Project Summary

The NHL season requires a rigorous amount of planning to ensure the optimal season schedule has been created. Our project creates a working NHL season schedule, on a week-by-week basis, taking into account the various rules and restrictions associated with the league. Additionally, our project will focus on the altered rules of the most recent 2020-2021, COVID influenced season. Based on the NHL schedule of this season, our project follows a division of 8 teams, each playing each other a total of 8 times.



Our model creates an entire 16-week season schedule by crafting a valid week, based on the restrictions defined by our constraints. It ensures the validity of the week, then adds it to the schedule, and repeats for 16 weeks. The model keeps track of all the possible games that need to be played, eliminating them once they have been played, to ensure all games are played during the season. Thus, creating a valid possibility for the season’s schedule.

Propositions

**Game(Team1, Team2):** Each game played must contain two teams, a home team, and an opposing away team. date: can be day 1-7. Any one of the days in the week.

Team1: Can be any one of the 8 teams in the division of play.

Team2: Can be anyone of the 8 teams, and not Team1, the other team in the game.

\**SideNote*: We recognize that Team2 can simply be written as ¬ Team1 on a single game basis. However, when trying to schedule the games for the entire week, we need to be able to keep track of more factors than simply the opposing team not being Team1. Hence the reason for Team2, so we can keep track of what team is a valid opposing team for Team1, based on the additional factors of our constraints.

**Day (game1 ,game2, game3, dayNum):** Each day holds three possible game slots, as well as a number to differentiate the days.

game1: An empty availability slot, that can be filled with a game between two teams.

game2: A different empty availability slot, that can be filled with a game between two teams.

game3: A different empty availability slot, that can be filled with a game between two teams.

dayNum: a number from 1-7, representing the day of the week that the day is on.

**Week(days, week\_num):**  Each week holds all the days of the week, as well as the number of the week in the schedule.

days: holds day 1, day 2, day 3, through to day 7, with each day holding a day proposition, containing the games and day number.

week\_num: a number from 1-16, to identify which week of the schedule it is.

Constraints

\**Note*: All of our constraints are based on a week rather than the entire schedule. Our model is created, by a week-by-week basis, in which a week is only added to the schedule, if it satisfies these constraints. Thus, the constraints are week focused.

**D = Day**  **T = Team** **G = Game**

* **Each day must contain at least one game being played**

D(G1,G2, G3, 1) ∧D(G1, G2, G3,2)∧ D(G1, G2, G3, 3) … ∧ D(G1, G2, G3, 7) must be true

* **Each team cannot play a game three days in a row**

D(G(Tx , Ty), G2, G3, x) ∧ D(G(Tx,Tz), G2, G3, x+1) → D(G(¬ Tx , Ty ), G2, G3, x+2)∧ D(G(¬ Tx , Ty ), G2, G3, x-1)

* **Each team must play at most 4 games a week**

D(G(Tx, Ty), G2, G3, x) ∧ D(G(Tx, Ty),G2,G3, y) ∧ D(G(Tx, Ty), G2,G3,z) ∧ D(G(Tx, Ty), G2,G3,q) → ¬ D((Tx, Ty), G2, G3, s)

* **Each team must play at most one game a day**

D(G1(Tx,Ty),G2, G3, x) → ¬ D(G1,G( Tx, Ty), G3 ),x) ∧ ¬ D(G1,G2, G( Tx, Ty),x)

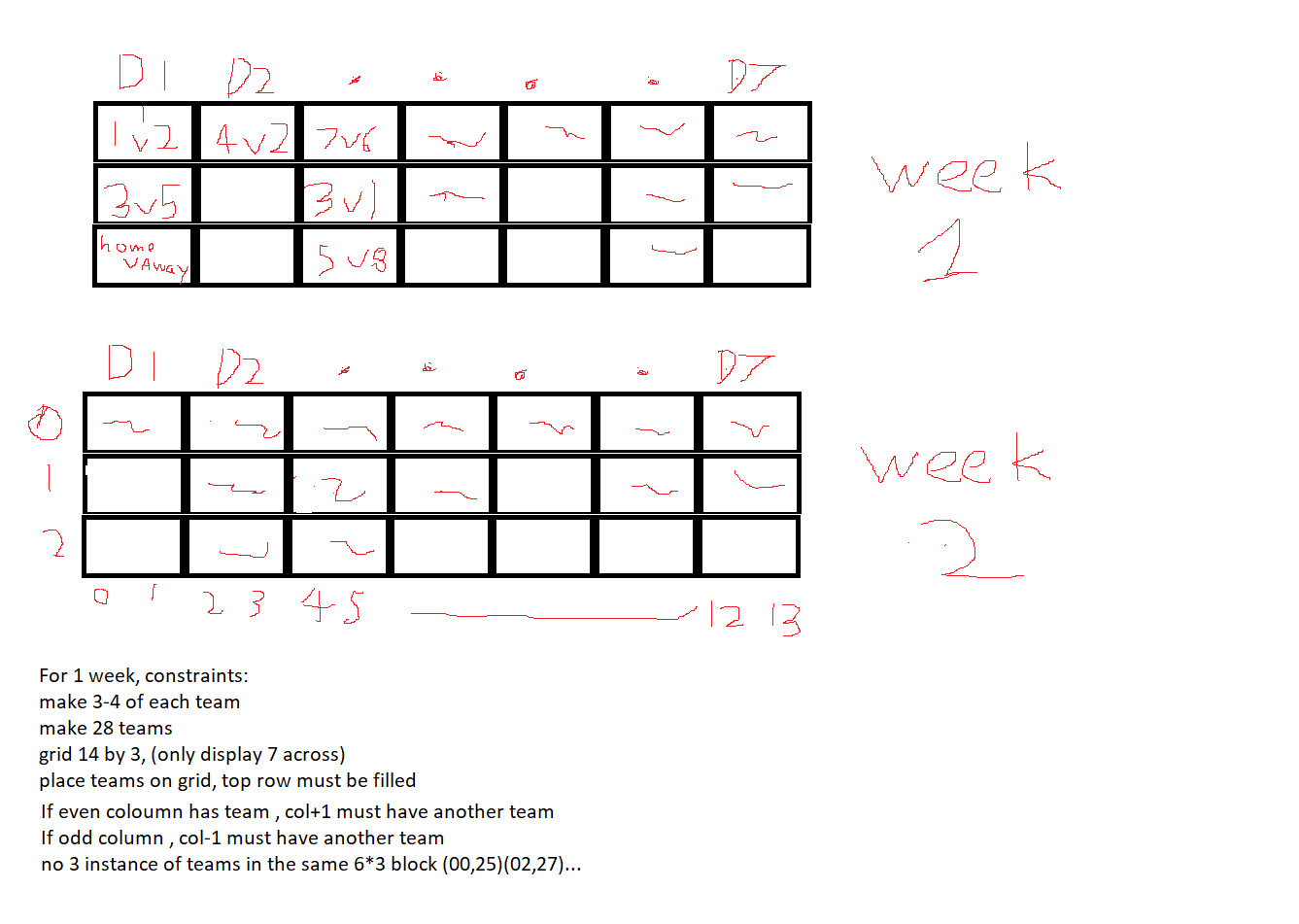
* **Each Team cannot play itself**

¬ G(Tx, Tx)

Model Exploration

1. Tried to model an entire schedule right away

Initially, we tried to explore the model by modelling the entire season’s schedule in one go. We tossed around many ideas for this and attempted to code it. But very early into the process of trying to model this way, we knew we had to focus our idea first, in order to create the entire schedule. In doing this, we began to draw up a rough sketch of how we could potentially focus the project’s ambitions.

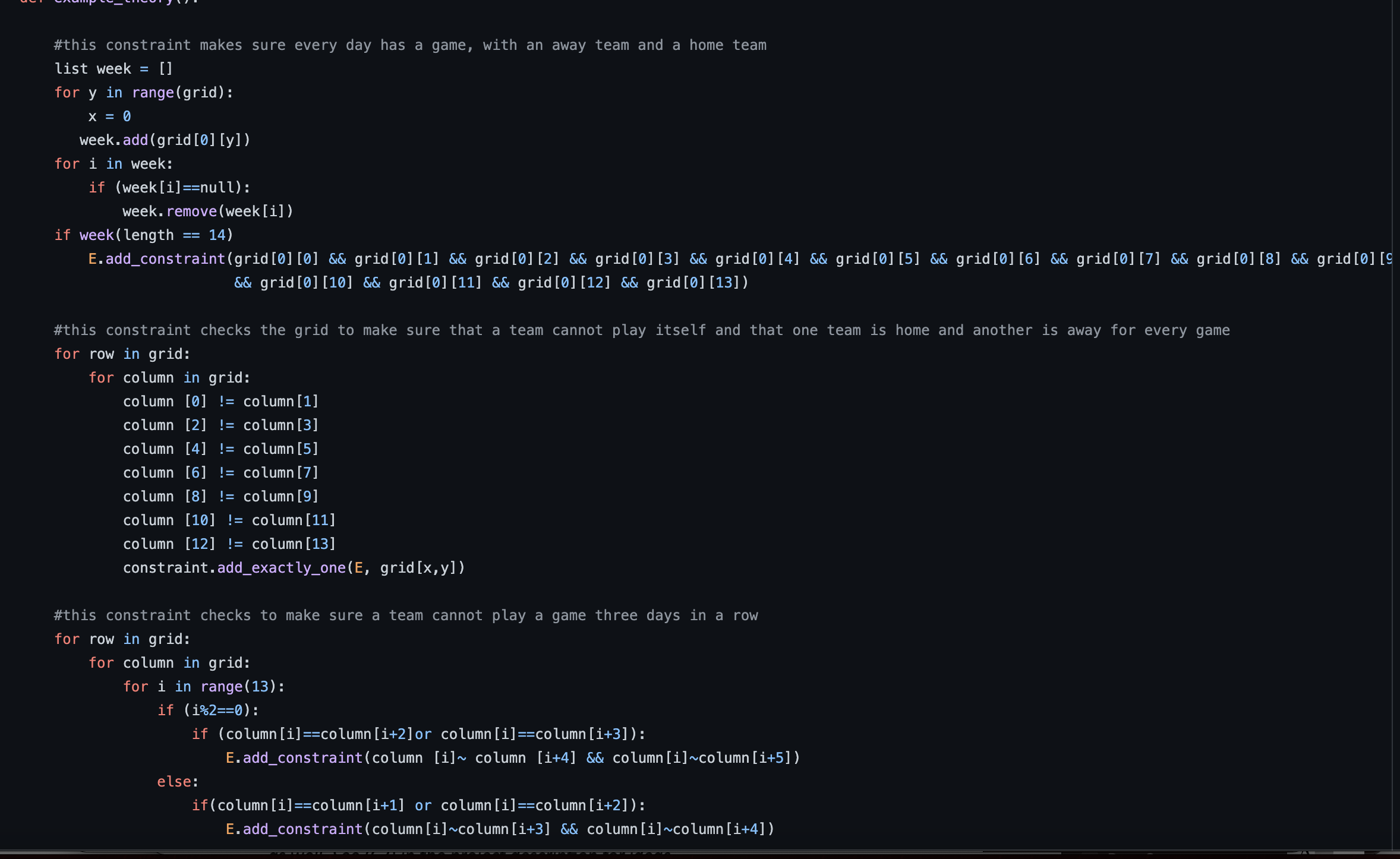


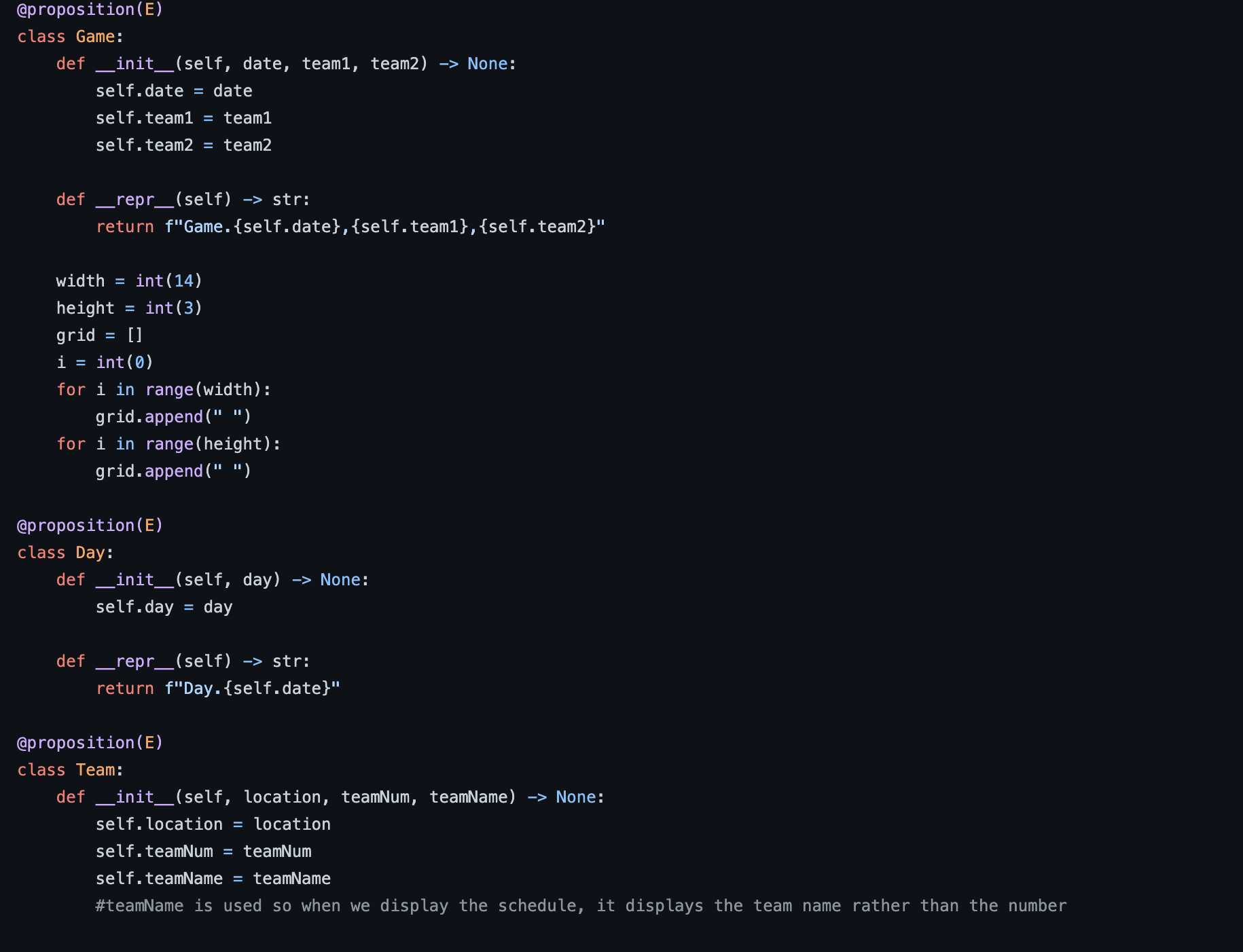
This sketch gave us a basic starting point in how we could begin to model the schedule on a weekly basis, rather an entire season at a time. We figured if we could produce a working week, we could then apply this to the 15 more times to create the 16 week season schedule. This sketch above gave us an idea of how we could approach the code to model this scenario.

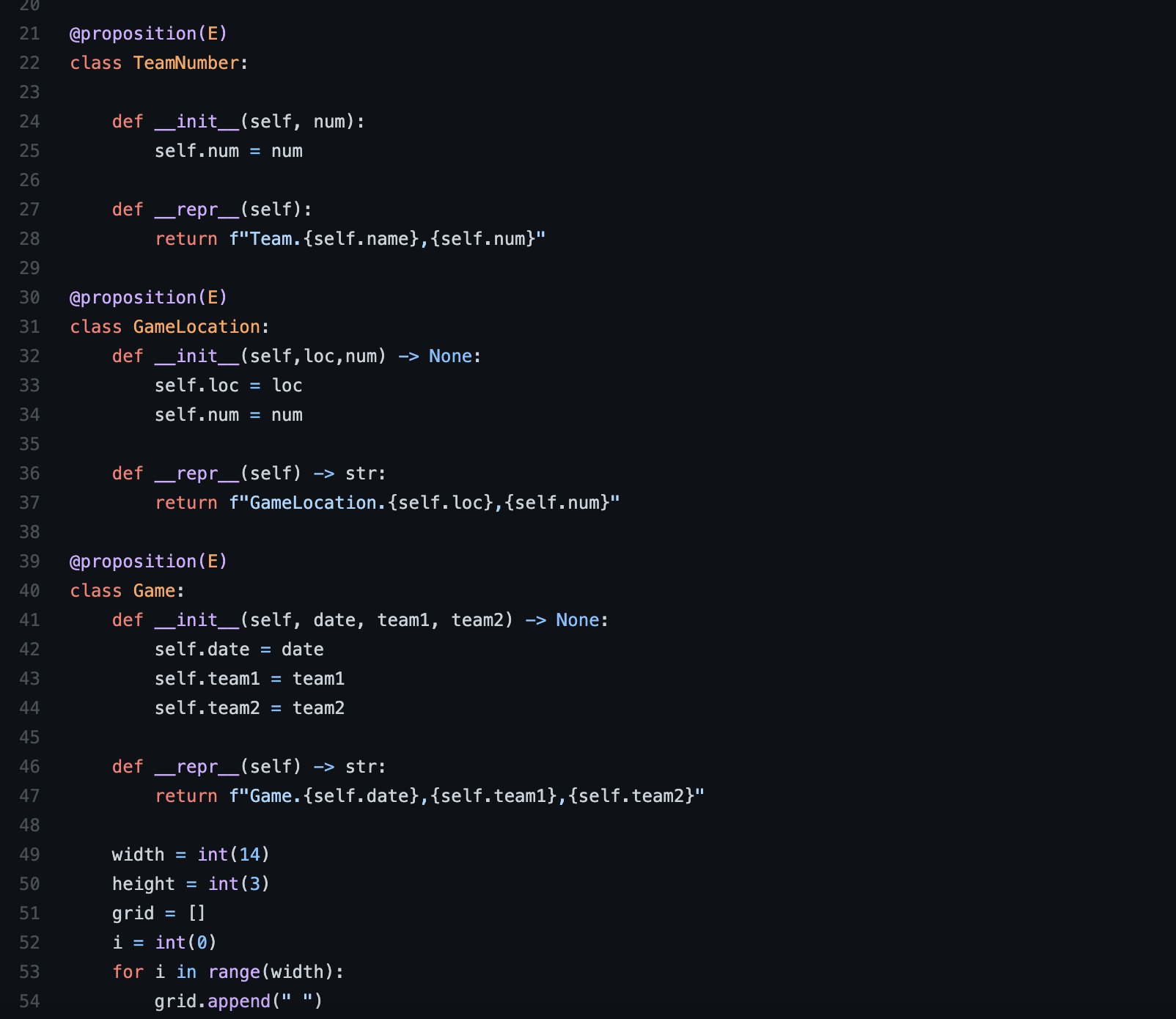
2. Initial Code for modelling a single week

Once we had the idea to model a single week, we began to start coding it, as seen in the screenshots below:









As we coded this model however, we recognized that changes had to be made. Initially we saw that we had to implement something in our code to generate a schedule, as we were using a grid with empty values. We didn’t want to take input to place the games in each spot in the schedule, so we knew we had to implement something that would produce a schedule, that we could then check for validity and rearrange accordingly.

Additionally, we also noticed redundancies in our propositions and constraints in doing this that we knew we could change. We initially had propositions and constraints that incorporated home and away location of teams, however as we explored the model we noticed that these were no longer needed as we could simply model the teams using their specific number, and one would always be a home team while the other is away in our model. Thus, we went and remade our propositions removing location as a factor all together. We removed the location constraints as well. We adjusted our propositions to better match the approach we had taken to model within the code, and alerted the constraints accordingly.

3. Improved approach to modelling a single week

To make the program easier we simplified the problem by first looking at a week. The program worked by pulling 14 random games from the total pool of games. The week structure is a 3 by 7 list, the 3 representing 3 time slots for each day and the 7 being the 7 days of the week. To start each time slot 1 is filled to meet the one game per day constraint. Next the remaining 7 games are placed by looking at each time slot one and seeing if they share a common team, if they don’t it places the game in time slot 2, if it does it moves to the next day. If the game can't be placed in time slot 2 it checks again in time slot 3. If it fails that it just places it in the first empty time slot 3. By going through these checks, it fills the constraint of a team not playing more than once a day. This program worked most of the time, but due to random chance it would sometimes get a pool of games that could not be placed in a valid configuration. This would be mostly solved in the next iteration of the program.

4. Loop it 16 times and solve each week

The final model makes 16 weeks and shows how many ways the days can be rearranged inside each week. The program works by selecting a pool of 14 games for each week. It selects games at random from the total pool of games and removes them from the overall pool. While selecting games if 4 of one team are already in the week pool and it selects another of the same team it will discard that game and pick another. With 14 games a week each team will play either 3-4 games. The problem is that by the third last week you start running in to the problem of there being too many of certain teams. If a team is mostly playing 3 games weeks by the end of the schedule, they have more games than 4 per week. For this reason, the last three weeks of the schedule just randomly pick games and don’t check for the limit of 4 games per week. The last 3 weeks often will not have any solutions as they are impossible to have a valid configuration under the given constraints. Using the same placement of games into days as mentioned in the one-week model, the program makes all 16 weeks and saves them in a list called schedule. Then each week is checked to see how many ways the days can be arranged. The program makes a list of all combinations of days and where they can be placed. By using the Bauhaus “add exactly one” function it makes sure only one of every position and day is true. It also checks each pair of days for common teams. If two days in a row have the same team adjacent days to the pair cannot have the same team. For example, if day 2 and 3 have team 1 playing, day 1 and 4 cannot have a game with team 1 in it. On the first run through of the code we had forgotten to add the only one of each day. This resulted in copies of the same day being true and having around 16000 solutions. Once the constraint was added the program worked as intended and was getting more reasonable numbers, less than 1000.

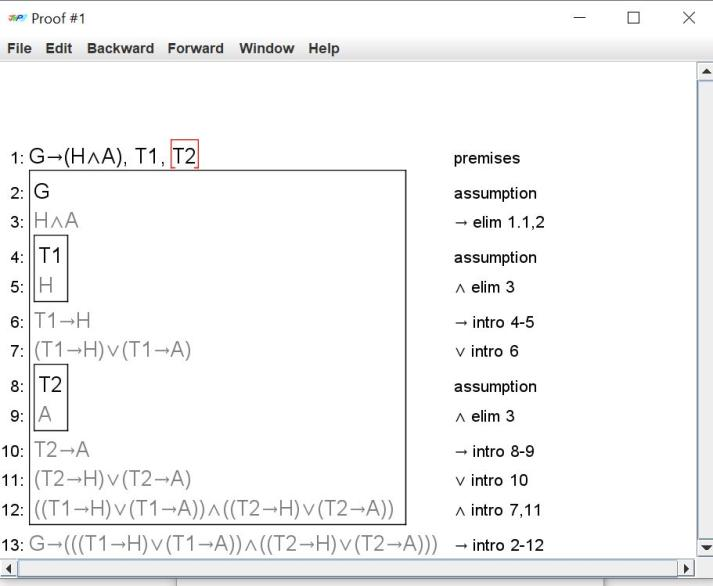
The next step in this program would be to incorporate Bauhaus into the making of each week and the schedule. This would allow us to make every week be satisfiable and allow us to find the total number of solutions for the overall schedule. However, due to only figuring out how to use Bauhaus late into the assignment we did not have enough time to implement this.

Jape Proofs

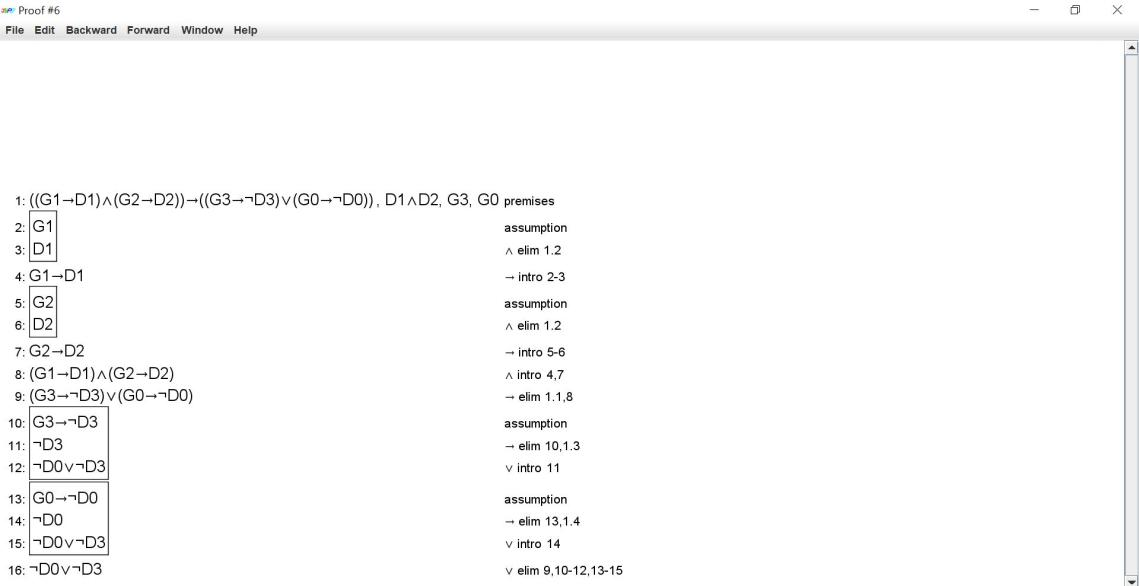
**D = Day**  **T = Team** **G = Game H = Home A = Away**

For our Jape proofs, we decided to model small instances of our larger project. Because our project is focused on scheduling games for a season, we decided to focus our Jape proofs on small instances that ensured the validity of games.

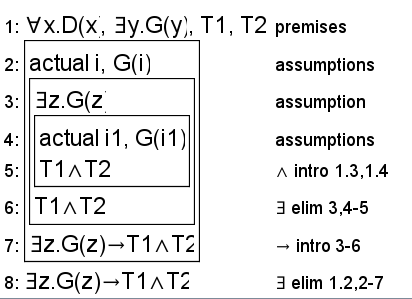
Our first Jape proof models that a game is valid, when one team is home, and the opposing team is away. This proof ensures that a game does in fact exist as a team cannot play itself. As seen below:



This second Jape proof models that a team cannot play a game, three days in a row. If we have each game implies a day, and thus based off the number of games we have as well as the number of days, we can deduce that a team cannot play a game three days in a row.



Our next proof had to incorporate some first order extension in order to properly model the instance. In this instance we are looking to prove that there is a game on each day of the week. Our Jape proof models that for all days in the week, there exists some game, and that implies that that game is played between Team 1 and Team 2.



First-Order Extension

To extend our project using first order extension, we will be applying first order extension to our constraints. We do not have to redefine our propositions as they were already constructed using predicates.

Therefore, in examining our constraints we can see that we can extend first order extension to a variety of our constraints, so that they can be better modelled with more complexity, as seen below:

**For all days in the week, a team does not play three games on three days in a row:**

∀x. ∀y. ∀z.D(x,y,z,T(x, ¬(G(x) ∧ G(y) ∧ G(z))))

**For all days in the week, every day in the week contains at least one game**:

∀x.W(x, ∀y.D(y, ∃z.G(z)))

**For all days there exists a game between two teams, that implies that the other games of that day do not contain either team:**

∀x.D(x, ∃y.G(y,(Tx ∧ Ty) → (¬ ∃z.G(z,(Tx v Ty)) ∧ ¬ ∃q.G(q,(Tx v Ty)))))

**For all days there exists a game, with two teams that need a game spot, that implies that the game is played between the two teams:**

∀x. D(x, ∃y. G(y), T1, T2)) → (∃z.G(z) → T1∧T2)