Fully Dynamic de Bruijn Graphs

Alan Kuhnle , Victoria Crawford, Yuanpu Xie , Zizhao Zhang, Ke Bo

April 10, 2017

1. Introduction
2. Data Structure
3. Implementation
4. Experiments

Genome assembly:De-Bruijn graph

De-Bruijn graph:Reconstructing a string from a set of its k-mers

1. Data structure method on genome assembly.
2. Consist of multiple K-mers which generated by genome sequence.
3. Same vertices, K-1 mers are glue together in the final step.

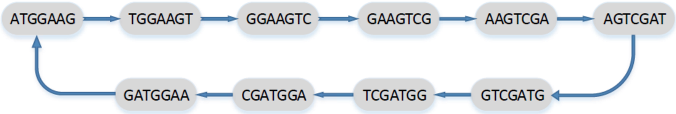


Fig. 1. The de Bruijn graph constructed from string ATGGAAGTCGATGGAAG, with $k = 7$.

Advantages

- Next-generation sequencing (NGS) data usually comes with large volume and short size, which has large amount of repetitive regions.

Advantages

- Next-generation sequencing (NGS) data usually comes with large volume and short size, which has large amount of repetitive regions.
- Compared to 'overlap-consensus-layout' method,De Bruijn graph-based assembly approach handles the assembly of repetitive regions better

Application problem

- Using de Bruijn graph in practice is the high memory occupation for certain organisms
- Human genome encoded in a de Bruijn graph with a k-mer size of 27 requires 15GB to store the node sequences
- Bulges and whirls occur because of sequencing errors or repeats in the genome, so would like to be able to efficiently add and remove edges from graph

How to efficiently update graph while maintaining memory space efficiency?

Fully Dynamic de Bruijn Graphs(Belazzougui et al. (2016))

Introduces compact, dynamic representation of De Bruijn graph

- Nodes and edges can be inserted and deleted efficiently
- k-mers are represented by integers using a combination of Karp-Rabin hashing and minimal perfect hashing.
- A partition of the graph into a forest allows efficient membership queries with no error.

Project

1. Implement the data structure from the paper.
2. Evaluate our data structure on graphs built from real sequencing data.
3. Compare our data structure with alternative approaches for De Bruijn graphs

Overview of Data Structure

- Static hash function

Overview of Data Structure

- Static hash function
- “Dynamic hash function”

Overview of Data Structure

- Static hash function
- “Dynamic hash function”
- IN, OUT matrices for storing the graph edges

Overview of Data Structure

- Static hash function
- “Dynamic hash function”
- IN, OUT matrices for storing the graph edges
- Membership query

Overview of Data Structure

- Static hash function
- “Dynamic hash function”
- IN, OUT matrices for storing the graph edges
- Membership query
- De Bruijn graph and forest

Hash function f

1. The hashing function we use is the combination of **Karp-Rabin** and **minimal perfect hashing**.

Hash function f

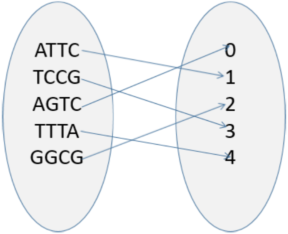
1. The hashing function we use is the combination of **Karp-Rabin** and **minimal perfect hashing**.
2. **Karp-Rabin**: Given a prime P and base r, a Rabin-Karp hash function f is a function defined over the space of all strings of length k such that $f(x_1 ... x_k) = (\sum_{i=1}^{n} x_i r^i) \bmod P$.

Hash function f

1. The hashing function we use is the combination of **Karp-Rabin** and **minimal perfect hashing**.
2. **Karp-Rabin**: 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 $f(x_1 \dots x_k) = (\sum_{i=1}^{n} x_i r^i) \mathrm{mod} P$.
3. **Minimal perfect hashing**: A minimal perfect hash function f for S is a function defined on the universe such that f is one-to-one on S and the range is $\{0, \dots, n-1\}$.

Hash function f

1. The hashing function we use is the combination of **Karp-Rabin** and **minimal perfect hashing**.
2. **Karp-Rabin**: 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 $f(x_1 \ldots x_k) = (\sum_{i=1}^{n} x_i r^i) \bmod P$.
3. **Minimal perfect hashing**: A minimal perfect hash function f for S is a function defined on the universe such that f is one-to-one on S and the range is $\{0, \ldots, n-1\}$.



Hash function properties

Lemma

Given a static set N of n k-tuples over an alphabet Σ of size σ, with high probability in $O(kn)$ expected time we can build a function $f : \Sigma^k \longrightarrow \{0, \cdots, n-1\}$ with the following properties:

1. when its domain is restricted to N, f is bijective.

Hash function properties

Lemma

Given a static set N of n k-tuples over an alphabet Σ of size σ, with high probability in $O(kn)$ expected time we can build a function $f : \Sigma^k \longrightarrow \{0, \cdots, n-1\}$ with the following properties:

1. when its domain is restricted to N, f is bijective.
2. we can store f in $O(n + logk + log\sigma)$

Hash function properties

Lemma

Given a static set N of n k-tuples over an alphabet Σ of size σ, with high probability in $O(kn)$ expected time we can build a function $f : \Sigma^k \longrightarrow \{0, \cdots, n-1\}$ with the following properties:

1. when its domain is restricted to N, f is bijective.
2. we can store f in $O(n + logk + log\sigma)$
3. given a k-tuple v, we can compute $f(v)$ in $O(k)$ time.

Hash function properties

Lemma

Given a static set N of n k-tuples over an alphabet Σ of size σ, with high probability in $O(kn)$ expected time we can build a function $f : \Sigma^k \longrightarrow \{0, \cdots, n-1\}$ with the following properties:

1. when its domain is restricted to N, f is bijective.
2. we can store f in $O(n + logk + log\sigma)$
3. given a k-tuple v, we can compute $f(v)$ in $O(k)$ time.
4. given u and v, such that suffix of u of length k-1 is the prefix of v of length, or vice versa, we can compute $f(v)$ in $O(1)$ time if we already computed $f(u)$.

Example of the hash function

- A string of nucleotides has an alphabet of size 4, so we can look at that string as a number written in base 4. For example, we can denote A as 0, C as 1, G as 2, and T as 3.

Example of the hash function

- A string of nucleotides has an alphabet of size 4, so we can look at that string as a number written in base 4. For example, we can denote A as 0, C as 1, G as 2, and T as 3.
- If we set P as 13. "ATTC" can be hashed to $(4^4 \cdot 0 + 4^3 \cdot 3 + 4^2 \cdot 3 + 4^1 \cdot 1) \mathrm{mod} 13$.

Example of the hash function

- A string of nucleotides has an alphabet of size 4, so we can look at that string as a number written in base 4. For example, we can denote A as 0, C as 1, G as 2, and T as 3.
- If we set P as 13. "ATTC" can be hashed to $(4^4 \cdot 0 + 4^3 \cdot 3 + 4^2 \cdot 3 + 4^1 \cdot 1) \mathrm{mod} 13$.
- Similarly, "TTCG" can be computed by $(4^4 \cdot 3 + 4^3 \cdot 3 + 4^2 \cdot 1 + 4^1 \cdot 2) \mathrm{mod} 13$.

Example of the hash function

- A string of nucleotides has an alphabet of size 4, so we can look at that string as a number written in base 4. For example, we can denote A as 0, C as 1, G as 2, and T as 3.
- If we set P as 13. "ATTC" can be hashed to $(4^4 \cdot 0 + 4^3 \cdot 3 + 4^2 \cdot 3 + 4^1 \cdot 1) \text{mod} 13$.
- Similarly, "TTCG" can be computed by $(4^4 \cdot 3 + 4^3 \cdot 3 + 4^2 \cdot 1 + 4^1 \cdot 2) \text{mod} 13$.
- Suppose we only have two k-mers, one qualified minimal perfect hashing function f needs to ensure that $f("ATTC") = 0$ and $f("TTCG") = 1$
  or
  $f("ATTC") = 1$ and $f("TTCG") = 0$.

Dynamic hash function

Lemma

If N is dynamic then we can maintain a function f as described in 1 except that:

1. the range of f becomes $\{0, \cdots, 3n-1\}$.

Dynamic hash function

Lemma

If N is dynamic then we can maintain a function f as described in 1 except that:

1. the range of f becomes $\{0, \cdots, 3n - 1\}$.
2. when its domain is restricted to N, f is injective.

Dynamic hash function

Lemma

If N is dynamic then we can maintain a function f as described in 1 except that:

1. the range of f becomes $\{0, \cdots, 3n - 1\}$.
2. when its domain is restricted to N, f is injective.
3. the space bound for f is $O(n(loglogn + loglog\sigma))$ bits with high probability.

Dynamic hash function

Lemma

If N is dynamic then we can maintain a function f as described in 1 except that:

1. the range of f becomes $\{0, \cdots, 3n-1\}$.
2. when its domain is restricted to N, f is injective.
3. the space bound for f is $O(n(loglogn + loglog\sigma))$ bits with high probability.
4. insertions and deletions take $O(k)$ amortized expected time.

Dynamic hash function

Lemma

If N is dynamic then we can maintain a function f as described in 1 except that:

1. the range of f becomes $\{0, \cdots, 3n-1\}$.
2. when its domain is restricted to N, f is injective.
3. the space bound for f is $O(n(loglogn + loglog\sigma))$ bits with high probability.
4. insertions and deletions take $O(k)$ amortized expected time.
5. the data structure may work incorrectly with very low probability (inverse polynomial in n).

Representation of edges in de Bruijn graph

1. The edges (E) of G are stored in two binary matrices, IN and OUT, each of size $n \times |\Sigma|$.
2. These two matrices are used to maintained the IN and OUT edge of each vertex. We can move each vertex forward and backward using this information.

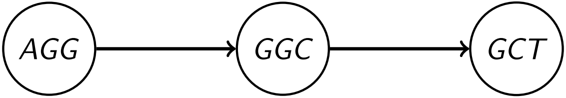
Representation of edges in de Bruijn graph

1. The edges (E) of G are stored in two binary matrices, IN and OUT, each of size $n \times |\Sigma|$.
2. These two matrices are used to maintained the IN and OUT edge of each vertex. We can move each vertex forward and backward using this information.
3. The IN and OUT matrices can be constructed as:

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

Example of *IN* and *OUT* matrices

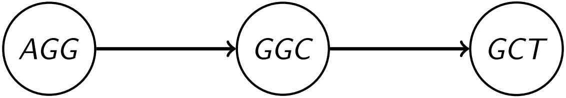
Here is a simple de bruijin graph:



(a)

Example of *IN* and *OUT* matrices

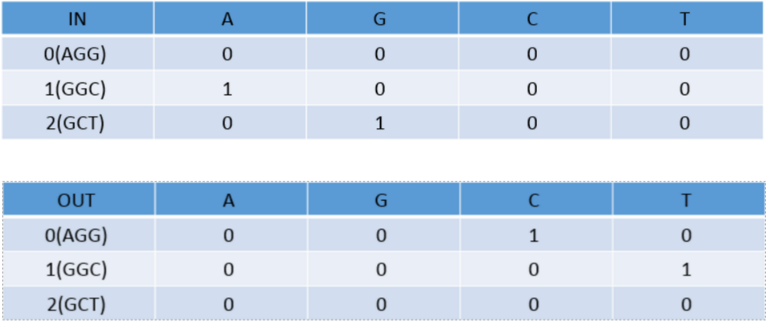
Here is a simple de bruijin graph:



(b)

Suppose $f(AGG) = 0, f(GGC) = 1, f(GCT) = 2$, the *IN* and *OUT* matrices can be initialized as:

Example of *IN* and *OUT* matrices



IN	A	G	C	T
0(AGG)	0	0	0	0
1(GGC)	1	0	0	0
2(GCT)	0	1	0	0

OUT	A	G	C	T
0(AGG)	0	0	1	0
1(GGC)	0	0	0	1
2(GCT)	0	0	0	0

(a)

Efficient membership query

1. Since the total graph nodes are only represented by integers, no k-mer is stored.

Efficient membership query

1. Since the total graph nodes are only represented by integers, no k-mer is stored.
2. How can we do membership queries maintaining a reasonable memory usage?

Efficient membership query

1. Since the total graph nodes are only represented by integers, no k-mer is stored.
2. How can we do membership queries maintaining a reasonable memory usage?
3. Our strategy is to sample a subset of nodes for which we store the plain-text k-tuple and connect all the un-sampled nodes to the sampled ones.

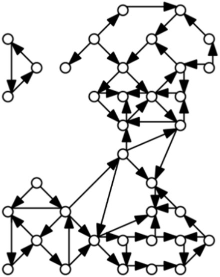
Efficient membership query

1. Since the total graph nodes are only represented by integers, no k-mer is stored.
2. How can we do membership queries maintaining a reasonable memory usage?
3. Our strategy is to sample a subset of nodes for which we store the plain-text k-tuple and connect all the un-sampled nodes to the sampled ones.
4. Given a start point, we can move forward and backward using *IN* and *OUT* matrices. Once we reached the root, we can check if the resulting k-mer matches with root k-mer.

De Bruijn graph and Forest

De Bruijn graph and Forest

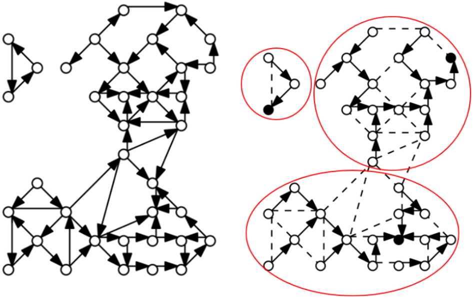
1. More specifically, partition an undirected graph G into a forest $\mathcal{F}$ where each $T \in \mathcal{F}$ with $\alpha \leq h(T) \leq 3\alpha$, where $h(T)$ is the height of tree T, $\alpha = k \log \sigma$.



De bruijn graph

De Bruijn graph and Forest

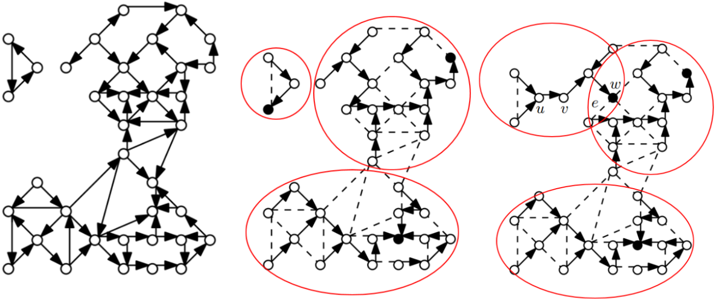
1. More specifically, partition an undirected graph G into a forest $\mathcal{F}$ where each $T \in \mathcal{F}$ with $\alpha \leq h(T) \leq 3\alpha$, where $h(T)$ is the height of tree T, $\alpha = k \log \sigma$.



De bruijn graph   Constructed forest

De Bruijn graph and Forest

1. More specifically, partition an undirected graph G into a forest $\mathcal{F}$ where each $T \in \mathcal{F}$ with $\alpha \leq h(T) \leq 3\alpha$, where $h(T)$ is the height of tree T, $\alpha = k \log \sigma$.



De bruijn graph   Constructed forest   Adding an edge

De Bruijn graph and forests properties

1. Given a static kth-order de Bruijn graph G with n nodes,

De Bruijn graph and forests properties

1. Given a static kth-order de Bruijn graph G with n nodes,
2. we can store G in $O(\sigma n)$ bits plus $O(klog\sigma)$ bits for each connected component in the underlying undirected graph, such that:

De Bruijn graph and forests properties

1. Given a static kth-order de Bruijn graph G with n nodes,
2. we can store G in $O(\sigma n)$ bits plus $O(klog\sigma)$ bits for each connected component in the underlying undirected graph, such that:
3. checking whether a node is in G takes $O(klog\sigma)$ time.

De Bruijn graph and forests properties

1. Given a static kth-order de Bruijn graph G with n nodes,
2. we can store G in $O(\sigma n)$ bits plus $O(klog\sigma)$ bits for each connected component in the underlying undirected graph, such that:
3. checking whether a node is in G takes $O(klog\sigma)$ time.
4. listing the edges incident to a node we are visiting takes $O(\sigma)$ time, and crossing an edge takes $O(1)$ time.

Overview of Implementation

- K-mer representation
- Hash function implementation
- the `BitArray` class
- IN, OUT, and forest implementation
- Forest construction procedure
- Membership query procedure
- Dynamic edges (Partially complete)

Overview of Implementation

- K-mer representation
- Hash function implementation
- the `BitArray` class
- IN, OUT, and forest implementation
- Forest construction procedure
- Membership query procedure
- Dynamic edges (Partially complete)
- Dynamic nodes (May not get to this)

Overview of Implementation

- K-mer representation
- Hash function implementation
- the `BitArray` class
- IN, OUT, and forest implementation
- Forest construction procedure
- Membership query procedure
- Dynamic edges (Partially complete)
- Dynamic nodes (May not get to this)
- “Semi-dynamic De Bruijn Graph”

K-mer representation

- Our program takes as input a fasta file and a kmer size, and constructs the data structure

---

[1] https://github.com/Kingsford-Group/kbf

K-mer representation

- Our program takes as input a fasta file and a kmer size, and constructs the data structure
- We use KBF[1] library in order to read in all kmers of length K and K+1

---

[1] https://github.com/Kingsford-Group/kbf

K-mer representation

- Our program takes as input a fasta file and a kmer size, and constructs the data structure
- We use KBF[1] library in order to read in all kmers of length K and K+1
- Each kmer is represented as an 64 bit integer where pairs of consecutive bits represent letters $A = 00$, $C = 01$, $G = 10$, and $T = 11$.

---

[1] https://github.com/Kingsford-Group/kbf

K-mer representation

- Our program takes as input a fasta file and a kmer size, and constructs the data structure
- We use KBF[1] library in order to read in all kmers of length K and K+1
- Each kmer is represented as an 64 bit integer where pairs of consecutive bits represent letters $A = 00$, $C = 01$, $G = 10$, and $T = 11$.
- Example:

$$\underbrace{\overbrace{0000\cdots}^{\text{Zeros}}\overbrace{11100100}^{\text{2K bits}}}_{\text{64 bits}} = TGCA$$

---

[1] https://github.com/Kingsford-Group/kbf

Building the Hash Function

- Recall: hash function f maps K-mer m to $\{0, ..., n-1\}$ where n is the number of kmers in the graph.

Building the Hash Function

- Recall: hash function f maps K-mer m to $\{0, \ldots, n-1\}$ where n is the number of kmers in the graph.
- Recall: the hash function is a minimal perfect hash function composed with a Karp-Rabin hash function.

Building the Hash Function

- Recall: hash function f maps K-mer m to $\{0, ..., n-1\}$ where n is the number of kmers in the graph.
- Recall: the hash function is a minimal perfect hash function composed with a Karp-Rabin hash function.
- How do we construct the Karp-Rabin hash function?

Building the Karp-Rabin Hash Function

- For the prime P, pick the smallest prime greater than Kn^2.

---

[2] http://www.boost.org/

Building the Karp-Rabin Hash Function

- For the prime P, pick the smallest prime greater than Kn^2.
- Pick a random number r in $\{1, \ldots, P\}$ for the base

---

[2] http://www.boost.org/

Building the Karp-Rabin Hash Function

- For the prime P, pick the smallest prime greater than Kn^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

---

[2] http://www.boost.org/

Building the Karp-Rabin Hash Function

- For the prime P, pick the smallest prime greater than Kn^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.

---

[2] http://www.boost.org/

Building the Karp-Rabin Hash Function

- For the prime P, pick the smallest prime greater than Kn^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.
- Powers of r: $r, r^2, \ldots, r^K \mod P$ are precomputed and stored for use in Karp-Rabin computation $\rightarrow$ $O(K)$ computation time

---

[2] http://www.boost.org/

Building the Karp-Rabin Hash Function

- For the prime P, pick the smallest prime greater than Kn^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.
- Powers of r: $r, r^2, \ldots, r^K \mod P$ are precomputed and stored for use in Karp-Rabin computation → $O(K)$ computation time
- Even though P and powers can be stored (barely) in 64-bit integer, computation requires slow 128-bit arithmetic, for which we used `Boost`[2] library type.

---

[2] http://www.boost.org/

Building the Karp-Rabin Hash Function

- For the prime P, pick the smallest prime greater than Kn^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.
- Powers of r: $r, r^2, \ldots, r^K \mod P$ are precomputed and stored for use in Karp-Rabin computation $\rightarrow$ $O(K)$ computation time
- Even though P and powers can be stored (barely) in 64-bit integer, computation requires slow 128-bit arithmetic, for which we used `Boost`[2] library type.
- For $K = 27$ on *E. coli*, lower bound for prime is

  $$16584693176107222092.$$

  Max value in unsigned 64-bit integer:

  $$18446744073709551615.$$

---

[2] http://www.boost.org/

Building the Hash Function

- Recall: A minimal perfect hash function on a set A of size n is a hash function that maps elements of A injectively to the set $\{0, ..., n-1\}$.

---

[3] https://github.com/rizkg/BBHash

Building the Hash Function

- Recall: A minimal perfect hash function on a set A of size n is a hash function that maps elements of A injectively to the set $\{0, ..., n-1\}$.
- In our case, A is Karp-Rabin values

---

[3] https://github.com/rizkg/BBHash

Building the Hash Function

- Recall: A minimal perfect hash function on a set A of size n is a hash function that maps elements of A injectively to the set $\{0, \ldots, n-1\}$.
- In our case, A is Karp-Rabin values
- We use the `BBHash`[3] library to build a minimal perfect hash function on the image of our kmers under the Karp-Rabin hash.

---

[3] https://github.com/rizkg/BBHash

Building the Hash Function

- Recall: A minimal perfect hash function on a set A of size n is a hash function that maps elements of A injectively to the set $\{0, ..., n-1\}$.
- In our case, A is Karp-Rabin values
- We use the `BBHash`[3] library to build a minimal perfect hash function on the image of our kmers under the Karp-Rabin hash.
- We store our base r, the prime P, and our MPHF object from BBHash, and we can now hash any kmer to $\{0, ..., n-1\}$.

---

[3] https://github.com/rizkg/BBHash

Building the Hash Function

- Recall: A minimal perfect hash function on a set A of size n is a hash function that maps elements of A injectively to the set $\{0, ..., n-1\}$.
- In our case, A is Karp-Rabin values
- We use the `BBHash`[3] library to build a minimal perfect hash function on the image of our kmers under the Karp-Rabin hash.
- We store our base r, the prime P, and our MPHF object from BBHash, and we can now hash any kmer to $\{0, ..., n-1\}$.
- Note that the hash function can hash any kmer, but is bijective when restricted to kmers that actually exist in our De Bruijn graph.

---

[3] https://github.com/rizkg/BBHash

Building the Hash Function

- Recall: A minimal perfect hash function on a set A of size n is a hash function that maps elements of A injectively to the set $\{0, \ldots, n-1\}$.
- In our case, A is Karp-Rabin values
- We use the `BBHash`[3] library to build a minimal perfect hash function on the image of our kmers under the Karp-Rabin hash.
- We store our base r, the prime P, and our MPHF object from BBHash, and we can now hash any kmer to $\{0, \ldots, n-1\}$.
- Note that the hash function can hash any kmer, but is bijective when restricted to kmers that actually exist in our De Bruijn graph.
- That completes the generation of the hash function.

---

[3] https://github.com/rizkg/BBHash

Construction of Hash function

```
procedure GENERATEHASH
Input $S$, a set of $n$ $k$-tuples over an alphabet $\sum$ of size $\sigma$
    $R = \max(\sigma, kn^2)$
    $P = \text{getPrime}(R)$
    $r = \text{randomNumber}(0, P - 1)$
    $f = \text{rabinHash}(r, P)$
    while isInjective($f$, $S$) is FALSE do
        $r = \text{randomNumber}(0, P - 1)$
        $f = \text{rabinHash}(r, P)$
    end while
    $g = \text{minimalPerfectHash}(f(S))$ ;
    return $g \circ f$ ;
end procedure
```

Updating a Karp-Rabin value

- If we have a K-mer m, and its (Karp-Rabin) hash value $f(m)$, suppose we want to move to a neighbor K-mer n and get its hash value $f(n)$;

Updating a Karp-Rabin value

- If we have a K-mer m, and its (Karp-Rabin) hash value $f(m)$, suppose we want to move to a neighbor K-mer n and get its hash value $f(n)$;
- We can update the KR value in $O(1)$ rather than recomputing from scratch in $O(K)$.

Updating a Karp-Rabin value

- If we have a K-mer m, and its (Karp-Rabin) hash value $f(m)$, suppose we want to move to a neighbor K-mer n and get its hash value $f(n)$;
- We can update the KR value in $O(1)$ rather than recomputing from scratch in $O(K)$.
- For example, if n is an OUT-neighbor with letter *last*, and m starts with letter *first*:

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

Updating a Karp-Rabin value

- If we have a K-mer m, and its (Karp-Rabin) hash value $f(m)$, suppose we want to move to a neighbor K-mer n and get its hash value $f(n)$;
- We can update the KR value in $O(1)$ rather than recomputing from scratch in $O(K)$.
- For example, if n is an OUT-neighbor with letter *last*, and m starts with letter *first*:

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

- What is the problem with naively implementing this update?

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

- First problem is that since $f(m)$ was computed mod P, first term might be negative.

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

- First problem is that since $f(m)$ was computed mod P, first term might be negative.
- Second problem is we can't use the ordinary division algorithm modulo P.

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

- First problem is that since $f(m)$ was computed mod P, first term might be negative.
- Second problem is we can't use the ordinary division algorithm modulo P.
- Solution to second problem: precompute $r^{-1} \mod P$ and store it. (Requires generalized Euclidean algorithm)

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

- First problem is that since $f(m)$ was computed mod P, first term might be negative.
- Second problem is we can't use the ordinary division algorithm modulo P.
- Solution to second problem: precompute $r^{-1} \mod P$ and store it. (Requires generalized Euclidean algorithm)
- Solution to first problem? Hint: What is $P \mod P$?

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

- First problem is that since $f(m)$ was computed mod P, first term might be negative.
- Second problem is we can't use the ordinary division algorithm modulo P.
- Solution to second problem: precompute $r^{-1} \mod P$ and store it. (Requires generalized Euclidean algorithm)
- Solution to first problem? Hint: What is $P \mod P$?
- Add $(4P - \mathit{first} \cdot r)$ to $f(m)$, as this will always be nonnegative.

The BitArray Class

- How to store bits compactly?

The BitArray Class

- How to store bits compactly?
- In C++, `bool` type takes at least 1 byte.

The BitArray Class

- How to store bits compactly?
- In C++, `bool` type takes at least 1 byte.
- Therefore, an array of `bool`s wastes 7 unused bits for each stored bit.

The BitArray Class

- How to store bits compactly?
- In C++, `bool` type takes at least 1 byte.
- Therefore, an array of `bool`s wastes 7 unused bits for each stored bit.
- In order to make our data structure as compact as possible, we implemented a `BitArray` class that is used in multiple places.

The BitArray Class

- How to store bits compactly?
- In C++, `bool` type takes at least 1 byte.
- Therefore, an array of `bool`s wastes 7 unused bits for each stored bit.
- In order to make our data structure as compact as possible, we implemented a `BitArray` class that is used in multiple places.
- A BitArray essentially is an array of 32-bit `int`s where we access individual bits of data by bitwise operations.

The BitArray Class

- How to store bits compactly?
- In C++, `bool` type takes at least 1 byte.
- Therefore, an array of `bool`s wastes 7 unused bits for each stored bit.
- In order to make our data structure as compact as possible, we implemented a `BitArray` class that is used in multiple places.
- A BitArray essentially is an array of 32-bit `int`s where we access individual bits of data by bitwise operations.
- Each bit of an integer is treated as an element in the array and can be set, returned as a bool, etc..

The BitArray Class

- How to store bits compactly?
- In C++, `bool` type takes at least 1 byte.
- Therefore, an array of `bool`s wastes 7 unused bits for each stored bit.
- In order to make our data structure as compact as possible, we implemented a `BitArray` class that is used in multiple places.
- A BitArray essentially is an array of 32-bit `int`s where we access individual bits of data by bitwise operations.
- Each bit of an integer is treated as an element in the array and can be set, returned as a bool, etc..
- This will slow data access down since multiple operations are required for each access, but much more memory efficient.

The BitArray Class

- Example with two `ints` (each containing 32 bits).

$$\underbrace{010\cdots}_{\text{int } 0}\overbrace{101\cdots}^{\text{int } 1}$$

The BitArray Class

- Example with two `ints` (each containing 32 bits).

$$\underbrace{010\cdots}_{\text{int 0}}\overbrace{101\cdots}^{\text{int 1}}$$

- `access`(33):

The BitArray Class

- Example with two `ints` (each containing 32 bits).

$$\underbrace{010\cdots}_{\text{int } 0}\overbrace{101\cdots}^{\text{int } 1}$$

- `access`(33):
  - First computes `int` index: $33/32 = 1$

The BitArray Class

- Example with two `ints` (each containing 32 bits).

$$\underbrace{010\cdots}_{\text{int } 0}\overbrace{101\cdots}^{\text{int } 1}$$

- `access`(33):
    - First computes `int` index: $33/32 = 1$
    - Next compute which bit in this `int`: $33 = 1(mod 32)$

The BitArray Class

- Example with two `int`s (each containing 32 bits).

$$\underbrace{010\cdots}_{\text{int } 0}\overbrace{101\cdots}^{\text{int } 1}$$

- `access`(33):
    - First computes `int` index: $33/32 = 1$
    - Next compute which bit in this `int`: $33 = 1(mod 32)$
    - To get value of this bit, shift it to right-most position and do bitwise AND with $00\ldots1$.

The BitArray Class

- Example with two `ints` (each containing 32 bits).

$$\underbrace{010\cdots}_{\text{int }0}\overbrace{101\cdots}^{\text{int }1}$$

- `access`(33):
  - First computes `int` index: $33/32 = 1$
  - Next compute which bit in this `int`: $33 = 1(mod 32)$
  - To get value of this bit, shift it to right-most position and do bitwise AND with $00\ldots1$.
- With this class, only waste at most 31 bits in memory per instance of the class.

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.
- We store all entries in single `BitArray` of size $n \cdot \sigma$.

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.
- We store all entries in single `BitArray` of size $n \cdot \sigma$.
- So we guarantee to use only $n\sigma + 31$ bits of memory for each of IN,OUT.

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.
- We store all entries in single `BitArray` of size $n \cdot \sigma$.
- So we guarantee to use only $n\sigma + 31$ bits of memory for each of IN,OUT.
- To construct, simply read through edge $k + 1$-mers and set the correct index of each of IN, OUT.

The Forest Data Structure

- The forest is stored as a `BitArray` and a map of root hash values to their kmers.

The Forest Data Structure

- The forest is stored as a `BitArray` and a map of root hash values to their kmers.
- Each node in the graph has 4 consecutive bits in the bit array.

The Forest Data Structure

- The forest is stored as a `BitArray` and a map of root hash values to their kmers.
- Each node in the graph has 4 consecutive bits in the bit array.
- The nodes are in the `BitArray` in order of their hash values.

The Forest Data Structure

- The forest is stored as a `BitArray` and a map of root hash values to their kmers.
- Each node in the graph has 4 consecutive bits in the bit array.
- The nodes are in the `BitArray` in order of their hash values.
- For each node, the first bit tells if that node is a root or not, the second tells whether the parent in the forest is accessed via IN or OUT, and the last two tells the letter that one needs to get to the parent.

The Forest Data Structure

- The forest is stored as a `BitArray` and a map of root hash values to their kmers.
- Each node in the graph has 4 consecutive bits in the bit array.
- The nodes are in the `BitArray` in order of their hash values.
- For each node, the first bit tells if that node is a root or not, the second tells whether the parent in the forest is accessed via IN or OUT, and the last two tells the letter that one needs to get to the parent.
- 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 as a `BitArray` and a map of root hash values to their kmers.
- Each node in the graph has 4 consecutive bits in the bit array.
- The nodes are in the `BitArray` in order of their hash values.
- For each node, the first bit tells if that node is a root or not, the second tells whether the parent in the forest is accessed via IN or OUT, and the last two tells the letter that one needs to get to the parent.
- 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

- Using the data for the forest, and an initial kmer sequence, we can traverse the forest from any node up to its root.

The Forest Data Structure

- Using the data for the forest, and an initial kmer sequence, we can traverse the forest from any node up to its root.
- For example, suppose we have initial kmer "ATTGA", we hash it and find data 0011 in the forest for this kmer.

The Forest Data Structure

- Using the data for the forest, and an initial kmer sequence, we can traverse the forest from any node up to its root.
- For example, suppose we have initial kmer "ATTGA", we hash it and find data 0011 in the forest for this kmer.
- So that means our kmer is not a root (so a parent exists), its parent is accessed via an OUT edge, and the letter "T" is how we get to the parent.

The Forest Data Structure

- Using the data for the forest, and an initial kmer sequence, we can traverse the forest from any node up to its root.
- For example, suppose we have initial kmer "ATTGA", we hash it and find data 0011 in the forest for this kmer.
- So that means our kmer is not a root (so a parent exists), its parent is accessed via an OUT edge, and the letter "T" is how we get to the parent.
- Therefore the parent's kmer is "TTGAT"

The Forest Data Structure

- Using the data for the forest, and an initial kmer sequence, we can traverse the forest from any node up to its root.
- For example, suppose we have initial kmer "ATTGA", we hash it and find data 0011 in the forest for this kmer.
- So that means our kmer is not a root (so a parent exists), its parent is accessed via an OUT edge, and the letter "T" is how we get to the parent.
- Therefore the parent's kmer is "TTGAT"
- This is how we will do membership queries (explained more later).

Building the Forest

- That was how the forest is stored, but how do we build it?

Building the Forest

- That was how the forest is stored, but how do we build it?
- We want a forest that covers all of the nodes in our De Bruijn graph

Building the Forest

- That was how the forest is stored, but how do we build it?
- We want a forest that covers all of the nodes in our De Bruijn graph
- Each tree should be between height α and 3α where $\alpha = k \log \sigma$.

Building the Forest

- That was how the forest is stored, but how do we build it?
- We want a forest that covers all of the nodes in our De Bruijn graph
- Each tree should be between height α and 3α where $\alpha = k \log \sigma$.
- Each tree has a root, and the kmer of that root is stored.

Building the Forest

- That was how the forest is stored, but how do we build it?
- We want a forest that covers all of the nodes in our De Bruijn graph
- Each tree should be between height α and 3α where $\alpha = k \log \sigma$.
- Each tree has a root, and the kmer of that root is stored.
- We do a breadth first search of the De Bruijn graph ignoring edge directions.

Building the Forest

- That was how the forest is stored, but how do we build it?
- We want a forest that covers all of the nodes in our De Bruijn graph
- Each tree should be between height α and 3α where $\alpha = k \log \sigma$.
- Each tree has a root, and the kmer of that root is stored.
- We do a breadth first search of the De Bruijn graph ignoring edge directions.
- We break the graph up into trees in the desired height range as we go along.

Building the Forest

Before this point, the forest has been initialized to be a `BitArray` of the correct size and an empty map of root kmers.

Building the Forest

Before this point, the forest has been initialized to be a `BitArray` of the correct size and an empty map of root kmers.

- First, we choose a kmer in our De Bruijn graph that has not yet been explored.

Building the Forest

Before this point, the forest has been initialized to be a `BitArray` of the correct size and an empty map of root kmers.

- First, we choose a kmer in our De Bruijn graph that has not yet been explored.
- Hash it to find its place in the forest's `BitArray`.

Building the Forest

Before this point, the forest has been initialized to be a `BitArray` of the correct size and an empty map of root kmers.

- First, we choose a kmer in our De Bruijn graph that has not yet been explored.
- Hash it to find its place in the forest's `BitArray`.
- Store it as being a root, its IN/OUT and parent bits are left alone since it has no parent.

Building the Forest

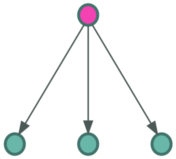
Before this point, the forest has been initialized to be a `BitArray` of the correct size and an empty map of root kmers.

- First, we choose a kmer in our De Bruijn graph that has not yet been explored.
- Hash it to find its place in the forest's `BitArray`.
- Store it as being a root, its IN/OUT and parent bits are left alone since it has no parent.
- Add its hash and kmer to the map.

Building the Forest

De Bruijn breadth first search

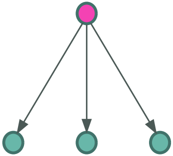
- We
  use IN and OUT to find
  all the kmers of neighbors
  in the De Bruijn graph.



Building the Forest

De Bruijn breadth first search

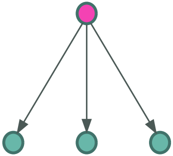
- We use IN and OUT to find all the kmers of neighbors in the De Bruijn graph.
- Get hash values of neighbor kmers, find their places in the forest's `BitArray`.



Building the Forest

De Bruijn breadth first search

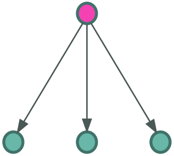
- We use IN and OUT to find all the kmers of neighbors in the De Bruijn graph.
- Get hash values of neighbor kmers, find their places in the forest's `BitArray`.
- Store the letter and IN/OUT data to get to the parent (pink).



Building the Forest

De Bruijn breadth first search

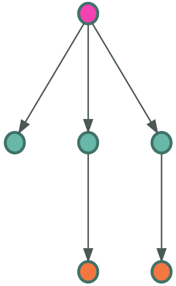
- We use IN and OUT to find all the kmers of neighbors in the De Bruijn graph.
- Get hash values of neighbor kmers, find their places in the forest's `BitArray`.
- Store the letter and IN/OUT data to get to the parent (pink).
- Set the root bit to false, and don't store these kmers.



Building the Forest

De Bruijn breadth first search

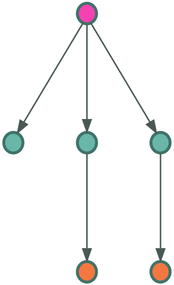
- Keep doing that until we get to a kmer that is of height α+1 from the root (orange)



Building the Forest

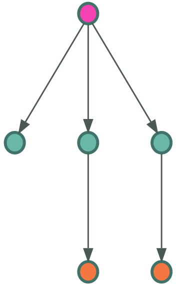
De Bruijn breadth first search

- Keep doing that until we get to a kmer that is of height $\alpha+1$ from the root (orange)
- Save the kmers of these, but don't store them in the forest just yet.



Building the Forest

De Bruijn breadth first search

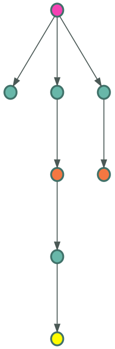


- Keep doing that until we get to a kmer that is of height α+1 from the root (orange)
- Save the kmers of these, but don't store them in the forest just yet.
- If we get to a height over the maximum allowed, we can break off at this root and be sure the remaining tree's height is still above the minimum allowed.

Building the Forest

De Bruijn breadth first search

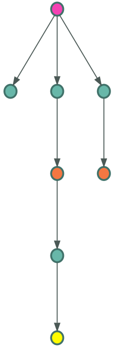
- Once we get to a kmer that is of height $3\alpha + 1$ from the root (yellow), we need a new tree.



Building the Forest

De Bruijn breadth first search

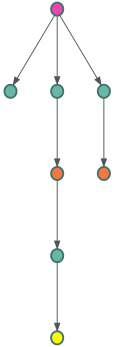
- Once we get to a kmer that is of height $3\alpha + 1$ from the root (yellow), we need a new tree.
- We have saved the kmer of the potential root found in the last part.



Building the Forest

De Bruijn breadth first search

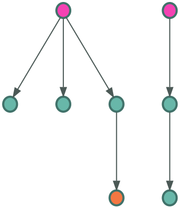
- Once we get to a kmer that is of height $3\alpha + 1$ from the root (yellow), we need a new tree.
- We have saved the kmer of the potential root found in the last part.
- We break a new tree off at the potential root …



Building the Forest

De Bruijn breadth first search

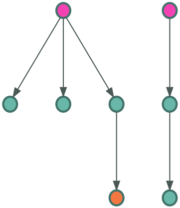
- In the forest data structure, set the potential node as a root and put its kmer in the map.



Building the Forest

De Bruijn breadth first search

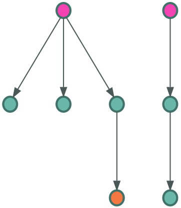
- In the forest data structure, set the potential node as a root and put its kmer in the map.
- Reset the height we are at for the new root.



Building the Forest

De Bruijn breadth first search

- In the forest data structure, set the potential node as a root and put its kmer in the map.
- Reset the height we are at for the new root.
- Continue on until the entire De Bruijn graph has been visited.



The Entire Data Structure

- We have now constructed the hash function, IN and OUT, and the forest.

The Entire Data Structure

- We have now constructed the hash function, IN and OUT, and the forest.
- That was the entire data structure, we no longer need all those kmers.

Membership Query

- This data structure allows us to do membership queries without errors.

Membership Query

- This data structure allows us to do membership queries without errors.
- Suppose we have a nucleotide sequence of length k and we want to find if it exists

Membership Query

- This data structure allows us to do membership queries without errors.
- Suppose we have a nucleotide sequence of length k and we want to find if it exists
- The only kmers that we have stored are the kmers of the roots in our tree.

Membership Query

- This data structure allows us to do membership queries without errors.
- Suppose we have a nucleotide sequence of length k and we want to find if it exists
- The only kmers that we have stored are the kmers of the roots in our tree.
- How do we check the membership of that kmer?

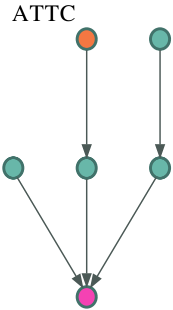
Membership Query

- This data structure allows us to do membership queries without errors.
- Suppose we have a nucleotide sequence of length k and we want to find if it exists
- The only kmers that we have stored are the kmers of the roots in our tree.
- How do we check the membership of that kmer?
- We travel up the tree to the root ...

Membership Query

Membership of "ATTC"

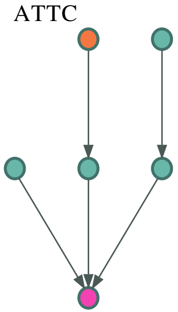
- First, hash the kmer to get i in $\{0, ..., n-1\}$



Membership Query

Membership of "ATTC"

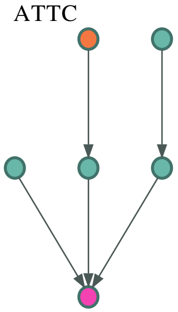
- First, hash the kmer to get i in $\{0, ..., n-1\}$
- Find the place corresponding to this hash value in the forest.



Membership Query

Membership of "ATTC"

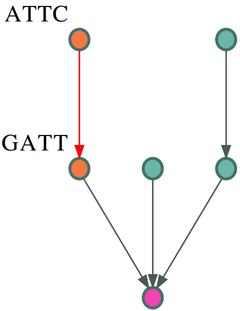
- First, hash the kmer to get i in $\{0, ..., n-1\}$
- Find the place corresponding to this hash value in the forest.
- Note:
  Even if the kmer is not in our De Bruijn graph, we can still hash it and get a place in the forest.



Membership Query

Membership of "ATTC"

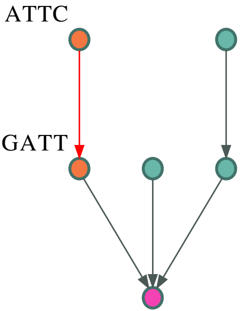
- Use the forest data and "ATTC" to do a hash update and find the parent in the forest.



Membership Query

Membership of "ATTC"

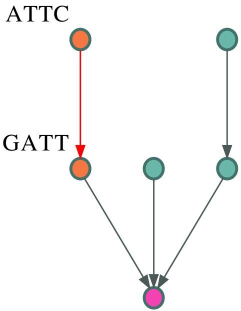
- Use the forest data and "ATTC" to do a hash update and find the parent in the forest.
- Check IN and OUT whether such an edge exists. Return false if it doesn't.



Membership Query

Membership of "ATTC"

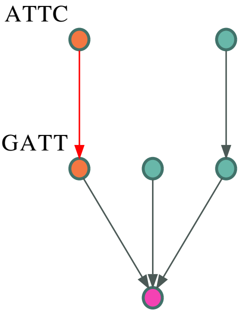
- Use the forest data and "ATTC" to do a hash update and find the parent in the forest.
- Check IN and OUT whether such an edge exists. Return false if it doesn't.
- Why would that possibly return false?



Membership Query

Membership of "ATTC"

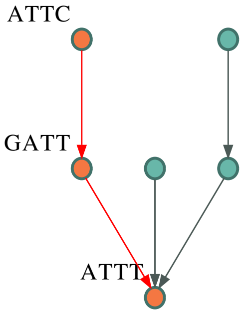
- Use the forest data and "ATTC" to do a hash update and find the parent in the forest.
- Check IN and OUT whether such an edge exists. Return false if it doesn't.
- Why would that possibly return false?
- It's possible that the hash for "GATT" doesn't have an OUT edge for "C", therefore "ATTC" can't possibly be an actual kmer.



Membership Query

Membership of "ATTC"

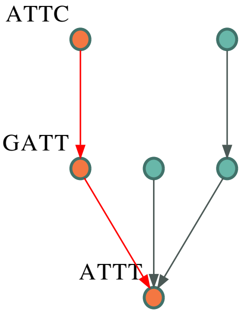
- Keep doing
  this until we arrive at a root



Membership Query

Membership of "ATTC"

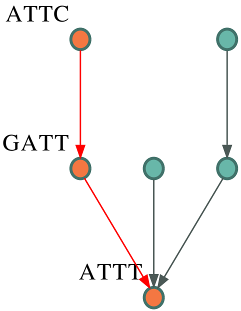
- Keep doing
  this until we arrive at a root
- The root has its kmer stored



Membership Query

Membership of "ATTC"

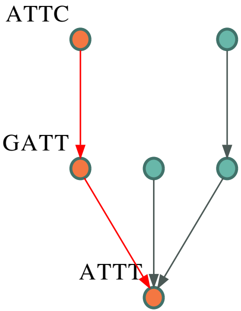
- Keep doing
  this until we arrive at a root
- The root has its kmer stored
- Compare the kmer
  we have from moving up the
  tree with the kmer that is stored



Membership Query

Membership of "ATTC"

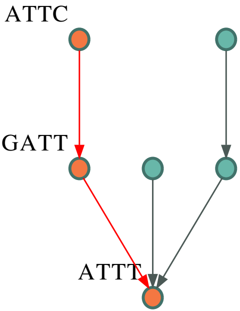
- Keep doing this until we arrive at a root
- The root has its kmer stored
- Compare the kmer we have from moving up the tree with the kmer that is stored
- If it matches, return true. Otherwise, return false.



Membership Query

Membership of "ATTC"

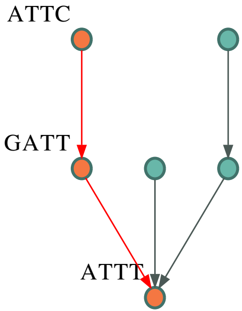
- Keep doing this until we arrive at a root
- The root has its kmer stored
- Compare the kmer we have from moving up the tree with the kmer that is stored
- If it matches, return true. Otherwise, return false.
- Which would take longer on average, membership queries that return true or queries that return false?



Membership Query

Membership of "ATTC"

- Keep doing this until we arrive at a root
- The root has its kmer stored
- Compare the kmer we have from moving up the tree with the kmer that is stored
- If it matches, return true. Otherwise, return false.
- Which would take longer on average, membership queries that return true or queries that return false?
- Queries that return true



Dynamic De Bruijn Graph

- We have (almost) implemented edge addition and removal.

Dynamic De Bruijn Graph

- We have (almost) implemented edge addition and removal.
- We have not implemented node addition and removal. The primary difficulty is having a dynamic hash function.

Dynamic De Bruijn Graph

- We have (almost) implemented edge addition and removal.
- We have not implemented node addition and removal. The primary difficulty is having a dynamic hash function.
- We rely on the library BBHash for our minimal perfect hash function, but we need a dynamic perfect hash function.

Dynamic Edges

- We will go over the case where we would like to add an edge to the De Bruijn graph.

Dynamic Edges

- We will go over the case where we would like to add an edge to the De Bruijn graph.
- First, we add an entry to IN and OUT to reflect the edge.

Dynamic Edges

- We will go over the case where we would like to add an edge to the De Bruijn graph.
- First, we add an entry to IN and OUT to reflect the edge.
- Do we do anything to the forest?

Dynamic Edges

- We will go over the case where we would like to add an edge to the De Bruijn graph.
- First, we add an entry to IN and OUT to reflect the edge.
- Do we do anything to the forest?
- This would affect the forest if the edge is between two graph components where at least one was not big enough to have a tree above the minimum height.

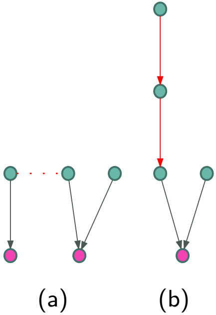
Dynamic Edges

- We will go over the case where we would like to add an edge to the De Bruijn graph.
- First, we add an entry to IN and OUT to reflect the edge.
- Do we do anything to the forest?
- This would affect the forest if the edge is between two graph components where at least one was not big enough to have a tree above the minimum height.
- We may be able to combine the trees so that we have trees with heights in the desired range.

Dynamic Edges

Adding an Edge

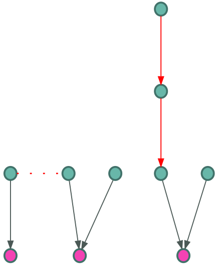
- Suppose that
the edge (dotted) is between two trees in the forest below the minimum height, shown in (a).



Dynamic Edges

Adding an Edge

- Suppose that the edge (dotted) is between two trees in the forest below the minimum height, shown in (a).
- We can combine the trees into one tree of height less than the maximum.

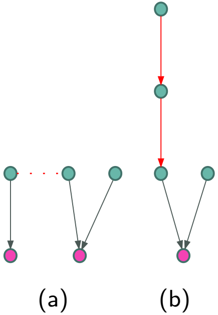


(a) (b)

Dynamic Edges

Adding an Edge

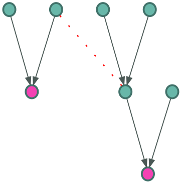
- Suppose that the edge (dotted) is between two trees in the forest below the minimum height, shown in (a).
- We can combine the trees into one tree of height less than the maximum.
- You have to add another edge to the forest, reverse the direction of some forest edges, then get rid of one of the tree's roots.



Dynamic Edges

Adding an Edge

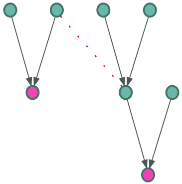
- Now suppose that
  only one of the trees are below
  the desired minimum height
  and we add an edge (dotted).



Dynamic Edges

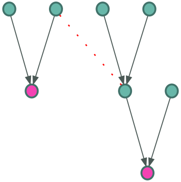
Adding an Edge

- Now suppose that
  only one of the trees are below
  the desired minimum height
  and we add an edge (dotted).
- What we do depends
  on the height of the node
  in the edge from the taller tree.



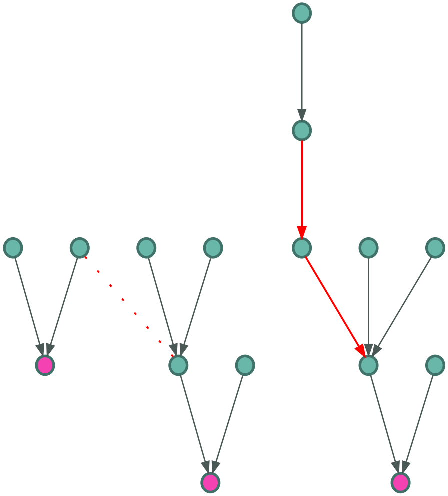
Dynamic Edges

Adding an Edge



- Now suppose that only one of the trees are below the desired minimum height and we add an edge (dotted).
- What we do depends on the height of the node in the edge from the taller tree.
- If the height is less than α, we can change the trees similar to before.

Adding Edges

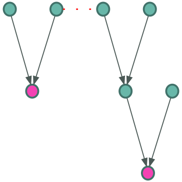


(a) Height $< \alpha$ (b)

Dynamic Edges

Adding an Edge

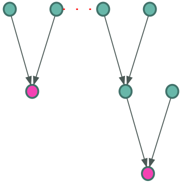
- Now suppose that the height is greater than or equal to α



Dynamic Edges

Adding an Edge

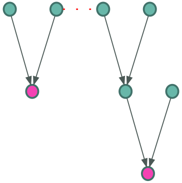
- Now suppose that the height is greater than or equal to α
- We can't necessarily just combine the trees like before, we could end up with a tree greater than the maximum height.



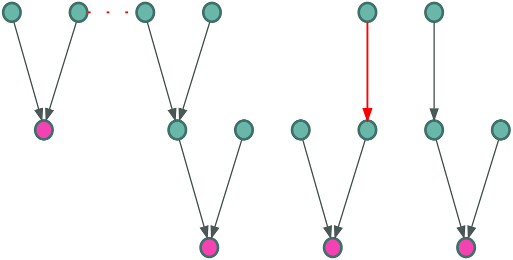
Dynamic Edges

Adding an Edge

- Now suppose that the height is greater than or equal to α
- We can't necessarily just combine the trees like before, we could end up with a tree greater than the maximum height.
- We break off some part of the bigger tree into the smaller.



Adding Edges



(a) Height $\geq \alpha$ (b)

Removing Edges

- We can also remove edges from our De Bruijn data structure

Removing Edges

- We can also remove edges from our De Bruijn data structure
- If the edge is not in our tree, we don't need to do anything.

Removing Edges

- We can also remove edges from our De Bruijn data structure
- If the edge is not in our tree, we don't need to do anything.
- If the edge is in our tree, then removing it breaks up a tree.

Removing Edges

- We can also remove edges from our De Bruijn data structure
- If the edge is not in our tree, we don't need to do anything.
- If the edge is in our tree, then removing it breaks up a tree.
- We need to find a new root for one of the trees

Removing Edges

- We can also remove edges from our De Bruijn data structure
- If the edge is not in our tree, we don't need to do anything.
- If the edge is in our tree, then removing it breaks up a tree.
- We need to find a new root for one of the trees
- If a tree is too short, use similar techniques to above to produce a taller tree.

Experiments

Datasets:

Dataset	Chromosomes	Read count	Read length	Size
S288C[4]	17	-	-	12M
E. coli	1	$27 \cdot 10^6$	101	3.3G

Platform:

- Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz (18 cores) with 396 GB RAM

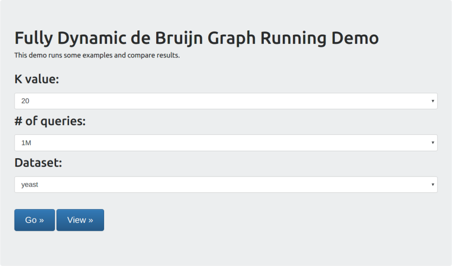
The results are tested by one run, which could be affected by computational resource conflicts.

---

[4] http://www.yeastgenome.org/strain/S288C/overview

Live Demo

1. We have developed a web-based demo to allow users do multiple types of comparisons, which makes the evaluation easier.
2. The link is here: `http://128.227.162.189:9999/`. Visit and play with it!



Fully Dynamic de Bruijn Graph Running Demo

This demo runs some examples and compare results.

K value: 20

# of queries: 1M

Dataset: yeast

Comparative Algorithms

1. **Bloom Filter:** Standard Bloom filter
2. **KBF1:** One-sided Bloom filter improves false postive rate three fold without using any additional storage.
3. **KBF2:** Two-sided Bloom filter improves FPR by an order of magnitude while using very little additional memory[5].

---

[5] https://github.com/Kingsford-Group/kbf

Query *k*-mer Generation

1. We generate query *k*-mers with three different number of queries:

-	1	2	3
number of queries (million)	0.5	1	10

2. We use two random ways to generate query *k*-mers
    - Muting one base of randomly extracted from the input *k*-mers
    - ~~Purely random *k*-mer generation.~~ However, it does not result in obvious difference compared with the first one.

FDBG Data Structure Information

Table : The FDBG data structure information on E coli.

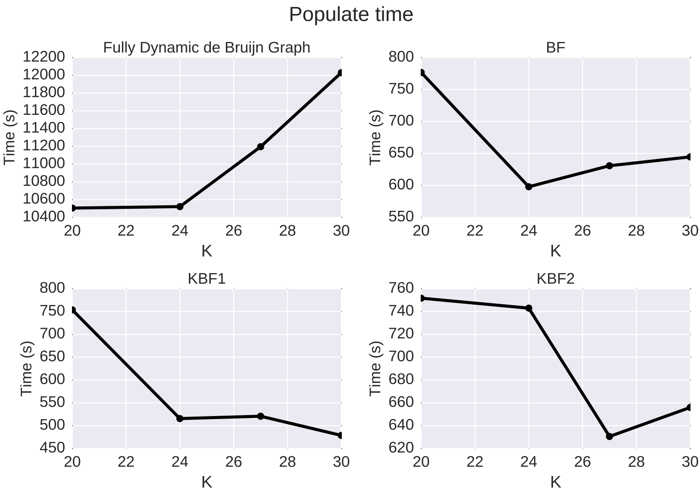
k	k-mers	RAM (MB)	Trees	Avg. height
20	770,956,037	1,485	5,492,320	48.14
24	784,990,222	1,519	6,412,386	50.45
27	783,739,686	1,517	6,532,142	47.28
30	776,321,600	1,505	6,752,622	48.46

- RAM = IN and OUT matrices + forest + minimal perfect hashing
- An example when $k = 20, N = 770956037, R = 5492320, \sigma = 4$:

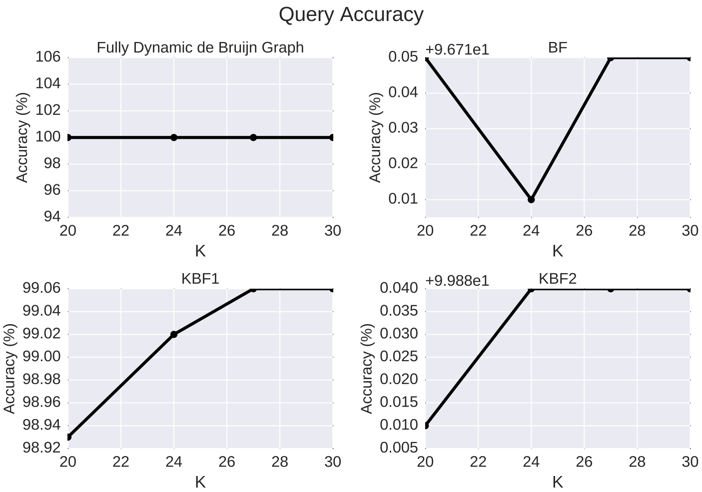
$$N \times \sigma \times 2 + (N \times 4 + R * 64) + \text{mph}$$

$$= (735 + (367 + 335) + 48) MB \times (bit/MB)$$

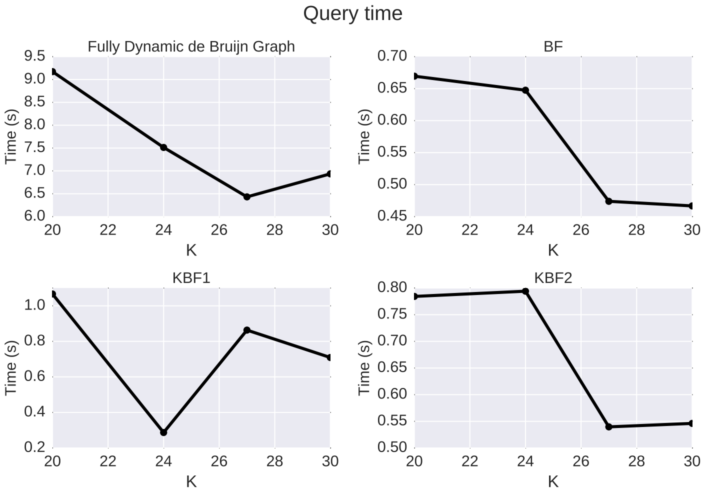
Populate Time w.r.t k



Query Accuracy w.r.t *k*



Query Time w.r.t *k*



Query Time w.r.t Tree Height

Table : Query time w.r.t the tree height.

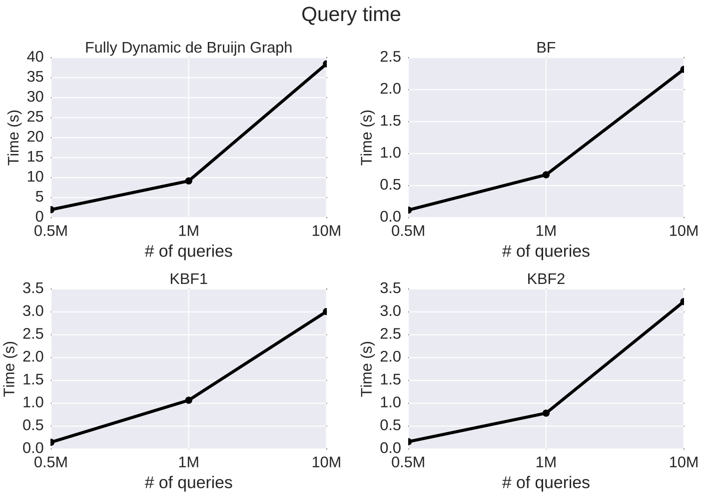
k	Avg. height	Query time (s)
20	48.14	6.69 [6]
24	50.45	7.51
27	47.28	6.43
30	48.46	6.93

Lower tree height gives rive to lower query time, because it needs fewer steps to trace the tree.

---

[6] Averaged by two runs.

Query Time w.r.t Number of Queries



Conclusion

1. Have implemented static De Bruijn graph data structure and will complete edge insertion and deletion.
2. Have tested the algorithms on two gene datasets. Compared with all Bloom filter based methods, ours has exactly 100% accuracy.
3. Have compared with three different methods.
4. Have made a website and a live demo.

End

Thanks and Questions?