CMPT417---Intelligent Systems

Report of Sequential Games

Presented by:

Zelin Han---301309342

Dec. 2Nd, 2019

Introduction statement:

This is the report for solving the Sequential Games by MiniZinc for the project of CMPT417 in Simon Fraser University. All parameters of MiniZinc is used as default setting. I do the decision and optimazation version of this problem.

Problem specification:

For this project, the Sequential Games is letting player plays a sequence of games under 5 conditions. The first condition is player may play each game multiple times, but all plays of the same game must be made after the previous one and before the next one. The second condition is that player pays one token each time for every play of a game, and player can play a game many times as long as he or she has a token to pay. The third condition is that player must play each game at least once. The fourth condition is that player has a given number(C) of tokens when he or she start playing the first game, and he or she will receive a “refill” of a given number(R) of tokens after finishing the current game and before starting the next game. But player has a “pocket capacity” (C) which is the same as the initial number of tokens given to the player, and any extra tokens exceeding the capacity will be wasted. The final condition is that each game has a “fun” value which may be negative for player. The goal of this project is to decide how many times to play each game and maximize player’s total fun.

Methodologies:

The first version I did for this project is following the three constraints on the number of tokens that player has (t) and the number of times that player plays for each game (p), which given by the professor. First time, I define the number(num) of games, the set of numbers of games(games) which form 1 to num, an array of fun(fun) for each game, the pocket capacity(cap), the refill amount(refill), a set of numbers of x(x_range) which from 0 to cap and the goal value(K). And then I create two array of various integer(t, plays) which represent the number of tokens that player has for each game and the number of times that player plays for each game. After these, I’m going to define the three constraints for “t” and “plays”. The first constraint is $\sum_{i \in [n]} P_i V_i \geq K$ and I write “*constraint sum(i in games) (plays[i] * fun[i]) >= K*” in MiniZinc. The second constraint is the number of tokens t_i available to play game i is C(cap) when we start playing the first game, and for i > 1 is the minimum of C(cap) and $t_{i-1} - p_{i-1} + R$, and I write “*constraint (t[1] = cap) /\ forall(i in games) ((1<i /\ i<=num) -> exists(x in x_range) ((x = (t[i-1] - plays[i-1] + refill)) /\ ((x > cap) -> (t[i] = cap)) /\ ((x <= cap) -> (t[i] = x))))*” in MiniZinc. The third constraint is player need to paly each game at least once and at most t_i times, and I write “*constraint forall (i in games) ((1 <= i /\ i <= num) -> (1 <= plays[i] /\ plays[i] <= t[i]))*” in MiniZinc. Finally, I write “*solve maximize sum(i in games)(plays[i] * fun[i])*”, “*output [show (sum(i in games) (plays[i] * fun[i]))]*” and comment the first constraint to get the optimal solution. The reason why I solve for optimal solution not goal value(K) is that optimal solution is fixed for each test no matter what goal value(K) is and it will always return the maximum fun value without unsatisfiable case which is easier to test the code. But I also do every test for satisfiable problem for goal value(K), and I will give the test result for satisfiable problem. Here is the decision and optimization for optimal solution. I create 14 test cases to test my code. It passes the first few test cases including the three cases that provided on the webpage. But when I test the eighth test cases, the result of optimal solution doesn't look right. It returns 4 as the optimal solution for the input which are num=10, cap=6, refill=6 and fun=[-4,1,2,-3,1,2,-4,5,1,-2]. I show the "plays" array for debugging and I find that the optimal solution plays each game for 6 times except plays the last game 1 time. This is definitely a false result, since it's obviously that player should only play once when the fun value is negative. Thus, I went back to looking for mistakes and loopholes in my code, and finally I found that the x_range is defined mistakenly. The x_range should be from 0 to (cap+refill), since "*x = (t[i-1] - plays[i-1] + refill)*" which x can equal to (cap-1+refill) and it much larger than the x_range I define in this test case. Hence, I redefined x_range as (cap+refill). After that, the result of optimal solution is 59 which is right for the test case. But another problem is that the program runs for 15s 893msec which is so slow, and I optimize it to run faster.

The second version of my code is optimized from the first one to run faster. Like I mentioned previously, it's obviously that the optimal solution for player is to play the game with negative fun value only once. Therefore, I add the fourth constraint which is "*constraint forall (i in games) (fun[i] < 1 -> plays[i] = 1)*" to let the plays[i] for game i automatically equal to 1. This improvement greatly reduces the running time which is reflected in the fact that running the same test case (test_case8) it only takes 161msec. This change improves nearly 100 times of running time.

Testing:

I do 14 test case for my program and here is the test result:

	num	cap	refill	fun[]	K	max_fun	Time1	Time2	SAT
Test1	4	5	2	[4,1,2,3]	25	35	125	143	sat
Test2	4	5	2	[4,-1,-2,3]	25	29	128	145	sat
Test3	5	3	2	[4,1,-2,3,4]	20	30	129	142	sat
Test4	6	4	2	[4,1,2,-3,-2,1]	25	22	134	148	not sat
Test5	7	5	4	[4,1,2,-3,2,3,-5]	40	48	159	152	sat
Test6	8	8	2	[-3,2,1,5,-3,6,2,-1]	50	67	159	144	sat
Test7	8	8	6	[3,7,-4,2,-5,-3,4,8]	110	166	1406	178	sat
Test8	10	6	6	[-4,1,2,-3,1,2,-4,5,1,-2]	60	59	15893	161	not sat
Test9	10	8	5	[-5,2,1,-3,6,3,8,-5,-1,3]	200	149	6499	178	not sat
Test10	10	6	6	[6,-3,2,4,3,-5,-9,3,4,-6]	150	109	1918	271	not sat
Test11	12	7	3	[3,2,5,-4,6,-7,-3,5,4,2,1,4]	150	122	>1.8e6	472	not sat
Test12	11	9	4	[3,2,-4,2,-3,4,-6,5,-8,3,-9]	150	109	9960	236	not sat
Test13	10	6	4	[2,1,2,3,4,2,4,5,2,1]	100	120	4517	4487	sat
Test14	10	7	4	[3,1,1,2,6,3,4,5,3,2]	100	150	11603	11288	sat

Note: max_fun is the optimal solution, time1 is the running time(msec) for version1, time2 is for version2.

The reason of all test cases I choose is to look for factors that affect the running time of the two versions of program by changing different parameters. According to the table above, we can see that from test1 to test6 increase num or cap with small refill will not have a big impact on running time. But with large num and cap, increasing refill will also increase the running time of the first version but not effect the running time of second version, according to the test7. According to the comparison of test7 and test 8, increasing num but decreasing cap with same refill will increase the running time for the first version. And also, from test11, when I increase the num to 12, the first version of program takes more than 30min without any result. Thus, we may can say that num has a greater effect on the running time than cap for the first version. From test13 and test14, we can see that the running time of second version has a great increase when I use only positive fun value with large num and cap.