# Final Report

**Zherui Shao**

**301326242**

# Abstraction

This report has three sections. The first section is about the problem and the solver I chosen and the basic implementation of the question as well as some instances used to test the correctness of the implementation. During the process of coming up with different instances I explored some aspect of declarative problem solving. The second section of the report is some idea about the declarative problem solving, and my question and investigation about it. The third section of the report is an appendix which contain a list of files.

# Section 1

I chose the ‘Sequence Games’ problem, and the MiniZinc solver. I worked on optimization version.

I used all default settings.

At first I used the basic idea on the lecture notes to construct the specification of the problem.

```
int: num;
int: cap;
int: refill;
set of int: index = 1..num;
array[index] of int: fun;

array[index] of var int: p;
array[index] of var int: t;

constraint t[1] = cap;
constraint forall (i in index where i > 1) (
     if t[i-1] - p[i-1] + refill >= cap
     then t[i] = cap
     else t[i] = t[i-1] - p[i-1] + refill endif
     );
constraint forall (i in index) (
    p[i] >= 1 /\ p[i] <= t[i]
     );
solve maximize sum (i in index) (fun[i]*p[i]);
output [show(sum (i in index) (fun[i]*p[i]))];
```

My input is: num (for the number of games), cap (for the pocket capacity), refill (for the refill amount), K (for the goal used in decision version) and array fun (for the fun value of each game).

I used an array p to represent the number of played times of each game, and I used an array t to represent the number of tokens the player had at the beginning of each game.

Here are some basic test instances that I used to help debug my specifications (for this part I mainly focus on whether the algorithm can solve the instance).

1. num = 4, cap = 3, refill = 1, fun = [1,1,1,1].
   Running time: 661msec. Cost function values: 6
   Reason: It is the most straightforward instance. Some values are the lower bound, and fun values are all same positive values.

2. num = 5, cap = 3, refill = 1, fun = [1,2,3,4,5].
   Running time: 635msec. Cost function values: 25
   Reason: Test if it can solve the case that has different fun values with increasing order.
3. num = 6, cap = 4, refill = 2, fun = [6,5,4,3,2,1].
   Running time: 658msec. Cost function values: 54
   Reason: Test if it can solve the case that has different fun values with decreasing order. Test if it can solve the case that cap value and refill value are general (not in lower bound).
4. num = 7, cap = 7, refill = 3, fun = [2,1,6,3,5,7,4].
   Running time: 743msec. Cost function values: 122
   Reason: Test if it can solve the case that has different distinct fun values.
5. num = 7, cap = 7, refill = 3, fun = [1,1,6,3,6,7,4].
   Running time: 738msec. Cost function values: 118
   Reason: Test if it can solve the case that has different fun values.
6. num = 10, cap = 10, refill = 3, fun = [5,1,-6,3,5,7,-4,3,1,-7].
   Running time: 1s 422msec. Cost function values: 135
   Reason: Test if it can solve the case that has some negative fun values.
7. num = 10, cap = 10, refill = 3, fun = [-5,-1,-6,-3,-5,-7,-4,-3,-1,-7].
   Running time: 726msec. Cost function values: -42
   Reason: Test if it can solve the case that has all negative fun values.
8. num = 10, cap = 10, refill = 5, fun = [5,1,6,3,5,7,4,3,1,7].
   Running time: 5m 6s. Cost function values: 302
   Reason: It seems like the correctness of the specification is not a problem now. I want to try some hard instances and see what will happen. I think maybe the solver will not spend much time on deciding the games with negative fun values, so I set all the numbers in fun are positive. Time cost has significant difference with before.
9. num = 20, cap = 10, refill = 10, fun = [5,1,6,3,5,7,4,3,1,7, 5,1,6,3,5,7,4,3,1,7].
   Running time: 772msec. Cost function values: 420
   Reason: Test the case that refill is equal to cap. I think for such case solver should not spend much time whether the fun values are, so I double the length of the fun.
10. num = 10, cap = 10, refill = 3, fun = [-5,-1,-6,-3,-5,7,4,3,1,7].
    Running time: 1s 80msec. Cost function values: 95
    Reason: Test the case that has continued negative fun values and continue positive fun values.

At first I tried the three instances provided by the website and I got the correct result in less than one second. Then I came up with more instances to test the correctness of the implementation thoroughly. During the process I found some instances can be solved in several seconds while some instances needed several minutes. It made sense to me that lager num value will need more time, but I found that when I kept the values of num, cap and refill same, and just changed the values in fun, the running times will have significant differences. I then started to mainly focus on the values in fun.

**Observation and how I come up with the question I want to explore**

I kept num = 10, cap = 10, refill = 5. I tried many different fun values. As I mentioned in the reason part in instance 8, I thought maybe the solver would not spend much time on deciding the games with negative fun values. I thought all the negative game should be played only once even if players had sufficient coins, because played more negative game would decrease the total fun. My assumption was that the solver would just assign 1 to the playtimes of negative game and did not do much calculation. However I tried some fun values that only differed in the value of negative numbers, running times had significant differences. One of the typical pair of instances is fun = [-9,-9,-9,-9,-9,-9,-9,9,8,10] and fun = [-1,-1,-1,-1,-1,-1,-1,9,8,10]. The running time of the first one is 1s 569msec, and the running time of the second one is 2m 2s. I realized that the solver did not perform as I thought. Maybe I could add a heuristic that constraints all the negative games to be played only once, and this may decrease the running time of the solver for some instance.

I added some new code:

```
constraint forall (i in index) (
    if fun[i] < 0 then p[i] = 1 endif
    );
```

I still tried the two instances that I tried before. Now the running time of them are 699msec and 714msec. They were nearly the same. I realized that if it is not a coincidence, some different choices such as redundant constraints will have a huge influence on the performance of the solver.

# Section 2

The question I want to explore is that whether redundant constraints can improve the performance of the solver, and further more what redundant constraints in the game problem will improve the performance of MiniZinc, and how they improve the performance.

I came up with two redundant constraints, and I created two new versions of my code. Each version contained one redundant constraint. I did two experiments to compare the performance of the solver between each new version and the original version.

The first one is:

```
constraint forall (i in index) (
    if fun[i] < 0 then p[i] = 1 endif
    );
```

In the section 1 of this report, I described the details about this idea. I added this constraint into ‘code1’.

The second one is:

```
constraint forall (i in index where i > 1) (
    if fun[i] > fun[i-1] then p[i] >= p[i-1] endif
    /\ if fun[i] < fun[i-1] then p[i] <= p[i-1] endif
```

```
);
```

By analyzing the game problem, I found that if one game had higher fun value than its neighbors, it should not be played less than its neighbors. (Proof by contradiction: Suppose we have an array p which has maximum total fun, and some game i and game j = i+1 with fun[i] > fun[j] and p[i] < p[j]. If after game i the player has no money left, p[i] >= refill and p[j] <= refill. That is p[i] cannot be less than p[j], so if p[i] < p[j], the game j uses at least one token left from game i. If we used this token at game i, the new total fun will be totalFun-fun[j]+fun[i]. The new total fun is greater than the old one. The old one is not maximum value. This is a contradiction. Suppose we have an array p which has maximum total fun, and some game i and game j = i+1 with fun[i] < fun[j] and p[i] > p[j]. p[i] is at most cap, so p[j] < cap. If we played game i one less time and used the token on game j, the new total fun will be totalFun+fun[j]-fun[i]. The new total fun is greater than the old one. The old one is not maximum value. This is a contradiction.) I added this constraint into 'code2'.

**Experiment:**

I always used the default setting of the MiniZinc solver.

I used the running time to represent the performance of the solver.

For all the experiment I checked the answers of the two versions. They were always same. The new codes did find the correct answers as the original code. I just showed the running time of two versions.

Experiment for the first constraint:

In this experiment I came up with different instances. At first I tested some instances which did not contain negative numbers. I thought the performance of the two versions of code would not have significant difference. The result was same as what I thought. I did not show the experiment data of this process. Then I focused on instances that contained negative numbers. I thought sometimes the solver would spend a lot of time on deciding how many times the negative games should be played. In these cases, the redundant constraint could help a lot.

| Instances | Running time of code0 | Running time of code1 |
|---|---|---|
| num = 10<br>cap = 10<br>refill = 5<br>fun = [-1,-1,-1,-1,-1,-1,-1,9,8,10] | 2m 2s | 714msec |
| num = 10<br>cap = 10<br>refill = 6<br>fun = [-1,-1,-1,-2,-5,-1,7,10,8,9] | 1m 11s | 852msec |
| num = 10<br>cap = 10<br>refill = 7<br>fun = [-1,-1,-1,-1,-1,-1,7,10,8,9] | 3m 57s | 1s 210msec |
| num = 10<br>cap = 10 | 11s 773msec | 803msec |

| | | |
|---|---|---|
| refill = 7<br>fun = [1,-2,-3,5,-1,-7,3,2,7,10] | | |
| num = 10<br>cap = 10<br>refill = 7<br>fun = [-1,-2,-3,-5,-1,-7,-3,-2,-7,-10] | 660msec | 690msec |
| num = 10<br>cap = 10<br>refill = 5<br>fun = [-1,-1,-1,-1,-1,-1,-1,9,8,10] | 1s 569msec | 699msec |
| num = 10<br>cap = 10<br>refill = 5<br>fun = [5,-9,2,-9,9,-9,-9,9,-8,10] | 704msec | 698msec |
| num = 10<br>cap = 10<br>refill = 5<br>fun = [5,-9,2,-9,9,-9,-9,9,-8,10] | 707msec | 798msec |
| num = 15<br>cap = 6<br>refill = 3<br>fun = [5,-4,2,7,-1,-2,-5,-9,-9,-5,9,-1,2,10] | Stopped.<br>16m 1s | 832msec |
| num = 10<br>cap = 10<br>refill = 5<br>fun = [-1,-2,-1,-2,-1,9,8,10,9,10] | 1s 794msec | 711msec |

The first four instances show that there are some instances that need a long time to be solved by the original code. After adding the constraint, the time costs are decreased significantly. The instance 5, 8 shows that for some instances (if the running time of original code is already very short), the running time of the new version may be slightly longer. The instance 9 shows that for some instances that need extremely long time to finish in my computer, the constraint can help to decrease the running time.

Experiment for the second constraint:

I thought if the fun values were all same, the constraint could not help to prove the performance of the solver. I tried some instances that had identical fun values. I chose one to show below. Then I tried some instances that did not have two identical fun values as neighbors. I thought in these cases, the redundant constraint could help a lot. At last I tried some instances that contain negative fun values or only contain negative fun values.

| Instances | Running time of code0 | Running time of code2 |
|---|---|---|
| num = 10 | 10s 808msec | 11s 200msec |

| | | |
|---|---|---|
| cap = 10<br>refill = 3<br>fun = [1,1,1,1,1,1,1,1,1,1] | | |
| num = 10<br>cap = 10<br>refill = 3<br>fun = [1,2,3,4,5,6,7,8,9,10] | 7s 576msec | 660msec |
| num = 10<br>cap = 10<br>refill = 3<br>fun = [10,9,8,7,6,5,4,3,2,1] | 5s 420msec | 677msec |
| num = 10<br>cap = 10<br>refill = 5<br>fun = [1,9,1,9,1,9,1,9,1,9] | 2m 23s | 3s 467msec |
| num = 10<br>cap = 10<br>refill = 5<br>fun = [2,1,3,2,1,5,2,4,1,3] | 11m 30s | 3s 393msec |
| num = 10<br>cap = 10<br>refill = 5<br>fun = [-2,1,-3,2,-1,5,-2,4,-1,3] | 3s 300msec | 7s 793msec |
| num = 10<br>cap = 10<br>refill = 5<br>fun = [-2,-1,-3,-2,-1,-5,-2,-4,-1,-3] | 698msec | 736msec |

The first instance shows that if the fun values are all same, the running times of two versions did not have significant difference. The instances 2, 3, 4, 5 show that if the fun values do not contain negative values, and the neighbors in fun values are not identical, the running time of the new version is much shorter than the old version. The instances 6, 7 show that if the fun values contain negative numbers, the running time of new version maybe slightly longer than the old version.

**Discussion:**

I explored the differences of running time of the implementation with and without redundant constraints in different instances. In the experiment I found for some instances some redundant constraints could help to improve the performance of the solver. However this conclusion is only for the group of instances I tested.

From the exploration, I realize that the solver may waste some time on making needless

decisions. For example, in order to find the optimal solution the solver may not have some simple greedy algorithm and tried to play different times of negative games. However this is needless attempts for use because we know that more negative games will only decrease the total fun. If we can state the problem more clear and more detailed we can constrain the solver to do not do some needless attempts. We need have a more clear understanding of the problem rather than just throw the basic input into a black box and wait some magic happened.

If I want to continue this study in future, I have three approaches to make the study better. I will come up with more test instances or use a code to produce random instances that have some features I wanted (such as containing negative numbers in fun for my first experiment) to test the redundant constraints. I will try to use a better computer to test more complicated instances, and I want to spend more time on experiment. This time I just showed some cases that the redundant constrains helped the solver to reduce the running time from several minutes to several seconds. I want to explore whether it can reduce the running time from several days or several months to an acceptable time in industry. I will try to find whether the instances which can be finished significantly faster in the new code than original code have some common features.

# Section 3

Code0: It is the original version without any redundant constrains. It is the basic implementation I used in section 1, and I used it in my section 2 experiments.
Code1: It is the basic implementation with redundant constrain1.
Code2: It is the basic implementation with redundant constrain2.