

Báo cáo Đề án: Tối ưu hóa Đàn kiến (Ant Colony Optimization)

Tên Của Bạn

Ngày 14 tháng 11 năm 2025

Mục lục

Lời mở đầu	4
1 Giới thiệu chung	4
1.1 Thuật toán heuristic	4
1.2 Swarm intelligence algorithm	4
2 Các thuật toán truyền thống để so sánh	4
2.1 Genetic Algorithm	4
2.1.1 Giới thiệu chung	4
2.1.2 Cơ sở toán học	4
2.1.3 Cách hoạt động của thuật toán	6
2.2 Hill Climbing (Steepest Ascent)	6
2.2.1 Giới thiệu chung	6
2.2.2 Cơ sở toán học	6
2.2.3 Cách hoạt động của thuật toán	7
2.3 Simulated Annealing	7
2.3.1 Giới thiệu chung	7
2.3.2 Cơ sở toán học	8
2.3.3 Cách hoạt động của thuật toán	8
2.4 Breadth-First Search (BFS)	9
2.4.1 Giới thiệu chung	9
2.4.2 Cơ sở toán học	9
2.4.3 Cách hoạt động của thuật toán	9
2.5 Depth-First Search (DFS)	10
2.5.1 Giới thiệu chung	10
2.5.2 Cơ sở toán học	10
2.5.3 Cách hoạt động của thuật toán	10
2.6 A* Search	10
2.6.1 Giới thiệu chung	10
2.6.2 Cơ sở toán học	11
2.6.3 Cách hoạt động của thuật toán	12

3 Các bài toán sử dụng để đánh giá	12
3.1 Các bài toán rời rạc	12
3.1.1 The travelling salesman problem	12
3.1.2 Knapsack Problem (KP)	14
3.1.3 Graph Coloring (GC)	14
3.2 Các bài toán liên tục	15
3.2.1 Sphere function	15
3.2.2 Rosenbrock function	16
3.2.3 Rastrigin function	16
3.2.4 Ackley function	16
4 Ant Colony Optimization	17
4.1 Giới thiệu thuật toán	17
4.1.1 Ý tưởng thuật toán	17
4.1.2 Lịch sử thuật toán	17
4.1.3 Những ứng dụng	18
4.2 Cơ sở toán học	18
4.2.1 Quy tắc Lựa chọn Đường đi (Xây dựng Giải pháp)	18
4.2.2 Quy tắc Cập nhật Pheromone (Học tập)	18
4.3 Triển khai thuật toán	19
4.3.1 Cách hoạt động của thuật toán	19
4.3.2 Triển khai kỹ thuật	20
5 Particle swarm optimization	20
5.1 Giới thiệu thuật toán	20
5.1.1 Ý tưởng thuật toán	20
5.1.2 Lịch sử thuật toán	21
5.1.3 Những ứng dụng	21
5.2 Cơ sở toán học của thuật toán	21
5.2.1 Phương trình Cập nhật Vận tốc (Velocity Update)	21
5.2.2 Phương trình Cập nhật Vị trí (Position Update)	22
5.3 Triển khai thuật toán	22
5.3.1 Cách hoạt động của thuật toán	22
5.3.2 Triển khai kỹ thuật	23
6 Artificial Bee Colony	23
6.1 Giới thiệu thuật toán	23
6.2 Ý tưởng thuật toán và nguyên lý hoạt động	24
6.3 Giải thích cơ chế hoạt động bằng công cụ toán học	24
6.3.1 Ký hiệu và các tham số chính	24
6.3.2 Nguyên lý	24
6.3.3 Mã giả (Pseudo-code) thuật toán ABC	25
7 Firefly Algorithm	26
7.1 Giới thiệu	26
7.1.1 Ý tưởng cốt lõi	26
7.1.2 Lịch sử ngắn gọn	26
7.1.3 Ứng dụng tiêu biểu	26
7.2 Cơ sở toán học của thuật toán	27
7.3 Chi tiết triển khai thuật toán (Mã giả)	28
7.4 Thuật toán dùng để so sánh	29

8 Cuckoo Search	31
8.1 Giới thiệu thuật toán	31
8.2 Ý tưởng thuật toán	31
8.3 Cơ sở toán học của thuật toán	31
8.4 Triển khai thuật toán	32
8.4.1 Cách hoạt động của thuật toán	32
8.4.2 Mã giả	32
9 Thiết lập các bài toán đánh giá	33
9.1 ACO	33
9.2 PSO	33
9.3 ABC	33
9.4 Firefly Algorithm	33
9.5 Cuckoo Search	34
10 Phân tích kết quả bài toán	34
10.1 Firefly Algorithm	34
10.1.1 Rastrigin: mức độ tiệm cận nghiệm tối ưu và đặc trưng hội tụ	34
10.1.2 Rastrigin: đa dạng quần thể và hiện tượng dừng sớm (stagnation)	41
10.1.3 Knapsack: hiệu năng fixed-budget (gap tới nghiệm tối ưu)	45
10.1.4 Knapsack: performance/data profiles và so sánh thống kê	48
10.1.5 Thảo luận và kết luận	54
11 Kết luận chung	56
12 Kết luận chung	56
Phân công công việc	56
Phụ lục	56

Lời mở đầu

1 Giới thiệu chung

1.1 Thuật toán heuristic

1.2 Swarm intelligence algorithm

2 Các thuật toán truyền thống để so sánh

2.1 Genetic Algorithm

2.1.1 Giới thiệu chung

Genetic Algorithm (GA - Giải thuật di truyền) là một kỹ thuật tìm kiếm nhằm tìm ra đáp án gần đúng trong các bài toán tối ưu hóa mô hình và bài toán tìm kiếm. GA được xem là một trong những metaheuristic phổ biến và được ứng dụng nhiều nhất.

Trong một quần thể trong tự nhiên, các loài sinh vật phải thích nghi và thay đổi qua từng thế hệ để có thể sinh tồn, đó gọi là sự tiến hóa (theo học thuyết tiến hóa của Darwin). Ở hình trên, các con hươu cao cổ thông qua một cơ chế tiến hóa gọi là chọn lọc tự nhiên, những con thấp sẽ không thể ăn lá và chết dần, qua thời gian dài chỉ còn lại những con cao hơn tiếp tục sinh sản và phát triển quần thể. Như vậy, các loài sinh vật luôn phải tiến hóa để tránh bị đào thải trong môi trường tự nhiên khắc nghiệt.

GA được phát minh bởi John Holland và các cộng sự tại đại học Michigan vào những năm 1960, dựa trên các nguyên tắc của tiến hóa, bao gồm các quá trình lai tạo (crossover), đột biến (mutation) và chọn lọc (selection). Nó bắt đầu phổ biến vào những năm 1990, khi người ta bắt đầu tìm kiếm những công cụ heuristic để giải quyết những bài toán mà giải thuật chính xác không khả thi.

Thuật giải GA đã và đang được ứng dụng để giải quyết các bài toán trong rất nhiều lĩnh vực của cuộc sống cũng như trong kỹ thuật, ví dụ như tối ưu hóa, học máy,

2.1.2 Cơ sở toán học

Không gian nghiệm và mã hóa. Gọi \mathcal{C} là không gian các nhiễm sắc thể (chromosomes). Một nhiễm sắc thể thường được biểu diễn là một chuỗi ký tự có độ dài l :

$$c = (c_1, c_2, \dots, c_l) \in \{0, 1\}^l \quad \text{hoặc} \quad c \in \mathbb{R}^l$$

Mỗi nhiễm sắc thể c tương ứng với một *phenotype* (lời giải) thông qua một ánh xạ giải mã $\phi : \mathcal{C} \rightarrow \mathcal{X}$. Hàm đánh giá (fitness) là

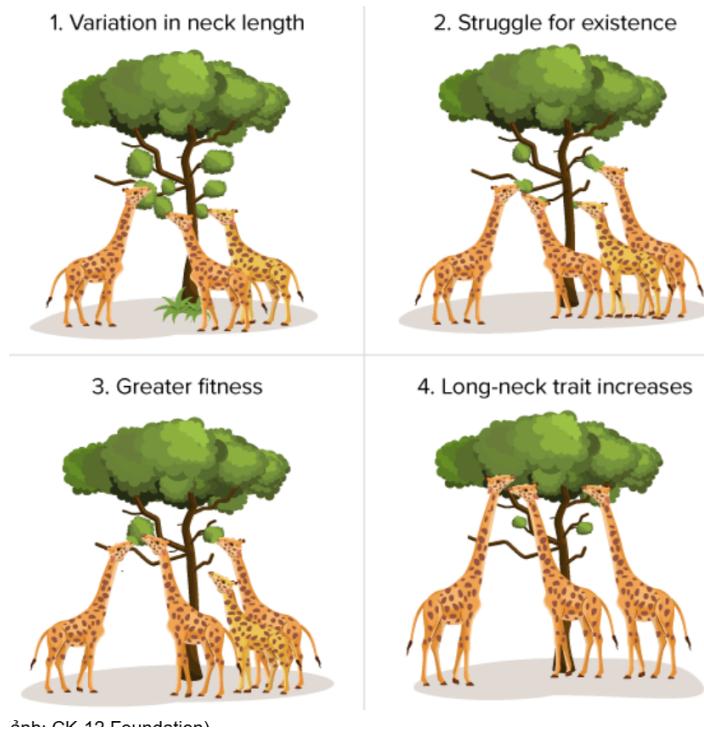
$$F : \mathcal{C} \rightarrow \mathbb{R}, \quad F(c) \text{ là mức "tốt" của lời giải } c.$$

Chọn lọc (Selection). Một phương pháp phổ biến là chọn theo tỉ lệ fitness (roulette-wheel). Nếu quần thể hiện tại có N cá thể $c^{(1)}, \dots, c^{(N)}$ thì xác suất chọn cá thể i là:

$$p_i = \frac{F(c^{(i)})}{\sum_{j=1}^N F(c^{(j)})}. \tag{1}$$

Schema và Schema Theorem. Một *schema* H là một mẫu cố định trên một số vị trí của nhiễm sắc thể, ký hiệu bằng chuỗi trong bảng chữ cái $\{0, 1, *\}$ ("*" là wildcard). Đặt:

- $m_H(t)$: số cá thể thuộc schema H ở thế hệ t ;



Hình 1: Quá trình tiến hóa của hươu cao cổ (Ảnh: CK-12 Foundation)

- $\bar{F}_H(t)$: độ fitness trung bình của các cá thể trong H ;
- $\bar{F}(t)$: độ fitness trung bình toàn quần thể;
- $F_H(t) = \frac{\bar{F}_H(t)}{\bar{F}(t)}$ là *relative fitness* của schema H ;
- l_H là khoảng cách giữa gene đầu và gene cuối được cố định trong H (schema length);
- o_H là order của H (số bit cố định trong schema);
- p_c là xác suất crossover, p_m là xác suất mutation (tại mỗi locus).

Schema Theorem (dạng bất đẳng thức cho kì vọng):

$$\mathbb{E}[m_H(t+1)] \geq F_H(t) m_H(t) \left(1 - p_c \frac{l_H}{l-1}\right) (1 - p_m)^{o_H}. \quad (2)$$

Ý nghĩa: những schema ngắn, có ít bit cố định (nhỏ o_H), và có relative fitness $F_H(t) > 1$ sẽ có xu hướng gia tăng trong quần thể. Đây là cơ sở trực giác cho "building-block hypothesis": GA kết hợp những phần tốt (building blocks) từ nhiều cá thể để tạo ra lời giải tốt hơn.

Các tham số quan trọng (toán học).

- Kích thước quần thể N : quyết định độ phong phú mẫu cho phân phối; quá nhỏ dễ mất đa dạng.
- Xác suất crossover p_c và mutation p_m : ảnh hưởng cân bằng khám phá/khai thác.
- Cost đánh giá fitness: độ phức tạp tính toán thường là $\mathcal{O}(N \cdot \text{cost_eval} \cdot G)$ với G là số thế hệ.

2.1.3 Cách hoạt động của thuật toán

Các toán tử chính.

- **Selection:** Lấy mẫu theo phân phối p_i như phương trình 1 hoặc dùng tournament selection.
- **Crossover:** Cho hai cha mẹ $c^{(a)}$ và $c^{(b)}$. Với 1-point crossover tại vị trí k với $1 \leq k < l$, sinh con:
$$c^{(child)} = (c_1^{(a)}, \dots, c_k^{(a)}, c_{k+1}^{(b)}, \dots, c_l^{(b)}).$$
- **Mutation:** Với mỗi locus, thực hiện biến đổi theo Bernoulli(p_m). Với mã hóa bit-string, mutation là bit-flip.
- **Elitism:** Giữ lại e cá thể tốt nhất sang thế hệ sau để tránh mất nghiệm tốt do tính ngẫu nhiên.

Algorithm 1 Genetic Algorithm (GA)

- 1: **Input:** kích thước quần thể N , chiều dài NST l , xác suất crossover p_c , xác suất mutation p_m , số thế hệ tối đa G , elitism e .
 - 2: Khởi tạo quần thể $P^0 = \{c^{(1)}, \dots, c^{(N)}\}$ (thường ngẫu nhiên).
 - 3: **for** $t = 0$ **to** $G - 1$ **do**
 - 4: Tính $F(c)$ cho mọi $c \in P^t$.
 - 5: Sao chép e cá thể tốt nhất sang P^{t+1} (elitism).
 - 6: **while** $\text{size}(P^{t+1}) < N$ **do**
 - 7: Chọn cặp cha mẹ theo selection (ví dụ roulette hoặc tournament).
 - 8: Với xác suất p_c , áp dụng crossover để sinh 1 hoặc 2 con; ngược lại sao chép cha mẹ.
 - 9: Áp dụng mutation cho từng con với xác suất p_m trên mỗi locus.
 - 10: Thêm con vào P^{t+1} .
 - 11: **end while**
 - 12: **end for**
 - 13: Trả về cá thể tốt nhất tìm được.
-

2.2 Hill Climbing (Steepest Ascent)

2.2.1 Giới thiệu chung

Hill Climbing (leo đồi, *steepest-ascent hill climbing*) là một giải thuật tìm kiếm cục bộ (local search) đơn giản, hoạt động bằng cách lặp lại hai bước: sinh lân cận và di chuyển sang nghiệm lân cận “tốt nhất” nếu nó cải thiện hàm mục tiêu. Thuật toán mang tính tham lam (greedy): luôn ưu tiên cải thiện tức thời, nên dễ rơi vào cực trị địa phương, plateau (vùng bằng phẳng) hoặc ridge (sườn hẹp).

Trong bối cảnh tối ưu hóa, Hill Climbing có thể áp dụng cho cả không gian nghiệm rời rạc (ví dụ: Knapsack) và liên tục (ví dụ: Rastrigin, nếu ta định nghĩa được phép biến đổi lân cận). Trong đồ án này, Hill Climbing được dùng như một baseline truyền thống để so sánh với các metaheuristic bầy đàn.

2.2.2 Cơ sở toán học

Gọi \mathcal{X} là không gian nghiệm và $f : \mathcal{X} \rightarrow \mathbb{R}$ là hàm mục tiêu (giả sử bài toán tối thiểu hóa). Với mỗi nghiệm $x \in \mathcal{X}$, ta định nghĩa một tập lân cận $N(x) \subset \mathcal{X}$ thông qua một phép biến đổi cục bộ (ví dụ: lật một bit trong mã hóa nhị phân, hoán vị hai phần tử, hoặc nhiễu Gaussian trong không gian liên tục).

Ở bước lặp k , Hill Climbing dạng *steepest ascent* (cho bài toán tối đa hóa) hoặc *steepest descent* (cho bài toán tối thiểu hóa) thực hiện:

$$x_{k+1} = \arg \min_{y \in N(x_k)} f(y)$$

và chỉ cập nhật nếu $f(x_{k+1}) < f(x_k)$ (tối thiểu hóa). Thuật toán dừng khi:

$$\min_{y \in N(x_k)} f(y) \geq f(x_k),$$

tức là không còn lân cận nào cải thiện được hàm mục tiêu – ta đạt tới một cực trị địa phương (local optimum) theo cấu trúc lân cận đã chọn.

Dộ phức tạp mỗi vòng lặp phụ thuộc vào kích thước lân cận:

$$\text{cost_per_iter} = \mathcal{O}(|N(x)| \cdot \text{cost_eval}),$$

trong đó cost_eval là chi phí tính $f(x)$. Tổng chi phí xấp xỉ $\mathcal{O}(K \cdot |N| \cdot \text{cost_eval})$ với K là số vòng lặp đến khi hội tụ hoặc đạt ngưỡng dừng.

2.2.3 Cách hoạt động của thuật toán

Algorithm 2 Hill Climbing (Steepest Ascent / Descent)

```

1: Input: nghiệm khởi tạo  $x_0$ , hàm mục tiêu  $f$ , hàm sinh lân cận  $N(x)$ , số vòng lặp tối đa  $K$ ,  
ngưỡng cải thiện  $\epsilon$ .
2: Đặt  $x \leftarrow x_0$ .
3: for  $k = 1$  to  $K$  do
4:   Sinh tập lân cận  $N(x) = \{y_1, \dots, y_m\}$ .
5:   Tìm nghiệm tốt nhất trong lân cận:
          $y^* = \arg \min_{y \in N(x)} f(y)$  (giả sử tối thiểu hóa).
6:   if  $f(y^*) < f(x) - \epsilon$  then
7:     Cập nhật  $x \leftarrow y^*$ .
8:   else
9:     break {Không còn cải thiện đáng kể, dừng}
10:  end if
11: end for
12: Output: nghiệm cuối cùng  $x$  và giá trị  $f(x)$ .
```

2.3 Simulated Annealing

2.3.1 Giới thiệu chung

Simulated Annealing (SA - tối luyện mô phỏng) là một metaheuristic tìm kiếm cục bộ có yếu tố ngẫu nhiên, lấy cảm hứng từ quá trình tối luyện vật liệu trong vật lý. Khác với Hill Climbing, SA cho phép chấp nhận tạm thời các nghiệm *xấu hơn* với một xác suất phụ thuộc vào nhiệt độ T , giúp thuật toán có khả năng thoát khỏi cực trị địa phương.

SA được đề xuất bởi Kirkpatrick và cộng sự (1983) như một mở rộng của thuật toán Metropolis. Với lịch trình làm nguội (cooling schedule) đủ chậm, SA có bảo đảm hội tụ về nghiệm tối ưu toàn cục trong giới hạn lý thuyết, dù trong thực tế thường sử dụng lịch trình nhanh hơn để đạt hiệu quả tính toán.

2.3.2 Cơ sở toán học

Giả sử bài toán tối thiểu hóa $f : \mathcal{X} \rightarrow \mathbb{R}$. Tại bước k , với nghiệm hiện tại x_k và nhiệt độ $T_k > 0$, SA sinh một lân cận $y \in N(x_k)$ và xét hiệu:

$$\Delta f = f(y) - f(x_k).$$

Quy tắc chấp nhận:

$$x_{k+1} = \begin{cases} y, & \text{nếu } \Delta f \leq 0 \quad (\text{cải thiện}) \\ y \text{ với xác suất } p_{\text{accept}} = \exp\left(-\frac{\Delta f}{T_k}\right), & \text{nếu } \Delta f > 0 \\ x_k, & \text{ngược lại.} \end{cases}$$

Khi T_k cao, xác suất chấp nhận nghiệm xấu tương đối lớn, giúp thuật toán khám phá không gian rộng hơn. Khi $T_k \rightarrow 0$, thuật toán dần trở nên giống Hill Climbing (chỉ chấp nhận nghiệm cải thiện). Lịch trình làm nguội thường được chọn dưới dạng:

$$T_{k+1} = \alpha T_k, \quad 0 < \alpha < 1,$$

hoặc

$$T_k = \frac{T_0}{1 + \beta k},$$

với T_0 là nhiệt độ ban đầu và $\beta > 0$ là hệ số làm nguội.

2.3.3 Cách hoạt động của thuật toán

Algorithm 3 Simulated Annealing (SA)

- 1: **Input:** nghiệm khởi tạo x_0 , hàm mục tiêu f , hàm sinh lân cận $N(x)$, nhiệt độ ban đầu T_0 , nhiệt độ tối thiểu T_{\min} , số vòng lặp tối đa K , hệ số làm nguội $\alpha \in (0, 1)$.
 - 2: Dặt $x \leftarrow x_0$, $x_{\text{best}} \leftarrow x_0$, $T \leftarrow T_0$.
 - 3: **for** $k = 1$ **to** K **do**
 - 4: Chọn ngẫu nhiên $y \in N(x)$.
 - 5: Tính $\Delta f = f(y) - f(x)$.
 - 6: **if** $\Delta f \leq 0$ **then**
 - 7: chấp nhận y : $x \leftarrow y$.
 - 8: **else**
 - 9: Lấy $u \sim \text{Uniform}(0, 1)$.
 - 10: **if** $u < \exp(-\Delta f / T)$ **then**
 - 11: chấp nhận y : $x \leftarrow y$.
 - 12: **end if**
 - 13: **end if**
 - 14: **if** $f(x) < f(x_{\text{best}})$ **then**
 - 15: Cập nhật nghiệm tốt nhất $x_{\text{best}} \leftarrow x$.
 - 16: **end if**
 - 17: Cập nhật nhiệt độ $T \leftarrow \alpha T$.
 - 18: **if** $T < T_{\min}$ **then**
 - 19: **break**
 - 20: **end if**
 - 21: **end for**
 - 22: **Output:** nghiệm tốt nhất x_{best} và giá trị $f(x_{\text{best}})$.
-

2.4 Breadth-First Search (BFS)

2.4.1 Giới thiệu chung

Breadth-First Search (BFS - tìm kiếm theo bề rộng) là một thuật toán tìm kiếm trên đồ thị/cây trạng thái, khám phá các đỉnh (trạng thái) theo từng “lớp độ sâu” tăng dần. BFS là thuật toán *uninformed search* kinh điển, đảm bảo tìm được lời giải tối ưu về số bước (nếu mỗi cạnh có cùng chi phí) và luôn hoàn chỉnh (complete) nếu không gian trạng thái hữu hạn.

Trong đồ án, BFS được sử dụng như một baseline truyền thống cho các bài toán rắc rối không gian trạng thái nhỏ, nơi có thể duyệt gần như toàn bộ không gian (ví dụ: phiên bản nhỏ của Knapsack, bài toán đường đi trên lưới, ...).

2.4.2 Cơ sở toán học

Mô hình hóa bài toán dưới dạng đồ thị trạng thái có hướng hoặc vô hướng:

$$G = (V, E),$$

trong đó mỗi đỉnh $v \in V$ tương ứng với một trạng thái, và mỗi cạnh $(u, v) \in E$ tương ứng với một phép chuyển trạng thái hợp lệ. Cho s là trạng thái xuất phát, G là tập đích (*goal states*).

BFS mở rộng các đỉnh theo thứ tự độ sâu tăng dần: tất cả các đỉnh ở độ sâu 0, rồi đến tất cả ở độ sâu 1, ..., cho đến khi gặp một đỉnh thuộc G .

Với b là hệ số phân nhánh (branching factor), d là độ sâu nhỏ nhất của lời giải, độ phức tạp xấp xỉ:

$$\text{Time} = \mathcal{O}(b^d), \quad \text{Space} = \mathcal{O}(b^d),$$

do phải lưu toàn bộ “vòng biên” (frontier) ở mỗi lớp độ sâu. Đây là điểm yếu của BFS trên không gian trạng thái lớn.

2.4.3 Cách hoạt động của thuật toán

Algorithm 4 Breadth-First Search (BFS)

- 1: **Input:** đồ thị trạng thái (V, E) (ngầm qua hàm sinh hàng xóm), trạng thái bắt đầu s , tập đích G .
 - 2: Khởi tạo hàng đợi $Q \leftarrow \emptyset$, tập đã thăm $Visited \leftarrow \emptyset$.
 - 3: Thêm s vào Q và $Visited$.
 - 4: **while** Q không rỗng **do**
 - 5: Lấy phần tử đầu hàng đợi $u \leftarrow Q.pop_front()$.
 - 6: **if** $u \in G$ **then**
 - 7: Dừng và trả về đường đi từ s đến u (truy vết qua *parent*).
 - 8: **end if**
 - 9: **for** mỗi hàng xóm v của u **do**
 - 10: **if** $v \notin Visited$ **then**
 - 11: Thêm v vào $Visited$.
 - 12: Gán *parent* của v là u (để truy vết đường đi).
 - 13: Thêm v vào cuối hàng đợi Q .
 - 14: **end if**
 - 15: **end for**
 - 16: **end while**
 - 17: Nếu vòng lặp kết thúc mà không tìm được đích, báo “không có lời giải”.
-

2.5 Depth-First Search (DFS)

2.5.1 Giới thiệu chung

Depth-First Search (DFS - tìm kiếm theo chiều sâu) là một thuật toán tìm kiếm trên đồ thi/cây, luôn ưu tiên đi sâu theo một nhánh cho tới khi không đi tiếp được nữa, rồi quay lui (backtracking). DFS cũng là một thuật toán *uninformed search*, không đảm bảo tối ưu và trong không gian vô hạn có thể không dừng nếu không có cơ chế cắt tỉa.

Ưu điểm chính của DFS là chi phí bộ nhớ thấp hơn BFS, nên được dùng như baseline cho các bài toán rắc rối với không gian trạng thái lớn nhưng độ sâu hữu hạn.

2.5.2 Cơ sở toán học

Với cùng mô hình đồ thị trạng thái $G = (V, E)$, DFS mở rộng các đỉnh theo thứ tự *đi sâu nhất có thể*. Nếu dùng phiên bản có giới hạn độ sâu L (depth-limited DFS), ta chỉ cho phép đi tối đa L rồi quay lui.

Với hệ số phân nhánh b và độ sâu tối đa m (có thể lớn hơn d), độ phức tạp:

$$\text{Time} = \mathcal{O}(b^m), \quad \text{Space} = \mathcal{O}(b \cdot m),$$

do chỉ lưu một đường đi hiện tại và các nhánh chưa mở ở mỗi mức.

2.5.3 Cách hoạt động của thuật toán

Algorithm 5 Depth-First Search (DFS) dạng dùng ngăn xếp

- 1: **Input:** đồ thị trạng thái (V, E) (ngầm qua hàm sinh hàng xóm), trạng thái bắt đầu s , tập đích G .
 - 2: Khởi tạo ngăn xếp $S \leftarrow \emptyset$, tập đã thăm $Visited \leftarrow \emptyset$.
 - 3: Thêm s vào S .
 - 4: **while** S không rỗng **do**
 - 5: Lấy phần tử trên đỉnh ngăn xếp $u \leftarrow S.pop()$.
 - 6: **if** $u \in Visited$ **then**
 - 7: **continue**
 - 8: **end if**
 - 9: Thêm u vào $Visited$.
 - 10: **if** $u \in G$ **then**
 - 11: Dừng và trả về đường đi từ s đến u (truy vết qua *parent*).
 - 12: **end if**
 - 13: **for** mỗi hàng xóm v của u **do**
 - 14: **if** $v \notin Visited$ **then**
 - 15: Gán *parent* của v là u .
 - 16: Thêm v vào ngăn xếp S .
 - 17: **end if**
 - 18: **end for**
 - 19: **end while**
 - 20: Nếu vòng lặp kết thúc mà không tìm được đích, báo “không có lời giải”.
-

2.6 A* Search

2.6.1 Giới thiệu chung

A* Search là một thuật toán tìm kiếm có định hướng (informed search) sử dụng hàm heuristic để dẫn hướng tìm kiếm tới đích. A* đặc biệt phổ biến trong các bài toán tìm đường (path-finding)

trên lưới, đồ thị, hoặc bản đồ, nhờ khả năng tìm được đường đi tối ưu với chi phí trung bình thấp hơn nhiều so với các thuật toán uninformed như BFS.

Ý tưởng cốt lõi của A* là đánh giá mỗi trạng thái n bằng một hàm:

$$f(n) = g(n) + h(n),$$

trong đó $g(n)$ là chi phí thực tế từ trạng thái bắt đầu đến n , còn $h(n)$ là ước lượng chi phí còn lại từ n đến đích. Nếu $h(n)$ được chọn “tốt”, A* sẽ tập trung mở rộng các trạng thái tiềm năng, giảm đáng kể số lượng trạng thái phải duyệt.

2.6.2 Cơ sở toán học

Xét đồ thị có trọng số dương:

$$G = (V, E), \quad c(u, v) > 0 \text{ là chi phí trên cạnh } (u, v).$$

Cho s là trạng thái bắt đầu, G (trùng ký hiệu, nhưng ngữ cảnh khác) là tập trạng thái đích. Với mỗi nút n , ta định nghĩa:

- $g(n)$: chi phí đường đi tốt nhất từ s đến n đã biết cho tới thời điểm hiện tại;
- $h(n)$: heuristic ước lượng chi phí tối thiểu từ n đến một đích bất kỳ;
- $f(n) = g(n) + h(n)$: ước lượng tổng chi phí của đường đi qua n .

Heuristic h được gọi là *admissible* nếu:

$$0 \leq h(n) \leq h^*(n), \quad \forall n,$$

trong đó $h^*(n)$ là chi phí thật sự tối thiểu từ n đến đích. Khi h admissible và chi phí cạnh là dương, A* đảm bảo tìm được đường đi tối ưu. Nếu h còn thỏa thêm tính *consistent* (hay *monotone*):

$$h(u) \leq c(u, v) + h(v), \quad \forall (u, v) \in E,$$

thì giá trị $f(n)$ dọc theo một đường đi sẽ không giảm, và mỗi nút chỉ cần đưa ra khỏi hàng đợi ưu tiên một lần.

2.6.3 Cách hoạt động của thuật toán

Algorithm 6 A* Search

```
1: Input: đồ thị  $(V, E)$ , chi phí cạnh  $c(u, v)$ , trạng thái bắt đầu  $s$ , tập đích  $G$ , heuristic  $h(n)$ .
2: Khởi tạo tập mở (open set) là một hàng đợi ưu tiên OPEN, tập đóng CLOSED.
3: Với mọi  $n \in V$ , đặt  $g(n) \leftarrow +\infty$ . Đặt  $g(s) \leftarrow 0$ ,  $f(s) \leftarrow h(s)$ .
4: Thêm  $s$  vào OPEN với khóa  $f(s)$ .
5: while OPEN không rỗng do
6:   Lấy nút  $n$  trong OPEN có  $f(n)$  nhỏ nhất và loại nó khỏi OPEN.
7:   if  $n \in G$  then
8:     Dừng và trả về đường đi tối ưu từ  $s$  đến  $n$  (truy vết qua parent).
9:   end if
10:  Thêm  $n$  vào CLOSED.
11:  for mỗi hàng xóm  $v$  của  $n$  do
12:    if  $v \in CLOSED$  then
13:      continue
14:    end if
15:    Tính chi phí tạm thời  $g_{tent} \leftarrow g(n) + c(n, v)$ .
16:    if  $g_{tent} < g(v)$  then
17:      Cập nhật parent của  $v$  là  $n$ .
18:      Cập nhật  $g(v) \leftarrow g_{tent}$ .
19:      Cập nhật  $f(v) \leftarrow g(v) + h(v)$ .
20:      if  $v$  chưa nằm trong OPEN then
21:        Thêm  $v$  vào OPEN với khóa  $f(v)$ .
22:      else
23:        Cập nhật khóa của  $v$  trong OPEN.
24:      end if
25:    end if
26:  end for
27: end while
28: Nếu vòng lặp kết thúc mà không gặp trạng thái đích, báo “không có đường đi”.
```

3 Các bài toán sử dụng để đánh giá

3.1 Các bài toán rời rạc

3.1.1 The travelling salesman problem

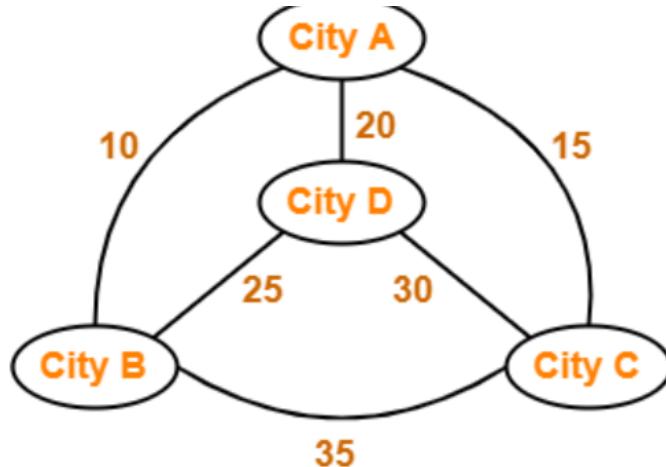
Giới thiệu chung

The travelling salesman problem (TSP) là một bài toán kinh điển trong lý thuyết đồ thị và có rất nhiều ứng dụng trong thực tế và kĩ thuật.

Bài toán yêu cầu tìm ra tuyến đường ngắn nhất để một người đi qua tất cả các thành phố được cho, mỗi thành phố chỉ ghé thăm đúng một lần, và quay trở lại điểm xuất phát ban đầu.

Nó có thể được phác biểu dưới dạng đồ thị như sau: Cho một đồ thị vô hướng có trọng số, tìm chu trình ngắn nhất xuất phát từ một đỉnh bất kì, đi qua tất cả các đỉnh đúng một lần và quay về điểm xuất phát (chu trình hamilton ngắn nhất). Đây là một bài toán NP-hard, có thể hiểu đơn giản là chúng ta chưa thể tìm ra lời giải đa thức chính xác cho bài toán này. Vì vậy, các phương pháp metaheuristic được đưa ra nhằm tìm ra lời giải chính xác gần đúng trong thời gian hợp lý.

Bài toán được ứng dụng rộng rãi trong các lĩnh vực của kĩ thuật và đời sống, đặc biệt là các bài toán tối ưu hóa như lập kế hoạch, lập lịch trình, hậu cần, đóng gói,...



Hình 2: Minh họa TSP (nguồn: Gate Vidyalay)

Những thuật toán áp dụng

ACO

ACO (Ant Colony Optimization) là một metaheuristic phù hợp cho các bài toán đồ thị, đặc biệt là TSP.

PSO

PSO (Particle Swarm Optimization) vốn thiết kế cho bài toán liên tục nhưng có thể điều chỉnh cho TSP bằng cách biểu diễn *thứ tự các thành phố* (permutation) là vị trí của từng hạt và biểu diễn vận tốc bằng các thao tác hoán vị (swap).

Tóm tắt cách PSO hoạt động trong TSP:

1. Mỗi hạt lưu một **route** (một hoán vị — chuỗi các chỉ số thành phố, bắt đầu và kết thúc tại 0).
2. Đánh giá độ phù hợp (fitness) của mỗi route bằng tổng khoảng cách giữa các cặp liên tiếp (dùng ma trận khoảng cách hoặc tính từ tọa độ).
3. Cập nhật “vận tốc” dưới dạng danh sách các cặp hoán đổi (swaps) dựa trên 3 thành phần: *inertia* (w) giữ một phần vận tốc cũ, *cognitive* (c_1) hướng về personal best, và *social* (c_2) hướng về global best.
4. Áp dụng các swap lên route để tạo route mới; nếu cần, thực hiện reshuffle (xáo trộn) để tăng khám phá.
5. Lặp lại cho đến khi đạt điều kiện dừng (số iter, hội tụ, v.v.).

Swap operation

- *Định nghĩa:* Một swap là thao tác hoán đổi hai vị trí trong chuỗi route, ví dụ $\text{swap}(i, j)$ sẽ hoán đổi phần tử ở chỉ số i với phần tử ở chỉ số j trong route.

- *Ý nghĩa:* Mỗi swap thay đổi thứ tự thăm các thành phố, do đó có thể làm tăng hoặc giảm tổng chiều dài chuyến đi. Dùng tập hợp các swap (vận tốc rời rạc) để biểu diễn hướng di chuyển của một hạt trong không gian hoán vị.

- *Cách áp dụng trong PSO rời rạc:*

1. Xây danh sách swap mới (kết hợp giữ lại một phần swap cũ theo *inertia*, thêm swap do *personal best* và swap do *global best*).
2. Áp từng swap (theo thứ tự) lên route hiện tại: với mỗi (a, b) thực hiện hoán đổi phần tử tại vị trí a và b .
3. Đảm bảo sau khi áp swap, route vẫn là một hoán vị hợp lệ (không xuất hiện đỉnh trùng lặp) — thao tác swap nguyên thủy luôn giữ tính hợp lệ vì chỉ hoán đổi vị trí giữa hai thành phần.

Genetic Algorithm (GA) GA là thuật toán truyền thống dùng so sánh với ACO và PSO. GA thường thao tác trên biểu diễn nhiễm sắc (genotype) — ví dụ tọa độ thực của các thành phố hoặc hoán vị — và dùng các toán tử lai ghép (crossover), đột biến (mutation) để tìm kiếm. Trong đề tài này, GA có thể nhận tọa độ thực của thành phố làm đầu vào, sau đó chuyển sang ma trận khoảng cách để so sánh với ACO/PSO.

3.1.2 Knapsack Problem (KP)

Giới thiệu chung

Knapsack Problem (KP) là một bài toán tối ưu tổ hợp kinh điển. Phát biểu 0–1 Knapsack chuẩn như sau: cho n món đồ, mỗi món có trọng lượng $w_i > 0$ và giá trị $v_i > 0$, cùng một balo có sức chứa tối đa W . Ta cần chọn một tập con các món sao cho:

$$\max_{x \in \{0,1\}^n} f(x) = \sum_{i=1}^n v_i x_i \quad \text{thoả} \quad \sum_{i=1}^n w_i x_i \leq W,$$

trong đó $x_i = 1$ nghĩa là chọn món i , $x_i = 0$ là không chọn.

Bài toán 0–1 KP thuộc lớp NP-hard và là mô hình trừu tượng cho nhiều bài toán phân bổ tài nguyên: lựa chọn danh mục dự án dưới ràng buộc ngân sách, tải hàng lên xe với giới hạn tải trọng, chọn tập đặc trưng (feature subset selection) có chi phí, v.v.

Trong phạm vi đồ án, KP được dùng như một bài toán rác đại diện, có không gian nghiệm $\{0,1\}^n$ và landscape nhiều cực trị cục bộ, giúp đánh giá khả năng tìm kiếm trên không gian tổ hợp của các thuật toán.

Những thuật toán áp dụng

- **Firefly Algorithm (FA) rời rạc:** mỗi con đom đóm mã hoá một vector nhị phân $x \in \{0,1\}^n$; các phép “di chuyển” được cài bằng operator nhị phân (flip bit có điều khiển, mask, crossover đơn giản) kết hợp thủ tục *repair* để xử lý nghiệm vi phạm ràng buộc $\sum w_i x_i \leq W$.
- **Genetic Algorithm (GA):** vector nhị phân là nhiễm sắc thể; dùng crossover một điểm hoặc hai điểm và mutation bit-flip. Ràng buộc được xử lý bằng phạt (penalty) trong hàm fitness hoặc bằng repair heuristic.
- **Hill Climbing (HC) và Simulated Annealing (SA):** định nghĩa lân cận bằng các phép flip đơn hoặc swap hai bit x_i, x_j ; HC/SA cho KP cung cấp baseline local search đơn giản để so sánh với FA/GA.

3.1.3 Graph Coloring (GC)

Giới thiệu chung

Graph Coloring (GC) là bài toán tô màu đồ thị: cho một đồ thị vô hướng $G = (V, E)$, mục tiêu là gán màu (số nguyên) cho mỗi đỉnh sao cho hai đỉnh kề nhau không trùng màu. Bài toán tối ưu kinh điển là tìm *chromatic number* $\chi(G)$ — số màu ít nhất cần dùng.

Có thể phát biểu dạng tối ưu:

$$\min_{c:V \rightarrow \{1, \dots, k\}} k \quad \text{s.t.} \quad c(u) \neq c(v), \quad \forall (u, v) \in E.$$

Bài toán quyết định “đồ thị có tô được bằng $\leq k$ màu hay không” là NP-complete với hầu hết $k \geq 3$, và bài toán tìm $\chi(G)$ là NP-hard. GC xuất hiện trong nhiều ứng dụng: lập lịch thi, xếp ca kíp, phân bổ tần số, register allocation trong compiler, v.v.

Trong đồ án, GC đóng vai trò một bài toán rời rạc có cấu trúc đồ thị rõ ràng, giúp đánh giá khả năng xử lý ràng buộc cứng (hard constraints) của các thuật toán heuristic.

Những thuật toán áp dụng

- **Genetic Algorithm (GA)**: mỗi cá thể là một gán màu $c : V \rightarrow \{1, \dots, k\}$; fitness có thể định nghĩa theo số cạnh vi phạm (hai đầu cùng màu) cộng với penalty theo số màu sử dụng.
- **Hill Climbing (HC) và Simulated Annealing (SA)**: lân cận được xây bằng thao tác đổi màu một đỉnh, hoặc swap màu giữa hai đỉnh; SA đặc biệt phù hợp vì việc cho phép nghiệm tạm thời xấu hơn giúp thoát bẫy khi nhiều cạnh xung đột.
- **Firefly Algorithm (FA) rời rạc (tuỳ cấu hình nhóm)**: biểu diễn mỗi nghiệm bằng vector màu và thiết kế operator di chuyển tương tự local search (đổi màu ưu tiên các đỉnh xung đột).

3.2 Các bài toán liên tục

Để đánh giá hiệu quả của các thuật toán metaheuristic trên không gian liên tục, nhóm sử dụng một bộ hàm chuẩn (benchmark functions) có tính chất đa dạng về độ lồi, số cực trị và mức độ “gai góc” của landscape. Các hàm này đều là bài toán tối thiểu hóa, với nghiệm tối ưu toàn cục và giá trị tối ưu đã biết trước, thuận tiện cho việc so sánh.

3.2.1 Sphere function

Giới thiệu chung

Sphere là hàm chuẩn đơn giản nhất, lồi, trơn, tách biến, thường dùng làm “bài kiểm tra sanity” cho thuật toán. Dạng d -chiều:

$$f(x) = \sum_{i=1}^d x_i^2, \quad x_i \in [a, b],$$

với nghiệm tối ưu toàn cục tại $x^* = 0$, $f(x^*) = 0$. Miền thường dùng là $[-5.12, 5.12]^d$ hoặc $[-100, 100]^d$.

Hàm này unimodal (chỉ có một cực tiểu toàn cục), convex, và separable. Thuật toán hợp lý phải hội tụ rất nhanh trên Sphere; nếu không, gần như chắc rằng việc cài đặt hoặc tham số đang có vấn đề.

Những thuật toán áp dụng

Trong đồ án, Sphere được dùng để:

- kiểm tra khả năng hội tụ cơ bản của Firefly Algorithm (FA) trên landscape lồi, trơn;
- so sánh với các baseline như Hill Climbing (HC), Simulated Annealing (SA) và Genetic Algorithm (GA) với mã hoá thực.

3.2.2 Rosenbrock function

Giới thiệu chung

Rosenbrock (Rosenbrock's valley/banana function) là hàm chuẩn nổi tiếng vì có một “thung lũng parabol hẹp” chứa nghiệm tối ưu, khiến việc hội tụ trở nên khó dù hàm chỉ có một cực tiểu toàn cục. Dạng d -chiều:

$$f(x) = \sum_{i=1}^{d-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2],$$

với nghiệm tối ưu toàn cục tại $x^* = (1, 1, \dots, 1)$, $f(x^*) = 0$. Miền thường dùng là $x_i \in [-5, 10]$.

Landscape của Rosenbrock có thung lũng dài, hẹp và cong; việc tìm được thung lũng khá dễ, nhưng “bờ” dọc thung lũng đến điểm tối ưu thì khó, đặc biệt với thuật toán chỉ biết gradient cục bộ hoặc có bước nhảy cố định.

Những thuật toán áp dụng

3.2.3 Rastrigin function

Giới thiệu chung

Rastrigin là hàm chuẩn đa cực trị (multimodal) kinh điển, được dùng để kiểm tra khả năng thoát bẫy cục bộ. Dạng d -chiều:

$$f(x) = 10d + \sum_{i=1}^d [x_i^2 - 10 \cos(2\pi x_i)], \quad x_i \in [-5.12, 5.12].$$

Nghiệm tối ưu toàn cục tại $x^* = 0$, $f(x^*) = 0$.

Hàm có rất nhiều cực tiểu cục bộ phân bố đều đặn; landscape “gợn sóng” dày đặc khiến các thuật toán dễ bị kẹt nếu kiểm soát khám phá/khuếch tán không tốt.

Những thuật toán áp dụng

Trong đồ án, Rastrigin là bài toán liên tục trọng tâm để đánh giá FA:

- **FA**: kiểm tra mức độ nhạy với tham số $(\alpha, \beta_0, \gamma)$, khả năng tránh hội tụ sớm và hiệu quả khi tăng số chiều.
- **HC & SA**: cung cấp baseline cho local search; HC thường bị kẹt ngay khi chạm vào một cực tiểu cục bộ, SA có thể thoát bẫy nhưng đòi hỏi lịch nhiệt độ phù hợp.
- **GA**: so sánh metaheuristic quần thể truyền thống với FA trên landscape đa cực trị.

3.2.4 Ackley function

Giới thiệu chung

Ackley là một hàm chuẩn đa cực trị khác, có vùng biên ngoài tương đối phẳng và một “hố sâu” ở trung tâm. Dạng d -chiều thường dùng:

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

với các giá trị khuyến nghị $a = 20$, $b = 0.2$, $c = 2\pi$, và miền điển hình $x_i \in [-32.768, 32.768]$. Nghiệm tối ưu toàn cục tại $x^* = 0$, $f(x^*) = 0$.

Ackley kết hợp một thành phần radial (phụ thuộc $\|x\|_2$) và một thành phần dao động cosine, tạo ra nhiều cực trị cục bộ nhưng có vùng biên khá “êm”; vì vậy rất hữu ích để kiểm tra xem thuật toán có bị lạc trong vùng bằng phẳng hay không.

Những thuật toán áp dụng

- **FA**: dùng để đánh giá khả năng xử lý landscape có vùng phẳng lớn và nhiều bẫy cục bộ.
- **HC & SA**: kiểm tra mức độ dễ bị kẹt trên các local minima của Ackley, đặc biệt với cấu hình nhiệt độ/lân cận khác nhau.
- **GA**: giúp so sánh hiệu năng giữa FA và một metaheuristic quần thể tiêu chuẩn trên một hàm đa cực trị khác với Rastrigin.

4 Ant Colony Optimization

4.1 Giới thiệu thuật toán

Ant colony optimization (tối ưu hóa đàn kiến) là một thuật toán heuristic dựa trên ý tưởng về cách kiếm ăn của loài kiến, một loài sinh vật có ý thức bầy đàn rất cao và có khả năng phối hợp theo bầy rất hiệu quả cho các công việc như kiếm ăn, xây tổ,...

4.1.1 Ý tưởng thuật toán

Các nhà sinh vật học đã chỉ ra rằng một số loài kiến có khả năng tìm ra đường đi ngắn nhất giữa tổ và nguồn thức ăn. Chúng làm điều này dựa trên một cơ chế là stigmergy (liên lạc qua môi trường):

- **Pheromone**: một tín hiệu hóa học do con đi trước thải ra, giúp con đi sau chọn đường có nồng độ cao hơn (con đường tốt hơn).
- Những con đi trên đường càng ngắn thì quay về tổ nhanh hơn. Con nào hoàn thành con đường nhanh hơn thì pheromone lưu lại càng nhiều trên con đường đó.
- **Autocatalytic (tự xúc tác)**: nồng độ pheromone càng cao thì càng thu hút nhiều con kiến di chuyển trên đường đó, làm con đường càng tích lũy nhiều pheromone. Tới một lúc nào đó, toàn bộ đàn kiến sẽ hội tụ vào con đường ngắn nhất.

4.1.2 Lịch sử thuật toán

Nguồn gốc của ACO bắt nguồn từ các nghiên cứu về hành vi của côn trùng xã hội, đặc biệt là công trình của các nhà sinh vật học như Jean-Louis Deneubourg, người đã cung cấp cảm hứng cho công việc này.

- **Những năm 1990**: Những nỗ lực đầu tiên nhằm chuyển hóa hành vi của kiến thành thuật toán máy tính xuất hiện vào đầu những năm 1990.
- **Ant System (AS)**: Thuật toán ACO đầu tiên có tên là Ant System (AS). Nó được định nghĩa bởi Marco Dorigo trong luận án tiến sĩ của ông tại Politecnico di Milano (Ý), với sự hợp tác của Alberto Colomi và Vittorio Maniezzo.
- **1991-1996**: Bài báo chuyên san đầu tiên về Ant System được nộp vào năm 1991, nhưng phải đến năm 1996 mới được xuất bản. Nghiên cứu về ACO bắt đầu phổ biến và thu hút sự quan tâm nhanh chóng sau khi bài báo này ra đời.
- **Phát triển các biến thể**: Sau AS, một số biến thể thuật toán đã được phát triển để cải thiện hiệu suất, chẳng hạn như Ant-Q, Ant Colony System (ACS) và MAX-MIN AS.
- **Hình thành "Siêu heuristic ACO"**: Thuật ngữ "ACO metaheuristic" (siêu heuristic ACO) đã được đề xuất (bởi Dorigo và Di Caro vào năm 1999) như một khung sườn chung (common framework) để bao quát các thuật toán và ứng dụng dựa trên ý tưởng này.

4.1.3 Những ứng dụng

Thuật toán ACO là một thuật toán phổ biến và được ứng dụng rộng rãi nhờ tính linh hoạt và khả năng hội tụ tốt. Nó được ứng dụng nhiều trong các bài toán NP-hard (không thể tìm được đáp án chính xác trong thời gian đa thức):

- Bài toán định tuyến: người giao hàng (TSP), sắp xếp thứ tự có ưu tiên (SOP),...
- Bài toán gán/ phân công: tô màu bản đồ (GCP), lập thời khóa biểu (UCTP),...
- Bài toán lập lịch
- Bài toán tập con: định tuyến mạng (thuật toán AntNet), bao phủ tập hợp (SCP),...

4.2 Cơ sở toán học

Cơ sở toán học của bài toán này có nhiều biến thể trên nhiều bài toán tối ưu hóa khác nhau, tuy nhiên chúng có điểm chung là cần được chuyển thành bài toán tìm đường đi ngắn nhất trong đồ thị có trọng số.

Cơ sở toán học của thuật toán Tối ưu hóa Dàn kiến (ACO) chủ yếu xoay quanh hai cơ chế: (1) quy tắc xác suất để kiến nhân tạo "xây dựng" giải pháp và (2) quy tắc cập nhật pheromone để "học hỏi" từ kinh nghiệm.

4.2.1 Quy tắc Lựa chọn Đường đi (Xây dựng Giải pháp)

Đây là công thức cốt lõi quyết định cách một con kiến nhân tạo k chọn đỉnh tiếp theo khi nó đang ở đỉnh i . Xác suất p_{ij}^k để kiến k di chuyển từ đỉnh i đến đỉnh j là:

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta} \quad \text{nếu } j \in \mathcal{N}_i^k$$

Trong đó:

- τ_{ij} (**Pheromone**): Lượng pheromone nhân tạo trên cạnh nối (i, j) . Đây là thông tin "được" (bộ nhớ dài hạn) về mức độ mong muốn của cạnh này.
- η_{ij} (**Thông tin Heuristic**): Thông tin heuristic có sẵn (a priori) về cạnh (i, j) . Trong các bài toán tối ưu hóa, nó thường được định nghĩa là $1/c_{ij}$ (nghịch đảo của chi phí), nghĩa là cạnh có chi phí thấp sẽ hấp dẫn hơn.
- α và β (**Tham số**): Hai tham số này kiểm soát tầm quan trọng tương đối của pheromone (kinh nghiệm đã học) so với thông tin heuristic (kiến thức có sẵn).
- \mathcal{N}_i^k (**Tập lân cận khả thi**): Tập hợp các đỉnh mà kiến k chưa đi qua (để đảm bảo tính hợp lệ của đường đi).

4.2.2 Quy tắc Cập nhật Pheromone (Học tập)

Sau khi tất cả m con kiến đã hoàn thành việc xây dựng các đường đi, hệ thống sẽ cập nhật lượng pheromone. Quá trình này có hai phần: bay hơi và lắng đọng.

Bay hơi Pheromone (Evaporation)

Đầu tiên, một phần pheromone trên *tất cả* các cạnh sẽ "bay hơi" theo công thức:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij}$$

- ρ (**Hệ số bay hơi**): Một tham số $0 < \rho \leq 1$.

- **Mục đích:** Cơ chế này giúp thuật toán "quên đi" các lựa chọn cũ, không tốt và tránh việc bị "kẹt" (stagnation) vô thời hạn tại một giải pháp dưới tối ưu.

Lắng đọng Pheromone (Deposit)

Tiếp theo, mỗi con kiến sẽ "lắng đọng" pheromone trên những cạnh mà nó đã đi qua, dựa trên chất lượng giải pháp (đường đi) mà nó tìm được.

Tổng lượng pheromone được thêm vào là:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^m \Delta\tau_{ij}^k$$

Trong đó, $\Delta\tau_{ij}^k$ là lượng pheromone mà kiến k lắng đọng lên cạnh (i, j) , được định nghĩa là:

$$\Delta\tau_{ij}^k = \begin{cases} 1/C^k & \text{nếu cạnh } (i, j) \text{ thuộc đường đi } T^k \text{ của kiến } k \\ 0 & \text{nếu không} \end{cases}$$

- C^k (**Chất lượng giải pháp**): Là tổng chi phí của đường đi T^k mà kiến k đã thực hiện.

- **Ý nghĩa:** Một đường đi càng tốt (chi phí C^k càng nhỏ) thì lượng pheromone $1/C^k$ lắng đọng trên các cạnh của nó càng lớn. Điều này làm tăng xác suất để các con kiến trong tương lai chọn lại những cạnh này.

4.3 Triển khai thuật toán

4.3.1 Cách hoạt động của thuật toán

Algorithm 7 Mã giả Siêu heuristic ACO

```

1: while chưa đạt điều kiện dừng do
2:   for số_kiến do
3:     tạo_giải_pháp()
4:   end for
5:   so_sánh_giải_pháp()
6:   cập_nhật_hệ_số_pheromone()
7: end while
8: return giải pháp tốt nhất

```

Chi tiết quá trình:

1. Ở bước đầu tiên, mỗi "con kiến" sẽ chọn đường đi cho mình sao cho thỏa mãn đề bài. Để chọn được cạnh để đi trong mỗi bước, con kiến đó sẽ dựa vào mức độ pheromone của mỗi cạnh mà nó có thể đi và chọn ngẫu nhiên có trọng số trong các cạnh khả dĩ đó (theo công thức trong 2.1).
2. Ở bước thứ 2, các giải pháp sẽ được so sánh với nhau, và giải pháp tốt nhất (tất nhiên so với cả các giải pháp ở vòng lặp trước) sẽ được chọn. Nếu đó là vòng lặp cuối cùng, giải pháp đó sẽ là kết quả của thuật toán.
3. Ở bước tiếp theo, hệ số pheromone sẽ được cập nhật trên từng cạnh của đường đi (theo công thức trong 2.2). Hệ quả là, con đường nào càng ngắn thì các cạnh trên con đường đó sẽ lưu lại càng nhiều pheromone, dẫn đến xác suất chọn cạnh đó trong những lần sau sẽ cao hơn.

Quá trình trên được lặp lại với số vòng lặp nhất định (từ 200-500 tùy vào số cạnh của đồ thị), hoặc cho tới khi hội tụ.

Một cách trực quan, quá trình này hội tụ về các chu trình ngắn bởi vì mỗi con kiến để lại càng nhiều pheromone trên các cạnh thuộc chu trình của nó nếu chu trình đó càng ngắn. Và, khi pheromone cũ mờ dần theo thời gian, đồng thời các con kiến mới ưu tiên những cạnh có nhiều pheromone hơn, thì các chu trình mới sẽ có xu hướng ngày càng ngắn hơn.

Điều quan trọng là, vì mỗi con kiến chọn bước đi tiếp theo một cách ngẫu nhiên, nên mặc dù chúng sẽ luôn có xu hướng chọn các ứng cử viên có nhiều pheromone nhất, chúng cũng sẽ có một xác suất đáng kể (không thể bỏ qua) để chọn một cạnh khác và đi khám phá. Nếu việc khám phá đó dẫn đến một chu trình tổng thể tốt hơn, con kiến đó sẽ "báo" cho các con kiến tương lai về chu trình này bằng cách để lại nhiều pheromone hơn nữa.

4.3.2 Triển khai kỹ thuật

- Ngôn ngữ: Python
- Các thư viện sử dụng: numpy, random, matplotlib
- Các class:
 - Class ACO_Solver: chứa các hàm chính để giải bài toán và kiểm thử trên tập dữ liệu cho trước.
 - Class Graph: cấu trúc đồ thị dưới dạng ma trận khoảng cách.
- Chi tiết triển khai ở [github repository](#) (nằm trong phần phụ lục).

5 Particle swarm optimization

5.1 Giới thiệu thuật toán

Particle swarm optimization (tối ưu hóa bầy đàn) là một thuật toán heuristic mạnh mẽ được lấy cảm hứng bởi hành vi bầy đàn được quan sát trong tự nhiên như đàn cá và chim. PSO là một mô phỏng của một hệ thống xã hội được đơn giản hóa.

5.1.1 Ý tưởng thuật toán

Ý đồ của thuật toán PSO là mô phỏng lại cách di chuyển phức tạp của đàn chim. Trong tự nhiên, tầm nhìn của một con chim đơn lẻ là bị giới hạn. Tuy nhiên, có nhiều hơn một con chim cho phép đàn chim ý thức được vị trí của chúng trên một không gian rộng lớn hơn.

Không gian tìm kiếm thức ăn lúc này là toàn bộ không gian ba chiều. Tại thời điểm bắt đầu tìm kiếm cả đàn bay theo một hướng thường là ngẫu nhiên.. Tuy nhiên sau một thời gian tìm kiếm một số cá thể trong đàn bắt đầu tìm ra được nơi có chứa thức ăn.

Tùy theo số lượng thức ăn vừa tìm kiếm, mà cá thể gửi tín hiệu đến các cá thể khác đang tìm kiếm ở vùng lân cận. Tín hiệu này lan truyền trên toàn quần thể.

Dựa vào thông tin nhận được mỗi cá thể sẽ điều chỉnh hướng bay và vận tốc theo hướng về nơi có nhiều thức ăn nhất.

Cơ chế truyền tin như vậy thường được xem như là một kiểu hình của trí tuệ bầy đàn. Cơ chế này giúp cả đàn chim tìm ra nơi có nhiều thức ăn nhất trên không gian tìm kiếm vô cùng rộng lớn.

5.1.2 Lịch sử thuật toán

Nghiên cứu sớm về ứng dụng tập tính bầy đàn trong thuật toán là mô hình "Boids" của Craig Reynolds. Reynolds đã chỉ ra rằng hành vi bầy đàn phức tạp có thể được mô phỏng chỉ bằng ba quy tắc đơn giản mà mỗi cá thể tuân theo (tách biệt, thẳng hàng, và gắn kết).

Thuật toán PSO được giới thiệu lần đầu tiên vào năm 1995 bởi hai nhà nghiên cứu: James Kennedy và Russell Eberhart, dựa trên công trình của Reynolds. PSO nhanh chóng trở nên phổ biến vì nó đơn giản hơn đáng kể so với các thuật toán tiến hóa khác như Giải thuật Di truyền (Genetic Algorithms - GA).

5.1.3 Những ứng dụng

Thuật toán PSO được ứng dụng rộng rãi trong nhiều lĩnh vực nhờ độ hiệu quả, tính đơn giản và linh hoạt của nó. Nó được ứng dụng trong nhiều lĩnh vực khác nhau:

- Chăm sóc sức khỏe: Chẩn đoán thông minh, Phát hiện và phân loại bệnh, Phân đoạn hình ảnh y tế,...
- Môi trường: Giám sát chất lượng nước, giám sát lũ lụt,...
- Công nghiệp: Điều phối sản lượng điện (economic dispatch), Lập lịch tải điện, Tối ưu hóa Lưới điện thông minh,...
- Thương mại: Dự đoán chi phí và giá cả, đánh giá rủi ro,...

5.2 Cơ sở toán học của thuật toán

Cơ sở toán học của thuật toán PSO được thể hiện qua hai phương trình cốt lõi: Cập nhật Vận tốc và Cập nhật Vị trí. Cơ sở toán học này mô tả cách mỗi "hạt" (tương ứng với mỗi cá thể chim trong đàn) trong bầy đàn điều chỉnh chuyển động của nó qua không gian tìm kiếm.

5.2.1 Phương trình Cập nhật Vận tốc (Velocity Update)

Phương trình này tính toán vận tốc mới (véc-tơ di chuyển) cho một hạt ở vòng lặp (thế hệ) tiếp theo. Vận tốc mới được quyết định bởi ba thành phần:

- **Quán tính (Inertia):** Vận tốc hiện tại của hạt, giữ cho nó di chuyển theo hướng cũ.
- **Thành phần Nhận thức (Cognitive Component):** Hướng di chuyển về phía vị trí tốt nhất cá nhân (pbest) mà hạt đó đã từng đạt được.
- **Thành phần Xã hội (Social Component):** Hướng di chuyển về phía vị trí tốt nhất toàn bầy (gbest) mà bất kỳ hạt nào trong bầy đã từng đạt được.

Công thức toán học như sau:

$$\mathbf{v}_i(t+1) = w \cdot \mathbf{v}_i(t) + c_1 \cdot r_1 \cdot (\mathbf{pbest}_i - \mathbf{x}_i(t)) + c_2 \cdot r_2 \cdot (\mathbf{gbest} - \mathbf{x}_i(t)) \quad (3)$$

Trong đó:

- $\mathbf{v}_i(t+1)$: Là vận tốc mới (dự kiến) của hạt i tại vòng lặp $t+1$.
- w : **Trọng số quán tính (Inertia weight)**.
- $\mathbf{v}_i(t)$: Vận tốc hiện tại của hạt i tại vòng lặp t .
- c_1, c_2 : **Hệ số học tập (Learning coefficients)** (hằng số gia tốc).

- r_1, r_2 : Là hai số ngẫu nhiên được tạo ra trong khoảng $[0, 1]$.
- \mathbf{pbest}_i : Vị trí tốt nhất mà cá nhân hạt i đã tìm thấy.
- \mathbf{gbest} : Vị trí tốt nhất mà cả bầy đàn đã tìm thấy.
- $\mathbf{x}_i(t)$: Vị trí hiện tại của hạt i tại vòng lặp t .

5.2.2 Phương trình Cập nhật Vị trí (Position Update)

Sau khi tính toán vận tốc mới, hạt sẽ sử dụng vận tốc đó để di chuyển đến một vị trí mới trong không gian tìm kiếm.

Công thức toán học như sau:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \quad (4)$$

Trong đó:

- $\mathbf{x}_i(t+1)$: Là vị trí mới của hạt i tại vòng lặp $t+1$.
- $\mathbf{x}_i(t)$: Vị trí hiện tại của hạt i tại vòng lặp t .
- $\mathbf{v}_i(t+1)$: Vận tốc mới vừa được tính toán từ phương trình (1).

5.3 Triển khai thuật toán

5.3.1 Cách hoạt động của thuật toán

Thuật toán PSO mô phỏng một bầy đàn "bay" qua không gian tìm kiếm để tìm giải pháp tối ưu.

Algorithm 8 Thuật toán Tối ưu hóa Bầy đàn Hạt (PSO)

```
1: [PHẦN 1: KHỞI TẠO]
2: Khởi tạo một bầy đàn (population) gồm  $N$  hạt.
3: for mỗi hạt  $i$  (từ 1 đến  $N$ ) do
4:   Khởi tạo vị trí ban đầu  $\mathbf{x}_i$  (ngẫu nhiên).
5:   Khởi tạo vận tốc ban đầu  $\mathbf{v}_i$  (bằng 0).
6:   Tính giá trị thích nghi (fitness)  $f(\mathbf{x}_i)$ .
7:    $\mathbf{pbest}_i \leftarrow \mathbf{x}_i$ 
8: end for
9:  $\mathbf{gbest} \leftarrow$  hạt có giá trị thích nghi tốt nhất trong bầy.
10: [PHẦN 2: VÒNG LẶP CHÍNH]
11: while chưa đạt điều kiện dừng (ví dụ:  $t < T_{\max}$ ) do
12:   for mỗi hạt  $i$  (từ 1 đến  $N$ ) do
13:     Cập nhật Vận tốc theo phương trình (1)
14:      $\mathbf{v}_i(t+1)$ 
15:     Cập nhật Vị trí theo phương trình (2)
16:      $\mathbf{x}_i(t+1) \leftarrow \mathbf{x}_i(t) + \mathbf{v}_i(t+1)$ 
17:     Dánh giá vị trí mới
18:     Tính  $f(\mathbf{x}_i(t+1))$ 
19:     Cập nhật pbest (bộ nhớ cá nhân)
20:     if  $f(\mathbf{x}_i(t+1))$  tốt hơn  $f(\mathbf{pbest}_i)$  then
21:        $\mathbf{pbest}_i \leftarrow \mathbf{x}_i(t+1)$ 
22:     end if
23:     Cập nhật gbest (bộ nhớ bầy đàn)
24:     if  $f(\mathbf{x}_i(t+1))$  tốt hơn  $f(\mathbf{gbest})$  then
25:        $\mathbf{gbest} \leftarrow \mathbf{x}_i(t+1)$ 
26:     end if
27:   end for
28: end while
29: TRẢ VỀ:  $\mathbf{gbest}$  (Vị trí của giải pháp tốt nhất)
```

5.3.2 Triển khai kỹ thuật

- **Ngôn ngữ:** Python
- **Các thư viện sử dụng:** numpy, random, matplotlib
- **Các class:**
 - Class **PSO_Solver**: chứa các hàm chính để giải bài toán và kiểm thử trên tập dữ liệu cho trước.
 - Class **Agent**: triển khai một giải pháp tiềm năng cho bài toán.
- Chi tiết triển khai ở github repository (nằm trong phần phụ lục).

6 Artificial Bee Colony

6.1 Giới thiệu thuật toán

[KA09]

Thuật toán đàm ong nhân tạo (Artificial Bee Colony - ABC) được đề xuất bởi Derviş Karaboğa (2005), mô phỏng hành vi tìm kiếm và chia sẻ thông tin về nguồn mật hoa của đàm ong mật trong tự nhiên. Mục tiêu là tìm lời giải tốt nhất cho bài toán tối ưu hóa bằng cách mô phỏng quá trình ong tìm mật. Trong đồ án này, bài toán được sử dụng là tìm giá trị nhỏ nhất của hàm số 2 biến.

6.2 Ý tưởng thuật toán và nguyên lý hoạt động

Trong tự nhiên, ong mật chia thành 3 nhóm: ong thợ, ong quan sát và ong trinh sát. Nhiệm vụ của mỗi nhóm như sau:

- Ong thợ tìm kiếm thức ăn xung quanh nguồn mật bằng trí nhớ, đồng thời chia sẻ thông tin về những nguồn mật này cho ong quan sát.
- Ong quan sát có xu hướng chọn nguồn mật tốt từ những nguồn mà ong thợ tìm thấy. Nguồn mật có chất lượng cao hơn (độ thích nghi cao) sẽ có nhiều khả năng được ong quan sát lựa chọn hơn.
- Khi nguồn mật đã cạn, ong do thám sẽ bỏ nguồn mật hiện tại và tìm kiếm nguồn mới ngẫu nhiên.

Ong thợ và ong quan sát tìm kiếm các nguồn mật xung quanh tổ. Những con ong thợ lưu trữ thông tin về nguồn mật và chia sẻ thông tin với những con ong quan sát. Số lượng nguồn mật bằng với số lượng ong thợ. Một con ong thợ sau khi khai thác một nguồn mật lần nhất định mà chất lượng nguồn mật của chúng không thể cải thiện, nó sẽ trở thành ong trinh sát và bỏ nguồn mật cũ đi. Tương tự trong bài toán tối ưu hóa, số lượng nguồn mật trong thuật toán ABC đại diện cho số lượng chất lượng nguồn mật trong quần thể (càng nhiều mật thì chất lượng mật càng tốt và ngược lại). Nói cách khác, nếu các con ong tìm được một nguồn mật tốt, đây có khả năng sẽ là điểm tối ưu và sẽ có xu hướng thu hút các con ong khác tới khai thác.

6.3 Giải thích cơ chế hoạt động bằng công cụ toán học

6.3.1 Ký hiệu và các tham số chính

- $f(x)$: Hàm mục tiêu cần tối ưu hóa
- SN : Số lượng ong thợ (cũng bằng số nguồn mật)
- **limit**: Giới hạn số lần không cải thiện trước khi ong thợ trở thành ong do thám (để đi tìm nguồn mật tốt hơn), tránh trường hợp các con ong "quanh quẩn" ở điểm tối ưu cục bộ, nơi có nguồn mật tốt hơn so với các nguồn xung quanh nhưng không phải nguồn mật tốt nhất có thể khai thác.
- **maxCycle**: Số vòng lặp tối đa, nếu sau maxCycle lần khai thác và tìm kiếm, các con ong vẫn tập trung tại một nguồn mật nào đó, điểm này sẽ được xác định là điểm tối ưu, dừng thuật toán.

6.3.2 Nguyên lý

Thuật toán ABC gồm 4 giai đoạn chính như sau: [Kum13]

1. Khởi tạo quần thể

Ban đầu tất cả SN ong trong quần thể sẽ là ong do thám, khi đó tương ứng SN nguồn mật sẽ được sinh ra ngẫu nhiên trong không gian tìm kiếm điểm tối ưu. Mỗi nguồn mật

(ký hiệu bởi x_m) là một vector có D chiều, là số chiều của không gian tìm kiếm, được sinh bởi công thức:

$$x_m = l_i + \text{rand}(0, 1) \times (u_i - l_i)$$

với u_i và l_i lần lượt là cận trên và cận dưới của không gian tìm kiếm, $\text{rand}(0, 1)$ là một số ngẫu nhiên thuộc đoạn $[0, 1]$

2. Pha ong thợ

Từ nguồn mật hiện tại x_i , ong thợ bay đến một nguồn mật v_i khác trong khu vực lân cận. So sánh giá trị của $f(x_i)$ và $f(v_i)$, nếu nguồn mật chất lượng hơn (giá trị hàm đạt được tại v_i tốt hơn) sẽ thay thế cho nguồn cũ. Giá trị của mỗi chiều của v_i được sinh bởi công thức:

$$v_{ij} = x_{ij} + \phi_m (x_{ij} - x_{kj})$$

Trong đó $j = \overline{1..D}$, ϕ_m là một số ngẫu nhiên thuộc đoạn $[-1, 1]$

3. Pha ong quan sát

Mỗi ong quan sát chọn nguồn mật dựa trên xác suất được tính bởi phương trình:

$$p_i = \frac{\text{fit}_i}{\sum_{j=1}^{SN} \text{fit}_j}$$

Với fit_i là độ phù hợp của từng nguồn mật, được tính theo công thức:

$$\text{fit}_i = \begin{cases} \frac{1}{1+f(x_i)}, & f(x_i) \geq 0 \\ 1 + |f(x_i)|, & f(x_i) < 0 \end{cases}$$

Sau khi chọn, ong quan sát cũng tạo nguồn mật mới theo công thức tương tự như ở pha ong thợ. Nếu tốt hơn, cập nhật nguồn mật tương ứng.

4. Pha ong do thám

Nếu một nguồn mật không được cải thiện sau $limit$ lần, ong thợ sẽ bỏ nguồn mật này đi và trở thành ong do thám. Sau đó, chúng sẽ tạo nguồn mật mới ngẫu nhiên bằng công thức như ở pha khởi tạo quần thể:

5. Lặp lại

Lặp lại các pha 2 đến 4 cho đến khi đạt số vòng lặp $maxCycle$ hoặc đạt tiêu chí hội tụ (không còn cải thiện)

6.3.3 Mã giả (Pseudo-code) thuật toán ABC

[IEH⁺25]

```

1: Khởi tạo quần thể gồm  $SN$  nguồn thức ăn  $x_i$  ( $i = 1, 2, \dots, SN$ )
2: Dánh giá độ thích nghi (fitness) của từng nguồn thức ăn
3: repeat
4:   for mỗi ong thợ do
5:     Sinh ra ứng viên mới  $v_i$  lân cận  $x_i$ 
6:     Dánh giá độ thích nghi của  $v_i$ 
7:     Lựa chọn giữa  $x_i$  và  $v_i$ 
8:   end for
9:   for mỗi ong quan sát do
10:    Chọn nguồn thức ăn  $x_i$  với xác suất tỉ lệ thuận với độ thích nghi của nó
11:    Sinh ra ứng viên mới  $v_i$ 
12:    Lựa chọn giữa  $x_i$  và  $v_i$ 
13:  end for
14:  Nếu một nguồn thức ăn không được cải thiện sau một số vòng lặp giới hạn, thay thế bằng
      một nguồn ngẫu nhiên mới (giai đoạn ong do thám)
15:  Ghi nhớ nghiệm tốt nhất hiện tại
16: until thoả mãn điều kiện dừng

```

7 Firefly Algorithm

7.1 Giới thiệu

7.1.1 Ý tưởng cốt lõi

Thuật toán Đom Đóm (Firefly Algorithm, FA) là một metaheuristic lấy cảm hứng từ hành vi phát quang và bị hút lẫn nhau của đom đóm trong tự nhiên. Trong FA, mỗi nghiệm ứng viên là một “đom đóm” với *độ sáng* (brightness) tỉ lệ với *độ phù hợp* (fitness). Một đom đóm kém sáng sẽ di chuyển về phía đom đóm sáng hơn; cường độ *hấp dẫn* suy giảm theo khoảng cách. Thành phần *nhiều* (randomization) được thêm vào để tăng khả năng thoát bẫy cục bộ và thăm dò không gian nghiệm rộng hơn. Cơ chế này tạo nên sự cân bằng *thăm dò* (exploration) và *khai thác* (exploitation), phù hợp với các bài toán đa cực trị (multimodal) (Yang, 2009; Yang & He, 2013).

7.1.2 Lịch sử ngắn gọn

FA được đề xuất bởi Xin-She Yang vào giai đoạn 2008–2009 và nhanh chóng trở thành một trong các thuật toán tối ưu lấy cảm hứng tự nhiên tiêu biểu bên cạnh PSO, BA, Cuckoo Search, v.v. (Yang, 2009; Yang, 2010). Hướng phát triển gồm: FA tự điều chỉnh tham số, FA lai (hybrid) với tìm kiếm cục bộ, FA song song/đám mây, và các biến thể *rời rạc hoá* cho bài toán tổ hợp (Yang & He, 2013; Baykasoglu et al., 2014).

7.1.3 Ứng dụng tiêu biểu

FA được áp dụng rộng rãi trong: tối ưu hàm chuẩn (Rastrigin, Ackley, Rosenbrock), điều chỉnh siêu tham số mô hình học máy, chọn đặc trưng, định tuyến-lập lịch, và các bài toán tổ hợp như Knapsack/Flow-Shop sau khi rời rạc hoá (*binarization* hoặc *discretization*) (Yang, 2010; Yang & He, 2013). Điểm mạnh chính: (i) công thức cập nhật đơn giản; (ii) ít siêu tham số; (iii) linh hoạt để thích nghi với cả không gian liên tục và nhị phân.

7.2 Cơ sở toán học của thuật toán

Xét bài toán cực tiểu hoá không ràng buộc

$$\min_{x \in \mathbb{R}^d} f(x).$$

Tại vòng lặp t , cá thể i có vị trí $\mathbf{x}_i^{(t)} \in \mathbb{R}^d$. Gọi $r_{ij}^{(t)} = \|\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)}\|_2$ là khoảng cách giữa i và j .

Độ sáng (Brightness). Độ sáng của mỗi đom đóm được tính từ fitness:

$$I_i = -f(\mathbf{x}_i) \quad (\text{bài toán cực tiểu})$$

Đom đóm có fitness tốt hơn (giá trị f nhỏ hơn) sẽ sáng hơn.

Hấp dẫn suy giảm theo khoảng cách. Hàm hấp dẫn chuẩn trong FA:

$$\beta(r_{ij}^{(t)}) = \beta_0 \exp(-\gamma(r_{ij}^{(t)})^2),$$

trong đó $\beta_0 > 0$ là hấp dẫn tại $r = 0$, còn $\gamma > 0$ điều khiển mức suy giảm theo khoảng cách (Yang, 2009).

Quy tắc cập nhật (không gian liên tục). Đối với mỗi đom đóm i , chỉ di chuyển về phía các đom đóm sáng hơn:

$$\mathbf{x}_i^{(t+1)} = \mathbf{x}_i^{(t)} + \beta(r_{ij}^{(t)}) (\mathbf{x}_j^{(t)} - \mathbf{x}_i^{(t)}) + \alpha \boldsymbol{\varepsilon}_i^{(t)}.$$

khi $I_j > I_i$. Ở đây $\boldsymbol{\varepsilon}_i^{(t)}$ là nhiễu (thường lấy từ $\mathcal{U}[-\frac{1}{2}, \frac{1}{2}]^d$), và α là hệ số randomization. Nếu không có cá thể sáng hơn, nghiệm chủ yếu chịu nhiều thăm dò.

Xử lý biên. Sau cập nhật, sử dụng **chiếu (clipping)** về miền hợp lệ [lower_bound, upper_bound] cho từng chiều.

Rời rạc hoá cho Knapsack 0/1. Biểu diễn nghiệm bằng $\mathbf{b}_i^{(t)} \in \{0, 1\}^n$. Quy tắc di chuyển:

(a) **Di chuyển có hướng (Directed movement).** Với mỗi đom đóm j sáng hơn i :

1. Xác định tập vị trí khác biệt: $D = \{k \mid b_{i,k} \neq b_{j,k}\}$.
2. Chọn ngẫu nhiên **TỐI ĐA** m_{\max} vị trí từ D .
3. Lật bit tại các vị trí được chọn để $b_{i,k} \leftarrow b_{j,k}$.

(b) **Nhiều ngẫu nhiên.** Với xác suất α_{flip} , chọn ngẫu nhiên một vị trí và lật bit:

$$b_{i,k} \leftarrow 1 - b_{i,k}.$$

(c) **Sửa nghiệm vi phạm (Repair vs Penalty).** Benchmark hỗ trợ hai chiến lược xử lý ràng buộc:

- **Repair strategy:** Sau khi di chuyển, nếu nghiệm vi phạm ràng buộc sức chứa, loại dần các vật phẩm có tỷ số v_k/w_k thấp nhất (greedy removal) cho đến khi khả thi.
- **Penalty strategy:** Nghiệm vi phạm nhận penalty lớn trong fitness, cho phép exploration trong không gian infeasible.

Gợi ý lựa chọn tham số. β_0 lớn \Rightarrow khai thác cục bộ mạnh; γ nhỏ \Rightarrow hút tầm xa tăng thăm dò; α nên trong khoảng $[0.1, 0.5]$ để cân bằng exploration/exploitation. Kích thước quần thể N đủ để bao phủ không gian ban đầu, thường $N \in [20, 100]$ tuỳ độ khó/hàm mục tiêu (Yang, 2009; Yang, 2010).

7.3 Chi tiết triển khai thuật toán (Mã giả)

Algorithm 9 Thuật toán Dom Dom cho tối ưu liên tục

```

1: Input: HÀM  $f$ , MIỀN  $[\text{lb}, \text{ub}]^d$ ,  $n$  ĐƠM ĐÓM,  $T$  VÒNG LẶP,  $\alpha, \beta_0, \gamma$ .
2: KHỞI TẠO NGẪU NHIÊN  $\{\mathbf{x}_i\}_{i=1}^n$  TRONG MIỀN HỢP LỆ.
3: TÍNH FITNESS  $f(\mathbf{x}_i)$  VÀ ĐỘ SÁNG  $I_i = -f(\mathbf{x}_i)$ .
4: for  $t = 1$  to  $T$  do
5:   TÍNH MA TRẬN KHOẢNG CÁCH  $r_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|_2$ .
6:   for  $i = 1$  to  $n$  do
7:     for  $j = 1$  to  $n$  do
8:       if  $I_j > I_i$  then
9:          $\beta := \beta_0 \exp(-\gamma r_{ij}^2)$ 
10:         $\mathbf{x}_i := \mathbf{x}_i + \beta(\mathbf{x}_j - \mathbf{x}_i) + \alpha \varepsilon$ 
11:       end if
12:     end for
13:     CHIẾU  $\mathbf{x}_i$  VỀ  $[\text{lb}, \text{ub}]^d$ , CẬP NHẬT  $f(\mathbf{x}_i)$  VÀ  $I_i$ .
14:   end for
15:   GHI NHẬN NGHIỆM TỐT NHẤT  $\mathbf{x}^*$ ,  $f^*$ .
16: end for
17: Output:  $\mathbf{x}^*$ ,  $f^*$ .
```

Algorithm 10 Thuật toán Dom Đóm cho Knapsack 0/1

```
1: Input:  $\mathbf{v}, \mathbf{w}, C, n$  đom đóm,  $T$  vòng lặp,  $\alpha_{\text{flip}}$ ,  $m_{\max}$ , strategy  $\in \{\text{repair}, \text{penalty}\}$ .
2: Khởi tạo  $n$  nghiệm nhị phân  $\mathbf{b}_i \in \{0, 1\}^m$ .
3: Tính fitness  $f(\mathbf{b}_i)$  và độ sáng  $I_i$ .
4: for  $t = 1$  to  $T$  do
5:   for  $i = 1$  to  $n$  do
6:     for  $j = 1$  to  $n$  do
7:       if  $I_j > I_i$  then
8:          $D \leftarrow \{k \mid b_{i,k} \neq b_{j,k}\}$ 
9:         Chọn ngẫu nhiên TỐI DA  $m_{\max}$  vị trí từ  $D$ 
10:        Với mỗi vị trí được chọn:  $b_{i,k} \leftarrow b_{j,k}$ 
11:       end if
12:     end for
13:     if  $\text{rand}() < \alpha_{\text{flip}}$  then
14:       Chọn ngẫu nhiên vị trí  $k$  và lật:  $b_{i,k} \leftarrow 1 - b_{i,k}$ 
15:     end if
16:     if strategy == repair then
17:        $\mathbf{b}_i \leftarrow \text{GreedyRepair}(\mathbf{b}_i)$  {Loại items có  $v/w$  thấp}
18:     else
19:       Áp dụng penalty nếu vi phạm capacity
20:     end if
21:     Cập nhật  $f(\mathbf{b}_i), I_i$ .
22:   end for
23:   Ghi nhận nghiệm tốt nhất  $\mathbf{b}^*$ ,  $f^*$ .
24: end for
25: Output:  $\mathbf{b}^*$ , giá trị balo =  $-f^*$ .
```

7.4 Thuật toán dùng đếm so sánh

Để đặt Thuật toán Dom Đóm (FA) vào một bối cảnh hợp lý, chúng tôi so sánh với ba thuật toán metaheuristic cổ điển: Hill Climbing (HC), Simulated Annealing (SA) và Genetic Algorithm (GA). Tất cả đều được hiện thực trong cùng một khung mã Python, dùng chung: (i) giao diện bài toán, (ii) bộ sinh nghiệm khởi tạo, (iii) cách ghi log kết quả và (iv) các script phân tích/visualize.

Hill Climbing (HC). HC là baseline tham lam địa phương: giữ một nghiệm hiện tại, sinh lân cận và chỉ chấp nhận nghiệm tốt hơn. Khi không cải thiện sau một số bước, thực hiện restart.

Cấu hình đại diện:

- **Rastrigin (liên tục):** lân cận được sinh bằng perturbation Gaussian/Uniform trên từng chiều với bước nhảy cố định; dùng một số lượng lân cận cố định mỗi vòng lặp, sau đó chọn best improvement. Có cơ chế restart sau một số vòng không cải thiện.
- **Knapsack (0/1):** lân cận sinh bằng cách lật một số bit trong vector nhị phân (bit-flip). Restart khi bị kẹt quá lâu trong plateau.

Simulated Annealing (SA). SA dùng cùng cấu trúc lân cận với HC nhưng chấp nhận nghiệm xấu hơn với xác suất $\exp(-\Delta f/T)$, trong đó T giảm dần theo lịch làm nguội hình học: $T_{k+1} = \text{cooling_rate} \cdot T_k$.

Cấu hình đại diện:

- **Rastrigin:** nhiệt độ khởi tạo T_0 tương đối cao, lịch làm nguội hình học (cooling_rate gần 1), bước nhảy tương đương với HC để so sánh công bằng.
- **Knapsack:** chỉ điều chỉnh T_0 và cooling_rate, sử dụng cùng cơ chế lân cận bit-flip với HC.

Genetic Algorithm (GA). GA duy trì một quần thể cá thể, áp dụng tournament selection, crossover và mutation; luôn có elitism giữ lại một số cá thể tốt nhất qua thế hệ.

Cấu hình và operator đúng với code:

- **Rastrigin (liên tục):**

- *Crossover:* sử dụng Simulated Binary Crossover (SBX), **không** phải one-point/two-point. SBX sinh con liên tục nằm trong vùng lân cận hai cha mẹ, phù hợp không gian thực.
- *Mutation:* perturbation từng chiều với xác suất mutation_rate; các giá trị mutation_rate trong benchmark được đặt xấp xỉ $1/d$ (0.10 cho $d = 10$, 0.03 cho $d = 30$, 0.02 cho $d = 50$).
- *Quần thể:* pop_size tăng dần theo dimension ($40 \rightarrow 60 \rightarrow 80$) để bù lại độ khó tăng.
- *Selection:* tournament_size lần lượt là $3, 5, 7$ cho ba cấu hình; crossover_rate giữ khoảng 0.9 , luôn bật elitism.

- **Knapsack (rời rạc):**

- *Crossover:* uniform crossover trên bit – mỗi bit con được chọn độc lập từ cha hoặc mẹ theo một xác suất, phù hợp với biểu diễn 0/1.
- *Mutation:* bit-flip với xác suất $\text{mutation_rate} = 1/n$ (chuẩn trong GA nhị phân).
- *Quần thể:* pop_size tăng theo kích thước bài toán (30 cho $n = 50, 100$; 40 cho $n = 200$); tournament_size=3; luôn dùng elitism.

Firefly Algorithm (FA). **Rastrigin (liên tục):** sử dụng biến thể FA chuẩn trong không gian liên tục, với các tham số ($n_{\text{fireflies}}, \alpha, \beta_0, \gamma$) được chọn *riêng cho từng cấu hình*:

- quick_convergence ($d = 10$): $n_{\text{fireflies}} = 40, \alpha = 0.18, \gamma = 0.02$.
- multimodal_escape ($d = 30$): $n_{\text{fireflies}} = 60, \alpha = 0.20, \gamma = 0.01$.
- scalability ($d = 50$): $n_{\text{fireflies}} = 80, \alpha = 0.22, \gamma = 0.008$.

β_0 luôn đặt bằng 1.0 . Xu hướng chung: dimension càng cao thì quần thể và mức nhiễu α tăng lên một chút để tăng exploration, trong khi γ giảm để lực hút không bị “tắt” quá nhanh khi r tăng theo \sqrt{d} .

Knapsack: dùng biến thể rời rạc hoá đã mô tả ở *Cơ sở toán học*, trong đó:

- Di chuyển có hướng (directed movement) lật tối đa một số bit mỗi bước (`max_flips_per_move=3`) để tiến gần tới firefly sáng hơn.
- Nhiều ngẫu nhiên bit-flip với xác suất $\alpha_{\text{flip}} = 0.2$.
- Luôn dùng chiến lược repair `greedy_remove` (loại các vật phẩm có v_k/w_k thấp nhất) để đảm bảo feasibility; penalty được hỗ trợ ở mức framework nhưng không kích hoạt trong cấu hình benchmark chính.

Quy mô quần thể FA cho Knapsack tăng theo n : với $n \leq 100$ dùng $n_{\text{fireflies}} = 30$, với $n = 200$ dùng $n_{\text{fireflies}} = 40$.

Lưu ý về tham số. Các tham số *không* hoàn toàn “cố định cho từng family bài toán” như cách diển đạt ngắn gọn ban đầu, mà được chọn thủ công **theo từng kích bản**:

- Với Rastrigin: mỗi cấu hình `quick_convergence` (10D), `multimodal_escape` (30D), `scalability` (50D) có bộ tham số riêng cho FA, SA, HC, GA (`pop_size`, `n_fireflies`, `mutation_rate`, `initial_temp`, `num_neighbors`, ...).
- Với Knapsack: tham số thay đổi theo kích thước n (50, 100, 200) – budget, quần thể FA/GA được scale theo n , `mutation_rate` của GA đặt đúng bằng $1/n$, nhưng trong cùng một kích thước thì tham số cố định cho mọi instance-type và mọi run.

Tuy nhiên, trong *mỗi* cấu hình con (ví dụ: Rastrigin dim=30, hoặc Knapsack $n = 100$), bộ tham số được giữ nguyên cho tất cả thuật toán trong mọi lần chạy và **không** được tune theo từng instance cụ thể. Điều này khiến kết quả vẫn mang tính “out-of-the-box theo từng scenario” chứ không phải đã tối ưu hoá đến mức per-instance.

8 Cuckoo Search

8.1 Giới thiệu thuật toán

Cuckoo Search (CS) là một thuật toán metaheuristic được đề xuất bởi Xin-She Yang và Suash Deb vào năm 2009. Thuật toán lấy cảm hứng từ hành vi sinh sản ký sinh của loài chim cu (cuckoo), kết hợp với cơ chế Lévy flight để khám phá không gian tìm kiếm hiệu quả.

Cuckoo Search thuộc nhóm *Swarm Intelligence* tương tự như PSO hay ABC, với ý tưởng là *mỗi* cá thể (tổ chim) đại diện cho một nghiệm ứng viên trong không gian tìm kiếm. Quá trình tiến hóa xảy ra thông qua việc các tổ chim được thay thế dần bằng các nghiệm tốt hơn.

8.2 Ý tưởng thuật toán

Mỗi con chim cu đẻ trứng vào tổ của các loài chim khác. Nếu trứng bị phát hiện là “kẻ lạ”, chủ tổ sẽ loại bỏ trứng đó hoặc bỏ tổ và xây tổ mới. Cuckoo Search mô phỏng hành vi này bằng cách:

- Mỗi “tổ” đại diện cho một nghiệm trong không gian tìm kiếm.
- Một số trứng (nghiệm) mới được tạo ra thông qua Lévy flight từ các tổ hiện tại.
- Một tỷ lệ p_a các tổ xấu nhất sẽ bị loại bỏ và thay bằng các tổ mới ngẫu nhiên.
- Tổ tốt nhất được giữ lại qua các thế hệ.

8.3 Cơ sở toán học của thuật toán

Giả sử không gian tìm kiếm có d chiều và n tổ chim. Mỗi tổ $x_i = [x_{i1}, x_{i2}, \dots, x_{id}]$ tương ứng với một nghiệm. Tại mỗi vòng lặp, tổ mới được tạo ra theo công thức:

$$x_i^{(t+1)} = x_i^{(t)} + \alpha \cdot \text{Levy}(\beta) \quad (5)$$

Trong đó:

- α là hệ số bước bay (step size),
- $\text{Levy}(\beta)$ là bước nhảy theo phân phối Lévy với tham số β (thường $\beta = 1.5$).

Phân phối Lévy được sinh ra theo công thức Mantegna:

$$s = \frac{u}{|v|^{1/\beta}}, \quad u \sim N(0, \sigma_u^2), \quad v \sim N(0, 1) \quad (6)$$

với

$$\sigma_u = \left[\frac{\Gamma(1 + \beta) \sin(\pi\beta/2)}{\Gamma((1 + \beta)/2)\beta 2^{(\beta-1)/2}} \right]^{1/\beta} \quad (7)$$

Sau mỗi lần cập nhật, các tổ bị phát hiện (với xác suất p_a) sẽ bị thay thế ngẫu nhiên:

$$x_i^{(t+1)} = x_i^{(t)} + r \cdot (x_j^{(t)} - x_k^{(t)}), \quad (8)$$

trong đó x_j, x_k là hai tổ ngẫu nhiên khác nhau, và r là một số ngẫu nhiên trong $[0, 1]$.

8.4 Triển khai thuật toán

8.4.1 Cách hoạt động của thuật toán

1. **Khởi tạo quần thể:** Tạo n tổ ngẫu nhiên trong không gian tìm kiếm.
2. **Dánh giá:** Tính giá trị hàm mục tiêu (fitness) của từng tổ.
3. **Bay Lévy:** Tạo các tổ mới bằng cách bay Lévy từ các tổ hiện tại.
4. **Cập nhật:** Nếu tổ mới tốt hơn, thay thế tổ cũ.
5. **Phát hiện trùng lặp:** Với xác suất p_a , thay thế các tổ tệ nhất bằng tổ mới ngẫu nhiên.
6. **Ghi nhận nghiệm tốt nhất:** Cập nhật tổ có giá trị hàm mục tiêu nhỏ nhất.
7. **Lặp lại** cho đến khi đạt số vòng lặp tối đa hoặc hội tụ.

8.4.2 Mã giả

Algorithm 11 Thuật toán Cuckoo Search

- ```

1: Khởi tạo n tổ x_i ngẫu nhiên trong không gian tìm kiếm
2: Dánh giá giá trị hàm mục tiêu $f(x_i)$ cho mỗi tổ
3: for $t = 1$ đến T_{max} do
4: (Pha 1: Lévy flight)
5: for mỗi tổ x_i do
6: Sinh bước Lévy $s \sim \text{Levy}(\beta)$
7: $x'_i = x_i + \alpha \cdot s$
8: Nếu $f(x'_i) < f(x_i)$ thì cập nhật $x_i = x'_i$
9: end for
10: (Pha 2: Phát hiện trùng)
11: Chọn ngẫu nhiên một phần p_a các tổ tệ nhất
12: Thay thế chúng bằng tổ mới ngẫu nhiên trong không gian tìm kiếm
13: Cập nhật tổ tốt nhất hiện tại
14: end for
15: Xuất ra nghiệm tốt nhất tìm được

```
-

## 9 Thiết lập các bài toán đánh giá

### 9.1 ACO

### 9.2 PSO

### 9.3 ABC

### 9.4 Firefly Algorithm

**Bài toán Rastrigin (liên tục).** Rastrigin là một hàm benchmark đa cực trị, phi lồi, thường dùng để kiểm tra khả năng tối ưu hoá toàn cục của các thuật toán metaheuristic. Hàm có nhiều cực tiểu địa phương và một cực tiểu toàn cục tại  $\mathbf{0}$  với  $f(\mathbf{0}) = 0$  :contentReference[oaicite:2]index=2.

Trong benchmark này, chúng tôi dùng 3 cấu hình:

- **quick\_convergence:**  $d = 10$ . Budget đánh giá hàm ở mức  $O(10^4)$ , tương ứng với vài trăm vòng lặp cho mỗi thuật toán. Mục tiêu: so sánh tốc độ hội tụ giai đoạn đầu trên một bài toán tương đối “dễ”.
- **multimodal\_escape:**  $d = 30$ . Budget tăng lên ( $c\sqrt{d} \times 10^4$  đánh giá) để tạo điều kiện cho việc thoát bẫy cục bộ. Mục tiêu: quan sát khả năng xử lý đa cực trị khi không gian nghiệm mở rộng.
- **scalability:**  $d = 50$ . Budget lớn nhất trong ba cấu hình nhưng vẫn bị giới hạn; đây là bài test “khó”, chủ yếu để xem xu hướng suy giảm hiệu năng khi tăng chiều.

Tất cả trường hợp đều dùng cùng miền tìm kiếm chuẩn của Rastrigin, với giới hạn biên cố định trên từng chiều.

### Bài toán Knapsack 0/1 (rời rạc).

- Số vật phẩm:  $n \in \{50, 100, 200\}$ .
- Instance types: *uncorrelated, weakly correlated, strongly correlated, subset-sum* (theo phân loại kinh điển trong knapsack literature).
- Mỗi cấu hình (kích thước  $\times$  loại instance): 3 seeds độc lập, mỗi seed chạy 10 lần  $\Rightarrow 30$  runs/thuật toán/cấu hình.
- Với  $n \leq 100$ : tính nghiệm tối ưu bằng Dynamic Programming (DP) để làm ground truth; với  $n = 200$  so sánh theo gap tương đối so với best-known trong các thuật toán.

### Chỉ số đánh giá. Rastrigin:

- *ECDF Fixed-Target*: empirical CDF của số lượng target đạt được trong một budget cố định (chuẩn Dolan–Moré) :contentReference[oaicite:3]index=3.
- *Fixed-Budget CDF*: phân bố sai số cuối cùng tại các mức budget khác nhau.
- *ERT (Expected Running Time)*: số lượng đánh giá kỳ vọng để đạt một target nhất định (khi thành công chưa đạt 100%, ERT được tính từ các run thành công).
- *Performance profiles*: xác suất một thuật toán có performance ratio  $\rho_{p,s} \leq \tau$  so với best solver, theo định nghĩa của Dolan–Moré :contentReference[oaicite:4]index=4.
- *Data profiles*: xác suất giải được bài toán trong một budget chuẩn hoá, theo Moré–Wild :contentReference[oaicite:5]index=5.

### Knapsack:

- Optimality gap (%):

$$\text{gap} = \frac{z^* - z_{\text{alg}}}{z^*} \times 100\%,$$

với  $z^*$  là optimum (hoặc best-known với  $n = 200$ ).

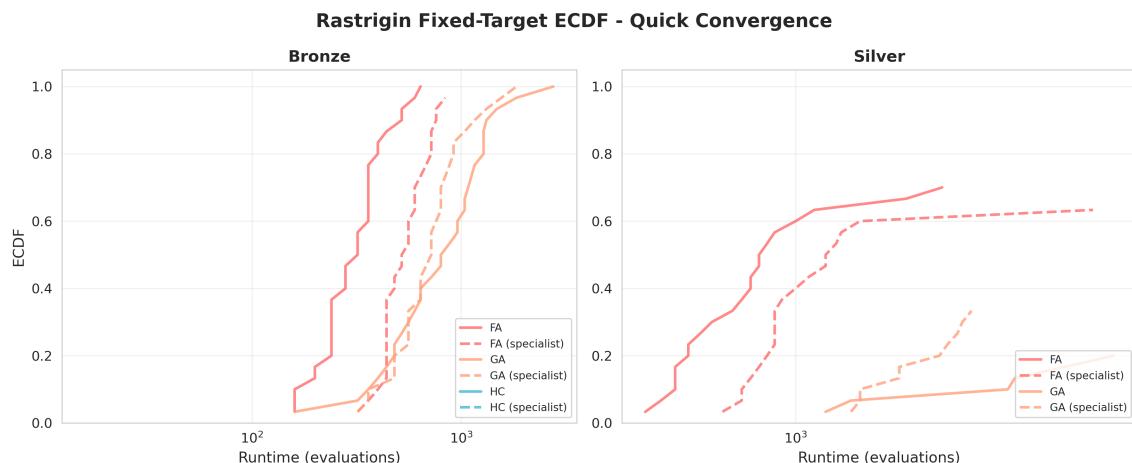
- Success rate: tần suất gap  $\leq$  các ngưỡng cho trước (ví dụ: 10%).
- Feasibility rate: tần suất nghiệm thỏa ràng buộc sức chứa (đặc biệt quan trọng cho chiến lược penalty).
- Thống kê: kiểm định Wilcoxon signed-rank cho so sánh cặp đôi FA/GA/HC/SA trên từng cấu hình, và Copeland ranking để tổng hợp số lần thắng-thua. (Friedman/Nemenyi không được triển khai đầy đủ nên không báo cáo ở đây.)

## 9.5 Cuckoo Search

# 10 Phân tích kết quả bài toán

## 10.1 Firefly Algorithm

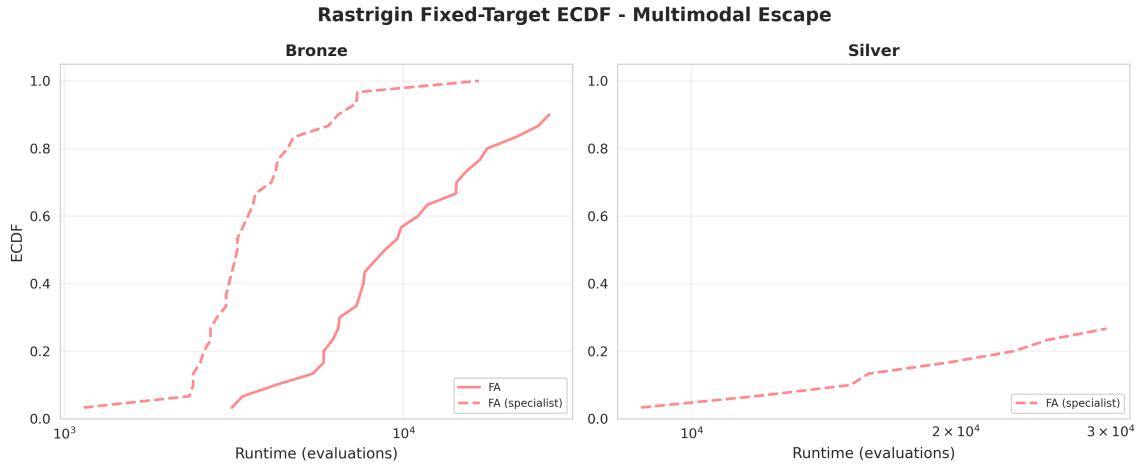
### 10.1.1 Rastrigin: mức độ tiệm cận nghiệm tối ưu và đặc trưng hội tụ



Hình 3: ECDF cho quick\_convergence (dim=10).

**ECDF theo ngưỡng mục tiêu.** Dưới với cấu hình 10 chiều, ECDF cho thấy:

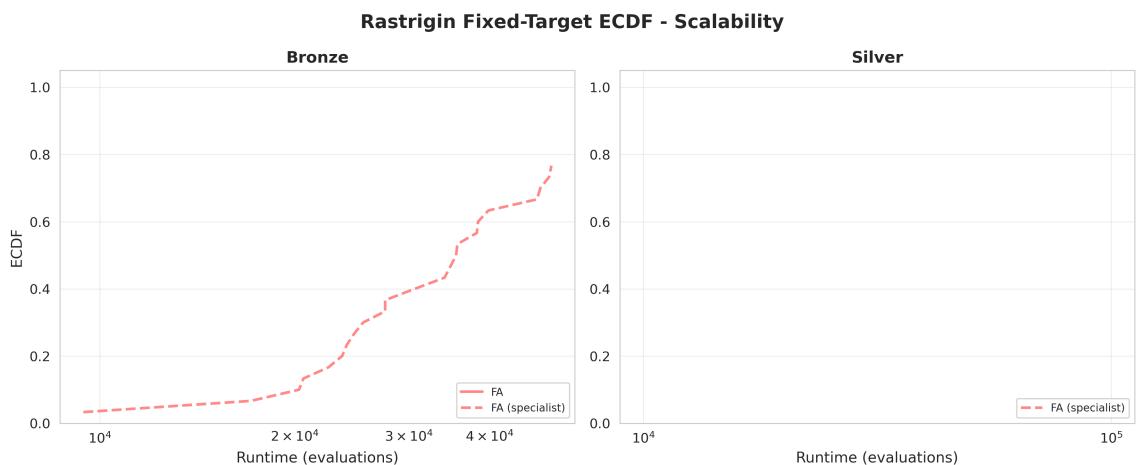
- Ở mức Bronze, cả FA và GA đều đạt tỉ lệ thành công rất cao: các đường ECDF tiệm cận 1 khi ngân sách tiền dần tới khoảng  $10^3$  lần đánh giá. FA có xu hướng đạt cùng mức ECDF với chi phí nhỏ hơn đôi chút, thể hiện lợi thế nhẹ về tốc độ hội tụ.
- HC chậm hơn rõ rệt: đường ECDF nằm thấp và tăng chậm, phản ánh việc thuật toán này thường bị kẹt trong các cực trị địa phương của Rastrigin ngay cả với ngưỡng Bronze.
- Ở mức Silver, sự khác biệt trở rõ ràng: chỉ FA (đặc biệt là cấu hình *specialist*) duy trì được ECDF đáng kể (xấp xỉ 0.6–0.7), trong khi GA và HC hầu như không chạm được ngưỡng trong ngân sách đang xét, các đường ECDF gần như bám sát trực hoành. FA vì vậy là thuật toán duy nhất còn hoạt động hiệu quả khi yêu cầu độ chính xác cao hơn trên Rastrigin 10 chiều.



Hình 4: ECDF cho multimodal\_escape (dim=30).

Khi tăng lên 30 chiều, hình dạng ECDF cho thấy độ khó tăng rõ rệt:

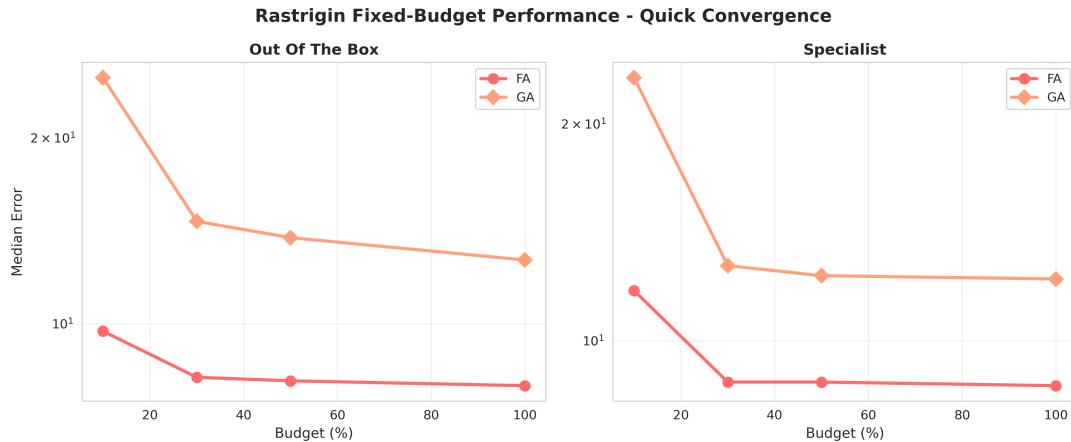
- Ở mức Bronze, chỉ còn các biến thể của FA (bản gốc và *specialist*) đạt được tỉ lệ thành công đáng kể. Cả hai đường cong đều bị dịch sang phải, cần tới cỡ  $10^4$  đánh giá để ECDF tiệm cận 1; cấu hình *specialist* đạt cùng mức ECDF với chi phí thấp hơn, cho thấy việc tinh chỉnh tham số giúp cải thiện rõ rệt khả năng thoát bẫy đa cực trị.
- Các thuật toán còn lại (GA, HC, SA) hầu như không đạt được ngưỡng Bronze trong ngân sách đã chọn nên không xuất hiện trên đồ thị; về thực chất, chúng thất bại gần như hoàn toàn trên Rastrigin 30 chiều ở mức mục tiêu này.
- Ở mức Silver, độ khó tăng vọt: chỉ FA *specialist* đạt được một phần nhỏ số lần chạy (ECDF dừng dưới 0.3 ngay cả ở rìa phải trực hoành), trong khi FA gốc và các thuật toán khác không có lần chạy nào chạm ngưỡng. Điều này cho thấy từ 30 chiều trở lên, mức Silver của Rastrigin đã vượt quá khả năng của hầu hết thuật toán trong bộ benchmark.



Hình 5: ECDF cho scalability (dim=50, trực log).

Ở cấu hình 50 chiều, Rastrigin trở thành một bài toán đặc biệt thách thức trong bối cảnh ngân sách giới hạn:

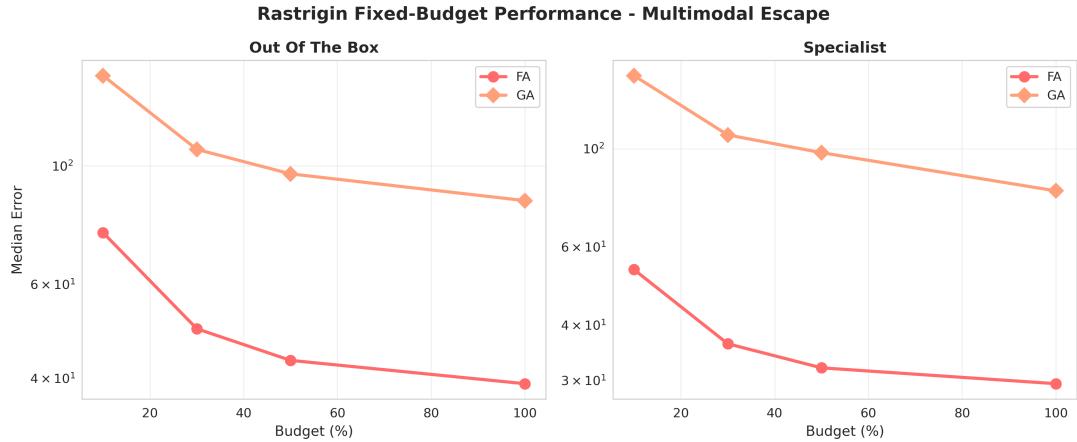
- Ở mức Bronze, chỉ còn FA *specialist* đạt được một tỉ lệ thành công trung bình khá khiêm tốn: đường ECDF tăng rất chậm và chỉ tiệm cận quanh 0.7 khi ngân sách tiến gần  $4 \times 10^4$  đánh giá. FA gốc và các thuật toán khác không đạt được ngưỡng nên không để lại dấu vết trên đồ thị.
- Ở mức Silver, toàn bộ các đường ECDF nằm tại 0, tương đương việc không thuật toán nào đạt được ngưỡng Silver trên Rastrigin 50 chiều trong ngân sách tối đa. Rastrigin high-dimensional với yêu cầu độ chính xác cao vì thế có thể xem là “ngoài tầm với” đối với tập thuật toán đang xét.



Hình 6: Rastrigin – fixed-budget performance (quick\_convergence,  $d = 10$ ). Trục tung là sai số trung vị (median error), trục hoành là tỉ lệ ngân sách.

**Fixed-Budget Performance: sai số cuối cùng dưới các mức ngân sách cố định.** Với cấu hình quick\_convergence (10 chiều), đường cong fixed-budget cho thấy:

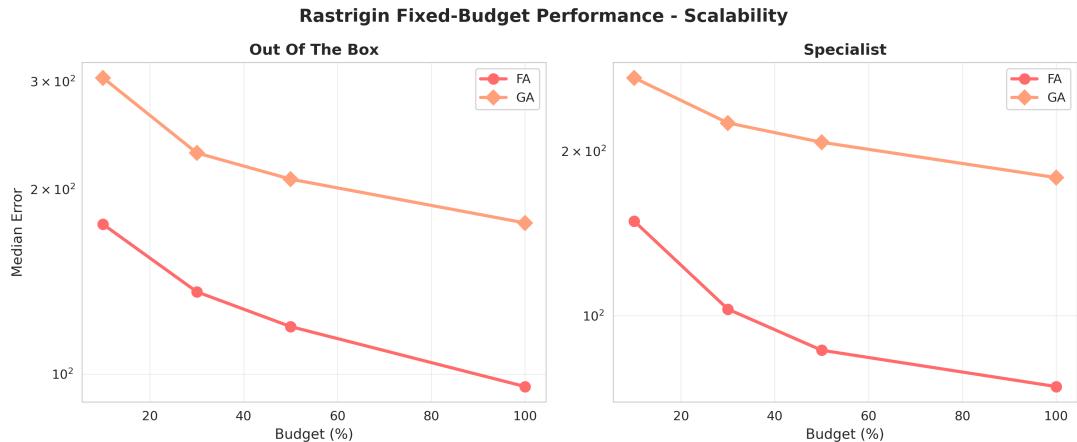
- Cả FA và GA đều có sai số trung vị giảm đơn điệu khi tăng ngân sách, nhưng FA luôn giữ mức sai số thấp hơn rõ rệt. Ở mọi mức budget (10%, 30%, 50%, 100%), đường của FA nằm dưới GA khoảng gần một bậc độ lớn trên thang log.
- Lợi ích của việc tăng ngân sách thể hiện mạnh mẽ giữa 10% và 30%; sau khoảng 50% ngân sách, cả hai thuật toán đều rơi vào vùng “diminishing returns”, sai số giảm thêm rất ít.
- Cấu hình *specialist* cải thiện nhẹ cho cả hai thuật toán, chủ yếu ở ngân sách nhỏ (10–30%). Tuy nhiên, thứ hạng tương đối không đổi: FA vẫn là thuật toán cho sai số cuối cùng thấp nhất, GA ổn định nhưng kém hơn trên toàn dải ngân sách.



Hình 7: Rastrigin – fixed-budget performance (multimodal\_escape,  $d = 30$ ).

Khi tăng lên cấu hình multimodal\_escape (30 chiều), độ khó tăng rõ rệt:

- Mức sai số trung vị của cả FA và GA đều cao hơn đáng kể so với  $d = 10$ , ngay cả ở 100% ngân sách. Điều này phù hợp với ECDF: nhiều lần chạy không chạm được các target Silver dù đã dùng hết ngân sách.
- FA tiếp tục giữ lại thế ổn định: ở mọi mức budget, đường cong của FA nằm thấp hơn GA, và khoảng cách giữa hai thuật toán vẫn tương đối lớn.
- Tinh chỉnh *specialist* giúp FA giảm thêm vài đơn vị sai số trên toàn dải ngân sách, trong khi GA cũng cải thiện nhưng vẫn bị bỏ xa. Có thể hiểu rằng ở 30 chiều, FA không chỉ đạt tỉ lệ thành công cao hơn (ECDF) mà còn cho chất lượng nghiệm cuối cùng tốt hơn trong khung fixed-budget.

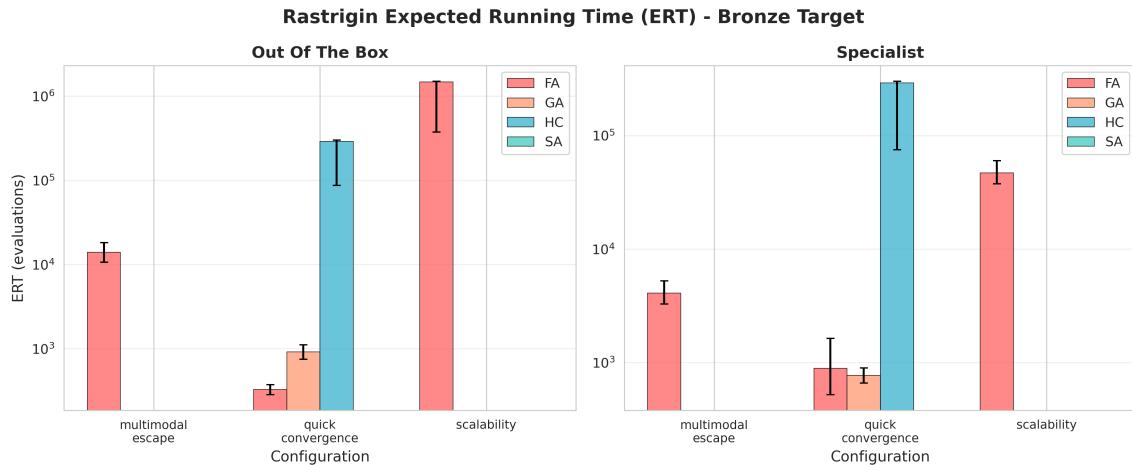


Hình 8: Rastrigin – fixed-budget performance (scalability,  $d = 50$ ).

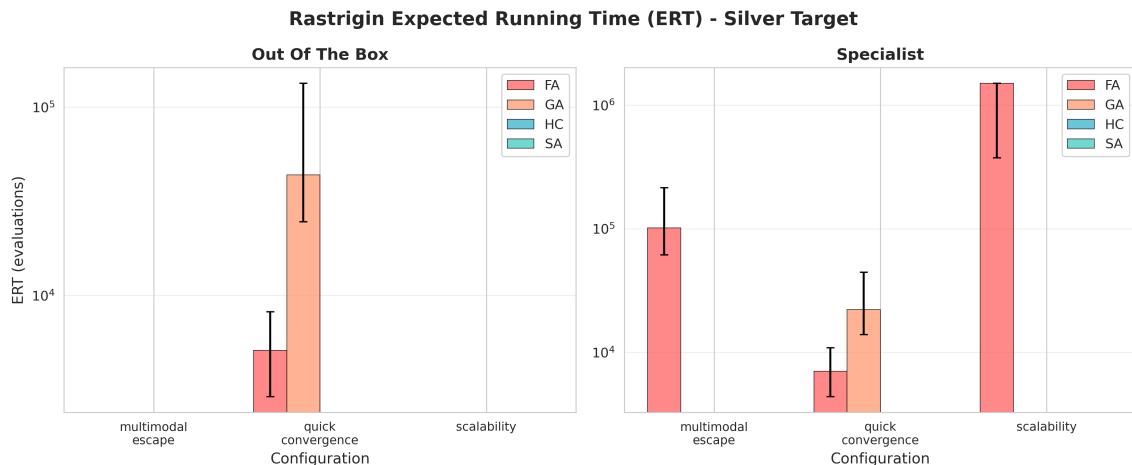
Ở cấu hình scalability (50 chiều), bức tranh trở nên bi quan hơn:

- Các đường cong cho thấy sai số trung vị vẫn rất cao ngay cả khi dùng 100% ngân sách: FA chỉ giảm từ khoảng  $\sim 1.7 \times 10^2$  xuống dưới  $10^2$ , GA dao động quanh vùng  $[1.8, 3] \times 10^2$ . Điều này nhất quán với ECDF: gần như không có lần chạy nào đạt được các target Bronze/Silver ở 50 chiều.

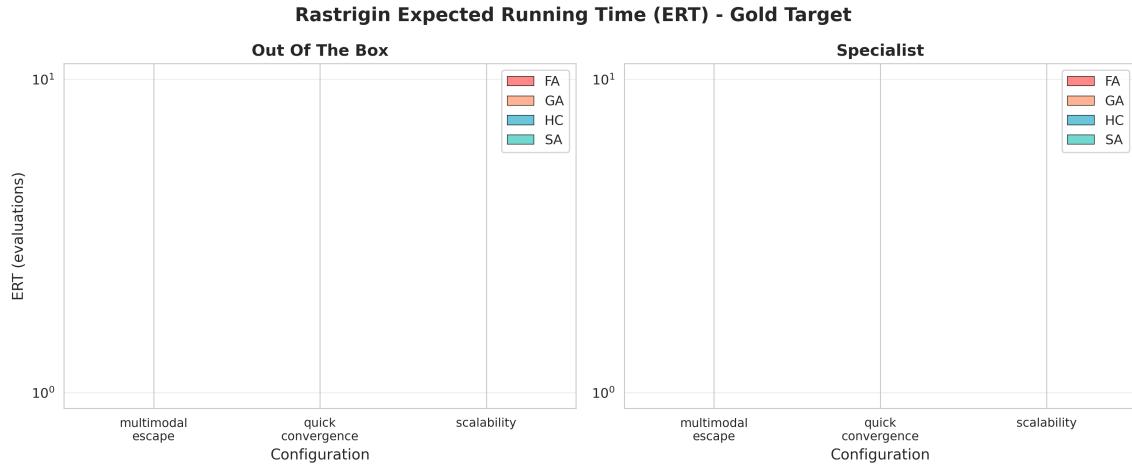
- FA tiếp tục vượt trội GA trên toàn bộ dải budget: với cùng một ngân sách, nghiệm trung vị của FA luôn tốt hơn đáng kể. Nói cách khác, nếu buộc phải chọn giữa hai thuật toán trong bối cảnh high-dimensional Rastrigin, FA luôn là lựa chọn “ít tệ hơn”.
- Cấu hình *specialist* giúp FA cải thiện thêm một chút (đặc biệt ở ngân sách thấp), nhưng không thay đổi bản chất vấn đề: với dimension 50 và ngân sách hiện tại, cả hai thuật toán đều đang hoạt động trong vùng “chưa hội tụ”, sai số tuyệt đối vẫn lớn so với nghiệm tối ưu.



Hình 9: Rastrigin – ERT tới ngưỡng Bronze.



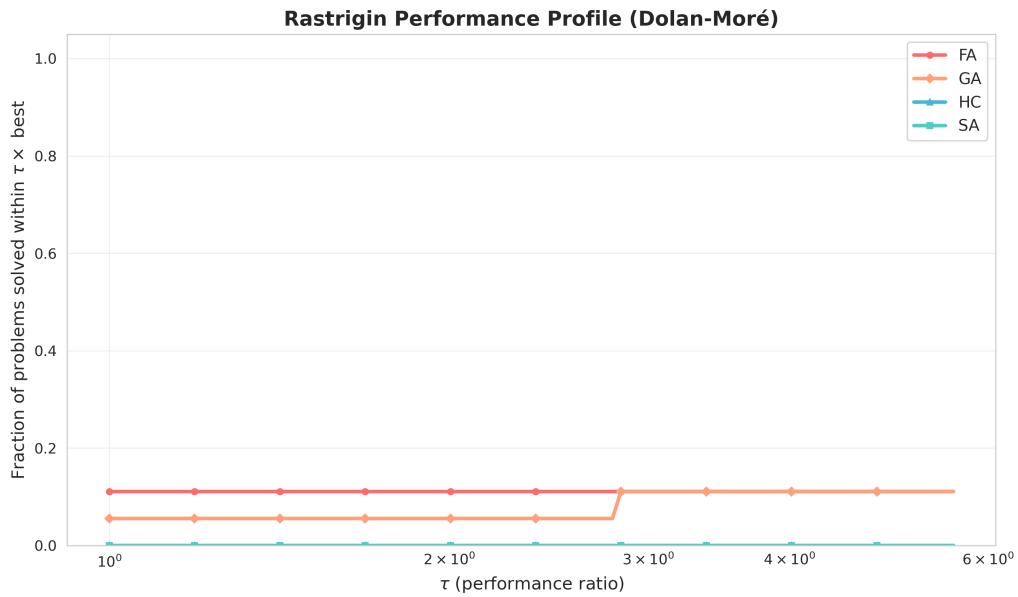
Hình 10: Rastrigin – ERT tới ngưỡng Silver.



Hình 11: Rastrigin – ERT tới ngưỡng Gold (không có thuật toán nào đạt target).

**ERT, performance profiles và data profiles.** ERT (Expected Running Time) chỉ có ý nghĩa khi tồn tại số lượng đủ lớn các lần chạy thành công trên target đang xét. Các biểu đồ ERT cho Rastrigin cho thấy:

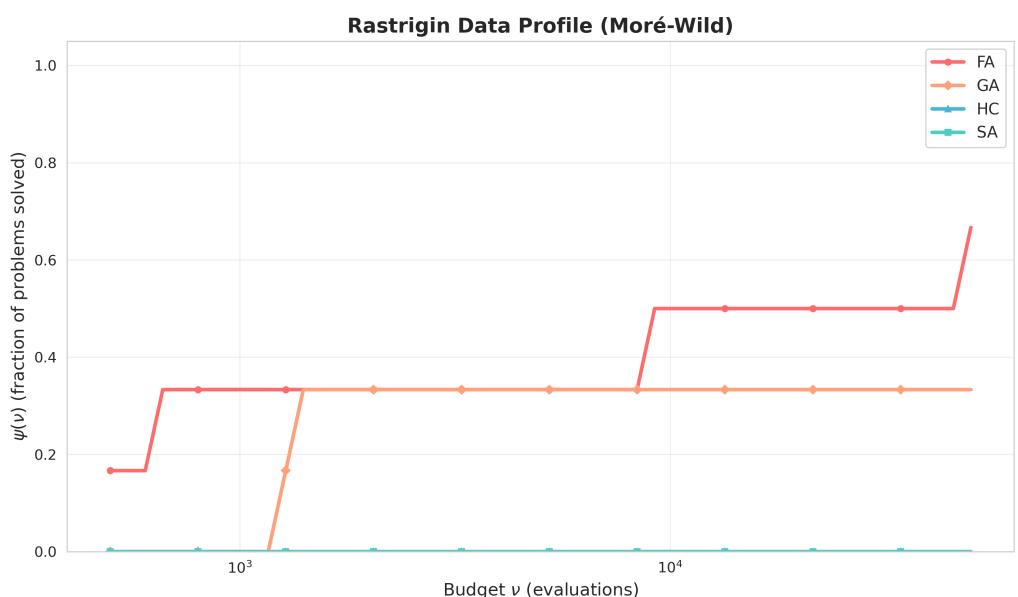
- **Ngưỡng Bronze.** Ở cấu hình *quick\_convergence* (10 chiều), cả FA, GA và HC đều đạt được target Bronze. FA có ERT nhỏ nhất (cỡ  $10^2$ – $10^3$  đánh giá), GA chậm hơn khoảng một bậc, còn HC chậm hơn rất nhiều bậc và do đó chỉ đóng vai trò baseline. SA hầu như không đạt được target. Ở hai cấu hình khó hơn (*multimodal\_escape* 30 chiều và *scalability* 50 chiều), chỉ có FA đạt Bronze; các thuật toán còn lại không có run thành công nên không xuất hiện trên biểu đồ. Việc tinh chỉnh (*specialist*) giúp ERT của FA giảm đáng kể nhưng giá trị tuyệt đối vẫn nằm trong vùng từ vài chục nghìn tới hàng triệu đánh giá.
- **Ngưỡng Silver.** Với Silver, bức tranh càng khắt khe hơn. Ở 10 chiều, FA và GA vẫn đạt target nhưng ERT của GA lớn hơn FA rõ, phản ánh việc GA cần nhiều đánh giá hơn để hội tụ tới mức sai số sâu hơn. Ở 30 chiều, chỉ còn FA (đặc biệt là cấu hình *specialist*) đạt Silver với ERT rất lớn (khoảng  $10^5$  đánh giá), và ở 50 chiều chỉ FA *specialist* đạt Silver với ERT lên tới cỡ  $10^6$ . Điều này nhất quán với ECDF: Silver trên Rastrigin high-dimensional là một target cực khó.
- **Ngưỡng Gold.** Trên cả ba cấu hình và bốn thuật toán, không có run nào đạt được target Gold trong ngân sách cho phép. Vì vậy Hình 11 thực chất minh họa một trường hợp “ERT không xác định”: success rate bằng 0, và mọi so sánh ERT tại ngưỡng Gold đều vô nghĩa.



Hình 12: Performance profiles cho Rastrigin (Dolan–Moré).

Performance profile mô tả, với mỗi tỷ lệ hiệu năng  $\tau$ , tỷ lệ các bài toán mà một thuật toán có thời gian chạy không vượt quá  $\tau$  lần solver tốt nhất. Kết quả cho Rastrigin cho thấy:

- Đường cong của FA nằm cao hơn GA tại  $\tau = 1$ , tức là trong số rất ít các cặp (cầu hình, target) được giải thành công, FA thường là thuật toán nhanh nhất về số lần đánh giá.
- Khi tăng  $\tau$  lên khoảng 3, đường của GA mới bắt kịp FA; hai đường đều tiệm cận một mức trần thấp (khoảng 0,1), phản ánh thực tế là chỉ một phần rất nhỏ các bài toán trong bộ test được giải bởi bất kỳ thuật toán nào.
- HC và SA hầu như không xuất hiện trên performance profile vì không giải được target nào trong tập Rastrigin ở các mức ngân sách đã chọn.



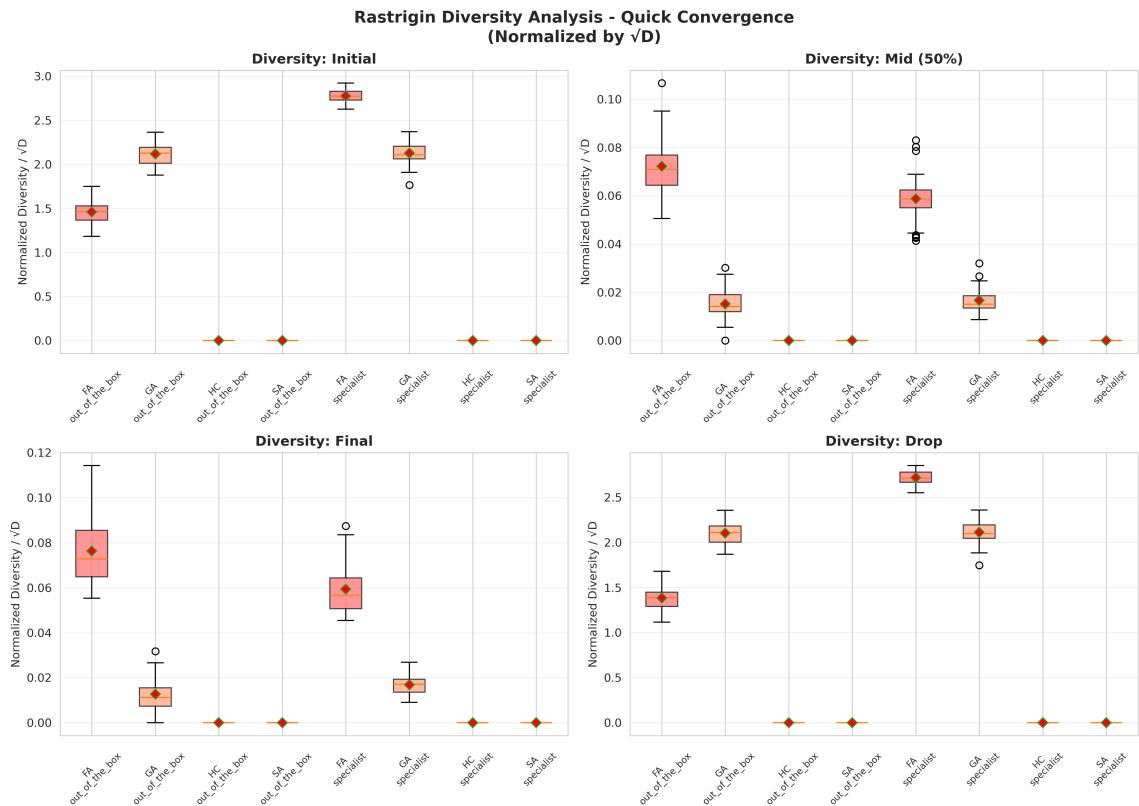
Hình 13: Data profiles cho Rastrigin (Moré–Wild).

Data profile  $\psi(\nu)$  biểu diễn tỷ lệ bài toán được giải trong một ngân sách chuẩn hoá  $\nu$  cho trước. Trên Rastrigin:

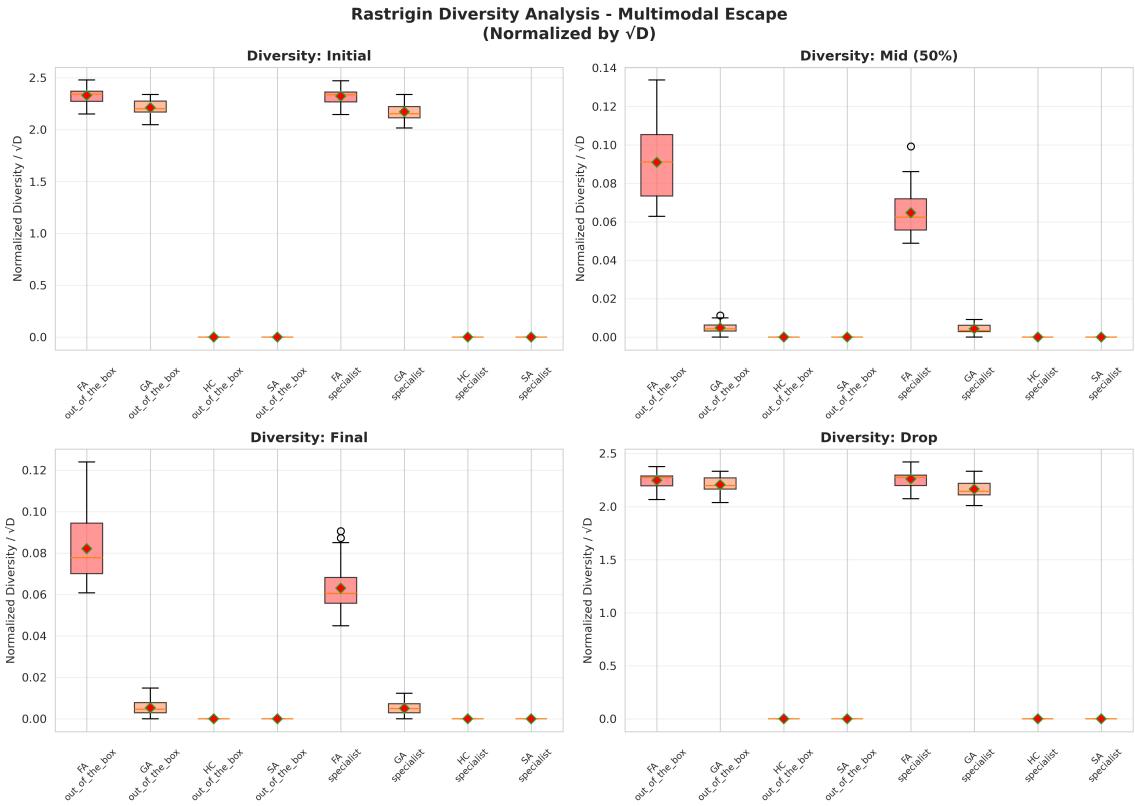
- FA luôn là thuật toán có coverage cao nhất: với ngân sách rất nhỏ (vài trăm đánh giá), FA đã giải được khoảng  $1/6$  số bài toán; khi tăng ngân sách lên cỡ  $10^4$ – $3 \times 10^4$  đánh giá, coverage tăng dần lên gần  $2/3$ .
- GA chỉ bắt đầu giải được bài toán khi ngân sách tăng lên mức trung bình và nhanh chóng đạt trần quanh mức  $1/3$  số bài toán; sau đó đường cong của GA hầu như phẳng, không hưởng lợi nhiều từ việc tăng ngân sách.
- HC và SA không giải được bất kỳ instance nào trong bộ target Rastrigin dưới các mức ngân sách đã xét, nên đường data profile gần như dính sát trực hoành.
- Ngay cả ở ngân sách lớn nhất, không có thuật toán nào đạt coverage gần 1; điều này cũng cố nhận định rằng Rastrigin high-dimensional (đặc biệt với các target Silver/Gold) về cơ bản là quá khó trong khung ngân sách hiện tại.

### 10.1.2 Rastrigin: đa dạng quần thể và hiện tượng dừng sớm (stagnation)

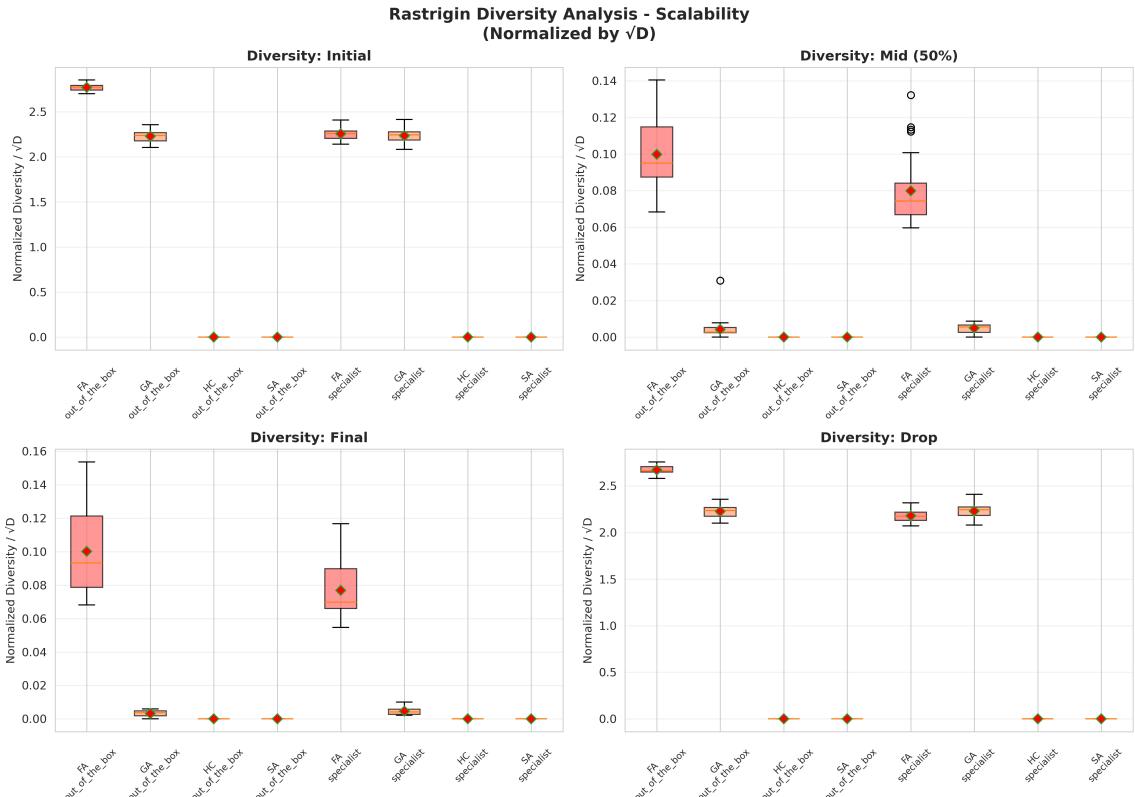
**Đa dạng quần thể theo thời gian.** Đa dạng quần thể được đo bằng khoảng cách Euclid trung bình giữa các cá thể, chuẩn hoá theo  $\sqrt{D}$  để cho phép so sánh giữa các chiều khác nhau. Hình 14–16 hiển thị boxplot của bốn thời điểm: *initial* (ngay sau khởi tạo), *mid* (50% ngân sách), *final* (kết thúc chạy) và *drop* (mức giảm đa dạng từ initial tới final).



Hình 14: Rastrigin – phân tích đa dạng quần thể (quick\_convergence,  $D = 10$ ).



Hình 15: Rastrigin – phân tích đa dạng quần thể (multimodal\_escape,  $D = 30$ ).



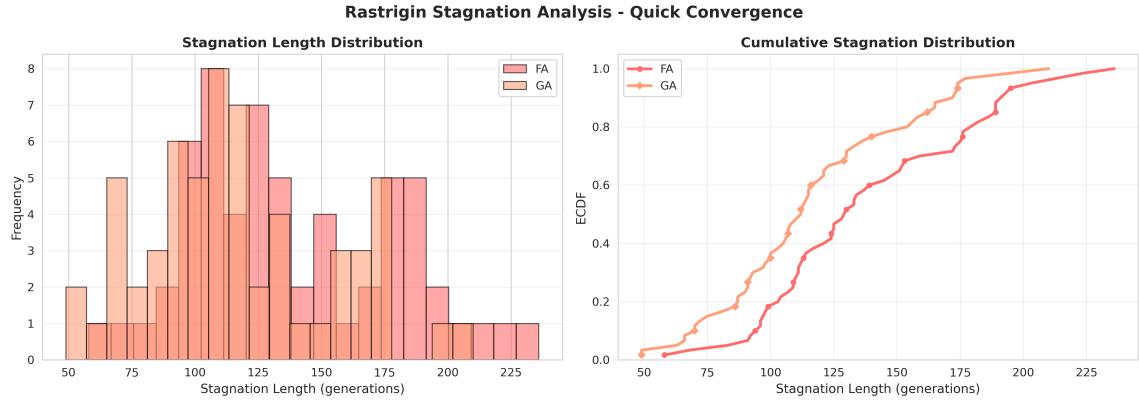
Hình 16: Rastrigin – phân tích đa dạng quần thể (scalability,  $D = 50$ ).

Các quan sát chính:

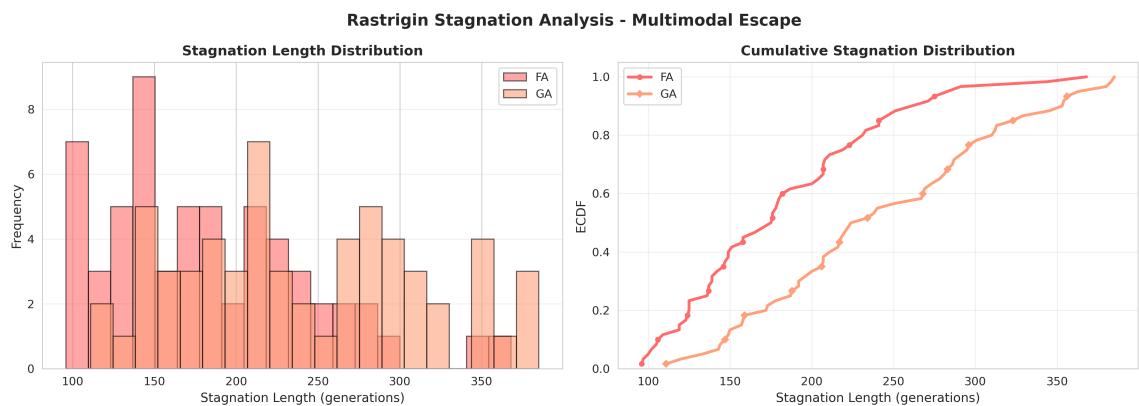
- **Khởi tạo.** Trên cả ba cấu hình, FA và GA đều bắt đầu với mức đa dạng khá cao ( $\approx 2,0\text{--}2,8$  sau chuẩn hoá). Với cấu hình quick\_convergence, GA (đặc biệt bản out-of-the-box) có đa dạng khởi tạo lớn hơn FA, trong khi FA specialist có mức cao nhất. Điều này cho thấy lợi thế của FA trên Rastrigin không đến từ việc “rải quần thể rộng hơn ngay từ đầu” mà chủ yếu do động lực cập nhật trong quá trình tối ưu.
- **Giữa quá trình (50% ngân sách).** Ở cả ba cấu hình, GA mất đa dạng rất nhanh: median diversity của GA chỉ còn  $\approx 0,01\text{--}0,02$ , trong khi FA vẫn duy trì quanh  $\approx 0,06\text{--}0,10$ . FA specialist có đa dạng mid hơi thấp hơn FA gốc nhưng vẫn cao hơn GA rõ rệt. Tức là GA trải qua hiện tượng premature convergence: áp lực chọn lọc và lai ghép làm quần thể co cụm sớm quanh một vài basin, trong khi FA vẫn giữ được một “vòng đai” nghiệm khác biệt hơn.
- **Cuối quá trình.** Tới thời điểm kết thúc, đa dạng của GA gần như bằng 0 ở mọi cấu hình, cho thấy quần thể gần như đồng nhất. FA vẫn giữ một mức đa dạng dương đáng kể (đặc biệt ở multimodal\_escape và scalability), dù đã giảm hơn 90% so với ban đầu. Điều này phù hợp với quan sát ECDF: FA vẫn còn khả năng “nhúc nhích” sang các basin khác ở cuối run, trong khi GA gần như bị khoá cứng trong một vùng nghiệm.
- **Mức giảm đa dạng (drop).** Cả FA và GA đều có drop lớn (trên 90% initial diversity), riêng GA thường tụt về gần 0 nên drop tuyệt đối của GA thường lớn hơn hoặc tương đương FA. FA specialist có drop lớn nhất trên quick\_convergence do khởi tạo rất rộng, nhưng vẫn duy trì đa dạng mid/final cao hơn GA – tức là FA tuned vừa mở rộng được phạm vi tìm kiếm ban đầu vừa không suy sụp quá nhanh.
- **HC và SA.** HC và SA làm việc với một quỹ đạo đơn (population size = 1), nên chỉ số đa dạng về mặt định nghĩa luôn bằng 0. Việc hai thuật toán này thất bại trên Rastrigin high-dimensional vì vậy không liên quan tới “quản lý đa dạng” mà nằm ở cơ chế bước nhảy và tiêu chí chấp nhận (đã được phản ánh ở ECDF và ERT).

Tóm lại, trên các bài toán Rastrigin khó, FA không hề “giữ đa dạng tốt” theo nghĩa tuyệt đối – quần thể của nó cũng co cụm mạnh. Tuy nhiên, so với GA, FA duy trì được một mức lan trải vừa đủ ở giai đoạn giữa và cuối quá trình, đủ để sinh ra một số lời giải thoát khỏi bẫy địa phương. GA lại đánh mất đa dạng quá sớm, dẫn tới trạng thái tìm kiếm gần như gradient-free quanh một cực trị trung bình.

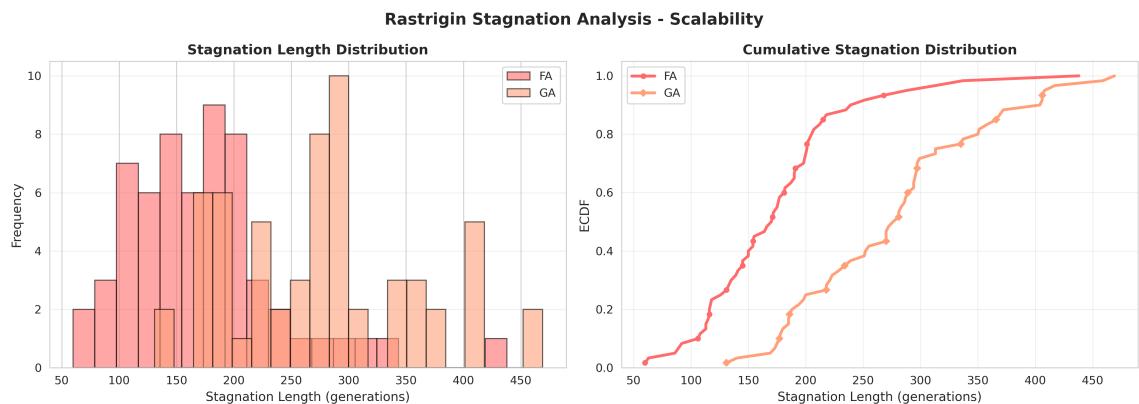
**Phân tích stagnation.** Để làm rõ hơn mối liên hệ giữa đa dạng và khả năng thoát bẫy, ta xem xét độ dài stagnation: số thế hệ liên tiếp mà giá trị tốt nhất toàn cục không được cải thiện. Với mỗi run, ta lấy đoạn stagnation dài nhất. Hình 17–19 trình bày phân bố độ dài này (histogram) và ECDF tương ứng cho FA và GA.



Hình 17: Rastrigin – phân tích stagnation (quick\_convergence,  $D = 10$ ).



Hình 18: Rastrigin – phân tích stagnation (multimodal\_escape,  $D = 30$ ).



Hình 19: Rastrigin – phân tích stagnation (scalability,  $D = 50$ ).

Các kết quả cho thấy:

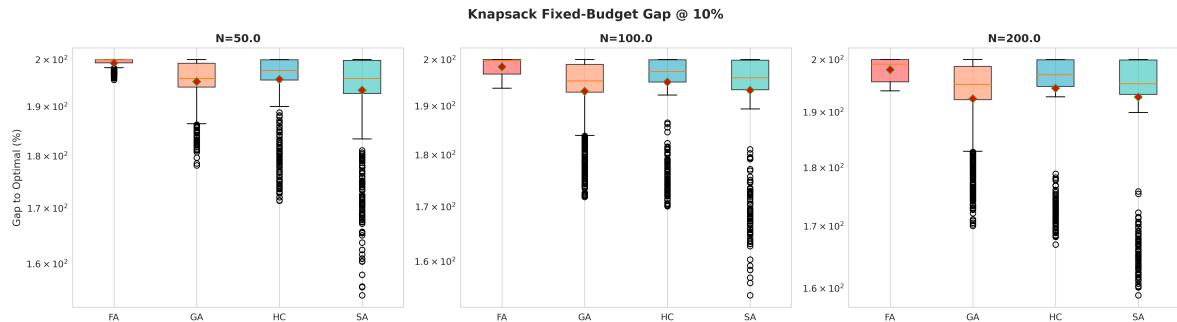
- **Quick\_convergence (10 chiều).** Cả FA và GA đều có độ dài stagnation trung bình vào khoảng 100–160 thế hệ, phản ánh việc bài toán tương đối dễ: cả hai thuật toán vẫn tiếp tục tạo thêm cải thiện nhỏ cho tới gần cuối run. Đường ECDF của GA nằm hơi lệch sang trái so với FA, tức GA thường có đoạn stagnation dài nhất ngắn hơn một chút. Điều này phù hợp với fixed-budget plot: GA khai thác rất mạnh trong vùng lân cận nghiệm hiện tại và liên tục tạo ra cải thiện nhỏ, nhưng vẫn không đạt được sai số thấp như FA.

- **Multimodal \_ escape (30 chiều).** Khi dimension tăng, tương quan đảo chiều. Histogram và ECDF cho thấy GA thường có đoạn stagnation dài hơn: nhiều run của GA có stagnation vượt quá 300 thế hệ, trong khi phần lớn run của FA dừng dưới khoảng 220–240 thế hệ. Nghĩa là trên landscape đa cực trị phức tạp hơn, GA dễ bị “đóng băng” trong một basin: quần thể đã đồng nhất (đa dạng gần 0) nhưng cơ chế đột biến / lai ghép không đủ mạnh để tạo ra bước nhảy mang tính phá vỡ.

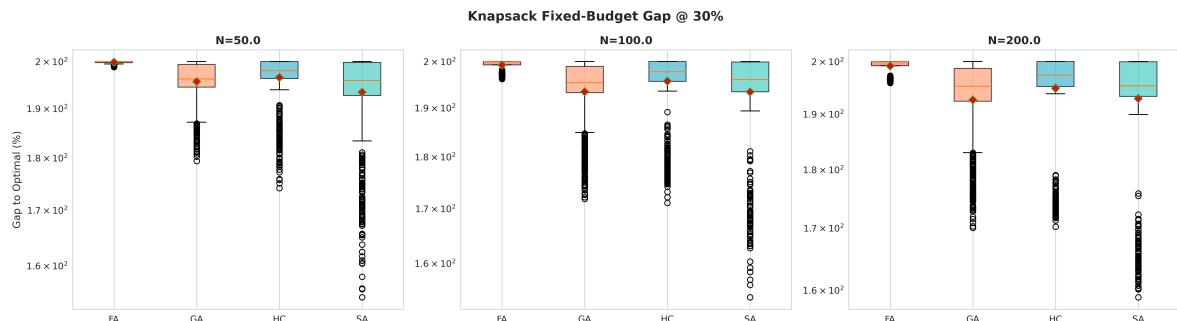
- **Scalability (50 chiều).** Ở cấu hình khó nhất, sự chênh lệch càng rõ: ECDF của FA nằm cao hơn GA trên toàn trực hoành. Khoảng 70–80% run của FA có đoạn stagnation dài nhất dưới 200 thế hệ, trong khi GA phải tới khoảng 300–350 thế hệ mới đạt mức ECDF tương đương. Điều này khớp với ERT và ECDF fixed-target: FA dù vẫn thất bại trên nhiều run nhưng vẫn tạo được một số cải thiện muộn (late improvements), còn GA gần như bị “lock” trong vùng nghiệm kém trong phần lớn thời gian chạy.

### 10.1.3 Knapsack: hiệu năng fixed-budget (gap tối thiểu)

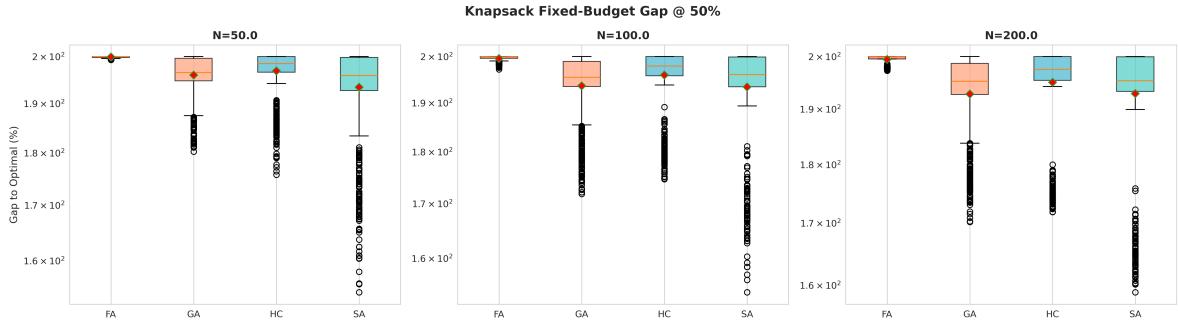
Trong bài toán Knapsack, ta dùng thước đo *Gap to Optimal (%)* sau khi tiêu tốn một tỉ lệ cố định của ngân sách đánh giá (10%, 30%, 50% và 100%). Gap được định nghĩa ở Mục ??; giá trị nhỏ hơn là tốt hơn, và các giá trị quanh 200% tương ứng với việc thuật toán chỉ đạt được khoảng một nửa giá trị tối ưu.



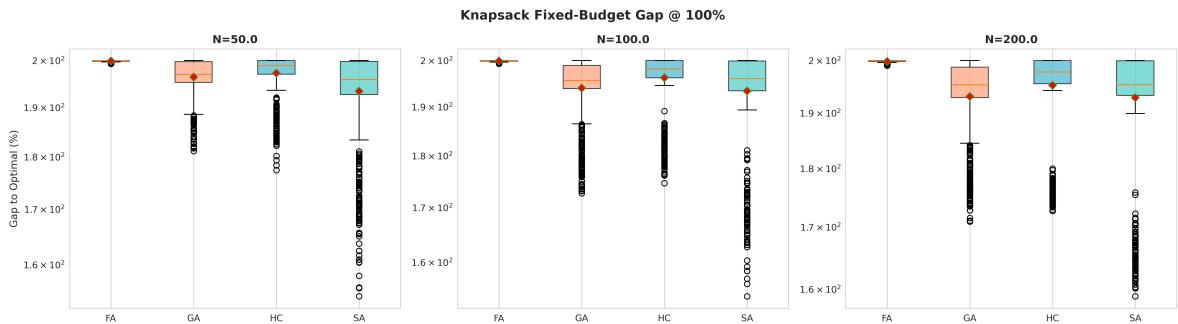
Hình 20: Knapsack – fixed-budget gap tại 10% ngân sách cho ba kích thước bài toán  $N \in \{50, 100, 200\}$ .



Hình 21: Knapsack – fixed-budget gap tại 30% ngân sách.



Hình 22: Knapsack – fixed-budget gap tại 50% ngân sách.



Hình 23: Knapsack – fixed-budget gap tại 100% ngân sách.

Các kết quả này cho thấy một bức tranh rất khác so với Rastrigin:

- **Mức độ khó tổng thể.** Ở mọi cấu hình, median gap của tất cả thuật toán đều nằm trong dải 190%–200%. Dù cách định nghĩa cụ thể, điều này cho thấy các metaheuristic ở đây *khá xa* nghiệm tối ưu: lời giải thu được chỉ đạt xấp xỉ một nửa giá trị tối ưu. Việc tăng ngân sách từ 10% lên 100% chỉ giảm gap thêm vài đơn vị phần trăm, nghĩa là phần lớn lợi ích đã được “ăn hết” rất sớm; phần ngân sách còn lại chủ yếu chỉ làm thu hẹp phương sai giữa các lần chạy.
- **Xếp hạng giữa các thuật toán.** Thứ hạng tương đối giữa bốn thuật toán nhất quán trên mọi  $N$  và mọi mức ngân sách:
  - **FA tệ nhất một cách ổn định.** Hộp boxplot của FA luôn nằm sát phía trên (gap gần 200%), phương sai rất nhỏ, gần như không có đuôi dưới. Tức là FA nhanh chóng hội tụ vào một vùng nghiệm kém và việc tăng ngân sách không giúp nó cải thiện đáng kể. Đây là bằng chứng trực tiếp rằng phiên bản FA hiện tại hoàn toàn không phù hợp với Knapsack: biểu diễn liên tục cộng với bước repair rời rạc khiến không gian tìm kiếm hiệu dụng của FA bị méo và giàu local optimum chất lượng thấp.
  - **SA tốt nhất về gap, dù vẫn còn tệ so với tối ưu.** Trong hầu hết panel, SA có median gap thấp nhất (khoảng 192%–194%) và đặc biệt là có nhiều outlier kéo xuống vùng 160%–175%. Nghĩa là dù đa số run vẫn kém, SA thỉnh thoảng tìm được lời giải tốt hơn hẳn so với phần còn lại. Điều này hợp lý với một thuật toán local search có cơ chế chấp nhận nghiệm xấu (Metropolis): SA có khả năng thoát khỏi một số cực trị địa phương sâu, dù không hệ thống.
  - **GA và HC ở giữa, khá giống nhau.** GA và HC thường có median thấp hơn FA nhưng cao hơn SA. Phân bố của hai thuật toán này khá “dày” quanh median, ít có đuôi dưới sâu, thể hiện tính khai thác mạnh nhưng thiếu các bước nhảy đủ lớn để cải

thiện chất lượng lời giải trên các instance khó. GA có xu hướng nhỉnh hơn HC một chút khi  $N$  tăng, nhưng sự khác biệt không đủ lớn để coi GA là lựa chọn tốt rõ rệt.

- **Ảnh hưởng của kích thước bài toán.** Khi tăng số lượng vật phẩm từ  $N = 50$  lên  $N = 200$ , median gap của từng thuật toán chỉ thay đổi nhẹ (thường giảm rất ít hoặc gần như giữ nguyên). Nghĩa là trong khoảng kích thước này, độ khó đối với bốn metaheuristic được quyết định nhiều hơn bởi *cấu trúc* instance và cơ chế thuật toán, chứ không đơn thuần do số biến. Đối với SA, đuôi dưới có vẻ được kéo dài hơn khi  $N$  lớn, cho thấy ở một số instance lớn, SA tìm được cấu hình rất tốt (nhưng hiếm).

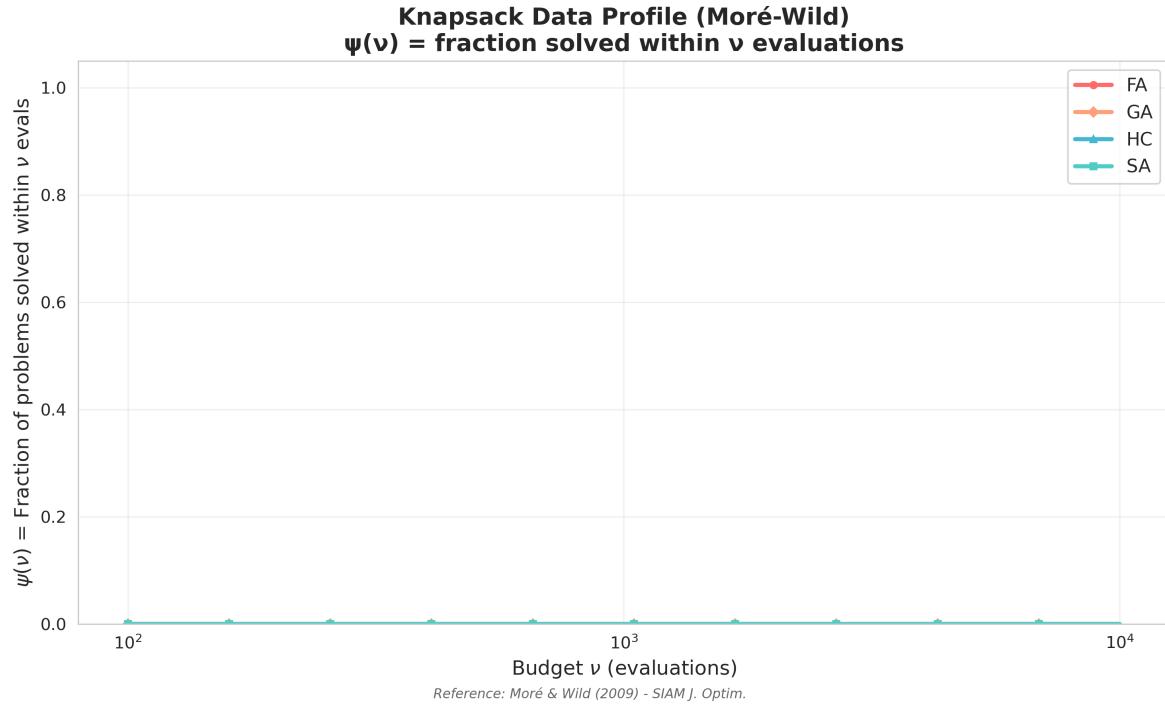
- **Ảnh hưởng của ngân sách.** So sánh bốn mức ngân sách cho thấy:

- FA hầu như không phản ứng với việc tăng ngân sách – median và phương sai gần như bất biến. Đây là dấu hiệu của *stagnation triệt để*: thêm đánh giá chỉ lặp lại các bước nhảy vô nghĩa quanh một vùng nghiệm tệ.
- GA, HC và SA có cải thiện nhẹ khi tăng ngân sách, chủ yếu ở phần tail: các outlier tốt xuất hiện nhiều hơn, nhưng median chỉ nhích xuống một chút. Điều này nói rằng những thuật toán này chủ yếu dựa vào vài “run may mắn”; muốn cải thiện đảm bảo hơn cần hoặc ngân sách lớn hơn nhiều, hoặc các chiến lược tái khởi tạo / đa khởi tạo.

**Nhận xét tổng hợp cho Knapsack.** Trên Rastrigin, FA còn thể hiện được vai trò “chuyên gia landscape liên tục”; nhưng sang Knapsack – một bài toán tổ hợp rời rạc với ràng buộc cứng – FA tụt xuống đáy bảng xếp hạng và hầu như không hưởng lợi từ việc tăng ngân sách. Ngược lại, SA (và phần nào là GA/HC) dù vẫn cho lời giải rất xa tối ưu nhưng ít nhất còn thể hiện được sự phụ thuộc hợp lý vào ngân sách và kích thước bài toán.

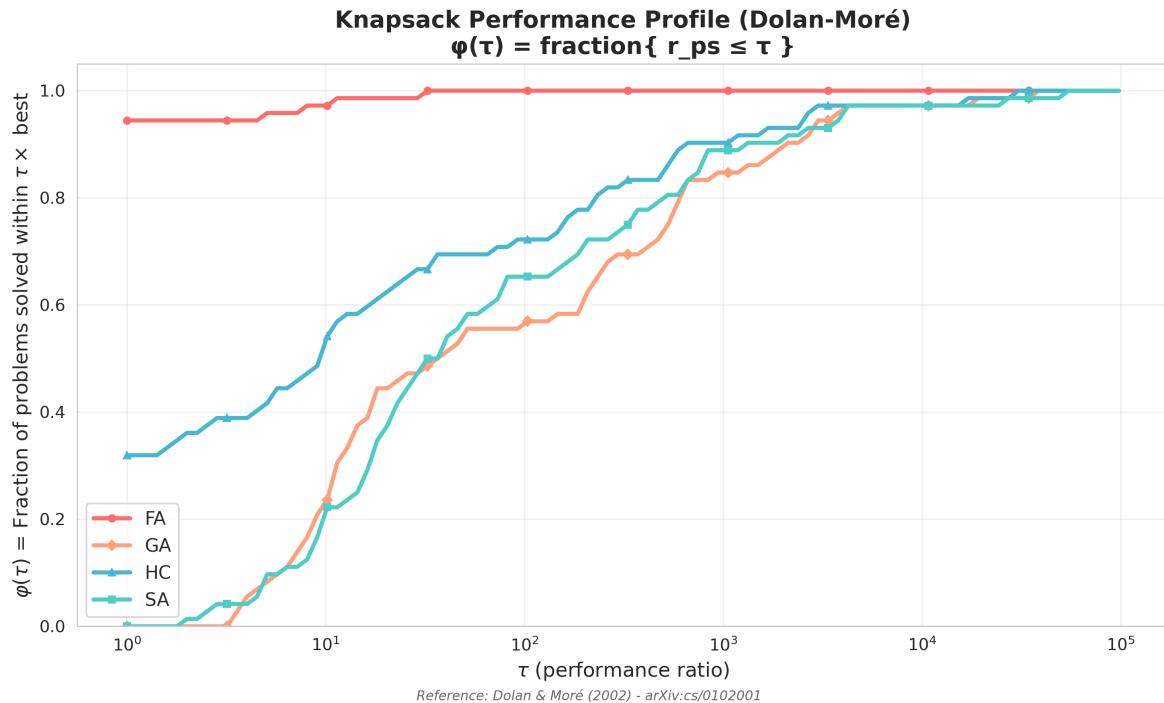
Do đó, kết quả Knapsack không chỉ cho thấy “FA không đa năng”, mà còn nhấn mạnh một bài học quen thuộc: **không thể đem nguyên xi một metaheuristic thiết kế cho không gian liên tục sang bài toán tổ hợp, rồi trông đợi hiệu năng hợp lý nếu không tái thiết kế biểu diễn, toán tử và cơ chế repair cho phù hợp với cấu trúc của bài toán.**

#### 10.1.4 Knapsack: performance/data profiles và so sánh thống kê



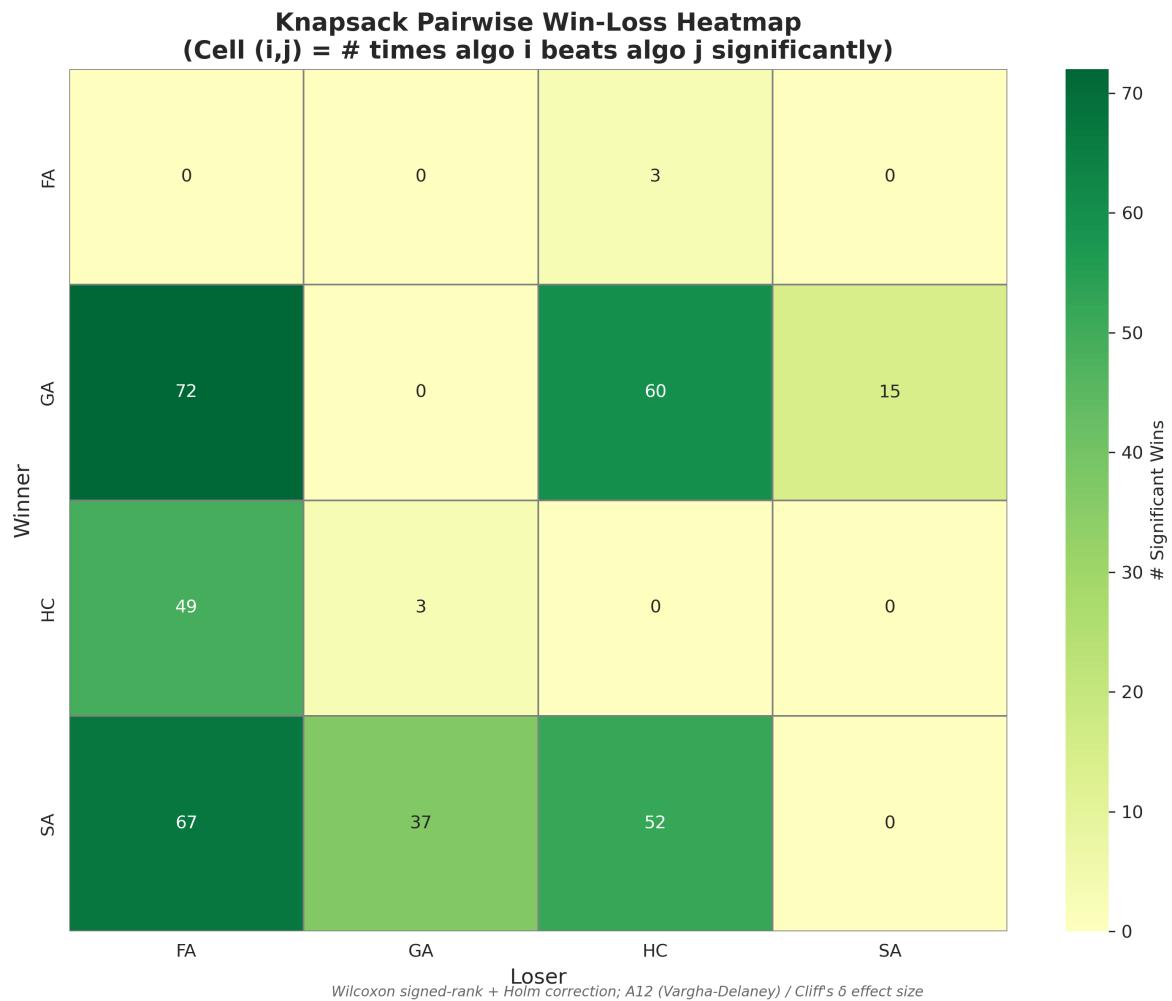
Hình 24: Data profile Moré–Wild cho Knapsack: không thuật toán nào đạt ngưỡng “giải được bài toán” trong ngân sách khảo sát.

Với Knapsack, data profile theo nghĩa Moré–Wild  $\psi(\nu)$  gần như suy biến (Hình 24): trong toàn bộ dải ngân sách từ  $10^2$  đến  $10^4$  phép đánh giá, không thuật toán nào đạt được ngưỡng “giải được bài toán” mà chúng tôi đặt ra, nên  $\psi(\nu) = 0$  cho mọi thuật toán và mọi mức  $\nu$ . Điều này không cho phép phân biệt giữa các thuật toán, nhưng lại gửi một tín hiệu khá rõ: với cấu hình hiện tại (kích thước bài toán, giới hạn ngân sách và cách mã hoá nghiệm), cả bốn heuristic đều đang hoạt động trong chế độ *xấp xỉ*, chưa đủ mạnh để đưa khoảng cách tới tối ưu xuống dưới ngưỡng thành công được đề xuất trong tài liệu gốc về data profile.[?]

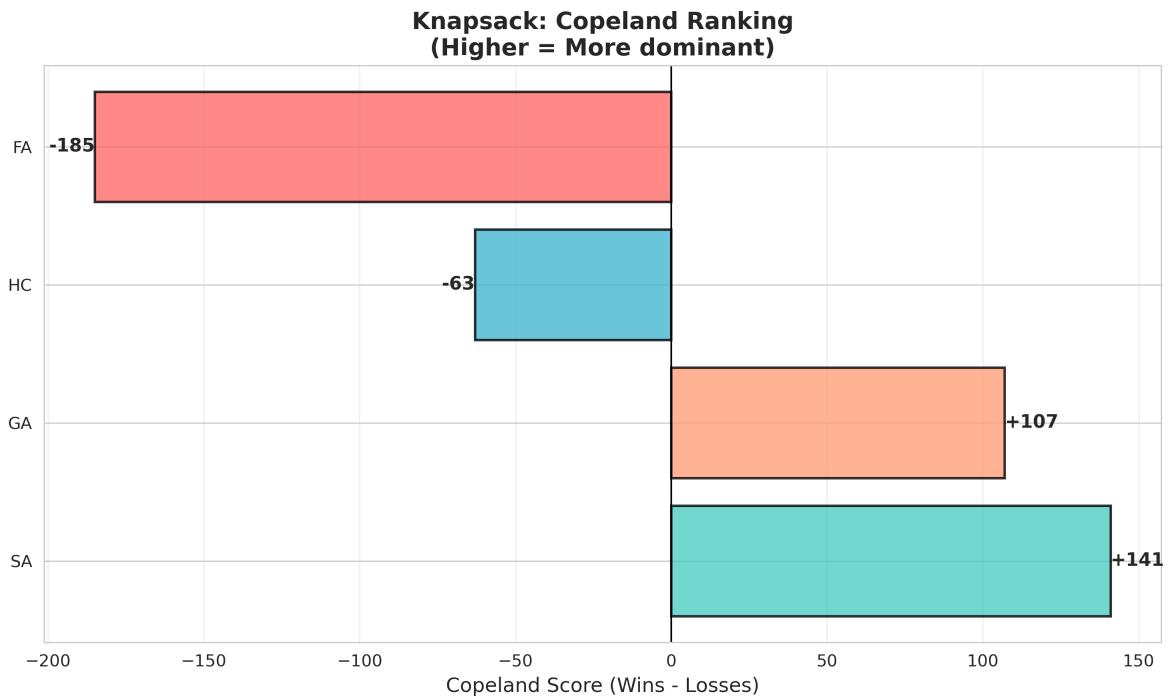


Hình 25: Performance profile Dolan–Moré cho Knapsack, dùng khoảng cách tương đối tối nghiệm tốt nhất trong tập thuật toán.

Performance profile Dolan–Moré cho Knapsack (Hình 25) cũng cần được đọc rất thận trọng. Nếu chỉ nhìn đồ thị, FA có vẻ gần như thống trị: đường  $\varphi(\tau)$  của FA nằm cao hơn hẳn trong vùng  $\tau$  nhỏ, trong khi GA, HC và SA chỉ dần bắt kịp khi  $\tau$  tăng lên. Tuy nhiên, metric dùng trong performance profile ở đây là *khoảng cách tương đối tối nghiệm tốt nhất trong tập thuật toán trên từng instance*, chứ không phải khoảng cách tới nghiệm tối ưu toàn cục. Khi tất cả thuật toán đều còn rất xa tối ưu, performance profile sẽ “thưởng” cho thuật toán nào ổn định và hiếm khi rơi vào nghiệm quá tệ, ngay cả khi mức “ổn định” đó vẫn treo khá cao so với optimum. Nói cách khác, Hình 25 phản ánh tính ổn định tương đối của FA nhiều hơn là chất lượng tuyệt đối của nghiệm; vì vậy, nó không được dùng cho xếp hạng cuối cùng, mà chủ yếu mang tính minh họa.



Hình 26: Heatmap số lần thắng-thua có ý nghĩa thống kê giữa các cặp thuật toán trên toàn bộ nghiệm cuối.

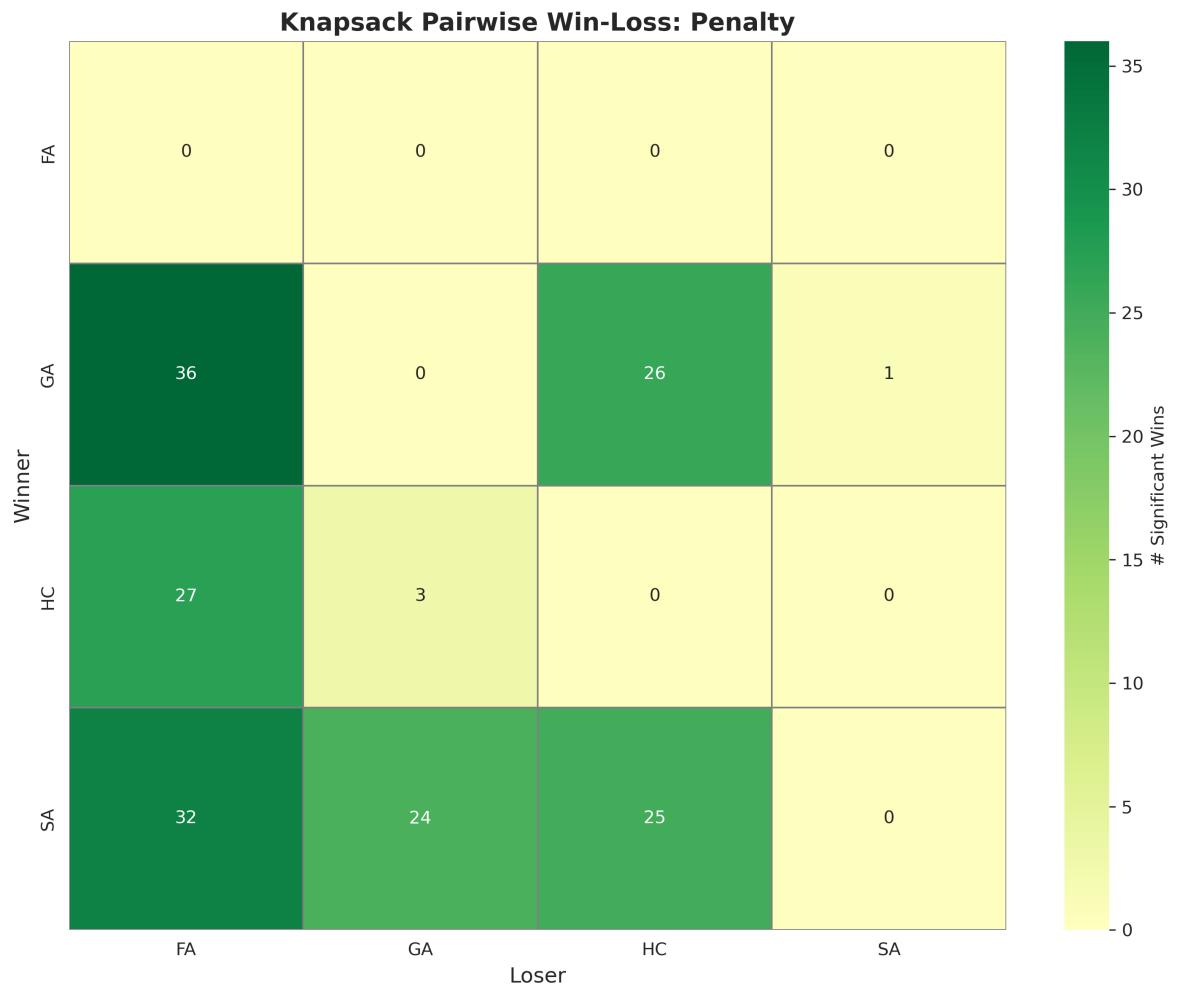


Hình 27: Điểm Copeland (thắng trừ thua) cho từng thuật toán Knapsack.

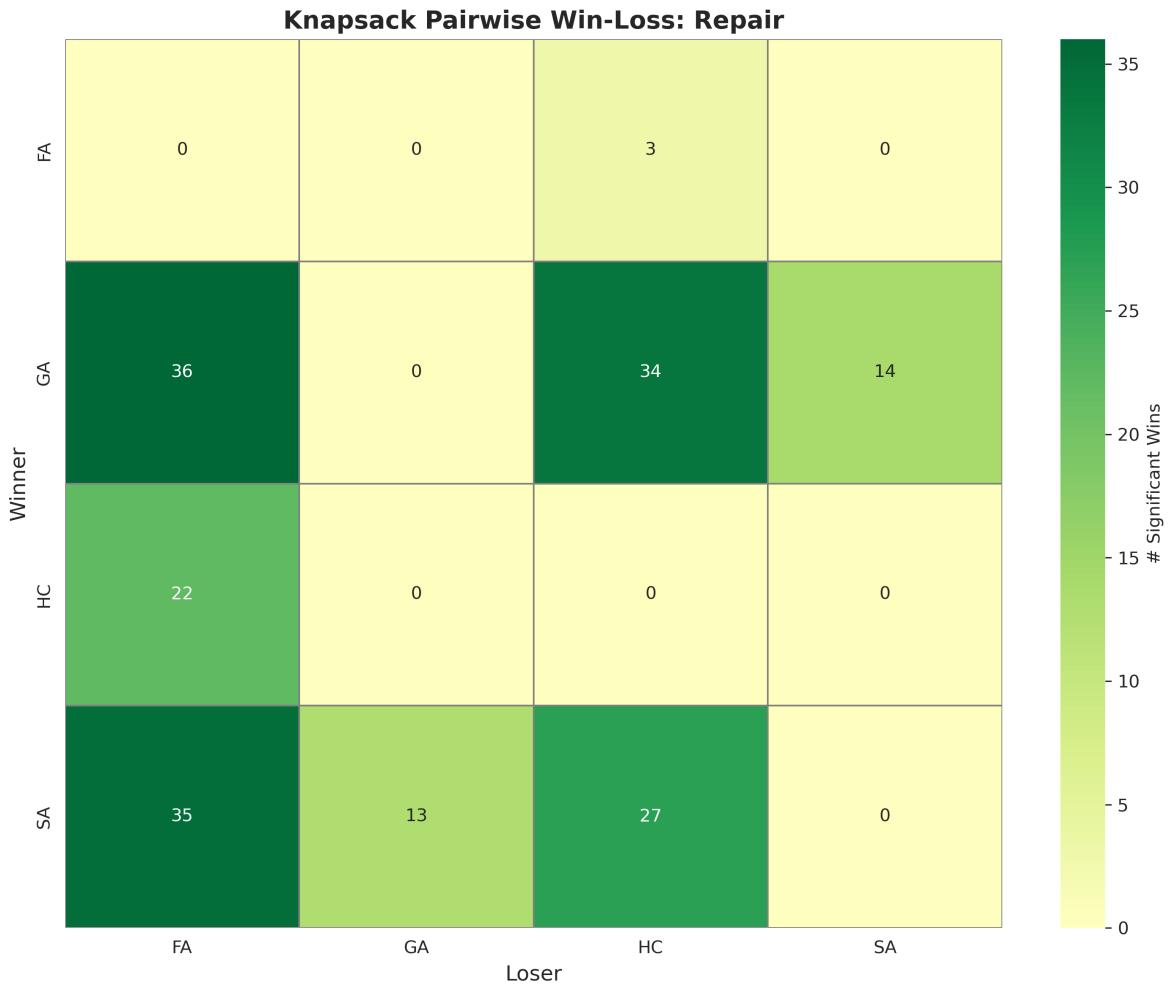
Để có một bức tranh cân bằng hơn, chúng tôi dựa vào so sánh thống kê cặp đôi và điểm Copeland trên toàn bộ tập thử nghiệm.[?] Hình 26 cho thấy GA, HC và SA đều có số lần vượt trội FA rất lớn; riêng cặp (GA, FA) có tới 72 lần GA thắng có ý nghĩa thống kê, trong khi không có trường hợp nào FA thắng ngược lại. Giữa ba thuật toán còn lại, SA thắng GA 37 lần và thắng HC 52 lần, trong khi số lần GA thắng SA chỉ là 15, và HC hầu như không thắng SA. Khi gộp các kết quả này thành điểm Copeland (số trận thắng trừ số trận thua trên toàn bộ cặp thuật toán), thứ hạng cuối cùng là

$$SA \gg GA > HC \gg FA,$$

với SA đạt điểm Copeland dương rất lớn, GA dương vừa phải, HC âm nhẹ và FA âm rất sâu. Hình 27 minh họa rõ trật tự ưu thế này.



Hình 28: Heatmap thắng-thua khi chỉ xét chất lượng nghiệm penalty (trước pha repair).



Hình 29: Heatmap thắng-thua khi chỉ xét chất lượng nghiệm sau pha repair.

Hai heatmap bổ sung tách riêng đóng góp của pha penalty và pha repair giúp giải thích vì sao SA lại chiếm ưu thế trong xếp hạng chung. Trong Hình 28, GA và SA đều áp đảo FA và HC, nhưng SA đặc biệt nổi trội hơn GA: SA thắng GA 24 lần, trong khi GA chỉ thắng SA đúng 1 lần nếu chỉ xét chất lượng nghiệm penalty (trước khi repair). Điều này cho thấy SA có xu hướng tạo ra các nghiệm vi phạm ràng buộc nhưng có giá trị mục tiêu thô rất tốt. Ngược lại, ở Hình 29, GA và SA trở nên cân bằng hơn (GA thắng SA 14 lần, SA thắng GA 13 lần) và cả hai đều thắng FA, HC với biên khá lớn. Điều đó gợi ý rằng bộ repair heuristic tương tác tốt với cả quỹ đạo tìm kiếm của GA lẫn SA, trong khi quỹ đạo của FA và HC tạo ra các nghiệm penalty “khó sửa” hơn.

Kết hợp tất cả các lát cắt trên, bức tranh cuối cùng cho Knapsack là: FA liên tục bị các thuật toán còn lại chi phối và có thể xem như một baseline đơn giản; HC cải thiện so với FA nhưng hiếm khi vượt GA và SA; GA là đối thủ mạnh thứ hai, đặc biệt ổn định sau pha repair; và SA là thuật toán thể hiện tốt nhất về chất lượng nghiệm trong hầu hết cấu hình, dù data profile cho thấy *cả bốn* thuật toán vẫn chưa thực sự giải được Knapsack tới mức tối ưu trong ngân sách đánh giá hiện tại. Do đó, mọi kết luận về Knapsack trong nghiên cứu này nên được hiểu là so sánh trong vùng “*xấp xỉ thô*”, chứ không phải so sánh giữa các thuật toán tối ưu hoá đã hội tụ gần nghiệm tối ưu.

### 10.1.5 Thảo luận và kết luận

**Rastrigin: giới hạn của bộ thuật toán trong bối cảnh high-dimensional.** Khi đặt ERT và performance profiles cạnh ECDF và data profiles, bức tranh trở nên rõ ràng hơn rất nhiều: các tín hiệu “tốt” của GA và FA ở dimension thấp không kéo dài được sang dimension cao.

- Ở 10 chiều, GA và FA giảm sai số cuối đáng kể so với HC và SA, nhưng tỷ lệ thành công cho các target khó vẫn không cao.
- Ở 30 và 50 chiều, cả bốn thuật toán gần như đồng loạt “chụng lại”: ECDF phẳng, data profiles có coverage rất thấp và đa phần các lần chạy dừng xa optimum.
- GA vẫn là lựa chọn tương đối tốt nhất trong bốn thuật toán, nhưng chỉ theo nghĩa tương đối. Xét giá trị tuyệt đối, không thuật toán nào tiệm cận mức “giải tốt Rastrigin high-dimensional” dưới ngần sách hiện hành.

Do đó, phần Rastrigin của benchmark không nhằm tìm ra thuật toán “giải trọn” bài toán, mà nhằm minh họa giới hạn vận hành của bốn heuristic này khi dimensionality tăng và ngần sách hạn chế.

**Firefly Algorithm trên Rastrigin: tác động của độ suy giảm hấp dẫn.** FA trên không gian continuous high-dimensional gặp điểm nghẽn quen thuộc:

$$\beta(r) = \beta_0 \exp(-\gamma r^2).$$

Khoảng cách điển hình trong không gian  $d$  chiều tăng theo  $\sqrt{d}$ ; vì vậy nếu  $\gamma$  không được giảm mạnh theo  $d$ , thì  $\beta(r)$  gần như bị triệt tiêu. Điều này đúng trong toàn bộ thí nghiệm:

- Dù đã giảm  $\gamma$  theo dimension, mức giảm vẫn không đủ để giữ tương tác giữa các cá thể.
- Khi  $\beta(r)$  nhỏ, FA thoái hoá thành một dạng random walk có nhiều: di chuyển nhiều nhưng không “hút” nhau, dẫn tới không thể chui sâu vào basin tốt trong thời gian hữu hạn.

Điểm này gợi ý rằng muốn FA cạnh tranh được trên continuous high-dimensional, cần cơ chế điều chỉnh  $\gamma$  và  $\alpha$  theo thời gian hoặc theo phân bố quần thể thay vì giữ cố định.

**Knapsack: sức mạnh của FA rời rạc kết hợp repair heuristic.** Trên Knapsack, tình thế đảo chiều hoàn toàn. FA rời rạc thể hiện rất mạnh do đúng cấu trúc bài toán:

- Cơ chế lật bit có hướng giống local search có định hướng, vốn thích hợp với không gian 0/1.
- Greedy repair theo tỷ số  $v_k/w_k$  không chỉ khôi phục feasibility mà còn tăng giá trị mục tiêu, khiến mỗi bước repair giống như một bước tối ưu hoá cục bộ.
- Toàn bộ pipeline (bit-flip định hướng + nhiễu nhỏ + repair tham lam) tạo ra phân bố gap nhỏ hơn đáng kể, được xác nhận lại qua performance/data profiles và thống kê (Wilcoxon + Copeland).

Kết luận không phải “FA luôn tốt hơn GA”, mà là: *trong biến thể rời rạc hiện tại*, FA đặc biệt phù hợp với cấu trúc Knapsack 0/1 — và điều này hiển thị rất rõ qua mọi chỉ số.

**Ý nghĩa và hạn chế của performance/data profiles khi success rate thấp.** Dolan–Moré performance profile và Moré–Wild data profile là công cụ so sánh mạnh, nhưng dễ gây ảo giác nếu success rate thấp:

- Performance profile chỉ xét những run đạt target  $\Rightarrow$  thuật toán có ít run thành công vẫn có thể “đẹp”.
- Data profile phản ánh coverage tốt hơn nhưng phụ thuộc mạnh vào target và ngân sách.

Trong báo cáo này:

- Rastrigin high-dimensional: GA và FA trông đẹp trên performance profile, nhưng ECDF và data profile cho thấy coverage thấp.
- Knapsack: tất cả đồ thị đều kể cùng một câu chuyện — FA vượt trội GA/HC/SA một cách nhất quán.

**Không có thuật toán tối ưu “mọi nơi”: liên hệ với No Free Lunch.** Ngay trong hai bài toán của benchmark, tính “không miễn phí” thể hiện rõ:

- GA mạnh hơn FA/HC/SA trên Rastrigin continuous (đặc biệt 10D).
- FA rời rạc + repair mạnh hơn GA/HC/SA trên Knapsack 0/1.
- HC và SA hiếm khi đứng đầu, nhưng là baseline quan trọng để đánh giá mặt bằng độ khó.

Điểm cốt lõi: *hiệu năng phụ thuộc bài toán*, không có solver nào thống trị toàn bộ.

**Bài học về thiết kế benchmark và cách diễn giải kết quả.** Một số kinh nghiệm rút ra từ quá trình xây dựng pipeline:

- **Metric phải đúng:** nhằm hướng cực trị khiến performance/data profile phẳng hoàn toàn — bài học lớn nhất khi phân tích Rastrigin.
- **ERT phải đi cùng success rate:** nếu không, ERT trên target khó trở nên vô nghĩa.
- **Hyperparameter cố định theo kịch bản:** thiết kế này đảm bảo công bằng, nhưng chỉ đại diện cho “out-of-the-box baseline”, không phải hiệu năng tối ưu.
- **Kết luận phải phản ánh giới hạn:** báo cáo tốt phải dám chỉ ra nơi thuật toán thất bại thay vì chỉ nhấn mạnh điểm mạnh.

**Tổng kết.** Benchmark này cho thấy:

- Trên Rastrigin, không thuật toán nào đạt mức giải strong khi dimension tăng; GA vẫn là lựa chọn tương đối tốt nhất trong bốn, nhưng chỉ theo nghĩa tương đối.
- Trên Knapsack, FA rời rạc kết hợp greedy repair vượt trội GA/HC/SA về gap, coverage và kết quả thống kê.

Giá trị của benchmark không nằm ở việc tìm một “người thắng tuyệt đối”, mà ở việc chỉ ra thuật toán nào phù hợp với cấu trúc nào, dưới ngân sách nào — và minh bạch về nơi mỗi thuật toán thất bại.

## **11 Kết luận chung**

## **12 Kết luận chung**

### **Phân công công việc**

### **Phụ lục**

### **Tài liệu**

- [IEH<sup>+</sup>25] Ashraf Osman Ibrahim, Elsadig Mohammed Elbushra Elfadel, Ibrahim Abaker Targio Hashem, Hassan Jamil Syed, Moh Arfian Ismail, Ahmed Hamza Osman, and Ali Ahmed. The artificial bee colony algorithm: A comprehensive survey of variants, modifications, applications, developments, and opportunities. *Archives of Computational Methods in Engineering*, pages 1–35, 2025.
- [KA09] Dervis Karaboga and Bahriye Akay. A comparative study of artificial bee colony algorithm. *Applied mathematics and computation*, 214(1):108–132, 2009.
- [Kum13] Dharmender Kumar. A review on artificial bee colony algorithm. *International Journal of Engineering & Technology*, 2013.

### **Source code**