via IN or OUT
  - last two bits specify the letter

The Forest Data Structure

- The forest is stored in two parts:
  - array of bits stores forest edge information
  - a map taking hash values of roots to their k-mer value
- Each node is assigned 4 bits:
  - first bit tells if the node is a root
  - second bit tells whether the parent in the forest is accessed via IN or OUT
  - last two bits specify the letter

The Forest Data Structure

- The forest is stored in two parts:
  - array of bits stores forest edge information
  - a map taking hash values of roots to their k-mer value
- Each node is assigned 4 bits:
  - first bit tells if the node is a root
  - second bit tells whether the parent in the forest is accessed via IN or OUT
  - last two bits specify the letter
- Example:

$$\cdots \text{Other nodes data} \cdots 01001 \cdots \overbrace{\underbrace{0}_{\text{root?}}\ \underbrace{1}_{\text{IN?}}\ \underbrace{01}_{\text{Letter}}}^{\text{Data for node } i} \cdots$$

The Forest Data Structure

- The forest is stored in two parts:
  - array of bits stores forest edge information
  - a map taking hash values of roots to their k-mer value
- Each node is assigned 4 bits:
  - first bit tells if the node is a root
  - second bit tells whether the parent in the forest is accessed via IN or OUT
  - last two bits specify the letter
- Example:

$$\cdots \text{Other nodes data} \cdots 01001 \cdots \overbrace{\underbrace{0}_{\text{root?}}\ \underbrace{1}_{\text{IN?}}\ \underbrace{01}_{\text{Letter}}}^{\text{Data for node } i} \cdots$$

Overview of Implementation

- Hash function implementation
- IN, OUT, and forest implementation
- **Forest construction procedure**
- Membership query procedure
- Dynamic edges
- Limited dynamic nodes

Forest Construction

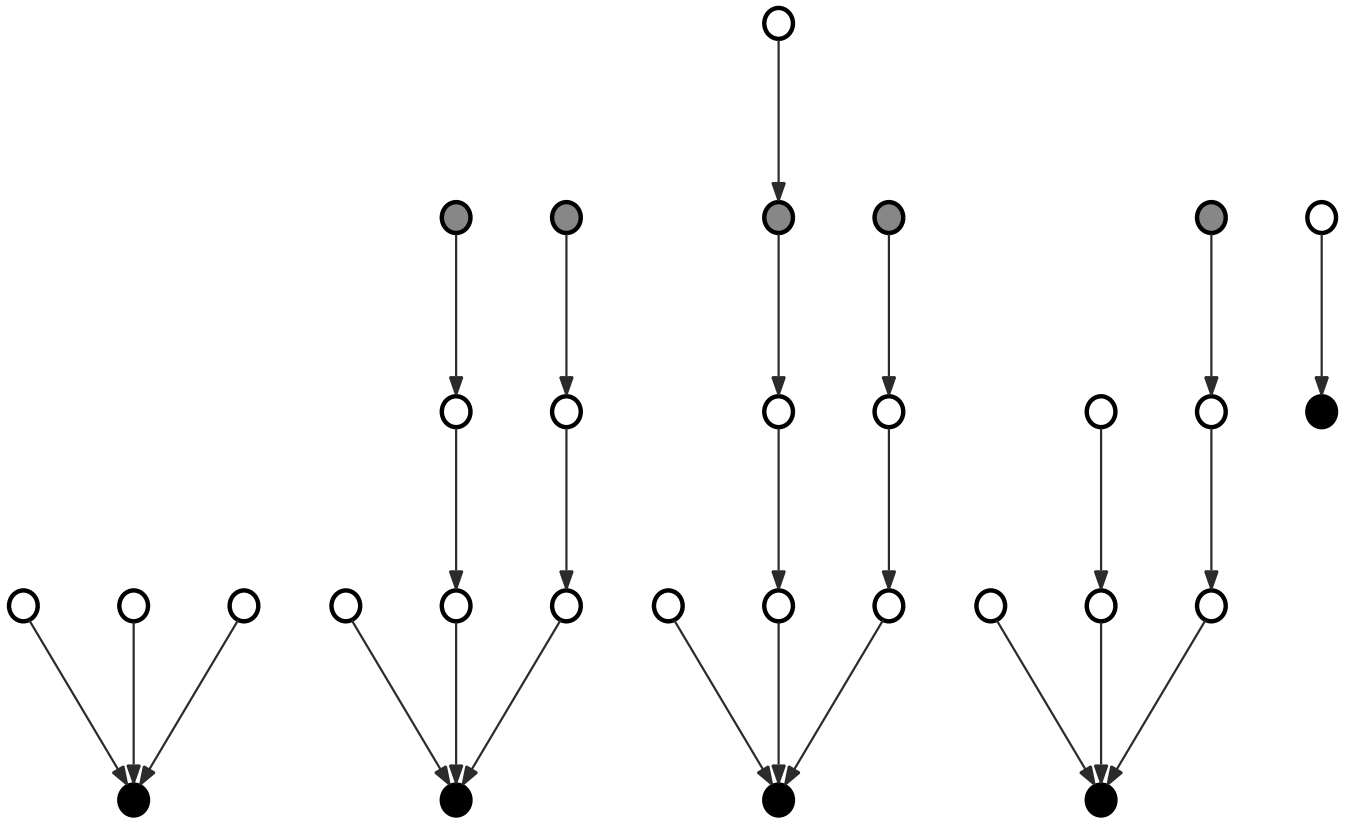
- Each tree should be between height α and 3α where $\alpha = k \log \sigma$.

Forest Construction

- Each tree should be between height α and 3α where $\alpha = k \log \sigma$.
- Conduct breadth first search of the de Bruijn graph ignoring edge directions

Forest Construction

- Each tree should be between height α and 3α where $\alpha = k \log \sigma$.
- Conduct breadth first search of the de Bruijn graph ignoring edge directions
- Break the graph up into trees in the desired height range during BFS.


(a)
(b)
(c)
(d)

Overview of Implementation

- Hash function implementation
- IN, OUT, and forest implementation
- Forest construction procedure
- **Membership query procedure**
- Dynamic edges
- Limited dynamic nodes

Membership Query

Membership Query

- Exact membership queries using the forest

Membership Query

- Exact membership queries using the forest
- Q. Is k-mer $u \in G$?

Membership Query

- Exact membership queries using the forest
- Q. Is k-mer $u \in G$?
  1. Hash u to get $f(u)$

Membership Query

- Exact membership queries using the forest
- Q. Is k-mer $u \in G$?
  1. Hash u to get $f(u)$
  2. Follow forest edge towards the root

Membership Query

- Exact membership queries using the forest
- Q. Is k-mer $u \in G$?
  1. Hash u to get $f(u)$
  2. Follow forest edge towards the root
     - Update k-mer using IN and OUT, check validity

Membership Query

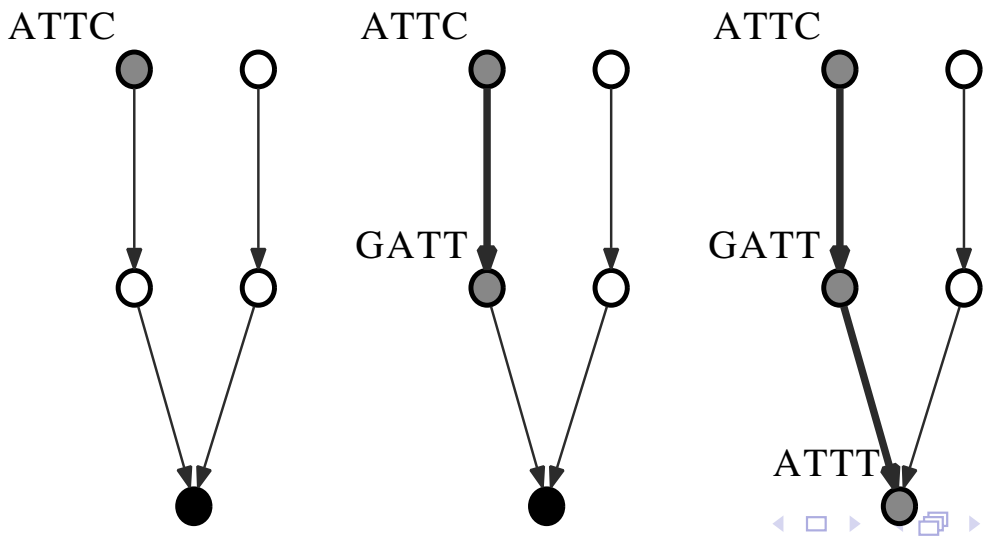
- Exact membership queries using the forest
- Q. Is k-mer $u \in G$?
  1. Hash u to get $f(u)$
  2. Follow forest edge towards the root
     - Update k-mer using IN and OUT, check validity
     - Compare against stored k-mer of root

Membership Query

- Exact membership queries using the forest
- Q. Is k-mer $u \in G$?
  1. Hash u to get $f(u)$
  2. Follow forest edge towards the root
     - Update k-mer using IN and OUT, check validity
     - Compare against stored k-mer of root
- Time: $O(k)$ for first hash computation; $O(1)$ for each update $\rightarrow O(k \log \sigma)$ by max tree height.

Membership Query

- Exact membership queries using the forest
- Q. Is k-mer $u \in G$?
  1. Hash u to get $f(u)$
  2. Follow forest edge towards the root
     - Update k-mer using IN and OUT, check validity
     - Compare against stored k-mer of root
- Time: $O(k)$ for first hash computation; $O(1)$ for each update $\rightarrow O(k \log \sigma)$ by max tree height.



Overview of Implementation

- Hash function implementation
- IN, OUT, and forest implementation
- Forest construction procedure
- Membership query procedure
- **Dynamic edges**
- Limited dynamic nodes

Edge Addition

Edge Addition

- Update IN and OUT

Edge Addition

- Update IN and OUT
- Only update the forest if at least one endpoint of the edge lies within a small connected component

Edge Addition

- Update IN and OUT
- Only update the forest if at least one endpoint of the edge lies within a small connected component
  - Merge trees in time $O(k \log \sigma)$

Edge Removal

Edge Removal

- Update IN and OUT

Edge Removal

- Update IN and OUT
- Only update the forest if edge removed was in forest

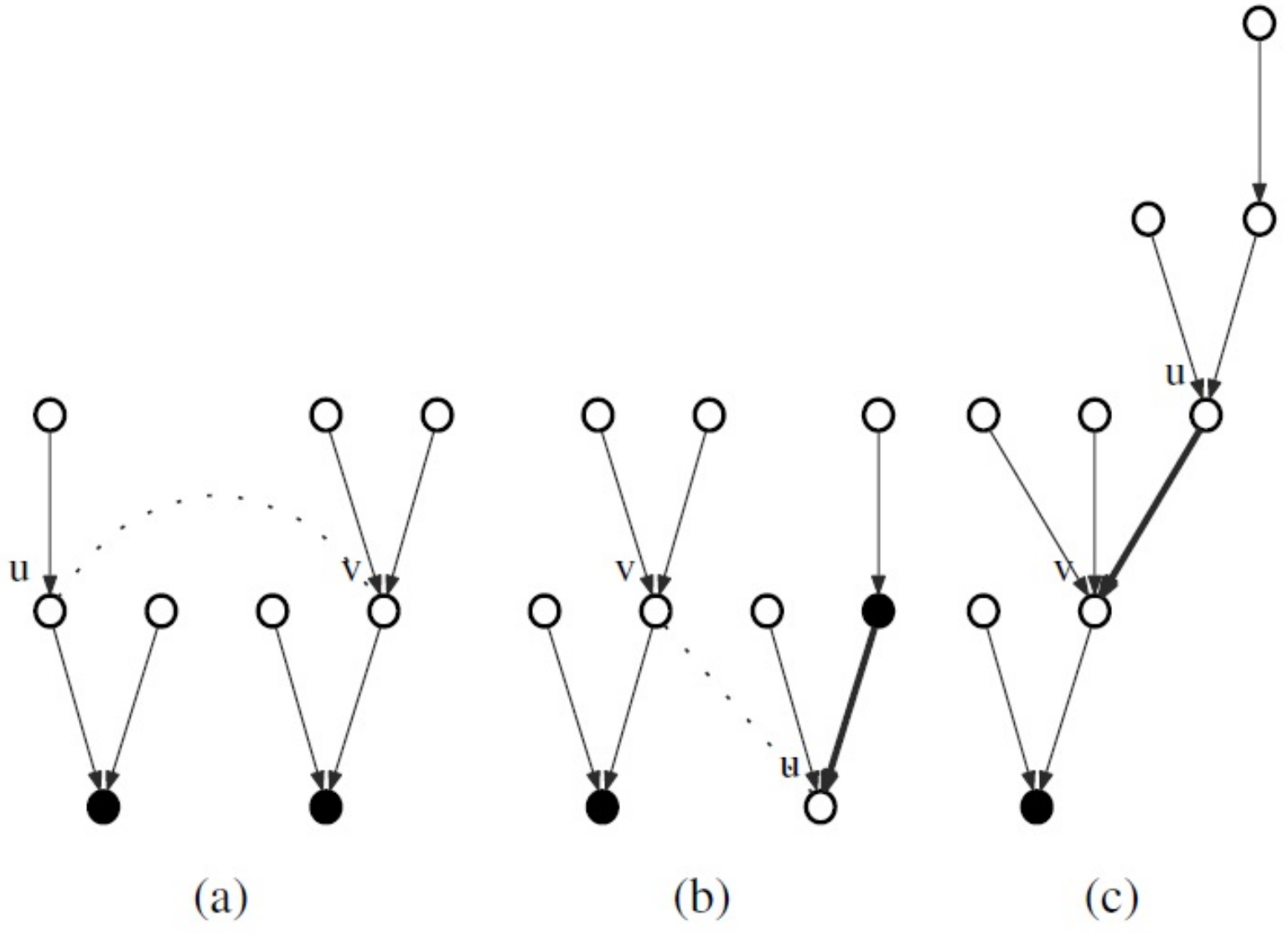
Edge Removal

- Update IN and OUT
- Only update the forest if edge removed was in forest
  - Split into two trees

Edge Removal

- Update IN and OUT
- Only update the forest if edge removed was in forest
  - Split into two trees
  - If either is below the minimum size, search along edges to find a tree with which to merge

Merge Trees



Overview of Implementation

- Hash function implementation
- IN, OUT, and forest implementation
- Forest construction procedure
- Membership query procedure
- Dynamic edges
- **Limited dynamic nodes**

Node Addition

Node Addition

- Support for small number of additions

Node Addition

- Support for small number of additions
- Supplement hash function f with a hash table

Node Addition

- Support for small number of additions
- Supplement hash function f with a hash table
- A new k-mer is assigned the value $x+1$, where x is the max hash value before addition

Node Addition

- Support for small number of additions
- Supplement hash function f with a hash table
- A new k-mer is assigned the value $x+1$, where x is the max hash value before addition
- Add additional row to IN, OUT

Node Addition

- Support for small number of additions
- Supplement hash function f with a hash table
- A new k-mer is assigned the value $x+1$, where x is the max hash value before addition
- Add additional row to IN, OUT
- Add to the forest

Node Addition

- Support for small number of additions
- Supplement hash function f with a hash table
- A new k-mer is assigned the value $x+1$, where x is the max hash value before addition
- Add additional row to IN, OUT
- Add to the forest
  - isolated tree with one node

Node Addition

- Support for small number of additions
- Supplement hash function f with a hash table
- A new k-mer is assigned the value $x+1$, where x is the max hash value before addition
- Add additional row to IN, OUT
- Add to the forest
  - isolated tree with one node
  - k-mer is stored as a root

Node Removal

Node Removal

- Remove every incident edge sequentially

Node Removal

- Remove every incident edge sequentially
  - Carry out the forest update procedure for edge removal

Node Removal

- Remove every incident edge sequentially
  - Carry out the forest update procedure for edge removal
- Now, the isolated node forms a singleton tree in the forest

Node Removal

- Remove every incident edge sequentially
  - Carry out the forest update procedure for edge removal
- Now, the isolated node forms a singleton tree in the forest
- Unstore its k-mer from the forest

Size of Data Structure

Size of Data Structure

- IN and OUT are two matrices of size $4n$ bits holding the edge information

Size of Data Structure

- IN and OUT are two matrices of size $4n$ bits holding the edge information
- The forest is an array of size $4n$ bits plus size of map that stores the kmers of roots.

Size of Data Structure

- IN and OUT are two matrices of size $4n$ bits holding the edge information
- The forest is an array of size $4n$ bits plus size of map that stores the kmers of roots.
- By minimum size bound, root k-mers add at most n bits

Size of Data Structure

- IN and OUT are two matrices of size $4n$ bits holding the edge information
- The forest is an array of size $4n$ bits plus size of map that stores the kmers of roots.
- By minimum size bound, root k-mers add at most n bits
- Yields size: $13n$ + size needed for hash function

Size of Data Structure

- IN and OUT are two matrices of size $4n$ bits holding the edge information
- The forest is an array of size $4n$ bits plus size of map that stores the kmers of roots.
- By minimum size bound, root k-mers add at most n bits
- Yields size: $13n$ + size needed for hash function
- BBHash requires roughly $3n$ bits.

Experiments

Comparisons:

- Bloom Filter Trie (BFT)[3]
- BOSS[4]

[3] Holley, G., Wittler, R., and Stoye, J. (2015). Bloom filter trie—a data structure for pan-genome storage. In Proc. of *WABI*, pages 217–230.

[4] Bowe, A., Onodera, T., Sadakane, K., and Shibuya, T. (2012). Succinct de Bruijn graphs. In Proc. of *WABI*, pages 225–235.

Table: Sizes of each data structure in bits per node.

$N_{27\text{-mer}}$	BOSS	BFT	FDBG
93K	5.7	60.4	16.6
370K	5.5	55.0	16.5
1.5M	5.4	50.5	16.5
5.4M	5.1	48.6	16.4
18M	4.8	47.8	16.2
66M	5.1	47.0	16.2
210M	5.3	45.4	16.1
860M	5.3	44.8	16.2

Evaluated using read data from *E. coli* K-12 substr. MG1655.

Experiments

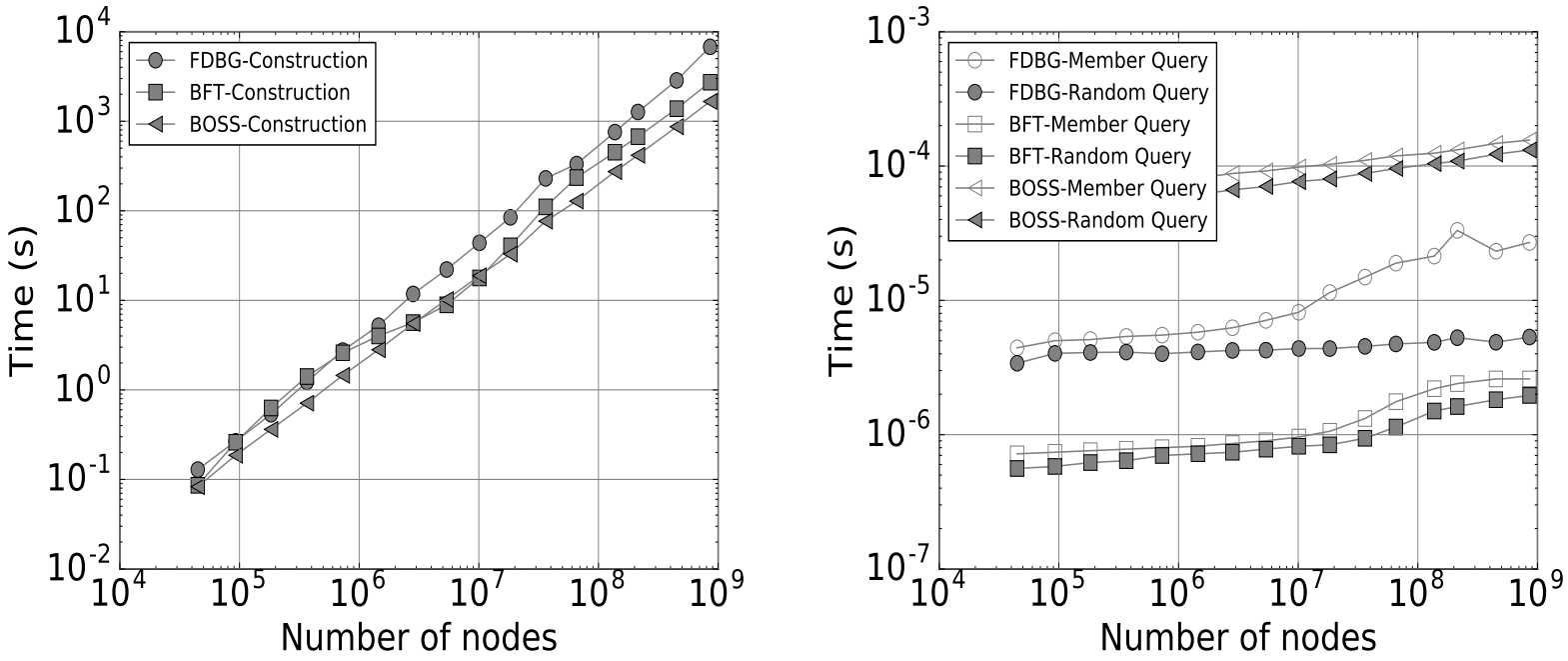


Figure: **(a)**: Construction time (s). **(b)**: Mean time (s) for membership queries.

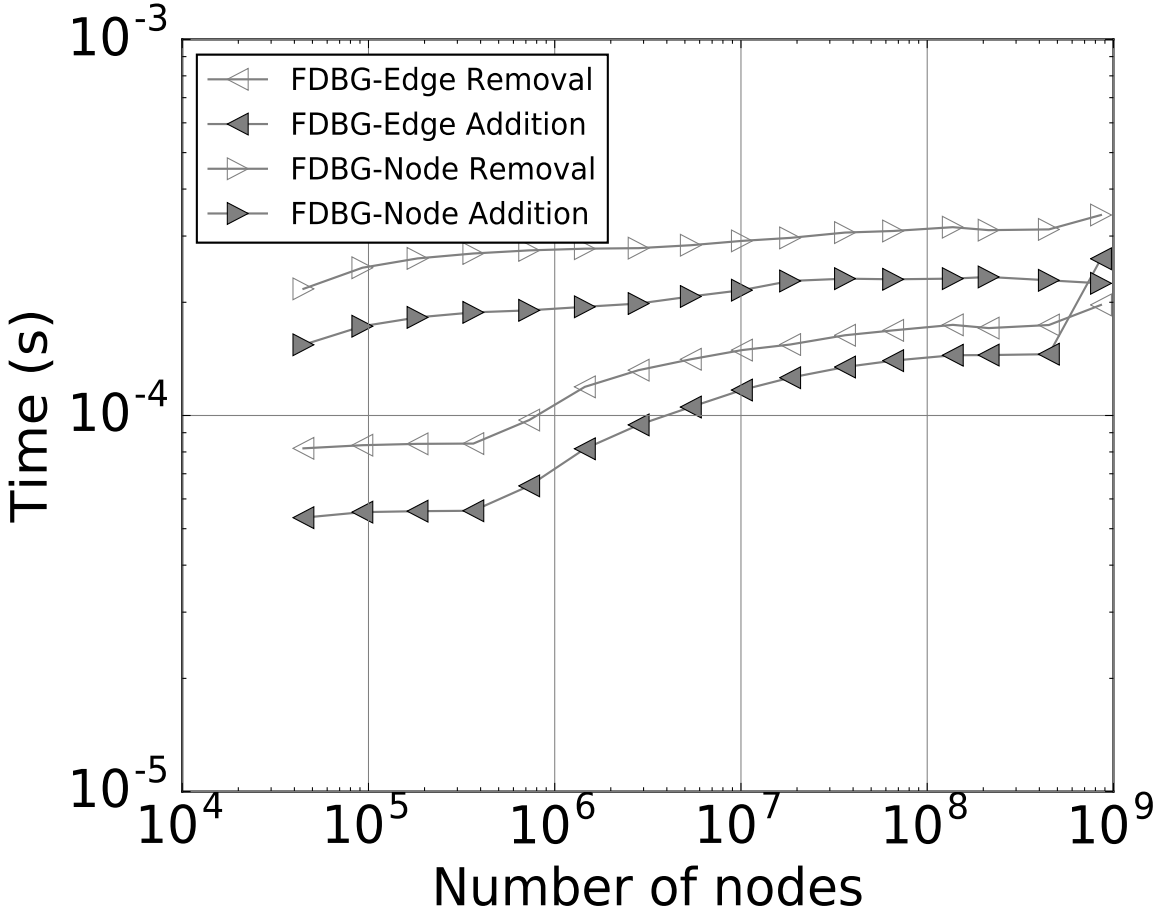


Figure: Mean time (s) for changes to the FDBG data structure.

Conclusion

- compact de Bruijn graph implementation, requires roughly 16 bits per node
- Implementation available at: `https://github.com/csirac/dynamicDBG`
- competitive with BOSS, BFT in terms of size, query times
- supports fully dynamic edge operations, limited number of dynamic node operations

Thank you! Questions?