**ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ**

**ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ**

**ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ**

Διπλωματική εργασία για το Μεταπτυχιακό Δίπλωμα Ειδίκευσης στα

<<Ολοκληρωμένα Συστήματα Υλικού και Λογισμικού>>



Τίτλος Μεταπτυχιακής Διπλωματικής Εργασίας:

**«Αλγόριθμοι Εξόρυξης Διαδικασιών στο Περιβάλλον Ανάπτυξης Spark»**

**Bλάσης Πίτσιος**

**Α.Μ.: 220**

**Επιβλέπων Καθηγητής:**

Χρήστος Ζαρολιάγκης, Καθηγητής Παν. Πατρών

**Τριμελής Εξεταστική Επιτροπή:**

Χρήστος Ζαρολιάγκης, Καθηγητής Παν. Πατρών

Σπυρίδων Σιούτας, Καθηγητής Παν. Πατρών

Γιάννης Τζήμας, Αναπ. Καθηγητής ΤΕΙ Δυτικής Ελλάδος

**Ευχαριστίες**

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. **Χρήστο Ζαρολιάγκη** που ανέλαβε την επίβλεψη της διπλωματικής μου εργασίας και μου έδωσε την δυνατότητα να ασχοληθώ με το περιβάλλον ανάπτυξης εφαρμογών Spark.

Επίσης, θα ήθελα να ευχαριστήσω τους κ.κ. **Σπυρίδων Σιούτα** και **Γιάννη Τζήμα** που δέχτηκαν να είναι μέλη της εξεταστικής επιτροπής για την εξέταση και κρίση της μεταπτυχιακής μου εργασίας.

Ακόμα θα ήθελα να ευχαριστήσω το διδάκτορα **Μανόλη Βιέννα**, για τη βοήθεια και τις συμβουλές του κατά την διάρκεια της εκπόνησης της μεταπτυχιακής μου εργασίας.

**Σύνοψη**

Η εξόρυξη διαδικασιών (process mining) αποτελεί έναν συνδετικό κρίκο ανάμεσα στην εξόρυξη δεδομένων (data mining) και στην διαχείριση διαδικασιών σε επιχειρήσεις (business process management). Συγκεκριμένα αποτελεί μια οικογένεια από τεχνικές που υποστηρίζουν την ανάλυση των διαδικασιών βασιζόμενες σε διάφορα σύνολα γεγονότων (event logs). Σκοπός της εξόρυξης διαδικασιών (process mining) είναι η κατανόηση αλλά και η βελτίωση της απόδοσης των διαδικασιών μιας επιχείρησης.

Η παρούσα πρόταση έρευνας επικεντρώνεται στο κατά πόσο είναι δυνατή η παραλληλοποίηση τέτοιων αλγορίθμων εξόρυξης διαδικασιών και ποιο είναι το όφελος από την παράλληλη εκτέλεση τέτοιων αλγορίθμων. Συγκεκριμένα μελετάται ο αλγόριθμος Alpha (Alpha Algorithm), ο οποίος κατασκευάζει διαγράμματα PetriNet από ακολουθίες γεγονότων. Στόχος αποτελεί η ανάπτυξη του συγκεκριμένου αλγορίθμου χωρίς την χρήση τεχνικών παραλληλοποίησης και στη συνέχεια η ανάπτυξη του ίδιου αλγορίθμου με τεχνικές mapReduce σε περιβάλλον ανάπτυξης Spark, όπου ο αλγόριθμος θα εκτελείται παράλληλα. Στη συνέχεια θα γίνει εκτέλεση πειραμάτων σε πραγματικά δεδομένα και αξιολόγηση των αποτελεσμάτων της σειριακής εκτέλεσης του αλγορίθμου αλλά και της παράλληλης εκτέλεσης σε μια συστάδα υπολογιστών (cluster) με χρήση του Spark framework. Η ανάπτυξη του αλγόριθμου Alpha θα υλοποιηθεί με πηγαίο κώδικα σε Scala και θα αξιολογηθεί εκτενώς πειραματικά σε πραγματικά δεδομένα.

**Λέξεις κλειδιά:** Spark, Scala, Process mining, Databricks, Alpha Algorithm

**Abstract**

Process mining is a link between data mining and business process management.   
In particular, it is a family of techniques that support the analysis of processes based on event logs. The purpose of process mining is to understand and improve the performance of business processes.

This research proposal focuses on how parallelization of such process mining algorithms is possible and what is the benefit of the parallel execution of such algorithms. In particular, the Alpha Algorithm (a process mining algorithm), which builds PetriNet graphs from events sequences, is studied. The aim is to develop this algorithm without using parallelism techniques and then to develop the same algorithm with mapReduce techniques in Spark development environment, where the algorithm will run in parallel. In the next stage,  
experiments will be performed on real data and the results will be evaluated by comparing the serial execution of the algorithm with the parallel execution on a cluster using the Spark framework. The development of the Alpha algorithm will be implemented with Scala source code and will be extensively evaluated experimentally on real data.

**Keywords:** Spark, Scala, Process mining, Databricks, Alpha Algorithm

**Περιεχόμενα**

Contents

[1. Εισαγωγή στο “process mining” 6](#_Toc19631)

[1.1 Tι είναι “process mining” 6](#_Toc19632)

[1.2 Eπιστήμη των δεδομένων και Big Data 7](#_Toc19633)

[1.3 Διάφορα είδη process mining 11](#_Toc19634)

[1.4 Πως σχετίζεται το process mining με το data mining 16](#_Toc19635)

[2. Μοντέλα και Ανακάλυψη Διαδικασιών 19](#_Toc19636)

[2.1 Εισαγωγή στα Μοντέλα διαδικασιών 19](#_Toc19637)

[2.2 Petri Nets 20](#_Toc19638)

[2.3 Alpha Algorithm – Δομικά στοιχεία (patterns) 22](#_Toc19639)

[2.4 Alpha Algorithm – Βήματα 27](#_Toc19640)

[2.5 Alpha Algorithm – Παράδειγμα 30](#_Toc19641)

[2.6 Alpha Algorithm – Περιορισμοί 32](#_Toc19642)

[3. Εισαγωγή στην γλώσσα προγραμματισμού “Scala” 36](#_Toc19643)

[3.1 Γλώσσα προγραμματισμού Scala 36](#_Toc19644)

[3.2 Λόγοι για να ασχοληθεί κάποιος με την Scala 39](#_Toc19645)

[4. Apache Spark Framework 41](#_Toc19646)

[4.1 Εισαγωγή στο Apache Spark Framework 41](#_Toc19647)

[4.2 Apache Spark Components 42](#_Toc19648)

[4.3 Ροή Προγράμματος Αpache Spark 45](#_Toc19649)

[4.4 Αρχιτεκτονική του Αpache Spark 47](#_Toc19650)

[4.5 RDD Δομές δεδομένων 49](#_Toc19651)

[4.6 Spark SQL, DataFrames and Datasets 51](#_Toc19652)

[4.7 Βασικές μέθοδοι (operations) του Apache Spark 51](#_Toc19653)

[4.8 Shared variables και αποθήκευση μεταβλητών στο Spark 53](#_Toc19654)

[5. Υλοποίηση του Alpha Algorithm 55](#_Toc19655)

[5.1 Κώδικας Υλοποίησης 55](#_Toc19656)

[5.2 Επιβεβαίωση Ορθής Λειτουργίας του Κώδικα Υλοποίησης 68](#_Toc19657)

[6. Εκτέλεση πειραματικών μετρήσεων 75](#_Toc19658)

[6.1 Πλατφόρμα Databricks 75](#_Toc19659)

[6.2 Μετρήσεις 77](#_Toc19660)

[7. Βιβλιογραφία 82](#_Toc19661)

# 1. Εισαγωγή στο “process mining”

# 1.1 Tι είναι “process mining”

Στην καθημερινή ζωή καταγράφονται πάρα πολλα γεγονότα (**events**), παραδείγματος χάριν στατιστικά από μέσα κοινωνικής δικτύωσης (facebook, twitter, linkedin), από ιστοσελίδες παραγγελιών προιόντων (ebay, amazon) κτλπ. Αυτό παρέχει μια αστείρευτη πηγή πληροφοριών που μπορούν να παρέχουν πολύτιμη γνώση χρησιμοποιώντας την εξόρυξη διαδικασιών (**process mining**). To **process mining** μπορεί να χρησιμοποιηθεί για να παράγει μοντέλα συμπεριφοράς (**behavioral models**) που να δείχνουν τι πραγματικά κάνουν οι άνθρωποι και οι επιχειρήσεις. To αντικείμενο ενδιαφέροντος των Big Data είναι υπερβολικά επικεντρωμένο στα δεδομένα (**data**), στην αποθήκευση (**storage**) και στην επεξεργασία (**processing**) τους, και όχι τόσο στις διαδικασίες (**processes**) που θα ήταν χρήσιμο για μια επιχείρηση να αναλυθούν και να βελτιωθούν. Για αυτό ακριβώς το λόγο υπάρχει η εξόρυξη διαδικασιών (**process mining**) η οποία παρέχει μια καινούρια οπτική στα δεδομένα (Big or Small data) και παρέχει τα απαραίτητα εργαλεία για να ξεκινήσει η ανάλυση πραγματικών συμπεριφορών βασιζόμενοι σε πραγματικά γεγονότα (**event data**) που μπορούν να βρεθούν σε οποιαδήποτε επιχείρηση.

H εξόρυξη διαδικασιών (**process mining**) είναι ένα ξεχωριστό εργαλείο για κάποιον επιστήμονα δεδομένων (**data scientist**). H εξόρυξη διαδικασιών (**process mining**) γεφυρώνει το χάσμα που υπάρχει ανάμεσα στην παραδοσιακά βασιζόμενη σε μοντέλα ανάλυση διαδικασιών (**model-based process analysis**) όπως η προσομοίωση και των τεχνικών ανάλυσης που βασίζονται σε δεδομένα (**data-centric analysis techniques**) όπως το machine learning και το data mining.

Σύμφωνα με τα παραπάνω, μπορούμε να ορίσουμε την εξόρυξη διαδικασιών (**process mining**) ως μια τεχνική διαδικασίας μάνατζμεντ, που είναι χρήσιμη για την ανάλυση επιχειρηματικών διαδικασιών βασισμένες σε καταγραφή γεγονότων. Η βασική ιδέα είναι να εξορύξουμε γνώση από καταγεγραμμένα γεγονότα αποθηκευμένα σε ένα σύστημα πληροφοριών. Η εξόρυξη διεργασιών στοχεύει στο να βελτιώσει αυτό το σύστημα πληροφοριών με παρεχόμενες τεχνικές και εργαλεία για να ανακαλύψει τη διαδικασία, τον έλεγχο, τα δεδομένα και τις κοινωνικές δομές από την καταγραφή γεγονότων.

Οι τεχνικές της εξόρυξης διεργασιών συχνά χρησιμοποιούνται όταν δεν υπάρχει καμιά επίσημη περιγραφή της διαδικασίας για το πώς μπορεί να αποκτηθεί με άλλους τρόπους, ή όταν η ποιότητα μιας υπαρκτής τεκμηρίωσης είναι αμφισβητήσιμη. Για παράδειγμα, οι οικονομικοί έλεγχοι ενός συστήματος ροής μάνατζμεντ, η διεξαγωγή καταγραφής ενός συστήματος σχεδιασμού επιχειρηματικών πόρων, και το ηλεκτρονικό σύστημα καταχώρησης ενός νοσοκομείου μπορεί να χρησιμοποιηθεί για να ανακαλύψει μοντέλα που να περιγράφουν διαδικασίες, οργανισμούς και προϊόντα. Ακόμη, αυτή η καταγραφή γεγονότων μπορεί να χρησιμοποιηθεί για να συγκρίνουμε καταγεγραμμένα γεγονότα με κάποιο προγενέστερο πρότυπο για να δούμε εάν η παρατηρούμενη πραγματικότητα συμμορφώνεται σε κάποιο κατευθυντήριο ή περιγραφικό μοντέλο.

# 1.2 Eπιστήμη των δεδομένων και Big Data

Σήμερα, περισσότερο από ποτέ, υπάρχει διαθέσιμη τεράστια ποσότητα δεδομένων η οποία συλλέγεται από διάφορες πηγές. Αυτό μπορεί να γίνει αντιληπτό, αν σκεφτεί κανείς τον όγκο των δεδομένων που έχουν παραχθεί από τα προιστορικά χρόνια μέχρι το 2003 και το ότι αυτή η ποσότητα πλέον παράγεται μέσα σε 10 λεπτά της ώρας. Αυτό ακριβώς δείχνει την απίστευτη ανάπτυξη των δεδομένων. Αυτά τα δεδομένα παράγονται οποιαδήποτε στιγμή από οποιονδήποτε άνθρωπο την στιγμή που κάνει χρήση της πιστωτικής του κάρτας, την στιγμη που συνομιλεί στο τηλέφωνο, την στιγμή που αγοράζει τρόφιμα από το super market.

Η πιο σημαντική όμως πηγή δεδομένων είναι τα δεδομένα που διακινούνται μέσω Internet (**internet of events**). Υπάρχουν 4 είδη τέτοιων δεδομένων

* **Internet of content:** Στην συγκεκριμένη κατηγορία ανήκουν ιστοσελίδες από το κλασικό internet, όπως είναι γνωστό σε όλους, δηλαδή Google και Wikipedia. Και συνήθως όταν οι άνθρωποι μιλάνε για Big Data αναφέρονται σε αυτό το είδος δεδομένων.
* **Internet of people:** Στην συγκεκριμένη κατηγορία ανήκουν τα social media, όπως το Twitter και το Facebook, τα οποία παράγουν τεράστιο όγκο δεδομένων.
* **Internet of things:** Στην συγκεκριμένη κατηγορία ανήκουν τα δεδομένα που παράγονται από διάφορες συσκευές που είναι συνδεδεμένες στο Internet όπως ο εκτυπωτής, η τηλεόραση, οι παιχνιδομηχανές. Στο μέλλον όλο και περισσότερες συσκευές θα συνδέονται στο Internet, όπως το το ψυγείο ή το πλυντήριο και αυτό θα έχει σαν αποτέλεσμα την παραγωγή ακόμα μεγαλύτερου όγκου δεδομένων.
* **Internet of places:** Όταν παραδείγματος χάριν ο χρήστης χρησιμοποιεί το κινητό του τηλέφωνο το οποίο έχει πάνω του διάφορους αισθητήρες τοποθεσίας και καταγράφει την θέση του. Αυτό είναι άλλη μια πηγή δεδομένων.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img1.png  **Εικόνα 1:**  Οι 4 πηγές παραγωγής δεδομένων |

Εύκολα λοιπόν γίνεται κατανοητό το πως τα δεδομένα έχουν αποκτήσει τεράστια σημασία τα τελευταία χρόνια αλλά και γιατί ο κόσμος ασχολείται όλο και περισσότερο μαζί τους. Αλλά η πρόκληση σήμερα δεν είναι η παραγωγή περισσότερων δεδομένων αλλά η μετατροπή των δεδομένων αυτών σε δεδομένα με πραγματική αξία.

Τα δεδομένα (**Big Data**) έχουν τα ακόλουθα τέσσερα(4) χαρακτηριστικά

* **Volume** (όγκος δεδομένων)
* **Velocity** (ταχύτητα): Η ταχύτητα είναι μια μεγάλη πρόκληση για τους μηχανικούς που επεξεργάζονται τα δεδομένα καθώς δεν είναι μόνο τεράστιος ο όγκος των δεδομένων που αποτελεί πρόκληση αλλά και το ότι παράγονται με μεγάλο ρυθμό και αλλάζουν πολύ γρήγορα.
* **Variety** (ποικιλία): Δεν υπάρχει μόνο ένα είδος δεδομένων, αλλά πάρα πολλά διαφορετικά είδη τα οποία ποικίλουν από κείμενο σε εικόνες αλλά και πολλά άλλα είδη. Έτσι είναι μεγάλη η πρόκληση του να συνδυαστούν όλα αυτά τα δεδομένα από εντελώς διαφορετικές πηγές.
* **Veracity** (εγκυρότητα): Αυτό σημαίνει ότι πότε δεν μπορούμε να είμαστε σίγουροι για την ακρίβεια των δεδομένων που έχουν καταγραφεί. Ένα παράδειγμα είναι ότι ποτέ δεν μπορούμε να είμαστε σίγουροι για το ποιος είναι ο χρήστης της συσκευής που παράγει τα δεδομένα καθώς αυτά μπορεί να διαφοροποιούνται από χρήστη σε χρήστη.

Λόγω λοιπόν των παραπάνω, τα τελευταία χρόνια έχει δημιουργηθεί ένα νέο επάγγελμα στην επιστήμη των υπολογιστών, αυτό του επιστήμονα δεδομένων (**data scientist**). Οι στόχοι ενός data scientist είναι να συλλέξει, να αναλύσει αλλά και να μεταφράσει τα δεδομένα από πολλές διαφορετικές πηγές αλλά και να τα επεξεργαστεί με τέτοιο τρόπο ώστε να εξάγει χρήσιμα αποτελέσματα. Είναι βέβαιο ότι τα επόμενα χρόνια λόγω της τεράστιας ανάπτυξης των δεδομένων το συγκεκριμένο επάγγελμα θα αποκτήσει όλο και μεγαλύτερη αξία.

Στόχος ενός data scientist είναι να απαντήσει τις παρακάτω ερωτήσεις που σχετίζονται με κάποια γεγονότα (**events**)

* Τι συνέβει; Αν παράδειγμα εμφανιστούν κάποια bottlenecks ή κάποιες αποκλίσεις (deviations) στα καταγεγραμμένα γεγονότα (**events**) τότε αυτό σημαίνει ότι κάτι μη προβλεπόμενο προέκυψε στην διαδικασία.
* Η δεύτερη ερώτηση που προκύπτει είναι γιατί προέκυψαν τα παραπάνω. Που υπήρξε η καθυστέρηση; Γιατί τα γεγονότα ακολούθησαν ένα μη αναμενόμενο μονοπάτι;
* Οι δυο παραπάνω ερωτήσεις αφορούν το παρελθόν. Αλλά φυσικά ο data scientist πρέπει να απαντήσει ερωτήσεις για το μέλλον. Άρα η τρίτη ερώτηση είναι τι θα συμβεί στο μέλλον και τι μπορούμε να προβλέψουμε από τις πληροφορίες που έχουμε ώστε να κάνουμε προβλέψεις για το τι θα συμβεί.
* Η τέταρτη ερώτηση της επιστήμης των δεδομένων είναι να αναρωτηθεί κάποιος ποιο είναι το καλύτερο επιθυμητό αποτέλεσμα που μπορεί να συμβεί σε μια διαδικασία. Άρα λοιπόν, ο data scientist πρέπει να προτείνει συγκεκριμένα πράγματα ώστε να βελτιωθεί η τρέχουσα καταστασή και να αφαιρεθούν τα bottlenecks και τα γεγονότα (events) να μην αποκλίνουν από την επιθυμητή συμπεριφορά που θα έχει σαν επακόλουθο την παροχή μιας καλύτερης υπηρεσίας.

Για παράδειγμα, μπορούμε να ανατωτηθούμε για τις 4 αυτές ερωτήσεις στις ροές παροχής διαδικασιών φροντίδας σε ένα νοσοκομείο. Θα ήταν δυνατό να ερωτηθούν οι παρακάτω ερωτήσεις

* Γιατί οι ασθενείς περιμένουν τόσο πολύ;
* Τι προκαλεί αυτές τις καθυστερήσεις;
* Ακολουθούν οι γιατροί τις κατευθυντήριες γραμμές του εκάστοτε νοσοκομείου;
* Μπορούμε να προβλέψουμε τις καθυστερήσεις;
* Πόσο προσωπικό θα χρειαστεί τις επόμενες μέρες ένα νοσοκομείο;
* Πως μπορούν να μειωθούν τα κόστη χωρίς να μειωθεί η ποιότητα;

Έχει γίνει αρκετή ανάλυση δεδομένων στα νοσοκομεία και έχει διαπιστωθεί ότι διαφορετικοί γιατροί επεξεργάζονται παρόμοια πράγματα με εντελώς διαφορετικό τρόπο και αυτός είναι ένας λόγος για τις μεγάλες αναμονές των ασθενών. Αυτές λοιπόν οι ερωτήσεις ως ένα σημείο μπορούν να οριστούν σαν ένα **business process**.

Αλλά ακόμα και αν ψάξουμε για δεδομένα σε μια συσκευή, όπως μια συσκευή ακτίνων Χ, θα δούμε ότι παράγεται ένας πολύ μεγάλος αριθμός δεδομένων και μπορούν να ερωτηθούν διάφορες ερωτήσεις που όλες μπορούν να απαντηθούν αναλύοντας τα δεδομένα προσεκτικά. Ορισμένα παραδείγματα ερωτήσεων

* Πως ακριβώς χρησιμοποιούνται αυτές οι συσκευές στα νοσοκομεία;
* Πότε αυτές οι συσκευές ξεκινάνε να παρουσιάζουν προβλήματα;
* Πότε καταστρέφονται εντελώς και δεν είναι πλέον λειτουργικές;
* Ποια εξαρτήματα πρέπει να αντικατασταθούν;
* Όταν η συσκευή σταματήσει να λειτουργεί μπορούμε να παράγουμε διαγνωστικές πληροφορίες που να υποδεικνύουν ποια εξαρτήματα έχουν καταστραφεί, ώστε να είναι δυνατή η άμεση επισκευή από έναν μηχανικό;
* Μπορούμε να προβλέψουμε πότε μια συσκευή θα έχει βλάβη;
* Μπορούμε να μάθουμε από τα υπάρχοντα προβλήματα ποια εξαρτήματα χρειάζονται βελτίωση ώστε μια συσκευή να έχει μεγαλύτερη διάρκεια λειτουργείας χωρίς προβλήματα;

Όπως γίνεται λοιπόν αντιληπτό, κάποιος data scientist χρειάζεται διαφορετικές δεξιότητες για να απαντήσει όλες αυτές τις ερωτήσεις. Κάποιες προφανείς δεξιότητες είναι η στατιστική (**statistics**) και η εξόρυξη δεδομένων (**data mining**). Ακόμα θα πρέπει να μπορεί να χειριστεί απίστευτα μεγάλες βάσεις δεδομένων αλλά και να έχει γνώση κοινωνικών επιστημών για να μπορεί να αντιλαμβάνεται έννοιες όπως η δεοντολογία (**ethics**) και η ιδιωτικότητα (**privacy**). Το πιο σηματικό όμως είναι να μπορεί να εφαρμόσει τεχνικές εξόρυξης διαδικασιών (**process mining techniques**) πάνω στα δεδομένα ώστε να μπορεί να βελτιώσει αλλά και να μάθει τις διαδικασίες μέσα σε μια επιχείρηση ή έναν οργανισμό.

Με λίγα λόγια μπορούμε να πούμε ότι το **process mining** είναι μια υπο-κατηγορία του **data science**, αλλά στοχεύει στην ανάλυση των διαδικασιών βασιζόμενο στα δεδομένα και το κλειδί είναι οι διαδικασίες (**processes**). Το **process mining** διαφέρει από την εξόρυξη δεδομένων (**data mining**) διότι δεν ενδιαφερόμαστε να απομονώσουμε αποφάσεις ή πρότυπα χαμηλού επιπέδου (low level patterns). Ο βασικός στόχος του **process mining** είναι η βελτιστοποίηση των end-to-end διαδικασιών σύμφωνα με την αλληλεπίδραση ανάμεσα στα δεδομένα γεγονότων (**event data**), στις διαδικασίες (**processes**) και στα μοντέλα διαδικασιών (**process models**). Όλα αυτά μπορούν να αναφερθούν και σαν **process intelligence**.

# 1.3 Διάφορα είδη process mining

Από το προηγούμενο κεφάλαιο γίνεται κατανοητό ότι έχουμε πολύ μεγάλο μέγεθος δεδομένων, αλλά το **process mining** δεν έχει να κάνει με την συλλογή δεδομένων αλλά με την ανάλυση διαδικασιών. Το **process mining** γεφυρώνει το κενό ανάμεσα στην κλασική ανάλυση διαδικασιών μέσω μοντέλων (**process model analysis**) και της ανάλυσης η οποία βασίζεται στα δεδομένα όπως τo data mining και το machine learning. Αυτό συμβαίνει γιατί το **process mining** επικεντρώνεται στις διεργασίες χρησιμοποιώντας ταυτόχρονα πραγματικά δεδομένα. Στο κλασικό **data mining**,συνήθως οι **data scientists** συνήθως δεν επεξεργάζονται τις διαδικασίες και ειδικά τις end-to-end διαδικασίες. Αντίθετα στις περιοχές που αφορούν την ανάλυση διαδικασιών μέσω μοντέλων (**process model analysis**), συνήθως τα δεδομένα αγνοούνται. Για αυτό το λόγο το **process mining** είναι πολύ σημαντικό επειδή απαντάει ερωτήσεις σχετιζόμενες με την απόδοση (**performance**) των διαδικασιών αλλά και με την συμμόρφωση (**compliance**) των δεδομένων πάνω σε μια συγκεκριμένη διαδικασία.

Το αρχικό σημείο από το οποίο θα ξεκινήσει κάποιος το **process mining** είναι τα **event data**. Στην παρακάτω εικόνα βλέπουμε έναν πίνακα στον οποίο κάθε γραμμή αναφέρεται σε ένα **event**.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img2.png  **Εικόνα 2:**  Αλληλουχία γεγονότων δεδομένων (events) |

Ένα **event** έχει συγκεκριμένες ιδιότητες (**properties**) και για αυτό υπάρχουν διαφορετικοί τύποι ιδιοτήτων.

* Το πρώτο **property** είναι το **caseId**. Στην συγκεκριμένη περίπτωση κάθε εγγραφή του πίνακα αναφέρεται σε έναν μαθητή ο οποίος δίνει μια εξέταση σε κάποιο μάθημα. Άρα το **caseId** αναφέρεται στον μαθητή.
* Το δεύτερο **property** είναι το **activity name** και αναφέρεται στην εξέταση.
* Το τρίτο **property** είναι το **timestamp** που στην συγκεκριμένη περίπτωση είναι η ημερομηνία της εξέτασης.

Ακόμα θα μπορούσαν να υπάρχουν και άλλα δεδομένα στο παράδειγμα όπως η βαθμολογία, αλλά τα παραπάνω τρία **properties** είναι τα απολύτως απαραίτητα για μια **process mining** ανάλυση.

Κύριως στόχος του **process mining** είναι η σχέση ανάμεσα στα μοντέλα διαδικασιών (**process models**) και τα γεγονότα (**event data**). Υπάρχουν τρία είδη σχέσεων ανάμεσα σε **process models** και **event data.** Τα οποία είναι τα **Play-out**, **Play-in** και **Replay**.

Στο **Play-out** η βασική ιδέα είναι ότι ξεκινάς από ένα μοντέλο και από αυτό το μοντέλο παράγεις την συμπεριφορά της διαδικασίας. Αυτό φαίνεται ξεκάθαρα στην παρακάτω εικόνα όπου έχουμε ένα **BPMN model**.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img3.png  **Εικόνα 3:**  Play-out. Από ένα BPMN model κατασκευάζεται ένα event (abdeg) |

Tο συγκεκριμένο BPMN model έχει να κάνει με τον χειρισμό αιτήσεων ταξιδιού. Πάντα ξεκινάει από κάποιο **activity a** όπου καταχωρείται το αίτημα ταξιδιού. Συμβαίνει σε μια συγκεκριμένη χρονική στιγμή και εκτελείται από ένα συγκεκριμένο άτομο. Παρατηρούμε ότι στο μοντέλο αυτό παράγεται το event abdeg με τελικό στάδιο το **activity g**. Το μοντέλο αυτό βέβαια μπορεί να παράξει ένα μεγάλο συνδιασμό events ανάλογα με τις επιλογές που κάνει ο κάθε χρήστης που το εκτελεί. Το event abdeg είναι μόνο ένα από αυτά. Με λίγα λόγια τα events αυτά αποτελούν πιθανούς τρόπους να γίνει playing out το μοντέλο. Αν κάποιος χρήστης χρησιμοποιεί simulation ή χτίζει κάποιο πληροφοριακό σύστημα το οποίο οδηγείται από τέτοια μοντέλα, τότε το play-out είναι αυτό ακριβώς που κάνει.

Στην ακριβώς αντίστροφη περίπτωση (**Play-In**), ξεκινάμε από τα **event data** προσπαθώντας να κατασκευάσουμε το αντίστοιχο **process model**. Φυσικά υπάρχουν πολλοί αλγόριθμοι που μπορούν να επιτύγχουν την δημιουργία ενός **process model** από **event data**. Ένας από αυτούς είναι και ο **Alpha Algorithm**, η μελέτη του οποίου είναι και αντικείμενο της παρούσας εργασίας. Η βασική ιδέα είναι ότι κοιτώντας έναν αριθμό από παραδείγματα συμπεριφοράς, δηλαδή **events**, αυτόματα παράγεται ένα μοντέλο από αυτά. Αυτό φαίνεται στην παρακάτων εικόνα.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img4.png  **Εικόνα 4:**  Play-in. Από έναν αριθμό από events κατασκευάζεται ένα BPMN model. |

Στην εικόνα 4 φαίνεται η διαδικασία του **Play-In**. Αν παρατηρήσουμε προσεκτικά τα **events** ή **traces** που φαίνονται στο πάνω μέρος της εικόνας, βλέπουμε ότι όλα τα **events** ξεκινούν με **a**. Άρα συμπεραίνουμε ότι και το εξαγώμενο μοντέλο θα ξεκινάει και αυτό με **a**. Όλα τα **events** καταλήγουν σε **g** ή **h**. Άρα και στο εξαγώμενο μοντέλο στο τέλος υπάρχει μια επιλογή για να γίνει αποδεκτή ή όχι ένα activity **g** ή **h**. Φυσικά με ένα πιο πολύπλοκο σύνολο **events** ή **traces,** τόσο πιο πολύπλοκο θα γίνει και το παραγώμενο μοντέλο. Το σημαντικότερο που πρέπει να γίνει κατανοητό σε αυτό το σημείο είναι ότι δεν υπάρχει κάποιο αρχικό μοντέλο, αλλά παράγεται αυτόματα από την επεξεργασία των αρχικών **events**.

Το τρίτο είδος σχέσεων ανάμεσα σε **process models** και **event data** λέγεται **Replay**. Είναι ένα πολύ βασικό χαρακτηριστικό του **process mining** επειδή επιτρέπει να ξανά επαναληφθεί η πραγματικότητα πάνω σε ένα μοντέλο. Δεν έχει σημασία πως παρήχθησαν τα δεδομένα είτε με το χέρι είτε μέσω κάποιας διαδικασίας που ανακαλύπτει μοντέλα (**process discovery**), το σημαντικό είναι ότι μπορεί να ξανα γίνει επανάληψη της πραγματικότητας πάνω στο μοντέλο. Κατά την διάρκεια του **Replay** ενός event μπορεί να ταιριάζει το event που δοκιμάζουμε στο μοντέλο αλλά υπάρχει και η πιθανότητα να μην ταιριάζει. Αυτή η τεχνική ονομάζεται συμμόρφωση (**conformance checking**).

Το **Replay** δεν έχει να κάνει όμως μόνο με το conformance checking αλλά και με την ανάλυση της απόδοσης (**performance analysis**) του μοντέλου. Μπορούμε πολύ εύκολα να ξανά τρέξουμε διάφορα event που έχουν καταγραφεί και έχουν πάνω τους χρονική πληροφορία (**timestamp**). Έτσι εκτελώντας τα event πάνω στο μοντέλο στο τέλος γνωρίζουμε πόσος χρόνος δαπανήθηκε σε όλα τα διαφορετικά σημεία της διαδικασίας. Έτσι μπορούμε να δούμε πόσο χρόνο πήραν τα διάφορα **activities** αλλά και τις καθυστερήσεις μεταξύ των **activities**. Αυτές οι μετρήσεις μπορούν να γίνουν για δεκάδες χιλιάδες περιπτώσεις και έτσι να παραχθεί μια πιθανοτική κατανομή για αυτές.

Συνοψίζοντας τα παραπάνω, υποθέτουμε ότι έχουμε ένα σύστημα (**software system**) το οποίο με κάποιο τρόπο καταγράφει γεγονότα (**events**) τα οποία συμβαίνουν. Τέτοια παραδείγματα είναι οι κάτοικοι μιας πόλης που κάνουν ένσταση στον τρόπο αξιολόγησης των σπιτιών τους θεωρώντας ότι οι φόροι που καταβάλουν είναι υψηλοί, οι ασθενείς σε ένα νοσοκομείο για τον τρόπο που εξυπηρετούνται, οι πελάτες σε ένα εστιατόριο για την ποιότητα των γευμάτων κτλπ. Με αυτό τον τρόπο συγκεκτρώνονται τα γεγονότα (**events**). Από αυτά τα γεγονότα μπορεί να γίνει η ανακάλυψη ενός **process model** και έπειτα να γίνει το **conformance checking** πάνω στο παραγόμενο μοντέλο. Συγκρίνοντας τα **events** με το **process model** μπορούμε να εμπλουτίσουμε το μοντέλο με πληροφορία σχετική με τις αποκλίσεις (**deviations**) και την απόδοση (**performance**).

Οπότε σύμφωνα με τα παραπάνω, το **Play-out** είναι η κλασική χρήση των μοντέλων διαδικασιών (**process model**), τα οποία δεν περιλαμβάνουν κάποια **event data**. Το **Play-in** αναφέρεται στην ανακάλυψη (**discovery**) ενός μοντέλου από events. Μπορείς αυτόματα να μάθεις ένα **process model** χωρίς καμία μοντελοποίηση από απλά δεδομένα (**raw event data**) χρησιμοποιώντας κάποιον αλγόριθμο όπως ο **Alpha Algorithm**. Και φυσικά υπάρχει η δυνατότητα για το χρήστη να ξανά κάνει **Replay** τα δεδομένα για να συγκρίνει ένα **process model** με ένα **event log**, ώστε ναελέγξεικαινα εκτελέσει **conformance checking**.

Αυτή η διαδικασία φαίνεται στην παρακάτω εικόνα

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img5.png  **Εικόνα 5:**  Σύνοψη των τριών ειδών **process mining** |

Συνοψίζοντας λοιπόν έχουμε τρεις κατηγορίες τεχνικών εξόρυξης διαδικασίων. Η ταξινόμηση αυτή βασίζεται στο κατά πόσο υπάρχει ένα προγενέστερο μοντέλο και, αν ναι, πώς χρησιμοποιείται.

* ***Ανακάλυψη (discovery)***: Δεν υπάρχει ένα προγενέστερο πρότυπο, δηλαδή, ένα έτοιμο **process model**. Για παράδειγμα, χρησιμοποιώντας τον **Alpha Algorithm**, ένα μοντέλο διαδικασίας (**process model**) μπορεί να ανακαλυφθεί βασιζόμενο σε χαμηλού επιπέδου συμβάντα (**events**). Υπάρχουν πολλές τεχνικές για να κατασκευάσουν αυτόματα μοντέλα διαδικασιών (για παράδειγμα Petri net ή BPMN), με βάση ορισμένες καταγραφές συμβάντων (**events**). Πρόσφατα, η ερευνητική διαδικασία εξόρυξης άρχισε επίσης να στοχεύει σε άλλες προοπτικές (π.χ. τα δεδομένα, τους πόρους, το χρόνο, κλπ.).
* ***Συμμόρφωση (conformance checking)***: Υπάρχει ένα προγενέστερο μοντέλο (**process model**). Στόχος είναι να βρεθεί κατά πόσο διαφέρει η πραγματικότητα από το υπάρχων **process model**. Αυτό το μοντέλο συσχετίζεται με το αρχείο καταγραφής συμβάντων (**events**) και οι διαφορές μεταξύ των **events** και του μοντέλου αναλύονται. Για παράδειγμα, μπορεί να υπάρχει ένα μοντέλο της διαδικασίας που αναφέρει ότι για εντολές αγοράς άνω του 1 εκατομμυρίου ευρώ, απαιτούνται δύο έλεγχοι για να ολοκληρωθεί με επιτυχία η αγορά. Συμμόρφωση μπορεί να χρησιμοποιηθεί για την ανίχνευση αποκλίσεων ώστε να εμπλουτιστεί το μοντέλο.
* ***Επέκταση (extension)***: Υπάρχει ένα μοντέλο εκ των προτέρων. Το μοντέλο αυτό επεκτείνεται με στόχο όχι να ελέγξει τη συμμόρφωση, αλλά για να εμπλουτίσει το μοντέλο.

# 1.4 Πως σχετίζεται το process mining με το data mining

Η εξόρυξη διαδικασιών (**process mining**) είναι ο συνδετικός κρίκος ανάμεσα ανάμεσα στην ανάλυση βασιζόμενη σε μοντέλα (**model based analysis**) και στην ανάλυση που επικεντρώνεται στα δεδομένα (**data oriented analysis**) όπως η εξόρυξη δεδομένων (**data mining**). Με την εξόρυξη διαδικασιών μπορούμε να απαντήσουμε ερωτήματα σχετικά με την απόδοση (**performance**) των διαδικασιών αλλά και ερωτήματα τα οποία σχετίζονται με την συμμόρφωση (**compliance**) πάνω σε κάποια μοντέλο. Άρα θα μπορούσαμε να πούμε ότι η εξόρυξη διαδικασιών (**process mining**) είναι η «κόλλα» ανάμεσα στα δεδομένα και στις διαδικασίες, ανάμεσα στους εργαζομένους του ΙΤ αλλά και του business αλλά και ανάμεσα στην απόδοση (**performance**) και την συμμόρφωση (**compliance**). Και φυσικά το **process mining** μπορεί ναγίνει και στο runtime αλλά και στο design time. Είναι ο συνδετικός κρίκος ανάμεσα σε πολλά πράγματα και μπορεί να φανεί απίστευτα πολύτιμο.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img6.png  **Εικόνα 6:**  Σύνδεση **business process management (BPM)** με τα κλασικά data analytics όπως το **data mining.** |

Η παραπάνω εικόνα απεικονίζει πως το **process mining** είναι ο συνδετικός κρίκος ανάμεσα στο **business process management (BPM)** και στις κλασικές κατηγορίες ανάλυσης δεδομένων όπως το **data mining**. Στην εικόνα αυτή φαίνεται ξεκάθαρα και το ότι το **process mining** αποτελείται από άλλα είδη εξόρυξης όπως η ανακάλυψη διαδικασιών (**process discovery**) αλλά και ο έλεγχος συμμόρφωσης (**compliance checking**). Ακόμα στην εικόνα εμφανίζεται και η ανάλυση προβλέψεων (**predictive analysis**) που σχετίζεται με την πρόβλεψη γεγονότων στο μέλλον.

Κάποιος μπορεί να σκεφτεί το **process mining** σαν μια διαδικασία εύρεσης επιθυμητών γραμμών. Αυτές οι γραμμές δείχνουν τι συνήθως κάνουν οι άνθρωποι. Παράδειγμα ένα μονοπάτι που το ακολουθούν οι άνρθωποι με κάποιες ταμπέλες που δείχνουν που δεν πρέπει να πατήσουν. Άρα αυτές οι γραμμές μπορούν να παρομοιαστούν σαν **event data** καιοι ταμπέλες σαν **process model**. Αυτός ο παραλληλισμός δείχνει τι προσπαθεί να κάνει το **process mining** δηλαδή να ανακαλύψει επιθυμητές γραμμές. Φυσικά πολλές φορές η διαδικασία του **process mining** μπορεί να ανακαλύψει μη-αναμενόμενα γεγονότα που να δείχνουν ότι πολλές φορές οι άνθρωποι δεν ακολουθούν το **process model**, δηλαδή τις επιθυμητές γραμμές.

Αρχικά, ο όρος εξόρυξη δεδομένων (**data mining**) είχε αρκετά αρνητική έννοια καθώς παρέπεμπε σε όρους όπως data snooping, fishing όπου κακόβουλοι χρήστες προσπαθούν να υποκλέψουν ευαίσθητα δεδομένα χρήστων όπως username, password, αιρθμούς πιστωτικών καρτών κτλπ. Αλλά πλέον λόγω της τεράστιας αύξησης των δεδομένων αυτό έχει αλλάξει καθώς όλο και περισσότεροι οργανισμοί ενδιαφέρονται για την οργάνωση και την ανάλυση των δεδομένων τους.

Το κοινό στοιχείο ανάμεσα στην εξόρυξη δομένων (**data mining**) και στην εξόρυξη διαδικασιών (**process mining**) είναι ότι η υλοποίηση τους ξεκινάει από τα δεδομένα (**data**), αλλά κατά τα άλλα υπάρχουν πολλές διαφορές ανάμεσα τους. Για τις διαδικασίες της εξόρυξης δεδομένων (**data mining**), οι γραμμές και οι στήλες στους πίνακες με τα δεδομένα μπορούν να σημαίνουν οτιδήποτε. Αντίθετα, στις τεχνικές της εξόρυξης διαδικασιών (**process mining**), υποθέτουμε ότι τα δεδομένα έχουν συγκεκριμένη μορφή και αναφέρονται σε **activities** σε συγκεκριμένες χρονικές στιγμές. Επιπλέον τα **events** είναι ταξινομημένα και για αυτό το λόγο ενδιαφερόμαστε σε **end-to-end** διαδικασίες.

Η βασική διαφορά τους είναι ότι η εξόρυξη δομένων (**data mining**) έχει σαν κέντρο τα δεδομένα και όχι τις διαδικασίες. Έτσι γίνεται κατανοητό ότι η ανακάλυψη διαδιακασιών (**process discovery**) , η συμμόρφωση σε ένα μοντέλο (**conformance checking**) και η ανάλυση για την εύρεση κάποιων σημείων όπου καθυστερεί η ομαλή λειτουργεία ενός οργανισμού (**bottlenecks discovery**) δεν αποτελεί τμήμα της εξόρυξης δεδομένων (**data mining**) αλλά τμήμα της εξόρυξης διαδικασιών (**process mining**). Και φυσικά τίποτα από αυτά δεν μπορεί να επιτευχθεί χρησιμοποιώντας την εξόρυξη δομένων (**data mining**).

H εξόρυξη διαδικασιών (**process mining**) συνδιάζει μοντέλα διαδικασιών (**process models**) και δεδομένα γεγονότων (**event data**) με πολλούς διαφορετικούς καινοτόμους τρόπους. Σαν αποτέλεσμα είναι ότι κάποιος χρησιμοποιώντας την εξόρυξη διαδικασιών (**process mining**) μπορεί να ανακαλύψει τι κάνουν στην πραγματικότητα οι άνθρωποι και οι οργανισμοί. Για παράδειγμα, μοντέλα διαδικασιών μπορούν να ανακαλυφθούν αυτόματα από **event data** (**process model discovery**). Η συμμόρφωση δεδομένων (**compliance**) μπορεί να ελεγχθεί συγκρίνοντας μοντέλα με **event data**. Διάφορα προβλήματα σε διαδικασίες μπορούν να ανακαλυφθούν ξανά εφαρμόζοντας διάφορα **events** πάνω σε μοντέλα που έχουν ήδη βρεθεί. Έτσι η εξόρυξη διαδικασιών (**process mining**) μπορεί να χρησιμοποιηθεί για την ανακάλυψη και κατανόηση αποκλίσεων, ρίσκων αλλά και προβλήματων στην καθημερινή λειτουργία μιας εταιρείας ή ενός οργανισμού.

Τα **end-to-end** μοντέλα διαδικασιών είναι πάρα πολύ σημαντικά και για την ανακάλυψη τους είναι απαραίτητο να εξεταστούν διαδικασίες που εκτελούνται παράλληλα (**concurrency**). Ακόμα η εξόρυξη διαδικασιών (**process mining**) προϋποθέτει τα δεδομένα που θα αναλυθούν να έχουν μια συγκεκριμένη δομή. Τα δεδομένα αυτά ονομάζονται γεγονότα (**events**) όπως είδαμε σε προηγούμενα κεφάλαια και πρέπει να φέρουν οπωσδήποτε ένα **timestamp,** ένα **eventName** αλλά και κάποια **ids** που να αναφέρονται σε διάφορες περιπτώσεις (**cases**). Αυτή είναι μια σημαντική διαφορά σε σχέση με την μορφή που έχουν τα δεδομένα τα οποία θα αναλυθούν από κάποιο αλγόριθμο εξόρυξης δεδομένων (**data mining**).

Συνήθως όμως σε πολύ σύνθετα προβλήματα η εξόρυξη διαδικασιών (**process mining**) και η εξόρυξη δομένων (**data mining**) συνδυάζονται καθώς αποτελούν συμπληρωματικές προσεγγίσεις που η μια δυναμώνει την άλλη.

# 2. Μοντέλα και Ανακάλυψη Διαδικασιών

# 2.1 Εισαγωγή στα Μοντέλα διαδικασιών

Σκοπός αυτού του κεφαλαίου είναι να γίνει η παρουσίαση του αλγορίθμου **Alpha Algorithm** που υλοποιήθηκε στα πλαίσια αυτής της εργασίας. Ο συγκεκριμένος αλγόριθμος επιτρέπει σε μόλις 8 βήματα την μετατροπή δεδομένων γεγονότων (**event data**) σε μοντέλα διαδικασιών τα οποία έχουν την δυνατότητα να εκφράσουν μια μεγάλη γκάμα συμπεριφορών, όπως ακολουθίες (**sequences**), παραλληλισμό (**concurrency**), επιλογές (**choices**) και επαναλήψεις (**loops**). Φυσικά αυτός ο αλγόριθμος έχει κάποιους περιορισμούς στο τι μπορεί να κάνει, αλλά απεικονίζει τους βασικούς μηχανισμούς που χρησιμοποιούνται από πιο εξελιγμένους αλγορίθμους ανακάλυψης διαδικασιών (**process discovery**) , όπως ο **Inductive miner**, ο **Heuristic miner** και ο **Fuzzy miner**. Για να γίνει όμως κατανοητός ο τρόπος λειτουργίας του **Alpha Algorithm** πρέπει πρώτα να γίνουν κατανοητά τα βασικά της μοντελοποίησης διαδικασιών (**process modeling**) και των **Petri nets**.

Υπάρχει μεγάλη ποικιλία από μοντέλα διαδικασιών (**process models**) τόσο στην βιομηχανία όσο και στην πανεπιστημιακή έρευνα. Υπάρχουν κάποια μοντέλα χαμηλότερου επιπέδου (**low-level notations**) τα οποία δεν μπορούν να απεικονήσουν παραλληλισμό (**concurrency**) αλλά και κάποια συγκεκριμένα δομικά στοιχεία (**routing constructs**) αλλά και κάποια άλλα υψηλότερου επιπέδου μοντέλα (**higher level notations**) τα οποία έχουν καλύτερες δυνατότητες απεικόνησης διαδικασιών, όπως τα BPMN, UML activity diagrams, EPCs, Petri Nets κτλπ. Επιπλέον, τα διάφορα μοντέλα (**notations**) συνήθως βρίσκουν εφαρμογή σε διαφορετικά πεδία, παράδειγμα οι **business analysts** χρησιμοποιούν **BPMN**, οι software engineers χρησιμοποιούν **UML**, οι SAP consultants χρησιμοποιούν **EPCs** κτλπ. Υπάρχει πληθώρα μοντέλων (**notations**) που χρησιμοποιούνται για να εκφράσουν το ίδιο ακριβώς αντικείμενο πράγμα το οποία δημιουργεί σύγχυση.

Αυτή η πληθώρα στα μοντέλα διαδικασιών οφείλεται στο ότι ο καθένας μπορεί να υλοποιήσει ένα σύνολο από νέα σύμβολα και να δημιουργήσει μια νέα γλώσσα για εμπορικούς κυρίως λόγους. Αλλά όμως αν κοιτάξουμε στα θεμέλια της δυναμικής συμπεριφοράς που θέλουμε να απεικονήσουμε, υπάρχουν κάποιες βασικές κοινές έννοιες. Αυτό μπορεί να γίνει κατανοητό από την ανακάλυψη μοντέλων διαδικασιών (**process models**) από γεγονότα (**event data**) όπως με τον **Alpha Algorithm**. Τα γεγονότα είναι μερικώς ταξινομημένα (**ordered**) και αναφέρονται σε στιγμιότυπα διαδικασιών (**case** ή **trace**) και δραστηριοτήτων (**activities**). Βασιζόμενοι στα παραπάνω (**cases** και **traces**), κάποιος αλγόριθμος μπορεί να κατασκευάσει ένα **process model** από ακολουθίες (**sequence**), επιλογές (**choice**), παραλληλισμούς (**parallel**) και επαναλήψεις (**iteration**) σε όποιο μοντέλο διαδικασίας (**process model**) επιθυμεί.

Πολλοί αλγόριθμοι εξόρυξης διαδικασιών (**process mining**) χρησιμοποιούν Petri nets για την απεικόνηση του αποτελέσματος του αλγορίθμου αλλά και για την συμμόρφωση πάνω σε ένα μοντέλο (**conformance checking**). Αλλά αυτό δεν σημαίνει ότι οι end-to-end χρήστες θα χρεαστεί να δουν απαραίτητα ένα **Petri net**. Εργαλεία όπως το ProM μπορούν να μετατρέψουν ένα **Petri net** σε άλλους τύπους μοντέλων όπως τα BPMN. Στο επόμενο κεφάλαιο παρουσιάζεται με λεπτομέρειες το μοντέλο απεικόνησης διαδικασιών **Petri net**, το οποίο χρησιμοποιήθηκε στην συγκεκριμένη εργασία.

# 2.2 Petri Nets

Τα **Petri Net** αποτελούν μια τεχνική περιγραφής και ανάλυσης κατανεμημένων (**distributed**) συστημάτων και διαδικασιών. Έχουν ένα εκφραστικό γραφικό περιβάλλον και αποτελούν επέκταση της μαθηματικής θεωρείας αυτομάτων (**Automata theory**). Ένα **Petri Net** είναι ένα κατευθυνόμενο διμερές γράφημα (**directed** **bipartite graph**), το οποίο αποτελείται από κύκλους (**places**) και τετράγωνα (**squares**). Τα τετράγωνα απεικονίζουν τις μεταβάσεις από κάποια δραστηριότητα (**activity**) σε κάποια άλλη.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img7.png  **Εικόνα 7:**  Αναπαράσταση ενός Petri Net |

Τα **places** μπορούν να είναι είσοδοι ή έξοδοι σε κάποιες μεταβάσεις (**transitions**) του συστήματος. Τα **places** απεικονίζουν την κατάσταση (**state**) του συστήματος. Οι μεταβάσεις (**transitions**) απεικονίζουν τις αλλαγές της κατάστασης του συστήματος.

Ένα **Petri Net** αποτελείται από κύκλους (places), τετράγωνα (transitions) και πλευρές (arcs) και σαν **Petri Net** μπορεί να οριστεί μια τριάδα **(P, T, F)** όπου

* **P** είναι ένα πεπερασμένο σύνολο από **places**
* **T** είναι ένα πεπερασμένο σύνολο από **transitions.** Το σύνολο αυτό είναι και το σύνολο των **events** σε ένα μοντέλο.
* **F** είναι το σύνολο των ακμών σε ένα **Petri Net**, δηλαδή ένα υποσύνολο ακμών. Παράδειγμα για την εικόνα 7 το πεπερασμένο σύνολο ακμών F είναι το F= { ( , ) , ( , )}

Κάθε **place** μπορεί να περιέχει ένα ή περισσότερα **tokens**. Ένα **token** είναι μια μονάδα εργασίας που πρέπει να εκτελεστεί.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img8.png  **Εικόνα 8:**  Αναπαράσταση των **tokens**. Στο πρώτο σχήμα υπάρχει ένα **token** ενώ στο δεύτερο δύο. |

**Marking** είναι η κατάσταση ενός Petri Net που αποτυπώνει την κατανομή των **tokens** σε όλα τα **places** του Petri Net. Μια μετάβαση (**transition**) αλλάζει την κατάσταση (**state**) ενός Petri Net με την πυροδότηση (**firing**).

Aς δούμε τα παρακάτω παραδείγματα.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img9.png  **Εικόνα 9:**  Στην παραπάνω εικόνα φαίνεται ότι η πυροδότηση (**firing**) είναι εφικτή, καθώς υπάρχουν δύο tokens, ένα σε κάθε είσοδο της μετάβασης . Το αποτέλεσμα φαίνεται στην παρακάτω εικόνα όπου οι είσοδοι της μετάβασης δεν έχουν πλέον token, και η έξοδος της μετάβασης έχει ακριβώς ένα token.  C:\Users\Vlasis\Dropbox\osyl\report\images\img10.png  **Εικόνα 10:**  Δηλαδή, σύμφωνα με τα παραπάνω για να είναι δυνατή η πυροδότηση μιας μετάβασης σε ένα Petri Net είναι απαραίτητη η ύπαρξη ενός τουλάχιστον token σε κάθε είσοδο της μετάβασης. Ακόμα και στις εξόδους του Petri Net είναι δυνατό να δημιουργηθούν τόσα tokens όσα είναι οι έξοδοι της μετάβασης.  C:\Users\Vlasis\Dropbox\osyl\report\images\img11.png  **Εικόνα 11:**  Στην παρακάνω εικόνα φαίνεται μια κατάσταση σε ένα Petri Net όπου δεν μπορεί να γίνει η πυροδότηση μιας μετάβασης καθώς λείπει ένα token το οποίο έπρεπε να είναι διαθέσιμο. |

# 2.3 Alpha Algorithm – Δομικά στοιχεία (patterns)

Ο **Alpha Algorithm** είναι ένας αλγόριθμος ανακάλυψης διαδικασιών (**process discovery**) από δεδομένα (**event data**). Σύμφωνα με τα όσα είδαμε στο κεφάλαιο 1.3, είναι κατανοητό ότι η διαδικασία του **process discovery** ανήκει στην κατηγορία **Play-In**. Έτσι από ένα σύνολο από γεγονότα (**event log**), ο **Alpha Algorithm** θα ανακαλύψει αυτόματα ένα μοντέλο διαδικασίας (**process model**) και συγκεκριμένα ένα **Petri Net**. Ο **Alpha Algorithm** ήταν ο πρώτος αλγόριθμος ο οποίος ήταν ικανός να ανακαλύψει παράλληλα γεγονότα (**concurrency**), να ανακαλύψει επαναλήψεις (**loops**) αλλά και επιλογές στην διαδικασία (**choices**) ενώ ταυτόχρονα μπορεί και εγγυάται κάποιες συγκεκριμένες ιδιότητες.

Αλλά τι είδους είσοδο χρησιμοποιεί ο **Alpha Algorithm**;

Συγκεκριμένα σε σύνολο δεδομένων (**data set**), o **Alpha Algorithm** αγνοεί τα resources και άλλα δεδομένα. Ακόμα αγνοεί και τα **timestamps** των events που συμβαίνουν, όπως και το **caseID** των **traces** που επεξεργάζεται. Το μόνο που ενδιαφέρει τον Alpha Algorithm για να παράγει αποτελέσματα είναι η ταξινόμηση (**ordering**) των γεγονότων (**events**). Έτσι ένα πρώτο βήμα για τον **Alpha Algorithm** είναι η μετατροπή του **event log** σε ένα σύνολο από **traces**, όπου ένα **trace** είναι μια ακολουθία από ονόματα δραστηριοτήτων (**activity names**).

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img12.png  **Εικόνα 12:**  Στην εικόνα αυτή φαίνεται ένα σύνολο από **events** που βασίζεται σε δεδομένα από την πραγματική ζωή. Ακόμα γίνεται κατανοητό ότι ο Alpha Algorithm εδιαφέρεται μόνο για τα activities και όχι για άλλες πληροφορίες. Ακόμα βλέπουμε πως γίνεται η κατασκευή των **traces** από το order number. |

Στην συνέχεια φαίνεται παρακάτω ένα πιο απλό παράδειγμα.

= [ ]

Εδώ βλέπουμε το **log** , το οποίο αποτελείται από 6 **traces**, τις οποίες μπορούμε να τις θεωρήσουμε σαν 6 διαφορετικές περιπτώσεις (**cases**). Αυτά τα **cases** μοντελοποιούνται σαν μια ακολουθία από **activity names**. Έτσι η ακολουθία a,b,c,d εκτελέστηκε 3 φορές και έτσι υπάρχουν 3 **traces** αυτού του τύπου. Έτσι ένα **event log** είναι ένα πολύ-σύνολο (**multiset**) από **traces** επειδή το ίδιο **trace** μπορεί να εμφανιστεί πολλαπλές φορές, και ένα **trace** είναι μια ακολουθία από **activity names** τα οποία μπορούν να απομονωθούν σε σχέση με όλες τις υπόλοιπες ιδιότητες.

Σκοπός του **Alpha Algorithm** είναι αν τροφοδοτήσουμε κάποιο **event log** όπως το , πρέπει αυτόματα να μάθουμε ένα μοντέλο διαδικασίας (**process model**) το οποίο θα απεικονίζει ακριβώς την συμπεριφορά του **event log** . Στην παρακάτων εικόνα φαίνεται το Petri Net που εξάγει ο **Alpha Algorithm** για το **event log** . Φυσικά το αποτέλεσμα θα μπορούσε να ήταν και οποιαδήποτε άλλη μορφή αναπαράστασης μοντέλων (**notation**), όπως ένα BPMN μοντέλο, ένα EPC μοντέλο ή ένα UML activity diagram, καθώς η κύρια ευθύνη του αλγορίθμου είναι να καταλάβει την συμπεριφορά μέσα από το **event log.**

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img13.png  **Εικόνα 13:**  Το γράφημα Petri Net που παράγει ο Alpha Algorithm για το **event log** |

Το σημείο εκκίνησης του **Alpha Algorithm** είναι οι σχέσεις ταξινόμησης (**ordering relations**). Ο **Alpha Algorithm** δεν έχει κανένα ενδιαφέρον πάνω στις συχνότητες που εμφανίζονται τα **traces** μέσα στο **event log**, όπως και για οποιαδήποτε άλλα χαρακτηριστικά που έχουν αποθηκευτεί για κάθε **event**. Αυτό λοιπόν που ενδιαφέρει τον **Alpha Algorithm** είναι να βρει μέσω των **events** ποια **activities** ακολουθούν ποια. Για παράδειγμα για την ακολουθία a,b,c,d, ο αλγόριθμος θα δει ότι το a ακολουθείται από το b, το b από το c και το c από το d. Αυτή η σχέση ονομάζεται απευθείας διαδοχή (**direct succession**) και ισχύει όταν κάποιο **case** x ακολουθείται απευθείας από το y και συμβολίζεται σαν **x>y**.

Για παράδειγμα στο **event log** βρίσκουμε τις παρακάτω σχέσεις που ανήκουν στην κατηγορία **direct succession.**

(a>b , a>c , a>e , b>c , b>d , c>b , c>d , e>d)

Η δεύτερη σχέση που εξετάζει ο **Alpha Algorithm** είναι η **causality** και συμβολίζεται με ένα βέλος (**x->y**) και σημαίνει ότι το x ακολουθείται από το y, αλλά το y δεν ακολουθείται **ποτέ** από το x. Στο **event log** βρίσκουμε τις παρακάτω σχέσεις που ανήκουν στην κατηγορία **causality.**

(a->b , a->c , a->e , b->d , c->d , e->d)

Η τρίτη σχέση ισχύει όταν η **direct succession** σχέση είναι έγκυρη και για τις δύο κατευθύνσεις, τότε λέμε ότι η σχέση είναι παράλληλη (**parallel**). Άρα όταν μερικές φορές το x ακολουθείται από το y και μερικές φορές το y ακολουθείται από το x τότε το x είναι παράλληλο με το y (**x||y**). Στο **event log** βρίσκουμε τις παρακάτω σχέσεις που ανήκουν στην κατηγορία **parallel.**

(b||c , c||b)

Η τέταρτη και τελευταία σχέση ισχύει όταν το x δεν ακολουθείται ποτέ απευθείας από το y και το y δεν ακολουθείται ποτέ απευθείας από το x (**x#y**). Τότε λέμε ότι αυτό είναι μια **choice (exclusiveness) relation**. Στο **event log** βρίσκουμε τις παρακάτω σχέσεις που ανήκουν στην κατηγορία **choice.**

(b#e , e#b , c#e , a#d …)

Άρα συνοψίζοντας έχουμε τις παρακάτω σχέσεις

* **Direct succession (x>y)** Υπάρχουν sub-traces …xy….
* **Causality (x->y)** Υπάρχουνsub-traces …xy…. Αλλά όχι …yx….
* **Parallel (x||y)** Υπάρχουνsub-traces …xy…. και …yx….
* **Choice- exclusiveness** (**x#y**) Δεν υπάρχουνsub-traces …xy…. ούτε και …yx….

Αυτές οι παραπάνω σχέσεις χρησιμοποιούνται για να εξαγωγή διάφορων μοτίβων (**patterns**) στην διαδιακασία (**process**). Στη παρακάτω εικόνα (**Εικόνα 14**) φαίνεται η σχέση **causality**. Δηλαδή αυτό το **pattern** περιμένουμε να δούμε σε ένα Petri Net όταν μερικές φορές το a μερικές φορές ακολουθείται από το b, αλλά ποτέ δεν συμβαίνει το αντίθετο.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img14.png  **Εικόνα 14:**  **Causality pattern (a->b)** |

Στην παρακάτω (**Εικόνα 15**) φαίνεται το pattern **XOR-split** σε ένα Petri Net που προκύπτει

* όταν το a μερικές φορές ακολουθείται από το b, αλλά ποτέ όμως αντίστροφα **(a->b)**
* όταν το a μερικές φορές ακολουθείται από το c, αλλά ποτέ όμως αντίστροφα **(a->c)**
* όταν το b και το c δεν ακολουθεί ποτέ το ένα το άλλο (**b#c**)

Αυτό το pattern σημαίνει ότι μετά το a, υπάρχει μια επιλογή (**choice**) μεταξύ των b και c.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img15.png  **Εικόνα 15:**  **XOR-split pattern (a->b , a->c , b#c) (choice** ανάμεσα στο **b** και **c)** |

Το αντίστοιχο pattern του **XOR-split** είναι το **XOR-join,** όπως φαίνεται στην εικόνα 16. Ισχύει όταν

* το b έχει **causal relation** με το d (**b->d**)
* το c έχει **causal relation** με το d (**c->d**)
* το b με το c ποτέ δεν ακολουθεί το ένα το άλλο (**b#c**)

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img16.png  **Εικόνα 16:**  **XOR-join pattern (b->d , c->d , b#c)** |

Ως προς την παραλληλία (**concurrency**) σε ένα Petri Net μοντέλο υπάρχουν δύο **patterns**, το **AND-split** και το **AND-join**.

Στην παρακάτω (**Εικόνα 17**) φαίνεται το pattern **AND -split** σε ένα Petri Net που προκύπτει

* όταν το a μερικές φορές ακολουθείται από το b, αλλά ποτέ όμως αντίστροφα **(a->b)**
* όταν το a μερικές φορές ακολουθείται από το c, αλλά ποτέ όμως αντίστροφα **(a->c)**
* όταν το b και το c δεν ακολουθεί ποτέ το ένα το άλλο (**b||c**)

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img17.png  **Εικόνα 17:**  **AND-split pattern (a->b ,a->c , b||c)** |

Το αντίστοιχο pattern του **AND -split** είναι το **AND -join,** όπως φαίνεται στην **εικόνα 18**. Ισχύει όταν

* το b έχει **causal relation** με το d (**b->d**)
* το c έχει **causal relation** με το d (**c->d**)
* το b με το c ποτέ δεν ακολουθεί το ένα το άλλο (**b||c**)

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img18.png  **Εικόνα 18:**  **AND-join pattern (b->d ,c->d , b||c)** |

Άρα βασιζόμενοι στα παραπάνω **patterns** είναι δυνατή η κατασκευή ενός Petri Net. Έτσι λοιπόν όλες οι σχέσεις ταξινόμησης (**ordering relations**) που εξάγονται από ένα event log, όπως το **event log** , αποτελούν ένα πίνακα footprint και αυτός είναι ακριβώς ο πίνακας στην Εικόνα 19 για το **event log** . Κάθε κελί (**cell**) σε αυτόν τον πίνακα έχει μια από τις τέσσερεις σχέσεις που παρασουσιάστηκαν νωρίτερα (**Direct succession, causality, parallel, choice**). Έτσι λοιπόν σύμφωνα με τον πίνακα είναι δυνατόν να έχουμε **causality** ως προς την μία κατεύθυνση αλλά και ως προς την άλλη.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img19.png  **Εικόνα 19:**  **Footprint graph** |

# 2.4 Alpha Algorithm – Βήματα

Ο **Alpha Algorithm** είναι ένας πολύ βασικός αλγόριθμος που περίεχει μόνο 8 βήματα αλλά μοιάζει πολύπλοκος όταν κάποιος τον μελετήσει για πρώτη φορά. Παρά την απλότητα του ο **Alpha Algorithm** μπορεί να ανακαλύψει ένα τεράστιο εύρος από διαφορετικές διαδικασίες. Διαδικασίες οι οποίες περιέχουν παραλληλισμό (concurrency), διαδικασίες οι οποίες περιέχουν επαναλήψεις (**loops**) και διαδικασίες οι οποίες περιέχουν επιλογές (**choices**). Έτσι λοιπόν, η εκτέλεση του **Alpha Algorithm** ξεκινάει από ένα σύνολο από **traces** με σκοπό την κατασκευή ενός Petri Net.

* **Πρώτο Βήμα**

Στο πρώτο βήμα ο **Alpha Algorithm** σκανάρει το **event log** για να δει ποιες **activities** ή **transitions** λαμβάνουν μέρος. Άρα ψάχνει για το ποια μοναδικά **activities** υπάρχουν στο **event log.**

* **Δεύτερο Βήμα**

Στο δεύτερο βήμα ο **Alpha Algorithm** βρίσκει ένα σύνολο από όλες τις αρχικές **activities** πουυπάρχουν στο **event log,** δηλαδή το πρώτο στοιχείο κάθε **trace** που ανήκει στο **event log**.

* **Τρίτο Βήμα**

Στο τρίτο βήμα ο **Alpha Algorithm** βρίσκει ένα σύνολο από όλες τις τελικές **activities** πουυπάρχουν στο **event log,** δηλαδή το τελευταίο στοιχείο κάθε **trace** που ανήκει στο **event log**.

* **Τέταρτο Βήμα**

Στο τέταρτο βήμα ο **Alpha Algorithm** κατασκευάζει το **footprint graph,** δηλαδή τον πίνακα που περιέχει όλες τις σχέσεις μεταξύ των διάφορων **activities** του **event log**. Η διαδικασία της ανακάλυψης μιας διαδιακασίας έχει να κάνει κυρίως με την ανακάλυψη **patterns** / **places** πάνω σε ένα παραγώμενο Petri Net στην περίπτωση του **Alpha Algorithm.** Για την εύρεση των places πρέπει να γίνει ο προσδιορισμός ενός συνόλου από μεταβάσεις (**transitions**) A και Β, όπου το Α είναι η μετάβαση εισόδου για ένα **place** και Β είναι η μετάβαση εξόδου για ένα **place**.

Μετά την κατασκευή του **footprint graph** πρέπει σύμφωνα με τις πληροφορίες που παρέχει το **footprint graph** να βρεθούν δύο σύνολα από activities Χ και Υ. Τα **activities** αυτά πρέπει να έχουν την ακόλουθη ιδιότητα. Αν επιλεγούν δύο **activities** από το σύνολο Χ δεν πρέπει ποτέ να ακολουθεί το ένα το άλλο, δηλαδή πρέπει να ισχύει η σχέση “**choice**”. Το ίδιο ακριβώς πρέπει να ισχύει και για το σύνολο Υ. Το δεύτερο προαπαιτούμενο είναι ότι αν επιλεγεί ένα **activity** από το Χ και ένα **activity** από το Υ πρέπει να έχουν μια **causal relation** μεταξύ τους. Αυτό φυσικά θα πρέπει να ισχύει για όλους τους συδιασμούς από ζεύγη για όλα τα **activities** από το Χ στο Υ. Δηλαδή πρέπει να βρεθούν ζεύγη (Χ,Υ) από σύνολα από **activities** τέτοια ώστε κάθε στοιχείο xX και κάθε στοιχείο yY να ανήκουν στη σχέση **causal relation** (x->y) και τα στοιχεία των Χ και Υ να είναι ανεξάρτητα μεταξύ τους ( # ) και ( # ). Αυτό φαίνεται ξεκάθαρα και στην **Εικόνα 20**.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img20.png  **Εικόνα 20:**  Εύρεση όλων των ζεύγων από σύνολα (Α,Β) για τα οποία ισχύει   * Κάθε Α πρέπει να συνδέεται με όλα τα Β για κάθε **place** p * a Α και Β πρέπει να ισχύει a->b * , Α : # και , B : #   Στο συγκεκριμένο παράδειγμα τα 2 παράπανω σύνολα προέκυψαν από τις σχέσεις  a#b , c#d , a->c , a->d , b->c , b->d και όλες οι μεταβάσεις (**transitions**) από το A τοποθετούν tokens στο **place** p και οι μεταβάσεις (**transitions**) από το Β καταναλώνουν τα tokens από το **place** p. |

* **Πέμπτο Βήμα**

Στο πέμπτο βήμα γίνεται διαγραφή από το σύνολο των **places** που βρέθηκαν στο βήμα 4 όλων των ζευγών (**pairs**) τα οποία δεν είναι **maximal**. Δηλαδή από το αρχικό σύνολο θα διατηρηθούν μόνο εκείνα που περιέχουν τον μέγιστο αιρθμό από στοιχεία που μπορούν να συνδεθούν μέσω ενός συγκεκριμένου (place) (**Εικόνα 21**).

* **Έκτο Βήμα**

Στο έκτο βήμα σχηματίζεται ένα σύνολο από **places** . Από τα maximal pairs που βρέθηκαν στο βήμα 5 τα οποία είναι **places** του τελικού Petri Netπροσθέτουμε ένα αρχικό place I και ένα τελικό place Ο. Αυτό το σύνολο αποτελεί και το σύνολο P της τριάδας **(P, T, F)** που περιγράφει το Petri Net που θα παραχθεί.

* **Έβδομο Βήμα**

Στο έβδομο βήμα καθορίζεται το **flow relation** του Petri Net που θα παραχθεί, δηλαδή παράγονται τα βέλη (arcs) που υπάρχουν στο Petri Net. Από τα προηγούμενα βήματα του αλγορίθμου έχουν βρεθεί οι μεταβάσεις (**transitions**) και τα **places** (στο πέμπτο βήμα). Έτσι στο έβδομο βήμα για κάθε **place** p(A,B) ενώνεται για κάθε στοιχείο a που ανήκει στο σύνολο του Α με το αντίστοιχο transition και κάθε στοιχείο b που ανήκει στο σύνολο του Β με το αντίστοιχο transition. Επιπλέον σχεδιάζεται ένα τόξο από την αρχική κατάσταση (place) προς κάθε αρχίκη μετάβαση (start transition) και ένα τόξο από κάθε τελική transition προς την τελική κατάσταση . Άρα σε αυτό το βήμα γίνονται οι συνδέσεις του Petri Net αλλά όπως φαίνεται ξεκάθαρα η όλη πολυπλοκότητα του αλγορίθμου βρίσκεται στα βήματα 4 και 5. Το σύνολο που θα παραχθεί από το βήμα 7 αποτελεί και το σύνολο **F** της τριάδας **(P, T, F)** που περιγράφει το Petri Net που θα παραχθεί.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img21.png  **Εικόνα 21:**  Απεικόνηση του βήματος 5. Εύρεση των maximal pairs. Αν το παραγώμενο από το βήμα 4 σύνολο , περιέχει τα 3 παραπάνω ζεύγη συνόλων από **activities** τότε το βήμα 5 θα κρατήσει μόνο το συνόλο 1 (αριστερά) γιατί περιέχει και τα άλλα 2 σύνολα (δεξιά) |

* **Όγδωο Βήμα**

Στο όγδωο και τελευταίο βήμα του Alpha Algorithm γίνεται η κατασκευή του τελικού Petri Net το οποίο αποτελείται από μια τριάδα συνόλων **(P, T, F).** Το σύνολο **P** αποτελεί τα **places / states** του Petri Net τα οποία έχουν παραχθεί από το βήμα 5 του Alpha Algorithm. Το σύνολο **Τ** αποτελεί τις μεταβιβάσεις **(transitions)** του Petri Net και είναι το σύνολο των μοναδικών activities names που υπάρχουν στο εξεταζόμενο **event log**. Το σύνολο **F** αποτελεί τα **τόξα / arcs** του Petri Net τα οποία έχουν παραχθεί από το βήμα 7 του **Alpha Algorithm**.

# 2.5 Alpha Algorithm – Παράδειγμα

Σε αυτό το κεφάλαιο θα παρουσιαστεί η εκτέλεση του **Alpha Algorithm** για το **event log** εκτελώντας όλα τα βήματα του αλγορίθμου ένα προς ένα

= [ ]

* **Πρώτο Βήμα**

Εύρεση όλων των μοναδικών **activities/transitions : Τ = {a,b,c,d,e}**

* **Δεύτερο Βήμα**

Αρχικές **activities** : = {a}

* **Τρίτο Βήμα**

Τελικές **activities** : = {e}

* **Τέταρτο Βήμα**

Εύρεση footprint graph και πιθανών καταστάσεων (places / states) του Petri Net ().

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **a** | **b** | **c** | **d** | **e** |
| **a** | # | -> | -> | # | -> |
| **b** | <- | # | || | -> | # |
| **c** | <- | || | # | -> | # |
| **d** | # | <- | <- | # | <- |
| **e** | <- | # | # | -> | # |

= {

( {a} , {b} ),

( {a} , {c} ),

( {a} , {e} ),

( {a} , {b, e} ),

( {a} , {c, e} ),

( {b} , {d} ),

( {c} , {d} ),

( {e} , {d} ),

( {b,e} , {d} ),

( {c,e} , {d} )

}

* **Πέμπτο Βήμα**

Εύρεση των **maximal pairs** από το σύνολο που βρέθηκε στο βήμα 4. Παρατηρούμε ότι στο σύνολο περιέχονται πολλά στοιχεία τα οποία δεν είναι maximal και ορισμένα από αυτά περιέχονται σε άλλα στοιχεία. Παράδειγμα τα ( {a} , {b} ), ( {a} , {c} ), ( {a} , {e} ), ( {b} , {d} ), ( {c} , {d} ), ( {e} , {d} ) τα οποία θα διαγραφούν από το πέμπτο βήμα του **Alpha Algorithm.** Έτσι το σύνολο των maximal pairs είναι το = { ( {a} , {b, e} ), ( {a} , {c, e} ), ( {b,e} , {d} ), ( {c,e} , {d} ) }

* **Έκτο Βήμα**

Στο βήμα αυτό προστίθεται στο σύνολο μια αρχική και μια τελική κατάσταση και έτσι σχηματίζεται το σύνολο P της τριάδας **(P, T, F)** που περιγράφει το Petri Net που θα παραχθεί.

* **Έβδομο Βήμα**

Σε αυτό το βήμα σχηματίζεται το σύνολο **F** της τριάδας **(P, T, F)** που περιγράφει το Petri Net που θα παραχθεί. Το σύνολο αυτό περιέχει όλα τα τόξα του Petri Net.

* **Όγδωο Βήμα**

Σε αυτό το βήμα σχηματίζεται το τελικό Petri Net που φαίνεται στην εικόνα 22.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img22.png  **Εικόνα 22:**  Παραγόμενο Petri Net για το event log = [ ] |

Στην **Εικόνα 22** βλέπουμε τις καταστάσεις (**states** / **places**) που παρήχθησαν στο πέμπτο βήμα του **Alpha Algorithm.** Συγκεκριμένα

* το p1 αντιστοιχεί στο ( {a} , {b, e} )
* το p2 αντιστοιχεί στο ( {a} , {c, e} )
* το p3 αντιστοιχεί στο ( {b,e} , {d} )
* το p4 αντιστοιχεί στο ( {c,e} , {d} )

Ακόμα βλέπουμε και την αρχική (**start**) και τελική (**end**) κατάσταση που προστέθηκαν στο έκτο βήμα του **Alpha Algorithm.**

Συνοψίζοντας όλα τα παραπάνω, ο **Alpha Algorithm** παρέχει μια βασική **process discovery** προσέγγιση. Έχει όμως και πολλούς περιορισμούς οι οποίοι θα αναλυθούν στο επόμενο κεφάλαιο. Αλλά παρά τους περιορισμούς, ο αλγόριθμος αυτός είναι πολύ χρήσιμος και παρουσιάζει τα βασικά συστατικά κάθε άλλου αλγόριθμου ανακάλυψης διαδικασιών (**process discovery**), όπως την ανακάλυψη επαναλήψεων (loops), παραλληλισμών (concurrency), επιλογών (choices) κτλπ.

# 2.6 Alpha Algorithm – Περιορισμοί

Στο προηγούμενο κεφάλαιο είδαμε ότι ο **Alpha Algorithm** αποτελείται από μόλις οκτώ βήματα και παρά την απλότητα του είναι αρκετά δυνατός ώστε να εξάγει διαδικασίες από μεγάλα **event logs**. Αλλά όμως δεν είναι αρκετά δυνατός ώστε να μπορέσει να ανταπεξέλθει σε όλες τις περιπτώσεις που η εφαρμογή του είναι χρήσιμη καθώς έχει κάποιους περιορισμούς και δεν μπορεί να παράγει τα αποτελέσματα που περιμένει ο χρήστης.

Ένα μειονέκτημα λοιπόν του **Alpha Algorithm** είναι ότι δεν μπορεί να προσδιορίσει τα **implicit places**. Τα **implicit places** δεν είναι απαραίτητα λάθος στο τελικό Petri Net που θα παραχθεί αλλά αν τα αφαιρέσουμε από το Petri Net, τότε η συμπεριφορά του μοντέλου δεν θα αλλάξει. Άρα ο επιθυμητός στόχος είναι να παραχθεί ένα Petri Net χωρίς αυτά τα 2 places καθώς δεν προσθέτουν καμία λειτουργικότητα και μόνο περιπλέκουν το μοντέλο.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img23.png  **Εικόνα 23:**  Τα δύο places p1 και p2 είναι implicit places. |

Ένα δεύτερο μειονέκτημα του **Alpha Algorithm** είναι ότι δεν μπορεί να αντιμετωπίσει επαναλήψεις (**loops**) συγκεκριμένου μήκους (μήκους ένα και μήκους δύο). Παράδειγμα το **event log** = [ ] το οποίο πάντα ξεκινάει με a και καταλήγει με c, αλλά στο ενδιάμεσο μπορεί να υπάρξει οποιοσδήποτε αριθμός από b. Αν εφαρμοστεί ο **Alpha Algorithm** πάνω σε αυτό το **event log** τότε θα βρεθούν οι παρακάτω σχέσεις

* (a>b , a>c , b>b , b>c)
* (a->b , a->c , b->c)
* (b||b)
* (a#a , c#c)

και θα βρεθεί το Petri Net της παρακάτω εικόνας 24 όπου το **transition b** δεν συνδέεται με κανένα state/place του Petri Net, γιατί αν εξεταστεί με προσοχή ο **Alpha Algorithm** θα δούμε ότι αν το b ακολουθεί τον εαυτό του και είναι σε σχέση παραλληλίας (**concurrency**) τότε δεν μπορεί να συνδεθεί με κανένα άλλο στοιχείο του παραγώμενου μοντέλου. Αυτό που πραγματικά έπρεπε να επιστρέψει ο αλγόριθμος είναι το Petri Net της εικόνας 25.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img24.png  **Εικόνα 24:**  Petri Net που παράγεται από τον **Alpha Algorithm** για **event log** . Σε αυτήν την περίπτωση ο αλγόριθμος δεν μπορεί να ανιχνεύσει loops μήκους ένα.  C:\Users\Vlasis\Dropbox\osyl\report\images\img25.png  **Εικόνα 25:**  Επιθυμητό αποτέλεσμα **process mining** αλγορίθμου για επαναλήψεις μήκους 1. |

Ένα τρίτο μειονέκτημα του **Alpha Algorithm** είναι ότι δεν μπορεί να ανιχνεύσει σωστά επαναλήψεις (**loops**) μήκους δύο. Για παράδειγμα το **event log**

= [ ]

Στο **event log**  βλέπουμε τα **transitions** b και d και ανάμεσα τους οποιοσδήποτε αριθμός από επαναλήψεις cb, cb, cb. Αν εφαρμοστεί ο **Alpha Algorithm** πάνω στο **event log**  τότε θα βρεθούν οι παρακάτω σχέσεις

* (a>b, b>c, b>d, c>b)
* (a->b , b->d)
* (b||c)

και θα βρεθεί το Petri Net της παρακάτω εικόνας 26 όπου το **transition c** δεν συνδέεται με κανένα state/place του Petri Net. Όμως χρησιμοποιώντας τεχνικές όπως το pre και post processing το πρόβλημα αυτό μπορεί να ξεπεραστεί.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img26.png  **Εικόνα 26:**  Petri Net που παράγεται από τον **Alpha Algorithm** για **event log** . Σε αυτήν την περίπτωση ο αλγόριθμος δεν μπορεί να ανιχνεύσει loops μήκους δύο. |

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img27.png  **Εικόνα 27:**  Επιθυμητό αποτέλεσμα **process mining** αλγορίθμου για επαναλήψεις μήκους 2. |

Με τα υπόλοιπα μήκη επαναλήψεων ο **Alpha Algorithm** συμπεριφέρεται χωρίς κανένα πρόβλημα και παράγει τα αναμενόμενα Petri Net μοντέλα.

Ένα τέταρτο μειονέκτημα του **Alpha Algorithm** είναι ότι δεν μπορεί να ανιχνεύσει τις **non-local dependencies**. Ο λόγος που συμβαίνει αυτό είναι ότι τέτοιου είδους σχέσεις μςταξύ των στοιχείων του **event log** δεν είναι ορατές στα **logs**. Αυτό βέβαια δεν είναι μόνο πρόβλημα του **Alpha Algorithm,** αλλά και οποιουδήποτε **process mining** αλγορίθμου.

Για παράδειγμα το **event log**

= [ ]

το οποίο είναι πάρα πολύ απλό και βλέπουμε το a που ακολουθείται από το c το οποίο ακολουθείται από το d. Ακόμα βλέπουμε το b που ακολουθείται από το c το οποίο ακολουθείται από το e. Αν τρέξουμε τον **Alpha Algorithm** θα έχουμε σαν αποτέλεσμα το μοντέλο της **Εικόνας 28** και βλέπουμε ότι τα δύο traces που υπάρχουν στο **event log**  είναι όντως πιθανά. Αλλά όμως υπάρχει πιθανότητα να υπάρξει και το trace bcd και ace τα οποία δεν είναι διαθέσιμο στο **event log** . Αυτό συμβαίνει γιατί προφανώς υπάρχει μια σχέση ανάμεσα στα a και d αλλά και στα b και e, αλλά δεν ακολουθούν το ένα το άλλο άμεσα.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img28.png  **Εικόνα 28** |

Το επιθυμητό μοντέλο που έπρεπε να παράγει ο **Alpha Algorithm** είναι το μοντέλο της **εικόνας 29**, στην οποία φαίνονται οι 2 καταστάσεις p1 και p2 οι οποίες δεν βρέθηκαν από τον αλγόριθμο (**non local dependencies**), επειδή τα a και d αλλά και b και e δεν ακολουθούν άμεσα το ένα το άλλο, αλλά μόνο έμμεσα.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img29.png  **Εικόνα 29:**  Τα states p1 και p2 είναι τα **non local dependencies** τα οποία δεν βρέθηκαν από τον αλγόριθμο. |

# 3. Εισαγωγή στην γλώσσα προγραμματισμού “Scala”

# 3.1 Γλώσσα προγραμματισμού Scala

Στην συγκεκριμένη διπλωματική εργασία η υλοποίηση του **Alpha Algorithm** έγινε στην γλώσσα προγραμματισμού **Scala**. Σε αυτό το κεφάλαιο θα γίνει μια σύντομη παρουσίαση των βασικών χαρακτηριστικών αυτής της γλώσσας προγραμματισμού που χρησιμοποείται ευρέως σε **Big Data** εφαρμογές σε συνδυασμό με το **Spark Framework**. Πολλές σχεδιαστικές αποφάσεις της **Scala** έχουν σαν στόχο να αντιμετωπίσουν αδυναμίες της **Java**.

Η **Scala** μια γλώσσα προγραμματισμού γενικού σκοπού που παρέχει υποστήριξη για **functional programming** και είναι μια **strong** **static typing** γλώσσα. **Static typing** σημαίνει ότι ο έλεγχος του τύπου των δεδομένων (**data type**) συμβαίνει στο **compile time**. Ο προγραμματιστής πρέπει να ορίσει τον τύπο των μεταβλητών μέσα στον κώδικα και οποιαδήποτε ενέργεια (**operation**) γίνει στα δεδομένα θα πρέπει να ελεγχθεί από τον **compiler** για τον αν είναι δυνατή ή όχι. Το ακριβώς αντίθετο είναι οι **Dynamic typing** γλώσσες (σε αυτήν την κατηγορία **δεν** ανήκει η **Scala**) στις οποίες ο έλεγχος του τύπου δεδομένων γίνεται στo **runtime** και τα **errors** παράγονται κατά την διάρκεια εκτέλεσης του προγράμματος αν γίνει προσπάθεια εκτέλεσης μιας μη επιτρεπτής ενέργειας (**operation**) πάνω σε μη συμβατά δεδομένα (**incompatible types**). Ακόμα όπως αναφέρθηκε και πιο πάνω η **Scala** είναι **strong typed**, που σημαίνει ότι επιτρέπονται πράξεις (**operations**) πάνω στα δεδομένα που έχουν συγκεκριμένο τύπο (**data type**). Αντίθετα **weak typing** γλώσσες είναι αυτές που επιτρέπουν χειρισμό δεδομένων χωρίς να λάβουν υπόψιν τον τύπο δεδομένων.

Ο πηγαίος κώδικας (**source code**) της **Scala** πρέπει να γίνει compile σε Java bytecode, ώστε το παραγόμενο εκτελέσιμο να μπορεί να τρέξει σε ένα **JVM** (**Java Virtual machine**). Η **Scala** παρέχει **language interoperability** με την Java, ώστε βιβλιοθήκες που έχουν γραφεί στην μία ή στην άλλη γλώσσα να μπορούν να γίνουν διαθέσιμες άμεσα σε κώδικα **Java** ή **Scala.** Επιπλέον όπως η C και η Java είναι μια **object-oriented** γλώσσα.

Η **Scala** διαθέτει πολλά στοιχεία του **functional programming**, όπως:

* **Pure Functions: Pure functions** είναι αυτές οι συναρτήσεις οι οποίες δεν έχουν side effects, δηλαδή δεν αποθηκεύουν κάτι στην μνήμη ή δεν εκτελούν κάποια ενέργεια εισόδου / εξόδου (I/O) αλλά και δεν έχουν εξάρτηση σε τίποτα άλλο πέραν της εισόδου τους. Για παράδειγμα ο παρακάτω κώδικας, ο οποίος δεν επεξεργάζεται τη είσοδο, που σημαίνει ότι το αποτέλεσμα είναι σταθερό ανάλογα με την λίστα των παραμέτρων.

**def** add(a:Int,b:Int) = a + b

Δηλαδή για κάθε είσοδο, το μόνο αποτέλεσμα (**effect**) είναι η έξοδος η οποία παράγει. Το αποτέλεσμα της πρόσθεσης δεν παράγει κανένα **side-effect**. Δεν αλλάζει τις τιμές των εισόδων και επιστρέφει πίσω το άθροισμα.

Μία από τις βασικές αρχές του **Functional Programming** είναι να γράφονται οι εφαρμογές ώστε ο πυρήνας της εφαρμογής να είναι γραμμένος από **pure functions**, και τα οι μέθοδοι που έχουν **side effects** να βρίσκονται σε ένα εξωτερικό επίπεδο. Αυτό έχει σαν αποτέλεσμα ο ο κώδικας να τεστάρετε πολύ πιο εύκολα καθώς αρκεί να ελεγθεί μόνο η έξοδος της μεθόδου και όχι το αποτέλεσμα από κάποιο **side effect**, όπως θα γινόταν σε μια **impure function**. Ακόμα αυτές οι μέθοδοι είναι πιο εύκολο να γίνουν debug γιατί δεν είναι απαραίτητο ο προγραμματιστής να κοιτάξει έξω από το scope της μεθόδου.

* **Immutable Data**: Στο functional programming είναι απαραίτητη η χρήση immutable data. Τα δεδομένα αυτά δεν μπορούν να αλλαχθούν μετά την δημιουργία τους. Ο μόνος τρόπος να αλλάξουν τέτοια δεδομένα είναι να δημιουργηθεί ένα mutable αντίγραφο και να περαστούν νέα δεδομένα πάνω σε αυτό κατά την δημιουργία του. Πλεονέκτημα των **immutable data** είναι ότι δεν χρειάζεται να γίνουν lock κατά την εκτέλεση του προγράμματος, ακόμα και αν τα δεδομένα αυτά είναι απαραίτητα από πολλά threads καθώς κανένα δεν μπορεί να αλλάξει την τιμή τους. Έτσι αυτό βοηθάει στο **concurrency**. Ένα άλλο πλεονέκτημα είναι το **persistence** των δεδομένων καθώς παλαιότερες εκδόσεις των δεδομένων μπορούν να επαναχρησιμοποιηθούν με ασφάλεια διότι δεν πρόκειται να αλλάξουν ποτέ στο μέλλον. Ακόμα δεν είναι απαραίτητο να υπάρχει ιστορικό με το πότε και με τι τιμές άλλαξαν τα δεδομένα, καθώς δεν αλλάζουν ποτέ.
* **First class objects:** Στην **Scala** οι συναρτήσεις είναι **first-class object** που σημαίνει ότι μπορούν να αποθηκευτούν σαν μεταβλητές (**variables**), αντικείμενα (**objects**) ή πίνακες (**arrays**). Ακόμα μπορούν να περαστούν σαν παράμετροι (**arguments**) σε μια μέθοδο αλλά και να επιστραφούν από μια μέθοδο.
* **Currying :** Είναι η τεχνική που επιτρέπει την μετατροπή μιας συνάρτησης (**function**) με πολλά ορίσματα (**arguments**) σε μια συνάρτηση με ένα μόνο όρισμα (**argument**). Το μόνο **argument** είναι η τιμή (**value**) του πρώτου **argument** από την αρχική συνάρτηση (**function**) η οποία επιστρέφει μια άλλη μέθοδο με ένα **argument**. Αυτή η επιστρεφόμενη μέθοδος με την σειρά της θα πάρει σαν όρισμα το δεύτερο όρισμα της αρχικής μεθόδου και θα επιστρέψει μια άλλη μέθοδο με ένα όρισμα. Αυτό το chaining μεταξύ των μεθόδων συμβαίνει για τον αριθμό των ορισμάτων της αρχικής μεθόδου. Το τελευταίο στην αλυσίδα θα έχει πρόσβαση σε όλα τα ορίσματα και θα μπορεί να τα χειριστεί όπως θέλει. Παρακάτω φαίνεται ένα παράδειγμα υλοποίησης της τεχνικής **currying** στην **Scala**.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img30.png  **Εικόνα 30:**  Αυτή η μέθοδος δέχεται ένα όρισμα και επιστρέφει πίσω μια μέθοδο η οποία εκτελεί άθροιση. Η εκτέλεση της γίνεται με “**add(1).apply(1)**” και επιστρέφει σαν αποτέλεσμα 2 (το αναμενόμενο αποτέλεσμα της άθροισης). |

Επειδή η **Scala** υποστηρίζει **curried functions**, ο κώδικας της **Εικόνας 30,** μπορεί να μετατραπεί σε μια πιο απλή **curried version** απλά ξεχωρίζοντας τις παραμέτρους (**arguments**) (**Εικόνα 31**).

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img31.png  **Εικόνα 31:**  Αυτή η μέθοδος εκτελείται με δύο τρόπους   * (add(1) \_).apply(1) με αποτέλεσμα 2 * add(1)(1) με αποτέλεσμα 2 |

* **Type inference:** Αναφέρεται στον αυτόματο εντοπισμό ένος τύπου δεδομένων (**data type**) σε μια γλώσσα προγραμματισμού. Είναι ένα χαρακτηριστικό που υπάρχει σε μερικές **strongly statically typed** γλώσσες. Επιπλέον είναι ένα χαρακτηριστικό των **functional programming** γλωσσών προγραμματισμού. Ορισμένες γλώσσες που έχουν αυτό το χαρακτηριστικό είναι η C++11, C#, Go, Haskell, Java(από την έκδοση 10), Kotlin, Scala.

Στην Scala συγκεκριμένα ο compiler μπορεί να καταλάβει τον τύπο δεδομένων χωρίς να έχει οριστεί ξεκάθαρα από τον προγραμματιστή. Στο παρακάτω παράδειγμα ο **compiler** μπορεί να καταλάβει ότι το **businessName** είναι τύπου String

**val** businessName = **"Montreux Jazz Café"**

Ακόμα στο επόμενο παράδειγμα ο compiler βλέπει το τύπο δεδομένου της επιστρεφόμενης τιμής της μεθόδου σαν Integer και έτσι δεν χρειάζεται να δηλωθεί συγκεκριμένα ο επιστρεφόμενος τύπος δεδομένου (data type).

**def** squareOf(x: Int) = x \* x

* **Lazy evaluation**: Είναι μια τεχνική εκτίμησης η οποία καθυστερεί τον υπολογισμό μιας έκφρασης (**expression**) έως ότου η τιμή της έκφρασης αυτής χρειάζεται πραγματικά. Σήμερα οι περισσότερες μοντέρνες γλώσσες προγραμματισμού υποστηρίζουν την τεχνική του **lazy evaluation**. Σε αντίθεση με την **eager** ή **strict evaluation,** η οποία υπολογίζει τιμές το συντομότερο δυνατό, η **lazy evaluation** παρουσιάζει πλεονεκτήματα όπως ότι μπορεί να βελτιώσει την απόδοση της εφαρμογής επειδή δεν θα εκτελέσει κανέναν υπολογισμό μέχρι αυτός να χρειαστεί πραγματικά. Αυτό είναι πολύ θετικό γιατί μπορεί να υπάρχουν υπολογισμοί σε μια εφαρμογή οι οποίοι τελικά δεν θα εκτελεστούν ποτέ οπότε με το **lazy evaluation** γίνεται οικονομία σε υπολογιστική ισχύ. Ακόμα μπορεί να αυξηθεί το **response time** μιας εφαρμογής αναβάλοντας βαριές λειτουργίες έως ότου είναι απολύτως απαραίτητες. Από την άλλη πλευρά το μειονέκτημα είναι ότι η απόδοση (**performance**) δεν είναι προβλέψιμη επειδή δεν είναι γνωστό το πότε θα χρειαστεί να γίνει **evaluate** (να εκτιμηθεί) η τιμή αυτή.

Η **Scala** μπορεί να κάνει χρήση του lazy evaluation χρησιμοποιώντας τα **lazy vals**. Μια μεταβλητή στην **Scala** μπορεί να χαρακτηριστεί σαν **lazy** (δηλαδή η τιμή της να υπολογιστεί μόλις γίνει η κλήση της μεταβλητής), χρησιμοποιώντας απλά το **lazy keyword**. Για παράδειγμα

**lazy val** lval = 10

Άλλος τρόπος για να κάνει χρήση του lazy evaluation η **Scala** είναι να χρησιμοποιεί **call by name** παραμέτρους. Η τιμή της παραμέτρου θα υπολογιστεί (**evaluated**) μόνο όταν πρόκειται να χρησιμοποιηθεί μέσα στην μέθοδο. Για παράδειγμα το παρακάτων signature της μεθόδου έχει μια **call by name** παράμετρο.

**def** method(n :=> Int)

# 3.2 Λόγοι για να ασχοληθεί κάποιος με την Scala

Υπάρχουν αρκετοί λόγοι για να μάθει κάποιος προγραμματιστής την γλώσσα προγραμματισμού **Scala**. Τα τελευταία χρόνια η **Scala** έχει μετατραπεί στην καλύτερη εναλλακτική λύση της Java, καθώς είναι και αυτή μια **JVM** (Java Virtual Machine) γλώσσα, η οποία έχει αφήσει αρκετά πίσω άλλες **JVM** γλώσσες όπως η **Groovy** και η **Clojure**.

Η **Scala** υποστηρίζει δύο παραδείγματα προγραμματισμού. Το **object-oriented programming** (**OOP**) αλλά και το **functional programming** (**FP**). Για κάποιον προγραμματιστή ο οποίος επιθυμεί να διευρύνει τις γνώσεις του στην επιστήμη των υπολογιστών είναι καλό να γνωρίζει τουλάχιστον μια γλώσσα από κάθε παράδειγμα προγραμματισμού (**imperative**, **functional**, **OOP** και **logical**) και η **Scala** παρέχει την δυνατότητα της συγγραφής κώδικα πάνω σε δύο από αυτά. Ο συνδιασμός αυτών των δύο χαρακτηριστικών κάνει δυνατή την συγγραφή προγραμμάτων τα οποία είναι καθαρογραμμένα (**clean code**) και ευανάγνωστα (**readable**). Το να χρησιμοποιεί δύο παραδείγματα πργραμματισμού (**functional** και **OOP**) η **Scala,** είναι ένα από τα δυνατά χαρτιά της γλώσσας το οποίο ακολούθησε και η **Java** με την έκδοση 8 με την εισαγωγή των **lamda expressions**.

Ένα από τα πλεονεκτήματα της **Scala** είναι ότι υποστηρίζει **interoperability** με την **Java.** Αυτό σημαίνει ότι ο κώδικας της **Scala** μπορεί να χρησιμοποιήσει βιβλιοθήκες της **Java** απευθείας χωρίς καμία παραμετροποίηση. Ακόμα είναι δυνατόν και το αντίθετο, δηλαδή να καλέσει ο προγραμματιστής από **Java** κώδικα, κώδικα **Scala.** Έτσι είναι δυνατόν κάποιος να κρατήσει τα υπάρχοντα προγράμματα σε **Java** και να γράψει τα καινούρια σε **Scala** πάρα πολύ εύκολα.

Η **Scala** ακολουθεί πολλά **best practices** και **patterns** τα οποία είναι ενσωματωμένα στην **built-in** γλώσσα. Σκοπός της **Scala** ήταν να εφαρμόσει πολλές πρόσφατες καινοτομίες σε μια γλώσσα ώστε να χρησιμοποιηθεί όσο το δυνατόν περισσότερο από τους προγραμματιστές, όπως ακριβώς και η **Java**. Ορισμένα **best practices** που ακολουθεί η **Scala** είναι το **immutability**, **type inference, closures** κτλπ**.**

Πολλές εταιρείες τα τελευταία χρόνια σκέφτονται να χρησιμοποιήσουν ή χρησιμοποιούν ήδη **Scala**. Παραδείγματα τέτοιων εταιρειών είναι το Twitter, η Linkedin, το Foursquare και το Quora. Έτσι όπως γίνεται κατανοητό η γνώση της **Scala** θα κάνει κάποιον προγραμματιστή ανταγωνιστικότερο στην αγορά εργασίας. Ακόμα υπάρχουν πολλά αναπτυσσόμενα **frameworks** όπως το **Akka**, το **Play** και το **Spark**. Το **Akka** είναι ένα **Scala based concurrent framework** το οποίο χρησιμοποιείται για την δημιουργία **concurrent**, **distributed** **event-driven** εφαρμογών σε **JVM**. Ακόμα η **Scala** έχει χρησιμοποιηθεί και στην κατηγορία των **Big Data** καθώς πάνω σε αυτή τη γλώσσα έχει γραφεί το **Spark,** ένα **framework** για επεξεργασία δεδομένων σε μια συστάδα υπολογιστών (**cluster)**. Επιπλέον όλο και περισσότεροι προγραμματιστές ασχολούνται πλέον με την **Scala** και αυτό έχει σαν επακόλουθο όλο και περισσότερα **frameworks** και **libraries** να χτίζονται πάνω στην **Scala**. Όλο και περισσότερα IDEs υποστηρίζουν την σύνταξη της **Scala**, όπως το **Eclipse** και το **IntelliJ** αλλά και πολλά build tools (όπως το SBT, Maven και Ant).

Τέλος, για κάποιον προγραμματιστή **Java**, το να μάθει κάποια **functional programming** γλώσσα όπως η Haskell ή η OCaml είναι αρκετά δύσκολο. Σε αντίθεση με την **Scala** που μπορεί να μάθει σχετικά εύκολα καθώς είναι μια γλώσσα που βασίζεται στο **object oriented programming** και έχει αρκετά εύκολη σύνταξη με πολλά κοινά στοιχεία με την **Java**.

# 4. Apache Spark Framework

# 4.1 Εισαγωγή στο Apache Spark Framework

Το **Apache Spark** συνήθως ορίζεται σαν μια γρήγορη, γενικού σκοπού, κατανεμημένης εργασίας πλατφόρμα (**distributed computing platform**). Το **Apache Spark** έφερε επανάσταση στον χώρο των **Big Data** επειδή κάνει αποδοτική χρήση της μνήμης και μπορεί να εκτελέσει διάφορες εργασίες (**jobs**) 10 έως 100 φορές γρηρότερα από το **Hadoop’s MapReduce**. Επιπλέον οι δημιουργοί του **Apache Spark** κατάφεραν να κάνουν abstract το πως ο προγραμματιστής χειρίζεται τα μηχανήματα (**machines**) ενός **cluster** και αντί αυτού να παρουσιάζουν ένα σύνολο από **collections-based APIs**. Η εργασία με τα **collections** του **Spark** είναι παρεμφερής με την εργασία με τα **collections** της **Scala**, της **Java** ή της **Python**, αλλά τα **collections** του **Spark** κατανέμονται σε πολλά **nodes** του **cluster**. Η επεξεργασία αυτών των δεδομένων (**operations**) μεταφράζονται σε πολύπλοκα παράλληλα προγράμματα χωρίς να το αντιλαμβάνεται απαραίτητα αυτό ο χρήστης, κάτι το οποίο είναι πολύ δυνατό χαρακτηριστικό του **Spark** καθώς απλοποιεί την διαδικασία συγγραφής **Spark** προγραμμάτων.

Το **Apache Spark** είναι μια πολύ ενδιαφέρουσα νέα τεχνολογία που συνεχώς ξεπερνάει το **Hadoop’s MapReduce** σαν την προτεινόμενη **Big Data processing** πλαφόρμα. Το **Hadoop** είναι ένα **open source**, κατανεμημένο (**distributed**) framework υπολογισμού δεδομένων γραμμένο σε **Java** και αποτελείται από το **Hadoop Distributed File System** (**HDFS**) και το **MapReduce** (τον μηχανισμό εκτέλεσης εντολών). Το **Apache Spark** μοιάζει με το **Hadoop** στο ότι είναι μία κατανεμημένη (**distributed**), γενικού σκοπού (**general purpose**) υπολογιστική πλατφόρμα (**computing platform**). Αλλά η μοναδική σχεδίαση του **Apache Spark,** η οποία δίνει την δυνατότητα για αποθήκευση μεγάλου όγκου δεδομένων στην κύρια μνήμη, προσφέρει πάρα πολύ μεγάλη αύξηση της απόδοσης. Τα προγράμματα που τρέχουν σε **Spark** μπορεί να είναι έως και 100 φορές γρηγορότερα από αντίστοιχα προγράμματα γραμμένα με την τεχνική του **MapReduce**.

Αν και το **Apache Spark** είναι **open source**, η **Databricks** είναι ο κύριος παράγοντας πίσω από το **Apache Spark** συνεισφέροντας πάνω από το 75% του κώδικα. Ακόμα προσφέρει και το **Databricks Cloud**, ένα εμπορικό προιόν για ανάλυση **Big Data** που βασίζεται πάνω στο **Apache Spark**.

Χρησιμοποιώντας το API και την αρχιτεκτονική του **Apache Spark**, ο προγραμματιστής μπορεί να γράψει κατανεμημένα προγράμματα (**distributed programs**) με τρόπο σαν να έγραφε ένα πρόγραμμα για να τρέξει σε ένα μόνο μηχάνημα. Οι δομές δεδομένων (**collections**) του κρύβουν (**abstract**) το γεγονός ότι τα δεδομένα κατανέμονται (**distributed**) σε ένα μεγάλο αιρθμό από μηχανήματα (**nodes**) σε ένα **cluster**. Επιπλέον το **Apache Spark** επιτρέπει την χρήση μεθόδων που βασίζονται στο **functional programming**, κάτι το οποίο ταιριάζει πολύ σε εργασίες επεξεργασίας δεδομένων (**data-processing tasks**).

Το **Apache Spark** υποστηρίζει μια πληθώρα γλωσσών προγραμματισμού στις οποίες ο προγραμματιστής μπορεί να γράψει προγράμματα επεξεργασίας **Big Data**. Παραδείγματα είναι η **Python**, η **Scala**, η **Java** και η **R**, κάνοντας έτσι το **Apache Spark** εύκολα διαθέσιμο προς ένα μεγάλο εύρος επαγγελματιών. Συνήθως άνθρωποι από τον χώρο των επιστημών προτιμούν την **Python** και την **R,** ενώ προγραμματιστές που ασχολούνται ήδη με **Java** θα προτιμήσουν την **Java** ή την **Scala,** η οποία συνδιάζει **object oriented programming** (**OOP**) και **functional programming** πάνω σε ένα **Java Virtual Machine** (**JVM**).

Το **Apache Spark** συνδιάζει ικανότητες που μοιάζουν με αυτές του **Hadoop’s MapReduce** για επξεργασία παρτίδων (**batch processing**), συναρτήσεις επεξεργασίας δεδομένων σε πραγματικό χρόνο (**real-time data processing functions**), χειρισμό δομημένων δεδομένων (**structured data**) με τεχνικές **SQL**, αλγόριθμοι γράφων (**graph algorithms**) και machine learning όλα σε ένα μοναδικό framework. Έτσι το **Apache Spark** μπορεί να καλύψει τις περισσότερες ανάγκες ενός **Big Data** μηχανικού και αυτός είναι και ο λόγος που το **Apache Spark** έχει γίνει ένα από τα πιο σημαντικά projects της **Apache Software Foundation** σήμερα.

Παρ’ όλα αυτά ορισμένες εφαρμογές δεν είναι κατάλληλες για το το **Apache Spark**. Επειδή το **Apache Spark** διαθέτει μια καταναμημένη αρχιτεκτονική (**distributed architecture**), αυτό έχει σαν επακόλουθο ένα επιπλέον βάρος (**overhead**) στον χρόνο υπολογισμού (**processing time**). Αυτό το **overhead** είναι αμελητέο για το χειρισμό πολύ μεγάλου όγκου δεδομένων, αλλά αν το **dataset** που διαθέτει ο **big data developer** είναι τέτοιο ώστε να μπορεί να επεξεργαστεί και από ένα μόνο μηχάνημα (**single machine**), πράγμα που συμβαίνει όλο και περισσότερο στις μέρες μας, μπορεί να είναι πιο αποδοτικό να χρησιμοποιηθεί κάποιο άλλο framework το οποίο είναι καταλληλότερο για τέτοιου είδους επεξεργασία δεδομένων.

# 4.2 Apache Spark Components

Το **Apache Spark** αποτελείται από διάφορα components συγκεκριμένου σκοπού (**purpose-built components**). Αυτά είναι τα **Spark Core**, **Spark SQL, Spark Streaming, Spark GraphX** και **Spark MLlib**. Αυτά τα **components** φαίνονται στην **Εικόνα 32**. Αυτά τα **components** κάνουν το **Spark** μια ενοποιημένη πλατφόρμα από διάφορα **features** που μπορούν να χρησιμοποιηθούν από πολλές εργασίες που προηγουμένως έπρεπε να εκπληρωθούν από διαφορετικά **frameworks**.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img32.png  **Εικόνα 32:**  Apache Spark components. |

Ακολουθεί σύντομη περιγραφή του κάθε **Apache Spark** component.

* **Spark core**: Περιέχει την βασική λειτουργικότητα του **Apache Spark** η οποία είναι απαραίτητη για την εκτέλεση διάφορων εργασιών στο **Spark (jobs)**, και χρειάζεται για την λειτουργία και άλλων **Spark components**. Το πιο σημαντικό στοιχείο του **Spark core** είναι το **resilient distributed dataset** (**RDD**) τα οποία είναι και το κεντρικό στοιχείο του **Spark API**. Αποτελεί abstraction μιας κατανεμημένης δομής δεδομένων (**distributed collection of data**) με λειτουργίες (**operations**) και μετασχηματισμούς (**transformations**) που είναι εφαρμόσιμες πάνω σε ένα **dataset**. Είναι **resilient** επειδή το **RDD** είναι ικανό να ξανά χτίσει το **dataset** σε περίπτωση που ο κόμβος (**node**) του cluster αποτύχει να λειτουργήσει σωστά για οποιοδήποτε λόγο. Το **Spark core** περιέχει λογική ώστε να μπορεί να προσπελαύνει διάφορα **filesystems**, όπως το HDFS, GlusterFS, Amazon S3 και άλλα. Ακόμα παρέχει διάφορους τρόπους διαμοιρασμού πληροφορίας ανάμεσα σε κόμβους (**nodes**) ενός cluster με τις **broadcast** μεταβλητές (**broadcast variables**) και τους **accumulators**. Ακόμα πολλές βασικές λειτουργίες του **Spark**, όπως η ασφάλεια (**security**), η δικτύωση (**networking**), ηχρονοδρονολόγηση (**scheduling**), και η ανάμειξη δεδομένων (**data shuffling**) είναι επίσης τμήμα του **Spark core**.
* **Spark SQL**: Περιέχει συναρτήσεις (**functions**) για την επεξεργασία μεγάλου όγκου δομημένων (**structured**) και κατανεμημένων (**distributed**) δεδομένων χρησιμοποιώντας εντολές SQL που υποστηρίζονται από το **Spark** και Hive SQL (**HiveQL**). Με την εισαγωγή των **DataFrames** στην έκδοση 1.3 του **Spark** και των **DataSets** στην έκδοση 1.6 του **Spark**, ο χειρισμός των structured data απλουστεύεται και έτσι γίνεται δυνατή η τεράστια αύξηση της απόδοσης των **Spark** προγραμμάτων. Έτσι το **Spark SQL** έγινε ένα από τα σημαντικότερα components του **Spark**. Ακόμα το **Spark SQL** μπορεί επίσης να χρησιμοποιείται για την ανάγνωση και την εγγραφή δεδομένων από και προς διάφορες δομημένες μορφές (**structured format**) και πηγές δεδομένων (**data sources**), όπως τα αρχεία JSON (JavaScript Object Notation), τα **Parquet** αρχεία (όλο και περισσότερο δημοφιλής μορφή αρχείου που επιτρέπει την αποθήκευση ενός σχήματος-**schema** μαζί με τα δεδομένα), σχεσιακές βάσεις δεδομένων (**relational databases**), **Hive database** και άλλα. Οι λειτουργίες (**operations**) πάνω στα DataFrames και στα DataSets σε κάποιο σημείο μεταφράζονται σε λειτουργίες (**operations**) πάνω στα RDDs και εκτελούνται ως κανονικές **Spark** εργασίες (**Spark jobs**).
* **Spark Streaming**: Είναι ένα framework για την επεξεργασία δεδομένων ροής σε πραγματικό χρόνο (**real-time streaming data**) από διάφορες πηγές. Οι υποστηριζόμενες **streaming** πηγές περιλαμβάνουν HDFS, Kafka, Flume, Twitter, ZeroMQ και διάφορες άλλες. Οι **Spark Streaming** λειτουργίες (**operations**) σε περίπτωση αποτυχίας της ορθής λειτουργίας τους μπορούν και επανέρχονται αμέσως στην φυσιολογική χωρίς προβλήματα λειτουργία, κάτι το οποίο είναι σημαντικό για την επεξεργασία δεδομένων σε πραγματικό χρόνο (**real-time**). Το **Spark Streaming** αναπαριστά δεδομένα ροής (**streaming data**), χρησιμοποιώντας διακριτές ροές (**discretized streams** ή αλλιώς **DStreams**), οι οποίες περιοδικά δημιουργούν **RDDs** που περιέχουν δεδομένα που εισήχθησαν κατά το τελευταίο χρονικό παράθυρο (**time window**).Το **Spark Streaming** μπορεί να συνδιαστεί με άλλα **Spark components** μέσα σε ένα πρόγραμμα, ενοποιώντας έτσι την επεξεργασία δεδομένων σε πραγματικό χρόνο (**real-time processing**) με την μηχανική μάθηση (**machine learning**), την SQL αλλά και τις λειτουργίες πάνω σε γράφους (**graph operations**).
* **Spark Mllib:** Είναι μια βιβλιοθήκη αλγορίθμων μηχανικής μάθησης που αναπτύχθηκε από το έργο MLbase στο UC Berkeley. Οι υποστηριζόμενοι αλγόριθμοι περιλαμβάνουν την logistic regression, την ταξινόμηση Bayes (**Bayes classification**), δέντρα αποφάσεων (**decision trees**), την γραμμική παλινδρόμηση (**linear regression**) και την ομαδοποίηση k-means (**k-means clustering**). Το **Spark Mllib** χειρίζεται machine learning μοντέλα που χρειάζονται για να μετασχηματίζουν datasets, τα οποία αναπαρίστανται από RDDs ή DataFrames.
* **Spark GraphX:**  Τα γραφήματα είναι δομές δεδομένων που περιλαμβάνουν τις κορυφές και τις ακμές που τις συνδέουν. Το **Spark GraphX** παρέχει συναρτήσεις για την κατασκευή γράφων, οι οποίες παρουσιάζονται ως graph **RDDs** (δηλαδή **EdgeRDD** και **VertexRDD**). Το **Spark GraphX** περιέχει υλοποιήσεις των πιο σηματικών αλγορίθμων της θεωρείας γράφων, όπως page rank, connected components, συντομότερα μονοπάτια (shortest paths) και πολλούς άλλους.

# 4.3 Ροή Προγράμματος Αpache Spark

Σε αυτό το κεφάλαιο θα γίνει παρουσίαση ενός απλού προγράμματος **Spark**. Υποθέτουμε ότι ένα 300 mb log file είναι αποθηκευμένο σε ένα HDFS cluster με τρία nodes. Το HDFS (Hadoop Distributed File System) αποτελεί ένα σύστημα για κατανεμημένη αποθήκευση αρχείων βασιζόμενη στο Hadoop. Το HDFS αυτόματα χωρίζει το αρχείο σε τμήματα των 128 mb (σε block στην ορολογία του Hadoop) και τοποθετεί το κάθε τμήμα σε ένα ξεχωριστό node του cluster (**Εικόνα 33**). Υποθέτουμε ότι το Spark τρέχει στο YARN, μέσα στο ίδιο Hadoop cluster.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img33.png  **Εικόνα 33:**  Αποθήκευση ενός αρχείου 300mb σε ένα Hadoop cluster με 3 nodes. |

Σε έναν Spark data engineer ανατίθεται το πρόβλημα να αναλύσει τα δεδομένα αυτά και να βρει πόσα errors του τύπου OutOfMemory συνέβησαν τις τελευταίες δύο εβδομάδες. Υποθέτουμε ότι τα logs για τις τελευταίες δύο εβδομάδες είναι αποθηκευμένα στο Hadoop cluster με 3 nodes που αναφέρθηκε πιο πάνω. Το πρόγραμμα αυτό αποτελείται από τις τρεις παρακάτω γραμμές σε Scala

|  |
| --- |
| **val** lines = sc.textFile(**"hdfs://path/to/the/file"**) **val** oomLines = lines.filter(l => l.contains(**"OutOfMemoryError"**)).cache() **val** result = oomLines.count() |

Στην γραμμή (1) γίνεται η φόρτωση του log file από τον HDFS χρησιμοποιώντας μόνο μια γραμμή κώδικα **Scala**. Aυτό φαίνεται και στην **Εικόνα 34** όπου τα δεδομένα μεταφέρονται στην μνήμη RAM κάθε node του cluster. Τώρα το **Spark** έχει ένα δείκτη προς κάθε **partition** των δεδομένων στην μνήμη RAM. Το άθροισμα των **partitions** είναι μια κατανεμημένη συλλογή των γραμμών ενός log file το οποίο γίνεται reference από ένα **RDD** που έχει αρχικοποιηθεί στην πρώτη γραμμή του κώδικα. Με λίγα λόγια, μπορούμε να πούμε ότι τα **RDDs** επιτρέπουν να δουλέψει κάποιος με μια κατανεμημένη δομή δεδομένων (**distributed data collection**) με τον ίδιο ακριβώς τρόπο που θα δούλευε με οποιαδήποτε δομή δεδομένων η οποία δεν είναι κατανεμημένη σε κάποιο cluster αλλά βρίσκεται αποθηκευμένη σε ένα μηχάνημα (local machine). Επιπλέον ο προγραμματιστής δεν χρειάζεται να χειριστεί θέματα που προκύπτουν επειδή τα δεδομένα είναι κατανεμημένα (**distributed**) σε πολλά μηχανήματα (**nodes**) ούτε και διάφορα προβλήματα που μπορεί να προκύψουν σε ορισμένα μηχανήματα του cluster (**node failures**).

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img34.png  **Εικόνα 34:**  Φόρτωση ενός αρχείου κειμένου από τον HDFS. |

Επιπλέον το RDD παρέχει και ένα API, που επιτρέπει στον προγραμματιστή να δουλεύει με μια συλλογή δεδομένων (**collection of data**) με **functional** λογική. Για παράδειγμα είναι δυνατό το φιλτράρισμα των δεδομένων με την εντολή **filter**, η μετατροπή σε κάποιον άλλον τύπο δεδομένων χρησιμοποιώντας την εντολή **map** και μια συνάρτηση (**function**) και το **reduction** των δεδομένων σε μια **cumulative** τιμή με την εντολή reduce. Ακόμα είναι δυνατή η αφαίρεση (**subtract**), η τομή (**intersect**) και η δημιουργία ένωσης (**union**) σε σχέση με κάποιο άλλο RDD.

Έτσι λοιπόν έχοντας ήδη ένα reference στο RDD το οποίο έχει δημιουργηθεί από την γραμμή 1 του παραπάνω κώδικα, για να βρεθεί ο αιρθμός των OutOfMemory errors που συνέβησαν τις τελευταίες δύο εβδομάδες, πρώτα πρέπει να αφαιρεθούν όλες οι γραμμές οι οποίες δεν περιέχουν το substring OutOfMemory. Αυτό μπορεί να γίνει χρησιμοποιώντας τον **τελεστή** filter όπως φαίνεται στην γραμμή 2 του κώδικα. Με την κλήση της μεθόδου **cache()** στην ίδια γραμμή δίνεται η εντολή στο **Spark** να αφήσει το RDD στην μνήμη για όλες τις εκτελέσεις (**jobs**). Έτσι εκτελώντας την εντολή στην τρίτη γραμμή το **Spark** υπολογίζει τον αριθμό των OutOfMemory errors που συνέβησαν τις τελευταίες δύο εβδομάδες.

Σύμφωνα με τα παραπάνω, βλέπουμε ότι το Spark επιτρέπει την εκτέλεση filtering και counting κατανεμημένων δεδομένων (**distributed data**) μόλις σε τρεις γραμμές κώδικα εκτελώντας τον κώδικα σε τρία nodes παράλληλα. Αν ο προγραμματιστής θελήσει να καλέσει ένα επιπλέον filtering στα δεδομένα, το **Spark** δεν θα ξανά φορτώσει το αρχείο από το **HDFS** αλλά θα το φορτώσει από την cache, καθώς αποθηκεύτηκε εκεί στην τελευταία γραμμή του κώδικα.

Το παραπάνω παράδειγμα δείχνει ότι τα RDDs αποτελούν abstraction σε κατανεμημένα δεδομένα (**distributed collections**) και ότι ο χειρισμός και η ανάλυση μεγάλου όγκου δεδομένων μπορεί να γίνει παράλληλα με σχετικά απλή υλοποίηση.

# 4.4 Αρχιτεκτονική του Αpache Spark

Το **Apache Spark framework** έχει μια καλά ορισμένη και layered αρχιτεκτονική όπου όλα τα **Spark components** κα τα **Spark layers** είναι χαλαρά συνδεδεμένα και αλληλεπιδρούν με πολλές βιβλιοθήκες και διάφορες επεκτάσεις (**extentions**). Η αρχιτεκτονική του **Apache Spark** βασίζεται σε δύο βασικές αφαιρέσεις (**abstractions**) οι οποίες είναι οι παρακάτω

* **Resilient Distributed Datasets (RDD)**. Είναι δεδομένα που χωρίζονται σε partitions και αποθηκεύονται στην μνήμη των **worker nodes** σε ένα cluster. Τα **RDDs** υποστηρίζουν δύο διαφορετικούς τύπους λειτουργιών (**operations**), τους μετασχηματισμούς (**transformations**) και τις ενέργεις (**actions**).
* **Directed Acyclic Graph (DAG). DAG** είναι μια ακολουθία από υπολογισμούς που εφαρμόζεται πάνω στα δεδομένα όπου κάθε ακμή είναι ένα RDD partition και κάθε ακμή είναι ένας μετασχηματισμός (**transformation**) πάνω στα δεδομένα. Το γράφημα αυτό λέγεται acyclic επειδή κάποιο transformation δεν μπορεί να επιστρέψει σε ένα παλαιότερο partition.

Το **Apache Spark framework** χρησιμοποιεί μια master / worker αρχιτεκτονική. Υπάρχει κάποιο **node** στο **cluster** το οποίο παίζει τον ρόλο του **driver** ο οποίος μιλάει με έναν μοναδικό coordinator ο οποίος λέγεται **master (cluster manager)** και διαχειρίζεται τους υπόλοιπους nodes του cluster οι οποίοι λέγονται **workers** και στους οποίους τρέχουν οι **executors**. Αυτή η αρχιτεκτονική φαίνεται στην **εικόνα 35**.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img35.png  **Εικόνα 35:**  Master / worker αρχιτεκτονική του **Apache Spark**. |

Ο **driver** και οι **executors** τρέχουν τις δικές τους Java διεργασίες (**processes**). Μπορούν να τρέχουν στο ίδιο μηχάνημα (**horizontal cluster**) ή σε ξεχωριστά μηχανήματα (**vertical cluster**).

Οι εφαρμογές **Spark** τρέχουν σαν ανεξάρτητα σύνολα από διαδικασίες πάνω σε ένα cluster, το οποίο ενορχηστρώνεται (coordinated) από ένα **SparkContext** αντικείμενο το οποίο βρίσκεται στο main πρόγραμμα και ονομάζεται **driver program**. Συγκεκριμένα για να τρέξει σε ένα cluster, το **SparkContext** συνδέεται σε κάποιον **cluster manager**, είτε στον standalone cluster manager του ίδιου του Spark, είτε στον **Mesos** ή στον **YARN**, από τους οποίους εντοπίζει τα resources που θα χρειαστεί για το application που θα τρέξει. Μόλις συνδεδεί, το **Spark** αποκτά executors πάνω στα διαθέσιμα **nodes** του cluster, οι οποίοι είναι διεργασίες που εκτελούν υπολογισμούς και αποθηκεύουν δεδομένα για την εφαρμογή που τρέχει στο cluster. Έπειτα, στέλνει τον κώδικα της εφαρμογής (που ορίζεται σαν JAR και περνάει στο **SparkContext**) στους **executors**. Τέλος το **SparkContext** στέλνει tasks στους **executors** για να εκτελεστούν.

Το **Spark** **driver** πρόγραμμα τρέχει την main μέθοδο της εφαρμογής και είναι το σημείο όπου το **SparkContext** δημιουργείται. Το πρόγραμμα **Spark** **driver** περιέχει διάφορα component όπως ο DAGScheduler, TaskScheduler και BlockManager οι οποίοι είναι υπεύθυνοι για την μετάφραση του **Spark** κώδικα που έχει γράψει ο χρήστης σε πραγματικά **Spark jobs** τα οποία θα εκτελεστούν στο cluster. Το **Spark** **driver** πρόγραμμα είναι υπεύθυνο για τα ακόλουθα

* Τρέχει στο master node του Spark cluster, δρομολογεί την εκτέλεση των jobs και επικοινωνεί με τον **cluster manager**.
* Μεταφράζει τα RDDs σε γράφημα εκτέλεσης και χωρίζει το γράφημα αυτό σε πολλά **stages**.
* Σώζει τα metadata για όλα τα RDDs και τα partitions τους.
* Δημιουργεί τα jobs και τα tasks. Χωρίζει την εφαρμογή του χρήστη σε μικρότερες μονάδες εκτέλεσης που ονομάζονται tasks. Τελικά τα tasks θα εκτελεστούν από τους **executors**.
* Συλλέγει τα αποτελέσματα από τους **executors**.

Ο **executor** είναι ένα κατανεμημένο (distributed) πρόγραμμα υπεύθυνο για την εκτέλεση των tasks. Οι **executors** τρέχουνκαθ’ όλητην διάρκεια μιας **Spark** εφαρμογής αλλά οι χρήστες έχουν την δυνατότητα εάν θελήσουν να προσθέσουν ή να αφαιρέσουν **executors** δυναμικά την ώρα που τρέχει η εφαρμογή.Συγκεκριμένα οι **executors** είναι υπεύθυνοι για

* Την επεξεργασία των δεδομένων (**data processing**)
* Το διάβασμα και την εγγραφή σε εξωτερικές πηγές.
* Την αποθήκευση αποτελεσμάτων από την επεξεργασία των δεδομένων στην μνήμη, στην cache ή στον σκλήρο δίσκο.

Ο **cluster manager** σε μια **Spark** είναι ένα εξωτερικό service το οποίο είναι υπεύθυνο για την παροχή των απαραίτητων πόρων (**resources**) σε ένα **Spark** cluster και την διάθεση τους σε ένα **Spark job**. Υπάρχουν τρεις διαφορετικοί τύποι από **cluster managers** που είναι διαθέσιμοι σε μια **Spark** εφαρμογή, ο Hadoop YARN, ο Apache Mesos και ο απλός standalone Spark cluster manager. Η επιλογή του κατάλληλου **cluster manager** εξαρτάται από τον σκοπό της **Spark** εφαρμογής γιατί κάθε **cluster manager** παρέχει διαφορετικά σύνολα από δυνατότητες προγραμματισμού. Ο απλός standalone Spark cluster manager είναι ο πιο εύκολος στην χρήση σε σχέση με τους υπόλοιπους.

# 4.5 RDD Δομές δεδομένων

Μία από τις βασικές δομές δεδομένων του Spark είναι τα **Resilient Distributed Datasets** (**RDD**). Είναι μια **immutable** κατανεμημένη (**distributed**) δομή δεδομένων από διάφορα αντικείμενα (**objects**). Κάθε σύνολο δεδομένων σε ένα RDD χωρίζεται σε λογικά partitions, τα οποία μπορεί να επεξεργάζονται σε διαφορετικά μηχανήματα (**nodes**) σε ένα cluster. Τα **RDDs** μπορούν να περιέχουν οποιονδήποτε τύπο δεδομένων της Python, της Java ή της Scala. Τα **RDDs** είναι μια **fault tolerant** δομή δεδομένων καθώς όταν κάποιο partition καταστραφεί για κάποιο λόγο τότε το σύστημα διαθέτει αρκετή πληροφορία ώστε να το κατασκευάσει ξανά. Το **Spark** χρησιμοποιεί τα **RDDs** για να επιτύχει ταχύτερες και αποδοτικότερες ενέργειες (**operations**) MapReduce.

Στην **Εικόνα 36** φαίνεται παράδειγμα 2 διαφορετικών τύπων **RDDs**.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img36.png  **Εικόνα 36:**  RDDs από Strings (αριστερά) και RDDs από Pairs (δεξιά) |

Υπάρχουν δύο τρόποι για να δημιουργηθεί ένα **RDD**

* Με τη παραλληλοποίηση μια υπάρχουσας δομής δεδομένων στο **Spark** **driver** πρόγραμμα. Στην ουσία το **Spark** **driver** διαχωρίζει σε partitions τα δεδομένα και τα στέλνει στα nodes του cluster.
* Από κάποια αρχεία που βρίσκονται σε κάποιο εξωτερικό σύστημα αποθήκευσης όπως ένα κοινόχρηστο σύστημα αρχείων (**shared file system**), ένα HDFS ή οποιαδήποτε πηγή δεδομένων που προσφέρει ένα Hadoop format.

Συνοψίζοντας τα **RDDs** έχουν τα ακόλουθα χαρακτηριστικά

* **In-memory**: Τα δεδομένα μέσα σε ένα **RDD** αποθηκεύονται στην μνήμη για όσο περισσότερο χρόνο γίνεται.
* **Immutable ή Read-only**: Δεν γίνεται να αλλάξουν από την στιγμή που δημιουργηθούν και μπορούν μόνο να μετασχηματιστούν με **transformations** σε νέα **RDDs.**
* **Lazy evaluated**: Τα δεδομένα μέσα στο **RDD** δεν είναι διαθέσιμα έως ότου εκτελεστεί σε αυτά μια εντολή τύπου **action**.
* **Cacheable**: Τα δεδομένα των **RDD** μπορούν να αποθηκεύονται στην μνήμη (που είναι το επιθυμητό για λόγους ταχύτητας) αλλά και στο σκληρό δίσκο.
* **Parallel**: Τα **RDDs** μπορούν να επεξεργάζονται παράλληλα.
* **Typed:** Τα **RDDs** είναι **typed** δεδομένα για παράδειγμα RDD[Float] ή RDD[(Int, String)]
* **Partitioned**: Τα **RDDs** διαχωρίζονται σε λογικά τμήματα τα partitions και μοιράζονται σε όλα τα nodes του cluster.

# 4.6 Spark SQL, DataFrames and Datasets

Το **Spark SQL** είναι ένα module του **Spark** το οποίο χρησιμοποιείται για επεξεργασία δομημένων δεδομένων. Σε αντίθεση με το API των **RDD**, το API που παρέχει το **Spark SQL** περιέχει περισσότερες πληροφορίες για την δομή τόσο των δεδομένων όσο και των υπολογισμών που θα γίνουν. Εσωτερικά το **Spark SQL** εκμεταλλεύεται αυτές τις πληροφορίες για να βελτιώσει την απόδοση της εφαρμογής. Υπάρχουν πολλοί τρόποι αλληλεπίδρασης με το **Spark SQL** συμπεριλαμβανομένου της **SQL** αλλά και του Dataset API. Μια χρήση του **Spark SQL** είναι η εκτέλεση **SQL queries**, τα οποία όταν εκτελούνται μέσα από κάποια γλώσσα προγραμματισμού πάντα επιστρέφουν τα αποτελέσματα σαν **DataFrames** ή σαν **Datasets**.

**Dataset** είναι μια κατανεμημένη (**distributed**) συλλογή από δεδομένα. Το **Dataset** είναι ένα interface το οποίο προστέθηκε στην έκδοση 1.6 του **Spark** και παρέχει όλα τα πλεονεκτήματα των **RDDs** (strong typing, χρήση εκφράσεων lamdas) με τα οφέλη της βελτιωμένης **Spark SQL** μηχανής επεξεργασίας δεδομένων. Ένα **Dataset** μπορεί να κατασκευαστεί από JVM αντικείμενα και να χρησιμοποιηθεί από functional μετασχηματισμούς (όπως map, flatMap, filter κα). Το **Dataset API** είναι διαθέσιμο σε Scala και Java. Τα **Dataset** είναι **lazy** δηλαδή ο υπολογισμός τους πυροδοτείται μόνο όταν καλεστεί κάποιο action.

**Dataframe** είναι ένα **Dataset** το οποίο οργανώνεται σε στήλες με όνομα. Είναι ισοδύναμο με ένα πίνακα σε μια σχεσιακή βάση δεδομένων ή με ένα data frame στην Python και την R, αλλά με πολλές πλούσιες βελτιστοποιήσεις. Τα **Dataframes** μπορούν να κατασκευαστούν από πολλές πηγές δεδομένων όπως δομημένα αρχεία δεδομένων, πίνακες σε ένα dataware house σαν το Hive, από εξωτερικές βάσεις δεδομένων και υπάρχοντα **RDDs**. Το **Dataframe API** είναι διαθέσιμο σε Scala, Java, Python και R. Στην Scala και στην Java ένα **Dataframe** παρουσίαζεται σαν **Dataset** από **Row**. Δηλαδή στην Java σαν Dataset<Row> και στην Scala σαν DataSet[Row].

# 4.7 Βασικές μέθοδοι (operations) του Apache Spark

Οι μέθοδοι (**operations**) στο **Spark** (και για τα **RDDs** και τα **Datasets**) χωρίζονται σε μετασχηματισμούς (**transformations**) και ενέργεις (**actions**). Τα **transformations** θα παράξουν νέα **RDDs** ή **Datasets** και δεν μπορούν να τροποποιήσουν τα δεδομένα εισόδου, και τα **actions** θα πυροδοτήσουν τον υπολογισμό και θα επιστρέψουν τα αποτελέσματα. Τα actions ακόμα μεταφέρουν τα δεδομένα που έχουν υπολογιστεί από τα nodes του cluster, δηλαδή τους executors, στον **driver** του **Spark**. Αυτές οι μέθοδοι (**operations**) ταιριάζουν με το MapReduce του Hadoop, καθώς το Map υλοποείται από τα **transformations** του **Spark** και το Reduce από τα **actions.**

Ακόμα υπάρχουν συγκεκριμένα **operations** στο **Spark** που πυροδοτούν το “ανακάτεμα” (**shuffling**) των δεδομένων. Το **shuffle** είναι ένας μηχανισμός του **Spark** για να ξανά-κατανήμει (**re-distribute**) τα δεδομένα που έχουν ομαδοποιηθεί διαφορετικά σε όλα τα partitions. Αυτό συνήθως περιλαμβάνει αντιγραφή των αρχείων ανάμεσα στους executors και στα μηχανήματα του cluster, κάτι το οποίο αποτελεί και μια πολύ ακριβή διαδικασία.

Ορισμένα παραδείγματα **Spark** **transformations** είναι:

* **map**
* **filter**
* **flatMap**
* **mapPartitions**
* **mapPartitionsWithIndex**
* **sample**
* **union**
* **intersection**
* **distinct**
* **groupByKey**
* **reduceByKey**
* **aggregateByKey**
* **sortByKey**
* **join**
* **cartesian**

Ορισμένα παραδείγματα **Spark** **actions** είναι:

* **reduce**
* **collect**
* **count**
* **first**
* **take(n)**
* **takeSample**
* **saveAsTextFile**
* **countByKey**
* **foreach**

# 4.8 Shared variables και αποθήκευση μεταβλητών στο Spark

Όπως φαίνεται στα προηγούμενα κεφάλαια, μια μεγάλη αφαίρεση (**abstraction**) που προσφέρει το **Apache Spark** είναι τα **Resilient Distributed Datasets** (**RDD**), τα οποία είναι μια συλλογή δεδομένων που κατανέμονται σε όλα τους κόμβους (**nodes**) του cluster και μπορούν να επεξεργαστούν παράλληλα. Ο προγραμματιστής έχει την δυνατότητα να αποθηκεύσει τα **RDDs** στην μνήμη κάθε node ώστε να χρησιμοποιείται αποδοτικότερα στην παράλληλη επεξεργασία των δεδομένων.

Μια δεύτερη αφαίρεση (**abstraction**) που προσφέρει το **Apache Spark** είναι οι **shared variables** που μπορούν να χρησιμοποιήσουν παράλληλες ενέργειες πάνω στα δεδομένα. Όταν το **Spark** τρέχει κάποια συνάρτηση παράλληλα σαν ένα σύνολο από ενέργειες (**tasks**) σε διαφορετικά **nodes** του cluster, στέλνει ένα αντίγραφο κάθε μεταβλητής που χρησιμοποείται στην συνάρτηση σε κάθε task. Κάποιες φορές, αυτή η μεταβλητή χρειάζεται να είναι κοινή ανάμεσα σε όλα τα tasks ή κοινή ανάμεσα στα tasks και στο **driver program**. Το **Spark** διαθέτει δύο είδη **shared variables**

* τις **broadcast variables**, που μπορούν να χρησιμοποιηθούν για να αποθηκευτεί μια read-only τιμή της μεταβλητής στην μνήμη όλων των nodes του cluster, παρά να μεταφέρεται ένα αντίγραφο στο node που θα εκτελέσει την ενέργεια. Για παράδειγμα μπορεί να δωθεί σε ένα node ένα πολύ μεγάλο αντίγραφο ενός dataset αποφεύγοντας έτσι το πολύ μεγάλο network transfer I/O. Το **Spark** διαθέτει διάφορους broadcast αλγορίθμους για μειώνει το κόστος τις επικοινωνίας μταξύ των nodes και του driver program. Στον παρακάτω Scala κώδικα φαίνεται η δημιουργία και η χρήση μιας **broadcast variable.**

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().getOrCreate()  **val** *broadcastVar* = *spark*.sparkContext.broadcast(*Array*(1, 2, 3)) *broadcastVar*.value |

* τους **accumulators**, που είναι μεταβλητές στις οποίες μπορούν να προστεθούν τιμές, όπως για παράδειγμα μετρητές (counters) και αθροίσματα (sums). Στον παρακάτω Scala κώδικα φαίνεται η δημιουργία και η χρήση ενός **accumulator.**

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().getOrCreate() **val** *accum* = *spark*.sparkContext.accumulator(0, **"Accumulator Example"**) *spark*.sparkContext.parallelize(*Array*(1, 2, 3)).foreach(x => *accum* += x) *accum*.value |

Εκτός από τις **broadcast variables** και τους **accumulators**, ο χρήστης του **Spark** έχει την δυνατότητα να αποθηκεύσει **RDD** στην μνήμη. Αυτή η τεχνική ονομάζεται **caching** ή **persistence** και είναι τεχνικές βελτιστοποίησης για επαναλαμβανόμενους **Spark** υπολογισμούς. Συγκεκριμένα σώζουν ενδιάμεσα αποτελέσματα ώστε να μπορούν να επαναχρησιμοποιηθούν σε επόμενα στάδια. Τα αποτελέσματα αυτά μπορούν να σωθούν στην μνήμη ή σε κάποιον σκληρό δίσκο.

Τα **RDDs** μπορούν να αποθηκευτούν χρησιμοποιώντας τις εντολές **cache()** ή **persist()**. Η διαφορά μεταξύ αυτών των δύο εντολών είναι μόνο συντακτική. Η **cache()** είναι ακριβώς ίδια με την εντολή persist(MEMORY\_ONLY). Ακόμα τα **RDDs** μπορούν να διαγραφούν από την μνήμη ή τον σκληρό δίσκο που αποθηκεύτηκαν χρησιμοποιώντας την εντολή **unpersist()**.

# 5. Υλοποίηση του Alpha Algorithm

# 5.1 Κώδικας Υλοποίησης

Για την υλοποίηση του **Alpha Algorithm** χρησιμοποιήθηκε η γλώσσα προγραμματισμού **Scala**, η οποία παρουσιάστηκε σε προηγούμενο κεφάλαιο. Υλοποιήθηκαν δύο εκδόσεις του **Alpha Algorithm**, η μία χρησιμοποιώντας το **Spark framework** και η άλλη χωρίς την χρήση του εν λόγω framework. Σκοπός είναι η μέτρηση των χρόνων και για τις δύο υλοποιήσεις, ώστε να γίνει εξαγωγή συμπερασμάτων για το πότε είναι χρήσιμη η χρήση του **Spark framework** και πότε όχι.

Σαν είσοδο δεδομένων (**input dataset**) χρησιμοποιήθηκε ένα **event log** από έναν οργανισμό τηλεπικοινωνιών. Το συγκεκριμένο csv αρχείο έχει μέγεθος περίπου 400mb και περιέχει 36100 traces.

Πολλά από τα παραπάνω **traces** επαναλαμβάνονται πάρα πολλές φορές για αυτό υλοποιήθηκε κώδικας ο οποίος είναι παραμετροποιήσιμος και μπορεί να τροφοδοτήσει στον **Alpha Algorithm** τα πιο συχνά εμφανιζόμενα traces ή έναν συγκεκριμένο αριθμό από traces (ο οποίος προσδιορίζεται από το χρήστη) ή αλλιώς όλα τα traces που διαθέτει το **event log**. Αυτή η υλοποίηση είναι απαραίτητη γιατί το παραγόμενο αποτέλεσμα του **Alpha Algorithm** δεν εξαρτάται από το πόσες φορές εμφανίζεται ένα trace στο **event log.** Ακόμα κάποια πολύ σπάνια εμφανιζόμενα traces μπορεί να μην είναι επιθυμητό να επεξεργαστούν από τον **Alpha Algorithm** καθώς ενδέχεται να αλλάξουν το τελικό Petri net κατά πολύ και τελικά να μην απεικονίζει τις συνήθεις διαδικασίες (**processes**).

|  |
| --- |
| **//Κώδικας 1**  **def** main(args: Array[String]): Unit = {  Logger.*getLogger*(**"org"**).setLevel(Level.*ERROR*)  **val** traceTools: TraceTools = **new** TraceTools()  **val** logPath = **"src/main/resources/data.csv"  val** numOfTraces = 3  **val** percentage : Float = 1 **val** readAll : Boolean = **false  val** filtering : Boolean = **true   val** spark = SparkSession  .*builder*()  .appName(**"AlphaAlgorithm"**)  .master(**"local[\*]"**).getOrCreate()   **val** tracesDS : Dataset[(String, List[String])] = traceTools.getTracesToInspect(logPath, numOfTraces, readAll, filtering, percentage)   **val** petriNet: PetriNet = *executeAlphaAlgorithm*(tracesDS)  *println*(petriNet)   *// Stop the session* spark.stop() } |

Στον παραπάνω κώδικα (**Κώδικας 1**) φαίνεται η **main** συνάρτηση από την οποία ξεκινάει ο **Alpha Algorithm** καθώς και οι διάφοροι παράμετροι που πρέπει να επιλέξει ο χρήστης ώστε να ξεκινήσει η εκτέλεση του **Alpha Algorithm**.Αυτές οι μεταβλητές είναι

* **logPath:** Το path που θα διαβάσει το πρόγραμμα το event log που θα επεξεργαστεί.
* **readAll:** Boolean μεταβλητή η οποία αν είναι true διαβάζει όλα τα traces από το event log, αλλιώς θα διαβάσει τόσα traces όσα έχουν δηλωθεί στην μεταβλητή **numOfTraces.**
* **filtering:** Boolean μεταβλητή η οποία αν είναι true φιλτράρει τα traces και κρατάει μόνο όσα έχουν ποσοστό εμφάνισης άνω της τιμής της μεταβλητής **percentage.**
* **percentage:** Ποσοστόπου δείχνει ποιο είναι το κάτω όριο ποσοστού εμφάνισης ώστε να επεξεργαστεί ένα trace από τον **Alpha Algorithm.** Αν για παράδειγμα η τιμής της μεταβλητής είναι 1, τότε αν και το **filtering** είναι **true,** ο **Alpha Algorithm** θα επεξεργαστεί μόνο traces που έχουν ποσοστό εμφάνισης πάνω από 1%.
* **numOfTraces:** Ο αριθμός των traces που θα επεξεργαστεί ο **Alpha Algorithm**. Για να λάβει το πρόγραμμα υπόψιν αυτήν την παράμετρο πρέπει το **readAll** να είναι **false**.

Ακόμα στην main μέθοδο αρχικοποιείται το **SparkSession**, τα traces μέσω της μεθόδου **getTracesToInspect,** πυροδοτείται ο **Alpha Algorithm** μέσω της μεθόδου **executeAlphaAlgorithm** και εκτυπώνεται το παραγόμενο Petri Net.

|  |
| --- |
| **//Κώδικας 2**  **def** readAllTracesFromCsvFile(path: String) : Dataset[(String, List[String])] = {  **val** spark = SparkSession.*builder*().getOrCreate()  **import** spark.implicits.\_   **val** df = spark.read.format(**"csv"**).option(**"header"**, **"true"**).load(path)   *//Dataset[(String, List[String])]* **return** df.select(**"orderid"**, **"eventname"**, **"starttime"**, **"endtime"**, **"status"**)  .filter(df(**"status"**).isin(*List*(**"Completed"**):\_\*)) .orderBy(**"starttime"**)  .map(x=>(x.get(0).toString,x.get(1).toString))  .groupByKey(x=>x.\_1)  .mapGroups{**case**(k, iter) => (k, iter.map(x => x.\_2).toList)} } |

Στον **Κώδικα 2** παρουσιάζεται η μέθοδος η **readAllTracesFromCsvFile** οποία διαβάζει όλα τα traces από το event log (ένα csv file εν προκειμένω). Συγκεκριμένα διαλέγει κάποιες στήλες από το csv file, φιλτράρει τις γραμμές ώστε να κρατήσει μόνο όσες έχουν status “**Completed**” (καθώς μόνο τα συγκεκριμένα events πρέπει να επεξεργαστούν), τις τοποθετεί σε σειρά σύμφωνα με το πότε ξεκίνησαν “**starttime**” και τέλος τις ομαδοποιεί σύμφωνα με το **orderid** (functional operators **groupByKey** και **mapGroups**). Έτσι δημιουργεί traces με την μορφή **Dataset[(String, List[String])]**, όπου το πρώτο στοιχείο του tuple είναι το **orderid** και το δεύτερο είναι μια λίστα από events.

|  |
| --- |
| **//Κώδικας 3**  **def** readSpecificNumberOfTracesFromCsvFile(path: String, numOfTraces: Int) : Dataset[(String, List[String])] = {  **val** spark = SparkSession.*builder*().getOrCreate()  **import** spark.implicits.\_   **val** df = spark.read.format(**"csv"**).option(**"header"**, **"true"**).load(path)   **val** orderIds = df.select(**"orderid"**)  .distinct()  .limit(numOfTraces)  .as(Encoders.*STRING*)  .collect()  .toList   *//Dataset[(String, List[String])]* **return** df.select(**"orderid"**, **"eventname"**, **"starttime"**, **"endtime"**, **"status"**)  .where( df(**"orderid"**).isin(orderIds:\_\*))  .filter(df(**"status"**).isin(*List*(**"Completed"**):\_\*)) .orderBy(**"starttime"**)  .map(x=>(x.get(0).toString,x.get(1).toString))  .groupByKey(x=>x.\_1)  .mapGroups{**case**(k, iter) => (k, iter.map(x => x.\_2).toList)} } |

Ομοίως ο **Κώδικας 3**, διαβάζει έναν συγκεκριμένο αιρθμό από traces. Ο αριθμός αυτός δίνεται σαν είσοδος στην μέθοδο (**numOfTraces**) και επιστρέφει τα traces στην μορφή **Dataset[(String, List[String])]**. Η διαφορά με τον **Κώδικα 2** είναι ότι βρίσκει από την αρχή τα orderIds που θα επιστρέψει σαν traces και έπειτα εκτελεί ένα **where operation** ώστε να κρατήσει μόνο τα traces με τα συγκεκριμένα orderIds.

Έχοντας υπολογίσει πλέον τα traces (έστω tracesDS), το επόμενο βήμα του προγράμματος είναι να φιλτράρει τα traces ανάλογα με την boolean παράμετρο **filtering**. Αν η **filtering** έχει τιμή true τότε τα αρχικό σύνολο από traces θα επεξεργαστεί ώστε να διαθέτει μόνο τα traces εκείνα που ικανοποιούν την παράμετρο **percentage**. Η διαδικασία αυτή του φιλτραρίσματος του αρχικού συνόλου από traces φαίνεται στον **Κώδικας 4.**

|  |
| --- |
| **//Κώδικας 4**  **def** filterTraces(tracesDS: Dataset[(String, List[String])], percentage: Float): Dataset[(String, List[String])] = {  **implicit def** listStringEncoder: org.apache.spark.sql.Encoder[List[String]] = org.apache.spark.sql.Encoders.*kryo*[List[String]]  **implicit def** tupleListStringEncoder: org.apache.spark.sql.Encoder[(String, List[String])] = org.apache.spark.sql.Encoders.*kryo*[(String, List[String])]  **implicit def** longStringEncoder: org.apache.spark.sql.Encoder[(List[String], Long)] = org.apache.spark.sql.Encoders.*kryo*[(List[String], Long)]  **implicit def** floatStringEncoder: org.apache.spark.sql.Encoder[(List[String], Float)] = org.apache.spark.sql.Encoders.*kryo*[(List[String], Float)]   **val** initNumberOfTraces = tracesDS.count()  *println*(**"Initial number of traces = "** + initNumberOfTraces)   **import** org.apache.spark.sql.functions.\_  **val** tracesToInspect = tracesDS  .toDF(**"traceId"**, **"trace"**)  .groupBy(**"trace"**)  .agg(*count*(**"trace"**))  .map(trace=>(trace.getAs[Seq[String]](**"trace"**).toList,trace.getAs[Long](**"count(trace)"**)))  .filter(trace => (trace.\_2.toFloat / initNumberOfTraces) > (percentage/100) )  .map(trace=>(**"xxx"**, trace.\_1))   *println*(**"Number of traces to inspect = "** + tracesToInspect.count())  tracesToInspect } |

Στην αρχή του **Κώδικα 4** τέσσερεις **encoders** οι οποίοι είναι απαραίτητοι για την εκτέλεση του προγράμματος **Spark**. Ο **encoder** είναι ένα βασικό στοιχείο για το serialization και το deserialization των δεδομένων στο **Spark** ώστε να είναι δυνατή η αποστολή τους από το driver program στα nodes του cluster σε όσο το δυνατόν μικρότερο χρονικό διάστημα αλλά και σε όσο το δυνατόν μικρότερο μέγεθος. Στον **Κώδικα 4** έχουν δημιουργηθεί **encoders** για τους παρακάτω τύπους Scala δεδομένων, οι οποίοι επεξεργάζονται ώστε να κατασκευαστεί το **tracesToInspect (**τύπου **Dataset[(String, List[String])] ).** Οι τύποι δεδομένων είναι

* **List[String] :** λίστα από String.
* **(String , List[String] ) :** Tuple που περίεχει String στο πρώτο στοιχείο και λίστα από String στο δεύτερο στοιχείο.
* **(List[String] , Long) :** Tuple που περίεχει λίστα από String στο πρώτο στοιχείο και Long στο δεύτερο στοιχείο.
* **(List[String] , Float) :** Tuple που περίεχει λίστα από String στο πρώτο στοιχείο και Float στο δεύτερο στοιχείο.

Για τον υπολογισμό του φιλτραρισμένου event log, όπως φαίνεται στον **Κώδικα 4,** ακολουθείταιηπαρακάτων διαδικασία

* Μετατροπή του αρχικό Dataset από traces σε Dataframe με στήλες **traceId** και **trace** (**toDF** operation)
* Ομαδοποίηση των γραμμών του Dataframe σύμφωνα με την δεύτερη στήλη **trace** (**groupBy** operation). Έτσι ο κώδικας βρίσκει μοναδικά traces και υπολογίζει πόσες φορές εμφανίζετε το καθένα (**agg** operation).
* Μετασχηματισμός των δεδομένων σε ένα Tuple με πρώτο στοιχείο το trace και δεύτερο στοιχείο των αριθμό των εμφανίσεων του στο event log (πρώτο **map** operation).
* Φιλτράρισμα των δεδομένων ανάλογα με το ποσοστό εμφάνισης στο event log. Αν το ποσοστό εμφάνισης (**trace.\_2.toFloat / initNumberOfTraces**) είναι πάνω από την τιμή **percentage** τότε το trace θα επεξεργαστεί από τον **Alpha Algorithm** (**filter** operation).

Μέχρι αυτό το σημείο έχει γίνει ο υπολογισμός των traces που θα επεξεργαστούν από τον Alpha Algorithm. Τα βήματα εκτέλεσης του αλγορίθμου φαίνονται στον **Κώδικα 5.** Στον κώδικα φαίνονται τα 8 βήματα που ακολουθεί ο Alpha Algorithm ώστε να υπολογίσει το τελικό Petri Net από το event log δέχεται σαν είσοδο (**traceDS**). Συγκεκριμένα κάθε βήμα του αλγορίθμου αντιστοιχεί και στην κλήση μιας συνάρτησης, εκτός από το βήμα 4 που λόγω της πολυπλοκότητας του έχει σπάσει σε δύο διαφορετικές μεθόδους, μία για τον υπολογισμό του **footprint graph** και η δεύτερη για τον υπολογισμό των **causal groups**.

|  |
| --- |
| **//Κώδικας 5**  **def** executeAlphaAlgorithm(tracesDS : Dataset[(String, List[String])]) : PetriNet = {   **val** steps : AlphaAlgorithmSteps = **new** AlphaAlgorithmSteps()  tracesDS.cache()   *//Step 1 - Find all transitions / events, Sorted list of all event types* **val** events = steps.getAllEvents(tracesDS)   *//Step 2 - Construct a set with all start activities (Ti)* **val** startActivities = steps.getStartActivities(tracesDS)   *//Step 3 - Construct a set with all final activities (To)* **val** finalActivities = steps.getFinalActivities(tracesDS)   *//Step 4 - Footprint graph - Causal groups* **val** logRelations : Dataset[(Pair, String)] = steps.getFootprintGraph(tracesDS, events)  tracesDS.unpersist()  **val** causalGroups : Dataset[CausalGroup[String]] = steps.getCausalGroups(logRelations) *//Step 5 - compute only maximal groups* **val** maximalGroups : List[CausalGroup[String]] = steps.getMaximalGroups(causalGroups)   *//step 6 - set of places/states* **val** places : Places = steps.getPlaces(maximalGroups, startActivities, finalActivities)   *//step 7 - set of arcs (flow)* **val** edges : List[Edge] = steps.getEdges(places)   *//step 8 - construct petri net* **return new** PetriNet(places, events, edges) } |

Το πρώτο βήμα του αλγορίθμου είναι να βρεθούν όλα τα δυνατά events στο event log που πρέπει να επεξεργαστεί. Αυτό συμβαίνει στον **Κώδικα 6**. Συγκεκριμένα από όλα τα traces στο Dataset tracesDS (με την χρήση του **map** operator) ξεχωρίζονται τα events τα οποία αποτελούν το trace (**flatMap** operator) και και στην συνέχεια μοναδικοποιούνται (toSet μέθοδος).

|  |
| --- |
| **//Κώδικας 6**  **def** getAllEvents(tracesDS: Dataset[(String, List[String])]) : List[String] = {  **return** tracesDS  .map(x=>x.\_2)  .flatMap(x=>x.toSet)  .collect()  .toSet  .toList  .sorted } |

Το δεύτερο βήμα του αλγορίθμου είναι να κατασκευαστεί ένα σύνολο από όλα τα αρχικά events δηλαδή ένα σύνολο από τα αρχικά events όλων των traces. Αυτό συμβαίνει στον **κώδικα 7**.

|  |
| --- |
| **//Κώδικας 7**  **def** getStartActivities(tracesDS: Dataset[(String, List[String])]) : Set[String] = {  **return** tracesDS.map(x=>x.\_2.head).collect().toSet } |

Το τρίτο βήμα του αλγορίθμου είναι να κατασκευαστεί ένα σύνολο από όλα τα τελικά events δηλαδή ένα σύνολο από τα τελικά events όλων των traces. Αυτό συμβαίνει στον **κώδικα 8**.

|  |
| --- |
| **//Κώδικας 8**  **def** getFinalActivities(tracesDS: Dataset[(String, List[String])]) : Set[String] = {  **return** tracesDS.map(x=>x.\_2.last).collect().toSet } |

Στην συνέχεια ο Alpla Algorithm πρέπει να κατασκευάσει το **footprint graph** το οποίο είναι ένας πίνακας που περίεχει τις σχέσεις μεταξύ των events. Αυτό συμβαίνει στον **κώδικα 9**.

|  |
| --- |
| **//Κώδικας 9**  */\*\*  \* Step 4 Calculate pairs - Footprint graph  \* construct a list of pair events for which computations must be made  \** **@param tracesDS** *\** **@param events** *\** **@return** *\*/* **def** getFootprintGraph(tracesDS: Dataset[(String, List[String])], events: List[String]): Dataset[(Pair, String)] = {  **val** followRelation: FindFollowRelation = **new** FindFollowRelation()  **val** findLogRelations: FindLogRelations = **new** FindLogRelations()  **val** traceTools: TraceTools = **new** TraceTools()  **val** pairsToExamine = traceTools.constructPairsForComputationFromEvents(events)   */\*\*  \* pairInfo is in the following form  \* AB,PairNotation(DIRECT, FOLLOW)  \* AB,PairNotation(INVERSE, FOLLOW)  \*/* **val** pairInfo = tracesDS  .map(traces => followRelation.findFollowRelation(traces, pairsToExamine))  .map(x=>x.getPairsMap())  .flatMap(map=>map.toSeq) *//map to collection of tuples* .map(x=> *List*(**new** PairInfo((x.\_1, **new** PairNotation(x.\_2.\_1.pairNotation))), **new** PairInfo((x.\_1, **new** PairNotation(x.\_2.\_2.pairNotation)))))  .flatMap(x=>x.toSeq)   pairInfo.cache()   */\*\*  \* relations in the following form, Footprint graph  \* (FB,CAUSALITY)  \* (BB,NEVER\_FOLLOW)  \* (AB,PARALLELISM)  \*/* **val** logRelations = pairInfo  .groupByKey(x=> x.getPairName())  .mapGroups{**case**(k, iter) => (**new** Pair(k.getFirstMember().toString, k.getSecondMember().toString), iter.map(x => x.getPairNotation()).toSet)}  .map(x=>findLogRelations.getDirectAndInverseFollowRelations(x))  .map(x=>findLogRelations.extractFootPrintGraph(x.\_1, x.\_2, x.\_3))   pairInfo.unpersist()  **return** logRelations } |

Στον **κώδικα 9** φαίνεται ότι το πρώτο βήμα για την κατασκευή του footprint graph είναι η εύρεση των σχέσεων απευθείας διαδοχής (**direct succession**) μεταξύ των events των traces. Οι σχέσεις αυτές στον κώδικα είναι η μεταβλητή **pairInfo** η οποία είναι τύπου **Dataset[PairInfo]**. Το **PairInfo** είναι μια κλάση η οποία περιέχει ένα tuple στο οποίο το πρώτο στοιχείο είναι ένα ζεύγος από events (κλάση **Pair**) και το δεύτερο στοιχείο είναι μια άλλη κλάση η οποία δείχνει αν το ζεύγος από events είναι ευθή ή αντίστροφο καθώς και το είδος της σχέσης (κλάση **PairNotation**). Για παράδειγμα, έστω ότι η μεταβλητή **pairInfo** περιέχει τα ακόλουθα δεδομένα σε μορφή Tuples

( AB, PairNotation(DIRECT, FOLLOW) )  
( AB, PairNotation(INVERSE, FOLLOW) )

Από την παραπάνω αναπαράσταση για το ζεύγος event AB, γίνεται αντιληπτό ότι ισχύει Α>Β και Β>Α, καθώς για το ΑΒ το αντίστοιχο PairNotation με Directionality DIRECT έχει σχέση FOLLOW. Το ίδιο ακριβώς ισχύει και για δεύτερο Tuple που αναφέρεται στο Directionality INVERSE.

Για τον υπολογισμό της μεταβλητής **pairInfo** εκτελείται η διαδικάσια που φαίνεται στον κώδικα 9 και συγκεκριμένα πρώτα υπολογίζονται τα δυνατά ζεύγη που πρέπει να εξεταστούν. Ο υπολογισμός αυτών των ζευγών φαίνεται στο **κώδικα 10.** Συγκεκριμένα για όλα τα πιθανά events που υπάρχουν στο event log υπολογίζουμε όλα τα δυνατά ζεύγη τα οποία δεν μπορεί να έχουν ίδια και τα μέλη του ζεύγους και το αριστερό μέλος να μην είναι μεγαλύτερο από το δεξί μέλος του ζεύγους (συνθήκη **idxX == idxY || idxX < idxY**).

Έτσι για ένα πιθανό σύνολο από events (A,B,C,D,E) τα εξεταζόμενα ζεύγη θα είναι τα (ΑΑ, ΑΒ, ΑC, AD, BB, BC, BD, CC, CD, DD).

|  |
| --- |
| **//Κώδικας 10**  */\*\*  \* We assume that the events list contains no duplicates and they are sorted  \* If the events are A,B,C,D,E then pairs for computation are  \* AA, AB AC AD  \* BB BC BD  \* CC CD  \* DD  \** **@param events** *\** **@return** *\*/* **def** constructPairsForComputationFromEvents(events: List[String]): List[Pair] = {  **for** {  (x, idxX) <- events.zipWithIndex  (y, idxY) <- events.zipWithIndex  **if** (idxX == idxY || idxX < idxY)  } **yield new** Pair(x,y) } |

Στην συνέχεια τα ζεύγη αυτά τροφοδοτούνται σαν είσοδο στην κλάση FindFollowRelation η οποία για κάποιο trace υπολογίζει τις σχέσεις απευθείας διαδοχής (**direct succession / follow relation**) μεταξύ των events του trace.Αυτό φαίνεται στον **κώδικα 11**, όπου κατασκευάζεται ένα Map με όλα τα ζεύγη που υπολογίστηκαν στο προηγούμενο βήμα. Αυτό το Map περίεχει σαν key το ζεύγος από events και σαν value ένα Tuple της μορφής (PairNotation, PairNotation) όπου το πρώτο στοιχείο του Tuple περιέχει πληρφορία για την σχέση του DIRECT pair και το δεύτερο στοιχείο για την σχέση του INVERSE pair. Η μέθοδος **findFollowRelation** δέχεται σαν όρισμα μια παράμετρο **traceWithCaseId** η οποίαείναι της μορφής (String, List[String]). Για παράδειγμα μια τιμή για την παράμετρο αυτή θα μπορούσε να ήταν η (case1, ABCD). Η μέθοδος αυτή έχει σκοπό να βρει τις σχέσεις **FOLLOW** και **NOT\_FOLLOW** μεταξύ των events αυτού του trace (δηλαδή των A B C D) και να τις μετατρέψει στην μορφή (PairAB, (DIRECT, FOLLOW/NOT\_FOLLOW), (INVERSE, FOLLOW/NOT\_FOLLOW)).

|  |
| --- |
| **//Κώδικας 11****def** findFollowRelation(traceWithCaseId: (String, List[String]), pairsToExamine: List[Pair]):FullPairsInfoMap = {  **var** pairInfoMap = pairInfoInit(pairsToExamine)   *Range*(0, traceWithCaseId.\_2.length-1)  .map(i=> **new** Pair(traceWithCaseId.\_2(i), traceWithCaseId.\_2(i+1)))  .map(pair=> {  **val** pairExists : Boolean = checkIfPairExists(pair, pairInfoMap)  pairInfoMap = matchPairExists(pairExists, pair, pairInfoMap)  }  )   **return new** FullPairsInfoMap(pairInfoMap) } |

Με τον τρόπο που αναλύθηκε παραπάνω η μεταβλητή **pairInfo** πλέον διαθέτει όλες τις πληροφορίες για τις σχέσεις FOLLOW και NOT\_FOLLOW για όλα τα ζεύγη events που μπορούν να προκύψουν για το event log που έχει δωθεί σαν είσοδο στον **Alpha Algorithm**. Η μεταβλητή **pairInfo** επειδή θα χρησιμοποιηθεί και σε επόμενο στάδιο για τον υπολογισμό του footprint graph, αποθηκεύεται στην μνήμη κάθε node του cluster με την εντολή **pairInfo.cache()** και όταν πλέον δεν θα είναι απαραίτητη θα διαγραφεί από την μνήμη με την εντολή **pairInfo.unpersist()**.

Ο υπολογισμός του footprint graph γίνεται στην μεταβλητή **logRelations** (βρίσκεται στον **κώδικα 9**). Σκοπός είναι για τα διαθέσιμα ζεύγη από την μεταβλητή **pairInfo** να βρεθούν οι υπόλοιπες σχέσεις που πρέπει να υπάρχουν στο footprint graph όπως οι **Causality (x->y)**, **Parallel (x||y)** και **Choice** (**x#y**). Για τον σκοπό αυτό έχει δημιουργηθεί η κλάση **FindLogRelations**, η οποία αποτελείται από δύο μεθόδους

* Την **getDirectAndInverseFollowRelations**: η οποία έχοντας σαν είσοδο ένα Τuple τύπου (Pair, Set[PairNotation])υπολογίζει αν για το ζεύγος αυτό ισχύει η σχέση DIRECT FOLLOW (δηλαδή a>b) και η σχέση INVERSE FOLLOW (δηλαδή b>a). Για παράδειγμα η είσοδος αυτής της μεθόδου θα μπορούσε να ήταν η (ΑΒ, Set[PairNotation(DIRECT, FOLLOW) , PairNotation(INVERSE, FOLLOW)]) και η έξοδος θα ήταν δύο λίστες με όνομα **directFollow** και **inverseFollow** αντίστοιχα που θα περιέχουν τις σχέσεις αυτές (**κώδικας 12**).
* Την **extractFootPrintGraph**: η οποία δέχεται σαν παραμέτρους το αποτέλεσμα της προηγούμενης μεθόδου και ψάχνει για τις παρακάτω σχέσεις αν υπάρχουν.
* a->b iff a>b && !(b>a) (Relation.CAUSALITY)
* a||b iff a>b && b>a (Relation.PARALLEL)
* a#b iff !(a>b)&& !(b>a) (Relation.NEVER\_FOLLOW)

Συγκεκριμένα όπως φαίνεται στον **κώδικα 12**, ελέγχει τις λίστες **directFollow** και **inverseFollow** και αν και οι δύο δεν είναι άδειες τότε η σχέση μεταξύ των δύο **events** είναι παραλληλία (Relation.PARALLEL), αν και οι δύο είναι άδειες τότε έχουν σχέση Relation.NEVER\_FOLLOW (ποτέ το ένα event δεν ακολουθεί το άλλο και αντίστροφα) και αν η μία από τις δύο δεν είναι άδεια τότε η σχέση των events είναι Relation.CAUSALITY.

|  |
| --- |
| **//Κώδικας 12**  **def** getDirectAndInverseFollowRelations(pairInfo: (Pair, Set[PairNotation])): ((Pair, Set[PairNotation]), List[PairNotation], List[PairNotation]) = {  **val** directFollow = pairInfo.\_2.toSeq  .filter(x=> x.getDirectionality()==Directionality.*DIRECT* && x.getRelation()==Relation.*FOLLOW*)  .toList   **val** inverseFollow = pairInfo.\_2.toSeq  .filter(x=> x.getDirectionality()==Directionality.*INVERSE* && x.getRelation()==Relation.*FOLLOW*)  .toList   (pairInfo, directFollow, inverseFollow) }  **def** extractFootPrintGraph(pairInfo: (Pair, Set[PairNotation]), directFollow: List[PairNotation], inverseFollow: List[PairNotation]): (Pair, String) = {  **if** (directFollow.nonEmpty && inverseFollow.nonEmpty) {  (pairInfo.\_1, Relation.*PARALLELISM*.toString)  } **else if** (directFollow.nonEmpty && inverseFollow.isEmpty) {  (pairInfo.\_1, Relation.*CAUSALITY*.toString)  } **else if** (directFollow.isEmpty && inverseFollow.nonEmpty) {  (**new** Pair(pairInfo.\_1.member2, pairInfo.\_1.member1), Relation.*CAUSALITY*.toString)  } **else if** (directFollow.isEmpty && inverseFollow.isEmpty) {  (pairInfo.\_1, Relation.*NEVER\_FOLLOW*.toString)  } **else** {  *//this never must happen. Default case* (pairInfo.\_1, Relation.*FOLLOW*.toString)  } } |

Μετά τον υπολογισμό του footprint graph, ακολουθεί το επόμενο τμήμα του τέταρτου βήματος του Alpha algorithm, στο οποίο πρέπει να υπολογιστούν τα causal groups. Αυτό υλοποιείται με την κλάση **FindCausalGroups** η οποία δέχεται σαν όρισμα το footprint graph με την μεταβλητή **logRelations**. Η μεταβλητή αυτή είναι της μορφής Dataset[(Pair, String)] και ένα παράδειγμα των δεδομένων που θα μπορούσε να περιέχει είναι το παρακάτω

(Pair(A, D), CAUSALITY)  
(Pair(Α, C), CAUSALITY)  
(Pair(B, D), CAUSALITY)  
(Pair(B, C), CAUSALITY)

(Pair(A, B), NEVER\_FOLLOW)

(Pair(C, D), NEVER\_FOLLOW)

Σκοπός είναι από τα παραπάνω δεδομένα να βρεθούν δύο σύνολα από events έστω (Q,R) για τα οποία κάθε στοιχείο του Q θα ενώνεται με οποιοδήποτε άλλο στοιχείο του R με Causal σχέση, ενώ παράλληλα τα events των Q και R μεταξύ τους θα έχουν Never Follow σχέση. Για την εύρεση των causal groups ακολουθήθηκαν τα παρακάτω βήματα

1. Εύρεση όλων των μοναδικών events από κάθε πλευρά όλων των causal σχέσεων. Για το παραπάνω παράδειγμα πρέπει να υπολογιστούν δύο σύνολα, ένα με τα event {a,b} και ένα δεύτερο με τα {c,d}.
2. Εύρεση όλων των δυνατών συνδιασμών για κάθε ένα από τα παραπάνω σύνολα. Δηλαδή για το συγκεκριμένο παράδειγμα πρέπει να υπολογιστούν οι δύο λίστες List[{a,b} {a} {b}] and List[{c,d} {c} {d}] .
3. Διαγραφή από τις δύο παραπάνω λίστες των συνόλων αυτών τα οποία έχουν έστω μια μη-NeverFollow σχέση. Στο συγκεκριμένο παράδειγμα δεν θα γίνει καμία διαγραφή συνόλου.
4. Σύνδεση όλων των συνόλων από τις δύο λιστές μόνο αν όλα τα στοιχεία των δύο λιστών είναι σε causal σχέση μεταξύ τους και σχηματισμός των causal groups.

Στον κώδικα 13 φαίνονται τα παραπάνω βήματα για τον υπολογισμό των causal groups.

|  |
| --- |
| **//Κώδικας 13**  **def** extractCausalGroups():Dataset[CausalGroup[String]] = {  **val** directCausalRelations = logRelations  .filter(x=>x.\_2==Relation.*CAUSALITY*.toString)  .map(x=>x.\_1)  .map(x=>(x.member1, x.member2))   *//1) Find all events from both sides (left and right) {a,b} and {c,d}* **val** uniqueEventsFromLeftSideEvents = directCausalRelations  .map(causal => (**"left"**, causal.\_1))  .groupByKey(x=>x.\_1)  .mapGroups{**case**(k, iter) => (k, iter.map(x => x.\_2).toSet)}   **val** uniqueEventsFromRightSideEvents = directCausalRelations  .map(causal => (**"right"**, causal.\_2))  .groupByKey(x=>x.\_1)  .mapGroups{**case**(k, iter) => (k, iter.map(x => x.\_2).toSet)}   *//2) Find all possible combination for each set (left and right) List[{a,b} {a} {b}] and List[{c,d} {c} {d}]  //3) Delete from above lists all the sets which are not in never follow relation with each other* **val** groups = uniqueEventsFromLeftSideEvents  .union(uniqueEventsFromRightSideEvents)  .flatMap(uniqueEvents => possibleSubsets(uniqueEvents))  .filter(subset=> allRelationsAreNeverFollow(subset.\_2))  .groupByKey(x=>x.\_1)  .mapGroups{**case**(k, iter) => (k, iter.map(x => x.\_2).toList)}   **import** *spark*.implicits.\_  **val** leftGroups = groups.first()  **val** rigthGroups = groups.orderBy(**$"value"**.desc).first()   *//4) Connect all the sets (as causal groups) from two lists and keep only those for which all events are in causal relations* **val** causalGroups = leftGroups.\_2.toDS().as(**"left"**)  .crossJoin(rigthGroups.\_2.toDS().as(**"right"**))  .where(**$"left.value"** !== **$"right.value"**)  .toDF(**"left"**, **"right"**)  .filter(group=>isCausalRelationValid(group.getAs[Seq[String]](**"left"**).toSet, group.getAs[Seq[String]](**"right"**).toSet))  .map(group=>**new** CausalGroup(group.getAs[Seq[String]](**"left"**).toSet,group.getAs[Seq[String]](**"right"**).toSet))   causalGroups } |

Για να βρεθεί αν ένα ζεύγος από events ανήκει σε Never Follow ή Causal σχέση χρησιμοποιήθηκαν broadcast μεταβλητές οι οποίες είναι κοινόχρηστες μεταβλητές (shared variables) και χρησιμοποιούνται για να επιτρέψουν στον προγραμματιστή να κρατήσει ένα read-only αντίγραφο των δεδομένων σε κάθε node του cluster αντί να στέλνει τα δεδομένα μαζί με τα διάφορα task που πρέπει να εκτελεστούν. Έτσι με αυτόν τον τρόπο μπορεί κάθε node να ελέγξει πολύ γρήγορα αν ένα ζεύγος από events ανήκει σε κάποια από τις δύο σχέσεις. Παράδειγμα η μέθοδος **allRelationsAreNeverFollow** στον **Κώδικας 14** που ελέγχει αν όλα τα πιθανά ζεύγη ενός σύνολου από events ανήκουν στην Never Follow σχέση χρησιμοποιώντας broadcast variables. Ίδια ακριβώς χρήση των broadcast variables κάνει και η μέθοδος **allEventsAreInCausalityRelation** για να ελέγξει αν όλα τα στοιχεία δύο συνόλων από events είναι σε causal σχέση μεταξύ τους.

|  |
| --- |
| **//Κώδικας 14**  **val** *never* = logRelations  .filter(x=>x.\_2==Relation.*NEVER\_FOLLOW*.toString)  .map(neverRelation => neverRelation.\_1)  .distinct()  .collect()  .toList  **val** *neverBc* = *spark*.sparkContext.broadcast(*never*)  **val** *causal* = logRelations  .filter(x=>x.\_2==Relation.*CAUSALITY*.toString)  .map(neverRelation => neverRelation.\_1)  .distinct()  .collect()  .toList  **val** *causalBc* = *spark*.sparkContext.broadcast(*causal*)  **def** allRelationsAreNeverFollow(possibleGroup: Set[String]): Boolean = {  **val** allPossiblePairs = **for** {  (x, idxX) <- possibleGroup.zipWithIndex  (y, idxY) <- possibleGroup.zipWithIndex  **if** idxX < idxY  } **yield new** Pair(x,y)  **val** notNeverFollow = allPossiblePairs  .find(x=> (!*neverBc*.value.contains(x) && !*neverBc*.value.contains(createInversePair(x))))   notNeverFollow.isEmpty }  **def** allEventsAreInCausalityRelation(groupA: Set[String], groupB: Set[String]): Boolean = {  **val** pairs = **for** {  eventA <- groupA  eventB <- groupB  } **yield new** Pair(eventA, eventB)   **for** {  pair <- pairs  } **yield if** (!*causalBc*.value.contains(pair)) { **return false** }   **true** } |

Το πέμπτο βήμα του Alpha Algorithm είναι ο υπολογισμός μόνο των maximal causal groups. Δηλαδή από τα causal groups που υπολογίστηκαν στο προηγούμενο βήμα πρέπει να διατηρηθούν για την συνέχεια της εκτέλεσης του αλγορίθμου μόνο αυτά τα causal groups που είναι maximal, δηλαδή causal groups που δεν συμπεριλαμβάνονται σε άλλα. Για παράδειγμα έστω ότι υπάρχουν 2 causal groups ({a} , {b}) και ({a} , {b,c}). Από αυτά τα causal groups μόνο το δεύτερο ({a} , {b,c}) πρέπει να διατηρηθεί γιατί είναι maximal καθώς περιέχει και το ({a} , {b}). Αυτό το βήμα του **Alpha Algorithm** εκτελείται στην κλάση **FindMaximalPairs** (**Κώδικας 15**). Συγκεκριμένα γίνεται χρήση των **accumulators** του Spark αλλά και των broadcast variables. Οι **accumulators** είναι μεταβλητές που μπορούν να χρησιμοποιηθούν σαν containers για την συσσώρευση μερικών αποτελεσμάτων που εκτελούνται σε διαφορετικούς executors του Spark.

Η διαδικασία εύρεσης των maximal groups είναι αρκετά απλή και για κάθε causal group στο αρχικό σύνολο ελέγχει αν υπάρχει κάποιο άλλο group που να είναι υποσύνολο του (**isSubsetOf** μέθοδος). Αν δεν βρεθεί κανένα τότε το causal group προστίθεται στον accumulator (maximalCausalGroups.add(group)).

|  |
| --- |
| **//Κώδικας 15**  **val** *spark* = SparkSession.*builder*().getOrCreate() **val** *causalGroupsList* = causalGroups  .distinct()  .collect()  .toList  **val** *causalGroupsListBc* = *spark*.sparkContext.broadcast(*causalGroupsList*) **def** extract(): List[CausalGroup[String]] = {  **val** maximalCausalGroups : CollectionAccumulator[CausalGroup[String]] = *spark*.sparkContext.collectionAccumulator(**"maximalCausalGroups"**)   causalGroups  .foreach(group => **if** (toBeRetained(group)) {  maximalCausalGroups.add(group)  })   maximalCausalGroups.value.asScala.toList }  **def** toBeRetained(toCheck: CausalGroup[String]): Boolean = {  !*causalGroupsListBc*.value.exists(x => x != toCheck && isSubsetOf(toCheck, x)) }  */\*\*  \* If true, the group2 must be retained  \** **@param group1** *\** **@param group2** *\** **@return** *\*/* **def** isSubsetOf(group1 : (CausalGroup[String]), group2 : CausalGroup[String]) : Boolean = {  **return** group1.getFirstGroup().subsetOf(group2.getFirstGroup()) && group1.getSecondGroup().subsetOf(group2.getSecondGroup()) } |

Το έκτο βήμα του αλγορίθμου είναι η εύρεση των states/places του παραγόμενου από τον **Alpha Algorithm** PetriNet. Η αναπαράσταση ενός state του Petri Net γίνεται με την κλάση State η οποία διαθέτει δύο παραμέτρους όπως φαίνεται στην αρχή του **κώδικα 16**. Η μία παράμετρος είναι η input και είναι ένα σύνολο από δεδομένα τύπου String, καθώς ένα state μπορεί να έχει σαν είσοδο παραπάνω από ένα events. Ομοίως και για την έξοδο του State. Έτσι λοιπόν τα maximal groups που υπολογίστηκαν στο προηγούμενο βήμα μετατρέπονται σε states του Petri Net, όπως φαίνεται στον **κώδικα 16**. Ακόμα δημιουργούνται και δύο επιπλέον states, ένα το αρχικό state και ένα το τελικό state του Petri Net. Οι κλάση Places συνοψίζει τα places του Petri Net σε ένα tuple με τρία στοιχεία.

|  |
| --- |
| **//Κώδικας 16**  @SerialVersionUID(100L) **class** State(**val** input: Set[String], **val** output: Set[String]) **extends** Serializable {}  @SerialVersionUID(100L) **class** Places(**val** places: (State, State, List[State])) **extends** Serializable {}  **def** getPlaces(maximalGroups : List[CausalGroup[String]], startActivities : Set[String], finalActivities : Set[String]): Places = {  **val** states = maximalGroups  .map(x=> **new** State(x.getFirstGroup(), x.getSecondGroup()))   **val** initialState = **new** State(*Set*.empty, startActivities)  **val** finalState = **new** State(finalActivities, *Set*.empty)   **new** Places(initialState, finalState, states) } |

Το έβδομο βήμα του **Alpha Algorithm** είναι ο υπολογισμός των τόξων (arcs) του Petri Net. Η υλοποίηση αυτού του βήματος γίνεται στην κλάση FindEdges, όπως φαίνεται στον **κώδικα 17**. Η κλάση αυτή δέχεται σαν παράμετρο ένα στιγμιότυπο της κλάσης Places το οποίο υπολογίστηκε στο προηγούμενο βήμα. Σκοπός της είναι να παράγει μια λίστα από στιγμιότυπα της κλάσης Edge, που θα αναπαριστούν τα edges του γραφήματος Petri Net. Κάθε στιγμιότυπο της κλάσης Edge αποτελείται από τρεις παραμέτρους ένα event, ένα State και μια Boolean μεταβλητή που δείχνει ποιο από τα δύο είναι η αρχή και ποιο το πέρας του τόξου (Edge). Αν η Boolean τιμή είναι true, τότε το event είναι η αρχή του τόξου και το state το πέρας. Αν είναι false, τότε συμβαίνει το αντίθετο. Η κατασκευή των edges γίνεται στην μέθοδο **constructEdgesFromStates** η οποία δέχεται σαν είσοδο ένα State και μετατρέπει τις εισόδους του State σε στιγμιότυπο της κλάσης Edge με boolean true (καθώς το state είναι το πέρας του edge) και τις εξόδους του State σε στιγμιότυπο της κλάσης Edge με boolean false (καθώς το state είναι η αρχή του edge)). Ακόμα κατασκευάζει τα στιγμιότυπα της κλάσης Edge που ξεκινάνε από το αρχικό State με την μέθοδο **getInitialEdges** και τα στιγμιότυπα της κλάσης Edge που καταλήγουν στο τελικό State με την μέθοδο **getFinalEdges**.

|  |
| --- |
| **//Κώδικας 17**  */\*\*  \* Direct == true -> (event, State)  \* Inverse == false -> (State, event)**\*/* @SerialVersionUID(100L) **class** Edge(**val** event: String, **val** state: State, **val** direct: Boolean) **extends** Serializable {}  @SerialVersionUID(100L) **class** FindEdges(**val** places: Places) **extends** Serializable {   **def** find():List[Edge] = {  **val** edges = places.getStates()  .flatMap(x=>constructEdgesFromStates(x))   **return** edges ::: getInitialEdges() ::: getFinalEdges()  }   **def** constructEdgesFromStates(state : State) : List[Edge] = {  **val** inputEdges = state.getInput()  .map(x=> **new** Edge(x, state, **true**))  .toList   **val** outputEdges = state.getOutput()  .map(x=> **new** Edge(x, state, **false**))  .toList   **return** inputEdges ::: outputEdges  }   **def** getInitialEdges(): List[Edge] = {  **return** places.getInitialState().getOutput()  .map(x=> **new** Edge(x, places.getInitialState(), **false**))  .toList  }   **def** getFinalEdges(): List[Edge] = {  **return** places.getFinalState().getInput()  .map(x=> **new** Edge(x, places.getFinalState(), **true**))  .toList  }  } |

Το όγδωο και τελευταίο βήμα του **Alpha Algorithm** είναι η κατασκευή του τελικού Petri Net, που είναι η δημιουργία ενός στιγμιοτύπου της κλάσης **PetriNet** (**κώδικας 18**), η οποία διαθέτει τρεις παραμέτρους μια για τα Places, μια για τα events και μία για τα Edges του Petri Net.

|  |
| --- |
| **//Κώδικας 18**  @SerialVersionUID(100L) **class** PetriNet (**val** places: Places, **val** events: List[String], **val** edges: List[Edge]) **extends** Serializable {} |

Ο ίδιος ακριβώς αλγόριθμος υλοποιήθηκε και δεύτερη φορά χωρίς την χρήση **Spark** βιβλιοθηκών ώστε να γίνει σύγκριση στο ίδιο περιβάλλον εκτέλεσης του Alpha Algorithm σε **Spark** περιβάλλον αλλά και σε ένα περιβάλλον χωρίς την χρήση του **Spark**.

# 5.2 Επιβεβαίωση Ορθής Λειτουργίας του Κώδικα Υλοποίησης

Για την επαλήθευση της ορθής λειτουργίας του υλοποιημένου Alpha Algorithm έχουν γραφτεί συνολικά 34 unit tests, τα οποία ελέγχουν την λειτουργία του αλγορίθμου από την αρχή έως το τέλος (end to end) αλλά και τις κάθε κλάσης ξεχωριστά. Αυτό επιτρέπει την αλλαγή της υλοποίησης σε διάφορα component του αλγορίθμου γνωρίζοντας ότι πάντα η συνολική λειτουργικότητα του αλγορίθμου μένει ανεπηρέαστη.

Για τον end to end έλεγχο χρησιμοποιήθηκαν 5 παραδείγματα που είναι γνωστό εκ των προτέρων το Petri Net το οποίο πρέπει να παραχθεί. Αντίστοιχα έγινε η υλοποίηση των αντίστοιχων unit tests τα οποία περιμένουν το ίδιο ακριβώς Petri Net να παραχθεί και από την **Spark** υλοποίηση του Alpha Algorithm. Παρακάτω παροσιάζονται σύντομα τα 5 παραδείγματα και τα αντίστοιχα unit test.

**Παράδειγμα 1**

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img37.png  **Εικόνα 37:**  Το Petri Net παράγεται από το event log w = {ABD , ACD} |

Στον **κώδικα 19** φαίνεται το αντίστοιχο Unit test σε Scala που υλοποιήθηκε και επαληθεύει την ορθή λειτουργία του υλοποιημένου Alpha Algorithm. Συγκεκριμένα από τον παρακάτω κώδικα βλέπουμε ότι υπάρχουν 4 διαφορετικά events (A,B,C,D) όσα ακριβώς υπάρχουν και στο Petri Net. Ακόμα βλέπουμε ότι ο αριθμός των Edges στο Petri Net είναι 8, όσα ακριβώς Edges βρέθηκαν και από τον υλοποιημένο Alpha Algorithm (τελευταίο assertion). Ακόμα στο Petri Net είναι φανερό ότι υπάρχουν τέσσερα states (κύκλοι πάνω στο γράφημα). Ακριβώς τα ίδια state έχουν βρεθεί και από τον υλοποιημένο Alpha Algorithm και είναι τα ακόλουθα

* State(*Set*.empty, *Set*(**"A"**)) : Το αρχικό state του Petri Net (start) που δεν έχει είσοδο και σαν έξοδο έχει το event A.
* State(*Set*(**"A"**), *Set*(**"B"**, **"C"**)) : Το δεύτερο state του Petri Net που έχει είσοδο το event A και σαν έξοδο έχει τα events B και C.
* State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)): Το τρίτο state του Petri Net που έχει είσοδο τα events B και C και σαν έξοδο έχει το event D.
* State(*Set*(**"D"**), *Set*.empty): Το τέταρτο state του Petri Net είναι το τελικό και έχει σαν είσοδο το event D ενώ δεν έχει καμία έξοδο.

Ακόμα από το Petri Net και το unit test βλέπουμε ότι ο υλοποιημένος Alpha Algorithm έχει βρει και τις σωστές ακμές (Edges) οι οποίες είναι 8 και είναι οι ακόλουθες

* Edge(**"A"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"C"**)), **true**)) : Ενώνει το event A με το δεύτερο state του Petri Net. Το boolean flag είναι true γιατί το event είναι η αρχή της ακμής και το state είναι το πέρας της.
* Edge(**"B"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"C"**)), **false**)) : Ενώνει το δεύτερο state του Petri Net με το event Β. Το boolean flag είναι false γιατί το state είναι η αρχή της ακμής και το event είναι το πέρας της.
* Edge(**"C"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"C"**)), **false**)) : Ενώνει το δεύτερο state του Petri Net με το event C. Το boolean flag είναι false γιατί το state είναι η αρχή της ακμής και το event είναι το πέρας της.
* Edge(**"B"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **true**)) : Ενώνει το event B με το τρίτο state του Petri Net. Το boolean flag είναι true γιατί το event είναι η αρχή της ακμής και το state είναι το πέρας της.
* Edge(**"C"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **true**)) : Ενώνει το event C με το τρίτο state του Petri Net. Το boolean flag είναι true γιατί το event είναι η αρχή της ακμής και το state είναι το πέρας της.
* Edge(**"D"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **false**)) : Ενώνει το τρίτο state του Petri Net με το event D. Το boolean flag είναι false γιατί το state είναι η αρχή της ακμής και το event είναι το πέρας της.
* Edge(**"A"**, **new** State(*Set*.empty, *Set*(**"A"**)), **false**)) : Ενώνει το πρώτο state του Petri Net με το event Α. Το boolean flag είναι false γιατί το state είναι η αρχή της ακμής και το event είναι το πέρας της.
* Edge(**"D"**, **new** State(*Set*(**"D"**), *Set*.empty), **true**)) : Ενώνει το event D με το τελευταίο state του Petri Net. Το boolean flag είναι true γιατί το event είναι η αρχή της ακμής και το state είναι το πέρας της.

Ακόμα στον κώδικα 19 υπάρχουν και δύο assertions τα οποία ελέγχουν συνθήκες οι οποίες δεν πρέπει να ισχύουν.

|  |
| --- |
| **//Κώδικας 19**  test(**"Check Alpha Algorithm functionality - Log 2"**) {  **val** logPath = **"src/test/resources/log2.txt"  val** traceTools: TraceTools = **new** TraceTools()  **val** tracesDS : Dataset[(String, List[String])] = traceTools.tracesDSFromLogFile(logPath)  **val** petriNet = AlphaAlgorithm.*executeAlphaAlgorithm*(tracesDS)   *//check edges* assert(petriNet.getEdges().contains(**new** Edge(**"A"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"C"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"C"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"C"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"C"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"C"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **false**)))   *//check initial edges* assert(petriNet.getEdges().contains(**new** Edge(**"A"**, **new** State(*Set*.empty, *Set*(**"A"**)), **false**)))   *//check final edges* assert(petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"D"**), *Set*.empty), **true**)))   *//check wrong directionality* assert(!petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **false**)))  assert(!petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **true**)))   assert(petriNet.getEdges().size==8) } |

**Παράδειγμα 2**

|  |
| --- |
| **C:\Users\Vlasis\Dropbox\osyl\report\images\img38.png**  **Εικόνα 38:**  Το Petri Net παράγεται από το event log w = {ABCEFGH , ABCEGFH, ADEGFH, ADEFGH } |

Ο κώδικας 20 ελέγχει την ορθή λειτουργία του **Alpha Algorithm** για event log w = {ABCEFGH , ABCEGFH, ADEGFH, ADEFGH }. Παρατηρούμε, ακριβώς με την ίδια ανάλυση του παραδείγματος 1, ότι ο υλοποιημένος **Alpha Algorithm** κατασκευάζει το σωστό Petri Net με ακριβώς 18 ακμές και 9 states.

|  |
| --- |
| **//Κώδικας 20**  test(**"Check Alpha Algorithm functionality - Log 5"**) {  **val** logPath = **"src/test/resources/log5.txt"  val** traceTools: TraceTools = **new** TraceTools()  **val** tracesDS : Dataset[(String, List[String])] = traceTools.tracesDSFromLogFile(logPath)  **val** petriNet = AlphaAlgorithm.*executeAlphaAlgorithm*(tracesDS)   *//check edges* assert(petriNet.getEdges().contains(**new** Edge(**"A"**, **new** State(*Set*(**"A"**), *Set*(**"B"**,**"D"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"A"**), *Set*(**"B"**,**"D"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"A"**), *Set*(**"B"**,**"D"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"B"**), *Set*(**"C"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"C"**, **new** State(*Set*(**"B"**), *Set*(**"C"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"C"**, **new** State(*Set*(**"C"**,**"D"**), *Set*(**"E"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"C"**,**"D"**), *Set*(**"E"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"E"**, **new** State(*Set*(**"C"**,**"D"**), *Set*(**"E"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"E"**, **new** State(*Set*(**"E"**), *Set*(**"F"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"F"**, **new** State(*Set*(**"E"**), *Set*(**"F"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"E"**, **new** State(*Set*(**"E"**), *Set*(**"G"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"G"**, **new** State(*Set*(**"E"**), *Set*(**"G"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"F"**, **new** State(*Set*(**"F"**), *Set*(**"H"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"H"**, **new** State(*Set*(**"F"**), *Set*(**"H"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"G"**, **new** State(*Set*(**"G"**), *Set*(**"H"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"H"**, **new** State(*Set*(**"G"**), *Set*(**"H"**)), **false**)))   *//check initial edges* assert(petriNet.getEdges().contains(**new** Edge(**"A"**, **new** State(*Set*.empty, *Set*(**"A"**)), **false**)))   *//check final edges* assert(petriNet.getEdges().contains(**new** Edge(**"H"**, **new** State(*Set*(**"H"**), *Set*.empty), **true**)))   *//check wrong directionality* assert(!petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **false**)))  assert(!petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **true**)))   assert(petriNet.getEdges().size==18) } |

**Παράδειγμα 3**

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img39.png  **Εικόνα 39:**  Το Petri Net παράγεται από το event log w = {ABDEFH , ACGH, ABEDFH } |

Ο κώδικας 21 ελέγχει την ορθή λειτουργία του **Alpha Algorithm** για event log w = {ABDEFH , ACGH, ABEDFH }. Παρατηρούμε, ακριβώς με την ίδια ανάλυση του παραδείγματος 1, ότι ο υλοποιημένος **Alpha Algorithm** κατασκευάζει το σωστό Petri Net με ακριβώς 18 ακμές και 9 states.

|  |
| --- |
| **//Κώδικας 21**  test(**"Check Alpha Algorithm functionality - Log 3"**) {  **val** logPath = **"src/test/resources/log3.txt"  val** traceTools: TraceTools = **new** TraceTools()  **val** tracesDS : Dataset[(String, List[String])] = traceTools.tracesDSFromLogFile(logPath)  **val** petriNet = AlphaAlgorithm.*executeAlphaAlgorithm*(tracesDS)   *//check edges* assert(petriNet.getEdges().contains(**new** Edge(**"A"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"C"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"C"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"C"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"C"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"C"**, **new** State(*Set*(**"C"**), *Set*(**"G"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"G"**, **new** State(*Set*(**"C"**), *Set*(**"G"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"B"**), *Set*(**"D"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"B"**), *Set*(**"D"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"D"**), *Set*(**"F"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"F"**, **new** State(*Set*(**"D"**), *Set*(**"F"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"B"**), *Set*(**"E"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"E"**, **new** State(*Set*(**"B"**), *Set*(**"E"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"E"**, **new** State(*Set*(**"E"**), *Set*(**"F"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"F"**, **new** State(*Set*(**"E"**), *Set*(**"F"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"F"**, **new** State(*Set*(**"F"**, **"G"**), *Set*(**"H"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"G"**, **new** State(*Set*(**"F"**, **"G"**), *Set*(**"H"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"H"**, **new** State(*Set*(**"F"**, **"G"**), *Set*(**"H"**)), **false**)))   *//check initial edges* assert(petriNet.getEdges().contains(**new** Edge(**"A"**, **new** State(*Set*.empty, *Set*(**"A"**)), **false**)))   *//check final edges* assert(petriNet.getEdges().contains(**new** Edge(**"H"**, **new** State(*Set*(**"H"**), *Set*.empty), **true**)))   *//check wrong directionality* assert(!petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **false**)))  assert(!petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **true**)))   assert(petriNet.getEdges().size==18) } |

**Παράδειγμα 4**

|  |
| --- |
| **C:\Users\Vlasis\Dropbox\osyl\report\images\img40.png**  **Εικόνα 40:**  Το Petri Net παράγεται από το event log w = {ABCEBCD , ABCD} |

Ο κώδικας 22 ελέγχει την ορθή λειτουργία του **Alpha Algorithm** για event log w = {ABCEBCD , ABCD}. Παρατηρούμε, ακριβώς με την ίδια ανάλυση του παραδείγματος 1, ότι ο υλοποιημένος **Alpha Algorithm** κατασκευάζει το σωστό Petri Net με ακριβώς 10 ακμές και 5 states και επιπλέον διαθέτει και μια επανάληψη (loop).

|  |
| --- |
| **//Κώδικας 22**  test(**"Check Alpha Algorithm functionality - Log 4"**) {  **val** logPath = **"src/test/resources/log4.txt"  val** traceTools: TraceTools = **new** TraceTools()  **val** tracesDS : Dataset[(String, List[String])] = traceTools.tracesDSFromLogFile(logPath)  **val** petriNet = AlphaAlgorithm.*executeAlphaAlgorithm*(tracesDS)   *//check edges* assert(petriNet.getEdges().contains(**new** Edge(**"A"**, **new** State(*Set*(**"A"**, **"E"**), *Set*(**"B"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"E"**, **new** State(*Set*(**"A"**, **"E"**), *Set*(**"B"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"A"**, **"E"**), *Set*(**"B"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"B"**), *Set*(**"C"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"C"**, **new** State(*Set*(**"B"**), *Set*(**"C"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"C"**, **new** State(*Set*(**"C"**), *Set*(**"D"**, **"E"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"C"**), *Set*(**"D"**, **"E"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"E"**, **new** State(*Set*(**"C"**), *Set*(**"D"**, **"E"**)), **false**)))   *//check initial edges* assert(petriNet.getEdges().contains(**new** Edge(**"A"**, **new** State(*Set*.empty, *Set*(**"A"**)), **false**)))   *//check final edges* assert(petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"D"**), *Set*.empty), **true**)))   *//check wrong directionality* assert(!petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **false**)))  assert(!petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **true**)))   assert(petriNet.getEdges().size==10) } |

**Παράδειγμα 5**

|  |
| --- |
| **C:\Users\Vlasis\Dropbox\osyl\report\images\img22.png**  **Εικόνα 41:**  Το Petri Net παράγεται από το event log w = {ABCD , ACBD, AED} |

Ο κώδικας 22 ελέγχει την ορθή λειτουργία του **Alpha Algorithm** για event log w = { ABCD , ACBD, AED}. Παρατηρούμε, ακριβώς με την ίδια ανάλυση του παραδείγματος 1, ότι ο υλοποιημένος **Alpha Algorithm** κατασκευάζει το σωστό Petri Net με ακριβώς 14 ακμές και 6 states και επιπλέον διαθέτει και μια επανάληψη (loop).

|  |
| --- |
| **//Κώδικας 23**  test(**"Check Alpha Algorithm functionality - Log 1"**) {  **val** logPath = **"src/test/resources/log1.txt"  val** traceTools: TraceTools = **new** TraceTools()  **val** tracesDS : Dataset[(String, List[String])] = traceTools.tracesDSFromLogFile(logPath)  **val** petriNet = AlphaAlgorithm.*executeAlphaAlgorithm*(tracesDS)   *//check edges* assert(petriNet.getEdges().contains(**new** Edge(**"A"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"E"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"E"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"E"**, **new** State(*Set*(**"A"**), *Set*(**"B"**, **"E"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"A"**, **new** State(*Set*(**"A"**), *Set*(**"C"**, **"E"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"C"**, **new** State(*Set*(**"A"**), *Set*(**"C"**, **"E"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"E"**, **new** State(*Set*(**"A"**), *Set*(**"C"**, **"E"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"B"**, **"E"**), *Set*(**"D"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"E"**, **new** State(*Set*(**"B"**, **"E"**), *Set*(**"D"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"B"**, **"E"**), *Set*(**"D"**)), **false**)))  assert(petriNet.getEdges().contains(**new** Edge(**"C"**, **new** State(*Set*(**"C"**, **"E"**), *Set*(**"D"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"E"**, **new** State(*Set*(**"C"**, **"E"**), *Set*(**"D"**)), **true**)))  assert(petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"C"**, **"E"**), *Set*(**"D"**)), **false**)))   *//check initial edges* assert(petriNet.getEdges().contains(**new** Edge(**"A"**, **new** State(*Set*.empty, *Set*(**"A"**)), **false**)))   *//check final edges* assert(petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"D"**), *Set*.empty), **true**)))   *//check wrong directionality* assert(!petriNet.getEdges().contains(**new** Edge(**"B"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **false**)))  assert(!petriNet.getEdges().contains(**new** Edge(**"D"**, **new** State(*Set*(**"B"**, **"C"**), *Set*(**"D"**)), **true**)))   assert(petriNet.getEdges().size==14) } |

# 6. Εκτέλεση πειραματικών μετρήσεων

# 6.1 Πλατφόρμα Databricks

Η **Databricks** είναι μια εταιρεία η οποία ιδρύθηκε από τους δημιουργούς του **Apache Spark** και σκοπεύει να βοηθήσει τους πελάτες που χρειάζονται ανάλυση Big Data βασιζόμενοι στο cloud χρησιμοποιώντας το **Spark**. Το **Databricks** είναι μια web πλατφόρμα που δουλεύει με το **Spark**, και παρέχει αυτόματη διαχείρηση ενός cluster και διάφορα notebooks, τα οποία παρέχουν πολύ εύκολη συγγραφή και εκτέλεση **Spark** κώδικα στο cluster που έχει δημιουργήσει ο χρήστης.

Για την μέτρηση και αξιολόγηση του **Alpha Algorithm** με την χρήση του Spark, αλλά και χωρίς αυτήν, χρησιμοποιήθηκε η **Community edition** του **Databricks**, η οποία παρέχει στον χρήστη ένα cluster με μόνο ένα **driver** (αλλά χωρίς **workers**) το οποίο διαθέτει 88 πυρήνες (cores) και 6 GB μνήμης RAM.

Για την εκτέλεση του **Alpha Algorithm** δημιουργήθηκαν στο **Databricks**, δύο notebooks. Το πρώτο notebook εκτελεί τον **Alpha Algorithm** στην **Spark** έκδοση του, ενώ το δεύτερο εκτελεί τον **Alpha Algorithm** χωρίς την χρήση **Spark** βιβλιοθηκών. Στις παρακάτω δύο εικόνες (42 έως 44) φαίνεται το **notebook** που δημιουργήθηκε στο **Databricks** για την εκτέλεση του **Alpha Algorithm**  με την χρήση **Spark** βιβλιοθηκών.

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img42.png  **Εικόνα 42:**  Notebook στο **Databricks** που τρέχει σε δύο commands την δημιουργία των traces που θα επεξεργαστεί ο Alpha Algorithm και το πρώτο βήμα του αλγορίθμου. |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img43.png  **Εικόνα 43:**  Notebook στο **Databricks** με τα βήματα 3 έως 5 του Alpha Algorithm. |

|  |
| --- |
| C:\Users\Vlasis\Dropbox\osyl\report\images\img44.png  **Εικόνα 44:**  Notebook στο **Databricks** με τα βήματα 6 έως 8 του Alpha Algorithm. |

# 6.2 Μετρήσεις

Στο κεφάλαιο αυτό θα γίνει παρουσίαση των μετρήσεων που έγιναν στο **Databricks**, τόσο στην **Community Edition** αλλά και στην κανονική έκδοση που παρέχει cluster με διαφορετικό αριθμό από nodes. Οι μετρήσεις έγιναν πάνω στο ίδιο dataset αλλά με διαφορετικές παραμέτρους κάθε φορά ώστε στον **Alpha Algorithm** να τροφοδοτείται διαφορετικός αριθμός από traces και διαφορετικός αριθμός συνολικών events. Σε κάθε πίνακα παρουσιάζονται οι τέσσερεις (4) παρακάτω χρόνοι

* Για την εκτέλεση της **non-Spark** έκδοσης του **Alpha Algorithm** στο **Databricks Community Edition** (1 driver χωρίς κανένα node-worker).
* Για την εκτέλεση της **Spark** έκδοσης του **Alpha Algorithm** στο **Databricks Community Edition** (1 driver χωρίς κανένα node-worker).
* Για την εκτέλεση της **Spark** έκδοσης του **Alpha Algorithm** στο **Databricks** με 1 driver και 2 worker-nodes.
* Για την εκτέλεση της **Spark** έκδοσης του **Alpha Algorithm** στο **Databricks** με 1 driver και 4 worker-nodes

Στις μετρήσεις αυτές φαίνεται ο συνολικός χρόνος που χρειάστηκε ο αλγόριθμος για να τρέξει, συμπεριλαμβανομένου και του χρόνου φιλτραρίσματος των δεδομένων. Σε όλους τους πίνακες ο Alpha Algorithm έτρεξε για όλα το event log με αρχικό αριθμό από traces 36094. Χρησιμοποιήθηκε η τεχνική του φιλτραρίσματος που περιγράφηκε σε προηγούμενο κεφάλαιο και διατηρήθηκαν μόνο τα traces εκείνα που έχουν ποσοστό εμφάνισης πάνω από την τιμή της μεταβλητής percentage. Συγκεκριμένα έγιναν οι παρακάτω μετρήσεις

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Πίνακας 1**  **percentage** = 2  **readAll** = true  **filtering** = true  **Initial number of traces** = 36094  **Number of traces to inspect** = 7  **Total number of events** = 16 | | | | |
|  | **Non-Spark** | **Spark (1 driver – 0 nodes)** | **Spark (1 driver – 2nodes)** | **Spark (1 driver – 4 nodes)** |
| **Time to execute Alpha Algorithm** | 2.60 *minutes* | 3.60 *minutes* | 1.54 *minutes* | 1.2 *minutes* |

Για τον πίνακα 1, διατηρήθηκαν μόνο τα traces εκείνα που έχουν ποσοστό εμφάνισης πάνω από 2%. Έτσι τελικά ο αλγόριθμος χρειάστηκε να επεξεργαστεί μόνο 7 traces καθώς τα υπόλοιπα επαναλαμβάνονται ή έχουν ποσοστό εμφάνισης κάτω του 2%. Σε αυτήν την περίπτωση υπήρξαν 16 διαφορετικά events. Παρατηρούμε ότι ο **non-Spark Alpha Algorithm** είναι πιο γρήγορος από τον **Spark Alpha Algorithm** όταν αυτός τρέχει σε ένα cluster με έναν **driver** αλλά χωρίς κανένα **worker**. Αυτό μπορεί να δικαιολογηθεί λόγω του ότι το dataset είναι αρκετά μικρό και έτσι ο **non-Spark Alpha Algorithm** μπορεί να εκτελέσει τον αλγόριθμο αρκετά γρήγορα αποφεύγοντας τον επιπλέον χρόνο που χρειάζεται ο **Spark Alpha Algorithm** για να ξεκινήσει την εκτέλεση ενός Spark προγράμματος αλλά και για το serialization και deserialization των δεδομένων. Όμως ακόμα και για αυτό το μικρό dataset ο **non-Spark Alpha Algorithm** είναι αρκετά πιο αργός από τον **Spark Alpha Algorithm** όταν αυτός τρέχει σε ένα cluster με έναν **driver** και 2 ή 4 **nodes**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Πίνακας 2**  **percentage** = 1  **readAll** = true  **filtering** = true  **Initial number of traces** = 36094  **Number of traces to inspect** = 8  **Total number of events** = 16 | | | | |
|  | **Non-Spark** | **Spark (1 driver – 0 nodes)** | **Spark (1 driver – 2nodes)** | **Spark (1 driver – 4 nodes)** |
| **Time to execute Alpha Algorithm** | 2.47 *minutes* | 3.50 *minutes* | * 1. *minutes* | *1.10 minutes* |

Για τον πίνακα 2, διατηρήθηκαν μόνο τα traces εκείνα που έχουν ποσοστό εμφάνισης πάνω από 1%. Έτσι τελικά ο αλγόριθμος χρειάστηκε να επεξεργαστεί μόνο 8 traces καθώς τα υπόλοιπα επαναλαμβάνονται ή έχουν ποσοστό εμφάνισης κάτω του 1%. Σε αυτήν την περίπτωση υπήρξαν 16 διαφορετικά events. Παρατηρούμε ότι ο **non-Spark Alpha Algorithm** είναι πιο γρήγορος από τον **Spark Alpha Algorithm** όταν αυτός τρέχει σε ένα cluster με έναν **driver** αλλά χωρίς κανένα **worker**. Αυτό μπορεί να δικαιολογηθεί λόγω του ότι το dataset είναι αρκετά μικρό και έτσι ο **non-Spark Alpha Algorithm** μπορεί να εκτελέσει τον αλγόριθμο αρκετά γρήγορα αποφεύγοντας τον επιπλέον χρόνο που χρειάζεται ο **Spark Alpha Algorithm** για να ξεκινήσει την εκτέλεση ενός Spark προγράμματος αλλά και για το serialization και deserialization των δεδομένων. Όμως ακόμα και για αυτό το μικρό dataset ο **non-Spark Alpha Algorithm** είναι αρκετά πιο αργός από τον **Spark Alpha Algorithm** όταν αυτός τρέχει σε ένα cluster με έναν **driver** και 2 ή 4 **nodes**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Πίνακας 3**  **percentage** = 0.8  **readAll** = true  **filtering** = true  **Initial number of traces** = 36094  **Number of traces to inspect** = 9  **Total number of events** = 17 | | | | |
|  | **Non-Spark** | **Spark (1 driver – 0 nodes)** | **Spark (1 driver – 2nodes)** | **Spark (1 driver – 4 nodes)** |
| **Time to execute Alpha Algorithm** | 10.35 *minutes* | 8.17 *minutes* | 6.07 *minutes* | 3.35 *minutes* |

Για τον πίνακα 3, διατηρήθηκαν μόνο τα traces εκείνα που έχουν ποσοστό εμφάνισης πάνω από 0.8%. Έτσι τελικά ο αλγόριθμος χρειάστηκε να επεξεργαστεί μόνο 9 traces καθώς τα υπόλοιπα επαναλαμβάνονται ή έχουν ποσοστό εμφάνισης κάτω του 0.8%. Σε αυτήν την περίπτωση υπήρξαν 17 διαφορετικά events. Παρατηρούμε ότι η **Spark** έκδοση του **Alpha Algorithm** είναι γρηγορότερη από την **non-Spark** έκδοση του **Alpha Algorithm** για όλες τις περιπτώσεις cluster που χρησιμοποιήθηκαν (0-2-4 **nodes**). Αυτό οφείλεται στο γεγονός ότι το dataset είναι πιο πολύπλοκο και υπάρχουν περισσότερα events που πρέπει να επεξεργαστούν και έτσι η ύπαρξη παραπάνω από ένα μηχανημάτα (**nodes**) παίζει καταλυτικό ρόλο στην μείωση του χρόνου εκτέλεσης του **Alpha Algorithm**. Έτσι παρατηρούμε την μεγάλη διαφορά των χρόνων εκτέλεσης για την **non-Spark** έκδοση του **Alpha Algorithm** έναντι του χρόνου εκτέλεσης για την **Spark** έκδοση του **Alpha Algorithm** σε ένα cluster με 4 **nodes** (10.35 λεπτά έναντι 3.35 λεπτά).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Πίνακας 4**  **percentage** = 0.7  **readAll** = true  **filtering** = true  **Initial number of traces** = 36094  **Number of traces to inspect** = 10  **Total number of events** = 18 | | | | |
|  | **Non-Spark** | **Spark (1 driver – 0 nodes)** | **Spark (1 driver – 2nodes)** | **Spark (1 driver – 4 nodes)** |
| **Time to execute Alpha Algorithm** | 34.60 *minutes* | 17.35 *minutes* | 15.40 *minutes* | 9.92 *minutes* |

Για τον πίνακα 4, διατηρήθηκαν μόνο τα traces εκείνα που έχουν ποσοστό εμφάνισης πάνω από 0.7%. Έτσι τελικά ο αλγόριθμος χρειάστηκε να επεξεργαστεί μόνο 10 traces καθώς τα υπόλοιπα επαναλαμβάνονται ή έχουν ποσοστό εμφάνισης κάτω του 0.7%. Σε αυτήν την περίπτωση υπήρξαν 18 διαφορετικά events. Παρατηρούμε ότι η **Spark** έκδοση του **Alpha Algorithm** είναι γρηγορότερη από την **non-Spark** έκδοση του **Alpha Algorithm** για όλες τις περιπτώσεις cluster που χρησιμοποιήθηκαν (0-2-4 **nodes**). Αυτό οφείλεται στο γεγονός ότι το dataset είναι πιο πολύπλοκο και υπάρχουν περισσότερα events που πρέπει να επεξεργαστούν και έτσι η ύπαρξη παραπάνω από ένα μηχανημάτα (**nodes**) παίζει σημαντικό ρόλο στην μείωση του χρόνου εκτέλεσης του **Alpha Algorithm**. Έτσι παρατηρούμε την μεγάλη διαφορά των χρόνων εκτέλεσης για την **non-Spark** έκδοση του **Alpha Algorithm** έναντι του χρόνου εκτέλεσης για την **Spark** έκδοση του **Alpha Algorithm** σε ένα cluster με 4 **nodes** (34.60 λεπτά έναντι 9.92 λεπτά).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Πίνακας 5**  **percentage** = 0.5  **readAll** = true  **filtering** = true  **Initial number of traces** = 36094  **Number of traces to inspect** = 12  **Total number of events** = 19 | | | | |
|  | **Non-Spark** | **Spark (1 driver – 0 nodes)** | **Spark (1 driver – 2nodes)** | **Spark (1 driver – 4 nodes)** |
| **Time to execute Alpha Algorithm** | 2.75 ***hours*** | 1.06 ***hours*** | 52.25 ***minutes*** | 43.58 ***minutes*** |

Για τον πίνακα 5, διατηρήθηκαν μόνο τα traces εκείνα που έχουν ποσοστό εμφάνισης πάνω από 0.5%. Έτσι τελικά ο αλγόριθμος χρειάστηκε να επεξεργαστεί μόνο 12 traces καθώς τα υπόλοιπα επαναλαμβάνονται ή έχουν ποσοστό εμφάνισης κάτω του 0.5%. Σε αυτήν την περίπτωση υπήρξαν 19 διαφορετικά events. Παρατηρούμε ότι η **Spark** έκδοση του **Alpha Algorithm** είναι γρηγορότερη από την **non-Spark** έκδοση του **Alpha Algorithm** για όλες τις περιπτώσεις cluster που χρησιμοποιήθηκαν (0-2-4 **nodes**). Αυτό οφείλεται στο γεγονός ότι το dataset είναι πιο πολύπλοκο και υπάρχουν περισσότερα events που πρέπει να επεξεργαστούν και έτσι η ύπαρξη παραπάνω από ένα μηχανημάτα (**nodes**) παίζει σημαντικό ρόλο στην μείωση του χρόνου εκτέλεσης του **Alpha Algorithm**. Έτσι παρατηρούμε την μεγάλη διαφορά των χρόνων εκτέλεσης για την **non-Spark** έκδοση του **Alpha Algorithm** έναντι του χρόνου εκτέλεσης για την **Spark** έκδοση του **Alpha Algorithm** σε ένα cluster με 4 **nodes** (2.75 ώρες έναντι 43.58 λεπτά).

Από τα πειράματα που έγιναν, στους πίνακες 1 και 2 φαίνεται ότι η **non-Spark** έκδοση του **Alpha Algorithm** είναι γρηγορότερη από την **Spark** έκδοση για ένα cluster με κανένα node. Αυτό συμβαίνει γιατί το **Databricks Community Edition** παρέχει cluster μόνο με ένα driver node χωρίς κανένα worker. Έτσι λόγω του ότι τα δεδομένα που θα επεξεργαστούν χωράνε στην κύρια μνήμη του συστήματος (RAM) και επειδή το setup που χρειάζεται για να ξεκινήσει η εκτέλεση ενός Spark προγράμματος είναι αρκετά μεγάλο σε σχέση με το να τρέξει το ίδιο ακριβώς πρόγραμμα σε καθαρή Scala, έχουμε σαν αποτέλεσμα η εκτέλεση της **non-Spark** υλοποίησης να είναι πιο γρήγορη. Φυσικά αυτό δεν ισχύει για την εκτέλεση της **Spark** έκδοσης για ένα cluster με 2 ή 4 nodes, καθώς εκεί το Spark εκμεταλλεύεται πλήρως τις δυνατότητες του cluster και έτσι η εκτέλεση του αλγορίθμου είναι πολύ γρηγορότερη σε σχέση με την **non-Spark** έκδοση του **Alpha Algorithm**.

Από τις μετρήσεις που έγιναν φάνηκε ότι το πιο χρονοβόρο σημείο του **Alpha Algorithm** είναι η εύρεση των causal groups. Η εύρεση των causal groups είναι μια πολύ βαριά διαδικασία γιατί το πρόγραμμα πρέπει να υπολογίσει πολλούς συνδιασμούς από σύνολα και υποσύνολα από events ενώ ταυτόχρονα πρέπει να ελέγχει αν τα ζεύγη μεταξύ των events ανηκούν σε συγκεκριμένες σχέσεις. Σε αυτό το στάδιο παρατηρούμε ότι η **Spark** υλοποίηση είναι πολύ πιο γρήγορη είτε τρέχει σε cluster χωρίς κανένα node είτε σε cluster με πολλά nodes.

# 7. Βιβλιογραφία

[1] Joerg Evermann (2016), “Scalable Process Discovery Using Map-Reduce”, IEEE Transactions on Services Computing

[2] Aalst, Wil M. P. & Weijters, A & Măruşter, Laura. (2004). Workflow Mining: Discovering Process Models from Event Logs, IEEE Transactions on Knowledge and Data Engineering.

[3] Aalst, van der, W. M. P. (2013).Decomposing Petri nets for process mining : a genericapproach.Distributed and Parallel Databases*.*

[4] J. Wang, R. K. Wong, J. Ding, Q. Guo and L. Wen, "Efficient Selection of Process Mining Algorithms," inIEEE Transactions on Services Computing.

[5] Petar Zecevic, Marko Bonaci, “Spark in Action”, Manning Publications CO

[6] Wil van der Aalst, “Process Mining: Data science in Action”, [www.coursera.org](http://www.coursera.org)

[7] Frank Kane, “Apache Spark 2 with Scala – Hands on with Big Data”, [www.udemy.com](http://www.udemy.com)

[8] Cluster overview, <https://spark.apache.org/docs/latest/cluster-overview.html>

[9] SQL programming guide, <https://spark.apache.org/docs/latest/sql-programming-guide.html>

[10] Petri Nets, <http://mlwiki.org/index.php/Petri_Nets>

[11] Alpha Algorithm, <http://mlwiki.org/index.php/Alpha_Algorithm>

[12] Spark RDDs, <https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-rdd.html>

[13] Spark RDDs, <https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm>

[14] Spark caching, <https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-rdd-caching.html>

[15] Apache Spark Executor, <https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-Executor.html>

[16] Apache Spark Driver, <https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-driver.html>

[17] Apache Spark Datasets, <https://databricks.com/spark/getting-started-with-apache-spark/datasets>

[18] Apache Spark Architecture, [https://www.dezyre.com/article/apache-spark-architecture-explained-in-detail/338](https://www.dezyre.com/article/apache-spark-architecture-explained-in-detail/338%20)

[19] Apach Spark Tuning, <https://spark.apache.org/docs/latest/tuning.html>

[20] Java Garbage collection for Spark applications, <https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html>

[21] Scala Programming Language, <https://en.wikipedia.org/wiki/Scala_(programming_language)>

[22] Databricks Community Edition, <https://community.cloud.databricks.com>