

UNIVERSITÀ DEGLI STUDI DI MILANO
POLO DIDATTICO E DI RICERCA DI CREMA

Laurea triennale in Informatica



PROBLEMA DELLO ZAINO
DI VALORE MASSIMO

Progetto di

ALGORITMI E STRUTTURE DATI

Studente:

Lena Cota Guido

Raimondi Dario

713957

706021

Docente del corso:

Roberto Aringhieri

Anno Accademico 2006/2007

Indice

Cenni teorici.....	3
Scopo del progetto.....	4
Algoritmo utilizzato.....	6
Scelte implementative.....	7
Test di correttezza.....	9
Test di efficienza.....	12

Cenni teorici

Le classi di problemi computazionali risolvibili con un algoritmo possono essere ridotte a tre: decisionali, di ricerca, di ottimizzazione. In particolare, nei problemi di ottimizzazione è associata ad ogni soluzione ammissibile una misura (detta costo), e si vuole trovare una soluzione ottima, sottoinsieme degli elementi del problema iniziale, che minimizzi (o massimizzi) tale misura.

Tra le tecniche di progetto più consolidate, quella *greedy* è particolarmente indicata per i problemi di ottimizzazione. Essa si basa sulla cosiddetta strategia dell'ingordo, ovvero costruire la soluzione effettuando, ad ogni passo, la scelta migliore nell'immediato piuttosto che adottare una strategia a lungo termine.

Operativamente, tale metodo si compone di due parti: l'ordinamento degli elementi del problema in base a un *criterio di ottimalità*, e la *costruzione della soluzione* in modo incrementale, considerando gli oggetti uno alla volta e inserendo (se possibile) quello che genera il risultato migliore. Da notare come ad ogni scelta il problema si riduca ad un sottoproblema dello stesso tipo, ma di dimensione più piccola, e come ogni scelta non venga mai rimessa in discussione (cosa che accade invece nella tecnica *backtrack*). Per quanto riguarda il criterio di ottimalità, esso consiste nell'associare un certo valore ai dati in ingresso che indichi la bontà della loro scelta, e quindi ordinarli in base ad esso.

I vantaggi della strategia greedy sono la possibilità di ottenere algoritmi semplici, intuitivi e facilmente implementabili, caratterizzati da un ridotto tempo di calcolo per la determinazione della soluzione. Per contro hanno invece il fatto che non sempre forniscono una soluzione ottima, pur trattandosi comunque di una buona soluzione di partenza per algoritmi più sofisticati. Ad esempio, i risultati di un algoritmo greedy possono essere migliorati con un algoritmo di *ricerca locale*, basato sulla ricerca di una soluzione in qualche modo “vicina” a quella iniziale, e che la migliori.

Scopo del progetto

Scopo del progetto sarà realizzare un algoritmo per la soluzione del problema dello *zaino di valore massimo*, formulato come segue:

Dati n oggetti di valore v_i e peso p_i con $i = 1, \dots, n$ ed uno zaino di capacità C , determinare un sottoinsieme di oggetti $Z \subseteq \{1, \dots, n\}$ tali che il peso totale degli oggetti in Z non ecceda la capacità C ($\sum_{i \in Z} p_i \leq C$) e la somma dei valori degli oggetti contenuti in Z sia massima ($z = \max \sum_{i \in Z} v_i$).

Il programma dovrà ricevere in input un'istanza I del problema e dovrà restituire il valore totale degli oggetti messi nello zaino, la capacità residua dello stesso, il tempo di calcolo e (se richiesto) gli oggetti aggiunti in soluzione.

Per istanze si intendono file con il seguente formato:

```
<numero di oggetti>
1 <valore[1]> <peso[1]>
2 <valore[2]> <peso[2]>
...
n <valore[n]> <peso[n]>
<capacità zaino>
```

I dati sono raccolti in tre archivi di file .dat :

- *kp-strongcorrelated.zip* Contiene 10 istanze, ognuna con 5000 oggetti. valore[i] e peso[i] sono tali che ciascun peso[i] è generato indipendentemente da distribuzione uniforme [1..10000], ed il corrispondente valore[i] è generato in un intorno ristretto del corrispondente peso[i].
- *kp-uncorrelated.zip* Contiene 10 istanze, ognuna con 5000 oggetti. valore[i] e peso[i] sono generati indipendentemente da distribuzioni uniformi in [1..10000].
- *kp-test.zip* Contiene due istanze di test composte da 20 oggetti di tipo *kp-uncorrelated*.

Implementato l'algoritmo, andrà infine testato secondo due criteri:

- di *correttezza*, da effettuare a mano, verificando che la soluzione ottenuta coincida con quella generata dal programma;
- di *efficienza*, che introdotto il concetto di gap come differenza tra la soluzione ottima e quella ottenuta fratto la soluzione ottima stessa, controlla che il suo valore non sia superiore al 10%. I valori ottimi sono riportati di seguito:

kp-astrongcorrelated.zip									
01	02	03	04	05	06	07	08	09	10
4048100	2355745	1736520	1407960	1199927	1055047	947010	863587	796104	741339

kp-uncorrelated.zip									
01	02	03	04	05	06	07	08	09	10
9025522	6394075	5206791	4506324	4029233	3677243	3402355	3178744	2994777	2838214

Il linguaggio di programmazione previsto per la realizzazione dell'algoritmo è C.

L'ambiente di sviluppo è *Dev-C++* (versione 4.9.9.2) della *Bloodshed Software*.

<http://www.bloodshed.net/dev/>

Algoritmo utilizzato

Il problema dello zaino di valore massimo è un classico esempio di problema di ottimizzazione, risolvibile in modo semplice con un algoritmo di tipo greedy.

Possiamo applicare diversi criteri di ottimalità, a seconda che si voglia dare maggior rilevanza al valore degli oggetti, al loro peso, o a entrambe le cose. Nel primo e secondo caso cercheremo di introdurre in soluzione, ad ogni passo dell'algoritmo, gli oggetti rispettivamente di valore o di peso massimo. Se invece volessimo considerare entrambe le proprietà, potremo fare un rapporto tra le due grandezze e inserire in soluzione gli oggetti con rapporto maggiore.

Indipendentemente dal criterio scelto, si ordinerebbero in base ad esso tutti gli oggetti in modo decrescente.

Per quanto riguarda la costruzione della soluzione, l'unico controllo per ogni elemento selezionato è che il suo peso non superi la capacità residua dello zaino. In caso contrario, l'oggetto non può essere aggiunto in soluzione.

Considerando che un algoritmo greedy non produce necessariamente soluzioni ottime, abbiamo deciso di applicare in modo facoltativo un algoritmo di ricerca locale alla soluzione ricavata al fine di migliorarla.

Scelte implementative

Gli oggetti sono modellati nel codice come strutture dati, caratterizzati da un id identificativo, il valore e il peso dell'oggetto, e il rapporto tra le due grandezze.

Anche lo zaino è una struttura dati, composto da due array dinamici (uno per segnalare se l'oggetto i -esimo è stato aggiunto in soluzione, uno per tener traccia degli id degli oggetti inseriti nello zaino), dal numero degli oggetti dati in ingresso e da quelli inseriti in soluzione, dalla capacità totale dello zaino, e infine dal peso e dal valore finale.

L'ordinamento degli oggetti avviene con un algoritmo di quicksort, implementato nella funzione `qsort()` inclusa nella libreria standard `stdlib.h`, mentre la misurazione del tempo di calcolo è resa possibile dalla funzione `clock()` contenuta nella libreria `time.h`.

Per ottenere in ogni istanza del problema la soluzione che massimizza il valore, si è scelto di applicare per ognuna di esse tutti i criteri di ottimalità illustrati nella pagina precedente, scegliendo poi quello che produce la soluzione migliore. Viene inoltre fornita all'utente la possibilità di migliorare ulteriormente la soluzione ottenuta richiamando un algoritmo di ricerca locale sulla soluzione dell'algoritmo greedy.

La funzione `main` si aspetta come parametro il nome del file di testo che contiene la lista dei nomi dei file con le istanze del problema ed eventuali valori ottimi noti, secondo questo formato:

```
<nome file .dat> <valore ottimo (facoltativo)>  
<nome file .dat> <valore ottimo (facoltativo)>  
...
```

Il programma risolve in sequenza i vari problemi, calcolando il gap solo nel caso in cui sia indicato il valore ottimo delle istanze.

Se il nome della lista dei file è un parametro fondamentale per la corretta esecuzione della funzione *main*, ve ne sono altri tre facoltativi ma altrettanto utili:

- *--debug* , che visualizza a schermo le informazioni parziali di ogni computazione
- *--rl* , che effettua una ricerca locale sulla soluzione dell'algoritmo greedy
- *--vis* , che visualizza a schermo l'id degli elementi inseriti nello zaino.

Ad esempio, passare alla *main* i parametri:

```
listaFile.txt --rl -vis
```

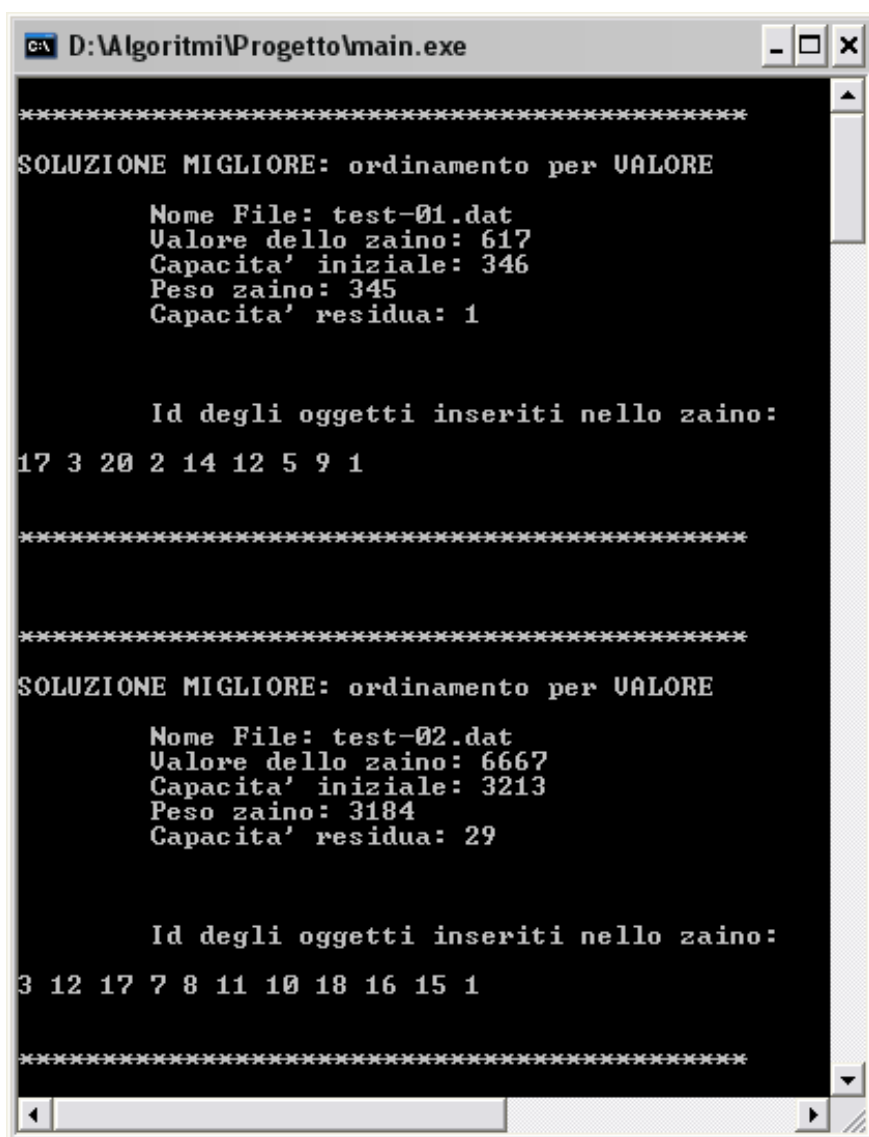
significa calcolare gli zaini massimi dei file *.dat* elencati nel file di testo *listaFile.txt*, applicare un algoritmo di ricerca locale sulla soluzione e visualizzare gli id degli oggetti inseriti (altrimenti non visibili). L'ordine con cui sono passati i tre parametri facoltativi del *main* non è importante.

La nostra implementazione di ricerca locale prevede la simulazione di uno scambio di un elemento già presente in soluzione con uno fuori, col conseguente controllo di un effettivo miglioramento a livello globale. Terminata una fase in cui vengono analizzati tutti gli scambi possibili, viene scelto tra questi quello che apporta un vantaggio maggiore. L'algoritmo viene ripetuto finché vengono rilevate migliorie, in caso contrario termina.

Test di correttezza

Il test di correttezza consiste nell'eseguire il programma sulle due istanze di test *test-01.dat* e *test-02.dat* e verificare a mano che i risultati ottenuti siano corretti.

Output



```
C:\ D:\Algoritmi\Progetto\main.exe

*****
SOLUZIONE MIGLIORE: ordinamento per VALORE

Nome File: test-01.dat
Valore dello zaino: 617
Capacita' iniziale: 346
Peso zaino: 345
Capacita' residua: 1

Id degli oggetti inseriti nello zaino:
17 3 20 2 14 12 5 9 1

*****

*****
SOLUZIONE MIGLIORE: ordinamento per VALORE

Nome File: test-02.dat
Valore dello zaino: 6667
Capacita' iniziale: 3213
Peso zaino: 3184
Capacita' residua: 29

Id degli oggetti inseriti nello zaino:
3 12 17 7 8 11 10 18 16 15 1

*****
```

Di seguito, una su tutte, la verifica del *test-01.dat* .

Ordiniamo innanzitutto gli oggetti in base al loro valore

ID Oggetto	VALORE	PESO
17	98	5
3	97	67
20	91	39
2	87	96
14	76	76
11	75	89
12	70	25
16	61	56
10	54	63
5	47	13
19	39	45
9	32	23
15	31	23
7	30	22
18	27	47
4	22	90
6	22	74
1	19	1
13	7	100
8	5	86

Eseguiamo quindi passo passo la procedura di riempimento dello zaino:

- inserisco nello zaino l'elemento **17**

VALORE ZAINO: **98**

CAPACITA' RESIDUA: **346 - 5 = 341**

- inserisco nello zaino l'elemento **3**

VALORE ZAINO: **98 + 97 = 195**

CAPACITA' RESIDUA: **274**

- inserisco nello zaino l'elemento **20**

VALORE ZAINO: **286**

CAPACITA' RESIDUA: **235**

- inserisco nello zaino l'elemento **2**

VALORE ZAINO: **373**

CAPACITA' RESIDUA: **139**

- inserisco nello zaino l'elemento **14**
VALORE ZAINO: **449** CAPACITA' RESIDUA: **63**
- non inserisco l'oggetto **11** perché il suo peso supera la capacità residua, quindi passo ad inserire il **12**
VALORE ZAINO: **519** CAPACITA' RESIDUA: **38**
- non inserisco gli oggetti **16** e **10** perché il loro peso supera la capacità residua, quindi passo ad inserire il **5**
VALORE ZAINO: **566** CAPACITA' RESIDUA: **25**
- non inserisco l'oggetto **19** perché il suo peso supera la capacità residua, quindi passo ad inserire il **9**
VALORE ZAINO: **598** CAPACITA' RESIDUA: **2**
- non inserisco gli oggetti **15, 7, 18, 4, 6** perché il loro peso supera la capacità residua, quindi passo ad inserire l'**1**
VALORE ZAINO: **617** CAPACITA' RESIDUA: **1**
- non inserisco gli oggetti **13** e **8** perché il loro peso supera la capacità residua

VALORE FINALE ZAINO: 617 **CAPACITA' RESIDUA: 1**

I risultati corrispondono a quelli computati dal programma.

Il test di correttezza della seconda istanza di test del problema avviene con le stesse procedure sopra illustrate (che non riportiamo per brevità).

Test di efficienza

Il test di efficienza consiste nel confrontare i risultati ottenuti con quelli ottimi dati dal problema, calcolandone il gap in termini percentuali e verificando che questo non superi il 10%. Il gap è calcolato come:

$$\frac{(\text{soluzione ottima}) - (\text{soluzione ottenuta})}{(\text{soluzione ottima})}$$

Le soluzioni ottime sono quelle riportate nel capitolo *Scopo del progetto*, e vengono passate al programma inserendole nella lista delle istanze del problema da risolvere, subito dopo il loro nome. Ad esempio:

```

astrongcorrelated01.dat 4048100
astrongcorrelated02.dat 2355745
uncorrelated01.dat 9025522
...

```

Di seguito i risultati ottenuti:

NOME FILE	VALORE OTTIMO	VALORE OTTENUTO	GAP (%)
astrongcorrelated01.dat	4048100	4046568	0,038
astrongcorrelated02.dat	2355745	2354597	0,049
astrongcorrelated03.dat	1736520	1735736	0,045
astrongcorrelated04.dat	1407960	1407034	0,066
astrongcorrelated05.dat	1199927	1199006	0,077
astrongcorrelated06.dat	1055047	1054866	0,017
astrongcorrelated07.dat	947010	945982	0,109
astrongcorrelated08.dat	863587	862772	0,094
astrongcorrelated09.dat	796104	795104	0,126
astrongcorrelated10.dat	741339	741092	0,033

NOME FILE	VALORE OTTIMO	VALORE OTTENUTO	GAP (%)
uncorrelated01.dat	9025522	9025040	0,005
uncorrelated02.dat	6394075	6394075	0,000
uncorrelated03.dat	5206791	5206559	0,004
uncorrelated04.dat	4506324	4506031	0,007
uncorrelated05.dat	4029233	4029139	0,002
uncorrelated06.dat	3677243	3677203	0,001
uncorrelated07.dat	3402355	3402174	0,005
uncorrelated08.dat	3178744	3178476	0,008
uncorrelated09.dat	2994777	2994701	0,003
uncorrelated10.dat	2838214	2838209	0,001

Si può notare come il gap sia sempre abbondantemente sotto il limite del 10%, in particolar modo nelle istanze di tipo *uncorrelated*.

Applicando l'algoritmo di ricerca locale otteniamo evidenti miglioramenti soprattutto nelle istanze di tipo *strong correlated*, mentre negli *uncorrelated* solo in casi sporadici.

NOME FILE	VALORE OTTIMO	VALORE RICERCA LOCALE	GAP (%)
astrongcorrelated01.dat	4048100	4047919	0,004
astrongcorrelated02.dat	2355745	2355569	0,007
astrongcorrelated03.dat	1736520	1736411	0,006
astrongcorrelated04.dat	1407960	1407842	0,008
astrongcorrelated05.dat	1199927	1199800	0,011
astrongcorrelated06.dat	1055047	1055000	0,004
astrongcorrelated07.dat	947010	946868	0,015
astrongcorrelated08.dat	863587	863455	0,015
astrongcorrelated09.dat	796104	795948	0,020
astrongcorrelated10.dat	741339	741998	0,006

NOME FILE	VALORE OTTIMO	VALORE RICERCA LOCALE	GAP (%)
uncorrelated01.dat	9025522	9025199	0,004
uncorrelated02.dat	6394075	6394075	0,000
uncorrelated03.dat	5206791	5206573	0,004
uncorrelated04.dat	4506324	4506031	0,007
uncorrelated05.dat	4029233	4029139	0,002
uncorrelated06.dat	3677243	3677203	0,001
uncorrelated07.dat	3402355	3402174	0,005
uncorrelated08.dat	3178744	3178476	0,008
uncorrelated09.dat	2994777	2994701	0,003
uncorrelated10.dat	2838214	2838209	0,001

Possiamo dunque affermare che il programma supera il test di efficienza.