Quantitative Dependency Syntax with the Linear Arrangement Library (LAL). An introduction.

Lluís Alemany-Puig & Ramon Ferrer-i-Cancho

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Department of Computer Science
Institute of Mathematics of UPC-Barcelona Teach (IMTech)
Universitat Politecnica de Catalunya
Barcelona, Catalonia

IMTech

June 1, 3 and 4, 2021
Barcelona-Hangzhou-Guangzhou-Beijing-Dalian-Ningbo-Guilin-Changsha.

The team

Lluís Alemany-Puig
Main developer and designer

Juan Luis Esteban
Contributor

Ramon Ferrer-i-Cancho
Designer and principal investigator

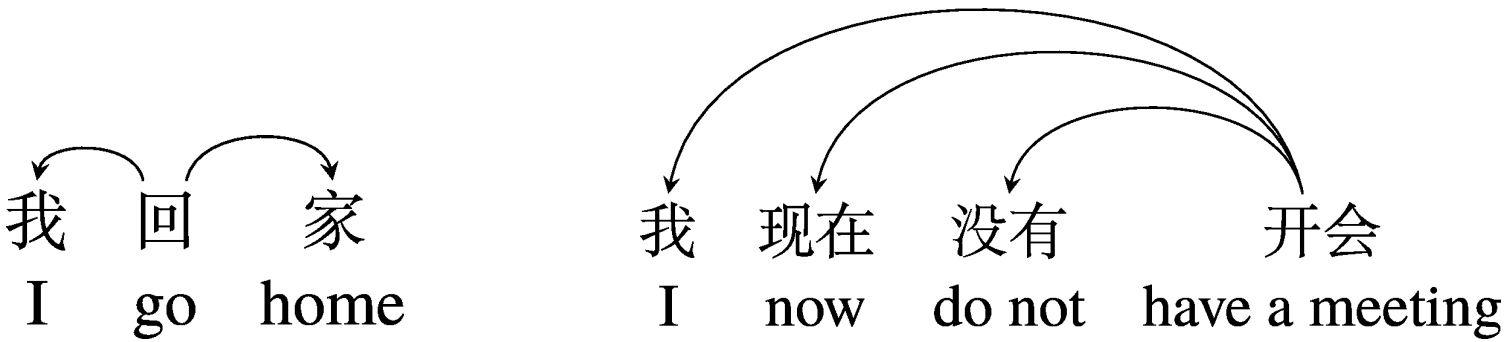
Many thanks to Haitao Liu and Jianwei Yan!

Dependency syntax

Dependency grammar (versus phrase structure grammar).

Syntactic dependency structure:

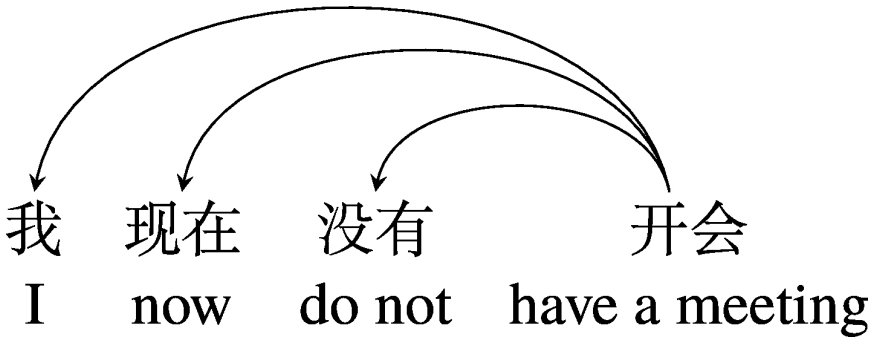
- Tree (head-dependent arcs)
- Linear arrangement (linear ordering)
- Labels (words, type of dependencies,...).



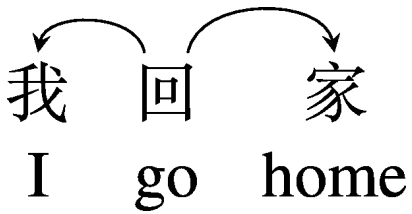
What can you do with LAL? I

Quantitative dependency syntax: dependency syntax + quantitative linguistics

Measuring distance: Euclidean distance versus topological distance.



$\mathsf{MDD} = (1+2+3)/3 = 2,$
$\mathsf{MHD} = (1+1+1)/3,\ \Omega = -1,\ \mathsf{HI} = 0,$
$\langle k^2 \rangle = 12/4 = 3,\ h = 1$



$\mathsf{MDD} = 1,\ \mathsf{MHD} = (1+1)/2,\ \Omega = 1,$
$\mathsf{HI} = 1/2\ ,\ \langle k^2 \rangle = 2,\ h = ...$

Calculate measures (and scores) on syntactic dependency structures

- Structural measures (**do not** depend on linear ordering): mean hearchical distance (MHD), $\langle k^2 \rangle$, hubiness (h),..)
- Linear ordering measures (**do** depend on the linear ordering): proportion of head initial dependencies (HI), mean dependency distance (MDD), sum of dependency distances (D)...brand new Ω.

What can you do with LAL? II

Quantitative dependency syntax research on

- Single syntactic dependency structures
- Treebanks (collections of these structures, usually from the same language). Examples: ...
- Collections of treebanks (collections of these syntactic dependency structures). Examples: Universal Dependencies collection (more than 100 languages from 13 linguistic families).

Parallel treebanks: a crucial resource.

Is LAL just about computing measures?

Hypothesis testing I

Quantitative linguistics (quantitative dependency syntax): measuring/counting + **hypothesis testing** + theory construction + philosophy of science + ...

Syntactic dependency structure: tree + **linear arrangement** + ...

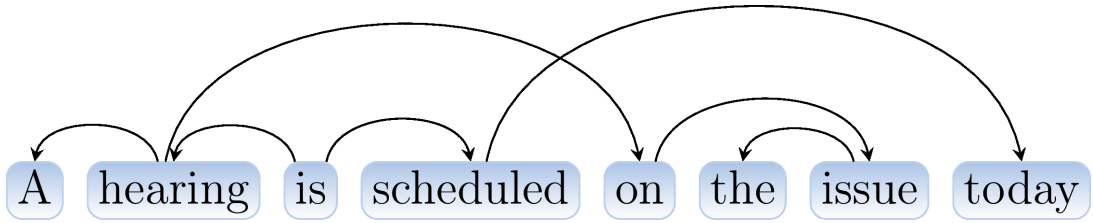
Research question 1. Is MDD significantly low in a certain sentence?

- Motivation: cognitive pressure to reduce MDD in sentences (DDm principle).
- Implementation: keep the tree fixed and consider (random) shufflings.
- Random baseline: random shuffling (random linear arrangement)

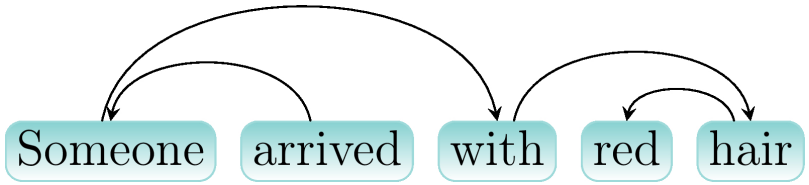
我回家, 我家回,
回我家, 回家我,
家我回, 家回我

Formal constraints on linear order

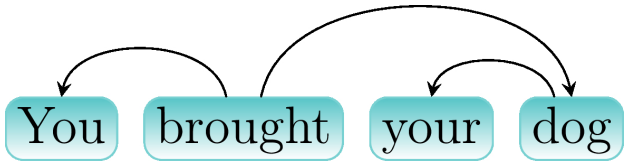
Syntactic dependency structure: **tree** + **linear arrangement** + labels



Planarity: no edge crossings.



Projectivity: planarity + the root is not covered.



Three kinds of random shufflings:

- Unconstrained: all $n!$ possible shufflings are valid.
- Planar: only planar orderings are allowed
- Projective: only projective orderings are allowed.

$\rightarrow$ 3 variants of the statistical test

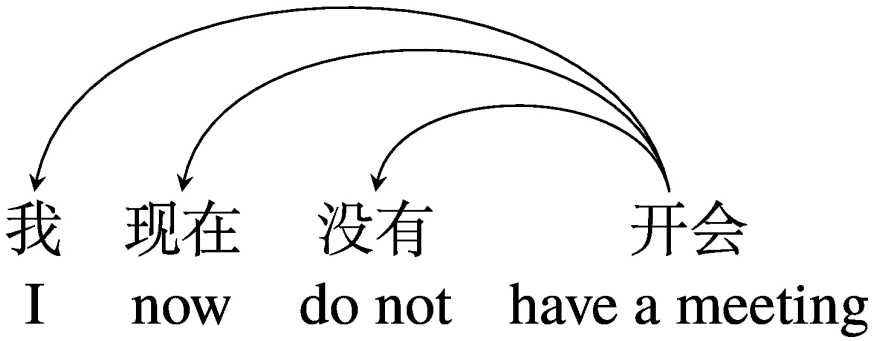
Hypothesis testing II

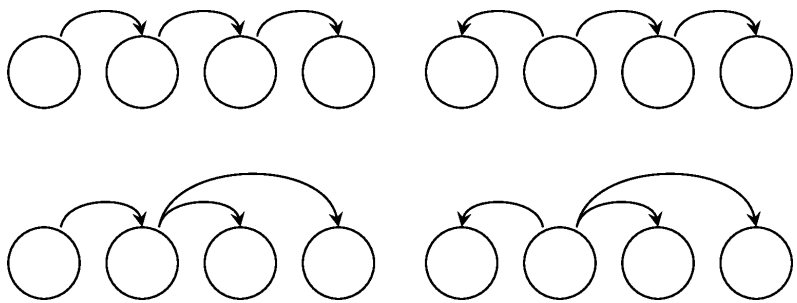
Syntactic dependency structure: **tree** + linear arrangement + ...

Research question 2. Is MHD significantly low in a real sentence?

- Hypothesis: cognitive pressure to reduce MHD in sentences?
- Implementation: consider random trees from the ensemble of all possible trees (linear order is irrelevant).
- Baseline: random tree

Same approach for any linear ordering score.





The power of LAL

Easy without LAL:

- Calculate real values of measures/scores.

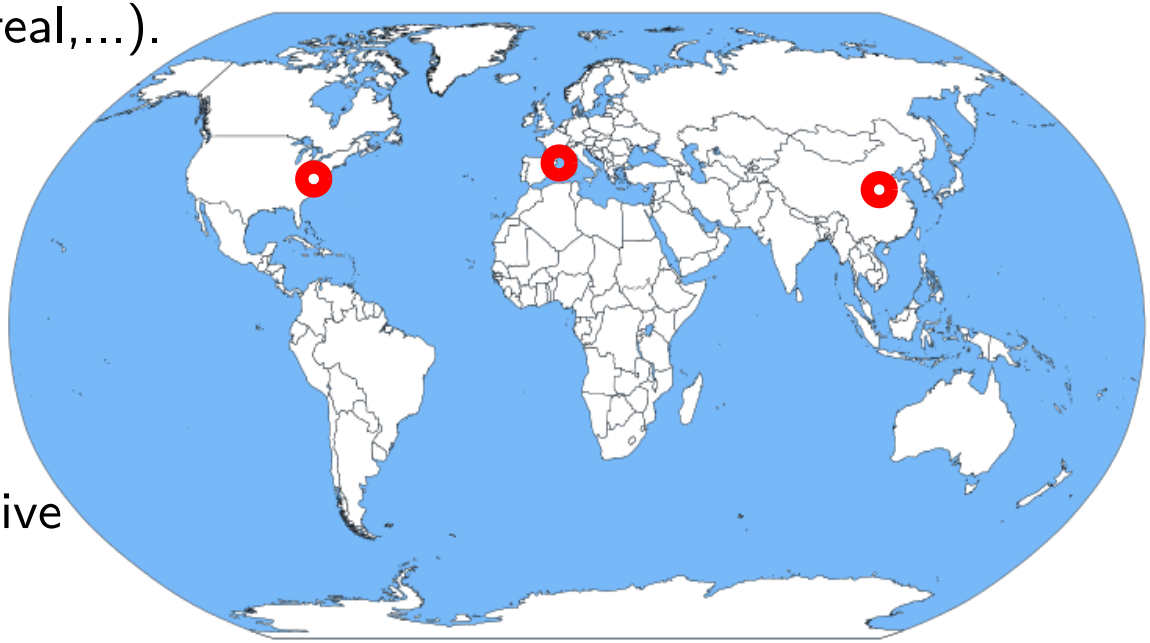
Not that easy without LAL:

- Perform statistical tests.
- Calculate theoretical values of optimality scores according to baselines:
  - Expected value (random baselines). Superfast exact algorithm (e.g., D_{rla}) (backup: Monte Carlo estimation).
  - Minimum value. Superfast exact algorithm (e.g., D_{min}) (backup: simple brute force).
  - *Linear order scores.* Different formal constraints (unconstrained, planar or projective).
  - *Structural scores.* Different kinds of trees and different criteria to determine equality.

The design principles of LAL: uniting ...

Different traditions in language research:

- Asia. China (Haitao Liu's), Japan
- Europe. France (Kim Gerdes and Sylvaine Kahane's: flux weight, surface syntactic universal universal dependencies. Catalonia: ...
- America: USA (Gibson, Futrell, Temperley, Gildea,...: unaveraged dependency distances, formal constraints as real,...).



Different research fields:

- Language research (typology, dependency linguistics, quantitative linguistics, cognitive science).
- Computer science (algorithmics).
- Discrete mathematics and graph theory.

Design principles of LAL: ease of use

```
import laldebug as lal
err = lal.io.process_treebank("Cantonese.txt", "output_file.txt")
print(err)
```

- Python (but also C++) as programming language.
- Scripting (rather than hard programming) as in R.
- A few lines of code in many cases. Many functions to simplify common tasks.
- Gradual introduction to programming (as in this course).

More design principles of LAL

Speed

- The fastest algorithms (e.g., D_{min}).
- Internal parallelization.
- All the scores in LAL on the whole UD collection in < 2min on a laptop.
- Use C++ if you like ;-).

Reliability

- Thoroughly tested. Exhaustive testing. Transfer of knowledge from Esteban & Ferrer-i-Cancho (2017), Alemany-Puig, Mora & Ferrer-i-Cancho (2021).
- Critical algorithms tested for all trees up to n vertices.
- Two branches of the project: library + testing of the library.
- Think twice when borrowing somebody else's code.

Open source (coming out soon!)

Overview of this minicourse

Session 1 (today).

- Introduction to quantitative dependency syntax.
- The design principles of LAL.
- **Working with single trees.**
- **Guided & non-guided exercises.**

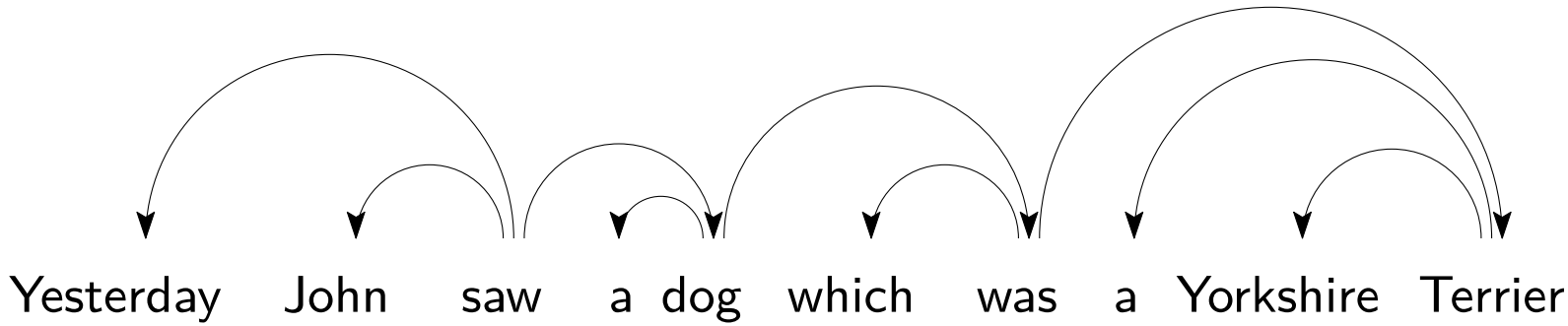
Session 2

- Working with single trebanks and treebank collections.
- Guided exercises.
- Random & exhaustive generation of trees.
- Guided & non-guided exercises.

Session 3

- Continuation of guided & non-guided exercises.
- Random & exhaustive generation of linear arrangements.
- Guided & non-guided exercises.

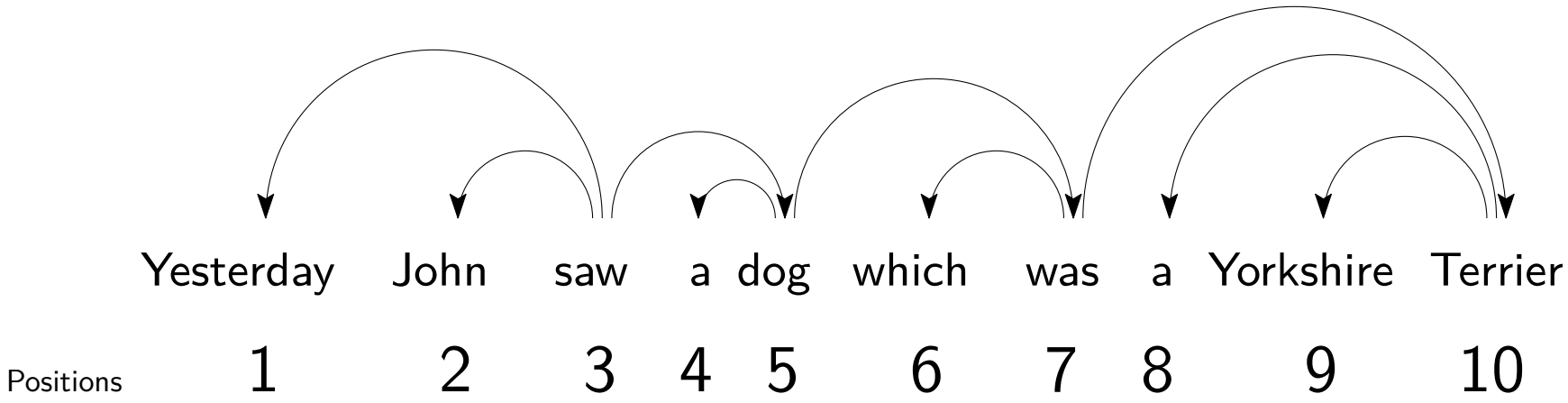
Library representation of syntactic dependencies



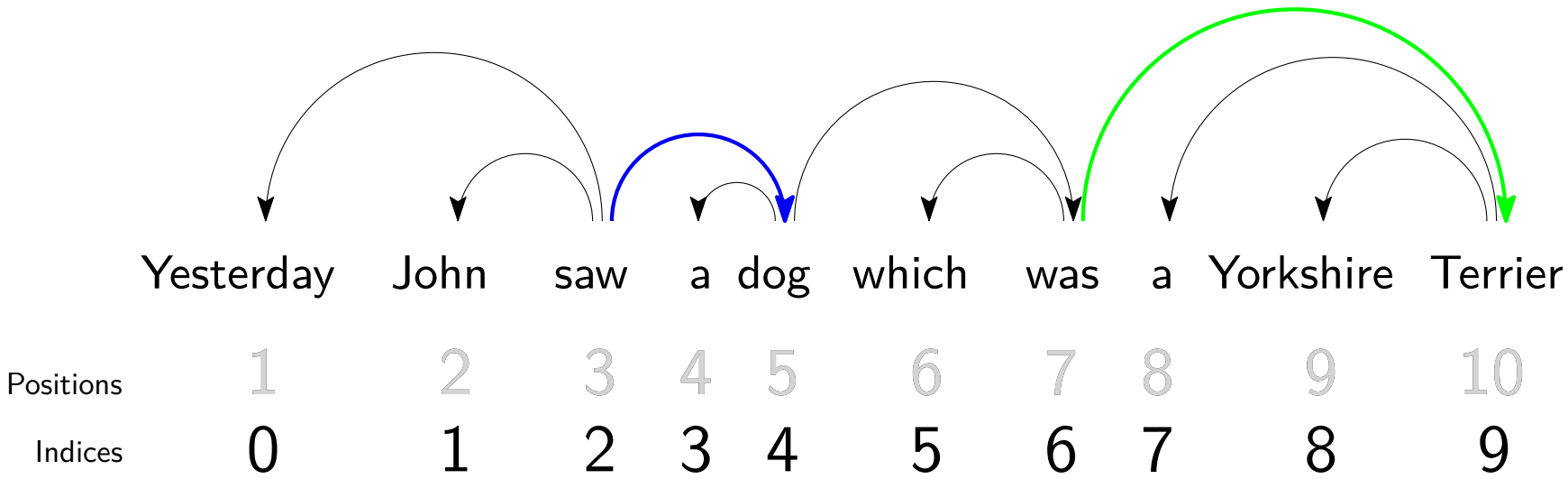
head/parent $\longrightarrow$ dependent

saw $\longrightarrow$ dog
was $\longrightarrow$ Terrier
⋮

Library representation of syntactic dependencies

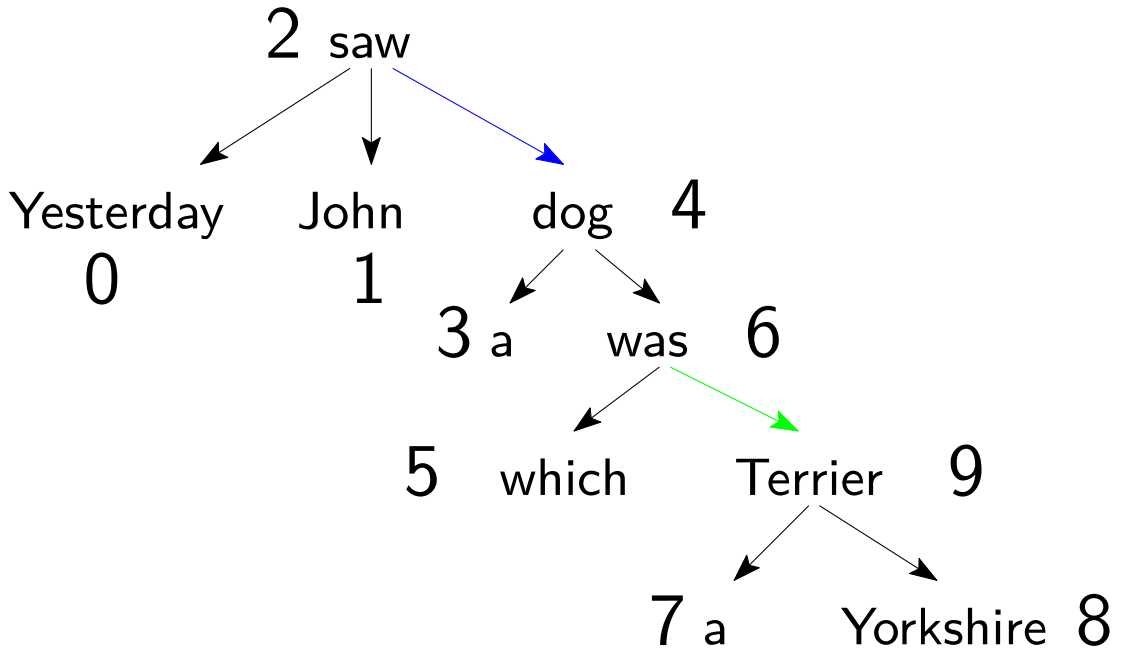


Library representation of syntactic dependencies

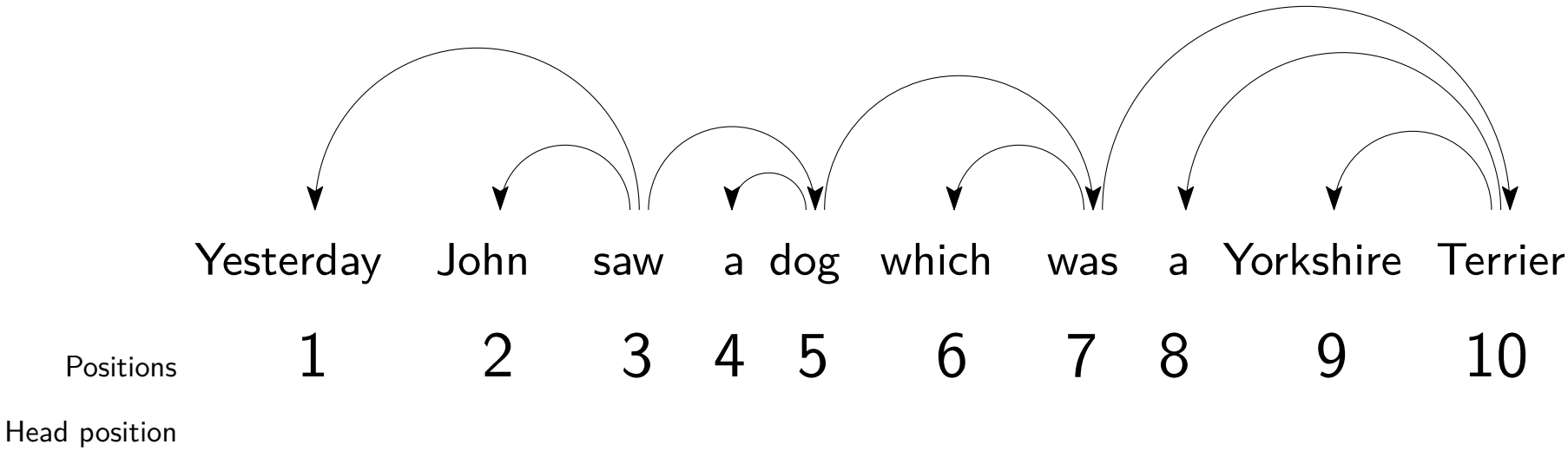


Edge list

2 0
2 1
2 4
4 3
4 6
6 5
6 9
9 7
9 8

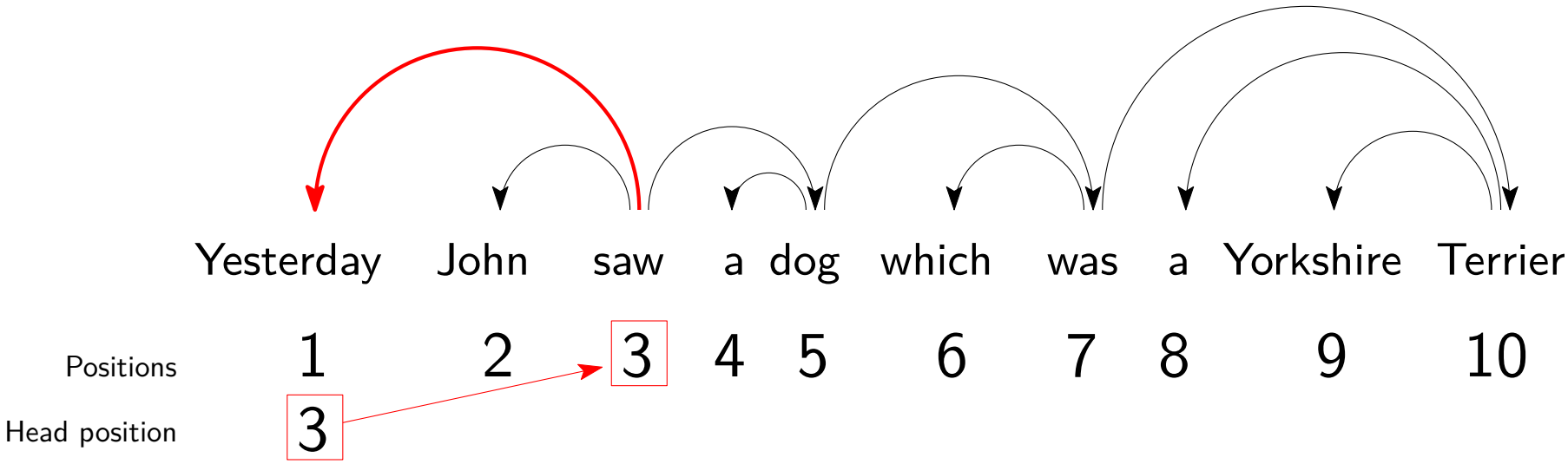


Library representation of syntactic dependencies



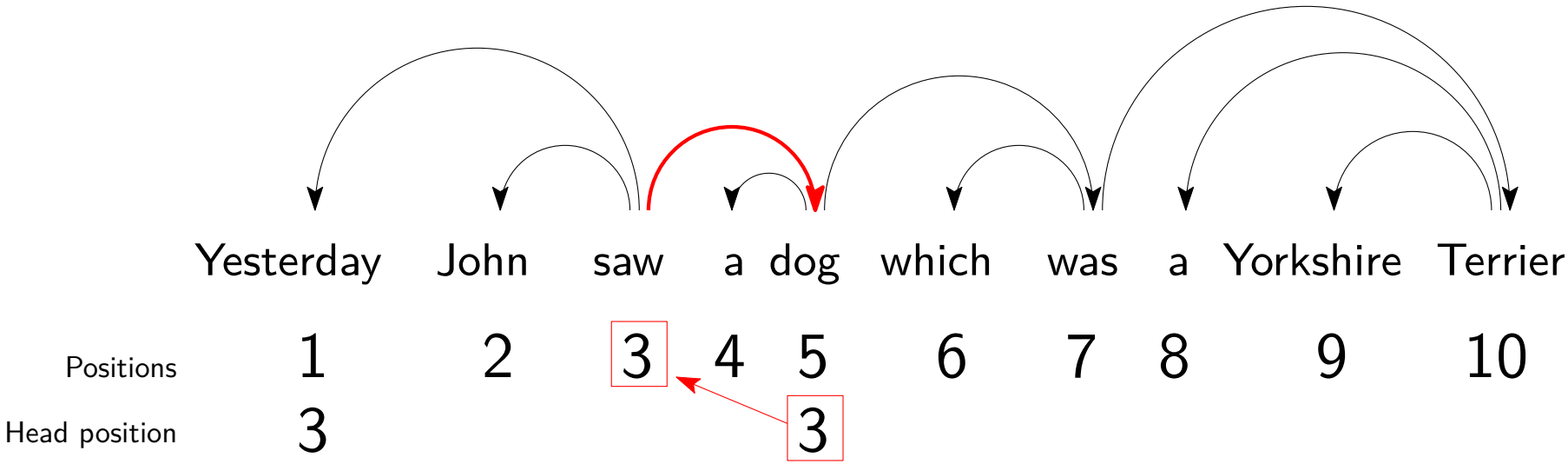
Head vector

Library representation of syntactic dependencies



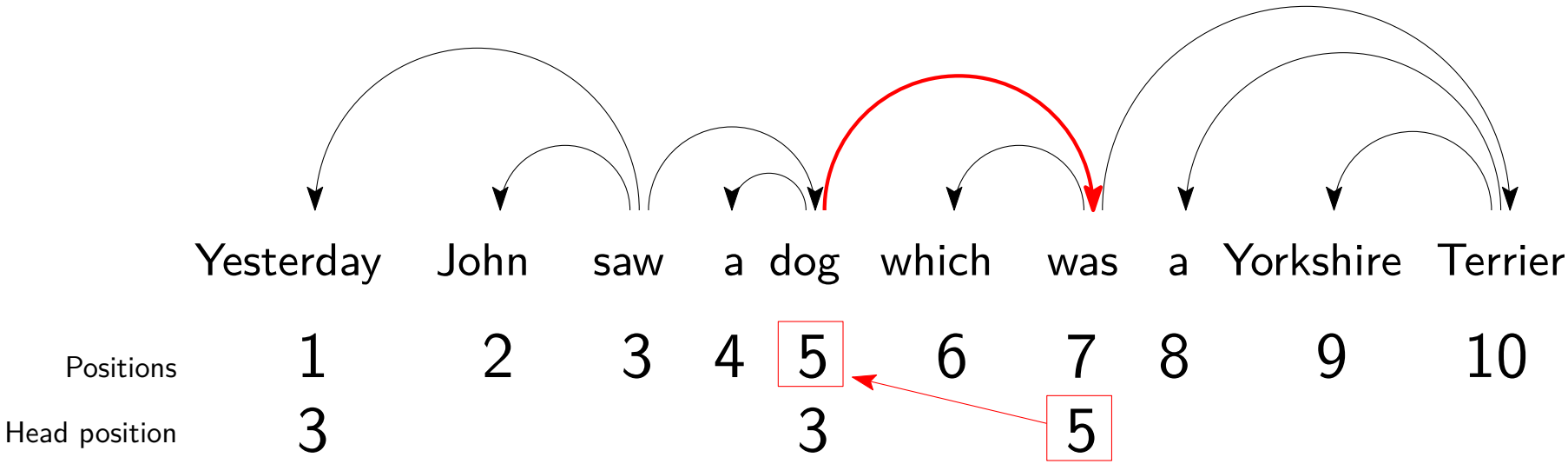
Head vector [3,

Library representation of syntactic dependencies



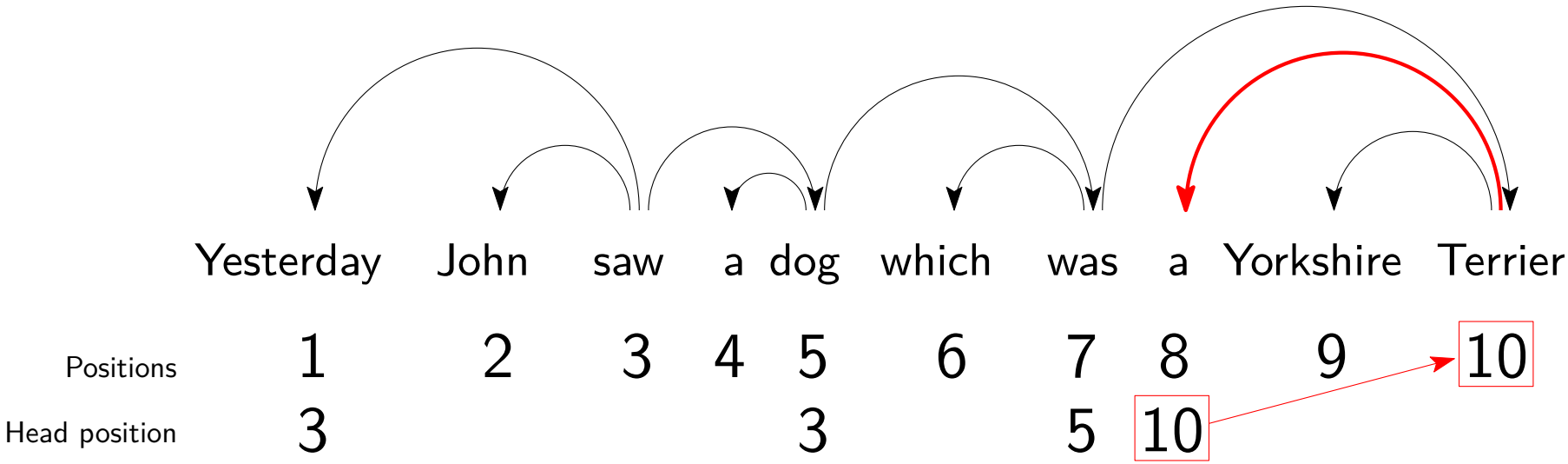
Head vector [3,-,-,-,3,

Library representation of syntactic dependencies



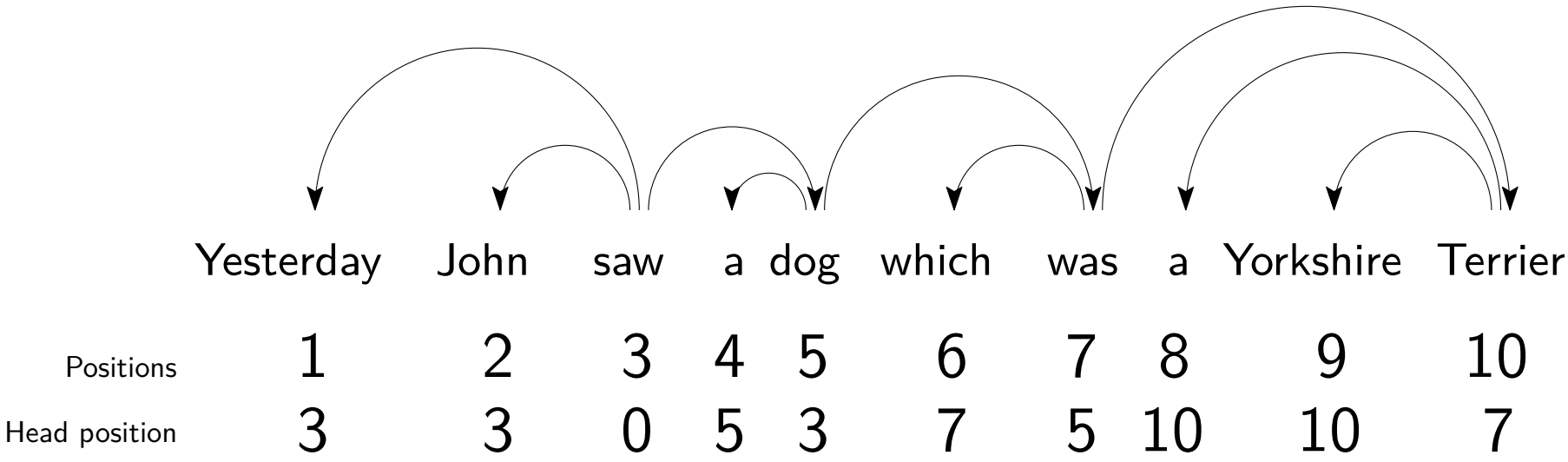
Head vector [3,-,-,-,3,-,5,

Library representation of syntactic dependencies



Head vector [3,-,-,-,3,-,5,10,

Library representation of syntactic dependencies



Head vector [3,3,0,5,3,7,5,10,10,7]

Reading dependency structures from disk

Read an edge list

```
import laldebug as lal
rt = lal.io.read_edge_list("rooted_tree", "path/to/edge_list.txt")
```

the type of tree to be constructed

file containing the edge list

contents of the file

```
2 0
2 1
2 4
4 3
4 6
6 5
6 9
9 7
9 8
```

Reading dependency structures from disk

Read an edge list

```
import laldebug as lal
rt = lal.io.read_edge_list("rooted_tree", "path/to/edge_list.txt")
```

the type of tree to be constructed

file containing the edge list

contents of the file

```
2 0
2 1
2 4
4 3
4 6
6 5
6 9
9 7
9 8
```

Read a head vector

```
import laldebug as lal
rt = lal.io.read_head_vector("rooted_tree", "path/to/head_vector.txt")
```

the type of tree to be constructed

file containing the head vector

```
3 3 0 5 3 7 5 10 10 7
```

contents of the file

A head vector must be written in a single line

Creating dependency structures manually

From an edge list

```
import laldebug as lal
rt = lal.graphs.rooted_tree(10)
rt.set_edges([(2,0),(2,1),(2,4),(4,3),(4,6),(6,5),(6,9),(9,7),(9,8)])
print(rt)
print(rt.get_edges())
```

From a head vector

```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
print(rt)
print(rt.get_edges())
```

Creating dependency structures manually

From an edge list

```
import laldebug as lal
rt = lal.graphs.rooted_tree(10)
rt.set_edges([(2,0),(2,1),(2,4),(4,3),(4,6),(6,5),(6,9),(9,7),(9,8)])
print(rt)
print(rt.get_edges())
```

The number of vertices must agree with the edge list!

From a head vector

```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
print(rt)
print(rt.get_edges())
```

Checking correctness of trees

Out-neighbours of every vertex (dependent word)

The asterisk denotes the root

In-neighbours of every vertex (parent/head word)

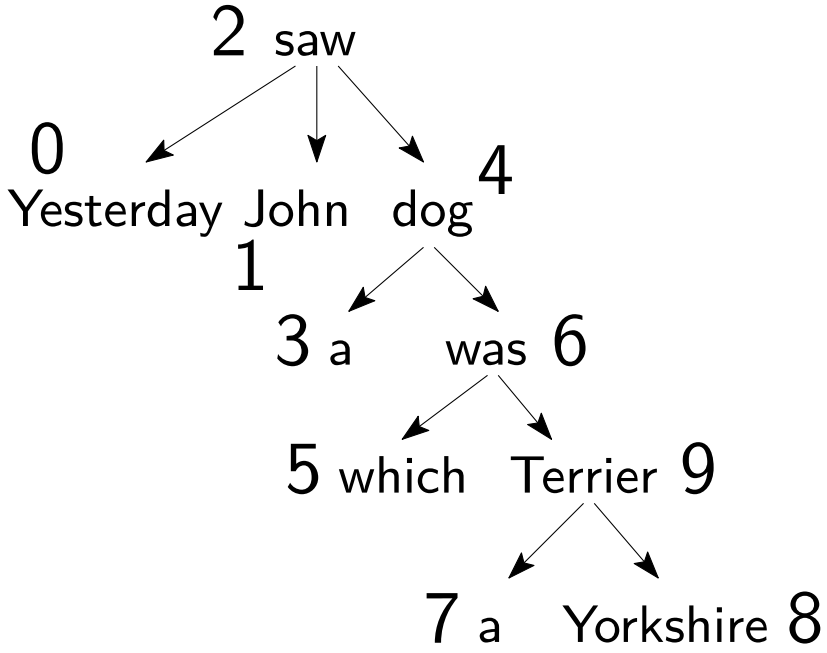
The root shouldn't have in-neighbours (parent/head word)

```
print(rt)
```

```
out:
 0:
 1:
*2: 0 1 4
 3:
 4: 3 6
 5:
 6: 5 9
 7:
 8:
 9: 7 8
in:
 0: 2
 1: 2
*2:
 3: 4
 4: 2
 5: 6
 6: 4
 7: 9
 8: 9
 9: 6
```

```
print(rt.get_edges())
```

```
((2, 0),
 (2, 1),
 (2, 4),
 (4, 3),
 (4, 6),
 (6, 5),
 (6, 9),
 (9, 7),
 (9, 8))
```



Checking correctness of trees

Out-neighbours of every vertex (dependent word)

The asterisk denotes the root

In-neighbours of every vertex (parent/head word)

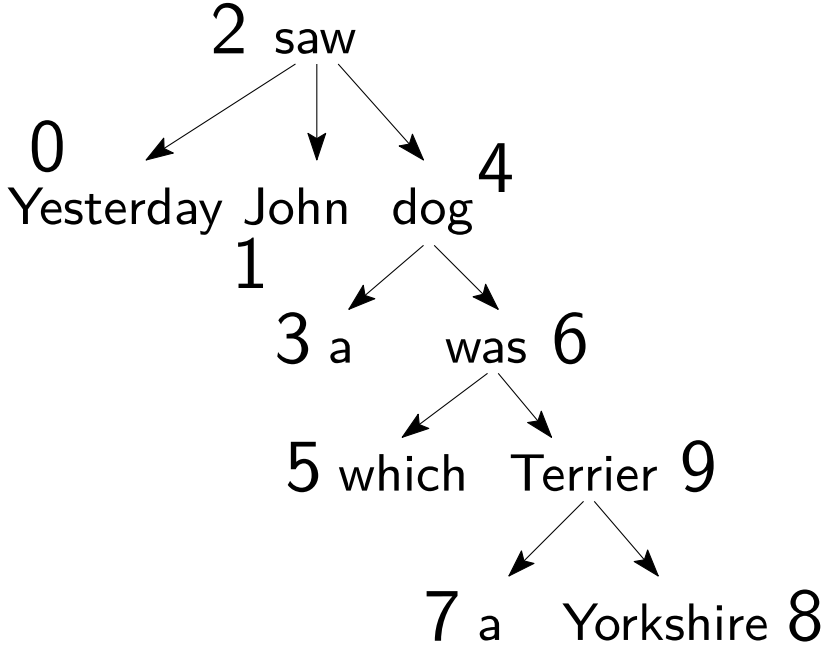
The root shouldn't have in-neighbours (parent/head word)

```
print(rt)
```

```
out:
 0:
 1:
*2: 0 1 4
 3:
 4: 3 6
 5:
 6: 5 9
 7:
 8:
 9: 7 8
in:
 0: 2
 1: 2
*2:
 3: 4
 4: 2
 5: 6
 6: 4
 7: 9
 8: 9
 9: 6
```

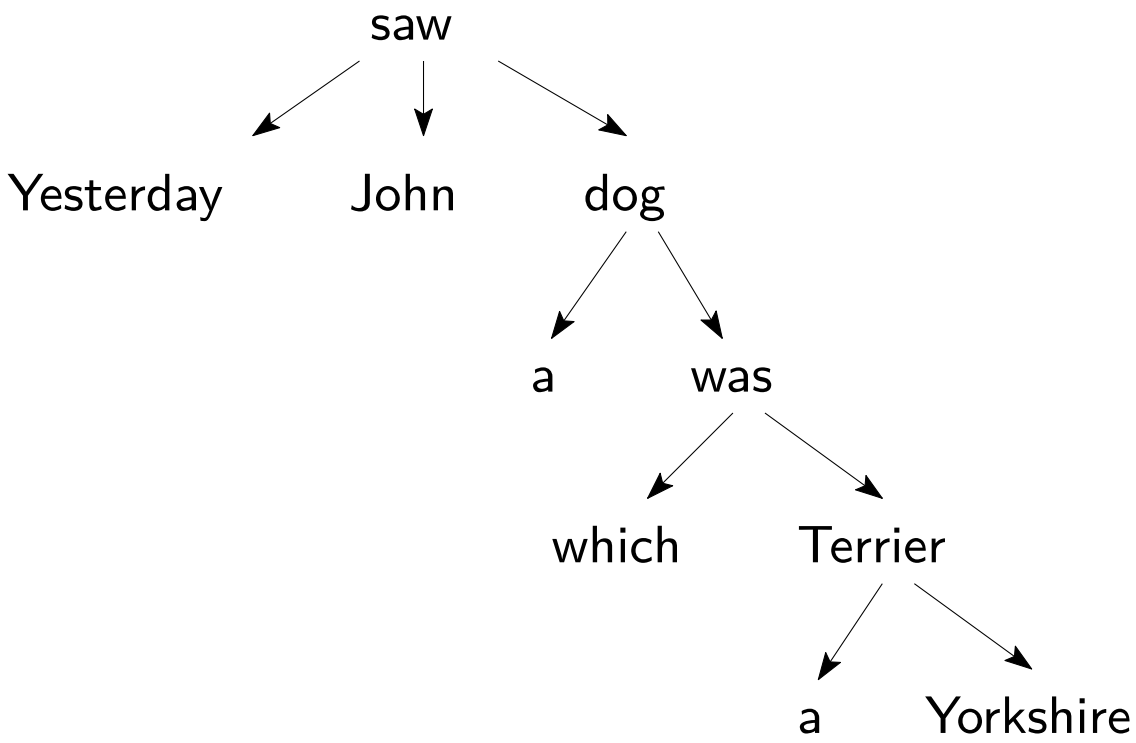
```
print(rt.get_edges())
```

```
(2, 0),
(2, 1),
(2, 4),
(4, 3),
(4, 6),
(6, 5),
(6, 9),
(9, 7),
(9, 8))
```



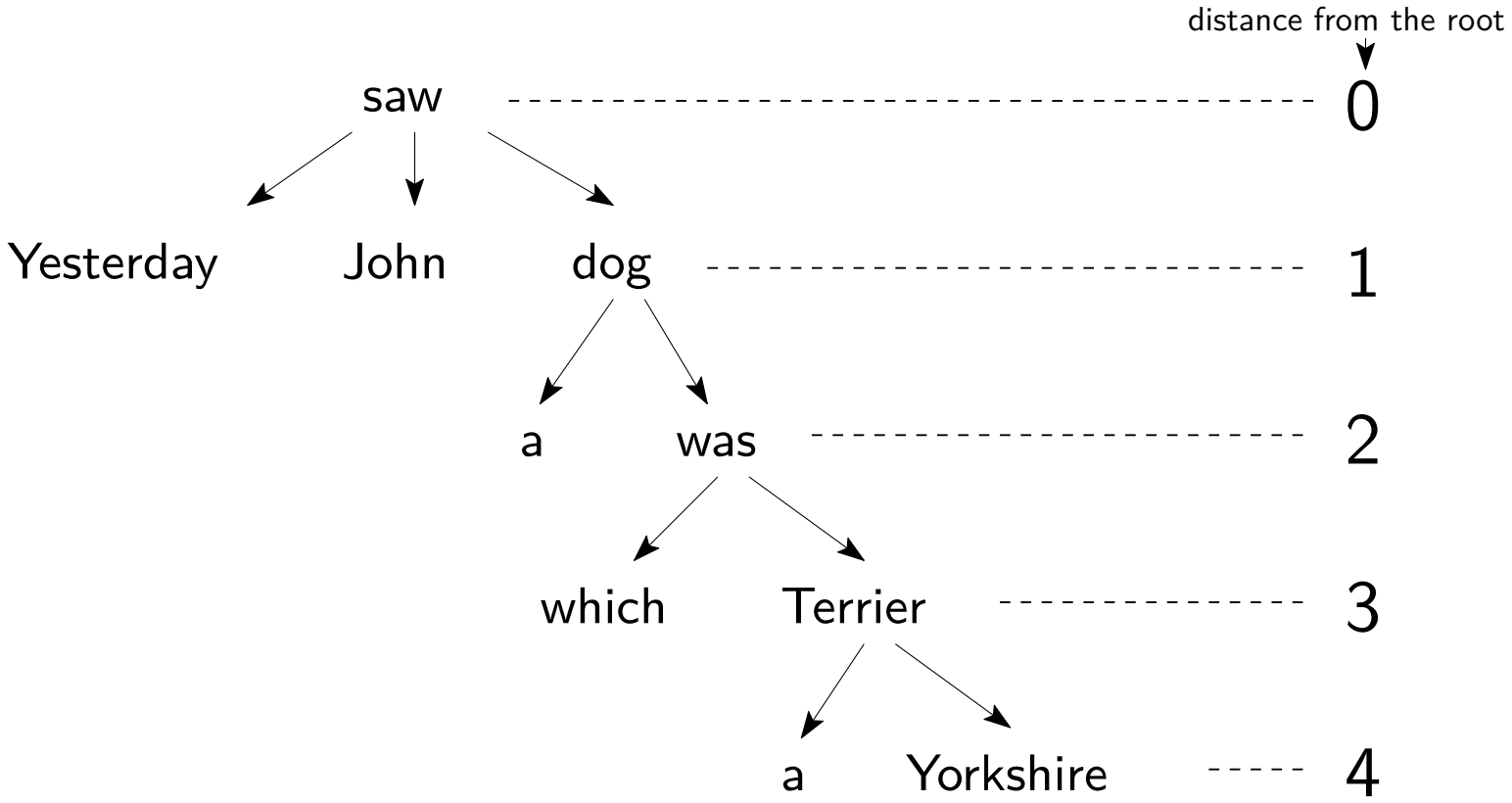
A structural property on dependency structures

Mean hierarchical distance (MHD) (Jing & Liu, 2015)



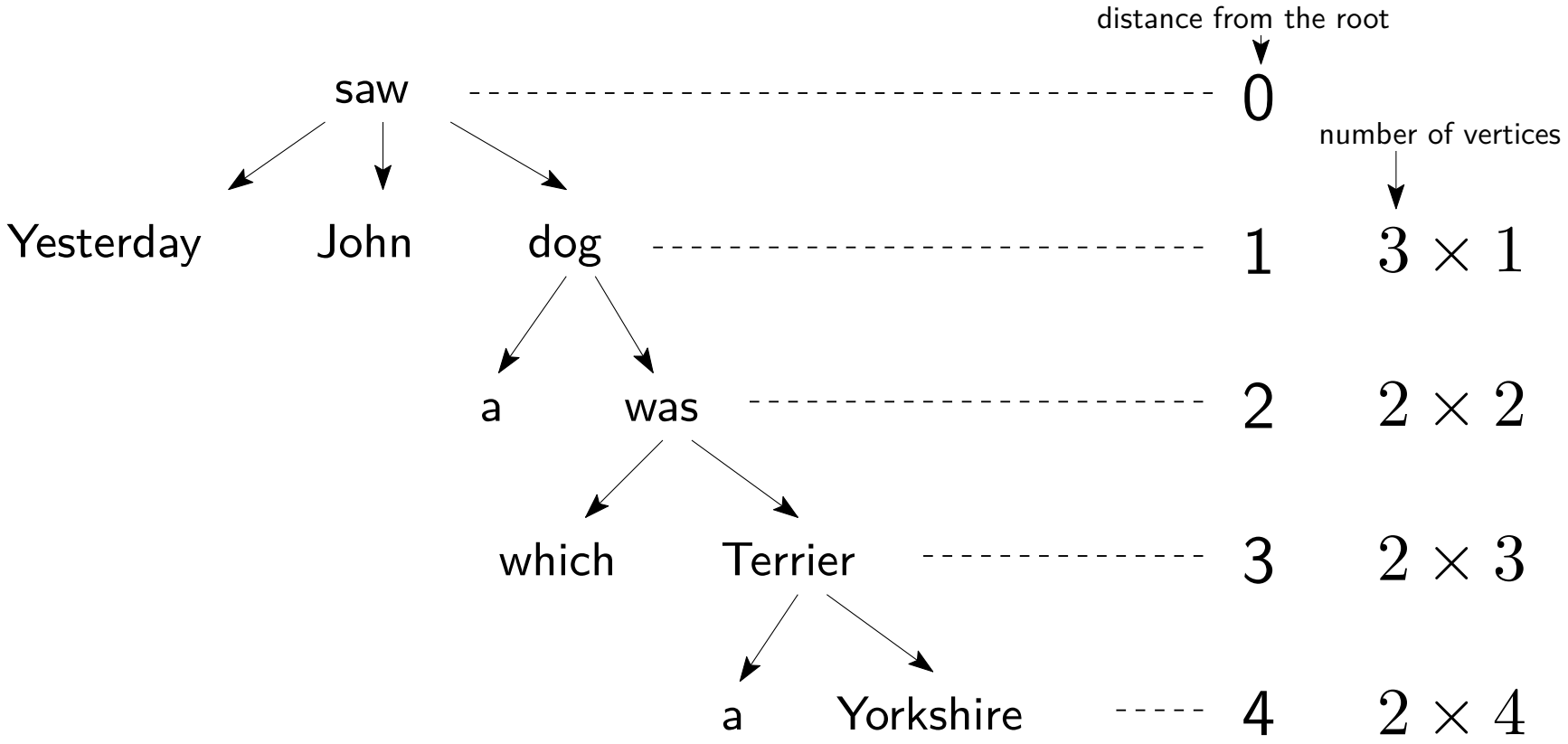
A structural property on dependency structures

Mean hierarchical distance (MHD) (Jing & Liu, 2015)



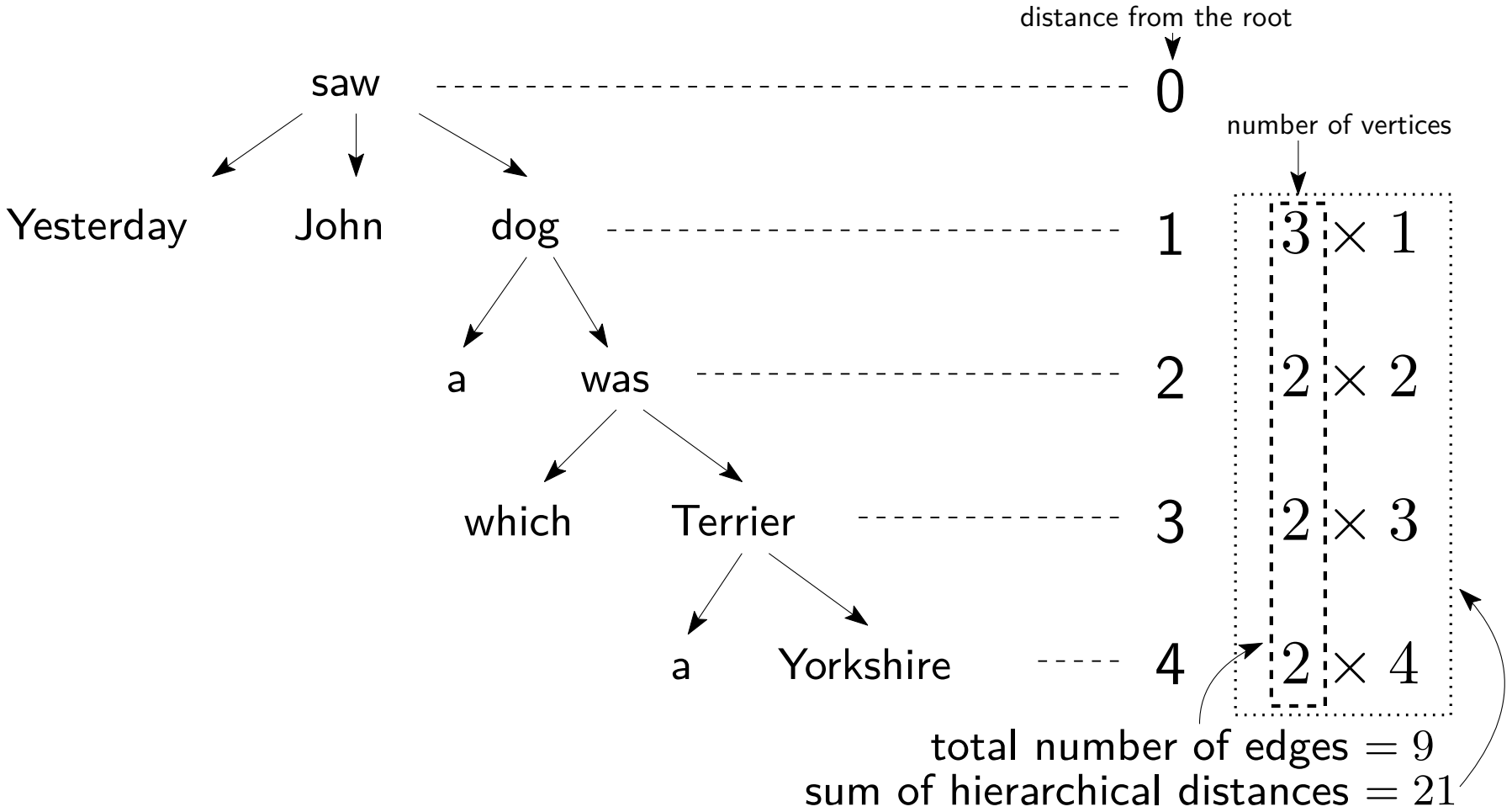
A structural property on dependency structures

Mean hierarchical distance (MHD) (Jing & Liu, 2015)



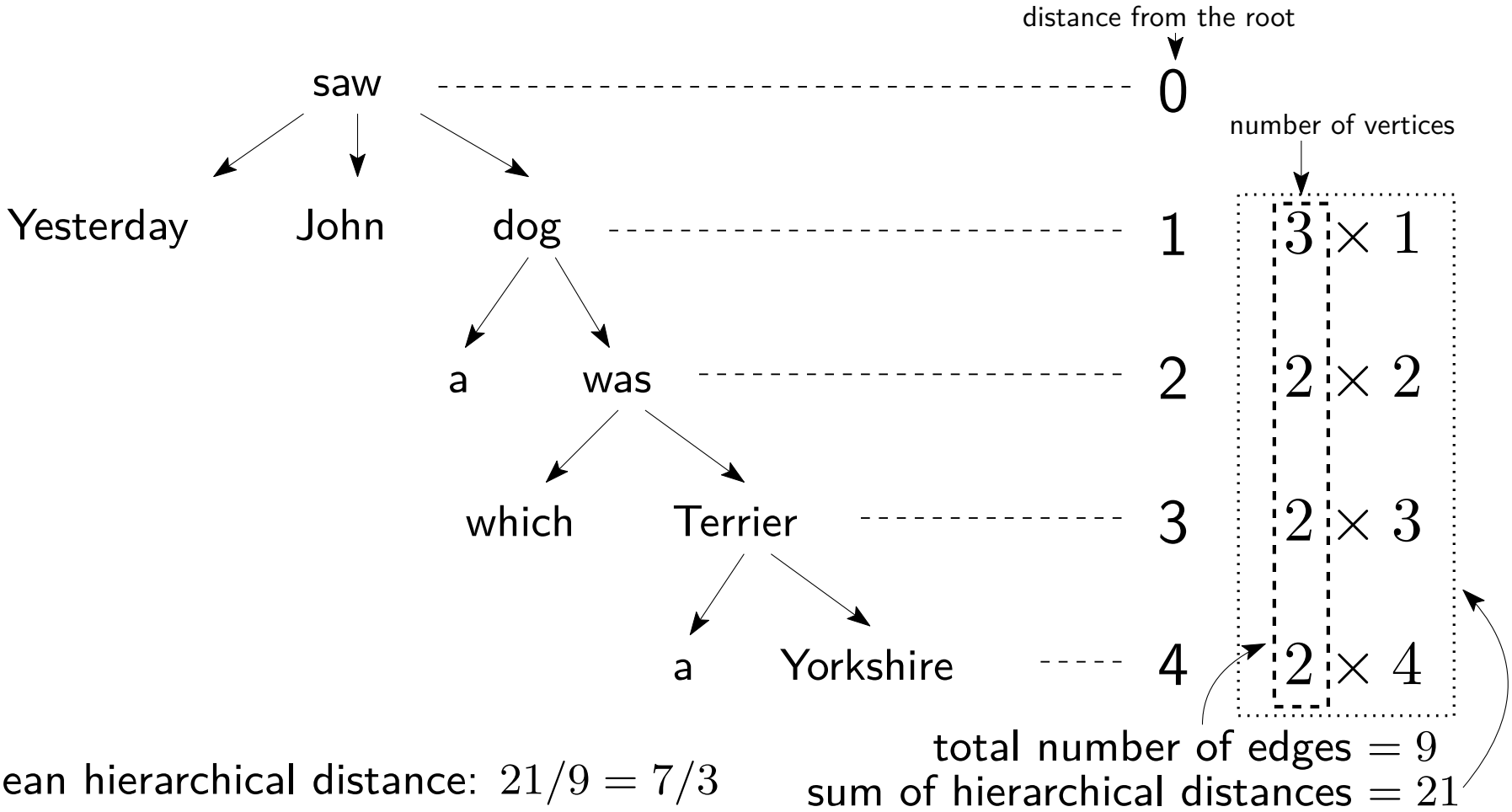
A structural property on dependency structures

Mean hierarchical distance (MHD) (Jing & Liu, 2015)



A structural property on dependency structures

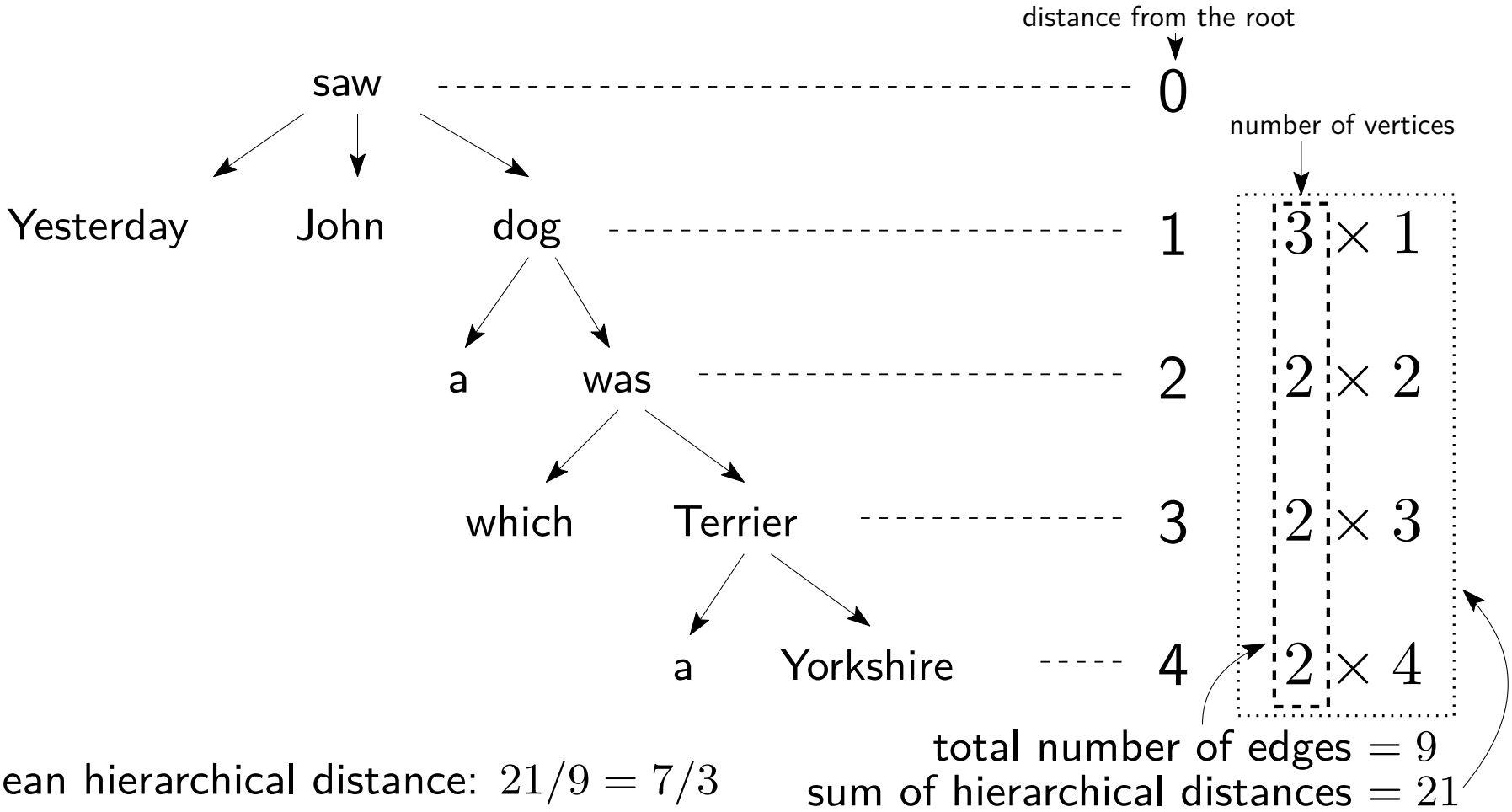
Mean hierarchical distance (MHD) (Jing & Liu, 2015)



Mean hierarchical distance: $21/9 = 7/3$

A structural property on dependency structures

Mean hierarchical distance (MHD) (Jing & Liu, 2015)



Mean hierarchical distance: $21/9 = 7/3$

```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
mhd = lal.properties.mean_hierarchical_distance(rt)
```

Linear arrangements in the library

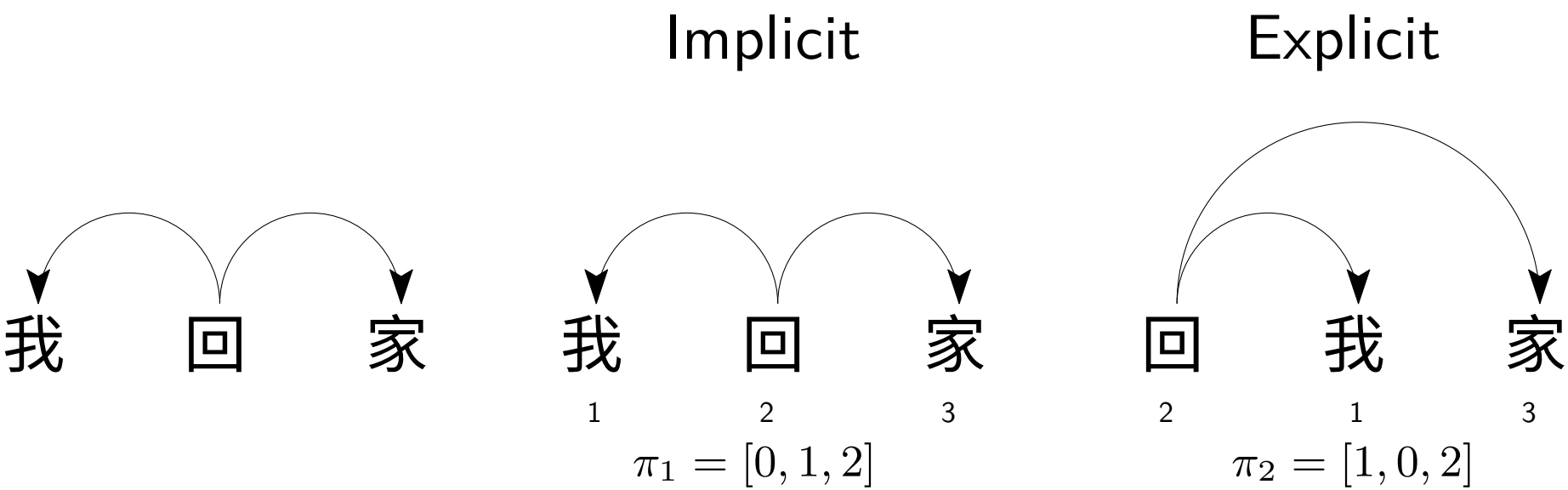
A linear arrangement indicates the position that every word should occupy in the sentence. It can represent **any** shuffling of the words, so there are $n!$

$\pi = [3, 5, 1, 2, 0, 4]$: vertex 0 goes to position 3, vertex 1 goes to position 5, ...

We use the convention that the positions indicated by the arrangement are between 0 and $n-1$ (n is the number of words). Head vectors use numbers from 1 to n.

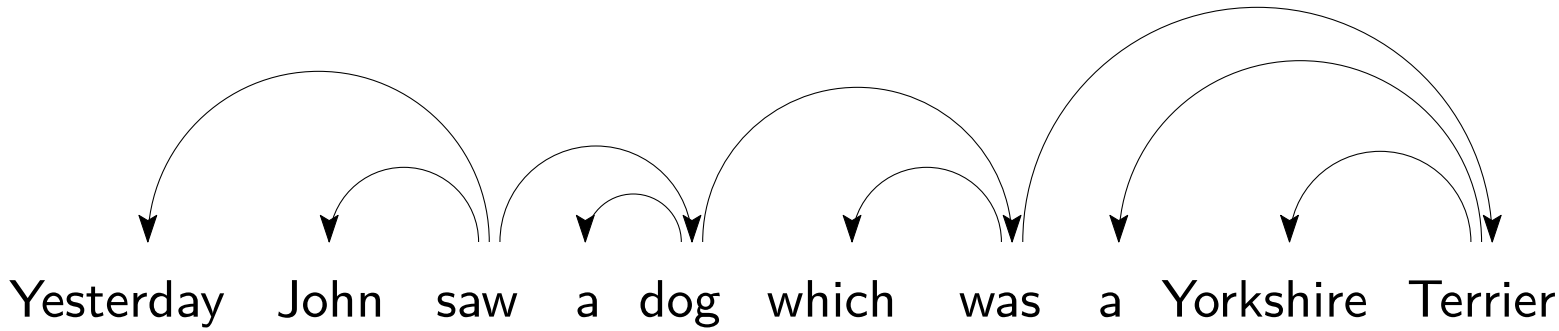
We will see in the following slides how to use linear arrangements

- Implicitly: where the arrangement is assumed to be $[0, 1, \cdots, n-2, n-1]$.
- Explicitly: a linear arrangement of our choice



Calculating arrangement-dependent metrics

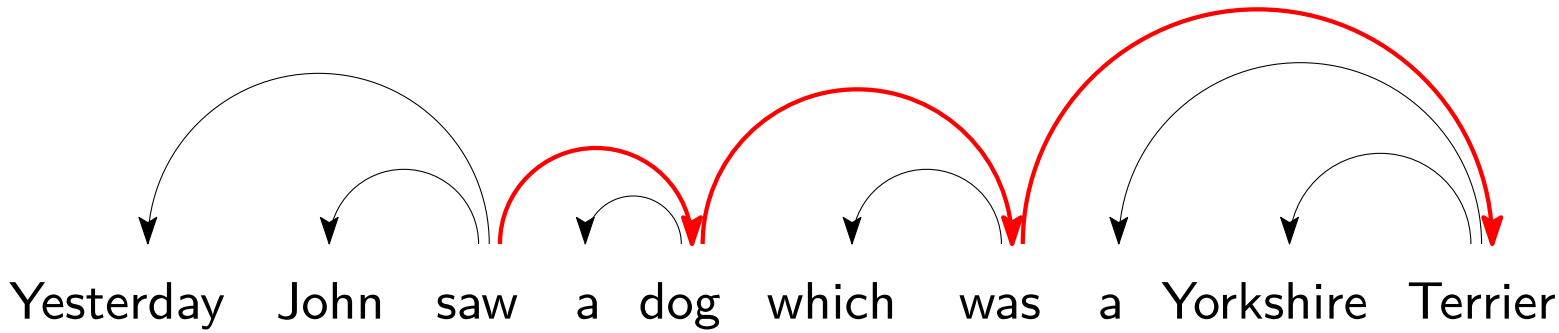
Head initial dependencies (Liu, 2010)



Head *initial* dependencies: head before dependent

Calculating arrangement-dependent metrics

Head initial dependencies (Liu, 2010)

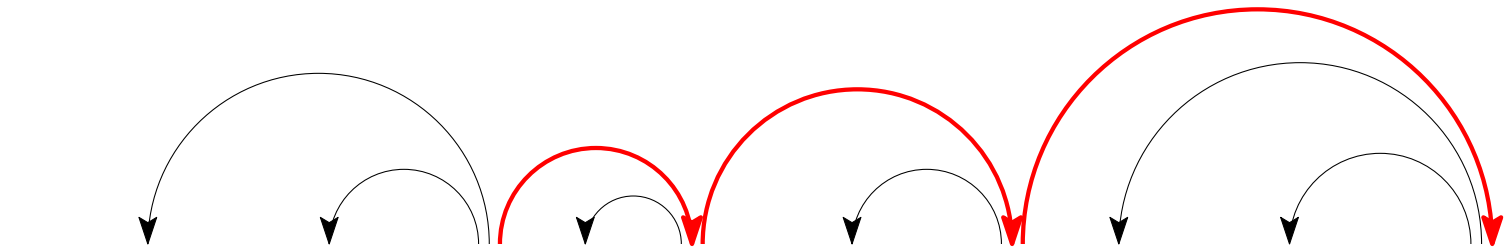


Head *initial* dependencies: head before dependent

Proportion of head-initial dependencies: $3/9 = 1/3$

Calculating arrangement-dependent metrics

Head initial dependencies (Liu, 2010)



	Yesterday	John	saw	a	dog	which	was	a	Yorkshire	Terrier
Positions	1	2	3	4	5	6	7	8	9	10
Head position	3	3	0	5	3	7	5	10	10	7

Head *initial* dependencies: head before dependent

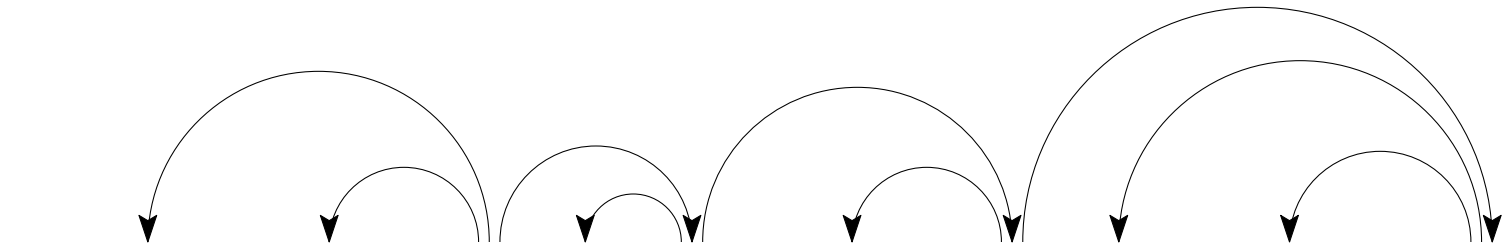
Proportion of head-initial dependencies: $3/9 = 1/3$

```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
head_initial_implicit = lal.linarr.head_initial(rt)
head_initial_explicit = lal.linarr.head_initial(rt, [0,1,2,3,4,5,6,7,8,9])
assert(head_initial_implicit == head_initial_explicit)
head_initial_reverse = lal.linarr.head_initial(rt, [9,8,7,6,5,4,3,2,1,0])
assert(head_initial_reverse == 1 - head_initial_implicit)
```

The same arrangement but reveresed

Calculating arrangement-dependent metrics

Sum of dependency distances

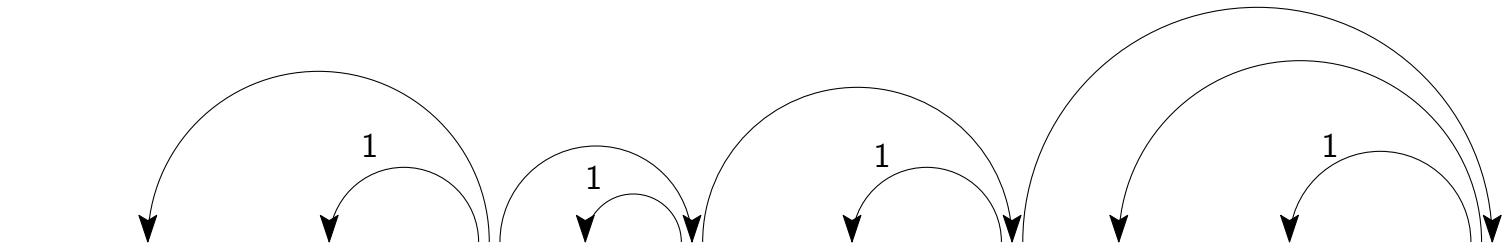


Positions	1	2	3	4	5	6	7	8	9	10
Head position	3	3	0	5	3	7	5	10	10	7

Dependency distance between syntactically-related words: the number of words inbetween plus 1

Calculating arrangement-dependent metrics

Sum of dependency distances

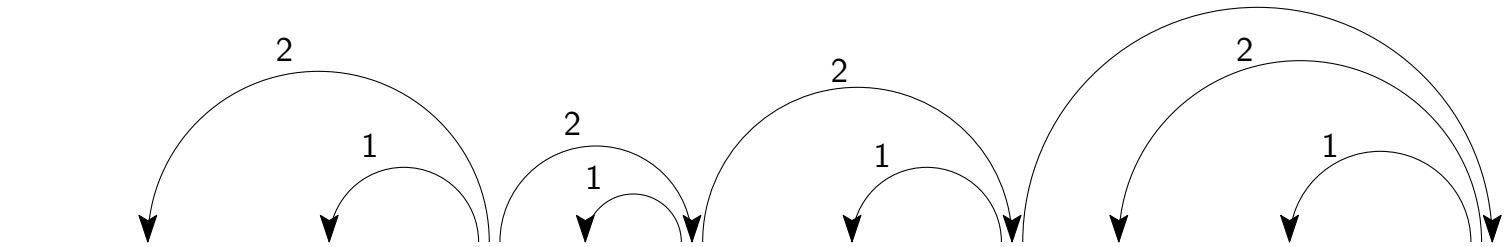


Positions	1	2	3	4	5	6	7	8	9	10
Head position	3	3	0	5	3	7	5	10	10	7

Dependency distance between syntactically-related words: the number of words inbetween plus 1

Calculating arrangement-dependent metrics

Sum of dependency distances

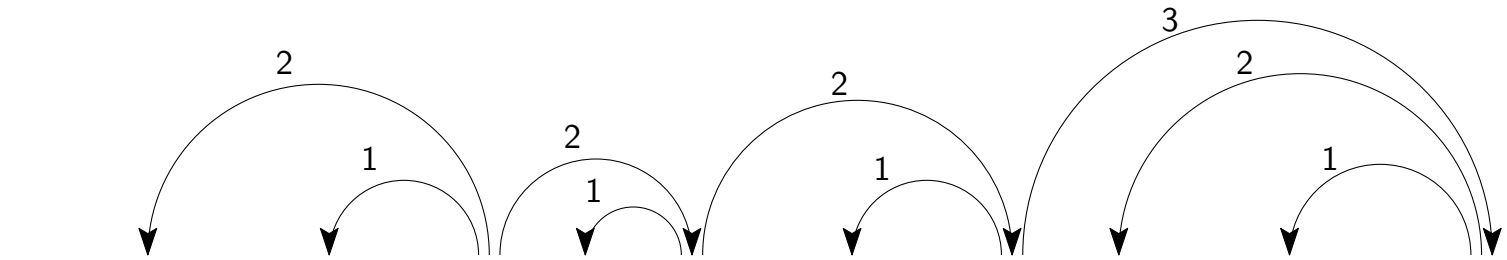


Positions	1	2	3	4	5	6	7	8	9	10
Head position	3	3	0	5	3	7	5	10	10	7

Dependency distance between syntactically-related words: the number of words inbetween plus 1

Calculating arrangement-dependent metrics

Sum of dependency distances

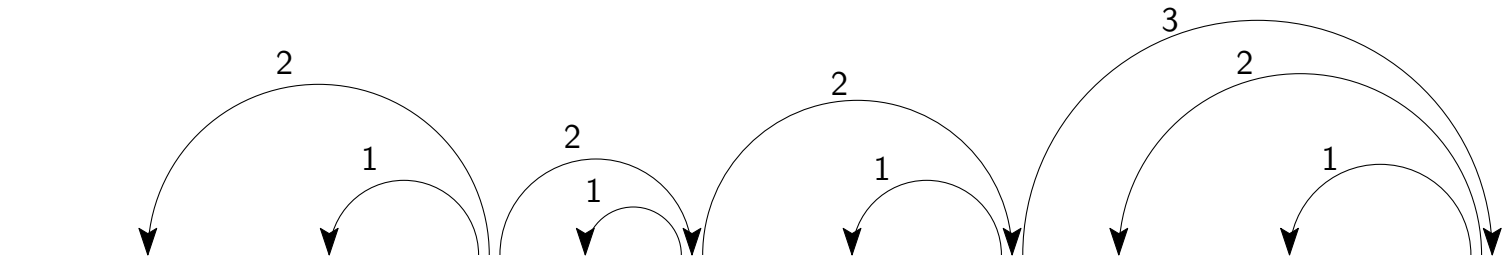


Positions	1	2	3	4	5	6	7	8	9	10
Head position	3	3	0	5	3	7	5	10	10	7

Dependency distance between syntactically-related words: the number of words inbetween plus 1

Calculating arrangement-dependent metrics

Sum of dependency distances



Positions	1	2	3	4	5	6	7	8	9	10
Head position	3	3	0	5	3	7	5	10	10	7

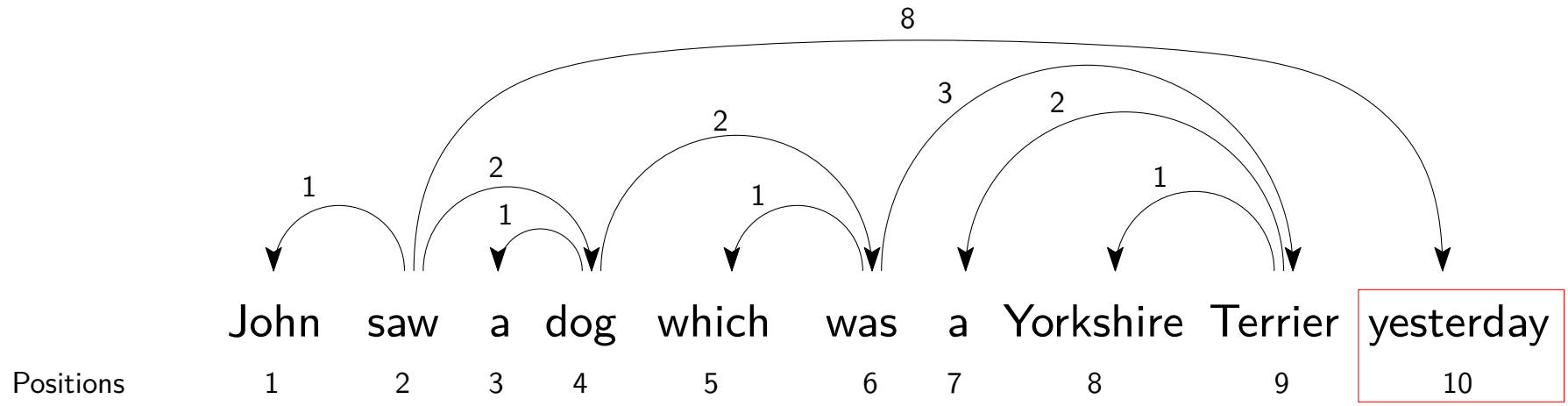
Dependency distance between syntactically-related words: the number of words inbetween plus 1

```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
D1 = lal.linarr.sum_edge_lengths(rt)
D1 = lal.linarr.sum_edge_lengths(rt, [0,1,2,3,4,5,6,7,8,9])
```

$D1 = 15$

Calculating arrangement-dependent metrics

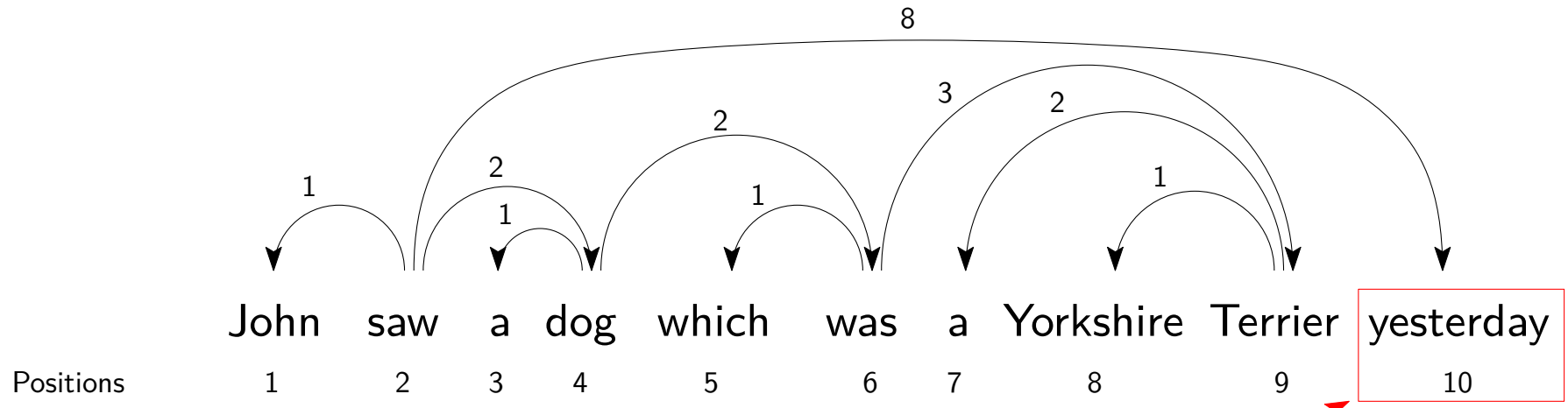
Sum of dependency distances




```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
D2 = lal.linarr.sum_edge_lengths(rt, [x,x,x,x,x,x,x,x,x,x])
```

Calculating arrangement-dependent metrics

Sum of dependency distances

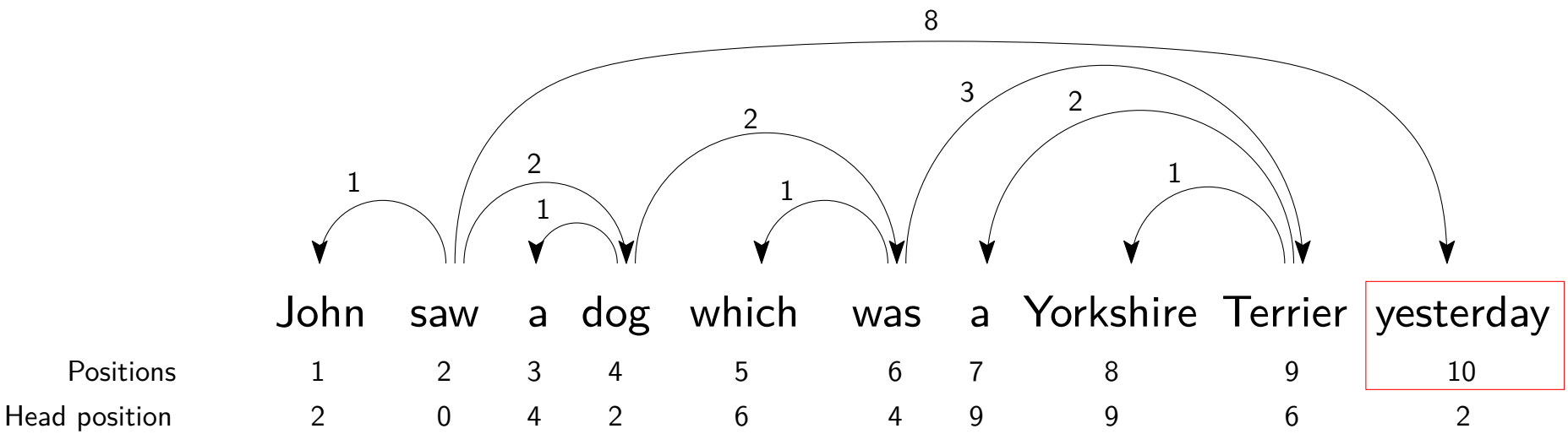



```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
D2 = lal.linarr.sum_edge_lengths(rt, [9,0,1,2,3,4,5,6,7,8])
```

$D1 = 15,\ D2 = 21$

Calculating arrangement-dependent metrics

Sum of dependency distances




```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
D2 = lal.linarr.sum_edge_lengths(rt, [9,0,1,2,3,4,5,6,7,8])
```

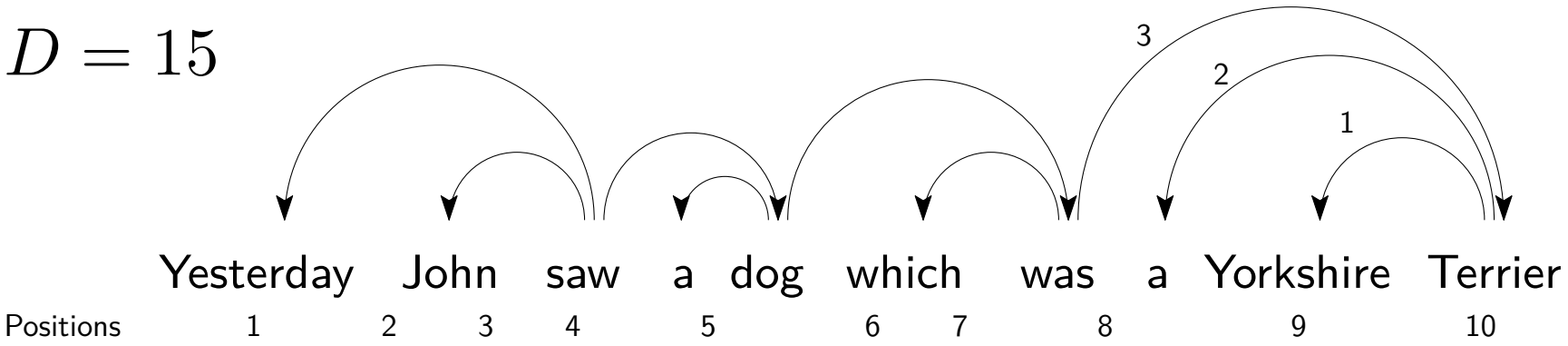
```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree([2,0,4,2,6,4,9,9,6,2])
D2 = lal.linarr.sum_edge_lengths(rt)
```

$D1 = 15,\ D2 = 21$

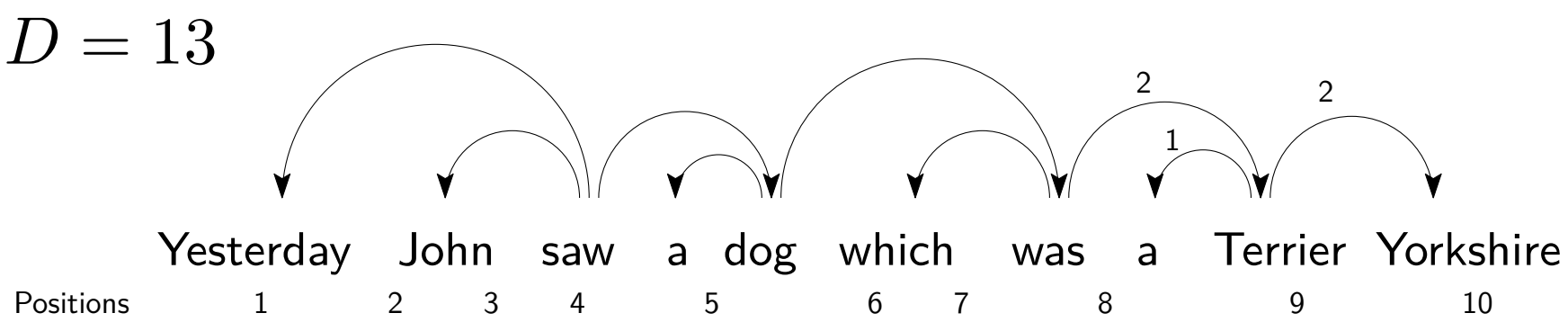
Calculating arrangement-dependent metrics

Minimum arrangements

$D = 15$

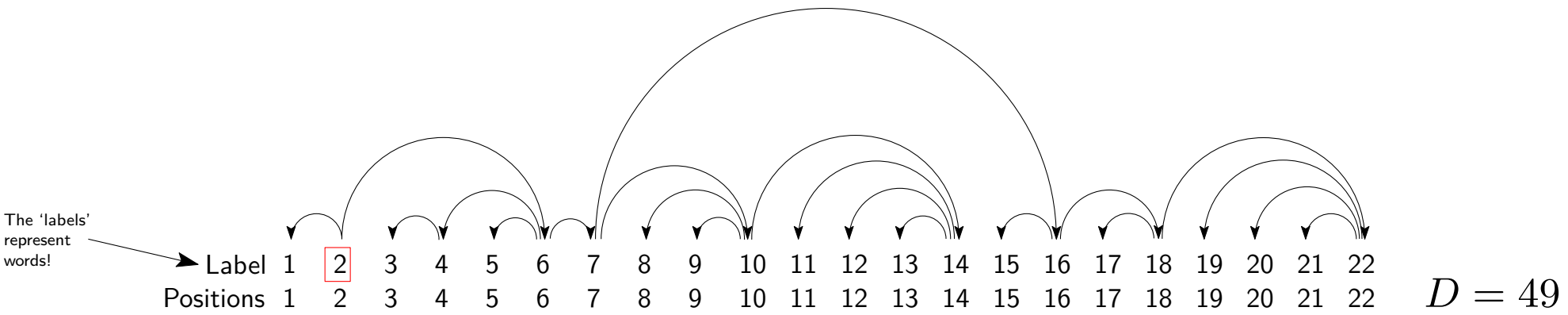


$D = 13$



Calculating arrangement-dependent metrics

Minimum arrangements

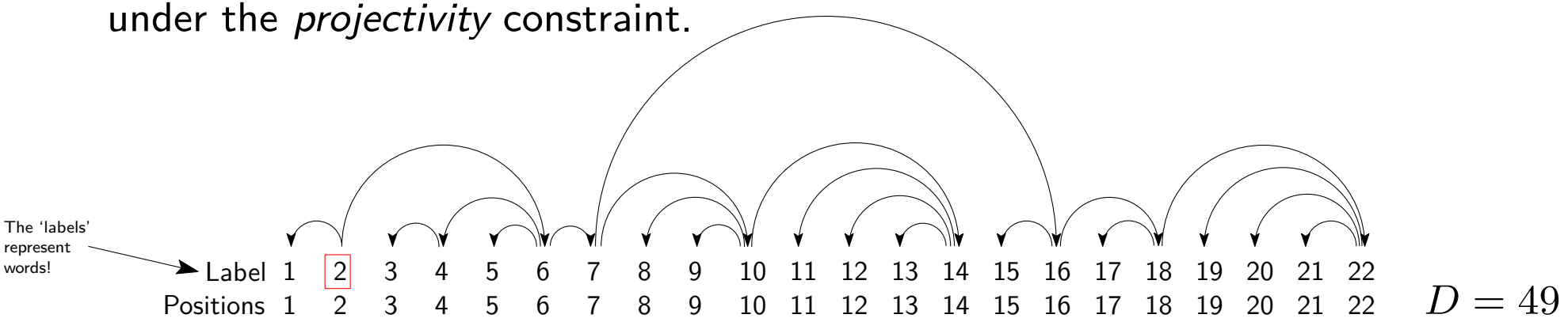


$D = 49$

Calculating arrangement-dependent metrics

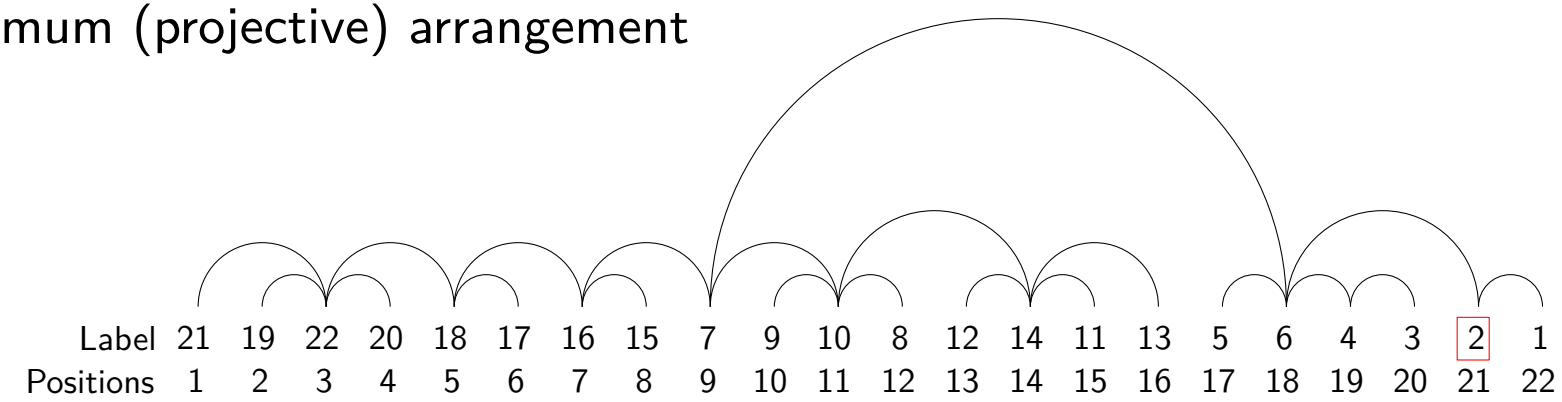
Minimum arrangements

- If the root can't be covered and edge crossings are disallowed, we are minimizing under the *projectivity* constraint.



$D = 49$

Minimum (projective) arrangement

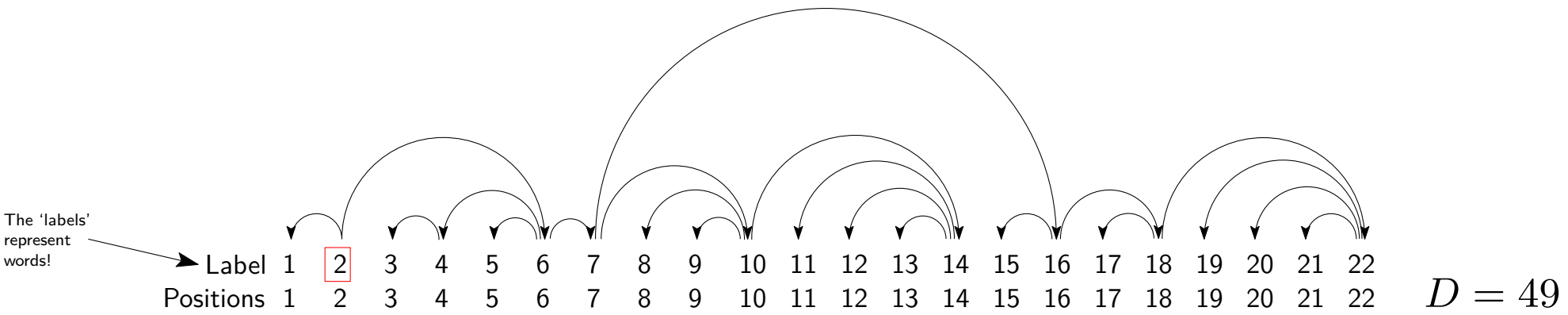


$D = 39$

Calculating arrangement-dependent metrics

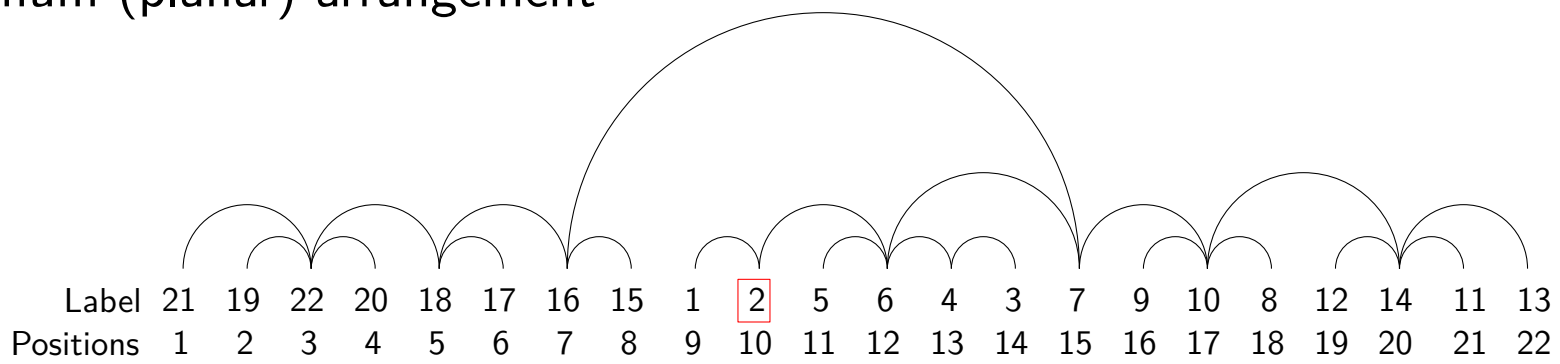
Minimum arrangements

- If edge crossings are disallowed, but the root can be converted then we are minimizing under the *planarity* constraint.



$D = 49$

Minimum (planar) arrangement

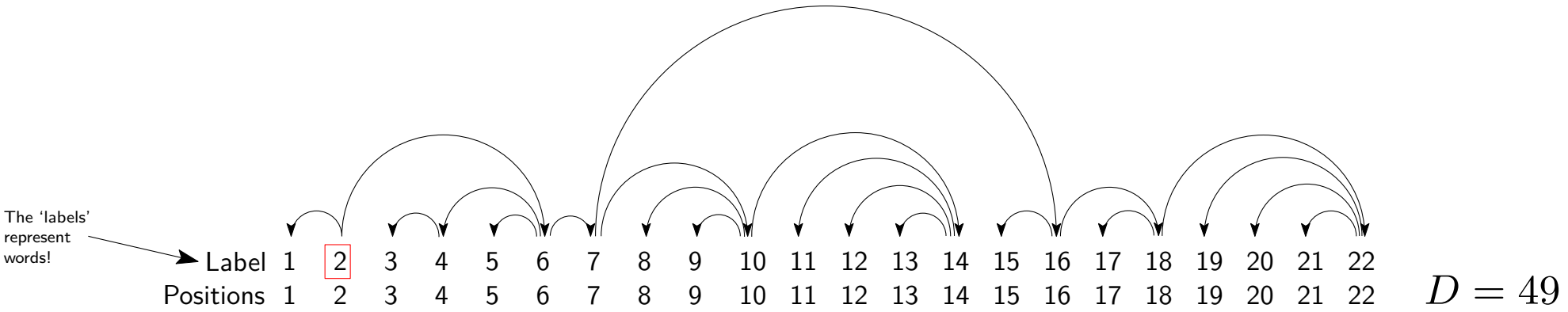


$D = 38$

Calculating arrangement-dependent metrics

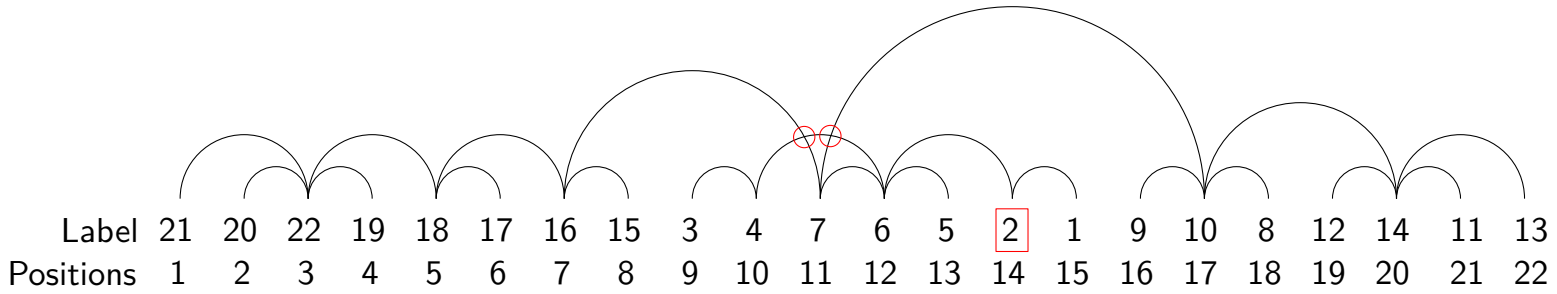
Minimum arrangements

- Sometimes we require edge crossings to achieve the lowest D possible



$D = 49$

Minimum (unconstrained) arrangement



$D = 37$

Calculating arrangement-dependent metrics

Minimum arrangements

- Sometimes we require edge crossings to achieve the lowest D possible
- If edge crossings are disallowed, but the root can be converted then we are minimizing under the *planarity* constraint.
- If the root can't be covered and edge crossings are disallowed, we are minimizing under the *projectivity* constraint.

```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree(
    [2,0,4,6,6,2,6,10,10,7,14,14,14,10,16,7,18,16,22,22,22,18]
)
# projective arrangements
D_min_proj, arr_min_proj = lal.linarr.min_sum_edge_lengths_projective(rt)
print("Projective:   ", arr_min_proj)
# planar arrangements
D_min_plan, arr_min_plan = lal.linarr.min_sum_edge_lengths_planar(rt)
print("Planar:       ", arr_min_plan)
# unconstrained arrangements
D_min, arr_min = lal.linarr.min_sum_edge_lengths(rt)
print("Unconstrained:", arr_min)
# by definition we have...
assert(D_min <= D_min_plan and D_min_plan <= D_min_proj)
```

```
Projective:    (21, 20, 19, 18, 16, 17, 8, 11, 9, 10, 14, 12, 15, 13, 7, 6, 5, 4, 1, 3, 0, 2)
Planar:        (8, 9, 13, 12, 10, 11, 14, 17, 15, 16, 20, 18, 21, 19, 7, 6, 5, 4, 1, 3, 0, 2)
Unconstrained: (14, 13, 8, 9, 12, 11, 10, 17, 15, 16, 20, 18, 21, 19, 7, 6, 5, 4, 3, 1, 0, 2)
```

Calculating arrangement-dependent metrics

Minimum arrangements

- Sometimes we require edge crossings to achieve the lowest D possible
- If edge crossings are disallowed, but the root can be converted then we are minimizing under the *planarity* constraint.
- If the root can't be covered and edge crossings are disallowed, we are minimizing under the *projectivity* constraint.

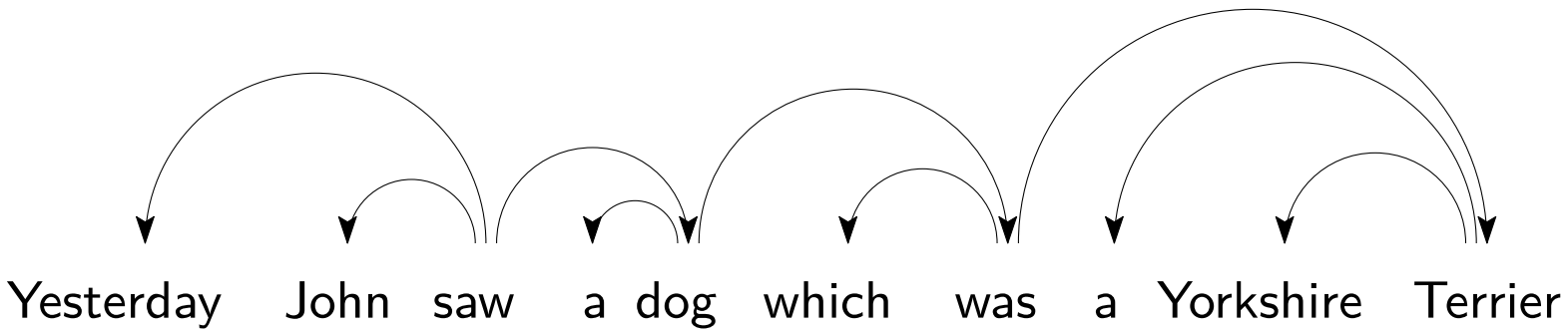
```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree(
    [2,0,4,6,6,2,6,10,10,7,14,14,14,10,16,7,18,16,22,22,22,18]
)
# projective arrangements
D_min_proj, arr_min_proj = lal.linarr.min_sum_edge_lengths_projective(rt)
print("Projective:   ", arr_min_proj)
# planar arrangements
D_min_plan, arr_min_plan = lal.linarr.min_sum_edge_lengths_planar(rt)
print("Planar:       ", arr_min_plan)
# unconstrained arrangements
D_min, arr_min = lal.linarr.min_sum_edge_lengths(rt)
print("Unconstrained:", arr_min)
# by definition we have...
assert(D_min <= D_min_plan and D_min_plan <= D_min_proj)
```

```
lal.linarr.sum_edge_lengths(rt, arr_proj)
```

```
Projective:    (21, 20, 19, 18, 16, 17, 8, 11, 9, 10, 14, 12, 15, 13, 7, 6, 5, 4, 1, 3, 0, 2)
Planar:        (8, 9, 13, 12, 10, 11, 14, 17, 15, 16, 20, 18, 21, 19, 7, 6, 5, 4, 1, 3, 0, 2)
Unconstrained: (14, 13, 8, 9, 12, 11, 10, 17, 15, 16, 20, 18, 21, 19, 7, 6, 5, 4, 3, 1, 0, 2)
```

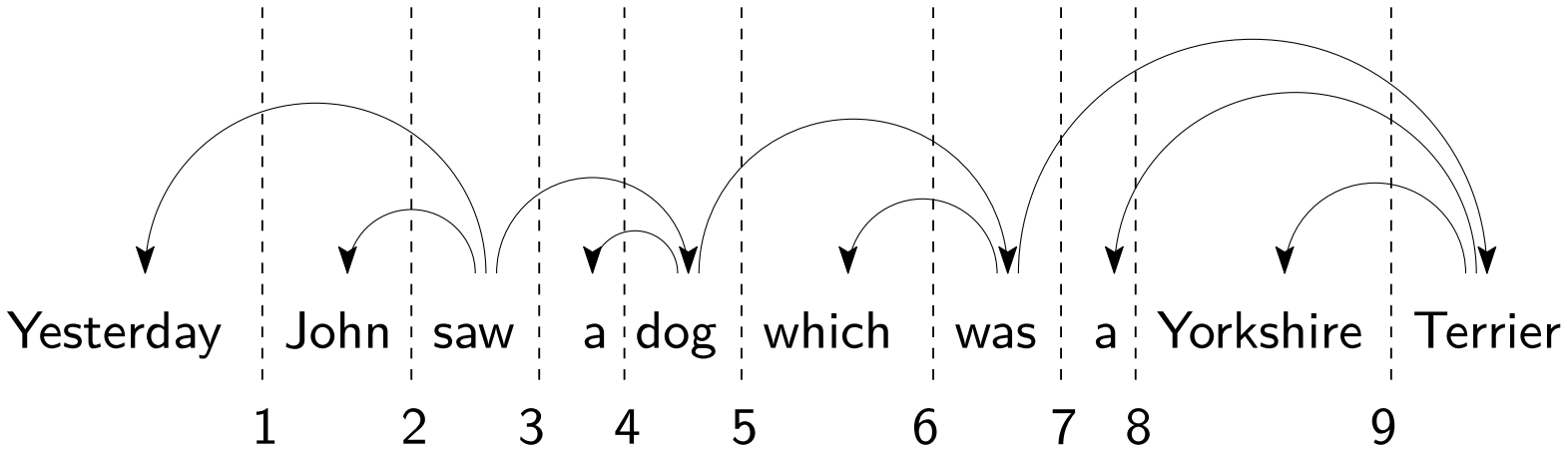
Calculating arrangement-dependent metrics

Dependency fluxes (Kahane et al., 2017)



Calculating arrangement-dependent metrics

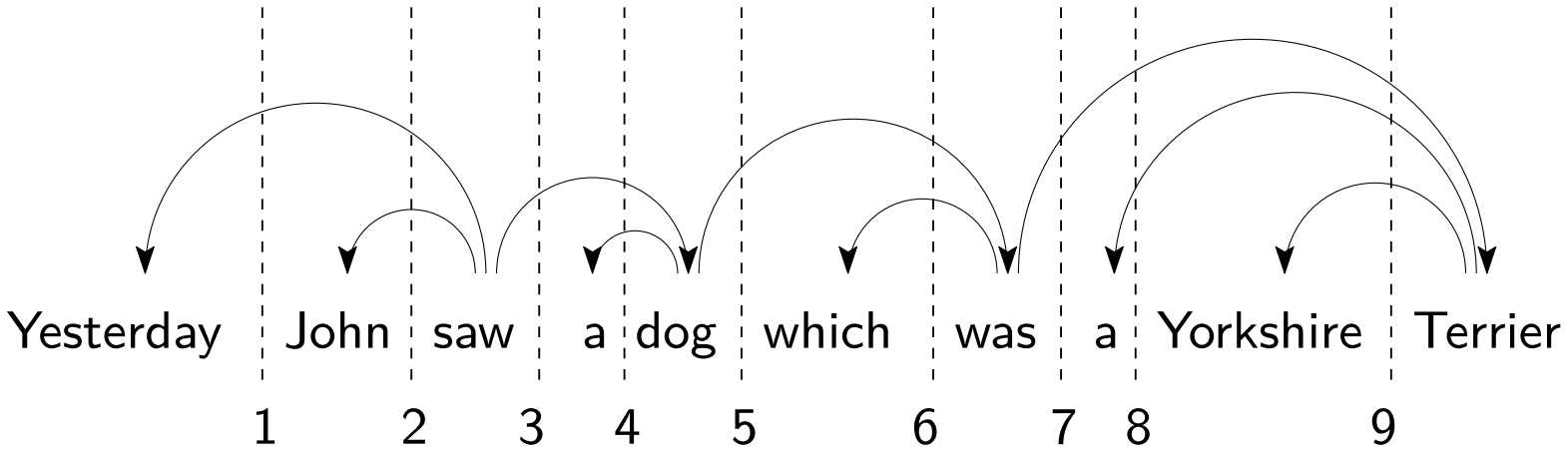
Dependency fluxes (Kahane et al., 2017)



Fluxes are found *inbetween* words

Calculating arrangement-dependent metrics

Dependency fluxes (Kahane et al., 2017)

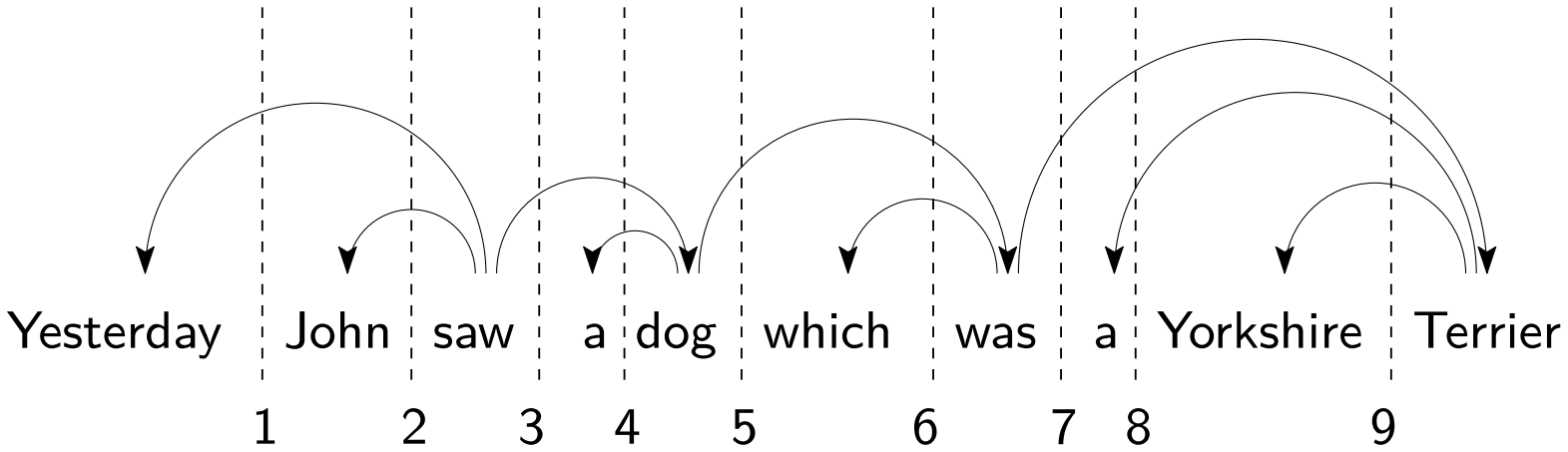


Fluxes are found *inbetween* words

- *size*: the number of dependencies

Calculating arrangement-dependent metrics

Dependency fluxes (Kahane et al., 2017)

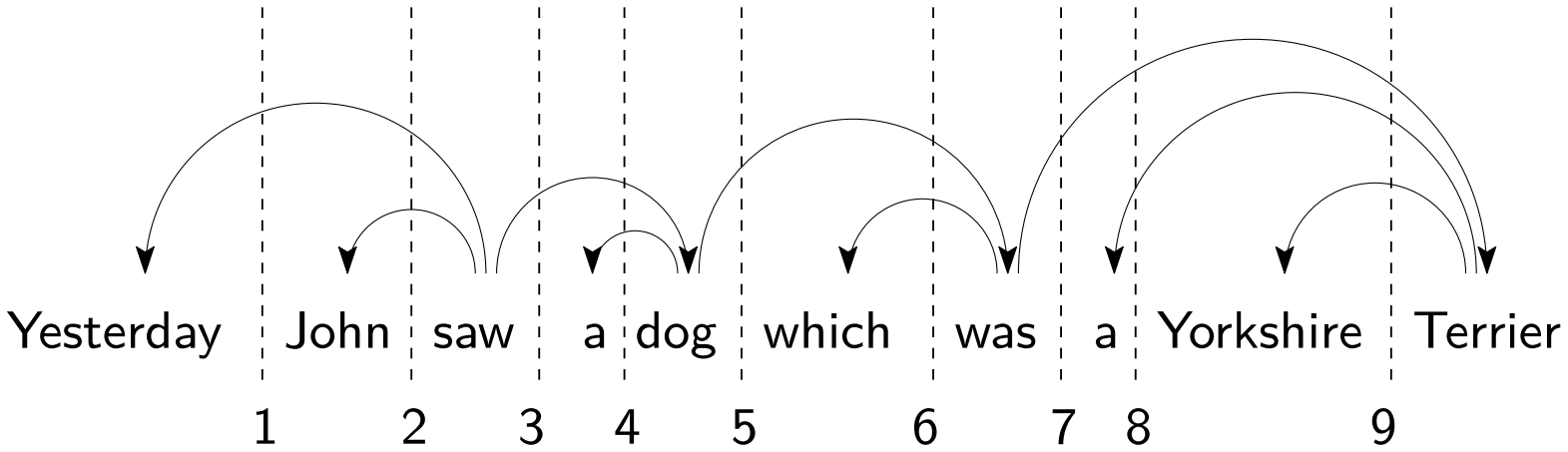


Fluxes are found *inbetween* words

- *size*: the number of dependencies
- *weight*: the size of the largest subset of disjoint dependencies (that do not share words among each other)

Calculating arrangement-dependent metrics

Dependency fluxes (Kahane et al., 2017)

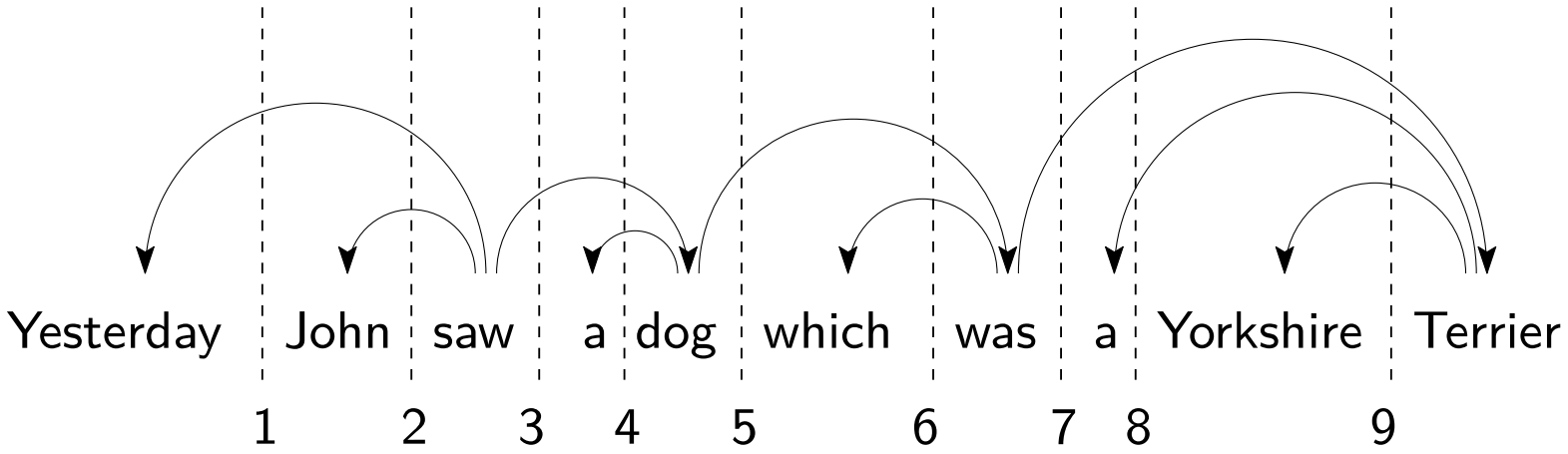


Fluxes are found *inbetween* words

- *size*: the number of dependencies
- *weight*: the size of the largest subset of disjoint dependencies (that do not share words among each other)
- *W/S* ratio: weight divided by size

Calculating arrangement-dependent metrics

Dependency fluxes (Kahane et al., 2017)

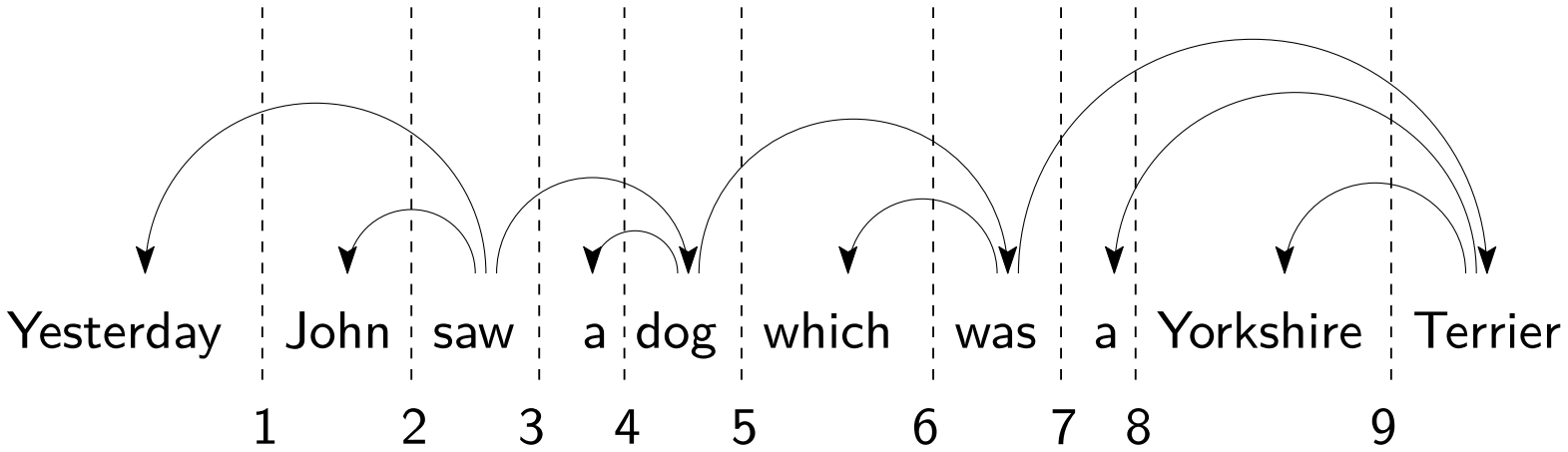


Fluxes are found *inbetween* words

- *size*: the number of dependencies
- *weight*: the size of the largest subset of disjoint dependencies (that do not share words among each other)
- *W/S* ratio: weight divided by size
- *left* (resp. *right*) *span* of the flux is the number of words to the *left* (resp. *right*) which are vertices of a dependency

Calculating arrangement-dependent metrics

Dependency fluxes (Kahane et al., 2017)

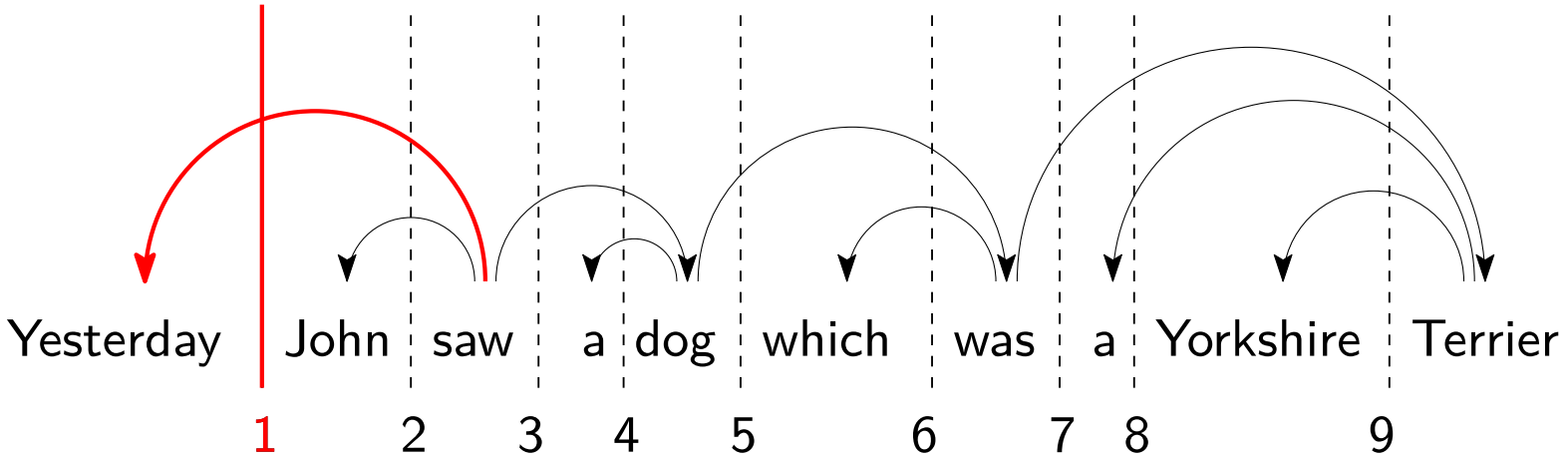


Fluxes are found *inbetween* words

- *size*: the number of dependencies
- *weight*: the size of the largest subset of disjoint dependencies (that do not share words among each other)
- *W/S* ratio: weight divided by size
- *left* (resp. *right*) *span* of the flux is the number of words to the *left* (resp. *right*) which are vertices of a dependency
- *R/L* ratio: right span divided by left span

Calculating arrangement-dependent metrics

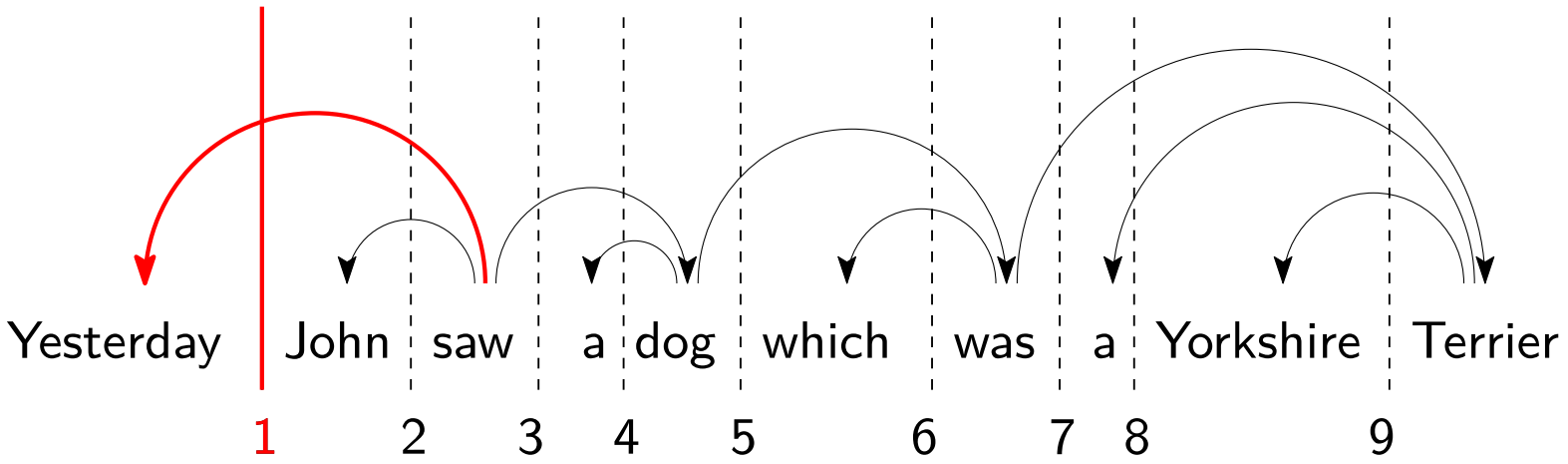
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1						

Calculating arrangement-dependent metrics

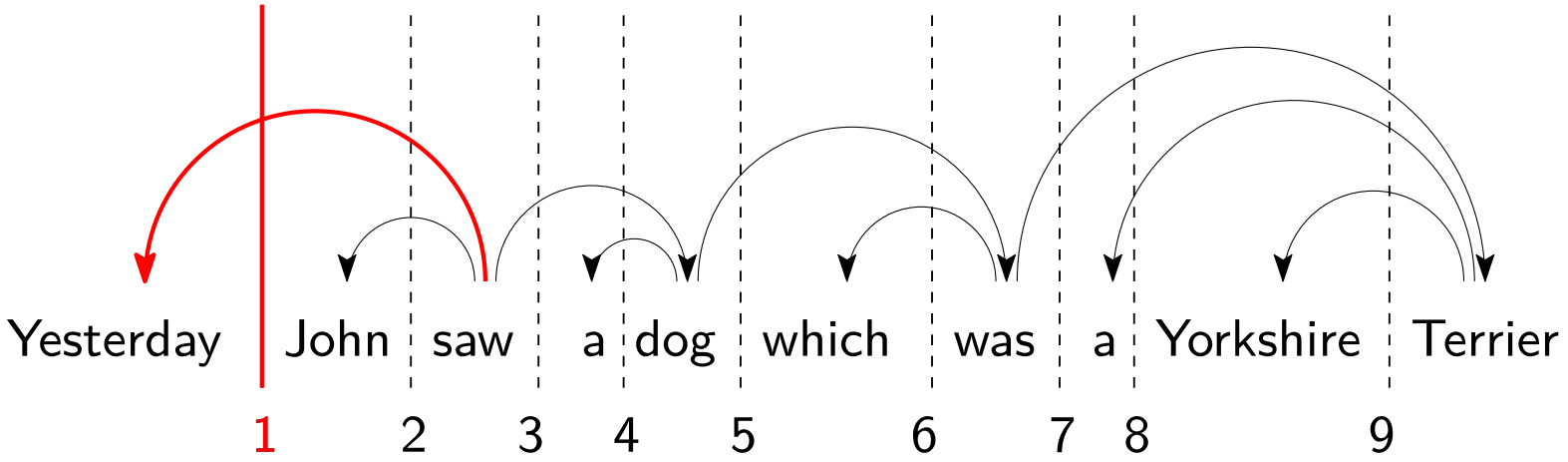
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1					

Calculating arrangement-dependent metrics

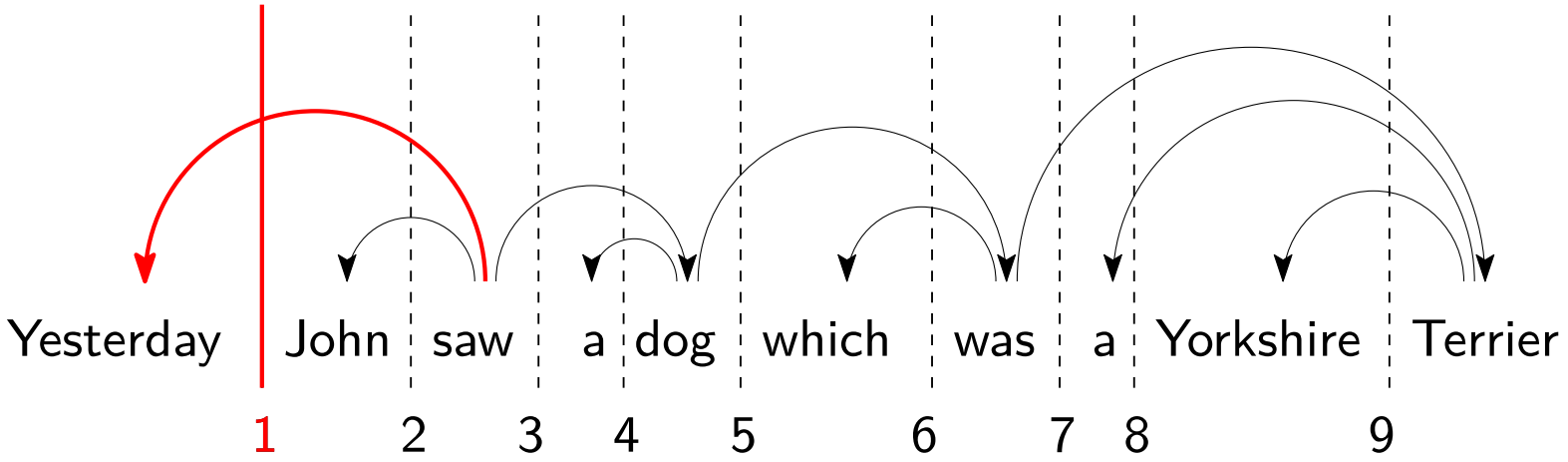
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1				

Calculating arrangement-dependent metrics

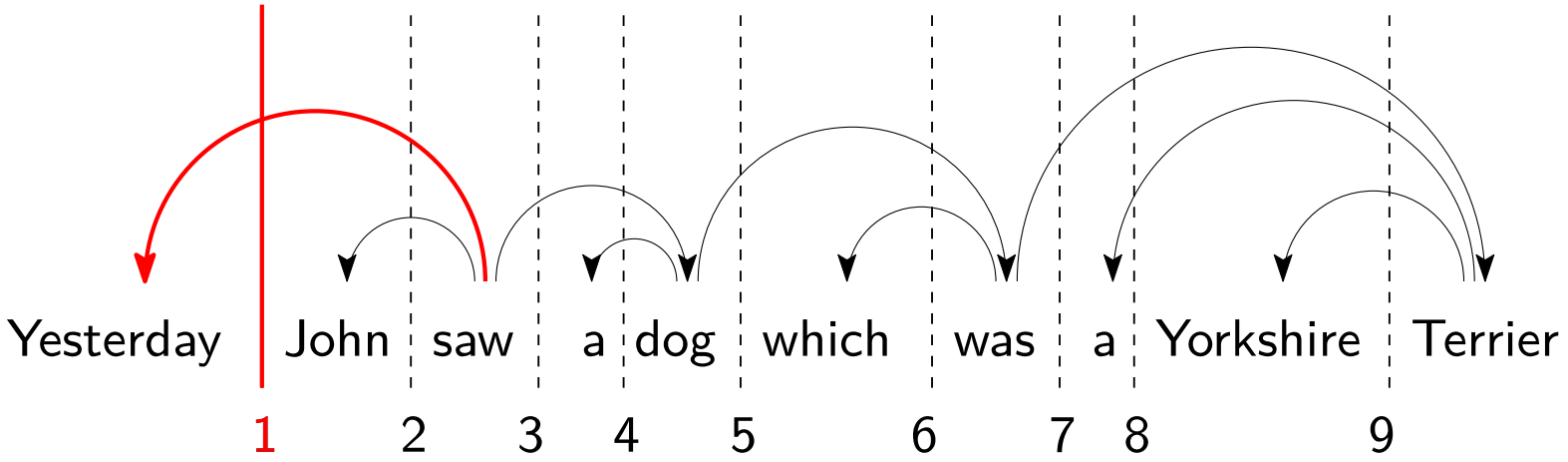
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1			

Calculating arrangement-dependent metrics

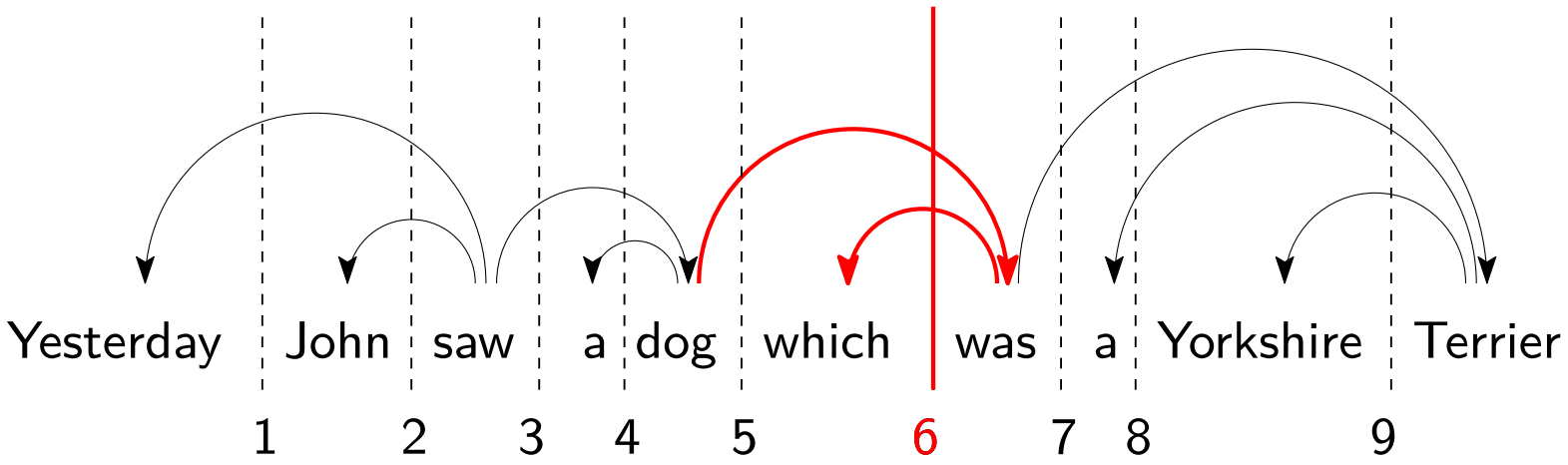
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1	1	1	1

Calculating arrangement-dependent metrics

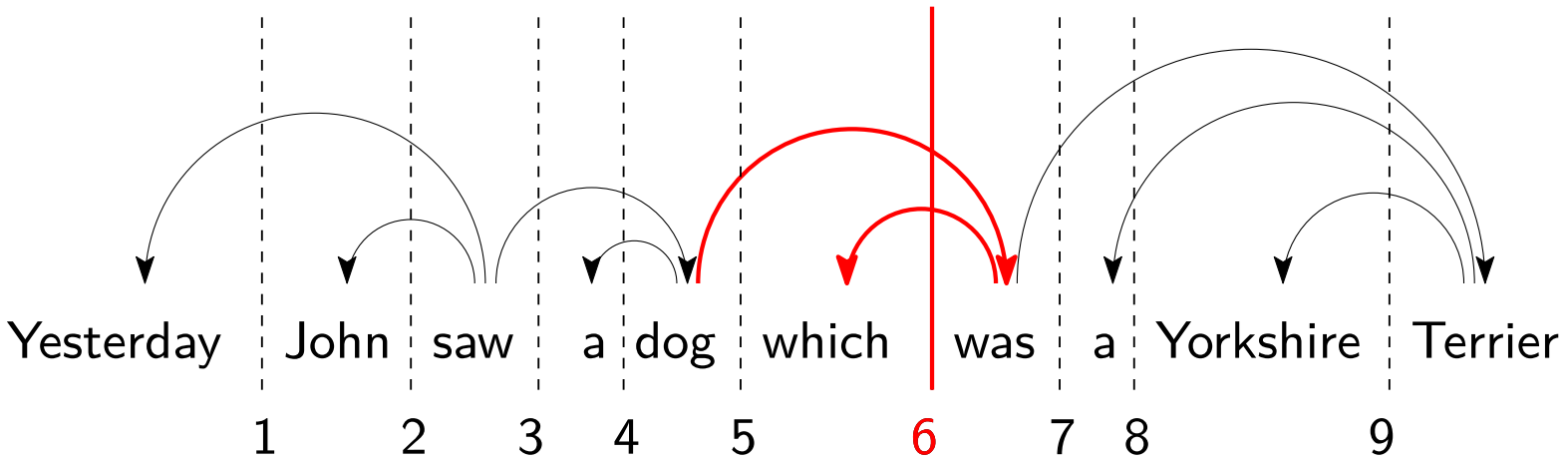
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1	1	1	1
6						

Calculating arrangement-dependent metrics

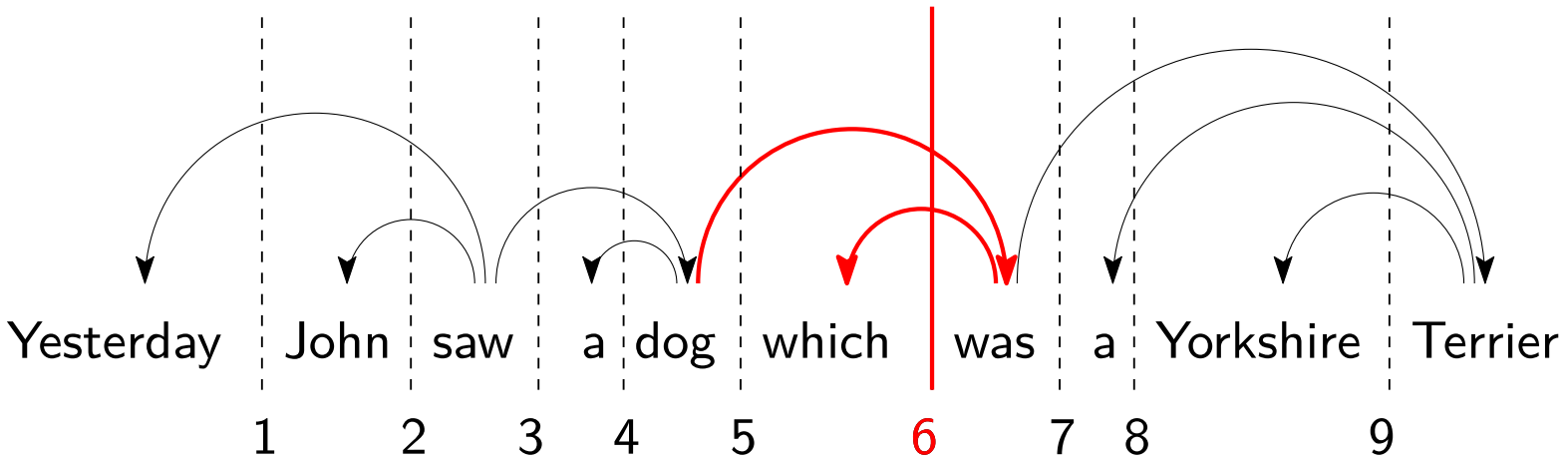
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1	1	1	1
6	2					

Calculating arrangement-dependent metrics

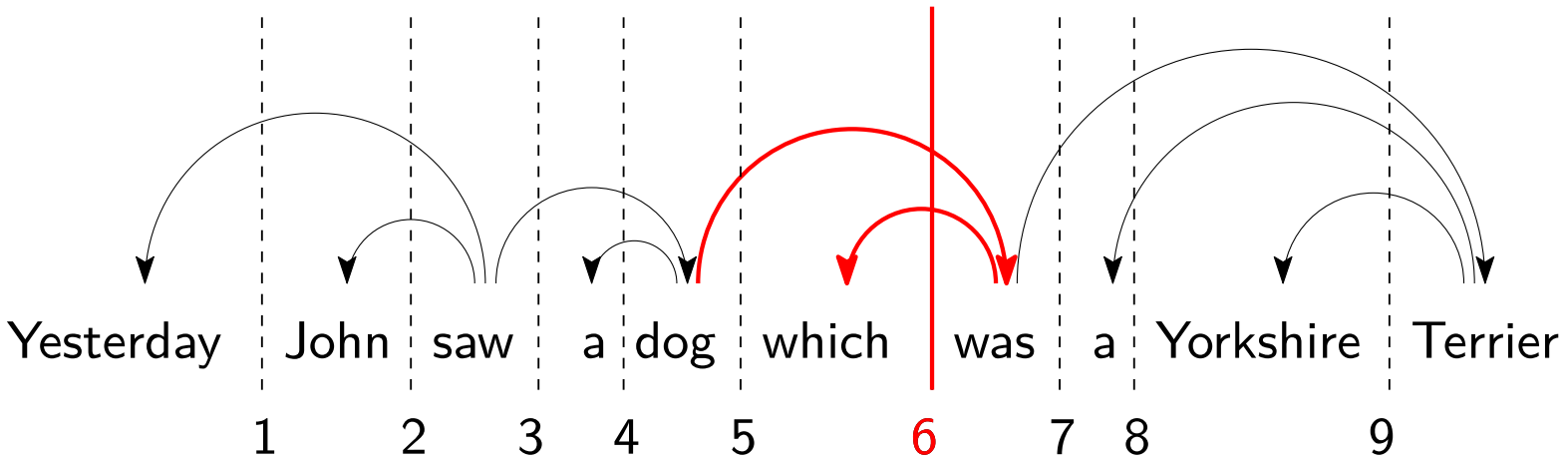
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1	1	1	1
6	2	1				

Calculating arrangement-dependent metrics

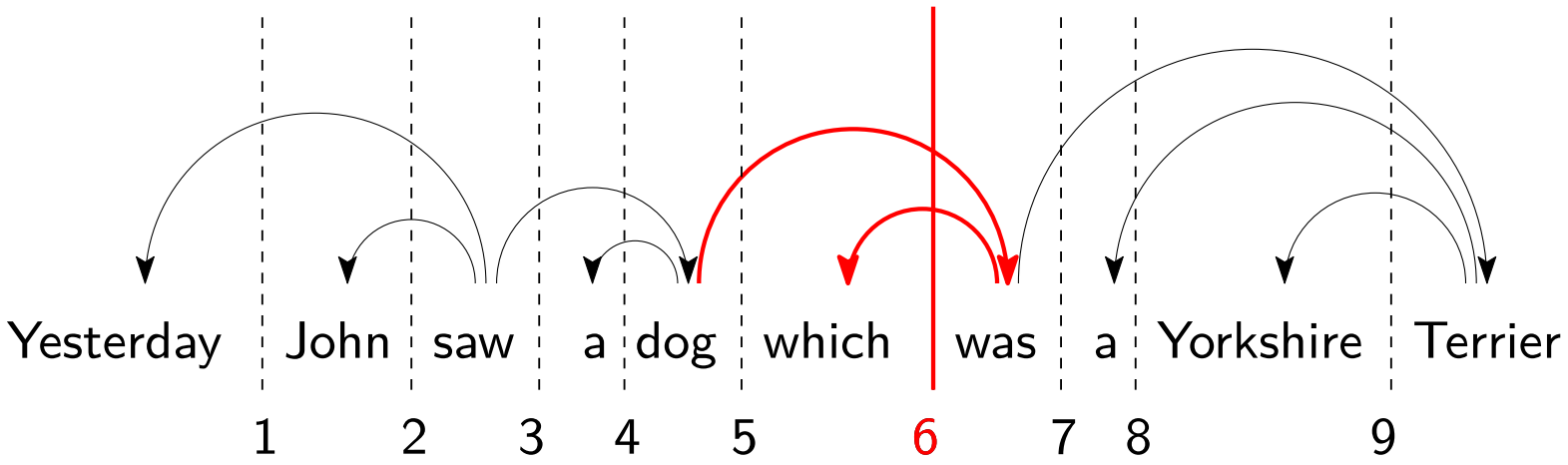
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1	1	1	1
6	2	1	2			

Calculating arrangement-dependent metrics

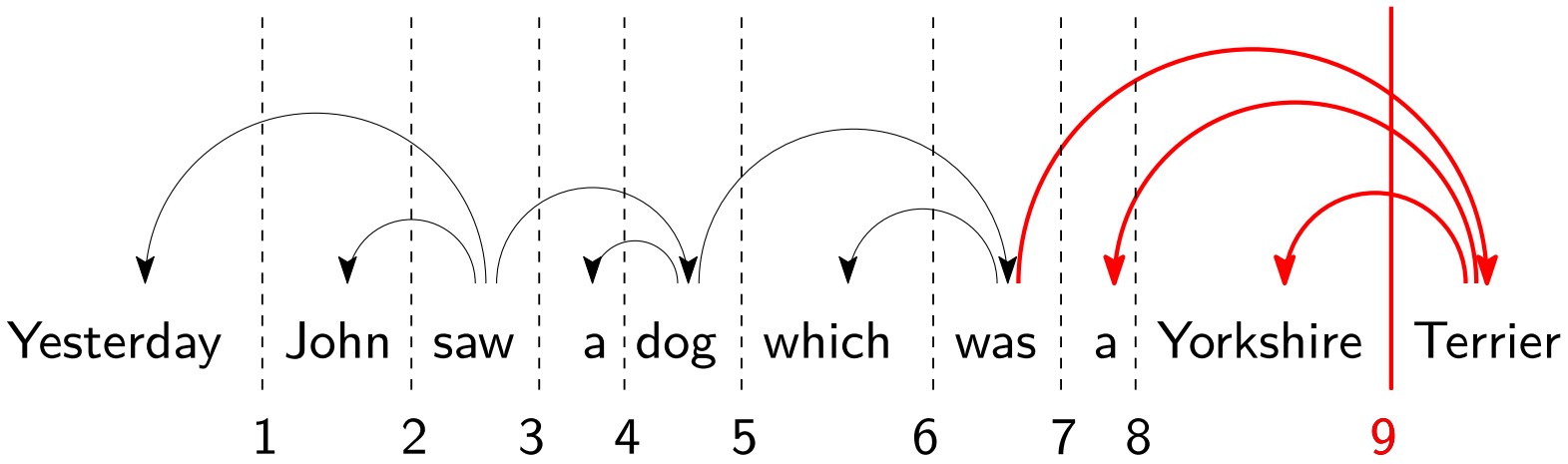
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1	1	1	1
6	2	1	2	1	1/2	1/2

Calculating arrangement-dependent metrics

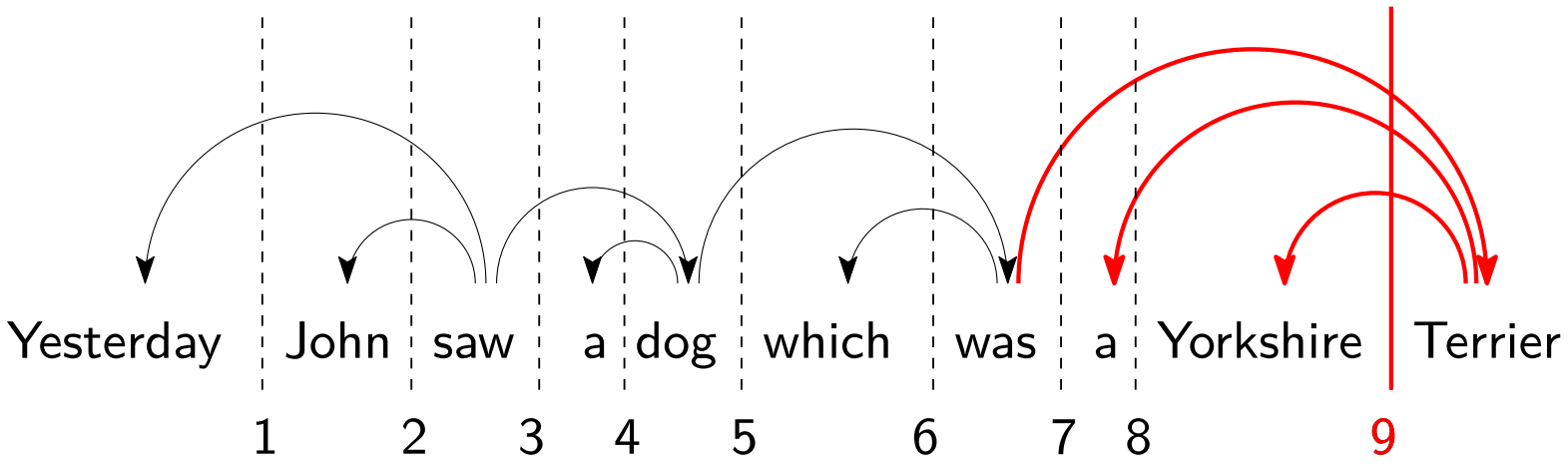
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1	1	1	1
6	2	1	2	1	1/2	1/2
9						

Calculating arrangement-dependent metrics

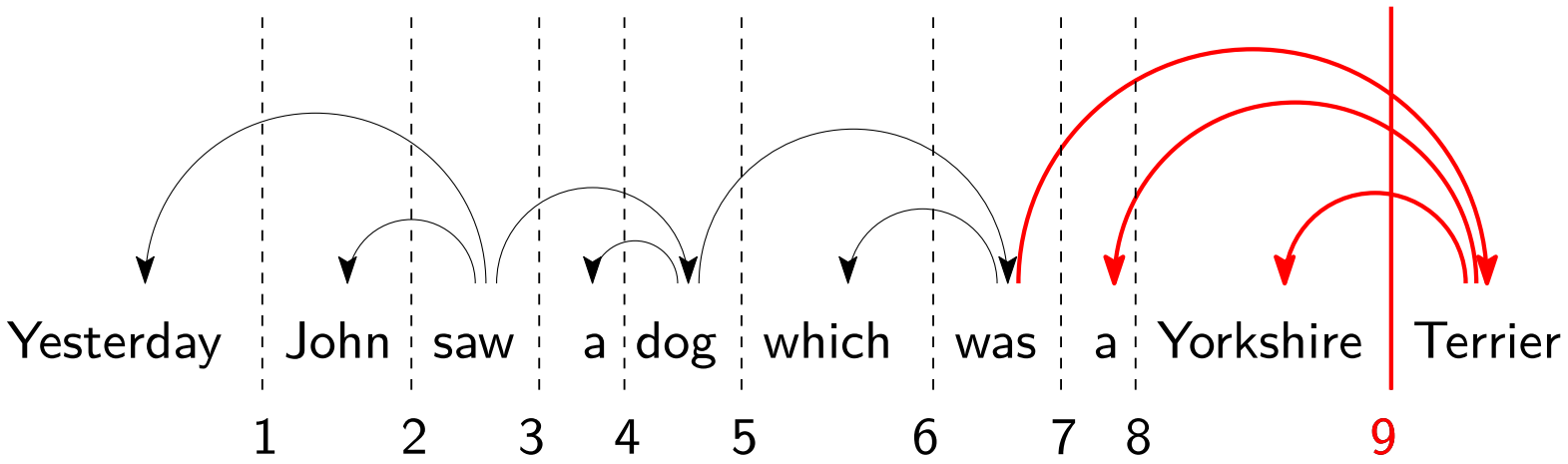
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1	1	1	1
6	2	1	2	1	1/2	1/2
9	3					

Calculating arrangement-dependent metrics

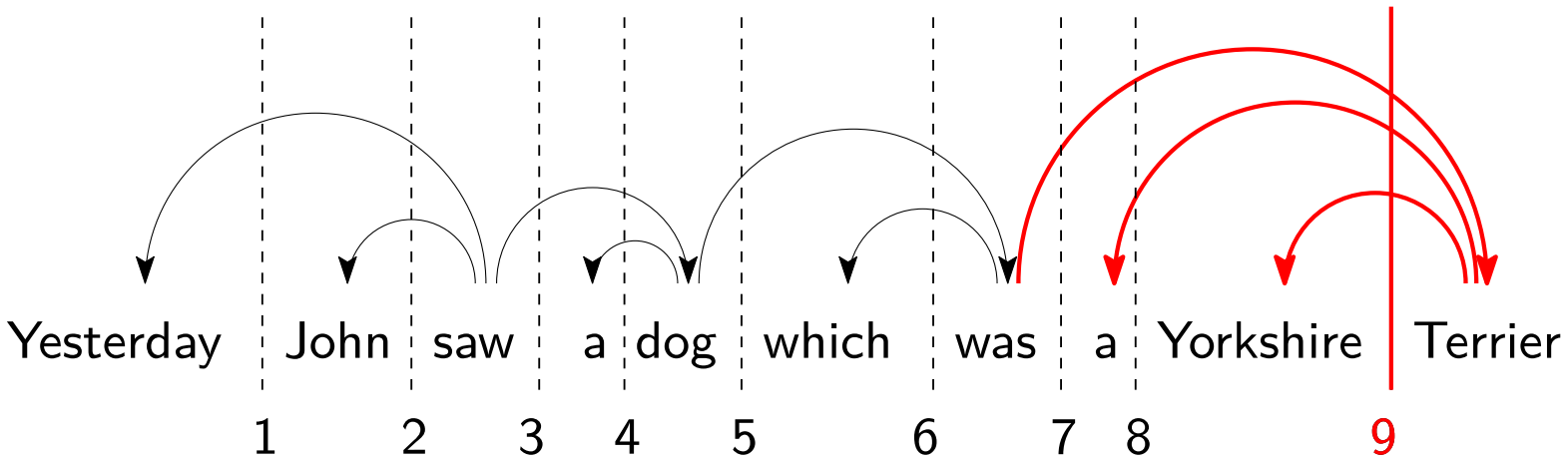
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1	1	1	1
6	2	1	2	1	1/2	1/2
9	3	1				

Calculating arrangement-dependent metrics

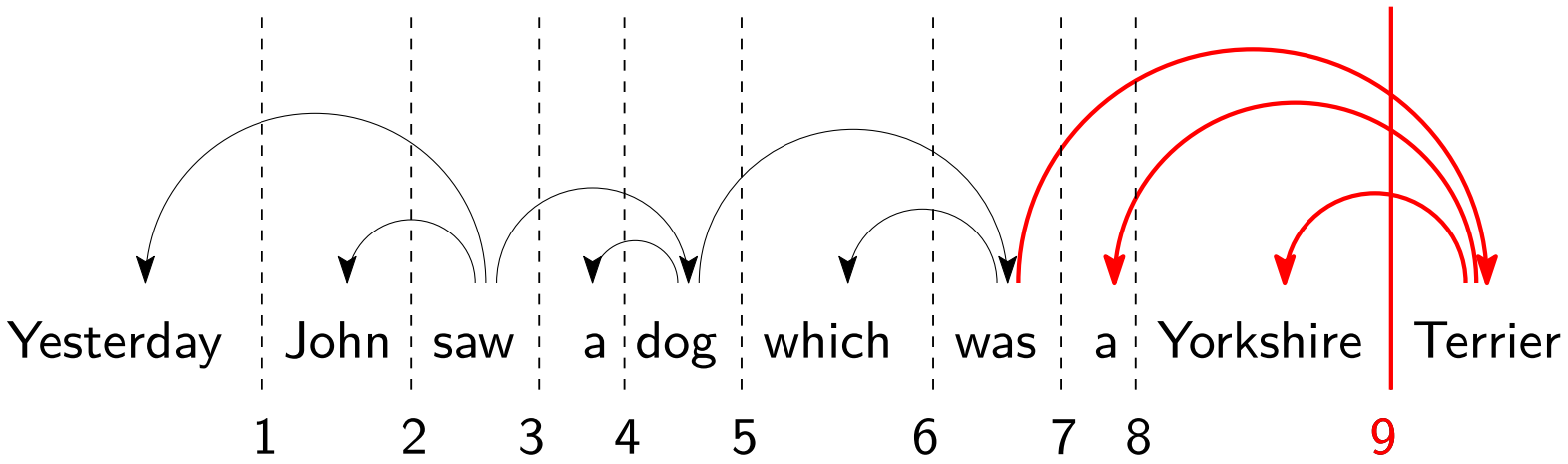
Dependency fluxes (Kahane et al., 2017)



			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1	1	1	1
6	2	1	2	1	1/2	1/2
9	3	1	3			

Calculating arrangement-dependent metrics

Dependency fluxes (Kahane et al., 2017)

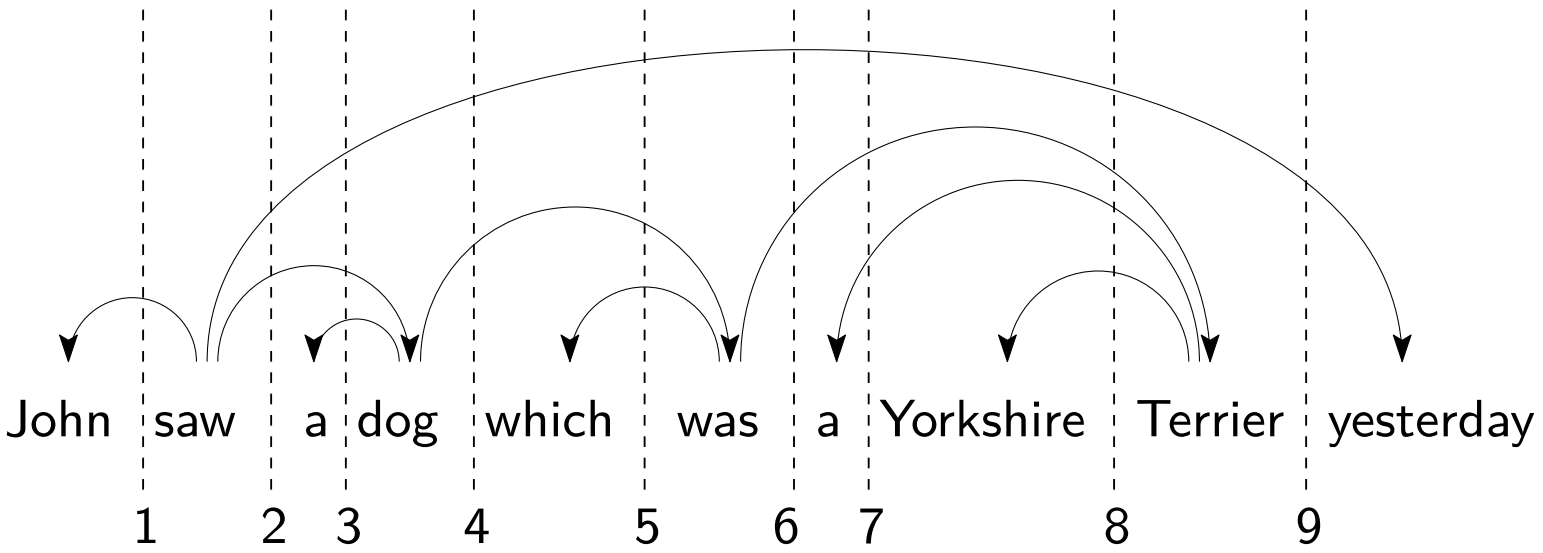


			span			
flux	size	weight	left	right	R/L	W/S
1	1	1	1	1	1	1
6	2	1	2	1	1/2	1/2
9	3	1	3	1	1/3	1/3

Calculating arrangement-dependent metrics

Dependency fluxes (Kahane et al., 2017)

Which of these fluxes have weight > 1? Which of these have right span > 1?

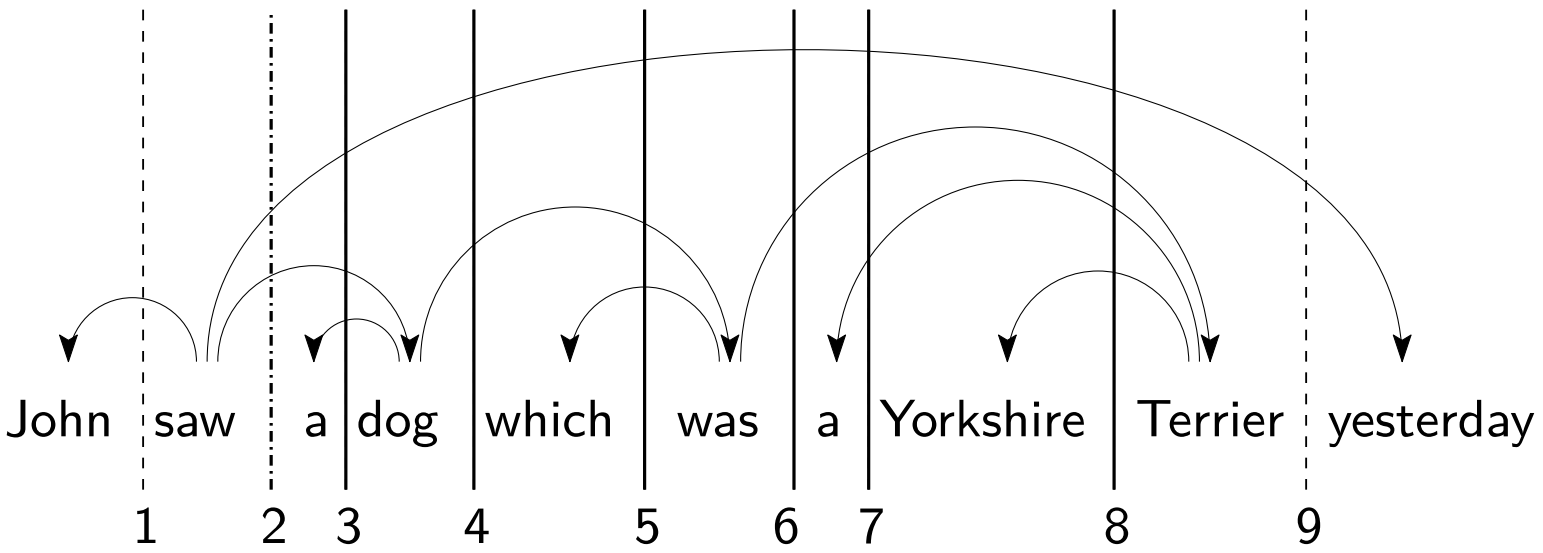


- *weight*: the size of the largest subset of disjoint dependencies (that do not share words among each other)
- *left* (resp. *right*) *span* of the flux is the number of words to the *left* (resp. *right*) which are vertices of a dependency

Calculating arrangement-dependent metrics

Dependency fluxes (Kahane et al., 2017)

Which of these fluxes have weight > 1? Which of these have right span > 1?



- *weight*: the size of the largest subset of disjoint dependencies (that do not share words among each other)
- *left* (resp. *right*) *span* of the flux is the number of words to the *left* (resp. *right*) which are vertices of a dependency

Calculating arrangement-dependent metrics

Dependency fluxes (Kahane et al., 2017)

```
import laldebug as lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
# calculate the fluxes
fluxes = lal.linarr.compute_flux(rt)
number_of_fluxes = len(fluxes)
# print the fluxes
for i in range(0, number_of_fluxes):
    print("Flux", i+1)
    print("    Dependencies:", fluxes[i].get_dependencies())
    print("    Left span:", fluxes[i].get_left_span())
    print("    Right span:", fluxes[i].get_right_span())
    print("    Weight:", fluxes[i].get_weight())
    print("    R/L ratio:", fluxes[i].get_RL_ratio())
    print("    W/S ratio:", fluxes[i].get_WS_ratio())
```

Self assessment

Choose a sentence in Chinese of your own and produce a translation equivalent into English. Calculate the metrics explained in the previous slides on each sentence and compare them.

We will answer questions and doubts as they arise while you work on your own.