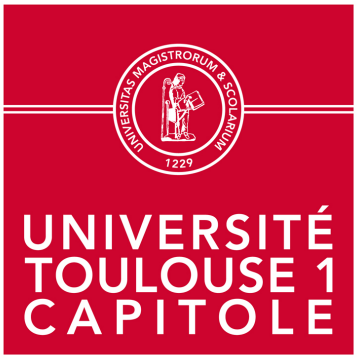

COMPUTATIONAL INTELLIGENCE AI PROJECT: REASONING

Yi fan HAO
yyyifan.hao@gmail.com

Gonzalo
jose.dominguez@ut-capitole.fr

Aygun Ismayilova
aygun.ismayilova@ut-capitole.fr

Benjamin
benjamin2.bordier@ut-capitole.fr

Abstract

This paper addresses at how to schedule the Six-Nations rugby tournament efficiently which uses a double round-robin format with specific rules and specific constraints. We study two ways to solve this problem; as a regular search problem and as a constraint satisfaction problem (CSP). DFS, BFS and CSP solvers are used to determine which method works best in different situations. When we deployed these different approaches, we found big differences in how fast they are able to process the problem. Our results show that the CSP method, especially when we use techniques like forward checking and minimum conflicts, is quicker and better at handling the tournament's complex schedule. Thus, this research helps us understand how to operate to solve similar search problems in others areas.

Keywords — computational intelligence, constraint satisfaction problem, search algorithms, depth-first search, breadth-first search, algorithm performance comparison, tournament scheduling

1 Introduction

The tournament organizers have proposed a new format that adjusts the traditional structure to enhance competitive balance and viewer engagement. This new format necessitates a complete new reevaluation of the scheduling process to accommodate additional constraints such as location, matches order, and fairness in play.

The complexity of sports scheduling provides a rich domain for the application of various computational intelligence approaches. These methods are very interesting for this problem as they are able to determine possible solutions within acceptable time frames. This paper explores the applicability of depth-first search (DFS), breadth-first search (BFS), and Constraint Satisfaction Problem (CSP) to the task of scheduling the revised Six-Nations tournament. Each method's effectiveness is measured based on execution time and the ability to satisfy the constraints, providing insights into their suitability for similar problems in sports and other fields.

This study contributes to the theoretical frameworks of computational problem solving and also provides practical insights that can be directly applied by tournament organizers to optimize scheduling under new and evolving conditions.

2 Problem Description

In the context of finding the best generalizable problem solving approach to the specific problem, we were assigned the challenge with the description detailed below. We were expected to find a solution to the problem of scheduling matches for a new version of the 6-nations rugby tournament, if exists. The follow-

ing rugby tournament teams will face each other in a round-robin tournament: FRANCE, ITALY, SCOTLAND, IRELAND, WALES and ENGLAND. The organisers wanted us to consider the following new rules:

1. Each team faces another team only twice during the whole tournament that consists of 10 rounds and satisfies the following mirroring scheme: (1, 6)(2, 7)(3, 8)(4, 9)(5, 10) Teams that played together in 1,2,3,4 and 5th rounds should face each other again in 6,7,8,9,10 respectively, however with the change of home country. So it is important to make sure that every team plays no more or less, but 1 time at home with the same team, in other words, plays in every country only once.
2. Teams are not supposed to play at consecutive rounds away or at home.
3. Teams are not supposed to play against ITALY or FRANCE away twice in consecutive rounds
4. On the last round ENGLAND and FRANCE should play together

We were expected to provide a good solution and solving algorithm respecting all these constraints, also considering the short execution time. If no solution exists, we were encouraged to provide the solution with the most close to the given constraints. To start with, we modeled and solved the problem of 4 teams, checked all the constraints, and compared the execution time of each of the search algorithms and constraint satisfaction problem solver.

3 Problem Modelling

Before searching for a solution we must model the problem. First, we need to use abstraction, i.e. selecting only relevant actions and states. Because if we don't we will not be able to solve a real life complex problem. Second, we need to do the analysis of the environment. There are 3 questions we need to answer:

1. Is there a current state?
2. Is there a finite of actions at each state?
3. Is there a deterministic effect of actions? Are they feasible?

When we talk about the deterministic effects of actions, we're talking about results that are completely predictable given the acts that were taken. This means that given a specific set of inputs or actions, the resulting outcomes could be precisely determined. We must determine the following:

- Set of states including initial state
- Actions for each state
- Transition model explaining resulting state for each state and action (Other states excluding initial state)
- Goal-test to check the goal states
- Cost function for optimization

3.1 Breadth-First Search and Depth-First Search with Graph implementation

For BFS and DFS Search with graph implementation we identified states as a round schedule that is composed of the list of matches. The list size here depends on the number of matches played per round. Goal-test is the function isEnd() that defines if the state/node founded is composed of complete rounds considering all the constraints. In this problem, we skip the optimization part, as for now, in the scope of this project we were more focused on comparing different techniques. So we stop the search when we find the one solution that satisfies all the rules and considered complete. Let look at all steps in details:

- **Set of states including initial state:** In the graph approach, we defined nodes as a list of three tuples, where an initial state is an empty list, i.e. no round is scheduled. 3 tuples define 3 matches playing in a round. The match is composed of 2 teams, a team in position 1 is the team playing home, a team in position 2 is the team playing away.
- **Actions for each state:** Each node/state produces actions that are defining at the end the next node/state. In this case, action is a round that is chosen from all possible round combinations.
- **Transition model explaining resulting state for each state and action**:

  Current state: [ ] <– empty state

  Action: [('EN', 'FR'), ('WA', 'IT'), ('IR', 'SC')]

  Resulting state:
  [('EN', 'FR'), ('WA', 'IT'), ('IR', 'SC')]

  Action: [('WA', 'FR'), ('IR', 'IT'), ('EN', 'SC')]

  Resulting state:
  [('EN', 'FR'), ('WA', 'IT'), ('IR', 'SC')]
  [('WA', 'FR'), ('IR', 'IT'), ('EN', 'SC')]

  Action: [('IR', 'FR'), ('SC', 'IT'), ('EN', 'WA')]
  Resulting state:

[('EN', 'FR'), ('WA', 'IT'), ('IR', 'SC')]
[('WA', 'FR'), ('IR', 'IT'), ('EN', 'SC')]
[('IR', 'FR'), ('SC', 'IT'), ('EN', 'WA')]

- **Goal-test to check the goal states**: A goal state is found if all 5 rounds(in the case of 6 teams) are scheduled. We do not consider 10 rounds, as we will use a mirroring scheme to define another 5 rounds based on the previous 5 rounds.

3.2 DFS with hypothetical states

For Deepth-First Search (DFS) with hypothetical states, we identify the state as a class containing the ordered matches, the unordered matches, the subclasses, the parent class and the list of matches already has been selected once. The hierarchy of the class depends on the length of the list of ordered matches. The goal test is that the unordered matches are empty, which represents that all matches have been completed in order. Each advance (arranging the races) needs to follow all the constraints mentioned above.

3.3 CSP

In this approach, we use a **variable x [i, j]** of type integer, which defines a matrix of size n x m. In this case, n = m, representing the number of teams in the tournament.

The idea is to assign each value in the matrix a number between 0 and 5 which represents the number of the round. For example, let consider x[0,1]: if i = 0 is France and j = 1 means Italy, then for x[0, 1] = 3 means the France and Italy will play in round 5 in France. If x[i, j] = 0, that means the game is not played. This feature will be explained more in detail in the solving section.

- **Set of states including initial state:** In CSP, a state is an assignment of a value to some of the variables (not necessarily to all variables). In this case, as a **initial state**, there exists a matrix of size n x m (n = m = n^o teams)‘where every value is 0. The **set of states** will be all the assignments of the cells in the matrix while solving the problem. Since we have 36 cells in the matrix for 6 teams, and for every cell we have 6 possible values [0,5] then that results in a total of 216 states. In later sections, it will be explained why round value goes up to 5 and not 10. Due to this fact, we have lower number of states.
- **Actions for each state:** assign(x[i, j], q), q in [0,5].
- **Transition model explaining resulting state for each state and action** (Other states excluding initial state):

  Current state: Action:

  | 0 0 0 0 0 0 | assign(x[0,1], 1)

  | 0 0 0 0 0 0 |

  | 0 0 0 0 0 0 |

  | 0 0 0 0 0 0 |

  | 0 0 0 0 0 0 |

  | 0 0 0 0 0 0 |

  Resulting state:

  | 0 1 0 0 0 0 |

  | 0 0 0 0 0 0 |

  | 0 0 0 0 0 0 |

  | 0 0 0 0 0 0 |

  | 0 0 0 0 0 0 |

  | 0 0 0 0 0 0 |

- **Goal-test to check the goal states**: A goal state is found if the matrix have been assigned completely.

  assign(x[i, j], q) == True, q in [0,5]

4 Problem Solving Methods

4.1 Depth-First Search

Depth-First Search (DFS) is a fundamental algorithm widely used in computer science, artificial intelligence, and operations research for navigating and searching in graph and tree structures.DFS works by exploring as far as possible along each branch before backtracking. This property makes it an efficient algorithm for searching in spaces where not all solutions are feasible, as it allows for quick penetration into the structure before needing to examine other alternatives.DFS first explores down to the depths until it is not possible to go any deeper, then it returns to the previous level and then moves to the neighboring unvisited node, and then explores down to another branch. This process continues until the correct node is found, or all nodes have been explored.

DFS It is especially suitable for scenarios where the solution is deep and not decentralized. The permutation problem explored in this paper for six teams and thirty games fits well with the use of DFS.

4.1.1 DFS with Graph Implementation

The DFS algorithm begins by initializing an initial state. The initial state is pushed onto the stack to begin exploration. While the stack is not empty, the algorithm repetitively pops a state from the stack for examination. If the current state satisfies the goal condition (i.e., five rounds scheduled, adhering to constraints), the algorithm terminates, returning the current state as the solution. Otherwise, the algorithm generates feasible actions (i.e., combinations of potential match schedules) from the current state. For each action, it applies the action to generate a new state, combines this state into the graph, and adds it to the stack for further exploration. The process iterates until a valid tournament schedule is set.

4.1.2 DFS: with hypothetical states

As mentioned above, in this approach our state is a class containing the parent and subclass, the list of ordered and unordered matches, and the list of matches has been selected. In this case, the hierarchy of the state depends entirely on and is dependent on the number of matches ordered, i.e., the length of the list of ordered matches: what we ultimately want as a result is the order of matches obtained by combining 30 matches, so whether or not we actually find and represent all the possible states in the code, they all exist. All we need to do is start with the Initial state, where the list of ordered matches is empty, and randomly select the matches from the list of unordered matches, and then go deeper and deeper in the hierarchy.

Random extraction here replaces the step of completely finding all possible states, ensuring that we can explore all possible nodes, and the selected matches ensure that we don't repeat the last selection after backtracking causing the code to fall into an infinite loop.

4.2 Breadth-First Search

Breadth-First Search (BFS) is the one of the useful search techniques. It is an algorithm for searching a tree or graph data structure for a node that meets a set of criteria. It starts at the root node and traverses through all nodes level by level until it finds a goal state, i.e. solution. In our BFS algorithmic approach, we have traversed level by level the graph that is structured in a way that is validating all possible actions for each state. In the problems that may have cycles it is important to store visited nodes to eliminate the high possibility of the non-stop loop during traversal. This algorithm uses the concept of First In First Out (FIFO) that is explained by its working principle. All children nodes are stored in a queue are ready to be expanded in the direct order of how they were stored.

BFS have its own drawbacks, and it is important to consider some cases before applying it on the problem. However, as this project is done for educational purposes, we have implemented this algorithm even if we consider it to be not the best approach dealing with this type of scheduling problems. The reason for this, is that BFS is slower than DFS and more suitable for searching nodes closer to the initial state (root node in this case). For this problem we had to find the complete graph satisfying all the provided constraints, and as we modeled the problem so that states/nodes are different round schedules, we know that the complete and correct solution will be in the last round.

4.3 Constraint Satisfaction Problem

A **constraint satisfaction problem** consists of several mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints and variables, which is solved by constraint satisfaction methods. It is a subject of research in both AI and operations research, they often exhibit high complexity.

In the six nations problem with 6 teams and 10 rounds results in 30 matches, 3 for each round. At the first glance, this **problem domain** will be factorial of 30 (30!) which is considered high complexity. Nevertheless the model, limitations and constraints makes this problem much more easy to work with. To compute a solution we use **Python** programming language and its module called constraint.

The **python-constraint** module offers efficient solvers for Constraint Satisfaction Problems (CSPs) over finite domains in an accessible Python package. CSP is class of problems which may be represented in terms of variables (a, b, ...), domains (a in [1, 2, 3], ...), and constraints (a $<$ b, ...).

Our model consists of a variable x[i, j] which represents a matrix, as it has been explained in depth in [3.3] section. As it can be seen in that section, the value of each cell just goes up to 5 which, in theory, it will compute only the first 5 rounds. But this way to organize the tournament **reduces the complexity by half**. From the problem definition, we know that matches will be repeated after 5 rounds but in different country.

For example: if France and England play in France in the 1º round then in 6º round they will play again but in England. Because of this situation and the fact that we also know the first 5 rounds then the last rounds can be easily computed. In order to

obtain a solution, the following constraints will be added to our problem:

1. **A team can not play against itself:**
   problem.addConstraint(lambda value, i=i, j=j: value == 0, ((i, j),)).
   It sets the main diagonal to 0.

2. **A team only plays once in each round:**
   problem.addConstraint(AllDifferentConstraint(), [(i, j) for j in range(len(matrix[i]))])
   problem.addConstraint(AllDifferentConstraint(), [(i, j) for i in range(len(matrix[j]))])

3. **England-France has to be the last game(round 10):**
   if i== 0 and j==5:
   problem.addConstraint(lambda value, i=i, j=j: value == 5, ((i, j),))

4. **If a team plays in Italy then it will not play in France in next round and vice versa:**
   if i == 1 and j in [2, 3, 4, 5]:
   problem.addConstraint(lambda val1, val2: abs(val1 - val2) > 1, [(0, j), (i, j)])
   Since we know that i=0 is France and i = 1 is Italy, then we have to compare and assure a difference greater than 1 in columns [2,3,4,5] which are all the teams without France and Italy.

5. **Mirror matrix constraint:**
   problem.addConstraint(lambda val1, val2: val1 ==val2, [(i, j), (j, i)])
   Since arrows and columns point to the same countries (row 0 is France as well as column 0) then we have to assure that x[i, j] = x[j, i] making the matrix symmetrical with the axis in the main diagonal.

5 Comparison of results

Table 1: AlgorithmResults (ms)

Config	Algorithm Results			
	DFS1	DFS2	BFS	CSP
4 no const	1.93	6.95	37.78	0.18
4 with const	9.19	9.83	25.13	0.53
6 no const	136.82	9.04		0.53
6 with const	74.34	60.16		1.81

From these results we can deduce that the shortest execution times are obtained when computing the competition for only 4 teams and without constraints. And that **CSP approach is the fastest** by a great difference with respect to others. For example, when computing the tournament of 6 teams with the constraints, CSP gets a solution 33 times faster than second faster approach which is DFS2.

6 Conclusion

Our study is based on the challenge of organising the Six-Nations rugby tournament using depth-first search (DFS), breadth-first search (BFS), and constraint satisfaction problem (CSP) solvers. We found that **CSP**, especially with techniques like forward checking and minimum conflicts, outperformed other methods in terms of efficiency and constraint satisfaction. This research provides interesting insights of AI for optimizing scheduling in sports and other domains.

Through this project, we learnt how to implement computational intelligence algorithms to solve real life problems. In future, we would like to explore machine learning approaches and advanced optimization techniques to improve the efficiency and accuracy of such solutions. Russell et al. [2010]

References

S. J. Russell, P. Norvig, and E. Davis. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 3 edition, 2010.