CMPT 417 – Intelligent Systems

Project Report: Sequential Games

Presented by:

Bowen Wang – 301267523

Dec. 9th, 2019

Table of Content

Introduction

This project aims to solving the Sequential Games problem given in *LP/CP Programming Contest 2015.* There are *n* game booths G1, G2, ..., Gn from one end of the street to the other. A player wants to play all of the games in the order from G1 to Gn without skipping any. The goal is with some rules of game playing, help the player manage his tokens so that he can gain total fun at least K. Solving for solution that can exceed K is a decision problem, but it makes more sense to solve for maximum of total fun can be attain with certain setting of the playground. As we are looking for the sequence of game playing with maximum total fun, the problem is an optimization problem combine with decision problem.

In Part I, I designed vocabularies and find satisfy instances to original Sequential Games problem with the default setting of MiniZinc IDE. Giving several test cases to ensure the constraints do solve the problem and record the solving time of each input instance.

In Part II, I'm going to try different solver in MiniZinc with different input size/value to see if there is any relationship between the input and solver's solving time. Moreover, as the problem can also be implemented by algorithms, I develop a Python program to solve the optimization version of the problem and compare the performance of MiniZinc solvers and Python codes.

Part I

Problem Specification

This problem is defined in course notes section 12:

you are to play a sequence of games G1, G2, . . ., Gn, under the following conditions:

1. 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.
2. 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.
3. You must play each game at least once.
4. You have C 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 R tokens. However, you have some "pocket capacity" C, and you are never allowed to have more then C tokens.
5. Each game has a "fun" value for you, which may be negative.

The goal is to decide how many times to play each game, so that the total fun can be maximized.

The problem can be described as:

Input Vocabulary [*num, fun, cap, refill,* K]**:**

– Number *num* ∈ N of games;
– Fun value *fun*[i] ∈ Z for each game i ∈ [n];
– Pocket Capacity *cap* ∈ N;
– Refill amount *refill* ∈ N;
– Goal K ∈ N.

Output Vocabulary [*num, fun, cap, refill,* K, *p, t*]**:**

–*p*[i] ∈ Z for each game i∈ Z, times of playing game i.
–*t*[i] ∈ Z for each game i∈ Z, number of tokens before playing game i

Goal: Find a number of plays *p*[i] for each game i ∈ N such that maximize the total fun if total fun is at least K.

The constraints on t and p are:

1. Total fun gained by playing all n games must be at least K

$$\sum_{i=1}^{n} \quad sum\,(i, i \in N, play(i) \cdot fun(i)) \geq K$$

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:

t(1)=C ∧ ∀i[1<i≤n→∃x((x=t(i−1)−p(i−1)+R)∧
(x > C → t(i) = C) ∧ (x ≤ C → t(i) = x))]

3. We play each game at least once, and at most t[i] times:

∀i[(1 < i ≤ n) → (1 ≤ p(i) ≤ t(i))]

Testing

The constraints are implemented in *MiniZinc* file *Sequential_Games.mzn* with 14 test cases named *test_case1.dzn* to *test_case14.dzn,* details of each test is shown in Appendix I. All the testing in this project is under default setting and the records of running each test are added to Appendix II: Stat of Each Test.

The test case 1-3 is testing the functionality and correctness of the code with small set of positive and negative values of input. Then, I designed test 4-7 with identical input value changed to find the relationship between input values and satisfiability. Lastly, test 8-14 are testing the running time of the solver when solving problem with different refill amount to find out whether there is a relationship between refill amount and solving time. While doing test 8-14, I find there is a significant increase on running time when

increase refill amount from 2 to 3. After that, the running time trend to be linearly increasing. This is the interesting observation I find in part 1 of this project.

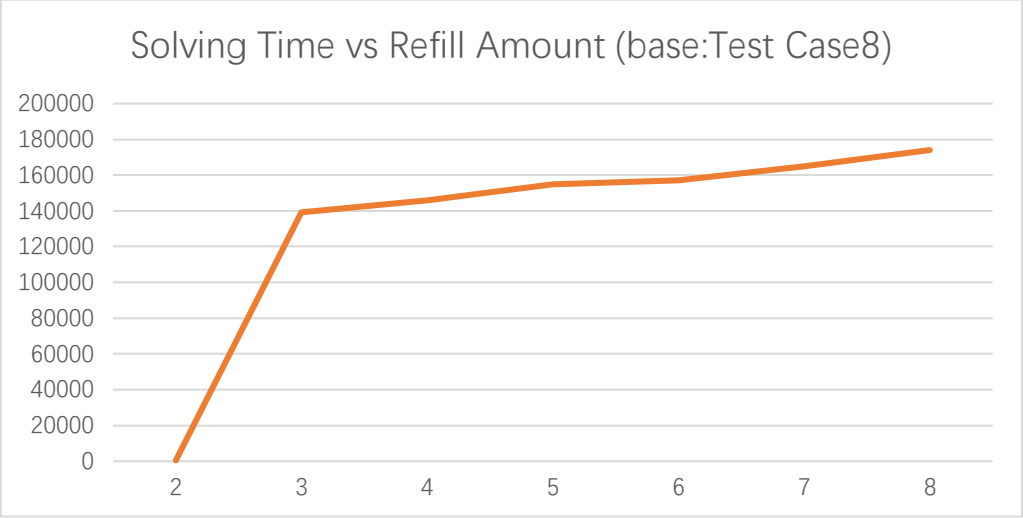

Figure I: Interesting Fact from Part I

Part II

Question Exploring

1. Are there any other constraints can be added to help solver solve faster? Is there a difference between different solver packaged in MiniZinc?

   The problem specification from Part I gives a basic structure to solve the problem, however, it is mentioned in class that more proper constraints can reduce the solving time by restricting searching space. In Part II of the report, I'm going to add more constraints to help MiniZinc solve faster, and use different solvers packaged in MiniZinc solve same structure to find if there is any difference on solver's solving time.
2. Can we come up with a specific algorithm solve this problem that can solve the problem faster than MiniZinc?

   During the test on solving time, I find there are limited instances can be solved by packaged solver due to instances size. For example, when the number of games greater than 30, it takes the solver more than 5 minutes to come up with a satisfiable assignment, which is not efficient. Considering the solver can be based on Brute-Force search algorithm, a better algorithm specifically for this problem can run more efficiently and more reliable for further property findings.

3. What are the relationships between each element in input instances and running time of the new designed algorithm?

   From the observation on solving time at the end of part I, I believe there are relationships between instances' value and solving time. By the improvement on performance, the test can be implemented in a larger scale. For this question, I'm going to find the relationship between number of games, refill amount, capacity of pocket and running time of my algorithm.

Improve Problem Specification

When reviewing the previous problem specification and simulate it in real-world scenario, I find there is several specifications can be added into our structure.

Constraint 4: If a game has non-positive fun value (ie. the player can't gain fun from it), only play this game once to satisfy the 3rd constraint.

∀(i in 1..num) ((fun[i] < 1) ->(p[i] = 1))

Constraint 5: If the current game has the maximal fun value comparing to all the following games, player spend all the tokens in the pocket to play current game as it can be regarded as a local maximum similar to greedy algorithm.

∀ (i in 1..(num-1)) ((∀(j in (i+1)..num)((fun[i]>=fun[j])/\fun[i]>0)) -> (p[i] = t[i]))

The correctness of constraint 4 is trivial, as $total\_fun = \sum\ p(i) * fun(i)$, if $fun(i) \leq 0$, the more $p(i)$, the less $total\ fun$. For constraint 5, we suppose $fun(i)$ is greater than any following $fun(k)$ $(i < k < num)$, t(i) < p(i). After playing i^{th} game, t(i) – p(i) token(s) has been saved for following games, as $fun(i) > all\ fun(k)$, the gap of fun gain from the saved token is $fun(i) - fun(k)$, which is a loss to maximize $total\_fun$. Contradict to our goal. Therefore, the local maximum equals global maximum.

MiniZinc Performance Comparison

In order to compare the performance of different solve condition, I designed *Data_Generator.py* to create 15 test data with game set size 1-15, fun value of each game assigned between -6 and 6 randomly. Use different solvers ("Gecode", "Chuffed") with different constraints (constraints used in Part I and constraints used in Part I plus the 2 constraints descript above) to solve each test 10 times and record the average solving time. All the raw data are attached in Appendix.

After adding the new constraints into MiniZinc script, and using "Gecode" solver, the performance has a significant improvement (Figure II). As 2 new constraints have effect on limiting the search space of MiniZinc solver, the 4^{th} constraint cut off all negative and

zero fun value games and the 5th constraint has a chance to determine token spend on local max fun value games (best case: if the game set coincidently sorted in decreasing order, the search space can be reduced to constant for each game. Worst case: if the game set coincidently sorted in increasing order, the search space keep the same)

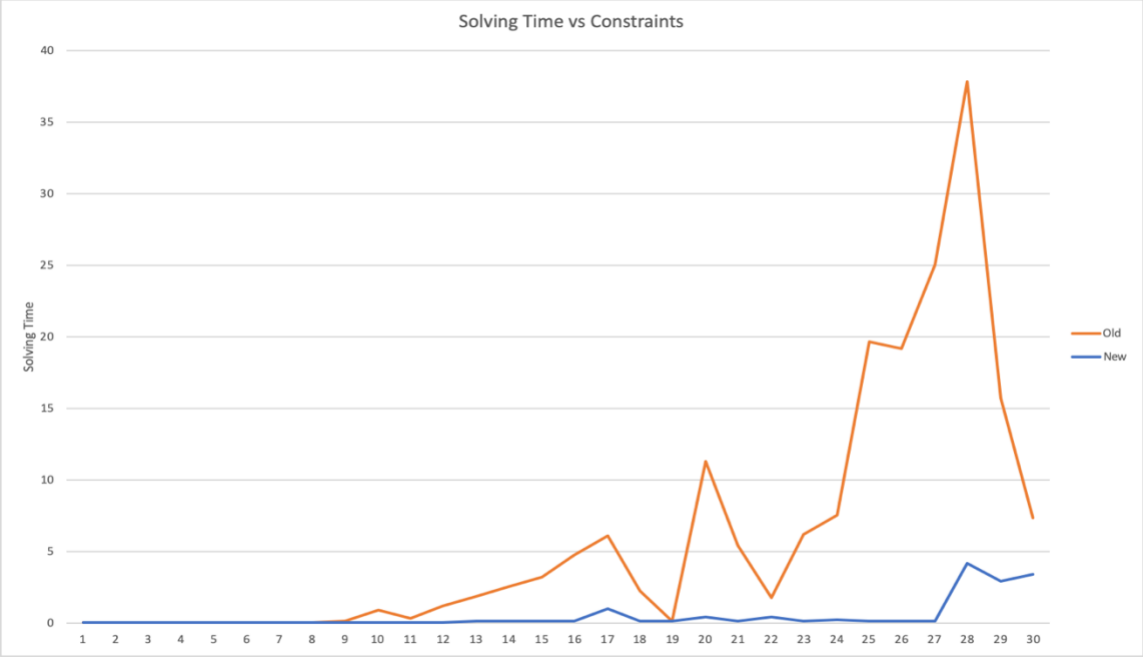

Figure II: New Constraints Helps to Solve Faster

Besides adding constraints, changing solver can also have a significant improvement on solving time. To examine the difference between solvers, I choose “Chuffed” solver to and MiniZinc’s default “Gecode” solve new constraints and record the solving time (Figure III). From the figure, I find there are 2 peek on Gecode Solver at num = 25 and 29. After I review the *fun* sets, both set has max fun at the end of the set, which is almost the worst case of the 5th constraint. That makes the speed up by adding constraints failed. But surprisingly, “Chuffed” solver deal with this senario very well, that indicates the solver optimiaztion is beyound the constraints optimazation.

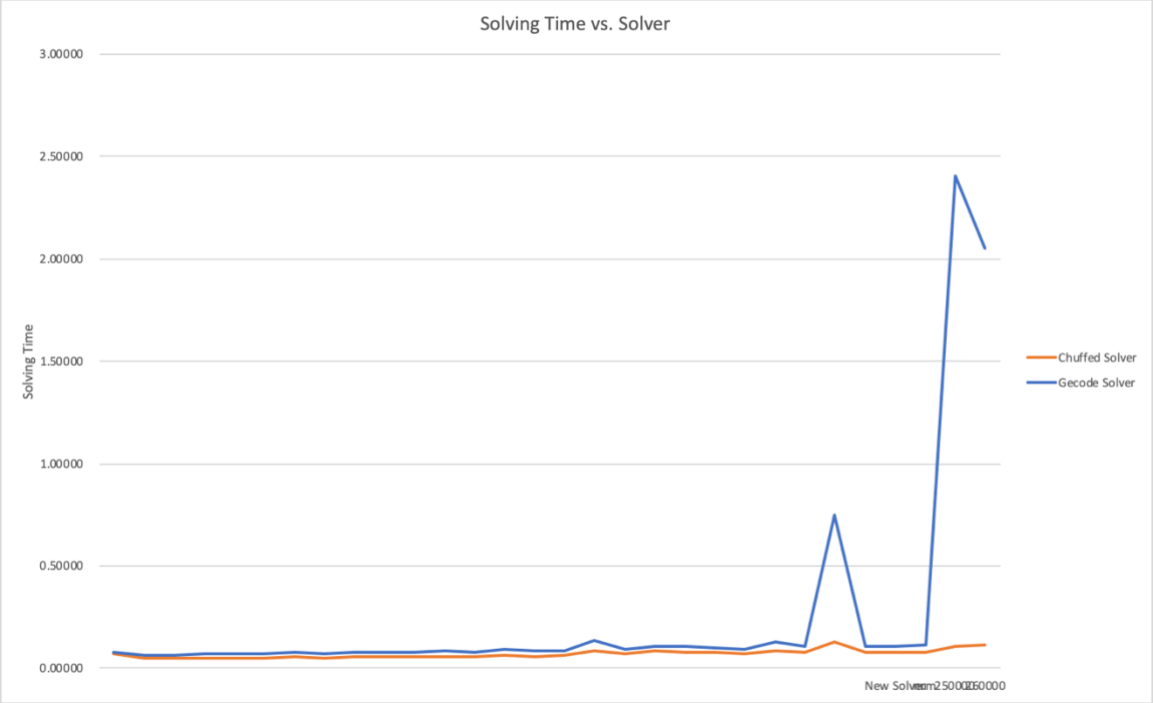

Figure III: Choice of Solver Have Effect on Solving Speed

Python Algorithm

The solver itself is an organized algorithm, also it can be regarded as a universal Turing Machine that can simulate any logic problems. However, simulation always require more resources to do the same task. If we can design identical program to solve the problem, the running time ideally can have significant reduction. Afterwards, we can solve the problem in a larger scale of input and find more feature on each input element and its relation to running time.

For each game in the set, we calculate its *p(i)* rather than guess. Recap the local maximum algorithm, how many times we can play a game depends on whether there is a higher fun value game come after that and before the *cap* been refilled. If the *cap* can be refilled, we use all the token in pocket on i^{th} game, otherwise, we save some tokens to ensure we can have maximum token for the upcoming local max fun value game.

The Pseudocode can be written like:

```
for i in range (num):
        play[i] += 1                    ## every game play at least once
        cur_token -= 1                  ## token spend on first play
        if (fun[i] <= 0):               ## check if fun(i) non-positive
            max_fun += fun[i]           ## if non-positive, switch to next game and
                                        refresh the total fun
            cur_token = min(cur_token+refill, cap) ## refresh token
        else:
            future_token = refill       ## token can refill before the game to check
            next_game = i+1             ## next game index
            reserve = 0                 ## number of token save before switching to next
            while (future_token < cap):                 ##
                if (next_game<num):                     ## Check every game before
                    if (fun[next_game] > fun[i]):       ## refilling cap of tokens, if
                        reserve = cap - future_token    ## a game with more fun
                        if (reserve > cur_token):       ## occur before refilling
                            reserve = cur_token         ## done, save some token
                        break                           ## so that cap tokens can be
                else:                                   ## refilled before playing
                    break                               ## that max fun game
                next_game += 1                          ##
                future_token += refill-1                ##
            play[i] += cur_token – reserve          ## calculate play times of i
            max_fun += play[i]*fun[i]               ## refresh max fun
            cur_token = min(reserve+refill, cap)    ## refresh token
```

Python Algorithm Performance & Observation

The performance of the python solver is extraordinarily good. It can handle over 100000 games problem in a second, which seems impossible to any MiniZinc solver. With the high performance, now I can try to figure out the relationship between each input elements.

Solving Time vs. num (Figure IV, Appendix V)

I created 30 tests for python algorithm with number of games increase from 25000 to 54000 with equal space. The fun value of each game is assigned randomly from -6 to 6. Refill 2 tokens when switching games and the capacity of pocket is 5 tokens.

The figure shows a linear-like relationship between number of games and solving time.

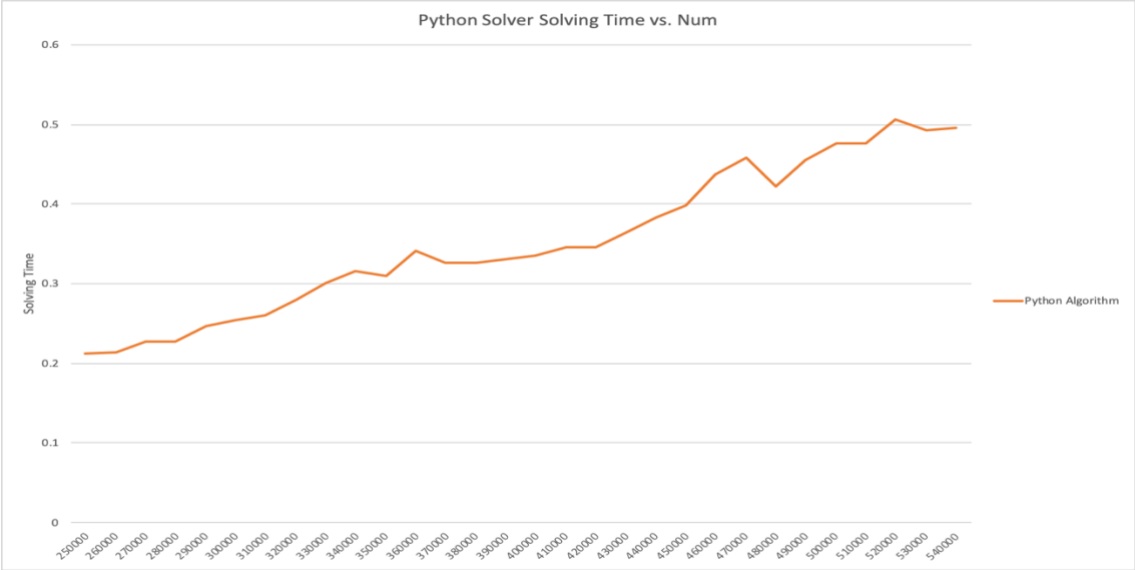

Figure IV: Linear-like Relationship Solving Time vs. Num

Solving Time vs. Refill (Figure V, Appendix VI)

I created 30 tests for python algorithm with number of refilling increases from 2 to 31 with equal space. The fun value of each game is assigned randomly from -6 to 6. 340000 games included in each test and the capacity of pocket is 5 tokens.

The figure shows a decrease in first few tests then remain stable after Refill = cap.

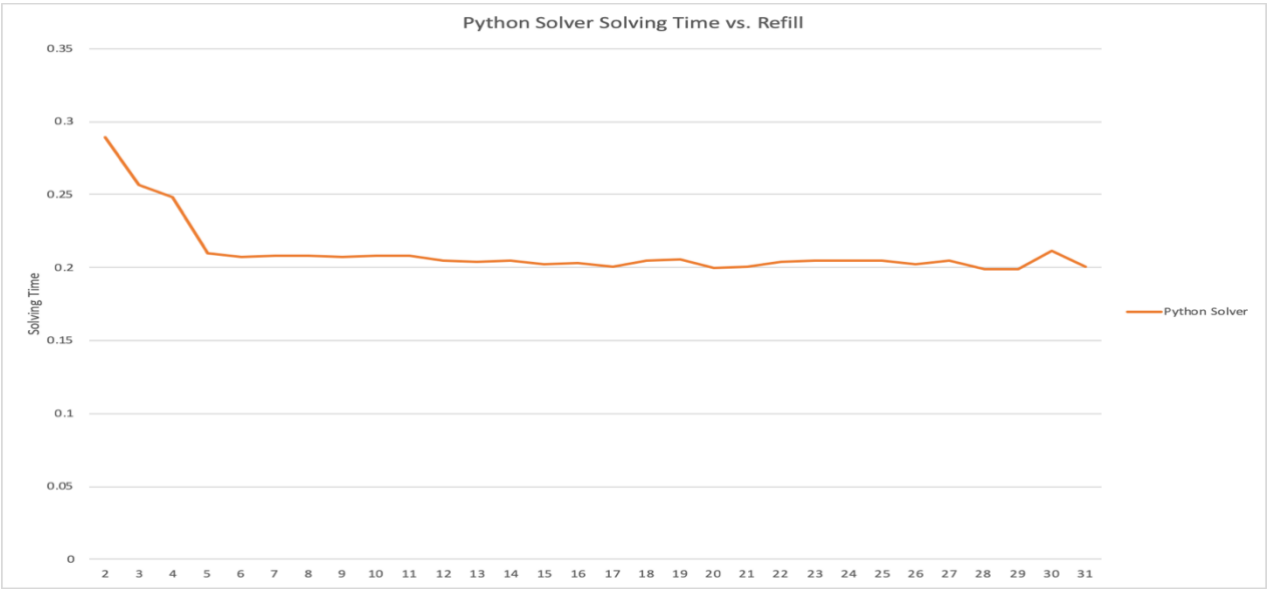

Figure V: Reduce then Stable

Solving Time vs. Cap (Figure VI, Appendix VII)

I created 30 tests for python algorithm with capacity of pocket increases from 2 to 31 with equal space. The fun value of each game is assigned randomly from -6 to 6. 340000 games included in each test and the refill after switching game is 2 tokens.

The solving time keeps increasing but the slope is reducing.

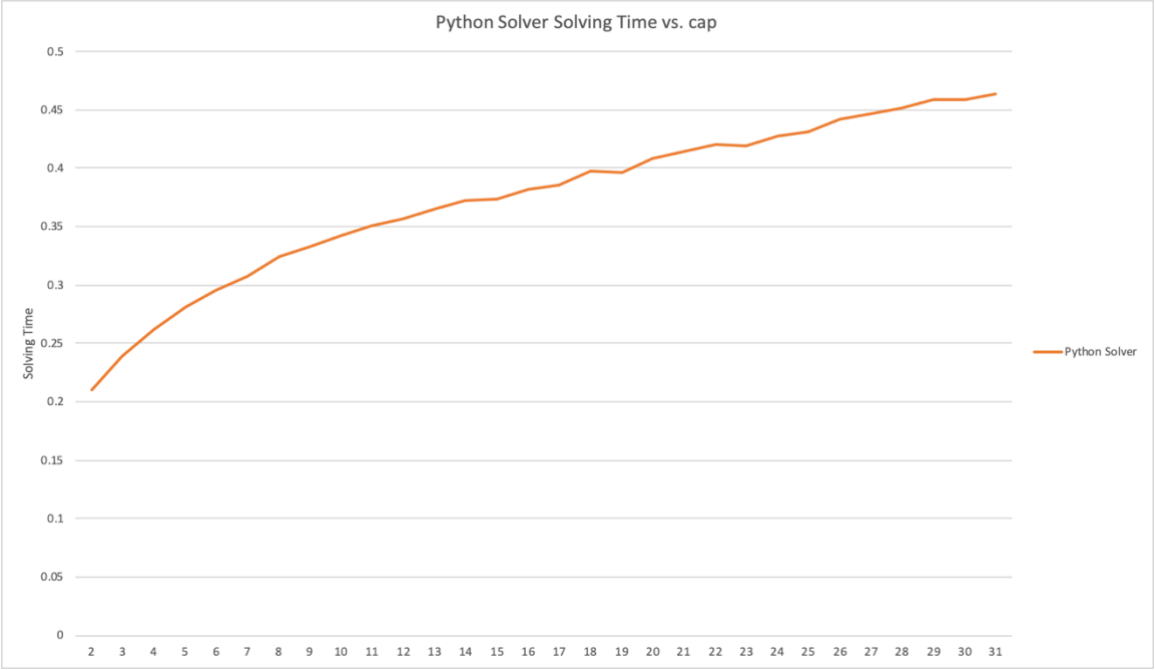

Figure VI: Keep Increasing with Reducing Slope

Analysis

Solving Time vs. Num

Number of games plays a significant role in problem solving. Recap our algorithm, every game in the set should be assigned a number of play times, which is O(n) complexity. Within each assignment, we are going through few games following current game, the number of games went through is roughly bounded by (cap/(refill-1)). If cap and refill are fixed, the bound is constant, which lead to the algorithm complexity is O(n). In Figure IV, we see a linear-like trend, which support the analysis

Solving Time vs. Refill (Figure V, Appendix VI)

Refill after each game switching controls the speed of pocket fill up, a larger refill amount lead to fill player's pocket faster. However, the capacity of player's pocket is limited, extra refill has no power to reduce search space of each game (just spend all the tokens, player's pocket will full charge when switch to next game). When the amount of

refill close to capacity, the solving time reaches its lower bound that if the game has positive fun value, spend all tokens on that then get refill for next game. In Figure V, the solving time reaches lower bound slightly above 0.2s.

An interesting observation is that when I choose refill = 1, the running time become very long. This can be explained by the upper bound discussed previously that (cap/(refill-1)), when refill = 1, cap/0 can be regarded as a very large number. At this time, the searching space is bounded by another factor which is the number of games. Therefore, under this worst case, the time complexity is $O(n^2)$, where n is number of games.

Solving Time vs. Cap (Figure VI, Appendix VII)

Capacity of players' pocket controls the speed of pocket fill up as well, a large capacity slows down the pocket to be filled up which enlarge the search space for times of play at the same time. But an extremely large capacity can lead to another scenario that for every game, there exist a following game has more fun. This becomes a upper bound of the effect on solving time that the capacity can generate. In Figure VI, the slope is reducing, and the curve tend to parallel with X-axis.

Conclusion

During the part I of this project, I learned to use MiniZinc IDE to build a logic structure to solve problem. The solver built in the logic can help solve both the decision and the optimization problem, which is easy to implement and test the logic itself with small portion of input.

In Part II, I added more constraints into the structure and find out proper constraints can speed up the solver by eliminate some assignments and reduce the searching space. In some ways, determinable constraints are similar to heuristic to an AI model, which give the agent a direction rather than doing random walk.

The algorithm that used by solvers packaged in MiniZinc is basically Brute-Force search with some heuristic added. The efficiency of Brute-Force search algorithm is very low, which makes it can't solve complex instances with limited resources. If willing to solve large portion of instances, a specific algorithm is more realistic and reliable.

For the optimal version of problem given in *LP/CP Programming Contest 2015,* there are several relationships between each instance element and solving time: A positive linear relationship exists between number of games and solving time, an inverse proportional relationship with a lower bound exists between refill amount and solving time, and a positive slope decreasing relationship with an upper bound exists between capacity of pocket and solving time.

Appendix

I. The detailed list of test instances is as below:

- Test1: Given in *LP/CP Programming Contest 2015*

  num = 4;
  cap = 5;
  refill = 2;
  fun = [4,1,2,3];
  K = 25;

- Test2: Given in *LP/CP Programming Contest 2015*

  num = 4;
  cap = 5;
  refill = 2;
  fun = [4,-1,-2,3];
  K = 25;

- Test3: *Similar to test 1, but change refill from 2 to 3*

  num = 4;
  cap = 5;
  refill = 3;
  fun = [4,1,2,3];
  K = 25;

- Test4: *add more games into the playground*

  num = 6;
  cap = 3;
  refill = 2;
  fun = [1,-2,9,0,-1,3];
  K = 20;

- Test5: *similar to test 4, change the order of negative fun value*

  num = 6;
  cap = 3;
  refill = 2;
  fun = [1,-2,-1,0,9,3];
  K = 20;

- Test6: *similar to test 4, change the value of K larger*

  num = 6;
  cap = 3;
  refill = 2;
  fun = [1,-2,9,0,-1,3];
  K = 45;

- Test7: *similar to test 1, with more negative fun values*

  num = 4;
  cap = 5;
  refill = 2;
  fun = [-4,1,-2,-3];
  K = 25;

- Test8: *test for run time and refill amount*

  num = 10;
  cap = 8;
  refill = 2;
  fun = [-2,1,2,-3,0,2,-4,5,1,-2];
  K = 79;

- Test9: *test for run time and refill amount*

  num = 10;
  cap = 8;
  refill = 3;
  fun = [-2,1,2,-3,0,2,-4,5,1,-2];
  K = 79;

- Test10: *test for run time and refill amount*

  num = 10;
  cap = 8;
  refill = 4;
  fun = [-2,1,2,-3,0,2,-4,5,1,-2];
  K = 79;

- Test11: *test for run time and refill amount*

  num = 10;
  cap = 8;
  refill = 5;
  fun = [-2,1,2,-3,0,2,-4,5,1,-2];
  K = 79;

- Test12: *test for run time and refill amount*

  num = 10;
  cap = 8;
  refill = 6;
  fun = [-2,1,2,-3,0,2,-4,5,1,-2];
  K = 79;

- Test13: *test for run time and refill amount*

  num = 10;
  cap = 8;
  refill = 7;
  fun = [-2,1,2,-3,0,2,-4,5,1,-2];
  K = 79;

- Test14: *test for run time and refill amount*

  num = 10;
  cap = 8;
  refill = 8;
  fun = [-2,1,2,-3,0,2,-4,5,1,-2];
  K = 79

	num	cap	refill	Max Fun	Min Fun	Neg. Fun	Pos. Fun	K	Run Time (ms)	Satisfiable	Max Total Fun
Test 1	4	5	2	4	1	0	4	25	66	Sat	35
Test 2	4	5	2	4	-1	2	2	25	68	Sat	29
Test 3	4	5	3	4	1	0	4	25	70	Sat	42
Test 4	6	3	2	9	-2	2	4	20	64	Sat	36
Test 5	6	3	2	9	-2	2	4	35	65	Sat	40
Test 6	6	3	2	9	-2	2	4	45	65	Un-Sat	41
Test 7	4	5	2	1	-4	3	1	25	61	Un-Sat	-4
Test 8	10	8	2	5	-4	4	6	79	378	Un-Sat	47
Test 9	10	8	3	5	-4	4	6	79	139000	Un-Sat	59
Test 10	10	8	4	5	-4	4	6	79	146000	Un-Sat	67
Test 11	10	8	5	5	-4	4	6	79	155000	Un-Sat	67
Test 12	10	8	6	5	-4	4	6	79	157000	Un-Sat	73
Test 13	10	8	7	5	-4	4	6	79	165000	Un-Sat	67
Test 14	10	8	8	5	-4	4	6	79	174000	Un-Sat	77

Appendix II: Stat of each test

Solving Time vs. Constraints					
num	cap	refill	fun	Old Constraints Solving Time	New Constraints Solving Time
1	5	2	[4]	0.06902957	0.087180376
2	5	2	[−5, 2]	0.068463469	0.069028807
3	5	2	[5, 0, 0]	0.073997402	0.070669746
4	5	2	[1, 0, −3, 0]	0.074605989	0.072666264
5	5	2	[1, 0, −2, 4, 0]	0.076234627	0.075763226
6	5	2	[6, −6, 4, 0, −6, −2]	0.078907204	0.072445011
7	5	2	[4, −2, −4, −1, −6, 6, −6]	0.082295752	0.080456781
8	5	2	[−5, 5, 6, 6, −1, −6, 3, 0]	0.089378357	0.081277895
9	5	2	[−5, 1, 1, −3, 3, −3, −1, −2, 3]	0.114954615	0.086970282
10	5	2	[−3, 6, 1, 3, 3, 3, 2, 2, 2, −2]	0.886340857	0.087312603
11	5	2	[−6, 4, 0, 5, −1, 0, 1, −3, −1, 3, −6]	0.316681623	0.092460251
12	5	2	[−6, −6, −5, −2, −6, 0, 6, 3, −4, 2, 6, 3]	1.238131762	0.092751408
13	5	2	[−6, −2, 2, −6, −4, 5, 0, −4, 5, 6, −2, −5, 1]	1.830163145	0.10203867
14	5	2	[−4, 3, 0, 1, 0, −1, −5, 4, 1, −6, 0, −1, −2, 3]	2.50934186	0.105514193
15	5	2	[−5, −4, 6, −2, 1, 5, 2, −4, 6, 1, −3, −3, −4, −4, 0]	3.184809923	0.10256834
16	5	2	[3, 4, 5, −5, 6, 6, −6, 6, −5, −3, 3, −3, 0, −4, 4, −4]	4.778085279	0.113082314
17	5	2	[−4, 3, 1, −1, 0, −5, 1, −5, 4, −3, −6, 0, 5, 1, 2, 5, 6]	6.093752575	0.976919556
18	5	2	[2, 2, −1, 2, 0, −6, −5, −5, −4, 5, 0, 6, −6, −5, 2, 6, −6, 0]	2.220759726	0.135017252
19	5	2	[−3, −5, −2, 2, −6, −1, −5, −1, −4, −5, −6, −1, 0, 3, −4, −6, −5, −4, 0]	0.147578239	0.127154779
20	5	2	[−1, 5, 2, 5, −1, 3, −5, 0, 1, 2, 0, −4, −1, −6, −4, 6, 3, 4, 0, 0]	11.3152494	0.410936928
21	5	2	[−6, −5, 5, −2, −5, 6, 4, −4, −1, −3, 6, 5, −2, 0, 6, 5, −2, 1, 4, 1, −4]	5.386826277	0.136270857
22	5	2	[−5, −6, −2, −3, −5, 2, −5, 3, −3, 1, −5, −6, −1, −5, 2, 0, 0, 4, 6, 1, 1, 6]	1.797477484	0.443943405
23	5	2	[6, 6, −1, −6, −5, 2, −6, 5, 5, 4, −5, 2, 2, −4, −4, −6, −5, −6, −2, −4, 0, −2, 0]	6.150270224	0.130625963
24	5	2	[6, −1, 2, 0, 0, −5, −1, 2, 2, 5, −5, 1, 0, −5, −1, 1, 2, 0, 5, −2, −4, 1, −6, −5]	7.511993265	0.210443926
25	5	2	[−5, −2, 0, 0, 6, −5, 2, −6, 0, −1, −5, 4, −2, −3, 0, −4, −2, 0, 2, −4, −5, 0, 1, −5, 3]	19.61439557	0.15947094
26	5	2	[−4, −5, 0, 6, 0, 6, 0, 6, 2, −3, −5, 3, −2, 0, −1, −6, −2, 3, 4, −4, −1, 6, 2, 4, −1, 3]	19.1698071	0.187074852
27	5	2	[−6, 3, 5, −6, −1, −1, −6, 5, −2, −3, 5, 4, 2, −4, −1, −2, 4, −5, −4, −3, 0, −2, 0, −2, −1, 1, 4]	25.01416879	0.172618055
28	5	2	[−5, −6, 2, −4, 2, 0, 5, 5, −2, 2, −4, −4, −3, 5, 2, 1, 1, −1, 4, 1, 0, 6, 3, −6, −1, 1, −3, 5]	37.8124589	4.202316809
29	5	2	[−5, −2, 4, −6, −1, 3, −1, 2, 4, 6, 0, 2, 5, −1, 2, −5, 3, 5, −3, −6, 6, 3, 3, −6, −3, −6, 1, −1, −3]	15.73503432	2.905288553
30	5	2	[−4, −2, 1, 3, −2, 0, 1, −3, −1, 1, 1, 2, −2, −4, 4, −1, −3, −3, 0, 4, −6, −3, −3, 2, −6, −4, 5, −1, 6, 4]	7.345496178	3.404387426

Appendix III: Raw Data of Solving time vs. Constraints

All Constraints Solve with Different Solver						
num	cap	refill	fun	Chuffed Solving Time	Gecode Solving Time	Rate of Time Spend
1	5	2	[3]	0.06989	0.07516	0.92993
2	5	2	[−5, 4]	0.04824	0.06573	0.73388
3	5	2	[−1, −4, 5]	0.05071	0.06657	0.76185
4	5	2	[−4, 1, 0, −6]	0.04991	0.07258	0.68763
5	5	2	[5, 2, 0, 3, 1]	0.05312	0.07202	0.73758
6	5	2	[−2, 5, 2, −6, 2, −2]	0.05154	0.07378	0.69861
7	5	2	[2, 2, −2, −3, −4, 5, 0]	0.05665	0.07766	0.72950
8	5	2	[−2, −3, −2, 0, 6, −4, 0, −3]	0.05271	0.07268	0.72528
9	5	2	[0, 0, −2, −5, 2, 3, −3, 4, 5]	0.05798	0.07678	0.75519
10	5	2	[−3, 3, −3, −4, 3, −3, 6, −4, −4, 2]	0.05776	0.08066	0.71611
11	5	2	[6, −2, 1, 2, 0, 1, 0, 0, −3, −5, 0]	0.05607	0.07723	0.72596
12	5	2	[3, 2, 5, −5, 0, 5, 6, −3, −1, −1, −4, 4]	0.06011	0.08252	0.72843
13	5	2	[0, −1, −2, 1, 5, −3, −5, 5, −3, 5, −4, 2, −1]	0.05766	0.07821	0.73720
14	5	2	[4, −1, −3, −1, 4, −6, 2, −1, 0, 6, 1, 1, 2, 4]	0.06594	0.09105	0.72425
15	5	2	[−4, −5, 0, −3, 6, 6, −2, 2, 0, −5, 0, 6, −1, 5, 0]	0.06036	0.08260	0.73081
16	5	2	[4, −3, 4, 6, 0, 2, −5, 4, 5, 5, −2, −4, −5, 0, −3, −6]	0.06388	0.08425	0.75823
17	5	2	[−4, −1, 2, 4, −4, −2, 4, 4, −3, 3, 0, 0, 1, −5, 4, 1, 5]	0.08619	0.13654	0.63123
18	5	2	[6, −6, 2, −1, −1, 1, 0, 0, −1, −4, 0, −1, −4, −5, 1, −5, −5, 6]	0.06947	0.09201	0.75508
19	5	2	[−4, −4, −6, 3, 2, −1, 6, −2, 3, 0, 4, 2, 0, 3, −5, 3, −4, 0, 6]	0.08338	0.11038	0.75536
20	5	2	[1, −2, 1, −2, 6, 5, 5, 0, −2, 4, −5, −4, 5, 0, 3, −6, 0, 6, 1, −4]	0.07837	0.11012	0.71167
21	5	2	[6, −5, 2, 5, −5, 5, 4, −5, 1, 4, 0, 3, 0, 1, 5, 2, −3, −5, −4, 5, 4]	0.07631	0.10043	0.75989
22	5	2	[−6, 3, 5, 4, 5, 1, −1, 5, 2, 3, −2, 3, 1, −4, −6, −2, −4, −4, 1, −3, −6, 2]	0.07399	0.09656	0.76625
23	5	2	[6, −2, 0, −1, 1, 1, −6, 3, −4, −3, 6, 0, 3, 5, 3, 0, 4, −6, −6, 2, 6, −5, −3]	0.08268	0.12891	0.64134
24	5	2	[3, −3, −2, 5, −2, 4, 5, 0, −4, −5, 0, 2, 1, −3, 0, −2, 0, 3, 4, 0, 3, 4, −1, −4]	0.07590	0.11011	0.68930
25	5	2	[−5, −6, 4, 2, 1, 0, 5, 2, 0, −2, 4, 1, −4, 4, 5, −1, 4, −2, −6, −1, −1, 6, 0, 3, 2]	0.13152	0.75051	0.17524
26	5	2	[6, −2, −2, 5, 0, −1, −4, −2, 5, −6, −2, −3, −2, 5, 6, 2, −4, −2, −4, −4, −3, −1, −1, −1, 1, 6]	0.08152	0.10396	0.78418
27	5	2	[−6, −2, 1, −5, −5, 4, 6, 0, 5, −1, −1, 6, −4, 4, −5, −2, 0, −5, −5, −6, 0, 1, 0, 5, 6, −5, −1]	0.08148	0.10706	0.76105
28	5	2	[−3, −2, −5, −1, −4, 1, −3, 4, 1, 4, −4, −2, 3, 6, −1, −4, 0, −4, −5, 1, −2, 1, −1, 5, 0, 2, −4, −6]	0.07905	0.11330	0.69766
29	5	2	[−5, −5, 5, 2, −5, −4, −1, −1, 2, 4, −5, 1, −4, 0, 4, 0, 1, 2, −6, 0, −6, 4, −1, 5, 1, −4, 6, 3, −2]	0.10592	2.40769	0.04399
30	5	2	[1, 3, 0, −5, −2, 4, −5, −3, 4, 2, 5, −1, −2, −3, −5, 3, −5, −4, −5, −4, 0, −4, −2, 1, 4, 2, −5, −5, 5, 6]	0.11189	2.05035	0.05457

Appendix IV: Raw Data of Solver vs. Solving time

Python Solver vs. Num			
num	cap	refill	Solving Time
250000	5	2	0.211975908
260000	5	2	0.214164495
270000	5	2	0.227190733
280000	5	2	0.227617836
290000	5	2	0.246705246
300000	5	2	0.254308033
310000	5	2	0.260949945
320000	5	2	0.280231533
330000	5	2	0.30059391
340000	5	2	0.315776653
350000	5	2	0.310240269
360000	5	2	0.341158485
370000	5	2	0.325730562
380000	5	2	0.326330662
390000	5	2	0.331057596
400000	5	2	0.335939169
410000	5	2	0.346508837
420000	5	2	0.346516895
430000	5	2	0.363355942
440000	5	2	0.382754841
450000	5	2	0.398904877
460000	5	2	0.436892395
470000	5	2	0.458186102
480000	5	2	0.421823692
490000	5	2	0.45517478
500000	5	2	0.476152706
510000	5	2	0.476926994
520000	5	2	0.506868124
530000	5	2	0.493291521
540000	5	2	0.495414352

Appendix V

Python Solver vs. Refill			
num	cap	refill	Solving Time
340000	5	2	0.289395571
340000	5	3	0.256372166
340000	5	4	0.247841883
340000	5	5	0.209693956
340000	5	6	0.206986809
340000	5	7	0.208302593
340000	5	8	0.208424139
340000	5	9	0.207482147
340000	5	10	0.20785923
340000	5	11	0.208225965
340000	5	12	0.205075359
340000	5	13	0.20400629
340000	5	14	0.205075359
340000	5	15	0.202320004
340000	5	16	0.202931786
340000	5	17	0.200906754
340000	5	18	0.20447135
340000	5	19	0.2053442
340000	5	20	0.200079203
340000	5	21	0.200252676
340000	5	22	0.204003668
340000	5	23	0.204374313
340000	5	24	0.204475689
340000	5	25	0.204398155
340000	5	26	0.201884747
340000	5	27	0.204955244
340000	5	28	0.199237156
340000	5	29	0.198725367
340000	5	30	0.211338377
340000	5	31	0.200878

Appendix VI

Python Solver vs. Cap			
num	cap	refill	Solving Time
340000	2	2	0.209855461
340000	3	2	0.239405775
340000	4	2	0.262027454
340000	5	2	0.281292915
340000	6	2	0.294890213
340000	7	2	0.307143164
340000	8	2	0.324502993
340000	9	2	0.332557964
340000	10	2	0.342656565
340000	11	2	0.350297022
340000	12	2	0.357004404
340000	13	2	0.365444613
340000	14	2	0.371984434
340000	15	2	0.373246574
340000	16	2	0.38129487
340000	17	2	0.385590792
340000	18	2	0.397928858
340000	19	2	0.395982742
340000	20	2	0.408091736
340000	21	2	0.41439476
340000	22	2	0.419801521
340000	23	2	0.419533491
340000	24	2	0.427646589
340000	25	2	0.43118186
340000	26	2	0.441831017
340000	27	2	0.446897745
340000	28	2	0.451513767
340000	29	2	0.458841228
340000	30	2	0.458951187
340000	31	2	0.463753176

Appendix VII

File List:

Sequential_Games.mzn

MiniZinc model with all constraints mentioned in the report.

Data_Generator.py

Generate random test data automatically, and save generated data into two types: .txt and .dzn. Variables can be edited for different data size.

Sequential_Games_Solver.py

A Python program that runs Sequential_Games.mzn in MiniZinc to solve several instances and record running time of each instances.

Can choose different solver by comment current solver line and uncomment the other solver line.

Python_Solver.py

Specific algorithm for this problem implemented in Python. Read instances from .txt files and record the running time for each instance.

test_case#.dzn / test_case#.txt

Instance data files in two file types for different use in MiniZinc and Python.

ReadMe.txt

Guide file with method of running each code