CMPT417 Final Project Report Sequential Games

Andy Ng
301221017

Andy Ng
301221017

Introduction

The target question of this project is Sequential Games provided by the LP/CP Programming Contest 2015. In the question, we have a series of games and each game has a positive or negative "fun" value associated with it. Each game must be played at least once, and they must be played in order. The games are not free and each game costs 1 token to play once. Your pocket can contain a max capacity of tokens and before each playing each game, you can refill a certain number of tokens. However, the number of refilled tokens cannot exceed the capacity. The goal of this question is to find a sequence of games that is higher than K and maximizes the total fun.

Solvers

We used 2 solvers for this project, MiniZinc IDE and the IDP system. We chose these two systems because they have tools for solving satisfiability problems, such as support for propositional logic, solve maximize and solve satisfy. We worked on both writing the specifications based off the notes provided by Prof. Mitchell as well as optimizing the solution in Part 2. For Part 1, We used Gecode as the solver for MiniZinc. The settings used to MiniZinc and IDP in Part 1 are the default settings.

Specifications

MiniZinc:

For this problem, **fun** is a unary function, while **K, num, cap** and **refill** are constants. The unary function **plays** contains the solutions. **Tokens** is a unary function that represents the number of tokens in our pocket at the start of each game. Our solution is **Total_fun**, which is represented by `sum(i in games)(plays[i]*fun[i])`

```
% Sequential Games variables:
int: K;                         % Goal K ∈ N;
int: num;                       %  Number n ∈ N of games;
set of int: games = 1..num;
array [games] of int: fun;      % Fun value vi ∈ Z for each game i ∈ [n];
int: cap;                       % Pocket Capacity C ∈ N;
int: refill;                    % Refill amount R ∈ N;
set of int: x_range = 0..(cap + refill);

array[games] of var int: tokens;
array[games] of var int: plays;
```

The first constraint is **The sum of Pi and Vi >= K .** This requires **Total_fun** to be larger than **K** for the test case to be satisfiable.

```
% 1. ∑i∈[n] pivi ≥ K:
constraint sum(i in games) (plays[i] * fun[i]) >= K;
```

Andy Ng
301221017

The second constraint is **The number of tokens t(i) available to play game i is C before the first game and for i >1 min{C, t(i-1) - P(i-1) +R}**. This is represented in the form of:

$$t(1)=C \wedge \forall i \left[1 < i \leq n \rightarrow \exists x \left((x = t(i-1) - p(i-1) + R) \wedge (x > C \rightarrow t(i) = C) \wedge (x \leq C \rightarrow t(i) = x) \right] \right]$$

```
% 2. The number of tokens t(i) available to play game i is C before the first game and for i >1 min{C, t(i-1) -
P(i-1) +R}:
constraint (tokens[1] = cap) /\
forall(i in games)(
  (1<i /\ i<=num) -> exists(x in x_range)(
    (x = (tokens[i-1] - plays[i-1] + refill)) /\
    ((x > cap) -> (tokens[i] = cap)) /\
    ((x <= cap) -> (tokens[i] = x))
  )
);
```

The third constraint is **We play each game at least once, at most t(i) times.** This is represented by:

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

```
% 3. We play each game at least once, and at most t(i) times
constraint forall (i in games)((1 <= i /\ i <= num) -> (1 <= plays[i] /\ plays[i] <= tokens[i]));
```

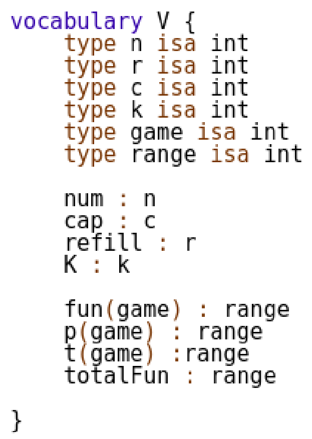
For the result, we find the maximum Total_fun in our solutions.

```
% Result:
solve maximize sum(i in games)(plays[i] * fun[i]);
```

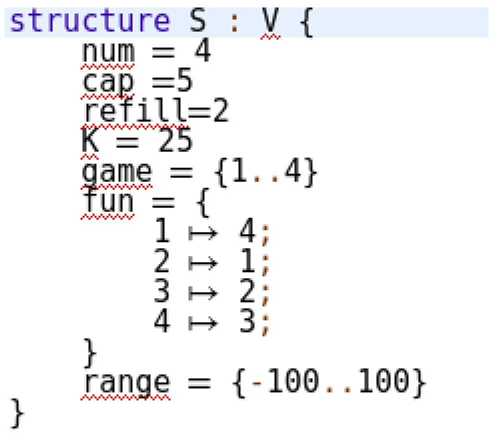
Andy Ng
301221017

IDP:

For IDP, we declare a vocabulary for constants and functions, and fill out a structure with test case values.

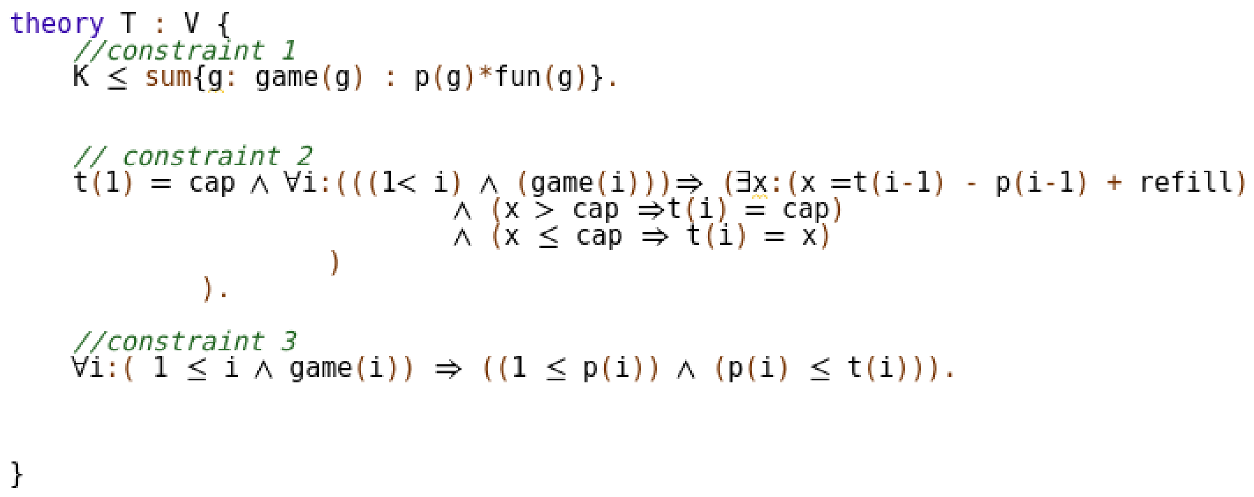


```
vocabulary V {
    type n isa int
    type r isa int
    type c isa int
    type k isa int
    type game isa int
    type range isa int

    num : n
    cap : c
    refill : r
    K : k

    fun(game) : range
    p(game) : range
    t(game) :range
    totalFun : range
}
```



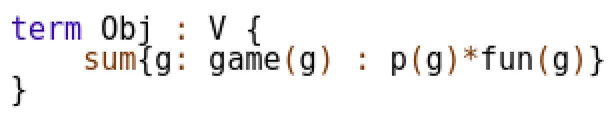
```
structure S : V {
     num = 4
     cap =5
     refill=2
     K = 25
     game = {1..4}
     fun = {
          1 ↦ 4;
          2 ↦ 1;
          3 ↦ 2;
          4 ↦ 3;
     }
     range = {-100..100}
}
```

For the constraints in IDP, we use the same constraints as MiniZinc and place the specifications in theory:

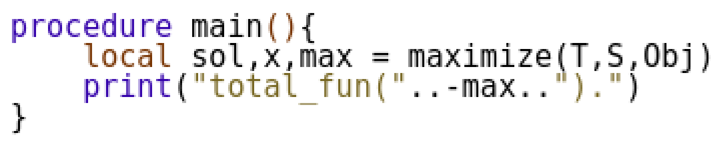


```
theory T : V {
     //constraint 1
     K ≤ sum{g: game(g) : p(g)*fun(g)}.


     // constraint 2
     t(1) = cap ∧ ∀i:(((1< i) ∧ (game(i)))⇒ (∃x:(x =t(i-1) - p(i-1) + refill)
                              ∧ (x > cap ⇒t(i) = cap)
                              ∧ (x ≤ cap ⇒ t(i) = x)
                         )
               ).

     //constraint 3
     ∀i:( 1 ≤ i ∧ game(i)) ⇒ ((1 ≤ p(i)) ∧ (p(i) ≤ t(i))).


}
```

For the solution, the obj represents Total_fun and we maximize to solve the question:



```
term Obj : V {
     sum{g: game(g) : p(g)*fun(g)}
}
```



```
procedure main(){
     local sol,x,max = maximize(T,S,Obj)
     print("total_fun("..-max..").")
}
```

Andy Ng
301221017

Test Cases:

test case	num	cap	refill	fun	K	totalFun
1	4	5	2	[4,1,2,3]	25	35
2	4	5	2	[4,-1,-2,3]	25	29
3	5	3	2	[4,1,-2,3,4]	20	30
4	6	4	2	[4,1,2,-3,-2,-1]	25	22
5	7	5	4	[8,1,1,-3,2,6,-4]	40	79
6	8	8	2	[-5,3,7,6,-2,1,2,-1]	55	75
7	8	8	6	[4,7,-5,3,-5,-7,4,9]	110	183
8	10	6	6	[-4,1,2,-3,1,2,-4,5,1,-2]	60	59
9	10	8	5	[-5,2,1,-3,6,3,8,-5,-1,3]	200	149
10	10	6	7	[6,-3,2,4,3,-5,-9,3,4,-6]	150	109
11	12	7	5	[3,2,5,-4,6,-7,-3,5,4,2,1,-4]	150	156
12	11	9	5	[3,2,-4,2,-3,4,-6,5,-8,3,-9]	150	109
13	10	7	3	[2,1,2,3,4,2,4,5,2,1]	100	104
14	10	8	5	[3,1,1,2,6,3,4,5,3,2]	100	180
15	15	10	7	[3,1,-1,2,6,3,-4,5,3,-2,4,-5,7,6,-9]	100	334
16	8	8	6	[-4,-7,-5,-3,-5,-7,-4,-9]	110	0

For this project, we're using 16 test cases in Part 1 for uniformity between MiniZinc and IDP. Test cases 1 to 3 are taken from the LP/CP Programming Contest page. Test cases 4 to 14 are standard test cases with increasing values in **num, cap and refill**. Test case 15 is a special test case where the num is abnormally large and Test 16 is the only test case with all negative values (guaranteed to be UNSAT). Half the test cases are SAT and other case are UNSAT.

Andy Ng
301221017

Performance

The performance of both solvers is calculated using the average obtained from 3 instances of the **time** command. Real-life time is used, and the time is recorded in seconds.

MiniZinc:

Minizinc Gecode without constraint optimization

test case	num	cap	refill	fun	K	totalFun	time 1	time 2	time3	avg(time)	SAT
1	4	5	2	[4,1,2,3]	25	35	0.095	0.09	0.094	0.093	SAT
2	4	5	2	[4,-1,-2,3]	25	29	0.098	0.091	0.095	0.094667	SAT
3	5	3	2	[4,1,-2,3,4]	20	30	0.097	0.091	0.093	0.093667	SAT
4	6	4	2	[4,1,2,-3,-2,-1]	25	22	0.097	0.095	0.092	0.094667	UNSAT
5	7	5	4	[8,1,1,-3,2,6,-4]	40	79	0.105	0.106	0.109	0.106667	SAT
6	8	8	2	[-5,3,7,6,-2,1,2,-1]	55	75	0.126	0.122	0.148	0.132	SAT
7	8	8	6	[4,7,-5,3,-5,-7,4,9]	110	183	1.396	1.453	1.475	1.441333	SAT
8	10	6	6	[-4,1,2,-3,1,2,-4,5,1,-2]	60	59	16.725	16.769	16.749	16.74767	UNSAT
9	10	8	5	[-5,2,1,-3,6,3,8,-5,-1,3]	200	149	6.975	7.23	7.18	7.128333	UNSAT
10	10	6	7	[6,-3,2,4,3,-5,-9,3,4,-6]	150	109	4.569	4.636	4.564	4.589667	UNSAT
11	12	7	5	[3,2,5,-4,6,-7,-3,5,4,2,1,-4]	150	156	1200	1246	1167	1204.333	SAT
12	11	9	5	[3,2,-4,2,-3,4,-6,5,-8,3,-9]	150	109	199	200	197	198.6667	UNSAT
13	10	7	3	[2,1,2,3,4,2,4,5,2,1]	100	104	3.868	3.874	3.854	3.865333	SAT
14	10	8	5	[3,1,1,2,6,3,4,5,3,2]	100	180	499	501	496	498.6667	SAT
15	15	10	7	[3,1,-1,2,6,3,-4,5,3,-2,4,-5,7,6,-9]	100	334	16166	n/a	n/a	16166	SAT
16	8	8	6	[-4,-7,-5,-3,-5,-7,-4,-9]	110	0	0.096	0.094	0.092	0.094	UNSAT

Using Gecode, the default solver configuration in MiniZinc, we obtained the average time by running each test case 3 times and averaging the time in seconds. For test cases with smaller num's, the average time is quite short, usually less than a second. However, we can see that starting from num =10, the solver runtime gets exponentially higher. For example, test case 11 is num = 12 and took 20 minutes to complete, while test case 15 was num = 15 and took over 260 mins until the tester decided to terminate the program. From the data, it seems that after num = 12, MiniZinc with the current 3 specifications is not a suitable solver due to the long run times. There does not appear to be a correlation between runtime and the satisfiability of the test case. As expected, if the player has a higher cap and refill, then the total_fun will also be higher because of more tokens available.

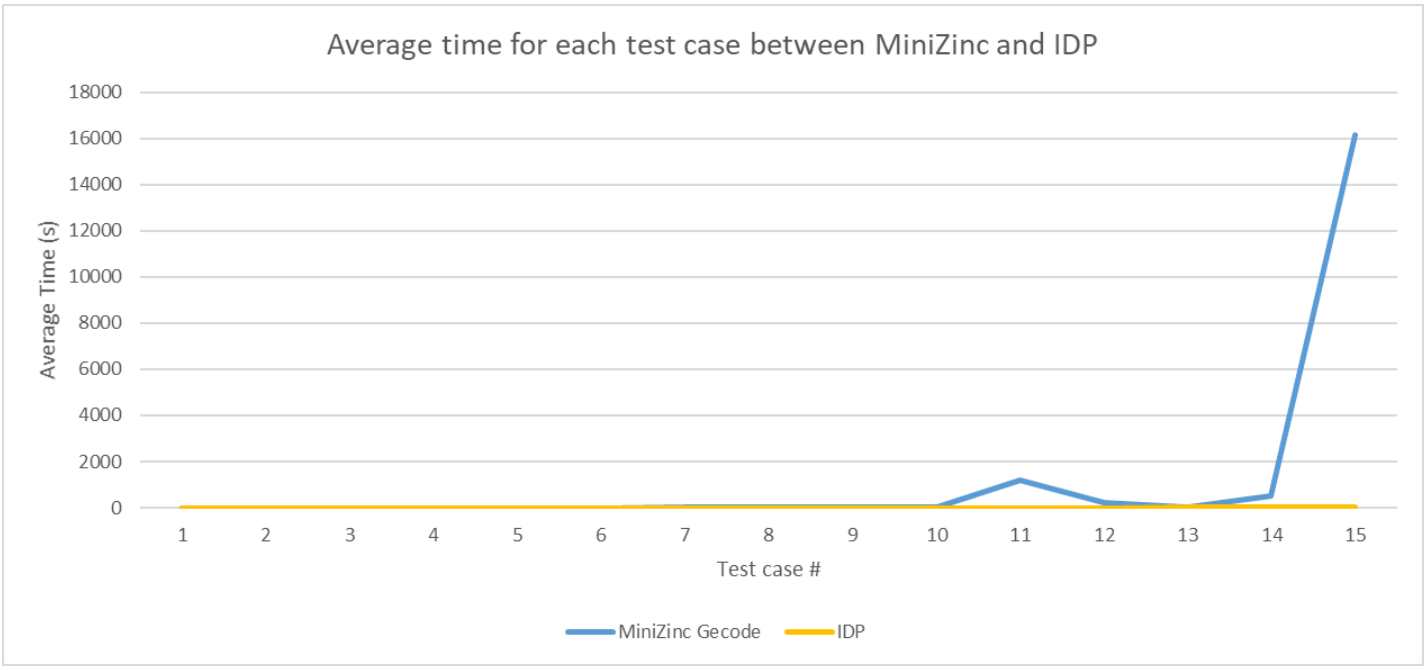
Andy Ng
301221017

IDP:

IDP

test case	num	cap	refill	fun	K	totalFun	time 1	time 2	time3	avg(time)	SAT
1	4	5	2	[4,1,2,3]	25	35	0.208	0.175	0.18	0.187667	SAT
2	4	5	2	[4,-1,-2,3]	25	29	0.193	0.177	0.185	0.185	SAT
3	5	3	2	[4,1,-2,3,4]	20	30	0.177	0.181	0.186	0.181333	SAT
4	6	4	2	[4,1,2,-3,-2,-1]	25	22	0.183	0.184	0.183	0.183333	UNSAT
5	7	5	4	[8,1,1,-3,2,6,-4]	40	79	0.258	0.333	0.239	0.276667	SAT
6	8	8	2	[-5,3,7,6,-2,1,2,-1]	55	75	0.262	0.239	0.245	0.248667	SAT
7	8	8	6	[4,7,-5,3,-5,-7,4,9]	110	183	0.452	0.396	0.403	0.417	SAT
8	10	6	6	[-4,1,2,-3,1,2,-4,5,1,-2]	60	59	0.26	0.239	0.33	0.276333	UNSAT
9	10	8	5	[-5,2,1,-3,6,3,8,-5,-1,3]	200	149	0.805	0.816	0.79	0.803667	UNSAT
10	10	6	7	[6,-3,2,4,3,-5,-9,3,4,-6]	150	109	0.337	0.315	0.311	0.321	UNSAT
11	12	7	5	[3,2,5,-4,6,-7,-3,5,4,2,1,-4]	150	122	0.484	0.4	0.393	0.425667	UNSAT
12	11	9	5	[3,2,-4,2,-3,4,-6,5,-8,3,-9]	150	109	0.311	0.396	0.403	0.37	UNSAT
13	10	7	3	[2,1,2,3,4,2,4,5,2,1]	100	104	4.104	4.128	4.1	4.110667	SAT
14	10	8	5	[3,1,1,2,6,3,4,5,3,2]	100	180	1.981	1.992	1.999	1.990667	SAT
15	15	10	7	[3,1,-1,2,6,3,-4,5,3,-2,4,-5,7,6,-9]	100	334	2.59	2.584	2.584	2.586	SAT
16	8	8	6	[-4,-7,-5,-3,-5,-7,-4,-9]	110	0	0.182	0.179	0.178	0.179667	UNSAT

In contrast, IDP's solving times are slower by 2x for test cases 1 to 7, all tests with smaller num's that performed very well for MiniZinc. However, for test cases 8 to 15, IDP performs more than a thousand times better than MiniZinc. The most extreme example is test case 15, where it took IDP 2.58s to complete on average as opposed to MiniZinc's 16166+s. Another example is test case 11, where IDP took 0.4s as opposed to MiniZinc's 1204s. The graph below illustrates how IDP maintains it's low average times across different test cases with increasing num's, whereas Minizinc's average times skyrocket as num >= 12. Once again, there does not appear to be a correlation between runtime and satisfiability of the test case.



Andy Ng
301221017

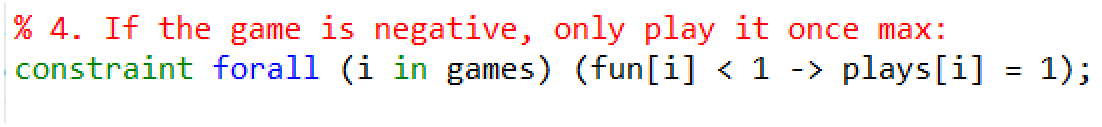
Questions/Optimizations

For part 2, we ask questions about ways and areas to optimize our program. While the program works and we obtain the correct answer in terms of maximum fun out of possible solutions, it is evident that there is a lot of room for improvement in the run time for our program. Getting the correct result will be unimportant if it requires hours for our program to solve test cases with **num** > 15. Currently the performance between MiniZinc Gecode and IDP is incomparable. We will focus on optimizing the MiniZinc program with extra/redundant constraints and improving solver configuration choice. Out goal would be to make MiniZinc comparable to IDP in terms of runtime for the same test sets.

Question 1: Can we improve how the program handles negative games?

Experiment:

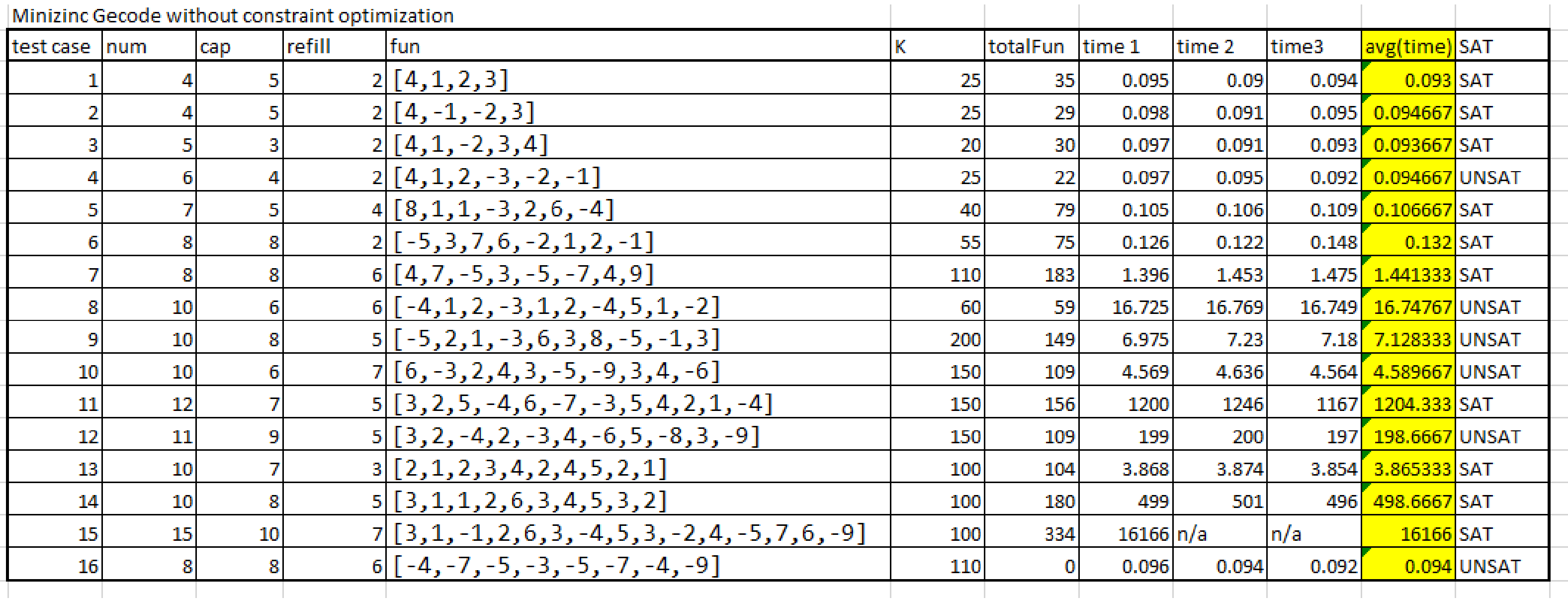
In most test cases, there will be games with negative values that are detrimental to the total fun of the player. Since we are forced to play every game at least once, we could write a constraint that limits the plays of games with negative fun to just 1. This constraint is redundant as it doesn't change the outcome or correctness of our solution.



```
% 4. If the game is negative, only play it once max:
constraint forall (i in games) (fun[i] < 1 -> plays[i] = 1);
```

Now, instead of testing instances with multiple plays of negative games, all negative fun games will be played only once. We hypothesize that this constraint should save resources and allow the program to solve the problem faster.

Discussion:



Minizinc Gecode without constraint optimization

test case	num	cap	refill	fun	K	totalFun	time 1	time 2	time3	avg(time)	SAT
1	4	5	2	[4,1,2,3]	25	35	0.095	0.09	0.094	0.093	SAT
2	4	5	2	[4,-1,-2,3]	25	29	0.098	0.091	0.095	0.094667	SAT
3	5	3	2	[4,1,-2,3,4]	20	30	0.097	0.091	0.093	0.093667	SAT
4	6	4	2	[4,1,2,-3,-2,-1]	25	22	0.097	0.095	0.092	0.094667	UNSAT
5	7	5	4	[8,1,1,-3,2,6,-4]	40	79	0.105	0.106	0.109	0.106667	SAT
6	8	8	2	[-5,3,7,6,-2,1,2,-1]	55	75	0.126	0.122	0.148	0.132	SAT
7	8	8	6	[4,7,-5,3,-5,-7,4,9]	110	183	1.396	1.453	1.475	1.441333	SAT
8	10	6	6	[-4,1,2,-3,1,2,-4,5,1,-2]	60	59	16.725	16.769	16.749	16.74767	UNSAT
9	10	8	5	[-5,2,1,-3,6,3,8,-5,-1,3]	200	149	6.975	7.23	7.18	7.128333	UNSAT
10	10	6	7	[6,-3,2,4,3,-5,-9,3,4,-6]	150	109	4.569	4.636	4.564	4.589667	UNSAT
11	12	7	5	[3,2,5,-4,6,-7,-3,5,4,2,1,-4]	150	156	1200	1246	1167	1204.333	SAT
12	11	9	5	[3,2,-4,2,-3,4,-6,5,-8,3,-9]	150	109	199	200	197	198.6667	UNSAT
13	10	7	3	[2,1,2,3,4,2,4,5,2,1]	100	104	3.868	3.874	3.854	3.865333	SAT
14	10	8	5	[3,1,1,2,6,3,4,5,3,2]	100	180	499	501	496	498.6667	SAT
15	15	10	7	[3,1,-1,2,6,3,-4,5,3,-2,4,-5,7,6,-9]	100	334	16166	n/a	n/a	16166	SAT
16	8	8	6	[-4,-7,-5,-3,-5,-7,-4,-9]	110	0	0.096	0.094	0.092	0.094	UNSAT

Andy Ng
301221017

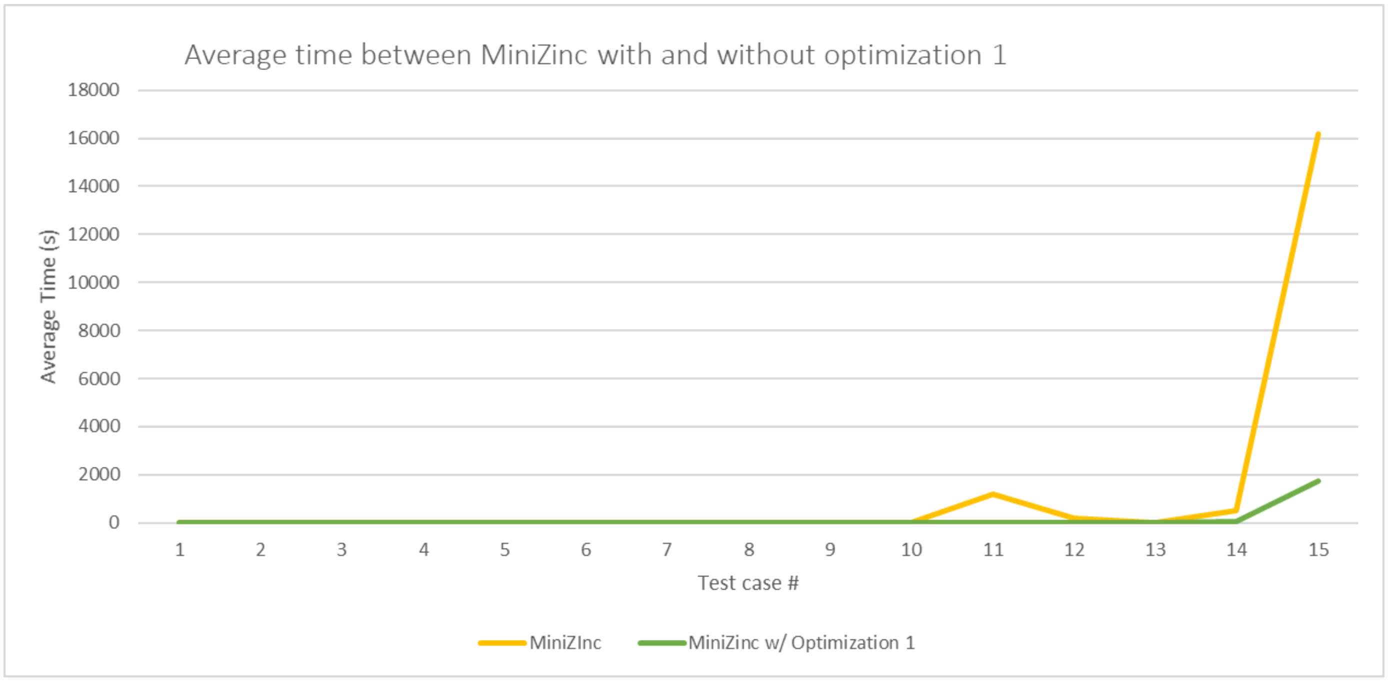
Minizinc Gecode with negative fun optimization

test case	num	cap	refill	fun	K	totalFun	time 1	time 2	time3	avg(time)	SAT
1	4	5	2	[4,1,2,3]	25	35	0.107	0.093	0.104	0.101333	SAT
2	4	5	2	[4,-1,-2,3]	25	29	0.094	0.094	0.106	0.098	SAT
3	5	3	2	[4,1,-2,3,4]	20	30	0.095	0.089	0.092	0.092	SAT
4	6	4	2	[4,1,2,-3,-2,-1]	25	22	0.097	0.095	0.092	0.094667	UNSAT
5	7	5	4	[8,1,1,-3,2,6,-4]	40	79	0.105	0.102	0.113	0.106667	SAT
6	8	8	2	[-5,3,7,6,-2,1,2,-1]	55	75	0.117	0.106	0.104	0.109	SAT
7	8	8	6	[4,7,-5,3,-5,-7,4,9]	110	183	0.112	0.114	0.116	0.114	SAT
8	10	6	6	[-4,1,2,-3,1,2,-4,5,1,-2]	60	59	0.113	0.118	0.133	0.121333	UNSAT
9	10	8	5	[-5,2,1,-3,6,3,8,-5,-1,3]	200	149	0.131	0.128	0.128	0.129	UNSAT
10	10	6	7	[6,-3,2,4,3,-5,-9,3,4,-6]	150	109	0.113	0.115	0.114	0.114	UNSAT
11	12	7	5	[3,2,5,-4,6,-7,-3,5,4,2,1,-4]	150	156	0.806	0.81	0.766	0.794	SAT
12	11	9	5	[3,2,-4,2,-3,4,-6,5,-8,3,-9]	150	109	0.149	0.145	0.142	0.145333	UNSAT
13	10	7	3	[2,1,2,3,4,2,4,5,2,1]	100	104	3.65	3.8	3.75	3.733333	SAT
14	10	8	5	[3,1,1,2,6,3,4,5,3,2]	100	180	49.6	51.9	49.6	50.36667	SAT
15	15	10	7	[3,1,-1,2,6,3,-4,5,3,-2,4,-5,7,6,-9]	100	334	1716	n/a	n/a	1716	SAT
16	8	8	6	[-4,-7,-5,-3,-5,-7,-4,-9]	110	0	0.055	0.054	0.054	0.054333	UNSAT

With this new constraint, we can see clear improvements in run time for test cases with higher num values. Test cases 8 to 11 have been cut down to less than 1 second. Test 11 (1204s -> 0.794s), test 12 (198.7s -> 0.145s), test 14 (498.7s -> 50.4s) and test 15 (16166s -> 1716s) were the most notable decreases. In addition to the significant decrease in runtime for test 15, the program succeeded in completing the solving process after this constraint was added, making it a key improvement for the functionality of the program despite its redundancy. The graph below shows that this constraint drastically lowered the runtimes for test cases 10 to 15.

ANSWER:

The 1st optimization reduces the runtime for MiniZinc and the results are significant for test cases with high num values and multiple negative numbers.



Andy Ng
301221017

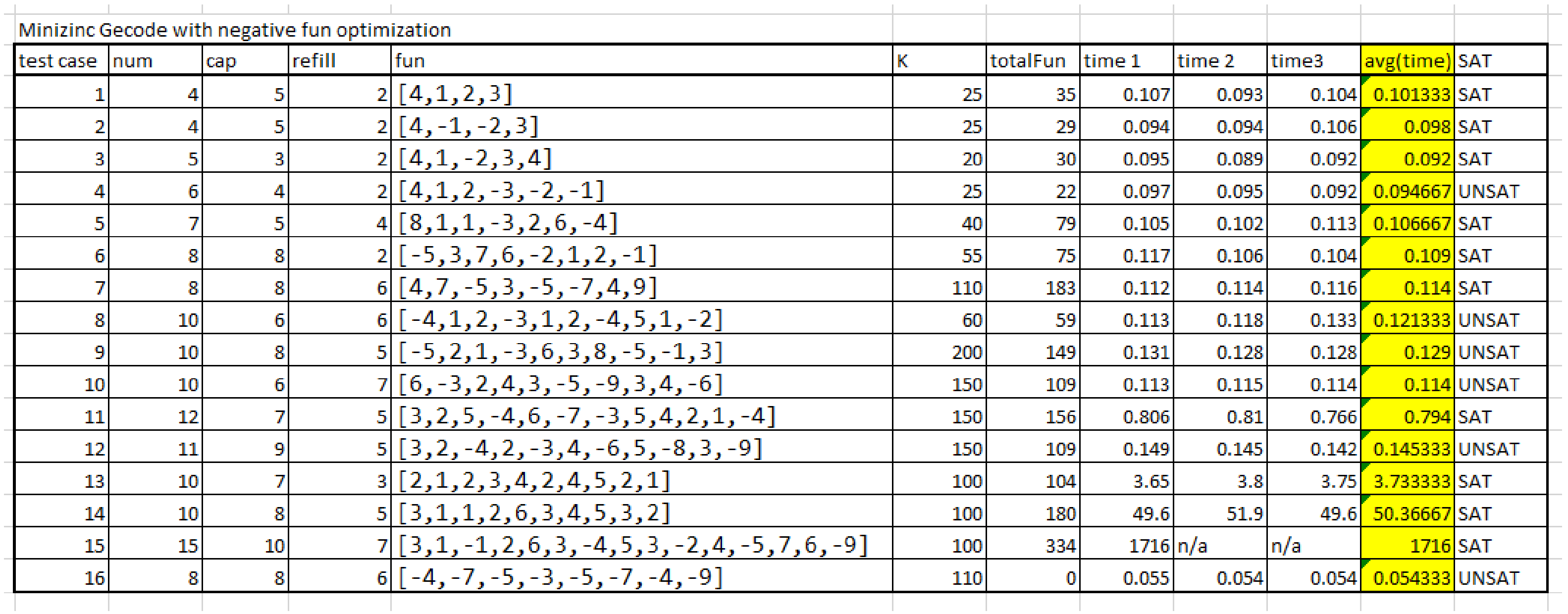
Question 2: Can we make sure more fun games are played more than less fun games?

Experiment:

Essentially, this question is trying to add a redundant constraint to skip instances where less fun games are played more than more fun games for the sake of brute forcing through every possible solution. If we check beforehand that a game is more fun than the game before it or vice versa, we can add a condition that the plays of the less fun game does not exceed those of the more fun game. This will ensure that we play more fun games more than less fun games.

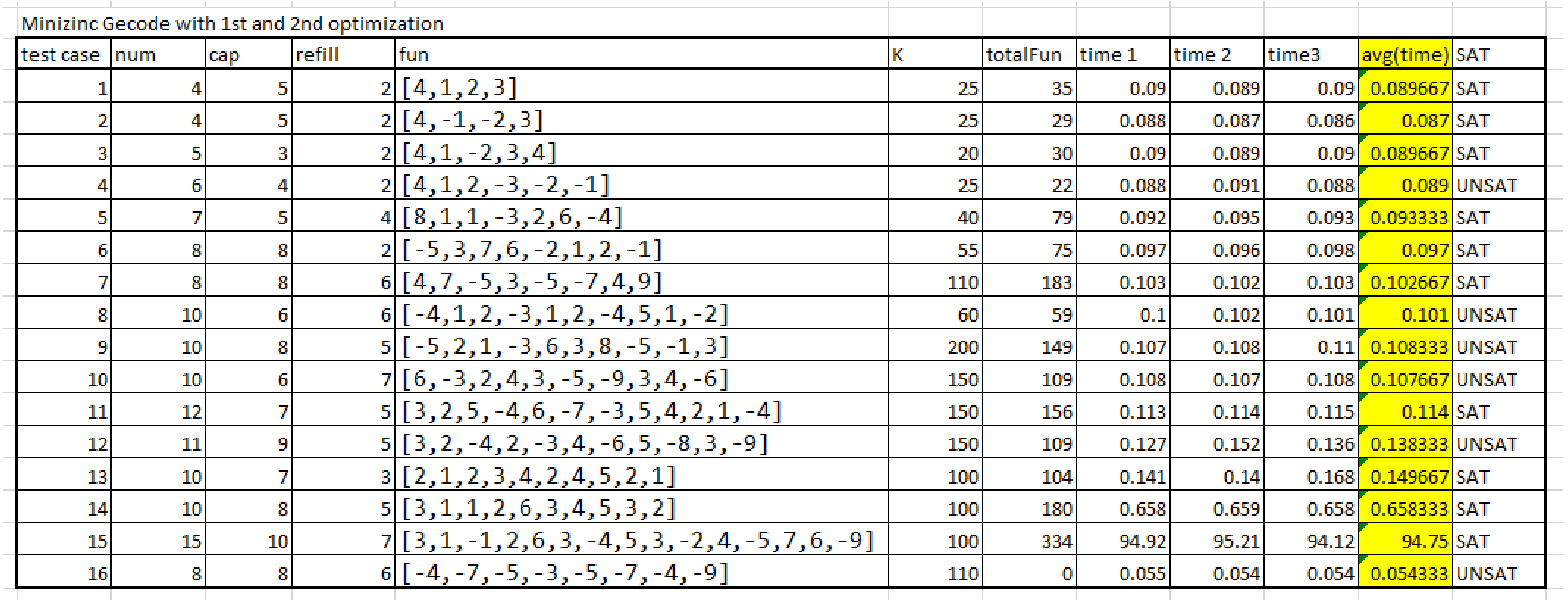
```
% 5. We play fun games no less than less fun games
constraint forall (i in games)(i > 1 /\ fun[i] > fun[i-1] -> plays[i] >= plays[i-1]);
constraint forall (i in games)(i > 1 /\ fun[i] < fun[i-1] -> plays[i] <= plays[i-1]);
```

The constraint we wrote will check the fun values of games i and i-1, making sure that if i has a higher fun value than i-1, i will be played more times. The opposite is true as well, i will be ensured to be played less times than i-1 if i has a lesser fun value.



Minizinc Gecode with negative fun optimization

test case	num	cap	refill	fun	K	totalFun	time 1	time 2	time3	avg(time)	SAT
1	4	5	2	[4,1,2,3]	25	35	0.107	0.093	0.104	0.101333	SAT
2	4	5	2	[4,-1,-2,3]	25	29	0.094	0.094	0.106	0.098	SAT
3	5	3	2	[4,1,-2,3,4]	20	30	0.095	0.089	0.092	0.092	SAT
4	6	4	2	[4,1,2,-3,-2,-1]	25	22	0.097	0.095	0.092	0.094667	UNSAT
5	7	5	4	[8,1,1,-3,2,6,-4]	40	79	0.105	0.102	0.113	0.106667	SAT
6	8	8	2	[-5,3,7,6,-2,1,2,-1]	55	75	0.117	0.106	0.104	0.109	SAT
7	8	8	6	[4,7,-5,3,-5,-7,4,9]	110	183	0.112	0.114	0.116	0.114	SAT
8	10	6	6	[-4,1,2,-3,1,2,-4,5,1,-2]	60	59	0.113	0.118	0.133	0.121333	UNSAT
9	10	8	5	[-5,2,1,-3,6,3,8,-5,-1,3]	200	149	0.131	0.128	0.128	0.129	UNSAT
10	10	6	7	[6,-3,2,4,3,-5,-9,3,4,-6]	150	109	0.113	0.115	0.114	0.114	UNSAT
11	12	7	5	[3,2,5,-4,6,-7,-3,5,4,2,1,-4]	150	156	0.806	0.81	0.766	0.794	SAT
12	11	9	5	[3,2,-4,2,-3,4,-6,5,-8,3,-9]	150	109	0.149	0.145	0.142	0.145333	UNSAT
13	10	7	3	[2,1,2,3,4,2,4,5,2,1]	100	104	3.65	3.8	3.75	3.733333	SAT
14	10	8	5	[3,1,1,2,6,3,4,5,3,2]	100	180	49.6	51.9	49.6	50.36667	SAT
15	15	10	7	[3,1,-1,2,6,3,-4,5,3,-2,4,-5,7,6,-9]	100	334	1716	n/a	n/a	1716	SAT
16	8	8	6	[-4,-7,-5,-3,-5,-7,-4,-9]	110	0	0.055	0.054	0.054	0.054333	UNSAT



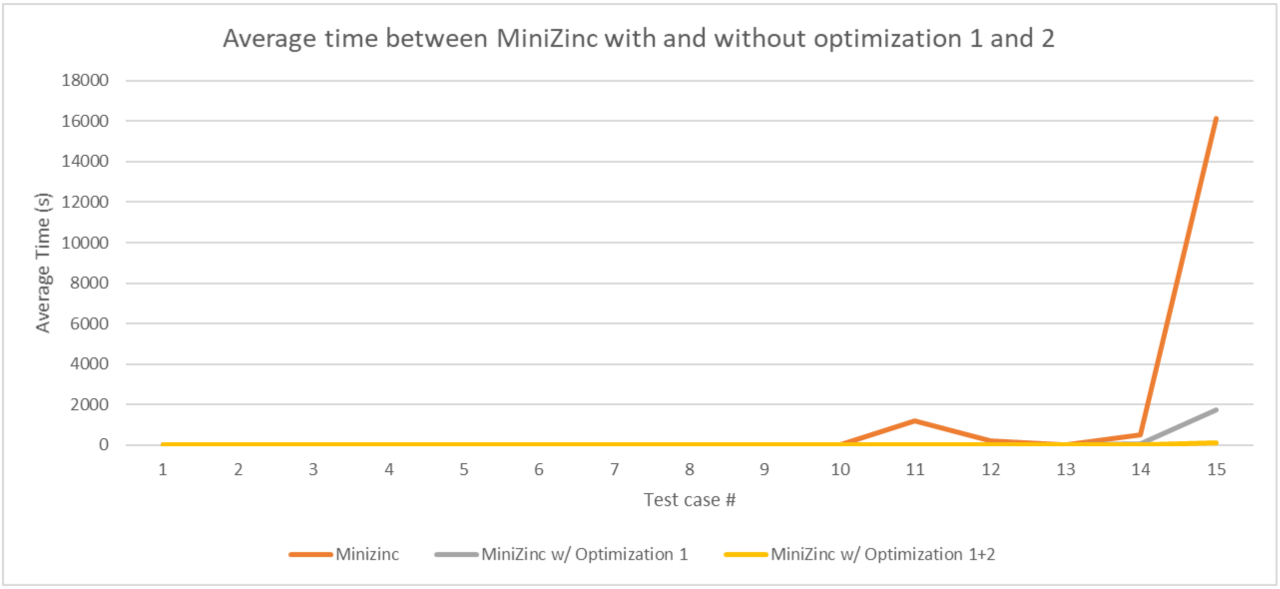
Minizinc Gecode with 1st and 2nd optimization

test case	num	cap	refill	fun	K	totalFun	time 1	time 2	time3	avg(time)	SAT
1	4	5	2	[4,1,2,3]	25	35	0.09	0.089	0.09	0.089667	SAT
2	4	5	2	[4,-1,-2,3]	25	29	0.088	0.087	0.086	0.087	SAT
3	5	3	2	[4,1,-2,3,4]	20	30	0.09	0.089	0.09	0.089667	SAT
4	6	4	2	[4,1,2,-3,-2,-1]	25	22	0.088	0.091	0.088	0.089	UNSAT
5	7	5	4	[8,1,1,-3,2,6,-4]	40	79	0.092	0.095	0.093	0.093333	SAT
6	8	8	2	[-5,3,7,6,-2,1,2,-1]	55	75	0.097	0.096	0.098	0.097	SAT
7	8	8	6	[4,7,-5,3,-5,-7,4,9]	110	183	0.103	0.102	0.103	0.102667	SAT
8	10	6	6	[-4,1,2,-3,1,2,-4,5,1,-2]	60	59	0.1	0.102	0.101	0.101	UNSAT
9	10	8	5	[-5,2,1,-3,6,3,8,-5,-1,3]	200	149	0.107	0.108	0.11	0.108333	UNSAT
10	10	6	7	[6,-3,2,4,3,-5,-9,3,4,-6]	150	109	0.108	0.107	0.108	0.107667	UNSAT
11	12	7	5	[3,2,5,-4,6,-7,-3,5,4,2,1,-4]	150	156	0.113	0.114	0.115	0.114	SAT
12	11	9	5	[3,2,-4,2,-3,4,-6,5,-8,3,-9]	150	109	0.127	0.152	0.136	0.138333	UNSAT
13	10	7	3	[2,1,2,3,4,2,4,5,2,1]	100	104	0.141	0.14	0.168	0.149667	SAT
14	10	8	5	[3,1,1,2,6,3,4,5,3,2]	100	180	0.658	0.659	0.658	0.658333	SAT
15	15	10	7	[3,1,-1,2,6,3,-4,5,3,-2,4,-5,7,6,-9]	100	334	94.92	95.21	94.12	94.75	SAT
16	8	8	6	[-4,-7,-5,-3,-5,-7,-4,-9]	110	0	0.055	0.054	0.054	0.054333	UNSAT

Andy Ng
301221017

After adding our 2nd constraint, the program was further optimized, and the runtimes were decreased against substantially. Notably, test 11 (0.794s -> 0.114), test 13 (3.73s -> 0.150s), test 14 (50.3 -> 0.658s), and test 15 (1716s -> 94.75s). Even after the 1st optimization, test 15 still required ~28.6 mins to complete. After adding the 2nd optimization, test 15 now only requires ~1.57 mins, effectively an 18x reduction.

This graph below shows how unoptimized Minizinc with compares to Minizinc with the 1st and 2nd optimizations over the same set of test cases.



ANSWER:

The 2nd optimization further reduces the runtime for MiniZinc and the results are significant for test cases with high num values. This decrease in runtime is predicted to be due to how the 2nd optimization reduces unnecessary comparisons and instances.

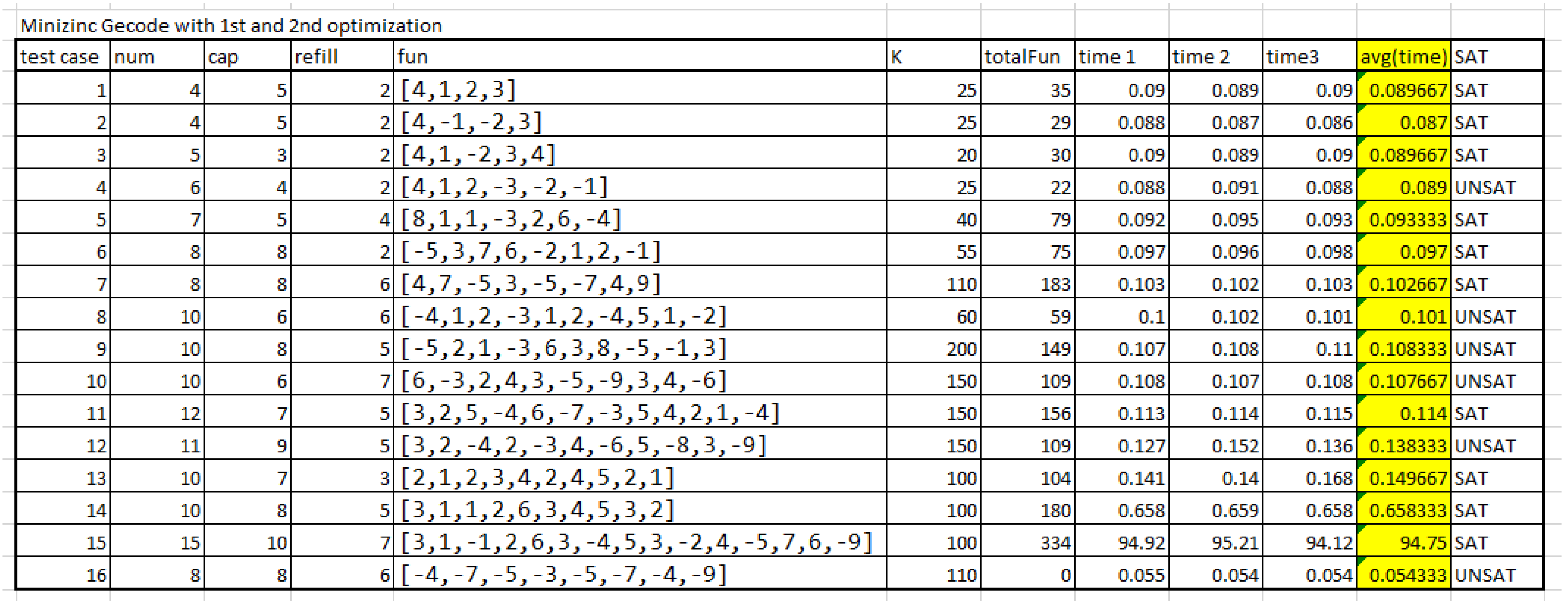
While the runtimes are drastically decreased, we are still not yet at IDP's level of efficiency. After reducing test case 15's runtime from 16166s+ to 94.75s, it is still 47x longer than IDP's 2.6s runtime. What other ways could we optimize MiniZinc?

Andy Ng
301221017

Question 3: Can we change to another solver configuration for a better solving speed?

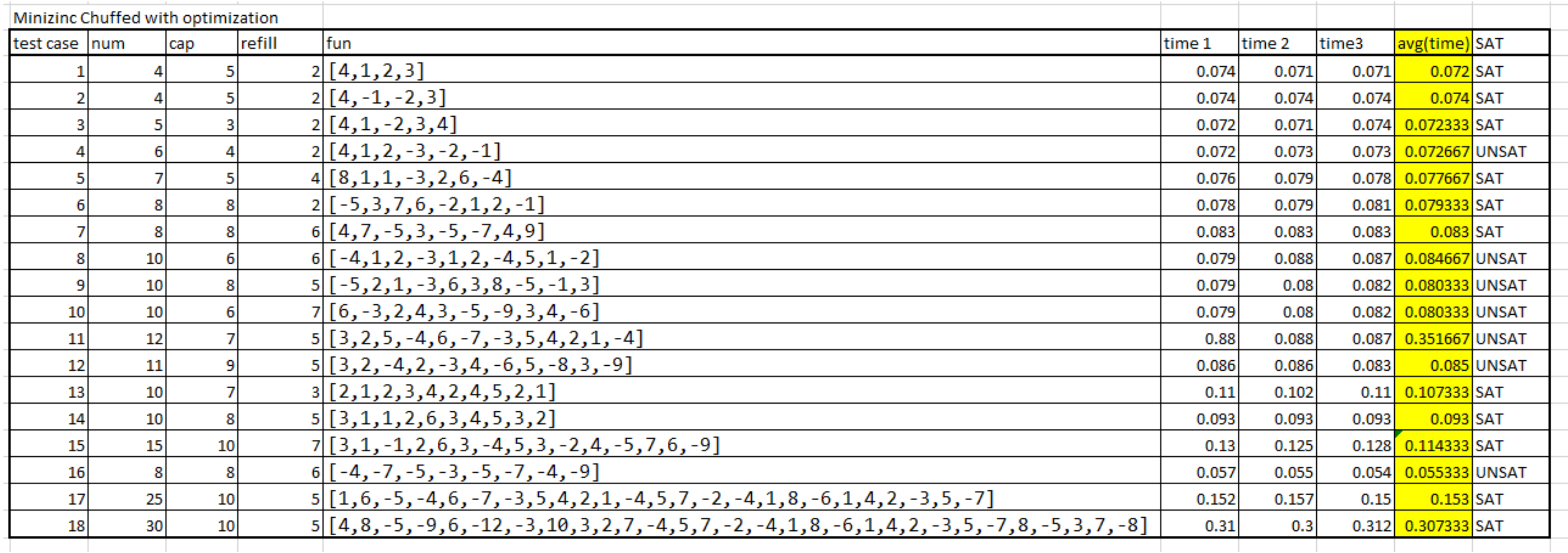
Experiment:

The default configuration for MiniZinc is Gecode, which stands for Generic Constraint Development Environment. For this question, we are switching to another configuration: Chuffed, and seeing if there are significant differences in runtime. The change this time is simple, we installed Chuffed and switched to using it as our solver in MiniZinc.



Minizinc Gecode with 1st and 2nd optimization

test case	num	cap	refill	fun	K	totalFun	time 1	time 2	time3	avg(time)	SAT
1	4	5	2	[4,1,2,3]	25	35	0.09	0.089	0.09	0.089667	SAT
2	4	5	2	[4,-1,-2,3]	25	29	0.088	0.087	0.086	0.087	SAT
3	5	3	2	[4,1,-2,3,4]	20	30	0.09	0.089	0.09	0.089667	SAT
4	6	4	2	[4,1,2,-3,-2,-1]	25	22	0.088	0.091	0.088	0.089	UNSAT
5	7	5	4	[8,1,1,-3,2,6,-4]	40	79	0.092	0.095	0.093	0.093333	SAT
6	8	8	2	[-5,3,7,6,-2,1,2,-1]	55	75	0.097	0.096	0.098	0.097	SAT
7	8	8	6	[4,7,-5,3,-5,-7,4,9]	110	183	0.103	0.102	0.103	0.102667	SAT
8	10	6	6	[-4,1,2,-3,1,2,-4,5,1,-2]	60	59	0.1	0.102	0.101	0.101	UNSAT
9	10	8	5	[-5,2,1,-3,6,3,8,-5,-1,3]	200	149	0.107	0.108	0.11	0.108333	UNSAT
10	10	6	7	[6,-3,2,4,3,-5,-9,3,4,-6]	150	109	0.108	0.107	0.108	0.107667	UNSAT
11	12	7	5	[3,2,5,-4,6,-7,-3,5,4,2,1,-4]	150	156	0.113	0.114	0.115	0.114	SAT
12	11	9	5	[3,2,-4,2,-3,4,-6,5,-8,3,-9]	150	109	0.127	0.152	0.136	0.138333	UNSAT
13	10	7	3	[2,1,2,3,4,2,4,5,2,1]	100	104	0.141	0.14	0.168	0.149667	SAT
14	10	8	5	[3,1,1,2,6,3,4,5,3,2]	100	180	0.658	0.659	0.658	0.658333	SAT
15	15	10	7	[3,1,-1,2,6,3,-4,5,3,-2,4,-5,7,6,-9]	100	334	94.92	95.21	94.12	94.75	SAT
16	8	8	6	[-4,-7,-5,-3,-5,-7,-4,-9]	110	0	0.055	0.054	0.054	0.054333	UNSAT



Minizinc Chuffed with optimization

test case	num	cap	refill	fun	time 1	time 2	time3	avg(time)	SAT
1	4	5	2	[4,1,2,3]	0.074	0.071	0.071	0.072	SAT
2	4	5	2	[4,-1,-2,3]	0.074	0.074	0.074	0.074	SAT
3	5	3	2	[4,1,-2,3,4]	0.072	0.071	0.074	0.072333	SAT
4	6	4	2	[4,1,2,-3,-2,-1]	0.072	0.073	0.073	0.072667	UNSAT
5	7	5	4	[8,1,1,-3,2,6,-4]	0.076	0.079	0.078	0.077667	SAT
6	8	8	2	[-5,3,7,6,-2,1,2,-1]	0.078	0.079	0.081	0.079333	SAT
7	8	8	6	[4,7,-5,3,-5,-7,4,9]	0.083	0.083	0.083	0.083	SAT
8	10	6	6	[-4,1,2,-3,1,2,-4,5,1,-2]	0.079	0.088	0.087	0.084667	UNSAT
9	10	8	5	[-5,2,1,-3,6,3,8,-5,-1,3]	0.079	0.08	0.082	0.080333	UNSAT
10	10	6	7	[6,-3,2,4,3,-5,-9,3,4,-6]	0.079	0.08	0.082	0.080333	UNSAT
11	12	7	5	[3,2,5,-4,6,-7,-3,5,4,2,1,-4]	0.88	0.088	0.087	0.351667	UNSAT
12	11	9	5	[3,2,-4,2,-3,4,-6,5,-8,3,-9]	0.086	0.086	0.083	0.085	UNSAT
13	10	7	3	[2,1,2,3,4,2,4,5,2,1]	0.11	0.102	0.11	0.107333	SAT
14	10	8	5	[3,1,1,2,6,3,4,5,3,2]	0.093	0.093	0.093	0.093	SAT
15	15	10	7	[3,1,-1,2,6,3,-4,5,3,-2,4,-5,7,6,-9]	0.13	0.125	0.128	0.114333	SAT
16	8	8	6	[-4,-7,-5,-3,-5,-7,-4,-9]	0.057	0.055	0.054	0.055333	UNSAT
17	25	10	5	[1,6,-5,-4,6,-7,-3,5,4,2,1,-4,5,7,-2,-4,1,8,-6,1,4,2,-3,5,-7]	0.152	0.157	0.15	0.153	SAT
18	30	10	5	[4,8,-5,-9,6,-12,-3,10,3,2,7,-4,5,7,-2,-4,1,8,-6,1,4,2,-3,5,-7,8,-5,3,7,-8]	0.31	0.3	0.312	0.307333	SAT

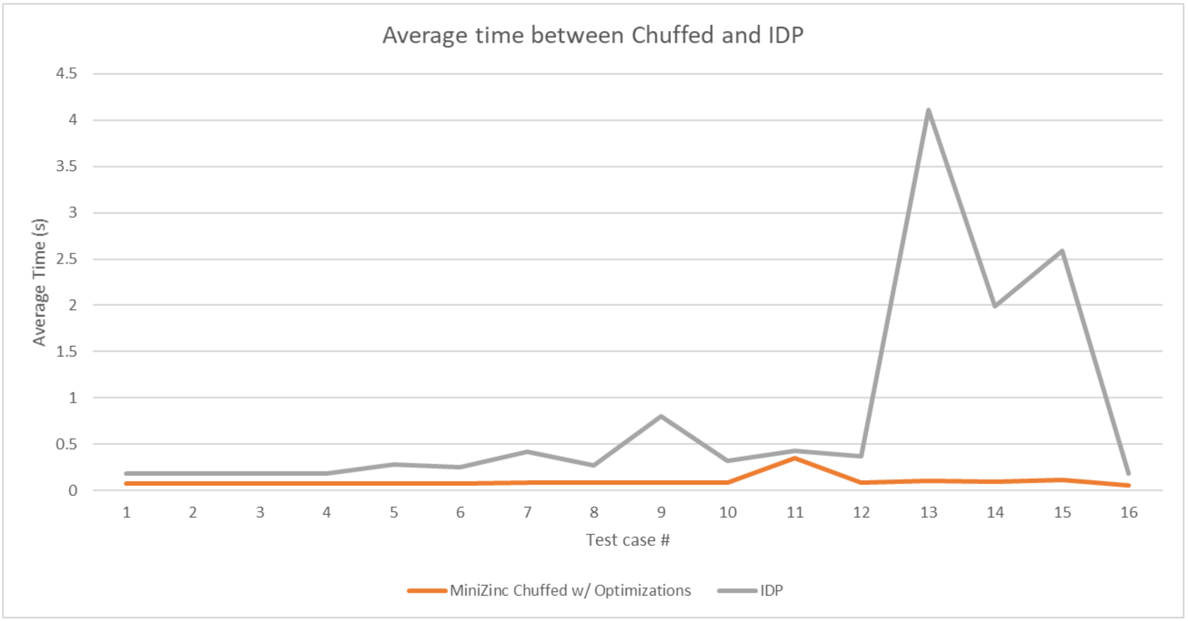
Switching to Chuffed led to obvious decreases in almost every test case. Every test case is completed in less than 1s when using Chuffed. To further test the solving ability of Chuffed, we added 2 extra test cases of num size 25 and 30, sizes that are impossible to solve in Gecode, and both tests were completed in less than a second (0.153s and 0.307s respectively).

What was once an impossible to solve test case for unoptimized Gecode, Test 15 is solved by Chuffed in 0.114s. Through our optimizations, we effectively lowered test 15's runtime from 16166s+ -> 1716s -> 94.5s -> 0.114s. This shows the immense potential in optimization solving programs through adding redundant constraints and picking optimal solving configurations.

Answer:

Switching to Chuffed massively improved runtimes for MiniZinc. As for why Chuffed is so much faster than Gecode, we attempted to research why Chuffed was so much more effective at solving the Sequential games problem than Gecode. Chuffed is a state of the art lazy clause solver designed with lazy clause generation in mind. Lazy clause generation is a hybrid approach to constraint solving that combines features of finite domain propagation and Boolean satisfiability. The goal of Chuffed is to be as simple and effective as possible, and from this experiment, we find it hard to disagree with the results. Further research into Chuffed would be required to accurately explain the reasoning behind its efficiency in comparison to Gecode.

The goal of these optimizations was to attempt to improve MiniZinc's runtimes to resemble IDP's runtimes across the same set of tests. The graph below shows the results.



Conclusion

After applying the 3 optimizations (no negatives more than once constraint, more fun games than less fun games constraint, switching to Chuffed), we have succeeded in improving MiniZinc's runtime speeds beyond unoptimized IDP. Although the constraints were simple, they resulted in significant improvements for MiniZinc, and simply switching to a different configuration allowed MiniZinc to surpass IDP. This observation taught us that small and simple changes can have exponential results for declarative problem solving. In future experiments, we will apply the same constraints to IDP to see if the improvements in runtime are consistent regardless of solver, which we hypothesize that IDP will likely see improvements in its already fast runtimes. In addition, we would like to experiment with other solving configurations too to find a possibly even better one for sequential games than Chuffed.

Andy Ng
301221017

Appendix

MiniZinc folder: contains all files for MiniZinc, including .mzn and .dzn test cases

- Games.mzn
    - Main file for MiniZinc, including all constraints after optimizations
- Test_caseX.dzn
    - Test case #X

IDP folder: contains all files for IDP, including .idp main file and test cases

- Games.idp
    - Main file for IDP, including all basic constraints for sequential games
- Test_caseX.idp
    - Test case #X

Data.xlsx: raw data acquired when calculating the runtimes and making graphs

CMPT417 Final Project Report.pdf: This final project report