Popping Superbubbles and Discovering Clumps

Recent Developments in Biological Sequence Analysis

Costas S. Iliopoulos, Ritu Kundu & Fatima Vayani

King's College London

April 12, 2016

California State University, Channel Islands

Outline

Outline

Introduction

Computational genomics can be divided into two complex and major stages, each of which presents many problems:

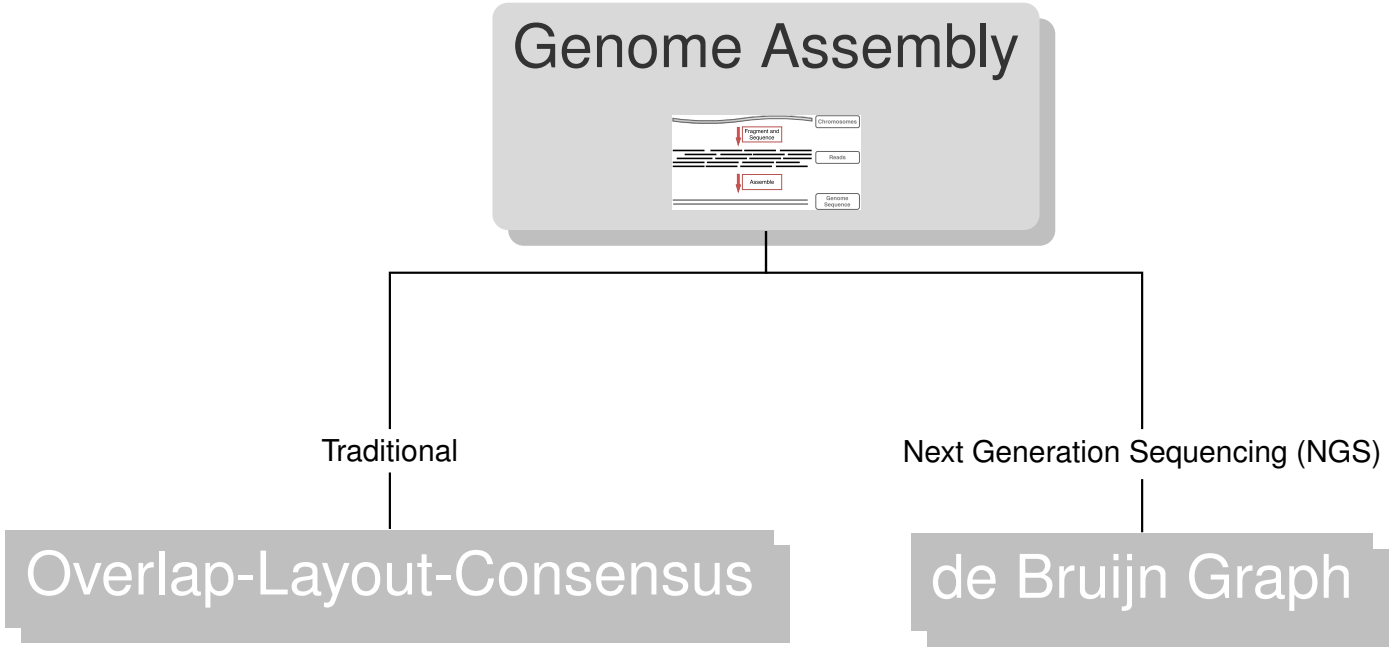
1. Assembly (error correction)
2. Annotation (motif discovery, gene prediction)

We focus on one problem from each stage:

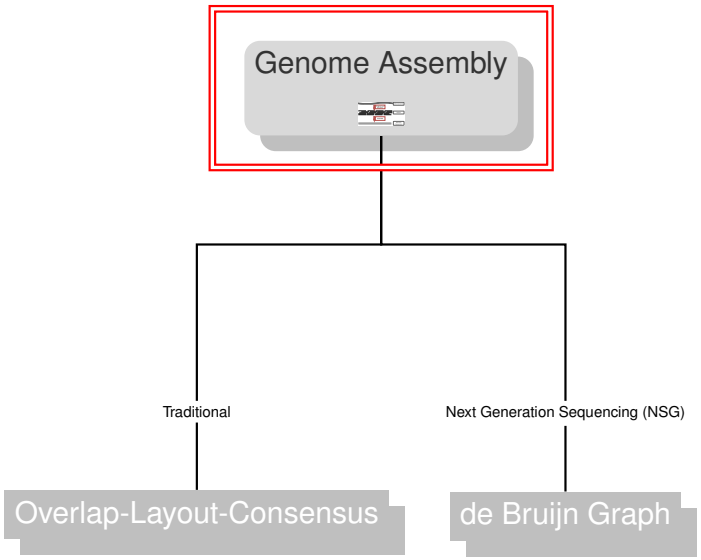
1. *Superbubble* detection during genome assembly
2. Finding *clumps* for genome annotation

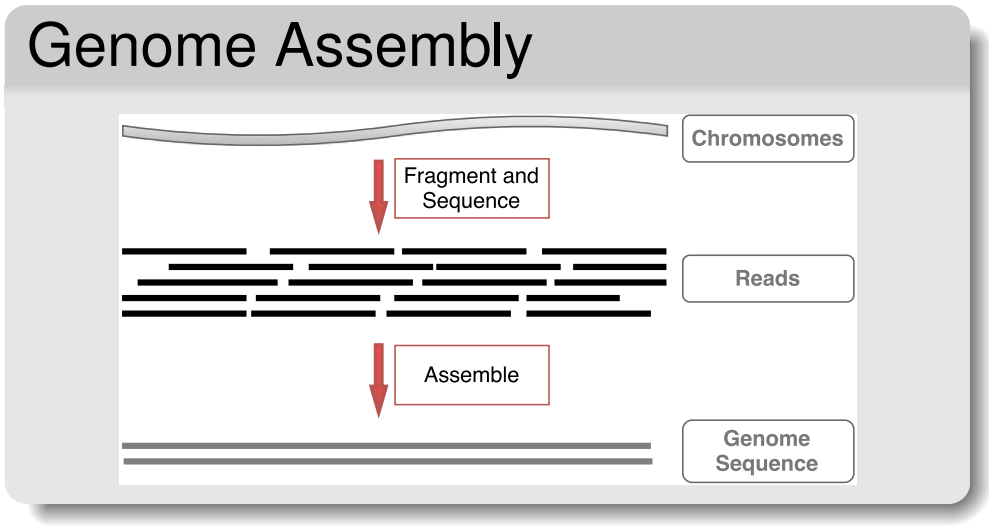
Outline

Motivation

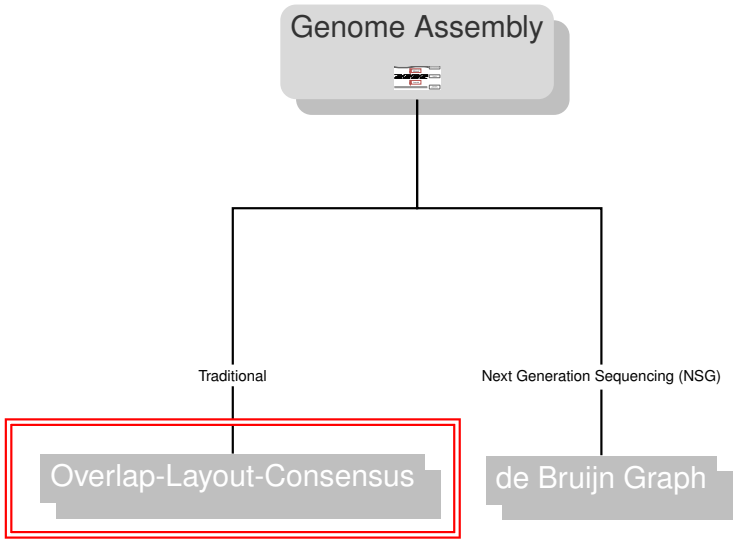

Genome Assembly
Chromosomes
Fragment and Sequence
Reads
Assemble
Genome Sequence
Traditional
Next Generation Sequencing (NGS)
Overlap-Layout-Consensus
de Bruijn Graph

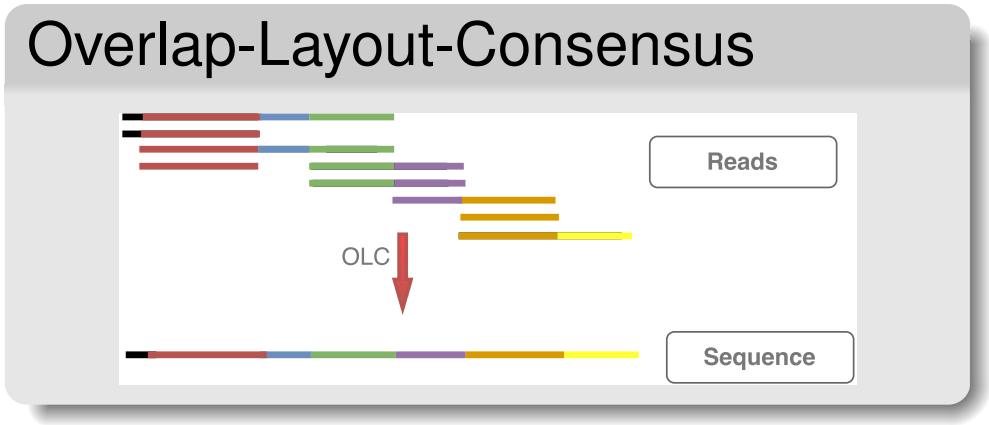
Motivation


Genome Assembly
Traditional
Next Generation Sequencing (NSG)
Overlap-Layout-Consensus
de Bruijn Graph

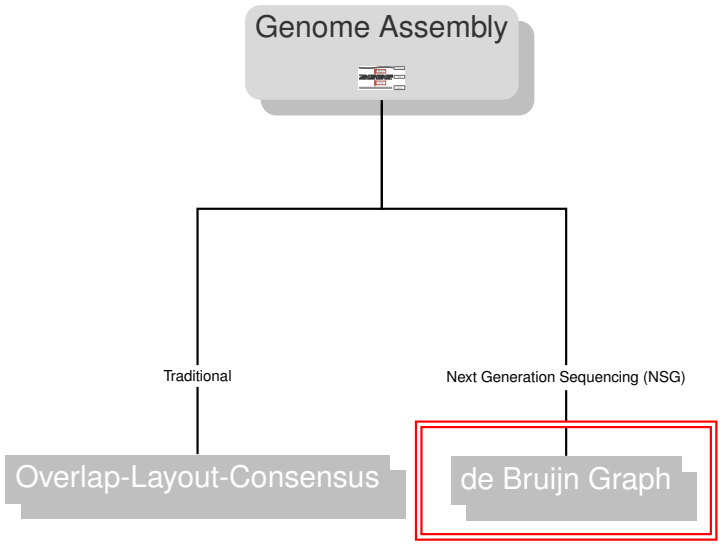

Genome Assembly
Chromosomes
Fragment and Sequence
Reads
Assemble
Genome Sequence

Motivation

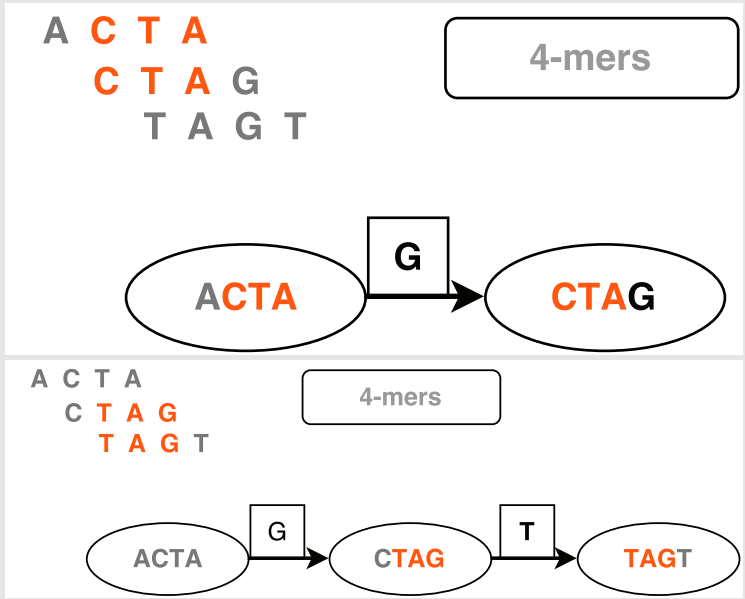

Genome Assembly
Traditional
Next Generation Sequencing (NSG)
Overlap-Layout-Consensus
de Bruijn Graph


Overlap-Layout-Consensus
Reads
OLC
Sequence

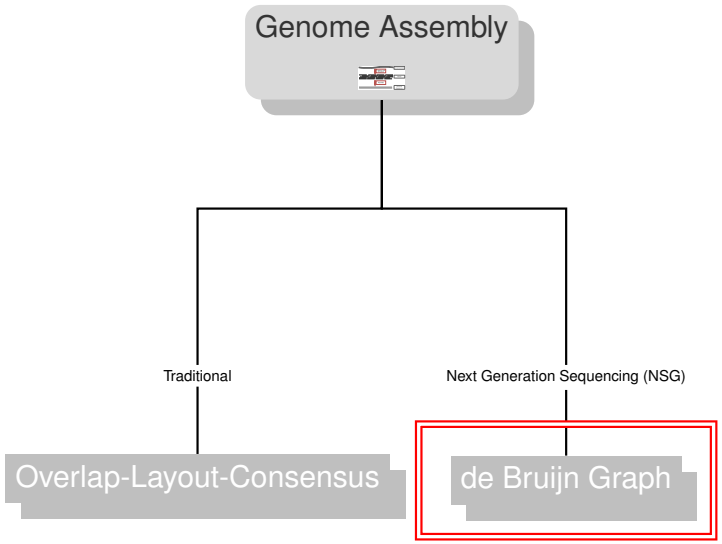
Motivation



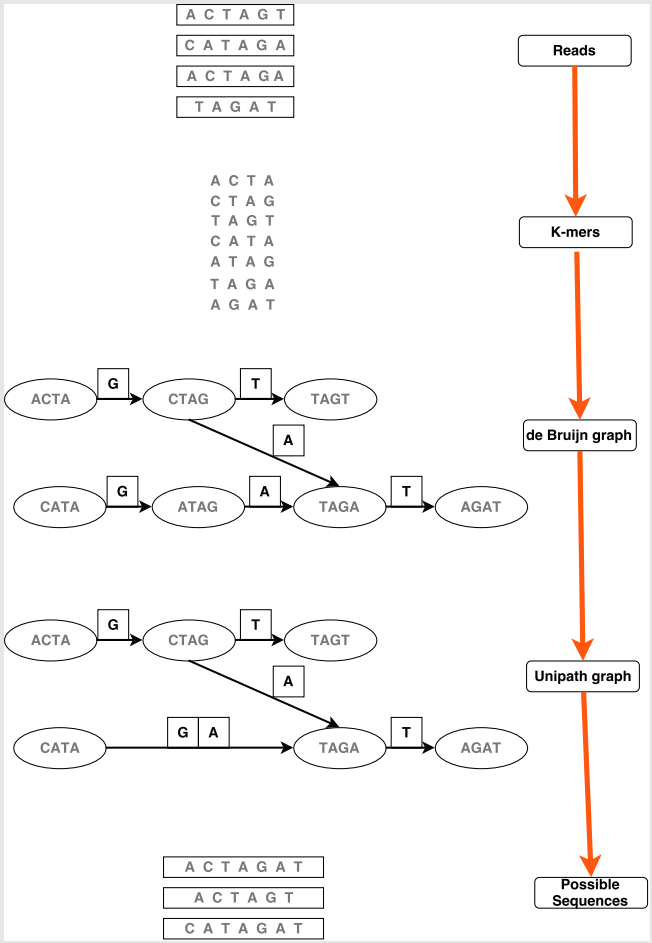
de Bruijn Graph



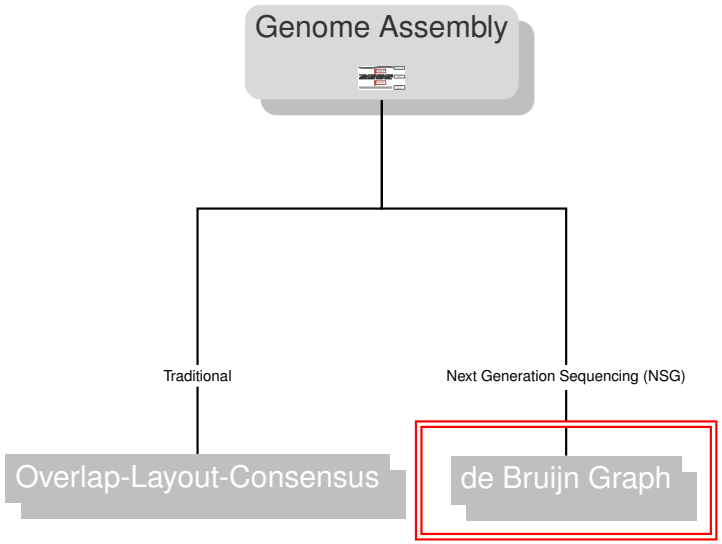
Motivation



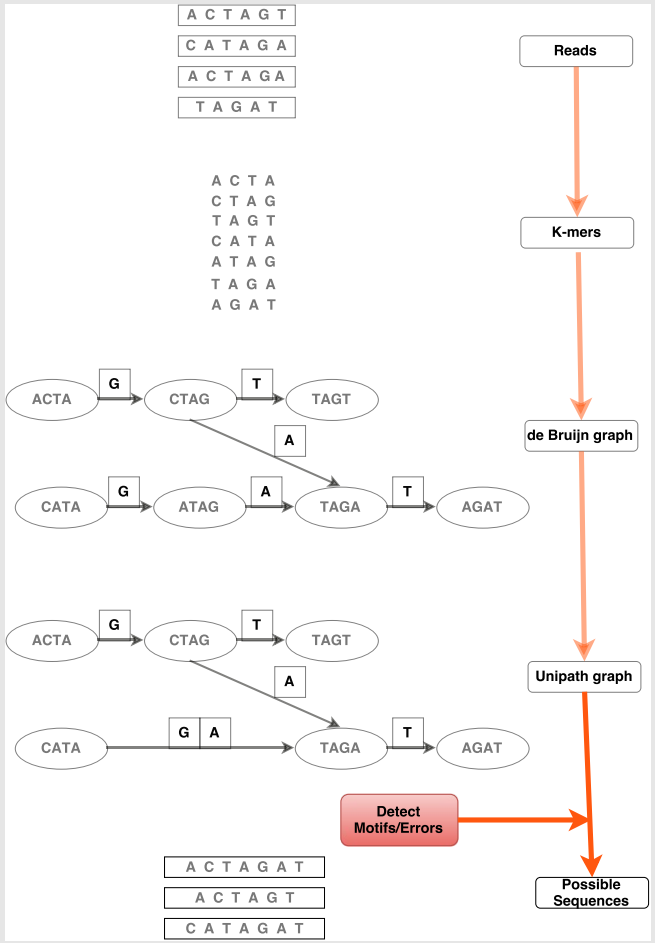
NSG



Motivation

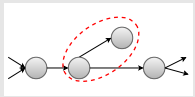


NSG

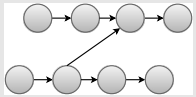


Motifs

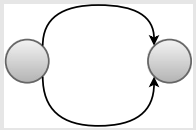
- Tips
  - A low-frequency edge whose end (or start) vertex has no outgoing (resp. incoming) edges, which goes out from (resp. comes into) a high-frequency vertex.
  - Often appears in case there are some error(s) around the end of a read.



- Cross-links
  - It is a low-frequency edge that lies between high-frequency vertices.
  - Appears when a substring of a read accidentally becomes (by error) the same substring that appears in a different region.



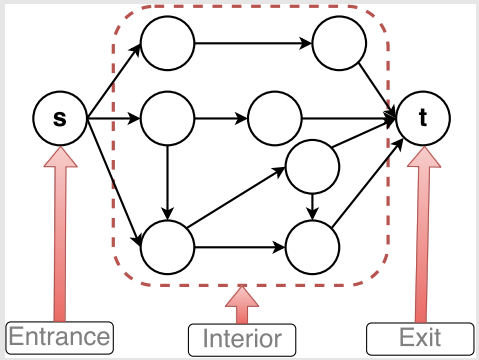
- Bubbles
  - It consists of multiple edges (with the same direction) between a pair of vertices.
  - Often caused by error(s) somewhere in the middle of a read.



- More Complex Structures? **Superbubble**

Superbubble

Superbubble: $\langle s,\ t \rangle$

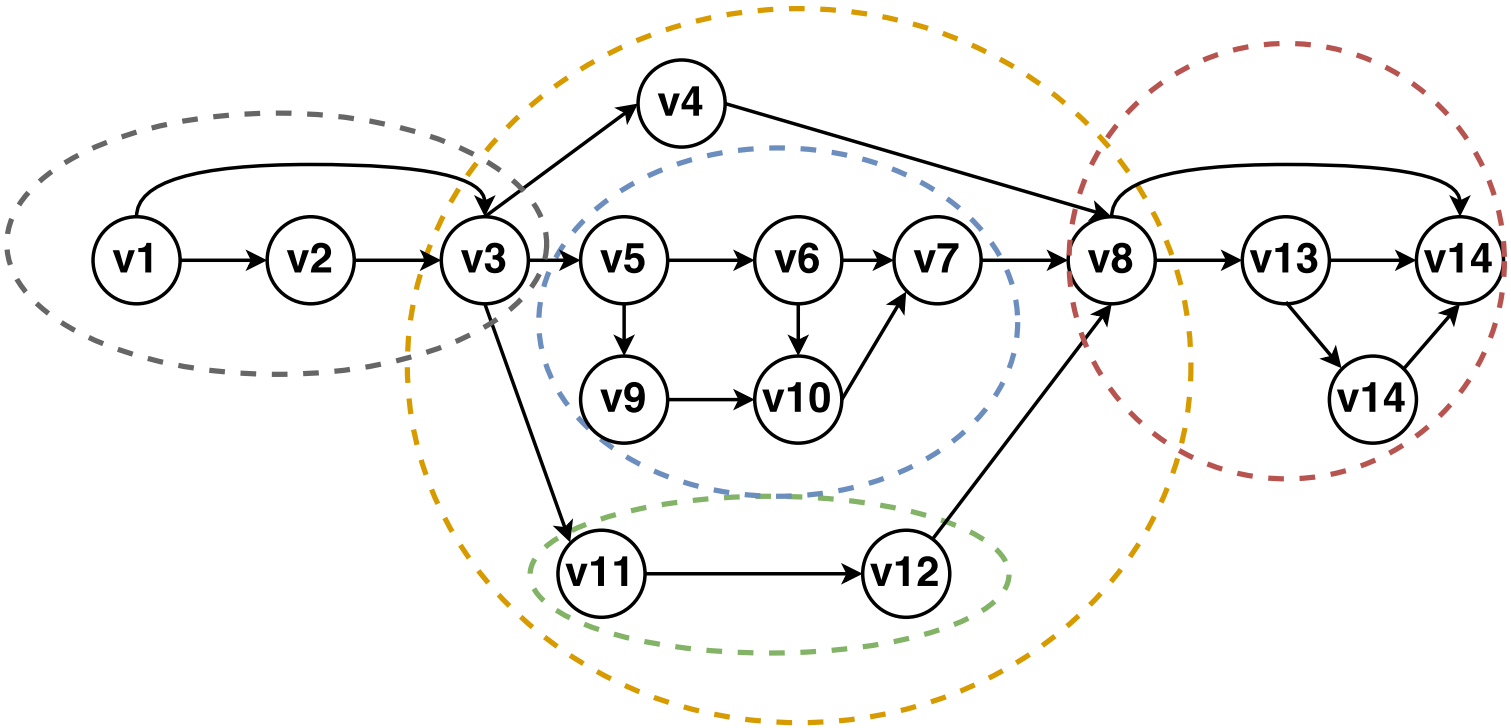


Definition [Onodera et al., 2013]

Let $G = (V, E)$ be a directed graph. For any ordered pair of distinct nodes s and t, $\langle s, t \rangle$ is called a *superbubble* if it satisfies the following:

- **reachability:** t is reachable from s;
- **matching:** the set of nodes reachable from s without passing through t is equal to the set of nodes from which t is reachable without passing through s;
- **acyclicity:** the subgraph induced by U is acyclic, where U is the set of nodes satisfying the matching criterion;
- **minimality:** no node in U other than t forms a pair with s that satisfies the conditions above;

Example



Properties of superbubbles

Lemma ([Onodera et al., 2013])

Any node can be the entrance (respectively exit) of at most one superbubble.

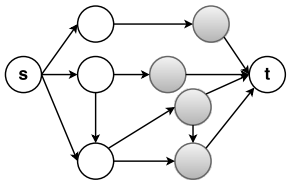
Lemma ([Sung et al., 2015])

Let G be a directed acyclic graph. We have the following two observations.
1) Suppose (p, c) is an edge in G, where p has one child and c has one parent, then $\langle p, c\rangle$ is a superbubble in G.
2) For any superbubble $\langle s, t\rangle$ in G, there must exist some parent p of t such that p has exactly one child t.

Lemma ([Brankovic et al., 2015])

For any superbubble $\langle s, t\rangle$ in a directed acyclic graph G, there must exist some child c of s such that c has exactly one parent s.

Properties of superbubbles



Lemma ([Onodera et al., 2013])

Any node can be the entrance (respectively exit) of at most one superbubble.

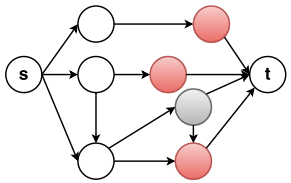
Lemma ([Sung et al., 2015])

Let G be a directed acyclic graph. We have the following two observations.
1) Suppose (p, c) is an edge in G, where p has one child and c has one parent, then $\langle p, c\rangle$ is a superbubble in G.
2) For any superbubble $\langle s, t\rangle$ in G, there must exist some parent p of t such that p has exactly one child t.

Lemma ([Brankovic et al., 2015])

For any superbubble $\langle s, t\rangle$ in a directed acyclic graph G, there must exist some child c of s such that c has exactly one parent s.

Properties of superbubbles



Lemma ([Onodera et al., 2013])

Any node can be the entrance (respectively exit) of at most one superbubble.

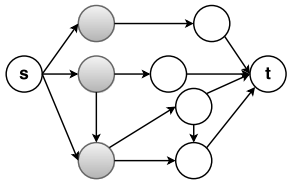
Lemma ([Sung et al., 2015])

Let G be a directed acyclic graph. We have the following two observations.
1) Suppose (p, c) is an edge in G, where p has one child and c has one parent, then $\langle p, c\rangle$ is a superbubble in G.
2) For any superbubble $\langle s, t\rangle$ in G, there must exist some parent p of t such that p has exactly one child t.

Lemma ([Brankovic et al., 2015])

For any superbubble $\langle s, t\rangle$ in a directed acyclic graph G, there must exist some child c of s such that c has exactly one parent s.

Properties of superbubbles



Lemma ([Onodera et al., 2013])

Any node can be the entrance (respectively exit) of at most one superbubble.

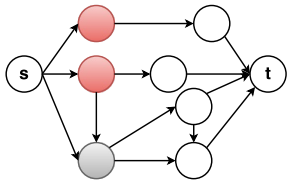
Lemma ([Sung et al., 2015])

Let G be a directed acyclic graph. We have the following two observations.
1) Suppose (p, c) is an edge in G, where p has one child and c has one parent, then $\langle p, c\rangle$ is a superbubble in G.
2) For any superbubble $\langle s, t\rangle$ in G, there must exist some parent p of t such that p has exactly one child t.

Lemma ([Brankovic et al., 2015])

For any superbubble $\langle s, t\rangle$ in a directed acyclic graph G, there must exist some child c of s such that c has exactly one parent s.

Properties of superbubbles



Lemma ([Onodera et al., 2013])

Any node can be the entrance (respectively exit) of at most one superbubble.

Lemma ([Sung et al., 2015])

Let G be a directed acyclic graph. We have the following two observations.
1) Suppose (p, c) is an edge in G, where p has one child and c has one parent, then $\langle p, c\rangle$ is a superbubble in G.
2) For any superbubble $\langle s, t\rangle$ in G, there must exist some parent p of t such that p has exactly one child t.

Lemma ([Brankovic et al., 2015])

For any superbubble $\langle s, t\rangle$ in a directed acyclic graph G, there must exist some child c of s such that c has exactly one parent s.

Problem : Report All Superbubbles

Input

A directed graph $G = (V, E)$.

Output

All ordered pairs of distinct nodes s and t, such that $\langle s, t \rangle$ is a *superbubble.*

Solutions

- **Solution 1:** $\mathcal{O}(nm)$**-time algorithm [Onodera et al., 2013]**
- **Solution 2:** $\mathcal{O}(m \log m)$**-time algorithm [Sung et al., 2015]**
- **Solution 3:** $\mathcal{O}(n + m)$**-time algorithm [Brankovic et al., 2015]**

Solution 1

- Visit nodes in an order that follows the standard topological sorting, starting from a given node s in the given directed graph $G = (V, E)$, to eventually report a node t such that $\langle s, t \rangle$ is a *superbubble* (if any).
  - Abort every time either a tip or a cycle is encountered.
- Iterate for all $s \in V$.

Solution 2

- Partition graphs into a set of sub-graphs -
  - subgraphs corresponding to each non-singleton strongly connected component
  - a subgraph corresponding to the set of all the nodes involved in singleton strongly connected components.
- Convert each subgraph into acyclic if it is cyclic.
- Find superbubbles in each of the subgraph.

Solution 2

- Partition graphs into a set of sub-graphs - → linear
  - subgraphs corresponding to each non-singleton strongly connected component
  - a subgraph corresponding to the set of all the nodes involved in singleton strongly connected components.
- Convert each subgraph into acyclic if it is cyclic. → linear
- Find superbubbles in each of the subgraph. → $\mathcal{O}(m \log m)$

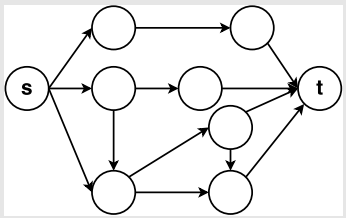
Solution 3

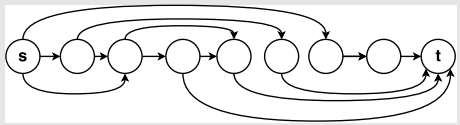
- Topologically order the vertices
- Identify possible entrance and exit candidates: *Candidate-list*
- Traverse candidate-list (in reverse topological order) to find superbubbles using subroutines:
  - *Report*
  - *Validate*

Solution 3

- Topologically order the vertices
- Identify possible entrance and exit candidates: *Candidate-list*
- Traverse candidate-list (in reverse topological order) to find superbubbles using subroutines:
  - *Report*
  - *Validate*

Example





Solution 3

- Topologically order the vertices
- Identify possible entrance and exit candidates: *Candidate-list*
- Traverse candidate-list (in reverse topological order) to find superbubbles using subroutines:
  - *Report*
  - *Validate*

Candidate

A node v is an

- **exit candidate:** if it has at least one parent with exactly one child (out-degree 1)
- **entrance candidate:** if it has at least one child with exactly one parent (in-degree 1)

(From Lemmas 2 and 3)

Identifying Candidates

Check each node in V, in topological order, to identify whether it is an exit or an entrance candidate (or both).

- If both, add twice (first as entrance and then as exit).

Solution 3

- Topologically order the vertices
- Identify possible entrance and exit candidates: *Candidate-list*
- Traverse candidate-list (in reverse topological order) to find superbubbles using subroutines:
  - *Report*
  - *Validate*

- Reports all the possible superbubbles (including nested ones) between given *start* and *exit*
- Called for each exit candidate in decreasing order either by *main* routine or through a recursive call to identify a nested superbubble.

Solution 3

- Topologically order the vertices
- Identify possible entrance and exit candidates: *Candidate-list*
- Traverse candidate-list (in reverse topological order) to find superbubbles using subroutines:
  - *Report*
  - *Validate*

- Valid: For a valid superbubble $\langle s, t \rangle$, every $x \in U \backslash \{s, t\}$ has
  - t as its *topologically furthest* child
  - s as its *topologically furthest* parent
- Maintain arrays of topological furthest child and parent resp., for each vertex and using RMQ to verify the conditions for validity.

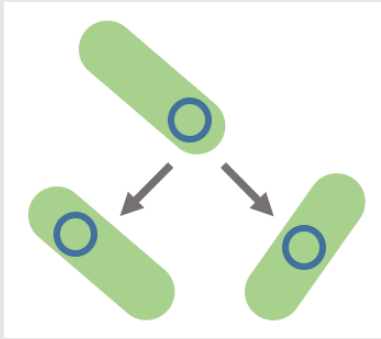
Discussion

- A critical step of assembly algorithms utilizing de Bruijn graphs is to detect typical motif structures in the graph caused by sequencing errors and genome repeats, and filter them out; one such complex subgraph class is a so-called *superbubble*.
- A superbubble $\langle s, t \rangle$ is equivalent to a single source, single sink, acyclic directed subgraph of G with source s and sink t, which does not have any cut nodes and preserves all in-degrees and out-degrees of nodes in $U \backslash \{s, t\}$, as well as the out-degree of s and in-degree of t.
- Given a directed acyclic graph $G = (V, E)$, where $n = |V|$ and $m = |E|$, all superbubbles in G can be identified in $\mathcal{O}(n + m)$-time.
- What's next? More complex motifs.

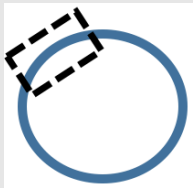
Outline

Motivation

DNA replication occurs during bacterial proliferation.



The Origin of Replication (OriC) is the region where DNA replication begins in a circular bacterial genome.



OriC can be identified by the occurrence of specific patterns in a region.

Motivating Example

E. coli

OriC in *E. coli* is 245 base pairs long. It contains:

- Three `AT`-rich 13-mers
- Three 6-mers (`AGATCT`) called DnaA boxes
- Seven 4-mers (`GATC`) which are enzyme recognition sites

Adjacent to this region, five 9-mers occur (`TTWTNCACA`).

Note

`W` $= \{$`T,A`$\}$ and `N` $= \{$`A,G,C,T`$\}$. Thus, degenerate symbols must be allowed.

Preliminary Definitions

Degenerate Symbol

A *degenerate* (or *non-solid*) symbol $\tilde{\sigma}$ over an alphabet Σ is a non-empty subset of Σ. For example, for the English alphabet, $\tilde{\sigma} = \{a, b, d, f\}$ is a degenerate symbol.

Degenerate String

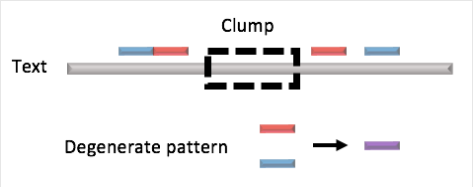
A *degenerate string* is built over the potential $2^{|\Sigma|} - 1$ non-empty sets of letters belonging to Σ. In other words, a degenerate string $\tilde{X} = \tilde{X}[1..n]$, such that every $\tilde{X}[i]$ is a degenerate symbol, $1 \leq i \leq n$. For example, $\tilde{X} = \{a, b\}\{a\}\{c\}\{b, c\}\{a\}\{a, b, c\}$ is a degenerate string of length 6 over $\Sigma = \{a, b, c\}$.

Conservative Degenerate String

A *conservative degenerate string* is a degenerate string where its number of degenerate symbols is upper-bounded by a fixed positive constant k.

Clumps

Clump



Definition

For a given text T, an (ℓ, k)-clump in T is defined as a factor $T[i..i+\ell]$, which contains a conservative degenerate pattern $\tilde{P}$, such that $|\tilde{P}| = m$ and

$$Occ_{T[i..i+\ell]}(\tilde{P}) \geq k,$$

where $Occ_T(\tilde{P})$ is the number of occurrences of pattern $\tilde{P}$ in T, and the number of degenerate symbols in $\tilde{P} \leq d$.

Problem: Finding Clumps

Input

A text T of length n, and integers $\ell < n$, $k < \ell/2$, $m < \ell$ and $d < m$.

Output

All (ℓ, k)-clumps in T, that is, all factors $T[i..i + \ell]$, which contain a conservative degenerate pattern $\tilde{P}$.

Solution

- Construct the suffix array SA of the string T.
- Construct the longest common prefix array LCP of the string T.
- Assign a rank for each prefix (of length m) of each suffix, based on the order of the suffix array. Use these ranks to construct a new string T'.
- Construct and maintain a *Parikh vector* $\mathcal{V}(T')$, where each component of $\mathcal{V}(T')$ denotes the number of occurrences of the corresponding rank.
- At step i, when the window is shifted one position to the right, to position $i+1$, we have to update $\mathcal{V}(T')$.
- Compute the number of occurrences of every possible conservative degenerate pattern $\tilde{P}$.

Solution

- Construct the suffix array SA of the string T.
- Construct the longest common prefix array LCP of the string T.
- Assign a rank for each prefix (of length m) of each suffix, based on the order of the suffix array. Use these ranks to construct a new string T'.
- Construct and maintain a *Parikh vector* $\mathcal{V}(T')$, where each component of $\mathcal{V}(T')$ denotes the number of occurrences of the corresponding rank.
- At step i, when the window is shifted one position to the right, to position $i+1$, we have to update $\mathcal{V}(T')$.
- Compute the number of occurrences of every possible conservative degenerate pattern $\tilde{P}$.

i	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	A	G	C	T	T	G	C	T	A	G	C	T
SA$[i]$	9	1	11	7	3	10	6	2	12	8	5	4

Solution

- Construct the suffix array SA of the string T.
- Construct the longest common prefix array LCP of the string T.
- Assign a rank for each prefix (of length m) of each suffix, based on the order of the suffix array. Use these ranks to construct a new string T'.
- Construct and maintain a *Parikh vector* $\mathcal{V}(T')$, where each component of $\mathcal{V}(T')$ denotes the number of occurrences of the corresponding rank.
- At step i, when the window is shifted one position to the right, to position $i+1$, we have to update $\mathcal{V}(T')$.
- Compute the number of occurrences of every possible conservative degenerate pattern $\tilde{P}$.

i	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	A	G	C	T	T	G	C	T	A	G	C	T
SA$[i]$	9	1	11	7	3	10	6	2	12	8	5	4
LCP$[i]$	0	4	0	2	2	0	3	3	0	1	1	1

Solution

- Construct the suffix array SA of the string T.
- Construct the longest common prefix array LCP of the string T.
- Assign a rank for each prefix (of length m) of each suffix, based on the order of the suffix array. Use these ranks to construct a new string T'.
- Construct and maintain a *Parikh vector* $\mathcal{V}(T')$, where each component of $\mathcal{V}(T')$ denotes the number of occurrences of the corresponding rank.
- At step i, when the window is shifted one position to the right, to position $i+1$, we have to update $\mathcal{V}(T')$.
- Compute the number of occurrences of every possible conservative degenerate pattern $\tilde{P}$.

i	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	A	G	C	T	T	G	C	T	A	G	C	T
SA$[i]$	9	1	11	7	3	10	6	2	12	8	5	4
LCP$[i]$	0	4	0	2	2	0	3	3	0	1	1	1
$T'[i]$	0	3	2	6	5	3	1	4	0	3		

Solution

- Construct the suffix array SA of the string T.
- Construct the longest common prefix array LCP of the string T.
- Assign a rank for each prefix (of length m) of each suffix, based on the order of the suffix array. Use these ranks to construct a new string T'.
- Construct and maintain a *Parikh vector* $\mathcal{V}(T')$, where each component of $\mathcal{V}(T')$ denotes the number of occurrences of the corresponding rank.
- At step i, when the window is shifted one position to the right, to position $i+1$, we have to update $\mathcal{V}(T')$.
- Compute the number of occurrences of every possible conservative degenerate pattern $\tilde{P}$.

- The vector $\mathcal{V}(T')$ is initialised with the numbers of occurrences of all ranks in the prefix of T' of length $\ell' = \ell + m - 1$.
- After the initialisation, we proceed using a sliding window of length ℓ' to maintain the Parikh vector $\mathcal{V}(T')$.

Solution

- Construct the suffix array SA of the string T.
- Construct the longest common prefix array LCP of the string T.
- Assign a rank for each prefix (of length m) of each suffix, based on the order of the suffix array. Use these ranks to construct a new string T'.
- Construct and maintain a *Parikh vector* $\mathcal{V}(T')$, where each component of $\mathcal{V}(T')$ denotes the number of occurrences of the corresponding rank.
- At step i, when the window is shifted one position to the right, to position $i+1$, we have to update $\mathcal{V}(T')$.
- Compute the number of occurrences of every possible conservative degenerate pattern $\tilde{P}$.

If $\mathcal{V}_{T'[i+\ell']}(T')$ is at least k; then the factor $T[i..i+\ell-1]$ is a (ℓ, k)-clump with at least k occurrences of a some solid pattern P of length m.

Solution

- Construct the suffix array SA of the string T.
- Construct the longest common prefix array LCP of the string T.
- Assign a rank for each prefix (of length m) of each suffix, based on the order of the suffix array. Use these ranks to construct a new string T'.
- Construct and maintain a *Parikh vector* $\mathcal{V}(T')$, where each component of $\mathcal{V}(T')$ denotes the number of occurrences of the corresponding rank.
- At step i, when the window is shifted one position to the right, to position $i+1$, we have to update $\mathcal{V}(T')$.
- Compute the number of occurrences of every possible conservative degenerate pattern $\tilde{P}$.

Each such pattern $\tilde{P}$ is formed by merging components of $\mathcal{V}(T')$ which have at most d mismatches with the factor of rank $T'[i+\ell']$. We calculate the number of occurrences of $\tilde{P}$ by adding the appropriate components of $\mathcal{V}(T')$.

Example

i	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	A	G	C	T	T	G	C	T	A	G	C	T
SA$[i]$	9	1	11	7	3	10	6	2	12	8	5	4
LCP$[i]$	0	4	0	2	2	0	3	3	0	1	1	1
$T'[i]$	0	3	2	6	5	3	1	4	0	3		

Here, rank 0 represents AGC and occurs twice in T'; rank 1 represents CTA; rank 2 represents CTT; rank 3 represents GCT and occurs three times in T'; rank 4 represents TAG; rank 5 represents TGC; and rank 6 represents TTG .

Example

Supposing $\ell = 7$, the tables below represent $\mathcal{V}(T')$ when it is initialised (Step 0) and after each of the first 5 steps of the computation.

Step 0

0	1
1	0
2	1
3	1
4	0
5	1
6	1

$\Rightarrow$

Step 1

0	0
1	0
2	1
3	2
4	0
5	1
6	1

$\Rightarrow$

Step 2

0	0
1	1
2	1
3	1
4	0
5	1
6	1

$\Rightarrow$

Step 3

0	0
1	1
2	0
3	1
4	1
5	1
6	1

$\Rightarrow$

Step 4

0	1
1	1
2	0
3	1
4	1
5	1
6	0

$\Rightarrow$

Step 5

0	1
1	1
2	0
3	2
4	1
5	0
6	0

- Suppose $d = 0$, two $(7, 2)$-clumps are reported at Step 1 & Step 5 both associated with the solid pattern `GCT` of rank 3.
- Suppose $d = 1$, then three more $(7, 2)$-clumps are reported at Step 0, 2 & 3 associated with {A, T}GC, CT{A, T}, and T{A, T}G.

Discussion

Algorithm complexity

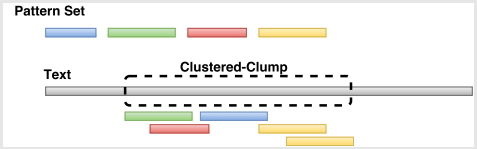
Reporting all possible (ℓ, k)-clumps in a given text T can be achieved in $\mathcal{O}(nm^d)$ time.

Future work

Compare the efficiency and accuracy of our approach with OriFinder [Gao and Zhang, 2008], a tool which uses statistical analysis.

Clustered-Clumps

Clustered-clump



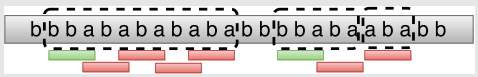
Definition [Bassino et al., 2008]

For a given set of patterns $\mathcal{P} = \{P_1, \cdots, P_{|\mathcal{P}|}\}$, a *clustered-clump* is a maximal set of occurrences of patterns of $\mathcal{P}$ such that

- any consecutive letters of the clustered-clump is a *f*actor of at least one occurrence of a pattern from $\mathcal{P}$.
- either the clustered-clump is composed of a *s*ingle occurrence that overlaps no other occurrence, or each occurrence *o*verlaps at least one other occurrence.

Example

If $\mathcal{P} = \{\texttt{aba}, \texttt{bba}\}$ and the text $T = \texttt{bbbabababababbbbabaabababb}$,



Motivation

The locations of genes can be predicted by the occurrences of regulatory sequences adjacent to them.

- Gene promoters
- Start/stop codons
- Protein-binding sites

Problem: Finding Clustered-Clumps

Input

A text T of length n, a set of conserved degenerate patterns $\tilde{\mathcal{P}} = \{\tilde{P}_1, \cdots, \tilde{P}_{|\tilde{\mathcal{P}}|}\}$, and integers d and m.

Output

All clustered-clumps in T, where the total number of non-solid symbols in $\tilde{\mathcal{P}} \leq d$ and $m = \sum_{1 \leq i \leq |\tilde{\mathcal{P}}|} |\tilde{P}_i|$.

Solution

- Split every pattern, $\tilde{P}_i$, into subpatterns, $P_{i,j}$, by chopping out the parts containing non-solid symbols so that each of the subpatterns is solid: this new set of solid subpatterns is $\mathcal{P}$.
  - Let *sub*(i) denote the number of subpatterns obtained from a pattern $\tilde{P}_i$.
- Build the Aho-Corasick Automaton, $\mathcal{S}(\mathcal{P})$, of the set $\mathcal{P}$.
  - $\mathcal{S}(\mathcal{P})$ is the minimal deterministic finite automaton whose language is the set of suffixes of subpatterns in $\mathcal{P}$.
- Process the occurrences of each of the subpatterns such that an occurrence of $P_{i,sub(i)}$ corresponds to an occurrence of a non-solid pattern $\tilde{P}_i$ in T.
- Populate an array of size n that stores the length of the longest pattern occurring at each position of the text.

Solution

- Split every pattern, $\tilde{P}_i$, into subpatterns, $P_{i,j}$, by chopping out the parts containing non-solid symbols so that each of the subpatterns is solid: this new set of solid subpatterns is $\mathcal{P}$.
  - Let $sub(i)$ denote the number of subpatterns obtained from a pattern $\tilde{P}_i$.
- Build the Aho-Corasick Automaton, $\mathcal{S}(\mathcal{P})$, of the set $\mathcal{P}$.
  - $\mathcal{S}(\mathcal{P})$ is the minimal deterministic finite automaton whose language is the set of suffixes of subpatterns in $\mathcal{P}$.
- Process the occurrences of each of the subpatterns such that an occurrence of $P_{i,sub(i)}$ corresponds to an occurrence of a non-solid pattern $\tilde{P}_i$ in T.
- Populate an array of size n that stores the length of the longest pattern occurring at each position of the text.

Example

Suppose $\tilde{\mathcal{P}}$= {AC{TG}AA{CG}TAA, AT{C,G}TT{A,G}C}. Then $\mathcal{P}$ = {AC, AA, TAA, AT, TT, C}, and $sub(1) = sub(2) = 3$.

Solution

- Split every pattern, $\tilde{P}_i$, into subpatterns, $P_{i,j}$, by chopping out the parts containing non-solid symbols so that each of the subpatterns is solid: this new set of solid subpatterns is $\mathcal{P}$.
  - Let $sub(i)$ denote the number of subpatterns obtained from a pattern $\tilde{P}_i$.
- Build the Aho-Corasick Automaton, $\mathcal{S}(\mathcal{P})$, of the set $\mathcal{P}$.
  - $\mathcal{S}(\mathcal{P})$ is the minimal deterministic finite automaton whose language is the set of suffixes of subpatterns in $\mathcal{P}$.
- Process the occurrences of each of the subpatterns such that an occurrence of $P_{i,sub(i)}$ corresponds to an occurrence of a non-solid pattern $\tilde{P}_i$ in T.
- Populate an array of size n that stores the length of the longest pattern occurring at each position of the text.

Compute all the occurrences of the solid subpatterns in the text T (linear time).

Solution

- Split every pattern, $\tilde{P}_i$, into subpatterns, $P_{i,j}$, by chopping out the parts containing non-solid symbols so that each of the subpatterns is solid: this new set of solid subpatterns is $\mathcal{P}$.
  - Let $sub(i)$ denote the number of subpatterns obtained from a pattern $\tilde{P}_i$.
- Build the Aho-Corasick Automaton, $\mathcal{S}(\mathcal{P})$, of the set $\mathcal{P}$.
  - $\mathcal{S}(\mathcal{P})$ is the minimal deterministic finite automaton whose language is the set of suffixes of subpatterns in $\mathcal{P}$.
- Process the occurrences of each of the subpatterns such that an occurrence of $P_{i,sub(i)}$ corresponds to an occurrence of a non-solid pattern $\tilde{P}_i$ in T.
- Populate an array of size n that stores the length of the longest pattern occurring at each position of the text.

If an occurrence of $P_{i,j}$ for $j > 1$ is found, then we need to check:

- Whether the non-solid symbol in $\tilde{P}_i$ preceding $P_{i,j}$ matches the corresponding position in T.
- Whether $P_{i,j-1}$ occurs in the corresponding position in T.

Solution

- Split every pattern, $\tilde{P}_i$, into subpatterns, $P_{i,j}$, by chopping out the parts containing non-solid symbols so that each of the subpatterns is solid: this new set of solid subpatterns is $\mathcal{P}$.
  - Let *sub*(i) denote the number of subpatterns obtained from a pattern $\tilde{P}_i$.
- Build the Aho-Corasick Automaton, $\mathcal{S}(\mathcal{P})$, of the set $\mathcal{P}$.
  - $\mathcal{S}(\mathcal{P})$ is the minimal deterministic finite automaton whose language is the set of suffixes of subpatterns in $\mathcal{P}$.
- Process the occurrences of each of the subpatterns such that an occurrence of $P_{i,sub(i)}$ corresponds to an occurrence of a non-solid pattern $\tilde{P}_i$ in T.
- Populate an array of size n that stores the length of the longest pattern occurring at each position of the text.

A single scan of this array can report the indices of all clustered-clumps in T.

Discussion

Algorithm complexity

The solution finds all the clustered-clumps in the text in time equal to $\mathcal{O}(n+m)$ (for constant d)

- Computing both $\mathcal{P}$ and $\mathcal{S}(\mathcal{P})$ takes $\mathcal{O}(m)$.
- $\mathcal{O}(dn)$ time is required for finding the occurrences of all solid subpatterns (to fill the matrix) and the occurrences of non-solid patterns subsequently.
- Scanning of the array in the last step can be done in $\mathcal{O}(n)$ time.

Future work

Extension of our solutions for finding clustered-clumps in texts with non-solid symbols.

Outline

Summary

Two research problems:

1. Error correction during genome assembly
   - Linear-time superbubble identification algorithm.
2. Genome annotation given complex parameters
   - Different types of clumps.
   - Full results published soon.

References I


Bassino, F., Clément, J., Fayolle, J., and Nicodème, P. (2008).
Constructions for clumps statistics.
CoRR, abs/0804.3671.


Brankovic, L., Iliopoulos, C. S., Kundu, R., Mohamed, M., Pissis, S. P., and Vayani, F. (2015).
Linear-time superbubble identification algorithm for genome assembly.
Theoretical Computer Science.


Gao, F. and Zhang, C.-T. (2008).
Ori-finder: a web-based system for finding orics in unannotated bacterial genomes.
BMC bioinformatics, 9(1):79.

References II

Onodera, T., Sadakane, K., and Shibuya, T. (2013).
Detecting superbubbles in assembly graphs.
In *WABI*, pages 338–348.

Sung, W., Sadakane, K., Shibuya, T., Belorkar, A., and Pyrogova, I. (2015).
An O(m log m)-time algorithm for detecting superbubbles.
IEEE/ACM Trans. Comput. Biology Bioinform., 12(4):770–777.

Thank You!