PART II. OPTIMIZATION: NUMERICAL APPROACHES
(LECTURE 4)

Shpilev Petr Valerievich
Faculty of Mathematics and Mechanics, SPbU

September, 2025

Comments

In this lecture, we continue the study of population-based and heuristic optimization methods, starting with genetic algorithms. We examine crossover and mutation operators, their different types, and their implementation in Julia, followed by an illustrative example of a full genetic algorithm search. We then expand to other evolutionary and swarm-based methods, including differential evolution, particle swarm optimization (PSO), the firefly algorithm, and cuckoo search, highlighting their underlying ideas, search mechanisms, and practical applications. The lecture also introduces hybrid methods that combine global exploration with local refinement to enhance performance. In the second part, we shift focus to discrete optimization, beginning with integer programming and the relax-and-round strategy, discussing potential pitfalls. We cover the role of totally unimodular matrices and methods to test unimodularity, before concluding with the cutting plane method, both in its formal description and geometric interpretation.

Crossover: Combining Parent Information

Purpose of Crossover: to create new candidate solutions by mixing genetic material (vector coordinates) from two parents.

- **Input:** Two vectors $x, x' \in \mathbb{R}^n$ (parents).
- **Output:** Two new vectors $\bar{x}, \bar{x}' \in \mathbb{R}^n$ (offspring).

Standard Crossover Operators

- **Single-point:** Choose index $i \in \{1, \dots, n-1\}$ at random. Swap tail segments.
- **Two-point:** Choose $i < j$ and exchange segment $[i, j]$.
- **Uniform:** For each coordinate k, pick from x_k or x'_k at random.

- **All methods are symmetric:** swapping parents gives same distribution over outputs.
- **Trade-off:** more mixing vs. preserving structure (adjacent coordinates).

Interpolation-Based Crossover: For real-valued vectors, we can define crossover as interpolation:

$$\bar{x} = (1-\alpha)x + \alpha x', \quad \bar{x}' = (1-\alpha)x' + \alpha x, \quad \text{for } \alpha \in [0,1]$$

Comments

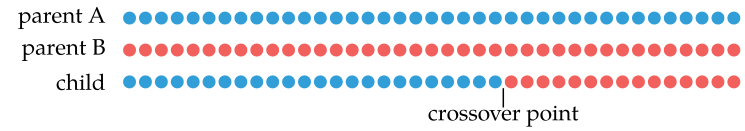
Crossover is a key step in genetic algorithms. It combines information from two parent solutions to generate new ones, ideally inheriting good properties from both. The basic idea is to create two offspring vectors by mixing the coordinates of the two parents.

There are several standard crossover methods. In single-point crossover, we randomly choose a position in the vector and swap the tail segments between the two parents. In two-point crossover, we select a segment of coordinates and exchange it. In uniform crossover, we go coordinate by coordinate, randomly deciding which parent's value to take.

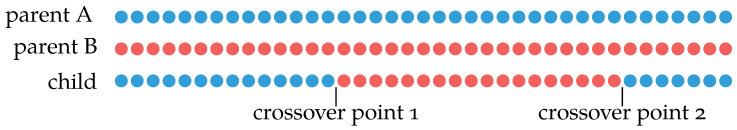
All these methods are symmetric, meaning the choice of which vector is labeled "parent 1" or "parent 2" doesn't affect the final distribution. There is a trade-off: more aggressive mixing can promote exploration, while preserving longer blocks of coordinates can help retain useful structure.

For real-valued vectors, we can also use interpolation-based crossover. Instead of swapping coordinates, we take convex combinations: each offspring becomes a weighted average of the two parents. This is especially useful when solution vectors represent continuous parameters — it allows smooth transitions and fine-grained control over offspring traits.

Single-point crossover:



Two-point crossover:



Uniform crossover:

Comments

This slide shows visual illustrations of the three main crossover strategies discussed earlier.

In single-point crossover, we randomly pick one crossover point and take the left segment from parent A and the right segment from parent B. This creates a child that blends two contiguous parts from different parents.

In two-point crossover, we pick two random points and swap the segment between them. This allows for more diverse recombination, especially when meaningful structure is localized in the middle of the vector.

In uniform crossover, each coordinate of the child is independently sampled from either parent with 50% probability. This creates highly mixed offspring and provides the most randomness among the three.

Crossover Operators in Julia

```julia
abstract type CrossoverMethod end  # Abstract supertype for all crossover strategies
struct SinglePointCrossover <: CrossoverMethod end # Single-point crossover
function crossover(::SinglePointCrossover, a, b)
    i = rand(1:length(a))                          # choose crossover point
    return vcat(a[1:i], b[i+1:end])                # merge segments
end
struct TwoPointCrossover <: CrossoverMethod end    # Two-point crossover
function crossover(::TwoPointCrossover, a, b)
    n = length(a)
    i, j = rand(1:n, 2)                            # choose two points
    if i > j; (i, j) = (j, i); end                 # sort them
    return vcat(a[1:i], b[i+1:j], a[j+1:n])        # swap segment
end
struct UniformCrossover <: CrossoverMethod end     # Uniform crossover
function crossover(::UniformCrossover, a, b)
    child = copy(a)
    for i in 1:length(a)
        if rand() < 0.5                            # 50% chance to copy from b
            child[i] = b[i]
        end
    end
    return child
end
```


Comments

This slide shows the Julia implementation of the crossover strategies we've already discussed. All three strategies follow the same interface by using CrossoverMethod as a common abstract supertype. This allows us to write modular and extensible genetic algorithm code.

Each method defines a crossover function that takes two parent vectors a and b and returns one child. The implementations assume the parents are vectors of equal length.

The SinglePointCrossover selects a random position and takes the first part from parent a and the remainder from b.

The TwoPointCrossover selects two random indices and swaps the segment between them, combining elements from both parents in three parts.

The UniformCrossover builds the child element by element, choosing at each position randomly between the two parents.

Using concrete subtypes and dispatch, we can easily switch crossover strategies just by passing a different object, without changing the algorithm's overall structure.

Mutation: Introducing New Traits

Goal: Prevent loss of diversity by introducing new genes into the population.

- If crossover is the only source of variation, some traits may never appear.
- Mutation enables spontaneous emergence of new traits and better exploration of the search space.
- Applied to each child after crossover.

Mutation Strategies

- **Binary strings:** Each bit has small probability of being flipped.
  - Typical mutation rate: $\lambda = 1/m$ for m-bit chromosome.
  - Ensures ≈ 1 mutation per offspring.
- **Real-valued vectors:** Add zero-mean Gaussian noise.
  - σ: standard deviation of noise.
  - Allows smooth variation of traits.

- Bitwise mutation promotes exploration by breaking convergence.
- Gaussian mutation supports fine-grained control and continuous diversity.

Comments

Mutation is essential for maintaining diversity in a genetic algorithm. If we relied only on crossover, we would be limited to recombining traits already present in the population. Over time, this can cause premature convergence: the best individuals dominate, and the population becomes homogeneous. Mutation counters this by introducing random changes to the offspring, allowing the algorithm to explore previously unreachable regions of the search space.

For binary chromosomes, mutation is typically implemented by flipping each bit with a small probability. If the chromosome has m bits, a mutation rate of $1/m$ gives, on average, one mutation per individual — enough to diversify without causing chaos.

For real-valued vectors, flipping bits is less practical. Instead, we add small Gaussian noise to each coordinate. This has the effect of perturbing the solution slightly, which is especially useful for local search and fine-tuning.

Overall, mutation complements selection and crossover. It prevents loss of genetic material and helps the algorithm escape local optima by injecting variability into the population.

Mutation Operators in Julia

```julia
# Define an abstract base type for mutation strategies
abstract type MutationMethod end
# Bitwise mutation: flip each bit with probability λ
struct BitwiseMutation <: MutationMethod
    λ  # mutation rate
end
function mutate(M::BitwiseMutation, child)
    return [rand() < M.λ ? !v : v for v in child]
end
# Gaussian mutation: add noise to each coordinate
struct GaussianMutation <: MutationMethod
    σ  # standard deviation of noise
end
function mutate(M::GaussianMutation, child)
    return child + randn(length(child)) * M.σ
end
```

These functions modify a child after crossover, injecting randomness to promote diversity.

Comments

This slide shows how two common mutation strategies are implemented in Julia. First, we define an abstract type `MutationMethod`, which serves as a common interface for all mutation types.

The `BitwiseMutation` strategy is designed for binary chromosomes. It flips each bit of the child vector with a probability λ. The function uses a list comprehension: for each bit, we generate a random number; if it's less than λ, we flip the bit using !v, otherwise we keep it unchanged.

The `GaussianMutation` strategy works for real-valued vectors. It adds random noise sampled from a normal distribution with standard deviation σ to each coordinate. This allows small, continuous changes in the solution space, which helps fine-tune solutions during optimization.

These functions are typically applied after crossover, just before the new individual is added to the next generation. They ensure that the algorithm does not get stuck exploring only previously seen combinations, preserving long-term search diversity.

Example: Applying a Genetic Algorithm

```
import Random: seed!
import LinearAlgebra: norm

seed!(0)                          # for reproducibility
f = x -> norm(x)                  # objective: minimize Euclidean norm
m = 100                           # population size
k_max = 10                        # number of generations
population = rand_population_uniform(m, [-3, 3], [3, 3])
S = TruncationSelection(10)       # top 10 individuals selected
C = SinglePointCrossover()        # one-point crossover operator
M = GaussianMutation(0.5)         # mutation with σ = 0.5

x = genetic_algorithm(f, population, k_max, S, C, M)
@show x
# Output: x = [-0.0099, -0.0520]
```

Note: This example demonstrates how selection, crossover, and mutation work together in a simple genetic search. The components are modular and easily adaptable to more complex tasks.

Comments

This example illustrates how to use a genetic algorithm in Julia for a simple optimization task. The objective is to minimize the Euclidean norm of a 2D vector — in other words, we are trying to find a point close to the origin.

We begin by creating an initial population of 100 random vectors, with each coordinate sampled uniformly between -3 and 3. Then we run the genetic algorithm for 10 generations.

In each generation, we apply truncation selection to retain the top 10 individuals based on their performance. These serve as parents for the next generation. Crossover is performed using the single-point method, which swaps tails of the parent vectors to generate offspring. After that, Gaussian mutation adds randomness by slightly perturbing the offspring with noise of standard deviation 0.5.

All components — selection, crossover, and mutation — are modular and interchangeable. This makes the implementation flexible and easy to modify for other optimization problems.

Finally, the best individual found is displayed. Although this is a toy example with a simple norm function, the same framework can be applied to more complex and high-dimensional problems, simply by changing the objective function or parameter values.

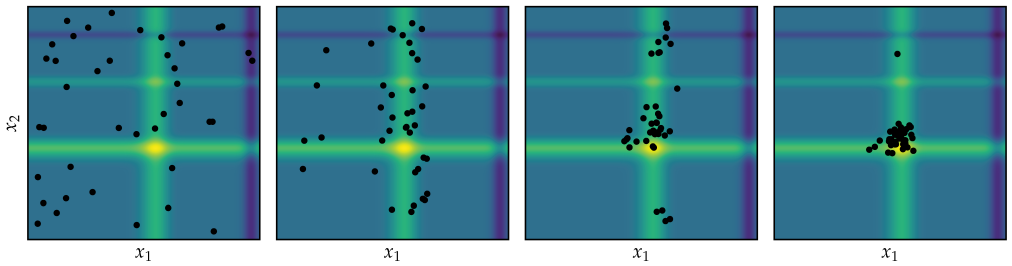


Figure: A genetic algorithm with truncation selection, single point crossover, and Gaussian mutation with $\sigma = 0.1$ applied to **Michalewicz function**.

The Michalewicz function is a d-dimensional optimization function with several steep valleys

$$f(x) = -\sum_{i=1}^{d} \sin(x_i) \cdot \sin^{2m}\left(\frac{ix_i^2}{\pi}\right),$$

where the parameter m, typically 10, controls the steepness. The global minimum depends on the number of dimensions. In two dimensions the minimum is at approximately $[2.20,\ 1.57]$ with $f(x*) = -1.8011$.

Comments

This slide shows the output of applying our genetic algorithm to the Michalewicz function, a well-known multimodal benchmark. This function is challenging because it contains many narrow valleys and local minima — making gradient-based methods prone to getting stuck.

The plotted result demonstrates that genetic algorithms can still make progress in such complex landscapes. By balancing exploration (through mutation and crossover) and exploitation (via selection), the algorithm can find good solutions even when the search space is rugged.

Below the figure, we define the Michalewicz function mathematically. It includes a parameter m, typically set to 10, which controls the steepness of the valleys. In two dimensions, the global minimum is around with a function value of approximately -1.8011.

Differential Evolution: Idea and Intuition

Goal: Improve each individual by combining other individuals via vector arithmetic.

- **Parameters:**
  - Differential weight w (typically, $w \in [0.4, 1.0]$) — scales mutation step.
  - Crossover probability p — controls how often mutated values are accepted.
- **For each individual x in the population:**
  1. Randomly choose three distinct individuals a, b, c.
  2. Compute mutant vector: $z = a + w \cdot (b - c)$.
  3. Choose random dimension $j \in \{1, \ldots, n\}$.
  4. Construct candidate x' as:
$$x'_i = \begin{cases} z_i & \text{if } i = j \text{ or with prob. } p \\ x_i & \text{otherwise} \end{cases}$$
  5. Replace x with x' if $f(x') < f(x)$.

Key Idea: Combine individuals using directional differences — a simple yet effective evolutionary strategy.

Comments

Now we turn to a different population-based optimization method called Differential Evolution. Like genetic algorithms, it operates on a population of candidate solutions and iteratively refines them. However, the way it generates new individuals is conceptually simpler and relies entirely on vector operations — without explicit recombination of parent "genetic material" like in crossover.

Instead of selecting parents and performing crossover and mutation separately, Differential Evolution constructs new candidates by adding weighted differences of other individuals to a base vector. This operation, referred to as "mutation" in DE terminology, is deterministic once the individuals are chosen. It introduces variation by moving along directions spanned by other vectors.

To add stochasticity, a crossover step is applied: for each coordinate, the algorithm decides whether to keep the original value or use the mutated one. At least one coordinate is always changed. This ensures the new candidate differs from the current individual.

Finally, if the new candidate has better objective value, it replaces the current one. This simple strategy — combining directional mutation with greedy selection — makes Differential Evolution both efficient and robust, especially for continuous optimization problems.

Algorithm 17: Differential Evolution

```julia
using StatsBase
function differential_evolution(f, population, k_max; p=0.5, w=1)
    n, m = length(population[1]), length(population) # dimensions & pop. size
    for k in 1:k_max                                 # main loop
        for (k, x) in enumerate(population)          # for each candidate x
            a, b, c = sample(population,  # sample a, b, c distinct from x
                Weights([j != k for j in 1:m]), 3, replace=false)
            z = a + w * (b - c)                      # mutation step
            j = rand(1:n)                  # ensure at least one mutation
            # crossover step: mix x and z
            x′ = [i == j || rand() < p ? z[i] : x[i] for i in 1:n]
            if f(x′) < f(x)                          # selection step
                x[:] = x′                            # replace if better
            end
        end
    end
    return population[argmin(f.(population))]      # return best solution
end
```


Comments

This slide presents a Julia implementation of the differential evolution algorithm.The function `differential_evolution` takes as input: an objective function f, an initial population of candidate vectors, the number of iterations k_max, a crossover probability p, and a differential weight w. The function returns the best solution found in the population.

Inside the main loop, we go over each individual in the population one by one. For each of them, we randomly select three other individuals. Using them, we construct a new trial vector. To do this, we start from the first of the three and add to it a weighted difference between the second and the third. This gives us the mutation vector z.

Next, we build a new candidate by using this the mutation vector with the current one. For each coordinate, we either take the mutated value or keep the original value, depending on the crossover probability. To make sure the new candidate is not identical to the original one, we always include at least one coordinate from the mutated version.

If this new candidate has a lower objective value than the original, it replaces the original one in the population. After all iterations are done, the best vector in the final population is returned.

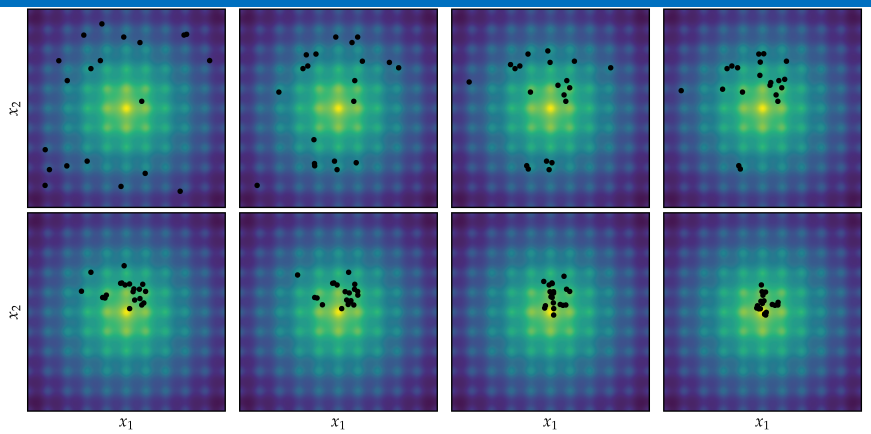


Figure: Differential evolution with p = 0.5 and w = 0.2 applied to **Ackley's function**.

Ackley's function is defined for any number of dimensions d :

$$f(x) = -ae^{\left(-b\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right)} - e^{\left(\frac{1}{n}\sum_{i=1}^{n}\cos(cx_i)\right)} + a + e$$

and has a global minimum at the origin with an optimal value of zero. Typically, a = 20, b = 0.2, and c = 2π.

Comments

This slide shows the behavior of the differential evolution algorithm when applied to a two-dimensional version of the Ackley function — a popular test function in optimization.The parameters used here are a crossover probability of 0.5 and a differential weight of 0.2.

The Ackley function is known for its many local minima, making it a challenging landscape to search. However, it has a single global minimum at the origin, where the function reaches zero. The difficulty lies in navigating the complex surface without getting trapped in one of the many local optima.

What we see in the figure is how the population of candidate solutions evolves over time. Initially, the points are spread across the domain. As iterations progress, they begin to cluster and move toward the center. This gradual convergence shows how differential evolution manages the balance between exploration and exploitation: it tries out new directions while still being guided by the objective function.

The success of the method in this case comes from its ability to combine solutions and explore new regions in a controlled manner. Even with relatively mild settings like low differential weight, the algorithm can make consistent progress toward the optimum.

Particle Swarm Optimization: Idea and Intuition

Key Idea: Maintain a swarm of candidate solutions that evolve over time by sharing information about the best positions found.

- **Swarm:** A population of particles, each with:
  - current position x,
  - velocity v,
  - personal best position x_{best}.
- **Global Best:** The best position found by any particle.
- **Update Rules:**
  - Update velocity:

$$v \leftarrow w \cdot v + c_1 \cdot r_1 \cdot (x_{best} - x) + c_2 \cdot r_2 \cdot (x_{global} - x)$$

  - Update position:

$$x \leftarrow x + v$$

Parameters:
- w — inertia weight (momentum),
- c_1 — cognitive parameter (personal attraction),
- c_2 — social parameter (swarm attraction).

Comments

Now we introduce a method known as Particle Swarm Optimization, or PSO for short. It belongs to the family of population-based optimization methods and draws inspiration from the way groups of animals behave in nature — for example, how birds flock or fish swim in schools. Each individual in the swarm, called a particle, represents a potential solution to the problem.

The basic idea is that these particles move around in the search space, adjusting their positions based on their own previous best results and the best result found by the entire swarm. Each particle has two key properties: a current position and a velocity that determines how it moves.

At every iteration, each particle updates its velocity by taking into account three things: first, its current momentum; second, the best position it has found so far; and third, the best position found by the swarm as a whole. These three influences are balanced by certain parameters, which control how much each factor matters. This lets the particle explore the space while still being guided toward promising regions.

Once the velocity is updated, the particle simply moves in that direction. If the new position leads to a better solution, the particle updates its personal best. And if it's better than the swarm's global best, that gets updated too.

One reason this method works well for continuous optimization is that it allows smooth and adaptive movement through the search space. The behavior of the swarm naturally leads it to cluster around good solutions, even when no gradient information is available.

Algorithm 18: Particle Swarm Optimization

```
function particle_swarm_optimization(f, population, k_max;
    w=1, c1=1, c2=1)              # inertia and acceleration parameters
    n = length(population[1].x)   # dimensionality of search space
    x_best, y_best = copy(population[1].x_best), Inf
    for P in population           # find initial global best
        y = f(P.x)
        if y < y_best; x_best[:], y_best = P.x, y;  end
    end
    for k in 1 : k_max            # main loop over generations
        for P in population
            r1, r2 = rand(n), rand(n)  # random vectors for stochastic update
            P.x += P.v                 # update position
            P.v = w*P.v +              # inertia component
                  c1*r1.*(P.x_best - P.x) +  # cognitive (personal best)
                  c2*r2.*(x_best - P.x)      # social (global best)
            y = f(P.x)
            if y < y_best; x_best[:],y_best = P.x,y; end # update global best
            if y < f(P.x_best); P.x_best[:] = P.x;  end    # update personal best
        end
    end
    return population                 # return updated population
end
```


Comments

On this slide, we see the Julia implementation of the particle swarm optimization algorithm.The algorithm takes as input an objective function f, a list of particles - population, and the number of iterations k_{max}. Optional parameters include w for inertia, c1 and c2 for cognitive and social acceleration respectively. Each particle in the population is a mutable structure with current position x, velocity v, and the best position found so far x_{best}.

The algorithm first identifies the best particle in the current population. Then, in each iteration, every particle updates its position by adding its velocity. Its velocity is in turn updated by three terms: the previous velocity, an attraction toward its own best known position, and an attraction toward the globally best known position, each scaled by random factors and coefficients c1 and c2.

If the new position is better than the previous one, both the global best and personal best are updated.This process continues until the maximum number of iterations is reached.The population is then returned, with each particle hopefully close to an optimal solution.

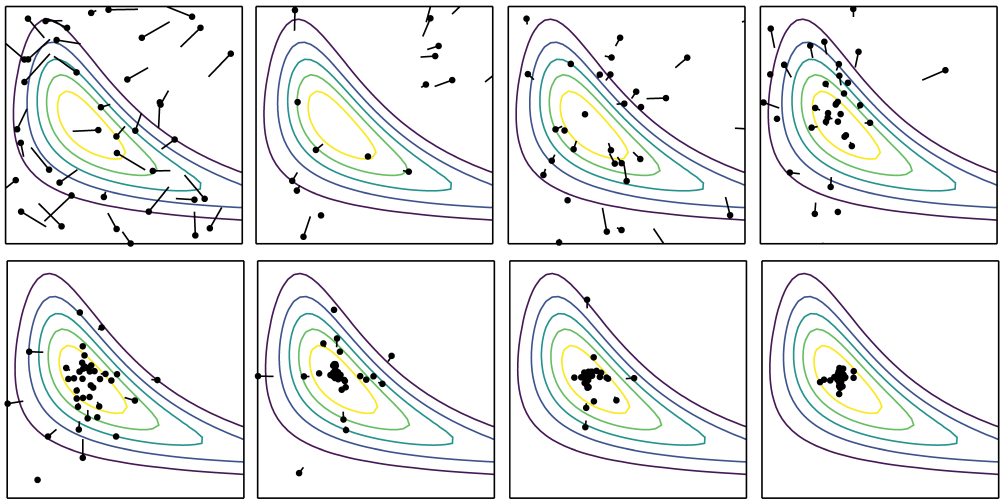


Figure: The particle swarm method with w = 0.1, $c_1 = 0.25$, and $c_2 = 2$.

Comments

This figure illustrates the search process of Particle Swarm Optimization using specific parameter values: inertia weight w=0.1, cognitive coefficient c1=0.25, and social coefficient c2=2. The setting favors convergence toward the globally best solution while limiting the influence of each particle's personal experience.

A low value of reduces momentum, meaning particles quickly adapt their direction instead of coasting along previous trajectories. The small c1 value means each particle gives relatively little weight to its own best-found position, which reduces individualism and encourages alignment with the swarm. In contrast, the large c2 places strong emphasis on the global best position discovered so far, causing particles to move collectively toward promising regions.

The result is a fast and focused convergence pattern, as seen in the figure. This configuration is particularly effective when the optimization landscape is smooth and the global optimum is not hidden behind many local minima. However, in more complex or deceptive search spaces, such aggressive convergence can lead to premature stagnation. Parameter tuning in PSO is therefore a balance between exploration and exploitation.

Firefly Algorithm: Idea and Principles

Inspiration: Fireflies use light to attract mates — brightness represents quality of a solution.

- **Each firefly = a candidate solution.**
- **Attraction proportional to brightness:** brighter fireflies attract others.
- **Movement:** less attractive fireflies move toward more attractive ones.
- **Randomness:** small random steps prevent premature convergence.

Motion rule: $a \leftarrow a + \beta I(\|b - a\|)(b - a) + \alpha\varepsilon$
where ε is a Gaussian random vector, α controls step size, β is base attraction, and $I(\cdot)$ is light intensity.

- **Key component:** intensity decreases with distance:
  $I(r) = e^{-\gamma r^2}$ (Gaussian drop-off)

Goal: Converge toward brighter regions — better solutions.

Comments

The Firefly Algorithm is inspired by the mating behavior of fireflies. Each individual emits a light signal whose intensity represents how good the solution is. Brighter fireflies attract others, and those with lower brightness move toward them. The idea is simple but powerful: better solutions attract others and guide the population through the search space.

Each firefly is a candidate solution, and attraction between fireflies depends on both their brightness and distance. As fireflies get farther apart, the attraction decreases — modeled here by a Gaussian intensity drop-off function. The movement of a firefly has two components: one deterministic — a step toward a more attractive neighbor, and one stochastic — a random perturbation to ensure diversity and prevent premature convergence.

The motion rule combines these elements: it includes a bias toward brighter individuals and a small Gaussian noise term. The parameters β, α, and γ control attraction strength, randomness, and light absorption, respectively.

In summary, the Firefly Algorithm explores the landscape by mimicking natural signaling and attraction behavior, balancing exploitation of known good areas and exploration of new regions.

Algorithm 19: Firefly Optimization

```julia
using Distributions
function firefly(f, population, k_max;
    β=1, α=0.1, brightness=r -> exp(-r^2))
    m = length(population[1])                    # dimension of solution
    N = MvNormal(Matrix(1.0I, m, m))             # Gaussian noise
    for k in 1:k_max
        for a in population, b in population     # compare every pair
            if f(b) < f(a)                       # if b is "brighter"
                r = norm(b - a)                  # compute distance
                # move a toward b + random walk
                a[:] += β * brightness(r) * (b - a) + α * rand(N)
            end
        end
        return population[argmin([f(x) for x in population])]
    end
```

Note: Updates are in-place — each vector `a` is modified directly in the population.

Comments

This code provides a complete implementation of the Firefly Algorithm in the Julia language. The function takes three main inputs: an objective function, a population of candidate solutions, and the number of optimization steps to perform. Additionally, the user can adjust three parameters: the attraction coefficient beta, the strength of random motion alpha, and the brightness decay function. By default, brightness decreases with distance following a Gaussian curve.

The algorithm compares every pair of individuals in the population. If one firefly is brighter — meaning it corresponds to a lower value of the objective function — then the other one moves in its direction. This movement combines two effects: a deterministic step toward the brighter solution, and a random displacement drawn from a Gaussian distribution.

After all optimization steps, the function returns the best solution in the updated population — that is, the firefly with the lowest function value.

Now an important point about how Julia works. In this implementation, each candidate solution is stored as a vector. In Julia, vectors are mutable and passed by reference. This means that when we update the elements of a vector inside a loop, we are directly modifying the contents of the original population. Even though we don't replace the population as a whole, its contents evolve over time. That's why the final result reflects the accumulated movement of all fireflies.

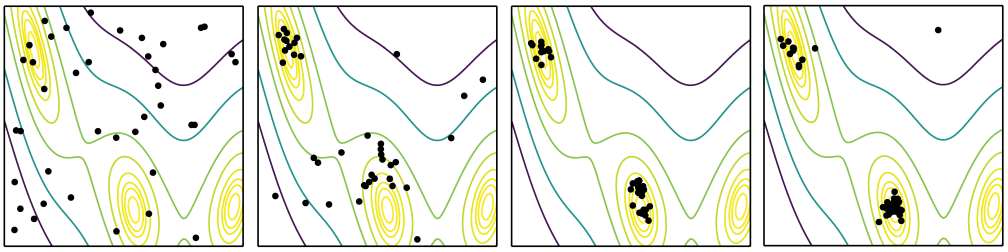


Figure: Firefly search with $\alpha = 0.5$, $\beta = 1$, and $\gamma = 0.1$ applied to the **Branin function**.

The Branin function is a two-dimensional function

$$f(x) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)\cos(x_1) + s,$$

with recommended values $a = 1$, $b = 5.1/(4\pi^2)$, $c = 5/\pi$, $r = 6$, $s = 10$, and $t = 1/(8\pi)$. It has no local minima aside from global minima with $x_1 = \pi + 2\pi m$ for integral m. Four of these minima are:

$$\left\{\begin{pmatrix}-\pi\\12.275\end{pmatrix}, \begin{pmatrix}\pi\\2.275\end{pmatrix}, \begin{pmatrix}3\pi\\2.475\end{pmatrix}, \begin{pmatrix}5\pi\\12.875\end{pmatrix}\right\} \quad \text{with } f(x^*) \approx 0.397887.$$

Comments

This slide shows how the Firefly Algorithm performs when applied to the Branin function — a well-known test case in global optimization. The parameters used are: alpha equals zero point five, which controls the amount of random motion, beta equals one, which sets the base attraction strength, and gamma equals zero point one, which controls how quickly brightness fades with distance.

The Branin function is two-dimensional and designed to test the ability of an algorithm to locate multiple global optima. It has no local minima aside from its global ones, which are located at specific x-values: pi, three pi, five pi, and so on. The function's value at each of these global minima is approximately zero point three nine eight.

What we see in the figure is how a swarm of fireflies converges toward these minima. The brighter individuals, which represent better solutions, attract the others over time. Thanks to the balance between directed movement and random noise, the algorithm manages to escape shallow regions and spread out over the landscape.

The shape of the Branin function makes it ideal for illustrating how a population-based algorithm like this can find multiple promising regions simultaneously. In this example, the fireflies successfully explore different valleys and gather around multiple global solutions.

Cuckoo Search: Idea and Principles

Biological inspiration: Cuckoos lay eggs in nests of other species. Some are accepted, others are destroyed.

- Each nest = one solution.
- New solutions = "eggs" placed in random nests.
- Best solutions survive to the next generation.
- Bad nests may be replaced with new random solutions.

Key idea: Perform global search via random walks known as Lévy flights, which have heavy-tailed step lengths.

- **Lévy flight:** Random step drawn from a Cauchy-like distribution.
- Better solutions replace worse ones.
- Discovery probability: With some chance, host detects the foreign egg — solution is discarded.

Balance: Exploitation via selection of best nests, exploration via Lévy flights and random replacement.

Comments

Cuckoo Search is a population-based optimization method inspired by the unusual reproductive strategy of cuckoo birds. Cuckoos lay their eggs in the nests of other birds, often of different species. Sometimes the host bird raises the foreign chick, and sometimes it recognizes the egg as foreign and throws it out. This behavior is used as a metaphor for how solutions are introduced and replaced during optimization.

In the algorithm, each solution is represented by a nest. New candidate solutions — referred to as eggs — are placed into random nests. If a new egg has better quality than the existing one, it replaces it. Otherwise, it might be discarded. The best solutions are always preserved, while poor ones are periodically replaced by entirely new random solutions.

What makes this method powerful is the way it generates new solutions. Instead of using standard Gaussian random walks, Cuckoo Search uses Lévy flights — random steps with heavy-tailed distributions. These allow occasional long jumps, which help the algorithm escape local optima and explore the search space more broadly.

Another key feature is the discovery probability — the chance that a host detects and removes a foreign egg. This models the occasional loss of unpromising candidates and injects additional randomness. Overall, the method combines exploration through Lévy flights with exploitation by preserving high-quality solutions.

Algorithm 20: Cuckoo Search with Lévy flights and selective replacement

```
using Distributions
        # Each solution is stored as a "nest" with its position and value
mutable struct Nest
    x                                          # current position (vector)
    y                                               # objective value f(x)
end

function cuckoo_search(f, population, k_max;
    p_a=0.1, C=Cauchy(0,1))  # p_a: discovery prob., C: step distribution
    m, n = length(population), length(population[1].x)  # pop. size & dim
    a = round(Int, m * p_a)                    # number of nests to replace

    for k in 1 : k_max
                                    # Generate new solution via Lévy flight
        i, j = rand(1:m), rand(1:m)              # select two random nests
        x = population[j].x + [rand(C) for _ in 1:n]      # new candidate
        y = f(x)
```


Comments

This first part of the code sets up the foundation of the Cuckoo Search algorithm in Julia. Since most of you may not have used Julia before, let's walk through the syntax step by step.

The first line `using Distributions` loads a library that gives us access to probability distributions, such as the Cauchy distribution. We will need this later to generate random steps for the search.

Next, we define a custom data structure with `mutable struct Nest`. Recall in Julia, a struct is like a container that can hold related data. Here, each Nest represents a candidate solution. It stores two things: x, which is the position in the search space (a vector of numbers), and y, which is the value of the objective function f(x) at that position. The keyword mutable means that once a nest is created, we can still change its contents during the algorithm — this is useful because we want to update solutions over time.

The main function `cuckoo_search` takes several arguments: f is the objective function we are trying to minimize; `population` is the initial set of nests; `k_max` is the number of iterations we will run.

There are also optional parameters: `p_a`, the discovery probability, and C, the step distribution.

Inside the function, m is the number of nests (population size), and n is the number of dimensions in each solution vector. We also calculate a, the number of nests to replace in each iteration, as a fraction of the total population.

Finally, the loop for `k in 1:k_max` defines the main optimization process. In each iteration, the algorithm selects two random nests and generates a new candidate solution by adding random steps (sampled from a Cauchy distribution) to the position of one nest. This step is called a Lévy flight, and it allows the search to explore the solution space more broadly.

Cuckoo Search implementation (Part 2)

```
        if y < population[i].y
             population[i].x[:] = x                    # Replace if better
             population[i].y = y
        end
                                # Sort nests by objective value (descending)
        p = sortperm(population, by = nest -> nest.y, rev = true)
              # Replace worst 'a' nests with Lévy-flight-based solutions
        for i in 1:a
            j = rand(1:(m - a)) + a      # choose better nest to imitate
            new_x = population[p[j]].x + [rand(C) for _ in 1:n]
            new_y = f(new_x)                    # evaluate at new position
            population[p[i]] = Nest(new_x, new_y)
        end
    end
    return population
end
```

Note: Worst nests are refreshed by mutation from better ones — preserving diversity.

Comments

We evaluate this new candidate and compare it to another randomly chosen individual from the population. If the new candidate is better — meaning it has a lower value of the objective function — then we update the existing individual with the new data.

Now here's a technical point: in Julia, when we copy one vector into another, we usually don't create a new object. Instead, we overwrite the contents of the existing array. So even though it looks like a simple assignment, the code is actually replacing the elements of the old solution with the values of the new one — in place, without reallocation.

The last part of the code is responsible for replacing bad solutions to preserve diversity in the population. First, we sort the entire population from worst to best, based on the objective function values. That gives us an ordering where the least promising solutions come first.

Then we go through the worst individuals — meaning the ones with the highest function values — and replace them with new candidates. But we don't generate these new candidates completely at random. Instead, for each bad solution, we pick a better one from the rest of the population and add a random Cauchy step to it. This way, the new solution is still influenced by something that already performs well, but also introduces variation.

This mutation mechanism allows the algorithm to get rid of poor-performing solutions while still exploring new areas of the search space. It's important that we don't just keep the best individuals — we need variation to avoid getting stuck.

At the end, the entire updated population is returned, and we can then pick the best solution from it externally.

Cuckoo Search: Example

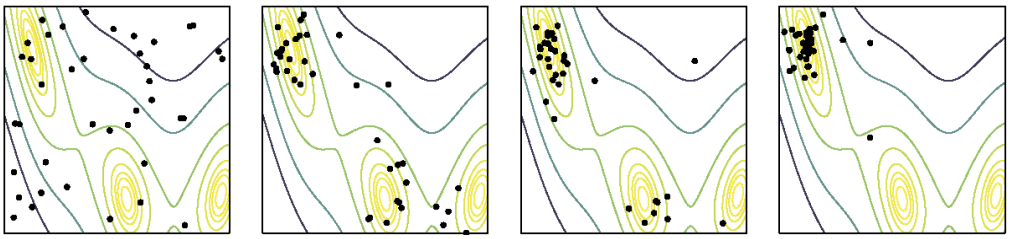


Figure: Cuckoo search applied to the **Branin function**.

Advantages:

- simple structure;
- global search via Lévy flights;
- minimal parameter tuning.

Limitations:

- slow convergence in fine-tuning phase;
- random replacement may discard useful diversity.

Comments

This slide shows the behavior of the Cuckoo Search algorithm applied to the Branin function — a standard benchmark with several global minima. As we saw earlier, the Branin function has its minima located along the x-axis at pi, three pi, and so on, and its landscape is smooth but multi-modal.

What you see in the figure is the result of running Cuckoo Search with Lévy flight steps. The heavy-tailed nature of the step distribution allows the algorithm to cover wide areas of the space. Unlike algorithms that rely on small local steps, this method occasionally explores distant regions — which is why it can escape local traps and locate multiple optima.

Now let's briefly summarize the main strengths and limitations of Cuckoo Search. One major advantage is its simplicity — the algorithm has a clean structure and requires relatively few parameters to tune. The use of Lévy flights gives it strong global exploration capabilities. This makes it effective for finding promising regions in complex landscapes.

However, there are trade-offs. Because the replacement step is random, it may sometimes eliminate useful diversity in the population. Also, since the algorithm doesn't explicitly refine solutions near the optimum, its convergence can be slower in the final fine-tuning stage compared to gradient-based methods.

So in practice, Cuckoo Search is best used as a global optimizer — either standalone, or as part of a hybrid method where a second algorithm handles local refinement.

Hybrid Methods: Combining Global and Local Search

Core Idea: Combine population-based global search with local optimization for refined solutions.

- **Global methods** (e.g., evolutionary algorithms, swarm intelligence):
  - Robust to local minima
  - Explore wide regions of the search space
- **Local methods** (e.g., gradient descent, quasi-Newton):
  - Efficient near optima
  - Provide fast convergence and precision

Two Hybrid Strategies

- **Lamarckian Learning:** Replace individuals with locally optimized versions.
- **Baldwinian Learning:** Keep original individuals, but evaluate selection using locally improved fitness.

- **Goal:** Balance exploration (diversity) and exploitation (accuracy).

Comments

Hybrid methods combine the strengths of two worlds — population-based global exploration and local refinement. Global methods, such as evolutionary algorithms or particle swarm optimization, are effective at navigating large, complex search spaces. They are especially helpful in avoiding poor local minima and identifying promising regions.

However, once a region is located, global methods tend to be inefficient in fine-tuning the solution. That's where local search methods, such as gradient descent or quasi-Newton techniques, shine. They offer rapid convergence and high precision when close to a good solution.

A hybrid method brings these two elements together. There are two main ways for doing this: Lamarckian and Baldwinian learning. The difference lies in how we integrate the results of local search into the population.

In Lamarckian learning, once an individual is locally improved — for example, by gradient descent — we replace the original version in the population with the improved one. As a result, the population itself becomes more refined. This approach directly fuses global and local search: global methods explore broadly, and then local optimization polishes the best candidates, which are fed back into the evolutionary process. So the improvement is inherited, just like acquired traits passed down genetically.

In Baldwinian learning, local optimization is still applied, but it's used only for evaluation. The original individual stays in the population unchanged, but its fitness is measured using the improved version. This means that even though the individual hasn't changed, its chances of being selected for reproduction increase — simply because its locally optimized version performs better. The key here is that the population remains diverse, but the selection pressure is guided by local search results.

In both cases, local search helps accelerate convergence, while global methods ensure robustness and exploration. The difference is whether local improvements affect the population directly (Lamarckian) or only indirectly through fitness (Baldwinian).

Hybrid Methods: Example

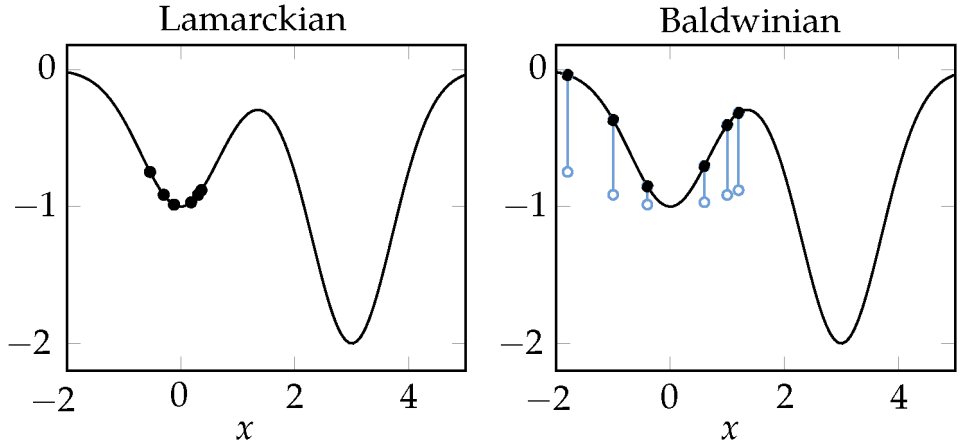


Figure: Hybrid method combining particle swarm optimization with local search.

Observation:

- the Baldwinian strategy maintains greater diversity during the search;
- the Lamarckian approach leads to faster convergence.

Comments

This example compares two hybrid strategies based on particle swarm optimization. Both approaches incorporate local search, but they do so differently: the left plot uses the Baldwinian strategy, while the right plot applies the Lamarckian variant.

In the Baldwinian case, local optimization is applied only during evaluation — the swarm itself is not changed. As a result, particles retain more of their original structure and continue to explore the search space. You can see that the trajectories are more dispersed, and the swarm preserves diversity over time. This slows convergence slightly but helps prevent premature stagnation.

In the Lamarckian strategy, local search updates are directly injected into the swarm. As soon as a particle is improved locally, it replaces the original. Consequently, the population converges much faster, since each iteration incorporates refinement. However, this can reduce diversity and lead to earlier convergence to a possibly suboptimal region.

The choice between these strategies depends on the problem. If maintaining diversity is important — for instance, when there are many local minima — the Baldwinian method may be better. If we value fast convergence near a known region, Lamarckian updates can be more effective.

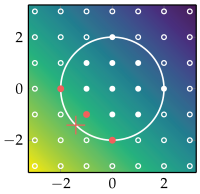
Key Difference: A discrete optimization problem constrains design variables to a discrete set, often integers.

- **Typical applications:** manufacturing with fixed-size components, route selection, scheduling.
- Design space may be finite or infinite, but enumeration is usually infeasible.
- Many methods from earlier topics (e.g., simulated annealing, genetic programming) can be adapted, but focus here is on new categories of techniques.

Example: Minimize $x_1 + x_2$ subject to $\|x\| \leq 2$ and x integer.

- Continuous optimum: $x^* = [-\sqrt{2}, -\sqrt{2}]$, value $y = -2\sqrt{2} \approx -2.828$.
- Best integer solution: $y = -2$ at $x^* \in \{[-2, 0], [-1, -1], [0, -2]\}$.

The typical effect: The feasible set becomes more restricted, and the optimal value is often worse than in the continuous counterpart.



Comments

Earlier we have focused on optimizing problems involving design variables that are continuous. Now we will consider a special area - discrete optimization.

In discrete optimization, our design variables can take only certain discrete values — most often integers. This restriction naturally arises in many applications, such as manufacturing, where parts come in fixed sizes, or in navigation, where we choose between a finite set of paths. Some discrete problems have infinite design spaces, but many are finite. Even when the space is finite, enumerating all possibilities is usually computationally impractical. Therefore, we use specialized optimization methods that avoid brute force search.

A key observation is that many optimization techniques developed for continuous problems, such as simulated annealing or genetic programming, can be adapted to handle discrete constraints. However, the present discussion will concentrate on categories of methods that have not yet been covered, and which are specifically designed for discrete optimization.

To illustrate the nature of discrete constraints, consider the task of minimizing the sum of two variables, x_1 and x_2, subject to the condition that the Euclidean norm of the vector x is less than or equal to two, and both variables must be integers. If there were no integrality constraint, the optimal solution would be the vector with both components equal to minus the square root of two, giving an objective value equal to minus two times the square root of two, approximately minus two point eight two eight. Imposing integrality changes the outcome: the best achievable value is minus two, reached at the integer vectors minus two, zero, minus one, minus one, or zero, minus two. This example highlights the typical effect of discrete constraints: the feasible set becomes more restricted, and the optimal value is often worse than in the continuous counterpart.

Definition: An integer program is a linear program with some or all design variables constrained to be integers.

- Applications: operations research, communications, scheduling, logistics.
- Solvers: Gurobi, CPLEX, and others can handle millions of variables.

Standard form:

$$\min_{x} \quad c^\top x$$
$$\text{s.t.} \quad Ax \le b, \quad x \ge 0, \quad x \in \mathbb{Z}^n$$

Equality form:

$$\min_{x,s} \quad c^\top x$$
$$\text{s.t.} \quad Ax + s = b, \quad x \ge 0,\ s \ge 0, \quad x \in \mathbb{Z}^n$$

Mixed integer program: Some variables are discrete, others continuous:

$$\min_{x} c^\top x \quad \text{s.t.} \quad Ax = b,\ x \ge 0,\ x_{\mathfrak{D}} \in \mathbb{Z}^{\|\mathfrak{D}\|}$$

where $\mathfrak{D}$ is the set of indices of discrete design variables, and $x = [x_{\mathfrak{D}}, x_{\mathfrak{C}}]$ splits x into discrete ($x_{\mathfrak{D}}$) and continuous ($x_{\mathfrak{C}}$) components.

Comments

An integer program is a special case of a linear program in which some or all design variables are restricted to take integer values. This type of problem appears in many practical domains: operations research, where tasks must be assigned to discrete resources; communication networks, where channel allocations are integers; scheduling, where jobs are assigned to discrete time slots; and logistics, where numbers of transported units are whole numbers. The field of integer programming is highly developed, and modern solvers such as Gurobi and CPLEX are capable of handling very large problems, with millions of variables, often within reasonable computation times.

In standard form, an integer program seeks to minimize the scalar product of the cost vector c and the design vector x, subject to the system $Ax \le b$, nonnegativity of x, and the requirement that x belongs to the set of n-dimensional integer vectors. Equality form is often used in practice, obtained by adding slack variables s that are nonnegative but not necessarily integers, so that the constraints take the form $Ax + s = b$.

A more general setting is the mixed integer program, where the vector of design variables is divided into two parts: xD, which must be integral, and xC, which may take continuous values. The index set D indicates which components are discrete. This structure allows modeling problems that combine discrete and continuous decision variables within a unified optimization framework.

Idea: Remove integrality, solve a continuous relaxation, then round discrete variables to the nearest integers.

Procedure:

1. Relax integrality: solve LP without $x \in \mathbb{Z}^n$ to get x_c^*.
2. Round discrete components: $x_i \leftarrow \text{round}(x_i)$ for $i \in D$.
3. If needed, repair feasibility.

Caveats

- Rounding can produce infeasible solutions.
- The nearest integer point may be far from optimal.

Bound for integer A

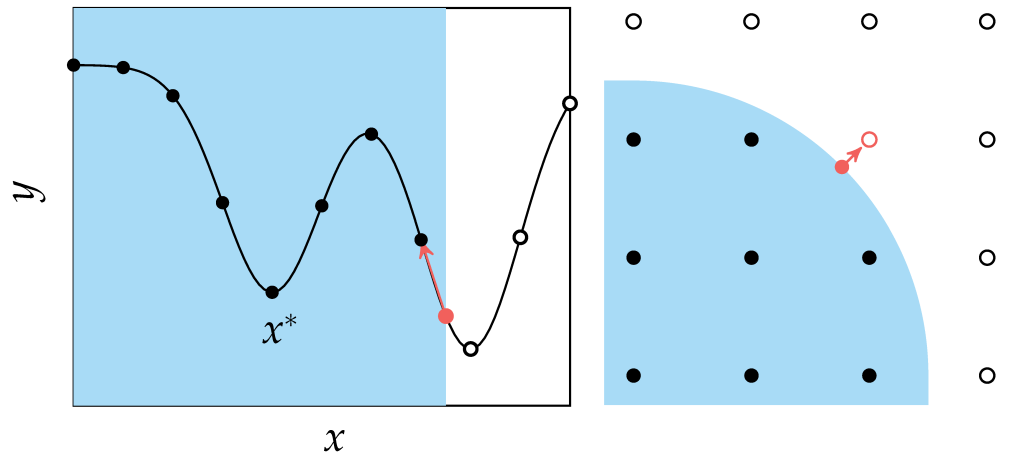
There exists an optimal integer x_d^* with $\|x_c^* - x_d^*\|_\infty \leq n \cdot \Delta(A)$, where $\Delta(A)$ is the maximum absolute determinant of any square submatrix of A.

Comments

The relax-and-round approach begins by ignoring the integrality constraints and solving the resulting continuous relaxation, for example a linear program with the same objective and constraints but allowing all variables to be real numbers. Let the optimal continuous solution be denoted by x_c^*. In the second step, each variable that must be discrete is replaced by the nearest integer. This produces a candidate solution for the original problem. If the rounded point violates some constraints, an additional repair step is applied — for instance, adjusting certain coordinates to restore feasibility.

This method is attractive because solving the relaxation is usually much faster than tackling the integer problem directly, and rounding is computationally trivial. However, there are two important drawbacks. First, rounding may lead to infeasible solutions if the constraints are tight and not aligned with the integer grid. Second, even if feasibility is preserved, the rounded point can be far from the true optimal integer solution.

For problems where the matrix A in the constraints has only integer entries, it is possible to bound how far the rounded solution can be from an optimal integer solution. Specifically, there exists an optimal integer vector x_{d^*} whose maximum coordinate-wise deviation from x_c^* is at most n times $\Delta(A)$, where $\Delta(A)$ denotes the largest absolute value of the determinant among all square submatrices of A. This gives a worst-case proximity guarantee, although in practice the gap can still be large.



Left: Even when feasible, the nearest integer point can be far from optimal.

Right: Rounding may yield an infeasible point that violates constraints.

Comments

The limitations of the relax-and-round approach can be understood clearly from two simple geometric illustrations. In the first diagram, shown on the left, rounding the continuous optimum to the nearest feasible discrete point does produce a valid solution, but this point is not the best among the discrete options. In fact, it can be significantly worse in terms of the objective value than another discrete point that is farther away from the continuous optimum. This happens because the geometry of the discrete feasible set can be very different from that of the continuous relaxation, so proximity in Euclidean distance does not guarantee proximity in objective performance.

The second diagram, shown on the right, depicts a different issue. Here, the feasible region for the discrete problem consists of isolated points. The continuous relaxation has a feasible region covering a larger, continuous area. The optimal continuous point lies within this region, but when we round it to the nearest integer coordinates, the resulting point falls outside the original discrete feasible set. This makes the rounded solution infeasible, meaning it violates one or more constraints of the integer problem.

These two cases highlight why rounding, although simple, must be applied with caution. Feasibility is not guaranteed, and even when it is preserved, there is no assurance that the objective value will be close to the best possible discrete solution.

Totally Unimodular Matrices

Definition: A matrix A is totally unimodular if the determinant of every square submatrix is 0, 1, or −1.

- If A is totally unimodular and b is integral, every vertex of $\{x \mid Ax = b, x \geq 0\}$ is integral.
- The inverse of a totally unimodular matrix (if it exists) is also integral.
- In such cases, solving the LP relaxation gives an exact integer solution.

Example: The matrices

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} -1 & -1 & 0 & 0 & 0 \\ 1 & 0 & -1 & -1 & 0 \\ 0 & 1 & 1 & 0 & -1 \end{bmatrix}$$

are totally unimodular, while

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{bmatrix}$$

is not (determinant −2 in a 2×2 submatrix).

Comments

So, we have shown that rounding a continuous relaxation can produce solutions that are infeasible or far from optimal. This happens because, in general, the feasible region of a linear program contains fractional vertices when integrality is removed. However, there is an important structural exception.

If the matrix A satisfies the property of total unimodularity, and the right-hand side vector b is integer, then every vertex of the feasible region of the linear program is already an integer point. In such cases, solving the continuous relaxation gives an exact integer optimum without any rounding or additional integer programming methods. This makes certain discrete problems as easy to solve as their continuous counterparts.

A matrix is called totally unimodular if the determinant of every square submatrix is zero, plus one, or minus one. If such a matrix is invertible, its inverse also has only integer entries. Many important combinatorial optimization problems, such as network flows and bipartite matchings, have constraint matrices with this property, which explains why they can be solved efficiently using only linear programming.

To illustrate, among the example matrices shown, two are totally unimodular, while one is not — it contains a two-by-two submatrix with determinant equal to minus two. Detecting total unimodularity can be done algorithmically, and this property is valuable to recognize, as it eliminates the need for integer-specific solution techniques.

Algorithm 21: Testing Total Unimodularity

```
isint(x, ϵ=1e-10) = abs(round(x) - x) ≤ ϵ
function is_totally_unimodular(A::Matrix)# Returns true if x is within ϵ of an integer
    if any(a ∉ (0,-1,1) for a in A)  # Step 1: All entries must be 0, 1, or -1
        return false
    end    # Step 2: Brute force check of every square submatrix determinant
    r, c = size(A)                              # number of rows and columns
    for i in 1 : min(r,c)                       # submatrix sizes: 1×1 up to k×k
        for a in subsets(1:r, i)                # all combinations of i rows
            for b in subsets(1:c, i)            # all combinations of i columns
                B = A[a,b]                      # extract submatrix
                if det(B) ∉ (0,-1,1)            # determinant not allowed
                    return false
                end
            end
        end
    end
    return true
end
function is_totally_unimodular(MIP) # Check TU property for A, & integrality of b & c
    return is_totally_unimodular(MIP.A) && #Here MIP is a mutable struct with A, b, c
           all(isint, MIP.b) && all(isint, MIP.c) #applies isint to elements of b & c
end
```


Comments

This algorithm verifies whether a given matrix, or the constraint matrix of a mixed integer program, is totally unimodular. The helper function isint checks if a number is effectively an integer, allowing for a small numerical tolerance of ten to the power minus ten, to avoid issues with floating-point rounding errors.

The main function is$_{\text{t}}$otally$_{\text{u}}$nimodular first ensures that every entry of the matrix is either zero, plus one, or minus one. If any element fails this test, the matrix cannot be totally unimodular. Next, the algorithm systematically examines every possible square submatrix. This is done by iterating over all combinations of i rows and i columns, where i ranges from one up to the smaller of the row and column counts. For each submatrix B, it computes the determinant and checks if it lies in the set 0,1,-1. If any determinant is outside this set, the matrix is not totally unimodular.

The second function is$_{\text{t}}$otally$_{\text{u}}$nimodular(MIP) is designed for mixed integer programs. It calls the matrix-based check on A, and also verifies that both the right-hand side vector b and the cost vector c consist entirely of integers. If all conditions pass, the matrix or program is classified as totally unimodular.

In practice, this brute force determinant checking is computationally expensive for large matrices, but it is exact and works for small to medium-sized problems, making it suitable for teaching and verification purposes.

Purpose: An exact method for solving mixed integer programs when the constraint matrix is not totally unimodular.

- Start with the linear programming relaxation of the integer problem.
- Solve the relaxed problem to obtain an optimal (possibly fractional) solution.
- Detect violation of integer constraints and add **cutting planes** — extra linear inequalities.
- Each cut removes some fractional solutions but leaves all integer-feasible solutions intact.
- Repeat until the solution satisfies all integer constraints.

Practical use: Modern solvers employ branch-and-cut — combining cutting planes with branch-and-bound for efficiency.

Comments

The cutting plane method is a classical exact approach for solving mixed integer programs in cases where the constraint matrix is not totally unimodular. This situation is common in practice, meaning that the relaxation method described earlier will often produce fractional optima without a simple rounding guarantee.

The key idea is to begin by ignoring the integrality constraints and solving the corresponding linear programming relaxation. This provides an optimal solution to the relaxed problem, but that solution may have fractional values in variables that should be integers. We then check whether the current solution violates integrality. If it does, we attempt to construct a new linear inequality — called a cutting plane — that is satisfied by every integer-feasible point but violated by the current fractional solution.

By adding this cut to the set of constraints and resolving the linear program, we eliminate the current fractional solution and possibly many others, while keeping all integer-feasible points. This process is repeated: solve, detect violation, add cut, solve again. The iterations continue until the optimal solution is integer-feasible.

In practice, pure cutting plane algorithms can be inefficient if many cuts are needed. Modern solvers therefore use a branch-and-cut approach, which integrates cutting planes with branch-and-bound approach which we'll discuss further.

Cutting Plane Method: Formal Description

Starting point: Solve the LP relaxation and obtain a vertex x_c^* of $Ax = b$. If all discrete components $x_{\mathfrak{D}}^*$ are integers, the method stops.

- If some $x_{\mathfrak{D}}^*$ are fractional $\Rightarrow$ construct a cutting plane that excludes x_c^* but keeps all integer-feasible points.
- Partition the vertex:

$$A_B x_B^* + A_V x_V^* = b, \quad x_V^* = 0$$

- Fractional values occur only in x_B^*.

Cut for each $b \in B$ with fractional x_b^*

$$x_b^* - \lfloor x_b^* \rfloor - \sum_{v \in V} \left(\bar{A}_{bv} - \lfloor \bar{A}_{bv} \rfloor \right) x_v \leq 0, \text{ where } \bar{A} = A_B^{-1} A_V.$$

These inequalities depend only on V-components (x_v) and cut out the current fractional solution while preserving all integer-feasible solutions.

Comments

Let us now give a formal description of this approach. Let "x_c^*" be the solution to the relaxed linear program, which must be a vertex of the polyhedron defined by "$Ax = b$" If all components of "x_c^*" corresponding to the discrete index set script D are integers, then this point is also optimal for the original mixed integer program and the procedure stops.

Otherwise, we construct a new linear inequality — the cutting plane — that is satisfied by all integer-feasible points but violated by the current "x_c^*". To do this, we use the usual vertex partition: the index set is split into basic variables, denoted B, and nonbasic variables, denoted V. At a vertex, the nonbasic components "x_V^*" are equal to zero, so any fractional values can only appear in "x_B^*" For each basic variable with a fractional value, we derive a specific inequality, shown in the blue box. This inequality, based on the fractional part of x_b^* and the transformed constraint matrix, ensures that the fractional solution is excluded.

As highlighted in the orange box, these inequalities depend only on the non-basic variables and are carefully designed to maintain all integer-feasible solutions while cutting out the fractional ones. By adding these cuts to the LP relaxation and resolving, we refine the feasible region, moving closer to an integer solution. This iterative process of solving, checking, and adding cuts continues until an integer-feasible solution is found.

Cutting Plane Method: Violation and Equality Form

Why the cut excludes the fractional solution x_c^*: Introducing a cutting plane constraint cuts out the relaxed solution x_c^*, because all x_v are zero in the current vertex:

$$x_b^* - \lfloor x_b^* \rfloor \underbrace{\geq 0}_{\text{fractional part}} - \sum_{v \in V} \left(\bar{A}_{bv} - \lfloor \bar{A}_{bv} \rfloor \right) x_v \underbrace{= 0}_{x_v^* = 0} > 0$$

This violates the inequality ≤ 0, so x_c^* is excluded from the feasible region.

- For integer-feasible solutions, where x_b is integer, $x_b - \lfloor x_b \rfloor = 0$, and the sum is non-negative (since $x_v \geq 0$ and fractional coefficients ≥ 0), so $-\sum(\ldots)x_v \leq 0$, preserving them.

Converting the cut to equality form

A cutting plane is written in equality form using an additional integral slack variable $x_k \geq 0$:

$$x_k + \sum_{v \in V} \left(\lfloor \bar{A}_{bv} \rfloor - \bar{A}_{bv} \right) x_v = \lfloor x_b^* \rfloor - x_b^*$$

This slack variable absorbs the inequality, allowing the augmented LP to remain in equality form ($Ax = b$). Each iteration adds one constraint and one variable per fractional basic component.

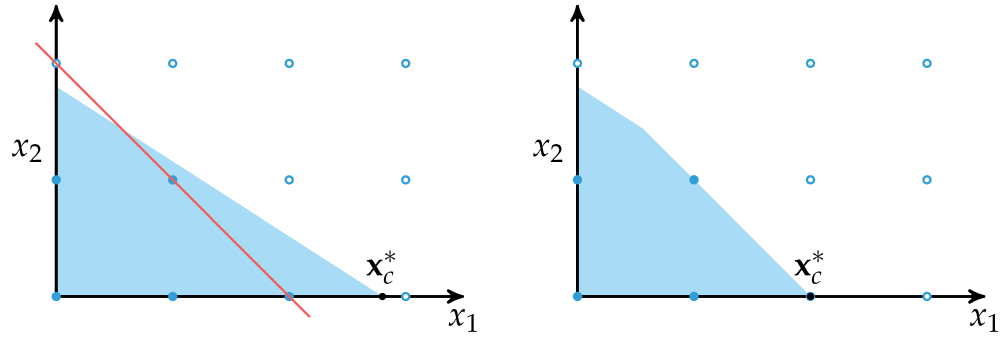
Comments

Having derived the inequality, we now explain why it works and how it can be incorporated into the LP system. In the current fractional vertex “x star sub c,” all nonbasic components “x star sub V” are zero. This means the sum term in the inequality disappears, leaving only the fractional part of “x star sub b” — a strictly positive number — on the left-hand side. Since the inequality requires the left-hand side to be less than or equal to zero, the current solution violates it and is therefore excluded from the feasible region.

For any integer-feasible solution, the basic variable “x sub b” is integer, so “x sub b minus floor of x sub b” equals zero. The coefficients associated with nonbasic variables are fractional but nonnegative, and since “x sub v” are nonnegative as well, the sum term cannot make the left-hand side positive. Thus the cut preserves all integer-feasible points.

In practice, LP solvers work with equality systems “A x equals b.” To integrate the cutting plane without altering this structure, we introduce an additional nonnegative slack variable, “x sub k,” which converts the inequality into an equality. This slack variable accounts for any unused portion of the bound. Each time we add a cut, we append one new constraint and one new variable, maintaining the LP’s tableau form. Iteratively applying this procedure for each fractional basic component gradually tightens the feasible region until only integer-feasible points remain.

Cutting Plane Method: Geometric Illustration



Idea illustrated:

- Left: LP relaxation admits a fractional optimal vertex (black point). A cutting plane (red line) removes it but keeps all integer-feasible points.
- Right: feasible region of the LP after adding the cut — smaller, but still containing all integer solutions.

Comments

This figure shows the geometric meaning of the cutting plane method. On the left, we have the feasible region of the LP relaxation: it is a polygon that contains both integer-feasible points and fractional ones. The optimal solution to the relaxation, marked as a black point, lies at a fractional vertex. The red line represents the cutting plane we derived earlier. It slices off this fractional point while leaving every integer-feasible point intact.

On the right, we see the new feasible region after adding the cut. It is strictly smaller than before, because we have removed not only the previous fractional vertex but also any other points violating the cut. However, all integer-feasible solutions remain inside. The process will now be repeated: solve the new LP, check for fractional components, and if necessary, add another cut. Geometrically, each cut "shaves" part of the polyhedron, bringing the LP feasible region closer to the convex hull of integer-feasible points.