Εικόνα που περιέχει κείμενο, γραμματοσειρά, λογότυπο, γραφικά

Περιγραφή που δημιουργήθηκε αυτόματα

Διπλωματική Εργασία

**«Το πρόβλημα χρονοπρογραμματισμού**

**κατανεμημένων ροών εργασιών βάσει**

**μεταθέσεων με περιορισμούς ημερομηνιών**

**τερματισμού εργασιών»**

**«Distributed Permutation Flow-shop**

**Scheduling Problem with Due Dates»**

ΚΙΟΣΣΕΣ ΔΗΜΗΤΡΙΟΣ

ΑΜ:163

Επιβλέπων καθηγητής: Γκόγκος Χρήστος

**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 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 παράξει ολοκληρωμένες λύσεις. Θα χρησιμοποιηθούν βιβλιοθήκες καθώς και εξειδικευμένοι επιλυτές (π.χ. επιλυτής προγραμματισμού με περιορισμούς κ.α.) έτσι ώστε να επιτευχθούν λύσεις υψηλής ποιότητας για δημόσια διαθέσιμα προβλήματα.

Περιεχόμενα

[1. ΕΙΣΑΓΩΓΗ 5](#_Toc180269443)

[2. ΙΣΤΟΡΙΚΗ ΑΝΑΦΟΡΑ 6](#_Toc180269444)

[3. ΠΕΡΙΓΡΑΦΗ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ 8](#_Toc180269445)

[3.1 ΚΑΤΗΓΟΡΙΕΣ ΧΡΟΝΟΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ 8](#_Toc180269446)

[3.1.1 JOB SHOP PROBLEM 8](#_Toc180269447)

[3.1.2 FLOW SHOP PROBLEM 9](#_Toc180269448)

[3.2 PFSP 10](#_Toc180269449)

[3.3 DPFSP 12](#_Toc180269450)

[4. ΠΕΡΙΓΡΑΦΗ ΤΟΥ ΜΟΝΤΕΛΟΥ 14](#_Toc180269451)

[4.1 ΠΕΡΙΟΡΙΣΜΟΙ ΚΑΙ ΣΗΜΑΝΣΕΙΣ 14](#_Toc180269452)

[4.1.1 Δείκτες 14](#_Toc180269453)

[4.1.2 Παράμετροι 14](#_Toc180269454)

[4.1.3 Μεταβλητή απόφασης 14](#_Toc180269455)

[4.2 MILP 15](#_Toc180269456)

[4.3 Αριθμητική απεικόνιση 16](#_Toc180269457)

[5. ΤΡΟΠΟΙ ΑΝΤΙΜΕΤΩΠΙΣΗΣ ΠΡΟΒΛΗΜΑΤΟΣ 18](#_Toc180269458)

[5.1 Ευρετικές Μέθοδοι 18](#_Toc180269459)

[5.1.1 Ο ΑΛΓΟΡΙΘΜΟΣ NEH 18](#_Toc180269460)

[5.1.3 Ο ΑΛΓΟΡΙΘΜΟΣ TABOO Search 27](#_Toc180269461)

[5.1.2 Ο ΑΛΓΟΡΙΘΜΟΣ NEHedd 29](#_Toc180269462)

[6. ΑΝΑΠΤΥΞΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ 31](#_Toc180269463)

[6.1 ITERATED LOCAL SEARCH (ILS) 32](#_Toc180269464)

[6.2 ITERATED GREEDY ALGORITHM (IG) 34](#_Toc180269465)

[6.2.1 RANDOM SUBSEQUENCE LOCAL SEARCH 36](#_Toc180269466)

[6.2.2 RANDOM SINGLE POINT LOCAL SEARCH 37](#_Toc180269467)

[6.3 MLL BASED MECHANISM 40](#_Toc180269468)

[6.4 ITERATED GENETIC ALGORITHM 41](#_Toc180269469)

[6.5 HYBRID GENETIC ALGORITHM 41](#_Toc180269470)

[6.6 ΣΧΕΔΙΑΣΜΌΣ ΠΕΙΡΑΜΑΤΩΝ 41](#_Toc180269471)

[7 ΑΞΙΟΛΟΓΗΣΗ ΚΑΙ ΑΝΑΛΥΣΗ ΑΠΟΤΕΛΕΣΜΑΤΩΝ 42](#_Toc180269472)

[7.1 ΠΑΡΟΥΣΙΑΣΗ ΑΠΟΤΕΛΕΣΜΑΤΩΝ 42](#_Toc180269473)

[7.1.2 ΑΝΑΛΥΣΗ ΑΠΟΤΕΛΕΣΜΑΤΩΝ 42](#_Toc180269474)

[7.1.3 ΣΥΓΚΡΙΣΗ ΚΑΙ ΣΥΜΠΕΡΑΣΜΑΤΑ 42](#_Toc180269475)

[8 ΣΥΜΠΕΡΑΣΜΑΤΑ ΚΑΙ ΜΕΛΛΟΝΤΙΚΕΣ ΠΡΟΚΛΗΣΕΙΣ 43](#_Toc180269476)

[Βιβλιογραφία 44](#_Toc180269477)

# ΕΙΣΑΓΩΓΗ

Το Distributed Permutation Flow Shop Problem (DPFSP) είναι ένα κλασικό πρόβλημα βελτιστοποίησης που ανήκει στην κατηγορία των προβλημάτων προγραμματισμού παραγωγής. Συγκεκριμένα, το DPFSP αφορά την εύρεση της βέλτιστης σειράς εκτέλεσης μιας ομάδας εργασιών σε πολλαπλές μηχανές ή σταθμούς εργασίας, 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 προγραμματισμού παραγωγής, 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 χρονοπρογραμματισμού κατανεμημένης μετατροπής ροών (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 αλγορίθμους 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. Εδώ θα περιγράψουμε αναλυτικότερα τα προβλήματα PFSP και DPFSP στα οποία και επικεντρώνεται και η εργασία από την πλευρά των χρόνων καθυστέρησης, σύμφωνα πάντα με τους χρόνους due dates της κάθε εργασίας.

**….**

# ΙΣΤΟΡΙΚΗ ΑΝΑΦΟΡΑ

Η έννοια του χρονοπρογραμματισμού δεν είναι καινούργια, οι πυραμίδες κατασκευάστηκαν πριν από 3000 έτη, o 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* τη δεκαετία του 1950, οι οποίοι εισήγαγαν αλγορίθμους για απλούστερα συστήματα δύο σταδίων [1].

**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 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 εγκατάστασης είναι ανεξάρτητοι από την ακολουθία και περιλαμβάνονται στους χρόνους επεξεργασίας ή αγνοούνται.
* Επιτρέπεται άπειρη αποθήκευση εντός της διαδικασίας.

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 αλγορίθμους 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 την σημειολογία τριών σημείων του Graham. To 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) 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 μεγαλύτερη ακρίβεια και ταχύτητα. Ταυτόχρονα, αυξήθηκε το ενδιαφέρον για την εφαρμογή αυτών των μεθόδων σε πραγματικά σενάρια παραγωγής σε βιομηχανίες, από την αυτοκινητοβιομηχανία μέχρι την κατασκευή ηλεκτρονικών. Όσον αφορά τις μεταευρετικές μεθόδους, υπάρχει επίσης μια τεράστια βιβλιογραφία με διαφορετικές προτάσεις για το PFSP με διαφορετικά κριτήρια. Αξιοσημείωτες μέθοδοι αναζήτησης Tabu Search είναι αυτές των Nowicki και Smutnicki (1996) ή των Grabowski και Wodecki (2004). Οι αλγόριθμοι προσομοιωμένης ανόπτησης προτείνονται από τους Osman and Potts (1989), ενώ οι γενετικοί αλγόριθμοι παρουσιάζονται από τους Reeves (1995) και τους Ruiz, Maroto and Alcaraz (2006). Άλλες μεταευρετικές μέθοδοι όπως η Ant Colony Optiomization, η Scatter Search, η Discrete Differential Evolution, η Particle Swarm Optimization ή η Iterated Greedy παρουσιάζονται στους Rajendran and Ziegler (2004), Nowicki and Smutnicki (2005), Onwubolu and Davendra (2006), Tasgetiren et al. (2007) και Ruiz and Stützle (2007), αντίστοιχα. Πρόσφατες και υψηλής απόδοσης προσεγγίσεις περιλαμβάνουν μεθοδολογίες παράλληλου υπολογισμού, όπως αυτή που παρουσιάζεται στους Vallada and Ruiz (2009).

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.

# ΠΕΡΙΓΡΑΦΗ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ

Ο προγραμματισμός του χρόνου είναι μια φυσική διαδικασία που οι άνθρωποι κάνουν καθημερινά. Ο προγραμματισμός όμως της καθημερινής μας ζωής είναι ως επί το πλείστων 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. [2]

Στην επιστημονική θεωρία, ο χρονοπρογραμματισμός τυποποιείται στο έργο της κατανομής πεπερασμένων πόρων κατά τη διάρκεια του χρόνου για την εκπλήρωση ενός δεδομένου συνόλου εργασιών [3]. Η δομή ενός προβλήματος χρονοπρογραμματισμού από επιστημονική εξέταση, 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 σε έναν 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 flowhop 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 ΚΑΤΗΓΟΡΙΕΣ ΧΡΟΝΟΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Στα παραδοσιακά προβλήματα χρονοπρογραμματισμού συναντάμε τις παρακάτω κύριες κατηγορίες:

3.1.1 JOB SHOP PROBLEM (**JSP** (n!)m πιθανές λύσεις): Το JSP αποτελεί μια κλασική πρόκληση συνδυαστικής βελτιστοποίησης στον προγραμματισμό και τον προγραμματισμό παραγωγής. Χρησιμεύει ως μια θεμελιώδης αφαίρεση των εργασιών προγραμματισμού στην Παραγωγή και σε περιβάλλοντα υπηρεσιών, όπου ένας πεπερασμένος αριθμός εργασιών, καθεμία από τις οποίες περιλαμβάνει μια ακολουθία εργασιών (που αναφέρονται επίσης ως λειτουργίες), πρέπει να εκτελεστούν σε ένα σύνολο μηχανών. 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[3]

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***. *Η FSP παραλληλίζεται στενά με την JSP όσον αφορά την εννοιολόγησή της. Στο πλαίσιο του FSP, ένα σύνολο εργασιών υπόκειται σε επεξεργασία σε μια ακολουθία μηχανών, καθεμία με την ίδια προκαθορισμένη σειρά εργασιών. Αυτός ο διαδοχικός περιορισμός διαφοροποιεί την FSP, η οποία διατηρεί μια συνεπή σειρά εργασιών σε όλες τις εργασίες και από το JSP, όπου οι ακολουθίες εργασιών μπορούν να διαφέρουν μεταξύ μηχανές. Από την άλλη πλευρά, στο FSP, η σειρά των εργασιών που εκτελούνται σε κάθε μηχανή μπορεί να να είναι διαφορετική.*[3]

**3.1.3 PERMUTATION FLOW SHOP PROBLEM** (**PFSP** n! πιθανές λύσεις): *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.*[3]

**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 Pji f τον χρόνο επεξεργασίας της εργασίας j στο μηχανή i του εργοστασίου f, ισχύει ότι Pji0 = Pji1 = Pji2. Ο τυπικός στόχος του DPFSP είναι η ελαχιστοποίηση του χρονικού διαστήματος επεξεργασίας (makespan) ή του ελάχιστου χρόνου καθυστέρησης σχετικά με του αναμενόμενου χρόνου τερματισμού (due date / tardiness).*[3]

## 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 tij 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 την Βελτιστοποίηση της παραγωγικότητας, την εξοικονόμηση του κόστους, την επιτάχυνση παράδοσης και την βελτίωση της ποιότητας ενός προϊόντος.

**Τύποι προβλημάτων PFSP**

* ***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 ή το συνολικό χρόνο ολοκλήρωσης, ενώ μόνο μία αναφορά ασχολείται με τη συνολική καθυστέρηση.[4]
* ***Τα προβλήματα DPFS με περιορισμούς*** είναι προβλήματα ενός στόχου που περιλαμβάνουν τουλάχιστον έναν πρόσθετο περιορισμό. Οι πιο συνηθισμένοι περιορισμοί είναι (μικτοί) αποκλεισμός, μη αναμονή, (μικτοί) μη αδράνεια και χρόνοι εγκατάστασης. Ο πιο μελετημένος στόχος είναι το makespan, αν και ο συνολικός χρόνος ολοκλήρωσης έχει εξεταστεί για το μπλοκάρισμα και το no-idle, και η συνολική ταλαιπωρία για το μπλοκάρισμα. Επιπλέον, ορισμένες συνεισφορές εξετάζουν πρόσθετους περιορισμούς, με πιο εκτεταμένους εκείνους που σχετίζονται με την προληπτική συντήρηση.[4]
* ***Τα πολυαντικειμενικά (πολυκριτηριακά) προβλήματα DPFS*** ταξινομούνται ανάλογα με την πολυαντικειμενική προσέγγιση που υιοθετείται, δηλαδή: Pareto ή Γραμμικός συνδυασμός στόχων. Δεδομένης της υπεροχής των εργασιών που ασχολούνται με ενεργειακά ζητήματα με προσέγγιση Pareto, εξετάστηκε μια συγκεκριμένη κατηγορία. Και πάλι, το makespan είναι ο πιο συχνά χρησιμοποιούμενος στόχος μαζί με άλλα μέτρα κόστους. Η πλειονότητα των αναφορών περιλαμβάνει περιορισμούς που προέρχονται από πραγματικές περιπτώσεις, με την ταχύτητα των μηχανών να είναι ο πιο εκτεταμένος περιορισμός στην προσέγγιση της ενεργειακής απόδοσης[4]
* ***Τα μη ντετερμινιστικά προβλήματα DPFS*** εξετάζουν αναφορές σε ασαφή, στοχαστικά και αβέβαια προβλήματα. Δεν υπάρχουν πολλές εργασίες σε αυτή την κατηγορία και όλες εξετάζουν στόχους που σχετίζονται με το makespan.[4]
* ***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. [4]

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, ορθογώνιο παραλληλόγραμμο, διάγραμμα

Περιγραφή που δημιουργήθηκε αυτόματα

## 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) 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 τον δεδομένο μέτρο απόδοσης.

Οι περισσότερες από τις προηγούμενες εργασίες για το DPFSP αναγνώριζαν μόνο το makespan ως μέτρο απόδοσης. Ωστόσο, στο σημερινό πελατοκεντρικό σύστημα ανταγωνιστική αγορά, η έγκαιρη παράδοση του προϊόντος έχει γίνει πιο κρίσιμη από ποτέ [5]. Η ελαχιστοποίηση της συνολικής καθυστέρησης επιτρέπει στις βιομηχανίες παραγωγής την ολοκλήρωση των παραγγελιών των πελατών πριν από τις ημερομηνίες λήξης τους (αποτρέποντας είτε την απώλεια φήμης, είτε ακόμη και την απώλεια πελάτη). Ωστόσο, παρά το γεγονός ότι πρόκειται για ένα ρεαλιστικό μέτρο απόδοσης, η ελαχιστοποίηση της συνολικής καθυστέρησης (TT) για το DPFSP δεν έχει 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 DF|prmu|Tj. [1] Το εξεταζόμενο πρόβλημα (DF|prmu|Tj) είναι ένα NP-δύσκολο πρόβλημα καθώς αποτελεί επέκταση του κανονικού PFSP με συνολική καθυστέρηση κριτήριο (F|prmu|)

**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.’Εχουν γίνει τέσσερις 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). Ανέπτυξαν εποικοδομητικές ευρετικές τεχνικές. Αργότερα, ο Kim (1993) πρότεινε μια ευρετική που βασιζόταν στη διαδικασία αναζήτησης tabu των Widmer και Hertz (1989). Οι Parthasarathy και Rajendran (1998) 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). [6]

# ΠΕΡΙΓΡΑΦΗ ΤΟΥ ΜΟΝΤΕΛΟΥ

## 4.1 ΠΕΡΙΟΡΙΣΜΟΙ ΚΑΙ ΣΗΜΑΝΣΕΙΣ

Το πρόβλημα αφορά περισσότερα από ένα πανομοιότυπα εργοστάσια, δηλ. f > 1, που βρίσκονται σε διαφορετικές περιοχές, με το την ευθύνη της επεξεργασίας ενός συνόλου n εργασιών.

* Όλα τα εργοστάσια έχουν το ίδιο σύνολο m μηχανών στην σταθερή μετάθεση.
* 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.[7]

### 4.1.1 Δείκτες

* **j, k** Δείκτης για θέσεις εργασίας και θέσεις εργασίας- 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 [7]

### 4.1.2 Παράμετροι

* **n** Αριθμός εργασιών
* **m** Αριθμός μηχανών
* **F** Αριθμός εργοστασίων
* **pi,j** Χρόνος επεξεργασίας της εργασίας j στη μηχανή i
* **dj**Ημερομηνία λήξης της εργασίας

### 4.1.3 Μεταβλητή απόφασης

* **Xj,k** 1 εάν η θέση εργασίας j ανατίθεται στη θέση k- 0 διαφορετικά
* **Yk,f** 1 εάν η θέση εργασίας k ανατίθεται στο εργοστάσιο f - διαφορετικά 0
* **Ck,i** Χρόνος ολοκλήρωσης της εργασίας στη θέση k στο μηχάνημα i
* **Uk** Ημερομηνία λήξης της εργασίας στη θέση k
* **Tk** Καθυστέρηση της εργασίας j στη θέση k
* **TT** Συνολική καθυστέρηση

|  |  |  |
| --- | --- | --- |
|  | (1) | Καθορίζει την συνάρτηση για την ελαχιστοποίηση της συνολικής καθυστέρησης TT |
|  | (2)  (3) | Μαζί εξασφαλίζουν ότι κάθε εργασία πρέπει να κατανέμεται σε ακριβώς μία θέση και κάθε θέση πρέπει να ανατίθεται ακριβώς σε μία εργασία. |
|  | (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. |
|  | (6) | Εξασφαλίζει ότι η έναρξη εργασίας σε μια συγκεκριμένη μηχανή μπορεί να ξεκινήσει αφού έχει τελειώσει η επεξεργασία της προηγούμενης εργασίας στην ίδια μηχανή. |
|  | (7) | Βρίσκει την ημερομηνία λήξης σύμφωνα με την θέση της εργασίας. |
|  | (8) | Υπολογίζει την καθυστέρηση μιας εργασίας στην θέση k. |
|  | (9)  (10)  (11)  (12) | Ορίζουν τις μεταβλητές απόφασης. |

## 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.[8]

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.

Κύριες Προκλήσεις

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.

Παρουσιάζουμε ένα μοντέλο μεικτού ακέραιου γραμμικού προγραμματισμού (MILP) για την περιγραφή του εξεταζόμενου προβλήματος το οποίο μπορούμε να περιγράψουμε με τον παρακάτω συμβολισμό:

Στόχος είναι η κατανομή των εργασιών σε διάφορα εργοστάσια και ο ταυτόχρονος καθορισμός των ακολουθιών επεξεργασίας τους σε κάθε εργοστάσιο για την ελαχιστοποίηση του στόχου της συνολικής καθυστέρησης (TT). [7]

## 4.3 Αριθμητική απεικόνιση

Η αναπαράσταση της λύσης ενός προβλήματος DPFSP, όπως παρουσιάζεται από τους Naderi and Ruiz, γίνεται με ένα σύνολο F λιστών λύσεων. Κάθε λίστα αναπαριστά μια ακολουθία πf, όπου f=[1,2,3,…,F]. H λίστα π={π1, π2, π3, π4, π5, π6, … πf, … πF} αναφέρεται σε F λίστες. Ένα πιθανό παράδειγμα με 12 jobs και 3 εργοστάσια, θα μπορούσαν να είναι οι παρακάτω ακολουθίες π=[{1,5,8,11,12}, {2,4,7,10}, {3,6,9}].

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 = 5, machines = 2, factories = 2. Οι χρόνοι εκτέλεσης και οι ημερομηνίες τερματισμού φαίνονται στον παρακάτω πίνακα:



Θέλουμε να υπολογίσουμε την μικρότερη χρονική καθυστέρηση για την ακολουθία π = {5,4,3,2,1}.

Αρχικά τοποθετούμε την εργασία 5 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 4 στα δύο εργοστάσια. Στο πρώτο εργοστάσιο θα τερματίσει μετά από 20 μονάδες χρόνου (Έναρξη επεξεργασίας στην μηχανή 1 μετά από 3 μονάδες και έναρξη στην μηχανή 2 μετά από 14 μονάδες, συν 6 μονάδες επεξεργασίας μας κάνει 20). Στο εργοστάσιο 2 η εργασία 4 θα ολοκληρωθεί μετά από 15 μονάδες. Επιλέγουμε να τοποθετήσουμε την εργασία 4 στο εργοστάσιο 2. Ακολουθεί η διαχείριση της εργασίας 3. Η εργασία 3 ολοκληρώνεται μετά από 18 μονάδες χρόνου στο εργοστάσιο 1 και μετά από 21 μονάδες στο εργοστάσιο 2. Επιλέγουμε το εργοστάσιο 1. Συνεχίζοντας με την ίδια λογική υπολογίζουμε τις τοποθετήσεις των υπόλοιπων εργασιών για να καταλήξουμε στο παρακάτω διάγραμμα Gant.[7]

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, διάγραμμα, παράλληλα

Περιγραφή που δημιουργήθηκε αυτόματα

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 5 έχει χρόνο εκτέλεσης 5 μονάδων, αλλά ολοκληρώθηκε μετά από 14 μονάδες και ο χρόνος καθυστέρησης T5 είναι 9. Για τις υπόλοιπες εργασίες οι χρόνοι καθυστέρησης είναι T­4 = 5, T3 = 0, T2 = 17 Τ1 = 16. Ο συνολικός χρόνος καθυστέρησης είναι:

# 5. ΤΡΟΠΟΙ ΑΝΤΙΜΕΤΩΠΙΣΗΣ ΠΡΟΒΛΗΜΑΤΟΣ

## 5.1 Ευρετικές Μέθοδοι

Το πρόβλημα του προγραμματισμού ροής αποτελεί έντονο πεδίο έρευνας εδώ και πολλά χρόνια. Καθώς η συντριπτική πλειονότητα των προβλημάτων χρονοπρογραμματισμού flowshop είναι NP-πλήρη. Τα τελευταία χρόνια έχει μειωθεί ο αριθμός των αποτελεσματικών ευρετικών αλγορίθμων για την επίλυση προβλημάτων DPFSP με n εργασίες και m μηχανές. Εξάλλου έχει αποδειχθεί από τον ερευνητική ομάδα του Garey et al ότι τα προβλήματα PFSP για περισσότερες από 3 μηχανές είναι NP-πλήρη.[9]Ως αποτέλεσμα έχουμε την έρευνα να κατευθύνεται κυρίως προς την ανάπτυξη ευρετικών ή σχεδόν βέλτιστων 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), Dannenbring (1977), Nawaz et al. (1983), Widmer and Hertz (1989), Leisten (1990), Ogbu and Smith (1990), Ishibuchi et al. (1995), Rajendran (1995), Nowicki and Smutnicki (1996), Rajendran and Ziegler (1997), Ben-Daya and Al-Fawzan (1998), και Framinan et al. (2001).

## 5.1.1 Ο ΑΛΓΟΡΙΘΜΟΣ NEH

Το 1983 η ομάδα Nawaz, Enscore, Ham παρουσίασαν τον αλγόριθμο NEH, o οποίος θεωρείται ο «άρχοντας» των ευρετικών αλγόριθμων. Ο αλγόριθμος ΝΕΗ 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).[10]

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.

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])              #print("WorkSequense   %s"%tmp\_seq) # Για να εκτυπώσουμε την ακολουθία              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              #print("tmpTime: %i"%tmpTime) # Για να εκτυπώσουμε το makespan για την συγκεκριμένη ακολουθία              #Έαν 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      #print("=====================")      #Επιστρέφει την καλύτερη τελευταία ακολουθία που είναι και η καλύτερη ακολουθία του αλγόριθμου      return bestSequense |

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 παρουσιάζω την 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 (ο παρακάτω πίνακας) 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 | | |  |  | C (1 - 4 - 0 – 3) | | |
| 0 | 54 | 79 | 16 |  | 1 | 83 | 83+3 = 86 | 86 + 89 = 175 |
| 1 | 83 | 3 | 89 |  | 4 | 77 + 83 = 160 | 160 + 56 = 216 | 216 + 89 = 305 |
| 2 | 15 | 11 | 49 |  | 0 | 160+54 = 214 | 216 + 79 = 295 | 305 + 16 = 321 |
| 3 | 71 | 99 | 15 |  | 3 | 214 + 71 = 285 | 295 + 99 = 394 | 394 + 15 = **409** |
| 4 | 77 | 56 | 89 |  |  |  |  |  |

|  |
| --- |
| 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] |

#### 5.1.1.1 ΒΕΛΤΙΩΜΈΝΟΣ ΑΛΓΟΡΙΘΜΟΣ 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. 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 ei,j της εργασίας i στην μηχανή j |
| e0j = 0, ei0 = 0  eij = max { ei, j-1 , ei-1, j } + tij  (i = 1, … , k-1) (j = 1, … , m) |
| **BHMA 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. |
| qkj = 0, qi, m+1 = 0  qij = max { qi, j+1 , ei+1, j } + tij  (i = k-1, … ,1) (j = m, … , 1) |
| ΒΗΜΑ 3ο Υπολογίζει τον συντομότερο χρόνο fij στην j-οστή μηχανή της k-οστής εργασίας που εισάγεται στην i-οστή θέση. |
| fi0 = 0,  fij = max { fi, j-1 , ei-1, j } + tkj  (i = 1, … , k) (j = 1, … , m) |
| ΒΗΜΑ 4ο Καταχωρείται η τιμή του χρόνου MAKESPAN Μi όταν προστεθεί η εργασία k στην i-οστή θέση. |
| Mi = maxj (fij + qij)  (i = 1, … , k) (j = 1, … , m) |

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 παρακάτω πίνακα 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 αντίστροφή 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** |

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

**fi,j = max(fi,j-i , ei-1,j) + pk,j**

Π.χ. για την τοποθέτηση της τιμής p2,1 (11) στην θέση f2,1 συγκρίνουμε τις τιμές f2,0 (140) και e1,1 (232), παίρνουμε την μεγαλύτερη τιμή και αθροίζουμε την τιμή p2,1, άρα στην θέση f2,1 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 δηλώσουμε 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 |

Ο αλγόριθμος NEH έχει πολυπλοκότητα Ο(n3m) 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")  print("-----------------------------------------------------------------------------------------------------------------------") |

Όπως μπορούμε να δούμε στην παρακάτω εικόνα και οι δύο αλγόριθμοι καταλήγουν στην ίδια ακολουθία και στο ίδιο makespan. O αλγόριθμος NEH εκτελέστηκε σε 13,96 ms ενώ ο βελτιωμένος αλγόριθμος NEH\_f σε 3,02 ms.

Εικόνα που περιέχει κείμενο, γραμματοσειρά, γραμμή, στιγμιότυπο οθόνης

Περιγραφή που δημιουργήθηκε αυτόματα

## 5.1.3 Ο ΑΛΓΟΡΙΘΜΟΣ 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 στην 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 επαναλήψεις 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 |

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 ΑΛΓΟΡΙΘΜΟΣ 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, και στη συνέχεια λαμβάνεται μια τελική λύση με εποικοδομητικό τρόπο, προσθέτοντας σε κάθε βήμα μια νέα εργασία σε αυτή τη σειρά και στη συνέχεια εισάγοντας την στην καλύτερη θέση, δηλαδή σε αυτή που οδηγεί στην καλύτερη μερική λύση. Δεδομένου ότι στο πρόβλημά μας λαμβάνονται υπόψη οι ημερομηνίες λήξης, μπορεί να υπάρχουν διάφορες μέθοδοι για την ταξινόμηση των εργασιών. Στο NEHedd, οι εργασίες ταξινομούνται κατά μη φθίνουσα σειρά των ημερομηνιών λήξης. Η EDD (edd) υποδηλώνει την συντομότερη ημερομηνία λήξης. Μια παραλλαγή του αλγόριθμου NEHedd εκτελεί μια αρχική ταξινόμηση των εργασιών σε περισσότερα του ενός εργοστάσια σύμφωνα με τον ελάχιστο χρόνο καθυστέρησης, ο οποίος υπολογίζεται από τους χρόνους due date.[11]

# 6. ΑΝΑΠΤΥΞΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ

Για την αντιμετώπιση δύσκολων προβλημάτων βελτιστοποίησης, χρησιμοποιούνται αλγόριθμοι υψηλής απόδοσης οι οποίοι τις περισσότερες φορές καταλήγουν σε μεταευρετικές μεθόδους. 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 άριστα και να εξερευνούν πιο αποτελεσματικά τον χώρο των πιθανών λύσεων. [12]

Κατά τον σχεδιασμό μια μεταευρετικής μεθόδου πρέπει να προτιμάται η απλότητα, τόσο εννοιολογικά όσο και πρακτικά. Φυσικά, πρέπει επίσης να οδηγεί σε αποτελεσματικούς αλγορίθμους. 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 στην σχετική με το πρόβλημα γνώση, αντιμετωπίζοντας το ως ρουτίνα «μαύρου κουτιού». 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 άριστα.
* **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 και τοπική αναζήτηση για να ανακαλύψει καλύτερες λύσεις. [13]

## 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 άριστα 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 προσέγγιση μηχανικής αλγορίθμων όπου, σταδιακά, η απόδοση του αλγορίθμου μπορεί να βελτιστοποιηθεί περαιτέρω.

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 άριστο).
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 άριστο, και η διαταραχή επιτρέπει την εξερεύνηση νέων περιοχών του χώρου λύσεων.

Εικόνα που περιέχει διάγραμμα, γραμμή, γράφημα, κείμενο

Περιγραφή που δημιουργήθηκε αυτόματα

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:

|  |
| --- |
| **Procedure ILS**  s0 = GenerateInitialSolution //π.χ. NEHedd  s\* = LocalSearch(s0)  repeat  s' = Perturbation(s\*, history)  s\*' = LocalSearch(s')  s\* = AcceptanceCriterion(s\*, s\*', history)  until termination condition metend |

Αρχικώς χρησιμοποιούμε έναν άπληστο αλγόριθμο για να καταλήξουμε στην αρχική λύση του προβλήματος. Για τα προβλήματα χρονοπρογραμματισμού (DPFSP) ο αλγόριθμος ΝΕΗ φαίνεται να είναι η ευρετική κατασκευή με τα 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. [14]

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

Perturbation: Για την διαταραχή (Perturbation) 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. [14] 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 [15] και [10] 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. [16]

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 [16]. 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 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 ITERATED GREEDY ALGORITHM (IG)

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. Εξ όσων γνωρίζουμε, αυτός ο αλγόριθμος είναι η πρώτη εφαρμογή του αλγορίθμου IG για την ελαχιστοποίηση της συνολικής καθυστέρησης του DPFSP.[17]

Ο Iterated Greedy Algorithm 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 Iterated Greedy 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, καταστρέφουμε ένα μέρος της λύσης και κατόπιν το ξαναχτίζουμε (rebuild) με άπληστο τρόπο. Η διαδικασία αυτή επιτρέπει στον αλγόριθμο να εξερευνά διαφορετικά μονοπάτια, αποφεύγοντας τοπικά ελάχιστα, και προσπαθεί να βρει μια καλύτερη λύση από την αρχική.

Ο αλγόριθμος IG εκτελεί τις παρακάτω λειτουργίες:

**1.Αρχικοποίηση**: Ξεκινάμε 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 εξετάζονται οι αλγόριθμοι NEHedd και ESL, όπως παρουσιάστηκαν 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 αφαιρέθηκε ξαναενσωματώνεται 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 πR. Στην φάση της κατασκευής κάθε εργασία από την λίστα pR 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 παγκόσμια βέλτιστη λύση.

Για την ενίσχυση της απόδοσης του αλγόριθμου IG 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. [7]

Στο RSLS\_s, πραγματοποιείται εισαγωγή ή αντιστροφή εργασιών εντός ενός τυχαία επιλεγμένου εργοστασίου, ενώ στο RSLS\_m, η εναλλαγή ή (εναλλαγή+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 που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, διάγραμμα

Περιγραφή που δημιουργήθηκε αυτόματα

Στο πρώτο παράδειγμα του παραπάνω σχήματος επιλέγεται ένα εργοστάσιο τυχαία (π2). Στην συνέχεια επιλέγεται μια υπό ακολουθία τυχαίου μήκους g (το μήκος πρέπει να είναι μικρότερο από το σύνολο των εργασιών που έχουν ανατεθεί στο εργοστάσιο) για να ακολουθήσει η επιλογή τυχαίου σημείου εισαγωγής της ακολουθίας στο εργοστάσιο, στο οποίο θα εισάγουμε την υπό ακολουθία. Στο παράδειγμά μας επιλέγεται η ακολουθία των θέσεων 2,3 και 4 με τιμές 3,7,9 και εισάγεται μετά το σημείο εισαγωγής θέση 5 με τιμή 8.

Στο δεύτερο παράδειγμα αναπαριστάτε η λειτουργία αντιστροφής. Επιλέγεται τυχαία ένα εργοστάσιο από το οποίο θα επιλέξουμε στην συνέχεια μια υπό ακολουθία την οποία θα αντιστρέψουμε στην συνέχεια.

Τα τελευταία δύο παραδείγματα αφορούν την ανταλλαγή δύο τυχαία επιλεγμένων υπό ακολουθιών μεταξύ δύο τυχαία επιλεγμένων εργοστασίων και στο ένα από τα δύο παραδείγματα εφαρμόζουμε και αντιστροφή της σειράς εργασιών των επιλεγμένων υπό ακολουθιών.

**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') < ΤΤ(π)) then  π = π'  flag = true;  end if  end while  return π;  end |

### 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 που περιέχει κείμενο, γραμματοσειρά, διάγραμμα, αριθμός

Περιγραφή που δημιουργήθηκε αυτόματα

Στον παραπάνω πίνακα παρουσιάζονται παραδείγματα της μεθόδου RPLS. Στο πρώτο παράδειγμα επιλέγεται αρχικά ένα εργοστάσιο και την συνέχεια δύο σημεία. Το σημείο που επιλέγεται πρώτο αφαιρείται από την θέση που βρίσκεται και τοποθετείται μετά την θέση του δεύτερου σημείου. Παρομοίως γίνονται και οι ενέργειες στα υπόλοιπα παραδείγματα.

#### 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)[18]

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 παγκόσμιο ελάχιστο. Η SA είναι επίσης σχετικά εύκολη στην εφαρμογή και δεν απαιτεί εκ των προτέρων γνώση του χώρου αναζήτησης.

#### 6.2.2.2 ΚΡΙΤΗΡΙΟ ΑΠΟΔΟΧΗΣ

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 ΣΑ προέρχεται από τη στατιστική μηχανική και βασίζεται στην κατανομή πιθανοτήτων Boltzmann.[7]

Στην περίπτωσή μας εξετάζεται ένα σημείο αποδοχής το οποίο παρουσιάζεται με την παρακάτω εξίσωση:

(26)

όπου T0 είναι μια παράμετρος θερμοκρασίας και πρέπει να βαθμονομηθεί. Αυτό το κριτήριο αποδοχής θεωρεί τη λύση που παράγεται από την τοπική αναζήτηση για την επόμενη επανάληψη μόνο σε δύο περιπτώσεις. Εάν η λύση είναι καλύτερη από την τρέχουσα ή εάν ένας τυχαίος αριθμός r[0, 1] είναι μικρότερος από exp{-(TT(π ) - TT(π))/T}. Εδώ, TT(π’) και TT(π) υποδηλώνουν τις συνολικές τιμές καθυστέρησης για τις ανακατασκευασμένες και τις υπάρχουσες λύσεις, αντίστοιχα. [7]

Παρακάτω παρουσιάζεται ο ψευδοκώδικας της λύσης:

|  |
| --- |
| Procedure IG algorithm  π=NEHedd;  π=local search(π);  πb = π;  while (termination criterion not satisfied)  π'=π;  for y=1 to (d \* n)  πR = remove one job from π' regardless of the factories and insert it in πR;  end for  for y=1 to (d \* n)  x = πR(y)  for f=1 to F  Test job x in all posible positions of π'f;  TTf is the lowest TT obtained at position pf for factory f;  end for  fmin = arg(minf=1F)(TTf));  Insert job x in π'fmin at position pfmin resulting in lowest TT;  end for  π' = local search (π');  if (TT(π') < ΤΤ(π))  π = π';  if (TT(π) < ΤΤ(πb)  πβ = π  end if  elseif (rand <= exp{-(TT(π') - ΤΤ(π))/Τ})  π = π'  end if  end while  return πb  end |

## MLL BASED MECHANISM

Στοχεύοντας πάντα σε αποδοτικότερες μεθόδους για την βελτιστοποίηση του προβλήματος DPFSP προτείνεται μια υβριδική μέθοδος στην εφαρμογή του αλγόριθμου IG εισάγοντας μηχανισμούς με βάση την μέθοδο MLL προσθέτοντας 4 βασικές και 2 υβριδικές κινήσεις τοπικής αναζήτησης. [19]

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

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

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.

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.

THE HYBRID SWAP (HS) Οι κινήσεις JS και IS εκτελούνται ταυτόχρονα.

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

## 6.4 ITERATED GENETIC ALGORITHM

## 6.5 HYBRID GENETIC ALGORITHM

## 6.6 ΣΧΕΔΙΑΣΜΌΣ ΠΕΙΡΑΜΑΤΩΝ

# 7 ΑΞΙΟΛΟΓΗΣΗ ΚΑΙ ΑΝΑΛΥΣΗ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

## 7.1 ΠΑΡΟΥΣΙΑΣΗ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

### 7.1.2 ΑΝΑΛΥΣΗ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

### 7.1.3 ΣΥΓΚΡΙΣΗ ΚΑΙ ΣΥΜΠΕΡΑΣΜΑΤΑ

# 8 ΣΥΜΠΕΡΑΣΜΑΤΑ ΚΑΙ ΜΕΛΛΟΝΤΙΚΕΣ ΠΡΟΚΛΗΣΕΙΣ

# Βιβλιογραφία

|  |  |
| --- | --- |
| [1] | B. Naderi και R. Ruiz, «The distributed permutation flowshop scheduling problem,» 2018. |
| [2] | J. Kuhpfahl, Job Shop Scheduling with Consideration of Due Dates, 2015. |
| [3] | C. Gogos, «Solving the Distributed Permutation Flow-Shop Scheduling Problem Using Constrained Programming,» 2023. |
| [4] | P. Perez-Gonzalez και J. M.Framinan, «A review and classification on distributed permutation flowshop scheduling problems,» 2023. |
| [5] | G. Minella, R. Ruiz και M. Ciavotta, «A Review and Evaluation of Multiobjective Algorithms for the Flowshop Scheduling Problem,» 2008. |
| [6] | S. HASIJA και C. RAJENDRAN, «Scheduling in flowshops to minimize total tardiness of jobs,» 2003. |
| [7] | A. Khare και S. Agrawal, «Effective heuristics and metaheuristics to minimise total tardiness for the distributed permutation flowshop scheduling problem,» 2020. |
| [8] | C. . A. FLOUDAS και X. LIN, «Mixed Integer Linear Programming in Process Scheduling: Modeling, Algorithms, and Applications,» 2005. |
| [9] | M. R. Garey, D. S. Johnson και R. Sethi, «The Complexity of Flowshop and Jobshop Scheduling,» 1976. |
| [10] | E. Taillard, «Some efficient heuristic methods for the flow shop sequencing problem,» 1990. |
| [11] | K. YEONG-DAE KIM, «Heuristics for Flowshop Scheduling Problems,» 1993. |
| [12] | J. Thompson, Heuristics: An Overview, 2023. |
| [13] | H. Ramalhinho Lourenço, O. C. Martin και T. Stuetzle, HANDBOOK OF METAHEURISTICS (CHAPTER 5), 2010. |
| [14] | R. Ruiz και M. Concepcion, «A comprehensive review and evaluation of permutation flowshop heuristics to minimize flowtime». |
| [15] | OSMAN και POTTS, «Simulated annealing for permutation flow-shop scheduling». |
| [16] | T. Stuetzle, «Applying Iterated Local Search to the Permutation Flow Shop Problem». |
| [17] | R. RUIZ και T. STUETZLE, «A simple and effective IGA for the PFSP,» 2007. |
| [18] | S. Kirkpatrick, «Optimization by Simulated Annealing,» 1983. |
| [19] | P. Pourhejazy, C.-Y. Cheng, K.-C. Ying και N. H. Nam, «Meta-Lamarckian-based iterated greedy for optimizing,» 2022. |
| [20] | V. Fernandez-Viagas και J. M. Framinan, «NEH-based heuristics for the permutation flowshop scheduling problem to minimise total tardiness,» 2017. |
| [21] | R. Ruiz και P. Quan-Ke, «Iterated search methods for earliness and tardiness minimization in hybrid flowshops with due windows,» 2016. |