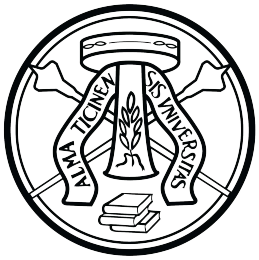

UNIVERSITÀ
DI PAVIA

Dipartimento di Matematica “Felice Casorati”

Corso di Laurea Magistrale in Matematica

Estensioni del Problema del Paging:

Analisi di Competitività e Valutazione Sperimentale per la Stampa 3D in Medicina

Relatore
Ch.mo Prof. Davide Duma

Candidato
Fabio Masnaghetti (507011)

Anno Accademico 2025–2026

A Daniela Ronchi
e Matteo Masnaghetti

Indice

Abstract

The optimization of production processes is a topic of great interest from both a theoretical and an applied perspective. This work originates from a problem observed at the 3D4Med laboratory, where anatomical models are produced through 3D printing for surgical applications. In this context, the limited number of available material cartridge slots requires deciding, during the production process, which materials should remain installed and which should be replaced, with the objective of reducing both the number of material changes and their associated costs.

The problem is addressed within the framework of online optimization, under the assumption that the sequence of models to be produced is not known in advance but is revealed progressively. As a reference model, the paging problem is considered, describing the management of a cache memory with limited capacity, and subsequently extended to the case in which each request may simultaneously involve multiple items (batch paging).

To provide a more realistic representation of the practical problem, a weighted version of the model is also introduced, where materials are associated with non-uniform replacement costs. In this setting, several online algorithms from the literature are adapted and analyzed by introducing a protected batch version, in which the materials required by the same print job cannot be removed while processing the corresponding request. In parallel, a Mixed-Integer Linear Programming (MILP) model is formulated to compute the optimal offline solution and to provide a benchmark for evaluating the performance of the online algorithms.

Finally, an experimental analysis is carried out on instances inspired by data provided by the 3D4Med laboratory, comparing the proposed algorithms

with the optimal solution obtained through the MILP model. The results show that the batch extension of GreedyDual achieves performance very close to the offline optimum, highlighting the effectiveness of the proposed approach for managing material replacements in the context of medical 3D printing.

Sommario

L'ottimizzazione dei processi produttivi rappresenta un tema di grande interesse sia dal punto di vista teorico sia da quello applicativo. Il presente lavoro nasce da una problematica osservata presso il laboratorio 3D4Med, dove vengono realizzati modelli anatomici mediante stampa 3D per applicazioni chirurgiche. In tale contesto, la limitata disponibilità di slot per le cartucce dei materiali rende necessario decidere, durante la produzione, quali materiali mantenere installati e quali sostituire, con l'obiettivo di ridurre il numero dei cambi e il relativo costo.

Il problema è stato affrontato nell'ambito dell'ottimizzazione online, assumendo che la sequenza dei modelli da produrre non sia nota a priori ma venga rivelata progressivamente. Come modello di riferimento è stato considerato il problema del paging, che descrive la gestione di una memoria cache di capacità limitata, successivamente esteso al caso in cui ciascuna richiesta possa coinvolgere simultaneamente più elementi (batch paging).

Per rappresentare in modo più fedele il problema reale, è stata inoltre introdotta una versione pesata del modello, nella quale i materiali sono associati a costi di sostituzione non uniformi. In questo contesto sono stati adattati e analizzati alcuni algoritmi online della letteratura, introducendo una versione batch con protezione, nella quale i materiali richiesti dalla stessa stampa non possono essere rimossi durante l'elaborazione della richiesta. Parallelamente, è stato formulato un modello di Programmazione Lineare Intera Mista (MILP), utilizzato per determinare la soluzione ottima offline e come termine di confronto per la valutazione degli algoritmi online.

Infine, è stata condotta un'analisi sperimentale su istanze ispirate ai dati del laboratorio 3D4Med, confrontando le prestazioni degli algoritmi propo-

sti con la soluzione ottima ottenuta tramite MILP. I risultati mostrano che l'estensione batch di GreedyDual è in grado di ottenere prestazioni molto vicine all'ottimo offline, evidenziando come l'approccio proposto costituisca una strategia efficace per la gestione delle sostituzioni di materiale nel contesto della stampa 3D medicale.

Capitolo 1

Contesto e motivazioni

L'ottimizzazione dei processi di produzione rappresenta un tema di grande interesse sia dal punto di vista applicativo sia da quello teorico, poiché problemi concreti possono spesso dare origine a nuove formulazioni algoritmiche e a modelli di ottimizzazione di carattere più generale.

Questo lavoro di tesi nasce da una problematica reale osservata presso il laboratorio 3D4Med, nato nel 2018 dalla collaborazione tra l'Università di Pavia e l'IRCCS Policlinico San Matteo di Pavia, con l'obiettivo di realizzare modelli anatomici personalizzati a partire da immagini mediche mediante tecnologie di stampa 3D.

Questi modelli vengono utilizzati in ambito chirurgico sia per la pianificazione pre-operatoria sia per il training. In particolare, consentono ai team medici di comprendere con maggiore precisione l'anatomia e la patologia specifiche di ciascun paziente, facilitando la valutazione dei rischi, la scelta della strategia chirurgica più appropriata e la familiarizzazione con il caso clinico prima dell'intervento [9].

Le tecnologie di stampa 3D impiegate permettono inoltre l'utilizzo di materiali con proprietà meccaniche differenziate, in grado di riprodurre il comportamento di diversi tessuti biologici. Questo rende possibile un'esperienza di simulazione realistica, utile per l'addestramento e la riduzione dei tempi operatori, contribuendo al miglioramento degli esiti clinici.

L'impiego di questi materiali incide in modo significativo sul costo com-

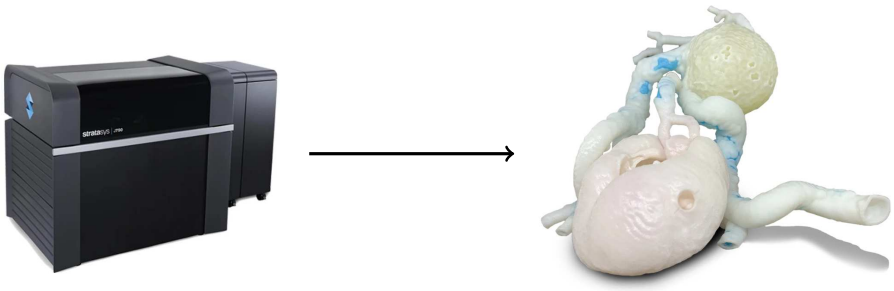

Figura 1.1: Immagine di una stampante J750 DIGITAL ANATOMY - Stratasys® ed esempio di modello anatomico 3D

plessivo di produzione, rendendo le stampe 3D particolarmente onerose. In questo contesto, l'obiettivo della presente tesi è analizzare e proporre strategie di ottimizzazione dei costi, affrontando il risultante problema di ottimizzazione sia dal punto di vista teorico che computazionale.

Formalizziamo il problema nel seguente modo. La stampa di ciascun modello 3D richiede l'utilizzo di un certo numero di materiali. Ogni stampante è dotata di un numero limitato di slot per le cartucce di materiale. Una volta occupati tutti gli slot disponibili, qualora la stampa del nuovo modello richieda uno o più materiali non installati, è necessario decidere quali cartucce rimuovere per far spazio alle nuove. Ci poniamo due obiettivi che sono, in prima analisi, il numero delle rimozioni e, successivamente, il costo complessivo del processo di produzione. Infatti, ogni volta che una cartuccia viene sostituita, viene avviata la pulizia dei tubi di alimentazione dei filamenti, con un conseguente spreco dei materiali sostituiti. Inoltre, tale pulizia comporta la perdita di una piccola quantità di materiale presente nelle cartucce non sostituite [5].

Per definire i parametri del problema quindi abbiamo bisogno di sapere i costi effettivi delle transizioni tra materiali, intese come le sostituzioni di materiali o la permanenza di uno stesso materiale in presenza di una sostituzione presso un altro slot della stampante. Questi costi vengono rappresentati attraverso una matrice dei costi, che costituisce la base informativa su cui si fondano le successive analisi e strategie di ottimizzazione. Una matrice con

le stime dei costi delle transizioni è stata fornita dal personale del laboratorio 3D4Med.

Tale matrice è asimmetrica: il costo di rimuovere il materiale j per inserire il materiale i non è lo stesso di togliere i per inserire j, ovvero in generale $c_{i,j} \neq c_{j,i}$. Inoltre, come osservato precedentemente, anche mantenere un materiale all'interno della matrice comporta un costo positivo, cioè $c_{i,i} > 0$.

La caratteristica principale che rende sfidante il problema della minimizzazione del numero o del costo dei cambi materiale è la mancata conoscenza dei futuri modelli da stampare e, conseguentemente, dei materiali necessari. Le richieste di stampa giungono infatti in modo dinamico, con i chirurghi che contattano il laboratorio pochi giorni prima dell'intervento chirurgico e con urgenze variabili che rendono non deterministici l'assegnamento degli ordini alle stampanti e la loro sequenza di stampa. Pertanto, l'ipotesi alla base di questa tesi é di conoscere gli ordini uno alla volta, vale a dire a ogni avvio di una nuova stampa, senza conoscere quali saranno i prossimi modelli da produrre sulla stessa stampante.

Il problema affrontato rientra nella categoria dei problemi di ottimizzazione online [1]. A differenza dei corrispondenti problemi offline dell'ottimizzazione classica, l'algoritmo non dispone dell'intera sequenza di input fin dall'inizio dell'esecuzione. Le informazioni vengono infatti rese disponibili progressivamente e, in ogni istante, le decisioni devono essere prese sulla base dei soli dati osservati fino a quel momento, senza alcuna conoscenza degli input futuri.

La prima analisi svolta riguarda l'ottimizzazione del numero di inserimenti e rimozioni delle cartucce, basandosi sul problema del paging, ovvero della gestione efficiente del trasferimento delle pagine tra una memoria veloce (*cache*) e una memoria lenta. Quando una pagina richiesta non è presente nella cache, essa deve essere caricata dalla memoria lenta; se lo spazio disponibile è esaurito, è necessario selezionare una pagina da rimuovere per fare posto a quella nuova. Il problema consiste quindi nel determinare una strategia di sostituzione delle pagine che minimizzi il numero di rimozioni/inserimenti. Nel caso visto per 3D4Med, le pagine rappresentano i vari materiali delle cartucce e la cache gli slot della stampante, mantenendo l'obiettivo di minimizzare

le rimozioni dei materiali. In tale contesto, una cartuccia viene assimilata a una pagina, e vengono studiati i principali algoritmi classici di gestione delle pagine, come LRU (*Least Recently Used*) e FIFO (*First In First Out*), assumendo un costo unitario per ogni operazione di inserimento.

Successivamente, per avvicinare maggiormente il modello al caso reale delle stampanti, il problema è stato esteso introducendo la possibilità di gestire richieste multiple contemporanee, modificando di conseguenza la struttura delle richieste e adattando i problemi di ottimizzazione considerati.

Come estensione al modello del paging, introduciamo il problema del k-server, soffermandoci sulla generalizzazione del problema, ovvero il paging pesato, dove ogni pagina ha un costo di inserimento all'interno della stampante. L'obiettivo di questa parte è ottimizzare i costi, introducendo alcuni algoritmi tipici del paging pesato e generalizzandoli alla versione batch con protezione, in cui vengono richieste più pagine per volta, che devono essere presenti contemporaneamente nella cache.

Inoltre, abbiamo studiato il problema dal punto di vista della sua versione offline, attraverso una formulazione di Programmazione Lineare Intera Mista (MILP - *Mixed Integer Programming Model*) che consente, data un'istanza, di calcolare la soluzione ottima (offline) e confrontarla con quelle fornite dagli algoritmi online. In conclusione, durante la fase implementativa sono stati analizzati gli algoritmi LRU, FIFO e GreedyDual nelle rispettive versioni batch, considerando le stampanti Stratasys Object 260 Connex 3 e Stratasys J750 DA. Successivamente, le prestazioni degli algoritmi sono state confrontate con MILP su sequenze di richieste batch di lunghezza variabile.

La tesi è strutturata come segue. Nel Capitolo 2 verrà analizzato il problema del paging nel contesto dell'ottimizzazione online. Dopo un'introduzione ai concetti fondamentali dell'ottimizzazione online, verrà presentato l'algoritmo offline ottimo LFD (*Longest Forward Distance*), che viene utilizzato per analizzare la competitività e le proprietà dei principali algoritmi di paging, tra cui LRU e FIFO. Infine, il modello verrà esteso al caso in cui si consideri anche il costo completo di accesso alle pagine.

Nel Capitolo 3 verrà presentata un'estensione del problema del paging al caso batch. In particolare, saranno analizzati gli algoritmi introdotti per

il problema classico, opportunamente adattati a tale contesto, e ne verrà studiata l'analisi competitiva.

Nel Capitolo 4 verrà introdotto il problema del k-server, presentato come una generalizzazione del problema del paging. In tale contesto, il costo associato al caricamento di una pagina non sarà più unitario, ma dipenderà dalla pagina da inserire. Dopo aver formalizzato il problema, verranno analizzati alcuni algoritmi, tra cui Greedy e Balance. Successivamente sarà introdotta un'ulteriore estensione, rappresentata dal paging pesato, nel quale a ciascuna pagina è associato un costo di caricamento fisso. In questo contesto verranno studiati gli algoritmi Greedy, Balance e GreedyDual. Quest'ultimo riveste un ruolo di particolare interesse, poiché può essere interpretato come una generalizzazione sia di LRU sia di Balance.

Nel Capitolo 5.1, gli algoritmi LRU, FIFO e GreedyDual verranno adattati al problema introdotto nel contesto del caso di studio del Laboratorio 3D4Med. In particolare, sarà introdotto un meccanismo di protezione dei materiali già presenti nella cache, al fine di impedirne la rimozione durante l'elaborazione della stessa richiesta batch. Infine, verrà formulato un modello di Programmazione Lineare Intera Mista (MILP), utilizzato per determinare una soluzione ottima della sequenza di richieste e come termine di confronto per la valutazione delle prestazioni degli algoritmi proposti.

Nel Capitolo 6, verranno confrontati sperimentalmente gli algoritmi proposti per il problema di 3D4Med. Le loro prestazioni saranno valutate confrontandole sia con la soluzione ottima ottenuta mediante il modello MILP, sia con una strategia *Random*, che simula la rimozione casuale delle cartucce dalla stampante. Al fine di capire la competitività degli algoritmi e il risparmio di costo.

Il Capitolo 7 chiude la tesi con le conclusioni e possibili sviluppi futuri del progetto di ricerca.

Capitolo 2

Algoritmi online e il problema del paging

Il problema di ottimizzazione del cambio delle cartucce delle stampanti 3D per la medicina, ispirato dal caso di studio del Laboratorio 3D4Med, rientra nella categoria dei problemi di ottimizzazione online. La differenza più significativa rispetto ai problemi tradizionali, di tipo offline, riguarda la conoscenza dell'istanza, che non è nota a priori ma, ad ogni istante di tempo, l'algoritmo ha a disposizione solamente le informazioni disponibili fino al tempo presente, nel quale deve gestire la richiesta e fornire un output.

2.1 Introduzione all'ottimizzazione online e all'analisi competitiva

Per formalizzare un generico problema di ottimizzazione online consideriamo $\sigma = (\sigma_1, ..., \sigma_n)$ la sequenza degli *input* (anche detti *richieste*) che vengono effettuati in n istanti di tempo distinti. Allora un algoritmo online ALG per il problema considerato produce una sequenza di *output* (anche detti *risposte*) $\tilde{\sigma} = (\tilde{\sigma_1}, ..., \tilde{\sigma_n})$ con la restrizione che al tempo t, per rispondere alla richiesta σ_t, l'informazione nota ad ALG sarà data dalla sottosequenza delle richieste passate $\sigma_1, ..., \sigma_t$.

Ad ogni algoritmo associamo anche un costo delle operazioni, che chiameremo con un leggero abuso di notazione $ALG(\sigma)$ rispetto a tutta la sequenza, nello studio più approfondito del paging nei prossimi capitoli, vedremo principalmente due casi, il primo è PIUC (Per Item Unit Cost) dove il costo di ogni elemento è unitario, mentre il secondo, più in linea con lo studio del problema delle stampanti 3D, avrà costo che dipenderà dalla pagina (item) lo indicheremo con $\omega(p)$.

Definizione 1. Dato un algoritmo ALG e un ottimo OPT, diciamo che che ALG è *c-competitivo* se esiste una costante c tale che

$$\mathrm{ALG}(\sigma) \leq c \cdot \mathrm{OPT}(\sigma) + \alpha,$$

dove α è una costante indipendente dalla richiesta σ.

Per quanto riguarda $OPT(\sigma)$, esso rappresenta il valore della soluzione ottima sulla sequenza σ. Il quale può essere rappresentato mediante un algoritmo che produce un ottimo offline, ovvero avendo come input tutta la sequenza.

Questo ci permette di confrontare la competitività del nostro algoritmo rispetto a un avversario molto più forte di esso. Infatti, conoscere l'intera sequenza aumenta notevolmente le performance. Si è parlato di avversario perché, generalmente, si va a considerare una sequenza chiamata *bad sequence* dove l'algoritmo produce il massimo degli errori per vedere la competitività. In ambito di teoria dei giochi possiamo vedere l'ottimo offline come un avversario onnisciente che riesce sempre a rispondere nel migliore dei modi. Formalmente, quindi, per definire la competitività del nostro algoritmo abbiamo bisogno di studiare anche un altro algoritmo che ci faccia le veci del nostro ottimo.

A questo punto, sul piano teorico, nasce una discussione etica: non è che l'avversario sia un po' troppo forte? Dare in input tutta la sequenza e su quella cercare l'algoritmo migliore è un vantaggio grande e questo potrebbe portare ad algoritmi che in realtà funzionano bene e sono buoni ma che confrontati con l'ottimo risultano pessimi. Per ridurre la forza dell'ottimo offline

si sono studiati diversi metodi come limitare la capacità della cache nel caso del paging. Per concludere e capire meglio come funziona la competitività degli algoritmi, analizziamo un caso semplice e molto presente in letteratura che è il problema del noleggio sci.

2.1.1 Problema del noleggio sci

Un esempio classico dell'ottimizzazione online è il problema del noleggio sci (o *ski rental problem*). Supponiamo di essere in vacanza in montagna per sciare e abbiamo due scelte: o noleggiare gli sci o comprarli. Nel primo caso paghiamo un costo unitario; nel secondo, una cifra $B > 1$ e poi non paghiamo più per l'intera vacanza. La sequenza d'ingresso sarà composta da tutti uno $\sigma = (1, 1, ..., 1)$, vogliamo gli sci tutta la vacanza, mentre quella di output sarà composta da R se l'algoritmo procede a noleggiare gli sci, B seguito da S se procede a comprali a un certo punto, con S intendo che da quel punto in poi non pago più nulla. Un esempio di sequenza potrebbe essere: $\tilde{\sigma} = (R, ..., R, B, S, ..., S)$.

Analizziamo ora meglio il problema e cercando di capirne la competitività. L'ottimo offline, vale a dire la soluzione ottima sapendo tutta la sequenza, ha valore $OPT = \min\{|\sigma|, B\}$ poiché se la vacanza dura meno del costo degli sci, procederà a noleggiare per tutta la durata, nel caso contrario invece si procederà a pagare subito l'importo degli sci per poi non pagare più.

Per quanto riguarda l'algoritmo online, abbiamo due strade possibili:

- noleggio gli sci per tutta la vacanza chiameremo A_∞ questo algoritmo;
- a un certo tempo i procedo ad acquistare gli sci, lo indichiamo con A_i.

Analizziamoli singolarmente. A_∞ non è ottimale, mi basta considerare una sequenza di lunghezza $|\sigma| \geq c \cdot B$, dato che $OPT = B$, la nostra competitività sarà $\frac{A_\infty}{B} \geq c$, che possiamo rendere grande quanto vogliamo e di conseguenza il nostro algoritmo non è competitivo.

Per quanto riguarda A_i, abbiamo che il costo dell'algoritmo è $i - 1 + B$, dato che noleggia per i primi $i - 1$ giorni e poi procede a comprare. Per dimostrarne la competitività, analizziamo tre casi: $i < B$, $i > B$ e $i = B$.

Per i primi due casi il ragionamento è simile. Supponiamo $i < B$ che posso riscrivere come $B - 1 \geq i$, fisso $|\sigma| = i$, di conseguenza OPT andrà a pagare i, e dato che A_i paga sempre $i - 1 + B$ otteniamo

$$\frac{A_i}{OPT} = \frac{i - 1 + B}{i} = 1 + \frac{B - 1}{i} \geq 2.$$

Analogamente, per il caso $i > B$, abbiamo che OPT paga B in questo caso. Quindi, riscrivendo la condizione iniziale $i \geq B - 1$, abbiamo

$$\frac{A_i}{OPT} = \frac{i - 1 + B}{B} = 1 + \frac{i - 1}{B} \geq 2.$$

Per concludere, rimane il caso $i = B$. Prendiamo $|\sigma| < B$, di conseguenza $OPT = |\sigma|$ e

$$\frac{A_i}{OPT} = \frac{2B - 1}{|\sigma|} > \frac{2B - 1}{B} = 2 - \frac{1}{B}$$

, mentre se $|\sigma| < B$ si ha $OPT = B$ e quindi

$$\frac{A_i}{OPT} = \frac{2B - 1}{B} = 2 - \frac{1}{B}.$$

Possiamo concludere che per i primi due casi abbiamo un algoritmo $2-$competitivo, mentre per l'ultimo otteniamo $\left(2 - \frac{1}{B}\right)$-competitivo e dato che $B > 1$ è migliore del caso precedente, da cui l'algoritmo A_i è $\left(2 - \frac{1}{B}\right)$-competitivo.

2.2 Problema del paging

Consideriamo due memorie, una lenta (*slow memory*) composta da un insieme di pagine $P = \{p_1, ..., p_n\}$ e una veloce (*cache*) di dimensione $k < N$ che può contenere un sottoinsieme di P. L'algoritmo deve gestire una sequenza di richieste $\sigma = (p_1, ...p_n)$, dove ogni pagina va caricata nella cache. Una volta che la cache è piena, quando arriva una nuova richiesta p_i che non è presente in essa, si genera un *page fault* e una pagina dalla cache va tolta per far spazio alla nuova. L'obiettivo è di minimizzare il numero di page fault.

Stiamo sempre considerando il problema come un problema online, dove l'algoritmo non conosce tutta la sequenza in ingresso σ ma solo le entrate

fino al tempo t. Considereremo in questo capitolo due modelli: il primo è il page fault model dove si paga 1 in caso di page fault per portare la pagina dentro la cache, il secondo e il modello di costo ad accesso completo dove paghiamo s per portare dentro la pagina, e 1 per accedere.

Iniziamo introducendo alcuni algoritmi, in particolare analizziamo la strategia che utilizzano quando avviene un page fault:

- Least Recently Used (LRU): viene rimosso l'elemento che non è stato utilizzato da più tempo;
- First In First Out (FIFO): tolgo la pagina che è stata inserita per prima;
- Flush When Full (FWF): tolgo tutte le pagine, in questo modo la cache è vuota e ricomincio;
- Last In First Out (LIFO): tolgo la pagina che è stata inserita per ultima;
- Longest Forward Distance (LFD): tolgo la pagina la cui prossima richiesta è più in là col tempo.

Si osservi che l'algoritmo LFD, a differenza degli altri, richiede la conoscenza di tutta la sequenza. È quindi un algoritmo offline.

Osservazione. Per come abbiamo posto il modello è ragionevole pensare che una maggior dimensione della cache porti a meno page fault, questo in generale non è vero, come dimostrato nella **Belady's anomaly**, che ci mostra che in FIFO aumentare la grandezza della cache può causare un maggior numero di page fault. Prendiamo in considerazione la seguente sequenza:

$$(1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5).$$

Come dimensione della cache consideriamo k = 3, 4.

Dalla tabella è evidente che FIFO con $k = 4$ commette più page fault rispetto alla versione con $k = 3$ sulla stessa sequenza in ingresso e quindi risulta evidente la Belady's anomaly.

Per studiare la competitività degli algoritmi confrontiamo l'algoritmo con un ottimo offline su una sequenza d'ingresso σ.

σ	1	2	3	4	1	2	5	1	2	3	4	5
FIFO k = 3												
Fault	×	×	×	×	×	×	×	✓	✓	×	×	✓
FIFO k = 4												
Fault	×	×	×	×	✓	✓	×	×	×	×	×	×

× = fault, ✓ = non fault

Tabella 2.1: Esempio in cui si presenta la Belady's anomaly con FIFO

2.2.1 Algoritmo LFD e ottimo offline

Iniziamo con definire un algoritmo equivalente all'ottimo offline, LFD(longest forward distance) quando è presente un page fault, va a togliere l'elemento che viene richiesto più avanti nella richiesta. Prendiamo come esempio la sequenza della tabella 1 e $k = 3$, nel momento in cui entra 4 la cache è piena e LFD deve scegliere quale elemento togliere tra 1,2,3, dato che 1 e 2 vengono richiesti nelle due richieste successive, LFD andrà a togliere 3. Mostriamo ora che è equivalente come ottimo offline.

Teorema 2.2.1. *LFD è ottimale per il paging.*

Dimostrazione. Mostro che posso modificare un algoritmo ALG per agire come LFD. Considero una richiesta σ e $i = 1, ..., |\sigma|$. Costruisco un algoritmo ALG_i tale che:

a) si comporta come ALG sulle prime $i - 1$ componenti

b) se al tempo i ho un page fault, ALG_i elimina la pagina più distante, ovvero agisce come LFD

c) $ALG_i(\sigma) \leq ALG(\sigma)$, il numero di page fault che compie ALG_i è al più quello di ALG.

Chiamo X = pagine in comune tra i due algoritmi($|X| = k-1$) e supponiamo che ALG e ALG_i, contengano a un certo punto $X \cup v$ e $X \cup u$, suppongo inoltre che agiscano allo stesso modo, tranne che quando ALG toglie v, ALG_i toglie u. Una volta che le pagine in comune tornano a essere k si identificano

e procedono uguali. Supponiamo ora che arrivi una richiesta per v prima che si siano identificati, in questo caso ALG_i compie un page fault mentre ALG no. Allo stesso tempo però se a un certo punto ALG_i elimina v al posto di u vuol dire che v è il più lontano, quindi vorrà dire che ci sarà prima una richiesta per u su cui ALG ha un page fault mentre ALG_i no e di conseguenza il punto c rimane valido. Per concludere osservo che mi basta considerare come ALG un ottimo offline OPT, ad ogni step uso la costruzione di ALG_i, quindi quando $i = 1$ ottengo OPT_1 che fa gli stessi page fault di OPT e che agisce come LFD, e ripeto questo procedimento fino a $i = |\sigma|$ ottenendo $OPT_{|\sigma|} = LFD$. □

2.2.2 Competitività degli algoritmi di marcatura

Introduco un argomento che permette di calcolare la competitività degli algoritmi ovvero la divisione in fasi.

Definizione 2. Siano k la dimensione della cache e $\sigma = \sigma_1, ..., \sigma_n$ la richiesta. Chiamiamo *fase*, che indicheremo con $\varphi_i = (\{)\sigma_{i_1}, ..., \sigma_{i_n})$, la più grande sequenza che segue φ_{i-1} che contiene al più k pagine distinte, ovvero soddisfa le due seguenti condizioni:

- $|\varphi_i| \leq k \; \forall i = 1, ..., n$,
- $|\varphi_i \cup \sigma_{i_{n+1}}| > k$,

dove $\sigma_{i_{n+1}}$ è la prima richiesta della fase φ_{i+1}.

Esempio 1. *Sia $k = 3$ e consideriamo la seguente sequenza d'ingresso:*

$$\sigma = (1, 2, 3, 1, 2, 4, 1, 3, 4, 1, 5, 6, 7).$$

La divisione in fasi per questa sequenza sarà la seguente: $\varphi_1 = (1, 2, 3, 1, 2)$, $\varphi_2 = (4, 1, 3, 4, 1)$, $\varphi_3 = (5, 6, 7)$. Osserviamo che, per come abbiamo definito la divisione in fasi, ognuna inizia con un page fault.

Costruiamo ora una categoria di algoritmi noti come *algoritmi di marcatura*. Consideriamo la divisione in fasi (Definizione 2), ad ogni pagina viene associato un bit chiamato *mark* e diremo che la pagina è *marcata* se il bit associato

vale 1, altrimenti diremo *unmark*. Ogni volta che una pagina entra nella cache, questa viene marcata (il bit associato viene messo a 1) e all'inizio di una nuova fase vengono tolti tutti i mark (i bit associati tornano a 0). Questa categoria di algoritmi soddisfa la stessa condizione: un algoritmo di marcatura non toglie mai dalla cache una pagina marcata. Alcuni algoritmi che fanno parte di questa categoria sono, per esempio, LRU e FWF.

Lemma 1. *LRU è un algoritmo di marcatura.*

Dimostrazione. Supponiamo per assurdo che LRU non sia un algoritmo di marcatura, cioè mostriamo che LRU toglie una pagina marcata in una fase. Supponiamo sempre che la dimensione della cache sia fissata a k. La pagina x entra nella cache e viene marcata. Per far sì che esca seguendo LRU, devo prima riempire la cache. Servono quindi $k-1$ pagine distinte diverse da x. Senza perdere di generalità, supponiamo siano le prime $k-1$, ovvero $p_1, ..., p_{k-1}$. Deve poi entrare un'ulteriore pagina p_i con $i > k-1$ per far sì che esca x dalla cache. Poiché la fase sarebbe composta da $\varphi = (p_1, ..., p_{k-1}, p_i, x)$ e, di conseguenza, $|\varphi| > k$ giungiamo a un assurdo per la definizione di fase. □

Esempio 2. *Entra* x, $\{x, 1, 2\} \xrightarrow{1} \{1, x, 2\} \xrightarrow{2} \{2, 1, x\} \xrightarrow{3} \{2, 1, 3\}$ *la sequenza è composta da* $\{x, 1, 2, 3\}$ *ma dato che* $|\varphi| = 4 > 3$, φ *non rispecchia la condizione di fase della Def 2.*

Lemma 2. *FWF è un algoritmo di marcatura.*

Dimostrazione. In modo simile a LRU, supponiamo per assurdo che FWF non sia un algoritmo di marcatura, ovvero a un certo punto elimini una pagina marcata. Sia x tale pagina. Per far sì che FWF elimini x, dobbiamo riempire la cache in questa fase e avere una nuova richiesta non presente nella cache; ma in questo caso la fase corrente contiene $k+1$ pagine, il che è assurdo. □

Teorema 2.2.2. *Ogni algoritmo di marcatura è k-competitivo, dove k è la dimensione della cache.*

Dimostrazione. Consideriamo la suddivisione in fasi di una sequenza σ. Dato che ogni algoritmo di marcatura non può togliere dalla cache una pagina che è mark, al più si possono commettere k page fault durante una fase. Se come ottimo offline consideriamo la soluzione dell'algoritmo LFD, nel caso migliore esso compie solamente un page fault, e non può farne di meno per come abbiamo definito la fase. Abbiamo infatti osservato che una fase inizia necessariamente con un page fault. Basta ripetere il ragionamento per il numero di fasi e si ottiene

$$\frac{MARK(\sigma)}{OPT(\sigma)} \leq k + \alpha,$$

dove α è una costante che dipende dall'ultima fase.
Sia m il numero delle fasi e sia $\tilde{\alpha}$ il numero di page fault sull'ultima fase. Otteniamo che

$$\sum_{i=1}^{m} MARK(\varphi_i) \leq m \cdot k + \tilde{\alpha}$$

e

$$\sum_{i=1}^{m} OPT(\varphi_i) \geq m,$$

da cui la tesi ridefinendo $\frac{\tilde{\alpha}}{m} = \alpha$. □

Per generalizzare il problema del paging introduciamo il modello (h, k)-*paging*, dove si ha sempre la cache di memoria fissata k per l'algoritmo di cui vogliamo studiare le performance, mentre per l'ottimo usiamo una cache di memoria $h < k$. Questo ci permette di ridurre "la forza" dell'algoritmo offline, e tramite il parametro, controllarla.

Teorema 2.2.3. *Ogni algoritmo di marcatura è $\frac{k}{k-h+1}$-competitivo rispetto al (h, k)-paging.*

Dimostrazione. Come nel caso precedente, ogni algoritmo di marcatura, non potendo togliere pagine marcate dalla cache, commette al più k page fault. Si consideri una richiesta per q; l'ottimo offline OPT ha $h - 1$ pagine diverse da q e k richieste nella fase, di conseguenza potrà compiere al massimo $k - (h - 1) = k - h + 1$ page faut. Da cui, la tesi. □

Sia $\sigma = (1, 2, 3, 1, 4, 2, 5)$, allora le due fasi saranno $\varphi_1 = (1, 2, 3, 1)$ e $\varphi_2 = (4, 2, 5)$. L'algoritmo FIFO procede a togliere la pagina che è stata inserita per prima.
Fase φ_1: $1 \to \{\dot{1}-, -\}, 2 \to \{\dot{1}, \dot{2}, -\}, 3 \to \{\dot{1}, \dot{2}, \dot{3}\}, 1 \to \{\dot{1}, \dot{2}, \dot{3}\}$
Fase $\varphi_2 : 4 \to \{2, 3, \dot{4}\}, 2 \to \{\dot{2}, 3, \dot{4}\}, 5 \to \{3, \dot{4}, 5\}$
Avendo eliminato nell'ultimo passaggio una pagina che è stata marcata, di conseguenza, FIFO non è un algoritmo di marcatura.

Teorema 2.2.4. *FIFO è k-competitivo per il paging e $\frac{k}{k-h+1}$-competitivo rispetto al (h, k)-paging.*

Dimostrazione. Per arrivare alla tesi basta dimostrare che il numero massimo di page fault che commette FIFO è k, poiché abbiamo già dimostrato che OPT ≥ 1 nel caso standard, mentre OPT $\geq k-h+1$ per il (h, k)-paging. Ad ogni step, FIFO toglie l'elemento che è stato inserito per primo nella cache, quindi il caso peggiore si verifica quando nessun elemento della fase è già presente nella cache e, dato che $|\varphi_i| \leq k \ \forall i$, abbiamo la tesi: può compiere al più k page fault. □

2.2.3 Modello di costo ad accesso completo

Consideriamo il modello di costo ad accesso completo, l'algoritmo paga un costo pari a 1 per accedere alla pagina e un costo pari a s per spostarla dalla slow alla fast memory. Sia quindi $\sigma = (p_1, ..., p_n)$ la richiesta e $\sigma = (\varphi_1, ..., \varphi_m)$ la divisione in fasi, definiamo $L(\sigma) = \frac{|\sigma|}{m}$ come la lunghezza media della fase, osservando che $L(\sigma) \geq k$.

Teorema 2.2.5. *Sia k la dimensione della cache, allora vale*

$$\frac{ALG(\sigma)}{OPT(\sigma)} \leq 1 + \frac{(k-1)s}{L(\sigma) + s}.$$

Dimostrazione. Siano $n = |\sigma|$ e $m = \frac{n}{L(\sigma)} \leq \frac{n}{k}$.
Su ogni fase ALG compie al più k page fault, ho m fasi e ad ogni page fault caricare la pagina ha costo s; in più, accedere ad un elemento di σ costa 1,

quindi in totale si ha un costo

$$ALG(\sigma) \leq kms + n.$$

Sappiamo che l'ottimo offline OPT compie almeno un page fault su ogni fase (per la definizione di fase), il quale ha sempre costo s, e abbiamo m fasi con costo di accesso pari a 1. Otteniamo quindi

$$OPT(\sigma) \geq n + ms.$$

Pertanto

$$\frac{ALG(\sigma}{OPT(\sigma)} \leq \frac{n + kms}{n + ms} = \left(n + \frac{kns}{L(\sigma)}\right) \cdot \frac{L(\sigma)}{L(\sigma)n + sn} = \frac{L(\sigma) + ks}{L(\sigma) + s}$$

$$= \frac{L(\sigma) + s}{L(\sigma) + s} \cdot \frac{ks - s}{L(\sigma) + s} \leq 1 + \frac{(k-1)s}{L(\sigma) + s}.$$

□

2.2.4 Analisi competitiva

Per concludere introduciamo un analisi competitiva per riuscire a capire quanto sia peggiore un algoritmo rispetto all'ottimo offline.

Sia ALG l'algoritmo, e σ la richiesta con $|\sigma| = m$, $F_{ALG}(\sigma)$ numero di page fault di ALG e n_{ALG}, n_{OPT} il numero di pagine nella cache di ALG e OPT.

Teorema 2.2.6. *Esiste una sequenza $\tilde{\sigma}$ tale che*

$$F_{ALG}(\tilde{\sigma}) \geq \frac{n_{ALG}}{n_{ALG} - n_{OPT} + 1} \cdot F_{OPT}(\tilde{\sigma}).$$

Dimostrazione. Definiamo la sequenza di lunghezza n_{ALG} dove ALG compie n_{ALG} page fault, e OPT ne compie $n_{ALG} - n_{OPT} + 1$. Creiamo quindi una sequenza dove le prime $n_{ALG} - n_{OPT} + 1$ richieste non stanno né in ALG né in OPT. Le restanti pagine vengono prese o nella cache di OPT o nelle

$n_{OPT} - 1$ pagine restanti. In questo modo, ALG continua a fare page fault mentre OPT no. □

Mostriamo anche che LRU è in generale il migliore algoritmo online in termini di performance.

Teorema 2.2.7.

$$F_{LRU}(\tilde{\sigma}) \leq \frac{n_{LRU}}{n_{LRU} - n_{OPT} + 1} \cdot F_{OPT}(\tilde{\sigma}) + n_{OPT}.$$

Dimostrazione. Dopo il primo accesso, LRU e OPT hanno almeno una pagina in comune. Considero una sottosequenza t di σ dove LRU compie f page fault di lunghezza al più n_{LRU}. Se LRU fa un page fault due volte sulla sequenza t, vuol dire che t contiene esattamente $n_{LRU}+1$ pagine. Un risultato analogo si ottiene anche se viene richiesta la pagina prima dell'inizio della sequenza t.
Quindi LRU compie f page fault e OPT $f - n_{OPT} + 1$, da cui basta dividere σ in $\sigma_0, \sigma_1, \ldots$ in modo che per ogni $i \geq 1$ LRU compia esattamente $f = n_{LRU}$ page fault. Il termine correttivo viene fuori dal fatto che sulla prima sequenza LRU compie f_0 page fault, mentre OPT al più $f_0 - n_{OPT}$. □

Teorema 2.2.8. *LIFO e LFU non sono competitivi.*

1) Basta considerare una sequenza dove le ultime due richieste vengono ripetute continuamente, basta prendere $k = 3$ e
$\sigma = (1, 2, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, ...)$, si nota facilmente che LIFO$(\sigma) = |\sigma|$ dato che quando entra 3 lo toglie per far entrare 4, poi toglie 4 per far entrare 3 e continua a fare page fault, mentre OPT$(\sigma) = 4$ cioè (k+1).

2) Per LFU basta prendere una sequenza dove chiedo i primi $k-1$ elementi l volte e gli ultimi due alternati $l - 1$ volte, in modo che ogni volta che entrano si scartano a vicenda e producono un page fault su ogni entrata. Un prototipo di sequenza è $\sigma = (p_1^l, ..., p_{k-1}^l, (p_k, p_{k+1})^{l-1})$, LFU compie page fault ogni volta dopo le $l(k-1)$ prime richieste, mentre OPT commette solo un page fault.

Capitolo 3

Problema del batch paging

In questo capitolo viene esteso il problema del paging precedentemente analizzato a una versione multi-variabile, dove in input si possono avere più di una pagina per volta.

Consideriamo una cache di dimensione k fissata, e una richiesta $\sigma = (\sigma_1, ..., \sigma_t)$ di pagine in entrata, ogni entrata $\sigma_i = (p_1, ..., p_n)$ dove p_i indica l'i-esima pagina richiesta, e chiediamo che $|\sigma_i| \leq k$ per ogni $i = 1, ..., n$.

Per semplicità, consideriamo un costo unitario, ovvero l'algoritmo andrà a pagare 1 ogni volta che c'è un page fault, ovvero ogni volta che la pagina richiesta non è presente nella cache, mentre 0 se è già presente. Per ogni richiesta, si possono quindi avere tre casi:

- la richiesta σ_i ha tutti gli elementi presenti nella cache, costo 0;
- la richiesta σ_i non ha nessun elemento presente nella cache, pago $|\sigma_i|$;
- la richiesta σ_i è parzialmente presente nella cache, pago il costo delle pagine non presenti nella cache.

La prima modifica sostanziale è il concetto di fase, per il paging avevamo che ogni fase inizia con una nuova richiesta e può contenere al massimo k richieste distinte. In questa versione, ogni richiesta può ricevere più di una pagina per volta, quindi va modificata.

Definizione 3. Siano σ una sequenza di richieste e k la dimensione della cache. Ogni *fase* $\varphi_i = (\sigma_{i_1}, ..., \sigma_{i_n})$, deve rispettare $|\bigcup_{j=1}^{n} \sigma_{i_j}| \leq k$, ovvero la

cardinalità dell'unione delle richieste non supera la dimensione della cache. E la successiva fase inizierà quando una richiesta produce un page fault, ovvero vale $|\varphi_i \cup \sigma_{i_{n+1}}| \geq k+1$.

Esempio 3. *Sia* $\sigma = (\{1,5\},\{3,5\},\{2,1\},\{2,3,4\},\{1,3\},\{1,2\},\ldots)$*, e consideriamo una cache di dimensione* $k = 4$*, la prima fase sarà composta da* $\{1,5\},\{3,5\},\{2,1\}$*. Se, infatti, consideriamo la cardinalità dell'unione, risulta* $|\{1,2,3,5\}| = 4$*. La fase successiva sarà* $\{2,3,4\},\{1,3\},\{1,2\}$*, dato che* $\{1,2\}$ *non produce page fault, e la cardinalità rimane minore di* k*.*

Osservazione. La definizione di fase è ben posta poiché $|\sigma_i| \leq k\ \forall i = 1,...,t$*. Possiamo anche omettere questa restrizione e nel caso in cui una richiesta abbia cardinalità superiore a* k*, la suddividiamo nel modo seguente. Sia* $\sigma_i = (p_{i_1},...,p_{i_n})$ *con* $|\sigma_i| > k$*, creiamo una nuova sotto-richiesta* $\sigma_{i_1} = \emptyset$*. Ad ogni passo, aggiungiamo una pagina di* σ_i *a partire dalla prima,* $\sigma_{i_1} \cup p_1$*, e iteriamo finché* $|\sigma_{i_1}| = k$*. Al passo successivo, ricreiamo* σ_{i_2} *e continuiamo analogamente. In questo modo non salta la definizione di fase, ma questa richiesta che produceva più page fault della dimensione della cache ora è divisa e agisce in due fasi.*

3.1 Algoritmo LFD e ottimo offline

Sempre nel setting descritto nei precedenti capitoli abbiamo che la richiesta di ingresso è $\sigma = (\sigma_1,...,\sigma_n)$ dove ogni $\sigma_i = (p_{i_1},...,p_{i_n})$, con $p_1,...,p_n$ pagine. Un algoritmo che vogliamo prendere in considerazione è LFD (Last Forward Distance), che come nel caso del paging tradizionale in caso di page fault, toglie la pagina la cui richiesta sia più lontana. Nel nostro caso andiamo a vedere dove le pagine presenti nella cache appaiono nelle richieste successive ed eliminiamo quella che viene chiesta più in là.

Esempio 4. *Riprendiamo l'esempio di inizio capitolo e supponiamo che la cache sia composta da quattro elementi e che sia già piena:*

$$cache = \{1,2,3,4\}$$

Entra la prima richiesta $(1, 5)$. *In questo caso 1 è già presente nella cache, mentre per 5 abbiamo un page fault. Quale elemento togliamo? Individuiamo quello con la distanza maggiore dalla prossima richiesta:*

1 appare nella richiesta 3

2 appare nella richiesta 3

3 appare nella richiesta 2

4 appare nella richiesta 4

Di conseguenza sostituiamo 4 con 5, e la nuova cache diventa $(1, 2, 3, 5)$. *Le richieste successive,* σ_2 *e* σ_3 *non producono page fault. Ripetiamo il ragionamento per la richiesta* $\sigma_4 = (2, 3, 4)$. *La pagina 4 non risulta presente nella cache, quindi andiamo a trovare la richiesta più lontana:*

1 appare nella richiesta 5

2 appare nella richiesta 5

3 appare nella richiesta 6

5 non appare più nella richiesta

Quindi sostituiamo 5 con 4.

Osservazione. Osserviamo che LFD si comporta nell'esempio come un ottimo offline, infatti produce un page fault per fase. Se consideriamo la partizione in fasi dell'Esempio 3, ognuna inizia con un page fault (per la definizione di fase) e abbiamo un solo errore. L'idea infatti è che anche nel caso del PIUC abbiamo che LFD si comporta come un ottimo offline.

Teorema 3.1.1. *LFD è un algoritmo ottimo offline.*

Dimostrazione. Dato un ottimo offline (OPT) riusciamo sempre a costruire un algoritmo ALG che si comporta come OPT fino alla richiesta σ_{i-1} e in caso di page fault agisce come LFD. Quindi che soddisfa queste tre proprietà:

I) ALG_i agisce come OPT fino alla richiesta σ_{i-1}

II) Se la richiesta σ_i, produce un page fault e scegliamo le pagine da sostituire seguendo LFD

III) $ALG_i(\sigma) \leq OPT(\sigma)$

Mostriamo che è sempre possibile costruire questo algoritmo e che non produce più page fault di OPT.

Chiamiamo $X = \{\textit{pagine in comune tra } ALG_i \textit{ e } OPT\}$. Supponiamo quindi che ALG_i abbia nella cache le pagine $X \cup u$, mentre per OPT abbiamo $X \cup v$. Per semplicità iniziale supponiamo che $u = (p_{u_1}, p_{u_2})$ e $v = (p_{v_1}, p_{v_2})$. ALG_i toglie u quando OPT toglie v, e supponiamo che almeno una delle pagine sia diversa, questo possiamo farlo senza perdere generalità, poiché nel caso in cui siano entrambe uguali avrei che OPT e ALG_i si comportano allo stesso modo. Abbiamo quindi due casi, o non hanno pagine in comune o ne hanno una. Nel secondo caso, senza perdere in generalità, supponiamo che sia la prima, cioè $p_{u_1} = p_{v_1}$. Ci basta ridefinire $\bar{X} = X \cup \{p_{u_1}\}$ e si ottiene lo stesso caso del paging standard. Ora supponiamo che entrambe le pagine siano diverse, e che nella sequenza venga richiesto v prima che le cache siano identificate, in questo caso ALG_i produce due page fault mentre OPT, che contiene v, non commette page fault. D'altro canto, per come abbiamo definito l'algoritmo ALG_i, esso agisce da LFD, quindi vuol dire che se entrambe le pagine contenute in v vengono tolte dalla cache, esse sono anche quelle che hanno distanza maggiore. Quindi, vuol dire che nella richiesta σ c'è almeno una richiesta per ogni pagina contenuta in u prima di v. Ricordando che u non è presente nella cache per OPT, di conseguenza anche OPT produce lo stesso numero di page fault di ALG_i. Per il principio di induzione, supponiamo che sia possibile costruire ALG_i che soddisfa le ipotesi I, II e III per sequenze di lunghezza al più $k-1$ e dimostriamo che possiamo costruire l'algoritmo anche per sequenze di lunghezza k. Come nel caso precedente, dobbiamo analizzare solo la situazione in cui u e v hanno tutte pagine distinte. Nei casi in cui almeno una pagina è in comune, è sufficiente ridefinire le pagine in comune $\tilde{X} = X \cup \{p_{u_1}, ..., p_{u_{k-1}}\}$ e otteniamo il caso del paging standard o uno dei precedenti k-1 casi. Se la richiesta ha $|u| = k$, vuol dire che $X = \emptyset$ e quindi che a un certo punto OPT e ALG_i

si troverebbero ad avere due cache completamente distinte. Ancora come il caso precedente, dato che ALG_i agisce come LFD abbiamo che, se le cache non vengono unificate prima della richiesta dell'intera sequenza v, ancora abbiamo che ALG_i produce un page fault. Ma essendo che ALG_i si comporta come LFD, tutte le pagine dentro ALG_i sono le ultime ad essere richieste e quindi u viene richiesto prima di v, quindi anche in questo caso commettono lo stesso numero di page fault.

Per concludere mostriamo che otteniamo un ottimo offline, basta ripetere questa claim tante volte quante sono il numero delle richieste $|\sigma|$. In questo modo, generiamo una sequenza $OPT_1, OPT_2, \ldots$ che agisce come ottimo e come LFD, da cui la tesi. □

3.2 Algoritmo LRU e competitività degli algoritmi di marcatura

Cerchiamo adesso di analizzare il rapporto competitivo dei vari algoritmi, partendo da LRU e estendendo il ragionamento ai mariking algoritm. Su una singola richiesta si comporta come nel paging standard, arriva una richiesta $\sigma_i = (p_1, ..., p_t)$, valuto se le pagine sono nella cache, nel caso non fossero presenti sostituisco le pagine usate meno di recente, e procedo con le richieste successive.

Grazie alla nuova definizione di fase possiamo estendere i risultati già noti per il paging a PIUC

Teorema 3.2.1. *LRU è k-competitivo.*

Dimostrazione. Mostriamo che LRU compie al più k page fault. Ad ogni step, nel caso di un page fault, LRU espelle la pagina che è stata utilizzata per ultima, e dato che la cardinalità dell'unione delle fasi è al più k, può compiere al più k page fault. Quindi diciamo che il worst case è sempre lo stesso del caso standard del paging, dove come input si ha una sequenza di richieste in ingresso tale che ogni richiesta produce un page fault.

Per quando riguarda OPT incorre al minimo in un page fault che è la richiesta nuova di ogni fase, come nel caso del paging standard. Da cui otteniamo che la competitività è di k.

Dividiamo la sequenza delle richieste in fasi, $\sigma = (\varphi_1, \varphi_2, ...)$. Ogni $\varphi_i = (\sigma_{i_1}, ..., \sigma_{i_n})$ soddisfa due condizioni:

- $|\varphi_i| \leq k$
- $|\varphi_i \cup \sigma_{i+1_1}| \geq k + 1$

Di conseguenza, il costo massimo che LRU può pagare per ciascuna fase è k. □

Possiamo estendere quanto visto per LRU ai generici algoritmi di marcatura. Di fatto, ognuno di questi algoritmi non espelle mai una pagina marcata, e quindi, giocando sul fatto che in ogni fase ho al massimo k pagine distinte, non si possono commettere più di k page fault.

Teorema 3.2.2. *Ogni algoritmo di marcatura è k-competitivo.*

Dimostrazione. Dato che le pagine marcate non possono essere sostituite, in ogni richiesta possono essere prodotti page fault per al massimo la cardinalità dell'unione delle pagine, e quindi si possono produrre al massimo k page fault. Di conseguenza otteniamo lo stesso risultato di LRU. □

3.3 Analisi di competitività dell'algoritmo FIFO

Analogamente a quanto fatto per i precedenti due algoritmi, riusciamo a mostrare che anche i risultati ottenuti per FIFO visti per il paging standard possono essere estesi al batch paging a costo unitario.

Teorema 3.3.1. *FIFO è k-competitivo.*

Dimostrazione. La bad sequence è simile a quella del paging classico dove suppongo che nella fase vengano commessi k page fault. Non se ne possono

commetterne di più k, in quanto nel caso in cui FIFO eliminasse un elemento della cache che viene richiesto successivamente nella stessa fase, vorrebbe dire che la cardinalità dell'unione della fase supera k, il che è assurdo.

Per quanto riguarda OPT, chiamiamo p_i l'ultima pagina inserita nella fase, dato che la fase successiva inizia quando c'è una pagina che produce un page fault, ci deve essere almeno una pagina che non è presente nella cache e quindi OPT paga 1. Quindi il rapporto competitivo rimane k. □

3.4 Modello di costo ad accesso completo

Nel modello di costo ad accesso completo si ha una modifica rispetto al paging standard, dove il costo di caricamento della pagina dalla slow alla fast memory è $s \geq 1$. Viene pagato quindi un costo pari a 1 per accedere e pari a s per caricare. Sia quindi $\sigma = \sigma_1, ..., \sigma_t$ la sequenza di richieste come nel caso precedente, dove per ogni i scriviamo $\sigma_i = \{p_1, ..., p_n\}$. Dividiamo la richiesta in fasi come nella Definizione 3, $\sigma = \varphi_1, ..., \varphi_p$, e definiamo la lunghezza media della fasi $L(\sigma) = \frac{\sum_{i=1}^{t} |\sigma_i|}{p}$, dove p indica il numero delle fasi.

Osservazione. $L(\sigma) \geq k$. Se così non fosse vorrebbe dire che in media φ_i ha $k-1$ elementi, e quindi si potrebbe costruire una successione di fasi in modo che ogni fase abbia ogni elemento distinto e che siano al più $k-1$. Sia ora σ_{i+1} il primo elemento della richiesta φ_{i+1}, avremmo che $|\varphi_i \bigcup \sigma_{i+1}| = k$, che è assurdo per la Definizione 3.

Teorema 3.4.1. *Sia ALG un algoritmo con cache size k, e sia σ la richiesta. Allora*

$$\frac{ALG(\sigma)}{OPT(\sigma)} \leq 1 + \frac{(k-1)s}{L(\sigma)+s}.$$

Dimostrazione. Chiamiamo $n = \sum_{i=1}^{t} |\sigma_i|$. In ogni fase ALG compie al più k page fault, quindi il costo totale dell'algoritmo è $n + kps$, mentre OPT compie almeno un page fault e quindi il suo costo risulta $ps + n$. Ricordando che

$$p = \frac{n}{L(\sigma)},$$

otteniamo

$$\frac{ALG(\sigma)}{OPT(\sigma)} \leq \frac{n+kps}{n+sp} = \frac{nL(\sigma)+ks}{nL(\sigma)+sn} = 1 + \frac{(k-1)s}{L(\sigma)+s}.$$

Inoltre, ricordando l'osservazione $L(\sigma) \geq k$, otteniamo :

$$\frac{ALG(\sigma)}{OPT(\sigma)} \leq 1 + \frac{(k-1)s}{k+s} = \frac{k(s+1)}{k+s}.$$

□

Capitolo 4

Problema del k-server

In questa sezione analizziamo il *problema del k-server* come generalizzazione del paging per introdurre un modello di algoritmo che meglio rappresenti il problema della sostituzione delle cartucce nella stampa 3D per la medicina.

Il problema del k-server fu introdotto da Manasse, McGeoch e Sleator nel 1988 [8] ed è considerato uno dei problemi più importanti e aperti della teoria degli algoritmi competitivi.

Sia $\mathcal{M} = (M, d)$ uno spazio metrico, con M insieme dei punti e d distanza ovvero che soddisfa le seguenti proprietà

- $d(x, y) = 0 \iff x = y \ \ \forall x, y \in M$ *(Identità degli indiscernibili)*;
- $d(x, y) = d(y, x) \ \ \forall x, y \in M$ *(Simmetria)*;
- $d(x, z) \leq d(x, y) + d(y, z) \ \ \forall x, y, z \in M$ *(Disuguaglianza triangolare)*.

L'obiettivo è gestire k server mobili, $S = s_1, ..., s_k$, in uno spazio metrico, servendo una sequenza di richieste che arrivano online minimizzando la distanza totale percorsa dai server.

Sia dunque $\sigma = \sigma_1, ..., \sigma_n$ la mia richiesta, con σ_i un punto nello spazio M, e A_0 la configurazione iniziale. Una richiesta è soddisfatta se uno dei k server si trova su σ_j. Per servire σ_j, l'algoritmo deve:

1. Scegliere un server $i \in \{1, \ldots, k\}$;
2. Spostarlo dal punto corrente $s_i^{(j-1)}$ al punto σ_j;

3. Aggiornare la posizione del server: $s_i^{(j)} \leftarrow \sigma_j$.

Di fatto, i server servono la richiesta σ_j seguendo le configurazioni descritte sopra che chiamiamo A_j. Al passo j, il costo per servire σ_j è dato dalla distanza tra le configurazioni A_{j-1} e A_j, quindi $d(A_{j-1}, A_j)$. Il costo complessivo dell'algoritmo $\mathcal{A}$, data la configurazione iniziale e la richiesta, sarà pari a

$$C_{\mathcal{A}}(A_0, \sigma) = \sum_{j=1}^{n} d(A_{j-1}, A_j).$$

Il contesto rimane quello dei problemi di ottimizzazione online, di conseguenza, A_j dipende solo dalla configurazione iniziale A_0 e dalle richieste fino a j, $\sigma_1, ..., \sigma_j$. Chiamiamo $OPT(A_0, \sigma)$ la configurazione ottima avendo a disposizione tutta la richiesta. L'algoritmo $\mathcal{A}$ risulta c-competitivo se

$$C_{\mathcal{A}}(A_0, \sigma) \leq c \cdot OPT(A_0, \sigma) + \alpha.$$

Se si considera uno spazio metrico con al più k punti, ogni server ricopre una richiesta; di conseguenza, il rapporto competitivo è uguale a 1: l'algoritmo ricopre tutti i punti e non avrà più bisogno di spostarli. Di interesse maggiore è il caso in cui $|M| > k$. La prima osservazione riguarda il teorema dimostrato da Manasse, Mc Geoch e Sleator, il quale ci permette di determinare un lower bound per il problema del k-server.

Teorema 4.0.1. *Sia $\mathcal{M} = (M, d)$ uno spazio metrico, $|M| > k$, allora k è un lower bound del rapporto competitivo per ogni problema del k-server online.*

La dimostrazione, per non appesantire la trattazione, è riportata integralmente nell'Appendice A.

Dato quindi il teorema, è ragionevole supporre la seguente congettura.

Congettura del k-server: Per ogni spazio metrico esiste un algoritmo online con rapporto competitivo k.

In questo capitolo vediamo due algoritmi per il problema del k-server: il primo è un approccio greedy, dove semplicemente ad ogni step viene tolto

l'elemento con costo minore; il secondo è un algoritmo dove $|M| = k + 1$: in questo caso si riesce a raggiungere il rapporto competitivo k e dunque dimostrare la congettura.

Successivamente generalizziamo il problema al paging pesato, che rispecchia il problema del cambio delle cartucce delle stampanti 3D per la medicina analizzando due algoritmi, sempre una versione greedy, il *Balance* e *GreedyDual*, una generalizzazione di LRU per la quale si riesce a soddisfare la congettura.

4.1 Algoritmi online

Definiamo un nuovo tipo di avversario che ci servirà per le dimostrazioni successive.

Definizione 4. Chiamiamo l'avversario *lazy* se soddisfa le seguenti condizioni:

- sposta un server solo se ha una richiesta per quel punto e non ha già un server in quella posizione;
- sposta solo un server per volta.

4.1.1 Algoritmo Greedy

Come primo approccio al problema del k-server, analizziamo un algoritmo greedy. In questo caso, intendiamo un algoritmo che minimizzi il costo per singola richiesta, ovvero, nel momento di un page fault, va a togliere l'elemento che risulta meno distante tra tutti quelli presenti nella cache.

Mostriamo che questo algoritmo non può essere competitivo. Basta considerare una richiesta per quattro punti con cache di dimensione $k = 3$. Prendiamo tali punti A, B, C, D come in Figura 4.1, ovvero disposti in modo tale che

$$d(i, j) << d(i, D) \quad \forall i, j \in \{A, B, C\}.$$

Figura 4.1: Esempio di disposizione dei punti che determina la non competitività dell'algoritmo Greedy per il problema del k-server.

Come istanza basta considerare una qualsiasi sequenza di richieste del tipo $\sigma = D, A, B, C, A, B, C, A, B, C, \ldots$. In questo modo si ha che una volta entrato C, l'algoritmo procederà ad eliminare A o B, ma mai D producendo quindi un page fault ad ogni richiesta successiva, dato che D è quello che dista più rispetto agli altri e quindi non uscirà mai.

D'altro canto un algoritmo ottimo procederà nel modo seguente: al primo page fault procede ad eliminare D per poi non aver più page fault. Questo ci mostra come questo algoritmo non sia competitivo.

4.1.2 Algoritmo Balance

Per concludere questa parte introduttiva sugli algoritmi online per il problema del k-server, analizziamo un caso interessante che soddisfa la congettura introdotta precedentemente in uno spazio metrico con $|M| = k + 1$. Siano $s_1, ..., s_k$ i server. L'algoritmo Balance è il seguente:

- per ogni server s_i teniamo traccia della distanza D_i percorsa da esso fino a quel momento;
- nel momento in cui arriva una richiesta σ_i per uno dei server, scegliamo s_i in modo che minimizzi la distanza $D_i + d(s_i)$.

Teorema 4.1.1. *Balance è k-competitivo per il problema del k-server nel caso* $|M| = k + 1$.

Questo risultato è dimostrato nel testo: *Online Computation and Competitive Analysis* di Borodin ed El-Yaniv [1].

Osservazione. L'algoritmo non è competitivo nel caso $|M| > k + 1$. Prendiamo due server, detti *azzurro* e *verde*, e quattro punti nello spazio a, b, c, d

collegati come in Figura 4.2 con distanze m e M. Si consideri la sequenza di richieste $\sigma = (a, b, c, d, a, b, c, d, ...)$. All'inizio, arriva una richiesta per a, quindi bisogna spostare uno dei due server, dato che quello azzurro percorre solo M, mentre quello verde $m + M$, allora verrà spostato il primo e aumentata la sua distanza percorsa, indicata con D_1, di M.

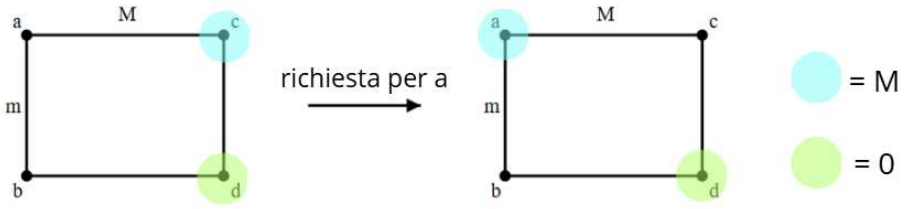

Figura 4.2: Esempio con due server ($k = 2$) e quattro punti - Spostamento del server azzurro da parte di Balance in seguito alla prima richiesta (a).

Successivamente arriva una richiesta per b. Ora abbiamo due scelte possibili, spostare il server azzurro o quello verde, percorrendo m per il primo o M per il secondo. Dato che $D_1 + m = M + m > D_2 + M = M$, di conseguenza spostiamo il server verde come in Figura 4.3.

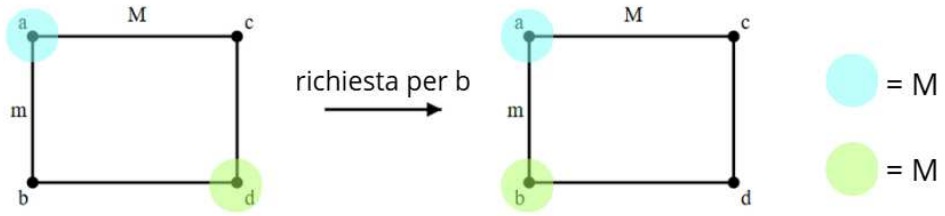

Figura 4.3: Esempio con due server ($k = 2$) e quattro punti - Spostamento del server verde da parte di Balance in seguito alla seconda richiesta (b).

Ora, dato che la configurazione è uguale a quella iniziale, si ripercorrono esattamente gli stessi spostamenti percorsi nei due casi prima, in questo modo

abbiamo che ad ogni richiesta il costo è sempre M. Per l'ottimo invece il primo caso procede esattamente come Balance: tra la scelta di postare azzurro o verde percorre verde perché il costo è minore. Differentemente, quando si trova nella seconda configurazione, andrà a spostare ancora il server azzurro (vedi Figura 4.4), e così fino alla fine della sequenza. In questo modo abbia-

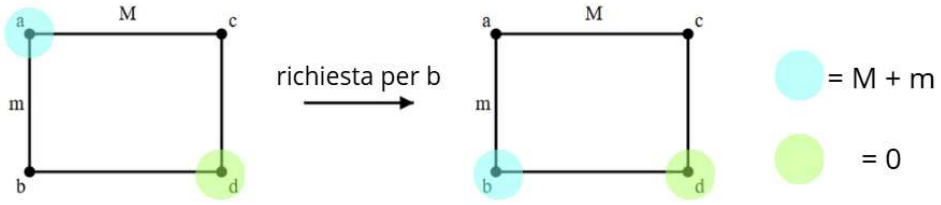

Figura 4.4: Esempio con due server ($k = 2$) e quattro punti – Spostamento del server azzurro nella soluzione ottima in seguito alla seconda richiesta (b).

mo che la soluzione ottima costa $M + m \cdot |\sigma|$, mentre Balance paga $M \cdot |\sigma|$ e di conseguenza il rapporto competitivo è $\frac{M}{m}$, quindi non potrà mai essere competitivo per arbitrarietà di m e M.

4.2 Problema del paging pesato

Il problema del *paging pesato* (o *weighted paging*) rappresenta un'estensione fondamentale del classico problema del paging, analizzata nel contesto più ampio del problema del k-server visto come un problema assimetrico.

Ogni pagina ha un costo di caricamento/estrazione fissato $\omega(p_i)$, e l'obiettivo rimane quello di minimizzare i costi totali. Nel contesto del problema del k-server la distanza perde la proprietà di simmetria, infatti siano $x, y \in M$, $d(x, y) = \omega(y)$ mentre $d(y, x) = \omega(x)$, e in generale $\omega(y) \neq \omega(x)$.

In generale il problema del problema del k-server asimmetrico è complesso e non è possibile in generale dimostrarne la competitività, e sebbene il paging pesato sia formalmente un problema di k-server in uno spazio metrico asimmetrico, esso rappresenta una struttura favorevole di asimmetria; infatti nel

paging pesato il costo di spostamento verso un nodo x è determinato esclusivamente dal peso $\omega(x)$ della pagina di destinazione, indipendentemente dal punto di partenza del server.

Questo ci permette di dimostrare la competitività di alcuni algoritmi, in particolare analizzeremo i seguenti algoritmi:

- **Greedy**: Come nel caso simmetrico andiamo a rimuovere ad ogni step la pagina con il costo più alto.
- **Balance**: È l'algoritmo di riferimento per questo problema. Esso decide quale server muovere basandosi sul costo totale accumulato da ciascun server, cercando di "bilanciare" il lavoro svolto.
- **GreedyDual**: Rappresenta un'ulteriore evoluzione che generalizza Balance e LRU, tenendo conto sia del peso sia del tempo di inserimento.

4.2.1 Algoritmo Greedy

Richiamiamo il problema del *paging pesato*. Consideriamo una cache di dimensione k fissata e una sequenza di richieste $\sigma = (\sigma_1, ..., \sigma_t)$ di pagine in entrata, che indichiamo con $\sigma_i = (p_1, ..., p_n)$, dove le p_i sono le pagine richieste. Chiediamo inoltre che $|\sigma_i| \leq k$ per ogni $i = 1, ..., n$. Ogni pagina p_i ha costo di rimozione fissato pari a $\omega(p_i)$.

Di fronte a una richiesta σ_i, l'algoritmo Greedy agisce nel modo più intuitivo, ovvero eliminando le pagine il cui costo di rimozione è minore. L'algoritmo Greedy agisce nel modo seguente:

- conta quante pagine della richiesta σ_i non sono presenti nella cache;
- cerca tra le pagine della cache quelle con il peso minore e le toglie per inserire le nuove.

Il problema di questo algoritmo greedy è che apparentemente minimizza la richiesta, ma senza preoccuparsi del futuro. Infatti, basta una qualsiasi sequenza di richieste in cui le pagine sono sempre le stesse, ma alternate, producendo page fault ad ogni richiesta, come vediamo nell'esempio seguente.

Esempio 5. *Supponiamo che la cache sia composta da* $\{1, 2, 3, 6\}$ *e che* $\omega(3) \geq \omega(6) > \omega(i)$ *con* $\forall i \neq 3, 6$. *Consideriamo come richiesta:*

$$\sigma = ((4, 5), (1, 2), (4, 5), (1, 2), (4, 5), \dots)$$

Durante la prima richiesta $\sigma_1 = (4, 5)$*, l'algoritmo produce due page fault e di conseguenza le pagine che verranno tolte dalla cache sono le due pagine con il costo minore, ovvero* 1 *e* 2*. La richiesta successiva si comporta nello stesso modo, entra* $\sigma_2 = (1, 2)$*, comporando due page fault e di conseguenza bisogna togliere due pagine, che saranno ancora* $(4, 5)$*. Si è così costruita una sequenza che genera page fault ad ogni ingresso, a differenza dell'ottimo che toglie subito* $3, 6$ *e poi non produce più page fault sull'intera sequenza.*

4.2.2 Algoritmo Balance

L'algoritmo agisce nel seguente modo: ad ogni server s_i associo S_i che tiene traccia di tutti i costi di cui il server si è fatto carico. Ad ogni nuova richiesta l'algoritmo procede nel modo seguente:

- se il punto della richiesta è già occupato da un server, allora l'algoritmo non muove alcun server;
- se il punto della richiesta non è coperto da un server, allora l'algoritmo muove il server s_α in modo che $S_\alpha = min\{S_1, ..., S_k\}$.

Teorema 4.2.1. *Balance è k-competitivo.*

Dimostrazione. Siano BAL l'algoritmo Balance e ADV il suo avversario. Assumiamo, senza perdere generalità, che l'avversario sia lazy e che non ci siano richieste per punti già occupati. Chiamiamo W_F la somma dei pesi dei punti finali e W_{NF} la somma dei pesi di tutti i punti esclusi quelli finali. Di conseguenza $ADV = W_F + W_{NF}$. Inoltre sia $T = min\{S_1, ..., S_k\}$ prima dell'ultima richiesta servita e $\tilde{T}$ lo stesso valore all'ultima richiesta. Per la dimostrazione abbiamo bisogno di due lemmi: il primo ci garantisce che il peso totale meno il peso attuale di un server non supera mai il minimo

globale; il secondo afferma che il valore finale del minimo lavoro accumulato è limitato dalla somma dei pesi dei punti "non finali" dell'avversario.

Lemma 3. $S_i - \omega(s_i) \leq T \quad \forall i = 1, ..., k.$

Lemma 4. $\tilde{T} \leq W_{NF}.$

Il Lemma 3 ci dice che $S_i - \omega(s_i) \leq \tilde{T}$ e quindi che $S_i - \tilde{T} \leq \omega(s_i)$. Allora otteniamo

$$\begin{aligned} BAL &= \sum_{i=1}^{k} S_k = \sum_{i=1}^{k} (S_k - \tilde{T} + \tilde{T}) = \sum_{i=1}^{k} (S_k - \tilde{T}) + k \cdot \tilde{T} \\ &\leq \sum_{i=1}^{k} \omega(s_i) + k \cdot \tilde{T} = W_F + k \cdot W_{NF} \leq k(W_F + W_{NF}) \\ &= k \cdot ADV. \end{aligned}$$

□

4.2.3 Algoritmo GreedyDual

L'algoritmo GreedyDual è una strategia online che generalizza l'algoritmo Balance per il paging pesato. L'idea è quella di unire la gestione dei pesi con la logica dell'utilizzo tipica di LRU.

Per questo algoritmo abbiamo bisogno di due contatori dinamici, H e L. Inizialmente settiamo $L = 0$. Supponiamo arrivi una richiesta σ_i, allora abbiamo due casi:

- se la richiesta è già presente nella cache (analogamente se la cache è vuota), aggiorniamo $H(\sigma_i) = L + \omega(\sigma_i)$;
- nel caso in cui la richiesta produca un page fault, settiamo $L = \min_{p \in C} H(p)$ dove C è l'insieme delle pagine nella cache, e $H(\sigma_i) = L + \omega(\sigma_i)$.

In questo modo teniamo conto sia del peso delle pagine sia del loro utilizzo.

Esempio 6. *Prendiamo in considerazione tre punti* A, B, D *di peso* $\omega(A) = 3, \omega(B) = 5, \omega(D) = 12$. *Consideriamo la sequenza di richieste*

$$\sigma = (D, B, A, B, A, B, \ldots)$$

L'algoritmo carica D *e* B *assegnando* $H(D) = 12$ *e* $H(B) = 5$. *Dopo di che, arriva la richiesta per* A, *quindi l'algoritmo setta* $L = \min\{12, 5\} = 5$, *inserisce* A, *al posto di* B *e calcola* $H(A) = L + \omega(A) = 8$.

Ora arriva la richiesta per B. *Dato che* $L = \min\{12, 8\} = 8$, *l'algoritmo toglie* A *per inserire* B, *aggiornando* $H(B) = 13$. *In questo modo, alla nuova richiesta per* A, *vale* $L = 12$ *e viene tolto* D *dalla memoria, non producendo più page fault sulla sequenza ed evitando il problema che si era presentato con l'algoritmo Greedy.*

Cao e Irani hanno dimostrato la competitività di questo algoritmo [2].

Teorema 4.2.2. *GreedyDual è* k*-competitivo.*

L'intuizione alla base di GreedyDual è quella di generalizzare due degli algoritmi che sono risultati i migliori per il paging e il paging pesato, ovvero LRU e Balance. Infatti:

- **LRU**: se poniamo il peso $\omega(s) = 1$ il meccanismo dei crediti, allora GreedyDual funziona allo stesso modo;
- **Balance**: se ignoriamo le richieste dei server già serviti, GreedyDual agisce esattamente allo stesso modo.

Capitolo 5

Problema del batch paging pesato con protezione

In questa sezione introduciamo una nuova variante del *batch paging* illustrato nel Capitolo 3 del *paging pesato* presentato nella Sezione 4.2 al fine di ottenere un modello e degli algoritmi che possano essere utilizzati per affrontare il problema del cambio delle cartucce delle stampanti 3D per la medicina.

Ciò che caratterizza questa variante, che diremo *batch paging pesato con protezione*, rispetto al paging classico è:

- ogni richiesta σ_t consiste di un insieme di pagine $\{p_1, \ldots, p_n\}$ che devono essere presenti contemporaneamente nella cache;
- il costo c_{ij} di un page fault dipende sia dalla pagina scaricata i che dalla pagina caricata j.

Nelle sezioni seguenti, vengono analizzate le modifiche apportate agli algoritmi LRU, FIFO e GreedyDual per adattarli al batch paging pesato con protezione. Successivamente viene presentata una modellizzazione del problema mediante un modello di Programmazione Lineare Intera Mista (MILP), con l'obiettivo di determinarne una soluzione ottima (offline).

Poiché gli algoritmi sono stati adattati al caso delle cartucce delle stampanti 3D, nel seguito si farà riferimento ai materiali anziché alle pagine e alla stampante anziché alla cache.

5.1 LRU-protetto

Ogni volta che si verifica un page fault, l'algoritmo LRU rimuove dalla cache la pagina il cui utilizzo è meno recente.

Nel caso batch, ogni richiesta è costituita da un insieme di materiali. Di conseguenza, è necessario introdurre un meccanismo di protezione degli elementi già presenti nella stampante. In particolare, durante l'elaborazione di una richiesta, un materiale appartenente al batch e già presente nella stampante non può essere rimosso per fare spazio a un altro della stessa richiesta. Questa modifica permette di modellare correttamente il problema e costituisce un'estensione dell'algoritmo LRU al caso batch. Poiché tutti i materiali appartenenti a una stessa richiesta sono richiesti nello stesso istante, non avrebbe senso rimuovere un materiale già richiesto per reinserirlo immediatamente durante l'elaborazione della medesima richiesta.

La versione LRU-protetto dunque richiede due fasi:

- Nella prima fase verifica quali materiali richiesti siano già presenti all'interno della stampante e li protegge dalla rimozione.
- Nella seconda fase, l'algoritmo si comporta come LRU. In particolare, a ogni page fault vengono rimossi i materiali, tra quelli non protetti, il cui utilizzo risale al momento meno recente, rendendo disponibile lo spazio necessario per l'inserimento del nuovo materiale richiesto.

5.2 FIFO-protetto

Ogni volta che si verifica un page fault, l'algoritmo FIFO rimuove il materiale che è stato inserito per primo all'interno della stampante.

Per adattarlo al caso batch, come mostrato per LRU-protetto, si ha bisogno di una protezione dei materiali della richiesta già presenti nella stampante. L'algoritmo FIFO-protetto agisce dunque in due fasi:

- Nella prima fase verifica quali materiali richiesti siano già presenti all'interno della stampante e le protegge dalla rimozione.

- Nella seconda fase, l'algoritmo si comporta come FIFO. In particolare, a ogni page fault vengono rimossi i materiali, tra quelli non protetti, che sono stati inseriti per primi nella stampante.

5.3 GreedyDual-protetto

Come prima osservazione, si ricorda che essendo GreedyDual un algoritmo progettato per il problema del paging pesato, esso richiede l'associazione di un costo di caricamento a ciascun materiale che sia indipendente dal materiale rimosso. Nel caso di studio considerato, tuttavia, non sono disponibili tali costi in forma esplicita, bensì si ha una matrice dei costi di transizione tra i materiali. Per questo motivo, si compie un'approssimazione, assumendo come costo associato a ciascun materiale il valore medio dei costi di transizione che lo riguardano.

GreedyDual, nella versione batch, agisce nel seguente modo:

- Prima attua la protezione, ovvero controlla se alcuni materiali della richiesta sono già presenti nella cache, e li protegge evitando che vengano tolti dalla stampante e aggiorna il loro parametro $H = L + \omega_i$ dove ω_i è il costo del materiale i.

- Per ogni materiale che non è presente nella stampante, vengono tolti i materiali con H minore e aggiornato $H = L + \omega_i$.

Il parametro di labeling globale L viene aggiornato al termine dell'elaborazione dell'intera richiesta come:

$$L = \min\{H_1, H_2, H_3, \ldots\},$$

dove $H_1, H_2, H_3, \ldots$ rappresentano i valori dei materiali attualmente presenti all'interno della stampante. Questo garantisce che L evolva gradualmente nel tempo invece di aumentare dopo ogni inserimento.

5.4 Un modello di Programmazione Lineare Intera Mista

L'idea alla base del modello MILP consiste nel rappresentare il problema su un grafo $(T+1)$-partito, dove T rappresenta il numero di richieste di batch di materiale, ovvero corrisponde al numero di stampe 3D da eseguire. Ogni partizione del grafo consiste in K nodi, ognuno associato a uno dei materiali. Gli archi del grafo rappresentano, invece, le transizioni tra i materiali nel tempo (vale a dire tra il tempo t e $t+1$, con $t = 0, \ldots, T$) e ad essi è associato un costo che tiene conto del costo del materiale sprecato durante il cambio di materiale, che prevede la pulizia dei tubi di alimentazione.

L'evoluzione della configurazione della stampante viene quindi descritta mediante un insieme di cammini nel grafo, uno per ciascun slot disponibile. Ogni cammino rappresenta la sequenza di materiali assegnati a uno specifico slot durante l'intero processo, con l'obiettivo di minimizzare il costo complessivo del cambio dei materiali.

Esempio 7. *Nell'esempio consideriamo un problema con due slot, cinque materiali e cinque stampe come riportato nella Tabella 5.1.*

N° stampa	R_1	R_2	R_3	R_4	R_5
Materiali	3, 4	4, 5	3	3	1, 4

Tabella 5.1: Sequenza di richieste generata.

Nella Figura 5.1 è riportato il risultato dei percorsi relativi al caso con due slot e cinque materiali disponibili, supponendo che al tempo $t = 0$ siano presenti nella stampante le cartucce dei materiali 2 e 4. Si osserva come il percorso ottimo mantenga costantemente il materiale 4 all'interno della cache, sostituendo esclusivamente l'altro materiale (ad esempio, tra il tempo $t = 0$ e il tempo $t = 1$, vale a dire prima di avviare la prima stampa, il secondo materiale viene sostituito con il terzo, che successivamente verrà sostituito con il quinto, e così via). Questo comportamento evidenzia la stabilità della soluzione ottima, che identifica nel materiale 4 un elemento conveniente da mantenere permanentemente in cache.

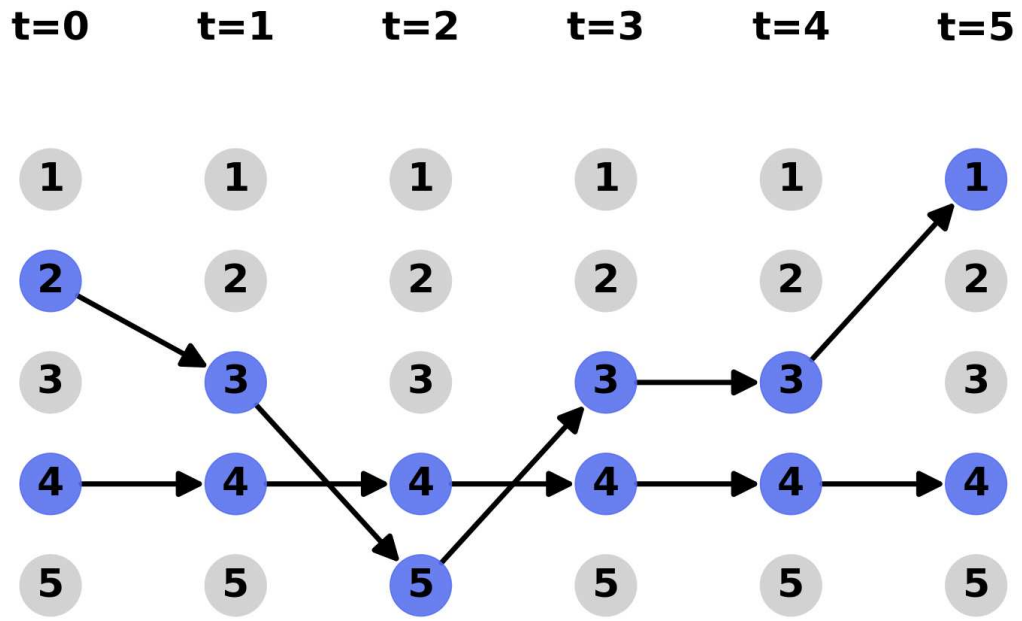

Figura 5.1: Esempio di soluzione del cambio dei materiali rappresentata tramite cammini su un grafo.

5.4.1 Parametri

I dati a disposizione sono i seguenti:

- il numero di slot della stampante K;
- il numero dei materiali M;
- il numero delle richieste T;
- un parametro binario $s_{k,i}$ che vale 1 se il materiale i occupa lo slot k al tempo $t = 0$, 0 altrimenti;
- data la sequenza di richieste $\sigma = \{\sigma_1, \ldots, \sigma_T\}$, dove ogni entrata viene indicata con $\sigma_t = \{m_1, ..., m_n\}$ con m_i che indicano materiali richiesti, allora introduciamo gli insiemi

$$M^t_{req} = \{m_1, ..., m_n\} \subset \{1, \ldots, M\} \quad \forall t = 1, \ldots, T;$$

- una matrice $C = (c_{i,j})_{i,j=1,\dots,M}$ la che rappresenta il costo di togliere il materiale i per inserire il materiale j e che quindi definisce i pesi degli archi del grafo.

5.4.2 Variabili decisionali

Per modellizzare il problema, introduciamo due variabili binarie che renderemo soggette a vincoli. La prima variabile

$$p_{k,i,t} = \begin{cases} 1 & \text{se al tempo } t \text{ il materiale } i \text{ occupa lo slot } k \\ 0 & \text{altrimenti} \end{cases}$$

rappresenta i vertici del grafo che sono visitati dai K cammini, indicando quindi se i vari materiali sono "attivi", ovvero installati nella stampante, o non attivi.

La seconda variabile

$$z_{k,i,j,t} = \begin{cases} 1 & \text{il materiale } i \text{ è sostituito da } j \text{ nello slot } k \text{ tra i tempi } t \text{ e } t{+}1 \\ 0 & \text{altrimenti} \end{cases}$$

indica invece quali archi archi del grafo sono selezionati dai K cammini.

5.4.3 Funzione obiettivo

L'obiettivo è minimizzare il costo totale di tutti i cambi dei materiali, vale a dire la somma dei pesi associati agli archi selezionati dai K cammini

$$\min \sum_{k=0}^{K} \sum_{i=0}^{M} \sum_{j=0}^{M} \sum_{t=0}^{T-1} c_{i,j} \cdot z_{k,i,j,t}.$$

5.4.4 Vincoli

Ora abbiamo bisogno di alcuni vincoli che stabiliscono quali soluzioni sono ammissibili per il problema che stiamo studiando.

- Il primo vincolo ci permette di fissare le condizioni iniziali, i nodi iniziali a tempo $t = 0$ devono coincidere con quelli di partenza, ovvero che

$$p_{k,i,0} = s_{k,i} \quad \forall k = 1, \ldots, K.$$

- Dobbiamo imporre che deve essere presente uno e un solo materiale alla volta in ciascuno slot della stampante; di fatto la richiesta diventa che possiamo avere solo un nodo attivo fissato lo slot e il tempo

$$\sum_{i=1}^{M} p_{k,i,t} = 1 \quad \forall k = 1, \ldots, K, \forall t = 0, ..., T.$$

- Per avere dei cammini, il numero degli archi in uscita e in entrata di ogni nodo deve coincidere. Inoltre, occorre rendere coerenti i valori delle due variabili decisionali, facendo in modo che se un nodo viene visitato ($p_{k,i,t} = 1$), allora deve avere un arco in entrata e uno in uscita selezionati, altrimenti nessuno degli archi entranti o uscenti può essere selezionato. Introduciamo un insieme di vincoli per gli archi in uscita

$$\sum_{j=1}^{M} z_{k,i,j,t} = p_{k,i,t} \quad \forall k = 1, \ldots, K, \forall i = 1, \ldots, M, \forall t = 0, \ldots, T-1,$$

  e un insieme di vincoli per gli archi in entrata

$$\sum_{j=1}^{M} z_{k,i,j,t} = p_{k,i,t+1} \quad \forall k = 1, \ldots, K, \forall i = 1, \ldots, M, \forall t = 1, \ldots, T.$$

- Serve una condizione che ci garantisca che i materiali richiesti ad ogni istante temporale vengano sempre visitati

$$\sum_{k=1}^{K} p_{k,i,t} \geq 1 \quad \forall i \in M_{req}^{t}, \forall t = 0, \ldots, T.$$

- Ogni materiale non può essere presente in due cammini allo stesso tem-

po, dato che significherebbe inserire due cartucce uguali in due slot distinti

$$\sum_{k=1}^{K} p_{k,i,t} \geq 1 \quad \forall i \in M, \forall t = 0, \ldots, T.$$

- Infine, specifichiamo i vincoli di dominio

$$p_{k,i,t} \in \{0,1\} \quad \forall k = 1, \ldots, K, i = 1, \ldots, M, t = 0, \ldots, T,$$

$$z_{k,i,j,t} \in \{0,1\} \quad \forall k = 1, \ldots, K, i, j = 1, \ldots, M, t = 0, \ldots, T-1.$$

Disponendo di tutta la sequenza di richieste a priori, il modello MILP determina l'ottimo offline, rispetto al quale valutare le prestazioni degli algoritmi online analizzati in questa tesi, ossia LRU-protetto, FIFO-protetto e GreedyDual-protetto.

Capitolo 6

Analisi sperimentale e risultati

Ai fini di introdurre l'analisi sperimentale, il primo step è la formalizzazione del problema in riferimento al caso di studio del Laboratorio 3D4Med. L'obiettivo è ridurre i costi di produzione delle stampanti ottimizzando il cambio delle cartucce.

Sono disponibili 14 materiali distinti, tra cui il materiale di supporto *FullCure 706*, che viene utilizzato per tutte le stampe e, pertanto, occuperà sempre uno degli slot della stampante. I restanti 13 materiali sono: *VeroMagenta, VeroGray, VeroCyan, VeroYellow, VeroClear, AgilusClear, ABS RGD 513, ABS RGD 515, TangoPlus 930, VeroPureWhite, TissueMatrix, GelMatrix, BoneMatrix*. A ciascun materiale è associato un costo di caricamento, il cui valore dipende dal materiale precedentemente presente nello stesso slot della stampante come riportato nella Figura 6.1.

I costi di caricamento non sono simmetrici, ma dipendono dalla coppia di materiali coinvolti. Ad esempio, considerando ABS RGD 515 e AgilusClear, la sostituzione del primo con il secondo comporta un costo pari a 236,25 €, mentre la sostituzione inversa ha un costo di 225,68 €.

Nel complesso, tuttavia, la matrice dei costi presenta una struttura piuttosto regolare. In particolare, alcuni materiali sono caratterizzati da costi di caricamento pressoché uniformi; ciò è evidente, ad esempio, osservando le prime quattro righe della matrice. Questa osservazione suggerisce che il problema possa essere modellato come una variante del problema del batch

paging pesato con protezione, rendendo potenzialmente efficaci gli algoritmi progettati per tale contesto.

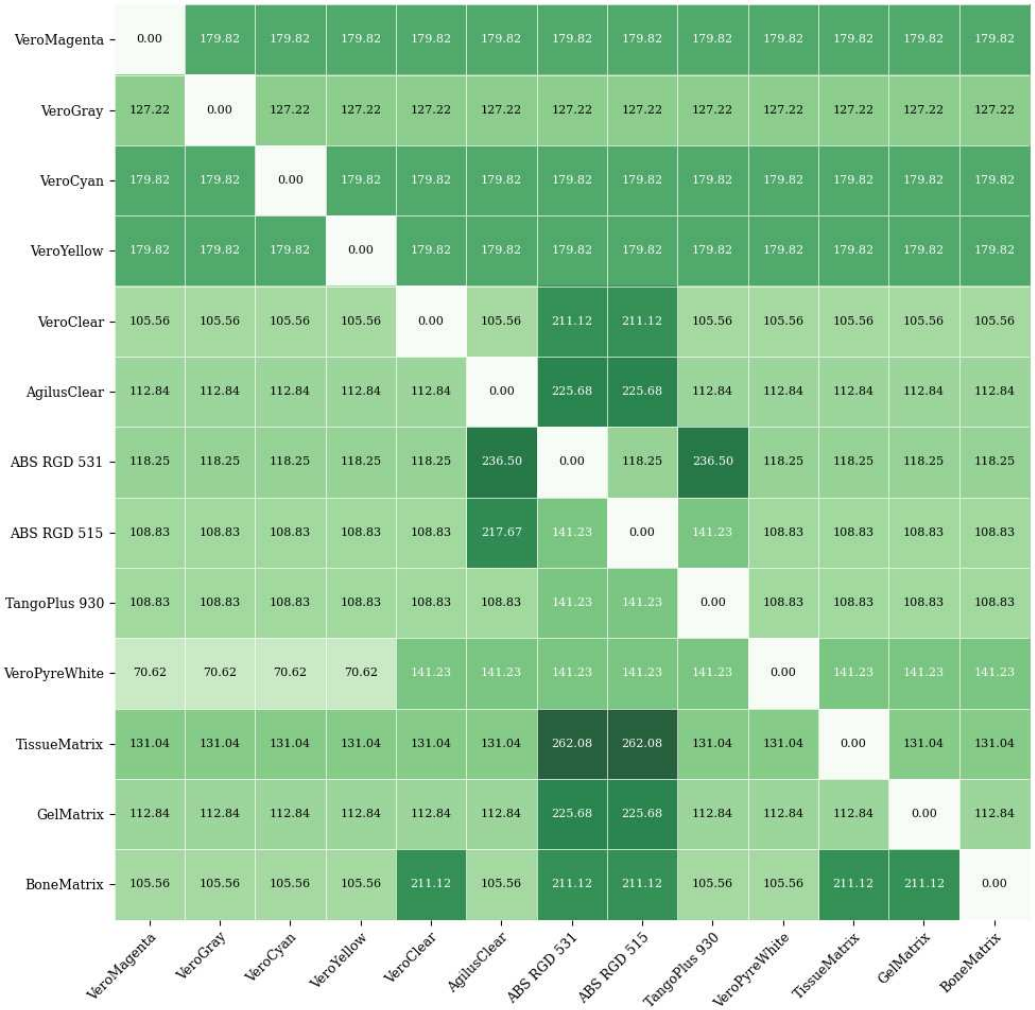

Figura 6.1: Matrice dei costi del cambio dei materiali, rappresentati in euro (€)

Nell'analisi sperimentale vengono considerate entrambe le stampanti disponibili in laboratorio. Per semplicità, nel modello proposto non viene considerato il costo di permanenza dei materiali all'interno della stampante, nonostante durante la fase di pulizia anch'essi subiscano un consumo parziale.

Le stampanti considerate sono:

- Stratasys Objet 260 Connex 3, con quattro slot per le cartucce e compatibile con i primi nove materiali (Figura 6.2).

Figura 6.2: Stratasys Objet 260 Connex 3

- Stratasys J750 DA, con sette slot per le cartucce e compatibile con tutti i materiali in tabella (Figura 6.3).

Figura 6.3: Stratasys J750 DA

Di conseguenza, poiché uno slot sarà perennemente occupato dal materiale di supporto, considereremo un numero di slot pari a $K = 3$ per la prima stampante e $K = 6$ per la seconda

In questo capitolo, analizziamo gli algoritmi LRU-protetto, FIFO-protetto e GreedyDual-protetto. Inoltre, come ulteriore termine di confronto, viene introdotto *Random-protetto*, un algoritmo baseline che, in caso di page fault, seleziona casualmente i materiali da rimuovere tra quelli non protetti.

Le prestazioni di tali algoritmi vengono infine confrontate con quelle di un modello di Programmazione Lineare Intera Mista (MILP), utilizzato per determinare una soluzione ottima del problema.

Si assume che la stampante sia inizialmente sempre piena e non si considera il costo di caricamento delle cartucce iniziali, poiché tale costo risulta comune a tutti i metodi utilizzati.

Nei casi che prenderemo in analisi, consideriamo la stampante Stratasys J750DA piena coi seguenti materiali: *VeroGray, TangoPlus930, VeroCyan, AgilusClear, ABS RGD 531, VeroPureWhite*; e la stampante Stratasys Object 260 Connex 3 con *VeroGray, TangoPlus930, VeroCyan*.

L'algoritmo GreedyDual-protetto richiede che a ciascun materiale venga associato un costo di caricamento. Come descritto nella Sezione 5.3, tale costo è assunto pari alla media dei costi di transizione, che per il caso di studio considerato sono riportati nella Tabella 6.1.

Materiali	**Costi (€)**
1: VeroMagenta	179.82
2: VeroGray	127.22
3: VeroCyan	179.82
4: VeroYellow	179.82
5: VeroClear	123.15
6: AgilusClear	131.65
7: ABS RGD 531	137.96
8: ABS RGD 515	123.30
9: TangoPlus 930	114.23
10: VeroPyreWhite	117.69
11: TissueMatrix	152.88
12: GelMatrix	131.65
13: BoneMatrix	149.54

Tabella 6.1: Elenco dei materiali e dei relativi costi.

Tutti gli algoritmi sono stati implementati in `Python 3.14` utilizzando `Gurobi 13.0.2` per la risoluzione delle istanze con il modello MILP.

Le richieste di input sono state generate in modo casuale, considerando sequenze con elementi complessivi di lunghezza crescente ($N = 10, 25, 50, 75, 100$).

Ogni richiesta R_i avrà dimensione casuale tra 1 e il minimo tra la dimensione della cache e N, e ogni elemento sarà composto da materiali casuali da 1 a 13. Successivamente N verrà ridotto di $|R_i|$ e generata una nuova sequenza fino a che $N = 0$.

Un esempio di istanza è riportato in Tabella 6.2, dove si può osservare una sequenza di $T = 15$ richieste che corrisponde a $N = 50$ materiali richiesti.

Richiesta	Batch
R_1	$[9, 5, 10, 11, 6, 3]$
R_2	$[6]$
R_3	$[1, 2, 12, 10, 6, 8]$
R_4	$[11, 10]$
R_5	$[13, 3, 6]$
R_6	$[3, 7, 9]$
R_7	$[6, 4]$
R_8	$[1]$
R_9	$[7, 10, 5]$
R_{10}	$[12, 7, 6, 9, 13]$
R_{11}	$[7, 4, 10, 3, 9, 13]$
R_{12}	$[8, 3, 7]$
R_{13}	$[12, 6, 4]$
R_{14}	$[8, 12, 7, 9]$
R_{15}	$[13, 8]$

Tabella 6.2: Esempio di sequenza delle richieste utilizzata nell'analisi sperimentale.

6.1 Confronto LRU e FIFO per il batch paging

Prima di procedere con l'analisi competitiva degli algoritmi in termini di costo, è utile valutarne sperimentalmente il comportamento nel caso di costo unitario, considerando esclusivamente il numero di *page fault*. A tal fine, i diversi algoritmi saranno confrontati con LFD, introdotto nella Sezione 3.1, assumendo come riferimento il caso con due stampanti e utilizzando sequenze di richieste di lunghezza crescente.

Come dimostrato nella Sezione 3.1, LFD costituisce l'algoritmo offline ottimo per il problema del batch paging. Pertanto, confrontando il numero di *page fault* prodotti dai diversi algoritmi con quelli ottenuti da LFD, è possibile ricavarne sperimentalmente il rapporto competitivo.

Per prima cosa analizziamo la stampante Stratasys Objet260 Connex3. In Figura 6.4 è riportato il numero di *page fault* prodotti dai diversi algoritmi al variare della lunghezza della sequenza di richieste. Si osserva che il rap-

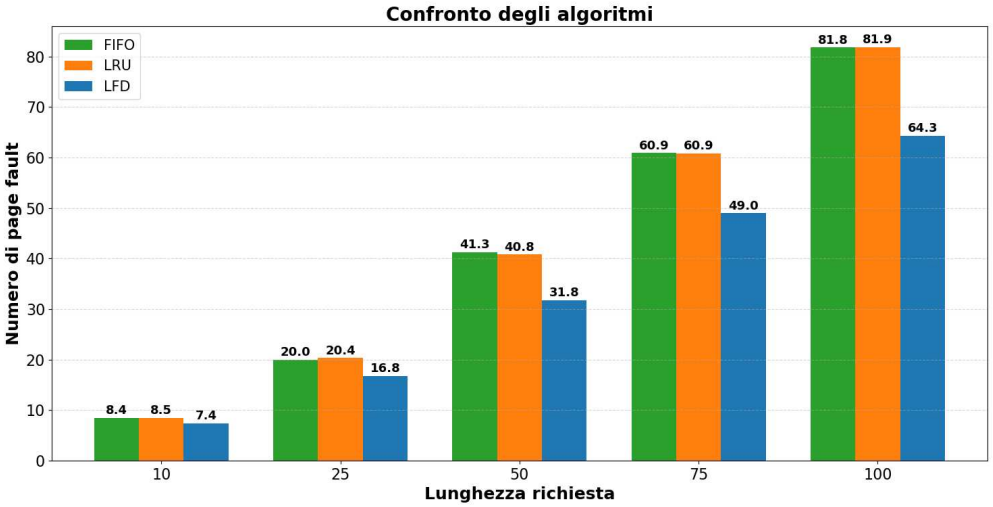

Figura 6.4: Confronto LFD batch e FIFO batch con LFD batch Stratasys Objet 260 Connex 3

porto competitivo sperimentale risulta significativamente inferiore al limite teorico. In particolare, il valore massimo osservato è pari a 1.5 (Tab 6.3), mentre il minimo è pari a 1.38, ottenuto per sequenze di richieste di lunghezza ridotta. Tali risultati evidenziano come, nelle istanze considerate, il comportamento dell'algoritmo sia sensibilmente migliore rispetto al rapporto teorico di competitività, pari a 3.

Lo stesso studio viene ora condotto per la stampante Stratasys J750 DA. Nella Figura 6.5 osserviamo il numero di *page fault* confrontati tra i vari algoritmi.

Competitività	N = 10	N = 25	N = 50	N = 75	N = 100
LRU	1.5	1.46	1.38	1.38	1.4
FIFO	1.5	1.38	1.39	1.35	1.38

Tabella 6.3: Competitività sperimentale di LRU e FIFO rispetto a LFD al variare della lunghezza della sequenza di richieste per Stratasys Objet 260 Connex 3, dove N rappresenta il numero complessivo di elementi contenuti nelle richieste.

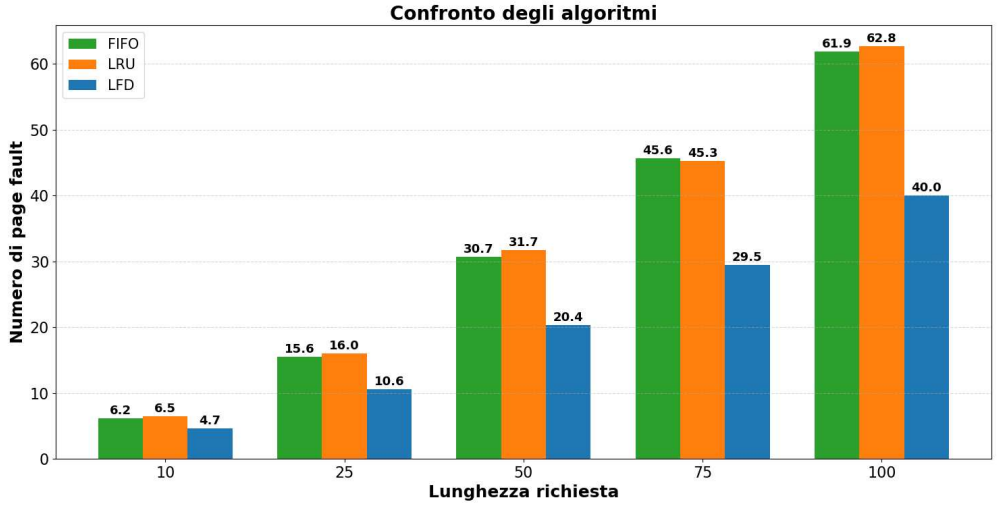

Figura 6.5: Confronto LFD batch e FIFO batch con LFD batch per Stratasys J750 DA

Competitività	N = 10	N = 25	N = 50	N = 75	N = 100
LRU	2.0	2.0	1.73	1.80	1.78
FIFO	2.0	1.86	1.73	1.88	1.78

Tabella 6.4: Competitive ratio sperimentale di LRU e FIFO rispetto a LFD al variare della lunghezza della sequenza di richieste Stratasys J750 DA, dove N rappresenta la lunghezza complessiva degli algoritmi.

Analogamente al caso della stampante precedente, anche in questo caso il rapporto di competitività sperimentale risulta notevolmente inferiore al limite teorico. In particolare, il valore massimo osservato è pari a 2 per LRU e FIFO (vedi Tabella 6.4), mentre il valore minimo è pari a 1.73 per

entrambi gli algoritmi. Tali risultati sono ancora più significativi rispetto al caso precedente, poiché il rapporto teorico di competitività per questa stampante è pari a $k = 6$.

6.2 Confronto dei costi

Per confrontare le prestazioni degli algoritmi in termini di costo, una volta determinati i percorsi associati ai singoli materiali, è stata utilizzata la matrice dei costi per calcolare il costo complessivo delle operazioni di caricamento e mantenimento effettuate durante l'esecuzione. In questo modo è stato possibile stimare il costo effettivo di ciascun algoritmo e confrontarlo con la soluzione ottima ottenuta tramite il modello MILP.

Sono state generate venti sequenze per ogni algoritmo e per ogni dimensione dell'istanza, confrontando la media dei costi e il gap tra i vari algoritmi rispetto alla soluzione ottima prodotta tramite MILP.

La prima stampante che consideriamo è Stratasys Objet 260 Connex 3, con cache di dimensione tre e solo nove materiali compatibili. I risultati sono riportati in Tabella 6.5.

Algoritmo	$N = 10$	$N = 25$	$N = 50$	$N = 75$	$N = 100$
LRU	1014.12	1869.94	4932.61	7640.24	9863.74
FIFO	1011.24	1869.51	4945.04	7581.73	9888.86
Random	1017.73	1894.53	5020.81	7604.81	10062.23
GreedyDual	998.91	1834.07	4898.22	7489.28	9832.73
MILP	855.60	1523.46	4051.77	6107.32	8093.24

Tabella 6.5: Media dei costi totali (in euro) su 20 istanze al variare del numero di materiali richiesti per Stratasys Object 260 Connex 3.

Come prima osservazione, si nota che il costo complessivo cresce all'aumentare del numero di richieste. Inoltre, sebbene la differenza tra la soluzione ottima e quelle prodotte dagli algoritmi online sia contenuta per sequenze di dimensioni ridotte, tale divario tende ad aumentare al crescere della lunghezza della sequenza.

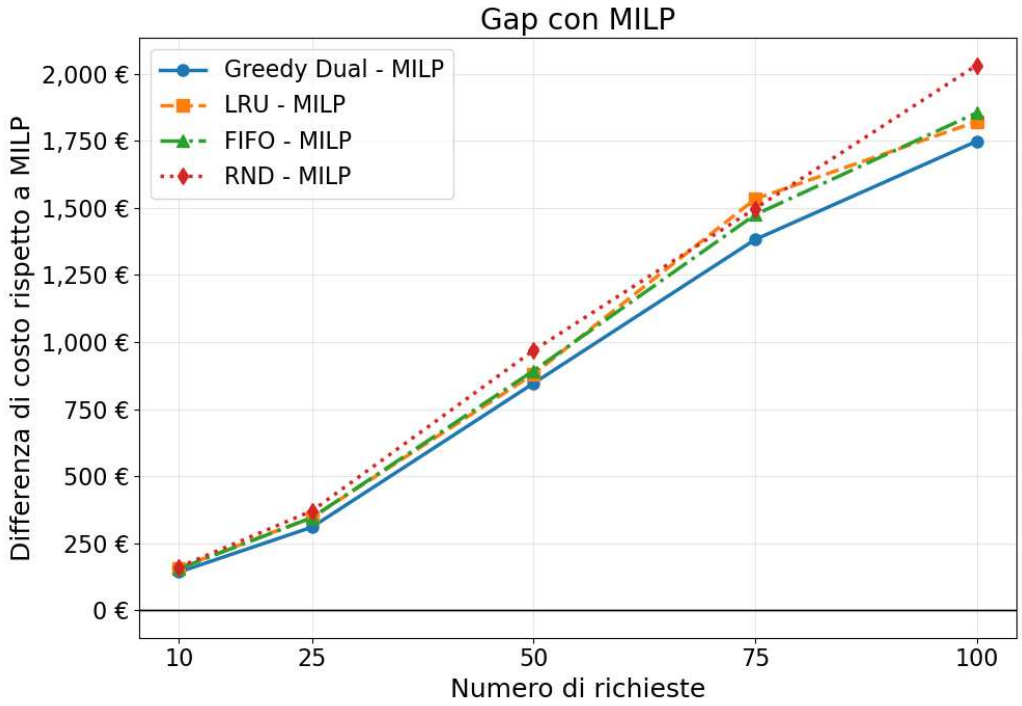

Figura 6.6: Gap con MILP per Stratasys Objet 260 Connex 3

Questo comportamento è coerente con quanto atteso. Il modello MILP, avendo a disposizione tutta la sequenza di richieste, è infatti in grado di pianificare le sostituzioni in modo ottimale, mentre gli algoritmi online prendono decisioni basandosi esclusivamente sulle informazioni disponibili fino al tempo corrente. All'aumentare del numero di richieste, il vantaggio derivante dalla conoscenza preventiva dell'intera sequenza diventa sempre più significativo, consentendo al modello ottimo di ridurre ulteriormente il costo complessivo rispetto alle strategie online.

In secondo luogo, il numero limitato di pagine unito alla memoria della cache ridotta porta gli algoritmi ad avere un numero minore di scelte possibili e quindi a prendere decisioni simili. La protezione, di fatto, limita la libertà di sostituzione dei materiali, costringendo l'algoritmo in alcuni casi ad effettuare scelte forzate.

Un altro dato utile a comprendere gli algoritmi è il gap di distacco con l'ottimo offline come mostrato in Figura 6.6.

Nonostante il gap aumenti con la crescita del numero di richieste, GreedyDual-protetto riesce comunque a mantenere un rapporto migliore rispetto alle versioni protette di LRU, FIFO e Random. Il risparmio su sequenze di 100 elementi rispetto a una strategia di rimozione casuale è di 229.5 € medi a sequenza, mentre il risparmio complessivo sulle venti sequenze simulate è pari a 4590.10 €.

Come si osserva dai risultati in Tabella 6.6, per la seconda stampante il grado di complessità aumenta: infatti, Stratasys J750 DA ha a disposizione sei slot per le cartucce e può scegliere tra tutti e tredici i materiali.

Algoritmo	$N = 10$	$N = 25$	$N = 50$	$N = 75$	$N = 100$
LRU	762.03	1809.44	3915.28	5734.78	7686.64
FIFO	762.92	1829.80	3934.85	5781.79	7638.67
Random	749.92	1851.82	3899.20	5741.32	7659.04
GreedyDual	685.18	1737.69	3822.95	5625.43	7483.71
MILP	571.98	1263.14	2784.98	4166.95	5553.24

Tabella 6.6: Media dei costi totali (in euro) su 20 istanze al variare del numero di materiali richiesti per Stratasys J750 DA.

A differenza del caso precedente, dove su piccole sequenze la differenza con MILP era molto ridotta, ora, avendo a disposizione più cartucce da muovere MILP agisce meglio.

Anche in questo caso abbiamo che GreedyDual-protetto performa meglio rispetto agli altri metodi, generando un risparmio complessivo rispetto a Random-protetto pari a 3506.48 €.

Le Figure 6.10, 6.11, 6.12, 6.13, 6.14, mostrano le transizioni dei materiali nei diversi slot della stampante Stratasys J750DA in corrispondenza della sequenza di richieste riportata nella Tabella 6.2. Si osserva come la soluzione ottenuta dal modello MILP (Figura 6.14) presenti un'evoluzione più stabile rispetto a quelle prodotte dagli altri algoritmi. Infatti, grazie alla conoscenza dell'intera sequenza di richieste, il modello è in grado di pianificare le sostituzioni in modo più efficiente, ottenendo percorsi dei singoli slot meno soggetti a variazioni.

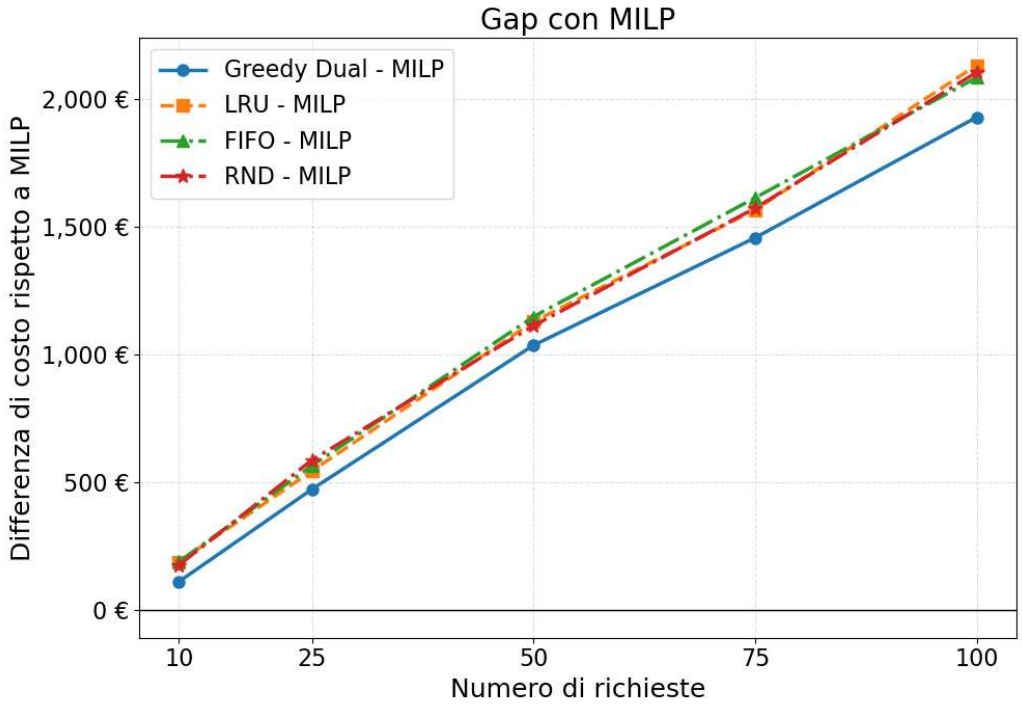

Figura 6.7: Gap con MILP per Stratasys J750 DA

6.3 Confronto del numero di page fault

Prima di confrontare le prestazioni rispetto alla soluzione ottima, si analizzano gli algoritmi LRU-protetto, FIFO-protetto e GreedyDual-protetto nel contesto del problema del batch paging. Il confronto viene effettuato in termini di numero di page fault.

In Figura 6.8 è riportato il numero medio di page fault calcolato sulle venti sequenze considerate per la stampante Stratasys Objet 260 Connex 3 con tre slot per i materiali. Dal grafico emerge come il meccanismo di protezione rappresenti il principale limite degli algoritmi analizzati: l'obbligo di sostituire esclusivamente pagine non protette porta frequentemente a rimuovere gli stessi elementi dalla cache, determinando di conseguenza un numero di page fault molto simile tra i diversi algoritmi. GreedyDual-protetto registra mediamente un numero di page fault leggermente inferiore rispetto a LRU-protetto e FIFO-protetto, ma il vantaggio risulta contenuto.

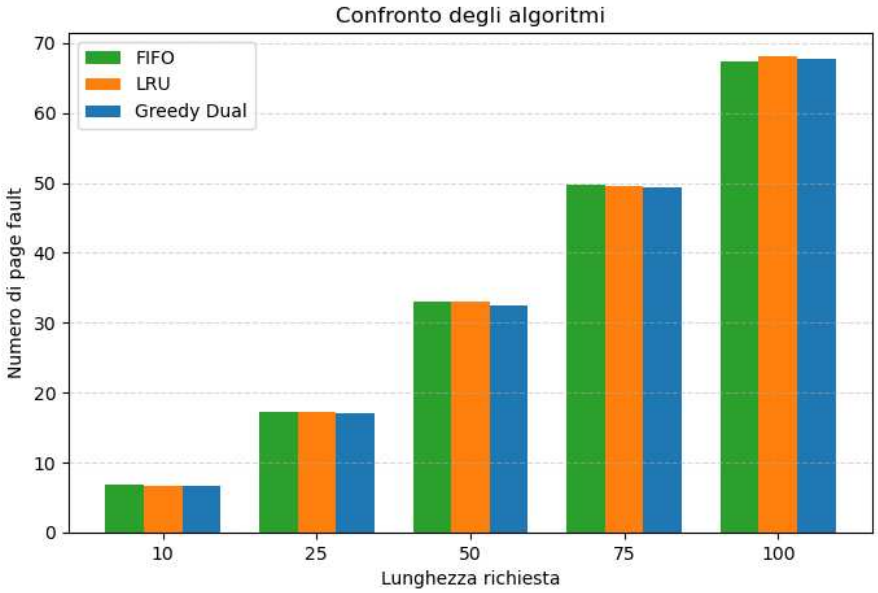

Figura 6.8: Confronto numero di page fault per Stratasys Objet 260 Connex 3

La situazione cambia nel caso della stampante Stratasys J750 DA(Fig 6.9) con sei slot per i materiali. La maggiore libertà di sostituzione delle pagine consente di evidenziare differenze nel comportamento dei tre algoritmi. In particolare, GreedyDual-protetto registra mediamente un numero di page fault inferiore rispetto a FIFO-protetto e LRU-protetto, mostrando un vantaggio, seppur contenuto, rispetto agli altri.

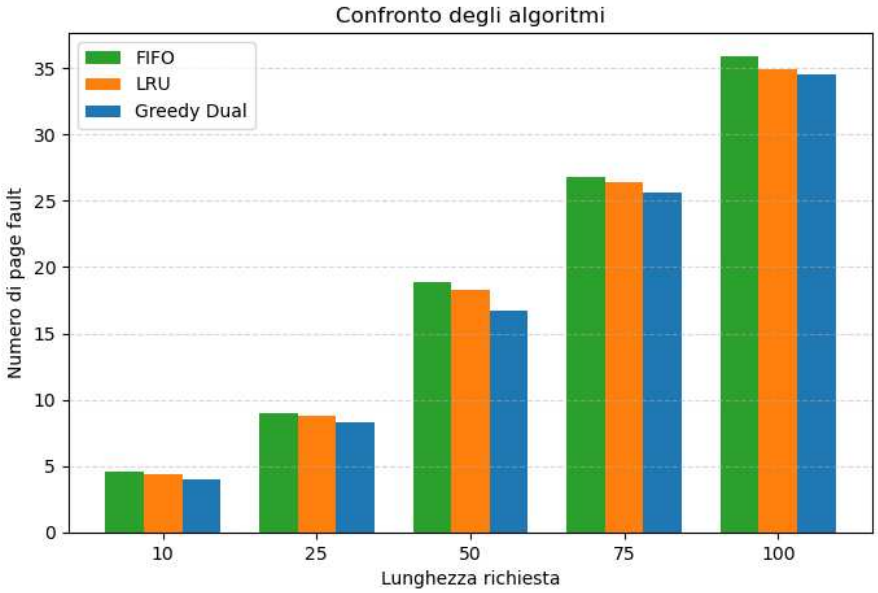

Figura 6.9: Confronto numero di page fault per Stratasys J750 DA

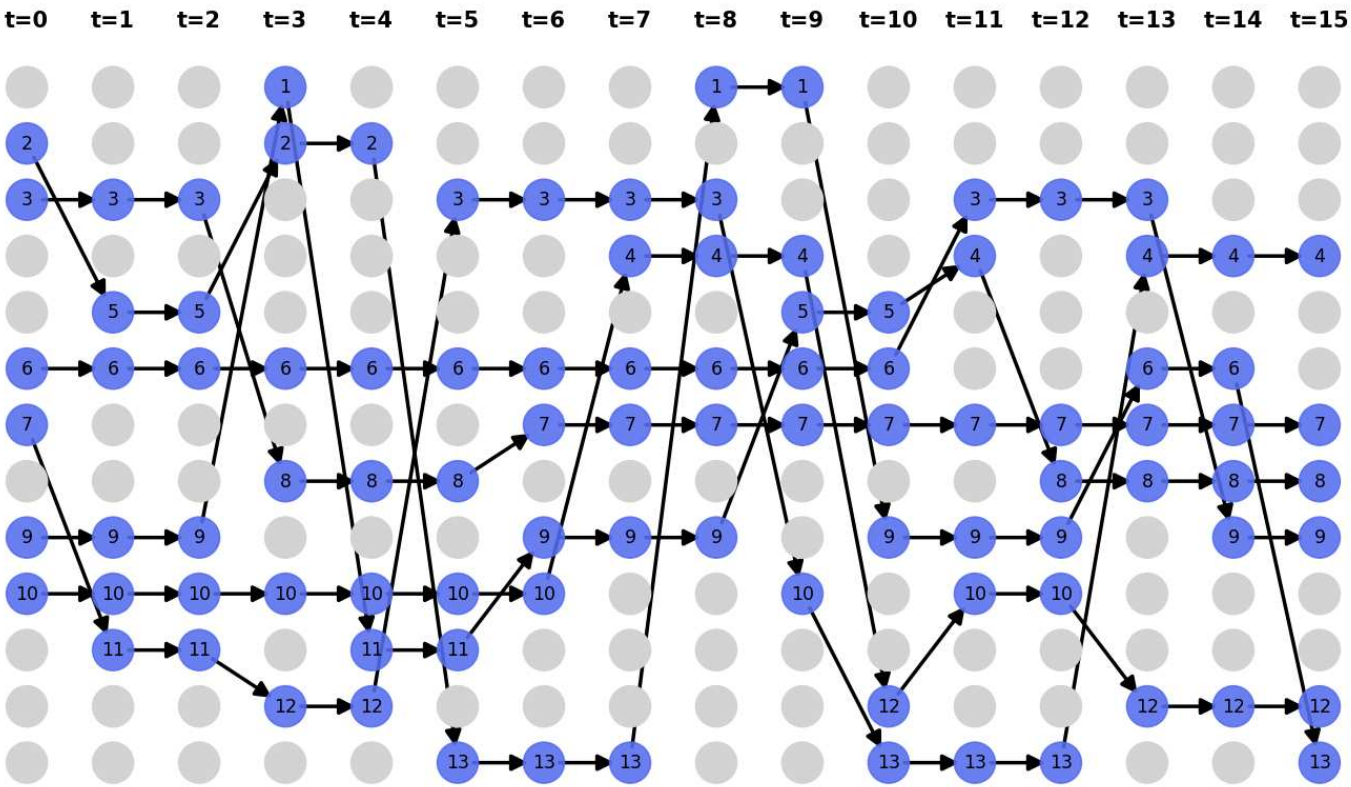

Figura 6.10: Esempio paths LRU-protetto per Stratasys J750DA per la richiesta nella tabella 6.2

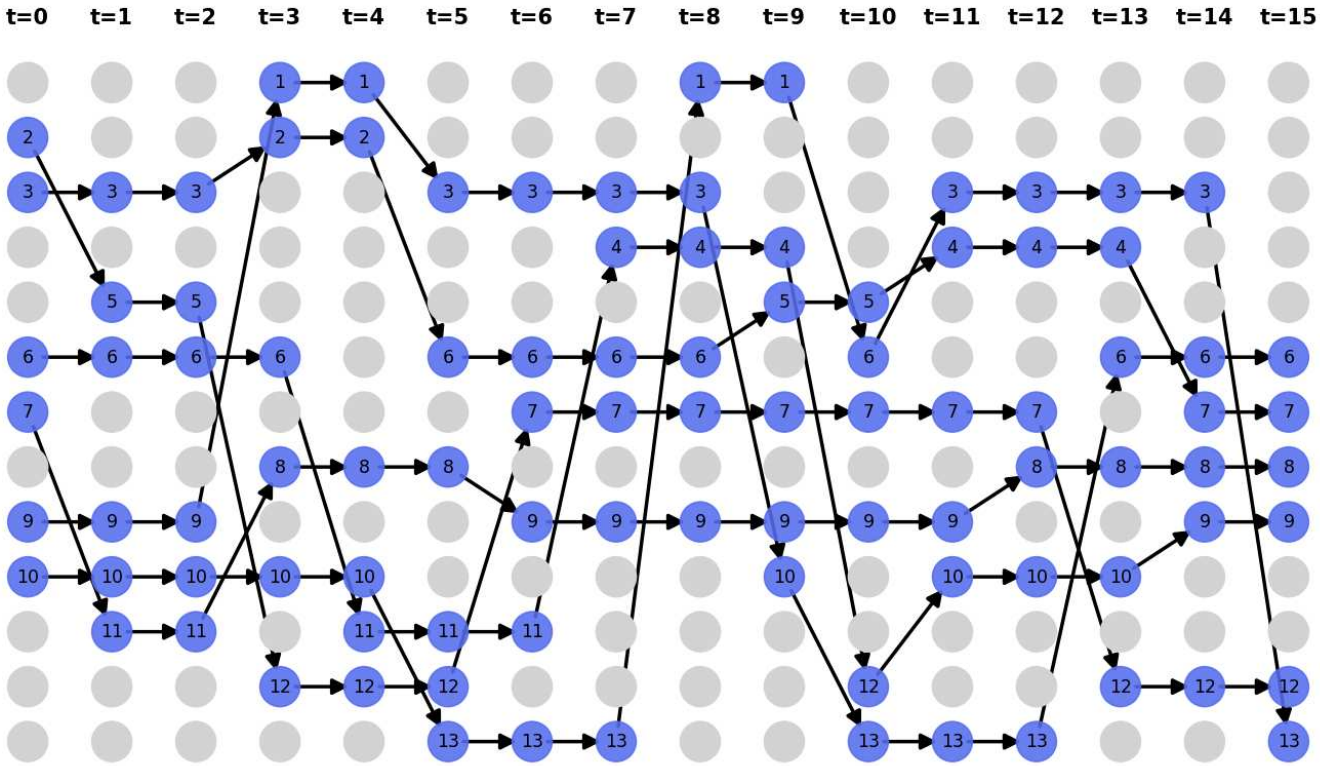

Figura 6.11: Esempio paths FIFO-protetto per Stratasys J750DA per la richiesta nella tabella 6.2

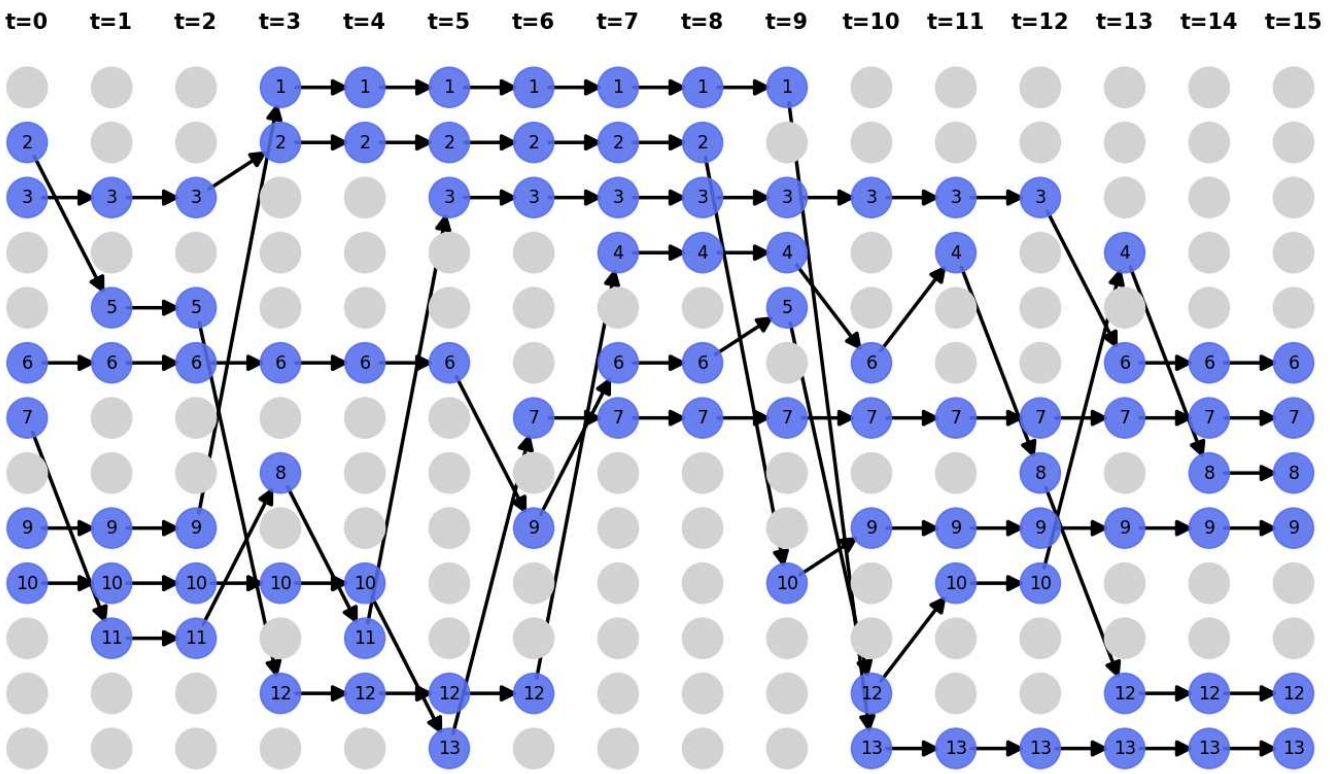

Figura 6.12: Esempio paths Random-protetto per Stratasys J750DA per la richiesta nella tabella 6.2

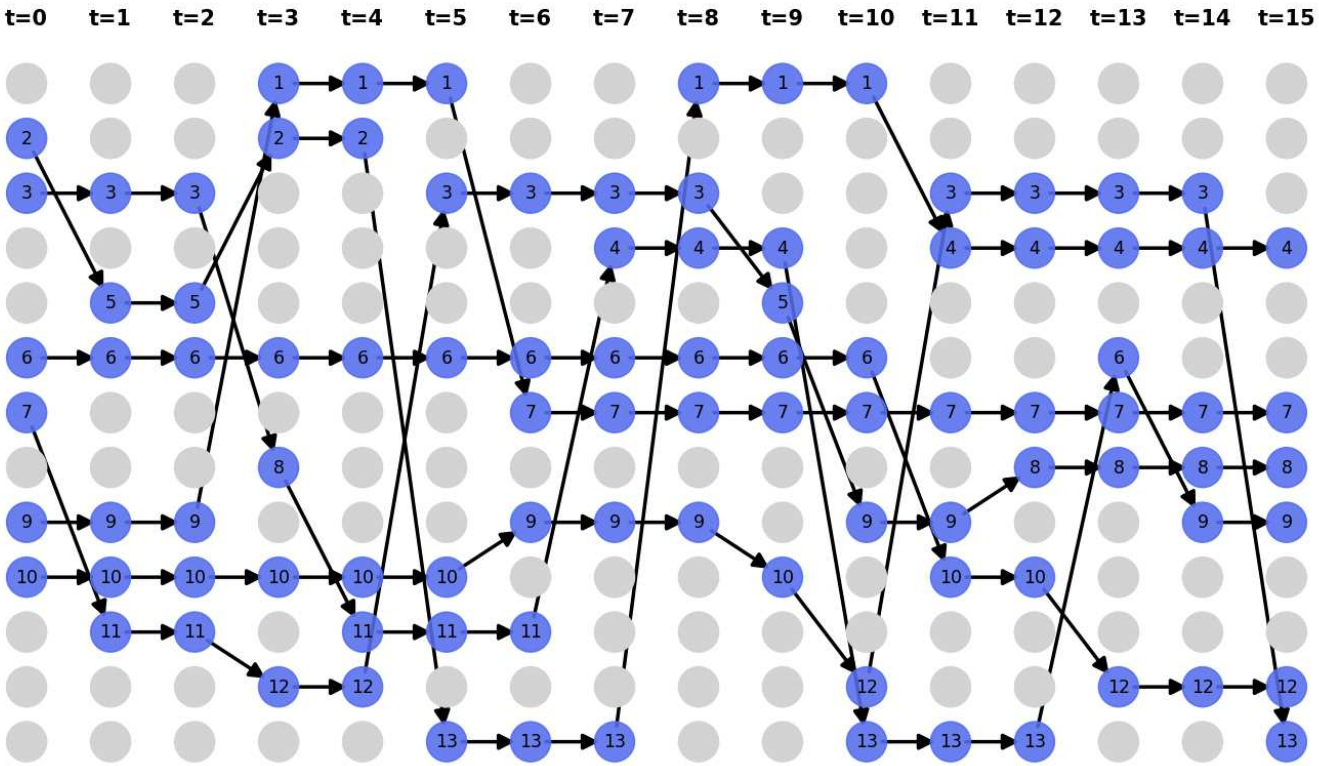

Figura 6.13: Esempio paths GreedyDual per Stratasys J750DA per la richiesta nella tabella 6.2

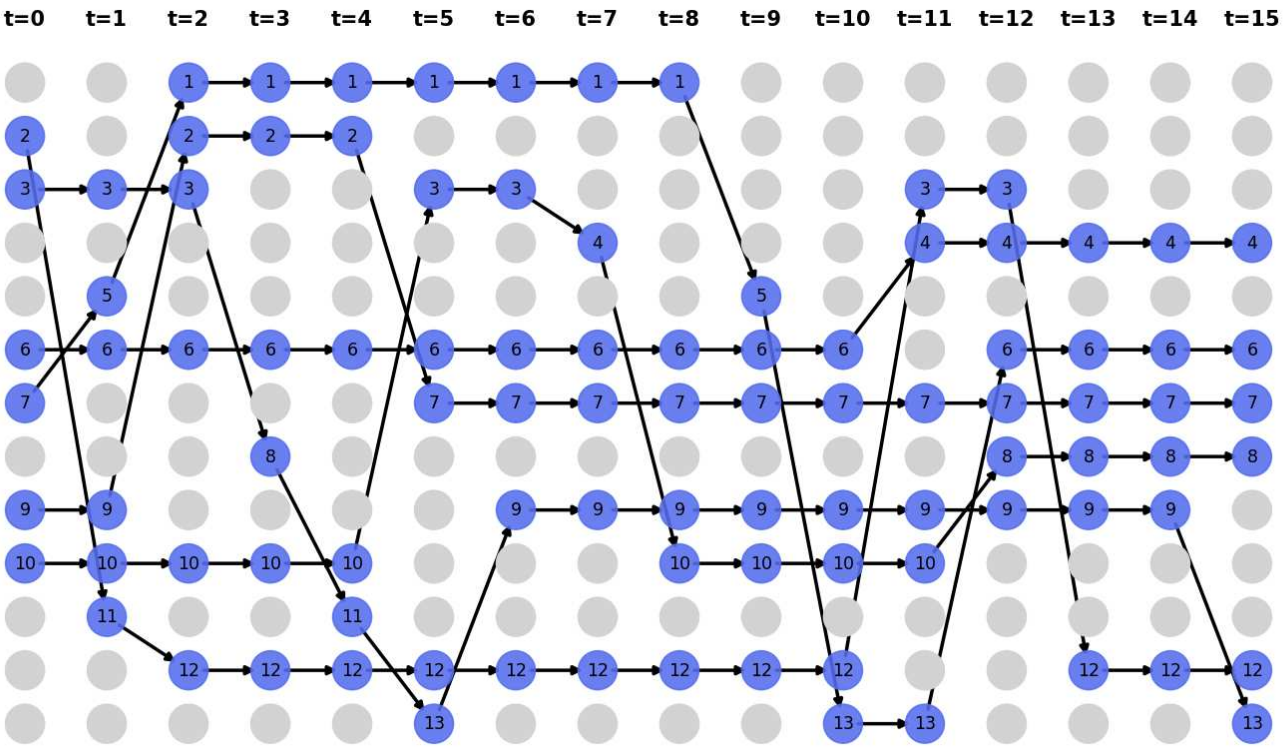

Figura 6.14: Esempio paths MILP per Stratasys J750DA per la richiesta nella tabella 6.2

Capitolo 7

Conclusioni

In questa tesi è stato affrontato il problema dell'ottimizzazione delle stampanti 3D per applicazioni in ambito medico. Il problema è stato modellato come un problema online, assumendo il paging come modello di riferimento. Sono stati analizzati i principali algoritmi per il paging dal punto di vista della competitività e successivamente estesi a un modello batch, preservandone le principali proprietà teoriche. L'analisi è stata quindi estesa al caso del paging pesato, nel quale il costo associato a ciascuna pagina non è più unitario. In tale contesto sono stati studiati i principali algoritmi per il paging pesato, successivamente adattati al modello batch protetto e valutati sperimentalmente mediante un confronto con una soluzione ottima ottenuta tramite un modello MILP sul caso di studio 3D4Med.

Il principale contributo di questo lavoro consiste nell'adattamento degli algoritmi di paging nella versione protetta al contesto applicativo di 3D4Med. Un ulteriore contributo è rappresentato dalla proposta di una possibile generalizzazione del problema del paging classico alla sua variante batch protetta e uno studio di competitività, con la dimostrazione della competitività delle versioni batch di LRU, FIFO e LFD.

La valutazione sperimentale mostra che GreedyDual-protetto, grazie all'utilizzo di un costo associato ai materiali oltre che dell'ordine temporale delle richieste, ottiene prestazioni migliori rispetto a LRU-protetto, FIFO-protetto e Random-protetto, avvicinandosi maggiormente alla soluzione ot-

tima. Al contrario, LRU-protetto e FIFO-protetto non considerano il costo delle transizioni tra i materiali, ma effettuano le sostituzioni esclusivamente sulla base dell'ordine di inserimento o dell'ultimo utilizzo delle pagine. Di conseguenza, le loro prestazioni dipendono fortemente dalla sequenza di richieste considerata e, in molti casi, risultano paragonabili a quelle ottenute da Random-protetto. Tale comportamento è dovuto alle limitazioni introdotte dal meccanismo di protezione delle pagine, che riduce la libertà di scelta dell'algoritmo durante le sostituzioni.

Anche GreedyDual-protetto presenta tuttavia alcune limitazioni. Sebbene introduca una politica di sostituzione basata su un costo associato ai materiali, tale costo è assunto costante e indipendente dalla specifica transizione tra due materiali. Di conseguenza, l'algoritmo non sfrutta completamente le informazioni contenute nella matrice dei costi di transizione, limitando la qualità della soluzione ottenuta. Inoltre, il meccanismo di protezione delle pagine richieste nel batch restringe ulteriormente l'insieme dei candidati alla sostituzione, riducendo la flessibilità della cache.

Una delle principali criticità comuni agli algoritmi considerati risiede inoltre nell'impossibilità di sfruttare pienamente la non simmetria della matrice dei costi di transizione. Un possibile sviluppo futuro consiste quindi nella progettazione di politiche di sostituzione che tengano esplicitamente conto del costo della specifica transizione tra il materiale rimosso e quello inserito nella cache. Sebbene tale approccio comporti un aumento della complessità computazionale, esso potrebbe consentire di ridurre ulteriormente il divario rispetto alla soluzione ottima ottenuta mediante il modello MILP.

Un'ulteriore direzione di ricerca riguarda l'introduzione di criteri di ottimizzazione aggiuntivi rispetto al solo costo economico. In particolare, la riduzione dei tempi di stampa potrebbe costituire un obiettivo di interesse, contribuendo ad aumentare l'efficienza complessiva del laboratorio e a rendere il modello maggiormente aderente alle esigenze operative del caso di studio.

Appendice A

Dimostrazione del Teorema 4.0.1

Dimostrazione. La dimostrazione si basa sulla versione del (h, k)-server problem, come nel caso del paging si riduce la dimensione della cache per l'ottimo in modo da limitarne le performance. Quindi la tesi diventa: $\frac{k}{k-h+1}$ è un lower bound per ogni (h, k)-server problem. Come ipotesi iniziale supponiamo che ci sia un algoritmo, ALG, che occupi i primi k punti e una richiesta in modo che ogni richiesta successiva produca un page fault, ovvero che richieda un punto fuori dai primi k valori. Sia $\sigma = r_1, ..., r_n$ tale richiesta, allora $ALG(\sigma) = \sum_{i=1}^{n-1} d(r_{i+1}, r_i)$. L'obiettivo ora della dimostrazione è mostrare che esiste un ottimo che non produce più di $\frac{1}{k-h+1} \sum_{i=1}^{n-1} d(r_{i+1}, r_i)$ page fault. Supponiamo, senza perdere di generalità, che r_1 non appartenga ai server iniziali di ALG. Per ogni sotto-insieme $S \subset M$ di cardinalità h con $r_1 \in S$, consideriamo l'algoritmo B_S in modo che risolva la richiesta r_i a partire da S spostando il server nella posizione precedente, ovvero r_{i-1}. Sia $\mathcal{B}$ l'unione di questi algoritmi, provo che il numero di questi algoritmi è costante, ovvero che configurazioni diverse portano a diversi algoritmi.

Chiamiamo S^i set di configurazioni dopo la richiesta r_i, dimostro che se $S_1 \neq S_2$, cioè ho due diverse configurazioni iniziali, allora B_{S_1} e B_{S_2} sono fatti in modo che $S_1^i \neq S_2^i$ per ogni $i = 0, \ldots, n$. Partiamo da i=0, $S_1^0 = S_1 \neq S_2 = S_2^0$, e induciamo per casi:

- **Caso 1:** $r_i \in S_1^{i-1}$ e $r_i \in S_2^{i-1}$, di conseguenza non muoviamo i server, $S_j^i = S_j^{i-1}, j = 1, 2$, e quindi per ipotesi di induzione $S_1^i = S_1^{i-1} \neq S_2^{i-1} = S_2^i$.

- **Caso 2:** $r_i \in S_1^{i-1}$ e $r_i \notin S_2^{i-1}$, di conseguenza viene mosso solo S_2, e spostato r_{i-1} per far spazio a r_i. Quindi $S_1^i \neq S_2^i$.

- **Caso 3:** $r_i \notin S_1^{i-1}$ e $r_i \notin S_2^{i-1}$, quindi $S_1^{i-1} - r_{i-1} \neq S_2^{i-1} - r_{i-1}$.

Da cui possiamo chiudere il claim.

Abbiamo dunque che tutti gli elementi di $\mathcal{B}$ sono distinti, e quindi abbiamo $\binom{k}{h-1}$ algoritmi. Mentre abbiamo $\binom{k-1}{h-1}$ server che si muovono per risolvere il problema, tutti tranne r_1. Quindi usando la simmetria otteniamo

$$\binom{k-1}{h-1} \sum_{i=2}^{n} d(r_{i-1}, r_i) = \binom{k-1}{h-1} \sum_{i=1}^{n-1} d(r_i, r_{i+1}).$$

Di conseguenza:

$$\frac{\binom{k-1}{h-1}}{\binom{k}{h-1}} = \frac{(k-1)!}{(h-1)!(k-h)!} \cdot \frac{(h-1)!(k-h+1)!}{k!}$$

$$= \frac{(k-1)!}{k!} \cdot \frac{(k-h+1)(k-h)!}{(k-h)!} = \frac{k-h+1}{k}.$$

Da cui il costo totale

$$\frac{k-h+1}{k} \sum_{i=1}^{n-1} d(r_i, r_{i+1}).$$

□

Appendice B

Scripts

B.1 Script LRU-protetto

```
import pandas as pd
import matplotlib.pyplot as plt
import networkx as nx
import random
from grafo_network import CacheGraph

df = pd.read_excel(
  "cost_matrix_new.xlsx", header=None,
)
C = df.to_numpy()
print(C.shape)

init = [2, 9, 3, 6, 7, 10]
req = {}
max_elementi = 50
totale = 0
i = 1
dim_cache = 6

# genera le request in modo casuale
while totale < max_elementi:
    num = random.randint(1, min(dim_cache, max_elementi -
   totale))
```

```
    req[i] = random.sample(range(1, 14), num)

    totale += num
    i += 1

def lru_batch(req, k_cache, frames_init, recent_init=None):
    frames = frames_init.copy()
    recent_use = recent_init.copy() if recent_init else
   frames.copy()

    page_fault = 0
    evicted = []

    protected = set(req).intersection(frames)

    for page in req:
        if page in frames:
            evicted.append((page, page))

            if page in recent_use:
                recent_use.remove(page)
            recent_use.append(page)

            continue

        page_fault += 1

        if len(frames) < k_cache:
            frames.append(page)
            recent_use.append(page)
            evicted.append((-1, page))
            continue

        candidates = [p for p in recent_use if p not in
   protected]

        if not candidates:
            victim = recent_use[0]
        else:
```

```
            victim = candidates[0]

        recent_use.remove(victim)

        idx = frames.index(victim)
        frames[idx] = page

        recent_use.append(page)
        evicted.append((victim, page))

    return page_fault, evicted, frames, recent_use

paths = {0: init.copy()}
recent_use = None
tot_fault = 0

for j in sorted(req.keys()):
    fault, ev, frames, recent_use = lru_batch(
        req[j],
        dim_cache,
        paths[j - 1],
        recent_use
    )

    paths[j] = frames.copy()
    tot_fault += fault

    print("Page fault:", fault)
    print("Evicted pages:", ev)
    print("frames", paths)

graph = CacheGraph()
graph.draw(paths)

print(req)
```

B.2 Script FIFO-protetto

```
import pandas as pd
import matplotlib.pyplot as plt
import networkx as nx
import random
from grafo_network import CacheGraph

df = pd.read_excel("cost_matrix_new.xlsx", header=None)
C = df.to_numpy()
print(C.shape)

# genera le request in modo casuale
while totale < max_elementi:
    num = random.randint(1, min(dim_cache, max_elementi -
   totale))
    req[i] = random.sample(range(1, 14), num)

    totale += num
    i += 1

paths = {0: [2, 9, 3, 6, 7, 10]}  # inizializzazione
age = None
time = None
tot_fault = 0

def fifo_batch(req, k_cache, frames_init, age_init=None,
   time_init=None):
    frames = frames_init.copy()
    evicted = []
    page_fault = 0

    if age_init is None:
        age = {p: i for i, p in enumerate(frames)}
        time = len(frames)
    else:
        age = age_init.copy()
        time = time_init

```

```
    protected = set(req).intersection(frames)

    for page in req:
        if page in frames:
            evicted.append((page, page))
            continue

        page_fault += 1

        if len(frames) < k_cache:
            frames.append(page)
            age[page] = time
            time += 1
            evicted.append((-1, page))
            continue

        candidates = [p for p in frames if p not in
    protected]

        if not candidates:
            candidates = frames.copy()

        victim = min(candidates, key=lambda p: age[p])

        idx = frames.index(victim)
        frames[idx] = page

        del age[victim]
        age[page] = time
        time += 1

        evicted.append((victim, page))

    return page_fault, evicted, frames, age, time

for t in sorted(req.keys()):
    fault, ev, frames, age, time = fifo_batch(
        req[t],
        4,
```

```
        paths[t - 1],
        age,
        time
    )

    paths[t] = frames.copy()
    tot_fault += fault

print("paths:", paths)

graph = CacheGraph()
graph.draw(paths)

tot_cost = 0
for j in range(len(paths[0])):
    for i in range(len(paths) - 1):
        tot_cost += C[paths[i][j] - 1][paths[i + 1][j] - 1]

print(tot_cost)
```

B.3 Script GreedyDual-protetto

```
import matplotlib.pyplot as plt
import networkx as nx
import pandas as pd
import numpy as np
from grafo_network import CacheGraph
import random

V = [2, 9, 3, 6, 7, 10]

# Carico la matrice dei costi
df = pd.read_excel(r"cost_matrix.xlsx",
    header=None)
C = df.to_numpy()
print(C.shape)

max_elementi = 50
```

```
totale = 0
i = 1
dim_cache = 6

while totale < max_elementi:
    num = random.randint(1, min(dim_cache, max_elementi -
   totale))
    req[i] = random.sample(range(1, 14), num)

    totale += num
    i += 1

T = list(range(len(req) + 1))

Cost = [
    179.82, 127.22, 179.82, 179.82, 123.15, 131.65, 137.96,
    123.30, 114.23, 117.69, 152.88, 131.65, 149.54
]

H = [
    Cost[V[0] - 1],
    Cost[V[1] - 1],
    Cost[V[2] - 1],
    Cost[V[3] - 1],
    Cost[V[4] - 1],
    Cost[V[5] - 1]
]

L = min(H)

Remove_vect = []
cache_vect = []
paths = {}

paths[0] = V.copy()

for i in range(1, len(req) + 1):

    protected = set()
```

```
55
56      for j in range(len(V)):
57          if V[j] in req[i]:
58              H[j] = L + Cost[V[j] - 1]
59              protected.add(V[j])
60
61
62      for page in req[i]:
63          if page not in V:
64
65              candidates = [
66                  j for j in range(len(V))
67                  if V[j] not in protected
68              ]
69
70              idx = min(candidates, key=lambda j: H[j])
71
72              Remove_vect.append(V[idx])
73
74              V[idx] = page
75              H[idx] = L + Cost[page - 1]
76
77              protected.add(page)
78
79      # STEP 3: aggiorna L una sola volta a fine richiesta
80      L = min(H)
81
82      cache_vect.append(V.copy())
83      paths[i] = V.copy()
84
85  print(cache_vect)
86  print(paths)
87
88  graph = CacheGraph()
89  graph.draw(paths)
90
91  tot_cost = 0
92
93  for j in range(len(paths[0])):
```

```
    for i in range(len(paths) - 1):
        tot_cost += C[paths[i][j] - 1][paths[i + 1][j] - 1]

print(tot_cost)
```

B.4 Random-protetto

```
import random
import networkx as nx
from grafo_network import CacheGraph
import pandas as pd


def random_batch(req_batch, k_cache, frames_init):
    frames = frames_init.copy()

    if len(req_batch) > k_cache:
        raise ValueError("Batch troppo grande per la cache")

    requested = set(req_batch)

    missing = [p for p in req_batch if p not in frames]
    candidates = [p for p in frames if p not in requested]

    page_faults = len(missing)
    evicted = []

    for page in missing:
        victim = random.choice(candidates)
        candidates.remove(victim)

        idx = frames.index(victim)
        frames[idx] = page

        evicted.append((victim, page))

    return page_faults, evicted, frames
```

B.5 Script MILP tramite Gurobi

```
import gurobipy as gp
from gurobipy import GRB
import pandas as pd
import matplotlib.pyplot as plt
import networkx as nx
import random

slot = [0, 1, 2, 3, 4, 5]  # cache slots
V = list(range(1, 14))     # materiali 1..13
req = {}
max_elementi = 100
totale = 0
i = 1

req = {
    1: [9, 5, 10, 11, 6, 3],
    2: [6],
    3: [1, 2, 12, 10, 6, 8],
    4: [11, 10],
    5: [13, 3, 6],
    6: [3, 7, 9],
    7: [6, 4],
    8: [1],
    9: [7, 10, 5],
    10: [12, 7, 6, 9, 13],
    11: [7, 4, 10, 3, 9, 13],
    12: [8, 3, 7],
    13: [12, 6, 4],
    14: [8, 12, 7, 9],
    15: [13, 8]
}

T = [0] + sorted(req.keys())

init_nodes = {
    0: 2,
    1: 9,
```

```
    2: 3,
    3: 6,
    4: 7,
    5: 10
}

df = pd.read_excel("cost_matrix_new.xlsx", header=None)
C = df.to_numpy()

def cost(i, j):
    return C[i - 1][j - 1]

m = gp.Model("milp")

p = m.addVars(slot, V, T, vtype=GRB.BINARY, name="p")
z = m.addVars(slot, V, V, T[:-1], vtype=GRB.BINARY, name="z")

m.setObjective(
    gp.quicksum(
        cost(i, j) * z[k, i, j, t]
        for k in slot
        for i in V
        for j in V
        for t in T[:-1]
    ),
    GRB.MINIMIZE
)

# Nodi iniziali
for k in slot:
    for i in V:
        m.addConstr(
            p[k, i, 0] == (1 if i == init_nodes[k] else 0)
        )

# Un materiale per slot
for k in slot:
    for t in T:
        m.addConstr(
```

```
            gp.quicksum(p[k, i, t] for i in V) == 1
        )

# Conservazione
for k in slot:
    for t in T[:-1]:

        for i in V:
            m.addConstr(
                gp.quicksum(z[k, i, j, t] for j in V)
                == p[k, i, t]
            )

        for j in V:
            m.addConstr(
                gp.quicksum(z[k, i, j, t] for i in V)
                == p[k, j, t + 1]
            )

# Request soddisfatte
for t in req.keys():
    for i in req[t]:
        m.addConstr(
            gp.quicksum(p[k, i, t] for k in slot) >= 1,
            name=f"req_{t}_{i}"
        )

# Materiali unici
for t in req.keys():
    for i in V:
        m.addConstr(
            gp.quicksum(p[k, i, t] for k in slot) <= 1
        )

m.optimize()

# Stampo costo totale
print("Costo totale:", m.ObjVal)
```

Bibliografia

[1] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[2] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, 1997.

[3] Marek Chrobak, Howard Karloff, Tom Payne, and Sundar Vishwanathan. New results on server problems. *SIAM Journal on Discrete Mathematics*, 4(2):172–181, 1991.

[4] Christian Coester, Roie Levin, Joseph Naor, and Ohad Talmon. Competitive algorithms for block-aware caching. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, page 32, 2022.

[5] Davide Duma, Stefania Marconi, and Ettore Lanzarone. Multi-task scheduling of in-hospital 3d-printed model production: A matheuristic approach. Submitted, 2026.

[6] Esteban Feuerstein. Paging more than one page. *Theoretical Computer Science*, 181(1):75–90, 1997.

[7] Elias Koutsoupias and Christos H. Papadimitriou. On the k-server conjecture. *Journal of the ACM*, 42(5):971–983, 1995.

[8] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990.

[9] Stefania Marconi, Valeria Mauri, Erika Negrello, Matteo Ghiara, Andrea Peri, Luigi Pugliese, Ferdinando Auricchio, Francesco Benazzo, and Andrea Pietrabissa. The experience of a clinical 3d printing laboratory: Current and future perspectives. *Minerva Orthopedics*, 21:395–406, 2021.

[10] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[11] Neal Young. *Competitive Paging and Dual-Guided On-Line Weighted Caching and Matching Algorithms*. PhD thesis, Princeton University, 1991.

[12] Neal Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.

Ringraziamenti

"Le pagine di questa tesi raccontano un percorso di studio; il cuore di questo percorso appartiene alle persone che mi hanno amato e sostenuto."

Il primo e più sentito ringraziamento va al Prof. Duma, per la disponibilità, la competenza e il sostegno offerti durante tutto il percorso. Un grazie speciale per i suoi modi di confrontarsi: mi hanno fatto riappassionare e scoprire un mondo nuovo. Grazie davvero di cuore.

Volevo inoltre ringraziare la Prof.essa Marconi coordinatrice del laboratorio 3D4Med, per aver condiviso i dati e gli studi effettuati presso il suo laboratorio, senza il suo contributo non sarebbe stato possibile intraprendere questa tesi.

Grazie alla mia famiglia, Daniela, Matteo e Archi, a cui ho deciso di dedicare questo progetto. *L'amore è la forza silenziosa che rende più leggeri anche i percorsi più impegnativi* e senza di loro non sarei mai riuscito a portare a termine questo percorso. Grazie per esserci sempre stati, per essere il mio riferimento, e per essere le persone a cui ambisco di essere. Non potevo chiedere genitori migliori.

Grazie allo zio Pippo, la mia vecchia trippa, e a tutti i nonni e zii che mi guardano lassù, in questi anni li ho sentiti sempre vicini e mi han dato la forza per affrontare le mie paure.

Grazie a Albi, Provi e Teoz, grazie per essermi stati vicini e accompagnato in questo percorso. Grazie a voi ho ritrovato energia e la voglia di non mollare. Grazie anche per le serate, le birre e i momenti di svago, perchè alla fine chi ha tanti fratelli è figlio unico.

Grazie a Tami, con il tuo modo di fare, la tua simpatia e la tua risata sei stata un punto di riferimento di questo percorso. La mia sveglia per le giornate in biblioteca. Difficile trovare delle amicizie così vere, ma con te sono sicuro di aver trovato il senso dell'amicizia. Sei stata il mio raggio di sole nelle giornate grigie.

Grazie alla mia amata S.G.G. vi vorrei ringraziare a uno a uno tutti voi siete stati e siete parte fondamentale della mia vita e del mio percorso, siete persone rare e sono felice di essere cresciuto con voi. Grazie Basse, grazie Gerry, grazie Piro, grazie Marco, grazie Dave, grazie Vale, grazie Sara, grazie Mosca, grazie Calo, grazie Greta, grazie Ele, grazie Alice, grazie Ale.

Grazie ai falsoni, Luca e Felox, e a Miky le giornate passate nelle varie trasferte mi han fatto bene all'animo, sono contento di aver trovato dei compagni di team come voi.

Grazie a tutti i ragazzi del Big, dal giorno uno a oggi, non dimenticherò mai questa community. Mi avete aiutato nei momenti di crisi e sono sono qui è anche grazie a voi.

Grazie ai QuasiAttori, salire sul palco con voi e mettersi in gioco mi ha dato la spinta a osare, a credere in quello che sono.

Grazie a Francesca, so che ci teneva ad essere ultima perché per ultime vanno le persone più importanti ed è davvero così. Non sarei qui senza te. Volevo dedicarti questo progetto insieme ai miei genitori. Non avevo mai trovato una persona fantastica come te, sei riuscita a farti spazio nel mio mondo, con delicatezza, gentilezza, ascoltando le mie insicurezze, e ogni volta che ero per terra sei riuscita ad avvicinarti, prendendomi la mano e guidandomi a rialzarmi. Senza te questo progetto non sarebbe arrivato a una fine, e per questo te ne sarò sempre grato. Alla mia zu, dedico un pezzo di questo traguardo, sperando in tanti altri da tagliare insieme.