



The Impossible Scheduling Problem

Armaan Johal, Valeria Gonzalez Perez

A Real Life Problem

1 in 36 children have Autism Spectrum Disorder

Companies (like Centria Autism) offer Applied Behavior Analysis (ABA) therapy through licenced BCBA clinicians

However, it is **really difficult to schedule ABA therapy** sessions because of numerous constraints (scheduling, certifications, minimum hour requirements, etc)

As a result, tens of thousands of children do not get care and millions of dollars are lost.

Is there a way to **optimize the matching and scheduling** of clients and clinicians?



Not a new problem - Nurse Scheduling Problem

In 1976, Miller et al proposed an optimization problem called the “Nurse Scheduling Problem” in his paper *Nurse Scheduling Using Mathematical Programming*

The nurse scheduling problem involves the **assignment of shifts and holidays to nurses**. Each nurse has their own wishes and restrictions, as does the hospital. The problem is described as finding a schedule that both **respects the constraints** of the nurses and **fulfills the objectives** of the hospital.



Not a new problem - Nurse Scheduling Problem

This problem may seem easy, but it is actually:

NP-Hard!



Nurse Scheduling Problem - Why is it NP Hard?

A Simple Example - Three People

However, a simple example demonstrates why the task is in fact extremely challenging:

- Imagine that you have 3 people (A, B & C) and 3 jobs (1, 2 & 3).
- Each person can do one job.
- Some people are better at certain types of jobs than others.
- Our task is to work out the best allocation of work to each person.

Six Possible Solutions

What are all the possible ways of assigning these three jobs? One possibility is:

- Person A does Job 1
- Person B does Job 2
- Person C does Job 3

Or, Person C could do Job 1 and Person B could do Job 3, etc. In fact there are 6 options for the allocation of work to people, as shown below.

	Person A	Person B	Person C
Solution 1	Job 1	Job 2	Job 3
Solution 2	Job 1	Job 3	Job 2
Solution 3	Job 2	Job 1	Job 3
Solution 4	Job 2	Job 3	Job 1
Solution 5	Job 3	Job 1	Job 2
Solution 6	Job 3	Job 2	Job 1

So, in theory we have to consider all 6 options and choose the one which best matches people's skills to the type of work. Of course this simple case is very easy and takes no time to solve.

Four or More People

However, imagine there are 4 people and 4 jobs instead. If you work out the number of combinations you will find that there are actually 24 different ways of allocating the work.

The difficulty grows surprisingly quickly with the number of people and jobs:

- **4 jobs** => 24 possible schedules
- **5 jobs** => 120 possible schedules
- **6 jobs** => 720 possible schedules
- **7 jobs** => 5,040 possible schedules
- **8 jobs** => 40,320 possible schedules

For a workforce of 15 people the number of combinations is huge: 1,307,674,368,000 (that's **1.3 trillion**).

For hundreds of employees there are more possible ways of allocating the work to people than there are atoms in the universe. Even a bank of supercomputers would not be able to consider all these options in a reasonable amount of time.



The Impossible Scheduling Problem

The Nurse Scheduling Problem can be generalized.

Any problem that is NP-hard and involved finding an optimal “schedule” can be classified as an **impossible scheduling problem**.

It is impossible (computationally intractable) to find an optimal solution to an Impossible scheduling problem, but there are ways to get close.

We want to explore different approaches to “solving” the Impossible Scheduling Problem.

Our “Impossible Scheduling Problem” use case is that of ABA therapy scheduling.



Why Should you Care?

Our Goal:

To illuminate the steps in solving a **real world optimization problem**. We want to help answer the question “How do we actually use these algorithms?”

Learning Objectives:

- Ability to **identify** which real-world problems are Impossible Scheduling Problems
- Understand **how to formulate** a real-world constrained optimization problem
- Get a high-level understanding of various optimization algorithms
- Be able to determine which algorithms are **best suited** for which problems.
- Learn the **thought-process** for approaching real-world optimization problems.
- Feel more confident when approaching real-world optimization problems in the future.

In what scenarios would scheduling be useful?



Menti.com : 9319 7703

Before Scheduling: Matching



Before Scheduling: Matching

In real life, organizations have hundreds of clinicians and thousands of clients. Before they can schedule ABA therapy sessions , **they must first assign clients to clinicians.**

There are certain constraints for matching:

1. Clinicians must meet a client's requirement profile (specialty match, experience match, certification match)
2. Clients who haven't been receiving therapy recently should be prioritized.
3. Clients who need the most therapy (hours) should be prioritized.
4. Each clinician has a maximum number of hours they can work.
5. A client can only be matched with one clinician, but a clinician can have multiple clients

How do we match clients with clinicians?

Matching algorithms: Bipartite Graphs?, Network Flow?, Hungarian maximum matching?, Hopcroft-Karp Algorithm? Which one do we use?



Before Scheduling: Matching

Actually, there is no need for a fancy matching algorithm. We can just use Greedy Filtering.

Why?

Key Point: Because we **ONLY care that the hard constraints are met**. There are no soft constraints (like preferences). There is nothing to really optimize.

Intuition: For each clinician, we can simply filter for the “eligible” clients and pick all of the clients until the clinician’s total hours are filled.

Matching - Algorithm



1. **Filter Clients:**

- Clients are first filtered based on the time since their last supervision session.
- Only clients who meet the hard criteria (specialty match, experience match, certification match) are considered.

2. **Sort Clients:**

- Clients are sorted primarily by the time since their last supervision session (descending).
- If two clients have the same time since their last supervision, they are sorted by the total supervision time required (descending).

3. **Sort Clinicians:**

- Clinicians are sorted by their available hours in descending order.

4. **Greedy Assignment:**

- For each clinician, attempt to assign clients from the sorted list until the clinician's available time is filled or there are no more suitable clients.

Matching - Algorithm



```
class Clinician:
    def __init__(self, name, specialties, years_of_experience, certifications, available_hours):
        self.name = name
        self.specialties = specialties
        self.years_of_experience = years_of_experience
        self.certifications = certifications
        self.available_hours = available_hours

class Client:
    def __init__(self, name, specialty, required_experience, certification, required_hours, last_supervision_days):
        self.name = name
        self.specialty = specialty
        self.required_experience = required_experience
        self.certification = certification
        self.required_hours = required_hours
        self.last_supervision_days = last_supervision_days

def meets_criteria(clinician, client):
    return (
        client.specialty in clinician.specialties and
        clinician.years_of_experience >= client.required_experience and
        client.certification in clinician.certifications
    )

def prioritize_clients(clients):
    return sorted(clients, key=lambda c: (-c.last_supervision_days, -c.required_hours))

def match_clinicians_to_clients(clinicians, clients):
    # Sort clinicians by available hours in descending order
    sorted_clinicians = sorted(clinicians, key=lambda c: c.available_hours, reverse=True)

    assigned_clients = {clinician.name: [] for clinician in sorted_clinicians}

    for clinician in sorted_clinicians:
        # Filter and prioritize clients for this clinician
        filtered_clients = [client for client in clients if meets_criteria(clinician, client)]
        prioritized_clients = prioritize_clients(filtered_clients)

        remaining_hours = clinician.available_hours

        for client in prioritized_clients:
            if client.required_hours <= remaining_hours:
                assigned_clients[clinician.name].append(client)
                remaining_hours -= client.required_hours
                # Remove the assigned client from the list of all clients
                clients.remove(client)

    return assigned_clients
```

Matching - Running the Algorithm

```
# Example usage w/ dummy data
clinicians = [
    Clinician("Dr. Smith", {"aggression", "non-verbal"}, 10, {"BT", "RBT"}, 40),
    Clinician("Dr. Jones", {"self-injurious", "non-verbal"}, 8, {"RBT"}, 35),
    Clinician("Dr. Brown", {"aggression", "self-injurious"}, 12, {"BT"}, 30),
    Clinician("Dr. White", {"aggression", "non-verbal", "self-injurious"}, 15, {"QBS"}, 50),
    Clinician("Dr. Black", {"non-verbal"}, 5, {"BT", "QBS"}, 20)
]

clients = [
    Client("Client 1", "aggression", 7, "BT", 5, 10),
    Client("Client 2", "non-verbal", 6, "RBT", 10, 15),
    Client("Client 3", "self-injurious", 5, "BT", 8, 5),
    Client("Client 4", "aggression", 9, "QBS", 3, 20),
    Client("Client 5", "non-verbal", 4, "BT", 7, 25),
    Client("Client 6", "self-injurious", 6, "RBT", 12, 30),
    Client("Client 7", "aggression", 8, "BT", 4, 12),
    Client("Client 8", "non-verbal", 5, "QBS", 9, 10),
    Client("Client 9", "self-injurious", 7, "BT", 6, 18),
    Client("Client 10", "aggression", 5, "RBT", 5, 8),
    Client("Client 11", "non-verbal", 6, "BT", 11, 22),
    Client("Client 12", "self-injurious", 8, "QBS", 10, 14),
    Client("Client 13", "aggression", 10, "BT", 4, 5),
    Client("Client 14", "non-verbal", 7, "RBT", 6, 3),
    Client("Client 15", "self-injurious", 9, "BT", 7, 20),
    Client("Client 16", "aggression", 6, "QBS", 8, 17),
    Client("Client 17", "non-verbal", 5, "BT", 9, 13),
    Client("Client 18", "self-injurious", 4, "RBT", 5, 16),
    Client("Client 19", "aggression", 8, "BT", 7, 7),
    Client("Client 20", "non-verbal", 6, "QBS", 4, 12),
    Client("Client 21", "self-injurious", 7, "RBT", 6, 25),
    Client("Client 22", "aggression", 5, "BT", 8, 11),
    Client("Client 23", "non-verbal", 8, "BT", 10, 6),
    Client("Client 24", "self-injurious", 9, "QBS", 5, 4),
    Client("Client 25", "aggression", 6, "RBT", 3, 2),
    Client("Client 26", "non-verbal", 7, "BT", 4, 30),
    Client("Client 27", "self-injurious", 8, "BT", 6, 21),
    Client("Client 28", "aggression", 5, "QBS", 5, 19),
    Client("Client 29", "non-verbal", 9, "RBT", 7, 23),
    Client("Client 30", "self-injurious", 4, "BT", 8, 24)
]

assigned_clients = match_clinicians_to_clients(clinicians, clients)

for clinician_name, clients in assigned_clients.items():
    print(f"{clinician_name} is assigned the following clients:")
    for client in clients:
        print(f" - {client.name}")
```

Dr. White is assigned the following clients:

- Client 4
- Client 28
- Client 16
- Client 12
- Client 20
- Client 8
- Client 24

Dr. Smith is assigned the following clients:

- Client 26
- Client 5
- Client 29
- Client 11
- Client 2

Dr. Jones is assigned the following clients:

- Client 6
- Client 21
- Client 18
- Client 14

Dr. Brown is assigned the following clients:

- Client 30
- Client 27
- Client 15
- Client 9

Dr. Black is assigned the following clients:

- Client 17



The Key Point

Not all problems require a fancy algorithm!

When approaching a real world problem (or sub-problem), formulate the problem first to determine what solution is best. Don't lead with a solution.

Scheduling

Scheduling Problem Formulation



Based on certain constraints, we want to optimally create a calendar for that clinician so that they fulfill all the constraints the best that they can. The four constraints that are relevant for scheduling are:

1. Each client needs X hours of supervision each week
2. Supervision sessions cannot exceed 3 hours each
3. There must be 8 working hours between supervision sessions
4. Each client needs at least 2 supervision sessions per week.

Every time we run this algorithm for a particular clinician, we want to output the optimal schedule for the week based on the clinician's hourly availability and these four constraints.

**Constraint Satisfaction Problem:
a tree search over states and constraints**

CSP - Problem Formulation



1. Variables:

- Each supervision session for each client will be a variable. Variables represent the start times of these sessions within the available time blocks of the clinician.

2. Domains:

- The domain for each variable (supervision session) is the set of available time blocks of the clinician for each day of the week.

3. Constraints:

- Each client must have X hours of supervision per week.
- Each supervision session must be between 1 and 3 hours.
- There must be at least 8 working hours between supervision sessions for the same client.
- Each client must have at least 2 supervision sessions per week.

CSP - Python Libraries



- Constraint Library
 - backtracking, recursive backtracking, minimum conflicts
- Google's `ortools.sat.python` > **CP-SAT: Constraint Programming - Satisfiability**
 - Default SAT Search: Variable State Independent Decaying Sum (VSIDS)
 - *additive bumping and multiplicative decay algorithm*

CSP - Underlying Algorithms



- Arc Consistency: AC-3 (mentioned in class)
- Backtracking (constraint library)
- **VSIDS** (what we are using with Google's CP-SAT library)
 - “Search heuristics for making decisions - select an integer variable, a value, and a direction for branching” - Laurent Perron (Google Ops Researcher)
 - Keep in mind: variable phase with the **highest activity** is the one chosen for branching

What is VSIDS?

1. Decision heuristic that maintains a floating point for each variable - floating point is called 'activity'
2. Initially, activity is set to 0
3. When there is a conflict **or** a conflict resolution, activity of the variables involved are bumped to 1
 - a. **Involvement** of a variable in conflict analysis → “conflict scores” bumped
4. Variable activities are **decayed** periodically (multiplied by a decaying or discount factor)
 - a. Constant factor **a** - where $0 < a < 1$
 - b. Encourages to explore *new* states of the search space and to not focus *too* much on the past

Moskewicz et al.

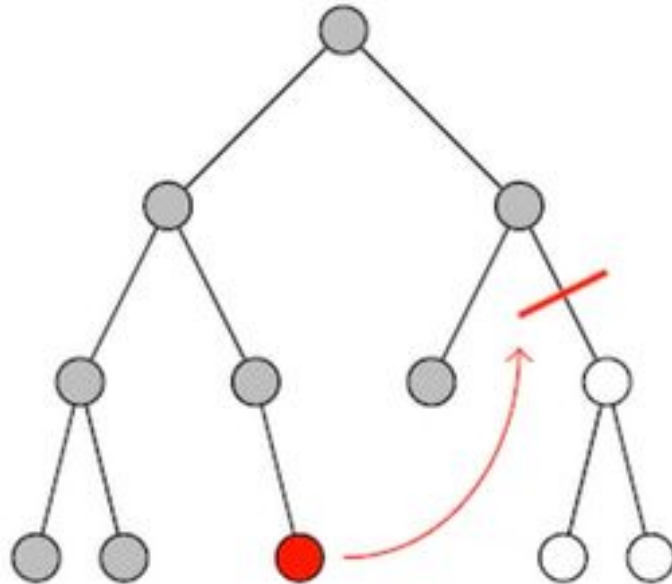
- (1) Each variable in each polarity has a counter, initialized to 0.
- (2) When a clause is added to the database, the counter associated with each literal in the clause is incremented.
- (3) The (unassigned) variable and polarity with the highest counter is chosen at each decision.
- (4) Ties are broken randomly by default, although this is configurable
- (5) *Periodically, all the counters are divided by a constant.*

What is VSIDS? Main Idea



- **Main idea:** to keep track of “learnt clauses” or variables involved in a conflict to guide the search
 - Clauses encode the constraints of a problem in a format that can be handled by the solver
 - Violation of a conflict = violation of at least one clause
- Learning *new* clauses is crucial for SAT solvers to more efficiently prune the search space and find solutions (they help restrict or guide the search)
 - “Keeps statistics for how much a variable causes local infeasibility”

VSIDS



- 1. Variable 2 with **highest activity** is chosen for expansion
 - a. v_2 becomes a decision point
- 2. Once v_2 is assigned with a value of either true or false, it becomes fixed (decision is made)
- 3. the tree is reconstructed based on the fixed value of v_2 (value has been propagated through constraints)
- 4. v_2 is not reconsidered for further assignments and algorithm focuses on more current variables

VSIDS

Variables = $\{v_1, v_2, v_3, v_4, v_5, v_6\}$

activity(v_1) = 2

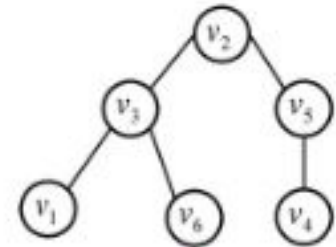
activity(v_2) = 100

activity(v_3) = 19

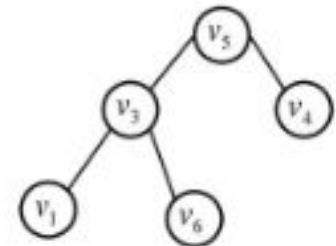
activity(v_4) = 25

activity(v_5) = 36

activity(v_6) = 3



Pick v_2 , reconstruct



Pros and Cons of VSIDS



Pros

- Effectively focusing on conflict involved variables - leading to faster conflict resolution
- Easy to implement (no need for extensive parameter tuning)
- Wide range applicability

Cons

- Short term focus > can get stuck in local optima
- More heavily weighing *recent* conflicts > Lack of global awareness
- Finding the optimal decay rate α
- “Highest variable activity” - Greedy search

A limitation of the short term focus of VSIDS is that:



- a. It can get stuck in local optima - leading to convergence on a potentially suboptimal solution
- b. lacks global awareness - leading to convergence on a potentially suboptimal solution
- c. It does not find the optimal decay rate
- d. Both a and b

A limitation of the short term focus of VSIDS is that:



- a. It can get stuck in local optima - leading to convergence on a potentially suboptimal solution
- b. lacks global awareness - leading to convergence on a potentially suboptimal solution
- c. It does not find the optimal decay rate
- d. Both a and b

CSP for UCSD Rec Scheduling?

“Our proprietary AutoFill algorithm balances a number of competing **constraints** like max hours per day/week, max shift per day, days per week, as well as time between shifts, work time preferences, position preferences, approved Time Off, number & type of positions allowed to work, and any additional priorities you choose like seniority or priority grouping” - WhenToWork



Integer Linear Programming

ILP - Problem Formulation

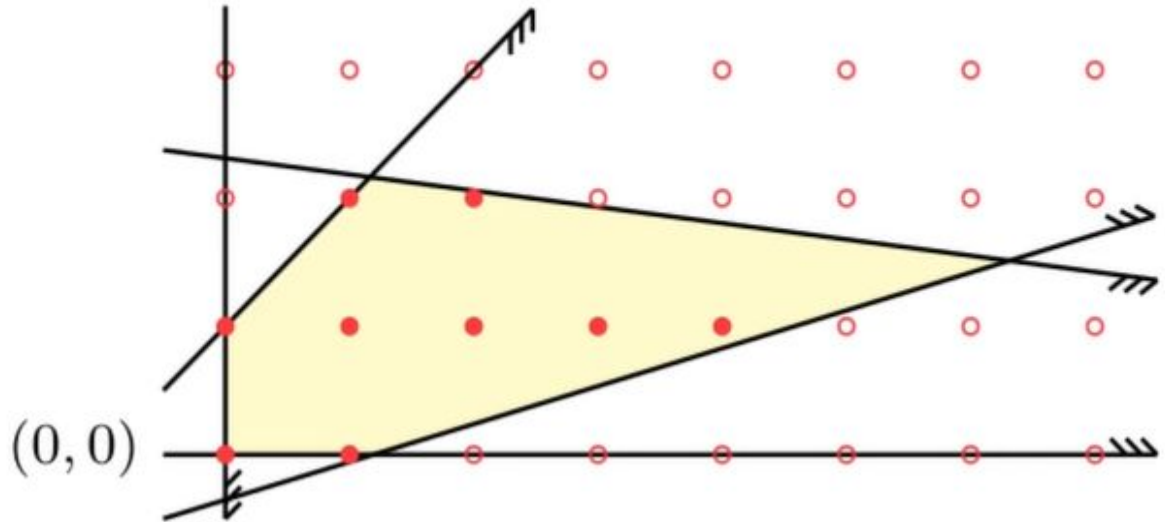


- **Decision Variables:** for each possible assignment of a clinician to a client in a time slot
 - (clinician_id, client_id, slot) - each value is a binary variable indicating whether a specific clinician is scheduled to supervise a particular client during a specific time slot.
- **Objective:** maximizing the utilization of clinician's available hours / maximizing the scheduling of clinician's available working hours
- **Constraints:** same as before

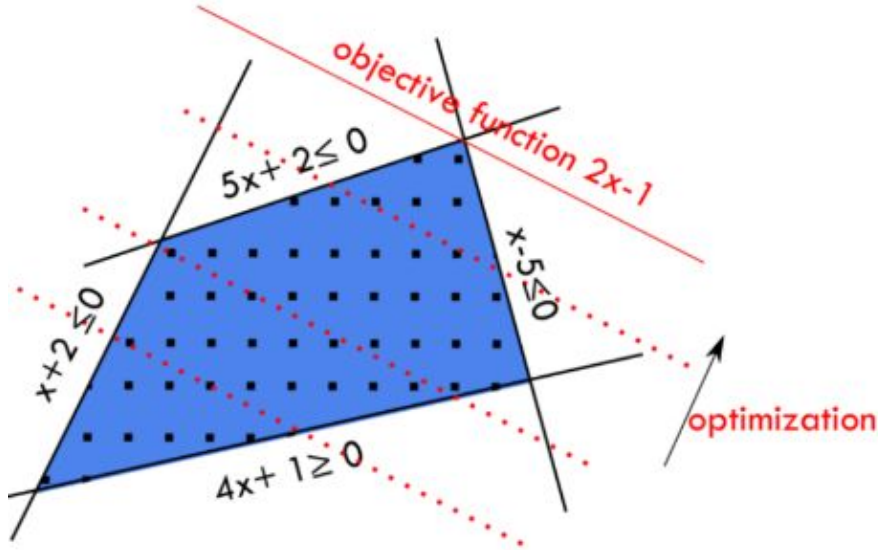
What is ILP?



1. Plot constraints
2. **Identify feasible region**
 - Where all constraints overlap
3. **Find optimal solution**
 - Find values of variables that give you the best scheduling while satisfying constraints



ILP - Python Library



Dashed lines: Different levels of the objective function
Dots: Feasible integer solutions

- PuLP
 - `model = LpProblem("Clinician_Scheduling", LpMaximize)`
 - `model.solve()`
 - **Optimal Solution:** point within the feasible region that is closest to optimal objective function

ILP - Underlying Algorithm



- **Branch and Bound Algorithm**
 - Used for optimization problems, it keeps track of the best solution found so far and prunes or eliminates branches that cannot yield better solutions
- **Steps**
 - Solve the linear programming **relaxation** of the ILP problem = allowing variables to be fractional values and solving the problem without integer constraints
 - **Bounding**: Calculate objective function value of solution obtained from the LP relaxation
 - **Branching**: if solution is not an integer → divide or branch the problem into smaller subproblems (create new constraints to force variables to be integers)
 - **Pruning**: when evaluating the new subproblems, if they are not better than the current best integer solution, they are pruned
 - **Repeat!**

Pros and Cons of ILP



Pros

- Optimality - guarantees optimal solution given constraints
- Wide range applicability
- Extension of LP > for discrete variables

Cons

- ILP is NP-Hard and although Branch and Bound is applied to address this issue (not NP-Hard), it is still computationally complex
- Integer solutions introduce additional constraints
- Assumes linear relationships

What is the process for solving Branch and Bound/ILP?



- a. Bounding \rightarrow Linear Relaxation \rightarrow Branching \rightarrow Pruning
- b. Pruning \rightarrow Branching \rightarrow Bounding \rightarrow Linear Relaxation
- c. Linear Relaxation \rightarrow Bounding \rightarrow Branching \rightarrow Pruning
- d. Linear Relaxation \rightarrow Branching \rightarrow Bounding \rightarrow Pruning

What is the process for solving Branch and Bound/ILP?



- a. Bounding \rightarrow Linear Relaxation \rightarrow Branching \rightarrow Pruning
- b. Pruning \rightarrow Branching \rightarrow Bounding \rightarrow Linear Relaxation
- c. **Linear Relaxation \rightarrow Bounding \rightarrow Branching \rightarrow Pruning**
- d. Linear Relaxation \rightarrow Branching \rightarrow Bounding \rightarrow Pruning

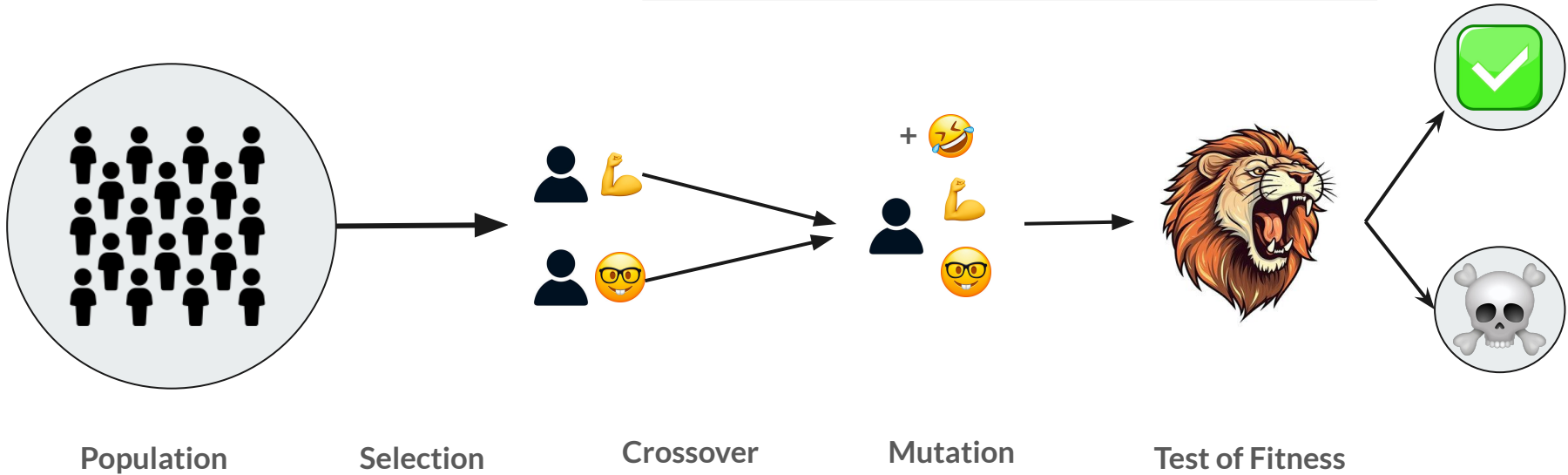
Genetic Algorithms

Inspiration

Inspired by Nature: Mimics the process of natural selection.

Population-Based: Maintain a population of candidate solutions.

Evolution: Solutions evolve over generations.

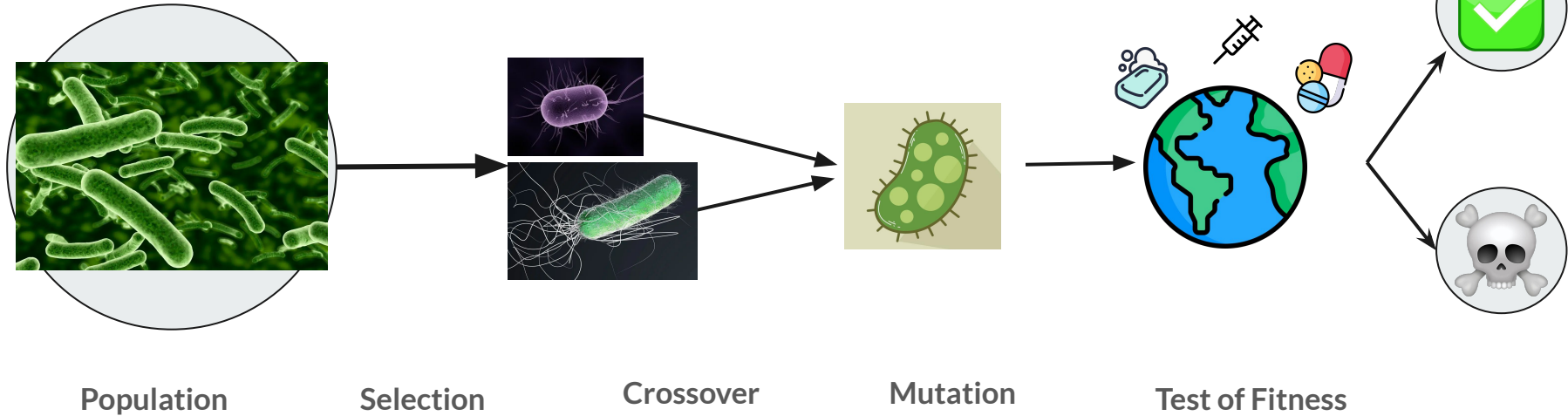


Inspiration

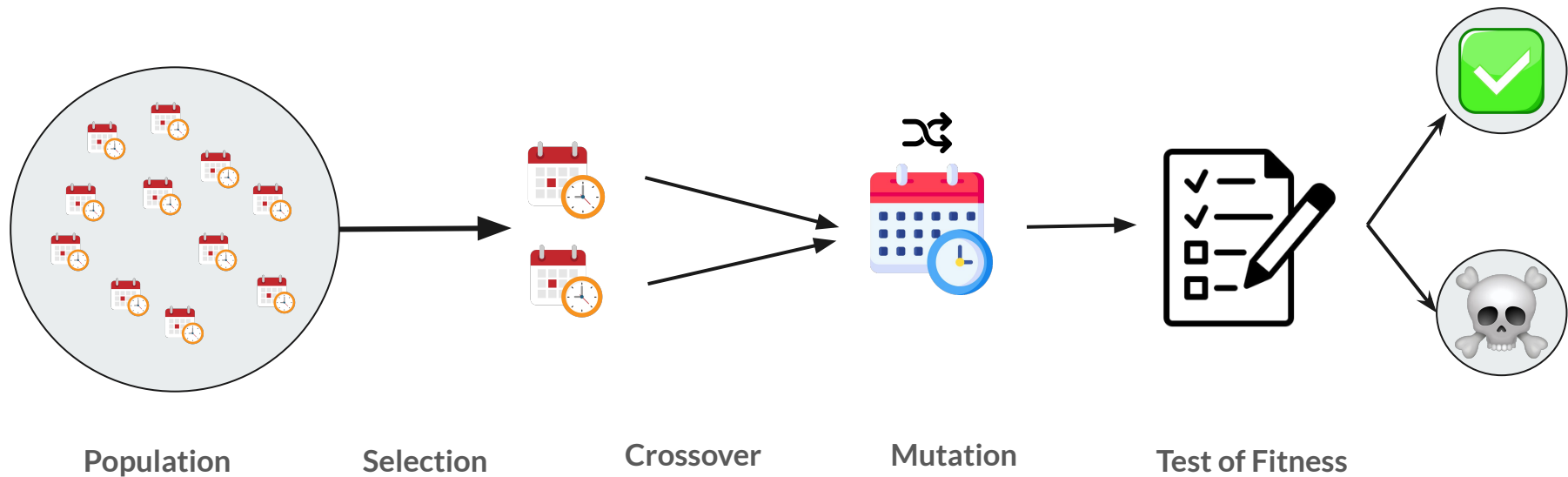
Inspired by Nature: Mimics the process of natural selection.

Population-Based: Maintain a population of candidate solutions.

Evolution: Solutions evolve over generations.



Implementation





Key Terms

- **Genes:** Represent parts of a solution.
- **Chromosomes:** Combination of genes (a complete solution).
- **Fitness Function:** Evaluates how good a solution is.
- **Selection:** Chooses the best solutions to reproduce.
- **Crossover:** Combines two solutions to create new ones.
- **Mutation:** Introduces random changes to maintain diversity.



Key Terms

- **Gene:** A Client-Clinician Time Block
- **Chromosome:** The entire schedule for all clients
- **Fitness Function:** Constraint check
- **Selection:** Chooses the best schedules
- **Crossover:** Combines schedules to make a new one
- **Mutation:** Random shuffle of schedule after crossover



Applying GAs for the Impossible Scheduling Problem

Problem Context:

- Schedule clients for a clinician's available time blocks.
- **Constraints:**
 - Each client needs X hours/week.
 - Sessions ≤ 3 hours.
 - 8 hours gap between sessions.
 - At least 2 sessions/week.

Steps:

1. **Initialization:**
 - Generate initial population of schedules randomly.
2. **Fitness Function:**
 - Score schedules based on how well they meet the constraints.
 - Example: Reward for meeting hours, having ≥ 2 sessions, ensuring 8-hour gaps.
3. **Selection:**
 - Select schedules with higher fitness scores for reproduction.
4. **Crossover:**
 - Combine parts of two parent schedules to create offspring.
5. **Mutation:**
 - Randomly modify schedules to introduce variability.
6. **Iteration:**
 - Repeat selection, crossover, and mutation for multiple generations



Population

Purpose: Initializes a population of random schedules for the genetic algorithm.

Logic:

- Create an empty population list.
- For each individual in the population (loop runs `population_size` times):
 - Create an empty schedule dictionary.
 - For each client:
 - Determine the required hours and calculate the number of sessions needed (minimum 2).
 - Generate random supervision sessions for the client, ensuring each session falls within the clinician's available hours and does not exceed 3 hours.
 - Add the sessions to the client's entry in the schedule.
 - Add the generated schedule to the population.

```
def generate_initial_population(population_size, clinician_availability, clients):
    population = []
    for _ in range(population_size):
        schedule = {}
        for client in clients:
            required_hours = clients[client]["required_hours"]
            num_sessions = max(2, (required_hours + 2) // 3)
            sessions = []
            for _ in range(num_sessions):
                day = random.choice(list(clinician_availability.keys()))
                start_hour = random.choice(clinician_availability[day])
                duration = min(3, required_hours)
                sessions.append((day, start_hour, duration))
                required_hours -= duration
            schedule[client] = sessions
        population.append(schedule)
    return population
```

Purpose: Evaluates how well a schedule meets the constraints and assigns a fitness score.

Logic:

- Initialize a score variable.
- For each client in the schedule:
 - Calculate the total scheduled hours and compare it to the required hours.
 - Ensure each client has at least 2 sessions per week.
 - Check that sessions for the same client on the same day have at least an 8-hour gap.
- The score is incremented based on how well the schedule meets these criteria.

```
def fitness(schedule, clients):
    score = 0
    for client, sessions in schedule.items():
        required_hours = clients[client]["required_hours"]
        total_hours = sum(duration for _, _, duration in sessions)
        if total_hours == required_hours:
            score += 1
        if len(sessions) >= 2:
            score += 1
        for i in range(len(sessions)):
            for j in range(i + 1, len(sessions)):
                day1, start1, duration1 = sessions[i]
                day2, start2, duration2 = sessions[j]
                if day1 == day2:
                    if abs(start1 - start2) >= 8:
                        score += 1
    return score
```



Selection

Purpose: Selects schedules from the population based on their fitness to form the next generation.

Logic:

- Use a weighted random selection where the probability of selecting a schedule is proportional to its fitness.
- Use Python's `random.choices` with the `weights` parameter to perform the selection.

```
def selection(population, fitnesses, num_parents):  
    selected = random.choices(population, weights=fitnesses, k=num_parents)  
    return selected
```



Crossover

Purpose: Combines pairs of parent schedules to create new offspring schedules.

Logic:

- For each client, randomly choose sessions from either parent to create two new child schedules.

```
def crossover(parent1, parent2):  
    child1, child2 = {}, {}  
    for client in parent1.keys():  
        if random.random() < 0.5:  
            child1[client] = parent1[client]  
            child2[client] = parent2[client]  
        else:  
            child1[client] = parent2[client]  
            child2[client] = parent1[client]  
    return child1, child2
```




Mutation

Purpose: Introduces random changes to schedules to maintain genetic diversity.

Logic:

- For each client in the schedule:
 - With a probability of `mutation_rate`, randomly change one of the client's sessions.
 - Select a random day, start hour, and duration within the constraints for the new session.

```
def mutate(schedule, clinician_availability, mutation_rate=0.01):  
    for client, sessions in schedule.items():  
        if random.random() < mutation_rate:  
            session_idx = random.randint(0, len(sessions) - 1)  
            day = random.choice(list(clinician_availability.keys()))  
            start_hour = random.choice(clinician_availability[day])  
            duration = min(3, clients[client]["required_hours"] - sum(d for _, d  
                                in schedule[client][session_idx]))  
            schedule[client][session_idx] = (day, start_hour, duration)  
    return schedule
```



Putting it All Together

Purpose: Orchestrates the entire genetic algorithm process, from initialization to producing the final schedule.

Logic:

- Generate an initial population of schedules.
- For a fixed number of generations:
 - Calculate fitness for each schedule.
 - Select parents based on fitness.
 - Perform crossover and mutation to generate a new population.
 - Print the best fitness in each generation.
- Return the best schedule found based on fitness.

```
def genetic_algorithm(clinician_availability, clients, population_size=100,
generations=1000, mutation_rate=0.01):
    population = generate_initial_population(population_size, clinician_availability, clients)
    for generation in range(generations):
        fitnesses = [fitness(schedule, clients) for schedule in population]
        if max(fitnesses) == 2 * len(clients):
            break
        new_population = []
        for _ in range(population_size // 2):
            parents = selection(population, fitnesses, 2)
            child1, child2 = crossover(parents[0], parents[1])
            new_population.append(mutate(child1, clinician_availability, mutation_rate))
            new_population.append(mutate(child2, clinician_availability, mutation_rate))
        population = new_population
        print(f"Generation {generation + 1}: Best fitness = {max(fitnesses)}")
    best_schedule = max(population, key=lambda s: fitness(s, clients))
    return best_schedule

# Run the genetic algorithm
best_schedule = genetic_algorithm(clinician_availability, clients)
print("Best schedule found:")
for client, slots in best_schedule.items():
    print(f"{client}: {slots}")
```



Solution

Best schedule found:

Client 1: [('Monday', 13, 3), ('Wednesday', 9, 3), ('Tuesday', 17, 2)]

Client 2: [('Friday', 10, 3), ('Monday', 13, 3)]

Client 3: [('Thursday', 16, 3), ('Thursday', 14, 1)]

Client 4: [('Wednesday', 10, 3), ('Thursday', 17, 2)]

Client 5: [('Monday', 14, 3), ('Friday', 10, 3), ('Tuesday', 16, 1)]

Client 6: [('Wednesday', 15, 3), ('Saturday', 12, 3), ('Monday', 11, 3)]

Client 7: [('Friday', 12, 3), ('Monday', 16, 0)]

Client 8: [('Thursday', 16, 3), ('Monday', 15, 3)]

Client 9: [('Thursday', 15, 3), ('Sunday', 14, 3), ('Monday', 16, 2)]

Client 10: [('Wednesday', 17, 3), ('Wednesday', 14, 1)]



Genetic Algorithms - Pros and Cons

Pros:

Flexibility: Can handle complex, non-linear constraints.

Global Search: Can escape local optima, exploring a broader search space.

Adaptability: Can be applied to various problems with different constraints..

Cons:

Computationally Expensive: Requires many evaluations.

Parameter Tuning: Needs careful tuning of population size, mutation rate, etc.

Uncertainty: May not guarantee optimal solution; results can vary



Genetic Algorithms - When to use?

Ideal Use Cases:

Complex Optimization Problems: Where constraints are non-linear and difficult to solve with traditional methods.

Large Search Spaces: Where an exhaustive search is impractical.

Flexible Requirements: Where solutions can be approximate and still valuable.

Examples:

Scheduling Problems: Employee shift scheduling, exam timetabling.

Resource Allocation: Project management, supply chain optimization.

Design Optimization: Engineering design, parameter tuning.

Ant Colony Optimization

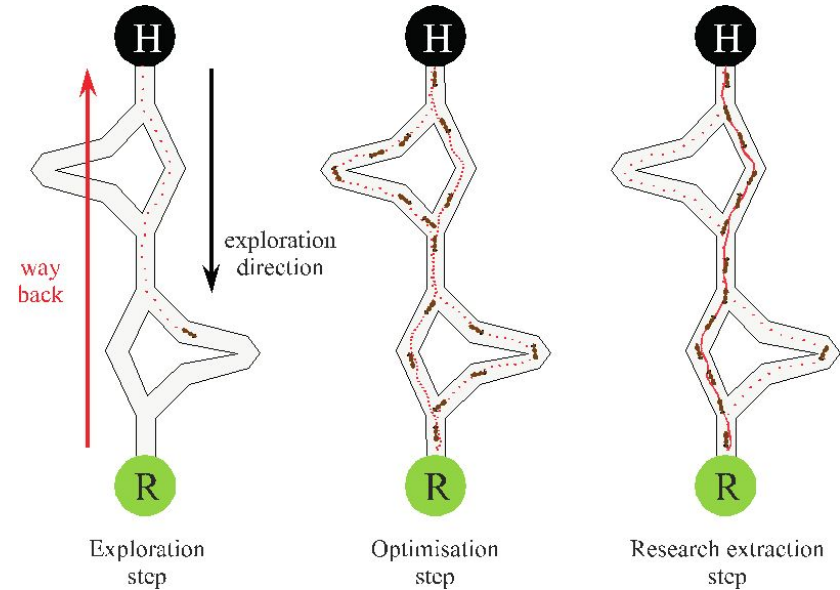
Ant Colony Optimization

Represent the scheduling problem as a graph where nodes are time slots and edges are possible transitions between sessions.

Ants traverse this graph to construct feasible schedules.

Pheromones guide ants to prefer certain paths based on previous successful schedules.

Update pheromones based on the quality of the schedules found by the ants.



Quiz!



What is a primary advantage of using Integer Linear Programming (ILP) over Constraint Satisfaction Problems (CSP) for optimization tasks?

- A) ILP can handle non-linear constraints.
- B) ILP guarantees an exact optimal solution.
- C) ILP is faster for all problem sizes.
- D) ILP requires less detailed problem modeling.




What is a primary advantage of using Integer Linear Programming (ILP) over Constraint Satisfaction Problems (CSP) for optimization tasks?

- A) ILP can handle non-linear constraints.
- **B) ILP guarantees an exact optimal solution.**
- C) ILP is faster for all problem sizes.
- D) ILP requires less detailed problem modeling.



A university needs to schedule exams for various courses. Each course has specific time slots when exams can be held, and no student should have overlapping exams. Time efficiency is not important. Which algorithm would be the best choice to solve this problem and why?

- A) Integer Linear Programming (ILP) because it guarantees an exact optimal solution.
- B) Genetic Algorithms because they are effective at escaping local optima.
- C) Constraint Satisfaction Problems (CSP) because they are highly flexible and can efficiently handle complex constraints.
- D) Ant Colony Optimization (ACO) because they adapt dynamically to changes.




A university needs to schedule exams for various courses. Each course has specific time slots when exams can be held, and no student should have overlapping exams. Time efficiency is not important. Which algorithm would be the best choice to solve this problem and why?

- A) Integer Linear Programming (ILP) because it guarantees an exact optimal solution.
- B) Genetic Algorithms because they are effective at escaping local optima.
- **C) Constraint Satisfaction Problems (CSP) because they are highly flexible and can efficiently handle complex constraints.**
- D) Ant Colony Optimization (ACO) because they adapt dynamically to changes.



A company wants to design a new product by optimizing multiple conflicting objectives such as cost, durability, and performance. The design space is large and complex with many possible configurations. Which algorithm would be the best choice and why?

- A) Integer Linear Programming (ILP) because it guarantees an optimal solution for linear constraints.
- B) Genetic Algorithms because they can handle complex, non-linear constraints and are good for exploring large search spaces.
- C) Constraint Satisfaction Problems (CSP) because they can efficiently reduce the search space through constraint propagation.
- D) Ant Colony Optimization (ACO) because they are effective for combinatorial optimization problems.



A company wants to design a new product by optimizing multiple conflicting objectives such as cost, durability, and performance. The design space is large and complex with many possible configurations. Which algorithm would be the best choice and why?

- A) Integer Linear Programming (ILP) because it guarantees an optimal solution for linear constraints.
- **B) Genetic Algorithms because they can handle complex, non-linear constraints and are good for exploring large search spaces.**
- C) Constraint Satisfaction Problems (CSP) because they can efficiently reduce the search space through constraint propagation.
- D) Ant Colony Optimization (ACO) because they are effective for combinatorial optimization problems.

Summary



Constraint Satisfaction Problems (CSP)

Strengths:

- Highly flexible and can model complex constraints.
- Efficient constraint propagation reduces search space.
- Suitable for problems with strict constraints.

Weaknesses:

- Can be computationally intensive for large problems.
- May require detailed problem modeling.

Use Cases:

- Scheduling, timetabling, resource allocation, puzzle solving.

Integer Linear Programming (ILP)

Strengths:

- Provides exact solutions for optimization problems.
- Well-suited for linear constraints and objectives.
- Can handle large-scale problems with commercial solvers.

Weaknesses:

- Linear constraints might limit modeling complex relationships.
- Computationally expensive for very large or non-linear problems.

Use Cases:

- Supply chain optimization, financial modeling, scheduling with linear constraints.

Genetic Algorithms (GA)

Strengths:

- Can handle complex, non-linear constraints.
- Good for large search spaces and combinatorial optimization.
- Effective at escaping local optima.

Weaknesses:

- Requires careful parameter tuning.
- No guarantee of finding the optimal solution; results may vary.
- Computationally expensive.

Use Cases:

- Complex scheduling, design optimization, evolving neural networks.

Ant Colony Optimization (ACO)

Strengths:

- Effective for combinatorial optimization problems.
- Can dynamically adapt to changes in the problem.
- Good at finding near-optimal solutions in large search spaces.

Weaknesses:

- Requires careful parameter tuning.
- Computationally expensive.
- Pheromone evaporation and deposit rates need balancing.

Use Cases:

- Vehicle routing, network routing, scheduling, and traveling salesman problem.



Which algorithm would you choose for scheduling?

Questions?

Resources



<https://arxiv.org/pdf/1506.08905> Understanding VSIDS

<https://when2work.com/help/mgr/autofill-advanced-details/> W2W

<https://www.geeksforgeeks.org/introduction-to-branch-and-bound-data-structures-and-algorithms-tutorial/> ILP

<https://www.youtube.com/watch?v=lmy1ddn4cyw&t=87s> Google CP-SAT

<https://www.youtube.com/watch?v=1Z4-FNUXj60> Branching - VSIDS

<https://www.geeksforgeeks.org/introduction-to-branch-and-bound-data-structures-and-algorithms-tutorial/> - ILP