A Memory-Efficient Tree Edit Distance Algorithm

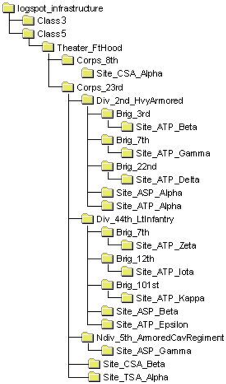
Mateusz Pawlik and Nikolaus Augsten

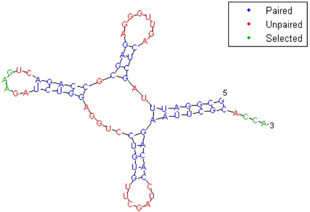
University of Salzburg, Austria

DEXA 2014

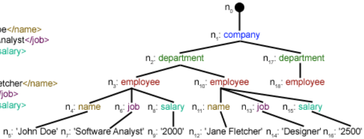
Tree-structured data

- hierarchical data is often modelled as trees
- an interesting query computes the similarity between trees






```
<company>
 <department>
  <employee>
   <name>John Doe</name>
   <job>Software Analyst</job>
   <salary>2000</salary>
  </employee>
  <employee>
   <name>Jane Fletcher</name>
   <job>Designer</job>
   <salary>2500</salary>
  </employee>
 </department>
 <department>
  <employee>
  </employee>
 </department>
</company>
```

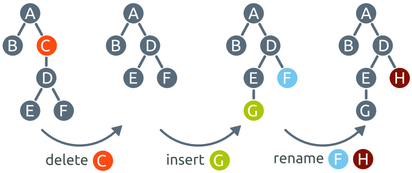


Tree Edit Distance (TED)

- TED is a standard measure to tree similarity

Tree Edit Distance (TED)

- TED is a standard measure to tree similarity
- TED is the minimum number of edit operations to transform one tree into another



Recursive solution for TED

- TED has a recursive solution which decomposes trees into subforests

Recursive solution for TED

- TED has a recursive solution which decomposes trees into subforests
- pairs of subforests make subproblems

Recursive solution for TED

- TED has a recursive solution which decomposes trees into subforests
- pairs of subforests make subproblems
- use distances of smaller subproblems to compute larger

Recursive solution for TED

- TED has a recursive solution which decomposes trees into subforests
- pairs of subforests make subproblems
- use distances of smaller subproblems to compute larger
- at each step of the recursion there are two possibilities (LEFT or RIGHT) to obtain smaller subforests

remove		root node
remove	LEFT / RIGHT	subtree
leave only		subtree

Recursive solution for TED

- TED has a recursive solution which decomposes trees into subforests
- pairs of subforests make subproblems
- use distances of smaller subproblems to compute larger
- at each step of the recursion there are two possibilities (LEFT or RIGHT) to obtain smaller subforests

remove		root node
remove	LEFT / RIGHT	subtree
leave only		subtree

- every choice leads to the correct result

Recursive solution for TED

- TED has a recursive solution which decomposes trees into subforests
- pairs of subforests make subproblems
- use distances of smaller subproblems to compute larger
- at each step of the recursion there are two possibilities (LEFT or RIGHT) to obtain smaller subforests

remove		root node
remove	LEFT / RIGHT	subtree
leave only		subtree

- every choice leads to the correct result
- different choices lead to different #subproblems

Recursive solution for TED

- TED has a recursive solution which decomposes trees into subforests
- pairs of subforests make subproblems
- use distances of smaller subproblems to compute larger
- at each step of the recursion there are two possibilities (LEFT or RIGHT) to obtain smaller subforests

remove		root node
remove	LEFT / RIGHT	subtree
leave only		subtree

- every choice leads to the correct result
- different choices lead to different #subproblems
- GOAL of TED algorithms - minimize #subproblems

State of the art in TED

Algorithm		Time	Space	Comments
Tai	1979	$O(n^6)$	$O(n^6)$	first algorithm
Zhang&Shasha	1989	$O(n^4)$	$O(n^2)$	efficient for balanced trees $O(n^2 \log^2 n)$
Klein	1998	$O(n^3 \log n)$	$O(n^3 \log n)$	bad space complexity
Demaine et al.	2009	$O(n^3)$	$O(n^2)$	worst case is frequent
Pawlik&Augsten (RTED)	2011	$O(n^3)$	$O(n^2)$	compute optimal strategy

State of the art in TED

Algorithm		Time	Space	Comments
Tai	1979	$O(n^6)$	$O(n^6)$	first algorithm
Zhang&Shasha	1989	$O(n^4)$	$O(n^2)$	efficient for balanced trees $O(n^2 \log^2 n)$
Klein	1998	$O(n^3 \log n)$	$O(n^3 \log n)$	bad space complexity
Demaine et al.	2009	$O(n^3)$	$O(n^2)$	worst case is frequent
Pawlik&Augsten (RTED)	2011	$O(n^3)$	$O(n^2)$	compute optimal strategy

- TED algorithms:
  - implement the recursive solution using dynamic programming
  - develop strategies to minimize #subproblems

State of the art in TED

Algorithm		Time	Space	Comments
Tai	1979	$O(n^6)$	$O(n^6)$	first algorithm
Zhang&Shasha	1989	$O(n^4)$	$O(n^2)$	efficient for balanced trees $O(n^2 \log^2 n)$
Klein	1998	$O(n^3 \log n)$	$O(n^3 \log n)$	bad space complexity
Demaine et al.	2009	$O(n^3)$	$O(n^2)$	worst case is frequent
Pawlik&Augsten (RTED)	2011	$O(n^3)$	$O(n^2)$	compute optimal strategy

- TED algorithms:
  - implement the recursive solution using dynamic programming
  - develop strategies to minimize #subproblems
- hard-coded stratgies are bad for specific tree shapes

State of the art in TED

Algorithm		Time	Space	Comments
Tai	1979	$O(n^6)$	$O(n^6)$	first algorithm
Zhang&Shasha	1989	$O(n^4)$	$O(n^2)$	efficient for balanced trees $O(n^2 \log^2 n)$
Klein	1998	$O(n^3 \log n)$	$O(n^3 \log n)$	bad space complexity
Demaine et al.	2009	$O(n^3)$	$O(n^2)$	worst case is frequent
Pawlik&Augsten (RTED)	2011	$O(n^3)$	$O(n^2)$	compute optimal strategy

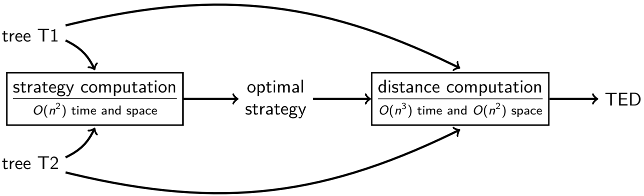
- TED algorithms:
  - implement the recursive solution using dynamic programming
  - develop strategies to minimize #subproblems
- hard-coded stratgies are bad for specific tree shapes
- RTED uses the optimal strategy

State of the art in TED

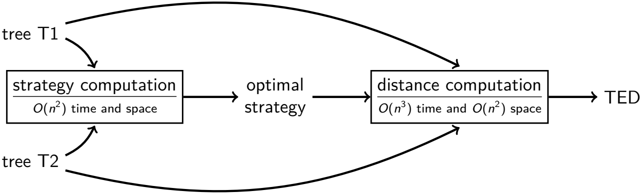
Algorithm		Time	Space	Comments
Tai	1979	$O(n^6)$	$O(n^6)$	first algorithm
Zhang&Shasha	1989	$O(n^4)$	$O(n^2)$	efficient for balanced trees $O(n^2 \log^2 n)$
Klein	1998	$O(n^3 \log n)$	$O(n^3 \log n)$	bad space complexity
Demaine et al.	2009	$O(n^3)$	$O(n^2)$	worst case is frequent
Pawlik&Augsten (RTED)	2011	$O(n^3)$	$O(n^2)$	compute optimal strategy

- TED algorithms:
  - implement the recursive solution using dynamic programming
  - develop strategies to minimize #subproblems
- hard-coded stratgies are bad for specific tree shapes
- RTED uses the optimal strategy
- unfortunately, RTED has a memory problem

RTED algorithm

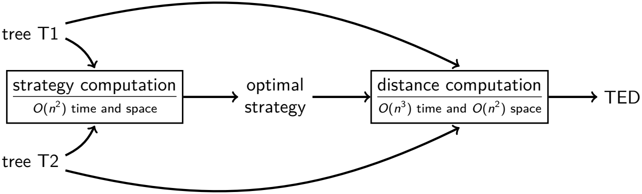


RTED algorithm



- overhead of strategy computation is very low

RTED algorithm



- overhead of strategy computation is very low
- #subproblems is minimal compared to previous solutions

Memory problem in RTED

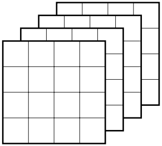
- both, strategy and distance, are computed in $O(n^2)$ space

Memory problem in RTED

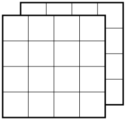
- both, strategy and distance, are computed in $O(n^2)$ space
- the devil lies in the constants

Memory problem in RTED

- both, strategy and distance, are computed in $O(n^2)$ space
- the devil lies in the constants
- strategy and distance computations are complex processes:
  - based on computing values in quadratic-size matrices



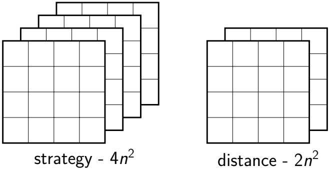
strategy - $4n^2$



distance - $2n^2$

Memory problem in RTED

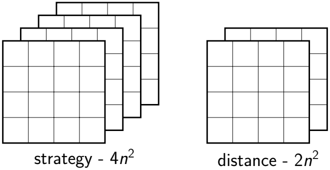
- both, strategy and distance, are computed in $O(n^2)$ space
- the devil lies in the constants
- strategy and distance computations are complex processes:
  - based on computing values in quadratic-size matrices



- strategy may use 2x the memory of distance computation: $4n^2$ vs $2n^2$

Memory problem in RTED

- both, strategy and distance, are computed in $O(n^2)$ space
- the devil lies in the constants
- strategy and distance computations are complex processes:
  - based on computing values in quadratic-size matrices

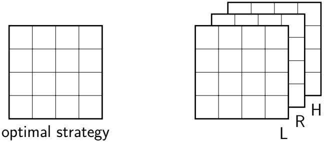


- strategy may use 2x the memory of distance computation: $4n^2$ vs $2n^2$
- strategy computation is a memory bottleneck

Memory allocation in RTED strategy

4 matrices of quadratic size are used:

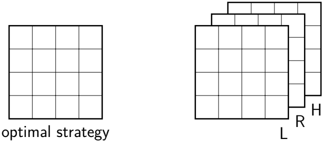
- 1 for optimal strategy
- 3 for intermediate results



Memory allocation in RTED strategy

4 matrices of quadratic size are used:

- 1 for optimal strategy
- 3 for intermediate results

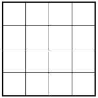


strategy computation (high-level description):

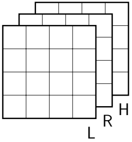
Memory allocation in RTED strategy

4 matrices of quadratic size are used:

- 1 for optimal strategy
- 3 for intermediate results



optimal strategy



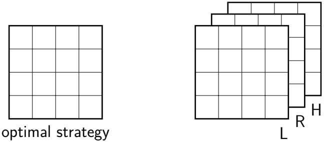
strategy computation (high-level description):

- old (RTED): for each node in one tree a row is allocated and filled

Memory allocation in RTED strategy

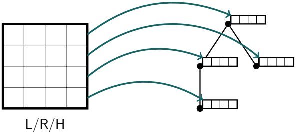
4 matrices of quadratic size are used:

- 1 for optimal strategy
- 3 for intermediate results



strategy computation (high-level description):

- old (RTED): for each node in one tree a row is allocated and filled
- observation: after a node is processed, its row is not needed any more



Early deallocation technique

- new: allocate a row when needed and deallocate not needed rows

Early deallocation technique

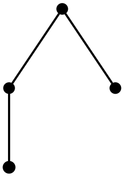
- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v

Early deallocation technique

- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4

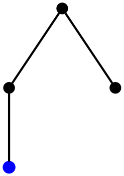
Early deallocation technique

- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



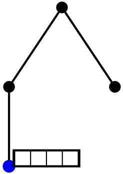
Early deallocation technique

- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



Early deallocation technique

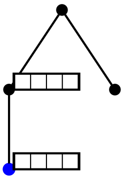
- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



allocate + read

Early deallocation technique

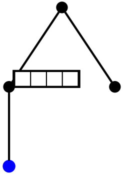
- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



allocate + update

Early deallocation technique

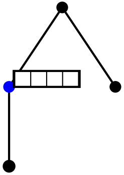
- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



deallocate

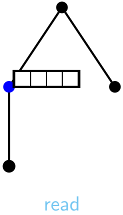
Early deallocation technique

- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



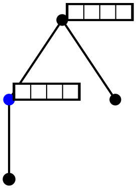
Early deallocation technique

- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



Early deallocation technique

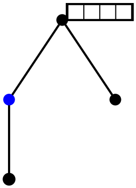
- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



allocate + update

Early deallocation technique

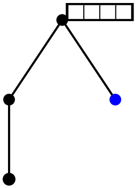
- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



deallocate

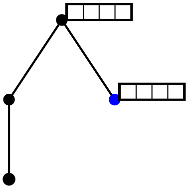
Early deallocation technique

- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



Early deallocation technique

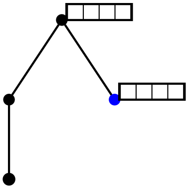
- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



allocate + read

Early deallocation technique

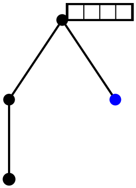
- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



update

Early deallocation technique

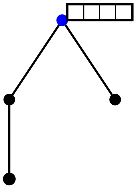
- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



deallocate

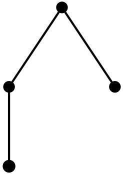
Early deallocation technique

- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4



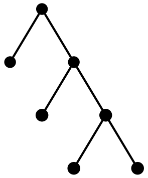
Early deallocation technique

- new: allocate a row when needed and deallocate not needed rows
- for each node v in postorder:
  read row v
  (allocated when v is leaf)
  update row $p(v)$
  (allocated when v is the leftmost child of $p(v)$)
  deallocate row v
- example: 2 rows instead of 4
- What is the max #rows?



Upper bound for the number of rows

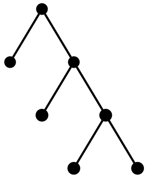
calculate max #rows for postorder:



Upper bound for the number of rows

calculate max #rows for postorder:

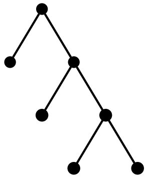
- (1) only leftmost-child leaf nodes add 1 to max #rows



Upper bound for the number of rows

calculate max #rows for postorder:

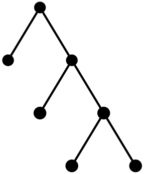
- (1) only leftmost-child leaf nodes add 1 to max #rows
- (2) other nodes do not affect max #rows or reduce it by 1



Upper bound for the number of rows

calculate max #rows for postorder:

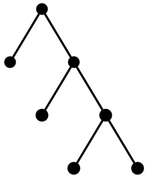
- (1) only leftmost-child leaf nodes add 1 to max #rows
- (2) other nodes do not affect max #rows or reduce it by 1
- there are maximum $n/2$ leftmost-child leaf nodes



Upper bound for the number of rows

calculate max #rows for postorder:

- (1) only leftmost-child leaf nodes add 1 to max #rows
- (2) other nodes do not affect max #rows or reduce it by 1
- there are maximum $n/2$ leftmost-child leaf nodes

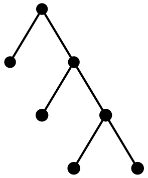


right-to-left postorder

Upper bound for the number of rows

calculate max #rows for postorder:

- (1) only leftmost-child leaf nodes add 1 to max #rows
- (2) other nodes do not affect max #rows or reduce it by 1
- there are maximum $n/2$ leftmost-child leaf nodes



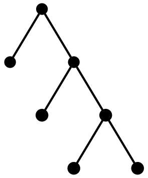
right-to-left postorder

- max #rows = #rightmost-child leaf nodes

Upper bound for the number of rows

calculate max #rows for postorder:

- (1) only leftmost-child leaf nodes add 1 to max #rows
- (2) other nodes do not affect max #rows or reduce it by 1
- there are maximum $n/2$ leftmost-child leaf nodes



right-to-left postorder

- max #rows = #rightmost-child leaf nodes

rule: if #leftmost-child leaves $\leq$ #rightmost-child leaves

- use postorder and right-to-left postorder otherwise

Upper bound for the number of rows

calculate max #rows for postorder:

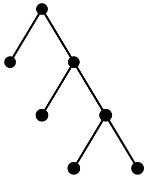
- (1) only leftmost-child leaf nodes add 1 to max #rows
- (2) other nodes do not affect max #rows or reduce it by 1
- there are maximum $n/2$ leftmost-child leaf nodes

right-to-left postorder

- max #rows = #rightmost-child leaf nodes

rule: if #leftmost-child leaves $\leq$ #rightmost-child leaves

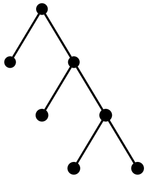
- use postorder and right-to-left postorder otherwise
- max #rows = $n/3$



Upper bound for the number of rows

calculate max #rows for postorder:

- (1) only leftmost-child leaf nodes add 1 to max #rows
- (2) other nodes do not affect max #rows or reduce it by 1
- there are maximum $n/2$ leftmost-child leaf nodes



right-to-left postorder

- max #rows = #rightmost-child leaf nodes

rule: if #leftmost-child leaves $\leq$ #rightmost-child leaves

- use postorder and right-to-left postorder otherwise
- max #rows $= n/3$

	strategy	**L/R/H**	**total**
old (RTED)	n^2	$3n^2$	$4n^2$
new	n^2	$3n\frac{n}{3} = n^2$	$2n^2$

Experiments

dataset	#trees	old (RTED)		new	
		#rows	memory(MB)	#rows	memory(MB)
TreeBank-200	50				
TreeBank-400	10				
SwissProt-1000	20				
SwissProt-2000	10				
TreeFam-400	50				
TreeFam-1000	20				

Experiments

dataset	#trees	old (RTED)		new	
		#rows	memory(MB)	#rows	memory(MB)
TreeBank-200	50	195.4			
TreeBank-400	10	371.3			
SwissProt-1000	20	987.5			
SwissProt-2000	10	1960.1			
TreeFam-400	50	402.6			
TreeFam-1000	20	981.9			

Experiments

dataset	#trees	old (RTED)		new	
		#rows	memory(MB)	#rows	memory(MB)
TreeBank-200	50	195.4		6.0	
TreeBank-400	10	371.3		6.3	
SwissProt-1000	20	987.5		3.0	
SwissProt-2000	10	1960.1		3.0	
TreeFam-400	50	402.6		12.3	
TreeFam-1000	20	981.9		14.1	

Experiments

dataset	#trees	old (RTED)		new	
		#rows	memory(MB)	#rows	memory(MB)
TreeBank-200	50	195.4	1.10	6.0	
TreeBank-400	10	371.3	2.86	6.3	
SwissProt-1000	20	987.5	16.64	3.0	
SwissProt-2000	10	1960.1	63.20	3.0	
TreeFam-400	50	402.6	3.33	12.3	
TreeFam-1000	20	981.9	16.73	14.1	

Experiments

dataset	#trees	old (RTED)		new	
		#rows	**memory(MB)**	**#rows**	**memory(MB)**
TreeBank-200	50	195.4	1.10	6.0	0.72
TreeBank-400	10	371.3	2.86	6.3	1.30
SwissProt-1000	20	987.5	16.64	3.0	5.13
SwissProt-2000	10	1960.1	63.20	3.0	17.80
TreeFam-400	50	402.6	3.33	12.3	1.46
TreeFam-1000	20	981.9	16.73	14.1	5.58

Conclusion

- RTED is the best TED algorithm due to its optimal strategy

Conclusion

- RTED is the best TED algorithm due to its optimal strategy
- strategy computation is the memory bottleneck in RTED

Conclusion

- RTED is the best TED algorithm due to its optimal strategy
- strategy computation is the memory bottleneck in RTED
- early deallocation technique solves the memory problem

Future work

- classical TED approaches are infeasible for large inputs

  e.g., trees of 1.000.000 nodes may require 1TB and 100h

Future work

- classical TED approaches are infeasible for large inputs

  e.g., trees of 1.000.000 nodes may require 1TB and 100h

- there is a need for other, better solutions

  e.g., efficient pruning of the search space