UNIVERSITÀ
DI PARMA

Soluzione Randori in ASP

Progetto Programmazione Dichiarativa

Arianna Cipolla - Leopoldo Antozzi
arianna.cipolla@studenti.unipr.it - leopoldo.antozzi@studenti.unipr.it

10 febbraio 2025

Indice

1 Answer Set Programming - ASP

La programmazione può essere affrontata secondo due principali approcci: quello **imperativo** e quello **dichiarativo**. Nel primo, la prerogativa principale è il **come** realizzare un qualcosa e il come sarà descritto attraverso una serie di istruzioni impartite dal programmatore, che, passo dopo passo andranno a modificare lo stato del computer. L'approccio dichiarativo, invece, segue una logica diversa, più basata sul **cosa**, ovvero il programmatore descrive il problema da risolvere e le condizioni che devono essere soddisfatte, senza specificare esplicitamente il modo in cui la soluzione deve essere calcolata.

Un programma ASP è costituito da un insieme di **regole logiche**, ognuna delle quali specifica come gli elementi di un certo dominio sono collegati tra loro. L'esecuzione di un programma ASP non segue una sequenza di operazioni da eseguire, ma consiste nella ricerca di **insiemi di fatti** (chiamati *answer set*) che rispettano tutte le regole imposte.

Questo metodo lo rende particolarmente adatto per problemi **NP-completi**, ovvero problemi in cui trovare una soluzione può essere molto difficile, ma verificarne la correttezza è relativamente semplice. ASP è ideale quando il numero di soluzioni è finito e deve essere esplorato in modo esaustivo.

Parlando di applicazione pratica, possiamo considerare ASP come maggiormente efficace nella risoluzione di **problemi combinatori**, dove è necessario esplorare un ampio numero di combinazioni per individuare quelle che rispettano criteri o vincoli stabiliti.

Altro punto di forza è sicuramente la capacità di gestire **negazione e incertezza**, ovvero:

1. **Negazione**: ASP utilizza due tipi di negazione:
   - **Negazione classica** ($\neg p$): afferma esplicitamente che un fatto non è vero.
   - **Negazione per fallimento** (*not* p): indica che non si può dimostrare che un fatto sia vero, ma senza affermare che sia necessariamente falso.

   Questo permette di modellare situazioni in cui non si ha una conoscenza completa, evitando di dover esplicitare tutte le informazioni.
2. **Incertezza**: In molti problemi reali, alcune informazioni potrebbero essere incomplete o soggette a cambiamenti. ASP gestisce questa situazione grazie alla sua **semantica non-monotona**, che data l'aggiunta di nuove informazioni può cambiare le conclusioni precedenti. Questo lo rende particolarmente utile in contesti in cui le condizioni possono variare o in cui non è possibile conoscere tutti i dettagli a priori.

E' proprio grazie ai **solver ASP**, che identificano automaticamente le soluzioni valide, e ci permettono di concentrarci sulla modellazione del problema attraverso regole e vincoli, senza dover direttamente sviluppare algoritmi per la sua risoluzione.

2 Introduzione al problema

Il problema che vogliamo risolvere riguarda la pianificazione degli allenamenti settimanali in una palestra di Judo, che devono tenere conto di alcuni vincoli, tra cui: la dimensione del tatami sul quale gli atleti possono allenarsi e fare randori, gli orari disponibili per gli allenamenti, assegnabili ai vari gruppi e categoria di peso e cintura. La palestra dispone di un *tatami* con una capienza limitata, che consente l'allenamento simultaneo di un massimo di:

- k agonisti adulti.
- $\frac{3}{2}k$ agonisti ragazzi.
- 2k non agonisti (bambini).

I gruppi di atleti sono distinti e non possono mischiarsi tra loro.

Gli allenamenti possono iniziare dalle 16:00 dal lunedì al venerdì, con tre turni da due ore ciascuno, mentre il sabato mattina è riservato esclusivamente agli agonisti adulti e prevede due turni. Inoltre, il maestro impone che, se viene svolto almeno un turno in una giornata, allora debbano essercene almeno due, preferibilmente tre (eccetto il sabato, dove due turni sono accettabili). Non possono esserci turni intermedi vuoti.
Gli atleti sono suddivisi in tre categorie:

- **Agonisti adulti** (a1, ..., am)
- **Agonisti ragazzi** (r1, ..., rn)
- **Non agonisti (bambini)** (b1, ..., bp)

Come detto in precedenza ogni atleta ha una **categoria di peso w** e una **cintura c**, che determinano le modalità con cui gli allenamenti devono essere organizzati per garantire incontri di pratica (randori) adeguati.

2.1 Interpretazione dei vincoli

Randori con atleti della stessa categoria di peso o del peso immediatamente inferiore

Ogni atleta dovrebbe avere la possibilità di fare randori almeno una volta a settimana con tutti gli altri atleti che appartengono alla sua stessa categoria di peso, o eventualmente con quelli della categoria immediatamente inferiore.

Randori con atleti della stessa categoria di peso e con la stessa cintura o cintura immediatamente inferiore

Ogni atleta deve poter effettuare randori almeno due volte a settimana con avversari che appartengono alla stessa categoria di peso e che hanno la stessa cintura o quella immediatamente inferiore.

Frequenza minima degli allenamenti

Ad ogni gruppo è indispensabile che vengano assegnati un numero minimo di allenamenti che dovranno essere organizzati durante la settimana, i quali corrispondono a:

- **Agonisti adulti** devono allenarsi almeno **4 volte a settimana**.
- **Agonisti ragazzi** devono allenarsi almeno **3 volte a settimana**.
- **Non agonisti**, ovvero i bambini devono allenarsi almeno **2 volte a settimana**.

Riposo obbligatorio per i non agonisti

I bambini devono avere almeno un giorno di riposo tra due allenamenti.

Gestione disponibilità del sabato mattina

Se il numero di agonisti adulti supera la capienza del tatami il sabato, coloro che non trovano posto devono allenarsi obbligatoriamente il venerdì.

Restrizioni sui turni e continuità degli allenamenti

Solo gli agonisti adulti hanno la possibilità di allenarsi durante il terzo turno. Inoltre, se si decide di iniziare gli allenamenti in un determinato giorno, è necessario che vengano programmati almeno due turni. Non è consentito lasciare turni intermedi vuoti: se il secondo turno è già occupato, allora devono essere utilizzati anche il primo o il terzo turno.

3 Soluzione applicata al codice

La soluzione sviluppata prevede un'integrazione tra il linguaggio C++ ed il linguaggio ASP.
Questo perché C++ può essere sfruttato per la generazione dinamica degli input, che saranno poi gestiti da ASP, che, secondo i vincoli stabiliti, andrà ad allocare gli atleti nelle varie sessioni di allenamento.

3.1 C++

Questo codice rappresenta il nucleo centrale del progetto, in quanto gestisce l'intero processo di organizzazione dei dati. Non solo genera dinamicamente l'input in base alle direttive fornite dall'utente, ma si occupa anche della creazione del file di input per il programma ASP. Inoltre, è sempre il codice C++ a eseguire il solver ASP, che elabora i dati contenuti nei file **inputASP/input_*.asp**, gestisce i valori di input e calcola gli answer set. Infine, i risultati ottenuti vengono raccolti e salvati nei file testuali, **outputASP/output_*.txt**.

Inizialmente il codice definisce una struttura che rappresenta un atleta, specificando attributi come: l'identificativo univoco(*id*), il gruppo di appartenenza(*group*), la cintura(*belt*) e il peso(*weigth*).

```
struct athlete {
    string id;
    string group;
    string belt;
    double weight;
};
```

Ogni atleta avrà quindi degli attributi che saranno inizializzati. Il colore della cintura(*belt*) viene assegnato in modo casuale attraverso la funzione *getRandomBelt()*, che sceglie tra sette possibili colori(bianco, giallo, arancione, verde, blu, marrone, nero), i quali saranno mappati in ordine di importanza dalla struttura mappa *beltRanking*, dove ad ogni colore della cintura viene associato un valore numerico. Questa operazione viene effettuata perché quando si lavora con asp, l'uso della cintura come valore numerico rende l'informazione più facilmente gestibile.

```
map<string, int> beltRanking = {
        {"bianca", 1},
        {"gialla", 2},
        {"arancione", 3},
        {"verde", 4},
        {"blu", 5},
        {"marrone", 6},
        {"nera", 7}
};
```

Il valore della proprietà peso(*weigth*) è anch'esso casuale, generato con la funzione *getRandomWeight(group)* ma con un intervallo differente a seconda del parametro *group*:

- Gli adulti tra 50 e 100 kg.
- I ragazzi tra 40 e 70 kg.
- I bambini tra 10 e 40 kg.

Questi dati casuali contribuiscono a simulare una popolazione di atleti realistici, che saranno poi organizzati in sessioni di allenamento.

Per rappresentare un allenamento è stata definita la struttura **training**, che consente di memorizzare: giorno della settimana in cui l'allenamento si svolge(*day*), fascia oraria dell'allenamento(*time*) e l'insieme degli atleti che partecipano alla sessione(*participants*). L'insieme a cui facciamo riferimento è composto solo da valori univoci, di conseguenza, lo stesso atleta non potrà essere registrato più volte allo stesso allenamento.

```
struct training{
    int day;
    int time;
    set<athlete> participants;
};
```

La parte di incontri pratici è organizzata attraverso la struttura *randori*, attraverso la quale sono rappresentate le sessioni di combattimento tra coppie di atleti, attraverso la definizione delle seguenti proprietà: *day* che fa riferimento al giorno della settimana in cui si svolge l'incontro, *time* che indica la fascia oraria dell'incontro e *participantsCouple* che definisce l'insieme di elementi univoci, dove ogni elemento è una coppia di atleti che si sfiderà.

```
struct randori {
    set<pair<athlete, athlete>> participantsCouple;
    int day;
    int time;
};
```

Dopo aver definito alcune delle strutture dati fondamentali del programma, è necessario studiare ciò che avviene nel main, che si occuperà lato pratico delle funzioni più importanti del progetto.
Nel *main*, vengono inizializzati alcuni vettori fondamentali per la creazione dei fatti in ASP. Questi vettori contengono le seguenti definizioni: giorni della settimana, le fasce orarie disponibili per gli allenamenti e i gruppi di atleti coinvolti.
Il primo vettore definito è *days*, che contiene i giorni della settimana in cui possono svolgersi gli allenamenti:

```
vector<string> days = {"lun", "mar", "mer", "gio", "ven", "sab"};
```

Successivamente, vengono definiti due vettori per rappresentare gli orari disponibili per gli allenamenti. Il primo vettore, *afternoonHours*, contiene le ore pomeridiane in cui è possibile allenarsi, ovvero le 16:00, le 18:00 e le 20:00:

```
vector<int> afternoonHours = {16, 18, 20};
```

Il vettore *morningHours* memorizza gli orari disponibili per gli allenamenti del sabato mattina, che si svolgono alle 8:00 e alle 10:00:

```
vector<int> morningHours = {8, 10};
```

Infine, il vettore *group* elenca le diverse categorie di atleti che partecipano agli allenamenti.

```
vector<string> group = {"adulti", "ragazzi", "bambini"};
```

Fondamentale nel progetto è anche il valore K, ovvero la variabile che indica la capienza del tatami, sulla cui base saranno calcolati il numero massimo di iscritti per ogni corso, per evitare che si verifichi un sovraffollamento che non permetta poi al solver di asp di trovare soluzioni. Per calcolare il numero di iscritti massimo per ogni gruppo è anche necessario sapere il rapporto tra la grandezza del tatami e quanti atleti per ogni categoria riusciranno ad allenarsi. Il rapporto per ogni gruppo è definito come di seguito:

Partendo dal fatto che un tatami ha dimensione k, ovvero k rappresenta la capienza del tatami, il numero di adulti che si possono allenare in contemporanea su di un tatami è pari a k.

```
int numSpaceTatamiA = k;
```

Per i ragazzi invece il rapporto è $\frac{3}{2}k$.

```
int numSpaceTatamiT = (3 * k) / 2;
```

Mentre il rapporto tra dimensione del tatami k e bambini che vi si possono allenare è $2k$.

```
int numSpaceTatamiC = 2 * k;
```

Avendo ora chiari i rapporti tra gruppo e numero di persone che si possono allenare in base alla capienza del tatami, per evitare che si verifichino casi di sovrapposizione per cui poi ASP non sarà più in grado di generare soluzioni, dobbiamo assicurarci che il numero di atleti per ogni categoria rimanga sotto una soglia massima stabilita.

La soglia massima per ciascuna categoria è rappresentata dalle variabili *numMaxA*, *numMaxT* e *numMaxC*, in cui i valori 7, 5 e 3 utilizzati nei rispettivi calcoli derivano direttamente dal numero massimo di allenamenti disponibili per ciascun gruppo durante la settimana, rispettando i vincoli imposti.

```
int numMaxA = (7 * numSpaceTatamiA) / 4;
int numMaxT = (5 * numSpaceTatamiT) / 3;
int numMaxC = (3 * numSpaceTatamiC) / 2;
```

Per determinare il numero di iscritti a ciascun corso, si utilizza un'assegnazione casuale che garantisce che il valore sia compreso tra la capienza minima del tatami e il numero massimo di iscritti previsti per quella categoria. Il valore iniziale per ogni gruppo (*numSpaceTatamiA, numSpaceTatamiT, numSpaceTatamiC*), rappresenta la capienza minima del tatami per quella categoria. A questo valore viene aggiunto un valore casuale, generato tramite *rand()*. L'operazione modulo si assicura che il valore casuale rientri nell'intervallo compreso tra 0 e la differenza tra il massimo numero di iscritti consentito (*numMaxA, numMaxT, numMaxC*) e la capienza minima.

```
int numAdults = numSpaceTatamiA + rand() % (numMaxA - numSpaceTatamiA + 1);
int numTeens = numSpaceTatamiT + rand() % (numMaxT - numSpaceTatamiT + 1);
int numChildren = numSpaceTatamiC + rand() % (numMaxC - numSpaceTatamiC + 1);
```

Il codice che segue è finalizzato alla generazione di 100 diverse combinazioni di input, ognuna delle quali conterrà un numero di atleti diverso ed anche con caratteristiche degli atleti stessi diverse(il contenuto specifico di questi file .asp viene approfondito nel capitolo [Input ASP generati in C++, 3.1.1]).

Prima dell'esecuzione del ciclo di generazione delle combinazioni, è necessario definire la variabile *startTotale*, la quale riceverà il momento esatto in cui viene eseguita l'istruzione, che in seguito servirà per verificare il tempo totale che il ciclo avrà impiegato per la generazione di tutte le combinazioni.

```
auto startTotale = chrono::high_resolution_clock::now();
```

Il codice prosegue con la generazione di un un ciclo *for* che ripete l'operazione di generazione delle combinazioni dei dati 100 volte. Ad ogni iterazione viene dichiarato un vettore *dataset*, formato da elementi *athlete*(atleti).

```
for (int combination = 0; combination < 100; ++combination) {
    vector<athlete> dataset;
```

A questo punto ogni blocco si occuperà della generazione di un *atleta*, assegnandogli un ID univoco ("a1", "a2", ecc.), una cintura casuale generata dalla funzione *getRandomBelt()* e un peso casuale, generato da *getRandomWeight(gruppo)* in un range di valori stabiliti dal gruppo di appartenenza.

```
for (int i = 0; i < numAdults; ++i) {
    dataset.push_back({"a" + to_string(i + 1), "adulti", getRandomBelt(),
        getRandomWeight("adulti")});
}

for (int i = 0; i < numTeens; ++i) {
    dataset.push_back({"r" + to_string(i + 1), "ragazzi", getRandomBelt(),
        getRandomWeight("ragazzi")});
}

for (int i = 0; i < numChildren; ++i) {
    dataset.push_back({"b" + to_string(i + 1), "bambini", getRandomBelt(),
        getRandomWeight("bambini")});
}
```

Una volta generato il dataset per ogni combinazione, il codice salva i dati in un file CSV tramite la funzione *generateCSV()*. I dati vengono scritti nella directory **combinazioniCSV/**, separati per ciascuna combinazione:

```
generateCSV(combination + 1, dataset);
```

A questo punto il codice prepara un'altra variabile *start*, che ha il compito di tenere traccia il tempo delle operazioni compiute da ASP per generare una soluzione a una sola combinazione.

```
auto start = chrono::high_resolution_clock::now();
```

Dopodiché viene chiamata la funzione *generateASPInput()*, che riceve come argomenti vari parametri legati ai giorni della settimana, orari delle sessioni e gruppi di atleti. Questo crea un file *input_N.asp* per ogni combinazione, che contiene la rappresentazione formale delle informazioni necessarie per Clingo. I file vengono generati nella directory **inputASP/**.

```
generateASPInput(k, combination + 1, days, afternoonHours, morningHours, group,
    dataset, n_training_min_adults, n_training_min_teens, n_training_min_children);
```

Una volta creato il file di input, clingo, un solver ASP, utilizzando il comando di sistema *system()*, invia il file di input a clingo che elabora la distribuzione degli allenamenti. Inoltre, Il comando specifica un tempo massimo di 300 secondi e richiede di trovare una sola soluzione ottimale (indicato dal parametro *-n 1*) per la distribuzione degli allenamenti. Il risultato dell'esecuzione viene salvato in un file di output denominato *output_N.txt*:

```
string outputFileName = "outputASP/output_" + to_string(combination + 1) + ".txt";
string inputFileName = "inputASP/input_" + to_string(combination + 1) + ".asp";

string command = "clingo␣--time-limit=300␣-n␣1␣" + inputFileName + "␣./randori.asp␣>␣"
     + outputFileName;
system(command.c_str());
```

Una volta completata l'esecuzione di Clingo, il codice cronometra il tempo impiegato per risolvere ogni combinazione.

```
auto end = chrono::high_resolution_clock::now();
chrono::duration<double> elapsed = end - start;
```

I risultati generati e salvati in formato testuale però presentano un'organizzazione poco chiara. Per questo motivo, il programma utilizza vettori e funzioni per riorganizzare i dati, facilitandone la lettura. La funzione *parseTraning()* analizza il file di output generato e ne estrae le informazioni relative agli allenamenti, elencando i partecipanti e i possibili incontri di Randori in base al peso e alla cintura degli atleti.
La funzione *writeSortedOutput()* organizza e scrive i dati in un nuovo file nella directory *outputOrdinato/*, fornendo un riepilogo strutturato e facilmente leggibile degli allenamenti e degli incontri.

```
vector<training> vTraining;
vector<randori> vRandoriPeso;
vector<randori> vRandoriCintura;

string outputTidyFileName = "outputOrdinato/output_ordinato_" + to_string(combination
    + 1) + ".txt";

parseTraning(outputFileName, vTraining, vRandoriPeso, vRandoriCintura);

writeSortedOutput(vTraining, vRandoriPeso, vRandoriCintura, outputTidyFileName);
```

Infine, una volta completate tutte le 100 combinazioni, il programma misura il tempo totale impiegato per elaborare tutte le soluzioni, compreso il tempo di: generazione dei dataset, creazione dei file ASP e l'elaborazione delle soluzioni con clingo attraverso randori.asp.

```
auto endTotale = chrono::high_resolution_clock::now();
chrono::duration<double> timeTotal = endTotale - startTotale;

cout << "Tempo␣totale␣di␣esecuzione:␣" << timeTotal.count() << endl;
```

3.1.1 Input ASP generati in C++

Ogni file **inputASP/input_*.asp** prodotto avrà lo stesso schema degli altri, ciò in cui differirà sarà il contenuto, che può essere descritto attraverso i seguenti blocchi principali. E' importante sottolineare che ogni elemento presente nei file di input è un *fatto*, ovvero una dichiarazione che esprime un valore di verità certa.

Definizione dei limiti e dei vincoli

- *max_partecipanti(N, Categoria)*: Indica il numero massimo di atleti ammissibili ad un allenamento per ogni categoria (adulti, ragazzi, bambini).

```
max_partecipanti(10, adulti).
max_partecipanti(15, ragazzi).
max_partecipanti(20, bambini).
```

- *min_allenamenti(N, Categoria)*: Specifica il numero minimo di allenamenti settimanali richiesti per ciascun gruppo. Ad esempio, per gli adulti sono richiesti almeno quattro allenamenti settimanali, per i ragazzi tre e per i bambini due.

```
min_allenamenti(4, adulti).
min_allenamenti(3, ragazzi).
min_allenamenti(2, bambini).
```

Dati relativi alla programmazione degli allenamenti

- *giorno(N)*: Elenca i giorni della settimana in cui è possibile programmare gli allenamenti. Il giorno è specificato attraverso il suo corrispettivo numerico.

```
giorno(1).
giorno(2).
giorno(3).
```

- *gruppo(Categoria)*: Definisce le categorie esistenti, suddividendo gli atleti in *adulti, ragazzi e bambini*.

```
gruppo(adulti).
gruppo(ragazzi).
gruppo(bambini).
```

- *orario_pom(Ora)* e *orario_matt(Ora)*: specificano le fasce orarie disponibili, sia per la mattina che per il pomeriggio.

```
orario_pom(16).
orario_pom(18).
orario_pom(20).


orario_matt(8).
orario_matt(10).
```

Informazioni sugli atleti iscritti

- *atleta(ID, Categoria, Cintura, Peso)*: definisce ogni atleta con un identificativo univoco (*ID*), la categoria di appartenenza, il numero della categoria a cui appartiene la sua cintura e il suo peso corporeo.

```
atleta(a1, adulti, 1, 86).
atleta(a2, adulti, 3, 56).
atleta(a3, adulti, 3, 54).
```

3.2 ASP

Una volta generati i file di input, il codice ASP entra in azione per ottimizzare la distribuzione degli atleti negli allenamenti. L'esecuzione avviene tramite il risolutore *clingo*, configurato con l'opzione *-n 1* per produrre un'unica soluzione (answer set) per ogni input. Ciascun file viene elaborato singolarmente da *randori.asp*, che definisce i vincoli e le regole per l'allocazione degli atleti nelle sessioni di allenamento.

Per ogni input viene generato un output corrispondente, contenente la soluzione ottimale per organizzare i randori settimanali nel rispetto di tutti i vincoli imposti. Infine, grazie alla gestione in C++, i risultati vengono salvati come file testuali nella directory *outputASP/*, con un identificativo nel nome che rimanda al file di input elaborato(ad esempio a input1.asp corrisponderà output1.txt).

Oltre ai vincoli che saranno successivamente descritti, vi sono ulteriori restrizioni di base imposte dalla palestra riguardanti la suddivisione degli allenamenti nei giorni della settimana e la distribuzione degli atleti nei diversi turni di allenamento.

- Un primo vincolo impone che gli allenamenti del sabato mattina siano riservati esclusivamente agli adulti. Questo viene garantito definendo che se si tiene un allenamento di sabato (giorno 6), esso deve avvenire la mattina, coinvolgendo esclusivamente il gruppo degli adulti.

  ```
  allenamento(6, Orario, adulti) :-
      orario_matt(Orario).
  ```

  Per gli altri giorni della settimana (dal lunedì al venerdì), viene stabilito che dev'esserci almeno un allenamento per ogni gruppo nell'orario pomeridiano, vediamo come viene scritta la regola:

  ```
  1 {allenamento(Giorno, Orario, Gruppo) : gruppo(Gruppo)} :-
      giorno(Giorno),
      orario_pom(Orario),
      Giorno != 6.
  ```

  Ma questo potrebbe portare a una sovrapposizione degli allenamenti di diversi gruppi negli stessi orari, dunque viene aggiunto il vincolo seguente, in cui si dice che non possono esserci due gruppi diversi che si allenano nello stesso orario nello stesso giorno.

  ```
  :- allenamento(Giorno, Orario, Gruppo1), allenamento(Giorno, Orario, Gruppo2),
     Gruppo1 != Gruppo2
  ```

  Infine, per gestire la relazione tra gli allenamenti programmati e la presenza degli atleti, si introduce una regola che associa un atleta a un determinato allenamento solo se questo è stato programmato per il gruppo di appartenenza dell'atleta stesso. Il predicato *allenamento_con_presenze* stabilisce quindi che un atleta *Id* può essere presente a un allenamento in un certo giorno e orario solo se esiste un allenamento per il suo gruppo in quella fascia oraria.

  ```
  allenamento_con_presenze(Giorno, Orario, Gruppo, Id) :-
      allenamento(Giorno, Orario, Gruppo), presente(Id, Giorno, Orario).
  ```

Ora vediamo l'implementazione e la spiegazione di come i vincoli descritti nel capitolo [Interpretazione dei vincoli, 2.1] sono stati creati:

- Il primo vincolo richiesto era che all'interno di ogni gruppo, ogni settimana ciascuno deve poter fare randori con tutti quelli della stessa categoria di peso o del peso immediatamente inferiore. Per la creazione di questo vincolo è necessario definire due regole, *compatibile_peso_per_randori(Id1, Peso, Id2, Peso)*, le quali sono scritte in modo da rappresentare un OR logico, nel senso che basta che sia vera una delle 2 definizioni di questa regola per fare in modo che poi nel vincolo la regola venga considerata come vera.

  La 1° definizione indica che la regola sarà considerata vera se gli atleti appartengono allo stesso *Gruppo* e se entrambi hanno esattamente lo stesso *Peso*. Inoltre, sarà specificato che **Id1<Id2** perché consente di evitare nello stesso allenamento la ripetizione dello scontro tra i due atleti organizzati in modo inverso.

  ```
  compatibile_peso_per_randori(Id1, Peso, Id2, Peso) :-
      atleta(Id1, Gruppo, _, Peso),
      atleta(Id2, Gruppo, _, Peso),
      Id1 < Id2.
  ```

Questa 2° definizione della regola invece fa riferimento al fatto che gli atleti possono fare randori anche con la categoria di peso immediatamente inferiore, sarà quindi considerata vera quando: gli atleti saranno diversi tra loro, apparterranno allo stesso gruppo e se la differenza di peso tra i due atleti sarà di esattamente 1 unità.

```
compatibile_peso_per_randori(Id1, Peso1, Id2, Peso2) :-
    atleta(Id1, Gruppo, _, Peso1),
    atleta(Id2, Gruppo, _, Peso1 + 1),
    Id1 != Id2,
    Peso2 = Peso1 + 1.
```

Dopo aver definito le due versioni delle regole, andiamo a definire il vincolo che stabilirà quali combinazioni saranno considerate vere e che quindi saranno inserite nell'answer set e quali invece saranno false. Il vincolo andrà quindi ad imporre che ogni coppia di atleti compatibili per peso debba avere almeno un allenamento in comune nella settimana per potersi sfidare. Questo viene fatto grazie a *#count { Giorno, Orario : presente(Id1, Giorno, Orario), presente(Id2, Giorno, Orario) } = 0* che conta il numero di occasioni in cui i due atleti Id1 e Id2 si trovano nello stesso allenamento. Se il risultato sarà 0 significa che i 2 atleti non avranno alcun giorno in comune.

```
:- compatibile_peso_per_randori(Id1, Peso1, Id2, Peso2),
   #count { Giorno, Orario : presente(Id1, Giorno, Orario),
                             presente(Id2, Giorno, Orario) } = 0.
```

Per concludere viene definita la regola *randori_in_allenamento_peso(Id1, Peso1, Id2, Peso2, Giorno, Orario)* che ha lo scopo di raggruppare tutte le possibili combinazioni trovate, per poi stamparle attraverso *#show randori_in_allenamento_peso/6.* Nella stampa rientreranno quindi gli atleti che saranno compatibili per peso, presenti allo stesso allenamento e che effettivamente faranno randori.

```
randori_in_allenamento_peso(Id1, Peso1, Id2, Peso2, Giorno, Orario) :-
    compatibile_peso_per_randori(Id1, Peso1, Id2, Peso2),
    presente(Id1, Giorno, Orario),
    presente(Id2, Giorno, Orario),
    Id1 != Id2.
```

- Altro vincolo richiesto è che all'interno di ogni gruppo, ogni settimana ciascuno deve poter fare randori due giorni con tutti quelli della sua categoria di peso e della sua cintura o della cintura immediatamente inferiore. Per la creazione di questo vincolo la regola *compatibile_cintura_per_randori(Id1, Cintura, Id2, Cintura)* viene definita in due versioni differenti come nel caso precedente. L'unica differenza è che non viene confrontato la differenza di peso, ma l'oggetto di controllo sarà la cintura.

  Nella regola seguente controlliamo che la cintura sia della stessa categoria.

```
compatibile_cintura_per_randori(Id1, Cintura, Id2, Cintura) :-
    atleta(Id1, Gruppo, Cintura, Peso),
    atleta(Id2, Gruppo, Cintura, Peso),
    Id1 < Id2.
```

  Mentre in questa ridefinizione successiva della regola abbiamo il controllo della categoria della cintura nel caso in cui sia inferiore di uno.

```
compatibile_cintura_per_randori(Id1, Cintura1, Id2, Cintura2) :-
    atleta(Id1, Gruppo, Cintura1, Peso),
    atleta(Id2, Gruppo, Cintura1 + 1, Peso),
    Id1 != Id2,
    Cintura2 = Cintura1 + 1.
```

Il vincolo andrà quindi ad imporre che ogni coppia di atleti compatibili per cintura debba avere almeno due allenamenti in comune nella settimana per potersi sfidare. Questo viene fatto grazie a *#count { Giorno, Orario : presente(Id1, Giorno, Orario), presente(Id2, Giorno, Orario) }<2.*

```
:- compatibile_cintura_per_randori(Id1, Cintura1, Id2, Cintura2),
    #count {Giorno, Orario : presente(Id1, Giorno, Orario),
                             presente(Id2, Giorno, Orario)} < 2.
```

Per concludere viene definita la regola *randori_in_allenamento_cintura(Id1, Cintura1, Peso1, Id2, Cintura2, Peso2, Giorno, Orario)* che ha lo scopo di raggruppare tutte le possibili combinazioni trovate, per poi stamparle attraverso *#show randori_in_allenamento_cintura/8*. Nella stampa rientreranno quindi gli atleti che saranno compatibili per peso, cintura, presenti allo stesso allenamento e che effettivamente possono fare randori.

```
randori_in_allenamento_cintura(Id1, Cintura1, Peso1,
                               Id2, Cintura2, Peso2, Giorno, Orario) :-
    compatibile_cintura_per_randori(Id1, Cintura1, Id2, Cintura2),
    presente(Id1, Giorno, Orario),
    presente(Id2, Giorno, Orario),
    atleta(Id1, _, _, Peso1),
    atleta(Id2, _, _, Peso2),
    Id1 != Id2.
```

- Come stabilito è necessario che ci sia un numero minimo di allenamenti a settimana, che corrispondono a 4 per gli agonisti adulti, 3 per gli agonisti ragazzi e 2 per i non agonisti ovvero i bambini.

  Questo vincolo è implementabile partendo dalla definizione di una regola che afferma che: per ogni atleta *Id* appartenente a un determinato *Gruppo*, deve esserci un numero *N* di allenamenti in cui è presente. La condizione di presenza è espressa da *presente(Id, Giorno, Orario)* che indica che l'atleta *Id* partecipa a un allenamento che si svolge in un determinato giorno e orario. La lista di allenamenti *allenamento(Giorno, Orario, Gruppo)* si riferisce a tutti gli allenamenti programmati per un determinato *Gruppo* in un dato giorno e orario.

  La parte successiva della regola *min_allenamenti(N, Gruppo)* stabilisce che il numero minimo di allenamenti *N* a cui ogni atleta deve partecipare dipende dal *Gruppo* a cui l'atleta appartiene. La variabile *N* è quindi vincolata al gruppo dell'atleta, e il fatto *min_allenamenti* definisce quanti allenamenti devono essere presenti per ciascun gruppo specifico.

  ```
  N {presente(Id, Giorno, Orario) : allenamento(Giorno, Orario, Gruppo)} :-
  atleta(Id, Gruppo, _, _), min_allenamenti(N, Gruppo).
  ```

- Per garantire che i bambini non si allenino in giorni consecutivi, è stato introdotto un vincolo che impedisce la programmazione di allenamenti per il gruppo *bambini* in due giorni consecutivi. Per fare ciò, è stata necessaria la definizione di alcune regole che permettono di stabilire quando due giorni sono considerati consecutivi.

  La prima regola definisce la relazione *a_destra(Giorno, Giorno + 1)*, che specifica che un giorno è immediatamente successivo a un altro. La condizione *giorno(Giorno), giorno(Giorno + 1)* assicura che entrambi i valori rappresentino giorni validi del calendario settimanale.

  ```
  a_destra(Giorno, Giorno + 1) :- giorno(Giorno), giorno(Giorno + 1).
  ```

  La seconda regola *a_sinistra(Giorno + 1, Giorno)*, invece, stabilisce che un giorno è immediatamente precedente a un altro. Anche in questo caso, la condizione *giorno(Giorno), giorno(Giorno + 1)* garantisce che i valori siano giorni validi.

  ```
  a_sinistra(Giorno + 1, Giorno) :- giorno(Giorno), giorno(Giorno + 1).
  ```

Successivamente, viene definita la relazione *vicino(Giorno1, Giorno2)*, che indica che due giorni sono adiacenti. Questa relazione viene costruita sfruttando le due regole precedenti: un giorno può essere considerato vicino a un altro se si trova immediatamente alla sua destra o alla sua sinistra. Di conseguenza, la relazione sarà considerata vera in questi casi.

```
vicino(Giorno1, Giorno2) :- a_destra(Giorno1, Giorno2).
vicino(Giorno1, Giorno2) :- a_sinistra(Giorno1, Giorno2).
```

Infine, viene introdotto il vincolo che impedisce ai bambini di allenarsi in giorni consecutivi. Il vincolo utilizza la relazione *vicino(Giorno1, Giorno2)*, che assicura che due giorni siano consecutivi. La condizione *allenamento(Giorno1, _, bambini)* verifica che sia stato programmato un allenamento per il gruppo *bambini* nel giorno *Giorno1*, mentre *allenamento(Giorno2, _, bambini)* controlla la presenza di un altro allenamento per lo stesso gruppo nel giorno *Giorno2*. Se entrambi gli allenamenti esistono e i giorni sono consecutivi, il vincolo viene violato, e quindi questa configurazione non verrà inclusa nell'answer set.

```
:- allenamento(Giorno1, _, bambini), allenamento(Giorno2, _, bambini),
   vicino(Giorno1, Giorno2).
```

- Per garantire che gli atleti adulti abbiano la possibilità di allenarsi nel caso in cui il numero massimo di partecipanti agli allenamenti del sabato mattina venga raggiunto, è stato introdotto un vincolo che impone agli atleti che non trovano posto il sabato di allenarsi il venerdì pomeriggio. Prima di definire il vincolo, vengono introdotte delle regole ausiliarie per verificare la saturazione dell'allenamento del sabato mattina.

  La prima regola introduce il predicato *pieno_sabato*, che stabilisce quando il numero massimo di partecipanti consentiti il sabato mattina è stato raggiunto. Questo avviene utilizzando l'aggregato $\#count\{Id : presente(Id, 6, Orario), orario\_matt(Orario)\} \geq K$, che conta il numero di atleti presenti negli allenamenti del sabato mattina. Il valore massimo di partecipanti ammessi è determinato da *max_partecipanti(K, adulti)*, che associa il numero massimo di posti disponibili al gruppo degli adulti.

```
pieno_sabato :-
    #count{Id : presente(Id, 6, Orario), orario_matt(Orario)} >= K,
    max_partecipanti(K, adulti).
```

Una volta stabilita la condizione di saturazione del sabato mattina, viene definito il vincolo che impone agli atleti senza posto il sabato di allenarsi il venerdì pomeriggio. Questo vincolo viene espresso nel modo seguente: se il sabato mattina è pieno (*pieno_sabato*), un atleta che si allena il venerdì pomeriggio (*presente(Id, 5, Orario), orario_pom(Orario)*) deve obbligatoriamente essere presente anche il sabato mattina (*presente(Id, 6, Orario), orario_matt(Orario)*). L'operatore *not* viene utilizzato per verificare che l'atleta non sia già presente il sabato mattina, nel qual caso il vincolo viene violato e l'assegnazione non sarà inclusa nell'Answer set.

```
:- pieno_sabato,
   presente(Id, 5, Orario), orario_pom(Orario),
   not presente(Id, 6, Orario), orario_matt(Orario).
```

Infine, viene introdotto un ulteriore vincolo che impedisce agli atleti di frequentare più di un allenamento consecutivo il sabato mattina o anche durante la settimana.

```
:- presente(Id, 6, Orario1), presente(Id, 6, Orario2), Orario1 != Orario2,
   orario_matt(Orario1), orario_matt(Orario2).

:- presente(Id, Giorno, Orario1), presente(Id, Giorno, Orario2), Orario1 != Orario2,
   orario_pom(Orario1), orario_pom(Orario2)
```

- Il vincolo da implementare stabilisce che solo gli adulti possono allenarsi al terzo turno (20:00), che non ci devono essere turni intermedi vuoti e che ogni giorno devono esserci almeno due turni occupati, preferibilmente tre (tranne il sabato). Per ottenere questa restrizione, vengono definite diverse regole e vincoli.

  Il primo vincolo assicura che il gruppo dei bambini non possa allenarsi più di una volta al giorno. Questo è implementato verificando la presenza di due allenamenti per i bambini nello stesso giorno con orari diversi (*Orario1 != Orario2*). Se questa condizione è verificata, la soluzione viene scartata.

  ```
  :- allenamento(Giorno, Orario1, bambini), allenamento(Giorno, Orario2, bambini),
     Orario1 != Orario2.
  ```

  Successivamente, viene definito il vincolo che impedisce ai gruppi diversi dagli adulti di allenarsi al terzo turno delle 20:00. Questo è espresso verificando la presenza di un allenamento a quell'ora per un qualsiasi gruppo *Gruppo* diverso da *adulti* e, se la condizione è verificata, la soluzione viene esclusa.

  ```
  :- allenamento(Giorno, 20, Gruppo), Gruppo != adulti.
  ```

  Per garantire che non ci siano turni intermedi vuoti, viene introdotto un vincolo che impedisce la presenza di un allenamento alle 20:00 se non è presente un allenamento anche nel turno precedente delle 18:00

  ```
  :- allenamento(Giorno, 20, _), not allenamento(Giorno, 18, _).
  ```

  Un altro vincolo assicura che ogni giorno ci siano almeno due turni occupati. Questo è ottenuto imponendo la presenza di almeno due istanze di *allenamento(Giorno, Orario, _)* tra tutti gli orari pomeridiani, utilizzando l'aggregato *2 { allenamento(Giorno, Orario, _) : orario_pom(Orario) }*. Se questa condizione non è verificata, la soluzione viene esclusa.

  ```
  :- not 2 { allenamento(Giorno, Orario, _) : orario_pom(Orario) }.
  ```

  Infine, viene definito un ulteriore vincolo che stabilisce che gli atleti possono essere presenti solo agli allenamenti del proprio gruppo e che il numero massimo di partecipanti per allenamento non deve superare il limite K, determinato dal predicato *max_partecipanti(K, Gruppo)*. Questa condizione è implementata utilizzando un aggregato che impone che ogni allenamento abbia almeno 2 e al massimo K atleti.

  ```
  2 { presente(Id, Giorno, Orario) : atleta(Id, Gruppo, _, _) } K
  :- allenamento(Giorno, Orario, Gruppo), max_partecipanti(K, Gruppo).
  ```

4 Test

In questo capitolo analizzeremo i risultati restituiti dal programma ASP, in termini teorici, discuteremo quindi degli answer set. Inoltre, forniremo dati tecnici per analizzare le tempistiche riportate dalle varie configurazioni del problema. [1]

[1] Ovviamente, gli esempi riportati sono solo un estratto del file; per avere il quadro completo della settimana, si consiglia di consultare la directory apposita (outputOrdinato/). Attenzione, dato che il codice è stato rieseguito potremmo trovare diverse configurazioni risultanti.

4.1 Analisi output

Vediamo, come prima cosa, una semplice esecuzione del codice C++, che, come già accennato, genera 100 combinazioni differenti delle caratteristiche degli atleti (peso, cintura) e delle quantità di iscritti a ciascun corso (Adulti, Ragazzi, Bambini). Analizzando il file presente nella directory '*outputOrdinato/output_ordinato_1.txt*':

```
----------------------- giorno LUNEDI alle 16 gruppo RAGAZZI -----------------------
r10, r12, r17, r18, r19, r2, r20, r21, r22, r3, r4, r6, r7, r8, r9
****************** Randori per PESO ******************
|r10 56Kg| VS |r21 56Kg|
|r12 58Kg| VS |r7 58Kg|
|r17 48Kg| VS |r3 49Kg|
|r17 48Kg| VS |r9 48Kg|
|r19 54Kg| VS |r22 54Kg|
|r19 54Kg| VS |r6 55Kg|
|r2 64Kg| VS |r20 65Kg|
|r2 64Kg| VS |r4 64Kg|
|r22 54Kg| VS |r6 55Kg|
|r4 64Kg| VS |r20 65Kg|
|r6 55Kg| VS |r10 56Kg|
|r6 55Kg| VS |r21 56Kg|
|r9 48Kg| VS |r3 49Kg|

****************** Randori per PESO e CINTURA ********
|r19 54Kg blu| VS |r22 54Kg blu|
|r21 56Kg arancione| VS |r10 56Kg verde|
```

Possiamo osservare che, ad esempio, il lunedì alle 16 è programmato un allenamento per i ragazzi, durante il quale parteciperanno gli atleti elencati. Inoltre, per ogni allenamento, è indicato chi potrebbe avere la possibilità di fare randori. Quelli compatibili per peso devono avere una differenza massima di un chilogrammo. Successivamente, troviamo la lista degli atleti che possono fare randori in base al peso e alla cintura. Come spiegato nei capitoli precedenti, possono scontrarsi solo atleti con lo stesso peso e con una cintura della stessa categoria o, al massimo, di una categoria inferiore.

Nell'esempio precedente è stata usata una grandezza del tatami k pari a 10. Poiché per i ragazzi la grandezza era $\frac{3}{2}k$, il massimo di atleti per allenamento è 15. Per coprire tutte le necessità, il numero massimo di iscritti al corso dei ragazzi è pari a $(5 \cdot \frac{3k}{2})/3 = 25$. Questo conferma che tutti i vincoli sono stati rispettati, poiché il numero di ragazzi presenti all'allenamento è 15, mentre il numero massimo di iscritti al corso è 22.

```
----------------------- giorno MERCOLEDI alle 20 gruppo ADULTI -----------------------
a1, a10, a12, a13, a4, a5, a6, a7, a8, a9
****************** Randori per PESO ******************
|a1 73Kg| VS |a10 73Kg|
|a1 73Kg| VS |a12 74Kg|
|a10 73Kg| VS |a12 74Kg|
|a5 54Kg| VS |a13 55Kg|
```

In questo estratto di codice, riportato qui sopra, sempre tratto dallo stesso file, possiamo osservare che il terzo turno della giornata, stabilito alle 20:00, è dedicato esclusivamente agli adulti.

Per quanto riguarda i non agonisti (bambini), possiamo notare nell'output successivo, che hanno almeno un giorno di riposo tra gli allenamenti (casualmente, l'orario di tutti gli allenamenti è risultato essere alle 18:00).

```
----------------------- giorno LUNEDI alle 18 gruppo BAMBINI -----------------------
b1, b10, b11, b12, b13, b14, b15, b16, b17, b18, b2, b20, b22, b23, b24, b25, b3, b6, b8, b9
```

```
----------------------- giorno MERCOLEDI alle 18 gruppo BAMBINI -----------------------
b1, b10, b11, b12, b13, b14, b15, b16, b17, b18, b19, b2, b21, b3, b4, b5, b6, b7, b8, b9

----------------------- giorno VENERDI alle 18 gruppo BAMBINI -----------------------
b1, b12, b13, b16, b18, b19, b2, b20, b21, b22, b23, b24, b25, b3, b4, b5, b6, b7, b8, b9
```

Mentre per la frequenza degli allenamenti, ogni atleta, a seconda della categoria (adulto, ragazzo, bambino), aveva una frequenza minima da rispettare. Possiamo osservare come questo vincolo venga rispettato nel seguente estratto di output, tratto dallo stesso esempio, in cui l'atleta ***a1*** si allena esattamente quattro volte alla settimana. Notiamo, inoltre, che l'allenamento del sabato mattina è riservato esclusivamente agli adulti.

```
----------------------- giorno MERCOLEDI alle 20 gruppo ADULTI -----------------------
a1, ...

----------------------- giorno GIOVEDI alle 20 gruppo ADULTI -----------------------
a1, ...

----------------------- giorno VENERDI alle 20 gruppo ADULTI -----------------------
a1, ...

----------------------- giorno SABATO alle 8 gruppo ADULTI -----------------------
a1, ...
```

Oppure, nel caso di un atleta della categoria ragazzi, abbiamo almeno tre allenamenti a settimana. Ad esempio, osserviamo ***r5***, che si allena esattamente quattro volte.

```
----------------------- giorno MARTEDI alle 18 gruppo RAGAZZI -----------------------
..., r5, ...

----------------------- giorno MERCOLEDI alle 16 gruppo RAGAZZI -----------------------
..., r5, ...

----------------------- giorno GIOVEDI alle 18 gruppo RAGAZZI -----------------------
..., r5, ...

----------------------- giorno VENERDI alle 16 gruppo RAGAZZI -----------------------
..., r5, ...
```

4.2 Analisi tempi di esecuzione

A questo punto, verificato il corretto funzionamento del programma, non ci resta che analizzarne il comportamento in termini di tempo di esecuzione. Il primo test effettuato ha mantenuto fissa la grandezza del Tatami, variando il numero di iscritti. Possiamo osservare il seguente grafico, in cui il cronometro parte nel momento in cui il programma C++ genera la prima combinazione e si ferma non appena ha trovato tutti gli answer set per tutte le 100 combinazioni.

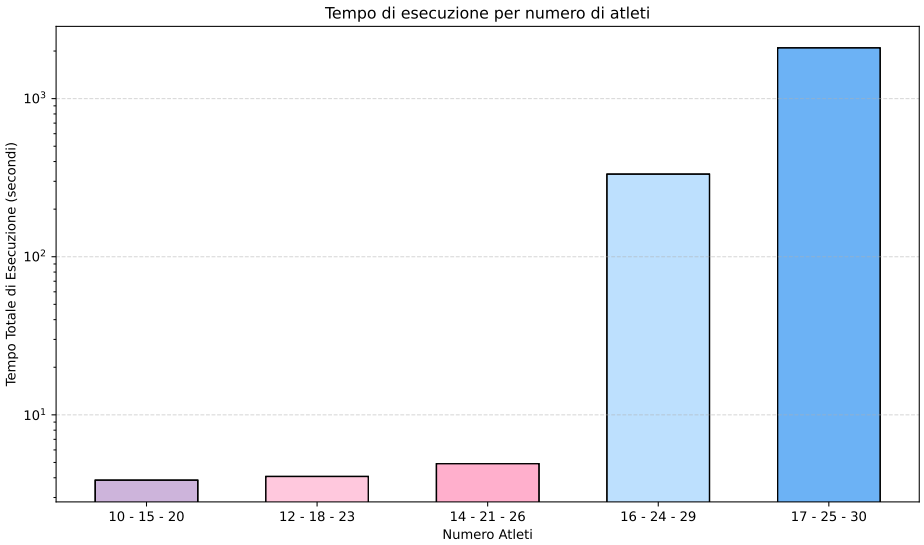

Figura 1: Grafico con diverse grandezze di iscritti e K = 10

Andiamo ad analizzare approfonditamente il risultato. Nell'asse delle ascisse troviamo il numero di atleti per gruppo; ad esempio, nel caso di "10 - 15 - 20", il numero di iscritti al corso degli adulti è pari a 10, al corso dei ragazzi è pari a 15 e al corso dei bambini è pari a 20, e così via per tutti gli altri valori su quell'asse.

Per quanto riguarda l'asse delle ordinate, sono riportati i secondi impiegati dal programma, in scala logaritmica con base 10, per una lettura più chiara dei risultati.

Notiamo che, con k pari a 10, nel primo test (10 - 15 - 20), gli iscritti non superano la capacità di contenere le varie categorie. Di conseguenza, per incastrare tutte le necessità della palestra, il programma ASP non trova difficoltà, concludendo in un tempo di circa 4 secondi per 100 combinazioni diverse.

Man mano che ci avviciniamo al numero massimo di iscritti, già definito, il tempo aumenta. Infatti, nell'ultimo caso testato, in cui abbiamo saturato il numero di iscritti per ogni gruppo, il programma impiega molto più tempo: circa 30 minuti per 100 combinazioni, perché deve incastrare tutte le necessità. In particolare, a rallentare il processo è il fatto che ASP deve garantire un numero minimo di allenamenti per ogni atleta e distribuirli nel miglior modo possibile per soddisfare la richiesta. Inoltre, ASP deve considerare la possibilità di far fare randori a ogni atleta compatibile, organizzando allenamenti comuni quando necessario.

Per assicurare che questa osservazione sia corretta, abbiamo testato lo stesso funzionamento su un Tatami con maggiore capienza (k = 20) e ottenuto il seguente grafico:

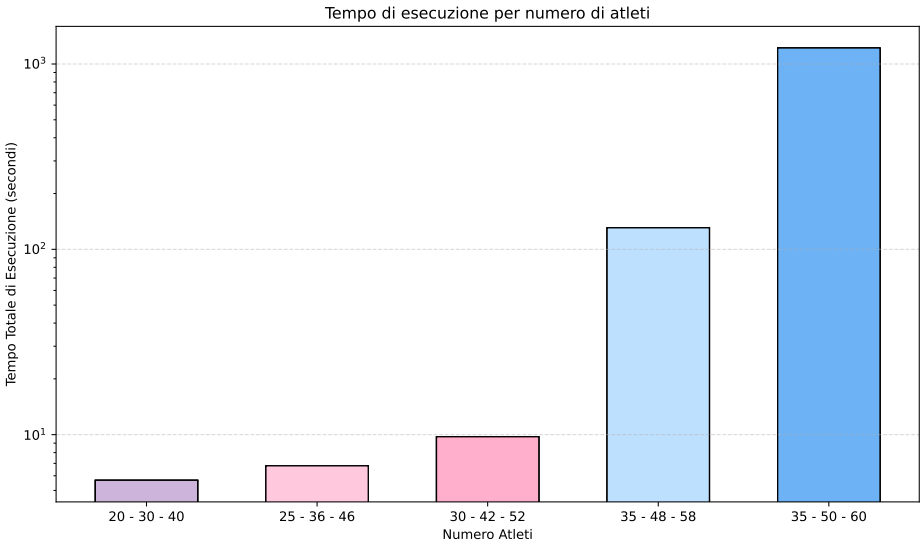

Figura 2: Grafico con diverse grandezze di iscritti e K = 20

Possiamo confermare che anche in questo caso, più ci avviciniamo alla saturazione del numero massimo di iscritti, più il programma impiega tempo per trovare una soluzione che soddisfi tutti i vincoli richiesti. In questo caso, la grandezza del Tatami è pari a 20. Notiamo che il numero di iscritti ai vari corsi è maggiore rispetto al test precedente, ma i tempi di esecuzione rimangono sostanzialmente gli stessi.

Successivamente, per curiosità, abbiamo voluto capire quanto tempo, in media, il programma impiegherebbe su 100 iterazioni per trovare una soluzione valida per un singolo caso. Abbiamo tenuto fissa la grandezza del Tatami, pari a 15, e abbiamo fatto variare il numero di atleti iscritti ai corsi. Di seguito possiamo osservare il grafico:

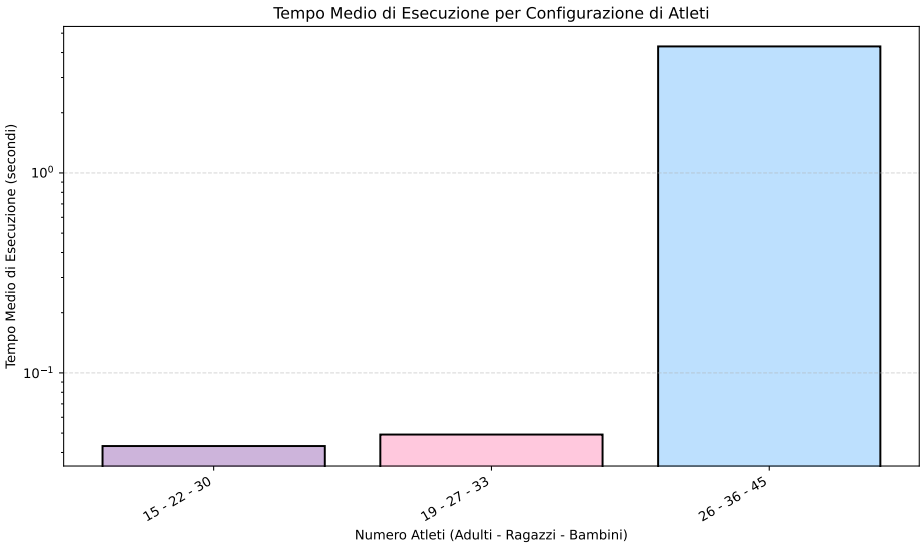

Figura 3: Grafico con k = 15 e diverse quantità di atleti

Nel primo caso, quello viola, il numero di iscritti è pari al numero di atleti che il Tatami può contenere per ogni corso, mentre nel secondo caso, quello rosa, il numero di iscritti ai corsi è pari a $\frac{3}{4}$ della capienza massima settimanale. Infine, nell'ultimo caso, il numero di atleti è pari al numero massimo di iscrizioni possibili.

Analizzando i tempi, possiamo osservare che, come ci aspettavamo, il programma impiega pochissimo tempo per trovare una soluzione nel primo caso, in quanto all'allenamento ci sarà posto per tutti e non è necessario "incastrare" gli atleti con le esigenze di ogni corso. In media, possiamo notare che il caso più "semplice" impiega poco meno di un decimo di secondo, come anche nel secondo caso, anche se il tempo è leggermente più alto, poiché iniziano a subentrare tempistiche per soddisfare tutte le necessità. Infine, il caso "peggiore" si verifica nell'ultimo scenario, in cui diventa molto più difficile generare una soluzione, dato che il numero di atleti è molto alto, il che rende più complesso accontentare tutte le disposizioni dei corsi e le esigenze degli atleti. Tuttavia, il tempo resta comunque abbastanza buono: in media, ci impiega circa 4 secondi per trovare una soluzione per un singolo dataset.

Un ulteriore test interessante riguarda i tempi, per capire quale corso impiega più tempo ad essere inserito nella settimana con tutte le esigenze descritte precedentemente. Il prossimo grafico si basa su 3 iterazioni delle 100 combinazioni, in cui abbiamo scelto, per ognuna di queste iterazioni, di saturare un corso, lasciando gli altri corsi pieni per i $\frac{3}{4}$ della loro capienza. Vediamo cosa riporta il grafico:

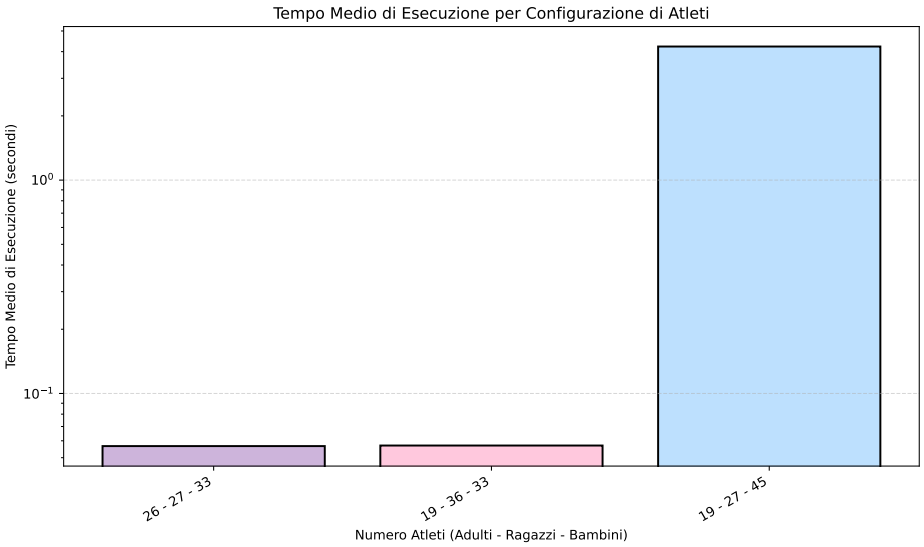

Figura 4: Grafico con k = 15 e diverse quantità di atleti

Per un Tatami con capienza di k = 15, il massimo di iscritti che possiamo avere per il corso degli adulti è 26, per il corso dei ragazzi è 36 e per quello dei bambini è 45. Nel grafico a barre in figura 4, possiamo osservare che, nel primo caso, quando il numero massimo di iscritti al corso degli adulti è saturo, il tempo medio per configurare gli allenamenti non è molto lungo, meno di un decimo di secondo. Questo potrebbe essere dovuto al fatto che gli atleti adulti hanno molti momenti durante la settimana in cui possono allenarsi rispetto ai ragazzi o ai bambini. Inoltre, il range di peso degli adulti è molto più ampio (circa 50 kg di divario tra il meno pesante e il più pesante), quindi ci saranno poche persone compatibili per fare randori tra di loro, rendendo più facile trovare combinazioni di allenamenti, dato che sarà raro trovare compatibilità tra gli atleti.

Una situazione simile si verifica per i ragazzi: il problema non è tanto il divario di peso (circa 30 kg), ma il fatto che abbiano molti allenamenti disponibili per fare randori, aumentando la possibilità di allenarsi insieme e di condividere lo stesso allenamento.

Infine, il gruppo dei bambini è il più difficile da collocare. Anche in questo caso, c'è un divario di circa 30 kg tra i partecipanti, ma il problema principale riguarda i vincoli a cui il corso deve sottostare, come il riposo tra gli allenamenti. Inoltre, la capienza del Tatami è doppia rispetto a quella degli adulti, e trovare allenamenti compatibili per una così grande quantità di iscritti, con pochi allenamenti disponibili, diventa particolarmente complesso. Tuttavia, il nostro programma riesce comunque a gestire questa situazione, impiegando mediamente circa 4 secondi per trovare una soluzione valida per un singolo dataset.

L'ultimo test ha riguardato l'analisi di eventuali comportamenti inaspettati del programma, un aspetto che può verificarsi quando i dati vengono generati in modo randomico. Per questo motivo,

abbiamo deciso di analizzare il prossimo grafico in cui abbiamo variato il valore di k e, in relazione a esso, abbiamo anche modificato il numero di iscritti ai corsi.

L'obiettivo di questo test è verificare se, aumentando k e variando di conseguenza il numero di iscritti per ciascun corso, il programma possa presentare anomalie o problemi nel generare soluzioni, dato che l'introduzione di numeri casuali potrebbe influenzare i risultati. Analizzando il grafico, possiamo verificare se ci sono fluttuazioni o comportamenti atipici nei tempi di esecuzione o nella corretta assegnazione degli atleti agli allenamenti, in modo da comprendere meglio la stabilità e l'affidabilità del sistema.

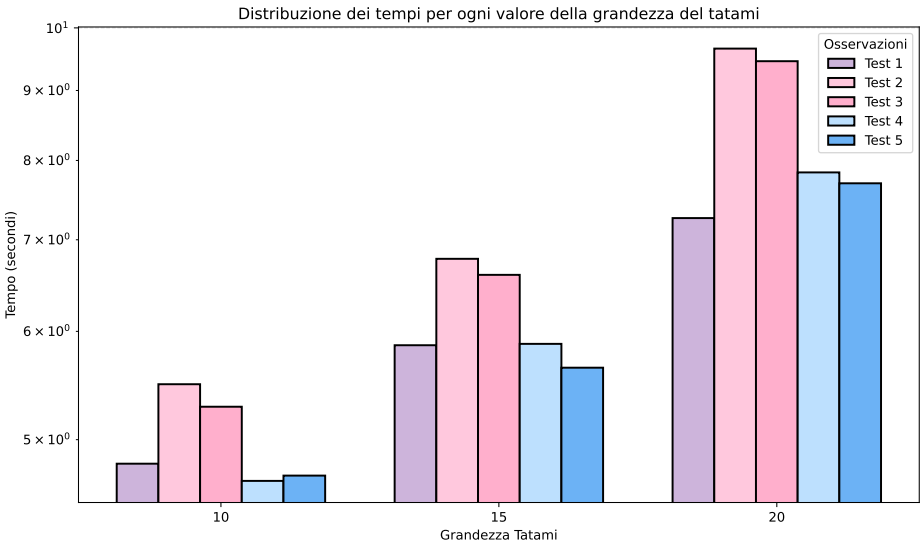

Figura 5: Grafico con k che varia e diverse quantità di atleti rapportate a k

In questo caso, il tempo non è più quello medio per un solo dataset, ma è il tempo totale per eseguire 100 dataset. Vediamo cosa succede, in pratica, abbiamo variato k da 10 a 20, con un aumento di 5 ogni volta, e scelto in modo casuale il numero di iscritti per $k = 10$. Abbiamo poi calcolato il rapporto tra il numero di iscritti e il numero massimo di iscritti possibili, per andare a vedere se, anche nel caso di $k = 15$ e $k = 20$, applicando lo stesso rapporto precedentemente calcolato per trovare il numero di iscritti rispetto alla variazione di k, abbiamo un aumento esponenziale del tempo. Ebbene sì, non si tratta di un aumento esponenziale vero e proprio a causa di qualche piccolo errore di misurazione nei tempi da tenere in considerazione. Probabilmente, abbiamo un'esecuzione del programma inaspettata tra la quarta e la quinta barra di $k = 10$ e quelle di $k = 15$ e 20. Notiamo che, in questi casi, la curva non è più esponenziale, probabilmente a causa di un intoppo nel calcolo di $k = 10$ per il quinto caso, oppure un intoppo nel calcolo della soluzione del quarto caso per quanto riguarda $k = 15$ e 20. [2]

5 Conclusioni

Arriviamo ora a parlare della conclusione di questo capitolo. Possiamo dire che il programma risulta abbastanza veloce dal punto di vista dei tempi di esecuzione, anche se vi è sempre una margine di miglioramento. Infatti, per avere, magari in futuro, risposte ancora più precise e aderenti a quelle che sono le esigenze reali e quotidiane, potrebbe essere utile apportare al programma ASP un piccolo intervento. Le modifiche riguarderebbero, ad esempio, l'introduzione di vincoli più stretti per ogni categoria di atleti, al fine di riflettere meglio le necessità pratiche e organizzative. In particolare, sarebbe utile affinare i parametri relativi alla distribuzione degli allenamenti e alla gestione dei randori, nonché ottimizzare il processo di allocazione degli atleti in base alla loro disponibilità e compatibilità. Con questi miglioramenti, il programma potrebbe diventare ancora più efficiente e aderente alle reali esigenze di gestione degli allenamenti, ma potrebbe anche avere tampistiche maggiori in quanto la complessità di esso inizierebbe a farsi troppo pesante.

[2] L'esponenzialità è stata calcolata con i dati veri e propri presenti nel file *tempistiche_k_nRatio.csv*.