ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

«Το πρόβλημα χρονοπρογραμματισμού κατανεμημένων ροών εργασιών βάσει μεταθέσεων με περιορισμούς ημερομηνιών τερματισμού εργασιών»

«Distributed Permutation Flow-shop Scheduling Problem with Due Dates»

Κιοσσές Δημήτριος

ΑΜ:163

Επιβλέπων Καθηγητής: Γκόγκος Χρήστος

Άρτα, Ιανουάριος, 2025

Εγκρίθηκε από τριμελή εξεταστική επιτροπή

Άρτα, 29/01/2025

ΕΠΙΤΡΟΠΗ ΑΞΙΟΛΟΓΗΣΗΣ

1. Επιβλέπων Καθηγητής
   Γκόγκος Χρήστος

2. Μέλος επιτροπής
   Τζάλλας Αλέξανδρος

3. Μέλος επιτροπής
   Καρβέλης Πέτρος

Δήλωση μη λογοκλοπής

Δηλώνω υπεύθυνα και γνωρίζοντας τις κυρώσεις του Ν. 2121/1993 περί Πνευματικής Ιδιοκτησίας, ότι η παρούσα μεταπτυχιακή εργασία είναι εξ ολοκλήρου αποτέλεσμα δικής μου ερευνητικής εργασίας, δεν αποτελεί προϊόν αντιγραφής ούτε προέρχεται από ανάθεση σε τρίτους. Όλες οι πηγές που χρησιμοποιήθηκαν (κάθε είδους, μορφής και προέλευσης) για τη συγγραφή της περιλαμβάνονται στη βιβλιογραφία.

Κιοσσές Δημήτριος

Υπογραφή

ΠΕΡΙΛΗΨΗ

Ο χρονοπρογραμματισμός (scheduling, timetabling) είναι ένα ερευνητικό πεδίο με πολλές πρακτικές εφαρμογές στη βιομηχανία, στις μεταφορές, στον κατασκευαστικό κλάδο, στο λογισμικό των ηλεκτρονικών υπολογιστών και αλλού. Η δε σημασία της αποδοτικής επίλυσης προβλημάτων χρονοπρογραμματισμού έχει οδηγήσει στην ανάπτυξη πολλών τεχνικών αντιμετώπισης των σχετικών προβλημάτων.

Στην παρούσα διπλωματική θα εξεταστεί το γνωστό πρόβλημα χρονοπρογραμματισμού κατανεμημένων ροών εργασιών βάσει μεταθέσεων με περιορισμούς ημερομηνιών τερματισμού εργασιών (Distributed Permutation Flow-shop Scheduling Problem with Due Dates) και θα επιδιωχθεί η παραγωγή ανταγωνιστικών αποτελεσμάτων με τα State of the Art αποτελέσματα που εντοπίζονται στη βιβλιογραφία.

Η εργασία θα αναλύσει το πρόβλημα, θα μελετήσει τη βιβλιογραφία και θα εξετάσει διάφορες προσεγγίσεις επίλυσης του προβλήματος. Θα αναπτυχθεί κώδικας ικανός να πράξει ολοκληρωμένες λύσεις και θα χρησιμοποιηθούν οι απαραίτητες βιβλιοθήκες έτσι ώστε να επιτευχθούν λύσεις υψηλής ποιότητας για δημόσια διαθέσιμα προβλήματα.

Λέξεις-κλειδιά: *Χρονοπρογραμματισμός, Κατανεμημένος προγραμματισμός, Προγραμματισμός Ροής Παραγωγής, Βελτιστοποίηση Προγραμματισμού, Προθεσμίες εκτέλεσης, Ευρετικοί Αλγόριθμοι, Μεταευρετικοί Αλγόριθμοι, Γενετικοί Αλγόριθμοι, Υβριδικοί Αλγόριθμοι, Προγραμματισμός Παραγωγής, Τοπική Αναζήτηση*.

ABSTRACT

Scheduling (timetabling) is a research field with many practical applications in industry, transport, construction and computer software. The importance of efficiently solving scheduling problems has led to the development of many techniques for dealing with the related problems.

In this thesis, the well-known Distributed Permutation Flow-shop Scheduling Problem with Due Dates will be considered and will aim to produce competitive results with the State of the Art results found in the literature.

The paper will analyze the problem, study the literature and examine various approaches to solving the problem. Code capable of generating complete solutions will be developed to achieve high quality solutions for publicly available problems.

Keywords: *Scheduling (timetabling), Distributed Scheduling, Flow-shop Scheduling, Scheduling Optimization, Due Dates, Heuristic Algorithms, Metaheuristic Algorithms, Genetic Algorithms – GA, Hybrid Algorithms, Production Planning, Local Search*.

Περιεχόμενα

1. Εισαγωγή

Το Distributed Permutation Flow Shop Problem (DPFSP) είναι ένα κλασικό πρόβλημα βελτιστοποίησης που ανήκει στην κατηγορία των προβλημάτων προγραμματισμού διαδικασιών παραγωγής. Συγκεκριμένα, το DPFSP αφορά την εύρεση της βέλτιστης σειράς εκτέλεσης μιας ομάδας εργασιών σε πολλαπλές μηχανές ή σταθμούς εργασίας, σε πολλά πανομοιότυπα εργοστάσια με στόχο την ελαχιστοποίηση του συνολικού χρόνου παραγωγής, γνωστού ως makespan ή κάποιο άλλο κριτήριο απόδοσης όπως ο χρόνος καθυστέρησης (tardiness) σχετικά με τους χρόνους ή ημερομηνίες λήξης συγκεκριμένων διεργασιών (due dates).

Στο πρόβλημα αυτό, κάθε εργασία πρέπει να περάσει από όλες τις μηχανές με την ίδια σειρά, και η πρόκληση είναι να βρεθεί η καλύτερη αλληλουχία των εργασιών που θα μειώσει το χρόνο ολοκλήρωσης.

Το DPFSP αποτελεί ένα από τα πιο σημαντικά και μελετημένα προβλήματα στη θεωρία του χρονοπρογραμματισμού, καθώς έχει άμεσες εφαρμογές στη βιομηχανία και στη διαχείριση εφοδιαστικής αλυσίδας. Λόγω της NP-πληρότητάς του, η εύρεση της βέλτιστης λύσης είναι ιδιαίτερα δύσκολη για μεγάλα σύνολα εργασιών, καθιστώντας απαραίτητη τη χρήση ευρευτικών και μεταευρετικών αλγορίθμων.

Κατά τη διάρκεια των τελευταίων ετών, το πρόβλημα DPFSP έχει γίνει ένας πολύ ενεργός τομέας έρευνας. Ωστόσο, η ελαχιστοποίηση της συνολικής καθυστέρησης στο DPFSP, ένας πολύ ουσιαστικός και σχετικός στόχος για τη σημερινή αγορά που είναι προσανατολισμένη στον πελάτη, δεν έχει μελετηθεί πολύ.

Στην παρούσα εργασία, θα αναλύσουμε τις βασικές αρχές του προβλήματος DPFSP, ως κριτήριο απόδοσης, θα μελετήσουμε τις περιπτώσεις του χρόνου καθυστέρησης, θα εξετάσουμε κλασικούς αλγόριθμους επίλυσής του, και θα προτείνουμε μεθόδους βελτιστοποίησης που μπορούν να βελτιώσουν τα αποτελέσματα σε σύνθετες περιπτώσεις.

Ξεκινώντας θα γίνει μια σύντομη ιστορική αναφορά, σχετικά με την έρευνα και την προσέγγιση προβλημάτων που αφορούν τον χρονοπρογραμματισμό για να συνεχίσουμε με την παρουσίαση των κατηγοριών προβλημάτων χρονοπρογραμματισμού που είναι θεμελιώδεις, για την βελτίωση της απόδοσης σε παραγωγικά και επιχειρησιακά περιβάλλοντα. Εδώ θα περιγράψουμε αναλυτικότερα τα προβλήματα PFSP και DPFSP στα οποία επικεντρώνεται η εργασία από την πλευρά των χρόνων καθυστέρησης, σύμφωνα πάντα με τους χρόνους επιθυμητής ολοκλήρωσης (due dates) της κάθε εργασίας. Ακολουθεί η παρουσίαση του μοντέλου και η γραφική απεικόνιση του προβλήματος για να περάσουμε στις μεθόδους επίλυσης του προβλήματος. Για την λύση του προβλήματος αναπτύχθηκε ένας ευρετικός αλγόριθμος ο οποίος προσαρμόζει τον αλγόριθμο NEH σε προβλήματα με χρόνους due dates. Στη συνέχεια δημιουργήθηκαν δύο κύριοι αλγόριθμοι ένας άπληστος και ένας γενετικός οι οποίοι χρησιμοποιούν επιμέρους λύσεις που προκύπτουν από αλγόριθμους τοπικής αναζήτησης. Τέλος παρουσιάζονται τα αποτελέσματα των λύσεων και συγκρίνονται με τις καλύτερες λύσεις τις βιβλιογραφίας.

2. Ιστορική Αναφορά

Η έννοια του χρονοπρογραμματισμού δεν είναι καινούργια, οι πυραμίδες κατασκευάστηκαν πριν από 3000 έτη, ο Sun Tzu έγραψε για την διαχείριση χρόνου, από στρατιωτική άποψη πριν 2500 χρόνια και οι διηπειρωτικοί σιδηρόδρομοι κατασκευάζονται εδώ και περίπου 200 χρόνια. Καμία από τις προαναφερθείσες δραστηριότητες δεν θα είχαν επιτευχθεί χωρίς κάποια μορφή χρονοδιαγράμματος και χωρίς την κατανόηση δραστηριοτήτων και ακολουθιών. Το *Permutation Flow Shop Problem* (PFSP), γνωστό και ως πρόβλημα ακολουθιακής παραγωγικής διαδικασίας (ή ροής), είναι ένα από τα κλασικά προβλήματα βελτιστοποίησης που εντάσσεται στον τομέα της επιχειρησιακής έρευνας και της θεωρίας προγραμματισμού. Πρόκειται για ένα από τα προβλήματα χρονοπρογραμματισμού που μελετώνται για τη βελτίωση της παραγωγικής διαδικασίας σε γραμμές παραγωγής.

Αρχικές έρευνες και επινοήσεις (1950-1960): Το πρόβλημα προέκυψε αρχικά από την ανάγκη βελτιστοποίησης της διαδικασίας παραγωγής σε βιομηχανικά περιβάλλοντα, όπου πολλαπλές εργασίες πρέπει να εκτελούνται σε μια σειρά από μηχανές με καθορισμένη σειρά. Η βασική ιδέα ήταν να μειωθεί ο συνολικός χρόνος εκτέλεσης (makespan) για ένα σύνολο εργασιών. Η πρώτη σημαντική συνεισφορά σε αυτό το πρόβλημα ήρθε από τους *Johnson* και *Jackson* [1] τη δεκαετία του 1950, οι οποίοι εισήγαγαν αλγόριθμους για απλά συστήματα δύο σταδίων.

Η επέκταση του προβλήματος (1970-1980): Κατά τη δεκαετία του 1970, το πρόβλημα του Permutation Flow Shop άρχισε να εξετάζεται σε πιο σύνθετες μορφές, όπου η ακολουθία των εργασιών έπρεπε να διατηρείται ίδια για όλες τις μηχανές. Αυτή η νέα μορφή του προβλήματος περιγράφει ένα σύστημα όπου όλες οι εργασίες ακολουθούν την ίδια σειρά εκτέλεσης, αλλά η χρονική διάρκεια και η διαδοχή αυτών μπορεί να ποικίλλουν. Το 1974 προστέθηκαν από τον Baker [1] οι εξής παραδοχές:

- Όλες οι εργασίες είναι ανεξάρτητες και διαθέσιμες για επεξεργασία τη χρονική στιγμή 0.
- Οι μηχανές είναι συνεχώς διαθέσιμες.
- Κάθε μηχανή επεξεργάζεται μόνο μια εργασία κάθε φορά.
- Κάθε εργασία μπορεί να επεξεργαστεί μόνο σε μια μηχανή κάθε φορά.
- Από τη στιγμή που η επεξεργασία μιας συγκεκριμένης εργασίας έχει ξεκινήσει σε μια συγκεκριμένη μηχανή, δεν μπορεί να διακοπεί και η επεξεργασία συνεχίζεται μέχρι την ολοκλήρωσή της.
- Οι χρόνοι προετοιμασίας είναι ανεξάρτητοι από την ακολουθία και περιλαμβάνονται στους χρόνους επεξεργασίας ή αγνοούνται.
- Επιτρέπεται άπειρη αποθήκευση εντός της διαδικασίας, που σημαίνει ότι υπάρχει απεριόριστος χώρος αποθήκευσης μεταξύ των σταδίων παραγωγής, για τις εργασίες που βρίσκονται σε εξέλιξη.

Αυτή η επέκταση αύξησε την πολυπλοκότητα και την ανάγκη για νέες μεθόδους επίλυσης.

Ανάπτυξη των αλγορίθμων (1980-2000): Από τη δεκαετία του 1980 και μετά, με την ανάπτυξη των υπολογιστικών μεθόδων, οι ερευνητές άρχισαν να χρησιμοποιούν αλγόριθμους όπως η προσομοιωμένη ανόπτηση (Simulated Annealing), οι γενετικοί αλγόριθμοι και οι αλγόριθμοι διακλάδωσης και φραγής (Branch and Bound) για την επίλυση του PFSP. Η πολυπλοκότητα του προβλήματος, ιδιαίτερα για μεγάλα σύνολα εργασιών και μηχανών, έκανε τους ακριβείς αλγόριθμους δύσχρηστους, οδηγώντας στην ανάγκη για πιο αποδοτικές ευρετικές μεθόδους. Εμφανίστηκε μια πληθώρα ευρετικών μεθόδων για την επίλυση του προβλήματος Permutation Flow Shop το οποίο συμβολίζεται F | prmu | Cmax ακολουθώντας τη σημειολογία τριών πεδίων του Graham [2]. Το 1983 παρουσιάστηκε από τους Nawaz, Enscore και Ham ο περίφημος αλγόριθμος NEH ο οποίος μέχρι σήμερα παραμένει ο αποτελεσματικότερος ευρετικός αλγόριθμος, σύμφωνα με τους Ruiz και Maroto (2005) [3] οι οποίοι αξιολόγησαν μια σειρά από ευρετικές μεθόδους.

Σύγχρονες προσεγγίσεις (2000-σήμερα): Η έλευση πιο σύγχρονων μεθόδων, όπως οι υβριδικές ευρετικές και οι μεταευρετικές μέθοδοι (όπως Tabu Search, Particle Swarm Optimization και Ant Colony Optimization), επέτρεψε την επίλυση πιο σύνθετων προβλημάτων PFSP με καλύτερη ποιότητα αποτελεσμάτων. Ταυτόχρονα, αυξήθηκε το ενδιαφέρον για την εφαρμογή αυτών των μεθόδων σε πραγματικά σενάρια παραγωγής σε βιομηχανίες, από την αυτοκινητοβιομηχανία μέχρι την κατασκευή ηλεκτρονικών συσκευών. Όσον αφορά τις μεταευρετικές μεθόδους, υπάρχει επίσης εκτενής βιβλιογραφία με διαφορετικές προτάσεις για το PFSP με διαφορετικά κριτήρια. Αξιοσημείωτες μέθοδοι αναζήτησης Taboo Search είναι αυτές των Nowicki και Smutnicki (1996) [4] ή των Grabowski και Wodecki (2004) [5]. Οι αλγόριθμοι προσομοιωμένης ανόπτησης προτείνονται από τους Osman and Potts (1989) [6], ενώ οι γενετικοί αλγόριθμοι παρουσιάζονται από τους Reeves (1995) [7] και τους Ruiz, Maroto and Alcaraz (2006) [8]. Άλλες μεταευρετικές μέθοδοι όπως η Ant Colony Optimization, η Scatter Search, η Discrete Differential Evolution, η Particle Swarm Optimization ή η Iterated Greedy παρουσιάζονται από τους Rajendran and Ziegler (2004) [9], Nowicki and Smutnicki (2005) [10], Onwubolu and Davendra (2006) [11], Tasgetiren et al. (2007) [12] και Ruiz and Stützle (2007) [13], αντίστοιχα. Σχετικά πρόσφατες και υψηλής απόδοσης προσεγγίσεις περιλαμβάνουν μεθοδολογίες παράλληλου υπολογισμού, όπως αυτή που παρουσιάζεται στους Vallada and Ruiz (2009) [14] .

Το πρόβλημα του Permutation Flow Shop παραμένει έως και σήμερα ένα ενεργό πεδίο έρευνας, καθώς η βελτιστοποίηση της παραγωγικής διαδικασίας αποτελεί κομβικό στοιχείο για τη μείωση του κόστους και τη βελτίωση της αποδοτικότητας σε πολλούς βιομηχανικούς τομείς.

3. Περιγραφή του Προβλήματος

Ο προγραμματισμός του χρόνου είναι μια φυσική διαδικασία που οι άνθρωποι κάνουν καθημερινά. Ο προγραμματισμός όμως της καθημερινής μας ζωής είναι ως επί το πλείστον υποσυνείδητος και δεν κατευθύνεται από μια συγκεκριμένη διαδικασία και κανόνες. Συνήθως δεν υπάρχει η εφαρμογή μιας εξελιγμένης διαδικασίας λήψης αποφάσεων, ούτε η ανάγκη μιας υπολογιστικής τεχνικής. Ο κύριος λόγος γι' αυτό είναι ότι δεν υπάρχει μια πραγματική οικονομική συνάρτηση προς βελτίωση [15].

Στην επιστημονική θεωρία, ο χρονοπρογραμματισμός είναι το έργο της κατανομής πεπερασμένων πόρων κατά τη διάρκεια του χρόνου για την εκπλήρωση ενός δεδομένου συνόλου εργασιών. Η δομή ενός προβλήματος χρονοπρογραμματισμού, ενσωματώνει πολλές εργασίες και περίπλοκα χαρακτηριστικά που πρέπει να ληφθούν υπόψη για να χωριστεί στην συνέχεια η κάθε εργασία σε λειτουργίες που πρέπει να διεκπεραιωθούν από ένα σύνολο εφικτών μέσων. Η αντιμετώπιση αυτών των προβλημάτων, από την άποψη της επιστήμης, σημαίνει την εύρεση ενός χρονοδιαγράμματος που εκπληρώνει τους στόχους με τον καλύτερο δυνατό τρόπο.

Το πρόβλημα του χρονοπρογραμματισμού (Scheduling Problem) είναι μια διαδικασία λήψης αποφάσεων που χρησιμοποιείται σε τακτική βάση σε πολλές βιομηχανίες παραγωγής και υπηρεσιών. Στοχεύει στην βέλτιστη κατανομή των εργασιών σε μέσα παραγωγής και επεξεργασίας. Ένα πρόβλημα χρονοπρογραμματισμού φορτώνει ή αναθέτει εργασίες σε ένα μέσο με μια συγκεκριμένη ακολουθία με την οποία επεξεργάζονται οι εργασίες σε κάθε μέσο.

Ένα flowshop είναι ένα σύστημα παραγωγής όπου όλες οι μηχανές είναι τοποθετημένες στη σειρά και η ακολουθία των εργασιών στις μηχανές είναι η ίδια για όλες τις εργασίες.

3.1 Κατηγορίες Χρονοπρογραμματισμού

Στα παραδοσιακά προβλήματα χρονοπρογραμματισμού συναντάμε τις παρακάτω κύριες κατηγορίες:

3.1.1 Job Shop Problem (JSP):

Το Job Shop Scheduling Problem (JSP) είναι ένα κλασικό πρόβλημα συνδυαστικής βελτιστοποίησης που συναντάται στον προγραμματισμό παραγωγής. Αποτελεί μια θεμελιώδη αφαιρετική προσέγγιση για τη διαχείριση χρονοδιαγραμμάτων τόσο στην παραγωγή όσο και σε περιβάλλοντα υπηρεσιών. Συγκεκριμένα, περιλαμβάνει έναν πεπερασμένο αριθμό εργασιών, καθεμία από τις οποίες αποτελείται από μια σειρά λειτουργιών που πρέπει να εκτελεστούν σε ένα σύνολο μηχανών, σύμφωνα με προκαθορισμένους περιορισμούς και ακολουθίες. Κάθε εργασία απαιτεί προκαθορισμένο χρόνο επεξεργασίας και μπορεί να εκτελεστεί μόνο σε

μια συγκεκριμένη μηχανή. Ο στόχος του JSP είναι η κατασκευή ενός χρονοδιαγράμματος που βελτιστοποιεί ένα επιλεγμένο κριτήριο απόδοσης, το οποίο συχνά είναι η ελαχιστοποίηση του makespan [16].

3.1.2 Flow Shop Problem (**FSP** $(n!)^m$ πιθανές λύσεις) : Όλες οι εργασίες έχουν την ίδια σειρά επεξεργασίας μέσω των μηχανών. **Η σειρά των εργασιών σε κάθε μηχανή μπορεί να είναι διαφορετική**. Το FSP είναι εννοιολογικά κοντά με το JSP. Στο FSP ένα σύνολο εργασιών υπόκειται σε επεξεργασία σε μια ακολουθία μηχανών, καθεμία με την ίδια προκαθορισμένη σειρά εργασιών. Αυτός ο περιορισμός σειράς διαφοροποιεί το FSP, που διατηρεί την ίδια σειρά εργασιών σε όλες τις εργασίες από το JSP, όπου οι ακολουθίες εργασιών μπορούν να διαφέρουν μεταξύ των μηχανών. Από την άλλη πλευρά, στο FSP, η σειρά των εργασιών που εκτελούνται σε κάθε μηχανή μπορεί να είναι διαφορετική [16].

3.1.3 Permutation Flow Shop Problem : Όλες οι πιθανές εργασίες έχουν την ίδια σειρά επεξεργασίας μέσω των μηχανών. **Κάθε μηχάνημα επεξεργάζεται τις εργασίες με την ίδια σειρά**. Η PFSP προκύπτει ως παραλλαγή της FSP, εισάγοντας έναν πρόσθετο περιορισμό που δηλώνει ότι μια σταθερή σειρά εργασιών πρέπει να διατηρείται για όλες τις μηχανές. Ταυτόχρονα, όπως και στην FSP, όλες οι εργασίες έχουν την ίδια σειρά επεξεργασίας μέσω των μηχανών. Η PFSP αντιπροσωπεύει μια πραγματική απαίτηση αρκετών εγκαταστάσεων όταν, για παράδειγμα, ένας μεταφορέας τροφοδοτεί τις εργασίες στις μηχανές. Τότε, η σειρά των εργασιών που πρέπει να επεξεργάζεται κάθε μηχανή πρέπει να είναι η ίδια για όλες τις εργασίες [16].

3.1.4 Distributed Permutation Flow Shop Problem (DPFSP): Τέλος, το DPFSP είναι ένα πρόβλημα που επεκτείνει το PFSP επιτρέποντας περισσότερα από ένα πανομοιότυπα εργοστάσια να υπάρχουν στο πρόβλημα. Έτσι, κάθε εργοστάσιο περιέχει ένα πανομοιότυπο σύνολο μηχανών με άλλα εργοστάσια. Συμβολίζοντας με P_{ji} f τον χρόνο επεξεργασίας της εργασίας j στο μηχανή i του εργοστασίου f, ισχύει ότι $P_{ji}0 = P_{ji}1 = P_{ji}2$. Ο τυπικός στόχος του DPFSP είναι η ελαχιστοποίηση του χρονικού διαστήματος επεξεργασίας (makespan) ή του ελάχιστου χρόνου καθυστέρησης σχετικά με τον αναμενόμενο χρόνο τερματισμού (due date / tardiness) [16].

3.2 PFSP

Το πρόβλημα PFSP είναι ένα πρόβλημα προγραμματισμού μιας διαδικασίας παραγωγής, κατά την οποία θα πρέπει να εκτελεστεί ένας αριθμός εργασιών, σε έναν αριθμό μηχανών σε μια συγκεκριμένη ακολουθία.

Συγκεκριμένα έχουμε:

- n εργασίες και κάθε εργασία χαρακτηρίζεται από έναν μοναδικό αριθμό π.χ. i = 1, 2, 3, … , n.
- m μηχανές και κάθε μηχανή χαρακτηρίζεται από έναν μοναδικό αριθμό π.χ. j = 1, 2, 3, … , m.
- ο χρόνος εκτέλεσης t_{ij} της εργασίας i στην μηχανή j. (Οι χρόνοι εκτέλεσης είναι σταθεροί, θετικοί αριθμοί και μπορούν να πάρουν την τιμή 0 όταν μια εργασία δεν εκτελείται καθόλου σε μια μηχανή).

Στόχος της επίλυσης του προβλήματος είναι να βρεθεί μια σειρά εκτέλεσης των εργασιών, ώστε να ελαχιστοποιηθεί ο συνολικός χρόνος εκτέλεσης, από την στιγμή που θα εισέλθει η πρώτη εργασία στην πρώτη μηχανή, μέχρι να εξέλθει η τελευταία εργασία από την τελευταία μηχανή (makespan).

Θα πρέπει όμως να λάβουμε υπόψη τις εξής παραδοχές:

- Κάθε εργασία εκτελείται μια φορά σε κάθε μηχανή 1,2,3, ... ,n και με την συγκεκριμένη σειρά π.χ. μια εργασία δεν μπορεί να εκτελεστεί πρώτα στην μηχανή 2 και μετά στην μηχανή 1.
- Κάθε μηχανή εκτελεί μια εργασία κάθε φορά.
- Κάθε εργασία εκτελείται σε μια μηχανή την φορά.

Το πρόβλημα PFSP είναι ένα πρόβλημα βελτιστοποίησης και εμφανίζεται σε πολλές βιομηχανικές εφαρμογές. Έχει ως στόχο τη βελτιστοποίηση της παραγωγικότητας, την εξοικονόμηση του κόστους, την επιτάχυνση παράδοσης και τη βελτίωση της ποιότητας ενός προϊόντος.

3.3 DPFSP

Το παραδοσιακό πρόβλημα χρονοπρογραμματισμού (PFSP) ακολουθεί ένα περιβάλλον παραγωγής όπου όλες οι διαδικασίες παραγωγής λαμβάνουν χώρα στο ίδιο εργοστάσιο. Ωστόσο, στη σημερινή αποκεντρωμένη και παγκοσμιοποιημένη οικονομία, οι μεγάλες επιχειρήσεις έχουν συνήθως πολλά κέντρα παραγωγής σε όλο τον κόσμο. Αυτό το πρότυπο παραγωγής σε πολλά εργοστάσια όχι μόνο επιτρέπει στις εταιρείες να μειώσουν το κόστος παράδοσης και κατασκευής, αλλά τις βοηθά επίσης στην επίτευξη καλύτερης ποιότητας προϊόντων [1].

Για να αντιμετωπιστεί αυτό το πολυπαραγωγικό μοντέλο παραγωγής, οι Naderi και Ruiz (2010) [1] πρότειναν τον κατανεμημένο προγραμματισμό ροών με μεταθέσεις (flowshop scheduling), πρόβλημα (DPFSP) το οποίο μπορεί να θεωρηθεί ως μια επέκταση του κανονικού PFSP. Το DPFSP αποτελείται από F (περισσότερα από ένα) πανομοιότυπα εργοστάσια, το καθένα με m μηχανές. Η διαδικασία χρονοπρογραμματισμού στο DPFSP μπορεί να χωριστεί σε δύο μέρη: την ανάθεση εργασιών σε διαφορετικά εργοστάσια και την αλληλουχία των εργασιών που ανατίθενται σε κάθε εργοστάσιο σύμφωνα με το δεδομένο μέτρο απόδοσης.

Οι περισσότερες από τις προηγούμενες εργασίες για το DPFSP αναγνώριζαν μόνο το makespan ως μέτρο απόδοσης. Ωστόσο, στο σημερινό πελατοκεντρικό σύστημα ανταγωνιστικής αγοράς, η έγκαιρη παράδοση του προϊόντος έχει γίνει πιο κρίσιμη από ποτέ [17]. Η ελαχιστοποίηση της συνολικής καθυστέρησης επιτρέπει στις βιομηχανίες παραγωγής την ολοκλήρωση των παραγγελιών των πελατών πριν από τις ημερομηνίες λήξης τους. Ωστόσο, παρά το γεγονός ότι πρόκειται για ένα ρεαλιστικό μέτρο απόδοσης, η ελαχιστοποίηση της συνολικής καθυστέρησης (TT – Total Tardiness) για το DPFSP δεν έχει προσελκύσει μεγάλη προσοχή μέχρι σήμερα. Για την αντιμετώπιση αυτού του μειονεκτήματος, στην παρούσα εργασία, διερευνούμε το DPFSP με συνολική καθυστέρηση το οποίο μπορεί να συμβολιστεί ως

$$\boldsymbol{DF \mid prmu \mid \sum T_j}$$ [1] .

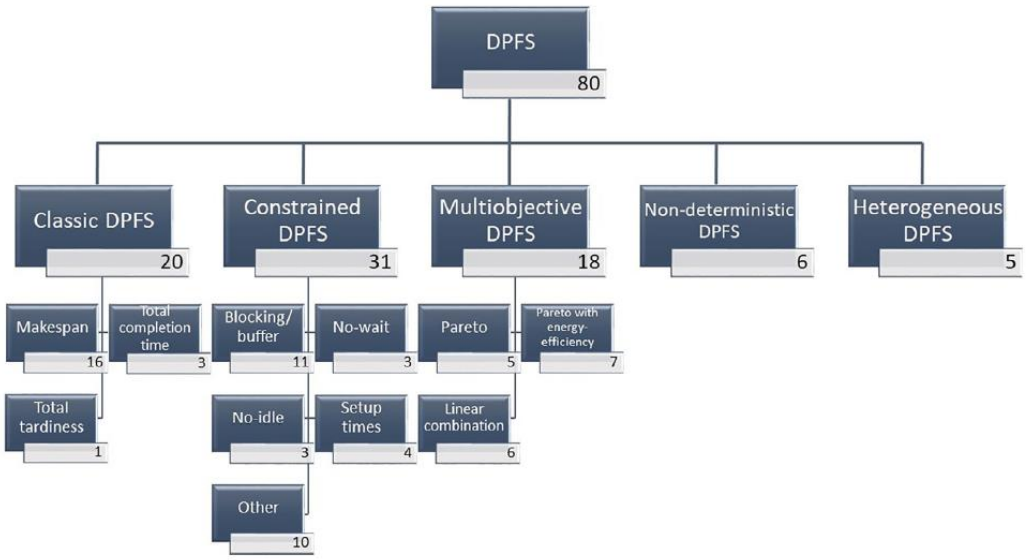
Το εξεταζόμενο πρόβλημα είναι ένα NP-Hard πρόβλημα καθώς αποτελεί επέκταση του κανονικού PFSP με κριτήριο την συνολική καθυστέρηση.

Τύποι προβλημάτων PFSP [18]

- ***Τα κλασικά προβλήματα DPFS*** περιλαμβάνουν συνεισφορές στο πρόβλημα DPFS χωρίς πρόσθετους περιορισμούς (εκτός από τον περιορισμό prmu) και έναν μόνο στόχο. Οι περισσότερες από αυτές τις εργασίες ασχολούνται με το makespan (συνολικό χρόνο ολοκλήρωσης).
- ***Τα προβλήματα DPFS με περιορισμούς*** είναι προβλήματα που περιλαμβάνουν τουλάχιστον έναν πρόσθετο περιορισμό. Οι συνηθέστεροι περιορισμοί που εξετάζονται αφορούν χρόνους προετοιμασίας του εξοπλισμού, απαγόρευση αναμονής και αδράνειας μεταξύ διαδοχικών εργασιών. Ορισμένοι στόχοι που μελετώνται είναι ο συνολικός χρόνος ολοκλήρωσης ενός συνόλου εργασιών και η συνολική

καθυστέρηση ολοκλήρωσης ενός συνόλου εργασιών όταν έχουμε επιθυμητούς χρόνους τερματισμού (due dates)

- ***Τα πολυκριτηριακά προβλήματα DPFS*** είναι προβλήματα όπου δεν υπάρχει μόνο ένας στόχος προς επίτευξη, αλλά περισσότεροι, οι οποίοι πρέπει να συνδυαστούν. Υπάρχουν δύο βασικοί τρόποι για να λυθούν αυτά τα προβλήματα: Η προσέγγιση Pareto και ο γραμμικός συνδυασμός. Κατά την προσέγγιση Pareto εξετάζονται διάφορες λύσεις που εξυπηρετούν διαφορετικούς στόχους, χωρίς να θυσιάζεται κανένας απόλυτα, ενώ κατά την προσέγγιση γραμμικού συνδυασμού οι στόχοι συνδυάζονται σε μια ενιαία βαθμολογία, ώστε να λυθεί το πρόβλημα σαν να υπήρχε ένας στόχος προς επίτευξη.
- ***Τα μη ντετερμινιστικά προβλήματα DPFS*** εξετάζουν αναφορές σε ασαφή, στοχαστικά και αβέβαια προβλήματα. Οι στόχοι που εξετάζονται κυρίως αφορούν το makespan.
- ***Τα ετερογενή προβλήματα DPFS*** αναφέρονται σε προβλήματα υψηλής πολυπλοκότητας καθώς αναζητούν λύσεις σε μη πανομοιότυπα εργοστάσια, λαμβάνοντας υπόψη τις περιπτώσεις με διαφορετικό αριθμό μηχανών, διατάξεις ή μόνο διαφορετικούς χρόνους επεξεργασίας μεταξύ των εργοστασίων.



Εικόνα 1 Τύποι Προβλημάτων DPFS [18]

Due Dates

Ενώ αρκετές έρευνες προσεγγίζουν το πρόβλημα DPFSP από την πλευρά του συντομότερου χρόνου εκτέλεσης (makespan), μια άλλη προσέγγιση επικεντρώνεται στην ανάπτυξη αλγορίθμων με στόχο την ελαχιστοποίηση της συνολικής ή της μέσης καθυστέρησης των θέσεων εργασίας. Έχουν γίνει αρκετές προσπάθειες, για την ανάπτυξη ευρετικών μεθόδων, για την επίλυση του χρονοπρογραμματισμού flowshop, με στόχο την ελαχιστοποίηση της συνολικής καθυστέρησης των εργασιών. Η πρώτη προσπάθεια ήταν από τους Gelders και Sambandam (1978) [19] οι οποίοι ανέπτυξαν ευρετικές τεχνικές. Αργότερα, ο Kim (1993) πρότεινε μια ευρετική που βασιζόταν στη διαδικασία αναζήτησης taboo των Widmer και Hertz (1989) [20] και οι Parthasarathy και Rajendran (1998) [21] πρότειναν δύο ευρετικές που βασίζονται στην τεχνική της προσομοιωμένης ανόπτησης.

Η ελαχιστοποίηση του συνολικού χρόνου καθυστέρησης επιτρέπει στις μεταποιητικές βιομηχανίες την ολοκλήρωση των παραγγελιών των πελατών πριν από την ημερομηνία λήξης τους (αποτρέποντας είτε την απώλεια φήμης, είτε ακόμη και την απώλεια πελάτη) [22].

4. Περιγραφή του Μοντέλου

4.1 Συμβολισμοί και περιορισμοί

Για την περιγραφή του προβλήματος υιοθετήθηκε το μοντέλο των Khare και Agrawal (2020) [23], που έχει ως στόχο την κατανομή των εργασιών στα εργοστάσια με τέτοια σειρά, ώστε να ελαχιστοποιείται η συνολική καθυστέρηση. Το πρόβλημα αφορά περισσότερα από ένα πανομοιότυπα εργοστάσια, δηλ. f > 1, που βρίσκονται σε διαφορετικές περιοχές, με την ευθύνη της επεξεργασίας ενός συνόλου n εργασιών.

- Όλα τα εργοστάσια έχουν το ίδιο σύνολο m μηχανών.
- Όλες οι μηχανές και οι εργασίες είναι διαθέσιμες τη χρονική στιγμή μηδέν.
- Κάθε μηχανή μπορεί να επεξεργαστεί μόνο μία εργασία κάθε φορά και κάθε εργασία μπορεί να διεκπεραιωθεί σε μία μηχανή κάθε φορά και ο χρόνος επεξεργασίας μιας εργασίας σε μια συγκεκριμένη μηχανή είναι η ίδια από εργοστάσιο σε εργοστάσιο [23].

4.1.1 Δείκτες

- **j, k** Δείκτες εργασιών και θέσεων εργασιών - j, k ∈ {1, 2, ... , n}
- **i** Δείκτης για τις μηχανές- i ∈ {1, 2, ... ,m}
- **f** Δείκτης για τα εργοστάσια, f ∈ {1, 2, ... , F}
- **M** Ένας επαρκώς μεγάλος θετικός αριθμός [23]

4.1.2 Παράμετροι

- **n** Αριθμός εργασιών
- **m** Αριθμός μηχανών
- **F** Αριθμός εργοστασίων
- $\mathbf{p_{i,j}}$ Χρόνος επεξεργασίας της εργασίας j στη μηχανή i
- $\mathbf{d_j}$ Ημερομηνία λήξης της εργασίας

4.1.3 Μεταβλητή απόφασης

- $\mathbf{X_{j,k}}$ 1 εάν η θέση εργασίας j ανατίθεται στη θέση k και 0 διαφορετικά
- $\mathbf{Y_{k,f}}$ 1 εάν η θέση εργασίας k ανατίθεται στο εργοστάσιο f και διαφορετικά 0
- $\mathbf{C_{k,i}}$ Χρόνος ολοκλήρωσης της εργασίας στη θέση k στο μηχάνημα i
- $\mathbf{U_k}$ Ημερομηνία λήξης της εργασίας στη θέση k
- $\mathbf{T_k}$ Καθυστέρηση της εργασίας j στη θέση k
- **TT** Συνολική καθυστέρηση

$MinTT = \sum_{k=1}^{n} T_k$	(1)	Καθορίζει την συνάρτηση για την ελαχιστοποίηση της συνολικής καθυστέρησης TT
$\sum_{k=1}^{n} X_{j,k} = 1 \; \forall \; j$ $\sum_{j=1}^{n} X_{j,k} = 1 \; \forall \; k$	(2) (3)	Μαζί εξασφαλίζουν ότι κάθε εργασία πρέπει να κατανέμεται σε ακριβώς μία θέση και κάθε θέση πρέπει να ανατίθεται ακριβώς σε μία εργασία.
$C_{k,j} \geq \; C_{k,j} - 1 + \sum_{j=1}^{n} X_{j,k} \; p_{i,j} \; \forall \; k, j$	(5)	Εξασφαλίζει ότι η έναρξη μιας εργασίας σε μια συγκεκριμένη θέση μηχανής μπορεί να ξεκινήσει εάν έχει τελειώσει η επεξεργασία της ίδιας εργασίας στην προηγούμενη θέση.
$C_{k,j} \geq C_{k,j} - 1 + \sum_{j=1}^{n} X_{j,k} \; p_{i,j} - M\left(1 - Y_{k,f}\right) -$ $-M\left(1 - Y_{l,f}\right) \; \forall \; k > 1, l < k, i, f$	(6)	Εξασφαλίζει ότι η έναρξη εργασίας σε μια συγκεκριμένη μηχανή μπορεί να ξεκινήσει αφού έχει τελειώσει η επεξεργασία της προηγούμενης εργασίας στην ίδια μηχανή.
$U_k = \sum_{j=1}^{n} X_{j,k} \; d_j \; \forall \; k$	(7)	Βρίσκει την ημερομηνία λήξης σύμφωνα με την θέση της εργασίας.
$T_k = \max\left(C_{k,i} - U_k, 0\right) \forall \; k, i = m$	(8)	Υπολογίζει την καθυστέρηση μιας εργασίας στην θέση k.
$T_k \geq 0 \; \forall \; k$ $C_{k,j} \geq 0 \; \forall \; k, j$ $X_{j,k} \; \in \{0,1\} \forall \; j, k$ $Y_{k,f} \; \in \{0,1\} \forall \; k, f$	(9) (10) (11) (12)	Ορίζουν τις μεταβλητές απόφασης.

Πίνακας 1 Αντικειμενική συνάρτηση και περιορισμοί του προβλήματος

4.2 MILP

Το MILP (Mixed Integer Linear Programming) είναι ένας κλάδος των μαθηματικών βελτιστοποίησης που συνδυάζει τις τεχνικές του γραμμικού προγραμματισμού (Linear Programming - LP) με τον περιορισμό ότι ορισμένες μεταβλητές πρέπει να είναι ακέραιες (Integer Programming - IP). Αποτελεί μια ισχυρή μέθοδο για την επίλυση προβλημάτων όπου ζητείται ο βέλτιστος τρόπος να επιτευχθεί ένα συγκεκριμένο αποτέλεσμα υπό δεδομένους περιορισμούς, ενώ κάποιες από τις μεταβλητές πρέπει να λάβουν ακέραιες τιμές [24] **.**

Η λειτουργία του MILP περιγράφεται ως εξής:

1. **Μοντελοποίηση**: Το πρώτο βήμα είναι η μοντελοποίηση του προβλήματος, δηλαδή ο ορισμός της συνάρτησης στόχου και των περιορισμών που διέπουν το σύστημα.

2. **Χρήση μαθηματικών εργαλείων**: Χρησιμοποιούνται αλγόριθμοι όπως ο simplex ή αλγόριθμοι διακλάδωσης και φραγμάτων (branch and bound) για την εύρεση της βέλτιστης λύσης.

3. **Αποτελέσματα**: Το σύστημα επιστρέφει την τιμή των μεταβλητών που μεγιστοποιούν ή ελαχιστοποιούν τη συνάρτηση στόχου, ικανοποιώντας όλους τους περιορισμούς. Σημαντικό είναι ότι για τις ακέραιες μεταβλητές, η λύση πρέπει να είναι ακριβώς ακέραια.

Το MILP χρησιμοποιείται σε ένα πλήθος εφαρμογών, όπως:

- Βιομηχανική παραγωγή: Για την οργάνωση της παραγωγικής διαδικασίας με βέλτιστο τρόπο.
- Διαχείριση εφοδιαστικής αλυσίδας: Για τον βέλτιστο προγραμματισμό αποθεμάτων και μεταφορών.
- Ενεργειακός τομέας: Για τη βελτιστοποίηση της κατανομής των ενεργειακών πόρων.
- Χρηματοοικονομικός τομέας: Για τον καθορισμό βέλτιστων χαρτοφυλακίων επενδύσεων.

Η κύρια πρόκληση του MILP είναι η υπολογιστική πολυπλοκότητα. Όταν αυξάνονται οι μεταβλητές και οι περιορισμοί, η επίλυση του προβλήματος γίνεται πιο απαιτητική σε υπολογιστική ισχύ και χρόνο, ειδικά όταν οι ακέραιες μεταβλητές παίζουν κεντρικό ρόλο.

Το μοντέλο που παρουσιάσαμε στην ενότητα 4.1 έχει ως στόχο την κατανομή των εργασιών σε διάφορα εργοστάσια και τον καθορισμό των ακολουθιών επεξεργασίας τους σε κάθε εργοστάσιο για την ελαχιστοποίηση του χρόνου της συνολικής καθυστέρησης (TT) και μπορούμε να το περιγράψουμε με τον παρακάτω συμβολισμό [23] :

$$\boldsymbol{DF \mid prmu \mid \sum T_j}$$

4.3 Αριθμητική απεικόνιση

Η αναπαράσταση της λύσης ενός προβλήματος DPFSP, όπως παρουσιάζεται από τους Naderi and Ruiz [1], γίνεται με ένα σύνολο F λιστών λύσεων. Κάθε λίστα αναπαριστά μια ακολουθία π_f, όπου f=[1,2,3,…,F]. Η λίστα π={π_1, π_2, π_3, π_4, π_5, π_6, … π_f, … π_F} αναφέρεται σε F λίστες. Ένα πιθανό παράδειγμα με 12 jobs και 3 εργοστάσια, θα μπορούσαν να είναι οι παρακάτω ακολουθίες π=[{3,10,11,1}, {0,5,8,2}, {7,4,9,6}].

Σύμφωνα με τους Naderi and Ruiz, ο κανόνας το συντομότερου χρόνου ολοκλήρωσης (ECT) δίνει τα καλύτερα αποτελέσματα με το μικρότερο υπολογιστικό κόστος.

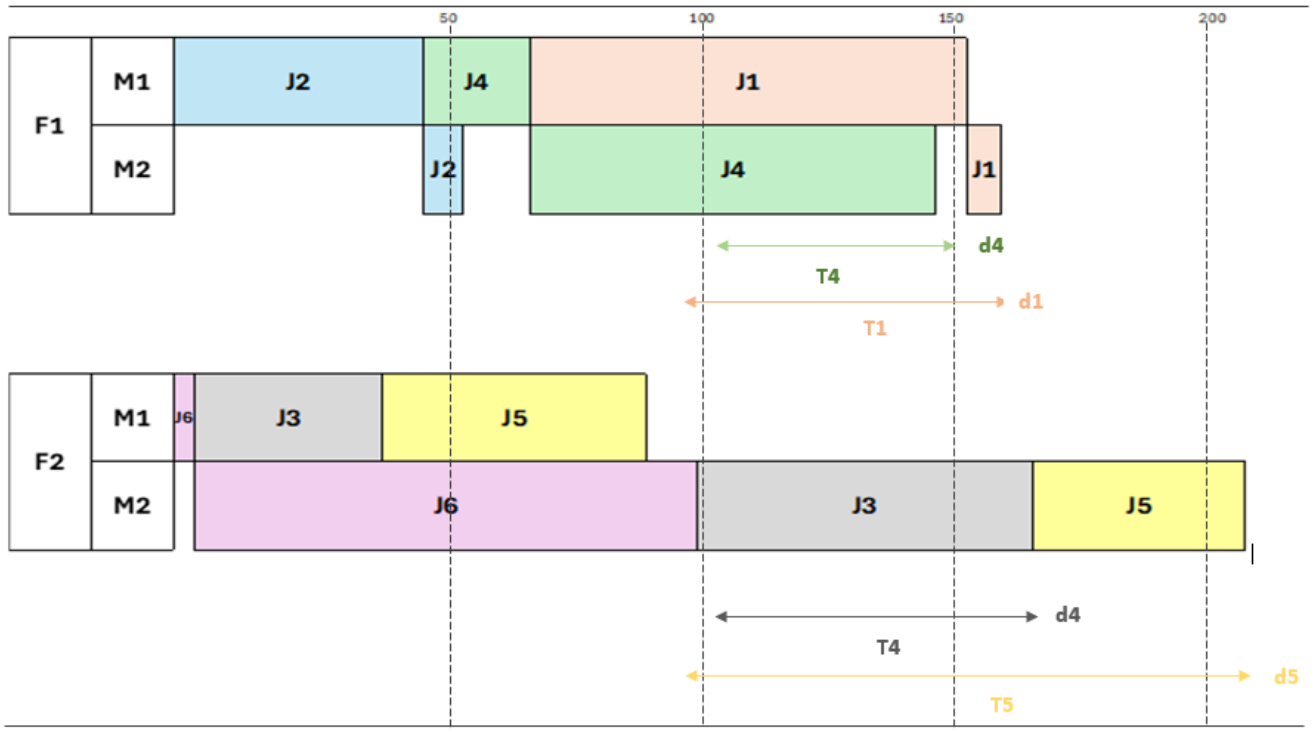
Για την αριθμητική απεικόνιση του προβλήματος θα χρησιμοποιήσουμε ένα παράδειγμα DPFSP όπου έχουμε τα παρακάτω δεδομένα: jobs = 6, machines = 2, factories = 2. Οι χρόνοι εκτέλεσης και οι ημερομηνίες τερματισμού φαίνονται στον παρακάτω πίνακα:

	Jobs					
MACHINE	J1	J2	J3	J4	J5	J6
M1	86	49	37	21	52	4
M2	7	8	66	80	42	99
Due date(dj)	96	58	103	103	95	107

Εικόνα 2 Αναπαράσταση Παραδείγματος

Αφού ταξινομήσουμε τις εργασίες σύμφωνα με τα due dates από το μικρότερο στο μεγαλύτερο, υπολογίζουμε την μικρότερη χρονική καθυστέρηση για την ακολουθία π = {J2, J5, J1, J3, J4, J6}.

Αρχικά τοποθετούμε την εργασία J2 στην πρώτη μηχανή του πρώτου εργοστασίου, καθώς και τα δυο εργοστάσια είναι άδεια, δεν επηρεάζει η επιλογή εργοστασίου την έκβαση του αποτελέσματος. Στην συνέχεια υπολογίζουμε τον χρόνο ολοκλήρωσης της εργασίας J5 στα δύο εργοστάσια. Στο δεύτερο εργοστάσιο θα τερματίσει νωρίτερα καθώς είναι άδειο, οπότε η εργασία J5 τοποθετείται στο δεύτερο εργοστάσιο. Η εργασία J1 θα τερματίσει στο εργοστάσιο 1 μετά από 142 μονάδες χρόνου, ενώ στο εργοστάσιο 2 μετά από 145 μονάδες χρόνου. Θα τοποθετηθεί στο εργοστάσιο 1. Με παρόμοιο τρόπο τοποθετούνται και οι υπόλοιπες εργασίες στα εργοστάσια για να καταλήξουμε στο παρακάτω διάγραμμα Gantt.



Εικόνα 3 Αναπαράσταση Λύσης με Ημερομηνίες Τερματισμού

Στο επόμενο βήμα θα πρέπει να υπολογίσουμε τον συνολικό χρόνο καθυστέρησης σύμφωνα με τις παραπάνω τοποθετήσεις των εργασιών και των ακολουθιών που προέκυψαν.

Η εργασία J2 τερματίζει σε 57 μονάδες χρόνου ενώ έχει due date 58 άρα δεν καθυστερεί και ο χρόνος καθυστέρησης T_2 είναι 0, καθώς δεν μπορεί να πάρει αρνητικές τιμές. Η εργασία J4 τερματίζει σε 150 μονάδες ενώ έχει due date 103. Οπότε η καθυστέρηση T_4 είναι 150 – 103 = 47. Η εργασία J1 τερματίζει σε 163 μονάδες ενώ έχει due date 96. Η καθυστέρηση T_1 είναι 163 – 96 = 67.

Για τις υπόλοιπες εργασίες οι χρόνοι καθυστέρησης είναι T_6 = 103 – 107 = -4 άρα 0, T_3 = 169 – 103 = 66, T_5 = 211 – 95 = 116. Ο συνολικός χρόνος καθυστέρησης είναι:

$$TT = \sum_{j=1}^{n} Tj = 47 + 67 + 66 + 116 = 296 \ \mu ονάδες \ χρόνου$$

5. Τρόποι Αντιμετώπισης Προβλήματος

5.1 Ευρετικές Μέθοδοι

Το πρόβλημα του προγραμματισμού ροής ή χρονοπρογραμματισμού αποτελεί έντονο πεδίο έρευνας εδώ και πολλά χρόνια με την συντριπτική πλειονότητα αυτών των προβλημάτων, για περισσότερες από 3 μηχανές να είναι NP-complete [25]. Ως αποτέλεσμα έχουμε την έρευνα να κατευθύνεται κυρίως προς την ανάπτυξη ευρετικών ή προσεγγιστικών (approximation) μεθόδων. Ο στόχος της ελαχιστοποίησης του makespan ήταν ο στόχος πολλών ευρετικών μεθόδων που έχουν αναπτυχθεί μέχρι σήμερα. Ορισμένες από τις αξιοσημείωτες ευρετικές μεθόδους για την ελαχιστοποίηση του makespan έχουν αναπτυχθεί από τους Campbell et al. (1970) [26], Dannenbring (1977) [27], Nawaz et al. (1983) [28], Widmer and Hertz (1989) [20], Leisten (1990) [29], Ogbu and Smith (1990) [30],Nowicki and Smutnicki (1996) [4], Rajendran and Ziegler (1997) [9], Ben-Daya and Al-Fawzan (1998) [31], και Framinan et al. (2001) [32].

5.1.1 Ο Αλγόριθμος NEH

Το 1983 η ομάδα Nawaz, Enscore, Ham παρουσίασαν τον αλγόριθμο NEH, ο οποίος θεωρείται ο «άρχοντας» των ευρετικών αλγόριθμων. Ο αλγόριθμος NEH είναι αλγόριθμος που χρησιμοποιείται για την επίλυση του προβλήματος PFSP και είναι από τους πιο αποδοτικούς ευρετικούς αλγόριθμους στην εύρεση του ελάχιστου χρόνου εκτέλεσης n εργασιών σε m μηχανές (makespan) [33].

Ο αλγόριθμος ακολουθεί τα εξής βήματα:

1. Ταξινόμηση εργασιών: Ταξινομεί τις εργασίες σύμφωνα με το συνολικό χρόνο εκτέλεσης τους σε όλες τις μηχανές σε φθίνουσα σειρά.
2. Δημιουργία ακολουθίας για τις πρώτες 2 εργασίες (α): Συγκρίνουμε τις δύο πρώτες εργασίες μετά την παραπάνω ταξινόμηση και επιλέγουμε τον καλύτερο συνδυασμό π.χ. εάν οι εργασίες στις πρώτες δύο θέσεις μετά την ταξινόμηση ήταν οι εργασίες 3 και 5 θα υπολογίσουμε τον συνδυασμό 3,5 και τον συνδυασμό 5,3 και θα επιλέξουμε αυτόν με τον ελάχιστο χρόνο.
3. Δημιουργία ακολουθίας για κάθε επόμενη εργασία: Κάθε εργασία τοποθετείται σε όλες τις πιθανές θέσεις και υπολογίζεται το makespan. Στη συνέχεια επιλέγεται πάντα η ακολουθία με το καλύτερο makespan. Π.χ. έχουμε την ακολουθία εργασιών [4,2,5,0] και θέλουμε να εισάγουμε την εργασία 3 στην ακολουθία. Θα υπολογίσουμε το makespan για όλες τις δυνατές θέσεις που μπορεί να εισαχθεί η εργασία 3 δλδ. [**3**,4,2,5,0] , [4, **3**,2,5,0] , [4,2, **3**,5,0] , [4,2,5, **3**,0] και [4,2,5, 0, **3**] και θα κρατήσουμε την ακολουθία με το καλύτερο makespan.
4.

Για την υλοποίηση του αλγορίθμου έχουν δημιουργηθεί 2 αρχεία:

- neh.py : Υλοποιεί την λογική του αλγόριθμου NEH και περιέχει τις συναρτήσεις:
  - job_sum_time(job_j, n_machines, p) : υπολογίζει τον συνολικό χρόνο εκτέλεσης κάθε εργασίας σε όλες τις μηχανές
  - start_order_neh(n_jobs, n_machines, p) : Ταξινομεί τις εργασίες σύμφωνα με τους συνολικούς χρόνους εκτέλεσης.
  - insertion(seq, i_position, timi) : Δημιουργεί τους συνδυασμούς ακολουθιών για κάθε εργασία.
  - neh(n_jobs, n_machines, p) : Ο κορμός του αλγόριθμου NEH

- makeShedule.py: Υλοποιεί τον υπολογισμό του makespan για μια συγκεκριμένη ακολουθία και περιέχει τις συναρτήσεις:
  - schedule(n_jobs, n_machines, p, solution) : υπολογίζει τους χρόνους (ακολουθεί αναλυτική περιγραφή).
  - makespan(job_sequence, C) : επιστρέφει το τελευταίο στοιχείο του πίνακα με τους χρόνους που αντιστοιχεί και στο συνολικό makespan.

```
#Συνάρτηση που υπολογίζει το άθροισμα των χρόνων του κάθε job
def job_sum_time(job_j, n_machines, p):
  sumJob = 0
  for i in range(n_machines):
    sumJob += p[job_j,i]
  return sumJob
```

```
#Ταξινομεί τα jobs με βάση το άθροισμά τους από το μεγαλύτερο στο μικρότερο
def start_order_neh(n_jobs, n_machines, p):
  startSeq = []
  for j in range(n_jobs):
    startSeq.append(j) # Προσθέτει το job j στη λίστα
#Επιστρέφει την λίστα ταξινομημένη σε φθίνουσα σειρά σύμφωνα με το άθροισμα τις συνάρτησης
job_sum_time()
    return sorted(startSeq, key=lambda x: job_sum_time(x, n_machines, p), reverse=True)
```

```
#Προσθέτει στην ακολουθία το στοιχείο στην σωστή θέση
#Η ακολουθία είναι η λίστα που περιέχει την σειρά που θα εκτελεστούν τα jobs
#Ξεκινάμε με μια τιμή την πρώτη από την ταξινόμηση των αθροισμάτων
def insertion(seq, i_position, timi):
  new_seq = seq[:]
  new_seq.insert(i_position, timi)
  return new_seq
```

```
#Εκτέλεση του αλγόριθμου NEH
#Αποθηκέυουμε την ακολουθία αφού την ταξινομήσουμε στην λίστα seq
def neh(n_jobs, n_machines, p):
#ταξινόμηση των εργασιών
 seq = start_order_neh(n_jobs, n_machines, p)
 workSequense = [seq[0]] # ακολουθία για το τρέξιμο του αλγόριθμου
 bestSequense = [seq[0]] # καλύτερη ακολουθία μετά από κάθε κύκλο

 #Διατρέχουμε για όλα τα jobs
 for i in range(1, n_jobs):
   bestTime=float("inf") #Δίνουμε αρχική τιμή στον καλύτερο χρόνο μια πολύ μεγάλη τιμή
   tmpTime=0
   #Διατρέχουμε για όλες τις θέσεις της ακολουθίας
   #Το πρώτο στοιχείο το έχουμε τοποθετήσει παραπάνω
   #Το δεύτερο στοιχείο θα έχει 2 πιθανές θέσεις
   #Το τρίτο στοιχείο έχει 3 πιθανές θέσεις κοκ
   for j in range(0, i + 1):   #
      tmp_seq = insertion(workSequense, j, seq[i])
       C = sm.schedule(n_jobs, n_machines, p, tmp_seq) # καλούμε την συνάρτηση shedule(x,y,z,p)
που περιγράφεται παρακάτω
        tmpTime=sm.makespan(tmp_seq, C) # επιστρέφει το makespan για μια συγκεκριμένη ακολουθία
#Εαν ο χρόνος τις τρέχουσας ακολουθίας είναι καλύτερος από τον καλύτερο χρόνο του συγκεκριμένου
#κύκλου ορίζουμε την ακολουθία ως καλύτερη και ενημερώνουμε τον χρόνο
            if bestTime > tmpTime:
                bestTime = tmpTime
                bestSequense = tmp_seq
#Μετά το τέλος ενός κύκλου μιας εργασίας ενημερώνουμε την ακολουθία που εργαζόμαστε με την
#καλύτερη ακολουθία του προηγούμενου κύκλου για να χρησιμοποιηθεί από τον επόμενο κύκλο.
        workSequense=bestSequense
#Επιστρέφει την καλύτερη τελευταία ακολουθία που είναι και η καλύτερη ακολουθία του αλγόριθμου
    return bestSequense
```

Κώδικας 1 Ο Αλγόριθμος NEH (1)

Η συνάρτηση η οποία υπολογίζει τους χρόνους εκτέλεσης των εργασιών σύμφωνα με μια ακολουθία (solution), δέχεται ως ορίσματα τον αριθμό των εργασιών (n_jobs), τον αριθμό των μηχανών (n_machines), τους χρόνους κάθε εργασίας σε κάθε μηχανή (p) και μια ακολουθία εκτέλεσης των εργασιών (solution).

Παρακάτω παρουσιάζεται η εκτέλεση της συνάρτησης π.χ. schedule(n_jobs, n_machines, p, solution) με ένα παράδειγμα. Έστω ότι η συνάρτηση παίρνει ως ορίσματα τις τιμές n_jobs = 5, n_machines = 3, p (πίνακας 2) και Solution=1,4,0,3. Αρχικά υπολογίζουμε τους χρόνους για την πρώτη εργασία τις ακολουθίας που είναι η εργασία 1. Για κάθε μηχανή της εργασίας 1 προσθέτουμε τον χρόνο της προηγούμενης μηχανής. Η δεύτερη εργασία της ακολουθίας είναι η εργασία 4. Αρχικά αθροίζουμε τον χρόνο της προηγούμενης εργασίας και της τρέχουσας στην

μηχανή 0. Στη συνέχεια αθροίζουμε τον μεγαλύτερο χρόνο ανάμεσα στον χρόνο της τρέχουσας εργασίας στην προηγούμενη μηχανή και τον χρόνο της προηγούμενης εργασίας στην τρέχουσα μηχανή (max(C[j,i-1], C[solution[idx-1],i]).

	p		
0	54	79	16
1	83	3	89
2	15	11	49
3	71	99	15
4	77	56	89

	C (1 - 4 - 0 – 3)		
1	83	83+3 = 86	86 + 89 = 175
4	77 + 83 = 160	160 + 56 = 216	216 + 89 = 305
0	160+54 = 214	216 + 79 = 295	305 + 16 = 321
3	214 + 71 = 285	295 + 99 = 394	394 + 15 = **409**

Πίνακας 2 Εκτέλεση μιας Ακολουθίας του Αλγόριθμου NEH

```
def schedule(n_jobs, n_machines, p, solution):
  C = np.zeros((n_jobs, n_machines)) #Δημιουργεί έναν πίνακα με μέγεθος n_jobs x n_machines και τον γεμίζει με μηδενικά
  for idx, j in enumerate(solution): #Για κάθε θέση της ακολουθίας (όπου idx->η θέση του πίνακα ακολουθίας και όπου j->η τιμή της θέσης)
    for i in range(n_machines):
      if idx == 0: #Εάν εξετάζουμε την πρώτη εργασία της ακολουθίας
        if i==0: #Εάν βρισκόμαστε στην πρώτη μηχανή
          C[j,i] = p[j,i] #Αντιστοιχούμε την αρχική τιμή
        else:
          C[j,i] = C[j,i-1] + p[j,i] #Για τις επόμενες μηχανές και την πρώτη εργασία, αθροίζουμε την τρέχουσα τιμή με την προηγούμενη τιμή
       else:
          if i == 0: #Εάν δεν βρισκόμαστε στην πρώτη εργασία της ακολουθίας αλλά στην πρώτη μηχανή
            C[j,i] = C[solution[idx-1],i] + p[j,i] # Αθροίζουμε την τρέχουσα τιμή με την προηγούμενη εργασία στην ίδια μηχανή
       else:
#Αλλιώς επιλέγουμε την μεγαλύτερη τιμή ανάμεσα στην τιμή του χρόνου της εργασίας στην προηγούμενη μηχανή και του χρόνου της προηγούμενης εργασίας στην ίδια μηχανή και το αθροίζουμε με τον τρέχων χρόνο εργασίας
          C[j,i] = max(C[j,i-1], C[solution[idx-1],i]) + p[j,i]
    return C
```

```
# Χρόνος ολοκλήρωσης μιας συγκεκριμένης ακολουθίας
def makespan(job_sequence, C): # Με τους παρακάτω τελεστές παίρνουμε την τελευταία τιμή του πίνακα με τους χρόνους που αντιστοιχεί στον τελικό χρόνο εκτέλεσης
    return C[job_sequence[-1], -1]
```

Κώδικας 2 Ο Αλγόριθμος NEH (2)

5.1.1.1 Βελτιωμένος Αλγόριθμος NEH_F

Ο βελτιωμένος αλγόριθμος NEH_F, εξετάζει σε κάθε βήμα μια εργασία σύμφωνα με την ακολουθία που έχει προκύψει από την ταξινόμηση των χρόνων εκτέλεσης κάθε εργασίας σε κάθε μηχανή. Ονομάζεται «βελτιωμένος αλγόριθμος NEH» καθώς καταφέρνει να μειώσει την πολυπλοκότητα του αλγόριθμου από $O(n^3m)$ σε $O(n^2m)$ όπως εξηγείται και παρακάτω. Παρόμοια με τον αλγόριθμο NEH ταξινομεί για αρχή τις εργασίες σύμφωνα με τους χρόνους εκτέλεσης σε κάθε μηχανή. Στην συνέχεια δημιουργεί τρεις πίνακες και εκτελεί τα εξής βήματα:

ΒΗΜΑ 1ο Υπολογίζει τον συντομότερο χρόνο $e_{i,j}$ της εργασίας i στην μηχανή j
$e_{0j} = 0, e_{i0} = 0$ $e_{ij} = \max \{ e_{i, j-1} , e_{i-1, j} \} + t_{ij}$ $(i = 1, \dots , k-1) (j = 1, \dots , m)$
ΒΗΜΑ 2ο Υπολογίζει την ουρά qij δηλ. την διάρκεια από την ώρα έναρξης της i-οστής εργασίας στην j-οστή μηχανή μέχρι το τέλος των εργασιών.
$q_{kj} = 0, q_{i, m+1} = 0$ $q_{ij} = \max \{ q_{i, j+1} , e_{i+1, j} \} + t_{ij}$ $(i = k-1, \dots ,1) (j = m, \dots , 1)$
ΒΗΜΑ 3ο Υπολογίζει τον συντομότερο χρόνο fij στην j-οστή μηχανή της k-οστής εργασίας που εισάγεται στην i-οστή θέση.
$f_{i0} = 0,$ $f_{ij} = \max \{ f_{i, j-1} , e_{i-1, j} \} + t_{kj}$ $(i = 1, \dots , k) (j = 1, \dots , m)$
ΒΗΜΑ 4ο Καταχωρείται η τιμή του χρόνου MAKESPAN M_i όταν προστεθεί η εργασία k στην i-οστή θέση.
$M_i = \max_j (f_{ij} + q_{ij})$ $(i = 1, \dots , k) (j = 1, \dots , m)$

Πίνακας 3 Βήματα Βελτιωμένου Αλγόριθμου NEH_F

Δημιουργεί τρεις πίνακες e, q και f:

e : πίνακας στον οποίο καταχωρούνται οι εργασίες σύμφωνα με την τρέχουσα ακολουθία, έστω ότι έχουμε την ακολουθία 4 – 3 – 1 – 0 - 2 και εξετάζουμε την εργασία 3 ως προς την εργασία 4.

q : πίνακας στον οποίο καταχωρούνται οι εργασίες από το τέλος της ακολουθίας προς την αρχή, ξεκινώντας πάντα από την τελευταία εργασία των εργασιών που εξετάζονται.

f : πίνακας στον οποίο καταχωρούνται οι συντομότεροι χρόνοι της εξεταζόμενης εργασίας σε σχέση με τις υπόλοιπες εργασίες.

Στον πίνακα 4 περιγράφεται με ένα παράδειγμα ο βελτιωμένος αλγόριθμος NEH_F. Έχουμε τον πίνακα p με τους χρόνους (t). Η ακολουθία έναρξης είναι η ακολουθία 4 – 3 – 1 – 0 – 2. Τοποθετούμε την εργασία που εμφανίζεται πρώτη στην ακολουθία στον πίνακα e, στην συνέχεια τοποθετούμε την εργασία 3 που εξετάζεται στον πίνακα f. Τέλος τοποθετούμε την ουρά των εργασιών που επεξεργαζόμαστε δηλ. των εργασιών 4 και 3 εκτός της εργασίας που εξετάζουμε. Στην ουρά μπαίνει η εργασία 4 με αντίστροφη φορά. Μετά τον πρώτο κύκλο η ακολουθία γίνεται 3 – 4 – 1 – 0 – 2, και εξετάζεται η εργασία 1.

4 - 3 - 1 - 0 - 2

p			e			q			f			sum		
54	79	16	77	133	222	222	145	89	71	170	185	293	**315**	274
83	3	89							148	247	262	148	247	**262**
15	11	49										0	**0**	0
71	99	15										0	0	**0**
77	56	89												

3 - 4 - 1 - 0 - 2

p			e			q			f			sum		
54	79	16	71	170	185	315	244	104	83	86	175	**398**	330	279
83	3	89	148	226	315	222	145	89	154	173	274	**376**	318	363
15	11	49							231	234	404	231	234	**404**
71	99	15										0	0	**0**
77	56	89												

3 - 1 - 4 - 0 - 2

p			e			q			f			sum		
54	79	16	71	170	185	376	292	193	54	133	149	**430**	425	342
83	3	89	154	173	274	305	181	178	125	249	265	430	430	**443**
15	11	49	231	287	376	222	145	89	208	287	303	430	**432**	392
71	99	15							285	366	392	285	366	**392**
77	56	89												

0 - 3 - 1 - 4 - 2

p			e			q			f			sum		
54	79	16	54	133	149	430	371	209	15	26	75	**445**	397	284
83	3	89	125	232	247	376	292	193	69	144	198	**445**	436	391
15	11	49	208	235	336	305	181	178	140	243	296	445	424	**474**
71	99	15	285	341	430	222	145	89	223	246	385	445	391	**474**
77	56	89							300	352	479	300	352	**479**

Πίνακας 4 Αναπαράσταση Βελτιωμένου Αλγόριθμου NEH_F

Μετά από τα παραπάνω περάσματα, καταλήγουμε στον τελευταίο πίνακα για τον οποίο ακολουθεί μια αναλυτικότερη περιγραφή:

Σε αυτόν τον κύκλο εξετάζουμε την εργασία 2.

Ξεκινάμε και τοποθετούμε στον πίνακα e τις εργασίες 0 , 3 , 1 και 4 (προσέχουμε να αθροίσουμε τον μεγαλύτερο προηγούμενο χρόνο).

Τοποθετούμε στον πίνακα q τις εργασίες ξεκινώντας από το τέλος προς την αρχή εκτός της εργασίας 2.

Τοποθετούμε στον πίνακα f τους χρόνους της εξεταζόμενης εργασίας 2 σε σχέση με τις υπόλοιπες εργασίες. Στην πρώτη γραμμή τοποθετούμε την εργασία 2 αθροίζοντας τους χρόνους. Στην δεύτερη γραμμή τοποθετούμε τους χρόνους της εργασίας 2 σε σχέση με την πρώτη γραμμή του πίνακα e δλδ. της πρώτης εργασίας της τρέχουσας ακολουθίας και λαμβάνουμε υπόψη τον εξής κανόνα

$$f_{i,j} = \max(f_{i,j-i} , e_{i-1,j}) + p_{k,j}$$

Π.χ. για την τοποθέτηση της τιμής $p_{2,1}$ (11) στην θέση $f_{2,1}$ συγκρίνουμε τις τιμές $f_{2,0}$ (140) και $e_{1,1}$ (232), παίρνουμε την μεγαλύτερη τιμή και αθροίζουμε την τιμή $p_{2,1}$, άρα στην θέση $f_{2,1}$ τοποθετούμε την τιμή 232+11 = 243.

Στη συνέχεια αθροίζουμε τους πίνακες q και f και κρατάμε για κάθε εργασία την μεγαλύτερη τιμή. Η θέση στην οποία βρίσκεται η ελάχιστη μέγιστη τιμή αντιστοιχεί στην θέση στην οποία θα τοποθετηθεί στην ακολουθία η εξεταζόμενη εργασία. Στην περίπτωση του παραδείγματος η εξεταζόμενη εργασία 2 θα τοποθετηθεί στην θέση 1 ή 2 καθώς εμφανίζουν τις ελάχιστες μέγιστες τιμές μεταξύ των 445, 445 , 474 και 474. Ο αλγόριθμος επιλέγει την πρώτη ελάχιστη τιμή, άρα η εργασία 2 θα τοποθετηθεί στην θέση 0 της ακολουθίας.

Ακολουθεί ο βελτιωμένος αλγόριθμος NEH_F. Δημιουργήθηκε το αρχείο neh_f.py το οποίο περιέχει τις παρακάτω συναρτήσεις:

- `job_sum_time(job_j, n_machines, p)` : υπολογίζει τον συνολικό χρόνο εκτέλεσης κάθε εργασίας σε όλες τις μηχανές
- `start_order_neh(n_jobs, n_machines, p)` : Ταξινομεί τις εργασίες σύμφωνα με τους συνολικούς χρόνους εκτέλεσης.
- `insertion(seq, i_position, timi)` : Δημιουργεί τους συνδυασμούς ακολουθιών για κάθε εργασία.
- `neh_f(n_jobs, n_machines, p)` : Ο κορμός του βελτιωμένου αλγόριθμου NEH_F

Οι συναρτήσεις `job_sum_time(job_j, n_machines, p)` , `start_order_neh(n_jobs, n_machines, p)` και `insertion(seq, i_position, timi)` είναι οι ίδιες με τις συναρτήσεις στο αρχείο neh.py. Θα μπορούσαμε να τις δηλώ-

σουμε μια φορά και να τις χρησιμοποιήσουμε και στην περίπτωση του αλγόριθμου neh_f. Για λόγους ευκολότερης ανάγνωσης του κώδικα επέλεξα να τις επαναλάβω. Παρακάτω παρουσιάζεται η συνάρτηση `neh_f(n_jobs, n_machines, p)` που υλοποιεί τον βελτιωμένο αλγόριθμο neh_f:

```
#Εκτέλεση του αλγόριθμου NEH_f
#Αποθηκέυουμε την ακολουθία αφού την ταξινομήσουμε στην λίστα seq
def neh_f(n_jobs, n_machines, p):
  test='jim'
  seq = start_order_neh(n_jobs, n_machines, p)
  print(seq)
#Η μεταβλητή k παίρνει τιμές από 1 έως n_jobs-1 και η μεταβλητή job παίρνει τιμές από το δεύτερο στοιχείο και μετά της λίστας seq
  for k, job in zip(range(1, n_jobs), seq[1:]):
# e -> Ορίζουμε την δομή για την καταχώρηση
     e = np.zeros((k+1, n_machines+1))
#q->Ορίζουμε την δομή για την καταχώρηση της ουράς των εργασιών (της εργασίας i στην μηχανή j)
     q = np.zeros((k+1, n_machines+1))
# f -> Ορίζουμε την δομή για την καταχώρηση του ελάχιστου χρόνου εκτέλεσης της εργασία k στην θέση i της μηχανής j
     f = np.zeros((k+1, n_machines+1))
# Βρίσκουμε τον ελάχιστο χρόνο για κάθε θέση στην οποία μπορεί να εισαχθεί η εργασία k
# Υπολογίζουμε τον ελάχιστο χρόνο εκτέλεσης, την ουρά και τον σχετικό χρόνο εκτέλεσης
     for i in range(k + 1):
        for j in range(n_machines):
           if i < k: #Τοποθέτηση αντίστοιχης εργασίας στον πίνακα e
             e[i, j] = max(e[i, j-1], e[i-1, j]) + p[seq[i], j]
             if i > 0: #Τοποθέτηση αντίστοιχης εργασίας στον πίνακα q
                q[k-i, n_machines-j-1] = max(q[k-i, n_machines-j], q[k-i+1, n_machines-j-1]) + p[seq[k-i], n_machines-j-1] #Τοποθέτηση αντίστοιχης εργασίας στον πίνακα f
                f[i, j] = max(f[i, j-1], e[i-1, j]) + p[job, j]
# Partial makespans inserting job in i-th position
# Υπολογίζεται το άθροισμα των πινάκων f και q , επιλέγεται αυτό με την μεγαλύτερη τιμή ανά γραμμή και αποθηκεύεται στον πίνακα Mi
      Mi = np.amax(f + q, axis=1)[:-1]
# Επιλέγει την θέση του ελάχιστου από τα παραπάνω μέγιστα αθροίσματα.
# Στην συγκεκριμένη θέση θα τοποθετηθεί η τρέχουσα εργασία
      position = np.where(Mi == Mi.min())[0][0]
#makespan = int(Mi[position])
#Εισάγουμε την τρέχουσα εργασία στην θέση που ελαχιστοποιεί το makespan
       seq.remove(job)
       seq.insert(position, job)
       print(seq)
    return seq
```

Κώδικας 3 Βελτιωμένος Αλγόριθμος NEH_F (1)

Ο αλγόριθμος NEH έχει πολυπλοκότητα $O(n^3m)$ καθώς διατρέχει 3 φορές το σύνολο των εργασιών για κάθε μηχανή, ενώ ο βελτιωμένος αλγόριθμος NEH_f διατρέχει 2 φορές το σύνολο των εργασιών για κάθε μηχανή. Κατά την εκτέλεση των δύο αλγορίθμων, έγιναν μετρήσεις στους χρόνους εκτέλεσης. Παρακάτω παρουσιάζεται ο τρόπος κλήσης των αλγορίθμων και το αποτέλεσμα:

```
file = lf.load_files()
print(file)
n_jobs, n_machines, p = lf.read_pfsp_instance(file)

#n_jobs, n_machines, p = lf.read_pfsp_instance('./toy')

start1 = time.time()
job_sequence_neh = neh.neh(n_jobs, n_machines, p)
end1 = time.time()
C = sm.schedule(n_jobs, n_machines, p, job_sequence_neh)
bestTimeNeh = sm.makespan(job_sequence_neh, C)
print("*****************************************   NEH  ************************************
************************")
print("Η αποδοτικότερη ακολουθία με τον αλγόριθμο NEH είναι:%s"%job_sequence_neh)
print("Ο καλύτερος χρόνος σύμφωνα με τον αλγόριθμο NEH είναι:%i"%bestTimeNeh)
print("ΧΡΟΝΟΣ ΕΚΤΕΛΕΣΗΣ NEH :", (end1-start1) * 10**3, "ms")
print("-------------------------------------------------------------------------------------
-------------------------")
start2 = time.time()
job_sequence_neh_f = neh_f.neh_f(n_jobs, n_machines, p)
end2 = time.time()
C = sm.schedule(n_jobs, n_machines, p, job_sequence_neh_f)
bestTimeNeh_F = sm.makespan(job_sequence_neh_f, C)
print("*****************************************  ΒΕΛΤΙΩΜΕΝΟΣ   NEH_f  ************************
************************")
print("Η αποδοτικότερη ακολουθία με τον ΒΕΛΤΙΩΜΕΝΟ αλγόριθμο NEH_F είναι:%s"%job_sequence_neh_f)
print("Ο καλύτερος χρόνος σύμφωνα με τον αλγόριθμο NEH είναι:%i"%bestTimeNeh_F)
print("ΧΡΟΝΟΣ ΕΚΤΕΛΕΣΗΣ NEH_F :", (end2-start2) * 10**3, "ms")
```

Κώδικας 4 Σύγκριση NEH με NEH_F

Όπως μπορούμε να δούμε στην εικόνα 4 και οι δύο αλγόριθμοι καταλήγουν στην ίδια ακολουθία και στο ίδιο makespan. Ο αλγόριθμος NEH εκτελέστηκε σε 13,96 ms ενώ ο βελτιωμένος αλγόριθμος NEH_f σε 3,02 ms. Οι δοκιμές έγιναν σε ένα μηχάνημα με επεξεργαστή Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz 2.81 GHz και 16GB μνήμη RAM.

```
****************************************   NEH   ************************************************************
Η αποδοτικότερη ακολουθία με τον αγόριθμο NEH είναι:[2, 16, 8, 7, 14, 13, 10, 15, 12, 18, 5, 3, 4, 17, 0, 1, 9, 6, 19, 11]
Ο καλύτερος χρόνος σύμφωνα με τον αγόριθμο NEH είναι:1286
ΧΡΟΝΟΣ ΕΚΤΕΛΕΣΗΣ NEH : 13.962268829345703 ms
-------------------------------------------------------------------------------------------------------------
**************************************** ΒΕΛΤΙΩΜΕΝΟΣ  NEH_f ************************************************
Η αποδοτικότερη ακολουθία με τον ΒΕΛΤΙΩΜΕΝΟ αγόριθμο NEH_F είναι:[2, 16, 8, 7, 14, 13, 10, 15, 12, 18, 5, 3, 4, 17, 0, 1, 9, 6, 19, 11]
Ο καλύτερος χρόνος σύμφωνα με τον αγόριθμο NEH είναι:1286
ΧΡΟΝΟΣ ΕΚΤΕΛΕΣΗΣ NEH_F : 3.0159950256347656 ms
-------------------------------------------------------------------------------------------------------------
PS D:\DEV\2023f_aads_prj>
```

Εικόνα 4 Αποτελέσματα Σύγκρισης NEH ΜΕ NEH_F

5.1.3 Ο Αλγόριθμος TABOO Search

Ένας επιπλέον ευρετικός αλγόριθμος ο οποίος βασίζεται στην τεχνική TABOO Search, ξεκινάει με μια τυχαία ακολουθία π.χ. [0,1,2,3,4 …] και στη συνέχεια δημιουργεί νέες ακολουθίες αλλάζοντας θέσεις σε 2 στοιχεία της ακολουθίας. Ορίζουμε για πόσες επαναλήψεις θα επαναλάβει την διαδικασία δημιουργίας γειτόνων όπως ονομάζονται οι ακολουθίες που δημιουργούνται. Εάν ο αριθμός των εργασιών ενός στιγμιότυπου είναι n τότε σε κάθε κύκλο δημιουργούνται $(n-1)^2$ ακολουθίες. Κατά την διάρκεια των κύκλων σίγουρα επαναλαμβάνονται ίδιες ακολουθίες.

Κάθε ακολουθία που εξετάζεται αποθηκεύεται στην λίστα TABOO για να μην εξετάζουμε ίδιες λύσεις συνεχώς. Βέβαια η λίστα TABOO ορίζεται να έχει ένα συγκεκριμένο μέγεθος, οπότε όταν γεμίσει αφαιρεί μια ακολουθία κάθε φορά που προσθέτει μια. Κάθε φορά που εξετάζουμε μια ακολουθία ελέγχουμε εάν βρίσκεται στην λίστα TABOO. Εάν δεν υπάρχει στην λίστα TABOO υπολογίζουμε το makespan και ελέγχουμε εάν είναι καλύτερο από το μέχρι τώρα καλύτερο makespan. Κρατάμε πάντα το καλύτερο makespan και την καλύτερη ακολουθία σε μια μεταβλητή αντίστοιχα.

Επειδή η αναζήτηση TABOO συχνά μπορεί να κολλήσει σε μια τιμή, προσθέτουμε έναν επιπλέον μηχανισμό τερματισμού έτσι ώστε όταν βρεθεί η καλύτερη λύση για πάνω από 10 φορές στη σειρά ο αλγόριθμος τερματίζεται.

```
def taboo_search(n_jobs, n_machines, p, max_iterations):
  stackCounter = 0
#Αρχικοποιούμε τις μεταβλητές
    current_sequence = list(range(n_jobs))                  #Τρέχουσα ακολουθία
    best_sequence = current_sequence.copy()                 #Καλύτερη ακολουθία
    C = sm.schedule(n_jobs, n_machines,p, best_sequence)
#Υπολογίζει το makespan για την καλύτερη ακολουθία. Στην αρχή είναι η ακολουθία 1,2,3,4 .....
   best_makespan = C[-1][-1] #Αρχικοποιεί τον καλύτερο χρόνο makespan
```

```
  taboo_list = []  #Λίστα με απαγορευμένες ακολουθίες, για να μην εξετάζουμε συνέχεια τις ίδιες ακολουθίες
  taboo_size = min(n_jobs // 2, 5) # Ορίζεται η λίστα TABOO να έχει μέγεθος όσο οι μισές εργασίες που εξετάζονται. Εάν ο αριθμός των εργασιών είναι πολύ μεγάλος η λίστα θα έχει μέγεθος 5
  for _ in range(max_iterations): # Επαναλαμβάνουμε την εύρεση βέλτιστης λύσης για όσες επανα-λήψεις ορίσαμε.
      neighbors = []  # Γείτονες, αφορά τις ακολουθίες που δημιουργούμε τυχαία για εύρεση λύσης
      for i in range(n_jobs - 1):
        for j in range(i + 1, n_jobs):
         neighbor_sequence = current_sequence.copy()
#Η current_sequence αλλάζει κάθε φορά
#Η ακολουθίες (γείτονες) δημιουργούνται αλλάζοντας δύο στοιχεία στον τρέχων πίνακα ακολουθίας
        neighbor_sequence[i], neighbor_sequence[j] = neighbor_sequence[j], neighbor_sequence[i]
        neighbors.append(neighbor_sequence) #Τοποθετούμε στην λίστα γειτόνων την ακολουθία που δημιουργήσαμε
        neighbors.sort(key=lambda x: sm.makespan_taboo(x, p,n_jobs, n_machines))
# Ταξινομούμε την λίστα με τους γείτονες σύμφωνα με το καλύτερο makespan
      for neighbor in neighbors: #Εξετάζουμε όλους τους γείτονες
        if neighbor not in taboo_list: #Εάν ο γείτονας δεν βρίσκεται στην απαγορευμένη λίστα θα εξεταστεί
          current_sequence = neighbor #Εδώ αλλάζουμε την ακολουθία current για να δημιουργηθούν νέες ακολουθίες κατά την δημιουργία γειτόνων παραπάνω
          current_makespan = sm.makespan_taboo(current_sequence, p, n_jobs, n_machines)
#Υπολογίζουμε το makespan της συγκεκριμένης ακολουθίας (Γείτονα)
          if current_makespan < best_makespan:
#Εάν η συγκεκριμένη ακολουθία έχει καλύτερο makespan από το μέχρι τώρα καλύτερο. Ενημερώνουμε τις μεταβλητές
             best_sequence = current_sequence.copy()
             best_makespan = current_makespan
             stackCounter = 0
          elif current_makespan == best_makespan:
# Κρατάμε έναν μετρητή για να ελέγχουμε πόσες φορές βρίσκουμε τον καλύτερο χρόνο
              stackCounter += 1
             taboo_list.append(neighbor)
# Προσθέτουμε τον γείτονα που εξετάσαμε στην λίστα TABOO
              if len(taboo_list) > taboo_size:
#Εάν το μέγεθος της λίστας έχει φτάσει στο μέγιστο αφαιρούμε έναν γείτονα
                taboo_list.pop(0)
                break
        if stackCounter >= 10:
            break
    return best_sequence, best_makespan # Επιστρέφουμε την καλύτερη ακολουθία με το καλύτερο makespan
```

Κώδικας 5 Ο Αλγόριθμος TABOO SEARCH

Ο αλγόριθμος TABOO Search είναι ένας αλγόριθμος επίλυσης προβλημάτων βελτιστοποίησης. Έχει μεγάλους χρόνους εκτέλεσης καθώς δοκιμάζει πολλούς συνδυασμούς. Στην περίπτωση της λύσης του προβλήματος PFSP, έχει καταφέρει αρκετές φορές να βρει καλύτερη λύση από τους αλγόριθμους NEH. Συγκεκριμένα 7 στις 10 φορές ο αλγόριθμος TABOO έδωσε καλύτερη λύση από τους αλγόριθμους NEH.

5.1.2 Ο Αλγόριθμος NEHedd

Όπως παρουσιάστηκε στην προηγούμενη ενότητα στον αρχικό αλγόριθμο NEH, οι εργασίες ταξινομούνται σε φθίνουσα (μη αύξουσα) σειρά ως προς το άθροισμα των χρόνων επεξεργασίας στις μηχανές, δίνοντας προτεραιότητα στις εργασίες που απαιτούν περισσότερο χρόνο. Στη συνέχεια δημιουργείται ένα χρονοδιάγραμμα στο οποίο προσθέτουμε διαδοχικά κάθε νέα εργασία από την ταξινομημένη λίστα, σε όλες τις πιθανές θέσεις του υπάρχοντος χρονοδιαγράμματος και κρατάμε το χρονοδιάγραμμα με το μικρότερο makespan. Δεδομένου ότι στο πρόβλημά μας λαμβάνονται υπόψη οι ημερομηνίες λήξης, στον αλγόριθμο NEHedd, οι εργασίες ταξινομούνται κατά μη φθίνουσα σειρά των ημερομηνιών λήξης. Η EDD (edd=earliest due date) υποδηλώνει τη συντομότερη ημερομηνία λήξης.

Σύμφωνα με τον αλγόριθμο NEHedd εκτελείται μια αρχική ταξινόμηση των εργασιών σε περισσότερα του ενός εργοστάσια σύμφωνα με τη συντομότερη ημερομηνία λήξης, η οποία υπολογίζεται από τους χρόνους due date [34] . Σε αντίθεση με τον αλγόριθμο NEH, ο οποίος υπολογίζει το ελάχιστο Makespan, ο αλγόριθμος NEHedd υπολογίζει την ελάχιστη καθυστέρηση (Tardiness).

Σον παρακάτω ψευδοκώδικα παρουσιάζονται τα βήματα εκτέλεσης του αλγόριθμου NEHedd.

```
Procedure NEHedd:
Πedd = Ταξινόμηση των n εργασιών κατά αύξουσα σειρά των due dates (πedd).
Για κάθε εργοστάσιο f από 1 έως F:
  Αρχικοποίηση της λύσης για το εργοστάσιο f με κενό σύνολο (πf = ∅).
Για κάθε εργασία j από 1 έως n:
  x = πedd(j) # Εργασία με την j-η μικρότερη due date
  Για κάθε εργοστάσιο f από 1 έως F:
    Δοκιμή της εργασίας x σε όλες τις πιθανές θέσεις της λύσης πf.
      Υπολογισμός της συνολικής καθυστέρησης (Total Tardiness - TT) στο
      εργοστάσιο f για κάθε πιθανή θέση.
    Βρες το εργοστάσιο fmin με τη μικρότερη συνολική καθυστέρηση.
    Πρόσθεσε την εργασία x στο εργοστάσιο fmin στην κατάλληλη θέση που μειώνει τη
    συνολική καθυστέρηση.
Επιστροφή της βέλτιστης λύσης (π).
```

Κώδικας 6 Ο Αλγόριθμος NEHedd

Η εκτεταμένη υπολογιστική αξιολόγηση των ευρετικών μεθόδων για το πρόβλημα

$$\boldsymbol{DF \mid prmu \mid \sum T_j}$$

έχει δείξει ότι ο αλγόριθμος NEHedd αποτελεί μια βασική ευρετική μέθοδος σύμφωνα με τα αποτελέσματα της αποδοτικότητας του. Πιο συγκεκριμένα, περισσότεροι από τους μισούς βελτιωτικούς ευρετικούς ή μεταευρετικούς αλγόριθμους, της τελευταίας τεχνολογίας για το πρόβλημα DPFSP edd χρησιμοποιούν το NEHedd ως αρχική λύση [35].

Παρά την εξαιρετική απόδοση της ευρετικής NEHedd, πιστεύεται ότι θα μπορούσαν να επιτευχθούν πρόσθετες βελτιώσεις με περαιτέρω ανάλυση του υπό εξέταση προβλήματος. Το προς εξέταση πρόβλημα έχει ως στόχο την ελαχιστοποίηση της καθυστέρησης (tardiness) του τερματισμού των εργασιών, μετά την προθεσμία τους. Η φύση αυτού του προβλήματος μπορεί να μοιάζει με διαφορετικά προβλήματα, ανάλογα με τις προθεσμίες των εργασιών για κάθε περίπτωση: Αν όλες οι εργασίες έχουν ίδια προθεσμία (due date), το πρόβλημα μοιάζει με το πρόβλημα ελαχιστοποίησης του makespan. Αν οι προθεσμίες είναι πολύ χαλαρές, η καθυστέρηση μπορεί να είναι μηδενική, οπότε το πρόβλημα μπορεί να είναι πολύ απλό. Από την άλλη αν οι προθεσμίες είναι αυστηρές, το πρόβλημα μοιάζει με την ελαχιστοποίηση του αριθμού των καθυστερημένων εργασιών (lateness). Με τη διενέργεια ανάλυσης αυτών των πιθανών σεναρίων, μπορούν να αποκτηθούν περαιτέρω γνώσεις σχετικά με το πρόβλημα, ώστε να βελτιωθεί η απόδοση της διαδικασίας NEHedd [36].

Για την αποφυγή τέτοιων σεναρίων και για την καλύτερη και εγκυρότερη παρουσίαση των αποτελεσμάτων η παρούσα εργασία χρησιμοποιεί σύνολα δεδομένων του Taillard, όπου εξετάζεται μια πληθώρα χρόνων λήξης για να καλύψει ένα μεγάλο μέρος των λύσεων ως προς το εξέταση πρόβλημα. Όπως παρουσιάζεται και παρακάτω οι χρόνοι προθεσμιών (due dates) του συνόλου δεδομένων που χρησιμοποιείται κατά την εκτέλεση των αλγορίθμων υπολογίστηκαν με την μέθοδο των Hasija και Rajendran (2004) [22] και μας εξασφαλίζουν, ώστε οι προθεσμίες να μην είναι ούτε υπερβολικά αυστηρές αλλά ούτε υπερβολικά χαλαρές.

6. Ανάπτυξη και Υλοποίηση

Οι μεταευρετικοί αλγόριθμοι είναι προηγμένες τεχνικές βελτιστοποίησης που χρησιμοποιούνται για την επίλυση πολύπλοκων προβλημάτων. Η έννοια του μεταευρετικού αλγορίθμου αναφέρεται σε μια κατηγορία αλγορίθμων βελτιστοποίησης που είναι σχεδιασμένοι να βρίσκουν καλές ή κοντά στις βέλτιστες λύσεις σε προβλήματα που δεν μπορούν να λυθούν αποτελεσματικά με κλασικές μεθόδους (π.χ. εξαντλητική αναζήτηση). Οι μεταευρετικοί αλγόριθμοι ξεπερνούν τις παραδοσιακές ευρετικές μεθόδους εισάγοντας μηχανισμούς που τους επιτρέπουν να αποφεύγουν την παγίδευση σε τοπικά βέλτιστα και να εξερευνούν πιο αποτελεσματικά τον χώρο των πιθανών λύσεων [37] .

Κατά τον σχεδιασμό μια μεταευρετικής μεθόδου πρέπει να προτιμάται η απλότητα, τόσο εννοιολογικά όσο και πρακτικά. Φυσικά, πρέπει επίσης να οδηγεί σε αποτελεσματικούς αλγόριθμους. Εάν σκεφτούμε μια μεταευρετική ως μια απλή κατασκευή για την καθοδήγηση (ειδικών για το πρόβλημα) ευρετικών, θα ήταν ιδανικό η μεταευρετική μέθοδος που θα χρησιμοποιηθεί να είναι ανεξάρτητη του προς επίλυση προβλήματος έτσι ώστε να πετύχουμε μια γενικότητα του αλγόριθμου. Με αποτέλεσμα να μπορούν να εφαρμοστούν σε μεγάλη ποικιλία προβλημάτων, ανεξαρτήτως του συγκεκριμένου τους πεδίου ή δομής. Μιας και το όριο μεταξύ ευρετικών και μεταευρετικών μεθόδων δεν είναι πάντα σαφές, διατρέχουμε τον κίνδυνο να χάσουμε την απλότητα και την γενικότητα ενός μεταευρετικού αλγόριθμου. Για την αντιμετώπιση αυτού του προβλήματος αποσυνθέτουμε έναν μεταευρετικό αλγόριθμο σε μικρότερα ανεξάρτητα μέρη. Κάθε μέρος καλείται να επιλύσει μέρος του προβλήματος προσπαθώντας να ξεχωρίσει την γενικότερη γνώση από την γνώση του προβλήματος. Τέλος στο μέτρο του δυνατού, προτιμούμε να αφήσουμε ανέγγιχτο το ενσωματωμένο ευρετικό σύστημα το οποίο θα περιέχει και την σχετική με το πρόβλημα γνώση, αντιμετωπίζοντας το ως ρουτίνα «μαύρου κουτιού» [38]. Οι μεταευρετικοί αλγόριθμοι συνήθως δεν εγγυώνται το απόλυτο βέλτιστο, αλλά βρίσκουν καλές λύσεις μέσα σε εύλογο χρόνο. Μερικά παραδείγματα μεταευρετικών αλγορίθμων αποτελούν οι παρακάτω αλγόριθμοι:

- **Simulated Annealing (Προσομοιωμένη Ανόπτηση)**: Βασίζεται σε στατιστικές μεθόδους και την ιδέα της προσομοίωσης της διαδικασίας ανόπτησης μετάλλων, όπου το σύστημα επιτρέπει σταδιακές μεταβολές στη λύση με πιθανότητα αποδοχής χειρότερων λύσεων για να αποφύγει τοπικά βέλτιστα.
- **Genetic Algorithms (Γενετικοί Αλγόριθμοι)**: Βασισμένοι στη φυσική επιλογή και τη γενετική, δημιουργούν λύσεις με αναπαραγωγή, μετάλλαξη και επιλογή με βάση την "καταλληλότητα" των λύσεων, προσπαθώντας να προσομοιώσουν τη φυσική εξέλιξη.
- **Iterated Local Search ILS (Επαναληπτική Τοπική Αναζήτηση)**: Ξεκινά από μια λύση και εφαρμόζει επαναλαμβανόμενες διαταραχές της τρέχουσας λύσης και τοπική αναζήτηση για να ανακαλύψει καλύτερες λύσεις.

6.1 Iterated Local Search (ILS)

Η μέθοδος **Iterated Local Search (ILS)** είναι μια μεταευρετική μέθοδος βελτιστοποίησης που συνδυάζει την αναζήτηση τοπικών βελτιώσεων με συστηματικές διαταραχές της λύσης, ώστε να αποφευχθεί το πρόβλημα της παγίδευσης σε τοπικά βέλτιστα σημεία. Η μέθοδος αυτή εφαρμόζεται σε προβλήματα όπου οι λύσεις βελτιώνονται μέσω επαναλαμβανόμενης βελτίωσης μικρών τοπικών τροποποιήσεων.

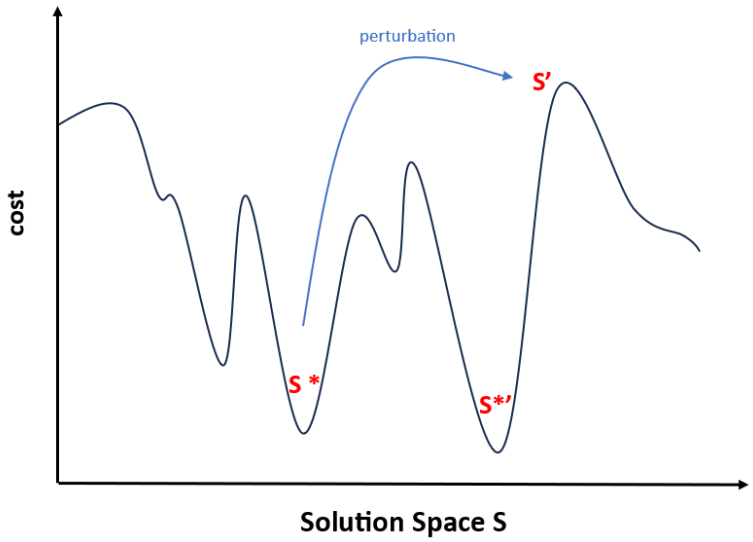
Η βασική ιδέα που διέπει την μέθοδο επαναλαμβανόμενης τοπικής αναζήτησης (ILS) είναι να επιστρέφονται οι υποψήφιες λύσεις ενός προβλήματος, από κάποιον υποκείμενο αλγόριθμο, συνήθως έναν ευρετικό μηχανισμό τοπικής αναζήτησης και όχι στον πλήρη χώρο όλων των λύσεων. Η προκύπτουσα συμπεριφορά αναζήτησης μπορεί να χαρακτηριστεί ως επαναληπτική δημιουργία μιας αλυσίδας λύσεων αυτού του ενσωματωμένου αλγορίθμου. Το αποτέλεσμα είναι επίσης ένας εννοιολογικά απλός μεταευρετικός αλγόριθμος, ο οποίος ωστόσο έχει οδηγήσει σε αλγορίθμους τελευταίας τεχνολογίας για πολλά υπολογιστικά δύσκολα προβλήματα. Στην πραγματικότητα, πολύ καλές επιδόσεις επιτυγχάνονται συχνά ήδη από μάλλον απλές υλοποιήσεις της μεταευρετικής. Επιπλέον, η αρθρωτή αρχιτεκτονική της επαναλαμβανόμενης τοπικής αναζήτησης την καθιστά πολύ κατάλληλη για μια μεθοδολογία βελτιστοποίησης αλγορίθμων όπου, σταδιακά, η απόδοση του αλγόριθμου μπορεί να βελτιστοποιηθεί περαιτέρω [39] .

Ένας αλγόριθμος ILS δημιουργεί επαναληπτικά μια ακολουθία λύσεων που δημιουργούνται από το ενσωματωμένο ευρετικό, οδηγώντας σε πολύ καλύτερες λύσεις από ό,τι αν χρησιμοποιούσε επαναλαμβανόμενες τυχαίες δοκιμές αυτού του ευρετικού.

Τα βήματα που εκτελεί ένας αλγόριθμος ILS, μπορούν να συνοψιστούν ως εξής:

1. **Αρχική Λύση**: Δημιουργία μιας τυχαίας αρχικής λύσης.
2. **Τοπική Αναζήτηση**: Εφαρμογή τοπικής αναζήτησης (π.χ., hill climbing, simulated annealing) στην αρχική λύση για να βελτιωθεί.
3. **Διαταραχή (Perturbation)**: Μια διαταραχή εφαρμόζεται στην τρέχουσα λύση για να αλλάξει ριζικά η λύση (έξοδος από το τοπικό μέγιστο).
4. **Επανάληψη Τοπικής Αναζήτησης**: Εφαρμογή τοπικής αναζήτησης στην τροποποιημένη λύση.
5. **Αποδοχή**: Σύγκριση της νέας λύσης με την καλύτερη που έχουμε βρει μέχρι στιγμής, και αντικατάστασή της αν είναι καλύτερη.
6. **Επανάληψη**: Επαναλαμβάνουμε τα βήματα 3-5 μέχρι να ικανοποιηθεί κάποιο κριτήριο τερματισμού (π.χ. μέγιστος αριθμός επαναλήψεων ή χρονικός περιορισμός).

Η ιδέα είναι ότι η τοπική αναζήτηση θα φέρει μια λύση κοντά σε ένα τοπικό μέγιστο, και η διαταραχή επιτρέπει την εξερεύνηση νέων περιοχών του χώρου λύσεων.



Εικόνα 5 Local Search

Θα μπορούσαμε να περιγράψουμε την μέθοδο ILS ως εξής: Εξερευνώντας τον χώρο λύσεων S, ξεκινάμε από μια κατάσταση s*, για να καταλήξουμε με μια αλλαγή ή διαταραχή perturbation σε μια ενδιάμεση κατάσταση s'. Εφαρμόζοντας τοπική αναζήτηση στην s' καταλήγουμε στην s*'. Έπειτα εκτελείται έλεγχος ακαταλληλότητας στην s*'. Εάν η s*' αποτελεί μια βελτιωμένη λύση γίνεται το επόμενο στοιχείο προς διερεύνηση, σε διαφορετική περίπτωση επιστρέφουμε στην s*. Παρακάτω παρουσιάζεται ο ψευδοκώδικας της μεθόδου ILS όπως περιγράφεται και από τους Khare και Agrawal (2020) [23]:

```
Procedure ILS
        s0 = GenerateInitialSolution //π.χ. NEHedd για το DPFSPDD
        s* = LocalSearch(s0)
        repeat
                s'     = Perturbation(s*, history)
                s*' = LocalSearch(s')
                s*     = AcceptanceCriterion(s*, s*', history)
        until termination condition metend
```

Κώδικας 7 Iterated Local Search

Αρχικώς χρησιμοποιούμε έναν άπληστο αλγόριθμο για να καταλήξουμε στην αρχική λύση του προβλήματος. Για τα προβλήματα χρονοπρογραμματισμού (DPFSP) ο αλγόριθμος NEH φαίνεται να είναι η ευρετική κατασκευή με τις καλύτερες επιδόσεις. Προφανώς ο αλγόριθμος ILS θα μπορούσε να ξεκινήσει από οποιαδήποτε τυχαία αρχική

λύση. Ωστόσο για μικρούς χρόνους εκτέλεσης σε μεγάλο σύνολο δεδομένων παρατηρήσαμε ότι με την χρήση του αλγόριθμου NEH επιτυγχάνεται καλύτερη ποιότητα λύσης [3] .

Για την εφαρμογή του ILS σε προβλήματα DPFSP πρέπει να οριστούν τρεις γενικοί τελεστές, οι οποίοι περιγράφονται συνοπτικά παρακάτω:

Perturbation: Για την διαταραχή εφαρμόσαμε απλές τροποποιήσεις που δεν μπορούν εύκολα να αντιστραφούν από τον αλγόριθμο τοπικής αναζήτησης. Η ανταλλαγή σε τρεις ή τέσσερεις θέσεις δεν βελτιώνει τα αποτελέσματα [3] . Τελικά οι μικρές τροποποιήσεις επαρκούν για να αποδώσουν πολύ καλές επιδόσεις.

Local Search: Οι ενέργειες του αλγόριθμου τοπικής αναζήτησης επικεντρώνονται στην (i) ανταλλαγή δύο γειτονικών θέσεων (θέσεις i και i+1), (ii) την ανταλλαγή δύο τυχαίων θέσεων (θέση i και θέση j) και (iii)αφαίρεση στοιχείου στην θέση i και τοποθέτηση στην θέση j. Σύμφωνα με [6] και [33] η μέθοδος (iii) αποτελεί την καλύτερη προσέγγιση του προβλήματος PFSP.

Acceptance Criterion: Ως κριτήριο αποδοχής αποδεχόμαστε την καλύτερη των λύσεων μεταξύ των λύσεων s*’ και s*. Εδώ διατρέχουμε τον κίνδυνο να εγκλωβιστεί ο αλγόριθμος σε καλές λύσεις χωρίς να καταφέρει να εντοπίσει την καλύτερη λύση, θα πρέπει να λάβουμε λοιπόν υπόψη ότι το κριτήριο αποδοχής θα πρέπει να μας επιτρέπει να επισκεφτούμε χώρους λύσεων χειρότερες από την τρέχουσα [40] .

Η πρώτη εφαρμογή του ILS στο PFSP με στόχο το makespan, το οποίο είναι το πιο ευρέως μελετημένο πρόβλημα χρονοπρογραμματισμού PFSP, αναφέρθηκε από τον Stützle [40]. Αυτός ο αλγόριθμος ILS βασίζεται σε μια απλή τοπική αναζήτηση πρώτης βελτίωσης χρησιμοποιώντας τη γειτονιά εισαγωγής, ενώ η διαταραχή αποτελείται από κινήσεις ανταλλαγής, οι οποίες ανταλλάσσουν τις θέσεις δύο γειτονικών εργασιών, και κινήσεις ανταλλαγής, οι οποίες δεν έχουν περιορισμό γειτνίασης. Ο ILS έχει χρησιμοποιηθεί για την επίλυση προβλημάτων flow-shop με άλλους στόχους εκτός του makespan, όπως ο συνολικός χρόνος flow-time. Η ομάδα Pan [39] εφάρμοσαν πρόσφατα τον ILS στο υβριδικό πρόβλημα flow-shop με ημερομηνία λήξης (DUE DATES) και στόχους πρωιμότητας (earliness) και καθυστέρησης (tardiness) στο οποίο πρόβλημα επικεντρώνεται και η έρευνα του συγγράμματος.

6.2 Hybrid Local Search Greedy Algorithm (HYLG)

Ο αλγόριθμος IteratedGreedy (IG) για τον χρονοπρογραμματισμό προτάθηκε για πρώτη φορά από τους Ruiz και Stützle (2007). Όπως αναφέρθηκε, ο αλγόριθμος IG έχει επιτυχώς εφαρμοστεί αρκετές φορές για την ελαχιστοποίηση του makespan του DPFSP στο παρελθόν [13] .

Θέλοντας να τροποποιήσουμε τον αλγόριθμο (IG) και να τον προσαρμόσουμε έτσι ώστε να βελτιστοποιεί προβλήματα συνολικής καθυστέρησης, συμβουλευτήκαμε τον αλγόριθμο που παρουσιάζεται από τους Ankit Khare & Sunil Agrawal [23] και αναπτύξαμε τον αλγόριθμο Hybrid Local Search Greedy (HYLG). Είναι μια στοχαστική μέθοδος βελτιστοποίησης που βασίζεται σε επαναλαμβανόμενες διαδικασίες δημιουργίας και

βελτίωσης λύσεων. Χρησιμοποιείται συχνά για την επίλυση δύσκολων συνδυαστικών προβλημάτων, όπως αυτά που περιλαμβάνουν χρονοπρογραμματισμό, δρομολόγηση, και προβλήματα κατανομής.

Βασική Ιδέα:

Ο αλγόριθμος HYLG επαναλαμβάνει συνεχώς δύο βασικές φάσεις:

1. Κατασκευή (Construction): Δημιουργείται μια αρχική λύση χρησιμοποιώντας μια "άπληστη" στρατηγική (greedy), η οποία σημαίνει ότι κάθε βήμα γίνεται με τέτοιο τρόπο ώστε να βελτιστοποιεί τοπικά την απόδοση.

2. Καταστροφή και Επιδιόρθωση (Destruction and Reconstruction): Σε κάθε επανάληψη, καταστρέφεται ένα μέρος της λύσης και κατόπιν το ξαναχτίζεται με άπληστο τρόπο. Η διαδικασία αυτή επιτρέπει στον αλγόριθμο να εξερευνά διαφορετικά μονοπάτια, αποφεύγοντας τοπικά ελάχιστα, και να προσπαθεί να βρει μια καλύτερη λύση από την αρχική.

Ο αλγόριθμος HYLG εκτελεί τις παρακάτω λειτουργίες:

1.Αρχικοποίηση: Ξεκινάει με μια αρχική λύση, η οποία συνήθως κατασκευάζεται με άπληστη στρατηγική. Αυτή η λύση μπορεί να είναι γρήγορη, αλλά ενδέχεται να μην είναι η καλύτερη δυνατή. Στην περίπτωσή μας χρησιμοποιείται ο αλγόριθμος NEHedd, όπως παρουσιάστηκε παραπάνω.

2.Καταστροφή (Destruction): Εφαρμόζεται μια διαδικασία που αφαιρεί τυχαία ή στρατηγικά ένα υποσύνολο της λύσης. Για παράδειγμα, σε ένα πρόβλημα δρομολόγησης οχημάτων, μπορεί να αφαιρεθούν μερικές στάσεις από τη διαδρομή.

3.Επιδιόρθωση (Reconstruction): Το υποσύνολο της λύσης που ενσωματώνεται ξανά στο σύστημα με κάποιο άπληστο αλγόριθμο. Ο στόχος είναι η επαναδημιουργία μιας λύσης καλύτερης από την προηγούμενη.

4.Βελτίωση: Μετά την καταστροφή και επιδιόρθωση, μπορεί να εφαρμοστούν τοπικές βελτιώσεις (local search) για περαιτέρω βελτιστοποίηση της νέας λύσης.

5. Επανάληψη: Η διαδικασία καταστροφής και επιδιόρθωσης επαναλαμβάνεται πολλές φορές, δημιουργώντας νέες λύσεις. Αν κάποια νέα λύση είναι καλύτερη από την τρέχουσα λύση, η καλύτερη λύση αντικαθιστά την τρέχουσα.

6. Τερματισμός: Ο αλγόριθμος σταματά όταν ικανοποιηθεί κάποιο κριτήριο τερματισμού, όπως το να μην βελτιώνεται η λύση για συγκεκριμένο αριθμό επαναλήψεων ή όταν περάσει προκαθορισμένος χρόνος.

Στην φάση της καταστροφής, ένα ποσοστό της τάξης d των συνολικών θέσεων εργασίας, αφαιρείται τυχαία από την λύση και τοποθετείται σε μια λίστα π_R. Στην φάση της κατασκευής κάθε εργασία από την λίστα π_R δοκιμάζεται σε όλες τις πιθανές θέσεις σε όλα τα εργοστάσια. Το εργοστάσιο και η θέση με την μικρότερη συνολική καθυστέρηση επιλέγεται.

Πλεονεκτήματα:

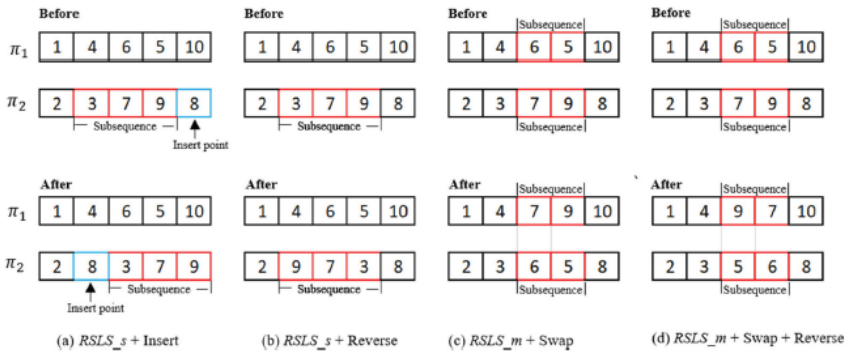
- **Απλότητα:** Είναι ένας απλός και εύκολος στην κατανόηση αλγόριθμος, με μικρό αριθμό παραμέτρων.
- **Ευελιξία:** Μπορεί να προσαρμοστεί εύκολα σε πολλά διαφορετικά προβλήματα.
- **Αποτελεσματικότητα**: Συνήθως παρέχει καλές λύσεις για προβλήματα όπου η άπληστη αναζήτηση δεν επαρκεί για να βρει την καθολικά βέλτιστη λύση.

Για την ενίσχυση της απόδοσης του αλγόριθμου HYLG χρησιμοποιούνται δύο μέθοδοι τοπικής αναζήτησης (local search), Η μέθοδος τοπικής αναζήτησης τυχαίας υποακολουθίας (Random Subsequence Local Search - RSLS) και η μέθοδος τοπικής αναζήτησης τυχαίου σημείου (Random Single Point Local Search – RPLS).

6.2.1 Random Subsequence Local Search

Η RSLS έχει δύο λειτουργίες: multi-factory (RSLS_m) και single-factory (RSLS_s). Ένας τυχαίος αριθμός v μεταξύ 0 και 1 παράγεται για να αποφασιστεί η επιλογή μεταξύ αυτών των δύο τρόπων [23] .

Στο RSLS_s, πραγματοποιείται εισαγωγή ή αντιστροφή εργασιών εντός ενός τυχαία επιλεγμένου εργοστασίου, ενώ στο RSLS_m, η εναλλαγή (ή εναλλαγή και περιστροφή) πραγματοποιείται μεταξύ δύο τυχαία επιλεγμένων εργοστασίων. Ένας τυχαίος αριθμός w μεταξύ 0 και 1 παράγεται για να αποφασιστεί η επιλογή μεταξύ αυτών των εργοστασίων.



Εικόνα 6 Random Subsequence Local Search [23]

Στην Εικόνα 6 παρουσιάζονται 4 διαφορετικά παραδείγματα εφαρμογής του αλγόριθμου RSLS (Random Subsequence Local Search):

a) Στην πρώτη περίπτωση (RSLS_s + Insert) επιλέγεται τυχαία ένα από τα δύο εργοστάσια, στην περίπτωση μας επιλέγεται το εργοστάσιο π_2. Συνεχίζεται επιλέγοντας μια τυχαία υπό ακολουθία (3, 7, 9) η οποία αφαιρείται και επανατοποθετείται μετά από ένα τυχαία επιλεγμένο σημείο (Insert point) (8). Οπότε η νέα ακολουθία του εργοστασίου π_2 είναι η 2, ***8*, 3, 7, 9**.

b) Στην δεύτερη περίπτωση (RSLS_s + Reverse) επιλέγεται τυχαία ένα από τα δύο εργοστάσια, στην περίπτωση μας επιλέγεται το εργοστάσιο π_2. Συνεχίζεται επιλέγοντας μια τυχαία υπό ακολουθία (3, 7, 9) η οποία αντιστρέφεται (9, 7, 3) και επανατοποθετείται στο ίδιο σημείο της ακολουθίας. Οπότε η νέα ακολουθία του εργοστασίου π_2 είναι η 2, **9, 7, 3**, 8.

c) Στην Τρίτη περίπτωση (RSLS_m + Swap) επιλέγονται 2 τυχαία εργοστάσια (π_1 και π_2). Επιλέγονται 2 ίσες υπό ακολουθίες και στα δύο εργοστάσια (π_1 -> 6, 5 και π_2 -> 7, 9), οι οποίες ανταλλάσσονται μεταξύ τους. Οι ακολουθίες των 2 εργοστασίων διαμορφώνονται ως εξής: π_1 -> 1, 4, **7, 9**, 10 και π_2 -> 2, 3, **6, 5**, 8.

d) Στην τέταρτη και τελευταία περίπτωση (RSLS_m + Swap + Reverse) εφαρμόζονται τα βήματα όπως στην Τρίτη περίπτωση αλλά αντιστρέφοντας στο τέλος την σειρά των υπό ακολουθιών. Ως αποτέλεσμα έχουμε τις εξής ακολουθίες για τα δύο επιλεγμένα εργοστάσια: π_1 -> 1, 4, **9, 7**, 10 και π_2 -> 2, 3, **5, 6**, 8

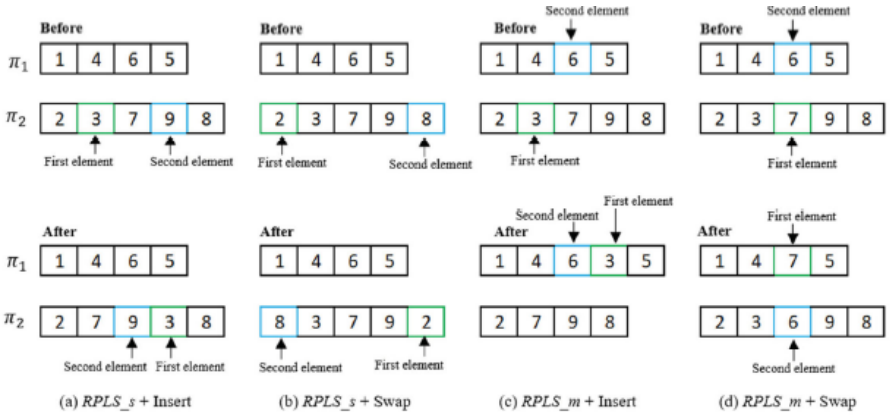
ΨΕΥΔΟΚΩΔΙΚΑΣ RSLS(π)

```
Procedure RSLS(π)
v=rand(), w=rand()
while flag = true
       flag=false
       if v<0.5
              choose a factory randomly;
              if w<0.5
                     choose a subsequence and an insert point;
                     π' = insert the subsequence after the insert point;
              else
                     select a subsequence;
                     π' = reverse the subsequence;
              end if
       else
              select two factories f1 and f2 randomly;
              randomly choose starting point at each factory;
              if w<0.5
                     π' = swap the subsequence;
              else
                     π' = swap and reverse the subsequences;
              end if
       end if
       if (TT(π') < TT(π)) then
              π = π'
              flag = true;
       end if
end while
return π;
end
```

Κώδικας 8 Random Subsequence Local Search

6.2.2 Random Single Point Local Search

Παρόμοια με την μέθοδο RSLS η μέθοδος τοπικής αναζήτησης τυχαίου σημείου RPLS έχει λειτουργίες ενός ή πολλαπλών εργοστασίων. Στην μέθοδο RPLS_s εκτελούνται η διαδικασίες εισαγωγής και ανταλλαγής σε ένα εργοστάσιο ενώ στην μέθοδο RPLS_m εκτελούνται οι παραπάνω διαδικασίες μεταξύ δύο τυχαία επιλεγμένων εργοστασίων. Η απόφαση για την διαδικασία που θα ακολουθηθεί εισαγωγή ή ανταλλαγή εξαρτάται από τυχαία μεταβλητή w. Εάν η μεταβλητή w έχει τιμή μικρότερη από 0,5 επιλέγεται η εισαγωγή διαφορετικά γίνεται ανταλλαγή.



Εικόνα 7 Random Single Point Local Search [23]

Η διαφορά μεταξύ των δύο μεθόδων *Random Subsequence Local Search* και Random Single Point Local Search είναι, ότι στην πρώτη περίπτωση επιλέγουμε υπό ακολουθίες τις οποίες επεξεργαζόμαστε ενώ στην δεύτερη περίπτωση επιλέγουμε μεμονωμένα σημεία. Στην εικόνα 7 παρουσιάζονται οι τέσσερεις περιπτώσεις εφαρμογής του αλγόριθμου RPLS (Random Single Point Local Search):

a) Στην πρώτη περίπτωση (RPLS_s + Insert) επιλέγεται τυχαία ένα από τα δύο εργοστάσια, στην περίπτωση μας επιλέγεται το εργοστάσιο π_2. Συνεχίζεται επιλέγοντας ένα τυχαίο σημείο (First Element) (3) το οποίο αφαιρείται και επανατοποθετείται μετά από ένα δεύτερο τυχαία επιλεγμένο σημείο (Second element) (9). Οπότε η νέα ακολουθία του εργοστασίου π_2 είναι η 2, 7**, 9, 3,** 8.

b) Στην δεύτερη περίπτωση (RPLS_s + Swap) επιλέγεται τυχαία ένα από τα δύο εργοστάσια, στην περίπτωση μας επιλέγεται το εργοστάσιο π_2. Συνεχίζεται επιλέγοντας ένα τυχαίο σημείο (First Element) (2) το οποίο εναλλάσσεται με ένα δεύτερο σημείο (Second Element) (8) . Οπότε η νέα ακολουθία του εργοστασίου π_2 είναι η **8**, 3**,** 7**,** 9, **2**.

c) Στην Τρίτη περίπτωση (RPLS_m + Insert) επιλέγονται 2 τυχαία εργοστάσια (π_1 και π_2). Έπειτα επιλέγεται τυχαία ένα σημείο (First Element) (3) στο εργοστάσιο π_2, το οποίο αφαιρείται από το εργοστάσιο π_2 και

τοποθετείται μετά από ένα τυχαία επιλεγμένο σημείο (Second Element) (6) του εργοστασίου π_1. Οι ακολουθίες των 2 εργοστασίων διαμορφώνονται ως εξής: π_1 -> 1, 4, **6 ,3** ,5 και π_2 -> 2, 7, 9, 8.

d) Στην τέταρτη και τελευταία περίπτωση (RPLS_m + Swap) επιλέγονται όπως και στην τρίτη περίπτωση από ένα σημείο σε κάθε εργοστάσιο (First Element) (7) και (Second Element) (6), τα οποία εναλλάσσονται μεταξύ τους. Ως αποτέλεσμα έχουμε τις εξής ακολουθίες για τα δύο επιλεγμένα εργοστάσια: π_1 -> 1, 4, **7**, 5 και π_2 -> 2, 3, **6**, 9, 8.

6.2.2.1 Simulated Annealing (SA)

Η SA είναι μια μεταευρετική τεχνική βελτιστοποίησης που εισήχθη από τους Kirkpatrick et al. το 1983 για την επίλυση του προβλήματος του πλανόδιου πωλητή (Travelling Salesman Problem - TSP) [41].

Ο αλγόριθμος SA βασίζεται στη διαδικασία ανόπτησης που χρησιμοποιείται στη μεταλλουργία, όπου ένα μέταλλο θερμαίνεται γρήγορα σε υψηλή θερμοκρασία και στη συνέχεια ψύχεται σταδιακά. Σε υψηλές θερμοκρασίες, τα άτομα κινούνται γρήγορα και όταν η θερμοκρασία μειώνεται, μειώνεται και η κινητική τους ενέργεια. Στο τέλος της διαδικασίας ανόπτησης, τα άτομα πέφτουν σε μια πιο διατεταγμένη κατάσταση και το υλικό είναι πιο όλκιμο και ευκολότερο στην επεξεργασία.

Ομοίως, στη SA, μια διαδικασία αναζήτησης ξεκινά με μια κατάσταση υψηλής ενέργειας (μια αρχική λύση) και μειώνει σταδιακά τη θερμοκρασία (μια παράμετρος ελέγχου) μέχρι να φτάσει σε μια κατάσταση ελάχιστης ενέργειας (τη βέλτιστη λύση).

Η SA έχει εφαρμοστεί με επιτυχία σε ένα ευρύ φάσμα προβλημάτων βελτιστοποίησης, όπως το TSP, η αναδίπλωση πρωτεϊνών, η διαμέριση γραφημάτων και ο προγραμματισμός job-shop. Το κύριο πλεονέκτημα της SA είναι η ικανότητά της να ξεφεύγει από τα τοπικά ελάχιστα και να συγκλίνει σε ένα καθολικό ελάχιστο. Η SA είναι επίσης σχετικά εύκολη στην εφαρμογή και δεν απαιτεί εκ των προτέρων γνώση του χώρου αναζήτησης.

6.2.2.2 Κριτήριο Αποδοχής

Το κριτήριο αποδοχής καθορίζει αν μια νέα λύση γίνεται αποδεκτή ή απορρίπτεται. Η αποδοχή εξαρτάται από τη διαφορά ενέργειας μεταξύ της νέας λύσης και της τρέχουσας λύσης, καθώς και από την τρέχουσα θερμοκρασία. Το κλασικό κριτήριο αποδοχής της SA προέρχεται από τη στατιστική μηχανική και βασίζεται στην κατανομή πιθανοτήτων Boltzmann [23].

Ένας τρόπος για να καθορίσουμε μια κατάλληλη αρχική θερμοκρασία T, που χρησιμοποιείται σε έναν αλγόριθμο SA, με βάση τις παραμέτρους του προβλήματος, είναι η παρακάτω εξίσωση:

$$T = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} p_{ij}}{10 * n * m}$$

Εφαρμόζοντας την παραπάνω εξίσωση σε ένα τυχαίο πρόβλημα της μορφής $P = \begin{matrix} 3 & 7 & 2 \\ 6 & 5 & 9 \\ 8 & 4 & 1 \end{matrix}$, έχουμε δηλ. τρεις εργασίες οι οποίες θα εκτελεστούν σε τρεις μηχανές. Αρχικά υπολογίζουμε τον αριθμητή της παραπάνω εξίσωσης ($\sum_{i=1}^{n}\sum_{j=1}^{m} p_{ij}$) 3+7+2+6+5+9+8+4+1=45. Σύμφωνα πάντα με την παραπάνω εξίσωση η αρχική τιμή της θερμοκρασίας T θα πρέπει να οριστεί ως $T = \frac{45}{10*3*3} = \frac{45}{90} = 0{,}5$.

Η επιλογή μιας νέας λύσης που παράγεται από την τοπική αναζήτηση, εξαρτάται από δύο παράγοντες: Εάν η νέα λύση είναι καλύτερη από την προηγούμενη ή εάν ένας τυχαίος αριθμός r[0, 1] είναι μικρότερος από

$$\exp\{-(TT(\pi') - TT(\pi))/T\}$$

Όπου το π’ αναφέρεται στην νέα λύση και το π στην μέχρι στιγμής καλύτερη λύση [23].

Τέλος δίνουμε μια τιμή μεταξύ 0 και 1 στην μεταβλητή d για να ορίσουμε τον αριθμό των εργασιών που θα αφαιρούνται από μια λύση και θα τοποθετούνται ξανά στην καλύτερη θέση της ακολουθίας. Οι τιμές τις d δεν θα πρέπει να είναι ούτε πολύ μικρές αλλά ούτε πολύ μεγάλες. Η επιλογή της τιμής d παρουσιάζεται στην ενότητα 7.2.

Παρακάτω παρουσιάζεται ο ψευδοκώδικας της λύσης:

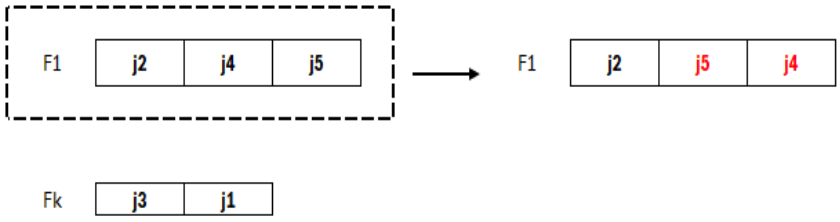
```
Procedure HYLG algorithm
        π=NEHedd;
        π=local search(π);
        π_b = π;
        while (termination criterion not satisfied)
                π'=π;
                for y=1 to (d * n)
                        π_R = remove one job from π' and insert it in π_R;
                end for
                for y=1 to (d * n)
                        x = π_R(y)
                        for f=1 to F
                                Test job x in all possible positions of π'_f;
                                TT_f is the lowest TT obtained at position pf for factory f;
                        end for
                        f_min = arg(min_f=1^F)(TT_f));
                        Insert job x in π'_fmin at position p^fmin resulting in lowest TT;
                end for
                π' = local search (π');
                if (TT(π') < TT(π))
                        π = π';
                        if (TT(π) < TT(π_b)
                                π_β = π
                        end if
                elseif (rand <= exp{-(TT(π') - TT(π))/T})
                        π = π'
                end if
        end while
        return π_b
end
```

Κώδικας 9 Iterated Greedy Algorithm

6.3 MLL Based Mechanism

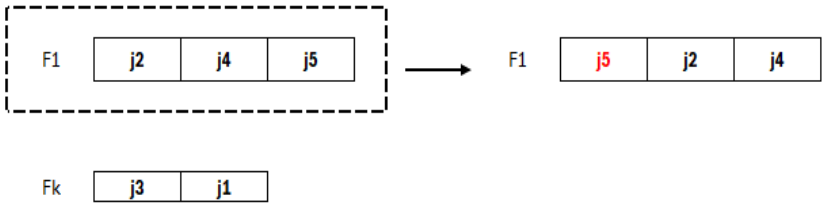
Στοχεύοντας πάντα σε αποδοτικότερες μεθόδους για την βελτιστοποίηση του προβλήματος DPFSP προτείνεται μια υβριδική μέθοδος στην εφαρμογή του αλγόριθμου HYLG εισάγοντας μηχανισμούς με βάση την μέθοδο MLL (Mechanism of Mutation with Lamarckian Learning) προσθέτοντας 4 βασικές και 2 υβριδικές κινήσεις τοπικής αναζήτησης. Η μέθοδος MLL είναι μια μέθοδος που συνδυάζει την ιδέα των γενετικών αλγορίθμων και της διαδικασίας εξέλιξης με μια τροποποιημένη προσέγγιση που εμπνέεται από τις Λαμαρκιανές αρχές μάθησης. Η θεωρία του Λαμάρκ περιγράφει ότι τα επίκτητα χαρακτηριστικά ενός οργανισμού μπορούν να κληροδοτηθούν στους απογόνους του. Στην περίπτωση των αλγορίθμων, αυτό σημαίνει ότι οι βελτιώσεις που γίνονται κατά τη διάρκεια της τοπικής αναζήτησης αποθηκεύονται και μεταβιβάζονται. Ο Meta-Lamarckian-based Iterated Greedy Algorithm (MLIGA) είναι μια εξελικτική τεχνική βελτιστοποίησης, η οποία βασίζεται στον αλγόριθμο iterated greedy (επαναληπτική μέθοδος άπληστων αλγορίθμων) και εμπνέεται από τη θεωρία του **Λαμάρκ** (Lamarckian Evolution). Οι κινήσεις τοπικής αναζήτησης που εξετάζονται είναι οι παρακάτω [42] :

JOB SWAP (JS) — Ανταλλαγή δύο τυχαίων εργασιών σε ένα τυχαία επιλεγμένο εργοστάσιο



Εικόνα 8 MLL Job Swap

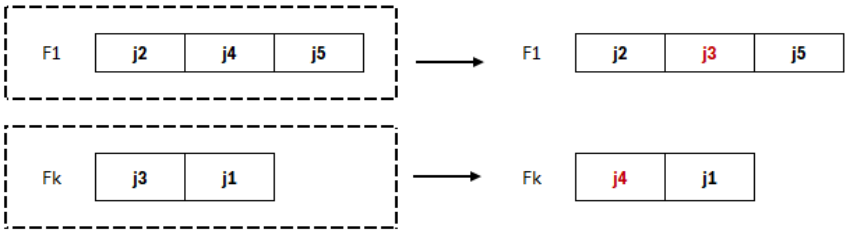
JOB COMPETITIVE INSERTION (JCI) — Επιλέγεται τυχαία μια εργασία ενός εργοστασίου και τοποθετείται σε όλες τις πιθανές θέσεις του ίδιου εργοστασίου με στόχο τη βέλτιστη λύση μειώνοντας την καθυστέρηση.



Εικόνα 9 MLL Job Competitive Insertion

INTER-FACTORY SWAP (IS)

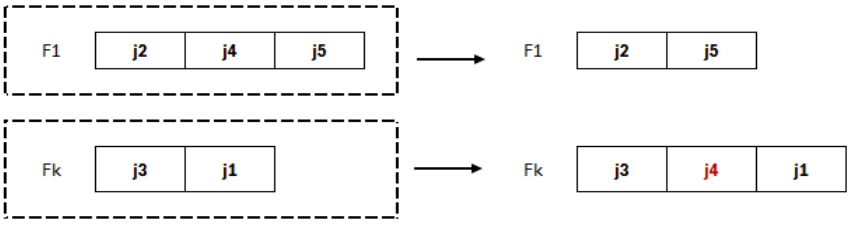
Επιλέγονται δυο εργοστάσια τυχαία και στην συνέχεια μια εργασία από κάθε εργοστάσιο για να προχωρήσουμε στην συνέχεια στην αντιμετάθεση των συγκεκριμένων εργασιών. Η εργασία του 1ου εργοστασίου θα τοποθετηθεί στην θέση της εργασίας που επιλέχθηκε από το δεύτερο εργοστάσιο και η εργασία του 2ου εργοστασίου θα τοποθετηθεί στην θέση της εργασίας που επιλέχθηκε από το πρώτο εργοστάσιο.



Εικόνα 10 MLL Inter-Factory Swap

INTER-FACTORY COMPETITIVE INSERTION (ICI)

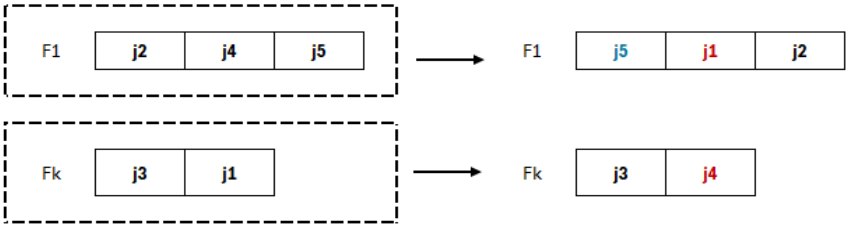
Επιλέγεται μια τυχαία εργασία από ένα τυχαίο εργοστάσιο και τοποθετείται σε όλες τις πιθανές θέσεις ενός δεύτερου τυχαία επιλεγμένου εργοστασίου, για να παραμείνει στην θέση που βελτιώνει την λύση.



Εικόνα 11 MLL Inter-Factory Competitive Insertion

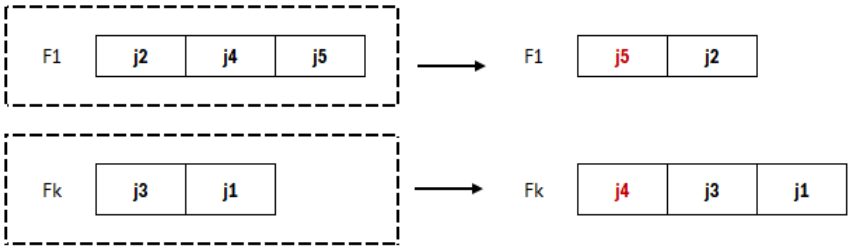
THE HYBRID SWAP (HS)

Οι κινήσεις JS και IS εκτελούνται ταυτόχρονα.



Εικόνα 12 MLL Hybrid Swap

HYBRID COMPETITIVE INSERTION (HCI) Οι κινήσεις JCI και ICI εκτελούνται χωρίς να διακόπτονται.



Εικόνα 13 MLL Hybrid Competitive Insertion

6.4 Hybrid Genetic Algorithm

6.4.1 Genetic Algorithm

Η χρήση γενετικών αλγόριθμων για την επίλυση του προβλήματος DPFSP, έδειξε από τις πρώτες κιόλας προσπάθειες ότι οι γενετικοί αλγόριθμοι μπορούν να είναι πολύ αποτελεσματικοί στην επίλυση του DPFSP, ειδικά όταν συνδυάζονται με ευρετικές μεθόδους και κατάλληλες βελτιστοποιήσεις. Οι γενετικοί αλγόριθμοι έχουν ένα μεγάλο πλεονέκτημα στο ότι μπορούν να εξερευνήσουν μεγάλους χώρους λύσεων και να αποφύγουν τη σύγκλιση σε τοπικά βέλτιστα προσφέροντας καλύτερες λύσεις, σε σχέση μα ανταγωνιστικές μεθόδους, σε αρκετές περιπτώσεις [43] .

Η αρχική προσέγγιση της ομάδας του Reeves [7] χρησιμοποιεί έναν πληθυσμό από πιθανές λύσεις, και μέσω επαναλαμβανόμενων γενεών επιτυγχάνει την βελτιστοποίηση του makespan. Η γενική δομή του γενετικού αλγόριθμου για την επίλυση προβλημάτων flow shop scheduling έχει ως εξής:

- ***Αναπαράσταση λύσης***: Ένα χρωμόσωμα αποτελεί μια σειρά αριθμών που αναπαριστούν μια ακολουθία εργασιών.
- ***Αρχικός πληθυσμός***: Συνήθως ο αλγόριθμος ξεκινάει με ένα τυχαίο σύνολο από ακολουθίες εργασιών.
- ***Διασταύρωση (Crossover)***: Γίνεται ένας συνδυασμός λύσεων, ελπίζοντας σε νέες βελτιωμένες λύσεις.
- ***Μετάλλαξη (Mutation)***: Εφαρμόζονται τυχαίες μικρές αλλαγές στις λύσεις για να διατηρήσουμε την ποικιλομορφία στον πληθυσμό και να αποφευχθεί η σύγκλιση σε τοπικά βέλτιστα.
- ***Επιλογή (Selection)***: Οι καλύτερες λύσεις επιλέγονται για να δημιουργήσουν την επόμενη γενιά ενώ οι λιγότερο καλές λύσεις απορρίπτονται [7] .

Οι Murata, Ishibuchi και Tanaka [43] προχώρησαν σε πιο εξελιγμένες τεχνικές για την βελτίωση της απόδοσης των γενετικών αλγορίθμων. Αρχικά εφάρμοσαν μια βελτιστοποιημένη αρχικοποίηση του πληθυσμού

χρησιμοποιώντας ευρετικές μεθόδους που επιταχύνουν την εύρεση καλών λύσεων. Στην περίπτωση της παρούσας εργασίας χρησιμοποιείται ο αλγόριθμος NEHedd. Στην συνέχεια οι συγγραφείς εισάγουν και αξιολογούν διάφορους μηχανισμούς για τις διασταυρώσεις προσαρμοσμένες σε προβλήματα χρονοπρογραμματισμού. Τέλος διερευνώνται διάφορες στρατηγικές επιλογής των πιο κατάλληλων λύσεων για τις επόμενες γενιές.

Οι Murata, Ishibuchi και Tanaka υπογραμμίζουν τη δύναμη των GAs στην επίλυση πολύπλοκων προβλημάτων και τη δυνατότητα βελτίωσης της απόδοσής τους με την προσαρμογή βασικών στοιχείων του αλγόριθμου.

Ο αλγόριθμος που προτείνει ο Reeves επιδεικνύει τη δύναμη των γενετικών αλγορίθμων στην εύρεση κοντινών βέλτιστων λύσεων σε πολύπλοκα προβλήματα που είναι δύσκολο να επιλυθούν με κλασικές μεθόδους [43] [7] .

6.4.2 Hybrid Genetic Algorithm For The DPFSP

Ένας υβριδικός γενετικός αλγόριθμος συνδυάζει τις βασικές αρχές των γενετικών αλγορίθμων με άλλα ευρετικά εργαλεία. Πρόκειται για μια παραλλαγή των κλασικών γενετικών αλγορίθμων, που ενσωματώνει επιπλέον στρατηγικές βελτιστοποίησης, όπως η τοπική αναζήτηση, για να βελτιώσει την ποιότητα των λύσεων και την αποδοτικότητα της διαδικασίας αναζήτησης. Επιπλέον ενσωματώνει ειδικές στρατηγικές διασταύρωσης και μετάλλαξης για να εξερευνήσει αποτελεσματικά τον χώρο των λύσεων και με μηχανισμούς ελέγχου της πολυπλοκότητας μειώνει τον χρόνο επεξεργασίας [44] .

Λαμβάνοντας υπόψη την έρευνα των Jian Gao, Rong Chen (2011) [44] σχετικά με την χρήση γενετικών αλγορίθμων σε συνδυασμό με τοπικές αναζητήσεις, για την εύρεση ενός βέλτιστου χρόνου για το κλασσικό πρόβλημα DPFSP, προσαρμόζονται οι προτεινόμενες λύσεις στην επίλυση του προβλήματος DPFSP με due dates. Για την μέθοδο της τοπικής αναζήτησης χρησιμοποιείται μια προσέγγιση που ονομάζεται Variable Neighborhood Descent (VND), ενώ για τον υβριδικό αλγόριθμο χρησιμοποιείται ο όρος Genetic Algorithm – Local Search (GA_LS). Θα πρέπει να αναφερθεί ότι η παρούσα προσπάθεια επίλυσης του προβλήματος DPFSP με Γενετικούς αλγόριθμους και τοπική αναζήτηση, είναι από τις πρώτες προσεγγίσεις του συγκεκριμένου προβλήματος με την συγκεκριμένη μεθοδολογία.

6.4.2.1 Variable Neighborhood Descent (VND)

Η μέθοδος VND είναι μια ευρετική τεχνική βελτιστοποίησης που χρησιμοποιείται για την επίλυση προβλημάτων συνδυαστικής βελτιστοποίησης. Βασίζεται στην ιδέα της αλλαγής γειτονιών για την αποφυγή τοπικών βέλτιστων λύσεων και την εύρεση καλύτερων σφαιρικών λύσεων.

Ως γειτονιά μιας λύσης εννοούμε το σύνολο λύσεων που προκύπτουν από μικρές μεταβολές των μεταβλητών της λύσης. Σε κάθε βήμα η VND αναζητά βελτιώσεις της λύσης σε μια γειτονιά. Εάν βρεθεί μια καλύτερη λύση

συνεχίζεται η αναζήτηση καλύτερης λύσης στην ίδια γειτονιά. Σε διαφορετική περίπτωση μεταβαίνει σε μια πιο «ευρεία» ή διαφορετική γειτονιά. Αυτή η μετάβαση βοηθάει την αναζήτηση να ξεφύγει από τοπικά βέλτιστα σημεία. Η παραπάνω διαδικασία επαναλαμβάνεται μέχρι να μη βρεθεί καμιά περαιτέρω βελτίωση, οπότε και ολοκληρώνεται η διαδικασία. Η μέθοδος VND είναι αποδοτική σε συνδυαστικά προβλήματα βελτιστοποίησης και πολύ απλή στην εφαρμογή της [1]. Ακολουθεί μια σταδιακή προσέγγιση της λύσης, όπου αναζητείται η βελτίωση μέσω διαφορετικών επιπέδων (γειτονιών). Ας δούμε ένα παράδειγμα χρονοπρογραμματισμού με την μέθοδο VND:

Έστω ότι έχουμε τις παρακάτω εργασίες [A,B,C,D,E] και τις ακολουθίες

(i). [B,D,C,E,A] και (ii). [E,D,C,A,B].

Οι δύο ακολουθίες αποτελούν και δύο επίπεδα ή γειτονιές. Προσπαθούμε να βρούμε καλύτερες λύσεις σε μια γειτονιά ανταλλάσσοντας δύο εργασίες μεταξύ τους, προσπαθώντας να βελτιώσουμε την λύση. Από την στιγμή που η λύση δεν βελτιώνεται, προχωράμε στην επόμενη γειτονιά στην οποία εφαρμόζουμε την ίδια μέθοδο. Από τη στιγμή που οι λύσεις δεν βελτιώνονται η διαδικασία ολοκληρώνεται.

Η εφαρμογή της μεθόδου VND στην επίλυση του προβλήματος DPFSP ξεκινάει από την λύση που επιστρέφει ο αλγόριθμος NEHedd με τις διαμοιρασμένες λύσεις σε κάθε εργοστάσιο. Στο επόμενο στάδιο μεταφέρει εργασίες από το εργοστάσιο με την μεγαλύτερη καθυστέρηση στα υπόλοιπα εργοστάσια.

6.4.2.2 GA_LS

Ο όρος GA_LS αναφέρεται σε έναν υβριδικό γενετικό αλγόριθμο (GA) που συνδυάζεται με τεχνικές τοπικής αναζήτησης (LS). Ο καθαρός γενετικός αλγόριθμος απαιτεί πολλές γενιές για να βελτιώσει σημαντικά τις λύσεις, ενώ η τοπική αναζήτηση βοηθά στην επιτάχυνση αυτής της διαδικασίας. Γι’ αυτό και η λύση που προτείνεται επεκτείνεται μόνο σε μερικές γενιές από 20 έως 50. Όσο αυξάνεται ο αριθμός των γενεών τόσο αυξάνεται και η πιθανότητα για εύρεση καλύτερης λύσης, αλλά ταυτόχρονα αυξάνεται και ο χρόνος εκτέλεσης και τερματισμού του αλγόριθμου.

Ο γενετικός αλγόριθμος μπορεί να παγιδευτεί σε τοπικά βέλτιστα, ενώ η τοπική αναζήτηση επιτρέπει καλύτερη εξερεύνηση του χώρου λύσεων για ταχύτερη και ακριβέστερη σύγκλιση σε βέλτιστες λύσεις.

6.4.4.3 Αναπαράσταση Λύσης Και Αρχικοποιήσεις

Ως μηχανισμός επιλογής των ατόμων που θα χρησιμοποιηθούν για τις λειτουργίες μετάλλαξης και διασταύρωσης, επιλέγεται η κλασσική μέθοδος των γενετικών αλγόριθμων, όπου τα άτομα κατατάσσονται σύμφωνα με την καταλληλότητα τους και στην συνέχεια επιλέγονται.

Άτομα του γενετικού αλγόριθμου θεωρούνται οι συνολικές ακολουθίες για όλα τα εργοστάσια μιας συγκεκριμένης λύσης. Για την καλύτερη κατανόηση αναφερόμαστε σε ένα τυχαίο παράδειγμα με 7 εργασίες (A,B,C,D,E,F,G), δύο μηχανές {1,2} και 2 εργοστάσια [α, β]. Για το συγκεκριμένο παράδειγμα δύο διαφορετικά άτομα θα μπορούσαν να είναι οι δύο παρακάτω λύσεις:

(i). [α] (C, D, E, F), [β] (A, B, G) και

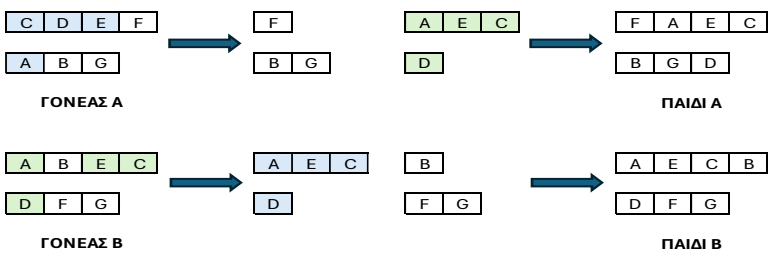
(ii). [α] (A, B, E, C), [β] (D, F, G)

Η καταλληλότητα των ατόμων ορίζεται από την συνολική καθυστέρηση που προκύπτει από το σύνολο των ακολουθιών όλων των εργοστασίων μιας λύσης, σύμφωνα με τις ημερομηνίες/χρόνους τερματισμού των εργασιών.

Αν και έχουν παρουσιαστεί διάφορες μέθοδοι διασταύρωσης στην βιβλιογραφία, οι περισσότερες αφορούν προβλήματα PFSP δηλ. προβλήματα με ένα εργοστάσιο. Ο τελεστής διασταύρωσης που χρησιμοποιείται στην παρούσα εργασία είναι αρκετά παρόμοιος με τον τελεστή διασταύρωσης ενός σημείου (OP) [43], ο οποίος είναι αρκετά εύκολος να υλοποιηθεί και να επεκταθεί στην αναπαράσταση DPFSP.

Για την λειτουργία της διασταύρωσης επιλέγεται το καλύτερο άτομο της γενιάς Α και ένα τυχαίο άτομο της γενιάς Β. Στην συνέχεια αφαιρούνται τυχαία εργασίες από τον γονέα Α και τοποθετούνται στο παιδί Β. Οι υπόλοιπες εργασίες του γονέα Α κληρονομούνται απευθείας στο παιδί Α, στο οποίο προστίθενται στα αντίστοιχα εργοστάσια οι εργασίες του γονέα Β που υπολείπονται αφού έχουν αφαιρεθεί οι εργασίες που έχουν κληρονομηθεί από τον γονέα Α.

Για να γίνει καλύτερα κατανοητό παρουσιάζεται το παρακάτω παράδειγμα:



Εικόνα 14 Hybrid GA - Crossover

Με την εφαρμογή της συγκεκριμένης μεθόδου διασταύρωσης υπάρχει ο κίνδυνος να μην μοιράζονται ισορροπημένα οι εργασίες στα εργοστάσια ενός παιδιού. Για την αντιμετώπιση του συγκεκριμένου προβλήματος αποφασίστηκε να χρησιμοποιηθεί μια διαφορετική μέθοδος διασταύρωσης η οποία σε συνδυασμό με τις μεθόδους τοπικής αναζήτησης που θα παρουσιαστούν στην συνέχεια, παρακάμπτουν αυτή την δυσκολία.

Σχετικά με την διασταύρωση επιλέγονται δύο γονείς, ένα άτομο με πολύ καλή λύση και ένα τυχαίο άτομο. Ο τελεστής διασταύρωσης επιλέγει τυχαία σημεία για όλα τα εργοστάσια του ενός γονέα, τα οποία χρησιμοποιούνται για την διαίρεση των ακολουθιών εργασιών του συγκεκριμένου γονέα σε δεξιές και αριστερές πλευρές. Οι θέσεις εργασίας στις δεξιές πλευρές συμβολίζονται με R και κληρονομούνται απευθείας στο παιδί, ενώ οι υπόλοιπες εργασίες τοποθετούνται σε μια λίστα L. Ακολουθεί για κάθε εργασία στην λίστα L η τοποθέτηση της στην κατάλληλη θέση που μας δίνει καλύτερο χρόνο καθυστέρησης.

Οι διαδικασίες τοπικής αναζήτησης χρησιμοποιούνται ευρέως στους εξελικτικούς αλγόριθμους, είναι ιδιαίτερα ικανές στην επιτάχυνση της σύγκλισης και την εύρεση καλύτερων λύσεων. Στόχος είναι να αφαιρεθούν εργασίες από εργοστάσια με πολύ μεγάλη καθυστέρηση και να τοποθετηθούν σε θέσεις που μειώνουν την καθυστέρηση. Σύμφωνα με την μέθοδο VND που παρουσιάστηκε στην ενότητα 6.4.2.1 προτείνονται τρεις μέθοδοι μετακίνησης θέσεων εργασίας:

- **Εισαγωγή εργασίας (insertion_jobs)**: Επιλέγονται δύο εργασίες για κάθε εργοστάσιο. Έστω ότι εξετάζεται στο εργοστάσιο f η εργασία j_1, τότε επιλέγεται μια δεύτερη εργασία του εργοστασίου η j_2 και αφαιρούνται οι δύο εργασίες από το εργοστάσιο. Στη συνέχεια τοποθετείται η εργασία j_1 σε όλες τις πιθανές θέσεις με τελική την θέση που μας δίνει τον καλύτερο χρόνο καθυστέρησης, το ίδιο εφαρμόζεται και για την εργασία j_2

```
Procedure insertion_jobs
Εξετάζουμε το factory f;
foreach job j1 in factory f
       select j2 randomly (j1≠j2)
       remove j1, j2 from f
       find the best position of j1 in f
       find the best position of j2 in f
end
```

Κώδικας 10 LS Insertion Jobs

- **Μετακίνηση εργασίας (move_jobs)**: Επιλέγεται το εργοστάσιο με την μεγαλύτερη καθυστέρηση fmax και το εργοστάσιο με την μικρότερη καθυστέρηση fmin. Για κάθε εργασία στο fmax, θα δοκιμαστούν όλες οι πιθανές θέσεις στο fmin. Εάν μια μετακίνηση βελτιώνει την συνολική καθυστέρηση δηλ. εάν η καθυστέρηση του fmin είναι μικρότερη από την καθυστέρηση του fmax (ddmax) τότε έχουμε καλύτερους χρόνους, και τοποθετείται η εργασία στην συγκεκριμένη θέση και διακόπτεται η διαδικασία. Σε διαφορετική περίπτωση συνεχίζεται η διαδικασία μέχρι να εξεταστούν όλες οι εργασίες του fmax.

```
Procedure move_jobs
flag <- true
while flag do
       flag <- false
       let fmax be the factory with maximum tardiness and ddmax=tardiness of fmax
       let fmin be the factory with minimum tardiness
       foreach job j in fmax
              find the best position for j in fmin
              if new tardiness of fmin is smaller than ddmax
                     remove j from fmax and assign it to the best position fmin
                     flag <- true
                     break
       end
end
```

Κώδικας 11 LS Move Jobs

- **Ανταλλαγή εργασίας (exchange_jobs)**: Παρόμοια με την μέθοδο μετακίνησης εργασίας επιλέγεται το εργοστάσιο με την μεγαλύτερη καθυστέρηση fmax και το εργοστάσιο με την μικρότερη καθυστέρηση fmin. Μια εργασία j από το εργοστάσιο fmax ανταλλάσσεται με όλες τις εργασίες στο fmin. Εάν μια ανταλλαγή βελτιώσει την συνολική καθυστέρηση πραγματοποιείται. Ενημερώνονται οι fmin και fmax και συνεχίζεται η διαδικασία έως ότου δεν θα μπορεί να επιτευχθεί περαιτέρω βελτίωση.

```
Procedure exchange_jobs
flag <- true
while flag do
       flag <- false
       let fmax be the factory with maximum tardiness and ddmax=tardiness of fmax
       let fmin be the factory with minimum tardiness
       foreach job j in fmax
              try to exchange j with each job in fmin
              find the best exchange and denote the job in fmin by j1
              if new tardiness of fmin and fmax smaller than ddmax
                     exchange j and j1
                     flag <- true
                     break
       end
end
```

Κώδικας 12 LS Exchange Jobs

Εφαρμόζοντας την διαδικασία της τοπικής αναζήτησης, χρησιμοποιώντας τις τρεις διαδικασίες που περιγράψαμε παραπάνω insertion_jobs, move_jobs και exchange_jobs, καταλήγουμε στην παρακάτω σύνοψη:
Σε κάθε εργοστάσιο εφαρμόζεται πρώτα insertion_jobs, για να ακολουθήσουν οι διαδικασίες move_jobs και exchange_jobs. Οι δύο τελευταίες διαδικασίες μπορεί να αλλάξουν τις ακολουθίες των εργοστασίων οπότε επαναλαμβάνεται η διαδικασία από την αρχή, εφαρμόζοντας insertion_jobs. Η διαδικασία της τοπικής αναζήτησης θα σταματήσει σε ένα τοπικό ελάχιστο όταν οι move_jobs και exchange_jobs δεν θα μπορούν να κάνουν καμία αλλαγή.

```
Procedure local search
flag <- true
foreach factory f
       perform insertion_jobs on f
end
while flag do
       perform move_jobs
       perform exchange_jobs
       if there any factory f changed
              perform insertion_jobs on f
       else
              flag <- false
       end
end
```

Κώδικας 13 Local Search Hybrid

Τέλος για την αναπαράσταση του υβριδικού αλγόριθμου καταλήγουμε στις λειτουργείες που παρουσιάζονται στον παρακάτω ψευδοκώδικα.

```
Procedure GA_LS
While stopping criterion is not satisfied do
       evaluate fitness
       apply local search on the best individual and a randomly selected individual
       update the best solution
       apply selection operator
       assign the best solution to a randomly selected individual
       apply crossover operator
end
```

Κώδικας 14 Hybrid GA with Local Search

Ο αλγόριθμος εκτελεί τις παρακάτω λειτουργίες με την σειρά που παρουσιάζονται:

1. Δημιουργεί τον αρχικό πληθυσμό της πρώτης γενιάς στον οποίο ενσωματώνει σε τυχαία θέση την ακολουθία (άτομο) που επιστρέφει ο αλγόριθμος NEHedd. Με αυτό τον τρόπο παρέχεται αρχικά μια καλή λύση η οποία δεν δημιουργείται τυχαία.
2. Ταξινομεί τον πληθυσμό με βάση την συνάρτηση fitness σε αύξουσα σειρά. Στην περίπτωσή μας η συνάρτηση fitness είναι ο χρόνος καθυστέρησης της κάθε ακολουθίας (ατόμου).
3. Εκτελείται τοπική αναζήτηση όπως περιγράφεται παραπάνω στο καλύτερο άτομο και σε ένα τυχαίο άτομο. Τα αποτελέσματα της τοπικής αναζήτησης αποθηκεύονται στα άτομα. Ξεκινώντας ο αλγόριθμος της τοπικής αναζήτησης εκτελεί την μέθοδο **insertion_jobs** για όλα τα εργοστάσια. Ακολουθούν οι τεχνικές **move_jobs** και **exchange_jobs.** Αν μετά την εκτέλεση των δύο τελευταίων τοπικών αναζητήσεων (move_jobs και exchange_jobs) το άτομο παραμείνει το ίδιο τερματίζεται η διαδικασία της τοπικής αναζήτησης. Σε διαφορετική περίπτωση η τοπική αναζήτηση εκτελείται από την αρχή.
4. Ταξινομεί τον πληθυσμό με βάση την συνάρτηση fitness σε αύξουσα σειρά, διότι η καλύτερη λύση μπορεί να έχει αλλάξει.
5. Επιλέγεται το καλύτερο άτομο και ένα τυχαίο άτομο και εκτελείται διασταύρωση όπως περιγράφεται παραπάνω. Τα παιδιά που προκύπτουν από την διαδικασία της αναπαραγωγής προστίθενται στις τελευταίες θέσει του πληθυσμού. Με αυτόν τον τρόπο επιτυγχάνεται οι νέες γενιές να αποτελούνται από καλύτερες λύσεις. Θα μπορούσε να προστεθεί ένας τελεστής για να ορίζει σε πόσα άτομα θα εφαρμοστεί διασταύρωση.
6. Επαναλαμβάνονται οι εργασίες για όσες γενιές έχουν επιλεγεί. Ένα δεύτερο κριτήριο τερματισμού θα μπορούσε να είναι η επιστροφή της ίδιας καλύτερης λύσης για έναν συγκεκριμένο αριθμό επαναλήψεων.

Ένας γενετικός αλγόριθμος, όπως αυτός που παρουσιάζεται στην παρούσα εργασία μπορεί να αποτελέσει εργαλείο το οποίο θα χρησιμοποιηθεί σε βιομηχανικά περιβάλλοντα για την βελτιστοποίηση της παραγωγής, με στόχο την ελαχιστοποίηση των καθυστερήσεων και την βελτίωση της αποδοτικότητας.

6.4.4.3 Βιβλιοθήκη PYGAD

Αν και για την ανάπτυξη του συγκεκριμένου γενετικού αλγόριθμου δεν έχουν χρησιμοποιηθεί έτοιμες βιβλιοθήκες, κρίνεται απαραίτητο να γίνει μια αναφορά στην βιβλιοθήκη PYGAD [45] της python.

Η PYGAD είναι μια βιβλιοθήκη python που επιτρέπει την εύκολη υλοποίηση γενετικών αλγορίθμων. Παρέχει έτοιμα εργαλεία και παραμετροποιήσεις για την δημιουργία, την εκτέλεση και την βελτιστοποίηση λύσεων για προβλήματα μέσω γενετικών αλγορίθμων.

Τα βασικά συστατικά του PYGAD είναι:

- **Πληθυσμός (Population)**: Ο πληθυσμός αποτελείται από λύσεις (άτομα), όπου κάθε λύση αντιπροσωπεύεται ως διάνυσμα τιμών (γονίδια).
    - **sol_per_pop**: Το μέγεθος του πληθυσμού.
    - **num_genes**: Αριθμός των γονιδίων σε κάθε άτομο.
- **Χωρικός Περιορισμός Γονίδιων (Gene Space)** : Ορίζει το εύρος τιμών για κάθε γονίδιο.
    - **gene_space**: Εύρος τιμών γονιδίου.
- **Συνάρτηση Καταλληλότητας (fitness function)** : Καθορίζει την ποιότητα κάθε λύσης
    - **fitness_func**
- **Γενετικοί τελεστές:**
    - **Επιλογή:** Επιλέγει τα καλύτερα άτομα για αναπαραγωγή
        - **Roulette Wheel:** Τυχαία με βάση την καταλληλότητα
        - **Rank Selection:** Βάση ταξινόμησης
        - **Tournament Selection:** Αγώνας μεταξύ υποομάδων
    - **Διασταύρωση (Crossover):** Δημιουργεί νέα παιδιά από δύο γονείς
        - **Single-Point Crossover**
        - **Two-Point Crossover**
        - **Uniform Crossover**

- **Μετάλλαξη (Mutation):** Τροποποιεί τυχαία γονίδια για να εισάγει ποικιλία στον πληθυσμό
  - **Random Mutation:** Τυχαία αλλαγή τιμών
  - **Swap Mutation:** Ανταλλαγή τιμών μεταξύ γονιδίων

- **Παράμετροι Πληθυσμού και Γενεών:**
  - **num_generations:** Ο αριθμός των γενεών
  - **num_parents_mating:** Ο αριθμός των γονιών που θα συμμετάσχουν στην αναπαραγωγή
  - **keep_parents:** Πόσα γονικά άτομα θα διατηρηθούν για την επόμενη γενιά.
- **Αποτελέσματα και Ανάλυση:** Εργαλεία για την εξαγωγή και ανάλυση των αποτελεσμάτων
  - **best_solution ():** Επιστρέφει την καλύτερη λύσης και την τιμή fitness
  - **plot_fitness ():** Δημιουργεί γράφημα με την πορεία της καταλληλότητας στις γενεές
- **Εξατομίκευση:** Επιτρέπει την δημιουργία προσαρμοσμένων τελεστών από τον χρήστη
  - **Custom_Crossover:** Δημιουργία μεθόδου διασταύρωσης από τον χρήστη
  - **Custom_Mutation:** Δημιουργία μεθόδου μετάλλαξης από τον χρήστη
  - **Callbacks:** Παρακολούθηση της διαδικασίας σε κάθε γενιά
- Κριτήριο Τερματισμού: Μπορούμε να ορίσουμε κριτήρια για την τερματισμό του αλγόριθμου:
  - Αριθμός γενεών (num_generations)
  - Βελτίωση καταλληλότητας (stop_criteria)

Η βιβλιοθήκη PYGAD είναι μια ευέλικτη βιβλιοθήκη που παρέχει πλήρη έλεγχο στις λεπτομέρειες των γενετικών αλγορίθμων, ενώ παράλληλα προσφέρει απλότητα για βασικές υλοποιήσεις.

Στην παρούσα εργασία δεν έγινε χρήση της συγκεκριμένης βιβλιοθήκης καθώς ο γενετικός αλγόριθμος αποτελείται από απλές εφαρμογές των τελεστών αναπαραγωγής και μιας και συνδυάζεται με τεχνικές τοπικής αναζήτησης δεν θα ήταν πολύτιμη η βοήθεια της βιβλιοθήκης PYGAD. Παρόλα αυτά κρίνεται επιτακτική η χρήση της βιβλιοθήκης PYGAD σε περίπτωση επέκτασης του συγκεκριμένου αλγόριθμου.

6.4.4 Λογισμικό επίλυσης προβλημάτων συνδυαστικής βελτιστοποίησης Google OR TOOLS

Το OR-Tools είναι λογισμικό ανοικτού κώδικα για *συνδυαστική βελτιστοποίηση (combinatorial optimization)*, η οποία επιδιώκει να βρει την καλύτερη λύση σε ένα πρόβλημα από ένα πολύ μεγάλο σύνολο πιθανών λύσεων. Ορισμένα παραδείγματα προβλημάτων τα οποία μπορούν να επιλυθούν με το εργαλείο OR-Tools είναι τα παρακάτω:

- **Δρομολόγηση οχημάτων**: Εύρεση βέλτιστων διαδρομών για στόλους οχημάτων που παραλαμβάνουν και παραδίδουν πακέτα δεδομένων περιορισμών (π.χ. «αυτό το φορτηγό έχει μέγιστη χωρητικότητα 20.000 κιλά» ή «όλες οι παραδόσεις πρέπει να γίνουν μέσα σε ένα παράθυρο δύο ωρών»).
- **Χρονοπρογραμματισμός**: Εύρεση του βέλτιστου χρονοδιαγράμματος για ένα σύνθετο σύνολο εργασιών, ορισμένες από τις οποίες πρέπει να εκτελεστούν πριν από άλλες, σε ένα σταθερό σύνολο μηχανημάτων ή άλλων πόρων.
- **Συσκευασία αντικειμένων**: Συσκευασία όσο το δυνατόν περισσότερων αντικειμένων διαφόρων μεγεθών σε σταθερό αριθμό κάδων με μέγιστη χωρητικότητα.

Στις περισσότερες περιπτώσεις, τέτοια προβλήματα έχουν έναν τεράστιο αριθμό πιθανών λύσεων, το οποίο συνεπάγεται τεράστιο υπολογιστικό κόστος. Για να το ξεπεράσει αυτό, το εργαλείο OR-Tools χρησιμοποιεί αλγορίθμους τελευταίας τεχνολογίας για να περιορίσει το σύνολο αναζήτησης, προκειμένου να βρει μια βέλτιστη (ή σχεδόν βέλτιστη) λύση.

Το OR-Tools περιλαμβάνει επιλυτές για:

Βελτιστοποίηση με περιορισμούς, ή προγραμματισμός με περιορισμούς (CP), είναι η ονομασία που δίνεται στον εντοπισμό εφικτών λύσεων από ένα πολύ μεγάλο σύνολο υποψήφιων, όπου το πρόβλημα μπορεί να μοντελοποιηθεί με περιορισμούς διαφόρων τύπων (γραμμικούς και μη γραμμικούς). Προβλήματα CP προκύπτουν σε πολλούς επιστημονικούς κλάδους και κλάδους της μηχανικής. Θα πρέπει να σημειωθεί ότι εδώ η λέξη προγραμματισμός αναφέρεται στη διευθέτηση ενός σχεδίου και όχι στον προγραμματισμό σε μια γλώσσα προγραμματισμού. Στα προβλήματα CP οι χρήστες δηλώνουν τους περιορισμούς στις εφικτές λύσεις για ένα σύνολο μεταβλητών απόφασης, καθορίζοντας τις ιδιότητες μιας λύσης που πρέπει να βρεθεί καθώς και μια μέθοδο για την ικανοποίηση αυτών των περιορισμών.

Γραμμική βελτιστοποίηση ή *γραμμικός προγραμματισμός* είναι το όνομα που δίνεται στον υπολογισμό της καλύτερης λύσης σε ένα πρόβλημα που μοντελοποιείται ως σύνολο γραμμικών σχέσεων.

6.4.3.2 OR Tools για την επίλυση του προβλήματος Flow Shop (FSP) [46]

Για την επίλυση του προβλήματος χρονοπρογραμματισμού flow shop Problem με την βοήθεια του λογισμικού OR-Tools ακολουθούμε τις παρακάτω ενέργειες (Για την αναπαράσταση του παραδείγματος χρησιμοποιούμε την γλώσσα προγραμματισμού python) [47]:

- Εισαγωγή κατάλληλων βιβλιοθηκών:

```
import collections
from ortools.sat.python import pomade
```

- Εισαγωγή δεδομένων προς επεξεργασία: Εδώ υπάρχουν διάφορες δυνατότητες εισαγωγής των δεδομένων εισάγοντας τα δεδομένα από ένα αρχείο ή εισάγοντας δεδομένα απευθείας σε κάποιον πίνακα.
- Ορισμός του μοντέλου:

```
mdl = CpoModel()
```

- Ορισμός των μεταβλητών:

```
# Δημιουργία μεταβλητής εργασίας
operations = [[interval_var(size=OP_DURATIONS[j][m], name='J{}-M{}'.format(j, m)) for m
in range(NB_MACHINES)] for j in range(NB_JOBS)]

# Δημιουργία ακολουθίας για κάθε μηχανή
op_sequences = [sequence_var([operations[i][j] for i in range(NB_JOBS)],
name='M{}'.format(j)) for j in range(NB_MACHINES)]
```

- Ορισμός των περιορισμών:

```
# Κάθε λειτουργία θα ξεκινάει μετά το πέρας της προηγούμενης
for j in range(NB_JOBS):
    for m in range(1, NB_MACHINES):
        mdl.add(end_before_start(operations[j][m-1], operations[j][m]))

# Μη επικάλυψη για λειτουργίες που εκτελούνται στο ίδιο μηχάνημα
for m in range(NB_MACHINES):
    mdl.add(no_overlap(op_sequences[m]))

# Οι ακολουθίες θα πρέπει να είναι πανομοιότυπες σε όλες τις μηχανές
for m in range(1, NB_MACHINES):
    mdl.add(same_sequence(op_sequences[0], op_sequences[m]))
```

- Καθορισμός του στόχου:

  Για ένα πρόβλημα Flow Shop ο στόχος είναι να μειώσουμε τον χρόνο makespan. *Εδώ θα πρέπει να προσαρμόσουμε το πρόβλημα στην λύση με due dates ορίζοντας ως στόχο την μείωση του χρόνου καθυστέρησης.*

```
# Ελαχιστοποίηση του χρόνου τερματισμού (makespan)
mdl.add(minimize(max([end_of(operations[i][NB_MACHINES-1]) for i in range(NB_JOBS)])))
```

- Κλήση του επιλυτή

```
print('Solving model...')
res = mdl.solve(FailLimit=10000, TimeLimit=10)
```

- Εκτύπωση αποτελεσμάτων:

```
import docplex.cp.utils_visu as visu
if res and visu.is_visu_enabled():
    visu.timeline('Solution for permutation flow-shop ' + filename)
    visu.panel('Jobs')
    for i in range(NB_JOBS):
        visu.sequence(name='J' + str(i),
                      intervals=[(res.get_var_solution(operations[i][j]),  j,  'M'  +
str(j)) for j in range(NB_MACHINES)])
    visu.panel('Machines')
    for j in range(NB_MACHINES):
        visu.sequence(name='M' + str(j),
                      intervals=[(res.get_var_solution(operations[i][j]),  j,  'J'  +
str(i)) for i in range(NB_JOBS)])
    visu.show()
```

Το **OR-Tools** της Google είναι ένα ισχυρό εργαλείο βελτιστοποίησης που διευκολύνει την επίλυση προβλημάτων χρονοπρογραμματισμού, όπως το DPFSP. Υποστηρίζει πολλαπλές μεθόδους, όπως γραμμικό προγραμματισμό και προγραμματισμό με περιορισμούς, και επιτρέπει τον ορισμό περιορισμών, όπως προθεσμίες, αλληλεξαρτήσεις εργασιών, και κατανομή πόρων.

Με τη χρήση ευρετικών και μεταευρετικών αλγορίθμων, το OR-Tools είναι ιδιαίτερα αποδοτικό για την αντιμετώπιση σύνθετων προβλημάτων. Παράλληλα, επιτρέπει την εύκολη διατύπωση μοντέλων για πραγματικά σενάρια, όπως χρονοδιαγράμματα εργοστασίων ή δρομολόγηση οχημάτων. Αποτελεί ιδανική λύση για τη βελτιστοποίηση συστημάτων που απαιτούν διαχείριση χρόνου και πόρων με υψηλή ακρίβεια και αποδοτικότητα.

6.6 Σχεδιασμός Πειραμάτων

6.6.1 Δεδομένα

Για την εξέταση της απόδοσης των λύσεων χρησιμοποιήθηκαν δύο διαφορετικά σύνολα δεδομένων:

Τα δεδομένα των Naderi και Ruiz [1] τα οποία εμπλουτίστηκαν από τους Ankit Khare & Sunil Agrawal [23] με χρόνους τερματισμού. Το συγκεκριμένο σύνολο αποτελείται από 420 μικρού μεγέθους προβλήματα τα οποία εξετάστηκαν στις περισσότερες περιπτώσεις και κυρίως τα μεγαλύτερα από αυτά τα προβλήματα.

Τα δεύτερο σύνολο δεδομένων αφορά τα δεδομένα Taillard [48] τα οποία αποτελούνται από 720 μεγάλα και δύσκολα προβλήματα. Από το συγκεκριμένο σύνολο αντλήθηκαν προβλήματα δειγματοληπτικά, τα αποτελέσματα των οποίων παρουσιάζονται σε δεύτερο παράρτημα.

Η σύγκριση των αποτελεσμάτων των λύσεων μας γίνεται με τις γνωστές καλύτερες λύσεις όπως αυτές διατίθενται από τους Ankit Khare & Sunil Agrawal στον παρακάτω σύνδεσμο:

https://data.mendeley.com/datasets/4tkhdx4jjx/2.

Καθώς η εκτέλεση των μεγαλύτερων προβλημάτων απαιτεί υπολογιστικούς πόρους και πολύ μεγάλους χρόνους εκτέλεσης, χρησιμοποιήθηκε ένα σύνολο δεδομένων με μικρότερα προβλήματα το οποίο αποτελείται από 2 έως 4 εργοστάσια και από 2 έως 16 εργασίες. Το συγκεκριμένο σύνολο μας επιτρέπει να εκτελέσουμε τους αλγόριθμους επίλυσης των προβλημάτων σε σύντομα χρονικά διαστήματα. Με τον τρόπο αυτό αξιοποιούνται τα μικρότερα σύνολα δεδομένων για να ελεγχθεί αν οι αλγόριθμοι αποδίδουν όπως αναμένεται και οδηγούν στην βέλτιστη λύση.

Ακολουθεί η εκτέλεση των αλγορίθμων με δεδομένα που αντλήθηκαν από το σύνολο δεδομένων του TAILARD και εξετάζονται δειγματοληπτικά προβλήματα με 2, 4, 6 και 7 εργοστάσια, 5, 10 και 20 μηχανές και 20, 50, 100 και 200 εργασίες.

Οι χρόνοι προθεσμιών (due dates) του συνόλου δεδομένων που χρησιμοποιείται κατά την εκτέλεση των αλγορίθμων υπολογίστηκαν με την μέθοδο των Hasija και Rajendran (2004) [22] με τον παρακάτω τρόπο:

- Συνολικός χρόνος επεξεργασίας κάθε εργασίας:.

$$P_j = \sum_{i=1}^{m} p_{i,j}$$, όπου j=1,2, … ,n

- Υπολογισμός προθεσμίας για κάθε εργασία:

$$d_j = P_j x (1 + rand\ x \left(\frac{1}{f}\right) x\ a)$$

Όπου:

- Rand: Τυχαίος αριθμός από ομοιόμορφη κατανομή στο [0, 1]
- 1/f: Παράγοντας που λαμβάνει υπόψη τον αριθμό των εργοστασίων, καθιστώντας τις προθεσμίες πιο αυστηρές όταν υπάρχουν περισσότερα εργοστάσια.1
- α: Παράμετρος για την ισορροπία αυστηρότητας, η οποία καθορίστηκε σε 0,5 έπειτα από πειράματα, ώστε οι προθεσμίες να μην είναι ούτε υπερβολικά αυστηρές αλλά ούτε υπερβολικά χαλαρές.

Με αυτόν τον τρόπο εξασφαλίζεται η προσαρμογή των δεδομένων σε μικρά καθώς και σε μεγάλα προβλήματα.

6.6.2 Εφαρμογή Αλγόριθμων

Στα πλαίσια των δοκιμαστικών πειραμάτων εφαρμόστηκαν οι υλοποιημένοι αλγόριθμοι για τον σχεδιασμό των οποίων έγινε αναφορά στις ενότητες 6.1 – 6.5.

Για την υλοποίηση των πειραμάτων και για την εκτέλεση των αλγορίθμων ακολουθήθηκαν τα παρακάτω βήματα:

1. Αρχικά εισάγονται τα δεδομένα από τα αντίστοιχα αρχεία και κάθε αρχείο εξετάζεται ξεχωριστά.
2. Όλες οι λύσεις δέχονται ως αρχική ακολουθία την λύση που εξάγεται από τον αλγόριθμο NEHedd.
3. Εκτελείται ο αλγόριθμος ILS που αποτελεί μια απλή εφαρμογή τοπικής αναζήτησης. Ο συγκεκριμένος αλγόριθμος εφαρμόζει τυχαίες εναλλαγές μεμονωμένης εργασίας μεταξύ εργοστασίων, χωρίς να λαμβάνει υπόψη τους χρόνους καθυστέρησης. Ο συγκεκριμένος αλγόριθμος εκτελείται και επιστρέφει την δική του λύση.
4. Εκτελείται ο αλγόριθμος RSLS ο οποίος εφαρμόζει τυχαίες εναλλαγές ακολουθιών με περισσότερες από μια εργασίες μεταξύ εργοστασίων αλλά και στο ίδιο το εργοστάσιο όπως περιγράφεται στην ενότητα 6.2.1. Ο συγκεκριμένος αλγόριθμος εκτελείται και επιστρέφει την δική του λύση αλλά συμμετέχει και στην λύση IG.
5. Εκτελείται ο αλγόριθμος RSLS_II ο οποίος λειτουργεί όπως ο αλγόριθμος RSLS αλλά κάνει εναλλαγές με μεμονωμένες εργασίες. Ο συγκεκριμένος αλγόριθμος εκτελείται και επιστρέφει την δική του λύση αλλά συμμετέχει και στην λύση IG.
6. Εκτελείται ο αλγόριθμος ls_insertion_job ο οποίος αποτελεί μέρος της λύσης του γενετικού αλγόριθμου και εφαρμόζει τοπική αναζήτηση αφαιρώντας 2 εργασίες από ένα εργοστάσιο και επανατοποθετώντας τες στην καλύτερη δυνατή θέση, με στόχο την μείωση της καθυστέρησης. Ο συγκεκριμένος αλγόριθμος εκτελείται και επιστρέφει την δική του λύση αλλά συμμετέχει και στην λύση GA_LS.

7. Εκτελείται ο αλγόριθμος ls_move_job ο οποίος αποτελεί μέρος της λύσης του γενετικού αλγόριθμου και εφαρμόζει τοπική αναζήτηση αφαιρώντας μια εργασία από το εργοστάσιο με την μεγαλύτερη καθυστέρηση maxTT και τοποθετώντας την στο εργοστάσιο με την μικρότερη καθυστέρηση εάν μετά την τοποθέτηση το εργοστάσιο με την μικρότερη καθυστέρηση δεν έχει καθυστέρηση μεγαλύτερη από maxTT. Ο συγκεκριμένος αλγόριθμος εκτελείται και επιστρέφει την δική του λύση αλλά συμμετέχει και στην λύση GA_LS.
8. Εκτελείται ο αλγόριθμος ls_exchange_job ο οποίος αποτελεί μέρος της λύσης του γενετικού αλγόριθμου και εφαρμόζει τοπική αναζήτηση εναλλάσσοντας δύο εργασίες μεταξύ των εργοστασίων με την μεγαλύτερη και την μικρότερη καθυστέρηση εάν επιτευχθεί μείωση της συνολικής καθυστέρησης. Ο συγκεκριμένος αλγόριθμος εκτελείται και επιστρέφει την δική του λύση αλλά συμμετέχει και στην λύση GA_LS.
9. Ο αλγόριθμος που συνδυάζει τους δύο αλγόριθμους RSLS και RSLS II είναι ένας άπληστος αλγόριθμος ο οποίος εκτελεί αρχικώς μια τοπική αναζήτηση στην λύση που έχει εξάγει ο αλγόριθμος NEH. Στην συνέχεια εκτελείται μια μέθοδος τοπικής αναζήτησης από την οποία προκύπτουν βελτιωμένες συνήθως λύσεις. Στην συνέχεια και σύμφωνα μα τον τελεστή d ο οποίος παίρνει τιμές μεταξύ του 0 και του 1, ορίζεται ο αριθμός εργασιών που θα αφαιρεθούν από ένα τυχαίο εργοστάσιο για να τοποθετηθούν ξανά στο σύστημα στην θέση που δίνει την μικρότερη καθυστέρηση. Εάν η λύση που προκύπτει είναι καλύτερη από την προηγούμενη την κρατάμε. Σε διαφορετική περίπτωση χρησιμοποιούμε simulated annealing σύμφωνα με το τελεστή T ο οποίος παίρνει αρχική τιμή μεταξύ 0 και 1. Όσο μεγαλύτερος ορίζεται ο τελεστής T, τόσο πιο πολλές χειρότερες λύσεις γίνονται αποδεκτές.
10. Τέλος εκτελείται ο γενετικός αλγόριθμος ο οποίος ξεκινάει με τυχαίο πληθυσμό και ένα ή περισσότερα άτομα που επιστρέφει ο ευρετικός αλγόριθμος NEHedd. Ακολουθεί η εκτέλεση τεχνικών τοπικής αναζήτησης που βελτιώνουν το πρόβλημα, για να ακολουθήσει μια μέθοδος διασταύρωσης η οποία δημιουργεί 2 νέα άτομα. Μπορούν να αυξηθούν τα άτομα που θα αναπαραχθούν με διασταύρωση για να επιτευχθούν καλύτερες λύσεις. Τέλος επαναλαμβάνεται η συγκεκριμένη διαδικασία για όσες γενιές έχουν οριστεί.

7 Αξιολόγηση και Ανάλυση Αποτελεσμάτων

7.1 Παρουσίαση Αποτελεσμάτων

Η παρούσα ενότητα επικεντρώνεται στην αξιολόγηση των αποτελεσμάτων που προέκυψαν από την επίλυση του προβλήματος DPFSP με due dates. Το πρόβλημα αφορά τη βελτιστοποίηση της εκτέλεσης εργασιών σε πολλαπλά εργοστάσια, λαμβάνοντας υπόψη τις χρονικές προθεσμίες (due dates) για την ολοκλήρωση κάθε εργασίας.

Ο στόχος της ανάλυσης είναι να διερευνηθεί η αποτελεσματικότητα της προτεινόμενης μεθόδου/αλγορίθμου ως προς τον βασικό δείκτη απόδοσης, που είναι η συνολική καθυστέρηση όλων των εργασιών πέρα από τους χρόνους τερματισμού κάθε εργασίας (due dates).

Γίνεται μια σύγκριση της απόδοσης μεταξύ των ευρετικών και των μεταευρετικών μεθόδων καθώς εξετάζεται επίσης η απόδοση κάθε αλγόριθμου ξεχωριστά. Εξετάζονται οι παράμετροι οι οποίοι επηρεάζουν το αποτέλεσμα όπως το μέγεθος του προβλήματος (πλήθος εργασιών, πλήθος εργοστασίων), παράμετροι όπως οι χρόνοι εκτέλεσης καθώς και οι διαδοχικές εκτελέσεις/επαναλήψεις μιας μεθόδου.

Τέλος παρουσιάζονται τα αποτελέσματα σε πίνακες και ακολουθεί μια ανάλυση των αποτελεσμάτων με διαγράμματα και στατιστικές μετρικές.

Καθώς το βασικό κριτήριο αποδοτικότητας στην παρούσα εργασία είναι η συνολική καθυστέρηση σε προβλήματα χρονοπρογραμματισμού (DPFSP), για την οποία δεν υπάρχει μεγάλη βιβλιογραφία ήταν δύσκολο να βρεθεί ένα ευρέως διαδεδομένο σύνολο δεδομένων και οι βέλτιστες λύσεις αυτών. Κάθε πρόβλημα της κατηγορίας

$$\boldsymbol{DF \mid prmu \mid \sum T_j}$$

ορίζεται από τέσσερεις μεταβλητές: τον αριθμό των εργοστασίων, τον αριθμό των εργασιών, τον αριθμό των μηχανών και των χρόνων τερματισμού κάθε εργασίας.

7.2 Ανάλυση Αποτελεσμάτων

Για την παρουσίαση και την ανάλυση των αποτελεσμάτων εφαρμόζεται η μέθοδος της σχετικής ποσοστιαίας απόκλισης (Relative Percentage Deviation – RPD) σε σχέση με τις καλύτερες λύσεις που υπάρχουν στην βιβλιογραφία:

$$RPD = \frac{INV_{sol} - BEST_{sol}}{BEST_{sol}} * 100$$

Όπου INVsol αφορά το αποτέλεσμα μιας συγκεκριμένης λύσης ενός συγκεκριμένου προβλήματος και BESTsol είναι η καλύτερη γνωστή λύση του προβλήματος από το σημείο αναφοράς των Khare & Agrawal.

Ξεκινώντας παρουσιάζονται τα αποτελέσματα για το σύνολο δεδομένων με τα προβλήματα των Naderi και Ruiz. Στον πίνακα που παρουσιάζονται οι λύσεις μαρκάρονται με διαφορετικό χρώμα οι λύσεις που δίνουν το καλύτερο αποτέλεσμα μεταξύ των δύο κύριων λύσεων αλγορίθμων HYLG και GA LS. Επίσης στην τελευταία στήλη υπολογίζεται η σχετική ποσοστιαία απόκλιση της καλύτερης των λύσεων σε σχέση με τις καλύτερες λύσεις της βιβλιογραφίας.

Αρχικά εμφανίζονται προβλήματα που αφορούν 2 εργοστάσια και από 4 έως 6 εργασίες:

2	File	FACTORIES	JOBS	MACHINES	NEHEDD	ILS	RSLS	RSLS_II	LS IN	LS MV	LS EX	HYLG	GA LS	BEST	RPD
1.1	I_2_4_2_1	2	4	2	28	28	28	28	28	27	28	**28**	27	27	0
1.2	I_2_4_2_2	2	4	2	51	44	44	44	51	51	44	44	44	44	0
1.3	I_2_4_2_3	2	4	2	137	137	137	137	137	137	137	**137**	133	133	0
1.4	I_2_4_2_4	2	4	2	58	57	57	57	58	58	57	**58**	57	57	0
1.5	I_2_4_2_5	2	4	2	128	128	128	128	128	128	138	128	128	128	0
2.1	I_2_4_3_1	2	4	3	6	6	6	6	6	6	6	6	6	6	0
2.2	I_2_4_3_2	2	4	3	133	133	133	133	133	133	133	133	133	133	0
2.3	I_2_4_3_3	2	4	3	139	113	139	113	139	139	139	**139**	113	113	0
2.4	I_2_4_3_4	2	4	3	86	86	88	86	86	86	88	86	86	86	0
2.5	I_2_4_3_5	2	4	3	75	30	75	30	75	75	30	**75**	30	30	0
3.1	I_2_4_4_1	2	4	4	118	118	133	118	118	118	127	118	118	118	0
3.2	I_2_4_4_2	2	4	4	65	49	65	49	65	65	49	**65**	49	49	0
3.3	I_2_4_4_3	2	4	4	104	92	92	92	104	104	92	**104**	92	92	0
3.4	I_2_4_4_4	2	4	4	58	58	94	58	58	58	58	58	58	58	0
3.5	I_2_4_4_5	2	4	4	76	76	76	76	76	76	96	76	76	76	0
4.1	I_2_4_5_1	2	4	5	60	60	60	60	60	60	67	60	60	60	0
4.2	I_2_4_5_2	2	4	5	62	62	62	62	62	62	62	62	62	62	0
4.3	I_2_4_5_3	2	4	5	52	52	52	52	52	52	52	52	52	52	0
4.4	I_2_4_5_4	2	4	5	174	104	174	104	174	174	174	**174**	104	104	0
4.5	I_2_4_5_5	2	4	5	22	22	22	22	22	22	22	22	22	22	0
5.1	I_2_6_2_1	2	6	2	168	168	164	168	168	168	178	168	**192**	164	0
5.2	I_2_6_2_2	2	6	2	151	151	151	151	151	151	151	151	151	151	0
5.3	I_2_6_2_3	2	6	2	296	263	240	263	272	296	290	224	224	224	0
5.4	I_2_6_2_4	2	6	2	100	97	144	97	100	100	100	100	100	97	0
5.5	I_2_6_2_5	2	6	2	432	415	419	415	432	432	434	414	**419**	414	0
6.1	I_2_6_3_1	2	6	3	146	146	146	146	146	146	146	146	146	146	0
6.2	I_2_6_3_2	2	6	3	286	286	287	286	286	286	286	286	286	286	0
6.3	I_2_6_3_3	2	6	3	269	251	251	251	269	269	269	251	**268**	251	0
6.4	I_2_6_3_4	2	6	3	185	167	167	167	185	185	167	167	167	167	0
6.5	I_2_6_3_5	2	6	3	119	109	109	109	119	119	109	**119**	109	109	0

Πίνακας 5 Αποτελέσματα 2 εργοστάσια

Όπως παρατηρούμε στο συγκεκριμένο σύνολο δεδομένων οι δύο αλγόριθμοι μας έχουν δώσει λύσεις ακριβώς όπως και οι καλύτερες γνωστές λύσεις για κάθε ένα από τα προβλήματα. Καθώς το συγκεκριμένο σύνολο δεδομένων αποτελείται από μικρά προβλήματα δεν κρίνεται απαραίτητη η περαιτέρω ανάλυση των αποτελεσμάτων διότι οι καλύτερες γνωστές λύσεις δεν μπορούν να βελτιωθούν περαιτέρω.

Στο αποτέλεσμα των προβλημάτων με 3 εργοστάσια και 4 έως 16 εργασίες οι χρόνοι καθυστέρησης που δίνουν οι αλγόριθμοι HYLG και GA LS έχουν πολύ μικρές αποκλίσεις σε σχέση με τις καλύτερες λύσεις.

3	File	FACTORIES	JOBS	MACHINES	NEHEDD	ILS	RSLS	RSLS_II	LS IN	LS MV	LS EX	HYLG	GA LS	BEST	RPD
1.1	I_3_4_2_1	3	4	2	2	2	2	2	2	2	2	2	2	2	0
1.2	I_3_4_3_5	3	4	3	8	8	8	8	8	8	8	8	8	8	0
1.3	I_3_4_4_1	3	4	4	52	47	47	52	52	47	47	47	47	47	0
1.4	I_3_4_5_5	3	4	5	6	6	6	6	6	6	6	6	6	6	0
2.1	I_3_6_2_1	3	6	2	82	82	82	82	82	82	82	82	82	82	0
2.2	I_3_6_3_5	3	6	3	174	153	153	153	174	174	185	150	146	146	0
2.3	I_3_6_4_1	3	6	4	126	116	112	112	126	126	116	116	111	111	0
2.4	I_3_6_5_5	3	6	5	86	62	68	62	86	80	86	64	50	50	0
3.1	I_3_8_2_1	3	8	2	288	273	288	273	288	288	275	288	273	273	0
4.1	I_3_10_2_5	3	10	2	314	278	282	278	314	291	293	308	278	278	0
4.2	I_3_10_3_1	3	10	3	614	583	565	583	614	616	598	566	566	563	0.36
4.3	I_3_10_4_5	3	10	4	487	487	487	485	487	487	487	487	487	485	0
4.4	I_3_10_5_1	3	10	5	580	464	448	447	561	643	582	474	420	396	6.06
5.1	I_3_12_2_5	3	12	2	939	882	844	882	870	939	914	838	852	837	0.12
5.2	I_3_12_3_5	3	12	3	480	473	473	473	480	473	471	471	447	445	0.45
5.3	I_3_12_4_1	3	12	4	869	849	849	849	869	869	869	869	844	830	1.69
5.4	I_3_12_5_5	3	12	5	760	714	684	705	760	823	755	702	683	677	0.89
6.1	I_3_14_2_1	3	14	2	814	802	801	796	796	824	806	799	801	786	1.27
6.2	I_3_14_3_5	3	14	3	1,271	1,222	1,216	1,222	1,271	1,239	1,295	1,187	1,194	1,184	0.25
6.3	I_3_14_4_1	3	14	4	1,112	1,015	947	1,015	1,112	1,123	1,017	945	1,238	930	1.61
6.4	I_3_14_5_5	3	14	5	1,602	1,478	1,501	1,478	1,602	1,614	1,531	1,428	1,428	1,413	1.06
7.1	I_3_16_2_1	3	16	2	1,249	1,199	1,199	1,199	1,241	1,249	1,177	1,155	1,155	1,150	0.43
7.2	I_3_16_5_5	3	16	5	2,115	2,077	2,043	2,077	2,109	2,134	2,251	1,991	2,054	1,968	1.17

Πίνακας 6 Αποτελέσματα 3 Εργοστάσια

Το ίδιο παρατηρούμε και στα προβλήματα με 4 εργοστάσια και από 4 έως 16 εργασίες

4	File	FACTORIES	JOBS	MACHINES	NEHEDD	ILS	RSLS	RSLS_II	LS IN	LS MV	LS EX	HYLG	GA LS	BEST	RPD
1.1	I_4_4_2_1.	4	4	2	0	0	0	0	0	0	0	0	0	0	0
1.2	I_4_4_4_5.	4	4	4	0	0	0	0	0	0	0	0	0	0	0
1.3	I_4_4_5_1.	4	4	5	0	0	0	0	0	0	0	0	0	0	0
2.1	I_4_6_4_1.	4	6	4	29	29	29	29	29	29	29	29	29	29	0
2.2	I_4_6_5_5.	4	6	5	60	48	60	60	60	60	60	60	48	48	0
3.1	I_4_8_4_2.	4	8	4	159	159	159	159	159	147	159	159	147	147	0
3.2	I_4_8_5_5.	4	8	5	223	194	222	194	223	223	211	192	173	173	0
4.1	I_4_10_2_1	4	10	2	182	144	121	144	182	183	156	111	112	111	0
4.2	I_4_10_3_5	4	10	3	399	317	311	316	399	399	399	307	307	307	0
4.3	I_4_10_4_1	4	10	4	390	335	392	335	390	390	371	326	301	297	1.35
4.4	I_4_10_5_5	4	10	5	335	329	335	329	335	342	332	324	328	324	0
5.1	I_4_12_2_1	4	12	2	486	475	472	475	486	486	490	472	466	466	0
5.2	I_4_12_4_5	4	12	4	433	420	439	420	433	456	433	352	389	352	0
5.3	I_4_12_5_1	4	12	5	428	424	421	421	428	428	428	425	425	421	0
6.1	I_4_14_3_5	4	14	3	592	588	589	575	592	592	615	534	532	512	3.91
6.2	I_4_14_5_5	4	14	5	854	790	759	790	849	930	863	721	759	709	1.69
7.1	I_4_16_3_5	4	16	3	810	751	739	751	810	810	753	720	744	706	1.98
7.2	I_4_16_5_4	4	16	5	1,270	1,238	1,159	1,230	1,270	1,270	1,284	1,181	1,110	1,086	2.21

Πίνακας 7 Αποτελέσματα με 4 Εργοστάσια

Αξίζει όμως να παρουσιάσουμε τα αποτελέσματα των πιο μεγάλων προβλημάτων του συγκεκριμένου συνόλου δεδομένων, σχετικά με τους μεγαλύτερους χρόνους καθυστέρησης, ανεξάρτητα από τον αριθμό εργοστασίων και εργασιών.

BIG	File	FACTORIES	JOBS	MACHINES	NEHEDD	ILS	RSLS	RSLS_II	LS IN	LS MV	LS EX	HYLG	GA LS	BEST	RPD
1.2	I_2_10_2_3	2	10	2	1,054	1,049	1,049	1,032	1,052	1,036	1,063	1,010	1,029	1,010	0
1.2	I_2_10_4_2	2	10	4	1,161	1,133	1,075	1,133	1,161	1,161	1,138	1,064	1,080	1,058	0.57
1.3	I_2_10_4_2	2	10	4	1,161	1,133	1,133	1,127	1,161	1,161	1,138	1,058	1,080	1,058	0
2.1	I_2_12_3_4	2	12	3	1,402	1,323	1,269	1,323	1,399	1,402	1,278	1,253	1,248	1,236	0.97
2.2	I_2_12_4_3	2	12	4	1,316	1,242	1,228	1,242	1,316	1,316	1,242	1,223	1,228	1,223	0
2.3	I_2_12_5_3	2	12	5	1,171	1,162	1,163	1,151	1,156	1,265	1,220	1,119	1,151	1,099	1.82
3.1	I_2_14_3_2	2	14	3	1,841	1,805	1,749	1,805	1,734	1,904	1,902	1,639	1,639	1,632	0.43
3.2	I_2_14_3_5	2	14	3	1,857	1,843	1,843	1,843	1,848	1,857	1,847	1,751	1,752	1,744	0.4
3.3	I_2_14_4_2	2	14	4	2,228	2,105	2,110	2,105	2,116	2,229	2,331	2,002	2,009	1,994	0.4
4.1	I_2_16_3_4	2	16	3	2,279	2,199	2,102	2,199	2,266	2,279	2,218	2,067	2,072	2,067	0
4.2	I_2_16_4_1	2	16	4	2,483	2,437	2,395	2,437	2,482	2,483	2,433	2,318	2,372	2,300	0.78
4.3	I_2_16_5_1	2	16	5	2,664	2,548	2,427	2,523	2,625	2,664	2,595	2,251	2,266	2,285	-1.49
4.4	I_2_16_5_2	2	16	5	2,895	2,827	2,792	2,827	2,846	2,895	2,975	2,657	2,638	2,638	0
4.5	I_2_16_5_3	2	16	5	2,659	2,589	2,582	2,589	2,659	2,659	2,663	2,515	2,515	2,514	0.04
4.6	I_2_16_5_4	2	16	5	2,450	2,427	2,376	2,410	2,432	2,450	2,496	2,265	2,271	2,249	0.71
4.7	I_2_16_5_5	2	16	5	2,958	2,909	2,759	2,857	2,947	2,958	2,890	2,749	2,655	2,680	-0.93
5.1	I_3_14_3_5	3	14	3	1,271	1,222	1,216	1,222	1,271	1,239	1,295	1,187	1,194	1,184	0.25
5.2	I_3_14_4_1	3	14	4	1,112	1,015	947	1,015	1,112	1,123	1,017	945	1,238	930	1.61
5.3	I_3_14_5_5	3	14	5	1,602	1,478	1,501	1,478	1,602	1,614	1,531	1,428	1,428	1,413	1.06
6.1	I_3_16_2_1	3	16	2	1,249	1,199	1,199	1,199	1,241	1,249	1,177	1,155	1,155	1,150	0.43
6.2	I_3_16_5_5	3	16	5	2,115	2,077	2,043	2,077	2,109	2,134	2,251	1,991	2,054	1,968	1.17

Πίνακας 8 Αποτελέσματα σε προβλήματα μεγαλύτερου μεγέθους της κατηγορίας small

Στον πίνακα 9 παρατηρούμε ότι οι χρόνοι καθυστέρησης των συγκεκριμένων προβλημάτων είναι οι μεγαλύτεροι από το σύνολο δεδομένων των Naderi και Ruiz. Στα μεγαλύτερα προβλήματα υπάρχει περιθώριο καλύτερης βελτιστοποίησης των χρόνων καθυστέρησης και όπως μπορούμε να διακρίνουμε στο πρόβλημα 4.3 (I_2_16_5_1) με 2 εργοστάσια 16 εργασίες και 5 μηχανές ο αλγόριθμος HYLG έδωσε μια καλύτερη λύση από αυτή της βιβλιογραφίας. Το ίδιο ισχύει και για τον αλγόριθμο GA_LS ο οποίος μας δίνει καλύτερη λύση στο πρόβλημα 4.7 (I_2_16_5_5) με 2 εργοστάσια 16 εργασίες και 5 μηχανές. Οι αρνητικές τιμές στην σχετική ποσοστιαία απόκλιση RPD που σημαίνονται με μωβ χρώμα, μας φανερώνουν καλύτερες λύσεις από αυτές τις βιβλιογραφίας.

Αναλύοντας περισσότερο τα αποτελέσματα του πρώτου συνόλου δεδομένων και πριν παρουσιαστούν τα αποτελέσματα του συνόλου Taillard, παρουσιάζονται μερικά στατιστικά στοιχεία σχετικά με την απόδοση και την συμπεριφορά των μεθόδων επίλυσης στις οποίες αναφέρεται η παρούσα εργασία.

Από τις μέσες τιμές των αποτελεσμάτων του πρώτου συνόλου δεδομένων και όπως αυτές παρουσιάζονται στον πίνακα 10 παρατηρούμε ότι οριακά τους καλύτερους χρόνους μας τους δίνει ο γενετικός αλγόριθμος GA και ακολουθεί ο άπληστος αλγόριθμος HYLG.

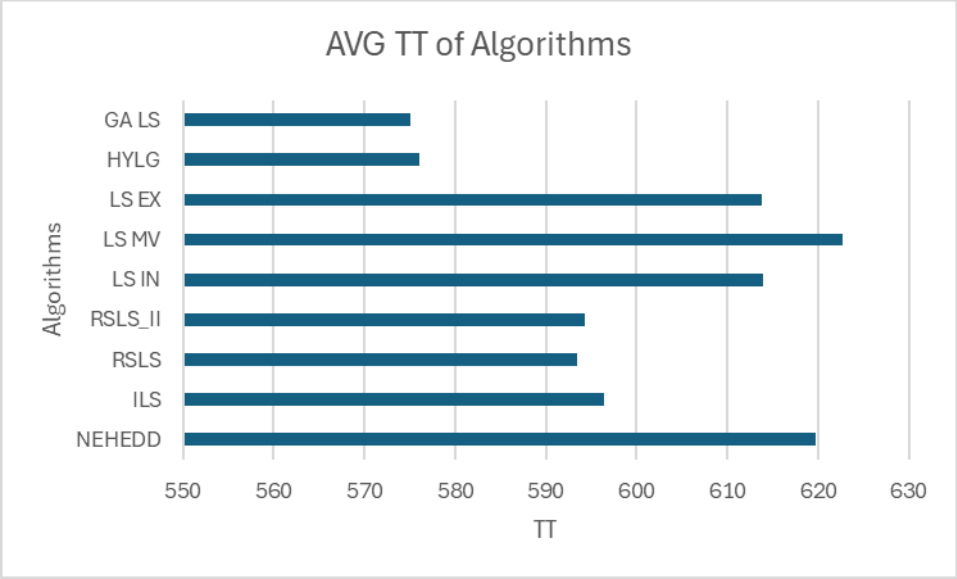
NEHEDD	ILS	RSLS	RSLS_II	LS IN	LS MV	LS EX	HYLG	GA LS	BEST
619.76	596.48	593.5	594.32	614.02	622.73	613.77	**576**	**575.02**	564.65

Πίνακας 9 Μέσος όρος συνολικής καθυστέρησης για όλα τα προβλήματα της κατηγορίας small, ανά τρόπο επίλυσης

Παρόλα αυτά βέβαια θα πρέπει να αναφερθεί ότι η απόδοση αυτών των αλγορίθμων εξαρτάται και από τα κριτήρια τερματισμού τους. Για όλους του αλγόριθμους έχουν εφαρμοστεί ως κριτήριο τερματισμού, οι επαναλήψεις εκτέλεσης όπως ορίζονται για τον καθένα. Για να τερματίζονται οι αλγόριθμοι μέσα σε ένα αποδεκτό χρονικό διάστημα προσαρμόζονται και οι επαναλήψεις εκτέλεσης τους. Για παράδειγμα:

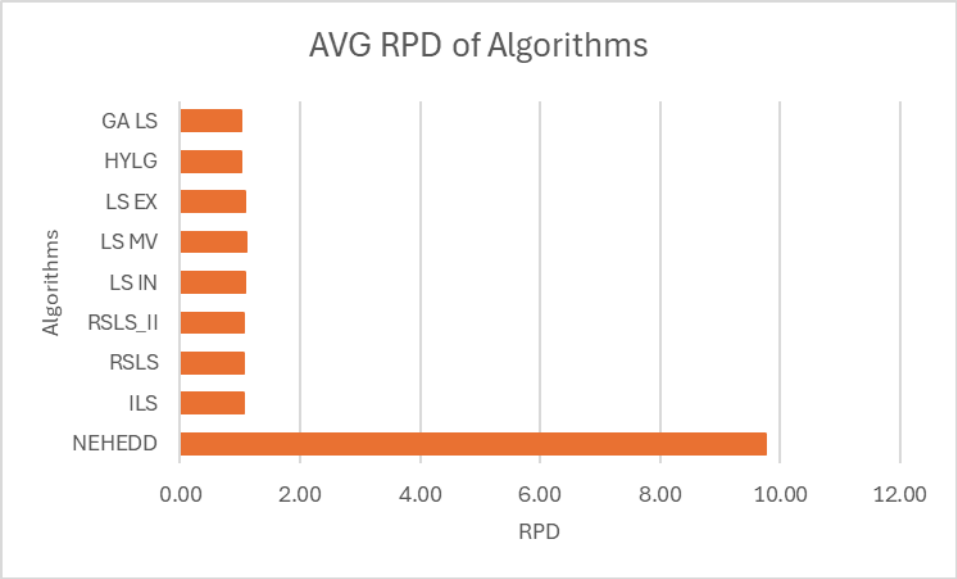
- αλγόριθμος ILS ο οποίος εφαρμόζει μια απλή τοπική αναζήτηση ανταλλάσσοντας μεμονωμένα εργασίες μεταξύ εργοστασίων ο οποίος βασίζεται και στον παράγοντα τύχη μπορεί σε μικρό διάστημα να εκτελέσει πολλές επαναλήψεις γι' αυτό του έχουμε ορίσει να τερματίζει μετά από 10000 επαναλήψεις.
- Οι αλγόριθμοι RSLS και RSLS II ανταποκρίνονται επίσης πολύ καλά και οι λύσεις που μας επιστρέφουν δεν απέχουν πάρα πολύ από τις καλύτερες λύσεις που μας δίνουν οι αλγόριθμοι HYLG και GA LS. Οι συγκεκριμένοι αλγόριθμοι κάνουν περισσότερους υπολογισμούς από τον ILS και έχουν οριστεί να εκτελούν 1000 επαναλήψεις.
- Οι αλγόριθμοι LS IN, LS MV και LS EX αποτελούν τις τοπικές αναζητήσεις που εκτελούνται στον γενετικό αλγόριθμο και είναι απαιτητικοί αλγόριθμοι οπότε οι επαναλήψεις τους ορίζονται σε 100.
- Ο αλγόριθμος HYLG εκτελεί 1000 φορές τις διαδικασίες βελτιστοποίησης. Παρατηρώντας την καλή απόδοση του αλγόριθμου ILS τον εκτελούμε σε κάθε επανάληψη. Η συγκεκριμένη λειτουργία βελτίωσε την απόδοση του αλγόριθμου και είναι και το σημείο που διαφοροποιείται από τις λύσεις της βιβλιογραφίας. Οι τελεστές T και d του αλγόριθμου HYLG έχουν οριστεί ως εξής: T για την θερμοκρασία της προσομοιωμένης ανόπτησης σε 0,8 και d το ποσοστό των εργασιών που αφαιρούνται και τοποθετούνται ξανά σε καλύτερες θέσεις είναι 0,4 (σε ένα πρόβλημα με 20 εργασίες και 2 εργοστάσια, ο αριθμός των εργασιών είναι 0,4*20/2 = 4) . Οι συγκεκριμένες τιμές έδωσαν τις καλύτερες λύσεις όπως θα δούμε και στην ανάλυση των δεδομένων TAILARD.
- Ο γενετικό αλγόριθμος GA_LS έχει οριστεί μετά από σχετικές δοκιμές να τρέχει σε 50 γενεές με 200 επαναλήψεις τοπικής αναζήτησης σε κάθε γενεά.

Η εκτέλεση των αλγορίθμων για κάθε πρόβλημα κυμαίνεται από μερικά δευτερόλεπτα για τα μικρότερα προβλήματα της κατηγορίας small μέχρι αρκετές ώρες για τα μεγαλύτερα προβλήματα της κατηγορίας large.

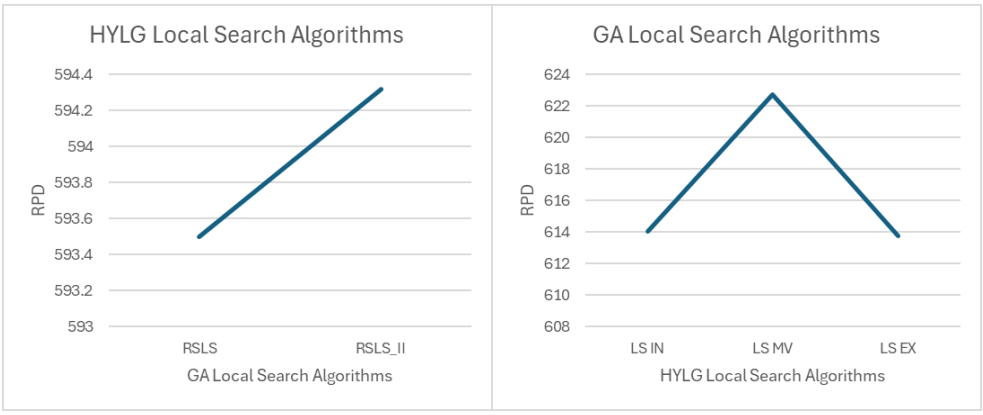


Εικόνα 15 Μέσες Τιμές Λύσεων

Στην Εικόνα 15 παρατηρείται ότι οι καλύτερες λύσεις δίνονται από τους αλγόριθμους HYLG και GA_LS. Επίσης όπως παρατηρείται στην εικόνα 16 οι μεταευρετικοί αλγόριθμοι δίνουν πολύ καλύτερα αποτελέσματα από τις ευρετικές λύσεις.

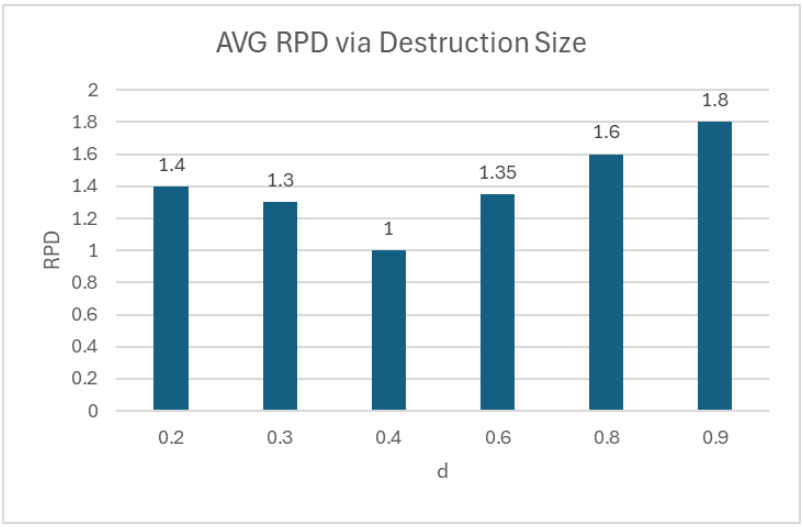


Εικόνα 16 Μέσες τιμές RPD ανά προσέγγιση επίλυσης



Εικόνα 17 Local Search Algorithms

Συγκρίνοντας μεταξύ τους, τους αλγόριθμους τοπικής αναζήτησης οι οποίοι ενσωματώνονται στις λύσεις HYLG και GA_LS, παρατηρείται ότι παίρνουμε καλύτερα αποτελέσματα από τον αλγόριθμο RSLS_II σε σχέση με τον αλγόριθμο RSLS. Για τους αλγόριθμους που συμμετέχουν στην λύση γενετικών αλγόριθμων με τοπική αναζήτηση παρατηρείται ότι ο αλγόριθμος που συμβάλει λιγότερο στην εύρεση των λύσεων είναι ο αλγόριθμος τοπικής αναζήτησης με μετακίνηση.



Εικόνα 18 HYLG vs GA LS and HYLG d

Στην εικόνα 18 παρατηρείται ότι ο αλγόριθμος HYLG επηρεάζεται δραστικά από την τιμή της μεταβλητής d η οποία ορίζει τον αριθμό των εργασιών που θα αφαιρούνται από μια λύση και θα τοποθετούνται ξανά στην καλύτερη θέση της ακολουθίας. Παρατηρούμε ότι οι τιμές τις d δεν θα πρέπει να είναι ούτε πολύ μικρές αλλά ούτε πολύ μεγάλες. Άρα μια τιμή κοντά στο 0,4 με 0,5 είναι οι καλύτερα αποδεκτές τιμές.

Τα αποτελέσματα που χρησιμοποιήθηκαν από το δείγμα των δεδομένων TAILARD παρουσιάζονται στον πίνακα 19. Και σε αυτές τις περιπτώσεις παρατηρούμε παρόμοια συμπεριφορά με τα δεδομένα Naderi και Ruiz.

Taillard	File	FACTORIES	JOBS	MACHINES	NEHEDD	ILS	RSLS	RSLS_II	LS IN	LS MV	LS EX	HYLG	GA LS	BEST	RPD
1.1	Ta013_2	2	20	10	2,350	2,209	2,343	2,209	2,350	2,350	2,358	2,208	2,167	2,087	3.83
1.2	Ta064_2	2	100	5	55,871	54,962	56,231	54,962	55,762	55,871	55,251	54,006	50,255	50,677	-0.83
1.3	Ta091_2	2	200	10	265,488	265,464	278,047	264,920	264,802	265,488	265,300	260,857	251,551	252,671	-0.44
2.1	Ta003_3	3	20	5	2,022	1,947	1,880	1,879	1,929	2,063	2,192	1,608	1,634	1,599	0.56
2.2	Ta053_3	3	50	20	9,623	9,319	8,070	9,185	9,243	9,449	8,865	6,885	6,244	6,287	-0.68
2.3	Ta069_3	3	100	5	44,791	44,538	42,724	44,454	44,477	44,791	44,807	42,262	40,048	40,580	-1.31
3.1	Ta004_4	4	20	5	1,587	1,513	1,412	1,498	1,587	1,614	1,627	1,405	1,427	1,341	4.77
4.1	Ta029_4	6	20	10	1,020	956	890	956	1,020	1,020	970	821	817	792	3.16
4.2	Ta015_6	6	20	10	1,240	1,118	1,092	1,118	1,240	1,240	1,246	906	956	812	11.58
4.3	Ta088_5	6	20	5	1,020	989	989	989	1,020	1,039	986	910	951	886	2.71
4.4	Ta016_6	6	20	10	1,434	1,341	1,274	1,341	1,405	1,434	1,409	1,034	1,168	1,018	1.57
5.1	Ta010_7	7	20	5	753	692	652	691	753	787	787	595	715	591	0.68
5.2	Ta005_7	7	20	5	755	730	752	730	755	755	750	701	745	691	1.45
5.3	Ta003_7	7	20	5	1,086	1,004	994	1,004	1,086	1,049	1,060	779	876	746	4.42
5.4	Ta004_7	7	20	5	1,088	1,025	904	1,025	1,088	1,087	1,087	839	948	837	0.24
5.5	Ta071_7	7	100	10	17,088	16,861	18,085	16,798	16,977	17,088	17,035	14,776	15,024	14,580	1.34
					26991.07	**26830.60**	**27599.73**	**26770.00**	**26876.27**	**26985.00**	**26891.47**	**25892.27**	**24890.60**		

Πίνακας 10 Αποτελέσματα δείγματος TAILARD

Αυτό που αξίζει να σημειωθεί αφορά τα πολύ μεγάλα προβλήματα, στα οποία και παίρνουμε καλύτερες τιμές από αυτές τις βιβλιογραφίας. Παρατηρούμε ότι στις λύσεις των προβλημάτων Ta064_2, Ta091_2, Ta053_3 και Ta069_3 οι χρόνοι καθυστέρησης είναι τεράστιοι σε σχέση με τα υπόλοιπα προβλήματα. Σε αυτά τα προβλήματα φαίνεται ο γενετικός αλγόριθμος να αποδίδει καλύτερα και να βελτιώνει το πρόβλημα περαιτέρω.

Στην εικόνα 19 παρουσιάζονται τα αποτελέσματα της εκτέλεσης όλων των αλγόριθμων για τα προβλήματα Ta091_2 και Ta069_3, εμφανίζοντας επιπλέον τον συνολικό χρόνο εκτέλεσης.

File	FACTORIES	JOBS	MACHINES	NEHEDD	ILS	RSLS	RSLS_II	LS IN	LS MV	LS EX	HYLG	GA LS	BEST	RPD
Ta091_2	2	200	10	265,488	265,464	278,047	264,920	264,802	265,488	265,300	260,857	251,551	252,671	-0.44

Συνολικός χρόνος εκτέλεσης: **41.008** seconds

File	FACTORIES	JOBS	MACHINES	NEHEDD	ILS	RSLS	RSLS_II	LS IN	LS MV	LS EX	HYLG	GA LS	BEST	RPD
Ta069_3	3	100	5	44,791	44,538	42,727	44,454	44,477	44,791	44,807	42,262	40,048	40,580	-1.31

Συνολικός χρόνος εκτέλεσης: **7.178** seconds

Εικόνα 19 Αποτέλεσμα μεγάλων προβλημάτων

Σχετικά με τις παραμέτρους του γενετικού αλγόριθμου και μέχρι να καταλήξουμε στον αριθμό των γενεών που επιστρέφουν καλύτερα αποτελέσματα έγιναν δοκιμές με διαφορετικές τιμές. Ξεκινώντας ορίσαμε τις γενεές 20 και τις επαναλήψεις τοπικής αναζήτησης 2. Στην ίδια εκτέλεση αλλάξαμε και τις τιμές του T και του d του αλγόριθμου HYLG σε T=0,5 και d=0,5.

File	FACTORIES	JOBS	MACHINES	NEHEDD	ILS	RSLS	RSLS_II	LS IN	LS MV	LS EX	HYLG	GA LS	BEST	RPD
Ta053_3	3	50	20	9,623	9,319	8,682	9,185	9,403	9,449	8,865	7,208	6,936	6,287	10.32

Εικόνα 20 Γενετικός 20 GEN 2 LS

Ακολούθησε μια εκτέλεση με τις τιμές των γενεών 100 αφήνοντας τις επαναλήψεις σε 2 ανά γενεά. Εδώ αλλάξαμε τις μεταβλητές του αλγόριθμου T σε 0,4 και d σε 0,8. Παρατηρούμε ότι οι λύση του γενετικού έχει βελτιωθεί ενώ η λύση του HYLG έχει χειροτερέψει.

File	FACTORIES	JOBS	MACHINES	NEHEDD	ILS	RSLS	RSLS_II	LS IN	LS MV	LS EX	HYLG	GA LS	BEST	RPD
Ta053_3	3	50	20	9,623	9,319	7,826	9,185	9,332	9,449	8,865	7,325	6,453	6,287	2.64

Εικόνα 21 Γενετικός 100 GEN 2 LS

Τελικώς καταλήγουμε στις 50 γενεές με 200 επαναλήψεις της τοπικής αναζήτησης. Τιμές που μας έδωσαν επανειλημμένα καλύτερες λύσεις. Και στον αλγόριθμο HYLG καταλήξαμε στις τιμές T=0,8 και d = 0,4, οι οποίες μας έδωσαν τα καλύτερα αποτελέσματα κατά την διάρκεια των πειραμάτων.

File	FACTORIES	JOBS	MACHINES	NEHEDD	ILS	RSLS	RSLS_II	LS IN	LS MV	LS EX	HYLG	GA LS	BEST	RPD
Ta053_3	3	50	20	9,623	9,319	8,070	9,185	9,243	9,449	8,865	6,885	6,244	6,287	-0.68

Εικόνα 22 Γενετικός 50 GEN 200 LS

Κλείνοντας την παρουσίαση και ανάλυση των αποτελεσμάτων καταλήγουμε ότι ο γενετικός αλγόριθμος δείχνει να έχει την καλύτερη απόδοση στην βελτίωση των χρόνων καθυστέρησης, κυρίως στα μεγαλύτερα προβλήματα. Προτιμήθηκε το κριτήριο τερματισμού να είναι η διαδοχικές επαναλήψεις εκτέλεσης των αλγορίθμων. Προτείνεται βέβαια μια επέκταση των λύσεων με κριτήριο τερματισμού τον χρόνο εκτέλεσης ο οποίος θα δίνεται λαμβάνοντας υπόψη το μέγεθος (τα εργοστάσια, τις μηχανές και τις εργασίες) των προβλημάτων.

8 Συμπεράσματα και Μελλοντικές Προκλήσεις

Η παρούσα εργασία μελετά το πρόβλημα χρονοπρογραμματισμού κατανεμημένων ροών εργασιών βάσει μεταθέσεων με περιορισμούς ημερομηνιών τερματισμού εργασιών (DPFSP-DD). Το πρόβλημα DPFSP-DD αποτελεί ένα κρίσιμο ερευνητικό πεδίο, ιδίως στον τομέα της κατανεμημένης παραγωγής. Το κύριο κριτήριο που μελετήθηκε αφορά τους χρόνους καθυστέρησης των εργασιών σε σύγκριση με τους επιθυμητούς χρόνους τερματισμού (due dates). Η ελαχιστοποίηση της καθυστέρησης τερματισμού των εργασιών και της τήρησης των προθεσμιών (due dates) αυξάνει την πολυπλοκότητα του προβλήματος, καθιστώντας το εξαιρετικά απαιτητικό τόσο από υπολογιστικής όσο και από πρακτικής πλευράς. [1]

Η έρευνα έδειξε πως η συνδυασμένη χρήση πολλαπλών εργαλείων βελτιστοποίησης, όπως υβριδικών αλγορίθμων, μπορεί να αποδώσει λύσεις υψηλής ποιότητας. Έχουν αναπτυχθεί δύο κύριοι υβριδικοί αλγόριθμοι, ένας άπληστος αλγόριθμος και ένας γενετικός αλγόριθμος στους οποίους προστέθηκαν μέθοδοι τοπικής αναζήτησης για να βελτιωθεί η αποδοτικότητα τους. Τα αποτελέσματα των πειραμάτων για τα μικρότερα όσο και για τα μεγαλύτερα προβλήματα του συνόλου δεδομένων, έδειξαν ότι οι αλγόριθμοι που αναπτύχθηκαν δίνουν λύσεις εξίσου αποδοτικές σε σχέση με τις καθολικά βέλτιστες και σε αρκετές περιπτώσεις, κυρίως σε μεγαλύτερα προβλήματα, βελτιώνουν περαιτέρω ορισμένα προβλήματα.

Από την βιβλιογραφία διαπιστώσαμε ότι υπάρχουν ελάχιστες αναφορές για την επίλυση του προβλήματος ελαχιστοποίησης της συνολικής καθυστέρησης για το DPFSP-DD, δημιουργώντας προκλήσεις για περαιτέρω μελέτη του συγκεκριμένου προβλήματος, διευρύνοντας τα πεδία της έρευνας. Συγκεκριμένα μπορεί να επεκταθεί η έρευνα σε ρεαλιστικά προβλήματα συνδυάζοντας θεωρία, πειραματική αξιολόγηση και πρακτική εφαρμογή [23]. Επίσης θα μπορούσαν να εξεταστούν ξεχωριστά η μέγιστη καθυστέρηση, η συνολική καθυστέρηση καθώς και η μέγιστη αργοπορία και ο αριθμός καθυστερημένων εργασιών, για να δίνεται η δυνατότητα προσαρμογής των χρονοδιαγραμμάτων σε πραγματικό χρόνο και να εντοπίζονται τα σημεία με την μεγαλύτερη καθυστέρηση. Καθώς οι επιχειρήσεις επιδιώκουν την εξισορρόπηση του φόρτου εργασίας μεταξύ των εργοστασίων, κρίνεται επιτακτική η ανάγκη για περαιτέρω μελέτη των στόχων που συσχετίζονται με τον μέγιστο μεταξύ των χρόνων ολοκλήρωσης κάθε εργοστασίου ή ο μέσος χρόνος παραγωγής κάθε εργοστασίου, σε ένα κατανεμημένο σύστημα, κάτι το οποίο έχει αντιμετωπιστεί ελάχιστα. Ορισμένοι δείκτες ποιότητας που προτείνεται να εξεταστούν είναι: *Τα ποσοστά έγκαιρης παράδοσης εργασιών* και η *Εξισορρόπηση φόρτου εργασίας μεταξύ εργοστασίων*.

Η διερεύνηση των στοιχείων και των δεικτών ποιότητας που αναφέρονται παραπάνω σε συνδυασμό με την αξιοποίηση τεχνικών μηχανικής μάθησης για την πρόβλεψη και βελτιστοποίηση των due dates μπορεί να επιταχύνει τις διαδικασίες και να αυξήσει την αποδοτικότητα των μεθόδων βελτιστοποίησης.

Τέλος πέραν των ευρετικών και των υβριδικών μεθόδων βελτιστοποίησης των προβλημάτων DPFSP-DD, οι οποίες κατάφεραν να προσεγγίσουν με επιτυχία τα συγκεκριμένα προβλήματα, προτείνεται η επέκταση της μελέτης με

τεχνικές προγραμματισμού με περιορισμούς (Constraint Programming - CP) οι οποίες έχουν αποδειχθεί αποτελεσματικές για τη διαχείριση σύνθετων χρονικών περιορισμών, ειδικά σε σενάρια όπου η παραβίαση των due dates επιφέρει σημαντικές ποινές. Η χρήση εξειδικευμένων χαρακτηριστικών του CP, όπως οι μεταβλητές διαστημάτων και οι περιορισμοί μη-επικάλυψης, συμβάλλουν στη διαμόρφωση μοντέλων που αποτυπώνουν με ακρίβεια το πρόβλημα, οδηγώντας σε βελτιστοποιημένες λύσεις. [16]

Η μελέτη του DPFSP-DD βρίσκεται σε συνεχή εξέλιξη, με τις ημερομηνίες τερματισμού εργασιών (due dates) να αποτελούν κρίσιμο στοιχείο για τη διαμόρφωση αποτελεσματικών και ευέλικτων συστημάτων κατανεμημένης παραγωγής.

9 Εκτέλεση Αλγόριθμων

9.1 Αρχεία και Φάκελοι του Κώδικα

Ο κώδικας αποτελείται από τα παρακάτω αρχεία και φακέλους:

main.py : Αρχείο εκτέλεσης των αλγορίθμων

loadfile.py : Περιέχει συναρτήσεις για την ανάγνωση στιγμιότυπων από αρχείο

utils.py : Βοηθητικές συναρτήσεις

nehedd_new.py : Αλγόριθμος NEHedd

calcShedule.py : Υπολογισμός απόδοσης μιας ακολουθίας (Επιστρέφει την συνολική καθυστέρηση)

ils.py : Αλγόριθμος τοπικής αναζήτησης, ο οποίος μετακινεί δυο εργασίες μεταξύ τους

rsls.py : Αλγόριθμος που περιγράφεται στην ενότητα 6.2.1

rsls_II.py : Αλγόριθμος που περιγράφεται στην ενότητα 6.2.2

ls_move_job.py : Αλγόριθμος που περιγράφεται στην ενότητα 6.4.4.3

ls_insertion_job.py : Αλγόριθμος που περιγράφεται στην ενότητα 6.4.4.3

ls_exchange_job.py : Αλγόριθμος που περιγράφεται στην ενότητα 6.4.4.3

ig.py : Συνολικός αλγόριθμος HYLG

ga_ls.py : Συνολικός αλγόριθμος GA_LS

Best_Result_small.csv : Αρχείο με τα καθολικά βέλτιστα αποτελέσματα για τα μικρότερα προβλήματα

Best_Result_large.csv : Αρχείο με τα καθολικά βέλτιστα αποτελέσματα για τα μεγαλύτερα προβλήματα

/RESOURCES : Φάκελος με το τελικό κείμενο της διπλωματικής εργασίας

/dataSet/Large : Σύνολα δεδομένων των προβλημάτων. Αφορά μεγάλα προβλήματα (Taillard)

/dataSet/Small : Σύνολα δεδομένων των προβλημάτων. Αφορά μεγάλα προβλήματα (Ruiz, Stuetzle)

9.2 Εκτέλεση του Κώδικα

Η συγγραφή και η εκτέλεση του κώδικα έγινε στο Visual Studio Code και σε γλώσσα python κάνοντας χρήση της πλατφόρμας Anaconda. Επιπλέον χρησιμοποιήθηκαν οι βιβλιοθήκες: copy, os, csv, time, math, random και η εξωτερική βιβλιοθήκη rich.

Ο κώδικας εκτελείται απευθείας από το VS code εκτελώντας το αρχείο main.py. Οι αλγόριθμοι επιλύουν όποια αρχεία προβλημάτων βρίσκουν τοποθετημένα στον φάκελο dataSet, οπότε θα πρέπει να τοποθετούνται τα αρχεία των προβλημάτων προς επίλυση στον συγκεκριμένο φάκελο. Αρχικά θα υπάρχουν στον φάκελο dataSet δύο μικρά προβλήματα. Ένα από το σύνολο δεδομένων μικρών προβλημάτων και ένα από το σύνολο δεδομένων μεγάλων προβλημάτων. Τα τελικά αποτελέσματα παρουσιάζονται με την βοήθεια της βιβλιοθήκης rich.

9.3 Χαρακτηριστικά Συστήματος

Τα χαρακτηριστικά του υπολογιστή στον οποίο εκτελέστηκαν όλες οι δοκιμές των αλγόριθμων είναι τα παρακάτω:

Processor	Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz 2.81 GHz
Installed RAM	16.0 GB
System type	64-bit operating system, x64-based processor
Operating System	Windows 10 Pro
Python –version	Python 3.11.5
Anaconda –version	conda 23.7.4

9.4 Git Repository

Ο πλήρης κώδικας που χρησιμοποιήθηκε για την ανάπτυξη και την ανάλυση αυτής της μελέτης είναι διαθέσιμος στο αποθετήριο GitHub στη διεύθυνση: https://github.com/dimkios/DPFSP.git

Η εκτέλεση του κώδικα γίνεται όπως φαίνεται στην εικόνα

```
(py312) ➜  DPFSP git:(main) ✗ python main.py

fileName = I_2_4_2_2.txt
ΤΑΞΙΝΟΜΗΣΗ ΕΡΓΑΣΙΩΝ ΣΥΜΦΩΝΑ ΜΕ ΤΑ DUE DATES [3, 1, 0, 2]
ILS : 112065 timer: 2.000016927719116
RSLS : 99925 timer: 2.0000100135803223
RSLS2 : 92324 timer: 2.0000789165496826
LSIN : 117460 timer: 2.0000078678131104
LSMV : 35117 timer: 2.000027894973755
LSXC : 36680 timer: 2.000030040740967
HYLG : 46018 timer: 2.0001089572906494
GALS : 3120 timer: 2.000088930130005
fileName = Ta001_2.txt
ΤΑΞΙΝΟΜΗΣΗ ΕΡΓΑΣΙΩΝ ΣΥΜΦΩΝΑ ΜΕ ΤΑ DUE DATES [2, 12, 16, 5, 8, 13, 15, 18, 14, 7, 9, 11, 10, 1, 3, 0, 6, 19, 4, 17]
ΕΚΤΕΛΕΣΗ ΑΛΓΟΡΙΘΜΩΝ για DPSFSP... ━━━━━━━━━━━━━━━━━━ 50% -:--:--
```

Εικόνα 23 Εκτέλεση του κώδικα

Βιβλιογραφία

[1] B. Naderi και R. Ruiz, «The distributed permutation flowshop scheduling problem,» *Computers & operations research, 2010, 37.4: 754-768,* 2010.

[2] R. Graham, E. Lawler, J. Lentsra και A. Rinnooy Kan, «Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey,» *Annals of discrete mathematics. Elsevier, 1979. p. 287-326,* 1979.

[3] R. Ruiz και M. Concepcion, «A comprehensive review and evaluation of permutation flowshop heuristics to minimize flowtime,» *European journal of operational research, 2005, 165.2: 479-494,* 2005.

[4] E. Nowicki και C. Smutnicki, «A fast taboo search algorithm for the job shop problem.,» *Management science, 1996, 42.6: 797-813,* 1996.

[5] J. Grabowski και M. Wodecki, «A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion,» *Computers & Operations Research, 2004, 31.11: 1891-1909,* 2004.

[6] I. Osman και C. Potts, «Simulated annealing for permutation flow-shop scheduling,» *Omega, 1989, 17.6: 551-557.,* 1989.

[7] C. R. Reeves, «A Genetic Algorithm For Flowshop Sequencing,» *Computers & operations research, 1995, 22.1: 5-13,* 1995.

[8] R. Ruiz, C. Maroto και J. Alcaraz, «Two new robust genetic algorithms for the flowshop scheduling problem,» *Omega, 2006, 34.5: 461-476.,* 2006.

[9] C. Rajendran και H. Ziegler, «Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs,» *European Journal of Operational Research,* 2004.

[10] E. Nowicki και C. Smutnicki, «Some new ideas in TS for job shop scheduling. Metaheuristic Optimization via Memory and Evolution: Tabu Search and Scatter Search,» *Kluwer Academic,* 2005.

[11] G. Onwubolu και D. Davendra, «Scheduling flow shops using differential evolution algorithm,» *European Journal of Operational research, 2006, 171.2: 674-692.,* 2006.

[12] M. F. e. a. Tasgetiren, «A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem,» *European journal of operational research,* 2007.

[13] R. Ruiz και T. Stuetzle, «A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem,» *European journal of operational research, 2007, 177.3: 2033-2049,* 2007.

[14] E. Vallada και R. Ruiz, «Cooperative metaheuristics for the permutation flowshop scheduling problem,» *European Journal of Operational Research,* 2009.

[15] J. Kuhpfahl, Job shop scheduling with consideration of due dates: Potentials of local search based solution techniques, Springer, 2015.

[16] C. Gogos, «Solving the Distributed Permutation Flow-Shop Scheduling Problem Using Constrained Programming,» *Applied Sciences, 2023, 13.23: 12562,* 2023.

[17] G. Minella, R. Ruiz και M. Ciavotta, «A Review and Evaluation of Multiobjective Algorithms for the Flowshop Scheduling Problem,» *INFORMS Journal on Computing, 2008, 20.3: 451-471,* 2008.

[18] P. Perez-Gonzalez και J. M.Framinan, «A review and classification on distributed permutation flowshop scheduling problems,» *European Journal of Operational Research, 2024, 312.1: 1-21,* 2024.

[19] L. F. Gelders και N. Sambandam, «Four simple heuristics for scheduling a flow-shop,» *The International Journal of Production Research,* 1978.

[20] M. Widmer και A. Hertz, «A new heuristic method for the flow shop sequencing problem,» *European Journal of Operational Research,* 1989.

[21] S. Parthasarathy και C. RAjendran, «Scheduling to minimize mean tardiness and weighted mean tardiness in flowshop and flowline-based manufacturing cell,» *Computers & Industrial Engineering,* 1998.

[22] S. Hasija και C. Rajendran, «Scheduling in flowshops to minimize total tardiness of jobs,» *International Journal of Production Research, 2004, 42.11: 2289-2301,* 2004.

[23] A. Khare και S. Agrawal, «Effective heuristics and metaheuristics to minimise total tardiness for the distributed permutation flowshop scheduling problem,» *International Journal of Production Research, 2021, 59.23: 7266-7282,* 2021.

[24] C. A. Floudas και X. Lin, «Mixed Integer Linear Programming in Process Scheduling: Modeling, Algorithms, and Applications,» *Annals of Operations Research, 2005, 139: 131-162,* 2005.

[25] M. R. Garey, D. S. Johnson και R. Sethi, «The Complexity of Flowshop and Jobshop Scheduling,» *Mathematics of operations research, 1976, 1.2: 117-129,* 1976.

[26] H. Campbell, R. Dudek και M. Smith, «A heuristic algorithm for the n job, m machine sequencing problem,» *Management science, 1970, 16.10: B-630-B-637,* 1970.

[27] D. Dannenbring, «An evaluation of flow shop sequencing heuristics,» *Management science, 1977, 23.11: 1174-1182,* 1977.

[28] M. Nawaz, E. Enscore JR και I. Ham, «A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem,» *Omega,* 1983.

[29] R. Leisten, «Flowshop sequencing problems with limited buffer storage,» *The International Journal of Production Research, 1990, 28.11: 2085-2100.,* 1990.

[30] F. A. Ogbu και D. Smith, «The application of the simulated annealing algorithm to the solution of the n/m/Cmax flowshop problem,» *Computers & Operations Research,* 1990.

[31] M. Ben-Daya και M. Al-Fawzan, «A tabu search approach for the flow shop scheduling problem,» *European journal of operational research, 1998, 109.1: 88-95,* 1998.

[32] J. Framinan, R. Ruiz-Usano και R. Leisten, «Sequencing CONWIP flow-shops: analysis and heuristics,» *International Journal of Production Research, 2001, 39.12: 2735-2749,* 2001.

[33] E. Taillard, «Some efficient heuristic methods for the flow shop sequencing problem,» *European journal of Operational research, 1990, 47.1: 65-74,* 1990.

[34] K. Yeong-Dae Kim, «Heuristics for flowshop scheduling problems minimizing mean tardiness,» *Journal of the Operational Research Society, 1993, 44.1: 19-28,* 1993.

[35] E. Vallada, R. Ruiz και G. Minella, «Minimising total tardiness in the m-machine flowshop problem: A review and evaluation of heuristics and metaheuristics,» *Computers & Operations Research, 2008, 35.4: 1350-1373,* 2008.

[36] V. Fernandez-Viagas και J. M. Framinan, «NEH-based heuristics for the permutation flowshop scheduling problem to minimise total tardiness,» *Computers & Operations Research, 2015, 60: 27-36,* 2015.

[37] J. Thompson, Heuristics: An Overview - ISBN 978-3-319-91085-7, Springer International Publishing AG, 2023.

[38] H. Ramalhinho Lourenço, O. C. Martin και T. Stuetzle, HANDBOOK OF METAHEURISTICS (CHAPTER 5), Springer, 2010.

[39] R. Ruiz, P. Quan-Ke και P. Alfaro-Fernandez, «Iterated search methods for earliness and tardiness minimization in hybrid flowshops with due windows,» *Computers & Operations Research, 2017, 80: 50-60,* 2016.

[40] T. Stuetzle, «Applying Iterated Local Search to the Permutation Flow Shop Problem,» *Technical Report AIDA-98-04, FG Intellektik, TU Darmstadt,* 1998.

[41] S. Kirkpatrick, C. D. Gelatt Jr και M. P. VECCHI, «Optimization by Simulated Annealing,» *science, 1983, 220.4598: 671-680,* 1983.

[42] P. Pourhejazy, C.-Y. Cheng, K.-C. Ying και N. H. Nam, «Meta-Lamarckian-based iterated greedy for optimizing,» *Annals of Operations Research, 2023, 322.1: 125-146,* 2023.

[43] T. Murata, H. Ishibuchi και H. Tanaka, «Genetic Algorithms for Flowshop Scheduling Problems,» *Computers & Industrial Engineering, 1996, 30.4: 1061-1071,* 1996.

[44] J. Gao και R. Chen, «A hybrid genetic algorithm for the distributed permutation flowshop scheduling problem,» *International Journal of Computational Intelligence Systems, 2011, 4.4: 497-508,* 2011.

[45] A. Ghad, «PyGAD - Python Genetic Algorithm!,» [Ηλεκτρονικό]. Available: https://pygad.readthedocs.io/en/latest/.

[46] «Google OR-Tools - The Job Shop Problem,» Google, 2024. [Ηλεκτρονικό]. Available: https://developers.google.com/optimization/scheduling/job_shop.

[47] V. Beraudier , «IBM® Decision Optimization Modeling for Python (DOcplex),» IBM Decision Optimization, 2022. [Ηλεκτρονικό]. Available: https://github.com/IBMDecisionOptimization/docplex-examples/blob/master/examples/cp/visu/flow_shop_permutation.py.

[48] E. Taillard, «Benchmarks for basic scheduling problems,» *European journal of operational research 64.2: 278-285,* 1993.

Εικόνες

Πίνακες

Κώδικες

ΚΩΔΙΚΕΣ

```
import loadfile as lf
import nehedd_new as nhd_n
import calcShedule as cS
import ils
import rsls
import rsls_II as rs2
import ls_insertion_job as ls_ij
import ls_move_job as ls_mv
import ls_exchange_job as ls_xc
import ig
import ga_ls as ga
import copy
import os
import csv
import time

from rich.console import Console
from rich.table import Table
from rich import print
from rich import box
from rich.progress import Progress

arxeia = lf.load_files()
fileCnt = len(arxeia)

#rich table title
table = Table(title="DPFSP ALGORITHM ARENA", style="bold")
table.add_column("FILE", justify="center", style="cyan", no_wrap=True)
table.add_column("FACTORIES", justify="center", style="gold1", no_wrap=True)
table.add_column("JOBS", justify="center", style="gold1")
table.add_column("MACHINES", justify="center", style="gold1")
table.add_column("NEHEDD", justify="right", style="blue")
table.add_column("ILS", justify="right", style="blue")
table.add_column("RSLS", justify="right", style="blue")
table.add_column("RSLS II", justify="right", style="blue")
table.add_column("LS insert", justify="right", style="blue")
table.add_column("LS move", justify="right", style="blue")
table.add_column("LS exchange", justify="right", style="blue")
table.add_column("HYLG", justify="right", style="medium_spring_green")
table.add_column("GA_LS", justify="right", style="medium_spring_green")
table.add_column("Best Result", justify="right", style="red", no_wrap=False)
table.add_column("RPD", justify="right", style="bright_green", no_wrap=False)

#ΦΟΡΤΩΣΗ DATASET
with Progress() as progress:
    print()
    #Έναρξη εκτέλεσης progressbar
    task = progress.add_task("[cyan]ΕΚΤΕΛΕΣΗ ΑΛΓΟΡΙΘΜΩΝ για DPSFSP...", total=fileCnt)

    #Εκτελουμε τους αλγόριθμους για κάθε αρχείο
    start_time = time.time()
```

```
    for i in range(fileCnt):

      file = arxeia[i+1]
      filename = os.path.basename(arxeia[i+1])

      print("fileName =", filename)

      if filename.startswith("I_"):
        csv_file = "Best_Result_small.csv"
      elif filename.startswith("Ta"):
        csv_file = "Best_Result_large.csv"
      else:
        print(f"[red]Το αρχείο {filename} δεν είναι έγκυρο για αυτή τη διαδικασία![/red]")
        continue

      with open(csv_file, mode='r', newline='', encoding='utf-8') as csv_file_obj:
        reader = csv.DictReader(csv_file_obj)
        for row in reader:
          if row['Instance'] == filename:
            best_value = row['Best']
            break

      n,m,F,p,d = lf.read_dpfsp_dataset(arxeia[i+1])
      startSequence = {}


      startNEHedd = nhd_n.nehedd(d,n,m,p,F)
      bestTT = cS.calcTT(d,n,m,p,startNEHedd)
      bestNEHedd = bestTT
      startSequence = copy.deepcopy(startNEHedd)
      bestTT = float("inf")
      bestTTnewI = float("inf")
      bestTT = cS.calcTT(d,n,m,p,startSequence)
      if bestTT < bestTTnewI:
            bestTTnewI = bestTT
            bestSequence = copy.deepcopy(startSequence)
      for i in range(10000):
            bestTT, bestSeq2 = ils.ils(d,n,m,p, startSequence, bestTT)
            if bestTT < bestTTnewI:
                bestTTnewI = bestTT
                bestSequence = copy.deepcopy(startSequence)
      bestILS = bestTT

      startSequenceRSLS = copy.deepcopy(startNEHedd)
      bestTTRSLS = float("inf")
      bestTTnewIRSLS = float("inf")

        for i in range(10000):
            print("RSLS :", i)
            bestTTRSLS, startSequenceRSLS = rsls.rsls(d,n,m,p,startSequenceRSLS, bestTTRSLS)
            if bestTTRSLS < bestTTnewIRSLS:
                bestTTnewIRSLS = bestTTRSLS
                bestSequenceRSLS = copy.deepcopy(startSequenceRSLS)
```

```
        bestRSLS = bestTTRSLS
        startSequenceRSLSII = copy.deepcopy(startNEHedd)
        bestTT = float("inf")
        bestTTRSLSII = float("inf")
        bestTTnewIRSLSII = float("inf")
        bestTTRSLSII = cS.calcTT(d,n,m,p,startSequenceRSLSII)
        if bestTTRSLSII < bestTTnewIRSLSII:
                bestTTnewIRSLSII = bestTTRSLSII
                bestSequenceRSLSII = copy.deepcopy(startSequenceRSLSII)

        for i in range(10000):
            print("RSLS II :", i)
            checkSeqRS_II = copy.deepcopy(startNEHedd)
            bestTTRSLSII, startSequenceRSLSII = rs2.rsls_II(d,n,m,p,checkSeqRS_II, bestTTRSLSII)
            if bestTTRSLSII < bestTTnewIRSLSII:
                bestTTnewIRSLSII = bestTTRSLSII
                bestSequenceRSLSII = copy.deepcopy(startSequenceRSLSII)
        bestRSLS_II = bestTTRSLSII

       startSequenceLS_INSERTION_JOB = copy.deepcopy(startNEHedd)
       sequence_InsertionJob = {}
       sequence_BestTime = {}
       bestTT = float("inf")
       for i in range(10):
            print("LS in :", i)
            sequence_InsertionJob,
       bestTTnew=copy.deepcopy(ls_ij.ls_insertion_job(d,n,m,p,startSequenceLS_INSERTION_JOB))
            if bestTTnew < bestTT:
                bestTT = bestTTnew
                sequence_BestTime = copy.deepcopy(sequence_InsertionJob)
        bestLSinsert = bestTT
        startSequenceLS_MOVE_JOB = copy.deepcopy(startNEHedd)
        sequence_moveJob = {}
        sequence_BestTime_move = {}
        bestTTmove = float("inf")
        for i in range(10):
          bestTTnewLSmove, startSequenceLSmove =
                                          ls_mv.ls_move_job(d,n,m,p,F,startSequenceLS_MOVE_JOB)
            if bestTTnewLSmove < bestTTmove:
                bestTTmove = bestTTnewLSmove
                bestSequenceLSmove = copy.deepcopy(startSequenceLSmove)
        bestLSmove = bestTTmove

        startSequenceLS_EXCHANGE_JOB = copy.deepcopy(startNEHedd)
        sequence_exchangeJob = {}
        sequence_BestTime_exchange = {}
        bestTTexchange = float("inf")

        sequence_exchangeJob,
       bestTTexchangeNew=copy.deepcopy(ls_xc.ls_exchange_job(d,n,m,p,F,startSequenceLS_EXCHANGE_JOB
       ))
        for i in range(10):
            print("LS xc :", i)
```

```
            bestTTnewLSexchange, startSequenceLSexchange =
                                ls_xc.ls_exchange_job(d,n,m,p,F,startSequenceLS_EXCHANGE_JOB)
            if bestTTnewLSexchange < bestTTexchange:
                bestTTexchange = bestTTnewLSexchange
                bestSequenceLSexchange = copy.deepcopy(startSequenceLSexchange)
        bestLSexchange = bestTTexchange

       startSeq_IG = copy.deepcopy(startNEHedd)
        sequence_IG = {}
        bestTTIG = float("inf")
        bestTTnewIG = float("inf")

        startSeq_IG, bestTTIG = ig.ig(d,n,m,p,F,startSeq_IG)

       startSeq_IG = copy.deepcopy(startNEHedd)
        sequence_IG = {}
        bestTTnewIG = float("inf")
        bestTTGA = None

        startSeq_GA, bestTTGA = ga.ga(d,n,m,p,F,startSeq_IG)

        RPD = 0
        if float(best_value) != 0:
            RPD = 0
            RPD_NEHEDD = 0
            RPD_ILS = 0
            RPD_RSLS = 0
            RPD_RSLS_II = 0
            RPD_LS_IN = 0
            RPD_LS_MV = 0
            RPD_LS_EX = 0
            RPD_HYLG = 0
            RPD_GA_LS = 0

            min_value = min(bestNEHedd, bestILS, bestRSLS, bestRSLS_II, bestLSinsert,
              bestLSmove, bestLSexchange, bestTTIG, bestTTGA)
            RPD = (float(min_value) - float(best_value)) / float(best_value) * 100
            RPD = round(RPD,2)

        table.add_row(filename,str(F), str(n), str(m), str(bestNEHedd), str(bestILS),
              str(bestRSLS), str(bestRSLS_II), str(bestLSinsert), str(bestLSmove),
              str(bestLSexchange), str(bestTTIG), str(bestTTGA), str(best_value), str(RPD))
        progress.update(task, advance=1)
    end_time = time.time()
    execution_time = end_time - start_time
console = Console()
print()
console.print(table)
console.print(f"ΣΥΝΟΛΙΚΟΣ ΧΡΟΝΟΣ ΕΚΤΕΛΕΣΗΣ {execution_time:.2f} δευτερόλεπτα.")
```

Κώδικας 15 main function

```
import calcShedule as cS
import ils
import rsls
import rsls_II as rs2
import ls_insertion_job as ls_ij
import ls_move_job as ls_mv
import ls_exchange_job as ls_xc

import copy
import os
import csv
import time
import math
import random

def ig(d,n,m,p,Factories,startSeqls):

    max_iterations = 1000
    djobs = 0.4
    T = 0.8
    bestTT = float("inf")
    minTT = float("inf")
    removed_jobs = []
    pi = copy.deepcopy(startSeqls)
    bestTT = cS.calcTT(d,n,m,p,pi)
    bestTT, pi = rsls.rsls(d,n,m,p,pi, bestTT)
    bestTT = cS.calcTT(d,n,m,p,pi)
    pib = copy.deepcopy(pi)

    for iteration in range(max_iterations):
        removed_jobs = []
        pi_prime = pi.copy()
        print("iteration:", iteration)
        # Remove one job at a time
        range1 = djobs * n/Factories

        for _ in range(int(range1)):
            random_factory = random.choice(list(pi_prime.keys()))
            if len(pi_prime[random_factory]) > 0:
              removed_jobs.append(pi_prime[random_factory].pop(random.randint(0,
                     len(pi_prime[random_factory]) - 1)))

        for job in removed_jobs:
            best_factory, best_position, min_tt = None, None, float('inf')
            for factory in range(Factories):
                for position in range(len(pi_prime[factory]) + 1):
                    test_schedule = copy.deepcopy(pi_prime)
                    test_schedule[factory].insert(position, job)
                    tt = cS.calcTT(d,n,m,p,test_schedule)
                    if tt < min_tt:
                        best_factory, best_position, min_tt = factory, position, tt
            pi_prime[best_factory].insert(best_position, job)
```

```
        # Local search
        bestTTIls2, pi_prime = ils.ils(d,n,m,p, pi_prime, bestTT)
        ttpi_prime = cS.calcTT(d,n,m,p,pi_prime)
        ttpi = cS.calcTT(d,n,m,p,pi)
        ttpib = cS.calcTT(d,n,m,p,pib)

        if ttpi_prime < ttpi:
            pi = copy.deepcopy(pi_prime)
            if ttpi < ttpib:
                pib = copy.deepcopy(pi_prime)
        elif random.random() <= math.exp(-(ttpi_prime - ttpi) / T):
            pi = copy.deepcopy(pi_prime)
    ttpib = cS.calcTT(d,n,m,p,pib)
    return pib, ttpib
```

Κώδικας 16 HYLG Algorithm

```
import calcShedule as cS
import utils
import rsls
import rsls_II as rs2
import ls_insertion_job as ls_ij
import ls_move_job as ls_mv
import ls_exchange_job as ls_xc
import nehedd_new as neh

import copy
import random as rand
import os
import csv
import time
import math

import random

def ga(d,n,m,p,Factories,startSeqls):
    workingSeq_GA = copy.deepcopy(startSeqls)

    #GA PARAMETERS
    population_Size = 50
    generation = 0
    generations = 100
    bestTTGARet = float("inf")
    bestSolution = {}
    child1 = {}
    child2 = {}
    R1 = {}
    R2 = {}
    L1 = {}
```

```
    L2 = {}

    #Create Population
    population = create_population(population_Size, Factories, n)
    population_tt = {}
    for pop in range(len(population
        tt = cS.calcTT(d,n,m,p,population[pop])
        population_tt[pop] = tt
    population[19] = workingSeq_GA

    for generation in range(generations):
        sorted_data = sorted(population_tt.items(), key=lambda item: item[1])
        first_key, first_value = sorted_data[0]
        workingSeq_GA = copy.deepcopy(population[first_key])
        workingSeq_GARet, bestTTGARet = localSearch(d,n,m,p,Factories, workingSeq_GA)
        population[first_key] = copy.deepcopy(workingSeq_GARet)
        selectPop = rand.randint(1, population_Size-1)
        workingSeq_GA = copy.deepcopy(population[selectPop])
        workingSeq_GAInv, bestTTGAInv = localSearch(d,n,m,p,Factories, workingSeq_GA)
        population[selectPop] = copy.deepcopy(workingSeq_GAInv)

        #Sort population after Local Search
        for pop in range(len(population)):
            tt = cS.calcTT(d,n,m,p,population[pop])
            population_tt[pop] = tt
        sorted_data = sorted(population_tt.items(), key=lambda item: item[1])

        #Crossover
        # Select Parent 1 the best Individual
        first_key, first_value = sorted_data[0]
        parent1 = copy.deepcopy(population[first_key])
        # Select Parent 2 random Individual
        selectPop = rand.randint(1, population_Size-1)
        parent2 = copy.deepcopy(population[selectPop])
        if len(parent1) == len(parent2):
            for facts in range(len(parent1)):
                cutPosition = int(len(parent1[facts])/2)
                R1[facts] = parent1[facts][:cutPosition]
                L1[facts] = parent1[facts][cutPosition:]
                cutPosition = int(len(parent2[facts])/2)
                R2[facts] = parent2[facts][:cutPosition]
                L2[facts] = parent2[facts][cutPosition:]
            child1 = R1
            child2 = R2

        if len(L1) > 0:
            for ins in range(len(L1)):
                for insItem in range(len(L1[ins])):
                    check1 = {}
                    fullCheck = {}
                    bestTTfact = float("inf")
                    for fact in range(len(child1)):
                        jobstoCH = len(child1[fact])
```

```
                        for g in range(0, jobstoCH+1):
                            check1 = copy.deepcopy(child1)
                            tmp_seqlsGA = utils.insertion(child1[fact], g, L1[ins][insItem])
                            check1[fact] = copy.deepcopy(tmp_seqlsGA)
                            timiTTfact = cS.calcTT(d,n,m,p,check1)
                            if(timiTTfact<bestTTfact):
                                bestTTfact = timiTTfact
                                fullCheck = copy.deepcopy(check1)
                    child1 = copy.deepcopy(fullCheck)
                population[-1] = child1

        if len(L2) > 0:
            for ins in range(len(L2)):
                for insItem in range(len(L2[ins])):
                    check1 = {}
                    fullCheck = {}
                    bestTTfact = float("inf")
                    for fact in range(len(child2)):
                        jobstoCH = len(child2[fact])

                        for g in range(0, jobstoCH+1):
                            check2 = copy.deepcopy(child2)
                            tmp_seqlsGA = utils.insertion(child2[fact], g, L2[ins][insItem])
                            check2[fact] = copy.deepcopy(tmp_seqlsGA)
                            timiTTfact = cS.calcTT(d,n,m,p,check2)
                            if(timiTTfact<bestTTfact):
                                bestTTfact = timiTTfact
                                fullCheck = copy.deepcopy(check2)
                    child2 = copy.deepcopy(fullCheck)
                population[-2] = child2

    # Return the best Solution
    first_key, first_value = sorted_data[0]
    workingSeq_GARet = copy.deepcopy(population[first_key])
    bestTTGARet = cS.calcTT(d,n,m,p, workingSeq_GARet)
    return workingSeq_GARet, bestTTGARet


def create_population(pop_size, num_factories, num_jobs):
    return [create_random_individual(num_factories, num_jobs) for _ in range(pop_size)]

def create_random_individual(num_factories, num_jobs):
    jobs = list(range(num_jobs))  # Jobs ξεκινούν από το 0
    random.shuffle(jobs)
    factories = {i: [] for i in range(num_factories)}
    for i, job in enumerate(jobs):
        factory_key = i % num_factories
        factories[factory_key].append(job)
    return factories

def localSearch(d,n,m,p,Factories, workingSeq_GA):
    bestSequenseLS = copy.deepcopy(workingSeq_GA)
```

```
    ttBestLS = cS.calcTT(d,n,m,p,workingSeq_GA)

    for rounds in range(100):
        flag = True
        for factory in range(Factories):
            workingSeq_GA = ls_ij.insertion_Job_OneFact(d,n,m,p,workingSeq_GA,factory)
        count = 0
        while flag:
            count = count+1
            workingSeq_GA_check = copy.deepcopy(workingSeq_GA)
            bestW, workingSeq_GA = ls_mv.ls_move_job(d,n,m,p,Factories,workingSeq_GA)
            bestW, workingSeq_GA = ls_xc.ls_exchange_job(d,n,m,p,Factories,workingSeq_GA)
            ttMid = cS.calcTT(d,n,m,p,workingSeq_GA)
            for factory in range(Factories):
                if (workingSeq_GA_check[factory] == workingSeq_GA[factory]) or count > 20:
                   flag = False
                else:
                    workingSeq_GA = ls_ij.insertion_Job_OneFact(d,n,m,p,workingSeq_GA,factory)

        bestTTGA = cS.calcTT(d,n,m,p,workingSeq_GA)
        if(bestTTGA < ttBestLS):
            bestSolution = copy.deepcopy(workingSeq_GA)
            workingSeq_GARet = copy.deepcopy(workingSeq_GA)
            ttBestOveraAll = bestTTGA
            bestTTGARet = bestTTGA
            bestSequenseOverAll = copy.deepcopy(workingSeq_GA)
            ttBestOveraAll = cS.calcTT(d,n,m,p,workingSeq_GA)
        else:
            bestSolution = copy.deepcopy(bestSequenseLS)
            bestTTGARet = ttBestLS

    return bestSolution, bestTTGARet
```

Κώδικας 17 Genetic Algorithm with Local Search