Suffix trees, Affix trees, and some of their applications

Jens Stoye

Genome Informatics Group, Faculty of Technology
and
Institute of Bioinformatics, Center of Biotechnology
Bielefeld University, Germany

CeBiTec
(Center of Biotechnology)

Institute of Bioinformatics

- Practical Computer Science
- Bio- and Medical Informatics
- Genome Informatics
- Analysis of Mass Spectra

- Bibiserv Tools
- Bibiserv Media

- Theoretical Bioinformatics
- Biomathematics

Graduate School

- 2 scientific groups
- 2 mentors
- 21 PhD grants (up to 3 years each)

Bioinformatics Resource

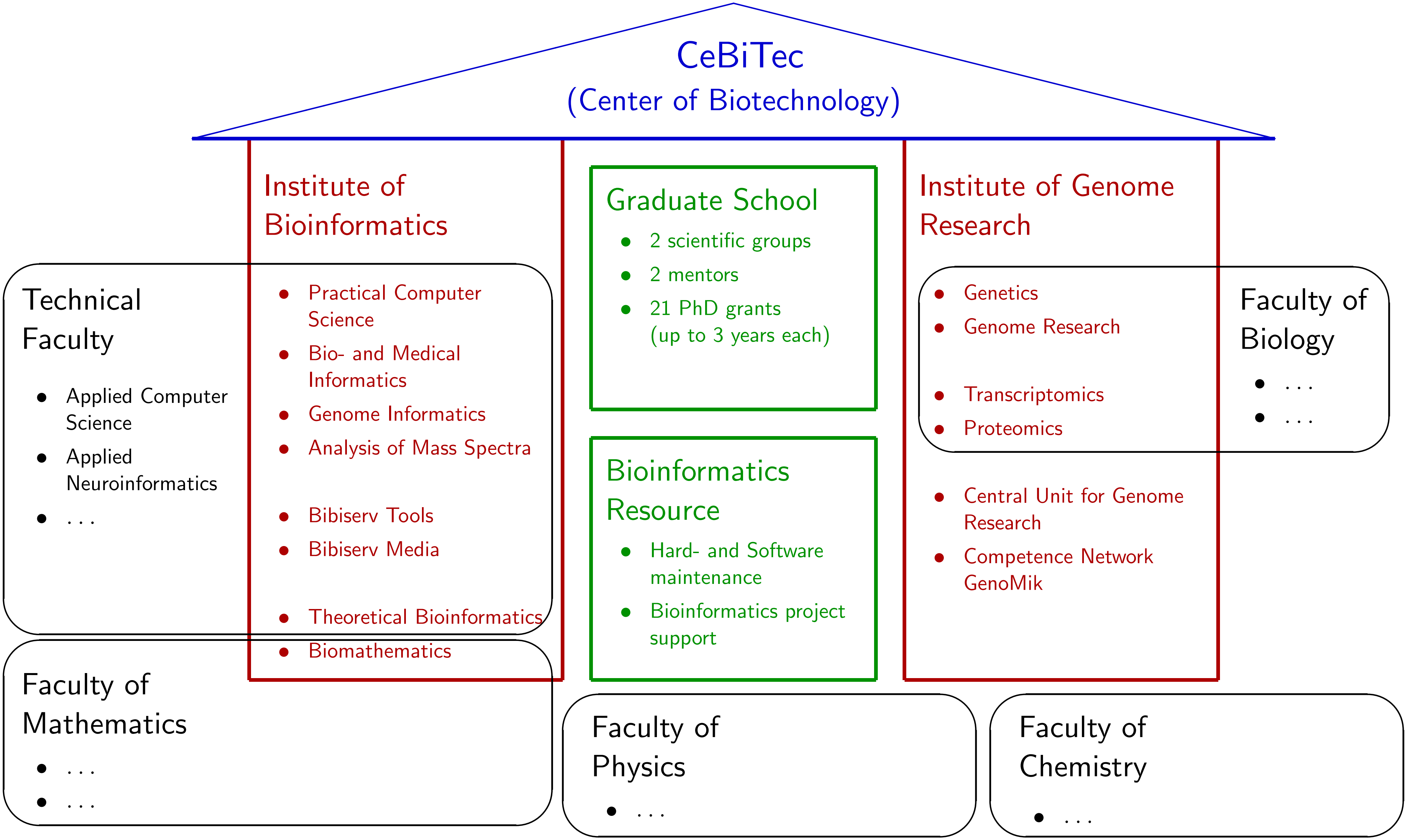
- Hard- and Software maintenance
- Bioinformatics project support

Institute of Genome Research

- Genetics
- Genome Research

- Transcriptomics
- Proteomics

- Central Unit for Genome Research
- Competence Network GenoMik



CeBiTec
(Center of Biotechnology)

Institute of Bioinformatics

- Practical Computer Science
- Bio- and Medical Informatics
- Genome Informatics
- Analysis of Mass Spectra

- Bibiserv Tools
- Bibiserv Media

- Theoretical Bioinformatics
- Biomathematics

Graduate School

- 2 scientific groups
- 2 mentors
- 21 PhD grants (up to 3 years each)

Bioinformatics Resource

- Hard- and Software maintenance
- Bioinformatics project support

Institute of Genome Research

- Genetics
- Genome Research

- Transcriptomics
- Proteomics

- Central Unit for Genome Research
- Competence Network GenoMik

Technical Faculty

- Applied Computer Science
- Applied Neuroinformatics
- . . .

Faculty of Biology

- . . .
- . . .

Faculty of Mathematics

- . . .
- . . .

Faculty of Physics

- . . .

Faculty of Chemistry

- . . .

LOKALARTIKEL

29.04. 2003

Die Uni wächst weiter

Neues Laborgebäude für Naturwissenschaften / Geplanter Baubeginn 2004

Bielefeld. Die Universität Bielefeld bekommt mehr Platz. Im Nordwesten der Hochschule soll ein neues Laborgebäude für Genomforschung, Biophysik und -informatik entstehen. Baubeginn könnte nach Informationen der Redaktion des Haller Kreisblatts im Frühjahr 2004 sein.

Standort des 44 Meter langen und 20 Meter breiten fünfgeschossigen Gebäudes ist ein Teil des Parkplatzes neben den Sportanlagen im Nordwesten (Wellensiek). Zunächst sollen Labore und Büros mit einer Nutzfläche von 2.100 Quadratmetern gebaut werden, später könnte die Fläche auf 3.300 Quadratmeter vergrößert werden. Bauvolumen der ersten Bauphase: zwölf Millionen Euro. Die Planung liegt beim Bau- und Liegenschaftsbetrieb des Landes Nordrhein-Westfalen.

Diese Erweiterung sei "dringend notwendig", sagt Hartmut Krauß, Planungsdezernent der Universität. Für die seit zehn Jahren wachsende Technische Fakultät sind Labors mit höchstem Standard geplant. Der Neubau ist eine Reaktion auf die Raumnot der aufstrebenden Naturwissenschaften. Das Land Nordrhein-Westfalen hat das Bielefelder Projekt ganz oben auf seine Prioritätenliste gesetzt.

Schwerpunkte der Biologen im neuen Erweiterungsbau werden die mikrobiologische und Pflanzen-Genomforschung sein. Die Biophysik wird vor allem Nanoforschung betreiben. Für ultraempfindliche Mikroskope etwa müssen besondere Raumbedingungen geschaffen werden.

Mit dem Hauptgebäude der 30 Jahre jungen Universität, den so genannten Zähnen, wird der Neubau zunächst nicht verbunden sein, wie Uni-Kanzler Hans-Jürgen Simm erklärt. Für den zweiten Bauabschnitt sei diese Verbindung im Gespräch.

VON ELMAR KRAMER

BILD: Die Uni-Zähne bei Nacht: Läuft alles nach Plan, bekommen sie im nächsten Jahr Zuwachs. FOTO: ANDREAS FRÜCHT

Overview: Suffix trees, affix trees, ...

- Introduction: Pattern matching in biosequence analysis
- A flexible index structure for sequence analysis: The suffix tree
  - Basic definitions
  - Application 1: Repeats, tandem repeats
  - Application 2: Multiple Genome Aligner (MGA)
- An even more flexible data structure: The affix tree
  - Basic definitions
  - Application: Search for palindromic patterns
- Conclusion

Overview: Suffix trees, affix trees, ...

- Introduction: Pattern matching in biosequence analysis
- A flexible index structure for sequence analysis: The suffix tree
  - Basic definitions
  - Application 1: Repeats, tandem repeats
  - Application 2: Multiple Genome Aligner (MGA)
- An even more flexible data structure: The affix tree
  - Basic definitions
  - Application: Search for palindromic patterns
- Conclusion

Pattern matching in biological sequence analysis

Finding *known* patterns: (exact/approximate)

- Search for homologous proteins
  - assumption: similar sequence ⟶ similar structure ⟶ similar function
- Search for given sequence or structural pattern
  - mapping of *expressed sequence tags* (ESTs) on genomic DNA
  - palindromic or other RNA structural patterns
- *Known repeats* or *low complexity regions* (for further exclusion from analysis)

Pattern matching in biological sequence analysis

Finding *known* patterns: (exact/approximate)

- Search for homologous proteins
  - assumption: similar sequence → similar structure → similar function
- Search for given sequence or structural pattern
  - mapping of *expressed sequence tags* (ESTs) on genomic DNA
  - palindromic or other RNA structural patterns
- *Known repeats* or *low complexity regions* (for further exclusion from analysis)

Finding *structural* patterns: (exact/approximate)

- *Ab initio* gene prediction (start/stop codons, exons/introns)
- Search for over-/underrepresented substrings/-sequences, for example
  - unknown promoter binding sites
  - repeats, tandem repeats
  - possible DNA methylation sites
- Calculation of RNA secondary structure

Requirements for computational tools

- Efficiency: linear in space and time
- Flexibility: applicable to as many problems as possible
- Statistical assessment of significance of results
- Visualization

Requirements for computational tools

- Efficiency: linear in space and time
- Flexibility: applicable to as many problems as possible
- Statistical assessment of significance of results
- Visualization

For routine tasks, even linear time is intolerable → index

Many indices for massive sequence data use the property that the text is partitioned into words (e.g. natural language, syntactic tags).

Genomic data is not divided into obvious “words”.

We need an index that allows access to any substring of the text.

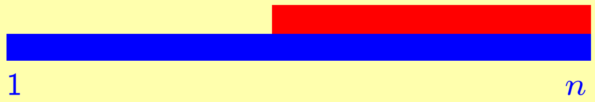
→ Suffix Tree

Overview

- Introduction: Pattern matching in biosequence analysis
- A flexible index structure for sequence analysis: The suffix tree
  - Basic definitions
  - Application 1: Repeats, tandem repeats
  - Application 2: Multiple Genome Aligner (MGA)
- An even more flexible data structure: The affix tree
  - Basic definitions
  - Application: Search for palindromic patterns
- Conclusion

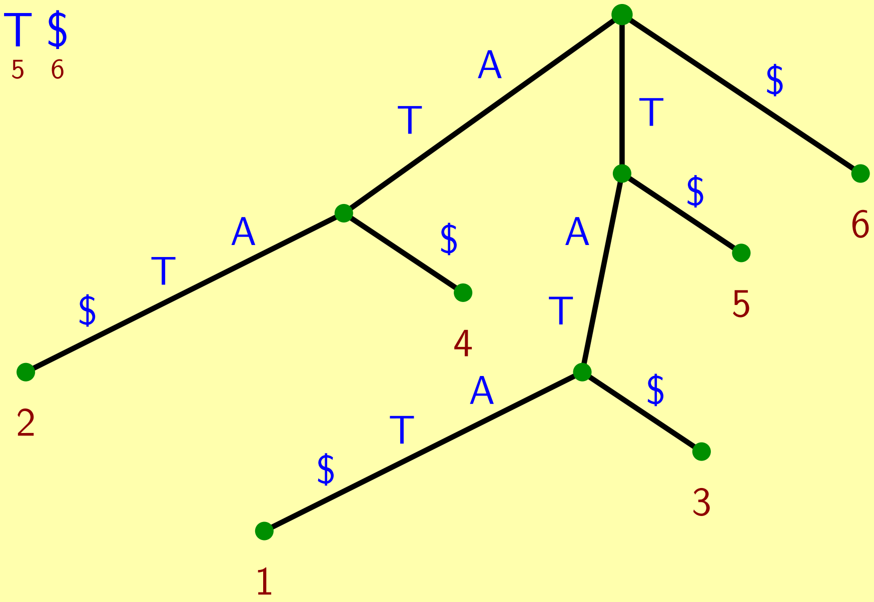
Suffix Tree: Definition

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



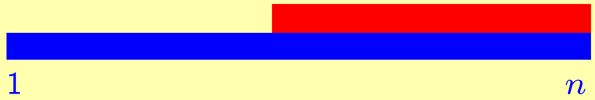
- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - all edges leaving a node begin with different characters and
  - the paths from the root to the leaves represent all the suffixes of S.

$S =$ T A T A T \$ (positions 1 2 3 4 5 6)



Suffix Tree: Definition

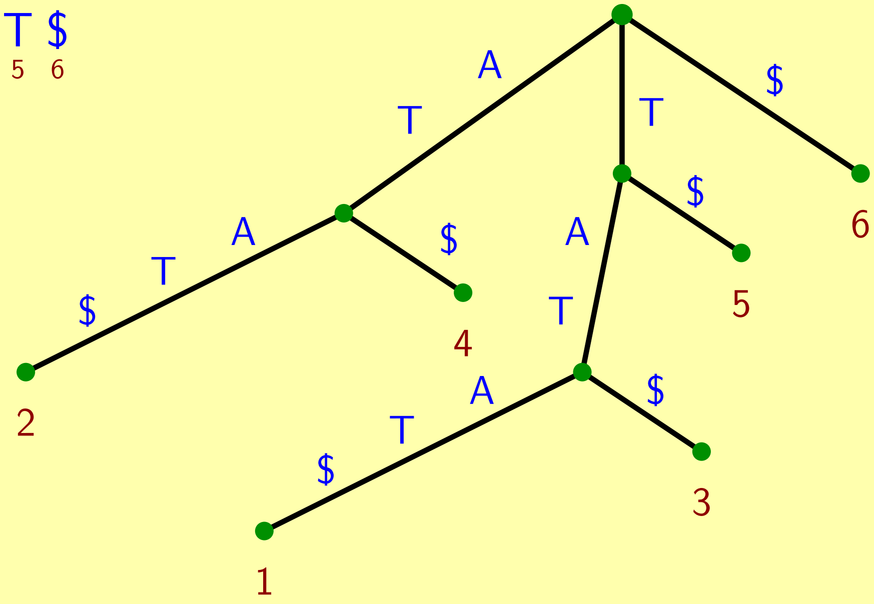
- A suffix of a string S of length n is a substring of S that ends at position n.



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

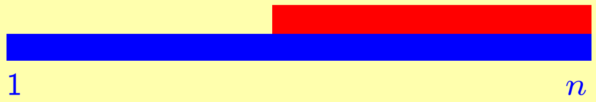
$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\mathsf{\$}}$

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



Suffix Tree: Definition

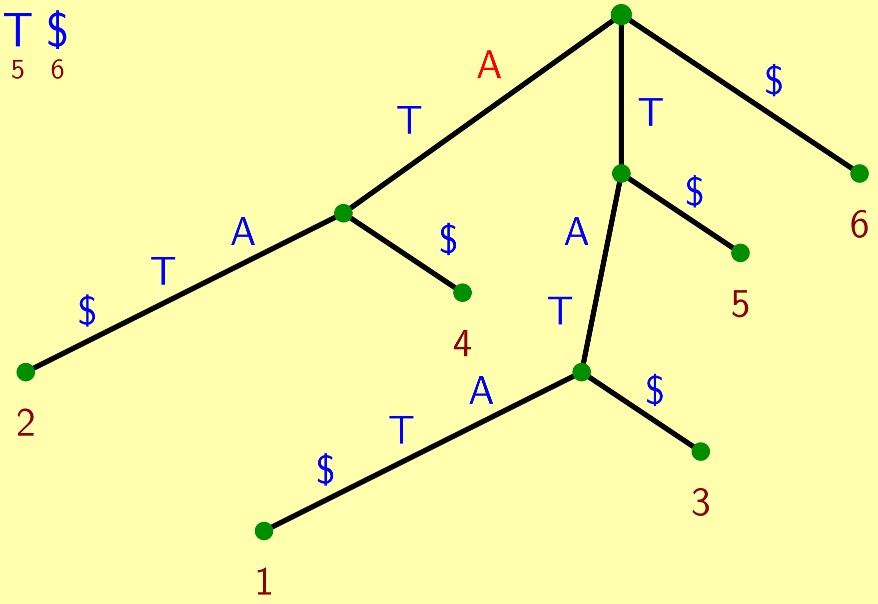
- A suffix of a string S of length n is a substring of S that ends at position n.



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

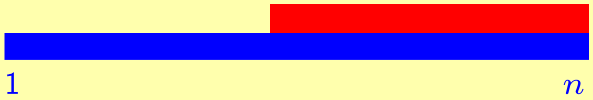
$S = $ T A T A T \$ (positions 1 2 3 4 5 6)

$P = $ A T A



Suffix Tree: Definition

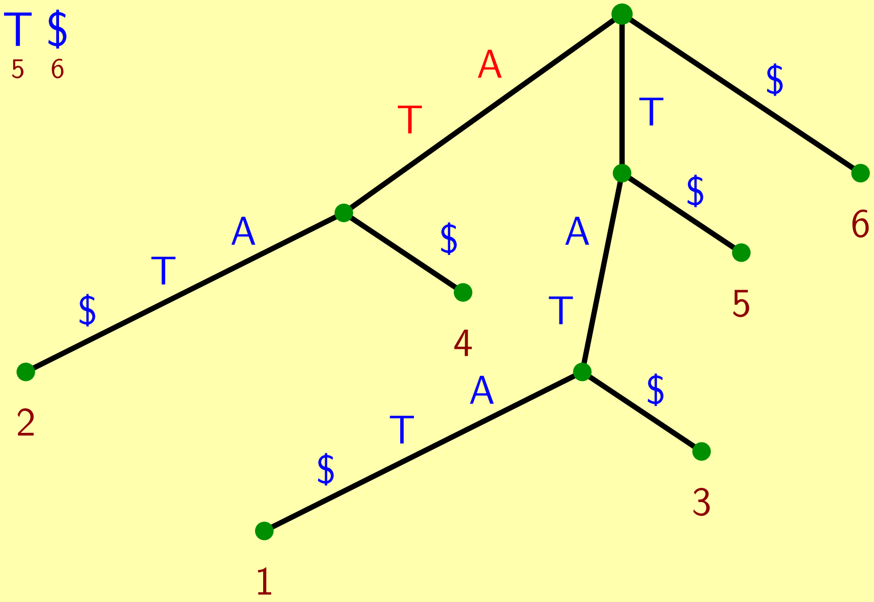
- A suffix of a string S of length n is a substring of S that ends at position n.



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

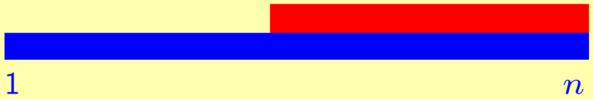
$S = \underset{1}{\text{T}}\ \underset{2}{\text{A}}\ \underset{3}{\text{T}}\ \underset{4}{\text{A}}\ \underset{5}{\text{T}}\ \underset{6}{\text{\$}}$

$P =$ A T A



Suffix Tree: Definition

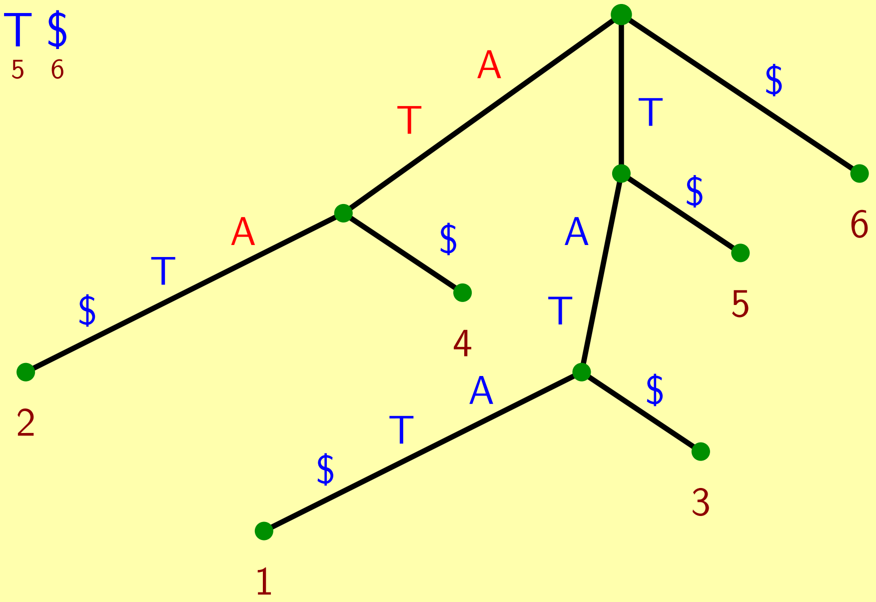
- A suffix of a string S of length n is a substring of S that ends at position n.



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

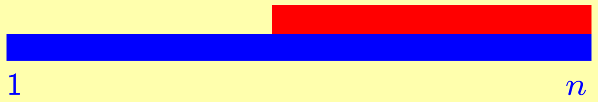
$S = \underset{1}{\text{T}}\,\underset{2}{\text{A}}\,\underset{3}{\text{T}}\,\underset{4}{\text{A}}\,\underset{5}{\text{T}}\,\underset{6}{\text{\$}}$

$P = \text{A T A}$



Suffix Tree: Definition

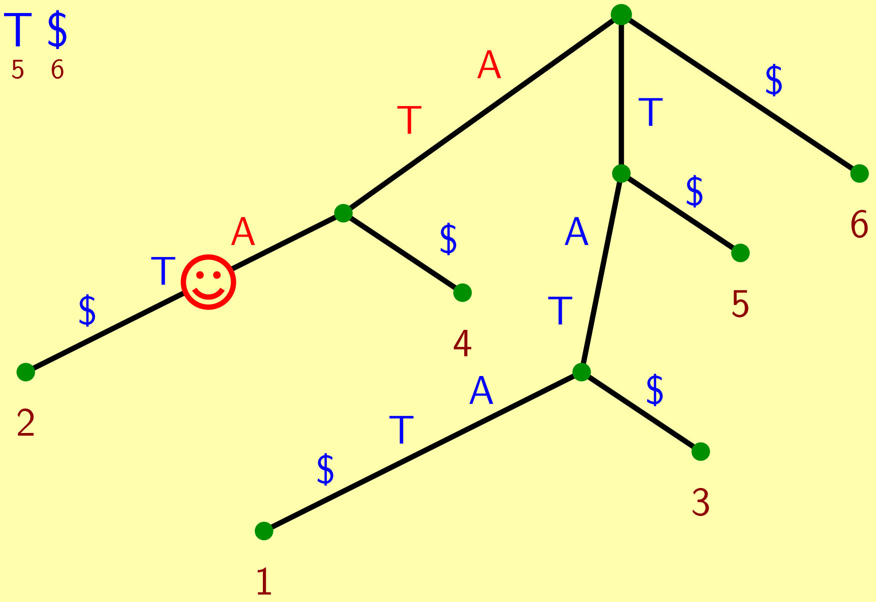
- A suffix of a string S of length n is a substring of S that ends at position n.



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

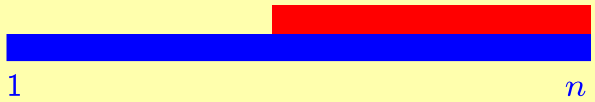
$S = \underset{1}{\mathrm{T}}\,\underset{2}{\mathrm{A}}\,\underset{3}{\mathrm{T}}\,\underset{4}{\mathrm{A}}\,\underset{5}{\mathrm{T}}\,\underset{6}{\$}$

$P = \mathrm{ATA}$



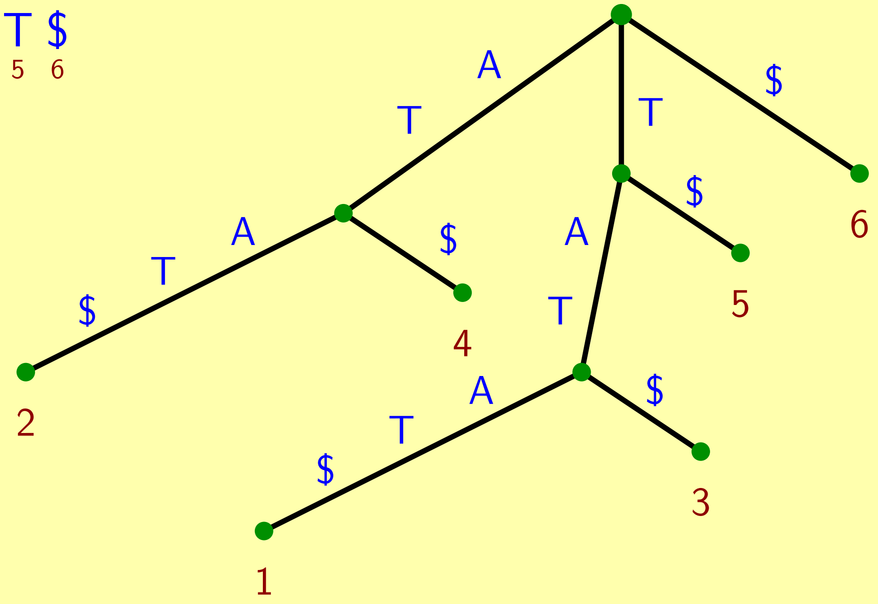
Suffix Tree: Definition

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



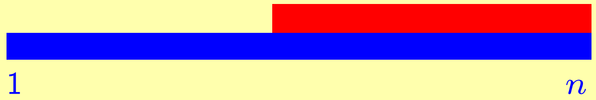
- The suffix tree of S, $T(S)$, is a rooted tree whose edges are labeled with strings such that
  - all edges leaving a node begin with different characters and
  - the paths from the root to the leaves represent all the suffixes of S.

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



Suffix Tree: Definition

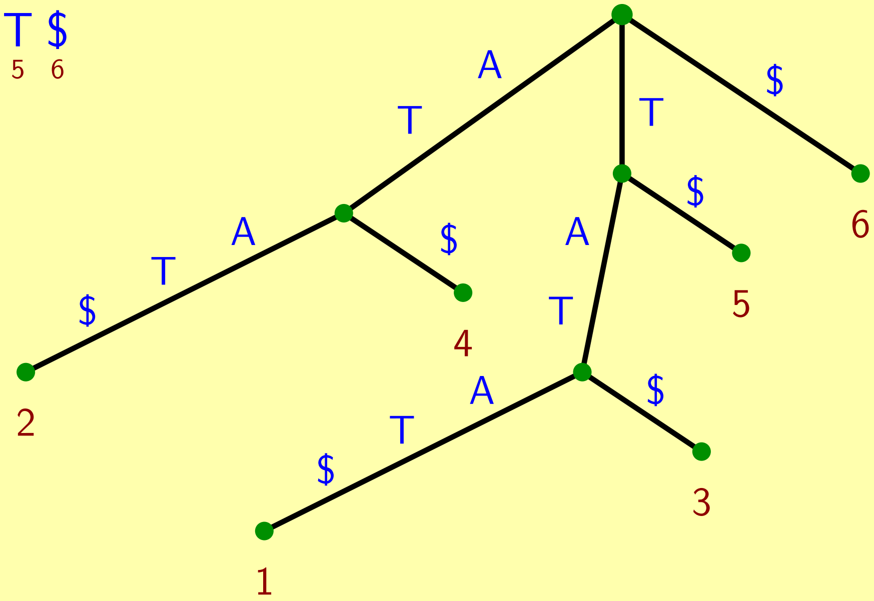
- A suffix of a string S of length n is a substring of S that ends at position n.



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

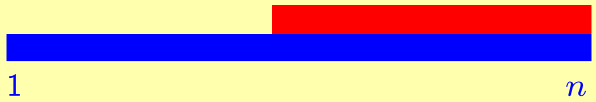
$S = \underset{1}{\text{T}}\,\underset{2}{\text{A}}\,\underset{3}{\text{T}}\,\underset{4}{\text{A}}\,\underset{5}{\text{T}}\,\underset{6}{\text{\$}}$

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



Suffix Tree: Definition

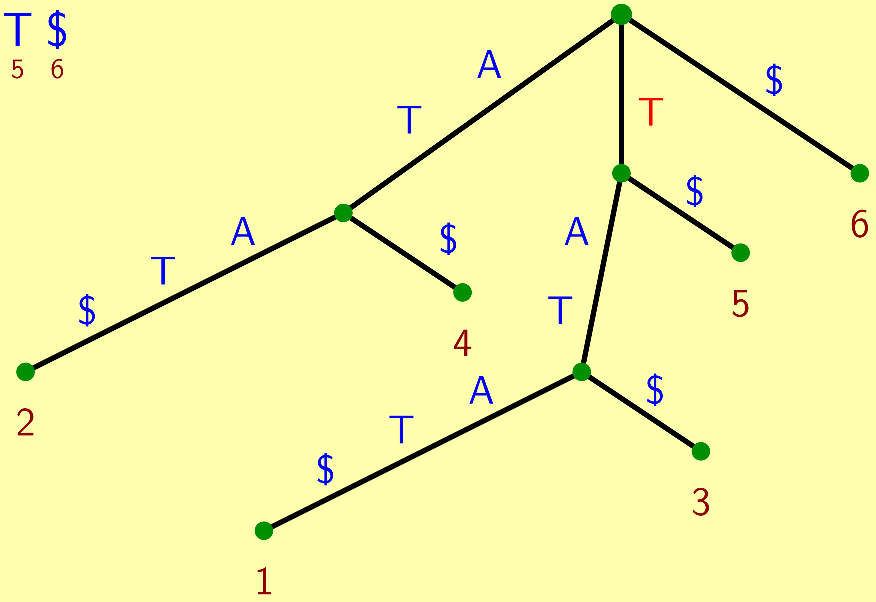
- A suffix of a string S of length n is a substring of S that ends at position n.



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

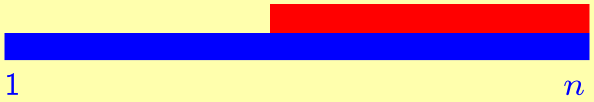
$S = \underset{1}{T}\,\underset{2}{A}\,\underset{3}{T}\,\underset{4}{A}\,\underset{5}{T}\,\underset{6}{\$}$

$P = TATT$



Suffix Tree: Definition

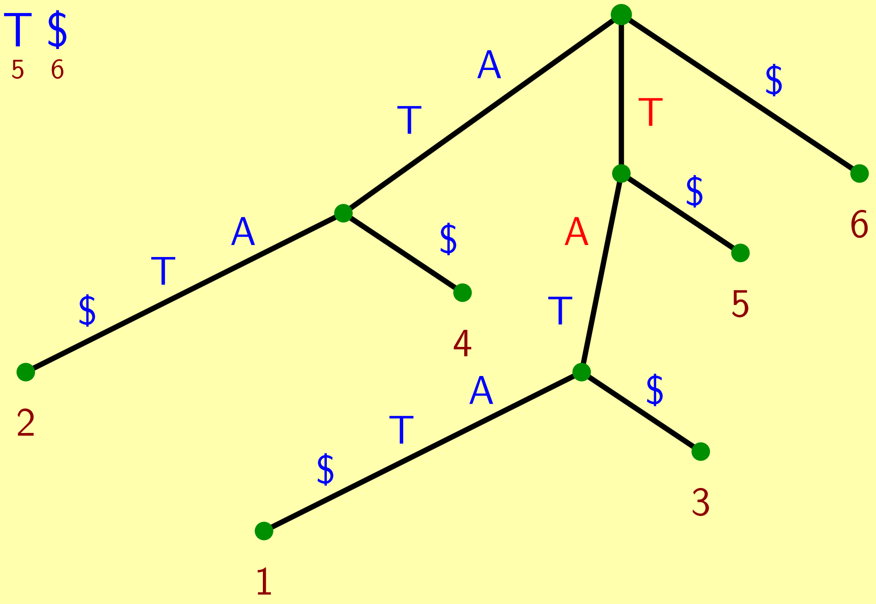
- A suffix of a string S of length n
  is a substring of S that ends at position n.



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

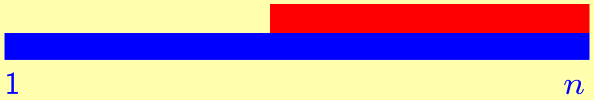
$S = \underset{1}{\text{T}}\,\underset{2}{\text{A}}\,\underset{3}{\text{T}}\,\underset{4}{\text{A}}\,\underset{5}{\text{T}}\,\underset{6}{\$}$

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



Suffix Tree: Definition

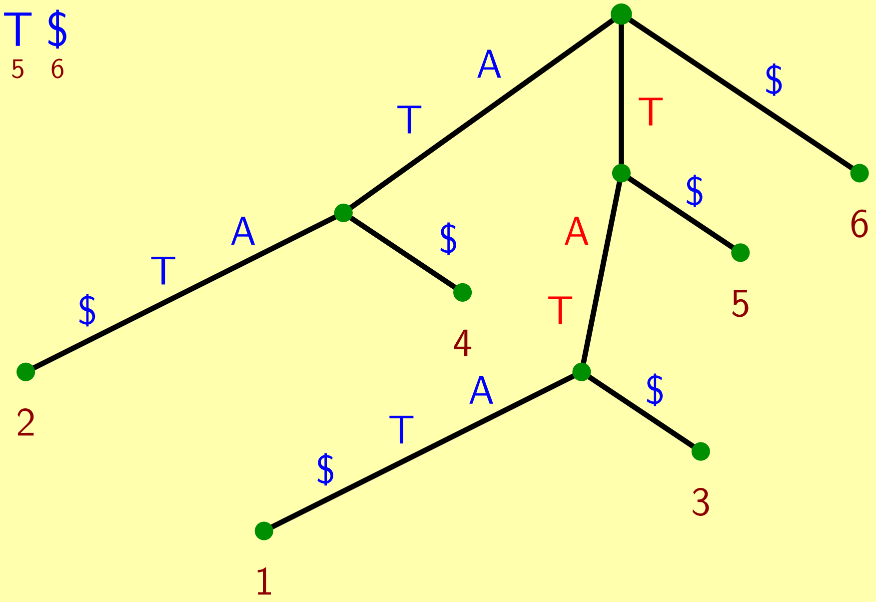
- A suffix of a string S of length n is a substring of S that ends at position n.



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

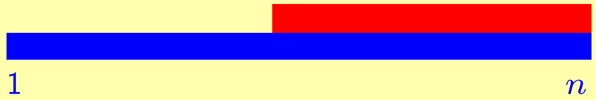
$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\$}$

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



Suffix Tree: Definition

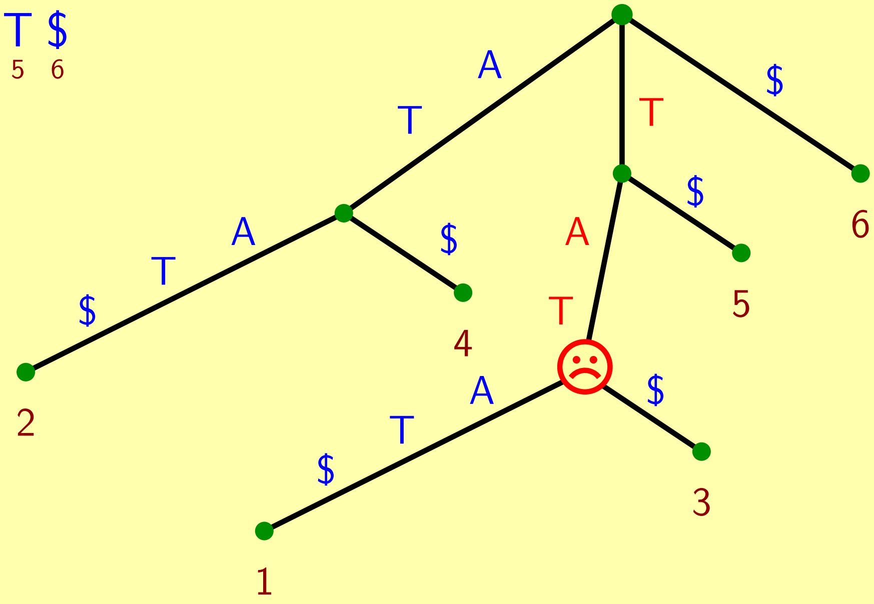
- A suffix of a string S of length n is a substring of S that ends at position n.



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

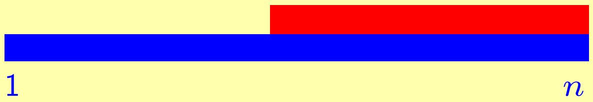
$S = \underset{1}{\mathsf{T}}\,\underset{2}{\mathsf{A}}\,\underset{3}{\mathsf{T}}\,\underset{4}{\mathsf{A}}\,\underset{5}{\mathsf{T}}\,\underset{6}{\$}$

$P = \mathsf{TATT}$



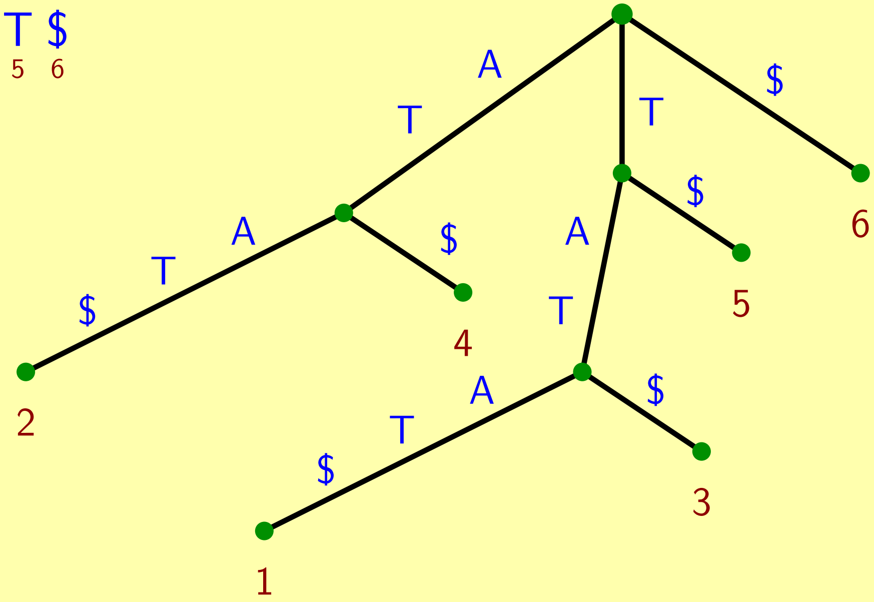
Suffix Tree: Definition

- 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
  - all edges leaving a node begin with different characters and
  - the paths from the root to the leaves represent all the suffixes of S.

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



Suffix tree 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:

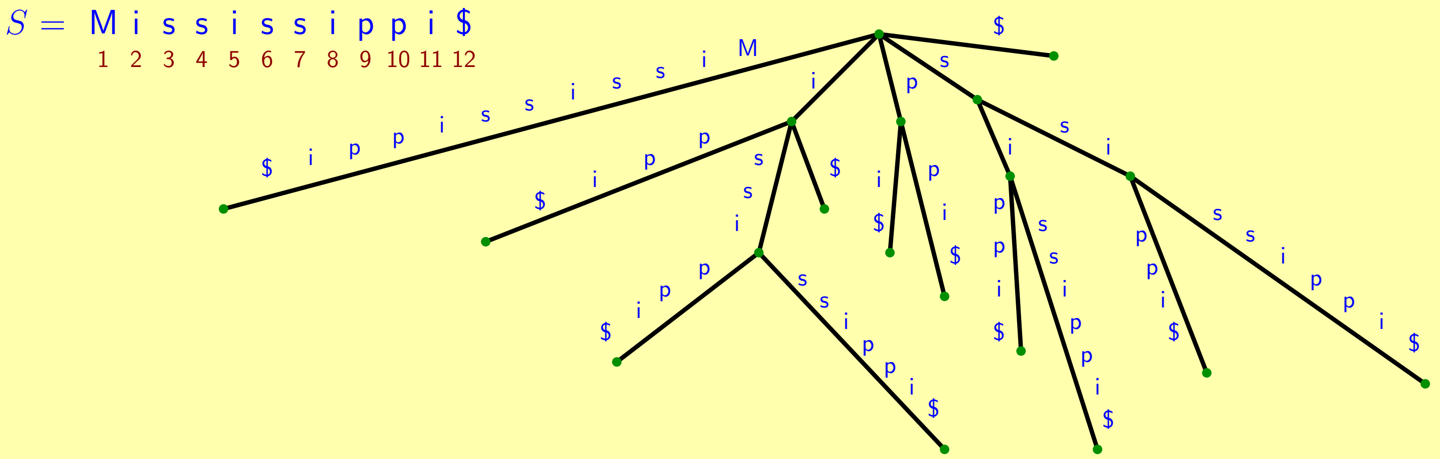
- A. Apostolico: The myriad virtues of subword trees, 1985.
- D. Gusfield: Algorithms on strings, trees, and sequences, 1997.

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 $\Rightarrow$ 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.



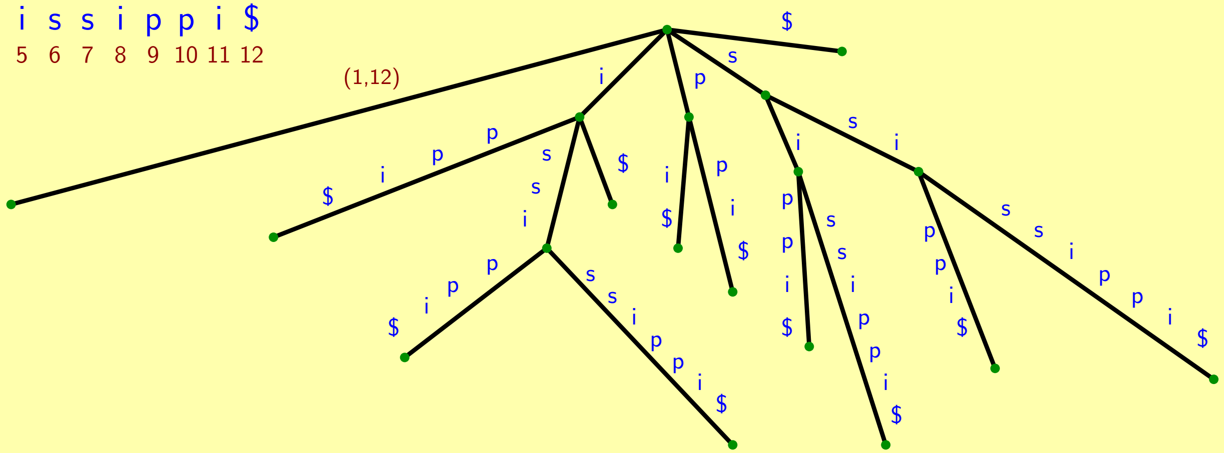
Space usage of suffix trees

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

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ 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.

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



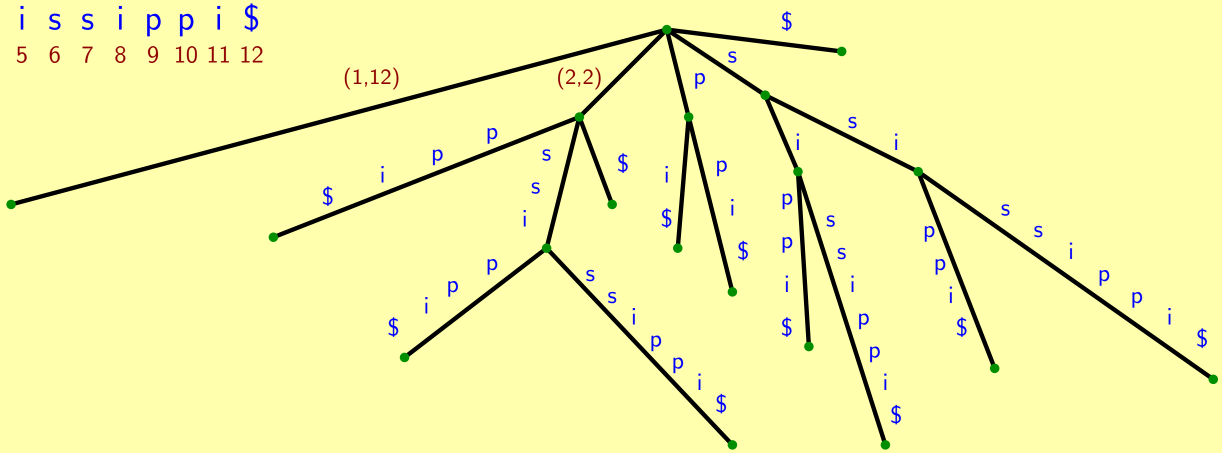
Space usage of suffix trees

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.

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



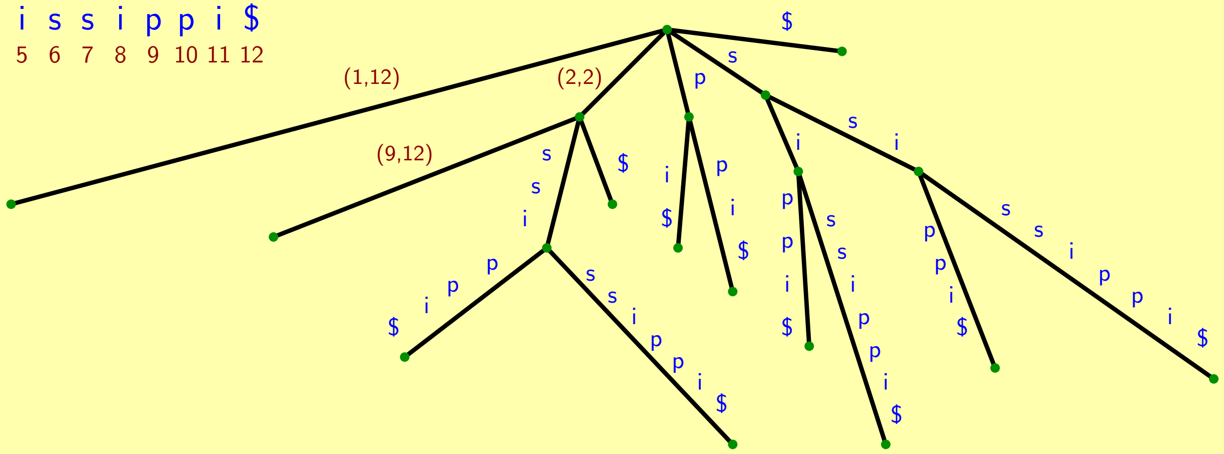
Space usage of suffix trees

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.

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

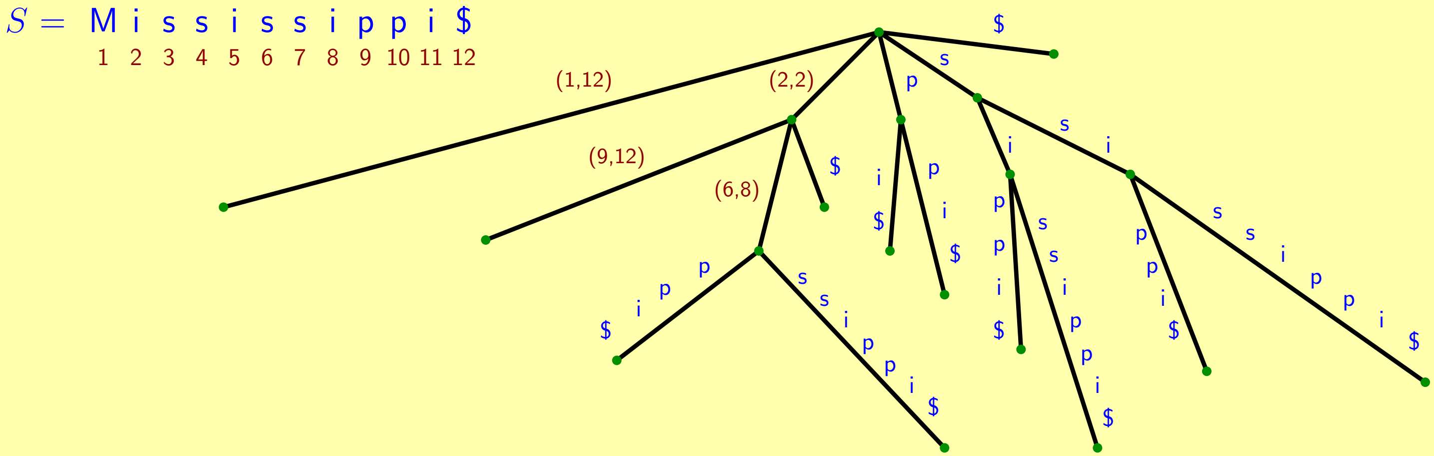


Space usage of suffix trees

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.

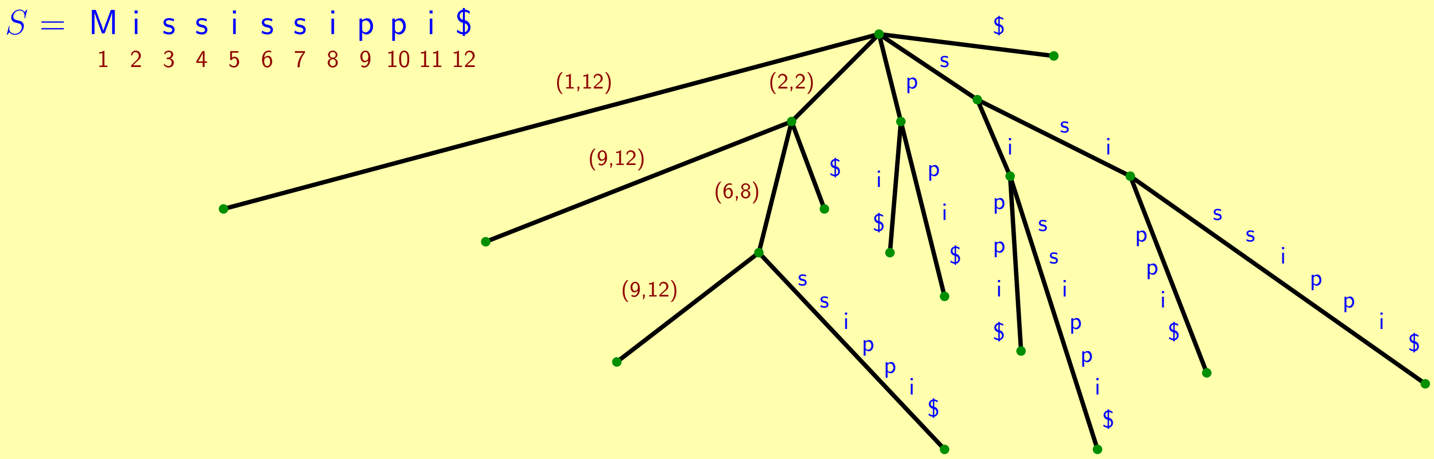


Space usage of suffix trees

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

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ 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.



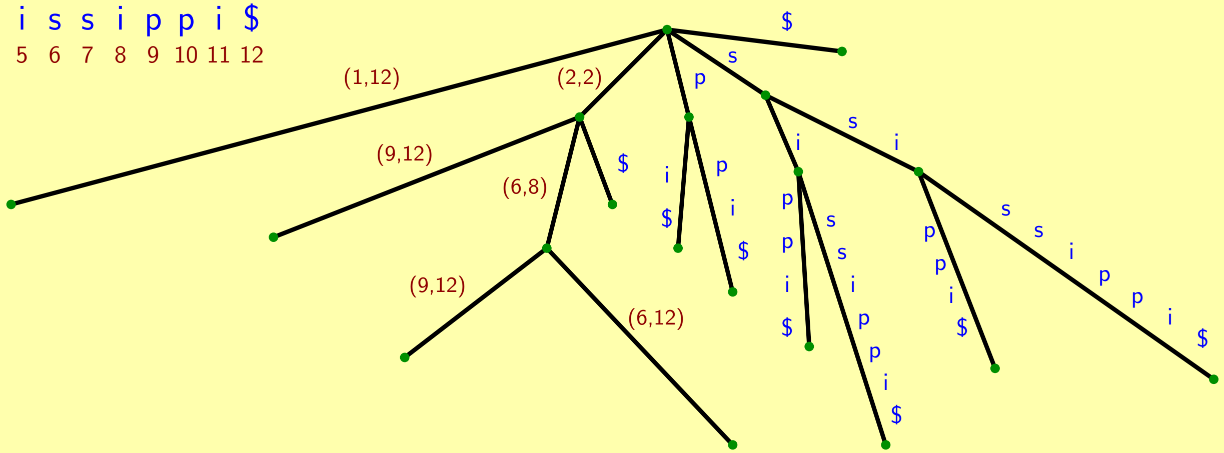
Space usage of suffix trees

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

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ 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.

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

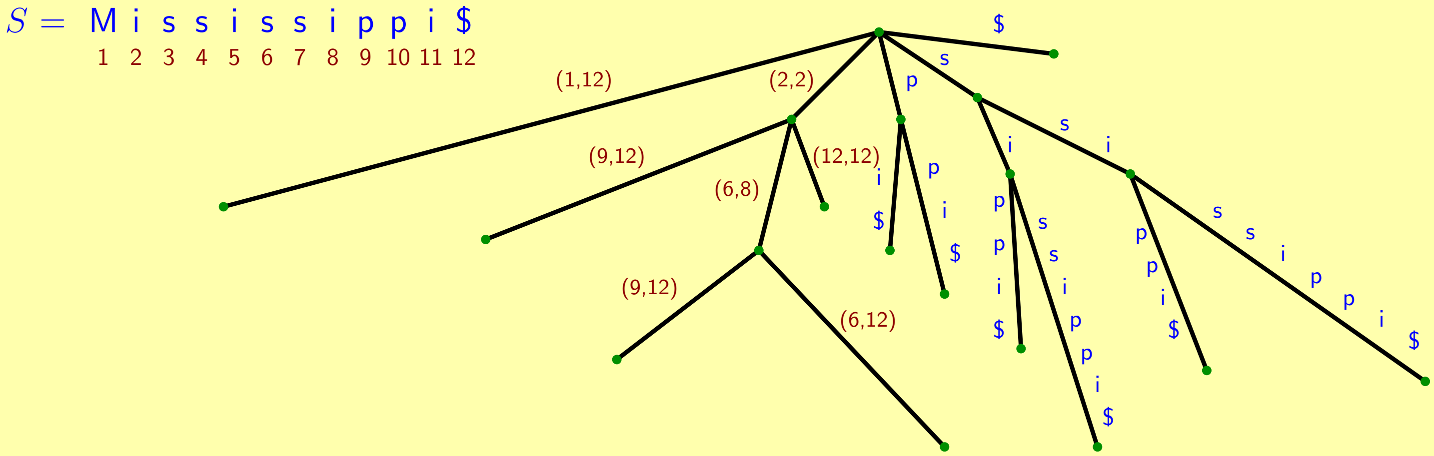


Space usage of suffix trees

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.

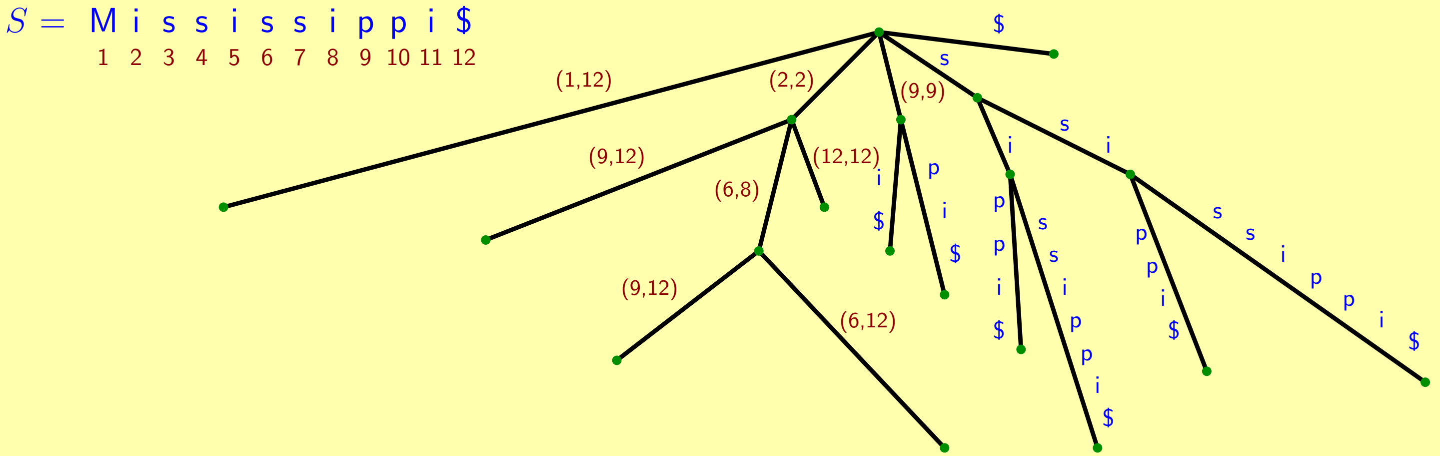


Space usage of suffix trees

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

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ 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.

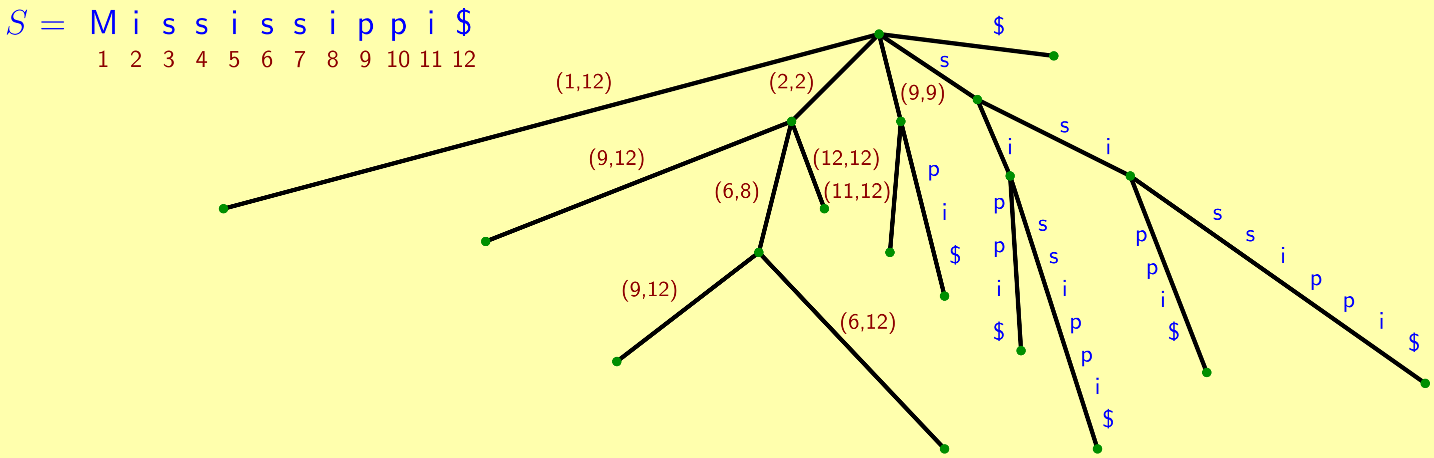


Space usage of suffix trees

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

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ 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.



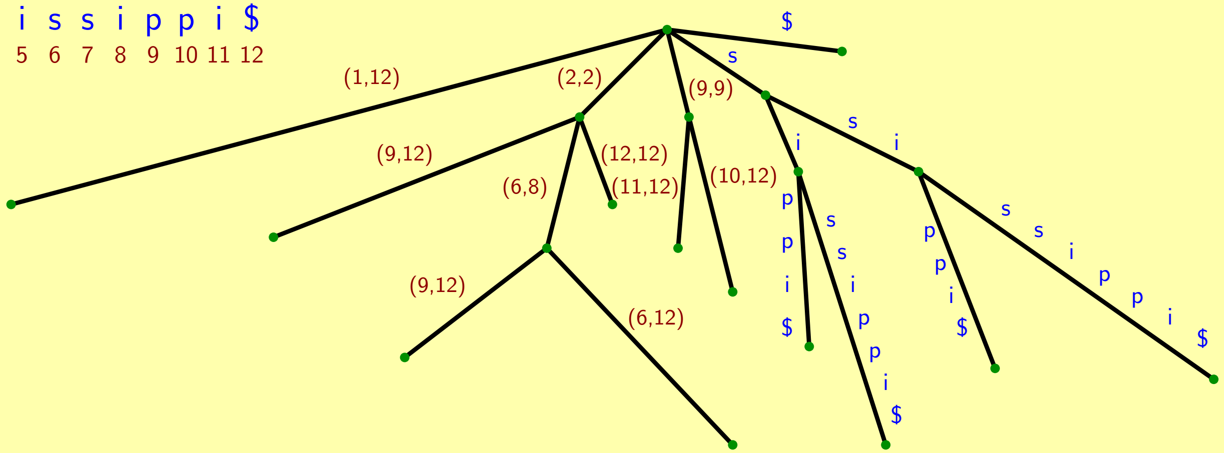
Space usage of suffix trees

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

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ 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.

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

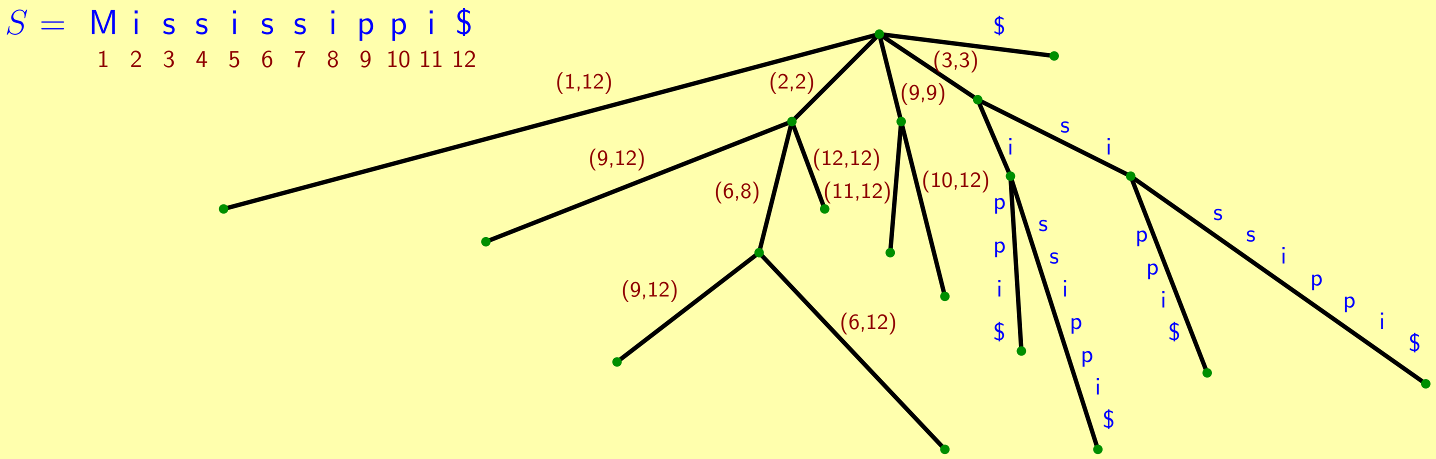


Space usage of suffix trees

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.

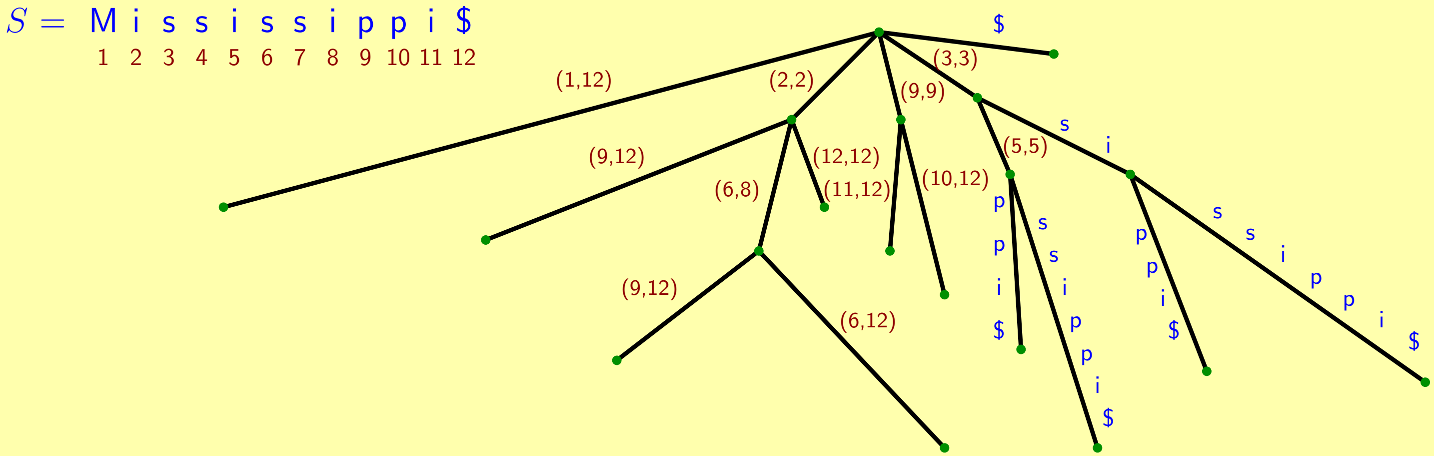


Space usage of suffix trees

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

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ 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.

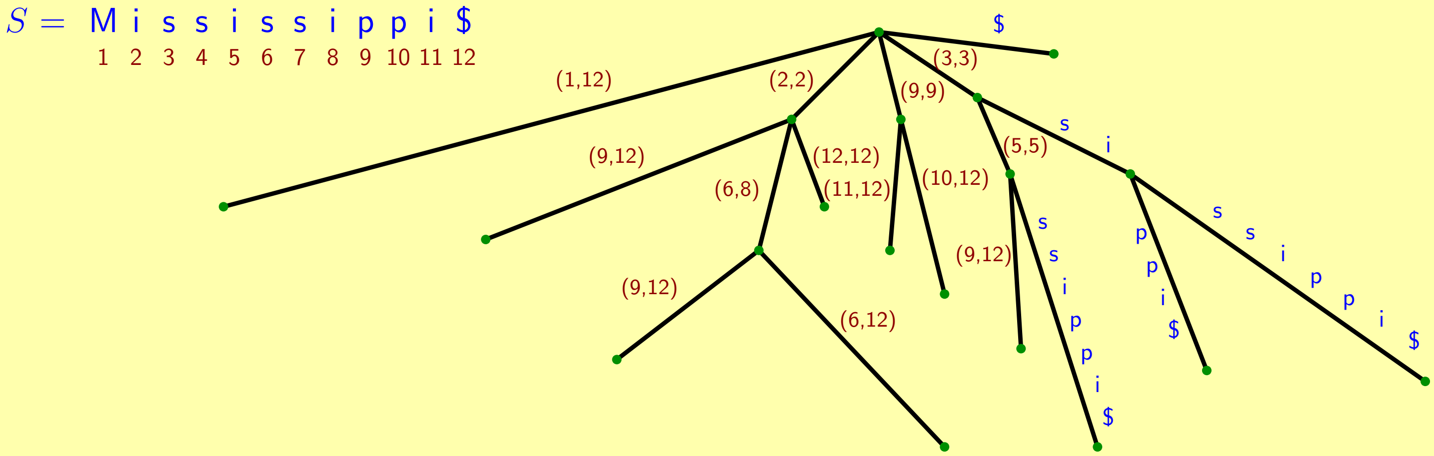


Space usage of suffix trees

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

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ 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.

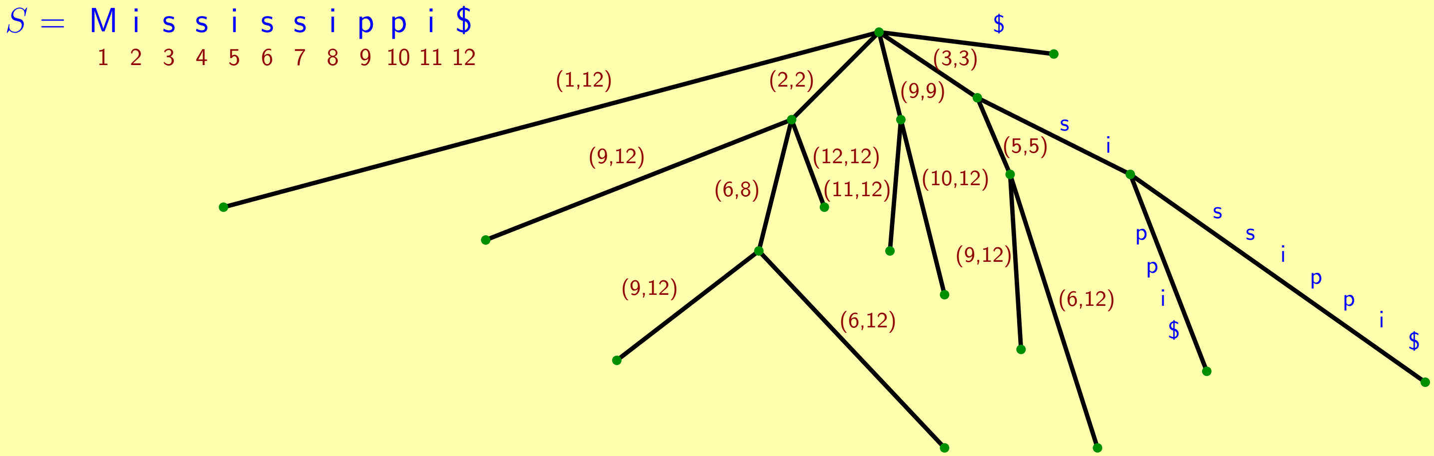


Space usage of suffix trees

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

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ 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.

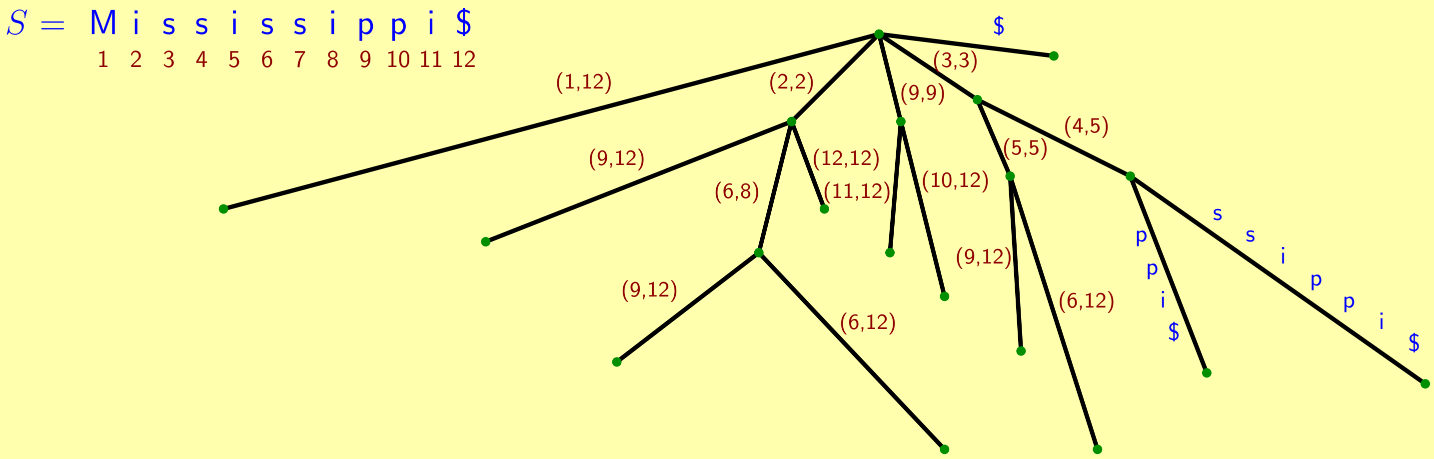


Space usage of suffix trees

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

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ 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.



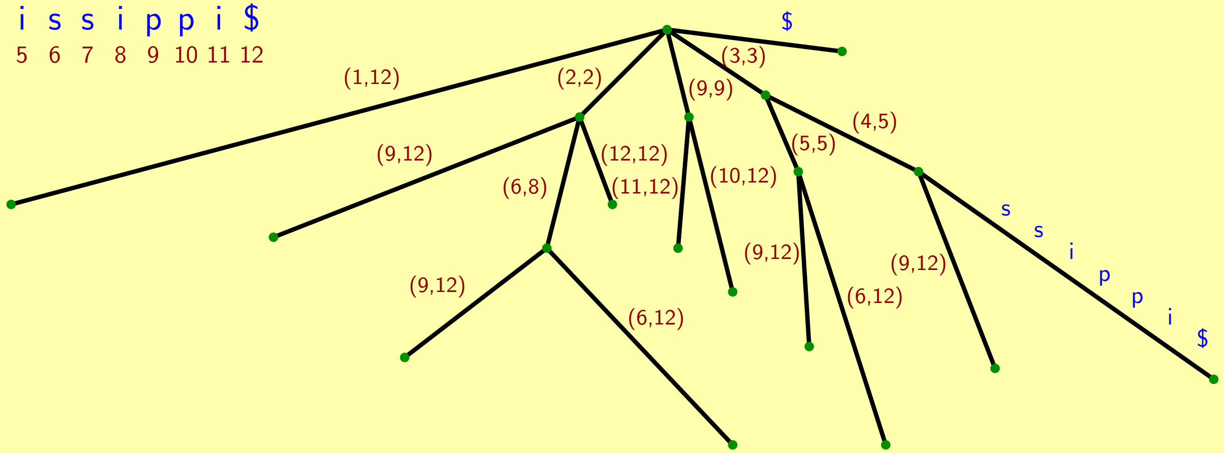
Space usage of suffix trees

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

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ 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.

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



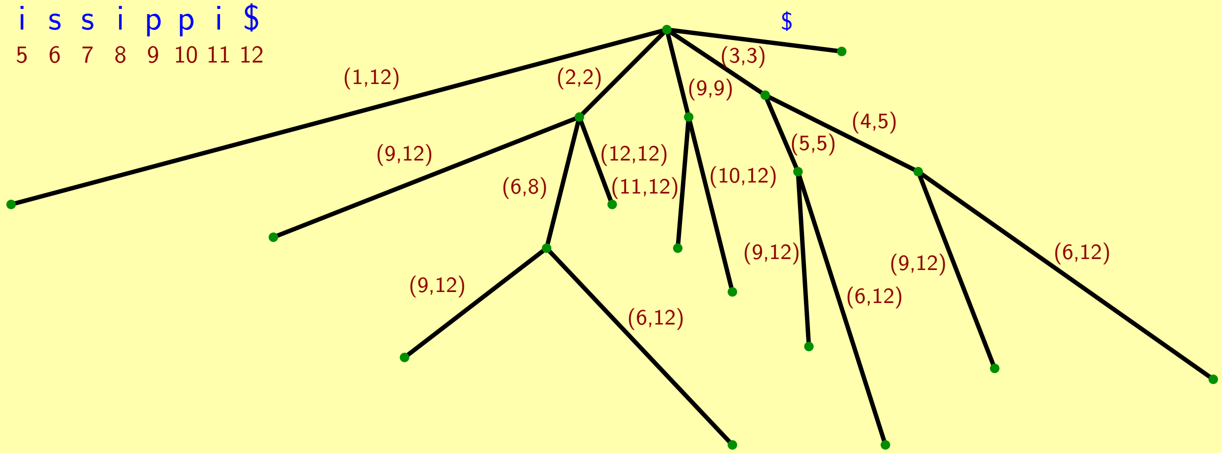
Space usage of suffix trees

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.

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



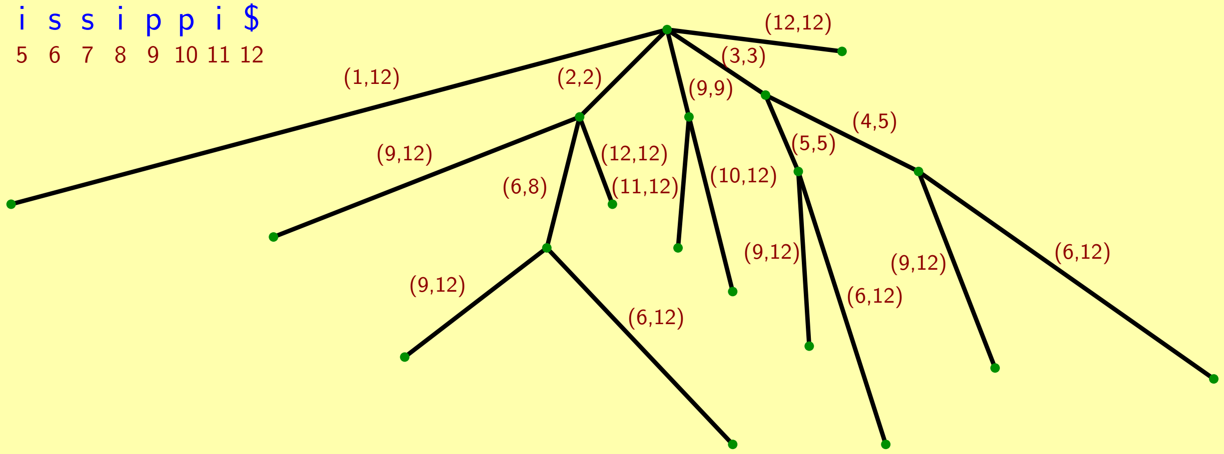
Space usage of suffix trees

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.

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

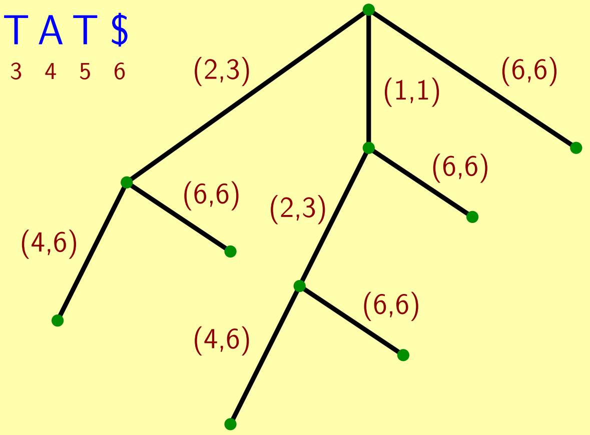


Representation of suffix trees

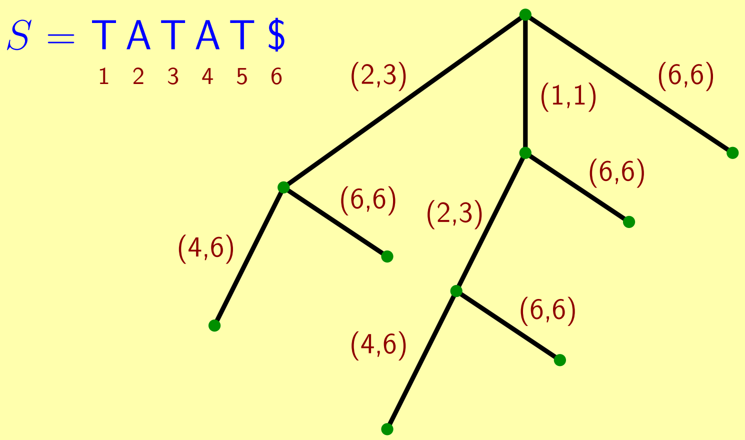
Representation of suffix trees

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

1 2 3 4 5 6



Representation of suffix trees

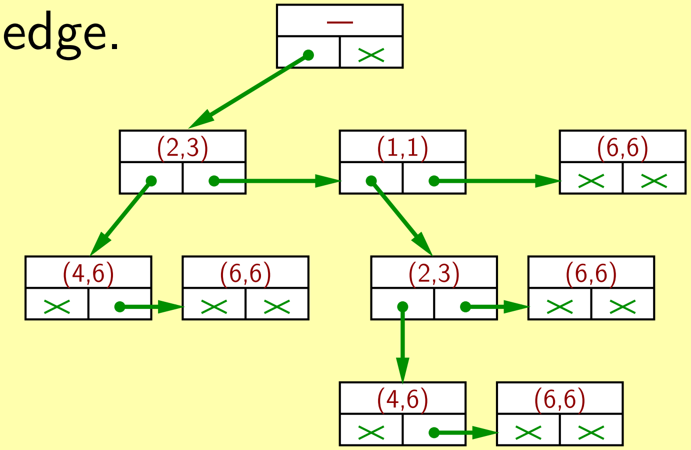


Standard representation of trees:

- Store nodes as records with child and sibling pointer.
- An edge label (i, j) is stored at node below the edge.

⇒ about $32n$ bytes in the worst case

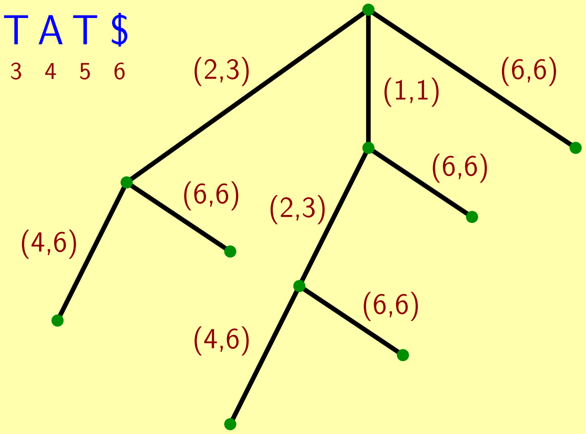
$2n$ nodes × (2 integers + 2 pointers)



Representation of suffix trees

$S =$ T A T A T \$

1 2 3 4 5 6

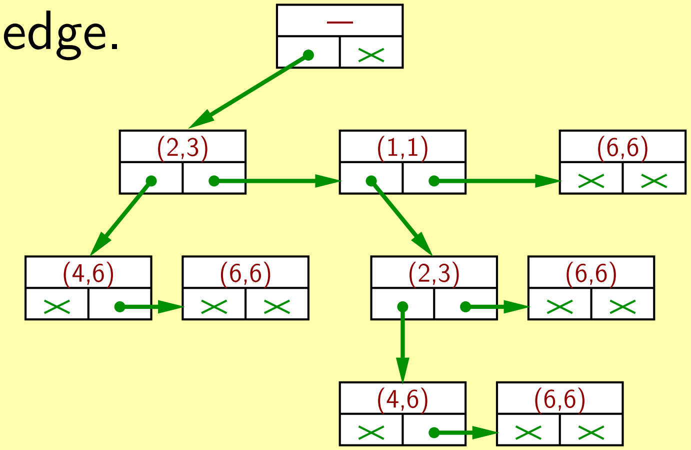


Standard representation of trees:

- Store nodes as records with child and sibling pointer.
- An edge label (i, j) is stored at node below the edge.

$\Rightarrow$ about $32n$ bytes in the worst case

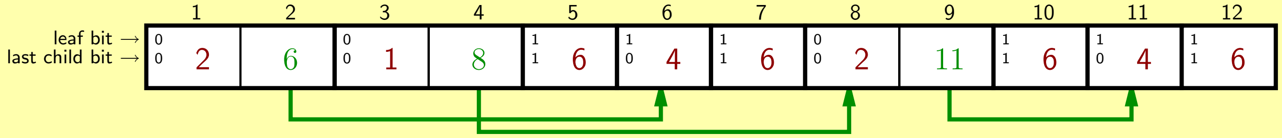
$2n$ nodes $\times$ (2 integers + 2 pointers)



Ideas for more efficient representation:

- Do not represent leaves explicitly.
- Avoid sibling pointers by storing all children of the same node in a row.
- Do not represent the right pointer of an edge label.

$\Rightarrow$ 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

Construction of suffix trees

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

There are two practical algorithms that construct the suffix tree in linear time: McCreight (1976) and Ukkonen (1993).

Construction of suffix trees

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

There are two practical algorithms that construct the suffix tree in linear time: McCreight (1976) and Ukkonen (1993).

A simpler algorithm is the WOTD (write-only, top-down) algorithm:

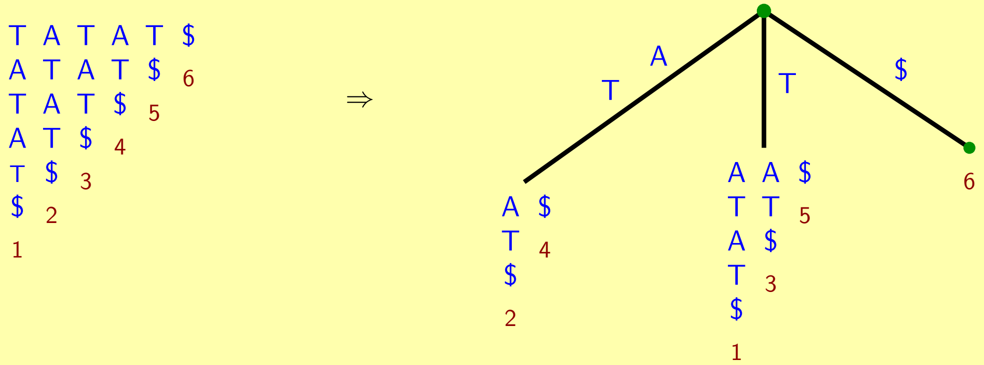
1. Let X be the set of all suffixes of S.
2. Sort the suffixes in X according to their first character.
3. For each group X_c ($c \in \Sigma$):
   - (i) if X_c is a singleton, create a leaf;
   - (ii) otherwise, find the longest common prefix of the suffixes in X_c, create an internal node, and recursively continue with Step 2, X being the set of remaining suffixes from X_c after splitting off the longest common prefix.

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

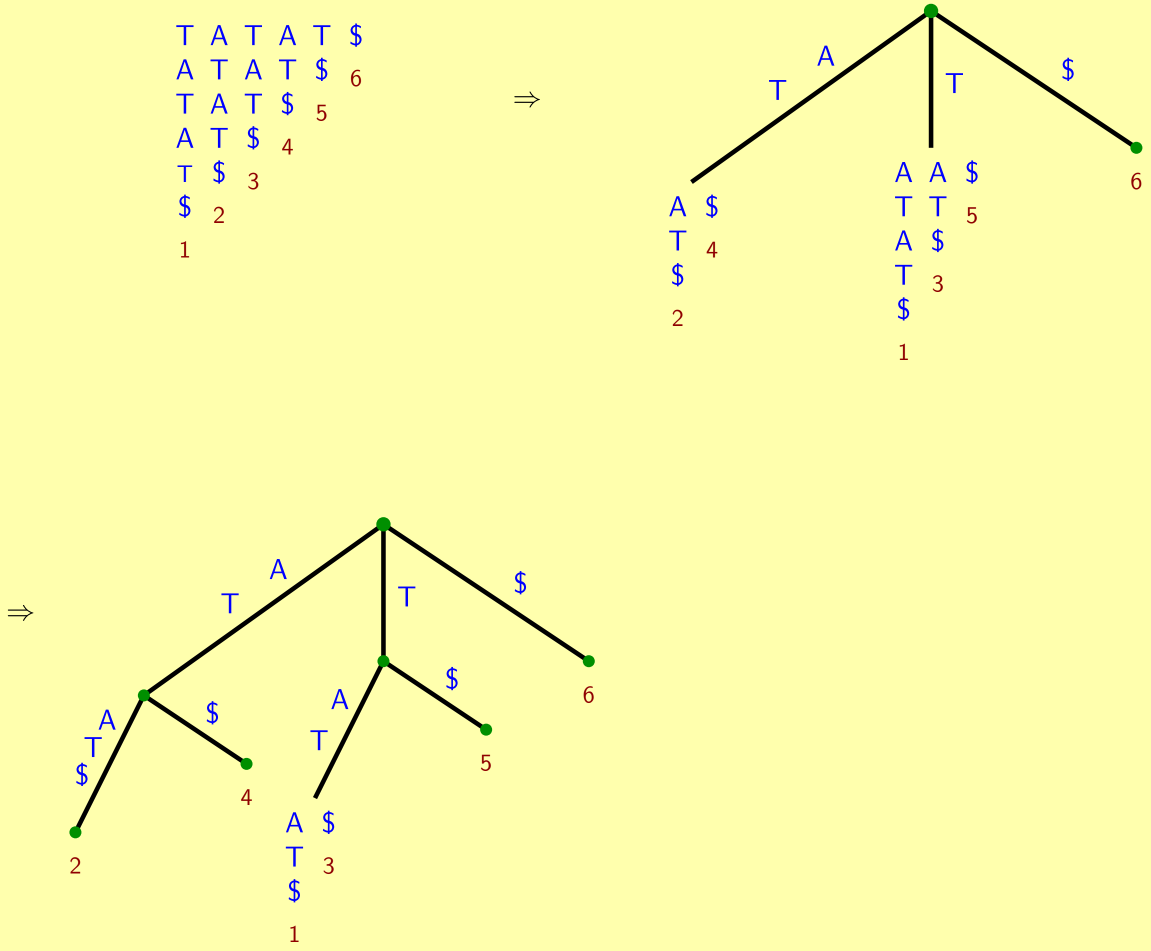
The WOTD construction algorithm

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

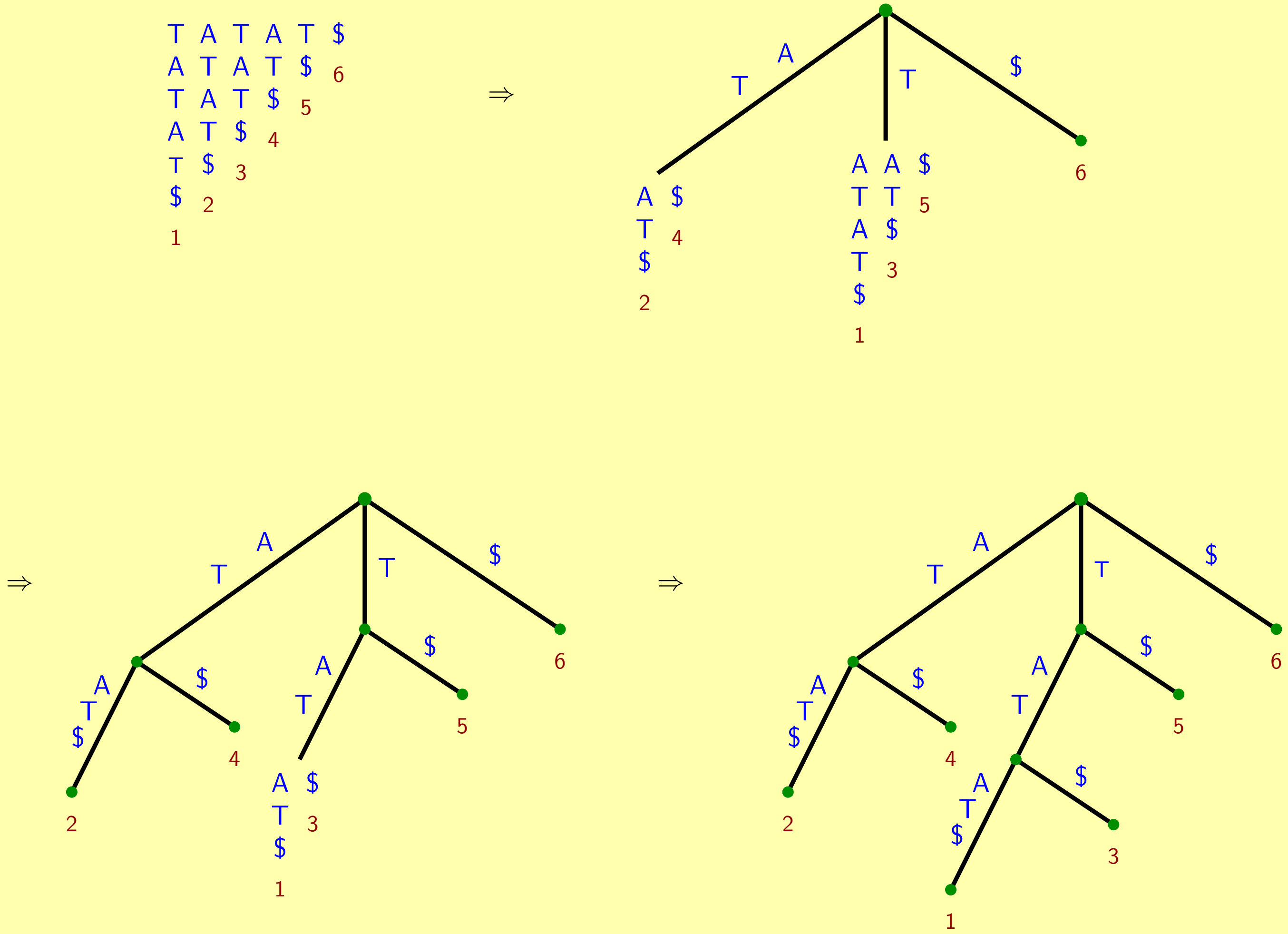
The WOTD construction algorithm



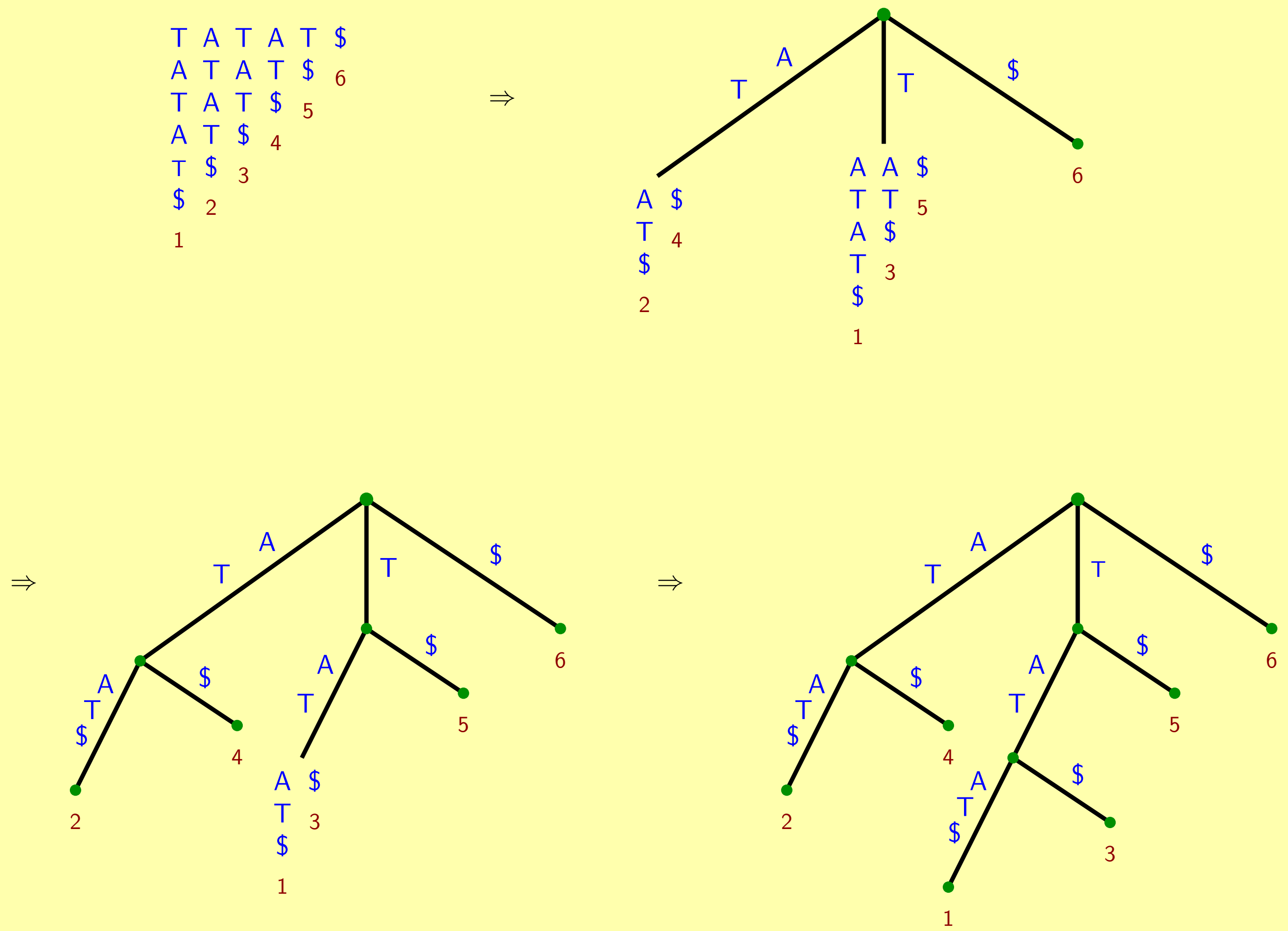
The WOTD construction algorithm



The WOTD construction algorithm



The WOTD construction algorithm



Note: The WOTD algorithm is well suited for a lazy construction of suffix trees.

Comparison: Exact string matching online and offline

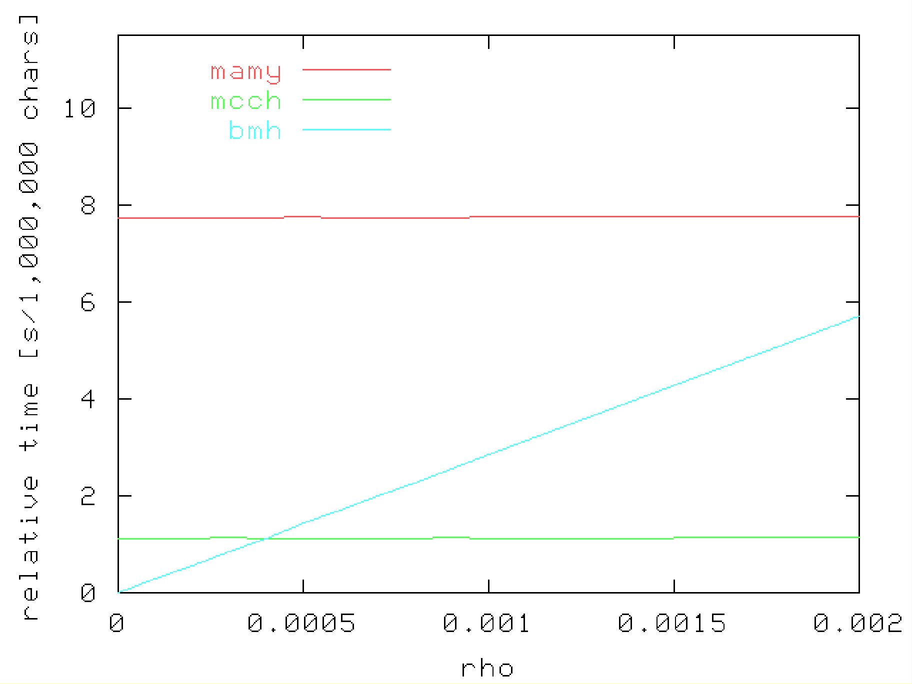
Theoretical results:

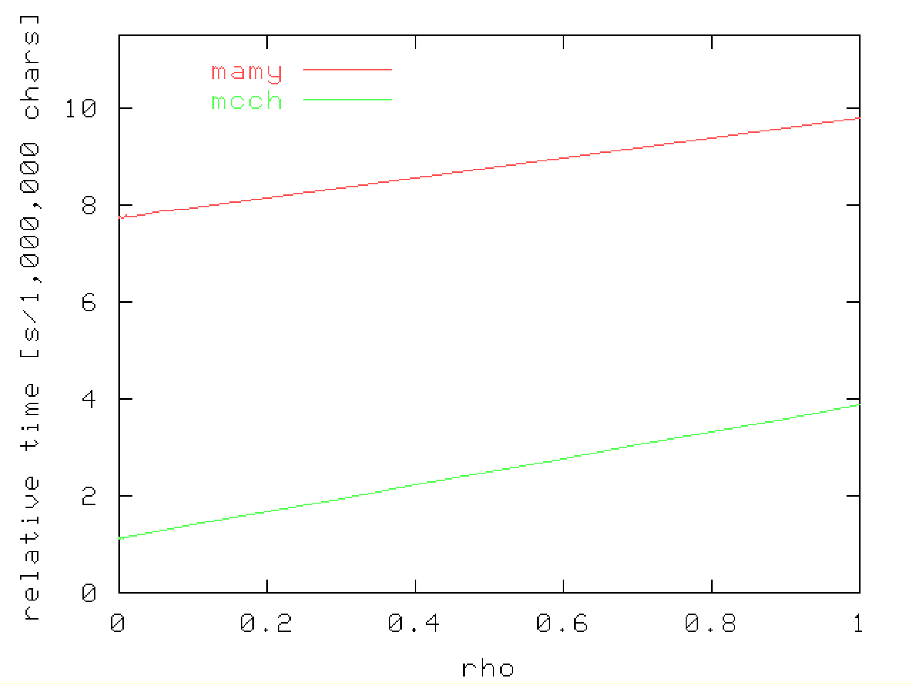
	online (no preprocessing)	offline (suffixtree)
1 pattern search	$\mathcal{O}(n+m)$	$\mathcal{O}(n+m)$
k pattern searches	$\mathcal{O}(k\ (n+m))$	$\mathcal{O}(n+km)$

n = text length m = pattern length

Comparison: Exact string matching online and offline

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

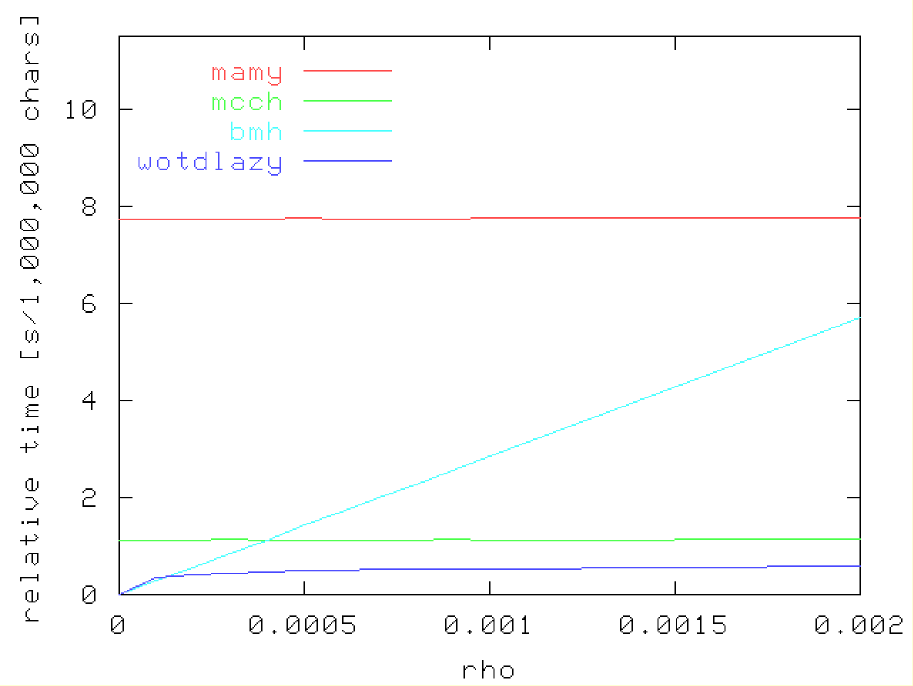


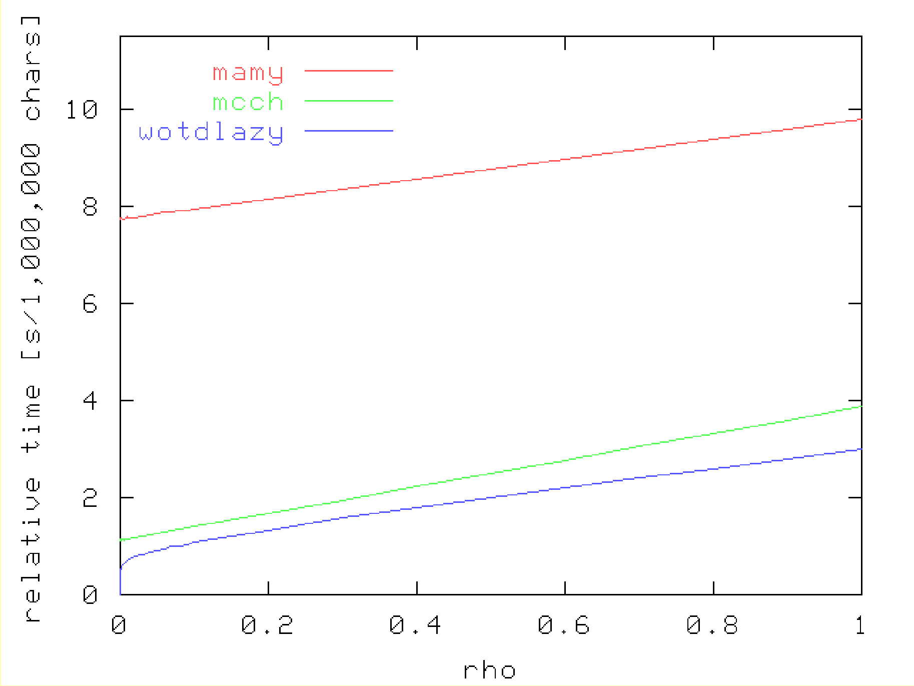


mamy = suffix array (implementation by Manber/Myers)
mcch = suffix tree (McCreight's algorithm with hash tables)
bmh = online search (Boyer-Moore-Horspool algorithm)

Comparison: Exact string matching online and offline

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





mamy = suffix array (implementation by Manber/Myers)

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)

Overview

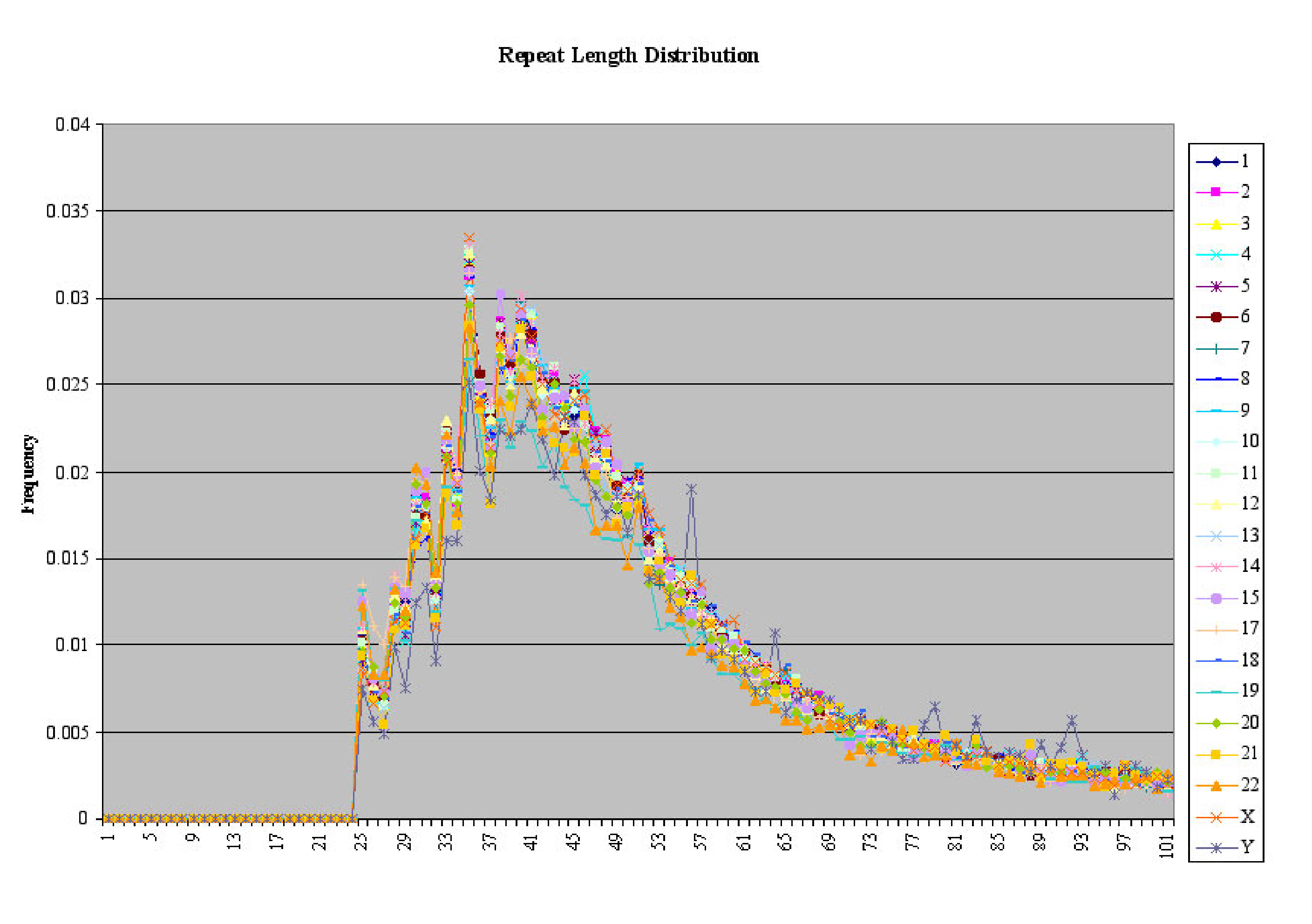
- Introduction: Pattern matching in biosequence analysis
- A flexible index structure for sequence analysis: The suffix tree
  - Basic definitions
  - Application 1: Repeats, tandem repeats
  - Application 2: Multiple Genome Aligner (MGA)
- An even more flexible data structure: The affix tree
  - Basic definitions
  - Application: Search for palindromic patterns
- Conclusion

Repeats in biosequence analysis

- DNA of eukaryotes is highly repetitive.
  - 30% in human genome?
  - 10% introduced by retroviruses?
- Repeat regions are rapidly changing *hot spots* in evolution.
- Vast literature on repetitive structures and their hypothesized functional and evolutionary roles: ALUs, SINEs, LINEs, satellites, ...
- Repeats are involved in several biological mechanisms, including genetically inherited diseases.
  - e.g. Huntington's disease
- Repeats tend to confuse sequence analysis programs and hence should be masked in a preprocessing step.

⇒ Repeats are very important when studying genomic DNA.

Repeats in the human genome



(Human Genome Sequencing Center, Baylor College of Medicine, Houston, Texas)

Basic definitions

A pair of substrings $R = (S[i_1, j_1], S[i_2, j_2])$ is called a repeat.

$\longrightarrow$ *exact* repeat if $S[i_1, j_1] = S[i_2, j_2]$

Basic definitions

A pair of substrings $R = (S[i_1, j_1], S[i_2, j_2])$ is called a repeat.

⟶ *exact* repeat if $S[i_1, j_1] = S[i_2, j_2]$

Basic definitions

A pair of substrings $R = (S[i_1, j_1], S[i_2, j_2])$ is called a repeat.

⟶ *exact* repeat if $S[i_1, j_1] = S[i_2, j_2]$

⟶ *k-mismatch* repeat if there are k mismatches between $S[i_1, j_1]$ and $S[i_2, j_2]$

Basic definitions

A pair of substrings $R = (S[i_1, j_1], S[i_2, j_2])$ is called a repeat.

- → *exact* repeat if $S[i_1, j_1] = S[i_2, j_2]$

- → *k-mismatch* repeat if there are k mismatches between $S[i_1, j_1]$ and $S[i_2, j_2]$

- → *k-differences* repeat if there are k differences (mismatches, insertions, deletions) between $S[i_1, j_1]$ and $S[i_2, j_2]$

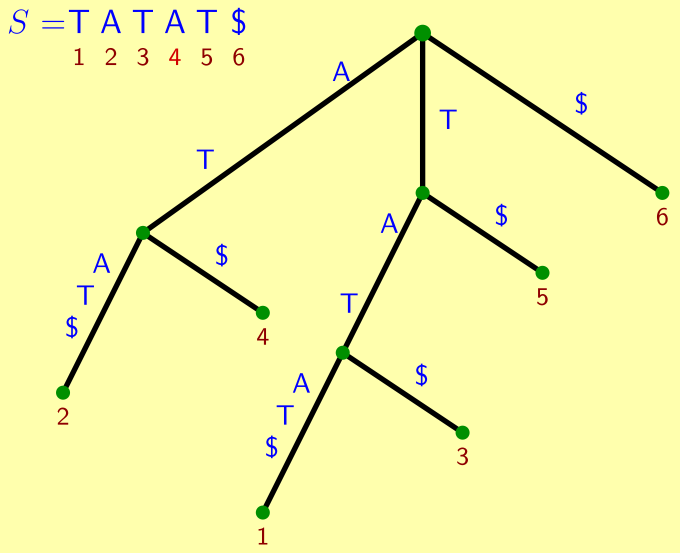
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.

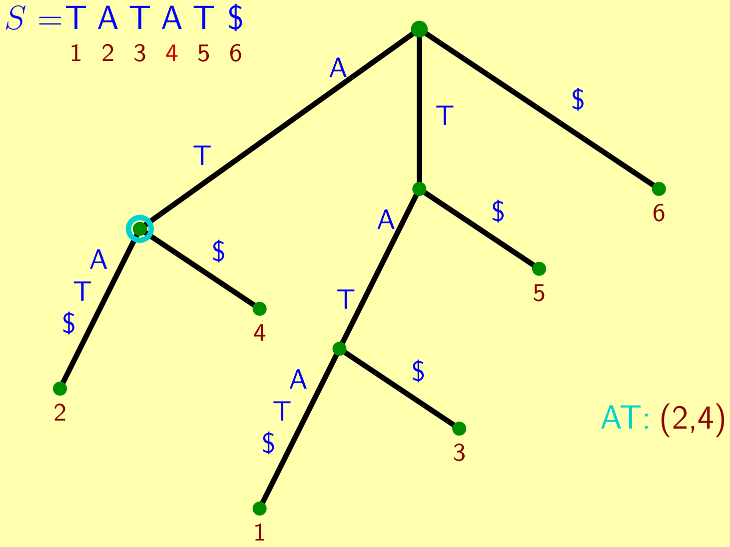


Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:



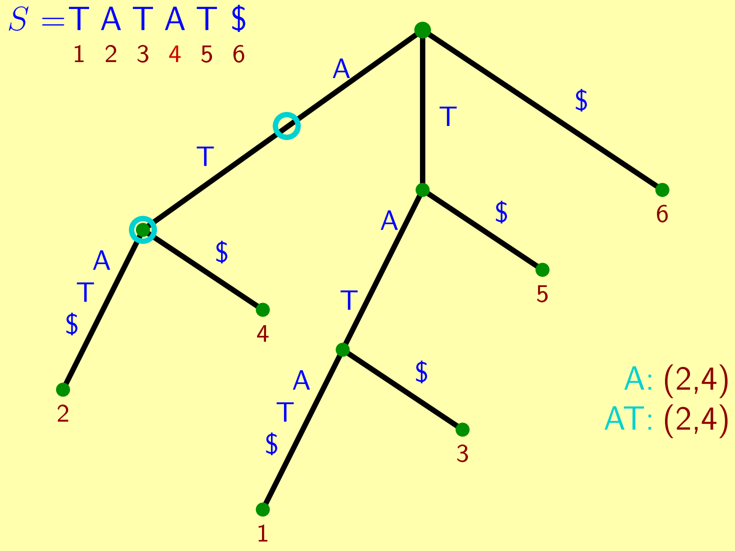
- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.

Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:



A: (2,4)
AT: (2,4)

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.

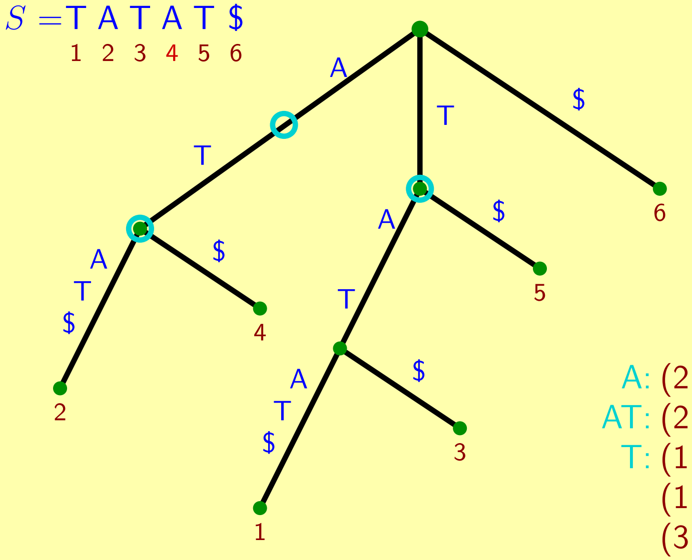
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.



A: (2,4)
AT: (2,4)
T: (1,3)
(1,5)
(3,5)

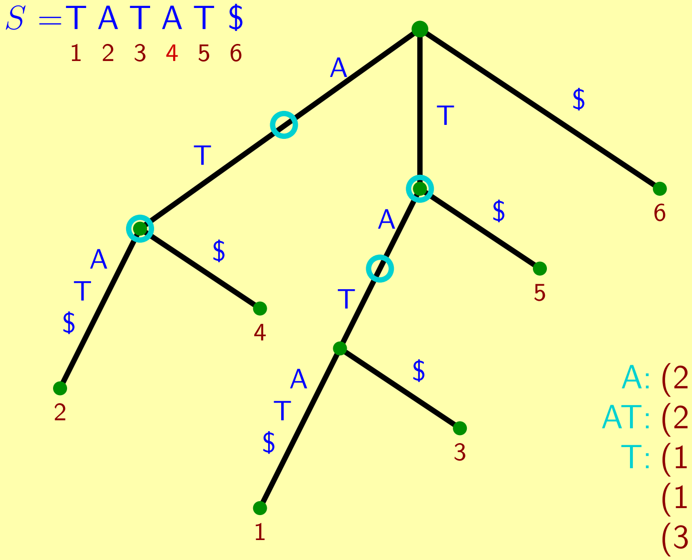
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.



A: (2,4)
AT: (2,4)
T: (1,3)
(1,5)
(3,5)
TA: (1,3)

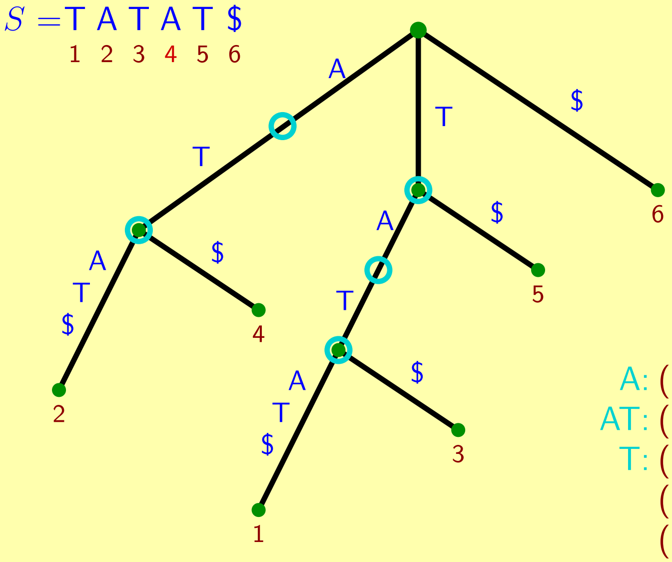
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.



A: (2,4)
AT: (2,4)
T: (1,3)
(1,5)
(3,5)
TA: (1,3)
TAT: (1,3)

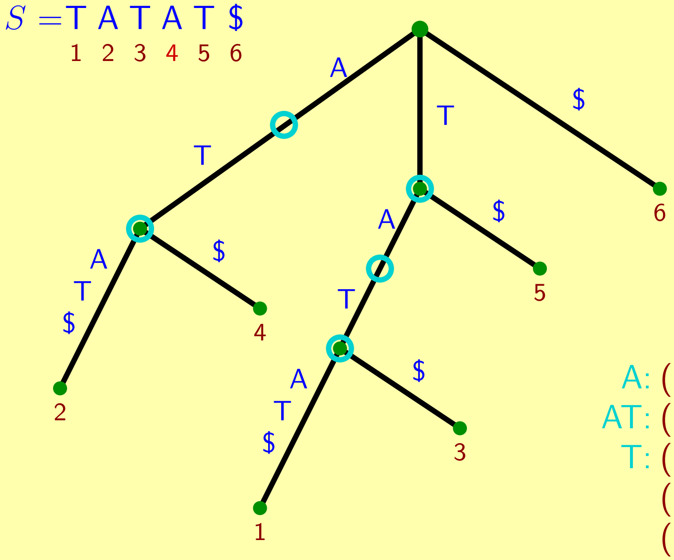
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.



A: (2,4)
AT: (2,4)
T: (1,3)
(1,5)
(3,5)
TA: (1,3)
TAT: (1,3)

Analysis: $\mathcal{O}(n+z)$ time with $z = |\text{output}|$, $\mathcal{O}(n)$ space

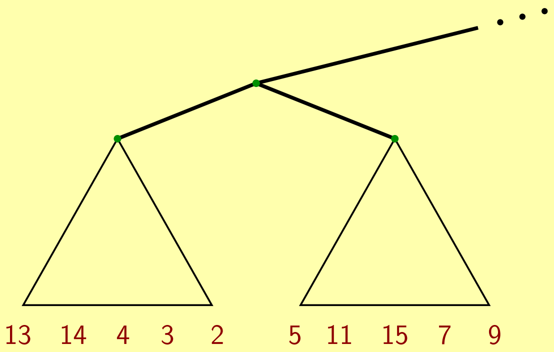
Finding *maximal* exact repeats

Finding *maximal* exact repeats

Finding *maximal* exact repeats

Finding *maximal* exact repeats

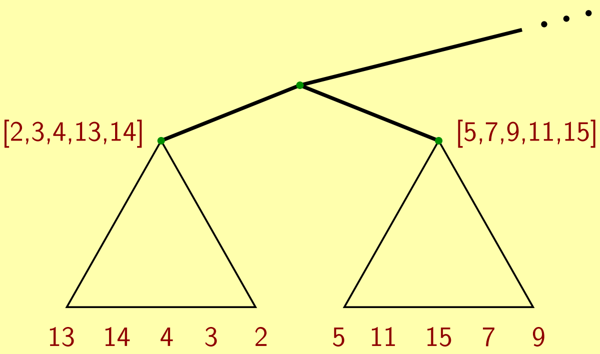
Idea:



- For right-maximality (X $\neq$ Y)
  - consider only internal nodes of $T(S)$
  - report only pairs of leaves from different subtrees

Finding *maximal* exact repeats

Idea:

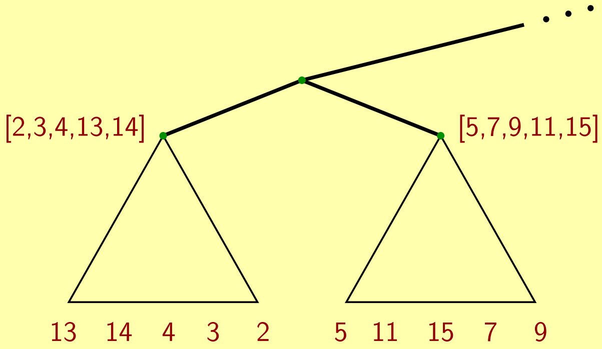


- For right-maximality (X $\neq$ Y)
  - consider only internal nodes of $T(S)$
  - report only pairs of leaves from different subtrees (or from different leaf-lists)

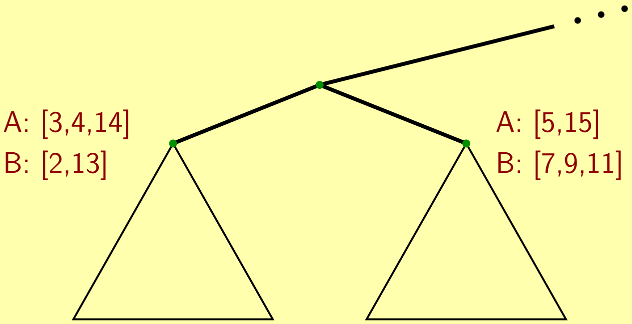
Finding *maximal* exact repeats

Idea:

- For right-maximality (X $\neq$ Y)
  - consider only internal nodes of $T(S)$
  - report only pairs of leaves from different subtrees (or from different leaf-lists)

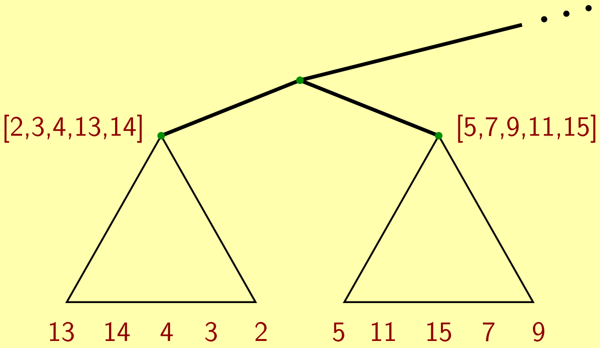


- For left-maximality (A $\neq$ B)
  - keep lists for the different left-characters
  - report only pairs from different lists

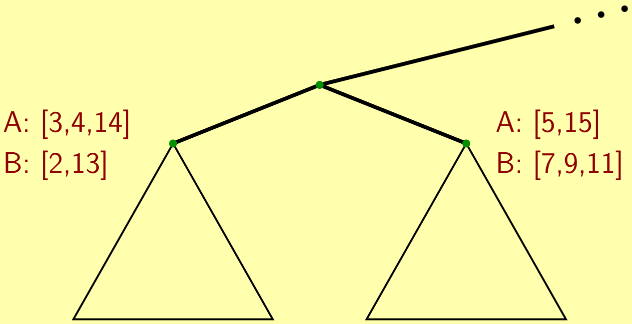


Finding *maximal* exact repeats

Idea:



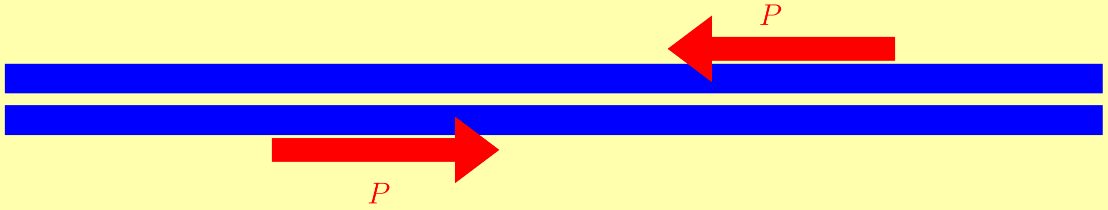
- For right-maximality (X $\neq$ Y)
  - consider only internal nodes of $T(S)$
  - report only pairs of leaves from different subtrees (or from different leaf-lists)



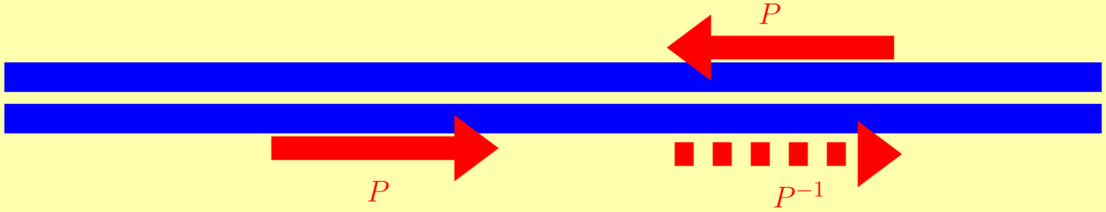
- For left-maximality (A $\neq$ B)
  - keep lists for the different left-characters
  - report only pairs from different lists

Analysis: $\mathcal{O}(n+z)$ time with $z = |\text{output}|$, $\mathcal{O}(n)$ space

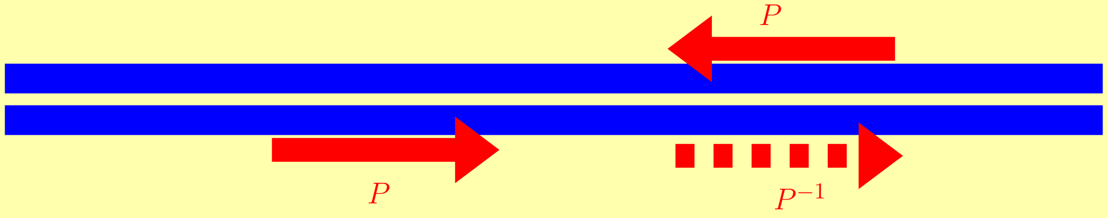
Variation: Palindromic repeats



Variation: Palindromic repeats

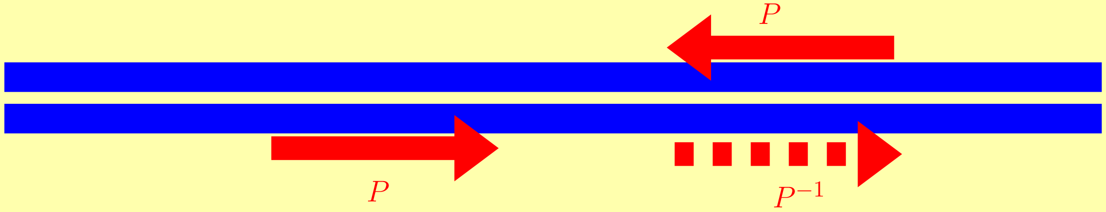


Variation: Palindromic repeats

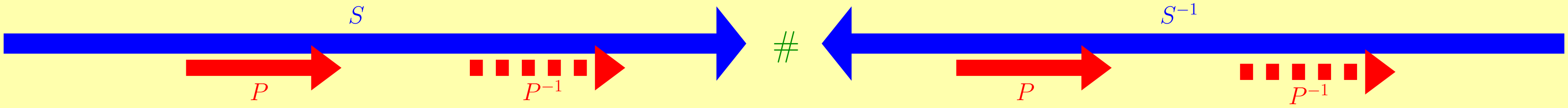


- One repeat instance must be reverse Watson/Crick complement P^{-1}.
- Essentially same problem as computing direct repeats.
- Instead of S use $S\#S^{-1}$ (where S^{-1} is the reverse complement of S).

Variation: Palindromic repeats



- One repeat instance must be reverse Watson/Crick complement P^{-1}.
- Essentially same problem as computing direct repeats.
- Instead of S use $S\#S^{-1}$ (where S^{-1} is the reverse complement of S).



- $\#$ is a unique separator symbol.
- One of the duplicates must be in S and the other in S^{-1}.
- Calculate position in S^{-1} relative to the beginning of S.

Summary: Repeats and suffix trees

Some results: (z is always the output size)

- Find all z maximal repeats in $\mathcal{O}(n+z)$ time.
- Find all z maximal palindromic repeats in $\mathcal{O}(n+z)$ time.
- Find all z tandem repeats in $\mathcal{O}(n \log n + z)$ time or $\mathcal{O}(n+z)$ time.
- Find all z maximal repeats with bounded gap in $\mathcal{O}(n \log n + z)$ time.
- Find all z maximal repeats with lower-bounded gap in $\mathcal{O}(n+z)$ time.
- Find all degenerate repeats with $\leq k$ errors in $\mathcal{O}(n+\zeta k^3)$ time ($E(\zeta) = \mathcal{O}\left(n^2/4^s\right)$).

Tandem repeats: Definitions

- tandem repeat (square)

$$w = \boxed{\alpha}\boxed{\alpha} \qquad \alpha \in \Sigma^{+}$$

Tandem repeats: Definitions

- tandem repeat (square)

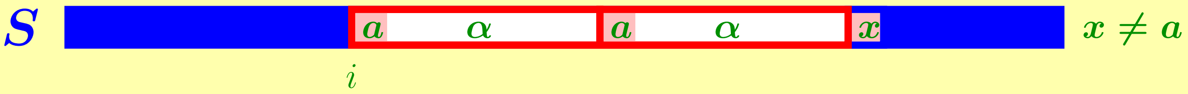
- occurrence of a tandem repeat

Tandem repeats: Definitions

- tandem repeat (square)

- occurrence of a tandem repeat

- (right-) branching occurrence of a tandem repeat

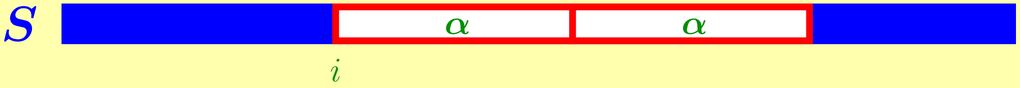


Tandem repeats: Definitions

- tandem repeat (square)

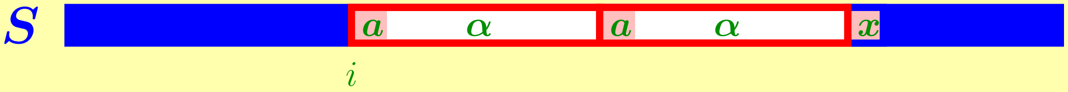
$\alpha \in \Sigma^+$

- occurrence of a tandem repeat



$(i, |\alpha|, 2)$

- (right-) branching occurrence of a tandem repeat



$x \neq a$

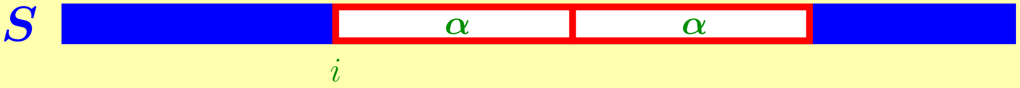
- a string w is primitive if and only if $w = u^k$ implies $k = 1$

Tandem repeats: Definitions

- tandem repeat (square)

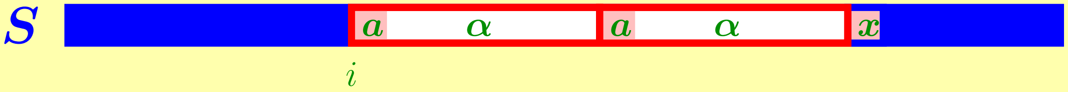
$\alpha \in \Sigma^+$

- occurrence of a tandem repeat



$(i, |\alpha|, 2)$

- (right-) branching occurrence of a tandem repeat



$x \neq a$

- a string w is primitive if and only if $w = u^k$ implies $k = 1$
- a tandem repeat $\alpha\alpha$ is primitive if and only if α is primitive

Finding tandem repeats: Overview

A. Find all ccurrences of tandem repeats in a string.

- Main/Lorentz, 1979/1984
- Landau/Schmidt, 1993

B. Find all occurrences of *primitive* tandem repeats in a string.

- Crochemore, 1981
- Apostolico/Preparata, 1983

C. Find all occurrences of primitive tandem *arrays* in a string.

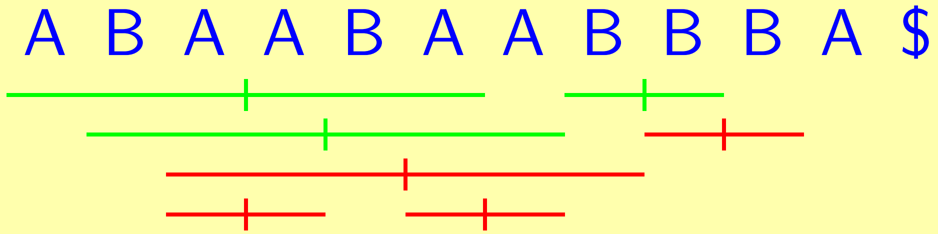
Here:
Simple and flexible detection of all of these in optimal time using a suffix tree.
(TCS 2002, joint work with Dan Gusfield)

Basic observation

Lemma:

Any non-branching occurrence $(i, l, 2)$ of a tandem repeat is the left-rotation of another tandem repeat $(i+1, l, 2)$, starting one position to its right.

Example:



Suffix trees and tandem repeats

Lemma: (folklore)
Consider two positions i and j of S, $1 \leq i < j \leq n$, let $l = j - i$. Then the following assertions are equivalent:

(a) $(i, l, 2)$ is an occurrence of a tandem repeat;

(b) i and j occur in the same leaf-list of some node v in $T(S)$ with depth $D(v) \geq l$.

Example:

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

Suffix trees and tandem repeats

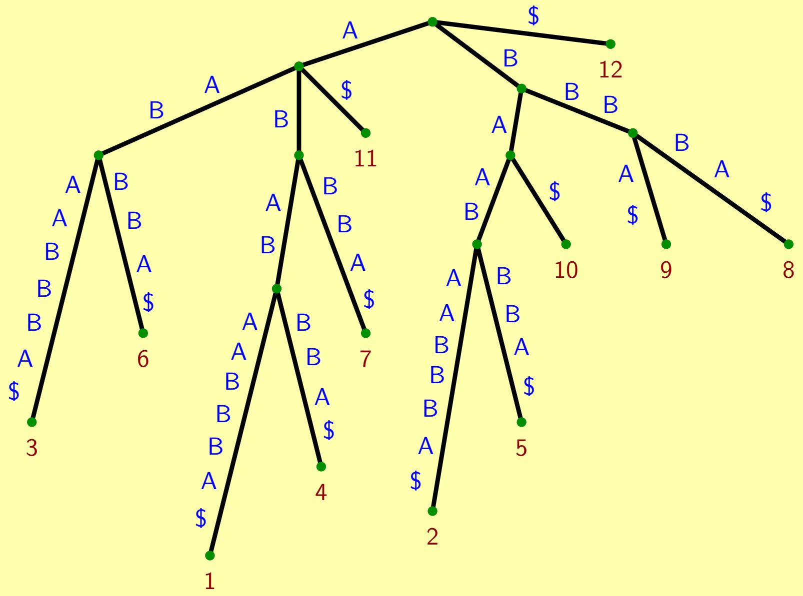
Lemma: (folklore)
Consider two positions i and j of S, $1 \leq i < j \leq n$, let $l = j - i$. Then the following assertions are equivalent:

(a) $(i, l, 2)$ is an occurrence of a tandem repeat;

(b) i and j occur in the same leaf-list of some node v in $T(S)$ with depth $D(v) \geq l$.

Example:

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



Suffix trees and tandem repeats

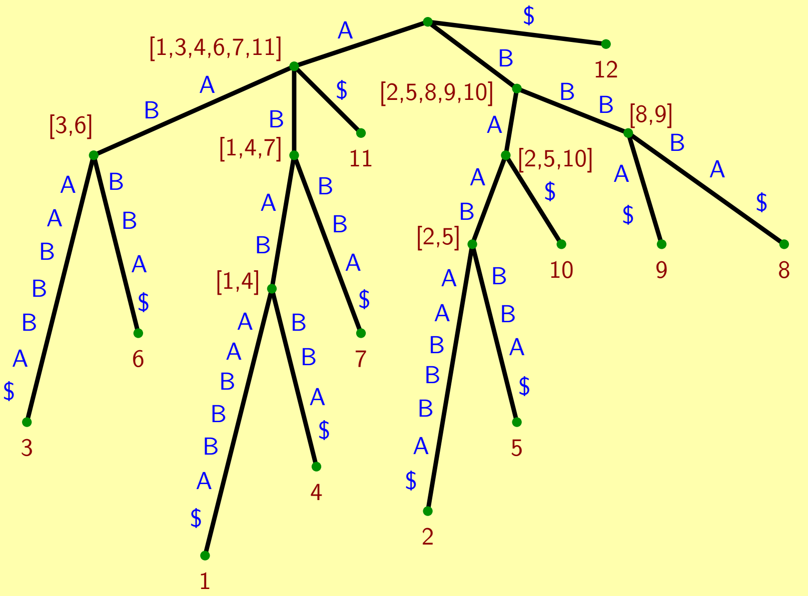
Lemma: (folklore)
Consider two positions i and j of S, $1 \leq i < j \leq n$, let $l = j - i$. Then the following assertions are equivalent:

(a) $(i, l, 2)$ is an occurrence of a tandem repeat;

(b) i and j occur in the same leaf-list of some node v in $T(S)$ with depth $D(v) \geq l$.

Example:

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



Suffix trees and tandem repeats

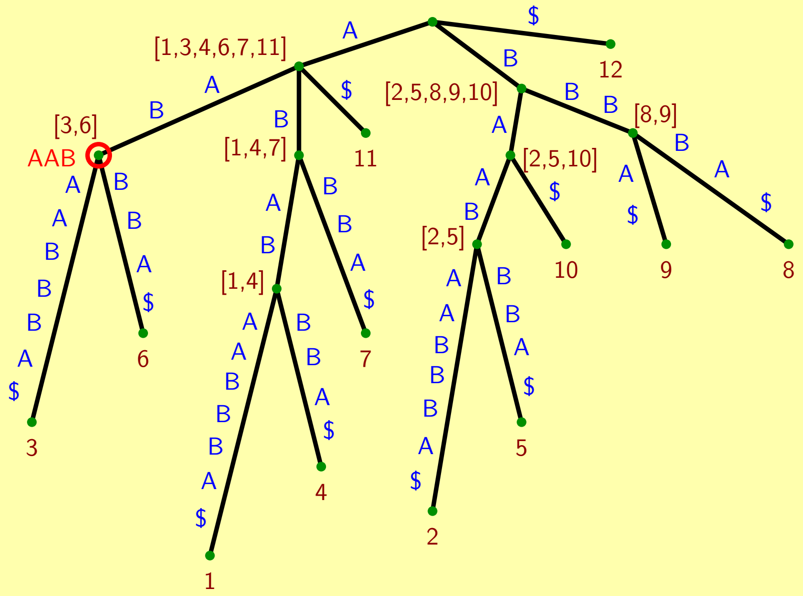
Lemma: (folklore)
Consider two positions i and j of S, $1 \leq i < j \leq n$, let $l = j - i$. Then the following assertions are equivalent:

(a) $(i, l, 2)$ is an occurrence of a tandem repeat;

(b) i and j occur in the same leaf-list of some node v in $T(S)$ with depth $D(v) \geq l$.

Example:

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



Suffix trees and tandem repeats

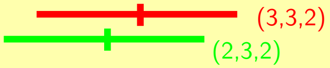
Lemma: (folklore)
Consider two positions i and j of S, $1 \leq i < j \leq n$, let $l = j - i$. Then the following assertions are equivalent:

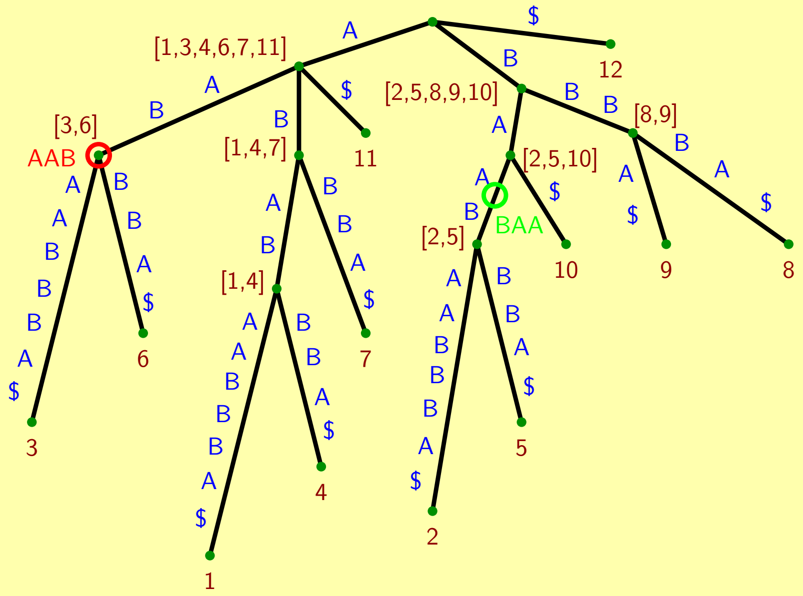
(a) $(i, l, 2)$ is an occurrence of a tandem repeat;

(b) i and j occur in the same leaf-list of some node v in $T(S)$ with depth $D(v) \geq l$.

Example:

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





Suffix trees and *branching* tandem repeats

Lemma:

Consider two positions i and j of S, $1 \leq i < j \leq n$, let $l = j - i$. Then the following assertions are equivalent:

(a) $(i, l, 2)$ is an occurrence of a branching tandem repeat;

(b) i and j occur in the same leaf-list of some node v in $T(S)$ with depth $D(v) = l$, but do not appear in the same leaf-list of any node with depth greater than l.

Suffix trees and *branching* tandem repeats

Lemma:

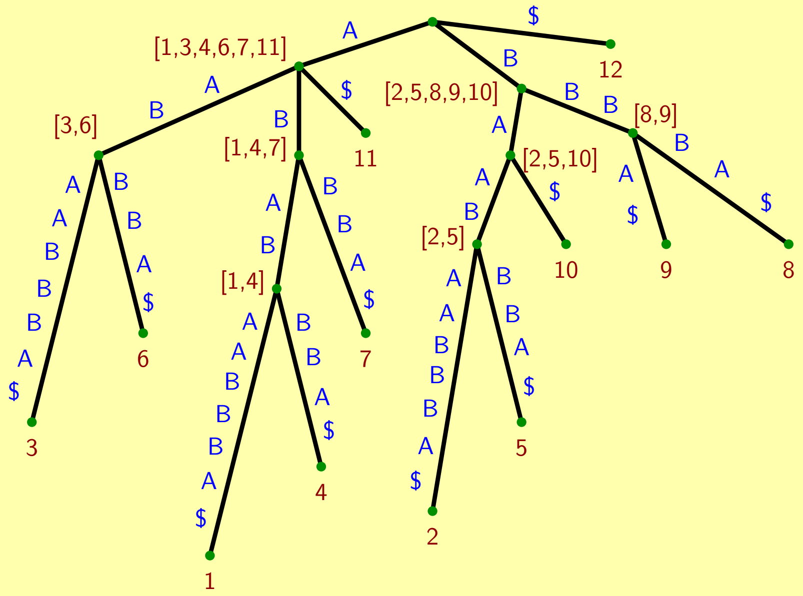
Consider two positions i and j of S, $1 \leq i < j \leq n$, let $l = j - i$. Then the following assertions are equivalent:

(a) $(i, l, 2)$ is an occurrence of a branching tandem repeat;

(b) i and j occur in the same leaf-list of some node v in $T(S)$ with depth $D(v) = l$, but do not appear in the same leaf-list of any node with depth greater than l.

Example:

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



Suffix trees and *branching* tandem repeats

Lemma:

Consider two positions i and j of S, $1 \leq i < j \leq n$, let $l = j - i$. Then the following assertions are equivalent:

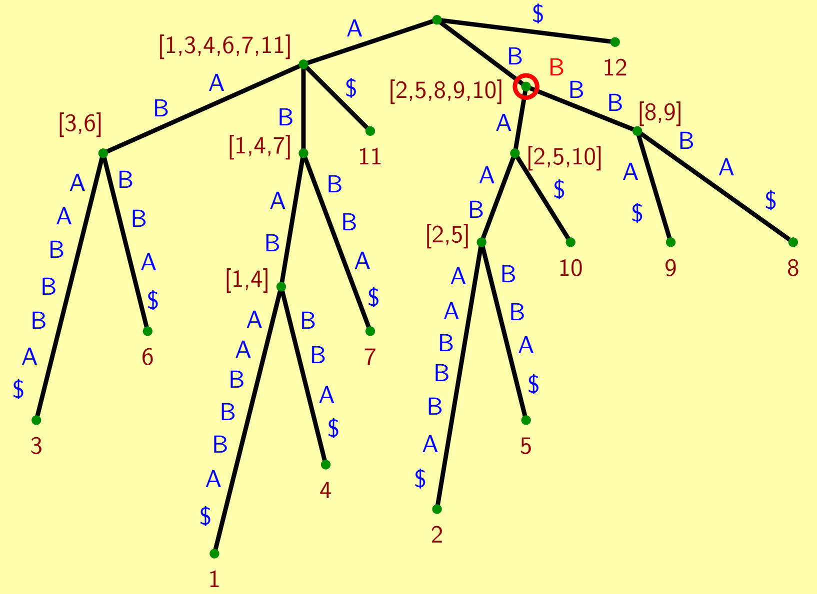
(a) $(i, l, 2)$ is an occurrence of a branching tandem repeat;

(b) i and j occur in the same leaf-list of some node v in $T(S)$ with depth $D(v) = l$, but do not appear in the same leaf-list of any node with depth greater than l.

Example:

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

(9,1,2)



Suffix trees and *branching* tandem repeats

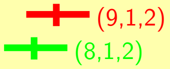
Lemma:
Consider two positions i and j of S, $1 \leq i < j \leq n$, let $l = j - i$. Then the following assertions are equivalent:

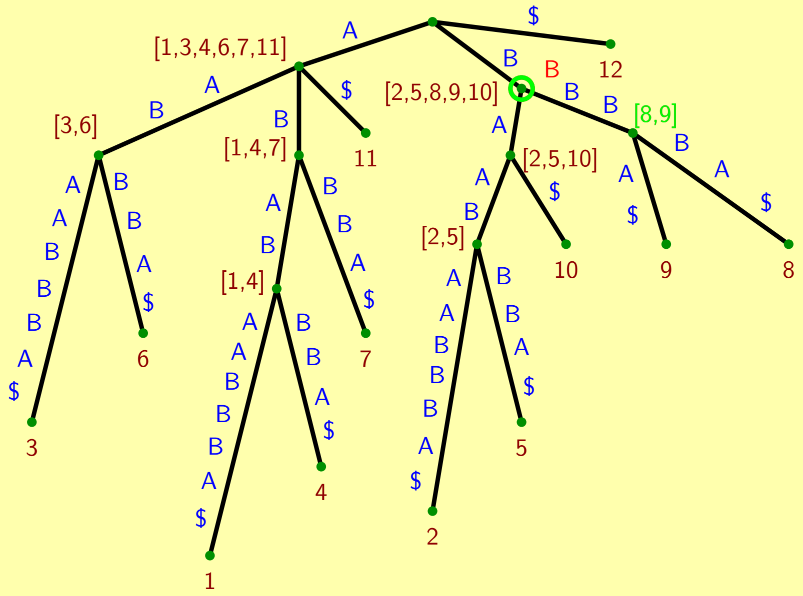
(a) $(i, l, 2)$ is an occurrence of a branching tandem repeat;

(b) i and j occur in the same leaf-list of some node v in $T(S)$ with depth $D(v) = l$, but do not appear in the same leaf-list of any node with depth greater than l.

Example:

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





Basic algorithm

Idea:
For each node v of $T(S)$, test if $\alpha\alpha = L(v)L(v)$ is a branching tandem repeat.

Algorithm:

All nodes of $T(S)$ begin unmarked.
Step 1 is repeated until all nodes are marked.

1. Select an unmarked internal node v.
   Mark v and execute steps 2a and 2b for node v.

2a. Collect the leaf-list $LL(v)$.

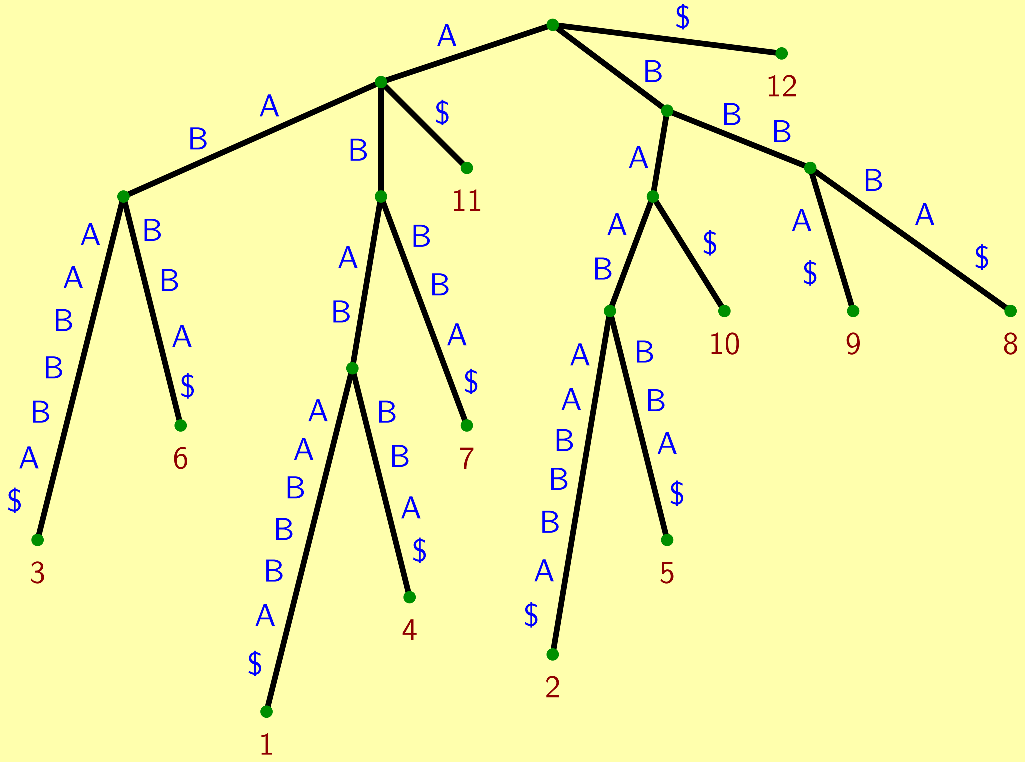
2b. For each leaf i in $LL(v)$, test whether the leaf $j = i + D(v)$ is in $LL(v)$.
If so, test whether $S[i] \neq S[i + 2D(v)]$. There is a branching tandem repeat of length $2D(v)$ starting at position i if and only if both tests return true.

Analysis: $\mathcal{O}(n^2)$ time, $\mathcal{O}(n)$ space

Testing in constant time

Depth-first numbering and look-up table:

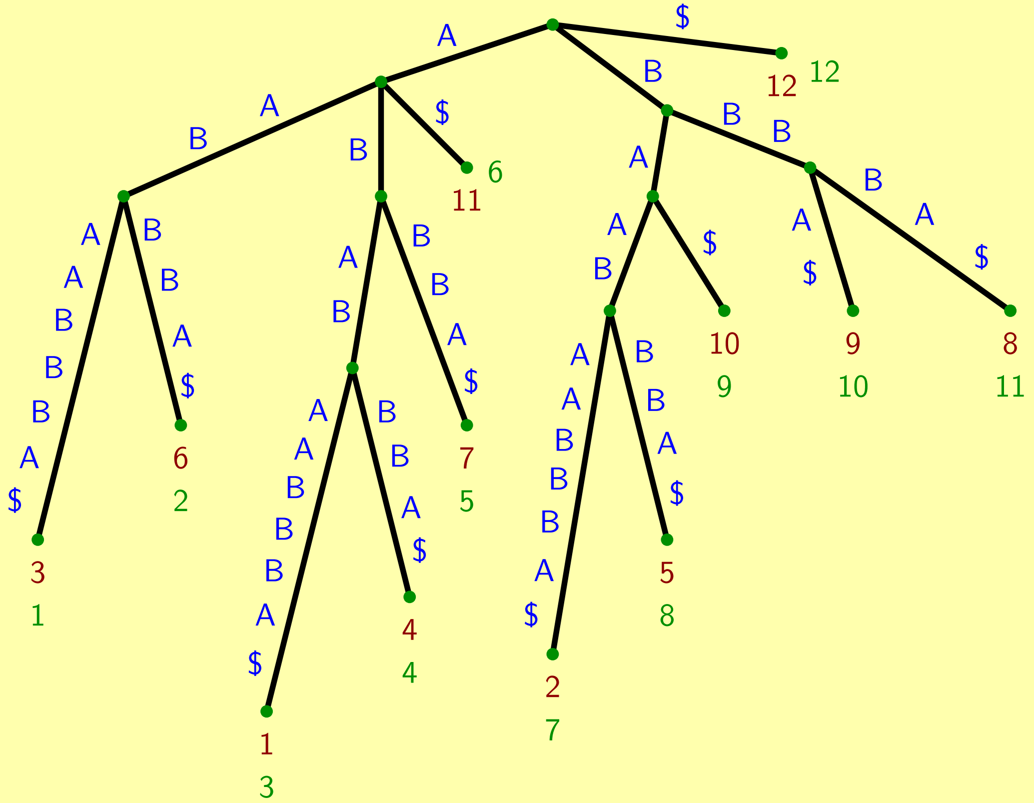
A	B	A	A	B	A	A	B	B	B	A	$
1	2	3	4	5	6	7	8	9	10	11	12



Testing in constant time

Depth-first numbering and look-up table:

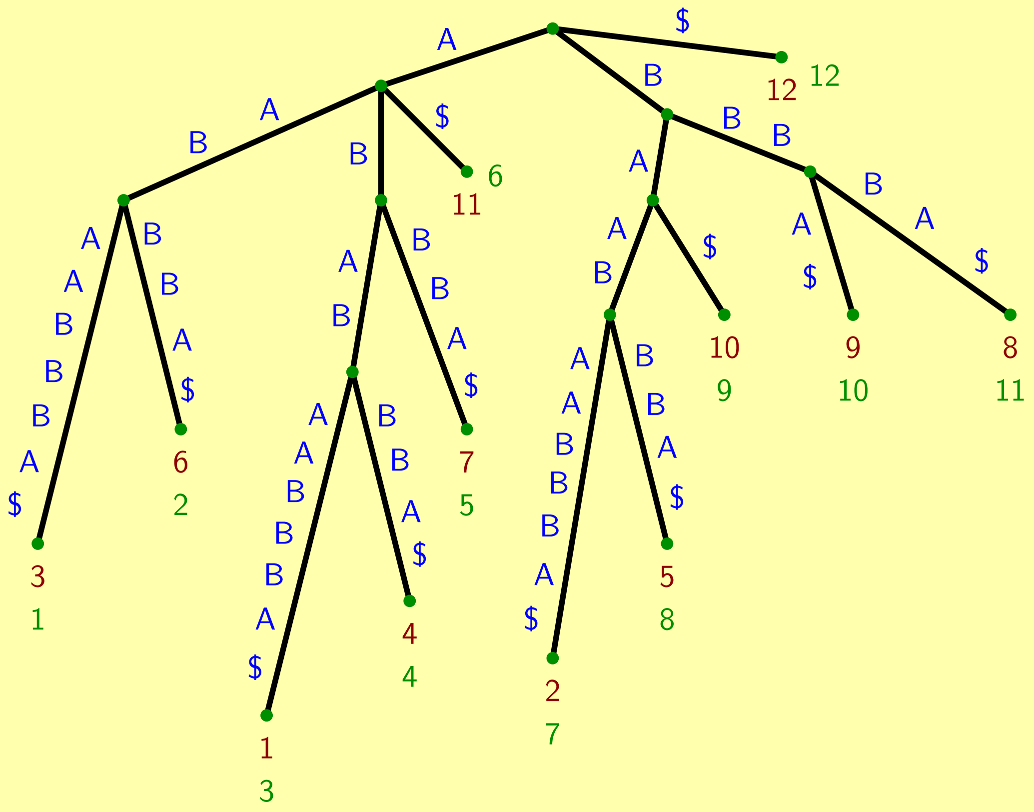
A	B	A	A	B	A	A	B	B	B	A	$
1	2	3	4	5	6	7	8	9	10	11	12



Testing in constant time

Depth-first numbering and look-up table:

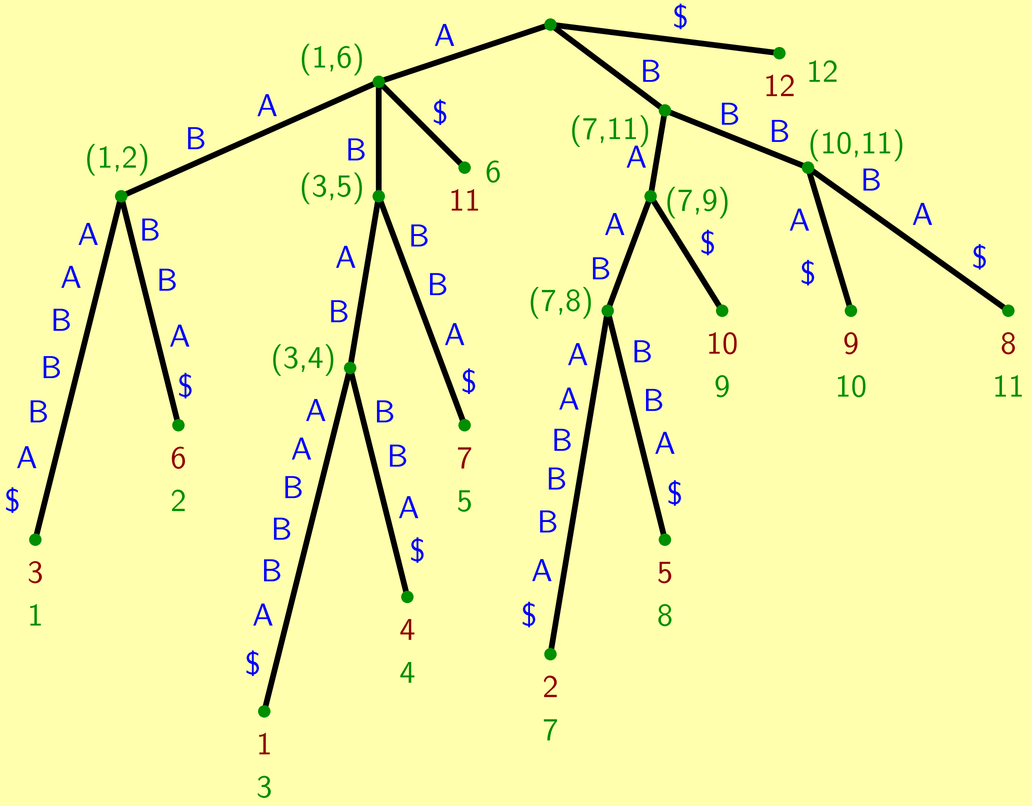
A	B	A	A	B	A	A	B	B	B	A	$
1	2	3	4	5	6	7	8	9	10	11	12
3	7	1	4	8	2	5	11	10	9	6	12



Testing in constant time

Depth-first numbering and look-up table:

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



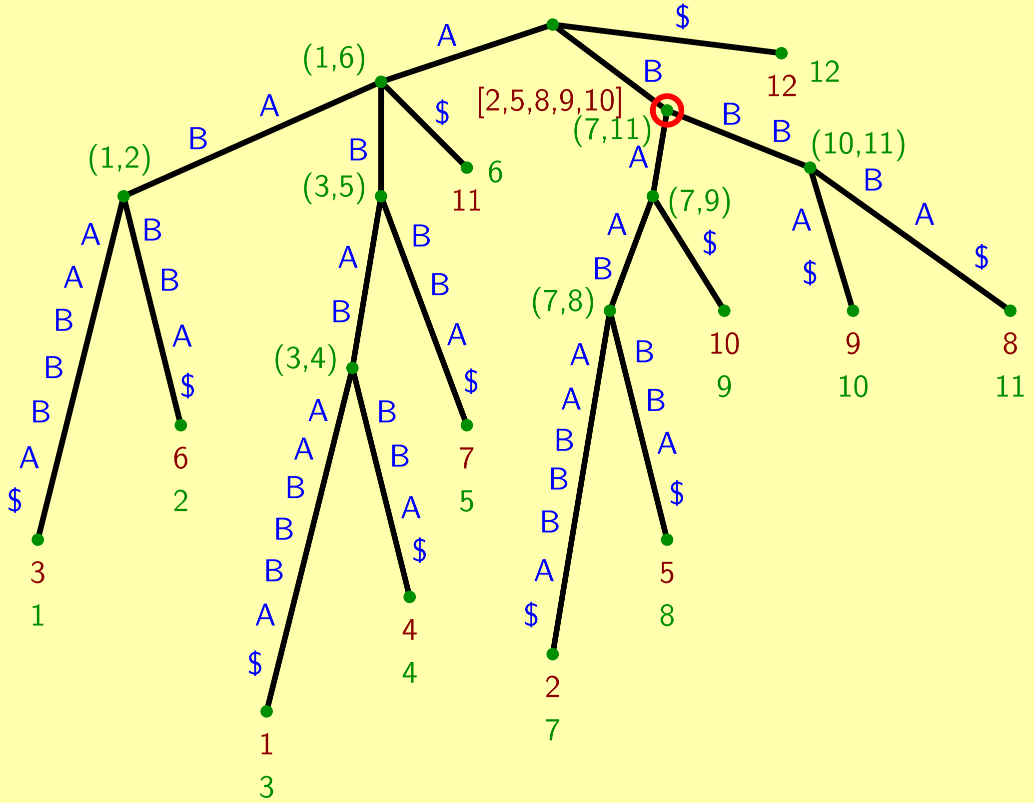
Testing in constant time

Depth-first numbering and look-up table:

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

B: $8 \xrightarrow{D=1} 9? \xrightarrow{table} 10 \in (7,11)$

but: $(8,9)$ not branching



Testing in constant time

Depth-first numbering and look-up table:

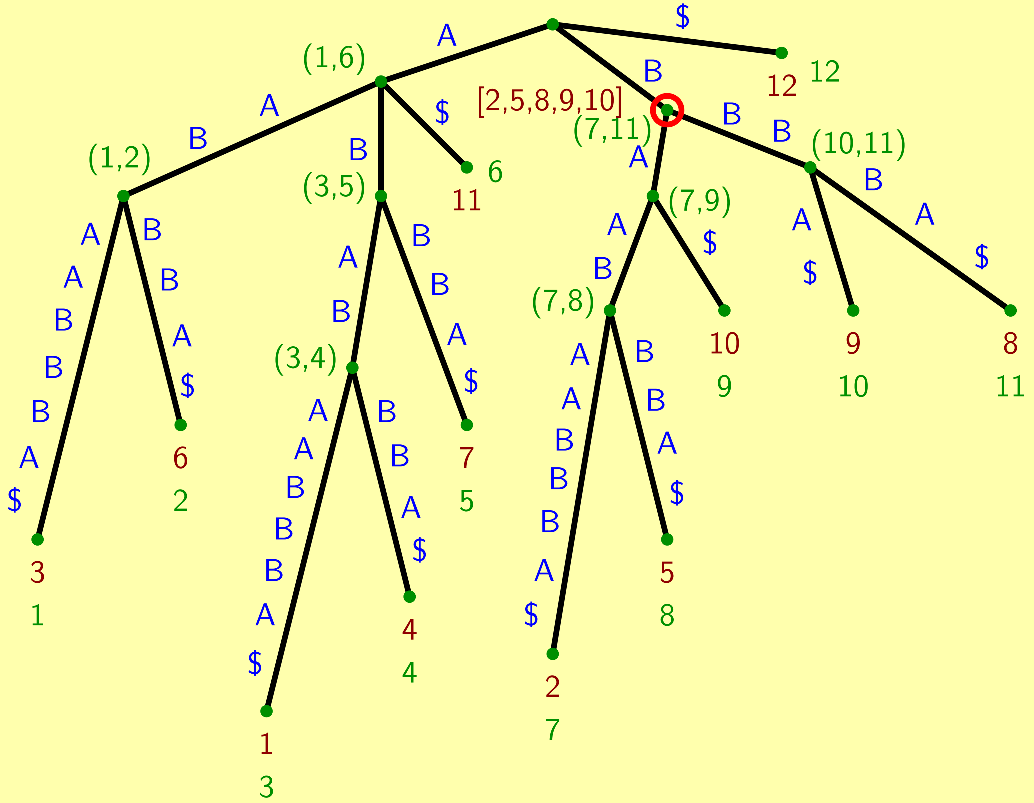
A	B	A	A	B	A	A	B	B	B	A	$
1	2	3	4	5	6	7	8	9	10	11	12
3	7	1	4	8	2	5	11	10	9	6	12

B: $8 \xrightarrow{D=1} 9? \xrightarrow{table} 10 \in (7,11)$

but: $(8,9)$ not branching

$9 \xrightarrow{D=1} 10? \xrightarrow{table} 9 \in (7,11)$

and: $(9,10)$ is branching



Testing in constant time

Depth-first numbering and look-up table:

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

B: $8 \xrightarrow{D=1} 9? \xrightarrow{\textit{table}} 10 \in (7,11)$

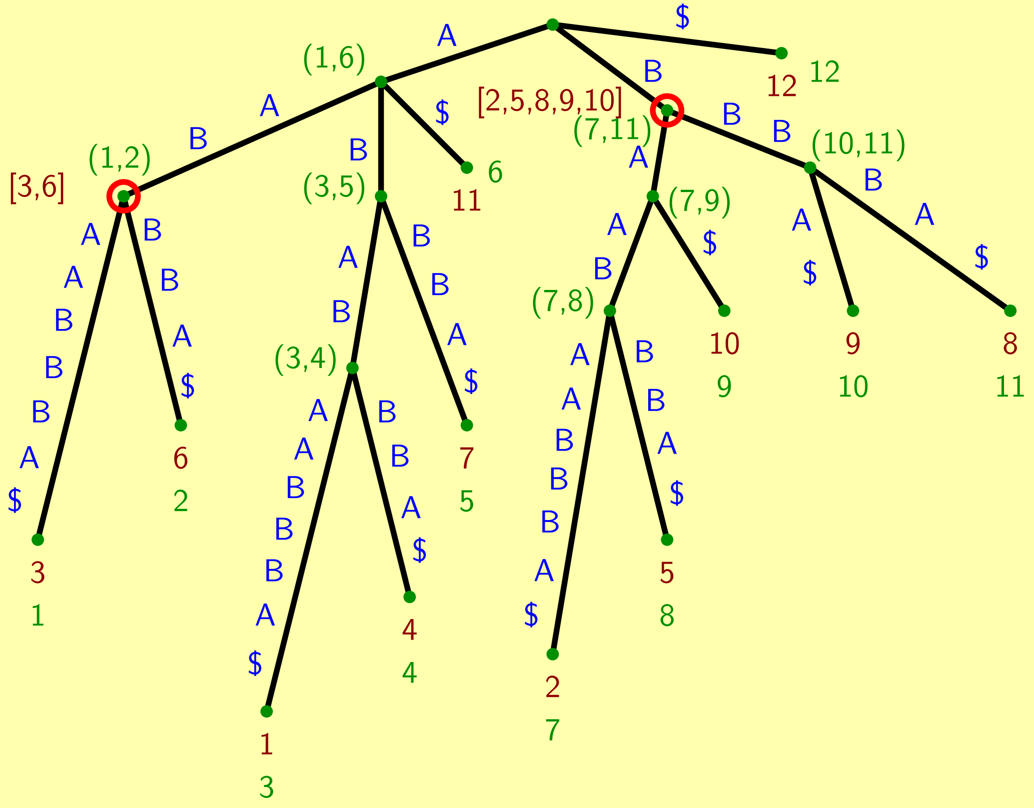
but: $(8,9)$ not branching

$9 \xrightarrow{D=1} 10? \xrightarrow{\textit{table}} 9 \in (7,11)$

and: $(9,10)$ is branching

AAB: $3 \xrightarrow{D=3} 6? \xrightarrow{\textit{table}} 2 \in (1,2)$

and: $(3,6)$ is branching



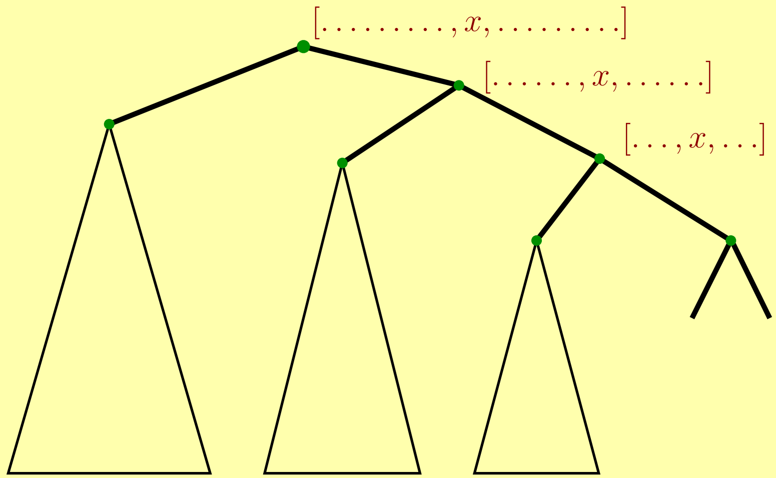
Speedup of the basic algorithm

Definitions

- For each node v, v' denotes the child of v with the largest leaf-list.
- $LL'(v)$ denotes $LL(v) - LL(v')$.

The “Smaller Half” Trick

It is well known that $\sum_v |LL'(v)| \;\leq\; n \log_2 n$.



Any value x can be
in at most $\log_2 n$ leaf-lists LL'.

Optimized basic algorithm

Algorithm:

All nodes of $T(S)$ begin unmarked.
Step 1 is repeated until all nodes are marked.

1. Select an unmarked internal node v.
   Mark v and execute steps 2a, 2b and 2c for node v.

2a. Collect the list $LL'(v)$ for v.

2b. For each leaf i in $LL'(v)$, test whether leaf $j = i + D(v)$ is in $LL(v)$, the leaf-list of v. If so, test whether $S[i] \neq S[i + 2D(v)]$. There is a branching tandem repeat of length $2D(v)$ starting at position i if and only if both tests return true.

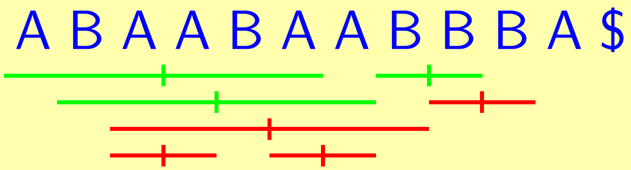
2c. Do the same test for each leaf j in $LL'(v)$, and $i = j - D(v)$.

Analysis: $\mathcal{O}(n \log n)$ time, $\mathcal{O}(n)$ space

Putting things together

Finding all tandem repeats

Starting at each of the branching occurrences, do a series of consecutive left-rotations to find all z tandem repeats.



Analysis: $\mathcal{O}(n \log n + z)$ time, no additional space

Tandem repeats in linear time!

Fraenkel & Simpson, 1998:

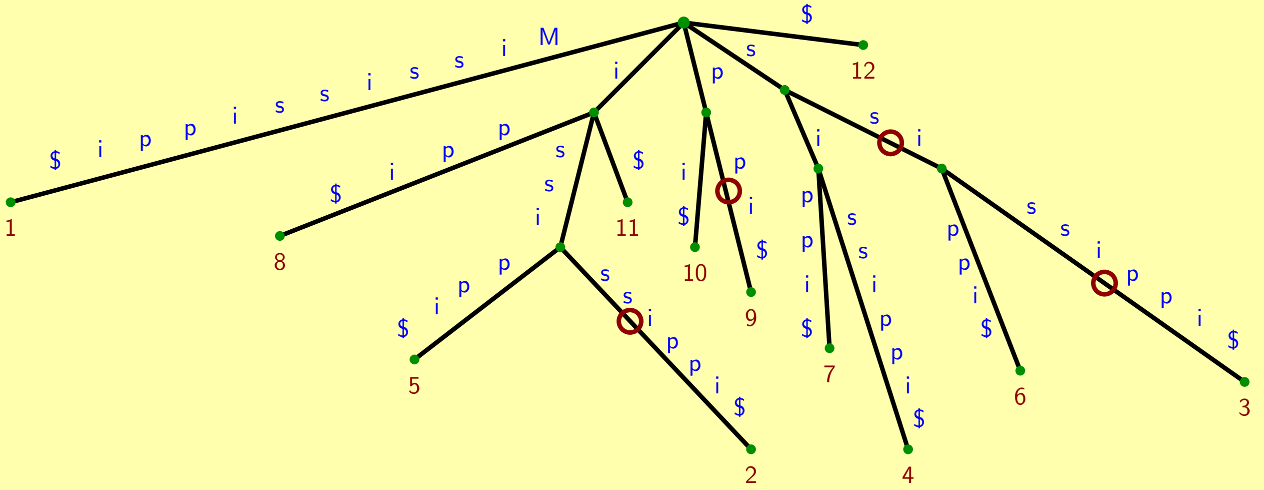
- The vocabulary of all tandem repeats in S has only $\mathcal{O}(n)$ elements.

Idea:

- Mark in $T(S)$ all end points of tandem repeats.

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

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



Analysis: $\mathcal{O}(n + |\text{output}|)$ time, $\mathcal{O}(n)$ space

Overview

- Introduction: Pattern matching in biosequence analysis
- A flexible index structure for sequence analysis: The suffix tree
  - Basic definitions
  - Application 1: Repeats, tandem repeats
  - Application 2: Multiple Genome Aligner (MGA)
- An even more flexible data structure: The affix tree
  - Basic definitions
  - Application: Search for palindromic patterns
- Conclusion

Multiple Genome Aligner

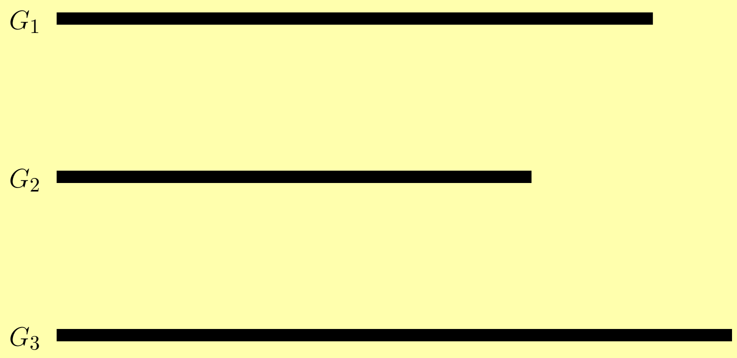
Simultaneous comparison of multiple genomes (Höhl *et al.*, ISMB 2002).

Multiple Genome Aligner

Simultaneous comparison of multiple genomes (Höhl *et al.*, ISMB 2002).

Algorithm:

1. Find all maximal multiple exact matches (multiMEMs) in the given genomes (similar to repeats, using the *generalized* suffix tree).

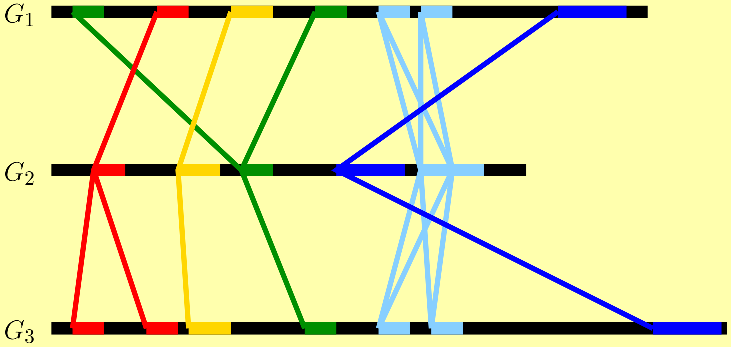


Multiple Genome Aligner

Simultaneous comparison of multiple genomes (Höhl *et al.*, ISMB 2002).

Algorithm:

1. Find all maximal multiple exact matches (multiMEMs) in the given genomes (similar to repeats, using the *generalized* suffix tree).

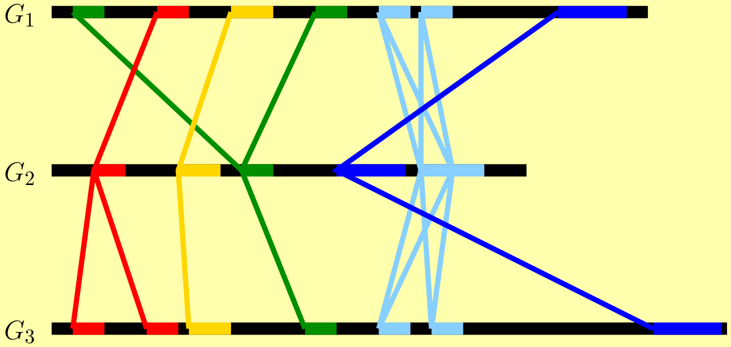


Multiple Genome Aligner

Simultaneous comparison of multiple genomes (Höhl *et al.*, ISMB 2002).

Algorithm:

1. Find all maximal multiple exact matches (multiMEMs) in the given genomes (similar to repeats, using the *generalized* suffix tree).
2. Select from all multiMEMs an optimal set, i.e. a chain of non-overlapping multiMEMs of maximal weight where the weight of a chain is the sum of the lengths of its members.

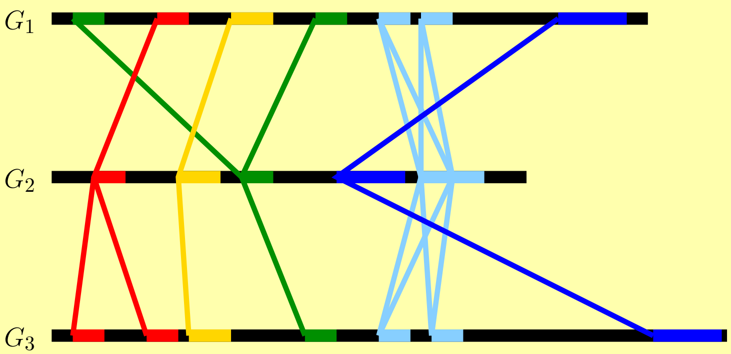


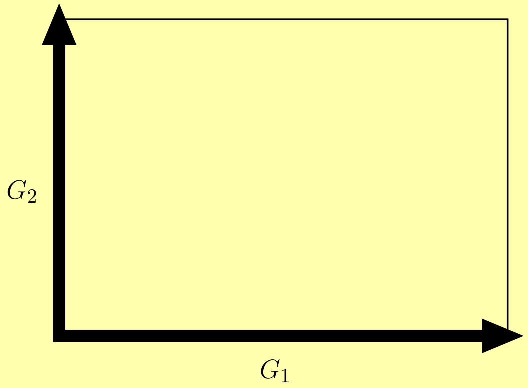
Multiple Genome Aligner

Simultaneous comparison of multiple genomes (Höhl *et al.*, ISMB 2002).

Algorithm:

1. Find all maximal multiple exact matches (multiMEMs) in the given genomes (similar to repeats, using the *generalized* suffix tree).
2. Select from all multiMEMs an optimal set,
   i.e. a chain of non-overlapping multiMEMs of maximal weight
   where the weight of a chain is the sum of the lengths of its members.



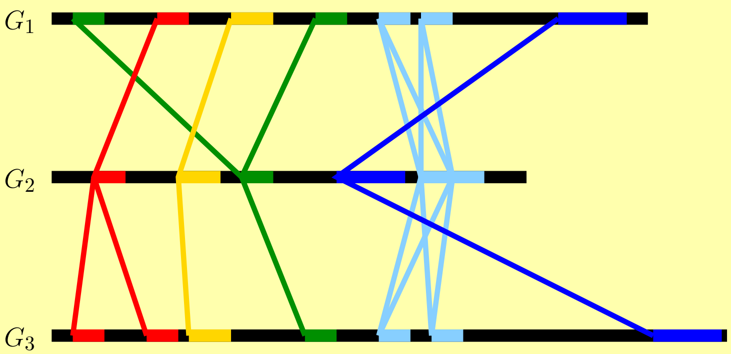


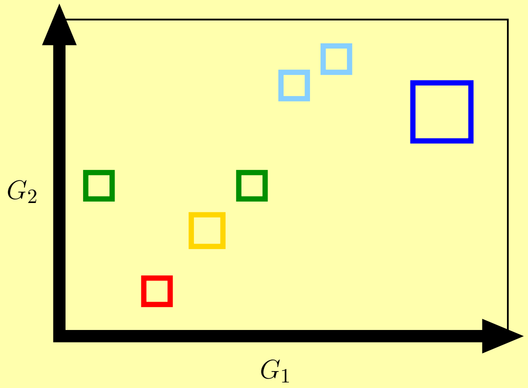
Multiple Genome Aligner

Simultaneous comparison of multiple genomes (Höhl *et al.*, ISMB 2002).

Algorithm:

1. Find all maximal multiple exact matches (multiMEMs) in the given genomes (similar to repeats, using the *generalized* suffix tree).
2. Select from all multiMEMs an optimal set,
   i.e. a chain of non-overlapping multiMEMs of maximal weight
   where the weight of a chain is the sum of the lengths of its members.



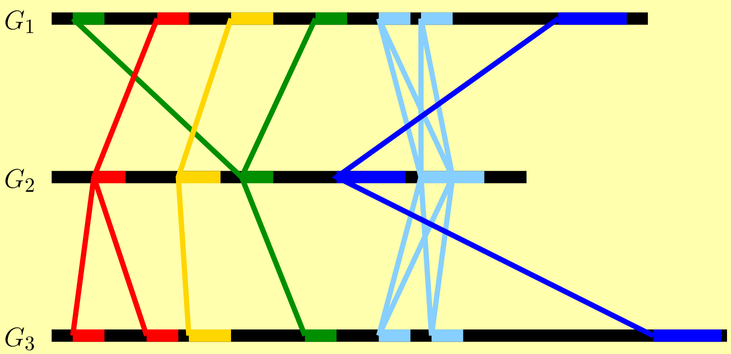


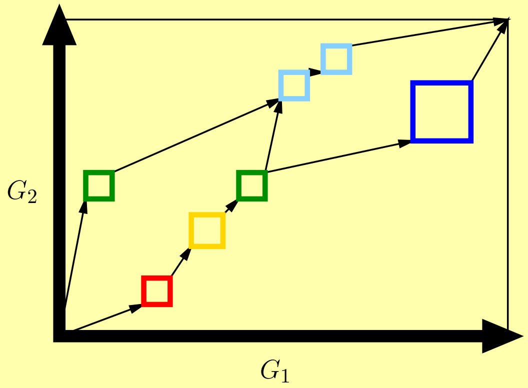
Multiple Genome Aligner

Simultaneous comparison of multiple genomes (Höhl *et al.*, ISMB 2002).

Algorithm:

1. Find all maximal multiple exact matches (multiMEMs) in the given genomes (similar to repeats, using the *generalized* suffix tree).
2. Select from all multiMEMs an optimal set,
   i.e. a chain of non-overlapping multiMEMs of maximal weight
   where the weight of a chain is the sum of the lengths of its members.



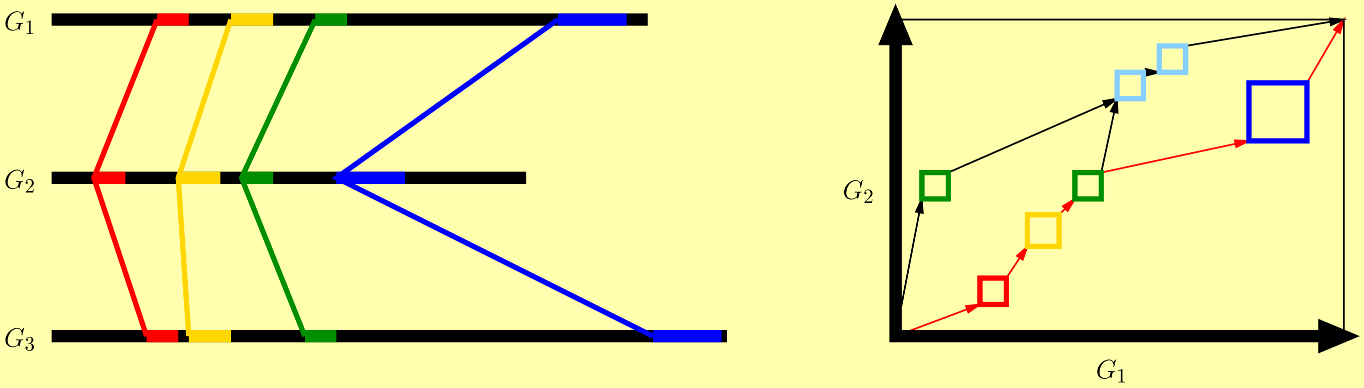


Multiple Genome Aligner

Simultaneous comparison of multiple genomes (Höhl *et al.*, ISMB 2002).

Algorithm:

1. Find all maximal multiple exact matches (multiMEMs) in the given genomes (similar to repeats, using the *generalized* suffix tree).
2. Select from all multiMEMs an optimal set,
   i.e. a chain of non-overlapping multiMEMs of maximal weight
   where the weight of a chain is the sum of the lengths of its members.

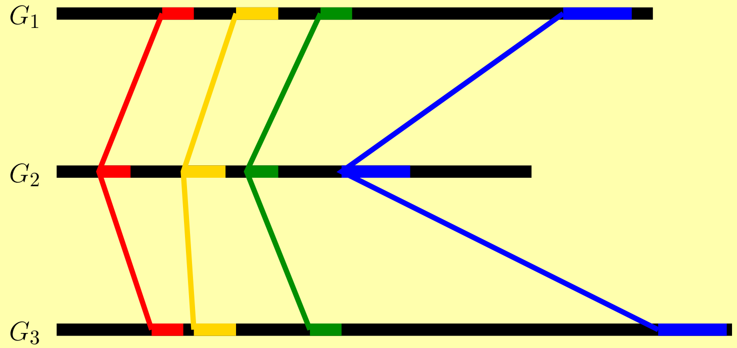


Multiple Genome Aligner

Simultaneous comparison of multiple genomes (Höhl *et al.*, ISMB 2002).

Algorithm:

1. Find all maximal multiple exact matches (multiMEMs) in the given genomes (similar to repeats, using the *generalized* suffix tree).
2. Select from all multiMEMs an optimal set,
   i.e. a chain of non-overlapping multiMEMs of maximal weight
   where the weight of a chain is the sum of the lengths of its members.

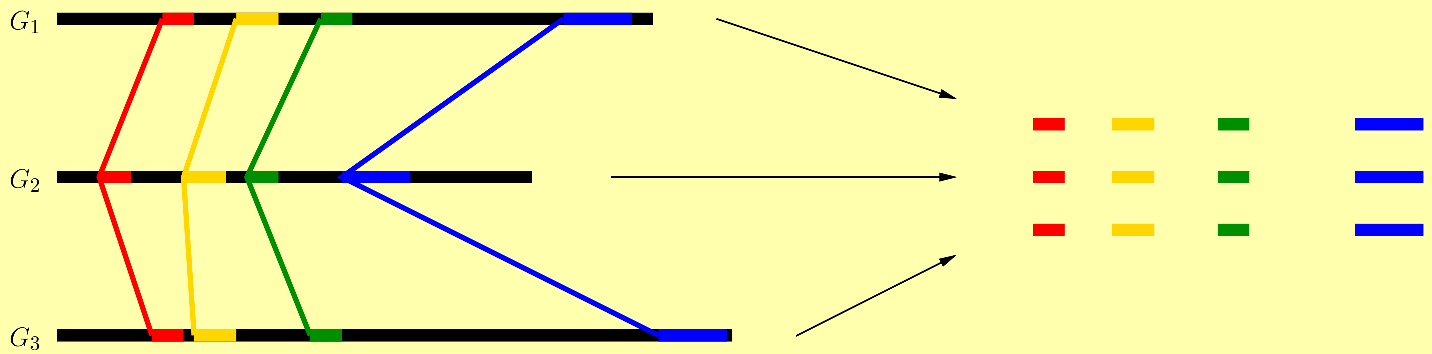


Multiple Genome Aligner

Simultaneous comparison of multiple genomes (Höhl *et al.*, ISMB 2002).

Algorithm:

1. Find all maximal multiple exact matches (multiMEMs) in the given genomes (similar to repeats, using the *generalized* suffix tree).
2. Select from all multiMEMs an optimal set,
   i.e. a chain of non-overlapping multiMEMs of maximal weight
   where the weight of a chain is the sum of the lengths of its members.
3. Close the gaps recursively, and finally by a standard alignment procedure.

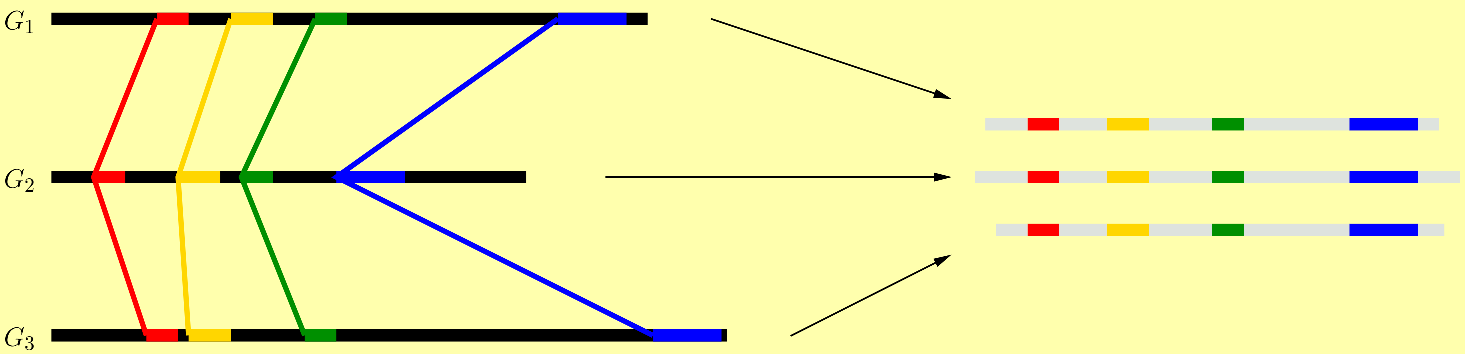


Multiple Genome Aligner

Simultaneous comparison of multiple genomes (Höhl *et al.*, ISMB 2002).

Algorithm:

1. Find all maximal multiple exact matches (multiMEMs) in the given genomes (similar to repeats, using the *generalized* suffix tree).
2. Select from all multiMEMs an optimal set,
   i.e. a chain of non-overlapping multiMEMs of maximal weight
   where the weight of a chain is the sum of the lengths of its members.
3. Close the gaps recursively, and finally by a standard alignment procedure.



Overview

- Introduction: Pattern matching in biosequence analysis
- A flexible index structure for sequence analysis: The suffix tree
  - Basic definitions
  - Application 1: Repeats, tandem repeats
  - Application 2: Multiple Genome Aligner (MGA)
- An even more flexible data structure: The affix tree
  - Basic definitions
  - Application: Search for palindromic patterns
- Conclusion

Other data structures for genomic sequence analysis

- DAWG (directed acyclic word graph)
- Suffix Array: space-efficient alternative with slightly longer search time; well suited for persistent memory representation
- Enhanced Suffix Array
- Level Compressed Trie
- Suffix Cactus
- Suffix Oracle
- Affix Tree: specialized for bi-directional pattern searches

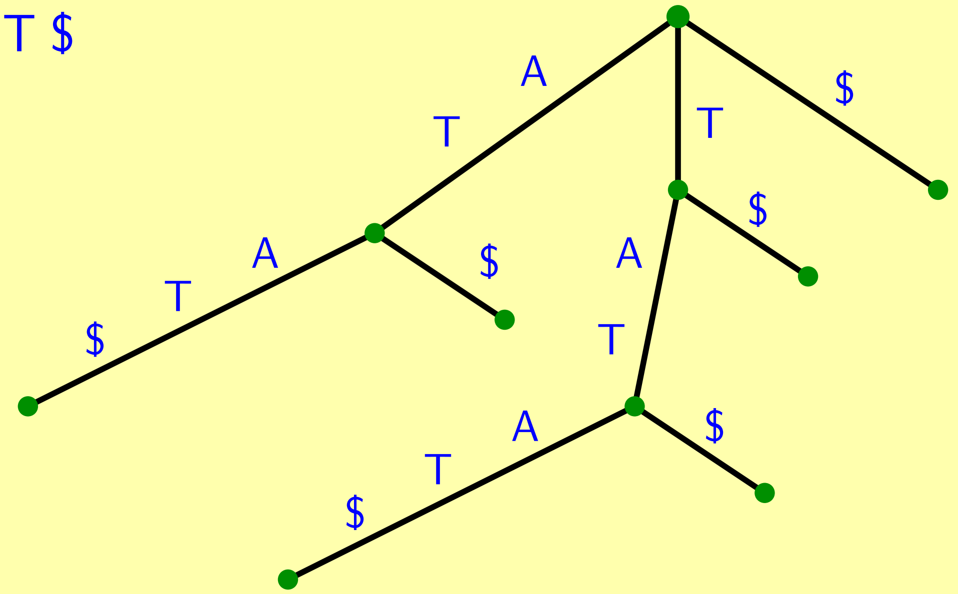
Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- But: What if we have an “inside out” pattern matching problem?

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
- But: What if we have an “inside out” pattern matching problem?

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

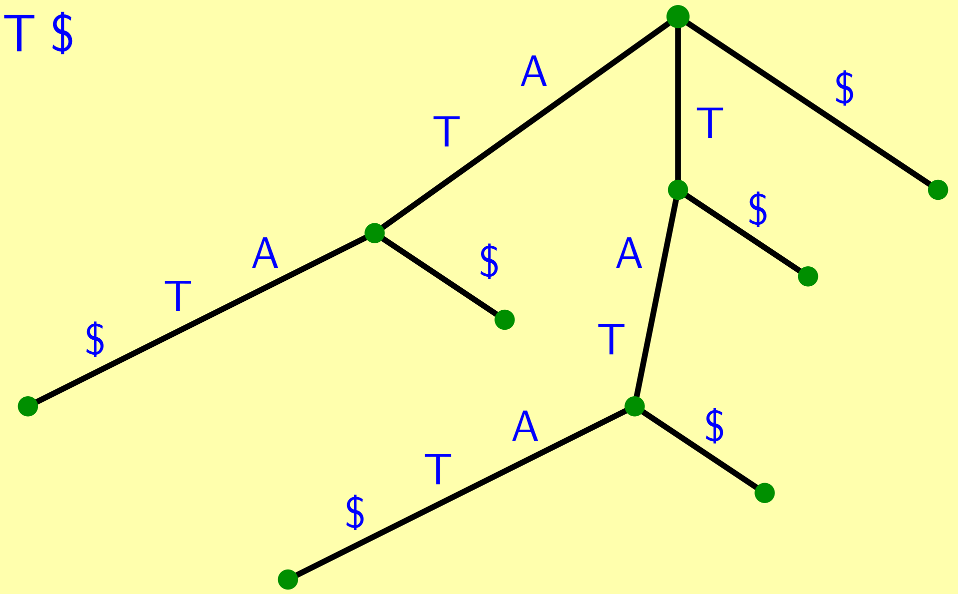


Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- But: What if we have an "inside out" pattern matching problem?

$S =$ T A T A T $

$P =$ A T A T

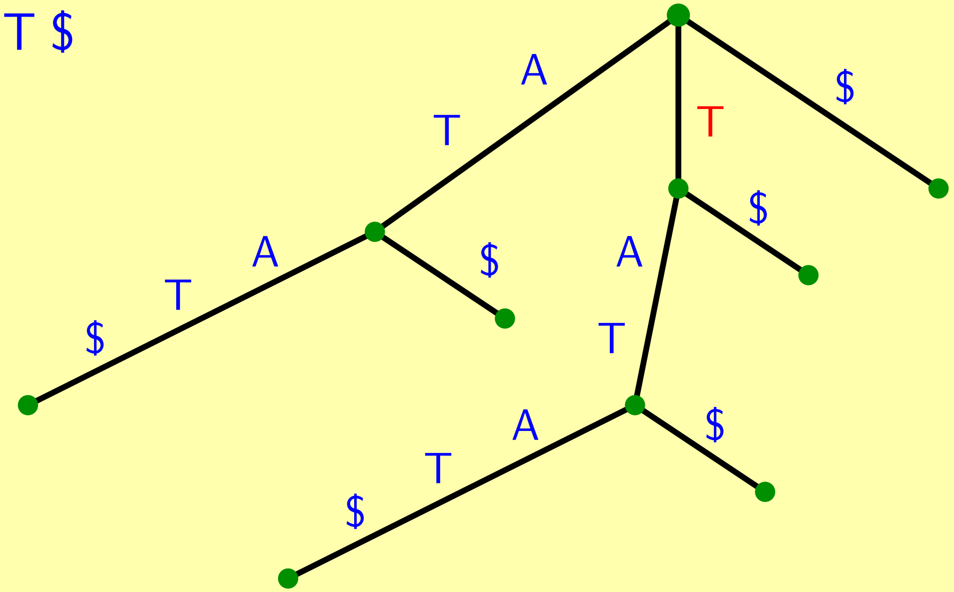


Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- But: What if we have an “inside out” pattern matching problem?

$S =$ T A T A T $

$P =$ A T A T

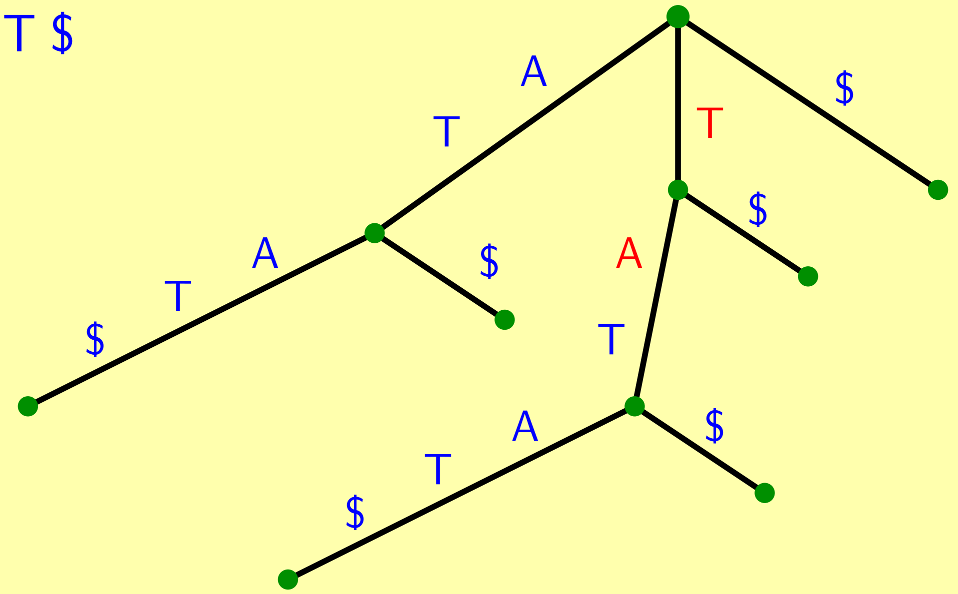


Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- But: What if we have an "inside out" pattern matching problem?

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

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

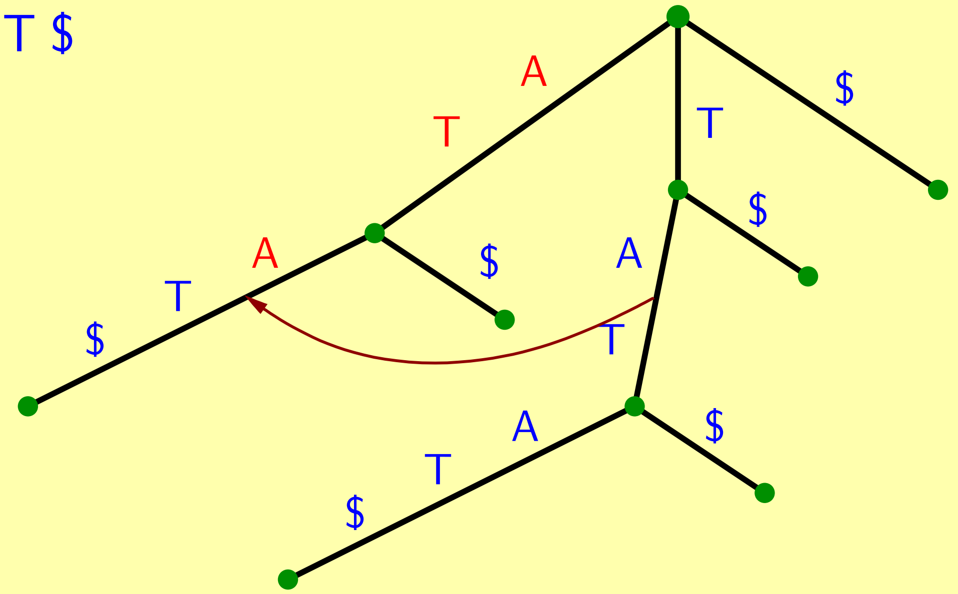


Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- But: What if we have an "inside out" pattern matching problem?

$S =$ T A T A T $

$P =$ A T A T

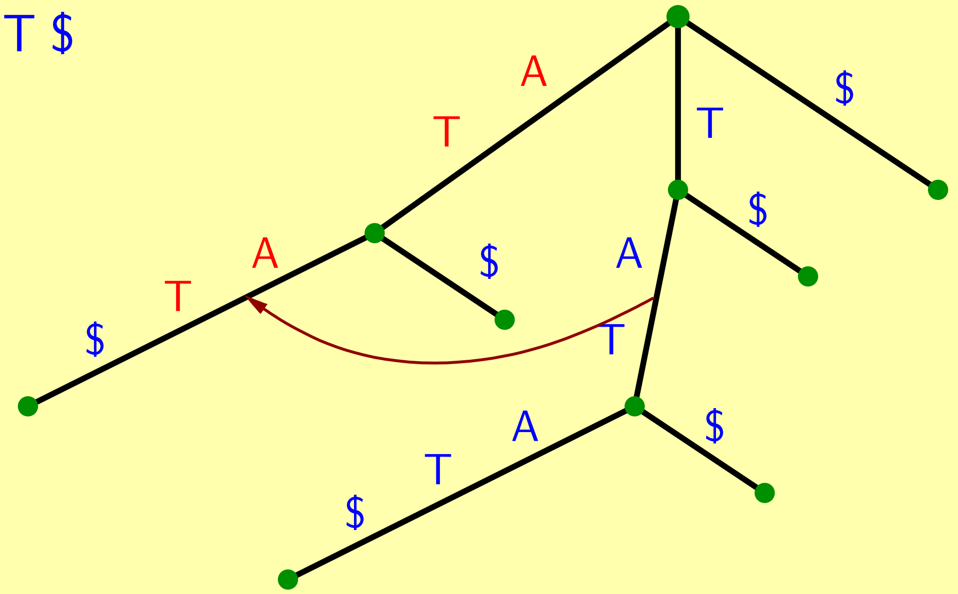


Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- But: What if we have an “inside out” pattern matching problem?

$S =$ T A T A T $

$P =$ A T A T

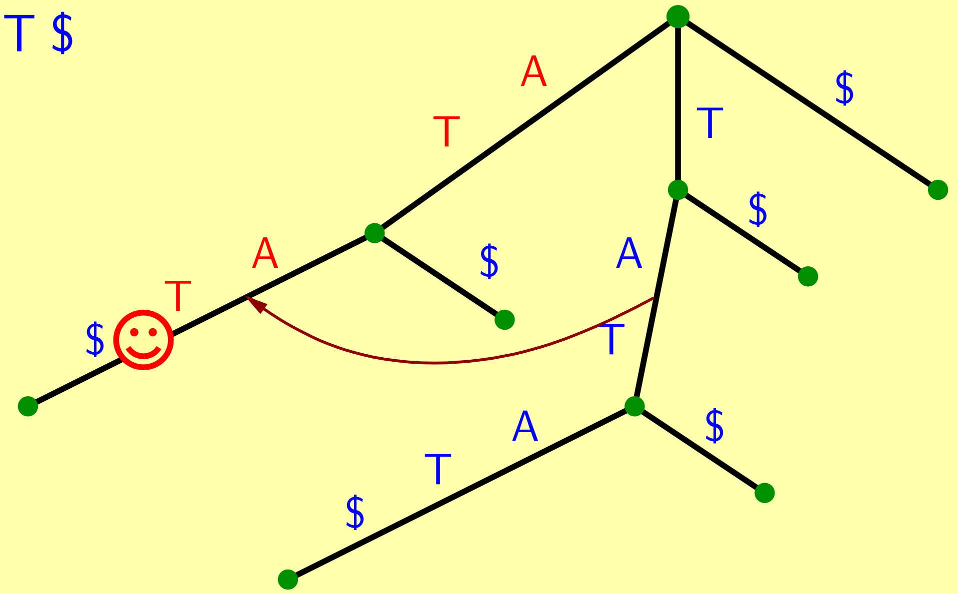


Towards a bi-directional data structure

- Suffix tree is asymmetric: left-to-right matching only
- Similar data structure for right-to-left matching: reverse prefix tree
- But: What if we have an “inside out” pattern matching problem?

$S =$ T A T A T $

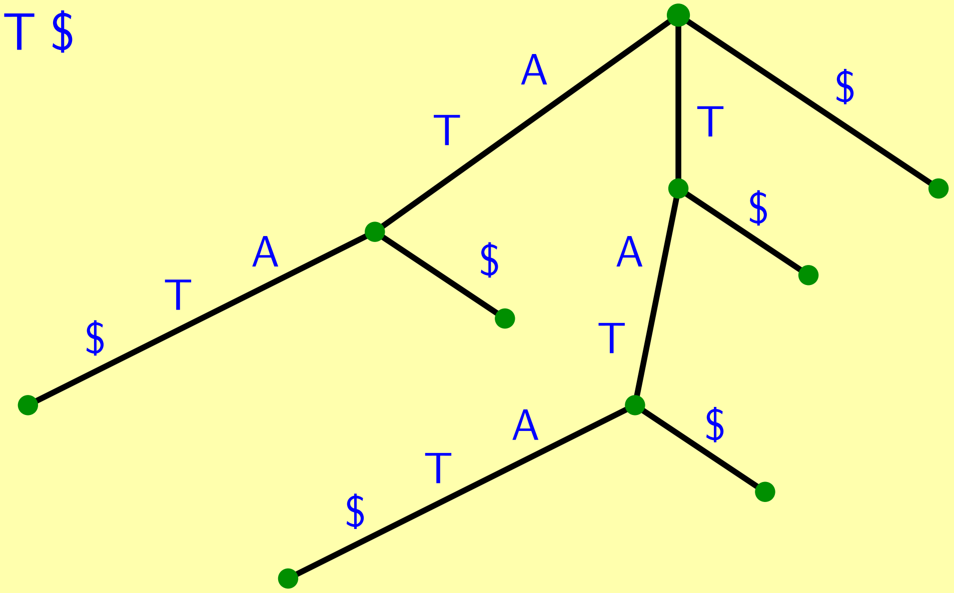
$P =$ A T A T



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
- But: What if we have an “inside out” pattern matching problem?

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



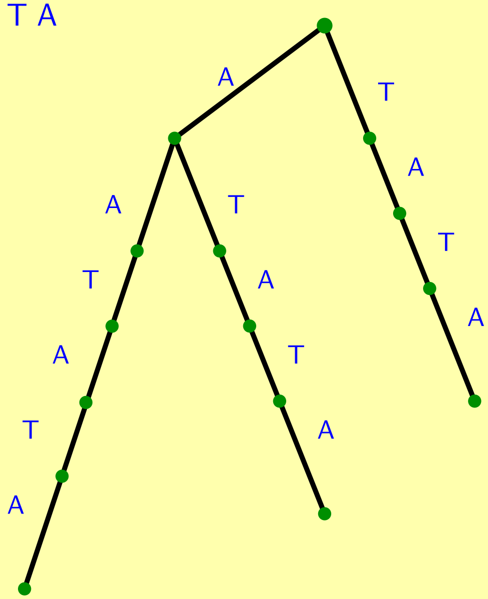
Towards a bi-directional data structure (2)

- It would be nice to have all “prefix edges” inside the same data structure.
- Idea: Create the *atomic* suffix tree and reverse prefix tree.

Towards a bi-directional data structure (2)

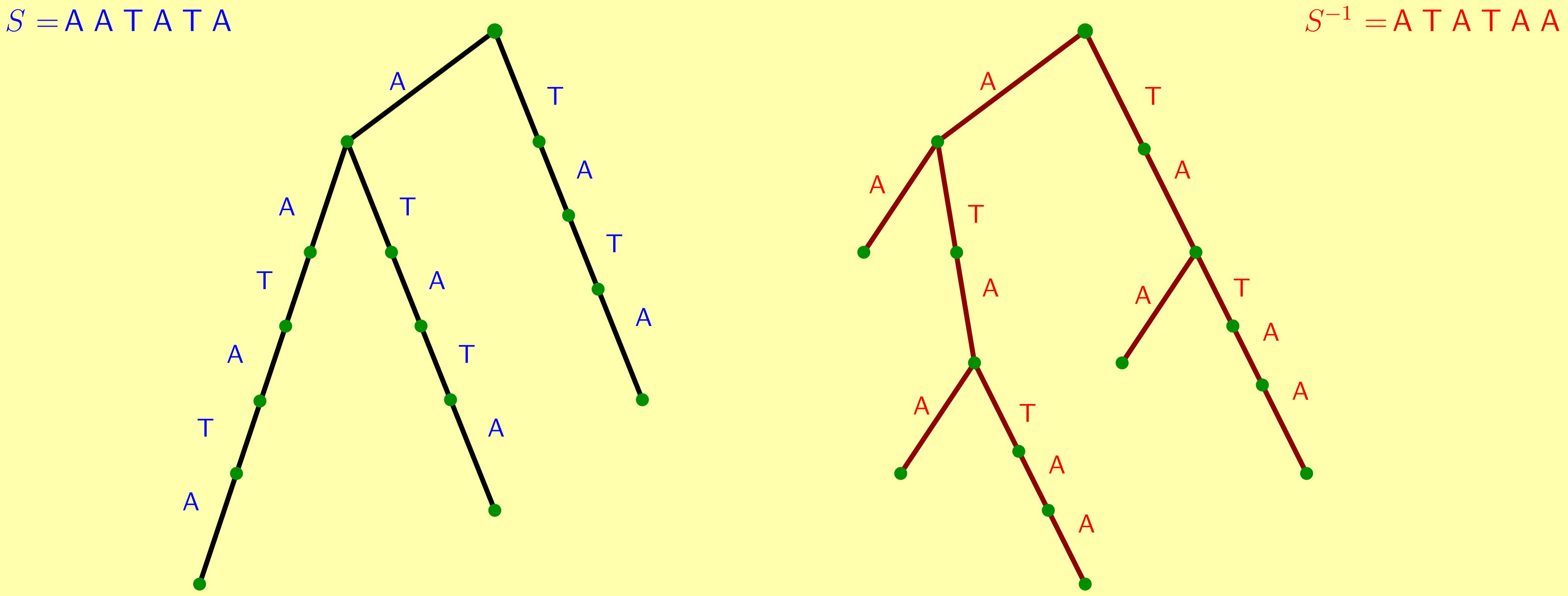
- It would be nice to have all “prefix edges” inside the same data structure.
- Idea: Create the *atomic* suffix tree and reverse prefix tree.

$S =$ A A T A T A



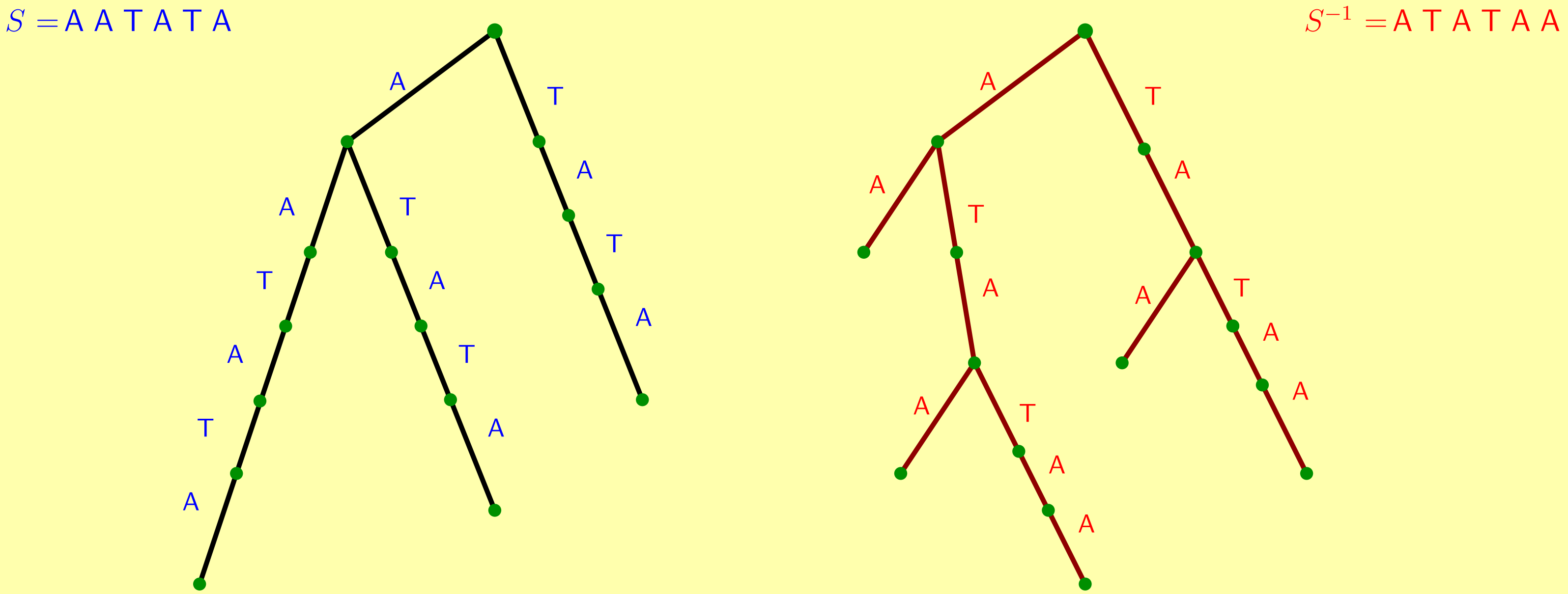
Towards a bi-directional data structure (2)

- It would be nice to have all “prefix edges” inside the same data structure.
- Idea: Create the *atomic* suffix tree and reverse prefix tree.



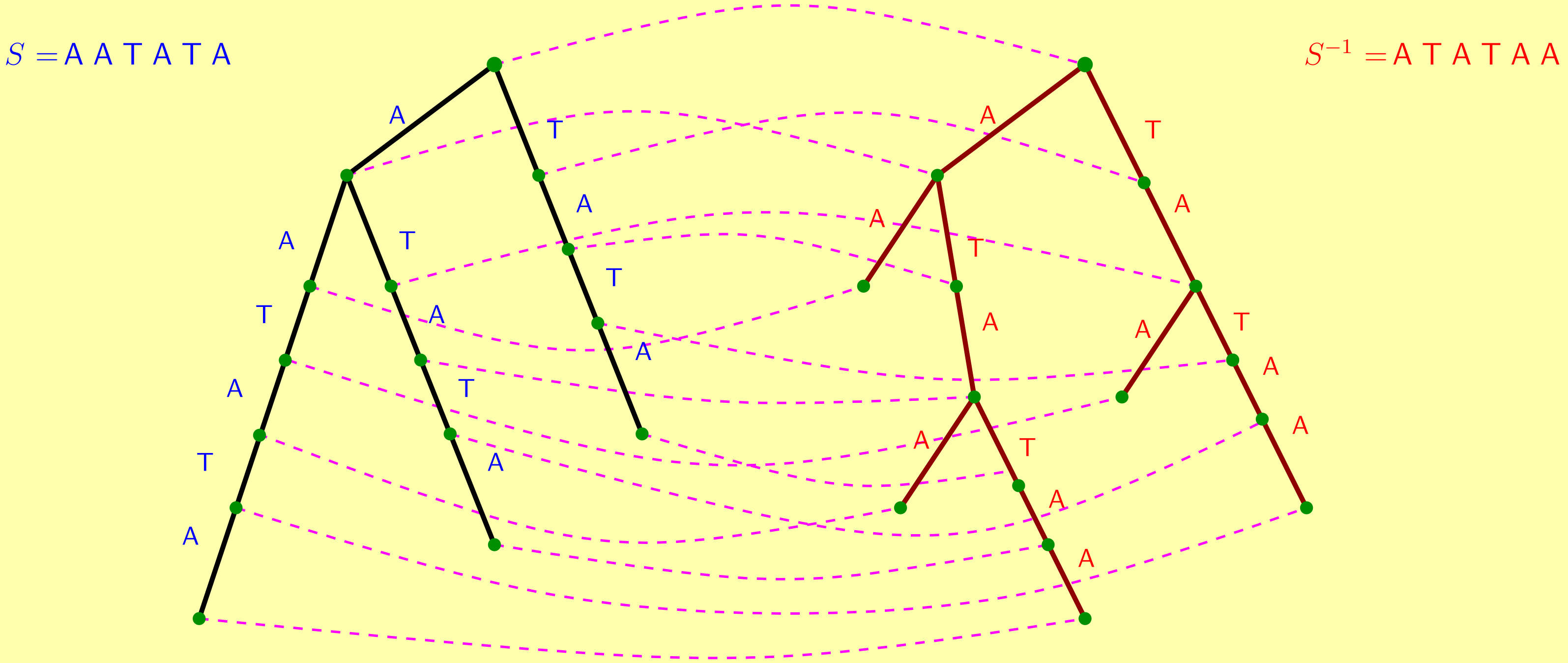
Towards a bi-directional data structure (2)

- It would be nice to have all “prefix edges” inside the same data structure.
- Idea: Create the *atomic* suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



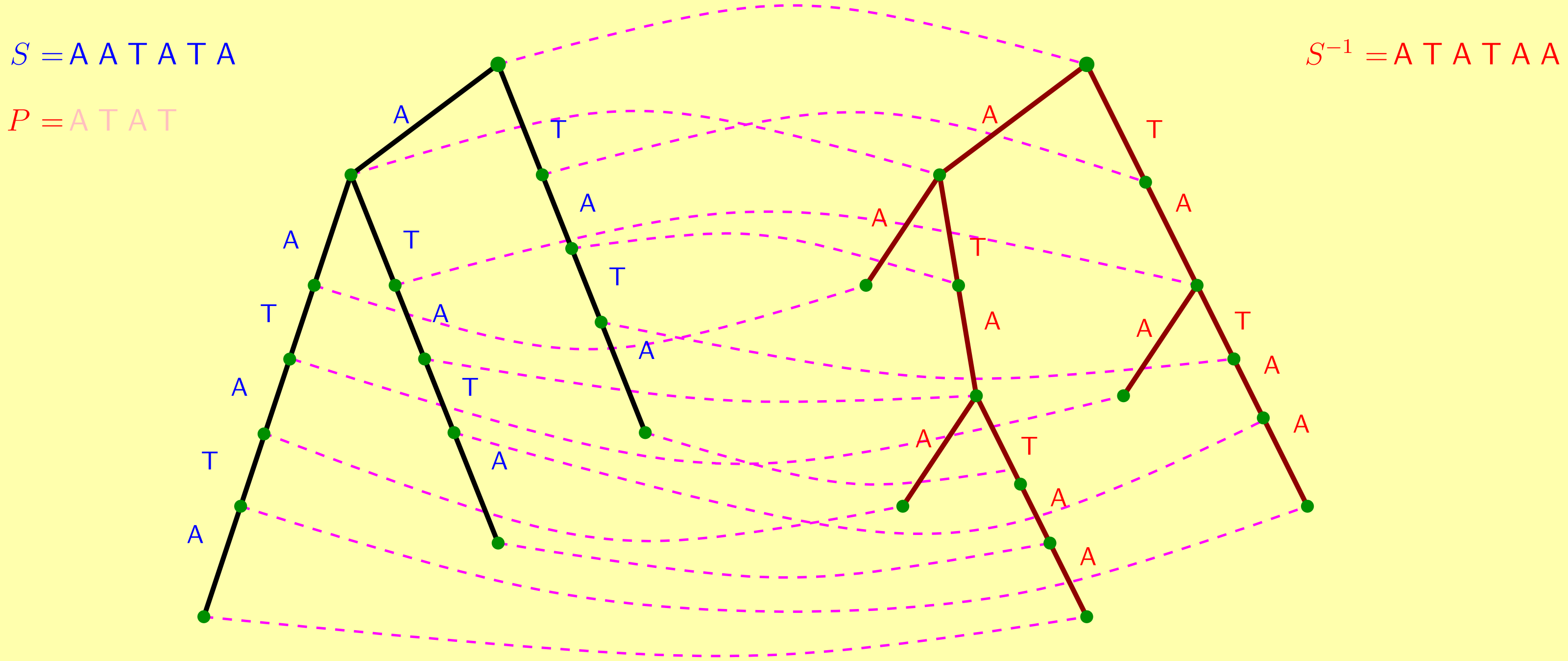
Towards a bi-directional data structure (2)

- It would be nice to have all "prefix edges" inside the same data structure.
- Idea: Create the *atomic* suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



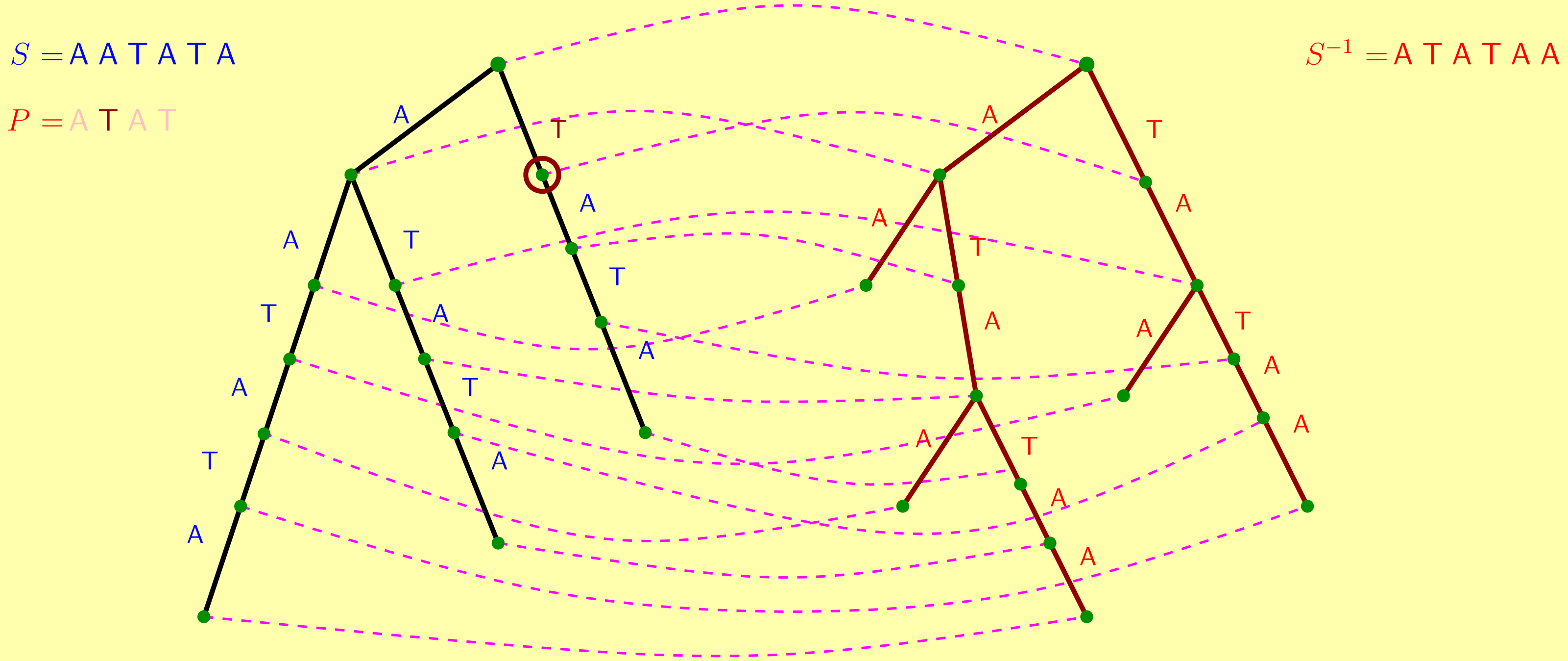
Towards a bi-directional data structure (2)

- It would be nice to have all "prefix edges" inside the same data structure.
- Idea: Create the *atomic* suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



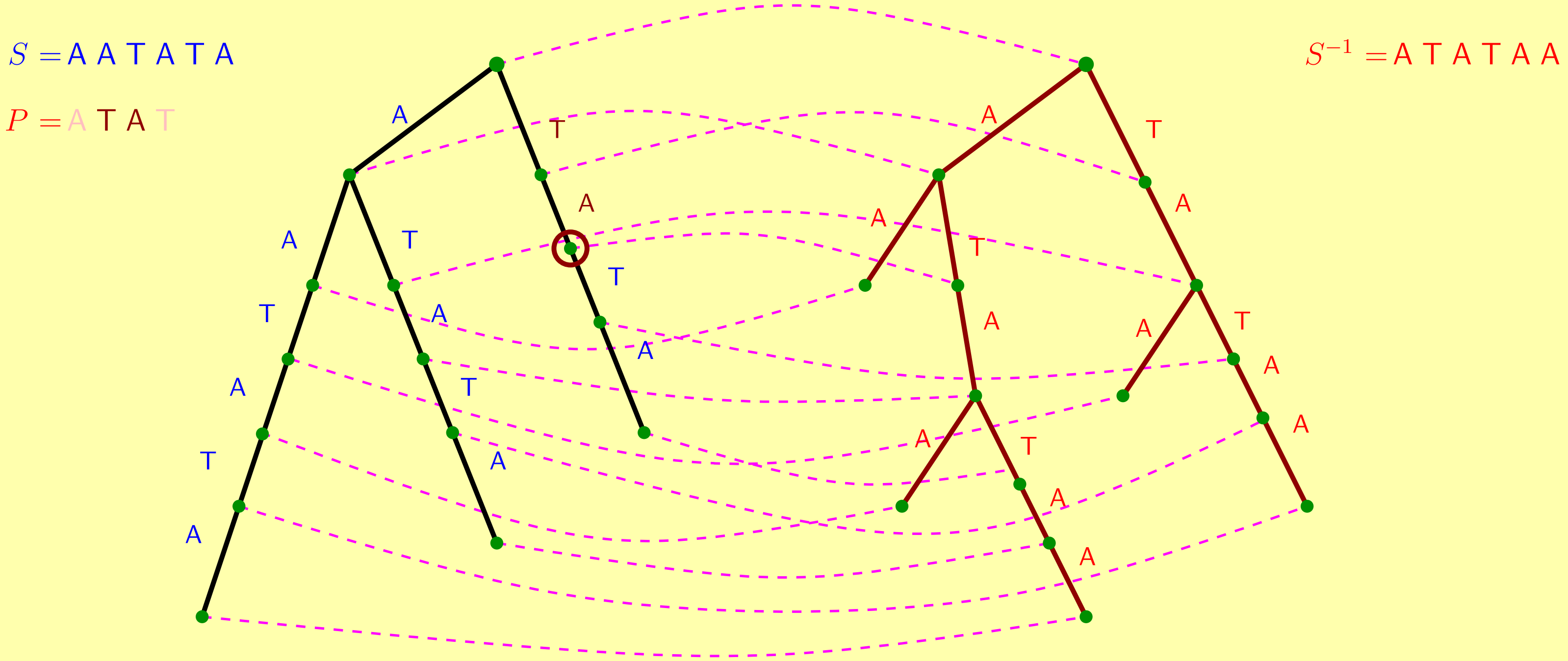
Towards a bi-directional data structure (2)

- It would be nice to have all "prefix edges" inside the same data structure.
- Idea: Create the *atomic* suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



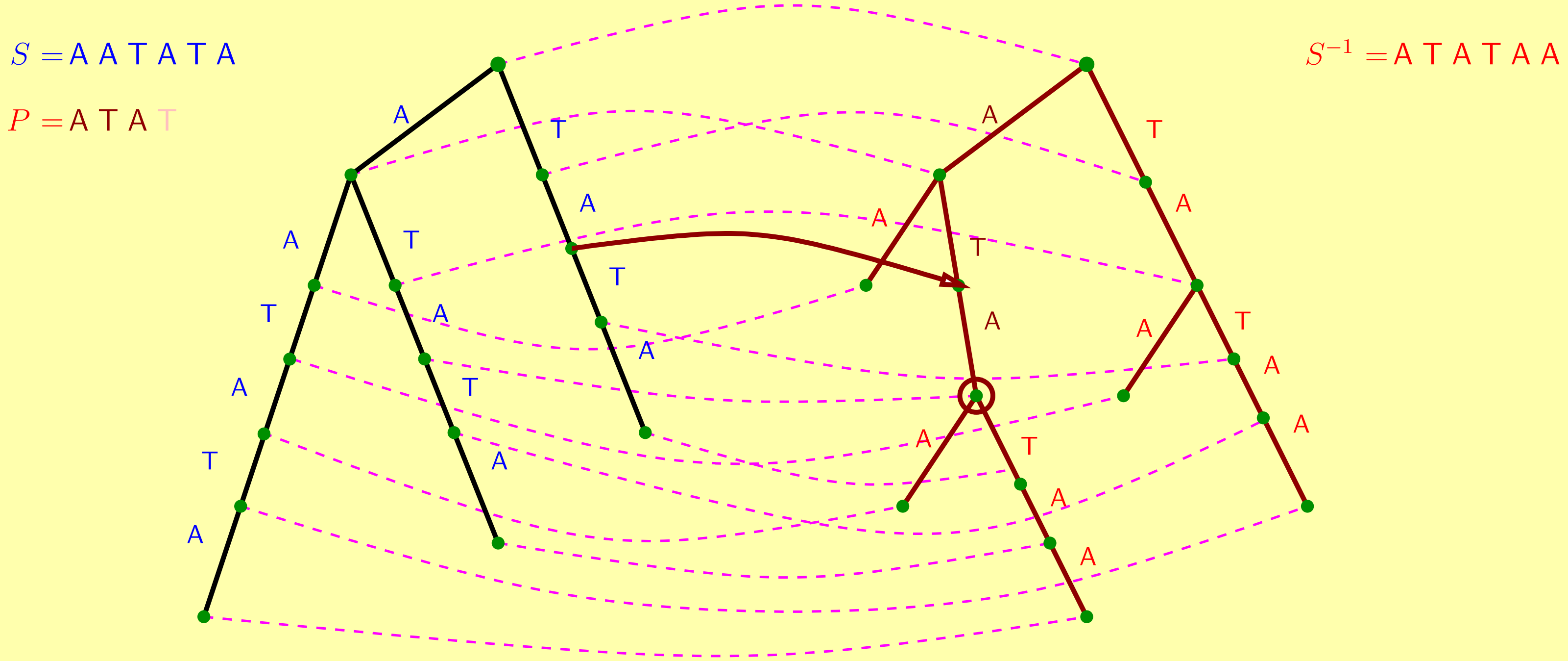
Towards a bi-directional data structure (2)

- It would be nice to have all "prefix edges" inside the same data structure.
- Idea: Create the *atomic* suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



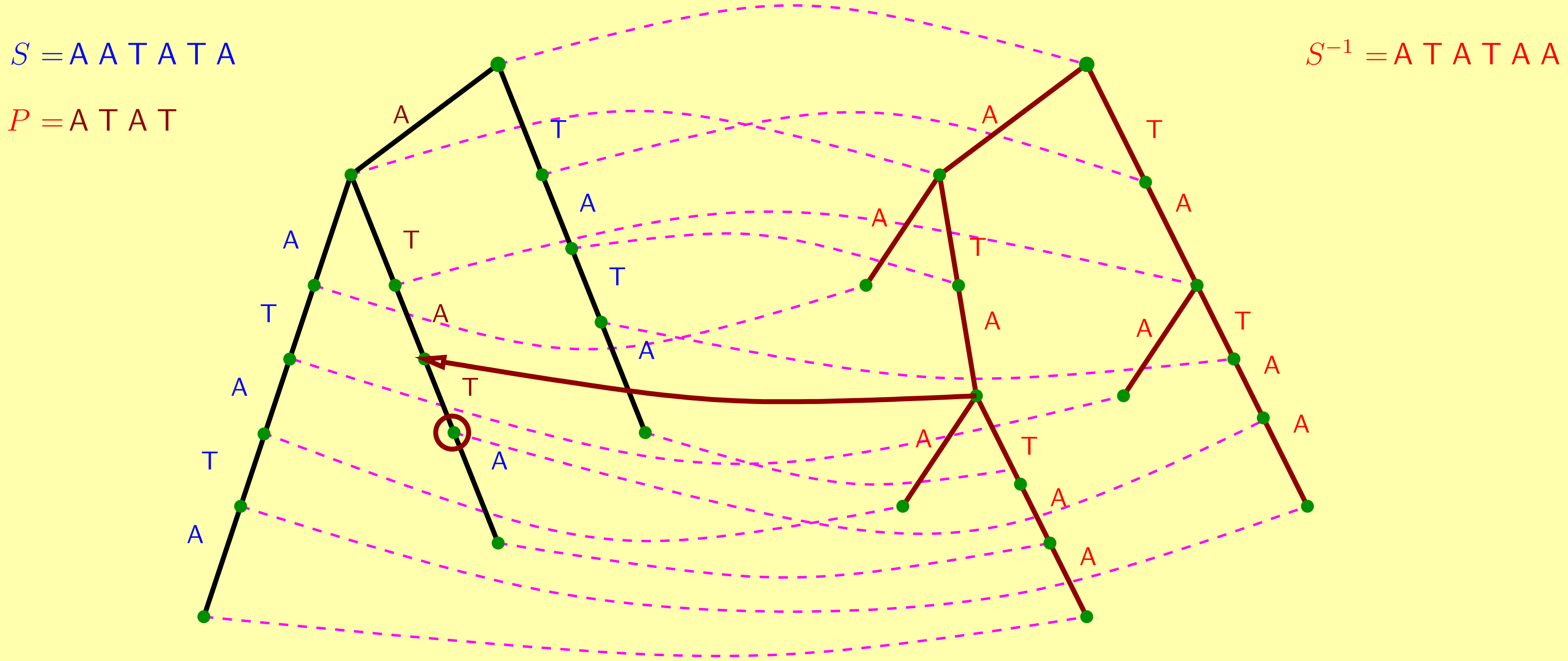
Towards a bi-directional data structure (2)

- It would be nice to have all “prefix edges” inside the same data structure.
- Idea: Create the *atomic* suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



Towards a bi-directional data structure (2)

- It would be nice to have all "prefix edges" inside the same data structure.
- Idea: Create the *atomic* suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



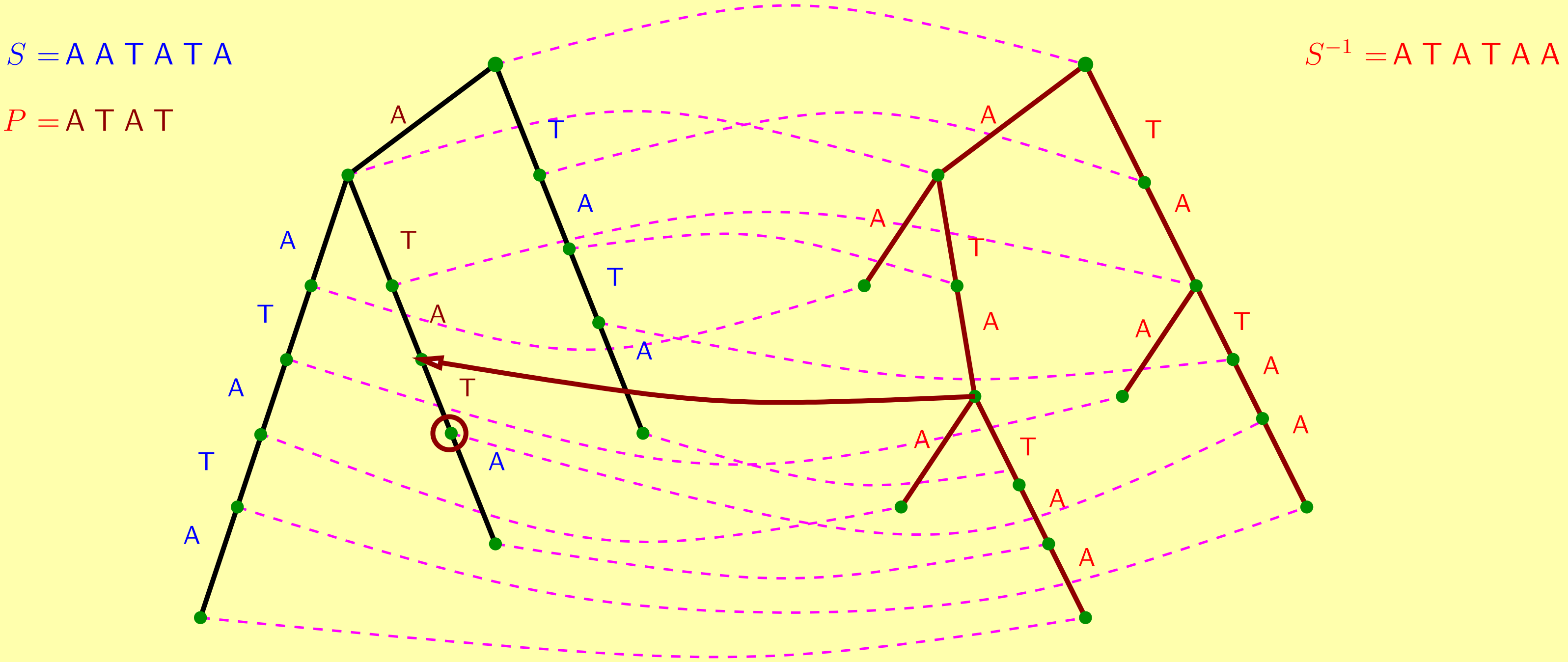
$S =$ A A T A T A

$P =$ A T A T

$S^{-1} =$ A T A T A A

Towards a bi-directional data structure (2)

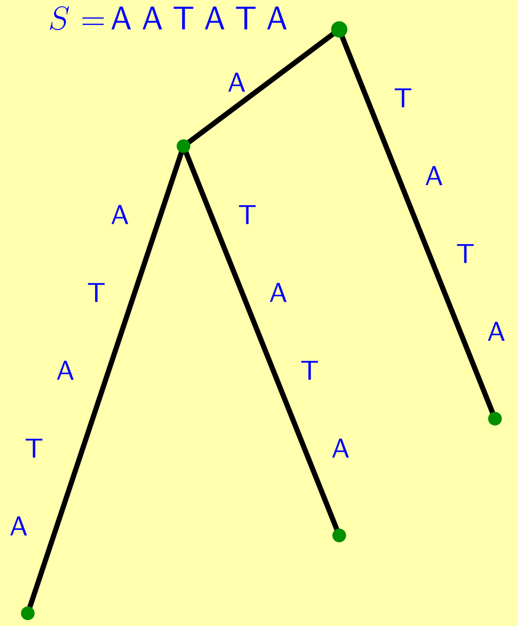
- It would be nice to have all “prefix edges” inside the same data structure.
- Idea: Create the *atomic* suffix tree and reverse prefix tree.
- Create bi-directional links between corresponding nodes.



Problem: quadratic space

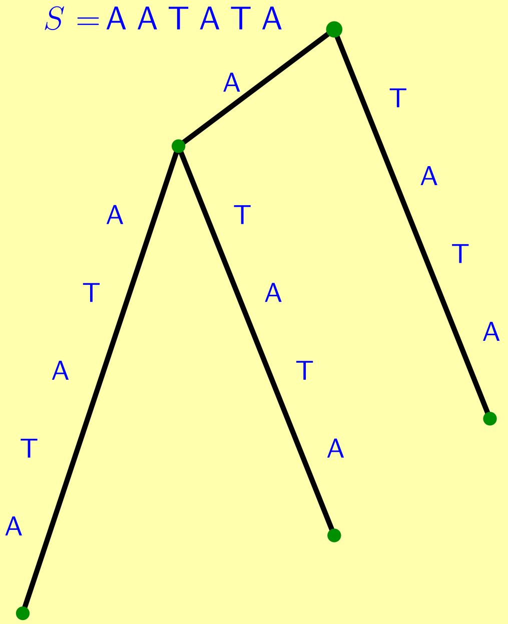
The affix tree

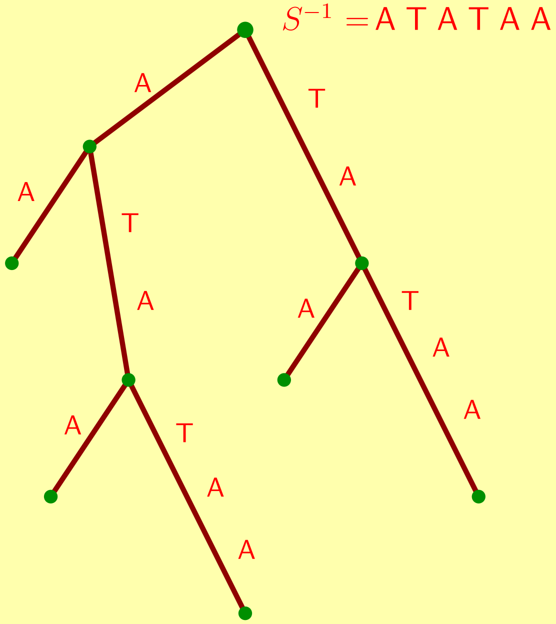
- To save space apply same idea to *compact* suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the affix tree of S.



The affix tree

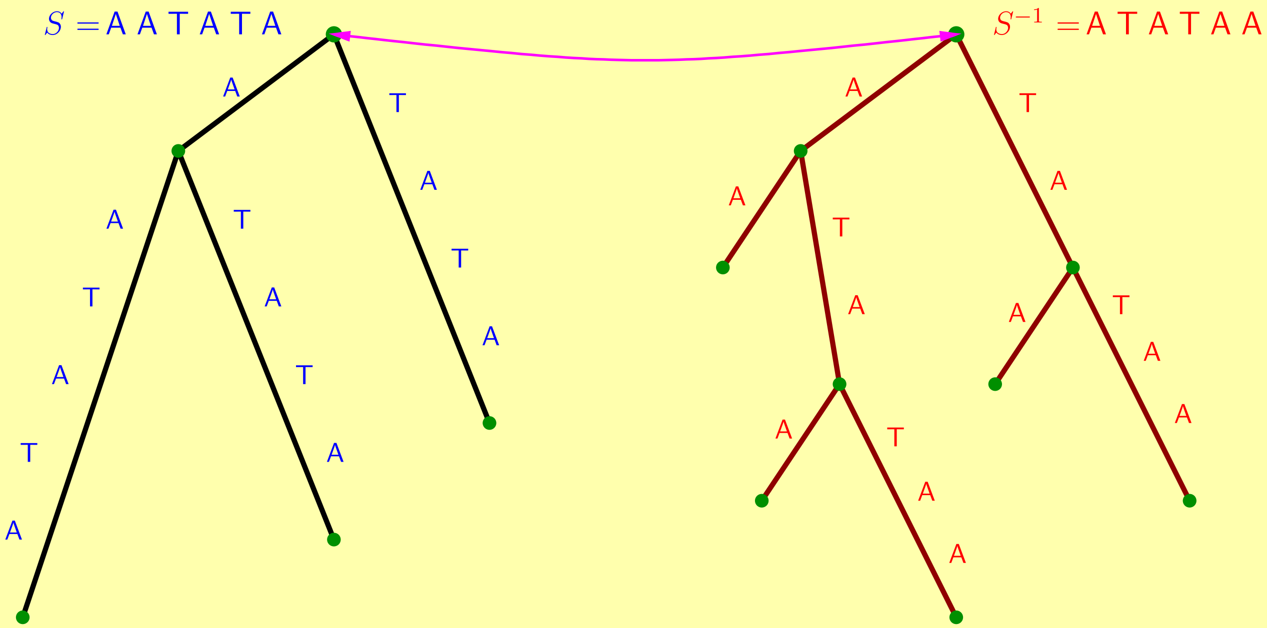
- To save space apply same idea to *compact* suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing $\Rightarrow$ create the missing nodes.
- The joined data structure is called the affix tree of S.





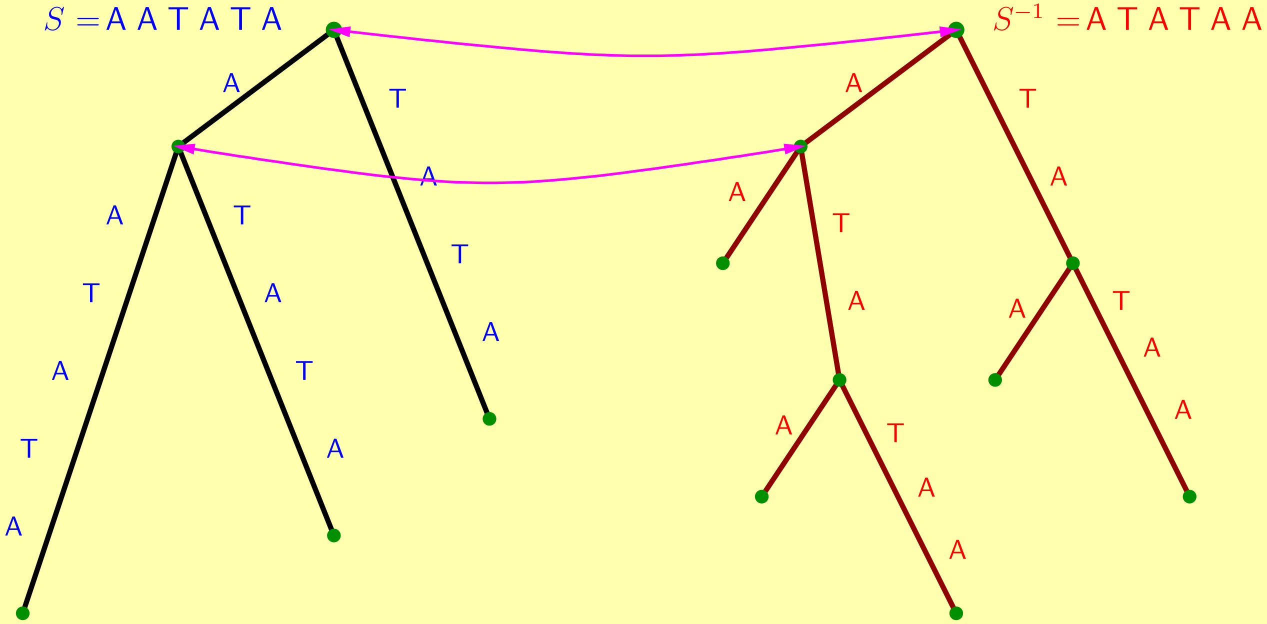
The affix tree

- 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.



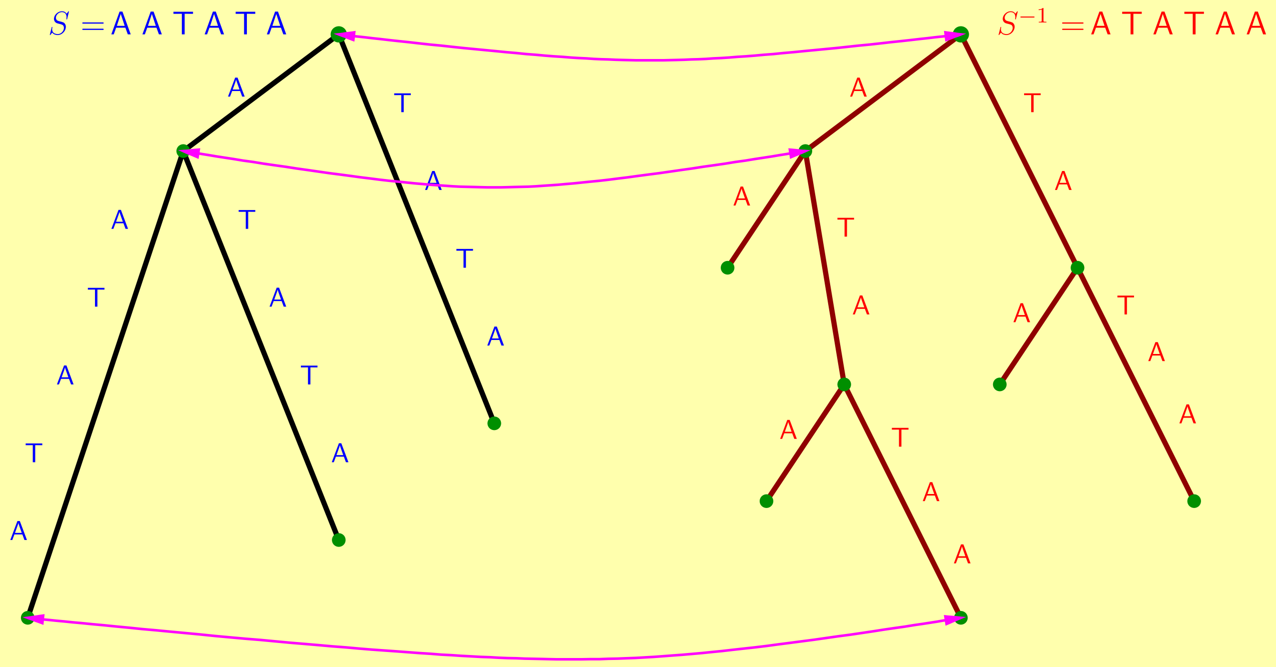
The affix tree

- 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.



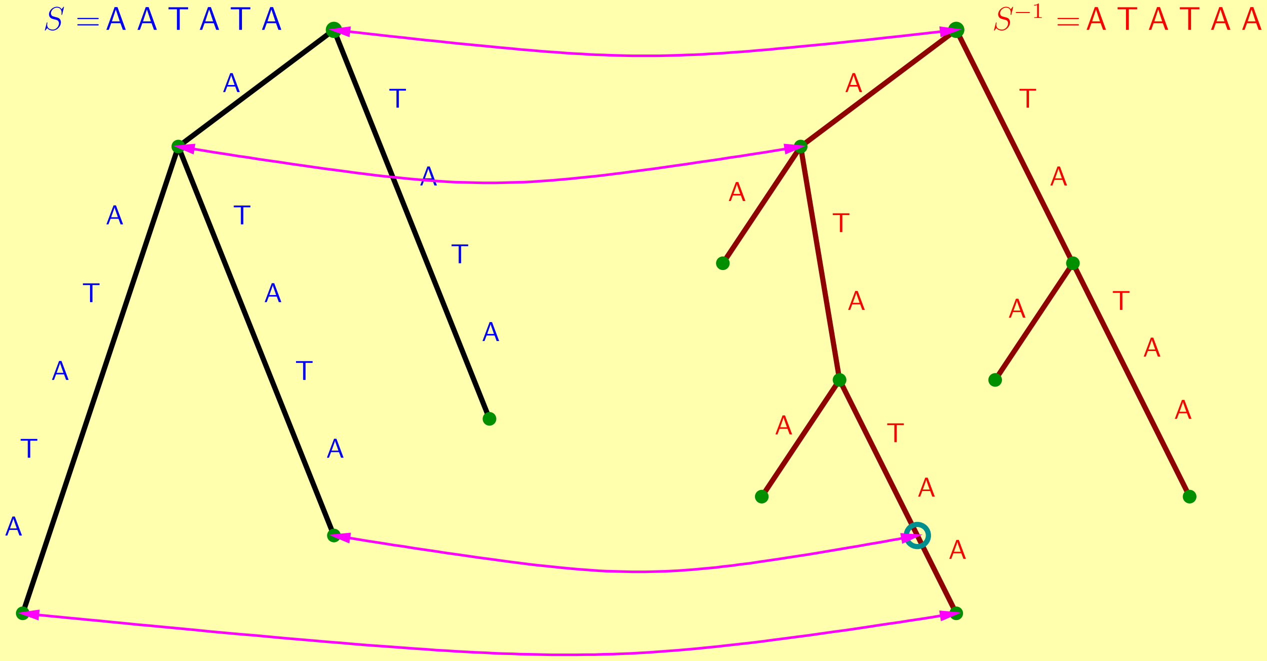
The affix tree

- 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.



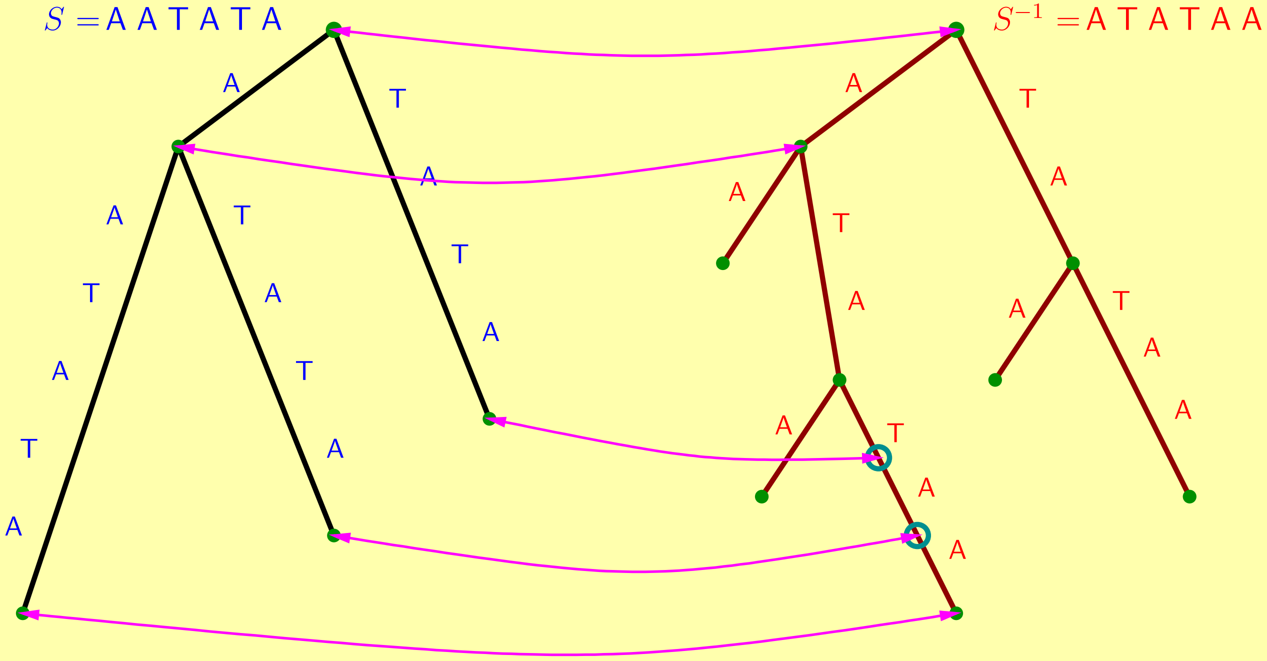
The affix tree

- 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.



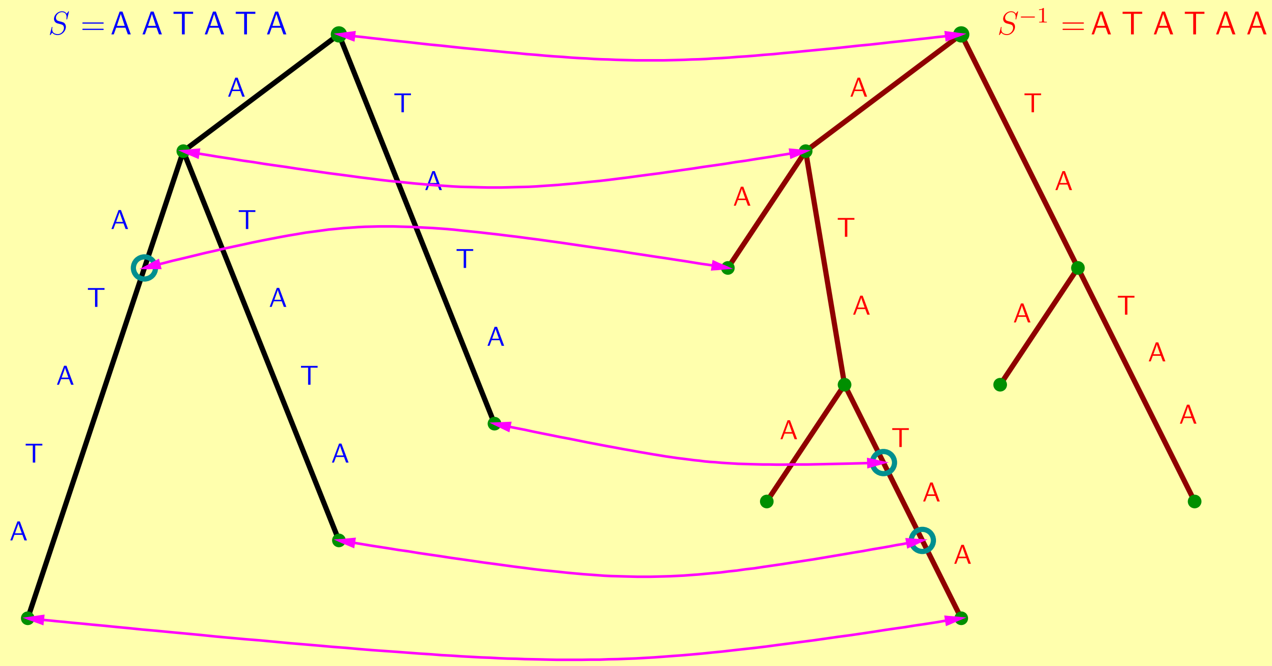
The affix tree

- 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.



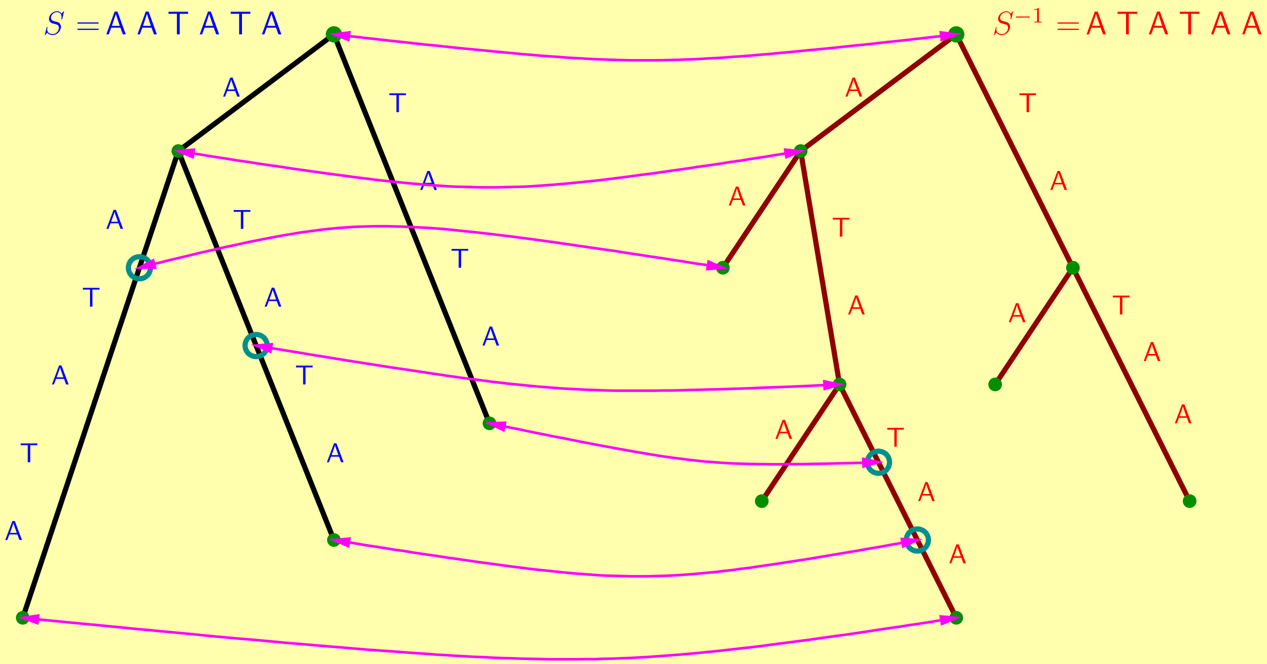
The affix tree

- 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.



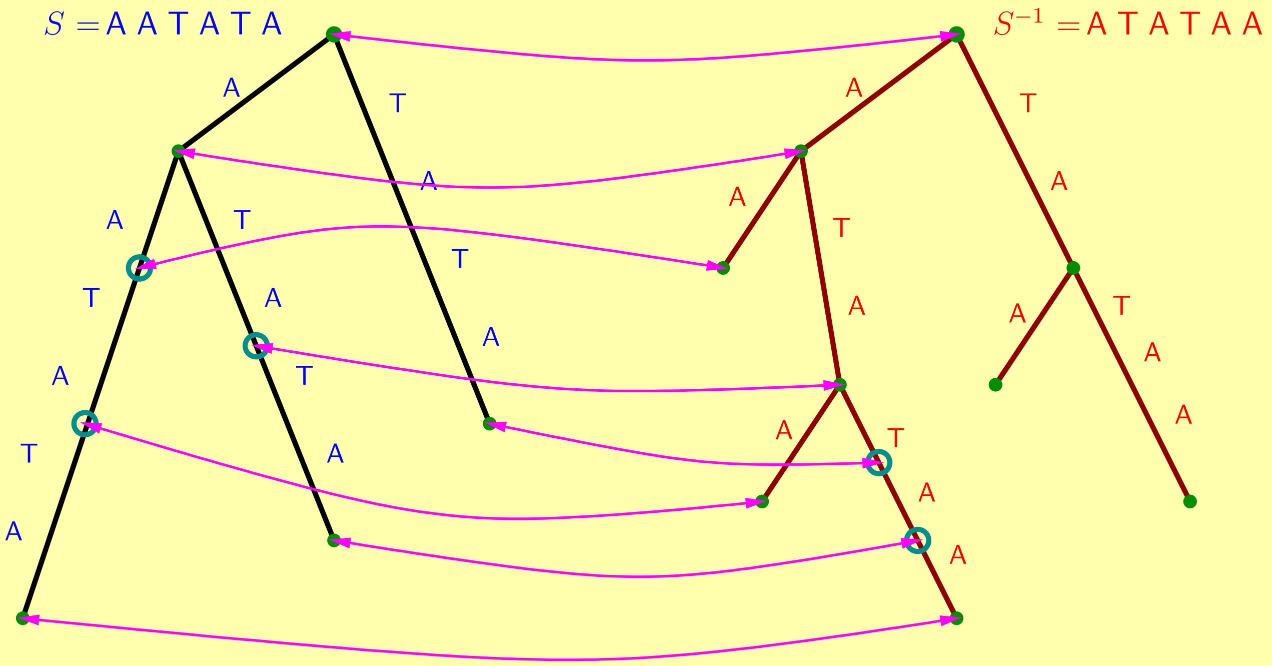
The affix tree

- 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.



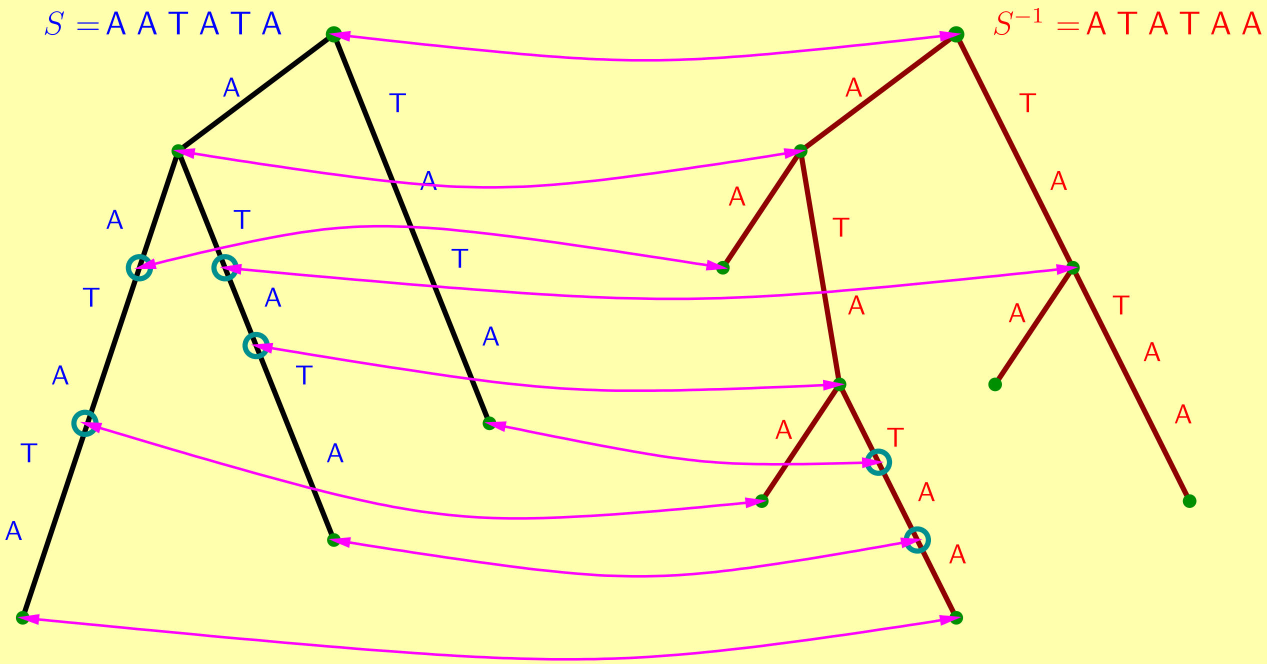
The affix tree

- 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.



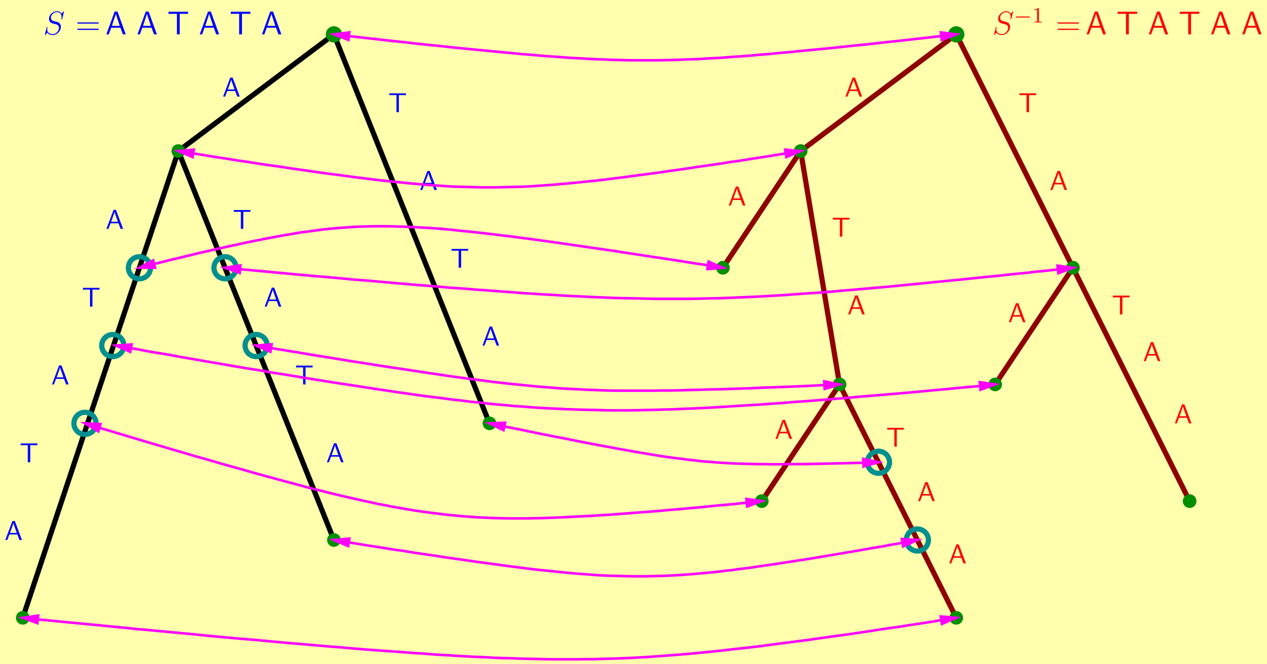
The affix tree

- 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.



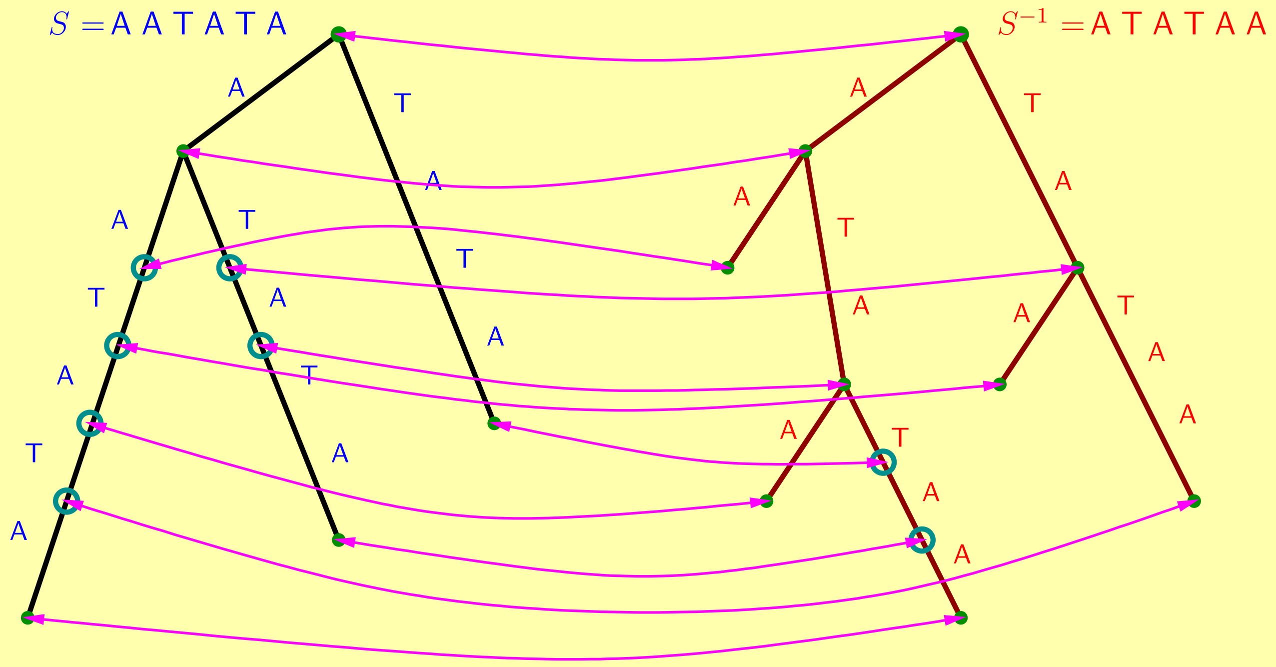
The affix tree

- 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.



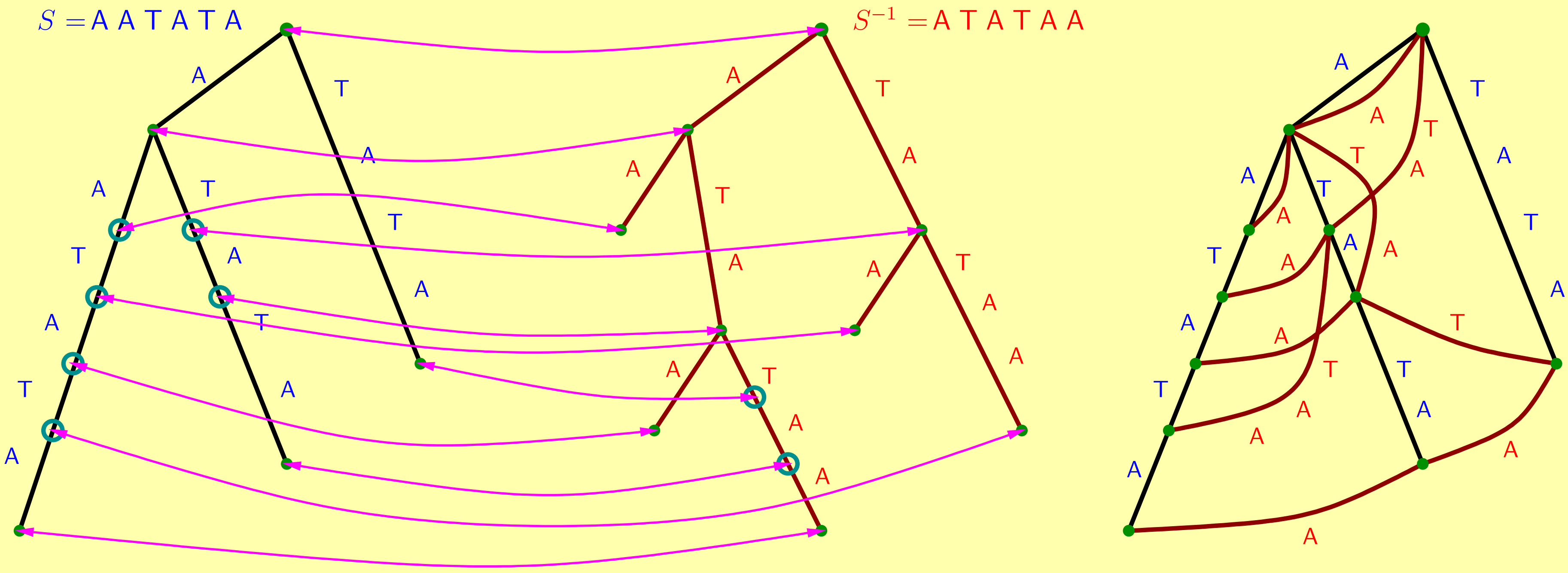
The affix tree

- 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

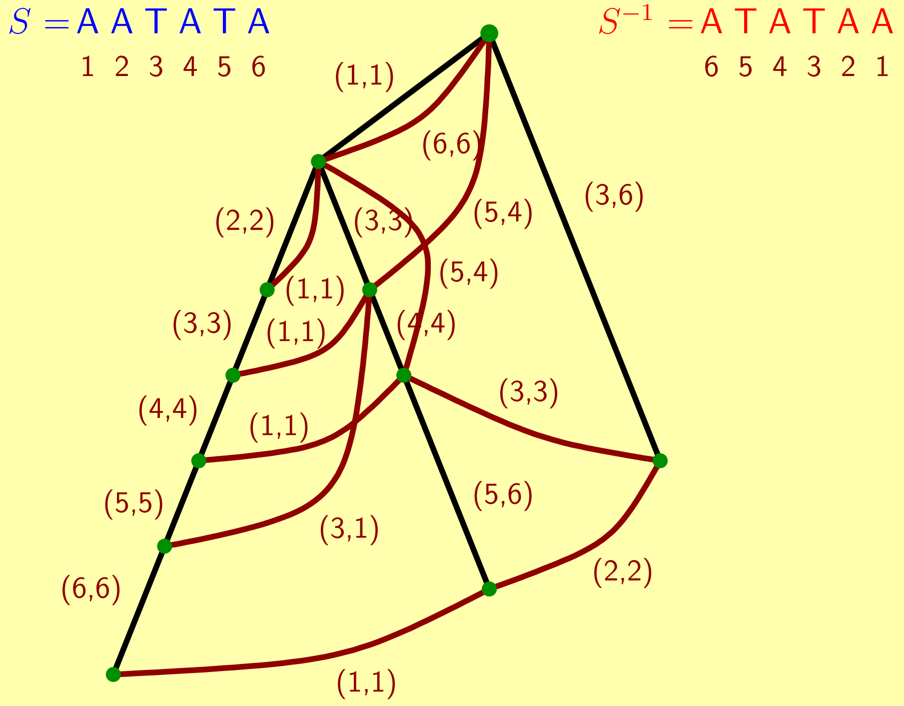
- To save space apply same idea to *compact* suffix tree and reverse prefix tree.
- Problem: Corresponding node might be missing ⇒ create the missing nodes.
- The joined data structure is called the affix tree of S.



Space usage of affix trees

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



Construction of affix trees

Possible construction methods:

- By reverse union of suffix and prefix tree:
  no algorithm known faster than $\mathcal{O}(n \log n)$ time.
- Bidirectional online algorithm: $\mathcal{O}(n)$ time and space (Maaß, CPM 2000).

Idea of bi-directional online construction:

- read string in any direction (also inside-out possible)
- for each character update suffix tree part and prefix tree part
- one-step algorithm: extension of Ukkonen's algorithm for suffix tree

→ track active suffix and active prefix
→ at most one node can be deleted in each step

Overview

- Introduction: Pattern matching in biosequence analysis
- A flexible index structure for sequence analysis: The suffix tree
  - Basic definitions
  - Application 1: Repeats, tandem repeats
  - Application 2: Multiple Genome Aligner (MGA)
- An even more flexible data structure: The affix tree
  - Basic definitions
  - Application: Search for palindromic patterns
- Conclusion

Application: Search for palindromic patterns

Define an RNA hairpin by the following “mirror pattern” (in HyPaL syntax):

```
stem = .4
hairpin = @stem CGCG reverse(@stem)
```

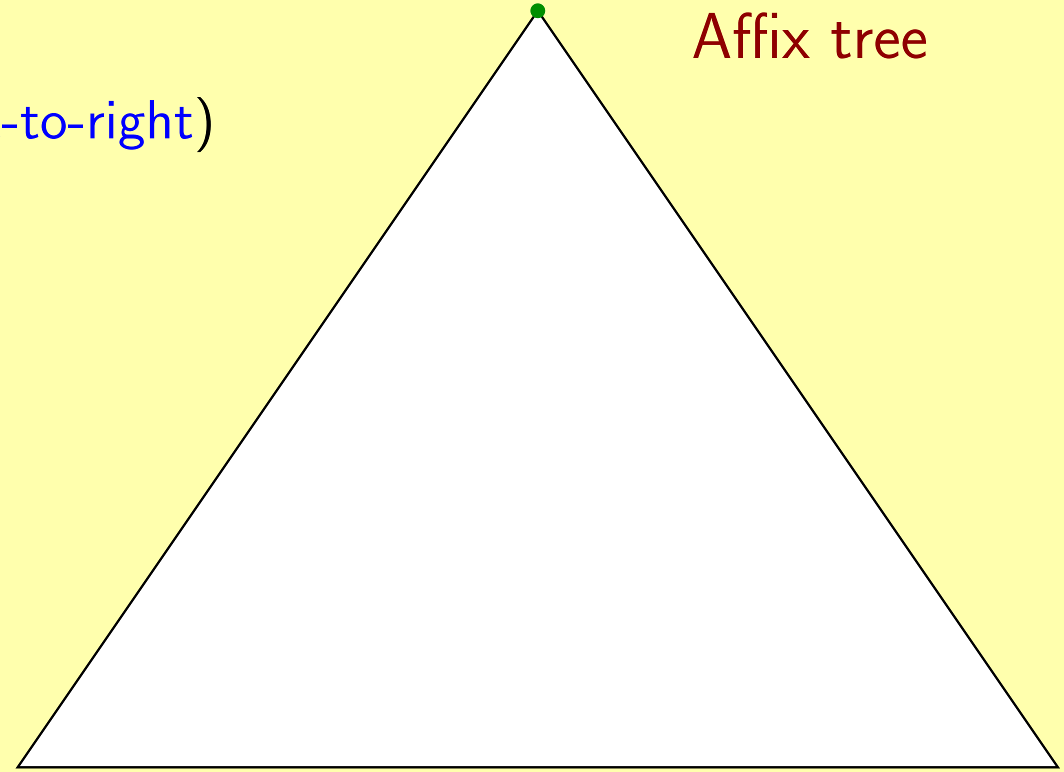
Application: Search for palindromic patterns

Define an RNA hairpin by the following "mirror pattern" (in HyPaL syntax):

```
stem = .4
hairpin = @stem CGCG reverse(@stem)
```

Search strategy:

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



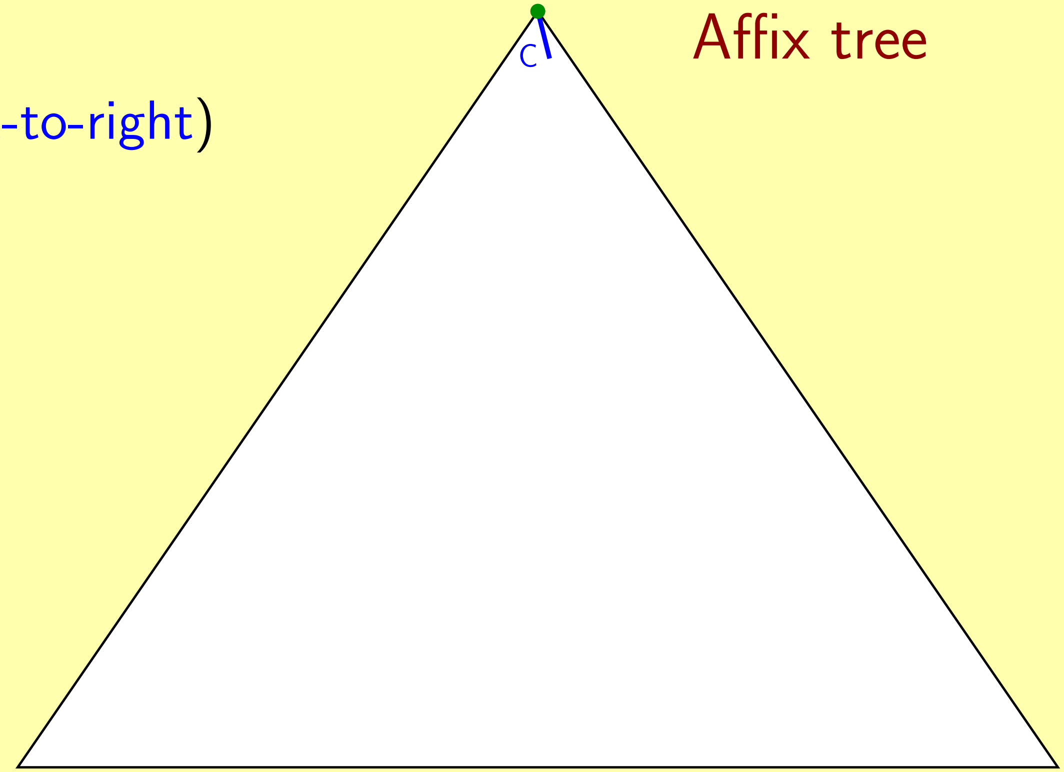
Application: Search for palindromic patterns

Define an RNA hairpin by the following “mirror pattern” (in HyPaL syntax):

```
stem = .4
hairpin = @stem CGCG reverse(@stem)
```

Search strategy:

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



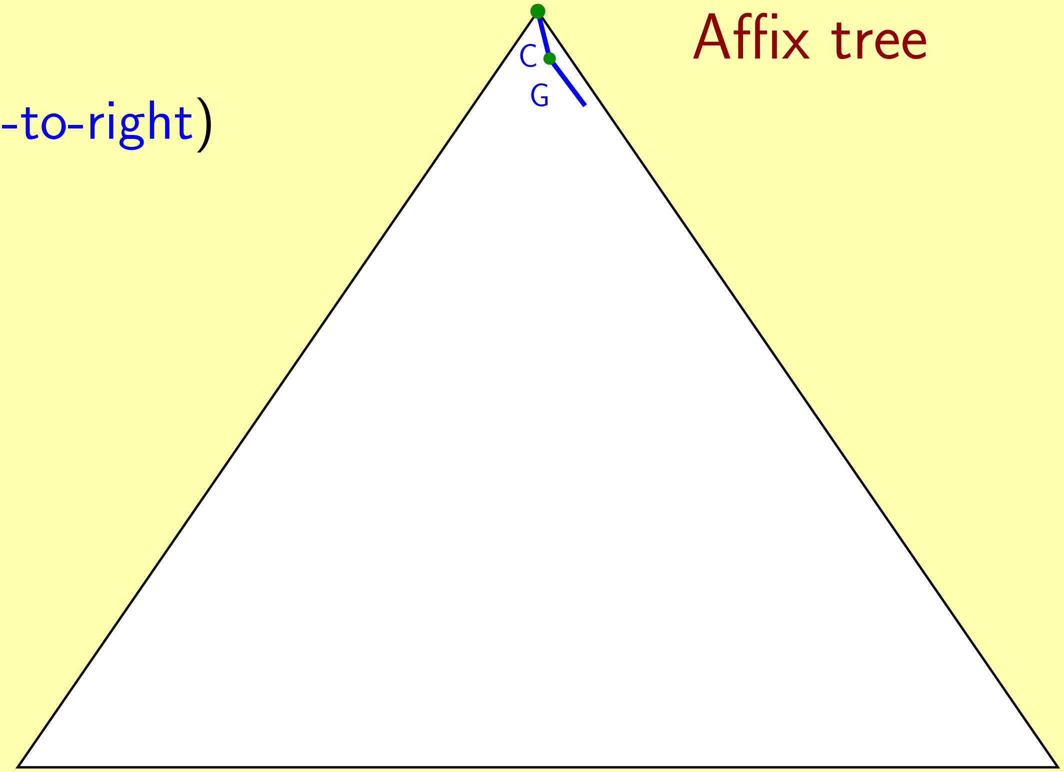
Application: Search for palindromic patterns

Define an RNA hairpin by the following “mirror pattern” (in HyPaL syntax):

```
stem = .4
hairpin = @stem CGCG reverse(@stem)
```

Search strategy:

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



Affix tree

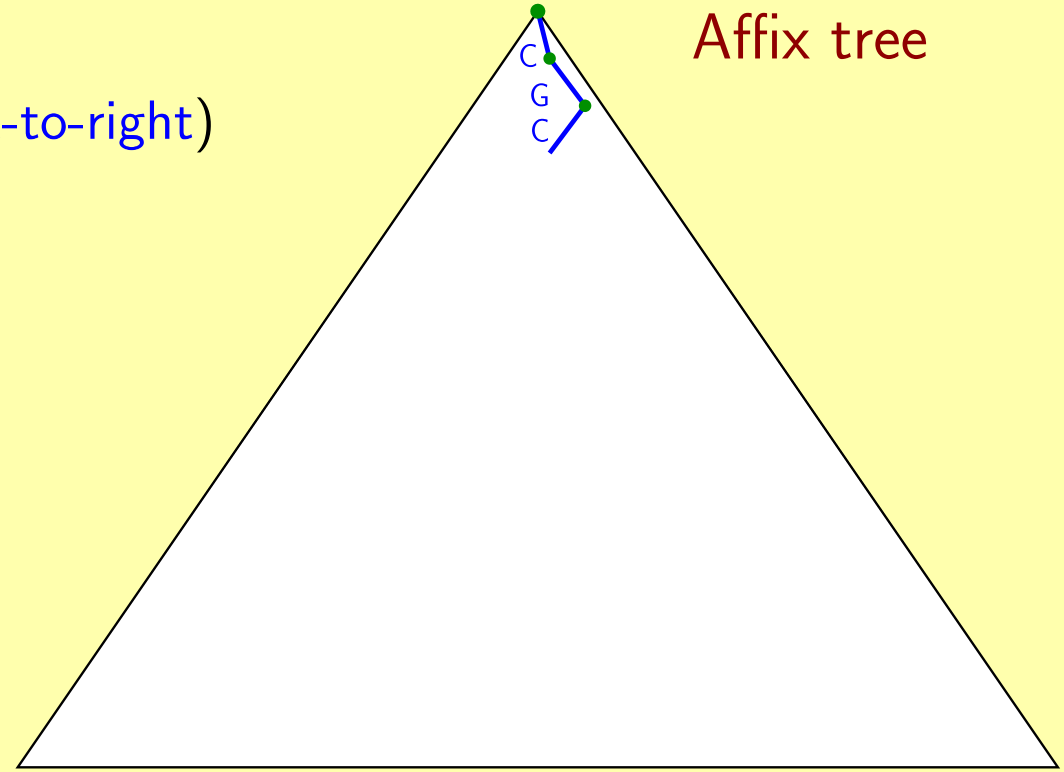
Application: Search for palindromic patterns

Define an RNA hairpin by the following “mirror pattern” (in HyPaL syntax):

```
stem = .4
hairpin = @stem CGCG reverse(@stem)
```

Search strategy:

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



Affix tree

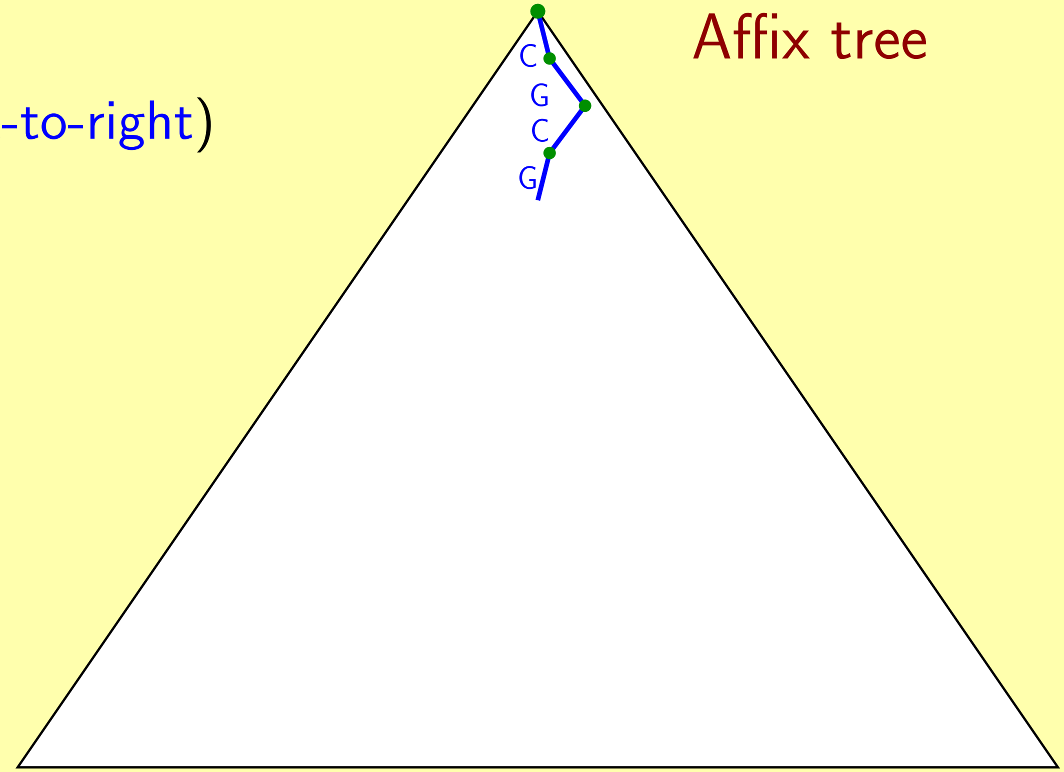
Application: Search for palindromic patterns

Define an RNA hairpin by the following "mirror pattern" (in HyPaL syntax):

```
stem = .4
hairpin = @stem CGCG reverse(@stem)
```

Search strategy:

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



Affix tree

Application: Search for palindromic patterns

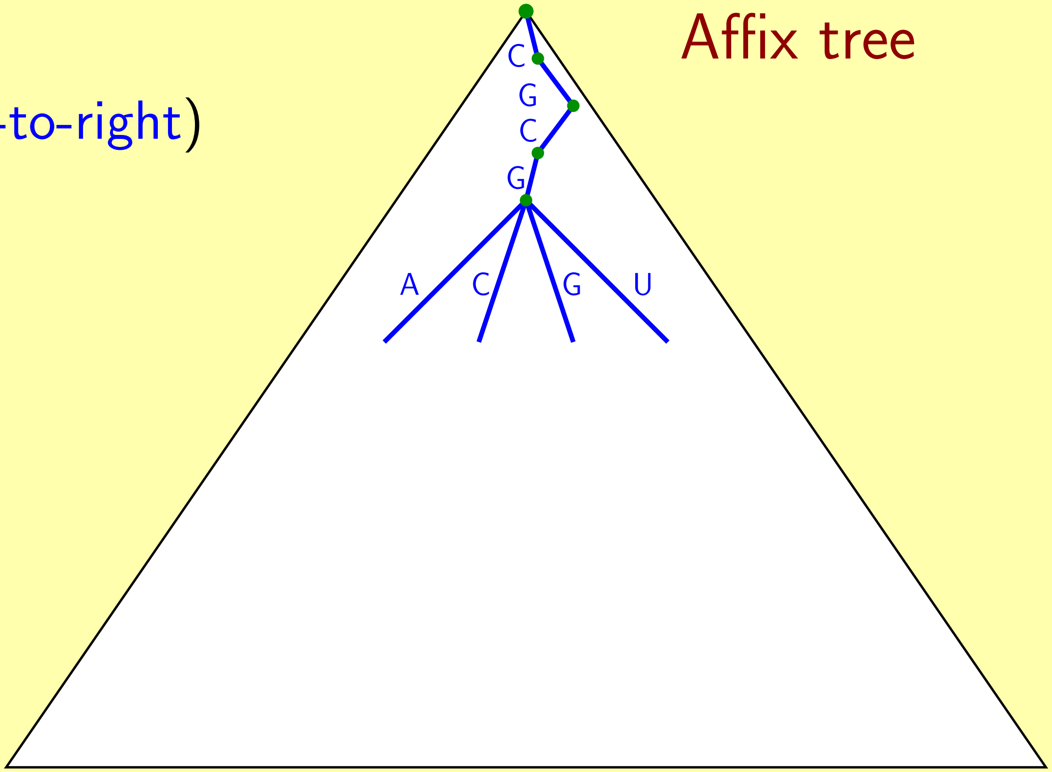
Define an RNA hairpin by the following “mirror pattern” (in HyPaL syntax):

```
stem = .4
hairpin = @stem CGCG reverse(@stem)
```

Search strategy:

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

2. Extend by
    - an arbitrary character to the right



Application: Search for palindromic patterns

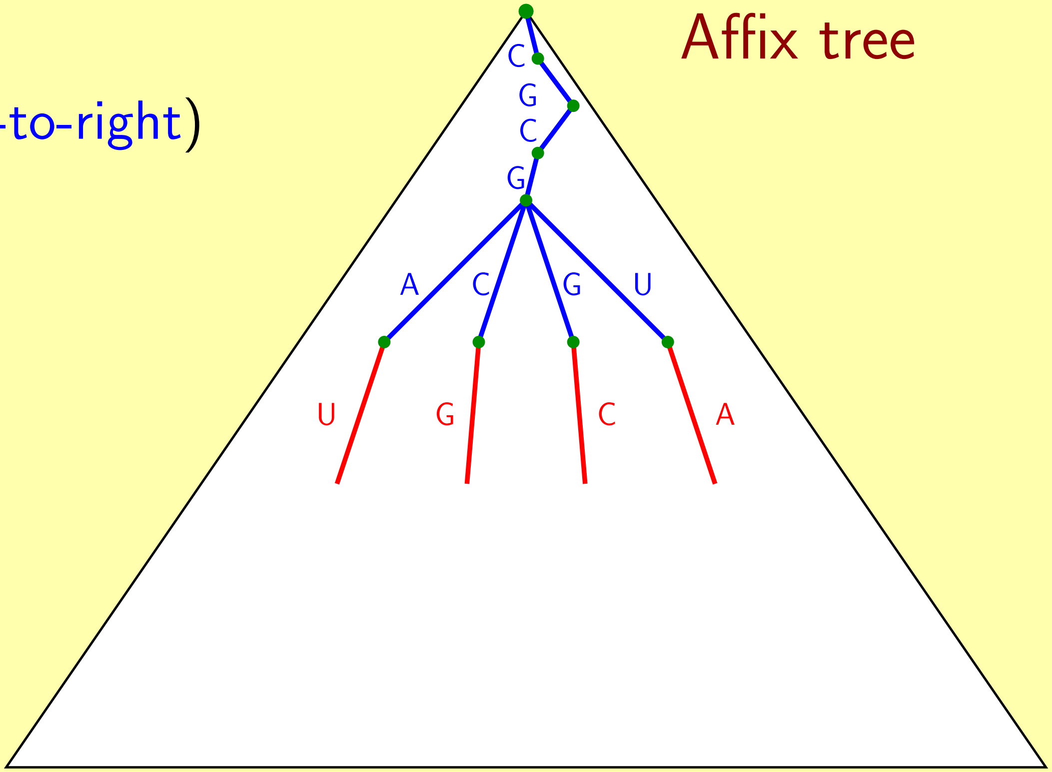
Define an RNA hairpin by the following "mirror pattern" (in HyPaL syntax):

```
stem = .4
hairpin = @stem CGCG reverse(@stem)
```

Search strategy:

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

2. Extend by
   - an arbitrary character to the right
   - *the same* character to the left



Application: Search for palindromic patterns

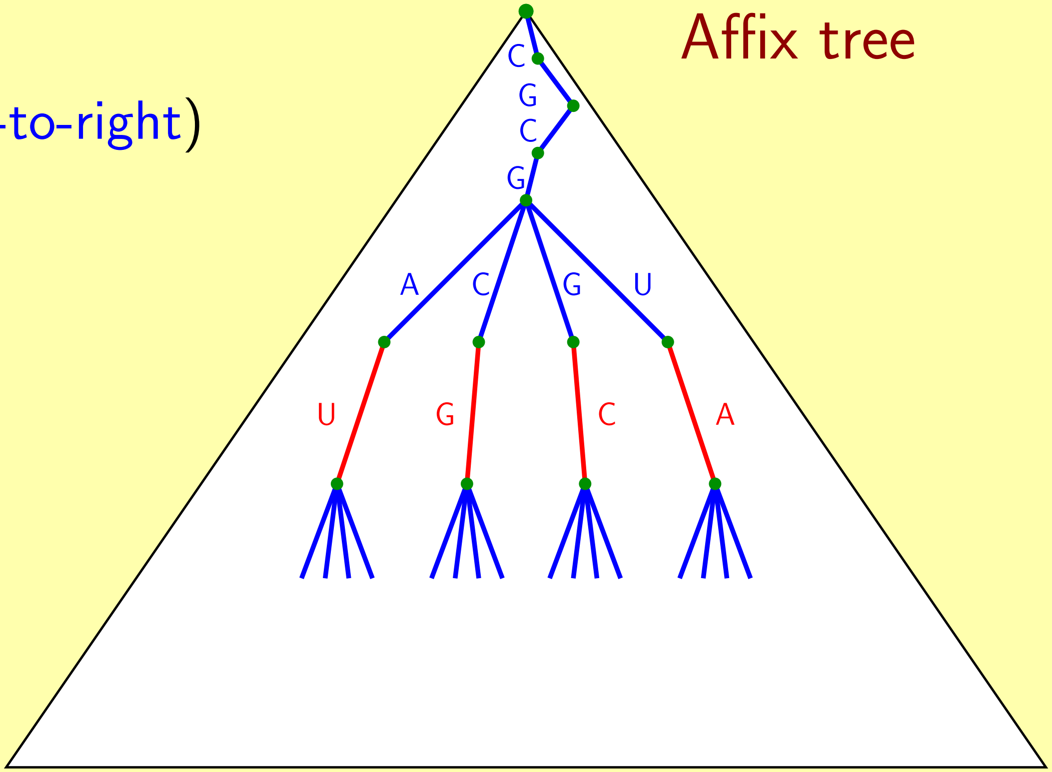
Define an RNA hairpin by the following “mirror pattern” (in HyPaL syntax):

```
stem = .4
hairpin = @stem CGCG reverse(@stem)
```

Search strategy:

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

2. Extend by
   - an arbitrary character to the right
   - *the same* character to the left
   - an arbitrary character to the right



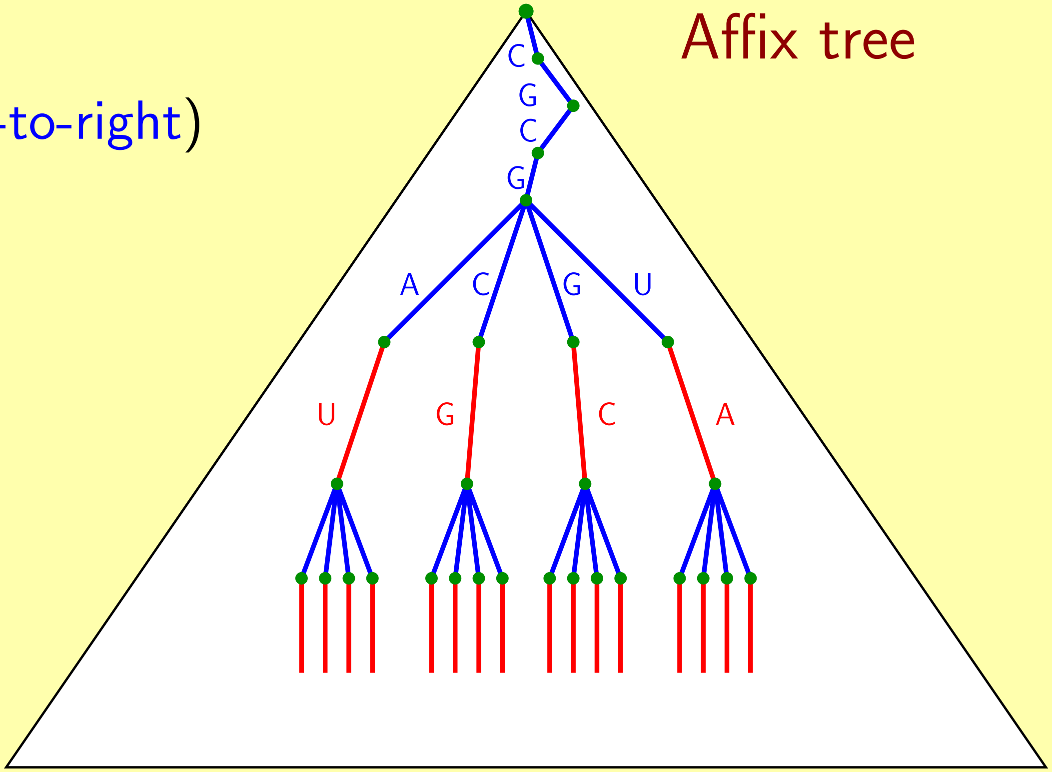
Application: Search for palindromic patterns

Define an RNA hairpin by the following "mirror pattern" (in HyPaL syntax):

```
stem = .4
hairpin = @stem CGCG reverse(@stem)
```

Search strategy:

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



Application: Search for palindromic patterns

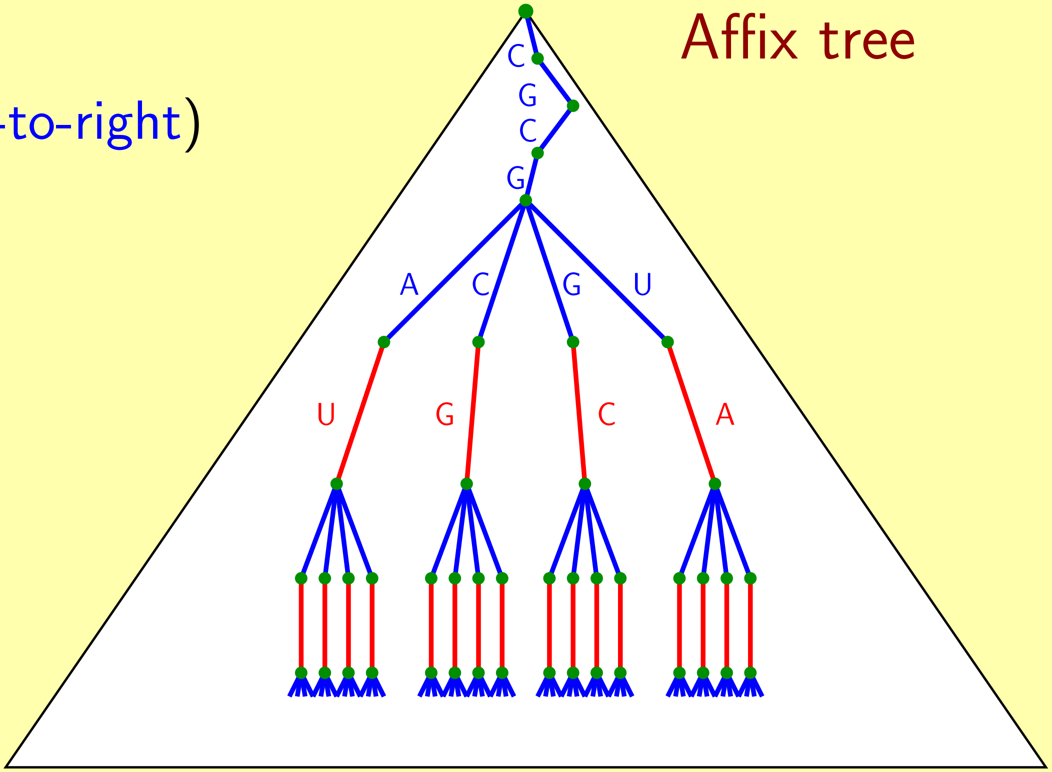
Define an RNA hairpin by the following "mirror pattern" (in HyPaL syntax):

```
stem = .4
hairpin = @stem CGCG reverse(@stem)
```

Search strategy:

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

2. Extend by
    - an arbitrary character to the right
    - *the same* character to the left
    - an arbitrary character to the right
    - *the same* character to the left
    - ...



Overview

- Introduction: Pattern matching in biosequence analysis
- A flexible index structure for sequence analysis: The suffix tree
  - Basic definitions
  - Application 1: Repeats, tandem repeats
  - Application 2: Multiple Genome Aligner (MGA)
- An even more flexible data structure: The affix tree
  - Basic definitions
  - Application: Search for palindromic patterns
- Conclusion

Conclusion

The analysis of biological sequence data produces several interesting computational questions. Several CS disciplines involved:

- Algorithms and data structures
- Algorithm engineering
- Software engineering
- Visualization

Not only does Biology profit from Computer Science, but also vice versa!

Acknowledgments

Suffix trees

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

Repeats

- Dan Gusfield (UC Davis)

Affix trees

- Dirk Strothmann (U Bielefeld)

Black slide