RELAZIONE PROGETTO ALGORITMI FEBBRAIO 2023

OSSERVAZIONI DA SATELLITE

Immagine che contiene testo

Descrizione generata automaticamente

Margherita Marina Pitimada

Matricola 933155

*Università degli Studi di Milano*

*Anno Accademico 2021-2022*

*Capitolo 1: Il problema*

**Il problema:** Consideriamo le riprese da satellite della superficie terrestre secondo due semiorbite, una ascendente (da Sud a Nord) e una discendente (da Nord a Sud). Le due orbite risultano inclinate rispetto all’asse terrestre a causa della rotazione della Terra, questo fa sì che le due riprese si incorcino trasversalmente. Modellizziamo quindi le due riprese come colonne e righe (rispettivamente ripresa Ascendente e Discendente) di una matrice (approssimazione di tutta la superficie terrestre), le cui celle corrispondono alle intersezioni delle due semiorbite. Dal momento che non tutte le zone della superficie terrestre hanno lo stesso interesse, associamo a ogni cella un premio, corrispondente a quanta porzione notevole è contenuta nell’immagine, e due pesi, uno per riga e uno per colonna, che indicano la memoria occupata per salvarla. Poichè la memoria dei satelliti non è illimitata, abbiamo dei vincoli sulla capacità massima di ripresa.

Il nostro obiettivo è dunque quello di massimizzare la porzione notevole di ripresa (premio), rimanendo nel vincolo di capacità consentita (i.e. la somma dei pesi delle celle corrispondenti a una stessa riga/colonna non deve superare lo spazio di memoria disponibile).

Introduciamo due ulteriori vincoli ragionevoli:

1. Evitare la doppia ripresa incassando due volte il premio associato: da qui la scelta della direzione più promettente in base al rapporto maggiore tra premio e peso;
2. Evitare la doppia ripresa incassando una sola volta il premio, ad effetto dannoso sia sulla memoria che sull’ottimo computato.

Ci troviamo quindi di fronte a un problema di zaino “generalizzato”: vogliamo riempire uno zaino per ogni riga e per ogni colonna scegliendo tra le celle della matrice, rispettando lo specifico vincolo di capacità, in modo da ottimizzare il premio associato, con l’ulteriore divieto di doppia ripresa.

Il problema è fortemente NP-completo, per cui tutti gli algoritmi che andremo ad implementare per risolverlo sono Euristici.

|  |
| --- |
| Premio = 153  Peso = 70 < 72 |

Matrice Premi: (pi) Zaino Riga 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 54 | 60 | 84 | 53 | 9 |
| 39 | 58 | 8 | 30 | 2 |
| 32 | 74 | 73 | 50 | 36 |
| 21 | 89 | 70 | 64 | 34 |

|  |  |  |
| --- | --- | --- |
| (1,2)  pi = 60  rho = 62 | (1,3)  pi = 80  rho = 3 | (1,5)  pi = 9  rho = 5 |

Matrice Pesi Riga: (rho)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 80 | 62 | 3 | 21 | 5 |
| 91 | 17 | 7 | 42 | 78 |
| 47 | 26 | 59 | 50 | 90 |
| 54 | 10 | 57 | 77 | 75 |

Vettore capacità riga: (r)

|  |  |  |  |
| --- | --- | --- | --- |
| 72 | 72 | 72 | 72 |

*Capitolo 2: L’Algoritmo, implementazione Top-Down*

Il problema si può decomporre nei seguenti sottoproblemi:

1. Caricare dal file di testo in input la dimensione delle matrici, le tre matrici DATA, i vettori capacità righe e colonne
2. Costruire gli zaini per le righe e per le colonne, calcolando la stima Zaino
3. Costruire la lista concatenata con tutte le celle, prese prima per righe e poi per colonne e ordinarla in base ai criteri forniti dal problema
4. Computare le matrici soluzione per le quattro Euristiche e calcolare il valore delle soluzioni associate
5. Stampare a video le soluzioni ottenute, specificando la direzione di ripresa delle celle (calcolo della stima Totale)

Per la realizzazione dei vari Algoritmi e la gestione delle strutture dati, sono presenti quattro librerie, la cui descrizione dettagliata insieme ai costi è discussa nel Capitolo 3 (ogni qual volta verrà fatto uso di una funzione presente in una libreria esterna al main, verrà fatto un riferimento specifico alla sezione da consultare). L’analisi dei costi computazionali si basa sul modello uniforme (in linea di massima sul caso pessimo), e fornisce un limite superiore della stima asintotica.

***Sezione 2.1: Caricamento dei Dati da File di Testo***

Richiamata la funzione **InterpretaLineaComando**, che legge il nome del file di testo contenente i dati dalla linea di comando e ne controlla l’effettiva riuscita, passiamo alla funzione **Parse** le tre matrici, rispettivamente dei Premi, dei PesiRiga e dei PesiColonna, i vettori capacità righe e colonne cap\_row e cap\_column, gli interi m ed n dove salvare il numero di righe e di colonne, e il file di testo da cui leggere i dati. Tutte queste strutture dati e variabili sono passate dal main alla funzione Parse “per indirizzo” (o meglio, dal momento che tale passaggio di parametri non esiste in C, per valore di indirizzo attraverso un puntatore), poiché il nostro obiettivo è di popolare, e quindi modificare, tali parametri.[[1]](#footnote-1)

Manipoliamo qui le strutture dati Matrice e Vcap, rispettivamente implementate come matrice dinamica di interi e vettore dinamico di interi, per una descrizione più dettagliata consultare il file header MatriceVettori.h.

Seguendo quanto indicato nel testo del problema, scorriamo il file una riga alla volta, servendoci della fgets per saltare le righe vuote, quelle con le indicazioni sul tipo di dato in ingresso (p.e. param pi := ) e per andare a capo, della fscanf per estrarre dal file i dati significativi:

* Letti per prima cosa gli interi m ed n, possiamo costruire le tre matrici e i due vettori con le funzioni **costruisci\_matrice** e **costruisci\_vettore**

**COSTI**  
SPAZIO: O(mxn) per le matrici, O(m) per il vettore capacità riga e O(n) per il vettore capacità colonna 🡪 *semitot = O(mxn)*

TEMPO: O(m) per le matrici, O(1) per il vettore capacità riga e per il vettore capacità colonna 🡪 *semitot = O(m)*

* Una volta inizializzate le strutture dati necessarie, passiamo al parsing del file di testo per riempirle, servendoci di cicli while sia per le matrici che per i vettori, con condizione di uscita quando la fscanf non trova più i dati formattati corretti (arriviamo al “;”)

**COSTI**

SPAZIO: O(mxn) per il while sulle matrici, O(m) per il while su vettore capacità riga e O(n) per il while su vettore capacità colonna 🡪 *semitot = O(mxn)*

TEMPO: O(mxn) per il while sulle matrici, O(m) per il while su vettore capacità riga e O(n) per il while su vettore capacità colonna 🡪 *semitot = O(mxn)*

|  |
| --- |
| **>SPAZIO: O(mxn)** |
| **>TEMPO: O(mxn)** |

**COSTO GLOBALE DI PARSE:**

***Sezione 2.2: Costruzione degli zaini e stima Zaino***

Possiamo ora passare all’allocazione e al popolamento degli zaini righe e colonne. Entrano qui in gioco le strutture dati nodo, stack e stato, implementate nella libreria stack.h.

L’idea è di vedere gli zaino\_r e zaino\_c come doppi puntatori a stati, i quali rappresentano la soluzione corrente del problema KnapSack tramite campi premio, spazio e uno zaino[[2]](#footnote-2), cioè uno stack con le celle associate alla soluzione (i nodi): per lo zaino righe avremo m stato\* con al più n nodi nello stack e per lo zaino colonne n stato\* con al più m nodi nello stack.

**COSTI**

SPAZIO: O(mxn) per lo zaino\_r, O(nxm) per lo zaino\_c 🡪 *semitot = O(mxn)*

TEMPO: Costante 🡪 *semitot = O(1)*

Una volta allocati gli zaini, possiamo popolarli[[3]](#footnote-3) richiamando la funzione **Zaino,** che somma i valori ottimi dei problemi di zaino associati a tutte le righe e poi a tutte le colonne, aggiornando passo passo con due cicli for, uno per le righe e uno per le colonne, i due zaini da popolare con lo stato corrente s. Ci serviamo qui della libreria Euristiche.h, in particolare usiamo le funzioni **KSD\_Righe** e **KSD\_Colonne,** le quali risolvono i problemi di zaino in modo esatto attraverso l’utilizzo del KnapSack Dinamico.

**COSTI**

SPAZIO: Costante per allocazione, O(max(m,n)xk) per la costruzione degli zaini con il KSD 🡪 *semitot =* O(max(m,n)xk)

TEMPO: for per le righe O(mx(*tempoKSD\_Righe)*) = O(mxnxk) e for per le colonne O(nx(*tempoKSD\_Colonne)*) = O(nxmxk) 🡪 *semitot = O(mxnxk)[[4]](#footnote-4)*

|  |
| --- |
| **>SPAZIO: O(max(m,n)xk)** |
| **>TEMPO: O(mxnxk)** |

**COSTO GLOBALE DI COSTRUZIONE + ZAINO:**

***Sezione 2.3: Lista Concatenata e Ordinamento + Libreria list.h[[5]](#footnote-5)***

Vediamo ora l’implementazione della struttura dati lista per gli algoritmi Greedy e Approx: l’idea di base è di costruire una lista con tutte le celle della matrice, srotolandola prima per righe e poi per colonne. Una volta costruita, useremo l’algoritmo *QuickSort* per ordinarla, secondo le disposizioni del problema, dall’elemento massimo. In questo modo potremo scorrere la lista e valutare l’eventuale selezione della cella corrente per lo zaino.

Gli elementi della lista, che viene qui implementata come unidirezionale, con radice tramite l’utilizzo di puntatori, sono appunto le celle della matrice, con campi informazione

* indice di riga e di colonna per tracciarne le posizioni
* il ratio, i.e. il rapporto tra premio e peso, che sarà il nostro criterio di ordinamento primario
* la direzione, enum definito nella libreria MatriceVettori, che indica il verso di ripresa della cella
* un puntatore alla cella successiva

Per la fase di costruzione della lista concatenata ci serviamo delle cinque funzioni di libreria

1. **lista\_concatenata**, che scorre prima per righe e poi per colonne le matrici con due cicli for, che richiamano a ogni passo la funzione costuisci\_lista\_celle
2. **costuisci\_lista\_celle**, che restituisce la lista costituita dalla riga/colonna corrispondente, richiamando a sua volta la funzione costruisci\_cella
3. **costruisci\_cella**, che inizializza una nuova cella con i campi informazione necessari, facendo anche un controllo sull’allocazione con **check\_allocazione\_cella**
4. **ultima\_cella**, che scorre la lista e restituisce l’elemento in coda, usata in questa fase per appendere in lista\_concatenata

**COSTI**

SPAZIO: Costante per *check\_allocazione\_cella* O(1), Costante per *costuisci\_cella* O(1), Lineare per *costuisci\_lista\_celle* O(max(m,n)), O(mxnx2) = O(mxn) per *lista\_concatenata,* Costante per *ultima\_cella* O(1) 🡪 *semitot = O(mxn)*

TEMPO: Costante per *check\_allocazione\_cella* O(1), Costante per *costuisci\_cella* O(1), Lineare per *costuisci\_lista\_celle* O(max(m,n)), O(mxnx2) = O(mxn) per *lista\_concatenata,*  O(mxn) per *ultima\_cella* 🡪 *semitot = O(mxn)*

Temp

Testa

succ

Cella

Lista

(i,j)

ratio

dir

Per la fase di Ordinamento usiamo l’Algoritmo *QuickSort:* questo Algoritmo di Sorting ricorsivo è già stato ampiamente discusso a lezione, ci limitiamo quindi a sottolineare che stiamo ordinando in modo decrescente secondo quattro criteri:

1. Ratio
2. A parità di ratio, indice di riga maggiore
3. A parità di indice di riga, indice di colonna maggiore
4. A parità di cella, per direzione Discendente

**COSTI**

SPAZIO: Costante per *ultima\_cella* e per *swap*, O(mxn) per *partition* e per *quick\_sort*🡪 *semitot = O(mxn)*

TEMPO: Costante per *swap*, O(mxn) per *ultima\_cella* e *partition,* O((m^2)x(n^2)) per *quick\_sort* 🡪 *semitot = O((m^2)x(n^2))[[6]](#footnote-6)*

|  |
| --- |
| **>SPAZIO: O(mxn)** |
| **>TEMPO**: **O((m^2)x(n^2))** |

**COSTO GLOBALE PER LISTA E ORDINAMENTO:**

***Sezione 2.4: Euristiche e Stime***

Ora abbiamo tutto il necessario per implementare gli algoritmi Euristici, i quali restituiscono come soluzione una matrice di marcatura, che indica se la cella è stata aggiunta alla soluzione dall’euristica considerata e in caso positivo indica la direzione di ripresa.

***2.4.1) Algoritmo Greedy***

Iniziamo costruendo la matrice soluzione e i vettori copia delle capacità[[7]](#footnote-7).

Abbiamo già la lista ordinata, per cui quello che basta fare è scorrerla, e per ogni cella valutare se rispetta o meno i vincoli di capacità e di ripresa: in caso positivo l’aggiungiamo alla soluzione marcando MatSol con la direzione e decrementiamo il valore della capacità della riga/colonna di cui la cella fa parte.

Una volta controllate tutte le celle della lista, distuggiamo i vettori ausiliari: a questo punto, infatti, tutte le decisioni possibili o sono parte della soluzione o sono inammissibili, in quanto violano qualche vincolo.

Vedi libreria MatriceVettori per implementazione di **costruisci\_vettore**, **costruisci\_matrice**, **copia\_vettore** e **distruggi\_vettore**.

**COSTI GREEDY**

SPAZIO: Costruzione O(mxn) [O(mxn) per MatSol, O(m) per cap\_r, O(n) per cap\_c], copia Lineare O(m)/O(n), distruzione O(1) 🡪 *semitot = O(mxn)*

TEMPO: Costruzione O(m) [O(m) per MatSol, O(1) per cap\_r e per cap\_c], copia Lineare O(m)/O(n), distruzione O(1), for sulla lista O(mxnx2) = O(mxn) 🡪 *semitot = O(mxn)*

***2.4.2) Algoritmo DiscAsc & AscDisc***

Analizziamo soltanto l’algoritmo DiscAsc, dal momento che AscDisc è del tutto analogo, con la sola differenza che risolve prima per colonne e poi per righe.

Dopo aver costruito la matrice delle soluzioni, procediamo con le due fasi:

* *Disc:* Abbiamo già risolto il problema dello zaino per righe quando abbiamo costruito gli zaini, perciò ci basta scorrere lo zaino\_righe e salvare i risultati nelle matrice soluzione con l’aiuto della funzione **aggiorna\_soluzione**, contenuta nel file header Euristiche.h
* *Asc:* è ora necessario risolve il KP per colonne, tenendo conto di quanto computato nella fase Disc, calcoliamo lo stato corrente con **KSD\_Colonne** e aggiorniamo la soluzione, ricordandoci di liberare lo stato ad ogni iterazione del for sulle colonne con la funzione **free\_stato** (stack.h).

Rimando alla sezione 3.3 per l’implementazione dell’algoritmo iterativo KP dinamico tramite tecnica bottom-up.

**COSTI DISCASC & ASCDISC**

SPAZIO: O(mxn) per MatSol🡪 *semitot = O*(mxn*)*

TEMPO: O(m) per MatSol, for sulle righe O(mx*tempoaggiornasoluzione*) = O(mxn), for sulle colonne O(mxnxk)🡪 *semitot = O*(mxnxk*)*

***2.4.3) Algoritmo Approx***

L’ultima euristica, Approx, risolve indipendentemente tra loro tutti i problemi di zaino associati alle righe e alle colonne, sfruttando quanto già computato negli zaini e utilizzando la funzione **aggiorna\_soluzione** per marcare le due matrici MatSol1 e MatSol2, rispettivamente matrici soluzione per righe e per colonne. Una volta inizializzate in questo modo le due matrici, le scorre, e per tutte le celle marcate in entrambe le matrici, sceglie quella con peso minore (in caso di parità quella Discendente) e salva la scelta in MatSol1. Le celle non marcate in Matsol1 ma marcate in Matsol2, vengono quindi aggiunte alla soluzione MatSol1 in direzione Ascendente.

Questo procedimento fa sì che possa essersi creato dello spazio libero in memoria: passiamo quindi alla ricerca di eventuali celle non ancora aggiunte alla soluzione e che rispettino i vincoli, da poter inserire, scorrendo man mano la lista ordinata. Usiamo in questa fase la funzione della libreria Euristiche.h **residuo.**

**COSTI APPROX**

SPAZIO: O(mxn) per MatSol1 e MatSol2 🡪 *semitot = O*(mxn*)*

TEMPO: O(m) per MatSol1 e MatSol2, for sulle righe O(mx*tempoaggiornasoluzione*) = O(mxn), for sulle colonne O(nx*tempoaggiornasoluzione*) = O(nxm), for sulle matrici O(max(m,n)^2xmin(m,n)), for sulla lista O(mxnx2)🡪*semitot = O(max(m,n)^2xmin(m,n))*

Poiché l’algoritmo risolve tutti i problemi di zaino all’ottimo nella prima fase e poi nella seconda ne controlla l’ammissibilità, il valore della soluzione finale sarà almeno pari a metà di un valore non inferiore all’ottimo: l’algoritmo si dice quindi 2-approssimato.

***2.4.4) Stime Euristiche***

Una volta computate le matrici soluzione, non ci manca che calcolare le stime per le quattro euristiche: scorriamo le matrici soluzioni e per ogni cella ripresa, ne sommiamo il premio al totale. Questo procedimento viene eseguito dalla funzione **Stima\_Euristica.**

**COSTI STIME**

SPAZIO: Costante 🡪 *semitot = O(1)*

TEMPO: O(mxn) per il doppio ciclo annidato for 🡪 *semitot = O(mxn)*

|  |
| --- |
| **>SPAZIO: O(mxn)** |
| **>TEMPO**: **O(mxnxk)** |

**COSTO GLOBALE PER EURISTICHE:**

***Sezione 2.5: Stampe***

Liberate le strutture dati ormai inutili, stampiamo a video i risultati nel formato richiesto dal problema, utilizzando la funzione **stampa\_soluzioni** (MatriciVettori.h). Richiamiamo in questa fase la funzione **Totale**, per calcolare la somma di tutti i premi delle celle.

Concludiamo con la deallocazione delle restanti strutture.

**COSTI DEALLOCAZIONI E STAMPE**

SPAZIO: Costante 🡪 *semitot = O(1)*

TEMPO: O(mxn) per le deallocazioni, per *Totale* e per le stampe 🡪 *semitot = O(mxn)*

|  |
| --- |
| **>SPAZIO: O(1)** |
| **>TEMPO**: **O(mxn)** |

**COSTO STAMPE:**

*Capitolo 3: Librerie*

***Sezione 3.1: MatriceVettori.h***

Questa libreria gestisce le strutture dati Matrice e Vcap, che sono rispettivamente una matrice dinamica di interi e un vettore dinamico di interi. Definiamo nell’header l’enum *Direzione* per indicare il verso di ripresa delle celle, e simuliamo il tipo booleano.

Sono qui implementate le funzioni:

* **Costruisci\_matrice,** che ha un costo spaziale O(mxn) e temporale O(m)
* **Distruggi\_matrice,** con costo temporale O(m)
* **Stampa\_soluzioni,** che scorre la matrice soluzione e ne stampa le celle marcate con la direzione di ripresa. Costo temporale O(mxn)
* **Costruisci\_vettore,** che ha un costo spaziale O(m)/O(n) a seconda della dimensione di input e temporale O(1)
* **Copia\_vettore,** che copia il Vcap origine in quello destinazione, con costo spaziale O(1) e temporale lineare O(n)
* **Distruggi\_vettore,** con costo spaziale e temporale costanti
* **Check\_allocazione\_matrice e check\_allocazione\_vettore,** entrambi costanti sia nel tempo che nello spazio

***Sezione 3.2: stack.h***

Questa libreria implementa uno stack dinamico (LIFO) tramite linked list, struttura dati usata per il KSDinamico.

Implementiamo lo zaino con le celle soluzione come uno stack, in cui appendiamo ed eliminiamo in testa. Ogni nodo dello stack ha campi informazione che ne identificano la posizione nella matrice (i,j) e un puntatore al nodo successivo.

top

Stato

|  |
| --- |
| (i,j) |
|  |
|  |
|  |
|  |
|  |

nodo

Zaino (stack)

Spazio

Premio

Zaino

Ogni stato dello zaino (come stato\*\*) contiene il premio e lo spazio

Occupato dalla soluzione corrente, e uno zaino (come stack) con le

Celle selezionate dalla stessa.

NULL

Sono presenti le classiche funzioni per uno stack

* **isEmpty**, per controllare se lo stack è vuoto
* **push,** per il popolamento dello stack, tempo costante
* **pop**, per liberare lo stack eliminando in testa, tempo costante
* **free\_stato,** che distrugge lo stato e lo stack associato in tempo lineare
* **check\_allocazione\_nodo e check\_allocazione\_stato**

***Sezione 3.3: Euristiche.h***

Questa libreria implementa una serie di funzioni ausiliarie per gli algoritmi euristici computati.

Di particolare rilevanza è l’algoritmo **KnapSackDinamico**, che risolve il problema di zaino esattamente per la riga/colonna selezionata. Ecco che ci torna utile l’allocazione di matrici (m+1)x(n+1) con la calloc: se per le matrici finora utilizzate poteva essere in linea con l’indicizzazione “naturale” delle stesse, ora avere a che fare con matrici con una riga e una colonna in più risulta necessario, dal momento che la prima riga corrisponderà all’ottimo al tempo 0, e la prima colonna corrisponderà all’ottimo con capacità 0.

L’idea è infatti quella di costruire una matrice (knapsack) con n/m righe a seconda che si stia considerando una riga o una colonna della matrice originale (se valuto una riga allora ho m elementi e viceversa), e con k colonne, dove k è la capacità della riga/colonna selezionata.

Procediamo quindi per righe:

* per prima cosa copiamo l’ottimo alla riga precedente e alla stessa colonna
* se l’elemento di riga ha una capacità non superiore alla capacità della colonna in esame, allora controlliamo che non sia già stato inserito nella soluzione e che migliori effettivamente l’ottimo
* aggiorno l’ottimo con la scelta migliore: computo il massimo tra quanto ottenuto al passo precedente & la somma del premio dato dall’elemento e la soluzione alla riga precedente e colonna decrementata del peso dell’elemento (scelgo se la soluzione è migliore al passo precedente o se invece l’elemento migliorerebbe la soluzione).
* una volta costruita la matrice knapsack, ricostruiamo le soluzioni ripercorrendo all’indietro la matrice per righe, partendo dall’ultima entrata di knapsack:

1. Se nella riga precedente il valore è lo stesso, allora l’elemento della riga considerata non migliora l’ottimo e quindi non è selezionato dall’algoritmo, si prosegue quindi iterativamente con la cella alla riga precedente e alla stessa colonna
2. Se invece il valore alla riga precedente è diverso (e per costruzione sarà minore), allora l’elemento fa parte della soluzione e aggiorno quindi l’ottimo con push, scorro l’indice di colonna all’indietro della quantità peso(elemento) e itero il procedimento alla riga precedente

**COSTI KSDinamico**

SPAZIO: Matrice knapsack costruzione O(mxk)/O(nxk) 🡪 *semitot = O(max(m,n)xk)*

TEMPO: O(mxn) per le costruzione, O(m) per distruzione, for annidato O(max(m,n)xk), for per ricostruzione della soluzione O(max(m,n)xk) 🡪 *semitot = O(max(m,n)xk)*

|  |
| --- |
| **>SPAZIO: O(max(m,n)xk)** |
| **>TEMPO**: **O(max(m,n)xk)** |

**COSTO KSDinamico:**

**KSD\_Righe/Colonne** semplicemente richiamano il KnapSackDinamico per la risoluzione del problema di zaino per righe e per colonne rispettivamente. I costi sono quindi gli stessi.

**Aggiorna\_soluzione**, prende lo stato attuale ks, e marca la matrice soluzione con le celle riprese nel campo zaino di ks con la direzione *visita*. Costo spaziale costante e temporale O(mxn).

**Residuo**, controlla che la cella rispetti i vincoli del problema: se la visita è Ascendente, allora si scorre la colonna index-esima da cui la cella è stata selezionata e si sommano i pesi delle celle riprese (analogamente se la visita è Discendente si scorre la riga index-esima). Se la cella non sfora il vincolo di capacità una volta aggiunta alla soluzione, allora lo comunichiamo alla funzione chiamante (nel nostro caso ad Approx). Costo spaziale costante e temporale lineare.

Esempio implementazione KSDinamico: k = 4 elementi {1,2,3} con Premi {1,2,4} e Pesi{2,3,1}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| / | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0->1 | 0->1 | 0->1 |
| 2 | 0 | 0 | 1 | 1->2 | 1->2 |
| 3 | 0 | 0->4 | 1->4 | 2->5 | 2->6 |

Ottimo = {2,3} Premio = 6

*Capitolo 4: Costo Globale e Conclusioni*

**SPAZIO:** O(Mxk) dove M = max(m,n) e k = max (capacità)

**TEMPO:** Nel caso medio O(mxnxk) = O(k^3) essendo k>>, nel caso peggiore O(m^2xn^2) = O(M^4) [quicksort]

Nonostante tutti gli algoritmi implementati siano non polinomiali, il codice implementato ha una buona resa temporale su tutti i file in ingresso: i test condotti stimano tempistiche massime di 52 sec cc per il file più grande esempio06.txt (escluse le stampe).

1. In realtà, il codice sfrutta questo tipo di passaggio di parametri per quasi tutte le funzioni implementate, fatte poche eccezioni, anche quando la modifica del parametro non avviene: questo per ottimizzare la gestione della memoria (così facendo non copiamo ogni volta i parametri). [↑](#footnote-ref-1)
2. N.B. Attenzione a non confondere gli zaini campo degli stati, che sono stack con le soluzioni, e gli zaini righe e colonne, che sono invece stato\*\*. Questa scelta potrebbe essere lessicalmente un po’ ambigua, ma trovo sia giustificata dal fatto che ci troviamo di fronte a dover risolvere dei problemi di zaino, per uno zaino “più grande” dato da tutte le righe o, rispettivamente, tutte le colonne. [↑](#footnote-ref-2)
3. Non è qui necessario passare gli zaini tramite puntatore, poiché li abbiamo già definiti come doppi pointer. [↑](#footnote-ref-3)
4. k = massimo tra la capacità righe e colonne. [↑](#footnote-ref-4)
5. Contrariamente a quanto anticipato, mi sembra qui più ragionevole descrivere tutta la libreria dedicata alla lista, dal momento che tutte le funzioni (a parte distuggi\_lista) vengono richiamate per la costruzione e l’ordinamento della stessa. [↑](#footnote-ref-5)
6. Usiamo QuickSort perché nel caso medio ha un costo di O(mxnxlog(mxn)) e nel caso migliore O(mxn) [↑](#footnote-ref-6)
7. Risulta qui necessario copiarli poiché andremo a modificarli nel corpo dell’algoritmo [↑](#footnote-ref-7)