

The Knapsack Problem

Davide Pedone - Matr. 809468

26 giugno 2013

Indice

1	Il problema dello zaino	5
1.1	Descrizione del problema	5
1.2	Algoritmo adottato	5
2	Linguaggi di programmazione utilizzati	7
2.1	Programmazione funzionale: Scheme	7
2.1.1	Considerazioni	7
2.2	Programmazione multi-paradigma: Python	8
2.2.1	Considerazioni	10
2.3	Programmazione logica: Prolog	10
2.3.1	Considerazioni	10
2.3.2	Conclusioni	11

Capitolo 1

Il problema dello zaino

1.1 Descrizione del problema

Il problema dello zaino è un problema di ottimizzazione combinatoria: dato un insieme di oggetti, ognuno dei quali avente un peso ed un valore, occorre determinare quanti e quali oggetti inserire nello zaino in modo che la somma dei pesi non superi la capacità dello zaino e che la somma dei valori sia la massima possibile.

1.2 Algoritmo adottato

Per lo svolgimento del progetto si è preso in considerazione il problema dello zaino 0-1; esso prevede che ogni oggetto debba essere inserito intero nello zaino, al più una volta, oppure scartato. Non sono pertanto ammessi frazionamenti o ripetizioni.

L'algoritmo risolutivo scelto, presente in letteratura, sfrutta le tecniche di programmazione dinamica.

Indicando con $n = |I|$ il numero di oggetti, e supponendo che l'insieme I possa essere rappresentato come $I = 1, 2, \dots, n$, f_{kc} rappresenta il valore della miglior combinazione di oggetti tale che:

- include solo oggetti di indice compreso tra 1 e k ;
- ha somma dei pesi pari a c .

Il valore $\max_{0 \leq c \leq C} \{f_{nc}\}$ corrisponde alla soluzione ottima del problema perchè può includere qualsiasi oggetto ed ammette una somma dei pesi qualsiasi che

non ecceda la capacità dello zaino. Per ogni $k \in 0 \dots n$ e per i valori di capacità utilizzata $c \in 0 \dots C$, i valori di f_{kc} possono essere calcolati in modo ricorsivo come segue:

$$f_{kc} = \begin{cases} 0 & \text{se } k \leq 0 \\ f_{k-1,c} & \text{se } k > 0 \text{ e } w_k > c \\ \max\{f_{k-1,c}, p_k + f_{k-1,c-w_k}\} & \text{altrimenti.} \end{cases}$$

Nel caso si scelga di utilizzare la ricorsione diretta il costo in tempo è esponenziale, mentre optando per l'utilizzo di tecniche di memoization tutti i valori f_{kc} possono essere calcolati in tempo pseudo-lineare $O(nC)$.

Capitolo 2

Linguaggi di programmazione utilizzati

2.1 Programmazione funzionale: Scheme

Scheme è un linguaggio di programmazione funzionale sviluppato negli anni settanta.

Per implementare l'algoritmo in questo linguaggio si è scelto di utilizzare la ricorsione semplice, ben supportata ed agevolata dai linguaggi funzionali. Per prima cosa sono state definite due liste (w e p), contenenti i pesi ed i valori degli oggetti da inserire nello zaino, ed una variabile C il cui valore rappresenta la capacità dello zaino.

Si è passati poi a definire una funzione *statement* che data in ingresso la coppia ordinata di valori (k, c) applica l'algoritmo descritto in 1.2.

Infine si è definita la funzione *knapsack* che per, ogni valore di k , tale che $0 \leq k \leq \text{length}(w)$, e per i valori di c , tale che $0 \leq c \leq C$ calcola il valore di *statement* (k, c) e lo mostra a video.

Si ottiene così una tabella di k righe per C colonne contenente la soluzione al problema.

2.1.1 Considerazioni

Inizialmente l'approccio a questo linguaggio si è rivelato ostico, in quanto si è erroneamente cercato di riscrivere il codice Python nel linguaggio Scheme. Questo tentativo oltre ad aver portato a perdere di vista la caratteristica funzionale del linguaggio ha introdotto numerose complicazioni e creato una

situazione di stallo.

Si è quindi optato per riscrivere ex novo il programma, concentrandosi sul paradigma funzionale tipico del linguaggio. Tutto ciò ha permesso di ottenere la soluzione attraverso poche righe di codice facilmente comprensibile.

2.2 Programmazione multi-paradigma: Python

La scelta di Python è dovuta sia alla sua caratteristica multi paradigma, sia alla buona familiarità già acquisita con esso.

Dato il tipo di problema da risolvere e la relativa facilità di gestione dei thread fornita dal linguaggio, si è optato per una risoluzione che utilizzasse un approccio concorrente per il calcolo dei valori $f_{k,c}$.

In particolare sono state proposte due versioni:

1. con concorrenza a livello di statement, che consiste nel calcolo dei valori di $f(k, c)$ riga per riga. Per ogni valore di k si calcolano in parallelo di tutti i valori di $f(k, c)$ per quel singolo k .
2. con concorrenza a livello di sottoprogrammi, che consiste nel calcolare tutti i valori $f(k, c)$ in modo indipendente. Questo approccio richiede però che i processi siano sincronizzati per consentire l'esecuzione del processo per $f(k, c)$ solo quando i processi per $f(k-1, c)$ e $f(k-1, c-w_k)$ hanno terminato la propria esecuzione.

È possibile individuare alcune parti comuni nelle due versioni del programma:

- utilizzo del modulo *threading*
- utilizzo di due array contenenti peso e valore degli oggetti, rispettivamente w e p
- utilizzo di una variabile C per memorizzare la capacità dello zaino
- utilizzo di una tabella f in cui memorizzare i valori di $f(k, c)$ calcolati ed inizializzata a 0
- utilizzo di un dizionario in cui memorizzare lo stato dei thread in esecuzione

- utilizzo di un ciclo *for* per percorrere tutte le celle della tabella f
- utilizzo di una funzione *statement* per il calcolo di $f(k, c)$

Concorrenza a livello di statement

All'interno del ciclo *for* si fa crea un thread per tutti i valori di k , con $1 \leq k \leq \text{length}(w)$. Se $k \neq 1$ il programma attende che il thread di $k-1$ abbia completato la sua esecuzione, altrimenti il thread può partire immediatamente. Sapendo infatti che, $f(k, c) = 0$ quando $k \leq 0$, e che la tabella è inizializzata a 0, non esiste alcun thread k_0 e non è pertanto necessario attenderlo. L'avvio di un thread richiama la funzione *processLine*, che a sua volta per ogni valore di c , con $0 \leq c \leq C$, crea e fa immediatamente partire un thread per la funzione *statement*. Questo programma, oltre agli elementi elencati in precedenza, utilizza un array *cCounter* in cui tenere traccia per, ogni valore di k , del numero di valori di c calcolati. La funzione *statement*, dopo aver calcolato $f(k, c)$ incrementa il di 1 *cCounter*[k] e ne controlla il valore. Se è uguale a C significa che tutte le celle della tabella f per la riga k sono state popolate, quindi si può passare alla riga successiva. Questo avviene settando il flag interno del thread k a *true*, azione che fa immediatamente partire il thread $k+1$ in condizione di *wait()* su di esso. Terminati tutti i thread il problema è risolto, la tabella f è completamente popolata e si può quindi procedere a stamparla a video.

Concorrenza a livello di sottoprogrammi

Con un ciclo *for* si percorrono tutte le celle della tabella f , creando un thread per ognuna e salvandone la condizione nel dizionario. Se ci si trova a percorrere la prima riga della tabella, come visto in precedenza, il valore di ogni cella è già stato popolato dall'inizializzazione e si può pertanto settare il flag di tutti i thread a *true*, così da risvegliare i thread che li stanno attendendo. Per tutte le altre righe viene richiamata la funzione *statement* che si può trovare a dover aspettare uno o due thread:

- se $w[k] < c$ deve aspettare il thread di $(k-1, c)$
- se $w[k] \geq c$ deve aspettare i thread $(k-1, c)$ e $(k-1, c-w[k])$

Calcolati tutti i valori di $f(k, c)$ il problema è risolto e viene stampata a video la tabella.

2.2.1 Considerazioni

La parte che ha richiesto maggior tempo è stata la comprensione e l'ideazione di una strategia per la gestione dei thread. Superato questo piccolo scoglio la scrittura del restante codice si è rivelata semplice, molto leggibile, grazie all'indentazione resa obbligatoria dal linguaggio, e facilmente comprensibile.

2.3 Programmazione logica: Prolog

Prolog è un linguaggio di programmazione che segue il paradigma di programmazione logica, come il nome stesso suggerisce. Anche per Prolog la ricorsione è un concetto fondamentale.

Per questo linguaggio sono state implementate due versioni dell'algoritmo che sebbene molto simili, in quanto entrambe sfruttano la ricorsione, differiscono in termini di prestazioni.

Le regole definite nelle due implementazioni condividono la seguente logica:

1. definizione delle liste peso e valore
2. definizione della capacità massima dello zaino
3. definizione della regola $knapsack(K, C, F)$ che dati in input K e C restituisce il valore di F per i tre possibili casi dell'algoritmo:
 - caso base $f(k, c) = 0$ per $k \leq 0$
 - caso $w[k] > c$
 - caso $w[k] \leq c$

Un'implementazione l'algoritmo sfrutta la ricorsione semplice, mentre l'altra sfrutta il predicato *asserta* nativo di Prolog per aggiungere scrivere nel database del programma. Così facendo, ogni volta che occorre calcolare un valore di $F(k, c)$ il programma interroga prima il database per controllare se è già presente una risposta.

In caso positivo ritorna il valore cercato, altrimenti lo calcola e lo memorizza.

2.3.1 Considerazioni

Memori dell'esperienza avuta con Scheme, si è cercato di avere sin da subito un codice il più possibile snello ed efficiente, concentrandosi unicamente

sulla definizione delle regole descritte dall'algoritmo e dimenticando qualsiasi riferimento a strutture dati tabellari.

Sebbene anche se per questo linguaggio i sorgenti finali siano composti da poche righe di codice, per un osservatore che dovesse trovarsi al suo primo approccio con Prolog potrebbero risultare poco intuitivi.

2.3.2 Conclusioni

Concludendo, il linguaggio che si è preferito durante l'implementazione è stato Python perchè, come detto, si disponeva già di una buona familiarità con esso. Scheme, dopo le difficoltà iniziali dovute più ad un approccio errato che alla complessità del linguaggio, si è rivelato molto versatile e comprensibile. Le maggiori problematiche si sono avute con Prolog, per cui è stato più difficile l'approccio iniziale. Il passaggio da una programmazione tradizionale utilizzata quotidianamente al paradigma logico di Prolog ha richiesto un ulteriore sforzo.