KNOWLEDGE-BASED COMPUTATIONAL INTELLIGENCE AND DATA MINING AND BIOMEDICINE

Associative Structures, Neural Networks, Knowledge-based Learning and Inferences





Adrian Horzyk

horzyk@agh.edu.pl

AGH

AGH University of Science and Technology
Krakow, Poland

Associative?



Associative processes can be observed in the human brain where input data are automatically associated and form knowledge that is used by our intelligence to conclude and make various inferences.

Associative means related ideas, images, objects, data in many different ways:

by the similarity of objects

by time succession

by place/location

by definition, composition, inclusion

by context

Knowledge-Based Inferences



Knowledge is a fundamental element of smart inferences that is produced by human intelligence.

Knowledge draws from the ability to store, use and generalize about relationships between remembered objects, their features, classes, and possible actions.

Intelligent systems (like our brains) cannot work without knowledge and knowing about the matter where they should act smartly, react context-sensitively and draw intelligent conclusions about the environment and the objects in it.

Relationships can be stored and represented by the associative systems that can automatically associate representations of objects, their features, and sequences, and allow for efficient analyses and inference about data and relationships.

Knowledge-based associative systems storing **relationships** help us to find appropriate (e.g. most similar) data, objects, and their sequences quickly and produce many valuable inferences about the data and their relationships that have not to be searched exhaustively.

Data Tables



We mostly use tables to store, organize and manage data:

	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica

but common **relationships** like identity, similarity, neighborhood, minima, maxima, or counts of duplicates must be found.

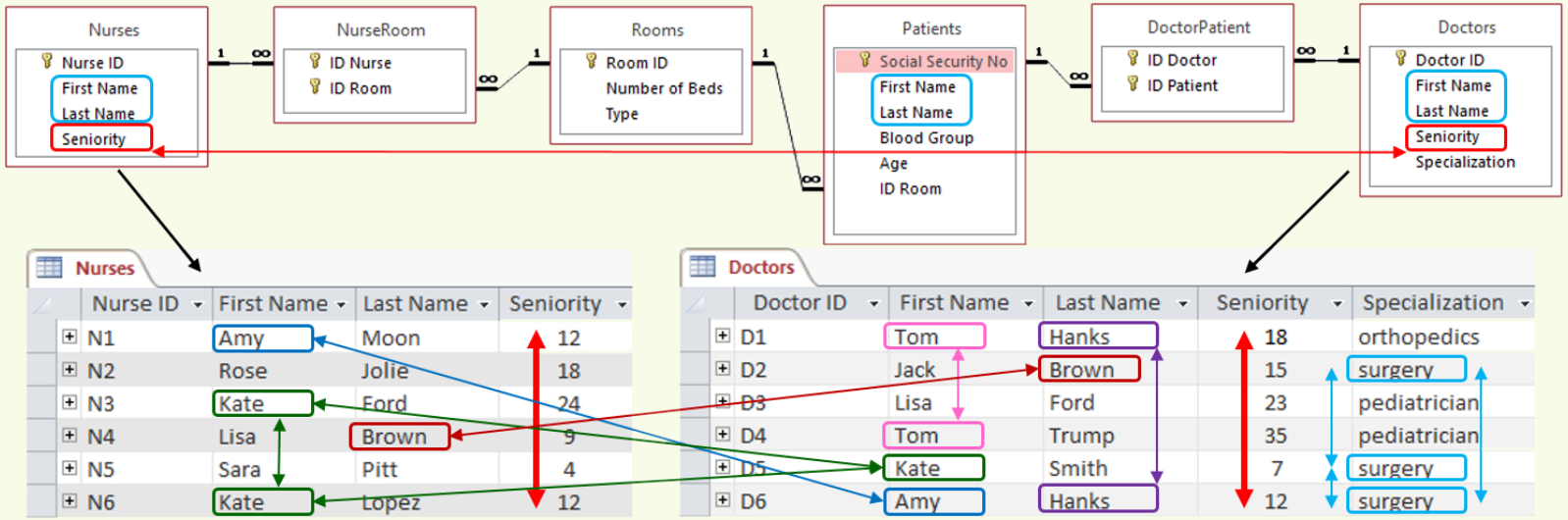
Moreover, the more data we have, the more time losses we face!

Tabular data organization does not allow us to develop brain-like knowledge-based intelligent systems that associate data together using a more abundant number of relationships which is necessary for efficient reasoning!

Relational Databases



Relational databases relate stored data only horizontally, not vertically, so we still have to search for duplicates, neighbor, or similar values and objects.



Nurses

Nurse ID	First Name	Last Name	Seniority
N1	Amy	Moon	12
N2	Rose	Jolie	18
N3	Kate	Ford	24
N4	Lisa	Brown	9
N5	Sara	Pitt	4
N6	Kate	Lopez	12

Doctors

Doctor ID	First Name	Last Name	Seniority	Specialization
D1	Tom	Hanks	18	orthopedics
D2	Jack	Brown	15	surgery
D3	Lisa	Ford	23	pediatrician
D4	Tom	Trump	35	pediatrician
D5	Kate	Smith	7	surgery
D6	Amy	Hanks	12	surgery

Even horizontally, data are not related perfectly and many duplicates of the same categories occur in various tables which are not related anyhow. In result, we need to lose a lot of computational time to search out necessary data relations to compute results or make conclusions.



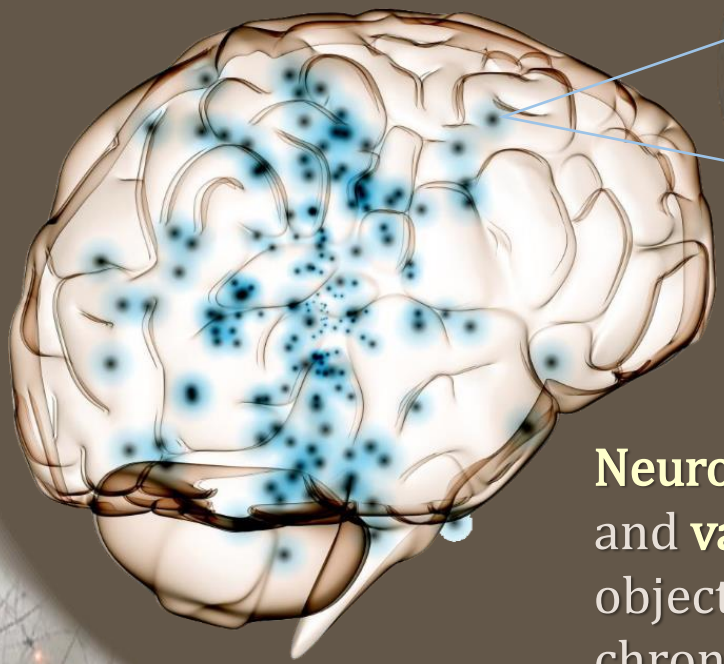
Is it wise to lose the majority of the computational time for searching for data relations?!

Brain Structures

is the main source of inspiration for developing AI!



Brains consist of complex graphs of variously connected neurons and other elements.

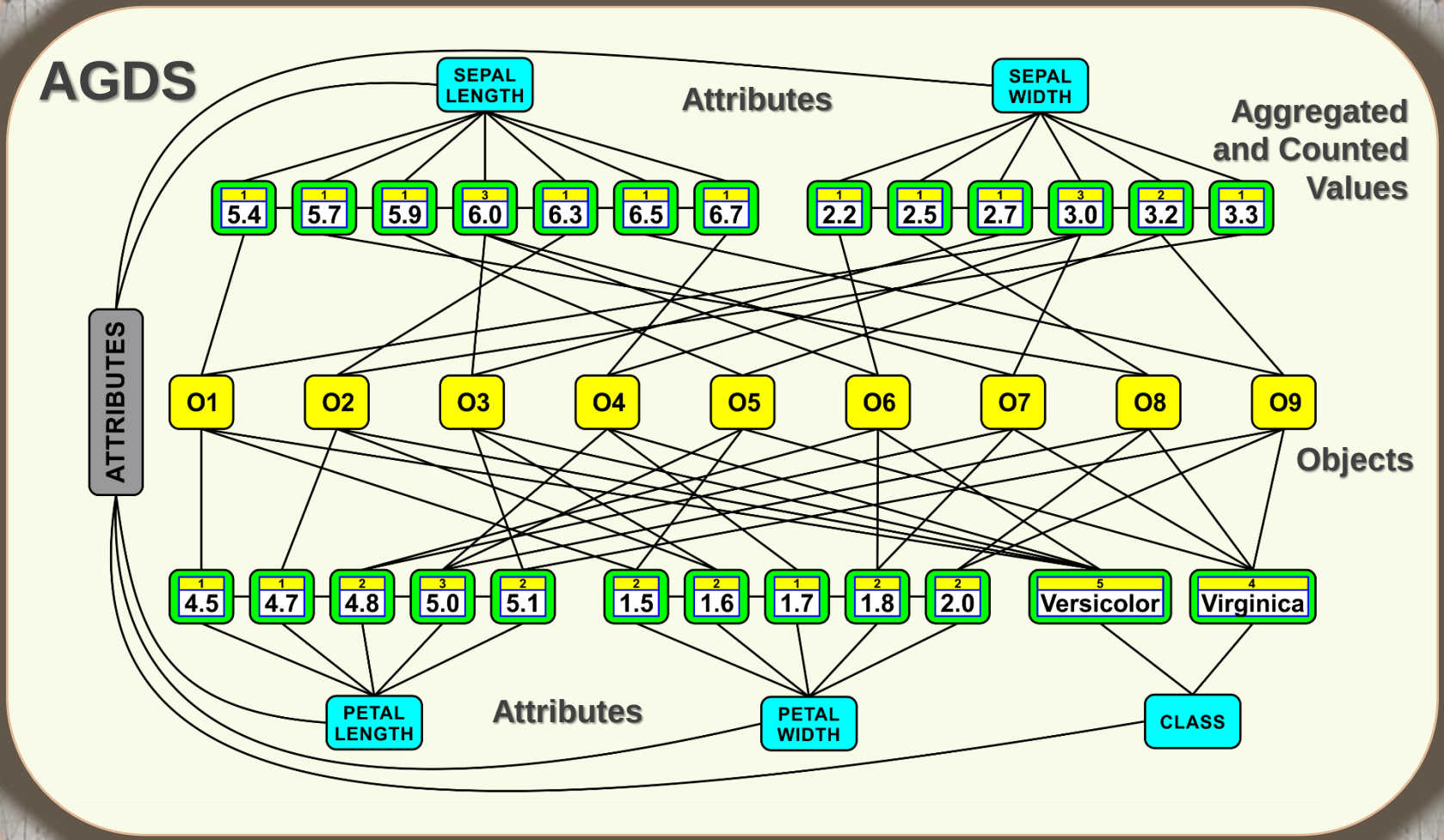




Neurons and their connections represent input data and **various relationships** between them, defining objects and similarities, proximities, sequence, chronology, context, and establishing causal relationships between them.

Why the brain structures look so complex and irregular?

AGDS
Associative Graph Data Structure



Connections represent various relations between AGDS elements like similarity, proximity, neighborhood, definition etc.

AGDS
Associative Graph Data Structure



Associative Graph Data Structures consist of:

- **Nodes** representing single-value data, ranges, subsets, objects, clusters, classes etc.
- **Edges** representing various relations between nodes like similarity, definition, sequence, neighborhood etc.

We can use it to represent any tabular data without any information loss, i.e. the transformation of tables into AGDS structure is reversible, so we can always transform back data to the tabular structure.

This transformation enriches the set of directly represented relationships between data stored in the transformed tables.

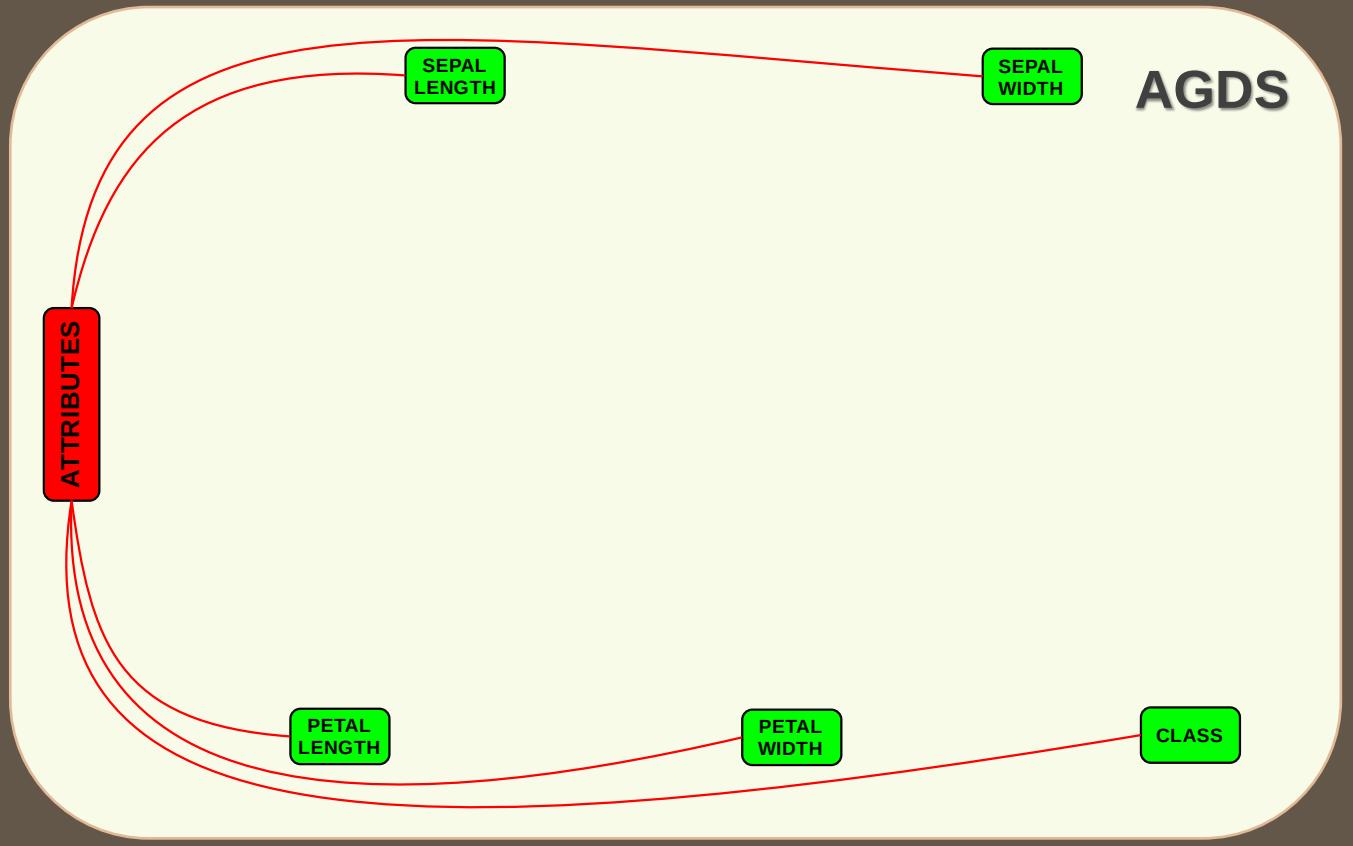
Associative Transformation



The associative transformation process of a table into an AGDS structure starts from the creation of an attributes node and the nodes representing labels of the attributes. Labels of attributes will be linked to the unique attribute values that will be sorted and counted during the insertion of next values.

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica



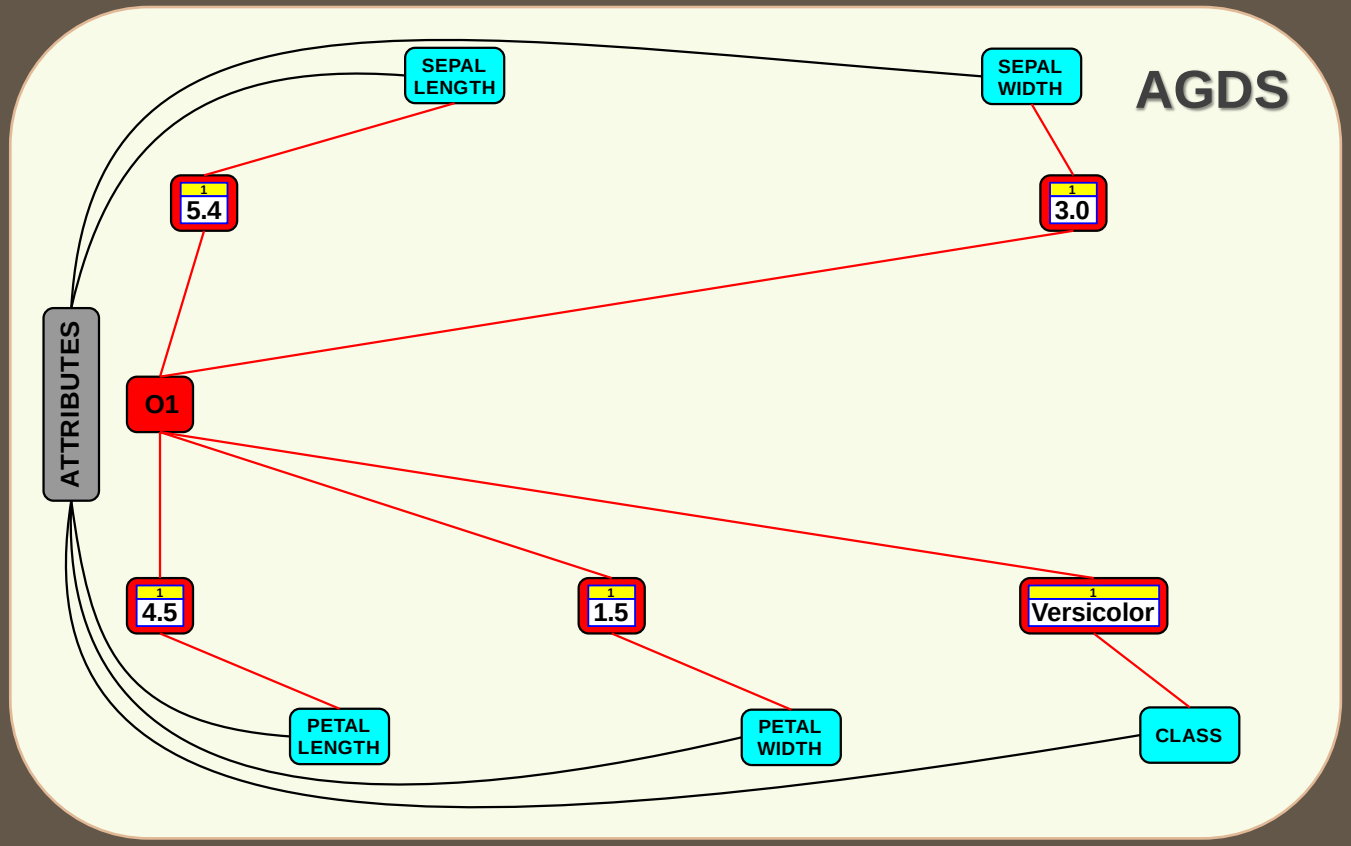


Associative Transformation



To the previously created backbone structure, the first object (record, entity) O1 is added together with all defining features.
The features and the object are connected mutually and to the label nodes of the attributes.

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica

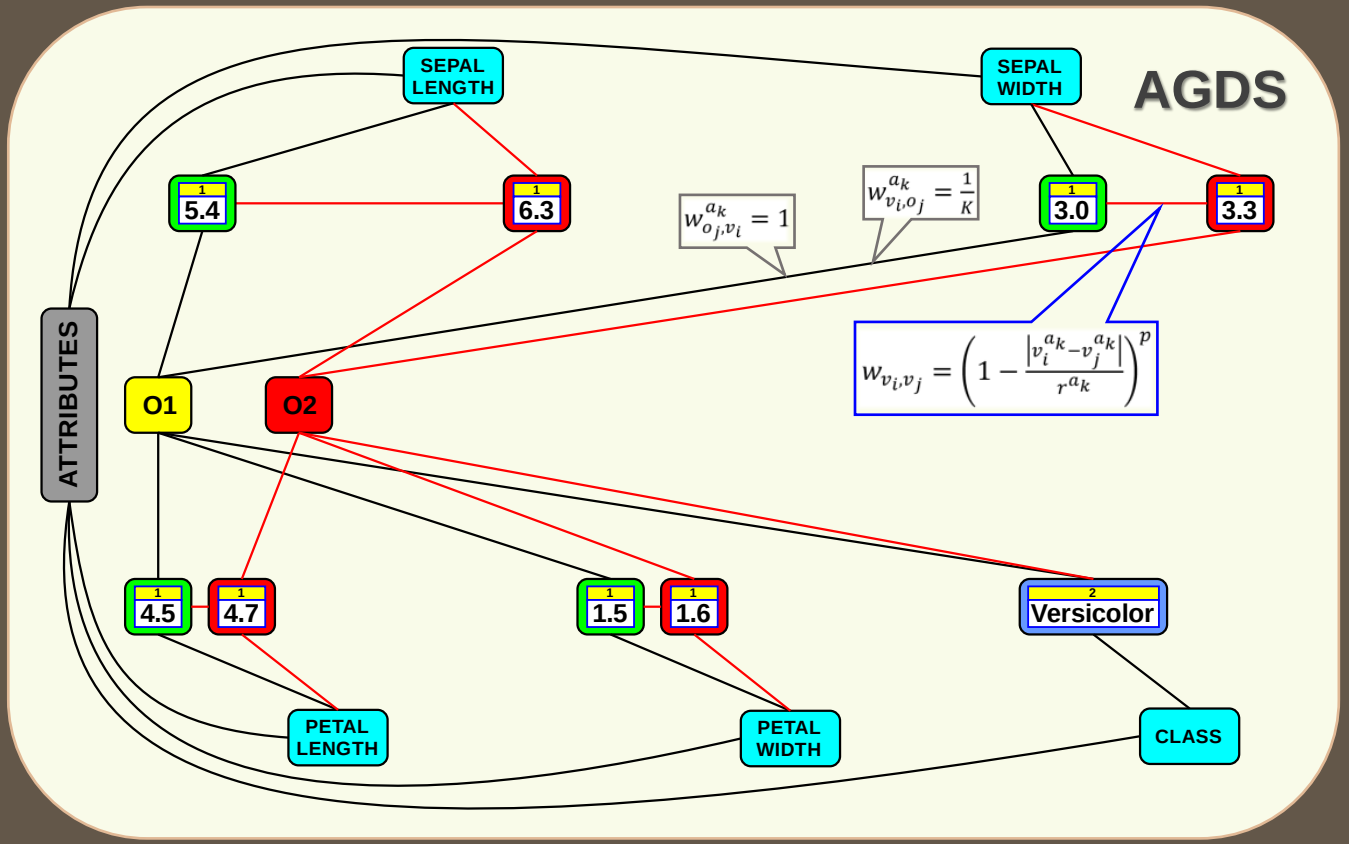


Associative Transformation



The second object is added to the AGDS structure and all its defining features are represented by values nodes that are connected to attribute labels, this new object, and neighbor values nodes which were already in this structure.

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica

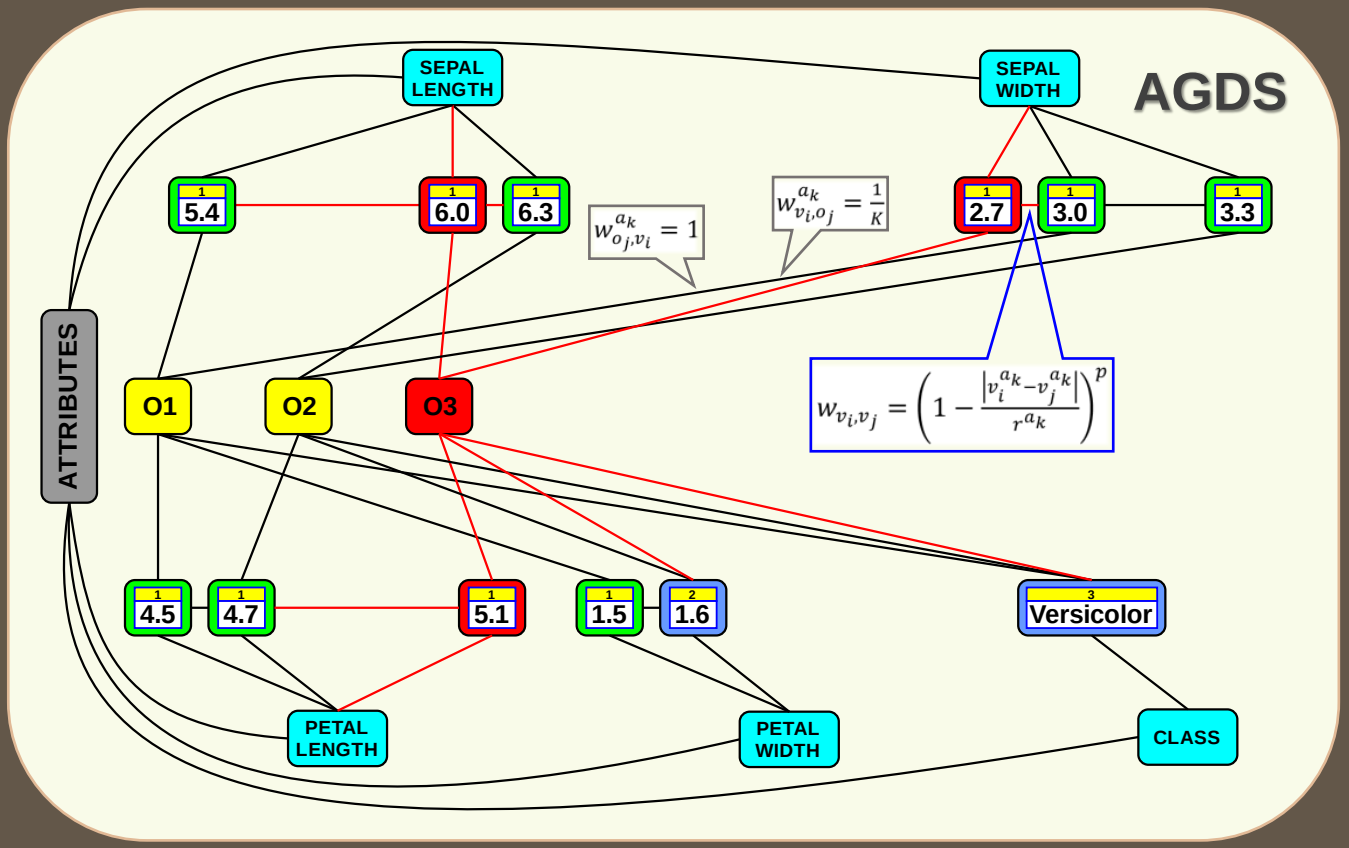


Associative Transformation



During the addition of the next object, we can notice that not all defining features have created new values nodes (e.g. 1.6 of the petal width or Versicolor of a class label) because some values had been already represented in this structure, so the duplicates (in blue) have been aggregated and counted.

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica

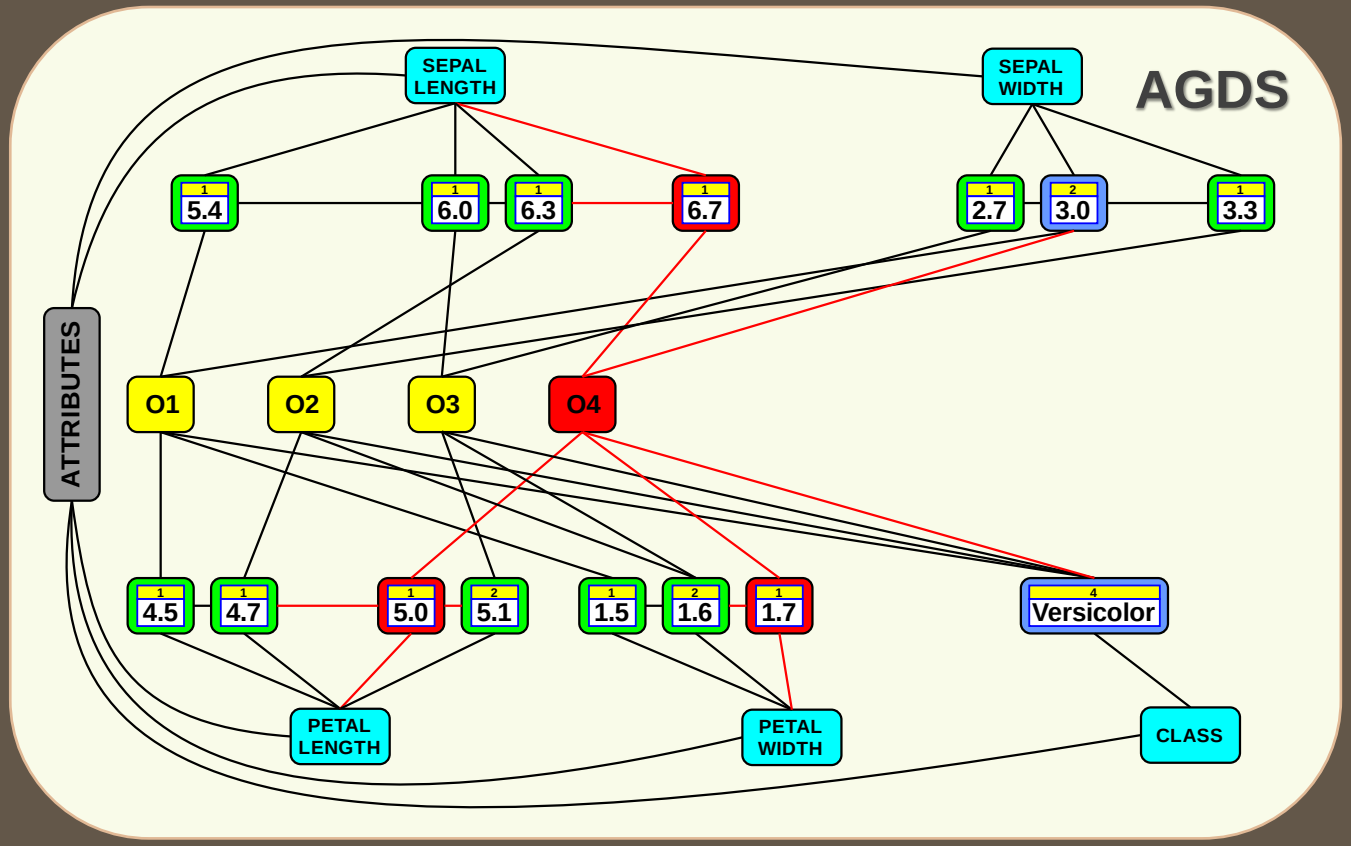


Associative Transformation



The following object creates some new values nodes and uses two of the existing values nodes, incrementing their counters of aggregated duplicates. The aggregation process od duplicates is very important from the knowledge representation point of view because it allows to draw deeper conclusions.

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica



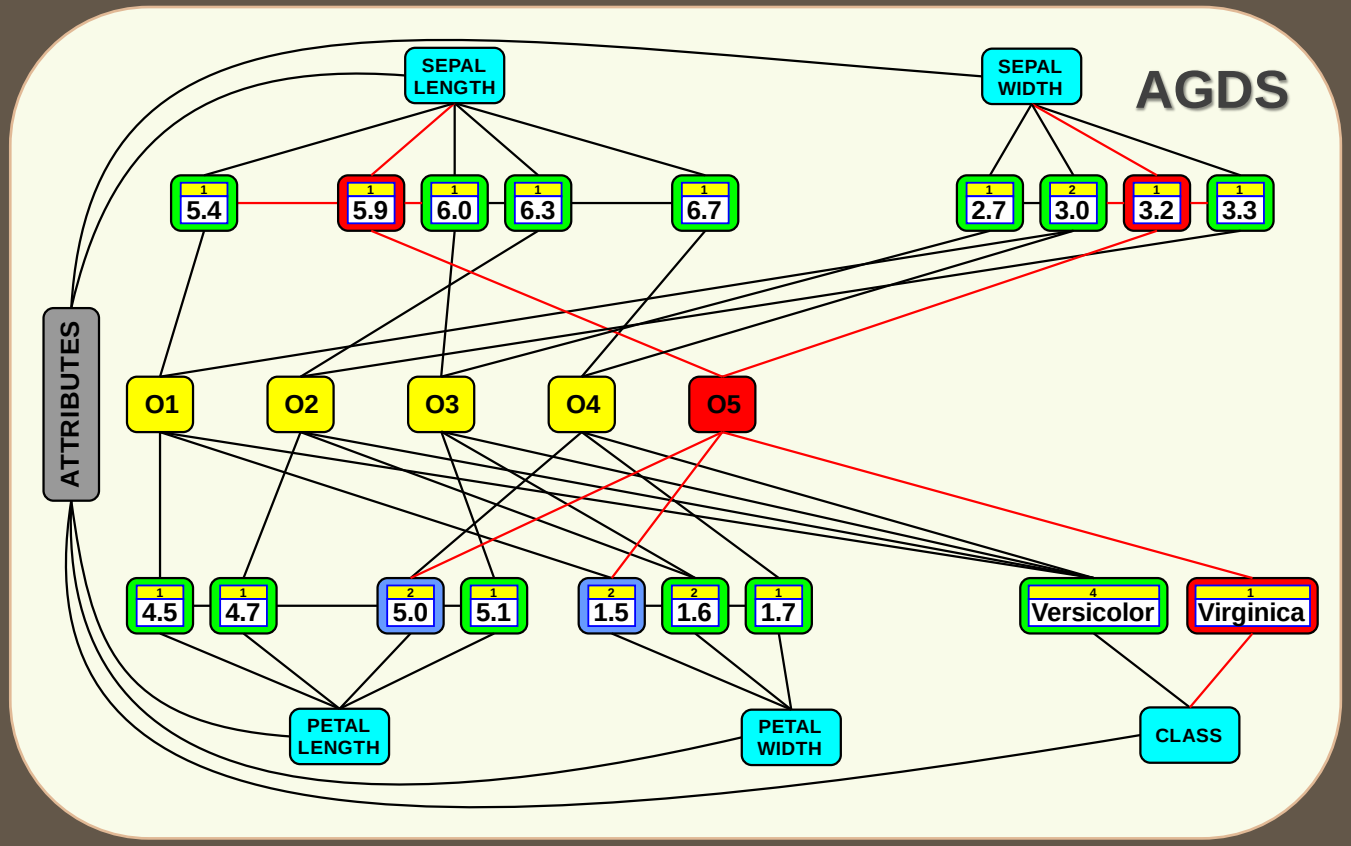
Associative Transformation



Object O5 represents a different (new) class Virginica, so a new node representing this class has been added. Notice, that symbolic (non-numerical) values are not connected as numerical features that are always connected to their neighbors and the connections are weighted.

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica





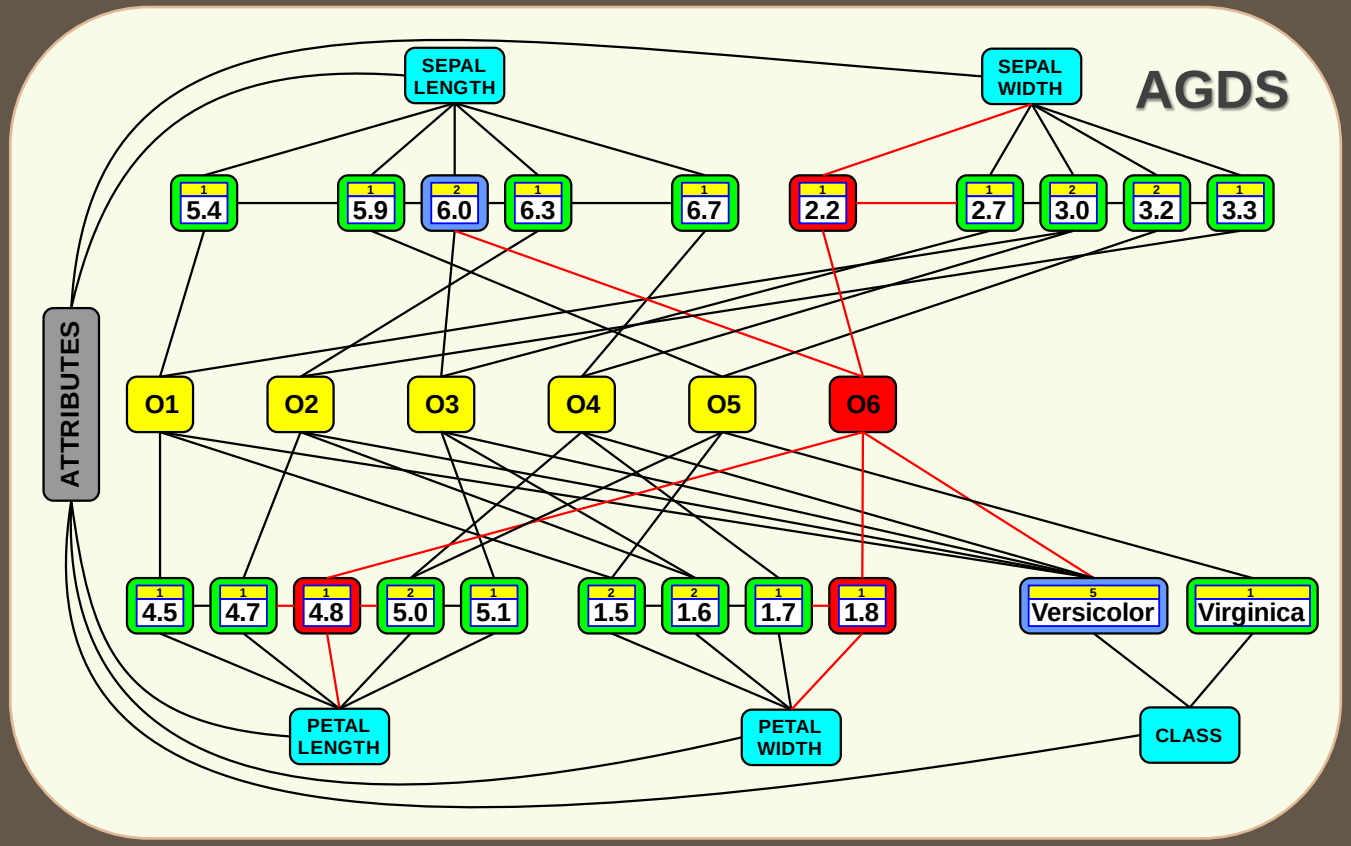
Associative Transformation



The more objects we add to this structure, the less number of new values nodes are added when the transformed table (dataset) contains duplicates. All object nodes connected to the mutually connected values nodes to other object nodes automatically create indirect associations between such objects.

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica



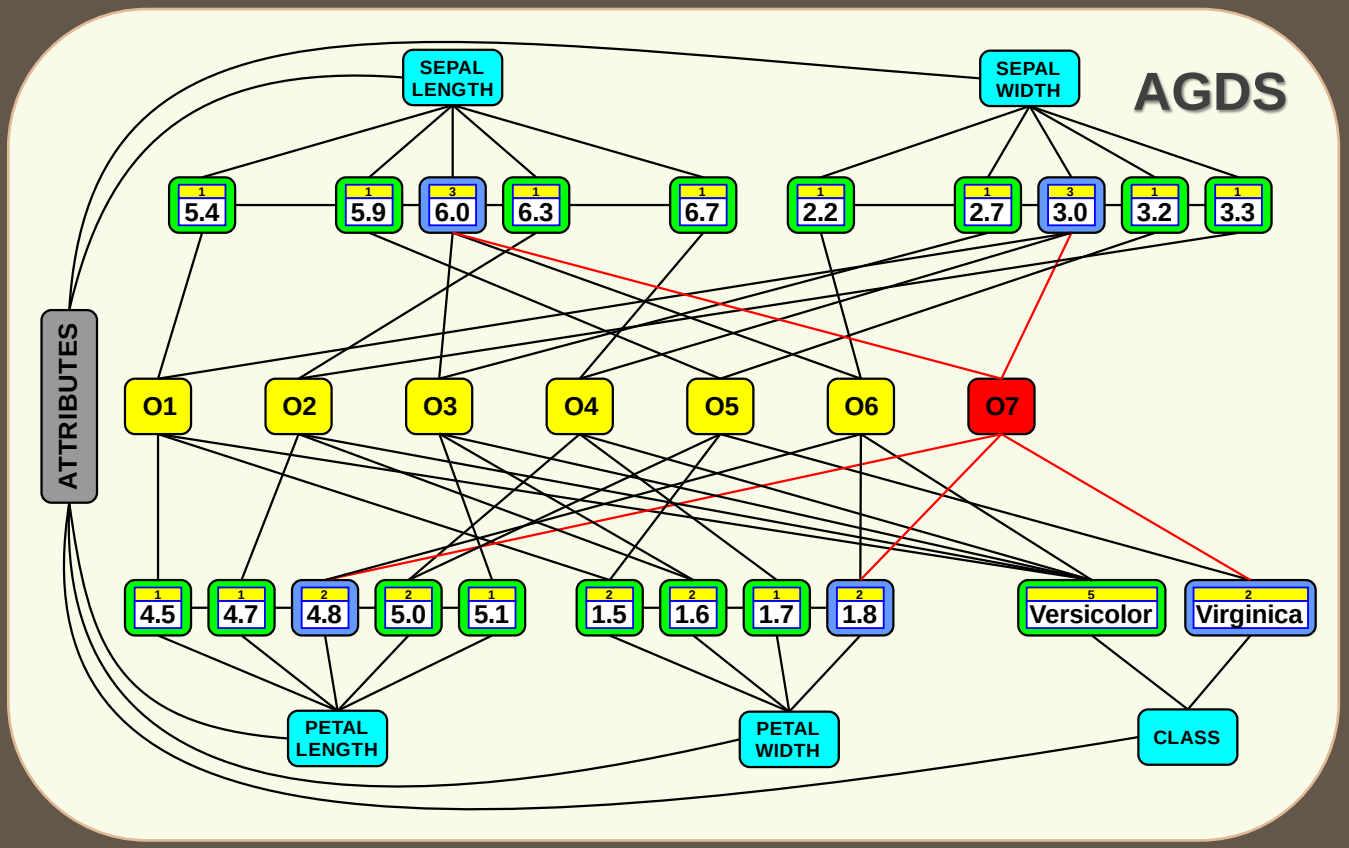


Associative Transformation



In this case, object O7 is added without addition of any new values nodes because all of them have been already added to this structure, so only new connections to the existing nodes are added, and their counters of represented duplicates are incremented. It saves memory when there are many duplicates!

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica

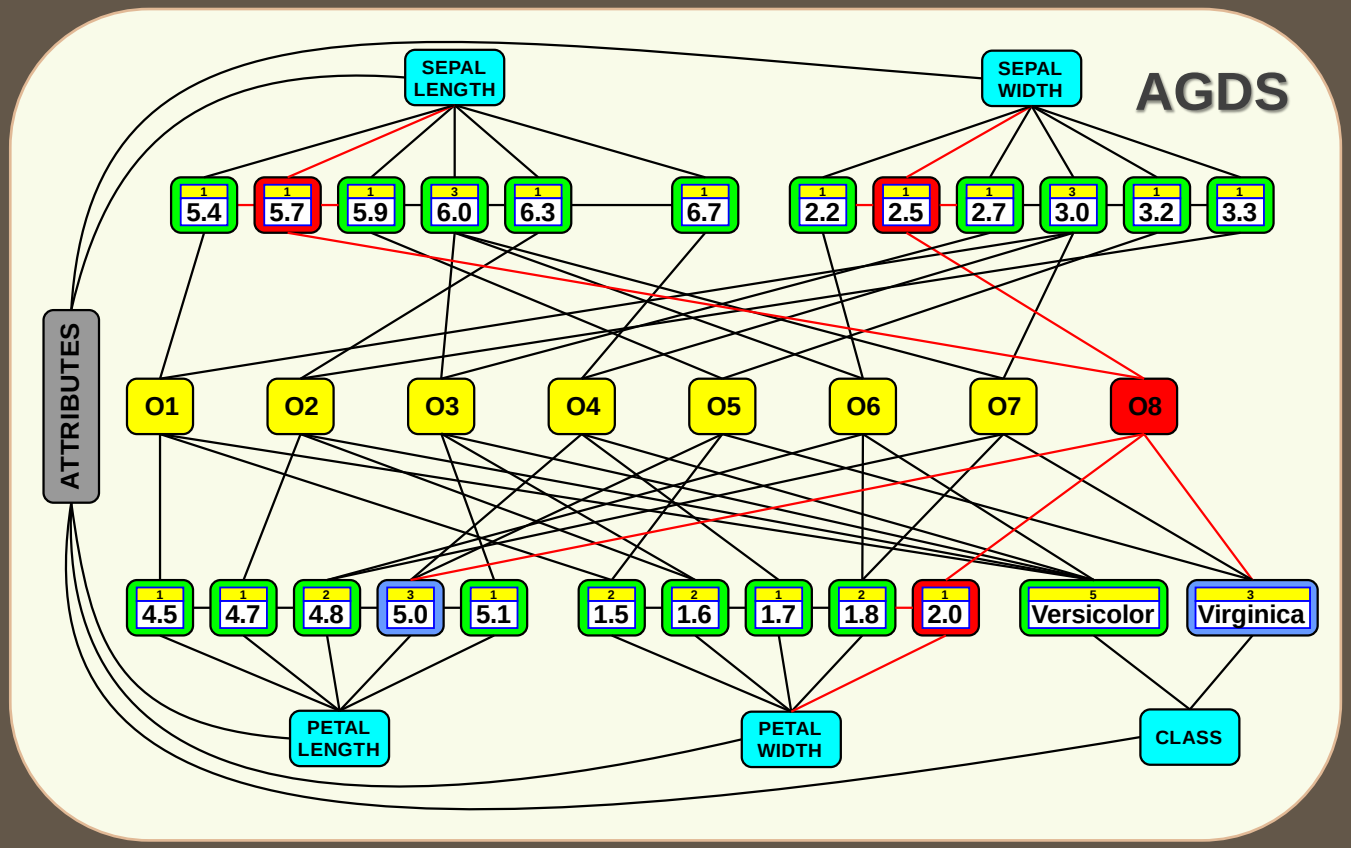


Associative Transformation



Object O8 is also connected to the values node 5.0 which now defines five objects (O4, O5 and O8), so there is a visible similarity between these objects. The similarity between objects O5 and O8 is bigger than between O4 and O8 because there is another shared feature (Virginica) between the first pair!

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica



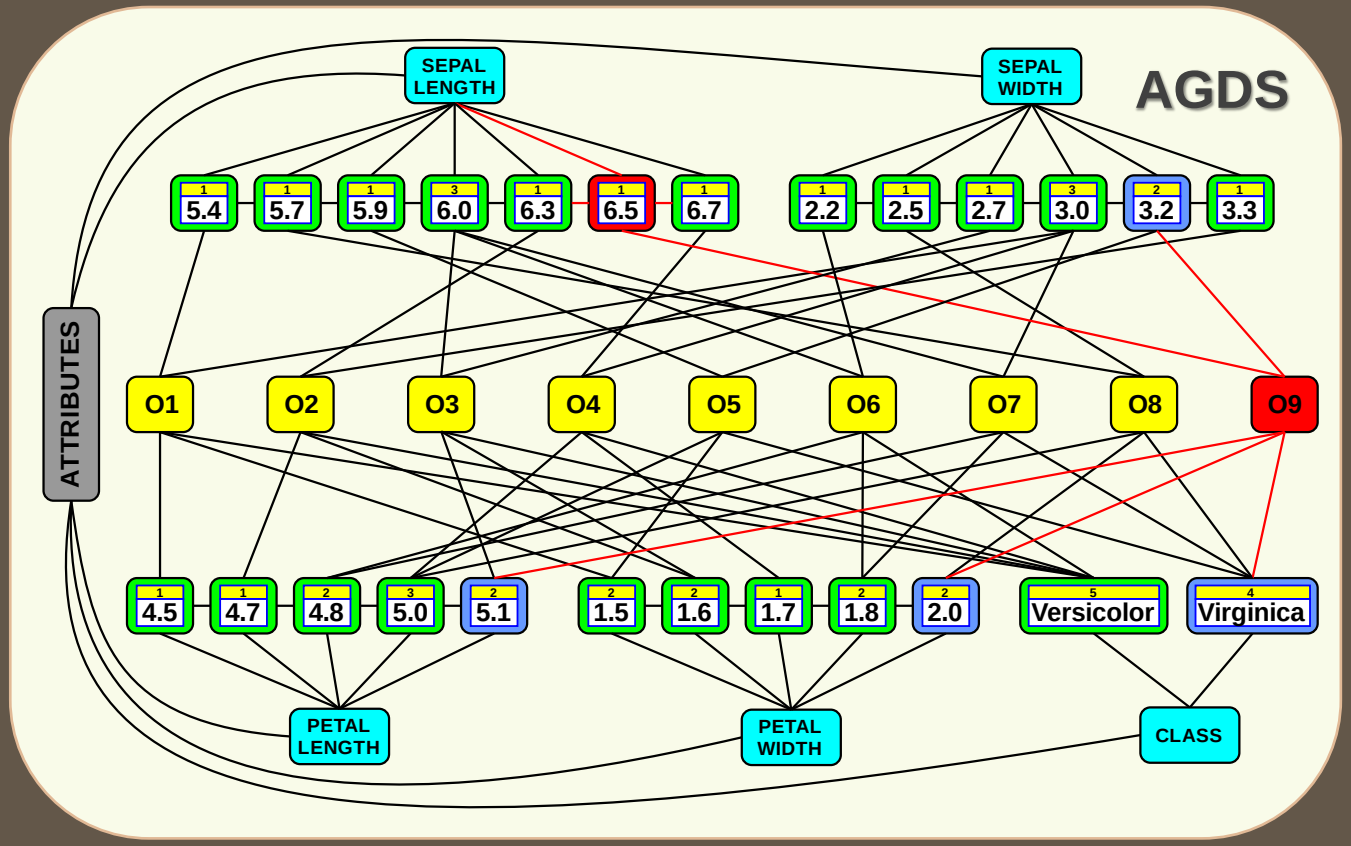
Associative Transformation



Object O9 has added only one new feature to this structure because the other feature values had been already represented.
Now, the transformation process for this small table is already finished,
and we can try to compare these structures and take advantages of this graph!

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica



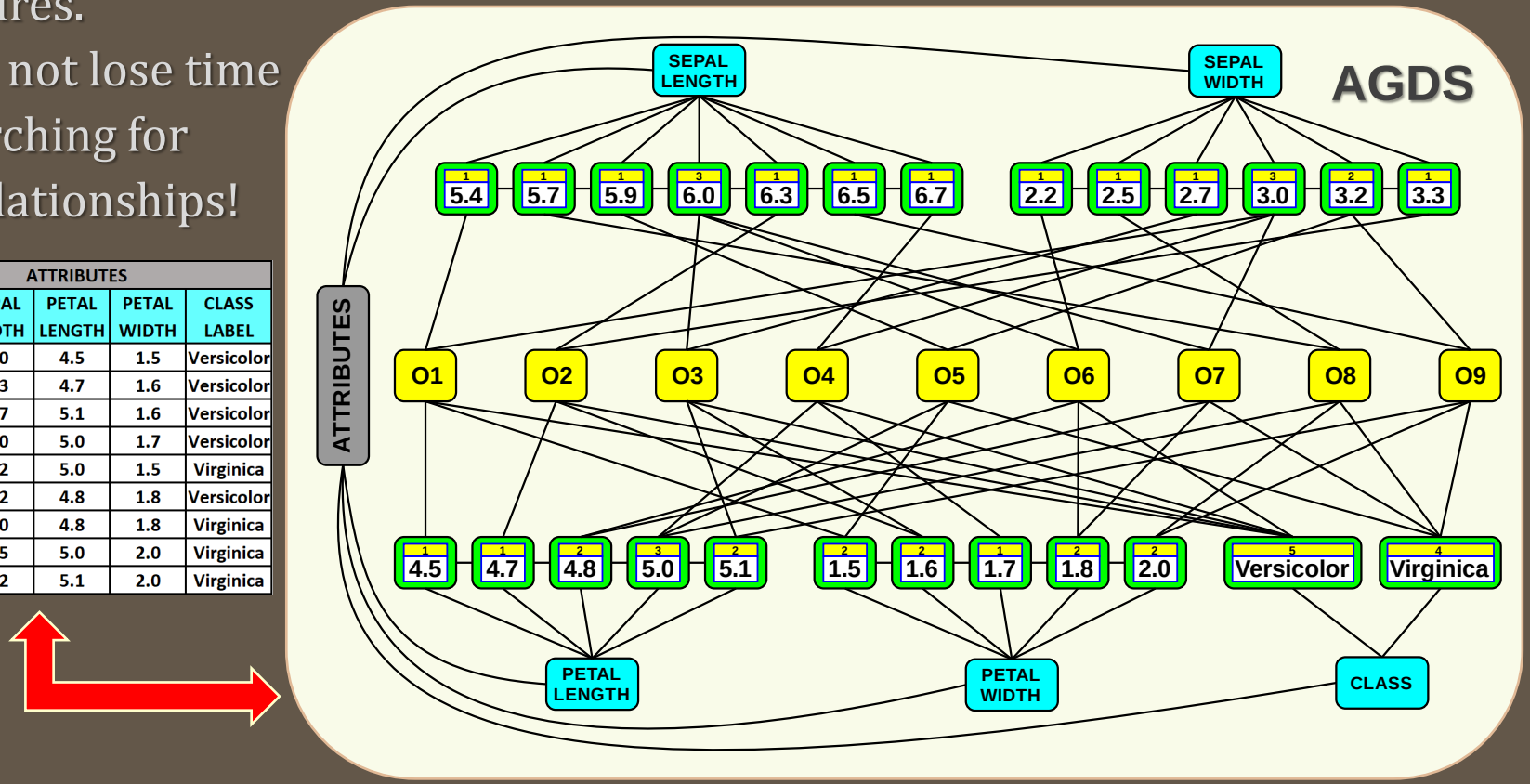


Comparison of Structures



Which structure of the two presented do you like more?
The tabular structure represents data and very basic relations between them.
The AGDS structure additionally represents neighborhood, order, similarity, minima, maxima, counts of duplicates, number of unique values, and ranges of all features.
We will not lose time for searching for such relationships!

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica

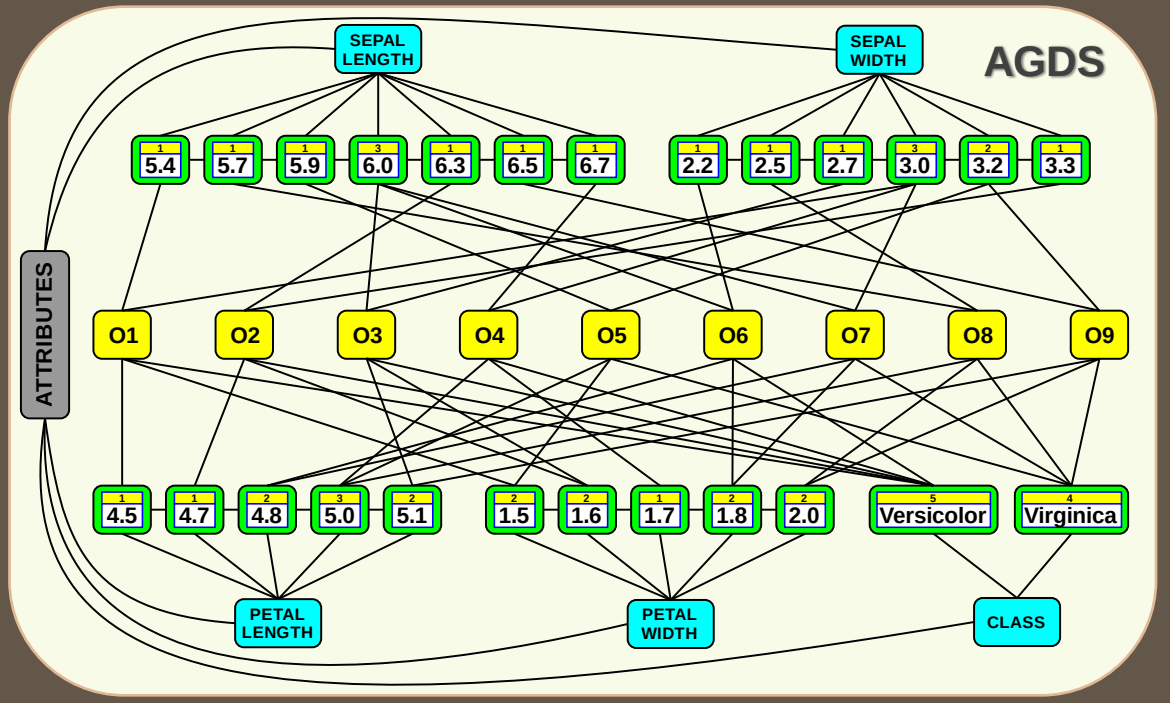


Alternative Construction of AGDS



We can create this structure in an alternative way when the dataset (table) is static and does not change in time (no records are added, removed or updated).

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica



CAT OFF AND SEPARATE DATA FOR EACH ATTRIBUTE SEPARATELY

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica

SORT DATA FOR EACH ATTRIBUTE SEPARATELY

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	2.2	4.5	1.5	Versicolor
O2	5.7	2.5	4.7	1.5	Versicolor
O3	5.9	2.7	4.8	1.6	Versicolor
O4	6.0	3.0	4.8	1.6	Versicolor
O5	6.0	3.0	5.0	1.7	Versicolor
O6	6.0	3.0	5.0	1.8	Virginica
O7	6.3	3.2	5.0	1.8	Virginica
O8	6.5	3.2	5.1	2.0	Virginica
O9	6.7	3.3	5.1	2.0	Virginica

REMOVE DUPLICATES OF ALL ATTRIBUTES SEPARATELY

DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	2.2	4.5	1.5	Versicolor
O2	5.7	2.5	4.7	1.6	Virginica
O3	5.9	2.7	4.8	1.7	
O4	6.0	3.0	5.0	1.8	
O5	6.3	3.2	5.1	2.0	
O6	6.5	3.3			
O7	6.7				
O8					
O9					

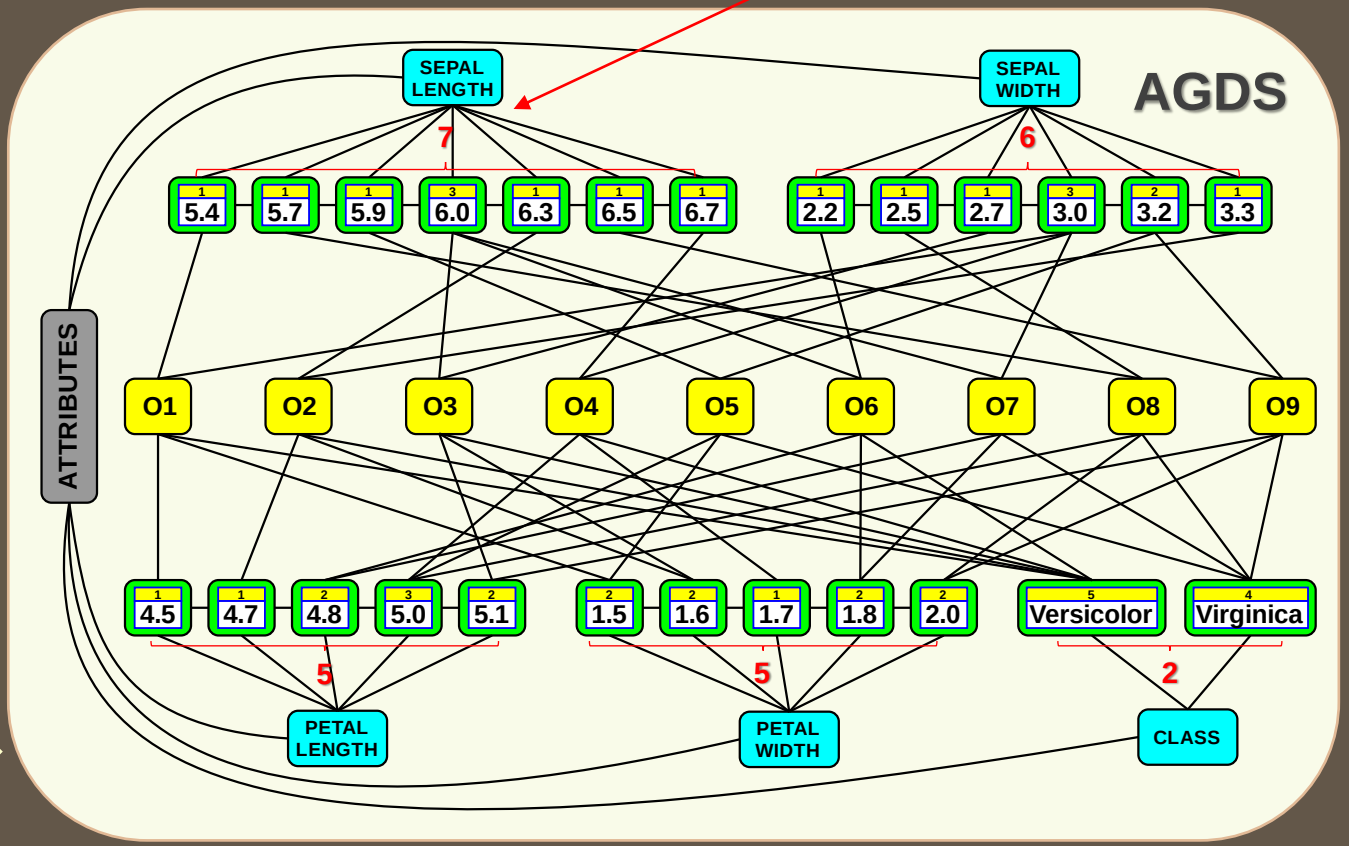
Efficiency of Data Access



Features of each attribute can be organized using: sorted tables, sorted lists, hash tables or AVB+trees to provide quick access to them!

Notice, that the number of unique features for each attribute is less or equal to the number of all features in the dataset (table).

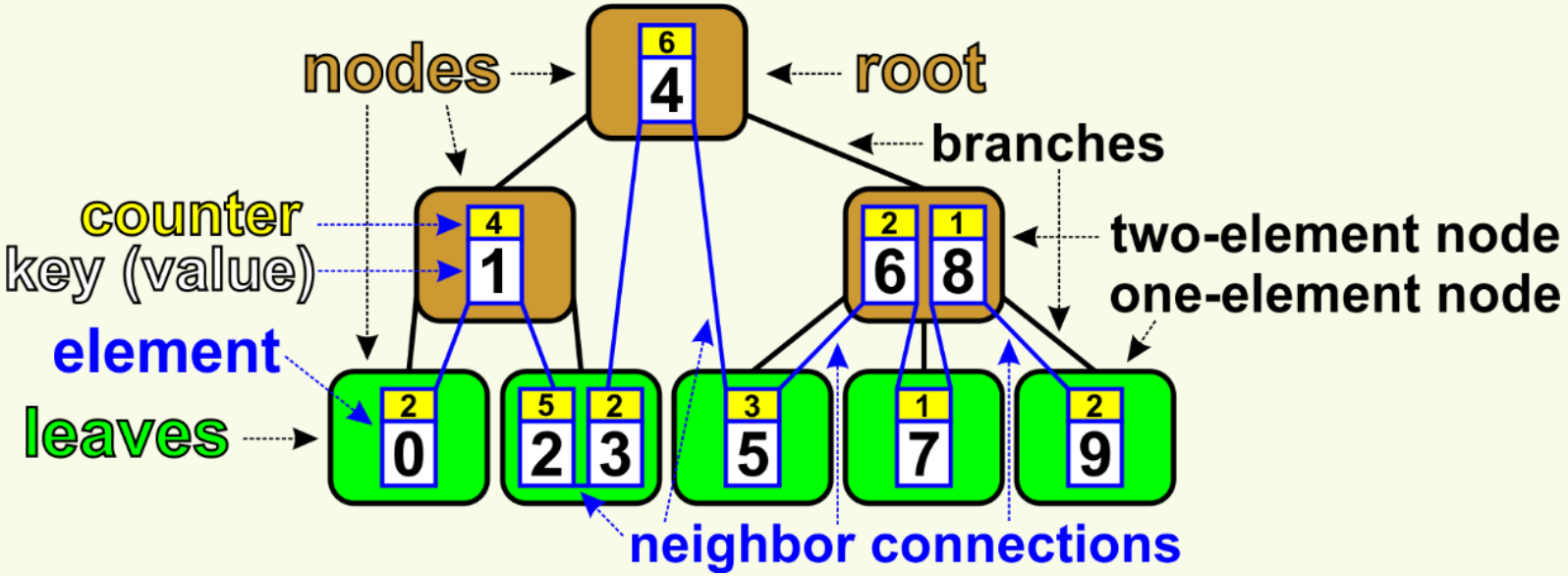
DATASET	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5 9	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica



AVB+Trees
Sorting Aggregated-Value B-Trees



An AVB+tree is a hybrid structure that represent sorted list of elements which are quickly accessed via self-balancing B-tree structure. Elements aggregate and count up all duplicates of represented values.



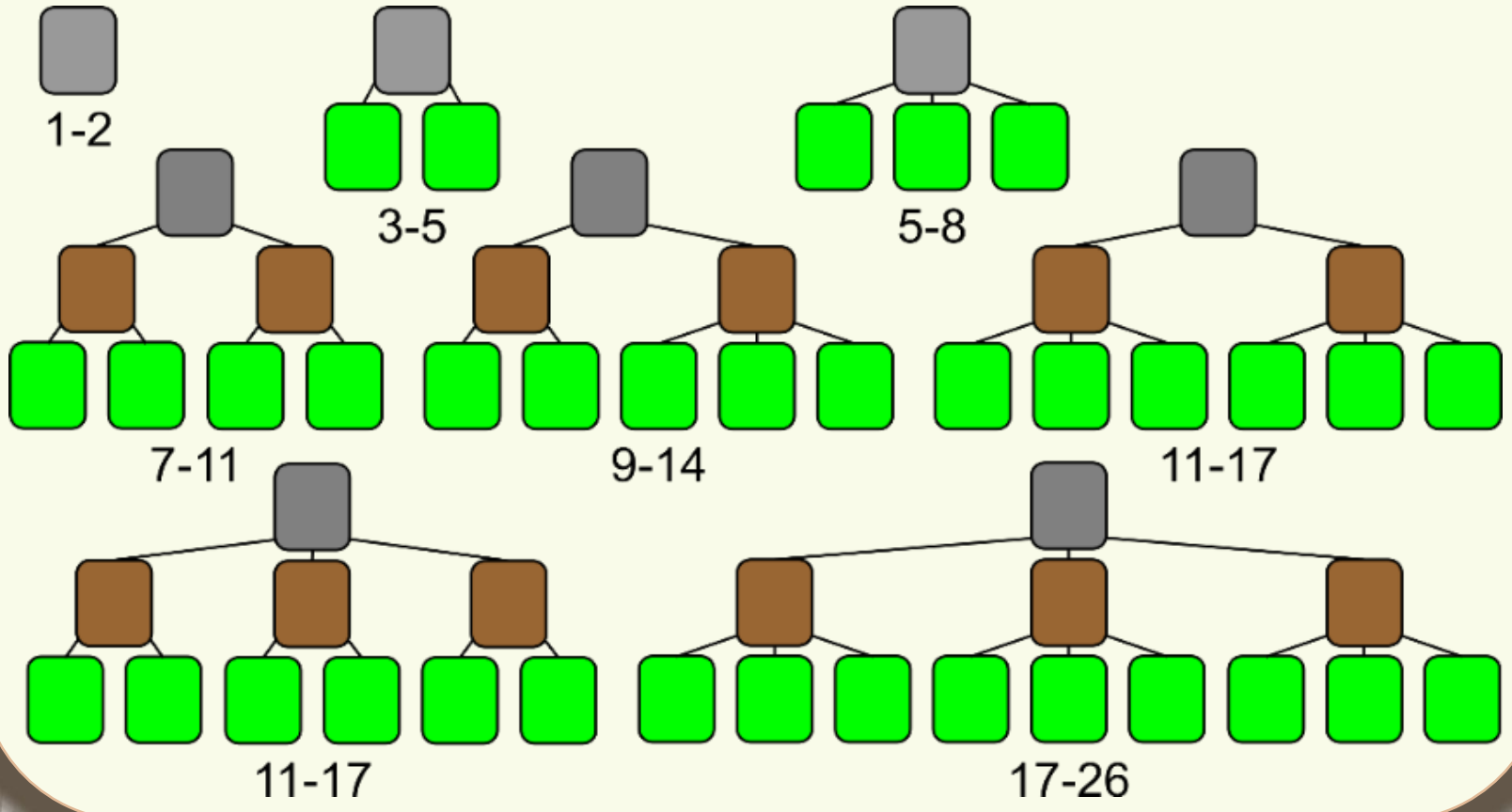
AVB+trees are typically much smaller in size and height than B-trees and B+trees thanks to the aggregations of duplicates and not using any extra internal nodes as signposts as used in B+trees.

Capacity of AVB+Trees

accelerating the speed of search in AGDS



Capacities of elements of the smallest AVB+trees.



The same number of elements can be stored by various AVB-tree structures, e.g. 11 or 17 elements!

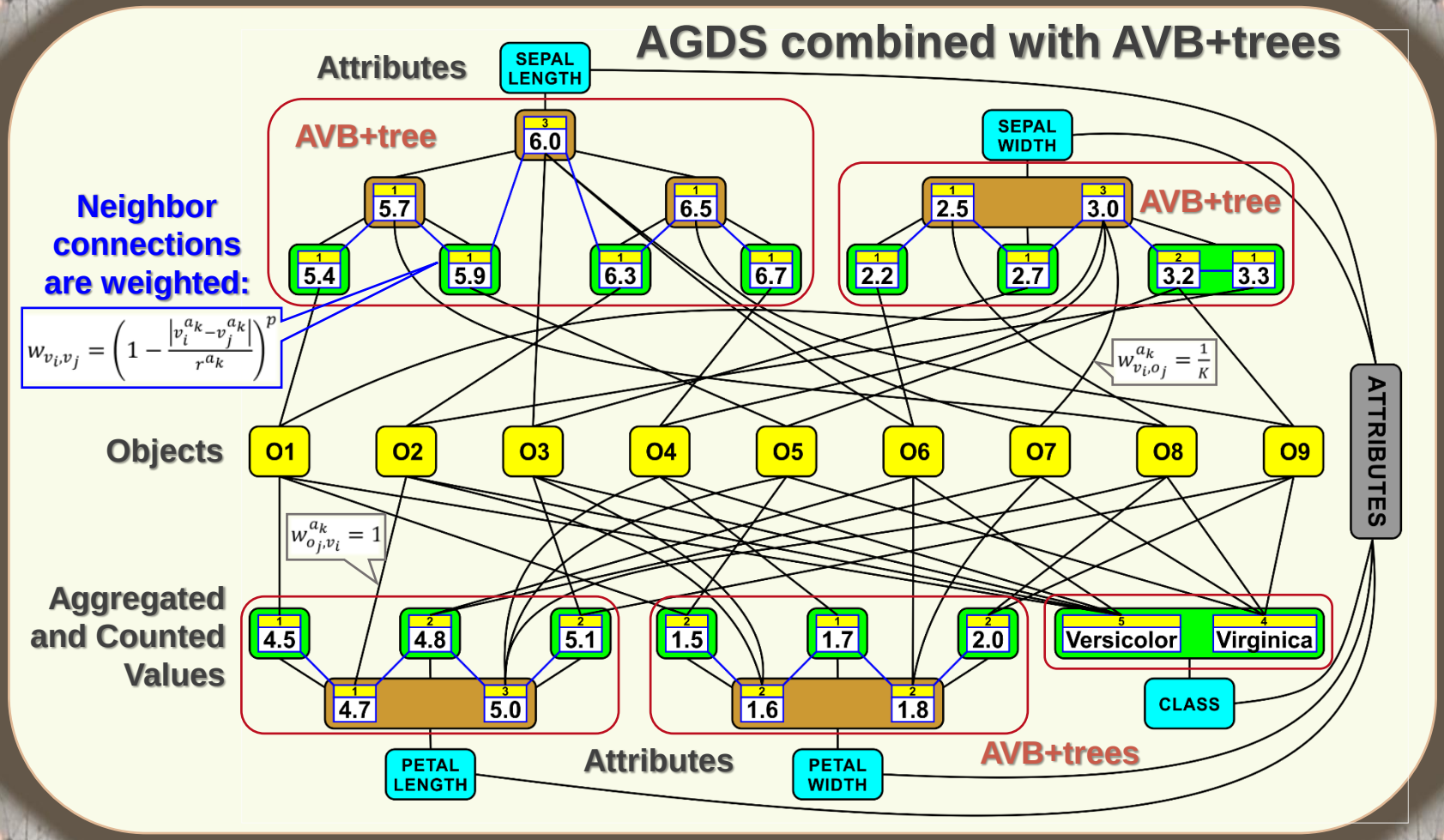
Properties of AVB+trees



- ✓ Each tree node can store one or two elements.
- ✓ Elements aggregate representations of duplicates and store counters of aggregated duplicates of values.
- ✓ Elements are connected in a sorted order, so it is possible to move between neighbor values very quickly.
- ✓ AVB+trees do not use extra nodes to organize access to the elements stored in leaves as B+trees.
- ✓ AVB+trees use all advantages of B-trees, B+trees, and AVB-trees removing their inconvenience.
- ✓ They implement common operations like Insert, Remove, Search, GetMin, GetMax, and can be used to compute Sums, Counts, Averages, Medians etc. quickly.
- ✓ They supply us with sorted lists of elements which are quickly accessible via this tree structure and thanks to the aggregations of duplicates that substantially reduce the number of elements storing values.

Efficient hybrid structure!

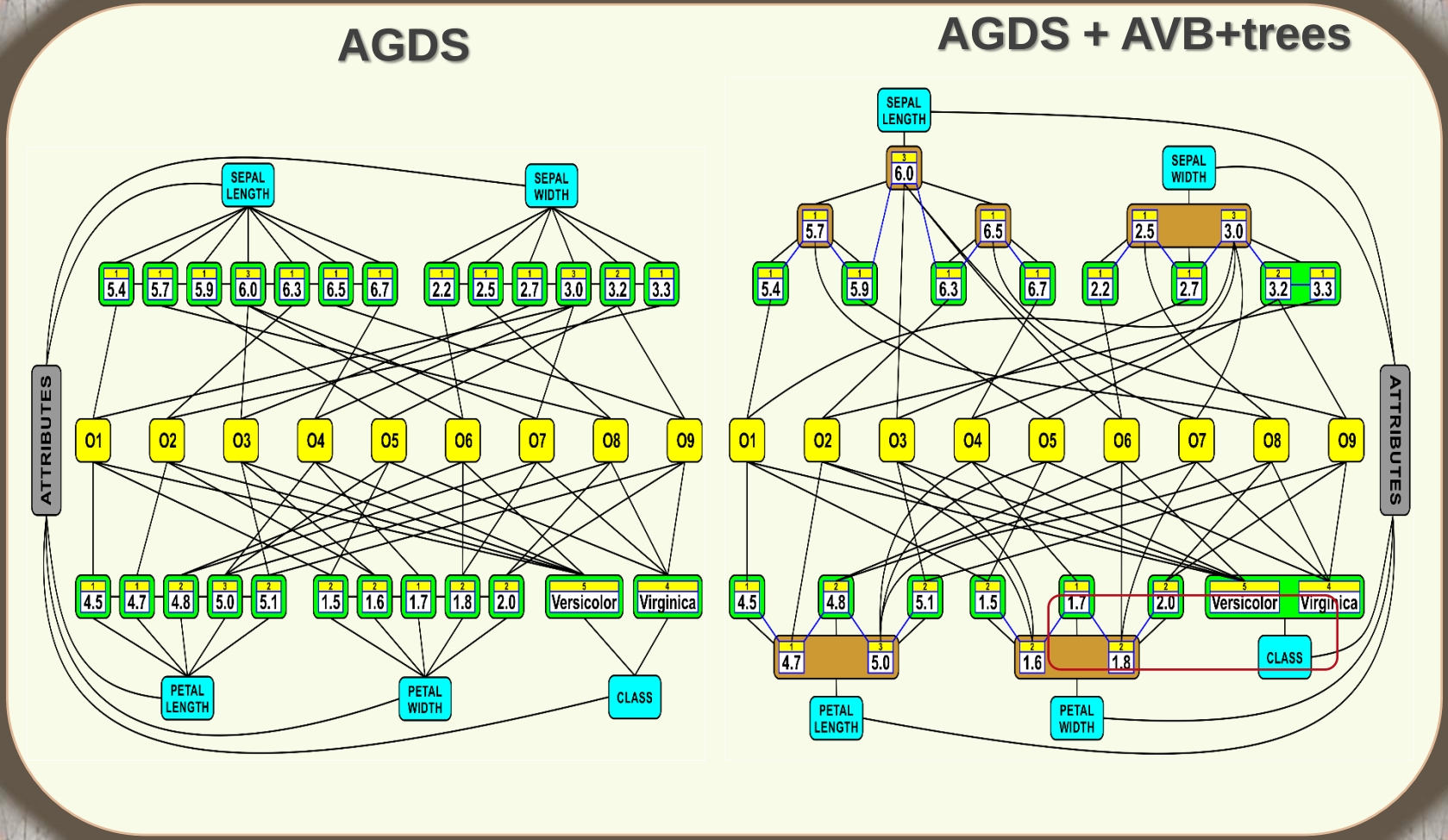
AGDS + AVB+trees
as a still more efficient solution



AVB+trees implemented to AGDS structures make the data access faster especially for Big Data datasets and databases.

Comparison of AGDS with AGDS + AVB+trees



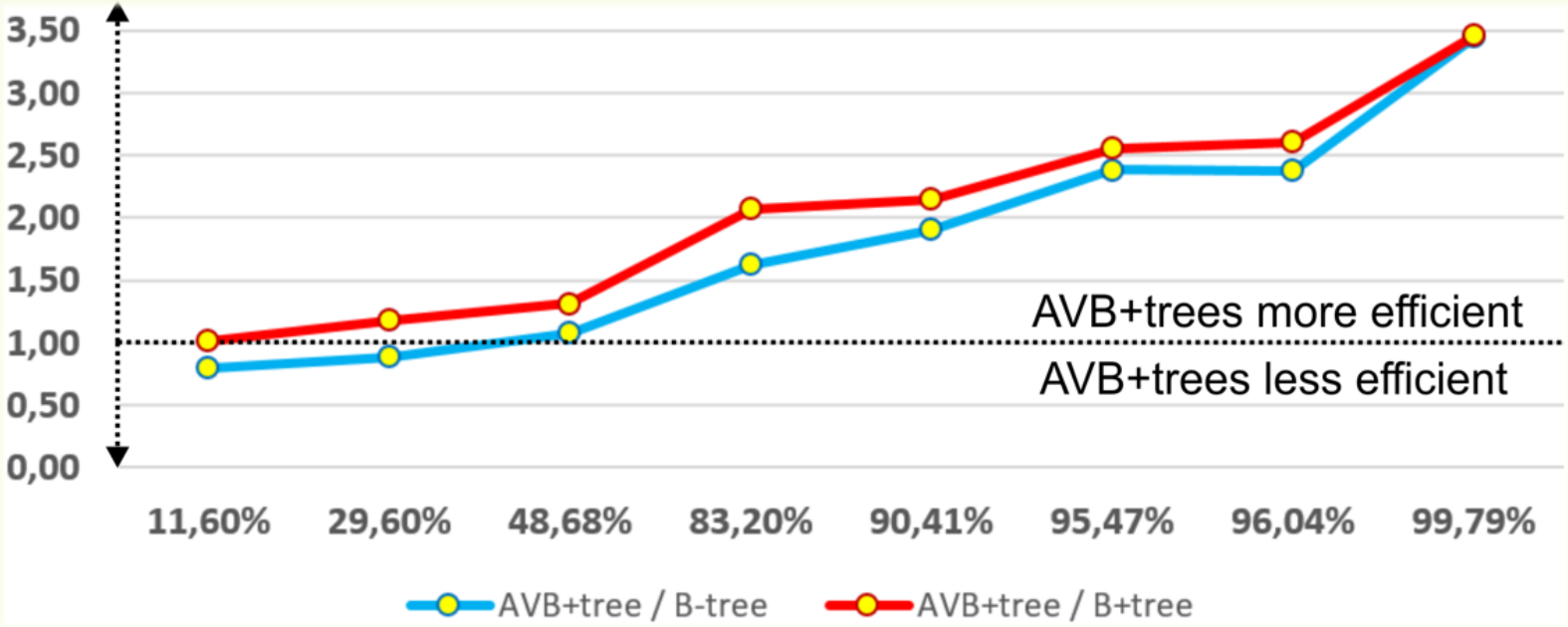


When data contain many duplicates
we practically achieve the constant access
to all data stored in AGDS + AVB+trees.

Comparisons of Efficiencies



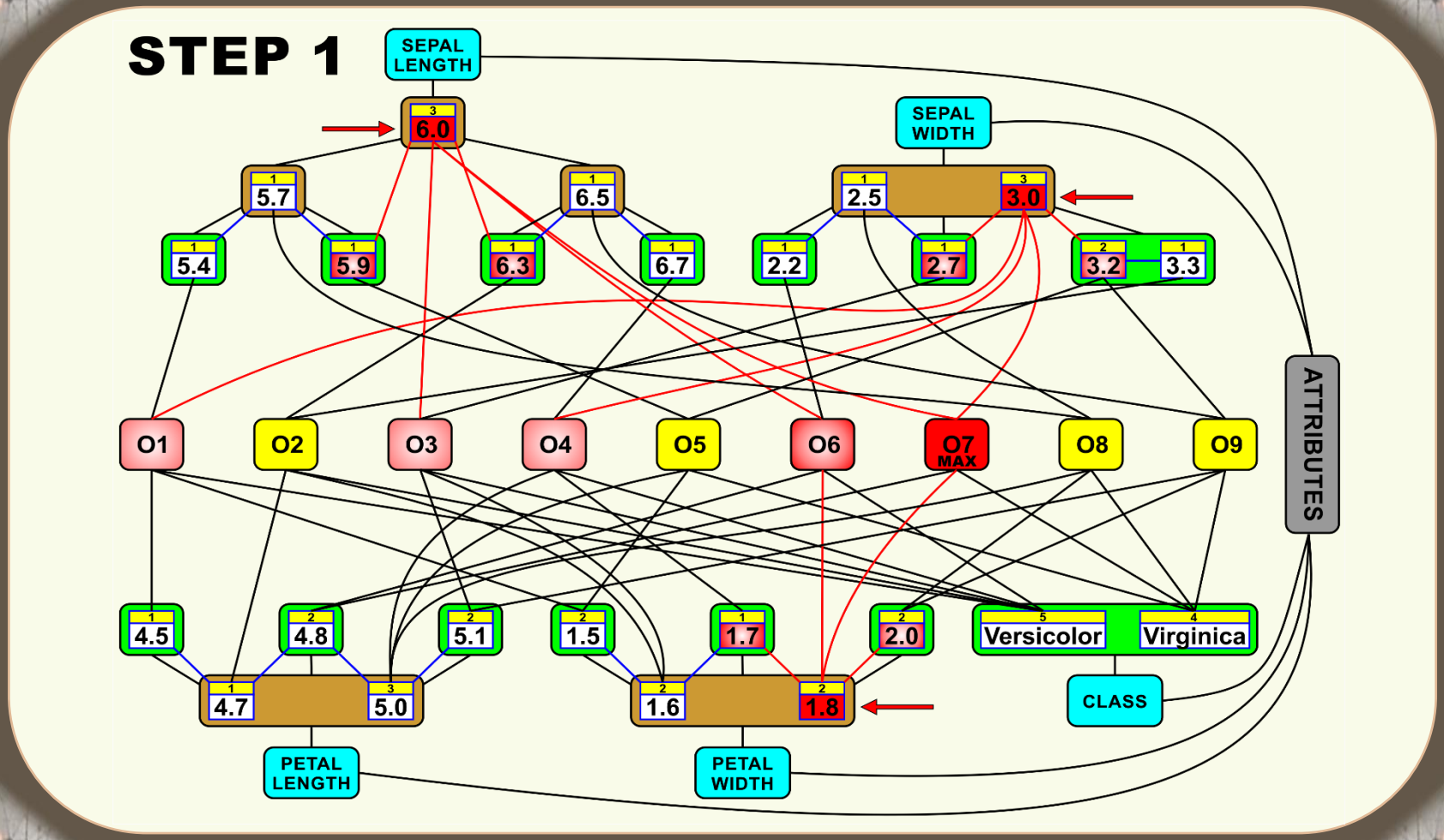
The efficiencies of the same operations on the same datasets from UCI ML Repository were compared on B-trees, B+trees, AVB-trees, and AVB+trees.



The achieved results proved the concept that AVB+trees are always faster than B+trees commonly used in databases, and AVB-trees are usually faster than B-trees when data contain more than 30% of duplicates.

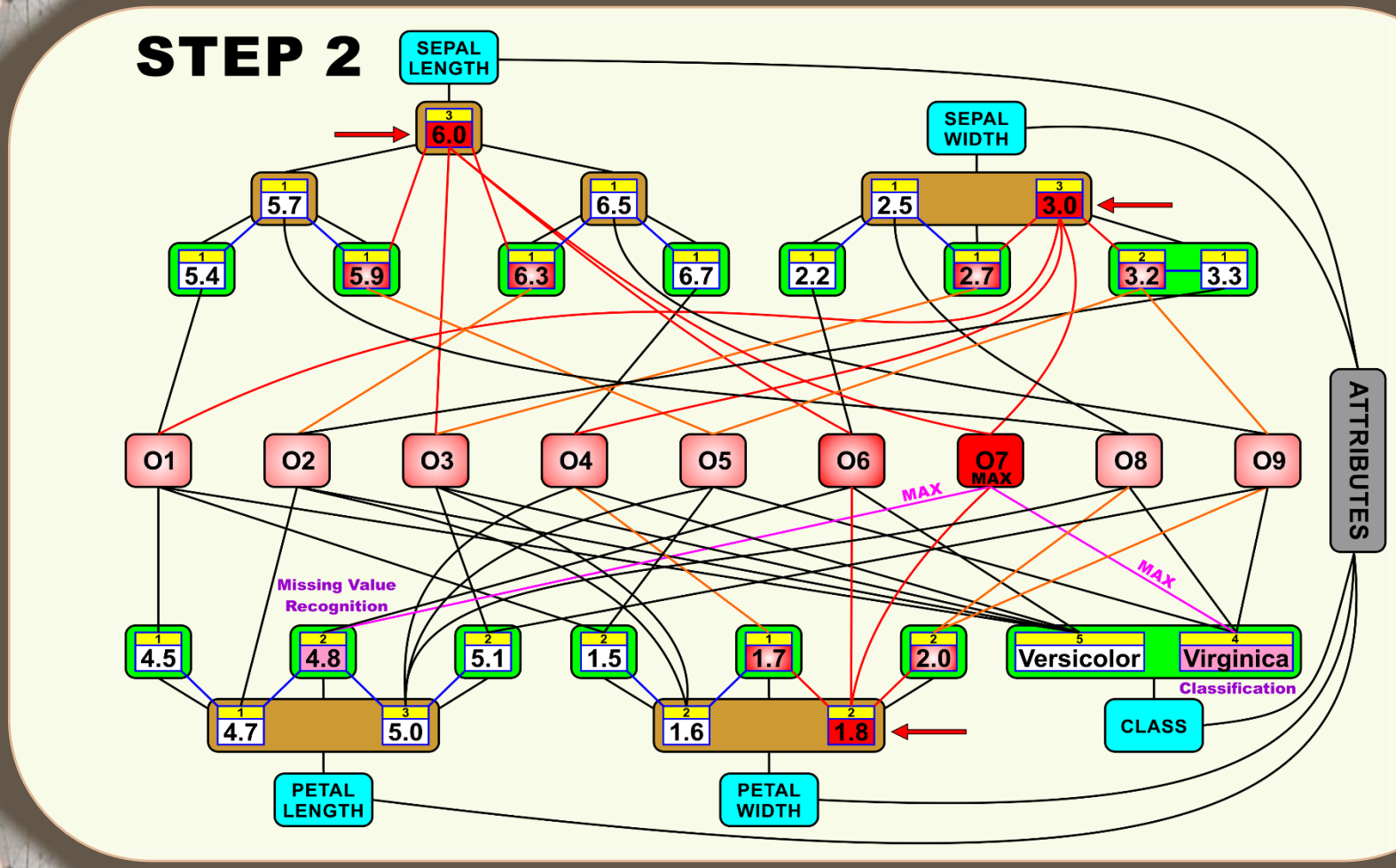
AVB-trees and AVB+trees outperform commonly used B-trees and B+trees in most cases!

Inferences on AGDS combined with AVB+trees



We do not need to search for common relations in many (nested) loops but we simply go along the connections and get results.

Inferences on AGDS combined with AVB+trees



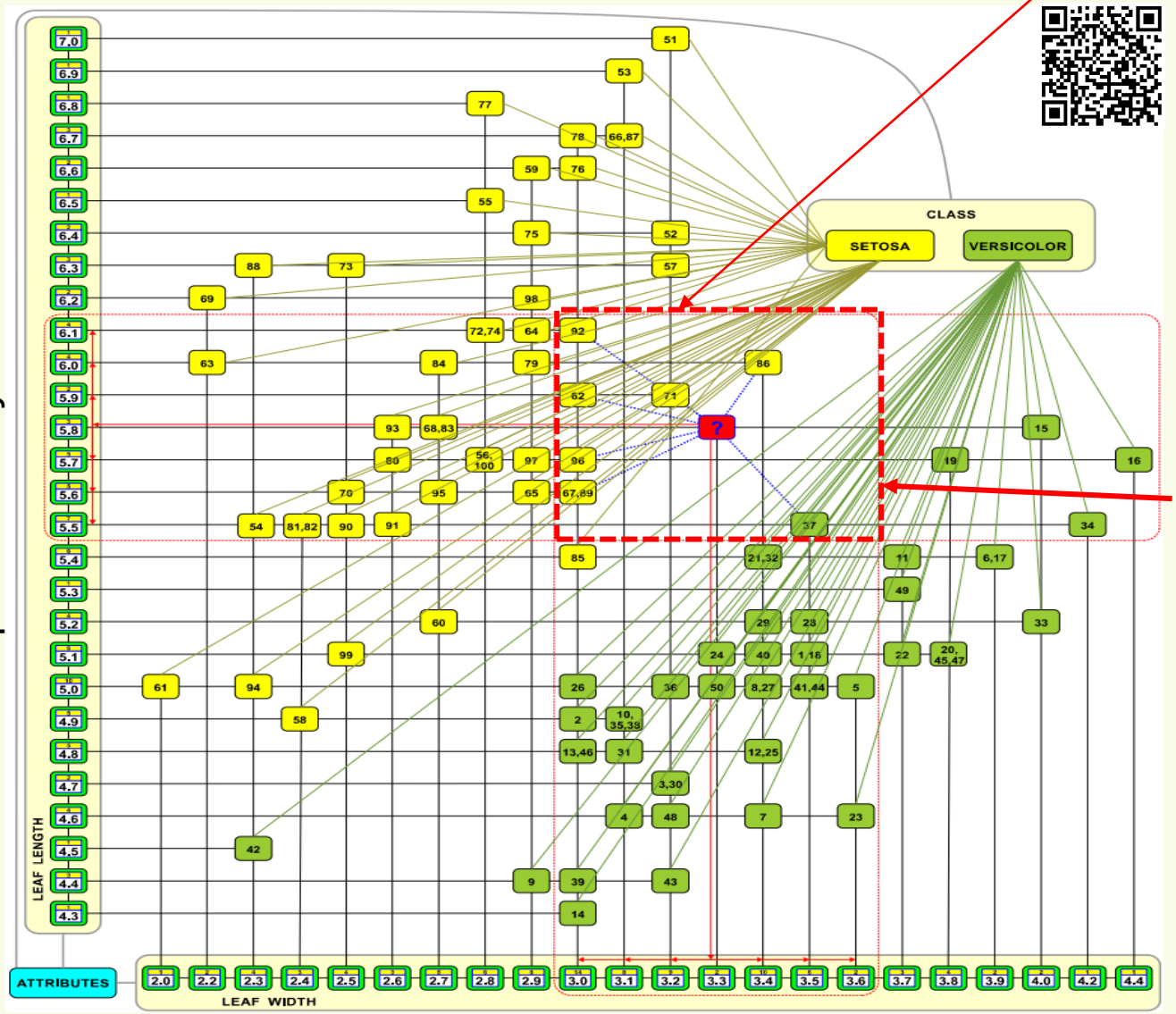
Such structures can also be used for very fast recognition, clustering, classification, searching for the most similar objects etc.

AGDS and Local Data Analyses



AGDS structures allow for the search in a limited and a small region where neighbors (the most similar) objects can be found. In can be applied to make KNN more efficient.

100 values represented by 28 value nodes!



AGDS structure created for two selected attributes and 100 training samples of Iris data.

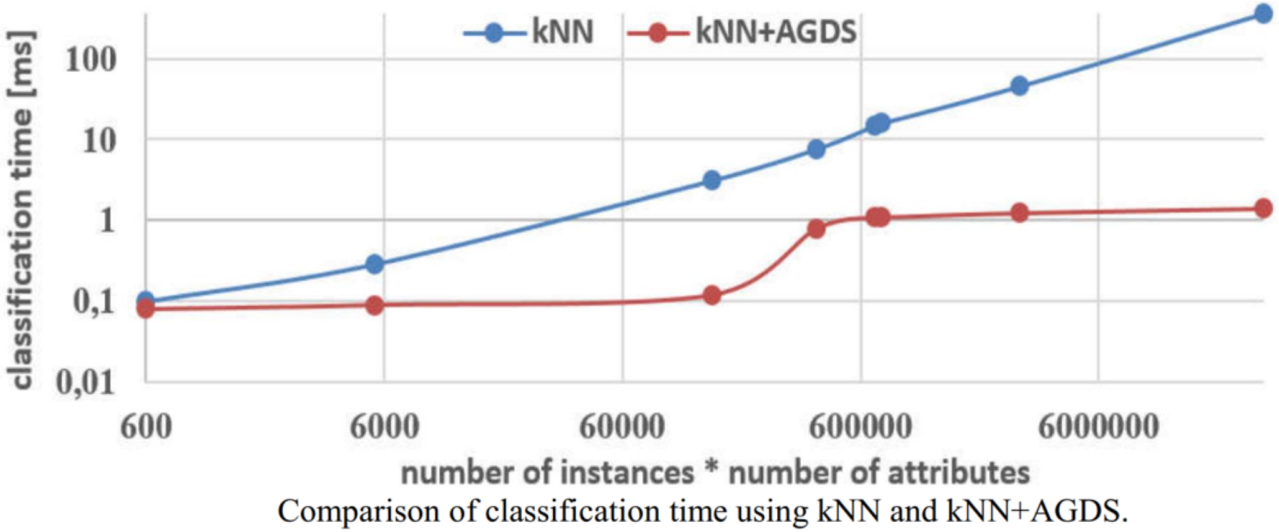
K Nearest Neighbors are searched locally in the neighborhood of the classified sample.

We can save a lot of computational time using created associations in the AGDS!

AGDS and Local Data Analyses



Classification time for the kNN+AGDS classifier is almost constant regardless of the size of the used training data sets, while classic kNN classification time grows linearly.



Comparison of classification time using kNN and kNN+AGDS.

Dataset	Number of instances	Number of attributes	kNN classification time [ms]	kNN+AGDS classification time [ms]	kNN+AGDS construction time [ms]
Iris	150	4	0.10	0.08	1
Banknote	1372	4	0.29	0.09	5
HTRU2	17898	8	3.14	0.09	134
Shuttle	43500	9	7.67	1.06	278
Credit Card	30000	23	8.69	1.07	499
Skin	245057	3	26.87	1.10	683
Drive	58509	48	46.15	1.24	2224
HEPMASS	1048576	28	362.32	1.41	31214

The size of training data and the number of attributes do not substantially influence kNN+AGDS efficiency as it is in the classic kNN classifiers. Therefore, the use of associative structures is very beneficial.

Example of Associative Inferences



Let's have a table of data about candidates for employ-ment in the company. We want to find the best candidate for the open position. We have five candidates! Who is the best one?!

	Technical Skills			PersonalSkills			LanguageSkills			Education Level	Sallary	Time Work Type	Field Importance
	Name	Years	Weight	Name	Level	Weight	Name	Level	Weight				
Job Offer	C#	≥ 3	(10/10)	Communication skills	60%-80%	(8/10)	English	≥ C1	(8/10)	Bachelor	7500	FullTime	TechnicalSkills (9/10)
	Entity Framwork	≥ 3	(9/10)										PersonalSkills (8/10)
				Ability to work under pressure	80%-100%	(10/10)							LanguageSkills (5/10)
	T-SQL	≥ 2	(8/10)				Polish	≥ Native	(7/10)				Education (10/10)
				Decision making	60%-80%	(7/10)							Sallary(10/10)
	Cloud dev	≥ 1	(6/10)										TimeWorkType (10/10)
Candidate 1	C#	2	X	Ability to work under pressure	70%	X	Polish	Native	X	Bachelor	7000	HalfTime	X
	T-SQL	3											
	Cloud dev	2		Communication skills	80%								
Candidate 2	Entity Framwork	3	X	Communication skills	70%	X	English	B2	X	Master	6500	FullTime	X
	C#	1		Decision making	90%								
	T-SQL	1		Ability to work under pressure	90%								
Candidate 3	Cloud dev	2	X	Communication skills	80%	X	Polish	Native	X	Master	6500	HalfTime	X
	C#	2		Decision making	90%		English	C1					
Candidate 4	Entity Framwork	1	X	Ability to work under pressure	70%	X	English	Native	X	Bachelor	8000	FullTime	X
	C#	1											
	T-SQL	2		Decision making	50%								
	Cloud dev	2											
Candidate 5	Cloud dev	1	X	Communication skills	70%	X	Polish	C1	X	Bachelor	7000	HalfTime	X
	Entity Framwork	2		Ability to work under pressure	80%								
	T-SQL	3		Decision making	90%								

Setup of the AGDS structure



The **charging level x** of the internally stimulated node is defined as a weighted sum (as in the 2nd ANN generation):

$$x_n = \sum_{k=1}^{s_n} x_k \cdot w_k$$

Reciprocal edges are created between value nodes $V_i^{a^k}$ and $V_j^{a^k}$ representing similar values $v_i^{a^k}$ and $v_j^{a^k}$ of the same attribute a^k and forward stimuli in both directions with the same **weight**:

$$w_{v_i^{a^k}, v_j^{a^k}} = 1 - \frac{\left| v_i^{a^k} - v_j^{a^k} \right|}{r^{a^k}}$$

where

$$r^{a^k} = v_{max}^{a^k} - v_{min}^{a^k}$$

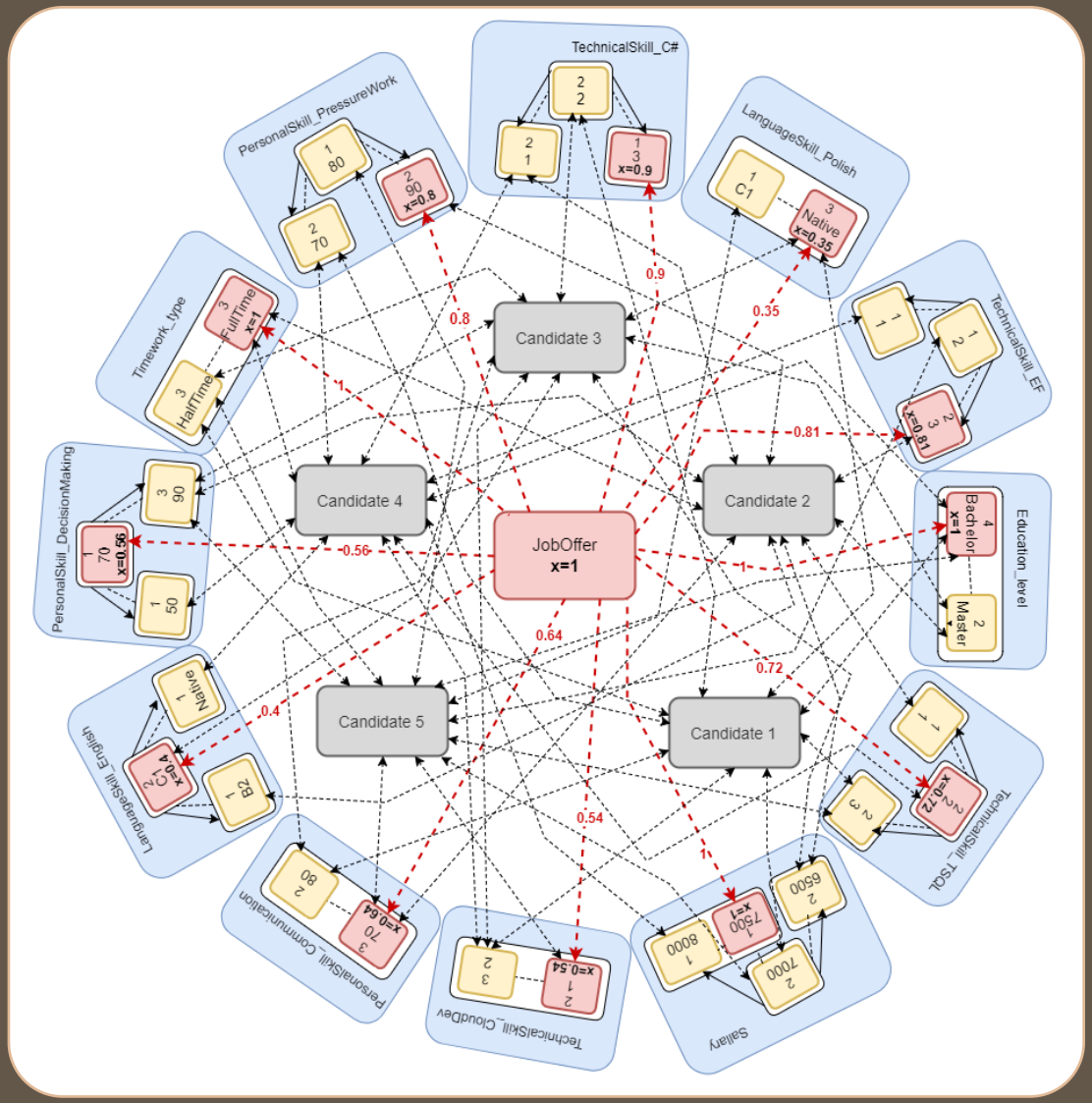
is a variation range of values of the attribute a^k.

The **weight** of the edge for the signal passing from the value node $V_i^{a^k}$ to the object node O_n can be calculated after:

$$w_{O_m, O_n} = \frac{1}{\theta_n}$$

The stimuli passing through the edge in the opposite direction

$$w_{O_n, v_i^{a^k}} = 1, w_{O_n, O_m} = 1$$

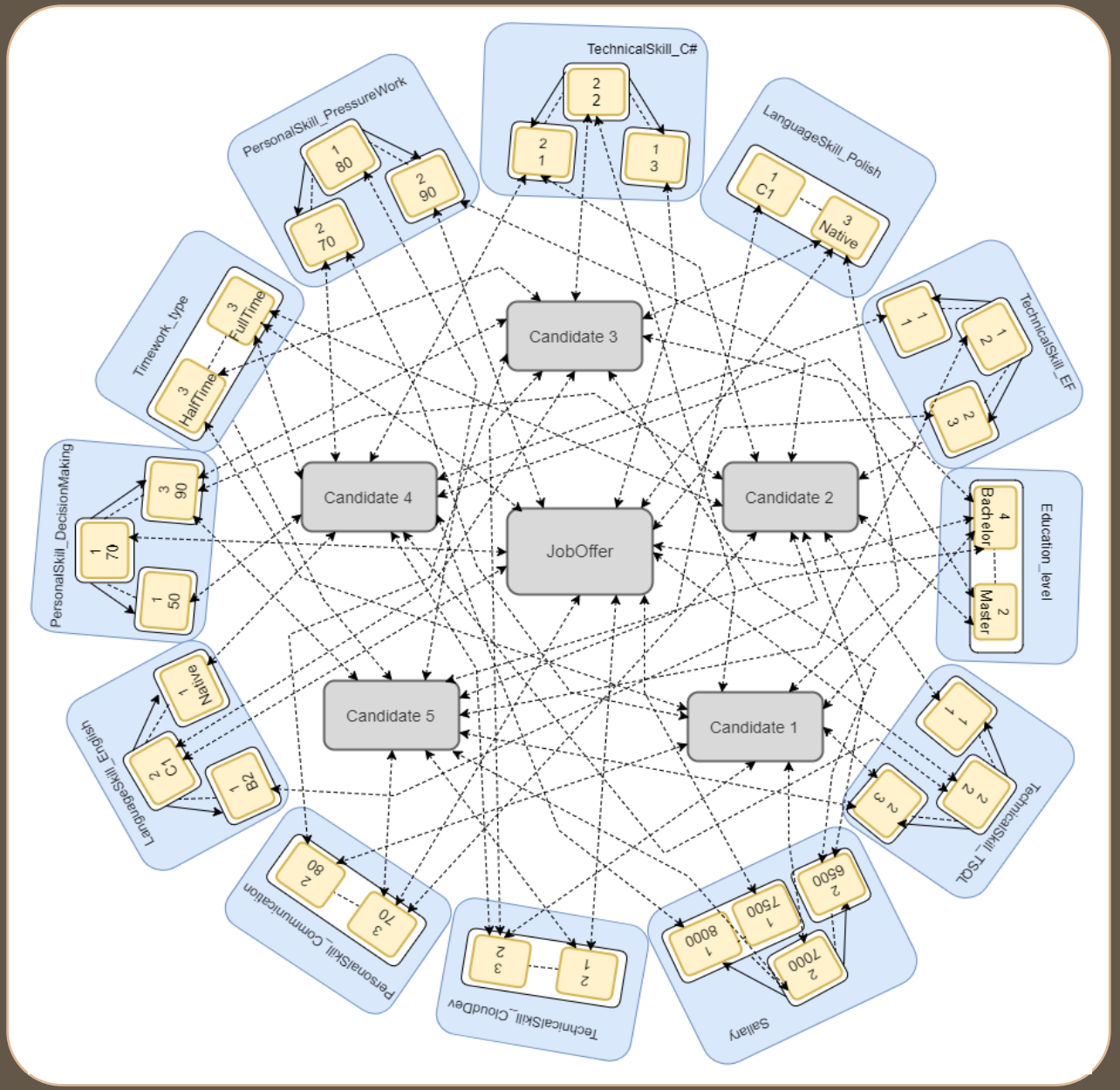


where the **threshold θ_n** is the number of values and objects that define the object nodes O_n and activate this node.

Example of Associative Inferences



First, we create AGDS structure for a given inference task.
The JobOffer node represents the input conditions, e.g. the skills of the demanded candidate.

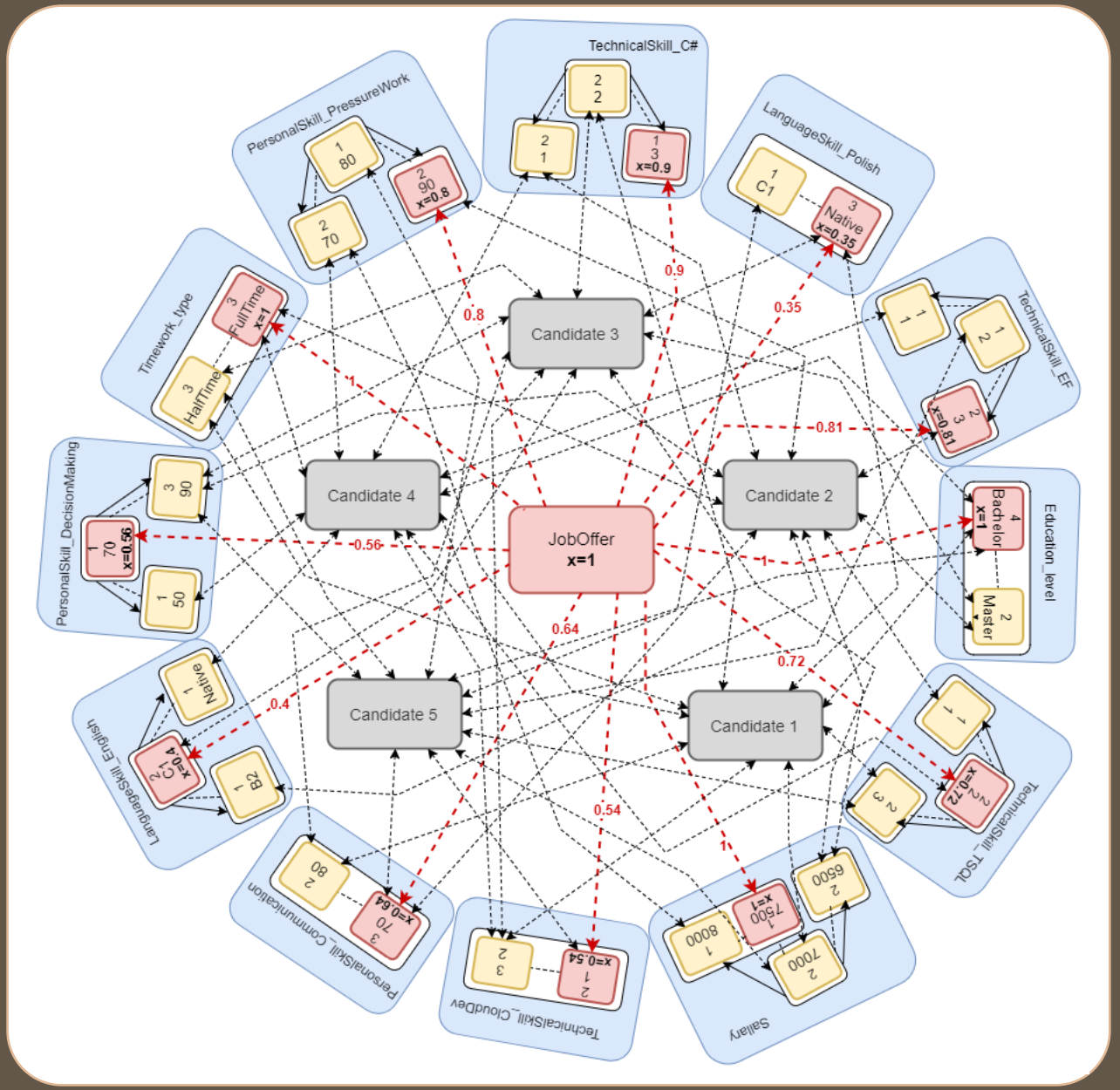


Example of Associative Inferences



We start the inference from the JobOffer node that represents the skills of the demanded candidate.

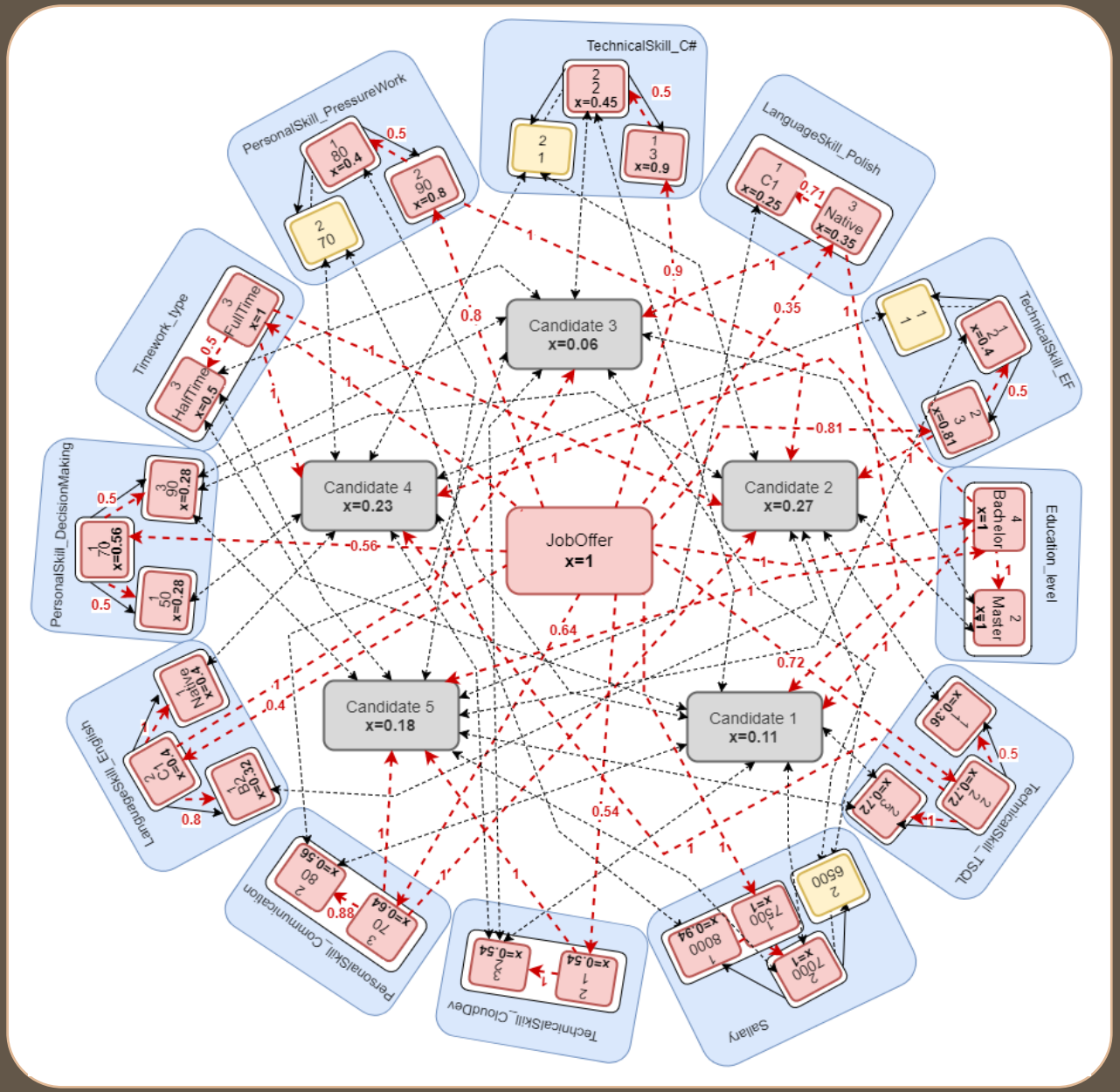
Next, we go along the connections to the values nodes and further to similar values nodes and objects nodes defined by them.



Example of Associative Inferences



We compute the associative strengths of the connected nodes multiplying the stimuli by the weights in the BFS order starting from the JobOffer node. The BFS search algorithm does not go through all AGDS nodes, but it gradually stretches the BFS tree over the mostly associated nodes starting from the node defining the input criteria and finishing in the nodes representing results.
The nodes are stimulated until they do not achieved their stimulation thresholds.
Next, the destination nodes present the answers to the question about their fitting strengths to the JobOffer node.

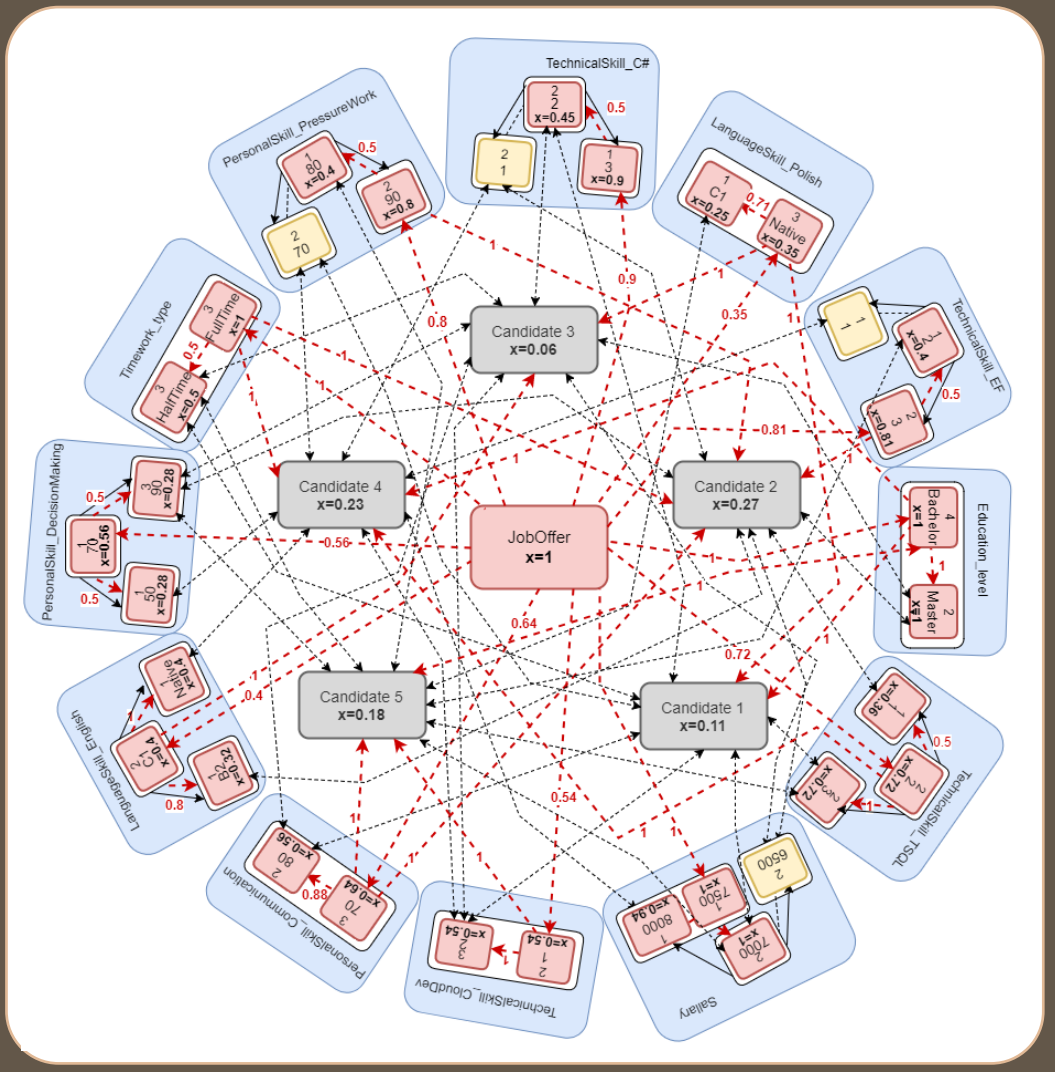


Example of Associative Inferences



Compare which structure is more suitable for inferences? Data analysis stored in tables are time-consuming, labor-intensive and resource-consuming.

	Technical Skills			PersonalSkills			LanguageSkills			Education Level	Sallary	Time Work Type	Field Importance
	Name	Years	Weight	Name	Level	Weight	Name	Level	Weight				
Job Offer	C#	≥ 3	(10/10)	Communication skills	60%-80%	(8/10)	English	≥ C1	(8/10)	Bachelor	7500	FullTime	TechnicalSkills (9/10)
	Entity Framwork	≥ 3	(9/10)										PersonalSkills (8/10)
				Ability to work under pressure	80%-100%	(10/10)							LanguageSkills (5/10)
	T-SQL	≥ 2	(8/10)				Polish	≥ Native	(7/10)				Education (10/10)
				Decision making	60%-80%	(7/10)							Sallary(10/10)
	Cloud dev	≥ 1	(6/10)										TimeWorkType (10/10)
Candidate 1	C#	2	X	Ability to work under pressure	70%	X	Polish	Native	X	Bachelor	7000	HalfTime	X
	T-SQL	3											
	Cloud dev	2		Communication skills	80%								
Candidate 2	Entity Framwork	3	X	Communication skills	70%	X	English	B2	X	Master	6500	FullTime	X
	C#	1		Decision making	90%								
	T-SQL	1		Ability to work under pressure	90%								
Candidate 3	Cloud dev	2	X	Communication skills	80%	X	Polish	Native	X	Master	6500	HalfTime	X
	C#	2		Decision making	90%		English	C1					
Candidate 4	Entity Framwork	1	X	Ability to work under pressure	70%	X	English	Native	X	Bachelor	8000	FullTime	X
	C#	1											
	T-SQL	2		Decision making	50%								
	Cloud dev	2											
Candidate 5	Cloud dev	1	X	Communication skills	70%	X	Polish	C1	X	Bachelor	7000	HalfTime	X
	Entity Framwork	2		Ability to work under pressure	80%								
	T-SQL	3		Decision making	90%								

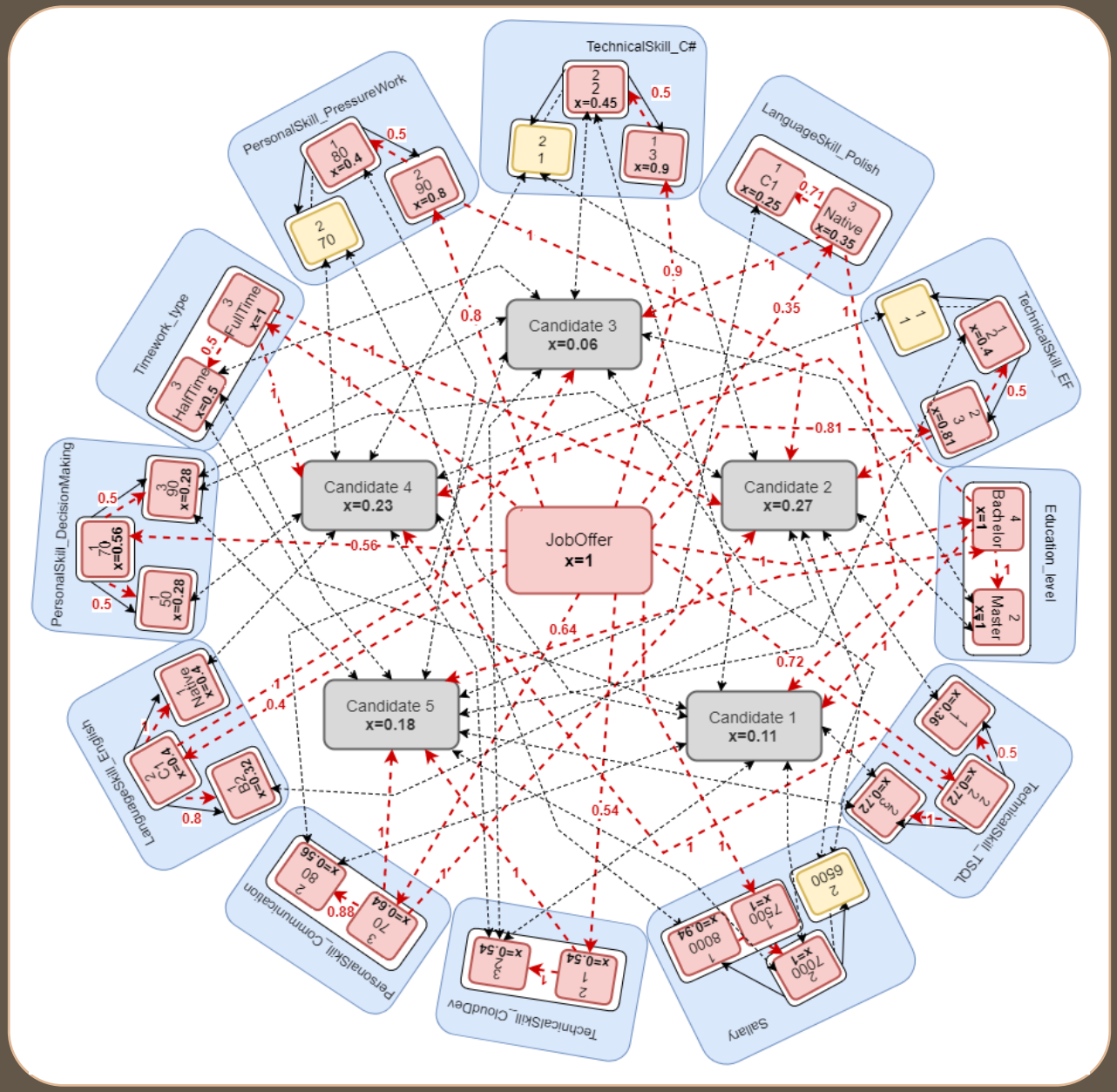


Example of Associative Inferences



AGDS graph stimulates nodes according to the associations strengths.

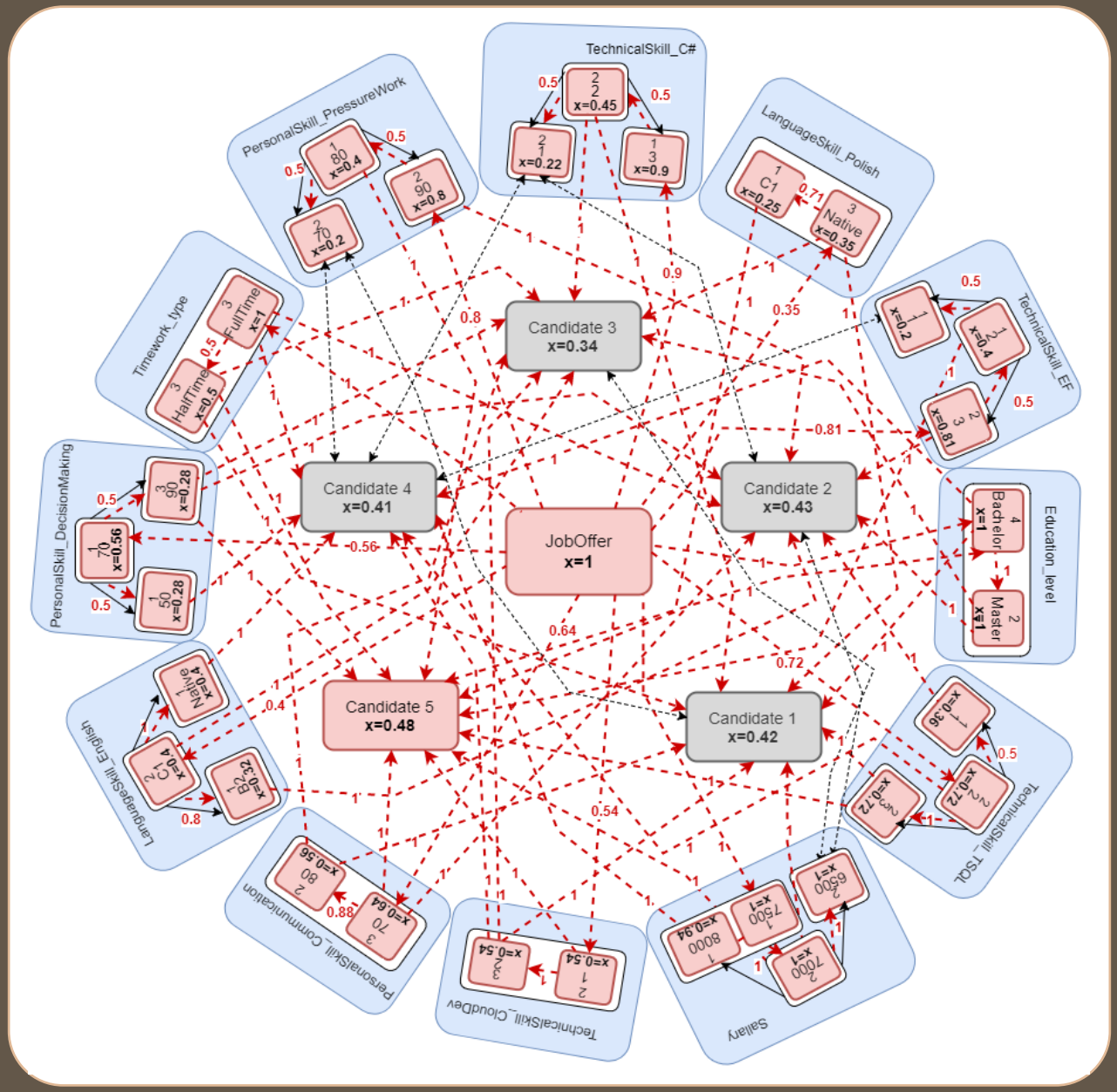
Object node	Stimulus strength
Candidate1	$\frac{1}{12} \cdot (1 + 0.35) \approx 0.11$
Candidate2	$\frac{1}{12} \cdot (1 + 0.8 + 0.81 + 0.64) \approx 0.27$
Candidate3	$\frac{1}{12} \cdot (0.4 + 0.35) \approx 0.06$
Candidate4	$\frac{1}{12} \cdot (1 + 1 + 0.72) \approx 0.23$
Candidate5	$\frac{1}{12} \cdot (0.64 + 0.54 + 1) \approx 0.18$



Example of Associative Inferences



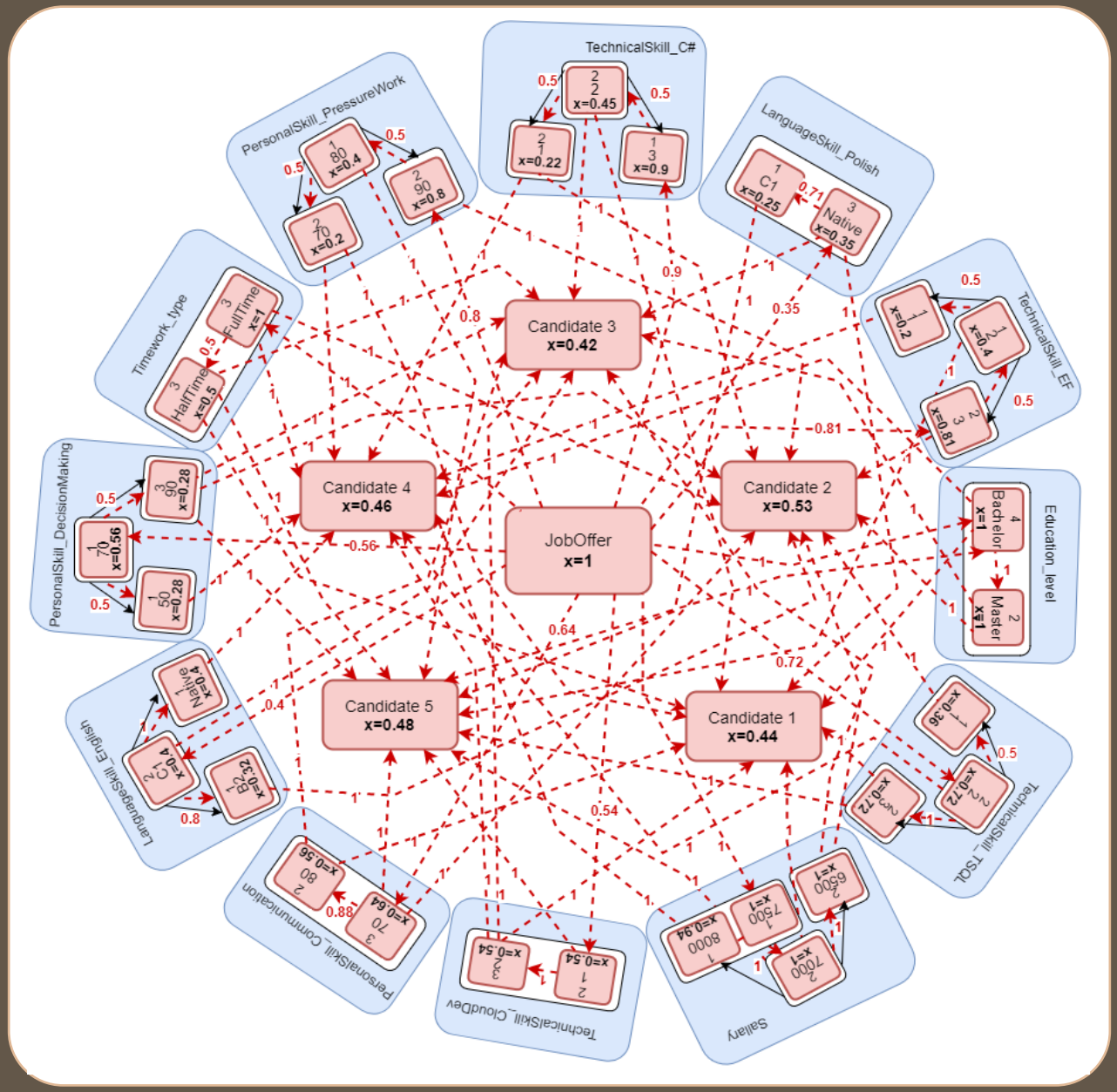
We follow with stimulation of the next open nodes in the BFS order (we gradually span BFS activation tree on the AGDS graph).



Example of Associative Inferences

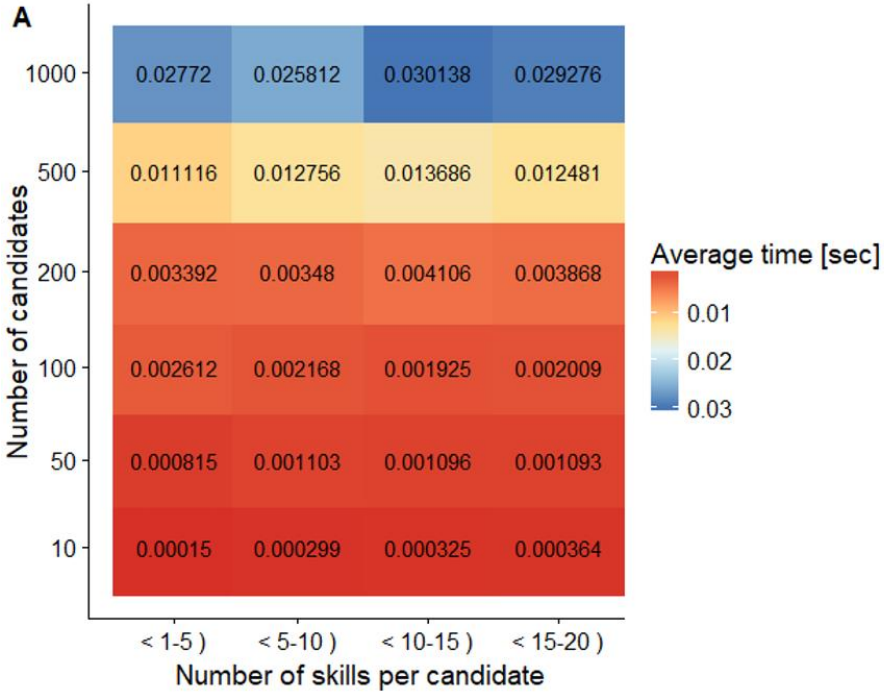


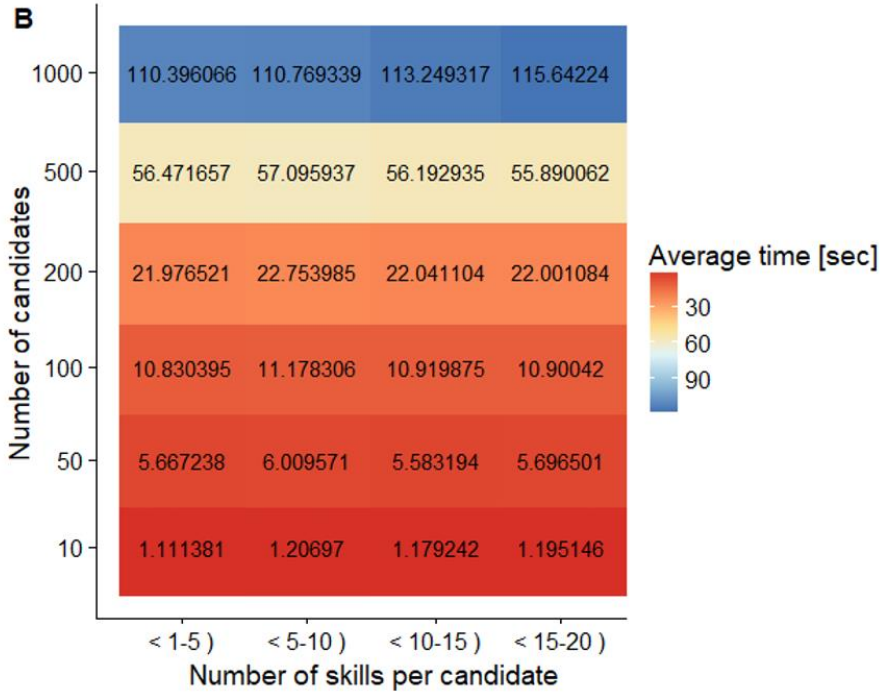
We finish when the appropriate number of the destination (output) nodes are activated and closed. Next, we can find out the final inference on the basis of reading the stimulation strength of those destination nodes which determine the associative strength to the input conditions.



Efficiency of Associative Inferences







The heatmaps present the comparison of the time efficiencies of searching for ideal candidate using:

A) AGDS structures and

B) classic tabular structures

for different number of candidates and considered skills.

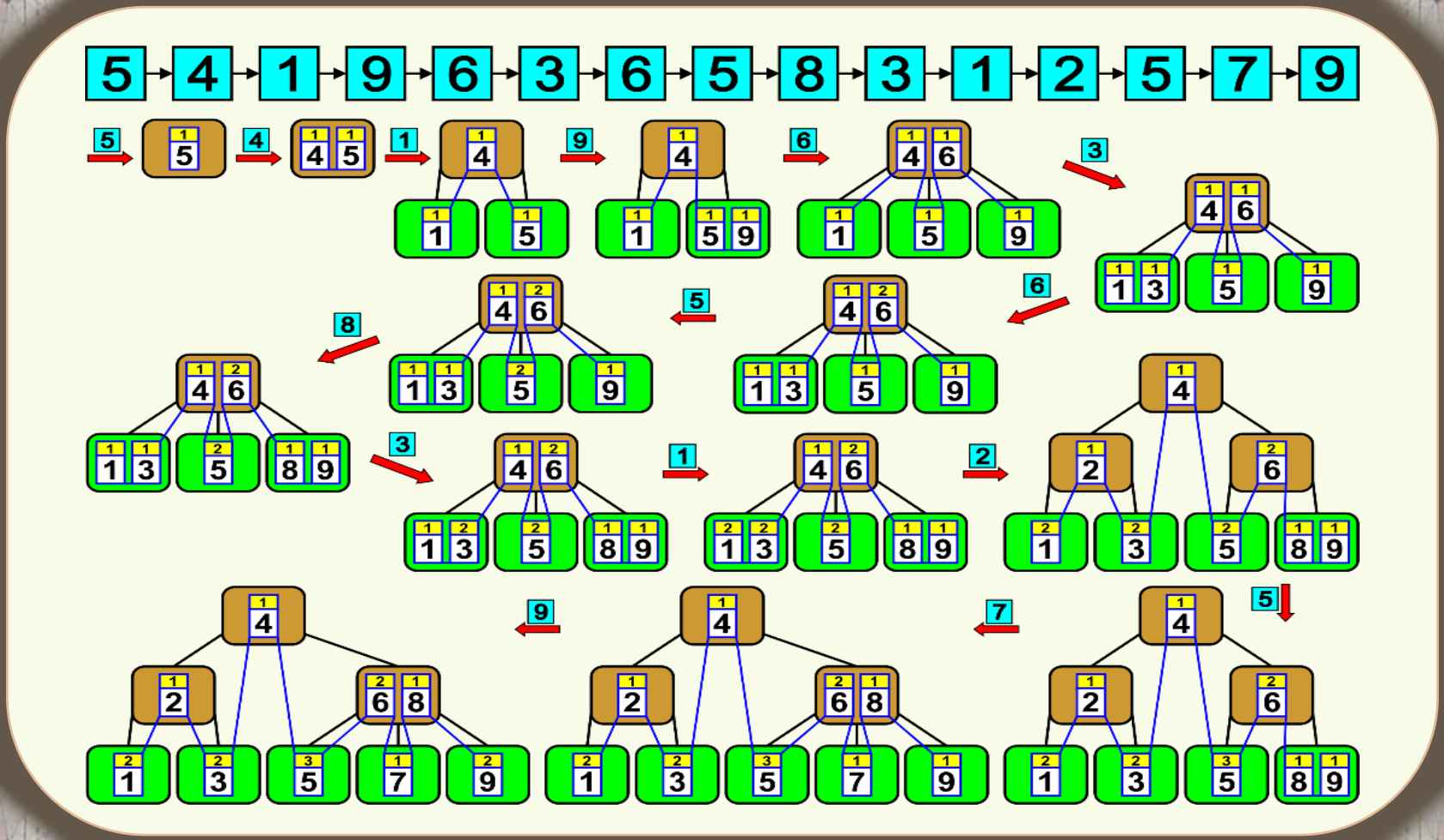
AVB+trees Operations



- ✓ AVB+trees are used to optimize data management, data access, data sorting due to speed and memory size.
- ✓ They support aggregative and associative mechanisms of various associative and cognitive structures and implement and supports the quick calculation of common operations like:
  - Insert (new value),
  - Remove (stored values),
  - Search for any values (it can exist or not),
  - GetMin, GetMax to find extreme values of the data collection,
  - Sum (of selected or all values in a given range or all of them),
  - Count (of selected or all values in a given range or all of them),
  - Average (of selected or all values in a given range or all of them),
  - Median,
  - etc.

AVB+tree structure is suitable
to work with AGDS structures!

Insert Operation on AVB+Trees



AVB+trees self-balance, self-sort and self-organize the structure during the insert operation!

Insert Operation



The Insert operation on the AVB+tree is processed as follows:

1. Start from the root and go recursively down along the branches to the descendants until the leaf is not achieved after the following rules:
   - if one of the elements stored in the node already represents the inserted key, increment the counter of this element, and finish this operation;
   - else go to the left child node if the inserted key is less than the key represented by the leftmost element in this node;
   - else go to the right child node if the inserted key is greater than the key represented by the rightmost element in this node;
   - else go to the middle child node.
2. When the leaf is achieved:
   - and if the inserted key is already represented by one of the elements in this leaf, increment the counter of this element, and finish this operation;
   - else create a new element to represent the inserted key and initialize its counter to one, next insert this new element to the other elements stored in this leaf in the increasing order, update the neighbor connections, and go to step 3.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Insert Operation



3. If the number of all elements stored in this leaf is greater than two, divide this leaf into two leaves in the following way:

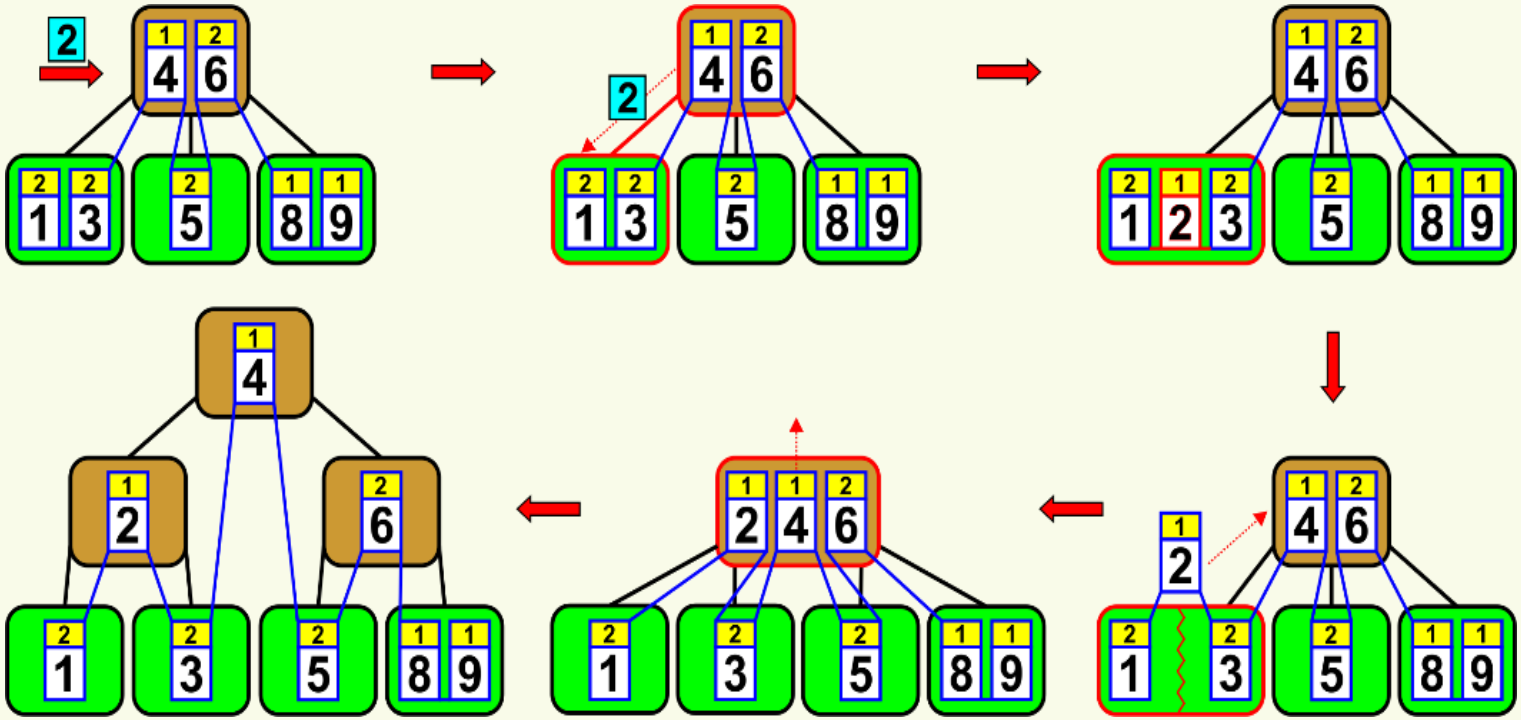
- let the divided leaf represent the leftmost element representing the least key in this node together with its counter;
- create a new leaf and let it represent the rightmost element representing the greatest key in this node together with its counter;
- and the middle element (representing the middle key together with its counter) and the pointer to the new leaf representing the rightmost element pass to the parent node if it exists, and go to step 4;
- if the parent node does not exist, create it (a new root of the AVB+tree) and let it represent this middle element (representing the middle key together with its counter), and create new branches to the divided leaf representing the leftmost element and to the leaf pointed by the passed pointer to the new leaf representing the rightmost element.
  Next, finish this operation.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Rebalancing during Insert Operation



A self-balancing mechanism of an AVB+tree during the Insert operation when adding the value (key) „2” to the current structure which must be reconstructed because the node is overfilled and must be divided.



Self-balancing and self-sorting mechanism of the Insert Operation when a node is overfilled and must be divided!

Insert Operation



4. Insert the passed element between the element(s) stored in this node in the key - increasing order after the following rules:

- if the element has come from the left branch, insert it on the left side of the existing element(s) in this node;
- if the element has come from the right branch, insert it on the right side of the existing element(s) in this node;
- if the element has come from the middle branch, insert it between the existing element(s) in this node.

5. Create a new branch to the new node (or leaf) pointed by the passed pointer and insert this pointer to the child list of pointers immediately after the pointer representing the branch to the divided node (or leaf).

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Insert Operation



6. If the number of all elements stored in this node is greater than two, divide this node into two nodes in the following way:

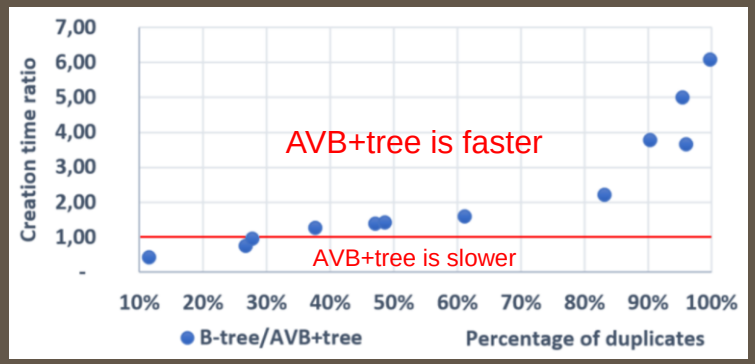
- let the existing node represent the leftmost element representing the least key in this node together with its counter;
- create a new node and let it represent the rightmost element representing the greatest key in this node together with its counter;
- the middle element (representing the middle key together with its counter) and the pointer to the new node representing the rightmost element pass to the parent node if it exists; and go back to step 4;
- if the parent node does not exist, create it (a new root of the AVB+tree) and let it represent this middle element (representing the middle key together with its counter), and create new branches to the divided node representing the leftmost element and to the node pointed by the passed pointer to the new node representing the rightmost element. Next, finish this operation.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

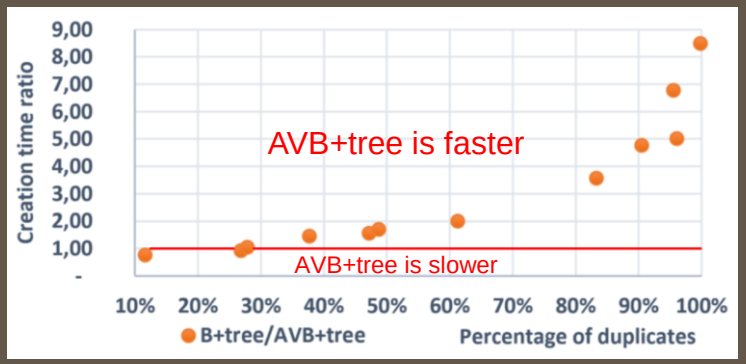
Efficiency of Insert Operation



The efficiency comparisons of Insert Operations of B-tree and AVB+tree:



The efficiency comparisons of Insert Operations of B+tree and AVB+tree:



Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



- ✓ The Remove operation allows to remove a key from the AVB+tree structure and next quickly rebalance and reorganize the structure automatically if necessary.
- ✓ If the removed key is duplicated in the current structure, then only the counter of the element which represents it is decremented.
- ✓ When the removed key is represented by the element which counter is equal one then the element is removed from the node.
- ✓ If this node is a leaf containing only a single element, then the leaf is removed as well, and a rebalancing operation of the AVB+tree is executed.

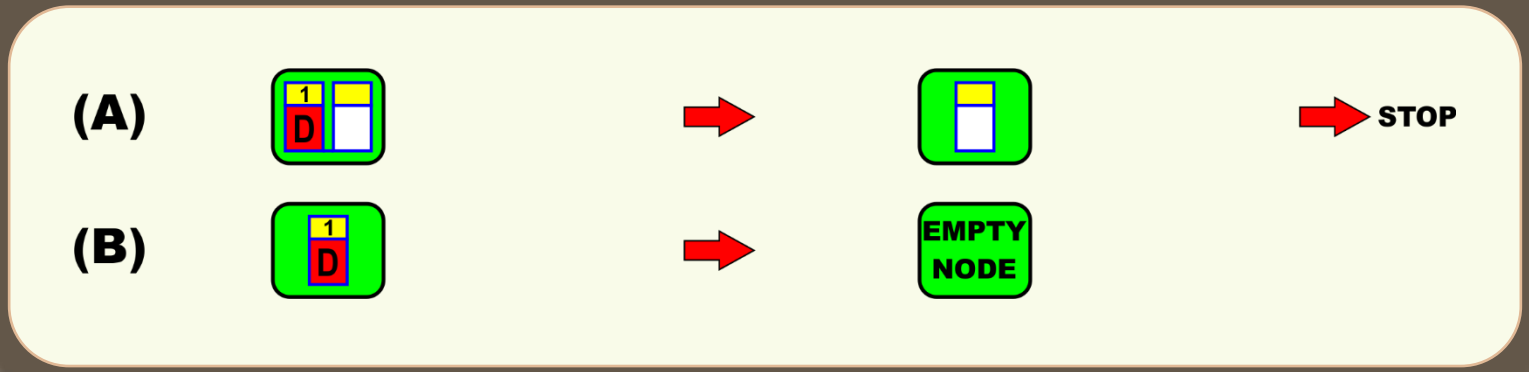
Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



The Remove operation on the AVB+tree is processed as follows:

1. Use the search procedure to find an element containing the key intended for removal. If this key is not found in the tree, finish the delete operation with no effect;
2. Else if the counter of the element storing the removed key is greater than one, decrement this counter, and finish the delete operation.
3. Else if the element storing the removed key is a leaf, then remove the element storing this key from this leaf, switch pointers from its predecessor and successor to point themselves as direct neighbors. Next, if this leaf is not empty, finish the delete operation (Fig. A), else go to step 7 (Fig. B).

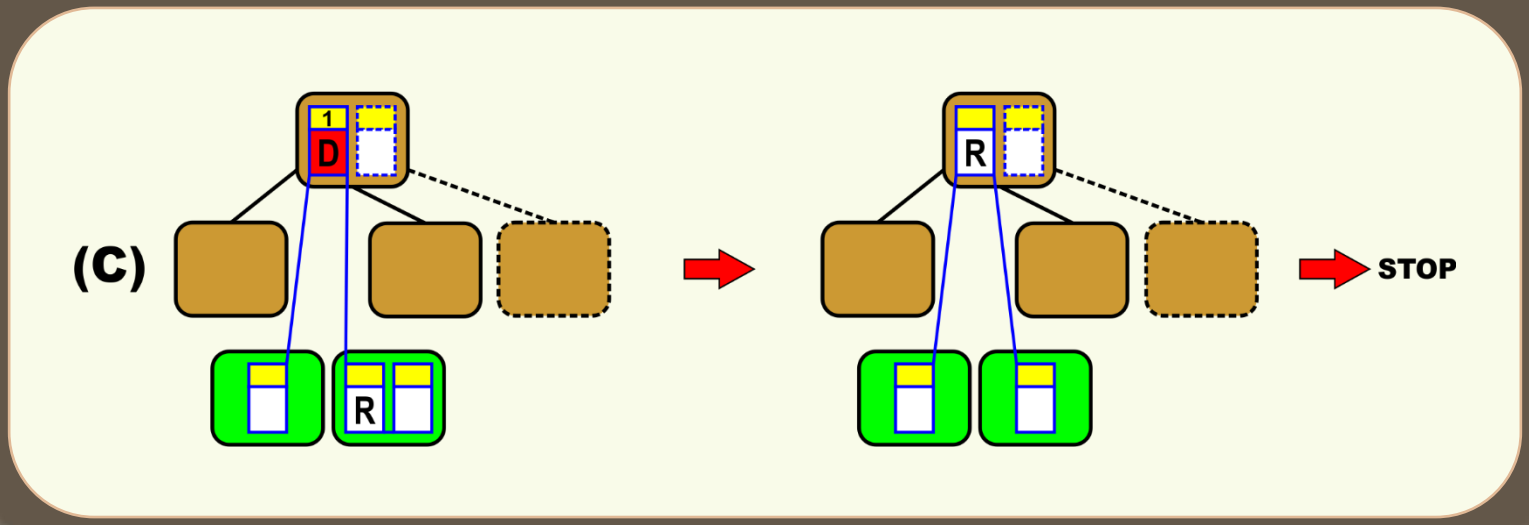


Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



4. Else the element storing the removed key is a non-leaf node that must be replaced by one of the neighbor connected elements stored in one of two leaves. If only one leaf from the leaves containing neighbor elements to the removed element contains two elements, then replace the removed element in the non-leaf node by this connected neighbor element from the leaf containing two elements, and finish the delete operation (Fig. 14C), else go to step 5.

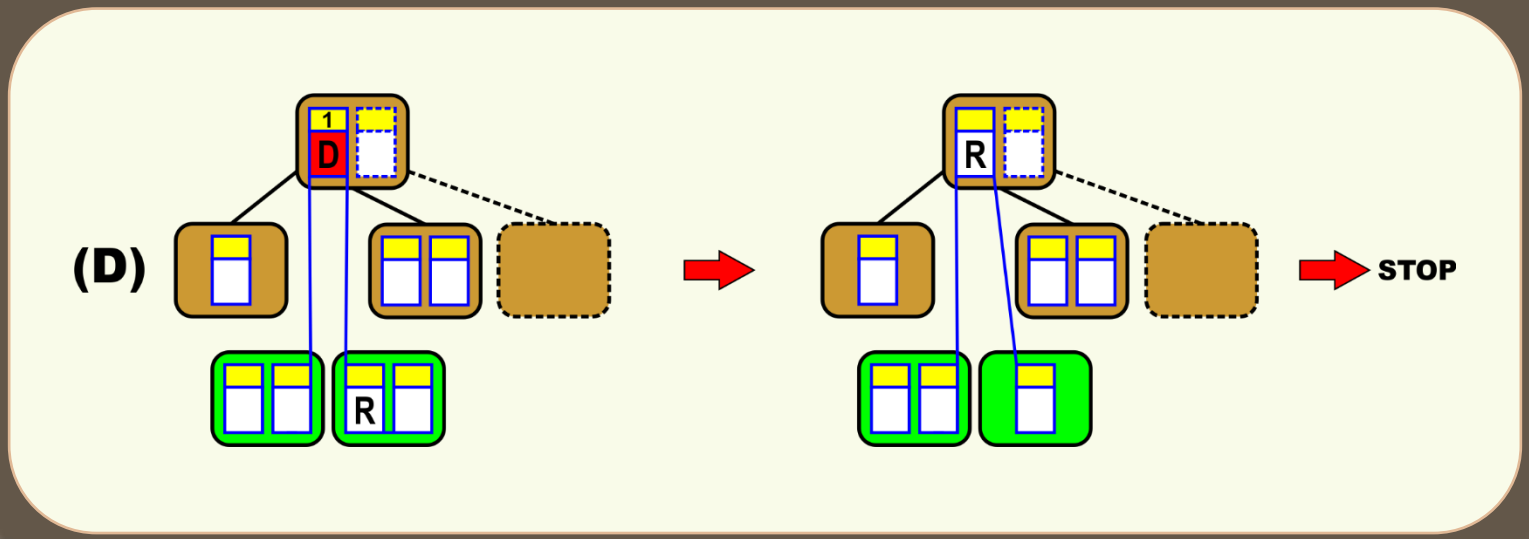


Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



5. Here, both leaves containing a neighbor element to the removed one contain two elements or one element both. In this case, check which one of the neighbor child nodes contains more elements. Next, replace the removed element by the neighbor element stored in the leaf of the subtree which root contains more elements, and finish the delete operation (Fig. D) in case when no leaf left without any element or go to step 8 when there is left an empty node; else go to step 6.



Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



6. Here, both neighbor child nodes contain the same number of elements. In this case, check whether the key stored in the rightmost element from the left neighbor child or the key stored in the leftmost element from the right neighbor child is more distant from the key stored in the removed element. The distance can be calculated differently dependently on compared data types. We can use different metrics for the string and numerical data types:

$$DISTANCE_{STR} = \sum_{i=1}^{max\{lengh(KEY_1),lengh(KEY_2)\}} \frac{1}{|KEY_1[i] - KEY_2[i]| \cdot \|X\|^{i-1}}$$

$$|KEY_1[i] - KEY_2[i]| = \begin{cases} distance(KEY_1[i], KEY_2[i]) \; in \; X \; if \; KEY_1[i] \in X \wedge KEY_2[i] \in X \\ \|X\| \qquad\qquad if \; KEY_1[i] \notin X \vee KEY_2[i] \notin X \end{cases}$$

$$DISTANCE_{NUM} = |KEY_1 - KEY_2|$$

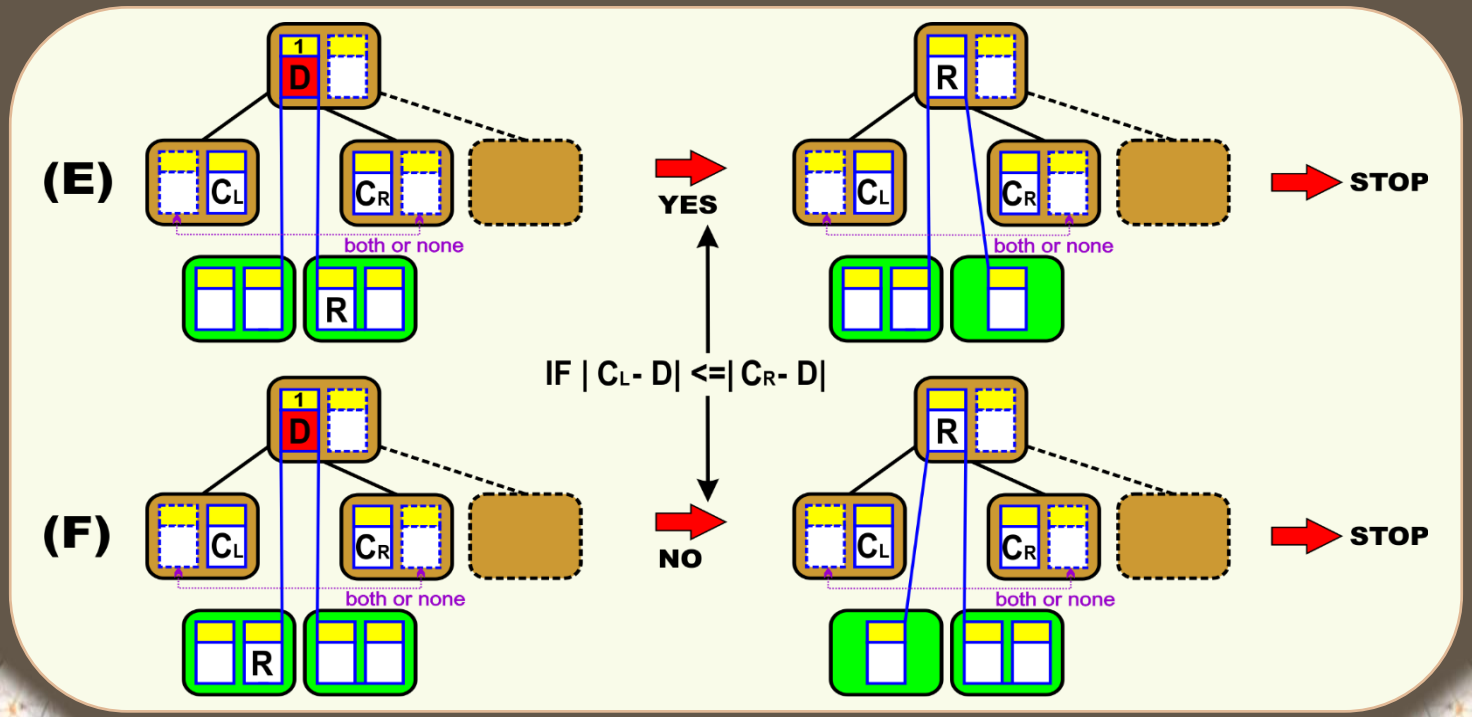
where $KEY_1[i] \in X$ means the i-th sign of the KEY_1-th string and $|KEY_1[i] - KEY_2[i]|$ is equal to the number of signs (e.g. letters) between $KEY_1[i]$ and $KEY_2[i]$ in a given sign set X (e.g. ASCII), and $\|X\|$ determines the number of signs in the set X.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



Next, replace the removed element by the right neighbor element when the distance of the key stored in this right neighbor child is greater or equal to the distance of the key stored in this left neighbor child (Fig. E or G), else replace it by the left neighbor element (Fig. F or H).
If the leaves containing the neighbor elements contain two elements both, then finish the delete operation (Fig. E and F), else (Fig. G and H) go to step 7.

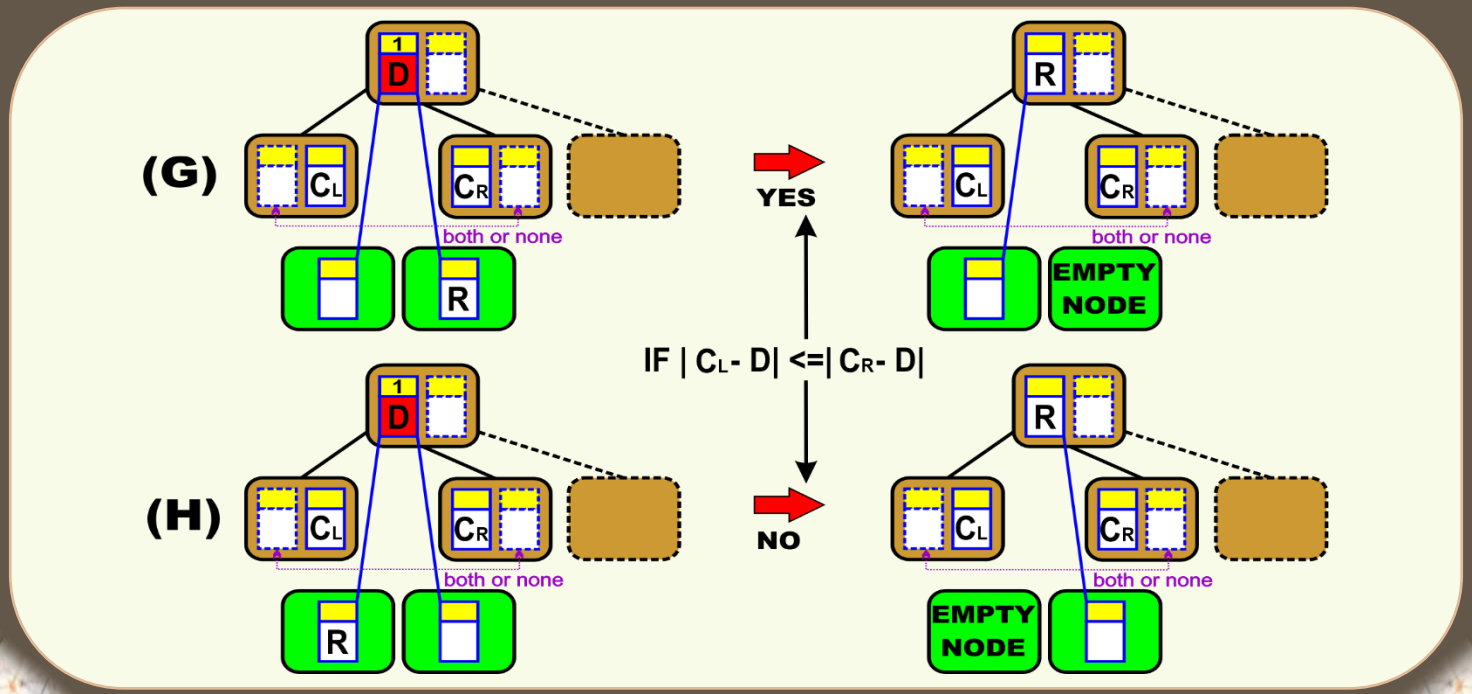


Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



7. After the removal of the element from the leaf or after the replacement of the removed element from the non-leaf node by the leaf element, there is left an empty leaf (Fig. B, G, or H) that must be filled by at least one element or removed from the tree. Next, the tree must be rebalanced to meet the AVB+tree requirements. First, try to take an element from the nearest sibling. In these cases, remove the empty leaf and go to its parent, and go to step 8.

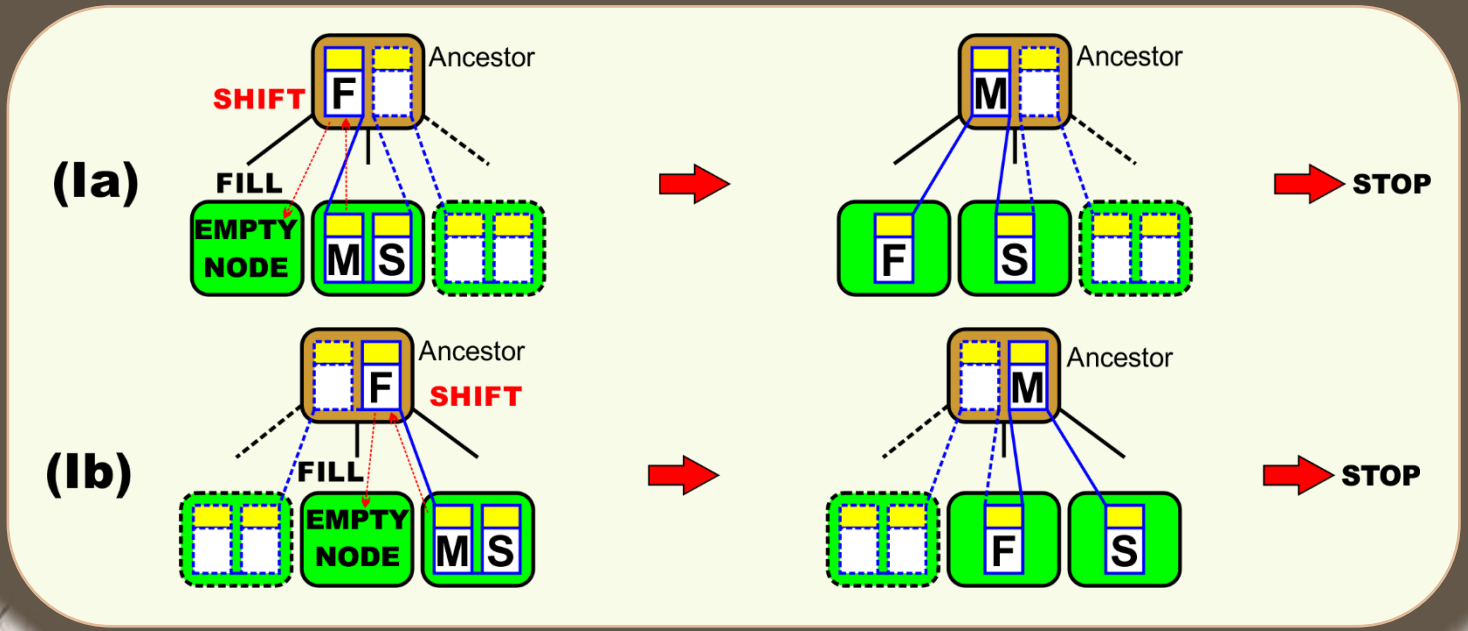


Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



8. If the nearest sibling (or cousin) of the empty leaf contains more than a single element, then move the closest key (2 in Fig. I) to the removed one from the empty node to the parent (or ancestor), and move the neighbor element (1 in Fig. I) (to the removed one from the empty node) from the parent (or ancestor) node to the empty leaf (Fig. I). Use the siblings before the cousins. Next, finish the delete operation.

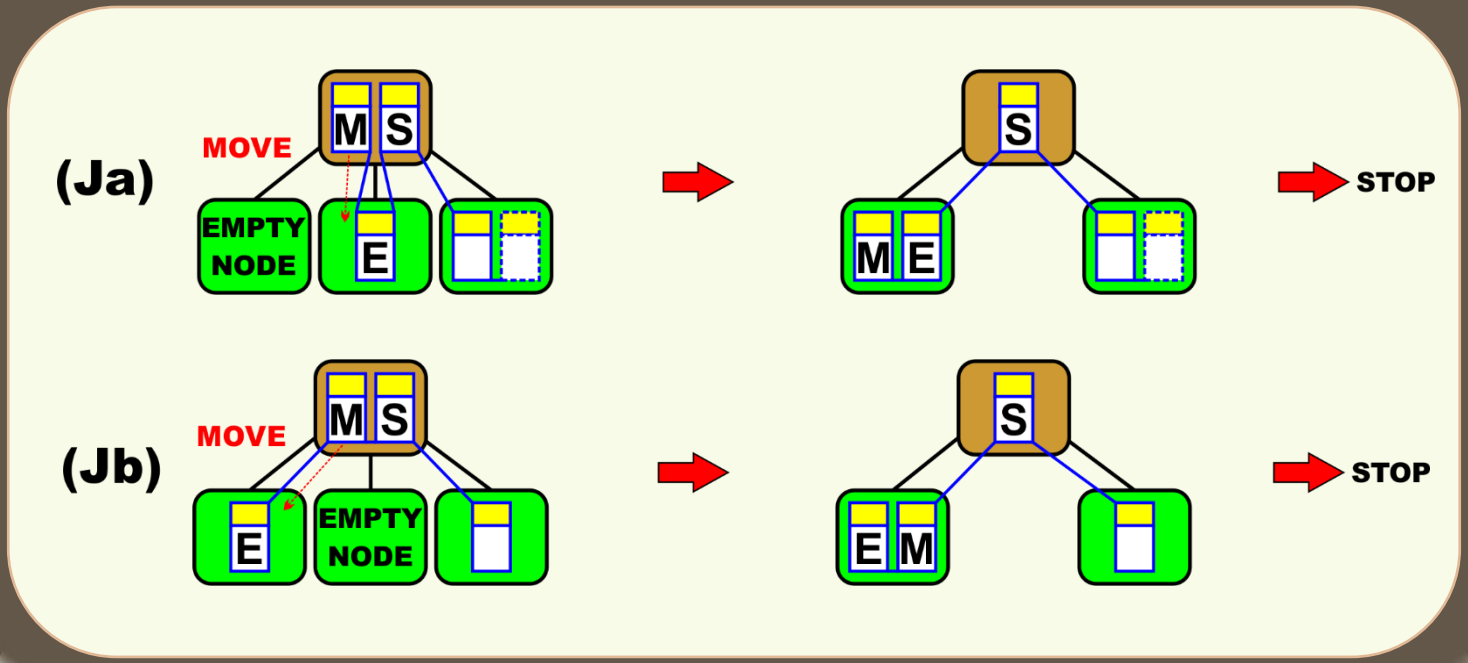


Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



9. Else if the nearest sibling of the empty leaf contains only a single element (2 in Fig. J), but its parent contains two elements, then move the closest parent element (1 in Fig. J) to the element removed from the empty node to this sibling in the right order, remove the empty node (Fig. J), and finish the delete operation.

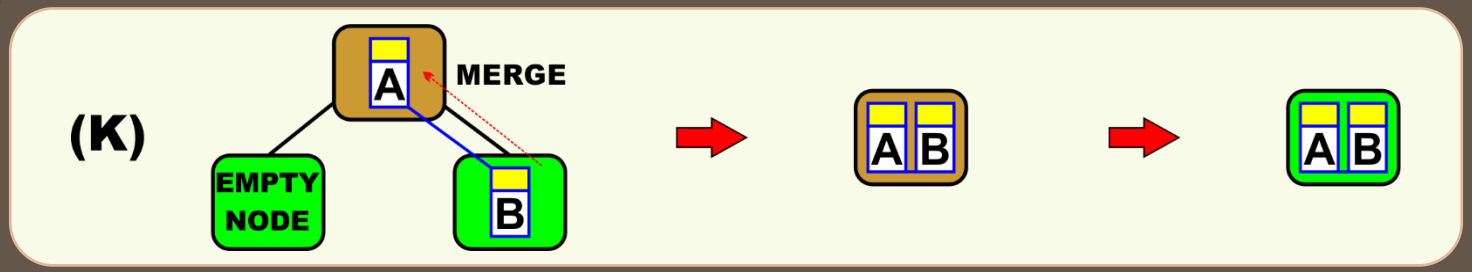


Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



10. Else both the parent and the sibling contain only a single element.
In this case, merge them in the parent node, moving the element from this sibling to its parent, and this parent node becomes to be a leaf which is placed one level higher than the other leaves (Fig. K). Hence, the tree must be rebalanced to meet the AVB+tree requirements in the subsequent routines described in the following steps.



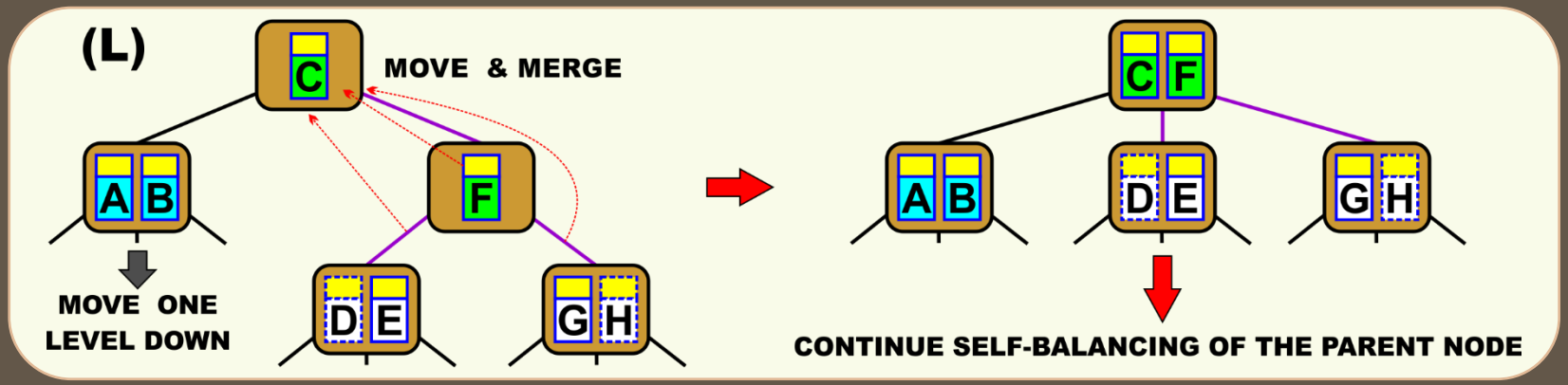
11. In this and following steps, there is always one reduced subtree which is one level up, i.e. all its leaves are one level higher than the other leaves of the tree. The smallest subtree can consist of the leaf containing two elements. The rebalancing operation is started from the root of the reduced subtree in step 12.

Less than logarithmic expected computational complexity
(typically constant) for data containing duplicates!

Remove Operation



12. If the parent node of the root of the reduced subtree contains two elements go to step 16, else go to step 13.
13. If the second child of this parent contains a single element go to step 14 (Fig. L), else go to step 15 (Fig. M).



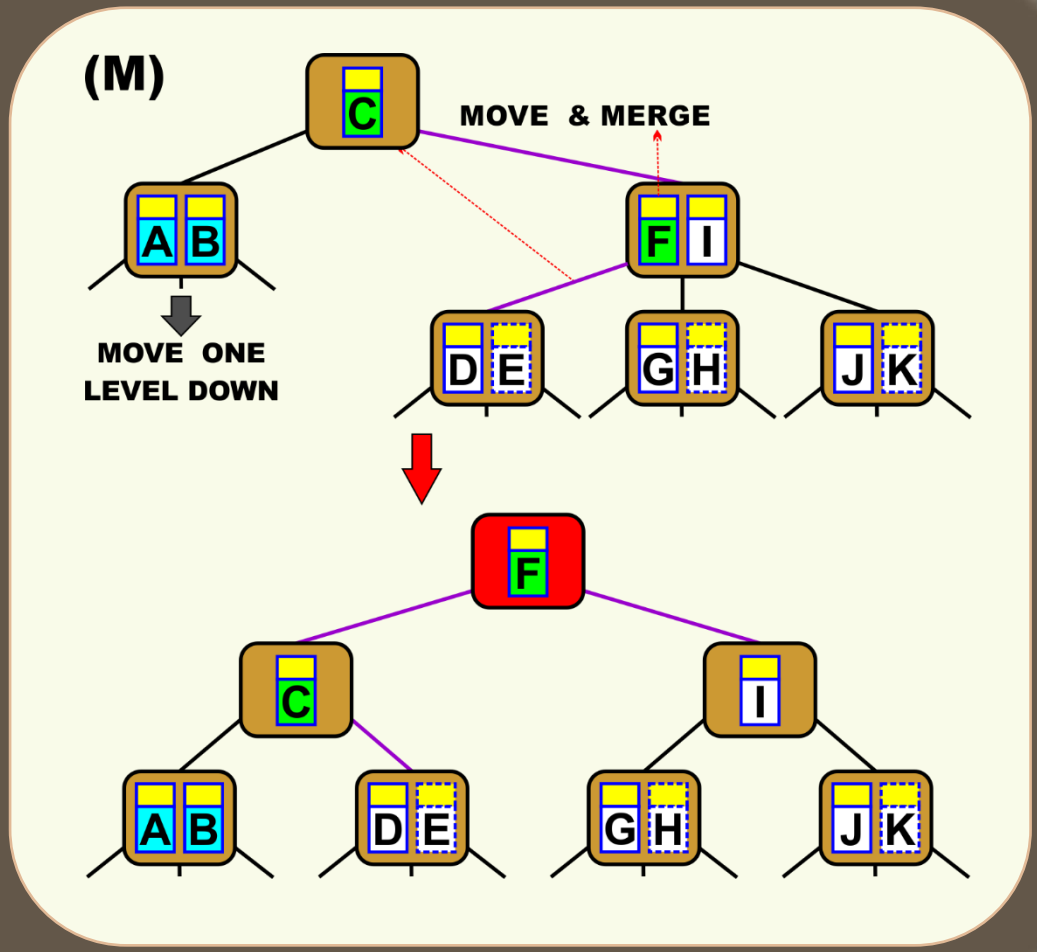
14. Merge this second child (containing a single element) with that parent as shown in Fig. L, and because the parent subtree of the reduced subtree has also lowered its height and must be rebalanced, go back to step 11 and rebalance the resultant subtree achieved after this transformation until the root of this subtree is not the root of the whole tree. If the main root is reached, it means that the tree is rebalanced, and its height was lowered by one, therefore finish the deletion operation; else go to step 15.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



15. Merge this second child (containing two elements) with that parent as shown in Fig. M, and because the merged parent node is overfilled, divide it and create a new root of this subtree (Fig. M). Next, finish the delete operation.



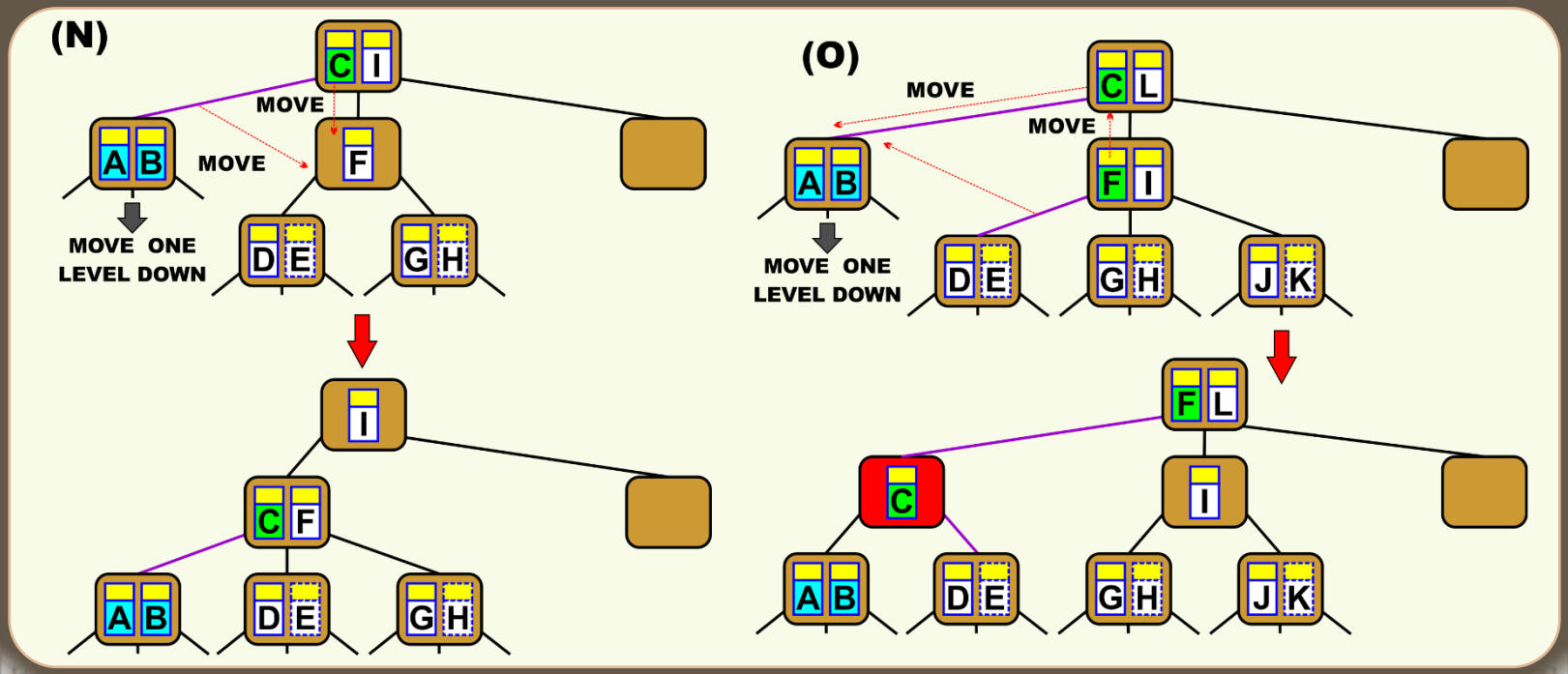
Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation



16. In this case, the parent node of the root of the reduced subtree contains two elements. If this reduced subtree root is a left or right child of its parent, then go to step 17 (Figs. N and O), else (it is a middle child) go to step 20 (Figs. P and Q).

17. If one of the nearest neighbor siblings of this reduced subtree root contains a single element, go to step 18 (Fig. N), else go to step 19 (Fig. O).

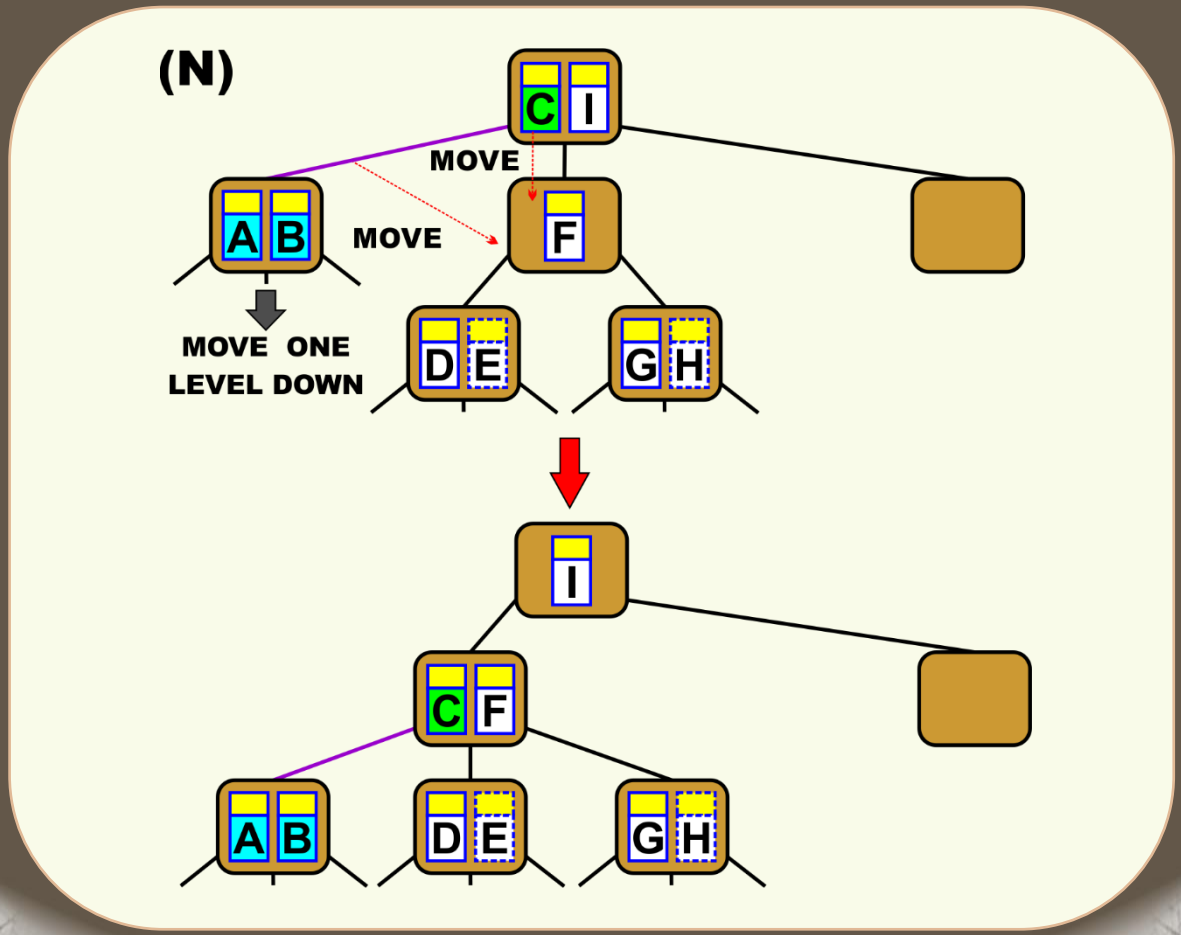


Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation

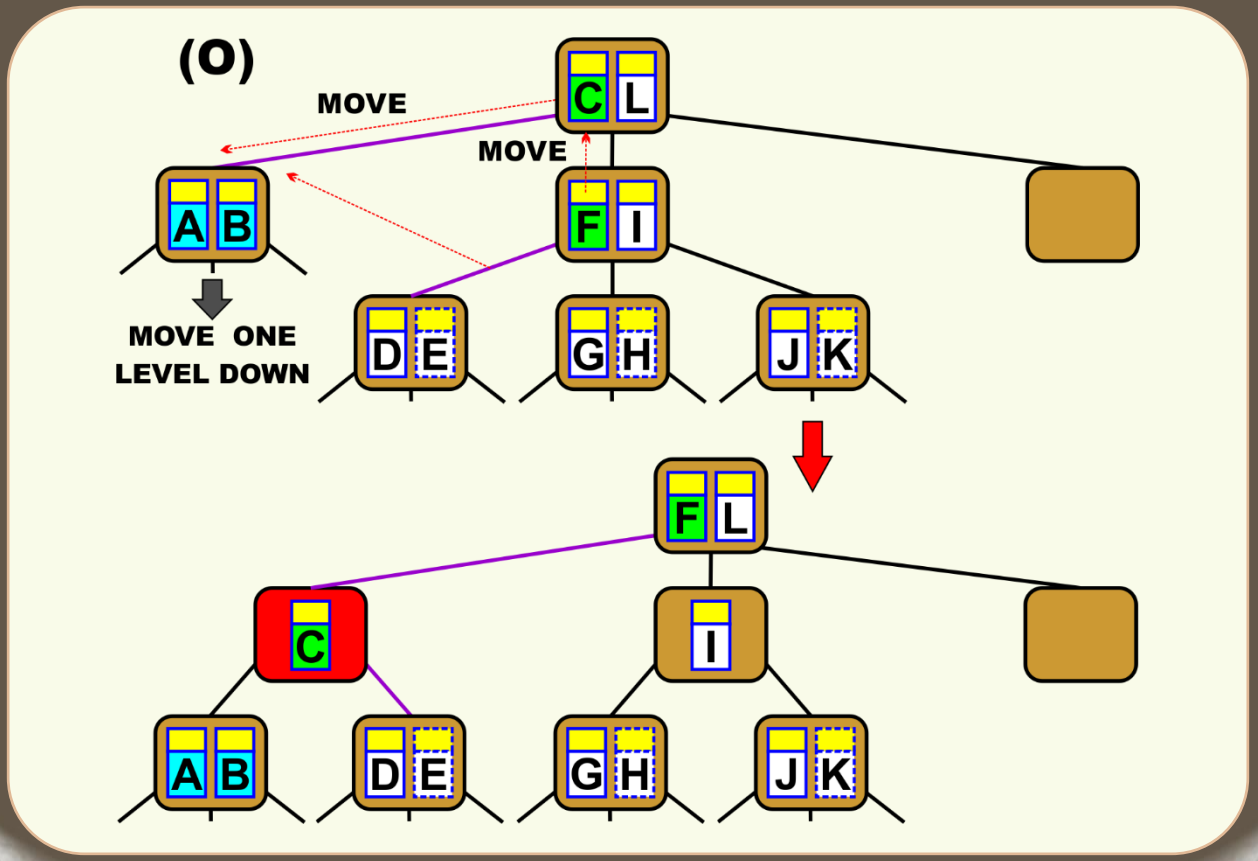


18. Move the element (key = C) from the parent node (C|I) to the one-element nearest sibling (F) together with the subtree (A|B) and the connection to its parent to this merged sibling (C|F). Connect the node (A|B) to this merged. Next, finish the delete operation.



Remove Operation

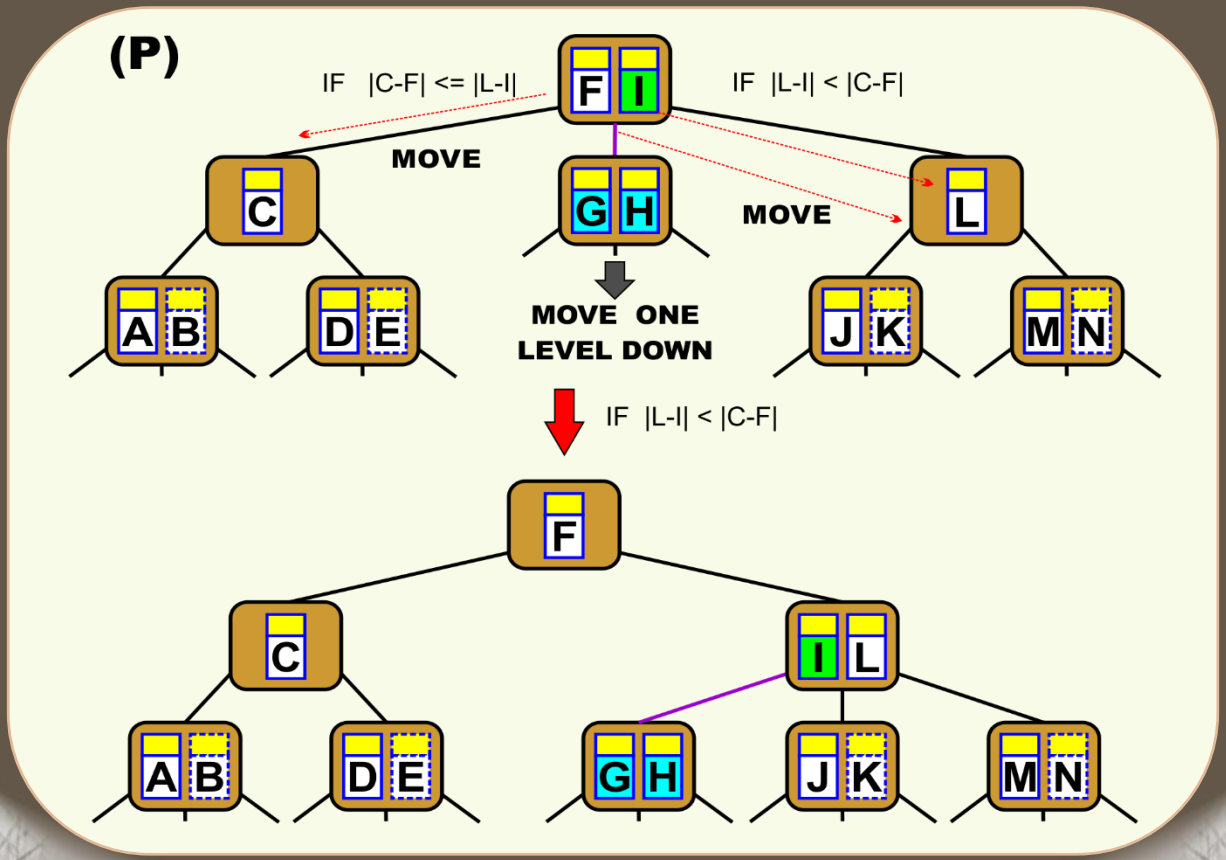
19. Create a new node and move the closest parent element (key = C) to this new node (in red). Next, move the nearest sibling element (F) to the parent node instead to the position of the moved parent element (C). Switch the closest child (D|E) of this nearest sibling to the newly created node. Connect the rebalanced subtree (A|B) to the newly created node as well. Next, finish the delete operation.



Remove Operation



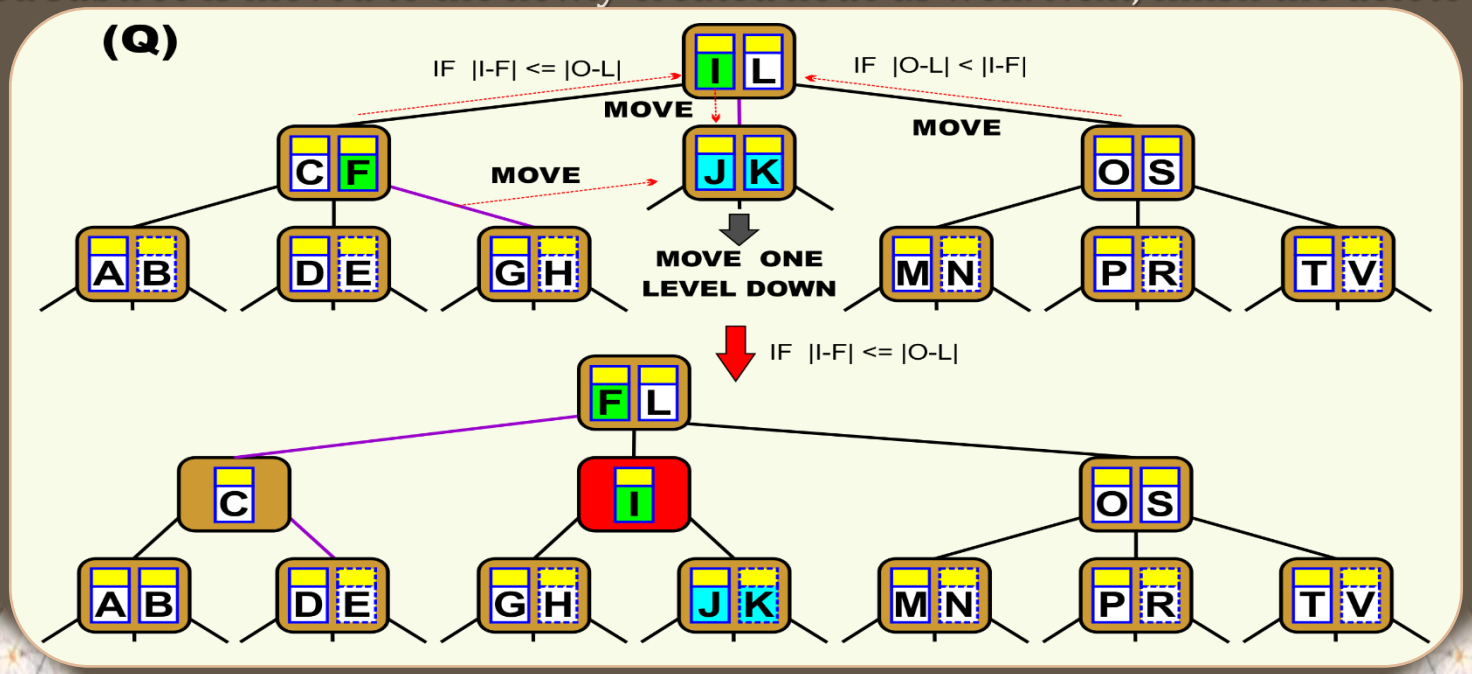
20. Move and merge the left or right element of the parent node of this reduced subtree root together with this subtree with its left or right sibling if only it contains only a single element, else go to step 21. Choose the sibling on the basis of the lower distance between the left parent element and the element of the left sibling or between the right parent element and the element of the right sibling. The Fig. P shows the situation when the distance to the right sibling is lower than to the left one. The second situation is symmetrical. Next, finish this operation.



Remove Operation



21. For the reduced subtree (A|B) which is a middle child of its parent, move the rightmost element from the left sibling if its key is more distant to the key of the right parent element than the distance of the key of the leftmost element for the right sibling to the left parent element. In the symmetric case, move the leftmost element of the right sibling. The selected sibling element is moved to the parent node, and the element from the parent node that is the closest to the elements of the reduced subtree is moved to the newly created node (in red). The closest child to the reduced subtree (G|H) of the subtree (C|F) from which the element was borrowed to the parent of the reduced subtree is moved to the newly created node as well. Next, finish the delete operation.

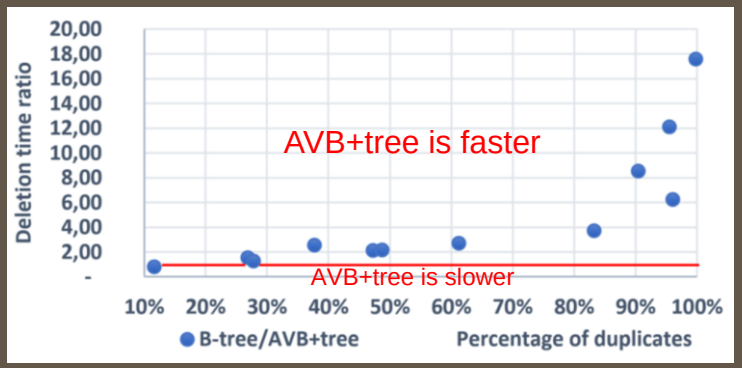


Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

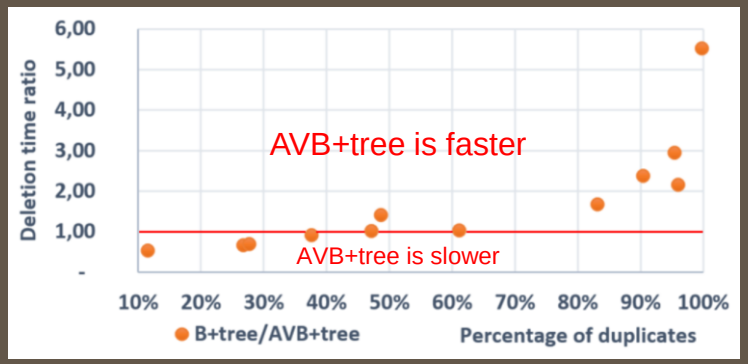
Efficiency of Remove Operation



The efficiency comparisons of Remove Operations of B-tree and AVB+tree:

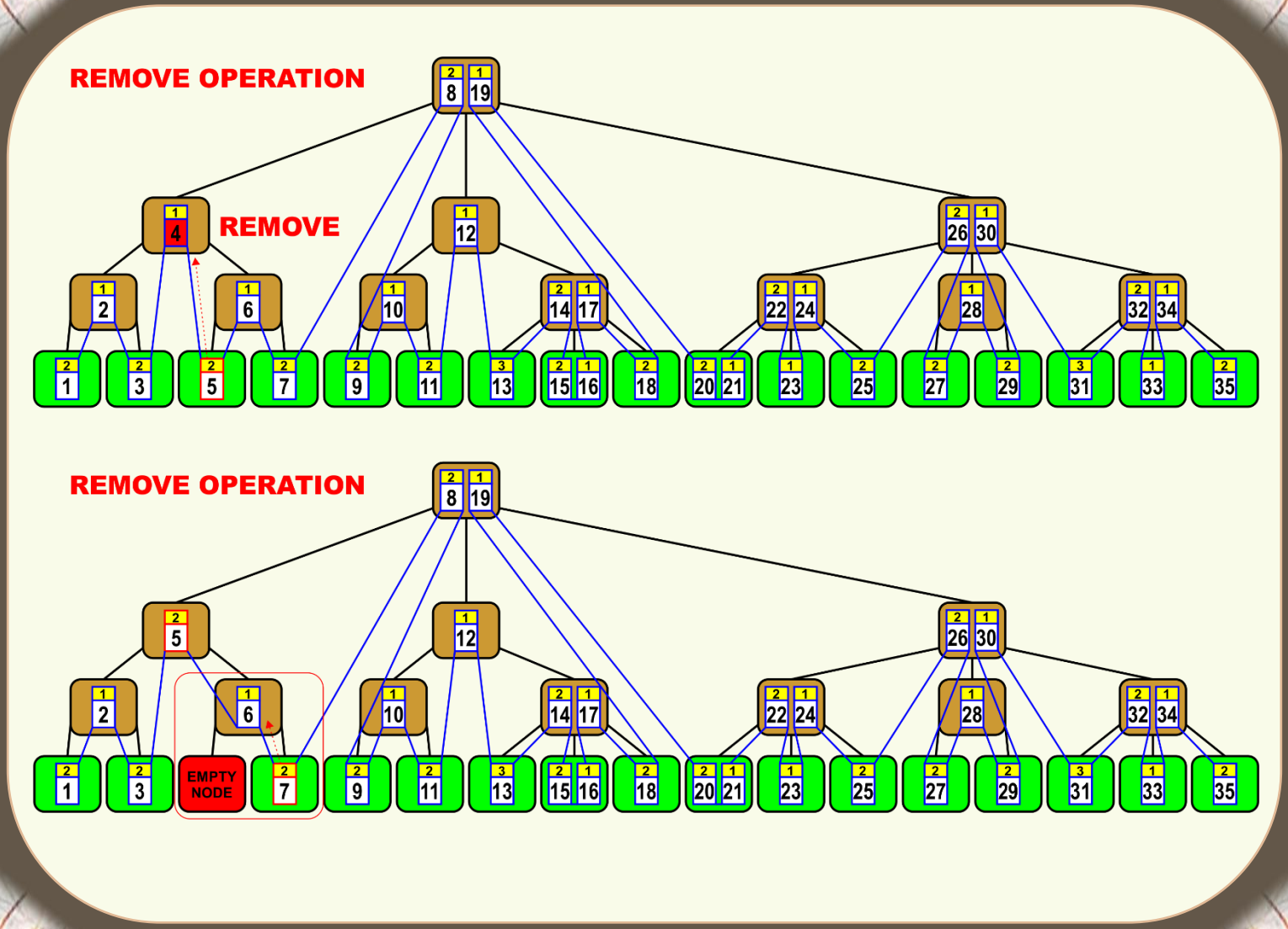


The efficiency comparisons of Remove Operations of B+tree and AVB+tree:



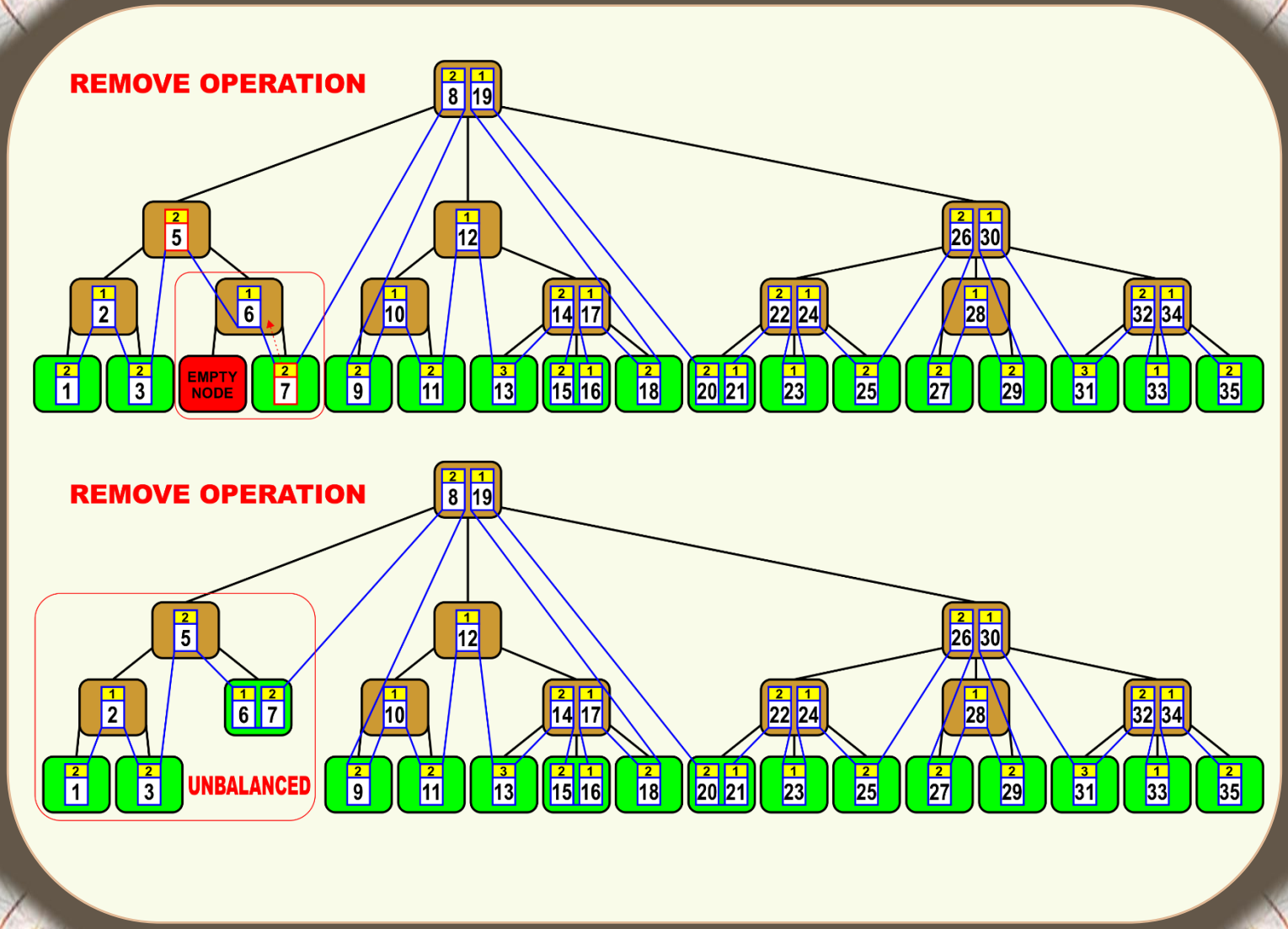
Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Example of Remove



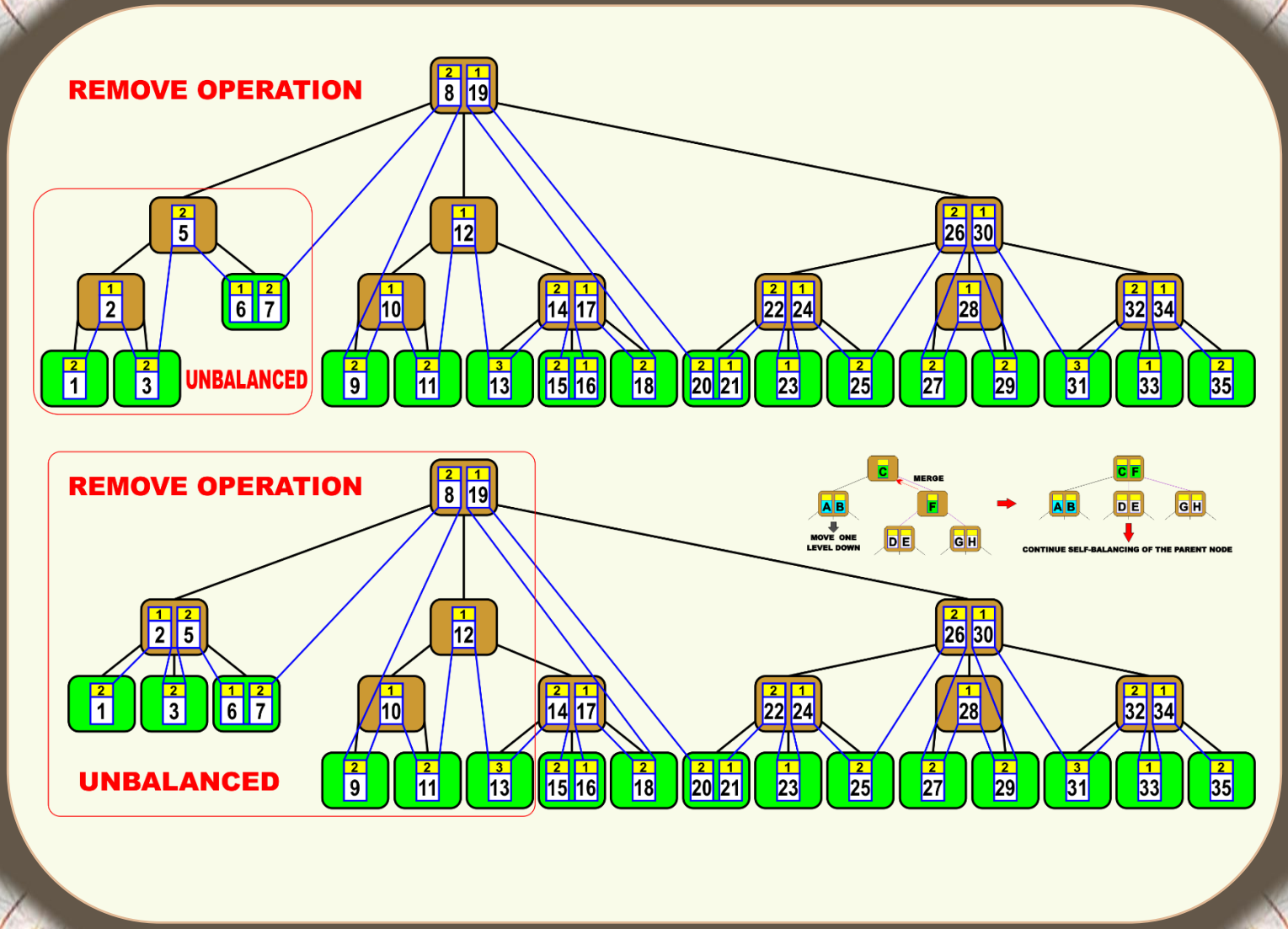
Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Example of Remove



Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

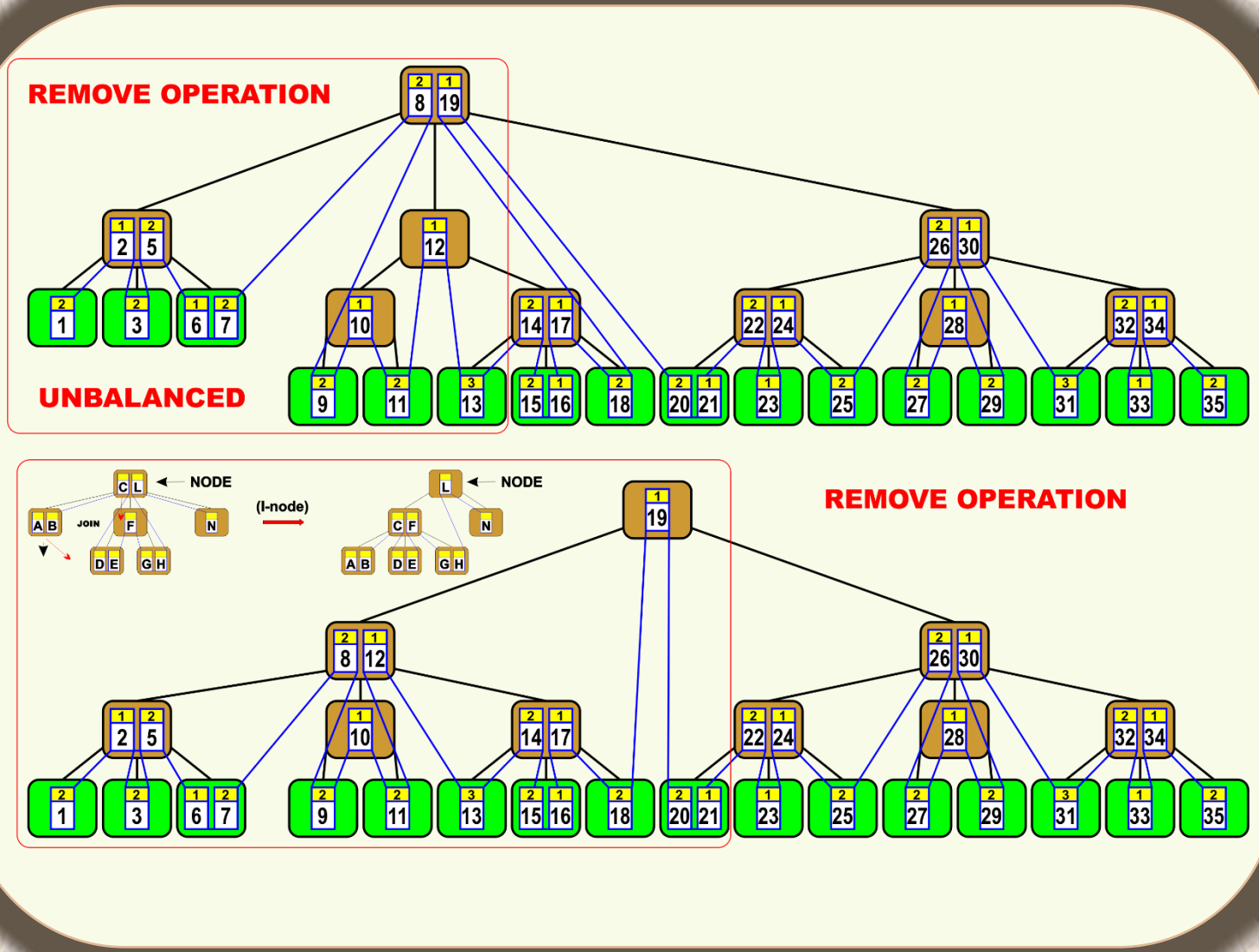
Example of Remove



Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Example of Remove





Less than logarithmic expected computational complexity
(typically constant) for data containing duplicates!

Update Operation



- ✓ The Update operation is a simple sequence of Remove and Insert operations because it is not possible to simply update a value in an element because of the structure of AVB+trees which represent various relations.
- ✓ Data can be easily updated (a value can be changed) only in those structures which do not represent relations, e.g. unsorted arrays, lists, or tables.
- ✓ The Update operation on an AVB+tree removes the old key (value) from this structure using the Remove operation and inserts an updated one using the Insert operation.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Search Operation



The Search operation in the AVB+tree is processed as follows:

1. Start from the root and go recursively down along the branches to the descendants until the searched key or the leaf is not achieved after the following rules:
   - If one of the keys stored in the elements of this node equals to the searched key, return the pointer to this element;
   - else go to the left child node if the searched key is less than the key represented by the leftmost element in this node;
   - else go to the right child node if the searched key is greater than the key represented by the rightmost key in this node;
   - else go to the middle child node.
2. If the leaf is achieved and one of the stored elements in this leaf contains the searched key, return the pointer to this element, else return the null pointer.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

GetMin and GetMax Operations



The GetMin and GetMax operations can be implemented in two different ways dependently on how often extreme elements are used in other computations using an AVB+tree structure:

1. The first way is used when extreme keys are not often used.
   In this case, it is necessary to start from the root node and always go along the left tree branches until the leaf is achieved and in its leftmost element (if there are two) is the minimum key (value) stored in this tree.
   Similarly, we go always along the right branches starting from the root node until the leaf is achieved and in its rightmost element (if there are two) is the maximum key (value) stored in this tree. These operations take log Ň time, where Ň is the number of elements stored in the tree, which is equal the number of unique keys (values) of the data.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

GetMin and GetMax Operations



The GetMin and GetMax operations can be implemented in two different ways dependently on how often extreme elements are used in other computations using an AVB+tree structure:

2. The second way is used when extreme keys are often used and should be quickly available (in constant time).
   In this case, the leftmost (minimum) and rightmost (maximum) elements of the leftmost and rightmost leaves appropriately are additionally pointed from the class implementing the AVB+tree. If using these extra pointers they are automatically updated when the minimum or maximum element is changed, and
   the minimum and maximum element can be easily recognized because its neighbor connection to the left or right neighbor element is set to null.

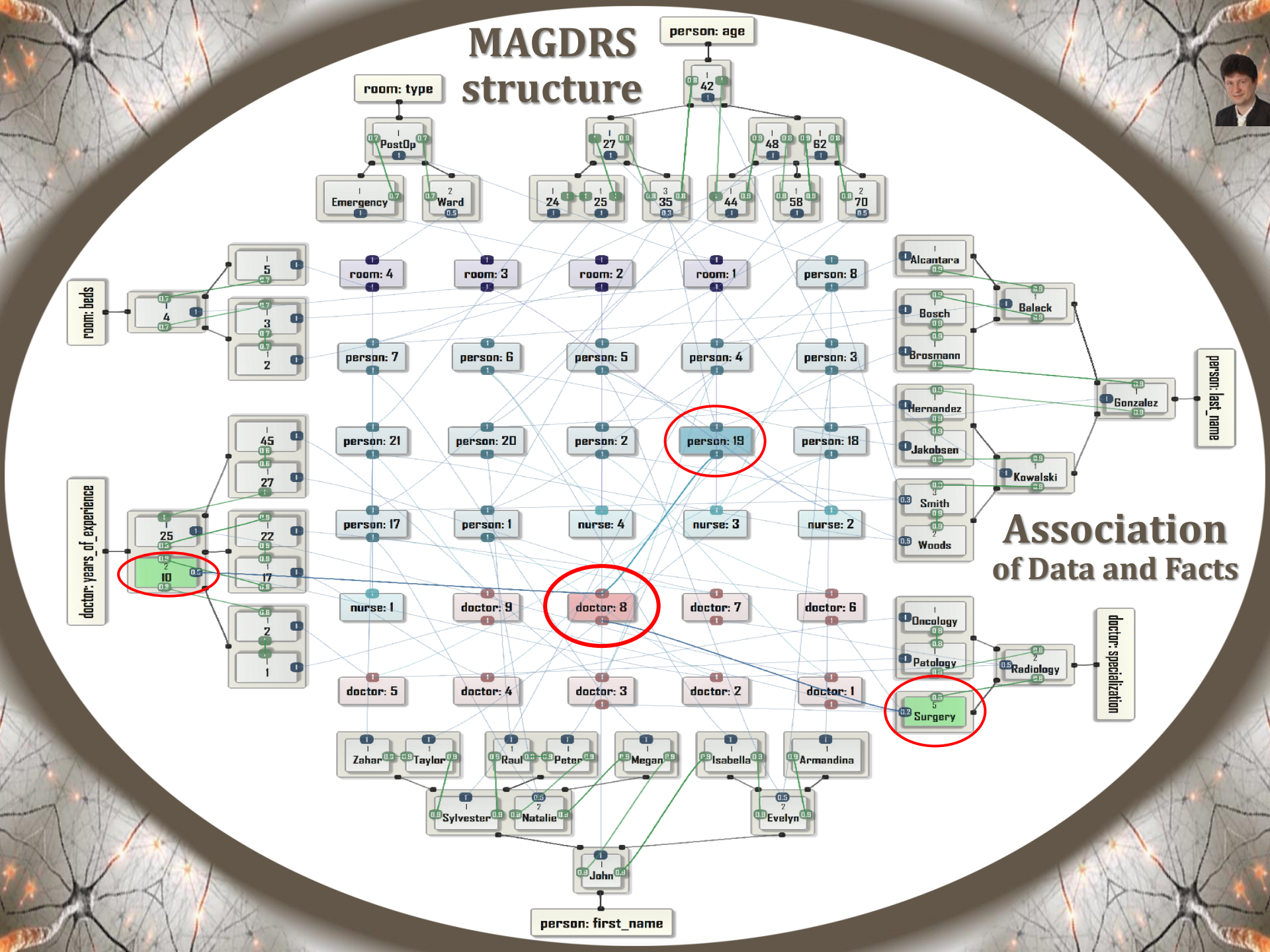
Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

MAGDRS
associative structures



- ✓ MAGDRS structure is a generalization of AGDS structure for storing data and their relationships from various databases in the associative.
- ✓ The objects of various kinds (from various tables) can be represented and connected according to the relationships that come from primary and foreign keys of the relational data model as well as being extended by new relations coming from data and relationships processing.
- ✓ They can also be used for various inferences, filtering, clustering, classification, similarity finding, or analysis of data and relationships.
- ✓ They also aggregate and all count duplicates of the same categories stored in various data tables of the whole database.
- ✓ They can save a lot of memory and computation time and effort of programmers because drawing conclusions is pretty easy and fast!

Less than logarithmic expected computational complexity
(typically constant) for data containing duplicates!


MAGDRS
structure
person: age
room: type
room: beds
doctor: years_of_experience
person: last_name
doctor: specialization
person: first_name
Association
of Data and Facts
PostOp
Emergency
Ward
room: 4
room: 3
room: 2
room: 1
person: 8
person: 7
person: 6
person: 5
person: 4
person: 3
person: 21
person: 20
person: 2
person: 19
person: 18
person: 17
person: 1
nurse: 4
nurse: 3
nurse: 2
nurse: 1
doctor: 9
doctor: 8
doctor: 7
doctor: 6
doctor: 5
doctor: 4
doctor: 3
doctor: 2
doctor: 1
Alcantara
Bosch
Brosmann
Balack
Hernandez
Jakobsen
Kowalski
Smith
Woods
Gonzalez
Oncology
Patology
Radiology
Surgery
Zahar
Taylor
Raul
Peter
Megan
Isabella
Armandina
Sylvester
Natalie
Evelyn
John



Conclusions

- ✓ **AGDS structures combined with AVB+trees provide incredibly fast access to any data stored and sorted for all attributes simultaneously, combining sorted lists, aggregation and counting of all duplicates with fast access coming from the idea of binary search trees (BST).**
- ✓ **AGDS + AVB+trees stores data together with the most common vertical and horizontal relations, so there is no need to loop and search for these relations but to grab them when needed.**
- ✓ **Typical operations on AGDS + AVB+trees structures have pessimistically logarithmic time, but the expected time complexity on typical real data containing many duplicates is constant.**
- ✓ **AGDS structures can be used as 1st and 2nd class generation neural networks to produce various inferences, for classification, clustering, similarity finding, recognition, and other CI or KE tasks.**



Literature and Bibliography:

1. **A. Horzyk**, A. Czajkowska, Associative Pattern Matching and Inference Using Associative Graph Data Structures, In: Proc. of 18-th Int. Conf. ICAISC 2019, Rutkowski, L., Scherer, R., Korytkowski, M., Pedrycz, W., Tadeusiewicz, R., Zurada, J.M. (Eds.), Artificial Intelligence and Soft Computing, Springer-Verlag, LNAI 10509, pp. 371-383, 2019, DOI 10.1007/978-3-030-20915-5_34.
2. **A. Horzyk** and K. Gołdon, Associative Graph Data Structures Used for Acceleration of K Nearest Neighbor Classifiers, In: 27th International Conference on Artificial Neural Networks (ICANN 2018), Springer-Verlag, LNCS 11139, pp. 648-658, 2018. presentation
3. **A. Horzyk**, Associative Graph Data Structures with an Efficient Access via AVB+trees, In: 2018 11th International Conference on Human System Interaction (HSI), 2018, IEEE Xplore, pp. 169 - 175, DOI: 10.1109/HSI.2018.8430973 - presentation
4. **A. Horzyk** and J.A. Starzyk, Multi-Class and Multi-Label Classification Using Associative Pulsing Neural Networks, In: 2018 IEEE World Congress on Computational Intelligence (WCCI 2018), 2018 International Joint Conference on Neural Networks (IJCNN 2018), IEEE Xplore, pp. 427-434, 2018. - presentation
5. **A. Horzyk**, J. A. Starzyk, J. Graham, ***Integration of Semantic and Episodic Memories***, IEEE Transactions on Neural Networks and Learning Systems, Vol. 28, Issue 12, Dec. 2017, pp. 3084 - 3095, 2017, DOI: 10.1109/TNNLS.2017.2728203.
6. **A. Horzyk**, J.A. Starzyk, ***Multi-Class and Multi-Label Classification Using Associative Pulsing Neural Networks***, IEEE Xplore, In: 2018 IEEE World Congress on Computational Intelligence (WCCI IJCNN 2018), 2018, (in print).
7. **A. Horzyk**, J.A. Starzyk, ***Fast Neural Network Adaptation with Associative Pulsing Neurons***, IEEE Xplore, In: 2017 IEEE Symposium Series on Computational Intelligence, pp. 339 -346, 2017, DOI: 10.1109/SSCI.2017.8285369.
8. **A. Horzyk**, ***Deep Associative Semantic Neural Graphs for Knowledge Representation and Fast Data Exploration***, Proc. of KEOD 2017, SCITEPRESS Digital Library, pp. 67 - 79, 2017, DOI: 10.13140/RG.2.2.30881.92005.
9. **A. Horzyk**, ***Neurons Can Sort Data Efficiently***, Proc. of ICAISC 2017, Springer-Verlag, LNAI, 2017, pp. 64 - 74, ICAISC BEST PAPER AWARD 2017 sponsored by Springer.
10. **A. Horzyk**, J. A. Starzyk and Basawaraj, ***Emergent creativity in declarative memories***, IEEE Xplore, In: 2016 IEEE Symposium Series on Computational Intelligence, Greece, Athens: Institute of Electrical and Electronics Engineers, Curran Associates, Inc. 57 Morehouse Lane Red Hook, NY 12571 USA, 2016, ISBN 978-1-5090-4239-5, pp. 1 - 8, DOI: 10.1109/SSCI.2016.7850029.
11. **Horzyk, A.**, ***How Does Generalization and Creativity Come into Being in Neural Associative Systems and How Does It Form Human-Like Knowledge?***, Elsevier, Neurocomputing, Vol. 144, 2014, pp. 238 - 257, DOI: 10.1016/j.neucom.2014.04.046.
12. **A. Horzyk**, ***Innovative Types and Abilities of Neural Networks Based on Associative Mechanisms and a New Associative Model of Neurons*** - Invited talk at ICAISC 2015, Springer-Verlag, LNAI 9119, 2015, pp. 26 - 38, DOI 10.1007/978-3-319-19324-3_3.
13. **A. Horzyk**, ***Human-Like Knowledge Engineering, Generalization and Creativity in Artificial Neural Associative Systems***, Springer-Verlag, AISC 11156, ISSN 2194-5357, ISBN 978-3-319-19089-1, ISBN 978-3-319-19090-7 (eBook), Springer, Switzerland, 2016, pp. 39 – 51, DOI 10.1007/978-3-319-19090-7.



Adrian Horzyk

horzyk@agh.edu.pl

Google: Horzyk

AGH

University of Science and Technology in Krakow, Poland