CMPT417

Project Report

LP/CP Programming Contest 2015
Sequential Games

Git: https://github.com/Duo-Lu/CMPT417.git

Duo Lu
301368672

Introduction

The goal of this project is to solve the **Sequential Games** problem given in the *LP/CP Programming Contest 2015*. We have sequence of games and each game has “fun” to play either positive or negative. However, we need to play each game for at least once and we can not “go back” to play previous games. We play each game for one token. In addition, we will get a refill for some token if we play a game for the first time(that does not mean each time we play then we get tokens, only for the first time.) But the number of tokens can not beyond capability we have. Finally, our goal is to find a sequence of play such that maximize the total fun.

Our problem given by lecture is to find the total fun such that total fun is greater than a nature number K. Since it is a decision problem(whether satisfied or unsatisfied), the solve will just find the first solution which total fun greater than K. I modified this problem by finding the optimal solution. That is, to find the maximum total fun. The reason why i wanted to do the search problem is that sometimes it is hard to find K and sometimes to find K is same as finding the optimal solution. Therefore, make it to search problem will more challenge and convenient to write test instance.

Solver System

I choose **MiniZinc** because **MiniZinc** have a nice IDE and the really good handbook, tutorial to teach you how to start. In addition, MiniZinc has comprehensive syntax such as **solve maximize** … or **solve satisfy** which are more powerful for our search problem which we decide to solve.

Problem Specification

Our instance vocabulary is (**num**, **cap**, **refill**, **fun**), where **fun** is a unary function and the other symbols are all constant symbols. A solution is a unary function **plays**. In addition, to represent the constraints of the problem, we will use a unary function, which gives the number of tokens in our pocket at the start of playing each game. That is, just before we start playing game i, we have **token(i)** tokens.

And we will give the interpretation to our problem again:

- You may play each game multiple times, but all plays of game Gi must be made after playing Gi−1 and before playing Gi+1.
- You pay 1 token each time you play a game, and you may play a game at most as many times as the number of tokens you have when you begin playing that game.
- You must play each game at least once.
- You have **“cap”** tokens when you start playing G1. After your last play of Gi , but before you begin playing Gi+1, you receive a “refill” of up to **“refill”** tokens. However, you have some “pocket capacity” **cap**, and you are never allowed to have more then **cap** tokens.

- Each game has a “fun” value for you, which may be negative

Therefore, we will be given the number of games we will play which is a constant symbol. The capacity which tokens can not allowed to exceed which is a constant symbol. The refill for each game when we play for the first time on that game which is also a constant symbol. And a unary function **fun : [num] → N** which give us the fun for each game.
The **MiniZinc** equivalent encode of our vocabulary is:

```
% Given
int: num;                    % Number n ∈ N of games;
set of int: games = 1..num;
int: refill;                 % Refill amount R ∈ N;
set of int: x_total = 0.. (cap + refill);
array [games] of int: fun; % Fun value vi ∈ Z for each game i ∈ [n];
int: cap;                    % Pocket Capacity C ∈ N;
```

And our problem is to find the sequence of play which can maximize the total fun and in order to find our solution. For each step, we also need to know how many tokens we have. Therefore, as soon as we find the sequence of play, we also find the number of tokens at each step. And we will find **play : [num] → N** and **token : [num] → N.** The **MiniZinc** equivalent encode of our vocabulary is :

```
% Find
array[games] of var int: t;
array[games] of var int: plays;
```

Notice that plays and tokens are arrays with decision variable so that we use **var int** which MiniZinc know that it should be decided by itself.

To find a solution to L-vocabulary M, the next step is to define our constraints to our problem:

- **We play each game at least once, and at most ti times**
  Our First Order Logic is:

$$\forall i[(1 < i \leq n) \rightarrow (1 \leq p(i) \leq t(i))]$$

  And the MiniZinc equivalent encode is:

```
% 1. We play each game at least once, and at most times
constraint forall (i in games) ((1 <= i /\ i <= num) -> (1 <= plays[i] /\ plays[i] <= t[i]));
```

- **The number of tokens ti available to play game i is C when we start playing the first game, and for i > 1 is the minimum of C and ti−1 − pi−1 + R**
  Our First Order Logic is:

$$t(1) = C \wedge \forall i[1 < i \leq n \rightarrow \exists x((x = t(i-1) - p(i-1) + R) \wedge (x > C \rightarrow t(i) = C) \wedge (x \leq C \rightarrow t(i) = x))]$$

And the MiniZinc equivalent encode is:

```
% 2. The number of tokens ti available to play game i is C when we start playing the
% first game, and for i > 1 is the minimum of C and ti−1 − pi−1 + R:
constraint (t[1] = cap) /\ ((forall (i in games)
((1 < i /\ i <= num) -> (exists(x in x_total) ( (x = (t[i-1] - plays[i-1] + refill)) /\
 ( (x > cap) -> (t[i] = cap) ) /\ ( (x <= cap) -> (t[i] = x) )  )))));
```

And finally we should tell the MiniZinc what to solve and what is the output. Due to the modified problem, what we solve is the search problem and find the optimal solution. Therefore, we should tell the MiniZinc,

```
solve maximize sum(i in games)(plays[i] * fun[i]);
```

And trivially, our output is,

```
output ["fun for each games          " ++ show(fun) ++ "\n" ++
    "number of tokens in each game " ++ show(t) ++ "\n" ++
    "sequence of play              " ++ show(plays) ++ "\n" ++
    "total fun: " ++ show (sum(i in games)(plays[i] * fun[i]))] ;
```

Our Problem specification is already done. The next step is to write some test instance to test whether our program is correct.

Test Instances

How many test instances we have? For testing general program correct and speed, we have probably 14 test instances and they are **basicData_num** in the data file which **num** represents the number of games

How we obtained or constructed them? Basically, we write a data_generator.py to generator the data files. What we test the number of games are from 4 to 17(solve 17 games need to spend 29 minutes). Moreover, for all **cap** and **refill** is 5 and 2 respectively. Most important, for the fun for each game, we generated them by Python random module and they are between -20 and 20. Let's take a look for our test instances.

```
% number of games is 4 and it is from LP/CP Programming Contest 2015
num = 4;
cap = 5;
refill = 2;
fun = [4,1,2,3];
```

And use this to test our problem

fun for each games [4, 1, 2, 3]
number of tokens in each game [5, 2, 3, 4]
sequence of play [5, 1, 1, 4]
total fun: 35

==========

The dot line indicates that here are one solution and equality line shows that no other solution **which means here are the optimal solution**. Let's show the correctness of this test. At the beginning of the game, we have the maximum tokens which is the capacity. Because the first game are the most fun game so play maximum time that we can play. That is correct for the first step. And then at the second game, we give 2 refills so our tokens become 2. However, the second game is the most boring game. Therefore, we only play once(due to the constraints that we play every game at least once). Same as the third game. And finally for the last game, we should play all the tokens we have. The result is 35 which is the same as the ***LP/CP Programming Contest 2015*** given us.

fun for each games [1, -1, 2, -1, 2, 0, -1, 2, -4, 2, -1, 5, -3, 12]
number of tokens in each game [5, 4, 5, 4, 5, 3, 4, 5, 4, 5, 4, 5, 4, 5]
sequence of play [3, 1, 3, 1, 4, 1, 1, 3, 1, 3, 1, 3, 1, 5]
total fun: 93

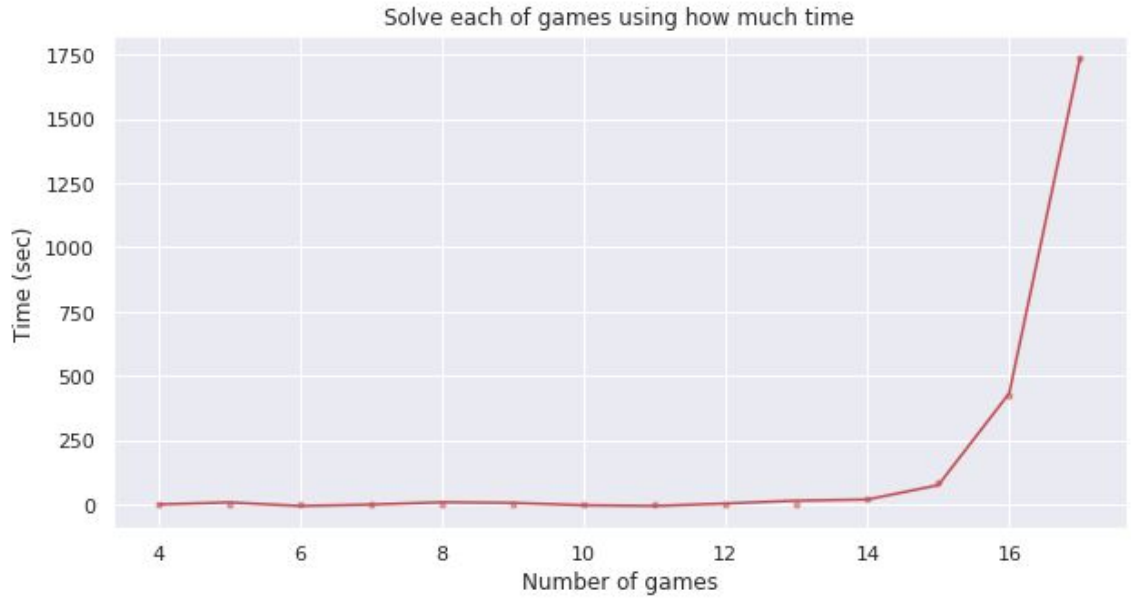
==========

And check for another test instance which included the negative numbers. We can clearly see that for each negative fun games we only play once(I use red color to highlight it make it more clear.) Moreover, for the last game which has the most fun and we should play most times. The last third game has the second fun so we play second most times and so for other games. In general, we can easily write polynomial checker(O(n)) to check whether the output is correct.

Performance Test

The running output below is using **Gecode** which is default solver configuration in MiniZinc. And data files are in basic_data folder. The postfix of data name represents the number of games. Moveover, the capacity and refill is 5,3 respectively. The average running time we can see that the number of games from 4 to 11 is running really fast. However, if the number of games is 17, solving it need to spend almost half an hour.

data	capacity	refill	fun	avg (time/s)
basicData_number 4	5	2	[14,-10,4,-11]	0.254
basicData_number 5	5	2	[9,-2,15,-12,4]	0.2376
basicData_number 6	5	2	[-19,-6,-14,-15,8,-9]	0.2542
basicData_number 7	5	2	[15,1,-8,-17,4,-20,10]	0.256
basicData_number 8	5	2	[-14,12,6,-15,-6,-9,-1,-6]	0.4328
basicData_number 9	5	2	[12,8,-17,9,6,6,3,-3,-11]	0.3632
basicData_number 10	5	2	[11,-4,-8,-7,16,-4,14,-13,-11,1]	0.3174
basicData_number 11	5	2	[-17,15,-13,17,-4,-16,9,-4,-6,-20,-10]	0.3122
basicData_number 12	5	2	[18,-17,-18,13,-9,0,-18,0,-7,1,19,11]	0.761
basicData_number 13	5	2	[19,-19,10,8,-19,-13,4,-1,15,-15,-10,-2,-4]	2.628
basicData_number 14	5	2	[19,-2,0,-17,4,-15,0,9,15,-12,-15,15,-14,1]	19.53
basicData_number 15	5	2	[-1,-10,2,-1,2,0,1,2,19,2,1,5,-3,12,-5]	84
basicData_number 16	5	2	[19,19,-15,-9,-16,16,-13,14,-4,12,-6,1,13,8,-13,17]	424
basicData_number 17	5	2	[-8,-12,2,-1,2,0,1,2,19,2,1,5,-3,12,-5,11,-8]	1740



Here is the graph for the performance test.
Moreover, I use python to build a **Machine Learning Model** to draw a regression line because I want to know when number of games become 20 or more. What solving time will be. **More information about it, please open it in jupyter-notebook.**
Here is the regression line:
Time = -1.177 + 892.33 * X - 1789.32 * X^2 + 127810.2 * X^3 - 43186.6 * X^4 + 749189.34 * X^5 - 645724.898 * X^6 + 219332.175 * X^7
And when we have a number of games 20. We will approximately spend **30974.97762045s** which probably around 8 hours to solve the problem.

QUESTION I

Is there any constraints to deal with the negative fun games?

ANSWER

Because the negative fun games can only decrease the total fun. We don't have to make decision for negative fun games and we just leave them playing once only.

EXPERIMENT

We just add one more constraint into MiniZinc.

```
% If we have negative fun for a game, We just want to play this game for once
% That is we just need to decide the number of plays for positive fun
% In other words, we ignore negative fun games
constraint forall (i in games) (fun[i] < 1 -> plays[i] = 1);
```

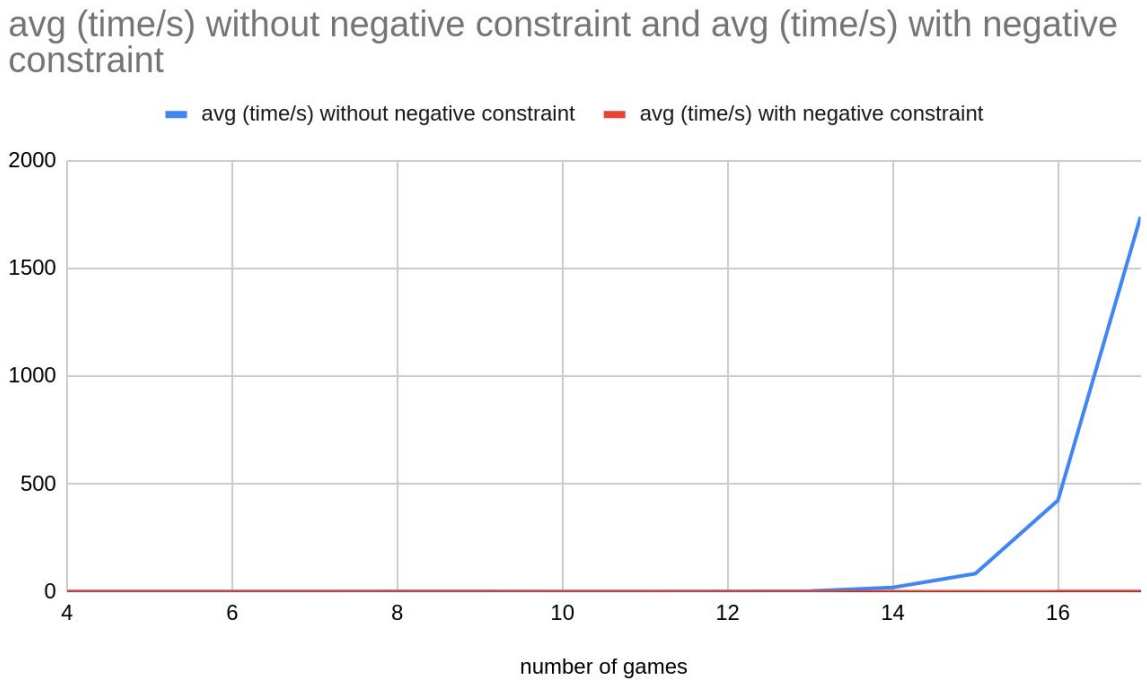
That says if the fun of game is less than 1, and that implies we just play it once only.

And how it improves the solver to solve the problem?

data	capacity	refill	fun	avg (time/s)	number of positive games	total_fun
negative_constraint 4	5	2	[14,-10,4,-11]	0.11	2	61
negative_constraint 5	5	2	[9,-2,15,-12,4]	0.128	3	100
negative_constraint 6	5	2	[-19,-6,-14,-15,8,-9]	0.102	1	-23
negative_constraint 7	5	2	[15,1,-8,-17,4,-20,10]	0.129	4	93
negative_constraint 8	5	2	[-14,12,6,-15,-6,-9,-1,-6]	0.109	2	21
negative_constraint 9	5	2	[12,8,-17,9,6,6,3,-3,-11]	0.131	6	103
negative_constraint 10	5	2	[11,-4,-8,-7,16,-4,14,-13,-11,1]	0.163	4	134
negative_constraint 11	5	2	[-17,15,-13,17,-4,-16,9,-4,-6,-20,-10]	0.111	3	76
negative_constraint 12	5	2	[18,-17,-18,13,-9,0,-18,0,-7,1,19,11]	0.134	7	192
negative_constraint 13	5	2	[19,-19,10,8,-19,-13,4,-1,15,-15,-10,-2,-4]	0.142	5	141
negative_constraint 14	5	2	[19,-2,0,-17,4,-15,0,9,15,-12,-15,15,-14,1]	0.184	7	192
negative_constraint 15	5	2	[-1,-10,2,-1,2,0,1,2,19,2,1,5,-3,12,-5]	0.192	10	167
negative_constraint 16	5	2	[19,19,-15,-9,-16,16,-13,14,-4,12,-6,1,13,8,-13,17]	0.31	9	349
negative_constraint 17	5	2	[-8,-12,2,-1,2,0,1,2,19,2,1,5,-3,12,-5,11,-8]	0.49	12	183
negative_constraint 18	5	2	[8,-10,-10,-8,14,0,-16,9,-6,18,9,-16,-5,7,-6,12,16,-4]	0.262	9	273
negative_constraint 19	5	2	[-1,-5,-18 ,16,-5 ,0,4,18,6,-16,9,3,14,10,17,-1,0,-4,-17]	0.257	11	271
negative_constraint 20	5	2	[17,-8,19,17,-9,18,20,1,19,-12,1,3,-2,5,4,-13,-11,-2,-12,-7]	1.962	11	303
negative_constraint 21	5	2	[-16,-11,-18,-14,15,-16,-16,6,15,16,1,19,15,4,13,19,-15,7,-7,-16,-1]	1.85	13	273
negative_constraint 22	5	2	[-5,10,-20,14,12,13,11,1,19,6,-13,7,3,14,-14,10,13,-2,14,-3,-11,-10]	180.023	14	366
negative_constraint 23	5	2	[15,-11,-18,-10,0,11,18,11,-7,-15,0,-19,19,7,-17,-5,17,-20,-16,-13,-12,-13,19]	0.187	10	315
negative_constraint 24	5	2	[-10,10,11,5,-17,14,5,-4,4,4,0,19,-17,-4,8,-10,-10,-14,-20,-2,12,-13,7,8]	120.009	13	266
negative_constraint 25	5	2	[-3,-15,19,9,-4,-12,3,5,-4,10,3,8,-12,10,14,6,9,2,17,14,12,-16,-3,0,6]	N/A	17	397

We can see that MiniZinc can solve more games if we add the constraint. Before we add the constraint, to solve the 17 games need to spend almost a half hour. However, we can easily solve 17 games in a half second now. **Moveover, we can solve number of games until 25**. If we consider the data a little bit, we find that the MiniZinc only need to make the decision for positive fun games. That means the solver only need to solve number of positive fun games. However, the maximum number of games MiniZinc can solve are around 25 or 26 which are still small size instance. **Although we can not write a logic proof to prove the correctness of this constraint. But after comparing the result with previous'. We can**

assume for small instances. Adding this constraint will not destroy the correctness of the program.



And here is the graph comparing adding the constraint and without the constraint.

QUESTION II

If we change another solver configuration, can it improves the solving speed?

ANSWER

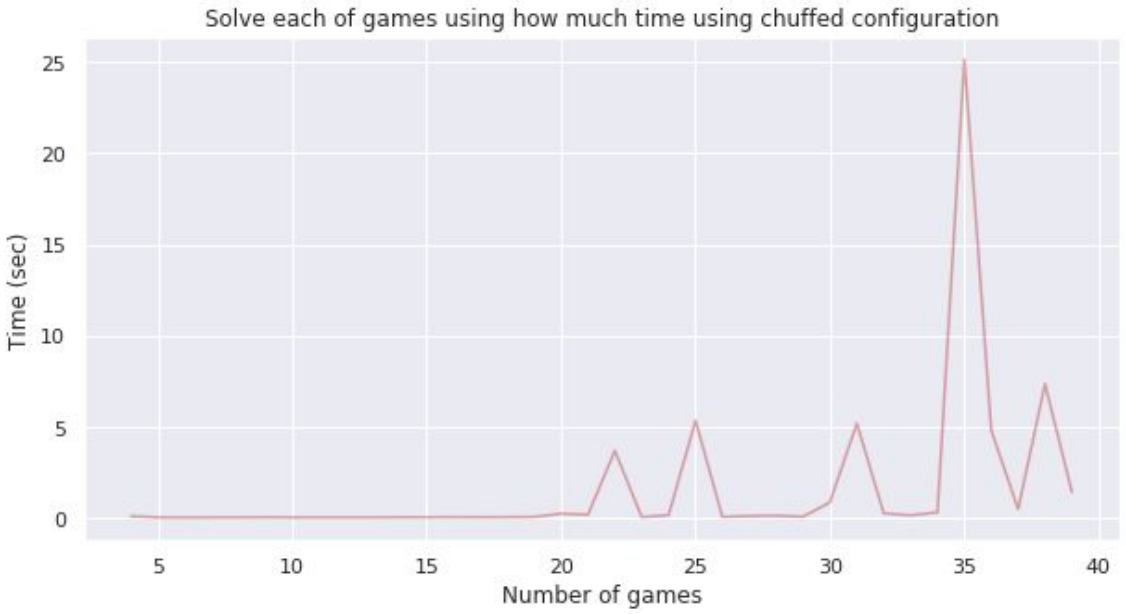
The default solver for MiniZinc is Gecode which Gecode represents Generic *Constraint Development Environment*. In addition. Gecode implement with C++ which gains high speed and memory efficiency. However, in the process of data collection, we find **Chuffed** is faster to solve Sequential Games problem. Therefore, we try to use **Chuffed** as our solve configuration to test the performance of it.

EXPERIMENT

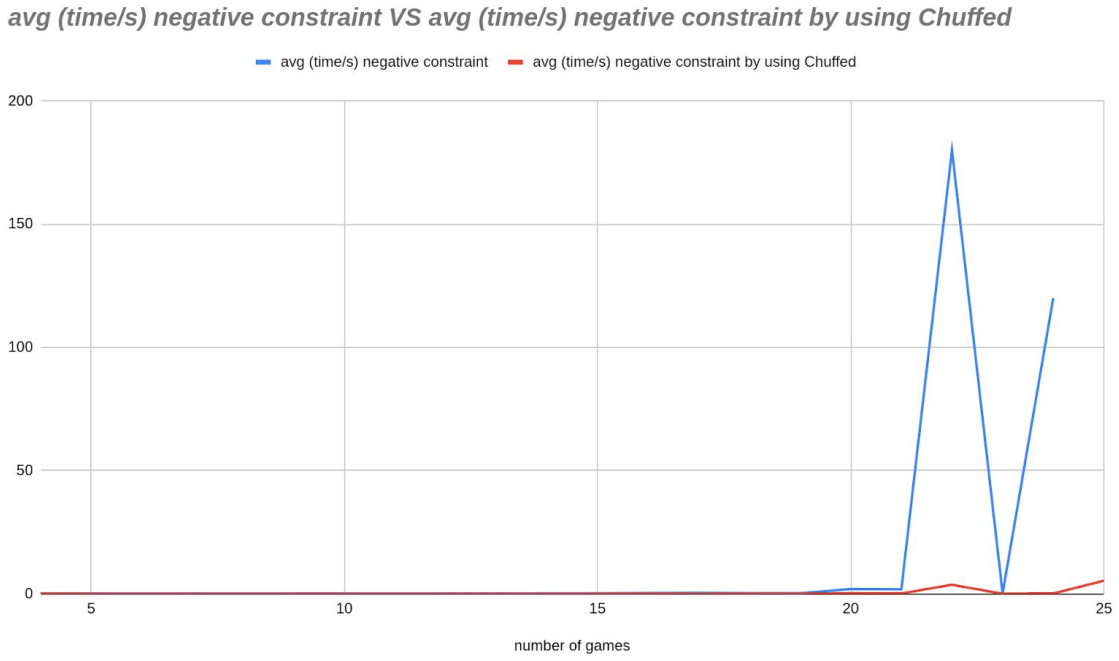
The experiment for this problem is actually quite easy. We just change the solver configuration in MiniZinc IDE. But the data output is quite amazing for us.

data	capacity	refill	fun	avg (time/s)	total fun
chuffed_configuration 4	5	2	[14,-10,4,-11]	0.11	61
chuffed_configuration 5	5	2	[9,-2,15,-12,4]	0.04	100
chuffed_configuration 6	5	2	[-19,-6,-14,-15,8,-9]	0.03	-23
chuffed_configuration 7	5	2	[15,1,-8,-17,4,-20,10]	0.04	93
chuffed_configuration 8	5	2	[-14,12,6,-15,-6,-9,-1,-6]	0.04	21
chuffed_configuration 9	5	2	[12,8,-17,9,6,6,3,-3,-11]	0.05	103
chuffed_configuration 10	5	2	[11,-4,-8,-7,16,-4,14,-13,-11,1]	0.04	134
chuffed_configuration 11	5	2	[-17,15,-13,17,-4,-16,9,-4,-6,-20,-10]	0.04	76
chuffed_configuration 12	5	2	[18,-17,-18,13,-9,0,-18,0,-7,1,19,11]	0.04	192
chuffed_configuration 13	5	2	[19,-19,10,8,-19,-13,4,-1,15,-15,-10,-2,-4]	0.04	141
chuffed_configuration 14	5	2	[19,-2,0,-17,4,-15,0,9,15,-12,-15,15,-14,1]	0.05	192
chuffed_configuration 15	5	2	[-1,-10,2,-1,2,0,1,2,19,2,1,5,-3,12,-5]	0.05	167
chuffed_configuration 16	5	2	[19,19,-15,-9,-16,16,-13,14,-4,12,-6,1,13,8,-13,17]	0.06	349
chuffed_configuration 17	5	2	[-8,-12,2,-1,2,0,1,2,19,2,1,5,-3,12,-5,11,-8]	0.05	183
chuffed_configuration 18	5	2	[8,-10,-10,-8,14,0,-16,9,-6,18,9,-16,-5,7,-6,12,16,-4]	0.06	273
chuffed_configuration 19	5	2	[-1,-5,-18 ,16,-5 ,0,4,18,6,-16,9,3,14,10,17,-1,0,-4,-17]	0.07	271
chuffed_configuration 20	5	2	[17,-8,19,17,-9,18,20,1,19,-12,1,3,-2,5,4,-13,-11,-2,-12,-7]	0.25	303
chuffed_configuration 21	5	2	[-16,-11,-18,-14,15,-16,-16,6,15,16,1,19,15,4,13,19,-15,7,-7,-16,-1]	0.19	273
chuffed_configuration 22	5	2	[-5,10,-20,14,12,13,11,1,19,6,-13,7,3,14,-14,10,13,-2,14,-3,-11,-10]	3.7	366
chuffed_configuration 23	5	2	[15,-11,-18,-10,0,11,18,11,-7,-15,0,-19,19,7,-17,-5,17,-20,-16,-13,-12,-13,19]	0.06	315
chuffed_configuration 24	5	2	[-10,10,11,5,-17,14,5,-4,4,4,0,19,-17,-4,8,-10,-10,-14,-20,-2,12,-13,7,8]	0.17	266
chuffed_configuration 25	5	2	[-3,-15,19,9,-4,-12,3,5,-4,10,3,8,-12,10,14,6,9,2,17,14,12,-16,-3,0,6]	5.35	397
chuffed_configuration 26	5	2	[-4, 1, 1, -19, 16, -20, 5, -20, 9, 0, -5, -20, 16, -18, 12, 11, 17, -11, -5, 16, -5, -13, -9, -2, -8, -12]	0.07	216
chuffed_configuration 27	5	2	[-13, 11, 17, -11, -6, 19, -4, 18, 0, -16, 6, -20, 17, 2, -17, -18, 9, 9, 4, -14, -12, -1, -8, 0, 10, 17, -7]	0.12	367
chuffed_configuration 28	5	2	[5, -10, -12, -6, 7, 14, -8, 5, 14, 12, -8, -7, 9, -10, 3, -15, -17, -1, 1, 0, 7, -5, -14, 3, 9, 2, -4, 10]	0.14	251
chuffed_configuration 29	5	2	[17, -4, 7, -20, -8, -11, -10, 1, -1, 11, 6, 2, -14, -3, -7, -19, 0, -5, -20, 5, 16, -18, -17, -3, 1, 2, 6, -13, -18]	0.08	115
chuffed_configuration 30	5	2	[14, -4, -9, 20, 2, 20, -2, -7, 8, 1, 7, 2, -11, -8, 17, 9, -4, -7, -6, 2, 8, -19, 13, 5, 3, 6, 0, -12, -16, 5]	0.87	422
chuffed_configuration 31	5	2	[-6, -1, 18, 9, 15, -6, -2, -16, 16, -5, 0, 3, 8, -20, -13, -19, 11, 3, 11, 19, 2, -8, -18, 14, 9, 12, 4, -5, 8, 14, 1	5.2	543
chuffed_configuration 32	5	2	[-7, -2, 12, -16, 13, -2, -16, 18, 11, 3, -18, -5, 11, 3, 7, 6, -13, -17, -13, 13, 5, 2, -18, -11, 18, 8, -15, -18, -	0.27	276
chuffed_configuration 33	5	2	[-1, -17, 5, 4, -3, -18, 16, -12, -17, -17, -14, -14, 0, -16, -15, 20, 3, 4, 8, 4, 5, -6, -6, 7, -3, -13, 12, 11, 5, -	0.16	163
chuffed_configuration 34	5	2	[-7, 13, -12, -6, -2, 15, -19, -1, 18, 3, 18, -5, 8, -15, -11, -19, 1, 9, 3, 4, -10, 15, -11, 12, -4, -20, 14, -11, -	0.32	486
chuffed_configuration 35	5	2	[0, 14, -7, 2, 10, 4, -19, 14, 6, 14, 5, -11, -4, 9, 2, 3, -7, 9, 0, 14, 17, 20, -8, 1, -14, 20, 8, -15, -19, -16, -16	25.13	497
chuffed_configuration 36	5	2	[-19, 8, -3, -3, 4, -13, -12, 11, 16, 8, 12, 7, 20, -16, -7, -5, 1, -6, 3, -6, -16, 14, 8, -9, 19, 6, -18, 19, -18, -1	4.81	501
chuffed_configuration 37	5	2	[-20, -4, 3, 9, -9, -12, -19, 2, 7, 6, -20, 1, 3, 19, -11, 11, -1, 3, -13, -9, -1, 7, -2, -1, 4, -14, -5, -8, 20, -12, 3	0.5	385
chuffed_configuration 38	5	2	[-17, -5, -13, 1, 14, -5, -19, -4, -11, -15, 9, 12, 19, 16, 6, 0, -6, 15, -1, 16, 14, 10, -8, -6, -13, -20, 6, -16, 1	7.36	473
chuffed_configuration 39	5	2	[-20, -15, 17, 4, 4, 15, 3, -6, -6, -16, -5, -14, -16, -12, 17, -1, -14, -4, 0, -10, 20, 19, 16, -8, -19, -11, 1, 18,	1.43	494
chuffed_configuration 40	5	2	[-7, 16, 13, -18, -6, -18, 12, 4, 6, 4, 16, -7, 11, 15, -6, -18, -7, -17, -1, 11, 10, -5, -19, 0, -7, 15, -3, 14, -5,	null	null

Notice: if you would like to see the data more clearly, please go to the folder and find data_info.xlsx. And if you would like to run the data to test the performance or correctness, please go to the folder and open the terminal. Enter exactly in python MiniZinc_data_runner.py. Next enter how many games you would like to run(maximum is 40). And you will get output exactly up to the number you enter(4 to that number). More information please go appendix.



For small problem instances, like the number of games from 4 to 17, Chuffed can amazingly solve it in roughly 0.05s compares to 0.1s which Gecode generally did before. I think this time may just be a set up time for converting constraints to CNF and call the SAT solver. Moreover, Gecode can only solve number of games around 25 or 26. But chuffed, can solve it until roughly 40 games. **And also, the solving time relative to how many positive fun games you have**. That's the reason why number of games **35** is solving in **25.13s** but number of games **37** is solving in **0.5s.** Solving more positive fun games spend more time.



Here is graph comparing using Gecode and use Chuffed to solve number of games until 25. Moreover, using Gecode can only solve number of games around 25.

Also we can not write a logic proof to prove the correctness of using different solve configuration. But after comparing the result with previous'. We can assume for small instances. Adding this constraint will not deprive the correctness of the program.

DISCUSSION

Here comes another question. Why **Chuffed** is faster than **Gecode** in this question. Chuffed is a progressive lazy clause solver. It is designed from the ground up and takes into account the generation of lazy clauses. Lazy clause generation is a hybrid method of constraint solving, which combines the characteristics of finite field propagation and Boolean satisfiability. Therefore lazy clause generation is an extremely important and useful technology. However, theory behind lazy clause generation is described in much greater detail in various papers. We decide to not go in deepening in Chuffed. **But in the future study, if we do more research about constraint satisfaction problem, we might spend some time to go through the implementation of Chuffed and gain more ideas from it.**

Notice: the coding part of this question is inspired by my classmate George He. We had discussed the idea together and I acknowledge that I had read his complete code.

QUESTION III

Can we write a **specific solver** aim to solve **only for sequential games** problem? Can it does in **polynomial time**? How correct will the solver be? Can it transfer to another CSP like pizza problem?

ANSWER

We can write a specific solver aims to solve the sequential games which implemented with **c++**. Most importantly, the solver can run in **O(n^2)** in the worst case. In experiment part, we will discuss more implementation of algorithm in detail.

EXPERIMENT

First of all, we need to generate some data. Therefore, we use data_generator.py which we wrote before to generate some instances. If you want to see more detail, please go to the folder and find solver and then go into the data folder.

```
for i in range of number of games:
        play[i]++;
        token[i] = current_token--;
        if game i has negative fun:
                total_fun += fun of game i;
                current_token = min(current_token + refill, capacity);
        end if
        else game i has positive fun:
                next_game = i + 1;
                future_token = refill;
                while future don't reach the capacity:
                        if fun of next game is greater than the current game AND next
                                                        game is not reach the number of games:
                                reserve = capacity - future_token;
                                If reserve greater than the current_token:
                                        reserve = current_token;
                                break;
                        next_game++;
                        future_token += refill - 1;
                end while
        end else
                current_token -= reserve;
                play[i] += current_token;
                total_fun += (play[i] * fun[i])
                reserve = 0;
end for
```

And then we need to think about the algorithm a little bit. The negative games, we just play it once and skip it as before. The key point for this algorithm to how to deal with the positive fun games and how to reserve some tokens to those games with more fun.

We assume the capacity is 5 and refill is 2.
The basic idea for this algorithm is we have a variable called current_token which remembers how many tokens we have in front of the game. Hence, in the first game, the current_token should be capacity.
In addition, we have another variable called future_token. Future_token is to calculate if we play all the tokens in the current game, how many tokens have in the next few games. For example, if we play all the tokens in the first game, we will have 2 tokens in the second game and 4 in the third game.
Moveover, we have a variable reserve and it calculate how many tokens we should reserve for the more funny game than the current game if we have.

We will show some examples running the algorithm.

% number of games is 5
num = 5;
cap = 5;
refill = 2;
fun = [14, 2, 18, 1, 20];

At the beginning of the algorithm, we have 5 tokens for the first game. And then we stand at the first game, we will check if there exists a game which the fun of it greater than current game in the next **floor(capacity/refill)** games. If we do not exist such games, that means the current game is the most fun game now. And then we could play all the tokens we have. If in the next **floor(capacity/refill)** games have more funny game than the current game, we calculate reserve as **capacity - future_token**, which means we should reserve tokens to next game that have **potential** to play the capacity times. In this example, 18 is greater than the first games and future_tokens is 3 thus we need to reserve 2 tokens to the third game(the third game at least have more qualification play more time than the first game now). Hence, the first game can only play 3 times and leave some tokens to the third game. Same as the second game, it needs to reserve 3 tokens to the next game. As we stand at the third game, we find that the last game should play the capacity. Therefore, we can not play the third game many times because we need to reserve 2 tokens to the last game. Hence, the answer should be **play = [3, 1, 3, 1, 5].**

Running time analysis:

The best case is capacity/refill is one and this case we just need to look at the next first game. And basically the running time is **O(N)**.
The worst case is capacity/refill is really large which greater than number of games and fun array in decreasing order. Hence, for one game, we need to look at all the remaining games. The number of critical operations is **N * (N -1) / 2** (N is number of games) which is actually **O(N^2)**.

data	capacity	refill	fun	avg (time/s)
solverData_num 5	5	2	[-4, -7, 17, 16, -11]	0.0000sec
solverData_num 20	5	2	[14, 3, -12, 14, 18, -10, -19, 5, -14, -20, 2, -1, 12, 17, 3, -17, 2, -4, -8, 12]	0.0001sec
solverData_num 30	5	2	[19, -11, -6, 15, -13, 17, -4, 4, -14, -1, 17, 18, -5, 15, 4, 9, -15, -1, 11, -9, 4, -13, 5, -17, -20, -13, -16, -4, -	0.0001sec
solverData_num 100	5	2	[-15, 9, -16, 4, 14, 10, 0, 5, -16, -2, -15, 5, -6, 8, -14, 20, -20, -14, -12, -3, -4, 16, -19, 15, 18, -8, -4, 1, 12,	0.001sec
solverData_num 500	5	2	[-8, -17, 12, -20, -8, 10, -8, 12, -2, 15, 4, 16, -2, -19, -19, -15, -4, -10, -10, 7, 6, 14, -3, -20, -20, 20, 13, 19	0.002sec
solverData_num 1000	5	2	[-13, -4, -7, -6, -16, -18, -7, -10, -18, 15, 3, -15, 19, 11, 2, 4, 1, -17, -4, -2, -2, -9, -2, -12, -1, 8, 16, -8, -8, -	0.1sec
solverData_num 5000	5	2	[10, 11, -1, -16, -6, -1, -17, -11, 0, -17, -20, 14, -1, -12, 20, -15, 6, -19, -14, -6, -2, -15, 8, -7, -9, 1, -2, 19,	0.5sec
solverData_num 10000	5	2	[11, -4, 8, -3, -5, -8, 16, -13, 7, 14, 2, 13, 9, -2, 12, 16, 12, -6, -12, 6, -3, -5, 7, 1, -15, 4, 9, 13, 15, -5, -1, 7	1.2sec

We can see that even the number of games becomes 10000, we can easily get the result in roughly one second. For the correctness, we compared the result of output and they are the same as the answer which MiniZinc has given us.

DISCUSSION

The reason why MiniZinc is worse than the solver we write is that MiniZinc might use brute force to try every possible solution until it runs out of search space. **However, our solver has more information about how to use token like it needs to reserve some tokens to the games which have more fun from the first game until the last game**, rather than blind trying. We cannot use the same idea applying to other CSP like pizza or logistic problems. If we want to solve pizza game, we need to think of another algorithm that totally different from sequential games. **MiniZinc might be suitable to solve more complicated problems like pizza, logistic or block queens problem. Moreover, MiniZinc can quickly let you write down some ideas and test the correctness.** That is the high-level language can not do. A simple constraint might implement with much more complicated syntax by using high-level programming language.

Notice: if you would like to run the solver implemented with c++, please see appendix.

QUESTION IV

In the previous question, we found out that the limitation of MiniZinc can only solve small size instance problems by using chuffed solve configuration. And we write a c++ solver can solve the sequential game in polynomial time. Here comes an interesting question. **How far MiniZinc can reach the optimal solution in a given time?** In other words, **MiniZinc can find a lower bound of the total fun. But how "lower" it is?**

ANSWER

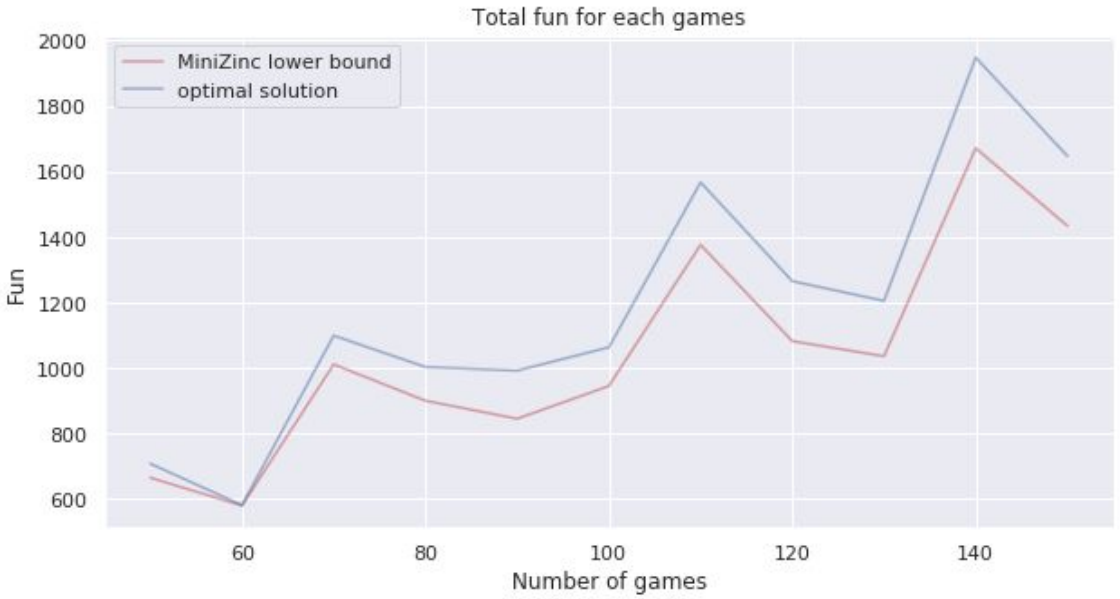
It depends on the number of games you solve. The number of games MiniZinc solves, the harder MiniZinc can reach the optimal solution.

EXPERIMENT

We will run the same problem instance and use c++ solver to find the optimal solution. And use MiniZinc to find a solution in a given time.

data	MiniZinc solving time	MiniZinc output	solver output
lower_bound_compare_solver 50	1m12s	665	708
lower_bound_compare_solver 60	1m16s	580	581
lower_bound_compare_solver 70	26s731	1011	1099
lower_bound_compare_solver 80	27s882	900	1003
lower_bound_compare_solver 90	26s621	845	992
lower_bound_compare_solver 100	29s922	945	1063
lower_bound_compare_solver 110	18s81	1376	1567
lower_bound_compare_solver 120	27s432	1082	1265
lower_bound_compare_solver 130	49s237	1036	1205
lower_bound_compare_solver 140	31s98	1671	1948
lower_bound_compare_solver 150	49s790	1434	1647

We use the size of the instance from 50 to 150 because MiniZinc can not solve the instance's size greater than 45. **You might be curious why we solve problems in a different given time. We collect data like this way. If in a long time, there is not a significant change of total fun(MiniZinc stuck to find the next answer), we stop immediately.**



We can see that as the number of games increases, the gap between MiniZinc output and the optimal solution becomes larger.

However, in general, MiniZinc can solve the problem well. We can stop MiniZinc somewhere and get the total fun which reasonably approaches to the optimal solution.

QUESTION IIV

Can we write a constraint to deal with the maximum fun games?

ANSWER

We just let MiniZinc play the funniest games as many as the number of tokens it has. In other words, let MiniZinc play the funniest games without hesitation. It doesn't need to think about other games. It just ran out of money on that game.

EXPERIMENT

We just add one more constraint into MiniZinc.

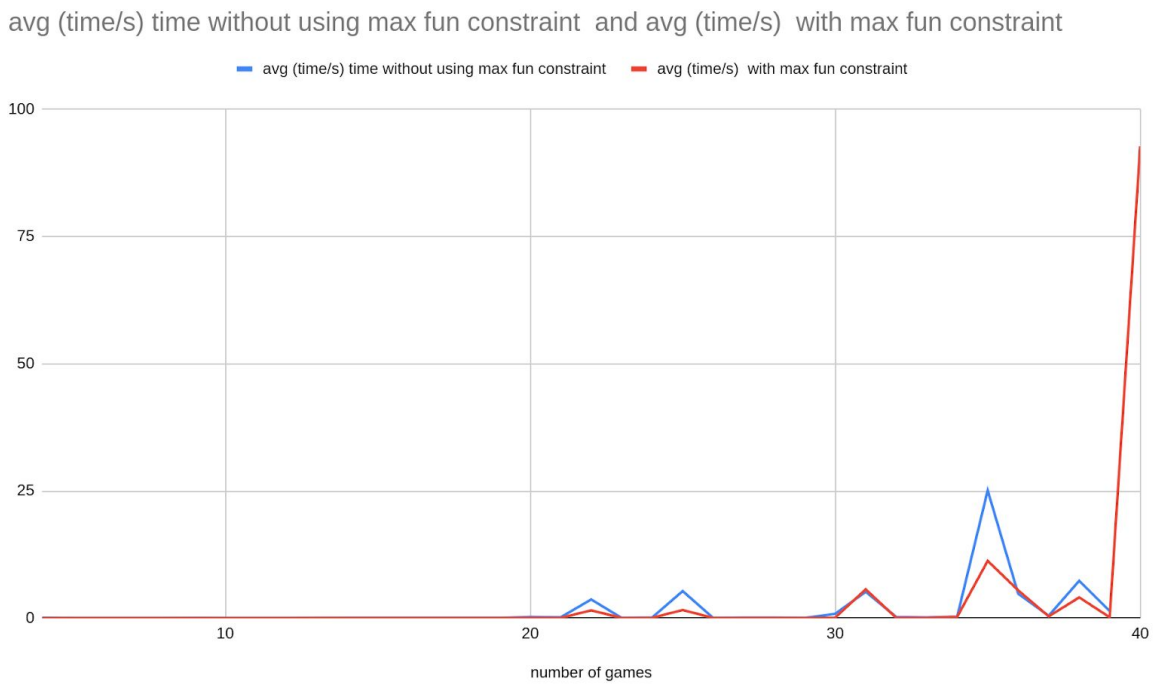
% Find a maximum games.
% If the most fun game have positive fun, then we just play cap times.
constraint forall (i in games) (((fun[i] = max(fun)) /\ fun[i] > 0) -> plays[i] = t[i]);

Here we need to be concerned about another problem is that **if the funniest game has negative fun**, we should ignore it and let negative constraint to deal with it.

The next step is running data by MiniZinc_data_runner.py

data	capacity	refill	fun	avg (time/s)	total fun
max_fun_constraint 4	5	2	[14,-10,4,-11]	0.04	61
max_fun_constraint 5	5	2	[9,-2,15,-12,4]	0.04	100
max_fun_constraint 6	5	2	[-19,-6,-14,-15,8,-9]	0.04	-23
max_fun_constraint 7	5	2	[15,1,-8,-17,4,-20,10]	0.04	93
max_fun_constraint 8	5	2	[-14,12,6,-15,-6,-9,-1,-6]	0.04	21
max_fun_constraint 9	5	2	[12,8,-17,9,6,6,3,-3,-11]	0.04	103
max_fun_constraint 10	5	2	[11,-4,-8,-7,16,-4,14,-13,-11,1]	0.04	134
max_fun_constraint 11	5	2	[-17,15,-13,17,-4,-16,9,-4,-6,-20,-10]	0.04	76
max_fun_constraint 12	5	2	[18,-17,-18,13,-9,0,-18,0,-7,1,19,11]	0.04	192
max_fun_constraint 13	5	2	[19,-19,10,8,-19,-13,4,-1,15,-15,-10,-2,-4]	0.05	141
max_fun_constraint 14	5	2	[19,-2,0,-17,4,-15,0,9,15,-12,-15,15,-14,1]	0.05	192
max_fun_constraint 15	5	2	[-1,-10,2,-1,2,0,1,2,19,2,1,5,-3,12,-5]	0.05	167
max_fun_constraint 16	5	2	[19,19,-15,-9,-16,16,-13,14,-4,12,-6,1,13,8,-13,17]	0.05	349
max_fun_constraint 17	5	2	[-8,-12,2,-1,2,0,1,2,19,2,1,5,-3,12,-5,11,-8]	0.05	183
max_fun_constraint 18	5	2	[8,-10,-10,-8,14,0,-16,9,-6,18,9,-16,-5,7,-6,12,16,-4]	0.06	273
max_fun_constraint 19	5	2	[-1,-5,-18 ,16,-5 ,0,4,18,6,-16,9,3,14,10,17,-1,0,-4,-17]	0.06	271
max_fun_constraint 20	5	2	[17,-8,19,17,-9,18,20,1,19,-12,1,3,-2,5,4,-13,-11,-2,-12,-7]	0.09	303
max_fun_constraint 21	5	2	[-16,-11,-18,-14,15,-16,-16,6,15,16,1,19,15,4,13,19,-15,7,-7,-16,-1]	0.08	273
max_fun_constraint 22	5	2	[-5,10,-20,14,12,13,11,1,19,6,-13,7,3,14,-14,10,13,-2,14,-3,-11,-10]	1.55	366
max_fun_constraint 23	5	2	[15,-11,-18,-10,0,11,18,11,-7,-15,0,-19,19,7,-17,-5,17,-20,-16,-13,-12,-13,19]	0.06	315
max_fun_constraint 24	5	2	[-10,10,11,5,-17,14,5,-4,4,4,0,19,-17,-4,8,-10,-10,-14,-20,-2,12,-13,7,8]	0.11	266
max_fun_constraint 25	5	2	[-3,-15,19,9,-4,-12,3,5,-4,10,3,8,-12,10,14,6,9,2,17,14,12,-16,-3,0,6]	1.61	397
max_fun_constraint 26	5	2	[-4, 1, 1, -19, 16, -20, 5, -20, 9, 0, -5, -20, 16, -18, 12, 11, 17, -11, -5, 16, -5, -13, -9, -2, -8, -12]	0.07	216
max_fun_constraint 27	5	2	[-13, 11, 17, -11, -6, 19, -4, 18, 0, -16, 6, -20, 17, 2, -17, -18, 9, 9, 4, -14, -12, -1, -8, 0, 10, 17, -7]	0.08	367
max_fun_constraint 28	5	2	[5, -10, -12, -6, 7, 14, -8, 5, 14, 12, -8, -7, 9, -10, 3, -15, -17, -1, 1, 0, 7, -5, -14, 3, 9, 2, -4, 10]	0.07	251
max_fun_constraint 29	5	2	[17, -4, 7, -20, -8, -11, -10, 1, -1, 11, 6, 2, -14, -3, -7, -19, 0, -5, -20, 5, 16, -18, -17, -3, 1, 2, 6, -13, -18]	0.07	115
max_fun_constraint 30	5	2	[14, -4, -9, 20, 2, 20, -2, -7, 8, 1, 7, 2, -11, -8, 17, 9, -4, -7, -6, 2, 8, -19, 13, 5, 3, 6, 0, -12, -16, 5]	0.14	422
max_fun_constraint 31	5	2	[-6, -1, 18, 9, 15, -6, -2, -16, 16, -5, 0, 3, 8, -20, -13, -19, 11, 3, 11, 19, 2, -8, -18, 14, 9, 12, 4, -5, 8, 14, 1	5.68	543
max_fun_constraint 32	5	2	[-7, -2, 12, -16, 13, -2, -16, 18, 11, 3, -18, -5, 11, 3, 7, 6, -13, -17, -13, 13, 5, 2, -18, -11, 18, 8, -15, -18, -	0.1	276
max_fun_constraint 33	5	2	[-1, -17, 5, 4, -3, -18, 16, -12, -17, -17, -14, -14, 0, -16, -15, 20, 3, 4, 8, 4, 5, -6, -6, 7, -3, -13, 12, 11, 5, -1	0.11	163
max_fun_constraint 34	5	2	[-7, 13, -12, -6, -2, 15, -19, -1, 18, 3, 18, -5, 8, -15, -11, -19, 1, 9, 3, 4, -10, 15, -11, 12, -4, -20, 14, -11, -1	0.28	486
max_fun_constraint 35	5	2	[0, 14, -7, 2, 10, 4, -19, 14, 6, 14, 5, -11, -4, 9, 2, 3, -7, 9, 0, 14, 17, 20, -8, 1, -14, 20, 8, -15, -19, -16, -16	11.27	497
max_fun_constraint 36	5	2	[-19, 8, -3, -3, 4, -13, -12, 11, 16, 8, 12, 7, 20, -16, -7, -5, 1, -6, 3, -6, -16, 14, 8, -9, 19, 6, -18, 19, -18, -1	5.5	501
max_fun_constraint 37	5	2	[-20, -4, 3, 9, -9, -12, -19, 2, 7, 6, -20, 1, 3, 19, -11, 11, -1, 3, -13, -9, -1, 7, -2, -1, 4, -14, -5, -8, 20, -12, 3	0.37	385
max_fun_constraint 38	5	2	[-17, -5, -13, 1, 14, -5, -19, -4, -11, -15, 9, 12, 19, 16, 6, 0, -6, 15, -1, 16, 14, 10, -8, -6, -13, -20, 6, -16, 1	4.1	473
max_fun_constraint 39	5	2	[-20, -15, 17, 4, 4, 15, 3, -6, -6, -16, -5, -14, -16, -12, 17, -1, -14, -4, 0, -10, 20, 19, 16, -8, -19, -11, 1, 18,	0.2	494
max_fun_constraint 40	5	2	[-7, 16, 13, -18, -6, -18, 12, 4, 6, 4, 16, -7, 11, 15, -6, -18, -7, -17, -1, 11, 10, -5, -19, 0, -7, 15, -3, 14, -5, -	92.74	595

And here is what we get by Chuffed. Comparing to not add this constraint, we can solve a number of games at 40 in practically 90 seconds.



We can clearly see that there is no difference between the number of games 4 and 22. For some games which Chuffed hard to solve, adding the max fun constraint makes them easier to solve. It looks like the constraint makes the hill more flat in the graph.

However, if we don't add **fun[i] > 0** and we expect the negative constraint can arrangement this situation that all the games have negative fun. What will happen? Hence, we generator another data file called max_negative_mix.dzn by hand.

Two constraints look like this:
constraint forall (i in games) (((fun[i] = max(fun))) -> plays[i] = t[i]);
constraint forall (i in games) (fun[i] < 1 -> plays[i] = 1);

WARNING: model inconsistency detected
Running Sequential_Games.mzn
=====UNSATISFIABLE=====
% Top level failure!
Finished in 102msec

We will get the UNSAT announcement. Thus we can not say that the max fun game play as many as you can but if you are negative, just play it once. Apparently, they have somewhere in conflict.

Although we can not write a logic proof to prove the correctness of this constraint. But after comparing the result with previous'. We can assume for small instances. Adding this constraint will not deprive the correctness of the program.

Appendix:

basic_data folder:

- **data_generator.py**: if you use python3 IDLE, please open it in IDE and click F5 to run. If you use command line, please enter python data_generator.py**.** And then **input how many number of games you want to generate**. Please pay attention that it is **weak to deal with bad use input**. Just enter the exact number of games you want to generate. And the default capacity is 5, refill is 2. About the fun array, it generates by random.randint(-20, 20).
- **basicData_num_*.dzn**: the postfix of the file is number of games. All the data files are generated by **data_generator.py. If you would like to run these data, please drag and drop to the MiniZinc IDLE then click command + R(mac os) or ctrl + R(windows)**. And MiniZinc will automatically recognize these data.
- max_negative_mix.dzn: to test the when all the fun are negative. Generate by hand

MiniZinc_data_runner.py:

- Automatically run all the data file in the data folder.
- If you use python3 IDLE, please open it in IDE and click F5 to run. If you use command line, please enter python MiniZinc_data_runner.py**. And input how many number of games you want to run.**
- **Please pay attention that the maximum number you can enter is 45.**
- **The default setting is that it will run Chuffed configuration.**

Project_data_analysis.ipynb:

- **Please open it in jupyter-notebook if you want to see the plot in detail.**
- They are basically just plot graphs.

Sequential_Games.mzn:

- Main file you would probably run. Please open it in MiniZinc IDLE and run as usual.

solver folder:

- **solver.cpp:** If you would like to run the solver.cpp, **Please enter g++ solver.cpp in command line and then run ./a.out** and then enter how many games you want to run. Please enter the number from the following: 5, 20, 30, 100, 500, 1000, 5000, 10000. If you would like to run different number, please go data folder and use data_generator.py to create a new instance.
- **data_generator.py:** basically as the data_generator.py described above**.**

If you have any questions to run the program, please email me.