CMPT 417 – FINAL PROJECT

Sheil Mehta

301307105

Simon Fraser University

Contents

Part I.

Introduction.

The problem that this project is trying to solve, is the Pizza problem. The problem arose in the University College Cork student dorms when there was a large order of pizzas for a party, and many of the students had vouchers/coupons for acquiring discounts in purchasing pizzas. I worked out the minimizing or optimizing solution for this problem. That is, I worked out the lowest amount the students would have to pay, using the coupons they had, in order to buy the number of pizzas they wanted. Since I was working on the optimization solution, I assumed that there would not be any unsatisfiable test cases.
I was able to use the 8 constraints as discussed in the lecture, along with an additional constraint, to get a working solver for the problem. The solvers I used were Minizinc, and the IDP solver.
I will be explaining the specifications as well as the solver parameters below.
I have used the words coupon and voucher interchangeably, as they mean the same thing in my report.

Part I covers the differences between the implementations within different solvers, their specifications, and my test cases.

Specification and Solver parameters.

For solving the pizza problem as per the specifications provided, the main aim is to buy all the ordered pizzas at minimal cost.
A coupon is a pair of numbers (c, f) where you pay for 'c' pizzas and obtain 'f' pizzas for free as long as the free pizzas cost no more than the cheapest 'c' pizza that was paid for.
And lastly, a coupon does not need to be completely used and not all of the coupon needs to be used.

We are trying to find the best assignment of coupons, in order to minimize the cost.

I started by determining a vocabulary to build constraints with.
A vocabulary L then can be defined by the symbols [n, m, price, buy, free, k] where n is the number of pizzas, m is the number of coupons, price is a unary function [n] -> $\mathbb{N}$ that maps the price of n pizzas, buy is a unary function [m] -> $\mathbb{N}$ that maps how many pizzas must be bought for the coupon m to be valid, while free is a unary function [m] -> $\mathbb{N}$ that maps how many pizzas can be ordered for free by coupon m. k is the cost that must be minimized.
Since we are trying to solve the minimized solution that is also valid, I also have additional symbols, [Paid, Used, Apply, Justify]. Paid [n] -> {Bool} is a unary symbol that represents the set of paid pizzas, in the same way, Used [m] ->{Bool} represents the set of used coupons. Apply and justify are binary symbols [m] -> [n] -> {Bool}. They hold when a pizza was paid for and validates a coupon, and when a pizza is free because of a valid coupon.

The problem constraints that I constructed were:

1. pay for exactly the pizzas that we don't get by using the coupons
2. Used is the set of applied coupons
3. Any coupon used must be justified by enough paid pizzas
4. No coupon is used for too many pizzas
5. Each free pizza costs at most as much as the cheapest pizza used to justify the coupon used
6. We pay for all pizzas that justify using coupons
7. The total cost is not too large (less than or equal to k)
8. Justify and apply hold only coupon-pizza pairs
9. Any pizza used to justify a coupon can only be used once

A conjunction of all the above constraint formulas gives us the specification for this problem.

The test cases were run using the default settings in both IDP and Minizinc when running the solvers. I ran the IDP program on the online IDE since I couldn't get it to run on my windows system.
I created test cases with smaller numbers for n and m, so that it would run on the online IDE without crossing the resource limit.

Test Cases.

I first tested my data with the 3 sample cases given on the website. I ran this for my Minizinc program, since the web IDE had limitations on resources so I couldn't input large numbers of n and m. Once I figured that the Minizinc program worked, I ran it on 10 test cases, and then ran the IDP program on the same 10 test cases. I compared the results for both run time as well as actual output. The output results were identical, however there was a huge difference in runtimes. This may be because the web IDE has been heavily optimized on the virtual machine it is running on. Additionally, it might be running on far better hardware than mine.

Test case 1: Sample from the website
```
n = 4;
price = [10,5,20,15];
m = 2;
buy = [1,2];
free = [1,1];
```
Test case 2: Sample from the website
```
n = 4;
price = [10,15,20,15];
m = 7;
buy = [1,2,2,8,3,1,4];
free = [1,1,2,9,1,0,1];
```
Test case 3: Sample from the website
```
n = 5;
price = [70,10,60,60,30];
m = 4;
buy = [1,2,1,1];
free = [1,1,1,0];
```

Test case 4:

    n = 5;
    price = [6,20,45,45,30];
    m = 5;
    buy = [4,1,4,2,2];
    free = [3,2,4,1,4];

Test case 5: I ran a test case in which all the prices were the same

    n = 5;
    price = [10,10,10,10,10];
    m = 5;
    buy = [3,1,1,2,3];
    free = [1,1,1,1,2];

Test case 6:

    n = 5;
    price = [6,80,22,45,30];
    m = 5;
    buy = [4,1,4,2,1];
    free = [3,4,4,1,4];

Test case 7:

    n = 5;
    price = [7,20,80,47,54];
    m = 4;
    buy = [3,5,3,2];
    free = [1,2,2,1];

Test case 8: I ran a test which had all the coupons were the same

    n = 5;
    price = [100,90,25,10,50];
    m = 5;
    buy = [2,2,2,2,2];
    free = [1,1,1,1,1];

Test case 9: I tried a simple/trivial solution test case

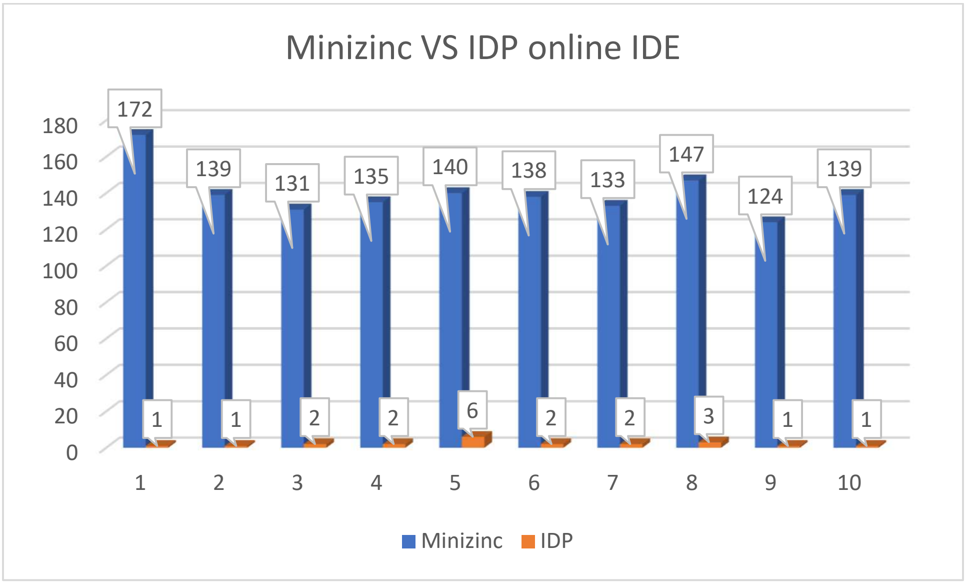
    n = 3;
    price = [5,8,8];
    m = 1;
    buy = [1];
    free = [1];

Test case 10:

    n = 5;
    price = [20,5,4,8,10];
    m = 5;
    buy = [3,4,5,3,4];
    free = [1,1,3,2,2];

Test#	Minizinc	IDP	Output
1	172	1	35
2	139	1	35
3	131	2	140
4	135	2	71
5	140	6	30
6	138	2	80
7	133	2	161
8	147	3	225
9	124	1	13
10	139	1	38

(Run times are in ms)



Part II.

My exploration.

I am exploring the question whether data types, additional constraints, and alterations to constraints have an impact on the runtime of a solver program written in Minizinc, with the default settings and parameters. This is to find out what makes a good specification when choosing to write a solver in Minizinc.

I developed alternate specifications, to create a second heuristic for my Minizinc code. This version incorporated changes to the code, in order to see if they affected the speed of the solver. I had two different Minizinc file; pizza1.mzn and pizza2.mzn. Pizza1.mzn was the first solver I coded, and pizza2.mzn was the one I re-wrote with a different set of constraints and data types.

Specification changes.

My first three test cases were the same ones as the sample inputs. I did this in-order to verify that my solvers were working correctly. Beyond that, I tried changing the different parameters in each test to test something different with every test run. I had 10 test cases. I tried different prices, different values for n and m, and I also tried some special edge cases, such as having only one coupon, having all prices the same, and all coupons being the same.
I increased the sizes for both n and m in these tests since I was running it on my own machine.

Test case 1: Sample from the website
```
    n = 4;
    price = [10,5,20,15];
    m = 2;
    buy = [1,2];
    free = [1,1];
```
Test case 2: Sample from the website
```
    n = 4;
    price = [10,15,20,15];
    m = 7;
    buy = [1,2,2,8,3,1,4];
    free = [1,1,2,9,1,0,1];
```
Test case 3: Sample from the website
```
    n = 10;
    price = [70,10,60,60,30,100,60,40,60,20];
    m = 4;
    buy = [1,2,1,1];
    free = [1,1,1,0];
```
Test case 4:
```
    n = 5;
    price = [6,20,45,45,30];
    m = 10;
    buy = [4,1,4,2,2,3,4,4,4,1];
```

free = [3,2,4,1,4,4,2,3,2,3];

Test case 5: I ran a test case in which all the prices were the same

```
n = 9;
price = [10,10,10,10,10,10,10,10,10];
m = 5;
buy = [3,1,1,2,3];
free = [1,1,1,1,2];
```

Test case 6:

```
n = 5;
price = [6,80,22,45,30];
m = 10;
buy = [4,1,4,2,1,3,4,4,4,2];
free = [3,4,4,1,4,4,2,3,2,3];
```

Test case 7:

```
n = 9;
price = [7,20,80,47,54,68,46,38,80];
m = 4;
buy = [3,5,3,2];
free = [1,2,2,1];
```

Test case 8: I ran a test which had all the coupons were the same

```
n = 5;
price = [100,90,25,10,50];
m = 5;
buy = [2,2,2,2,2];
free = [1,1,1,1,1];
```

Test case 9: I tried a simple/trivial solution test case

```
n = 3;
price = [5,8,8];
m = 1;
buy = [1];
free = [1];
```

Test case 10:

```
n = 7;
price = [20,5,4,8,10,30,60];
m = 5;
buy = [3,4,5,3,4];
free = [1,1,3,2,2];
```

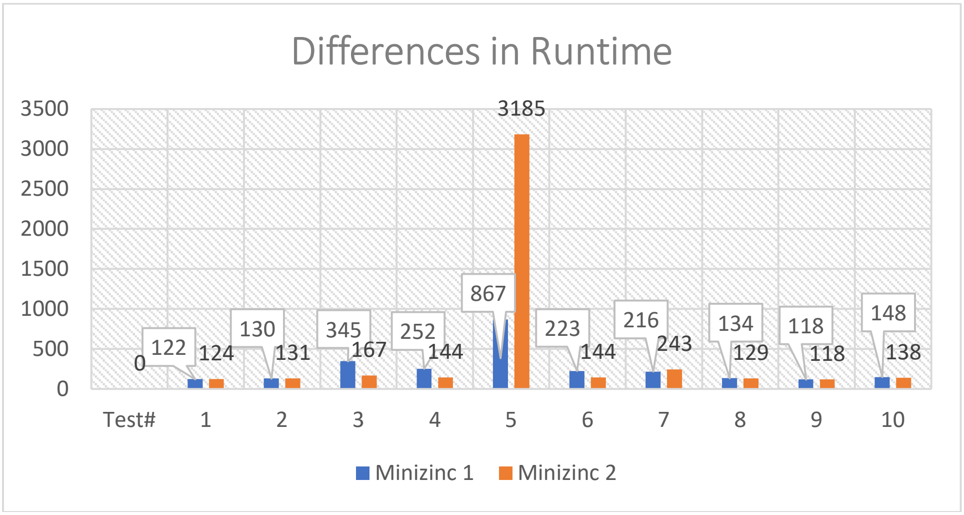
I used the same test cases for both of the programs so that it would be a fair test, and it would be easier to compare the two different versions of Minizinc code.
The reason I chose to re-write my code in Minizinc was because I was more comfortable with this language and it provided more documentation than IDP.
I carried out my tests on the same windows machine with a quad-core intel-i7 10th gen. with 16 gb or RAM and no dedicated gpu.

I used the function bool2int in pizza1.mzn and removed this from second version. Additionally, in the first program, I used sets while I modified this for my second program to directly use arrays

instead. This modification was just to see if different data types would influence the performance of the program. I also added a constraint to my second program, to check whether coupons are not used more than they exist, and pizzas are not more than required. This was a redundant clause and did not change the correctness of the code. This change was made to check whether adding additional clauses also had an impact on performance.
Lastly, I made a modification to one of the constraints from the first program. Constraint 6 now checks whether two coupons are not used by the same pizza.

Data or other observations.

	Minizinc 1		Minizinc 2	
Test#	Time (ms)	Output	Time (ms)	Output
1	122	35	124	35
2	130	35	131	35
3	345	340	167	340
4	252	51	144	51
5	867	50	3185	50
6	223	80	144	80
7	216	314	243	314
8	134	225	129	225
9	118	13	118	13
10	148	119	138	119



Minizinc 1 is the file pizza1.mzn and Minizinc 2 is the file pizza2.mzn.

	pizza1.mzn	pizza2.mzn
Average	255.5	452.3
average(without test 5)	187.55556	148.66667

I calculated the average performance times for each of the programs. I realised that the averages were being heavily skewed by the test case number 5, which did particularly poorly. So I calculated an average of the tests without including that test case too. I noticed that the average performance of the program actually improved with the second version of my program. However, the red herring in the data caused the total average of pizza2.mzn to be almost double of the pizza1.mzn.

Conclusions.

My exploration shows clearly that the two versions of code performed differently. It seems like pizza2 performs better when the size of n is large, however it does poorly when the size of m is large. I am not sure as to why that is, since the two values are of the same type in the respective sets of code. I believe my exploration is kind of inconclusive. Isolating the changes I made by making several iterations of code with only one change each, and then comparing their runtimes could be something for a future study, to detect what makes the Minizinc solver perform better or worse, and with which values.