Practical Methods for Constructing Suffix Trees

Advisor: Jignesh M. Patel

Yuanyuan Tian

Roadmap

- **Background & Motivation**
- Our Approaches
  - Top-Down Disk-Based Algorithm (TDD)
  - Merge-Based Algorithm (ST-Merge)
- Experiments and Analysis

Background

- Sequence data sets are ubiquitous in modern life science and traditional text applications, and querying sequences is a common and critical operation in these applications.
- Suffix trees are versatile data structures that can help execute a variety of sequence matching queries efficiently.

  e.g. pattern matching, biological sequence alignment (OASIS, MUMer, REPuter, etc.)

Suffix Tree

Suffixes:

S0: ATTAGTACA$

S1: TTAGTACA$

S2: TAGTACA$

S3: AGTACA$

S4: GTACA$

S5: TACA$

S6: ACA$

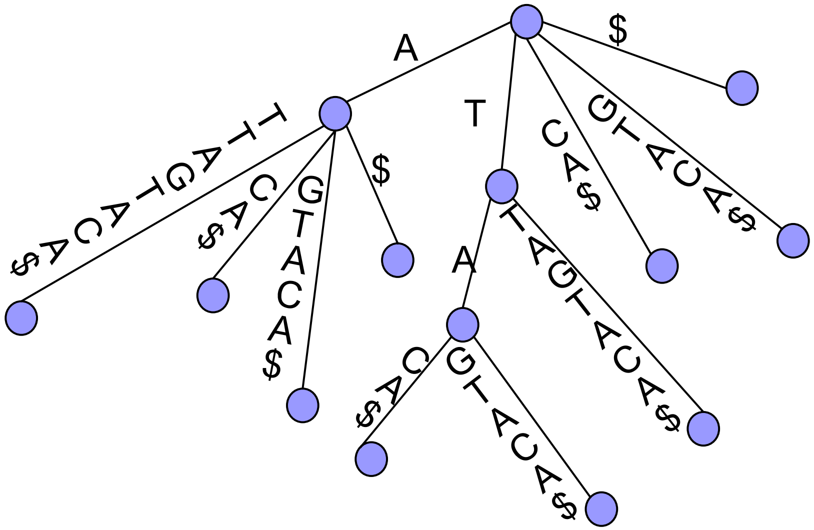
S7: CA$

S8: A$

S9: $

String: ATTAGTACA$

0 1 2 3 4 5 6 7 8 9

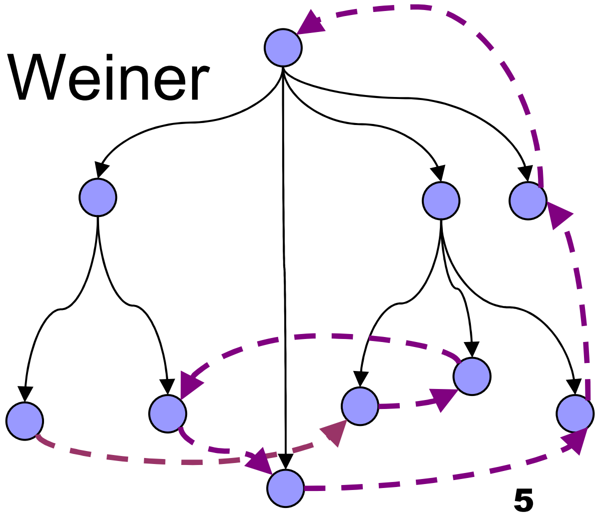


Storage requirement of a suffix tree is *O(n)*.

Existing Algorithms

In-Memory Algorithms

- Quadratic algorithms
  - Brute-force -- Insert a suffix at a time
  - WOTD -- good locality of reference
  - Worst case $O(n^2)$, average $O(nlogn)$
- Linear time algorithms
  - Best-known -- Ukkonen, McCreight, Weiner
  - Use suffix links
  - Poor locality of memory reference
  - Only suited for small input size

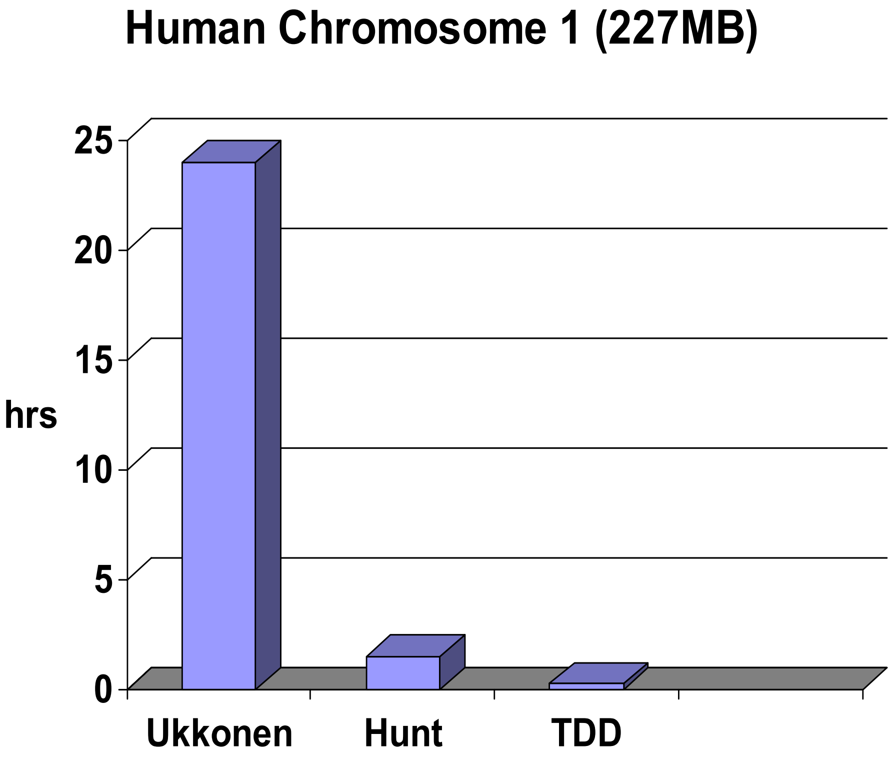


Existing Algorithms

On-Disk Algorithms

- Practical Algorithms
  - TOP-Q
    - Buffering strategy to improve the performance of Ukkonen's algorithm
  - Hunt's algorithm -- the best known practical disk-based algorithm
    - Construct different parts of the suffix tree independently
    - Use Brute-Force -- random access to the tree
- Theoretical Algorithms
  - Reduce complexity to sorting, but tricky to implement
  - Build a suffix array from which a suffix tree is built
    - SKEW -- linear suffix array construction algorithm

Existing Algorithms



>1 day ~1.5hrs ~17mins

Motivation

- Many sequence datasets are growing at exponential rates. (e.g. GenBank x2/16mo.)
- Existing techniques can only deal with **small trees**, for big datasets they are **impractical.**

Roadmap

- Background & Motivation
- **Our Approaches**
  - Top-Down Disk-Based Algorithm (TDD)
  - Merge-Based Algorithm (ST-Merge)
- Experiments and Analysis

Suffix Tree Representation

String: ATTAGTACA$

0 1 2 3 4 5 6 7 8 9

Alphabet Order: A<T<C<G<$

Suffixes:

S0: ATTAGTACA$

S1: TTAGTACA$

S2: TAGTACA$

S3: AGTACA$

S4: GTACA$

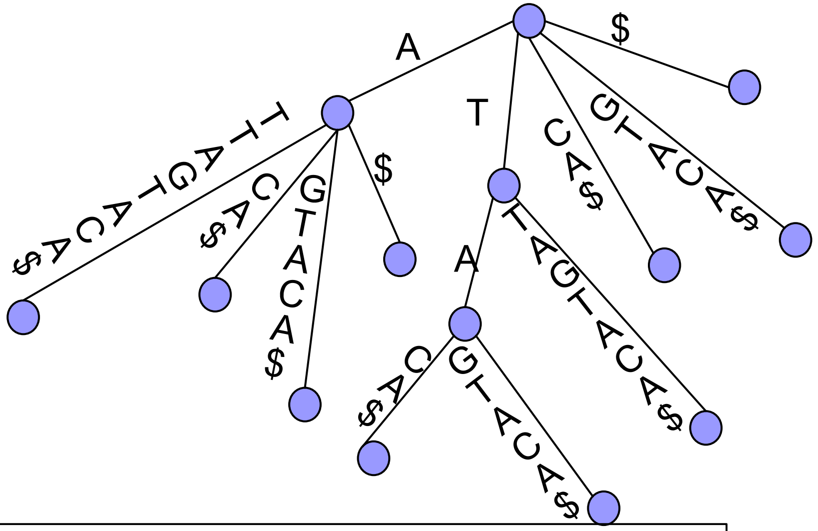
S5: TACA$

S6: ACA$

S7: CA$

S8: A$

S9: $



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	12	1	7	7	4	9_R	3	10	2_R	7	4_R	1	7	4	9_R
A		T		CA$	GTACA$	$	A		TAGTACA$	CA$	GTACA$	TTAGTACA$	CA$	GTACA$	$

Top-Down Disk-Based Algorithm (TDD)

- Leverage partitioning to make use of main-memory efficiently and reduce disk I/Os
  - Use Hunt's partitioning strategy
- Minimize random references and try to use sequential access
  - Based on WOTD -- good locality behavior concerning tree access
- Manage buffers for large structures so that disk I/Os are reduced

Data Structures

ATTAGTACA$

String

0	12	1	7	7	…

Tree (8.5x)

S0: ATTAGTACA$
S3: AGTACA$
S6: ACA$
S8: A$
S1: TTAGTACA$
S2: TAGTACA$
S5: TACA$
S7: CA$
S4: GTACA$
S9: $

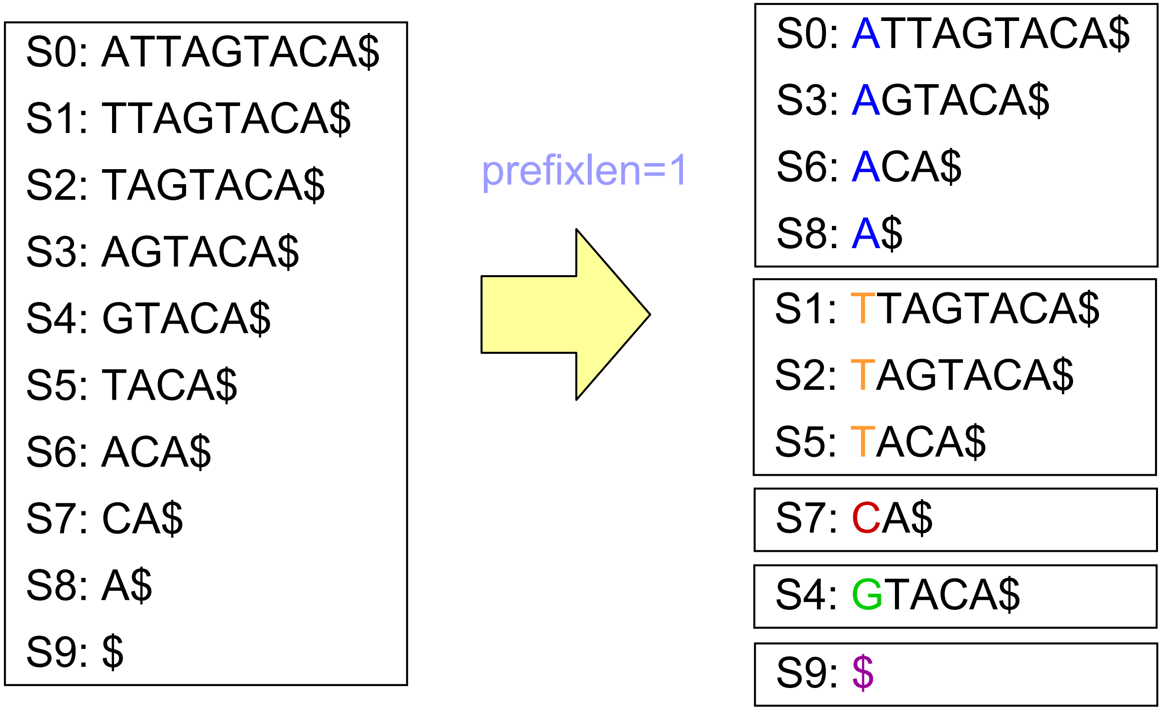
Suffixes (4x)

S0: ATTAGTACA$
S1: TTAGTACA$
S2: TAGTACA$
S3: AGTACA$
S4: GTACA$
S5: TACA$
S6: ACA$
S7: CA$
S8: A$
S9: $

Temp (4x)

TDD Execution

Phase 1: partitioning the suffixes of the input string into $|A|^{prefixlen}$ partitions



S0: ATTAGTACA$
S1: TTAGTACA$
S2: TAGTACA$
S3: AGTACA$
S4: GTACA$
S5: TACA$
S6: ACA$
S7: CA$
S8: A$
S9: $

prefixlen=1

S0: ATTAGTACA$
S3: AGTACA$
S6: ACA$
S8: A$

S1: TTAGTACA$
S2: TAGTACA$
S5: TACA$

S7: CA$

S4: GTACA$

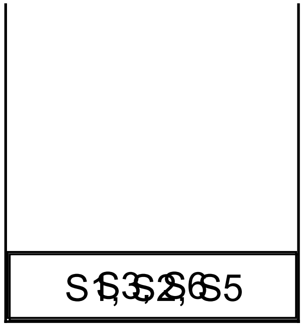
S9: $

Time Complexity $O(n*prefixlen)$

TDD Execution

Phase 2

String: ATTAGTACA$
0 1 2 3 4 5 6 7 8 9



Longest Common Prefix

S1: TTAGTACA$
S2: TAGTACA$
S5: TACA$

LCP=T

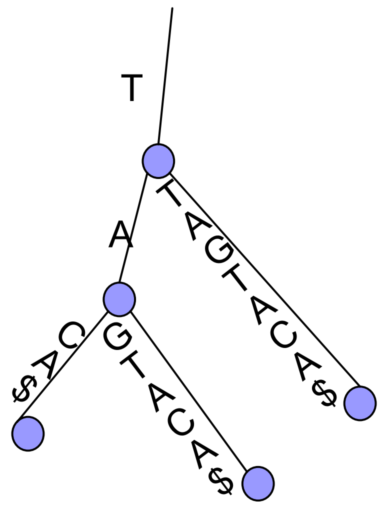
S2: TAGTACA$
S3: AGTACA$
S6: ACA$

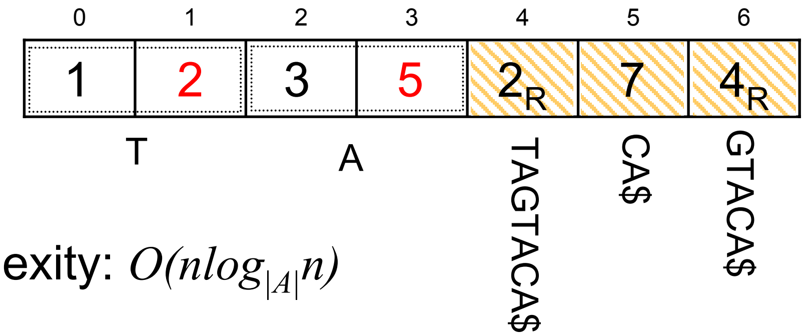
S3: AGTACA$
S6: ACA$
S2: TAGTACA$

LCP=A

S4: GTACA$
S7: CA$

S7: CA$
S4: GTACA$





Average Time Complexity: $O(nlog_{|A|}n)$

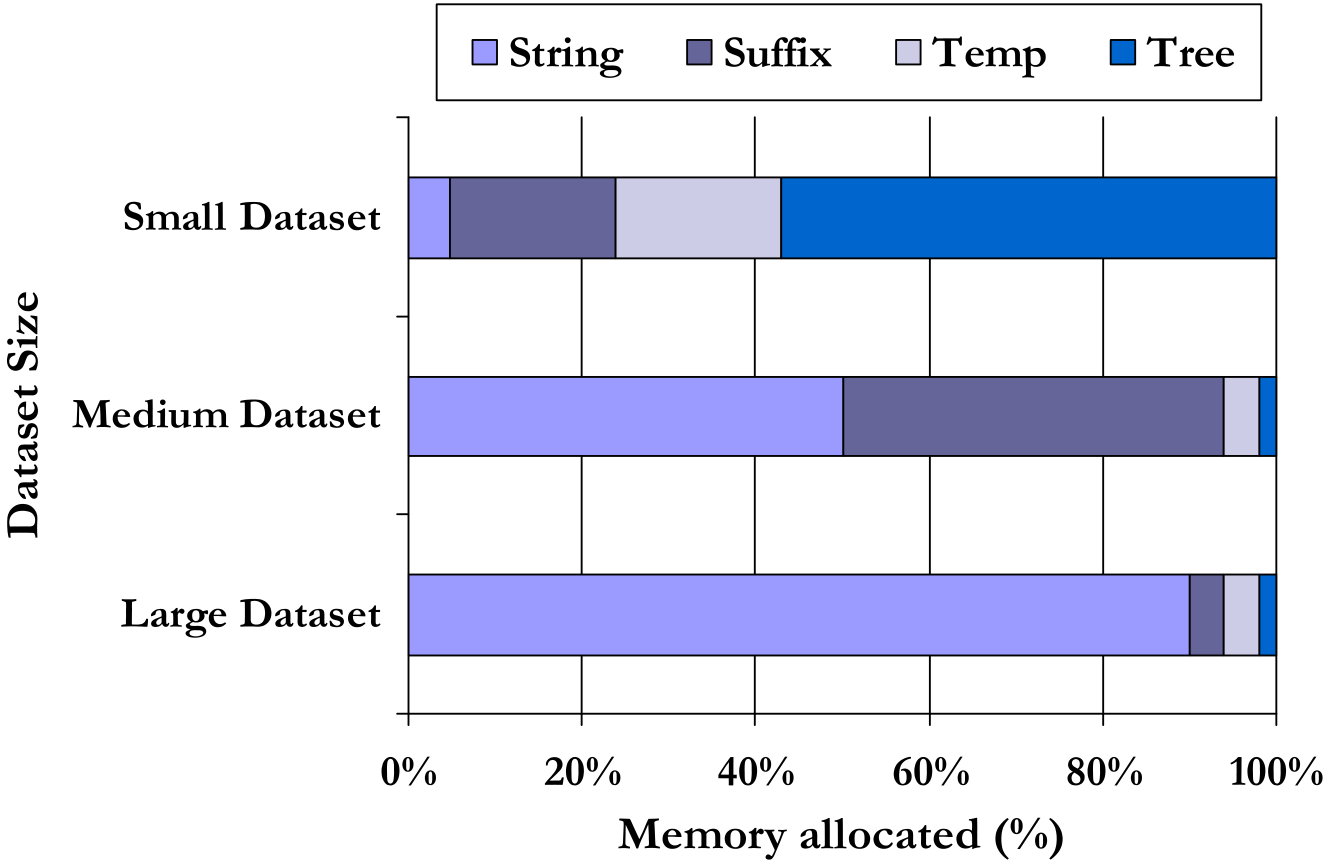
Buffer Management

Replacement Policy

Data Structure	**Access Pattern**	**Replacement Policy**
String	Random	Random
Suffixes	Less Random	Random/LRU
Temp	Sequential	MRU
Tree	Mostly Sequential	LRU

Buffer Management

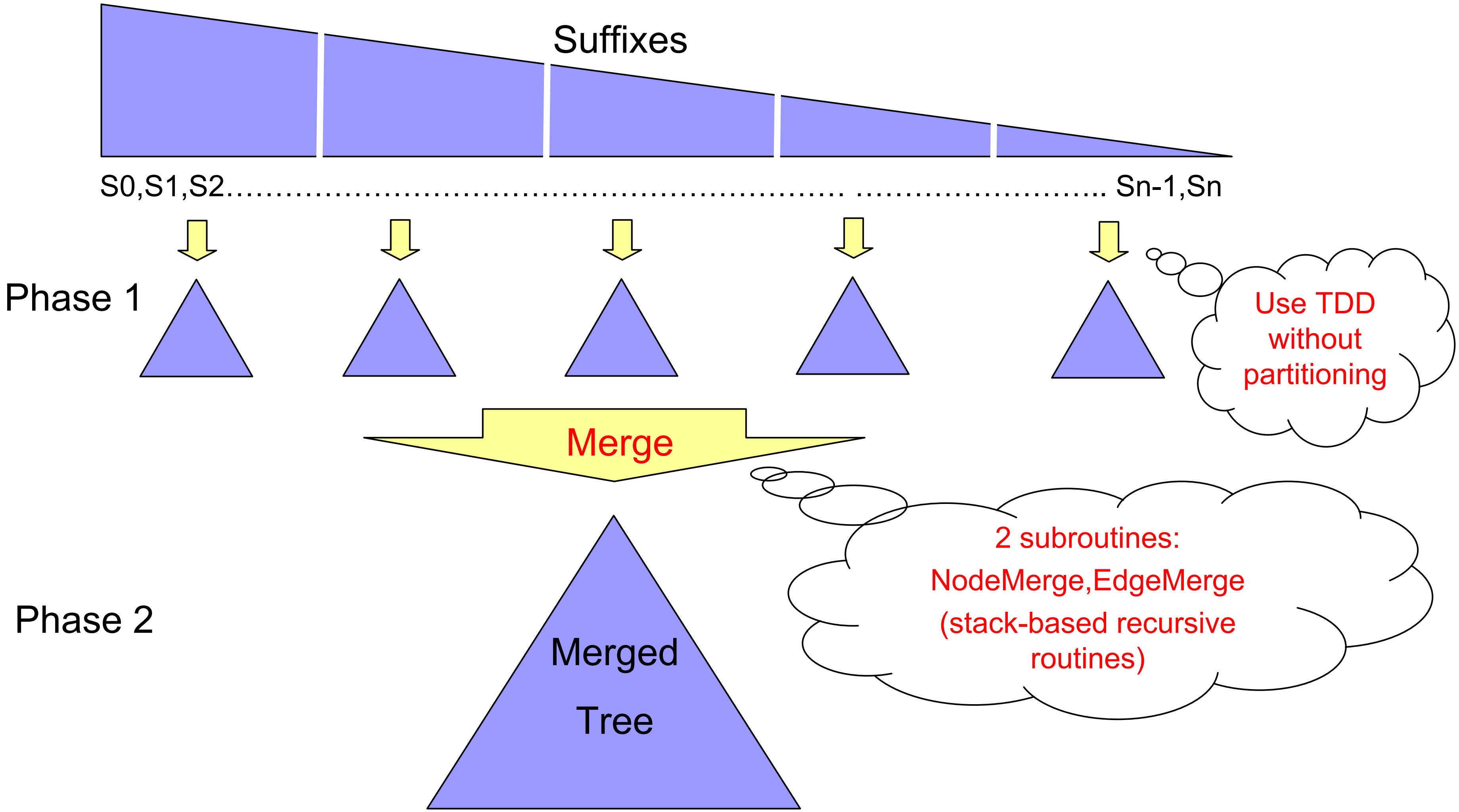
Buffer Allocation



Merge-Based Algorithm (ST-Merge)

- When string size is larger than main memory, the performance of TDD degrades rapidly.
- Reduce random accesses to large string by partitioning suffixes so that each partition contains adjacent suffixes.

ST-Merge Algorithm



Suffixes

S0,S1,S2…………………………………………………… …………………….. Sn-1,Sn

Phase 1

Use TDD without partitioning

Merge

Phase 2

Merged Tree

2 subroutines: NodeMerge,EdgeMerge (stack-based recursive routines)

Phase 1

String: ATTAGTACA$

0 1 2 3 4 5 6 7 8 9

S0: ATTAGTACA$

S1: TTAGTACA$

S2: TAGTACA$

S3: AGTACA$

S4: GTACA$

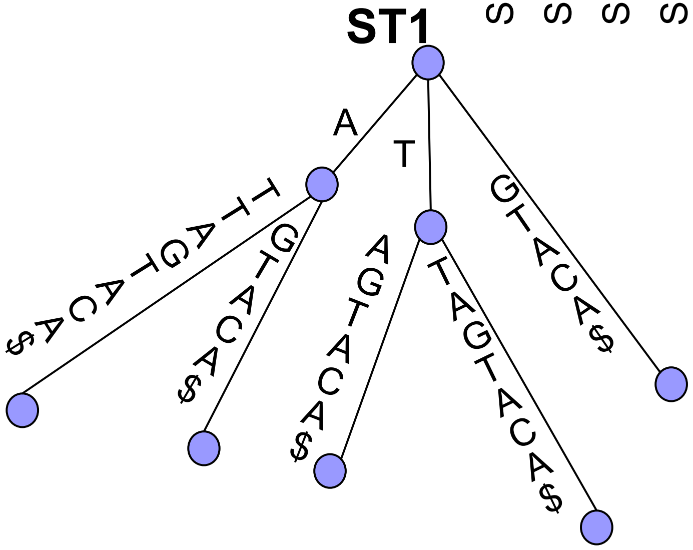
S5: TACA$

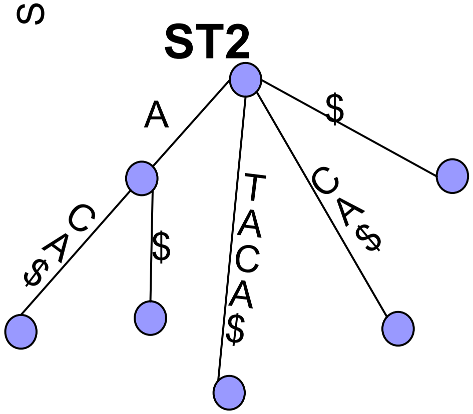
S6: ACA$

S7: CA$

S8: A$

S9: $





0	1	2	3	4	5	6	7	8
0	7	1	5	4_R	3	2_R	1	4_R

0	1	2	3	4	5	6
6	5	5	7	9_R	7	9_R

Access to String:

ATTAGTACA$

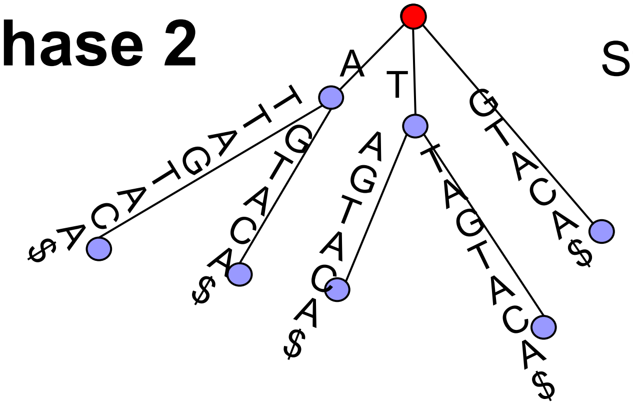
ATTAGTACA$

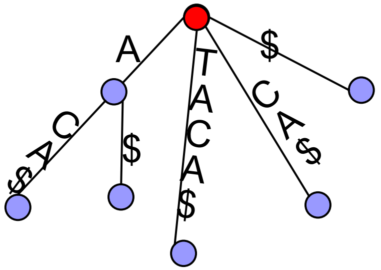
Better locality of reference to the string!

Phase 2

String: ATTAGTACA$

0 1 2 3 4 5 6 7 8 9



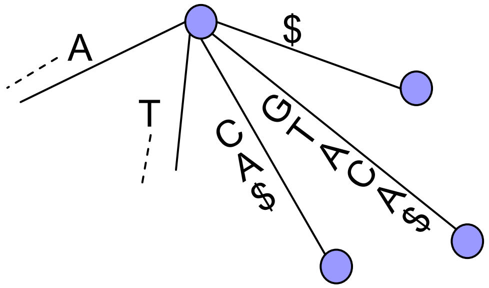


ST1:

0	1	2	3	4	5	6	7	8
0	7	1	5	4_R	3	2_R	1	4_R

ST2:

0	1	2	3	4	5	6
6	5	5	7	9_R	7	9_R



ST1[0,1]: A

ST2[0,1]: A

ST1[2,3]: T

ST2[2] : TACA$

ST2[3] : CA$

ST1[4] : GTACA$

ST2[4] : $

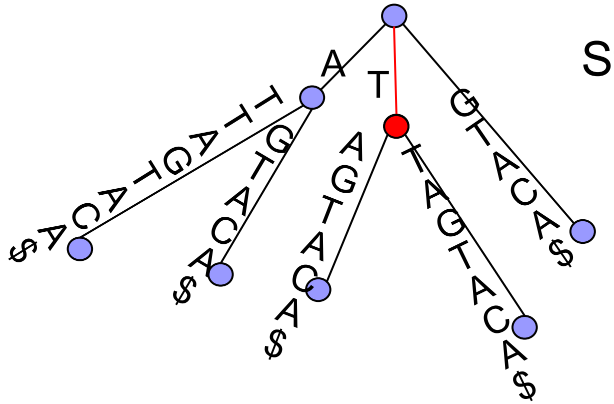
Edge_Merge

ST1[2], ST2[2]

Edge_Merge

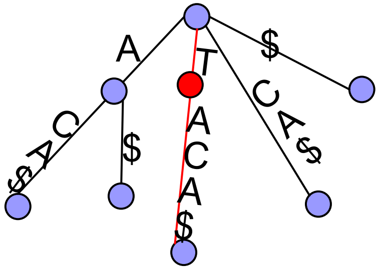
ST1[0], ST2[0]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		1		7	4	9_R									



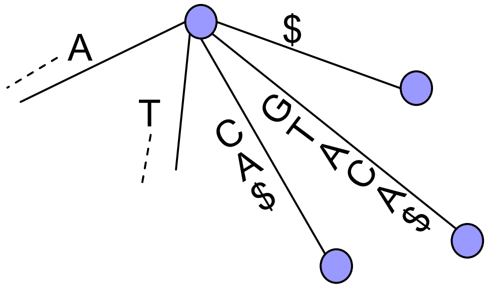
String: ATTAGTACA$

0 1 2 3 4 5 6 7 8 9



	0	1	2	3	4	5	6	7	8
ST1:	0	7	1	5	4_R	3	2_R	1	4_R

	0	1	2	3	4	5	6
ST2:	6	5	5	7	9_R	7	9_R



LCP=T

ST1[2,3]: T

ST2[2] : TACA$

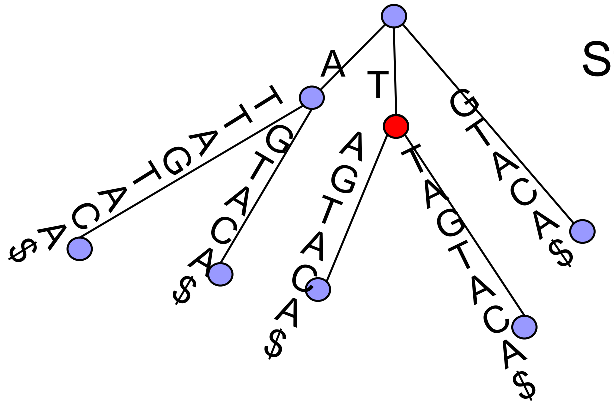
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		1		7	4	9_R									

Edge_Merge

ST1[2], ST2[2]

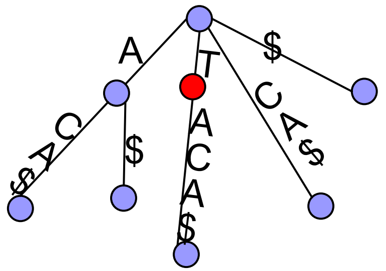
Edge_Merge

ST1[0], ST2[0]



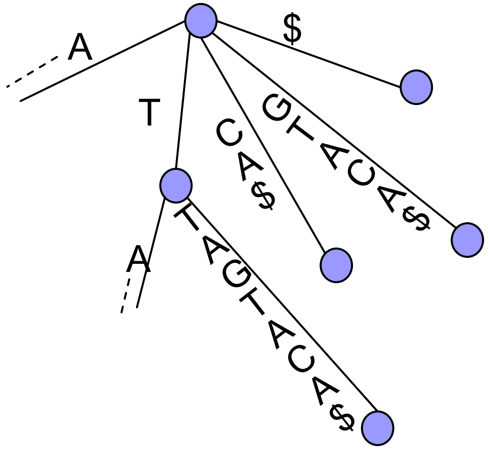
String: ATTAGTACA$

0 1 2 3 4 5 6 7 8 9



	0	1	2	3	4	5	6	7	8
ST1:	0	7	1	5	4_R	3	2_R	1	4_R

	0	1	2	3	4	5	6
ST2:	6	5	5	7	9_R	7	9_R



ST1[5] : AGTACA$

ST2[2](1): ACA$

ST1[6] : TAGTACA$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		1	7	7	4	9_R	3		2_R						

Edge_Merge

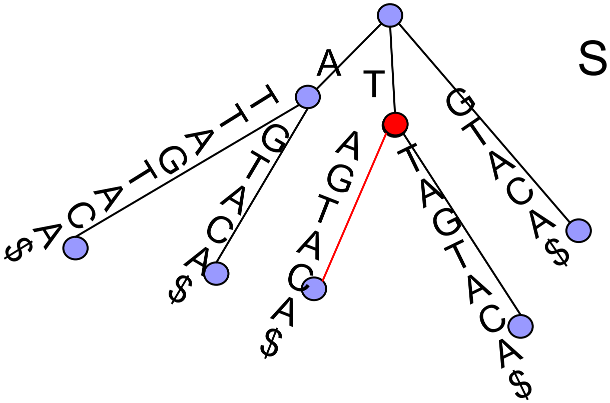
Node_Merge

ST1[5]:1,ST2[2]:2

ST1[5], ST2[2]:1

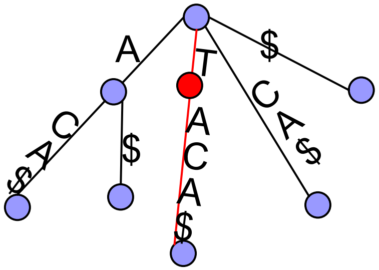
Edge_Merge

ST1[0], ST2[0]



String: ATTAGTACA$

0 1 2 3 4 5 6 7 8 9

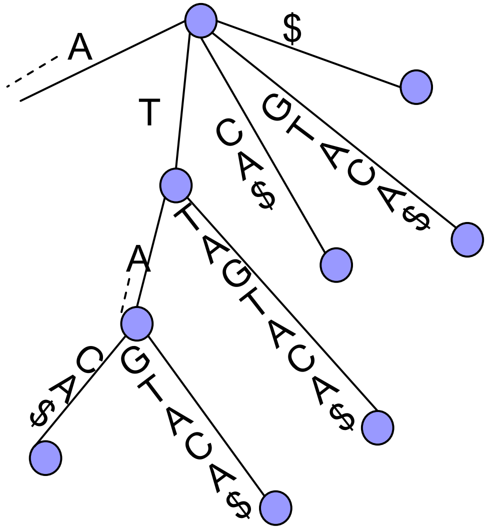


ST1:

0	1	2	3	4	5	6	7	8
0	7	1	5	4_R	3	2_R	1	4_R

ST2:

0	1	2	3	4	5	6
6	5	5	7	9_R	7	9_R



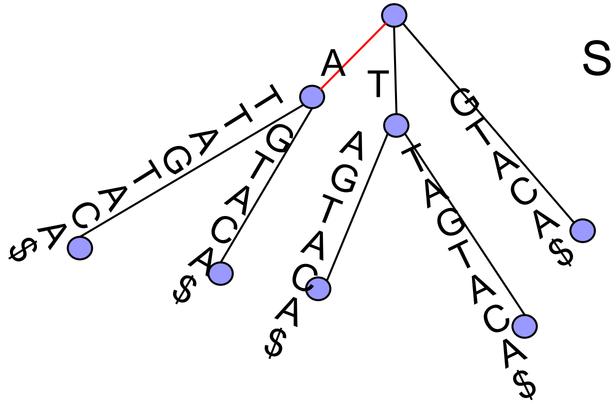
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		1	7	7	4	9_R	3	10	2_R	7	4_R				

Edge_Merge

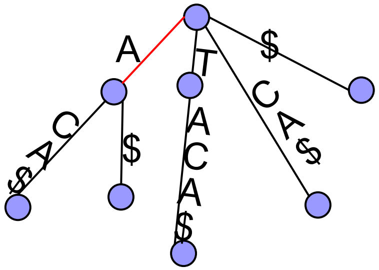
ST1[5]:1,ST2[2]:2

Edge_Merge

ST1[0], ST2[0]



String: ATTAGTACA$
0 1 2 3 4 5 6 7 8 9

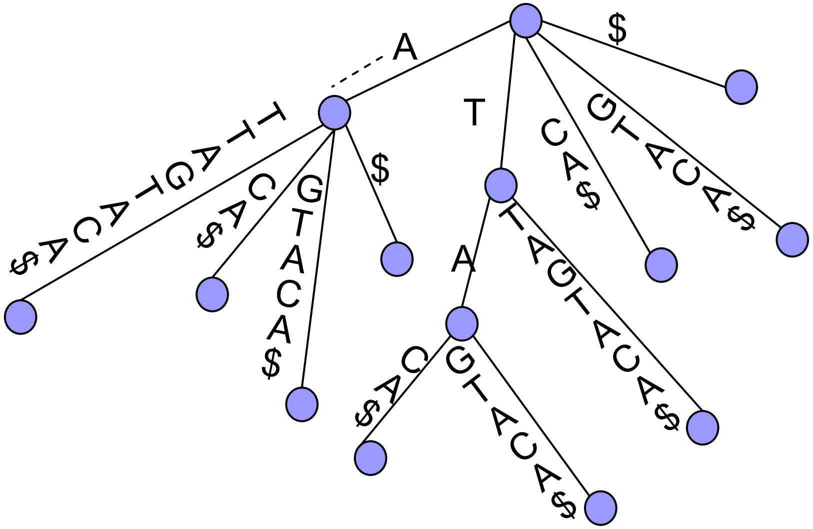


ST1:

0	1	2	3	4	5	6	7	8
0	7	1	5	4_R	3	2_R	1	4_R

ST2:

0	1	2	3	4	5	6
6	5	5	7	9_R	7	9_R



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	12	1	7	7	4	9_R	3	10	2_R	7	4_R	1	7	4	9_R

Edge_Merge

ST1[0], ST2[0]

Analysis of ST-Merge

- Average Time Complexity

  phase 1: $O(nlog_{|A|}(n/k))$

  phase 2: $O(n^2)$

- Further Minimize Random Accesses

  phase 1: access to the string focuses on a small portion of the string for each partition

  phase 2: input trees and output trees are mostly sequentially accessed; access to the string shows more locality

- When the string size is significantly larger then main memory, ST-Merge performs better than TDD.
- When the string size is smaller than main memory, ST-Merge reduces to TDD.

Roadmap

- Background & Motivation
- Our Approaches
  - Top-Down Disk-Based Algorithm (TDD)
  - Merge-Based Algorithm (ST-Merge)
- **Experiments and Analysis**

Experimental Setup

- Platform

  Intel Pentium 4 2.8GHZ, 2 GB Main Memory, Maxtor Atlas 10K IV SCSI Disk

- Use raw device

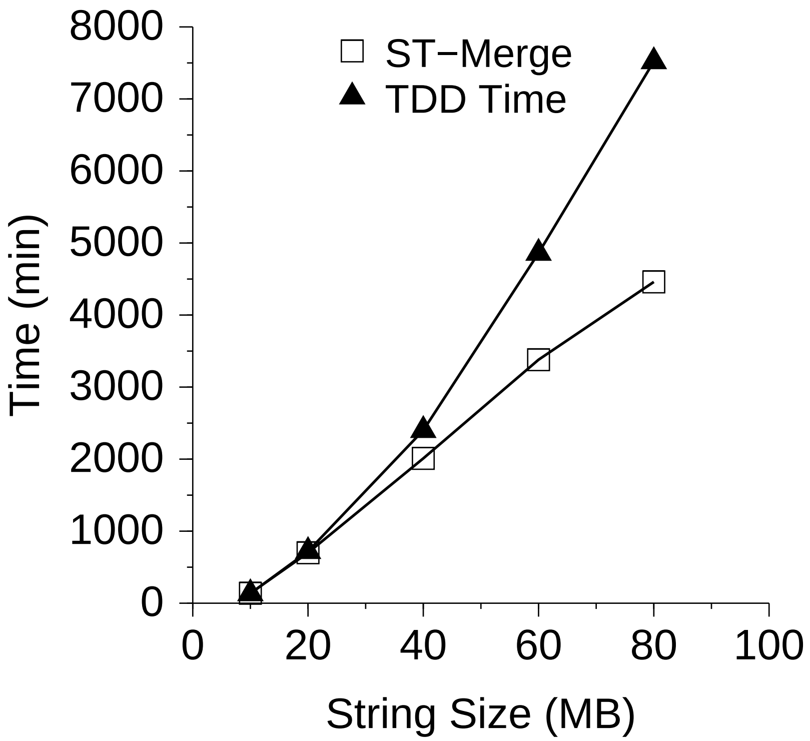
  Eliminate the effect of operating system buffering

Performance Comparison Disk-based Construction

Data Source	**Symbols (10^6)**	**Hunt (min)**	**TDD (min)**	**Speedup**
Trembl (Protein)	338	236.7	32.0	**7.4**
H.Chr1 (DNA)	227	97.50	17.83	**5.5**
Guten (English)	407	463.3	46.67	**9.9**
HG (DNA)	3,000	**N/A**	**30hrs**	**N/A**

TDD is 5-10 times faster!

TDD vs. ST-Merge



- Limit the main memory size to 6MB
- Uniformly distributed DNA sequence data from 10MB-80MB

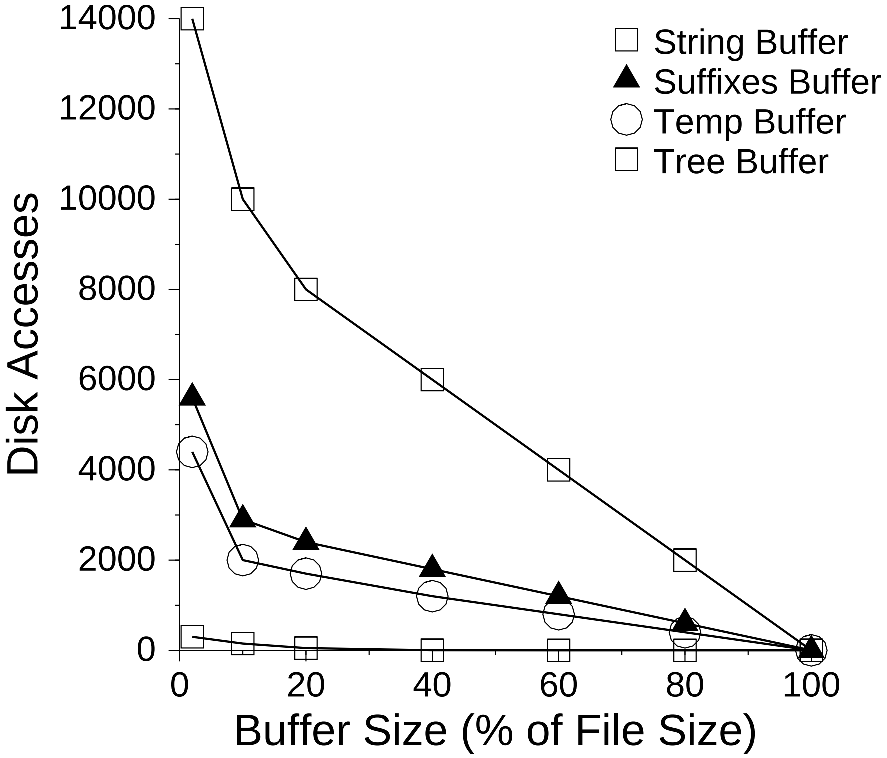
Conclusion

- Constructing large persistent suffix trees using existing algorithms is impractical
- We proposed 2 algorithms: TDD & ST-Merge
- When string size is roughly same as the memory size, TDD is faster than the best known on-disk construction algorithm by **4x-10x**
- ST-Merge beats TDD when the string size is significantly larger (x3 or more) than main memory
- Using TDD and ST-Merge, large scale suffix tree construction is now practical

Thanks!

Questions?

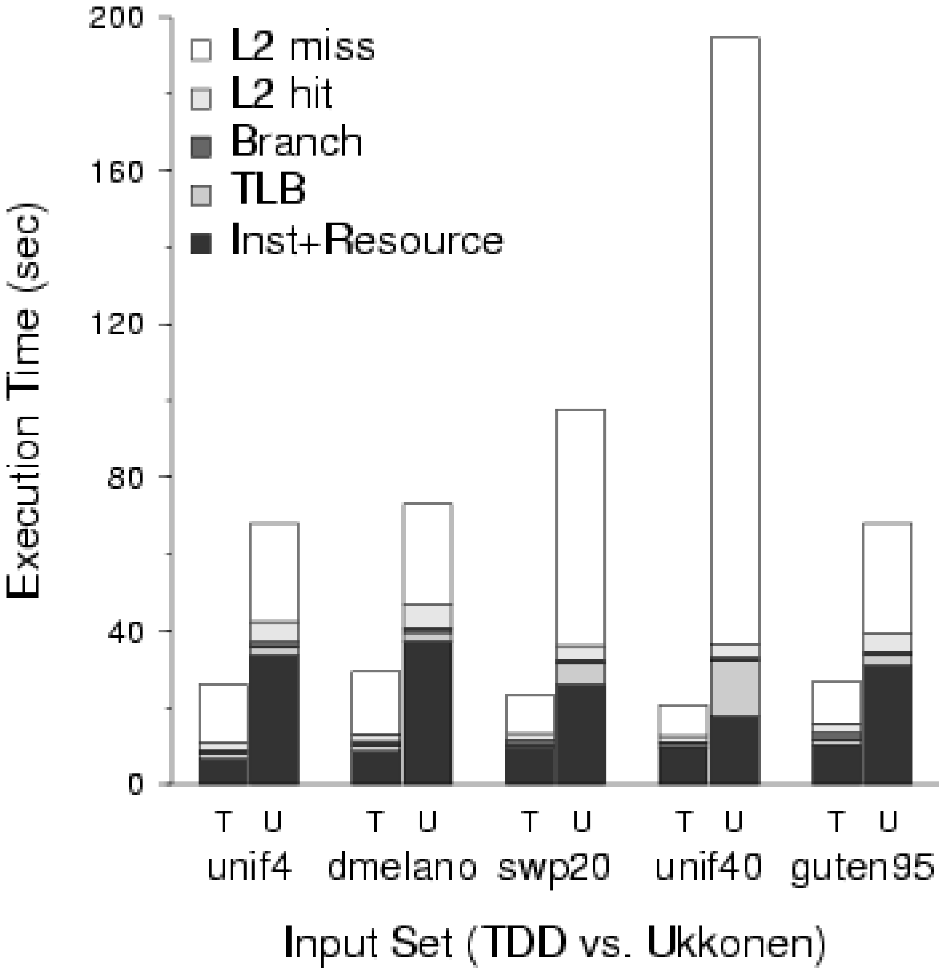
Buffer Allocation



Main Memory Data Sources

Data Sources	Description	Symbols (10^6)
Dmelano	D. Melanogaster Chr.2 (DNA)	20
Guten95	Gutenberg Project, Year 1995 (English Text)	20
Swp20	Slice of SwissProt (protein)	20
Unif4	4-char alphabet, uniform dist.	20
Unif40	40-char alphabet, uniform dist.	20

Main-memory construction: Cycle Time Breakdown



All data sources:
20 MB

On-Disk Data Sources

Data Sources	Description	Symbols(10^6)
Swp	Entire UniProt/SwissProt (Protein)	53
H.Chr1-50	50 MB slice of Human Chromosome -1 (DNA)	50
Guten03	2003 Directory of Gutenberg Project (English Text)	58
Trembl	TrEMBL (Protein)	338
H.Chr1	Entire Human Chromosome-1 (DNA)	227
Guten	Entire Gutenberg Collection (English Text)	407
HG	Entire Human Genome (DNA)	3,000

On-Disk Performance Comparison

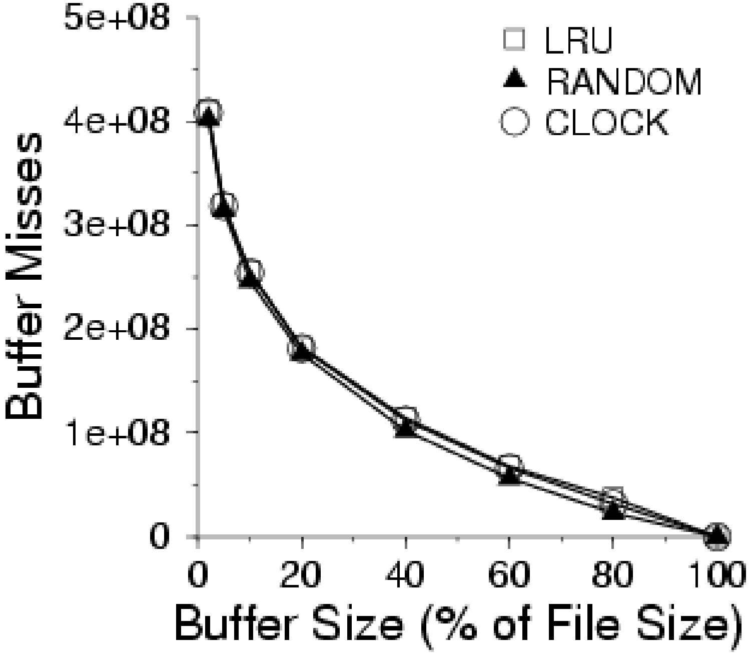
Data Source	Symbols (10^6)	Hunt (min)	TDD (min)	Speedup	Skew (min)
Swp	53	13.95	2.78	5.0	3.88
H.Chr1-50	50	11.47	2.02	5.7	3.21
Guten03	58	22.5	6.03	3.7	3.94
Trembl	338	263.7	32.0	7.4	>9hrs
H.Chr1	227	97.50	17.83	5.5	>9hrs
Guten	407	463.3	46.67	9.9	>9hrs
HG	3,000	N/A	30hrs	N/A	N/A

Datasets for Buffer Management Experiment

Data structure	SwissProt (53MB) (#pages)	Human DNA(50MB) (#pages)
String	6,250	6,250
Suffixes	1,250	6,250
Temp	1,250	6,250
Tree	4,100	16,200

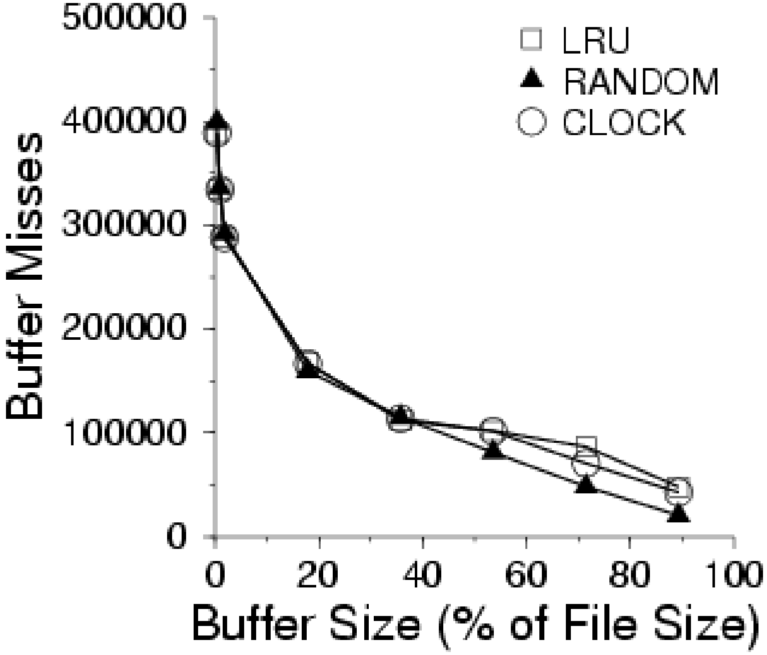
Page size: 8KB, prefixlen=1

Buffer Size Simulation (DNA)



(b) H.Chr1

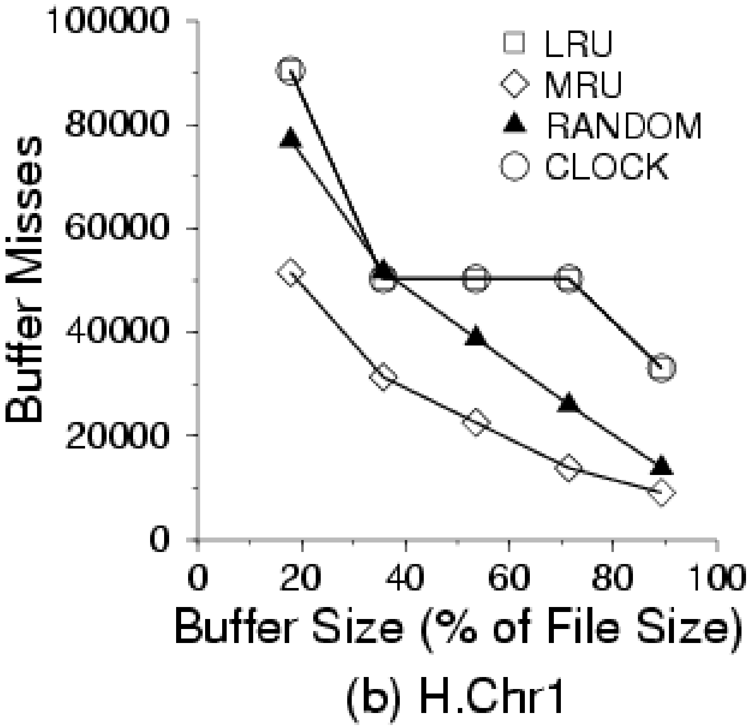
String Buffer



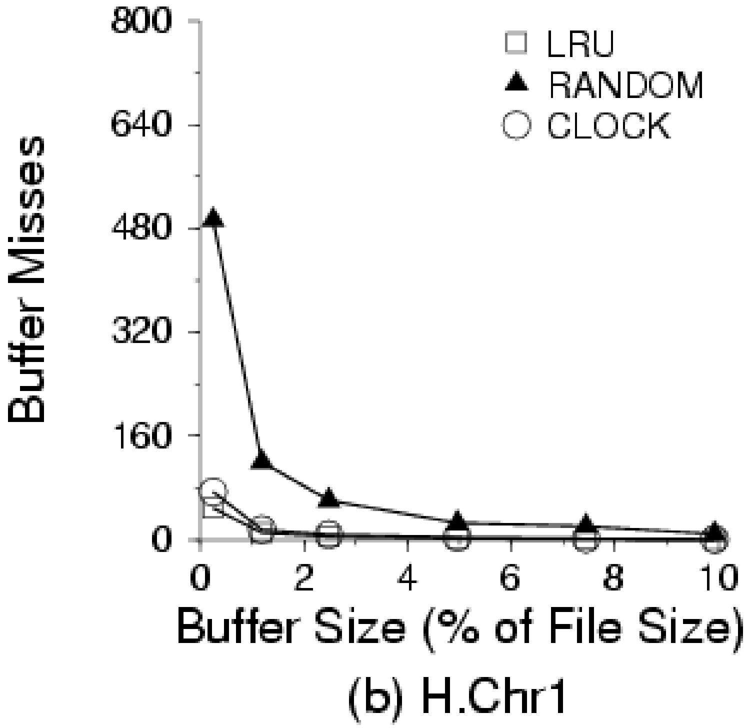
(b) H.Chr1

Suffix Buffer

Buffer Size Simulation (DNA)

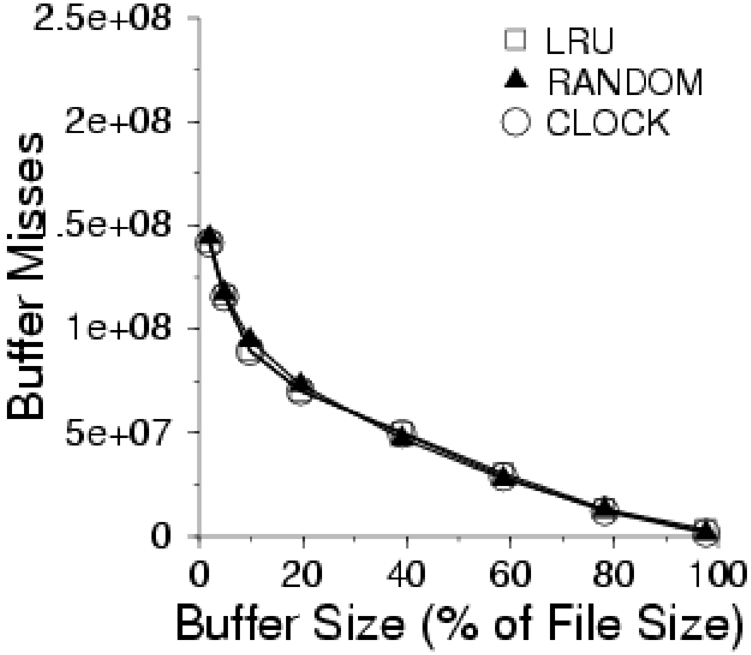


Temp Buffer



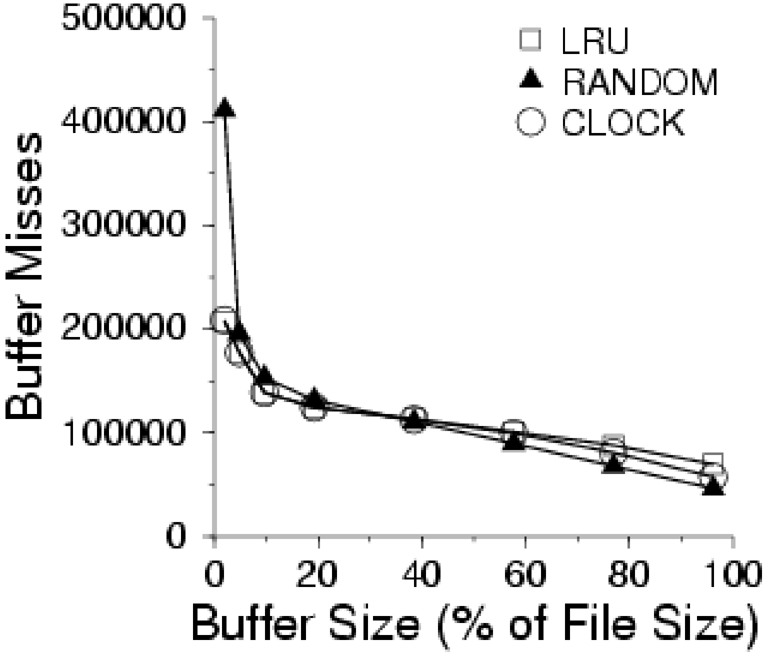
Tree Buffer

Buffer Size Simulation (Protein)



(a) SwissProt

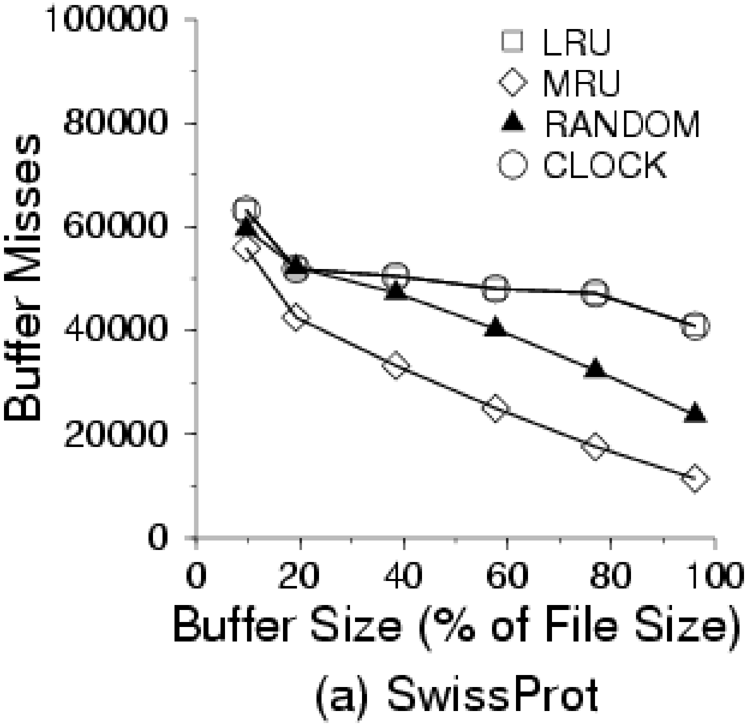
String Buffer



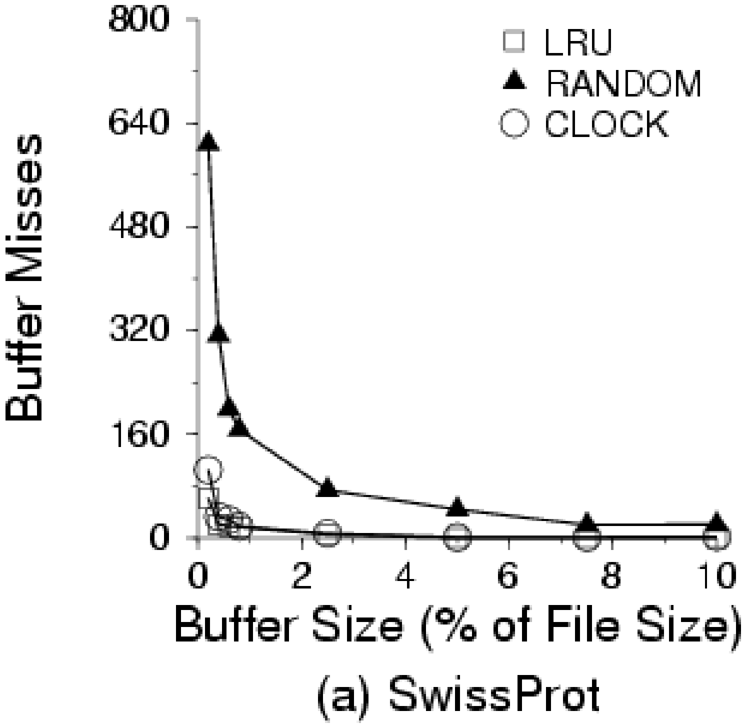
(a) SwissProt

Suffix Buffer

Buffer Size Simulation (Protein)

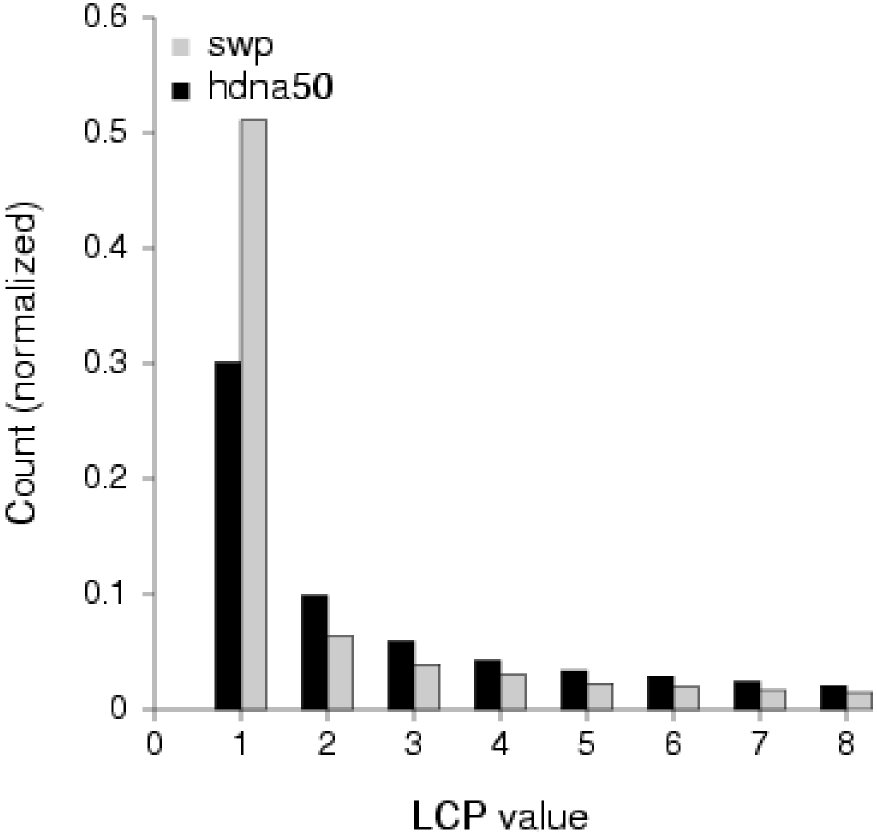


Temp Buffer



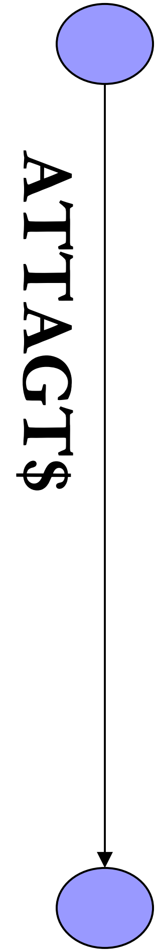
Tree Buffer

LCP Histogram



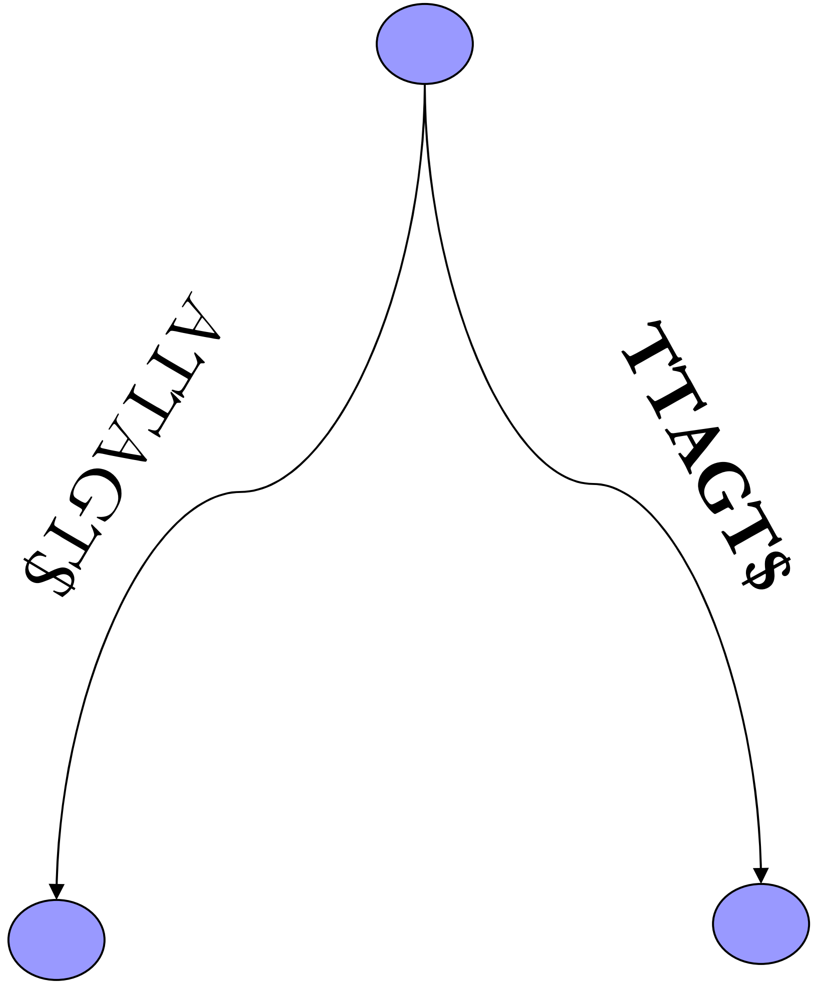
Brute-Force Algorithm

ATTAGT$



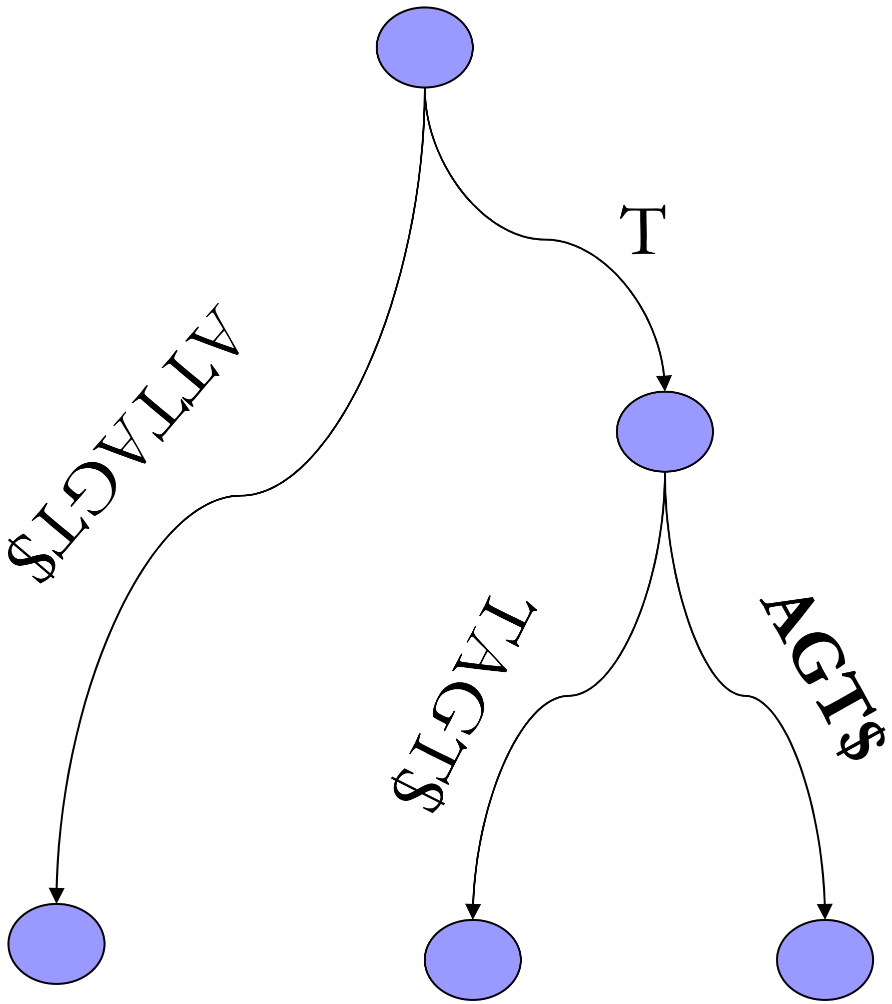
Brute-Force Algorithm

ATTAGT$



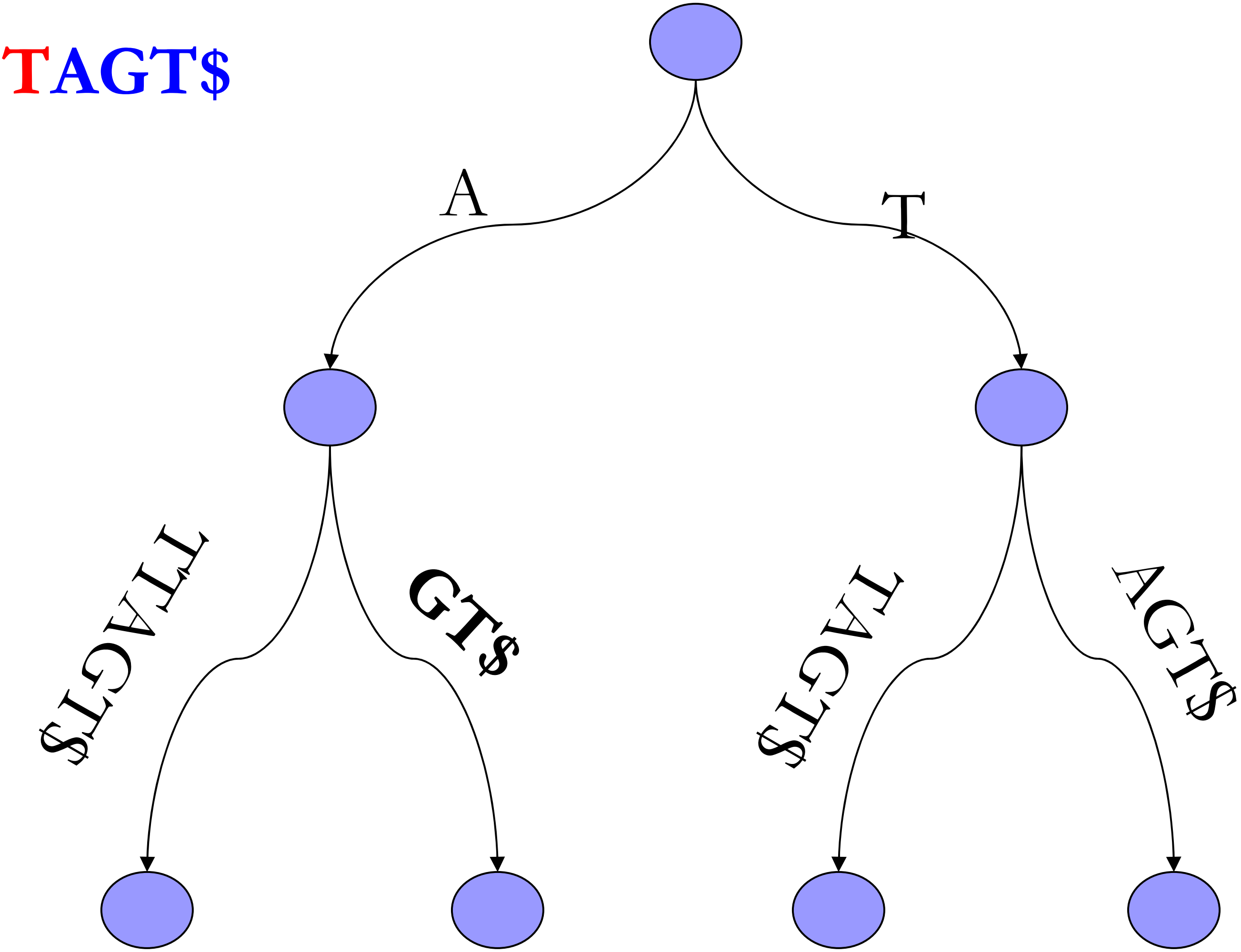
Brute-Force Algorithm

ATTAGT$



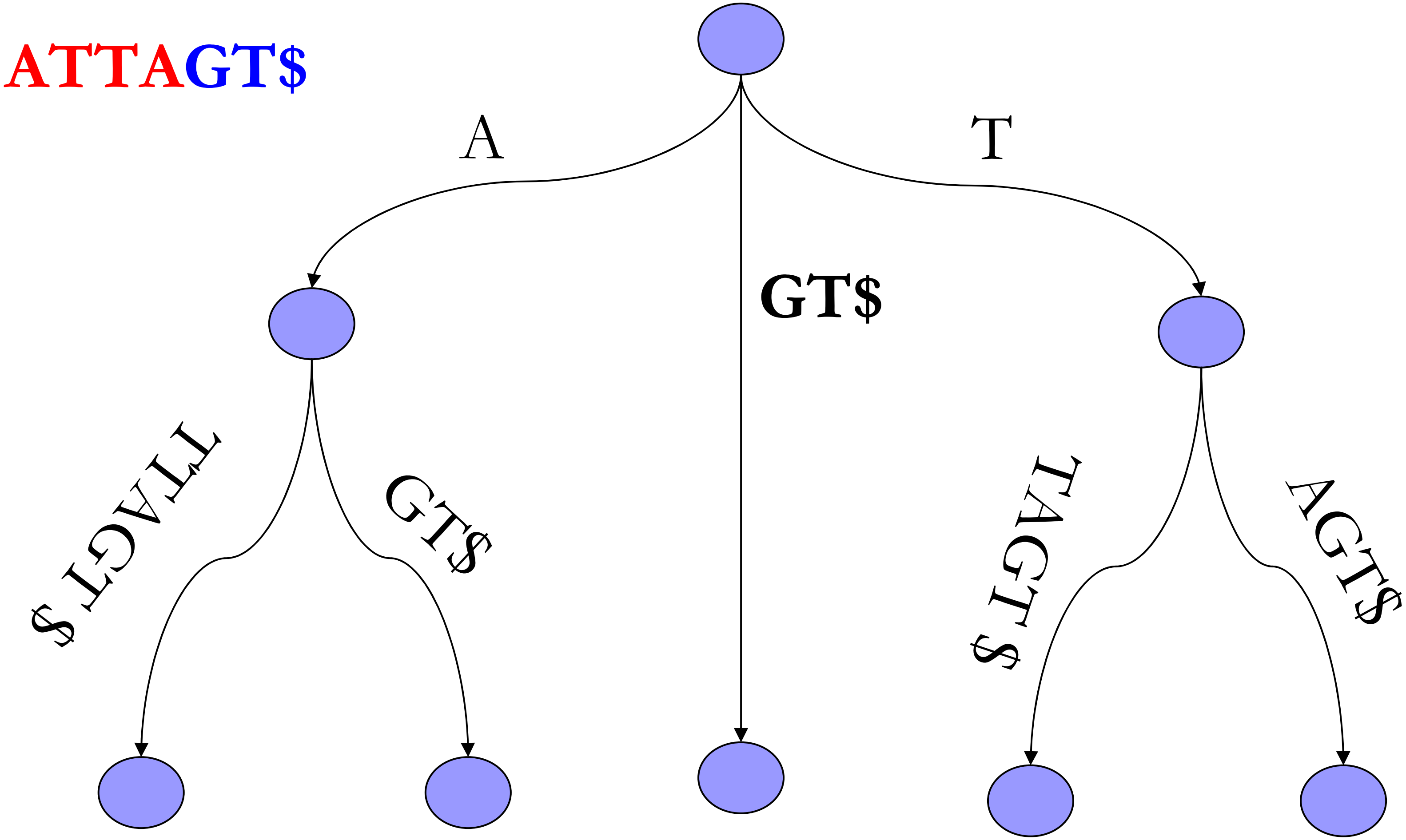
Brute-Force Algorithm

ATTAGT$



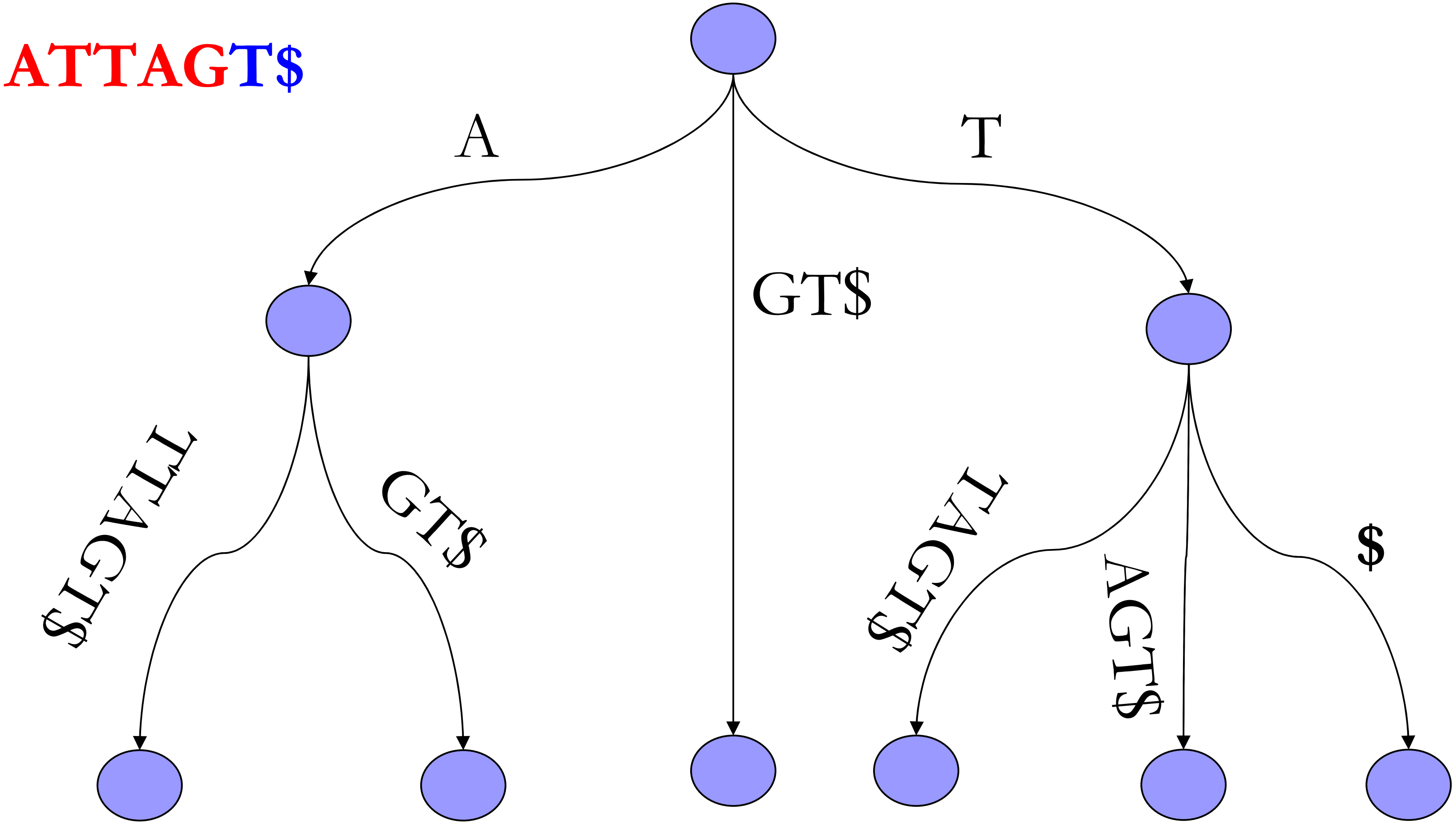
Brute-Force Algorithm

ATTAGT$



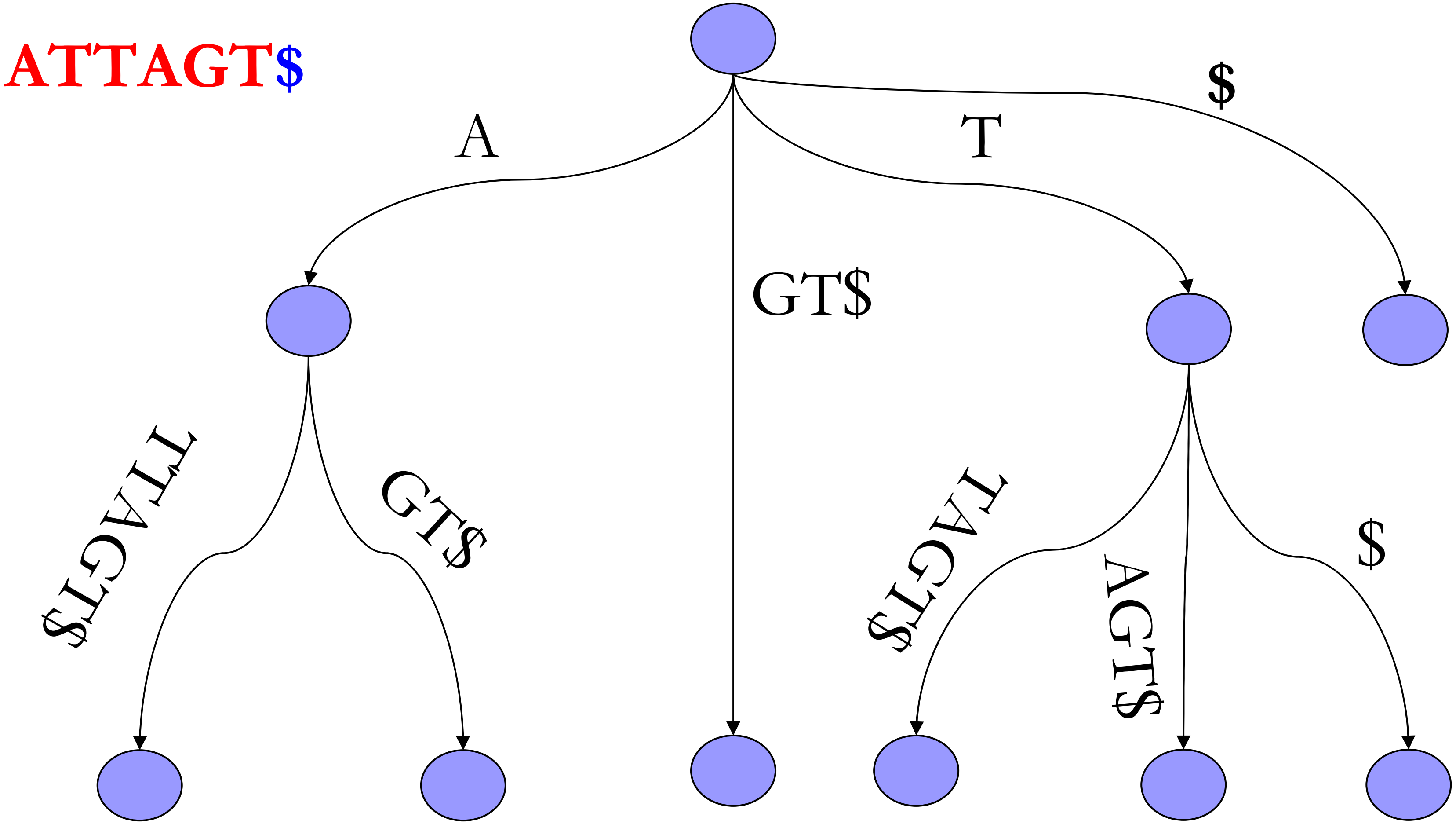
Brute-Force Algorithm

ATTAGT$

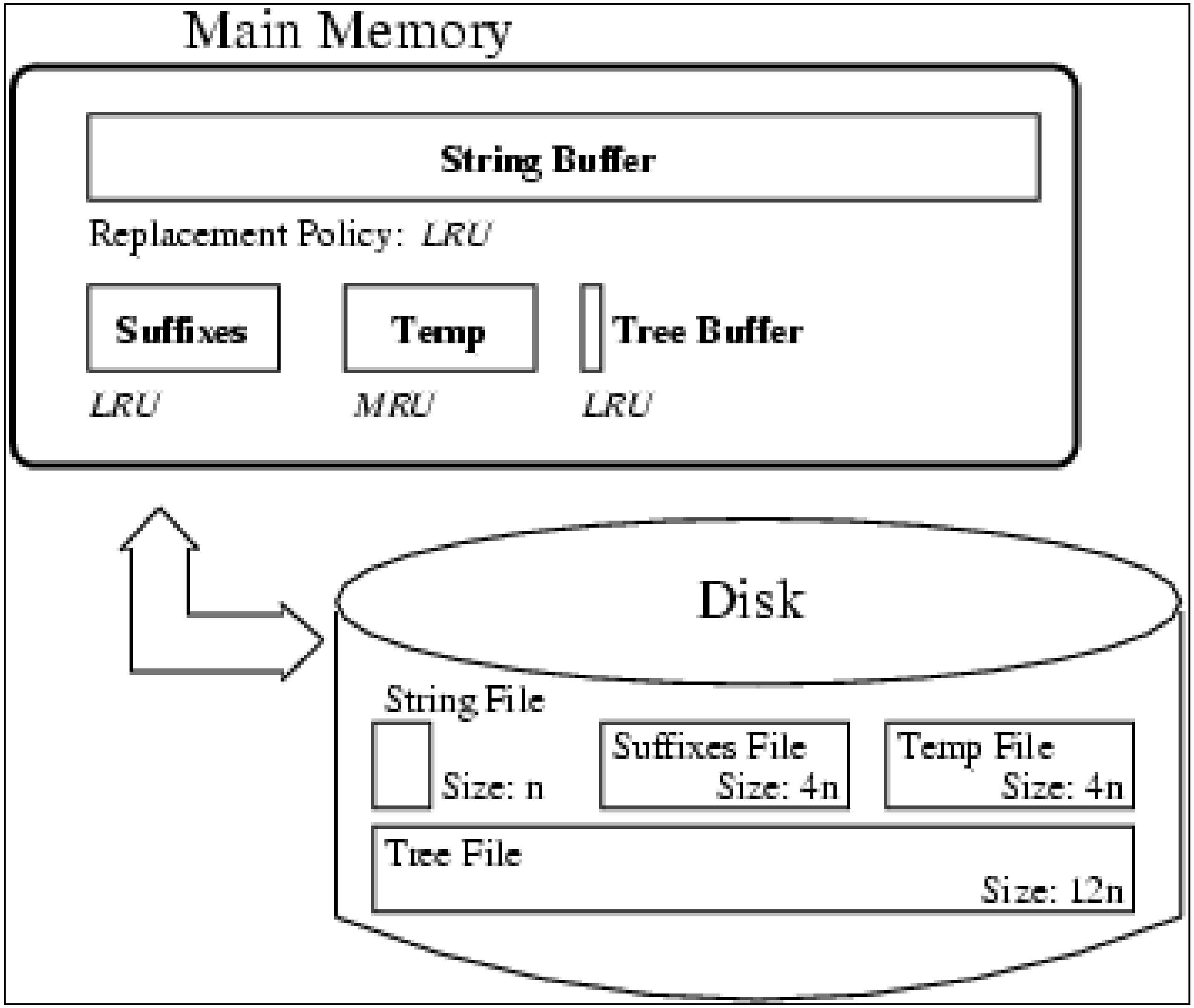


Brute-Force Algorithm

ATTAGT$



TDD Architecture


Main Memory
String Buffer
Replacement Policy: LRU
Suffixes
Temp
Tree Buffer
LRU
MRU
LRU
Disk
String File
Size: n
Suffixes File
Size: 4n
Temp File
Size: 4n
Tree File
Size: 12n