CMPT417 Project Report

LP/CP Programming Contest 2015
Sequential Games

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, they are obtained by hand. 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 generate by random number between -20 and 20. Here are the website we use https://www.random.org/. 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];
```

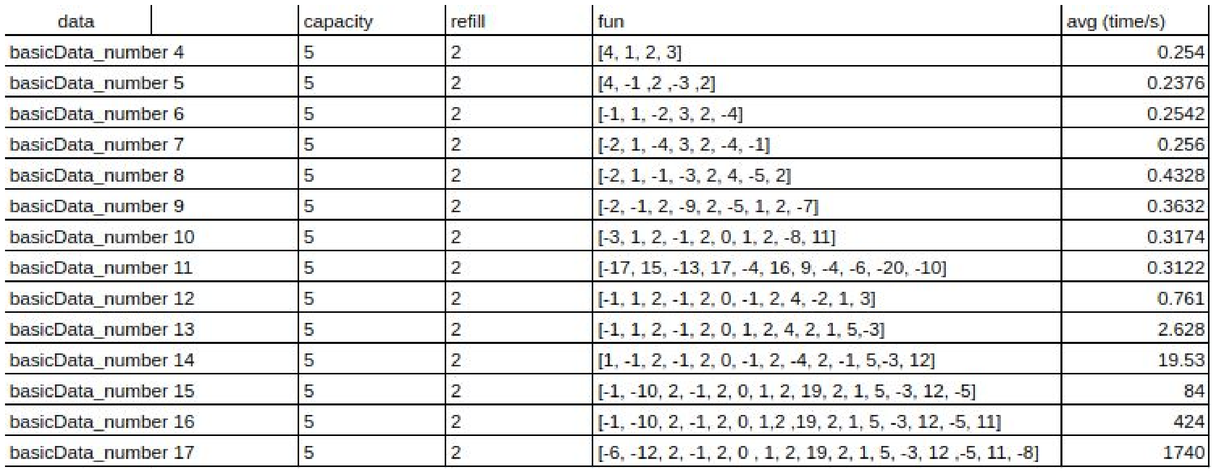
And use this to test our problem

```
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
----------
==========
```

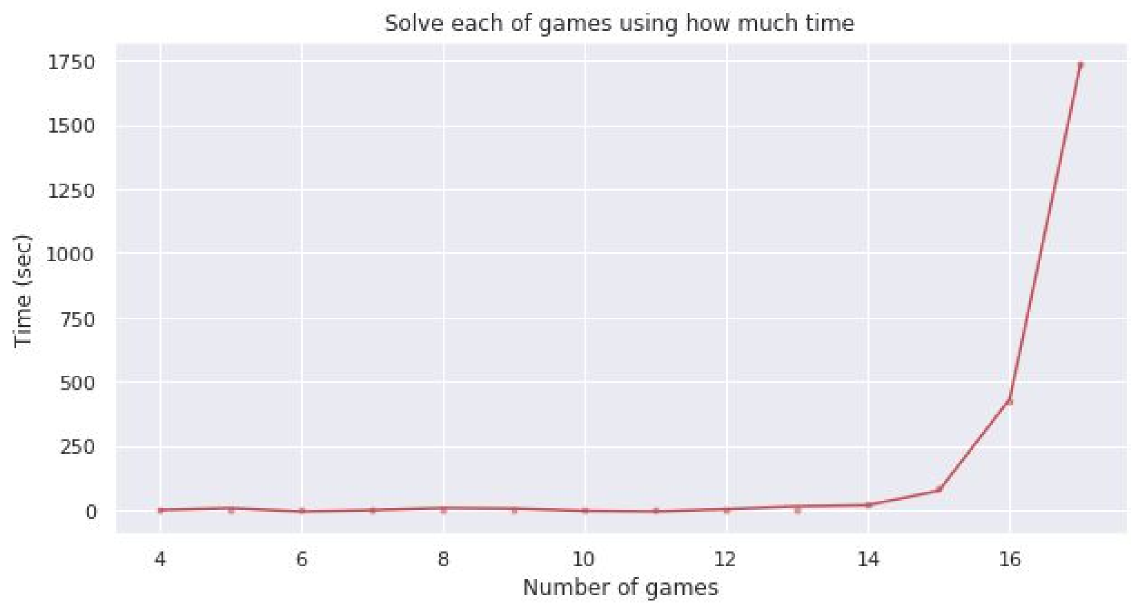
The dot line indicates that here are one solution and equality line shows that no other solution **which means here are the optimal solution**. Let’s show the correctness of this test. At the beginning of the game, we have the maximum tokens which is the capacity. Because the first game are the most 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(we will do in the partII.)



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

Part II ……