Index Structures in Biological Sequence Analysis

From Simplicity to Complexity and Back

Jens Stoye

AG Genominformatik, Technische Fakultät

Institute of Bioinformatics, Center of Biotechnology

Bielefeld University, Germany

Index structures in biological sequence analysis

1. Introduction
2. Suffix trees
3. Affix trees
4. Suffix arrays
5. The q-Gram index
6. Summary and Conclusion

Index structures in biological sequence analysis

1. Introduction
2. Suffix trees
3. Affix trees
4. Suffix arrays
5. The *q*-Gram index
6. Summary and Conclusion

Biological sequence analysis

The data:

- DNA sequences – may be very long, small alphabet A, C, G, T
- RNA sequences – usually moderately long
- protein sequences – usually short, larger alphabet size

The tasks:

- sequence comparison
- pattern matching
- pattern discovery

The challenges:

- efficient algorithms
- flexible tools
- statistical assessment of significance of results
- visualization

Biological sequence analysis

The data:

- DNA sequences – may be very long, small alphabet A, C, G, T
- RNA sequences – usually moderately long
- protein sequences – usually short, larger alphabet size

The tasks:

- sequence comparison
- pattern matching
- pattern discovery

The challenges:

- **efficient algorithms**
- flexible tools
- statistical assessment of significance of results
- visualization

Some applications

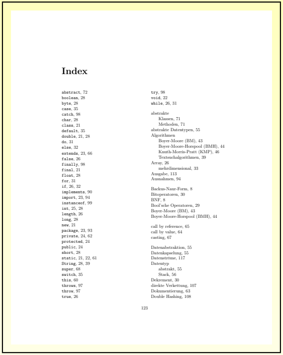
- Sequence comparison
  - alignment, multiple alignment
  - similar sequence → similar structure → similar function
- Pattern matching
  - mapping of *expressed sequence tags* (ESTs) on genomic DNA
  - targets of a given miRNA
  - palindromic or other RNA structural patterns
  - known repeats (for further exclusion from analysis)
- Pattern discovery
  - unknown promoter binding sites
  - repeats, tandem repeats
  - possible DNA methylation sites

Index structures

The result of **preprocessing** the data for faster search.

Index structures

The result of **preprocessing** the data for faster search.



Index

abstract, 72
boolean, 28
byte, 28
case, 35
catch, 98
char, 28
class, 21
default, 35
double, 21, 28
do, 31
else, 32
extends, 23, 66
false, 26
finally, 98
final, 21
float, 28
for, 31
if, 26, 32
implements, 90
import, 23, 94
instanceof, 99
int, 25, 28
length, 26
long, 28
new, 21
package, 23, 93
private, 24, 62
protected, 24
public, 24
short, 28
static, 21, 22, 61
String, 28, 39
super, 68
switch, 35
this, 60
throws, 97
throw, 97
true, 26
try, 98
void, 22
while, 26, 31

abstrakte
  Klassen, 71
  Methoden, 71
abstrakte Datentypen, 55
Algorithmen
  Boyer-Moore (BM), 43
  Boyer-Moore-Horspool (BMH), 44
  Knuth-Morris-Pratt (KMP), 46
  Textsuchalgorithmen, 39
Array, 26
  mehrdimensional, 33
Ausgabe, 113
Ausnahmen, 94

Backus-Naur-Form, 8
Bitoperatoren, 30
BNF, 8
Bool'sche Operatoren, 29
Boyer-Moore (BM), 43
Boyer-Moore-Horspool (BMH), 44

call by reference, 65
call by value, 64
casting, 67

Datenabstraktion, 55
Datenkapselung, 55
Datenströme, 117
Datentyp
  abstrakt, 55
  Stack, 56
Dekrement, 30
direkte Verkettung, 107
Dokumentierung, 63
Double Hashing, 108

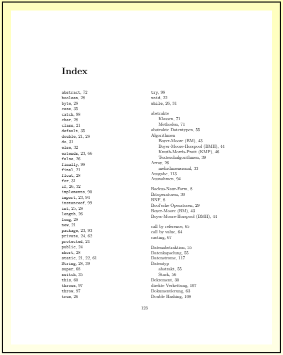
123

http://www.google.fr/

Done

Index structures

The result of **preprocessing** the data for faster search.



Index

abstract, 72
boolean, 28
byte, 28
case, 35
catch, 98
char, 28
class, 21
default, 35
double, 21, 28
do, 31
else, 32
extends, 23, 66
false, 26
finally, 98
final, 21
float, 28
for, 31
if, 26, 32
implements, 90
import, 23, 94
instanceof, 99
int, 25, 28
length, 26
long, 28
new, 21
package, 23, 93
private, 24, 62
protected, 24
public, 24
short, 28
static, 21, 22, 61
String, 28, 39
super, 68
switch, 35
this, 60
throws, 97
throw, 97
true, 26

try, 98
void, 22
while, 26, 31

abstrakte
  Klassen, 71
  Methoden, 71
abstrakte Datentypen, 55
Algorithmen
  Boyer-Moore (BM), 43
  Boyer-Moore-Horspool (BMH), 44
  Knuth-Morris-Pratt (KMP), 46
  Textsuchalgorithmen, 39
Array, 26
  mehrdimensional, 33
Ausgabe, 113
Ausnahmen, 94

Backus-Naur-Form, 8
Bitoperatoren, 30
BNF, 8
Bool'sche Operatoren, 29
Boyer-Moore (BM), 43
Boyer-Moore-Horspool (BMH), 44

call by reference, 65
call by value, 64
casting, 67

Datenabstraktion, 55
Datenkapselung, 55
Datenströme, 117
Datentyp
  abstrakt, 55
  Stack, 56
Dekrement, 30
direkte Verkettung, 107
Dokumentierung, 63
Double Hashing, 108

123

Google - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://www.google.fr/

Go

Accueil personnalisé | Connexion

Google France

Web Images Groupes Annuaire Actualités plus »

Recherche Google | J'ai de la chance

Recherche avancée
Préférences
Outils linguistiques

Rechercher dans : ● Web ○ Pages francophones ○ Pages : France

Publicité - Solutions d'entreprise - À propos de Google - Google.com in English

©2006 Google

Done

Many applications assume that the text is partitioned into words (natural language, syntactic tags, ...)

Genomic data is not divided into obvious “words”

→ we need indices that allow access to **any substring** of the text

Full-text index structures

Limitation:

- Most full-text indices allow only simple searches.

Full-text index structures

Limitation:

- Most full-text indices allow only simple searches.

But:

- Simple searches are often the core of more complex methods.

Full-text index structures

Limitation:

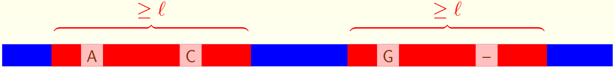
- Most full-text indices allow only simple searches.

But:

- Simple searches are often the core of more complex methods.

Example: degenerate repeats

Task: In a given string S of length n, find all pairs of occurrences of substrings of length at least ℓ that differ by at most k errors.

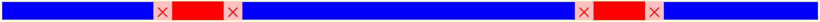


Finding degenerate repeats

Idea: A repeat of length ℓ with k errors contains an exact match of length at least $s := \lfloor \ell/(k+1) \rfloor$.

Algorithm:

1. Find all exact repeats of length $\geq s$. (Using an index)
2. Extend these by up to k errors.
3. Report matches whenever length ℓ is reached.

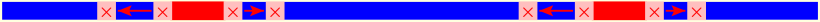


Finding degenerate repeats

Idea: A repeat of length ℓ with k errors contains an exact match of length at least $s := \lfloor \ell/(k+1) \rfloor$.

Algorithm:

1. Find all exact repeats of length $\geq s$. (Using an index)
2. Extend these by up to k errors.
3. Report matches whenever length ℓ is reached.

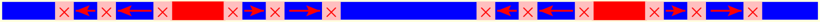


Finding degenerate repeats

Idea: A repeat of length ℓ with k errors contains an exact match of length at least $s := \lfloor \ell/(k+1) \rfloor$.

Algorithm:

1. Find all exact repeats of length $\geq s$. (Using an index)
2. Extend these by up to k errors.
3. Report matches whenever length ℓ is reached.

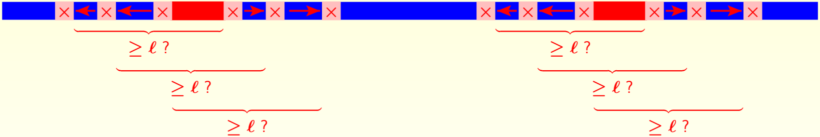


Finding degenerate repeats

Idea: A repeat of length ℓ with k errors contains an exact match of length at least $s := \lfloor \ell/(k+1) \rfloor$.

Algorithm:

1. Find all exact repeats of length $\geq s$. (Using an index)
2. Extend these by up to k errors.
3. Report matches whenever length ℓ is reached.

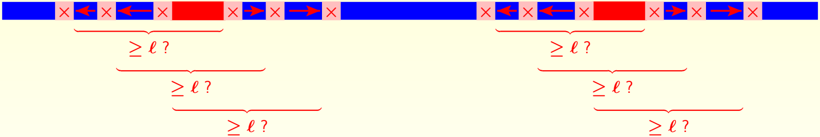


Finding degenerate repeats

Idea: A repeat of length ℓ with k errors contains an exact match of length at least $s := \lfloor \ell/(k+1) \rfloor$.

Algorithm:

1. Find all exact repeats of length $\geq s$. (Using an index)
2. Extend these by up to k errors.
3. Report matches whenever length ℓ is reached.



Analysis: $\mathcal{O}(n + \zeta k)$ time with $E(\zeta) = O\left(n^2/4^s\right)$.

String matching

Given a string S and a pattern P,
find all exact/approximate occurrences of P in S.

String matching

Given a string S and a pattern P,
find all exact/approximate occurrences of P in S.

(A) **Online:** no preprocessing of the text, linear search time

Exact string matching

- Finite automata, e.g. Knuth-Morris-Pratt, Aho-Corasick
- Boyer-Moore
- Boyer-Moore-Horspool

Approximate string matching

- Sellers' algorithm (dynamic programming)
- FASTA, BLAST (heuristic methods)

String matching

(B) **Offline:** preprocessing of the text, sublinear search time

Examples of full-text index structures:

- Suffix tree
- Patricia trie
- Directed acyclic word graph
- Suffix array
- String B tree
- Suffix cactus
- Suffix vector
- Factor oracle
- Enhanced suffix array
- Affix tree
- q-Gram index

String matching

(B) **Offline:** preprocessing of the text, sublinear search time

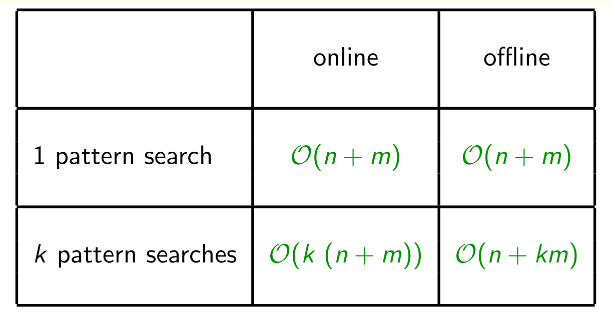
Examples of full-text index structures:

- **Suffix tree**
- Patricia trie
- Directed acyclic word graph
- **Suffix array**
- String B tree
- Suffix cactus
- Suffix vector
- Factor oracle
- Enhanced suffix array
- **Affix tree**
- **q-Gram index**

Exact string matching online and offline

Theoretical results:

Online search in $O(n+m)$ time possible
Offline search in $O(m)$ time after $O(n)$ time preprocessing



	online	offline
1 pattern search	$\mathcal{O}(n+m)$	$\mathcal{O}(n+m)$
k pattern searches	$\mathcal{O}(k\ (n+m))$	$\mathcal{O}(n+km)$

where n = text length, m = pattern length

Index structures in biological sequence analysis

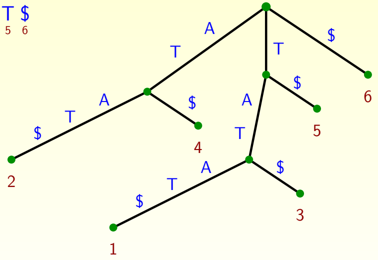
1. Introduction
2. Suffix trees
3. Affix trees
4. Suffix arrays
5. The q-Gram index
6. Summary and Conclusion

Suffix Tree: Definition

- A suffix of a string S of length n is a substring of S that ends at position n.

- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\$}$



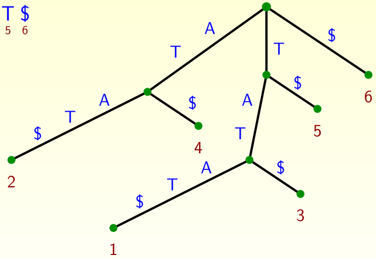
Suffix Tree: Definition

- A suffix of a string S of length n is a substring of S that ends at position n.

- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\$}$

$P = \mathsf{A\,T\,A}$



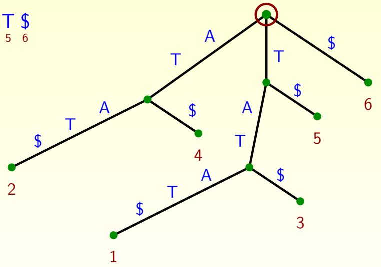
Suffix Tree: Definition

- A suffix of a string *S* of length *n* is a substring of *S* that ends at position *n*.

- The suffix tree of *S*, *T*(*S*), is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of *S*;
  - all edges leaving a node begin with different characters.

$S = \underset{1}{\text{T}}\,\underset{2}{\text{A}}\,\underset{3}{\text{T}}\,\underset{4}{\text{A}}\,\underset{5}{\text{T}}\,\underset{6}{\text{\$}}$

$P = \text{A T A}$



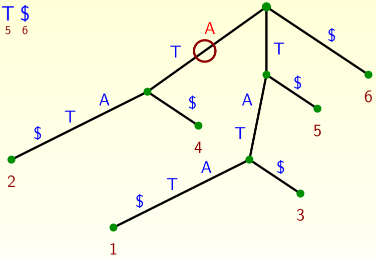
Suffix Tree: Definition

- A suffix of a string S of length n is a substring of S that ends at position n.

- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\$}$

$P = \mathsf{A\,T\,A}$



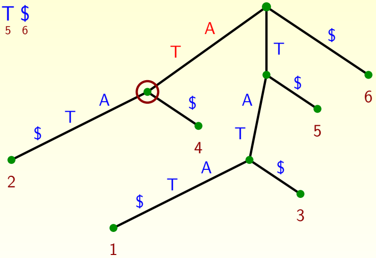
Suffix Tree: Definition

- A suffix of a string S of length n is a substring of S that ends at position n.

- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\$}$

$P = \mathsf{A\,T\,A}$



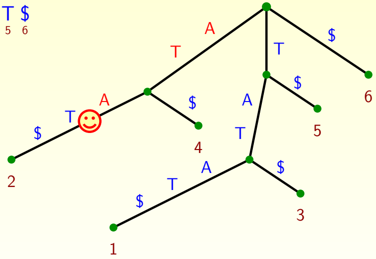
Suffix Tree: Definition

- A suffix of a string S of length n is a substring of S that ends at position n.

- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

$S = \underset{1}{\mathrm{T}}\,\underset{2}{\mathrm{A}}\,\underset{3}{\mathrm{T}}\,\underset{4}{\mathrm{A}}\,\underset{5}{\mathrm{T}}\,\underset{6}{\$}$

$P = \mathrm{A\,T\,A}$

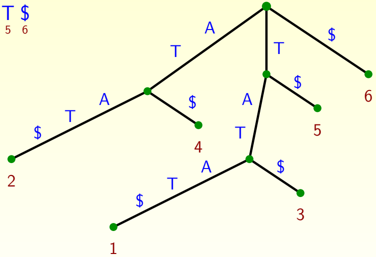


Suffix Tree: Definition

- A suffix of a string S of length n is a substring of S that ends at position n.

- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

$S = \underset{1}{\text{T}}\,\underset{2}{\text{A}}\,\underset{3}{\text{T}}\,\underset{4}{\text{A}}\,\underset{5}{\text{T}}\,\underset{6}{\text{\$}}$



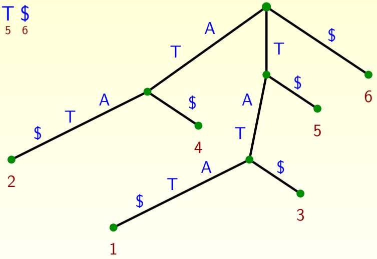
Suffix Tree: Definition

- A suffix of a string S of length n is a substring of S that ends at position n.

- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

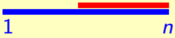
$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\$}$

$P = \mathsf{T\,A\,T\,T}$



Suffix Tree: Definition

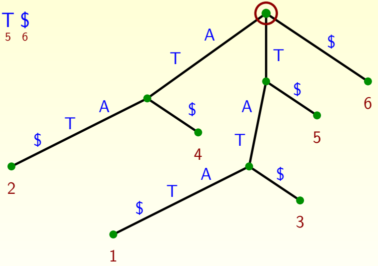
- A suffix of a string S of length n is a substring of S that ends at position n.



- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\$}$

$P = \mathsf{T\,A\,T\,T}$



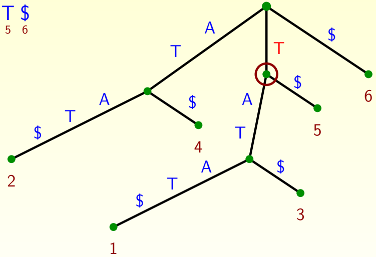
Suffix Tree: Definition

- A suffix of a string S of length n is a substring of S that ends at position n.

- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\$}$

$P = \mathsf{T\,A\,T\,T}$



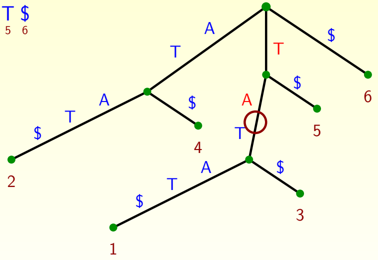
Suffix Tree: Definition

- A suffix of a string S of length n is a substring of S that ends at position n.

- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

$S = \underset{1}{\text{T}}\,\underset{2}{\text{A}}\,\underset{3}{\text{T}}\,\underset{4}{\text{A}}\,\underset{5}{\text{T}}\,\underset{6}{\text{\$}}$

$P = \text{T A T T}$



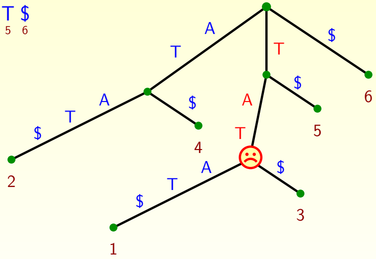
Suffix Tree: Definition

- A suffix of a string S of length n is a substring of S that ends at position n.

- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\$}$

$P = \mathsf{T\,A\,T\,T}$

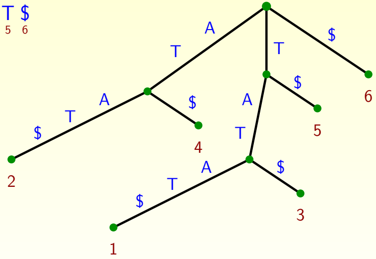


Suffix Tree: Definition

- A suffix of a string S of length n is a substring of S that ends at position n.

- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - the paths from the root to the leaves are the suffixes of S;
  - all edges leaving a node begin with different characters.

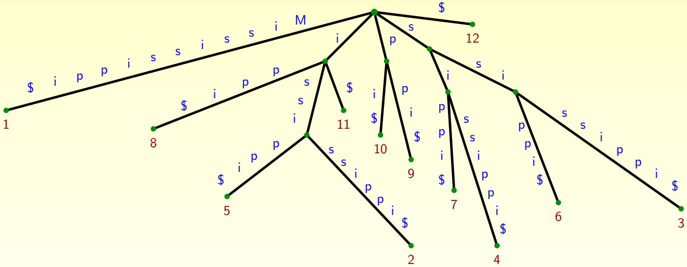
$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\$}$



A larger example

$S =$ M i s s i s s i p p i $

1 2 3 4 5 6 7 8 9 10 11 12



Suffix tree properties

- $T(S)$ represents exactly the substrings of S.
- $T(S)$ allows to enumerate these substrings and their locations in S in a convenient way.
- This is very useful for many pattern recognition problems, for example:
    - exact string matching as part of other applications, e.g. detecting DNA contamination
    - all-pairs suffix-prefix matching, important in fragment assembly
    - finding repeats and palindromes, tandem repeats, degenerate repeats
    - DNA primer design
    - DNA chip design
    - ...

See also:

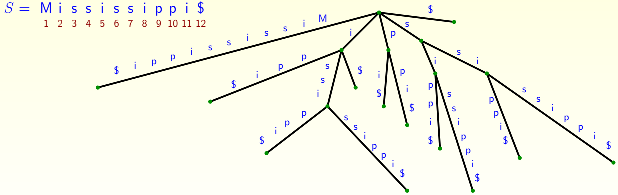
- A. Apostolico: The myriad virtues of subword trees, 1985.
- D. Gusfield: Algorithms on strings, trees, and sequences, 1997.

Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i,j) into S.

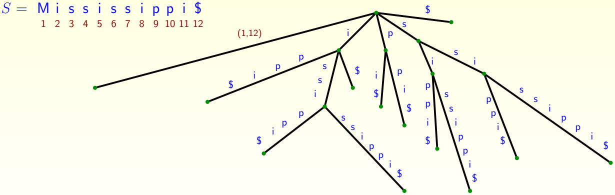


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S $\Rightarrow$ pair of pointers (i,j) into S.

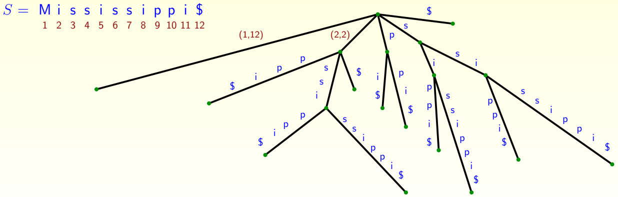


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i,j) into S.

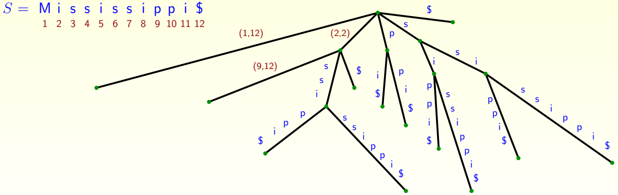


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i,j) into S.

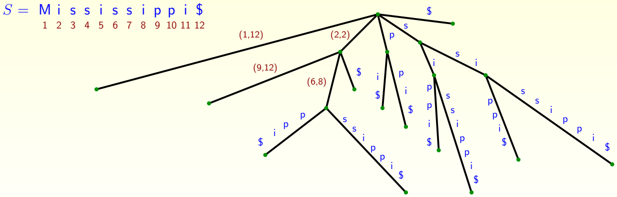


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S $\Rightarrow$ pair of pointers (i,j) into S.

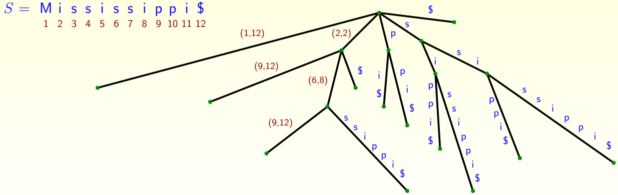


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i,j) into S.

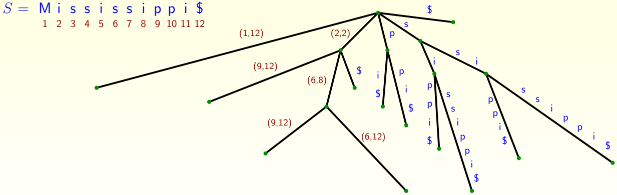


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S $\Rightarrow$ pair of pointers (i,j) into S.

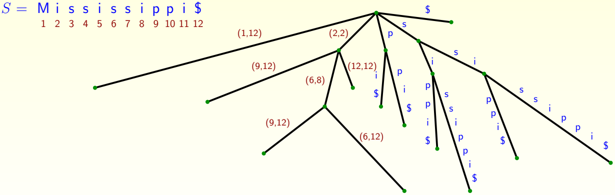


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i,j) into S.

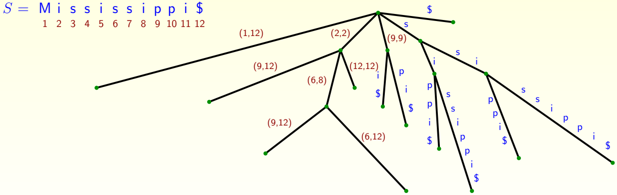


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i, j) into S.

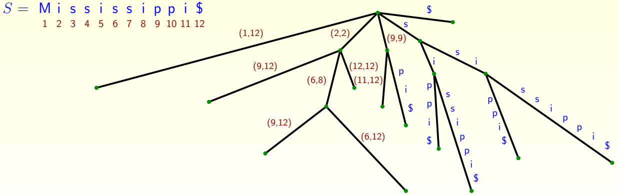


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i,j) into S.

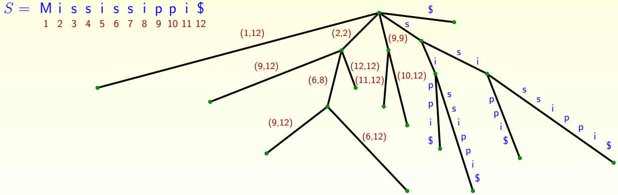


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i, j) into S.

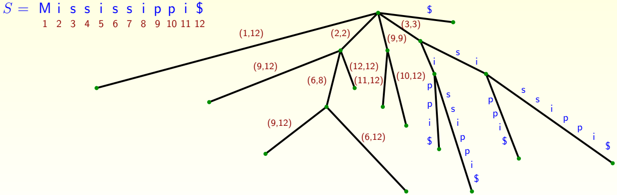


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S $\Rightarrow$ pair of pointers (i,j) into S.

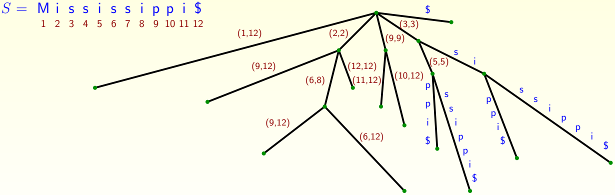


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i,j) into S.

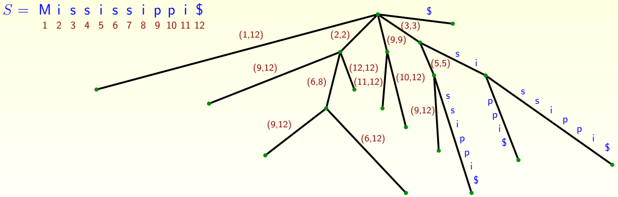


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i,j) into S.

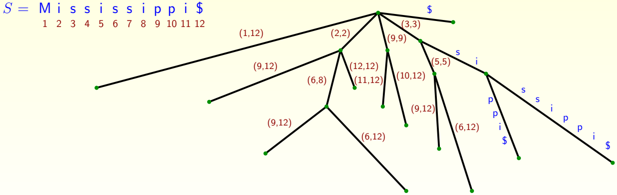


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i,j) into S.

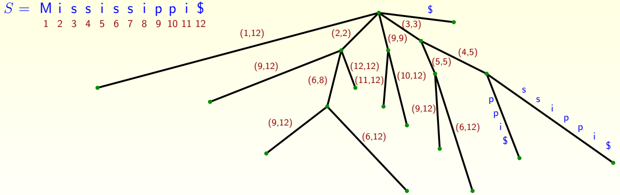


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i, j) into S.

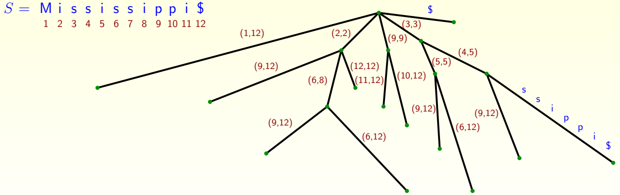


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S $\Rightarrow$ pair of pointers (i, j) into S.

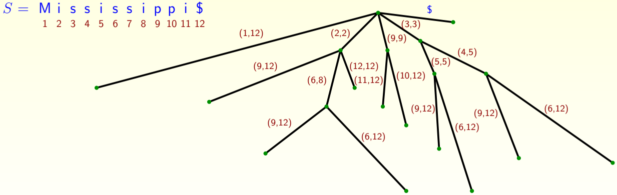


Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i,j) into S.



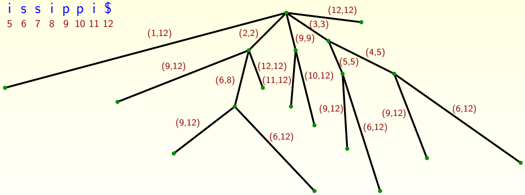
Space usage of suffix trees

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching ⇒ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of S ⇒ pair of pointers (i, j) into S.

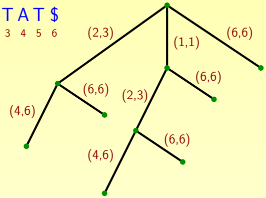
$S =$	M	i	s	s	i	s	s	i	p	p	i	$
	1	2	3	4	5	6	7	8	9	10	11	12

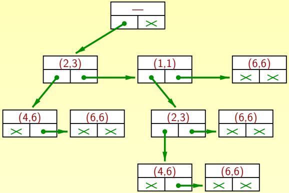


Representation of suffix trees

$S = \mathsf{TATAT\$}$

1 2 3 4 5 6



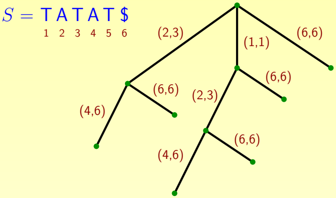


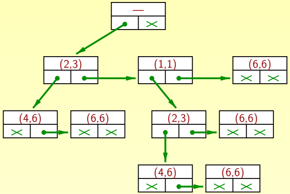
Standard representation of trees:

- Store nodes as records with child and sibling pointer.

⇒ about $32n$ bytes in the worst case

Representation of suffix trees





Standard representation of trees:

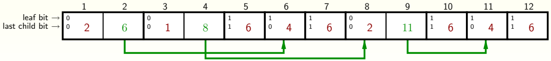
- Store nodes as records with child and sibling pointer.

⇒ about $32n$ bytes in the worst case

More efficient representation: (Giegerich, Kurtz & JS, *SP&E* 2003)

- Avoid storing redundant information.

⇒ below $12n$ bytes in the worst case, $8.5n$ on average



	1	2	3	4	5	6	7	8	9	10	11	12
leaf bit →	0		0		1	1	1	0		1	1	1
last child bit →	0		0		1	0	1	0		1	0	1
	2	6	1	8	6	4	6	2	11	6	4	6

Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.

Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.

Two practical algorithms: McCreight (1976) and Ukkonen (1993).

Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.

Two practical algorithms: McCreight (1976) and Ukkonen (1993).

A simpler algorithm: **Write-Only, Top-Down** (WOTD).

Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.

Two practical algorithms: McCreight (1976) and Ukkonen (1993).

A simpler algorithm: **Write-Only, Top-Down** (WOTD).

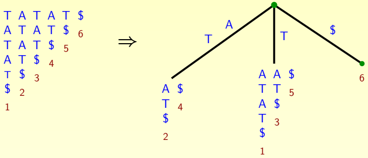
```
T A T A T $
A T A T $ 6
T A T $ 5
A T $ 4
T $ 3
$ 2
1
```

Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.

Two practical algorithms: McCreight (1976) and Ukkonen (1993).

A simpler algorithm: **Write-Only, Top-Down** (WOTD).

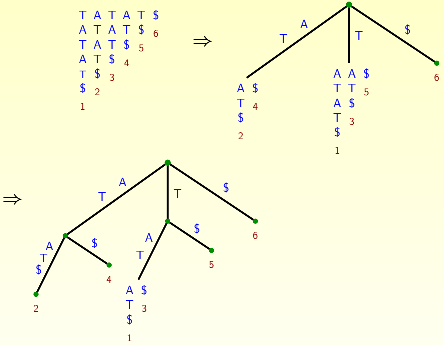


Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.

Two practical algorithms: McCreight (1976) and Ukkonen (1993).

A simpler algorithm: **Write-Only, Top-Down** (WOTD).

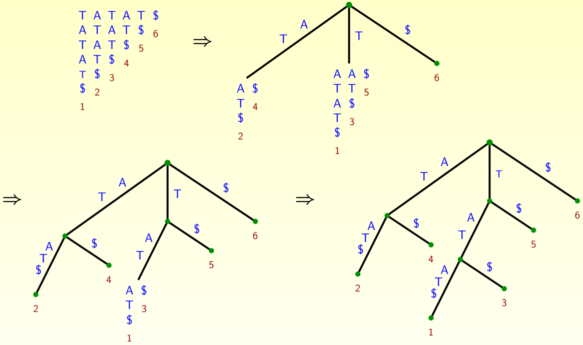


Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.

Two practical algorithms: McCreight (1976) and Ukkonen (1993).

A simpler algorithm: **Write-Only, Top-Down** (WOTD).

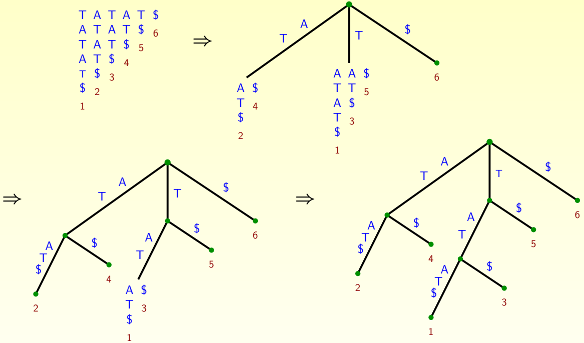


Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.

Two practical algorithms: McCreight (1976) and Ukkonen (1993).

A simpler algorithm: **Write-Only, Top-Down** (WOTD).



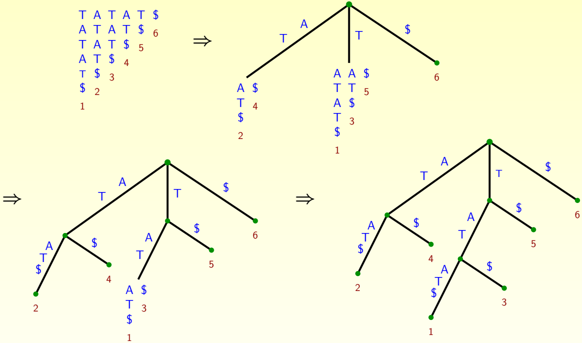
Analysis: $\mathcal{O}(n^2)$ worst-case, $\mathcal{O}(n \log n)$ expected time, $\mathcal{O}(n)$ space

Construction of suffix trees

Theorem [Weiner, 1973]: $T(S)$ can be constructed in $\mathcal{O}(n)$ time.

Two practical algorithms: McCreight (1976) and Ukkonen (1993).

A simpler algorithm: **Write-Only, Top-Down** (WOTD).



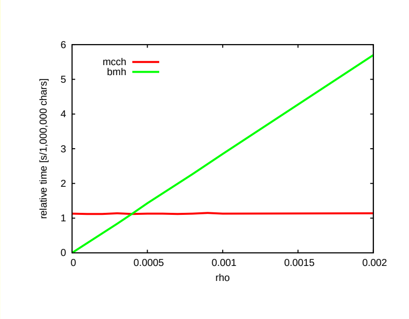
Analysis: $\mathcal{O}(n^2)$ worst-case, $\mathcal{O}(n \log n)$ expected time, $\mathcal{O}(n)$ space

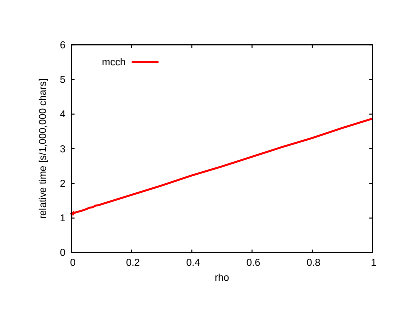
Note: The WOTD algorithm is well suited for a **lazy construction**.

Lazy construction of suffix trees

Experimental results:

index construction plus ρn pattern searches for $\rho \in [0, 1]$





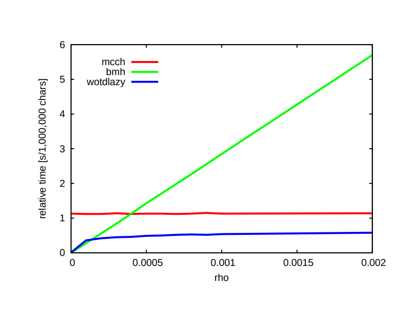
mcch = suffix tree (McCreight's algorithm with hash tables)

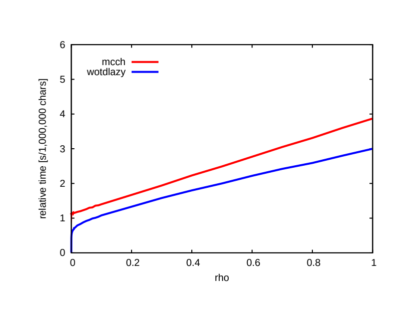
bmh = online search (Boyer-Moore-Horspool algorithm)

Lazy construction of suffix trees

Experimental results:

index construction plus ρn pattern searches for $\rho \in [0, 1]$





mcch = suffix tree (McCreight's algorithm with hash tables)

bmh = online search (Boyer-Moore-Horspool algorithm)

wotdlazy = suffix tree write-only top-down construction (lazy version)

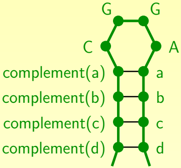
Index structures in biological sequence analysis

1. Introduction
2. Suffix trees
3. Affix trees
4. Suffix arrays
5. The *q*-Gram index
6. Summary and Conclusion

More complicated search: Inside-out

Define an RNA hairpin (in **HyPaL** syntax):

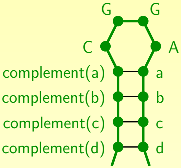
```
stem = .4
hairpin = @stem CGGA complement(@stem)
```



More complicated search: Inside-out

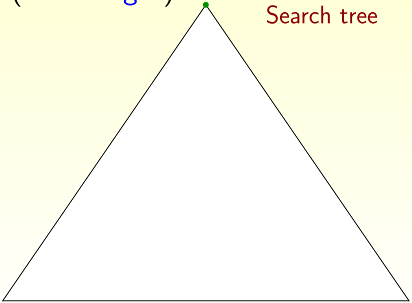
Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```



Search strategy:

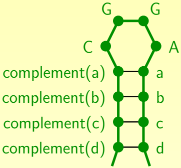
1. Find all exact matches of `CGGA` (left-to-right)



More complicated search: Inside-out

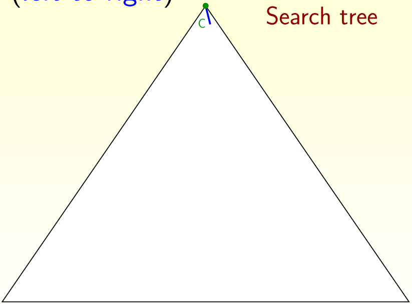
Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```



Search strategy:

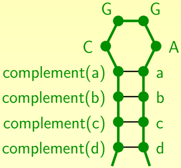
1. Find all exact matches of CGGA (left-to-right)



More complicated search: Inside-out

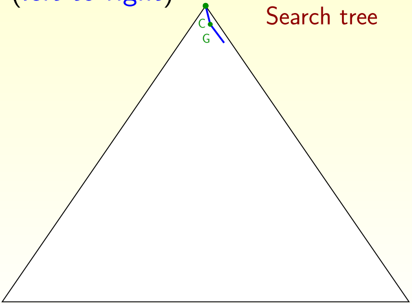
Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```



Search strategy:

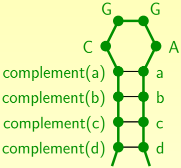
1. Find all exact matches of CGGA (left-to-right)



More complicated search: Inside-out

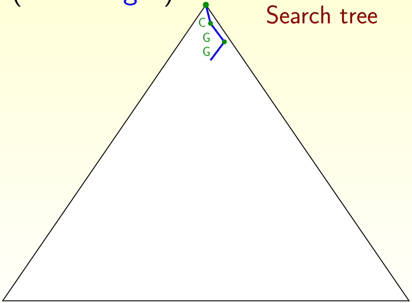
Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```



Search strategy:

1. Find all exact matches of `CGGA` (left-to-right)

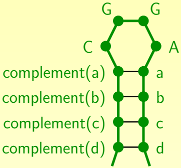


Search tree

More complicated search: Inside-out

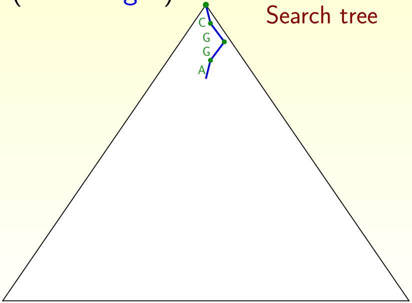
Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```



Search strategy:

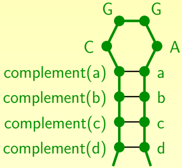
1. Find all exact matches of `CGGA` (left-to-right)



More complicated search: Inside-out

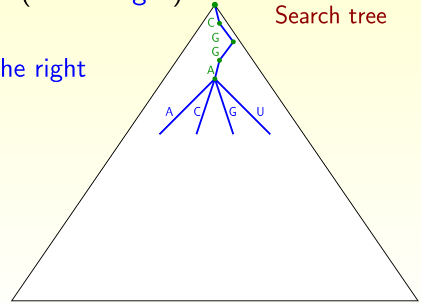
Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```

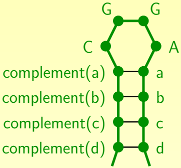


Search strategy:

1. Find all exact matches of `CGGA` (left-to-right)
2. Extend by
   - an arbitrary character *a* to the right



More complicated search: Inside-out

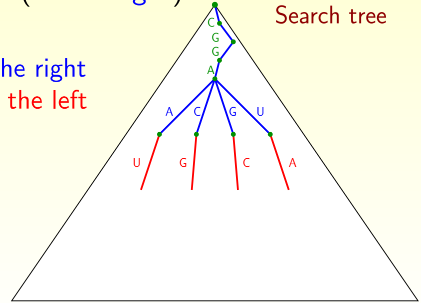


Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```

Search strategy:

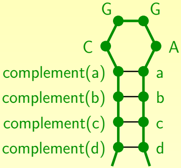
1. Find all exact matches of `CGGA` (left-to-right)
2. Extend by
   - an arbitrary character *a* to the right
   - *complementary* character to the left



More complicated search: Inside-out

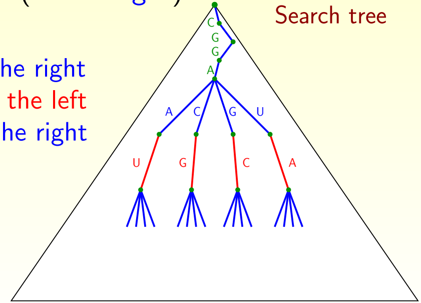
Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```



Search strategy:

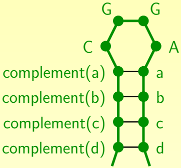
1. Find all exact matches of `CGGA` (left-to-right)
2. Extend by
   - an arbitrary character *a* to the right
   - *complementary* character to the left
   - an arbitrary character *b* to the right



More complicated search: Inside-out

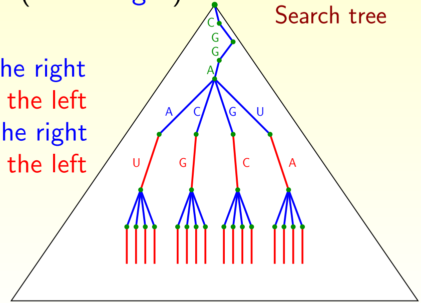
Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```



Search strategy:

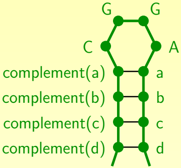
1. Find all exact matches of `CGGA` (left-to-right)
2. Extend by
   - an arbitrary character *a* to the right
   - *complementary* character to the left
   - an arbitrary character *b* to the right
   - *complementary* character to the left



More complicated search: Inside-out

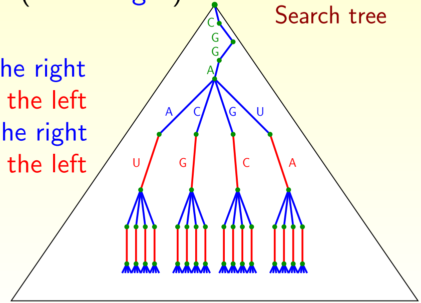
Define an RNA hairpin (in **HyPaL** syntax):

```
stem = .4
hairpin = @stem CGGA complement(@stem)
```



Search strategy:

1. Find all exact matches of `CGGA` (left-to-right)
2. Extend by
   - an arbitrary character *a* to the right
   - *complementary* character to the left
   - an arbitrary character *b* to the right
   - *complementary* character to the left
   - . . .

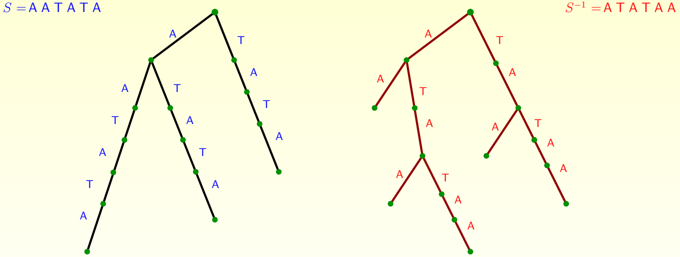

Search tree

Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree

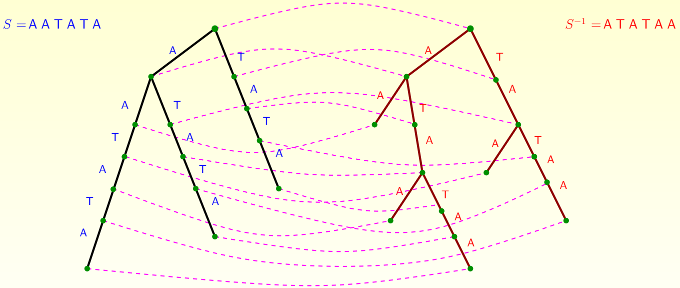
Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- Idea: Create the **atomic** suffix tree and reverse prefix tree.



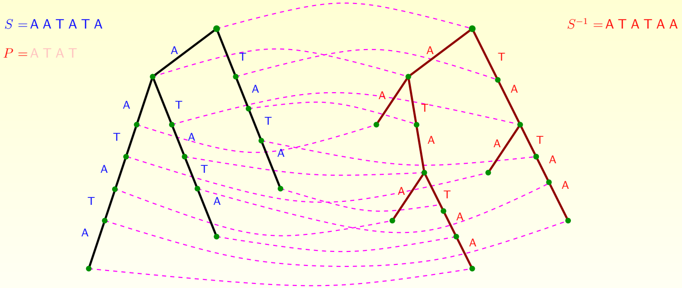
Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- Idea: Create the **atomic** suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



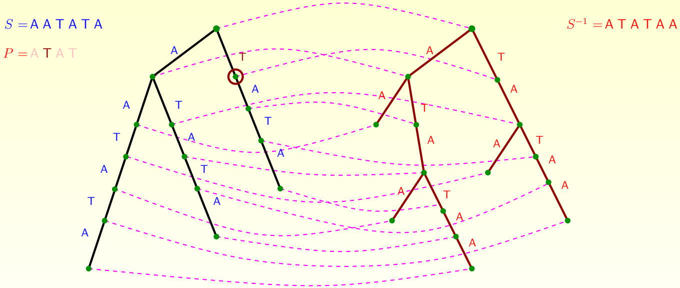
Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- Idea: Create the **atomic** suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



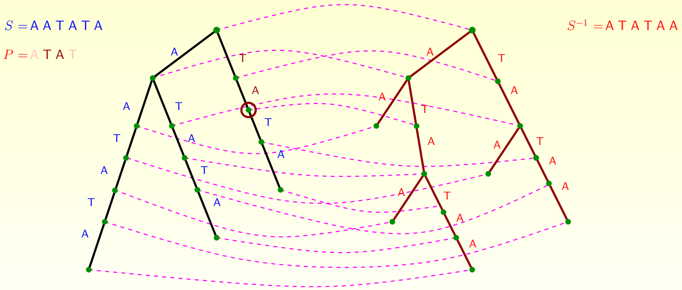
Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- Idea: Create the **atomic** suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



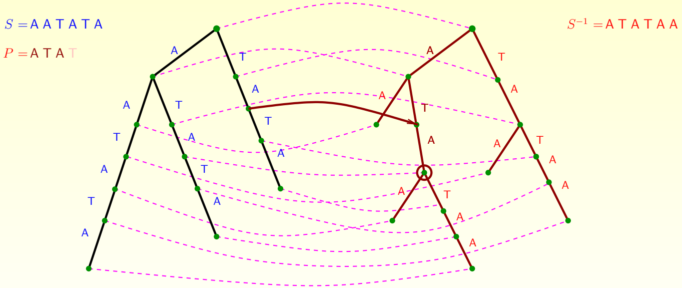
Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- Idea: Create the **atomic** suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



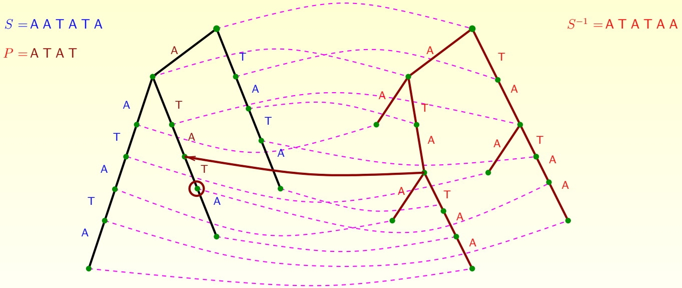
Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- Idea: Create the **atomic** suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



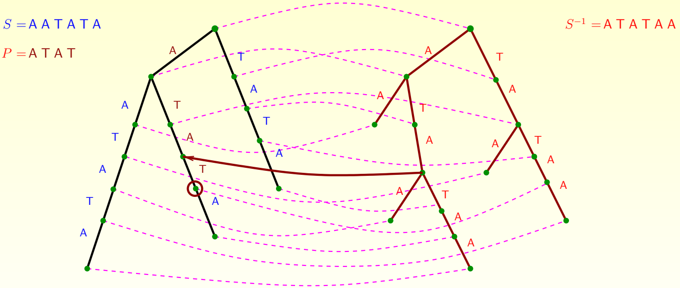
Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- Idea: Create the **atomic** suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



Towards a bi-directional data structure

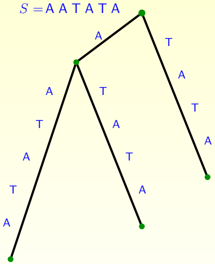
- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- Idea: Create the **atomic** suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



Problem: quadratic space

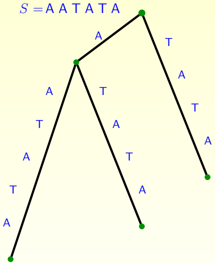
The affix tree

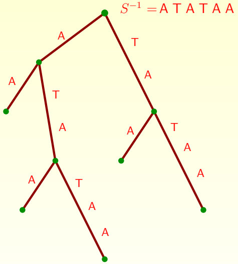
- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



The affix tree

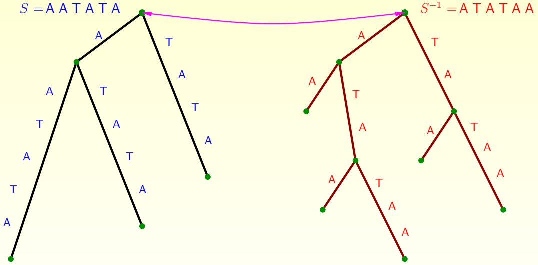
- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.





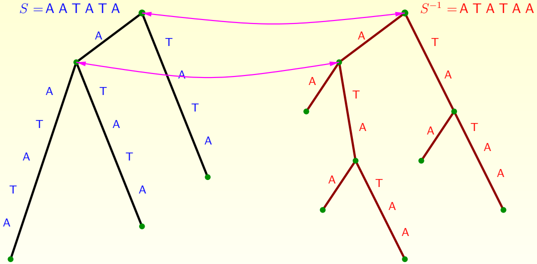
The affix tree

- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



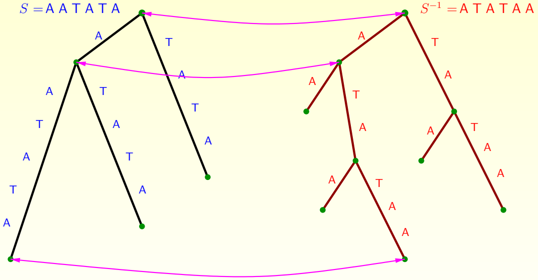
The affix tree

- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



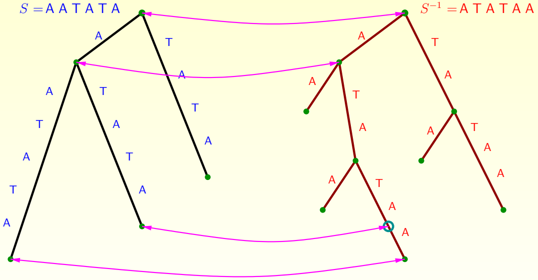
The affix tree

- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



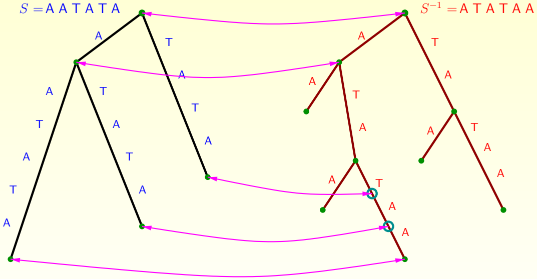
The affix tree

- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



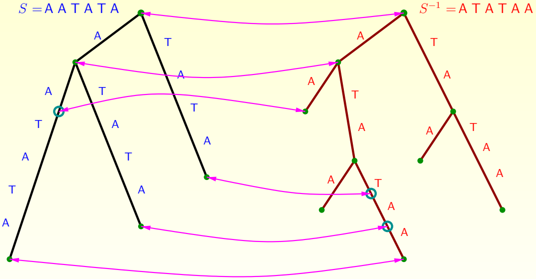
The affix tree

- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



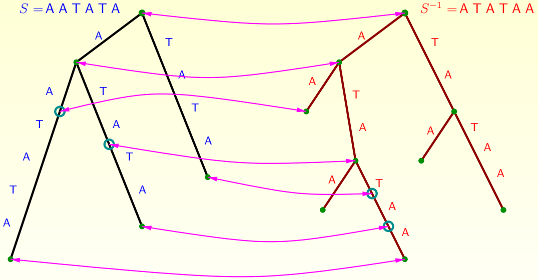
The affix tree

- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



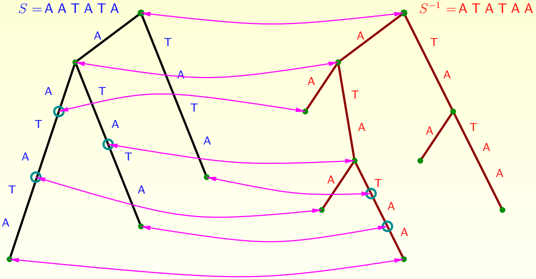
The affix tree

- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



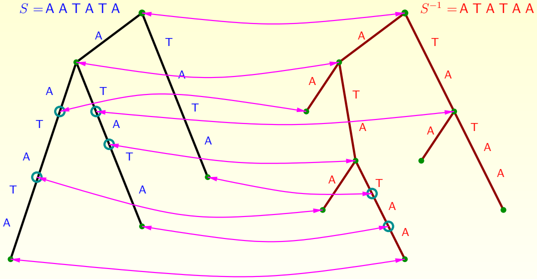
The affix tree

- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



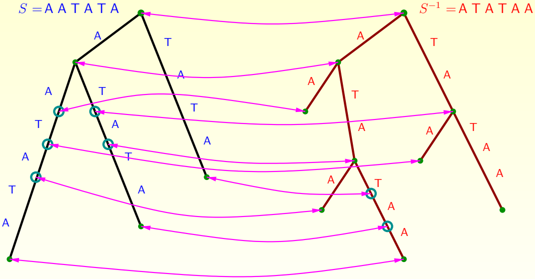
The affix tree

- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



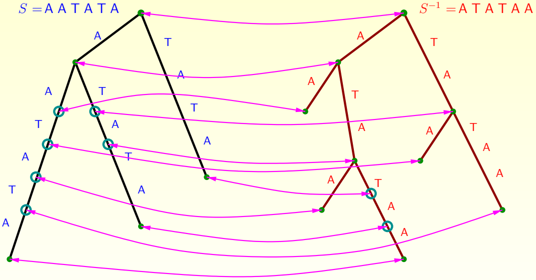
The affix tree

- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



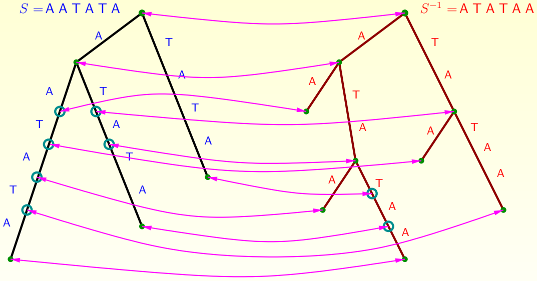
The affix tree

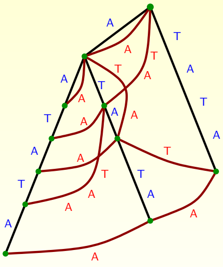
- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.



The affix tree

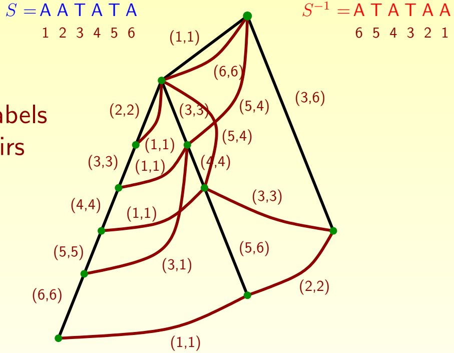
- To save space apply same idea to **compact** suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the **affix tree** of S.





Affix tree properties

- The affix tree of S requires $\mathcal{O}(n)$ space:
  - at most $2n-2$ nodes
  - at most $2n-4$ edges
  - as for suffix trees, edge labels can be represented by pairs of pointers into S

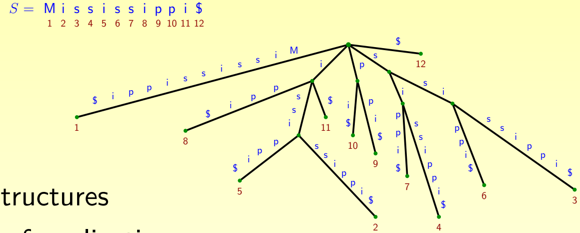


- The affix tree can be constructed in $\mathcal{O}(n)$ time and space (Maaß, CPM 2000).
- Supports all applications of suffix tree, and some more.

Index structures in biological sequence analysis

1. Introduction
2. Suffix trees
3. Affix trees
4. Suffix arrays
5. The *q*-Gram index
6. Summary and Conclusion

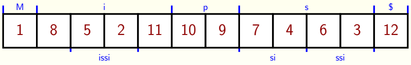
The suffix array



Suffix tree, affix tree:

- very flexible data structures
- support a ‘myriad’ of applications
- But: require considerable space in practice
  - Suffix tree: 10-20 bytes per text character
  - Affix tree: roughly twice as much

Alternative using less space: **suffix array** (Manber & Myers, 1993)

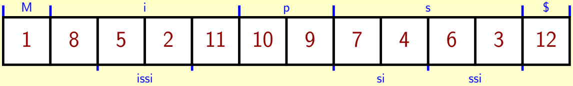


M	i				p		s				$
1	8	5	2	11	10	9	7	4	6	3	12
		issi					si		ssi		

The suffix array

$S =$ M i s s i s s i p p i $

M	i	s	s	i	s	s	i	p	p	i	$
1	2	3	4	5	6	7	8	9	10	11	12



1	8	5	2	11	10	9	7	4	6	3	12

- Array containing suffix numbers, lexicographically sorted by their suffixes
- Space usage: $4n$ bytes
- Query time: $\mathcal{O}(|P| \log n)$ time (or $\mathcal{O}(|P| + \log n)$ with tricks)
- Technique to simulate all suffix tree operations: **enhanced suffix array** (Abouelhoda *et al.*, JDA 2004)

Construction of suffix arrays

(a) Read leaf numbers of suffix tree
$\rightarrow \mathcal{O}(n)$ time

(b) Direct construction of suffix arrays:
simple algorithms use $\mathcal{O}(n^2)$ or $\mathcal{O}(n \log n)$ time

- Kim *et al.*, CPM 2003
- Ko & Aluru, CPM 2003
- Kärkkäinen & Sanders, ICALP 2003

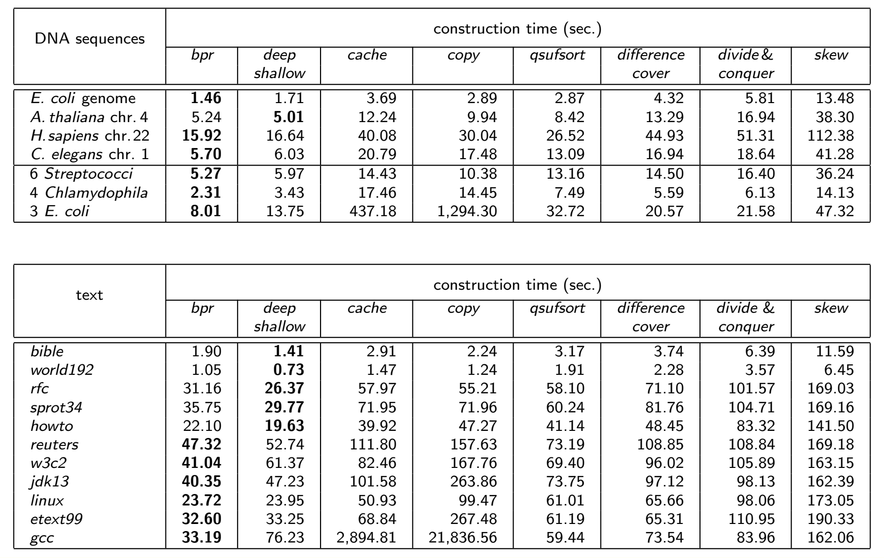
$\rightarrow \mathcal{O}(n)$ time

(c) Practical algorithms have worse time complexities

- *deep shallow sorting* (Manzini & Ferragina, *Algorithmica* 2004)
- Bucket-pointer refinement (Schürmann & JS, *SP&E* 2006)

$\rightarrow \mathcal{O}(n^2)$ time in worst case, **much better in practice**

The bucket-pointer refinement algorithm



DNA sequences	construction time (sec.)							
	bpr	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide & conquer*	*skew*
E. coli genome	**1.46**	1.71	3.69	2.89	2.87	4.32	5.81	13.48
A. thaliana chr. 4	5.24	**5.01**	12.24	9.94	8.42	13.29	16.94	38.30
H. sapiens chr. 22	**15.92**	16.64	40.08	30.04	26.52	44.93	51.31	112.38
C. elegans chr. 1	**5.70**	6.03	20.79	17.48	13.09	16.94	18.64	41.28
6 *Streptococci*	**5.27**	5.97	14.43	10.38	13.16	14.50	16.40	36.24
4 *Chlamydophila*	**2.31**	3.43	17.46	14.45	7.49	5.59	6.13	14.13
3 *E. coli*	**8.01**	13.75	437.18	1,294.30	32.72	20.57	21.58	47.32

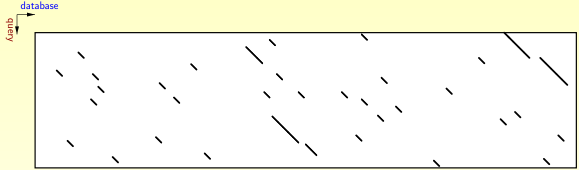
text	construction time (sec.)							
	bpr	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide & conquer*	*skew*
bible	1.90	**1.41**	2.91	2.24	3.17	3.74	6.39	11.59
world192	1.05	**0.73**	1.47	1.24	1.91	2.28	3.57	6.45
rfc	31.16	**26.37**	57.97	55.21	58.10	71.10	101.57	169.03
sprot34	35.75	**29.77**	71.95	71.96	60.24	81.76	104.71	169.16
howto	22.10	**19.63**	39.92	47.27	41.14	48.45	83.32	141.50
reuters	**47.32**	52.74	111.80	157.63	73.19	108.85	108.84	169.18
w3c2	**41.04**	61.37	82.46	167.76	69.40	96.02	105.89	163.15
jdk13	**40.35**	47.23	101.58	263.86	73.75	97.12	98.13	162.39
linux	**23.72**	23.95	50.93	99.47	61.01	65.66	98.06	173.05
etext99	**32.60**	33.25	68.84	267.48	61.19	65.31	110.95	190.33
gcc	**33.19**	76.23	2,894.81	21,836.56	59.44	73.54	83.96	162.06

Application to approximate DNA matching

QUASAR: **Q**-Gram based database search **U**sing **A** **S**uffix **AR**ray

(Burkhardt *et al.*, RECOMB 1999)

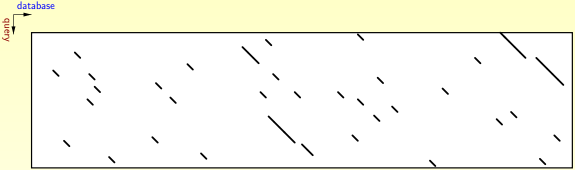
Idea: Search **dot-plot** for regions with many *q*-Grams.



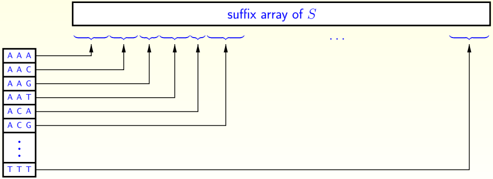
Application to approximate DNA matching

QUASAR: **Q**-Gram based database search **U**sing **A** **S**uffix **AR**ray
(Burkhardt *et al.*, RECOMB 1999)

Idea: Search **dot-plot** for regions with many q-Grams.



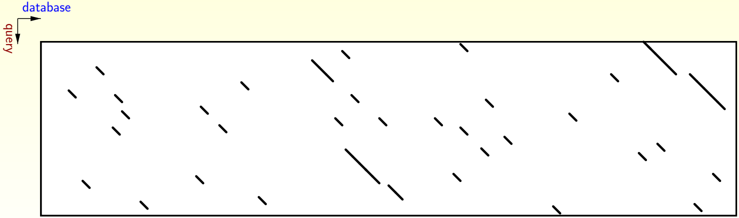
Use **suffix array** to locate all occurrences of a q-Gram in the database S:



QUASAR algorithm

Algorithm: Filter database for relevant blocks.

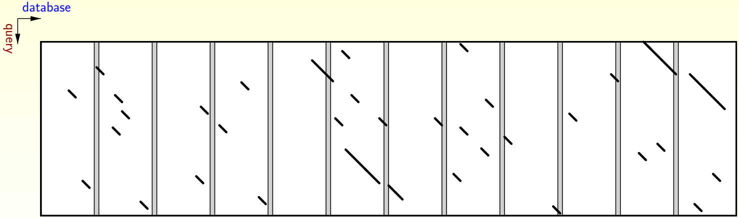
1. Divide database into overlapping blocks.
2. Shift window of certain size over query;
   for each database block find the number
   of matching q-Grams from the current window;
   if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$,
   consider this block *relevant*.
3. Test the relevant blocks with a more complex method.



QUASAR algorithm

Algorithm: Filter database for relevant blocks.

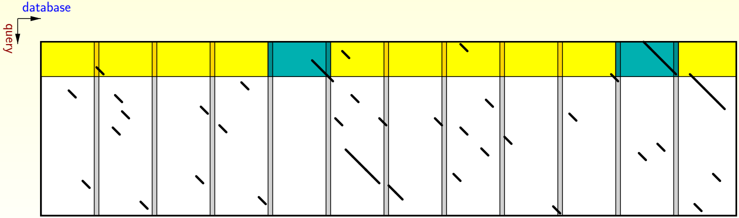
1. Divide database into overlapping blocks.
2. Shift window of certain size over query;
   for each database block find the number
   of matching q-Grams from the current window;
   if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$,
   consider this block *relevant*.
3. Test the relevant blocks with a more complex method.



QUASAR algorithm

Algorithm: Filter database for relevant blocks.

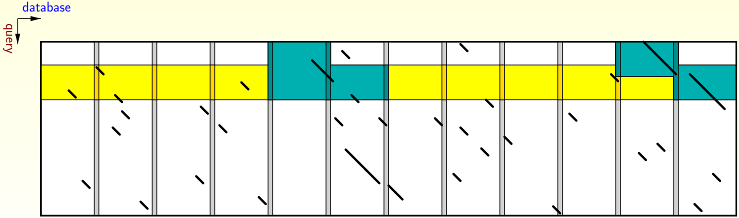
1. Divide database into overlapping blocks.
2. Shift window of certain size over query;
   for each database block find the number
   of matching q-Grams from the current window;
   if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$,
   consider this block *relevant*.
3. Test the relevant blocks with a more complex method.



QUASAR algorithm

Algorithm: Filter database for relevant blocks.

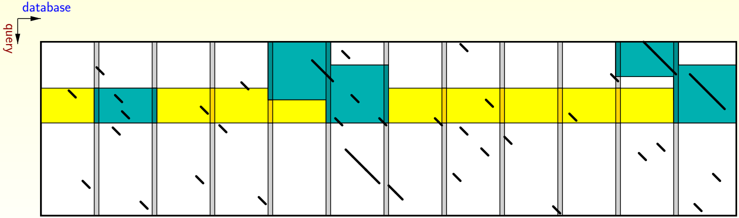
1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching q-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block *relevant*.
3. Test the relevant blocks with a more complex method.



QUASAR algorithm

Algorithm: Filter database for relevant blocks.

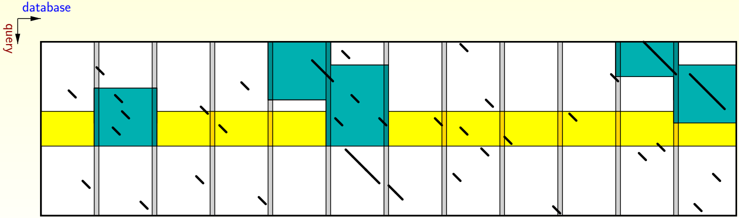
1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching q-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block *relevant*.
3. Test the relevant blocks with a more complex method.



QUASAR algorithm

Algorithm: Filter database for relevant blocks.

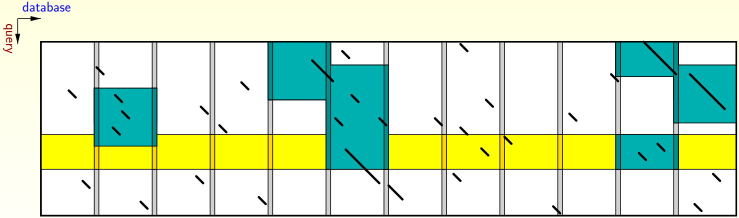
1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching q-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block *relevant*.
3. Test the relevant blocks with a more complex method.



QUASAR algorithm

Algorithm: Filter database for relevant blocks.

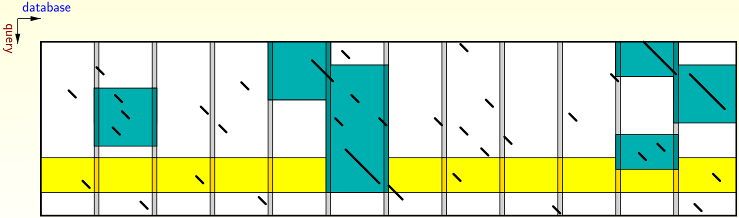
1. Divide database into overlapping blocks.
2. Shift window of certain size over query;
   for each database block find the number
   of matching q-Grams from the current window;
   if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$,
   consider this block *relevant*.
3. Test the relevant blocks with a more complex method.



QUASAR algorithm

Algorithm: Filter database for relevant blocks.

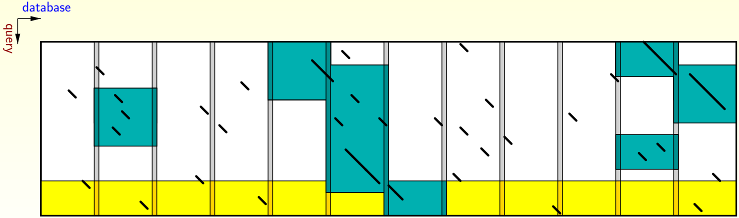
1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching *q*-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block *relevant*.
3. Test the relevant blocks with a more complex method.



QUASAR algorithm

Algorithm: Filter database for relevant blocks.

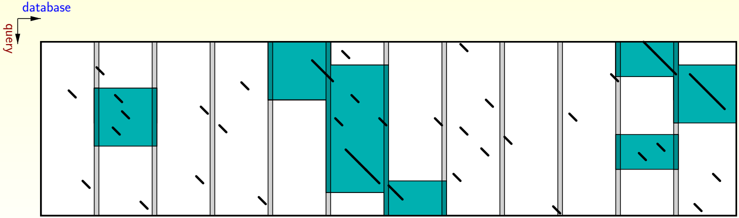
1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching q-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block *relevant*.
3. Test the relevant blocks with a more complex method.



QUASAR algorithm

Algorithm: Filter database for relevant blocks.

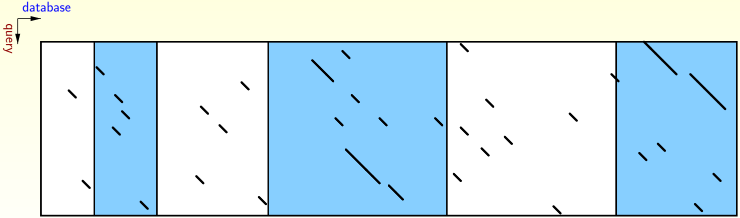
1. Divide database into overlapping blocks.
2. Shift window of certain size over query;
   for each database block find the number
   of matching q-Grams from the current window;
   if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$,
   consider this block *relevant*.
3. Test the relevant blocks with a more complex method.



QUASAR algorithm

Algorithm: Filter database for relevant blocks.

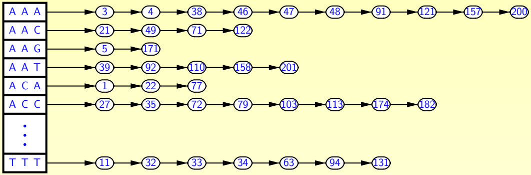
1. Divide database into overlapping blocks.
2. Shift window of certain size over query; for each database block find the number of matching q-Grams from the current window; if this number is at least $T(n, q, e) := n + 1 - q(e + 1)$, consider this block *relevant*.
3. Test the relevant blocks with a more complex method.



Two ideas for improving QUASAR

(Rasmussen, JS & Myers, *JCB* 2006)

1. Use a simpler index structure: **q-Gram index**



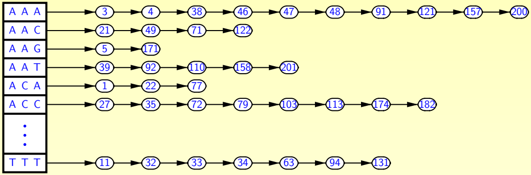
A A A → 3 → 4 → 38 → 46 → 47 → 48 → 91 → 121 → 157 → 200
A A C → 21 → 49 → 71 → 122
A A G → 5 → 171
A A T → 39 → 92 → 110 → 158 → 201
A C A → 1 → 22 → 77
A C C → 27 → 35 → 72 → 79 → 103 → 113 → 174 → 182
⋮
T T T → 11 → 32 → 33 → 34 → 63 → 94 → 131

Analysis: simple $\mathcal{O}(n)$ construction time

Two ideas for improving QUASAR

(Rasmussen, JS & Myers, *JCB* 2006)

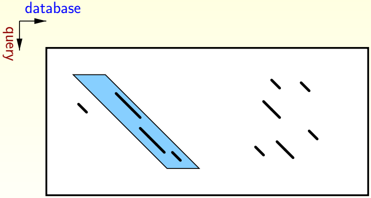
1. Use a simpler index structure: **q-Gram index**



A A A 3 4 38 46 47 48 91 121 157 200
A A C 21 49 71 122
A A G 5 171
A A T 39 92 110 158 201
A C A 1 22 77
A C C 27 35 72 79 103 113 174 182
⋮
T T T 11 32 33 34 63 94 131

Analysis: simple $\mathcal{O}(n)$ construction time

2. Reduce size of the relevant regions: parallelograms

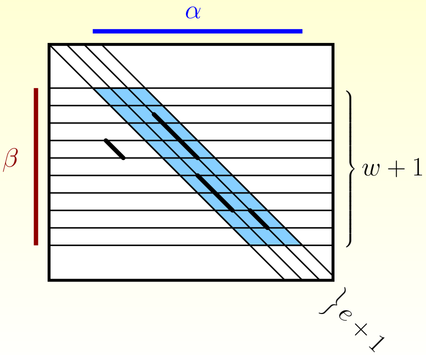


$\longrightarrow$ need proper filter criteria

Filter criteria for parallelograms

Define ϵ-match: match of database substring α and query substring β of length $n \geq n_0$ with at most $e := \lfloor \epsilon n \rfloor$ errors.

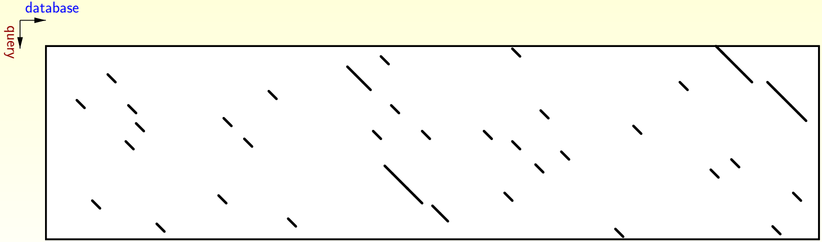
Given: q, n_0, ϵ
Compute: threshold τ such that for every ϵ-match there exists a $w \times e$ parallelogram containing at least τ q-Gram hits.



The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

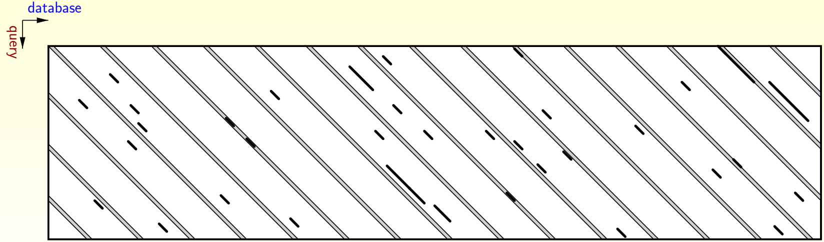
1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method,
   e.g. X-drop extension.



The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

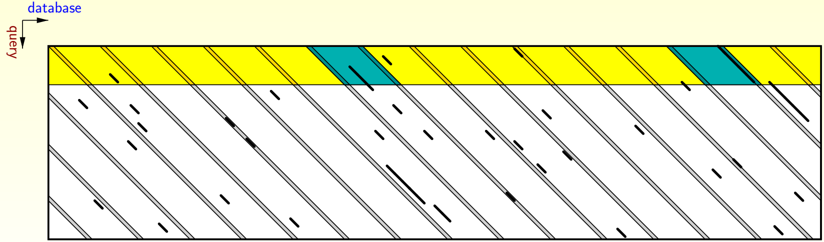
1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method,
   e.g. X-drop extension.



The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

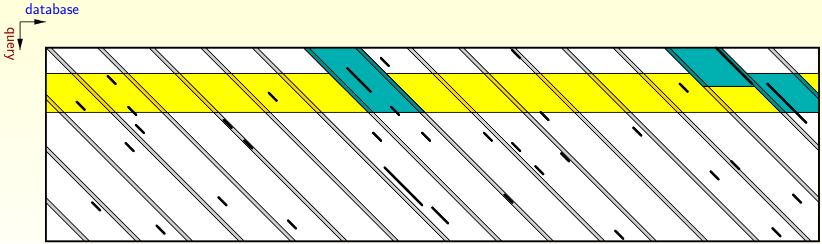
1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method, e.g. X-drop extension.



The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

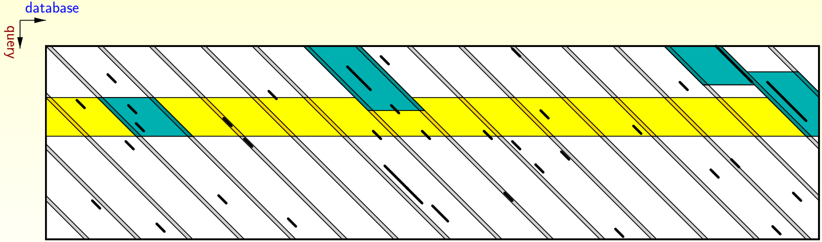
1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method,
   e.g. X-drop extension.



The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

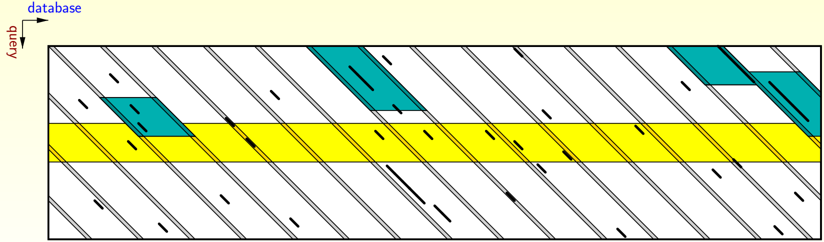
1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method,
   e.g. X-drop extension.



The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

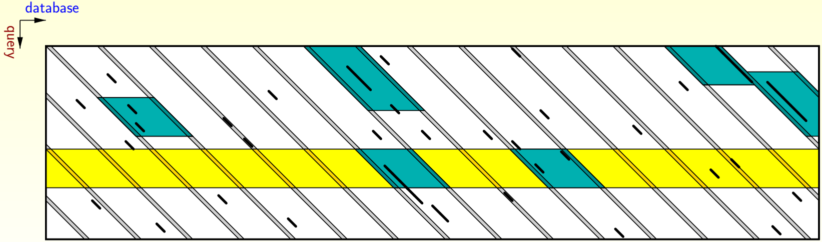
1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method,
   e.g. X-drop extension.



The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

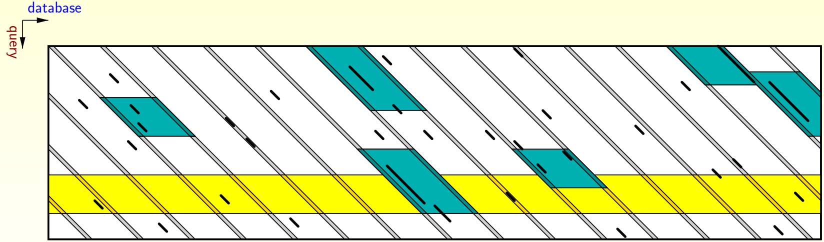
1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method, e.g. X-drop extension.



The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

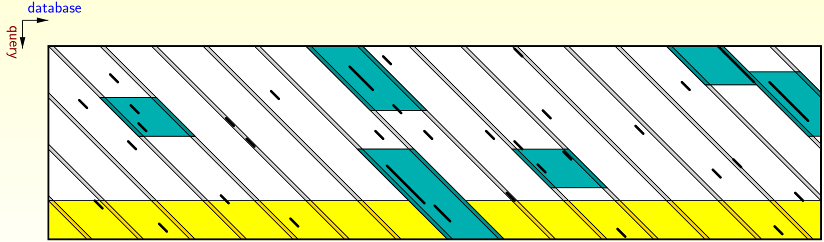
1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method, e.g. X-drop extension.



The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

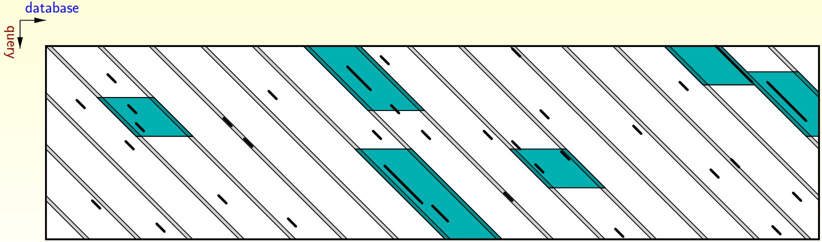
1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method, e.g. X-drop extension.



The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

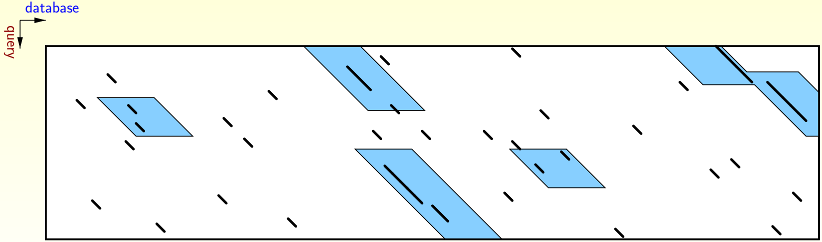
1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method,
   e.g. X-drop extension.



The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

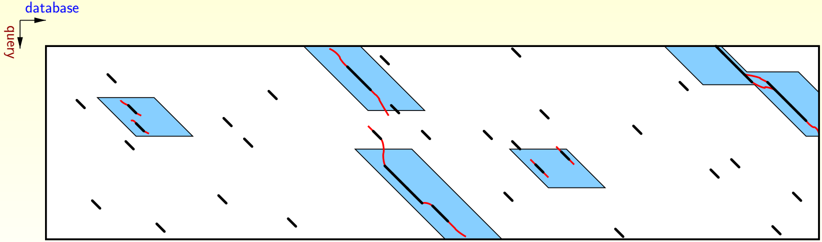
1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method, e.g. X-drop extension.



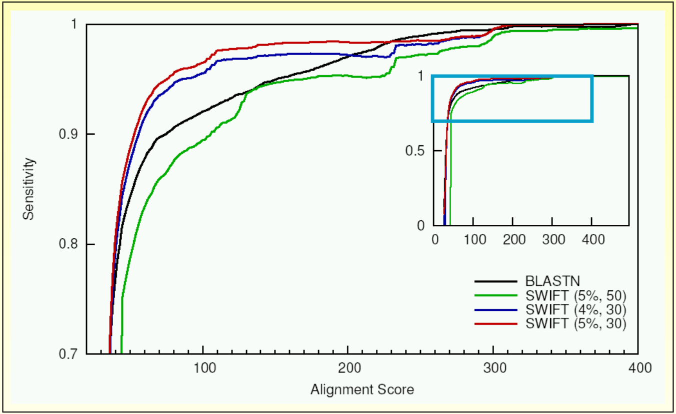
The SWIFT algorithm

Algorithm: Filter database for relevant parallelograms.

1. Divide database into overlapping diagonal blocks of width e.
2. Shift window of size w over query;
3. Find $w \times e$ parallelograms with more than τ q-grams.
4. Test these with a more complex method, e.g. X-drop extension.



Experimental results



	SSEARCH	BLASTN	SWIFT		
parameters (ϵ, n_0)	—	—	$(5\%, 50)$	$(4\%, 30)$	$(5\%, 30)$
running time	8h	773 s	18 s	29 s	35 s
filtration ratio	—	—	6.5e-6	4.5e-6	5.4e-6

Index structures in biological sequence analysis

1. Introduction
2. Suffix trees
3. Affix trees
4. Suffix arrays
5. The q-Gram index
6. Summary and Conclusion

Summary and Conclusion

- Handling large amounts of DNA sequence data is challenging
- Standard methods do not apply in bioinformatics
- Data structures discussed: suffix tree, affix tree, suffix array, q-Gram index
- Popular approach: filtration
- Tools that use index structures: MUMmer, Genalyzer, SSAHA, BLAT, SWIFT, PatternHunter, ...

- Simplicity is often a key element of practical algorithms!

Acknowledgments

Suffix trees

- Robert Giegerich (Bielefeld)
- Stefan Kurtz (Hamburg)

Repeats

- Dan Gusfield (Davis)
- Enno Ohlebusch (Ulm)
- Chris Schleiermacher (Köln)
- Jomuna Choudhuri (Mannheim)

Affix trees

- Moritz Maaß (München)

Suffix arrays

- Klaus-Bernd Schürmann (Bielefeld)

SWIFT

- Kim Roland Rasmussen (Bielefeld)
- Gene Myers (HHMI, Janelia Farm)

The end.