Queensland University of Technology

CAB203

Discrete Structures

Dr. Matthew McKague

Dinal Atapattu

May 19, 2024

Contents

Chapter 1
Tournament structure

The tournament structure is defined by two constraints: Given the set of games as a set of pairs

$$S = \{(p_1, p_2), (p_3, p_4), \ldots, (p_{n-1}, p_n)\} \tag{1.1}$$

where each pair (p_i, p_j) represents a game between player p_i and player p_j. Using this, we can define the matches of the tournament as

$$P = \{u : (u, v) \in S\} \cup \{v : (u, v) \in S\} \tag{1.2}$$

This can be modeled as the graph $G = (V, E)$ where $V = P$ and $E = S$. However, given the condition that a game between a and b is the same as a game between b and a, S must be symmetrised. Therefore, in order to define E, we must define a symmetrising function $\mathcal{S}$ [6] as follows

$$\mathcal{S}(S) = S \cup \{(v, u) : (u, v) \in S\} \tag{1.3}$$

Using this, we can represent the set of games as

$$E = \mathcal{S}(S) \tag{1.4}$$

As a player is unable to play against themselves, the set of games can be specified to be a **loop free undirected graph**. The constraints of the tournament structure can be defined as follows:

1. For every pair of distinct players, either they play against each other, or there are at least two other players that they both play against
2. All players have the same number of games

For the first constraint, we can define the following

$$\forall u, v \in V : ((u, v) \in E \wedge |N(u) \cap N(v)| \geq 2) \tag{1.5}$$

where $N(v)$ is the neighbour function returning the set of neighbours of a vertex. The above statement states that for all possible pairs of players, there must be an edge between them, or they must have at least two common neighbours.
For the second constraint, we can define the following

$$|\deg(v) : \forall v \in V| = 1 \tag{1.6}$$

where $\deg(v)$ is the degree of vertex v. The above statement states that all players must have the same number of games.
Combining the two constraints, we can define the tournament structure as

$$\forall u, v \in V : ((u, v) \in E \wedge |N(u) \cap N(v)| \geq 2) \wedge |\deg(v) : \forall v \in V| = 1 \tag{1.7}$$

The python implementation utilises `set` data structures to represent the vertices and edges of the graph. A function `symmetrise(E)` implements the symmetrisation function $\mathcal{S}$, in order to symmetrise the set of edges `E`.
`V` is calculated by taking every player from every edge in `E` and uses set comprehension to remove duplicates.
These are then used in conjunction with a ternary operator which implements a conditional that returns false if the conditions in equation 1.5 are not met. Cardinality of sets is calculated using the `len()` function.

Chapter 2
Assign referees

We begin by modelling an instance of the problem given the CSV file. Each row contains a referee and their conflicts of interest in the following format

$$R = [r_n, c_{1,1}, c_{1,2}, \ldots, c_{1,n}] \forall R \in F \tag{2.1}$$

Where R is any row excluding the header, F is the CSV file, r_n is a referees and $c_{1,2} \ldots c_{m,n}$ are the conflicts of interest.
This data in it's current state is not useful for the task at hand. We must first parse this data in a more useful format.

Given that P is the set of players, and

$$R = \{r : [r, c_1, \ldots, c_n] \in F\} \tag{2.2}$$

By defining the the relationship between the referees and their conflicts of interest as the function c, we can define the set of conflicts of interest as

$$c = R \to \mathcal{P}(p) \tag{2.3}$$
$$c(r) = \{c \in \mathcal{P}(p) : c \in R\} \tag{2.4}$$

Where $\mathcal{P}(p)$ is the power set of P and $c(r)$ is the set of conflicts of interest for referee r. $([r, c_1, \ldots, c_n])$
Note that r is included in the conflicts of interest as a referee cannot referee their own game.
Given this, we can now formulate a solution to the problem.
The task require assigning a referee to each game in the /tournament. This can be modelled as a **Bipartite Graph** [3] [5] with vertices $V = G \cup R$, where there is a bipartition of the vertices into two sets G and R where G is the set of games and R is the set of referees (Figure 2.1).

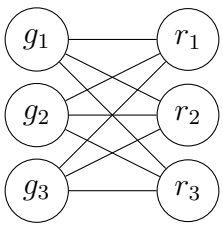

Figure 2.1: $K_{3,3}$ Bipartite graph of games and referees

The assignment of a referee $r \in R$ to a game $g \in G$ will be represented as an edge between a game and a referee (r, g), with a valid assignment containing $|G|$ edges (one referee per game).

$$E = (\{(g, r) : g \in G, g \cap c(r) = \emptyset\}) \tag{2.5}$$

Given the objective is to assign a referee to each game and maximising the number of valid assignments. This can be further defined in terms of **Maximum Matching Problem** [8]. Given the maximum matching of the graph $G = (V, E)$ as M, we can test the validity of this by comparing the cardinality of M to

the cardinality of G. If the cardinality of M is equal to the cardinality of G, then the matching is valid.

$$|M| = |G| \implies \forall g \in G : \exists r \in R : gRr \tag{2.6}$$

Otherwise, the matching is invalid as there are games without referees.

$$A = \{(g, r) : g \in G, r \in R, g \cap c(r) = \emptyset\} \tag{2.7}$$

A will be a bijective function that maps G (games) to R (referees) such that the referee assigned to a game is not in the conflicts of interest of the referee and every game has only one referee ($\forall g \in G : \exists r \in R : gRr$) [2]
This was implemented in Python by first defining the vertices `V` using set comprehension analogues to equation 2.1. The edges `E` were formed using set comprehension analogues to equation 2.7. The function c defined in equation 2.4 was implemented using dictionary indexation operators on the conflicts of interest and the native Python operator `in` was used to confirm the presence of a referee in the conflicts of interest. The maximum matching of the graph was found using the `digraphs.maxMatching` function, which returned the values in a dictionary format. The validity of the matching was tested using a conditional statement that compared the cardinality of the matching to the cardinality of the games.

Chapter 3
Game groups

Now that games have been assigned referees, we can now group games that can be played together in the same time slot. However, we must first define the constraints of the problem.

1. People are not double-booked: each person is involved in at most one game in any game group. either as a player or as a referee.
2. Games are played once. Each game is in exactly one game group

This problem can easily be translated to a **Minimum Colouring Problem** [7], where the vertices are the games and referees, and the edges are the conflicts of interest, and the objective is to find the minimum number of colours required to colour the vertices such that no two adjacent vertices share the same colour, and to group the vertices by colour.
For $((p_1, p_2), r) \in A$, where A is the set assignedReferees. This is also the set of vertices in the graph.

$$V = A \tag{3.1}$$

In order to define the edges, we must first define a function that will give the set of participants/members per game

$$\mathcal{M}((p_1, p_2), r) = \{p_1, p_2, r\} \tag{3.2}$$

Using this we can define the set of edges as

$$E = \left\{(u, v) \in V^2 : (\mathcal{M}(u) \cap \mathcal{M}(v) \neq \emptyset) \wedge u \neq v\right\} \tag{3.3}$$

This gives us the graph $G = (V, E)$ where V is the set of games and referees and E is the set of edges. We can now obtain a minimum colouring $[c_1, c_2, \ldots, c_n]$ of the edges of the graph, where c_i is the colour of the i^{th} edge. This will give us the game groups.

For the Python implementation, we form the vertices `V` using set comprehension analogues to equation 3.1. The edges `E` are formed using set comprehension analgoues to equation 3.3. The function $\mathcal{S}$ defined in equation 3.2 was implemented using the native `set` function and indexation operators to extract the participants of a game. The colouring of the graph was found using the `graphs.minColouring` function, which returned the values in a dictionary format. Which was then mapped to an array of groups of games using a dictionary comprehension where the colour values were equal.

Chapter 4
Game schedule

Given the set of assigned referees per game A and the set of game groups G, we are required to schedule each game group in a time slot such that

- People are not double-booked: each person is involved in at most one game in any game group. Either as a player or referee
- Games are played once: each game is in exactly one game group

Furthermore, the committe requires the smallest number of game groups possible.
Thie problem can easily be translated into a topological sorting problem, Where the graph $H = (V, E)$ is a **Directed Acyclic Graph (DAG)** [7] [4], where V is the set of game groups and E is the set of directed edges from a game group that must be played before another game group.

$$V = G \tag{4.1}$$

In order to define the edges, we must first define a function that will give the assigned referee per game

$$\mathcal{R}(g) = \{r : (g, r) \in A\} \tag{4.2}$$

This function will give the referee assigned to a game. Using this we can define the set of edges as

$$E = \left\{(u, v) \in V^2 : (\forall a \in u, \forall b \in v \exists a, b : \mathcal{M}(a) \cap \mathcal{R}(b) \neq \emptyset) \wedge u \neq v\right\} \tag{4.3}$$

This checks if there is a referee that is assigned to a game in u that is also a player in a game in v by checking the intersection of each element in u with the referees of each element in v.
Where u precedes v in the schedule. Note that an extra condition is added to ensure that u is not equal to v to avoid cycles as this would be antithetical to the nature of a DAG[4].
This gives us the directed graph $H = (V, E)$ where V is the set of game groups and E is the set of edges running from a game group that must be played before another game group.
We can now obtain a topological sorting of the graph H to obtain the schedule of the game groups.
By the properties of topological ordering, u will always precede v in the schedule if u is a parent of v in the graph H.
This was implemented in Python by first defining the vertices `V` using set comprehension analogues to equation 4.1, each element was typecast into a `frozenset` in order to type errors with conventional sets due to their mutability. The edges `E` were formed using set comprehension analogues to equation 4.3. The function $\mathcal{R}$ defined in equation 4.2 was implemented using dictionary indexation operators on assignedReferees and the native Python operator `in` was used to confirm the presence of a referee. The topological ordering of the graph was found using the `digraphs.topOrdering(V,E)`.

Chapter 5
Tournament winners

Given the outcome of all the games in a tournament, it was required that the score be calculated and the winners be determined.
Each player's maximum score is related to the primary and secondary wins that they have, where a primary win results in p points, a secondary win resulting in s points. However, each secondary and primary win pair's points are capped at c.
Given these constraints, we can translate this scenario to a **Maximum Flow Problem** where the graph $H = (V, E)$ is a directed graph where V is the set of players and E is the set of directed edges from a winner to a loser The set of directed edges E has been provided and is in the format $(w, l) \in E$ where w is the winner and l is the loser.
Using this, we can define the vertices as the union of winners and losers

$$V = \{w : (w, l) \in E\} \cup \{l : (w, l) \in E\} \tag{5.1}$$

We must also consider the drain and source vertices for each player, this can be accounted for by the super-sink and super-source vertices which are vertices that are mapped to themselves s and t respectively [1]

$$E = (w, l) \in E \cup \{(s, s) \forall s \in V\} \cup \{(t, t) \forall t \in V\} \tag{5.2}$$

Note that the capacities of the super source and super sink vertices are infinite.
The capacity of the edges are defined by the variable c which is given.
As different players have different relationships with each other, each graph will have different capacities for each edge.
The flow of an edge is given by either the variable p or s depending on the type of win. Where p is the primary win and s is the secondary win.
Primary wins are defined as the immediate out neighbours of a player, and secondary wins are defined as the out neighbours of the primary wins.

(5.3)

Bibliography

[1] Aleks Ignjatović. *Algorithms: COMP3121/3821/9101/9801*.

[2] Adrianne Jenner. "MXB102 Advanced Mathematical Reasoning". In: *Relations, Functions, and the Big-O Notation*. 2022. Chap. 27, p. 27.

[3] Matthew McKague. *CAB203 Lecture 7*. `https://canvas.qut.edu.au/courses/16665/files/3497116/download?download_frd=1`. Accessed: 2024. 2024.

[4] Matthew McKague. *CAB203 Lecture 8*. `https://canvas.qut.edu.au/courses/16665/files/3497113/download?download_frd=1`. Accessed: 2024. 2024.

[5] Matthew McKague. *CAB203 Lecture 9*. `https://canvas.qut.edu.au/courses/16665/files/3497110/download?download_frd=1`. Accessed: 2024. 2024.

[6] Matthew McKague. *CAB203 Tutorial 8*. `https://canvas.qut.edu.au/courses/16665/files/3829888/download?download_frd=1`. Accessed: 2024. 2024.

[7] Gordon J. Pace. *Mathematics of Discrete Structures for Computer Science*. eng. 1st ed. 2012. Berlin: Springer, 2012. ISBN: 3-642-29840-0.

[8] Vitaly I. Voloshin. *Introduction to Graph Theory*. eng. 1st ed. New York: Nova Science Publishers, 2009. ISBN: 1-61470-113-X.