Algoritmi di Ottimizzazione Combinatoria e su Rete - 2024/2025

Testing and comparing PLI and Genetic Algorithm on a Google Hashcode problem

Giuseppe Maglione - M63001777

Chapter 1

Introduzione del problema

1.1 Descrizione

Stai organizzando un hub di Hash Code e vuoi ordinare la pizza per i partecipanti. Fortunatamente, c'è una pizzeria nelle vicinanze con una pizza davvero buona. La pizzeria offre diversi tipi di pizza e, per rendere l'ordinazione del cibo interessante, possiamo ordinare al massimo una pizza di ogni tipo. Per fortuna, ci sono molti tipi di pizza tra cui scegliere! Ogni tipo di pizza ha una dimensione specificata: la dimensione è il numero di fette di pizza che contiene. In base al numero di partecipanti, hai stimato un numero massimo di fette che desideri ordinare – per ridurre gli sprechi, l'obiettivo è ordinare quante più fette di pizza possibile, ma non più del numero massimo.

1.2 Input

Ogni set di dati di input è fornito in un file di testo semplice contenente esclusivamente caratteri ASCII, con righe terminate da un singolo carattere "/n" (terminazioni di riga in stile UNIX). Quando una singola riga contiene più elementi, questi sono separati da singoli spazi.

La prima linea del dataset contiene le seguenti informazioni:

- Un intero M ($1 \leq M \leq 10^9$) – il numero massimo di fette di pizza da ordinare.
- Un intero N ($1 \leq N \leq 10^5$) – il numero di tipi di pizza diversi.

La seconda riga contiene N interi: il numero di fette per ciascun tipo di pizza, in ordine non decrescente:

- $S_0 \leq S_1 \leq ... \leq S_{N-1} \leq M$

1.3 Output

L'output deve contenere due righe:

- La prima riga deve contenere un solo intero K ($0 \leq K \leq N$) – il numero di pizze diverse da ordinare.
- La seconda riga deve contenere K numeri – i tipi di pizza da ordinare (numerati da 0 a $N-1$ nello stesso ordine in cui sono forniti in input).

Il numero dotale di fette nelle pizze ordinate non deve superare il valore M.

1.4 Punteggio

Il punteggio viene calcolato aggiungendo 1 punto per ogni fetta presente nelle pizze ordinate.
Si nota la presenza di diversi dataset che rappresentano istanze separate del problema. Il puntaggio finale sarà dato dalla somma dei punteggi per ciascun dataset individuale.

Chapter 2

Algoritmo Genetico

2.1 Richiami teorici

L'idea alla base degli Algoritmi Genetici, come accade per molti altri algoritmi euristici, si ispira a un principio della selezione naturale: l'evoluzione genetica. Il concetto è quello di far evolvere una popolazione iniziale di soluzioni attraverso specifici meccanismi di selezione, con l'obiettivo di ottenere soluzioni via via migliori in termini di funzione obiettivo.
L'evoluzione della popolazione avviene attraverso un processo di riproduzione: due individui si combinano per generare una nuova soluzione, che eredita alcune caratteristiche dai genitori, mentre altre vengono modificate in seguito all'influenza dell'ambiente. La scelta degli individui che contribuiranno alla generazione della nuova popolazione è guidata dal principio della selezione naturale, il quale stabilisce che hanno maggiori probabilità di sopravvivere e riprodursi gli individui con una fitness più elevata, ovvero quelli con una migliore capacità di adattamento. In generale, le fasi di un algoritmo genetico sono:

1. **Codifica**: Si stabilisce il formato di ciascuna soluzione del problema, stabilendone le variabili aleatorie, la loro disposizione e i valori che queste possono assumere.

2. **Inizializzazione e valutazione della fitness**: Si genera un insieme di soluzioni iniziale e si valuta, per ciascuna di esse, un valore di fitness.

3. **Selezione**: Si selezionano, secondo specifiche strategie, le coppie di soluzioni della popolazione a cui applicare gli operatori genetici.

4. **Generazione**: Si applicano gli operatori genetici alle coppie di soluzioni individuate al fine di generare nuovi individui.

5. **Sostituzione**: Le soluzioni esistenti della popolazione vengono sostituite con quelle appena generate.

Le fasi 3,4 e 5 vengono ripetute iterativamente fino al raggiungimento di una condizione di arresto prefissata.

2.2 Soluzione

Nel seguito sono descritti gli approcci utilizzati per la realizzazione dell'algoritmo. Per ciascuna delle fasi analizzate è presente uno pseucodice, mentre il codice completo viene riportato nell'ultima sezione del capitolo [2.3].

2.2.1 Codifica degli individui

È stata adottata una codifica degli individui, e quindi delle soluzioni, che prevede di rappresentarli tramite stringhe binarie di lunghezza fissa N (il numero di tipi di pizze disponibili). Il bit i-esimo all'interno della stringa assume valore 1 se la pizza di tipo i viene ordinata, altrimenti 0.

2.2.2 Prima popolazione e Fitness

Per la generazione della prima popolazione è stato scelto di adottare una tecnica mista, che prevede l'uso di tre strategie:

- Ordinamento decrescente delle fette (greedy),
- Ordinamento crescente (greedy inverso),
- Ordinamento casuale.

Questo permette di generare uno spazio delle soluzioni iniziali quanto più vasto possibile, evitando che l'algoritmo lavori su una porzione dello spazio delle soluzioni troppo ristretta. Per ciascun individuo, le fette vengono aggiunte finché la somma parziale rimane sotto il limite M.

```
def genera_prima_popolazione
    popolazione = lista vuota

    for individuo da 1 a DIMENSIONE_POPOLAZIONE:
        strategia = random(['decrescente', 'crescente', 'casuale'])
        indici = [0, 1, 2, ..., N-1]

        if strategia = 'decrescente':
            ordina indici per valore decrescente delle fette (greedy)
        else if strategia = 'crescente':
            ordina indici per valore crescente delle fette (greedy inverso)
        else if strategia = 'casuale':
            mescola casualmente gli indici

        totale = 0
        bitstring = [0, 0, 0, ..., 0]

        for indice i in indici:
            if (totale + fette[i]) <= M:
                aggiorna totale
                bitstring[i] = 1

        aggiungi bitstring a popolazione

    return popolazione
```

La funzione di fitness valuta la qualità di una soluzione sommando i valori delle fette selezionate (quelle per cui il bit è 1). Se la somma non supera M, si restituisce il totale. In caso contrario, viene applicata una penalità lineare proporzionale all'eccesso, in modo da scoraggiare le soluzioni non valide senza escluderle completamente.

```
def calcola_fitness
    totale = 0
    for indice i in bitstring:
        if bitstring[i] = 1:
            aggiorna totale

    if totale <= M:
        # Soluzione ammissibile: fitness = valore totale
        fitness = totale
    else:
        # Soluzione non ammissibile: applica penalita lineare
        penalita = totale - M
        fitness = M - penalita

    return fitness
```

2.2.3 Generazione

La generazione dei nuovi individui consiste nel far riprodurre una coppia di soluzioni tramite un'operazione di crossover. Il crossover avviene con due punti di taglio scelti in maniera casuale: un figlio eredita una porzione intermedia del secondo genitore, mantenendo le estremità del primo. Questo consente un buon bilanciamento tra conservazione e innovazione genetica.

```
def crossover
    punto1 = genera un punto casuale
    punto2 = genera un punto casuale
    figlio = lista vuota

    # Segmento 1: da genitore1 (posizioni 0 fino a punto1)
    for indice i da 0 a punto1:
        copia genitore1[i] a figlio

    # Segmento 2: da genitore2 (posizioni punto1 fino a punto2)
    for indice i da punto1 a punto2:
        copia genitore2[i] a figlio

    # Segmento 3: da genitore1 (posizioni punto2 fino alla fine)
    for indice i da punto2 a lunghezza_bitstream:
        copia genitore1[i] a figlio

    return figlio
```

La mutazione consiste nell'invertire ciascun bit (da 0 a 1 o viceversa) con una certa probabilità, permettendo di esplorare nuove soluzioni non presenti nei genitori. Notiamo anche come tale procedura sia applicata a soltanto uno dei due figli generati. Il motivo di una tale scelta è che permette di ottenere, per ciascuna riproduzione, una soluzione più conservativa che mantiene le caratteristiche dei genitori (quella senza mutazione), e una soluzione più esplorativa che subisce dei cambiamenti nei geni (quella con mutazione).

```
def muta_bitstring
    bitstring_mutato = lista vuota

    for bit in bitstring:
        probabilita = genera_numero_casuale(0, 1)

        if probabilita < TASSO_MUTAZIONE:
            # MUTAZIONE: inverte il bit
            bit_nuovo = NOT bit
            aggiungi bit_nuovo a bitstring_mutato
        else:
            # NESSUNA MUTAZIONE: mantiene il bit originale
            aggiungi bit a bitstring_mutato

    return bitstring_mutato
```

2.2.4 Evoluzione e Arresto

La selezione degli individui avviene tramite torneo, in cui si scelgono casualmente k individui e viene selezionato quello con la fitness migliore.

```
def selezione_torneo
    partecipanti = seleziona casualmente tra la popolazione

    partecipanti_valutati = lista vuota
    for ogni individuo in partecipanti:
        fitness_individuo = calcola_fitness
        aggiungi (individuo, fitness_individuo) a partecipanti_valutati

    ordina partecipanti_valutati per fitness (dal migliore al peggiore)
    vincitore = primo_elemento(partecipanti_valutati)

    return vincitore
```

L'evoluzione della popolazione continua per un numero massimo di generazioni o fino a quando non si raggiunge un tempo limite. L'algoritmo tiene traccia del miglior individuo visto finora, aggiornandolo se viene trovata una soluzione con fitness superiore. Data la conoscenza della soluzione ottima per ogni istanza del problema, il processo può terminare anticipatamente anche nel caso in cui questa venga trovata, ovvero quando la fitness assume valore pari a M.

2.2.5 Versioni precedenti

Per arrivare alla versione finale dell'algoritmo presentata, sono state sperimentate diverse soluzioni, con l'obiettivo di capire come indirizzare correttamente l'evoluzione della popolazione.

- Nel **primo tentativo**, è stato adottato un approccio abbastanza standard: ogni generazione produceva un solo figlio tramite un crossover a un punto, seguito da una fase di mutazione e da una funzione di filtro. Quest'ultima aveva il compito di rendere ammissibili eventuali soluzioni che eccedevano la capacità massima (M), rimuovendo pizze finché il vincolo non veniva rispettato. Il problema principale di questo approccio era che l'algoritmo tendeva a bloccarsi facilmente su falsi massimi, finendo per privilegiare soluzioni che sembravano buone, ma impedivano un'esplorazione più ampia dello spazio delle soluzioni.

- Nel **secondo tentativo**, per cercare di ridurre il rischio di rimanere intrappolati in massimi locali, è stata eliminata la funzione di filtro. Al suo posto, si è deciso di accettare anche soluzioni non ammissibili, penalizzandole opportunamente nella funzione di fitness.

In questo modo, tali soluzioni avevano comunque una possibilità, seppur bassa, di essere selezionate nelle generazioni successive. L'introduzione della penalità ha portato piccoli miglioramenti nella qualità delle soluzioni trovate.

- Con il **terzo tentativo**, sono state introdotte due modifiche principali: il crossover è passato a una versione a due punti, e ogni coppia di genitori produceva due figli anziché uno. Il primo figlio veniva mantenuto così com'era, mentre il secondo veniva sottoposto alla mutazione. L'idea era quella di ottenere due figli con ruoli diversi: uno più conservativo, che ereditava le caratteristiche dei genitori, e uno più esplorativo, che introduceva maggiore variabilità. In altre parole, si cercava di aumentare la distribuzione dello spazio delle soluzioni visitato.

- Infine, il **quarto tentativo** ha riguardato una vera e propria fase di tuning dei parametri, volta a individuare un insieme di valori in grado di far convergere l'algoritmo in maniera affidabile e stabile sulle diverse istanze del problema testate.

2.3 Codice

Si riporta il codice completo dell'algoritmo implementato.

```
import random
import time

# parameters
POPULATION_SIZE = 50
NUM_GENERATIONS = 500
MUTATION_RATE = 0.05
TOURNAMENT_SIZE = 3
TIME_LIMIT = 60

def fitness(bitstring, slices, M):
    total = sum(s for i, s in enumerate(slices) if bitstring[i])
    if total <= M:
        return total
    else:
        return M - (total - M)  # penalita lineare

def tournament_selection(population, slices, M, k=TOURNAMENT_SIZE):
    selected = random.sample(population, k)
    selected.sort(key=lambda x: fitness(x, slices, M), reverse=True)
    return selected[0]

def crossover_2point(parent1, parent2):
    point1, point2 = sorted(random.sample(range(len(parent1)), 2))
    return parent1[:point1] + parent2[point1:point2] + parent1[point2:]

```

```
27
28  def mutate(bitstring):
29      mutated_bitstring = []
30      for bit in bitstring:
31          if random.random() < MUTATION_RATE:
32              mutated_bitstring.append(bit ^ 1)    # inverte il bit
33          else:
34              mutated_bitstring.append(bit)    # lascia il bit inalterato
35      return mutated_bitstring
36
37  def generate_first_population(M, N, slices):
38      population = []
39
40      for _ in range(POPULATION_SIZE):
41          # scegliamo casualmente in che modo generare la prima popolazione
42          strategy = random.choice(['desc', 'asc', 'shuffle'])
43          indices = list(range(N))
44
45          if strategy == 'desc':
46              indices.sort(key=lambda i: -slices[i])      # greedy
47          elif strategy == 'asc':
48              indices.sort(key=lambda i: slices[i])       # greedy inverso
49          elif strategy == 'shuffle':
50              random.shuffle(indices)                     # casuale
51
52          total = 0
53          bitstring = [0] * N
54
55          for i in indices:
56              if total + slices[i] <= M:
57                  total += slices[i]
58                  bitstring[i] = 1
59
60          population.append(bitstring)
61
62      return population
63
64  def solve(M, N, slices):
65      start_time = time.perf_counter()
66      population = generate_first_population(M, N, slices)
67      best_individual = max(population, key=lambda x: fitness(x, slices, M))
68      best_score = fitness(best_individual, slices, M)
69
70      for gen in range(NUM_GENERATIONS):
71          elapsed = time.perf_counter() - start_time
72          if elapsed >= TIME_LIMIT:
73              print(f"\t\t[Geneneration␣{gen+1}]␣Early␣stop,␣time␣limit␣
                    reached␣({elapsed:.2f}s)")
74              break
75
76          new_population = []
77
78          while len(new_population) < POPULATION_SIZE:
79              parent1 = tournament_selection(population, slices, M)
80              parent2 = tournament_selection(population, slices, M)
81
82              child1 = crossover_2point(parent1, parent2)
83              child2 = child1[:]  # copia diretta
```

```
84
85              child1 = mutate(child1)  # solo il primo figlio mutato
86
87              new_population.append(child1)
88              if len(new_population) < POPULATION_SIZE:
89                  new_population.append(child2)
90
91          population = new_population
92
93          # salva il miglior individuo
94          current_best = max(population, key=lambda x: fitness(x, slices, M))
95          current_score = fitness(current_best, slices, M)
96          if current_score > best_score:
97              best_individual = current_best
98              best_score = current_score
99
100         if gen == 0 or (gen + 1) % 50 == 0:
101             print(f"\t\t[Genèneration␣{gen+1}]␣Current␣best␣fitness␣=␣{
                    best_score}")
102
103         if best_score == M:
104             print(f"\t\t[Genèneration␣{gen+1}]␣Early␣stop,␣fitness␣reached␣
                    max␣score␣({M})")
105             break
106
107     elapsed_time = time.perf_counter() - start_time
108     return best_score, elapsed
109
110 """
111 OLD FITNESS FUNC
112 def fitness(bitstring, slices, M):
113     total = sum(s for i, s in enumerate(slices) if bitstring[i])
114     return total if total <= M else 0  # penalizza se invalido
115 """
116
117 """
118 OLD REPAIR FUNC
119 def repair(bitstring, slices, M):
120     total = sum(s for i, s in enumerate(slices) if bitstring[i])
121     if total <= M:
122         return bitstring
123
124     # rimuove pizze finche rientra in M
125     indices = [i for i, bit in enumerate(bitstring) if bit == 1]
126     random.shuffle(indices)
127     for i in indices:
128         bitstring[i] = 0
129         total -= slices[i]
130         if total <= M:
131             break
132     return bitstring
133 """
```

Chapter 3

Programmazione Lineare Intera

3.1 Soluzione

Il problema proposto può essere facilmente modellato con le tecniche della Programmazione Lineare Intera (PLI). Come risolutore del problema è stato scelto Gurobi.

3.1.1 Variabili decisionali

Sono state introdotte le variabili decisionali binarie x_i, ciascuna delle quali assume valore 1 se la pizza i viene selezionata, altrimenti 0.

```
x = model.addVars(N, vtype=GRB.BINARY, name="x")
```

3.1.2 Funzione obiettivo

L'obiettivo del problema è avere a disposizione quante più fette possibili, e quindi massimizzare il numero di pizze da ordinare, senza però eccedere la quantità massima M. Poiché ad ogni pizza i viene associato un corrispondente numero di fette s_i, la funzione obiettivo può essere espressa nel seguente modo.

$$\max \sum_{i=1}^{N} s_i \cdot x_i$$

In Gurobi, l'espressione diventa.

```
model.setObjective(
    quicksum(slices[i] * x[i] for i in range(N)), GRB.MAXIMIZE
    )
```

3.1.3 Vincoli

Gli unici vincoli da aggiungere riguardano il numero massimo di fette da non eccedere e l'interezza della soluzione.

$$\sum_{i=1}^{N} s_i \cdot x_i \leq M$$

$$x_i \in \{0, 1\} \quad \forall i = 1, \ldots, N$$

Poiché il vincolo di interezza è già stato espresso nella definizione delle variabili x_i, non resta che aggiungere a Gurobi quello sul numero di fette.

```
model.addConstr(
    quicksum(slices[i] * x[i] for i in range(N)) <= M, name="max_slices"
    )
```

3.2 Codice

Si riporta il codice completo della soluzione con Gurobi.

```
from gurobipy import Model, GRB, quicksum
import time
import sys
import os
from contextlib import contextmanager

@contextmanager
# per evitare che gurobi scriva sul terminale
def suppress_stdout():
    with open(os.devnull, 'w') as devnull:
        old_stdout = os.dup(1)
        os.dup2(devnull.fileno(), 1)
        try:
            yield
        finally:
            os.dup2(old_stdout, 1)
            os.close(old_stdout)

def solve(M, N, slices):
    start_time = time.perf_counter()
    with suppress_stdout():
        model = Model("pizza")
        model.Params.OutputFlag = 0  # silenzia l'output

        # cerchiamo di rendere confrontabili le soluzioni
        # forziamo gurobi a usare un branch and bound "puro"
        model.Params.Threads = 1
        model.Params.MIPGap = 0.0       # impone un gap tra soluzione
           trovata e ottima pari a 0
        model.Params.Presolve = 0       # gurobi non semplifica il modello
           per risolvere
```

```
30          model.Params.Heuristics = 0      # disattiva l'uso di euristiche
31          model.Params.TimeLimit = 60
32
33          # variabili binarie: x[i] = 1 se prendo la pizza i, altrimenti 0
34          x = model.addVars(N, vtype=GRB.BINARY, name="x")
35
36          # vincolo: somma delle fette selezionate <= M
37          model.addConstr(quicksum(slices[i] * x[i] for i in range(N)) <= M,
                name="max_slices")
38
39          # funzione obiettivo: massimizzare il numero di fette
40          model.setObjective(quicksum(slices[i] * x[i] for i in range(N)),
                GRB.MAXIMIZE)
41
42          # risoluzione
43          model.optimize()
44
45          # recupera soluzione
46          if model.SolCount > 0:
47              total_slices = sum(slices[i] for i in range(N) if x[i].X > 0.5)
                    # somma delle fette selezionate
48              elapsed = time.perf_counter() - start_time
49              return total_slices, elapsed
50          else:
51              elapsed = time.perf_counter() - start_time
52              return 0, elapsed  # nessuna soluzione trovata
```

Chapter 4

Confronto tra le soluzioni

Le due soluzioni proposte sono state confrontate sulle 4 istanze del problema a disposizione. La difficoltà, espressa in termini di dimensioni del dataset, aumenta al proseguire delle istanze. Ad entrambi i risolutori è stato assegnato un tempo massimo di esecuzione pari a 60 secondi e una modalità di esecuzione su singolo thread (differentemente da come lavora in genere Gurobi), in modo da ottenere dei risultati che fossero quanto più confrontabili possibile. Per ciascuna delle soluzioni riportate, viene anche mostrato il tempo che è stato richiesto per calcolarla.

4.1 Risultati

I valori di output per ciascuna soluzione vengono riportati di seguito.

```
------------------------------------------------------------------------

== RISOLUZIONE ==
      Dataset: Datasets/small.in
      Max fette: 100
      Numero tipi di pizza: 10
      Lista fette per pizza: [4, 14, 15, 18, 29, 32, 36, 82, 95, 95]
      [] Risoluzione con PLI in corso...
      [] Risoluzione con Algoritmo Genetico in corso...
            [Geneneration 1] Current best fitness = 99
            [Geneneration 50] Current best fitness = 99
            [Geneneration 100] Current best fitness = 99
            [Geneneration 150] Current best fitness = 99
            [Geneneration 173] Early stop, fitness reached max score
               (100)
== RISULTATI ==
      PLI -> Score: 100, Tempo: 0.0085s
      Genetico -> Score: 100, Tempo: 0.1321s
------------------------------------------------------------------------
```

```
== RISOLUZIONE ==
      Dataset: Datasets/medium.in
      Max fette: 4500
      Numero tipi di pizza: 50
      Lista fette per pizza: [...]
      [] Risoluzione con PLI in corso...
      [] Risoluzione con Algoritmo Genetico in corso...
            [Genеneration 1] Current best fitness = 4498
            [Genеneration 50] Current best fitness = 4499
            [Genеneration 100] Current best fitness = 4499
            [Genеneration 113] Early stop, fitness reached max score
               (4500)
== RISULTATI ==
      PLI -> Score: 4500, Tempo: 0.0283s
      Genetico -> Score: 4500, Tempo: 0.1928s
--------------------------------------------------------------------------

== RISOLUZIONE ==
      Dataset: Datasets/big.in
      Max fette: 1000000000
      Numero tipi di pizza: 2000
      Lista fette per pizza: [...]
      [] Risoluzione con PLI in corso...
      [] Risoluzione con Algoritmo Genetico in corso...
            [Genеneration 1] Current best fitness = 999999725
            [Genеneration 50] Current best fitness = 999999725
            [Genеneration 100] Current best fitness = 999999725
            [Genеneration 150] Current best fitness = 999999725
            [Genеneration 200] Current best fitness = 999999725
            [Genеneration 250] Current best fitness = 999999725
            [Genеneration 300] Current best fitness = 999999725
            [Genеneration 350] Current best fitness = 999999725
            [Genеneration 400] Current best fitness = 999999725
            [Genеneration 450] Current best fitness = 999999725
            [Genеneration 500] Current best fitness = 999999725
== RISULTATI ==
      PLI -> Score: 1000000000, Tempo: 2.1111s
      Genetico -> Score: 999999725, Tempo: 16.0980s
--------------------------------------------------------------------------

== RISOLUZIONE ==
      Dataset: Datasets/extra.in
      Max fette: 505000000
      Numero tipi di pizza: 10000
      Lista fette per pizza: [...]
      [] Risoluzione con PLI in corso...
      [] Risoluzione con Algoritmo Genetico in corso...
            [Genеneration 1] Current best fitness = 504999983
            [Genеneration 50] Current best fitness = 504999983
            [Genеneration 100] Current best fitness = 504999983
            [Genеneration 150] Current best fitness = 504999983
            [Genеneration 200] Current best fitness = 504999983
            [Genеneration 250] Current best fitness = 504999983
            [Genеneration 300] Current best fitness = 504999983
            [Genеneration 350] Current best fitness = 504999999
            [Genеneration 400] Current best fitness = 504999999
            [Genеneration 450] Current best fitness = 504999999
            [Genеneration 495] Early stop, time limit reached (60.07s)
```

```
== RISULTATI ==
      PLI -> Score: 0, Tempo: 60.0578s
      Genetico -> Score: 504999999, Tempo: 60.0680s
--------------------------------------------------------------------------------
```

4.2 Conclusioni

Riportiamo in una forma più compatta i risultati precedentemente ottenuti.

Dataset	Pizze (N)	Fette (M)	Score PLI	Tempo PLI (s)	Score Genetico	Tempo Genetico (s)
small	10	100	100	0.0085	100	0.1321
medium	50	4500	4500	0.0283	4500	0.1928
big	2000	10^9	10^9	2.1111	999,999,725	16.0980
extra	10,000	505M	0	60.0578	504,999,999	60.0680

Table 4.1: Confronto tra PLI e Algoritmo Genetico su diversi dataset.

Per ciascuno dei test eseguiti, è possibile valutare il parametro di *gap*, che fornisce una misura della qualità della soluzione prodotta dall'algoritmo genetico in termini di errore massimo.

$$gap = \frac{|OPT(i) - EUR(i)|}{|OPT(i)|} \cdot 100$$

Per ciascun istanza i del problema, i valori del parametro vengono riportati nella seguente tabella.

Dataset	OPT	EUR	gap (%)
small	100	100	0.00
medium	4500	4500	0.00
big	10^9	999,999,725	0.0000275
extra	505,000,000	504,999,999	0.000000198

Table 4.2: Parametro *gap* per ciascuna delle soluzioni riportate dall'algoritmo genetico.

Dai test eseguiti sulle istanze di dimensioni crescenti, è possibile trarre alcune considerazioni finali sulle soluzioni con Programmazione Lineare Intera (PLI) e Algoritmo Genetico proposte.

- **Per le istanze di dimensioni ridottre o medie**, entrambi gli approcci trovano rapidamente la soluzione ottima.
  - Nei dataset small e medium (10–50 pizze), la PLI risolve in pochi millisecondi.
  - L'algoritmo genetico converge altrettanto bene, seppur con un tempo leggermente maggiore.

- **Nel caso di istanze grandi (2000 pizze)**, la soluzione con PLI resta ancora efficace.
  - Trova una soluzione ottima in appena 2 secondi.
  - L'algoritmo genetico fornisce una soluzione quasi ottima (con un errore di 275 fette su un miliardo), in circa 16 secondi.
- **Invece, su dataset molto grandi (10.000+ pizze)**, la PLI fallisce completamente.
  - Non restituisce alcuna soluzione entro il tempo limite impostato (60 secondi).
  - L'algoritmo genetico, pur richiedendo anch'esso l'intero tempo disponibile, riesce a costruire una soluzione altamente soddisfacente e prossima al massimo teorico (con un errore di 1 sola fetta su 505 milioni).

Come c'era da aspettarsi dalla teoria, i risultati ottenuti evidenziano come l'approccio genetico sia più robusto e scalabile quando si affrontano problemi su larga scala, mentre la PLI tende addirittura a non convergere entro tempi utili. L'algoritmo genetico, pur essendo più lento su istanze semplici, si dimostra quindi più stabile e adattabile quando la dimensione del problema cresce. In conclusione, in ambienti con vincoli di tempo o risorse computazionali, la metaeuristica genetica si rivela una scelta più affidabile.