Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Το Άμεσο Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βάρδας Εμμανουήλ

Επιβλέπων: Δημήτρης Φωτάκης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εργαστήριο Λογικής και Επιστήμης Υπολογισμών
Αθήνα, Αύγουστος 2019

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Το Άμεσο Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βάρδας Εμμανουήλ

Επιβλέπων: Δημήτρης Φωτάκης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29η Αυγούστου 2019.

....	....	....
Δημήτρης Φωτάκης	Άρης Παγουρτζής	Νικόλαος Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Εργαστήριο Λογικής και Επιστήμης Υπολογισμών

Αθήνα, Αύγουστος 2019

....
Βάρδας Εμμανουήλ
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Στην παρούσα διπλωματική εργασία, παρουσιάζουμε το Άμεσο Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου (Online Min-Sum Set Cover), ένα άμεσο (online) ανάλογο του προβλήματος Ελάχιστου Αθροίσματος Κάλυψης Συνόλου (Min-Sum Set Cover), που προτάθηκε από τους Feige, Lovász, Tetali. Το πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου μπορεί να χρησιμοποιηθεί στη μοντελοποίηση προβλημάτων διάταξης διαδικτυακών αποτελεσμάτων, όπου αποτελέσματα αναζήτησης ή ενημερώσεις από τα κοινωνικά δίκτυα χρειάζεται να είναι σε διάταξη προσαρμοσμένη στις προτιμήσεις του χρήστη. Τα διαδικτυακά αποτελέσματα μπορούν να θεωρηθούν ως μία λίστα από στοιχεία και οι χρήστες ως σύνολα αυτών των στοιχείων, που προκύπτουν από τις προτιμήσεις τους. Το ζητούμενο του προβλήματος είναι να παραχθεί μία διάταξη της λίστας που να ελαχιστοποιεί το μέσο χρόνο επαφής (hitting time) των συνόλων. Ο χρόνος επαφής ορίζεται ως η θέση του πρώτου στοιχείου στη διάταξη, που βρίσκεται στο σύνολο. Αυτό το πλαίσιο μοντελοποιεί το χρόνο που απαιτείται από ένα χρήστη να σκανάρει τη λίστα αποτελεσμάτων από πάνω προς τα κάτω για να βρει το πρώτο επιθυμητό αποτέλεσμα. Παρ΄ όλα αυτά, το πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου υποθέτει ότι όλα τα σύνολα δίνονται εξ΄ αρχής. Ένα ρεαλιστικό σενάριο είναι η παραγόμενη διάταξη αποτελεσμάτων να ανανεώνεται τακτικά, καθώς νέα σύνολα εμφανίζονται έπειτα από ενέργειες που κάνουν οι χρήστες. Το Άμεσο Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου επιχειρεί να επιλύσει αυτό το πρόβλημα υποθέτοντας ότι τα σύνολα έρχονται σταδιακά, με άμεσο (online) τρόπο.

Ένα πιο απλό πρόβλημα, στο οποίο εντοπίζουμε σύνδεση με το πρόβλημα μας είναι το πολύ γνωστό Πρόβλημα Πρόσβασης Λίστας (List Accessing Problem), όπου στοιχεία, και όχι σύνολα, έρχονται online. Η δουλειά μας βασίζεται κυρίως στο πρόβλημα Πρόσβασης Λίστας και στον 2-ανταγωνιστικό (2-competitive) ντετερμινιστικό αλγόριθμο *Move-To-Front*. Αποδεικνύουμε ένα κάτω φράγμα $A + 1 - \frac{A(A+1)}{l+1}$ για το λόγο ανταγωνισμού (competitive ratio) κάθε ντετερμινιστικού αλγορίθμου, όπου A είναι η μέση πληθικότητα των συνόλων που δίνονται ως είσοδος και l το μήκος της λίστας. Επιπλέον, προτείνουμε τρεις αλγορίθμους εμπνευσμένους από τον αλγόριθμο *Move-To-Front*, τους $MoveFront$, $MoveLast$ και $MoveSet$. Δείχνουμε ότι ο $MoveFront$ είναι $l - A + 1$-competitive και οι $MoveLast$, $MoveSet$ ακριβώς l-competitive. Στην περίπτωση των πιθανοτικών αλγορίθμων, δείχνουμε ότι οι προτεινόμενοι αλγόριθμοι *Randomized Static* και *Randomized Move-To-Front* δεν παρέχουν κάποια εγγύηση υπογραμμικού λόγου ανταγωνισμού. Όλοι αυτοί οι αλγόριθμοι είναι *χωρίς μνήμη*, δηλαδή οι αποφάσεις τους βασίζονται αποκλειστικά στο σύνολο που έρχεται κάθε φορά και στις θέσεις των στοιχείων του στη λίστα. Καταλήγουμε ότι τέτοιες πρακτικές σχεδιασμού αλγορίθμων χωρίς μνήμη δεν αποδίδουν για το Άμεσο Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου.

Λέξεις κλειδιά: Άμεσοι Αλγόριθμοι, Ανταγωνιστική Ανάλυση, Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου, Πρόβλημα Πρόσβασης Λίστας, Συσσωμάτωση προτιμήσεων, Προβλήματα διάταξης

Abstract

In this thesis, we introduce the *Online Min-Sum Set Cover Problem*, an online counterpart of the *Min-Sum Set Cover Problem*, introduced by Feige, Lovász and Tetali. Min-Sum Set cover can be used to model web ranking problems, where web search results or social networks feed need to be placed in an order adapted to the user's preferences. Web results can be modeled as a list of elements, whereas users can be represented as sets over these elements. The objective of Min-Sum Set Cover is to induce an ordering in the list of elements that minimizes the average hitting time of sets, where hitting time is defined as the the first time step in which an element from the set is scheduled. Such setting models the time overhead of a user to scan a list of results from top to bottom in order to find the first result in which he/she is interested. However, Min-Sum Set Cover assumes that sets are given offline. A realistic scenario is that the results ordering is updated frequently, under the arrival of new set requests induced by actions of users. The Online Min-Sum Set Cover attempts to resolve this problem with the assumption that sets are given online.

A simpler online problem with which we detect relation is the well-known *List Accessing Problem*, where the online requests are single elements instead of sets. Our work is primarily motivated by the List Accessing and the tight 2-competitive *Move-To-Front* deterministic algorithm. We obtain a lower bound of $A + 1 - \frac{A(A+1)}{l+1}$ for the competitive ratio of any deterministic algorithm, where A is the average set cardinality of request sequence and l the list length. Also, we propose three *Move-To-Front*-like algorithms, $MoveFront$, $MoveLast$ and $MoveSet$. We show that $MoveFront$ is $l-A+1$-competitive and $MoveLast$, $MoveSet$ are tight l-competitive. For the randomized case, we show that proposed algorithms *Randomized Static* and *Randomized Move-To-Front* do not provide sublinear guarantees for their competitiveness. These algorithms are *memoryless*, i.e. their decisions are based only on the current requested set and its elements' position in the list. We conclude that such memoryless policies perform poorly for Online Min-Sum Set Cover.

Keywords: Online Algorithms, Competitive Analysis, Min-Sum Set Cover, List Accessing, Preference Aggregation, Ranking Problems

Ευχαριστίες

Η παρούσα διπλωματική εργασία αποτελεί το τέλος του δρόμου μίας πορείας έξι υπέροχων ετών. Σε αυτά τα χρόνια έμαθα πάρα πολλά, εξελίχθηκα και έζησα αμέτρητες όμορφες στιγμές. Σημαντικότερο από όλα όμως ήταν οι άνθρωποι που είχα την τύχη να γνωρίσω, να διδαχτώ από αυτούς και να συμπορευτώ. Θα ήθελα να ευχαριστήσω βαθύτατα τον κ. Φωτάκη για την καθοδήγηση, τη μεγάλη υπομονή και κατανόηση που έδειξε κατά την πορεία της διπλωματικής. Τον ευχαριστώ θερμά ως δάσκαλο για τα κίνητρα και την ευκαιρία που μου έδωσε να μυηθώ στον κόσμο των αλγορίθμων και της Θεωρητικής Πληροφορικής. Σε αυτήν την αγαπημένη σχέση με την Επιστήμη Υπολογιστών συνέβαλαν τα μέγιστα οι κ. Παπασπύρου, κ. Παγουρτζής και κ. Ζάχος. Τους ευχαριστώ γιατί αποτέλεσαν πρότυπα δασκάλων για μένα. Επίσης, ευχαριστώ πολύ τα παιδιά από το Corelab. Τον Λουκά για τη συνεργασία και τις συζητήσεις μας πάνω στους online αλγορίθμους. Τον Στρατή για τη διάθεση και το άμεσο ενδιαφέρον που έδειξε να βοηθήσει σε οποιαδήποτε απορία. Τον Παναγιώτη για το χιούμορ και την ανάλαφρη διάθεση που έκανε να επικρατεί στο εργαστήριο.

Θέλω να ευχαριστήσω από τα βάθη της καρδιάς μου την οικογένεια μου και τους φίλους μου. Ευχαριστώ τον πατέρα μου, που αποτέλεσε τον μεγάλο μου δάσκαλο και με έκανε να αγαπήσω τα μαθηματικά. Ευχαριστώ την μητέρα μου για τη αμέριστη στήριξη σε κάθε επίπεδο όλα αυτά τα χρόνια. Ευχαριστώ τον Γιάννη Β. και Κώστα Ζ. που είναι πάντα δίπλα μου από τα πολύ παλιά χρόνια. Ένα μεγάλο ευχαριστώ για τους καλούς μου φίλους Κώστα Μ., Αλέξη Χ., Αθανασία Γ., Γιάννη Π., Νάγια Κ., Γιώργο Τ., Σαββίνα Δ., Ρομέο Τσ., Κοσμά Σ. για τα πανέμορφα φοιτητικά χρόνια που περάσαμε. Έχω παραλείψει αρκετούς, γιατί το περιθώριο της σελίδας δεν επαρκεί. Τους ευγνωμονώ όλους. Στη σχολή, είχα την ευτυχία να συμπορευτώ και να διδαχτώ πολλά πράγματα από αρκετούς συμφοιτητές μου. Τους εύχομαι κάθε καλό στα επόμενα βήματα μετά τη σχολή. Τέλος, οφείλω ένα τεράστιο ευχαριστώ στην Αλεξάνδρα για την τεράστια υπομονή και την αμέριστη στήριξη της σε δύσκολες στιγμές. Χωρίς αυτήν, δεν θα είχα καταφέρει πολλά όλα αυτά τα χρόνια. Ευχαριστώ όλους τους παραπάνω για άλλη μια φορά, γιατί πίστευαν σε μένα όταν σταματούσα να πιστεύω στον εαυτό μου. Όπως αφιερώνουν και οι Alan Borodin, Ran El-Yaniv στο βιβλίο Online Computation and Competitive Analysis, ακόμα και αν δεν μπορώ να προβλέψω το μέλλον, ξέρω πως μπορώ να στηρίζομαι σε εσάς.

Μανώλης

“Things turn out best for the people who make
the best of the way things turn out”
-John Wooden

Contents

Chapter 1

Εκτεταμένη Ελληνική Περίληψη

Στο κεφάλαιο αυτό, συνοψίζουμε το περιεχόμενο της παρούσας διπλωματικής, δίνοντας βασικούς ορισμούς και θεωρήματα, χωρίς αποδείξεις.

1.1 Εισαγωγή

Η διάταξη διαδικτυακών αποτελεσμάτων με βάση τις προτιμήσεις των χρηστών παίζει σημαντικό ρόλο στη σχεδίαση εφαρμογών φιλικών προς το χρήστη. Για παράδειγμα, στις πλατφόρμες κοινωνικών δικτύων, ενδιαφερόμαστε να δούμε αποτελέσματα από λογαριασμούς χρηστών με τους οποίους αλληλεπιδρούμε περισσότερο. Στο καθημερινό feed ειδήσεων, θέλουμε να λαμβάνουμε ενημερώσεις για θέματα που ανταποκρίνονται στα ενδιαφέροντα μας. Σε μία ιστοσελίδα, μπορεί να συναντήσουμε διαφημίσεις σχετικές με τις ανάγκες μας, από ένα σύνολο δυνατών διαθέσιμων. Στη σύγχρονη εποχή, πληροφορίες για τα ενδιαφέροντα των χρηστών συγκεντρώνονται από τις ενέργειες κάθε χρήστη, το ιστορικό επισκέψεων, τα clicks κτλ. με σκοπό την εξαγωγή ενός προφίλ για τον χρήστη και την δημιουργία μίας προσωποποιημένης διαδικτυακής αναζήτησης. Κάποια αποτελέσματα στις μηχανές αναζήτησης μπορούν πλέον να εμφανίζονται διατεταγμένα με βάση τις προτιμήσεις του κάθε χρήστη. Ένας από τους πολλούς στόχους αυτής της 'συσσωμάτωσης' προτιμήσεων σε μία διάταξη είναι να μειώσει την προσπάθεια του χρήστη να βρει αποτελέσματα που τον ενδιαφέρουν. Καθώς ο χρήστης σκανάρει μία λίστα αποτελεσμάτων από πάνω προς τα κάτω, αυτή η προσπάθεια μπορεί να ποσοτικοποιηθεί με τον χρόνο που χρειάζεται ο χρήστης να διαβάσει τα αποτελέσματα αναζήτησης μέχρι να βρει το πρώτο αποτέλεσμα που τον ενδιαφέρει.

Μία αφαίρεση αυτού το προβλήματος είναι η ακόλουθη: Δίνεται μία λίστα από στοιχεία, που αντιστοιχεί στα πιθανά αποτελέσματα αναζήτησης. Δίνονται επίσης σύνολα από στοιχεία αυτής της λίστας που φτάνουν σε ζωντανό χρόνο, που αντιστοιχούν σε αποτελέσματα για το οποία ενδιαφέρεται ο χρήστης. Αναλόγως την εφαρμογή, αυτά τα σύνολα μπορούν να αναφέρονται σε ένα μόνο χρήστη, που προκύπτουν από διάφορες ενέργειες και πιθανές τροποποιήσεις στα ενδιαφέροντα του, όπως στα κοινωνικά δίκτυα, είτε σε πολλαπλούς χρήστες, όπως στην περίπτωση μίας μηχανής αναζήτησης. Μας ενδιαφέρει λοιπόν να σχεδιάσουμε αλγορίθμους που επαναδιαττάσουν τα αποτελέματα καθώς νέα σύνολα χρηστών έρχονται, με σκοπό να μειώσουμε το χρόνο πρόσβασης για μελλοντικά σύνολα χρηστών που θα έρθουν. Ένα τέτοιο σενάριο μπορεί να μοντελοποιηθεί από το *Άμεσο Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου* (Online Min Sum Set Cover), που παρουσιάζουμε σε αυτή τη διπλωματική. Η υπάρχουσα δουλεία σχετικά με το πρόβλημα μας αφορά το πρόβλημα

Ελάχιστου Αθροίσματος Κάλυψης Συνόλου (Min-Sum Set Cover). Στο πρόβλημα αυτό θεωρείται ότι όλα τα σύνολα δίνονται εξαρχής, π.χ προερχόμενα από log files χρηστών. Σε ένα ρεαλιστικό σενάριο που τα σύνολα έρχονται σταδιακά, έχουμε την *άμεση* (online) εκδοχή του προβλήματος που θίγουμε εδώ. Επιπλέον, το πρόβλημα μας αποτελεί γείκευση του προβλήματος *Πρόσβασης Λίστας* (List Accessing), στο οποίο αντί για σύνολα, έρχονται στοιχεία με άμεσο τρόπο. Βασιζόμαστε σε τεχνικές από το πρόβλημα αυτό για να βγάλουμε κάποια αποτελεσματα για το πρόβλημα μας.

1.2 Άμεσοι Αλγόριθμοι

Ένας *άμεσος* (online) αλγόριθμος λαμβάνει ως είσοδο μία ακολουθία εισόδου της μορφής $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(n)$. Κάθε τμήμα εισόδου/αίτημα πρέπει να εξυπηρετηθεί από τον αλγόριθμο με τη σειρά εμφάνισης του, κατά τη στιγμή που αυτό έρχεται. Όταν ο αλγόριθμος εξυπηρετεί το αίτημα $\sigma(t)$, δεν έχει καμία γνώση για τα αιτήματα $\sigma(t')$, για $t' > t$, αλλά γνωρίζει όλα τα αιτήματα $\sigma(t')$, για $t' \leq t$. Επιπλέον, το μέγεθος n της ακολουθίας μπορεί να μην είναι γνωστό. Κάθε αίτημα προκαλεί ένα κόστος ή κέρδος για τον αλγόριθμο. Αναλόγως το πρόβλημα, ο στόχος είναι να ελαχιστοποιηθεί το συνολικό κόστος ή να μεγιστοποιηθεί το συνολικό κέρδος που προκύπτει από ολόκληρη την είσοδο/ακολουθία αιτημάτων.

Είναι προφανές ότι αυτή η μη ολοκληρωμένη εικόνα της εισόδου μαζί με τις αποφάσεις που παίρνει ο αλγόριθμος για κάθε κομμάτι εισόδου μπορεί να μην επιτρέψουν στον αλγόριθμο να φτάσει στη βέλτιστη λύση στο τέλος της εκτέλεσης. Ένα βασικό ερώτημα δημιουργείται: Πως μετράμε την επίδοση ενός τέτοιου αλγορίθμου· Την απάντηση δίνει η *ανταγωνιστική ανάλυση* (competitive analysis), ένας όρος που προτάθηκε από τους Karlin et. al [37] και συστήθηκε από τους Sleator και Tarjan [54]. Κατά το competitive analysis, η έξοδος ενός online αλγορίθμου συγκρίνεται με αυτή του βέλτιστου offline αλγορίθμου. Αυτός είναι ο αλγόριθμος που έχει γνώση για ολόκληρη την ακολουθία εισόδου από την αρχή της εκτέλεσης του και απαντάει βέλτιστα στο κάθε αίτημα. Το competitive analysis δεν κάνει καμία υπόθεση για την κατανομή που ακολουθεί η είσοδος. Αντίθετα, είναι μία μορφή ανάλυσης χειρότερης περίπτωσης, με την έννοια ότι ένας αλγόριθμος θεωρείται καλός όταν αποδίδει καλά στη χειρότερη δυνατή είσοδο, αυτή δηλαδή που δημιουργεί μεγάλο κόστος για τον online αλγόριθμο, αλλά μικρό κόστος για τον βέλτιστο offline. Παρουσιάζουμε τους ακόλουθους ορισμούς:

Ορισμός 1. *Για μία ακολουθία εισόδου σ, έστω $ALG(\sigma)$ και $OPT(\sigma)$ τα κόστη του online και offline αλγορίθμου αντίστοιχα. Ο online αλγόριθμος λέγεται **c-competitive** αν υπάρχει σταθερά a τέτοια ώστε για κάθε ακολουθία σ:*

- $ALG(\sigma) \leq c \cdot OPT(\sigma) + a$, *σε πρόβλημα ελαχιστοποίησης*
- $ALG(\sigma) \geq \frac{1}{c} \cdot OPT(\sigma) - a$, *σε πρόβλημα μεγιστοποίησης*

*Αν $a \leq 0$, ο αλγόριθμος λέγεται **strictly c-competitive**.*

Ορισμός 2. *Το infimum όλων των τιμών c για τις οποίες ο online αλγόριθμος είναι c-competitive ονομάζεται λόγος ανταγωνισμού (**competitive ratio**) του online αλγορίθμου και συμβολίζεται ως $R(ALG)$.*

Σε περίπτωση που χρησιμοποιούμε κάποιον πιθανοτικό online αλγόριθμο, τη θέση του $ALG(\sigma)$ παίρνει το $\mathbb{E}[ALG(\sigma)]$, δηλαδή το αναμενόμενο κόστος του ALG πάνω στις τυχαίες επιλογές που κάνει ο αλγόριθμος.

Τα online προβλήματα συναντούν πολλές εφαρμογές στο interactive computing, στις δομές δεδομένων, στις δυκτιακές εφαρμογές, στο σχεδιασμό κίνησης, στα προβλήματα δρομολόγησης, στη διαχείριση υπολογιστικών πόρων και σε πολλά άλλα. Μερικά πολύ γνωστά προβλήμα είναι το πρόβλημα Ski Rental [37], το πρόβλημα Paging [54] και το πρόβλημα k-server [41].

1.3 Το Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου

Το πρόβλημα συστάθηκε από τους Feige, Lovász, Tetali [27][28]. Δίνεται ένα σύνολο στοιχείων $E = \{x_1, x_2, \dots, x_n\}$ και μία συλλογή από υποσύνολα αυτών των στοιχείων $S = \{S_1, S_2, \dots, S_m\}$. Έστω μία μετάθεση π των στοιχείων του E. Τη μετάθεση αυτή μπορούμε να τη φανταστούμε ως μια διαδικασία δρομολόγησης των στοιχείων, δηλαδή το στοιχείο στη θέση i θεωρούμε πως δρομολογείται τη χρονική στιγμή i. Ορίζουμε ως *χρόνο κάλυψης f ενός συνόλου S_j* τη χρονική στιγμή i κατά την οποία $\pi(i) \in S_j$ δηλαδή:

$$f(S_j) = \min_{i \in [n]: x_i \in S_j} \pi^{-1}(i)$$

Ο στόχος είναι να βρούμε μία μετάθεση π^* ώστε να ελαχιστοποιεί το άθροισμα χρόνων κάλυψης των συνόλων, δηλαδή:

$$\pi^* = \arg\min_{\pi} \sum_{s \in S} f(s)$$

Το πρόβλημα έχει αποδειχθεί ότι είναι NP-Hard. Ένας απλός προσεγγιστικός αλγόριθμος είναι ο εξής **άπληστος αλγόριθμος**:

Σε κάθε χρονική στιγμή, επέλεξε το στοιχείο που καλύπτει τα περισσότερα εναπομείναντα ακάλυπτα σύνολα.

Οι Feige, Lovász, Tetali [27][28] έδειξαν τις ακόλουθες προτάσεις:

Θεώρημα 1.
1. *Ο άπληστος αλγόριθμος έχει λόγο προσέγγισης* 4.
2. *Είναι NP-Hard να προσεγγιστεί το Min-Sum Set Cover με λόγο προσέγγισης* $4 - \epsilon$, *για κάθε* $\epsilon > 0$.

Η σημασία της έννοιας Ελάχιστου Αθροίσματος φαίνεται από την ύπαρξη πολλών σχετικών προβλημάτων:

- *Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Κορυφών* σε ένα γράφο. [28] [27]
- *Πρόβλημα Ελάχιστου Αθροίσματος Χρωματισμού* ενός γράφου. [15]

- *Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου* με περιορισμούς της μορφής ‵το στοιχείο x_i πρέπει να προηγείται του στοιχείου x_j στη διάταξη′ (*precedence constraints*). [43]

- *Γενικευμένο Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου*, όπου ο χρόνος κάλυψης ενός στοιχείου ορίζεται ως η χρονικη στιγμή που δρομολογείται το k-οστό στοιχείο του συνόλου. [12]

- *Πρόβλημα Submodular Διάταξης*, όπου αντί για σύνολα υπάρχουν μονότονες μη αρνητικές submodular συναρτήσεις f_i και ως χρόνος κάλυψης ορίζεται η χρονική στιγμή t που $f_i(\{e_{\pi(1)}, e_{\pi(2)}, \ldots, e_{\pi(t)}\}) = 1$. [11]

1.4 Το Πρόβλημα Πρόσβασης Λίστας

Στο *Πρόβλημα Πρόσβασης Λίστας* μας δίνεται μία μη ταξινομημένη λίστα με l στοιχεία και μία ακολουθία από αιτήματα στοιχείων της λίστας που έρχεται με online τρόπο. Ο στόχος είναι να σχεδιαστούν αλγόριθμοι που μειώνουν τα κόστη πρόσβασης στη λίστα από μελλοντικά αιτήματα στοιχείων. Συγκεκριμένα, ένας αλγόριθμος πληρώνει ένα *κόστος πρόσβασης* ίσο με τη θέση του στοιχείου που ζητείται στη λίστα. Στη συνέχεια, μπορεί να πραγματοποιήσει αντιμεταθέσεις διαδοχικών στοιχείων στη λίστα με δύο διαφορετικούς τρόπους:

- *Δωρεάν αντιμεταθέσεις*: Ο αλγόριθμος δικαιούται να μετακινήσει το ζητούμενο στοιχείο όσες θέσεις επιθυμεί πιο μπροστά στη λίστα, χωρίς να πληρώσει τίποτα.

- *Αντιμεταθέσεις με πληρωμή*: Ο αλγόριθμος δικαιούται να αντιμεταθέσει οσαδήποτε διαδοχικά στοιχεία στη λίστα με κόστος 1 για κάθε αντιμετάθεση. Το κόστος αυτό ονομάζεται *κόστος μετακίνησης*.

Έτσι, ο στόχος είναι να ελαχιστοποιηθεί το συνολικό άθροισμα από τα κόστη πρόσβασης και κόστη μετακινησης, που επάγεται από ολόκληρη την ακολουθία εισόδου.

Κάποια βασικά αποτελέσματα στα οποία βασιζόμαστε για την εξαγωγή αποτελεσμάτων στο Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου παραθέτονται παρακάτω.

Το πρώτο σημαντικό αποτέλεσμα αποδίδεται στους Karp και Raghavan όπως αναφέρεται στο [34]. Αφορά ένα κάτω φράγμα για το competitive ratio κάθε ντετερμινιστικού αλγορίθμου.

Θεώρημα 2. *Κάθε ντετερμινιστικός αλγόριθμος έχει competitive ratio τουλάχιστον* $2-\frac{2}{l+1}$*, όπου* l *είναι το μέγεθος της λίστας.*

Η ιδέα της απόδειξης βασίζεται στο να δημιουργήσουμε μία ακολουθία εισόδου που να επιφέρει μεγάλο κόστος στον αλγόριθμο, συγκεκριμένα μπορεί να ζητείται πάντα το τελευταίο στοιχείο της λίστας, την ίδια ώρα που το κόστος του βέλτιστου offline αλγορίθμου να φράσσεται άνω από μία ποσότητα. Η ποσότητα αυτή, στη συγκεκριμένη περίπτωση, είναι το μέσο κόστος των static offline αλγορίθμων που μπορεί να υπολογιστεί εύκολα. Static offline ονομάζονται οι αλγόριθμοι που πληρώνουν ένα αρχικό κόστος μετακίνησης για να αναδιατάξουν τη λίστα σε μία από τις $l!$ δυνατές διατάξεις και δεν κάνουν καμία αλλαγή καθόλη την ακολουθία εισόδου.

Ένας online αλγόριθμος που προκύπτει πολύ φυσικά είναι ο ακόλουθος:

Move-To-Front (MTF): Μετά την πρόσβαση στο στοιχείο της λίστας που ζητήθηκε, μετακίνησε το στοιχείο στην κορυφή της λίστας, χωρίς να αλλάξεις τις σχετικές θέσεις οποιωνδήποτε άλλων στοιχείων.

Οι Sleator και Tarjan απέδειξαν ότι ο MTF είναι $2 - \frac{1}{l}$-ανταγωνιστικός [54]. Βασίστηκαν στην τεχνική potential function που χρησιμοποιείται από την Amortized Analysis, που προτάθηκε από τον Tarjan [56]. Η Amortized Analysis είναι μία ανάλυση *μέσης περίπτωσης*, κατά την οποία το αποτέλεσμα μίας πράξη ενός αλγορίθμου αξιολογείται στο σύνολο των πράξεων του αλγορίθμου και όχι μεμονωμένα τη στιγμή που γίνεται αυτή η πράξη.

Αργότερα, ο Irani [34] απέδειξε ότι ο MTF έχει competitive ratio ακριβώς ίσο με το κάτω φράγμα του προηγούμενου θεωρήματος. Συγκεκριμένα έχουμε:

Θεώρημα 3. *Ο αλγόριθμος Move-To-Front είναι* $2 - \frac{2}{l+1}$*-competitive, όπου* l *είναι το μέγεθος της λίστας.*

Το Πρόβλημα Πρόσβασης Λίστας έχει μελετηθεί εκτενώς στην online βιβλιογραφία. Κάποια αποτελέσματα που δεν αναλύουμε εδώ είναι:

- Το offline πρόβλημα έχει αποδειχθεί από τον Ambuhl ότι είναι NP-Hard [9].
- Το καλύτερο κάτω φράγμα μέχρι σήμερα για το competitive ratio πιθανοτικών αλγορίθμων είναι 1.5 και έχει αποδειχθεί από τον Teia [57].
- Ο καλύτερος πιθανοτικός αλγόριθμος μέχρι σήμερα έχει competitive ratio ίσο με 1.6 και έχει αποδειχθεί από τους Albers et. al [8].

1.5 Το Άμεσο Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου

Το Άμεσο Πρόβλημα Ελάχιστου Αθροίσματος Κάλυψης Συνόλου μπορούμε να το δούμε είτε ως μία online έκδοση του προβλήματος Ελάχιστου Αθροίσματος Κάλυψης Συνόλου (τα σύνολα έρχονται με online τρόπο), είτε ως μία πολυδιάστατη έκδοση του προβλήματος Πρόσβασης Λίστας (αντί για στοιχεία, έρχονται σύνολα με online τρόπο). Σε ένα online πρόβλημα όπως το δικό μας, πρέπει να επιτρέψουμε σε έναν αλγόριθμο να κάνει αναδιατάξεις στη λίστα με σκοπό να προσαρμόζεται στις νέες εισόδους και να μειώνει το κόστος πρόσβασης μελλοντικών ζητούμενων συνόλων. Ένας τέτοιος αλγόριθμος επομένως πρέπει να έχει την ελευθερία να μπορεί να επιλέγει ποιά στοιχεία του εκάστοτε ζητούμενου συνόλου πρέπει να προσπελάσει και ίσως να μετακινήσει στη λίστα. Έτσι, σαν *κόστος πρόσβασης* για κάθε ζητούμενο σύνολο που έρχεται online ορίζουμε τη μεγαλύτερη θέση του στοιχείου στη λίστα, από τα στοιχεία του συνόλου που ο αλγόριθμος πραγματοποίησε πρόσβαση. Στη συνέχεια ο αλγόριθμος, μπορεί να κάνει αντιμεταθέσεις διαδοχικών στοιχείων στη λίστα με δύο τρόπους:

- *Δωρεάν αντιμεταθέσεις:* Αν ο αλγόριθμος πλήρωσε κόστος πρόσβασης i λόγω ενός στοιχείου του συνόλου στη θέση αυτή, μπορεί να μετακινήσει οποιαδήποτε στοιχεία x_k του συνόλου, με $k \leq i$, όσες θέσεις επιθυμεί πιο μπροστά στη λίστα.
- *Αντιμεταθέσεις με πληρωμή:* Ο αλγόριθμος δικαιούται να αντιμεταθέσει οσαδήποτε διαδοχικά στοιχεία στη λίστα με κόστος 1 για κάθε αντιμετάθεση. Το κόστος αυτό ονομάζεται *κόστος μετακίνησης*.

Έτσι, το κόστος πρόσβασης σε κάθε ζητούμενο σύνολο είναι *διαζευκτικό*, με την έννοια ότι έγκειται στον αλγόριθμο το πόσο θα επιλέξει να πληρώσει ώστε να έχει την ελευθερία να μετακινήσει τα ανάλογα στοιχεία δωρεάν. Ο στόχος επομένως είναι να ελαχιστοποιηθεί το συνολικό κόστος από τα κόστη πρόσβασης και κόστη μετακίνησης, που επάγεται από ολόκληρη την ακολουθία εισόδου.

Παρακάτω παραθέτουμε χωρίς απόδειξη τα αποτελέσματα μας για το πρόβλημα.

Πρόταση 1. *Για ακολουθίες εισόδου με μέση πληθικότητα συνόλου* A *σε μία λίστα μεγέθους* l*, κάθε ντετερμινιστικός αλγόριθμος έχει competitive ratio τουλάχιστον* $A+1-\frac{A(A+1)}{l+1}$

Το θεώρημα αυτό βασίζεται στην averaging τεχνική που συζητήσαμε στο κάτω φράγμα για ντετερμινιστικούς αλγορίθμους στο Πρόβλημα Πρόσβασης Λίστας. Όπως βλέπουμε, υπάρχει μία εξάρτηση από την παράμετρο A. Στην περίπτωση που αναζητάμε έναν ανταγωνιστικό αλγόριθμο για όλες τις δυνατές ακολουθίες εισόδου, για την τιμή του A που μεγιστοποιεί την παραπάνω ποσότητα παίρνουμε:

Πρόταση 2. *Ένας ντετερμινιστικός αλγόριθμος που δέχεται όλες τις δυνατές ακολουθίες εισόδου έχει competitive ratio* $\Omega(l/4)$.

Η πρόταση αυτή είναι σημαντική. Κάθε αλγόριθμος είναι τετριμμένα l-competitive. Επομένως, το καλύτερο που μπορούμε να κάνουμε είναι να γεφυρώσουμε το γραμμικό διάστημα $(\frac{l}{4}, l]$ στο competitive ratio. Το αποτέλεσμα αυτό δηλώνει πως ενώ στο Πρόβλημα Πρόσβασης Λίστας που ζητείται ένα στοιχείο κάθε φορά, μπορούμε να βρούμε 2-competitive αλγόριθμο, όταν η είσοδος μετατρέπεται σε σύνολα στοιχείων οποιουδήποτε μεγέθους, δεν υπάρχει αλγόριθμος που μπορεί να πετύχει καλύτερο από γραμμικό competitive ratio. Για το λόγο αυτό μας απασχολούν αλγόριθμοι που να αποδίδουν καλά και για συγκεκριμένες τιμές της παραμέτρου A, προσπαθώντας να πλησιάσουμε την τιμή του κάτω φράγματος $A+1-\frac{A(A+1)}{l+1}$.

Στη συνέχεια, προτείνουμε τρεις ντετερμινιστικούς αλγορίθμους, βασισμένους στον αλγόριθμο Move-To-Front από το πρόβλημα Πρόσβασης Λίστας. Αυτοί είναι οι ακόλουθοι:

- ***MoveFront (MF)***: Κάνε πρόσβαση στο πρώτο στοιχείο του συνόλου στη λίστα και μετακίνησε το στην κορυφή της λίστας.
- ***MoveLast (ML)***: Κάνε πρόσβαση στο τελευταίο στοιχείο του συνόλου στη λίστα και μετακίνησε το στην κορυφή της λίστας.
- ***MoveSet (MS)***: Κάνε πρόσβαση στο τελευταίο στοιχείο του συνόλου στη λίστα και μετακίνησε όλα τα στοιχεία του συνόλου στην κορυφή της λίστας, διατηρώντας τη σχετική διάταξη τους.

Ο αλγόριθμος MF ακολυθεί την ιδέα ότι το πρώτο στοιχείο του συνόλου στη λίστα αντιπροσωπεύει το σύνολο οπότε πρέπει να μετακινηθεί μπροστά. Δείξαμε την ακόλουθη πρόταση:

Πρόταση 3. *Ο MoveFront είναι* $l-A+1$*-competitive για ακολουθίες εισόδου με μέση πληθικότητα* $A \geq 2$.

Ο αλγόριθμος ML βασίζεται στο ότι το τελευταίο στοιχείο του συνόλου στη λίστα είναι αυτό που μπορεί να επιφέρει το μεγαλύτερο κόστος πρόσβασης για αυτό το σύνολο οπότε πρέπει να μετακινηθεί μπροστά. Παρολαυτά, παίρνουμε το ακόλουθο αποτέλεσμα:

Πρόταση 4. *Ο MoveLast είναι ακριβώς l-competitive για ακολουθίες εισόδου με μέση πληθικότητα* $A \geq 2$.

Αφού ο ML πληρώνει πάντα για την πρόσβαση του τελευταίου στοιχείου του συνόλου στη λίστα, μπορεί να μετακινήσει όλο το σύνολο μπροστά με μηδενικό κόστος, όπως ορίσαμε για τις δωρεάν αντιμεταθέσεις. Αυτό κάνει ο αλγόριθμος MS. Όμως, παίρνουμε την ακόλουθη πρόταση:

Πρόταση 5. *Ο MoveSet είναι ακριβώς l-competitive για ακολουθίες εισόδου με μέση πληθικότητα* $A \geq 2$.

Η απόδειξη των παραπάνω προτάσεων βασίζεται στην ιδέα ότι μπορούμε πάντα να κατασκευάσουμε μία ακολουθία εισόδου που επιφέρει μεγάλο κόστος ανά ζητούμενο σύνολο, π.χ ζητώντας πάντα σε κάθε σύνολο το τελευταίο στοιχείο της λίστας και επιπλέον ένα σταθερό στοιχείο της λίστας που εμφανίζεται σε κάθε ζητούμενο σύνολο. Το σταθερό στοιχείο αυτό μπορεί ο βέλτιστος αλγόριθμος να το μετακινήσει εξ' αρχής στην κορυφή της λίστας και στη συνέχει να πληρώνει για κάθε σύνολο κόστος ίσο με 1. Έτσι, το competitive ratio μπορεί να γίνει μεγάλο.

Επιπλέον, μελετήσαμε τους παρακάτω δύο πιθανοτικούς αλγορίθμους:

- *Randomized Static:* Επίλεξε με ομοιόμορφα τυχαίο τρόπο μια αρχική διάταξη της λίστας. Σε κάθε ζητούμενο σύνολο, πλήρωσε τη θέση του πρώτου στοιχείου του συνόλου στη λίστα και μην κάνεις τίποτα.
- *Randomized Move-To-Front (RMTF):* Σε κάθε ζητούμενο σύνολο, επίλεξε με ομοιόμορφα τυχαίο τρόπο το στοιχείο του συνόλου προς πρόσβαση και μετακίνησε το στην κορυφή της λίστας.

Για τον Randomized Static δείξαμε την ακόλουθη πρόταση:

Πρόταση 6. *Ο Randomized Static έχει competitive ratio το πολύ* $\frac{l+1}{A+1}$*, για ακολουθίες συνόλων πληθικότητας* A.

Για τον Randomized Move-To-Front, μας απασχόλησε η συμπεριφορά του για σύνολα μικρής πληθικότητας. Έτσι δείξαμε ότι:

Πρόταση 7. *Ο Randomized Move-To-Front έχει competitive ratio* $\Omega(l/4)$*, για ακολουθίες εισόδου με σύνολα μεγέθους* 2.

Παρατηρούμε ότι ενώ το ντετερμινιστικό κάτω φράγμα για $A = 2$ είναι ουσιαστικά 3, παρά τη χρήση πιθανοτήτων, ο $RMTF$ δίνει competitive ratio γραμμικό $\Omega(l/4)$.

Το βασικό συμπέρασμα της παραπάνω δουλειάς είνα ότι τεχνικές της λογικής Move-To-Front δεν δίνουν αποτελέσματα κοντά στο κάτω φράγμα που αποδείξαμε. Αυτοί οι αλγόριθμοι είναι *χωρίς μνήμη*, οι αποφάσεις τους δηλαδή εξαρτώνται αοκλειστικά από το παρόν ζητούμενο σύνολο και την παρούσα λίστα. Επομένως, παρέχουμε έναν άξονα για μελλοντική δουλειά πάνω σε αλγορίθμους που να συνυπολογίζουν στις αποφάσεις τους και στοιχεία από τα σύνολα που ήρθαν στο παρελθόν.

Chapter 2

Introduction

Web search ranking plays an important role in the design of user-friendly web applications that interact with the users' preferences. For example, in social media platforms, we are interested in viewing the latest posts from page accounts with which we interact mostly. In our daily news feed, we want to receive updates on subjects that reflect our preferences. When accessing a website or web application, advertisements relative to web results that we searched in the past may pop up. Nowadays, it is the canon that web search engines and modern applications try to gather information from users' previous actions, clicks and searches in order to extract a user profile and induce a more personalized user experience. On the other hand, some web applications keep a global ordering of data, for example latest trends in videos or music, with which users interact mostly. All these problems lie in the field of preference aggregation that aims to set web data in a particular order that satisfies a specific goal. One such objective is to minimize the user effort to find information relevant to the user's interest. As the user scans web results from top to bottom, this effort can be considered as the amount of time it takes to find the first relevant result appeared in the list of web results.

One abstraction that can be used to model this problem is the following: A list of elements is given, representing the list of possible subject results. Sets of elements from this list, that represent results relevant to a user's interests and preferences, arrive in real time. Depending on the application, this sequence of sets may correspond to either one user, for example in case of social media feed, where each set may arrive on every time a user performs some new actions that perhaps modify existing preferences or introduce new ones, or in multiple users, like in the top trend case and web search results mentioned above. In these settings, we are interested in designing algorithms that reorder web results 'on the fly', in order to reduce access time in the arrival of future sets. Such setting can be modeled by the *Online Min-Sum Set Cover Problem* which we introduce in this thesis. To understand the problem, we first need to describe the terms *online* and *Min-Sum Set Cover.*

Online Algorithms

Online Min-Sum Set Cover is an *online problem.* In contrast to the traditional framework for algorithm design, in an online problem the input is not complete or available from the beginning of execution, but is revealed gradually in parts. On every arriving piece of data, the *online algorithm* must respond with an action before processing the next piece, based on the partial knowledge of pieces that have arrived so far. Any algorithm is completely unaware of future inputs. The goal of an online algorithm is to optimize an objective function as if it had all the input from the beginning. One well-studied measure of performance for online algorithms is the *competitive ratio* studied by *competitive analysis*, introduced by Sleator and Tarjan [54] and Karlin et. al [37]. The competitive ratio measures the performance of an online algorithm compared to that of the optimal offline solution OPT. Most important, it is a worst-case measure, i.e. an algorithm is considered 'good' if it performs 'well' on the hardest input instances. We thus say that an online algorithm ALG is c-competitive if for any sequence of input requests σ, there exists a constant b such that $ALG \leq c \cdot OPT + b$, in case of a minimization problem.

The online setting models a great number of problems that input arrives gradually and response need to be immediate. Many problems of that nature occur in the field of interactive computing, data structures, networks, motion planning, resource allocation and more [21] [5] [35]. For example, the online setting occurs naturally in the problems below.

- *Ski Rental Problem:* [37] Each day, we have to decide whether to rent a given good for this day or buy it for the rest of all days. Yet, we do not know the number of days in advance.

- *k-server Problem:* [41] We have k mobile servers and requested points in a metric space appear online. A point is served if a server is moved to it. We are interested in making a schedule of servers that minimizes the distance covered to serve all incoming requests. Nothing is known for the future requested points.

- *Paging:* [54] Which pages need to be evicted from a fast memory unit on the arrival of requested memory pages, in order to reduce future page faults? Pages arrive online.

Min-Sum Set Cover Problem

On the other hand, *Min-Sum Set Cover Problem*, introduced by Feige, Lovász, Tetali [27], provides a theoretical framework for many problems that aim to satisfy multiple demands under the goal of minimum total latency. It can be considered as a latency version of Min Set Cover. Specifically, a number of sets are given that jointly cover a number of elements. The goal is to find a scheduling for these elements such that the sum of cover times is minimized. The cover time of a set is defined as the first time step in which an element from the set is scheduled. The problem is NP-Hard. Feige et. al [27][28] proposed a 4-approximate greedy algorithm, also proving that the algorithm achieves a tight approximation ratio, unless $P = NP$. The greedy algorithm is very simple, namely on each time step schedule the element that hits the most uncovered sets. What is interesting is the analysis of the algorithm. The authors use a clever pricing technique along with a histogram argument.

Since then, many applications and variants of Min-Sum Set Cover have been proposed. Azar et. al [12] introduced the *Multiple Intents Re-Ranking* problem, motivated by applications in web search ranking based on search intents of different users. Each user is modeled by a subset of search results, relevant to its own preferences and a particular profile weighted vector over the elements of given subset, that models the user's intents of searching. The user scans the results from top to bottom, paying an overhead that depends on the position of the results in user subset. The goal is to provide a linear order of search results, so that the sum of total weighted cover time of sets is minimized. The authors note that in case where all profile vectors have the form $< 1, 0, \ldots, 0 >$, the problem is equivalent to Min-Sum Set Cover. These users are *navigational*, meaning that they are interested in the first search result that is relevant to their preferences. Another model that meets applications in web ranking is *Submodular Ranking*, studied by Azar and Gamzu [11], where instead of sets, there are non-negative monotone submodular functions that are 'covered' when they 'reach' value 1. The work in [12] assumes that the user sets are taken from user log files that are provided offline. This constructs the basic motivation of this thesis: What if the user sets arrive online? This idea captures a real scenario when web search results need to be rearranged online as new sets arrive. This is the concept behind Online Min-Sum Set Cover.

List Accessing

In Online Min-Sum Set Cover, we are interested in designing an algorithm that performs rearrangements in a list of elements in order to reduce the access costs (time overhead) incurred by future set requests. However, an algorithm that performs rearrangements needs to pay a cost for such element moves, as well. The goal thus becomes to minimize total sum of rearrangement costs and access costs incurred by the arriving sequence of sets. An algorithm is given the freedom to move elements from the set, in the rearrangement process, as long as it pays the necessary cost to access them.

In the above scenario, we are motivated by the idea that web search results are scanned from top to bottom. For this reason, we can imagine this super set of results to be organized in a list data structure. The list data structure has the property that it can only be accessed sequentially from its head. Thus, the Online Min-Sum Set Cover can be represented by a list of elements, for which set requests arrive online. A simpler scenario of the above is the famous *List Accessing problem*, one of the most well-studied problems in online literature. In this problem, a list of elements is given and requests of single elements arrive in online manner. The goal is to perform suitable rearrangements as data arrive in order to reduce future access costs. This is a simple problem scenario that motivates self-organizing data structures, i.e. design algorithms that maintain an 'efficient', according to accesses, data structure.

The problem was first studied under competitive analysis by Sleator and Tarjan [54]. Three natural heuristics for List Accessing are *Transpose*, the requested element is transposed with the element that is one position prior to it, *Frequency Count*, the elements are kept in decreasing order of their frequencies and *Move-To-Front (MTF)*, the requested element is moved to the front of the list. The first two of them are proved to be $\Theta(l)$-competitive. In contrast, *MTF* was proved to be 2-competitive by Sleator and Tarjan by deploying a potential function argument. The potential function method is a tool of amortized analysis, introduced by Tarjan [56] as a framework to measure the impact of each action or operation over the whole sequence of operations. What is interesting with *MTF* is that it is tight to the existing lower bound for deterministic algorithms, thus it is optimal in the deterministic case. This lower bound is proved by using an averaging technique [21], namely the optimal offline cost is bounded by the average cost of a known set of offline algorithms. In case of randomized algorithms, the best known randomized algorithm is due to Albers et. al [8] and achieves 1.6-competitive ratio. The best known lower bound for randomized algorithms is 1.5 and was proved by Teia [57].

Thesis Purpose

The goal of this thesis is to introduce the Online Min-Sum Set Cover and motivate further research work. We provide some results on both deterministic and randomized case. Most of our work is motivated by the work conducted in List Accessing. We prove a lower bound on the competitive ratio of deterministic online algorithms equal to $A+1-\frac{A(A+1)}{l+1}$, where A is the average set cardinality of request sequence and l is the list length, by deploying the averaging technique and comparing the total cost of optimal offline solution to the average cost of static offline algorithms. Fine tuning on parameter A gives a lower bound of $\Omega(l/4)$ for any deterministic algorithm that performs for all values of A. We propose three MTF-like algorithms: $MoveFront$, $MoveLast$ and $MoveSet$ and prove their competitive ratios. $MoveFront$ is shown to be tight $l-A+1$-competitive, while $MoveLast$ and $MoveSet$ are tight l-competitive. We construct proper adversarial request sequences that always incur worst-case costs, while optimal offline solution pays only a small cost for it. Finally, we show that two proposed algorithms, *Randomized Static* and *Randomized Move-To-Front* do not provide sublinear guarantees for their competitive ratios. A randomized lower bound is left as future work. These results are far enough from the proved deterministic lower bound, taking into account that any algorithm is at least l-competitive. All the above algorithms are *memoryless*, i.e. their decisions are based only on the current requested set and its elements' position in the list. We thus conclude that such memoryless policies do not help in designing competitive algorithms close to the proved lower bound and provide motivation for future work.

Chapters Overview

In Chapter 3, we make a brief introduction to *Online Computation*. We present the notion of online problems and algorithms along with the basic measure of their performance, competitive analysis. We discuss the use of randomization in online algorithms and how it can affect their performance against different types of adversaries. We also provide some famous online problems and their applications.

In Chapter 4, we present the offline *Min-Sum Set Cover*. We focus primarily on the proof of 4-approximate greedy algorithm as presented by Feige et. al [27]. We also enclose a bibliographic report on Min-Sum variants and their applications.

In Chapter 5, we discuss the *List Accessing Problem*. In particular, we present the averaging technique used for proving the deterministic lower bound of $2-\frac{2}{l+1}$. Then, we discuss competitiveness of algorithms $TRANS$ and FC, before proceeding with an analytic proof that MTF is strictly 2-competitive. Prior to this, we make a brief reference on amortized analysis, introduced by Tarjan [56] and discuss the potential function method in the setting of online algorithms. Finally, we make a comprehensive presentation of List Accessing through results, proposed algorithms and variants over the years.

In Chapter 6, we provide a formal definition of the *Online Min-Sum Set Cover Problem* and discuss its detected relations to Min-Sum Set Cover and List Accessing. We then present our results. First, we present a deterministic lower bound for the problem. Second, we present algorithms $MoveFront$, $MoveLast$ and $MoveFront$, motivated by algorithm *Move-To-Front* in List Accessing and prove their competitive ratios. Then, we discuss on the competitiveness of two simple randomized algorithms *Randomized Static* and *Randomized Move-To-Front*. We finally take some space to draw some conclusions on the current results.

Chapter 3

Online Computation

In computer science, a traditional framework for algorithm design is the following: Given an input I for a problem P, design an algorithm that produces an output $O(I)$ that satisfies the goal and restrictions defined by P. However, in many real applications, the entire input may not be given from the beginning, but rather may be revealed gradually. In this setting, an algorithm has to take an irrevocable decision on every incoming piece of input without knowledge of the future, based only on the partial sequence of input pieces revealed up to the current point of time. Such algorithms that must perform under uncertainty of partial input knowledge are called *online algorithms* and the problems they deal with, *online problems*.

In this chapter we make a brief introduction in the theoretical framework of online algorithms and competitive analysis. We also present some historic problems in the field, that help in the understanding of online algorithms and their significant presence in many real world applications.

3.1 Online Algorithms and Competitive Analysis

An *online algorithm* receives the input as a sequence of requests $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(n)$. Every request must be served by the algorithm in order of occurrence and at the time of arrival. When serving request $\sigma(t)$, the online algorithm has knowledge of requests $\sigma(t')$, for $t' \leq t$, but has no knowledge of requests $\sigma(t')$, for $t' > t$. Also, the size n of the request sequence may not be known in advance. Serving each request incurs a cost or profit. Depending on the problem, the goal is to minimize the total cost or maximize the total profit incurred by the entire input sequence.

It becomes obvious that this incomplete image of the input instance along with the irrevocable decisions on every request may not allow the online algorithm to reach the optimum value at the end of execution. A basic question arises naturally: How can we measure the performance of an online algorithm? The most well-known performance measure for analyzing online algorithms is *Competitive Analysis*, a term that was first coined by Karlin et al. [37] and introduced by Sleator and Tarjan [54]. In Competitive Analysis, the output of an online algorithm is compared to the output of the *optimal offline algorithm*. This is the algorithm that has knowledge of the entire input sequence from the beginning of its execution and performs optimally on that sequence. Competitive Analysis makes no assumptions on the statistical distribution of input data. It is a type of *Worst-Case Analysis* in the sense that we judge an algorithm only by its performance on the worst-case input, i.e. the input that brings the greatest imbalance between the outputs of online and optimal offline algorithm respectively. This imbalance is formulated by *Competitive Ratio*. More specifically, we introduce the following definitions:

Definition 3.1. *Given a request sequence σ, let $ALG(\sigma)$ and $OPT(\sigma)$ denote the costs of online and optimal offline algorithm, respectively. The online algorithm is called* ***c-competitive*** *if there exists constant a such that for every request sequence σ:*

- $ALG(\sigma) \leq c \cdot OPT(\sigma) + a$, *in a minimization problem*
- $ALG(\sigma) \geq \frac{1}{c} \cdot OPT(\sigma) - a$, *in a maximization problem*

If $a \leq 0$, the algorithm is called ***strictly c-competitive****.*

Definition 3.2. *The infimum over all values c, such that the online algorithm is c-competitive, is called* ***competitive ratio*** *of the online algorithm and is denoted by $R(ALG)$.*

The value of c can be a function of problem parameters, but must be independent of online input parameters, for example the size of the request sequence.

We can see that the competitive ratio for online algorithms is an extension of approximation ratio for offline algorithms. In fact, a *strictly c-competitive* algorithm is also a *c-approximate* algorithm for the offline problem, but with partial knowledge of input.

3.2 The Power of Randomization

Competitive Analysis introduces an alternative point of view for online algorithms, that of a request-answer game between an *online player* and an *adversary* [18]. The online player uses the online algorithm to respond on every request created by the adversary. The adversary's role is to produce the worst-case request sequence that maximizes the competitive ratio.

An online algorithm can be either *deterministic*, i.e. on identical request sequences it will have the same response on every request, or *randomized*, i.e. its decisions are random results from a probability distribution. In case of a deterministic algorithm, the adversary knows the online algorithm, we can imagine it reading the algorithm's code, so it can know the exact response of the online player on every request. Thus, it is able to produce the entire worst-case input in advance. The adversary and the optimal offline algorithm are often referred as the *offline player* or *oblivious adversary.*

By deploying randomization, an online algorithm is able to reduce the competitive ratio in comparison to acting only in deterministic case. This happens because part of the algorithm's actions are now concealed under uncertainty. The adversary has knowledge of the algorithm's description and the probability distribution, but cannot be sure of the exact actions of the algorithm because they are randomized. Thus, the worst-case input sequence is not one and only and depends on the algorithm's random choices.

Based on the adversary's knowledge for the online decisions and its ability to exploit them, a distinction can be made on the adversary models towards which the online player competes. As mentioned before, every adversary model knows the online algorithm and the probability distribution used. Also, the competitive ratio needs to be redefined for the randomized case as the ratio of expected online cost to ‘adversary cost’. In general, we have the following:

Definition 3.3. *A randomized online algorithm ALG is called* ***c-competitive*** *against adversary ADV if there exists constant a, such that for every request sequence σ:*

$$\mathbb{E}[ALG(\sigma) - c \cdot ADV(\sigma)] \leq a$$

where $\mathbb{E}$ is the expected cost of ALG taken over the random choices it makes. For a maximization problem, the definition is altered analogously.

The ***expected competitive ratio*** *of ALG against adversary ADV is defined as the infimum over all values c, such that the online algorithm is c-competitive and is denoted by $\bar{R}_{ADV}(ALG)$.*

The three adversary models, presented in [18], are described below:

- ***Oblivious Adversary (OBL)****:* Constructs the request sequence in advance and pays the optimal offline cost.
- ***Adaptive Online Adversary (ADON)****:* Constructs the request sequence in online fashion: serves the current request before the online player, then generates the next request based on the online algorithm’s previous actions.
- ***Adaptive Offline Adversary (ADOF)****:* Constructs the request sequence in online fashion: generates the next request based on the online algorithm’s previous actions, but pays the optimal offline cost for the entire generated request sequence. Randomization cannot help against this adversary.

Both $OBL(\sigma)$ and $ADOF(\sigma)$ are the optimal offline cost $OPT(\sigma)$. $ADOF(\sigma)$ and $ADON(\sigma)$ are random variables, as σ is a random variable whose construction depends on the random choices of ALG. Since OBL constructs the sequence in advance, it is not dependent of the random choices of ALG, thus definition of c-competitiveness for OBL can be simplified to $\mathbb{E}[ALG(\sigma)] - c \cdot OPT(\sigma) \leq a$. The adversaries above were sorted by their power. That is what the next theorem says:

Theorem 3.1. *Given a problem and a randomized online algorithm, it holds that $\bar{R}_{OBL}(ALG) \leq \bar{R}_{ADON}(ALG) \leq \bar{R}_{ADOF}(ALG)$*

Also, in [18] the following two theorems are proved:

Theorem 3.2. *If there is a randomized algorithm that is c-competitive against any adaptive offline adversary, then there also exists a c-competitive deterministic algorithm.*

Theorem 3.3. *If A is a c-competitive randomized algorithm against any adaptive online adversary, and there is a randomized d-competitive algorithm against any oblivious adversary, then A is a randomized $(c \cdot d)$-competitive algorithm against any adaptive offline adversary.*

3.3 Examples of Online Algorithms

Online algorithms provide a useful framework for problems that deal with an input arriving in pieces and the response needs to be immediate. Such problems occur naturally in the fields of interactive computing, data structures, network applications, motion planning, scheduling, resource management and many more. We present some famous online problems, as presented in [21] [5] [35] [48].

3.3.1 A Warmup: The Ski Rental Problem

The *Ski Rental Problem* is a toy example that helps in understanding the basic concepts of online computation. It also provides a general study framework for problems that involve decisions between paying a small repeating cost per time unit *(rent)* or switching to paying a larger one-time cost *(buy)* with no further payment. This cost tradeoff, the *rent/buy problem* as it is called, find applications in real problems such as snoopy caching, TCP acknowledgement and scheduling.

The problem can be modeled under the following simple scenario: A skier is going for ski for d days in total. Each day he has two options: Rent the ski equipment for today with cost R dollars or buy the ski equipment and use it for the rest of the days with a cost of $B > R$ dollars. In an offline problem the answer is easy, if $dR < B$ then rent every day, else buy the equipment from the first day. However, in the online setting, d is not known in advance, for example the ski resort may close unexpectedly.

So, the skier must follow a strategy of the form 'rent for a days, then buy', paying a total cost of $B + aR$. However, for every choice of a, the skier may have made a very bad decision, when the d days finally pass. For example, he could have decided to buy on day i and on day $i+1$ the ski resort would close without knowing it prior to his decision. In that case, it would be best for him to have rented on day i or to have bought some days before i. Such scenarios describe the optimal offline solution. So, can he predict such scenarios? The answer is no. What he can do however is to minimize the total cost of a decision, that in the end, may prove to be the worst among all other decisions he could have made. This is the concept of competitive analysis.

The ratio of online to offline cost is $\frac{B+aR}{\min(B,dR)}$. We are interested in finding a to minimize the maximum value of this ratio, i.e. the ratio on the worst-case scenario. Obviously, $a < d$, so the maximum value is $\frac{B+aR}{\min(B,(a+1)R)}$. This is the competitive ratio and describes the aforementioned worst-case scenario: skier buys on day $a+1$, which is the unexpected last day. The ratio is minimized when $B = (a+1)R \rightarrow a = \frac{B}{R} - 1$, giving a value of $2 - \frac{R}{B}$ and subsequently a strictly 2-competitive strategy. Thus, the skier's optimal strategy in terms of competitive analysis is to rent until the day when renting again incurs a total cost that exceeds the cost of having bought from the first day.

The deterministic 2-competitive ratio was proved by Karlin et al. in [37]. Also, in [36], a randomized algorithm was proposed that achieves a competitive ratio of $\frac{e}{e-1}$ against an oblivious adversary. Day i is chosen as the day of buying with probability $p_i = (\frac{b-1}{b})^{b-i}\frac{1}{b[1-(1-\frac{1}{b})^b]}$, for $i \leq b$, where the buying cost equals b and the renting cost equals 1.

3.3.2 The Paging Problem

The *Paging Problem*, one of the first and most well-studied problems in online literature, was first motivated by computer architecture and operating systems. A two-level memory system is given, consisting of a large slow memory (e.g. a hard disk) and a small fast memory (e.g. RAM). Each level stores a number of fixed-size memory units called *pages*, let N pages for slow memory and k pages for fast memory. A request sequence of pages is given in online fashion. If the requested page is in fast memory, it is served immediately and if not, a *page fault* occurs. In that case, the requested page needs to be loaded from slow memory into fast memory, resulting in the eviction of a page from the fast memory. The online paging algorithm must design an eviction strategy such that the number of page faults is minimized. Different algorithms had been studied extensively under specific distribution of the input sequence. Sleator and Tarjan were the first to study paging under competitive analysis [54].

Contrary to most online problems, the optimal offline algorithm for paging is known, which is proved to be helpful in the analysis of online paging algorithms. Belady [17] proved that the algorithm of evicting on a fault the page whose next request occurs furthest in the future is the optimal offline algorithm and was called MIN. Sleator and Tarjan [54] proved a deterministic lower bound of k. They also proved that LRU, namely evicting on a fault the page that was requested least recently and $FIFO$, i.e. evicting on a fault the page that has been in fast memory longest, are k-competitive. These two algorithms are part of a general class of algorithms called *marking algorithms*, that introduce the technique of *phase partitioning*. For marking algorithms, the request sequence is partitioned in phases according to the following. In the start of each phase, all pages in the memory system are *unmarked*. When a page is requested, it is *marked*. On a fault, only unmarked pages can be evicted. The phase ends when all pages in fast memory are marked and a page fault occurs. Then, all marks are erased and a new phase begins. Later, Torng [58] showed that any marking algorithm is k-competitive.

In case of randomization, Raghavan and Snir [49] proved that no randomized algorithm can do better that k-competitiveness against an adaptive online adversary. Fiat et al. [29] proved a lower bound of H_k (the kth Harmonic number) against oblivious adversaries and proposed a *randomized marking algorithm* that is $2H_k$-competitive. In particular, on fault, a page is chosen uniformly at random from the set of unmarked pages in the fast memory and is evicted. Finally, optimal H_k-competitiveness was proved for algorithms proposed by McGeoch and Sleator [44] and later by Achlioptas et al. [1].

3.3.3 The k-server Problem

In the *k-server Problem*, a metric space S and k mobile servers, represented as points in S, are given as standard input. A request sequence is provided in online fashion, where each request is also a point in S. Each time a request arrives, the online algorithm must move a server to the requested point, unless there is already one there. When a server is moved from point x to point y, it incurs a cost of d_{xy}, i.e. the distance between x and y. The goal is to minimize the total distance covered by all servers for the entire request sequence. The problem draws a lot of attention because it abstracts a large number of problems such as paging, caching, motion planning and more. It has also been a living field of applying novel techniques in online computation.

The problem was introduced by Manasse and McGeoch in [41]. The authors proved a lower bound of k for any deterministic algorithm in arbitrary metric space and they posed the famous *k-server conjecture*, according to which there exists a deterministic algorithm that is k-competitive. The conjecture was proved for special cases (tree metrics, resistive spaces, special values of k), before Koutsoupias and Papadimitriou [38] prove that the *Work Function Algorithm*, a general technique for online problems, is $(2k-1)$-competitive in the general case, the closest result to the conjecture so far. *Work function* $w(X)$ attempts to follow the optimal offline solution and represents the minimal cost of serving request sequence σ and ending in the configuration of servers X. When a new point $\sigma(t) = r$ arrives and the current configuration of servers is X, the algorithm will move that server s_i, located in current point x_i, which minimizes $w(X_i) + d_{x_i r}$, where $X_i = X - \{x_i\} + \{r\}$. As of today, the conjecture remains open.

In case of randomized algorithms, a lower bound of $\Omega(\frac{\log k}{\log^2 \log k})$ was proved for arbitrary metric spaces against an oblivious adversary by Bartal et al. [16]. The *randomized k-server conjecture* states that there exists a randomized $\Theta(\log k)$-competitive algorithm against an oblivious adversary. In 2017, Lee [39] proved a $O(\log^6 k)$-competitive randomized algorithm for any metric space.

Chapter 4

The Min-Sum Set Cover Problem

The *Min-Sum Set Cover Problem* was introduced by Feige, Lovász, Tetali [27][28]. It can be considered as a version of *Set Cover Problem* with latency. In every time step, exactly one set of elements over a collection of sets is chosen. In that way, every element is covered for the first time at a particular time step. The goal is to schedule the sets so that the sum of first time steps over all elements is minimum. It is a general scheduling problem that motivates applications from the fields of distributed resource allocation, web search ranking, query processing and others. Also, it introduces a general framework for many other problems. As mentioned in [32], Min-Sum Set Cover and its variants are related to all problems that involve multiple demands under the objective of overall minimum latency. Feige, Lovász, Tetali [27][28] provided a simple greedy algorithm that achieves a 4-approximation ratio. Moreover, no algorithm for the general instance can achieve a better ratio, unless $P = NP$, thus the algorithm is tight.

In this chapter, we formulate the problem and emphasize on the analysis of the 4-approximate algorithm. We also provide a short reference on related problems and their applications.

4.1 Problem Definition

4.1.1 Set Representation

In the *Min-Sum Set Cover* (*MSSC*) we are given as input a collection of sets $S = \{S_1, S_2, \dots, S_n\}$, whose union equals the universe of elements $E = \{e_1, e_2, \dots, e_m\}$. The objective is to schedule the sets, one at a time, such that the total cover time of the elements is minimized. More formally, given a permutation of sets $\pi \colon [n] \to [n]$, we define the *cover time of element* e_j as the earliest time step i at which $e_j \in \pi(i)$, i.e.

$$f(e_j) = \min_{i \in [n] : e_j \in S_i} \pi^{-1}(i)$$

The goal is to find a permutation $\pi^* \colon [n] \to [n]$ such that:

$$\pi^* = \arg\min_{\pi} \sum_{e \in E} f(e)$$

By $\pi(i) = j$ we mean that the ith left-most set in permutation is S_j. From the problem definition, it becomes clear that every element induces an amount of latency, the number of time steps it takes to be covered. We want to find a linear order of the sets in order to cover all elements "as soon as possible", i.e. minimizing the total latency induced by elements. Equivalently, the goal is to minimize the average cover time of elements, since the total sum of cover times is minimized in that case and vice versa.

4.1.2 Hypergraph Representation

An equivalent representation is that of a *Min-Sum Vertex Cover* in hypergraphs. The hypergraph representation is equivalent to set representation for MSSC, just like the *Hitting Set Problem* to the *Set Cover Problem*. Now, the permutations are over the vertex set of a hypergraph. Given a hypergraph $H(V, E)$ with vertex set $V = \{v_1, v_2, \ldots, v_n\}$, hyperedge set $E = \{e_1, e_2, \ldots, e_m\}$ and a permutation $\pi\colon [n] \rightarrow [n]$ we define the *cover time of hyperedge* e_j as the earliest time step i for which $\pi(i) \in e_j$, i.e.

$$f(e_j) = \min_{i \in [n] : v_i \in e_j} \pi^{-1}(i)$$

The goal is to find a permutation $\pi^*\colon [n] \rightarrow [n]$ such that:

$$\pi^* = \arg\min_{\pi} \sum_{e \in E} f(e)$$

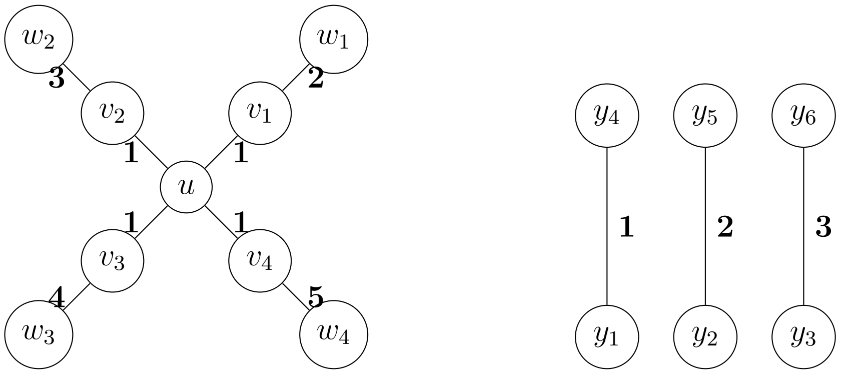
This representation seems easier to understand, since the ordering objects are single entities, i.e. vertices, rather than collections of elements. For the rest of the thesis, we make use of this representation. For simplicity, we use the notation *sets* instead of *hyperedges* and *elements* instead of *vertices*. Thus, we are searching for the optimal linear ordering of elements that covers sets. Finally, in the following, we are free to omit from the output permutation those last elements that, when scheduled, all sets have already been covered.

4.1.3 Differences with Set Cover

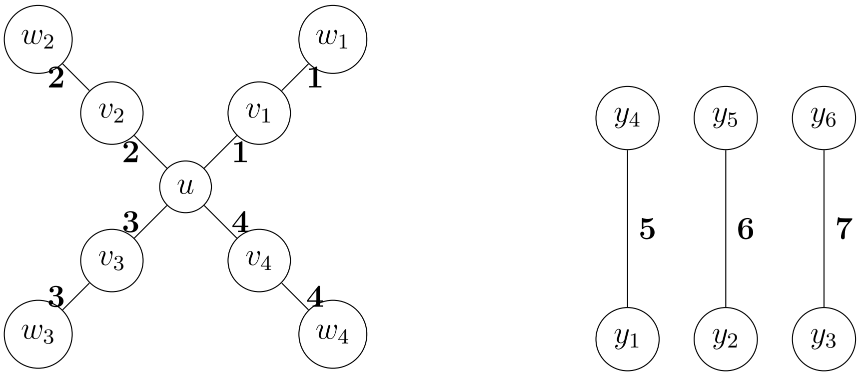
The MSSC problem is NP-Hard. Apart from inherent similarities with the Set Cover problem, the results and techniques used both in MSSC and its variants reveal a quite different problem that needs different approach than Set Cover.

For instance, MSSC does not hold the property that the optimal solution is a combination of the optimal solutions of disjoint sub-instances. A simple example provided in [27] can be seen in Fig.4.1. Consider graph G, comprising of graphs G_1 and G_2. The MSSC instances for graphs G_1 and G_2 independently give optimal solutions (u, v_1, v_2, v_3, v_4) with a total of 18 (Fig.4.1a) and (y_1, y_2, y_3) with a total of 6 (Fig.4.1b) respectively, while the MSSC instance for graph G gives the optimal solution $(v_1, v_2, v_3, v_4, y_1, y_2, y_3)$ with a total of 38 (Fig.4.1c). As it can be seen, vertices of G_1 are scheduled differently in the G instance. In G_1 instance, u is responsible for covering first edges $(u, v_1), (u, v_2), (u, v_3), (u, v_4)$, while in G instance u has no covering impact (and it is omitted from output). If vertices were scheduled just like in G_1 and G_2 instances, one after the other, the total cover time would be 39.

Another interesting property is that no polynomial time algorithm is known for simple graph instances such as trees, in contrast to the Vertex Cover problem. The authors in [27] detect different properties among the hardest instances of MSSC and Set Cover.



(a) G_1 (b) G_2



(c) G

Figure 4.1: Optimal solution in G is not a combination of optimal solutions in G_1 and G_2

4.2 The Greedy Algorithm

As mentioned before, MSSC is NP-Hard. Feige, Lovász, Tetali [28] [27] provided an algorithm that achieves 4-approximation ratio, also proving that this algorithm is tight. Their algorithm follows a very simple greedy rule, namely at each time step schedule the element that covers the largest number of uncovered sets.

Algorithm 1 Greedy Algorithm

```
Input: Elements $E$, Sets $S$ jointly covering $E$
Output: Linear order of elements $E$
Initialize $i = 1$
while $S \neq \varnothing$ do
    Select $e_i \in E$ to be the element that covers the largest number of sets in $S$
    $E = E \setminus \{e_i\}$, $S = S \setminus \bigcup_{S_j \ni e_i} S_j$
    $i = i + 1$
end while
```

Two main results hold:

Theorem 4.1. *1. The greedy algorithm approximates Min-Sum Set Cover within ratio of* 4.

2. *It is NP-Hard to approximate Min-Sum Set Cover within ratio of* $4-\epsilon$*, for every* $\epsilon > 0$.

The proof of *(2)* is based on a modifying reduction from *Max-3SAT-5* to *Max-k-Coverage* and is not presented here, as it goes beyond the purposes of this thesis.

In the following, we present the proof of *(1)* as shown in [27]. Each set is priced with a particular value according to the linear ordering produced by *greedy* and then a clever histogram argument is used. This proof is a simplification of the proof in the conference version of this paper [28]. The original proof is based on a primal-dual approach. MSSC is formulated as an integer program and then relaxed to a linear program. The value of the *dual* program is a lower bound for *opt*, for every feasible assignment of dual variables. The authors prove that $greedy \leq 4dual$ through a specific assignment of dual variables based on the output of greedy algorithm. The reader is prompted to study the proof for a better understanding of the idea behind the pricing and histogram argument. We proceed with the simplified proof.

Proof. At each time step i, the greedy algorithm picks an element from E and places it in the ith position at the linear ordering. For every $1 \leq i \leq n$ let:

$$X_i = \{s \in S \mid \textit{first covered in time step } i \textit{ by greedy}\}$$

$$R_i = S \setminus \bigcup_{j=1}^{i-1} X_j = \{s \in S \mid \textit{not covered prior to time step } i \textit{ by greedy}\}$$

$$P_i = \frac{|R_i|}{|X_i|}$$

$$p_s = P_i, \text{ for every } s \in X_i$$

Also, let *greedy*, *opt* be the values of the respective solutions and $price = \sum_{s\in S} p_s$. It is easy to prove the following:

$$greedy = \sum_{i=1}^{n} i|X_i| = \sum_{i=1}^{n} |R_i| \tag{4.1}$$

$$price = \sum_{s\in S} p_s = \sum_{i=1}^{n} |X_i| P_i = \sum_{i=1}^{n} |X_i| \frac{|R_i|}{|X_i|} = \sum_{i=1}^{n} |R_i| = greedy \tag{4.2}$$

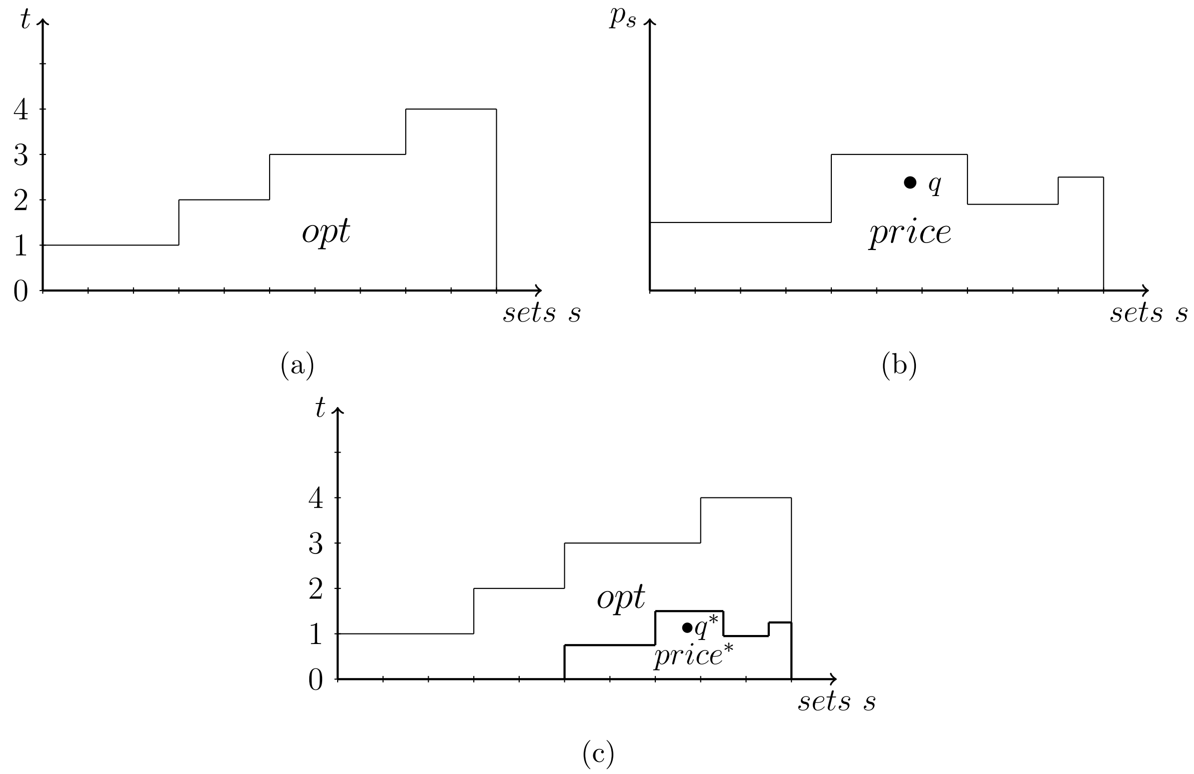
An intuition for (4.1) is the following, the contribution of every set to *greedy* value can be measured by two ways. Either each set increases the value of *greedy* by i units, when scheduled at time step i, or by one unit for every time step at which it remains uncovered (total i time steps). Now, charging p_s on every set $s \in X_i$ is a third way of measuring this contribution (4.2): at time step i, *greedy* is increased by $|R_i|$ units and $|X_i|$ sets are covered, so sets in X_i are selected to be charged this increase uniformly. Most important, the sum of these prices remains equal to the total value of *greedy*. This alternative pricing of each set's contribution will help in the proof.

From (4.1), (4.2) it suffices to show that $opt \geq price/4$.

The analysis is based on the histograms described below. The key idea is to draw two histograms with total areas the price of *opt* and *price* respectively and then prove that, by shrinking the area of the *price*-histogram by a factor of 4, the area of the shrunk histogram is not larger than that of *opt*-histogram.

In *opt*-histogram (Fig.4.2a), sets are placed on *x-axis* in the order that they were covered by *opt* and each one has width 1. *y-axis* shows the time step at which every set was covered. For that reason, the heights of the $|S|$ columns are non-decreasing integer values. Obviously, the area underneath the histogram equals the value of *opt*.

In *price*-histogram (Fig.4.2b), sets are placed on *x-axis* in the order that they were covered by *greedy* and each one has width 1. *y-axis* shows the value p_s of each set $s \in S$, as defined by the greedy process. The heights of the $|S|$ columns can be positive non-monotone rational numbers. Also, $area = \sum_{i=1}^{n} |X_i|P_i = \sum_{s \in S} p_s$, thus the area underneath the histogram equals the value of *price*



(a)

(b)

(c)

Figure 4.2: Histograms of *opt*, *price*, $price^*$

For the proof, the *price*-histogram is under-scaled on both axes by a factor of 2, thus leading to a new $price^*$-histogram with area equal to $price/4$. $Price^*$-histogram is aligned to the right of *opt*-histogram, thus its columns lie on the interval $[|S|/2+1, |S|]$ of *x-axis* (Fig. 4.2c). To show that the area of $price^*$ is smaller than the area of *opt*, it suffices to prove that the $price^*$-histogram fits completely within *opt*-histogram. This means that by picking any point q in *price*-histogram, the projected point q^* in the right-aligned $price^*$-histogram must lie within *opt*.

Let q belong to set s covered at time step i by *greedy*. Let h, h^*, r, r^* be the height and right hand side distance of these points respectively. Then:

$$h \leq p_s = \frac{|R_i|}{|X_i|} \;\rightarrow\; h^* \leq \frac{p_s}{2} = \frac{|R_i|}{2|X_i|} \tag{4.3}$$

$$d \leq |R_i| \;\rightarrow\; d^* \leq \frac{|R_i|}{2} \tag{4.4}$$

Now, what condition must hold for q^* to lie within *opt*-histogram? Since column heights in *opt* are non-decreasing and q^* has height h^*, q^* must be located somewhere inside the region of columns with heights $\lceil h^* \rceil$ or greater. Thus, the boundary at the start of this region must be at the left of q^*, i.e. it must have right hand side distance at least $\lceil d^* \rceil$. In the MSSC notation, this means that exactly before time step $\lceil h^* \rceil$, at least $\lceil d^* \rceil$ sets must have not be covered by *opt* yet.

Now, the greedy solution makes its appearance. The greedy algorithm picked the element at time step i that covers the largest number of elements from $|R_i|$, i.e. $|X_i|$. Thus, in $\lfloor h^* \rfloor$ time steps (remember, 'exactly' before time step $\lceil h^* \rceil$) *opt* could have covered from R_i at most $\lfloor h^* \rfloor |X_i| \overset{(4.3)}{\leq} \lfloor \frac{|R_i|}{2|X_i|} \rfloor |X_i| \leq \lfloor \frac{|R_i|}{2} \rfloor$ sets, leaving at least $\lceil \frac{|R_i|}{2} \rceil \overset{(4.4)}{\geq} \lceil d^* \rceil$ sets from R_i uncovered, hence the result. Thus, q^* lies within *opt*-histogram and the proof is complete. □

4.3 Min-Sum Variants

The Min-Sum framework appears in many different contexts. Generally, this setting finds many applications in web page ranking, distributed resource allocation problems, data base query processing, peer to peer networks and many more. The common objective for minimization is the overall (or average) latency. We make a brief presentation of some well-studied Min-Sum versions.

Min-Sum Set Cover has been studied under *precedence constraints* [43], i.e. the output permutation must satisfy a feasible set of constraints $e_i \prec e_j$, meaning that e_i must precede e_j ($\pi^{-1}(i) < \pi^{-1}(j)$). The problem meets applications in software test case prioritization, when the test suite constructed for fault detection needs to be scheduled under dependency constraints between test cases. Along with other results, the authors describe a greedy algorithm that is within $4\sqrt{|E|}$-approximation ratio and prove that there is no poly-time algorithm that approximates the problem within ratio $O(|E|^{\frac{1}{12}-\epsilon})$, for $\epsilon > 0$. Ideas such as histogram analysis and a greedy approach similar to that for MSSC are used.

Min-Sum Vertex Cover (*MSVC*) is a special case of Min-Sum Set Cover, also studied in [28][27]. In hypergaph representation, the hypergraph is a graph $G(V, E)$. The goal is to find a linear ordering of vertices V that minimizes the total cover time of the edges E. MSVC is used as heuristic in solving semidefinite programs faster. It is proved that the greedy algorithm used for MSSC cannot approximate MSVC within a ratio better than 4. Instead, formulating MSVC as an integer program and using proper randomized rounding for its linear relaxation proves to achieve an approximation ratio of 2. Finally, it is proved that there exists a constant $\rho > 1$ such that it is NP-Hard to approximate MSVC within ratio better that ρ.

Min-Sum Coloring (*MCS*) was studied extensively prior to MSSC and motivated its study. [15]. The objective is to find a vertex coloring in a given graph G such that the sum of color numbers assigned to vertices is minimized. The problem can model distributed resource allocation problems that impose resource conflicts among computational nodes, i.e. they cannot execute their tasks simultaneously. Conflicts among tasks can be modeled as edges connecting vertices in a *conflict graph*. The goal is to minimize the average time of task response. MCS is NP-Hard. The authors prove that it is NP-Hard to approximate MCS within a factor of $n^{1-\epsilon}$, for any $\epsilon > 0$. They also show that the greedy algorithm of finding iteratively a maximum independent set gives a 4-approximation solution that is lower bounded by 2.

In *Generalized Min-Sum Set Cover* (*GMSSC*), the cover time of set S_i is defined as the earliest time step at which at least k_i elements from S_i have been scheduled. Again, the goal is to minimize the total cover time. Hence, MSSC is a special case of GMSSC when $k_i = 1$, for every $i \in [S]$. GMSSC meets applications in web page ranking, where the goal is for a search engine to re-rank web search results, based on user query logs, in order to minimize average user effort in finding the web pages that satisfy their preferences. The problem was first introduced as *Multiple Intents Re-Ranking* in [12]. The authors made use of a shrunk histogram argument similar to that for MSSC to prove a greedy $O(\log \max_i k_i)$ approximation. Later, Bansal et al. [14] proved a constant 485-approximation algorithm, using a linear program relaxation strengthened with knapsack cover constraints and a randomized rounding scheme proceeding in stages. In [53] the approximation was improved to around 28, by modifying the previous rounding process using concepts from α-point scheduling. In [33], by using a different linear program and a modified α-rounding scheme, the approximation was further improved to 12.4. Proving a 4-approximation algorithm for GMSSC remains an open problem.

Submodular Ranking (*SR*) is a more general problem that includes GMSSC as a special case. A non-negative monotone submodular function $f_i : 2^{|E|} \to [0, 1]$ with $f_i(E) = 1$ is given, instead of each set S_i. The cover time of f_i is defined as the earliest time step t at which $f_i(\{e_{\pi(1)}, e_{\pi(2)}, \ldots, e_{\pi(t)}\}) = 1$. SR applies to mobile network broadcasting and web search ranking, where the submodular function models the information that every receiver/user gains from any subset of transmitting data segments/search results. Submodularity is compatible with the idea that pieces of information needed for each agent to complete its goal need not be disjoint. Azar and Gamzu [11] prove a greedy $O(\log(1/\epsilon)$-approximation algorithm, where ϵ is the minimum marginal positive increase of any function f_i. Histogram analysis is used in the proof. They also prove NP-Hardness in approximating the problem within ratio of $c \ln(1/\epsilon)$, for some $c > 0$, thus proving the optimality of the algorithm up to constant factors.

Chapter 5

The List Accessing Problem

Suppose we have an unsorted list data structure, that implements the *dictionary* abstract data type, i.e. supporting operations of access, insertion and deletion of an element in the structure. To perform each operation, the list needs to be accessed from its head and searched sequentially, one by one, until the desired position of the element subject is found. Each requested operation takes some time equal to the number of searched elements in the list. One major goal is to maintain an efficient list, i.e. the elements in list are ordered in such way so that requested operations are executed quickly. For example, frequently requested elements must be closer to the head of list. The intriguing point is that requests arrive online. We are thus interested in designing algorithms that reorganize the list as data arrive, in order to reduce future search costs. The goal, as always, is to minimize total search and reorganization costs. The above setting is modeled by the *List Accessing Problem* or *List Update Problem*, one of the most classic and well-studied problems in online literature.

The problem was first studied under competitive analysis by Sleator and Tarjan [54]. It provides a theoretical framework for modeling problems on self-organizing data structures, motivating more efficient data structures such as splay trees [55], while it finds applications in designing efficient data compression algorithms [2] and computing convex hulls [19].

In this chapter, we make a brief introduction in the List Accessing problem and present some basic results on the deterministic case. We focus primarily on techniques and algorithms that motivate our work in Chapter 6. Finally, we present some of the research work that has been done in the field of List Accessing in the past 40 years.

5.1 Problem Definition

Let L be an unsorted list of l elements $x_1, x_2, \ldots, x_l$ and $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ be an online sequence of requests on elements of the list. Each request for an element is associated with an *access cost*, that of the element's position in L. Any algorithm is allowed to reorganize the list by performing transpositions of consecutive elements. For these transpositions the algorithm must pay a *moving cost* according to the following:

- *free transpositions:* Immediately after accessing an element, it can move the requested element to any position closer to the front of the list with no extra cost.
- *paid transpositions:* At any time, it can perform any number of transpositions

between consecutive elements and pay a cost of 1 for each transposition.

The goal is to find an algorithm that minimizes the total cost incurred by σ. More formally, let L_t be the list configuration after the algorithm has processed request σ_t. We define as L_0 the initial list configuration. On the arrival of σ_t, every algorithm pays an access cost equal to the position of σ_t in L_{t-1}, denoted by $L_{t-1}(\sigma_t)$, performs some free transpositions and pays a moving cost $move(L_{t-1}, L_t)$, by using paid transpositions. Then, the goal is to find:

$$\min \sum_{t=1}^{n} [L_{t-1}(\sigma_t) + move(L_{t-1}, L_t)]$$

under the problem constraints defined above.

The above definition formulates the *static list accessing model.* If, apart from accesses, insertions or deletions are permitted in the list, we have the *dynamic list accessing model.* The access cost of every deletion is the element's position in the list and of every insertion of a new element is $l + 1$, where l is the current list length, before insertion. For the rest of the thesis, the static model will be used. Most of the results expand on the dynamic model.

The definition is motivated by the unsorted linked list data structure. In accessing the element in ith position, we traverse the list from the beginning and pay a cost of 1 for comparison with each preceding element. Insertion and deletion costs come naturally, too. Free transpositions are justified by the fact that, having accessed an element, we can keep a pointer at the preferred location along the way and insert the element there at no cost. The definition of paid transpositions is not well-justified. For example, any two consecutive elements are allowed to be transposed but are dismissed from paying a cost for accessing them. We should not forget that list accessing problem is used many times as an abstraction for other problems.

5.2 A Deterministic Lower Bound

In any online problem, proving a lower bound for the competitive ratio of any online algorithm is a strong argument that shows the limits of how well any algorithm can perform for that problem.

One simple method to prove lower bounds is the *averaging technique*, which is used here for proving a lower bound for any deterministic algorithm on the list accessing problem. The technique is based on that, though we do not know the optimal offline cost for an arbitrary request sequence, we can be sure that it will be at most the average cost of a particular known set of offline algorithms whose total cost can be computed easily. The following result is due to Karp and Raghavan, as reported in [34].

Theorem 5.1. *For the static list accessing problem with a list of l elements, any deterministic online algorithm has a competitive ratio of at least $2 - \frac{2}{l+1}$.*

Proof. To maximize the cost incurred by the list accessing, the adversary constructs an input sequence σ that, on every time step, requests the last element of the current list configuration. Remember, on the deterministic case, the adversary knows exactly the actions of the online algorithm, hence it can always request the last element. It is obvious that any worst-case sequence must have the above property. So, the online algorithm pays an access cost of l for each request, thus for a worst-case sequence of arbitrary length n, it will pay a total cost of at least nl (including any paid transpositions).

Now, consider the set of static offline algorithms, i.e. an initial permutation of the list is chosen and remains unchanged by the end of execution. There are $l!$ permutations of the list, each one corresponds to one distinct static offline algorithm. The algorithm pays an initial cost for paid transpositions, in order to configure the initial permutation and then only pays the access cost for each request. The cost for initial paid transpositions is a constant $b = O(l^2)$.

We can find the total cost of these $l!$ static algorithms for the entire request sequence. We first pick a single request and compute the total cost over all static algorithms. For this, we count the permutations in which the requested element appears on the ith position. Considering the element in fixed position i, there are $l-1$ positions in which the rest $l-1$ elements can be placed. Thus, there exist $(l-1)!$ such permutations, that each one of them will incur an access cost of i. So, the sum of access costs for a single request over all permutations is:

$$\sum_{i=1}^{l} i(l-1)! = (l-1)!\frac{l(l+1)}{2} = \frac{(l+1)!}{2}$$

Hence, the sum of total costs for the entire request sequence σ of arbitrary length n over all permutations is at most:

$$n\frac{(l+1)!}{2} + l!b$$

The averaging technique says that there exists a permutation π with total cost at most the average cost of static algorithms. Obviously, the optimal cost will be at most the cost of this static algorithm, i.e.

$$OPT(\sigma) \leq Static_{\pi}(\sigma) \leq \frac{n\frac{(l+1)!}{2} + l!b}{l!} = \frac{1}{2}n(l+1) + b$$

Finally, for any deterministic online algorithm ALG we have:

$$\frac{ALG(\sigma)}{OPT(\sigma)} \geq \frac{nl}{\frac{1}{2}n(l+1) + b} \xrightarrow{n\to\infty} \frac{ALG(\sigma)}{OPT(\sigma)} \geq \frac{l}{\frac{1}{2}(l+1)} \to R(ALG) \geq 2 - \frac{2}{l+1}$$

□

Another simple method to prove lower bounds is by upper bounding the unknown optimal offline cost with the cost of a known offline algorithm that can be computed easier. We present an alternative proof for the above lower bound, based on this technique.

Proof. (Alternative) We use the static offline algorithm A that reorders the list according to the frequency count of elements in the request sequence. For this reordering, the algorithm pays an initial moving cost of $b = O(l^2)$. Let $x_1, x_2, \ldots, x_l$ be the reordered list configuration with frequencies $f_1 \geq f_2 \geq \cdots \geq f_n$, respectively. Then, the offline algorithm will pay a total access cost of $cost_A = \sum_{i=1}^{l} if_i$. Let $cost_{A'} = \sum_{i=1}^{l}(l+1-i)f_i$. It holds that $cost_A \leq cost_{A'}$, because on $cost_A$ we perform in a greedy way and assign the smaller costs to elements with larger frequencies. Alternatively, we can see that $if_i + (l+1-i)f_{l+1-i} \leq (l+1-i)f_i + if_{l+1-i}$, for every $i \leq \frac{l+1}{2}$. Thus, we have:

$$\sum_{i=1}^{l} if_i \leq \sum_{i=1}^{l}(l+1-i)f_i \to 2\sum_{i=1}^{l} if_i \leq \sum_{i=1}^{l}(l+1)f_i = n(l+1) \to cost_A \leq \frac{1}{2}n(l+1)$$

Along with the moving cost, we have proved that there is an offline (static) algorithm with total cost at most $\frac{1}{2}n(l+1) + b$. The rest follows exactly the analysis of the previous proof.

□

5.3 Transpose, Frequency Count

Two basic algorithms that have been proposed for List Accessing are *Transpose* and *Frequency Count*. They use only free transpositions. Prior to competitive analysis, these algorithms were used as natural heuristics for self-organizing lists.

Transpose (TRANS): After accessing an element in position i, transpose it with the element in position $i-1$. If element is in the 1st position, do nothing.

Frequency Count (FC): Keep a frequency counter for every element, initialized to 0. After accessing an element, increment its counter by 1. Then, reorganize the list so that the elements are ordered in nonincreasing order of their frequencies.

For an online algorithm we can prove lower bounds for its competitive ratio by analyzing its performance on a specific input. The competitive ratio, as a worst-case measure, cannot be lower than its value on this specific input.

Theorem 5.2. *Algorithm Transpose has competitive ratio at least $\frac{2l}{3}$, for a list of length l.*

Proof. An adversarial sequence σ could request, on every time step, the last element of the current list configuration, so that $TRANS$ pays a cost of l for each request. Obviously, $TRANS$ transposes the last two elements of the list repetitively. On the other hand, the optimal offline algorithm OPT can move these two elements in the first and second position of the list by paid transpositions, paying an initial moving cost of $(l-1)+(l-2) = 2l-3$ and then paying a cost of 3 on every two requests. Assuming a sequence of arbitrary even length n, the competitive ratio for that sequence will be:

$$\frac{TRANS(\sigma)}{OPT(\sigma)} = \frac{nl}{3\frac{n}{2} + (2l-3)} \xrightarrow{n\to\infty} R(TRANS) \geq \frac{2l}{3}$$

□

Theorem 5.3. *Algorithm Frequency Count has competitive ratio at least $\frac{l+1}{2}$, for a list of length l.*

Proof. Let $x_1, x_2, \dots, x_l$ be the initial list configuration and let $k \geq l$. We construct an adversarial sequence σ of the form $A_1, A_2, \dots, A_l$, where segment A_i requests $k+1-i$ times element x_i. FC will not make any changes in the order of elements, as x_i is requested more times than x_{i+1}. Thus, FC's total cost will be:

$$FC(\sigma) = \sum_{i=1}^{l} i(k+1-i) = \frac{kl(l+1)}{2} + \frac{l(1-l^2)}{3}$$

On the other hand, OPT could pay the access cost of i for the first time that element x_i is requested and then move it to the front of the list, by free transpositions, paying a cost of 1 for the rest $k-i$ requests. Thus, OPT's cost will be:

$$OPT(\sigma) = \sum_{i=1}^{l} [i + (k-i)] = kl$$

This implies that:

$$\frac{FC(\sigma)}{OPT(\sigma)} \geq \frac{\frac{kl(l+1)}{2} + \frac{l(1-l^2)}{3}}{kl} \xrightarrow{k \to \infty} R(FC) \geq \frac{(l+1)}{2}$$

□

It is easy to see that any algorithm that does not perform paid transpositions is at least l-competitive. This observation comes from the argument that on arbitrary request sequence of length n, any algorithm will pay at most nl, while the optimal solution will pay at least n. As we saw, both *Transpose* and *Frequency Count* achieve a competitive ratio of $\Omega(l)$. In that thinking, we can say that both algorithms perform poorly. Perhaps, we can find a better algorithm that performs closer to the lower bound of $2 - \frac{2}{l+1}$.

5.4 Move-To-Front

Another natural algorithm for List Accessing is *Move-To-Front (MTF)*. Sleator and Tarjan [54] were the first to use amortized analysis for online problems and proved that MTF is strictly 2-competitive, by using the potential function method. The algorithm uses only free transpositions. As we will see, MTF is in fact the optimal online algorithm for List Accessing in the deterministic case.

Move-To-Front (MTF): After accessing an element, move it to the front of the list, without changing the relative order of any other elements.

Before we proceed with the proof, we present the concept of amortized analysis in online algorithms and the potential function method.

5.4.1 Amortized Analysis - The Potential Function Method

The lower bounds shown for $TRANS$ and FC in 5.3 provide a guarantee that they achieve a large competitive ratio for List Accessing. We are still in need of a better algorithm. But, how can we prove that such algorithm performs well? In that case, we need to prove an upper bound of its competitive ratio for any input. The simplest argument that we can use is to find an upper bound for the total cost of our algorithm and a lower bound for OPT. However, most of the times OPT is not known, so we might come up with some trivial lower bound, for instance in our problem, we have $OPT(\sigma_i) \geq 1$ for each σ_i. Such argument seems that it cannot bring strong results.

We can think of another strategy. For example, if we want to prove that our algorithm ALG is c-competitive we can possibly show that $ALG(\sigma_i) \leq c \cdot OPT(\sigma_i)$ for all i. But this may do not hold for all i, OPT may pay a large cost in the beginning for actions that may significantly reduce its cost on future requests, when ALG would be enforced, by a worst-case input, to pay large costs. However, perhaps we could try to prove some kind of argument which guarantees that, if ALG performs an action that pays a large cost for a request now, it will pay significantly smaller costs in the future or if not, then OPT will also have to pay a large cost. Obviously, such action, though it seems costly in the current time, is proved to be beneficial in the future. For this reason, perhaps a 'discount' should be made to the incurred cost. This discounted cost is called *amortized cost* and the idea behind lies in the field of *Amortized Analysis*, introduced by Robert Tarjan in [56]. In Tarjan's words, amortized complexity is described as "averaging the running times of operations in a sequence over the sequence". Amortized Analysis is an average-case analysis that was proved to be very helpful and efficient in analyzing data structures and online algorithms in comparison with worst-case analysis over a single input.

One tool of *Amortized Analysis* is the *potential function method*, which is presented here in terms of proving competitiveness of an online algorithm, following the presentation in [21].

Let an online algorithm ALG and the optimal offline algorithm OPT. We can assume that ALG and OPT process request sequence σ independently, with each one performing a number of specific actions on the arrival of every request. Thus, each algorithm is associated with a particular sequence of actions over the request sequence. We combine these two sequences into one sequence with the actions of two algorithms in any order, with the only restriction of keeping the chronological order of actions per request, i.e. actions for request σ_{j+1} cannot appear before actions for request σ_j have finished. This grand sequence is called *event sequence* and each segment of it is called *event*. The partition of the sequence in events is free to be chosen in any way that can simplify the proof.

We also define the *configuration* S_{ALG} of an algorithm ALG as its state with respect to the problem parameters. For instance, ALG's configuration for List Accessing is the current order of the list maintained by the algorithm. Obviously, the configuration can change on every request by ALG's actions. We can imagine ALG and OPT performing their actions in their own configurations independently, i.e. ALG does not interfere with S_{OPT} and vice versa. The event sequence only serializes their actions in the order they are considered by the proof.

The *potential function* Φ is defined as a mapping of configurations S_{ALG} and S_{OPT} to a real number, i.e. $\Phi\colon S_{ALG} \times S_{OPT} \to \mathbb{R}$. We are interested in defining a potential function that satisfies certain conditions with respect to the event sequence $e_1, e_2, \ldots, e_m$. In particular, let Φ_i be the value of Φ just after event e_i. We define Φ_0 to be a constant depending on the initial configurations of ALG and OPT before the start of the request sequence. Based on the problem and selection of Φ, there are two popular ways to prove competitiveness of ALG:

First Way: Amortized Costs

Let ALG_i and OPT_i be the actual costs incurred by the respective algorithms during event e_i. We define the amortized cost a_i of ALG for event e_i:

$$a_i = ALG_i + \Phi_i - \Phi_{i-1}$$

Then, ALG is c-competitive if for any request sequence σ:

1. $a_i \leq c \cdot OPT_i$, *for each* e_i
2. *There exists constant b independent of σ such that $\Phi_i \geq b$, for each e_i*

The above argument is proved in the following:

Proof. From above definitions, it holds that:

$$ALG(\sigma) = \sum_{i=1}^{m} ALG_i = \sum_{i=1}^{m} a_i - \sum_{i=1}^{m} (\Phi_{i-1} = \Phi_i) \rightarrow ALG(\sigma) = \sum_{i=1}^{m} a_i + \Phi_0 - \Phi_m$$

From *(1)* and *(2)* and the previous equality we have:

$$ALG(\sigma) \leq c \sum_{i=1}^{m} OPT_i + \Phi_0 - b \rightarrow ALG(\sigma) \leq c \cdot OPT(\sigma) + \Phi_0 - b$$

So, ALG is c-competitive.

□

It becomes obvious now that the aforementioned 'discount' is the value $\Delta\Phi_i = \Phi_i - \Phi_{i-1}$. We can consider the potential function as a measure of similarity between ALG and OPT configurations. The less value Φ has, the more similar they are. If $\Delta\Phi_i < 0$, ALG 'approaches' OPT's configuration, so it receives a discount for the actual cost of its actions on event e_i.

Second Way: Interleaving Moves

Let ALG_i and OPT_i be the actual costs incurred by the respective algorithms during event e_i. Then, ALG is c-competitive if for any request sequence σ:

1. $\Delta\Phi_i = \Phi_i - \Phi_{i-1} \leq c \cdot OPT_i$, *for each e_i in which only OPT performs actions*
2. $\Delta\Phi_i = \Phi_i - \Phi_{i-1} \leq -ALG_i$, *for each e_i in which only ALG performs actions*
3. *There exists constant b independent of σ such that $\Phi_i \geq b$, for each e_i.*

We have the following proof:

Proof. We partition the actions on request σ_i into events $e_{i_{ALG}}$ and $e_{i_{OPT}}$ in which only actions of ALG and OPT exist, respectively. We have:

$$\sum_{i=1}^{2m} \Delta\Phi_i = \sum_{i=1}^{m} (\Delta\Phi_{i_{OPT}} + \Delta\Phi_{i_{ALG}}) \overset{(1),(2)}{\leq} \sum_{i=1}^{m} (c \cdot OPT_i - ALG_i)$$

$$\rightarrow \Phi_{2m} - \Phi_0 \leq c \cdot OPT(\sigma) - ALG(\sigma) \xrightarrow{(3)} ALG(\sigma) \leq c \cdot OPT(\sigma) + \Phi_0 - b$$

So, ALG is c-competitive.

□

We can proceed with the proof of 2-competitiveness now.

5.4.2 Strictly 2-competitiveness

The following result was proved by Sleator and Tarjan in [54]. The amortized cost method, as presented in 5.4.1, is used in the proof.

Theorem 5.4. *Let a list of length l. Then, Move-To-Front is $(2 - \frac{1}{l})$-competitive.*

Proof. We define the potential function Φ_i as the total number of *inversions* in MTF's list configuration with respect to OPT's list configuration. Inversions are defined as the number of pairs of elements which are in one relative order in MTF's list and in reverse order in OPT's list. This number is also called *Kendall tau distance* and formally is defined as $|(x_i, x_j) : S_{ALG}(x_i) < S_{ALG}(x_j) \wedge S_{OPT}(x_j) < S_{OPT}(x_j)|$, where S_{ALG}, S_{OPT} are the list configurations for ALG and OPT, showing the positions of elements x_i and x_j in respective lists. Kendall tau distance is a very common distance metric between two lists/permutations, so it can fit to the role of potential function Φ. By definition, it holds $\Phi_i \geq 0$ for every i. We can also assume that list configurations of OPT and MTF are the same at the beginning of request sequence σ, so $\Phi_0 = 0$.

We define three types of events in the event sequence taking place on the ith request, each one having their own summing impact on $\Delta\Phi_i$:

1. free transpositions performed by MTF, inducing $\Delta\Phi_{i_1}$
2. free transpositions performed by OPT, inducing $\Delta\Phi_{i_2}$
3. paid transpositions performed by OPT, inducing $\Delta\Phi_{i_3}$

The goal is to prove $a_i \leq c \cdot OPT_i$, where a_i is the amortized cost. Let x_j be the requested element on the ith request, w.l.o.g located at position j in OPT's list and at position k in MTF's list. Let v be the number of inversions that correspond to elements that are located before x_j in MTF's list and after x_j in OPT's list. Then, $k - v - 1$ elements precede x_j in both lists. Since x_j is in jth position in OPT's list, this means that $k - v - 1 \leq j - 1 \rightarrow k - v \leq j$.

First, MTF pays an access cost of k, thus $MTF_i = k$. We examine now event e_{i_1}, i.e. MTF's contribution to $\Delta\Phi_i$. MTF moves x_j to the front of its own list. This means that v existing inversions are eliminated and $k - v - 1$ new inversions are created. So, $\Delta\Phi_{i_1} = (k - v - 1) - v = k - 2v - 1$.

Secondly, it is the turn for OPT to perform its actions on request i, on its own list. OPT pays an access cost of j, so it can move x_j closer to the front by using some free transpositions, which we do not know, let them be f in number. Since x_j has already been moved to the front in ALG's list, such transpositions will eliminate f existing inversions. Thus, $\Delta\Phi_{i_2} = -f$. Also, OPT may have performed some paid transpositions, let them be p in number. Each one of them can induce a cost of at most 1. Thus, $\Delta\Phi_{i_3} \leq p$. Finally, the total cost of OPT for the ith request is $OPT_i = j + p$. So, we have:

$$\begin{aligned} a_i &= MTF_i + \Delta\Phi_{i_1} + \Delta\Phi_{i_2} + \Delta\Phi_{i_3} \leq k + (k - 2v - 1) - f + p \\ &= 2(k - v) - 1 + p - f \leq 2j - 1 + p - f \\ &\leq 2(j + p) - 1 \rightarrow a_i \leq 2OPT_i - 1 \end{aligned}$$

Summing up over an entire request sequence σ of arbitrary length n we instantly receive that $MTF(\sigma) \leq 2OPT(\sigma) - n$. Obviously, $OPT(\sigma) \leq nl$, so finally we get:

$$MTF(\sigma) \leq (2 - \frac{1}{l})OPT(\sigma)$$

□

It can be shown that MTF matches exactly the deterministic lower bound of $(2 - \frac{2}{l+1})$, proved in 5.2. The proof was given by Irani in [34], using the list factoring technique that will be discussed in 5.5. Thus, MTF is the optimal deterministic algorithm for List Accessing in terms of competitive analysis. Finally, we have to mention that this result is quite impressing. MTF achieves strictly 2-competitiveness, that is a constant 2-approximation for the offline problem, but with the input arriving online!

5.5 Short Bibliographic Note

The List Accessing problem has been studied extensively throughout the years. We make a brief presentation of only some techniques, algorithms and variants that have appeared in List Accessing literature. For a more analytic list of references, the reader can refer to [21] [46].

The List Factoring Technique

One technique that is extensively used in List Accessing problems is the *List Factoring Technique*. This method enables the analysis to be reduced in lists of size 2. Such invention is proved to be helpful because many arguments can be simplified when applied to pairs of elements. For example, the optimal offline algorithm for a list of length 2 is known, i.e. on a run of at least two consecutive requests for element x, OPT must move x to the front, if not already there, after the first request, using one free transposition. The description of the method below is taken from [21].

The technique is based on the *partial cost model*, according to which the access cost for element x in position i is $i - 1$, motivated by the $i - 1$ elements that block the access to x. If $ALG^*(\sigma)$ is the cost of ALG, an algorithm that does not use paid transpositions, within the partial cost model, it can be proved that:

$$ALG^*(\sigma) = \sum_{\{x,y\}\subseteq L, x\neq y} ALG^*_{xy}(\sigma)$$

where $ALG^*_{xy}(\sigma)$ is the number of times that x is in front of requested element y plus the times that y is in front of requested element x, in request sequence σ.

The *projection* of σ over elements x and y is defined as the request sequence σ_{xy} with only x and y, keeping their relative order. Also, the projection of list L over elements x and y is defined as the two-element list L_{xy}, that contains only x and y. Then, $ALG^*(\sigma_{xy})$ is defined as the total partial cost of ALG for serving σ_{xy} in list L_{xy}. ALG is said to satisfy the *pairwise property* if:

$$ALG^*(\sigma_{xy}) = ALG^*_{xy}(\sigma)$$

Alternatively, according to *pairwise property lemma*, ALG satisfies the pairwise property iff for every request sequence σ, when ALG serves σ, the relative order of every two elements x and y in L is the same as their relative order in L_{xy}, when ALG serves σ_{xy}.

Finally, the *factoring lemma* can be proved, according to which if online algorithm ALG does not use paid transpositions, satisfies the pairwise property and $ALG^*(\sigma_{xy}) \leq c \cdot OPT^*(\sigma_{xy})$ holds, for every σ and every pair $\{x, y\} \subseteq L$, then ALG is strictly c-competitive. The proof is based on the above two equations. The reader can refer to [21] for an analytic description of the list factoring technique.

Finally, for an algorithm that makes decisions independent of the cost model, like MTF, $TRANS$ and FC, it can be proved that c-competitiveness in the partial cost model induces c-competitiveness in the full cost model.

Indicatively, we present some historic results drawn in List Accessing with the use of list factoring. The method was introduced by Bentley and McGeoch in [20]. Irani [34] used the technique to prove that MTF's competitiveness is indeed tight to the deterministic lower bound of $2 - \frac{2}{l+1}$ and provide the first randomized algorithm for List Accessing, called $SPLIT$. Albers [4] proposed improved randomized algorithm $TIMESTAMP$ and Albers et al. [8] gave an even better randomized algorithm, called $COMB$. Also, Teia [57] proved a strong result on the randomized lower bound against oblivious adversaries. For the rest of this chapter, we will make no further reference on the list factoring technique. However, the reader should be aware that most of the results make either implicit or explicit use of this method.

The Offline Problem

Many results not demand any knowledge of the optimal offline algorithm. For example, as we saw in 5.4.2, algorithm OPT was considered as a black box, we did not know anything about its decisions on free or paid transpositions, yet the potential function method led to a strong result. However, better understanding of the offline case may be helpful in the design of better online algorithms. In the offline case, all requested elements are known in advance and must be served in order. The offline List Accessing problem was proved to be NP-Hard by Ambuhl [9], by performing a reduction from the *minimum feedback arc set problem*. One of the proposed algorithms for OPT is by Reingold and Westbrook [51], running in $O(2^l(l-1)!n)$ time and $O(l!)$ space. The authors improved the previous $O((l!)^2 n)$ result of Manasse et. al [41], by showing that instead of checking on each request all $l!$ possible list rearrangements, they can be restricted to at most 2^l

of them, called *subset transfers*. The best optimal offline algorithm as of today runs in $O(l^2(l-1)!n)$ and has been proposed by Divakaran [25]. The work was based on a similar idea to that of subset transfers, to prove that optimal rearrangements can be restricted to only *element transfers* of the requested element.

Randomization

Much research has been conducted on randomized algorithms for the List Accessing problem. The first randomized algorithm for List Accessing was $SPLIT$, proposed by Irani [34]. $SPLIT$ maintains for each element x, a pointer $p(x)$ to some element in list and is initialized to x. With probability $1/2$, requested element x is moved to the front, else with probability $1/2$, it is inserted in front of $p(x)$. $p(x)$ is then set to the first element in list. $SPLIT$ was proved to be $31/16$-competitive against an optimal offline adversary, breaking the deterministic lower bound of 2.

Algorithm BIT was then proposed by Reingold et. al [52]. BIT initializes for each element x, independently and uniformly at random, a bit $b(x)$. When x is requested, $b(x)$ is complemented. Then, if $b(x) = 1$, x is moved to front, else it remains unchanged. The algorithm was proved to be strictly $7/4$-competitive, by using the potential function method. The algorithm is a special case of $COUNTER(s, S)$, according to which a mod s-counter $c(x)$ is initialized randomly for each element x. On each access of x, $c(x)$ is decremented by 1 mod s. If $c(x) \in S$, then x is moved to front. The authors prove that a modification of $COUNTER$ with a random reset process and appropriate parameters s, S can yield an improved $\sqrt{3}$-competitive ratio against an oblivious adversary. Albers and Mitzenmacher [7] used a specific mixture of two $COUNTER$ algorithms to prove a $12/7$-competitive ratio.

Later, Albers [4] proposed $TIMESTAMP(p)$ (TS) algorithms. TS was more complicated than the previous algorithms. On access of element x, with probability p, it is moved to front and with probability $1-p$, it is moved in front of y, where y is the first element in list such that either it was not requested since the last request for x or it was requested exactly once since the last request for x and that request was served by TS using the $1-p$ scenario. If such y does not exist or x is requested for the first time, the algorithm does nothing. Fine tuning on p giave a $(1+\sqrt{5})/2$-competitive ratio against optimal offline adversary. It is also interesting that deterministic algorithms $TS(0)$ and $TS(1)$ were proved to be strictly 2-competitive. Especially, $TS(1)$ is the MTF algorithm, hence an alternative proof was given for 2-competitiveness and $TS(0)$ was only the second deterministic algorithm that achieved 2-competitiveness.

Finally, $COMB$, proposed by Albers et. al [8], is the best-known randomized algorithm. $COMB$ selects algorithm BIT with probability $4/5$ and algorithm $TS(0)$ with probability $1/5$ for serving the entire request sequence. $COMB$ was proved to be 1.6 competitive.

As for lower bounds, the best-known lower bound is $1.5 - \frac{5}{l+5}$ against an oblivious adversary, proved by Teia [57]. Later, Ambuhl et. al [9] proved an improved lower bound of 1.50084 assuming the partial cost model.

Miscellaneous

Many different types of analyses, assumptions, cost models and algorithms have been proposed for List Accessing.

Due to their numerous applications, List Accessing algorithms have been studied in practice under request sequences produced from empirical data or proobability distributions. The results are not unanimous and some of them appear to be in contrast with the theoretical competitive results. To mention only some of these researches, Bentley and McGeoch [20] noticed that FC outperfroms $TRANS$ and MTF usually outperforms FC for request sequences that are taken from text files, while Bachrach and El Yaniv in [30] and Bachrach et. al in [13] made an extensive study on a large number of deterministic and randomized algorithms, taking data from benchmarks used for testing the performance in dictionary maintenance and compression, also examining the influence of data locality.

To mention only some of the variants, Albers [3] studied the List Accessing problem with *lookahead* i.e. on every time step, the algorithm has knowledge of some future requests according to two different models: the weak, where the next m requests are known and the strong, where m pairwise distinct elements are known. Another interesting variant is that of List Accessing with locality of reference, studied in [6] [10] [26], providing theoretical models that represent locality of reference in data such that theoretical and empirical results match. MTF was shown to be superior to other algorithms in that case. Also, List Accessing has been studied under a relaxed cost model [24], in which access to element x_i costs $c_i \leq c_{i+1}$, for all i, a setting of providing advice for unknown parts of input [22], using temporary memory-buffering [45], in double linked lists [50] and in particular types of request sequences [47]. The classic cost model has received criticism and more realistic cost models have been proposed [42] [31].

Chapter 6

The Online Min-Sum Set Cover Problem

In section 4.3, we discussed the Generalized Min-Sum Set Cover or Multiple Intents Re-Ranking. The problem is motivated by web search ranking. Azar and Gamzu [12] mention the importance of ranking web pages based on the interests of different users. Each user is represented as a subset of search results that projects a particular profile type. The profile type is defined as a weighted vector over the elements of given subset and models the intents of searching for that particular user. The user scans the results from top to bottom, paying an overhead that depends on the position of the results in user subset. The goal is to provide a linear order of search results, so that the sum of total weighted cover time of sets is minimized. The authors note that in case where all profile vectors have the form $< 1, 0, \dots, 0 >$, the problem is equivalent to Min-Sum Set Cover discussed in Chapter 5. Such profile vectors represent navigational users, interested in only the first relevant search result. However, their work is based only on user logs, i.e. offline data stored from web engines in order to produce an optimal ordering of results. However, such scenario is restricted in time and space. There exists a realistic need for changing the ordering as users access the results online. Motivated by this problem, we propose the *Online Min-Sum Set Cover Problem*.

In this chapter, we introduce the Online Min-Sum Set Cover Problem and present the first deterministic and randomized results. Our current work is based basically on techniques and algorithms presented for the List Accessing problem in Chapter 5.

6.1 Problem Definition

Let L be an unsorted list of l elements $x_1, x_2, \dots, x_l$ and a collection of sets $S = S_1, S_2, \dots, S_m$ over the elements of L. Let $\sigma = \sigma_1, \sigma_2, \dots, \sigma_n$ be an online request sequence of sets in S, i.e. $\sigma_i = S_j$, with $S_j \in S$, for every $i \in [n]$. On every set request, an online algorithm performs *access* to the set by accessing at least one of the elements in set. Then, the algorithm is associated with an *access cost*, i.e. among the accessed elements in set, that of the element's position that is furthest from the head of L. The algorithm is allowed to reorganize the list by performing transpositions of consecutive elements. For these transpositions the algorithm must pay a *moving cost* according to the following:

- *free transpositions:* Immediately after accessing set σ_j, and paying an access cost of i, for $x_i \in \sigma_j$, it can move any element $x_k \in \sigma_j$ that is preceding x_i in the list, including x_i, to any position closer to the front of the list with no extra cost.
- *paid transpositions:* At any time, it can perform any number of transpositions between consecutive elements in L and pay a cost of 1 for each transposition.

The goal is to find an algorithm that minimizes the total cost incurred by σ. More formally, let L_t be the list configuration after the algorithm has processed set request σ_t. We define as L_0 the initial list configuration. Also, let $L_{t-1}(x_j)$ denote the position of $x_j \in \sigma_t$ in current list configuration L_{t-1}. On the arrival of σ_t, every algorithm can select any element from σ_t to access. This access cost is denoted by disjunctive cost function $\bigvee_{x_j \in \sigma_t} L_{t-1}(x_j)$. Then, the algorithm performs some free transpositions on elements permitted by the constraint defined above. Finally, it may perform paid transpositions, denoted by $move(L_{t-1}, L_t)$. Then, the goal is to find:

$$\min \sum_{t=1}^{n} [\bigvee_{x_j \in \sigma_t} L_{t-1}(x_j) + move(L_{t-1}, L_t)]$$

under the problem constraints defined above.

We make some observations on the problem definition. List Accessing notation is used extensively. In fact, the problem can be interpreted as a *multidimensional version of List Update Problem.* Instead of element requests we have set requests over elements in the list.

However, the cost model is motivated by the Min-Sum Set Cover problem, presented in Chapter 4. In the offline Min-Sum Set Cover, all set requests are known and the goal is to find a static linear ordering of the elements that minimizes total sum of hitting times of sets. In the offline case, no moving costs are accounted, we only want to find the optimal linear ordering, without caring about the moving costs from initial configurations. Without any transpositions, paid or free, the disjunctive cost function gives naturally its place to the cost of first element's position in set in optimal static ordering L, i.e. in the optimal static ordering we have no reason to pay for accessing elements in greater positions. Hence, the previous objective function takes exactly the form presented in 4.1.2.

In the online counterpart, studied in this chapter, it is necessary to define a problem in which algorithms can adapt to incoming sets. This is the reason why we allow transpositions. The setting of how to cost these transpositions is already provided by List Accessing. The disjunctive cost function is initiated by the need to provide the algorithm with freedom of performing rearrangements, in order to reduce future accesses. In the setting of web search ranking, we said that we are interested in reducing future access costs to the element in set that occurs first in the ordering. Yet, the rest of elements in set may have some importance for future requests, so the algorithm is left free to select up to which element it needs to perform access.

It becomes obvious that the exact offline counterpart of our problem is that of sets arriving in sequential order, but all of them are known in advance. Also, the results of this thesis are proved against an optimal offline adversary and not restricted in the optimal static solution, which is Min-Sum Set Cover. Yet, Min-Sum Set Cover provides a theoretical framework that can be used in the future for algorithms that perform well against an optimal static adversary, since in that case we have a better understanding for the unknown value of OPT and a constant greedy approximation for it. We thus adopt the term Online Min-Sum Set Cover for our problem.

6.2 Our Results

We focus primarily on the deterministic case of *Online Min-Sum Set Cover*. We prove a deterministic lower bound and present algorithms $MoveFront$, $MoveLast$, $MoveSet$, motivated by *Move-To-Front* from List Accessing. On the randomized case, we present some simple arguments over two proposed algorithms, *Randomized Static* and *Randomized Move-To-Front*, to draw conclusions on their competitiveness. In this section, we make use of the following definitions.

Definition 6.1. *A request sequence σ is called A-**regular** when every set $\sigma_i \in \sigma$ has cardinality of A, i.e. $|\sigma_i| = A$. The sets of that sequence are called A-regular sets.*

Definition 6.2. *A request sequence σ is called **irregular** when there is no positive constant A such that σ is A-regular. The sets of that sequence are also called irregular sets.*

Trivially, the List Accessing Problem receives only 1-regular sequences as input.

6.2.1 A deterministic lower bound

Using the averaging technique, as seen in 5.2, we prove a lower bound for the competitiveness of deterministic algorithms.

Proposition 6.1. *(A-regular sets) For an A-regular request sequence on a list of length l, any deterministic online algorithm has a competitive ratio of at least $A + 1 - \frac{A(A+1)}{l+1}$.*

Proof. First, the adversary creates an A-regular request sequence such that on every request, the requested set contains the A last elements of the current list configuration. Every online algorithm ALG pays an access cost of at least $[l - (A - 1)]$ for each request, this is the position of the closest element to the front of the list. Thus, for an adversarial request sequence σ of arbitrary length n, it will hold that $ALG(\sigma) \geq n(l - A + 1)$, including any paid transpositions.

We consider the set of static offline algorithms. First, they pay for an initial cost $b = O(l^2)$ of paid transpositions for configuring the static permutation. Then, every algorithm pays for an access cost equal to the position of the element that is closer to the front of the list among elements in the set, for each set request. Since no reorderings are made, every static algorithm does not benefit from paying larger access costs, e.g. the position of the second closer element to the front, hence it pays the least possible on every request.

Now, consider an A-regular set request. First, we intend to find the total cost of the $l!$ algorithms for this request. To do this, we will count the permutations that have access cost of i, for every $1 \leq i \leq l$. For such counting, we use the combinatorial arguments below in the following order:

1. For a permutation that pays an access cost of i, it follows that, from the elements in requested set, the one closer to the front of the list is located in position i and no other element from the set is located in a position preceding i. Moreover, position i can be occupied by any of the A elements.

2. The rest $A-1$ elements of set can choose among $l-i$ positions to be ordered. Thus we have $A-1$-permutations of $l-i$, i.e. $P(l-i, A-1) = \frac{(l-i)!}{(l-i-A+1)!}$ ways to do that.

3. Having placed and ordered these A elements in positions from i to l, the rest $l-A$ elements of the list are left to be ordered. This can be done in $(l-A)!$ ways.

4. It must hold that $i \leq l-(A-1)$, since in the extreme case, the elements from the set will occupy the last A positions in permutation.

Gathering the above, we conclude that there are $A\frac{(l-i)!}{(l-i-A+1)!}(l-A)!$ permutations that pay an access cost of i for a single request. Thus, the sum of access costs over all permutations is:

$$\sum_{i=1}^{l-A+1} iA\frac{(l-i)!}{(l-i-A+1)!}(l-A)! = A!(l-A)! \sum_{i=1}^{l-A+1} i\binom{l-i}{A-1}$$
$$= A!(l-A)!\frac{1}{A(A+1)}(l+1)(l-A+1)\binom{l}{A-1}$$
$$= \frac{(l+1)!}{A+1}$$

Hence, the sum of total costs for the entire request sequence σ of arbitrary length n over all permutations is at most:

$$n\frac{(l+1)!}{A+1} + l!b$$

The optimal cost will be at most the average cost of static algorithms, i.e.

$$OPT(\sigma) \leq \frac{n\frac{(l+1)!}{A+1} + l!b}{l!} = n\frac{l+1}{A+1} + b$$

Finally, for any deterministic algorithm ALG, it will hold:

$$\frac{ALG(\sigma)}{OPT(\sigma)} \geq \frac{n(l-A+1)}{n\frac{l+1}{A+1} + b} \xrightarrow{n\to\infty} \frac{ALG(\sigma)}{OPT(\sigma)} \geq \frac{l-A+1}{\frac{l+1}{A+1}} \to R(ALG) \geq A+1 - \frac{A(A+1)}{l+1}$$

$\square$

As we can see, the above generalizes the deterministic lower bound for List Accessing. Setting $A=1$ we get $R(ALG) \geq 2 - \frac{2}{l+1}$, that is exactly the result in 5.2. Also we notice that setting $A = l$, we get a lower bound of 1. In that case, all elements are requested from the set, so an algorithm accessing the first element of the list is trivially optimal.

The previous result addressed only to regular input. For an irregular input sequence, we have the following proposition.

Proposition 6.2. *(Irregular sets) Let an irregular request sequence* $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ *with respective cardinalities* $A_1, A_2, \ldots, A_n$*, on a list of length* l*. Then, any deterministic online algorithm has a competitive ratio of at least* $\frac{n(l-A+1)}{(l+1)(\sum_i \frac{1}{A_i+1})}$*, where* $A = \frac{\sum_i A_i}{n}$ *is the average cardinality of set requests.*

Proof. An adversarial sequence could be just like the one in proof of Proposition 1, i.e. for request σ_i, the adversary requests the A_i last elements in the list, thus incurring a cost of at least $l - A_i + 1$. Similar to previous proof, the sum of access costs for the single request σ_i over all static offline algorithms is $\frac{(l+1)!}{A_i+1}$. So, from the averaging technique we get that for any deterministic algorithm ALG, it holds:

$$R(ALG) \geq \frac{\sum_i (l - A_i + 1)}{(l+1)(\sum_i \frac{1}{A_i+1})} = \frac{n(l - A + 1)}{(l+1)(\sum_i \frac{1}{A_i+1})}$$

□

We are interested in finding a general lower bound for our problem, that holds for all types of request sequences. The previous result depends on the values of A_i. Can we find an adversarial irregular request sequence that increases the lower bound over $A + 1 - \frac{A(A+1)}{l+1}$? The answer is no.

Proposition 6.3. *(General) For an arbitrary request sequence with average set cardinality* A *on a list of length* l*, any deterministic online algorithm has a competitive ratio of at least* $A + 1 - \frac{A(A+1)}{l+1}$*.*

Proof. We assume that A is a positive integer w.l.o.g. If the request sequence is A-regular, the proposition is true, as we already saw in Proposition 1. What we want to find is whether there exists a particular form of irregular request sequence that induces a greater lower bound. Let request sequence $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ with respective cardinalities $A_1, A_2, \ldots, A_n$. Given average cardinality A, maximizing value $\frac{n(l-A+1)}{(l+1)(\sum_i \frac{1}{A_i+1})}$ is equal to minimizing $\sum_i \frac{1}{A_i+1}$. From *Cauchy–Schwarz inequality* we have:

$$\begin{aligned}
(\sum_i \frac{1}{A_i+1})[\sum_i (A_i+1)] \;&\geq\; (\sum_i \sqrt{\frac{1}{A_i+1}}\sqrt{A_i+1})^2 \\
\rightarrow (\sum_i \frac{1}{A_i+1})[n(A+1)] \;&\geq\; n^2 \\
\rightarrow \sum_i \frac{1}{A_i+1} \;&\geq\; \frac{n^2}{n(A+1)} \\
\rightarrow \sum_i \frac{1}{A_i+1} \;&\geq\; \frac{n}{A+1}
\end{aligned}$$

Equality can hold only if $A_i = A_j$ for every i, j. This means that $A_i = A$ for every i, i.e. equality holds only when the request sequence is A-regular! Thus, replacing back $\frac{n}{A+1}$ to lower bound, we obviously receive the result of Proposition 1:

$$R(ALG) \geq \frac{n(l - A + 1)}{(l+1)\frac{n}{A+1}} = A + 1 - \frac{A(A+1)}{l+1}$$

□

From the formal definition, the competitive ratio should be independent of values related to the request sequence. Breaking this law, we presented the previous results dependent on A, either it holds for the constant cardinality of a regular request sequence, either for the average cardinality of an irregular request sequence. Such results can be seen as the limits of an online algorithm, when it is restricted to perform against sequences of a given price of A. But, since we study online algorithms under worst case, we would like to find the price of A that maximizes this lower bound. In that way, we can get a more comprehensive description of our lower bound. This is provided by the following corollary:

Corollary 6.1. *For arbitrary request sequences on a list of l elements, any deterministic online algorithm for Online Min-Sum Set Cover has competitive ratio of $\Omega(\frac{l}{4})$.*

Proof. We can see that $A + 1 - \frac{A(A+1)}{l+1}$ is maximized for $A = \frac{l}{2} - 1$. For that value, competitive ratio becomes $\dfrac{\frac{l}{2}(\frac{l}{2}+2)}{l+1} \in (\frac{l}{4}, \frac{l}{4}+1)$.

□

In this point, we make two important notes on the previous results:

1. In above propositions, we drew a lower bound for any deterministic algorithm ALG, such that $ALG(\sigma) \geq n(l - A + 1)$. This is the general guarantee we can have for the access cost of ALG in an adversarial request sequence. For a specific algorithm that may not pick for access the first element in the list among the elements in set, but succeeding elements, this lower bound can grow. For example, for an A-regular request sequence, if an algorithm performs access to the ith closer element to the front of the list and further, then the stronger inequality $ALG(\sigma) \geq n(l - A + i)$ will hold, inducing a lower bound of $\frac{l+i}{l+1}(A+1) - \frac{A(A+1)}{l+1}$, with $i \leq A$.

2. From the corollary, we can conclude that there is no algorithm that can achieve sublinear competitiveness for all values of A. Also, every deterministic algorithm is trivially at least l-competitive. Thus, the best we can do is to find an algorithm that is $O(\frac{l}{4})$-competitive, bridging the linear gap between $(\frac{l}{4}, l]$. This result is quite impressing: in List Accessing where a single element is requested every time, there exists a constant 2-competitive algorithm, but if the request sequence is composed by sets of elements of arbitrary cardinality, we cannot do better than $O(l)$-competitiveness scaled by up to a constant of 4. For this reason, we can allow the quest for algorithms that perform well on specific values of A and not all of them, aiming to achieve results close to lower bound $A + 1 = \frac{A(A+1)}{l+1}$.

6.2.2 MoveFront

When a set request arrives, any algorithm has a choice on which elements to access and pays the cost according to the cost model defined in 6.1. One algorithm that occurs naturally is to select and move the element that is at the front of the requested set in current list to the front of the list. This element is somewhat representative of the set, since its position is the least cost any algorithm has to pay for accessing the set. This is algorithm *MoveFront* and uses only free transpositions.

***MoveFront (MF)**:* After accessing the first element (front) of the set request in current list configuration, move it to the front of the list, without changing the relative order of any other elements.

Before we proceed with the result, we prove the lemma below, that will help us in the following.

Lemma 6.1. *Given a collection T of n natural numbers with average value $q \in \mathbb{Q}^+$, there exists a collection R of n natural numbers with average value q, such that each number is at least $\lfloor q \rfloor$.*

Proof. Let $T = \{x_1, x_2, \ldots, x_n\}$, such that $\frac{\sum_n x_i}{n} = q$. Let the partition of T into subsets $T_1 = \{x_i \in T | x_i < \lfloor q \rfloor\}$, $T_2 = \{x_i \in T | x_i = \lfloor q \rfloor\}$ and $T_3 = \{x_i \in T | x_i > \lfloor q \rfloor\}$. Also, let the 'complementary' sets $T_1' = \{y_i | y_i = \lfloor q \rfloor - x_i, \forall x_i \in T_1\}$ and $T_3' = \{y_i | y_i = x_i - \lfloor q \rfloor, \forall x_i \in T_3\}$. We have that:

$$\frac{\sum_{T_1} x_i + \sum_{T_2} x_i + \sum_{T_3} x_i}{n} \geq \lfloor q \rfloor \rightarrow \frac{\sum_{T_1'}(\lfloor q \rfloor - y_i) + \sum_{T_2} x_i + \sum_{T_3'}(y_i + \lfloor q \rfloor)}{n} \geq \lfloor q \rfloor$$

$$\rightarrow \frac{n\lfloor q \rfloor + \sum_{T_3'} y_i - \sum_{T_1'} y_i}{n} \geq \lfloor q \rfloor \rightarrow \sum_{T_3'} y_i \geq \sum_{T_1'} y_i$$

Due to the last inequality, we can take $\sum_{T_3'} y_i - \sum_{T_1'} y_i$ units from the surplus of T_3' and eliminate the deficit of T_1'. This means that we can construct a new collection R, where each $x_i \in T_1$ is increased by y_i such that $x_i + y_i = \lfloor q \rfloor$ and for T_3, we can distribute the decrease of $\sum_{T_3'} y_i - \sum_{T_1'} y_i$ units accordingly such that each $x_i \in T_3$ remains at least $\lfloor q \rfloor$ after the decrease. In that way, each number in R is at least $\lfloor q \rfloor$.

□

We now prove the following proposition for MF.

Proposition 6.4. *Let a request sequence of average set cardinality $A \geq 2$, on a list of length l. Then, $MoveFront$ is $l - A + 1$-competitive against an optimal offline adversary.*

Proof. On every set request σ_i, access to the first element of the set in the current ordering induces an access cost of at most $l - A_i + 1$. Trivially, the optimal offline solution OPT induces an access cost of at least 1 per request. Thus, for any request sequence σ of length n, we get:

$$\frac{MF(\sigma)}{OPT(\sigma)} \leq \frac{\sum_i (l - A_i + 1)}{n \cdot 1} = \frac{n(l - A + 1)}{n} \rightarrow R(MF) \leq l - A + 1$$

Now, we want to prove a lower bound of $l - A + 1$ for $R(MF)$, so it suffices to generate a specific request sequence for which MF achieves this competitive ratio. Let the initial list configuration $L = [x_1, x_2, \ldots, x_l]$. Supposing that there exists an (infinite) request sequence of average cardinality $A \geq 2$, from Lemma 6.1, the adversary can construct a sequence $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ with respective cardinalities $A_1, A_2, \ldots, A_n$, with $A_i \geq 2$, for every i, in which the last A_i elements in current configuration are requested every time. With that input, MF always moves to front the current element located in position $l - A_i + 1$, so a cost of $l - A_i + 1$ is induced per request.

Hence, since $A_i \geq 2$, the last element x_l remains unchanged in its position for the entire process and belongs to every requested set. An optimal offline solution for that sequence can be at least as good as the solution that initially moves x_l to the front of the list, by paid transpositions of total cost l. Since element x_l appears in every set request, after moved to the front, this solution will pay a cost of 1 on every request. Thus:

$$\frac{MF(\sigma)}{OPT(\sigma)} \geq \frac{\sum_i (l - A_i + 1)}{n \cdot 1 + l} = \frac{n(l - A + 1)}{n + l} \xrightarrow{n \to \infty} R(MF) \geq l - A + 1$$

From the two proven inequalities, we get that MF is $l - A + 1$-competitive.

□

We have to note that the proposition holds for any rational number $A \geq 2$ that can stand as the average cardinality of an infinite sequence of sets. This restriction comes from the need to construct an adversarial sequence with sets of cardinality at least 2, in order to induce at least one unchanged element, so that optimal solution can move it to the front. If a set of cardinality 1 occurs in the sequence, then we cannot apply the above argument of fixed element. But, if $A \geq 2$ then from Lemma 6.1, we can construct another sequence with that property. For $A < 2$, we cannot make modifications and produce such sequence.

This machinery was invented in order to include irregular sequences in our theorem. We can always construct an adversarial A-regular sequence for which MF is $l - A + 1$-competitive, just request the last A elements every time like in proof. But, if A is a rational number, we have to find particular values for A_i. So, given an existed sequence of average cardinality A, with Lemma 6.1 we generate another sequence of the same average cardinality for which MF performs worst.

Obviously, we are not interested in $A = 1$, the List Accessing case, since MTF projects a constant 2-competitiveness. We notice that MF performs better for large values of cardinality A. A large value of A intuitively means that the first element of the set in ordering is at smaller positions, so there cannot be large incongruities between access costs per request for OPT and MF. For this reason, performance of MF for sequences of small A is poor. A simple intuition is that MF always pays a large cost for accessing an element that is far away from the head of list, while OPT is able to select and move elements that are even further in order to pay small costs for future requests. Overall, as lower bound in 6.2.1 can be written as $\frac{(A+1)(l-A+1)}{l+1}$, we conclude that MF is not tight by a factor of $\frac{A+1}{l+1}$.

MF has also another drawback. Being unable to access elements other than the front one, it is vulnerable to an adversarial sequence in which optimal value is obtained by moving to the front an element, that is never moved by MF despite its great importance, hitting all the requested sets. In such request sequences, MF is unable to converge and follow the optimal solution.

6.2.3 MoveLast

Algorithm $MoveLast$ is an attempt to react to configurations of optimal solution as requests arrive and adapt to them. $MoveLast$ is motivated by the idea of reducing the largest cost incurred by accessing the requested set, that is the last element in list configuration among the elements in set. In that way, $MoveLast$ makes a somewhat repairing move that $MoveFront$ cannot. Like $MoveFront$, it makes only free transpositions.

MoveLast (ML): After accessing the last element of the set request in current list configuration, move it to the front of the list, without changing the relative order of any other elements.

However, we receive the following proposition for ML.

Proposition 6.5. *Let a request sequence of average set cardinality $A \geq 2$, on a list of length l. Then, MoveLast is l-competitive against an optimal offline adversary.*

Proof. On every set request, accessing the last element of the set in current ordering induces a cost of at most l. Trivially, the optimal offline solution induces a cost of at least 1 per request. Thus, for any request sequence σ of length n, we get:

$$\frac{ML(\sigma)}{OPT(\sigma)} \leq \frac{nl}{n \cdot 1} \rightarrow R(ML) \leq l$$

We are searching for a lower bound of $R(ML)$ now. Let the inital list configuration $L = [x_1, x_2, \ldots, x_l]$. For any $A \geq 2$ that can stand as average set cardinality of a sequence of sets, we can construct an adversarial sequence for which $A_i \geq 2$, for every i, from Lemma 6.1. We consider a request sequence $\sigma = \sigma_1, \sigma_2, \cdots, \sigma_n$ and an arbitrary element x_i, such that x_i is fixed in every σ_j, i.e. $x_i \in \sigma_j$, for every j. Also, every σ_j contains the current element located in position l of the list. The rest of elements in set, if any, can be arbitrarily chosen. In the request that x_i is located in position l, we can arbitrarily choose all other elements, too. Clearly, on every request, ML pays an access cost of l for the last element in list.

Hence, elements $x_l, x_{l-1}, \ldots, x_1$ pass from position l of the list, one after the other as requests arrive, and then are moved to the front of the list by ML, repetitively.

On the other hand, for sequence σ, optimal offline solution OPT can be at least as good as the solution that initially moves element x_i to the front, by paid transpositions of total cost $b = O(l)$. Since x_i appears in every set request, after moved to the front, this solution will pay a cost of 1 per request. So:

$$\frac{ML(\sigma)}{OPT(\sigma)} \geq \frac{nl}{n \cdot 1 + b} \xrightarrow{n \to \infty} R(ML) \geq l$$

This completes our proof that ML is l-competitive.

□

As we see, ML performs worse than MF in the worst case. Indeed, the argument of fixed element in every set request that we used for constructing the adversarial request sequence for MF can still be used, but this time the fixed element x_i does not stay unmoved throughout the process. Though this is the most important element, since it hits every requested set, the access cost is incurred by the element that is in the last position of the list every time.

ML makes a 'repairing' move of moving to the front the element that induces the largest access cost, yet the cost it pays is always that large and may appear to be unnecessary. In our example, moving x_i to the front would have proved to be the optimal choice.

However, we should note the following. In the end of section 6.2.1, we mentioned that for an A-regular sequence, an algorithm that accesses the ith element of set in ordering and further, cannot do better than $\frac{l+i}{l+1}(A+1) - \frac{A(A+1)}{l+1}$. ML moves always the Ath element in ordering on request σ_j, so the lower bound can become $\frac{(A+1)l}{l+1}$. We conclude that ML is not tight by a factor of $\frac{A+1}{l+1}$. This is exactly the factor that we got for MF. In that thinking, we can say that ML is not worse than MF, simply the lower bound for an algorithm that accesses the last element on every request, like ML, is larger and ML does not manage to lower this factor.

6.2.4 MoveSet

MoveLast pays the cost of accessing the last element of requested set and moves it to the front of the list by free transpositions. But in such case, the cost model defined in 6.1 permits further free transpositions for all elements in set. Algorithm *MoveSet* uses these transpositions to move the elements in set to the front of the list. We examine whether moving the entire set achieves better ratios.

***MoveSet (MS)**:* After accessing the last element of the set request in current list configuration, move all the elements in set to the front of the list, without changing their relative order and without changing the relative order of the rest elements in list.

For MS the proposition below holds:

Proposition 6.6. *Let a request sequence of average set cardinality $A \geq 2$, on a list of length l. Then, MoveSet is l-competitive against an optimal offline adversary.*

Proof. On every set request, accessing the last element of the set in current ordering induces a cost of at most l. Trivially, the optimal offline solution induces a cost of at least 1 per request. Thus, for any request sequence σ of length n, we get:

$$\frac{MS(\sigma)}{OPT(\sigma)} \leq \frac{nl}{n \cdot 1} \rightarrow R(MS) \leq l$$

Let the inital list configuration $L = [x_1, x_2, \ldots, x_l]$. Like before, from Lemma 6.1, given a sequence of average cardinality $A \geq 2$, we can construct an adversarial sequence of sets for which $A_i \geq 2$, for every i. Our adversarial argument and analysis is exactly the same with ML. This is a request sequence $\sigma = \sigma_1, \sigma_2, \cdots, \sigma_n$ for which there is an arbitrary element x_i, such that $x_i \in \sigma_j$, for every j and the current element located in position l of the list is contained in every set. The rest of elements in each set, if any, can be arbitrarily chosen. Thus, on every request, MS pays an access cost of l for the last element in list.

On the other hand, for sequence σ, optimal offline solution OPT can be at least as good as the solution that initially moves element x_i to the front, by paid transpositions of total cost $b = O(l)$. Since x_i appears in every set request, after moved to the front, this solution will pay a cost of 1 per request. So:

$$\frac{MS(\sigma)}{OPT(\sigma)} \geq \frac{nl}{n \cdot 1 + b} \xrightarrow{n\to\infty} R(MS) \geq l$$

This completes our proof that MS is l-competitive.

□

Finally, MS does not make any improvement in competitiveness. Again, it is vulnerable to a fixed point argument. Despite the fact that MS moves the entire set to the front of the list, thus also moving x_i to positions closer to the front, it is still bound to pay for that movement the position of the last element in set. By an adversarial sequence, this cost can be large enough, l per request, while OPT can initially move x_i to the front and then pay a very small cost. MS obviously fails to distinguish x_i among the elements of every requested set, thus it cannot benefit from its movement to positions at the front of the list.

6.2.5 Randomized Static

Our first randomized algorithm is *Randomize Static*, a 'dumb' algorithm of picking uniformly at random an initial static permutation of the list. This is our first attempt to receive randomized results and see an improvement in the competitive ratio in comparison with the proposed deterministic algorithms.

Randomized Static (RandStatic): Pick uniformly at random an initial static permutation of the list. On every requested set, pay for access the position of the top element of set in the static ordering.

We prove the following proposition:

Proposition 6.7. *Let an A-regular sequence on a list of length l. Then, Randomized Static has competitive ratio* $\bar{R}(RandStatic) \leq \frac{l+1}{A+1}$ *against an oblivious adversary.*

Proof. As we have seen in the proof for the deterministic lower bound 6.2.1, the sum of access costs for a single request over all permutations is $\frac{(l+1)!}{A+1}$. Since, the static permutation is selected uniformly at random, the expected access cost for one request will be $\frac{(l+1)!}{l!(A+1)} = \frac{l+1}{A+1}$. Thus, for any sequence σ of length n, by linearity of expectation and also including the initial moving cost $O(l^2)$, we get $\mathbb{E}[RandStatic(\sigma)] \leq \frac{n(l+1)}{A+1} + l^2$. This is exactly the expression that we found for the average cost of static algorithms in Proposition 1. Trivially, optimal offline OPT pays at least 1 per request, so we have:

$$\frac{\mathbb{E}[RandStatic(\sigma)]}{OPT(\sigma)} \leq \frac{\frac{n(l+1)}{A+1} + l^2}{n \cdot 1} \xrightarrow{n\to\infty} \bar{R}(RandStatic) \leq \frac{l+1}{A+1}$$

□

RandStatic performs better for large values of A, more elements from the requested set are located closer to the front with higher probability, thus a smaller access cost is induced on average. In general, *RandStatic* achieves a better competitive ratio than

MF and ML for any value of A. Without any intricate arguments, we can see that competitive ratio is improved with the use of randomization.

We did not include irregular request sequences in our theorem. The reason is that in such case, following steps of the proof in Proposition 2, we would get the expression $\frac{l+1}{n}\sum_i \frac{1}{A_i+1}$. Function $\sum_i \frac{1}{A_i+1}$ is a convex function, so it is maximized in the extreme points. For that reason, we can get a rather complicated expression on irregular sequences that does not say a lot for the expected competitive ratio. In fact, this technique fails to provide an upper bound guarantee for irregular sequences. We prefer to restrict to A-regular sequences that provide a compact result which shows the power of randomization.

6.2.6 Randomized Move-To-Front

Algorithm *Randomized Move-To-Front* picks an element from requested set uniformly at random and moves it to the front. By this way, it can pay on average smaller access cost for a requested set in comparison with $MoveLast$, while it responds on the fixed element argument used for constructing adversarial sequences, like in $MoveFront$.

***Randomized Move-To-Front (RMTF)**:* On every set request, access uniformly at random an element from the set and move it to the front of the list, without changing the relative order of any other elements.

RandStatic showed that it performs well for large values of A on average. We are interested in finding whether $RMTF$ performs well for small values of A. Avoiding the machinery for getting an exact result, we provide the following intuitive proof for showing that $RMTF$ does not perform well either.

Proof. (Sketch) Consider the case of a 2-regular sequence and let $L = [x_1, x_2, \dots, x_l]$ be the initial list configuration. We consider the request subsequence $\sigma' = \sigma'_1, \sigma'_2, \dots, \sigma'_{l-1}$, where $\sigma'_i = \{x_1, x_{i+1}\}$ (instead of x_1 we can fix any element). The adversarial sequence σ is constructed as an infinite repetition of σ'.

For an element x in position j, the expected access cost for set $\{x_1, x\}$ is at least $\frac{1+j}{2}$, because x_1 may be located in position greater than 1. In first repetition, when set σ'_i is requested, element x_i is located in position i, because $\sigma'_1, \sigma'_2, \dots, \sigma'_{i-1}$ contain elements that are located prior to x_i in list. Thus, for the first repetition we have $\mathbb{E}[RMTF(\sigma')] \geq \sum_{i=1}^{l-1} \frac{1+(i+1)}{2} = \frac{(l-1)(l+4)}{2}$.

In next repetitions of σ', we do not know the position of each x_i in list, so the above argument does not hold. However, we can make the following rough computation. Given the current position of x_i, we want to find its new expected position before the arrival of requested set $\sigma'_i = \{x_1, x_i\}$ in a new repetition of σ'. Let j be the position of x_i before σ'_i in kth repetition. Then, by the arrival of σ'_i in $(k+1)$th repetition, all sets of σ' will have been requested. On request σ'_i in kth repetition, for x_i we have:

1. With probability 1/2, x_i is moved to the front. During the rest $l-2$ requests, before the arrival of σ'_i in $(k+1)$th repetition, an expected number of $\frac{l-2}{2}$ elements (excluding x_1) is moved to the front. So, in that case, the new expected position for x_i is $1 + \frac{l-2}{2} = \frac{l}{2}$.

2. With probability 1/2, x_i remains in position j, since x_1 is moved to front in that case. During the rest $l-2$ requests, its new expected position can depend only on

the set requests that contain elements located in greater positions than x_i. These elements are $l - j$ in number, thus the expected number of elements that will move to the front is $\frac{l-j}{2}$. In that case, the new expected position for x_i is $j + \frac{l-j}{2} = \frac{l+j}{2}$.

Overall, the new expected position will be:

$$\frac{\frac{l}{2} + \frac{l+j}{2}}{2} = \frac{l}{2} + \frac{j}{4}$$

Thus, on new request σ_i' in $(k + 1)$th repetition, the expected cost will be at least $\frac{1+(\frac{l}{2}+\frac{j}{4})}{2} = \frac{l}{4} + \frac{j}{8} + \frac{1}{2}$. This means that $RMTF$ pays an expected access cost of $\Omega(l/4)$ per request. Trivially, optimal offline algorithm OPT keeps element x_1 in the front of the list and pays a cost of 1 per request, as x_1 hits every requested set. Thus, we conclude that $RMTF$ achieves an expected competitive ratio of $\Omega(l/4)$ for 2-regular request sequence. □

We remind that for $A = 2$, the deterministic lower bound is $3 - \frac{6}{l+1}$. Even the randomized algorithm $RMTF$ did not manage to induce a sublinear expected competitive ratio. Though the above computations are not exact, the previous proof provides an intuition on that even if $RMTF$ picks an element from set uniformly at random, the expected cost incurred is not decreased significantly and remains linear in relation to list length l.

6.2.7 Conclusion

The analysis we followed for proving competitive ratios of *MoveFront*, *MoveLast*, *MoveSet* was very simple. In contrast, as we saw in 5.4.2 for List Accessing, MTF was proved to be 2-competitive by deploying a potential function argument. However, the upper bounds for competitiveness of our proposed algorithms in Online Min-Sum Set Cover were based on trivial inequalities, specifically OPT was taken to pay at least a cost of 1 per request. Despite this naive approach, we managed to draw adversarial sequences for which the algorithms achieved competitive ratio tight to the respective upper bounds. Perhaps, this is an indication that we may come up with more subtle algorithms.

We have to note again that these *memoryless* algorithms were proved to perform poorly, e.g. MF is l-competitive when any algorithm can achieve such competitiveness. As we saw, none of them manages to handle the case of an element that covers each requested set. Such element should be moved closer to the front and incur small costs on future requests. A good algorithm perhaps should access this element without performing access to elements that are far from the front of the list and incur large costs. Also, *Randomized Move-To-Front* did not make significant improvements in terms of competitiveness. For example, the deterministic case for $A = 2$ gives a constant lower bound of 3, yet even the proposed randomized algorithm gives an expected ratio of $\Omega(l)$ scaled down by a constant factor.

The above arguments indicate that Online Min-Sum Set Cover is a radically different problem from List Accessing and needs different approach. The foremost reason for this is the defined cost model. In Online Min-Sum Set Cover, the cost is a disjunctive function of the positions of elements, i.e. any algorithm has the freedom to select which element in set to access and pay for its respective position in list. There exists a tradeoff of access cost and available free transpositions that the algorithm must handle. Moreover, we saw that the size of sets A is a significant parameter that any algorithm must take into account. If we are interested to design an algorithm that performs well for every value of A, then the lower bound explodes to $\Theta(\frac{l}{4})$. Just for comparison, the List Accessing has a constant lower bound.

In the beginning of this chapter, we discussed some obvious connections between the offline case of Online Min-Sum Set Cover and Min-Sum Set Cover. Yet, in our current work we did not make use of the theory and methods developed for Min-Sum Set Cover. The competitive ratios of the algorithms we applied hold against an optimal offline adversary. If we restrict to an optimal static adversary, then Min-Sum Set Cover can prove to be helpful, regarding knowledge of OPT. In any case, the greedy approach based on 'frequencies' of elements in given sets may lead to a new competitive online counterpart. This is basically our direction for future work.

Chapter 7

Future Work

In this thesis, we defined the *Online Min-Sum Set Cover Problem* and drew the first results for the deterministic and randomized case. Our goal was to present and track the inherent difficulties of this problem. We hope that our work was the first attempt for introducing the problem and motivating future research.

For the deterministic case, one major goal is to find an algorithm that achieves competitive ratio tight to the proven lower bounds. One potential direction is the deployment of greedy algorithm used in offline *Min-Sum Set Cover* for designing a novel online counterpart. Such algorithm may track for each element the number of sets that it hits. These frequencies need to be dependent on each other, e.g. if two elements hit many sets in common, then one of them must be of small significance. We believe that some kind of dynamic programming technique or a work function algorithm can help in this direction.

In List Accessing, in 5.5, we discussed the list factoring technique that was omnipresent in the proofs of many results throughout presented bibliography. The design of a list factoring technique for our problem is a challenging task that, if possible, may have great impact in future analysis.

Moreover, the theorems provided for the proposed deterministic algorithms were restricted in values $A \geq 2$. Perhaps, there exits some argument that generalizes the results for all values of A, including the case $A = 1$ of List Accessing.

Another direction can be the improvement of deterministic lower bound. The proof was based on a simple averaging technique. It is possible that a more complicated argument can provide a greater lower bound.

For the randomized case, there are many open problems. First and foremost, proving a randomized lower bound against some adversary model is a significant challenge. We made no reference in Yao's principle that is usually used in these proofs. Furthermore, a rigorous proof for competitiveness of $RMTF$ for different values of A needs to be given, avoiding the intuition that we provided for the case of $A = 2$. Of course, many other ideas, either new or from current bibliography in List Accessing and Min-Sum Set Cover, may be deployed for the design of competitive randomized algorithms.

Bibliography

[1] Dimitris Achlioptas, Marek Chrobak, and John Noga. “Competitive analysis of randomized paging algorithms”. In: *Algorithms — ESA ’96*. Springer Berlin Heidelberg, 1996, pp. 419–430. DOI: 10.1007/3-540-61680-2_72.

[2] S. Albers and M. Mitzenmacher. “Average Case Analyses of List Update Algorithms, with Applications to Data Compression”. In: *Algorithmica* 21.3 (July 1998), pp. 312–329. DOI: 10.1007/pl00009217.

[3] Susanne Albers. “A competitive analysis of the list update problem with lookahead”. In: *Theoretical Computer Science* 197.1-2 (May 1998), pp. 95–109. DOI: 10.1016/s0304-3975(97)00026-1.

[4] Susanne Albers. “Improved Randomized On-Line Algorithms for the List Update Problem”. In: *SIAM Journal on Computing* 27.3 (June 1998), pp. 682–693. DOI: 10.1137/s0097539794277858.

[5] Susanne Albers. “Online algorithms: a survey”. In: *Mathematical Programming* 97.1 (July 2003), pp. 3–26. DOI: 10.1007/s10107-003-0436-0.

[6] Susanne Albers and Sonja Lauer. “On List Update with Locality of Reference”. In: *Automata, Languages and Programming*. Springer Berlin Heidelberg, 2008, pp. 96–107. DOI: 10.1007/978-3-540-70575-8_9.

[7] Susanne Albers and Michael Mitzenmacher. “Revisiting the COUNTER algorithms for list update”. In: *Information Processing Letters* 64.3 (Nov. 1997), pp. 155–160. DOI: 10.1016/s0020-0190(97)00160-9.

[8] Susanne Albers, Bernhard von Stengel, and Ralph Werchner. “A combined BIT and TIMESTAMP algorithm for the list update problem”. In: *Information Processing Letters* 56.3 (Nov. 1995), pp. 135–139. DOI: 10.1016/0020-0190(95)00142-y.

[9] Christoph Ambühl. “Offline List Update is NP-hard”. In: *Algorithms - ESA 2000*. Springer Berlin Heidelberg, 2000, pp. 42–51. DOI: 10.1007/3-540-45253-2_5.

[10] Spyros Angelopoulos, Reza Dorrigiv, and Alejandro López-Ortiz. “List Update with Locality of Reference”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 399–410. DOI: 10.1007/978-3-540-78773-0_35.

[11] Yossi Azar and Iftah Gamzu. “Ranking with Submodular Valuations”. In: *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Jan. 2011. DOI: 10.1137/1.9781611973082.81.

[12] Yossi Azar, Iftah Gamzu, and Xiaoxin Yin. “Multiple intents re-ranking”. In: *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC ’09*. ACM Press, 2009. DOI: 10.1145/1536414.1536505.

[13] Bachrach, El-Yaniv, and Reinstädtler. “On the Competitive Theory and Practice of Online List Accessing Algorithms”. In: *Algorithmica* 32.2 (Feb. 2002), pp. 201–245. DOI: 10.1007/s00453-001-0069-8.

[14] Nikhil Bansal, Anupam Gupta, and Ravishankar Krishnaswamy. “A Constant Factor Approximation Algorithm for Generalized Min-Sum Set Cover”. In: *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Jan. 2010. DOI: 10.1137/1.9781611973075.125.

[15] Amotz Bar-Noy et al. “On Chromatic Sums and Distributed Resource Allocation”. In: *Information and Computation* 140.2 (Feb. 1998), pp. 183–202. DOI: 10.1006/inco.1997.2677.

[16] Y. Bartal, B. Bollobas, and M. Mendel. “A Ramsey-type theorem for metric spaces and its applications for metrical task systems and related problems”. In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001. DOI: 10.1109/sfcs.2001.959914.

[17] L. A. Belady. “A study of replacement algorithms for a virtual-storage computer”. In: *IBM Systems Journal* 5.2 (1966), pp. 78–101. DOI: 10.1147/sj.52.0078.

[18] S. Ben-David et al. “On the power of randomization in online algorithms”. In: *Proceedings of the twenty-second annual ACM symposium on Theory of computing - STOC ’90*. ACM Press, 1990. DOI: 10.1145/100216.100268.

[19] Jon L. Bentley, Kenneth L. Clarkson, and David B. Levine. “Fast linear expected-time algorithms for computing maxima and convex hulls”. In: *Algorithmica* 9.2 (Feb. 1993), pp. 168–183. DOI: 10.1007/bf01188711.

[20] Jon L. Bentley and Catherine C. McGeoch. “Amortized Analyses of Self-organizing Sequential Search Heuristics”. In: *Commun. ACM* 28.4 (Apr. 1985), pp. 404–411. ISSN: 0001-0782. DOI: 10.1145/3341.3349. URL: http://doi.acm.org/10.1145/3341.3349.

[21] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. New York, NY, USA: Cambridge University Press, 1998. ISBN: 0-521-56392-5.

[22] Joan Boyar et al. “On the list update problem with advice”. In: *Information and Computation* 253 (Apr. 2017), pp. 411–423. DOI: 10.1016/j.ic.2016.06.007.

[23] Niv Buchbinder and Joseph (Seffi) Naor. “The Design of Competitive Online Algorithms via a Primal—Dual Approach”. In: *Foundations and Trends® in Theoretical Computer Science* 3.2–3 (2009), pp. 93–263. DOI: 10.1561/0400000024.

[24] Marek Chrobak and John Noga. “Competitive Algorithms for Relaxed List Update and Multilevel Caching”. In: *Journal of Algorithms* 34.2 (Feb. 2000), pp. 282–308. DOI: 10.1006/jagm.1999.1061.

[25] Srikrishnan Divakaran. “An Optimal Offline Algorithm for List Update”. In: *arXiv:1404.7638* arXiv:1404.7638 (May 2014).

[26] Reza Dorrigiv and Alejandro López-Ortiz. “List update with probabilistic locality of reference”. In: *Information Processing Letters* 112.13 (July 2012), pp. 540–543. DOI: `10.1016/j.ipl.2012.04.002`.

[27] Uriel Feige, László Lovász, and Prasad Tetali. “Approximating Min Sum Set Cover”. In: *Algorithmica* 40.4 (July 2004), pp. 219–234. DOI: `10.1007/s00453-004-1110-5`.

[28] Uriel Feige, László Lovász, and Prasad Tetali. “Approximating Min-sum Set Cover”. In: *Approximation Algorithms for Combinatorial Optimization*. Springer Berlin Heidelberg, 2002, pp. 94–107. DOI: `10.1007/3-540-45753-4_10`.

[29] Amos Fiat et al. “Competitive paging algorithms”. In: *Journal of Algorithms* 12.4 (Dec. 1991), pp. 685–699. DOI: `10.1016/0196-6774(91)90041-v`.

[30] Ran Gilad-Bachrach and Ran El-Yaniv. “Online List Accessing Algorithms and Their Applications: Recent Empirical Evidence.” In: Jan. 1997, pp. 53–62. DOI: `10.1145/314161.314180`.

[31] Alexander Golynski and Alejandro López-Ortiz. “Optimal strategies for the list update problem under the MRM alternative cost model”. In: *Information Processing Letters* 112.6 (Mar. 2012), pp. 218–222. DOI: `10.1016/j.ipl.2011.12.001`.

[32] Sungjin Im. “Min-Sum Set Cover and Its Generalizations”. In: *Encyclopedia of Algorithms*. Springer New York, 2016, pp. 1331–1334. DOI: `10.1007/978-1-4939-2864-4_806`.

[33] Sungjin Im, Maxim Sviridenko, and Ruben van der Zwaan. “Preemptive and nonpreemptive generalized min sum set cover”. In: *Mathematical Programming* (Mar. 2013). DOI: `10.1007/s10107-013-0651-2`.

[34] Sandy Irani. “Two results on the list update problem”. In: *Information Processing Letters* 38.6 (June 1991), pp. 301–306. DOI: `10.1016/0020-0190(91)90086-w`.

[35] Sandy Irani and Anna R. Karlin. “On Online Computation”. In: *Approximation Algorithms for NP-Hard Problems, chapter 13*. PWS Publishing Company, 1997, pp. 521–564.

[36] A. R. Karlin et al. “Competitive randomized algorithms for nonuniform problems”. In: *Algorithmica* 11.6 (June 1994), pp. 542–571. DOI: `10.1007/bf01189993`.

[37] Anna R. Karlin et al. “Competitive snoopy caching”. In: *Algorithmica* 3.1-4 (Nov. 1988), pp. 79–119. DOI: `10.1007/bf01762111`.

[38] Elias Koutsoupias and Christos H. Papadimitriou. “On the k-server conjecture”. In: *Journal of the ACM* 42.5 (Sept. 1995), pp. 971–983. DOI: `10.1145/210118.210128`.

[39] James R. Lee. “Fusible HSTs and the Randomized k-Server Conjecture”. In: *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, Oct. 2018. DOI: `10.1109/focs.2018.00049`.

[40] Richard M. Karp. “On-Line Algorithms Versus Off-Line Algorithms: How Much is it Worth to Know the Future?” In: vol. 1. Jan. 1992, pp. 416–429.

[41] Mark Manasse, Lyle McGeoch, and Daniel Sleator. “Competitive Algorithms for On-line Problems”. In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC ’88. Chicago, Illinois, USA: ACM, 1988, pp. 322–333. ISBN: 0-89791-264-0. DOI: `10.1145/62212.62243`. URL: `http://doi.acm.org/10.1145/62212.62243`.

[42] Conrado Martı
nez and Salvador Roura. “On the competitiveness of the move-to-front rule”. In: *Theoretical Computer Science* 242.1-2 (July 2000), pp. 313–325. DOI: 10.1016/s0304-3975(98)00264-3.

[43] Jessica McClintock, Julián Mestre, and Anthony Wirth. “Precedence-Constrained Min Sum Set Cover”. en. In: *28th International Symposium on Algorithms and Computation (ISAAC 2017)* (2017). DOI: 10.4230/lipics.isaac.2017.55.

[44] Lyle A. McGeoch and Daniel D. Sleator. “A strongly competitive randomized paging algorithm”. In: *Algorithmica* 6.1-6 (June 1991), pp. 816–825. DOI: 10.1007/bf01759073.

[45] Rakesh Mohanty, Seetaya Bhoi, and Sasmita Tripathy. “A New Proposed Cost Model for List Accessing Problem using Buffering”. In: *International Journal of Computer Applications* 22.8 (May 2011), pp. 19–23. DOI: 10.5120/2604-3631.

[46] Rakesh Mohanty and N S. Narayanaswamy. “Online Algorithms for Self-Organizing Sequential Search - A Survey”. In: (Sept. 2009).

[47] Rakesh Mohanty, Burle Sharma, and Sasmita Tripathy. “Characterization of Request Sequences for List Accessing Problem and New Theoretical Results for MTF Algorithm”. In: *International Journal of Computer Applications* 22.8 (May 2011), pp. 35–40. DOI: 10.5120/2601-3627.

[48] *Online Algorithms Lecture Notes for CSCI 570 by David Kempe*. 2006.

[49] P. Rajhavan and M. Snir. “Memory versus randomization in on-line algorithms”. In: *IBM Journal of Research and Development* 38.6 (Nov. 1994), pp. 683–707. DOI: 10.1147/rd.386.0683.

[50] Mohanty Rakesh and Singh Kumar Rakesh. “New Results on Competitive Analysis of Move To Middle(MTM) List Update Algorithm using Doubly Linked List”. In: *Procedia Computer Science* 125 (2018), pp. 762–769. DOI: 10.1016/j.procs.2017.12.098.

[51] Nick Reingold and Jeffery Westbrook. “Off-line algorithms for the list update problem”. In: *Information Processing Letters* 60.2 (Oct. 1996), pp. 75–80. DOI: 10.1016/s0020-0190(96)00144-5.

[52] Nick Reingold, Jeffery Westbrook, and Daniel D. Sleator. “Randomized competitive algorithms for the list update problem”. In: *Algorithmica* 11.1 (Jan. 1994), pp. 15–32. DOI: 10.1007/bf01294261.

[53] Martin Skutella and David P. Williamson. “A note on the generalized min-sum set cover problem”. In: *Operations Research Letters* 39.6 (Nov. 2011), pp. 433–436. DOI: 10.1016/j.orl.2011.08.002.

[54] Daniel Dominic Sleator and Robert Endre Tarjan. “Amortized efficiency of list update rules”. In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing - STOC ’84*. ACM Press, 1984. DOI: 10.1145/800057.808718.

[55] Daniel Dominic Sleator and Robert Endre Tarjan. “Self-adjusting binary search trees”. In: *Journal of the ACM* 32.3 (July 1985), pp. 652–686. DOI: 10.1145/3828.3835.

[56] Robert Endre Tarjan. “Amortized Computational Complexity”. In: *SIAM Journal on Algebraic Discrete Methods* 6.2 (Apr. 1985), pp. 306–318. DOI: `10.1137/0606031`.

[57] Boris Teia. “A lower bound for randomized list update algorithms”. In: *Information Processing Letters* 47.1 (Aug. 1993), pp. 5–9. DOI: `10.1016/0020-0190(93)90150-8`.

[58] E. Torng. “A unified analysis of paging and caching”. In: *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE Comput. Soc. Press. DOI: `10.1109/sfcs.1995.492476`.