CS267 Assignment 3:

Parallelize Graph Algorithms for de Novo Genome Assembly

Spring 2016

Problem statement

- **Input**: A set of unique k-mers and their corresponding extensions.
  - k-mers are sequences of length k (alphabet is A/C/G/T).
  - An extension is a simple symbol (A/C/G/T/F).
  - The input k-mers form a de Bruijn graph, a special graph that is used to represent overlaps between sequences of symbols.
- **Output:** A set of contigs, i.e. connected components in the input de Bruijn graph.

Example

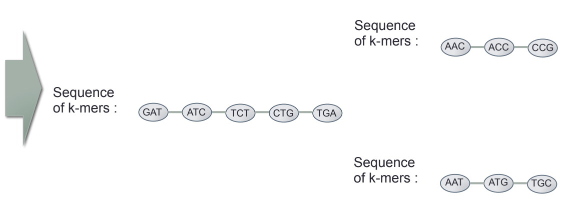
- **Input**: A set of unique k-mers and their corresponding extensions.
- Example for k = 3:
- Format: k-mer forward extension , backward extension

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA

Example

- **Input**: A set of unique k-mers and their corresponding extensions.
  - The input corresponds to a de Bruijn graph.
- Example for k = 3:

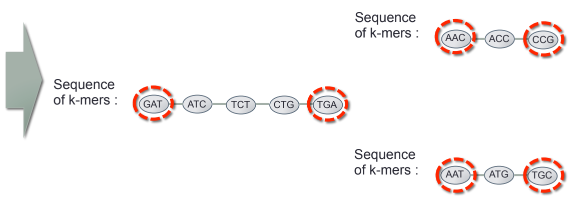
AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA



Example

- **Input**: A set of unique k-mers and their corresponding extensions.
  - The input corresponds to a de Bruijn graph.
- Example for k = 3:

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA

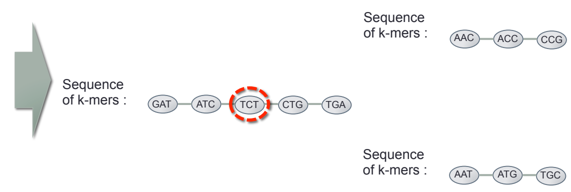


k-mers with "F" as an extension are *start vertices*

Example

- **Input**: A set of unique k-mers and their corresponding extensions.
  - The input corresponds to a de Bruijn graph.
- Example for k = 3:

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA

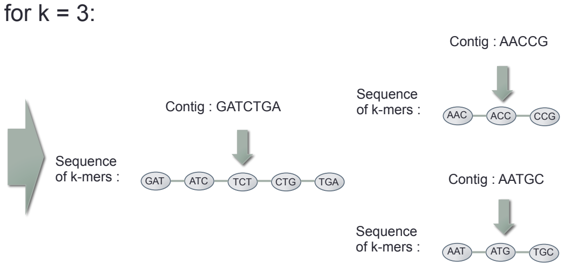


- Consider k-mer: TCT
- Concatenate last k-1 bases (CT) and forward extension (G) => CTG (following vertex)
- Concatenate backward extension (A) and first k-1bases (TC) =>ATC (preceding vertex)
- The graph is undirected, we can visit a vertex from both directions.

Example

- **Input**: A set of unique k-mers and their corresponding extensions.
  - The input corresponds to a de Bruijn graph.
- Example for k = 3:

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA

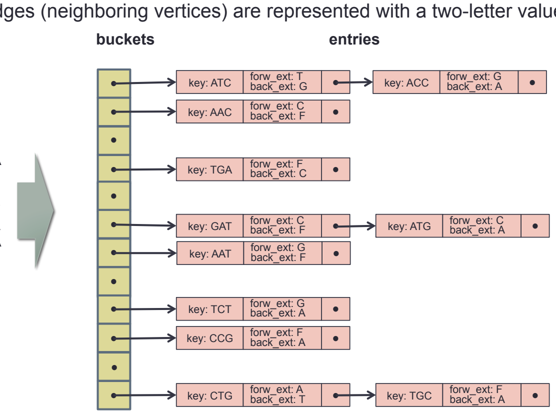


- **Output:** A set of contigs or equivalently the connected components in the de Bruijn graph

Compact graph representation: hash table

- The vertices are keys
- The edges (neighboring vertices) are represented with a two-letter value

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA



Serial algorithm

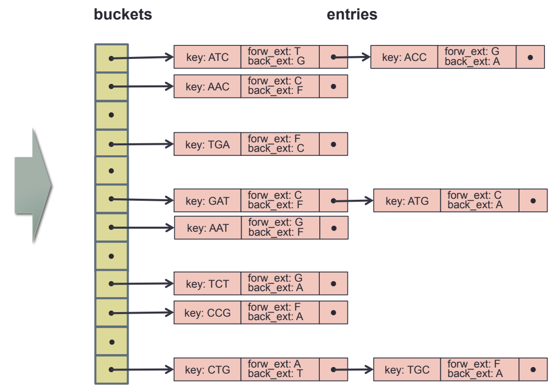
Algorithm 1 De Bruijn Graph Construction And Traversal

```
Input: A set of k-mers and their corresponding forward and backward extensions
Output: A set of contigs
/* Initialization */
hashTable ←CREATEHASHTABLE( )
startNodesList ←CREATEEMPTYLIST( )

/* De Bruijn Graph Construction */
for each (k-mer, forwardExt, backwardExt) in input do
    ADDKMERTOHASHTABLE(hashTable, (k-mer, forwardExt, backwardExt))
    if (backwardExt is F) then
        ADDKMERTOLIST(startNodesList, (k-mer, forwardExt))
    end if
end for

/* De Bruijn Graph Traversal */
for each (k-mer, forwardExt) in startNodesList do
    currentContig ←CREATENEWSEQUENCE(k-mer)
    currentForwardExtension ←forwardExt
    while (currentForwardExtension is not F) do
        ADDBASETOSEQUENCE(currentForwardExtension, currentContig)
        currentKmer ← LASTKBASES(currentContig)
        currentForwardExtension ← LOOKUP(hashTable, currentKmer)
    end while
    STORECONTIG(currentContig)
end for
```


Graph construction

- The vertices are keys
- The edges (neighboring vertices) are represented with a two-letter value

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA

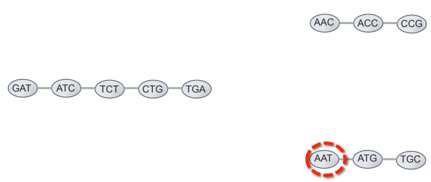


Graph traversal

- We pick a start vertex and we initiate a contig.

Contig: **A A T**

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA

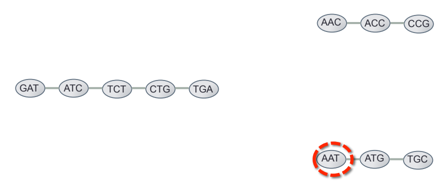


Graph traversal

- We add the forward extension to the contig.

Contig: **A A T G**

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA

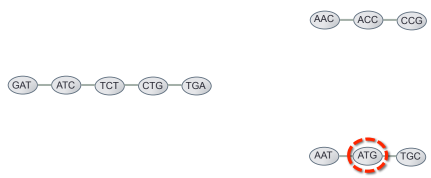


Graph traversal

- We take the last k bases of the contig and look them up in the hash table.

Contig: A A T G

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA

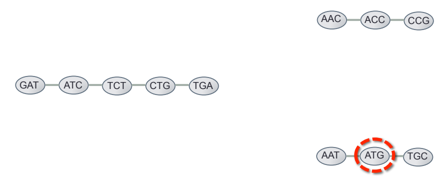


Graph traversal

- We add the new forward extension to the contig.

Contig: A A T G C

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA

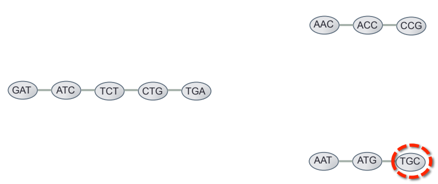


Graph traversal

- We take the last k bases of the contig and look them up in the hash table.

Contig: A A T G C

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA

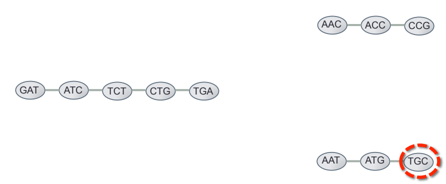


Graph traversal

- We terminate the current contig since the forward extension is an “F”.

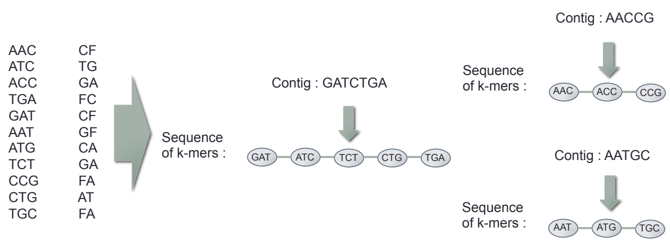
Contig: A A T G C

AAC	CF
ATC	TG
ACC	GA
TGA	FC
GAT	CF
AAT	GF
ATG	CA
TCT	GA
CCG	FA
CTG	AT
TGC	FA



Graph traversal

- We iterate until we exhaust all start vertices: we have found all the contigs.



Parallelization hints

1. Distribute the hash table among the processors.
   - UPC is convenient: Store the hash table in the shared address space.
   - You may want to use upc_all_alloc().
2. Each processor stores part of the input in the distributed hash table.
   - What happens if two processors try to write the same bucket at the same time?
   - We need to avoid race conditions (UPC provides locks and global atomics).
3. We want to traverse the graph in parallel.
   - Can we determine independent traversals by examining the input?
   - How can we distribute the work among processors?