University of Cape Town

STA5071Z: Optimisation

Wine Not!

Author:
Edward Baleni
Wayne Jiang

Student Number:
BLNEDW003
JNGWEN002

May 28, 2023

Contents

1 Problem Description

Wine assemblage consists of blending a number of different wine varieties together to create a preferred new wine. This method of wine making is one of the oldest techniques used to create rich and fine bodied wines from various types of base wines, to attain certain features wanted by drinkers. Wine makers have historically performed this technique using taste, smell and their expert knowledge to find the best combination, however, there is only so much wine that wine makers can taste in a day before they begin to experience sensory specific satiety, taste saturation (Vismara et al., 2016; Meillon et al., 2012); taste saturation is a deteriorating pleasure in the consumption of a substance often from excessive consumption. To remedy this this problem has been solved by means of neural networks (Ferrier and Block, 2001), mixed integer non-linear programming (Vismara et al., 2016) and other such methods.

This study will be looking into a more simplified method to optimise the process of wine assemblage than Vismara et al. (2016). A mixed integer linear programming (MILP) approach will be used to optimise the wine blending problem. Metaheuristic approaches will be also be utilised to search the solution space for a good-enough solution. Wine assemblage is a complex problem as it over-constrains, often the boundaries are quite tight and contradictory as will be shown below. It is possible that the linear program in such a scenario does not manage to find the solution. However, by relaxing the necessity to find the optimal solution, metaheuristics can be use to find an optimal solution. Along with the classical MILP, simulated annealing (SA) and genetic algorithms (GA) will also be utilised to find these near-optimal solutions.

Often in this blending problem, wine makers would have more than one goal when producing their blend. Below, a multi-objective goal programming (MOGP) strategy is used to attempt to achieve the goals of the wine maker. For the classical linear program, minimizing cost is the only objective; for the MOGP: cost, surplus and phenols are to be minimized; tannins and anthocyanins are to maximised; alcohol level, chroma, residual sugar and titrable acidity have specific goals that must be met.

In this study, the target audience is your everyday individual, not wine makers or oenophiles. The aim is to provide wine lovers at home a blending guideline, that they are able to use to blend their cheapest wines in hopes to create a better wine, or a wine of their preference. For the MILP and metaheuristics, the measurements are specified so that you are transporting proportions of each base wine bottle into an empty wine bottle, these proportions will be the make-up of the wine. For the MOGP approach, you will no longer be constrained to one wine bottle and will be able to make a number of wines, from a number of base wine bottles. When creating the blend, the user should not use an opened wine to minimise oxidization (only open when you are ready to blend), after blending wine you should immediately bottle it in a glass bottle and store in a dark place at 10°C. If oxidization was not minimised, the blend should be ready in about 2 weeks, however, for the ideal blend and if oxidization was minimised the blend will be ready in 1 to 2 months (Cáceres-Mella et al., 2013; Li et al., 2020; Wang et al., 2022).

1.1 Data

The data was obtained from Cáceres-Mella et al. (2013). This study provided information on the physicochemical properties, chemical composition, phenolic compound composition, anthocyanin composition, mouthfeel characteristics and other such necessary components of four monovarietal wines. The wines that were considered were Cabernet sauvignon (CS), Cabernet franc (CF), Merlot (ME) and Carménère (CR). The cheapest option of each of the four wines have prices as shown on www.totalwine.com. The cheapest wines were selected to make the study more practical to those

blending at home. The prices of these wines were converted from USD to ZAR at R19.32 to the dollar.

Auxiliary variables were added to the study to explore artificial supplementation to the wines. These supplements are food colouring (FC), artificial tannins (AT) and edible alcohol (EA). With the possibility that the solution space is difficult to find, or for ease of blending, these supplements can be used as a shortcut to blending, but they will come at a price. The data is capture in Table 1.

Table 1: Data capturing the composition of each monovarietal wine

	CS	CR	ME	CF	FC	AT	EA
Price (R/750ml)	269.00	346.00	230.00	269.00	150.00	100.00	120.00
Alcohol content (% v/v)	14.50	14.60	13.70	14.50	-	-	1.00
Abrasiveness (%)	12.40	6.81	6.19	5.32	-	-	-
Hardness (%)	18.24	19.71	6.05	5.27	-	-	-
Adhesiveness (%)	24.34	19.82	5.87	15.74	-	-	-
Dryness (%)	38.56	40.33	40.92	31.69	-	-	-
Mouthcoating (%)	6.46	13.34	40.97	41.98	-	-	-
Astringency	10.62	10.81	10.69	11.98	-	-	-
Bitterness	7.61	7.77	7.43	8.26	-	-	-
pH	3.53	3.52	3.58	3.52	-	-	-
Residual Sugar (mg/L)	2800.00	2200.00	2500.00	2700.00	-	-	-
Titratable Acidity,TA (mg/L)	3530.00	3590.00	3270.00	3500.00	-	-	-
Phenols (mg/L)	893.69	922.85	794.99	1006.78	-	-	-
Tannins (mg/L)	1704.12	1866.05	1790.84	2254.66	-	100.00	-
Anthocyanins (mg/L)	486.04	707.80	393.03	342.71	-	-	-
Colour intensity, CI	17.17	23.88	14.79	17.49	2.50	-	-
Colour coordinates, L^*	37.21	28.60	41.65	36.70	-	-	-
Chroma, C	58.06	60.29	55.69	58.38	15.00	-	-
Hue, H	17.62	20.30	15.47	18.66	5.0	-	-

A number of variables in Table 1 can be considered latent variables. Some of these latent variables are the mouthfeel characteristics, such as: abrasiveness, hardness, adhesiveness, dryness, mouthcouting, astringency and bitterness. The remainder of them are related to the colour related information, such as: colour intensity (CI), Colour coordinates (L^*), chroma (C) and hue (H). In particular it should be noted that colour coordinates was given abbreviation L^* so not to confuse it with litres. The observed variables that capture these latent variables are captured by the organic compounds (i.e. phenols and anthocyanins). "Flavanols" and "non-flavanols" are individual organic compounds that fall under the class of phenols, phenols are what determines the taste of the wine and is a contributor to the amount of anti-oxidants present in the wine (Wang et al., 2022). These flavanols were are the main influence on the role of astringency and bitterness in the wine. Anthocyanins are also made up of a combination of different flavanols, impacting indicators such as mouthfeel, colour intensity, L^*, C and H of the wine. The greater the total number of anthocyanins, the greater the greater the colour properties. The blending of different wines demonstrates change as levels of flavanols in anthocyanins are observed. In particular bitterness and astringency are also changed for varying flavanols. Increases in types of anthocyanins results in an increase in either bitterness or astringency. Factors such as abrasiveness and hardness are increased when blending, this increase is an additive effect based on the flavanols. Dryness has the opposite effect

as it diminishes with blending (Wang et al., 2022)

The purpose of the description of the flavanols and their impact on blended wines was to justify the reason for their exclusion in the remainder of the study. There are a large number of flavanols that affect how the wine will turn out, it is difficult to characterise all these traits and solve in a linear fashion as it would involve an overly constraint model that will likely not converge, such a model has been explored, Vismara et al. (2016). The latent variables here are meant to capture the basic idea of how the flavanols will interact in the formulation and also capture the characteristics that the wine maker may want in a simple manner. For simplicity we will describe the ideal mouthfeel and colour of a wine by a number of latent variables and this will characterise how the observed variables will behave in kind. This simplicity helps to not over-constrain the problem.

2 Methodology

2.1 Linear Programming

Linear programming is a technique to optimize an objective that is bounded by a set of constraints. To perform this optimization we used the **Rglpk** package (Theussl and Hornik, 2023), a the standard library used to solve linear problems. It provides an interface to the "GNU Linear Programming Kit" which is is open-source software for solving linear and integer programming problems. In this study we will be optimizing a mixed integer linear problem.

2.2 Simulated Annealing (SA)

Simulated Annealing is a metaheuristic technique that is used in optimization problems to search for ideal value in a large search space. The original idea comes from statistical physics in particular the cooling of metals. The algorithm works along a search space to iteratively move closer and closer to a local-optima or a global-optima, a new solution is accepted along this path only if it is an improvement of the previous one. This algorithm is prone to stopping at local minima, to remedy this, the algorithm accepts something called "up-hill moves". With the understanding that there is a possibility of a better solution "up-hill" moves are used to push the search towards a worse solution (out of the local minima) in order to continue the search to possibly find a better local-minima or the global-minima, as an exploratory strategy. Simulated annealing allows moves resulting in solutions of worse quality than the current solution, however, there is a probability that "up-hill" move will be accepted. The probability of doing such a move is as follows:

$$Pr(\text{Accept worse quality}) = \exp\left(-\frac{f(s') - f(s)}{T}\right)$$

$f(s') - f(s) > 0$ if the new solution is worse than the current solution. As $f(s') - f(s)$ becomes larger, the probability becomes smaller. The temperature, T, will also decrease as the algorithm iterates so as to ensure that the probability of accepting an "up-hill" move decreases with an increase in iterations. This basically means the algorithm is more likely to accept worse solutions in the early iterations.

Cooling schedules are a means of updating temperature every iteration. The cooling schedules we used to update temperature are,

- Logarithmic $T_k = \frac{T_0}{(1+\alpha\log(1+k))}$
- Geometric $T_k = T_0 a^k$

As can be seen from the equation, the cooling speed of the logarithmic algorithm is determined by both starting temperature and temperature factor, it uses log which will cool down very slowly over time. The cool speed of geometric algorithm is mainly dependent on temperature factor and it will cool down much quicker in comparison to the Logarithmic method. Therefore Logarithmic is likely to explore more of the search space in comparison to Geometric and it should converge slower than the Geometric algorithm.

2.3 Genetic Algorithms

Genetic algorithms are a metaheuristic in a broader category of evolutionary algorithm. By natural selection (survival of the fittest), genetic algorithms take a population of solutions (very many solutions), perform various operations between the observations in order to manoeuvre towards an optimal solution. The steps to conduct a genetic algorithm and detailed below.

- Initialisation - The first step is to create a number of feasible solutions. These solutions will make up the population. The number of these solutions can vary depending on what has been defined by the user.
- Evaluation - The next step is to evaluate each population. This means to calculate the objective for each solution. This will allow to attain which members of the population have better fitness.
  - Fitness is a measure of how strong each member of the population is. This is determined by the objective of the function, if the objective is to minimise/maximise then the member with the lowest/highest objective will be have the highest fitness.
- Selection - After evaluation it is necessary to re-sample from the population, this diminishes the number of less feasible solutions, by obtaining a new population with better solutions than the previous solution. There are many types of selection techniques but here the tournament selection and the rank based selection will be explored.
  - Tournament - A sample of the population is selected and the fittest in this sample is selected as a member of the new population. The number of samples will determine the convergence rate, so a lower number of samples is advised in this procedure. This is performed as many times as there are samples resulting in a new population.
  - Rank based - The members of the population are ranked according to their fitness, an adaptation to the better known proportion-to-fitness selection. The members for the new population are then chosen. This is done by sampling with repetition, where each member is given a probability of being sampled according to rank.
- Crossover - After selection, the new generation has to be selected. This step generates a truly new population from the old. A pair of parents (a pair of members of the current population) are selected for breeding to create a child (a new solution that has parts of each parent) with similar characteristics to the parents; sections of the parent solutions are chosen in such a way to produce a child. At this stage the choice of crossover should not affect the algorithm very much, the reason why will be explained at the mutation, most techniques can be used. Uniform crossover and n-point crossover will be used at this stage. Since we are dealing with real-valued numbers that do not depend on some order, order-based methods may be limited in finding the solution.
  - Uniform Crossover - Each value of the corresponding children are selected by flipping a coin. The coin can be biased to contain more of one parents information. For each index

a coin is flipped, a probability above or below p is selected (where p is the probability of heads, consider it as 50% here). If the probability is above p then child 1 gets item at that index from parent 1 and child 2 gets item at that index from parent 2, if the probability is below p then child 1 gets item at that index from parent 2 and child 2 gets item at that index from parent 1.

  - N-Point crossover - Choose crossover points are selected. Between each of these points the pair of children are created by swapping between parents at each cut point and taking that portion of a parents items.

- Mutation - The purpose of mutation is to introduce diversity in the population. This aids the algorithm in avoiding local minima by helping members stay relatively dissimilar to parents. Mutation is decided by a parameter called the mutation probability. This is a small probability that informs the algorithm if the member needs to undergo mutation. As this is a real-numbered problem, where the sum of the items for each member must be 1 or at least close to 1. All members are normalized if the sum of their items do not fall within the range [0.9, 1].
  - Insert mutation - Randomly pick two items of a member and move the second next to the first and move each one up by one.
  - Scramble mutation - Select a subset of items and rearrange them randomly.

2.4 Multi-Objective Goal Programming (MOGP)

The purpose of MOGP is to optimize many objectives simultaneously. The problem that comes when doing this is that most objectives will evaluate differently depending on the constraints; as a result when more than one objective is optimized at the same time, only one or none of the objectives will be optimised. MOGP is a method used to remedy this issue. It solves the issue by looking for an acceptable solution by minimizing distance from goals. There are a number of methods that can be used to explore this, below the Archimedean and Preemptive methods are explained. Using an example with 3 goals ($\{g_k : k = 1, 2, 3\}$) the goal programming solution can be found with the following equations.

2.4.1 Archimedean Goal Programming

$$\text{Minimize:} \qquad \sum_{i=1}^{K} \{w_k^- d_k^- + w_k^+ d_k^+\}$$

Subject to: All the constraint of the single objective linear program

$$\begin{aligned} &\vdots \\ z_1 + d_1^- - d_1^+ &= g_1 \\ z_2 - d_2^+ &= g_2 \\ z_3 + d_3^- &= g_3 \end{aligned}$$

2.4.2 Preemptive Goal Programming

Preemptive goal programming can be used if there is a priority level for each goal. In our case, we assume goal 1 and 2 have the same level of priority and they are prior than goal 3. Then we will begin with minimization of goal 1 and 2.

$$\text{Minimize: } \alpha = w_1^- d_1^- + w_1^+ d_1^+ + w_2^+ d_2^+$$

Subject to: All the constraint of the single objective linear program

$$\begin{aligned} &\vdots \\ z_1 + d_1^- - d_1^+ &= g_1 \\ z_2 - d_2^+ &= g_2 \\ z_3 + d_3^- &= g_3 \end{aligned}$$

The second step will be minimization of goal 3 using constraint of minimised value of goal 1 and 2.

$$\text{Minimize: } w_3^- d_3^-$$

Subject to: All the constraint of the single objective linear program

$$\begin{aligned} &\vdots \\ z_1 + d_1^- - d_1^+ &= g_1 \\ z_2 - d_2^+ &= g_2 \\ z_3 + d_3^- &= g_3 \\ w_1^- d_1^- + w_1^+ d_1^+ + w_2^+ d_2^+ &\leq \alpha \end{aligned}$$

3 Results

3.1 Linear Programming

Variables were used to describe the linear program, which will be explained. X_i for $i = 1, ...4$ denotes the decision variables related to proportion of wine, X, from which wine, i. X_1 is the proportion of CS, X_2 is the proportion of CR, X_3 is the proportion of ME and X_4 is the proportion of CF. Y_j for $j = 1, 2, 3$ denotes the decision variables related to the amount if supplement added, this decision variable is an integer. Y_1 is the amount of FC, Y_2 is the amount of AT and Y_3 is the amount of EA. The linear problem is formulated below,

$$\text{Minimise:} \quad 269X_1 + 346X_2 + 230X_3 + 269X_4 + 150Y_1 + 100Y_2 + 120Y_3$$

Subject to:

$$
\begin{aligned}
PH: &\quad 3.53X_1 + 3.52X_2 + 3.58X_3 + 3.52X_4 < 3.55 \\
PH: &\quad 3.53X_1 + 3.52X_2 + 3.58X_3 + 3.52X_4 > 3.52 \\
Abrasiveness: &\quad 12.4X_1 + 6.81X_2 + 6.19X_3 + 5.32X_4 > 8 \\
Dryness: &\quad 38.56X_1 + 40.33X_2 + 40.92X_3 + 31.69X_4 > 35 \\
Dryness: &\quad 38.56X_1 + 40.33X_2 + 40.92X_3 + 31.69X_4 < 35 \\
Bitternes: &\quad 7.61X_1 + 7.77X_2 + 7.43X_3 + 8.26X_4 > 7.7 \\
Hue: &\quad 17.62X_1 + 20.30X_2 + 15.47X_3 + 18.66X_4 < 18 \\
Hue: &\quad 17.62X_1 + 20.30X_2 + 15.47X_3 + 18.66X_4 > 17 \\
Proportions: &\quad X_1 + X_2 + X_3 + X_4 = 1 \\
Supplements: &\quad Y_1 + Y_2 + Y_3 \leq 5 \\
Integer: &\quad Y_1, Y_2, Y_3 \in \text{Integer} \\
Positive: &\quad X_1, X_2, X_3, X_4 \geq 0
\end{aligned}
$$

We were then able to solve the linear problem above using the R function **Rglpk**. Below are the decision variables of the optimal solution:

Table 2: Mixed integer linear program solution

	X_1	X_2	X_3	X_4	Y_1	Y_2	Y_3
LP Solution	0.33	0.00	0.41	0.26	0.00	0.00	0.00

The solution in Table 2 shows the optimal proportions of wine to create the specified blend while minimizing price. The final optimal objective was given as, R252.87; the blend of wine described in Table 2 will cost R252.87.

3.2 Simulated Annealing

Simulated Annealing has been performed with different cooling algorithms:

- Geometric with starting temperature $T_0 = 1$ and temperature factor 0.995
- Geometric with starting temperature $T_0 = 1$ and temperature factor 0.95
- Logarithmic with starting temperature $T_0 = 1$ and temperature factor 0.995
- Logarithmic with starting temperature $T_0 = 0.5$ and temperature factor 0.995
- Logarithmic with starting temperature $T_0 = 1$ and temperature factor 0.8

Figure 1 is the objective optimized over each iteration for each parameter setting using the SA algorithm. As discussed in the method section, we expected Geometric algorithms to converge quicker than Logarithmic algorithms. It was also expected that the Logarithmic algorithms would explore more in comparison to Geometric algorithms. We can see two of the Geometric converging after 5000 iterations and only one Logarithmic converging within 10000 iterations.

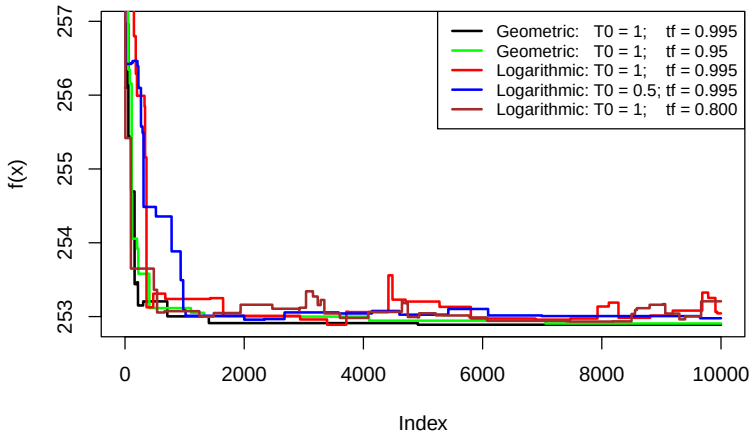

Figure 1: Simulated annealing convergence to solution

The Geometric algorithm with a temperature factor of 0.95 should converge sooner than the one which has a temperature factor of 0.995. But it is not the case on the plot.

Within Logarithmic algorithms, we expect the brown line to explore the search space the most, followed by the red line and finally the blue respectively. The plot seems to confirm this expectation, however, the blue line is the only one of the logarithmic cooling schedules that converges.

Table 3 illustrates the optimal solution for the different settings of the SA. It depicts the decision variable using the same convention of X_i and Y_j as in the MILP. The solutions are all the same and are also the optimal solution as they correspond with the solution found in the MILP section.

Table 3: Solutions found using SA

	SA1	SA2	SA3	SA4	SA5
X_1	0.33	0.33	0.33	0.33	0.33
X_2	0.00	0.00	0.00	0.00	0.00
X_3	0.41	0.41	0.41	0.41	0.41
X_4	0.26	0.26	0.26	0.26	0.26
Y_1	0	0	0	0	0
Y_2	0	0	0	0	0
Y_3	0	0	0	0	0

3.3 Genetic Algorithms (GA)

The population was initialised in such a way that each member was within the bounds of the constraints mentioned in the linear program above. This was done by repeatedly randomly selecting a range of values for each member until each member had a range of values within the feasible solution.

When evaluating, if a constraint was not obeyed then the fitness given was high. In this problem, for GA, a lower fitness was desired.

There were 8 GAs that were run to find the optimal/near-optimal solution. These were created from different permutations of methods. These methods were described above for selection, crossover and mutation. The 8 GAs are described in the list below:

- Genetic Algorithm 1 - Rank selection, uniform crossover, scramble mutation.
- Genetic Algorithm 2 - Rank selection, uniform crossover, insert mutation.
- Genetic Algorithm 3 - Rank selection, N-point crossover, insert mutation.
- Genetic Algorithm 4 - Rank selection, N-point crossover, scramble mutation.
- Genetic Algorithm 5 - Tournament selection, N-point crossover, insert mutation.
- Genetic Algorithm 6 - Tournament selection, N-point crossover, scramble mutation.
- Genetic Algorithm 7 - Tournament selection, uniform crossover, insert mutation.
- Genetic Algorithm 8 - Tournament selection, uniform crossover, scramble mutation.

For the tournament selection, the number of samples was chosen to be 5, these value gave convergence for most seeds set. A value of 3 sometimes arrived convergence but sometimes did not, when it did converge it was not as early as when the sample size was set to 5 since more possible solutions could be searched before coming close to the correct one. In the case of uniform crossover, the probability was set to 0.5, to allow both parents an equal opportunity to give their children their genes (items). In the mutation stage, since this is a real-number problem, it would not damage the solution to deviate the values in some way; in order to maintain the constraint that the proportions must be equal to 1, the continuous numbers were normalized before mutated if their sum was out of the range [0.9, 1]. For both mutations, the mutation rate was set to 0.05 as this was an adequate probability of mutation.

In Figure 2 we see the convergence of each of the 8 GA. It shows that GA 1 to 4 do not minimize and that GA 5 to 8 do. This was tested over multiple seeds, on some special occasions, one the first 4 GAs would minimise and convergence would be much slower than can be seen in the last 4 GAs. However, the seed here has been set to 1 and there is a clear difference between how the first 4 and the last 4 GAs handle the data. The first 4 GAs all used rank selection and the last 4 all used tournament selection; this shows that rank selection does not particularly work well in the case of wine assemblage. Looking at the last four GAs, different permutations of the crossover and mutation methods were utilised and each managed to minimize the solution, this would demonstrate that the other parts of the GA work well with wine assemblage but that rank selection does not.

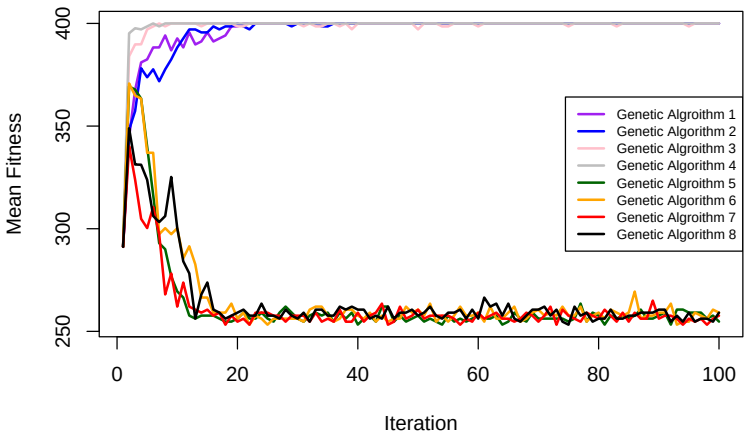

Figure 2: Genetic algorithm convergence diagram

Table 4 illustrates the fitness at the final generation. As shown in Figure 2 the first 4 do not converge but the last four do. These values illustrate a minimum price of R256.09 to R262.09, which is not as obtimal as the MILPs minimum price of R252.87, but is still adequate. In particular, GA5 has the best solution of the 8 methods tried.

Table 4: Fitness at the terminal generation

GA1	GA2	GA3	GA4	GA5	GA6	GA7	GA8
400.00	400.00	400.00	400.00	259.14	262.09	256.13	257.57

The decision variables of the last 4 GAs' solutions are captured in Table 5 using the same convention of X_i and Y_j as in the MILP. This simply illustrates the proportions of wine to use to minimise price while still getting the wine that you would desire. It would be advised to use the proportions shown by GA5. It can be seen that much like the MILP, that it is not worth it to add any supplements to help in wine making. These solutions unlike in SA are not identical to the MILP; GA are not exactly designed for real-valued problems, so it is justified to observe it have poor performance in comparison to the options above. With this said the sub-optimal solution is still good.

Table 5: Decision variables

	X_1	X_2	X_3	X_4	Y_1	Y_2	Y_3
GA5	0.329	0.004	0.405	0.262	0.000	0.000	0.000
GA6	0.329	0.004	0.403	0.263	0.000	0.000	0.000
GA7	0.328	0.004	0.411	0.256	0.000	0.000	0.000
GA8	0.327	0.004	0.414	0.254	0.000	0.000	0.000

3.4 Multi-Objective Goal Programming

The additional goals added to the formulation in this section are:

- Achieve target alcohol level: 15
- Achieve certain level of chroma: 59
- Achieve target residual sugar: 2300
- Maximise tannins: 2458.85
- Maximise anthocyanins: 414.91

To calculate weight for each goals, the importance of each objective was first set. Following this each objective was maximized to obtain an optimised value for each variable, for the solution of each the performance of the other objectives are recorded, this allows to record the range of values of each variable. The weight is then calculated as, $\frac{10\times\textbf{Importance}_i}{\textbf{Range of value}}$. Table 6 illustrates the procedure mentioned.

Table 6: Illustrating how weights for MOGP are calculated

Objective Maximised	Value Obtained z1	z2	z3	z4	z5	z6
z1	860.40	19.32	57.67	2691.01	1958.85	404.18
z2	860.40	19.32	57.67	2691.01	1958.85	404.18
z3	260.40	14.32	57.67	2691.01	1958.85	404.18
z4	260.40	14.32	57.67	2691.01	1958.85	404.18
z5	760.40	14.32	57.67	2691.01	2458.85	404.18
z6	253.95	14.17	57.19	2643.40	1879.3	414.91
Importance:	5	1	3	2	1	3
Range of value:	606.45	5.15	0.48	47.61	579.55	10.73
Weight (w_k):	0.08	1.94	62.5	0.42	0.02	2.80

Table 7 illustrates a summary of Table 6 capturing each goal with their weight, target value and importance.

Table 7: Summary depicting the goals and their weights

	Goals	Weight (w_k)	Target	Importance
z1	Minimise cost	0.08	0	5
z2	Achieve target alcohol level	1.94	15	1
z3	Achieve certain level of chroma	62.5	59	3
z4	Achieve target residual sugar	0.42	2300	2
z5	Maximise tannins	0.02	2458.85	1
z6	Maximise anthocyanins	2.80	414.91	3

To formulate the MOGP, we can use the original constraints of the MILP and add additional constraints for each goal:

$$
\begin{aligned}
Price: &\quad 269X_1 + 346X_2 + 230X_3 + 269X_4 + 150Y_1 + 100Y_2 + 120Y_3 - d_1^+ \geq 0 \\
Alcohol: &\quad 14.50X_1 + 14.60X_2 + 13.70X_3 + 14.50X_4 + Y_3 - d_2^+ + d_2^- = 15 \\
Chroma: &\quad 58.06X_1 + 60.29X_2 + 55.69X_3 + 58.38X_4 + 15.00Y_1 - d_3^+ + d_3^- = 59 \\
Sugar: &\quad 2800X_1 + 2200X_2 + 2500X_3 + 2700X_4 - d_4^+ + d_4^- = 2300 \\
Tannins: &\quad 1704.12X_1 + 1866.05X_2 + 1790.84X_3 + 2254.66X_4 + 100Y_2 + d_5^- \leq 2458.85 \\
Anthocyanins: &\quad 486.04X_1 + 707.80X_2 + 393.03X_3 + 342.71X_4 + d_6^- \leq 414.91 \\
Deviants: &\quad d_k^+, d_k^- > 0 \quad \forall k \\
Continuous: &\quad X_1 + X_2 + X_3 + X_4 > 0 \quad \forall i
\end{aligned}
$$

Using the constraints mentioned above, we can now specify the Archimedean and the preemptive goal programming approaches.

3.4.1 Archimedean Goal Progamming

$$\text{Minimise:} \quad \Sigma_{k=1}^{7} w_k \times d_k$$

Subject to all constraints mentioned above

3.4.2 Preemptive Goal Programming

Since we have 4 levels of importance. The Preemptive goal programming is done in 4 steps. Each step will have one more constraint created by previous step. All the other constraints mentioned above are included as well.

$$
\begin{aligned}
\text{Step 1: Minimise:} &\quad \alpha = w_1 \times d_1 \\
\text{Step 2: Minimise:} &\quad \beta = w_3 \times d_3 + w_6 \times d_6 \\
w_1 \times d_1 &\leq 20.23 \\
\text{Step 3: Minimise:} &\quad \lambda = w_4 \times d_4 \\
w_1 \times d_1 &\leq 20.23 \\
w_3 \times d_3 + w_6 \times d_6 &\leq 127.20 \\
\text{Step 4: Minimise:} &\quad w_2 \times d_2 + w_5 \times d_5 \\
w_1 \times d_1 &\leq 20.23 \\
w_3 \times d_3 + w_6 \times d_6 &\leq 127.20 \\
w_4 \times d_4 &\leq 147.03
\end{aligned}
$$

3.4.3 Result

Table 8 illustrates the solution from the two different approaches. The two solutions are almost identical, where they differ is in the proportion of CR and CF by 1 percent.

Table 8: Solutions of the Archimedean and Preemptive goal programming approaches

	Archimedean	Preemptive
X_1	0.33	0.33
X_2	0.01	0.00
X_3	0.41	0.41
X_4	0.25	0.26
Y_1	0	0
Y_2	0	0
Y_3	0	0
Deviations:		
z1	253.95^+	252.87^+
z2	0.83^-	0.83^-
z3	1.81^-	1.84^-
z4	343.40^+	350.06^+
z5	579.55^-	576.42^-
z6	0	4.42^-

The Archimedean gave an optimized objective of 290.72. The Preemptive approach gave an optimized objective of 410.9(20.23+127.20+147.03+116.44). Both gave similar solutions. Compare to Archimedean approach, Preemptive approach focus achieve goal with higher priority. Goals with less priority have larger deviations than Archimedean approach.

4 Discussion

The optimal blend of wine that satisfies the requirements is given by **CS:CR:ME:CF** $= 0.33 : 0 : 0.41 : 0.26$ with no food additives. Majority of the algorithms have lead to this result, or something close.

SA with a Geometric cooling schedule and with a Logarithmic cooling schedule had different levels of desire to explore the solution space over times. In our case, both cooling schedules lead to the same result, but the Logarithmic cooling schedule explored the solution space more and it seems to be more likely to find a global optimal.

In GA, it was found that rank selection did not work well with the problem. Otherwise, an appropriate solution was found. However, this solution was sub-optimal in comparison to all other methods used in this study.

To solve the multi-objective Goal problem, we implemented the Archimedean and Preemeptive approaches. Archimedean achieved a better optimized weighted objective value but on the other hand, Preemeptive had a lower cost which is our most prioritised goal. Preemeptive focused most on the prioritised goal and subsequently compromised on the other goals. Archimedean gives a more balanced solution. Users should choose them carefully by what is most important to them.

Overall, it would seem that a MILP is sufficient to study this level of wine assemblage. SA and MOGP both gave similar results and MOGP was more robust in optimizing many goals. GA is a poor chose in the modelling of wine assemblage as it was incapable of finding the global optima.

References

Cáceres-Mella, A., Peña-Neira, A., Avilés-Gálvez, P., Medel-Marabolí, M., Del Barrio-Galán, R., López-Solís, R., and Canals, J. M. (2013). Phenolic composition and mouthfeel characteristics resulting from blending chilean red wines. *J Sci Food Agric*, 94(4):666–676.

Ferrier, J. G. and Block, D. E. (2001). Neural-network-assisted optimization of wine blending based on sensory analysis. *American Journal of Enology and Viticulture*, 52(4):386–395.

Li, S.-Y., Zhao, P.-R., Ling, M.-Q., Qi, M.-Y., García-Estévez, I., Escribano-Bailón, M. T., Chen, X.-J., Shi, Y., and Duan, C.-Q. (2020). Blending strategies for wine color modification i: Color improvement by blending wines of different phenolic profiles testified under extreme oxygen exposures. *Food Research International*, 130:108885.

Meillon, S., Thomas, A., Havermans, R., Pénicaud, L., and Brondel, L. (2012). Sensory-specific satiety for a food is unaffected by the ad libitum intake of other foods during a meal. is SSS subject to dishabituation? *Appetite*, 63:112–118.

R Core Team (2023). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Theussl, S. and Hornik, K. (2023). *Rglpk: R/GNU Linear Programming Kit Interface*. R package version 0.6-5.

Vismara, P., Coletta, R., and Trombettoni, G. (2016). Constrained global optimization for wine blending. *Constraints*, 21(4):597–615.

Wang, H., Miao, Y., Xu, X., Ye, P., Wu, H., Wang, B., and Shi, X. (2022). Effects of blending on phenolic, colour, antioxidant and aroma components of cabernet sauvignon wine from xinjiang (china). *Foods*, 11(21).

A Code

(R Core Team, 2023)

```
---
title: "Optimisation Assignment"
author: "Edward Baleni, BLNEDW003, Wayne Jiang, JNGWEN002"
header-includes:
   - \usepackage{amsmath}
date: "`r Sys.Date()`"
output:
  pdf_document:
    fig_caption: yes
    extra_dependencies:
    - float
    - subfig
    keep_md: yes
  html_document:
    df_print: paged
always_allow_html: yes
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE, fig.align="center", out.width = "65%", fig.pos =
    "H")
```
```

```
```

```{r Packages}
require(Rglpk)
require(foreach)
```

Data

```{r Data}
X <- read.table("Data3.txt")
```

Linear Program
```{r Linear Formulation}
#LP
# Objective is to minimise total cost
my_obj1 <- X$Price

# Constraints
my_mat = rbind(X$pH,X$pH,X$Abrasiveness,
               X$Hardness,X$Dryness,X$Dryness,
               X$Bitterness,X$H,X$H,
               c(1,1,1,1,0,0,0),
               c(0,0,0,0,1,1,1))

# Bounds
my_dir=c(">","<",">","<",">","<",">",">","<","==","<=")

# RHS
my_rhs=c(3.52,3.55,8,10,35,38,7.7,17,18,1,5)

# Data types
my_types<-c("C","C","C","C","I","I","I")

# Run MILP
LP=Rglpk_solve_LP(obj=my_obj1,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=F
    )
```

Simulated Annealing

```{r Simulated Annealing}
#Simulated Annealing
#set seed to make result reproducible
set.seed(1)
#create a matrix to store all solution by different setting of Simulated Annealing
solu_SA=matrix(0,nrow=7,ncol=5)

#evaluation function
evaluate_x<-function(x){
  # Objective
  my_obj <- X$Price
  # Calculate objective
  eva=x%*%my_obj

  # If constraints are not met then give a large number (since we're minimizing)
  if(x%*%X$pH<3.52|x%*%X$pH>3.55|x%*%X$Abrasiveness<8
     |x%*%X$Hardness>10|x%*%X$Dryness<35|x%*%X$Dryness>38
     |x%*%X$Bitterness<7.7|x%*%X$H<17|x%*%X$H>18|sum(x[5:7])>5)
    {
    eva=1000
```

```
    }
  # Return output
  return(eva)
}

#function to give a feasible initial solution in order to find optimal
get_initial_x<-function(){
  # Get a sequence between 0 and 1
  x=seq(0,1,length.out=100)
  # initialise current solution
  cur_x=c(0.25,0.25,0.25,0.25,0,0,0)
  found=F

  # First find a feasible solution that meets all the constraints, might not be the
     optimal solution
  while(found==F)
  {
    sam=sample(1:4,4,replace = F)
    for(i in 1:100)
    {
      for(j in 1:100)
      {
        for(k in 1:100)
        {
          cur_x[sam[1]]=x[i]
          cur_x[sam[2]]=x[j]
          cur_x[sam[3]]=x[k]
          cur_x[sam[4]]=1-cur_x[sam[1]]-cur_x[sam[2]]-cur_x[sam[3]]
          if(cur_x%*%X$pH>3.52&cur_x%*%X$pH<3.55&cur_x%*%X$Abrasiveness>8
             &cur_x%*%X$Hardness<10&cur_x%*%X$Dryness>35&cur_x%*%X$Dryness<38
             &cur_x%*%X$Bitterness>7.7&cur_x%*%X$H>17&cur_x%*%X$H<18
             &cur_x[sam[4]]>=0)
          {
            cur_xt=cur_x
            found=T
          }
        }
      }
    }
  }
  return(cur_xt)
}

# function to change some value of the solution vector
# either wine proportion or number of food additive will be change
perturb_x <- function(cur_x){
  # Choose a number between 1 and 2
  sam=sample(1:2,1,replace = F)
  # If 1 is sampled then change the continuous variables
  if(sam==1){
    sam=sample(1:4,2,replace = F)
    value=runif(1,-min(cur_x[sam[1]],cur_x[sam[2]])/2,min(cur_x[sam[1]],cur_x[sam
    [2]])/2)
    cur_x[sam[1]]=cur_x[sam[1]]+value
    cur_x[sam[2]]=cur_x[sam[2]]-value
  }
  else{
    # if 2 is sampled then change the integer variables
    sam=sample(5:7,1,replace = F)
    cur_x[sam]=sample(c(max(cur_x[sam]-1,0),cur_x[sam]+1),1,replace = F)
  }
  return(cur_x)
```

```
}

#Geometric with temperature factor=0.995
start_temp <- 1
temp_factor <- 0.995
all_fx=c()
all_x=c()

# Get initial solutioin
initx=get_initial_x()
cur_x=initx
cur_fx=evaluate_x(cur_x)

# Perform SA
for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp * temp_factor ^ i
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)
  # accept or reject the candidate
  if(prop_fx < cur_fx){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }
  else{ if(runif(1) < accept_prob){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }}

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- c(all_x,cur_x)
}

all_fx_G1=all_fx
solu_SA[,1]=all_x[((which(all_fx==min(all_fx))[1]-1)*(7)+1):(which(all_fx==min(all_
    fx))[1]*(7))]

# Repeat for different temperature
#Geometric with temperature factor =0.95
start_temp <- 0.1
temp_factor <- 0.95
all_fx=c()
all_x=c()
cur_x=initx
cur_fx=evaluate_x(cur_x)

for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp * temp_factor ^ i
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)
  # accept or reject the candidate
  if(prop_fx < cur_fx){
    cur_x <- prop_x
    cur_fx <- prop_fx
```

```
  }
  else{ if(runif(1) < accept_prob){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }}

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- c(all_x,cur_x)
}

all_fx_G5=all_fx
solu_SA[,2]=all_x[((which(all_fx==min(all_fx))[1]-1)*(7)+1):(which(all_fx==min(all_
    fx))[1]*(7))]

# Repeat for different cooling schedule
#Logarithmic with starting temp =1 temp_factor=0.995
start_temp <- 1
temp_factor <- 0.995
all_fx=c()
all_x=c()
cur_x=initx
cur_fx=evaluate_x(cur_x)

for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp /(1+temp_factor*log(1+i))
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)
  # accept or reject the candidate
  if(prop_fx < cur_fx){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }
  else{ if(runif(1) < accept_prob){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }}

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- c(all_x,cur_x)
}

all_fx_L1=all_fx
solu_SA[,3]=all_x[((which(all_fx==min(all_fx))[1]-1)*(7)+1):(which(all_fx==min(all_
    fx))[1]*(7))]

# Repeat for different starting temp
#Logarithmic with starting temp =0.5,temp_factor=0.995
start_temp <- 0.5
temp_factor <- 0.995
all_fx=c()
all_x=c()
cur_x=initx
cur_fx=evaluate_x(cur_x)

for(i in 1:10000){
  # generate a candidate solution
```

```
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp /(1+temp_factor*log(1+i))
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)
  # accept or reject the candidate
  if(prop_fx < cur_fx){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }
  else{ if(runif(1) < accept_prob){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }}

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- c(all_x,cur_x)
}
all_fx_L5=all_fx
solu_SA[,4]=all_x[((which(all_fx==min(all_fx))[1]-1)*(7)+1):(which(all_fx==min(all_
    fx))[1]*(7))]

# Repeat for starting temp factor
#Logarithmic with starting temp =1 tf=0.8
start_temp <- 1
temp_factor <- 0.8
all_fx=c()
all_x=c()
cur_x=initx
cur_fx=evaluate_x(cur_x)

for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp /(1+temp_factor*log(1+i))
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)
  # accept or reject the candidate
  if(prop_fx < cur_fx){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }
  else{ if(runif(1) < accept_prob){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }}

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- c(all_x,cur_x)
}

all_fx_L18=all_fx
solu_SA[,5]=all_x[((which(all_fx==min(all_fx))[1]-1)*(7)+1):(which(all_fx==min(all_
    fx))[1]*(7))]

# Plot final solution
plot(all_fx_G1,type="l", ylab = "f(x)",lwd=2)
```

```
points(all_fx_G5,type="l",col="green",lwd=2)
points(all_fx_L1,type="l",col="red",lwd=2)
points(all_fx_L5,type="l",col="blue",lwd=2)
points(all_fx_L18,type="l",col="brown",lwd=2)
legend("topright", legend=c("Geometric:   T0 = 1;    tf = 0.995","Geometric:   T0 = 1;    tf = 0.95",
                            "Logarithmic: T0 = 1;    tf = 0.995","Logarithmic: T0 = 0.5; tf = 0.995",
                            "Logarithmic: T0 = 1;    tf = 0.800"),
       col=c("black","green","red", "blue","brown"), lty=1, cex=0.8,lwd=2)
```

Genetic Algorithm

```{r Set Seed}
set.seed(6)
```

```{r GA Initialize}
normal <- function(pop){
  # normalize to make sure that the proportions add up to 1
  if(!is.null(dim(pop)))
    pop/rowSums(pop)
  else
    pop/sum(pop)
}

init.pop <- function(n, p, X){
  # Obtain continuous
  Y <- normal(matrix(sample(1:100, n*(p-3), replace = T)/100, nrow = n, ncol = p-3))

    # Generate integer solutions if there are any
  test <- F
  hold <- matrix(NA, nrow = n, ncol = 3)
  for (i in 1:n) {
    while (test==F) {
      x <- sample(0:5, prob = c(50,10, 1, 0.5, 0.1 , 0.05), size = 3, replace = T)
      if(sum(x) <=5 )
        test <- T
      else
        test <- F
    }
    hold[i,] <- x
    test <- F
  }

  Y <- cbind(Y, hold)

  test <- FALSE
  count <-  1

  # Obtain a population that works for the solution
  #for (i in 1:n) {
  foreach(i=1:n) %do% {
    x <- Y[i,]
    while (test == F){
        # Calculate constraints
        ph      <- x %*% X$pH
        abras   <- x %*% X$Abrasiveness
```

```
        hard     <- x %*% X$Hardness
        dry      <- x %*% X$Dryness
        bitter   <- x %*% X$Bitterness
        hue      <- x %*% X$H

        # Check if solution is in feasible region
        test <- ifelse(ph < 3.52 | ph > 3.55, F,
          ifelse(abras < 8, F,
                 ifelse(hard > 10, F,
                        ifelse(dry < 35 | dry > 38, F,
                               ifelse(bitter < 7.7,  F,
                                      ifelse(hue < 17 | hue > 18, F,
                                             T))))))

        # Maybe it's the integers that are messing everything up
        if(test==F){
          x[5:7] <- sample(0:5, prob = c(50,10, 1, 0.5, 0.1 , 0.05), size = 3,
    replace = T)
          while (test == F & sum(x[5:7]) > 5) {
            x[5:7] <- sample(0:5, prob = c(50,10, 1, 0.5, 0.1 , 0.05), size = 3,
    replace = T)
          }
        }

        # Or maybe it's the continuous variables
        if (test==F){
          x <- c(normal(sample(1:100, p-3, replace = T)/100), x[5:7])
        }
        else{
          Y[i,] <- x
        }
    }
    test <- F
  }

  return(Y)
}
```


```{r GA Evaluate}
eval.pop <- function(pop, X){
  # Evaluate fitness
  n <- nrow(pop)

  # Initialize fitness
  fitness <- rep(0, n)
  # Calculate fitness
  fitness <- pop %*% X$Price
  ph <- pop %*% X$pH
  abras <- pop %*% X$Abrasiveness
  hard <- pop %*% X$Hardness
  dry <- pop %*% X$Dryness
  bitter <- pop %*% X$Bitterness
  hue <- pop %*% X$H

   # Handle constraints
   fitness <- ifelse(ph < 3.52 | ph > 3.55, 400,
          ifelse(abras < 8, 400,
                 ifelse(hard > 10, 400,
                        ifelse(dry < 35 | dry > 38, 400,
                               ifelse(bitter < 7.7,  400,
```

```
                                    ifelse(hue < 17 | hue > 18, 400,
                                           fitness))))))

  return(fitness)
}
```


```{r GA Select}
# Rank based selection
select.rank <- function(pop, fit){
  n <- nrow(pop)
  p <- ncol(pop)
  # Rank population based on fitness
  new.pop <- matrix(NA, nrow = n, ncol = p)
  # Obtain order of fitness
  ord <- order(fit, decreasing = T)
  # Arrange according to order
  fit2 <- fit
  fit2 <- fit2[ord]
  fit2 <- cbind(fit2, 1:n )
  # Place back in regular order
  fit2[ord,] <- fit2[1:n,]

  # Sample based on rank
  rank.samp <- sample(1:n, prob = fit2[,2], replace = T)

  # Input these samples into the new population
  new.pop <- pop[rank.samp,]

  return(new.pop)
}

# Tournament Selection
select.tournament <- function(pop, fit, s.size){
  n <- nrow(pop)
  p <- ncol(pop)
  hold <- list()
  # Tournament based on fitness
  new.pop <- matrix(NA, nrow = n, ncol = p)

  # Tournament
  for (i in 1:n) {
    sub.samp <- sample(1:n, s.size, replace = T)
    hold[[i]] <- sub.samp[which.min(fit[sub.samp])]
  }
  pop[unlist(hold),]
}
```


```{r GA Crossover}
# Uniform crossover
uni.cross <-  function(parents){
  # Georgina (2023)
  n <- nrow(parents)
  p <- ncol(parents)

  # Pick parents to mate
  parent_pairs <- matrix(sample(1:n), n/2, 2)

  # Initialise offspring
```

```
  offsprings <- matrix(NA, n, p)
  for(i in 1:n/2){
    # Get parents
    p1 <- parents[parent_pairs[i,1], ]
    p2 <- parents[parent_pairs[i,2], ]
    # Make kids
    c1 <- rep(NA, p)
    c2 <- rep(NA, p)
    # Apply uniform crossover to get kids
    for(j in 1:p){
      if(runif(1) <= 0.5){
        c1[j] <- p1[j]
        c2[j] <- p2[j]
      }else{
        c2[j] <- p1[j]
        c1[j] <- p2[j]
      }
    }
    # Store kids
    offsprings[2*i-1, ] <- c1
    offsprings[2*i, ] <- c2
  }
  return(offsprings)
}

# N-point crossover
n.cross <- function(parents){
    # N-point Crossover
  n <- nrow(parents)
  p <- ncol(parents)

  # Pick parents to mate
  pairs <- matrix(sample(1:n), n/2, 2)

  # Initialise offspring
  offsprings <- matrix(NA, n, p)

  # Perfom N-point
  for (i in 1:round(n/2)) {
    # Get parents
    p1 <- parents[pairs[i,1], ]
    p2 <- parents[pairs[i,2], ]

    # Make kids
    c1 <- rep(NA, p)
    c2 <- rep(NA, p)

    # Pick cross-point as 3
    c1 <- c(p1[1], p2[2], p1[3], p2[4], p1[5], p2[6], p1[7])
    c2 <- c(p2[1], p1[2], p2[3], p1[4], p2[5], p1[6], p2[7])


    # Store kids
    offsprings[2*i-1, ] <- c1
    offsprings[2*i, ] <- c2
  }
  return(offsprings)
}
```


```{r GA Mutate}
```

```r
scram.mut = function(cross, mutation_rate, check){
  # Georgina (2023)
  # Scramble mutation
  n <- nrow(cross)
  p <- ncol(cross)

  cross2 <- cross
  if (check){
    # Normalize if the proportions do not add up to between 0.9 and 1
    cross2 <- ifelse(matrix(rep(rowSums(cross), 4), n, p) < 0.9 |
                     matrix(rep(rowSums(cross), 4), n, p) > 1,
                     normal(cross),
                     cross)
  }

  # Initialise mutations
  mutations = matrix(NA, n,p)

  for(i in 1:n){
    persontomutate = cross2[i,]
    if(runif(1) <= mutation_rate){
      # Select two elements
      picks = sort(sample(1:p, 2, replace = FALSE))
      # Get sub-set
      temp = persontomutate[picks[1]:picks[2]]
      # Reshuffle
      temp = sample(temp, length(temp), replace = FALSE)
      # Add mutation
      persontomutate[picks[1]:picks[2]] = temp
      mutations[i,] = persontomutate
    }else{
      mutations[i,] = persontomutate
    }
  }
  return(mutations)
}

mut.func <- function(p){
  # Insert mutation function

  # Select two elements
  picks = sort(sample(1:p, 2, replace = FALSE))

  # Move second to first
  ord <- c(which(1:p <= picks[1]), picks[2], which(1:p > picks[1] & 1:p != picks
    [2]))

  # Return order
  return(ord)
}


insert.mut <- function(cross, mut.rate, check){
  # Insert mutation
  p <- ncol(cross)
  n <- nrow(cross)

  cross2 <- cross
  # Normalize if the proportions do not add up to between 0.9 and 1
  if (check){
    cross2 <- ifelse(matrix(rep(rowSums(cross), 4), n, p) < 0.9 |
                      matrix(rep(rowSums(cross), 4), n, p) > 1,
```

```
                    normal(cross),
                    cross)
  }
  # Initialise mutations
  mutations = matrix(NA, n, p)

  # Perform insert mutation
  rate <- replicate(n, runif(1))

  for (i in 1:n) {
    if(rate[i] <= mut.rate){
      mutations[i,] <- cross2[i,mut.func(p)]
    }
    else
      mutations[i,] <- cross2[i,]
  }
  return(mutations)
}
```

```{r GA}
# Initialise matrix
pop.in1 <- pop.in2 <- pop.in3 <- pop.in4 <- pop.in5 <- pop.in6 <- pop.in7 <- pop.in8 <- init.pop(100,7, X)

# Initialise list to store fittest and mean fitness
fittest <- list()
fit_mean <- list()

# Number of generations to run GA
gen <- 100

# Perform GA
for (i in 1:gen) {
  # evaluate function
  evals1 <- eval.pop(pop.in1, X)
  evals2 <- eval.pop(pop.in2, X)
  evals3 <- eval.pop(pop.in3, X)
  evals4 <- eval.pop(pop.in4, X)
  evals5 <- eval.pop(pop.in5, X)
  evals6 <- eval.pop(pop.in6, X)
  evals7 <- eval.pop(pop.in7, X)
  evals8 <- eval.pop(pop.in8, X)

  # select by rank
  nxt.parent.rank1 <-  select.rank(pop.in1, evals1)
  nxt.parent.rank2 <-  select.rank(pop.in2, evals2)
  nxt.parent.rank3 <-  select.rank(pop.in3, evals3)
  nxt.parent.rank4 <-  select.rank(pop.in4, evals4)
  # select by tournament
  nxt.parent.tourn5 <- select.tournament(pop.in5, evals5, 5)
  nxt.parent.tourn6 <- select.tournament(pop.in6, evals6, 5)
  nxt.parent.tourn7 <- select.tournament(pop.in7, evals7, 5)
  nxt.parent.tourn8 <- select.tournament(pop.in8, evals8, 5)

  # Rank - Uni - Scram
    # cross by uni      # can do by rank or tournament
  offspring.cross.uni.rank <-  uni.cross(nxt.parent.rank1)
    # By Mutation by scramble
  offspring.mut.scram.uni.rank1 <- scram.mut(offspring.cross.uni.rank[,1:4], 0.05, T)
```

```
687   offspring.mut.scram.uni.rank2 <- scram.mut(offspring.cross.uni.rank[,5:7], 0.05,
        F)
688   offspring.mut.scram.uni.rank <- cbind(offspring.mut.scram.uni.rank1, offspring.
        mut.scram.uni.rank2)
689     # Replace
690   pop.in1 <- offspring.mut.scram.uni.rank
691
692   # Rank - Uni - Insert
693     # cross by uni      # can do by rank or tournament
694   offspring.cross.uni.rank <-  uni.cross(nxt.parent.rank2)
695     # By Mutation by insert
696   offspring.mut.insert.uni.rank1 <- insert.mut(offspring.cross.uni.rank[,1:4],
        0.05, T)
697   offspring.mut.insert.uni.rank2 <- insert.mut(offspring.cross.uni.rank[,5:7],
        0.05, F)
698   offspring.mut.insert.uni.rank <- cbind(offspring.mut.insert.uni.rank1, offspring.
        mut.insert.uni.rank2)
699     # Replace
700   pop.in2 <- offspring.mut.insert.uni.rank
701
702   # Rank - N - Insert
703     # Cross by N
704   offspring.cross.n.rank <-  n.cross(nxt.parent.rank3)
705     # Mutation by insert
706   offspring.mut.insert.n.rank1 <- insert.mut(offspring.cross.n.rank[,1:4], 0.05, T)
707   offspring.mut.insert.n.rank2 <- insert.mut(offspring.cross.n.rank[,5:7], 0.05, F)
708   offspring.mut.insert.n.rank <- cbind(offspring.mut.insert.n.rank1, offspring.mut.
        insert.n.rank2)
709     # Replace
710   pop.in3 <- offspring.mut.insert.n.rank
711
712   # Rank - N - Scramble
713       # Cross by N
714   offspring.cross.n.rank <-  n.cross(nxt.parent.rank4)
715     # Mutation by insert
716   offspring.mut.scram.n.rank1 <- scram.mut(offspring.cross.n.rank[,1:4], 0.05, T)
717   offspring.mut.scram.n.rank2 <- scram.mut(offspring.cross.n.rank[,5:7], 0.05, F)
718   offspring.mut.scram.n.rank <- cbind(offspring.mut.scram.n.rank1, offspring.mut.
        scram.n.rank2)
719     # Replace
720   pop.in4 <- offspring.mut.scram.n.rank
721
722   # Tourn - N - Insert
723     # Cross by N
724   offspring.cross.n.tourn <- n.cross(nxt.parent.tourn5)
725     # Mutation by insert
726   offspring.mut.insert.n.tourn1 <- insert.mut(offspring.cross.n.tourn[,1:4], 0.05,
        T)
727   offspring.mut.insert.n.tourn2 <- insert.mut(offspring.cross.n.tourn[,5:7], 0.05,
        F)
728   offspring.mut.insert.n.tourn <- cbind(offspring.mut.insert.n.tourn1, offspring.
        mut.insert.n.tourn2)
729     # Replace
730   pop.in5 <- offspring.mut.insert.n.tourn
731
732   # Tourn - N - Scram
733     # Cross by N
734   offspring.cross.n.tourn <- n.cross(nxt.parent.tourn6)
735     # Mutation by Scram
736   offspring.mut.scram.n.tourn1 <- scram.mut(offspring.cross.n.tourn[,1:4], 0.05, T)
737   offspring.mut.scram.n.tourn2 <- scram.mut(offspring.cross.n.tourn[,5:7], 0.05, F)
738   offspring.mut.scram.n.tourn <- cbind(offspring.mut.scram.n.tourn1, offspring.mut.
```

```
    scram.n.tourn2)
    # Replace
  pop.in6 <- offspring.mut.scram.n.tourn

  # Tourn - Uni - Insert
    # Cross by Uni
  offspring.cross.uni.tourn <- uni.cross(nxt.parent.tourn7)
    # Mutation by Insert
  offspring.mut.insert.uni.tourn1 <- insert.mut(offspring.cross.uni.tourn[,1:4],
    0.05, T)
  offspring.mut.insert.uni.tourn2 <- insert.mut(offspring.cross.uni.tourn[,5:7],
    0.05, F)
  offspring.mut.insert.uni.tourn <- cbind(offspring.mut.insert.uni.tourn1,
    offspring.mut.insert.uni.tourn2)
    # Replace
  pop.in7 <- offspring.mut.insert.uni.tourn

  # Tourn - Uni - Scram
    # Cross by uni
  offspring.cross.uni.tourn <- uni.cross(nxt.parent.tourn8)
    # Mutation by Scram
  offspring.mut.scram.uni.tourn1 <- scram.mut(offspring.cross.uni.tourn[,1:4],
    0.05, T)
  offspring.mut.scram.uni.tourn2 <- scram.mut(offspring.cross.uni.tourn[,5:7],
    0.05, F)
  offspring.mut.scram.uni.tourn <- cbind(offspring.mut.scram.uni.tourn1, offspring.
    mut.scram.uni.tourn2)
    # Replace
  pop.in8 <- offspring.mut.scram.uni.tourn

  # Store
  fittest[[i]] <- cbind(min(evals1), min(evals2), min(evals3), min(evals4), min(
    evals5), min(evals6), min(evals7),min(evals8))
  fit_mean[[i]] <- cbind(mean(evals1), mean(evals2), mean(evals3), mean(evals4),
    mean(evals5), mean(evals6), mean(evals7), mean(evals8))
}
```

```{r GA Results}
# Plot convergence
fit <- matrix(unlist(fit_mean), nrow = gen, ncol = 8, byrow = T)
matplot(fit, col = c("purple", "blue", "pink", "grey", "darkgreen", "orange", "red"
    , "black"), lwd = "2", type = "l", lty = 1, xlab = "Iteration", ylab = "Mean
    Fitness")
legend("right", c("Genetic Algroithm 1", "Genetic Algorithm 2","Genetic Algroithm 3
    ", "Genetic Algorithm 4","Genetic Algroithm 5", "Genetic Algorithm 6","Genetic
    Algroithm 7", "Genetic Algorithm 8"),col = c("purple", "blue", "pink", "grey",
    "darkgreen", "orange", "red", "black"), lty=1, cex = 0.7, lwd = 2)
```

Multi-Objective Goal Programming

```{r MOGP}
#MOGP
#max each goal to find range of value
my_obj1 <- X$Price
my_obj2 <- X$Alcohol
my_obj3 <- X$C
my_obj4 <- X$Sugar
my_obj5 <- X$Tannins
```

```
my_obj6 <- X$Anthocyanins

s1=Rglpk_solve_LP(obj=my_obj1,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)
s2=Rglpk_solve_LP(obj=my_obj2,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)
s3=Rglpk_solve_LP(obj=my_obj3,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)
s4=Rglpk_solve_LP(obj=my_obj4,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)
s5=Rglpk_solve_LP(obj=my_obj5,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)
s6=Rglpk_solve_LP(obj=my_obj6,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)

ss=rbind(s1$solution,s2$solution)
ss=rbind(ss,s3$solution)
ss=rbind(ss,s4$solution)
ss=rbind(ss,s5$solution)
ss=rbind(ss,s6$solution)
#the matrix can be use calculate range of value for each variables
mmm=ss%*%cbind(X$Price,X$Alcohol,X$C,X$Sugar,X$Tannins,X$Anthocyanins)

#use calculated weight to perform MOGP by Rglpk function
#Archimedean approach
my_obj1 <- c(rep(0,7),0.08,1.94,1.94,62.5,62.5,0.42,0.42,0.02,0.02,2.80,2.80)

my_mat = matrix(c(c(X$pH,rep(0,11)),
                  c(X$pH,rep(0,11)),
                  c(X$Abrasiveness,rep(0,11)),
                  c(X$Hardness,rep(0,11)),
                  c(X$Dryness,rep(0,11)),
                  c(X$Dryness,rep(0,11)),
                  c(X$Bitterness,rep(0,11)),
                  c(X$H,rep(0,11)),
                  c(X$H,rep(0,11)),
                  c(1,1,1,1,0,0,0,rep(0,11)),
                  c(0,0,0,0,1,1,1,rep(0,11)),
                  c(X$Price,-1,rep(0,10)),
                  c(X$Alcohol,0,-1,1,rep(0,8)),
                  c(X$C,rep(0,3),-1,1,rep(0,6)),
                  c(X$Sugar,rep(0,5),-1,1,rep(0,4)),
                  c(X$Tannins,rep(0,7),-1,1,rep(0,2)),
                  c(X$Anthocyanins,rep(0,9),-1,1)),ncol=18,byrow=T)

my_dir=c(">","<",">","<",">","<",">",">","<","==","<=","<=","==","==","==","==")
my_rhs=c(3.52,3.55,8,10,35,38,7.7,17,18,1,5,0,15,59,2300,2458.85,414.91)


my_types<-c("C","C","C","C","I","I","I","C","C","C","C","C","C","C","C","C","C")

ss1=Rglpk_solve_LP(obj=my_obj1,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=F)
round(ss1$solution,2)
#Preemptive approach
#first step
my_obj2 <- c(rep(0,7),0.08,rep(0,10))
my_mat2 = matrix(c(c(X$pH,rep(0,11)),
                   c(X$pH,rep(0,11)),
                   c(X$Abrasiveness,rep(0,11)),
```

```
                    c(X$Hardness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Bitterness,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(1,1,1,1,0,0,0,rep(0,11)),
                    c(0,0,0,0,1,1,1,rep(0,11)),
                    c(X$Price,-1,rep(0,10)),
                    c(X$Alcohol,0,-1,1,rep(0,8)),
                    c(X$C,rep(0,3),-1,1,rep(0,6)),
                    c(X$Sugar,rep(0,5),-1,1,rep(0,4)),
                    c(X$Tannins,rep(0,7),-1,1,rep(0,2)),
                    c(X$Anthocyanins,rep(0,9),-1,1)),ncol=18,byrow=T)

ss2=Rglpk_solve_LP(obj=my_obj2,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=
    F)

#second step
#use optimal value from last step to constrain
my_obj3 <- c(rep(0,10),62.5,62.5,0,0,0,0,2.80,2.80)
my_mat3 = matrix(c(c(X$pH,rep(0,11)),
                    c(X$pH,rep(0,11)),
                    c(X$Abrasiveness,rep(0,11)),
                    c(X$Hardness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Bitterness,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(1,1,1,1,0,0,0,rep(0,11)),
                    c(0,0,0,0,1,1,1,rep(0,11)),
                    c(X$Price,-1,rep(0,10)),
                    c(X$Alcohol,0,-1,1,rep(0,8)),
                    c(X$C,rep(0,3),-1,1,rep(0,6)),
                    c(X$Sugar,rep(0,5),-1,1,rep(0,4)),
                    c(X$Tannins,rep(0,7),-1,1,rep(0,2)),
                    c(X$Anthocyanins,rep(0,9),-1,1),
                    c(rep(0,7),0.08,rep(0,10))),ncol=18,byrow=T)
my_dir3=c(">","<",">","<",">","<",">",">","<","==","<=","<=","==","==","==","==","
    ==","<=")
my_rhs3=c(3.52,3.55,8,10,35,38,7.7,17,18,1,5,0,15,59,2300,2458.85,414.91,ss2$
    optimum)
ss3=Rglpk_solve_LP(obj=my_obj3,mat=my_mat3,dir=my_dir3,rhs=my_rhs3,types=my_types,
    max=F)

#third step
#use optimal value from previous step to constrain
my_obj4 <- c(rep(0,12),0.42,0.42,rep(0,4))
my_mat4 = matrix(c(c(X$pH,rep(0,11)),
                    c(X$pH,rep(0,11)),
                    c(X$Abrasiveness,rep(0,11)),
                    c(X$Hardness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Bitterness,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(1,1,1,1,0,0,0,rep(0,11)),
                    c(0,0,0,0,1,1,1,rep(0,11)),
                    c(X$Price,-1,rep(0,10)),
                    c(X$Alcohol,0,-1,1,rep(0,8)),
```

```
                    c(X$C,rep(0,3),-1,1,rep(0,6)),
                    c(X$Sugar,rep(0,5),-1,1,rep(0,4)),
                    c(X$Tannins,rep(0,7),-1,1,rep(0,2)),
                    c(X$Anthocyanins,rep(0,9),-1,1),
                    c(rep(0,7),0.08,rep(0,10)),
                    c(rep(0,10),62.5,62.5,0,0,0,0,2.80,2.80)),ncol=18,byrow=T)
my_dir4=c(">","<",">","<",">","<",">",">","<","==","<=","<=","==","==","==","==","<=","<=")
my_rhs4=c(3.52,3.55,8,10,35,38,7.7,17,18,1,5,0,15,59,2300,2458.85,414.91,ss2$optimum,ss3$optimum)
ss4=Rglpk_solve_LP(obj=my_obj4,mat=my_mat4,dir=my_dir4,rhs=my_rhs4,types=my_types,max=F)

#last step
#use optimal value from previous step to constrain
my_obj5 <- c(rep(0,8),1.94,1.94,62.5,62.5,rep(0,6))
my_mat5 = matrix(c(c(X$pH,rep(0,11)),
                    c(X$pH,rep(0,11)),
                    c(X$Abrasiveness,rep(0,11)),
                    c(X$Hardness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Bitterness,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(1,1,1,1,0,0,0,rep(0,11)),
                    c(0,0,0,0,1,1,1,rep(0,11)),
                    c(X$Price,-1,rep(0,10)),
                    c(X$Alcohol,0,-1,1,rep(0,8)),
                    c(X$C,rep(0,3),-1,1,rep(0,6)),
                    c(X$Sugar,rep(0,5),-1,1,rep(0,4)),
                    c(X$Tannins,rep(0,7),-1,1,rep(0,2)),
                    c(X$Anthocyanins,rep(0,9),-1,1),
                    c(rep(0,7),0.08,rep(0,10)),
                    c(rep(0,10),62.5,62.5,0,0,0,0,2.80,2.80),
                    c(rep(0,12),0.42,0.42,rep(0,4))),ncol=18,byrow=T)
my_dir5=c(">","<",">","<",">","<",">",">","<","==","<=","<=","==","==","==","==","<=","<=","<=")
my_rhs5=c(3.52,3.55,8,10,35,38,7.7,17,18,1,5,0,15,59,2300,2458.85,414.91,ss2$optimum,ss3$optimum,ss4$optimum)
ss5=Rglpk_solve_LP(obj=my_obj5,mat=my_mat5,dir=my_dir5,rhs=my_rhs5,types=my_types,max=F)
round(ss1$solution,2)
```