THUẬT GIẢI DI TRUYỀN – GENETIC ALGORITHM - Kỳ 1

1. TỪ NGẪU NHIÊN ĐẾN THUẬT GIẢI DI TRUYỀN

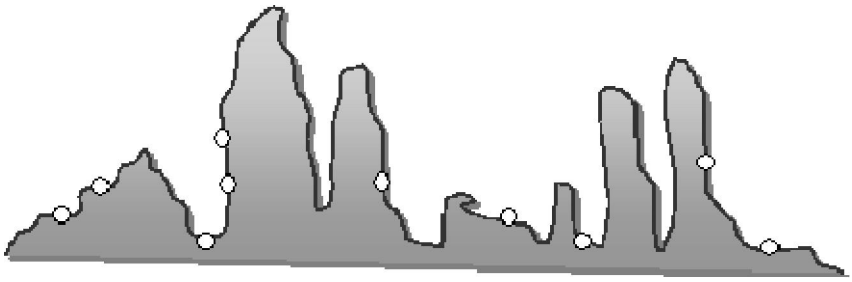
Với các số ngẫu nhiên, chúng ta đã có những lời giải thật độc đáo cho một số vấn đề-bài toán khó nhất định – những vấn đề-bài toán hiện chưa có một lời giải chính xác, tổng quát nào. Tuy nhiên, nếu chỉ dừng lại ở đó, ngẫu nhiên cũng chỉ là may rủi và hoàn toàn chưa đủ sức giải quyết các vấn đề-bài toán phức tạp hơn hoặc có không gian tìm kiếm lớn hơn.

Một ví dụ rất đơn giản là tìm mật mã để mở khóa với mật mã là một con số thập phân có 30 chữ số – giả định rằng ổ khóa này chỉ có thể được mở bằng một mật mã duy nhất. Với bài toán này, không gian tìm kiếm là 10^{30} – nghĩa là sẽ có tổng cộng 10^{30} mật mã khác nhau. Trước vấn đề này, ta thường chỉ nghĩ đến hai phương pháp – vét cạn toàn bộ hoặc thử ngẫu nhiên các mật mã. Ta sẽ phát sinh (ngẫu nhiên hoặc tuần tự theo một quy tắc duyệt nào đó) các mã khóa rồi thử xem mật mã này có thể là mã khóa đúng không. Với phương pháp này, để có được một mật mã với khả năng mở được ổ khóa là trên 50%, ta đã phải phát sinh nhiều hơn $10^{30}/2$ mật mã. Bạn có biết con số này lớn khủng khiếp đến mức nào không? Trên thực tế, nếu dùng siêu máy tính Cray và giả định rằng cứ mỗi *một phần tỷ giây* thì máy này có thể phát sinh và thử nghiệm được *một* mật mã (nghĩa là một giây Gray thử được 1 tỷ mật mã) thì nó phải chạy trong một khoảng thời gian tương đương với "tuổi" của trái đất để thử trên $10^{30}/2$ mật mã !!!.

Dĩ nhiên, khi đứng trước những vấn đề-bài toán như vậy, người ta thường tìm cách cải thiện thuật toán bằng cách cung cấp thêm một số thông tin khác. Chẳng hạn như với bài toán mở khóa trên là thông tin cho biết trong hai mật mã được phát sinh ra, mật mã nào là "tốt" hơn (nghĩa là có khả năng mở khóa cao hơn).

Có thể bạn đọc sẽ thắc mắc "bằng cách nào để biết được giữa hai mật mã, mật mã nào có khả năng mở khóa cao hơn?". Thông thường, khi mở khóa, người ta thường dựa trên các tác nhân vật lý – như tiếng động bên trong ổ khóa khi đưa vào một mật mã – để dự đoán được tính "tốt" của mật mã đang thử.

Khi biết được được độ "tốt" của các mật mã, ta sẽ sử dụng một phương pháp tìm kiếm thông minh hơn - mà người ta thường gọi là tìm kiếm theo kiểu leo đồi (hill-climbing). Với tìm kiếm leo đồi, ta tưởng tượng rằng không gian tìm kiếm của vấn đề-bài toán là một vùng đất gập ghềnh (landscape), có nhiều ngọn đồi cao thấp khác nhau. Trong đó, ngọn đồi *cao nhất* của vùng đất này sẽ là *lời giải tốt nhất* và vị trí có độ cao càng lớn thì càng "gần" với lời giải tốt nhất (độ cao đồng nghĩa với độ tốt của lời giải). Tìm kiếm theo kiểu leo đồi có nghĩa là chúng ta phải phát sinh các lời giải sao cho càng về sau các lời giải càng tiến "gần" tới lời giải tốt nhất hơn. Thao tác này cũng giống như thao tác leo đồi vậy (vì càng ngày ta càng lên cao hơn).



Thuật giải di truyền hoạt động giống leo đồi

Tuy nhiên, kiểu giải quyết này vẫn còn gặp trở ngại cơ bản là, nếu vùng đất của chúng ta có nhiều đồi nhỏ khác bên cạnh ngọn đồi cao nhất thì sẽ có khả năng thuật toán của chúng ta bị "kẹt" ở một ngọn đồi nhỏ. Do tư tưởng là "càng ngày càng lên cao" nên khi lên đến đỉnh một ngọn đồi nhỏ thuật toán sẽ không thể đi tiếp được (vì không thể lên cao được nữa, muốn tìm đến một ngọn đồi cao hơn thì phải xuống đồi hiện tại, mà xuống đồi thì không đúng tư tưởng càng ngày càng lên cao).

Bạn hãy tưởng tượng một máy tính giải quyết vấn đề-bài toán theo kiểu leo đồi là một người leo đồi với tư tưởng "càng leo càng cao". Nếu chỉ có một người leo đồi thì có khả năng người đó sẽ bị "kẹt" trên một đỉnh đồi thấp. Có lẽ bạn đã đoán được tư tưởng tiếp theo! Như vậy, nếu có nhiều người leo đồi cùng leo ở nhiều

địa điểm khác nhau thì khả năng có một người leo đến đỉnh núi cao nhất sẽ cao hơn. Càng nhiều người thì khả năng đến đỉnh núi cao nhất sẽ cao hơn. Nhưng tư tưởng này cũng chưa có gì mới mẻ, đơn giản chỉ là dùng nhiều máy tính để chia việc ra mà thôi. Hơn nữa, với không gian tìm kiếm cỡ 10^{30} như bài toán mở khóa, chúng ta cần phải dùng bao nhiêu siêu máy tính? Mà quan trọng hơn nữa, cho dù có nhiều người leo đồi, nhưng nếu số lượng người leo đồi quá ít so với số lượng đồi thì khả năng tất cả người leo đồi đều bị "kẹt" cũng vẫn còn rất cao.

Đến đây thì rất có thể trong đầu các bạn chợt nảy lên một ý nghĩ. Tại sao không cho nhiều "thế hệ" người leo đồi? Nghĩa là, nếu toàn bộ người leo đồi đầu tiên (giả sử 1000 người chẳng hạn) đều không đạt đến đỉnh đồi cao nhất thì ta sẽ cho 1000 người leo đồi khác tiếp tục leo. Tuy nhiên, sẽ nảy sinh một vấn đề, có khả năng là trong nhóm người leo đồi mới, có những người lại đi leo lại những ngọn đồi mà nhóm trước đã leo rồi. Bạn nghĩ thế nào? Vậy thì hãy ghi nhận lại những ngọn đồi đã leo để những nhóm sau còn thừa hưởng được kết quả của nhóm trước. Hay nói một cách tổng quát hơn : hãy làm sao để những *người giỏi nhất* (leo cao nhất) trong số những người leo đồi đầu tiên truyền lại "kinh nghiệm" leo đồi của mình cho 1000 người thế hệ sau để sao cho 1000 người "hậu duệ" này sẽ leo cao hơn họ. Và nếu 1000 người sau lại thất bại, những người giỏi nhất trong số họ sẽ lại truyền "kinh nghiệm" của mình cho thế hệ 1000 người tiếp nữa để những người thế hệ 3 này leo cao hơn nữa. Tiến trình cứ thế tiếp tục cho đến lúc đến một thế hệ nào đó, có một người leo đến đỉnh đồi cao nhất hoặc hết thời gian cho phép. Trong trường hợp hết thời gian cho phép thì trong toàn bộ các thế hệ, người nào leo cao nhất sẽ được chọn.

Thế đấy, bạn đã hiểu được tư tưởng chính của thuật giải di truyền rồi đó. Rất đơn giản, thay vì chỉ phát sinh một lời giải, ban đầu ta phát sinh một lúc *nhiều* (thậm chí rất nhiều) lời giải cùng lúc. Sau đó, trong số lời giải được tạo ra, chọn ra những lời giải tốt nhất để làm cơ sở phát sinh ra nhóm các lời giải sau với nguyên tắc "càng về sau" càng tốt hơn. Quá trình tiếp diễn cho đến lúc tìm được một lời giải tối ưu.

Đó là tư tưởng sơ khởi ban đầu của thuật giải di truyền. Càng về sau, người ta càng hoàn thiện hơn phương pháp luận của ý tưởng này, dẫn đến sự ra đời của một hệ thống hoàn chỉnh các phương pháp, nguyên lý dùng trong thuật giải di truyền. Người có công đầu trong lĩnh vực này là giáo sư John Holland. Ông đã đưa ra lý thuyết này tiên trong một cuốn sách mang tên "Adaptation in Natural and Artificial Systems", xuất bản năm 1975. Phần đọc thêm của chương sẽ cung cấp cho các bạn đầy đủ thông tin về John Holland.

2. THUẬT GIẢI DI TRUYỀN

Từ phần này trở đi, chúng ta sẽ cùng nhau tìm hiểu thuật giải di truyền – một mô hình khá mới mẻ, có nhiều ứng dụng hấp dẫn và vẫn được tiếp tục nghiên cứu trên khắp thế giới – thuật giải cho phép chúng ta tạo ra được những chương trình máy tính có khả năng tự lập trình cho chính nó.

Thuật giải di truyền (GA) là kỹ thuật chung giúp giải quyết vấn đề-bài toán bằng cách mô phỏng sự tiến hóa của con người hay của sinh vật nói chung (dựa trên thuyết tiến hóa muôn loài của Darwin) trong điều kiện qui định sẵn của môi trường. GA là một thuật giải, nghĩa là mục tiêu của GA không nhằm đưa ra lời giải chính xác tối ưu mà là đưa ra lời giải ***tương đối*** tối ưu.

Trong các tài liệu về GA, người ta thường đề cập đến hai thuật ngữ là "thuật giải di truyền" và "lập trình di truyền". Theo các tài liệu này, "thuật giải di truyền" chỉ sử dụng cấu trúc dữ liệu là chuỗi số nhị phân còn "lập trình di truyền" nghĩa là sử dụng cấu trúc dữ liệu tổng quát. Sở dĩ có cách hiểu như thế vì ý niệm thuật giải di truyền xuất hiện trước và ban đầu người ta chỉ áp dụng nó với cấu trúc dữ liệu là chuỗi nhị phân. Về sau, người ta mới đưa ra cách áp dụng thuật giải này trên các cấu trúc dữ liệu tổng quát hơn nên gọi là lập trình di truyền. Theo chúng tôi, trong tài liệu này, chúng tôi quan niệm rằng, "thuật giải di truyền" là một phương pháp giải quyết vấn đề-bài toán bằng cách mô phỏng quá trình tiến hóa-thích nghi của sinh vật. Còn "lập trình di truyền" là kỹ thuật lập trình sử dụng "thuật giải di truyền" để giải quyết vấn đề-bài toán trên máy tính. Do đó, khi nói đến "thuật giải di truyền" chúng ta chỉ lưu tâm đến khía cạnh thuật giải mà không quan tâm đến việc cài đặt nó ra sao. Ngược lại, khi nói đến "lập trình di truyền" ta quan tâm nhiều hơn đến việc cài đặt.

Theo đề xuất ban đầu của giáo sư John Holland, một vấn đề-bài toán đặt ra sẽ được mã hóa thành các chuỗi bit với chiều dài cố định. Nói một cách chính xác là các thông số của bài toán sẽ được chuyển đổi và biểu diễn lại dưới dạng các chuỗi nhị phân. Các thông số này có thể là các biến của một hàm hoặc hệ số của một biểu thức toán học. Người ta gọi các chuỗi bit này là mã *genome* ứng với mỗi cá thể, các genome

đều có cùng chiều dài. Nói ngắn gọn, một lời giải sẽ được biểu diễn bằng một chuỗi bit, cũng giống như mỗi cá thể đều được quy định bằng gen của cá thể đó vậy. Như vậy, đối với thuật giải di truyền, một cá thể chỉ có một gen duy nhất và một gen cũng chỉ phục vụ cho một cá thể duy nhất. Do đó, gen chính là cá thể và cá thể chính là gen nên ta sẽ dùng lẫn lộn thuật ngữ gen và cá thể từ đây về sau.

Ban đầu, ta sẽ phát sinh một số lượng lớn, giới hạn các cá thể có gen ngẫu nhiên - nghĩa là phát sinh một tập hợp các chuỗi bit ngẫu nhiên. Tập các cá thể này được gọi là quần thể ban đầu (initial population). Sau đó, dựa trên một hàm nào đó, ta sẽ xác định được một giá trị gọi là độ thích nghi - Fitness. Giá trị này, để đơn giản cho bạn đọc lúc đầu, có thể tạm hiểu chính là độ "tốt" của lời giải hay độ cao trong tìm kiếm theo kiểu leo đồi. Vì phát sinh ngẫu nhiên nên độ "tốt" của lời giải hay tính thích nghi của các cá thể trong quần thể ban đầu là không xác định.

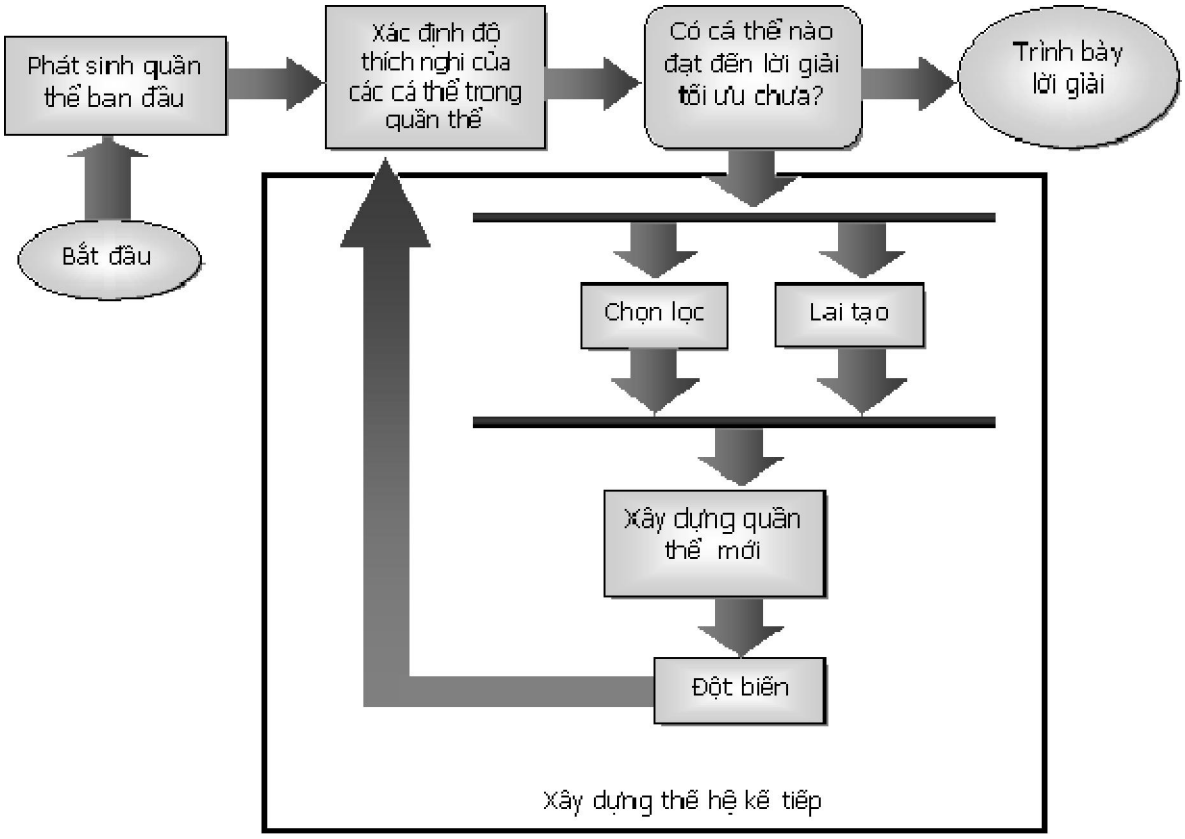
Để cải thiện tính thích nghi của quần thể, người ta tìm cách tạo ra quần thể mới. Có hai thao tác thực hiện trên thế hệ hiện tại để tạo ra một thế hệ khác với độ thích nghi tốt hơn.

Thao tác đầu tiên là sao chép nguyên mẫu một nhóm các cá thể tốt từ thế hệ trước rồi đưa sang thế hệ sau (selection). Thao tác này đảm bảo độ thích nghi của thế hệ sau luôn được giữ ở một mức độ hợp lý. Các cá thể được chọn thông thường là các cá thể có độ thích nghi cao nhất.

Thao tác thứ hai là tạo các cá thể mới bằng cách thực hiện các thao tác *sinh sản* trên một số cá thể được chọn từ thế hệ trước – thông thường cũng là những cá thể có độ thích nghi cao. Có hai loại thao tác sinh sản : một là lai tạo tác lai tạo (crossover), hai là đột biến (mutation). Trong thao tác lai tạo, từ gen của hai cá thể được chọn trong thế hệ trước sẽ được phối hợp với nhau (theo một số quy tắc nào đó) để tạo thành hai gen mới.

Thao tác chọn lọc và lai tạo giúp tạo ra thế hệ sau. Tuy nhiên, nhiều khi do thế hệ khởi tạo ban đầu có đặc tính chưa *phong phú* và chưa phù hợp nên các cá thể không rải đều được hết không gian của bài toán (tương tự như trường hợp leo đồi, các người leo đồi tập trung dồn vào một góc trên vùng đất). Từ đó, khó có thể tìm ra lời giải tối ưu cho bài toán. Thao tác đột biến sẽ giúp giải quyết được vấn đề này. Đó là sự biến đổi ngẫu nhiên một hoặc nhiều thành phần gen của một cá thể ở thế hệ trước tạo ra một cá thể hoàn toàn mới ở thế thệ sau. Nhưng thao tác này chỉ được phép xảy ra với tần suất rất thấp (thường dưới 0.01), vì thao tác này có thể gây xáo trộn và làm mất đi những cá thể đã chọn lọc và lai tạo có tính thích nghi cao, dẫn đến thuật toán không còn hiệu quả.

Thế hệ mới được tạo ra lại được xử lý như thế hệ trước (xác định độ thích nghi và tạo thế hệ mới) cho đến khi có một cá thể đạt được giải pháp mong muốn hoặc đạt đến thời gian giới hạn.



SƠ ĐỒ TỔNG QUÁT CỦA THUẬT GIẢI DI TRUYỀN

THUẬT GIẢI DI TRUYỀN – GENETIC ALGORITHM - Kỳ 2

3. CÁC NGUYÊN LÝ TRONG THUẬT GIẢI DI TRUYỀN

NGUYÊN LÝ VỀ XÁC ĐỊNH CẤU TRÚC DỮ LIỆU

Để có thể giải bài toán bằng thuật giải di truyền, cần "gen hóa" cấu trúc dữ liệu của bài toán

Để có thể thực hiện được các bước trong thuật giải di truyền như đã nêu trong số trước, thao tác quan trọng nhất – không chỉ riêng với vấn đề-bài toán được giải bằng thuật giải di truyền - là phải biết chọn một cấu trúc dữ liệu (CTDL) phù hợp. Để giải vấn đề-bài toán bằng thuật giải di truyền, ta thường chọn sử dụng một trong 3 loại CTDL sau : chuỗi nhị phân, chuỗi số thực và cấu trúc cây. Trong đó, cấu trúc chuỗi nhị phân và chuỗi số thực thường được sử dụng hơn.

Biểu diễn gen bằng chuỗi nhị phân

Về nguyên tắc, mọi cấu trúc dữ liệu trên máy tính, về máy tính, cuối cùng cũng được chuyển về các chuỗi nhị phân (từ số nguyên, số thực, âm thành và thậm chí cả hình ảnh cũng chỉ là các chuỗi nhị phân). Tuy nhiên, quá trình chuyển đổi sang chuỗi nhị phân được thực hiện "ngầm" bởi trình biên dịch của máy tính. Ở

đây, chúng ta sử dụng chuỗi nhị phân một cách tường minh để thể hiện cấu trúc "gen" của một cá thể và để có thể thực hiện các thao tác lai ghép, đột biến trên cấu trúc này.

Thông thường, có rất nhiều cách để chuyển đổi dữ liệu của bài toán về chuỗi nhị phân. Tuy nhiên, bạn cần lưu ý chọn cách biểu diễn hiệu quả nhất theo quy tắc sau.

Quy tắc biểu diễn gen qua chuỗi nhị phân : Chọn chuỗi nhị phân ngắn nhất nhưng đủ thể hiện được tất cả kiểu gen.

Thực chất, đây chính là một quy tắc cũ mà bạn đã biết ở tập 1 khi bàn về chọn kiểu biến. Lấy ví dụ, để chuyển biến X nguyên có giá trị trong khoảng [32,63] về chuỗi nhị phân, có thể bạn sẽ chọn dùng một chuỗi nhị phân dài 6 bit (vì 6 bit đủ để biểu diễn một số trong khoảng [0,63]). Dĩ nhiên , chọn chiều dài 6 bit là đúng nhưng chưa tối ưu. Thực chất, một con số trong khoảng [32,63] chỉ cần 5 bit là đủ. Tại sao vậy? Vì với 5 bit, ta chỉ có thể biểu diễn được một số Y trong khoảng [0,31]? Nhưng ta nhận xét rằng, với mọi số X trong khoảng [32,63] ta đều xác định được một quy tắc để tương ứng với một số Y trong khoảng [0,31]. Cụ thể là Y = X – 32. Và thay vì biểu diễn trực tiếp số X, ta biểu diễn thông qua trung gian Y. Khi đã có Y ta sẽ dễ dàng suy ra X thông qua quy tắc trên.

Để biểu diễn chuỗi nhị phân, ta thường dùng các cách sau : mảng byte, mảng bit biểu diễn bằng mảng byte, mảng bit biểu diễn bằng mảng INTEGER.#

3.1.1 Mảng byte:

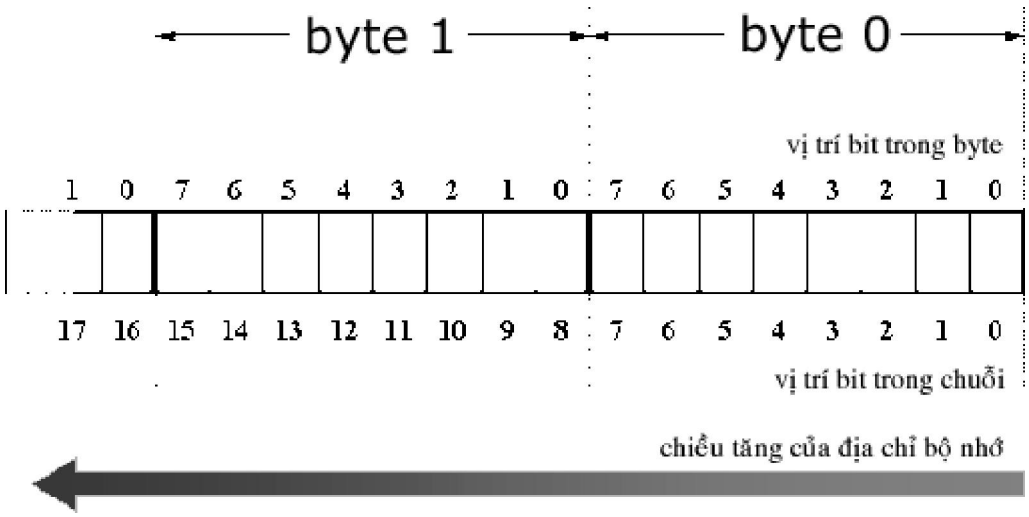
Đối với mảng byte, mỗi byte chính là một bit và chỉ nhận hai giá trị 0 và 1. Nếu byte trong mảng có giá trị khác, chương trình sẽ xem đó là lỗi trầm trọng. Cách biểu diễn này có lợi điểm là truy xuất nhanh hơn phương pháp, nhưng lượng bộ nhớ sẽ tốn gấp 8 lần so với phương pháp không nén (vì một byte gồm 8 bit, nhưng ta chỉ dùng 1 bit nên 7 bit còn lại sẽ bị phí).

P	TYPE TGen = ARRAY[0..N-1] OF BYTE
C	typedef unsigned char CGen[N]

với, N là chiều dài gen của cá thể.

3.1.2 Mảng byte nén:

Đối với kiểu biểu diễn này, một byte trong mảng sẽ biểu diễn cho 8 bit của chuỗi nhị phân. Bit 0 đến bit thứ 7 sẽ nằm trong byte thứ 0, bit 8 đến 15 sẽ nằm trong byte thứ 1, ... và cứ thế. Một cách tổng quát, bit thứ n sẽ nằm trong byte thứ (n DIV 8). Trong đó, (DIV là toán tử chia lấy phần nguyên).



Kiểu biểu diễn này là kiểu biểu diễn ít tốn kém bộ nhớ nhất nhưng ngược lại, thao tác truy xuất lại chậm hơn rất nhiều so với phương pháp mảng byte. Để lấy được một bit thứ n, ta phải thực hiện hai thao tác : đầu tiên là là xác định bit đó nằm ở byte thứ mấy (bằng cách thực hiện phép chia n DIV 8), kế đến là xác định xem bit đó nằm ở vị trí bit thứ mấy trong byte vừa lấy ra (bằng phép chia n MOD 8), bước cuối cùng là thực hiện một vài thao tác trên bit để lấy ra giá trị bit cần tính. Cho dù có tối ưu bằng cách nào đi chăng nữa, việc truy xuất trên mảng bit luôn chậm hơn rất nhiều so với thao tác trên mảng byte. Để thực hiện thật nhanh 3 thao tác này, ta dùng các thao tác trên bit (hai phép toán AND, OR và dịch trái, dịch phải trên bit).

1. Thực hiện phép chia 8 : có thể được biểu diễn bằng một phép dịch phải bit như sau :

P vi_tri_byte = n SHR 3
C vi_tri_byte = n >> 3

2. Thực hiện phép chia 8 lấy phần dư : có thể được biểu diễn bằng một phép AND như sau:

P vi_tri_bit = n AND 7
C vi_tri_bit = n & 7

3. Để lấy một bit bất kỳ ra khỏi mảng byte sau khi đã biết vị trí byte và vị trí bit trong byte, , ta làm thao tác sau:

```
P       TYPE

        TGen=ARRAY[0..N-1] OF BYTE

        ...

        OneGen: TGen;

        ...

        CASE vi_tri_bit OF

        0 : Kq = OneGen[vi_tri_byte] AND 1;

        1 : Kq = OneGen[vi_tri_byte] AND 2;

        2 : Kq = OneGen[vi_tri_byte] AND 4;

        3 : Kq = OneGen[vi_tri_byte] AND 8;

        4 : Kq = OneGen[vi_tri_byte] AND 16;

        5 : Kq = OneGen[vi_tri_byte] AND 32;

        6 : Kq = OneGen[vi_tri_byte] AND 64;
```

```
                7 : Kq = OneGen[vi_tri_byte] AND 128;

                END;
C               typedef unsigned char CGen[N]

                ...

                CGen OneGen;

                ...

                switch (vi_tri_bit)

                {

                case 0 : Kq = OneGen[vi_tri_byte] & 1; break;

                case 1 : Kq = OneGen[vi_tri_byte] & 2; break;

                case 2 : Kq = OneGen[vi_tri_byte] & 4; break;

                case 3 : Kq = OneGen[vi_tri_byte] & 8; break;

                case 4 : Kq = OneGen[vi_tri_byte] & 16; break;

                case 5 : Kq = OneGen[vi_tri_byte] & 32; break;

                case 6 : Kq = OneGen[vi_tri_byte] & 64; break;

                case 7 : Kq = OneGen[vi_tri_byte] & 128; break;

                }
```

4. Để ghi vào một bit bất kỳ trong mảng byte sau khi đã biết vị trí byte và vị trí bit trong byte, ta làm thao tác sau

```
P               TYPE

                TGen=ARRAY[0..N-1] OF BYTE

                ...

                OneGen: TGen;

                ...

                IF gia_tri_moi = 1 THEN BEGIN

                CASE vi_tri_bit OF
```

```
0 : Kq = OneGen[vi_tri_byte] OR (NOT 1);

1 : Kq = OneGen[vi_tri_byte] OR (NOT 2);

2 : Kq = OneGen[vi_tri_byte] OR (NOT 4);

3 : Kq = OneGen[vi_tri_byte] OR (NOT 8);

4 : Kq = OneGen[vi_tri_byte] OR (NOT 16);

5 : Kq = OneGen[vi_tri_byte] OR (NOT 32);

6 : Kq = OneGen[vi_tri_byte] OR (NOT 64);

7 : Kq = OneGen[vi_tri_byte] OR (NOT 128);

END;

END

ELSE

CASE vi_tri_bit OF

0 : Kq = OneGen[vi_tri_byte] AND (NOT 1);

1 : Kq = OneGen[vi_tri_byte] AND (NOT 2);

2 : Kq = OneGen[vi_tri_byte] AND (NOT 4);

3 : Kq = OneGen[vi_tri_byte] AND (NOT 8);

4 : Kq = OneGen[vi_tri_byte] AND (NOT 16);

5 : Kq = OneGen[vi_tri_byte] AND (NOT 32);

6 : Kq = OneGen[vi_tri_byte] AND (NOT 64);

7 : Kq = OneGen[vi_tri_byte] AND (NOT 128);

END;
```

C

```
typedef unsigned char CGen[N]

...

CGen OneGen;

...

if (gia_tri_moi == 1) {
```

```
switch (vi_tri_bit)
{
case 0 : OneGen[vi_tri_byte] | (~1); break;
case 1 : OneGen[vi_tri_byte] | (~2); break;
case 2 : OneGen[vi_tri_byte] | (~4); break;
case 3 : OneGen[vi_tri_byte] | (~8); break;
case 4 : OneGen[vi_tri_byte] | (~16); break;
case 5 : OneGen[vi_tri_byte] | (~32); break;
case 6 : OneGen[vi_tri_byte] | (~64); break;
case 7 : OneGen[vi_tri_byte] | (~128); break;
}
}
else
switch (vi_tri_bit)
{
case 0 : OneGen[vi_tri_byte] & (~1); break;
case 1 : OneGen[vi_tri_byte] & (~2); break;
case 2 : OneGen[vi_tri_byte] & (~4); break;
case 3 : OneGen[vi_tri_byte] & (~8); break;
case 4 : OneGen[vi_tri_byte] & (~16); break;
case5 : OneGen[vi_tri_byte] & (~32); break;
case 6 : OneGen[vi_tri_byte] & (~64); break;
case 7 : OneGen[vi_tri_byte] & (~128); break;
}
```

3.1.3. Mảng INTEGER nén để tối ưu truy xuất

Trong các máy tính ngày nay, thông thường thì đơn vị truy xuất hiệu quả nhất không còn là byte nữa mà là một bội số của byte. Đơn vị truy xuất hiệu quả nhất được gọi là độ dài tư ø(word length). Hiện nay, các máy Pentium đều có độ dài từ là 4 byte. Do đó, nếu ta tổ chức chuỗi nhị phân dưới dạng byte sẽ làm chậm phần nào tốc độ truy xuất. Để hiệu quả hơn nữa, ta sử dụng mảng kiểu INTEGER. Lưu ý kiểu INTEGER có độ dài phụ thuộc vào độ dài từ của máy tính mà trình biên dịch có thể nhận biết được. Chẳng hạn với các version PASCAL,C trên hệ điều hành DOS, kích thước của kiểu INTEGER là 2 byte. Trong khi đó, với các version PASCAL, C trên Windows 9x như Delphi, Visual C++ thì độ dài của kiểu INTEGER là 4 byte. Do đó, để chương trình của chúng ta chạy tốt trên nhiều máy tính khác nhau, bạn nên dùng hàm sizeof(<tên kiểu>). Hàm này sẽ trả ra độ dài của kiểu dữ liệu ta đưa vào.

Khi dùng mảng INTEGER với hàm sizeof, bạn cần thực hiện một số điều chỉnh điều chỉnh nhỏ với thao tác lấy một bit ra và thao tác ghi một bit vào. Thao tác này đơn giản, được xem như một bài tập đối với bạn đọc. Nếu bạn muốn tìm hiểu kỹ tại sao khi dùng mảng INTEGER sẽ nhanh hơn so với mảng byte, hãy xem thêm phần đọc thêm "Tại sao cần phải canh biên bộ nhớ"

3.1.4. Biểu diễn số thực bằng chuỗi nhị phân

Tuy có nhiều chọn lựa nhưng thông thường, để biểu diễn một số thực x, người ta chỉ dùng công thức đơn giản, tổng quát sau :

Giả sử ta muốn biểu diễn số thực x nằm trong khoảng [min, max] bằng một chuỗi nhị phân A dài L bit. Lúc đó, ta sẽ chia miền [min, max] (lượng hóa) thành 2^L-1 vùng. Trong đó, kích thước một vùng là :

$$g = \frac{\max - \min}{2^L - 1}$$

Người ta gọi g là độ chính xác của số thực được biểu diễn bằng cách này (vì g quy định giá trị thập phân nhỏ nhất của số thực mà chuỗi nhị phân dài L bit có thể biểu diễn được).

Giá trị của số thực x được biểu diễn qua chuỗi nhị phân sẽ được tính như sau :

x = min + Decimal(<A>)*g.

trong đó Decimal(<A>) là hàm để tính giá trị thập phân nguyên dương của chuỗi nhị phân A theo quy tắc đếm. Hàm này được tính theo công thức sau:

$$Decimal(<A>) = a_{L-1}.2^{L-1} + \ldots + a_2.2^2 + a_1.2^1 + a_0.2^0$$

Với a_i là bit thứ i trong chuỗi nhị phân tính từ phải sang trái (bit phải nhất là bit 0)

Ví dụ : giá trị hàm Decimal với chuỗi nhị phân vào A = 1 0 1 1 0 1 0 0 là :

$$Decimal\ (A) = 1.2^7 + 0.2^6 + 1.2^5 + 1.2^4 + 0.2^3 + 1.2^2 + 0.2^1 + 0.2^0 = 180$$

(*Ghi chú:* Đoạn mã có chữ P phía trước ứng với ngôn ngữ Pascal, đoạn mã có chữ C phía trước ứng với ngôn ngữ C)

THUẬT GIẢI DI TRUYỀN – GENETIC ALGORITHM - Kỳ 3

3.2. Biểu diễn gen bằng chuỗi số thực

Đối với những vấn đề-bài toán có nhiều tham số, việc biểu diễn gen bằng chuỗi số nhị phân đôi lúc sẽ làm cho kiểu gen của cá thể trở nên quá phức tạp. Dẫn đến việc thi hành cách thao tác trên gen trở nên kém

hiệu quả. Khi đó, người ta sẽ chọn biểu diễn kiểu gen dưới dạng một chuỗi số thực. Tuy nhiên, chọn biểu diễn kiển gen bằng chuỗi số thực, bạn cần lưu ý quy tắc sau :

Quy tắc biểu diễn kiểu gen bằng chuỗi số thực : Biểu diễn kiểu gen bằng số thực phải đảm bảo tiết kiệm không gian đối với từng thành phần gen.

Quy tắc này lưu ý chúng ta phải tiết kiệm về mặt không gian bộ nhớ đối với các từng thành phần gen. Giả sử nghiệm của bài toán được cấu thành từ 3 thành phần, thành phần X thực có giá trị trong khoảng [1.0, 2.0], thành phần Y nguyên trong khoảng [0,15] và thành phần Z trong khoảng [5,8]. Thì chúng ta rất không nên chọn biểu diễn kiểu gen bằng một chuỗi 3 thành phần số thực. Vì như chúng ta đã biết, ít nhất mỗi số thực được phải được biểu diễn bằng 6 byte. Chỉ với 3 số thực, ta đã tốn hết 18 byte. Như vậy với trường hợp cụ thể này, ta nên chọn biểu diễn bằng chuỗi nhị phân, trong đó dùng khoảng 10bit cho thành phần X (độ chính xác khoảng 0.001), 4 bit cho thành phần Y và 2 bit cho thành phần Z. Tổng cộng chỉ chiếm có 16 bit = 2 byte. Chúng ta đã tiết kiệm được rất nhiều bộ nhớ!

Chuỗi số thực được biểu diễn thông qua mảng số thực. Cách thể hiện khá đơn giản :

P	TYPE TGen=ARRAY[0..N-1] OF REAL;
C	typedef float CGen[N];

với N là hằng số quy định kích thước gen.

3.3. Cấu trúc cây

Cấu trúc cây thường được dùng trong trường hợp bản thân CTDL của bài toán cũng có dạng cây. Đây là một trường hợp phức tạp nên hiếm khi được sử dụng. Trong phạm vi cuốn sách này, ta sẽ không khảo sát nhiều về kiểu dữ liệu này mà chỉ đưa ra một ví dụ minh họa cụ thể để bạn đọc có được một số khái niệm ban đầu về CTDL dạng cây. Ngay cả các thao tác trên cây của thuật giải di truyền thường phụ thuộc nhiều vào bài toán đang xét. Do đó, ở đây ta sẽ không trình bày một cách tổng quát.

Một loại cây thường được sử dụng trong thuật giải di truyền là dạng cây hai nhánh (ở đây chúng tôi dùng chữ hai nhánh để phân biệt với loại cây nhị phân – thường dùng trong sắp xếp và tìm kiếm). Để hiểu được cấu trúc cây, bạn cần có một ít kiến thức về con trỏ. Bạn có thể tìm thấy những thông tin này trong các tài liệu về cấu trúc dữ liệu.

NGUYÊN LÝ VỀ XÁC ĐỊNH TÍNH THÍCH NGHI

Tính tốt của một cá thể (lời giải) trong một quần thể chỉ là một cơ sở để xác định tính thích nghi của cá thể (lời giải) đó.

Nguyên lý này ban đầu có vẻ hơi bất ngờ với bạn một khi bạn đã hiểu những ý tưởng chung của thuật giải di truyền mà chúng ta vừa mới tìm hiểu ban nãy. Thật đơn giản, người leo lên ngọn đồi cao nhất trong thế hệ hiện tại chưa chắc đã giúp cho thế hệ sau đến ngọn đồi cao nhất. Cũng vậy, một lời giải tốt nhất ở thế hệ hiện tại vẫn có khả năng bị "kẹt" trong các thế hệ sau cũng như một lời giải chưa tốt ở thế hệ hiện tại vẫn có khả năng tiềm tàng dẫn đến lời giải tối ưu. Tuy vậy, thường thì lời giải tốt ở thế hệ hiện tại sẽ có xác suất dẫn đến lời giải tối ưu cao hơn những lời giải xấu hơn. Do đó, người ta vẫn xem độ tốt của lời giải là một yếu tố căn bản để xác định tính thích nghi của lời giải. Thông thường, độ thích nghi của lời giải cũng chính là xác suất để cá thể đó được chọn lọc hoặc lai ghép khi tiến hành sinh ra thế hệ kế tiếp. Ta sẽ lần lượt tìm hiểu 3 phương pháp để xác định tính thích nghi của một cá thể.

3.4. Độ thích nghi tiêu chuẩn

Hàm mục tiêu là hàm dùng để đánh giá độ tốt của một lời giải hoặc cá thể. Hàm mục tiêu nhận vào một tham số là gen của một cá thể và trả ra một số thực. Tùy theo giá trị của số thực này mà ta biết độ tốt của cá thể đó (chẳng hạn với bài toán tìm cực đại thì giá trị trả ra càng lớn thì cá thể càng tốt, và ngược lại, với bài toán tìm cực tiểu thì giá trị trả ra càng nhỏ thì cá thể càng tốt).

Giả sử trong một thế hệ có N cá thể, cá thể thứ i được ký hiệu là ai. Hàm mục tiêu là hàm G. Vậy độ thích nghi của một cá thể ai tính theo độ thích nghi tiêu chuẩn là

$$F(a_i) = \frac{G(a_i)}{\sum_{j=1}^{N} G(a_j)}$$

Chẳng hạn, xét một thế hệ gồm có 6 cá thể với độ tốt (giá trị càng lớn thì cá thể càng tốt) lần lượt cho trong bảng sau

STT	Độ tốt $G(a_i)$
1	5.3
2	2.1
3	6
4	2.9
5	1.0
6	0.2

Theo công thức trên, tổng tất cả G của 6 phần tử là : 17.5

Như vậy, độ thích nghi của phần tử a_1 :

$F(a_1) = 5.3 / 17.5 » 0.303$

Độ thích nghi của phần tử a_2 :

$F(a_2) = 2.1 / 17.5 = 0.12$

Ta có bảng kết quả cuối cùng như sau :

STT	Độ tốt $G(a_i)$	Độ thích nghi $F(a_i)$
1	5.3	0.303
2	2.1	0.120
3	6	0.343
4	2.9	0.166
5	1.0	0.057
6	0.2	0.011

Nhận xét : độ thích nghi luôn có giá trị biến thiên trong khoảng [0,1]. Hơn nữa, vì độ thích nghi sẽ ứng với khả năng được chọn lọc trong việc sinh ra thế hệ sau nên người ta thường chọn cách tính sao cho độ thích nghi cuối cùng là một xác suất, nghĩa là tổng độ thích nghi của các cá thể phải nhỏ hơn hoặc bằng 1.

3.5. Độ thích nghi xếp hạng (rank method)

Cách tính độ thích nghi tiêu chuẩn như trên chỉ thực sự hiệu quả đối với những quần thể có độ tốt tương đối đồng đều giữa các cá thể. Nếu, vì một lý do nào đó – có thể do chọn hàm mục tiêu không tốt - có một cá thể có độ tốt quá cao, tách biệt hẳn các cá thể còn lại thì các cá thể của thế hệ sau sẽ bị "hút" về phía cá thể đặc biệt đó. Do đó, sẽ làm giảm khả năng di truyền đến thế sau của các cá thể xấu, tạo nên hiện tượng di truyền cục bộ, từ đó có thể làm giảm khả năng dẫn đến lời giải tốt nhất (vì cá thể đặc biệt đó chưa chắc đã dẫn đến lời giải tốt nhất).

Phương pháp xác định độ thích nghi xếp hạng sẽ loại bỏ hiện tượng di truyền cục bộ này. Phương pháp này không làm việc trên giá trị độ lớn của hàm mục tiêu G mà chỉ làm việc dựa trên thứ tự của các cá thể trên quần thể sau khi đã sắp xếp các thể theo giá trị hàm mục tiêu G. Chính vì vậy mà ta gọi là độ thích nghi xếp hạng. Phương pháp này sẽ cho ta linh động đặt một trọng số để xác định sự tập trung của độ thích nghi lên các cá thể có độ tốt cao, mà vẫn luôn đảm bảo được quy luật : cá thể có độ thích nghi càng cao thì xác suất được tồn tại và di truyền càng cao.

Một cách ngắn gọn, ta có độ thích nghi (hay xác suất được chọn) của cá thể thứ i được tính theo công thức sau :

$$F(i) = p*(1-p)^{i-1}$$

với p là một hằng số trong khoảng [0,1].

Công thức trên được xây dựng dựa trên quy tắc được trình bày ngay sau đây và chúng ta sẽ xem phần giải thích quy tắc này như một tư liệu tham khảo.

QUY TẮC

1) Sắp xếp các cá thể của quần thể giảm dần theo thứ tự của giá trị hàm mục tiêu.

2) Chọn một con số p trong khoảng [0,1]. Đây chính là trọng số xác định độ "hút" của các cá thể tốt.

3) Mỗi lượt chọn chỉ chọn một cá thể. Trong một lượt chọn, lần lượt xét các cá thể theo thứ tự đã sắp. Nếu xét đến cá thể thứ i mà cá thể đó được chọn thì lượt chọn kết thúc, ta thực hiện lượt chọn kế tiếp. Ngược lại, nếu cá thể thứ i không được chọn, ta xét đến cá thể thứ i+1. Ta quy ước rằng, khi đã xét đến một cá thể, thì xác suất để chọn cá thể đó (trong thao tác chọn lọc hoặc lai tạo) luôn là p. Rất hiển nhiên, khi đã xét đến một cá thể thì xác suất (XS) để KHÔNG chọn cá thể đó sẽ là 1-p.

Ta ký hiệu a[i] là cá thể thứ i. Từ quy tắc trên, suy ra để a[i] được xét đến thì :

+ a[i-1] đã phải được xét đến

+ nhưng a[i-1] phải KHÔNG được chọn.

Do đó, XS a[i] được xét đến (chứ không phải XS để được chọn!)

= XS a[i-1] được xét * XS a[i-1] KHÔNG được chọn.

= XS a[i-1] được xét * (1-p)

Trong đó, XS a[1] được xét =1 vì cá thể đầu tiên luôn được xét đến.

Bây giờ ta sẽ xây dựng công thức tổng quát để tính XS a[i] được xét đến dựa theo p.

XS a[1] được xét = 1 = $(1-p)^0$

XS a[2] được xét = XS a[1] được xét * (1-p)

= 1*(1-p) = $(1-p)^1$

XS a[3] được xét = XS a[2] được xét * (1-p)

= $(1-p)^1$ * (1-p) = $(1-p)^2$

XS a[4] được xét = XS a[3] được xét * (1-p)

= $(1-p)^2$ * (1-p) = $(1-p)^3$

...

Nói tóm lại :

XS a[i] được xét = XS a[i-1] được xét * (1-p)

= $(1-p)^{i-2}$ * (1-p) = $(1-p)^{i-1}$

Như vậy XS a[i] được chọn = XS a[i] được xét * p = $(1-p)^{i-1}$*p

Để thấy được tính linh động của phương pháp này, bạn hãy quan sát giá trị thích nghi ứng với mỗi giá trị p khác nhau trong bảng sau :

Hạng	$G(a_i)$	Tiêu chuẩn	Xếp hạng (p=0.8)	Xếp hạng (p=0.5)	Xếp hạng (p=0.2)
1	16	0.8	0.8	0.5	0.2
2	2	0.1	0.16	0.25	0.18
3	1	0.05	0.032	0.125	0.128
4	0.5	0.025	0.0064	0.0625	0.1024
5	0.5	0.025	0.00128	0.03125	0.08192

Giá trị p càng nhỏ thì độ giảm của tính thích nghi càng nhỏ. Dựa vào đặc tính này, ta có thể dễ dàng kiểm soát được tính "hút" của các cá thể tốt trong quần thể bằng cách tăng hoặc giảm trị p tương ứng.