Project Report

CMPT 417

LP/CP Programming Contest 2015

Sequential Games

Presented by:

Zelin Han --- 301309342

Dec. 11, 2019

Introduction

The is the report for solving the Sequential Games by MiniZinc and C++ for the project of CMPT 417 in Simon Fraser University. The Sequential Games problem is given by LP/CP Programming Contest 2015 and has been modified by professor Mitchell. This project gives player a sequence of games which associate with fun values for each game. Player can play these games by paying tokens and gain fun. The goal for us is to find whether the total fun can greater than a given K value or not.

I modified this problem to find the optimal solution for player to have a maximum total fun. Since the given problem is a decision problem which means finding satisfied or not, it will only gives the first solution that satisfies the K value or gives the result that K is not satisfiable which is easy. However finding the optimal solution not only can solve the given problem by comparing the maximum total fun with K value, but also can give us more information and can be more challengeable.

Solver System

I do this project by two solver which are MiniZinc and C++ to find optimal solution for player.

I choose MiniZinc since it is a good IDE and easy to use. And it has comprehensive syntax, such as ‘solve satisfy’, ‘solve maximize’ and some logic symbols which just look like we learned in class, which makes it easy to use and read. In addition, all parameters of MiniZinc is used as default setting.

For C++, I choose it since I want to provide a better solver for this problem to have a faster running time and C++ is the fastest language I learned. I use the regular format of C++ to write this solver, and I include a list of standard libraries which I will show in the following report.

Problem Specification

For this project, the Sequential Games is letting player plays a sequence of games G1, G2, …, Gn under 5 conditions.

1. Player may play each game multiple times, but all plays of the same game Gi must be made after the previous one Gi-1 and before the next one Gi+1.
2. Player pays one token each time for every play of a game, and player can play a game many times as long as he or she has a token to pay.
3. Player must play each game at least once.
4. Player has a given number C of tokens when he or she start playing the first game G1, and he or she will receive a "refill" of a given number R of tokens after finishing the current game Gi and before starting the next game Gi+1. But player has a "pocket capacity" C which is the same as the initial number of tokens given to the player, and any extra tokens exceeding the capacity will be wasted.
5. Each game has a "fun" value which may be negative for player.

The goal of this problem is to decide how many times to play each game and maximize player's total fun. And the problem can be described as follows.

Given: – Number $n \in N$ of games;
– Fun value $v_i \in Z$ for each game $i \in [n]$;
– Pocket Capacity $C \in N$;
– Refill amount $R \in N$;
– Goal $K \in N$.

Find: A number of plays p_i for each game $i \in N$ such that maximize total fun.

Our instance vocabulary is (n, C, R, K, v), where v is a unary function and the other symbols are all constant symbols. The solution is a unary function p.

Methodologies for MiniZinc

The first version I did for this project is following the three constraints on the number of tokens (t) that player has and the number (p) of times that player plays for each game, which given by the professor.

First time, I define the number(num) of games, the set of numbers of games(games) which form 1 to num, an array of fun(fun) for each game, the pocket capacity(cap), the refill amount(refill), a set of numbers of x(x_range) which from 0 to cap and the goal value(K). And then I create two array of various integer(t, plays) which represent the number of tokens that player has for each game and the number of times that player plays for each game. Here is the code:

```
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;
int: K;                           % Goal K ∈ N;

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

After these, I'm going to define the three constraints for "t" and "plays".

The first constraint is ∑i∈[n] PiVi ≥ K but I am finding the optimal solution for maximum total fun. So I comment it when I use this code in MiniZinc. Here is the code:

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

The second constraint is the number of tokens ti available to play game i is C(cap) when we start playing the first game, and for i > 1 is the minimum of C(cap) and ti−1 − pi−1 + R. Here is the code:

```
% 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:
constraint (t[1] = cap) /\
forall(i in games)(
  (1<i /\ i<=num) -> exists(x in x_range)(
    (x = (t[i-1] - plays[i-1] + refill)) /\
    ((x > cap) -> (t[i] = cap)) /\
    ((x <= cap) -> (t[i] = x))
  )
);
```

The third constraint is player need to play each game at least once and at most ti times. Here is the code:

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

Finally, I use ‘solve maximize’ to get the optimal solution and use ‘output’ to print it. Here is the code:

```
solve maximize sum(i in games)(plays[i] * fun[i]);
output [show (sum(i in games)(plays[i] * fun[i]))];
```

Up to now, it is a complete algorithm to solve the problem. I create 14 test cases to test my code. It passes the first few test cases including the three cases that provided on the webpage. But when I test the eighth test cases, the result of optimal solution doesn’t look right. It returns 4 as the optimal solution for the input which are num=10, cap=6, refill=6 and fun=[-4,1,2,-3,1,2,-4,5,1,-2]. I show the “plays” array for debugging and I find that the optimal solution plays each game for 6 times except plays the last game 1 time. This is definitely a false result, since it’s obviously that player should only play once when the fun value is negative. Thus, I went back to looking for mistakes and loopholes in my code, and finally I found that the x_range is defined mistakenly. The x_range should be from 0 to (cap+refill), since “x = (t[i-1] - plays[i-1] + refill)” which x can equal to (cap-1+refill) and it much larger than the x_range I define in this test case. Hence, I redefined x_range as (cap+refill). Here is the code:

```
set of int: x_range = 0..(cap + refill);
```

After this, the result of optimal solution is 59 which is right for the test case. But another problem is that the program runs for 15s 893msec which is so slow, and I optimize it to run faster.

The second version of my code is optimized from the first one to run faster. Like I mentioned previously, it's obviously that the optimal solution for player is to play the game with negative fun value only once. Therefore, I add the fourth constraint to let the plays[i] for game i automatically equal to 1. Here is the code:

```
% 4. Additional constraint:
constraint forall (i in games) (fun[i] < 1 -> plays[i] = 1);
```

This improvement greatly reduces the running time which is reflected in the fact that running the same test case (test_case8) it only takes 161msec. This change improves nearly 100 times of running time.

Testing for MiniZinc

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

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

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

The rationale of all test cases I choose is to look for factors that affect the running time of the two versions of program by changing different parameters. According to the table above, we can see that from test 1 to test 6 increase num or cap with small refill will not have a big impact on running time. But with large num and cap, increasing refill will also increase the running time of the first version but not affect the running time of second version, according to the test 7. According to the comparison of test 7 and test 8, increasing num but decreasing cap with same refill will increase the running time for the first version. And also, from test 11, when I increase the num to 12, the first version of program takes more than 30 min without any result. Thus, we may can say that num has a greater effect on the running time than cap for the first version. From test 13 and test 14, we can see that the running time of second version has a great increase when I use only positive fun value with large num and cap.

Question

Until now, the work I done by MiniZinc is for part 1 of this project (with some updates). I'm not going to insert some questions for that like the second version program of MiniZinc. For now, I prefer to give some more constructive questions and solutions.

I try to test the MiniZinc solver and find that it will work a very long time for a bit bigger 'num'. I think the limitation of running time is caused by the built-in algorithm of MiniZinc solver. So, is there another way to solve this problem faster? And how the different parameters affect the running time?

Solution

Using the knowledge I have learned in university, the most powerful and applicable solution is to write a solver in C++. Since C++ is the second fastest language after assembly and writing a solver by assembly is obviously unprocurable. Thus, I am planning to write a solver by C++ and give a plenty of test cases to show how the different parameters affect the running time.

Methodologies for C++

For C++, I am going to introduce it in two parts. The first part is solver which written in ‘solver.cpp’; the second part is data generator which written in ‘generator.cpp’.

– Solver

Notice: a part of the program is inspired by my classmate Duo Lu. We discuss the program together and I write a citation in the part of code.

The first thing of solver is to read file. In this way, it can read and solve a lot of test cases at a time. Here is the standard libraries I include and read file function:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <algorithm>
#include <fstream>
#include <string>
#include <iostream>
#include <vector>
#include <sstream>
#include <bits/stdc++.h>
#include <assert.h>
#include <ctime>
#include <time.h>
#include <chrono>

using namespace std;
using namespace std::chrono;
```

```
int read_file(vector<string> &v, string name){
    string line;
    ifstream myfile (name);
    if(myfile.is_open()){
        while(getline(myfile, line)){
            // cout << line << endl;
            v.push_back(line);
        }
        myfile.close();
    }
    else cout << "Unable to open file." << endl;
    return 0;
}
```

I write the ‘read_file()’ function to get every lines in the test case file and save them in the vector ‘v’ which is defined in main() function.

The second helping function is ‘get_data()’ which extracts all the data and stores them in the proper parameters. I define a series of global variables to store data which are:

```
int test, num, cap, refill, K, num_neg, num_pos, sum_neg, sum_pos;
vector<int> fun;
```

.

Here is the get_data() function:

```
int get_data(vector<string> &v){
    int signal;
    stringstream ss;
    for(int i = 0; i < v.size(); i++){
        signal = 0;
        ss.clear();
        ss << v[i];
        string str;
        int data;
        while(!ss.eof()){
            ss >> str;
            if(str == "num") {signal = 1;}
            if(str == "cap") {signal = 2;}
            if(str == "refill") {signal = 3;}
            if(str == "K") {signal = 4;}
            if(str == "fun") {signal = 5;}
            if(str == "test") {signal = 6;}
            if(str == "num_neg") {signal = 7;}
            if(str == "num_pos") {signal = 8;}
            if(str == "sum_neg") {signal = 9;}
            if(str == "sum_pos") {signal = 10;}

            if(str[0] == '['){
                str.erase(0, 1);
            }

            if(stringstream(str) >> data){
                switch(signal){
                    case 1:num = data;
                    break;
                    case 2:cap = data;
                    break;
                    case 3:refill = data;
                    break;
                    case 4:K = data;
                    break;
                    case 5:fun.push_back(data);
                    break;
                    case 6:test = data;
                    break;
                    case 7:num_neg = data;
                    break;
                    case 8:num_pos = data;
                    break;
                    case 9:sum_neg = data;
                    break;
                    case 10:sum_pos = data;
                    break;

                }
            }
        }
    }
}
```

I use ‘stringstream’ to convert values between string and integer. And I use for loop to get every line of data. The bunch of ‘if’ statements is to determine what data of this line belongs to. And then, when getting the data behind the data names, like ‘num’ and ‘cap’, I use ‘switch’ to separate different conditions and allocate data to the corresponding parameter.

For the main() function, I can introduce it by three parts.

First part is to prepare data for the solver. And here is the code:

```
int main(){
    vector<string> v;
    int signal = 1;
    string name, tmp_str;
    int number;
    stringstream ss;

    ofstream myfile;
    myfile.open("result_data_refill.csv");
    myfile << "test_number,num,cap,refill,K,num_neg,num_pos,sum_neg,sum_pos,total_fun,running_time" << endl;

for(int n = 0; n < 40; n++){
    for(int m = 0; m < 10; m++){

    ss.clear();
    v.clear();
    fun.clear();

    number = n*10 + m;
    ss << number;
    ss >> tmp_str;
    name = "data/test_" + tmp_str;

    read_file(v, name);

    // check what we get:
    // for(int i = 0; i < v.size(); i++){
    //  cout << i << ":" << v[i] << endl;
    // }
    // cout << "----------" << endl;

    get_data(v);

    // check what we get:
    // cout << "test: " << test << endl;
    // cout << "num: " << num << endl;
    // cout << "cap: " << cap << endl;
    // cout << "refill: " << refill << endl;
    // cout << "K: " << K << endl;
    // cout << "fun: ";
    // for(int i = 0; i < fun.size(); i++){
    //  cout << fun[i] << ", ";
    // }
    // cout << endl;
    // cout << "----------" << endl;

    assert(num == fun.size());
```

I initialize some variables, such that ‘v’, ‘name’, ‘tmp_str’, ‘number’ and etc. And then I create a .csv file, like ‘result_data_refill.csv’, which store the result data. I initially put the header of the data to the .csv file. After that, there are two ‘for’ loops which totaly run 400 times that the same number of test cases I create by ‘generator.cpp’. For every loop, I firstly clear some variables which will use multiple times. Secondly, I create the file name for different test cases. Moreover, I call the ‘read_file()’ and ‘get_data()’ functions to get the data I need. Finally, I give an ‘assert’ function to test that is the ‘num’ equal to the size of ‘fun’.

Second part of the main() function is to solve the optimal solution for player.

```
// ready to solve:
vector<int> play(num,0);
int tmp_token = cap;
int next_token = 0;
int total_fun = 0;
int save = 0;

auto start = high_resolution_clock::now();

// The following 'for loop' is inspired by Duo Lu:
for(int i = 0; i < num; i++){
    tmp_token--;
    play[i]++;

    if(fun[i] < 0){
        total_fun = total_fun + fun[i];
        tmp_token = min(tmp_token+refill, cap);
    }
    else{
        int next;
        next = i+1;
        next_token = refill;

        while(next_token < cap){
            if(next < num){
                if(fun[next] > fun[i]){
                    save = cap - next_token;
                    if(save > tmp_token){
                        save = tmp_token;
                    }
                    break;
                }
            }
            else{
                break;
            }
            next++;
            next_token = next_token + refill - 1;
        }
        tmp_token = tmp_token - save;
        play[i] = play[i] + tmp_token;
        total_fun = total_fun + (play[i]*fun[i]);
        tmp_token = min(save + refill, cap);
        save = 0;
    }
}

auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
```

For this part, I initialize some variables for the solver and give a time point before the solver algorithm. Afterwards, there is the solver algorithm which I discuss with my classmate Duo Lu. The first 'if' function is to play game [i] only once when fun[i] is negative and 'else' function is to consider the situation when fun[i] is greater than zero. The main idea of the algorithm in 'else' is to save tokens as much as possible, but without waste, when there is a greater fun in the future game. Some of the constraint is come from the MiniZinc part, like `tmp_token = min(save + refill, cap);`. I will prove the correctness of this algorithm in the next section. After the algorithm, there is another time point and use the first time point to calculate the duration of running time.

The third part is to save result data and end the two 'for' loops. Finally, close the .csv file and return 0 to terminate the program. Here is the code:

```
    myfile << test << ",";
    myfile << num << ",";
    myfile << cap << ",";
    myfile << refill << ",";
    myfile << K << ",";
    myfile << num_neg << ",";
    myfile << num_pos << ",";
    myfile << sum_neg << ",";
    myfile << sum_pos << ",";
    myfile << total_fun << ",";
    myfile << duration.count();
    myfile << endl;

    }
}

    myfile.close();

return 0;
}
```

I write some citations which I use in this program. Here are the citations:

```
// Citation:
// https://www.geeksforgeeks.org/extract-integers-string-c
// https://www.geeksforgeeks.org/measure-execution-time-function-cpp/
// https://stackoverflow.com/questions/25201131/writing-csv-files-from-c
```

– Generator

After finish the solver, I also write a generator.cpp to generate a large number of test cases at a time. I use the same header from solver.cpp and I only write a main() function in this code.

```
int main(){

    int num, cap, refill, K;
    string name, tmp_str;
    int number;

for(int n = 0; n < 40; n++){
    for(int m = 0; m < 10; m++){

    num = 60000;
    cap = 600;
    refill = 20 + 20*n;
    K = 100000;

    stringstream ss;
    number = n*10 + m;
    ss << number;
    ss >> tmp_str;
    name = "test_" + tmp_str;

    vector<int> fun;

    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<> dis(-9999, 9999);
    // normal_distribution<> dis(-9999, 9999);

    int num_neg=0;
    int sum_neg=0;
    int num_pos=0;
    int sum_pos=0;
    int zero=0;
    int tmp;
    for(int i = 0; i < num; i++){
        tmp = int(dis(gen));
        if(tmp < 0){
            num_neg++;
            sum_neg = sum_neg + num_neg;
        }
        else if(tmp > 0){
            num_pos++;
            sum_pos = sum_pos + num_pos;
        }
        else{
            zero++;
        }
        fun.push_back(tmp);
    }
```

The first part of main() function is to initialize variables and set two ‘for’ loops which run 400 times for different ‘refill’(200 times for ‘num’ and ‘cap’) to generate one test case for each loop. In the loop, I set the ‘num’, ‘cap’, ‘refill’ and ‘K’ to generate different data I need. Totaly, I generate three set of test cases which are changing one variable and keep others constant. The code is an example for different ‘refill’. After that I create file name and give the random model to generate random fun values. I use ‘for’ loop to generate ‘num’ number of fun values and calculate the number of negative value, the number of positive value, sum of negative value and sum of positive value.

The second part is to create a file with the name I got and save the data I generate to this file. Here is the code:

```
    ofstream myfile("data/"+name);
    if(myfile.is_open()){
        myfile << "% test "+tmp_str << endl;
        myfile << "\n";
        myfile << "num = " << num << endl;
        myfile << "cap = " << cap << endl;
        myfile << "refill = " << refill << endl;
        myfile << "K = " << K << endl;
        myfile << "num_neg = " << num_neg << endl;
        myfile << "num_pos = " << num_pos << endl;
        myfile << "sum_neg = " << sum_neg << endl;
        myfile << "sum_pos = " << sum_pos << endl;
        myfile << "fun = [";
        for(int i = 0; i < num-1; i++){
            myfile << fun[i] << ", ";
        }
        myfile << fun[num-1] << "];";

        myfile.close();
    }


    }
}


    return 0;
}


// Citation:
// https://www.cnblogs.com/egmkang/archive/2012/09/06/2673253.html
```

I save ‘num’, ‘cap’, ‘refill’, ‘K’, ‘num_neg’, ‘num_pos’, ‘sum_neg’, ‘sum_pos’ and ‘fun’ for future analysis which done by ‘analysis.py’.

Analysis for C++ Algorithm

In this section, I use Python to analyze result data I got which are 'result_data_num.csv', 'result_data_cap.csv' and 'result_data_refill.csv'. I write the code in 'analysis.ipynb' and I will introduce the result and give the conclusion I got from the three result data. Besides, here is the import header:

```
import pandas as pd
import numpy as np
import sys
import math
import matplotlib.pyplot as plt
from scipy import stats
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
import seaborn as seabornInstance
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import operator
```

1. running_time vs. num

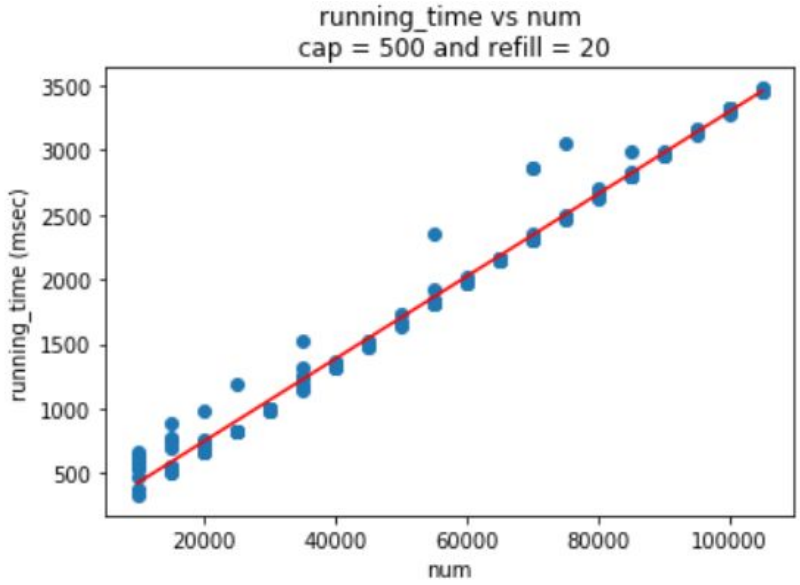
For 'num', I use n=20, m=10 for 'for' loop to generate data, which means there are 20 different 'num' values and for each value there are 10 test cases. Here is the definition of parameters:

```
num = 10000 + 5000*n;
cap = 500;
refill = 20;
K = 100000;
```

After using Python to read .csv file, I use following code to analyze data:

```
x = data['num']
y = data['running_time']

plt.scatter(x,y)

lin_reg = stats.linregress(x, y)
plt.plot(x, x*lin_reg.slope+lin_reg.intercept, 'r-')

plt.title('running_time vs num\n cap = 500 and refill = 20')
plt.xlabel('num')
plt.ylabel('running_time (msec)')
plt.show()

print("slope = ", lin_reg.slope)
print("intercept = ", lin_reg.intercept)
print('r value between log(cap) and running_time: ',
      stats.linregress(x, y).rvalue.round(10))
print('r-sqr is: ', ((stats.linregress(x, y).rvalue)**2).round(10))
```

This code gives us this output:




```
slope =  0.031906812030075185
intercept =  105.37330827067694
r value between log(cap) and running_time:  0.9938803355
r-sqr is:  0.9877981212
```

From the output, we can see that r-sqr is equal to 0.9877981212 which means there are 98.78% of ‘running_time’ value can be explained by ‘num’ value; and r value is equal to 0.9939 which means there is a strong linear correlation between running time and ‘num’. The function of regression line is ‘Y = 105.3733 + 0.0319 * X’ which Y is running time and X is ‘num’.

2. running_time vs. cap

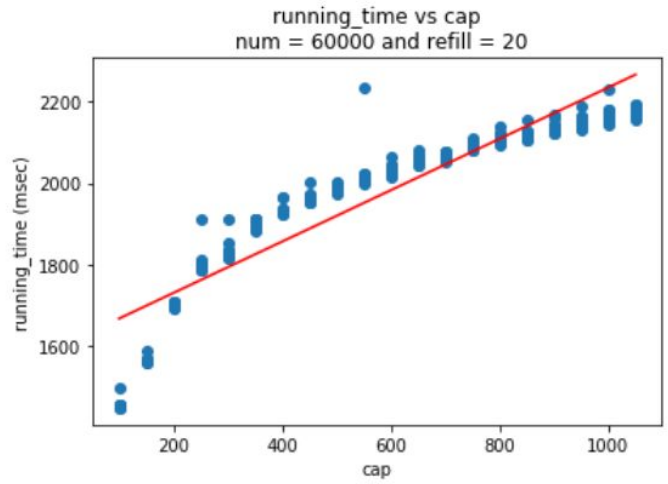
For ‘cap’, I use n=20, m=10 for ‘for’ loop to generate data, which means there are 20 different ‘cap’ values and for each value there are 10 test cases. Here is the definition of parameters:

```
num = 60000;
cap = 100 + 50*n;
refill = 20;
K = 100000;
```

After using Python to read .csv file, I use following code to analyze data:

```
x = data['cap']
y = data['running_time']

plt.scatter(x,y)

lin_reg = stats.linregress(x,y)
plt.plot(x, x*lin_reg.slope+lin_reg.intercept, 'r-')

plt.title('running_time vs cap\n num = 60000 and refill = 20')
plt.xlabel('cap')
plt.ylabel('running_time (msec)')
plt.show()

print("slope = ", lin_reg.slope)
print("intercept = ", lin_reg.intercept)
print('r value between log(cap) and running_time: ',
      stats.linregress(x, y).rvalue.round(10))
print('r-sqr is: ', ((stats.linregress(x, y).rvalue)**2).round(10))
```

This code gives us this output:

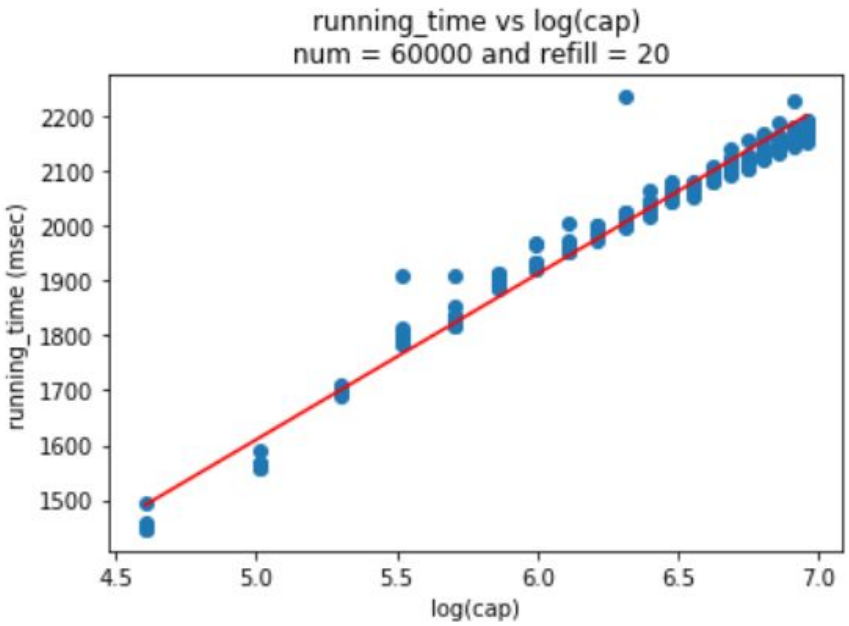



```
slope =  0.6315578947368421
intercept =  1603.5542105263157
r value between log(cap) and running_time:  0.9154462557
r-sqr is:  0.8380418471
```

From the scatter plot, we can see that it looks like a log function. Thus I log the 'cap' value and do the linear regression again. Here is the code:

```
x = np.log(data['cap'])
# data['log_cap'] = x
y = data['running_time']

plt.scatter(x,y)

lin_reg = stats.linregress(x,y)
plt.plot(x, x*lin_reg.slope+lin_reg.intercept, 'r-')

plt.title('running_time vs log(cap)\n num = 60000 and refill = 20')
plt.xlabel('log(cap)')
plt.ylabel('running_time (msec)')
plt.show()

print("slope = ", lin_reg.slope)
print("intercept = ", lin_reg.intercept)
print('r value between log(cap) and running_time: ',
      stats.linregress(x, y).rvalue.round(10))
print('r-sqr is: ', ((stats.linregress(x, y).rvalue)**2).round(10))
```

This code gives us this output:




```
slope =  301.9356755934919
intercept =  100.42654607686381
r value between log(cap) and running_time:  0.9876272769
r-sqr is:  0.9754076381
```

From the output, we can see that r-sqr is equal to 0.9754076381 which means there are 97.54% of ‘running_time’ value can be explained by log(cap) value; and r value is equal to 0.9876 which means there is a strong linear correlation between running time and log(cap). Thus we can say that there is a log relation between running time and cap; and the function of regression line is ‘Y = 100.4265 + 301.9357 * X’ which Y is running time and X is log(cap).

3. running_time vs. refill

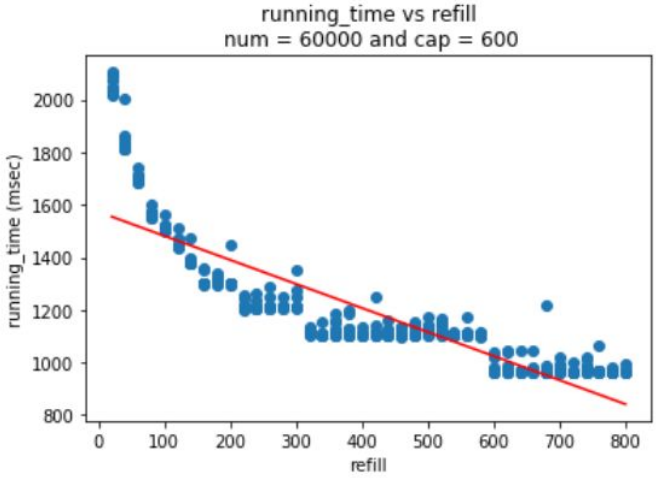
For ‘refill’, I use n=40, m=10 for ‘for’ loop to generate data, which means there are 40 different ‘cap’ values and for each value there are 10 test cases. Here is the definition of parameters:

```
num = 60000;
cap = 600;
refill = 20 + 20*n;
K = 100000;
```

After using Python to read .csv file, I use following code to analyze data:

```
x = data['refill']
y = data['running_time']

plt.scatter(x,y)
lin_reg = stats.linregress(x, y)
plt.plot(x, x*lin_reg.slope+lin_reg.intercept, 'r-')

plt.title('running_time vs refill\n num = 60000 and cap = 600')
plt.xlabel('refill')
plt.ylabel('running_time (msec)')
plt.show()

print("slope = ", lin_reg.slope)
print("intercept = ", lin_reg.intercept)
print('r value between refill and running_time: ',
      stats.linregress(x, y).rvalue.round(10))
print('r-sqr is: ', ((stats.linregress(x, y).rvalue)**2).round(10))
```

This code gives us this output:

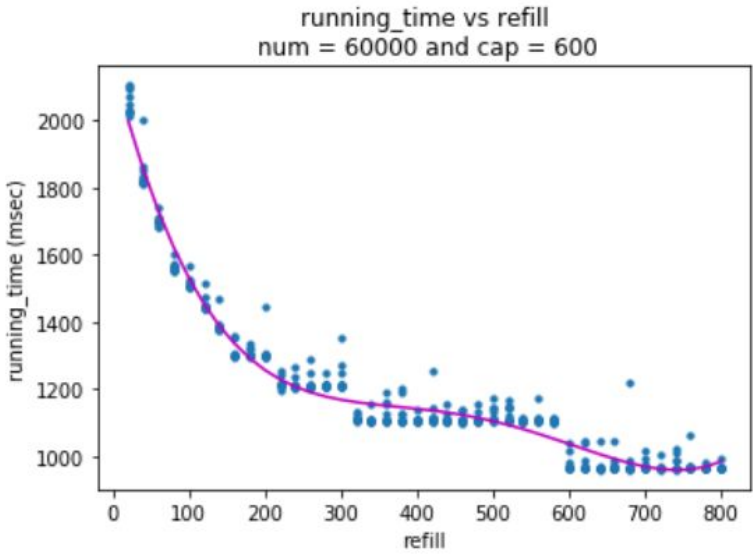



```
slope =  -0.9135661350844281
intercept =  1572.8596153846154
r value between refill and running_time:  -0.8527151076
r-sqr is:  0.7271230547
```

From the scatter plot, we can see that there is a negative relation between running time and refill. Thus decide to use polynomial regression to fit these data. And here is the code:

```
# https://towardsdatascience.com/polynomial-regression-bbe8b9d97491

tmp_data = data[['refill', 'running_time']]
x = x[:, np.newaxis]
y = y[:, np.newaxis]

polynomial_features = PolynomialFeatures(degree=4)
x_poly = polynomial_features.fit_transform(x)

model = LinearRegression(fit_intercept=False)
model.fit(x_poly, y)
y_poly_pred = model.predict(x_poly)

rmse = np.sqrt(mean_squared_error(y,y_poly_pred))
r2 = r2_score(y,y_poly_pred)
# print(rmse)
# print(r2)

plt.scatter(x, y, s=10)
# sort the values of x before line plot
sort_axis = operator.itemgetter(0)
sorted_zip = sorted(zip(x,y_poly_pred), key=sort_axis)
x, y_poly_pred = zip(*sorted_zip)
plt.plot(x, y_poly_pred, color='m')

plt.title('running_time vs refill\n num = 60000 and cap = 600')
plt.xlabel('refill')
plt.ylabel('running_time (msec)')
plt.show()

print('r-sqr is: ', r2)

name = polynomial_features.get_feature_names(tmp_data.columns)
value = model.coef_.reshape((1,5))

function = pd.DataFrame(value, columns = name)
function
```

This code gives us this output:




```
r-sqr is:  0.9720924197579291
```

	1	refill	refill^2	refill^3	refill^4
0	2162.811871	-8.83802	0.029071	-0.000042	2.140834e-08

I use degree = 4 for the polynomial regression. From the output, we can see that r-sqr is equal to 0.9720924197579291 which means there are 97.21% of ‘running_time’ value can be explained by refill value. And there are some coefficient for regression line. Thus we can get the function of regression line is ‘Y = 2162.8119 - 8.8380 * X’ + 0.0291 * X^2 - 4.2e-05*X^3 + 2.14e-08*X^4’ which Y is running time and X is refill.

Discussion

From the analysis I did, I can conclude that there is a linear relation between running time and num, a linear relation between running time and log(cap), a negative relation between running time and refill. Thus for this solver, it does a polynomial running time. So I work out a better solver that run much faster than MiniZinc. If I want to continue this project in the future, I will write assembly version solver for this project. It maybe still run in polynomial time, but I think it will run faster than C++ version.

Proof for C++ Algorithm

In this part, I'm going to prove the correctness of my C++ algorithm. My algorithm is a kind of greedy algorithm which follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. Now let's consider different situations to see why it's a correct algorithm.

Firstly, let's think about the situation when the temporary game has a negative fun value. In the 'for' loop, the first 'if' statement will set the 'play[i]' equals to 1 by calculating the total fun and the temporary token player has after refill, and then jumping into the next game.

Secondly, when the temporary game has a positive fun value, the program will execute the else statement. It will initialize a serial number of next game as 'next', and a number of token that player has in the next game as 'next_token'. After that, it gives a 'while' loop to search for the next few game to find the maximum fun game and save token as much as possible. In detail, it firstly gives a constraint that the 'next' number must smaller than 'num' which is obvious right, since you cannot search a game which doesn't exist. And then, it compares the 'fun[next]' with 'fun[i]' to find the maximum fun

game. If the ‘next’ game has more fun, it will calculate how many token you should save for the ‘next’ game to have a maximum total fun without exceeding the capacity of pocket. Moreover, it will increase the ‘next’ by 1 to search for the next game and let the ‘next_token’ for the ‘next’ game plus refill and minus 1 for a necessary play for each game. The ‘while’ loop will continually run until ‘next_token’ is no more smaller than ‘cap’. This is because the token for the ‘next’ game will exceed the pocket capacity which is meaningless for searching more games. After ‘while’ loop, the ‘tmp_token’ equals to ‘tmp_token’ minus ‘save’ which means after saving the token for larger fun value game, player has ‘tmp_token’ to play the temporary game. Thus it calculates ‘play[i]’, ‘total_fun’ and ‘tmp_token’ for the next game which equals to the minimum of (save + refill) and ‘cap’. At last, reset ‘save’ and jump to the next game.

This algorithm give the optimal choice for each steps. So at each stage, it gives the partial optimal solution. Thus, at the end of the algorithm, it will give the optimal solution for the hole game.

Appendix

CMPT417 Final Report.pdf:

Final report for CMPT417 project

CMPT417 Part1 Report.pdf:

Part1 report for CMPT417 project

A set of test cases for num in "data_num":

200 test cases for finding the relation between running time and ‘num’

A set of test cases for cap in "data_cap":

200 test cases for finding the relation between running time and ‘cap’

A set of test cases for refill in "data_refill":

400 test cases for finding the relation between running time and ‘refill’

A set of screenshot in "Screenshot":

All screenshots I use in this report

"MiniZinc" folder containing "Sequential Games.mzn" and 14 test cases:

Sequential Games.mzn is the solver for part 1 and 14 test cases

analysis.ipynb:

Analysis for result data created by C++ solver, can opened by jupyter notebook

generator.cpp:

Test case generator written in C++

solver.cpp:

Sequential Games solver written in C++

make:

Make file, which can execute .cpp files by typing ‘make file_name’ in terminal

redult_data_num.csv:

Result data on changing ‘num’

redult_data_cap.csv:

Result data on changing ‘cap’

redult_data_refill.csv:

Result data on changing ‘refill’