



# Constraint Implementation Documentation

This section documents how constraint handling in the scheduling algorithm aligns with the global constraints described in the research paper:

“Global Constraints for Round Robin Tournament Scheduling” by Henz, Müller, and Thiel (2004)

## 1. All-Different Constraint

Paper Reference: Section 4 – `all-different(ot,1, ..., ot,n-1)`

Code Implementation:

python

Copy

Edit

```
matches = list(combinations(teams, 2))
```

This line ensures that every match is unique and that each team pair plays only once.

Explanation:

The paper enforces that each team plays against different opponents across the rounds using an *all-different* constraint. This is implicitly implemented in the code by generating all possible unique team pairings using `combinations(teams, 2)`, ensuring each team faces all others exactly once.

## 2. One-Factor Constraint (Perfect Matchings per Round)

Paper Reference: Section 2 & 4 – One-Factorization of complete graphs

Code Implementation:

python

Copy

Edit

```
if (day in team_played_day[team1]) or (day in team_played_day[team2]):  
    continue
```

This check ensures that no team is assigned more than one match per day.

Explanation:

The one-factor constraint ensures that each round is a perfect matching of teams (i.e., no overlaps). In the code, this is enforced by tracking `team_played_day` for each team and preventing assignment of multiple matches to a single team on the same day. This simulates the behavior of one-factor scheduling by guaranteeing disjoint pairings per day.

### 3. Venue-Time Conflict Avoidance

Paper Reference: Section 6.5 – Intermural tournaments, resource conflict prevention

Code Implementation:

python

Copy

Edit

```
if len(matches_at_slot) > 1:
    penalty += 100 * (len(matches_at_slot) - 1)
```

Explanation:

This constraint ensures that no two matches are scheduled at the same time in the same venue, reflecting the physical constraint that a venue can host only one match at a time. The penalty discourages such conflicts during the evolutionary search.

### 4. Spacing Between Matches (Breaks and Carry-Over Approximation)

Paper Reference: Section 6.3 – Carry-over effect & Section 6.4 – Break minimization

Code Implementation:

python

Copy

Edit

```
for i in range(1, len(played_days)):
    if played_days[i] - played_days[i - 1] == 1:
        penalty += 20
```

Explanation:

The paper discusses minimizing the carry-over effect (e.g., teams playing consecutively without sufficient rest). In the code, a penalty is applied when a team plays on consecutive days. Although not a full carry-over matrix calculation, this approximates the idea by encouraging rest days between matches.

### 5. Schedule Diversity and Balance

Paper Reference: Section 3 – Constraint Propagation and search tree balance

Code Implementation:

python

Copy

Edit

```
var = statistics.variance(played_days)
bonus += var * 0.5
```

Explanation:

While the paper emphasizes propagation techniques for reducing search tree depth, this implementation indirectly encourages well-balanced schedules by rewarding higher variance in played days per team—thus distributing matches more evenly across the schedule.