NP 難題演算法系列 · 第 13 篇

遺傳演算法

Genetic Algorithm — Evolution as an Optimizer

把「物競天擇」寫成程式：不懂問題結構也能逼近好解的通用武器，本篇以 TSP 與背包實戰

Contents

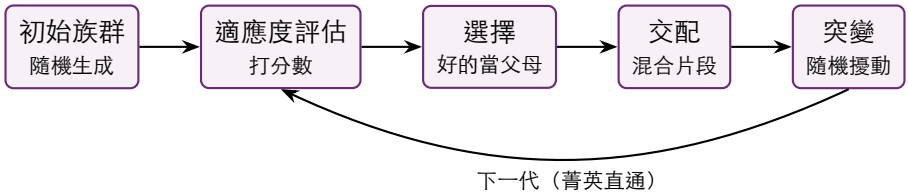
1 核心思想

白話比喻

你不知道最好的路線長什麼樣，但你會**比較**兩條路線誰好。那就這樣辦：養一群隨機路線，讓好的容易「生小孩」（混合兩條路線的片段），偶爾「突變」（隨機小改動），差的自然淘汰。幾百代之後，族群裡就長出很好的路線——這就是遺傳演算法，達爾文當工程師。

GA 是**元啟發式（metaheuristic）**：不保證最優、不保證近似比，但幾乎不挑問題——只要能**編碼**解、能**打分數**，就能跑。它是 NP-Hard 工具箱的最後一格：精確解太慢、近似演算法不存在或太差、問題結構混沌時的萬用鑰匙。

1.1 五個組件



- **編碼**：解如何表示成「染色體」。TSP 用**排列**、背包用**位元字串**——編碼決定一切後續設計。
- **選擇**：**錦標賽選擇**（隨機抓 k 個、取最好）簡單、好調、對適應度縮放不敏感，是實務首選。
- **交配**：把兩個好解的「基因片段」重組。**必須尊重編碼的合法性**（見下節）。
- **突變**：維持多樣性、逃離局部最優的保險絲。
- **菁英保留**：每代把最好的幾個直接抄進下一代——保證最佳解單調不退步。

2 關鍵設計：TSP 的排列編碼

2.1 為什麼不能用普通交配

染色體是城市排列。單點交配把 $[0,3,1,2,4]$ 和 $[0,2,4,1,3]$ 前後段拼起來會得到 $[0,3,1,1,3]$——**城市重複又缺漏，根本不是合法路線**。排列編碼需要專用算子。

2.2 順序交配 OX（Order Crossover）

1. 從父 A 抄一段連續區間到子代同位置。
2. 剩下的洞，按**父 B 的出現順序**把缺的城市依序填入。

$$\underbrace{A=[0,\boxed{3,1},2,4]}_{\text{抄中段}}+B=[0,2,4,1,3]\ \Rightarrow\ \text{子}=[0,\boxed{3,1},2,4](\text{B 順序補}2,4)$$

子代保有 A 的一個連續片段與 B 的相對順序——兩個父母的「好基因」都有機會傳下去，且**永遠是合法排列**。

2.3 突變：交換與反轉

交換突變（隨機兩城互換）擾動小；**反轉突變**（隨機區間反轉）正是 2-opt 的動作——把局部搜尋的智慧藏進突變裡，是 TSP GA 的標準配方。

3 第二種編碼：背包的位元字串

背包解 $=n$ 位 0/1（拿或不拿）。交配用單點切割、突變用位元翻轉——教科書原味 GA。多了一個問題：**超重的非法個體**怎麼辦？本篇採**修復**策略：反覆丟出「價值密度最低」的物品直到合法。（另兩種流派：罰分——適應度扣超重量的懲罰；或設計不會產生非法解的編碼。）

同一套引擎、換編碼與適應度就能解完全不同的問題——這就是 GA「泛用」的意思，也是它被工程師偏愛的原因。

4 調參的三個旋鈕

參數	太小	太大
族群大小	多樣性不足、早熟收斂	每代太貴、收斂慢
突變率	卡死在局部最優	退化成隨機亂搜
菁英數	好解可能絕種	過早壟斷、多樣性崩潰

本篇程式 Example 2、3 直接掃這些參數給你看：突變率 0 的族群明顯早熟（卡在 +0.8%），0.05–0.25 都健康；這個規模下族群 20 就夠——**參數的「感覺」必須靠實驗建立，不要背數字**。

5 完整 C++ 程式

包含：TSP GA（排列編碼、錦標賽、OX、交換 + 反轉突變、菁英）、Held–Karp 對照最優、族群/突變率參數掃描、背包 GA（位元編碼 + 修復）對照 DP。編譯：

```
g++ -std=c++17 -O2 -Wall -Wextra -o genetic_algorithm genetic_algorithm.cpp
```

```cpp
// genetic_algorithm.cpp
// ============================================================================
// Genetic Algorithm（遺傳演算法）：以 TSP 為載體的完整實作
//   染色體 = 城市排列（permutation encoding）
//   選擇   = 錦標賽選擇（tournament selection）
//   交配   = 順序交配 OX（Order Crossover，保持排列合法）
//   突變   = 交換突變 + 反轉突變（2-opt 式）
//   菁英   = 每代保留最好的個體
//
// 另附：GA 解 0/1 背包（位元字串編碼 + 修復函數）示範泛用性
//
// Compile: g++ -std=c++17 -O2 -Wall -Wextra -o genetic_algorithm
    genetic_algorithm.cpp
// ============================================================================
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <cmath>
#include <random>
#include <iomanip>

using namespace std;

// ================= GA for TSP =================
class GATSP {
public:
    struct Params {
        int populationSize = 200;
        int generations    = 500;
        double crossoverRate = 0.9;
        double mutationRate  = 0.25;
        int tournamentK      = 4;
        int eliteCount       = 4;
    };

    GATSP(const vector<vector<double>>& dist, unsigned seed)
        : d(dist), n((int)dist.size()), rng(seed) {}

```

```
    // 執行 GA，log 每隔 logEvery 代印出目前最佳
    pair<double, vector<int>> run(const Params& p, int logEvery = 0) {
        // --- 初始族群：隨機排列 ---
        vector<vector<int>> pop(p.populationSize);
        for (auto& c : pop) {
            c.resize(n);
            iota(c.begin(), c.end(), 0);
            shuffle(c.begin() + 1, c.end(), rng);   // 固定城市 0 為起點
        }
        vector<double> fit(p.populationSize);
        auto evalAll = [&]() {
            for (int i = 0; i < p.populationSize; ++i)
                fit[i] = tourLength(pop[i]);
        };
        evalAll();

        for (int gen = 1; gen <= p.generations; ++gen) {
            // --- 排序找菁英 ---
            vector<int> order(p.populationSize);
            iota(order.begin(), order.end(), 0);
            sort(order.begin(), order.end(),
                 [&](int a, int b) { return fit[a] < fit[b]; });

            vector<vector<int>> next;
            next.reserve(p.populationSize);
            for (int e = 0; e < p.eliteCount; ++e)       // 菁英直通
                next.push_back(pop[order[e]]);

            uniform_real_distribution<double> u01(0, 1);
            while ((int)next.size() < p.populationSize) {
                const auto& pa = pop[tournament(fit, p.tournamentK)];
                const auto& pb = pop[tournament(fit, p.tournamentK)];
                vector<int> child = (u01(rng) < p.crossoverRate)
                                    ? orderCrossover(pa, pb) : pa;
                if (u01(rng) < p.mutationRate) mutate(child);
                next.push_back(std::move(child));
            }
            pop = std::move(next);
            evalAll();

            if (logEvery && gen % logEvery == 0) {
                double best = *min_element(fit.begin(), fit.end());
                cout << "  gen " << setw(4) << gen
                     << "  best = " << fixed << setprecision(2) << best
                        << "\n";
            }
        }
        int bi = min_element(fit.begin(), fit.end()) - fit.begin();
        return {fit[bi], pop[bi]};
    }

    double tourLength(const vector<int>& t) const {
        double len = 0;
        for (int i = 0; i < n; ++i)
            len += d[t[i]][t[(i + 1) % n]];
        return len;
    }
private:
    const vector<vector<double>>& d;
```

```
    int n;
    mt19937 rng;

    int tournament(const vector<double>& fit, int k) {
        uniform_int_distribution<int> pick(0, (int)fit.size() - 1);
        int best = pick(rng);
        for (int i = 1; i < k; ++i) {
            int c = pick(rng);
            if (fit[c] < fit[best]) best = c;
        }
        return best;
    }

    // OX：抄下 A 的一段，剩下的城市按 B 的順序補齊
    vector<int> orderCrossover(const vector<int>& A, const vector<int>&
        B) {
        uniform_int_distribution<int> pos(1, n - 1);      // 位置 0
            固定為城市 0
        int l = pos(rng), r = pos(rng);
        if (l > r) swap(l, r);
        vector<int> child(n, -1);
        vector<bool> used(n, false);
        child[0] = 0;
        used[0] = true;
        for (int i = l; i <= r; ++i) {                    // 抄 A 的中段
            child[i] = A[i];
            used[A[i]] = true;
        }
        int idx = 1;
        for (int i = 0; i < n; ++i) {                     // 依 B 順序補洞
            int city = B[i];
            if (used[city]) continue;
            while (idx < n && child[idx] != -1) ++idx;
            child[idx] = city;
            used[city] = true;
        }
        return child;
    }

    void mutate(vector<int>& c) {
        uniform_int_distribution<int> pos(1, n - 1);
        uniform_real_distribution<double> u01(0, 1);
        if (u01(rng) < 0.5) {                             // 交換突變
            swap(c[pos(rng)], c[pos(rng)]);
        } else {                                          //
            反轉突變（2-opt 式）
            int l = pos(rng), r = pos(rng);
            if (l > r) swap(l, r);
            reverse(c.begin() + l, c.begin() + r + 1);
        }
    }
};

// ---------- Held-Karp（拿真正最優來對照，n 小） ----------
double heldKarp(const vector<vector<double>>& d) {
    int n = (int)d.size();
    const double INF = 1e18;
    vector<vector<double>> dp(1 << n, vector<double>(n, INF));
    dp[1][0] = 0;
```

```
153     for (int S = 1; S < (1 << n); ++S) {
154         if (!(S & 1)) continue;
155         for (int v = 0; v < n; ++v) {
156             if (dp[S][v] >= INF) continue;
157             for (int u = 0; u < n; ++u) {
158                 if (S & (1 << u)) continue;
159                 dp[S | (1 << u)][u] =
160                     min(dp[S | (1 << u)][u], dp[S][v] + d[v][u]);
161             }
162         }
163     }
164     double best = INF;
165     for (int v = 1; v < n; ++v)
166         best = min(best, dp[(1 << n) - 1][v] + d[v][0]);
167     return best;
168 }
169 
170 // ================= GA for 0/1 Knapsack（示範第二種編碼）
    ==================
171 // 染色體 = 位元字串；超重的用「丟出密度最低物品」修復
172 int gaKnapsack(const vector<int>& w, const vector<int>& v, int W,
173                unsigned seed, int generations = 300) {
174     int n = (int)w.size();
175     mt19937 rng(seed);
176     uniform_int_distribution<int> bit(0, 1), pos(0, n - 1);
177     uniform_real_distribution<double> u01(0, 1);
178     int POP = 100;
179 
180     auto repairAndValue = [&](vector<int>& c) {
181         int tw = 0, tv = 0;
182         for (int i = 0; i < n; ++i)
183             if (c[i]) { tw += w[i]; tv += v[i]; }
184         while (tw > W) {                                    //
               修復：丟密度最低
185             int worst = -1;
186             double worstDensity = 1e18;
187             for (int i = 0; i < n; ++i)
188                 if (c[i]) {
189                     double dens = (double)v[i] / w[i];
190                     if (dens < worstDensity) { worstDensity = dens; worst
                        = i; }
191                 }
192             c[worst] = 0;
193             tw -= w[worst];
194             tv -= v[worst];
195         }
196         return tv;
197     };
198 
199     vector<vector<int>> pop(POP, vector<int>(n));
200     vector<int> fit(POP);
201     for (int i = 0; i < POP; ++i) {
202         for (int& g : pop[i]) g = bit(rng);
203         fit[i] = repairAndValue(pop[i]);
204     }
205     for (int gen = 0; gen < generations; ++gen) {
206         vector<vector<int>> next;
207         // 菁英 x2
208         int b1 = max_element(fit.begin(), fit.end()) - fit.begin();
```

```
        next.push_back(pop[b1]);
        next.push_back(pop[b1]);
        auto tourn = [&]() {
            int a = pos(rng) % POP, b = pos(rng) % POP;
            return fit[a] > fit[b] ? a : b;
        };
        while ((int)next.size() < POP) {
            const auto& A = pop[tourn()];
            const auto& B = pop[tourn()];
            vector<int> child(n);
            int cut = pos(rng);                              // 單點交配
            for (int i = 0; i < n; ++i)
                child[i] = i <= cut ? A[i] : B[i];
            if (u01(rng) < 0.3) child[pos(rng)] ^= 1;     // 位元翻轉突變
            next.push_back(std::move(child));
        }
        pop = std::move(next);
        for (int i = 0; i < POP; ++i) fit[i] = repairAndValue(pop[i]);
    }
    return *max_element(fit.begin(), fit.end());
}

int knapsackDP(const vector<int>& w, const vector<int>& v, int W) {
    vector<int> dp(W + 1, 0);
    for (size_t i = 0; i < w.size(); ++i)
        for (int c = W; c >= w[i]; --c)
            dp[c] = max(dp[c], dp[c - w[i]] + v[i]);
    return dp[W];
}

int main() {
    cout << fixed << setprecision(2);
    mt19937 rng(20260724);
    uniform_real_distribution<double> coord(0, 100);

    // ---- 範例 1：GA 解 TSP（n=15），演化過程 + 對照最優 ----
    cout << "=== Example 1: GA on 15-city TSP (watch it evolve) ===\n";
    int n = 15;
    vector<pair<double,double>> pts;
    for (int i = 0; i < n; ++i) pts.push_back({coord(rng), coord(rng)});
    vector<vector<double>> d(n, vector<double>(n));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            d[i][j] = hypot(pts[i].first - pts[j].first,
                            pts[i].second - pts[j].second);
    GATSP ga(d, 12345);
    GATSP::Params p;
    auto [gaLen, gaTour] = ga.run(p, 100);
    double opt = heldKarp(d);
    cout << "GA best   = " << gaLen << "\n";
    cout << "true OPT  = " << opt << " (Held-Karp)\n";
    cout << "gap       = +" << 100 * (gaLen / opt - 1) << "%\n";
    cout << "GA tour   : ";
    for (int c : gaTour) cout << c << " ";
    cout << "\n\n";

    // ---- 範例 2：族群大小的影響 ----
    cout << "=== Example 2: does population size matter? ===\n";
    for (int popSize : {20, 100, 400}) {
```

```
        GATSP::Params q;
        q.populationSize = popSize;
        q.generations = 300;
        GATSP g2(d, 999);
        auto [len, t] = g2.run(q);
        (void)t;
        cout << "population " << setw(3) << popSize
             << " -> best " << len
             << "  (gap +" << 100 * (len / opt - 1) << "%)\n";
    }
    cout << "\n";

    // ---- 範例 3：突變率的影響（太低早熟、太高變亂槍） ----
    cout << "=== Example 3: mutation rate sweep ===\n";
    for (double mr : {0.0, 0.05, 0.25, 0.8}) {
        GATSP::Params q;
        q.mutationRate = mr;
        q.generations = 300;
        GATSP g3(d, 777);
        auto [len, t] = g3.run(q);
        (void)t;
        cout << "mutation " << setw(4) << mr << " -> best " << len
             << "  (gap +" << 100 * (len / opt - 1) << "%)\n";
    }
    cout << "\n";

    // ---- 範例 4：同一套 GA 思想換個編碼解背包 ----
    cout << "=== Example 4: GA on 0/1 knapsack (bit-string encoding)
        ===\n";
    mt19937 rng2(42);
    uniform_int_distribution<int> wd(5, 60), vd(10, 100);
    vector<int> w, v;
    for (int i = 0; i < 30; ++i) { w.push_back(wd(rng2));
        v.push_back(vd(rng2)); }
    int W = 500;
    int gaVal = gaKnapsack(w, v, W, 31337);
    int dpVal = knapsackDP(w, v, W);
    cout << "n=30, W=500 | GA = " << gaVal << ", DP optimal = " << dpVal
         << "  (gap " << 100.0 * (dpVal - gaVal) / dpVal << "%)\n";
    return 0;
}
```

程式碼 1: genetic_algorithm.cpp — 遺傳演算法（TSP 與背包）完整實作

6 執行結果與解讀

```
=== Example 1: GA on 15-city TSP (watch it evolve) ===
  gen   100  best = 316.32
  ...
GA best    = 316.32
true OPT   = 316.32 (Held-Karp)
gap        = +0.00%

=== Example 2: does population size matter? ===
population  20 -> best 316.32  (gap +0.00%)
population 100 -> best 316.32  (gap +0.00%)
population 400 -> best 316.32  (gap +0.00%)

=== Example 3: mutation rate sweep ===
```

```
mutation 0.00 -> best 318.84  (gap +0.80%)
mutation 0.05 -> best 316.32  (gap +0.00%)
mutation 0.25 -> best 316.32  (gap +0.00%)
mutation 0.80 -> best 316.32  (gap +0.00%)

=== Example 4: GA on 0/1 knapsack (bit-string encoding) ===
n=30, W=500 | GA = 1345, DP optimal = 1345  (gap 0.00%)
```

筆記

最有教學價值的是 Example 3 的**突變率 0**：只靠交配重組現有基因、沒有新變異注入，族群很快同質化，卡在離最優 0.8% 的地方**再也出不來**——這就是「早熟收斂」的長相。加上區區 5% 突變率就治好。另外注意：15 城的 TSP 對 GA 是小菜（Held–Karp 都能直接驗證），GA 的主場是 n 上百、精確解徹底絕望之後。

常見誤解

使用 GA 的三大紀律：(1) **先試簡單方法**——貪心、局部搜尋常常又快又好（上一篇的 NN+2opt 在小實例直接命中最優，GA 反而繞遠路）；(2) **隨機種子要記錄**，結果才可重現；(3) **多跑幾次取最好**，單次 GA 的結果是隨機變數，報告平均與變異數才誠實。

7 實務應用

- **排程與時刻表**：學校課表、護理師排班、機組排班——約束又多又雜、無乾淨結構，正是 GA 舒適圈。
- **工程設計最佳化**：NASA ST5 任務的天線形狀由演化演算法設計（著名的「演化天線」），性能勝過人工設計。
- **神經網路**：神經演化（NEAT、超參數搜尋）；強化學習的進化策略（ES）路線。
- **晶片布局**：placement 與 routing 的元啟發式家族成員。
- **近親**：模擬退火（單點 + 溫度）、禁忌搜尋（記憶體防繞圈）、蟻群（費洛蒙）——不同隱喻、同一精神：**有偏好的隨機探索 + 保留好解**。

8 練習題

1. 手算一次 OX：$A = [0, 4, 1, 3, 2]$、$B = [0, 1, 2, 3, 4]$、抄 A 的位置 2–3，寫出子代。
2. 把錦標賽 k 從 4 改成 2 與 12，觀察收斂速度與早熟的取捨（k 越大選擇壓力越強）。
3. 實作**輪盤選擇**（依適應度比例抽樣），與錦標賽比較——注意 TSP 是最小化問題，適應度要先轉換。
4. 給背包 GA 換成**罰分**策略（超重扣分），與修復策略比較最終品質與收斂代數。
5. 挑戰：Memetic GA——每個子代出生後先跑幾步 2-opt 再放回族群（「拉馬克演化」），對 $n = 50$ 的 TSP 比較純 GA 的差距。

9 小結：NP 難題工具箱的全景

工具	保證	什麼時候拿出來
精確（DP/分支限界）	最優	n 小或參數 k 小（FPT）
近似演算法	可證明的倍率	有已知近似（VC、SetCover、排程…）
局部搜尋	局部最優	快、簡單、常常夠用
GA 等元啟發式	無	結構混沌、約束雜、其他方法失靈

本系列 13 篇至此收官：從 3-SAT 的難度源頭，經過回溯、DP、位元壓縮、FPT、近似、貪心，到今天的演化——這就是人類面對 NP-Hard 的完整武器庫。**沒有魔法，只有取捨**：時間換品質、保證換速度、通用換效率。

延伸閱讀：Holland *Adaptation in Natural and Artificial Systems*（1975 開山）；Goldberg *Genetic Algorithms in Search, Optimization and Machine Learning*；Eiben & Smith *Introduction to Evolutionary Computing*；No Free Lunch 定理（Wolpert & Macready 1997——為什麼「萬用」必有代價）。