# CMPT417 Project Report - Pizza Problem Part I

Erbo Shan 301243663 erbos@sfu.ca

## 1 Introduction

The pizza problem is originally from LP/CP Programming Contest 2015. this problem arises in the University College Cork student dorms. They are preparing for a big party, and they need to order large amount of pizza. A lot of student from that college have many of coupons so that they could get free pizza in purchasing pizzas.

For each coupon there is a constrain which requires student buy x pizzas in order to get y free pizza. A coupon is a pair of numbers, for example, $\langle 3, 1\rangle$, this indicates that students have to buy three pizzas and they are able to claim 1 free pizza, but the maximum cost of free pizzas cannot be greater than the cheapest pizza they pay for. Say students have coupon, $\langle 5, 3\rangle$, student could get three free pizzas with purchasing 5 pizzas.

$$\max_{cost(free(pizza))} < \min_{cost(buy(pizza))}$$

For this project, we need to minimize the total cost so that we could pay the least amount money for desired number of pizzas. In another word, we need to find optimized solution for this problem instead of finding a satisfiable answer. We could have a trivial total cost by adding up all pizzas' prices without applying any coupons, this value will be the maximum cost. Let $k$ denote the maximum cost:

$$k = \sum_{i=1}^{n} cost(pizza[i])$$

We need to find the minimum cost with satisfying all the constraints, let *miniCost* denote the smaller cost, and we keep seeking for other satisfiable cost which has to be smaller than our previous *miniCost* then we overwrite our new *miniCost* to the old *miniCost*. We repeat this process til we cannot find such *miniCost* smaller than previous *miniCost*, which means we have the least possible value of *miniCost*. So the basic idea is keep try different combinations and find the such combination will lead to the least cost.

Ultimately, we will use **MiniZinc** solver to find the minimum cost.

## 2 Specifications, Constraints & Implementation

### 2.1 Specifications

Pizza Problem is defined as follow:

- We have a list of pizzas we could buy, and for each pizza there is a corresponding price.
- We have a list of coupons, $\langle x, y\rangle$, which means you need buy x pizzas and you will be eligible to get y free pizzas.
- For the each free pizza, the maximum cost of free pizzas cannot be greater than the minimum cost of the price of purchased pizzas.

We can define out instance vocabulary as [*price*, *buy*, *free*, $n$, $m$, $k$ ], *price*, *buy*, and *free* are unary function symbols and $n$, $m$ and $k$ are constant symbols.

- $n$ is the number of pizzas we want to obtain.
- $m$ is the number of coupons we have.
- $k$ is the total cost without applying coupons, or we can say, $k$ is the upper bound of total cost.
- *price* : [n] $\rightarrow \mathbb{N}$ is a unary function, it indicates the price of each pizza
- *buy* : [m] $\rightarrow \mathbb{N}$ is a unary function, it indicates the number of pizza you have to buy to claim free pizzas for coupon $c$.
- *free* : [m] $\rightarrow \mathbb{N}$ is a unary function, it indicates the number of pizza you can get for free for each coupon $c$.

We need to extend instance vocabulary by adding some relation symbols [*Paid*, *Used*, *Justifies*, *UsedFor* ]

- *Paid* is a unary symbol indicates the set of pizzas we purchase.
- *Used* is a unary symbol indicates the set of coupons we use.
- *Justifies* is a binary symbol, *Justifies(c, p)*, if we use purchased pizza $p$ to claim coupon $c$ then *Justifies(c, p)* holds.

- *UsedFor* is a binary symbol, *UsedFor(c, p)*, if we use coupon $c$ to get pizza $p$ then *UsedFor(c, p)* holds.

## 2.2 Constraints & Implementation

There are 9 constraints which could lead to the minimized result, we will discuss those constraints later. Before we dive into implementations, let's get familar with the solver, **MiniZinc**. Basicly, MiniZinc is a constraint modeling language, it allows user specify constrained optimization and decision problem. The biggest advantage MiniZinc has is that it allows user to write models/constraints as writing First Order Logic/ Second Order Logic formulas. So you could implement and optimize problems in a high-level.

We will use following syntax *(some examples show the syntax of MiniZinc, we will go deep later)* for implementations of constraints:

- **int: n;** asks user to enter a value for n.
- **var 1..n: x;** the range of $x$ is between 1 and n.
- **set of int: pizzas = 1..n;** defines a set of integer from 1 to n.
- **array[pizzas] of int: price;** creates an array of integer with length of pizzas.
- **array[coupons, pizzas] of var bool: UsedFor;** defines a binary relation *UsedFor* which indicates which pizza $p$ is used for claiming free pizza by using coupon $c$.
- **array[coupons] of var bool: Used;** initializes an array of Boolean with length of coupons, and it indicates which coupon $c$ is claimed.
- **forall, exists, not** are the keywords for $\forall$, $\exists$ and $\neg$.
- /\, \/ are relation symbols for $\wedge$ and $\vee$, the syntax is compound of backslach and forceslash.
- $>$, $<$ are relation symbols for greater and smaller.

### 2.2.1 Define functions and relations

Now, we have certain sense of syntax of MiniZinc, and we can dive into transforming Fist Order Logic formulas into code.

- First of all we need number of pizza and number of coupon:

```
int: n;
set of int: pizzas = 1..n;
int: m;
set of int: coupons = 1..m;
```

- After that we need to define unary functions:

```
array[pizzas] of int: price;
array[coupons] of int: buy;
array[coupons] of int: free;
```

- Some relation symbols will come in handy:

```
array[coupons, pizzas] of var bool:
                                Justifies;
array[coupons, pizzas] of var bool:
                                UsedFor;
array[pizzas] of var bool: Paid;
array[coupons] of var bool: Used;
```

### 2.2.2 Define Constraints

We need to define and use constraints to find a valid solution, if and only if solution satisfies all the constraints.

- Constraint 1: Set upper bound of minimum cost and make sure it will not be greater than the upper bound.

  minCost is the least cost to obtain all pizzas.
- Constraint 2: We only pay for the pizzas which is not the free pizzas by applyting coupon.
- Constraint 3: The *Used* set contains all the coupon we used.
- Constraint 4: We can only apply coupon $c$ if there is sufficient number of pizza we purchase.
- Constraint 5: We cannot apply more than one coupon $c$ to the pizza $p$ which is justified for free pizza (*this one is missing in the lecture notes*).
- Constraint 6: For each coupon $c$, the number of free pizza we get cannot be greater than the number on coupon $c$.
- Constraint 7: The maximum price of free pizzas we claim cannot be greater than the minimum price of purchased pizzas.

- Constraint 8: For each pizza $p$, if we use $p$ to justify/claim coupon$c$, $p$ has to be paid, which mean $p$ is in *Paid*.
- Constraint 9: We cannot justify/claim coupon $c$ which is not in the coupon set.

### 2.3 Constraints Implementation

The following implementations are based on the constraints we define above.

```
var int: minCost = sum(p in pizzas
        where Paid[p] == true)(price[p]);
var int: maxCost = sum(price);

constraint forall(p in pizzas)
        (Paid[p] <-> not exists
        (c in coupons)(UsedFor[c,p]));

constraint forall(c in coupons)
        (Used[c] <-> exists
        (p in pizzas)(UsedFor[c,p]));

constraint forall(c in coupons)
        (Used[c] -> sum(p in pizzas )
        (Justifies[c,p] == true) >= buy[c]);

constraint forall(c in coupons)
        (Used[c] -> (sum(p in pizzas)
        (UsedFor[c,p]) <= free[c]));

constraint forall(p in pizzas)
        ((sum(c in coupons)
        (Justifies[c,p])<=1));

constraint forall(c in coupons)
        (forall (p1 in pizzas)
            (forall (p2 in pizzas)
            (( p1 != p2 /\ UsedFor[c,p1]
            /\ Justifies[c,p2])
            -> (price[p1] <= price[p2]))));

constraint forall (p in pizzas)
        (forall(c in coupons)
        (Justifies[c,p] -> Paid[p]));

constraint forall(c in coupons)
        (forall (p in pizzas)
        (Justifies[c, p] ->
        (c in coupons /\ p in pizzas)));
constraint forall(c in coupons)
        (forall (p in pizzas)
        (UsedFor[c,p] ->
        (c in coupons /\ p in pizzas)));
```

We need to find minimum cost:

```
solve minimize minCost;
```

## 3 Test Instances

In this section, we will use 12 test instances to test if our implementations are correct. The first three test cases are from the contest website, because there are correct result we could make sure our program is working properly. To run test case, you can simply run this command:

```
minizinc --solver Gecode
            cmpt417Pizza.mzn test*.dzn
```

If you did not include path of MiniZinc to your system. you shall run this command:

```
export PATH=/Applications/
    MiniZincIDE.app/Contents/Resources:$PATH
```

### 3.1 Test Instance 1:

### 3.2 Test Instance 2:

### 3.3 Test Instance 3:

Test 1 to 3 are from contest website which could help us make sure if our program generate result correctly.

```
# Test 1
n = 4;
price = [10,5,20,15];
m = 2;
buy = [1,2];
free = [1,1];
# Test 2
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 3
```

```
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 Result:**

```
# Result of Test 1:
Minimum cost:
  35
Maximum cost:
  50

# Result of Test 2:
Minimum cost:
  35
Maximum cost:
  60

# Result of Test 3:
Minimum cost:
  340
Maximum cost:
  510
```

### 3.4 Test Instance 4:

There is no such a purchased pizza is used to claim more than 1 coupon:

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

**Test Result:**

```
Minimum cost:
  20
Maximum cost:
  30
```

### 3.5 Test Instance 5:

The minimum cost of free pizza is no more than the cheapest purchased pizza:

```
n = 10;
price = [70,10,60,60,30,100,60,40,60,20];
m = 1;
buy = [1];
free = [10];
```

**Test Result:**

```
Minimum cost:
  100
Maximum cost:
  510
```

### 3.6 Test Instance 6:

The number of pizza we obtain is always equal the number of pizza we want to obtain. In a word, if the coupon $c$ is buy one get ten free, but we want to have ten pizza, $c$ cannot be used:

```
n = 10;
price = [100, 100, 100, 100, 100,
         100, 100, 100, 100, 100];
m = 1;
buy = [10];
free = [1];
```

**Test Result:**

```
Minimum cost:
  1000
Maximum cost:
  1000
```

### 3.7 Test Instance 7:

This test case is extended version of *Test Instance 6*, the most expensive free pizza cannot be greater than the least cost of purchased pizza. In test instance 7, we have 5 coupon, and each of them is buy one get one free:

```
n = 10;
price = [100, 90, 80, 70, 60,
         50, 40, 30, 20, 10];
m = 5;
buy = [1, 1, 1, 1, 1];
free = [1, 1, 1, 1, 1];
```

**Test Result:**

```
Minimum cost:
  300
Maximum cost:
  550
```

### 3.8 Test Instance 8:

If there is no coupon we can use, we have to buy all pizzas:

```
n = 18;
price = [1,4,3,2,4,1,3,4,1,5,
        3,4,3,4,6,5,6,3];
m = 0;
buy = [];
free = [];
```

**Test Result:**

```
Minimum cost:
  62
Maximum cost:
  62
```

### 3.9 Stress Testing:

We will do four stress testings to see how is its performance. This snippet of python code will help us generate test case:

```
# generate random integer values
from random import seed
from random import randint
seed(10)
    combination = [[15,30], [15, 30], [25, 30]]
    pizza = [], buy = [], free = []
    for i in combination:
        pizzas = i[0]
        coupon = i[1]
        for _ in range(pizzas):
            value = randint(10, 50)
            pizza.append(value)
        for _ in range(coupon):
            value = randint(10, 20)
            buy.append(value)
            value = randint(5, 10)
            free.append(value)
        pizza.clear()
        buy.clear()
        free.clear()
```

#### 3.9.1 Stress Testing 1:

```
n = 15;
price = [27, 82, 18, 42, 25, 73,
         67, 70, 93, 58, 36, 22, 72, 13, 59];
m = 30;
buy = [23, 29, 10, 24, 18, 17,
        28, 13, 20, 10, 10, 10,
        30, 27, 10, 22, 16, 23,
        10, 26, 17, 24, 25, 27,
        17, 21, 17, 17, 24, 19];
free = [5, 8, 9, 10, 5, 6, 10,
        10, 7, 5, 10, 7, 10, 10,
        9, 8,9, 10, 6, 7, 7, 9,
        8, 9, 8, 9, 5, 8, 6, 10];
```

**Test Result:**

```
Minimum cost:
  652
Maximum cost:
  757
```

#### 3.9.2 Stress Testing 2:

We are still using 15 as total pizza number, and 30 as coupon number. You may notice that, in 3.9.1, a lot of coupon are not valid, because we just want 15 pizzas, and more than 50 percent coupons require purchasing more than 15 pizza so that you could claim free pizza. So this test we need make sure each coupon is asking us to purchase less than 15, and see how its performance will be.

```
n = 15;
price = [46, 45, 39, 41, 24, 30,
        20, 49, 27, 40, 29, 29,
        42, 45, 43];
m = 30;
buy = [11, 14, 11, 9, 9, 9, 14,
       12, 12, 13, 10, 7, 15, 7,
       12, 12, 10, 8, 11, 14, 13,
       9, 13, 14, 10, 15, 11, 15,
       14, 14];
free = [5, 6, 8, 10, 8, 8, 6, 8,
        9, 9, 7, 9, 8, 8, 10, 6,
        9, 8, 5, 8, 5, 5, 8, 9,
        5, 5, 7, 5, 6, 5];
```

**Test Result:**

```
Minimum cost:
  311
Maximum cost:
  549
```

**This test case takes more than 60 sec, we shall see the performance drops significantly, compared with test cases above**

#### 3.9.3 Stress Testing 3:

```
n = 25;
price = [48, 39, 41, 18, 39, 37,
         42, 38, 31, 26, 39, 37,
         31, 43, 43, 19, 25, 30,
         49, 12, 22, 47, 46, 50, 40];
m = 30;
buy = [15, 20, 18, 18, 12, 13,
       15, 19, 10, 20, 13, 20,
       19, 19, 20, 17, 10, 11,
       14, 16, 13, 14, 11, 10,
       19, 18, 11, 14, 12, 20];
free = [8, 6, 7, 9, 8, 6, 7,
        6, 5, 9, 6, 8, 6, 5,
        10, 7, 8, 10, 10, 9,
        10, 8, 8, 10, 10, 9,
        6, 7, 9, 7];
```

**Test Result:**

```
Minimum cost:
  559
Maximum cost:
  892
```

#### 3.9.4 Stress Testing 4:

```
n = 25;
price = [38, 16, 36, 12, 10, 38,
         44, 14, 13, 32, 15, 19,
         16, 48, 39, 41, 18, 39,
         37, 42, 38, 31, 26, 39,
         37];
m = 30;
buy = [15, 20, 18, 18, 12,
       13, 15, 19, 10, 20,
       13, 20, 19, 19, 20,
       17, 10, 11, 14, 16,
       13, 14, 11, 10, 19,
       18, 11, 14, 12, 20];
free = [8, 6, 7, 9, 8, 6,
        7, 6, 5, 9, 6, 8,
        6, 5, 10, 7, 8, 10,
        10, 9, 10, 8, 8, 10,
        10, 9, 6, 7, 9, 7];
```

**Test Result:**

```
Minimum cost:
  470
Maximum cost:
  738
```

# 4 Benchmark

In this section, we will do some benchmarks based on different aspects, such as different solvers, different test cases and different numbers of threads.

## 4.1 Default settings

We will run all test cases on default settings, which means all solvers are using only **one** thread, and we will run each test cases **10** times. We will remove the highest and the lowest, and calculate the **mean**.

| Test | Gecode6.11 | Chuffed | COIN-BC |
|---|---|---|---|
| 1 | 134 | 128 | 102 |
| 2 | 122 | 135 | 122 |
| 3 | 152 | 132 | 242 |
| 4 | 125 | 132 | 127 |
| 5 | 150 | 204 | 135 |
| 6 | 132 | 149 | 148 |
| 7 | 388 | 220 | 1765 |
| 8 | 152 | 149 | 103 |
| 9 | 807 | 372 | 1211 |
| 10 | 61100 | 473 | 1284 |
| 11 | N/A | 18350 | 120400 |
| 12 | N/A | 2692 | 60120 |

**The unit is** *ms*

## 4.2 Two Threads:

Since we cannot apply multi-threads on **Chuffed**, I jest leave it as it is in *Default Settings*, and we can remove **Chuffed** column from the table.

| Test | Gecode6.11 | COIN-BC |
|---|---|---|
| 1 | 155 | 99 |
| 2 | 150 | 125 |
| 3 | 177 | 242 |
| 4 | 148 | 99 |
| 5 | 176 | 140 |
| 6 | 148 | 148 |
| 7 | 353 | 1793 |
| 8 | 152 | 103 |
| 9 | 582 | 1126 |
| 10 | 60800 | 1684 |
| 11 | N/A | 47713 |
| 12 | N/A | 27660 |

**The unit is** *ms*

As we can see, for test instance 9-12, there are slight improvement of performance. Let's see if increasing number of threads improves performance in complex test instances **9-12**.

### 4.3 Eight Threads:

Since we cannot apply multi-threads on **Chuffed**, I jest leave it as it is in *Default Settings*.

| Test | Gecode6.11 | COIN-BC |
|---|---|---|
| 9 | 390 | 1148 |
| 10 | 23758 | 1695 |
| 11 | 75630 | 28225 |
| 12 | N/A | 42731 |

**The unit is** *ms*

As the table shown, multi-threading slightly boost the performance, but not too much. We can conclude that the performance of different solvers are vary. In complicated test instances, **COIN-BC** and **Chuffed** are outperformed **Gecode 6.11**, and **Chuffed** is better than **COIN-BC** in any circumstances.

## References

[1] David Mitchell. *Notes on Satisfiability-Based Problem Solving Representing Problems: Examples from the 2015 LP/CP Programming Competition.* October 28, 2019