Università di Perugia
Dipartimento di Matematica e Informatica

Computational Intelligence

Multidimensional Knapsack Problem

Professore
Prof. Marco Baioletti

Studente
Nicolò Vescera

Anno Accademico 2022-2023

Indice

1 Obiettivo

Il problema dello zaino multidimensionale (Multidimensional Knapsack Problem) è un'estensione del più noto problema dello Zaino. L'obbiettivo è sempre lo stesso, trovare un set di oggetti che massimizzi il profitto totale facendo in modo di non superare la capienza massima dello zaino, solo con l'aggiunta di più vincoli: non dovremmo solo preoccuparci della capienza dello zaino ma anche di altri n differenti fattori.

Questo problema può essere formalmente sintetizzato come segue:

$$\max \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to: } \sum_{j=1}^{n} r_{i,j} x_j \leq b_i, \quad i = 1, 2, \ldots, m$$

con $x_j = \{0, 1\}$, n il numero di oggetti, m il numero di vincoli, b il limite massimo per ogni vincolo, r il valore per ogni singolo vincolo di ogni oggetto.

2 Istanze del Problema

L'algoritmo per la risoluzione di questo problema verrà testato utilizzando il dataset OR-Library: una raccolta di varie istanze, di differenti dimensioni, per una svariata moltitudine di problemi. Una singola istanza si presenta come segue:

```
6
100,600,1200,2400,500,2000
10
8,12,13,64,22,41
8,12,13,75,22,41
3,6,4,18,6,4
5,10,8,32,6,12
5,13,8,42,6,20
5,13,8,48,6,20
0,0,0,0,8,0
3,0,4,0,8,0
3,2,4,0,8,4
3,2,4,8,8,4
80,96,20,36,44,48,10,18,22,24
3800
```

Codice 1: Esempio di Istanza di un problema MKP con 6 oggetti e 10 vincoli.

La prima riga contiene il numero di oggetti, la seconda il valore di ogni singolo oggetto, la terza il numero di parametri, le successive righe rappresentano i valori dei coefficienti del primo parametro, del secondo e così via fin quando non si raggiunge il numero di parametri. La penultima riga indica il valore massimo per ogni parametro (rappresenta quindi il vincolo che non si può superare) e l'ultima il valore della soluzione ottimale.

ID	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	Value
0	8.0	8.0	3.0	5.0	5.0	5.0	0.0	3.0	3.0	3.0	100.0
1	12.0	12.0	6.0	10.0	13.0	13.0	0.0	0.0	2.0	2.0	600.0
2	13.0	13.0	4.0	8.0	8.0	8.0	0.0	4.0	4.0	4.0	1200.0
3	64.0	75.0	18.0	32.0	42.0	48.0	0.0	0.0	0.0	8.0	2400.0
4	22.0	22.0	6.0	6.0	6.0	6.0	8.0	8.0	8.0	8.0	500.0
5	41.0	41.0	4.0	12.0	20.0	20.0	0.0	0.0	4.0	4.0	2000.0

Tabella 1: Rappresentazione sotto forma tabellare dell'istanza precedente

3 Implementazione

Il problema descritto in precedenza verrà risolto mediante l'implementazione di un Algoritmo Genetico. È stato scelto questo approccio per la sua semplicità di implementazione e relativa velocità di esecuzione. Di seguito saranno descritte le principali componenti che caratterizzano questo algoritmo.

3.1 Initial Population

La prima operazione per l'implementazione di un algoritmo genetico è quella di andare a generare la popolazione iniziale. In questa implementazione verrà fatto in maniera casuale applicando una strategia che permetterà di creare elementi della popolazione (noti anche come *cromosomi*) sempre ammissibili. Questa operazione può essere descritta come segue:

1. Crea una soluzione temporanea vuota (nessun oggetto),
2. Estrai casualmente un oggetto tra quelli disponibili,
3. Prova ad aggiungere questo oggetto alla soluzione temporanea,
   - se è possibile farlo:
     - aggiungi l'oggetto definitivamente alla soluzione,
     - rimuovi l'oggetto dagli oggetti disponibili,
     - continua dal punto 2.
   - se non lo è:
     - aggiungi la soluzione creata alla lista delle soluzioni generate fin ora.
4. Continua fin quando non è stato generato il numero desiderato di soluzioni.

```
PROCEDURE initialize_population
    INPUT:
        num_elem: number of solution to generate
        item_list: list of items
        num_items: number of items
        W: constraints upper bound

    population <- empty list of Solutions
    f_obj <- list of zeros with length num_elem

    FOR i <- 0 TO num_elem - 1 DO
        tmp_sol <- list of zeros with length num_items

        // list of item indexes
        T <- list of integers from 0 to num_items - 1

        R <- list of zeros with length equal to the
             number of constraints in the problem

        j <- randomly select an integer from T
        item <- item_list.pop(j)

        WHILE all elements of (R + item) <= W DO
            tmp_sol[j] <- 1
            R <- R + item

            IF length(T) <= 0 THEN
                EXIT WHILE loop
            END IF

            j <- randomly select an integer from T
            item <- item_list.pop(j)
        END WHILE

        APPEND tmp_sol to population
        f_obj[i] <- calculate the objective function value for tmp_sol
    END FOR
END PROCEDURE
```

Codice 2: Pseudocodice per la Selezione della Popolazione iniziale.

Possiamo notare come nel punto 3, grazie alle operazioni descritte, ogni soluzione generata sarà sicuramente ammissibile perchè verrà sempre controllato che, ogni qualvolta viene aggiunto un oggetto, la nuova soluzione non superi i limiti massimi imposti al problema.

3.2 Mating Pool Selection

Una volta descritto come inizializzare la popolazione, và specificato come selezionare gli individui per formare il Mating Pool, un insieme di $N/2$ coppie che verranno utilizzate al passaggio successivo: **Crossover**. La selezione del Mating Pool può avvenire in due modi:

tramite *Roulette Wheel* o *Tournaments*. Per la semplicità di implementazione e soprattutto per il basso costo computazionale è stato implementato il metodo basato sui Tornei. Vengono selezionati casualmente un numero k di elementi della popolazione e tra questi viene preso il cromosoma con valore di fitness più alto. Tramite questa operazione vengono generate $N/2$ coppie che poi verranno passate all'operatore di Crossover. Di seguito lo pseudocodice che riassume questa operazione.

```
FUNCTION tournament(k: INTEGER) RETURNS Solution
    random_select_solutions <- select k solution from population randomly

    max_index <- select the index of solution with max fitness value
                 in random_select_solutions

    RETURN population[max_index]
END FUNCTION

PROCEDURE select_mating_pool
    INPUT:
        population: List of all chromosomes (solutions)
        k: tournament parameter

    mating_pool <- empty list

    FOR i <- 1 TO LENGTH(population) // 2 DO
        c1 <- tournament(k)
        c2 <- tournament(k)

        APPEND (c1, c2) to mating_pool
    END FOR

    RETURN mating_pool
END PROCEDURE
```

Codice 3: Implementazione del metodo per la selezione del Mating Pool basata sui Tornei.

3.3 Crossover Operator

L'operatore di Crossover prende gli elementi del Mating Pool e genera un nuovo elemento chiamato *figlio*. Questo è un **Crossover Uniforme**: dati due cromosomi, che prendono il nome di *padri* (indicati con s_1 e s_2), viene generato un nuovo individuo figlio che eredita in modo uniforme i geni dai due padri. Il crossover viene eseguito con una probabilità data dal parametro `pcross`. Di seguito lo pseudocodice.

```
FUNCTION uniform_crossover_operator(s1, s2) RETURNS Solution
    c = ARRAY OF ZEROS with length LENGTH(s1)

    FOR i <- 1 TO LENGTH(s1) DO
        IF RANDOM_BOOLEAN() THEN
            c[i] <- s1[i]
        ELSE
            c[i] <- s2[i]
        END IF
    END FOR

    RETURN c
END FUNCTION

PROCEDURE do_crossover
    INPUT:
        mating_pool: N/2 couples from Selecting Mating Pool Phase
        pcross: Crossover probability

    children <- empty list

    FOR EACH (s1, s2) IN mating_pool DO
        IF RANDOM_FLOAT() < pcross THEN
            c <- uniform_crossover_operator(s1, s2)
            APPEND c to children
        ELSE
            APPEND s1 to children
            APPEND s2 to children
        END IF
    END FOR

    RETURN children
END PROCEDURE
```

Codice 4: Implementazione dell'operatore di Crossover.

3.4 Mutation Operator

L'operatore di Mutazione è il responsabile di alterare i geni dei cromosomi risultanti dalla precedente fase di Crossover (indipendentemente se sono genitori o figli). In base al parametro `pmut` (*mutation probability*), per ogni gene di ogni cromosoma, viene scelto se effettuare una mutazione oppure no. La mutazione consiste nel cambiare il rispettivo gene scelto tramite una semplice operazione di negazione.

```
PROCEDURE do_mutation
    INPUT:
        children: Crossover Phase Result
        pmut: Mutation Probability

    FOR EACH child in children DO
        FOR i <- 0 TO LENGTH(child) - 1 DO
            IF RANDOM_FLOAT() < pmut THEN
                child[i] <- not child[i]
            END IF
        END FOR
    END FOR
END PROCEDURE
```

Codice 5: Pseudocodice dell'operatore di Crossover.

3.5 Repair Operator

Una volta terminata la fase di Mutazione, abbiamo in output una serie di cromosomi figli che molto probabilmente non rispettano i criteri del problema e non sono quindi soluzioni ammissibili. C'è la necessità quindi di andare a "riparare" ogni cromosoma che porti ad una soluzione non valida. L'operatore che si occupa di questa procedura è stato implementato in 2 fai: la prima **Sottrattiva** e la seconda **Additiva**. L'idea che c'è dietro a questa procedura può essere riassunta come segue:

1. Per ogni figlio controlla se è una soluzione ammissibile.
2. Se lo è, torna al punto 1
3. Altrimenti:
   (a) **Fase Sottrattiva**: rimuovi un oggetto alla volta dalla soluzione, fin quando la soluzione non diventa ammissibile
   (b) **Fase Additiva**: aggiungi un oggetto alla volta alla soluzione, fin quando è possibile (fin quando la soluzione rimane ancora ammissibile).
   (c) Torna al punto 1

Durante le due fasi, gli oggetti vengo scelti in modo ordinato in base ad un parametro chiamato *Importanza*. L'idea di base è di rimuovere prima gli oggetti meno importanti e di aggiungere poi quelli più importanti. Questo parametro viene calcolato in base alla seguente formula:

$$Importance(j) = \frac{\sum_{i=1}^{m} r_{i,j} b_i}{\sum_{i=1}^{m} b_i} \frac{1}{p_j} \tag{1}$$

```
PROCEDURE repair_operator
    INPUT:
        children: Mutation Phase Result

    sorted_objects <- sort object according to equation 1
    sorted_index <- get indexes of sorted object by Importance

    FOR EACH child in children DO
        IF child is feasible solution DO
            continue
        END IF

        // DROP PHASE
        child <- remove object from child, starting from less Important,
                until child is feasible solution

        // ADD PHASE
        child <- add object to child, starting from most Important,
                until child is feasible solution
    END FOR
END PROCEDURE

```

Codice 6: Pseudocodice dell'operatore di Riparazione.

3.6 Select New Population

Una volta completata la fase di Riparazione e quindi con un'insieme di soluzioni sicuramente accettabili, và selezionata la nuova popolazione da passare alla generazione successiva dell'algoritmo genetico. Per farlo è stata implementata la strategia in cui *sopravvivono i migliori* (un caso particolare dell'elitismo). In questa fase vengono scelti i migliori individui, in base alla fitness, della popolazione indipendentemente se sono "figli" o "genitori" (non conta quindi l'età).

```
PROCEDURE select_new_population
    INPUT:
        population: Actual population
        children: Repair Phase Result
        num_elem: max population length

    total_solutions <- population + children
    total_fitnesses <- compute fitness value for all total_solutions

    total_solutions <- sort by fitness value (desc)
    total_fitnesses <- sort by fitness value (dec)

    population <- total_solutions[0...num_elem]
    fitnesses <- total_fitnesses[0...num_elem]
END PROCEDURE
```

Codice 7: Pseudocodice della fase di selezione della nuova popolazione.

4 Stima dei Parametri

Il punto cruciale quando si esegue un algoritmo genetico è l'individuazione dei vari parametri che lo caratterizzano.

- `pcross` (*Crossover Probability*): probabilità di effettuare il crossover di un elemento del Mating Pool (generalmente è elevata),
- `pmut` (*Mutation Probability*): probabilità di mutare un singolo gene di un elemento dell'insieme dei figli generato dalla fase di Crossover (generalmente è bassa),
- `ngen` (*Max Generation Number*): numero di Generazioni dopo il quale l'algoritmo si ferma,
- `plen` (*Population Length*): lunghezza della popolazione,
- `tk` (*Tournaments k*): massimo numero di elementi tra cui scegliere il migliore durante i tornei.

Per la stima di quest'ultimi è stato selezionato un dataset di *Tuning*, composto da 11 istanze più o meno grandi, che meglio rappresentavano il dataset di tutte le istanze a disposizione. In questo dataset abbiamo una varietà di istanze molto vasta, possiamo trovarne alcune che hanno 6 oggetti e 10 parametri fino ad altre con oltre i 100 oggetti. Di seguito una tabella riassuntiva.

file	Oggetti	Parametri
MKP01.txt	6	10
MKP03.txt	15	10
MKP05.txt	28	10
MKP07.txt	50	5
MKP11.txt	28	2
MKP17.txt	105	2
MKP21.txt	30	5
MKP23.txt	40	5
MKP33.txt	60	5
MKP41.txt	80	5
MKP47.txt	90	5

Tabella 2: Contenuto del dataset di Tuning.

Per ogni file nel dataset è stato eseguito l'algoritmo 5 volte e sono stati salvati i risultati: nome del file, soluzione trovata (*found*), soluzione originale (*target*), un bit per indicare il successo e la differenza tra *target* e *found*. Sono stati poi scelti altri parametri e rieseguita la fase di tuning. Per ogni esecuzione della fase di tuning si ottiene un file di questo tipo:

```
,file,gen,found,target,success,diff
0,data/MKP01.txt,52,3800.0,3800.0,True,0.0
1,data/MKP01.txt,1,3800.0,3800.0,True,0.0
2,data/MKP01.txt,0,3800.0,3800.0,True,0.0
3,data/MKP01.txt,48,3800.0,3800.0,True,0.0
4,data/MKP01.txt,87,3800.0,3800.0,True,0.0
5,data/MKP03.txt,20,3915.0,4015.0,False,100.0
```

Codice 8: File risultante da una esecuzione della fase di Tuning.

Sono stati selezionati i seguenti parametri e utilizzati per eseguire la fase di Tuning.

pmut	pcross	ngen	plen	tk
0.01	0.90	100	16	5
0.02	0.97	250	40	31
0.02	0.97	250	80	40
0.02	0.97	250	100	40
0.03	0.97	250	71	33
0.05	0.97	250	20	7
0.05	0.97	250	40	31
0.05	0.97	250	100	61
0.05	0.99	250	20	16
0.05	0.99	250	80	50
0.05	0.99	250	100	80

Tabella 3: Lista dei parametri scelti per essere utilizzati nella fase di Tuning.

Per ogni esecuzione poi sono stati generati 2 grafici: il primo ci mostra la differenza media, in percentuale, tra *target* e *found*; il secondo il numero di successi e fallimenti per ogni file.

Figura 1: Grafici ottenuti dalla fase di Tuning.

Dai precedenti grafici si possono individuare i 3 migliori set di parametri che verranno utilizzati nella successiva fase di Valutazione delle Performance.

Parameters	**Success Ratio (%)**	**Max AVG Diff (%)**
pmut 0.05;pcross 0.99;ngen 250;plen 100;tk 80	36.4	12.0
pmut 0.02;pcross 0.97;ngen 250;plen 100;tk 40	32.7	5.0
pmut 0.05;pcross 0.97;ngen 250;plen 100;tk 61	32.7	21.0

Tabella 4: Top 3 set di parametri individuati nella fase di Tuning.

5 Valutazione delle Performance

Infine, dopo aver individuato i 3 set di parametri che performano meglio, è stato scelto un dataset di *Testing* su cui testare le scelte della fase precedente.

file	Oggetti	Parametri
MKP02.txt	10	10
MKP04.txt	20	10
MKP06.txt	39	5
MKP08.txt	60	30
MKP10.txt	28	2
MKP12.txt	28	2
MKP20.txt	30	5
MKP22.txt	30	5
MKP24.txt	40	5
MKP30.txt	50	5
MKP36.txt	70	5
MKP40.txt	80	5
MKP46.txt	90	5
MKP48.txt	27	4
MKP50.txt	29	2
MKP54.txt	28	4

Tabella 5: Contenuto del dataset di Testing.

Similmente alla fase di Tuning, per ogni elemento del dataset di Testing è stato eseguito l'algoritmo genetico 10 volte, utilizzando i parametri della Tabella 4 e salvando sempre i risultati come in precedenza. I risultati finali sono riassunti dai seguenti grafici.

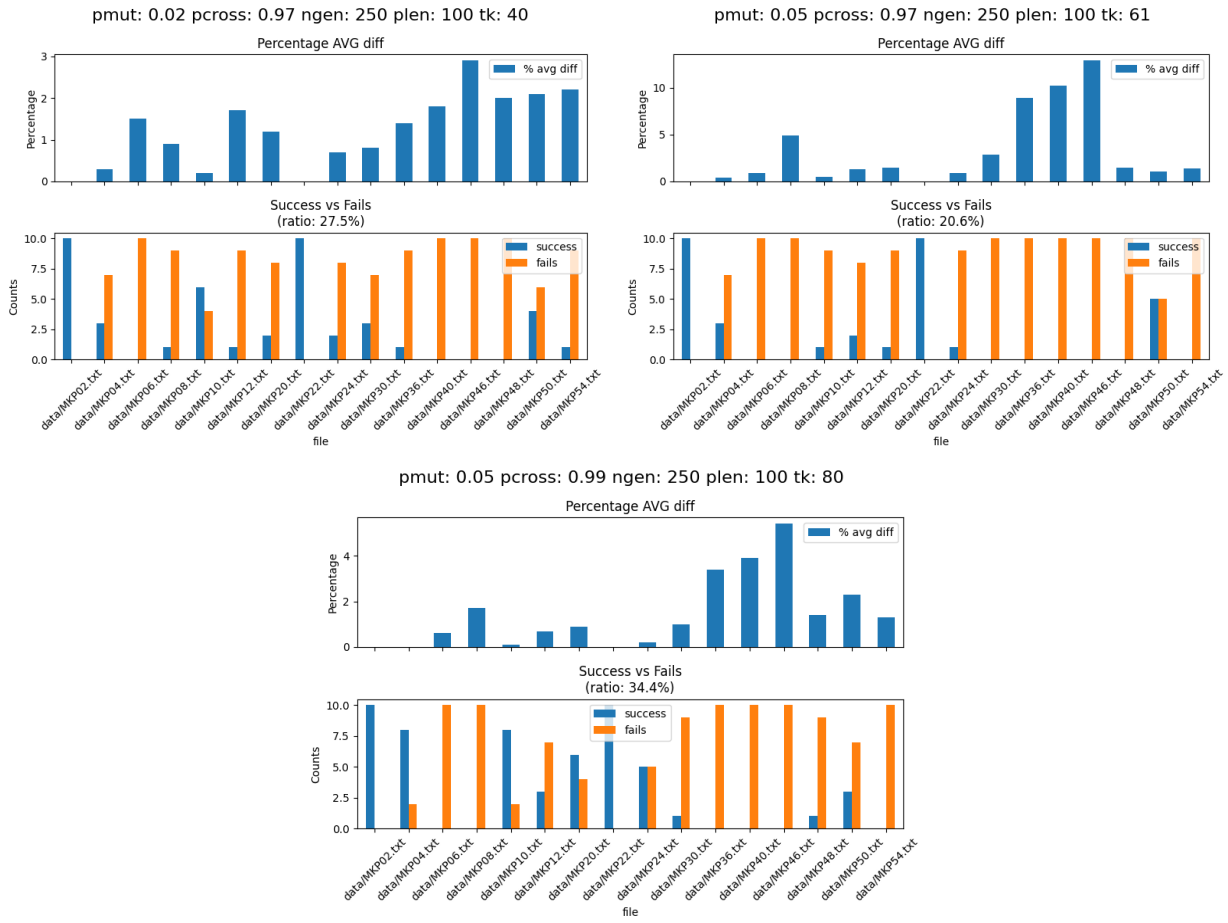

Figura 2: Grafici ottenuti dalla fase di Tuning.

Riferimenti

[1] J. E. Beasley. «OR-Library: Distributing Test Problems by Electronic Mail». In: *Journal of the Operational Research Society* 41.11 (nov. 1990), pp. 1069–1072. ISSN: 1476-9360. DOI: 10.1057/jors.1990.166. URL: https://doi.org/10.1057/jors.1990.166.

[2] P. C. Chu e J. E. Beasley. «A Genetic Algorithm for the Multidimensional Knapsack Problem». In: *Journal of Heuristics* 4.1 (giu. 1998), pp. 63–86. ISSN: 1572-9397. DOI: 10.1023/A:1009642405419. URL: https://doi.org/10.1023/A:1009642405419.

[3] Nicolò Vescera. *Multidimentional Knapsack Problem Solver using a Genetic Algorihtm*. URL: https://github.com/ncvescera/mkp-gasolver.