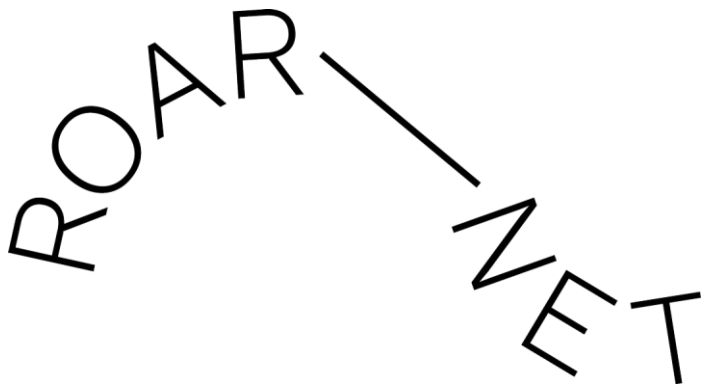


Distance-Based Search Operators

Alberto Moraglio
University of Exeter, UK
a.moraglio@exeter.ac.uk

cost
EUROPEAN COOPERATION IN SCIENCE & TECHNOLOGY

Funded by the European Union

Instructor Information

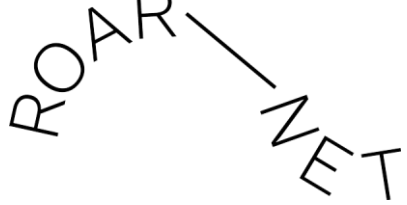
Dr. Alberto Moraglio is a Senior Lecturer in Computer Science at the University of Exeter, UK. He is the founder of the Geometric Theory of Evolutionary Algorithms, which unifies Evolutionary Algorithms across representations and has been used for the principled design of new successful search algorithms and for their rigorous theoretical analysis. He has been active in Evolutionary Computation research for the last 2 decades with a substantial publication record in the area.

Learning Objectives

By the end of this lecture, you will:

- **Understand** how distance metrics unify search operators across representations
- **Connect** evolutionary algorithms to neighborhood search and local search
- **Apply** geometric interpretations to tabu search
- **Understand** the distance-based API for practical optimization problems

Part I: Distance-Based Evolutionary Algorithms



What are Evolutionary Algorithms?

Definition: Optimization algorithms inspired by biological evolution

Biological Inspiration:

- **Population** of candidate solutions (individuals)
- **Variation** through crossover and mutation
- **Selection** pressure favors fitter individuals
- **Evolution** over multiple generations

Key Insight: Good solutions can be evolved rather than calculated

Evolutionary Algorithm Flavours

Many instantiations exist:

- Genetic Algorithms (GA)
- Evolution Strategies (ES)
- Genetic Programming (GP)
- Evolutionary Programming (EP)

Key insight: They differ substantially only in:

- **Underlying representation** (binary, real, permutations, trees)
- **Search operators** (mutation, crossover)

Same algorithmic structure: selection → variation → replacement

Question: Can We Unify These Approaches?

Challenge:

- Each representation seems to need its own "special" operators
- Binary strings ≠ Real vectors ≠ Permutations
- Different operators seem completely unrelated

Today's answer: Yes! Through distance-based operators

Concrete Examples: Binary Strings

Representation: [1, 0, 1, 1, 0, 0, 1, 0]

Mutation Operations:

- **Bit-flip:** Randomly flip bits
- Example: [1,0,1,1,0,0,1,0] → [1,0,0,1,0,0,1,0]

Crossover Operators: Binary Strings

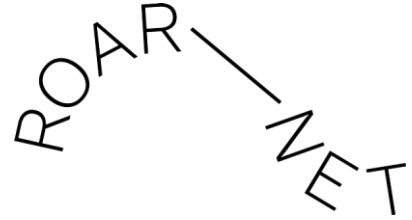
One-point crossover:

```
Parent 1: [1,0,1,1,0,0,1,0]
Parent 2: [0,1,0,0,1,1,0,1]
          ──────────┤ (cut point)

Offspring:[1,0,1,1,1,1,0,1]
```

Uniform crossover:

```
Parent 1: [1,0,1,1,0,0,1,0]
Parent 2: [0,1,0,0,1,1,0,1]
Mask:     [1,0,1,0,1,0,1,0]
(random)
Offspring:[1,1,1,0,0,1,1,1]
```



Concrete Examples: Real Vectors

Representation: [2.5, -1.3, 0.8, 4.2]

Mutation Operations:

- **Gaussian perturbation:** Add random noise
- Example: [2.5, -1.3, 0.8, 4.2] → [2.7, -1.1, 0.9, 4.0]

Crossover Operations:

- **Arithmetic recombination:** offspring = α·parent1 + (1-α)·parent2
- **Discrete recombination:** Inherit each component from either parent (like uniform crossover)

Concrete Examples: Permutations

Representation: [3, 1, 4, 2, 5] (e.g., TSP tour)

Swap mutation:

```
Original: [3,1,4,2,5]
Mutated:  [3,2,4,1,5] (swap positions 1 and 3)
```

Insertion mutation:

```
Original: [3,1,4,2,5]
Step 1:   [3,1,_,2,5] (remove 4 from position 2)
Step 2:   [3,1,2,5,4] (insert 4 at position 4)
```

ROAR NET

Crossover Operators: Permutations

Order Crossover (OX):

```
Parent 1: [3,1,4,2,5]
Parent 2: [2,5,1,3,4]
Step 1:   [_,_,4,2,5]
(substring from P1)
Step 2:   [1,3,4,2,5]
(fill from P2 in order)
```

Cycle Crossover:

```
Parent 1: [3,1,4,2,5]
Parent 2: [2,5,1,3,4]
```

Cycle 1: 3→2→3

Cycle 2: 1→5→4→1

Offspring: [3,5,1,2,4]

(take P1 for cycle 1, P2 for cycle 2)

Observation

Each representation has its **own "special" operators**

Questions:

- Are these really fundamentally different?
- Is there a deeper unifying principle?
- Can we define operators that work across representations?

Next: The geometric answer...

Geometric Unification via Distance Metrics

The Unifying Framework:

- **Geometric Mutation:** offspring ∈ ball(parent, radius) (**near**)
- **Geometric Crossover:** offspring ∈ segment(parent1, parent2) (**between**)

Formal definitions for any distance d:

- **Ball:** ball(center, radius) = {x | d(center, x) ≤ radius}
- **Segment**: segment(a, b) = {x | d(a, x) + d(x, b) = d(a, b)}

Key insight: Same geometric principles, different distance metrics

ROAR-NET

Geometric Operators: Real Vectors

Distance metric: Euclidean distance

Geometric Mutation:

- ball(parent, radius) = hypersphere around (**near**) parent
- **Gaussian mutation** ≈ sampling from Euclidean ball

Geometric Crossover:

- segment(parent1, parent2) = line segment **between** parents
- **Arithmetic recombination** = uniform sampling in Euclidean segment

Geometric Operators: Binary Strings

Distance metric: Hamming distance (number of differing bits)

Geometric Mutation:

- ball(parent, radius) = all strings within Hamming distance radius
- **Bit-flip mutation** = sampling from Hamming ball of radius 1

- **Example:** parent = [1,0,1,0], radius 1

Ball contains strings **near** the parent: [0,0,1,0], [1,1,1,0], [1,0,0,0], [1,0,1,1], [1,0,1,0]

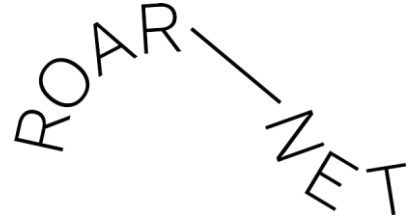
Geometric Crossover: Binary Strings

Geometric Crossover:

- segment(parent1, parent2) = all binary strings that lie **between** the two parents
- **Uniform crossover** = uniform sampling from Hamming segment

One-point crossover example:
Parent 1: [1,0,1,1,0,0,1,0]
Parent 2: [0,1,0,0,1,1,0,1]
Offspring:[1,0,1,1,1,1,0,1]

Distance check: HD(P1,O) + HD(O,P2) = HD(P1,P2): 2 + 4 = 6
Key insight: Valid crossover offspring lie between their parents!



Geometric Operators: Permutations

Distance metric: Swap distance (minimum number of swaps)

Geometric Mutation:

- ball(parent, radius) = all permutations within radius swaps
- **Swap mutation** = sampling from swap ball of radius 1

- **Example:** parent = [1,2,3,4], radius 1

Ball contains permutations **near** the parent: [2,1,3,4], [1,3,2,4], [1,2,4,3], [4,2,3,1], [1,4,3,2], [3,2,1,4], [1,2,3,4]

Geometric Crossover: Permutations

Geometric Crossover:

- segment(parent1, parent2) = all permutations that lie **between** the two parents
- **Geometric crossovers** = sampling from swap segment

Order Crossover (OX) example:
Parent 1: [1,2,3,4,5]
Parent 2: [3,1,5,2,4]
Offspring:[1,2,3,5,4]

Distance check: SD(P1,O) + SD(O,P2) = SD(P1,P2): 1 + 3 = 4
Key insight: OX produces offspring between parents in permutation space!

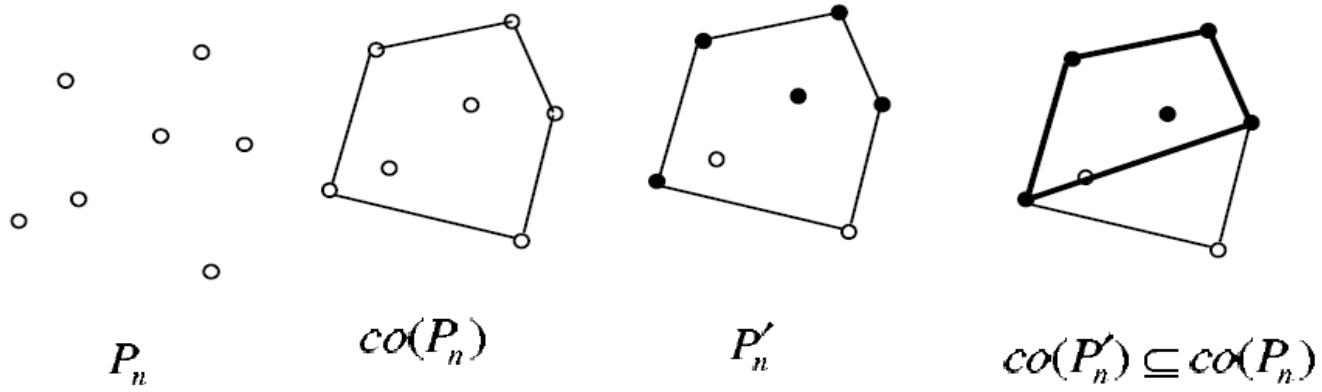
Implication: All* EAs Do Convex Search

Key insight: Geometric crossover always produces offspring "between" parents

Convex search property:

- Population remains in **convex hull** of initial population + mutations
- Search is inherently **conservative** and **exploitative**
- **Exploration** comes from mutation (expanding the hull)

Convex Evolutionary Search



Part II: Evolutionary Algorithms as Neighborhood Search

ROAR-NET

Fundamental Concepts

Neighborhood: Set of solutions reachable by a single move

Neighborhood Structure: Graph where nodes = solutions, edges = moves

- Defines which solutions are "adjacent" to each other
- **Determined by:** all local adjacency relations of all solutions together
- Creates the overall topology of the search space

Neighborhood Search: Algorithm that moves from solution to solution

- **Local Search:** Move to better neighbor (hill-climbing)
- **Random Walk:** Move to random neighbor
- **Guided Search:** Move based on some strategy (e.g., tabu search)

Connecting EAs to Local Search

Fundamental connection: EAs and local search use the same neighborhood structures

Two key relationships:

- **Mutation vs Local Search**
- **Crossover vs Path-Relinking**

Neighborhood Structure Connection

Now I'll reveal what "between" and "close" actually mean algorithmically:

Metric spaces ↔ Neighborhood structures:

- Distance = shortest path length in neighborhood graph
- "Between" = lying on shortest paths
- "Close" = reachable by few moves

Key insight: geometric segments are actually shortest paths through the neighborhood graph defined by the distance metric

Unification: EAs use the same neighborhood structure as local search

Representation-Search Space Duality

REPRESENTATION	SEARCH SPACE
Syntactic configuration (e.g., binary string)	Point in search space (e.g., point in Hamming space)
Search Operator (algebraic): defined in terms of manipulation of the representation	**Search Operator (geometric):** defined in terms of spatial relationships
Operational/Concrete	Declarative/Specification
Representation-specific	Representation-independent
Edit operations (bit-flip, swap)	Moves in metric space

ROAR—NET

Representation-Search Space Duality (example)

Traditional uniform crossover can be defined:

(i) geometrically: pick offspring uniformly at random in the Hamming segment between parent points

(ii) algebraically: generate a random recombination mask to position-wise select bits from parent strings

Key insight: Same operator, two perspectives!

Mutation vs Local Search

Geometric Mutation:

- Sample from ball(parent, radius)
- Radius parameter controls neighborhood size
- Single-parent variation

Local Search:

- Sample from direct neighborhood
- Move to better neighbor
- Single-solution evolution

Connection: Both use the same neighborhood structure defined by distance metric

Crossover vs Path-Relinking

Geometric Crossover:

- Sample from segment(parent1, parent2)
- Shortest path in neighborhood structure
- Two-parent recombination

Path-Relinking:

- Systematic exploration between two solutions
- Guided intensification along shortest paths
- Two-solution trajectory

Connection: Both trace paths through the same metric space

Crossover as Path Tracing

Binary example:

- Parent1: [0,0,0,0], Parent2: [1,1,1,1]
- Segment: [0,0,0,0] → [1,0,0,0] → [1,1,0,0] → [1,1,1,0] → [1,1,1,1]
- One-point crossover samples points along this path

Crossover Principled Design

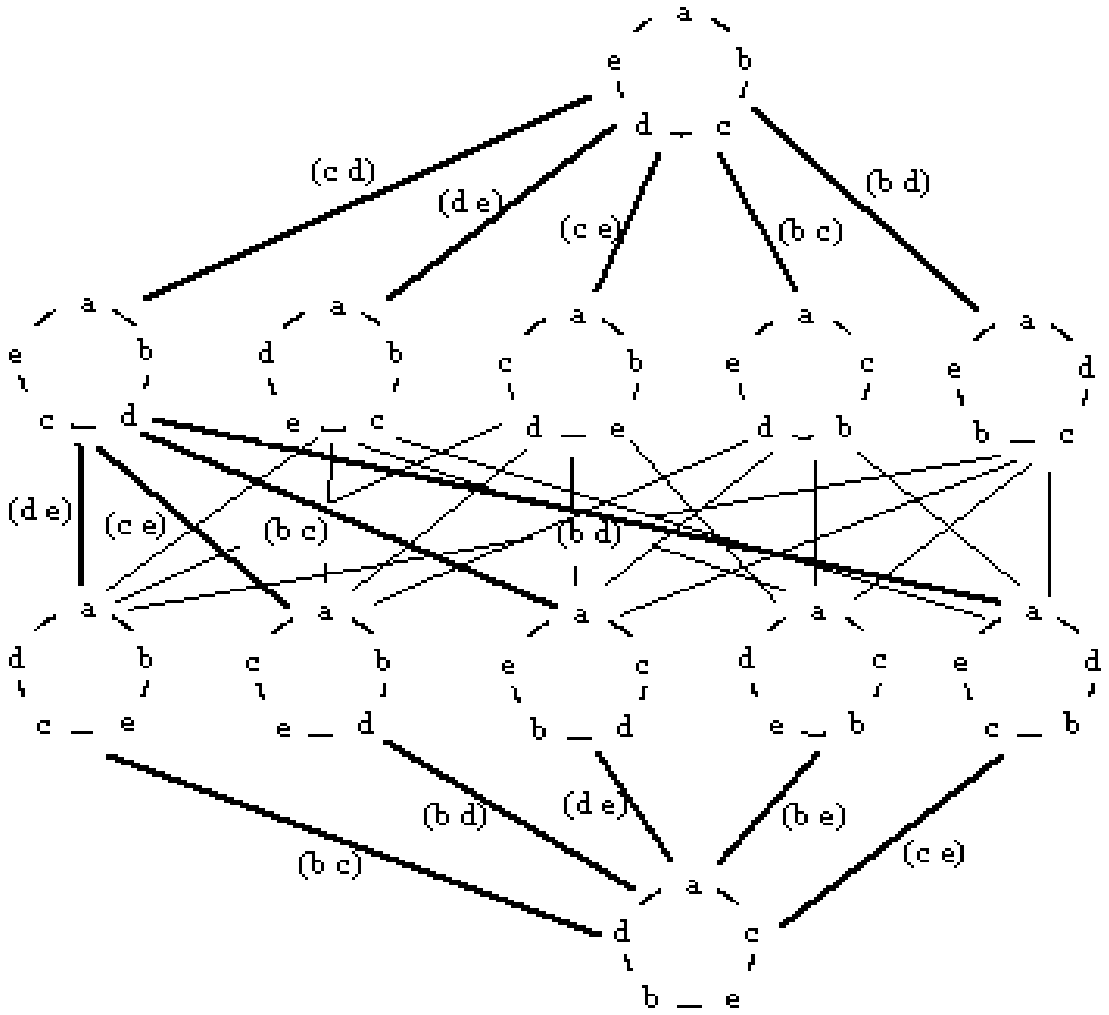
The representation-search space duality enables principled crossover design:

Step 1: Choose appropriate distance metric for your problem
Step 2: Geometric crossover = sample from segment between parents
Step 3: Implement using edit operations that define the distance

Key insight: Different distance metrics → Different crossover operators

- Hamming distance → Uniform crossover
- Swap distance → Order crossover
- Edit distance → Novel problem-specific crossovers

Example: TSP Crossover Design



Choose good edit distance: Reversal distance (i.e., TSP's 2-opt neighborhood)

Geometric definition: Shortest path = minimal sorting trajectory

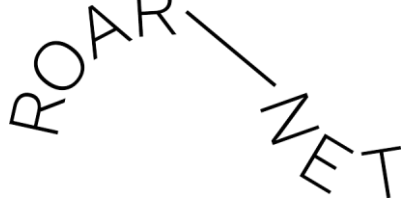
Implementation via edit distance: Use sorting algorithms (sorting by reversals)

Result: New crossover that outperforms traditional Edge Recombination

General principle: Good neighborhood structures → Good crossover operators

ROAR—NET

Part III: Distance-Based Tabu Search



Tabu Search: Basic Concept

- **Core problem:** Local search gets trapped in local optima
- **Tabu search solution:** Use **memory** to escape local optima

1. Start with initial solution
2. Generate neighborhood
3. Remove tabu (forbidden) moves
4. Select best remaining move
5. Update tabu list with new move
6. Repeat until stopping criterion

Key insight: Short-term memory prevents cycling, enables exploration

From Attributes to Tabu Regions

- Traditional tabu search uses **move attributes** (indices, values)
- Attributes define **forbidden regions** in solution space

Example: Binary string with bit-flip moves
Current solution: [1,0,1,0,1]
Tabu list: [pos1, pos2, pos3] (can't flip positions 1, 2, 3)

Tabu region: All solutions with seen values at positions 1, 2, 3
Tabu region: [*, *, *, 0, 1] which contains 8 solutions

Key insight: 3 tabu attributes → region of 8 solutions (not just 3)

Distance-Based Generalization

Replace attribute-based regions with geometric regions

Two types of generalizations:

1. **Convex Hull Regions ▲**
   - Tabu region = convex hull of recently visited solutions
2. **Ball-Based Regions ○**
   - Tabu region = ball centered at oldest solution in tabu list

Generalization 1: Convex Hull Regions

Algorithm:

1. Maintain recent solution history H
2. Tabu region = convex_hull(H)
3. Generate candidates in neighborhood of current solution outside hull
4. Select best non-tabu candidate

Properties:

- Exact generalization of attribute-based TS
- Requires convex hull algorithms (hard)

Generalization 2: Ball-Based Regions

Algorithm:

1. Maintain recent solution history H
2. Tabu region = ball(oldest in H, distance_to_current)
3. Generate candidates in neighborhood of current solution outside ball
4. Select best non-tabu candidate

Properties:

- Approximated generalisation of attribute-based TS
- Simple implementation based on ball (easy)

Visualization of Both Approaches

🎬 Interactive Demonstrations:

Convex Hull Tabu Search:
- Gray shaded region = forbidden area
- Evolving polygonal shape
- Complex but precise memory

Ball-Based Tabu Search:
- Red circle = forbidden area
- Simple circular shape
- Efficient but approximate memory

Comparison Analysis

Similarities:

- Both achieve **cycle prevention** through geometric constraints
- Both define **forbidden regions** in solution space
- Both generalize traditional attribute-based approaches

Differences:

- **Shape:** Polygon vs circle
- **Computation:** Complex vs simple

Convergence: Identical behavior in binary string spaces!

Student Activity

Predict: How will search behavior differ between:

1. Convex hull approach
2. Ball-based approach

Consider:

- Search trajectory shapes
- Escape from local optima
- Computational requirements

Part IV: API Specification

ROAR-NET

Distance-Based Segment API (segment)

Core concept: Segment abstraction for distance-based optimization algorithms

Main components:

1. **Segment** - Path between two solutions with distance operations
2. **Move** - Operations that can modify segment endpoints
3. **Neighbourhood** - Context for segment creation and move generation

Distance-Based Segment API (distance)

Distance relationship:

- **Segment length** represents the distance between two solutions
- **Distance metric** is defined implicitly by the Neighbourhood implementation
- **No explicit distance function** distance is encapsulated within segment operations

Key applications: Distance-based tabu search, path-relinking, geometric crossover

Segment Type and Inspection

```
# Core type
Segment

# Basic inspection operations
segment_length(Segment) -> int | float
```

- segment_length() **returns the distance** between the segment's two endpoints
- **Distance metric depends on the Neighbourhood** that created the segment
- **Examples:** Hamming distance for binary strings, swap distance for permutations

Segment Creation

```
# Create segment between two solutions
segment(Neighbourhood, Solution, Solution) -> Segment
```

- First solution becomes the "left end" of the segment
- Second solution becomes the "right end" of the segment
- Neighbourhood provides the context for distance calculation
- Returns a segment object representing the path between solutions

Distance-Based Move Generation

Core principle: Generate moves based on their effect on **distance between solutions**

Moves affecting right endpoint:

```
# Moves that bring right end closer to left end (REDUCE distance)
moves_right_end_closer(Neighbourhood, Segment) -> Move[0..*]
random_moves_right_end_closer(Neighbourhood, Segment) -> Move[0..1]

# Moves that take right end farther from left end (INCREASE distance)
moves_right_end_farther(Neighbourhood, Segment) -> Move[0..*]
random_moves_right_end_farther(Neighbourhood, Segment) -> Move[0..1]
```

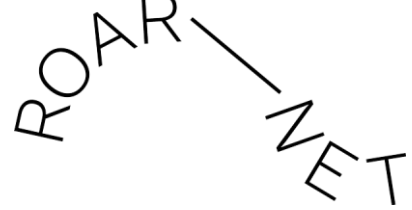
Closer moves: Path-relinking (systematically approach target solution)
Farther moves: Distance-based tabu search (avoid recently visited solutions)

Segment Modification Operations

```
# Apply move to segment endpoints
apply_move_left_end(Move, Segment) -> Segment
apply_move_right_end(Move, Segment) -> Segment
```

Key properties:

- Moves can be applied to either endpoint independently
- Operations return updated segment (may be in-place or new object)
- Enables incremental modification of distance-based constraints



Implementation Example: Hamming distance

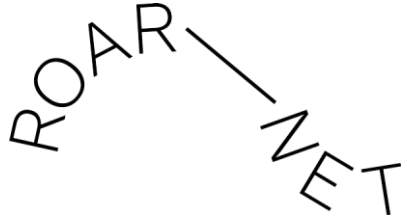
```
class HammingSegment:
    def __init__(self, left_bits, right_bits):
        self.left_bits = left_bits
        self.right_bits = right_bits
        self._length = sum(1 for i in range(len(left_bits))
                           if left_bits[i] != right_bits[i])

    def length(self):
        return self._length
```

Implementation Example: Hamming distance

```
class BitFlipMove:
    def apply_right_end(self, segment):
        # Flip bit and update distance incrementally
        bit_pos = self.position
        old_bit = segment.right_bits[bit_pos]
        segment.right_bits[bit_pos] = 1 - old_bit

        # Update cached distance
        if segment.left_bits[bit_pos] == old_bit:
            segment._length += 1  # Now differs
        else:
            segment._length -= 1  # Now matches
```



Design Principles

Preprocessing efficiency:

- Segment class caches expensive computations
- Avoid recomputing distances and move sets

Distance flexibility:

- Support edit distances and related metrics
- Allow domain-specific distance functions

Incremental updates:

- Dynamic segment modification for tabu search/recombination
- Efficient segment endpoint updates

Conclusions and Research Directions

Key Unifying Principles:

1. **Distance metrics unify search operators** across representations
2. **EAs and local search exploit same neighborhood structures**
3. **Geometric interpretations bridge theory and practice**
4. **Multiple geometric approaches achieve similar algorithmic goals**

ROAR—NET

Open Research Questions

Active research areas:

- **Optimal distance selection:** Which metrics work best for specific problems?
- **Adaptive geometric parameters:** Dynamic radius/neighborhood control
- **Hybrid geometric approaches:** Combining convex hull and ball-based methods
- **API extensions:** Support for new distance metrics and representations

Next Steps for Students

Immediate actions:

1. **Implement** geometric operators for your project representations
2. **Compare** convex hull vs ball-based tabu search on your problems
3. **Design** custom distance metrics for your problem domains
4. **Experiment** with EA/local search hybrids using the unified framework

Goal: Apply distance-based thinking to your optimization challenges

Thank You!

Questions & Discussion

Resources:

- API documentation: segment-spec.md
- Visualization tools: Available on GitHub
- Further reading: Geometric Theory of Evolutionary Algorithms

Contact: a.moraglio@exeter.ac.uk

ROAR NET

This presentation is based upon work from COST Action Randomised Optimisation Algorithms Research Network (ROAR-NET), CA22137, supported by COST (European Cooperation in Science and Technology).

COST (European Cooperation in Science and Technology) is a funding agency for research and innovation networks. Our Actions help connect research initiatives across Europe and enable scientists to grow their ideas by sharing them with their peers. This boosts their research, career and innovation.

www.cost.eu

cost
EUROPEAN COOPERATION IN SCIENCE & TECHNOLOGY

Funded by the European Union

ROAR-NET