CMPT 417 Final Project

Sequential Games

Huiyi Zou

301355563

Part I

Problem

The optimization version of sequential games problem from The 2015 LP/CP Programming Contest and our lecture notes. Here, I try to use two solvers MiniZinc and OR-Tools in Python to solve the problem.

Play a sequence of games $G_1, G_2, \ldots, G_n$ under the following conditions:

1. You may play each game multiple times, but all plays of game G_i must be made after playing G_{i-1} and before playing G_{i+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 G_1. After your last play of G_i, but before you begin playing G_{i+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.
6. The total fun you get must no less than K goal.

Given:

Number: $num \in \mathbb{N}$ of games;

Pocket capacity: $cap \in \mathbb{N}$;

Refill amount: $refill \in \mathbb{N}$;

Fun value: $fun_i \in \mathbb{Z}$ for each game $i \in [num]$;

Goal: $goal \in \mathbb{N}$.

Find:

A number of plays $play_i$ for each game $i \in [num]$ such that total fun is maximum and is at least $goal$.

Need:

Number of tokens before playing games: $token_i \in \mathbb{N}$ for each game $i \in [num]$;

Specification

I use the problem specification from notes:

1. total fun is at least $goal$:

$$sum(i, i \in [num], play(i) \cdot fun(i))$$

2. have cap tokens at first, have the minimum of cap and $token(i-1) - play(i-1) + refill)$ tokens when at game $i > 1$:

$$token(1) = cap \wedge \forall i[1 < i \leq num \rightarrow \exists x((x = token(i-1) - play(i-1) + refill) \wedge (x > cap \rightarrow token(i) = cap) \wedge (x \leq cap \rightarrow token(i) = x))]$$

3. play each game at least once and at most token times:

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

Solver 1: MiniZinc

I use Gecode as solver configuration.

```
int: num;
int: cap;
int: refill;
array [1..num] of int: fun;
int: goal;

array [1..num] of var int: token; % # of tokens before playing game i
array [1..num] of var int: play; % # of plays for game i
var int: total_fun;

constraint total_fun = sum(i in 1..num)(play[i]*fun[i]);
constraint total_fun >= goal;
constraint ((token[1]=cap)/\
           forall(i in 2..num)
           (exists(x in 0..(cap+refill))
            ((x = token[i-1]-play[i-1]+refill)/\
             (x>cap -> token[i]=cap)/\
             (x<=cap -> token[i]=x))));

constraint forall(i in 1..num)(play[i]>=1 /\ play[i]<=token[i]);

solve maximize total_fun;
output ["max total fun = \(total_fun)\n"];
```

I declare the parameter variables num, cap, refill, fun, and goal with their data types. The token, play and total_fun are decision variables that will be assigned. The constraints represent the above specifications. Since this is an optimization problem – find the maximum total_fun that satisfies the goal, this model use `solve maximize` to specifies to find a solution that maximizes `total_fun`. If the satisfied max total_fun is found, it will show `OPTIMAL_SOLUTION`, else it is `UNSATISFIABLE`.

Solver 2: OR-Tools for Python

I use cp_model module which can build and solve CP-SAT models.

```
def ortoolsSolver(num, cap, refill, fun, goal):
    model = cp_model.CpModel()
    token = [model.NewIntVar(-2147483648, 2147483647, 't%i' % i)
             for i in range(1, num + 1)]
    play = [model.NewIntVar(-2147483648, 2147483647, 'q%i' % i)
            for i in range(1, num + 1)]
    compare = [model.NewBoolVar('c%i' % i)
               for i in range(1, num + 1)]
    total_fun = model.NewIntVar(-2147483648, 2147483647, 'total_fun')
    model.Add(total_fun = sum([fun[i] * play[i] for i in range(num)])
    model.Add(total_fun >= goal)
    model.Add(token[0] == cap)
    for i in range(num):
        model.Add(token[i] - play[i] + refill > cap).OnlyEnforceIf(compare[i])
        model.Add(token[i] - play[i] + refill <=
                  cap).OnlyEnforceIf(compare[i].Not())
        model.Add(play[i] >= 1)
        model.Add(play[i] <= token[i])
    for i in range(1, num):
        model.Add(token[i] == cap).OnlyEnforceIf(compare[i - 1])
        model.Add(token[i] == token[i - 1] - play[i - 1] +
                  refill).OnlyEnforceIf(compare[i - 1].Not())
    model.Maximize(total_fun)
    solver = cp_model.CpSolver()
    status = solver.Solve(model)
    sat = solver.StatusName()
    time = solver.UserTime()
    if status == cp_model.INFEASIBLE:
        token = None
        play = None
        total_fun = None
    else:
        token = [solver.Value(token[i]) for i in range(num)]
        play = [solver.Value(play[i]) for i in range(num)]
        total_fun = solver.Value(total_fun)
    return [sat, token, play, total_fun, time]
```

The model creates variables token, play and total fun with range of int32. To decide the value of `token[i+1]`, we need use bool variable `compare`, that `compare[i]` is true if `token[i]-play[i]+refill > cap`, and is false if `token[i]-play[i]+refill ≤ cap`. When `compare[i]` is true, we let `token[i]=cap`, else we let `token[i]=token[i-1]-play[i-1]+refill`. The `model.Maximize(total_fun)` finds the maximum total_fun that satisfies the goal. If the satisfied max total_fun is found, it will show `OPTIMAL`, else it is `INFEASIBLE`, which means unsatisfiable.

Instances

Test instances are constructed by a Python program instances_genetator.py.

As description from The 2015 LP/CP Programming Contest, 4≤num≤10, 3≤cap≤10, 0<refill≤cap, -10≤fun[i]≤10. Therefore, the goal should be between -100 and 1000.

I generate 14 instances with increasing values of num from 4 to 10, and set other values randomly in their domains.

num	cap	refill	fun	goal
4	9	2	[9, 3, 4, 10]	815
4	6	5	[7, 6, 4, 9]	107
5	4	4	[-3, -9, -5, -9, -3]	-61
5	9	2	[-8, 2, -10, -4, 10]	273
6	7	6	[8, -9, -3, 3, 6, -6]	816
6	7	6	[-4, 10, -1, 8, 9, -6]	-1
7	4	2	[-4, 1, -10, -4, -3, 0, -9]	103
7	6	1	[-7, -8, -2, 8, 8, 1, -7]	524
8	6	4	[-7, 9, 8, 8, -9, 6, -1, -8]	-53
8	10	8	[6, -6, -3, 7, 0, -10, 2, 7]	544
9	7	4	[-9, -2, -4, 2, 4, -1, 10, 0, 3]	511
9	7	1	[-1, 2, 3, 2, 3, 0, 7, -9, -10]	615
10	10	1	[-6, 3, 0, 6, -8, 3, 6, 9, -5, -6]	563
10	9	3	[-8, 4, 1, 1, -7, -5, 4, -5, -10, 8]	-42

file	num	cap	refill	fun	goal
04_1.dzn	4	5	2	[4, 3, 2, 1]	0
04_2.dzn	4	5	2	[1, 2, 3, 4]	0
04_3.dzn	4	5	2	[4, 3, 2, 1]	150
04_4.dzn	4	5	2	[1, 2, 3, 4]	150
05_1.dzn	5	5	2	[5, 4, 3, 2, 1]	0
05_2.dzn	5	5	2	[1, 2, 3, 4, 5]	0
05_3.dzn	5	5	2	[5, 4, 3, 2, 1]	150
05_4.dzn	5	5	2	[1, 2, 3, 4, 5]	150
06_1.dzn	6	5	2	[6, 5, 4, 3, 2, 1]	0
06_2.dzn	6	5	2	[1, 2, 3, 4, 5, 6]	0
06_3.dzn	6	5	2	[6, 5, 4, 3, 2, 1]	150
06_4.dzn	6	5	2	[1, 2, 3, 4, 5, 6]	150
07_1.dzn	7	5	2	[7, 6, 5, 4, 3, 2, 1]	0
07_2.dzn	7	5	2	[1, 2, 3, 4, 5, 6, 7]	0
07_3.dzn	7	5	2	[7, 6, 5, 4, 3, 2, 1]	150
07_4.dzn	7	5	2	[1, 2, 3, 4, 5, 6, 7]	150
08_1.dzn	8	5	2	[8, 7, 6, 5, 4, 3, 2, 1]	0
08_2.dzn	8	5	2	[1, 2, 3, 4, 5, 6, 7, 8]	0
08_3.dzn	8	5	2	[8, 7, 6, 5, 4, 3, 2, 1]	150
08_4.dzn	8	5	2	[1, 2, 3, 4, 5, 6, 7, 8]	150
09_1.dzn	9	5	2	[9, 8, 7, 6, 5, 4, 3, 2, 1]	0
09_2.dzn	9	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9]	0
09_3.dzn	9	5	2	[9, 8, 7, 6, 5, 4, 3, 2, 1]	150
09_4.dzn	9	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9]	150
10_1.dzn	10	5	2	[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	0
10_2.dzn	10	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	0
10_3.dzn	10	5	2	[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	150
10_4.dzn	10	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	150

The num usually has a relatively larger effect on the runtime of solving the problem, I want to figure out whether the value of num affects the runtime for this problem.

I create another 28 instances (half satisfiable and half unsatisfiable) with the fixed values of cap and refill, where cap is 5 and refill is 2, and the ordered fun.

Results

The result of instances with random values: (runtime measured in seconds)

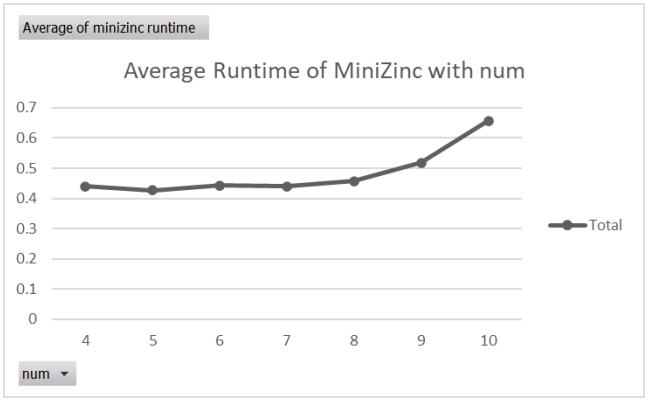
file	num	cap	refill	fun	goal	minizinc sat	minizinc token	minizinc play	minizinc total_fun	minizinc time
04_1.dzn	4	9	2	[9, 3, 4, 10]	815	UNSATISFIABLE				0.49
04_2.dzn	4	6	5	[7, 6, 4, 9]	107	OPTIMAL_SOLUTION	[6, 5, 5, 6]	[6, 5, 4, 6]	142	0.48
05_1.dzn	5	4	4	[-3, -9, -5, -9, -3]	-61	OPTIMAL_SOLUTION	[4, 4, 4, 4, 4]	[1, 1, 1, 1, 1]	-29	0.54
05_2.dzn	5	9	2	[-8, 2, -10, -4, 10]	273	UNSATISFIABLE				0.49
06_1.dzn	6	7	6	[8, -9, -3, 3, 6, -6]	816	UNSATISFIABLE				0.49
06_2.dzn	6	7	6	[-4, 10, -1, 8, 9, -6]	-1	OPTIMAL_SOLUTION	[7, 7, 6, 7, 7, 6]	[1, 7, 1, 6, 7, 1]	170	0.46
07_1.dzn	7	4	2	[-4, 1, -10, -4, -3, 0, -9]	103	UNSATISFIABLE				0.47
07_2.dzn	7	6	1	[-7, -8, -2, 8, 8, 1, -7]	524	UNSATISFIABLE				0.46
08_1.dzn	8	6	4	[-7, 9, 8, 8, -9, 6, -1, -8]	-53	OPTIMAL_SOLUTION	[6, 6, 4, 6, 4, 6, 4, 6]	[1, 6, 2, 6, 1, 6, 1, 1]	129	0.46
08_2.dzn	8	10	8	[6, -6, -3, 7, 0, -10, 2, 7]	544	UNSATISFIABLE				1.96
09_1.dzn	9	7	4	[-9, -2, -4, 2, 4, -1, 10, 0, 3]	511	UNSATISFIABLE				1.11
09_2.dzn	9	7	1	[-1, 2, 3, 2, 3, 0, 7, -9, -10]	615	UNSATISFIABLE				0.5
10_1.dzn	10	10	1	[-6, 3, 0, 6, -8, 3, 6, 9, -5, -6]	563	UNSATISFIABLE				143.01
10_2.dzn	10	9	3	[-8, 4, 1, 1, -7, -5, 4, -5, -10, 8]	-42	OPTIMAL_SOLUTION	[9, 9, 3, 5, 5, 7, 9, 5, 7, 9]	[1, 9, 1, 3, 1, 1, 7, 1, 1, 9]	105	1.37

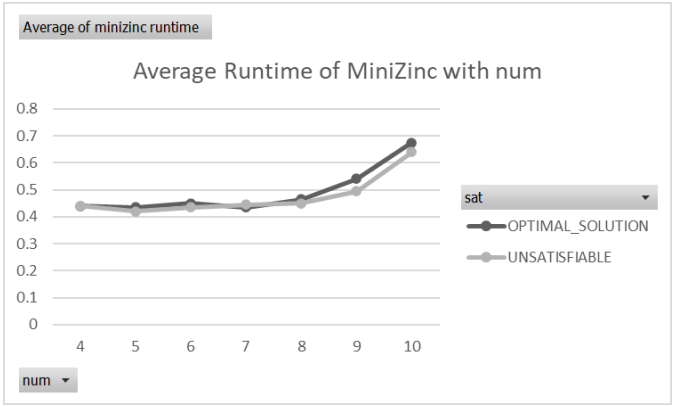
minizinc token	minizinc play	minizinc total_fun	minizinc time	ortools sat	ortools token	ortools play	ortools total_fun	ortools time
			0.49	INFEASIBLE				0.001517
[6, 5, 5, 6]	[6, 5, 4, 6]	142	0.48	OPTIMAL	[6, 5, 5, 6]	[6, 5, 4, 6]	142	0.002428
[4, 4, 4, 4, 4]	[1, 1, 1, 1, 1]	-29	0.54	OPTIMAL	[4, 4, 4, 4, 4]	[1, 1, 1, 1, 1]	-29	0.001048
			0.49	INFEASIBLE				0.001382
			0.49	INFEASIBLE				0.002251
[7, 7, 6, 7, 7, 6]	[1, 7, 1, 6, 7, 1]	170	0.46	OPTIMAL	[7, 7, 6, 7, 7, 6]	[1, 7, 1, 6, 7, 1]	170	0.002665
			0.47	INFEASIBLE				0.000332
			0.46	INFEASIBLE				0.000773
[6, 6, 4, 6, 4, 6, 4, 6]	[1, 6, 2, 6, 1, 6, 1, 1]	129	0.46	OPTIMAL	[6, 6, 4, 6, 4, 6, 4, 6]	[1, 6, 2, 6, 1, 6, 1, 1]	129	0.005848
			1.96	INFEASIBLE				0.001463
			1.11	INFEASIBLE				0.001549
			0.5	INFEASIBLE				0.000899
			143.01	INFEASIBLE				0.001079
[9, 9, 3, 5, 5, 7, 9, 5, 7, 9]	[1, 9, 1, 3, 1, 1, 7, 1, 1, 9]	105	1.37	OPTIMAL	[9, 9, 3, 3, 5, 7, 9, 5, 7, 9]	[1, 9, 3, 1, 1, 1, 7, 1, 1, 9]	105	0.012663

Those two solvers both get the same results. It seems that there is a relationship between runtime and the value of num. The runtime of the OR-Tools solver is much faster than the MiniZinc solver for this problem. And the runtime of the OR-Tools solver is relative to the satisfiability of instances. For the instance 10_1, the runtime of the MiniZinc solver is extremely large. It may because of the specification used and the values of variables.

The result of instances with fixed values running in MiniZinc solver: (runtime measured in seconds)

file	num	cap	refill	fun	goal	sat	token	play	total_fun	runtime
04_1.dzn	4	5	2	[4, 3, 2, 1]	0	OPTIMAL_SOLUTION	[5, 2, 2, 2]	[5, 2, 2, 2]	32	0.45
04_2.dzn	4	5	2	[1, 2, 3, 4]	0	OPTIMAL_SOLUTION	[5, 5, 5, 5]	[2, 2, 2, 5]	32	0.43
04_3.dzn	4	5	2	[4, 3, 2, 1]	150	UNSATISFIABLE				0.44
04_4.dzn	4	5	2	[1, 2, 3, 4]	150	UNSATISFIABLE				0.44
05_1.dzn	5	5	2	[5, 4, 3, 2, 1]	0	OPTIMAL_SOLUTION	[5, 2, 2, 2, 2]	[5, 2, 2, 2, 2]	45	0.44
05_2.dzn	5	5	2	[1, 2, 3, 4, 5]	0	OPTIMAL_SOLUTION	[5, 5, 5, 5, 5]	[2, 2, 2, 2, 5]	45	0.43
05_3.dzn	5	5	2	[5, 4, 3, 2, 1]	150	UNSATISFIABLE				0.43
05_4.dzn	5	5	2	[1, 2, 3, 4, 5]	150	UNSATISFIABLE				0.41
06_1.dzn	6	5	2	[6, 5, 4, 3, 2, 1]	0	OPTIMAL_SOLUTION	[5, 2, 2, 2, 2, 2]	[5, 2, 2, 2, 2, 2]	60	0.47
06_2.dzn	6	5	2	[1, 2, 3, 4, 5, 6]	0	OPTIMAL_SOLUTION	[5, 5, 5, 5, 5, 5]	[2, 2, 2, 2, 2, 5]	60	0.43
06_3.dzn	6	5	2	[6, 5, 4, 3, 2, 1]	150	UNSATISFIABLE				0.44
06_4.dzn	6	5	2	[1, 2, 3, 4, 5, 6]	150	UNSATISFIABLE				0.43
07_1.dzn	7	5	2	[7, 6, 5, 4, 3, 2, 1]	0	OPTIMAL_SOLUTION	[5, 2, 2, 2, 2, 2, 2]	[5, 2, 2, 2, 2, 2, 2]	77	0.43
07_2.dzn	7	5	2	[1, 2, 3, 4, 5, 6, 7]	0	OPTIMAL_SOLUTION	[5, 5, 5, 5, 5, 5, 5]	[2, 2, 2, 2, 2, 2, 5]	77	0.44
07_3.dzn	7	5	2	[7, 6, 5, 4, 3, 2, 1]	150	UNSATISFIABLE				0.44
07_4.dzn	7	5	2	[1, 2, 3, 4, 5, 6, 7]	150	UNSATISFIABLE				0.45
08_1.dzn	8	5	2	[8, 7, 6, 5, 4, 3, 2, 1]	0	OPTIMAL_SOLUTION	[5, 2, 2, 2, 2, 2, 2, 2]	[5, 2, 2, 2, 2, 2, 2, 2]	96	0.46
08_2.dzn	8	5	2	[1, 2, 3, 4, 5, 6, 7, 8]	0	OPTIMAL_SOLUTION	[5, 5, 5, 5, 5, 5, 5, 5]	[2, 2, 2, 2, 2, 2, 2, 5]	96	0.47
08_3.dzn	8	5	2	[8, 7, 6, 5, 4, 3, 2, 1]	150	UNSATISFIABLE				0.45
08_4.dzn	8	5	2	[1, 2, 3, 4, 5, 6, 7, 8]	150	UNSATISFIABLE				0.45
09_1.dzn	9	5	2	[9, 8, 7, 6, 5, 4, 3, 2, 1]	0	OPTIMAL_SOLUTION	[5, 2, 2, 2, 2, 2, 2, 2, 2]	[5, 2, 2, 2, 2, 2, 2, 2, 2]	117	0.52
09_2.dzn	9	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9]	0	OPTIMAL_SOLUTION	[5, 5, 5, 5, 5, 5, 5, 5, 5]	[2, 2, 2, 2, 2, 2, 2, 2, 5]	117	0.56
09_3.dzn	9	5	2	[9, 8, 7, 6, 5, 4, 3, 2, 1]	150	UNSATISFIABLE				0.49
09_4.dzn	9	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9]	150	UNSATISFIABLE				0.5
10_1.dzn	10	5	2	[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	0	OPTIMAL_SOLUTION	[5, 2, 2, 2, 2, 2, 2, 2, 2, 2]	[5, 2, 2, 2, 2, 2, 2, 2, 2, 2]	140	0.64
10_2.dzn	10	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	0	OPTIMAL_SOLUTION	[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 5]	140	0.71
10_3.dzn	10	5	2	[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	150	UNSATISFIABLE				0.64
10_4.dzn	10	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	150	UNSATISFIABLE				0.64

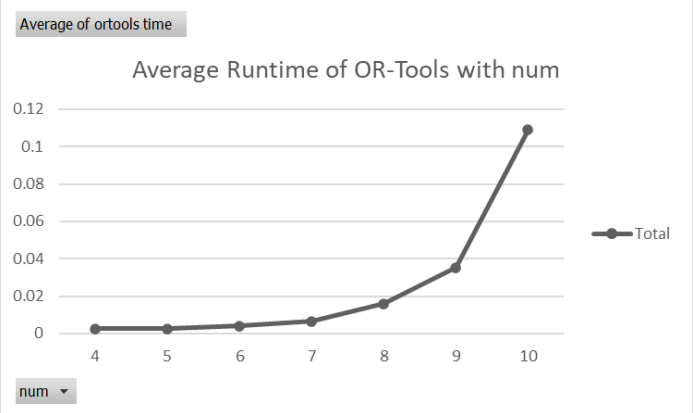


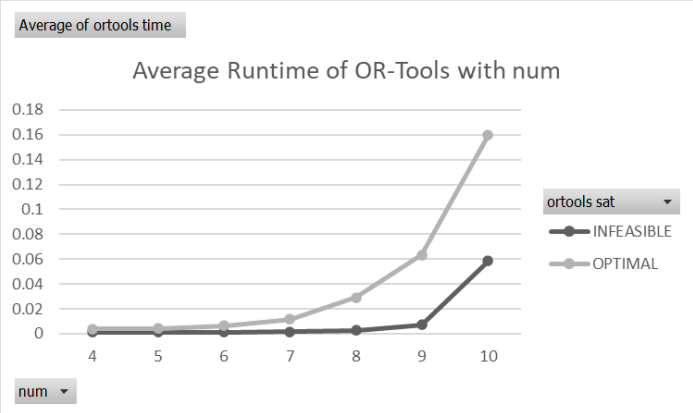


There is an upward trend of the average runtime of MiniZinc solver with an increasing value of num. The satisfiable instances sometimes need slightly more time to solve than the unsatisfiable instances.

The result of instances with fixed values running in OR-Tools solver: (runtime measured in seconds)

file	num	cap	refill	fun	goal	sat	token	play	total_fun	time
04_1.dzn	4	5	2	[4, 3, 2, 1]	0	OPTIMAL	[5, 2, 2, 2]	[5, 2, 2, 2]	32	0.00349
04_2.dzn	4	5	2	[1, 2, 3, 4]	0	OPTIMAL	[5, 5, 5, 5]	[2, 2, 2, 5]	32	0.003713
04_3.dzn	4	5	2	[4, 3, 2, 1]	150	INFEASIBLE				0.00103
04_4.dzn	4	5	2	[1, 2, 3, 4]	150	INFEASIBLE				0.00155
05_1.dzn	5	5	2	[5, 4, 3, 2, 1]	0	OPTIMAL	[5, 2, 2, 2, 2]	[5, 2, 2, 2, 2]	45	0.003411
05_2.dzn	5	5	2	[1, 2, 3, 4, 5]	0	OPTIMAL	[5, 5, 5, 5, 5]	[2, 2, 2, 2, 5]	45	0.00473
05_3.dzn	5	5	2	[5, 4, 3, 2, 1]	150	INFEASIBLE				0.00111
05_4.dzn	5	5	2	[1, 2, 3, 4, 5]	150	INFEASIBLE				0.001225
06_1.dzn	6	5	2	[6, 5, 4, 3, 2, 1]	0	OPTIMAL	[5, 2, 2, 2, 2, 2]	[5, 2, 2, 2, 2, 2]	60	0.00675
06_2.dzn	6	5	2	[1, 2, 3, 4, 5, 6]	0	OPTIMAL	[5, 5, 5, 5, 5, 5]	[2, 2, 2, 2, 2, 5]	60	0.006372
06_3.dzn	6	5	2	[6, 5, 4, 3, 2, 1]	150	INFEASIBLE				0.001558
06_4.dzn	6	5	2	[1, 2, 3, 4, 5, 6]	150	INFEASIBLE				0.001148
07_1.dzn	7	5	2	[7, 6, 5, 4, 3, 2, 1]	0	OPTIMAL	[5, 2, 2, 2, 2, 2, 2]	[5, 2, 2, 2, 2, 2, 2]	77	0.010804
07_2.dzn	7	5	2	[1, 2, 3, 4, 5, 6, 7]	0	OPTIMAL	[5, 5, 5, 5, 5, 5, 5]	[2, 2, 2, 2, 2, 2, 5]	77	0.012136
07_3.dzn	7	5	2	[7, 6, 5, 4, 3, 2, 1]	150	INFEASIBLE				0.00133
07_4.dzn	7	5	2	[1, 2, 3, 4, 5, 6, 7]	150	INFEASIBLE				0.001425
08_1.dzn	8	5	2	[8, 7, 6, 5, 4, 3, 2, 1]	0	OPTIMAL	[5, 2, 2, 2, 2, 2, 2, 2]	[5, 2, 2, 2, 2, 2, 2, 2]	96	0.029359
08_2.dzn	8	5	2	[1, 2, 3, 4, 5, 6, 7, 8]	0	OPTIMAL	[5, 5, 5, 5, 5, 5, 5, 5]	[2, 2, 2, 2, 2, 2, 2, 5]	96	0.029001
08_3.dzn	8	5	2	[8, 7, 6, 5, 4, 3, 2, 1]	150	INFEASIBLE				0.002705
08_4.dzn	8	5	2	[1, 2, 3, 4, 5, 6, 7, 8]	150	INFEASIBLE				0.002686
09_1.dzn	9	5	2	[9, 8, 7, 6, 5, 4, 3, 2, 1]	0	OPTIMAL	[5, 2, 2, 2, 2, 2, 2, 2, 2]	[5, 2, 2, 2, 2, 2, 2, 2, 2]	117	0.056776
09_2.dzn	9	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9]	0	OPTIMAL	[5, 5, 5, 5, 5, 5, 5, 5, 5]	[2, 2, 2, 2, 2, 2, 2, 2, 5]	117	0.069843
09_3.dzn	9	5	2	[9, 8, 7, 6, 5, 4, 3, 2, 1]	150	INFEASIBLE				0.004691
09_4.dzn	9	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9]	150	INFEASIBLE				0.009211
10_1.dzn	10	5	2	[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	0	OPTIMAL	[5, 2, 2, 2, 2, 2, 2, 2, 2, 2]	[5, 2, 2, 2, 2, 2, 2, 2, 2, 2]	140	0.15382
10_2.dzn	10	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	0	OPTIMAL	[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 5]	140	0.165058
10_3.dzn	10	5	2	[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	150	INFEASIBLE				0.038766
10_4.dzn	10	5	2	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	150	INFEASIBLE				0.078228





The average runtime increases in the OR-Tools solver when the value of num increases. The satisfiable instances need much more time to solve than the unsatisfiable instances.

Part II

The MiniZinc solver has a bad runtime result for the random instance 10_1. There must be other instances that those two solvers cannot solve efficiently. To optimize the performance of the solvers, I analyze the result of different implementations and evaluate the performance using runtime. For convenience, the instances with fixed values of cap, refill and ordered fun are tested. Referred to the MiniZinc tutorial section "Effective Modelling Practices in MiniZinc," I try to improve the performance through those ways:

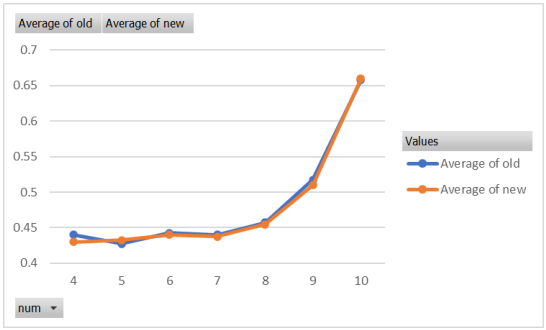
1. Reduce Variable.

The total_fun is a variable needed to be assigned in both original MiniZinc and OR-Tools solvers, but it can be defined by function $sum(i, i \in [num], play(i) \cdot fun(i))$. Therefore, I represent the total_fun by other variables to reduce the number of variables.

- MiniZinc

```
var int: total_fun;
constraint total_fun = sum(i in 1..num)(play[i]*fun[i]);
constraint total_fun >= goal;
```

➔ `constraint sum(i in 1..num)(play[i]*fun[i]) >= goal`

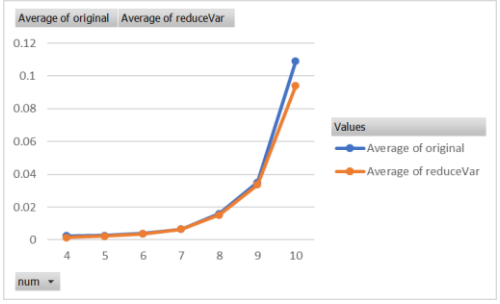


The plot of the average runtime of old and new MiniZinc solvers shows the runtime before and after reducing total_fun variable is very close. We cannot conclude this reduction improves performance.

- OR-Tools

```
total_fun = model.NewIntVar(-2147483648, 2147483647, 'total_fun')
model.Add(total_fun = sum([fun[i] * play[i] for i in range(num)])
```

➔ `total_fun = sum([fun[i] * play[i] for i in range(num)]`



The average runtime of the new OR-Tools solver after reducing total_fun variable is less than the original one. Reducing variables can improve the performance of this OR-Tools solver.

2. Add Redundant Constraints.

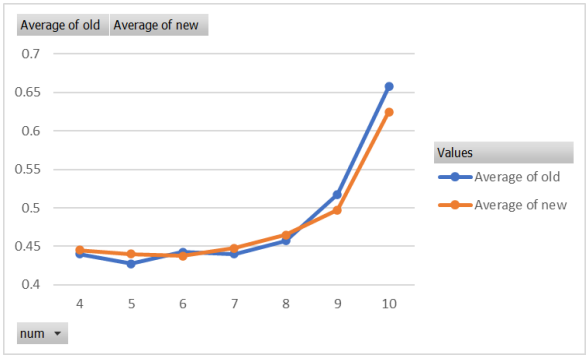
Sequential Games is an optimization problem. If make specification with more details, solvers may be more efficient. Since we need to find the max total_fun, if the fun at game i is negative, play game i one time.

- MiniZinc

```
constraint forall(i in 1..num)(play[i]>=1 /\ play[i]<=token[i]);
```

➔
```
constraint forall(i in 1..num)((if fun[i]<0 then
play[i]=1 else play[i]>=1 endif) /\
play[i]<=token[i]);
```

The MiniZinc solver with redundant constraints can have a faster runtime when the value of num is large enough, which is not effective for a small num.

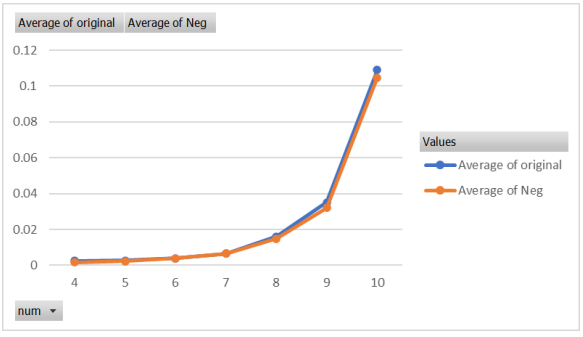


- OR-Tools

➔
```
neg = [model.NewBoolVar('n%i' % i) for i in range(1, num + 1)]
for i in range(num):
        model.Add(fun[i] < 0).OnlyEnforceIf(neg[i])
        model.Add(fun[i] >= 0).OnlyEnforceIf(neg[i].Not())
        model.Add(play[i] == 1).OnlyEnforceIf(neg[i])
        model.Add(play[i] >= 1).OnlyEnforceIf(neg[i].Not())
```

I add another Bool variable neg[i] to constraint the value of play[i]. If fun[i]<0, then neg[i] is true, which means the play[i] =1; if fun[i]>=0, then neg[i] is false, which means the play[i] >= 1.

The OR-Tools solver with the redundant constraints has a slightly faster runtime than the original.

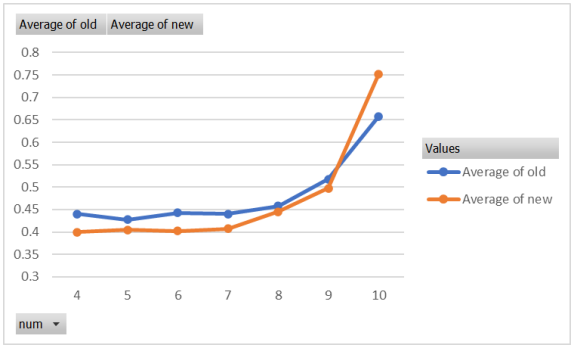


3. Limit Variable Range.

The solvers are more effective if the bounds of variables are tighter. The token and play are sets of integers in our original solvers, and we have known the range of variables in instances. We need to have at least one token before each game and the number of tokens cannot exceed the capacity of our pocket. We play each game at least once and we can play each game many times if we have enough tokens.

- MiniZinc

```
array [1..num] of var int: token;
array [1..num] of var int: play;
➔ array [1..num] of var 1..cap: token;
  array [1..num] of var 1..cap: play;
```

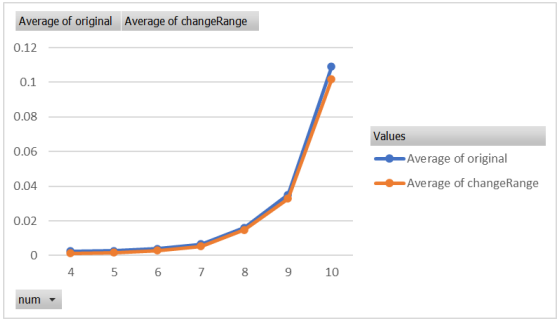


The original MiniZinc solver has unbounded sets of integer variables. Now, token[i] or play[i] can only be the integer between 1 and cap. The solver performs better when the value of num is small, but it seems worse for large num.

- OR-Tools

```
token = [model.NewIntVar(-2147483648, 2147483647, 't%i' % i)
         for i in range(1, num + 1)]
play = [model.NewIntVar(-2147483648, 2147483647, 'q%i' % i)
        for i in range(1, num + 1)]

➔ token = [model.NewIntVar(1, cap, 't%i' % i)
           for i in range(1, num + 1)]
  play = [model.NewIntVar(1, cap, 'q%i' % i)
          for i in range(1, num + 1)]
```



The token and play are sets of any int32 integers in the original OR-Tools solver. I let token[i] or play[i] can only be assigned the integer between 1 and cap. The new solver is more efficient with less average runtime.

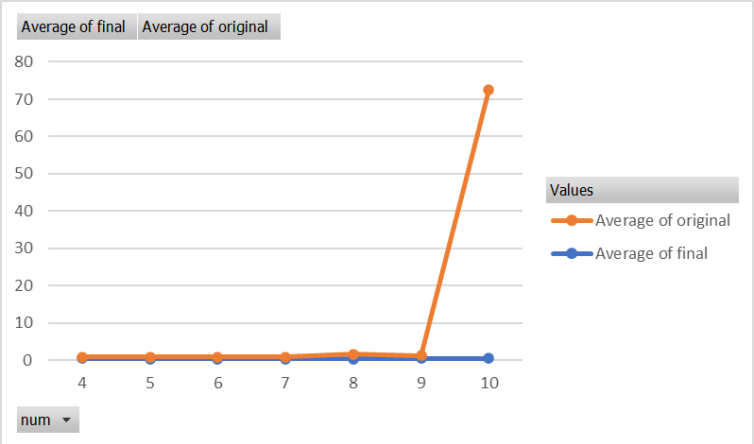
4. The combination of above

Those methods improve our solvers more or less, though it is not absolutely efficient for every instance. Let us combine all of those methods together and apply them to the random instances.

- MiniZinc

file	num	cap	refill	fun	goal	original	reduceVar	Neg	changeRange	final
04_1.dzn	4	9	2	[9, 3, 4, 10]	815	0.49	0.43	0.42	0.37	0.43
04_2.dzn	4	6	5	[7, 6, 4, 9]	107	0.48	0.43	0.44	0.44	0.45
05_1.dzn	5	4	4	[-3, -9, -5, -9, -3]	-61	0.54	0.43	0.44	0.47	0.42
05_2.dzn	5	9	2	[-8, 2, -10, -4, 10]	273	0.49	0.43	0.44	0.4	0.42
06_1.dzn	6	7	6	[8, -9, -3, 3, 6, -6]	816	0.49	0.44	0.42	0.43	0.44
06_2.dzn	6	7	6	[-4, 10, -1, 8, 9, -6]	-1	0.46	0.44	0.46	0.43	0.41
07_1.dzn	7	4	2	[-4, 1, -10, -4, -3, 0, -9]	103	0.47	0.44	0.45	0.39	0.39
07_2.dzn	7	6	1	[-7, -8, -2, 8, 8, 1, -7]	524	0.46	0.48	0.41	0.35	0.41
08_1.dzn	8	6	4	[-7, 9, 8, 8, -9, 6, -1, -8]	-53	0.46	0.42	0.48	0.43	0.41
08_2.dzn	8	10	8	[6, -6, -3, 7, 0, -10, 2, 7]	544	1.96	0.44	0.45	0.36	0.43
09_1.dzn	9	7	4	[-9, -2, -4, 2, 4, -1, 10, 0, 3]	511	1.11	0.46	0.43	0.37	0.43
09_2.dzn	9	7	1	[-1, 2, 3, 2, 3, 0, 7, -9, -10]	615	0.5	0.48	0.45	0.38	0.44
10_1.dzn	10	10	1	[-6, 3, 0, 6, -8, 3, 6, 9, -5, -6]	563	143	0.46	0.52	0.35	0.47
10_2.dzn	10	9	3	[-8, 4, 1, 1, -7, -5, 4, -5, -10, 8]	-42	1.37	0.49	0.5	0.47	0.44

num	Average of original	Average of reduceVar	Average of Neg	Average of changeRange	Average of final
4	0.485	0.43	0.43	0.405	0.44
5	0.515	0.43	0.44	0.435	0.42
6	0.475	0.44	0.44	0.43	0.425
7	0.465	0.46	0.43	0.37	0.4
8	1.21	0.43	0.465	0.395	0.42
9	0.805	0.47	0.44	0.375	0.435
10	72.19	0.475	0.51	0.41	0.455
Grand Total	**10.87785714**	**0.447857143**	**0.450714286**	**0.402857143**	**0.427857143**

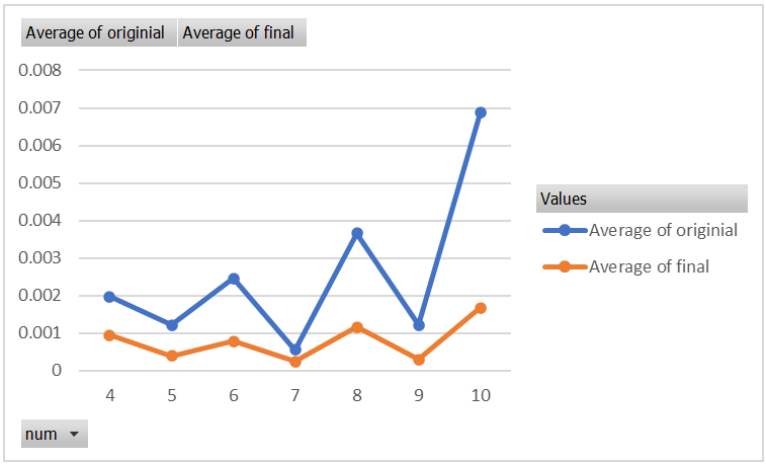


After applying the combination of optimizations, the MiniZinc solver is efficient. And for the instance 10_1, applying any method mentioned before can reduce the unreasonable time.

- OR-Tools

file	num	cap	refill	fun	goal	originial	reduceVar	Neg	changeRange	final
04_1.dzn	4	9	2	[9, 3, 4, 10]	815	0.001517	0.000397	0.000856	0.0001509	0.000189
04_2.dzn	4	6	5	[7, 6, 4, 9]	107	0.002428	0.003615	0.001525	0.0013611	0.001711
05_1.dzn	5	4	4	[-3, -9, -5, -9, -3]	-61	0.001048	0.001496	0.000412	0.0007648	0.000489
05_2.dzn	5	9	2	[-8, 2, -10, -4, 10]	273	0.001382	0.000437	0.000851	0.0002665	0.000306
06_1.dzn	6	7	6	[8, -9, -3, 3, 6, -6]	816	0.002251	0.000502	0.000999	0.0002386	0.00025
06_2.dzn	6	7	6	[-4, 10, -1, 8, 9, -6]	-1	0.002665	0.004717	0.001311	0.0028435	0.001325
07_1.dzn	7	4	2	[-4, 1, -10, -4, -3, 0, -9]	103	0.000332	0.000302	0.000287	0.0002216	0.00025
07_2.dzn	7	6	1	[-7, -8, -2, 8, 8, 1, -7]	524	0.000773	0.000409	0.00181	0.0002301	0.000252
08_1.dzn	8	6	4	[-7, 9, 8, 8, -9, 6, -1, -8]	-53	0.005848	0.005518	0.001922	0.0053517	0.001984
08_2.dzn	8	10	8	[6, -6, -3, 7, 0, -10, 2, 7]	544	0.001463	0.000669	0.001234	0.0003274	0.000335
09_1.dzn	9	7	4	[-9, -2, -4, 2, 4, -1, 10, 0, 3]	511	0.001549	0.000644	0.001311	0.0002901	0.00032
09_2.dzn	9	7	1	[-1, 2, 3, 2, 3, 0, 7, -9, -10]	615	0.000899	0.000448	0.000793	0.000281536	0.000279
10_1.dzn	10	10	1	[-6, 3, 0, 6, -8, 3, 6, 9, -5, -6]	563	0.001079	0.000666	0.001051	0.000340525	0.0004
10_2.dzn	10	9	3	[-8, 4, 1, 1, -7, -5, 4, -5, -10, 8]	-42	0.012663	0.008575	0.002741	0.012473708	0.00296

num	Average of originial	Average of reduceVar	Average of Neg	Average of changeRange	Average of final
4	0.00197242	0.00200605	0.00119035	0.000756	0.00095035
5	0.001214928	0.0009664	0.00063165	0.00051565	0.00039765
6	0.00245784	0.0026096	0.00115475	0.00154105	0.00078725
7	0.000552442	0.0003554	0.00104845	0.00022585	0.00025125
8	0.003655435	0.00309365	0.0015776	0.00283955	0.0011596
9	0.001223766	0.000546412	0.001052094	0.000285818	0.000299219
10	0.006870581	0.004620841	0.001896077	0.006407117	0.001680242
Grand Total	**0.002563916**	**0.002028336**	**0.001221567**	**0.001795862**	**0.000789366**



The OR-Tools solver is most efficient when applying the combination of optimizations, compared with only applying a single method.

I collect the hard instances generated when I use random_instances function, which can be solved by the original version of OR-Tools solver, but take too much time to run in the original version of MiniZinc solver.

file	num	cap	refill	fun	goal
hard_1.dzn	10	10	6	[6, 5, 8, 1, 4, -4, 7, 7, 1, -9]	526
hard_2.dzn	10	9	8	[5, -9, -7, 6, 8, 9, 5, -10, -8, 10]	504
hard_3.dzn	10	10	6	[3, 9, 7, 7, -7, -2, 4, 3, 1, -8]	684
hard_4.dzn	10	10	9	[4, 7, 6, 10, -5, 1, 6, -7, 5, 5]	200
hard_5.dzn	10	9	8	[7, 6, 0, 0, 6, 1, 3, 7, 1, 5]	244
hard_6.dzn	10	10	9	[-3, 4, 9, -1, -9, -1, 0, 8, -3, 1]	100
hard_7.dzn	10	10	9	[-3, 7, -3, 10, -4, -6, 3, 5, -6, -9]	755

But they can be solve very efficiently using the optimized version of MiniZinc.

file	num	cap	refill	fun	goal	minizinc sat	minizinc token	minizinc play	minizinc total_fun	minizinc time
hard_1.dzn	10	10	6	[6, 5, 8, 1, 4, -4, 7, 7, 1, -9]	526	UNSATISFIABLE				0.45
hard_2.dzn	10	9	8	[5, -9, -7, 6, 8, 9, 5, -10, -8, 10]	504	UNSATISFIABLE				0.44
hard_3.dzn	10	10	6	[3, 9, 7, 7, -7, -2, 4, 3, 1, -8]	684	UNSATISFIABLE				0.43
hard_4.dzn	10	10	9	[4, 7, 6, 10, -5, 1, 6, -7, 5, 5]	200	OPTIMAL_SOLUTION	[10, 10, 9, 10, 9, 10, 10, 9, 10, 10]	[9, 10, 8, 10, 1, 9, 10, 1, 9, 10]	406	0.48
hard_5.dzn	10	9	8	[7, 6, 0, 0, 6, 1, 3, 7, 1, 5]	244	OPTIMAL_SOLUTION	[9, 8, 8, 8, 9, 8, 9, 9, 8, 9]	[9, 8, 8, 1, 9, 7, 8, 9, 7, 9]	311	0.46
hard_6.dzn	10	10	9	[-3, 4, 9, -1, -9, -1, 0, 8, -3, 1]	100	OPTIMAL_SOLUTION	[10, 10, 10, 9, 10, 10, 10, 10, 9, 10]	[1, 9, 10, 1, 1, 1, 1, 10, 1, 10]	199	0.45
hard_7.dzn	10	10	9	[-3, 7, -3, 10, -4, -6, 3, 5, -6, -9]	755	UNSATISFIABLE				0.46

Future study

I follow the description from The 2015 LP/CP Programming Contest, that $4 \leq num \leq 10$, $3 \leq cap \leq 10$, $0 < refill \leq cap$, $-10 \leq fun[i] \leq 10$ and $-100 \leq goal \leq 1000$. The range of variables of input instances are limited. And there are many other methods to obtain better performance. The solvers can be more efficient.