CS498: Algorithmic Engineering

Lecture 16: Non-Convex Landscapes, Simulated Annealing & Genetic Algorithms

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 09 – 03/12/2026

Outline

Where We Are in the Course

Part I (Lectures 1–9):

- LP, IP, modeling patterns.
- Branch-and-bound.
- Row generation, lazy cuts.
- **Philosophy:** structured discrete problems $\Rightarrow$ exact solvers.

Where We Are in the Course

Part I (Lectures 1–9):

- LP, IP, modeling patterns.
- Branch-and-bound.
- Row generation, lazy cuts.
- **Philosophy:** structured discrete problems $\Rightarrow$ exact solvers.

Part II (Lectures 10–15):

- Gradient descent, PyTorch.
- Projected GD, penalty methods.
- Lagrangian duality, KKT, SDP.
- **Philosophy:** convexity $\Rightarrow$ every local min is global.

Where We Are in the Course

Part I (Lectures 1–9):

- LP, IP, modeling patterns.
- Branch-and-bound.
- Row generation, lazy cuts.
- **Philosophy:** structured discrete problems $\Rightarrow$ exact solvers.

Part II (Lectures 10–15):

- Gradient descent, PyTorch.
- Projected GD, penalty methods.
- Lagrangian duality, KKT, SDP.
- **Philosophy:** convexity $\Rightarrow$ every local min is global.

Today (last lecture in Part II): What do we do when *neither* assumption holds?

We will revisit gradient descent/ML after part III of the course (SAT/SMT Solvers).

When Gradient Descent Breaks

Recall from Lecture 10: for **convex** f, gradient descent finds the global minimum.

When Gradient Descent Breaks

Recall from Lecture 10: for **convex** f, gradient descent finds the global minimum.

But: consider minimizing this:

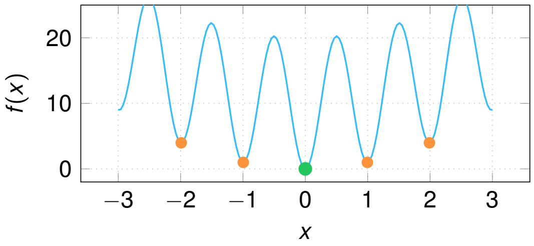
$$f(x) = x^2 - 10\cos(2\pi x) + 10$$

When Gradient Descent Breaks

Recall from Lecture 10: for **convex** f, gradient descent finds the global minimum.

But: consider minimizing this:

$$f(x) = x^2 - 10\cos(2\pi x) + 10$$

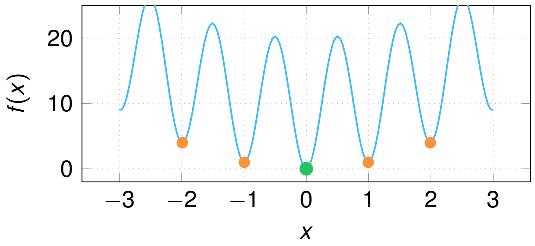


When Gradient Descent Breaks

Recall from Lecture 10: for **convex** f, gradient descent finds the global minimum.

But: consider minimizing this:

$$f(x) = x^2 - 10\cos(2\pi x) + 10$$



This is the **Rastrigin function**. Global minimum at $x = 0$, but local minima *everywhere*. Gradient descent converges to whichever local minimum is nearest to the starting point.

GD Gets Stuck: A Concrete Demonstration

```python
import torch

def rastrigin(x):
    return x**2 - 10 * torch.cos(2 * torch.pi * x) + 10

x = torch.tensor(1.7, requires_grad=True)    # start near a local min
alpha = 0.01
for k in range(500):
    loss = rastrigin(x)
    loss.backward()
    with torch.no_grad():
        x -= alpha * x.grad
        x.grad.zero_()

print(f"x = {x.item():.4f}, f(x) = {rastrigin(x).item():.4f}")
# x = 1.8203,  f(x) = 9.0374    <-- stuck at a local minimum!
```

GD Gets Stuck: A Concrete Demonstration

```python
import torch

def rastrigin(x):
    return x**2 - 10 * torch.cos(2 * torch.pi * x) + 10

x = torch.tensor(1.7, requires_grad=True)    # start near a local min
alpha = 0.01
for k in range(500):
    loss = rastrigin(x)
    loss.backward()
    with torch.no_grad():
        x -= alpha * x.grad
        x.grad.zero_()

print(f"x = {x.item():.4f}, f(x) = {rastrigin(x).item():.4f}")
# x = 1.8203,  f(x) = 9.0374    <-- stuck at a local minimum!
```

GD converged, but to the **wrong** answer. The global min has $f(0) = 0$.

GD Gets Stuck: A Concrete Demonstration

```python
import torch

def rastrigin(x):
    return x**2 - 10 * torch.cos(2 * torch.pi * x) + 10

x = torch.tensor(1.7, requires_grad=True)    # start near a local min
alpha = 0.01
for k in range(500):
    loss = rastrigin(x)
    loss.backward()
    with torch.no_grad():
        x -= alpha * x.grad
        x.grad.zero_()

print(f"x = {x.item():.4f}, f(x) = {rastrigin(x).item():.4f}")
# x = 1.8203,  f(x) = 9.0374    <-- stuck at a local minimum!
```

GD converged, but to the **wrong** answer. The global min has $f(0) = 0$.

Random restarts help, but in d dimensions with m local minima per axis, there are m^d local minima total. You cannot restart your way out.

A Caveat: GD in High Dimensions

We just saw GD fail badly on a 1D function. Does that mean GD is useless for non-convex problems?

A Caveat: GD in High Dimensions

We just saw GD fail badly on a 1D function. Does that mean GD is useless for non-convex problems?

Not necessarily. In very high dimensions (millions of parameters), the landscape looks different:

- Local minima tend to have objective values close to the global minimum.

A Caveat: GD in High Dimensions

We just saw GD fail badly on a 1D function. Does that mean GD is useless for non-convex problems?

Not necessarily. In very high dimensions (millions of parameters), the landscape looks different:

- Local minima tend to have objective values close to the global minimum.
- The real obstacles are **saddle points**, not local minima, and SGD escapes those naturally.

A Caveat: GD in High Dimensions

We just saw GD fail badly on a 1D function. Does that mean GD is useless for non-convex problems?

Not necessarily. In very high dimensions (millions of parameters), the landscape looks different:

- Local minima tend to have objective values close to the global minimum.
- The real obstacles are **saddle points**, not local minima, and SGD escapes those naturally.
- We don't fully understand *why* GD works so well in high dimensions. This is an active research question.

A Caveat: GD in High Dimensions

We just saw GD fail badly on a 1D function. Does that mean GD is useless for non-convex problems?

Not necessarily. In very high dimensions (millions of parameters), the landscape looks different:

- Local minima tend to have objective values close to the global minimum.
- The real obstacles are **saddle points**, not local minima, and SGD escapes those naturally.
- We don't fully understand *why* GD works so well in high dimensions. This is an active research question.

In Part IV (Machine Learning), we will train neural networks with millions of parameters using plain SGD on wildly non-convex losses. And it will work. We'll discuss why when we get there.

A Caveat: GD in High Dimensions

We just saw GD fail badly on a 1D function. Does that mean GD is useless for non-convex problems?

Not necessarily. In very high dimensions (millions of parameters), the landscape looks different:

- Local minima tend to have objective values close to the global minimum.
- The real obstacles are **saddle points**, not local minima, and SGD escapes those naturally.
- We don't fully understand *why* GD works so well in high dimensions. This is an active research question.

In Part IV (Machine Learning), we will train neural networks with millions of parameters using plain SGD on wildly non-convex losses. And it will work. We'll discuss why when we get there.

For now: when the problem is low-dimensional, combinatorial, or has no usable gradients at all, GD is not an option. That's where metaheuristics come in.

When Exact Solvers Break

What about using our tools from Part I?

When Exact Solvers Break

What about using our tools from Part I?

Recall Lectures 7 & 9: we solved TSP *exactly* using Gurobi:

- Arc variables $x_{ij} \in \{0, 1\}$, MTZ subtour elimination (Lecture 7).
- DFJ constraints via lazy callbacks (Lecture 9).

When Exact Solvers Break

What about using our tools from Part I?

Recall Lectures 7 & 9: we solved TSP *exactly* using Gurobi:

- Arc variables $x_{ij} \in \{0, 1\}$, MTZ subtour elimination (Lecture 7).
- DFJ constraints via lazy callbacks (Lecture 9).

That works great for moderate *n*. But:

When Exact Solvers Break

What about using our tools from Part I?

Recall Lectures 7 & 9: we solved TSP *exactly* using Gurobi:

- Arc variables $x_{ij} \in \{0, 1\}$, MTZ subtour elimination (Lecture 7).
- DFJ constraints via lazy callbacks (Lecture 9).

That works great for moderate n. But:

	B&B + lazy cuts	**Time**
$n = 20$	Optimal tour found	$<$ 1 second
$n = 200$	Optimal tour found	minutes to hours
$n = 5{,}000$	Does not finish	$>$ 24 hours

When Exact Solvers Break

What about using our tools from Part I?

Recall Lectures 7 & 9: we solved TSP *exactly* using Gurobi:

- Arc variables $x_{ij} \in \{0, 1\}$, MTZ subtour elimination (Lecture 7).
- DFJ constraints via lazy callbacks (Lecture 9).

That works great for moderate n. But:

	B&B + lazy cuts	**Time**
$n = 20$	Optimal tour found	$<$ 1 second
$n = 200$	Optimal tour found	minutes to hours
$n = 5{,}000$	Does not finish	$>$ 24 hours

The gap

GD needs convexity (or gradients). B&B needs structure and moderate size.
What if we have neither?

The Key Idea: Accept Bad Moves Sometimes

Gradient descent only moves **downhill**. That’s why it gets trapped.

The Key Idea: Accept Bad Moves Sometimes

Gradient descent only moves **downhill**. That’s why it gets trapped.

What if we allowed *uphill* moves?

The Key Idea: Accept Bad Moves Sometimes

Gradient descent only moves **downhill**. That's why it gets trapped.

What if we allowed *uphill* moves?

- Always accept moves that **improve** the objective.

The Key Idea: Accept Bad Moves Sometimes

Gradient descent only moves **downhill**. That’s why it gets trapped.

What if we allowed *uphill* moves?

- Always accept moves that **improve** the objective.
- **Sometimes** accept moves that make the objective worse, to escape local minima.

The Key Idea: Accept Bad Moves Sometimes

Gradient descent only moves **downhill**. That's why it gets trapped.

What if we allowed *uphill* moves?

- Always accept moves that **improve** the objective.
- **Sometimes** accept moves that make the objective worse, to escape local minima.
- **Gradually** reduce the probability of accepting bad moves, so we eventually settle down.

The Key Idea: Accept Bad Moves Sometimes

Gradient descent only moves **downhill**. That's why it gets trapped.

What if we allowed *uphill* moves?

- Always accept moves that **improve** the objective.
- **Sometimes** accept moves that make the objective worse, to escape local minima.
- **Gradually** reduce the probability of accepting bad moves, so we eventually settle down.

The question is: how *exactly* should we decide when to accept a bad move?

The Key Idea: Accept Bad Moves Sometimes

Gradient descent only moves **downhill**. That's why it gets trapped.

What if we allowed *uphill* moves?

- Always accept moves that **improve** the objective.
- **Sometimes** accept moves that make the objective worse, to escape local minima.
- **Gradually** reduce the probability of accepting bad moves, so we eventually settle down.

The question is: how *exactly* should we decide when to accept a bad move?

This leads us to **Simulated Annealing**.

The Physical Analogy

Annealing in metallurgy:

1. Heat metal to a high temperature. Atoms move freely, explore many configurations.

The Physical Analogy

Annealing in metallurgy:

1. Heat metal to a high temperature. Atoms move freely, explore many configurations.
2. Cool slowly. Atoms gradually settle into a low-energy crystalline structure.

The Physical Analogy

Annealing in metallurgy:

1. Heat metal to a high temperature. Atoms move freely, explore many configurations.
2. Cool slowly. Atoms gradually settle into a low-energy crystalline structure.
3. Cool too fast $\Rightarrow$ atoms get trapped in a disordered state (brittle metal).

The Physical Analogy

Annealing in metallurgy:

1. Heat metal to a high temperature. Atoms move freely, explore many configurations.
2. Cool slowly. Atoms gradually settle into a low-energy crystalline structure.
3. Cool too fast $\Rightarrow$ atoms get trapped in a disordered state (brittle metal).

Optimization analogy:

- "Energy" = objective $f(x)$.
- "Configuration" = solution x.
- "Temperature" T = willingness to accept bad moves.

The Physical Analogy

Annealing in metallurgy:

1. Heat metal to a high temperature. Atoms move freely, explore many configurations.
2. Cool slowly. Atoms gradually settle into a low-energy crystalline structure.
3. Cool too fast $\Rightarrow$ atoms get trapped in a disordered state (brittle metal).

Optimization analogy:

- "Energy" = objective $f(x)$.
- "Configuration" = solution x.
- "Temperature" T = willingness to accept bad moves.

- High T: explore broadly.
- Low T: refine locally.
- Slowly decrease $T \to 0$.

The Metropolis Acceptance Criterion

We need a rule for “accept a bad move with some probability.” Here is the standard one (Metropolis et al., 1953):

The Metropolis Acceptance Criterion

We need a rule for “accept a bad move with some probability.” Here is the standard one (Metropolis et al., 1953):

Given current solution x with cost $f(x)$, and a candidate neighbor x':

$$\Delta = f(x') - f(x).$$

The Metropolis Acceptance Criterion

We need a rule for “accept a bad move with some probability.” Here is the standard one (Metropolis et al., 1953):

Given current solution x with cost $f(x)$, and a candidate neighbor x':

$$\Delta = f(x') - f(x).$$

- If $\Delta < 0$ (improvement): **always accept**.

The Metropolis Acceptance Criterion

We need a rule for "accept a bad move with some probability." Here is the standard one (Metropolis et al., 1953):

Given current solution x with cost $f(x)$, and a candidate neighbor x':

$$\Delta = f(x') - f(x).$$

- If $\Delta < 0$ (improvement): **always accept**.
- If $\Delta \geq 0$ (worsening): accept with probability

$$\boxed{P(\text{accept}) = e^{-\Delta/T}}$$

The Metropolis Acceptance Criterion

We need a rule for "accept a bad move with some probability." Here is the standard one (Metropolis et al., 1953):

Given current solution x with cost $f(x)$, and a candidate neighbor x':

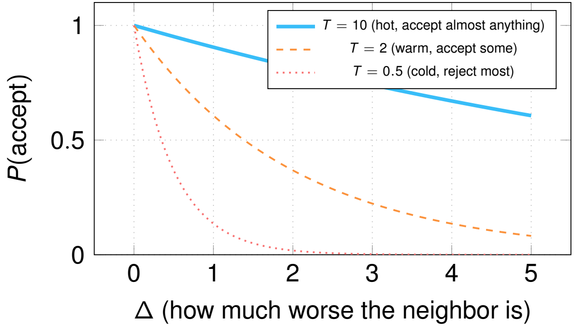
$$\Delta = f(x') - f(x).$$

- If $\Delta < 0$ (improvement): **always accept**.
- If $\Delta \geq 0$ (worsening): accept with probability

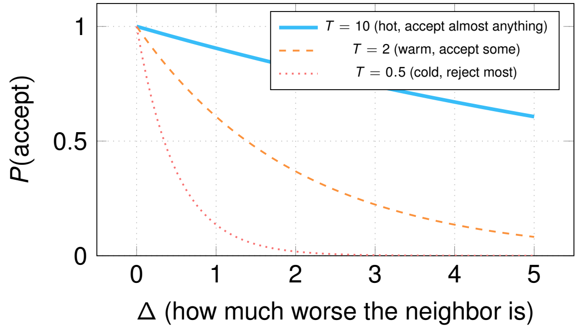
$$\boxed{P(\text{accept}) = e^{-\Delta/T}}$$

In words: small worsenings are accepted often; large worsenings are accepted rarely. At high temperature, almost everything is accepted. At low temperature, almost nothing bad is accepted.

Metropolis Criterion: The Picture



Metropolis Criterion: The Picture



As $T \to 0$, SA behaves like greedy descent. As $T \to \infty$, SA behaves like random walk.

The SA Algorithm

Simulated Annealing

1. Start with an initial solution x and temperature T_0.
2. **Repeat** for $k = 1, 2, \ldots$:
    1. Generate a random **neighbor** x' of x.
    2. Compute $\Delta = f(x') - f(x)$.
    3. If $\Delta < 0$: accept ($x \leftarrow x'$).
    4. Else: accept ($x \leftarrow x'$) with probability $e^{-\Delta/T}$.
    5. Decrease temperature: $T \leftarrow \alpha \cdot T \quad$ (for example $\alpha \approx 0.9995$).
3. Return the best solution found.

The SA Algorithm

Simulated Annealing

1. Start with an initial solution x and temperature T_0.
2. **Repeat** for $k = 1, 2, \ldots$:
    1. Generate a random **neighbor** x' of x.
    2. Compute $\Delta = f(x') - f(x)$.
    3. If $\Delta < 0$: accept $(x \leftarrow x')$.
    4. Else: accept $(x \leftarrow x')$ with probability $e^{-\Delta/T}$.
    5. Decrease temperature: $T \leftarrow \alpha \cdot T \quad$ (for example $\alpha \approx 0.9995$).
3. Return the best solution found.

That's it. The entire algorithm fits in 10 lines of code.

Temperature Schedules

How fast should we cool?

Temperature Schedules

How fast should we cool?

Geometric cooling (practical):

$$T(k) = T_0 \cdot \alpha^k$$

Typical: $\alpha \in [0.99, 0.9999]$.

Fast and easy to tune. No convergence guarantee, but works well in practice.

Temperature Schedules

How fast should we cool?

Geometric cooling (practical):

$$T(k) = T_0 \cdot \alpha^k$$

Typical: $\alpha \in [0.99, 0.9999]$.

Fast and easy to tune. No convergence guarantee, but works well in practice.

Logarithmic cooling (theoretical):

$$T(k) = \frac{c}{\ln(1 + k)}$$

Provably converges to the global optimum (Hajek, 1988).

But: logarithmic cooling is *so slow* it is essentially brute-force enumeration. Nobody uses it.

Temperature Schedules

How fast should we cool?

Geometric cooling (practical):

$$T(k) = T_0 \cdot \alpha^k$$

Typical: $\alpha \in [0.99, 0.9999]$.

Fast and easy to tune. No convergence guarantee, but works well in practice.

Logarithmic cooling (theoretical):

$$T(k) = \frac{c}{\ln(1+k)}$$

Provably converges to the global optimum (Hajek, 1988).

But: logarithmic cooling is *so slow* it is essentially brute-force enumeration. Nobody uses it.

Takeaway: use geometric cooling with $\alpha \approx 0.995$, tuned by trial and error. Run multiple independent restarts. No guarantees, but excellent results in practice.

SA on Rastrigin: The Fix

Recall: GD got stuck at a local minima. Let’s try SA on the same function.

```
import numpy as np

x = 1.7                         # same starting point as GD
best_x, best_f = x, x**2 - 10*np.cos(2*np.pi*x) + 10
T = 10.0

for k in range(200_000):
    x_new = x + np.random.randn() * 0.5          # random neighbor
    f_old = x**2 - 10*np.cos(2*np.pi*x) + 10
    f_new = x_new**2 - 10*np.cos(2*np.pi*x_new) + 10
    delta = f_new - f_old
```

SA on Rastrigin: The Fix

Recall: GD got stuck at a local minima. Let's try SA on the same function.

```python
import numpy as np

x = 1.7                          # same starting point as GD
best_x, best_f = x, x**2 - 10*np.cos(2*np.pi*x) + 10
T = 10.0

for k in range(200_000):
    x_new = x + np.random.randn() * 0.5         # random neighbor
    f_old = x**2 - 10*np.cos(2*np.pi*x) + 10
    f_new = x_new**2 - 10*np.cos(2*np.pi*x_new) + 10
    delta = f_new - f_old
    if delta < 0 or np.random.random() < np.exp(-delta / T):  # Metropolis
        x = x_new
    if f_new < best_f:
        best_x, best_f = x_new, f_new
    T *= 0.99997                                  # geometric cooling

print(f"x = {best_x:.4f}, f(x) = {best_f:.4f}")
# x = -0.0000,  f(x) = 0.0000   <-- found the global minimum!
```

SA on Rastrigin: The Fix

Recall: GD got stuck at a local minima. Let's try SA on the same function.

```python
import numpy as np

x = 1.7                           # same starting point as GD
best_x, best_f = x, x**2 - 10*np.cos(2*np.pi*x) + 10
T = 10.0

for k in range(200_000):
    x_new = x + np.random.randn() * 0.5          # random neighbor
    f_old = x**2 - 10*np.cos(2*np.pi*x) + 10
    f_new = x_new**2 - 10*np.cos(2*np.pi*x_new) + 10
    delta = f_new - f_old
    if delta < 0 or np.random.random() < np.exp(-delta / T):  # Metropolis
        x = x_new
    if f_new < best_f:
        best_x, best_f = x_new, f_new
    T *= 0.99997                                  # geometric cooling

print(f"x = {best_x:.4f}, f(x) = {best_f:.4f}")
# x = -0.0000,  f(x) = 0.0000   <-- found the global minimum!
```

Same starting point, but SA escaped the local minimum that trapped GD.

TSP: From ILP to Permutations

In Lectures 7 and 9, we modeled TSP as an **integer linear program**:

- Binary arc variables $x_{ij} \in \{0, 1\}$.
- Degree constraints: each city has exactly one incoming and one outgoing arc.
- Subtour elimination: MTZ ordering variables (Lecture 7) or DFJ lazy cuts (Lecture 9).

TSP: From ILP to Permutations

In Lectures 7 and 9, we modeled TSP as an **integer linear program**:

- Binary arc variables $x_{ij} \in \{0, 1\}$.
- Degree constraints: each city has exactly one incoming and one outgoing arc.
- Subtour elimination: MTZ ordering variables (Lecture 7) or DFJ lazy cuts (Lecture 9).

For SA, we throw all of that away.

TSP: From ILP to Permutations

In Lectures 7 and 9, we modeled TSP as an **integer linear program**:

- Binary arc variables $x_{ij} \in \{0, 1\}$.
- Degree constraints: each city has exactly one incoming and one outgoing arc.
- Subtour elimination: MTZ ordering variables (Lecture 7) or DFJ lazy cuts (Lecture 9).

For SA, we throw all of that away.

SA representation: a tour is just a **permutation** $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$.

$$f(\pi) = \sum_{i=1}^{n} d(\pi_i, \pi_{i+1}), \qquad \pi_{n+1} \equiv \pi_1.$$

TSP: From ILP to Permutations

In Lectures 7 and 9, we modeled TSP as an **integer linear program**:

- Binary arc variables $x_{ij} \in \{0, 1\}$.
- Degree constraints: each city has exactly one incoming and one outgoing arc.
- Subtour elimination: MTZ ordering variables (Lecture 7) or DFJ lazy cuts (Lecture 9).

For SA, we throw all of that away.

SA representation: a tour is just a **permutation** $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$.

$$f(\pi) = \sum_{i=1}^{n} d(\pi_i, \pi_{i+1}), \qquad \pi_{n+1} \equiv \pi_1.$$

No ILP. No Gurobi. No subtour constraints. Every permutation is automatically a valid tour.

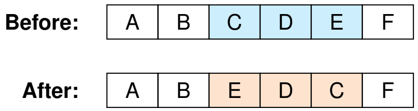
What Is a “Neighbor” for a Permutation?

SA requires generating a neighbor π' from the current tour π. Two standard choices:

What Is a “Neighbor” for a Permutation?

SA requires generating a neighbor π' from the current tour π. Two standard choices:

2-opt: pick two positions $i < j$, reverse the segment between them.

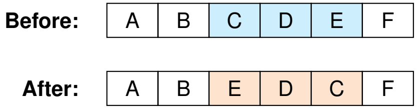


segment [C,D,E] reversed to [E,D,C]

What Is a “Neighbor” for a Permutation?

SA requires generating a neighbor π' from the current tour π. Two standard choices:

2-opt: pick two positions $i < j$, reverse the segment between them.



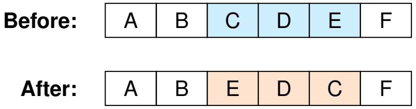
segment [C,D,E] reversed to [E,D,C]

Key observation: It is the most widely used neighborhood for TSP.

What Is a “Neighbor” for a Permutation?

SA requires generating a neighbor π' from the current tour π. Two standard choices:

2-opt: pick two positions $i < j$, reverse the segment between them.



segment [C,D,E] reversed to [E,D,C]

Key observation: It is the most widely used neighborhood for TSP.

An alternative is **swap**: exchange the positions of two random cities. Simpler, but less effective.

SA for TSP: Full Implementation

```python
import numpy as np

def tsp_sa(dist, T0=100.0, alpha=0.9995, n_iter=500_000):
    n = len(dist)
    tour = list(range(n))
    np.random.shuffle(tour)
    cost = sum(dist[tour[i]][tour[(i+1) % n]] for i in range(n))
    best_tour, best_cost = tour[:], cost
    T = T0
```

SA for TSP: Full Implementation

```
import numpy as np

def tsp_sa(dist, T0=100.0, alpha=0.9995, n_iter=500_000):
    n = len(dist)
    tour = list(range(n))
    np.random.shuffle(tour)
    cost = sum(dist[tour[i]][tour[(i+1) % n]] for i in range(n))
    best_tour, best_cost = tour[:], cost
    T = T0
    for k in range(n_iter):
        i, j = sorted(np.random.randint(0, n, size=2))  # 2-opt: pick segment
        if i == j: continue
        new_tour = tour[:i] + tour[i:j+1][::-1] + tour[j+1:]  # reverse it
        new_cost = sum(dist[new_tour[a]][new_tour[(a+1)%n]] for a in range(n))
        delta = new_cost - cost
```

SA for TSP: Full Implementation

```python
import numpy as np

def tsp_sa(dist, T0=100.0, alpha=0.9995, n_iter=500_000):
    n = len(dist)
    tour = list(range(n))
    np.random.shuffle(tour)
    cost = sum(dist[tour[i]][tour[(i+1) % n]] for i in range(n))
    best_tour, best_cost = tour[:], cost
    T = T0
    for k in range(n_iter):
        i, j = sorted(np.random.randint(0, n, size=2))  # 2-opt: pick segment
        if i == j: continue
        new_tour = tour[:i] + tour[i:j+1][::-1] + tour[j+1:]  # reverse it
        new_cost = sum(dist[new_tour[a]][new_tour[(a+1)%n]] for a in range(n))
        delta = new_cost - cost
        if delta < 0 or np.random.random() < np.exp(-delta / T):   # Metropolis
            tour, cost = new_tour, new_cost
            if cost < best_cost:
                best_tour, best_cost = tour[:], cost
        T *= alpha
    return best_tour, best_cost
```

SA for TSP: Full Implementation

```python
import numpy as np

def tsp_sa(dist, T0=100.0, alpha=0.9995, n_iter=500_000):
    n = len(dist)
    tour = list(range(n))
    np.random.shuffle(tour)
    cost = sum(dist[tour[i]][tour[(i+1) % n]] for i in range(n))
    best_tour, best_cost = tour[:], cost
    T = T0
    for k in range(n_iter):
        i, j = sorted(np.random.randint(0, n, size=2))  # 2-opt: pick segment
        if i == j: continue
        new_tour = tour[:i] + tour[i:j+1][::-1] + tour[j+1:]  # reverse it
        new_cost = sum(dist[new_tour[a]][new_tour[(a+1)%n]] for a in range(n))
        delta = new_cost - cost
        if delta < 0 or np.random.random() < np.exp(-delta / T):   # Metropolis
            tour, cost = new_tour, new_cost
            if cost < best_cost:
                best_tour, best_cost = tour[:], cost
        T *= alpha
    return best_tour, best_cost
```

20 lines of Python. No solver, no callbacks, no subtour constraints. Just a loop.

SA vs. B&B for TSP: The Tradeoff

	B&B + Lazy Cuts (Lec. 9)	**Simulated Annealing**
Solution quality	Provably optimal	Good (within 2–5%)
Scalability	$n \lesssim 200$	$n \sim 10{,}000+$
Running time	Exponential worst-case	User-controlled
Implementation	ILP + callbacks	20-line loop
Needs Gurobi?	Yes	No

SA vs. B&B for TSP: The Tradeoff

	B&B + Lazy Cuts (Lec. 9)	**Simulated Annealing**
Solution quality	Provably optimal	Good (within 2–5%)
Scalability	$n \lesssim 200$	$n \sim 10{,}000+$
Running time	Exponential worst-case	User-controlled
Implementation	ILP + callbacks	20-line loop
Needs Gurobi?	Yes	No

The deal: SA gives up the *proof* of optimality in exchange for being able to handle problems that B&B cannot touch. For TSP, SA tours are typically within 2–5% of optimal, which is good enough for many applications.

SA vs. B&B for TSP: The Tradeoff

	B&B + Lazy Cuts (Lec. 9)	**Simulated Annealing**
Solution quality	Provably optimal	Good (within 2–5%)
Scalability	$n \lesssim 200$	$n \sim 10{,}000+$
Running time	Exponential worst-case	User-controlled
Implementation	ILP + callbacks	20-line loop
Needs Gurobi?	Yes	No

The deal: SA gives up the *proof* of optimality in exchange for being able to handle problems that B&B cannot touch. For TSP, SA tours are typically within 2–5% of optimal, which is good enough for many applications.

Rule of thumb: if *n* is small enough for B&B and you need a certificate, use B&B. If you need a good solution on a large instance in bounded time, use SA.

From One Solution to a Population

SA maintains a **single** solution and perturbs it. This works well, but it explores one path at a time.

From One Solution to a Population

SA maintains a **single** solution and perturbs it. This works well, but it explores one path at a time.

Question: what if we maintained *many* solutions simultaneously, and let the good ones “mate”?

From One Solution to a Population

SA maintains a **single** solution and perturbs it. This works well, but it explores one path at a time.

Question: what if we maintained *many* solutions simultaneously, and let the good ones “mate”?

This is the idea behind **Genetic Algorithms** (GA). The analogy is evolution. Two parent solutions combine their genetic material to produce offspring.

From One Solution to a Population

SA maintains a **single** solution and perturbs it. This works well, but it explores one path at a time.

Question: what if we maintained *many* solutions simultaneously, and let the good ones “mate”?

This is the idea behind **Genetic Algorithms** (GA). The analogy is evolution. Two parent solutions combine their genetic material to produce offspring.

I trust I don’t need to explain to UIUC students how reproduction works.

From One Solution to a Population

SA maintains a **single** solution and perturbs it. This works well, but it explores one path at a time.

Question: what if we maintained *many* solutions simultaneously, and let the good ones “mate”?

This is the idea behind **Genetic Algorithms** (GA). The analogy is evolution. Two parent solutions combine their genetic material to produce offspring.

I trust I don’t need to explain to UIUC students how reproduction works.

- A **population** of candidate solutions. The fittest individuals are more likely to reproduce.

From One Solution to a Population

SA maintains a **single** solution and perturbs it. This works well, but it explores one path at a time.

Question: what if we maintained *many* solutions simultaneously, and let the good ones "mate"?

This is the idea behind **Genetic Algorithms** (GA). The analogy is evolution. Two parent solutions combine their genetic material to produce offspring.

I trust I don't need to explain to UIUC students how reproduction works.

- A **population** of candidate solutions. The fittest individuals are more likely to reproduce.
- Offspring inherit traits from **two parents** (crossover).

From One Solution to a Population

SA maintains a **single** solution and perturbs it. This works well, but it explores one path at a time.

Question: what if we maintained *many* solutions simultaneously, and let the good ones “mate”?

This is the idea behind **Genetic Algorithms** (GA). The analogy is evolution. Two parent solutions combine their genetic material to produce offspring.

I trust I don’t need to explain to UIUC students how reproduction works.

- A **population** of candidate solutions. The fittest individuals are more likely to reproduce.
- Offspring inherit traits from **two parents** (crossover).
- Random **mutations** maintain diversity.

From One Solution to a Population

SA maintains a **single** solution and perturbs it. This works well, but it explores one path at a time.

Question: what if we maintained *many* solutions simultaneously, and let the good ones “mate”?

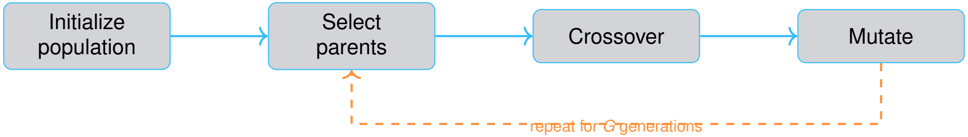
This is the idea behind **Genetic Algorithms** (GA). The analogy is evolution. Two parent solutions combine their genetic material to produce offspring.

I trust I don’t need to explain to UIUC students how reproduction works.

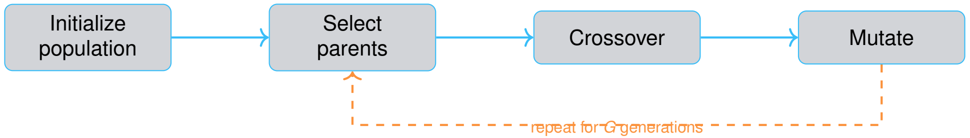
- A **population** of candidate solutions. The fittest individuals are more likely to reproduce.
- Offspring inherit traits from **two parents** (crossover).
- Random **mutations** maintain diversity.

Key insight: if two decent solutions have different “good parts,” combining them might produce something better than either parent.

GA: The Pipeline

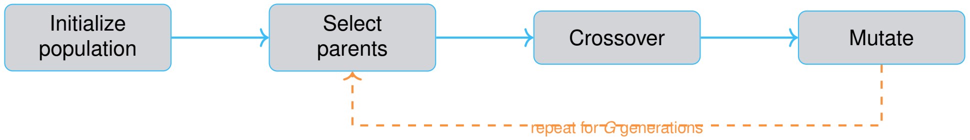


GA: The Pipeline



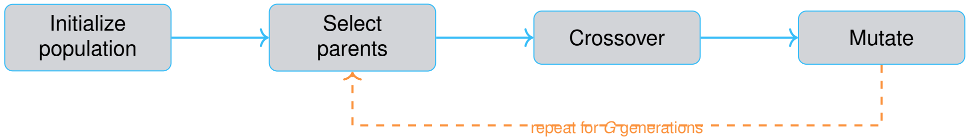
1. **Selection:** pick parents from the current population. Fitter = more likely to be picked. Common method: *tournament selection* (pick k random individuals, take the best).

GA: The Pipeline



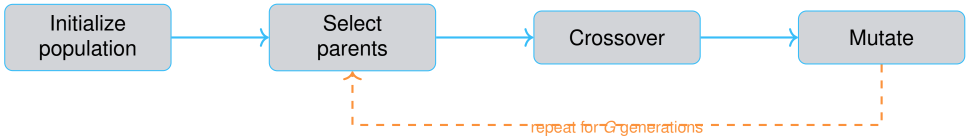
1. **Selection:** pick parents from the current population. Fitter = more likely to be picked. Common method: *tournament selection* (pick k random individuals, take the best).
2. **Crossover:** combine two parents to produce a child. The tricky part: must respect the problem structure.

GA: The Pipeline



1. **Selection:** pick parents from the current population. Fitter = more likely to be picked. Common method: *tournament selection* (pick k random individuals, take the best).
2. **Crossover:** combine two parents to produce a child. The tricky part: must respect the problem structure.
3. **Mutation:** small random perturbation on the child (swap two cities, flip a bit, add noise).

GA: The Pipeline



1. **Selection:** pick parents from the current population. Fitter = more likely to be picked. Common method: *tournament selection* (pick k random individuals, take the best).
2. **Crossover:** combine two parents to produce a child. The tricky part: must respect the problem structure.
3. **Mutation:** small random perturbation on the child (swap two cities, flip a bit, add noise).
4. **Elitism:** always keep the best few individuals from the previous generation.

Crossover for Continuous Problems

When solutions are vectors $x \in \mathbb{R}^n$, crossover is simple:

Crossover for Continuous Problems

When solutions are vectors $x \in \mathbb{R}^n$, crossover is simple:

Blend crossover: given parents $p_1, p_2 \in \mathbb{R}^n$, produce child:

$$\text{child} = \beta\, p_1 + (1 - \beta)\, p_2, \quad \beta \sim \text{Uniform}(0, 1).$$

Crossover for Continuous Problems

When solutions are vectors $x \in \mathbb{R}^n$, crossover is simple:

Blend crossover: given parents $p_1, p_2 \in \mathbb{R}^n$, produce child:

$$\text{child} = \beta\, p_1 + (1 - \beta)\, p_2, \quad \beta \sim \text{Uniform}(0, 1).$$

In words: the child is a random weighted average of the two parents. Simple and effective.

Crossover for Continuous Problems

When solutions are vectors $x \in \mathbb{R}^n$, crossover is simple:

Blend crossover: given parents $p_1, p_2 \in \mathbb{R}^n$, produce child:

$$\text{child} = \beta\, p_1 + (1 - \beta)\, p_2, \quad \beta \sim \text{Uniform}(0, 1).$$

In words: the child is a random weighted average of the two parents. Simple and effective.

Mutation: add Gaussian noise.

$$\text{child}_{\text{mutated}} = \text{child} + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2 I).$$

Crossover for Continuous Problems

When solutions are vectors $x \in \mathbb{R}^n$, crossover is simple:

Blend crossover: given parents $p_1, p_2 \in \mathbb{R}^n$, produce child:

$$\text{child} = \beta\, p_1 + (1-\beta)\, p_2, \quad \beta \sim \text{Uniform}(0,1).$$

In words: the child is a random weighted average of the two parents. Simple and effective.

Mutation: add Gaussian noise.

$$\text{child}_{\text{mutated}} = \text{child} + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2 I).$$

But: for **combinatorial** problems, we can't just average two permutations. The average of $(1,2,3,4,5)$ and $(5,4,3,2,1)$ is not a valid tour. We need something smarter.

The Problem with Naive Crossover on Permutations

Suppose we try “cut and paste” crossover on two TSP tours:

The Problem with Naive Crossover on Permutations

Suppose we try “cut and paste” crossover on two TSP tours:

P1:	3	7	1	4	2	5
P2:	2	5	4	6	3	1

The child is **not a valid permutation**. We need a crossover operator that preserves the permutation property.

The Problem with Naive Crossover on Permutations

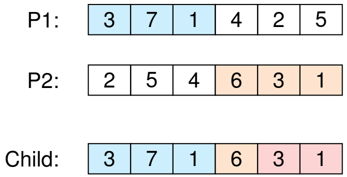
Suppose we try “cut and paste” crossover on two TSP tours:

P1:	3	7	1	4	2	5
P2:	2	5	4	6	3	1
Child:	3	7	1	6	3	1

3 and 1 appear twice! Cities 2,5 missing!

The Problem with Naive Crossover on Permutations

Suppose we try “cut and paste” crossover on two TSP tours:



3 and 1 appear twice! Cities 2,5 missing!

The child is **not a valid permutation**. We need a crossover operator that preserves the permutation property.

Order Crossover (OX)

The standard fix for permutation-based GAs:

Order Crossover (OX)

The standard fix for permutation-based GAs:

1. Pick two random cut points i, j in Parent 1.

Order Crossover (OX)

The standard fix for permutation-based GAs:

1. Pick two random cut points i, j in Parent 1.
2. Copy the segment $[i..j]$ from Parent 1 directly into the child.

Order Crossover (OX)

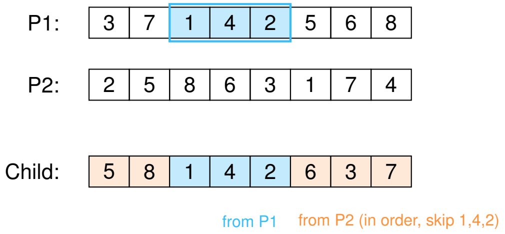
The standard fix for permutation-based GAs:

1. Pick two random cut points i, j in Parent 1.
2. Copy the segment $[i..j]$ from Parent 1 directly into the child.
3. Fill the remaining positions with cities from Parent 2, *in the order they appear*, skipping any city already used.

Order Crossover (OX)

The standard fix for permutation-based GAs:

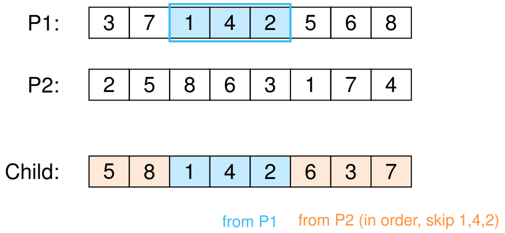
1. Pick two random cut points i, j in Parent 1.
2. Copy the segment $[i..j]$ from Parent 1 directly into the child.
3. Fill the remaining positions with cities from Parent 2, *in the order they appear*, skipping any city already used.



Order Crossover (OX)

The standard fix for permutation-based GAs:

1. Pick two random cut points i, j in Parent 1.
2. Copy the segment $[i..j]$ from Parent 1 directly into the child.
3. Fill the remaining positions with cities from Parent 2, *in the order they appear*, skipping any city already used.



The child is a valid permutation. Cities $\{1, 4, 2\}$ come from P1; the rest are filled from P2's ordering.

Order Crossover: Code

```python
def order_crossover(p1, p2):
    n = len(p1)
    i, j = sorted(np.random.choice(n, 2, replace=False))
    child = [None] * n
    child[i:j+1] = p1[i:j+1]                        # copy segment from P1
    used = set(child[i:j+1])
    fill = [c for c in p2 if c not in used]          # P2 cities, in order, skip used
    pos = 0
    for k in range(n):
        if child[k] is None:
            child[k] = fill[pos]
            pos += 1
    return child
```

Order Crossover: Code

```
def order_crossover(p1, p2):
    n = len(p1)
    i, j = sorted(np.random.choice(n, 2, replace=False))
    child = [None] * n
    child[i:j+1] = p1[i:j+1]                    # copy segment from P1
    used = set(child[i:j+1])
    fill = [c for c in p2 if c not in used]        # P2 cities, in order, skip used
    pos = 0
    for k in range(n):
        if child[k] is None:
            child[k] = fill[pos]
            pos += 1
    return child
```

Observation: the child inherits *local structure* (a contiguous segment) from P1, and *global ordering* from P2. If both parents have good sub-tours, the child has a chance of combining them.

GA for TSP: Putting It All Together

```python
def tsp_ga(dist, pop_size=100, n_gen=1000, mut_rate=0.02):
    n = len(dist)
    pop = [list(np.random.permutation(n)) for _ in range(pop_size)]
    cost = lambda t: sum(dist[t[i]][t[(i+1) % n]] for i in range(n))
```

GA for TSP: Putting It All Together

```
def tsp_ga(dist, pop_size=100, n_gen=1000, mut_rate=0.02):
    n = len(dist)
    pop = [list(np.random.permutation(n)) for _ in range(pop_size)]
    cost = lambda t: sum(dist[t[i]][t[(i+1) % n]] for i in range(n))

    for gen in range(n_gen):
        pop.sort(key=cost)
        new_pop = pop[:10]                          # elitism: keep top 10
        while len(new_pop) < pop_size:
            # tournament selection: pick 2 from top 20
            p1, p2 = [pop[np.random.randint(20)] for _ in range(2)]
            child = order_crossover(p1, p2)         # crossover
```

GA for TSP: Putting It All Together

```python
def tsp_ga(dist, pop_size=100, n_gen=1000, mut_rate=0.02):
    n = len(dist)
    pop = [list(np.random.permutation(n)) for _ in range(pop_size)]
    cost = lambda t: sum(dist[t[i]][t[(i+1) % n]] for i in range(n))

    for gen in range(n_gen):
        pop.sort(key=cost)
        new_pop = pop[:10]                         # elitism: keep top 10
        while len(new_pop) < pop_size:
            # tournament selection: pick 2 from top 20
            p1, p2 = [pop[np.random.randint(20)] for _ in range(2)]
            child = order_crossover(p1, p2)        # crossover

            if np.random.random() < mut_rate:      # mutation: swap two cities
                a, b = np.random.choice(n, 2, replace=False)
                child[a], child[b] = child[b], child[a]
            new_pop.append(child)
        pop = new_pop
    pop.sort(key=cost)
    return pop[0], cost(pop[0])
```

SA vs. GA: When to Use Which?

	Simulated Annealing	**Genetic Algorithm**
# of solutions	One	Population
Exploration source	Temperature	Crossover + mutation
Parameters to tune	T_0, α, iterations	Pop size, rates, selection
Parallelism	Limited	Natural (eval pop. in parallel)

SA vs. GA: When to Use Which?

	Simulated Annealing	**Genetic Algorithm**
# of solutions	One	Population
Exploration source	Temperature	Crossover + mutation
Parameters to tune	T_0, α, iterations	Pop size, rates, selection
Parallelism	Limited	Natural (eval pop. in parallel)

In practice:

- SA is often the **first thing to try**. Simpler, fewer parameters, surprisingly effective.

SA vs. GA: When to Use Which?

	Simulated Annealing	**Genetic Algorithm**
# of solutions	One	Population
Exploration source	Temperature	Crossover + mutation
Parameters to tune	T_0, α, iterations	Pop size, rates, selection
Parallelism	Limited	Natural (eval pop. in parallel)

In practice:

- SA is often the **first thing to try**. Simpler, fewer parameters, surprisingly effective.
- GA shines when good sub-solutions can be meaningfully **recombined**.

SA vs. GA: When to Use Which?

	Simulated Annealing	**Genetic Algorithm**
# of solutions	One	Population
Exploration source	Temperature	Crossover + mutation
Parameters to tune	T_0, α, iterations	Pop size, rates, selection
Parallelism	Limited	Natural (eval pop. in parallel)

In practice:

- SA is often the **first thing to try**. Simpler, fewer parameters, surprisingly effective.
- GA shines when good sub-solutions can be meaningfully **recombined**.
- Both are dramatically outperformed by **problem-specific heuristics** when available (e.g., Lin–Kernighan for TSP). Metaheuristics are general-purpose. That's their strength and their weakness.

Why Hybrid?

Metaheuristics are good at **global exploration** but bad at **local precision**.

Why Hybrid?

Metaheuristics are good at **global exploration** but bad at **local precision**.
Gradient descent is good at **local precision** but bad at **escaping traps**.

Why Hybrid?

Metaheuristics are good at **global exploration** but bad at **local precision**.
Gradient descent is good at **local precision** but bad at **escaping traps**.
MIP solvers are **exact** but need a good **initial bound** to prune efficiently.

Why Hybrid?

Metaheuristics are good at **global exploration** but bad at **local precision**.
Gradient descent is good at **local precision** but bad at **escaping traps**.
MIP solvers are **exact** but need a good **initial bound** to prune efficiently.

The question is: can we combine them?

Why Hybrid?

Metaheuristics are good at **global exploration** but bad at **local precision**.
Gradient descent is good at **local precision** but bad at **escaping traps**.
MIP solvers are **exact** but need a good **initial bound** to prune efficiently.

The question is: can we combine them?

Two natural hybrids:

1. **SA $\rightarrow$ GD:** metaheuristic finds the right basin, GD refines to high precision.
2. **SA $\rightarrow$ Gurobi:** metaheuristic provides a strong incumbent, Gurobi proves optimality faster.

Hybrid 1: SA + GD on Rastrigin

Phase 1: SA for global exploration (find the right basin).
Phase 2: gradient descent for local refinement (converge precisely).

```python
# Phase 1: SA (rough search)
x = np.random.uniform(-5, 5, size=2)
T = 10.0
for k in range(200_000):
    x_new = x + np.random.randn(2) * 0.5
    delta = rastrigin(x_new) - rastrigin(x)
    if delta < 0 or np.random.random() < np.exp(-delta / T):
        x = x_new
    T *= 0.99997
```

Hybrid 1: SA + GD on Rastrigin

Phase 1: SA for global exploration (find the right basin).
Phase 2: gradient descent for local refinement (converge precisely).

```
# Phase 1: SA (rough search)
x = np.random.uniform(-5, 5, size=2)
T = 10.0
for k in range(200_000):
    x_new = x + np.random.randn(2) * 0.5
    delta = rastrigin(x_new) - rastrigin(x)
    if delta < 0 or np.random.random() < np.exp(-delta / T):
        x = x_new
    T *= 0.99997

# Phase 2: GD refinement (starting from SA's answer)    <-- only new thing
x_t = torch.tensor(x, dtype=torch.float64, requires_grad=True)
opt = torch.optim.Adam([x_t], lr=0.01)
for _ in range(1000):
    loss = 10*2 + (x_t**2 - 10*torch.cos(2*torch.pi*x_t)).sum()
    opt.zero_grad(); loss.backward(); opt.step()
print(x_t.detach().numpy())   # [~0.0, ~0.0]
```

Hybrid 1: SA + GD on Rastrigin

Phase 1: SA for global exploration (find the right basin).
Phase 2: gradient descent for local refinement (converge precisely).

```
# Phase 1: SA (rough search)
x = np.random.uniform(-5, 5, size=2)
T = 10.0
for k in range(200_000):
    x_new = x + np.random.randn(2) * 0.5
    delta = rastrigin(x_new) - rastrigin(x)
    if delta < 0 or np.random.random() < np.exp(-delta / T):
        x = x_new
    T *= 0.99997

# Phase 2: GD refinement (starting from SA's answer)    <-- only new thing
x_t = torch.tensor(x, dtype=torch.float64, requires_grad=True)
opt = torch.optim.Adam([x_t], lr=0.01)
for _ in range(1000):
    loss = 10*2 + (x_t**2 - 10*torch.cos(2*torch.pi*x_t)).sum()
    opt.zero_grad(); loss.backward(); opt.step()
print(x_t.detach().numpy())   # [~0.0, ~0.0]
```

SA gets us to the right neighborhood. GD polishes the answer.

Hybrid 2: SA Warm-Starts for Gurobi

Recall from Lecture 5: B&B prunes branches using the **incumbent** (best integer solution found so far). A better incumbent $\Rightarrow$ more pruning $\Rightarrow$ smaller search tree.

Hybrid 2: SA Warm-Starts for Gurobi

Recall from Lecture 5: B&B prunes branches using the **incumbent** (best integer solution found so far). A better incumbent $\Rightarrow$ more pruning $\Rightarrow$ smaller search tree.

The idea: run SA for a few seconds to find a good (but not proven optimal) tour. Then hand it to Gurobi as a starting point.

Hybrid 2: SA Warm-Starts for Gurobi

Recall from Lecture 5: B&B prunes branches using the **incumbent** (best integer solution found so far). A better incumbent $\Rightarrow$ more pruning $\Rightarrow$ smaller search tree.

The idea: run SA for a few seconds to find a good (but not proven optimal) tour. Then hand it to Gurobi as a starting point.

"Hey Gurobi, I have this TSP model with lazy cuts, just like Lecture 9. But before you start, here's a tour I found with SA. It costs 1,847. Start from here. You don't have to search the whole tree, just prove you can't do better than 1,847 (or find something that does)."

Hybrid 2: SA Warm-Starts for Gurobi

Recall from Lecture 5: B&B prunes branches using the **incumbent** (best integer solution found so far). A better incumbent ⇒ more pruning ⇒ smaller search tree.

The idea: run SA for a few seconds to find a good (but not proven optimal) tour. Then hand it to Gurobi as a starting point.

"Hey Gurobi, I have this TSP model with lazy cuts, just like Lecture 9. But before you start, here's a tour I found with SA. It costs 1,847. Start from here. You don't have to search the whole tree, just prove you can't do better than 1,847 (or find something that does)."

Result: Gurobi still proves optimality, but the search tree can be dramatically smaller because it starts with a strong bound.

Warm-Starting Gurobi: Code

```python
# Phase 1: SA finds a good tour (seconds)
sa_tour, sa_cost = tsp_sa(dist, n_iter=500_000)

# Phase 2: Set up ILP (same as Lecture 9)
m = gp.Model("TSP")
x = {}
for i in range(n):
    for j in range(n):
        if i != j:
            x[i,j] = m.addVar(vtype='B', name=f"x_{i}_{j}")
# ... degree constraints, lazy callbacks, etc. (same as Lecture 9) ...
```

Warm-Starting Gurobi: Code

```
# Phase 1: SA finds a good tour (seconds)
sa_tour, sa_cost = tsp_sa(dist, n_iter=500_000)

# Phase 2: Set up ILP (same as Lecture 9)
m = gp.Model("TSP")
x = {}
for i in range(n):
    for j in range(n):
        if i != j:
            x[i,j] = m.addVar(vtype='B', name=f"x_{i}_{j}")
# ... degree constraints, lazy callbacks, etc. (same as Lecture 9) ...

# New!! Feed SA solution as initial incumbent
for k in range(n):
    i = sa_tour[k]
    j = sa_tour[(k + 1) % n]
    x[i, j].Start = 1.0                  # hint: this arc is in the tour

m.optimize()    # Gurobi starts with sa_cost as initial UB --> prunes faster
```

Warm-Starting Gurobi: Code

```
# Phase 1: SA finds a good tour (seconds)
sa_tour, sa_cost = tsp_sa(dist, n_iter=500_000)

# Phase 2: Set up ILP (same as Lecture 9)
m = gp.Model("TSP")
x = {}
for i in range(n):
    for j in range(n):
        if i != j:
            x[i,j] = m.addVar(vtype='B', name=f"x_{i}_{j}")
# ... degree constraints, lazy callbacks, etc. (same as Lecture 9) ...

# New!! Feed SA solution as initial incumbent
for k in range(n):
    i = sa_tour[k]
    j = sa_tour[(k + 1) % n]
    x[i, j].Start = 1.0                  # hint: this arc is in the tour

m.optimize()    # Gurobi starts with sa_cost as initial UB --> prunes faster
```

The `.Start` attribute is the hint mechanism. Gurobi verifies feasibility and uses the solution as its initial incumbent if valid.