Report Automated Reasoning

Alessandro Castelli (147073)

July 19, 2024

Contents

1 Introduzione

Il presente report si concentra sull'analisi delle soluzioni al problema assegnato, riguardante l'organizzazione di un torneo di calcio a 7. Il problema in questione presenta diverse sfide legate alla suddivisione ottimale dei giocatori in squadre, rispettando specifici vincoli e minimizzando le penalità assegnate alle diverse configurazioni di squadra.

1.1 Problema: Torneo di calcio a 7

> Consideriamo un insieme di n giocatori di calcio $p_1, p_2, \ldots, p_n$. Supponiamo che n sia un multiplo di 7. Ciascuno di loro può giocare come portiere (g), difensore (d), centrocampista (m), attaccante (f). Ognuno di loro sceglie esattamente due di queste opzioni (dati di input). Il problema è quello di dividerli in k squadre dove ciascuna squadra ha esattamente un g, almeno un d, almeno un m e almeno un f. Un ruolo unico viene assegnato a ciascuno di loro tra uno dei due dichiarati in input. Se una squadra ha esattamente 3 d, 2 m e 1 f, la sua penalità è 0. Se una squadra ha esattamente 2 d, 3 m e 1 f, la sua penalità è 0. Decidi (sei libero di assegnare valori) penalità non nulle per tutte le altre conformazioni. Trova una soluzione che minimizzi la somma delle penalità. Viene fornito anche un elenco di vincoli del tipo differentteam (p_i, p_j) in input che costringono p_i e p_j a essere in squadre diverse.

Il problema affrontato è un `Constraint Optimization Problem (COP)`. Per risolverlo, sono stati sviluppati due programmi che impiegano due tecniche di programmazione distinte: **programmazione per vincoli con MiniZinc** e **Answer Set Programming utilizzando il solver Clingo**.
Entrambi gli approcci sono stati selezionati per la loro efficacia nella gestione di problemi complessi e vincolati, sebbene differiscano nelle metodologie e negli strumenti adottati per ottenere la soluzione ottimale.
Nelle sezioni seguenti, verranno fornite descrizioni approfondite di entrambi gli approcci. Inoltre, sarà presentata un'analisi dei risultati ottenuti da ciascun metodo.

2 Modellazione del problema

Prima di iniziare la fase di programmazione, ho proceduto con lo studio e l'analisi del testo. Di seguito riporto alcune assunzioni che ho fatto:
Le assunzioni fatte per la risoluzione del problema sono le seguenti:

- I giocatori iniziali sono n e n deve sempre essere un numero multiplo di 7.
- Per le k squadre, non è necessario che ogni squadra abbia esattamente 7 giocatori; ogni squadra può avere un minimo di 4 giocatori (uno per ogni ruolo) e un massimo di n.
- Il numero di squadre k deve essere scelto in modo tale da avere senso rispetto al numero totale di giocatori; altrimenti, il problema non è soddisfacibile.
- Ogni giocatore può ricoprire solo uno dei due ruoli selezionati per la sua assegnazione finale.
- Ogni squadra deve avere esattamente un portiere (g), almeno un difensore (d), almeno un centrocampista (m) e almeno un attaccante (f).
- È previsto un sistema di penalità per configurazioni di ruolo specifiche, dove determinate combinazioni non comportano penalità.
- Sono forniti vincoli che richiedono che alcune coppie di giocatori siano assegnati a squadre diverse.

3 Soluzione Minizinc

Nella seguente sezione troverai la spiegazione del modello Minzinc che è stato costruito per risolvere il problema.

3.1 Input

```
n = 7;
k = 2;
roles = [
    {m, f}, % Ruolo giocatore 1
    {d, m}, % Ruolo giocatore 2
    {m, d},
    {d, g},
    {d, g},
    {g, d},
    {d, g}  % Ruolo giocatore n
];
num_pairs = 1;
player1 = [5];
player2 = [1];
```

Listing 1: Esempio di input MiniZinc

Nel codice mostrato in Listing 1, è possibile vedere quali sono i dati che il modello vuole in input. È necessario inserire:

- Il numero di giocatori `n`
- Il numero di squadre `k`
- Il ruolo dei giocatori `roles`
- Il numero di coppie di giocatori che non possono giocare nella stessa squadra `num_pairs`

- `player1` e `player2` contengono i giocatori che non possono giocare insieme. I giocatori contenuti in posizioni uguali negli array non possono giocare nella stessa squadra. Per esempio, in Listing1, il giocatore 5 e il giocatore 1 devono giocare in squadre diverse.

3.2 I vincoli Minizinc

Per la corretta modellazione del problema, sono stati definiti diversi `vincoli` che regolano l'assegnazione dei ruoli ai giocatori all'interno delle squadre, garantendo così il rispetto delle regole del gioco e l'equilibrio tra i team. Di seguito, vengono descritti i principali vincoli implementati:

```
constraint
forall(t in 1..k) (
    % Ogni squadra deve avere esattamente un
        portiere
    sum([assigned_role[j] == g | j in 1..n
        where team[j] == t]) == 1 /\
    % Almeno un difensore
    sum([assigned_role[j] == d | j in 1..n
        where team[j] == t]) >= 1 /\
    % Almeno un centrocampista
    sum([assigned_role[j] == m | j in 1..n
        where team[j] == t]) >= 1 /\
    % Almeno un attaccante
    sum([assigned_role[j] == f | j in 1..n
        where team[j] == t]) >= 1
);
```

Listing 2: Vincolo 1 Minizinc

```
% Ogni giocatore deve essere assegnato a uno
    dei suoi ruoli scelti
constraint
    forall(i in 1..n) (
        assigned_role[i] in roles[i]
    );
```

Listing 3: Vincolo 2 Minizinc

```
    constraint
    forall(i in 1..num_pairs) (
        team[player1[i]] != team[player2[i]]
    );
```

Listing 4: Vincolo 3 Minizinc

Il **Vincolo 1** assicura che ogni squadra abbia esattamente un portiere e almeno un giocatore per ciascuna delle altre posizioni chiave: difensore, centrocampista e attaccante. Questo garantisce che ogni team abbia una struttura bilanciata e conforme alle regole del gioco.
Il **Vincolo 2** specifica che ogni giocatore deve essere assegnato a uno dei ruoli per cui è stato selezionato, assicurando che le preferenze e le competenze dei giocatori siano rispettate nella formazione delle squadre.
Infine, il **Vincolo 3** stabilisce che due giocatori predefiniti non devono appartenere alla stessa squadra. Questo vincolo può essere utilizzato, ad esempio, per evitare che due giocatori incompatibili giochino insieme o per distribuire equamente i giocatori più forti tra le squadre.
Questi vincoli sono fondamentali per garantire una formazione delle squadre che sia equilibrata e conforme alle regole stabilite, facilitando così un'esperienza di gioco equa e competitiva.

3.3 Penalità

Nella competizione è stato previsto l'assegnamento di penalità alle squadre che non rispettano la configurazione richiesta del numero di giocatori nei vari ruoli. Questo criterio garantisce una formazione equilibrata e corretta per ciascuna squadra.
In particolare, le formazioni senza penalità (penalità 0) devono rispettare esattamente uno dei seguenti schemi:

- 3 difensori, 2 centrocampisti, 1 attaccante, 1 portiere
- 2 difensori, 3 centrocampisti, 1 attaccante, 1 portiere

Qualsiasi altra combinazione di giocatori comporta una penalità (penalità 1). Di seguito è riportato il codice MiniZinc utilizzato per calcolare le penalità per ogni squadra. L'array `penalties` contiene le penalità assegnate a ciascuna squadra in base alla conformità con i requisiti sopra descritti.

```
array[1..k] of var 0..1: penalties;
constraint
    forall(t in 1..k) (
        let {
            var int: d_count = sum([
                assigned_role[j] == d | j in
                 1..n where team[j] == t]),
            var int: m_count = sum([
                assigned_role[j] == m | j in
                 1..n where team[j] == t]),
            var int: f_count = sum([
                assigned_role[j] == f | j in
                 1..n where team[j] == t])
        } in
        if d_count == 3 /\ m_count == 2 /\
            f_count == 1 then
             penalties[t] = 0
        elseif d_count == 2 /\ m_count == 3
            /\ f_count == 1 then
             penalties[t] = 0
        else
             penalties[t] = 1
        endif
    );
```

Listing 5: Penalità

3.4 Obiettivo e risultati

L'obiettivo del programma è trovare la soluzione che minimizzi le penalità (vedi Listing 6).

```
solve minimize total_penalty;
```

Listing 6: Obiettivo del problema

Per valutare le performance del modello implementato, sono stati creati tre gruppi di test con livelli di difficoltà differenti: facile, medio e difficile. Ogni gruppo è composto da 10 istanze, ciascuna delle quali è stata generata in modo casuale, includendo variabili come il numero di giocatori, il numero di squadre, i ruoli dei giocatori e l'assegnazione dei giocatori a squadre diverse. I risultati dei test sono stati registrati e confrontati per analizzare l'efficacia dell'algoritmo utilizzato. I file di input e output relativi sono inclusi nel progetto e disponibili per consultazione.

3.5 Struttura dei Test

- **Facile**: 10 istanze con parametri relativamente semplici.
- **Medio**: 10 istanze con parametri di difficoltà intermedia.
- **Difficile**: 10 istanze con parametri complessi.

3.6 Metodologia

Per ciascuna istanza sono stati utilizzati due solver: Gecode e Chuffed. Il tempo massimo di esecuzione (timeout) per ogni istanza è stato impostato a 5 minuti. Se un solver non riesce a trovare una soluzione entro questo tempo, viene segnato come *TIMEOUT*.

3.7 Risultati

I risultati ottenuti sono riportati nella Tabella 1, che mostra il tempo di esecuzione in secondi per ogni istanza risolta dai solver Gecode e Chuffed.
Come è possibile vedere dai riusltati il sover Chuffed giugnge a conclusione molto più velocemnete del solver Gecode.
`TIMEOUT` vuol dire che ha superato la soglia massima di tempo che è stata impostata a 5 minuti.

3.8 Analisi dei Risultati

Come si può osservare dalla Tabella 1, il solver Chuffed riesce a concludere i calcoli in tempi significativamente più brevi rispetto al solver Gecode in tutte le istanze. In particolare:

- **Istanze Facili**: Entrambi i solver completano tutte le istanze entro il tempo limite, ma Chuffed risolve le istanze con tempi generalmente inferiori rispetto a Gecode.
- **Istanze Medie**: Gecode fallisce (*TIMEOUT*) in molte delle istanze, mentre Chuffed riesce a completare tutte le istanze entro il tempo limite.
- **Istanze Difficili**: Gecode non riesce a completare la maggior parte delle istanze entro il tempo limite, mentre Chuffed riesce a risolvere alcune istanze ma fallisce (*TIMEOUT*) nelle più complesse.

Si può osservare che, talvolta, istanze simili in `n` e `k` presentano tempi di esecuzione differenti. Questo avviene perché variano i ruoli dei giocatori e le assegnazioni dei giocatori a squadre diverse.

3.9 Conclusioni

L'analisi dei tempi di esecuzione dimostra che il solver Chuffed è significativamente più efficiente rispetto a Gecode, specialmente in situazioni di maggiore complessità. Questo suggerisce che Chuffed potrebbe essere una scelta più appropriata per applicazioni che richiedono la risoluzione di problemi complessi in tempi brevi.

Table 1: Performance Minizinc

Livello	n	k	Gecode (s)	Chuffed (s)
Facile 1	7	2	0.59	0.16
Facile 2	7	1	0.18	0.13
Facile 3	7	2	0.28	0.17
Facile 4	7	2	0.23	0.18
Facile 5	7	1	0.14	0.13
Facile 6	7	2	0.31	0.18
Facile 7	7	1	0.16	0.13
Facile 8	7	2	0.17	0.17
Facile 9	7	2	0.47	0.17
Facile 10	7	2	0.18	0.19
Medio 1	14	3	TIMEOUT	1.77
Medio 2	14	3	TIMEOUT	3.12
Medio 3	21	2	15.22	0.82
Medio 4	14	3	TIMEOUT	0.27
Medio 5	21	3	2.15	0.34
Medio 6	21	3	23.8	0.37
Medio 7	14	3	TIMEOUT	0.44
Medio 8	14	3	TIMEOUT	1.18
Medio 9	14	3	TIMEOUT	1.96
Medio 10	14	3	TIMEOUT	1.96
Difficile 1	28	2	83.84	0.53
Difficile 2	28	3	TIMEOUT	15.71
Difficile 3	28	4	TIMEOUT	1.27
Difficile 4	28	4	TIMEOUT	10.02
Difficile 5	28	2	TIMEOUT	0.55
Difficile 6	28	4	TIMEOUT	30.36
Difficile 7	28	3	TIMEOUT	TIMEOUT
Difficile 8	35	3	TIMEOUT	13.43
Difficile 9	28	4	TIMEOUT	TIMEOUT
Difficile 10	28	4	TIMEOUT	TIMEOUT