KARADENİZ TECHNICAL UNIVERSITY
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

COMPUTER ENGINEERING GRADUATE PROGRAM

DESIGN AND OPTIMIZATION OF A GENERAL ALGORITHM TO CALCULATE POSSIBLE STATES OF FINAL TABLES OF SPORT COMPETITIONS

DOCTORATE THESIS

Mouslem DAMKHI

FEBRUARY 2021
TRABZON

KARADENİZ TECHNICAL UNIVERSITY

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

COMPUTER ENGINEERING GRADUATE PROGRAM

DESIGN AND OPTIMIZATION OF A GENERAL ALGORITHM TO CALCULATE POSSIBLE STATES OF FINAL TABLES OF SPORT COMPETITIONS

Mouslem DAMKHI

This thesis is accepted to give the degree of

DOCTOR OF PHILOSOPHY

By

The Graduate School of Natural and Applied Sciences at
Karadeniz Technical University

The Date of Submission : **30/12/2020**

The Date of Examination : **09/02/2021**

Thesis Supervisor : **Assoc. Prof. Dr. Hüseyin PEHLİVAN**

Trabzon 2021

FOREWORD

First and foremost, all the praises be to Allah for providing me strength to have this work done. All the esteem and appreciation to my gracious parents for their unconditional and unlimited support and solidarity.

I would like to express my deepest gratitude to my advisor Assoc. Prof. Dr. Hüseyin PEHLİVAN for his intellectual guidance and kind help given to me during the period of this study. Also, I am grateful to Prof. Dr. Mustafa ULUTAŞ and Asst. Prof. Dr. İbrahim SAVRAN for their valuable feedback.

Last but not least, special thanks to Mr. Abdallah SACI and Mr. Boudjemaa ROUDANE for their encouragement.

Mouslem DAMKHI

Trabzon 2021

THESIS STATEMENT

I declare that, this PhD thesis, I have submitted with the title "Design and Optimization of a General Algorithm to Calculate Possible States of Final Tables of Sport Competitions" has been completed under the guidance of my PhD supervisor Assoc. Prof. Dr. Hüseyin PEHLİVAN. All the data used in this thesis are obtained by simulation and experimental works done as parts of this work in our research labs. All referred information used in thesis has been indicated in the text and cited in reference list. I have obeyed all research and ethical rules during my research and I accept all responsibility if proven otherwise. 09/02/2021

Mouslem DAMKHI

TABLE OF CONTENTS

Page No

PhD Thesis

SUMMARY

DESIGN AND OPTIMIZATION OF A GENERAL ALGORITHM TO CALCULATE POSSIBLE STATES OF FINAL TABLES OF SPORT COMPETITIONS

Mouslem DAMKHI

Karadeniz Technical University
The Graduate School of Natural and Applied Sciences
Computer Engineering Graduate Program
Supervisor: Assoc. Prof. Hüseyin PEHLİVAN
2021, 198 Pages

The final positions in a particular single round-robin tournament can play a crucial role in the distribution of the participants' revenue, which would significantly influence the incomes of the tournament participants. So it would be of utmost importance to predict the final position of a participant at the end of a tournament. Determination of the possible states of a single round-robin final tournament table can provide a convenient way to ascertain what table data would be adequate to reach the desired position of a participant.

In this thesis, to generate the possible states of a final tournament table, backward and forward approaches were proposed. The backward approach starts by generating a state of a final tournament table and ends with trying to build a tournament graph based on it, while the forward approach starts by generation a tournament graph and ends with concluding its corresponding state of tournament final table, in which the state is taken into account as a valid one if the participants' points are in descending order and the state is not previously found.

General constraints related to the participants' points and their standings are proposed in this thesis to optimize the search space of each approach. Each participant holds a position in the final tournament table with which it is possible to determine its highest and lowest numbers of points. Optimized search spaces for each of the forward and backward approaches are proposed based on the highest and lowest possible numbers of points of the participants. To enhance the execution time of each approach, the performance of the used machine is exploited by implementing multi-threading based parallelization of the proposed optimized approaches.

Key Words: Single Round-Robin Tournaments, Tournament Table, Final Tournament Table State, Game Results, Graph, Enumeration, Combinatorics, Optimization, Multi-Threading

Doktora Tezi

ÖZET

SPOR YARIŞMALARININ NİHAİ TABLO DURUMLARININ HESAPLANMASI İÇİN GENEL BİR ALGORİTMANIN TASARIMI VE OPTİMİZASYONU

Mouslem DAMKHI

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Doç. Dr. Hüseyin PEHLİVAN
2021, 198 Sayfa

Belirli bir tekil dairesel sıralı turnuvadaki son pozisyonlar, turnuva katılımcılarının kazançlarını önemli ölçüde etkileyecek olan katılımcı paylarının dağılımında önemli bir rol oynayabilmektedir. Bu yüzden bir katılımcının bir turnuvanın sonundaki nihai pozisyonunu tahmin etmek son derece önemlidir. Tekil dairesel sıralı bir nihai turnuva tablosunun olası durumlarının belirlenmesi, bir katılımcının arzu edilen bir pozisyona ulaşması için hangi tablo verisinin yeterli olacağını saptamada uygun bir yol sağlayabilmektedir.

Bu tezde, bir turnuva tablosunun olası son durumlarını üretmek için geri yönlü ve ileri yönlü yaklaşımlar önerilmiştir. Geri yönlü yaklaşım, bir nihai turnuva tablosunun bir durumunu üreterek başlar ve buna dayanarak bir turnuva grafı oluşturmaya çalışır. İleri yönlü yaklaşım, bir turnuva grafının oluşturulmasıyla başlar ve ardından bu grafın karşılığı olan nihai urnuva tablosu durumunu belirler. Tablonun bu durumu, katılımcı puanları azalan sıradaysa ve daha once hesaplanan durumlar arasında bulunmuyorsa, geçerli durum olarak kabul edilir.

Bu tezde, her bir yaklaşımın arama uzayını optimize etmek için katılımcıların puanları ve oyun performansları ile ilgili genel kısıtlamalar önerilmiştir. Her katılımcının nihai turnuva tablosundaki pozisyonu, onun kazanabileceği en yüksek ve en düşük puanları belirlemeyi mümkün kılmaktadır. Katılımcı puanlarının en yüksek ve en düşük münkün sayılarına dayanarak ileri yönlü ve geri yönlü yaklaşımların her biri için optimize edilmiş arama uzayları önerilmiştir. Her bir yaklaşımın çalışma zamanını iyileştirmek için, önerilen yaklaşımların çoklu iş parçacığı tabanlı paralelleştirmelerini gerçekleyerek, kullanılan makinenin performansı yükseltilmiştir.

Anahtar Kelimeler: Tekil Dairesel Sıralı Turnuvalar, Turnuva Tablosu, Turnuva Tablosunun Nihai Durumu, Oyun Sonuçları, Graf, Sayım, Kombinatorik, Optimizasyon, Çoklu İş Parçacığı Kullanımı

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

A	Set of undirected edges or arrows
AFG	Afghanistan
BHS	Bahamas
BRA	Brazil
B_{RC}	Number of blind selections to find the possible tournament graphs
BWA	Botswana
c	Color
CAF	Confederation of African Football
CAN	Canada
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
CSK	Czechoslovakia
CZE	Czech Republic
C_1	Game results which are related to each other
C_2	Game results which are not related to other game results
D	Drawn games
$d(v)$	Degree of a vertex
E	Set of edges
EGY	Egypt
ESP	Spain
FIBA	The International Basketball Federation
FIFA	Fédération Internationale de Football Association (English: International Federation of Association Football)
FRA	France
G	Graph
g	Number of games
GA	Goals Against
GD	Goals Differences
GF	Goals For
GHA	Ghana

g_P	Number of the games played by a participant
GPU	Graphics Processing Unit
GR	The total of game results
ICC	International Cricket Council
IRI	Islamic Republic of Iran
ITA	Italy
JDK	Java Development Kit
JEY	Jersey
JPN	Japan
k	Rank or degree
KOR	South Korea
L	Lost games
*L*s	Lists of adjacency
M	Matrix of adjacency
MAR	Morocco
MEX	Mexico
MIMD	Multiple Instructions on Multiple Data
MISD	Multiple Instructions on Single Data
n	Number of participants
NB	Number of the generated states by a backward blind search algorithm
NP	Nondeterministic polynomial time
NR	Games with no results
N_{RC}	Number of result cases
O	Big *O* notation
OTL	Overtime lost games
OTW	Overtime won games
P	Polynomial time
P	Game points
PA	Points Against
PD	Points Differences
PF	Points For
P_{high}	The number of earned points from a high sub-tournament

P_{low}	The number of earned points from a low sub-tournament
POR	Portugal
Pts	Total number of points
q	Size of C_1
R	Game result
r	Size of C_2
RAM	Random Access Memories
S	Set of possible game results
SA	Scores Against
SAT	Boolean Satisfiability Problem
SD	Scores Differences
SF	Scores For
SIMD	Single Instruction on Multiple Data
SISD	Single Instruction on Single Data
SGP	Singapore
SL	Lost sets
SOL	Shoot-out Lost games
SOW	Shoot-out Won games
SRB	Serbia
SU	Set of the possible values of uniform states
SUI	Switzerland
SW	Won sets
T	Tied games
T_1	Tournament 1
T_2	Tournament 2
$T(n, m)$	Time complexity
UEFA	Union of European Football Associations
USA	The United States of America
V	Set of vertices or nodes
VFL	The Victorian Football League
W	Won games
X	Shoot-out or overtime won games

Y	Shoot-out or overtime lost games
$\$(n, m)$	Memory space complexity

1. GENERAL INFORMATION

1.1. Introduction

Mankind has shown great interest in sports since the prehistoric ages [1, 2]. The early civilizations have developed similar interest [3, 4], organizing many sporting events such as Olympic, Pythian, Nemean and Isthmian Games in ancient Greece [5, 6], and Ludi, Actia and Sebasta Games in the Roman Empire [7]. The industrial and scientific revolution and the mass production in the modern era have increased leisure-time [8], which provides enough time for people to actively participate in sports and follow the sporting events [9]. Thanks to the evolution of global media and communications, sporting activities have gained popularity among human race in the entire world [10].

Nowadays, sport is directly linked to economic interests [11], where the sport system is based on economic foundations that meet the need to fund the activities, programs and equipment [12, 13]. Also, the sport system creates several professional jobs [14, 15] and generates profits through advertisement [16, 17]. Since the 1980s, the relationship between sport and economy has evolved into a form of industry [18], where it has become one of the pillars of national economies [19]. Today's global sports industry which includes sporting kits and equipment, infrastructure construction, live sports events, and licensed products is worth up to 620 billion US dollars [20]. Within this economic relationship, the media plays an important role as a partner in this industry and one of its most important sources of success [21]. For example, the rights sales incomes of the Summer Olympic Games broadcasting between the years 1980 and 2008 were estimated at 1.715 billion US dollars [22].

The most used formats of tournaments in the sporting events are single elimination (or knockout), total points series (or aggregate) and round-robin. In a single-elimination tournament, the loser is disqualified from the championship immediately after losing a game, while the winner qualifies to the next level [23]. In the total points series tournament where two teams oppose each other twice, each game is played on their respective home field then the winner is determined by the cumulative results of the two games [24]. Meanwhile, round-robin is a format of tournament in which every participant

plays against all the other participants once [25] or twice [26] (often called double round-robin).

Combinatorics is one of the branches of mathematics that studies finite discrete structures and countable sets [27, 28]. In this way, it includes the counting of elements in groups [29], determining if they meet the required criteria [30], as well as studying the construction and analysis of the organisms that meet these criteria. Graph theory is one of the fields of combinatorics that studies graphs [31], which are abstract models of network drawings that link objects. These models are constituted by nodes and links between these nodes. The links between the nodes are called edges which may be directed or undirected [32]. In the context of graph theory, a round-robin tournament is a complete graph where each node represents a participant and each edge between a pair of nodes represents a game result [33].

The final positions in a particular tournament can play a crucial role in the distribution of the participants' revenue that comes from TV networks [34], advertisements [35] and so on. Thereby it would significantly influence the incomes of the tournament participants. So it would be of vital importance to predict the final position of a participant at the end of a round-robin tournament. The calculation of the possible states of a round-robin final tournament table can provide a convenient way to determine what table data would be adequate to reach the desired position for a participant.

To calculate the states of the final table of any round-robin tournament, we propose two approaches. The first one initially generates possible states and then checks if it is possible to construct tournament graphs based on them. The second one creates the tournament graphs and then derives their corresponding final state of the tournament table. To improve the results of each approach, we implement a two-step optimization strategy. Firstly, we optimize the search space based on the lowest and highest possible number of points that can be gained by each participant. Then, we use a multi-threading based parallelization technique to exploit the performance of the computer system on which the approaches are evaluated. In addition, we analyze and discuss both time and memory space complexities of the proposed approaches. Based on the results of this analysis, we define the related complexity class for the problem of determining the possible final states of a tournament table.

1.2. Literature Review

The big popularity of sports events such as the Olympic Games and the world championships of different sports like football, cricket and basketball attracted the researchers, thereby making them conduct several researches related to sports. Some of the works that have been done are presented later on in this section.

Among the interesting studies in sports are the ones which focus on the round-robin tournaments scheduling. Carlsson et. al. [36] proposed a model to perform a double round-robin tournament scheduling in a single step, where a traditional double round-robin format was extended with divisional single round-robin tournaments. Also, the study took into account a constraint programming model that characterizes the general double round-robin plus divisional single round-robin format. Pérez-Cáceres and Riff [37] solved travelling double round-robin tournament problem by proposing an algorithm which uses the team's home/away patterns based moves. Suksompong [38] came up with a new method to generate schedules for the asynchronous round-robin tournaments whether the number of teams is even or odd, where he considered three measures of a schedule that concern the quality and fairness of a tournament.

Some models have been developed to provide automated scheduling for several sports. These models can handle the issues that concern generating various types of leagues and tournaments, games venues selection and assigning referees to games. Atan and Hüseyinoğlu [39] treated the problem of simultaneously generating a game schedule and assigning main referees to football games, where they proposed a mixed integer linear program formulation for the problem by incorporating specific rules in the Turkish league. Because of the computational difficulties in solving the problem, it was approached using a genetic algorithm. Westphal [40] discussed a new approach to solve the problem of finding an optimal schedule for the German Basketball League, where time and place requirements were taken into account during the study. Kyngas and Nurmi [41] presented a successful solution method to schedule the Finnish first division ice hockey league based on another method used to schedule the Finnish major ice hockey league. The new method was a combination of local search heuristics and evolutionary methods.

Typical examples of these developed models are graphs [42], round-robin tournament [43, 44], travelling tournament problem [45] and playoff/first place elimination [46]. Januario et. al. [42] for example considered some basic sports

scheduling problems and introduced the notions of graph theory which are needed to build adequate models, where they showed how edge coloring can be used to construct schedules for sports leagues. Briskorn and Drexl [43] developed a branch-and-price approach to find optimal solutions for the scheduling of double round-robin tournament, where the approach seeks to generate schedules having the minimum number of breaks and minimizing the sum of the cost of arranged matches.

On the other hand, Croce and Oliveri [44] presented a solution to derive feasible schedules for round-robin tournament (the Italian Major Football League) with respecting the cable televisions' requirements. Goerigk and Westphal [45] presented an integer programming and local search heuristics hybrid approach to solve the problem of scheduling travelling tournaments, where this approach passes phases until the optimized scheduling is found. Kern and Paulusma [46] determined the complexity of whether a particular team in a tournament still has a chance to win the competition, where the number of competition's outcomes is arbitrary. The proposed model also includes competitions that are asymmetric in the sense that away playing teams possibly have an advantage than home playing teams.

The scheduling methods are usually based on approaches such as integer programming [47, 48], simulated annealing [49] and branch-and-bound [50]. Alarcón et. al. [47] used in their study the integer linear programming to address the referees scheduling problem in the First Division of the Chilean professional football league, where they considered balance in the number of matches each referee must officiate, the frequency of each referee being assigned to a given team, the distance each referee must travel throughout a season, and the appropriate pairings of referee experience or skill category with the importance of the matches.

Larson et. al. [48] presented an integrated constraint programming model that allows performing the scheduling of the top Swedish handball league in a single step, where the focus was particular to identify implied and symmetry-breaking constraints that reduce the computational complexity significantly. After that, an integer programming approach was used to assign actual teams to the numbers in the template in a manner that satisfies various constraints. Lim et. al. [49] made an optimized research based on the study of Easton et. al. [51], where they divided the search space and used simulated annealing instead of integer programming to search a timetable space and hill-climbing to explore a team assignment space for the travelling tournament problem. Bartsch et. al. [50] took into account the case of Austrian and German Football Leagues to generate a

regular season schedule, where they developed some branch-and-bound based algorithms which yield reasonable schedules quickly for both leagues.

Some other previous researches have focused on tournament environments from different perspectives. Eggar [52] for example addressed the number of individual players with a sufficiently high average of wins when teams play a table tennis tournament. McSherry [53] investigated the issue of inferring the numbers of wins, draws and losses of teams from their points in the final table of a sports tournament. In another research, Charon and Hudry [54] described the principles of an exact branch-and-bound search with a Lagrangean relaxation based method, which was designed to solve the linear ordering problem for any weighted tournament. Also, Hemasinha [55] presented an algorithm to generate score sequences of all tournaments up to a given size based on the Havel-Hakimi [56, 57] criterion that checks whether or not a given sequence of integers is the score sequence of a graph. In another context, Duckworth and Lewis [58] described a method to set revised target scores. The method was based on the number of runs and number of wickets fallen for the team batting second, once a limited-overs cricket match has been forcibly shortened after it has commenced.

Nabiyev and Pehlivan [59] described a tournament scoring problem which is concerned with the construction of valid initial states according to some given final state and constraints, where it can easily be generalized to incorporate various sports disciplines played in both tournament and league environments. Pehlivan and Nabiyev [60] also dealt with the issue of determining the scores of all matches involved in a football tournament, where the table data is firstly used to compute the possible results of all played matches, then possible scores of the matches are computed based on the results as well as the total number of scored and conceded goals. In another study, Damkhi and Pehlivan [61] explored all possible results of the games involved in a tournament, based on the generation of the complete graphs which match the data in its final table. The approach was evaluated by applying it to the table data of a qualifying group from previous FIFA World Cup Group Stages. In addition, the results was presented with respect to some particular data in the final tables of the tournaments contested by up to ten teams.

Concerning the determination of the final states of football tournaments, Damkhi and Pehlivan [62, 63] proposed two different methods. The first method [62] focuses on generating all possible final states based on the results of a blind search algorithm. Since most of the states generated by a blind search algorithm are inconsistent (i.e. invalid

states), an efficient algorithm was proposed to filter out such invalid states. The second method [63] describes a point-based algorithm to generate all possible final states that can take place in a football tournament with some number of teams. The algorithm attempts to determine the possible final *W*, *D* and *L* values of the teams using their sums of points.

The determination of the states of the final tournament table provides an elegant way to ascertain which participant's table data guarantees a particular position. This thesis focuses on generating the possible final states of round-robin tournaments tables of any sport discipline according to the number of participants. To achieve that, two approaches are developed, where the first approach generates every possible state and seeks to prove their validity through the construction of a tournament graph based on each generated state. The state is considered as a valid one in this approach only if it can construct at least one tournament graph. The second approach is based on building every possible tournament graph and concluding the equivalent state of the final tournament table from each generated graph. The state is considered to be valid in this approach only if it was not previously calculated. To optimize the relevant search space for both of the proposed approaches, general constraints related to the points of the participants and their standings are proposed. In addition to optimizing the search spaces, a multi-threading based parallelization technique is implemented to enhance the performance of each approach.

1.3. Scope and Purpose of the Thesis

It is of utmost importance for the competitors of a sport tournament to predict what ensures they obtain some dominant position at the end of the tournament. The generation of the states of a final tournament table provides a convenient way to determine what game results would be adequate for a team/player to reach a desired position. Every state of a final tournament table may be generated mentally or manually when the number of tournament participants is small. Conversely, as the number of the participants gets bigger, the number of the states of a final tournament table gets bigger too, and generating them will be quite hard if not impossible.

Round-robin tournament is one of many real problems that can be translated to graphs, where each participant can be represented with a node, while the result of a game between two participants can be represented with an edge. Graph theory is considered as

a mathematical field which often needs the use of algorithms. In this thesis, to generate the states of the final table of any round-robin tournament, two algorithms which are based on two different principles are proposed. The first approach is called the backward approach. In this approach, the states are initially generated, then checked if it is possible to construct tournament graphs based on them. We name the second approach as the forward approach. This approach starts with the generation of a tournament graph and ends with deriving its corresponding state of the tournament final table.

To improve the results of the proposed approaches, the study also concerns the implementation of a two-step optimization strategy. The first step is to optimize the search space based on the lowest and highest possible number of points that can be gained by each participant. The second step is to exploit the performance of the computer system on which the approaches are evaluated, using a multi-threading based parallelization technique. In addition, both time and memory space complexities of the proposed approaches are analyzed and discussed. Based on the results of this analysis, the complexity class of our problem is defined.

1.4. Tournaments Systems

A tournament is an organized form of competitions. The tournaments are organized in different systems, depending on certain contextual parameters including the number of participants and the duration of the tournament. The most commonly used tournament systems are summarized below.

1.4.1. Single-Elimination Tournament

A single-elimination or knockout tournament is a tournament where the loser of each game is eliminated from the tournament immediately [64]. However, the loser may not be eliminated from the competition in some cases, where extra games are added such as third place [65]. Each winner progresses to the next round until the final game, whose winner becomes the tournament champion. Usually, the number of games in a single-elimination tournament is equal to the number of participants.

In a single-elimination tournament, the games are progressive such as the quarter-final, followed by the semifinals and third place, and lastly the final. The process is

formed according to the number of participants, which is often a power of 2 (i.e., 4, 8, 16, 32 etc.). Occasionally, there are also competitions among the losers of the quarter-final matches to determine the fifth to the eighth positions (this usually occurs in the Olympic Games [66]). Figure 1.1 shows an example of a single-elimination tournament bracket.

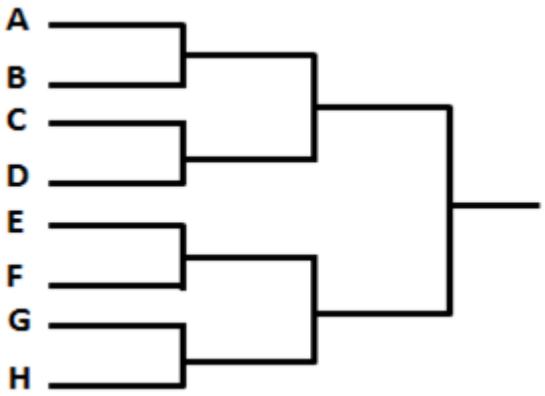

Figure 1.1. An example of a single-elimination tournament bracket

Competitors can be randomly assigned [67], however, seeding (i.e., classifying the teams to categories depending on their ranking [68]) is often used to avoid situations like a possibility of a draw which leads to the elimination of the big participants from the competition. Brackets are set so that the participants with the first and second ranking cannot meet before the final round, and none of the first four can meet before the semi-finals [68]. If no ranking is used, the tournament is called a random single-elimination (or knockout) tournament [69]. In some cases [67], the qualified participants are reseeded before playing the next stage, so that the highest qualified seed plays against the lowest seed.

1.4.2. Double-Elimination Tournament

A double-elimination tournament is a competition in which the participants are eliminated when they lose two games [70, 71]. This type of tournament allows fewer surprises than in a single-elimination tournament. A double-elimination tournament is divided into two parts [71]: the primary tournament (or tournament of the winners) and the secondary tournament (or tournament of the losers).

At the end of each round, the winners continue the primary tournament as in a knockout competition, while the losers join the secondary tournament and compete to stay in the competition, where a second defeat leads to the elimination. Compared to a team that wins all of its games before reaching the final, a team that loses its first game still has the chance to win the tournament by playing two more games. The final game opposes the winner of the primary tournament to the winner of the secondary tournament.

When the winner of the primary tournament wins the final game, the number of games in a double-elimination tournament would be two less than twice the number of the participants. For example, when the number of teams is eight, the number of games would be fourteen. If the winner of the secondary tournament wins the final, a second game takes place to respect the principle of the tournament (i.e., the participants or teams are only eliminated when they lose two games) and to avoid the elimination of a participant with only a loss [72]. So, the number of games in this case would be one less than twice the number of the participants. For example, when the number of teams is eight, the number of games would be fifteen.

This type of tournament is used for some sports like beach volleyball competitions [73], water volleyball [74] and baseball competitions [75]. Figure 2.2 shows an example of a double-elimination tournament bracket.

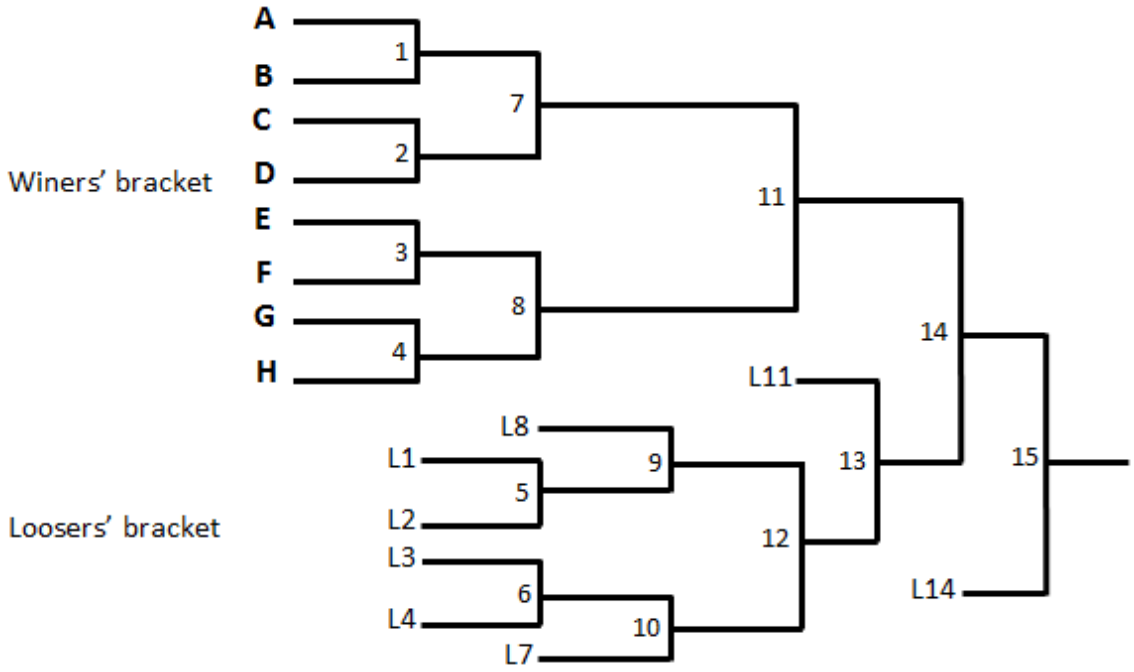

Figure 1.2. An example of a double-elimination tournament bracket

1.4.3. Round-Robin Tournament

A round-robin tournament is a tournaments system in which a team/player faces each of the other teams/players once or twice, where the winner is determined by the number of wins or the total points accumulated during the played games. The round-robin tournament is discussed in more details in Section 1.5.

1.4.4. Swiss-System Tournament

The Swiss-system tournament is a type of tournament commonly used in chess competitions [76]. This type of tournament was used for the first time at the Swiss Chess Championship in Zurich in 1895 [77], since then it was named "Swiss-System". This is a matching method for organizing tournaments with a large number of players in a reduced number of games. It is widely used for open chess tournaments [78], as well as in amateur level competitions [79].

To determine the winner of a competition, the fairest and simplest solution is when each participant plays against all the others. This method is inapplicable in a reasonable time since the number of participants is high. The knockout form is a way to designate a winner by limiting encounters. The disadvantage in this instance is the early elimination of candidates who could claim victory. The Swiss-system allows the championship formula to be maintained by limiting the duration of the competition, where the participants play the same number of games and the matching criteria make it possible to obtain a logical ranking with fairly good reliability.

The principle of the Swiss tournament is to order the players according to their rank, after that, each player in the first half of the standings plays a match against a player in the second half. For example, when there are 8 players, number 1 faces number 5, number 2 faces number 6 and so on. In the next round, each player will be opposed to the player who has done so far as well/bad as him. The tournament continues after a predefined number of rounds. Some adjustments may be done to assure that no two players oppose each other twice. After the last round, the players are ranked according to their score, where in case of a tie, they can be determined by the sum of the scores of their opponents or by another system determined in advance. In the case where the game is not symmetrical (such as chess where color is an important criterion or in football where

playing at home or away is not neutral), this criterion must be taken into account and an alternation must be realized.

Compared to a knockout system, the Swiss system has the implicit advantage of eliminating no one. This means that a player who starts such a tournament knows in advance how many rounds he will play, regardless of his results. The Swiss-system does not always end in the thrilling climate of a knockout tournament final. Sometimes, a player quickly takes such an advantage that he is guaranteed to win the tournament even if he loses his last game [80].

1.4.5. McMahon-System Tournament

The McMahon system is a tournament system which is based on the Swiss-system. The system is named after the computer scientist Lee E. McMahon [81]. This system is widely used for Go [81] competitions. As in the Swiss-system, in each round, the individual games are drawn in such a way that, if possible, participants with the same number of points meet each other. It differs from the Swiss-system by the fact that the players get some extra points before the beginning of the competition, where these points are based on the players' playing strength. This increases the chances of the high ranked participants to win the competition. The overall result of the tournament is obtained from the scores after the last round.

The advantage of this system is that it offers the possibility to determine an unequivocal winner in tournaments with many participants and a limited number of rounds, where it gives everyone the chance to play against players of suitable strength. Also, it requires fewer rounds to find a winner and avoids extreme confrontations (i.e., very strong players versus very weak players) in the first round.

Various criteria must be met in a McMahon-System tournament. Among these criteria, we mention, for example: the encounter between two participants should not take place a second time, a balanced ratio of games in black and white in games like Chess and Go, and avoiding matches between players from the same city in the national tournament.

1.4.6. Scheveningen-System Tournament

The Scheveningen-system refers to a tournament form which may be used in competitions between two teams [82]. This competition was named after the Scheveningen district since it was first used in an international competition which was held there in 1923 between Europe's best chess players and the Dutch masters [83]. Scheveningen system is somewhat similar to the round-robin system, where each player of a team plays against all the players of the other team. This form of matches would take place over several days within the tournament time. Both individual and team's results are taken into account in this system, where the individual score is based on the score earned by each player, while the team result is based on the total number of scores of players in each team.

The traditional Russian competition "Nutcracker Generation Tournament" adopts this system of tournaments, where more experienced juniors play against young Russian Grandmasters [84]. Another example is the Scheveningen competition of Saint Louis in 2011 between young American chess players and the world's best female chess players (Kings vs. Queens tournament) [85].

1.4.7. McIntyre-System Tournament

The McIntyre-system is a system of playoff which gives the chance to teams or participants to qualify to higher tournaments. The Australian K.G. McIntyre developed this system in 1931 for the VFL (the Victorian Football League) [86]. The McIntyre-system is used mainly in the Commonwealth countries, where several versions of the McIntyre-system were used by the Australian National Rugby League at different times [87, 88]. Also, the post-season in the Australian Football League is conducted using the McIntyre System [88].

The McIntyre-system passed through several updates and optimizations, where it was given many names depending on which updates. From the different versions of The McIntyre-system, there is Page–McIntyre system, where the participants are ranked based on their results of a round-robin tournament, and then the top 4 teams compete for the title based on a single-elimination or double-elimination tournament. Also, there are

McIntyre final five, six and eight systems, where five, six or eight top participants from the round-robin stage are taken into account instead of four.

1.5. Round-Robin Tournament

1.5.1. Definition

A round-robin tournament is a type of tournament in which participants play the same number of games against each other. The term round-robin is derived from the French term "ruban" which means "ribbon". With the time, the term became "robin" [89]. In a single round-robin tournament, all participants are opposed once during the competition. This type of tournaments is adopted during several events of many kinds of sports. As an example, we recall FIFA World Cups, UEFA European Championships, and Wrestling Olympic tournaments [90]. In sports where there are many games in each season, it is common for the championship to take place in two tournaments (i.e., double round-robin tournament), as is the case of football leagues [91] and College Basketball Conferences [92]. The final ranking of the participants is usually based on the number of gained points from the played games.

1.5.2. The Use of Round-Robin Tournaments

The tournaments in a round-robin fashion are widely used by many sports during different events. Double round-robin tournaments are usually used in sports events with a big number of participants like football leagues. Most of the football leagues around the world adopt this system, where every team plays against the other team twice, one game at home and the other away. Also, the double round-robin tournaments are used in the qualifications to the major tournaments like continental tournaments (e.g., UEFA, CAF, etc.) and the FIFA World Cup. This system is not exclusive to football events, as it is also used by other sports such as Chess [86]. The World Chess Championship which was held in 2005 [93] consisted of eight players which competed in a double round-robin tournament, where each player played against the others twice, one game as white and the other as black.

The single round-robin tournaments are frequently used in the group stages within a wider competition like the cases of the FIFA World Cup [94], CAF Africa Cup of Nations [95] and UEFA European Football Championship [96]. In addition to football, the single round-robin system was used by other sports like rugby during the 2010 Rugby Union ITM Cup in New Zealand [97] and the 2007 ICC Cricket World Cup [98].

1.5.3. Advantages and Disadvantages

From the theoretical side, the round-robin is the most accurate way to identify the champion from a known and determined number of participants, where each participant has the same chances as all other participants. Unlike the knockout tournament that determines the progress of the participants based on a single game, the round-robin tournament gives chances to redress the defeated games by the next games.

However, it also implies that the games which take place towards the end of the competition are sometimes opposed by participants who still compete for a prize to others who have nothing to gain. This asymmetry is detrimental to fairness since the motivation of the opponent is not the same at the beginning and the end of the competition. At the first chess tournament, held in London in July 1851, a prize was only awarded for first place, so that many players did not dispute all their games [99]. Because of that, at the London 1862 Chess Tournament, a prize was planned for the first six participants at the end of the tournament [100]. On the other hand, in the case of a qualifying tournament for another more important event, a participant that is already qualified for the next stage may try to reserve its/his forces or even deliberately lose if this defeat contributes to the promotion of a weaker future opponent, or it will help a friend opponent to qualify for an upper stage. Such a situation happened in what became known as the disgrace Gijon [101] during 1982 FIFA World Cup, where West Germany and Austria set up the result of the game between them to advance to next stage and eliminate Algeria on goal difference.

When the round-robin tournament is used as a qualifying round within a larger tournament, some participants ensure their qualification to the upper stage, but they lose some games intentionally to avoid playing against specific participants in the next stage, such as stronger participants or participants from the same country. For example, four pairs of women's doubles badminton in the Olympics of 2012 were disqualified after

qualifying for the next round for attempting to lose in the round-robin stage to avoid playing against higher ranked opponents [102].

Round-robin tournaments can be too long compared to other types of tournaments, for that a scheduling process may be necessary. The round-robin tournament requires a number of rounds that equals to one less than the number of participants if it is even, and equals to the number of participants if it is odd, unlike the knockout tournament where half of the participants are eliminated after each round. For example, an 8-teams round-robin tournament requires 7 rounds, where each round contains 4 games, which means a total of 28 games, while an 8-teams knockout tournament can be computed just in 3 rounds with a total of 8 games (i.e., taking the third place game into account).

Usually, there is no spectacular final game in a round-robin tournament. The final game is rarely between two participants which seek for the first standing. A notable instance of such a final was between Arsenal and Liverpool in the season of 1988/1989 of the Premier League competition [103]. Another disadvantage of round-robin tournaments appears especially in the small round-robin tournaments. For example, in a round-robin between three participants *A*, *B* and *C*, when *A* defeats *B*, *B* defeats *C*, and *C* defeats *A*, there will be no possibility to define the winner since all the participants have the same final results (i.e., one win and one loss). In this case, a tie-breaker is needed to separate the teams like the difference between the scored and received goals in the football tournaments [98]. This case was faced when all the teams of the Group E of 1994 FIFA World ended the group stage with one win, one draw, and one loss [104].

1.6. Graphs

1.6.1. Definition

There are several variations in the definition of graphs in graph theory. In general, a graph G is a pair $G = (V, E)$ formed of a set V of vertices, nodes or points and a set E of edges or lines which are associated with subsets with two elements of V. Other meanings for the term graph come from another interpretation of the set of edges. In a broader sense, E is a set with an incident relation that associates two vertices from V with each edge from E, where these vertices are considered as the edge's extremities. An example of a graph is shown in Figure 1.3.

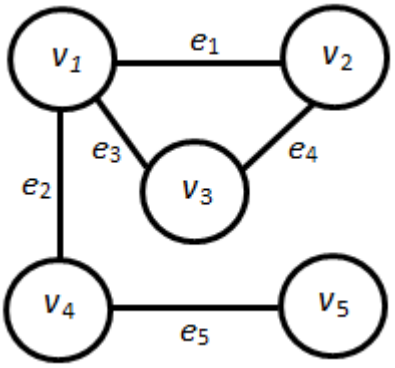

Figure 1.3. An example of a graph

Sets V and E are generally assumed to be finite. Moreover, V is often assumed as not empty, but E can be an empty set. The size of a graph depends on the number of its vertices or the number of its edges. The degree or valence of a vertex is the number of edges connected to that vertex. In a graph G, the vertices v_i and v_j are said to be adjacent if there is an edge between them. For example, the vertices v_1 and v_2 in Figure 1.3 are adjacent because they are related to each other by the edge e_1.

1.6.2. Graph Theory

Graph theory is the mathematical, algorithmic and computer science discipline that studies graphs, which are abstract models of network drawings linking objects. These models are constituted by the data of nodes or vertices, and of links between these nodes (or edges). The developed algorithms to solve problems concerning the graph theory have many applications in all domains related to the notion of network (such as: social networks, computer networks, telecommunications, etc.) and in many other domains related to the concept of a graph.

A large number of theorems have helped to establish this subject among mathematicians, which makes it a perennial branch of discrete mathematics. Graph theory originates initially by an article of the Swiss mathematician Leonhard Euler, which was presented at the St. Petersburg Academy in 1735 and published in 1741 [105]. This article dealt with the problem of the seven Königsberg bridges (see Figure 1.4). The problem was to find a path which starts from a given point and gets back to the same point, where it passes once and only once through each of the seven bridges in the city of Königsberg. Euler represented the problem with the graph shown in Figure 1.5.

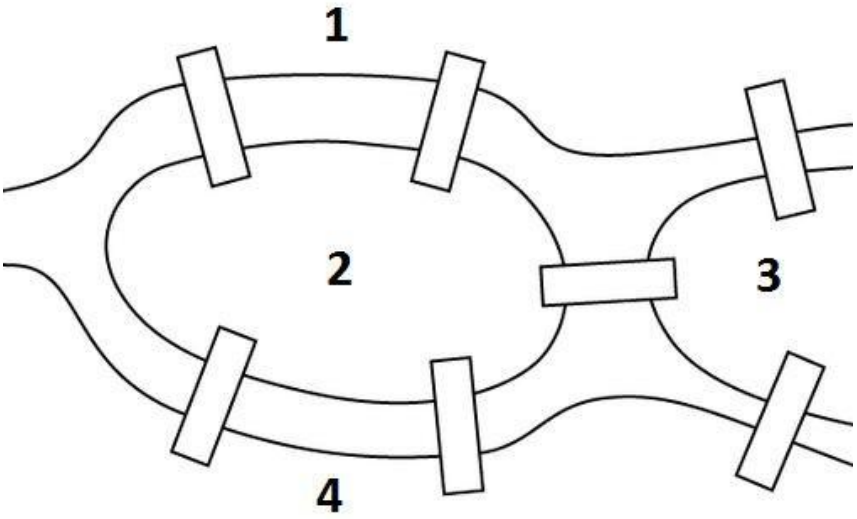

Figure 1.4. The problem of the seven Königsberg bridges

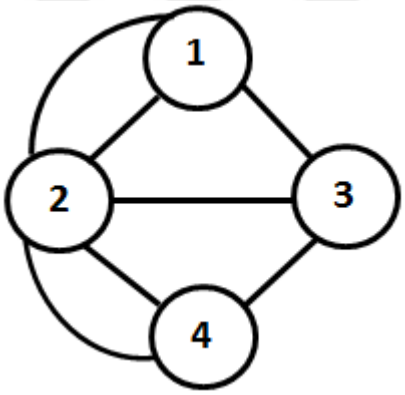

Figure 1.5. The graphical representation of the problem of the seven Königsberg bridges

1.6.3. Graph Representations

1.6.3.1. Matrix of Adjacency

Let $G = (V, E)$ be a graph where the vertices in V are numbered from 1 to n. In this case, we can say that G is a graph of order n. The adjacency matrix representation of G consists of a Boolean matrix M of size $n \times n$ (i.e., a square matrix) such that $M[i][j] = 1$ if $(v_i, v_j) \in E$, and $M[i][j] = 0$ if not. In the case of non-oriented graphs, the matrix is symmetrical with respect to its descending diagonal (i.e., $M[i][j] = M[i][j]$). In this way, only the upper triangular component of the adjacency matrix can be memorized.

There is a unique adjacency matrix for each graph, i.e. an adjacency matrix of a graph is an adjacency matrix of no other graph. The equivalent matrix of adjacency of the graph shown in Figure 1.3 is as follows:

$$M = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

The test of the existence of an edge with an adjacency matrix representation is immediate, where it is enough to check directly the corresponding element in the matrix. On the other hand, knowing the degree of a vertex requires the browsing of a whole row or a whole column of the matrix. In a more general way, browsing the set of edges requires the checking the whole matrix.

1.6.3.2. Lists of Adjacency

We can also represent a graph by giving for each of its vertices the list of vertices to which it is adjacent. Let $G = (V, E)$ be a graph of order n. We suppose that the vertices of V are numbered from 1 to n. The adjacency list representation of G consists of a table T of n lists, one for each vertex of V. For each vertex $v_i \in V$, the adjacency list $T[v_i]$ is a list of all vertices v_j such that there exists an edge $(v_i, v_j) \in E$. The vertices of each adjacency list are usually listed in an arbitrary order. In the case of non-oriented graphs, for each edge (v_i, v_j), v_j will belong to the list of $T[v_i]$, and also v_i will belong to the list of $T[v_j]$. The list of adjacency of the shown graph in Figure 1.3 is as follows: $T = [\{v_2, v_3, v_4\}, \{v_1, v_3\}, \{v_1, v_2\}, \{v_1, v_5\}, \{v_4\}]$.

There is no faster way than to traverse the adjacency list of $T[v_i]$ until finding v_j to test the existence of an edge (v_i, v_j) with an adjacency list representation. On the other hand, the calculation of the degree of a vertex, or the access to all the successors of a vertex is very effective in this case, where it is enough to browse the adjacency list associated with the vertex. In a more general way, browsing the set of edges requires the traversal of all adjacency lists. Contrarily, the computation of the predecessors of a vertex is not easy with this representation, where it requires the traversal of all the lists of adjacency of T.

1.6.4. Partial Graph and Sub-Graph

Let $G = (V, E)$ be a graph. The graph $G' = (V, E')$ is a partial graph of G, if E' is included in E. In other words, we obtain G' by removing one or more edges from the graph G. Figure 1.6 shows a partial graph from the graph shown in Figure 1.3. Compared to the graph in Figure 1.3, the graph in Figure 1.6 does not include the edge e_4 but it contains the same vertexes.

In another hand, the graph $G' = (V', E')$ is called a sub-graph of G, if V' is included in V and E' is included in E, where the extremities of each $e' \in E'$ must be included in V'. In other words, we obtain G' by removing one or more vertices from graph G, as well as all the edges incident to these vertices. Figure 1.7 shows a sub-graph from the graph in shown Figure 1.3. Compared to the graph in Figure 1.3, the graph in Figure 1.7 neither include the vertex v_5 nor the edge e_5 which is connected to it.

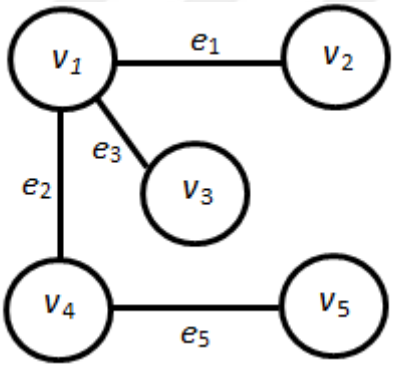

Figure 1.6. A partial graph from the graph shown in Figure 1.3

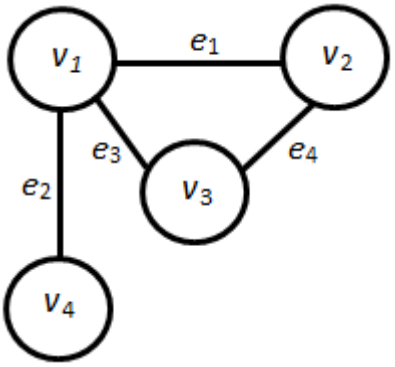

Figure 1.7. A sub-graph from the graph shown in Figure 1.3

1.6.5. Degrees

1.6.5.1. Degree of a Vertex

We denote the degree of vertex v as $d(v)$, which is the number of edges connected to this vertex (a loop on a vertex counts double). For example, in the graph of Figure 1.3, $d(v_1)=3$, $d(v_2)=2$, $d(v_3)=2$, $d(v_4)=2$ and $d(v_5)=1$. In a simple graph (i.e., a graph without loops and multiple edges), we can also define the degree of a vertex as the number of its neighbors (i.e., the size of his neighborhood). The sum of degrees of vertices in a graph is twice the number of edges. In the graph shown in Figure 1.3, for example, the graph contains 5 vertices and the total degree of $d(v_1) + d(v_2) + d(v_3) + d(v_4) + d(v_5) = 10$.

1.6.5.2. Degree of a Graph

The degree of a graph is the maximum degree of all its vertices. For example, the degree of the graph shown in Figure 1.3 is $\max(d(v_1), d(v_2), d(v_3), d(v_4), d(v_5))$ which is $d(v_1) = 3$. A graph whose all vertices have the same degree is said to be regular. If the common degree is k, then we say that the graph is k-regular.

1.6.6. Some Types of Graphs

1.6.6.1. Undirected Graph

An undirected graph is a graph whose edges have no orientations. The maximum number of edges in an undirected graph with n vertices is $n(n\text{-}1)/2$. The previously represented graphs in Figures 1.3, 1.5, 1.6 and 1.7 are undirected.

1.6.6.2. Directed Graph

A directed graph is a graph whose edges have orientations. The edges of a directed graph can be called arcs or arrows or directed edges. Such a graph is defined as $G = (V, A)$, where V is the set of vertices or nodes, and A is the set of directed edges (or arcs). An edge (v_i, v_j) is considered as oriented from v_i to v_j, where v_i is the origin or

beginning of the edge, and v_j is its end. The vertex v_j is a successor of v_i, and v_i is a predecessor of v_j. An example of a directed graph is shown in Figure 1.8. The edge a_1 or (v_1, v_2) is oriented from v_1 to v_2, where v_1 is the beginning of the edge, and v_2 is its end. The vertex v_1 is a successor of v_2, and v_1 is a predecessor of v_2.

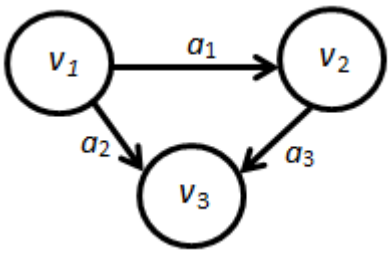

Figure 1.8. An example of a directed graph

1.6.6.3. Mixed Graph

A mixed graph is a graph composed of undirected edges as in undirected graphs, and directed ones as in directed graphs. Such a graph is defined as $G = (V, E, A)$ with V as the set of vertices or nodes, E as the set of the undirected edges, and A is the set of the directed edges (or arcs). Figure 1.9 shows an example of a mixed graph.

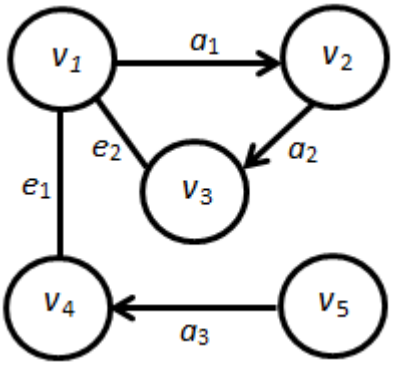

Figure 1.9. An example of a mixed graph

1.6.6.4. Simple Graph

A simple graph is a graph with no multiple edges and no loops. In a simple graph, each edge is a pair of distinct vertices, and each pair of distinct vertices represents a unique edge. The graphs previously shown in Figures 1.3, 1.5, 1.7, 1.8, 1.9 are simple.

1.6.6.5. Multigraph

Multigraph is a graph that consists of a finite set of vertices and edges that connect two vertices or a vertex with itself, which means that unlike the simple graph, a multigraph is a graph with multiple edges and/or loops. A multigraph can be undirected, directed, or mixed. The representative graph of the problem of the seven Königsberg bridges (see Figure 1.5) is considered as a multigraph because there are two edges between the vertices 1 and 2, and two other edges between the vertices 2 and 4.

1.6.6.6. Regular Graph

A regular graph is a graph where all vertices have the same number of neighbors, that is, the same degree. A regular graph whose vertices are of degree *k* is called a *k*-regular graph or regular graph of degree *k*. A 0-regular graph is a set of disconnected vertices, while a 1-regular graph has an even number of vertices, where each vertex is connected only with another vertex. Figure 1.10 shows an example of a regular graph (3-regular).

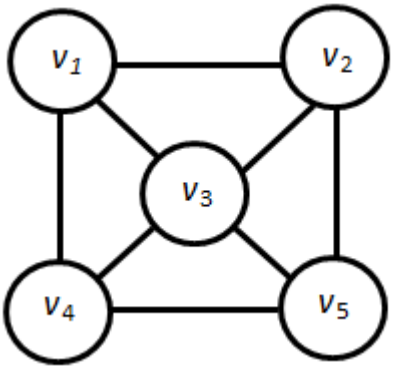

Figure 1.10. An example of a regular graph

1.6.6.7. Finite Graph

A finite graph is a graph whose set of its vertices is finite. In the opposite case, the graph is called an infinite graph. Most often, the graphs considered tacitly to be finite without explicitly saying so.

1.6.6.8. Complete Graph

A complete graph is a finite simple graph whose all vertices are adjacent, which means that any pair of its vertices is connected by an edge. A complete graph with *n* nodes is an (*n*-1)-regular graph, where each node is connected with the other *n*-1 nodes. The number of edges of a complete graph with *n* node equals to *n*(*n*-1)/2. Figure 1.11 shows an example of a complete graph with 4 nodes.

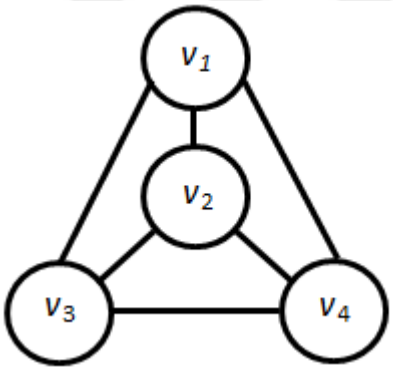

Figure 1.11. An example of a complete graph with 4 vertices

1.6.6.9. Tree Graph

A tree graph is an undirected graph with shape that evokes the ramification of the branches of a tree. There are two types of vertices in a tree, leaves whose degree is 1 and internal vertices whose degree is greater than 1. A set of trees is called a forest. A finite tree is a graph whose number of vertices exceeds the number of edges by exactly one unit. A vertex *v* is chosen in a tree to be root if there is a path from *v* to all the other vertices. Figure 1.12 shows an example of a tree graph where the vertex v_1 is its root, and the vertices v_5, v_6 and v_7 are its leaves.

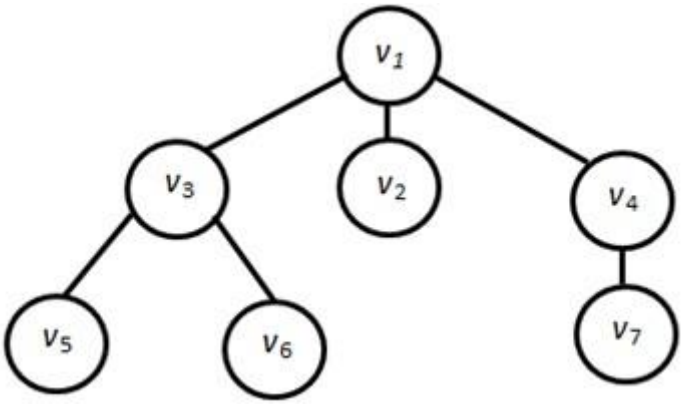

Figure 1.12. An example of a tree graph

1.6.7. Graphs Coloring

1.6.7.1. Vertices Coloring

Let $G = (V, E)$ be a graph. A subset S of V is a stable if it contains only non-adjacent vertices two by two. In the graph shown Figure 1.3 for example, $\{v_1, v_5\}$ form a stable, as well as $\{v_2, v_4\}$, $\{v_2, v_5\}$, $\{v_3, v_4\}$ and $\{v_3, v_5\}$. The coloring of the vertices of a graph involves assigning a color to all the vertices of the graph so that two adjacent vertices do not have the same color. A coloring with k colors is, therefore, a partition of the set of vertices in k stable. Coloring a vertex is just a way to differentiate between that vertex and its neighbors, which means that it is not necessarily that the vertices must be drawn with colors, but they can be represented with other means such as drawing them with different line dashes or labeling them with specific characters.

On the graph shown in Figure 1.13, three colors (denoted c_1, c_2 and c_3) are needed to color the vertices so that two adjacent vertices have different colors. We thus have three stables: $\{v_1, v_5\}$, $\{v_2, v_4\}$ and $\{v_3\}$. Figure 1.13 shows a vertices cultured version of the graph in Figure 1.3. The coloring of a graph is not necessarily unique. For example in the graph shown in Figure 1.13, vertices v_2 and v_5 could be colored in the same color, and also v_3 and v_5, while v_1 can take the remaining color.

The vertices of a simple graph can be colored using at most four colors so that all edges have different colors. This conjecture was formulated for the first time by the South African mathematician Francis Guthrie in 1852 [106]. It was then a question of coloring geography map where each state is represented with a vertex, while edges represent the borders between the states. The proof of this theorem was not ascertained until 1976,

thanks to Kenneth Appel and Wolfgang Haken [107]. The graphs coloring is used to solve some problems of incompatibility issues such as the problems of frequency allocation [108], scheduling [109, 110], and parallelism [111].

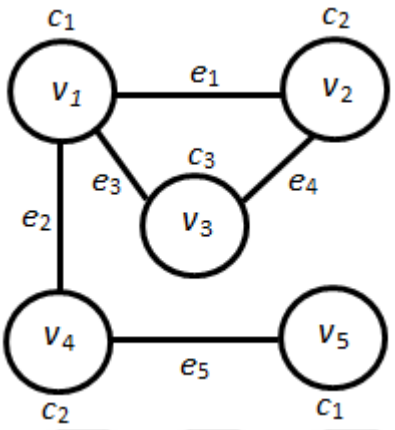

Figure 1.13. A vertices colored version of the graph in Figure 1.3

1.6.7.2. Edges Coloring

The coloring of graph edges involves assigning a color to every edge, in which two adjacent edges do not take the same color. The problem of vertices coloring can be adapted to edges coloring. In this circumstance, we will not deal with a graph *G* itself, but we deal with an alternative graph noted *G*', where each edge of $G = (V, E)$ is equivalent to a vertex of $G' = (E, E')$, and two vertices of *G*' are connected by an edge if the two corresponding edges of *G* are adjacent. The principles of the vertices coloring can be used to color the vertices of *G*'. Figure 1.14 shows a colored *G*' graph based on the represented graph in Figure 1.3. Once the coloring of *G*' is done, every edge in *G* will be colored with the same color as its corresponding vertices in *G*'. Figure 1.15 shows an edges colored version of the graph in Figure 1.3 based on the graph in Figure 1.14.

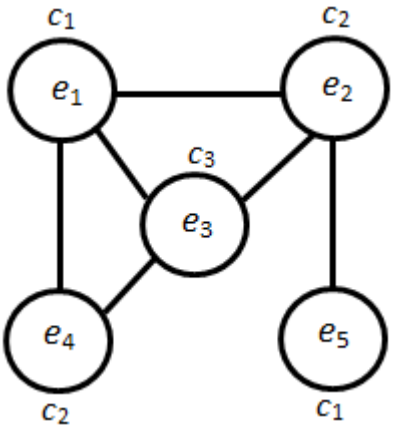

Figure 1.14. A colored *G*' graph based on the graph in Figure 1.3

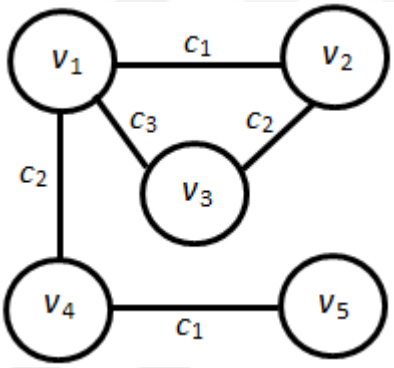

Figure 1.15. An edges colored version of the graph in Figure 1.3

1.7. Round-Robin Tournament in Graph Theory

From a graph theory perspective, a round-robin tournament is a complete directed graph. In other words, a graph $G = (V, A)$ is called tournament graph if it is a graph such that between any node (or vertex) x and any other vertex y, there exists a directed edge $a = (x, y)$.

The tournament graph represents the results of a tournament in which each participant, x and y confronts each other once and only once, where the encounter ends in either x wins or y wins. In this case, the nodes represent the participants while the edges represent the results of the played games. The directions of the edges are from the winner to the loser.

1.8. Round-Robin Tournament Table

1.8.1. Ranking

A ranking or standing is a relationship between a set of elements such that, for one or more criteria, the first one has a value greater than the second, this in turn greater than the third and so on, allowing two or more different elements to have the same position. The order is reflected by assigning to each element an ordinal, which is usually a positive integer. In this way, to provide a simpler and easier way to understand classification that replaces more complex information that can include multiple criteria, detailed measurements can be reduced to a sequence of ordinal numbers.

In many sports, individuals or teams are given rankings. In football for example, national teams are classified in the FIFA World Ranking [112]. In the Olympic Games, each country is classified based on gold, silver and bronze medal in the Olympic medal classifications [113]. In ice hockey, national teams are classified in the IIHF World Ranking [114]. With the same principle, the participants of sports competitions such as football leagues and qualifications are ranked based on their number of points in the final competition table.

In a situation, where two or more participants are having the same ranking (i.e., the same ordinal numbers), one of the following classifications will be taken into account.

1.8.1.1. Standard Competition Ranking (1224)

In this case, when an ordinal number appears more than once during ranking the participants of a competition, the k participants with the same number will have the same ranking r, while the ranking of the participant below will be $r+k$. That means if two (or more) participants tie for a position, the positions of all those classified below them are not affected. For example, if A is ranked before B and C (which both have the same ordinal number), which are ranked before D, then A is ranked the first, B and C are ranked second while D is ranked fourth. In this case, no participant would be ranked third and this position would remain as a gap.

1.8.1.2. Modified Competition Ranking (1334)

Unlike the previous case, when some participants have the same ranking, the gap in the ranking is left before them. In this case, the *k* participants with the same ordinal number will have the same ranking *r*, while the ranking of the participant below will be *r*+1, where *r* equals to the ranking of the participant before plus *k* (which is the number of participants with the same ordinal number). As in the previous case, this classification means that if two (or more) participants tie for a position, the position of all those classified below them are not affected. As an example, when a participant *A* is ranked before *B* and *C* which both have the same ordinal number and ranked before *D*, then *A* is ranked as the first, both *B* and *C* are ranked as third, and *D* as fourth of the classification. In this case, no participant would be ranked as second and this position would remain as a gap.

1.8.1.3. Dense Ranking (1223)

In this kind of ranking, the same rank is assigned to the participants with the same ordinal number, where the next participant receives immediately the next rank. In this case, there is no gap in the ranking, where the *k* participants with the same ordinal number will have the same ranking *r*, while the ranking of the participant below will be *r*+1. Therefore, in a competition between 4 participants, if a participant *A* is ranked the first, and participants *B* and *C* which have the same rank are both ranked after *A* and ahead of *D*, then *B* and *C* share the second rank, and *D* is the third.

1.8.1.4. Ordinal Ranking (1234)

All the participants of a competition in this case receive different ranks. In the event when two or more participants have the same ordinal number, the assignment of their ranks may be arbitrarily, but it is usually preferable to use a system that is arbitrary but coherent, such as one that uses the alphabetical order of the participants' names. In some competitions, when two or more participants share an ordinal number, other criteria are taken into account to determine the ranks of the participants (i.e., tie-breaking criteria) [115]. These criteria are ranked, in such that whenever there is a tie situation the next

criterion (i.e., the one ranked below) will be considered [116]. With this strategy, if *A* is ranked before *B* and *C* (which both have the same rank), which are ranked ahead of *D*, then *A* is the first, and *D* is ranked the fourth, while depending on the used criteria to break the tie situation, either *B* is ranked as the second and *C* as the third, or *C* is ranked as the second and *B* as the third.

1.8.1.5. Fractional Ranking (1 2.5 2.5 4)

In this classification, there will be an upper gap and a lower gap in a competition where two or more participants have the same ordinal number. Thus, their rank would be the average of the ranks of the upper and lower participants, which means that the rank would be exactly in the middle. Therefore, if a participant *A* is ahead of *B* and *C* which are sharing the same ordinal number and both ranked ahead of *D*, then *A* is ranked as the first, *D* obtains the fourth position, and the rank of each of *B* and *C* would be the average of the first and the fourth positions (i.e., (1+4) / 2 = 2.5).

1.8.2. Games Results Table

The game results table of a round-robin tournament is a table which represents the results of each participant against the others in the form of a matrix. The form of this table is somehow similar to the matrix of adjacency which is discussed earlier in Section 1.6.3.1. For that, this table can be considered as another way to represent a tournament graph. Supposing that the graph shown in Figure 1.8 is a tournament graph between 3 participants (i.e., v_1, v_2 and v_3), Table 1.1 represents the corresponding game results table with the graph in Figure 1.8, where the edges a_1, a_2 and a_3 mean that v_1 won against v_2, v_1 won against v_3 and v_2 won against v_3 respectively. In Table 1.1, a won game is represented with 1 while a lost game is represented with 2.

In general, let $G = (V, A)$ be a tournament graph of order n. We suppose that the vertices (or nodes) of V are numbered from 1 to n. The matrix of game results which represent G consists of a matrix M of size $n \times n$, where $M[i][j]$ contains the result of the game between the participants i and j. Since a participant i cannot play against itself, $M[i][i] = 0$. The information in the game results (i.e., the results of each participant

against the others) is used to determine the standing of the participants at the end of the tournament.

Table 1.1. The corresponding game results table of the graph shown in Figure 1.8

Participants	v_1	v_2	v_3
v_1	0	1	1
v_2	2	0	1
v_3	2	2	0

1.8.3. Sport Tournament Table

A sport tournament table is a list that shows how successful a participant is compared to other participants. Such tables are usually published in newspapers and other media like websites. The tournament table shows at least the names of the participants and their number of the achieved points or the winning percentage. However, the numbers of played games, wins, losses, draws (ties), scored goals, allowed goals, and goals difference of each participant are shown in many tournament tables. The information of the tournament table is a translation of the game results tables. For example Table 1.2 represents the tournament table which is derived from the game results presented in Table 1.1.

Table 1.2. The concluded tournament table from Table 1.1

Participants	**Wins**	**Loses**
v_1	2	0
v_2	1	1
v_3	0	2

The participants in a tournament table are ranked from the best to the worst based on a set of criteria such as the number of the gained points, winning percentage, scored

goals, received goals, etc. The participants' ranks can be used to determine the winner, the promoted, or the relegated participant. The structure of tournament tables differs according to the evaluation method and set rules of each sport. Generally, the number of points is taken into account as a criterion to rank the participants. The number of points of each participant is calculated based on its/his game results. In some sports, to rank a participant, the winning percentage is taken instead.

Some formats of the round-robin tournament tables, which are commonly adopted for the big events of different kinds of sports, can be categorized to the following classes based on the possible results of games:

1.8.3.1. Wins, Draws / Ties and Losses (*WDL*)

A tournament table with this format mainly contains the columns for each of the participants, the number of wins (*W*), the number of draws (*D*), the number of losses (*L*) and the number of points (*Pts*). Beside these columns, the tournament tables of sports like football [117] and handball [118] also contain other columns related to the goals for (*GF*), goals against (*GA*), and goals differences (*GD*). The information in the *Pts* column (i.e., number of points) is the main criterion to rank the participants, where the participants are ranked based on the descending order of their numbers of points. The following table presents the form of a football/handball round-robin tournament table:

Table 1.3. The standard form of a football/handball round-robin tournament table

Teams	***W***	***D***	***L***	***GF***	***GA***	***GD***	***Pts***
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

The term scores is used instead of goals in some sports tournaments like in Kabaddi [119]. The Kabaddi tournament table contains the same columns as the previous cases except for the *GF*, *GA* and *GD* columns, which are changed to be *SF*, *SA*, and *SD* (i.e., scores for, scores against, and scores differences respectively) [120]. In the same way,

rugby uses the term points instead of goals, where its tournament table contains the *PF*, *PA*, and *PD* columns instead of *GF*, *GA* and *GD* [121].

In regards to the total number of points, there are three systems that are followed by the addressed kind of sports in this class of tournament tables. The first system is used in football tournaments where each team earns three points for every win (i.e., P_W=3) and a single point for each draw (i.e., P_D=1), while it earns nothing in the case of a loss (i.e., P_L=0). The second system is used by most of the other kind of sports. It differs from the first system because the participant earns two points only instead of three in the case of a win (i.e., P_W=2). Unlike the other two systems, the adopted point system in the chess tournaments states that a player earns a single point in the case of a win (i.e., P_W=1), and a half of point in the case of a draw (i.e., P_D=1/2).

1.8.3.2. Wins and Losses (*WL*)

The tournament table of this class does not include *D* column, which means the participants will either win or lose their games. The basketball tournament table, for example, contains the same columns as the rugby table except for the *D* column [122]. Table 1.4 presents the form of a basketball round-robin tournament table.

Table 1.4. The standard form of a basketball round-robin tournament table

Teams	***W***	***L***	***PF***	***PA***	***PD***	***Pts***
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.

In regards to the volleyball round-robin tournaments, in addition to the *W*, *L* and *Pts* columns, the tournament table includes the columns for each of the number of the won sets (*SW*), the lost sets (*SL*), and the sets ratio. Also, the table contains columns for the total of the scored points (*PF*), the points against (*PA*), and the points' ratio [123]. Table 1.5 presents the form of a volleyball round-robin tournament table.

The total number of points in this class of tournament tables is calculated by giving the participant two points for a won game (i.e., P_W=2) and one point for a lost game (i.e.,

P_L=1). The number of points is not always the main criterion to rank the participants. For instance, in some kinds of sports tournaments such as lacrosse, the tournament table does not include the column *Pts*, and instead the number of the won games (i.e., the value of the column *W*) is taken into account [124].

Table 1.5. The standard form of a volleyball round-robin tournament table

Teams	***W***	***L***	***Pts***	**Sets**			**points**		
				SW	***SL***	**ratio**	***PF***	***PA***	**ratio**
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.

1.8.3.3. Wins *X*, *Y*, and Losses (*WXYL*)

This class includes the sport tournament tables which have no column *D* and contain two other columns related to each other, in the same way as the *W* and *L* columns. We generalize the names of these columns by symbolizing them by *X* and *Y*. In this case, when two participants *A* and *B* play against each other, if *A* ends the game in *X*, the result for *B* must be *Y*. In ice hockey [125], the tournament table is much like the football tournaments table except that the *D* column is replaced with the columns of the overtime wins (*OTW*) and overtime losses (*OTL*) as *X* and *Y* respectively. The evaluation system to calculate the total number of points of each participant gives three points to the winner (i.e., P_W=3), two points to the overtime winner (i.e., P_{OTW}=2), a point to the overtime loser (i.e., P_{OTL}=1), and none to the looser (i.e., P_L=0). The following table presents the form of an ice hockey round-robin tournament table:

Table 1.6. The standard form of an ice hockey round-robin tournament table

Teams	***W***	***OTW***	***OTL***	***L***	***GF***	***GA***	***GD***	***Pts***
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.

As *X* and *Y* columns, the curling round-robin tournaments table [126] contains the shoot-out wins (*SOW*) and shoot-out losses (*SOL*) columns respectively. These columns (i.e., *SOW* and *SOL*) are related to each other in the same way as *OTW* and *OTL*, as presented in ice hockey tournaments table. The point system in this case is similar to that of ice hockey, where each participant is given three points for a won game, two points for a shoot-out won game (i.e., P_{SOW}=2), one point for a shoot-out lost game (i.e., P_{SOL}=1), and none for a lost game.

1.8.3.4. Wins, Losses, Ties / Draws and No Result (*WLTNr*)

This case occurs in the cricket round-robin tournaments [127], where the tournament table contains the *W*, *L*, *T* and *NR* columns. The team is awarded two points for a won game (i.e., P_W=2), one point for a tied game (i.e., P_T=1) or a game with no result (i.e., P_{NR}=1), and none for a lost game (i.e., P_L=0). Table 1.7 presents the form of a cricket round-robin tournament table.

Table 1.7. The standard form of a cricket round-robin tournament table

Teams	***W***	***L***	***T***	***NR***	***Pts***
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

1.9. Parallel Computing

1.9.1. Overview

In computer engineering, parallel computing consists of implementing digital electronic architectures to make it possible to process information simultaneously, as well as the specialized algorithms for this process. These techniques aim to carry out the greatest number of operations in the shortest possible time. The processing speed which is linked to the increase in the frequency of processors eventually reached its limits. To solve this dilemma, multi-core processors have been created for the computers since the

mid-2000s [128], which made it possible to process several instructions simultaneously within the same component. These architectures can be made effective by using new methods for programming the various tasks. Such methods were initially developed to be used on supercomputers, which were at one time the only machines with many processors [129, 130]. After that, multi-core processors became more and more readily used by software developers due to the ubiquity of such architectures.

Certain types of calculations lend themselves particularly well to parallelization, such as: fluid dynamics [131], weather predictions [132], modeling and simulation of problems of larger dimensions [133, 134], information processing and data mining [135, 136], decryption of messages [137], password research [138], image processing [139], and computer generated imaging [140].

1.9.2. Flynn's Taxonomy

Flynn's taxonomy [141] was proposed by the American Michael J. Flynn in 1966, which is one of the first created systems to classify computers. Programs and architectures are classified according to the type of organization of the data flow and the instructions flow. The four models of this classification are shown in Table 1.8.

Table 1.8. Flynn's taxonomy

	Single instruction	**Multiple instructions**
Single data	SISD	MISD
Multiple data	SIMD	MIMD

1.9.2.1. Single Instruction on Single Data (SISD)

Single instruction on single data (shortly called SISD) is a term designating a hardware architecture in which a single processor executes a single instruction flow on data residing in a single memory. This means that there is no parallelization in this case. Figure 1.16 shows a representation of this architecture.

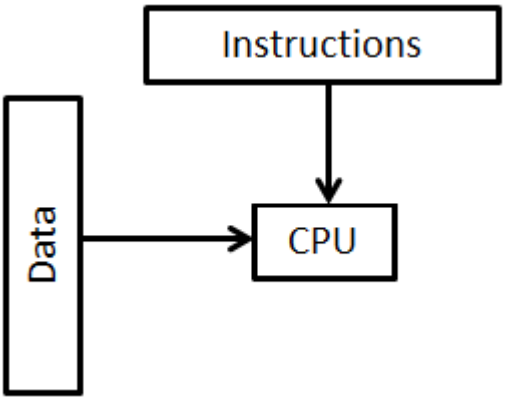

Figure 1.16. A representation of the architecture SISD

1.9.2.2. Multiple Instructions on Single Data (MISD)

Multiple instructions on single data (shortly called MISD) are an architecture which designates a mode of operation of computers equipped with various arithmetic and logical units operating in parallel. In this model, the same data is processed by several processors in parallel. This model can be used in digital signal processing [142] and treating graphics algorithms [143]. The architecture MISD is represented by Figure 1.17.

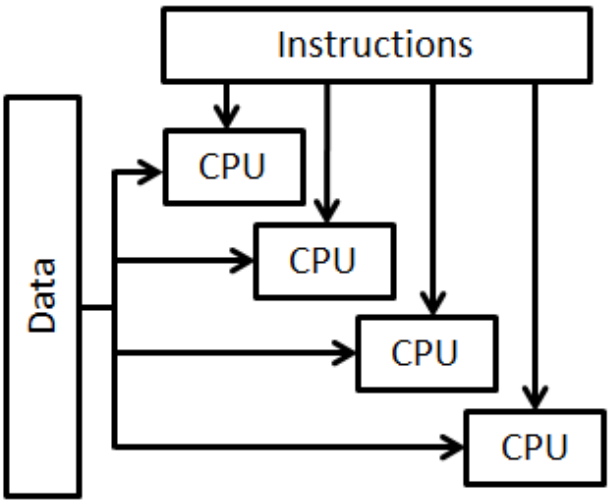

Figure 1.17. A representation of the architecture MISD

1.9.2.3. Single Instruction on Multiple Data (SIMD)

Single instruction on multiple data (shortly called SIMD) is an architecture which designates a mode of operation of computers with parallelism. In this mode, the same

instruction is applied simultaneously to multiple data to produce multiple results. The SIMD model is particularly well suited for treatments whose structure is very regular, as is the case for matrix calculation. Generally, applications that take advantage of SIMD architectures are those that use a lot of arrays, matrices, or similar data structures [144]. Figure 1.18 shows a representation of this architecture.

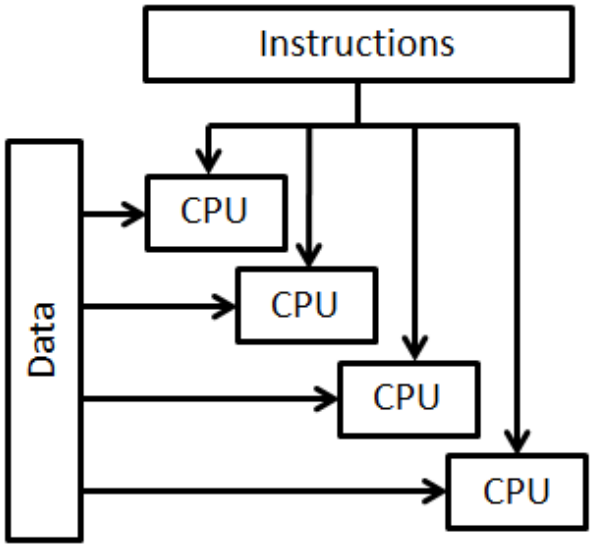

Figure 1.18. A representation of the architecture SIMD

1.9.2.4. Multiple Instructions on Multiple Data (MIMD)

Multiple instructions on multiple data (shortly called MIMD) are an architecture which designates multi-processor machines where each processor executes its code asynchronously and independently. To ensure data consistency in this architecture, it is often necessary to synchronize the processors with each other. Besides the newer work of networks of workstations, this architecture includes traditional multiprocessors (multicore and multi-threaded) [145]. A representation of this architecture is shown in Figure 1.19.

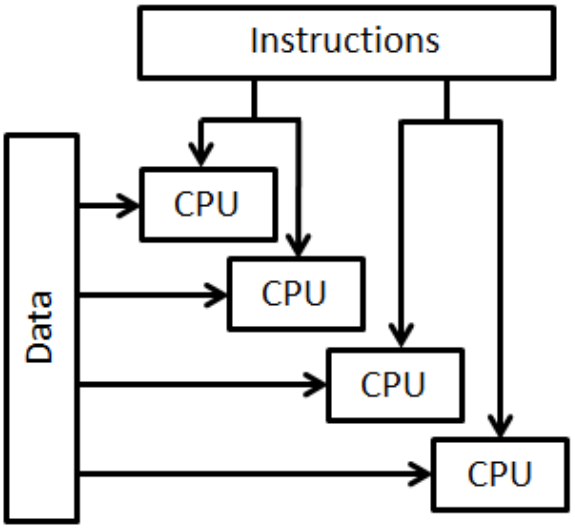

Figure 1.19. A representation of the architecture MIMD

1.9.3. Threads

A thread is a sequence of programmed instructions that can be handled separately by a scheduler, which is usually part of the operating system. A thread is similar to a process because both represent the execution of a set of machine language instructions. From the user's point of view, these executions seem to take place in parallel. However, each process has its own virtual memory [146] and all threads can share a common virtual memory [147]. In most of cases, programs which use threads are faster than classically structured programs, especially on machines with multiple processors. The main cost of using the threads is caused by switching the context of the threads [148].

Threads based programming (or multi-threading programming) is more rigorous than sequential programming, and access to certain shared resources must be restricted by the program itself. It is therefore compulsory to set up synchronization mechanisms like, for example, using semaphores [149], while keeping in mind that the use of synchronization can lead to deadlock [150] situations if it is poorly used.

The complexity of programs with threads is also significantly greater than that of the sequential programs [151]. This increased complexity, when poorly managed during the design or implementation phase of a program, can lead to some problems such as mutual exclusion [152].

1.9.4. Programming Languages and Threads

Some programming languages, such as Smalltalk [153] and some Java implementations [154] include support for threads implemented in user space (such as green threads), regardless of the capabilities of the host operating system. Most languages (like: Java, C#, C++, Ruby, etc.) use language extensions or libraries to directly use the operating system's multi-threading services.

Languages like Haskell use a hybrid system halfway between the two approaches [155]. Note that, for performance reasons depending on needs, most languages allow the use of native threads or green threads. Other languages, such as Ada, implement multitasking independent of the operating system without actually using the concept of thread [156].

Some other programming languages and extensions like OpenCL and Cuda try to fully abstract the concept of concurrency and threading from the developer, where they are designed to use GPU for sequential parallelism, without requiring concurrency or threads.

1.10. Algorithmic Complexity

1.10.1. Complexity Analysis

In computer science, the complexity analysis of a given algorithm consists of the formal study of the necessary resources' costs to execute this algorithm (usually execution time and memory space). The formal study provides a theoretical estimation for the required resources to solve the related computational problem. Most algorithms are designed to work with inputs of arbitrary sizes. In general, the complexity of an algorithm depends on the sizes and values of its inputs, the number of computer steps required to process it and the occupied memory space.

1.10.2. Asymptotic Algorithmic Complexity

Asymptotic algorithmic complexity is the use of asymptotic analysis to estimate the complexity of algorithms and computational problems. The term “complexity” in the field

of algorithmic information theory mainly refers to the asymptotic complexity. It is often associated with the use of "Big *O*" notation, which is the upper limit of the asymptotic complexity of an algorithm or problem (for example $O(n^2)$).

Calculating the number of operations performed by an algorithm and the required memory size may be impossible because of factors like the values of the inputs, the number of the iterations, the fulfillment of the conditions and the depth of the recursivity, etc. Therefore, using the concept of Big *O* to give an approximation of the number of operations and the memory size of an algorithm is more practical.

1.10.3. Big *O* Notation

Supposing that f and g are two functions, we say that f is of the order of g, if there exists a constant α and an integer n_0 such that: $f(n) \leq \alpha * g(n)$ for all $n \geq n_0$. In this case, f can be denoted as: $f(n) = O(g(n))$. Figure 1.20 shows an example of Big *O* notation, where $f(n) = O(g(n))$ as there exists $\alpha = 1$ and $n_0 = 8$ such that $f(n) \leq g(n)$ whenever $n \geq 8$.

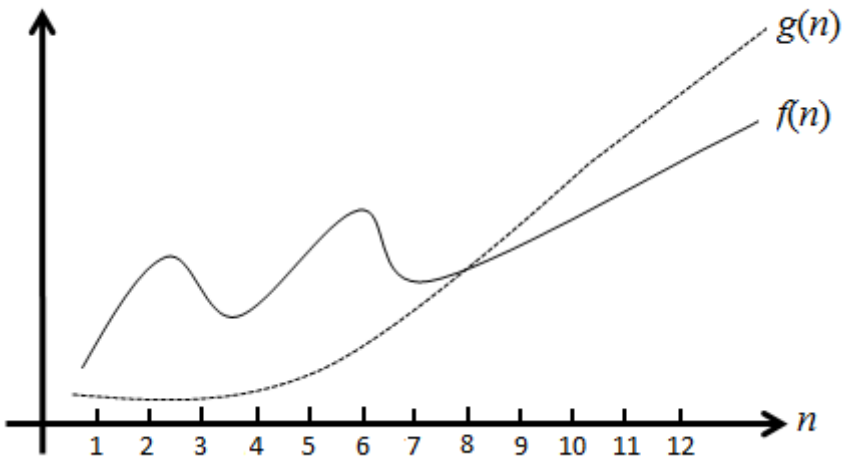

Figure 1.20. An example of Big *O* notation

The Big *O* notation can be extended to the case where the function f depends on multiple variables. For example let f and g be positive functions of two variables; we can say that $f(n, m) = O(g(n, m))$ if there exists a constant α and two integers n_0 and m_0 such that: $f(n, m) \leq \alpha * g(n, m)$ for all $n \geq n_0$ and $m \geq m_0$.

1.10.4. Properties of Big *O*

- The constants are not important. E.g., $O(c*n^k) = O(n^k)$, $O(c) = O(1)$, where c is a constant.
- Lower order terms are negligible. E.g., $O(c_0 + c_1*n + c_2*n^2 + \ldots + c_k*n^k) = O(c_k*n^k) = O(n^k)$, where $c_0, c_1, c_2, \ldots, c_k$ are constants.
- Sum of functions: $O(f) + O(g) = O(f + g)$.
- Product of functions: $O(f) * O(g) = O(f * g)$.
- Reflexivity: $f = O(f)$.
- Transitivity: if $f = O(g)$ and $g = O(h)$ then $f = O(h)$.

1.10.5. A Simple Example of Big *O*

Let us consider the following algorithm which calculates a sum of squares:

```
Inputs: n
Output: sum
1:  sum ← 0
2:  for i ← 1 to n do
3:    sum ← sum + i*i
```

We can calculate the time complexity $T(n)$ and the memory space complexity $\$(n)$ for this algorithm such that t_1, t_2, t_3, t_4, and t_5 are respectively the execution times of the assigning, incrementing, testing, addition, and multiplication, and s is the necessary memory space to store an integer. In this case:

$T(n) = t_1 + t_1 + n*(t_3+t_5+t_4+t_1+t_2) = 2*t_1 + n*\sum^{5}_{i=1} t_i = c_1*n + c_2 = O(n)$

$S(n) = 3*s = c_3 = O(1)$

1.10.6. Algorithmic Complexity Types

Let $T(n)$ be the maximum execution time of an algorithm for a data of size n. In the following we present a list of the functions commonly used in the asymptotic algorithmic complexity:

- $T(n) = O(1)$ (constant time): The execution time is independent of the size of data to be processed.
- $T(n) = O(\log(n))$ (logarithmic time): We generally encounter such complexity when the algorithm breaks a big problem into several small ones, so that the resolution of only one of these problems leads to the solution of the big problem.
- $T(n) = O(n*\log(n))$: The algorithm splits the problem into several smaller sub-problems which are solved independently. Solving all of these smaller problems solves the original one.
- $T(n) = O(n)$ (linear time): This complexity is generally obtained when the algorithm performs in a constant time on each input data.
- $T(n) = O(n^2)$ (quadratic time): This appears in particular when the algorithm considers all the pairs of data among the n inputs (e.g., two nested loops).
- $T(n) = O(c^n)$ (exponential time): It is often the result of a brute search for a solution.

The graphical representations of these functions are illustrated in the following figure:

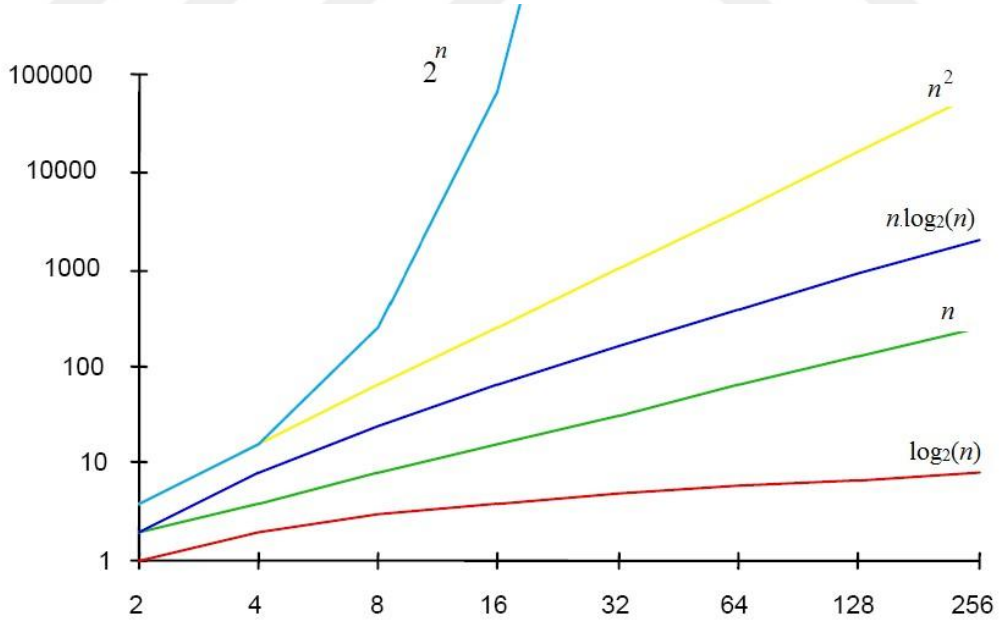

Figure 1.21. The graphical representation of the functions commonly used in the asymptotic algorithmic complexity

1.10.7. Complexity Classes

In theoretical computer science, and more precisely in complexity theory, a complexity class is a set of algorithmic problems whose resolution requires converging amounts of a certain resource (mostly time or memory space). A hierarchy of complexity classes is formed with less powerful classes being completely contained in the higher complexity classes [157]. Formal methods can be used to relate these classes.

1.10.8. P and NP Classes

We say that a problem is in P if it can be solved in a polynomial time with respect to the size of the input; i.e., if $O(f(n))$ is the complexity of a problem with an input of size n, $f(n)$ must be a polynomial function (e.g., $O(n)$, $O(n^2)$, $O(n^3)$, ...). On the other hand, if the problem fails to be solved in a polynomial time, but the validity of its output can be verified in a polynomial time, we say that the problem is in the NP class. For a problem that cannot currently be solved in a polynomial time, we cannot simply claim that it is impossible to solve that problem in a polynomial time, but a polynomial solution for it may be found in the future. So, we can refer to NP as the class of the problems that are solved in a non-deterministic polynomial time [158].

The relationship between the classes P and NP can be defined that P is a subset of NP [159], but there is no proof yet that P = NP. Although P ≠ NP is not yet proved, however, most computer scientists believe it [160]. Figure 1.22 represents P and NP classes under the assumption that P ≠ NP.

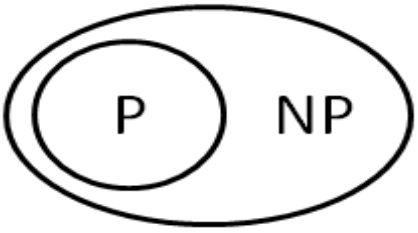

Figure 1.22. P and NP classes under the assumption that P≠NP

1.10.9. Reduction

In computability and complexity theory, the reduction is an algorithm that transforms a problem into another one. This algorithm is used to show that a problem is at least as difficult as another one [161]; i.e., if a problem *A* can be reduced to (or transformed into) a problem *B*, and *A* is difficult, then *B* is at least as difficult. In some cases, a reduction can also be used to show that a problem is easy [161]; i.e., if a problem *A* can be reduced to *B*, and *A* is easy, then *B* is at least as easy. The reduction is called polynomial when a problem *A* can be reduced to be *B* within a polynomial time [159].

1.10.10. NP-Hard and NP-Complete Classes

A problem is belong the class NP-hard if any problem of the class NP can be reduced to it in a polynomial reduction. To prove that a problem *B* is NP-hard, instead of reducing all the problems of the class NP to it, it would be enough to select a problem *A* which is known to be NP-hard and reduce it to *B* in a polynomial time [161].

If a problem is in NP and NP-hard classes, this implies that it is an NP-complete problem [161, 159]. Figure 1.22 represents each of P, NP, NP-hard and NP-complete classes under the assumption that P ≠ NP.

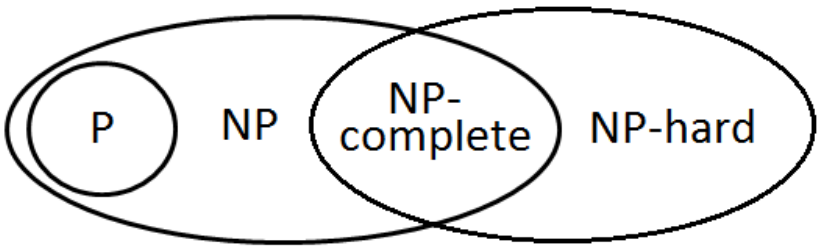

Figure 1.23. P, NP, NP-hard and NP-complete classes under the assumption that P ≠ NP

1.10.11. An Applied Example (Clique Problem)

In this example, we present the proof that the clique problem is an NP-complete problem. The clique problem is the problem of determining the maximum sized complete

sub-graph of a specific graph. For example, the represented graph in Figure 1.24 contains a clique of size 4 (i.e., the complete sub-graph consisting of the nodes 1, 2, 3 and 4).

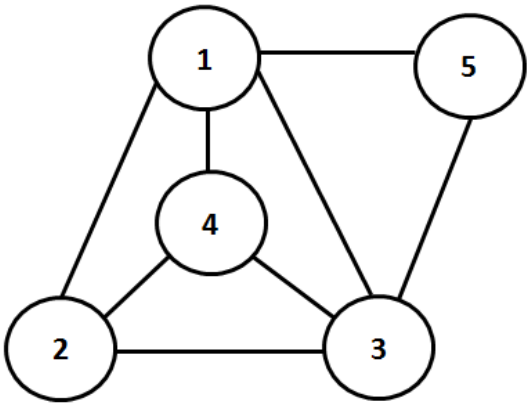

Figure 1.24. A graph with a clique of size 4

The clique problem has an exponential time complexity [163] (i.e., non-deterministic polynomial time). On the opposite side, verifying that a graph with *n* nodes contains a clique of size *k* can be done in polynomial time [162]. So, based on what is explained in Section 1.10.8, we can say that the clique problem is an NP problem and not a P problem.

Based on Section 1.10.10, in order to prove that the clique problem is an NP-hard problem, we have to select a known NP-hard problem and reduce it to the clique problem. Boolean satisfiability problem (shortly SAT) is an NP-complete problem [164] (i.e., NP-hard) that can be reduced to a clique problem. The SAT problem is a decision problem, which, given a propositional logic formula, determines whether there is an assignment of variables that makes the formula true. For example, given the following conjunctive normal form (shortly CNF): $\phi = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3)$, the values $x_1=1$, $x_2=1$ and $x_3=0$ satisfies ϕ (i.e., $\phi = 1$). Each of $(x_1 \vee x_2)$, $(\neg x_1 \vee \neg x_2)$, and $(x_1 \vee x_3)$ is called a clause. To reduce the represented SAT problem by ϕ to a clique problem, ϕ must be represented by a graph. To transform ϕ into a graph, the following rules must respected [162]:

- For CNF = $C_1 \wedge C_2 \wedge ... \wedge C_n$, where each $C_i = L_{i,1} \vee L_{i,2} \vee ... \vee L_{i,m}$, we place each $L_{i,j}$ as a node.
- There would be an edge between every two nodes unless they are from the same clause or one is the negation of the other.

For a CNF with n number of $L_{i,j}$, transforming it into a graph requires a time complexity of $O(n^2)$. Thus, we can say that the reduction is done in polynomial time. By applying the transformation rules on ϕ, we get the represented graph in Figure 1.25.

The resulted graph from a CNF with k clauses contains at least a clique of size k as the clique with the maximum size. We can see that the graph in Figure 1.25 contains 3 cliques of size 3. When we represent each node of those cliques with 1 (i.e, a true value), ϕ must be satisfied. For example, the represented clique by x_1 of C_1, $\neg x_2$ of C_2, and x_3 of C_3 implies that x_1=1, x_2=0 and x_3=1. In this case, $\phi = (1 \vee 0) \wedge (0 \vee 1) \wedge (1 \vee 1) = 1$.

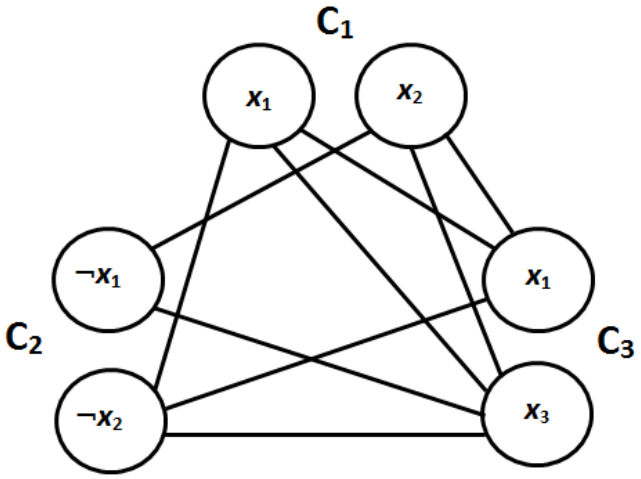

Figure 1.25. Transforming $\phi = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3)$ into a graph

Since a CNF problem is reduced into a clique problem in polynomial time and CNF problem is an NP-hard problem, we can say that the clique problem is an NP-hard problem. Given that the clique problem is an NP problem, we can also say that it is an NP-complete problem.

2. THE ACHIEVED WORK

2.1. Introduction

The study presented in this thesis aims to determine and enumerate all the possible states of the final tables for single round-robin tournaments. A table state obviously corresponds to a set of data components residing in that table, varying in range and number of values among different sports disciplines. Therefore, an analysis of the final table data is made for some popular sports disciplines such as football and basketball. Each participant in a sports tournament has a final position based upon their standings related to game results. The measure of standings differs according to sports disciplines. They are basically associated with the values such as *W*, *D* and *L* in the case of football, *W* and *L* in the case of basketball, *W*, *OTW*, *OTL*, and *L* in the case of ice hockey, and *W*, *L*, *T*, and *NR* in the case of the cricket. Firstly, we generalize tournament graphs to contain different sports disciplines, which are previously presented in Section 1.8.3. Then two result-based approaches are described to enumerate the possible states of a final tournament table.

The first approach generates every possible state and seeks to prove its validity, trying to construct a tournament graph based on that state. In this approach, the state is taken into account as a valid state only if at least one tournament graph can be sought. The second approach is based on the generation of every possible tournament graph, from which then its corresponding state is derived. In this approach, the state is taken into account only if it is not already found. To optimize the search space of both approaches, some general constraints are proposed in terms of the standings and points of the participants. Besides, a multi-threading based parallelization technique is implemented to enhance the performance of the approaches, assigning their non-overlapping operations to different threads.

Since the first approach starts by generating a state of a final tournament table and ends with trying to build a tournament graph based on it, we call it the backward generating approach. Similarly, as the second approach starts by generation a tournament graph and ends with concluding its corresponding state of tournament final table, we call it the forward generation approach. The control flow diagram of each approach is shown in Figure 2.1 and Figure 2.2.

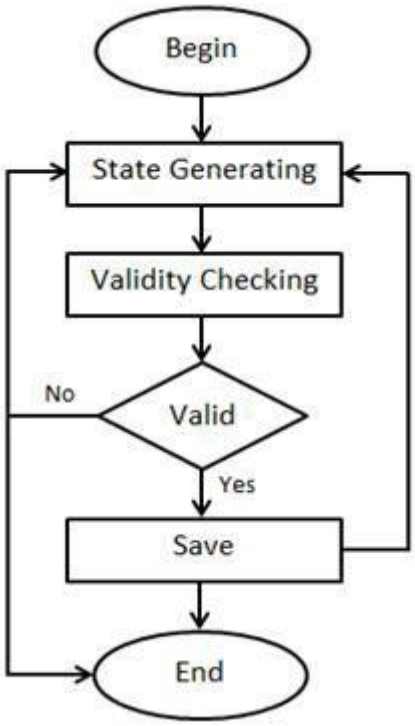

Figure 2.1. The control flow diagram of the backward approach

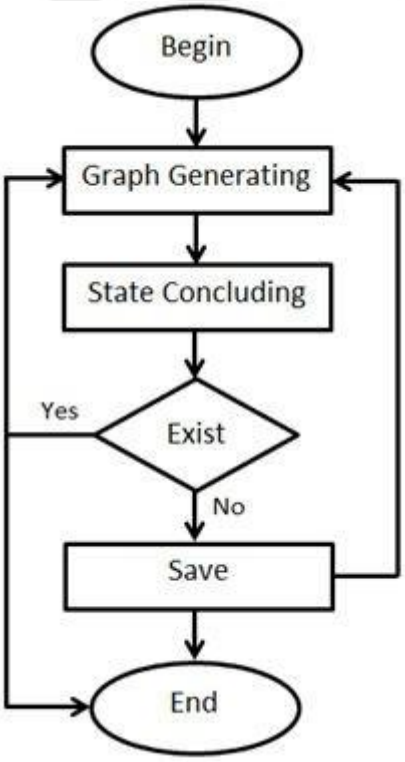

Figure 2.2. The control flow diagram of the forward approach

2.2. Classification of Game Results

As seen in Section 1.8.3, there are some dependencies among the table columns that hold game results. The result of a game can affect the data on more than one column. Considering the relationship of the columns to each other, we define two sets that contain game results; namely C_1 and C_2.

The set C_1 holds game results which can change the data on at least two columns of the tournament table. Let R_1 and R_2 denote two distinct game results. When a participant A plays against another participant B, the result of the game is differently interpreted for two teams. If the game is ended in R_1 for A, the result for B will be R_2. An example of such game results is W (Wins) and L (Losses); that is, the fact that a team A wins against another team B means that B loses the game. The game results are held as pairs of $\{R_1, R_2\}$ in C_1, expressing that, if R_1 is related to R_2, neither R_1 nor R_2 can be related to other possible game results. For example, in the case of the ice hockey table, W is related to L and OTW is related to OTL, but W is not related to either OTW or to OTL, and as well as L. Thus, C_1 can be represented as $\{\{W, L\},\{OTW, OTL\}\}$, which leads to the general form in Eq. (2.1).

$$C_1=\{\{R_1, R_2\},\{ R_3, R_4\}, ..., \{R_{2q-1}, R_{2q}\}\}, \text{ where } q = |C_1| \tag{2.1}$$

The other set C_2 contains game results which are related to the data on only one column of the tournament table. Let R denote a game result. When a participant A plays against B, the game is ended in R for both A and B. A typical example of such game results is D (Draws). Given cricket tournaments, C_2 can be represented as $\{T, NR\}$. However, C_2 can also be empty, as in the case of basketball tournament tables. In general, the set C_2 is written in the general form given by Eq. (2.2).

$$C_2=\{R_{2q+1}, R_{2q+2}, ..., R_{2q+r-1}, R_{2q+r}\}, \text{ where } q = |C_1| \text{ and } r = |C_2| \tag{2.2}$$

Table 2.1 shows the sets C_1 and C_2 for the tournament tables of some sports disciplines given in Section 1.8.3.

Table 2.1. The sets C_1 and C_2 for the tournament tables given in Section 1.8.3

Sports	**Game Results**	C_1	C_2
Football, rugby, handball, chess	*W*, *D* and *L*	{{*W*, *L*}}	{*D*}
Basketball, volleyball, tennis, lacrosse	*W* and *L*	{{*W*, *L*}}	{}
Ice hockey	*W*, *OTW*, *OTL* and *L*	{{*W*, *L*}, {*OTW*, *OTL*}}	{}
Curling	*W*, *SOW*, *SOL* and *L*	{{*W*, *L*}, {*SOW*, *SOL*}}	{}
Cricket	*W*, *L*, *T* and *NR*	{{*W*, *L*}}	{*T*, *NR*}

2.3. General Concepts of a Final Tournament Table

2.3.1. Played Games

In a round-robin tournament, every participant plays against all other participants. In general, the number of games played by a participant in a tournament between n participants is given by Eq. (2.3).

$$g_P = n - 1 \tag{2.3}$$

It is clear that the set of all the played games consists of possible ways to select two participants from the set of all participants without taking the order into account. Hence, the total number of the games which is played by the end of the tournament can be obtained based on Eq. (2.4).

$$g = C(n, 2) = n!/2!(n - 2)! = n(n - 1)/2 \tag{2.4}$$

Let us consider a tournament with $n = 4$ participants of *A*, *B*, *C* and *D*. In this case, a participant (for example, *A*) plays a total of $n - 1 = 3$ games against the others (i.e., *B*, *C* and *D*). The tournament contains a total of 6 games represented by the set $S_g(4) = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}\}$.

2.3.2. Possible Game Results

Tournament tables hold some properties associated with the values of game results. In a tournament table, the total number of game results from the set C_2 would be even, and the sum of the first elements of each pair from the set C_1 would be equal to the sum of the second elements. This follows from the fact that each game in a tournament contributes either two to an element in C_2 or one to each element of a pair in C_1.

In a final football tournament table, for example, each game in the tournament contributes either two to the total number of draws or one to the total number of wins and one to the total number of losses. For $1 \leq k \leq n$, we denote by $W(k)$, $D(k)$ and $L(k)$ respectively the numbers of the won, drawn, and lost games by team $T(k)$. Thus, the relations between the football game results can be formalized by:

$$0 \leq W(k), D(k), L(k) \leq n - 1 \tag{2.5}$$

$$[\sum_{k=1}^{n} D(k)] \equiv 0 \ mod\ 2 \tag{2.6}$$

$$\sum_{k=1}^{n} W(k) = \sum_{k=1}^{n} L(k) \tag{2.7}$$

Each of relations (2.5), (2.6) and (2.7) can be generalized to cover tournament tables of other sports as follows:

$$0 \leq R_1(k), R_2(k), ..., R_{2q+r}(k) \leq n - 1 \tag{2.8}$$

$$[\sum_{k=1}^{n} R_j(k)] \equiv 0 \ mod\ 2, \text{ where } R_j \in C \tag{2.9}$$

$$\sum_{k=1}^{n} R_i(k) = \sum_{k=1}^{n} R_{i+1}(k), \text{ where } \{R_i, R_{i+1}\} \in C_1 \tag{2.10}$$

In a final tournament table, the sum of corresponding values of the game results for each participant is equal to the number of the games played by each participant g_P. I.e.:

$$g_P = \sum_{i=1}^{2q} R_i(k) + \sum_{j=2p+1}^{2p+r} R_j(k), \text{ where } 1 \leq k \leq n \tag{2.11}$$

Table 2.2 shows the final table of Group B in the 2018 FIFA World Cup [165]. There are four teams playing against each other in a single round-robin tournament, where each team played 3 games. In Table 2.2, g_P equals to the sum of W, D and L for each team at any position k, i.e.: $g_P = 1+2+0 = 1+2+0 = 1+1+1 = 0+1+2 = 3$.

Table 2.2. The final table of group B in the 2018 FIFA World Cup

Team	***W***	***D***	***L***	***Pts***
Spain	1	2	0	5
Portugal	1	2	0	5
Iran	1	1	1	4
Morocco	0	1	2	1

Table 2.3 shows the application of Eq. (2.11) for the final tournament tables presented in Section 1.8.3:

Table 2.3. Application of Eq. (2.11) for the final tournament tables given in Section 1.8.3

Sports	C_1	C_2	g_p
Football, rugby, handball, chess	$\{\{W, L\}\}$	$\{D\}$	$W(k) + L(k) + D(k)$
Basketball, volleyball, tennis, lacrosse	$\{\{W, L\}\}$	$\{\}$	$W(k) + L(k)$
Ice hockey	$\{\{W, L\}, \{OTW, OTL\}\}$	$\{\}$	$W(k) + L(k) + OTW(k) + OTL(k)$
Curling	$\{\{W, L\}, \{SOW, SOL\}\}$	$\{\}$	$W(k) + L(k) + SOW(k) + SOL(k)$
Cricket	$\{\{W, L\}\}$	$\{T, NR\}$	$W(k) + L(k) + T(k) + NR(k)$

Table 2.4 shows the final table of Group A in the 2014 FIBA Basketball World Cup [166]. Six teams are participating in this group, where each team played 5 games. The total number of the played games g can be calculated based on the values of W and L using the following relation:

$$g = \sum_{k=1}^{n} W(k) = \sum_{k=1}^{n} L(k), \text{ where } 1 \le k \le n \quad (2.12)$$

By applying Eq. (2.12) on the data of Table 2.4 we find that: g = 5+4+3+2+1+0 = 0+1+2+3+4+5 = 15.

Table 2.4. The final table of Group A in the 2014 FIBA Basketball World Cup

Team	***W***	***L***	***Pts***
Spain	5	0	10
Brazil	4	1	9
France	3	2	8
Serbia	2	4	7
Iran	1	4	6
Egypt	0	5	5

Since C_2 = {} for basketball tournaments, we cannot rely on Eq. (2.12) to calculate g, for example, for the case of football. The total number of the played games in Table 2.2 can be calculated based on the values of W, D and L by the following relation:

$$g = [\sum_{k=1}^{n} W(k)] + [\sum_{k=1}^{n} D(k)]/2 = [\sum_{k=1}^{n} L(k)] + [\sum_{k=1}^{n} D(k)]/2, \text{ where } 1 \le k \le n \quad (2.13)$$

By applying Eq. (2.13) on the data of Table 2.2 we find that: g = (1+1+1+0) + (2+2+1+1)/2 = (0+0+1+2) + (2+2+1+1)/2 = 6.

To write a more general relation than Eq. (2.13) which can cover all the cases of final sports tournament tables, we assume that there is an n-participant tournament, where C_1 and C_2 contain q pairs and r elements, respectively. Thus, the general relation can be given as follows:

$$g = [\sum_{u=1}^{q} \sum_{k=1}^{n} R_{2u-1}(k)] + [\sum_{v=1}^{r} \sum_{k=1}^{n} R_{2q+v}(k)]/2 \quad (2.14)$$

$k=1 \quad 2q+v$

Table 2.5 shows the application of Eq. (2.14) for the final tournament tables presented in Section 1.8.3:

Table 2.5. Application of Eq. (2.14) for the final tournament tables given in Section 1.8.3

Sports	C_1	C_2	g
Football, rugby, handball, chess	$\{\{W, L\}\}$	$\{D\}$	$[\sum_{k=1}^{n} W(k)] + [\sum_{k=1}^{n} D(k)]/2 = [\sum_{k=1}^{n} L(k)] + [\sum_{k=1}^{n} D(k)]/2$
Basketball, volleyball, tennis, lacrosse	$\{\{W, L\}\}$	$\{\}$	$[\sum_{k=1}^{n} W(k)] = [\sum_{k=1}^{n} L(k)]$
Ice hockey	$\{\{W, L\}, \{OTW, OTL\}\}$	$\{\}$	$[\sum_{k=1}^{n} W(k)] + [\sum_{k=1}^{n} OTW(k)] = [\sum_{k=1}^{n} L(k)] + [\sum_{k=1}^{n} OTL(k)]$
Curling	$\{\{W, L\}, \{SOW, SOL\}\}$	$\{\}$	$[\sum_{k=1}^{n} W(k)] + [\sum_{k=1}^{n} SOW(k)] = [\sum_{k=1}^{n} L(k)] + [\sum_{k=1}^{n} SOL(k)]$
Cricket	$\{\{W, L\}\}$	$\{T, NR\}$	$[\sum_{k=1}^{n} W(k)] + [[\sum_{k=1}^{n} T(k)] + [\sum_{k=1}^{n} NR(k)]]/2 = [\sum_{k=1}^{n} L(k)] + [[\sum_{k=1}^{n} T(k)] + [\sum_{k=1}^{n} NR(k)]]/2$

2.3.3. Total Number of Points

The participants of a sport tournament get a specific number of points for each game result. In the case of football tournaments, for example, a team at the *k*th position of the final table gets 3 points for a won game, 1 point for a drawn game, and none for a lost game. In this case, the total number of points, $P(k)$, gained by a team at the *k*th position is calculated based on the following relation:

$$Pts(k) = 3W(k) + D(k), \text{ where } 1 \le k \le n \tag{2.15}$$

To generalize Eq. (2.15), we suppose that P_i is the number of points which is gained from a game in R_i, where $1 \le i \le 2q+r$. In this case, Eq. (2.15) can be generalized as follows:

$$Pts(k) = \sum_{i=1}^{2q+r} R_i(k) * P_i, \text{ where } 1 \le k \le n \tag{2.16}$$

Additionally, the total points scored by a participant at the kth position must be greater or equal to those scored by the next participant (i.e., the one at the (k+1)th position), resulting in the following relation must be valid:

$$Pts(k) \geq Pts(k+1), \text{ where } 1 \leq k \leq n - 1 \tag{2.17}$$

There is a case when the number of the won games is taken into account as the main metric to rank the participants as in the sport of lacrosse, which is shown in Section 1.8.3.2. In this circumstance, the number of the games, $W(k)$, won by every participant is considered as the number of points, $Pts(k)$.

2.3.4. Set of the Possible Games Results

For a 4-team round-robin basketball tournament, the general set of the possible W and L values (WL) that might be gained by a team is $S_{WL}(4) = \{(3, 0), (2, 1), (0, 2), (0, 3)\}$. In this case, there are 4 possible values for WL. Generally, the set $S_{WL}(n)$ can be generated as follows:

$$S_{WL}(n) = \{(w, l) \mid \forall\, w \in \{0, 1, \dots, n - 1\},\ \forall\, l \in \{0, 1, \dots, n - 1\}, \\ w + l = n - 1\} \tag{2.18}$$

In a round-robin football tournament with 4 teams, the general set of the possible W, D and L values (WDL) that might be gained by a team will be $S_{WDL}(4) = \{(3, 0, 0), (2, 1, 0), (2, 0, 1), (1, 2, 0), (1, 1, 1), (1, 0, 2), (0, 3, 0), (0, 2, 1), (0, 1, 2), (0, 0, 3)\}$. In this case, there are ten possible values for WDL. In general, $S_{WDL}(n)$ can be generated as follows:

$$S_{WDL}(n) = \{(w, d, l) \mid \forall\, w \in \{0, 1, \dots, n - 1\},\ \forall\, d \in \{0, 1, \dots, n - 1\}, \\ \forall\, l \in \{0, 1, \dots, n - 1\}, w + l = n - 1\} \tag{2.19}$$

For a tournament with n teams and a number of possible game results equals to $2q+r$, the set $S(n, 2q+r)$ can be generated as follows:

$$S(n, 2q+r) = \{\{r_1, r_2, ..., r_{2q+r}\} \mid \forall\, r_1 \in \{0, 1, ... , n - 1\}, \forall\, r_2 \in \{0, 1, ... , n - 1\}, ..., \forall\, r_{2q+r} \in \{0, 1, ... , n - 1\}, r_1 + r_2 + ... + r_{2q+r} = n - 1\} \quad (2.20)$$

We suppose that the columns of possible game results are positioned in the final tournament table as (R_1, R_2, …, R_{2q+r}). In this way, the elements in the set S can be determined through some simple arithmetic operations. For example, the first element of S corresponds to the one with $R_1 = n\text{-}1$ and $R_i = 0$, where $2 \leq i \leq 2q+r$; that is, the first element has the form (n-1, 0, ..., 0). Based on the first element of S, the other elements can be generated, which will have the form (n-1-j, j, 0, ..., 0), where $1 \leq j \leq n$-1. To differentiate between the elements, we suppose that the first element is at layer 1, while the elements generated from this one are at layer 2. The elements of layer 2 can be used to generate the elements of another layer (layer 3) with the same principle, where R_2 and R_3 will be taken into account instead of R_1 and R_2. By applying the same process on each new layer, all the elements of S will be generated up to layer $2q+r$.

The process of generating the elements of S can be represented by a tree data structure, where the root node of the tree contains the element at layer 1 and the leaves contain the elements of layer $2q+r$. Figure 2.3 shows the corresponding tree of the set S(4, 2), while Figure 2.4 represents the corresponding tree of S(4, 3).

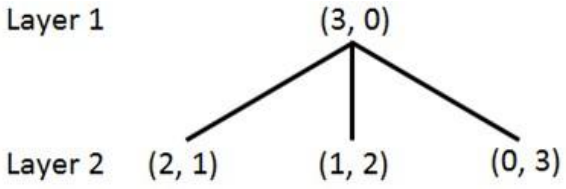

Figure 2.3. The corresponding tree of S(4, 2)

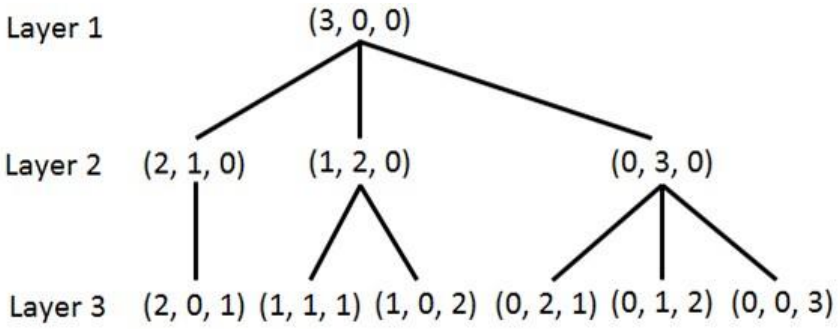

Figure 2.4. The corresponding tree of $S(4, 3)$

The number of the elements of S can be calculated via the number of the nodes of its corresponding tree. It is obvious that the number of the nodes in the tree presented in Figure 2.3 is 4, which means that $|S(4, 2)| = 4$. The value of $|S(4, 2)|$ can be calculated from the fact that there is only one node in layer 1 (root), and 3 branches are outgoing from that node, where each branch leads to a leaf. In this case $|S(n, 2)|$ can be written as:

$$|S(n, 2)| = 1 + n - 1 = 1 + \sum_{i=1}^{n-1} 1 \tag{2.21}$$

When the number of game results is greater than two, Eq. (2.21) will not be of much practical use, and there will be a need to use a recursive function to cover all the possible nodes. Such a recursive function must involve two parameters, where one of them indicates the number of layer (i.e. tree depth), which is also an index to a game result in the final tournament table. The other parameter indicates the value of the game result which is currently handled. Let us suppose that a function f calculates the number of all the nodes which can be generated based on a specific node in a specific layer. At a layer l, where the value of R_l is r_l, the number of the nodes that can be generated based on the related node can be calculated using the function f as follows:

$$f(l, r_l) = 1 + \sum_{v=1}^{r_l} f(l+1, v)$$
$$f(2q+r, x) = 1, \text{ where } 1 \le x \le n\text{-}1 \tag{2.22}$$

To calculate the number of the elements in a set $S(n, 2q+r)$ of an n-participant tournament table with $2q+r$ possible game results, it will be enough to call the function f with the related parameters to the root (l=1 and r_l=n-1). In this case, we can write $|S(n, 2q+r)|$ as:

$$|S(n, 2q+r)| = f(1, n\text{-}1) \tag{2.23}$$

The number of the nodes in the tree given in Figure 2.4 ($|S(4, 3)|$) can be calculated using Eq. (2.23) as follows:

$|S(4, 3)| = f(1, 3)$

$f(1, 3) = 1 + f(2, 1) + f(2, 2) + f(2, 3)$

$f(2, 1) = 1 + f(3, 1)$

$f(2, 2) = 1 + f(3, 1) + f(3, 2)$

$f(2, 3) = 1 + f(3, 1) + f(3, 2) + f(3, 3)$

$f(3, 1) = f(3, 2) = f(3, 3) = 1$

$f(2, 1) = 1 + 1 = 2$

$f(2, 2) = 1 + 1 + 1 = 3$

$f(2, 3) = 1 + 1 + 1 + 1 = 4$

$|S(4, 3)| = f(1, 3) = 1 + 2 + 3 + 4 = 10$

Table 2.6 shows the application of Eq. (2.23) for the final tournament tables presented in Section 1.8.3:

Table 2.6. Application of Eq. (2.23) for the final tournament tables presented in Section 1.8.3

Sports	C_1	C_2	$\|S(n, 2q+r)\|$
Football, rugby, handball, chess	$\{\{W, L\}\}$	$\{D\}$	$\|S(n, 3)\| = \sum_{i=1}^{n} i = n(n+1)/2$
Basketball, volleyball, tennis, lacrosse	$\{\{W, L\}\}$	$\{\}$	$\|S(n, 2)\| = \sum_{i=1}^{n} 1 = n$
Ice hockey	$\{\{W, L\}, \{OTW, OTL\}\}$	$\{\}$	$\|S(n, 4)\| = \sum_{j=1}^{n} \sum_{i=1}^{j} i = \sum_{j=1}^{n} [j(j+1)/2]$
Curling	$\{\{W, L\}, \{SOW, SOL\}\}$	$\{\}$	$\|S(n, 4)\| = \sum_{j=1}^{n} \sum_{i=1}^{j} i = \sum_{j=1}^{n} [j(j+1)/2]$
Cricket	$\{\{W, L\}\}$	$\{T, NR\}$	$\|S(n, 4)\| = \sum_{j=1}^{n} \sum_{i=1}^{j} i = \sum_{j=1}^{n} [j(j+1)/2]$

2.3.5. Game Result Cases

In a tournament, a single game can have $2q+r$ different results, namely R_1, R_2, ... R_{2q+r}, where q is the number of the pairs of C_1 and r is the number of the elements of C_2. We call each result case a possible result of played games. In a tournament of n participants, each participant can achieve one of $(2q+r)^{n-1}$ different result cases. Given a football tournament with $n = 4$ teams, for a single team, the set of the possible result cases would contain $3^{4-1} = 27$ different result cases.

Tables 2.7 and 2.8 show respectively the possible result cases which can be achieved by a participant playing 3 games in the case of tournament tables with *WL* and *WDL* game results, while Table 2.9 shows the possible result cases which can be achieved by a participant playing the same number of games in the case of tournament tables with *WXYL* and *WLTNr* game results. In the case of *WL*, for example, the result case of (1, 1, 2) means that the first and the second games end in a win, and the last game ends in a loss, while the result case of (1, 3, 2) in the case of *WDL* means that the first game ends in a win, the second ends in a draw, and the last one ends in a loss. The resulting case of (4, 1, 3) in the case of *WLTNr* means that the first game ends with no results, the second game ends in a win, and the last one ends in a tie, while the same result case in the case of *WXYL* means that the first game ends in *Y* (*OTL* or *SOL*), the second one ends in a win, and the last ends in *X* (*OTW* or *SOW*).

Table 2.7. Possible result cases in a 4-participant tournament with game results of *WL*

WL	**Result cases**
(3, 0)	(1, 1, 1)
(2, 1)	(1, 1, 2), (1, 2, 1), (2, 1, 1)
(1, 2)	(1, 2, 2), (2, 1, 2), (2, 2, 1)
(0, 2)	(2, 2, 2)

Table 2.8. Possible result cases in a 4-participant tournament with game results of *WDL*

WDL	**Result cases**
(3, 0, 0)	(1, 1, 1)
(2, 1, 0)	(1, 1, 3), (1, 3, 1), (3, 1, 1)
(2, 0, 1)	(1, 1, 2), (1, 2, 1), (2, 1, 1)
(1, 2, 0)	(1, 3, 3), (3, 1, 3), (3, 3, 1)
(1, 1, 1)	(1, 3, 2), (1, 2, 3), (3, 1, 2), (3, 2, 1), (2, 1, 3), (2, 3, 1)
(1, 0, 2)	(1, 2, 2), (2, 1, 2), (2, 2, 1)
(0, 3, 0)	(3, 3, 3)
(0, 2, 1)	(3, 3, 2), (3, 2, 3), (2, 3, 3)
(0, 1, 2)	(3, 2, 2), (2, 3, 2), (2, 2, 3)
(0, 0, 3)	(2, 2, 2)

Table 2.9. Possible result cases in a 4-participants tournament with game results of *WXYL* and *WLTNr*

WXYL* / *WLTNr	**Result cases in the case of *WXYL***	**Result cases in the case of *WLTNr***
(3, 0, 0, 0)	(1, 1, 1)	(1, 1, 1)
(2, 1, 0, 0)	(1, 1, 3), (1, 3, 1), (3, 1, 1)	(1, 1, 2), (1, 2, 1), (2, 1, 1)
(2, 0, 1, 0)	(1, 1, 4), (1, 4, 1), (4, 1, 1)	(1, 1, 3), (1, 3, 1), (3, 1, 1)
(2, 0, 0, 1)	(1, 1, 2), (1, 2, 1), (2, 1, 1)	(1, 1, 4), (1, 4, 1), (4, 1, 1)
(1, 2, 0, 0)	(1, 3, 3), (3, 1, 3), (3, 3, 1)	(1, 2, 2), (2, 1, 2), (2, 2, 1)
(1, 1, 1, 0)	(1, 3, 4), (1, 4, 3), (3, 1, 4), (3, 4, 1), (4, 1, 3), (4, 3, 1)	(1, 3, 2), (1, 2, 3), (3, 1, 2), (3, 2, 1), (2, 1, 3), (2, 3, 1)
(1, 1, 0, 1)	(1, 3, 2), (1, 2, 3), (3, 1, 2), (3, 2, 1), (2, 1, 3), (2, 3, 1)	(1, 4, 2), (1, 2, 4), (4, 1, 2), (4, 2, 1), (2, 1, 4), (2, 4, 1)
(1, 0, 2, 0)	(1, 4, 4), (4, 1, 4), (4, 4, 1)	(1, 3, 3), (3, 1, 3), (3, 3, 1)
(1, 0, 1, 1)	(1, 4, 2), (1, 2, 4), (4, 1, 2), (4, 2, 1), (2, 1, 4), (2, 4, 1)	(1, 3, 4), (1, 4, 3), (3, 1, 4), (3, 4, 1), (4, 1, 3), (4, 3, 1)
(1, 0, 0, 2)	(1, 2, 2), (2, 1, 2), (2, 2, 1)	(1, 4, 4), (4, 1, 4), (4, 4, 1)
(0, 3, 0, 0)	(3, 3, 3)	(2, 2, 2)
(0, 2, 1, 0)	(3, 3, 4), (3, 4, 3), (4, 3, 3)	(3, 2, 2), (2, 3, 2), (2, 2, 3)
(0, 2, 0, 1)	(3, 3, 2), (3, 2, 3), (2, 3, 3)	(4, 2, 2), (2, 4, 2), (2, 2, 4)
(0, 1, 2, 0)	(3, 4, 4), (4, 3, 4), (4, 4, 3)	(3, 3, 2), (3, 2, 3), (2, 3, 3)
(0, 1, 1, 1)	(3, 4, 2), (3, 2, 4), (4, 3, 2), (4, 2, 3), (2, 3, 4), (2, 4, 3)	(3, 2, 4), (3, 4, 2), (4, 3, 2), (4, 2, 3), (2, 3, 4), (2, 4, 3)
(0, 1, 0, 2)	(3, 2, 2), (2, 3, 2), (2, 2, 3)	(4, 4, 2), (4, 2, 4), (2, 4, 4)
(0, 0, 3, 0)	(4, 4, 4)	(3, 3, 3)
(0, 0, 2, 1)	(4, 4, 2), (4, 2, 4), (2, 4, 4)	(3, 3, 4), (3, 4, 3), (4, 3, 3)
(0, 0, 1, 2)	(4, 2, 2), (2, 4, 2), (2, 2, 4)	(3, 4, 4), (4, 3, 4), (4, 4, 3)
(0, 0, 0, 3)	(2, 2, 2)	(4, 4, 4)

It is quite possible that a single game result can be generated by many different result cases. As seen in Table 2.7, there are 1 or 3 result cases for a single value of *WL*, while in Tables 2.8 and 2.9, there are 1, 3 or 6 result cases for a single value of *WDL*,

WXYL or *WLTNr*. The number of the result cases based on a particular one of possible game results for a *k*th participant playing *n*-1 games is simply given by:

$$N_{RC}(k) = (R_1+R_2+ \ldots +R_{2q+r})!/(R_1!\ R_2!\ \ldots\ R_{2q+r}!)$$
$$= (n\text{-}1)!/(R_1!\ R_2!\ \ldots\ R_{2q+r}!) \qquad (2.24)$$

Table 2.10 shows the application of Eq. (2.24) for the tournament tables presented in Section 1.8.3:

Table 2.10. Application of Eq. (2.24) in the case of the tournament tables presented in Section 1.8.3

Sports	C_1	C_2	$N_{RC}(k)$
Football, rugby, handball, chess	{{*W*, *L*}}	{*D*}	(*n*-1)!/(*W*! *L*! *D*!)
Basketball, volleyball, tennis, lacrosse	{{*W*, *L*}}	{}	(*n*-1)!/(*W*! *L*!)
Ice hockey	{{*W*, *L*}, {*OTW*, *OTL*}}	{}	(*n*-1)!/(*W*! *L*! *OTW*! *OTL*!)
Curling	{{*W*, *L*}, {*SOW*, *SOL*}}	{}	(*n*-1)!/(*W*! *L*! *SOW*! *SOL*!)
Cricket	{{*W*, *L*}}	{*T*, *NR*}	(*n*-1)!/(*W*! *L*! *T*! *NR*!)

As one can see, result cases are considerably different from the values of R_1, R_2, ..., R_{2q+r}, although both can consist of the same digits. A result case refers to one possible result of each played game, while the values of R_1, R_2, ..., R_{2q+r} represent the number of games which was ended in R_1, R_2, ..., R_{2q+r}. So, the size of a result case (i.e., the number of its digits) equals to the number of the played games.

The notion of result case helps to estimate the maximum size of the search space for the problem of finding all possible tournaments graphs, and therefore the tables of the games results. Let us firstly investigate the cases in which there occurs the maximum number of result cases. In a tournament with *n* participants, each participant plays totally *n*-1 games which end up with one of result cases counted by Eq. (2.24). The values of (R_1, R_2, ..., R_{2q+r}) obviously vary within the range of (*n*-1, 0, ..., 0) to (0, 0, ..., *n*-1),

possibly generating different numbers of result cases. From Eq. (2.24) we observe that the largest number of the result cases would occur when R_1, R_2, ..., R_{2q+r} values take the same or very close values. For example, for n = 10, a team in a football tournament with the *WDL* value of (3, 3, 3) would cause the maximum number of 9!/(3!3!3!) = 1680 result cases, while a team with *WL* value of (5, 4) or (4, 5) in a basketball tournament would cause the maximum number of 9!/(5!4!) = 9!/(4!5!) = 126 result cases. On the other hand, when all the games end in a result which belongs to C_2 (i.e., R_{2q+1}, R_{2q+2}, ..., or R_{2q+r}) the participants receive the same R_1, R_2, ..., R_{2q+r} values. In this case there is only one result case for each participant.

In order to find the possible tournament graphs, a blind search algorithm needs to select and evaluate all the result cases of the teams in the worst case. So the total number of the resulting selections can be computed by the relation:

$$B_{RC}(n) = \prod_{k=0}^{n} {}^{N}_{N}(k) = (1)_{RC} * (N)_{RC} * (2)_{RC} * \ldots * (N)_{RC} * (n)_{RC} \quad (2.25)$$

However, given the fact that the result cases of the teams are tied to each other, after a result case of a team is selected, the number of the result case selections of the remaining teams would be less. This implies that there is no need for iterations over all the elements in $B_{RC}(n)$ for obtaining the possible game results.

2.3.6. Uniform Final States

When the set C_2 is not empty and all the games are ended in R_j, where $R_j \in C_2$, all the participants will have the same values of game results that lead to a uniform distribution of *WDL*s at the final tournament table. Thus, they will have the same number of points. For example, in a football tournament, if all games are ended in a draw, all the n participating teams will have the same values of *WDL*, with the general form (0, n-1, 0).

In actual fact, there are other cases in which each participant has the same values of R_1, R_2, ..., R_{2q+r}. We define a uniform final state of the tournament table to be a state in which all the teams have the same values of R_1, R_2, ..., R_{2q+r}. The set of such values in the case of a football tournament can be generated by:

$$SU_{football}(n) = \{(x, n\text{-}2x\text{-}1, x) \mid \forall\, x \in \{0, 1, 2, \ldots, \lfloor (n\text{-}1)/2 \rfloor\}\} \quad (2.26)$$

To generalize Eq. (2.26) to be suitable for other sports where the set C_2 is not empty, each component of a pair from C_1 take the value x, and an element from C_2 takes the value n-2x-1, while all the remaining elements of game results are zero. In this case the Eq. (2.26) can be written as:

$$SU(n) = \{\{x, n\text{-}2x\text{-}1, x, 0, \ldots, 0\} \mid \forall\, x \in \{0, 1, 2, \ldots, \lfloor (n\text{-}1)/2 \rfloor\}\} \quad (2.27)$$

The number of the elements in the set SU ($|SU|$) can be calculated by using the following relation:

$$\begin{aligned} |SU(n)| &= r(q(n\text{-}3)/2\text{+}1)\text{+}q, \text{ where } n \text{ is odd} \\ &= r(q(n\text{-}2)/2\text{+}1), \text{ where } n \text{ is even} \end{aligned} \quad (2.28)$$

Given that there is a tournament T_1 with C_1={{R_1, R_2}, {R_3, R_4}} and C_2={R_5, R_6}, the order of the possible game results in the final tournament table is R_1, R_3, R_5, R_6, R_4 and R_2. The possible uniform final states of the tournament table in the case of football, rugby, handball and chess are shown in Tables 2.11. Table 2.12 presents the possible uniform states of the final table in the case of cricket tournaments, while Tables 2.13 shows the uniform states of the final table of T_1. The number of the participants, n, ranges between 2 and 8 in each tournament.

Table 2.11. The possible uniform states of football, rugby, handball and chess final tournament tables

n	***WDL***
2	(0, 1, 0)
3	(0, 2, 0), (1, 0, 1)
4	(0, 3, 0), (1, 1, 1)
5	(0, 4, 0), (1, 2, 1), (2, 0, 2)
6	(0, 5, 0), (1, 3, 1), (2, 1, 2)
7	(0, 6, 0), (1, 4, 1), (2, 2, 2), (3, 0, 3)
8	(0, 7, 0), (1, 5, 1), (2, 3, 2), (3, 1, 3)

Table 2.12. The possible uniform states of cricket final tournament table

n	***WTLNR***
2	(0, 0, 1, 0), (0, 0, 0, 1)
3	(0, 0, 2, 0), (1, 1, 0, 0), (0, 0, 0, 2)
4	(0, 0, 3, 0), (1, 1, 1, 0), (0, 0, 0, 3), (1, 1, 0, 1)
5	(0, 0, 4, 0), (1, 1, 2, 0), (2, 2, 0, 0), (0, 0, 0, 4), (1, 1, 0, 2)
6	(0, 0, 5, 0), (1, 1, 3, 0), (2, 2, 1, 0), (0, 0, 0, 5), (1, 1, 0, 3), (2, 2, 0, 1)
7	(0, 0, 6, 0), (1, 1, 4, 0), (2, 2, 2, 0), (3, 3, 0, 0), (0, 0, 0, 6), (1, 1, 0, 4), (2, 2, 0, 2)
8	(0, 0, 7, 0), (1, 1, 5, 0), (2, 2, 3, 0), (3, 3, 1, 0), (0, 0, 0, 7), (1, 1, 0, 5), (2, 2, 0, 3), (3, 3, 0, 1)

Table 2.13. The possible uniform states of the final tournament table T_1

n	$\boldsymbol{R_1R_3R_5R_6R_4R_2}$
2	(0, 0, 1, 0, 0, 0), (0, 0, 0, 1, 0, 0)
3	(0, 0, 2, 0, 0, 0), (1, 0, 0, 0, 0, 1), (0, 1, 0, 0, 1, 0), (0, 0, 0, 2, 0, 0)
4	(0, 0, 3, 0, 0, 0), (1, 0, 1, 0, 0, 1), (0, 1, 1, 0, 1, 0), (0, 0, 0, 3, 0, 0), (1, 0, 0, 1, 0, 1), (0, 1, 0, 1, 1, 0)
5	(0, 0, 4, 0, 0, 0), (1, 0, 2, 0, 0, 1), (2, 0, 0, 0, 0, 2), (0, 1, 2, 0, 1, 0), (0, 2, 0, 0, 2, 0), (0, 0, 0, 4, 0, 0), (1, 0, 0, 2, 0, 1), (0, 1, 0, 2, 1, 0)
6	(0, 0, 5, 0, 0, 0), (1, 0, 3, 0, 0, 1), (2, 0, 1, 0, 0, 2), (0, 1, 3, 0, 1, 0), (0, 2, 1, 0, 2, 0), (0, 0, 0, 5, 0, 0), (1, 0, 0, 3, 0, 1), (2, 0, 0, 1, 0, 2), (0, 1, 0, 3, 1, 0), (0, 2, 0, 1, 2, 0)
7	(0, 0, 6, 0, 0, 0), (1, 0, 4, 0, 0, 1), (2, 0, 2, 0, 0, 2), (3, 0, 0, 0, 0, 3), (0, 1, 4, 0, 1, 0), (0, 2, 2, 0, 2, 0), (0, 3, 0, 0, 3, 0), (0, 0, 0, 6, 0, 0), (1, 0, 0, 4, 0, 1), (2, 0, 0, 2, 0, 2), (0, 1, 0, 4, 1, 0), (0, 2, 0, 2, 2, 0)
8	(0, 0, 7, 0, 0, 0), (1, 0, 5, 0, 0, 1), (2, 0, 3, 0, 0, 2), (3, 0, 1, 0, 0, 3), (0, 1, 5, 0, 1, 0), (0, 2, 3, 0, 2, 0), (0, 3, 1, 0, 3, 0), (0, 0, 0, 7, 0, 0), (1, 0, 0, 5, 0, 1), (2, 0, 0, 3, 0, 2), (3, 0, 0, 1, 0, 3), (0, 1, 0, 5, 1, 0), (0, 2, 0, 3, 2, 0), (0, 3, 0, 1, 3, 0)

When C_2 is empty, comparing with the previous case, the number of states where all the participants have the same game results may be none or small. For example, in a

basketball tournament, $|SU(n)| = 1$ when n is odd ($SU(n) = \{(n\text{-}1)/2, (n\text{-}1)/2)\}$), while $|SU(n)| = 0$ when n is even. Also, in the case of ice hockey and curling, $|SU(n)| = 2$ when n is odd ($SU(n) = \{((n\text{-}1)/2, 0, 0, (n\text{-}1)/2), (0, (n\text{-}1)/2, (n\text{-}1)/2, 0))\}$, while also $|SU(n)| = 0$ when n is even. When the set C_2 is empty ($r = 0$), based on the Eq. (2.28), the number of the uniform final states equals to 0 when n is even, while it equals to q when n is odd. In this case, the set of the game results which form the possible uniform tournament tables can be represented as:

$$SU(n) = \{\{n/2, n/2, 0, \dots, 0\} \mid n \text{ is odd}\} \tag{2.29}$$

Given that there is a tournament T_2 with $C_1=\{\{R_1, R_2\}, \{R_3, R_4\}, \{R_5, R_6\}\}$ and $C_2=\{\}$, the order of the possible game results in the final tournament table is R_1, R_3, R_5, R_6, R_4, and R_2. The possible uniform states of the final tables in the case of basketball, ice hockey, curling and T_2 are shown in Table 2.14, where the number of teams, n, ranges between 2 and 8.

Table 2.14. The possible uniform states of the final tables in the case of basketball, ice hockey, curling and T_2

n	**Basketball (*WL*)**	**Ice hockey and curling (*WXYL*)**	**T_2 ($R_1R_3R_5R_6R_4R_2$)**
2	None	None	None
3	(1, 1)	(1, 0, 0, 1), (0, 1, 1, 0)	(1, 0, 0, 0, 0, 1), (0, 1, 0, 0, 1, 0), (0, 0, 1, 1, 0, 0)
4	None	None	None
5	(2, 2)	(2, 0, 0, 2), (0, 2, 2, 0)	(2, 0, 0, 0, 0, 2), (0, 2, 0, 0, 2, 0), (0, 0, 2, 2, 0, 0)
6	None	None	None
7	(3, 3)	(3, 0, 0, 3), (0, 3, 3, 0)	(3, 0, 0, 0, 0, 3), (0, 3, 0, 0, 3, 0), (0, 0, 3, 3, 0, 0)
8	None	None	None

Table 2.15 shows the application of Eq. (2.27), (2.28) and (2.29) for the sports tournaments presented in Section 1.8.3.

Table 2.15. Application of Eq. (2.27), (2.28) and (2.29) for the sports tournaments presented in Section 1.8.3

Sports	$SU(n)$	$\lvert SU(n) \rvert$
Football, rugby, handball, chess	$\{(x, n\text{-}2x\text{-}1, x) \mid \forall\, x \in \{0, 1, 2, ..., \lfloor (n\text{-}1)/2 \rfloor\}\}$	$\lceil n/2 \rceil$
Basketball, volleyball, tennis, lacrosse	$\{((n\text{-}1)/2, (n\text{-}1)/2)\}$	n is odd: 1 n is even: 0
Ice hockey, Curling	$\{((n\text{-}1)/2, 0, 0, (n\text{-}1)/2), (0, (n\text{-}1)/2, (n\text{-}1)/2, 0)\}$	n is odd: 2 n is even: 0
Cricket	$\{(x, x, n\text{-}2x\text{-}1, 0) \mid \forall\, x \in \{0, 1, 2, ..., \lfloor (n\text{-}1)/2 \rfloor\}\} \cup$ $\{(x, x, 0, n\text{-}2x\text{-}1) \mid \forall\, x \in \{0, 1, 2, ..., \lfloor (n\text{-}1)/2 \rfloor\}\}$	$2*\lceil n/2 \rceil$

2.4. Graph Representation

The round-robin tournament graph is defined as a directed graph in Section 1.7, where its nodes present the participants, while its edges represent the game results between every two participants. The edges are directed from the winner participant to the loser one. The graphs are perfectly suitable for tournaments with each game result of either a win or a loss, which, for example, can be clearly seen in the representation of basketball tournaments. The tournament graph of Group A in the 2014 FIBA Basketball World Cup [166] is shown in Figure 2.5, where the group contains the teams of Spain (ESP), Brazil (BRA), France (FRA), Serbia (SRB), Iran (IRI), and Egypt (EGY). Note that the final tournament table in Table 2.4 can be easily obtained from the tournament graph in Figure 2.5.

The set C_2 is empty in the case of the final table of basketball tournaments, but it is not in other sports, which means that a directed tournament graph is not applicable to define game results of any sports tournament. For example, in the case of a round-robin football tournament, where $C_2=\{D\}$, it is not possible to define a drawn game with a directed edge.

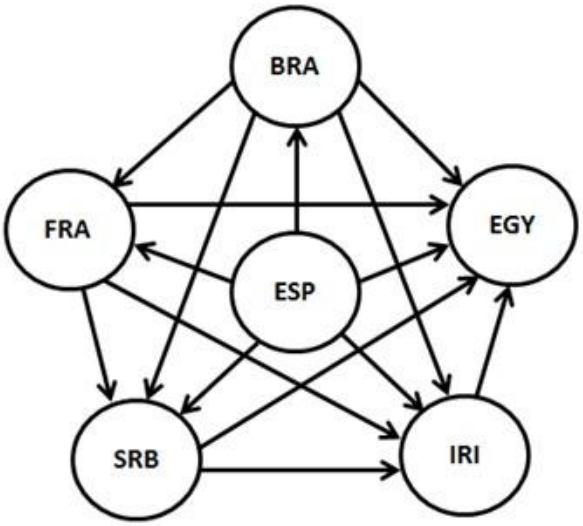

Figure 2.5. The tournament graph of Group A in the 2014 FIBA World Cup

In this section, we propose another definition for the tournament graph, where in addition to the representation of the won/lost games with directed edges, the drawn games will be represented with undirected ones. Figure 2.6 shows the tournament graph of Group B in the 2018 FIFA World Cup [165], participated in by Spain (ESP), Portugal (POR), Iran (IRI), and Morocco (MAR). The tournament graph of this tournament is shown in Figure 2.6, which represents the data in the final tournament table given in Table 2.2.

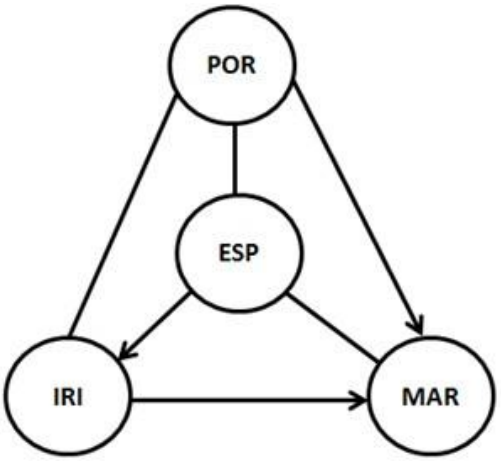

Figure 2.6. The tournament graph Group B in the 2018 FIFA World Cup

The set C_1 in the case of football and basketball contains a single pair of game results (i.e., $\{W, L\}$), but in other cases such as ice hockey and curling, C_1 contains more than a single pair. Given an ice hockey tournament, when a team *A* ends the game with an *OTW*, the other team *B* automatically ends its game with an *OTL*. It means that this result can be represented by the graph with a directed edge going from *A* to *B*.

The problem with such a representation is that it will not be possible to distinguish between the edges which represent the results of *W*/*L* and the ones which represent the results of *OTW*/*OTL*. As a solution to this problem, we propose to color the edges of the graph, where a possible game result in each pair from C_1 is given a specific color. As mentioned in Section 1.6.7, coloring an edge is just a manner to distinguish between that edge and the other edges, which means that the edges can be represented in other ways such as drawing them with different line dashes, or labeling them with specific characters.

Figure 2.7 shows the tournament graph of Group A of the ice hockey 2018 Olympic Winter Games, Men's Tournament [167], participated in by Czech (CZE), Canada (CAN), Switzerland (SUI), and South Korea (KOR), where a result of *W*/*L* is represented with a continued edge, while a result of *OTW*/*OTL* is represented with a discontinued edge.

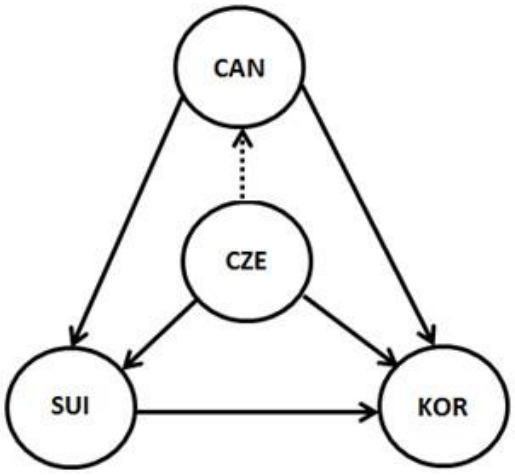

Figure 2.7. The tournament graph of Group A in the ice hockey 2018 Olympic Winter Games, Men's Tournament

Similarly, when C_2 contains more than one game result as in the tournaments of cricket, the related results can be represented with colored edges. Figure 2.8 shows the tournament graph of Group B in the 2008 ICC World Cricket League Division Five

[168], participated in by Jersey (JEY), Afghanistan (AFG), Singapore (SGP), Botswana (BWA), Japan (JPN) and Bahamas (BHS), where a result of *T* is represented with continued edge, while a result of *NR* is represented with discontinued edge.

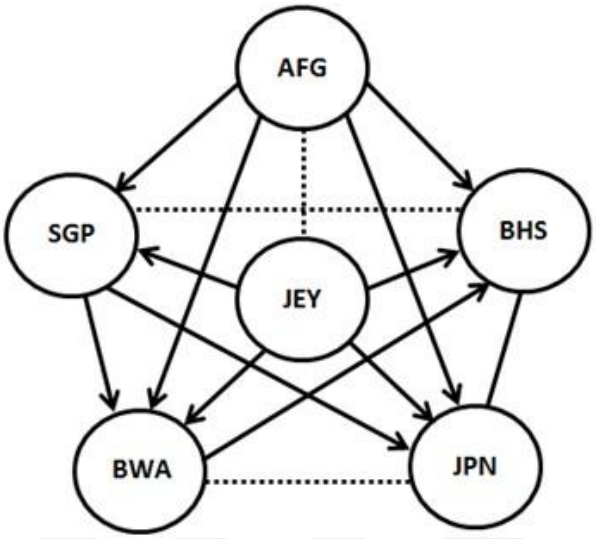

Figure 2.8. The tournament graph of Group B in the 2008 ICC World Cricket League Division Five

In the tournament T_1 which was addressed in the previous section, remember that C_1 contains two pairs and C_2 contains two elements; that is, the possible game results are $C_1=\{\{R_1, R_2\}, \{R_3, R_4\}\}$ and $C_2=\{R_5, R_6\}$. Figure 2.9 illustrates a tournament graph which corresponds to the data of Table 2.16, representing a state of the final table of T_1 between the participants *A, B, C* and *D*. In the graph, a result of R_1/R_2 is shown with a continued directed edge and a result of R_3/R_4 is shown with a discontinued directed edge, while a result of R_5 is shown with a continued undirected edge and a result of R_6 is shown with a discontinued undirected edge.

Table 2.16. A state of the final table of a tournament T_1 between 4 participants

Participants	R_1	R_3	R_5	R_6	R_4	R_2
A	2	1	0	0	0	0
B	0	1	1	0	1	0
C	0	0	1	1	0	1
D	0	0	0	1	1	1

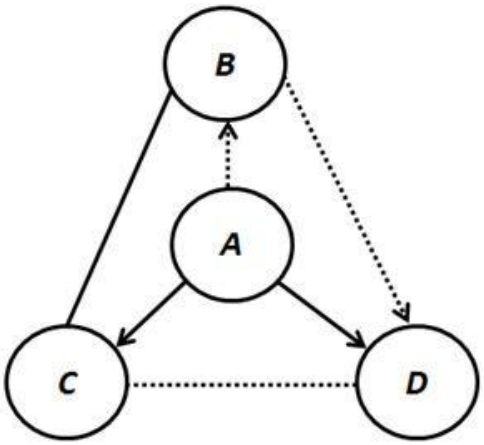

Figure 2.9. A tournament graph corresponding to the data of Table 2.16

2.5. Determination of Game Results

We describe, in this section, how to determine the game results based on a tournament final table. The tournaments of football and some other sports hold similar final tables that make it easier to calculate a possible result of a game. Unlike sports events like basketball tournaments, C_2 is not empty in a football tournament. Besides, on the contrary to other sports like cricket, the number of game results in the final table of a football tournament is lesser. Let us start with the problem of determining game results using the tournament final table of Group E in the 2006 FIFA World Cup [169] which is shown in Table 2.17. The group contains the teams Italy (ITA), Ghana (GHA), Czech Republic (CZE) and United States (USA).

Table 2.17. The final tournament table of Group E in the 2006 FIFA World Cup

Teams	***W***	***D***	***L***
ITA	2	1	0
GHA	2	0	1
CZE	1	0	2
USA	0	1	2

Based on the *WDL* values of each team, we aim to determine the tournament graphs which lead to the game results. Since C_1 contains one pair (W, L) and C_2 contains one element (D), there will not be a necessity for the edge coloring. In Table 2.17, the related data to Italy and the United States provides a convenient starting point of the manual solution, because both of them have one draw. Since the other teams have no draw, that means Italy must have drawn with the United States. The fact that Italy has two wins means that it must have won its remaining two games against Ghana and Czech. Now that Ghana lost against Italy, it must have won its remaining games against Czech and The United States. As losing against Italy and Ghana, Czech's remaining game must be against the United States, which must be ended in a win. Figure 2.10 presents the tournament graph of this case, while Table 2.18 shows the game results, where 1, 2 and 3 represent a win, loss and draw, respectively.

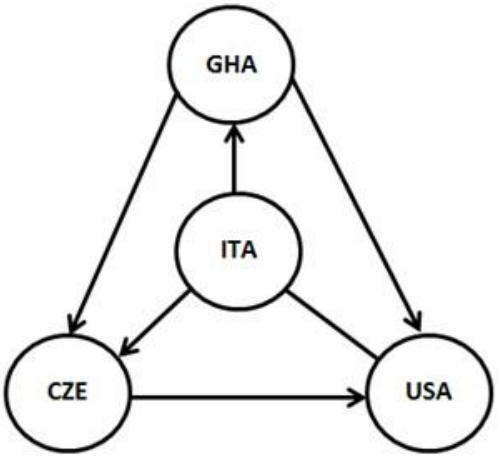

Figure 2.10. The tournament graph of the data in Table 2.17

Table 2.18. The game results of the tournament graph in Figure 2.10

Teams	ITA	GHA	CZE	USA
ITA	0	1	1	3
GHA	2	0	1	1
CZE	2	2	0	1
USA	3	2	2	0

For the final tournament table shown in Table 2.17, there is only one possible tournament graph. However, this might not be the case all the time. To illustrate the situation, we modify the *WDL* values of Ghana and the United States to be (1, 1, 1) and (1, 0, 2), respectively. In the FIFA World Cup history, this modified state occurred in Group 3 in the 1962 World Cup [170], participated in by Brazil (BRA), Czechoslovakia (CSK), Mexico (MEX) and Spain (ESP). In this case, there are two possible tournament graphs which are shown in Figures 11 and 12.

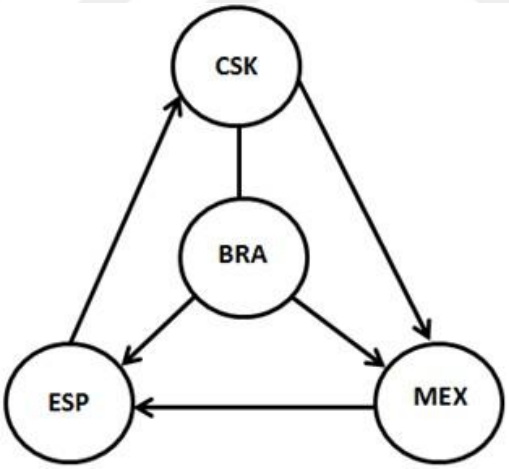

Figure 2.11. First possible tournament graph of Group 3 in 1962 FIFA World Cup

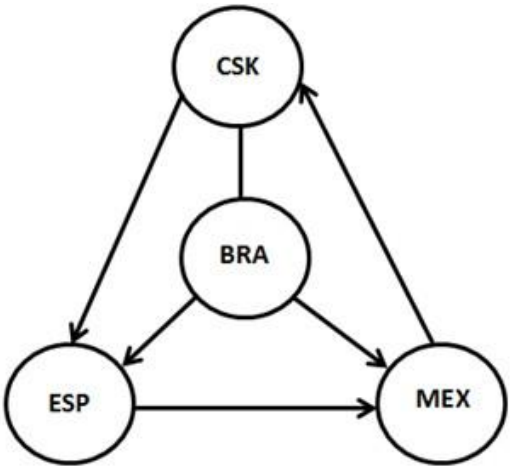

Figure 2.12. Second possible tournament graph of Group 3 in 1962 FIFA World Cup

The tournament final table shows only the number of game results, but not which other teams it wins against and draws with. In a group with four participants, it may be easy to find the game results of teams. However, this gets more difficult when the number of participants increases, which makes manual solution methods inapplicable. Table 2.19 illustrates how to select the edges in the football tournament graph and their directions based on the data in Table 2.17. Figure 2.13 shows the obtained tournament graph as described in Figure 2.10, where the edges are numbered based on the steps in which they are selected during the process in Table 2.19.

Table 2.19. Generation steps of a tournament graph for the data of Table 2.17

Step	**ITA**	**GHA**	**CZE**	**USA**	**Game**	**Edge**
1	(2, 1, 0)	(2, 0, 1)	(1, 0, 2)	(0, 1, 2)	ITA-GHA	-
2	(2, 1, 0)	(2, 0, 1)	(1, 0, 2)	(0, 1, 2)	ITA- CZE	-
3	(2, 1, 0)	(2, 0, 1)	(1, 0, 2)	(0, 1, 2)	ITA-USA	Undirected
4	(2, 0, 0)	(2, 0, 1)	(1, 0, 2)	(0, 0, 2)	ITA-GHA	From ITA to GHA
5	(1, 0, 0)	(2, 0, 0)	(1, 0, 2)	(0, 0, 2)	ITA-CZE	From ITA to CZE
6	(0, 0, 0)	(2, 0, 0)	(1, 0, 1)	(0, 0, 2)	GHA-CZE	From GHA to CZE
7	(0, 0, 0)	(1, 0, 0)	(1, 0, 0)	(0, 0, 2)	GHA-USA	From GHA to USA
8	(0, 0, 0)	(0, 0, 0)	(1, 0, 0)	(0, 0, 1)	CZE-USA	From CZE to USA
9	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	-	-

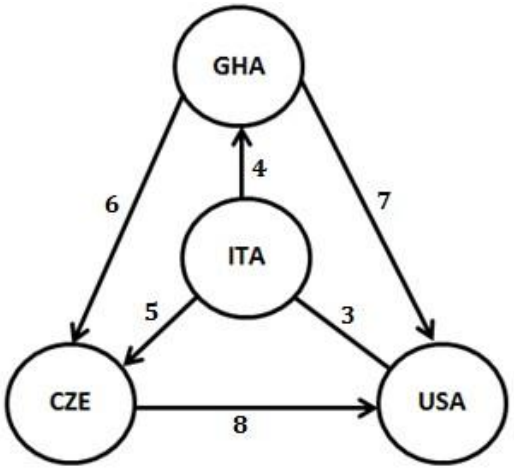

Figure 2.13. Generation of the tournament graph in Figure 2.10 through the steps numbered in Table 2.19

In Table 2.19, the edge between each two-node is selected by comparing the *WDL* values of the related teams. The related digits of the *WDL* values from which the edge selection is made are underlined at each step. According to the ascending order of the related *W*, *D* and *L* values for each team, the edge selections are conducted. The draw of Italy, for example, is first sought, because its value of *D* equals to 1. Only the United States among the other teams has a *D* value which is not 0. For that, an undirected edge is selected to connect the representative node of Italy to the one of the United States. To show the consequences of the selection, the relevant *WDL* values are gradually reduced after a valid edge selection is made. At step 4, for example, concerning the selected edge for the game between Italy and Ghana, a win is taken out of the *WDL* value of Italy and a loss is taken out of the *WDL* value of Ghana.

As discussed in Section 2.3.5, the possible partitions of *WDL* values in the tournament table for a *k*th team determines the maximum number of the selection of its edges (see Eq. (2.24)). The edge selection also depends on the result of the games played with the other teams. For example, As Table 2.20 shows, the number of possible selections for Italy is: N_{RC}(ITA) = 3!/(2!1!0!) = 3, but the fact that only the United States has a draw allows a unique selection for the game result of Italy.

Table 2.20. Possible results for the games of Italy

Games	Case 1	Case 2	Case 3
ITA- GHA	1	1	3
ITA-CZE	1	3	1
ITA-USA	3	1	1

Given the *WDL* values of all the teams in Table 2.17, according to Eq. (2.25), the total number of edge selections would blindly be:

$$B_{RC}(4) = N_{RC}(\text{ITA}) * N_{RC}(\text{GHA}) * N_{RC}(\text{CZE}) * N_{RC}(\text{USA}) = 3*3*3*3 = 81$$

By starting with the team whose N_{RC} value is minimum, and selecting the results based on the ascending order of its game results, the number of the selections can be kept smaller. Since the teams in Table 2.17 have the same N_{RC} value, the process can start arbitrarily with any team. For example, when the result of the game ITA-USA is selected as a draw, only $N_{RC}(\text{ITA}) = 2!/(2!0!0!) = 1$ possible selection remains for the results of the remaining games of Italy (ITA-GHA and ITA-CZE). The same occurs for the United States, where there is only $N_{RC}(\text{US}) = 2!/(0!0!2!) = 1$ selection for the results of the two remaining games (USA-GHA and USA-CZE). Thus, the actual number of the edge selections would be much less than 81.

With the same principle applied in Table 2.19, Table 2.21 shows how to obtain the tournament graphs in Figures 2.11 and 2.12. It can be seen in Table 2.21 that the value of *W* for Czechoslovakia after step 3 (steps 4_A and 4_B) equals to 1, and the value of *L* for both Mexico and Spain is equal to 1, which means that Czechoslovakia could win against Spain or win against Mexico. After taking both of these scenarios into account, the resulting tournament graphs are shown in Figures. 2.14 and 2.15, respectively.

Table 2.21. The steps followed to obtain the graphs in Figures 2.11 and 2.12

Step	BRA	CSK	MEX	ESP	Game	Edge
1	(2, 1, 0)	(1, 1, 1)	(1, 0, 2)	(1, 0, 2)	BRA-CSK	Undirected
2	(2, 0, 0)	(1, 0, 1)	(1, 0, 2)	(1, 0, 2)	BRA-MEX	From BRA to MEX
3	(1, 0, 0)	(1, 0, 1)	(1, 0, 1)	(1, 0, 2)	BRA-ESP	From BRA to ESP
4_A	(0, 0, 0)	(1, 0, 1)	(1, 0, 1)	(1, 0, 1)	CSK-MEX	From CSK to MEX
5_A	(0, 0, 0)	(0, 0, 1)	(1, 0, 0)	(1, 0, 1)	CSK-ESP	From ESP to CSK
6_A	(0, 0, 0)	(0, 0, 0)	(1, 0, 0)	(0, 0, 1)	MEX-ESP	From MEX to ESP
7_A	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	-	-
4_B	(0, 0, 0)	(1, 0, 1)	(1, 0, 1)	(1, 0, 1)	CSK-ESP	From CZE to ESP
5_B	(0, 0, 0)	(0, 0, 1)	(1, 0, 1)	(1, 0, 0)	CSK-MEX	From MEX to CZE
6_B	(0,0, 0)	(0, 0, 0)	(0, 0, 1)	(1, 0, 0)	MEX-ESP	From ESP to MEX
7_B	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	-	-

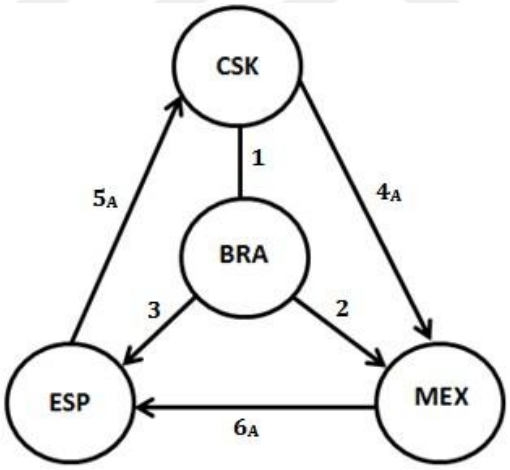

Figure 2.14. Generation of the tournament graph in Figure 2.11 through the steps numbered in Table 2.21

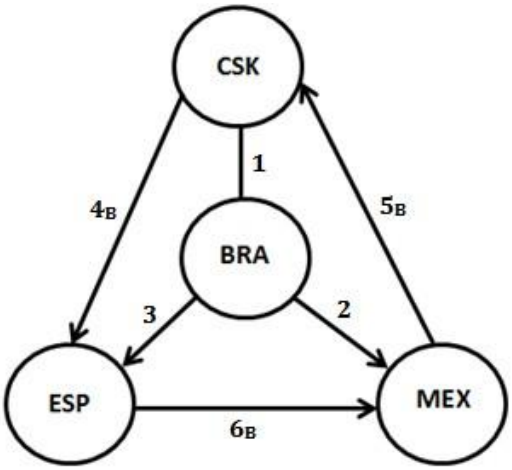

Figure 2.15. Generation of the tournament graph in Figure 2.11 through the steps numbered in Table 2.21

The same principle applied for the football tables can also be used for more complicated tournament tables. For example, we suppose that there are four participants are competing in a tournament of T_1, which is presented in Section 2.3.6, where $C_1=\{\{R_1, R_2\}, \{R_3, R_4\}\}$, $C_2=\{R_5, R_4\}$. In this case, coloring the edges is compulsory. The representative edges of the pair $\{R_1, R_2\}$ and the element R_5 are drawn in continued lines, while the representative edges of the pair $\{R_3, R_4\}$ and the element R_6 are drawn in dotted lines. Using a similar principle applied in Tables 2.19 and 2.21, Table 2.22 illustrates the way to obtain the possible tournament graphs based on the data of Table 2.16. At step 4_B the process reaches a situation where the value of R_1 for A is equal to 1, but there is no other participant with $R_2>0$ except D, but an edge between A and D was previously selected in step 1_B, subsequently making it impossible to reach a tournament graph from step 4_B. The tournament graph obtained through the steps numbered in Table 2.22 is shown in Figure 2.16, labeling each edge with their related steps.

Table 2.22. The steps followed to obtain the graph in Figure 2.16

Step	***A***	***B***	***C***	***D***	**Game**	**Edge**
1_A	(2, 1, 0, 0, 0, 0)	(0, 1, 1, 0, 1, 0)	(0, 0, 1, 1, 0, 1)	(0, 0, 0, 1, 1, 1)	*A-B*	Dotted from *A* to *B*
2_A	(2, 0, 0, 0, 0, 0)	(0, 1, 1, 0, 0, 0)	(0, 0, 1, 1, 0, 1)	(0, 0, 0, 1, 1, 1)	*A-C*	Continued from *A* to *C*
3_A	(1, 0, 0, 0, 0, 0)	(0, 1, 1, 0, 0, 0)	(0, 0, 1, 1, 0, 0)	(0, 0, 0, 1, 1, 1)	*A-D*	Continued from *A* to *D*
4_A	(0, 0, 0, 0, 0, 0)	(0, 1, 1, 0, 0, 0)	(0, 0, 1, 1, 0, 0)	(0, 0, 0, 1, 1, 0)	-	-
5_A	(0, 0, 0, 0, 0, 0)	(0, 1, 1, 0, 0, 0)	(0, 0, 1, 1, 0, 0)	(0, 0, 0, 1, 1, 0)	*B-D*	Dotted from *B* to *D*
6_A	(0, 0, 0, 0, 0, 0)	(0, 0, 1, 0, 0,0)	(0, 0, 1, 1, 0, 0)	(0, 0, 0, 1, 0, 0)	*B-C*	Continued undirected
7_A	(0, 0, 0, 0, 0, 0)	(0, 0, 0, 0, 0, 0)	(0, 0, 0, 1, 0, 0)	(0, 0, 0, 1, 0, 0)	*C-D*	Dotted undirected
8_A	(0, 0, 0, 0, 0,0)	(0, 0, 0, 0, 0, 0)	(0, 0, 0, 0, 0, 0)	(0, 0, 0, 0, 0, 0)	-	-
1_B	(2, 1, 0, 0, 0, 0)	(0, 1, 1, 0, 1, 0)	(0, 0, 1, 1, 0, 1)	(0, 0, 0, 1, 1, 1)	*A-D*	Dotted from *A* to *D*
2_B	(2, 0, 0, 0, 0, 0)	(0, 1, 1, 0, 1, 0)	(0, 0, 1, 1, 0, 1)	(0, 0, 0, 1, 0, 1)	*A-B*	-
3_B	(2, 0, 0, 0, 0, 0)	(0, 1, 1, 0, 1, 0)	(0, 0, 1, 1, 0, 1)	(0, 0, 0, 1, 0, 1)	*A-C*	Continued from *A* to *C*
4_B	(1, 0, 0, 0, 0, 0)	(0, 1, 1, 0, 1, 0)	(0, 0, 1, 1, 0, 0)	(0, 0, 0, 1, 0, 1)	-	-

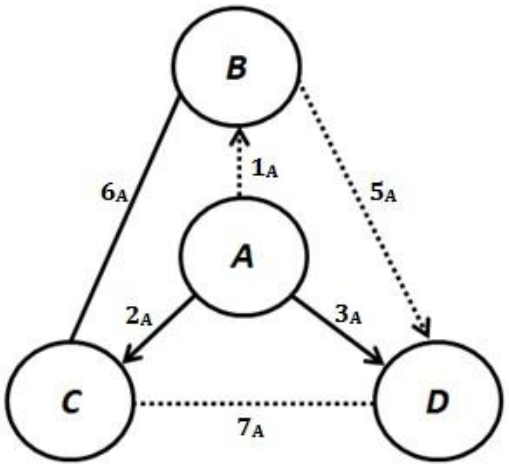

Figure 2.16. Generation of the tournament graph based on Table 2.16, through the steps numbered in Table 2.22

As proposed by Damkhi and Pehlivan [61], the following algorithm shows a generalized pseudo-code which describes the proposed approach:

Algorithm for determining game results

```
Inputs:
   n:  number of participants
   T:  final tournament table
   C1: game results which are related to each other
   q:  size of C1
   C2: game results which are not related to other game results
   r:  size of C2

Outputs:
   Gs[][n][n]: possible tournament graphs

Static data structures:
   G[n][n]: tournament graph

Main Procedure
1:  set all G[i][j] elements to 0
2:  trackingInds[2q+r]
3:  for i ← 1 to 2q+r do
4:    trackingInds[i] ← 2
5:  graphSearch(1, trackingInds)

Procedure: graphSearch(k, trackingInds_p[])
1:  trackingInds[2q+r]
2:  trackingInds ← trackingInds_p
3:  find the non-zero minimum Ri value for kth participant and
    set an index variable (minInd) to the position of Ri in T, or
    -1 if all is 0
4:  if minInd ≠ −1 then
5:    minInd
6:    if Ri ∈ C2 then
```

```
7:      minIndp ← minInd
8:    else
9:      minIndp ← the index of the pair of Ri in T
10:   for i ← trackingInds[minInd] to n do
11:     if T[i][minIndp] > 0 and G[k][i] = 0 then
12:       T[k][minInd]--
13:       T[i][minIndp]--
14:       G[k][i] ← minInd
15:       G[i][k] ← minIndp
16:       trackingInds[minInd] ← i + 1
17:       graphSearch(k, trackingInds)
18:       G[k][i] ← 0
19:       G[i][k] ← 0
20:       T[k][minInd]++
21:       T[i][minIndp]++
22: else
23:   if k = n then
24:     if T elements are all 0 then
25:       insert G into Gs
26:   else
27:     set all the elements of trackingInds to k+2
28:     graphSearch(k+1, trackingInds)
```

In the algorithm, if the participants at a lower position than k cannot satisfy the non-zero minimum of the values R_1, R_2, ..., R_{2q+r} belonging to the kth participant, the values R_1, R_2, ..., R_{2q+r} for the kth participant would never be zeros, and therefore the participant at the (k+1)th position would not be considered. When the initial data of the tournament final table are not valid (for example, if it does not satisfy Eq. (2.11)), there would be a team which cannot have the resulting R_1, R_2, ..., R_{2q+r} values of zeros as the algorithm runs, which means that the condition of the procedure graphSearch at line 24 would never be satisfied and thus there will not be a game result to be added into G. To traverse through all game results, where all the results are unique, the algorithm backtracks with the restoration of both T and G to their previous state through the loop at line 10 of the procedure graphSearch (lines 18, 19, 20 and 21).

2.6. Enumeration of Final Table States

2.6.1. Backward Algorithm

This algorithm aims to generate the possible states of the final table of a single round-robin tournament between n participants. The generation of the states in this instance is based on the elements of the set S which is previously defined by Eq. (2.20).

To enumerate the states of a tournament final table, based on the set S, the number of states that a blind search algorithm must evaluate would be related to the number of the participants and the number of the elements of S (see Eq. (2.23)). It follows the number of states which must be blindly evaluated would be:

$$NB(n, 2q+r) = (|S(n, 2q+r)|)^n \tag{2.30}$$

The following algorithm shows the pseudo-code which describes the blind search algorithm:

Blind Search Algorithm for enumerating the final table states

```
Inputs:
   n:  number of participants
   C1: game results which are related to each other
   q:  size of C1
   C2: game results which are not related to other game results
   r:  size of C2

Output:
   Ts[][n][2q+r]: states of the final tournament table

Static variables and data structures:
   N:          number of elements of the set S(n, 2q+r)
   S[N][2q+r]: set S(n, 2q+r)
   T[n][2q+r]: tournament table

Main Procedure
1:  calculate N based on Eq. (2.23)
2:  calculate S based on Eq. (2.20)
3:  stateGenerator(1)

Procedure: stateGenerator(k)
1:  if k ≤ n then
2:    for i ← 1 to N do
3:      for j ← 1 to 2q+r do
4:        T[k][j] ← S[i][j]
5:      stateGenerator(k+1)
6:  else
7:    insert T into Ts
```

Most of the states generated by the blind search algorithm for a tournament final table do not satisfy Eq. (2.9), (2.10), (2.14) or (2.17). For example, in the case of a football tournament between 4 teams, the state presented in Table 2.23 is generated by the blind search algorithm but it does not respect any of those relations. To generate the states

of a tournament final table, only the valid states must be taken into account, where the validity of each generated state must be checked.

Table 2.23. A state generated by the blind search algorithm which does not respect any of Eq. (2.9), (2.10), (2.14) and (2.17)

Teams	***W***	***D***	***L***
A	2	1	0
B	0	0	3
C	1	1	1
D	0	1	2

To filter the states generated by the blind search algorithm, we propose to use a state validity checker to evaluate the feasibility of the states. The state validity checker is an adapted version of the algorithm which is previously proposed in Section 2.5 to determine the game results. It seeks to reach a tournament graph based on a specific state of a tournament final table. In case of a state leading to more than one graph, the process of seeking a graph will be ended as soon as the algorithm finds the first graph. Given Table 2.21, it will end at step 7_A.

If a state of a tournament final table is not valid, the algorithms will be ended after trying all the possibilities to find a tournament graph. Table 2.24 shows the steps involved in trying to find a tournament graph based on the state presented in Table 2.23.

Table 2.24. The steps followed to obtain a tournament graph for the state represented by Table 2.23

Step	***A***	***B***	***C***	***D***	**Game**	**Edge**
1_A	(2, 1, 0)	(0, 0, 3)	(1, 1, 1)	(0, 1, 2)	*A*-*C*	Undirected
2_A	(2, 0, 0)	(0, 0, 3)	(1, 0, 1)	(0, 1, 2)	*A*-*B*	From *A* to *B*
3_A	(1, 0, 0)	(0, 0, 2)	(1, 0, 1)	(0, 1, 2)	*A*-*D*	From *A* to *D*
4_A	(0, 0, 0)	(0, 0, 2)	(1, 0, 1)	(0, 1, 1)	*B*-*C*	From *C* to *B*
5_A	(0, 0, 0)	(0, 0, 1)	(0, 0, 1)	(0, 1, 1)	-	-
1_B	(2, 1, 0)	(0, 0, 3)	(1, 1, 1)	(0, 1, 2)	*A*-*D*	Undirected
2_B	(2, 0, 0)	(0, 0, 3)	(1, 1, 1)	(0, 0, 2)	*A*-*B*	From *A* to *B*
3_B	(1, 0, 0)	(0, 0, 2)	(1, 1, 1)	(0, 0, 2)	*A*-*C*	From *A* to *C*
4_B	(0, 0, 0)	(0, 0, 2)	(1, 1, 0)	(0, 0, 2)	*B*-*C*	From *C* to *B*
5_B	(0, 0, 0)	(0, 0, 1)	(0, 1, 0)	(0, 0, 2)	*B*-*D*	-

At steps 5_A and 5_B the process reaches a situation where the value of *L* for the participant *B* is equal to 1 but all the *W* values of the other teams are zeros. In this case, we can say that the algorithm forms just partial graphs based on the current state of the tournament final table, but not a tournament graph, and therefore, the state is not valid.

As proposed by Damkhi and Pehlivan [62], the following algorithm shows the pseudo-code which contains the proposed filter:

Algorithm for validating states

```
Inputs:
   n:  number of participants
   T:  final tournament table
   C1: game results which are related to each other
   q:  the size of C1
   C2: game results which are not related to other game results
   r:  the size of C2

Outputs:
   true or false

Static data structures:
   T[n][2q+r]: tournament table
   G[n][n]:    tournament graph

Main Procedure: stateChecker(T_p)
1:  set all G[i][j] elements to 0
```

```
2:  set all the elements of T_p into T
3:  trackingInds[2q+r]
4:  for i ← 1 to 2q+r do
5:    trackingInds[i] ← 2
6:  return isStateValid(1, trackingInds)

Procedure: isStateValid(k, trackingInds_p[])
1:  valid ← false
2:  trackingInds[2q+r]
3:  trackingInds ← trackingInds_p
4:  find the non-zero minimum Ri value for kth participant and
    set an index variable (minInd) to the position of Ri in T, or
    -1 if all is 0
5:  if minInd ≠ −1 then
6:    minIndp
7:    if Ri ∈ C2 then
8:      minIndp ← minInd
9:    else
10:     minIndp ← the index of the pair of Ri in T
11:   for i ← trackingInds[minInd] to n do
12:     if T[i][minIndp] > 0 and G[k][i] = 0 then
13:       T[k][minInd]--
14:       T[i][minIndp]--
15:       G[k][i] ← minInd
16:       G[i][k] ← minIndp
17:       trackingInds[minInd] ← i + 1
18:       valid ← isStateValid(k, trackingInds)
19:       if valid = true then
20:         go to 33
21:       else
22:         G[k][i] ← 0
23:         G[i][k] ← 0
24:         T[k][minInd]++
25:         T[i][minIndp]++
26: else
27:   if k = n then
28:     if T elements are all 0 then
29:       valid ← true
30:   else
31:     set all the elements of trackingInds to k+2
32:     valid ← isStateValid(k+1, trackingInds)
33: return valid
```

The main difference of the state validation algorithm from the game result determination algorithm is that, instead of building all the tournament graphs, it returns a Boolean value as soon as it finds a tournament graph. In other words, through a recursive call to the procedure isStateValid (see line 18), if a tournament graph is detected, the process is ended and a value of true is returned through line 33. In the case that the procedure does not return a value of true, each of arrays *G* and *T* turns back into their

previous states just before line 13 through the lines 22, 23, 24, and 25, and the algorithm continues to find a tournament graph through the loop at line 11.

The integration of the state validating algorithm with the blind search algorithm leads to the backward state generation approach, shortly called the backward algorithm. The algorithm first generates the states of a tournament final table, and backwardly checks them to determine if at least one tournament graph can be constructed for each of them. In this way, checking the blindly generated states via the validating algorithm, we generate all the valid final table states of sport tournaments. The backward algorithm is given below.

Backward approach algorithm for enumerating the states of a tournament final table:

```
Inputs:
   n:  number of participants
   C1: game results which are related to each other
   q:  size of C1
   C2: game results which are not related to other game results
   r:  size of C2

Outputs:
   Ts[][n][2q+r]: states of the final tournament table

Static variables and data structures:
   N:          the number of elements of the set S(n, 2q+r)
   S[N][2q+r]: the set S(n, 2q+r)
   T[n][2q+r]: tournament table

Main Procedure
1:  calculate N based on Eq. (2.23)
2:  calculate S based on Eq. (2.20)
3:  stateGenerator(1)

Procedure: stateGenerator(k)
1:  if k ≤ n then
2:    for i ← 1 to N do
3:      for j ← 1 to 2q+r do
4:        T[k][j] ← S[i][j]
5:      stateGenerator(k+1)
6:  else
7:    if the points of the teams are in a descending order then
8:      valid ← stateChecker(T)
9:      if valid = true then
10:       insert T into Ts
```

Instead of saving the state directly as it is done in the blind search algorithm, the participants' points are checked if they are in a descending order, then the state generated

by the above algorithm is checked by the procedure stateChecker (see line 8), and is not saved unless their validity is agreed upon.

2.6.2. Forward Algorithm

Unlike the backward approach, this approach seeks to enumerate the states of a tournament final table by building every possible tournament graph, then converting them to the corresponding states. From the perspective of a tournament participant, a game can result in an R_1, R_2, ... or R_{2q+r}. Since the total of the played games, g, is equal to $n(n\text{-}1)/2$ (see Eq. (2.4)), the total of game results (i.e., tournament graphs) that can occur during a round-robin tournament with n participants would be:

$$GR(n) = (2q+r)^{n(n-1)/2} \tag{2.31}$$

Eq. (2.31) determines the number of graphs that can be generated by a blind search algorithm. The following pseudo-code represents such a blind search algorithm which generates all the possible tournament graphs:

Blind Search Algorithm for generating a tournament graph

```
Inputs:
   n:  number of participants
   C1: game results which are related to each other
   q:  size of C1
   C2: game results which are not related to other game results
   r:  size of C2

Output:
   Gs[][n][n]: possible tournament graphs

Static data structure:
   G[n][n]: tournament graph

Main Procedure
1:  set all the elements of G to 0
2:  graphGenerator(1, 2)

Procedure: graphGenerator(k, j)
1:  for i ← j to 2q+r do
2:    G[k][j] ← i
3:    if {Ri, Ri+1} ∈ C1 then
4:      G[j][k] ← i+1

5:    else if {Ri-1, Ri} ∈ C1 then
6:      G[j][k] ← i-1
```

```
7:     else
8:       G[j][k] ← i
9:     if j+1 ≤ n then
10:      graphGenerator(k, j+1)
11:    else
12:      if k+1 < n then
13:        graphGenerator(k+1, k+2)
14:      else
15:        insert G into Gs
```

Let us suppose that there is a football tournament between 3 teams (*A*, *B* and *C*), where *A* appears the first in the final table, while *B* does the second, and *C* does the last. The graphs of the tournament which can be generated by the blind search algorithm would be 3^3=27. Figures 2.17, 2.18, 2.19, 2.20, 2.21, 2.22, 2.23, 2.24 and 2.25 illustrate the process of how to generate all the possible graphs of this tournament.

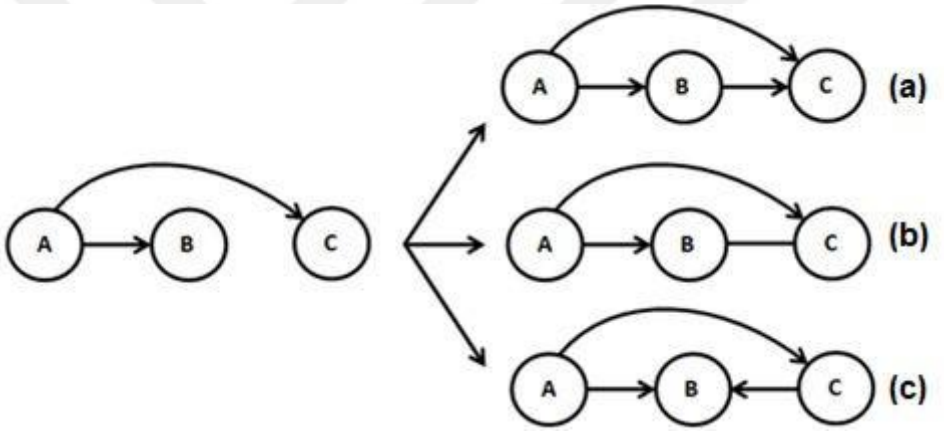

Figure 2.17. The graphs for the case when *A* wins all of its games

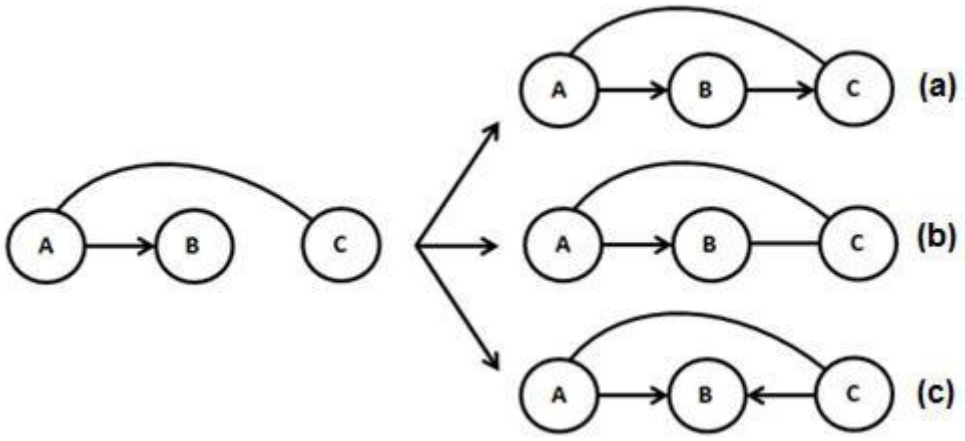

Figure 2.18. The graphs for the case when *A* wins against *B* and draws against *C*

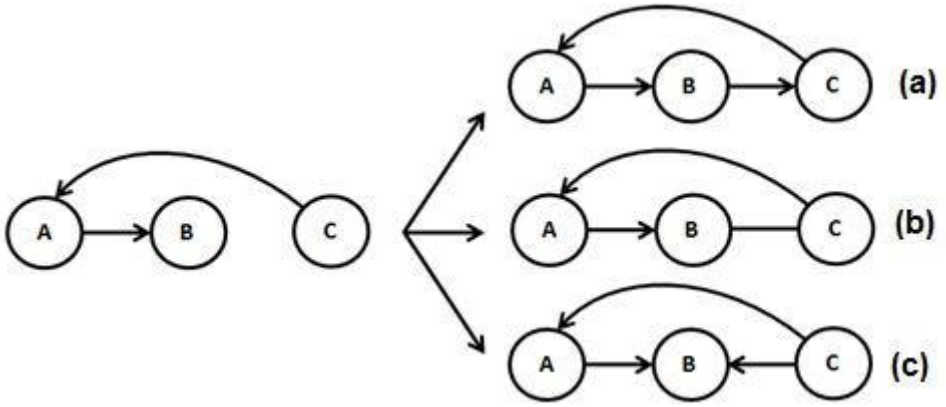

Figure 2.19. The graphs for the case when *A* wins against *B* and loses against *C*

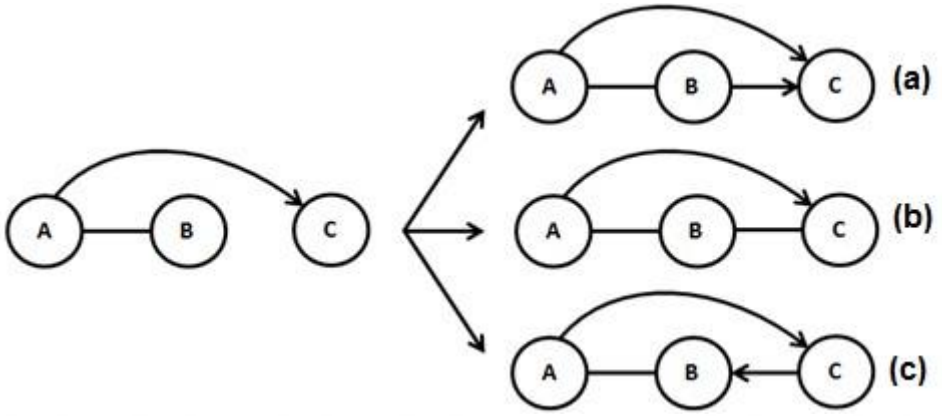

Figure 2.20. The graphs for the case when *A* draws against *B* and wins against *C*

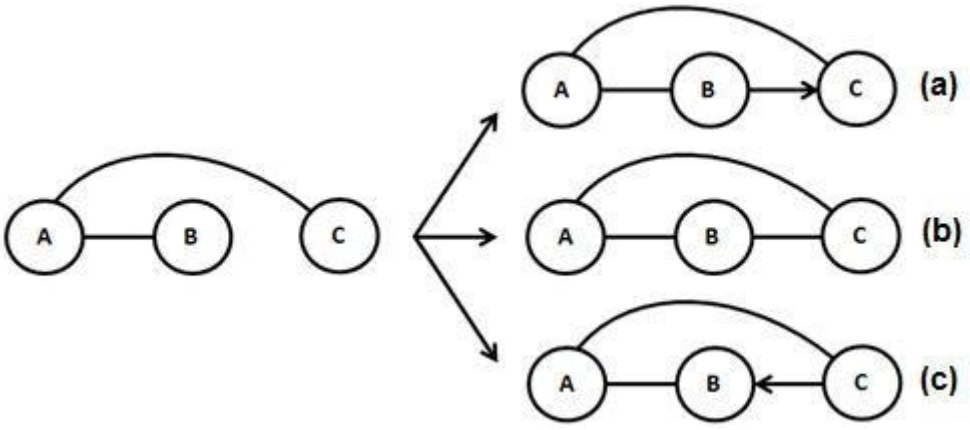

Figure 2.21. The graphs for the case when *A* draws all of its games

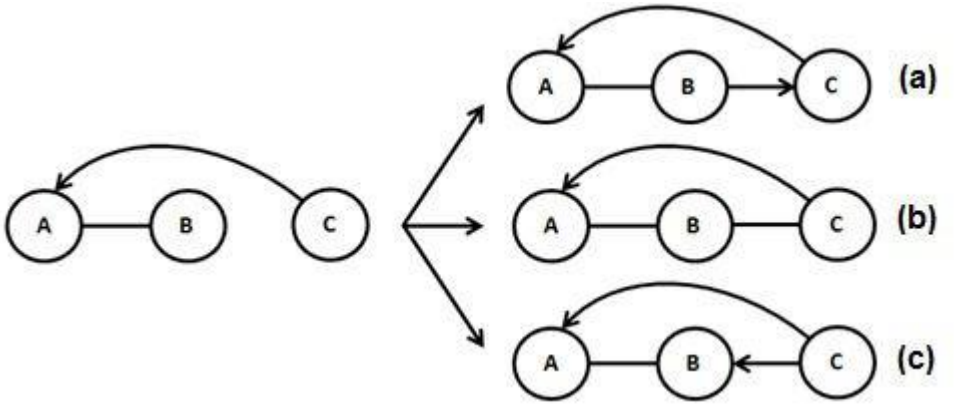

Figure 2.22. The graphs for the case when *A* draws against *B* and loses against *C*

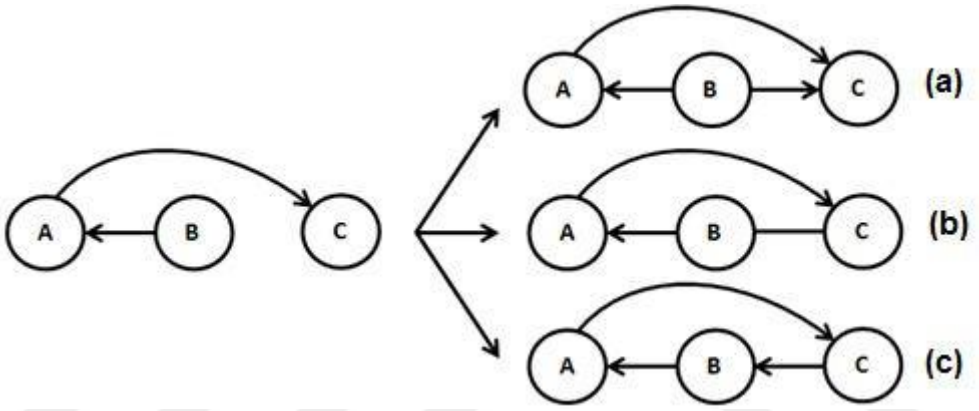

Figure 2.23. The graphs for the case when *A* loses against *B* and wins against *C*

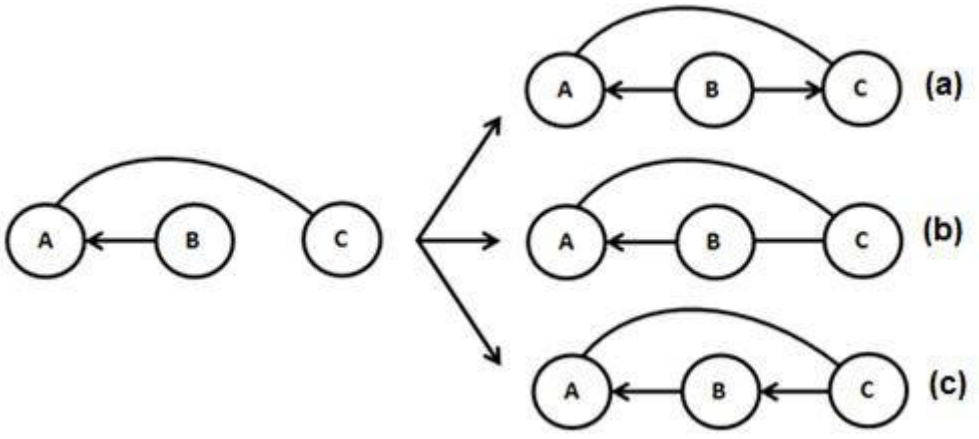

Figure 2.24. The graphs for the case when *A* loses against *B* and draws against *C*

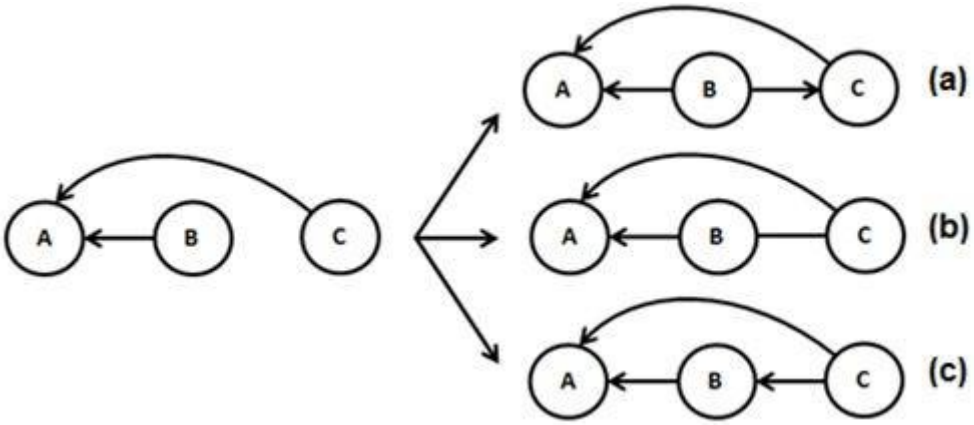

Figure 2.25. The graphs for the case when *A* loses all of its games

Table 2.25 shows the corresponding state of each tournament graph illustrated in Figures 2.17, 2.18, 2.19, 2.20, 2.21, 2.22, 2.23, 2.24 and 2.25. As seen in Table 2.25, only the graphs (a) and (b) from Figure 2.17, (a) from Figure 2.18, (a) from Figure 2.19, (a) and (b) from Figure 2.20, (b) from Figure 2.21, and (c) from Figure 2.23 lead to descending ordered states. This requires a check on the order of the resulting states before saving them. Besides, as it is discussed in Section 2.5, a state of a tournament final table could be resulted from many different tournament graphs, which explains why the graphs (a) from Figure 2.19 and (c) from Figure 2.23 led to the same table state.

Table 2.25. The corresponding state of each tournament graph shown in Figures 2.17, 2.18, 2.19, 2.20, 2.21, 2.22, 2.23, 2.24 and 2.25

Figure-Graph	**State**	**Status**	**Reason**
2.17-(a)	((2, 0, 0), (1, 0, 1), (0, 0, 2))	Accepted	-
2.17-(b)	((2, 0, 0), (0, 1, 1), (0, 1, 1))	Accepted	-
2.17-(c)	((2, 0, 0), (0, 0, 2), (1, 0, 1))	Not accepted	$Pts(2) < Pts(3)$
2.18-(a)	((1, 1, 0), (1, 0, 1), (0, 1, 1))	Accepted	-
2.18-(b)	((1, 1, 0), (0, 1, 1), (0, 2, 0))	Not accepted	$Pts(2) < Pts(3)$
2.18-(c)	((1, 1, 0), (0, 0, 2), (1, 1, 0))	Not accepted	$Pts(2) < Pts(3)$
2.19-(a)	((1, 0, 1), (1, 0, 1), (1, 0, 1))	Accepted	-
2.19-(b)	((1, 0, 1), (0, 1, 1), (1, 1, 0))	Not accepted	$Pts(2) < Pts(3)$
2.19-(c)	((1, 0, 1), (0, 0, 2), (2, 0, 0))	Not accepted	$Pts(1) < Pts(3)$
2.20-(a)	((1, 1, 0), (1, 1, 0), (0, 0, 2))	Accepted	-
2.20-(b)	((1, 1, 0), (0, 2, 0), (0, 1, 1))	Accepted	-
2.20-(c)	((1, 1, 0), (0, 1, 1), (1, 0, 1))	Not accepted	$Pts(2) < Pts(3)$
2.21-(a)	((0, 2, 0), (1, 1, 0), (0, 1, 1))	Not accepted	$Pts(1) < Pts(2)$
2.21-(b)	((0, 2, 0), (0, 2, 0), (0, 2, 0))	Accepted	-
2.21-(c)	((0, 2, 0), (0, 1, 1), (1, 1, 0))	Not accepted	$Pts(1) < Pts(3)$
2.22-(a)	((0, 1, 1), (1, 1, 0), (1, 0, 1))	Not accepted	$Pts(1) < Pts(2)$
2.22-(b)	((0, 1, 1), (0, 2, 0), (1, 1, 0))	Not accepted	$Pts(1) < Pts(2)$
2.22-(c)	((0, 1, 1), (0, 1, 1), (2, 0, 0))	Not accepted	$Pts(1) < Pts(3)$
2.23-(a)	((1, 0, 1), (2, 0, 0), (0, 0, 2))	Not accepted	$Pts(1) < Pts(2)$
2.23-(b)	((1, 0, 1), (1, 1, 0), (0, 1, 1))	Not accepted	$Pts(1) < Pts(2)$
2.23-(c)	((1, 0, 1), (1, 0, 1), (1, 0, 1))	Accepted	-
2.24-(a)	((1, 1, 1), (2, 0, 0), (0, 1, 1))	Not accepted	$Pts(1) < Pts(2)$
2.24-(b)	((1, 1, 1), (1, 1, 0), (0, 2, 0))	Not accepted	$Pts(1) < Pts(2)$
2.24-(c)	((0, 1, 1), (1, 0, 1), (1, 1, 0))	Not accepted	$Pts(1) < Pts(2)$
2.25-(a)	((0, 0, 2), (2, 0, 0), (1, 0, 1))	Not accepted	$Pts(1) < Pts(2)$
2.25-(b)	((0, 0, 2), (1, 1, 0), (1, 1, 0))	Not accepted	$Pts(1) < Pts(2)$
2.25-(c)	((0, 0, 2), (1, 0, 1), (2, 0, 0))	Not accepted	$Pts(1) < Pts(2)$

The possible duplicates of some resulting states require a check on the existence of every generated state. The simplest way to check the duplicate of a state is to store the states in an array, where a new state will be stored in that array only if it is not identical to any of the previously saved states. The use of the array trivially affects the performance when the number of the states is small. But as the number of tournament participants gets bigger, there occur so many states that increase the size of the array, which makes it time-consuming to compare a particular state to the others.

As a solution to this problem, we propose to save the states in a forest (i.e., a collection of trees) instead of a classic array, where the root of each tree from the forest would contain the R_1, R_2, ..., R_{2q+r} values which can be taken by the first participant of the tournament (i.e. $R_1(1)$, $R_2(1)$, ..., $R_{2q+r}(1)$). The states with the same values of $R_1(1)$, $R_2(1)$, ... , $R_{2q+r}(1)$ would be saved in the same tree. A state consists of all the nodes occurred in a path from a root to one of its leaves. The states which share the same values of $R_1(k)$, $R_2(k)$, ..., $R_{2q+r}(k)$ automatically share the path from the root to the node at a kth layer. Thus the number of the states of a tournament final table equals to the total number of leaves in all the trees of the forest. Figure 2.26 shows how the valid states listed in Table 2.25 can be represented as a forest.

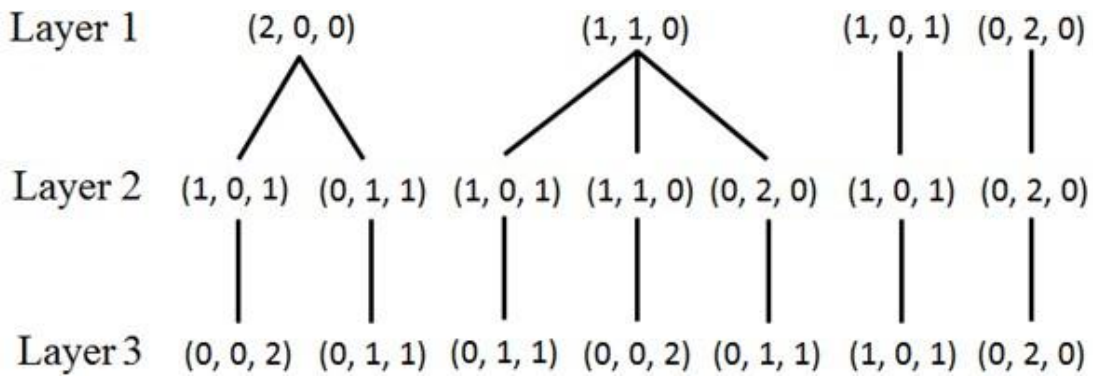

Figure 2.26. Representation of the valid states in Table 2.25 as a forest

Given the use of an array, when the state ((1, 0, 1), (1, 0, 1), (1, 0, 1)) (from the graph (c) in Figure 2.23) is generated, the array would have already contained seven states including the duplicate ones which were generated from the graph (a) in Figure 2.19. This state would be located at the fourth cell of the array, while, in the case of using a forest, the same state would be located at the third tree as seen in Figure 2.26. Similarly, when the state ((0, 2, 0), (0, 2, 0), (0, 2, 0)) from the graph (b) in Figure 2.21 is generated,

the array would have already contained a total of six other states. Thus the newly generated state would be compared with all the states contained in the array. In the case of using a forest, it would adequate to check the roots of the current trees to see if the state is not saved before.

In relation to the comparisons, saving the states in a forest instead of an array requires the less number of iterations to check the existence of a generated state. This decrease in the iterations may not seem effective when the number of states is small. In contrast, when the number of states gets bigger, saving them in the form of a forest will have a significant effect on the necessary time to determine whether the generated states are duplicated or not. The following algorithm shows the pseudo-code for a blind search based algorithm which implements the forward approach:

Forward approach algorithm for enumerating the states of a tournament final table:

```
Inputs:
   n:  number of participants
   C1: game results which are related to each other
   q:  size of C1
   C2: game results which are not related to other game results
   r:  size of C2

Output:
   Ts[][n][2q+r]: states of the final tournament table

Static data structures:
   G[n][n]:    tournament graph
   T[n][2q+r]: tournament table

Main Procedure
1:  set all the elements of G to 0
2:  graphGenerator(1, 2)

Procedure: graphGenerator(k, j)
1:  for i ← 1 to 2q+r do
2:    G[k][j] ← i
3:    if {Ri, Ri+1} ∈ C1 then
4:      G[j][k] ← i+1

5:    else if {Ri-1, Ri} ∈ C1 then
6:      G[j][k] ← i-1
7:    else
8:      G[j][k] ← i
9:    if j+1 ≤ n then
10:     graphGenerator(k, j+1)
11:   else
12:     if k+1 < n then
13:       graphGenerator(k+1, k+2)
14:     else
15:       save the equivalent state of G into T
```

```
16:       calculate the number of points for each participant
17:       if the participants' points are in a descending order
            then
18:         if T ∉ Ts then
19:           insert T into Ts
```

The algorithm directly focuses on the states of the tournament final table (see line 15), instead of the tournament graphs where, after generating a graph, its data is converted to the corresponding state of the tournament table. Following line 15, the points of the participants are calculated and verified whether they are in descending order or not (lines 16 and 17). If the points are not in descending order, the state will be ignored. Otherwise, it will be checked in line 18 and then be saved if it does not exist.

2.7. Search Space Analysis

In round-robin sport tournaments, for a particular position, there are highest and lowest numbers of points a participant can gain. Each participant holds a position in the final table with which it is possible to determine its highest and lowest numbers of points. The determination of the highest and lowest points of a participant helps the optimization of the search spaces for the proposed approaches.

During the generation of the states in the backward approach, if the point of a participant at a certain position does not range between a minimum number or a maximum number of points that are necessary for that position, the algorithm will not continue generating that state and thus its feasibility will not be checked. Similarly, in the case of the forward approach, if the game results of a participant at a specific position do not correspond to a point in the relevant range of possible points for that position, the process of generating the graph will stop, and the algorithm moves to the generation of the next graph.

To determine the minimum and maximum numbers of points which can be gained by a participant at the kth position of a single round-robin tournament, the uniform final tables, where all the participants obtain the same number of points, must be analyzed. For the uniform final tables, the minimum and maximum numbers of points are based mainly upon possible game results held in the sets C_1 and C_2 (see Section 2.3.6).

2.7.1. Uniform Final States With Minimum Number of Points

2.7.1.1. C_2 Is Not Empty

Suppose that there is a single round-robin tournament between n participants, where C_2 is not empty. A uniform state of the tournament final table is formed based on the columns R_i, R_{i+1}, and R_j of game results, where $\{R_i, R_{i+1}\}$ is a pair from C_1, R_j is an element from C_2, $i \in \{1, 3, ..., 2q\text{-}1\}$, and $j \in \{2q\text{+}1, 2q\text{+}2, ..., 2q\text{+}r\}$. Let us consider P_i, P_{i+1} and P_j as the number of points which can be gained by a participant from the games ending in R_i, R_{i+1} and R_j, respectively. In the case of a uniform state of the final table, the total number of points which is gained by every participant would be:

$$Pts(k) = P_i*x + P_j*(n\text{-}2x\text{-}1) + P_{i+1}*x, \text{ where } x \in \{0, 1, 2, ..., \lfloor (n\text{-}1)/2 \rfloor\} \quad (2.32)$$

which can be simplified to:

$$Pts(k) = (P_i\text{+}P_{i+1})x + P_j(n\text{-}2x\text{-}1), \text{ where } x \in \{0, 1, 2, ..., \lfloor (n\text{-}1)/2 \rfloor\} \quad (2.33)$$

Since each of x and n-2x-1 is positive, $Pts(k)$ reaches its minimum value when P_j becomes the minimum point of the elements in C_2, and $P_i + P_{i+1}$ becomes the minimum point of the pairs in C_1.

For the tournament T_1 which is previously presented in Section 2.3.6, let us suppose that P_1=7, P_2=0, P_3=5, P_4=1, P_5=3, and P_6=2 are the points which a participant gains from the games ending in R_1, R_2, R_3, R_4, R_5, or R_6 respectively. Given that the order of the possible results of the games in the final table of T_1 is R_1, R_3, R_5, R_6, R_4 and R_2, if the final table of T_1 has a uniform state, then the final data of all the participants would have one of the following forms: (x, 0, n-2x-1, 0, 0, x), (0, x, n-2x-1, 0, x, 0), (x, 0, 0, n-2x-1, 0, x), or

(0, x, 0, n-2x-1, x, 0), where $x \in \{0, 1, 2, ..., \lfloor (n\text{-}1)/2 \rfloor\}$. Since $\min(P_1\text{+}P_2, P_3\text{+}P_4) = P_3\text{+}P_4$ and $\min(P_5, P_6) = P_6$, the rows of the uniform state with minimum number of points would have the form (0, x, 0, n-2x-1, x, 0).

To find the values of x which minimizes the number of points in a uniform state of a final table, Eq. (2.33) can be written as follows:

$$Pts(k) = x(P_i+P_{i+1}-2P_j) + P_j(n\text{-}1), \text{ where } x \in \{0, 1, 2, ..., \lfloor (n\text{-}1)/2 \rfloor\} \quad (2.34)$$

The value of x which minimizes the value of $Pts(k)$ depends on the larger value of P_i+P_{i+1} and $2P_j$. If $P_i+P_{i+1} > 2P_j$, in order for the uniform state to reach the minimum number of points, x must be minimum (i.e. 0), while if $P_i+P_{i+1} < 2P_j$, for the uniform state to reach the minimum number of points, x must be maximum (i.e. $\lfloor (n\text{-}1)/2 \rfloor$).

In the tournament systems which are presented in Section 1.8.3, we observe that all of them respect the constraint:

$$\forall\, i \in \{1... q\} \wedge \forall\, j \in \{2q... 2q+r\} \rightarrow P_i + P_{i+1} \geq 2P_j \quad (2.35)$$

With respect to the constraint (2.35), we can say that a uniform final state reaches its minimum number of points when $x = 0$. By applying this principle on the tournament T_1, we find that the rows of the uniform state with the minimum number of points would have the form (0, 0, 0, n-1, 0, 0). Table 2.26 shows the uniform states with the minimum number of points, in the case of the tournament tables addressed in Section 1.8.3, where C_2 is not empty.

Table 2.26. The uniform final states with the minimum number of points in the case of the sports presented in Section 1.8.3, where C_2 is not empty

Sports	**Game Results**
Football, rugby, handball, chess	(0, n-1, 0)
Cricket	(0, 0, n-1, 0), (0, 0, 0, n-1)

2.7.1.2. C_2 Is Empty

As discussed in Section 2.3.6, when C_2 is empty, a uniform state of a tournament final table can be formed only when the number of participants, n, is odd. The uniform states in this instance are formed based on the game results R_i, R_{i+1} where {Ri, R_{i+1}} is a pair from C_1, and $i \in \{1, 3, ..., 2q\text{-}1\}$. Supposing that P_i and P_{i+1} are the numbers of points of each of R_i and R_{i+1}, respectively, the number of the total points which is gained by every participant can be calculated as follows:

$$Pts(k) = P_i(n\text{-}1)/2 + P_{i+1}(n\text{-}1)/2 = (P_i+P_{i+1})(n\text{-}1)/2 \quad (2.36)$$

Since the value of $(n\text{-}1)/2$ is positive, $Pts(k)$ reaches its minimum value when P_i+P_{i+1} become the minimum point of the pairs in C_1.

For the tournament T_2 which is previously presented in Section 2.3.6, let us suppose that P_1=7, P_2=0, P_3=5, P_4=1, P_5=3, and P_6=2 are the points which a participant gains from the games ending in R_1, R_2, R_3, R_4, R_5, or R_6 respectively. Given that the order of the possible results of the games in the final table of T_2 is R_1, R_3, R_5, R_6, R_4 and R_2, and n is odd, if the final table of T_2 has a uniform state, then the final data of all participants would have one of the following forms: $((n\text{-}1)/2, 0, 0, 0, 0, (n\text{-}1)/2)$, $(0, (n\text{-}1)/2, 0, 0, (n\text{-}1)/2, 0)$, or $(0, 0, (n\text{-}1)/2, (n\text{-}1)/2, 0, 0)$. Since $\min(P_1+P_2, P_3+P_4, P_5+P_6) = P_5+P_6$, the rows of the uniform state with the minimum number of points would have the form $(0, 0, (n\text{-}1)/2, (n\text{-}1)/2, 0, 0)$. Table 2.27 shows the uniform final states with the minimum number of points in the case of the sports tournaments given in Section 1.8.3, where C_2 is empty and n is odd.

Table 2.27. The uniform final states with the minimum number of points in the case of the sports tournaments presented in Section 1.8.3, where C_2 is empty and n is odd

Sports	**Game Results**
Basketball, volleyball, tennis, lacrosse	$((n\text{-}1)/2, (n\text{-}1)/2)$
Ice hockey, Curling	$((n\text{-}1)/2, 0, 0, (n\text{-}1)/2), (0, (n\text{-}1)/2, (n\text{-}1)/2, 0)$

2.7.2. Uniform Final States With Maximum Number of Points

2.7.2.1. C_2 Is Not Empty

Since each of x and n-2x-1 in Eq. (2.33) are positive, $Pts(k)$ reaches its maximum value when P_j becomes the maximum point of the elements in C_2, and P_i+P_{i+1} becomes the maximum point of the pairs in C_1. In this case, for the tournament T_1, since $\max(P_1+P_2, P_3+P_4) = P_1+P_2$ and $\max(P_5, P_6) = P_5$, to obtain a final state with the maximum number of points, the final data of all participants must have the form $(x, 0, n\text{-}2x\text{-}1, 0, 0, x)$.

By respecting the Eq. (2.35), a uniform final state reaches its maximum number of points when $x = \lfloor (n\text{-}1)/2 \rfloor$. By applying this principle on the tournament T_1, the rows of the uniform final state with the maximum number of points would have the form

$(\lfloor (n\text{-}1)/2 \rfloor, 0, n\text{-}2\lfloor (n\text{-}1)/2 \rfloor\text{-}1, 0, 0, \lfloor (n\text{-}1)/2 \rfloor)$. Table 2.28 shows the uniform final states with the maximum number of points in the case of the sports tournaments given in Section 1.8.3, where C_2 is not empty.

Table 2.28. The uniform final states with the maximum number of points in the case of the sports tournaments given in Section 1.8.3, where C_2 is not empty

Sports	**Game Results**
Football, rugby, handball, chess	$(\lfloor (n\text{-}1)/2 \rfloor, n\text{-}2\lfloor (n\text{-}1)/2 \rfloor\text{-}1, \lfloor (n\text{-}1)/2 \rfloor)$
Cricket	$(\lfloor (n\text{-}1)/2 \rfloor, \lfloor (n\text{-}1)/2 \rfloor, n\text{-}2\lfloor (n\text{-}1)/2 \rfloor\text{-}1, 0)$, $(\lfloor (n\text{-}1)/2 \rfloor, \lfloor (n\text{-}1)/2 \rfloor, 0, n\text{-}2\lfloor (n\text{-}1)/2 \rfloor\text{-}1)$

2.7.2.2. C_2 Is Empty

Since the value of $(n\text{-}1)/2$ in Eq. (2.36) is positive, $Pts(k)$ reaches its maximum value when P_i+P_{i+1} becomes the maximum point of the pairs in C_1. In this case, for the tournament T_2 between an odd number of participants, n, $\max(P_1+P_2, P_3+P_4, P_5+P_6) = P_1+P_2$, which means that the rows of the uniform state in the final table of T_2 with the maximum number of points would have the form $((n\text{-}1)/2, 0, 0, 0, 0, (n\text{-}1)/2)$. Table 2.29 shows the uniform final states with the maximum number of points in the case of the sports tournaments addressed in Section 1.8.3, where the set C_2 is empty and the value of n is odd.

Table 2.29. The uniform final states with the maximum number of points in the case of the sports tournaments given in Section 1.8.3, where C_2 is empty and n is odd

Sports	**Game Results**
Basketball, volleyball, tennis, lacrosse	$((n\text{-}1)/2, (n\text{-}1)/2)$
Ice hockey, Curling	$((n\text{-}1)/2, 0, 0, (n\text{-}1)/2), (0, (n\text{-}1)/2, (n\text{-}1)/2, 0)$

2.7.3. Minimum Final Points of a *k*th Participant

To determine the minimum number of points that can be earned by a participant at a *k*th position, the final points of the participant must be considered separately depending on the results of its games against the other participants at the higher and lower positions in the final table. We split the tournament into two sub-tournaments, where the graph of each sub-tournament is considered as a sub-graph of the overall tournament. By combining these sub-graphs we get the graph of the overall tournament. In a tournament between n participants, there are k-1 participants at higher positions than k, and n-k participants at lower positions. Thus, the resulting two sub-tournaments would be a sub-tournament between the first k participants, and another one between the last n-k+1 participants.

For a *k*th participant to get a minimum number of points, all its games against the participants at the higher positions must end in R_{i+1}, where $i \in \{1, 3, ..., 2q\text{-}1\}$. The number of points which can be gained from a game that ends in R_{i+1} is $P_{i+1} = \min(P_2, P_4, ..., P_{2q})$. If P_{high} denotes the number of points which a participant can gain from the higher sub-tournament, the minimum number of points which can be gained by a *k*th participant from this sub-tournament would be:

$$\min(P_{\text{high}}(k)) = (k\text{-}1)\min(P_2, P_4, ..., P_{2q}) \tag{2.37}$$

Moreover, since the *k*th participant appears the first in the sub-tournament with the participants at the lower positions than k, this sub-tournament must have a uniform state of the final table with a minimum number of points, which gets involved in discussing three particular scenarios. The first scenario takes place when C_2 is not empty. In this case, as in previously discussed in Section 2.7.1.1, the state of the final table is uniform with a minimum number of points when the games of all participants end in R_j, where $R_j \in C_2$ and the number of the points which is gained from a game that ends in R_j is $P_j = \min(P_{2q+1}, P_{2q+2}, ..., P_{2q+r})$. If P_{low} denotes the number of points obtained from the lower sub-tournament, the minimum number of points which can be gained by a *k*th participant from this sub-tournament would be:

$$\min(P_{\text{low}}(k)) = (n\text{-}k)\min(P_{2q+1}, P_{2q+2}, ..., P_{2q+r}), \text{ where } C_2 \neq \{\} \tag{2.38}$$

The second scenario occurs when C_2 is empty and n-k is even. In this scenario based on the issue discussed in Section 2.7.1.2, each participant earns a number of points obtained by $(P_i+P_{i+1})*(n-k)/2$, where (R_i, R_{i+1}) is a pair from C_1, $i \in \{1,3,..., 2q-1\}$, P_i and P_{i+1} are the points gained from a game ending in R_i and R_{i+1} respectively, and $P_i+P_{i+1} = \min(P_1+P_2, P_3+P_4, ..., P_{2q-1}+P_{2q})$. The minimum number of points that a kth participant can earn from this sub-tournament would be:

$$\min(P_{low}(k)) = ((n-k)/2)\ \min(P_1+P_2, P_3+P_4, ..., P_{2q-1}+P_{2q}),$$
$$\text{where } C_2 = \{\} \text{ and } n\text{-}k \text{ is even} \qquad (2.39)$$

The last scenario occurs when C_2 is empty and n-k is odd. In this scenario, it will not be possible to form a uniform state of the final table (see Section 2.3.6). The closest table state to a uniform one can be determined by dividing the participants into two halves. For the first half of the n-k+1 participants in the lower sub-tournament, the $\lfloor (n-k)/2 \rfloor$ games end in R_i, the $\lfloor (n-k)/2 \rfloor$ games end in R_{i+1} and a single game ends in R_l, where the participant playing the single game gains $P_l = \min(P_1, P_3, ..., P_{2q-1})$. For the second half of the participants, the $\lfloor (n-k)/2 \rfloor$ games end in R_i, the $\lfloor (n-k)/2 \rfloor$ games end in R_{i+1} and a single game ends in R_{l+1}, where $\{R_l, R_{l+1}\} \in C_1$. Thus, the minimum number of points which can be gained by a kth participant from this sub-tournament can be calculated as follows:

$$\min(P_{low}(k)) = \lfloor (n-k)/2 \rfloor\ \min(P_1+P_2, P_3+P_4, ..., P_{2q-1}+P_{2q}) +$$
$$\min(P_1, P_3, ..., P_{2q-1}), \text{ where } C_2 = \{\} \text{ and } n\text{-}k \text{ is odd} \qquad (2.40)$$

Eq. (2.39) and Eq. (2.40) can be combined into a single one like:

$$\min(P_{low}(k)) = \lfloor (n-k)/2 \rfloor\ \min(P_1+P_2, P_3+P_4, ..., P_{2q-1}+P_{2q}) +$$
$$(n-k-2\lfloor (n-k)/2 \rfloor)\ \min(P_1, P_3, ..., P_{2q-1}), \text{ where } C_2 = \{\} \qquad (2.41)$$

After calculating each of $\min(P_{high}(k))$ and $\min(P_{low}(k))$, the minimum number of points which can be gained by a participant at the kth position can be given by:

$$\min(Pts(k)) = \min(P_{high}(k)) + \min(P_{low}(k)) \qquad (2.42)$$

Table 2.30 presents each of $\min(P_{high}(k))$, $\min(P_{low}(k))$ and $\min(Pts(k))$ in the case of the sports tournaments presented in Section 1.8.3.

Table 2.30. $\min(P_{high}(k))$, $\min(P_{low}(k))$ and $\min(Pts(k))$ in the case of sports tournaments given in Section 1.8.3

Sports	$\min(P_{high}(k))$	$\min(P_{low}(k))$	$\min(Pts(k))$
Football, rugby, handball	0	$n-k$	$n-k$
Chess	0	$(n-k)/2$	$(n-k)/2$
Basketball, volleyball, tennis	$k-1$	$2(n-k)-\lfloor(n-k)/2\rfloor$	$2n-k-\lfloor(n-k)/2\rfloor-1$
lacrosse	0	$n-k-\lfloor(n-k)/2\rfloor$	$n-k-\lfloor(n-k)/2\rfloor$
Ice hockey, Curling	0	$2(n-k)-\lfloor(n-k)/2\rfloor$	$2(n-k)-\lfloor(n-k)/2\rfloor$
Cricket	0	$n-k$	$n-k$

2.7.4. Maximum Final Points of a *k*th Participant

Similar to the principle of determining the minimum number of points for a *k*th tournament participant, the maximum number of points must be considered separately depending on the results of its games against the other participants at the higher and lower positions in the tournament table.

For a *k*th participant to get a maximum number of points, all its games against the participants at the lower positions must end in R_i, where $i \in \{1, 3, ..., 2q-1\}$ and the number of points gained from a game that ends in R_i is $P_i=\max(P_1, P_3, ..., P_{2q-1})$. We can represent the maximum number of points which can be gained by a *k*th participant from this sub-tournament as:

$$\max(P_{low}(k)) = (n-k)\max(P_1, P_3, ..., P_{2q-1}) \tag{2.43}$$

Moreover, since the *k*th participant appears the last in the sub-tournament with the participants at the higher positions than *k*, this sub-tournament must have a uniform state of the final table with a maximum number of points, which gets involved in discussing three particular scenarios.

The first scenario takes place when C_2 is not empty. Let us suppose that P_i, P_{i+1} and P_j are the number of points which a participant can gain from a game that ends in R_i, R_{i+1} or R_j respectively, where $\{R_i, R_{i+1}\} \in C_1$ and $R_j \in C_2$. In this case, as in previously discussed in Section 2.7.2.1, the state of the final table is uniform with a maximum number of points when the $\lfloor(k\text{-}1)/2\rfloor$ games end in R_i, the $\lfloor(k\text{-}1)/2\rfloor$ games end in R_{i+1} and $k\text{-}2\lfloor(k\text{-}1)/2\rfloor\text{-}1$ games end in R_j, where $P_j = \max(P_{2q+1}, P_{2q+2}, ..., P_{2q+r})$, and $P_i+P_{i+1}= \max(P_1+P_2, P_3+P_4, ..., P_{2q-1}+P_{2q})$. The maximum number of points which can be gained by a kth participant from this sub-tournament in this instance would be:

$$\max(P_{high}(k)) = \lfloor(k\text{-}1)/2\rfloor \max(P_1+P_2, P_3+P_4, ..., P_{2q-1}+P_{2q}) + (k\text{-}2\lfloor(k\text{-}1)/2\rfloor\text{-}1) \max(P_{2q+1}, P_{2q+2}, ..., P_{2q+r}), \text{ where } C_2 \neq \{\} \quad (2.44)$$

The second scenario occurs when C_2 is empty and k-1 is even. In this scenario based on the issue discussed in Section 2.7.2.2, each participant earns a number of points obtained by $(P_i+P_{i+1})*(n\text{-}k)/2$, where $\{R_i, R_{i+1}\} \in C_1$, $i \in \{1, 3,..., 2q\text{-}1\}$, and $P_i+P_{i+1}=\max(P_1+P_2, P_3+P_4, ..., P_{2q-1}+P_{2q})$. The maximum number of points that a kth participant can earn from this sub-tournament would be:

$$\max(P_{high}(k)) = ((k\text{-}1)/2) \max(P_1+P_2, P_3+P_4, ..., P_{2q-1}+P_{2q}), \text{ where } C_2 = \{\} \text{ and } k\text{-}1 \text{ is even} \quad (2.45)$$

The third scenario occurs when C_2 is empty and k-1 is odd. In this scenario, as explained in Section 2.3.6, it will not be possible to form a uniform state of the final table. The closest table state to a uniform one can be determined by dividing the participants into two halves. For the second half of the participants, the $\lfloor(k\text{-}1)/2\rfloor$ games end in R_i, the $\lfloor(k\text{-}1)/2\rfloor$ games end in R_{i+1} and the remaining game ends in R_{l+1}, where the participant playing the latter game gains $P_{l+1} = \max(P_2, P_4, ..., P_{2q})$. For the first half of the kth participants in the upper sub-tournament, the $\lfloor(k\text{-}1)/2\rfloor$ games end in R_i, the $\lfloor(k\text{-}1)/2\rfloor$ games end in R_{i+1}, and one game ends in R_l, where $\{R_l, R_{l+1}\} \in C_1$. Thus, the maximum number of points which can be gained by a kth participant from this sub-tournament can be calculated as follows:

$$\max(P_{high}(k)) = \lfloor (k\text{-}1)/2 \rfloor \max(P_1+P_2, P_3+P_4, ..., P_{2q-1}+P_{2q}) + \max(P_2, P_4, ..., P_{2q}), \text{ where } C_2 = \{\} \text{ and } k\text{-}1 \text{ is odd} \quad (2.46)$$

Eq. (2.45) and Eq. (2.46) can be combined into a single one like

$$\max(P_{high}(k)) = \lfloor (k\text{-}1)/2 \rfloor \max(P_1+P_2, P_3+P_4, ..., P_{2q-1}+P_{2q}) + (k\text{-}2\lfloor (k\text{-}1)/2 \rfloor\text{-}1) \max(P_2, P_4, ..., P_{2q}), \text{ where } C_2 = \{\} \quad (2.47)$$

After calculating each of $\max(P_{low}(k))$ and $\max(P_{high}(k))$, the maximum number of points which can be gained by a participant at the kth position can be given by:

$$\max(Pts(k)) = \max(P_{high}(k)) + \max(P_{low}(k)) \quad (2.48)$$

Table 2.31 presents each of $\max(P_{high}(k))$, $\max(P_{low}(k))$ and $\max(Pts(k))$ in the case of sports tournaments presented in Section 1.8.3.

Table 2.31. $\max(P_{high}(k))$, $\max(P_{low}(k))$ and $\max(Pts(k))$ in the case of the sports tournaments given in Section 1.8.3

Sports	$\max(P_{high}(k))$	$\max(P_{low}(k))$	$\max(Pts(k))$
Football	$\lfloor (k\text{-}1)/2 \rfloor + k\text{-}1$	$3(n\text{-}k)$	$3n\text{-}2k+\lfloor (k\text{-}1)/2 \rfloor\text{-}1$
Rugby, handball	$k\text{-}1$	$2(n\text{-}k)$	$2n\text{-}k\text{-}1$
Chess	$(k\text{-}1)/2$	$n\text{-}k$	$n\text{-}(k+1)/2$
Basketball, volleyball, tennis	$\lfloor (k\text{-}1)/2 \rfloor + k\text{-}1$	$2(n\text{-}k)$	$2n\text{-}k+\lfloor (k\text{-}1)/2 \rfloor\text{-}1$
lacrosse	$\lfloor (k\text{-}1)/2 \rfloor$	$n\text{-}k$	$n\text{-}k+\lfloor (k\text{-}1)/2 \rfloor$
Ice hockey, Curling	$\lfloor (k\text{-}1)/2 \rfloor + k\text{-}1$	$3(n\text{-}k)$	$3n\text{-}2k+\lfloor (k\text{-}1)/2 \rfloor\text{-}1$
Cricket	$k\text{-}1$	$2(n\text{-}k)$	$2n\text{-}k\text{-}1$

2.7.5. Interval of Points of a *k*th Participant

The relations $\max(Pts(k))$ and $\min(Pts(k))$ help that the interval of $Pts(k)$ can be written as follows:

$$\min(Pts(k)) \leq Pts(k) \leq \max(Pts(k)), \text{ where } 1 < k \leq n \quad (2.49)$$

Based on Eq. (2.49), Table 2.32 presents the optimized search space of *WDL* values which can be obtained by each team in a 4-team football tournament, where the *WDL* values that a team at the *k*th position can have are marked by X. In the case of the backward approach, the number of states which must be evaluated based on this optimized search space is 6*7*7*7=1764, which is less than 1/5 of the states evaluated with the blind search algorithm (10^4 states).

Table 2.32. The optimized search space of *WDL* based on Eq. (2.49) of a 4-team football tournament

WDL	(3, 0, 0)	(2, 1, 0)	(2, 0, 1)	(1, 2, 0)	(1, 1, 1)	(0, 3, 0)	(1, 0, 2)	(0, 2, 1)	(0, 1, 2)	(0, 0, 3)
Pts / ***k***	9	7	6	5	4	3	3	2	1	0
1	X	X	X	X	X	X	-	-	-	-
2	-	X	X	X	X	X	X	X	-	-
3	-	-	X	X	X	X	X	X	X	-
4	-	-	-	-	X	X	X	X	X	X

Together with the relation $Pts(k) \leq \max(Pts(k))$, Eq. (2.17) must also be respected. Table 2.33 presents the final table of a football tournament of 4 teams, where the maximum number of points gained by the team at the second position (7 points) is bigger than the number of points gained by the first team (6 points). Table 2.34 presents the final table of an ice hockey tournament of 4-teams, where the maximum gained number of points for the team at the second position (7 points) is lower than the number of points for the first team (8 points). These two cases prove that $\max(Pts(k))$ might be bigger or smaller than $Pts(k\text{-}1)$, which means that $Pts(k)$ must satisfy the following relation:

$$Pts(k) \leq \min(\max(Pts(k)), Pts(k\text{-}1)) \quad (2.50)$$

By taking Eq. (2.50) into account, the interval of $Pts(k)$ can be represented as follows:

$$\min(Pts(k)) \leq Pts(k) \leq \min(\max(Pts(k)), Pts(k\text{-}1)), \text{ where } 1 < k \leq n \quad (2.51)$$

Table 2.33. The final table of a football tournament of 4 teams where $\max(Pts(2)) > Pts(1)$

Teams	***W***	***D***	***L***
A	2	0	1
B	1	2	0
C	1	1	1
D	0	1	2

Table 2.34. The final table of an ice hockey tournament of 4 teams where $Pts(1) > \max(Pts(2))$

Teams	***W***	***OTW***	***OTL***	***L***
A	2	1	0	0
B	2	0	1	0
C	0	1	0	2
D	0	0	1	2

2.8. Optimization of Final Table States

2.8.1. Optimized Backward Algorithm

In this section, the backward algorithm that is presented in Section 2.6.1 is optimized, where we respect Eq. (2.51). Instead of the selection of every value in the set *S* for every participant (see Section 2.3.4), the participants will be given only the possible numbers of points in accordance with their positions in the final table.

Before the optimized algorithm starts the generation of the states of a tournament final table, as well as the minimum and maximum number of points which can be

succeeded by each participant, the set S is constructed together with the total number of points of each element in that set, contained in the set PS. Afterwards, the sets S and PS are sorted in the descending order of the elements of PS. The optimized backward algorithm to enumerate the states of a final table can be organized as follows:

Optimized backward algorithm for enumerating the states of a tournament final table:

```
Inputs:
   n:  number of participants
   C1: game results which are related to each other
   q:  size of C1
   C2: game results which are not related to other game results
   r:  size of C2

Outputs:
   Ts[][n][2q+r]: states of final tournament table

Static variables and data structures:
   N:          number of elements of the set S(n, 2q+r)
   S[N][2q+r]: the set S(n, 2q+r)
   T[n][2q+r]: tournament table
   PS[N]:      number of points of each element of S
   minP[n]:    possible minimum points of each participant
   maxP[n]:    possible maximum points of each participant

Main Procedure
1:  calculate N based on Eq. (2.23)
2:  calculate S based on Eq. (2.20)
3:  PS ← number of points of each element from S
4:  sort the elements of S and PS depending on the descending
    order of the elements of PS
5:  minP ← minimum number of points which can be gained by
    each participant // based on Eq. (2.42)
6:  maxP ← maximum number of points which can be gained by
    each participant // based on Eq. (2.48)
7:  stateGenerator(1, 1)

Procedure: stateGenerator(k, i_p)
1:  if k ≤ n then
2:    i ← i_p
3:    while PS[i] > maxP[k] do
4:      i++
5:    while i ≤ N and PS[i] ≥ minP[k] do
6:      for j ← 1 to 2q+r do
7:        T[k][j] ← S[i][j]
8:      i2 ← minimum index to PS where PS[i]=PS[i2]
9:      stateGenerator(k+1, i2)
10:     i++
11: else
12:   valid ← stateChecker(T)
13:   if valid = true then
14:     insert T into Ts
```

Compared to the previous version of the algorithm (see Section 2.6.1), the procedure stateGenerator takes two parameters rather than one; namely the position of a participant and the index of the element selected from *S* for that participant. Note that *i* with the initial value *i_p* is an index to the elements of both *S* and *PS* (see line 2 of stateGenerator). If *PS*[*i*] > *maxP*[*k*], *i* keeps being incremented by 1 until the condition is false (see lines 3 and 4 of stateGenerator) and then the certain elements of *S* are assigned to *T*. This ensures that the number of the corresponding points of the elements assigned from *S* to *T* is less or equal to both max(*Pts*(*k*)) and *Pts*(*k*-1). The condition at line 5 of the procedure stateGenerator ensures that the elements from *S* keep being assigned to *T*, where *i* keep being increased by 1 (line 10) until the algorithm reaches an element from *S* with the number of points less or equal to min(*Pts*(*k*)). Thus, Eq. (2.51) would be satisfied.

By passing *i*2 (which holds the minimum index to *PS* where *PS*[*i*]=*PS*[*i*2]) as a parameter during the recursive call to the procedure stateGenerator at line 9, we ensure selecting every possible element of *S* with the same number of points for the next participant at the position (*k*+1). By sorting the elements of both *S* and *PS* in the descending order of the elements of *PS*, we ensure that the states are generated in the descending order of their corresponding points. So, there is no need to check whether the points of the teams are in descending order or not. In this way, the task performed at line 2 of the previous version of the algorithm is eliminated (see Section 2.6.1).

2.8.2. Optimized Forward Algorithm

In this section, the forward algorithm that is presented in Section 2.6.2 is optimized, taking Eq. (2.51) into account. In this case, the algorithm generates the edges of each node representing a different participant in which the total number of their corresponding points is between min(*Pts*(*k*)) and min(max(*Pts*(*k*)), *Pts*(*k*-1)) rather than any value of *Pts*(*k*).

Before the optimized algorithm starts the generation of the states of a tournament final table, with the same principle as the optimized backward algorithm, the minimum and maximum number of points which can be gained by each participant must be calculated. The pseudo-code of the optimized forward algorithm to enumerate the states of a final table is given below.

Optimized forward algorithm for enumerating the states of a tournament final table:

```
Inputs:
   n:  number of participants
   C1: game results which are related to each other
   q:  size of C1
   C2: game results which are not related to other game results
   r:  size of C2

Output:
   Ts[][n][2q+r]: states of final tournament table

Static data structures:
   G[n][n]:    tournament graph
   T[n][2q+r]: tournament table
   minP[n]:  possible minimum points of each participant
   maxP[n]:  possible maximum points of each participant
   Pts[n]:     number of points of each participant

Main Procedure
1:  set all the elements of G to 0
2:  minP ← minimum number of points which can be gained by
    each participant // based on formula (2.42)
3:  maxP ← maximum number of points which can be gained by
    each participant // based on formula (2.48)
4:  graphGenerator(1, 2)

Procedure: graphGenerator(k, j)
1:  for i ← 1 to 2q+r do
2:    G[k][j] ← i
3:    if {Ri, Ri+1} ∈ C1 then
4:      G[j][k] ← i+1
5:    else if {Ri-1, Ri} ∈ C1 then
6:      G[j][k] ← i-1
7:    else
8:      G[j][k] ← i
9:    if j+1 ≤ n then
10:     graphGenerator(k, j+1)
11:   else
12:     Pts[k] ← number of points of a kth participant
13:     if Pts[k] ≥ minP[k] then
14:       if (k > 1 and Pts[k] ≤ min(maxP[k], Pts[k-1])) or k = 1
          then
15:         if k+1 < n then
16:           graphGenerator(k+1, k+2)
17:         else
18:           Pts[n] ← total number of points of the n-th
              participant
19:           if Pts[n] ≤ min(maxP[n], Pts[n-1]) then
20:             T ← the equivalent state of G
21:             if T ∉ Ts then
22:               insert T into Ts
```

In the algorithm, after generating the edges of each node k and before performing the same task for the next node k+1, the total points must be checked whether it respects

Eq. (2.51) or not. As it is illustrated in lines 13 and 14 of the procedure graphGenerator, since the first node is not connected to a preceding one, the total number of its points is compared with min(*Pts*(1)) only, while the points of the other nodes except the last one are checked whether they are between min(max(*Pts*(*k*)), *Pts*(*k*-1)) and min(*Pts*(k)) or not.

Since all the edges of the last node are generated during the manipulation of the previous nodes, the procedure graphGenerator is not called to process that node. After completing the generation of the graph, a special test on the number of the corresponding points of the last node will be done. This node (the *n*th node) will never have the number of points less than 0 (i.e., held by min(*Pts*(*n*)) and so the number of its points will be compared only with min(max(*Pts*(*n*)), *Pts*(*n*-1)). This is illustrated in line 19 of the procedure graphGenerator.

After generating the edges of a *k*th node, if the number of its points does not respect Eq. (2.51), the following nodes will not be processed, and the graphs produced from the current sub-graph would be ignored. This is clearly shown in the case of each of Figures 2.22, 2.24 and 2.25. That is, after constructing the partial graph based on the generation of the edges of the first node, the number of its points would be less than the minimum points that it can gain, and therefore neither the graphs (a), (b) nor (c) in each figure would be generated.

A node will never have a number of points higher than the one of its previous nodes, because the number of the corresponding points of any node *k* is compared with min(max(*Pts*(*k*)), *Pts*(*k*-1)). So, there is no need to check the order of the points in the nodes after completing the generation of each graph.

Table 2.35 shows the steps of generating the final states of a football tournament using the optimized forward algorithm, where the tournament is involved by 3 teams.

Table 2.35. The steps of generating the final table states of a 3-team football tournament, using the optimized forward algorithm

Figure-Graph	**State**	**Status**	**Reason**
2.17-(a)	((2, 0, 0), (1, 0, 1), (0, 0, 2))	Accepted	-
2.17- (b)	((2, 0, 0), (0, 1, 1), (0, 1, 1))	Accepted	-
2.17-(c)	((2, 0, 0), (0, 0, 2), (1, 0, 1))	Not accepted	$Pts(2) < \min(Pts(2))$
2.18-(a)	((1, 1, 0), (1, 0, 1), (0, 1, 1))	Accepted	-
2.18-(b)	((1, 1, 0), (0, 1, 1), (0, 2, 0))	Not accepted	$Pts(2) < Pts(3)$
2.18-(c)	((1, 1, 0), (0, 0, 2), (1, 1, 0))	Not accepted	$Pts(2) < \min(Pts(2))$
2.19-(a)	((1, 0, 1), (1, 0, 1), (1, 0, 1))	Accepted	-
2.19-(b)	((1, 0, 1), (0, 1, 1), (1, 1, 0))	Not accepted	$Pts(2) < Pts(3)$
2.19-(c)	((1, 0, 1), (0, 0, 2), (2, 0, 0))	Not accepted	$Pts(2) < min(Pts(2))$
2.20-(a)	((1, 1, 0), (1, 1, 0), (0, 0, 2))	Accepted	-
2.20-(b)	((1, 1, 0), (0, 2, 0), (0, 1, 1))	Accepted	-
2.2-(c)	((1, 1, 0), (0, 1, 1), (1, 0, 1))	Not accepted	$Pts(2) < Pts(3)$
2.21-(a)	((0, 2, 0), (1, 1, 0), (0, 1, 1))	Not accepted	$Pts(1) < Pts(2)$
2.21-(b)	((0, 2, 0), (0, 2, 0), (0, 2, 0))	Accepted	-
2.21-(c)	((0, 2, 0), (0, 1, 1), (1, 1, 0))	Not accepted	$Pts(2) < Pts(3)$
2.22	-	Not accepted	$Pts(1) < \min(Pts(1))$
2.23-(a)	((1, 0, 1), (2, 0, 0), (0, 0, 2))	Not accepted	$Pts(1) < Pts(2)$
2.23-(b)	((1, 0, 1), (1, 1, 0), (0, 1, 1))	Not accepted	$Pts(1) < P(2)$
2.23-(c)	((1, 0, 1), (1, 0, 1), (1, 0, 1))	Not accepted	Already exists
2.24	-	Not accepted	$Pts(1) < \min(Pts(1))$
2.25	-	Not accepted	$Pts(1) < \min(Pts(1))$

2.9. Multi-threaded Optimization of Final Table States

2.9.1. Multi-threaded Optimized Backward Algorithm

In this section, the optimized backward algorithm that is presented in Section 2.8.1 is internally accelerated using multi-threading, instead of a single thread of execution. If a machine contains multiple processors, multi-threaded programs usually run on that machine faster than the classically structured programs. However, the threading

mechanism also has some certain disadvantages; for example, changing the context of threads and sharing critical resources cost some time, which has a negative effect on the execution time of the program. Therefore, not to degrade the algorithm performance, the optimum number of threads that allows the algorithm to exploit the maximum performance of the relevant machine must be determined.

The number of cores in the used machine is usually taken into account as the optimum number of threads, but this may not be always the right choice [171]. The relationship between the number of threads, the number of available cores, and the resulting speedups for multi-threaded programs is a complicated relationship [172]. This is due to the fact that other factors influence the performance other than the number of the threads [173], such as the thread context switch rate, the bandwidth utility, the thread migration rate, and the characteristics of the program itself. The overlapping of these factors makes estimating the optimum number of threads theoretically a challenging issue [174], where a change in any factor may affect the maximum performance, which requires adjusting the number of threads. It is possible to iteratively determine the optimum number of threads for a specific program [172, 175] through testing and comparing the performance of different numbers of threads. In this way, the right number of threads can be chosen, which gives the results in the shortest amount of time.

The multi-threaded optimized backward algorithm to generate the states of a tournament final table is presented below.

Multi-threaded optimized backward algorithm for enumerating the final states:

```
Inputs:
   n:       number of participants
   C1:      game results which are related to each other
   q:       size of C1
   C2:      game results which are not related to other game
            results
   r:       size of C2
   ThrLim: limit of threads that can run simultaneously

Outputs:
   Ts[][n][2q+r]: states of final tournament table

Static variables and data structures:
   N:           number of elements of the set S(n, 2q+r)
   S[N][2q+r]: set S(n, 2q+r)
   PS[N]:       the number of points of each element of S
   minP[n]:     possible minimum points of each participant
   maxP[n]:     possible maximum points of each participant
   thrNbr:      number of the running threads
```

```
Main Procedure
1:  calculate N based on formula (2.23)
2:  calculate S based on formula (2.20)
3:  PS ← number of points of each element from S
4:  sort the elements of S and PS depending on the descending
    order of the elements of PS
5:  minP ← minimum number of points which can be gained by
    each participant // based on formula (2.42)
6:  maxP ← maximum number of points which can be gained by
    each participant // based on formula (2.48)
7:  T[n][2q+r] // tournament table
8:  thrNbr ← 1
9:  stateGenerator(1, 1, T)

Procedure: stateGenerator(k, i_p, T_p)
1:  T ← T_p
2:  if k ≤ n then
3:    i ← i_p
4:    while PS[i] > maxP[k] do
5:      i++
6:    while i ≤ N and PS[i] ≥ minP[k] do
7:      T[k] ← S[i]
8:      i2 ← minimum index to PS where PS[i]=PS[i2]
9:      if thrNbr < thrLim then
10:       thrNbr++
11:       new thread:
12:         stateGenerator(k+1, i2, T)
13:         thrNbr--
14:     else
15:       stateGenerator(k+1, i2, T)
16:     i++
17: else
18:   valid ← stateChecker(T)
19:   if valid = true then
20:     insert T into Ts
```

In the algorithm, in addition to the position of a participant and the index of the element selected from *S* for that participant, the current state of the tournament table (*T*) is passed as a parameter. Since *T* is passed as a parameter during each call to the procedure stateGenerator, there is no need to declare it as a global data structure, which is locally defined in the procedure Main (line 7).

The variable *thrNbr* initially holds a value of 1, which represents the main thread. Before the procedure stateGenerator is called recursively, a check is performed on the number of the running threads (see line 9). If it is less than the optimum number (denoted by *thrLim*), *thrNbr* will be increased by 1 and the procedure will be called in a separate thread (see line 11). After creating a new thread for the call to stateGenerator, the current thread continues its execution in parallel and deals with the next iteration of the loop at line 6. The value of *thrNbr* will be decreased after the end of each thread. In the case

when the number of the running threads is equal to the optimum number of threads (i.e., *thrLim=thrNbr*), the recursive call to stateGenerator will be invoked sequentially by the same thread (i.e., the current thread).

Compared with the single-threaded version of the algorithm, the table *T* is passed as a parameter at each call to stateGenerator, instead of declaring it as a global variable. This is because the consideration of table *T* as a global variable allows it to be updated by many threads at the same time, which may affect the obtained results. Since each element of *Ts* (the possible states of the tournament final table) and *thrNbr* are global variables, they can be updated by each thread. Thus, they are treated as critical resources, and a synchronization mechanism is implemented to control access to them.

2.9.2. Multi-threaded Optimized Forward Algorithm

In this section, the optimized forward algorithm that is presented in Section 2.8.2 is internally accelerated using multi-threading, instead of a single thread of execution. In order to exploit the maximum performance of the relevant machine, the optimum number of threads must be determined. As previously explained in Section 2.9.1, the optimum number of threads can be determined iteratively by testing and comparing the performance of different numbers of threads. The multi-threaded optimized forward algorithm to generate the states of a tournament final table is described below.

Multi-threaded optimized forward algorithm for enumerating the final states:

```
Inputs:
   n:       number of participants
   C1:      game results which are related to each other
   q:       size of C1
   C2:      game results which are not related to other game
            results
   r:       size of C2
   thrLim: limit of threads that can run simultaneously
Output:
   Ts[][n][2q+r]: states of the final tournament table

Static data structures:
   minP[n]: possible minimum points of each participant
   maxP[n]: possible maximum points of each participant
   thrNbr:  number of the running threads

Main Procedure:
1:  minP ← minimum number of points which can be gained by
    each participant // based on formula (2.42)
```

```
2:  maxP ← maximum number of points which can be gained by
    each participant // based on formula (2.48)
3:  G[n][n] // tournament graph
4:  Pts[k] // number of points of each participant
5:  thrNbr ← 1
6:  graphGenerator(1, 2, G, Pts)

Procedure: graphGenerator(k, j, G_p, Pts_p)
1:  G ← G_p
2:  for i ← 1 to 2q+r do
3:    G[k][j] ← i
4:    if {Ri, Ri+1} ∈ C1 then
5:      G[j][k] ← i+1
6:    else if {Ri-1, Ri} ∈ C1 then
7:      G[j][k] ← i-1
8:    else
9:      G[j][k] ← i
10:   if j+1 ≤ n then
11:     if thrNbr < thrLim then
12:       thrNbr++
13:       new thread:
14:         graphGenerator(k, j+1, G, Pts_p)
15:         thrNbr--
16:     else
17:       graphGenerator(k, j+1, G, Pts_p)
18:   else
19:     Pts ← Pts_p
20:     Pts[k] ← total number of points of a kth participant
21:     if Pts[k] ≥ minP[k] then
22:       if (k > 1 and Pts[k] ≤ min(maxP[k], Pts[k-1])) or k = 1
          then
23:         if k+1 < n then
24:           if thrNbr < thrLim then
25:             thrNbr++
26:             new thread:
27:               graphGenerator(k+1, k+2, G, Pts)
28:               thrNbr--
29:           else
30:             graphGenerator(k+1, k+2, G, Pts)
31:         else
32:           Pts[n] ← total number of points of the n-th
              participant
33:           if Pts[n] ≤ min(maxP[n], Pts[n-1]) then
34:             T[n][2q+r] // tournament table
35:             T ← the equivalent state of G
36:             if T ∉ Ts then
37:               insert T into Ts
```

In the algorithm, in addition to the indices of a participant (i.e., *k*) and its opponent (i.e., *j*) in the graph *G*, the procedure graphGenerator takes two more parameters, which are the tournament graph (i.e., *G*) and the number of points of each participant (i.e., *Pts*).

Since *G* and *P* are passed as parameters, there is no need to declare them as global data structures, which are locally defined in the procedure Main (lines 3 and 4).

The variable *thrNbr* takes 1 as an initial value, which represents the main thread. Whenever the procedure graphGenerator is called recursively, the number of the running threads is checked (lines 11 and 24). If it is less than the optimum number of threads (denoted by *thrLim*), *thrNbr* will be increased by 1 and the procedure will be called in a separate thread (lines 14 and 27). After creating a new thread for the call to the procedure graphGenerator, the current thread deals with the execution of the next iteration of the loop at line 2 in parallel. The value of *thrNbr* will be decreased after the end of each thread. In the case when the number of the running threads is equal to the optimum number of threads (i.e., *thrLim=thrNbr*), the recursive call to the procedure stateGenerator will be invoked sequentially within the current thread.

Compared with the single-threaded version of the algorithm given in Section 2.8.2, instead of declaring *G* and *Pts* as global variables, they are passed as parameters at each call to the procedure stateGenerator. This is because the consideration of the tables *G* and *Pts* as global variables allows all the threads to update them at the same time, which may affect the obtained results. Since each element of *Ts* (the total states of the tournament final table) and *thrNbr* are global variables, they can be updated by many threads simultaneously. Thus, they are treated as critical resources, and a synchronization mechanism is implemented to control access to them.

3. COMPLEXITY ANALYSIS

3.1. Introduction

In this chapter, we analyze and discuss the complexities of the proposed approaches to provide a theoretical estimation for their required resources. During the analysis, we consider as resources each of the execution time and the memory space. The complexities of the proposed approaches mainly depend on the number of participants and the number of possible game results.

Both of the backward and forward approaches are based on blind search algorithms. To simplify the time/space complexity calculation of each approach, we first calculate the complexities of the blind search algorithms. After analyzing and discussing the complexities of the backward and forward algorithms, their optimized versions are analyzed. Next, we analyze the parallelized versions of the approaches. In this stage, in addition to the numbers of participants and possible game results, the optimum number of threads that can be executed in parallel in the used machine is taken into account as an influencing factor in the complexity calculation. Last and not least, based on the results of this analysis, the complexity class of the addressed problem (i.e., determining all the states of a tournament table) is defined.

3.2. Blind Search Algorithm for Enumerating The Final Table States

In Section 2.6.1, we have showed that the principle of the backward algorithm is directly related to the one of the blind search algorithm for enumerating the final table states. Thus, for simplicity, we calculate the complexity of the blind search algorithm before the complexity analysis of the backward approach.

3.2.1. Time Complexity

Let $T(n, m)$ be the time complexity of the blind search algorithm for n participants and m possible game results ($m = 2q+r$). As shown in Section 2.6.1, the main procedure

of the blind search algorithm contains 3 lines. The total time complexity of the algorithm is the sum of the complexities of these lines. So, we can write: $T(n, m) = T_1(n, m) + T_2(n, m) + T_3(n, m)$, where T_1, T_2 and T_3 are respectively the time complexities of lines 1, 2 and 3.

3.2.1.1. $T_1(n, m)$

Line 1 calculates the number, N, of possible game results (see Section 2.3.2) based on Eq. (2.23). This line can be represented by the following sub-algorithm:

```
Inputs: n, m
Output: N
1:  N ← 0
2:  calculateN(n-1, m)
Procedure: calculateN(r, d)
1:  N++
2:  if d > 1 then
3:    for i ← 1 to r do
4:      calculateN(i, d-1)
```

$T_1(n, m)$ is equivalent to the complexity $F_1(n\text{-}1, m)$ of the procedure calculateN. Supposing that $a = n\text{-}1$. $F_1(a, m)$ can be represented by the following recursive relation:

$F_1(x, 1) = O(1)$, where $1 \leq x \leq a$

$F_1(a, m) = \sum_{i=1}^{a} F_1(i, m\text{-}1)$

To simplify the calculations, we consider the following function called G_1 instead of F_1, where $G_1(a, m) \geq F_1(a, m)$. Thus, if a closed formal solution exists for G_1, we will have an upper bound on F_1. $G_1(a, m)$ is defined as follows:

$G_1(a, 1) = O(1)$

$G_1(a, m) = a*G_1(a, m\text{-}1) = a^2*G_1(a, m\text{-}2) = a^3*G_1(a, m\text{-}3) = a^i*G_1(a, m\text{-}i)$

For $i = m\text{-}1$:

$G_1(a, m) = a^{m-1}*G_1(a, 1) = O(a^{m-1})$

Since $G_1(a, m) \geq F_1(a, m)$ and $G_1(a, m) = O(a^{m-1})$, this means that $F_1(a, m) = O(a^{m-1})$. Since $T_1(n, m) = F_1(a, m)$, we can say that $T_1(n, m) = O(a^{m-1})$. When we substitute a for its value (i.e., n-1), $T_1(n, m) = O((n\text{-}1)^{m-1}) = O(n^{m-1})$.

3.2.1.2. $T_2(n, m)$

After calculating $T_1(n, m)$, we calculate the complexity of line 2 from the main procedure of the blind search algorithm (i.e., $T_2(n, m)$). Line 2 calculates the elements of the set S (set of the possible game results) based on Eq. (2.20). This line can be represented by the following sub-algorithm:

```
Inputs: n, m
Output: S[N][m]
Static data structure: s[m]
1:  calculateS(1)

Procedure: calculateS(j)
1:  for k ← 0 to n-1 do
2:    s[j] ← k
3:    if j+1 ≤ m then
4:      calculateS(j+1)
5:    else
6:      if ∑_{p=1}^{m} s[p] = m then
7:        add s into S
```

$T_2(n, m)$ is equivalent to the complexity the procedure calculateS. In line 6, the sub-algorithm is calculating the sum of the array s which has the size m. Line 7 adds the content of the array s into the last row of the two dimensional array S of the size N x m. It is clear that the complexity of each of lines 6 and 7 is $O(m)$. Thus, $T_2(n, m)$ can be represented by the following recurrence relation.

$T_2(n, 1) = n*O(m) = O(n*m)$

$T_2(n, m) = n*T_2(n, m\text{-}1) = n^2*T_2(n, m\text{-}2) = n^3*T_2(n, m\text{-}3) = n^i*T_2(n, m\text{-}i)$

For $i = m$-1:

$T_2(n, m) = n^{m-1}*T_2(n, 1) = O(m*n^m)$

3.2.1.3. $T_3(n, m)$

$T_3(n, m)$ is the complexity of the procedure stateGenerator. In lines 3 and 4 of stateGenerator we assign a row from the array S (of the size N x m) to the array T (of the size n x m), which causes this part of the algorithm to have a complexity of $O(m)$. Line 7 of the same procedure consists of assigning the values of the array T to the array Ts (which is an array of arrays of the size n x m), which causes line 7 to have a complexity of $O(n*m)$. $T_3(n, m)$ can be represented by the following recurrence relation.

$T_3(0, m) = O(n*m)$

$T_3(n, m) = N*[O(m) + T_3(n\text{-}1, m)]$

$= N*O(m) + N*T_3(n\text{-}1, m)$

$= N*O(m) + N^2*O(m) + N^2*T_3(n\text{-}2, m)$

$= N*O(m) + N^2*O(m) + N^3*O(m) + N^3*T_3(n\text{-}3, m)$

$= N*O(m) + N^2*O(m) + N^3*O(m) + \ldots + N^i*O(m) + N^i*T_3(n\text{-}i, m)$

For $i = n$:

$T_3(n, m) = N*O(m) + N^2*O(m) + N^3*O(m) + \ldots + N^n*O(m) + N^n*T_3(0, m)$

$= O(m)*\sum_{i=1}^{n} N^i + N^n*O(n*m)$

$= O(m)*[(N^{n+1} - 1)/(N - 1)] - O(m) + N^n*O(n*m)$

Based on the proposed sub-algorithm in Section 3.2.1.1 which calculates the value of N, we can say that n^{m-1} is an upper bound of the number N. By substituting N with n^{m-1} we get:

$T_3(n, m) = O(m)*[(n^{(m-1)*(n+1)} - 1)/(n^{m-1} - 1)]) + n^{(m-1)*n}*O(n*m)$

$= O(m)*O(n^{(m-1)*(n+1)} / n^{m-1}) + n^{(m-1)*n}*O(n*m)$

$= O(m)*O(n^{(m-1)*n}) + n^{(m-1)*n}*O(n*m)$

$= O(n^{(m-1)*n})*[O(m) + *O(n*m)]$

$= O(n^{(m-1)*n})*O(n*m)$

$= O(m*n^{n*m-n+1})$

3.2.1.4. *T*(*n*, *m*)

After calculating each of $T_1(n, m)$, $T_2(n, m)$ and $T_3(n, m)$, $T(n, m)$ would be the sum of these values. So, we can write:

$T(n, m) = T_1(n, m) + T_2(n, m) + T_3(n, m)$

$= O(n^{m-1}) + O(m*n^m) + O(m*n^{n*m-n+1})$

$= O(m*n^{n*m-n+1})$

This means that the complexity of the blind search algorithm mainly depends on the complexity of the procedure stateGenerator.

3.2.2. Space Complexity

To calculate the memory space complexity of the blind search algorithm, we distinguish the data structures whose sizes are related to *n* and/or *m*. We can note that the sizes of each of the arrays *S*, *T* and *Ts* are related to *n* and *m*. *S* and *T* are declared as abstract data structures, while *Ts* is the output of our algorithm. Unlike *S* and *T* whose sizes are predefined, the final size of *Ts* becomes known only at the end of the algorithm when all the generated states are saved into it. The memory space complexity $\$(n, m)$ in this situation equals the sum of the resultant complexities by each of *S*, *T* and *Ts*. The space complexities of *S* and *T* are equal respectively to $O(N*m)$ and $O(n*m)$. The complexity of *Ts* (denoted as $\$_1(n, m)$) can be defined as follows:

$\$_1(0, m) = O(n*m)$

$\$_1(n, m) = N*\$_1(n\text{-}1, m) = N^2*\$_1(n\text{-}2, m) = N^3*\$_1(n\text{-}3, m) = N^i*\$_1(n\text{-}i, m)$

When $i = n$:

$\$_1(n, m) = N^n*\$_1(0, m) = N^n*O(n*m)$

By substituting *N* with an upper bound of n^{m-1}, $\$_1(n, m)$ would be equal to $O(m*n^{n*m-n+1})$. After calculating $\$_1(n, m)$, $\$(n, m)$ will be as follows:

$$\begin{aligned}\$(n, m) &= O(N*m) + O(n*m) + O(m*n^{n*m-n+1}) \\ &= O(m*n^{m-1}) + O(m*n^{n*m}) + O(m*n^{n*m-n+1}) \\ &= O(m*n^{n*m-n+1})\end{aligned}$$

As seen above, the space complexity mainly depends on the occupied space by *Ts* (i.e., the generated states), and neither of *S* or *T* have an effect on it.

3.3. Backward Algorithm

3.3.1. Time Complexity

The only difference between the blind search algorithm and the backward algorithm is that the generated states are not saved unless their validities are agreed by the procedure stateChecker. This difference does not affect the complexities $T_1(n, m)$ and $T_2(n, m)$ (see Sections 3.2.1.1 and 3.2.1.2), but it may affect $T_3(n, m)$ (see Section 3.2.1.3). So, to calculate $T(n, m)$ for this algorithm, we have to recalculate $T_3(n, m)$.

3.3.1.1. $T_3(n, m)$

Since the number of the valid states is unknown, we suppose that the worst case of the execution time occurs when every generated case is valid. This means that every generated state is checked by the procedure stateChecker and saved into *Ts*. It is obvious that the test at line 7 has the complexity of $O(n)$. To calculate $T_3(n, m)$ for this case, we suppose that $T_4(n, m)$ is the complexity of stateChecker. $T_3(n, m)$ in this case is represented as:

$$T_3(0, m) = O(n) + T_4(n, m) + O(n*m) = T_4(n, m) + O(n*m)$$

$$T_3(n, m) = N*[O(m) + T_3(n\text{-}1, m)]$$

$$= N*O(m) + N^2*O(m) + \ldots + N^n*O(m) + N^n*T_3(0, m)$$

$$= O(m)*\sum_{i=1}^{n} N^i + N^n*[T_4(n, m) + O(n*m)]$$

$$= O(m)*[(N^{n+1} - 1)/(N - 1)] - O(m) + N^n*T_4(n, m) + N^n*O(n*m)$$

By substituting N with an upper bound of n^{m-1}, we get:

$$T_3(n, m) = O(m)*[(n^{(m-1)*(n+1)} - 1)/(n^{m-1} - 1)] + n^{(m-1)*n}*T_4(n, m) + n^{(m-1)*n}*O(n*m)$$

$$= O(m)*O(n^{(m-1)*(n+1)} / n^{m-1}) + n^{(m-1)*n}*T_4(n, m) + n^{(m-1)*n}*O(n*m)$$

$$= O(m)*O(n^{(m-1)*n}) + n^{(m-1)*n}*T_4(n, m) + n^{(m-1)*n}*O(n*m)$$

$$= O(n^{(m-1)*n})*[O(m) + *O(n*m)] + n^{(m-1)*n}*T_4(n, m)$$

$$= O(n^{(m-1)*n})*O(n*m) + n^{(m-1)*n}*T_4(n, m)$$

$$= O(m*n^{n*m-n+1}) + n^{n*m-n}*T_4(n, m)$$

Here, to be able to calculate $T_3(n, m)$, we have to calculate $T_4(n, m)$ (i.e., the complexity of the procedure stateChecker).

3.3.1.2. $T_4(n, m)$

It is clear that the complexity of line 1 in stateChecker is $O(n^2)$. Also, the resultant complexity from the lines 3 and 4 is $O(m)$. If we denote the complexity of the procedure isStateValid (called at line 5) by $T_5(n, m)$, $T_4(n, m)$ can be written as follows:

$$T_4(n, m) = O(n^2) + O(m) + T_5(n, m)$$

To be able to calculate $T_4(n, m)$, we have to calculate $T_5(n, m)$ (i.e., the complexity of the procedure isStateValid).

3.3.1.3. $T_5(n, m)$

Each of lines 4 and 10 of the procedure isStateValid has the complexity of $O(m)$, while line 7 has the complexity of $O(r)$. Since $m=2q+r$, we can say that O(m) is an upper bound of $O(r)$. The complexity of line 28 can be represented by $O(n*m)$, while the complexity of line 31 can be represented by $O(m)$. We suppose that $T_5(n, m) = F_5(n, n, m)$. The complexity of the procedure isStateValid is defined by the following recurrence relation:

$F_5(1, 1, m) = O(n*m)$

$F_5(1, x, m) = O(m) + F_5(x\text{-}1, x\text{-}1, m)$

$F_5(n, n, m) = O(m) + a*F_5(n\text{-}1, n, m)$, where $a = n\text{-}1$

$= O(m) + a*O(m) + a^2*F_5(n\text{-}2, n, m)$

$= O(m) + a*O(m) + a^2*O(m) + a^3*F_5(n\text{-}3, n, m)$

$= O(m) + a*O(m) + \ldots + a^{i-1}*O(m) + a^i*F_5(n\text{-}i, n, m)$

When $i = n\text{-}1$:

$F_5(n, n, m) = O(m) + a*O(m) + \ldots + a^{n-2}*O(m) + a^{n-1}*F_5(1, n, m)$

$= O(m)*\sum_{i=0}^{n-2} a^i + a^{n-1}*F_5(1, n, m)$

$= O(m)*[(a^{n-1} - 1)/(a - 1)] + a^{n-1}*[O(m) + F_5(n\text{-}1, n\text{-}1, m)]$

By substituting a with n-1, we get:

$F_5(n, n, m) = O[m*(n\text{-}1)^{n-2}] + (n\text{-}1)^{n-1}*[O(m) + F_5(n\text{-}1, n\text{-}1, m)]$

$= O[m*(n\text{-}1)^{n-2}] + O[m*(n\text{-}1)^{n-1}] + (n\text{-}1)^{n-1}*F_5(n\text{-}1, n\text{-}1, m)$

$= O[m*(n\text{-}1)^{n-1}] + (n\text{-}1)^{n-1}*F_5(n\text{-}1, n\text{-}1, m)$

$= O[m*(n\text{-}1)^{n-1}] + (n\text{-}1)^{n-1}*[O(m*(n\text{-}2)^{n-2}) + (n\text{-}2)^{n-2}*F_5(n\text{-}2, n\text{-}2, m)]$

$= O[m*(n\text{-}1)^{n-1}] + O[m*(n\text{-}1)^{n-1}*(n\text{-}2)^{n-2}] +$

$(n\text{-}1)^{n-1}*(n\text{-}2)^{n-2}*[O(m*(n\text{-}3)^{n-3}) + (n\text{-}3)^{n-3}*F_5(n\text{-}3, n\text{-}3, m)]$

$= O[m*(n\text{-}1)^{n-1}] + O[m*(n\text{-}1)^{n-1}*(n\text{-}2)^{n-2}] +$

$O[m*(n\text{-}1)^{n-1}*(n\text{-}2)^{n-2}*(n\text{-}3)^{n-3}] +$

$(n\text{-}1)^{n-1}*(n\text{-}2)^{n-2}*(n\text{-}3)^{n-3}*F_5(n\text{-}3, n\text{-}3, m)$

$= O[m*(n\text{-}1)^{n-1}] + O[m*(n\text{-}1)^{n-1}*(n\text{-}2)^{n-2}] + \ldots +$

$O[m*(n\text{-}1)^{n-1}*(n\text{-}2)^{n-2}*\ldots*(n\text{-}i)^{n-i}] +$

$(n\text{-}1)^{n-1}*(n\text{-}2)^{n-2}*\ldots*(n\text{-}i)^{n-i}*F_5(n\text{-}i, n\text{-}i, m)$

For $i=n\text{-}1$,

$F_5(n, n, m) = O[m*(n\text{-}1)^{n-1}] + O[m*(n\text{-}1)^{n-1}*(n\text{-}2)^{n-2}] + \ldots +$

$$O[m*(n\text{-}1)^{n-1}*(n\text{-}2)^{n-2}*\ldots*2^2*1^1] +$$
$$(n\text{-}1)^{n-1}*(n\text{-}2)^{n-2}*\ldots*2^2*1^1*F_5(1, 1, m)$$
$$= O(m) * \sum_{i=1}^{n-1} \prod_{j=1}^{i} (n\text{-}j)^{n-j} + F_5(1, 1, m) * \prod_{k=1}^{n-1}(n\text{-}k)^{n-k}$$
$$= O(m) * \sum_{i=1}^{n-1} \prod_{j=1}^{i} (n\text{-}j)^{n-j} + O(n*m) * \prod_{k=1}^{n-1}(n\text{-}k)^{n-k}$$

To simplify the calculations, we solve the following function called G_5 instead of F_5, where $G_5(n, n, m) \geq F_5(n, n, m)$. Thus, if a closed formal solution exists for G_5, we will have an upper bound on F_5. $G_5(n, n, m)$ is defined as follows:

$$G_5(n, n, m) = O(m) * \sum_{i=1}^{n-1} \prod_{j=1}^{n-1}(n\text{-}1)^{n-1} + O(n*m) * \prod_{k=1}^{n-1}(n\text{-}1)^{n-1}$$
$$= O(m)*(n\text{-}1)*(n\text{-}1)^{(n-1)*(n-1)} + O(n*m)*(n\text{-}1)^{(n-1)*(n-1)}$$
$$= O[(n\text{-}1)^{(n-1)*(n-1)}] * O[m*(n\text{-}1) + m*n]$$
$$= O(n^{(n-1)*(n-1)}) * O(m*n)$$
$$= O(m*n^{(n-1)*(n-1)+1})$$

Since $G_5(n, n, m) \geq F_5(n, n, m)$ and $F_5(n, n, m) = T_5(n, m)$, we can write

$T_5(n, m) = F_5(n, n, m) = O(m*n^{(n-1)*(n-1)+1})$

3.3.1.4. *T(n, m)*

After calculating $T_5(n, m)$, we have all the required values to determine $T(n, m)$, which can be calculated as follows:

$$T(n, m) = T_1(n, m) + T_2(n, m) + T_3(n, m)$$
$$= O(n^{m-1}) + O(n^m*m) + O(m*n^{n*m-n+1}) + n^{n*m-n}*T_4(n, m)$$
$$= O(n^{m-1}) + O(n^m*m) + O(m*n^{n*m-n+1}) + n^{n*m-n}*[O(n^2) + O(m) + T_5(n, m)]$$
$$= O(n^{m-1}) + O(n^m*m) + O(m*n^{n*m-n+1}) +$$
$$n^{n*m-n}*[O(n^2) + O(m) + O(m*n^{(n-1)*(n-1)+1})]$$
$$= O(m*n^{n*m-n+1}) + n^{n*m-n}*[O(m*n^{(n-1)*(n-1)+1})]$$
$$= n^{n*m-n}*[O(m*n) + O(m*n^{(n-1)*(n-1)+1})]$$
$$= n^{n*m-n}*O(m*n^{(n-1)*(n-1)+1})$$
$$= O(m*n^{n*(n+m-3)+2})$$

If we compare the complexity of the backward algorithm with the one of the blind search algorithm (i.e., $O(m*n^{n*m-n+1})$), we find that the complexity of this case is bigger. This leads to the conclusion that checking every generated state (by calling the procedure stateChecker) has a remarkable effect on the execution time.

3.3.2. Space Complexity

The difference between the space complexity calculations of the backward algorithm and the blind search algorithm is that the procedure stateChecker of the backward algorithm has some data structures related to *n* and/or *m*, which must be taken into account. These data structures are the static 2 dimensional arrays *T* and *G* which result in the complexities of $O(n*m)$ and $O(n^2)$ respectively, and the local array *trackingInds* which results in a complexity of $O(m)$. Supposing that $\$_2(n, m)$ is the complexity of stateChecker, the space complexity of the backward algorithm (i.e., $\$(n, m)$) would be like: $\$(n, m) = O(m*n^{n*m-n+1}) + \$_2(n, m)$.

The procedure stateChecker calls the recursive procedure isStateValid. Supposing that $\$_3(n, m)$ is the complexity of isStateValid, $\$_2(n, m)$ can be defined as:

$$\$_2(n, m) = O(n*m) + O(n^2) + O(m) + \$_3(n, m)$$
$$= O(n*m) + O(n^2) + \$_3(n, m)$$

The procedure isStateValid declares its own local array *trackingInds* every single recursive call. To calculate the resultant complexity of this procedure we have to define the maximum depth of the recursive calls. The maximum depth can be defined as follows:

$$d = \sum_{i=1}^{n-1} n-i = n*(n+1)/2$$

So, $\$_3(n, m)$ can be represented as:

$$\$_3(n, m) = [n*(n+1)/2] * O(m)$$
$$= O(n*m/2) * O(n+1)$$
$$= O(m*n^2).$$

After calculating $\$_3(n, m)$, $\$_2(n, m)$ would be like:

$$\$_2(n, m) = O(n*m) + O(n^2) + O(m*n^2)$$
$$= O(m*n^2).$$

After having the value of $\$_2(n, m)$, $\$(n, m)$ would be:

$$\$(n, m) = O(m*n^{n*m-n+1}) + O(m*n^2)$$
$$= O(m*n^{n*m-n+1}).$$

The space complexity remains the same in this case compared to the case of the blind search algorithm. This can be explained by the fact that the required memory space for the deepest recursive calls of the procedure isStateValid does not have a big effect on the memory space as the process of generating the final states does. The value of $\$(n, m)$ represents the complexity of this algorithm for the worst case, i.e. all the generated states

are saved, but this is not what really happens. We have just supposed that every generated state is saved because there is no practical way to determine the number of the generated states based on *n* and *m*. So the real complexity would be smaller than $O(m*n^{n*m-n+1})$, which is still an upper bound for it.

3.4. Optimized Backward Algorithm

3.4.1. Time Complexity

This algorithm differs from the previous one by taking into account some restrictions related to the maximum and minimum number of points that can be earned by a participant. These restrictions are applied during selecting the elements from the set *S*, which is then to be assigned to the tournament table. Applying the restrictions requires information related to the maximum and minimum number of points for each participant, which implies the need of preparing some new data structure before calling the procedure stateGenerator at line 7 of the main procedure.

The first required data structure is the array *PS* which contains the number of points for each element from *S* (see line 3). The process of calculating *PS* has a complexity of $O(N*m) = O(m*n^{m-1})$. In addition to calculating *PS* it needs to be sorted in a descending order. In the same time we sort the elements of *S* according to their related elements in *PS*. To achieve this, we propose the following sub-algorithm which is based on the selection sort algorithm:

```
Inputs: PS[N], S[N][m]
Output: PS[N], S[N][m]
1:  c, ind
2:  for i ← 1 to N-1 do
3:    ind ← 0
4:    for j ← i+1 to N do
5:      if PS[i] < PS[j] then
6:        ind ← j
7:    if ind ≠ 0 then
8:      c ← PS[i]
9:      PS[i] ← PS[ind]
10:     PS[ind] ← c
11:     for j ← 1 to m do
12:       c ← P[i][j]
13:       P[i][j] ← P[ind][j]
14:       P[ind][j] ← c
```

Supposing that $T_6(n, m)$ is the complexity of this sub-algorithm, $T_6(n, m)$ can be represented as:

$$T_6(n, m) = \sum_{i=1}^{N-1} i + N - 1$$
$$= (N^2-N)/2 + N - 1$$
$$= O(N^2-N) + O(N)$$
$$= O(N^2)$$

By substituting N with an upper bound of n^{m-1} we get $T_6(n, m) = O(n^{2*m-2})$.

Lines 5 and 6 of the optimized backward algorithm calculate the minimum and maximum number of points that can be gained by each participant. Each of these lines generates a complexity of $O(n)$. $T(n, m)$ in this case would be like:

$$T(n, m) = O(n^{m-1}) + O(m*n^m) + O(m*n^{m-1}) + O(n^{2*m-2}) + 2*O(n) + T_3(n, m)$$
$$= O(n^{2*m-2}) + T_3(n, m)$$

The backward algorithm uses a "for" loop to assign all the possible values of the set S to a specific participant in table T (see line 2 of the procedure stateGenerator). On the other hand, the optimized backward algorithm uses a "while" loop at line 4 to assign just the elements from S whose points respect the conditions. Since there is no useful way (i.e., based on n and m) to determine the number of elements from the set S which respect the points' range for a specific position of T, we are obligated to calculate the complexity for the worst case. The worst case would be considering all the elements of the set S, which means that we would be treating the backward algorithm again. This implies that the complexity $T_3(n, m)$ of the procedure stateGenerator for this case would also be equal to $O(m*n^{n*(n+m-3)+2})$. Although this value is greater than the real value, it is still an upper bound for $T_3(n, m)$. Thus, $T(n, m)$ would be:

$$T(n, m) = O(n^{2*m-2}) + O(m*n^{n*(n+m-3)+2}) = O(m*n^{n*(n+m-3)+2})$$

As seen from the final value of $T(n, m)$, the complexities of the achieved calculations by the main procedure (i.e., the lines from 1 to 6) do not affect the final complexity, where $T(n, m)$ is still directly related to $T_3(n, m)$ (the complexity of the procedure stateGenerator).

3.4.2. Space Complexity

The optimized backward algorithm keeps the same data structures appeared in the backward algorithm. In addition, there are some other data structures declared. These new

data structures are the static array *PS* which has a size of *N*, and the static arrays *minP* and *maxP* which have the size *n*. The calculated complexities of *PS*, *minP* and *maxP* are $O(n^{m-1})$, $O(n)$ and $O(n)$ respectively. The value of $\$(n, m)$ for this case would be the value of $\$(n, m)$ in the case of the backward algorithm plus the sum of the resultant complexities caused by the new declared data structures. So, we can write:

$$\$(n, m) = O(m*n^{n*m-n+1}) + O(n^{m-1}) + 2*O(n) = O(m*n^{n*m-n+1})$$

The space complexity remains the same in this case compared to the previous one. This can be explained by the fact that the new data structures have no effect on the complexity of the optimized backward algorithm, and $\$(n, m)$ still depends on the size of *Ts* (i.e., a 3-dimensional array which contains all the generated states).

3.5. Multi-threaded Optimized Backward Algorithm

3.5.1. Time Complexity

Compared to the sequential optimized backward algorithm, the multi-threaded algorithm has no difference in terms of fundamental processes. The only difference is that, in the multi-threaded version, a recursive call of the procedure stateGenerator is called in a new thread (parallel) whenever the number of the current threads (i.e., *thrNbr*) is less than the threads' limit (i.e., *thrLim*). To express the complexity of the multi-threaded algorithm, we have to take *thrLim* into account as a new parameter. The complexity of $T(n, m, thrLim)$ would be $T(n, m)/thrLim = O[(m*n^{n*(n+m-3)+2})/thrLim]$.

3.5.2. Space Complexity

The complexity $\$(n, m)$ is related to *Ts* in all the previous cases. This is due to the fact that *Ts* is the largest data structure in all the previous algorithms, which makes its resulting complexity dominant during the calculation of $\$(n, m)$. *Ts* also exists in this algorithm version, where it results in the same complexity (i.e., $O(m*n^{n*m-n+1})$). The difference in this algorithm compared to the sequential one is that the array *T* is not declared as static, but as a local array of each of the main procedure and the procedure stateGenerator. To calculate the resultant complexity for the procedure stateGenerator in this situation, we have to define the maximum depth of its recursive calls. The maximum

depth equals to $\sum_{i=1}^{n-1} n\text{-}i = n*(n+1)/2$. So, the resultant complexity would be equal to $[n*(n+1)/2]*O(n*m) = O(m*n^3)$.

Since the recursive calls of stateGenerator are directly related to the maximum number of threads (*thrLim*), there is a possibility that each thread reaches the maximum depth of the recursive calls of stateGenerator (i.e., a worst case), which means that a number of $[n*(n+1)/2]*thrLim$ of T arrays may be declared. In this situation, the complexity related to T would be equal to $O(thrLim*m*n^3)$. The complexity $\$(n, m, thrLim)$ of the multithreaded algorithm would be represented as follows:

$\$(n, m, thrLim) = O(m*n^{n*m-n+1}) + O(thrLim*m*n^3)$.

3.6. Blind Search Algorithm for generating a tournament graph

We have explained in Section 2.6.2 how the principle of the forward algorithm is directly related to the principle of the blind search algorithm for generating a tournament graph. Thus, for simplicity, we calculate the complexity of the blind search algorithm before the complexity analysis of the forward approach.

3.6.1. Time Complexity

We suppose that $T(n, m)$ is the time complexity of the blind search algorithm for n participants and m possible game results ($m = 2q+r$). In Section 2.6.2, the first line of the Main procedure of the blind search algorithm sets all the elements of a 2-dimensional array (i.e., G) to 0. Since G has a size of n x n, this makes the complexity of line 1 equal to $O(n^2)$. Line 2 of the same procedure is a call to the procedure graphGenerator. If we suppose that $T_1(n, m)$ represents the complexity of graphGenerator, $T(n, m)$ would be represented as: $T(n, m) = O(n^2) + T_1(n, m)$. To determine $T(n, m)$, $T_1(n, m)$ must be calculated.

Each of lines 3 and 5 of graphGenerator results in a complexity of $O(q)$. Since $m > q$, $O(m)$ can be considered as an upper bound of the complexities of lines 3 and 5. Besides, line 15 of the same procedure results in a complexity of $O(n^2)$. To calculate $T_1(n, m)$, we have to define the depth that the recursive calls of the procedure graphGenerator may reach. The depth d may be defined as:

$$d = \sum_{i=1}^{n-1} n\text{-}i = n*(n\text{-}1)/2$$

We suppose that $F_1(n, m, d) = T_1(n, m)$, where $F_1(n, m, d)$ can be represented by the following recurrence relation:

$F_1(n, m, 1) = m*[2*O(m) + O(n^2)] = O(m^2) + O(m*n^2)$

$F_1(n, m, d) = m*[2*O(m) + F_1(n, m, d\text{-}1)]$

$= m*O(m) + m*F_1(n, m, d\text{-}1)$

$= m*O(m) + m^2*O(m) + m^2*F_1(n, m, d\text{-}2)$

$= m*O(m) + m^2*O(m) + \ldots + m^i*O(m) + m^i*F_1(n, m, d\text{-}i)$

For $i = d\text{-}1$:

$F_1(n, m, d) = m*O(m) + m^2*O(m) + \ldots + m^{d-1}*O(m) + m^{d-1}*F_1(n, m, 1)$

$= O(m)*\sum_{i=1}^{d-1} m^i + m^{d-1}*O(m^2) + m^{d-1}*O(m*n^2)$

$= O(m)*[(m^d\text{-}1)/(m\text{-}1)] - O(m) + m^{d-1}*O(m^2) + m^{d-1}*O(m*n^2)$

$= O(m*m^{d-1}) + m^{d-1}*O(m^2) + m^{d-1}*O(m*n^2)$

$= O(m^d) + O(m^d*n^2)$

$= O(m^d*n^2)$

When we substitute d with $n(n\text{-}1)/2$: $T_1(n, m) = O(m^{n*(n-1)/2}*n^2)$. After calculating $T_1(n, m)$, $T(n, m)$ would be like:

$T(n, m) = O(n^2) + O(n^2*m^{n*(n-1)/2}) = O(n^2*m^{n*(n-1)/2})$.

As seen above, $T(n, m) = T_1(n, m)$. This means that the complexity of the blind search algorithm mainly depends on the complexity of the procedure graphGenerator.

3.6.2. Space Complexity

To calculate the memory space complexity of the blind search algorithm, we distinguish the data structure whose sizes are related to n and/or m. The size of both the arrays G and Gs are related to n. G is declared as an abstract data structure, while Gs is the output of the algorithm. The size of G is predefined (n x n), while the size of Gs becomes known at the end of the algorithm when all the generated graphs are saved into it. Denoting the memory space complexity as $\$(n, m)$, we can say that $\$(n, m)$ equals the sum of the of the complexities of G and Gs. Knowing that the complexity of G is $O(n^2)$, and supposing that $\$_1(n, m)$ is the complexity of Gs, $\$(n, m)$ would be written as:

$\$(n, m) = O(n^2) + \$_1(n, m)$

Supposing that $\$_1(n, m) = F_1(n, m, d)$, where d is the maximum depth of the recursivity of graphGenerator (i.e., $n(n\text{-}1)/2$). $F_1(n, m, d)$ can be defined as follows:

$F_1(n, m, 1) = m*O(n^2)$

$F_1(n, m, d) = m*F_1(n, m, d\text{-}1) = m^2*F_1(n, m, d\text{-}2) = m^i*F_1(n, m, d\text{-}i)$

For $i = d\text{-}1$:

$F_1(n, m, d) = m^{d-1}*F_1(n, m, 1) = m^d*O(n^2)$

When we substitute d with $n(n\text{-}1)/2$: $\$_1(n, m) = O(n^2*m^{n(n-1)/2})$.

$\$(n, m)$ in this case would be like:

$\$(n, m) = O(n^2) + O(n^2*m^{n(n-1)/2}) = O(n^2*m^{n(n-1)/2})$

As seen above, the space complexity mainly depends on the space occupied by *Gs* (i.e., the generated graphs), and the size of *G* has no effect on it.

3.7. Forward Algorithm

3.7.1. Time Complexity

The difference between the blind search algorithm and the forward algorithm is that, instead of saving a generated tournament graph, it is converted to a tournament table and then saved if the participants' points are in a descending order. This difference may affect the complexity of the procedure graphGenerator. Line 15 of graphGenerator in this case converts the data of *G* to a final tournament table state (i.e., *T*). We denote the complexity of this line as $T_2(n, m)$. Line 16 calculates the gained number of points by each participant of *T*. This line results in a complexity of $O(n*m)$. In line 17, the procedure checks whether the calculated points in line 16 are in a descending order or not. The resultant complexity from this line would be equal to $O(n)$. The generated state would not be saved unless its current occurrence is checked (see line 18). We denote the complexity of this process as $T_3(n, m)$. The last line of the procedure saves the contents of *T* into *Ts* (an array which contains all the generated states). This line results in a complexity of $O(n*m)$.

Supposing that $T(n, m)$ is the complexity of the forward algorithm. $T(n, m)$ is represented as: $T(n, m) = O(n^2) + T_1(n, m)$.

We suppose that $T_1(n, m) = F_1(n, m, d)$ is the complexity of the procedure graphGenerator, where *d* is the depth that the recursive calls of the procedure graphGenerator may reach (i.e., $n*(n\text{-}1)/2$). $F_1(n, m, d)$ would be represented as follows:

$$F_1(n, m, 1) = m*[O(n*m) + O(n) + O(n*m) + T_2(n, m) + T_3(n, m)]$$
$$= m*[O(n*m) + T_2(n, m) + T_3(n, m)]$$

$F_1(n, m, d) = m*[2*O(m) + F_1(n, m, d\text{-}1)]$

$= O(m)*\sum_{i=1}^{d-1} m^i + m^{d-1}*F_1(n, m, 1)$
$= O(m^d) + m^d*[O(n*m) + T_2(n, m) + T_3(n, m)]$

To continue calculating $F_1(n, m, d)$ (i.e., $T(n, m)$), each of $T_2(n, m)$ and $T_3(n, m)$ must be calculated.

3.7.1.1. $T_2(n, m)$

Line 15 of the procedure graphGenerator saves the corresponding state of G into T. This line can be represented by the following sub-algorithm:

```
Input: G[n][n]
Output: T[n][m]
1:  i, ind
2:  for k ← 1 to n do
3:    for j ← 1 to n do
4:      if k ≠ j then
5:        i ← G[k][j]
6:        ind ← the position of Ri in T
7:        T[k][ind]++
```

Line 6 of this sub-algorithm results in a complexity of $O(m)$. The process of this line is repeated $(n\text{-}1)^2$ times, which generates a total complexity (i.e., $T_2(n, m)$) of $O(m*n^2)$.

3.7.1.2. $T_3(n, m)$

Line 18 of the procedure graphGenerator verifies whether the generated tournament table state (i.e., T) has been calculated before or not. Line 18 can be illustrated by the following sub-algorithm:

```
Input: T[n][m], Ts[][n][m]
Output: true or false
1:  found ← false
2:  for i ← 1 to the size of Ts do
3:    if found = false then
4:      found ← compare(T, Ts[i])
5:    else
6:      return found

Procedure: compare(T1[][],T2[][])
```

```
2:  for i ← 1 to n do
3:    for j ← 1 to m do
4:      if T1[i][j] ≠ T2[i][j] then
4:        return false
1:  return true
```

It is clear that the procedure compare has the complexity $O(n*m)$. To calculate $T_3(n, m)$, the size of *Ts* which is required for the loop at line 2 must be defined. At the start of the forward algorithm, *Ts* would be empty and its number of elements grows up as far as the process approaches its end. The number of the current generated states is taken as a worst case for the number of the loop iterations at line 2. Since there is no possible way to determine the number of the generated states at a specific moment, we consider an upper value to be the worst case of the iterations of the loop at line 2. We suppose that each generated graph results in a valid tournament state. The number of the generated graphs, in this case, would be $m^{n*(n-1)/2}$. Considering this number of iterations at line 2 makes $T_3(n, m)$ equal to $m^{n*(n-1)/2}*O(n*m) = O(n*m^{[n*(n-1)/2]+1})$.

3.7.1.3. *T*(*n, m*)

After calculating $T_2(n, m)$ and $T_3(n, m)$, $F_1(n, m)$ would be like:

$$F_1(n, m, d) = O(m^d) + m^d*[O(n*m) + O(m*n^2) + O(n*m^{[n*(n-1)/2]+1})]$$
$$= m^d*O(n*m^{[n*(n-1)/2]+1})$$

Substituting d with $n(n-1)/2$, we get $T_1(n, m) = O(n*m^{n*(n-1)+1})$. After calculating $T_1(n, m)$, $T(n, m)$ would be $O(n^2) + O(n*m^{n*(n-1)+1}) = O(n*m^{n*(n-1)+1})$.

As seen above, $T(n, m) = T_1(n, m)$. This means that the complexity of the forward algorithm mainly depends on the complexity of the procedure graphGenerator. If we compare the complexity of the forward algorithm with the one of the blind search algorithm (i.e., $O(n^2*m^{n*(n-1)/2})$), we find that the complexity of this case is larger. This leads to the conclusion that checking the existence of every generated state (at line 18 of the procedure graphGenerator) has a remarkable effect on the execution time.

3.7.2. Space Complexity

The difference between the space complexity calculations of the forward algorithm and the blind search algorithm is that, instead of using *Gs* to save the generated graphs,

we use *Ts* to save the generated states. In addition to the array *G*, the forward algorithm uses the array *T* which contains the state obtained from the data of *G*. *G* and *T* have respectively the sizes *n* x *n* and *n* x *m*, and result in the complexities of $O(n^2)$ and $O(n*m)$ respectively. Supposing that $\$_1(n, m)$ is the memory space complexity of *Ts* and $\$(n, m)$ is the total space complexity of the forward algorithm, $\$(n, m)$ would be written as:

$\$(n, m) = O(n^2) + O(n*m) + \$_1(n, m)$

Supposing that $\$_1(n, m) = F_1(n, m, d)$, where *d* is the maximum depth of the recursivity of graphGenerator (i.e., $n(n\text{-}1)/2$). $F_1(n, m, d)$ can be defined as follows:

$F_1(n, m, 1) = m*O(n*m)$

$F_1(n, m, d) = m*F_1(n, m, d\text{-}1) = m^2*F_1(n, m, d\text{-}2) = m^i*F_1(n, m, d\text{-}i)$

For $i = d\text{-}1$:

$F_1(n, m, d) = m^{d-1}*F_1(n, m, 1) = m^d*O(n*m)$

Substituting *d* with $n(n\text{-}1)/2$: $\$_1(n, m) = O(n*m^{[n(n-1)/2]+1})$.

$\$(n, m)$ in this case would be like:

$\$(n, m) = O(n^2) + O(n*m) + O(n*m^{[n(n-1)/2]+1}) = O(n*m^{[n(n-1)/2]+1})$

The space complexity mainly depends on the occupied space by *Ts* (i.e., the generated final states of the tournament table), and neither of the sizes of *G* or *T* has an effect on it. $\$(n, m)$ represents the complexity of this algorithm for the worst case, i.e., all the generated graphs are converted to valid states, but this is not what really happens. We have supposed that every generated state is saved because there is no practical way to determine the number of the generated states based on *n* and *m*. So, the real complexity would be less than $O(n*m^{[n(n-1)/2]+1})$, but this value is still an upper bound.

3.8. Optimized Forward Algorithm

3.8.1. Time Complexity

The optimized forward algorithm differs from the forward algorithm by taking into account restrictions related to the maximum and minimum number of points that can be gained by each participant. Respecting these restrictions is checked after generating all the game results (i.e., the related graph edges) of a specific participant (see lines 13, 14 and 19 of the procedure graphGenerator). To check the point restrictions, information related to the maximum and minimum number of points of each participant is required.

This implies the need of preparing some new data structures (*maxP* at line 3 and *minP* at line 2) before calling the procedure graphGenerator at line 4 of the main procedure. Each of these lines generates a complexity of $O(n)$. Supposing that $T_1(n, m)$ is the resultant complexity by the procedure gaphGenerator, $T(n, m)$ in this case would be

$$T(n, m) = O(n^2) + 2*O(n) + T_1(n, m)$$
$$= O(n^2) + T_1(n, m)$$

To calculate $T_1(n, m)$, we determine the worst case of the procedure gaphGenerator. Since there is no useful way to know how many iterations of the loop at line 1 satisfy the conditions at lines 13, 14 and 19 (i.e., the maximum and minimum point restrictions), we consider that the conditions are satisfied in every iteration. Each of lines 12 and 18 of the procedure graphGenerator has a complexity of $O(m)$. As seen in Section 3.7.1, the complexities of each of lines 3, 5, 20, 21 and 22 of graphGenerator are respectively equal to $O(m)$, $O(m)$, $O(m*n^2)$, $O(n*m^{[n*(n-1)/2+1]})$ and $O(n*m)$. We suppose that $T_1(n, m) = F_1(n, m, d)$ is the complexity of the procedure graphGenerator, where d is the depth that the recursive calls of the procedure graphGenerator may reach (i.e., $n*(n-1)/2$). $F_1(n, m, d)$ would be represented as follows.

$$F_1(n, m, 1) = m*[O(m) + O(m*n^2) + O(n*m^{[n*(n-1)/2+1]}) + O(n*m)]$$
$$= m*O(n*m^{[n*(n-1)/2+1]})$$
$$F_1(n, m, d) = m*[3*O(m) + F_1(n, m, d\text{-}1)]$$

$$= O(m)*\sum_{i=1}^{d-1} m^i + m^{d-1}*F_1(n, m, 1)$$
$$= O(m^d) + m^d*[O(n*m^{[n*(n-1)/2+1]})]$$
$$= m^d*O(n*m^{[n*(n-1)/2]+1})$$

Substituting d with $n(n-1)/2$, we get $T_1(n, m) = O(n*m^{n*(n-1)+1})$. After calculating $T_1(n, m)$, $T(n, m)$ would be $T(n, m) = O(n^2) + O(n*m^{n*(n-1)+1}) = O(n*m^{n*(n-1)+1})$. As seen from the final value of $T(n, m)$, the complexities of the new achieved calculations by the main procedure (i.e., the line 2 and 3) do not affect the final complexity, where $T(n, m)$ is still directly related to $T_3(n, m)$ (the complexity of the procedure stateGenerator). Besides, the complexity of the optimized forward is the same as the one of the previous algorithm. The reason for this is the consideration that the conditions at lines 13, 14 and 19 are always satisfied.

3.8.2. Space Complexity

In addition to the considered data structures in the previous algorithm (see Section 3.6.3), in this version of the algorithm three new arrays are taken into account (i.e., *minP*, *maxP* and *Pts*), where each of them has a space complexity of $O(n)$. The space complexity $\$(n, m)$ of this case would be like

$$\$(n, m) = O(n^2) + O(n*m) + O(n*m^{[n(n-1)/2]+1}) + 3*O(n) = O(n*m^{[n*(n-1)/2]+1})$$

The space complexity remains the same in this case compared to the previous one. This can be explained by the fact that the new data structures have no effect on the complexity of the optimized forward algorithm, and $\$(n, m)$ is still depending on the size of *Ts*.

3.9. Multi-threaded Optimized Forward Algorithm

3.9.1. Time Complexity

Compared to the sequential optimized forward algorithm, the multi-threaded algorithm has no difference in terms of fundamental processes. The only remarkable difference is that, in the multi-threaded version, a recursive call of the procedure graphGenerator is invoked in a new thread (parallel) whenever the number of the current threads is less than the threads' limit (i.e., $thrLim > thrNbr$). To represent the complexity of the multi-threaded algorithm, we have to take *thrLim* into account as a new parameter. The complexity $T(n, m, thrLim)$ would be $T(n, m)/thrLim = O[(n*m^{n*(n-1)+1})/thrLim]$.

3.9.2. Space Complexity

The difference in this algorithm compared to the sequential one is that the neither of *G*, *Pts* or *T* are declared as static, but *G*, *Pts* are declared as local arrays in each of the main procedure and the procedure graphGenerator, and *T* is declared as a local array in the deepest recursive call of the procedure graphGenerator. *G*, *T* and *Pts* have respectively the space complexities of $O(n^2)$, $O(n*m)$ and $O(n)$. To calculate the resultant complexity by the procedure graphGenerator, we have to define the maximum depth of its recursive calls. The maximum depth equals to $\sum_{i=1}^{n-1} n\text{-}i = n*(n+1)/2$. So, the resultant

complexity by graphGenerator would be equal to $[n*(n+1)/2]*[O(n^2) + O(n)] + O(n*m) = O(n^4) + O(m*n)$.

Since the recursive calls of stateGenerator are directly related to the maximum number of threads (i.e., *thrLim*), there is a possibility that each thread reaches the maximum depth of the recursive calls of graphGenerator (i.e., a worst case). In this situation, the complexity related to *G*, *T* and *Pts* would be equal to $O[thrLim*(n^4+m*n)]$. The complexity $\$(n, m, thrLim)$ of the multithreaded algorithm would be represented as follows.

$$\$(n, m, thrLim) = O(n*m^{[n*(n-1)/2]+1}) + O[thrLim*(n^4+m*n)]$$

3.10. The Complexity Classes of Our Problem

3.10.1. The Class P

According to the calculated complexities in Sections 3.3, 3.4, 3.5, 3.7, 3.8 and 3.9, the problem of determining the final states of a tournament table has a non-deterministic polynomial complexity for each of the backward and forward approaches, in addition to their optimized and parallelized versions. So we can say that the problem does not belong to the class P.

3.10.2. The Class NP

We have showed in Section 3.10.1 that the problem has a non-deterministic polynomial complexity. To say that the problem belongs to class NP, its output (i.e., the generated states) must be able to be verified in a polynomial time. To verify whether a state generated by any of the proposed algorithms is valid or not, the participants' points must be in a descending order, and the state must result in a tournament graph.

It is obvious that checking the order of the participants has a complexity of $O(n*m)$. The verification of whether a tournament graph can be constructed based on a specific state or not is the task that is performed by the procedure stateChecker (see Algorithm for validating states at Section 2.6.1). The complexity of this procedure is calculated above (see Sections 3.3.1.2 and 3.3.1.3), which is equal to $O(m*n^{(n-1)*(n-1)+1})$. The required time

to check a generated state is not polynomial, which leads us to say that the problem does not belong to the NP class.

3.10.3. The Class NP-hard

We have mentioned in Section 2.6.2 (see Table 2.25) that a tournament graph can be transformed to a tournament table. It is possible to say that transforming a tournament graph into a tournament table is an operation of reduction (see Section 1.10.9). This reduction can easily be done in a polynomial time (i.e., $O(n*m)$), which means that if we prove that the problem of determining a complete graph is an NP-hard problem, the problem of determining a state of a tournament table will automatically be an NP-hard problem.

Based on Section 1.10.10, to prove that the problem of determining if a graph is complete is an NP-hard problem, we select a known NP-hard problem and reduce it to our problem (i.e., complete graph problem). Since we already know that SAT (i.e., Boolean satisfiability problem) is an NP-hard problem (see Section 1.10.11), we reduce it to a complete graph problem. Given the following CNF (i.e., conjunctive normal form):

$\phi = x_1 \wedge x_2 \wedge x_3$, the values x_1=1, x_2=1 and x_3=1 satisfies ϕ (i.e., $\phi = 1$). ϕ contains 3 clauses: $C_1 = L_{1,1} = x_1$, $C_2 = L_{2,1} = x_2$ and $C_3 = L_{3,1} = x_3$. To reduce the represented SAT problem by ϕ to a complete graph problem, we apply the presented rules in the example given in Section 1.10.11.

Transforming a CNF with n clauses, where each clause contains a single $L_{i,j}$ (i.e., $L_{i,1}$), into a graph requires a time complexity of $O(n^2)$. Thus, we can say that the reduction is done in polynomial time. By applying the transformation rules on ϕ, we get the represented graph in Figure 3.1.

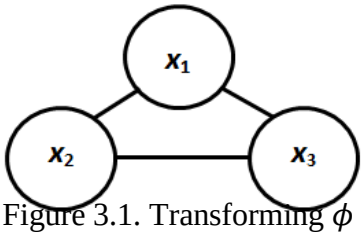

Figure 3.1. Transforming $\phi = x_1 \wedge x_2 \wedge x_3$ into a graph

The resultant graph from a CNF with *n* clauses contains *n* nodes (i.e, a complete graph). As we can see, the graph in Figure 3.1 is a complete one with 3 nodes. When we represent each node in the graph with 1 (i.e, a true value), ϕ must be satisfied. In this case, when $x_1=1$, $x_2=1$ and $x_3=1$, $\phi = 1 \wedge 1 \wedge 1 = 1$. Since we reduced a CNF problem into a complete graph problem in a polynomial time and the CNF problem is an NP-hard problem, we can say that the complete graph problem is an NP-hard problem. In accordance with the discussion at the beginning of this section, we can also say that the problem of determining a state of a tournament table belongs to the class NP-hard. Since the problem of calculating the possible states of final tables of sport competitions consists of determining all the states of a tournament table, we can say that the problem also belongs to the class NP-hard.

3.10.4. The Class NP-complete

We have showed above that the problem of calculating the tournament table states belongs to the class NP-hard, but it does not belong to the NP class. Thus, we can say that the problem does not belong to the class NP-complete.

4. EXPERIMENTAL RESULTS AND DISCUSSION

4.1. Introduction

Each of the proposed approaches (i.e., backward and forward), their optimized versions, and their multi-threaded optimized versions are programmed in the Java programming language. The source code of the approaches is compiled using Java Development Kit (JDK) 11, and the compiled code is run on Windows 7 Ultimate 64-bit operating system.

The experiments are performed for some distinct sets of single round-robin tournaments in several sports disciplines, which can have different numbers of teams and different organizations of final tables. During the experiments, in addition to the number of the generated states and the execution time, some other parameters related to the algorithms are measured for each final table.

The machine from which the experimental results are obtained is equipped with an Intel (R) Core (TM) i3-M330 2.13 GHz CPU with 4 cores, 6 GB of RAM and a Corsair (R) CSSD-F115GB2-A ATA hard drive device.

4.2. Backward Algorithm

4.2.1. The Case of *WDL*

Table 4.1 presents the results of the backward algorithm in the case of football tournaments, in which each played game can end in a win (*W*), draw (*D*) or lose (*L*), and each team gains 3 points for a win (i.e., P_W=3), a point for a draw (i.e., P_D=1) and no point for a loss (i.e., P_L=0). Besides the number of the possible states of the final table for each tournament, the presented results include the number of the checked invalid states and the execution time, where the number of teams ranges between 2 and 7 teams.

As explained in Section 2.6.1, the validity of every state generated by the backward algorithm is checked by the procedure stateChecker of the algorithm. For n=7, the process of checking the large number of the generated states for validity explains why it takes so much time to determine all the possible states. The backward algorithm fails to reach the

relevant results for a tournament with 8 teams. Eq. (2.30) tells that there would be 2821109907456 states (a total of valid and invalid ones) for n=8, which is about 208 times more than the states generated for n=7. This large number of states is evidently the main reason for the failure.

Table 4.1. Results of the backward algorithm for football tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	2	7	9	1 millisecond
3	7	209	216	2 milliseconds
4	40	9960	10000	22 milliseconds
5	404	758971	759375	1 second
6	6317	85759804	85766121	3 minutes
7	131288	13492797224	13492928512	284 hours

Table 4.2 presents the results of the same algorithm in the case of sports tournaments such as rugby and handball, where the team gains a point for a drawn game and 2 points (instead of 3 points as in the case of football) for a won game.

Table 4.2. Results of the backward algorithm for rugby and handball tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	2	7	9	1 millisecond
3	7	209	216	2 milliseconds
4	49	9951	10000	24 milliseconds
5	571	758804	759375	1 second
6	9981	85756140	85766121	3 minutes
7	223964	13492704548	13492928512	279 hours

Comparing the numbers of the valid states in Table 4.1, it can be easily seen that there occur the larger numbers of the states by starting with n=4 in Table 4.2. The reason behind this is that the use of a different point system (i.e., rewarding P_W=2 points for a

won game) affects the number of the corresponding points of the elements in the set S (see Eq. (2.20)). Speaking clearly, this point system lets some distinct *WDL* values result in the same number of points, while their corresponding points may be different in the previous system. For example, in the case of n=4, the number of points gained from a *WDL* value of (1, 1, 1) equals to 4 points when P_W=3, while it equals to 3 points when P_W=2. Thus, when P_W=2, if a team A has a *WDL* value of (1, 1, 1) and another team B has the value (0, 3, 0), A appears before B in the final table with the state ((2, 1, 0), (1, 1, 1), (0, 3, 0), (0, 1, 2)), or after B in the final table with the state ((2, 1, 0), (0, 3, 0), (1, 1, 1), (0, 1, 2)). However, when P_W=3, the latter state is considered as an invalid one, which explains why the number of valid states in Table 4.2 is larger than in Table 4.1.

There is a significant difference in the execution time of the valid states given for n=7 in Tables 4.1 and 4.2. A state is considered to be valid if a tournament graph can be constructed based on that state, while a state is said to be invalid if no graph can be found for that state after trying all the possible ways of building it up. So it often takes more time to identify an invalid state than a valid one. This is justified by the fact that the backward algorithm spends less time to determine each of the valid states for n=7, although there is a larger number of them in Table 4.2 than in Table 4.1. The advantage in term of the execution time which the case of P_W=2 has over the case of P_W=3 is only valid when n=7, since both algorithms cannot generate all the states for n>7.

Table 4.3 presents the results of the backward algorithm for chess tournaments, where the participant gains 1 point for a won game (i.e., P_W=1) and half a point for a draw (i.e., P_D=1/2).

Table 4.3. Results of the backward algorithm for chess tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	2	7	9	1 millisecond
3	7	209	216	3 milliseconds
4	49	9951	10000	25 milliseconds
5	571	758804	759375	1 second
6	9981	85756140	85766121	3 minutes
7	223964	13492704548	13492928512	280 hours

As seen in both Table 4.2 and Table 4.3, there are the same numbers of valid and invalid states. This arises due to the fact that the values of P_W and P_D for a chess tournament are exactly half of those for a rugby or handball tournament. The partition of the points gained from each game on the same number maintains the differences between the points of the participants, thereby allowing them to keep various positions in the final table. For example, in the point system with P_W=2 and P_D=1, there is one point difference between the participants in the state ((1, 1, 0), (1, 0, 1), (0, 1, 1)), as the gained points are 3, 2 and 1 respectively. Given the point system with P_W=1 and P_D=1/2, the gained points would be 3/2, 1 and 1/2 respectively, where the difference between the points changes in the ratio of two to one.

Although the numbers of the valid and invalid states are the same in Tables 4.2 and 4.3, the time execution differs only for n=7. This difference is likely because of the disparity of the CPU temperature during both tests [176-178], which have been performed in different environments with different weather conditions.

4.2.2. The Case of *WL*

Table 4.4 presents the results of the backward algorithm when the final table contains *W* and *L* columns only, where P_W=1 and P_L=0. The most noticeable sport in this tournament system is lacrosse. The results are obtained for the number of participating teams between 2 and 10.

The algorithm can calculate the number of the states up to n=10. The reason for this possibility is due to less table columns holding game results. As expressed by Eq. (2. 23), $|S(n, 2q+r)|$ is calculated based on the function f which has a direct relation with the number of the result-related columns (see Eq. (2.22)). As the number of these columns (i.e., $2q+r$) gets smaller, the recursive calls to the function f becomes less, leading to a smaller number of the elements of the set S (i.e., $|S|$). In this situation, the number of the states generated via a blind search algorithm will be less, which means that less time will be spent on calling the procedure stateChecker to verify their validity.

The results presented in Table 4.5 are obtained by the backward algorithm when the final table contains only *W* and *L* as the result columns, with P_W=2 and P_L=1 respectively. The most noticeable sport tournaments of this system are basketball, volleyball and tennis. The number of participating teams in this scenario ranges between 2 and 10.

Table 4.4. Results of the backward algorithm for lacrosse tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	1	3	4	0.4 millisecond
3	2	25	27	1 millisecond
4	4	252	256	2 milliseconds
5	9	3116	3125	3 milliseconds
6	22	46634	46656	18 milliseconds
7	59	823484	823543	2 seconds
8	167	16777049	16777216	57 seconds
9	490	387419999	387420489	34 minutes
10	1486	9999998514	10000000000	288 hours

Table 4.5. Results of the backward algorithm for basketball, volleyball and tennis tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	1	3	4	0.4 millisecond
3	2	25	27	1 millisecond
4	4	252	256	2 milliseconds
5	9	3116	3125	3 milliseconds
6	22	46634	46656	17 milliseconds
7	59	823484	823543	2 seconds
8	167	16777049	16777216	57 seconds
9	490	387419999	387420489	34 minutes
10	1486	9999998514	10000000000	290 hours

Although the results presented in Tables 4.4 and 4.5 are calculated through two different point systems, the numbers of valid and invalid states are the same. This is due to one point difference between the values of P_W and P_L in both cases. Adding the same number of points to each of P_W and P_L for all the participating teams maintains the same differences in their total number of points, thus ensuring that the participants keep the

same positions in the final table. For example, based on the point system with P_W=1 and P_L=0, the participants in the state ((3, 0), (2, 1), (1, 2), (0, 3)) gain a total number of points 3, 2, 1 and 0 respectively. In the point system with P_W=2 and P_L=1, the participants gain a total number of points 6, 5, 4 and 3 respectively. The difference in the total number of points between each two participants is still the same (i.e., 1 point) in both cases.

Tables 4.4 and 4.5 show the same results for n=10, but the execution times are different (i.e., a difference of 2 hours). As explained at the end of Section 4.2.1, these differences usually appear when the execution time is big, which create a big possibility that the tests are performed under different weather conditions. This may subsequently affect the CPU performance, thus creating contrast in the execution times.

4.2.3. The Case of *WXYL*

Table 4.6 shows the results of the same algorithm for the final tournament tables that include *W*, *X*, *Y* and *L* as the columns of game results. Such final tables occur in the case of ice hockey and curling tournaments. *X* is a unification symbol for the *OTW* column in ice hockey table and the *SOW* column in curling table, while *Y* is used as a unification symbol for the *OTL* column in ice hockey table and the *SOL* column in curling table. Each participant gains 3 points for a won game (i.e., P_W=3), 2 points for an *X* (i.e., P_X=2), a point for a *Y* (i.e., P_Y=1) and none for a lost game (i.e., P_L=0). The results of the backward algorithm are presented for the number of participants ranging between 2 and 6.

Table 4.6. Results of the backward algorithm for ice hockey and curling tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	2	14	16	1 millisecond
3	14	986	1000	6 milliseconds
4	239	159761	160000	228 milliseconds
5	8650	52513225	52521875	136 seconds
6	571990	30840407466	30840979456	95 hours

The algorithm, in this case, calculates the results up to n=6 only. Unlike the case where the tournament table includes only 2 game results (see Section 4.2.2), the reason behind the inability of the algorithm to calculate the results for n>6 is due to the increment in the number of the possible game results, which subsequently makes the number of the states generated by a blind search algorithm become bigger. As expressed by Eq. (2.22), when the number of game results (i.e., $2q+r$) increases there will be more recursive calls of the function f, which increases the number of elements of the set S (i.e., $|S|$), and therefore the number of the generated states based on a blind search algorithm will be increased as well, which requires spending more time to call the procedure stateChecker.

4.2.4. The Case of *WLTNr*

As in the previous case, the final tournament table in this instance also contains 4 columns of game results (i.e., *W*, *L*, *T* and *NR*). This case is usually seen in the cricket tournaments, where P_W=2, P_L=0, P_T=1 and P_{NR}=1. Table 4.7 shows the results of the backward algorithm for this case, where the number of participants ranges between 2 and 6.

Table 4.7. Results of the backward algorithm for cricket tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	3	13	16	1 millisecond
3	20	980	1000	6 milliseconds
4	393	159607	160000	217 milliseconds
5	18400	52503475	52521875	131 seconds
6	1769907	30839209549	30840979456	92 hours

Although the tournaments has the same number of the columns (i.e., 4 columns) related to game results, the results given in Tables 4.6 and 4.7 are different. This arises due to the difference in the nature of the columns. As in the previous case, the columns form the pairs which belong to the class C_1 (i.e., C_1={{*W*, *L*}, {*X*, *Y*}}), but in this case

only *W* and *L* form a pair which belongs to C_1 (i.e., C_1={{*W*, *L*}}), while each of *T* and *NR* belongs to C_2 (i.e., C_2={*T*, *NR*}). Besides, the values in *W* column are different, as a result of P_W=2. As expressed by Section 4.2.1, these differences have an effect on the number of points of the elements of the set *S*, thereby making the number of corresponding points of some elements equal. This reason makes some states, which are invalid in the previous case (i.e., *WXYL*), valid in this case (i.e., *WLTNr*).

The big difference between the valid states in Tables 4.6 and 4.7 for *n*=6 makes the execution times different. Given that the procedure stateChecker spends more time to check an invalid state than a valid state and there are bigger numbers of valid states for *n*=5 and *n*=6 in Table 4.7 compared to Table 4.6, these explain why the backward algorithm spends less time to calculate the results in the case of *WLTNr* in relation to *WXYL*.

4.3. Forward Algorithm

4.3.1. The Case of *WDL*

Table 4.8 presents the results of the forward algorithm in the case of football tournaments, where each game can end in a win (*W*), a draw (*D*) or a loss (*L*), and each team gains 3 points for a win (i.e., P_W=3), 1 point for a draw (i.e., P_D=1) and no point for a loss (i.e., P_L=0). Besides the number of the valid states of the final tournament tables, the presented results include the number of the repeated valid and invalid states as well as the execution time. The tests are carried out on a set of tournaments with the number of teams ranging between 2 and 7.

Comparing the execution times for *n*=6 and *n*=7, a large number of the states checked for *n*=7 justifies the increase in the required execution time as a result of the order of team points and existing states. The algorithm fails to reach the relevant results for a tournament with 8 teams. Since the algorithm is based on a blind search one, according to Eq. (2.31), the number of the generated states (the total of valid, repeated and invalid ones) for *n*=8 would be equal to 22876792454961. Such a big number of the states compared with those for *n*=7 (which is 2186 times less) explains the failure of the algorithm to calculate the results related to an 8-team tournament.

Table 4.8. Results of the forward algorithm for football tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	2	0	1	3	0.4 millisecond
3	7	1	19	27	1 milliseconds
4	40	23	666	729	9 milliseconds
5	404	851	57794	59049	61 milliseconds
6	6317	61410	14281180	14348907	12 seconds
7	131288	9675877	10450546038	10460353203	158 minutes

Table 4.9 presents the results of the same algorithm for sports tournaments such as rugby and handball, where the team gains 1 point for a drawn game and 2 points instead of 3 for a won game.

Table 4.9. Results of the forward algorithm for rugby and handball tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	2	0	1	3	0.4 millisecond
3	7	1	19	27	1 milliseconds
4	49	26	654	729	13 milliseconds
5	571	1184	57294	59049	73 milliseconds
6	9981	103321	14235605	14348907	13 seconds
7	223964	18754047	10441375192	10460353203	165 minutes

Given the numbers of the valid states in Tables 4.8 and 4.9, it can be observed that the numbers get bigger in Table 4.9 from n=4 on. As previously explained in Section 4.2.1, the reason for the increase in the number of the valid states in Table 4.9 is due to the point grading system, where P_W=2 is used instead of P_W=3. It subsequently affects the corresponding points of some elements of set S, thereby making the number of corresponding points of some elements the same (see Section 4.2.1). Therefore, some

states which are not previously considered to be valid for P_W=3 are considered to be valid in this case.

When the points of participants are in descending order in a calculated state, the state is considered as a valid one and checked by comparison with previous ones before it is saved, which will cause an increase in the execution time. Due to this reason, the big difference in the numbers of the valid states (whether they are repeated or not) for n=7 between Tables 4.8 and 4.9 makes the execution times bigger in Table 4.9. For n>3, it is easy to see that the number of the valid states in Table 4.9 is always bigger than the number of those in Table 4.8. This explains the failure of the algorithm to calculate the number of states with P_W=2 for n=8.

Table 4.10 presents the results of the forward algorithm in the case of tournaments such as chess, where each participant gains 1 point for a won game and half a point for a drawn game.

Table 4.10. Results of the forward algorithm for chess tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	2	0	1	3	0.4 millisecond
3	7	1	19	27	1 milliseconds
4	49	26	654	729	13 milliseconds
5	571	1184	57294	59049	73 milliseconds
6	9981	103321	14235605	14348907	13 seconds
7	223964	18754047	10441375192	10460353203	165 minutes

Tables 4.9 is exactly identical to Table 4.10, which occurs since the values of P_W and P_D in this case are exactly half of the values in the previous case. This maintains the differences in the total number of the points gained by the participants who therefore retain the same positions in the final table of the tournament.

4.3.2. The Case of *WL*

Table 4.11 shows the results of the forward algorithm in the case of lacrosse, where the final tournament table includes only the *W* and *L* columns (with the exclusion of *D* column), as well as the point system of P_W=1 and P_L=0. The number of teams in this situation ranges from 2 to 9.

The algorithm can calculate the results up to *n*=9, as justified by Eq. (2.31)). As the number of the possible game results (i.e., 2*q*+*r*) gets smaller (in this regard, 2), the number of the tournament graphs *GR*(*n*) generated by the blind search algorithm decreases. This means that the execution time required to generate all the states based on the forward algorithm will be less for 2*q*+*r*=2 than for 2*q*+*r*=3.

Table 4.11. Results of the forward algorithm for lacrosse tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	1	0	1	2	0.3 millisecond
3	2	1	5	8	1 millisecond
4	4	5	55	64	2 milliseconds
5	9	52	963	1024	4 milliseconds
6	22	507	32239	32768	16 milliseconds
7	59	11500	2085593	2097152	3 seconds
8	167	358666	268076623	268435456	6 minutes
9	490	24647272	68694828974	68719476736	27 hours

The results in the case of tournament tables of basketball, volleyball and tennis are presented in Table 4.12, where the final tournament table includes only the *W* and *L* columns, as well as the point system of P_W=2 and P_L=1.

Although there are differences between their values of P_W and P_L, Tables 4.11 and 4.12 have the same data. This similarity is due to the increase in the values of both P_W and P_L by one point in the case of Table 4.12. As explained in the backward algorithm (see Section 4.2.2), one point added to each of P_W and P_L for all the participants makes them

maintain the same differences in their total number of points, thus the same positions in the final table.

Table 4.12. Results of the forward algorithm for basketball, volleyball and tennis tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	1	0	1	2	0.3 millisecond
3	2	1	5	8	1 millisecond
4	4	5	55	64	2 milliseconds
5	9	52	963	1024	4 milliseconds
6	22	507	32239	32768	17 milliseconds
7	59	11500	2085593	2097152	3 seconds
8	167	358666	268076623	268435456	6 minutes
9	490	24647272	68694828974	68719476736	27 hours

4.3.3. The Case of *WXYL*

Table 4.13 shows the results of the forward algorithm in the case of ice hockey and curling tournaments where the final table includes 4 columns related to game results (i.e., *W*, *X*, *Y* and *L*), as well as the point system of P_W=3, P_X=2, P_Y=1 and P_L=0. The presented results are obtained for a number of teams ranging between 2 and 6.

As shown in Eq. (2.31), any increase in the number of game results (i.e., $2q+r$) increases the number of tournament graphs $GR(n)$ which can be generated by the blind search algorithm, thereby increases the execution time to calculate all the possible states. For n=7, based on Eq. (2.31), the forward algorithm must generate a total number of 4398046511104 states. This large number of states is apparently the main reason for the algorithm's failure to calculate all the states.

Table 4.13. Results of the forward algorithm for ice hockey and curling tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	2	0	2	4	0.4 millisecond
3	14	2	48	64	3 milliseconds
4	239	87	3770	4096	34 milliseconds
5	8650	14190	1025736	1048576	59 seconds
6	571990	4908802	1068261032	1073741824	15 minutes

4.3.4. The Case of *WLTNr*

Table 4.14 presents the results of the forward algorithm in the case of cricket, where the tournament table contains 4 columns of game results (i.e., *W*, *L*, *T* and *NR*). The point system of the tournament has the rewarding values of P_W=2, P_L=0, P_T=1, and P_{NR}=1. The number of teams (*n*) in Table 4.14 ranges between 2 and 6.

Table 4.14. Results of the forward algorithm for cricket tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	3	0	1	4	0.6 millisecond
3	20	1	43	64	4 milliseconds
4	393	106	3597	4096	44 milliseconds
5	18400	20171	1010005	1048576	63 seconds
6	1769907	9317482	1062654435	1073741824	16 minutes

Although the tournament tables in the case of *WXYL* and *WLTNr* have the same columns of game results (i.e., 4 ones), the diversity of their corresponding sets C_1 and C_2 makes the results of Tables 4.13 and 4.14 different. The difference between the valid states for *n*=5 and *n*=6 in Tables 4.13 and 4.14 also makes the execution times different. Since the algorithm generates more valid states in this case than in the case of *WXYL*, it spends more time in checking their existence among previously calculated ones.

Accordingly, it is expected that the number of the valid states in the case of *WLTNr* will also be more than that of *WXYL* for n=7. Thus the execution time will be huge, which explains why the algorithm fails to calculate the the related results.

4.4. Backward Algorithm Versus Forward Algorithm

Comparing the results of the backward and forward algorithms, both algorithms generate the same number of valid states of the final tournament tables for each tested case, except for the case of n=10 in Table 4.4 with respect to Table 4.11 and in Table 4.5 with respect to Table 4.12. However, the forward algorithm failed to calculate the results in each of Tables 4.11 and 4.12, while the backward algorithm is able to calculate them (see Tables 4.4 and 4.5). The execution times to calculate the results are not the same, as the forward algorithm is better compared to the backward algorithm for $2q+r > 2$, while the backward algorithm is better for $2q+r = 2$.

Table 4.15 shows a comparison between the execution times those are larger or equal 1 second for both algorithms (i.e., backward and forward), as Figure 4.1 shows a graphical representation of the decreased percentage in the execution time for these cases.

The values of the percentage decrease in the execution time presented in Table 4.15 prove the efficacy of the forward algorithm against the backward algorithm for $2q+r > 2$. In addition to the ability of the backward algorithm to calculate the results for n=10, it also shows better performance than the forward algorithm for $2q+r = 2$.

Table 4.15. The comparisons of the execution time in the backward algorithm against the forward algorithm

Tables	***n***	**Backward**	**Forward**	**Decreased percentage (%)**
4.1 vs. 4.8	5	1 second	61 milliseconds	93.9
	6	3 minutes	12 seconds	93.333
	7	284 hours	158 minutes	99.072
4.2 vs. 4.9	5	1 second	73 milliseconds	92.7
	6	3 minutes	13 seconds	92.777
	7	279 hours	165 minutes	99.014
4.3 vs. 4.10	5	1 second	73 milliseconds	92.7
	6	3 minutes	13 seconds	92.777
	7	280 hours	165 minutes	99.017
4.4 vs. 4.11 and 4.5 vs. 4.12	7	2 seconds	3 seconds	33.333
	8	57 seconds	6 minutes	84.166
	9	34 minutes	27 hours	97.901
4.6 vs. 4.13	5	136 seconds	59 seconds	56.617
	6	95 hours	15 minutes	99.736
4.7 vs. 4.14	5	131 seconds	63 seconds	51.908
	6	92 hours	16 minutes	99.71

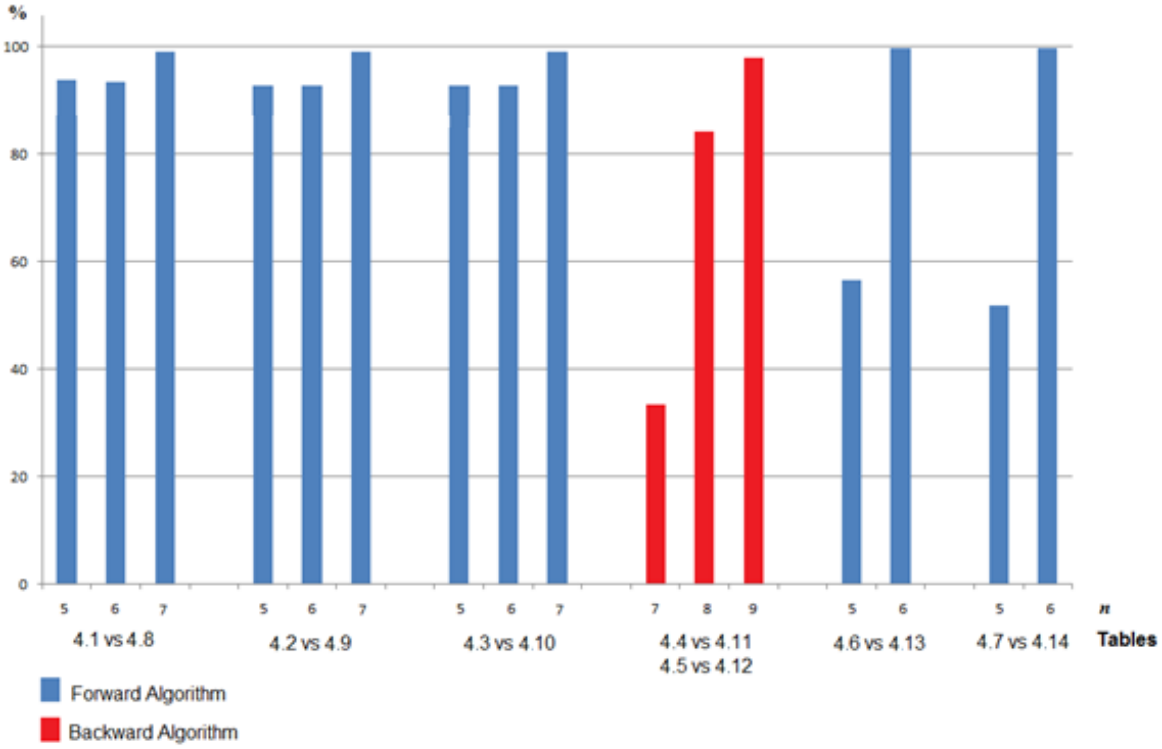

Figure 4.1. The percentage decrease in the execution time of the backward algorithm versus the forward algorithm

4.5. Optimized Backward Algorithm

4.5.1. The Case of *WDL*

Table 4.16 presents the results of the optimized backward algorithm for football tournaments, which have the point system of P_W=3, P_D=1 and P_L=0. The results include the possible states of the final tournament tables, the number of the checked invalid states and the execution time of each case, where the number of the participants ranges between 2 and 8.

Comparing the cases of n=7 to n=8, the large number of the states (a total of valid and invalid ones) checked by the procedure stateChecker for the latter case explains the large difference in the execution time. The number of generated states increases by approximately 3155%, which subsequently increases the execution time by about 77233%.

Table 4.17 presents the results of the same algorithm for sports like rugby and handball, which have the point system of P_W=2, P_D=1 and P_L=0. The results are obtained for the number of participants that ranges between 2 and 8.

Table 4.16. Results of the optimized backward algorithm for football tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	2	2	4	1 millisecond
3	7	33	40	2 milliseconds
4	40	583	623	10 milliseconds
5	404	11904	12308	96 milliseconds
6	6317	279595	285912	2 seconds
7	131288	7991894	8123182	9 minutes
8	3366444	261107702	264474146	116 hours

Table 4.17. Results of the optimized backward algorithm for rugby and handball tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	2	2	4	1 millisecond
3	7	41	48	2 milliseconds
4	49	675	724	12 milliseconds
5	571	15208	15779	126 milliseconds
6	9981	391337	401318	4 seconds
7	223964	12324568	12548532	17 minutes
8	6286424	436034441	442320865	189 hours

By comparing the numbers of the generated states (a total of valid and invalid ones) in Tables 4.16 and 4.17, it is observed that these numbers are larger in Table 4.17 starting from n=3 on. As previously explained in Section 4.2.1, the reason for this increase is attributed to the points awarded after a won game (i.e., 2 instead of 3), making the number of points of the participants approach each other and increasing the possibility that the generated states comply with Eq. (2.17) (i.e., $Pts(k) \geq Pts(k+1)$). The difference between the generated states in Tables 4.16 and 4.17 creates a significant one in the execution times starting from n=6 on, where the execution time is increased in Table 4.17 from 70% to 100%.

Table 4.18 consists of the results of the optimized backward algorithm for chess tournaments, where the participant gains 1 point for a won game (i.e., P_W=1) and half a point for a drawn game (i.e., P_D=1/2). The results are given for the number of participants which ranges between 2 and 8.

Table 4.18. Results of the optimized backward algorithm for chess tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	2	2	4	1 millisecond
3	7	41	48	2 milliseconds
4	49	674	723	12 milliseconds
5	571	15208	15779	132 milliseconds
6	9981	391337	401318	4 seconds
7	223964	12324568	12548532	18 minutes
8	6286424	436034441	442320865	193 hours

Tables 4.17 and 4.18 present the same numbers of the generated states (i.e., valid and invalid ones). This arises due to the values of both P_W and P_D in this occasion being exactly the half of the previous case, which makes the differences in the number of points between the participants stable. Despite the same numbers of generated states in Tables 4.17 and 4.18, there are some differences in the execution times for n=7 and n=8. The same situation is previously encountered in Section 4.2.1, where it is explained that when similar tests are performed in different environments with different weather conditions, their results may be different.

4.5.2. The Case of *WL*

Table 4.19 shows the results of the optimized backward algorithm for sports such as lacrosse, where the tournament tables contain only the W and L columns. The values of P_W and P_L are 1 and 0 respectively. The results are given for the number of participants which ranges between 2 and 10.

Table 4.19. Results of the optimized backward algorithm for lacrosse tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	1	0	1	0.4 millisecond
3	2	2	3	1 millisecond
4	4	8	12	2 milliseconds
5	9	36	45	3 milliseconds
6	22	139	161	15 milliseconds
7	59	547	606	143 milliseconds
8	167	2102	2269	14 seconds
9	490	8139	8629	23 minutes
10	1486	31395	32881	16 hours

Table 4.20 presents the results of the optimized backward algorithm in the case of sports like basketball, volleyball and tennis, where the tournament table includes W and L columns of game results, and each participant gains 2 points for a win (i.e., P_W=2) and 1 point for a loss (i.e., P_L=1). The presented results are valid for the number of participants between 2 and 10.

Table 4.20. Results of the optimized backward algorithm for basketball, volleyball and tennis tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	1	0	1	0.4 millisecond
3	2	2	3	1 millisecond
4	4	8	12	2 milliseconds
5	9	36	45	3 milliseconds
6	22	139	161	16 milliseconds
7	59	547	606	167 milliseconds
8	167	2102	2269	15 seconds
9	490	8139	8629	23 minutes
10	1486	31395	32881	16 hours

Although the results presented by Tables 4.19 and 4.20 are based on different values of P_W and P_L, we observe that they have the same numbers of the generated states. This occurs as a result of 1 point added to each of P_W and P_L for all the participants of the tournaments evaluated in Table 4.20. Thus, the differences between the points of the participants remain the same and the participants maintain the same positions in the final table.

4.5.3. The Case of *WXYL*

Table 4.21 shows the results of the optimized backward algorithm for tournament tables with *W*, *X* (*OTW* or *SOW*), *Y* (*OTL* or *SOL*) and *L* columns, such as ice hockey and curling, where the values of P_W, P_X, P_Y and P_L are 3, 2, 1 and 0 respectively. The results are given for participants ranging from 2 to 7.

Table 4.21. Results of the optimized backward algorithm for ice hockey and curling tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	2	2	4	1 millisecond
3	14	154	168	4 milliseconds
4	239	8522	8761	62 milliseconds
5	8650	833081	841731	2 seconds
6	571990	108045570	108617560	18 minutes
7	61551170	19969829838	20031381008	316 hours

Compared to the previous cases where there are 2 and 3 columns of game results, the optimized backward algorithm cannot get the results for more than 7 participants. The reason is that, as the number of the columns (in this case, 4 ones) related to game results increases, the number of the generated states also increases, which requires more time to process them by the procedure stateChecker.

4.5.4. The Case of *WLTNr*

Table 4.22 shows the results of the optimized backward algorithm for cricket tournaments, where the final table contains 4 columns (i.e., *W*, *L*, *T* and *NR*), based on the point system of P_W=2, P_L=0, P_T=1 and P_{NR}=1. The presented results are for the number of teams which ranges between 2 and 7.

Table 4.22. Results of the optimized backward algorithm for cricket tournaments

n	**Valid states**	**Invalid states**	**Total of states**	**Execution time**
2	3	6	9	1 millisecond
3	20	260	280	5 milliseconds
4	393	15687	16080	71 milliseconds
5	18400	1560772	1579172	5 seconds
6	1769907	223658929	225428836	38 minutes
7	267636358	45230865354	45498501712	748 hours

The difference in the nature of the columns and their corresponding points in the case of Tables 4.21 and 4.22 makes the algorithm generate different results. Starting from *n*=5 on, the difference in the execution times is due to the large difference in the number of the generated states. This arises because, when there are more generated states, there are more calls to the procedure stateChecker, costing more time.

4.6. Backward Algorithm Versus Optimized Backward Algorithm

Given the above results of the backward algorithm and the optimized backward algorithm, it can be observed that both algorithms generate the same numbers of the valid states of the final tournament tables for the tested cases. The case is not the same when the algorithms are applied to the invalid state; that is, the numbers significantly decrease after the application of the optimized backward algorithm. Table 4.23 shows a comparison between the numbers of the generated invalid states, while Figure 4.2 shows

a graphical representation of the decreased percentage in generating the invalid states for each the tested cases.

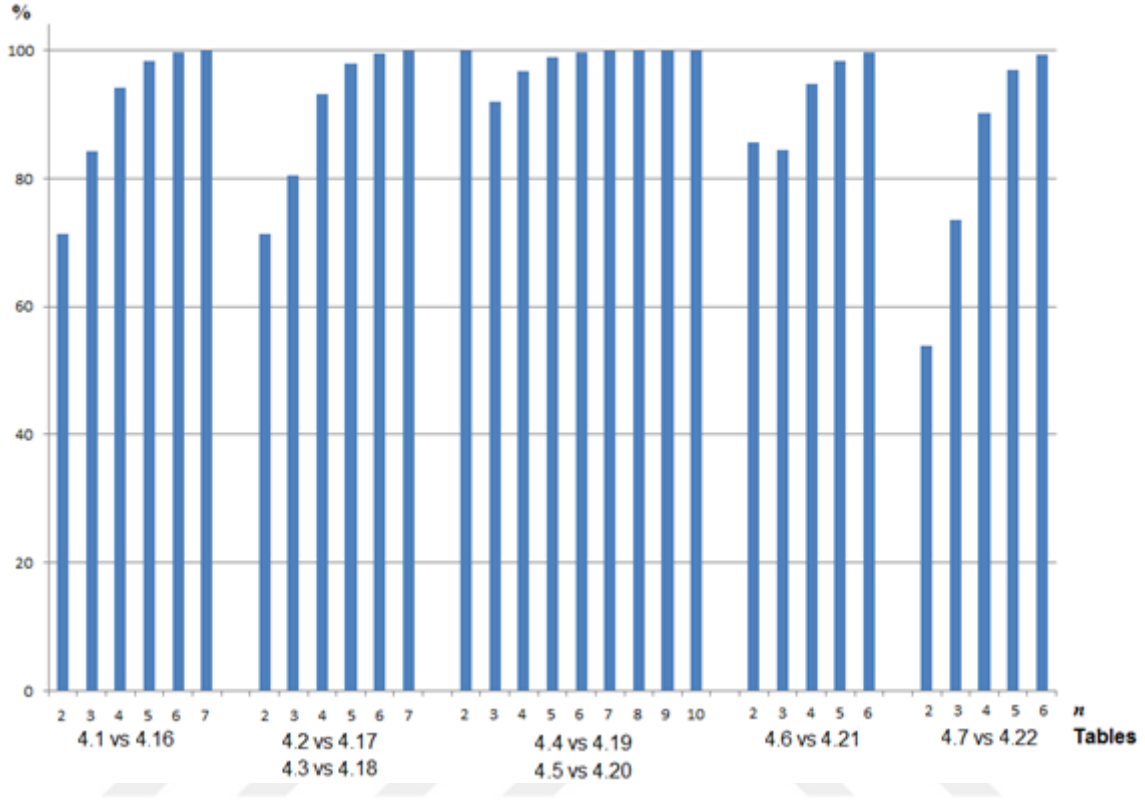

Figure 4.2. The percentage decrease of the generated invalid states in the optimized backward algorithm compared to the backward algorithm

The values of the percentage decrease in Table 4.23 prove the efficacy of the optimized backward algorithm against the backward algorithm, reducing the number of invalid states from 53.846% to 100%. This also explains the reason for the optimization in the execution time, especially when the number of the participants is large. The optimizations of the execution times those are larger or equal 1 second are presented in Table 2.24, while Figure 4.3 shows a graphical representation of the decrease percentage in the execution time for these cases.

Table 4.23. The number of the invalid states in the backward algorithm against the optimized backward algorithm

Tables	***n***	**Backward**	**Optimized backward**	**Decrease percentage (%)**
4.1 vs. 4.16	2	7	2	71.428
	3	209	33	84.21
	4	9960	583	94.146
	5	758971	11904	98.431
	6	85759804	279595	99.673
	7	13492797224	7991894	99.94
4.2 vs. 4.17 and 4.3 vs. 4.18	2	7	2	71.428
	3	209	41	80.382
	4	9951	675	93.216
	5	758804	15208	97.995
	6	85756140	391337	99.543
	7	13492704548	12324568	99.908
4.4 vs. 4.19 and 4.5 vs. 4.20	2	3	0	100
	3	25	2	92
	4	252	8	96.825
	5	3116	36	98.844
	6	46634	139	99.701
	7	823484	547	99.933
	8	16777049	2102	99.987
	9	387419999	8139	99.997
	10	9999998514	31395	99.999
4.6 vs. 4.21	2	14	2	85.71
	3	986	154	84.381
	4	159761	8522	94.665
	5	52513225	833081	98.413
	6	30840407466	108045570	99.649
4.7 vs. 4.22	2	13	6	53.846
	3	980	260	73.469
	4	159607	15687	90.171
	5	52503475	1560772	97.027
	6	30839209549	223658929	99.274

Table 4.24. The execution time in the backward algorithm against the optimized backward algorithm

Tables	***n***	**Backward**	**Optimized backward**	**Time optimization (%)**
4.1 vs. 4.16	5	1 second	96 milliseconds	90.4
	6	3 minutes	2 second	98.888
	7	284 hours	9 minutes	99.947
4.2 vs. 4.17	5	1 second	126 milliseconds	87.4
	6	3 minutes	5 second	97.222
	7	279 hours	17 minutes	99.898
4.3 vs. 4.18	5	1 second	132 milliseconds	86.8
	6	3 minutes	5 second	97.222
	7	280 hours	18 minutes	99.892
4.4 vs. 4.19	7	2 second	143 milliseconds	92.85
	8	57 second	14 second	75.438
	9	34 minutes	23 minutes	32.352
	10	288 hours	16 hours	94.444
4.5 vs. 4.20	7	2 second	167 milliseconds	91.65
	8	57 second	15 second	73.684
	9	34 minutes	23 minutes	32.352
	10	290 hours	16 hours	94.482
4.6 vs. 4.21	5	136 second	2 second	98.529
	6	95 hours	18 minutes	99.684
4.7 vs. 4.22	5	131 second	5 second	96.183
	6	92 hours	38 minutes	99.311

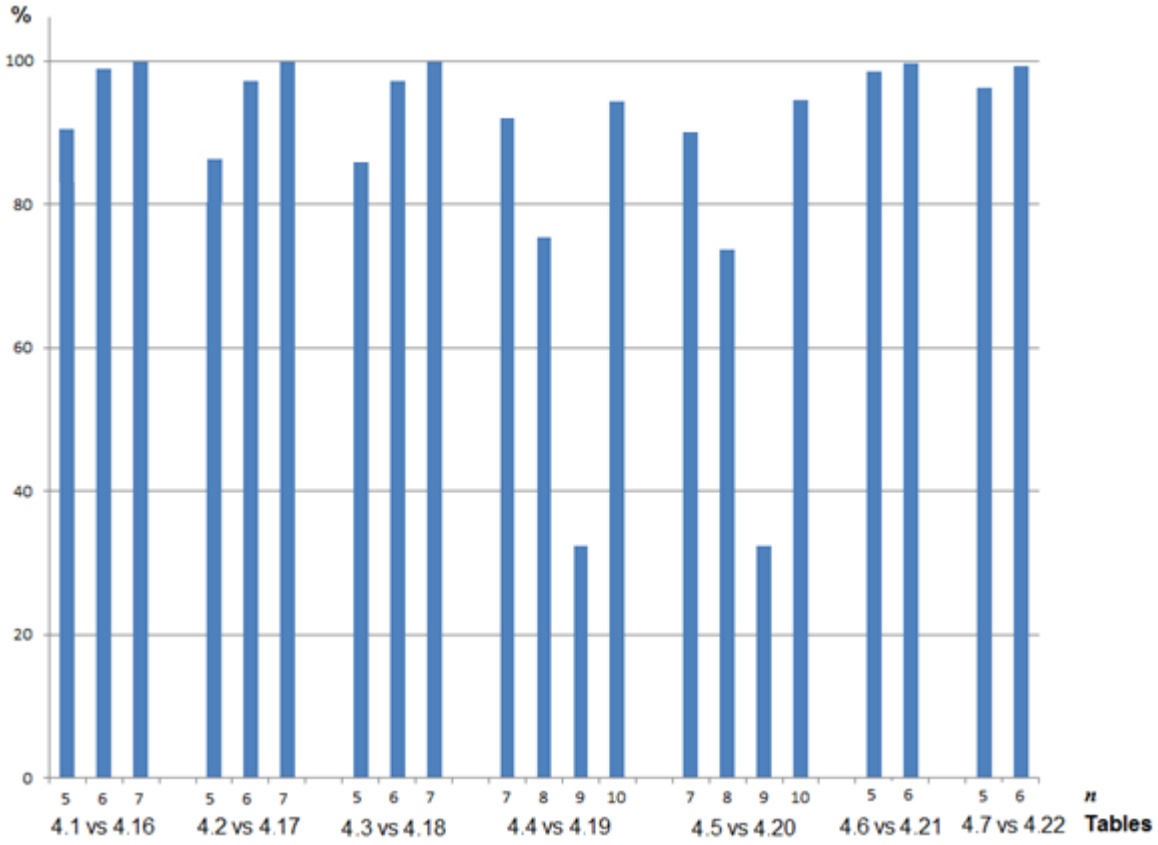

Figure 4.3. The execution time decrease in the optimized backward algorithm comparing to the backward algorithm

The reduction in the number of the invalid states enables the optimized backward algorithm to calculate the results of tournaments with a higher number of participants. It is not possible to calculate the related results of these tournaments via the backward algorithm, like the case of n=8 in Tables 4.16, 4.17 and 4.18, and n=7 in Tables 4.21 and 4.22.

4.7. Optimized Forward Algorithm

4.7.1. The Case of *WDL*

Table 4.25 presents the results of the optimized forward algorithm for sports tournament tables with *W*, *D* and *L* columns, such as football, where P_W=3, P_D=1 and P_L=0. In addition to the values of the execution time, the results shown in Table 4.25

include the numbers of the generated valid states, the repeated valid states, and the invalid states. The number of participating teams ranges between 2 and 8.

Table 4.25. Results of the optimized forward algorithm for football tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	2	0	0	2	1 millisecond
3	7	1	0	8	2 milliseconds
4	40	23	0	63	8 milliseconds
5	404	851	0	1255	58 milliseconds
6	6317	61410	0	67727	1 second
7	131288	9675877	0	9807165	5 minutes
8	3366444	3863316144	0	3866682588	238 hours

The optimized forward algorithm eliminates the generation of the invalid states. As the value of n increases, the number of the repeated states whose existence is checked also increases, affecting the execution time. This explains the big difference in the execution time between the cases of n=7 and n=8.

Table 4.26 presents the results of the same algorithm for sports with P_W=2, P_D=1 and P_L=0 such as rugby and handball. Gaining 2 points instead of 3 after a won game ensures that some states, which are not considered to be valid for P_W=3, are considered to be valid for P_W=2. For this reason, the numbers of the generated states (a total of valid and repeated ones) are larger in Table 4.26 compared to Table 4.25, starting from n=4 on. For n=8, the significant difference in the execution times in Tables 4.25 and 4.26 (which is 313 hours) is due to the large difference in the total of the generated states in Tables 4.26, where the existence of 4963135241 additional states is checked.

Table 4.27 shows the results of the optimized forward algorithm for chess tournaments, where the participants earn 1 point for a won game and half a point for a drawn game. The results in Tables 4.26 and 4.27 are similar except for some differences in the execution time, which are largely caused by the tests being performed in different environments under different weather conditions. The reason for the number similarities in Tables 4.26 and 4.27 is due to the values of P_W and P_D (i.e., 1 and 1/2 respectively)

which are exactly half of the values in the previous case (i.e., P_W=2 and P_D=1), thereby differences in total points of the participants remain the same, with no change in their positions in the final table.

Table 4.26. Results of the optimized forward algorithm for rugby and handball tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	2	0	0	2	1 millisecond
3	7	1	0	8	2 milliseconds
4	49	26	0	75	10 milliseconds
5	571	1184	0	1755	68 milliseconds
6	9981	103321	0	113302	2 seconds
7	223964	18754047	0	18978011	9 minutes
8	6286424	8823531405	0	8829817829	551 hours

Table 4.27. Results of the optimized forward algorithm for chess tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	2	0	0	2	1 millisecond
3	7	1	0	8	2 milliseconds
4	49	26	0	75	10 milliseconds
5	571	1184	0	1755	68 milliseconds
6	9981	103321	0	113302	2 seconds
7	223964	18754047	0	18978011	10 minutes
8	6286424	8823531405	0	8829817829	563 hours

4.7.2. The Case of *WL*

Table 4.28 shows the results of the optimized forward algorithm for lacrosse tournaments, where the table contains *W* and *L* columns and the point system of P_W=1 and

P_L=0. The presented results are given for tournaments with the number of participants between 2 and 10. Since the optimized forward algorithm does not generate invalid states, the execution time is reduced compared to the forward algorithm, thus providing the possibility of calculating the results with 10 participants.

Table 4.28. Results of the optimized forward algorithm for lacrosse tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	1	0	0	1	0.3 millisecond
3	2	1	0	3	0.8 millisecond
4	4	5	0	9	1 milliseconds
5	9	52	0	61	3 milliseconds
6	22	507	0	529	13 milliseconds
7	59	11500	0	11559	139 milliseconds
8	167	358666	0	358833	10 seconds
9	490	24647272	0	24647762	26 minutes
10	1486	2666134835	0	2666136321	196 hours

The results of the optimized forward algorithm for sports such as basketball, volleyball and tennis are presented in Table 4.29, where each participant gains 2 points for a win (i.e., P_W=2) and 1 point for a loss (i.e., P_L=1).

Although the results presented Tables 4.28 and 4.29 are based on the different values of P_W and P_L, we can see that they have the same numbers of valid and repeated states. This is due to the fact that adding 1 point to each of P_W and P_L for all the participants maintains the same differences in their total number of points, thus staying in the same positions in the final table. However, there is a difference in the execution time for n=8.

Table 4.29. Results of the optimized forward algorithm for basketball, volleyball and tennis tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	1	0	0	1	0.3 millisecond
3	2	1	0	3	0.8 millisecond
4	4	5	0	9	1 milliseconds
5	9	52	0	61	3 milliseconds
6	22	507	0	529	13 milliseconds
7	59	11500	0	11559	141 milliseconds
8	167	358666	0	358833	10 seconds
9	490	24647272	0	24647762	26 minutes
10	1486	2666134835	0	2666136321	201 hours

4.7.3. The Case of *WXYL*

Table 4.30 shows the results of the optimized forward algorithm for ice hockey and curling tournaments. The tournament table has 4 columns of possible game results (i.e., *W*, *X*, *Y* and *L*), where P_W=3, P_X=2, P_Y=1, and P_L=0. The results are given for the number of participants between 2 and 7.

Table 4.30. Results of the optimized forward algorithm for ice hockey and curling tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	2	0	0	2	1 millisecond
3	14	2	0	16	3 milliseconds
4	239	87	0	326	34 milliseconds
5	8650	14190	0	22840	316 milliseconds
6	571990	4908802	0	5480792	106 seconds
7	61551170	381554986	0	443106156	63 hours

Compared to the cases where there are 2 and 3 columns of game results, the optimized forward algorithm cannot calculate the results for more than 7 participants. This is due to the increase in the number of the columns related to game results (in this case, 4 ones). As the number of the columns increases, the number of generated states also increases and thus additional time will be spent on checking the existence of the generated states.

4.7.4. The Case of *WLTNr*

Table 4.31 presents the results of the optimized forward algorithm for cricket tournaments. The tournament table contains 4 columns of game results (i.e., *W*, *L*, *T* and *NR*), where P_W=2, P_L=0, P_T=1 and P_{NR}=1. The number of teams (*n*) ranges between 2 and 6.

Table 4.31. Results of the optimized forward algorithm for cricket tournaments

n	**Valid states**	**Repeated states**	**Invalid states**	**Total of states**	**Execution time**
2	3	0	0	3	1 millisecond
3	20	1	0	21	4 milliseconds
4	393	106	0	499	44 milliseconds
5	18400	20171	0	38571	1 second
6	1769907	9317482	0	11087389	144 seconds

The difference in the nature of the columns and their corresponding points in the cases presented Tables 4.30 and 4.31 causes the algorithm to generate different results. The large difference in the number of the generated states starting from *n*=5 in Tables 4.30 and 4.31 makes the execution times different. This is due to the generation of more valid states, which requires extra time to check their status (i.e., already generated or not). The size of memory required to save the valid states in the case of *n*=7 exceeds the one provided by the used machine, which results in the failure of the optimized forward algorithm to generate all the states.

4.8. Forward Algorithm Versus Optimized Forward Algorithm

Given the above results of the forward and the optimized forward algorithms, for each tested case, it can be observed that both algorithms generate the same numbers of valid and repeated valid states of the final tournament table. However, the optimized algorithm eliminates the generation of the invalid states, which has a positive effect on the required execution time to generate all the possible states.

The effect of optimization on the execution times those are larger or equal 1 second is presented in Table 4.32, while Figure 4.4 shows its graphical presentation. Eliminating the generation of the invalid states allows the optimized forward algorithm to calculate the results of tournaments that cannot be calculated by the forward algorithm, such as the cases of n=8 for *WDL* (see Tables 4.25, 4.26 and 4.27), n=10 for *WL* (see Tables 4.28 and 4.29) and n=7 for *WXYL* (see Table 4.30).

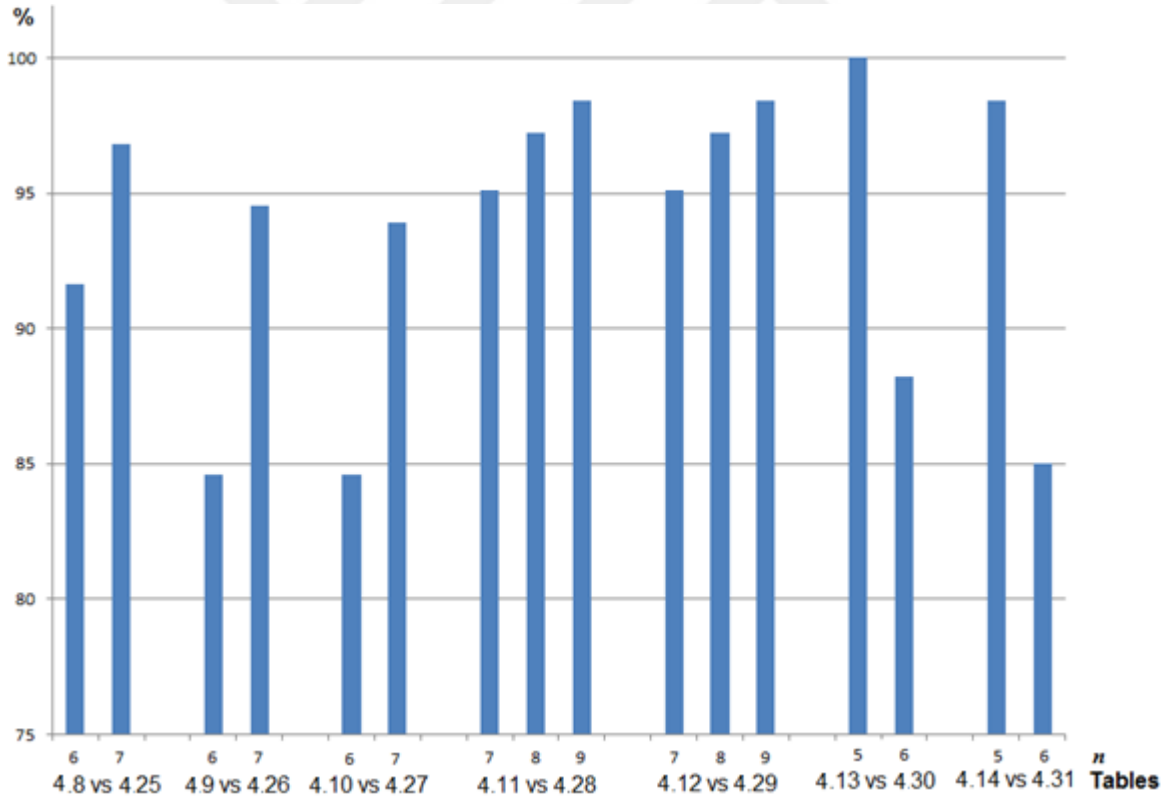

Figure 4.4. The execution time decrease in the optimized forward algorithm compared to the forward algorithm

Table 4.32. The execution time for the forward algorithm against the optimized forward algorithm

Tables	*n*	Forward	Optimized Forward	Time optimization (%)
4.8 vs. 4.25	6	12 seconds	1 second	91.666
	7	158 minutes	5 minutes	96.835
4.9 vs. 4.26	6	13 seconds	2 seconds	84.615
	7	165 minutes	9 minutes	94.545
4.10 vs. 4.27	6	13 seconds	2 seconds	84.615
	7	165 minutes	10 minutes	93.939
4.11 vs. 4.28	7	3 seconds	139 milliseconds	95.366
	8	6 minutes	10 seconds	97.222
	9	27 hours	26 minutes	98.395
4.12 vs. 4.29	7	3 seconds	141 milliseconds	95.3
	8	6 minutes	10 seconds	97.222
	9	27 hours	26 minutes	98.395
4.13 vs. 4.30	5	59 seconds	316 milliseconds	99.464
	6	15 minutes	106 seconds	88.222
4.14 vs. 4.31	5	63 seconds	1 second	98.412
	6	16 minutes	144 seconds	85

The percentage values of the time optimization in Table 4.32 prove the efficacy of the optimized forward algorithm against the forward algorithm, where the optimized forward algorithm reduces the execution time from 84.615% to 99.464%, which explains the importance of eliminating the generation of the invalid states.

4.9. Optimized Backward Algorithm Versus Optimized Forward Algorithm

Compared the results of optimized backward and optimized forward algorithms, both algorithms generate the same numbers of possible states of the final tournament table for each tested case, except for n=7 in the case of *WLTNr*, where the optimized forward algorithm cannot calculate the results, while the optimized backward algorithm

can calculate them (see Table 4.22). In terms of the execution time, the optimized forward algorithm performs better compared to the backward algorithm in the case of $n<8$, the reverse is the case for $n>8$. In the case of $n=8$, the optimized forward algorithm performs better for $2q+r=2$. In contrast, for $2q+r=3$, the optimized backward algorithm performs better.

Table 4.33 shows a comparison of the execution times those are larger or equal 1 second between the optimized backward and the optimized forward algorithms, while Figure 4.5 shows a graphical representation of the percentage decrease in execution time for these cases.

The values of the percentage decrease in execution time presented in Table 4.33 shows the efficacy of the optimized forward algorithm against the optimized backward for $n<8$, while the backward algorithm is more efficient for $n>8$. Given $n=8$, the execution times are better for $2q+r=2$ in the case of the optimized forward algorithm, while for $2q+r=3$, the execution times are better in the case of the optimized backward algorithm.

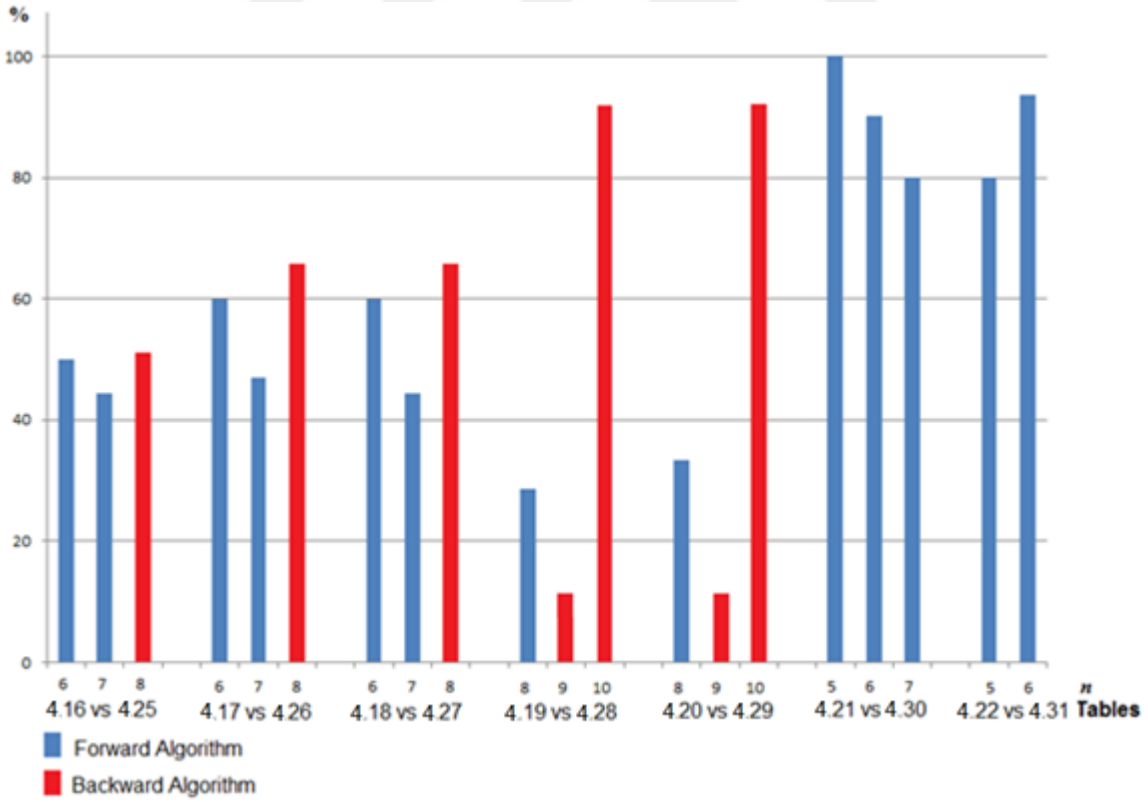

Figure 4.5. The percentage decrease in execution time for the optimized backward algorithm against the optimized forward algorithm

Table 4.33. The execution time for the optimized backward algorithm against the optimized forward algorithm

Tables	***n***	**Optimized Backward**	**Optimized Forward**	**Decrease percentage (%)**
4.16 vs. 4.25	6	2 seconds	1 second	50
	7	9 minutes	5 minutes	44.444
	8	116 hours	238 hours	51.26
4.17 vs. 4.26	6	5 seconds	2 seconds	60
	7	17 minutes	9 minutes	47.058
	8	189 hours	551 hours	65.698
4.18 vs. 4.27	6	5 seconds	2 seconds	60
	7	18 minutes	10 minutes	44.444
	8	193 hours	563 hours	65.719
4.19 vs. 4.28	8	14 seconds	10 seconds	28.571
	9	23 minutes	26 minutes	11.538
	10	16 hours	196 hours	91.836
4.20 vs. 4.29	8	15 seconds	10 seconds	33.333
	9	23 minutes	26 minutes	11.538
	10	16 hours	204 hours	92.156
4.21 vs. 4.30	5	2 seconds	316 milliseconds	84.2
	6	18 minutes	106 seconds	90.185
	7	316 hours	63 hours	80.063
4.22 vs. 4.31	5	5 seconds	1 second	80
	6	38 minutes	144 seconds	93.684

4.10. Multi-threaded Optimized Backward Algorithm

To exploit the maximum performance of the used machine for testing, the optimum number of threads must be determined. As explained in Section 2.9.1, to determine this number, we iteratively test the multi-threaded optimized backward algorithm for the previously presented sports, where the value of *thrLim* (the limit of threads that can run

simultaneously) ranges between 1 and 20 threads. The tests are done on tournaments with the particular numbers of participants which can easily allow the observation of the differences in the execution time. In the other words, the execution time of each test must not be as small as the difference between the tested cases cannot be observed, and must not be as large as the results of all the tests cannot be obtained. Table 4.34 shows the value selected for *n* in the case of each sports discipline in the performed tests.

Table 4.34. The value selected for *n* in the case of each sports discipline in the performed tests to determine the optimum number of threads

Football	**Chess**	**Rugby and handball**	**Lacrosse**	**Basketball, volleyball and tennis**	**Ice hockey and curling**	**Cricket**
7	7	7	9	9	6	6

Table 4.35 presents the duration in seconds taken by the multi-threaded backward algorithm in each performed test. When the value of *thrLim* equals 10, the performance of the multi-threaded optimized backward algorithm reaches its maximum for all the presented sports disciplines in Section 1.8.3, even though the used machine contains 4 cores.

Considering 10 threads, which are the optimum number of threads that can be run in parallel in the machine used for testing, Table 4.36 presents the required execution times to get the results of the multi-threaded optimized backward algorithm for the previously presented types of sports tournament tables.

Table 4.37 shows a comparison between the execution times those are larger or equal 1 second for the optimized backward algorithm against the multi-threaded optimized backward algorithm, while Figure 4.6 shows a graphical representation of the percentage decrease in the execution time for the different cases.

Table 4.35. The duration in seconds taken by the multi-threaded optimized backward algorithm in each performed test to determine the optimum number of threads

thrLim	**Football**	**Chess**	**Rugby and handball**	**Lacrosse**	**Basketball, volleyball and tennis**	**Ice hockey and curling**	**Cricket**
1	554	1107	1031	1418	1382	1107	2280
2	513	1029	990	1353	1319	1082	2236
3	495	995	922	1282	1266	1062	2211
4	460	954	908	1182	1168	1030	2192
5	437	896	847	1116	1113	1023	2175
6	410	868	826	989	1038	976	2171
7	391	821	765	920	973	962	2163
8	370	753	744	843	912	964	2128
9	329	713	692	798	871	969	2116
10	312	678	663	784	817	941	2084
11	320	696	701	816	838	962	2114
12	341	740	730	879	906	964	2161
13	357	767	739	941	960	1012	2216
14	374	813	762	984	986	1023	2241
15	403	860	807	1046	1034	1021	2253
16	423	886	829	1086	1124	1083	2280
17	452	922	865	1147	1199	1093	2303
18	473	965	879	1187	1243	1132	2354
19	489	989	924	1256	1295	1161	2381
20	506	1042	952	1309	1363	1236	2438

Table 4.36. The execution times of the multi-threaded optimized backward algorithm

n	**Football**	**Chess**	**Rugby and handball**	**Lacrosse**	**Basketball, volleyball and tennis**	**Ice hockey and curling**	**Cricket**
2	2 ms	2 ms	2 ms	2 ms	2 ms	2 ms	3 ms
3	8 ms	8 ms	8 ms	4 ms	4 ms	11 ms	16 ms
4	21 ms	25 ms	25 ms	6 ms	6 ms	97 ms	152 ms
5	182 ms	214 ms	211 ms	12 ms	12 ms	6 s	10 s
6	2 s	3 s	3 s	33 ms	33 ms	16 m	35 m
7	5 m	11 m	11 m	309 ms	307 ms	243 h	628 h
8	76 h	125 h	124 h	12 s	13 s	-	-
9	-	-	-	13 m	13 m	-	-
10	-	-	-	13 h	13 h	-	-

Table 4.37. The percentage decrease in the execution time for the multi-threaded optimized backward algorithm compared to the optimized backward algorithm

n	**Football**	**Chess**	**Rugby and handball**	**Lacrosse**	**Basketball, volleyball and tennis**	**Ice hockey and curling**	**Cricket**
5	-	-	-	-	-	-200	-100
6	50	25	25	-	-	11.111	7.894
7	44.444	38.888	38.888	-	-	23.101	16.042
8	59.788	35.233	35.751	14.285	13.333	-	-
9	-	-	-	43.478	43.478	-	-
10	-	-	-	18.75	18.75	-	-

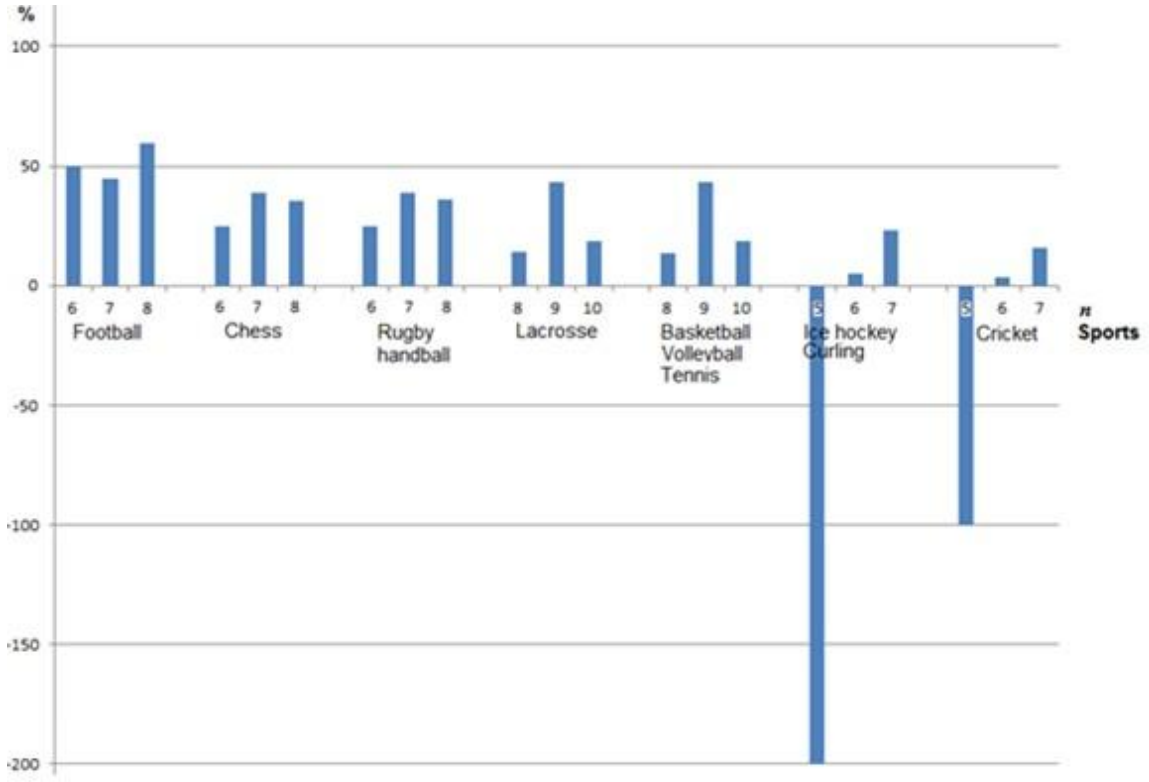

Figure 4.6. The percentage decrease in the execution time for the multi-threaded optimized backward algorithm compared to the optimized backward algorithm

Except for the cases of ice hockey, curling and cricket (i.e., $2q+r=4$) for $n=5$, the multi-threaded optimized backward algorithm has a positive effect on the execution time, where it is reduced from 7.894% to 59.788%. The worst case of the multi-threading appears for $n=5$ in the case of *WXYL* and *WLTNr*, where the algorithm become slower (from 100% to 200%). The difference in the execution times for these cases does not have much effect on the speed of calculation (not more than 5 seconds) compared to other cases where the execution times exceed hundreds of hours.

4.11. Multi-threaded Optimized Forward Algorithm

To determine the optimum number of threads for this case, we perform a set of tests on the multi-threaded optimized forward algorithm with the same principle addressed in Section 4.10. Table 4.38 presents the necessary execution time of the multi-threaded forward algorithm in seconds for each performed test. The same case comes into existence when the value of *thrLim* equals 10 and the performance of the multi-threaded

optimized forward algorithm reaches its maximum for all the presented sports disciplines in Section 1.8.3, even though the used machine contains 4 cores.

Table 4.38. The duration in seconds taken by the multi-threaded optimized forward algorithm in each performed test to determine the optimum number of threads

thrLim	**Football**	**Chess**	**Rugby and handball**	**Lacrosse**	**Basketball, volleyball and tennis**	**Ice hockey And curling**	**Cricket**
1	292	624	537	1592	1574	106	144
2	275	562	516	1457	1420	100	143
3	248	531	475	1262	1306	104	139
4	244	487	434	1187	1172	89	136
5	215	458	413	1052	1068	87	144
6	209	398	392	947	883	78	131
7	185	365	384	762	720	72	129
8	159	323	303	637	636	69	124
9	147	282	269	424	458	59	122
10	143	251	258	377	369	56	122
11	158	262	280	460	398	71	124
12	166	315	296	520	510	97	142
13	184	346	324	637	670	103	164
14	219	387	337	696	791	128	178
15	241	434	359	784	871	134	201
16	272	482	381	828	952	146	206
17	292	511	424	985	1022	152	220
18	325	539	446	1032	1173	171	224
19	337	584	458	1139	1303	197	238
20	359	637	481	1246	1374	213	263

Considering 10 threads, which are the optimum number of threads that can be run in parallel in the machine used for testing, Table 4.39 presents the necessary execution times

to get the results of the multi-threaded optimized forward algorithm for the previously presented sports.

Table 4.39. The execution times of the multi-threaded optimized forward algorithm

n	**Football**	**Chess**	**Rugby and handball**	**Lacrosse**	**Basketball, volleyball and tennis**	**Ice hockey and curling**	**Cricket**
2	1 ms	1 ms	1 ms	1 ms	1 ms	1 ms	1 ms
3	3 ms	3 ms	3 ms	2 ms	2 ms	6 ms	7 ms
4	19 ms	21 ms	21 ms	7 ms	7 ms	48 ms	63 ms
5	118 ms	134 ms	141 ms	19 ms	20 ms	1 s	1 s
6	1 s	1 s	1 s	37 ms	36 ms	56 s	122 s
7	2 m	4 m	4 m	278 ms	283 ms	51 h	-
8	112 h	363 h	367 h	5 s	5 s	-	-
9	-	-	-	6 m	6 m	-	-
10	-	-	-	86 h	87 h	-	-

Table 4.40 shows a comparison between the execution times those are larger or equal 1 second for optimized backward against the multi-threaded optimized backward algorithm, while Figure 4.7 shows a graphical representation of the percentage decrease in the execution time for these cases.

It can be observed from Table 4.40 and Figure 4.7 that the multi-threaded optimized backward algorithm has a positive effect on the execution time in most of the tested cases, where the optimization of the execution time is between 0% and 76.923%.

Table 4.40. The percentage decrease in the execution time of the multi-threaded optimized forward algorithm compared to the optimized forward algorithm

n	**Football**	**Chess**	**Rugby and handball**	**Lacrosse**	**Basketball, volleyball and tennis**	**Ice hockey and curling**	**Cricket**
5	-	-	-	-	-	-	0
6	0	50	50	-	-	47.169	15.277
7	60	60	55.555	-	-	19.047	-
8	52.941	35.523	33.393	50	50	-	-
9	-	-	-	76.923	76.923	-	-
10	-	-	-	56.122	56.716	-	-

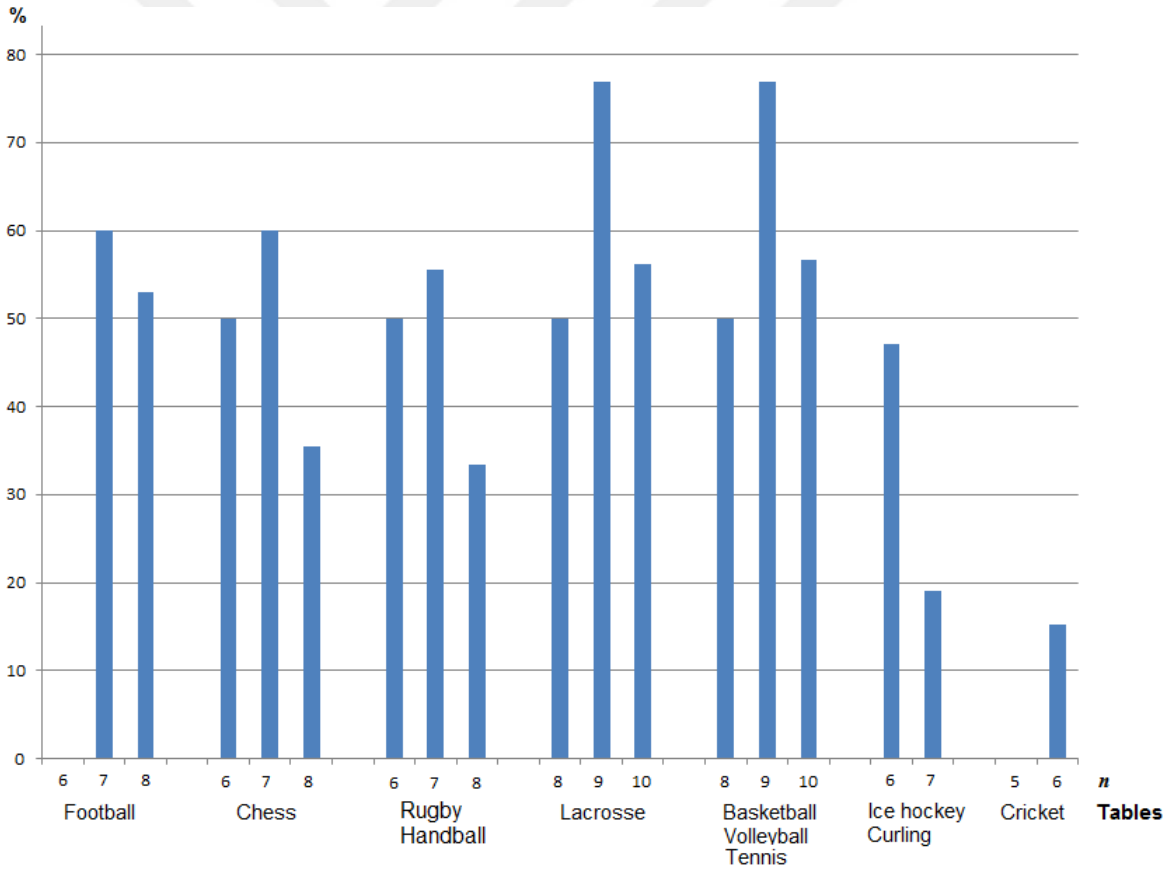

Figure 4.7. The decrease in the execution time for multi-threaded optimized forward algorithm compared to the forward algorithm

5. CONCLUSIONS AND RECOMMENDATIONS

This study seeks to enumerate the possible states of the final table of single round-robin sports tournaments with respect to the number of participants (n). The columns of the tournament tables which are taken into account during the study are the ones related to the possible game results such as W, D and L in the case of football, and W and L in the case of basketball. The determination of the possible states of a round-robin final tournament table can provide a convenient way to ascertain what table data would be adequate for a participant to reach a desired position, which may play a crucial role in the distribution of the participants' revenue.

Backward and forward approaches are developed to enumerate the possible states of a final tournament table. The backward approach is based on generating every possible state and seeks to prove their validity by trying to construct a tournament graph for each generated state. The state is considered to be valid in this approach only if at least one tournament graph can be sought. The forward approach is based on building every possible tournament graph, from which then its corresponding state is derived. The state is taken into account as a valid one in this case only if the points of the participants appear in descending order and the state is not previously generated.

Each participant holds a position in the tournament final table in which it is possible to determine its highest and lowest numbers of points that can be gained by them. To optimize the search space of both approaches, some general constraints are proposed in terms of the standings and points of the participants. A multi-threading based parallelization technique is implemented to enhance the performance of the approaches and to exploit the computation power of the used machine at the highest level, assigning their non-overlapping operations to different threads.

Both time and memory space complexities of the proposed approaches are analyzed and discussed. Tables 5.1 and 5.2 represent the time/space complexities of the algorithms for the presented kind of sports. As seen in the tables, as the number of the possible game results (i.e., $m=2q+r$) increases, the time and space complexities of each algorithm increase too. It can also be seen that the time/space complexity of each version from the backward approach is larger than its counterpart from the forward approach. The values of all the presented time/space complexities in Tables 5.1 and 5.2 do not represent the actual time/space function of the algorithms, but they represent upper bounds for them.

Thus, we cannot judge based on the data of Tables 5.1 and 5.2 that the forward approach is better than the backward, but we can confirm that all the algorithms have exponential time/space complexities.

Table 5.1. The time complexities of the proposed algorithms for the presented kind of sports

Sports / **Algorithms**	Football, handball, rugby and chess	Lacrosse, basketball, volleyball and tennis	Ice hockey, curling and cricket
Backward	$O(n^{n*n+2})$	$O(n^{n*(n-1)+2})$	$O(n^{n*(n+1)+2})$
Forward	$O(n*3^{n*(n-1)+1})$	$O(n*2^{n*(n-1)+1})$	$O(n*4^{n*(n-1)+1})$
Optimized backward	$O(n^{n*n+2})$	$O(n^{n*(n-1)+2})$	$O(n^{n*(n+1)+2})$
Optimized forward	$O(n*3^{n*(n-1)+1})$	$O(n*2^{n*(n-1)+1})$	$O(n*4^{n*(n-1)+1})$
Multi-threaded optimized backward	$O(n^{n*n+2}/thrLim)$	$O(n^{n*(n-1)+2}/thrLim)$	$O(n^{n*(n+1)+2}/thrLim)$
Multi-threaded optimized forward	$O(n*3^{n*(n-1)+1}/thrLim)$	$O(n*2^{n*(n-1)+1}/thrLim)$	$O(n*4^{n*(n-1)+1}/thrLim)$

Table 5.2. The memory space complexities of the proposed algorithms for the presented kind of sports

Sports / **Algorithms**	Football, handball, rugby and chess	Lacrosse, basketball, volleyball and tennis	Ice hockey, curling and cricket
Backward	$O(n^{2*n+1})$	$O(n^{n+1})$	$O(n^{3*n+1})$
Forward	$O(n*3^{[n*(n-1)/2]+1})$	$O(n*2^{[n*(n-1)/2]+1})$	$O(n*4^{[n*(n-1)/2]+1})$
Optimized backward	$O(n^{2*n+1})$	$O(n^{n+1})$	$O(n^{3*n+1})$
Optimized forward	$O(n*3^{[n*(n-1)/2]+1})$	$O(n*2^{[n*(n-1)/2]+1})$	$O(n*4^{[n*(n-1)/2]+1})$
Multi-threaded optimized backward	$O(n^{2*n+1}) + O(thrLim*n^3)$	$O(n^{n+1}) + O(thrLim*n^3)$	$O(n^{3*n+1}) + O(thrLim*n^3)$
Multi-threaded optimized forward	$O(n*3^{[n*(n-1)/2]+1}) + O(thrLim*n^4)$	$O(n*2^{[n*(n-1)/2]+1}) + O(thrLim*n^4)$	$O(n*4^{[n*(n-1)/2]+1}) + O(thrLim*n^4)$

A final state of a tournament table cannot be verified in a polynomial time. A SAT problem is also reduceable into a problem of determining the final states of a tournament table in a polynomial time. Based on these facts and the complexities of the approaches, NP-hard is determined as a class for the problem.

The results of the tests are presented and discussed, where the backward approach exhibits the better performance when the number of the columns related to possible game results equals to 2, while the forward approach is better when the number of the columns is greater than 2. The optimized version of each approach generates better results than the original one from which it is derived, where they are able to calculate the results of some cases that cannot be calculated via the original approaches. Besides, the results of the optimized forward approach are better than those of the optimized backward approach when the number of the participants is less than 8, while the performance of the optimized backward approach is better when the number of participants is more than 8. The multi-threading enhancements of the approaches show positive effects on the execution time in most of the tested cases.

With the aim of comparing the developed algorithms with one another, their computing results of execution time are presented for a certain value of n in Table 5.3. We choose the value of n to be the one that can be tested by all the algorithms (i.e., n=6). The selection of this common value of n allows us to clearly see the superiority of the

algorithms on each other. Accordingly, the forward approach is better than the backward one in the cases of football, rugby, handball, chess, ice hockey, curling and cricket (i.e., the number of the columns is greater than 2). The optimized version of each algorithm takes less time than the original one from which it is derived for all the sports given in Table 5.3. Besides, the optimized forward algorithm requires less execution time than the optimized backward algorithm. The multi-threading enhancement of each algorithm shows a positive effect on the execution time when the number of the columns is greater than 2 except in the case of football, where the execution time remained the same. On the contrary, when the number of the columns equals 2 (i.e., the cases of lacrosse, basketball, volleyball and tennis), multi-threading shows a negative effect on the execution time.

Table 5.3. The results of the proposed algorithms for n=6

Algorithms / **Sports**	Backward	Forward	Optimized backward	Optimized forward	Multi-threaded optimized backward	Multi-threaded optimized forward
Football	3 m	12 s	2 s	1 s	2 s	1 s
Rugby and handball	3 m	13 s	4 s	2 s	3 s	1 s
Chess	3 m	13 s	4 s	2 s	3 s	1 s
Lacrosse	18 ms	16 ms	15 ms	13 ms	33 ms	37 ms
Basketball, volleyball and tennis	17 ms	17 ms	16 ms	13 ms	33 ms	36 ms
Ice hockey and curling	95 h	15 m	18 m	106 s	16 m	56 s
Cricket	92 h	16 m	38 m	144 s	35 m	122 s

The limitation encountered in this study is the generation of the possible states of single round-robin tournament final tables. Therefore, it is recommended that the related results are exploited to generate the possible final states of league tables (i.e., double round-robin tournaments) within the limits of the calculated ones. The determination of the possible states in this circumstance can be done by unifying every possible two states of a single round-robin tournament final table. Also, the results of this work can be

exploited to realize algorithms which determine the possibility that a participant can occupy a specific position on the final table based on its current results.

6. REFERENCES

1. Blanchard, K. and Cheska, A.T., The Anthropology of Sport: An Introduction, First Edition, Bergin & Garvey Publishers, South Hadley, 1985.

2. Hoberman, J.M., Sport and Political Ideology, First Edition, University of Texas Press, Austin, 1984.

3. Mechikoff, R.A., A History and Philosophy of Sport and Physical Education: From Ancient Civilization to the Modern World, Sixth Edition, The McGraw-Hill, New York, 2014.

4. Schrodt, B., Sports of the Byzantine Empire, Journal of Sport History, 8, 3 (1981) 40-59.

5. Kyriazis, N. and Economou, E.M.L., Macroculture, sports and democracy in classical Greece, European Journal of Law and Economics, 40, 3 (2015) 431-455.

6. Scambler, G., Sport and society: History, power and culture, First Edition, McGraw-Hill, New York, 2005.

7. McClelland, J., Body and mind: sport in Europe from the Roman Empire to the Renaissance, First Edition, Routledge, New York, 2007.

8. McComb, D.G., Sports in World History, First Edition, Psychology Press, New York, 2004.

9. Lucas, J.A. and Smith, R.A., Saga of American sport, Lea & Febiger, U.S., 1978.

10. Raney, A.A. and Bryant, J., Handbook of sports and media, Lawrence Erlbaum Associates Publisher, 2006.

11. Eichberg, H., Olympic sport-neocolonization and alternatives, International review for the sociology of sport, 19, 1 (1984) 97-106.

12. Thibault, L. and Harvey J., Fostering interorganizational linkages in the Canadian sport delivery system, Journal of sport management, 11, 1 (1997) 45-68.

13. Thibault, L. and Babiak, K., Organizational changes in Canada's sport system: Toward an athlete-centred approach, European Sport Management Quartely, 5, 2 (2005) 105-132.

14. Samuel, Y.T., An exploratory investigation of sport management students' attraction to sport jobs, Int. J. Sport Management and Marketing, 4, 4 (2008) 323-337.

15. Rosentraub, M.S., Swindell, D., Przybylski, M. and Mullins, D.R., Sport and Downtown Development Strategy If You Build It, Will Jobs Come?, Journal of Urban Affairs, 16, 3 (1994) 221-239.

16. Bjelica, D., Gardasevic, J., Vasiljevic, I. and Popovic, S., Ethical dilemmas of sport advertising, Sport Mont, 13, 3 (2016) 41-43.

17. Jackson, S.J. and Andrews D.L., Sport, culture and advertising: identities, commodities and the politics of representation, Routledge, London, 2005.

18. Collignon, H. and Sultan, N., Winning in the Business of Sports, AT Kearney. https://cache.pressmailing.net/content/443f58fc-6bad-499f-9e9a-f5f66890778b/WinningintheBusinessofSports.pdf 23 March 2020

19. Russell, S., Barrios, D. and Andrews, M., Getting the Ball Rolling: Basis for Assessing the Sports Economy. http://growthlab.cid.harvard.edu/files/growthlab/files/cidwp_321_assessing_sports_economy.pdf 23 March 2020

20. Collignon, H., The sports market. https://www.kearney.com/communications-media-technology/article?/a/the-sports-market 23 March 2020

21. Stead D., Sport and the Media, Sport and society: A student introduction, (2003) 184-200.

22. Gratton, C. and Solberg, H.A., The economics of sports broadcasting, First Edition, Routledge, New York, 2007.

23. Marchand, E., On the comparison between standard and random knockout tournaments, Journal of the Royal Statistical Society, 51, 2 (2002) 169-178.

24. Flores, R., Forrest, D., de Pablo, C. and Tena, J.D., What is a good result in the first leg of a two-legged football match?, European Journal of Operational Research, 247, 2 (2015) 641-647.

25. Harary, F. and Moser, L., The theory of round robin tournaments, The American Mathematical Monthly, 73, 3 (1966) 231-246.

26. Russell, T. and Walsh, T., Manipulating tournaments in cup and round robin competitions, International Conference on Algorithmic Decision Theory, October 2009, Venice, 26-37.

27. Van Cutsem, B., Combinatorial structures and structures for classification, Computational Statistics & Data Analysis, 23, 1 (1996) 169-188.

28. Pemmaraju, S. and Skiena, S., Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Cambridge university press, Cambridge, 2003.

29. Goulden, I.P. and Jackson, D.M., Combinatorial enumeration, John Wiley & sons, New York, 1983.

30. Polya, G., Tarjan, R.E. and Woods, D.R., Notes on introductory combinatorics, Fourth Edition, Springer Science, 2013.

31. Bader, D.A., Agarwal, V., Madduri, K. and Kang, S., High performance combinatorial algorithm design on the Cell Broadband Engine processor, Parallel Computing, 33, 10-11 (2007) 720-740.

32. West, D.B., Introduction to graph theory, Second Edition, Pearson Education, Inc., 2001.

33. Wallis, W.D., A tournament problem, The ANZIAM Journal, 24, 4 (1983) 289-291.

34. Sæbø, O.S. and Hvattum, L.M., Modelling the financial contribution of soccer players to their clubs, Journal of Sports Analytics, 5, 1 (2019) 23-34.

35. Southall, R.M., Nagel, M.S., Amis, J.M. and Southall, C., A method to March madness? Institutional logics and the 2006 National Collegiate Athletic Association Division I men's basketball tournament, Journal of Sport Management, 22, 6 (2008) 677-700.

36. Carlsson, M., Johansson, M. and Larson, J., Scheduling double round-robin tournaments with divisional play using constraint programming, European Journal of Operational Research, 259, 3 (2017) 1180-1190.

37. Pérez-Cáceres, L., Riff, M.C., Solving scheduling tournament problems using a new version of CLONALG, Connection Science, 27, 1 (2015) 5-21.

38. Suksompong, W., Scheduling asynchronous round-robin tournaments, Operations Research Letters, 44, 1 (2016) 96-100.

39. Atan, T. and Hüseyinoğlu, O.P., Simultaneous scheduling of football games and referees using Turkish league data, International Transactions in Operational Research, 24, 3 (2017) 465-484.

40. Westphal, S., Scheduling the German basketball league, Interfaces, 44, 5 (2014) 498-508.

41. Kyngäs, J. and Nurmi, K., Scheduling the Finnish 1st division ice hockey league, Proceedings of the Twenty-Second International FLAIRS Conference, May 2009, Sanibel Island, 195-200.

42. Januario, T., Urrutia, S., Ribeiro, C.C. and De Werra, D., Edge coloring: A natural model for sports scheduling, European Journal of Operational Research, 254, 1 (2016) 1-8.

43. Briskorn, D. and Drexl, A., A branch-and-price algorithm for scheduling sport leagues, Journal of the Operational Research Society, 60, 1 (2009) 84-93.

44. Della Croce, F. and Oliveri, D., Scheduling the Italian football league: An ILP-based approach, Computers & Operations Research, 33, 7 (2006) 1963-1974.

45. Goerigk, M. and Westphal, S., A combined local search and integer programming approach to the traveling tournament problem, Annals of Operations Research, 239, 1 (2016) 343-354.

46. Kern, W., and Paulusma, D., The computational complexity of the elimination problem in generalized sports competitions, Discrete Optimization, 1, 2 (2004) 205-214.

47. Alarcón, F., Durán, G. and Guajardo M., Referee assignment in the Chilean football league using integer programming and patterns, International Transactions in Operational Research, 21, 3 (2014) 415-438.

48. Larson, J., Johansson, M. and Carlsson, M., An integrated constraint programming approach to scheduling sports leagues with divisional and round-robin tournaments, International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems, May 2014, Cork, 144-158.

49. Lim, A., Rodrigues, B. and Zhang, X., A simulated annealing and hill-climbing algorithm for the traveling tournament problem, European Journal of Operational Research, 174, 3 (2006) 1459-1478.

50. Bartsch, T., Drexl, A. and Kröger, S., Scheduling the professional soccer leagues of Austria and Germany, Computers & Operations Research, 33, 7 (2006) 1907-1937.

51. Easton, K., Nemhauser, G. and Trick, M., The traveling tournament problem description and benchmarks, International Conference on Principles and Practice of Constraint Programming, November 2001, Paphos, 580-584.

52. Eggar, M.H., A tournament problem, Discrete Mathematics, 263, 1-3 (2003) 281-288.

53. McSherry, D., Inferring wins, draws and losses from points scored in a sports tournament, Irish Math Soc Bull, 42, (1999) 48-53.

54. Charon, I. and Hudry, O., A branch-and-bound algorithm to solve the linear ordering problem for weighted tournaments, Discrete Applied Mathematics, 154, 15 (2006) 2097-2116.

55. Hemasinha, R., An algorithm to generate tournament score sequences, Mathematical and computer modelling, 37, 4-3 (2003) 377-382.

56. Havel, V., Poznámka o existenci konečných grafů, Časopis pro pěstování matematiky, 80, 4 (1955) 477-480.

57. Hakimi, S.L., On realizability of a set of integers as degrees of the vertices of a linear graph. I, Journal of the Society for Industrial and Applied Mathematics, 10, 3 (1962) 496-506.

58. Duckworth, F.C. and Lewis, A.J., A fair method for resetting the target in interrupted one-day cricket matches, Journal of the Operational Research Society, 49, 3 (1998) 220-227.

59. Nabiyev, V.V. and Pehlivan, H., Tournament scoring problem, Applied mathematics and computation, 199, 1 (2008) 211-222.

60. Pehlivan, H. and Nabiyev, V.V., Score calculation from final tournament tables, Computers & operations research, 36, 3 (2009) 936-950.

61. Damkhi, M. and Pehlivan, H., Determining the results of tournament games using complete graphs generation, Computational and Applied Mathematics, 37, 5 (2018) 6198-6211.

62. Damkhi, M. and Pehlivan, H., Generation and Enumeration of Final Table States of Football Tournaments Using a Blind Adaptive Filtering Algorithm, International Journal of Computing, 7, 2 (2018) 22-28.

63. Damkhi, M. and Pehlivan, H., A Point-Based Algorithm to Generate Final Table States of Football Tournaments, International Journal of Computing, 7, 3 (2018) 38-42.

64. Horen, J. and Riezman, R., Comparing draws for single elimination tournaments, Operations Research, 33, 2 (1985) 249-262.

65. Byl, J., Organizing Successful Tournaments, Fourth Edition, Human Kinetics, Illinois, 2013.

66. France, R.C., Introduction to physical education and sport science, Delmar, Cengage Learning, New York, 2008.

67. Scarf, P.A. and Yusof, M.M., A numerical study of tournament structure and seeding policy for the soccer World Cup Finals, Statistica Neerlandica, 65, 1 (2011) 43-57.

68. Groh, C., Moldovanu, B., and Sela, A. and Sunde, U., Optimal seedings in elimination tournaments, Economic Theory, 49, 1 (2012) 59-80.

69. Marchand, É., On the comparison between standard and random knockout tournaments, Journal of the Royal Statistical Society: Series D (The Statistician), 51, 2 (2002) 169-178.

70. Glenn, W.A., A comparison of the effectiveness of tournaments, Biometrika, 47, 3-4 (1960) 253-262.

71. Edwards, C.T., Double-elimination tournaments: Counting and calculating, The American Statistician, 50, 1 (1996) 27-33.

72. Stanton, I. and Williams, V.V., The structure, efficacy, and manipulation of double-elimination tournaments, Journal of Quantitative Analysis in Sports, 9, 4 (2013) 319-335.

73. Glasson, S., Jeremiejczyk, B. and Clarke, S.R., Simulation of Women's Beach Volleyball Tournaments, ASOR BULLETIN, 20, 2 (2001) 2-7.

74. Cosmo, L.L., The Practice of Water Volleyball as a Leisure and Fine Recreational Water Sport, Health Research, 1, 1 (2017) 1-15.

75. Smith, J.C., Organization of a college baseball tournament, IMA Journal of Management Mathematics, 20, 2 (2009) 213-232.

76. Csató L., Ranking by pairwise comparisons for Swiss-system tournaments, Central European Journal of Operations Research, 21, 4 (2013) 783-803.

77. Van Hecke, T., Monte-Carlo simulation of chess tournament classification systems, APPLIED MATHEMATICS, 3, 4 (2013) 128-131.

78. Csató L., On the ranking of a Swiss system chess team tournament, Annals of Operations Research, 254, 1-2, (2017) 17-36.

79. Arlegi, R. and Dimitrov, D., Fair competition design. http://www.gtcenter.org/Archive/2018/Conf/Dimitrov2839.pdf 24 March 2020

80. Csató, L., Ranking in Swiss system chess team tournaments. http://unipub.lib.uni-corvinus.hu/1830/1/cewp_201501.pdf 24 March 2020.

81. Vuorinen, T., Enhancing Go tournament pairings in Europe, Master Thesis, Tampere University of Technology, Faculty of Computing and Electrical Engineering, Tampere, 2010.

82. Samothrakis, S., Perez, D., Lucas, S. and Rohlfshagen, P., Predicting dominance rankings for score-based games, IEEE Transactions on Computational Intelligence and AI in Games, 8, 1 (2014) 1-12.

83. Hooper, D. and Whyld, K., The Oxford companion to chess, Second Edition, Oxford University Press, London, 1992.

84. https://ruchess.ru/en/championship/detail/2015/nutcracker_2015/ Nutcracker Generation Tournament: Moscow. 24 March 2020.

85. Crowther, M., Kings vs. Queens Tournament 2011. https://theweekinchess.com/chessnews/events/kings-vs.-queens-tournament-2011 24 March 2020.

86. https://trove.nla.gov.au/newspaper/article/182547662/20683097 Finals System Successful: Originator Explains the Reasons. 24 March 2020.

87. Stewart, B., Games Are Not the Same: The Political Economy of Football in Australia, Melbourne University Press, Carlton, 2007.

88. McLellan, C.P., Neuromuscular, Biochemical, Endocrine and Physiological Responses of Elite Rugby League Players to Competitive Match-Play, Doctoral Thesis, Bond University, Faculty of Health Sciences and Medicine, Robina, 2010.

89. Fürnkranz, J., Round robin classification, Journal of Machine Learning Research, 2, (2002) 721-747.

90. Krumer, A. and Lechner, M., First in first win: Evidence on schedule effects in round-robin tournaments in mega-events, European Economic Review, 100, (2017) 412-427.

91. Goossens, D. and Spieksma, F., Scheduling the Belgian soccer league, Interfaces, 39, 2 (2009) 109-118.

92. Zhang, H., Generating college conference basketball schedules by a SAT solver, Proceedings Of The Fifth International Symposium on the Theory and Applications of Satisfiability Testing, February 2002, Cincinnati, 281-291.

93. https://en.chessbase.com/news/2005/fide10.pdf The 2005 World Chess Championship to be held 27 September - 16 October in San Luis, Argentina under the aegis of the Province of San Luis. 24 March 2020.

94. Monks, J. and Husch, J., The impact of seeding, home continent, and hosting on FIFA World Cup results, Journal of Sports Economics, 10, 4 (2009) 391-408.

95. Mahlangu, S., What to expect from biggest ever Africa Cup of Nations tournament. https://www.enca.com/analysis/what-expect-biggest-ever-africa-cup-nations-tournament 24 March 2020.

96. Berker, Y., Tie-breaking in round-robin soccer tournaments and its influence on the autonomy of relative rankings: UEFA vs. FIFA regulations, European Sport Management Quarterly, 14, 2 (2014) 194-210.

97. Johnston, M. and Wright, M., Prior analysis and scheduling of the 2011 Rugby Union ITM Cup in New Zealand, Journal of the Operational Research Society, 65, 8 (2014) 1292-1300.

98. Petersen, C., Pyne, D.B., Portus, M.R., Cordy, J. and Dawson, B., Analysis of performance at the 2007 Cricket World Cup, International Journal of Performance Analysis in Sport, 8, 1 (2008) 1-8.

99. https://zanchess.wordpress.com/2016/03/05/london-chess-club-tournament-1851-a-first-look/ London (1851) - London Chess Club Tournament - a first look. 24 March 2020.

100. https://zanchess.wordpress.com/2016/01/19/london-1862-preliminary-results-and-xtabs/ London (1862) – Preliminary results and xtabs. 24 March 2020.

101. Schneider, J., Schvartzman, A., and Weinberg, S.M., Condorcet-consistent and approximately strategyproof tournament rules, arXiv, (2016).

102. Blair, K., The 2012 Olympic badminton scandal: Match-fixing, code of conduct documents, and women's sport, The International Journal of the History of Sport, 35, 2-3 (2018) 264-276.

103. Cowley, J., The Last Game: Love, Death, and Football, First Edition, Simon and Schuster, London, 2009.

104. https://www.fifa.com/worldcup/archive/usa1994/groups/ 1994 fifa world cup USA™. 24 March 2020.

105. Assad, A.A., Leonhard Euler: A brief appreciation, Networks, 49, 3 (2007) 190-198.

106. May, K.O., The origin of the four-color conjecture, Isis, 56, 3 (1965) 346-348.

107. Appel, K. and Haken, W., Every planar map is four colorable, Bulletin of the American mathematical Society, 82, 5 (1976) 711-712.

108. Riihijarvi, J., Petrova, M. and Mahonen, P., Frequency allocation for WLANs using graph colouring techniques, Second Annual Conference on Wireless On-demand Network Systems and Services, January 2005, St. Moritz, 216-222.

109. Leighton, F.T., A graph coloring algorithm for large scheduling problems, Journal of research of the national bureau of standards, 84, 6 (1979) 489-506.

110. Arkin, E.M. and Silverberg, E.B., Scheduling jobs with fixed start and end times, Discrete Applied Mathematics, 18, 1 (1987) 1-8.

111. Zhang, W., Wang, G. and Wittenburg, L., Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance, Proceedings of AAAI Workshop on Probabilistic Approaches in Search, 2002, 53-59.

112. Albert, J. and Koning, R.H., Statistical thinking in sports, Chapman & Hall/CRC, Boca Raton, 1985.

113. Li, Y., Liang, L., Chen, Y. and Morita, H., Models for measuring and benchmarking Olympics achievements, Omega, 36, 6 (2008) 933-940.

114. Tuominen, M., Stuart, M.J., Aubry, M., Kannus, P. and Parkkari, J., Injuries in men's international ice hockey: a 7-year study of the International Ice Hockey Federation Adult World Championship Tournaments and Olympic Winter Games, Br J Sports Med, 49, 1 (2015) 30-36.

115. Lee, I., The American Aversion to Ties in Sport and Intercollegiate Wrestling's Labyrinthine Tiebreaker Rules, Sw. L. Rev., 47, (2017) 115-135.

116. Berker, Y., Tie-breaking in round-robin soccer tournaments and its influence on the autonomy of relative rankings: UEFA vs. FIFA regulations, European Sport Management Quarterly, 14, 2 (2014) 194-210.

117. Csató, L., Was Zidane honest or well-informed? How UEFA barely avoided a serious scandal. arXiv, (2017).

118. Csató, L., Tournaments with subsequent group stages are incentive incompatible. https://mpra.ub.uni-muenchen.de/83269/1/MPRA_paper_83269.pdf 26 March 2020.

119. Sidhu, H., Kabaddi: A Vigorous Game, Journal of Physical Education, Recreation & Dance, 57, 5 (1986) 75-77.

120. http://globalkabaddileague.co.in/ League Table. 25 March 2020.

121. https://www.world.rugby/pnc/standings Pacific Nations Cup. 26 March 2020.

122. http://www.fiba.basketball/basketballworldcup/2019/groups#|tab=event_round_1 Standings: First Round. 26 March 2020.

123. http://www.fivb.org/EN/volleyball/competitions/WorldCup/2007/Men/Standings/Standings.asp FIVB Men's World Cup 2007. 26 March 2020.

124. http://www.worldlacrosse2014.com/nations/standings 2014 FIL World Lacrosse Championships. 26 March 2020.

125. http://elite.wttstats.pointstreak.com/playoffstandings.html STANDINGS: Elite Ice Hockey League. 26 March 2020.

126. https://howlingpixel.com/i-en/2018%E2%80%9319_Curling_World_Cup 2018–19 Curling World Cup. 26 March 2020.

127. https://www.cricketworldcup.com/standings ICC Cticket World Cup: England & Wales 2019. 26 March 2020.

128. https://www.computerhope.com/history/processor.htm Computer processor history. 26 March 2020.

129. Dennis, J.B., Data flow supercomputers, Computer, 11, (1980) 48-56.

130. Kuck, D.J., Davidson, E.S., Lawrie, D.H. and Sameh, A.H., Parallel supercomputing today and the Cedar approach, Science, 231, 4741 (1986) 967-974.

131. Aliabadi, S.K. and Tezduyar, T.E., Parallel fluid dynamics computations in aerospace applications, International Journal for Numerical Methods in Fluids, 21, 10 (1995) 783-805.

132. Ihshaish, H., Cortés, A. and Senar, M.A., Parallel multi-level genetic ensemble for numerical weather prediction enhancement, Procedia Computer Science, 9, (2012) 276-285.

133. Radeke, C.A., Glasser, B.J. and Khinast, J.G., Large-scale powder mixer simulations using massively parallel GPUarchitectures, Chemical Engineering Science, 65, 24 (2010) 6435-6442.

134. Yaun, G., Carothers, C.D. and Kalyanaraman, S., Large-scale tcp models using optimistic parallel simulation, Seventeenth Workshop on Parallel and Distributed Simulation, June 2003, California, 153-162.

135. Zhao, W., Ma, H. and He, Q., Parallel k-means clustering based on mapreduce, IEEE International Conference on Cloud Computing, December 2009, Beijing, 674-679.

136. Kargupta, H., Hamzaoglu, I. and Stafford, B., Scalable, Distributed Data Mining-An Agent Architecture. The Third International Conference on Knowledge Discovery and Data Mining, August 1997, California, 211-214.

137. Tsoi, K.H., Lee, K.H. and Leong, P.H.W., A massively parallel RC4 key search engine. In Proceedings, 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, April 2002, California, 13-21.

138. Ge, L. and Wang, L., Research of password recovery method for RAR based on parallel random search, International Conference on Applications and Techniques in Information Security, November 2014, Melbourne, 211-218.

139. Yang, Z., Zhu, Y. and Pu, Y., Parallel image processing based on CUDA, 2008 International Conference on Computer Science and Software Engineering, December 2008, Wuhan, Volume III: 198-201.

140. Kwon, K.C., Park, C., Erdenebat, M.U., Jeong, J.S., Choi, J.H., Kim, N., Park, J.H., Lim, Y.T. and Yoo, K.H., High speed image space parallel processing for computer-generated integral imaging system, Optics express, 20, 2 (2012) 732-740.

141. Flynn, M.J., Very high-speed computing systems. Proceedings of the IEEE, 54, 12 (1966) 1901-1909.

142. Catthoor, F. and De Man, H.J., Application-specific architectural methodologies for high-throughput digital signal and image processing, IEEE Transactions on Acoustics, Speech, and Signal Processing, 38, 2 (1990) 339-349.

143. Schneider, B.O. and Rossignac, J., M-Buffer: A flexible MISD architecture for advanced graphics, Computers & graphics, 19, 2 (1995) 239-246.

144. Akpan, O.H., A New Method for Efficient Parallel Solution of Large Linear Systems on a SIMD Processor, PhD Thesis, Louisiana State University, The Department of Computer Science, Louisiana, 1994.

145. Padua, D., Encyclopedia of parallel computing, Springer Science & Business Media, New York, 2011.

146. Moore, K.E., Hill, M.D. and Wood, D.A., Thread-level transactional memory. Technical Report 1524, University of Wisconsin-Madison, Department of Computer Sciences. 2005.

147. Appel, A.W. and Li, K., Virtual memory primitives for user programs, Proceedings of the fourth international conference on Architectural support for programming languages and operating systems. April 1991, California , 96-107.

148. Fiske, S. and Dally, W.J., Thread prioritization: A thread scheduling mechanism for multiple-context parallel processors, Future Generation Computer Systems, 11, 6 (1995) 503-518.

149. Tevanian, A., Black, D., Golub, D., Rashid, R., Cooper, E. and Young, M., Mach threads and the Unix kernel: the battle for control, Proceedings of the USENIX Summer Conference, June 1987, Phoenix, 53-68.

150. DeMartini, C., Iosif, R. and Sisto, R., A deadlock detection tool for concurrent Java programs, Software: Practice and Experience, 29, 7 (1999) 577-603.

151. Krinke, J., Static slicing of threaded programs, Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, July 1998, Montreal, 35-42.

152. Kleiman, S. and Eykholt, J., Interrupts as threads, ACM SIGOPS Operating Systems Review, 29, 2 (1995) 21-26.

153. Polito, G., Ducasse, S., Fabresse, L. and Bouraqadi, N., Virtual smalltalk images: Model and applications, 21th International Smalltalk Conference, September 2013, Annecy, 11-26.

154. Sung, M., Kim, S., Park, S., Chang, N. and Shin, H., Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread, Information processing letters, 84, 4 (2002) 221-225.

155. Marlow, S., Jones, S.P. and Thaller, W., Extending the Haskell foreign function interface with concurrency, Proceedings of the 2004 ACM SIGPLAN workshop on Haskell, September 2004, Utah, 22-32.

156. Stevenson, D.R., Algorithms for translating Ada multitasking, ACM SIGPLAN Notices, 15, 11 (1980) 166-175.

157. Babai, L., Moran, S., Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes, Journal of Computer and System Sciences, 36, 2 (1988) 254-276.

158. Hasan, B.H.F., Saleh, M.S.M., Evaluating the effectiveness of mutation operators on the behavior of genetic algorithms applied to non-deterministic polynomial problems, Informatica, 35, 4 (2011) 513-518.

159. Meek, J., P is a proper subset of NP, arXiv, (2008).

160. Hemaspaandra, L.A., Sigact news complexity theory column 36. ACM SIGACT News, 33, 2 (2002) 34-47.

161. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., Introduction to algorithms, Second Edition, MIT press, London, 2009.

162. Garey, M.R., Johnson, D.S., Computers and intractability, Vol. 174 , W.H. Freeman and Company. New York, 1979.

163. Lavnikevich, N., On the Complexity of Maximum Clique Algorithms: usage of coloring heuristics leads to the $\Omega(2^{0.2n})$ algorithm running time lower bound, arXiv, 2013.

164. Tovey, C.A., A simplified NP-complete satisfiability problem, Discret. Appl. Math., 8, 1 (1984) 85-89.

165. https://www.fifa.com/worldcup/archive/russia2018/groups/ 2018 FIFA World Cup Rusia™. 26 March 2020.

166. https://www.fiba.basketball/basketballworldcup/2014/groups#|tab=round_1 Final Standings. 26 March 2020.

167. https://www.timescolonist.com/olympics/2018-olympic-men-s-hockey-standings-1.23174581 2018 Olympic men's hockey standings. 26 March 2020.

168. https://www.espncricinfo.com/table/series/8781/icc-world-cricket-league-division-five ICC World Cricket League Division Five Table – 2017. 26 March 2020.

169. https://www.fifa.com/worldcup/archive/germany2006/groups/index.html 2006 FIFA World Cup Germany™. 26 March 2020.

170. https://www.fifa.com/worldcup/archive/chile1962/groups/index.html 1962 FIFA World Cup Chile™. 26 March 2020.

171. Suleman, M.A., Qureshi, M.K. and Patt, Y.N., Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs, ACM Sigplan Notices, 43, 3 (2008) 277-286.

172. Pusukuri, K.K., Rajiv, G. and Laxmi, N.B., Thread reinforcer: Dynamically determining number of threads via OS level monitoring, 2011 IEEE International Symposium on Workload Characterization (IISWC), November 2011, Austin, 116-125.

173. Ju, T., Wu, W., Chen, H., Zhu, Z. and Dong, X, Thread count prediction model: Dynamically adjusting threads for heterogeneous many-core systems, 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), December 2015, Melbourne, 456-464.

174. Cutillas-Lozano, L.G., José-Matías, C.L., and Domingo, G., The International Symposium on Distributed Computing and Artificial Intelligence 2012 (DCAI 2012), March 2012, Salamanca, 33-40.

175. Heirman, W., Carlson, T.E., Van Craeynest, K., Hur, I., Jaleel, A. and Eeckhout, L., Automatic SMT threading for OpenMP applications on the Intel Xeon Phi co-processor, Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers, June 2014, Munich, 1-7.

176. Kang, S., Choi, H.J., Kim, C.H., Chung, S.W., Kwon, D. and Na, J.C., Exploration of CPU/GPU co-execution: From the perspective of performance, energy, and temperature, Proceedings of the 2011 ACM Symposium on Research in Applied Computation, November 2011, Florida , 38-43.

177. Dadvar, P. and Skadron, K., Potential thermal security risks, Semiconductor Thermal Measurement and Management IEEE Twenty First Annual IEEE Symposium, March 2005, California, 229-234.

178. Skadron, K., Stan, M.R., Sankaranarayanan, K., Huang, W., Velusamy, S. and Tarjan, D., Temperature-aware microarchitecture: Modeling and implementation, ACM Transactions on Architecture and Code Optimization, 1, 1 (2004) 94-125.

CURRICULM VITAE

Mouslem DAMKHI graduated from Mostefa Ben Boulaïd High School - Batna in 2004. He got his B.Sc.E degree on Computer Science in 2009 form University of Batna - Algeria and M.Sc.E degree on Information Technology in 2012 from Northern University of Malaysia. In September of 2014, he started his Ph.D. education at Karadeniz Technical University - Turkey, Department of Computer Engineering. His research interests include Algorithmics and Operational Research. He has three publications listed below.

1. Damkhi, M. and Pehlivan, H., Determining the results of tournament games using complete graphs generation, Computational and Applied Mathematics, 37, 5 (2018) 6198-6211.

2. Damkhi, M. and Pehlivan, H., Generation and Enumeration of Final Table States of Football Tournaments Using a Blind Adaptive Filtering Algorithm, International Journal of Computing, 7, 2 (2018) 22-28.

3. Damkhi, M. and Pehlivan, H., A Point-Based Algorithm to Generate Final Table States of Football Tournaments, International Journal of Computing, 7, 3 (2018) 38-42.