CMPT 417

PIZZA COUPON PROBLEM

Jing Wen - 301327176

Instructor: David Mitchell

Description

We have a list of pizzas we want to buy, each with a price. We also have a collection of "buy x, get y free" coupons, each of which can be used to get a specified number of pizzas free, provided we have paid for some other specified number of pizzas. Each pizza that we get free by applying a coupon c must have a price no more than that of the cheapest paid-for pizza used to justify using coupon c. For example, coupon(2,4), which means if you pay for 2 pizzas then you can obtain for free up to 4 pizzas as long as they each cost no more than the cheapest of the 2 pizzas you paid for. The aim is to obtain all the ordered pizzas for the least possible cost. Note that not all vouchers need to be used, and a voucher does not need to be totally used.

Representing Problem

I am working on optimize version of the problem, assumption I am made is given correct data and data format, there will always be an answer for it. Language I am using is Minizinc, and tool is MinizincIDE, solver is Chuffed 0.10.4.

Instance vocabulary is [price, buy, free, n, m] where price, buy, and free are unary function symbols; n and m are constant symbols. So an instance structure consists of:

1. the number n of pizzas;
2. the number m of coupons;
3. Unary function price: [n] -> N, giving the price for each of the n pizzas;
4. Unary function buy : [m] → N, giving the number of paid pizzas required to justify using each of the m coupons;
5. Unary function free : [m] → N, giving the number of free pizzas that can be obtained by using each of the m coupons;
6. Cost bound k ∈ N.

Additional vocabulary symbols:

1. Unary relation symbol Paid, for the set of pizzas will be paid for;
2. Unary relation symbol Used, for the set of coupons that will be used;

3. Binary relation symbol Justifies, where Justifies(c, p) holds if pizza p is one of the pizzas we will pay for to justify using coupon c;
4. Binary relation symbol UsedFor, where UsedFor(c, p) holds if p is one of the pizzas we get free by using coupon c.

Formulas need to be satisfied:

1. We pay for exactly the pizzas that we don't get free by using coupons:

$$\forall p[Paid(p) \leftrightarrow \neg \exists c\ UsedFor(c, p)]$$

2. Used is the set of coupons that we use:

$$\forall c[Used(c) \leftrightarrow \exists p\ UsedFor(c, p)]$$

3. Any coupon that is used must be justified by sufficiently many purchased pizzas:

$$\forall c[Used(c) \rightarrow \#(p, Justifies(c, p)) \geq buy(c)]$$

4. The number of pizzas any coupon is used for is not more than the number it allows us to get free:

$$\forall c[\#(p, UsedFor(c, p)) \leq free(v)]$$

5. Each free pizza costs at most as much as the cheapest pizza used to justify use of the relevant coupon:

$$\forall c \forall p1 \forall p2[(UsedFor(c, p1) \wedge Justifies(c, p2)) \rightarrow price(p1) \leq price(p2)]$$

6. We pay for every pizza used to justify use of a coupon:

$$\forall p \forall c[Justifies(c, p) \rightarrow Paid(p)]$$

7. The total cost is not too large:

$$sum(p, Paid(p), price(p)) \leq k$$

8. We also must require that Justifies(c, p) and UsedFor(c, p) hold only of pairs consisting of a coupon and a pizza. (For example, if there are 3 coupons and 5 pizzas, we would not want Justifies(4, 5) to hold, as we might then be getting pizza 5 free by applying the non-existent coupon number 4.)

$$\forall c \forall p[Justifies(c, p) \rightarrow (c \in [m] \wedge p \in [n])]$$
$$\forall c \forall p[UsedFor(c, p) \rightarrow (c \in [m] \wedge p \in [n])]$$

9. Any pizza can be used only once to justify a coupon

$$\forall c1 \forall c2[\text{Justifies}(c1, p) \wedge \text{Justifies}(c2, p) \rightarrow c1 = c2]$$

Solver and Code

% Instance vocabulary[price, buy, free, n, m], an instance structure consists of:

% n: the number of pizzas
% m: the number of coupons
% price: [n] -> N, giving the prices for each of the n pizzas
% buy: [m] -> N, giving the number of paid pizzas required to justify using each of the m coupons
% free: [m] -> N, giving the number of free pizzas that can be obtained by using each of the m coupons
% cost bound k ∈ N

```
int: n;

set of int: Pizzas = 1 .. n;
array[Pizzas] of int: price;
int: m;


array[Coupons] of int: buy;
array[Coupons] of int: free;

set of int: Coupons = 1 .. m;

% Vocabulary symbols:

% Paid: the set of pizzas will be paid for
% Used: the set of coupons that will be used
% Justifies(c, p): holds if pizza p is one of the pizzas we will pay
for to justify using coupon c
% UsedFor(c, p): holds if p is one of the pizzas we get free by using
coupon c
array[Coupons, Pizzas] of var bool: UsedFor;
array[Coupons, Pizzas] of var bool: Justifies;


var set of Pizzas: Paid;
var set of Coupons: Used;

var int: cost = sum(i in Pizzas) (bool2int(i in Paid) * price[i]);
```

```
%Formulas to satisfy(constraints)
% 1. We pay for exactly the pizzas that we don't get free by using
coupons:
constraint forall(p in Pizzas)((p in Paid) <-> not exists(c in
Used)(UsedFor[c, p]));

%2. Used is the set of coupons that we use:
constraint forall(c in Coupons)((c in Used) <-> exists(p in
Pizzas)(UsedFor[c,p]));

%3. Any coupon that is used must be justified by sufficiently many
purchased pizzas:
constraint forall(c in Coupons) ((c in Used) -> sum(p in
Pizzas)(bool2int(Justifies[c,p] )) >=buy[c]);

%4. The number of pizzas any coupon is used for is not more than the
number it allows us to get free:
constraint forall(c in Coupons)((c in Used) -> (sum(p in
Pizzas)(bool2int(UsedFor[c, p])) <= free[c]));

%5. Each free pizza costs at most as much as the cheapest pizza used
to justify use of the relevant coupon:
constraint forall(c in Coupons) (
  forall (p1, p2 in Pizzas where p1 != p2) (
    ((UsedFor[c,p1] /\ Justifies[c,p2]) -> (price[p1] <=
price[p2])) ));
%6. We pay for every pizza used to justify use of a coupon:
constraint forall (p in Pizzas)(
  forall(c in Coupons) (
    Justifies[c,p] -> (p in Paid)));

%7. The total cost is not too large:
constraint cost <= sum(p in Pizzas)(price[p]);

%8. We also must require that Justifies(c, p) and UsedFor(c, p) hold
only of pairs consisting of a coupon and a pizza. (For example, if
there are 3 coupons and 5 pizzas, we would not want Justifies(4, 5) to
hold, as we might then be getting pizza 5 free by applying the non-
existent coupon number 4.)
constraint forall(c in Coupons) (
    forall(p in Pizzas) (
      (Justifies[c,p]) -> (
         ((1 <= c ) /\ (c <= m)) /\ ((1 <= p) /\ (p <= n)))
    ));

constraint forall(c in Coupons) (
    forall(p in Pizzas) (
      (UsedFor[c,p]) -> (
         ((1 <= c ) /\ (c <= m)) /\ ((1 <= p) /\ (p <= n)))
    ));
```

```
%9. Pizza can be used only once to justify a coupon
constraint forall(c in Coupons) (
  forall(p in Pizzas)
    (Justifies[c,p] -> not exists(c2 in Coupons where c !=
c2)(Justifies[c2,p])
));


solve minimize cost;
output ["cost(" ++ show(cost) ++ ")\n"]
```

Results Table

Input	Output	Run-Time

n = 4;
price = [10,5,20,15];
m = 2;
buy = [1,2];
free = [1,1];
	cost(35).	177msec

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];
	cost(35).	152msec

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];
	cost(340).	638msec

Testing

- Test1: given in LP/CP Programming Contest 2015
  n = 4;
  price = [10,5,20,15];
  m = 2;
  buy = [1,2];
  free = [1,1];

- Test2: given in LP/CP Programming Contest 2015
  n = 4;
  price = [10,5,20,15];
  m = 2;
  buy = [1,2];
  free = [1,1];

- Test3: given in LP/CP Programming Contest 2015
  n = 4;
  price = [10,5,20,15];
  m = 2;
  buy = [1,2];
  free = [1,1];

- Test4：test instances when pizzas have mixed prices
  n = 4;
  price = [10,5,20,15];
  m = 2;
  buy = [1,2];
  free = [1,1];

- Test5：test instances when n is large
  n = 9;
  price = [7,20,80,47,54,68,46,38,11];
  m = 4;
  buy = [3,5,1,4];

- Test6：test instances when pizzas have same prices

n = 6;
price = [10,10,10,10,10,10];
m = 3;
buy = [8,4,3];
free = [7,3,3];

- Test7：test instances when pizzas price have great range
  n = 5;
  price = [44,20,97,14,100];
  m = 10;
  buy = [1,4,4,3,1,1,2,4,1,3];
  free = [3,4,4,4,1,4,2,2,1,1];

- Test8：test instances with not obvious choice of coupons
  n = 5;
  price = [17,6,88,68,44];
  m = 10;
  buy = [3,2,3,3,4,1,3,3,1,3];
  free = [1,4,1,4,3,2,2,1,4,2];

- Test9：test instances when only buy one pizza
  n = 1;
  price = [22];
  m = 10;
  buy = [4,1,4,2,1,3,4,4,4,2];
  free = [3,4,4,1,4,4,2,3,2,3];

- Test10：test instances when have multiple choices of coupons
  n = 9;
  price = [7,20,80,47,54,68,46,38,7];
  m = 4;
  buy = [3,5,1,4];
  free = [1,1,1,1];

Data Generator

During the time I were analyzing the problem, one pain point is to input the data and follow the rule like the price number matches the number of pizza. So I create a data generator using python and numpy library, which enable users to generate as many files as they want and customize the number of pizza and coupons.

Analysis Performance

Thanks to the generator, I can generate a larger input size, for example, more than 20 pizzas. As the above fig 1 shows, before the number of pizzas reaches 27, the solver has a great performance with efficiency, which is all under 3 minutes(180 secs). While the number of pizzas equals 28 and 30, the running time comes to 12 hours and 25 hours long. The running time increases linearly.

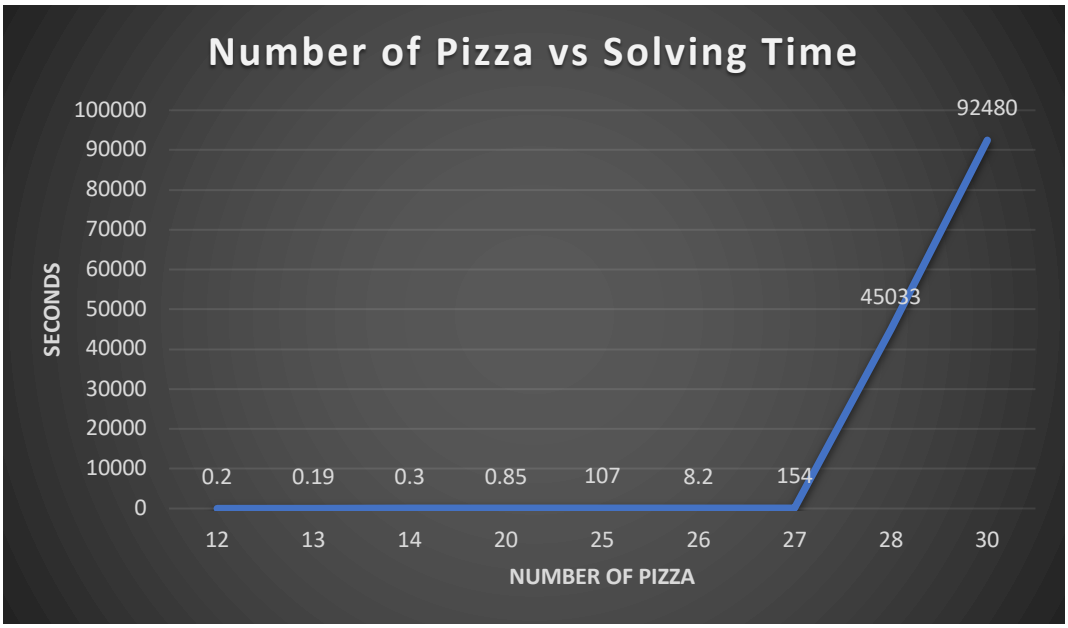

Fig 1

To be better analyze the solver performance, I did another test with price input are all the same, for example, price : [10, 10, 10, 10, 10……], with same coupons input as the previous experiment and under same solver configuration and language, unfortunately, the same price worse the solver performance.

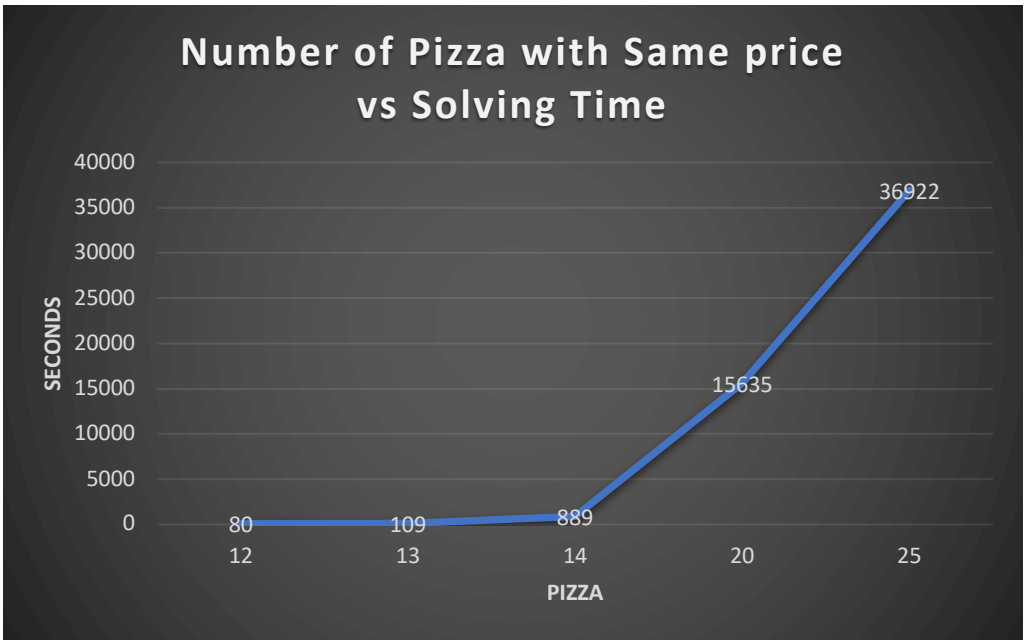

Fig 2

As the number of pizza increasing, the total solving time is increasing as well, especially when the number of pizza over 27. When n equals 20 and 25, the solver took 4.3 hours, and 10.25 hours to solve the problem respectively. As the above line graph shows, the solving time increasing dramatically from 14 pizzas, and the time is increasing linearly as the number of pizza increasing.

With those multiple experiences, I conclude with the following result:

1. Remove any one of the existing constraints will not decrease the solving time
2. Pizza with the same price will make the solver performance much worse

Improve the performance

Add Constraints

The first method I implemented is to constraint the cost with the total price:

```
int: total = sum(price);
constraint cost <= total;
```

The initial goal is to save searching time for the solver, remove unnecessary expanding. But unfortunately, from the above figure 3, I cannot confirm it make an improvement:

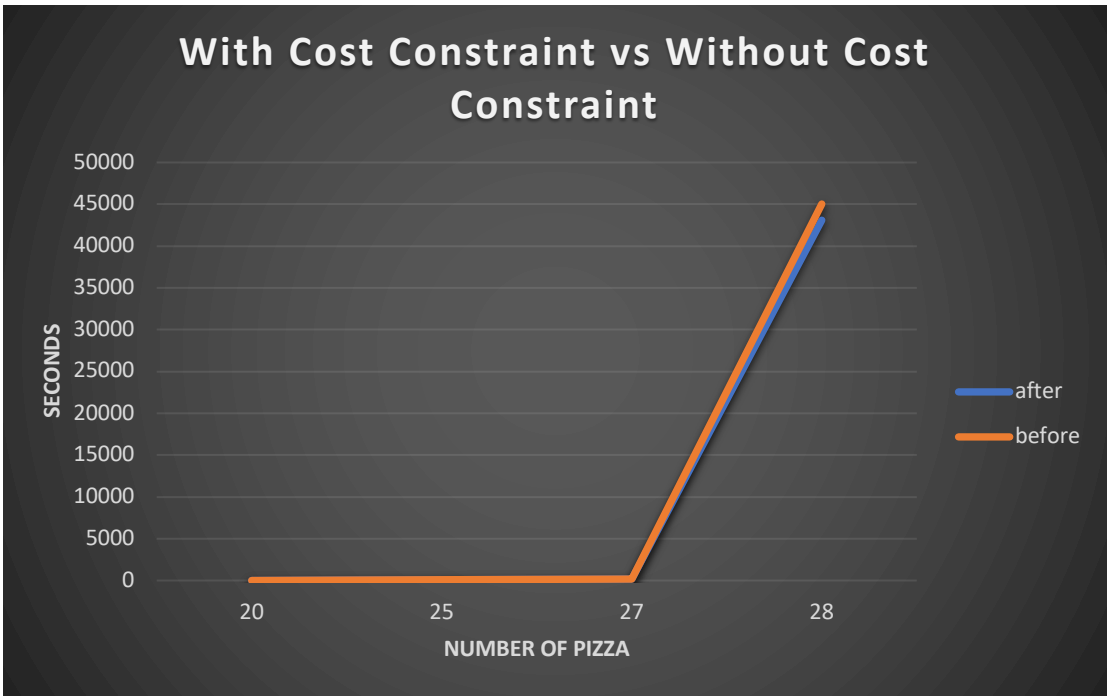

Fig3

Change Data Structure Set to Array

Another method is to change paid and used Set to Array. The solver is built by C++, according to the OOP language rule, Array is cheaper to access and read in terms of time than Set, so the code is changing in the following way:

```
var set of Pizzas: Paid;
var set of Coupons: Used;
```

```
array[Pizzas] of var bool: Paid;
array[Coupons] of var bool: Used;
```

And for the constraints:

```
bool2int(i in Paid)
```

```
bool2int(Paid[i])
```

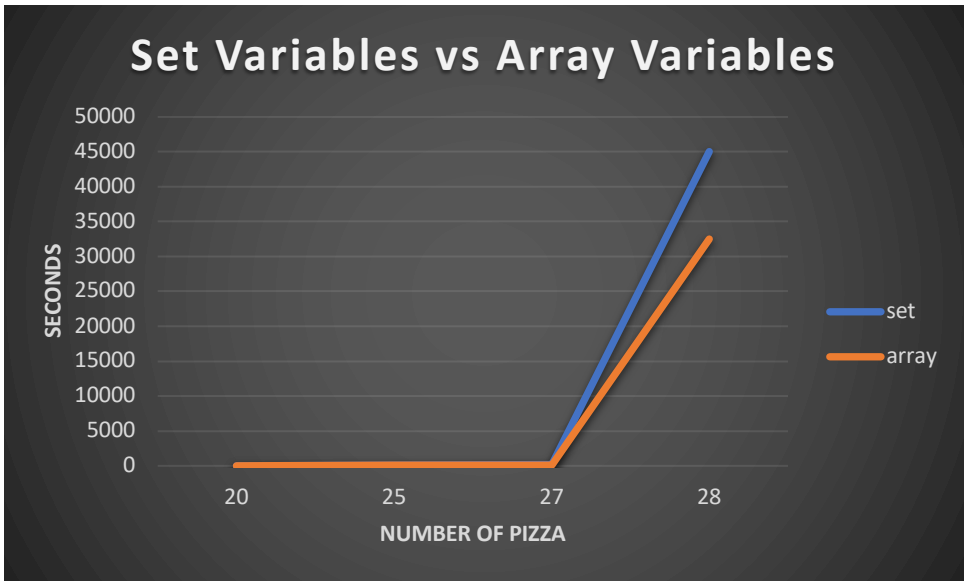

Fig 4

From the above figure 4, we can see there is a small improvement, for 28 pizzas with different prices it dropped 35% with around 3 hours reduced.

Limitation and Future Goal

Until the time I finished the report, I have not found an efficient way to reduce the same price pizza solving time, I suspect it related to the constraints, I will focus on the problem in my next goal. Another limitation is the laptop I am working on all the comparison is a little old, it may be one of the reason for the solving time for pizza with greater than 27 will take 10 hours. For your reference, here is my laptop's information:

Processor 2.4 GHz Dual-Core Intel Core i5
Memory 8 GB 1600 MHz DDR3
Startup Disk Macintosh HD
Graphics Intel Iris 1536 MB
Serial Number C02M6YYDFH00

Summary

My exploration answered part of my question, is there better way to improve the performance? By selecting more economy data structure to suite your goal and test instances with different options to compare against each methods.
From this project, I have deeper understanding of declarative problem solving and gain the hands-on experience from it.

Appendix

- pizza.mzn
  - Part 1 version code
  - language using: Minizinc
- pizza-optimize-version.mzn
  - Part 2 version code
  - language using: Minizinc
  - Including newly added #10 constraint
  - Including change Paid Set to Array
- generator.py
  - A python file to generate .dzn file for pizza solver by running:
    - e.g. python3 generator.py 2 27 5
    - the above command will generate 2 .dnz file each has 27 pizza prices and 5 coupons under the /data folder
- /data
  - Contains 10 test which explained in the above testing section
  - Once run the generator, newly generated data will be there
- report.pdf