Indice

Indice degli algoritmi

Capitolo I

Algoritmi on line e analisi competitiva

I.1 Introduzione agli algoritmi online

Gli algorimi *online* rispondono ad un'esigenza molto precisa: la necessità di prendere una decisione sulla base di una sequenza incompleta di dati in input, sienza alcuna conoscenza sulle richieste future.
Per questa loro caratteristica gli algoritmi online sono tutto sommato atipici rispetto a tutti gli algoritmi visti finora, i quali hanno come requisito fondamentale, affinché il loro comportamento sia corretto, quello di conoscere perfettamente tutti i dati in input.

Alcuni esempi di problemi "online" reali possono essere:

paging problem Man mano che arrivano richieste di pagine di memoria primaria dal processore, bisogna decidere subito quale pagina in memoria cache sostituire, in modo da massimizzare l'efficienza delle prestazioni, a fronte di una sequenza di richieste modellizzabile come una sequenza incompleta di dati;

routing Nell'ambito della telefonia, risulta cruciale il problema di come instradare una richiesta senza conoscere a priori la durata della telefonata e l'andamento delle richieste future;

radio-taxi Non meno banale risulta considerare il problema di inviare un taxi per servire una richiesta di un cliente in un dato punto della città, non sapendo quali saranno le richieste future.

Le applicazioni più recenti di questo tipo di algoritmi sono legate a campi quali lo studio dell'evoluzione delle strutture dati, il calcolo distribuito, movimento e inseguimento di obiettivi in robotica, telecomunicazioni, trasporti.

I.2 Analisi competitiva

Appare chiaro che lo strumento classico di analisi della complessità risulta in questo frangente poco rilevante, in quanto il generico algoritmo online ha di per se una comp-

lessità banale, in quanto deve semplicemente prendere una decisione alla volta cosa che può spesso essere fatta in tempo costante o lineare.
Diventa invece cruciale identificare delle opportune misure per valutare la performance dell'algoritmo. Il nostro intento sarà dunque quello di confrontare la prestazione dell'algoritmo online con il meglio che si potrebbe fare conoscendo il futuro, ovvero analizzando come si sarebbe comportato nel medesimo caso un algorimo *offline*, cioè un algoritmo che conosce completamente in anticipo la sequenza di input.
Questo tipo di analisi è detta analisi competitiva, e consiste nel calcolare, per ogni possibile sequenza di input, i costi di entrambi gli algoritmi (online e offline), per poi confrontarli. Confrontando i due algoritmi si ottiene il *fattore di competitività*.

Esempio I.1 (Stagione sciistica.). All'inizio della stagione sciistica dobbiamo decidere se acquistare l'attrezzatura necessaria oppure affittarla. Acquistare in blocco tutta l'attrezzatura ha un costo di 300 euro. Viceversa, affittare sci e scarponi ha un costo di 30 euro a weekend.
Ovviamente all'inizio della stagione non si ha conoscenza di quanti saranno i weekend in cui ci sarà la neve e sarà possibile andare a sciare, quindi bisogna prendere una tipica decisione online. Conoscendo le condizioni meteo future sarebbe immediato fornire come soluzione, ipotizzando j weekend utili:

$$min\{300, 30j\}$$

e questa costituirebbe la tipica soluzione offline ottima.
Senza invece conoscere a priori il futuro, analizziamo le possibili strategie da adottare:

comprare subito Comprando subito l'attrezzatura, avremmo un costo di 300 euro, indipendentemente dal numero di week end utili j. Nel caso in cui dopo il primo week end non nevichi più, a fronte di un costo ottimo di 30 euro che avremmo ottenuto affittando, spenderemmo 300 euro, ben dieci volte di più;

affittare sempre Adottando questa strategia, spenderemmo $30j$. Nel caso incontrassimo una sequenza infinita di week end utili j, il costo esploderebbe a fronte di un costo ottimo di 300 euro, che avremmo ottenuto acquistando direttamente l'attrezzatura;

strategia mista Una possibile soluzione potrebbe essere quella di affittare per k weekend e solo successivamente acquistare l'attrezzatura. Adottando questa strategia, con $k = 10$, avremmo un costo di:

$$costo = \begin{cases} 30j & \text{se } j \leq 10 \\ 30k + 300 = 600 & \text{se } j > 10 \end{cases}$$

Considerando che nel caso ottimo calcolato on line il costo sarebbe di 300, con questa soluzione andremmo a spendere il doppio. Risulta essere questa, dunque, la soluzione migliore.

È utile osservare come in questo esempio, per valutare quale strategia fosse la migliore, abbiamo dovuto metterci nel caso peggiore possibile per ogni strategia, come se ci fosse un avversario (in questo caso il destino) che volesse ostacolarci in ogni modo. ■

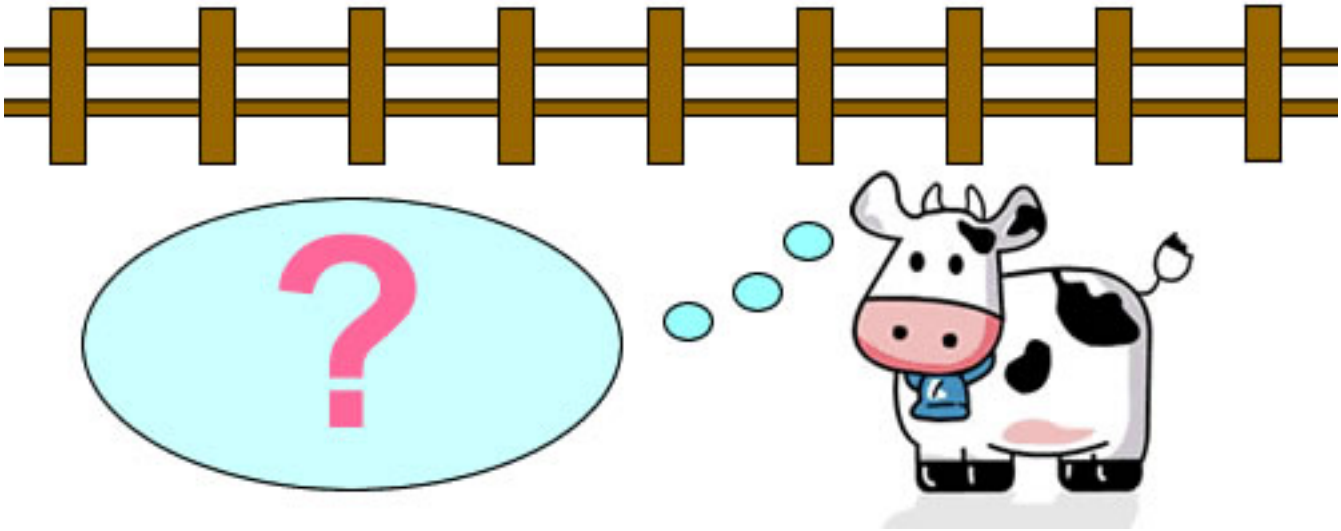
A questo punto è utile dotarsi di una serie di definizioni:

- $OPT \rightarrow$ algoritmo ottimo offline;

- $A \rightarrow$ algoritmo ottimo online;
- $\sigma \rightarrow$ sequenza di input;
- $C_{OPT}(\sigma) \rightarrow$ costo dell'algoritmo offline;
- $C_A(\sigma) \rightarrow$ costo dell'algoritmo online;

Diremo quindi che l'algoritmo A è C-competitivo se esiste una costante a tale che per ogni σ:

$$C_A(\sigma) \leq C \cdot C_{OPT}(\sigma) + a$$



Esempio I.2 (Problema della mucca smarrita.). Una mucca smarrita deve rientrare nel recinto, ma non sa dove si trova l'apertura. Per analizzare il problema, distinguiamo due casi:

1. La distanza d, cui si trova l'apertura è nota. La scelta più opportuna è quella di dirigersi a destra o a sinistra e muoversi per una distanza pari a d. Se l'apertura non viene trovata in questo modo, la mucca dovrà tornare indietro fino alla posizione di partenza, e quindi percorrere una distanza d in direzione opposta.

   In questo caso avremo che $C_{OPT}(\sigma) = d$ e $C_A(\sigma) = 3d$; la strategia risulta ottima, e l'algoritmo online è 3-competitivo.

2. La distanza d non è nota a priori. In questo caso ipotizziamo una strategia differente: la mucca procederà andando a sinistra e a destra raddoppiando la distanza da percorerre ogni volta che passa per il punto di partenza. Ad ogni passo k, la mucca percorrera una distanza 2^k, come mostrato in figura I.1.

   Ponendoci nell'ottica di un ideale avversario, intuiamo che il caso che maggiormente mette in crisi questa strategia è quello di posizionare l'apertura dopo un ε da dove si ferma la mucca prima di tornare indietro, ovvero a distanza $2^i + \varepsilon$. Analizzando la figura, possiamo dedurre la distanza totale percorsa dalla mucca:

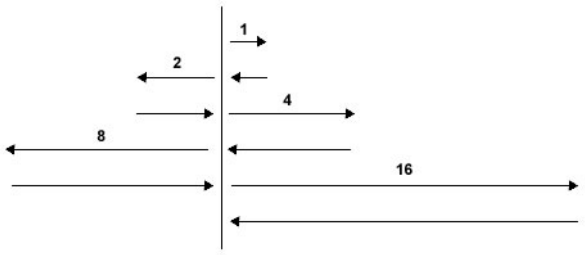


Figura I.1: Percorso della mucca

$$2\sum_{k=1}^{i+1} 2^k + 2^i + \varepsilon$$

Osserviamo inoltre che:

$$2\sum_{k=1}^{i+1} 2^k = 2(2^{i+2} - 2)$$

da cui possiamo dedurre che la distanza percorsa è:

$$2^i(2^3 + 1) + \varepsilon - 4 \leq 9(2^i + \varepsilon)$$

Possiamo di conseguenza affermare che l'algoritmo trovato è 9-competitivo, in quanto la distanza percorsa è 9 volte la soluzione ottima. Questo risultato risulta essere il meglio che si possa fare in questo contesto, e vale anche per lo stesso problema esteso su k direzioni. ■

I.3 Algoritmi randomizzati online

Come visto in altri contesti, spesso l'utilizzo del non determinismo permette di migliorare le prestazioni degli algoritmi. Anche nel caso degli algoritmi online questo può valere, sebbene sia necessario approfondire la caratterizzazione dell'avversario con cui confrontarsi.

I.3.1 Classificazione degli avversari

Fino ad ora ci siamo confrontati con un 'avversario' generico, mentre nella realtà possiamo individuare tre differenti tipologie di avversari:

Oblivious Fissa una sequenza σ a priori, e la adotta come strategia, indipendentemente dalle scelte dell'algoritmo online. E' sicuramente l'avversario meno temibile con cui confrontarsi.

Adattativo Genera la sequenza man mano che l'algoritmo online compie le sue decisioni. All'interno di questa categoria, possiamo ulteriormente distinguere tra avversari *adattativi online* (che calcolano il $C_{OPT}(\sigma)$ online, fissandone il valore man mano) e algoritmi *adattativi offline* (che invece calcolano il valore di $C_{OPT}(\sigma)$ conoscendo in precedenza la sequenza σ). Questi ultimi sono chiaramente gli avversari più coriacei.

Questa classificazione ci permette di evidenziare alcune importanti proprietà:

1. Se esite un algoritmo online randomizzato C-competitivo, contro un avversario adattativo offline, allora esiste un algoritmo online deterministico C-competitivo;

2. Se A è un algoritmo C-competitivo contro un avversario adattativo online ed esiste un algoritmo D-competitivo contro un avversario Oblivious, allora A è $(C \cdot D)$-competitivo contro un avversario adattativo offline.

La prima proprietà dice quindi che introducendo la randomizzazione, diventa fondamentamentale porre attenzione a quale tipologia di avversario si ha di fronte. Nel caso dell'avversario adattativo offline, non c'è nessun vantaggio dal punto di vista teorico ad adottare un algoritmo randomizzato, piuttosto che uno deterministico. La seconda proprietà invece da la misura di quanto il tipo di avversario che ho di fronte cambia la situazione in termini di competitività.

I.3.2 Strumenti per valutare la competitività

Concentrare la valutazione sulla singola operazione in questo contesto appare sicuramente insensato. Quello che occorre fare è invece considerare una valutazione media su tutta la sequenza σ. Per ogni elemento $t \in \sigma$, considerando $C_{OPT}(t)$ e $C_A(t)$, definiamo una funzione $\phi(t)$ che chiameremo funzione potenziale. La funzione potenziale fornirà una misura relativa alla differenza tra lo stato dell'algoritmo OPT e dell'algoritmo A. Detto questo, osserviamo che per affermare che l'algoritmo A è C-competitivo, occorre prima dimostrare che:

$$C_A(t) + \phi(t) - \phi(t-1) \leq C \cdot C_{OPT}(t)$$

ed estendendo il caso particolare su tutta la sequenza:

$$\sum_{t \in \sigma} [C_A(t) + \phi(t) - \phi(t-1)] = \sum_{t=1}^{m} C_A(t) + \phi(m) - \phi(0) \leq C \cdot \sum_{t \in \sigma} C_{OPT}(t)$$

Osservando che $\sum_{t \in \sigma} C_A(t) = C_A(\sigma)$, che $\sum_{t \in \sigma} C_{OPT}(t) = C_{OPT}(\sigma)$ e che la quantità $\phi(m) - \phi(0)$ è costante, l'espressione si riduce a:

$$C_A(\sigma) + cost \leq C \cdot C_{OPT}(t)$$

Da cui possiamo dedurre che l'algoritmo A è effettivamente C-competitivo.

I.3.3 Algoritmi di paginazione deterministici

Torniamo sul problema della paginazione, ovvero vogliamo studiare la migliore strategia per mantenere o sostituire le pagine in una memoria cache.
Adottiamo una strategia LRU (Least Recently Used): di fronte ad un page fault, togliamo dalla memoria cache la pagina richiesta meno recentemente. Con S_{LRU} indichiamo lo stato dell'algoritmo, che in questo caso possiamo identificare con lo stato della memoria cache. Con S_{OPT} indicheremo invece lo stato della memoria cache dell'algoritmo ottimo.
Definiamo quindi $S = S_{LRU} \backslash S_{OPT}$ come la misura di quante pagine sono presenti nella cache LRU e non nella cache OPT durante lo svolgimento dei rispettivi algoritmi.

Per valutare quanto recentemente sia stata richiesta una pagina, attribuiamo dei pesi opportuni $W[p]$ ad ogni pagina p, in modo che $W[p] < W[q]$ nel caso la pagina p sia stata richiesta meno recentemente della pagina q. Siano inoltre k la dimensione della cache e $a[p]$ la posizione nella coda, (infatti la cache può essere vista come una coda, in cui il primo elemento è quello usato più di recente) allora:

$$W[p] = k - a[p]$$

Possiamo a questo punto definire la funzione potenziale di cui abbiamo bisogno:

$$\phi(t) = \sum_{p \in S(t)} W[p]$$

Analizziamo a questo punto i differenti comportamenti degli algoritmi OPT e LRU, quando viene richiesta una pagina p all'istante t.

Per l'algoritmo OPT abbiamo che:

1. se $p \in S_{OPT} \Rightarrow C_{OPT}(t) = 0 \wedge \Delta\phi = 0$
2. se $p \notin S_{OPT} \Rightarrow C_{OPT}(t) = 1$; potrei quindi eliminare da S_{OPT} una pagina $q \in S_{LRU}$, modificando quindi in questo modo lo stato di S (infatti $S = S_{LRU} \backslash S_{OPT}$). Considerando poi che $0 \leq W[p] \leq k$, allora: $\Delta\phi \leq k$.

Per l'algoritmo LRU, invece, abbiamo che:

1. se $p \in S_{LRU} \Rightarrow C_{LRU}(t) = 0$, quindi $\Delta\phi = 0$
2. se $p \notin S_{LRU} \wedge p \in S_{OPT} \Rightarrow C_{LRU}(t) = 1$ e $\Delta\phi \leq -1$.
   Per dimostrare che $\Delta\phi \leq -1$, notiamo che esisterà sicuramente una pagina $p' \in S_{LRU} \wedge p' \notin S_{OPT}$: se LRU elimina p', si ottiene che $\Delta\phi \leq -1$ (caso minimo). Se invece la pagina p viene caricata nella cache, $W[p']$ diminuisce esattemente di 1, in quanto tutte le posizioni della coda scalerebbero di una posizione diminuendo il peso di p' di 1.

Riassumendo questi risultati, osserviamo che la funzione ϕ aumenta di al più k nel caso di un MISS nell'algoritmo OPT, mentre nel caso di un MISS in LRU diminuisce almeno di 1. Riscrivendo tutto in forma compatta, possiamo concludere che vale la seguente:

$$C_{LRU}(t) + \phi(t) - \phi(t-1) \leq k \cdot C_{OPT}(t)$$

Quindi LRU è k-competitivo.

I.4 Tecniche di progettazione

Piuttosto che l'analisi di competitività, ha un'importanza fondamentale approfondire il tema delle tecniche di progettazione degli algoritmi online.
Introduciamo il *problema dei k server*: in uno spazio metrico vengono disposti k serventi che devono servire delle richieste. La sequenza di richieste viene descritta mediante punti che compaiono in questo spazio.

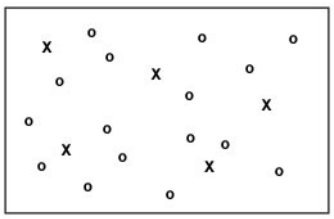


Figura I.2: Problema dei k server

Una volta ricevuta una richiesta, questa deve essere servita da uno dei k server 'recandosi' nella posizione in cui il punto compare. L'obiettivo è quello di minimizzare la distanza complessiva percorsa dai k server per servire la sequenza di richieste pervenute.

Osserviamo che in questo schema rientrano diversi problemi reali quali per esempio: inviare dei tecnici per servire gli utenti sul territorio e il problema del radio-taxi. Persino il problema della paginazione può essere ridotto ad un problema di k server, in cui si paga un costo pari ad 1 nel caso in cui il servente non sia nel luogo della pagina richiesta. Un'ulteriore applicazione potrebbe essere relativa al servire le richieste su disco fisso, in cui la distanza è relativa alla posizione della testina ed alla velocità di rotazione del disco.

Un banale algoritmo greedy potrebbe decidere di servire una richiesta con il server più vicino al punto in cui la richiesta compare. La politica adottata da un avversario sarà di conseguenza quella di posizionare la prossima richiesta nel punto che il server ha appena lasciato. Ad un avversario del genere bastano $k+1$ richieste per mettere in crisi l'algoritmo. In questo modo infatti, solo uno dei k server continuerà a fare avanti e indietro, creando un'evidente inefficienza.

Teorema I.1. *Dati k serventi, qualsiasi algoritmo deterministico A per k server ha competitività $\geq k$.*

Dimostrazione. Sia $S = \{p_1, p_2, ..., p_k, p_{k+1}\}$ l'insieme dei punti della sequenza minimale che mette in crisi l'algoritmo A, dove i primi k punti si riferiscono ai k serventi e sia $\sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ tale sequenza minimale di richieste.
Identifichiamo ora diverse strategie di fronte a queste richieste, il cui fattore caratteristico è costituito da quale servente si decide di spostare ad ogni richiesta. In quest'ottica, denominiamo $B_i(\sigma_j)$ l'algoritmo online generico che sposta il server i all'arrivo della richiesta σ_j. Si intuisce facilmente che, ad ogni richiesta, ciascuno di questi algoritmi ha un punto scoperto diverso. Quindi ad ogni richiesta ci sarà un solo algoritmo che muove un server verso quella richiesta, mentre gli altri algoritmi avranno un server già posizionato nel punto in cui arriva la richiesta.

Pertanto se considero a questo punto tutti gli algoritmi contemporaneamente, sommandone il costo:

$$\sum_{l=1}^{k} cost_{Bl}(\sigma) = \sum_{j=1}^{k} d(p_j, p_{k+1}) + \sum_{j=1}^{m-1} d(p_j, p_{j+1}) \tag{I.1}$$

Parafrasando la formula, possiamo dire che il costo totale è uguale alla somma del costo della prima richiesta ($\sum_{j=1}^{k} d(p_j, p_{k+1})$), effetuato da tutti gli algoritmi per forza, in quanto il punto p_{k+1} è inizialmente scoperto, più un termine ($\sum_{j=1}^{m-1} d(p_j, p_{j+1})$) che mi indica il fatto che ad ogni richiesta solo un algoritmo dei k muove un server. Analizzando quest'ultimo termine possiamo vedere come esso possa rappresentare il comportamento di un solo algoritmo che ad ogni richiesta sposta un server dal punto vicino. Tale algoritmo corrisponde proprio al nostro algoritmo A, poiché σ è proprio la sequenza che lo mette in crisi. Quindi:

$$cost_A(\sigma) = \sum_{j=1}^{m-1} d(p_j, p_{j+1})$$

Osserviamo che:

$$k \cdot cost_{OPT}(\sigma) \leq k \cdot min_{l=1..k} \cdot cost_{Bl}(\sigma) \leq \sum_{l=1}^{k} cost_{Bl}(\sigma)$$

Quindi dalla (I.1) possiamo arrivare a dire che :

$$k \cdot cost_{OPT}(\sigma) \leq cost_A(\sigma) + \alpha$$

Dove $\alpha = \sum_{j=1}^{k} d(p_j, p_{k+1})$.
Quindi la competitività di ogni algoritmo non può essere minore di k. □

Ritornando all'approccio greedy che serve una richiesta con il server più vicino, puntualizziamo il fatto che la competitività di tale strategia non è limitata, in quanto posso far muovere di continuo un solo server. Occorre dunque dotarsi di uno strumento più potente per cercare di migliorare la situazione.

I.4.1 Tecnica della Work Function

L'idea base è quella di combinare più strategie senza fissarne una in particolare, in modo da aumentare la flessibilità dell'algoritmo.

Fino ad ora abbiamo visto un algoritmo che chiameremo *GREEDY LOCALE*. L'alternativa a questa strategia è costituita da quello che da qui in poi chiameremo *GREEDY STORICO*.
Il greedy storico considera la sequenza parziale di dati ricevuti fino all'istante t, e su questa base ragiona come un algoritmo offline per decidere ad ogni istante t. In questo modo determina per ogni t un corrispondente stato ottimale X_t (in questo caso lo stato corrisponde con la configurazione dei server) e successivamente compie tutte le operazioni necessarie per portarsi dallo stato X_{t-1} allo stato X_t.

È immediato considerare che una strategia di questo genere potrebbe non risultare così efficace: potrebbe infatti esistere un percorso diretto tra lo stato iniziale e quello finale, e quindi sarebbe inutile (e maggiormente oneroso) vincolare la soluzione a passare per un insieme di stati intermedi X_t.

Con lo strumento delle WORK FUNCTION, l'intenzione diventa quella di combinare in modo proficuo il greedy locale con quello storico. Con questa operazione infatti l'algoritmo diventa $(2k-1)$-competitivo. Per descrivere in modo più formale le Work Function, occorre dotarsi di alcuni formalismi:

- $opt_t(\sigma, X)$ = costo ottimo che porta nello stato X servendo σ fino a t e partedo dallo stato iniziale X_0;
- $D(X, Y)$ = costo minimo per andare dallo stato X allo stato Y.

Le $opt_t(\sigma, X)$ sono dette Work Function e si calcolano in maniera ricorsiva, secondo il seguente procedimento:

$$opt_t(\sigma, X) = min_{Y:\sigma_t \in Y}\{opt_{t-1}(\sigma, Y) + D(Y, X)\}$$

Osserviamo alcune proprietà delle Work Function:

- sono monotone crescenti;
- possiamo scrivere qualcosa simile alle diseguaglianze triangolari, ovvero affermare che passare da uno stato X a uno stato Y, e successivamente da uno stato Y ad uno stato Z, ha un costo maggiore o al limite uguale al costo di passare direttamente dallo stato X allo stato Z. Ovvero: $opt_t(\sigma, X) \leq opt_t(\sigma, Y) + D(Y, X)$;
- $opt_t(\sigma, X) = min_{x \in X}\{(\sigma, X \backslash \{x\} \cup \sigma_t) + d(x, \sigma_t)\}$, ovvero calcolo lo stato migliore spostando a servire la richiesta σ_t il server x che minimizza la somma tra: lo stato ottimo (calcolato offline) che sposta il server x e la distanza che compio muovendo x nel punto dove si trova la richiesta σ_t da servire;

L'algoritmo $WORK\ FUNCION\ ALGORITHM(r)$, dove r è la richiesta nell'istante t, sarà quindi fatto così:

Algoritmo I.1 WFA(r)

1: muoviti nello stato X, con $r \in X$ che minimizza:
2: $opt_t(\sigma, X) + D(X, X_{t-1})$

In questo modo viene eseguito un bilanciamento delle componenti relative al greedy storico (la prima) e al greedy locale (la seconda).

Viene spontaneo chiedersi se esista una soluzione che migliori la competitività $(2k-1)$. L'unico modo per fare qualche passo avanti è adottare una versione randomizzata dell'algoritmo. In particolare, potremmo pensare di modificare l'approccio greedy spostando il server più vicino con una certa probabilità.

I.4.2 Algoritmo Randomizzato Harmonic

Consideriamo una richiesta r, che arrivi con una configurazione di server X_i. Sia quindi p_i la probablilità di spostare il server i. Definiamo questa quantità come:

$$p_i = \frac{1/d(x,r)}{\sum_{l=1}^{k} 1/d(x_l,r)}$$

Questo ci porta ad una complessità (calcolata contro un avversario adattativo online) di questo tipo:

$$\frac{k(k+1)}{2} \leq c \leq \frac{5}{4}k \cdot 2^k - 2k$$

La caratteristica positiva di questo algoritmo sta nel fatto di essere computazionalmente molto leggero ($O(k)$). Trovare un algoritmo randomizzato in grado di scendere sotto k, in questo momento, è ancora argomento di ricerca.

I.4.3 Tecnica del Balancing

Questa tecnica viene utilizzata per evitare, in modo deterministico, il problema dell'inefficienza dell'algoritmo greedy locale. Si usa l'idea seguente: memorizzando quanto ogni server abbia camminato fino all'istante t, si decide di spostare quello che ha camminato di meno.
Un'utile applicazione di questo algoritmo potrebbe essere quella in cui i k server sono dei tassisti, in questo caso è auspicabile che tutti percorrano una distanza simile tra loro in modo che il lavoro venga spartito in maniera il più possibile uguale.

Siano $i = 1, \ldots, k$ i server del sistema, e sia D_i la distanza complessiva percorsa fino all'istante t dal server i.

Secondo una strategia di $BALANCING$, di fronte ad una richiesta r non coperta, verrà inviato il server che minimizza la quantità $D_i + d(x_i, r)$. In questo modo, pondero la decisione considerando come cifra di merito la somma della distanza percorsa dal server i e la distanza che il server i dovrebbe percorrere per servire r.

Con una metrica di $k+1$ punti, questo algoritmo risulta essere k-competitivo, quindi otterrei una prestazione ottimale. Sfortunatamente, nel caso i punti siano più di $k+1$, l'algoritmo *non* è competitivo, e la prestazione degrada in maniera incontrollata.

Esempio I.3 (Multiprocessor Scheduling). La tecnica del balancing è molto utilizzata nell'ambito degli algoritmi online. In particolare in questa sezione applicheremo questa tecnica ad un problema di *scheduling di multiprocessori*.

Sia m il numero di macchine identiche a nostra disposizione, $\sigma = \{1, \ldots, n\}$ la sequenza di lavori da compiere. Considerando che il lavoro j ha durata p_j, e che questa durata è nota quando arriva j, vogliamo assegnare i lavori, non appena arrivano ad una delle m macchine, in modo da minimizzare il makespan ovvero il tempo di completamento dell'ultimo lavoro che termina la sequenza.
Per farlo posso assegnare il lavoro in arrivo alla macchina più scarica, facendo crescere il carico delle macchine in modo bilanciato, ed ottenendo un algoritmo $(2 - 1/m)$-competitivo.
Dimostriamolo:

Siano $T_B(\sigma)$ il tempo di fine del nostro algorimo e $T_{OPT}(\sigma)$ il tempo di fine dell'algoritmo ottimo. Supponiamo che l'istante t_1 sia l'ultimo istante in cui tutte le macchine sono attive e $t_2 = T_B(\sigma) - t_1$.

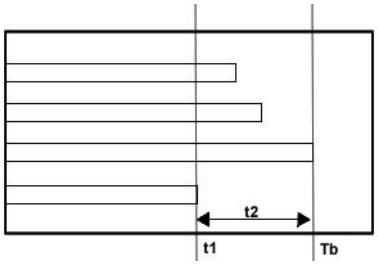


Figura I.3: Scheduling: svolgimento

Osserviamo dalla figura I.3 che l'ultimo lavoro deve essere cominciato prima di t_1 e deve durare più di t_2. In particolare, possiamo puntualizzare il fatto che:

$$t_2 \leq max(p_k)$$

$$t_1 \leq \frac{1}{m} \cdot \sum p_k$$

Allora possiamo dedurre che:

$$T_{OPT}(\sigma) \geq max\{\frac{1}{m}\sum p_k, max(p_k)\}$$

da cui possiamo ricavare che:

$$T_B(\sigma) = t_1 + t_2 \leq 2 \cdot max\{\frac{1}{m}\sum p_k, max(p_k)\} \leq 2 \cdot T_{OPT}(\sigma)$$

Da questa dimostrazione notiamo quindi che l'algoritmo è 2-competitivo, con un'analisi più raffinata sui tempi t_1 e t_2 (che ometteremo), si può dimostrare che l'algoritmo è $(2 - \frac{1}{m})$-competitivo.
La sequenza che manda in crisi questa strategia è $\sigma = \{1, 1, \ldots, 1, 1, m\}$, con un numero multiplo di m di pezzi di lunghezza 1 (o comunque di lunghezza molto minore rispetto all'ultimo pezzo). Per migliorare questa situazione, quello che si può fare è sbilanciare in parte il carico delle macchine, in modo da lasciare libero una parte di carico su una macchina, cioè dello spazio libero di riserva da utilizzare nel caso arrivi un lavoro molto lungo. Questo tipo di modifica prende il nome di strategia $INBALANCE$. ■

I.4.4 Tecnica di Inbalance

Per progettare una politica sbilanciata, si definisce una costante $\alpha = 1.945$, e si indica con h_i il carico della i-esima macchina più scarica.

In particolare definiamo A_i il carico medio delle $i-1$ macchine più scariche, ponendo logicamente $A_0 = \infty$. Il job k viene assegnato alla macchina j più carica tale che:

$$h_j + p_k \leq \alpha \cdot A_j$$

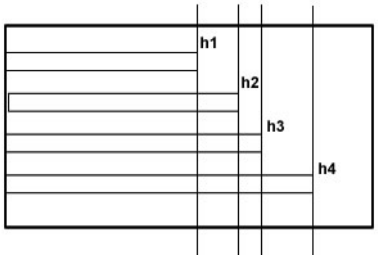


Figura I.4: Tecnica di Inbalance: svolgimento

Per esempio, in riferimento alla figura I.4, se è in arrivo il lavoro k con durata p_k, verrà dapprima calcolato A_4, cioè la media delle macchine h_1, h_2, h_3 e se $h_4 + p_k \leq \alpha \cdot A_4$ allora verrà assegnato k alla macchina 4. Altrimenti si proverà ad assegnare il lavoro a 3, calcolando A_3 e verificando la condizione $h_3 + p_k \leq \alpha A_3$, e così via finché non verrà trovata una macchina adatta.
In pratica si assegna il lavoro ad una macchina carica, ma valutando che non ecceda troppo rispetto al carico medio.
Si dimostra che questo algoritmo è 1.945-competitivo.

I.4.5 Problema del Load Balancing

In diversi casi reali, la durata p_j del lavoro j non è nota al momento di arrivo (es. la durata di una telefonata non è nota a priori).
Consideriamo quindi un problema con m macchine uguali e con una sequenza di lavori σ e associamo un peso w_k ad ogni lavoro.
Definiamo $l_i(t) = \sum_{k \in S_i} w_k$, dove S_i è l'insieme dei lavori assegnati ad i. Intuitivamente $l_i(t)$ è il carico della macchina i al tempo t.
Consideriamo inoltre il fatto che ogni lavoro può essere eseguito da un sottoinsieme di macchine.
L'obiettivo è di minimizzare il $max_i(l_i(t))$.

Applicando una politica greedy, la competitività aumenta notevolmente, passando da 2 a $O(m^{2/3})$, rispetto al caso in cui i lavori possono essere eseguiti su ogni macchina.
È possibile dimostrare (ma noi non lo faremo) che il lower bound è $\Omega(\sqrt{m})$.

I.4.5.1 L'algoritmo di Robin Hood

Un algoritmo che riesce a raggiungere il lower bound per il problema del Load Balancing è Robin Hood. Per raggiungere questo scopo classifichiamo prima di tutto le macchine in base al loro carico nel modo seguente:
Una macchina è *ricca* se $l_i(t) \geq \sqrt{m} \cdot L$;
Una macchina è *povera* se $l_i(t) < \sqrt{m} \cdot L$;
dove L è una stima del carico ottimo (cioè quello calcolato offline): $L \leq OPT$.
L'algoritmo funziona assegnando di volta in volta il lavoro in arrivo ad una macchina povera (se esiste tra quelle su cui può essere eseguito), altrimenti alla macchina diventata ricca più di recente. Il punto centrale dell'algoritmo risulta quindi essere in che modo una

macchina viene classificata come povera o ricca, cioè in ultima analisi in che modo viene calcolato L.
Per ogni lavoro k della sequenza, L viene definito nel seguente modo:

$$L \leftarrow max\{L, w_k, \frac{1}{m}(w_k + \sum_{i=1}^{m} l_i(t))\}$$

Teorema I.2. *Robin Hood è $O(\sqrt{m}) - competitivo$*

Dimostrazione. Osserviamo preliminarmente due risultati intuitivi:

1. ad ogni istante t ci sono al massimo $\lceil\sqrt{m}\rceil$ macchine ricche. Infatti se non fosse così, facendo la somma delle singole macchine povere supereremmo mL, ma per definizione $L \leq \frac{1}{m}\sum w_i$.
2. In ogni istante $L \leq OPT$.

Si consideri ora la macchina i all'istante t.
Si dimostra che Robin Hood è $O(\sqrt{m}) - competitivo$, dimostrando che:

$$l_i(t) \leq \lceil\sqrt{m}\rceil(L + OPT) \tag{I.2}$$

Distinguiamo ora 2 casi:

1. Se i è povera $l_i(t) \leq \sqrt{m} \cdot L$, quindi in questo caso la (I.2) è banalmente verificata
2. Se i è ricca $l_i(t) \geq \sqrt{m} \cdot L$, diventa meno immediato dimostrare che l'errore rispetto all'ottimo risulta comunque contentuto.

Nel secondo caso bisogna introdurre la seguente notazione per dimostrare che vale la (I.2):

- t_0: istante in cui la macchina i è diventata ricca.
- $M(t_0)$: insieme di macchine che sono ricche in t e che sono diventate ricche nel tempo $t' \leq t_0$ (Quindi anche $i \in M(t_0)$).
- $S_i(t_0)$: lavori assegnati ad i dopo l'istante t_0 (lavori assegnabili solo a macchine in $M(t_0)$).
- $j = |M(t_0)|$: numero di macchine che si possono utilizzare per i lavori.

A questo punto notiamo che il meglio che OPT può fare è di spalmare tutti i lavori su tutte le macchine ugualmente, cioè $OPT \geq \frac{1}{j}\sum_{k\in S_i(t_0)} w_k$.
Inoltre si nota che $j \leq \lceil\sqrt{m}\rceil$ perché le i sono tutte macchine ricche. Si presentano quindi 2 casi:

1. $j \leq \lceil\sqrt{m}\rceil - 1$

   Se chiamiamo q il lavoro che ha fatto diventare ricca i, si ha che:

   $$l_i(t) \leq \lceil\sqrt{m}\rceil \cdot L + w_q + \sum_{k\in S_i(t_0)} w_k \tag{I.3}$$

   Dove:

$\lceil\sqrt{m}\rceil \cdot L$ è il carico della macchina prima di diventare ricca

$w_q \leq OPT$ è il peso del lavoro q

e infine il peso dei lavori assegnati successivamente risulta essere:

$$\sum_{k \in S_i(t_0)} w_k \leq jOPT \leq (\lceil\sqrt{m}\rceil - 1)OPT \text{ dalla (I.3)}$$

Quindi si ottiene che:

$$l_i(t) \leq \lceil\sqrt{m}\rceil(L + OPT)$$

2. $j = \lceil\sqrt{m}\rceil$

   In questo caso si ha che:

   $l_i(t_0) = \sqrt{m} \cdot L$

   $l_i(t) \leq \sqrt{m} \cdot L + \sum_{k \in S_i(t_0)} w_k \leq \lceil\sqrt{m}\rceil(L + OPT)$ □

I.4.6 Il Problema del Bin Packing

Consideriamo una sequenza di oggetti $\sigma = \{1, \ldots, n\}$ e ogni oggetto $i \in \sigma$ ha una dimensione a_i, con $0 < a_i \leq 1$. Inoltre sono dati dei contenitori (bin) di capacità 1. L'obiettivo è distribuire gli oggetti nei contenitori man mano che arrivano in modo da minimizzare il numero di bin utilizzati.

Possibili applicazioni per un problema di questo tipo sono:

- Riempire camion per il trasporto merci, man mano che gli oggetti da trasportare arrivano, in modo da minimizzare il numero di camion utilizzati.
- Nel campo della televisione, durante la proiezione di un film o di una trasmissione, quando non si è a conoscenza di quanti spot arriveranno da mandare in onda, bisogna decidere man mano che gli spot arrivano quando inserire un break pubblicitario in modo da minimizzare il numero totale dei break.

Il problema in questione è simile come natura al problema già incontrato dello zaino, in quanto è come se si dovesse prendere tutti gli oggetti presenti, minimizzando il numero di zaini utilizzati. Per questa sua natura il problema equivalente offline è perciò NP-hard. Nonostante ciò esistono una serie di algoritmi competitivi per la soluzione del problema online.
Nel seguito per descrivere come funzionano i vari algoritmi utilizzeremo come esempio quello dei camion per il trasporto merci.

Il più banale di questi algoritmi è l'agloritmo $NEXT\ FIT$, che consiste nel riempire un camion alla volta, finché il camion non ha più spazio per contenere l'oggetto successivo nella sequenza.

Algoritmo I.2 NF(σ)

1: **if** $a_i \leq i - livello(B_j)$ **then**
2: assegna a_i a B_j
3: **else**
4: "chiudi" B_j; $j \leftarrow j + 1$
5: assegna a_i a B_j

Si può dimostrare (la dimostrazione utilizza tecniche simili a quelle utilizzate per il problema del multiprocessor scheduling) che $NF(\sigma)$ è 2-competitivo.
Un esempio di sequenza di un avversario che manda in crisi $NEXT\ FIT$ è la seguente:

$$\sigma = \{\frac{1}{2}, \frac{1}{2n}, \frac{1}{2}, \frac{1}{2n}, \frac{1}{2}, \frac{1}{2n}, \cdots\}$$

In questo caso, poiché è impossibile che più di due oggetti consecutivi trovino spazio nel camion, in quanto tre oggetti eccederebbero lo spazio possibile di $\frac{1}{2n}$, si ha che ogni due oggetti viene utilizzato un nuovo camion, con un grosso spreco di spazio.
La soluzione ottimale sarebbe quella di tenere 2 camion "attivi" alla volta e riempirne uno con i pezzi da $\frac{1}{2}$ e l'altro con i pezzi da $\frac{1}{2n}$. In questo caso se la sequenza è lunga $4n$ servono n camion per i pezzi da $\frac{1}{2}$ ed 1 camion per i pezzi da $\frac{1}{2n}$, per un totale di $n + 1$ camion.
L'alogritmo $NEXT\ FIT$ con la stessa sequenza utilizza ben $2n$ camion.
Da questo esempio si vede quindi chiaramente il fatto che $NF(\sigma)$ è 2-competitivo.
Nonostante la competitività non eccelsa, $NF(\sigma)$ ha il vantaggio di utilizzare un solo camion alla volta, fatto che può tornare utile, per esempio nel caso in cui i parcheggi in cui avviene il carico merci siano di dimensioni modeste.

Un algortimo più evoluto del precedente è $FIRST\ FIT$ ($FF(\sigma)$). $FIRST\ FIT$ ha a disposizione un numero teoricamente infinito di camion e man mano che i pezzi arrivano vengono messi nel primo camion che li può contenere.
Utilizzando $FF(\sigma)$ applicato alla sequenza critica vista in precedenza per $NF(\sigma)$, avrei che due camion (in particolare il primo e l'n-esimo) verrebbero riempiti con 1 oggetto da $\frac{1}{2}$ ed n oggetti da $\frac{1}{2n}$, mentre i rimanenti solo con oggetti da $\frac{1}{2}$.
Un risultato che si ottiene è che $FF(\sigma) \leq \lceil 1, 7 \cdot OPT \rceil$
Un esempio in cui raggiungo l'upper bound è dato dalla seguente sequenza:

$$\sigma = \{\frac{1}{7} + \varepsilon, \frac{1}{7} + \varepsilon, \cdots, \frac{1}{3} + \varepsilon, \frac{1}{3} + \varepsilon, \cdots, \frac{1}{2} + \varepsilon, \frac{1}{2} + \varepsilon, \cdots\}$$

Nella quale ci sono $6n$ oggetti da $\frac{1}{7} + \varepsilon$, $6n$ oggetti da $\frac{1}{3} + \varepsilon$, $6n$ oggetti da $\frac{1}{2} + \varepsilon$.
Quindi con $FF(\sigma)$ si hanno:

- 6 oggetti da $\frac{1}{7} + \varepsilon$ per ogni camion, per un totale di n camion.
- 6 oggetti da $\frac{1}{3} + \varepsilon$ per ogni camion, per un totale di $3n$ camion.
- 6 oggetti da $\frac{1}{2} + \varepsilon$ per ogni camion, per un totale di $6n$ camion.

Quindi il totale di camion utilizzati con $FF(\sigma)$ è $10n$

In questo caso il metodo migliore sarebbe quello di mettere su ogni camion ognuno dei 3 oggetti. In questo modo otterremmo che il numero totale di camion utilizzati sarebbe di $6n$, a patto che ε sia tale che:

$$\frac{1}{7} + \frac{1}{3} + \frac{1}{2} + 3\varepsilon = \frac{41}{42} + 3\varepsilon \leq 1$$

In questo modo si ha esattamente $FF(\sigma) \leq \lceil 1.7 \cdot OPT \rceil$, cioè:

$$10 \leq \lceil 1.7 \cdot 6n \rceil = \lceil 10.2n \rceil = 11$$

Nel tentativo di rendere il più competitivi possibile gli algoritmi online per questo problema, sono state pensate numerose varianti degli algoritmi visti in precedenza, passiamo in rassegna qui le principali:

- L'algoritmo $BEST\ FIT$ assegna l'oggetto corrente al bin più pieno. Si ha che $BF(\sigma) \leq \lceil 1.7 \cdot OPT \rceil$.
- $WORST\ FIT$ assegna l'oggetto corrente al bin più vuoto. Lo svantaggio in questo caso è che si rischia di trovarsi con troppi bin aperti. Ho che: $WF(\sigma) \leq 2 \cdot OPT - 1$.
- $ALMOST\ WORST\ FIT$ assegna l'oggetto corrente al secondo bin meno pieno. Ho che $AWF(\sigma) \leq \lceil 1.7 \cdot OPT \rceil$.

Nonostante tutte queste alternative non si è ancora trovato finora nessun algoritmo in grado di raggiungere il lower bound della competitività per il problema online, che si può dimostrare essere di 1.5.
Si è notato inoltre, da applicazioni pratiche, che tutti gli algoritmi citati, nel caso pessimo sono migliori di $FIRSTFIT$, ma nel caso medio $FIRST\ FIT$ risulta l'algoritmo migliore.

Molte volte capita che ci sia una limitazione rispetto al numero di bin che è possibile tenere aperti contemporaneamente. Per esempio nel caso dei camion per il trasporto merci, la dimensione del piazzale in cui avviene il carico darà un limite al numero di camion che è possibile tenere contemporaneamente nel piazzale.

Chiamiamo k il numero di bin che è possibile tenere aperti contemporaneamente. Banalmente se $k = 1$, l'unico algoritmo che si può utilizzare è $FIRST\ FIT$. Nel caso in cui $k > 1$, a seconda di quanto vale k si possono avere soluzioni migliori o peggiori in base all'algoritmo che si sceglie di usare. È possile quindi cercare tra gli algoritmi visti in precedenza il migliore rispetto al k del mio problema.

Esiste tuttavia un algoritmo specifico per questo problema in cui k è limitato: l'algoritmo $HARMONIC\text{-}\ K$ $(H_k(\sigma))$.
$H_k(\sigma)$ classifica i bin in base agli oggetti che possono ospitare, nel modo seguente:
L'algoritmo divide l'intervallo $(0, 1]$ in k sottointervalli:
$I_k = (0, \frac{1}{k}]$
$I_{k-1} = (\frac{1}{k}, \frac{1}{k-1}]$
$\dots$
$I_1 = (\frac{1}{2}, 1]$

Ognuno di questi sottointervalli ha un'ampiezza crescente, quindi per ogni oggetto l'algoritmo sceglie l'intervallo (I_h) compatibile con la dimensione dell'oggetto. I bin inoltre sono in corrispondenza uno a uno con gli intervalli, quindi si può parlare di bin di tipo h. L'idea base dell'agloritmo è quindi quella di assegnare l'oggetto di tipo h (classificato in base all'intervallo I_h di appartenenza) al bin di tipo h.
Per come si comporta l'algoritmo si può notare che i bin di tipo 1 potranno contenere al più un oggetto, quelli di tipo 2, 2 oggetti e così via.
Si può dimostrare che $H_k(\sigma)$ è 1.695-competitivo se $k \geq 7$.
In realtà la competitività reale di questo algoritmo rispetto ai precedenti è ancora maggiore perché in questo caso l'algoritmo OPT offline con cui ci confrontiamo è cambiato, in quanto il problema stesso è cambiato ed è più difficile.

I.4.7 Problemi di movimento in un ambiente noto

Una tipica problematica online è quella legata ai problemi di movimento in un ambiente noto, di cui abbiamo già visto il semplice esempio della mucca smarrita.

Tutti i problemi di movimento sono tipici problemi online, in quanto ad ogni passo bisogna prendere velocemente una decisione sulla direzione da seguire, la velocità da prendere, se imboccare o no una certa strada, e tutte queste decisioni devono tenere conto delle varie informazioni sull'ambiente circostante che man mano arrivano.

Un tipico problema di movimento online è il cosiddetto *problema del viaggiatore canadese*, in cui si viene a sapere se le strade sono innevate o libere solo quando si arriva al nodo di partenza.

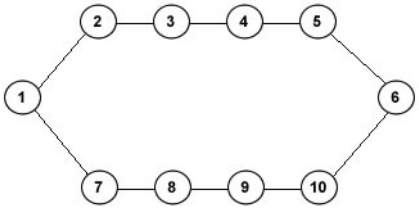


Figura I.5: Viaggiatore canadese: Problema.

Con riferimento alla figura I.5 il problema è il seguente: ogni arco del grafo ha un peso, inizialmente si è nel nodo 1 (il nodo di partenza) e si conoscono solo i pesi degli archi uscenti dal nodo in cui ci si trova. Ogni volta che ci si muove in un nodo, per esempio il nodo 7, si viene a conoscere oltre al peso dell'arco uscente dal nodo 7, anche il peso dell'arco uscente dal nodo opposto al nodo in cui ci si trova, in questo caso il nodo 2. Man mano che si percorre una strada le nuove informazioni raccolte sull'altra strada potrebbero rendere più conveniente tornare indietro e percorrere una strada alternativa. Il problema consiste nel trovare la strategia migliore che consente di arrivare alla destinazione minimizzando il peso del cammino percorso.

Un campo in cui il movimento in ambiente noto riveste una importanza notevole è nel movimento dei robot. Esistono un'infinità di problemi relativi al moto dei robot, per

adesso noi vedremo il seguente:
Il robot parte da un punto iniziale e deve raggiungere la destinazione che dista, in linea retta dal punto di partenza, n passi.
Il robot ha un sensore tattile grazie al quale percepisce gli ostacoli, che successivamente aggira. Non avendo una conoscenza a priori della dislocazione dei differenti ostacoli, l'obiettivo del robot è quello di minimizzare la strada percorsa per raggiungere la destinazione.

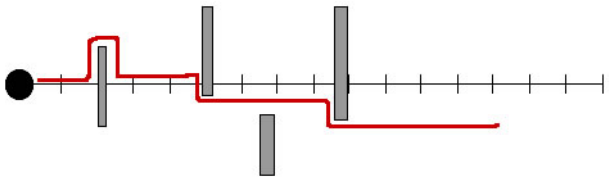


Figura I.6: Movimento del Robot

Si può dimostrare che non esistono strategie online a competitività limitata. Consideriamo infatti un avversario che disponga opportunamente ad ogni passo gli ostacoli di lunghezza n. In questo caso la strada totale percorsa per arrivare a destinazione evitando gli ostacoli è dell'ordine di $O(n^2)$.
Se invece il robot conoscesse in anticipo gli ostacoli (offline) potrebbe fare molto meglio.

Si può dimostrare che esiste una cordinata y tale che a quella coordinata si attraversano al massimo $\sqrt{n}$ ostacoli, $y \leq n \cdot \sqrt{n}$

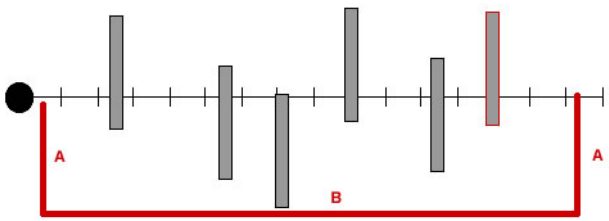


Figura I.7: Percorso del Robot

Appoggiandosi su questa assunzione, possiamo considerare il percorso in figura I.7. Prima di tutto il robot raggiunge la quota y attraverso il cammino A, percorrendo la distanza $n\sqrt{n}$; la stessa distanza viene percorsa al ritorno per tornare in quota, per un totale di $2n\sqrt{n}$. Nel percorso B, invece, dobbiamo considerare che il robot compie n passi incontrando $\sqrt{n}$ ostacoli, spendendo quindi $n\sqrt{n}$ per aggirarli.
Da questo ragionamento possiamo evincere che la strada percorsa in tutto risulta pari a $n + 3n\sqrt{n}$, cioè $O(n^{3/2})$.
Una possibile estensione di questa strategia è stata studiata nel caso di ostacoli di forma quadrata.

Si può considerare anche una variante del problema applicato ad un robot con visione. Una peculiarità di questo problema è che il raggio visivo della telecamera limita chiaramente la percezione degli ostacoli da parte del robot.