CPSC 433 - F25

Project Proposal

Team Turtwig

John Cummings	30219471
Victor Nguyen	30171290
Stephanie Sevilla	30176781
Jacob Situ	30144935
Kaylee Xiao	30173778
Raza Zaidi	30075408

Genetic Algorithm Set-Based Search

1. Model Definition

The set-based search model is defined as

$$A_{set} = (S_{set}, T_{set})$$

where each state in S_{set} represents a population of complete schedules, and transitions in T_{set} correspond to genetic algorithm (GA) extension rules (mutation, crossover, and purge).

A *fact* in this model is a **complete schedule $\boldsymbol{f \in F}$**. We represent a schedule by its assignment function (as defined in the project problem):

$$assign_f : Lectures + Tutorials \to Slots$$

which, in the project problem, must satisfy

$$Constr(assign_f) = true \text{ and } Eval(assign_f) \text{ is minimized.}$$

In a state, we will handle facts as a finite sequence of pairs, each pair consisting of a $class \in Tutorials \cup Lectures$ and a corresponding set of slots $(s_{i_1}, ..., s_{i_m}) \subset Slots$ where there is a pairing of corresponding time slots to each item $class$. Each schedule may or may not satisfy the hard constraints defined in the project specifications (e.g., pre-assignments, linked dats, AL availability).

Formally, we define the set of all facts as

$$\begin{aligned} F = \{f \mid f = \{&(class_1, (s_{1_1}, ..., s_{1_m})), ..., (class_n, (s_{n_1}, ..., s_{n_m}))\}, \\ &\forall\, class \in Tutorials \cup Lectures, (s_{i_1}, ..., s_{i_m}) \subset Slots, \\ &|f| = |Tutorials \cup Lectures| \\ \}, \end{aligned}$$

and the **state space** is defined as

$$S_{set} \subseteq 2^F,$$

so that each state $\boldsymbol{s \in S_{set}}$ represents a population of complete schedules currently being evaluated in the search.

Each individual fact $f \in F$ (a complete schedule) carries the following attributes:

- $Valid(assign_f)$: the total penalty based on the hard constraints
- $Eval(assign_f)$: the total penalty value based on soft constraints

To modify our set of facts F we will use our set of extension rules formally defined as

$$Ext = \{A \to B \mid A, B \subseteq F \text{ and } (Mutate(A, B) \text{ or } Crossover(A, B) \text{ or } Purge(A, B))\}$$

and the **transitions** are defined as

$$T_{set} = \{(s, s') | \exists A \to B \in Ext \text{ with } A \subseteq s \text{ and } s' = (s - A) \cup B\} \subseteq S_{set} \times S_{set}.$$

Mutate(A, B)

$$A = \{f\}$$
$$B = \{f, f'\}$$

where for a single $class_i \in$ Lectures $\cup$ Tutorials in complete schedule f, we will change the slot set $(s_{i_1}, ..., s_{i_m})$ that it is paired, to a different slot set $(s_{i_1}, ..., s_{i_m})'$, such that $(s_{i_1}, ..., s_{i_m}) \neq (s_{i_1}, ..., s_{i_m})'$ producing new complete schedule f' for which we add to our current set of complete schedules.

Mutate sub-extensions. When *Mutate* is performed we select one of the four sub-extensions:

***Mutate*$_{\text{evening}}$:** Where a random $class_i$ that has prefix DIV 9 is selected and we sample $(s_{i_1}, ..., s_{i_m})'$ from evening slot sets

***Mutate*$_{\text{AL}}$:** Where a random $class_i$ that requires AL is selected and we sample $(s_{i_1}, ..., s_{i_m})'$ from AL-capable slot sets

***Mutate*$_{\text{lecture}}$:** Where a random $class_i \in$ Lectures is selected and we sample $(s_{i_1}, ..., s_{i_m})'$ from lecture slot sets

***Mutate*$_{\text{tutorial}}$:** Where a random $class_i \in$ Tutorials is selected and we sample $(s_{i_1}, ..., s_{i_m})$ from tutorial slot sets

For each sub-extension, we choose $(s_{i_1}, ..., s_{i_m})'$ uniformly at random within the sample of slot sets given specific sub-extension, we apply the slot set change to produce f' and add f' to the population.

Crossover(A, B)

$$A = \{f_a, f_b\}$$
$$B = \{f_a, f_b, f_c\}$$

where for each $class_i$ in complete schedule f_c we will select the slot set $(s_{i_1}, ..., s_{i_m})$ for which it is paired by flipping a coin and matching the $(class_i, (s_{i_1}, ..., s_{i_m}))$ pairing from either f_a or f_b.

Purge(A, B)

$$A = \{f_1, ..., f_p\}$$
$$B = \{f_1, ..., f_{p-k}\}$$

where once we hit a maximum population size in our total set of facts, namely size k, we will remove p complete schedules from our set of facts, where the complete schedules to be removed are based on their score from the fitness function $Fitness$, with the lowest p scores being removed from our population of facts.

Search Instance

The set-based search instance is defined as

$$Ins_{set} = (s_0, G_{set})$$

$$s_0, s_{goal} \in 2^F$$

$$G_{set} : S \to \{yes, no\}$$

$G_{set}(s_i) = yes$ if and only if $s_{goal} \subseteq s_i$ or there is no extension rule applicable in s_i

Initial State: The search begins with a population of k complete schedules. Formally, the initial state $s_0 \in 2^F$ is defined as

$$s_0 = \{f_1, f_2, \ldots, f_k\}$$

where each complete schedule f_j is generated by pairing all $class_i \in$ Tutorials $\cup$ Lectures to a slot set $(s_{i_1}, ..., s_{i_m})$. While f_j is randomly generated, the random generation respects the slot groupings defined in the environment. Before instantiating the population, all abstract time slots in the input are pre-processed and grouped into linked sets. For instance, Monday/Wednesday/Friday slots at the same hour are treated as one lecture block, and Tuesday/Thursday slots as another. These slot groups are sorted in the environment since they remain constant throughout the search, preserving the linkage constraint by construction.

Goal Function: The goal function is defined as

$$G_{set}(s_i) = \begin{cases} \text{yes}, & \text{if } (\exists f \in s_i : Fitness(f) = 1) \vee (w \geq W) \vee (C \geq X) \\ \text{no}, & \text{otherwise} \end{cases}$$

Where:

- $Fitness(f_i)$ is the fitness score of schedule f_i (is defined later on).
- w is the number of consecutive iterations without improvement in the best fitness score of our set of facts.
- W is the maximum tolerated number of iterations without improvement (plateau threshold).
- C is a counter of total complete schedules produced.
- X is a predefined limit on the number of complete schedules to be produced.

2. Precise Description and Search Control

Search Process

The set-based search process is defined as

$$P_{set} = (A_{set}, Env, K_{set})$$

where A_{set} is the set-based search model previously defined, Env is the static environment that stores all fixed external and problem specific information required by A_{set}, and K_{set} corresponds to the search control direction.

Environment

Formally, we define the Environment(Env) as a tuple that contains all static data and helper functions required by the search model such that

$$\begin{aligned} Env = (&O,\ ST,\ G,\ SlotInfo,\ PreAssign,\ Unwanted,\ NotCompatible,\ ActiveLearning, Evening,\\ &TutorialMax,\ ALlectureMax,\ ALtutorialMax,\\ &LectureMin,\ TutorialMin,\ Preference,\\ &pen_{hard},\ pen_{lecturemin},\ pen_{tutorialmin},\ pen_{notpaired},\ pen_{section},\\ &PassEvening(x),\ PassAL(x),\ PassLectures(x),\ PassTutorials(x),\\ &Valid(x),\ Constr(x),\ Eval(x),\ Fitness(x),\\ &w_h,\ w_s,\ k,\ W,\ X,\ RNG) \end{aligned}$$

where

- **O**: The finite set of all offerings where $O = Lectures \cup Tutorials$. Each element represents either a lecture or tutorial section that must be assigned a slot group.
- **ST**: The finite set of all available abstract slots that can be assigned to offerings.
- **G**: A partition of ST into linked slot groups (e.g., Tu/Th or M/W/F). Each group $g \in G$ represents a set of slots that are linked together and must be assigned jointly.
- **SlotInfo** : $ST \rightarrow$ SlotMeta: A mapping that stores metadata for each slot s, including its day(s), time, slot type (lecture or tutorial), evening flag, active learning (AL) flag, and group identifier. This structure is implemented as a hashmap where each slot ID is the key, and the value is the corresponding metadata record.
  - **Formal definition:** For each $s \in ST$,

$$\textbf{SlotInfo}(s) = \Big(d_s,\ t_s,\ type_s,\ evening_s,\ AL_s,\ g_s\Big),$$

    where d_s is the day set, $t_s = (\text{start}_s, \text{end}_s)$ is the time interval with explicit start and end times, $type_s \in \{\text{lec}, \text{tut}\}$, $evening_s \in \{\text{T}, \text{F}\}$, $AL_s \in \{\text{T}, \text{F}\}$, and $g_s \in G$ is the group identifier.

- **PreAssign** $\subseteq O \times ST$: A relation containing all fixed (offering, slot) pairs that must be respected throughout the search.
- **Unwanted** $\subseteq O \times ST$: A relation of disallowed (offering, slot) pairs that must not appear in any valid schedule.
- **NotCompatible** $\subseteq O \times O$: A relation identifying offerings that cannot share the same slot or overlapping linked block.
- **ActiveLearning** $\subseteq O$: The subset of offerings that require AL-capable slots. Used by $\texttt{PassAL}(x)$ to verify that each $o \in$ ActiveLearning is assigned only to slots with $\textbf{SlotInfo}(s).AL =$ True, and to check per-slot AL capacities via **ALlectureMax** and **ALtutorialMax**.
- **Evening** $\subseteq O$: The subset of offerings that must be scheduled in evening slots (e.g., all DIV 9 lectures in the instance data). Used by $\texttt{PassEvening}(x)$ to verify that each $o \in$ Evening is assigned only to slots with $SlotInfo(s).evening =$ True.
- **TutorialMax** : $ST \to \mathbb{N}$: The maximum number of tutorials permitted in each slot.
- **ALlectureMax**, **ALtutorialMax** : $ST \to \mathbb{N}$: The maximum number of active-learning lectures or tutorials that may occupy each slot, respectively.
- **LectureMin**, **TutorialMin** : $ST \to \mathbb{N}$: The minimum number of lectures or tutorials desired per slot, respectively, for soft-constraint evaluation.
- **Preference** : $O \times ST \to \mathbb{R}_{\geq 0}$: A cost or penalty value representing how undesirable it is to assign offering o to slot s. These values are used in the soft evaluation function.
- $\textbf{pen}_{\textbf{hard}}, \textbf{pen}_{\textbf{lecturemin}}, \textbf{pen}_{\textbf{tutorialmin}}, \textbf{pen}_{\textbf{notpaired}}, \textbf{pen}_{\textbf{section}}$: Positive penalty constants corresponding to the hard and soft constraint categories defined in the model.
- **PassEvening**(x): A helper function that verifies whether a schedule satisfies all evening-related hard constraints (e.g., all DIV 9 lectures are placed in evening slots).
- **PassAL**(x): A helper function that verifies whether all active-learning offerings are placed in AL-capable slots and whether AL capacities are not exceeded.
- **PassLectures**(x): A helper function that checks lecture-related hard constraints, such as lecture slot capacity limits, overlap, and pre-assignments.
- **PassTutorials**(x): A helper function that checks tutorial-related hard constraints, including tutorial slot capacities and linkage rules.
- **Valid**(x): A function that sums all hard-constraint checks and returns a total hard-constraint penalty (zero if all constraints are satisfied).
- **Constr**(x): A Boolean function that evaluates to `True` if and only if all hard constraints are satisfied for the given assignment.
- **Eval**(x): A function that quantifies how well an assignment satisfies all soft constraints by summing penalties for minimum usage violations, section overlaps, unpaired lectures/tutorials, and preference scores.

- **Fitness**(x): A function that maps each complete schedule to a fitness score using weighted hard and soft penalties. A higher fitness indicates a better schedule.
- $\mathbf{w_h}, \mathbf{w_s} \in \mathbb{R}_{>0}$: Weighting constants applied to the hard and soft penalty components in the fitness function.
- $\mathbf{k} \in \mathbb{N}$: The maximum population size maintained in any state during the search.
- $\mathbf{W} \in \mathbb{N}$: The plateau threshold, i.e., the maximum number of iterations allowed without improvement in the best fitness value.
- $\mathbf{X} \in \mathbb{N}$: The maximum number of complete schedules that may be produced during the search.
- **RNG**: A random number generator used by the search process for stochastic operations such as parent selection and mutation choices.

Search Control

The search control, K_{set}, will rely on both f_{wert} and f_{select}, both are defined as

$$f_{wert}(Mutate, Crossover, Purge) = \begin{cases} 0 & \text{if } |Pop(A)| \geq k \wedge \text{Purge(A, B)} \\ 1 & \text{if } |Pop(A)| < k \wedge (\text{Mutate(A, B)} \vee \text{Crossover(A, B)}) \end{cases}$$

If $A \to B$ exists that fulfills Purge(A, B) i.e. the maximum threshold of k has been reached, then $f_{wert}(A, B, e) = 0$, where $e \in Env$. We always choose Purge(A, B) to elimiate the p lowest-valued individuals. In the other case, $A \to B$ exists that fulfills either Mutate(A, B) or Crossover(A, B), then $f_{wert}(A, B, e) = 1$, where we do either Mutate(A, B) or Crossover(A, B) using f_{select} to choose.

$$f_{select}(Mutate, Crossover) = \begin{cases} Mutate(A, B) & \text{with a probability of 0.5} \\ Crossover(A, B) & \text{with a probability of 0.5} \end{cases}$$

f_{select} will use a RNG as the tiebreaker to select either Mutate(A, B) or Crossover(A, B).

Once Mutate(A, B) or Crossover(A, B) is chosen, each $f \in F$ will then be assigned a fitness value using the function $Fitness(assign_f)$ where a value closer to 1 is preferred. Weights will be initialized for both the valid and eval values to give more precendence to hard constraint violations. For example, we can initialize w_h as 10 and w_s as 0.7.

$$Fitness(assign_f) = \frac{1}{1 + w_h \cdot Valid(assign_f) + w_s \cdot Eval(assign_f)}$$

A roulette wheel selection process will be used to pick either one parent/individual for Mutate, or two parents/individuals for Crossover. The fitness value of each each indvidual $f \in F$ corresponds with a probability to be selected. The probability can be calculated with

$$Probability(fi) = \frac{Fitness(assign_{fi})}{\sum_{j=1}^{|F|} Fitness(assign_{fj})}$$

For each $f \in F$, we will copy each f into a list, L_f, where it is sorted on increasing probability values. We will then generate a random number $r \in [0, 1]$, and compute a cumulative sum of the probabilities for each individual in L_f. If $r < RunningSum(f_1)$, then return the first individual. Otherwise, continue calculating the running sum of each individual until the nth individual satisfies $RunningSum(f_{n-1}) < r \leq RunningSum(f_n)$. Once this sum is greater than r, then select the current individual, f_n, as the parent. The RunningSum function can be defined as

$$RunningSum(fi) = \sum_{j=1}^{i} Probability(f_j)$$

If f_{select} selects Mutate(A, B), there will be 4 sub-variants of Mutate that can be selected to affect either tutorial, lecture, evening, or active learning slots for a more refined search control. The selected parent f, from the roulette wheel selection, will go through four helper functions in order: PassEvening(f), PassAL(f), PassLectures(f), PassTutorials(f). PassEvening has the highest precendence with PassTutorials having the lowest precedence. These functions test hard constraints concerning lectures/tutorials/labs that need evening slots, AL slots, or general lecture/tutorial slot issues. The first type of hard constraint the selected parent fails will be the type of mutation it undergoes.

$$f_{select}(Mutate(A, B)) = \begin{cases} Mutate_{evening}(A, B) & \text{if PassEvening(A) = False} \\ Mutate_{AL}(A, B) & \text{if PassAL(A) = False} \\ Mutate_{lecture}(A, B) & \text{if PassLectures(A) = False} \\ Mutate_{tutorial}(A, B) & \text{if PassTutorials(A) = False} \end{cases}$$

The hard constraints function $Valid(assign_f)$ quantifies how well a schedule satisfies the hard constraints of the problem, where $Valid(assign_f) = 0$ for a valid schedule. If a hard constraint is not satisfied for an assignment, a penalty value, pen_{hard} will be returned. We define $Valid(assign_f)$ as:

$$Valid(assign_f) = \begin{cases} 0 & \forall c \in C, c(assign_f) = True \\ \text{pen}_{hard} & \text{Otherwise} \end{cases}$$

$\text{Valid}(\text{assign}_f)$ is a summation of all of the hard constraints $c \in C$, where $C = \{c_1, ..., c_{16}\}$ and $Constr(assign_f) = PassEvening \wedge PassAL \wedge PassLectures \wedge PassTutorials = True$ if all of the following hard constraints are satisfied for an assignment. The hard constraints can be organized into four categories identified with an additional subscript: evening, AL, lectures, and tutorials, such that $C_{evening} \subseteq C$, $C_{AL} \subseteq C$, $C_{lectures} \subseteq C$, $C_{tutorials} \subseteq C$.

- $c_{lectures_1}$: no more than lecturemax(s) lectures can be assigned to $s \in Slots$
  - $\forall s \in Slots$: lecturemax(s) $\geq$ assignL(s), where assignL(s) is the total number of lectures assigned to s
- $c_{tutorials_2}$: no more than tutorialmax(s) tutorials can be assigned to $s \in Slots$
  - $\forall s \in Slots$: tutorialmax(s) $\geq$ assignT(s), where assignT(s) is the total number of tutorials assigned to s
- c_{AL_3} : no more than ALlecturemax(s) lectures that need activelearning can be assigned to $s \in Slots$
  - $\forall s \in Slots$: ALlecturemax(s) $\geq$ assignALL(s), where assignALL(s) is the total number of lectures that need activelearning assigned to s
- c_{AL_4} : no more than ALtutorialmax(s) tutorials that need activelearning can be assigned to $s \in Slots$
  - $\forall s \in Slots$: ALtutorialmax(s) $\geq$ assignALT(s), where assignALT(s) is the total number of tutorials that need activelearning assigned to s
- $c_{lectures_5}$: $\forall l, t \in Lectures \cup Tutorials : assign(l_i) \neq assign(t_{iki}), \forall i, k_i$
- $c_{lectures_6}$: notcompatible(a,b) with $a, b \in Lectures \cup Tutorials$, there should be no overlap for a and b such that $assign(a) \neq assign(b)$
  - $\forall a, b \in notcompatible \subseteq Lectures \cup Tutorials : assign(a) \neq assign(b)$
- $c_{lectures_7}$: $\forall a \in Lectures \cup Tutorials : assign(a) = partassign(a) \wedge partassign(a) \neq \{\$\}$
- $c_{lectures_8}$: unwanted(a,s), then $assign(a) \neq s$ should be true for $s \in Slots$
  - $\forall (a, s) \in unwanted : assign(a) \neq s$
- c_{AL_9} : assign(a) should be so that ALlecturemax(a) > 0 or ALtutorialmax(a) > 0 as necessary, for a list of activelearning(a) statements
  - $\forall a \in activelearning : (a \in Lectures \rightarrow ALlecturemax(a) > 0) \vee (a \in Tutorials \rightarrow ALtutorialmax(a) > 0)$
- $c_{lectures_{10}}$: lectures should occupy either corresponding blocks Mon/Wed/Fri or Tue/Thu (treated as one abstract general slots)
  - $\forall l \in Lectures : Block(assign(l)) \in \{MonWedFri, TueThur\}$ where Block is a function to map slots to their predefined grouping (lectures are mapped to MWF or TTH blocks)
- $c_{tutorials_{11}}$: tutorials/labs should occupy either corresponding blocks Mon/Wed, Tue/Thu, or Fri

  - $\forall t \in Tutorials : Block(assign(t)) \in \{MonWed, TueThur, Fri\}$, where Block is a function to map slots to their predefined grouping (tutorials are mapped to MW, TTH or F)

- $c_{evening_{12}}$: lectures with prefix "DIV 9" should go into evening slots (18:00 or later)
  - $\forall l \in Lectures : prefix(l) == \text{"DIV 9"} \rightarrow assign(l).start\_time \geq 18$, where prefix is a function to parse out the prefix from each lecture in Lectures

- $c_{lectures_{13}}$: lectures in all tiers with course number 5XX level should be scheduled into non-overlapping time slots
  - $\forall a, b \in 5XX \subseteq Lectures \cup Tutorials, a \neq b : assign(a) \neq assign(b)$

- $c_{lectures_{14}}$: no lectures are scheduled on Tuesday 11:00-12:30
  - $\forall l \in Lectures : assign(l) \neq s_t$, where s_t = (Tuesday, 11, 12.5)

- $c_{evening_{15}}$: CPSC 851 scheduled at Tue/Thu 18:00-19:00, with no overlap with CPSC 351 lectures/tutorials/labs
  - $Block(assign(\text{CPSC 851})) \in \{TuesThur\} \wedge assign(\text{CPSC 851}).start\_time == 18 \wedge assign(\text{CPSC 851}).end\_time == 19$, where assign(s).start_time and assign(s).end_time returns the start time and end time for the slot
  - $\forall sections \in \text{CPSC 351} : assign(sections) \neq assign(\text{CPSC 851})$

- $c_{evening_{16}}$: CPSC 913 scheduled at Tue/Thu 18:00-19:00, with no overlap with CPSC 413 lectures/tutorials/labs
  - $Block(assign(\text{CPSC 913})) \in \{TuesThur\} \wedge assign(\text{CPSC 913}).start\_time == 18 \wedge assign(\text{CPSC 913}).end\_time == 19$, where assign(s).start_time and assign(s).end_time returns the start time and end time for the slot
  - $\forall sections \in \text{CPSC 413} : assign(sections) \neq assign(\text{CPSC 913})$

The helper functions to determine if an individual satisfies various sub-categories of hard constraints can be defined as

$$PassEvening(assign_f) = \begin{cases} True & \forall c \in C_{evening}, c(assign_f) = True \\ False & \text{Otherwise} \end{cases}$$

$$PassAL(assign_f) = \begin{cases} True & \forall c \in C_{AL}, c(assign_f) = True \\ False & \text{Otherwise} \end{cases}$$

$$PassLectures(assign_f) = \begin{cases} True & \forall c \in C_{lectures}, c(assign_f) = True \\ False & \text{Otherwise} \end{cases}$$

$$PassTutorials(assign_f) = \begin{cases} True & \forall c \in C_{tutorials}, c(assign_f) = True \\ False & \text{Otherwise} \end{cases}$$

The evaluation function $Eval(assign_f)$ quantifies how well a schedule satisfies the soft constraints of the problem. We define this as:

$$\begin{aligned}
\text{Eval}(\text{assign}_f) = & \sum_{s \in \text{Slots}} \Big(\text{pen}_{\text{lecturemin}} \cdot \max(0,\, \text{lecturemin}(s) - \text{assignL}(s)) \Big) \\
& + \sum_{s \in \text{Slots}} \Big(\text{pen}_{\text{tutorialmin}} \cdot \max(0,\, \text{tutorialmin}(s) - \text{assignT}(s)) \Big) \\
& + \sum_{(a,b) \in \text{pair}} \Big(\text{pen}_{\text{notpaired}} \cdot [\text{assign}(a) \neq \text{assign}(b)] \Big) \\
& + \sum_{\substack{l_i, l_j \in \text{Lectures} \\ i<j}} \Big(\text{pen}_{\text{section}} \cdot [\text{section}(l_i) = \text{section}(l_j) \wedge \text{assign}(l_i) = \text{assign}(l_j)] \Big) \\
& + \sum_{a \in (\text{Lectures} \cup \text{Tutorials})} \Big(\text{preference}(a, \text{assign}(a)) \Big)
\end{aligned}$$

Eval is a function that measures how well an assignment fulfills the soft constraints by summing up the following penalties: $pen_{\text{tutorialmin}}, pen_{\text{lecturemin}}, pen_{\text{ection}}, pen_{\text{notpaired}}$, and $preference$. Each pen and $preference$ are postive penalty weights. The goal is for Eval(assign), for any given assign, to be minimized as a value equal to 0 satisfies all soft constraints.

- $pen_{\text{lecturemin}}$: Penalty applied for each slot where the number of lectures assigned are below the minimum needed
- $pen_{\text{tutorialmin}}$: Penalty applied for each slot where the number of tutorials assigned are below the minimum needed
- pen_{section}: Penalty applied for each pair of sections scheduled at the same time
- $pen_{\text{notpaired}}$: Penalty applied for each lecture and/or tutorial not scheduled at the same time
- $preference$: Scores of the preference-value of a lecture/tutorial being assigned to a slot is added up for each assignment

3. Demonstration on a Small Instance

To demonstrate how our set-based search process operates, we apply it to a simplified timetabling instance derived from the project problem. This example is deliberately small to make each transition clear while preserving all relevant model components.

Instance Setup

$$\begin{aligned} Lectures &= \{L_1,\ L_2\}, \quad Tutorials = \{T_1\} \\ Grp_{lec} &= \{\, g^{\text{MWF}}_{9:00},\ g^{\text{TTh}}_{9:30} \,\} \quad \text{(lecture groups)} \\ Grp_{tut} &= \{\, g^{\text{MW}}_{10:00},\ g^{\text{TTh}}_{10:00},\ g^{\text{F}}_{10:00} \,\} \quad \text{(tutorial groups)} \\ Grp &= Grp_{lec} \cup Grp_{tut} \end{aligned}$$

Constraints (tiny instance):

- $NotCompatible = \{(L_1, L_2)\}$ (the two lectures cannot occupy overlapping lecture groups).
- $PreAssign = \emptyset$, $Unwanted = \emptyset$, $ActiveLearning = \emptyset$, $Evening = \emptyset$ (kept empty to focus more on the mechanics).
- Per-slot capacities are sufficient (no forced capacity violations in this tiny setup).
- Penalties/evaluation use the model's definitions (pen_{hard} for hard violations; soft via $Preference$, $LectureMin$, $TutorialMin$, etc.).

Control parameters (read from Env):

$$k = 3 \quad \text{(population cap)}, \quad W = 3 \quad \text{(plateau)}, \quad X = 10 \quad \text{(max schedules)},$$
$$w_h > 0, \quad w_s > 0 \quad \text{(weights used by } \texttt{Fitness}\text{)}$$

Initial State

We seed three complete schedules (facts), assigning lectures to Grp_{lec} and the tutorial to Grp_{tut}:

$$\begin{aligned} f_1 &: \ L_1 \to g^{\text{MWF}}_{9:00},\ \ L_2 \to g^{\text{TTh}}_{9:30},\ \ T_1 \to g^{\text{MW}}_{14:00} \\ f_2 &: \ L_1 \to g^{\text{TTh}}_{9:30},\ \ L_2 \to g^{\text{TTh}}_{9:30},\ \ T_1 \to g^{\text{F}}_{14:00} \\ f_3 &: \ L_1 \to g^{\text{MWF}}_{9:00},\ \ L_2 \to g^{\text{MWF}}_{9:00},\ \ T_1 \to g^{\text{TTh}}_{14:00} \end{aligned}$$

Thus,

$$s_0 = \{\, f_1,\ f_2,\ f_3 \,\}.$$

Evaluation with $\texttt{Valid}(x)$ and $\texttt{Eval}(x)$:

- f_1: lectures are split across MWF vs. TTh $\Rightarrow \texttt{Valid}(f_1) = 0$.
- f_2: both lectures on TTh $\Rightarrow \texttt{Valid}(f_2) = pen_{hard} > 0$.
- f_3: both lectures on MWF $\Rightarrow \texttt{Valid}(f_3) = pen_{hard} > 0$.

$\texttt{Fitness}(x)$ is then computed as defined elsewhere (feasible first, then decrease soft).

Transition 1 – Purge (per f_{wert})

Since $|s_0| = 3$ and $k = 3$, we must purge before any variation. Remove the lowest-fitness schedule (largest hard+soft penalty). Assuming f_2 is worst:

$$s_1 \;=\; s_0 \setminus \{f_2\} \;=\; \{\, f_1,\; f_3 \,\}$$

Transition 2 – Variation: Mutation

Now $|s_1| = 2 < k$, so f_{wert} allows a variation to occur. Suppose the control chooses $Mutation$ with probability 0.5.

Once $Mutation$ is selected, f_{select} determines which $Mutation$ sub-extension to apply using the helper functions `PassEvening`(x), `PassAL`(x), `PassLectures`(x), and `PassTutorials`(x). Each of these functions corresponds to one of the four hard-constraint categories:

1. **Evening** : ensuring that DIV9 lectures are scheduled only in evening slots
2. **Active Learning (AL)** : ensuring AL courses are assigned to AL-capable slots and within AL capacities
3. **Lectures** : ensuring lecture slot capacities and linkage constraints are satisfied
4. **Tutorials** : ensuring tutorial slot capacities and linkage constraints are satisfied

These checks are prioritized in descending order: *Evening* $\rightarrow$ *AL* $\rightarrow$ *Lecture* $\rightarrow$ *Tutorial*, so that higher-precedence violations are addressed before lower-precedence ones.

If the parent schedule fails a particular hard-constraint category, the corresponding mutation sub-extension ($Mutate_{evening}, Mutate_{AL}, Mutate_{lecture}, Mutate_{tutorial}$) is chosen to generate a new child that may satisfy the constraint type that the parent violated.

Next, f_{select} is responsible for choosing which parent will undergo mutation. It employs a roulette wheel selection using the helper functions `Fitness`(x), `Probability`(x), `RunningSum`(x), `Valid`(x), and `Eval`(x) to assign each schedule a selection probability.

- Each schedule receives a `Fitness` value based on its hard (`Valid`) and soft (`Eval`) penalties.
- A corresponding `Probability` value is derived from `Fitness`.
- `RunningSum` is computed to form a cumulative distribution where more fit individuals occupy larger segments on the roulette wheel.
- A random number from `RNG` is drawn to select a parent, ensuring individuals with higher fitness have a greater chance of being chosen.

This ensures that selection balances choosing strong individuals while still retaining diversity.

Assume f_{select} chooses f_3 as the parent, and based on the failed hard-constraint category, $MutateLecture$ is selected because f_3 violates the lecture-related $NotCompatible$ rule.

$$
\begin{aligned}
f_3: &\quad L_1 \to g^{\text{MWF}}_{9:00},\ \ L_2 \to g^{\text{MWF}}_{9:00},\ \ T_1 \to g^{\text{TTh}}_{14:00} \\
\Downarrow\ & Mutate_{lecture} \\
f'_3: &\quad L_1 \to g^{\text{MWF}}_{9:00},\ \ L_2 \to g^{\text{TTh}}_{9:30},\ \ T_1 \to g^{\text{TTh}}_{14:00}
\end{aligned}
$$

The child f'_3 now satisfies all lecture related hard constraints:

$$Valid(f'_3) = 0$$

We insert the child into the state:

$$s_2 = s_1 \cup \{f'_3\} = \{\, f_1,\ f_3,\ f'_3 \,\}$$

Transition 3 – Variation: Crossover (then Purge if needed)

Since $|s_2| = 3 = k$, f_{wert} dictates that we purge before adding a new child. We remove the lowest-fitness schedule among $\{f_1, f_3, f'_3\}$. Since f_3 is the only schedule that violates a lecture related hard constraint, assume it is least fit:

$$s_3 = s_2 \setminus \{f_3\} = \{\, f_1,\ f'_3 \,\}.$$

Now $|s_3| < k$, so a variation step is permitted again. Suppose f_{select} chooses the $Crossover$ extension. Two parents are chosen using roulette wheel selection; assume f_1 and f'_3 are selected.

We apply crossover to each offering, flipping a coin for each offering to determine which parent's slot group to inherit:

$$
\begin{aligned}
L_1 &: \text{take from } f_1 \Rightarrow g^{\text{MWF}}_{9:00} \\
L_2 &: \text{take from } f'_3 \Rightarrow g^{\text{TTh}}_{9:30} \\
T_1 &: \text{take from } f_1 \Rightarrow g^{\text{MW}}_{14:00}
\end{aligned}
$$

The child is formed as:

$$f_c: \quad L_1 \to g^{\text{MWF}}_{9:00},\ \ L_2 \to g^{\text{TTh}}_{9:30},\ \ T_1 \to g^{\text{MW}}_{14:00}.$$

After insertion:

$$s_4 = s_3 \cup \{f_c\} = \{\, f_1,\ f'_3,\ f_c \,\}.$$

Termination and Goal

The search alternates between $Mutation$ and $Crossover$, purging when $|s_i| \geq k$, until:

$$Goal_{s_i} = \texttt{yes} \quad \Longleftrightarrow \quad \Big(\exists f \in s_i :\ \texttt{Fitness}(f) = 1\Big)\ \vee\ \Big(\text{plateau} \geq W\Big)\ \vee\ \Big(C \geq X\Big).$$

If all hard constraints are satisfied and soft penalties are minimized, the process terminates and the schedule with the highest `Fitness` is returned.

4. Natural Language Explanation and Rationale

Informal overview of our set-based search genetic algorithm. In this search our state will consist of a set of *complete schedules* as facts, where a *complete schedule* is when all the lectures and tutorials are assigned a time slot; these items in our state may or may not pass the hard and soft constraints of the search problem. We will modify our set of *complete schedules* by deploying our extension rules namely, *Mutation*, where we change the time slot of a single lecture or tutorial of a single *complete schedule* to produce a new *complete schedule*. With the *Mutation* rule, there are sub-extension rules, specifically pertaining to the type of change that we will perform at random given the specified *Mutation* sub-extension. $Mutation_{evening}$ will select a random evening class of a *complete schedule* and change it to different random evening time slot set; $Mutation_{AL}$ will select a random active learning class of a *complete schedule* and change it to different random active learning time slot set; $Mutation_{lecture}$ will select and random lecture of a *complete schedule* and change it to a different lecture slot set; $Mutation_{tutorial}$ will select and random tutorial from a *complete schedule* and change it to a different random tutorial slot set. *Crossover* where we take and combine time slot pairings from two parent *complete schedules* to create an offspring *complete schedule*; and *Purge* where we remove a certain population of *complete schedules* from our state based on their *Fitness* score.

In our starting state, we will parse the inputted information and randomly assign a "group" of time slots to each lecture and tutorial. More specifically, we will identify all of the corresponding sets of time slots. An example being {Monday 11:00-12:00, Wednesday 11:00-12:00, Friday 11:00-12:00} would be a "group" of time slots for lectures and {Monday 11:00-12:00, Wednesday 11:00-12:00} would be a "group" of time slots for tutorials and assign these sets to each lecture and tutorial at random, treating each set as an abstract slot. We will generate a certain amount of *complete schedules* with this process and then begin our search. We will end our search, if we run into a few different cases; namely if we find an optimal *complete schedule* that passes all the hard and soft constraints, or if we exceed a certain threshold of iterations where our "best" schedule has not improved (based on *Fitness* score), or we have reached a predefined limit of generated complete schedules. Once one of these cases have been encountered, our search algorithm will have reached it's goal state with the goal function outputting "yes" and we'd return our "best" *complete schedule*.

The search control direction and process is influenced by both f_{wert} and f_{select} to select the type of transitions to perform on the population and how the parent(s) are selected. f_{wert} is only concerned with the three operations: Mutate, Crossover and Purge. Mutate or Crossover will always be available and selected until the goal state is achieved. However, Purge, when it is viable, will always have a higher precedence and be performed before Mutate or Crossover. f_{select} is responsible for tiebreaking between Mutate and Crossover. If Mutate is selected, f_{select} is then responsible for choosing the Mutate sub-extensions with the help of the functions PassEvening, PassAL, PassLectures, and PassTutorials. The hard constraints are categorized into evening, active learning (AL), lectures, and tutorials, with evening types having the most precedence and tutorial types having the least precedence. These functions determine the type of hard constraint the selected parent does not satisfy, and chooses the appropriate sub-extension to hopefully produce a child that will pass the

specific type of hard constraint. f_{select} will be choosing the parent(s) for these operations. f_{select} will utilize a series of functions, such as Fitness, Probability, RunningSum, Valid, and Eval, to essentially give each individual a chunk on a roulette wheel before randomly selecting an individual (Roulette Wheel Selection). Individuals receive a Fitness value (calculated from Valid/hard constraints and Eval/soft constraints) and its associated Probability value (caculated from Fitness). Using RunningSum, individuals that are more fit will have a bigger chunk on the roulette wheel, or a higher probability of being chosen. The search control process relies on both the hard constraints and the soft constraints to be explicitly formalized and defined to be used in the functions/operations mentioned.

Why we chose this model. We chose a set-based genetic algorithm because it allows us to consider the overall quality of our population/fact set rather than just the quality of individuals. In this model, each state consists of a set of *complete schedules*, enabling the algorithm to search and modify the entire population directly through defined extension rules. This formulation integrates naturally with our formal set-based search framework, where transitions such as *Mutation*, *Crossover*, and *Purge* operate on sets of schedules to produce new sets. Set based search fits our project particularly well because the linked sessions require a group aware slot representation (lectures: MWF or Tu/Th; tutorials: MW, Tu/Th, or F) which lets functions treat these groups as 1 rather than abstract slots, reducing invalid children and accelerates search.

By considering the entire population, we gain explicit control over population diversity and can employ evolutionary principles to move closer to generating an optimal solution. Furthermore, the set-based representation allows the goal function G_{set} to evaluate global properties of the population, such as whether an optimal schedule has been found or whether the population has plateaued in performance. Our fitness treats hard constraints (via `Valid`) as dominant over soft constraints (via `Eval`), so the GA naturally respects these hard constraints while still improving preferences.

Overall, this model provides a clear and formal way to describe genetic search dynamics, allowing us to implement different heuristics into our search control and extension rules to best leverage the natural selection concepts of keeping and allowing the best individuals and their traits to be passed on, whilst allowing for mutation and an element of randomness that may lead to other, more desirable outcomes.