BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC MỎ - ĐỊA CHẤT

BÁO CÁO SEMINAR

TÊN ĐỀ TÀI

GIẢI QUYẾT VẤN ĐỀ BẰNG TÌM KIẾM

Người thực hiện: PGS. TS. Lê Văn Hưng (Mã cán bộ: 0801-01)

Đơn vị: Bộ môn Công nghệ phần mềm
Khoa Công nghệ Thông tin

Hà Nội - 2023

Mục lục

1 Mở đầu 2
2 Tại sao phải giải quyết vấn đề bằng tìm kiếm 2
3 Bài toán tìm kiếm trong không gian trạng thái 3
3.1 Phát biểu bài toán tìm kiếm 3
3.2 Thuật toán tìm kiếm tổng quát và cây tìm kiếm 4
2.2 Các tiêu chuẩn đánh giá thuật toán tìm kiếm 7
4 Tìm kiếm không có thông tin (tìm kiếm mù) 7
4.1 Tìm kiếm theo chiều rộng 8
4.2 Tìm kiếm với chi phí cực tiểu 10
4.3 Tìm kiếm theo chiều sâu 10
4.4 Tìm kiếm sâu dần 13
4.5 Tìm theo hai hướng 16
5 Tìm kiếm có thông tin 16
5.1 Tìm kiếm tham lam 17
5.2 Thuật toán A* 18
5.3 Hàm heuristic 20
6 Tìm kiếm cục bộ 23
6.1 Thuật toán leo đồi 24
6.1.1 Di chuyển sang trạng thái tốt nhất 25
6.1.2 Leo đồi ngẫu nhiên 27
6.2 Thuật toán tôi thép 28
6.3 Giải thuật di truyền 29
6.3.1 Nguyên lý chung. 30
6.3.2 Biểu diễn lời giải và không gian tìm kiếm 30
6.3.3 Giải thuật 31
6.3.4 Giá trị xác suất. 32
6.3.5 Kích thước quần thể. 33
6.3.6 Mã hoá lời giải 33
6.3.7 Các phương pháp lai ghép 34
6.3.8 Các phương pháp tạo đột biến 35
6.3.9 Hiệu ứng của các thao tác 35
Tài liệu tham khảo 35

1 Mở đầu

Báo cáo này sẽ trình bày về kỹ thuật giải quyết vấn đề bằng tìm kiếm, còn được gọi là tìm kiếm trong không gian trạng thái. Đây là các phương pháp được dùng phổ biến trong một lớp lớn các bài toán và là lớp kỹ thuật giải quyết vấn đề quan trọng, được nghiên cứu và ứng dụng nhiều của trí tuệ nhân tạo. Trong phạm vi báo cáo, ta sẽ xem xét cách phát biểu một vấn đề dưới dạng bài toán tìm kiếm, trước khi chuyển sang các giải thuật đã được phát triển để giải quyết bài toán này. Các giải thuật tìm kiếm được chia thành ba nhóm lớn: tìm kiếm mù (không có thông tin), tìm kiếm heuristics (có thông tin) và tìm kiếm cục bộ. Giải thuật thuộc nhóm hai và nhóm ba, đặc biệt là nhóm ba, được sử dụng cho các bài toán thực tế với kích thước lớn.

2 Tại sao phải giải quyết vấn đề bằng tìm kiếm

Khi quan sát các bài toán trên thực tế, có thể nhận thấy một lớp lớn bài toán có thể phát biểu và giải quyết dưới dạng tìm kiếm, trong đó yêu cầu có thể là tìm những trạng thái, tính chất thỏa mãn một số điều kiện nào đó, hoặc tìm chuỗi hành động cho phép đạt tới trạng thái mong muốn. Sau đây là một số ví dụ các bài toán như vậy.

- Trò chơi: nhiều trò chơi, ví dụ cờ vua, thực chất là quá trình tìm kiếm nước đi của các bên trong số những nước mà luật chơi cho phép, để giành lấy ưu thế cho bên mình.
- Lập lịch, hay thời khóa biểu: lập lịch là lựa chọn thứ tự, thời gian, tài nguyên (máy móc, địa điểm, con người) thỏa mãn một số tiêu chí nào đó. Như vậy, lập lịch có thể coi như quá trình tìm kiếm trong số tổ hợp phương án sắp xếp phương án đáp ứng yêu cầu đề ra.
- Tìm đường đi: cần tìm đường đi từ điểm xuất phát tới đích, có thể thỏa mãn thêm một số tiêu chí nào đó như tiêu chí tối ưu về độ dài, thời gian, giá thành v.v.
- Lập kế hoạch: là lựa chọn chuỗi hành động cơ sở cho phép đạt mục tiêu đề ra đồng thời thỏa mãn các yêu cầu phụ.

Sự phổ biến của các bài toán có tính chất tìm kiếm dẫn tới yêu cầu phát biểu bài toán tìm kiếm một cách tổng quát, đồng thời xây dựng phương pháp giải bài toán tìm kiếm sao cho hiệu quả, thuận lợi. Các bài toán tìm kiếm mới có thể đưa về dạng bài toán tìm kiếm tổng quát và áp dụng thuật giải đã được xây dựng.

Do tính quan trọng của lớp bài toán này, tìm kiếm đã được nghiên cứu từ rất sớm trong khuôn khổ toán rời rạc và lý thuyết giải thuật. Trong khuôn khổ trí tuệ nhân tạo, tìm kiếm được đặc biệt quan tâm từ khía cạnh xây dựng phương pháp cho phép tìm ra kết quả trong trường hợp không gian tìm kiếm có kích thước lớn khiến cho những phương pháp truyền thống gặp khó khăn. Rất nhiều thuật toán tìm kiếm được phát triển trong khuôn khổ trí tuệ nhân tạo được phát triển dựa trên việc tìm hiểu quá trình giải quyết vấn đề của con người, hay dựa trên sự tương tự với một số quá trình xảy ra trong tự nhiên như quá trình tiến hóa, quá trình hình thành mạng tinh thể, cách thức di chuyển của một số loại côn trùng.

Ngoài việc đứng độc lập như chủ đề nghiên cứu riêng, tìm kiếm còn là cơ sở cho rất nhiều nhánh nghiên cứu khác của trí tuệ nhân tạo như lập kế hoạch, học máy, xử lý ngôn ngữ tự nhiên, suy diễn. Chẳng hạn, bài toán suy diễn có thể phát biểu như quá trình tìm ra

chuỗi các luật suy diễn cho phép biến đổi từ quan sát ban đầu tới đích của quá trình suy diễn. Nhiều phương pháp học máy cũng coi quá trình học như quá trình tìm kiếm mô hình cho phép biểu diễn dữ liệu huấn luyện được dùng trong quá trình học.

3 Bài toán tìm kiếm trong không gian trạng thái

3.1 Phát biểu bài toán tìm kiếm

Một cách tổng quát, một vấn đề có thể giải quyết thông qua tìm kiếm bằng cách xác định tập hợp các phương án, đối tượng, hay trạng thái liên quan, gọi chung là *không gian trạng thái*. Thủ tục tìm kiếm sau đó sẽ khảo sát không gian trạng thái theo một cách nào đó để tìm ra lời giải cho vấn đề. Trong một số trường hợp, thủ tục tìm kiếm sẽ khảo sát và tìm ra chuỗi các hành động cho phép đạt tới trạng thái mong muốn và bản thân chuỗi hành động này là lời giải của bài toán, như trong bài toán tìm đường. Tùy vào cách thức khảo sát không gian trạng thái cụ thể, ta sẽ có những phương pháp tìm kiếm khác nhau.

Để có thể khảo sát không gian trạng thái, thuật toán tìm kiếm bắt đầu từ một trạng thái xuất phát nào đó, sau đó sử dụng những phép biến đổi trạng thái để nhận biết và chuyển sang trạng thái khác. Quá trình tìm kiếm kết thúc khi tìm ra lời giải, tức là khi đạt tới *trạng thái đích*.

Bài toán tìm kiếm cơ bản có thể phát biểu thông qua năm thành phần chính sau:

- Tập các trạng thái Q. Đây chính là không gian trạng thái của bài toán.
- Tập (không rỗng) các trạng thái xuất phát S ($S \subseteq Q$). Thuật toán tìm kiếm sẽ xuất phát từ một trong những trạng thái này để khảo sát không gian tìm kiếm.
- Tập (không rỗng) các trạng thái đích G ($G \subseteq Q$). Trạng thái đích có thể được cho một cách tường minh, tức là chỉ ra cụ thể đó là trạng thái nào, hoặc không tường minh. Trong trường hợp sau, thay vì trạng thái cụ thể, bài toán sẽ quy định một số điều kiện mà trạng thái đích cần thỏa mãn. Ví dụ, khi chơi cờ vua, thay vì chỉ ra vị trí cụ thể của quân cờ, ta chỉ có quy tắc cho biết trạng thái chiếu hết.
- Các *toán tử*, còn gọi là *hành động* hay *chuyển động* hay *hàm chuyển tiếp*. Thực chất đây là một ánh xạ P: $Q \to Q$, cho phép chuyển từ trạng thái hiện thời sang các trạng thái khác. Với mỗi trạng thái n, $P(n)$ là tập các trạng thái được sinh ra khi áp dụng toán tử hay chuyển động P cho trạng thái đó. Các trạng thái này được gọi là trạng thái hàng xóm hay láng giềng của n.
- Giá thành c: $Q \times Q \to R$. Trong một số trường hợp, quá trình tìm kiếm cần quan tâm tới giá thành đường đi. Giá thành để di chuyển từ nút x tới nút hàng xóm y được cho dưới dạng số không âm $c(x, y)$. Giá thành cụ thể được chọn tùy vào từng trường hợp và thể hiện mối quan tâm chính trong bài toán. Ví dụ, với cùng bài toán tìm đường đi, giá thành có thể tính bằng độ dài quãng đường, hay thời gian cần để di chuyển, hay giá thành nhiên liệu tiêu thụ. Giá trị $c(x, y)$ được gọi là *giá thành bước*, từ giá thành bước có thể tính ra giá thành toàn bộ đường đi bằng cách lấy tổng các bước đi đã thực hiện.

Với bài toán phát biểu như trên, *lời giải* là chuỗi chuyển động cho phép di chuyển từ trạng thái xuất phát tới trạng thái đích. Chất lượng lời giải được tính bằng giá thành đường đi, tức là tổng giá thành cần để thực hiện chuỗi chuyển động này, giá thành càng thấp thì lời

giải càng tốt. Cũng có những trường hợp ta không quan tâm tới chuỗi chuyển động mà chỉ quan tâm tới trạng thái đích, ví dụ trong bài toán lập lịch. Trong trường hợp đó, thuật toán tìm kiếm cần trả về lời giải là trạng thái đích, thay về trả về chuỗi chuyển động.

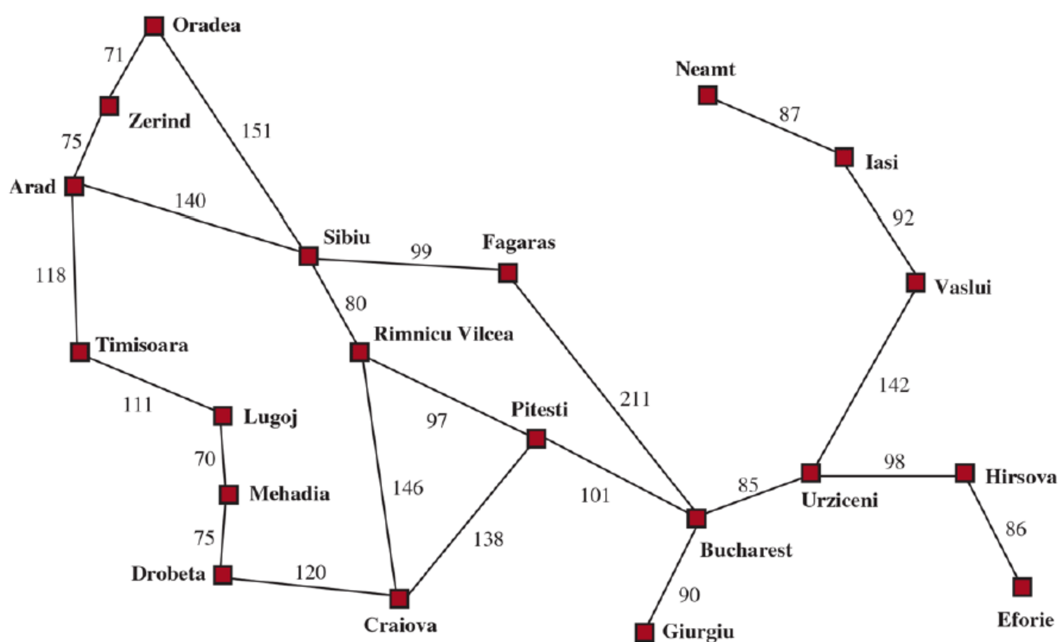
Cần lưu ý rằng, không gian trạng thái có thể cho một cách tường minh, bằng cách liệt kê các trạng thái như trong trường hợp bài toán tìm đường với mỗi địa điểm là một trạng thái. Tuy nhiên, trong nhiều trường hợp, không gian trạng thái được cho một cách không tường minh thông qua trạng thái xuất phát và các chuyển động: khi đó không gian trạng thái là tập hợp tất cả các trạng thái có thể đạt tới từ trạng thái xuất phát bằng cách áp dụng mọi tổ hợp chuyển động. Trong trường hợp này, không gian trạng thái có thể là hữu hạn hoặc vô hạn.

3.2 Thuật toán tìm kiếm tổng quát và cây tìm kiếm

Một cách tổng quát, các thuật toán tìm kiếm dựa trên nguyên lý chung như sau:

Nguyên lý chung: bắt đầu từ trạng thái xuất phát, sử dụng các hàm chuyển động để di chuyển trong không gian trạng thái cho tới khi đạt đến trạng thái mong muốn. Trả về chuỗi chuyển động hoặc trạng thái đích tìm được tùy vào yêu cầu bài toán.

Để minh họa nguyên lý tìm kiếm này, ta xét ví dụ về bài toán tìm đường đi từ Arad đến Bucharest trong Hình 1.



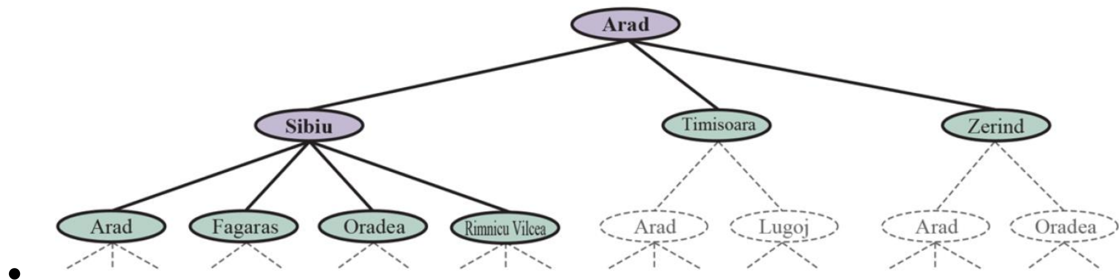
Hình 1. Bài toán tìm đường đi từ Arad đến Bucharest với khoảng cách tính bằng dặm.

Khởi đầu từ trạng thái xuất phát (Arad), thuật toán kiểm tra xem đây có phải đích chưa. Nếu chưa, ta áp dụng các chuyển động để sinh ra các trạng thái láng giềng. Tiếp theo, ta xét một trạng thái vừa được sinh ra, gọi là trạng thái hiện thời bằng cách kiểm tra xem đây đã phải đích chưa. Nếu chưa, ta *mở rộng* trạng thái hiện thời bằng cách áp dụng các chuyển động được phép đối với trạng thái đó để sinh ra các trạng thái khác. Quá trình tìm kiếm như vậy kết thúc khi tìm được trạng thái đích hoặc khi không còn trạng thái nào để mở rộng nữa.

Thuật toán tìm kiếm tổng quát như vậy sinh ra một *cây tìm kiếm*, trong đó mỗi trạng

thái tương ứng với một *nút* trên cây, mỗi nhánh tương ứng với một chuyển động tại nút đang xét. Trạng thái xuất phát tương ứng với gốc cây, những trạng thái được mở rộng tạo thành các nút thế hệ tiếp theo. Hình 2 là ví dụ một phần cây tìm kiếm sinh ra cho bài toán tìm đường đi.

Sau đây là một số thuật ngữ sử dụng khi trình bày về thuật toán tìm kiếm:

- *Mở rộng* nút là áp dụng các chuyển động lên trạng thái tương ứng để sinh ra các nút con.
- *Nút lá* là các nút không có nút con tại thời điểm đang xét.
- Các *nút biên* (còn gọi là nút mở): là tập các nút lá có thể mở rộng tiếp.
- Tập các nút đã được mở rộng được gọi là tập các *nút đóng*, hay đơn giản là *tập đóng*.
- 

Hình 2. Cây tìm kiếm cho bài toán tìm đường đi từ Arad đến Bucharest

Nguyên lý tìm kiếm trên được thể hiện qua thuật toán tìm kiếm tổng quát ở Hình 3.

```
Search(Q, S, G, P) (Q: không gian trạng thái, S: trạng thái bắt đầu, G: đích, P: hành động)

Đầu vào: bài toán tìm kiếm với 4 thành phần như trên

Đầu ra: trạng thái đích

-------------------------------------

Khởi tạo: O ← S (O: tập các nút biên, bước này khởi tạo giá trị ban đầu cho O bằng S)

While(O ≠ Ø) do

    1. Chọn nút n ∈ O và xóa n khỏi O

    2. If n ∈ G, return (đường đi tới n)

    3. Thêm P(n) vào O

Return: Không có lời giải
```

Hình 3. Thuật toán tìm kiếm tổng quát

Thuật toán duy trì tập các nút biên O được khởi tạo bằng tập trạng thái xuất phát. Qua mỗi vòng lặp, thuật toán lấy ra một nút từ tập biên O, kiểm tra xem nút này có phải đích không. Nếu nút được lấy ra là đích, thuật toán trả về kết quả. Trong trường hợp ngược lại, nút này được mở rộng, tức là dùng hàm chuyển động để sinh ra các nút con. Các nút mới sinh ra lại được thêm vào tập O. Thuật toán kết thúc khi tìm thấy trạng thái đích hoặc khi O

rỗng.

Cần lưu ý là trong thuật toán tìm kiếm tổng quát ở trên không quy định rõ nút nào trong số các nút biên (nằm trong tập O) được chọn để mở rộng. Việc lựa chọn nút cụ thể phụ thuộc vào từng phương pháp tìm kiếm và được trình bày trong những phần tiếp theo.

Tránh vòng lặp

Trong cây tìm kiếm trên Hình 2 và thuật toán trên Hình 3, ta đã giả sử rằng mỗi nút đã được duyệt sẽ không được duyệt lại lần nữa, do vậy không cần lưu trữ danh sách những nút đã duyệt. Trên thực tế, trong nhiều trường hợp, việc di chuyển trong không gian trạng thái sẽ dẫn tới những nút đã duyệt qua và tạo thành vòng lặp. Việc chuyển động theo vòng lặp ảnh hưởng không tốt tới thuật toán tìm kiếm. Một số thuật toán tìm kiếm như tìm kiếm theo chiều sâu (trình bày trong một phần sau) không thể tìm ra lời giải nếu rơi vào vòng lặp. Một số thuật toán khác như tìm theo chiều rộng, mặc dù vẫn tìm ra lời giải, nhưng sẽ phải tính toán lâu hơn, xem xét nhiều trạng thái hơn nếu gặp phải vòng lặp.

Để tránh không rơi vào vòng lặp cần có cách kiểm tra để không xem xét lại nút đã duyệt. Cách chung nhất để tránh duyệt lại các nút là duy trì *tập các nút đóng*, tức là các nút đã được mở rộng, và nhớ tất cả các nút đã được mở rộng vào tập đóng này. Nếu một nút mới sinh ra đã có mặt trong tập các nút đóng hoặc tập nút biên thì sẽ không được thêm vào tập các nút biên nữa.

Hình 4 là phiên bản của thuật toán tìm kiếm tổng quát trình bày trong Hình 3, trong đó tập các nút đóng được sử dụng để lưu các nút đã mở rộng và cho phép tránh vòng lặp.

Graph_Search(Q, S, G, P) (Q: tập trạng thái, S: trạng thái bắt đầu, G: đích, P: hành động)

Đầu vào: bài toán tìm kiếm với 4 thành phần như trên

Đầu ra: trạng thái đích

Khởi tạo: $O \leftarrow S$ // *O: tập các nút biên, bước này khởi tạo giá trị bao đầu cho O bằng S*

$D \leftarrow \emptyset$ // D là tập nút đóng, được khởi tạo bằng rỗng

----

While($O \neq \emptyset$) do

1. chọn nút $n \in O$ và xóa n khỏi O
2. If $n \in G$, return (đường đi tới n)
3. **Thêm n vào D**
4. Thêm các nút thuộc $P(n)$ vào O **nếu nút đó không thuộc D và O**

Return: Không có lời giải

Hình 4. Thuật toán Graph_Search với danh sách các nút đóng cho phép tránh vòng lặp.

Phần chữ đậm là các phần được bổ sung so với thuật toán cũ. Thuật toán này được đặt tên là Graph_Search để phân biệt với thuật toán không lưu các nút đóng.

Lưu ý: cần phân biệt khái niệm nút và trạng thái khi triển khai thuật toán cụ thể. Mỗi nút được gắn với một đường đi cụ thể trên cây tìm kiếm còn trạng thái là đặc trưng của thế giới bài toán. Hai nút khác nhau có thể chứa cùng một trạng thái nếu trạng thái đó được sinh ra từ hai đường đi khác nhau. Như vậy, trong nhiều thuật toán, ta chỉ không duyệt lại các nút đã được mở rộng, trong khi vẫn có thể xem xét các trạng thái đã được xem xét trước đó nhưng được sinh ra theo những đường khác trong cây tìm kiếm.

2.2 Các tiêu chuẩn đánh giá thuật toán tìm kiếm

Với bài toán tìm kiếm được phát biểu như trên, nhiều thuật toán tìm kiếm có thể sử dụng để khảo sát không gian và tìm ra lời giải. Để có thể so sánh với nhau, thuật toán tìm kiếm được đánh giá theo bốn tiêu chuẩn sau:

- *Tính đầy đủ:* nếu bài toán có lời giải thì thuật toán có đảm bảo tìm ra lời giải đó không? Nếu có, ta nói rằng thuật toán có tính đầy đủ, trong trường hợp ngược lại ta nói thuật toán không đầy đủ.
- *Tính tối ưu*: nếu bài toán có nhiều lời giải thì thuật toán có cho phép tìm ra lời giải tốt nhất không? Tiêu chuẩn tối ưu thường được dùng là giá thành đường đi. Lời giải tối ưu là lời giải có giá thành đường đi nhỏ nhất. Thuật toán luôn đảm bảo tìm ra lời giải tối ưu được gọi là thuật toán có tính tối ưu.

- *Độ phức tạp tính toán*: được xác định bằng khối lượng tính toán cần thực hiện để tìm ra lời giải. Thông thường, khối lượng tính toán được xác định bằng số lượng nút tối đa cần sinh ra trước khi tìm ra lời giải.
- *Yêu cầu bộ nhớ*: được xác định bằng số lượng nút tối đa cần lưu trữ đồng thời trong bộ nhớ khi thực hiện thuật toán.

Các tiêu chuẩn trên được đánh giá tùy theo độ khó (kích thước) của bài toán. Rõ ràng, bài toán có không gian trạng thái lớn hơn sẽ đòi hỏi tính toán nhiều hơn. Trong trường hợp không gian trạng thái được cho tường minh và hữu hạn, độ khó của bài toán được xác định bằng tổng số nút và số liên kết giữa các nút trong đồ thị tìm kiếm. Tuy nhiên, do không gian trạng thái có thể là vô hạn và được cho một cách không tường minh, độ khó của bài toán được xác định qua ba tham số sau:

- Mức độ rẽ nhánh *b*: là số lượng tối đa nút con có thể sinh ra từ một nút cha.
- Độ sâu *d* của lời giải: là độ sâu của lời giải nông nhất, trong đó độ sâu được tính bằng số nút theo đường đi từ gốc tới lời giải.
- Độ sâu *m* của cây tìm kiếm: là độ sâu lớn nhất của mọi nhánh trên cây tìm kiếm.

4 Tìm kiếm không có thông tin (tìm kiếm mù)

Định nghĩa: Tìm kiếm không có thông tin, còn gọi là tìm kiếm mù (blind, uninformed search) là phương pháp duyệt không gian trạng thái chỉ sử dụng các thông tin theo phát biểu của bài toán tìm kiếm tổng quát trong quá trình tìm kiếm, ngoài ra không sử dụng thêm thông tin nào khác.

Tìm kiếm không có thông tin bao gồm một số thuật toán khác nhau. Điểm khác nhau căn bản của các thuật toán là ở thứ tự mở rộng các nút biên. Sau đây ta sẽ xem xét

các thuật toán tìm theo chiều rộng, tìm theo chiều sâu, tìm kiếm sâu dần và một số biến thể của những thuật toán này.

4.1 Tìm kiếm theo chiều rộng

Thuật toán *tìm kiếm theo chiều rộng* (Breadth-first search, viết tắt là **BFS**) là một dạng tìm kiếm vét cạn.

Nguyên tắc của tìm kiếm theo chiều rộng là trong số những nút biên lựa chọn nút nông nhất (gần nút gốc nhất) để mở rộng. Như vậy, trước hết tất cả các nút có độ sâu bằng 0 (nút gốc) được mở rộng, sau đó tới các nút có độ sâu bằng 1 được mở rộng, rồi tới các nút có độ sâu bằng 2, và tiếp tục như vậy. Ở đây, độ sâu được tính bằng số nút nằm trên đường đi từ nút gốc tới nút đang xét.

Có thể nhận thấy, để thực hiện nguyên tắc tìm kiếm theo chiều rộng, ta cần lựa chọn nút được thêm vào sớm hơn trong danh sách nút biên *O* để mở rộng. Điều này có thể thực hiện dễ dàng bằng cách dùng một hàng đợi FIFO để lưu các nút biên.

Thuật toán tìm theo chiều rộng được thể hiện trên Hình 5.

BFS (Q, S, G, P)

Đầu vào: bài toán tìm kiếm

Đầu ra: trạng thái đích

Khởi tạo: $O \leftarrow S$ // trong thuật toán này, *O* là hàng đợi FIFO

While (*O* không rỗng) do

1. Chọn nút đầu tiên *n* từ *O* và xóa *n* khỏi *O*
2. If $n \in G$, return (đường đi tới n)
3. Thêm $P(n)$ vào cuối *O*

Return: Không có lời giải

Hình 5. Thuật toán tìm kiếm theo chiều rộng

Khác với thuật toán tìm kiếm tổng quát ở trên, tập nút biên *O* được tổ chức dưới dạng hàng đợi FIFO: các nút mới sinh ra được thêm vào cuối của *O* tại bước 3 của mỗi vòng lặp; nút đầu tiên của *O* sẽ được lấy ra để mở rộng như tại bước 1 của vòng lặp của thuật toán trên hình vẽ. Bước 2 của vòng lặp kiểm tra điều kiện đích và trả về kết quả trong trường hợp nút lấy ra từ *O* là nút đích. Thuật toán kết thúc trong hai trường hợp: 1) khi lấy được nút đích từ *O*; và 2) khi tập *O* rỗng. Hai trường hợp này tương ứng với hai lệnh return ở trong và ngoài vòng lặp.

Con trỏ ngược: khi mở rộng một nút ta cần sử dụng *con trỏ ngược* để ghi lại nút cha của nút vừa được mở ra. Con trỏ này được sử dụng để tìm ngược lại đường đi về trạng thái xuất phát khi tìm được trạng thái đích. Khi cài đặt thuật toán, mỗi nút được biểu diễn bằng một cấu trúc dữ liệu có chứa một con trỏ ngược trỏ tới nút cha. Sau khi tìm được nút đích,

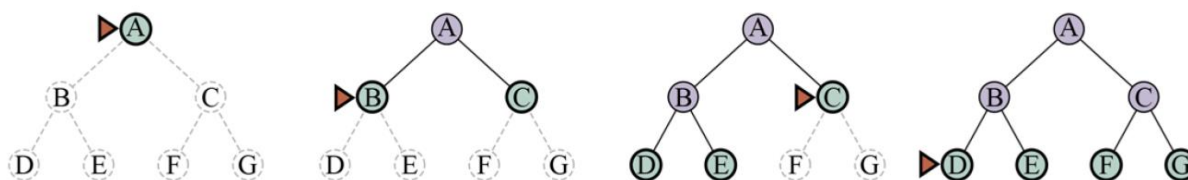
có thể khôi phục đường đi tới nút đó bằng cách lần theo các con trỏ ngược, bắt đầu từ nút đích.

Các cải tiến. Một số cải tiến sau đây có thể sử dụng kết hợp với thuật toán tìm theo chiều rộng vừa trình bày.

Tránh xem xét lại các nút đã mở rộng. Việc xem xét lại các nút đã mở rộng có thể dẫn tới vòng lặp. Mặc dù vòng lặp không ảnh hưởng tới khả năng tìm ra lời giải của tìm kiếm theo chiều rộng, xong việc có vòng lặp và xem xét lại các nút làm tăng độ phức tạp tính toán do phải khảo sát nhiều nút hơn. Vấn đề này có thể giải quyết bằng cách sử dụng tập đóng tương tự trong thuật toán Graph_Search trên Hình 4. Một trạng thái đã có mặt trong tập đóng hoặc tập biên sẽ không được thêm vào tập biên nữa.

Kiểm tra đích trước khi thêm vào tập biên. Trong thuật toán tổng quát, việc kiểm tra điều kiện đích được thực hiện khi nút được lấy ra khỏi *O* để mở rộng. Thay vào đó, ta có thể kiểm tra trước khi thêm nút vào *O*. Ưu điểm của cách làm này là giảm bớt số lượng nút cần lưu trong nút biên cũng như số nút được mở rộng.

Hình 6 mô tả quá trình tìm kiếm theo chiều rộng trên cây nhị phân đơn giản. Tại mỗi bước, nút được mở rộng tiếp theo được đánh dấu bằng hình tam giác.



Hình 6. Tìm kiếm theo chiều rộng trên cây nhị phân đơn giản

Tính chất của tìm theo chiều rộng:

Đối chiếu với các tiêu chuẩn ở trên, tìm kiếm theo chiều rộng có những tính chất sau:

- Thuật toán có tính đầy đủ, tức là nếu bài toán có lời giải, tìm kiếm theo chiều rộng đảm bảo tìm ra lời giải. Thật vậy, nếu lời giải nằm ở độ sâu hữu hạn *d* thì thuật toán sẽ đạt tới lời giải đó sau khi đã khảo sát hết các nút nông hơn, trừ khi yếu tố rẽ nhánh *b* là vô hạn. Tìm theo chiều rộng là tìm kiếm vét cạn, trong đó các nút có độ sâu nhỏ hơn được xem xét trước.
- Tính tối ưu: thuật toán đảm bảo tìm ra lời giải có độ sâu nhỏ nhất. Tuy nhiên, trong trường hợp giá thành đường đi giữa các nút không bằng nhau thì điều này chưa đảm bảo tìm ra đường đi ngắn nhất.
- Độ phức tạp tính toán: với mức độ rẽ nhánh là *b* và độ sâu lời giải *d*, thuật toán sinh ra $O(b^{d+1})$ nút trước khi tìm ra lời giải hay $O(b^d)$ nút nếu kiểm tra đích trước khi thêm nút vào tập biên. Độ phức tạp này là lớn và tăng rất nhanh khi *b* và *d* tăng.

Giả sử rằng, mỗi trạng thái khi được phát triển sẽ sinh ra *b* trạng thái kề. Như vậy từ nút gốc sẽ sinh ra *b* nút với độ sâu 1, các nút này lại sinh ra b^2 nút với độ sâu 2, và tiếp tục như vậy. Giả sử nút đích của bài toán nằm ở độ sâu *d*. Trong trường hợp xấu nhất, nút đích nằm cuối cùng trong số các nút ở độ sâu này và do vậy ta cần mở rộng tất cả nút ở độ sâu d trước khi tìm ra đích, tức là sinh ra b^{d+1} nút ở độ sâu $d+1$. Như vậy, tổng số nút cần mở rộng để tìm ra nút đích là (tính cả nút gốc):

$1 + b + b^2 + ... + b^{d+1} = O(b^{d+1})$

Nếu tiến hành kiểm tra điều kiện đích trước khi thêm vào tập biên như đề cập ở trên, ta sẽ không phải sinh ra các nút ở độ sâu $d + 1$ và do vậy số nút cần sinh ra chỉ còn là $O(b^d)$.

- Yêu cầu bộ nhớ: thuật toán cần lưu $O(b^{d+1})$ nút trong tập biên sau khi đã mở rộng tất cả các nút ở độ sâu d. Nếu sử dụng tập các nút đóng thì tập này cần lưu $O(b^d)$ nút. Như vậy độ phức tạp bộ nhớ của tìm kiếm rộng là $O(b^{d+1})$.

Như vậy, ưu điểm của tìm theo chiều rộng là tính đầy đủ và tối ưu nếu giá thành đường đi như nhau. Nhược điểm của thuật toán này là độ phức tạp tính toán lớn và yêu cầu về bộ nhớ lớn. Trong hai nhược điểm sau, độ phức tạp về bộ nhớ lớn là nghiêm trọng hơn do không thể kiếm được máy tính có bộ nhớ đủ lớn để chạy thuật toán, trong khi ta có thể đợi thêm thời gian để chờ thuật toán chạy xong nếu thời gian chạy không quá lâu. Trên thực thế, thuật toán tìm theo chiều rộng chỉ có thể sử dụng cho các bài toán có kích thước rất nhỏ (b và d không quá 10).

4.2 Tìm kiếm với chi phí cực tiểu

Trong trường hợp giá thành di chuyển giữa hai nút là không bằng nhau giữa các cặp nút, tìm theo chiều rộng không cho tìm ra lời giải có giá thành nhỏ nhất và do vậy không tối ưu. Để tìm ra đường đi ngắn nhất trong trường hợp này cần sử dụng một biến thể của tìm theo chiều rộng có tên gọi là *tìm kiếm với chi phí cực tiểu* (Uniform-Cost-Search).

Phương pháp: Thuật toán tìm với chi phí cực tiểu chọn nút n có giá thành đường đi $g(n)$ nhỏ nhất để mở rộng trước thay vì chọn nút nông nhất như trong tìm theo chiều rộng, trong đó $g(n)$ là giá thành đường đi từ nút xuất phát tới nút n.

Thuật toán: được biến đổi từ tìm kiếm theo chiều rộng bằng cách thay ba bước trong vòng lặp While như sau:

1. Chọn nút n có giá thành $g(n)$ nhỏ nhất thuộc O và xóa n khỏi O
2. If $n \in G$, return (đường đi tới n)
3. Thêm $P(n)$ và giá thành đường đi tới n ($g(n)$) vào O

Cách tránh vòng lặp được thực hiện như trong thuật toán Graph-search, tuy nhiên nếu nút đã có trong tập biên thì ta lưu lại bản có giá thành nhỏ hơn.

Thuật toán tìm kiếm với chi phí cực tiểu có một trường hợp riêng là **thuật toán Dijkstra**. Một trong những điểm khác nhau lớn nhất với thuật toán tìm theo chi phí cực tiểu là thuật toán Dijkstra tìm đường đi ngắn nhất từ nút gốc tới tất cả các nút còn lại thay vì chỉ tìm đường tới nút đích như trong trường hợp tìm theo chi phí cực tiểu.

Thuật toán tìm kiếm này luôn cho lời giải tối ưu, tức là lời giải với đường đi có giá nhỏ nhất. Do thuật toán không lựa chọn nút để mở rộng dựa trên độ sâu mà dựa trên giá thành nên không thể phân tích độ phức tạp tính toán cũng như yêu cầu bộ nhớ của thuật toán dựa trên tham số b và d. Trong trường hợp giá thành mọi chuyển động bằng nhau, tìm kiếm theo giá thống nhất sẽ giống với tìm theo chiều rộng.

4.3 Tìm kiếm theo chiều sâu

Thuật toán tìm kiếm mù được biết đến nhiều tiếp theo là *tìm theo chiều sâu* (Depth-First-Search, viết tắt là **DFS**).

Nguyên tắc của tìm kiếm theo chiều sâu là lựa chọn trong số những nút biên nút sâu nhất (xa nút gốc nhất) để mở rộng trước. Như vậy, thay thì khảo sát tất cả các nhánh của cây tìm kiếm một lúc như trong tìm theo chiều rộng, thuật toán tìm sâu sẽ di chuyển theo một nhánh trong cây tìm kiếm cho tới nút sâu nhất, tức là nút không có nút con, trước khi chuyển sang nhánh tiếp theo.

Để thực hiện nguyên tắc trên, ta cần lựa chọn nút được thêm vào sau cùng trong tập nút biên *O* để mở rộng. Điều này có thể thực hiện dễ dàng bằng cách dùng một ngăn xếp để lưu các nút biên, các nút được thêm vào và lấy ra theo nguyên lý LIFO (vào sau ra trước). Thuật toán tìm theo chiều sâu cũng có thể thực hiện bằng cách đệ quy và quay lui. Tuy nhiên, trong tài liệu này, ta không sẽ không xem xét phương án sử dụng đệ quy và quay lui.

Thuật toán tìm kiếm theo chiều sâu được thể hiện trên Hình 7, trong đó tập biên *O* được triển khai dưới dạng ngăn xếp LIFO. Các nút mới sinh ra được thêm vào đầu ngăn xếp *O* ở bước 3 của vòng lặp chính. Nút để mở rộng cũng được lấy ra từ đầu của *O* ở bước 1 của vòng lặp chính. Tương tự tìm theo chiều rộng, mỗi nút cần có con trỏ ngược về nút cha. Con trỏ ngược được sử dụng để khôi phục lại đường đi khi thuật toán đã tìm ra nút đích. Thuật toán kết thúc và trả về kết quả tại bước 2 của vòng lặp nếu nút lấy ra khỏi O là nút đích; hoặc thuật toán kết thúc (bằng lệnh return ở dưới cùng) mà không tìm được kết quả nếu tập O rỗng.

DFS(Q, S, G, P)

Đầu vào: bài toán tìm kiếm

Đầu ra: (đường đi tới) trạng thái đíchC

Khởi tạo: O←S // O được tổ chức như ngăn xếp LIFO

While(O ≠ Ø) do

1. Chọn nút n đầu tiên của O và xóa n khỏi O
2. If n ∈ G, return (đường đi tới n)
3. Thêm P(n) vào đầu O

Return: Không có lời giải

Hình 7. Thuật toán tìm kiếm theo chiều sâu

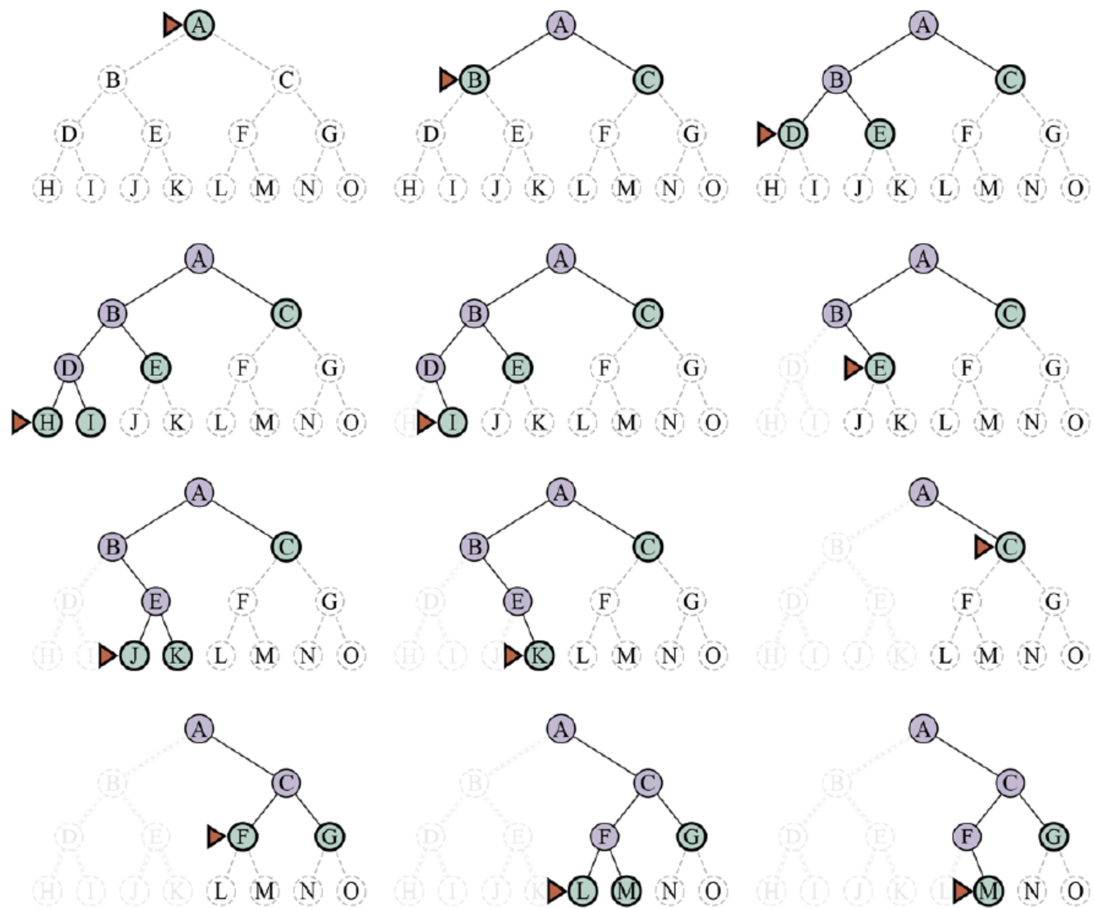
Thuật toán trên có thể dẫn tới vòng lặp. Ví dụ, trong bài toán tìm đường trên bản đồ, thuật toán có thể quay về vị trí trước đó và lặp đi lặp lại chuyển động giữa hai trạng thái liền nhau. Khác với tìm theo chiều rộng, tìm theo chiều sâu sẽ lặp vô hạn mà không tìm được lời giải

Để tránh vòng lặp khi tìm kiếm theo chiều sâu, có thể sử dụng một trong hai phương pháp. *Phương pháp thứ nhất* giống như phương pháp sử dụng trong thuật toán Graph_Search, tức là duy trì danh sách nút đóng và nhớ tất cả các nút đã mở rộng vào đây. Một nút mới

sinh ra chỉ được thêm vào nút biên nếu nó chưa xuất hiện trong tập đóng và tập biên. Phương pháp này đòi hỏi nhiều bộ nhớ để duy trì tập nút đóng.

Phương pháp thứ hai kiểm tra xem nút mới có thuộc đường đi từ gốc tới nút hiện thời không, nếu có, nút mới sẽ không được thêm vào tập nút biên. Cách này không đòi hỏi bộ nhớ để lưu tập đóng, tuy nhiên chỉ cho phép tránh vòng lặp vô hạn và không cho phép giảm khối lượng tính toán trong trường hợp nhiều nhánh của đồ thị tìm kiếm có những phần trùng nhau.

Hình 8 mô tả quá trình tìm kiếm theo chiều sâu trên cây nhị phân từ trạng thái bắt đầu A đến đích M (từ trái sang phải, từ trên xuống dưới). Tại mỗi bước, nút được đánh dấu bằng một hình tam giác sẽ được mở rộng tiếp theo.



Hình 8. Tìm kiếm theo chiều sâu trên cây nhị phân đơn giản

Tính chất thuật toán tìm theo chiều sâu:

- Nếu không gian trạng thái là hữu hạn thì thuật toán là đầy đủ. Ngược lại, nếu không gian trạng thái là vô hạn thì thuật toán là không đầy đủ do có thể di chuyển theo một đường đi không chứa nút đích và có độ sâu vô hạn (cứ đi theo nhánh không đúng mãi mà không chuyển sang nhánh khác được).
- Thuật toán không tối ưu: thuật toán có thể mở rộng những nhánh dẫn tới lời giải không tối ưu trước, đặc biệt trong trường hợp có nhiều lời giải.
- Độ phức tạp của thuật toán ở trường hợp xấu nhất là O(b^m) với m là độ sâu tối đa của cây tìm kiếm. Trong trường hợp không gian trạng thái là vô hạn thì *m* là vô hạn.

Trên thực tế DFS tìm ra lời giải nhanh hơn BFS, đặc biệt nếu tồn tại nhiều lời giải.

- Bộ nhớ cần nhớ tối đa *b**m* (mỗi mức chỉ nhớ *b* nút, với tối đa *m* mức), như vậy độ phức tạp về bộ nhớ của thuật toán này chỉ là O(*bm*). Để đánh giá độ phức tạp không gian của tìm kiếm theo độ sâu ta có nhận xét rằng, khi ta phát triển một đỉnh u trên cây tìm kiếm theo độ sâu, ta chỉ cần lưu các đỉnh chưa được phát triển mà chúng là các đỉnh con của các đỉnh nằm trên đường đi từ gốc tới đỉnh u. Như vậy đối với cây tìm kiếm có nhân tố nhánh *b* và độ sâu lớn nhất là *m*, ta chỉ cần lưu ít hơn *b**m* đỉnh. Ưu cầu bộ nhớ so với tìm theo chiều rộng là ưu điểm nổi bật nhất của tìm theo chiều sâu.

4.4 Tìm kiếm sâu dần

Tìm kiếm sâu dần (Iterative Deepening Search, viết tắt là IDS) là một phương pháp tìm kiếm dựa trên tìm theo chiều sâu nhưng có tính đầy đủ và cho phép tìm ra lời giải tối ưu.

Như đã nói ở trên, mặc dù có ưu điểm rất lớn là không yêu cầu nhiều bộ nhớ như tìm theo chiều rộng, tìm theo chiều sâu có thể rất chậm hoặc bế tắc nếu mở rộng những nhánh sâu (vô tận) không chứa lời giải. Để khắc phục, có thể sử dụng kỹ thuật *tìm kiếm với độ sâu hữu hạn*: tìm kiếm theo chiều sâu nhưng không tiếp tục phát triển một nhánh khi đã đạt tới một độ sâu nào đó, thay vào đó, thuật toán chuyển sang phát triển nhánh khác. Nói cách khác, các nút ở độ sâu giới hạn sẽ không được mở rộng tiếp. Thuật toán này có yêu cầu bộ nhớ nhỏ tương tự tìm theo chiều sâu, trong khi chắc chắn tìm được lời giải.

Kỹ thuật này có thể sử dụng trong trường hợp có thể dự đoán được độ sâu của lời giải bằng cách dựa trên đặc điểm bài toán cụ thể. Chẳng hạn, nếu ta biết rằng ở miền Bắc và Bắc Trung bộ không có nhiều hơn 15 thành phố thì khi tìm đường từ Hà Nội vào Vinh có thể giới hạn chiều sâu tìm kiếm bằng 15. Một số bài toán khác cũng có thể dự đoán trước giới hạn độ sâu như vậy. Trong trường hợp ta biết chính xác độ sâu của lời giải, thuật toán sẽ cho lời giải tối ưu (lời giải nông nhất).

Tuy nhiên, trong trường hợp chung, ta thường không có trước thông tin về độ sâu của lời giải. Trong trường hợp như vậy có thể sử dụng phương pháp tìm kiếm sâu dần. Thực chất tìm kiếm sâu dần là tìm kiếm với độ sâu hữu hạn, trong đó giới hạn độ sâu được khởi đầu bằng một giá trị nhỏ, sau đó tăng dần cho tới khi tìm được lời giải.

Phương pháp: Tìm theo DFS nhưng không bao giờ mở rộng các nút có độ sâu quá một giới hạn nào đó. Giới hạn độ sâu được bắt đầu từ 0, sau đó tăng lên 1, 2, 3 v.v. cho đến khi tìm được lời giải.

Thuật toán tìm kiếm sâu dần thể hiện trên Hình 9, trong đó tìm kiếm sâu được lặp lại, tại mỗi bước lặp, độ sâu được giới hạn bởi biến C. Sau mỗi vòng lặp, giá trị của C được tăng thêm một đơn vị và thuật toán xây dựng lại cây tìm kiếm từ đầu. Với mỗi giá trị của C, thuật toán tiến hành tìm kiếm theo chiều sâu nhưng không thêm vào ngăn xếp *O* các nút có độ sâu lớn hơn *C*. Việc kiểm tra này được thực hiện ở bước c) tại vòng lặp trong của thuật toán. Lưu ý rằng trong trường hợp này khó xác định điều kiện kết thúc của thuật toán trong trường hợp không tìm được lời giải.

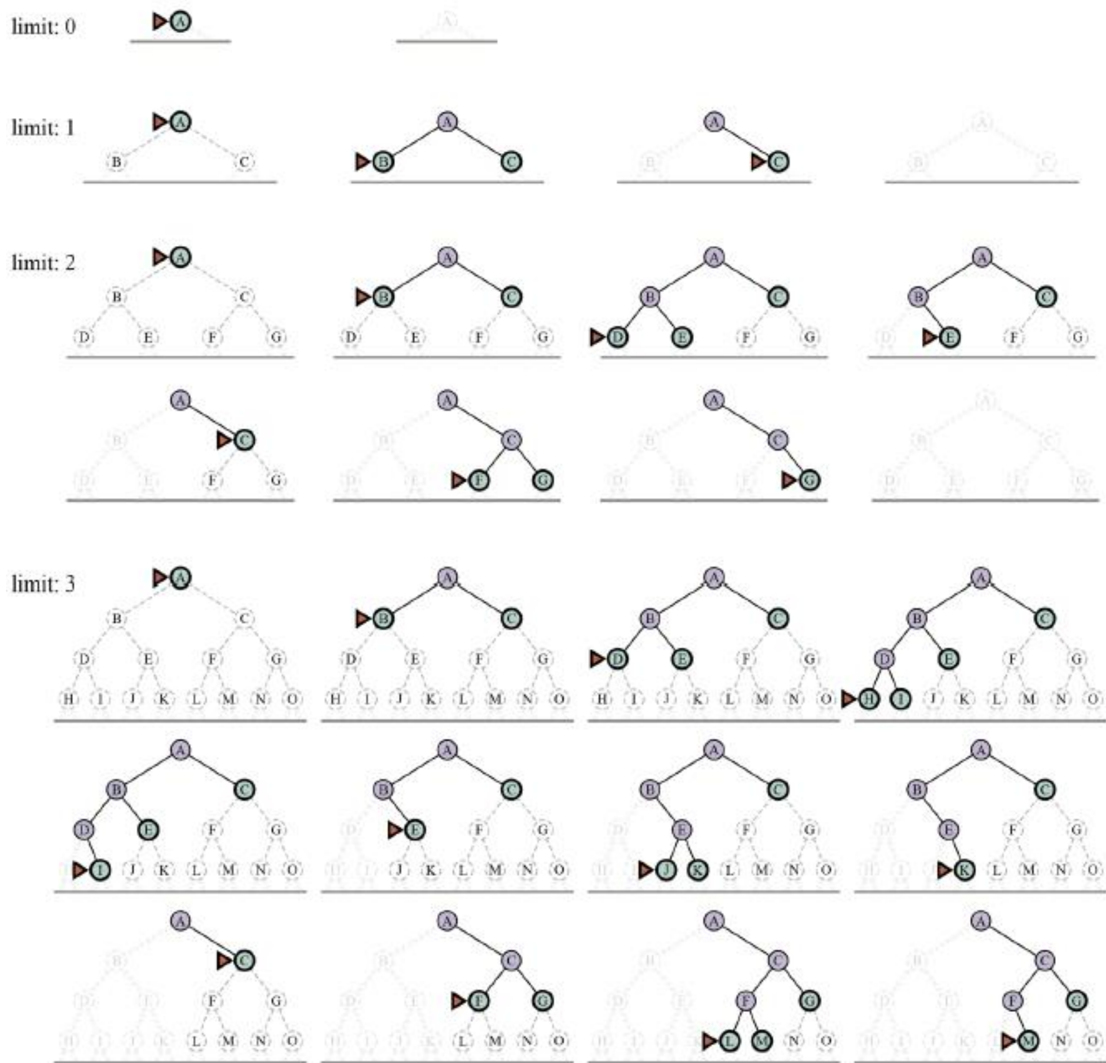
```
IDS(Q, S, G, P)

Đầu vào: thuật toán tìm kiếm

Đầu ra: trạng thái đích

Khởi tạo:   O←S (O: ngăn xếp LIFO như trong DFS)

            C ← 0 (C là giới hạn độ sâu tìm kiếm)

--------------------------------

While (điều kiện kết thúc chưa thoả mãn) do

    While(O ≠ Ø) do

        //Thực hiện 3 bước tương tự tìm kiếm sâu

        a)   Lấy nút đầu tiên n từ đầu của O

        b)   If (n ∈ G) then return (đường đi tới n)

        c)   If (độ sâu của n nhỏ hơn hoặc bằng C) then

                  Thêm P(n) vào đầu O

    C++, O←S

Return: không tìm được lời
```

Hình 9. Thuật toán tìm kiếm sâu dần

Hình 10 biểu diễn quá trình tìm kiếm sâu dần từ trạng thái khởi đầu A tới đích M trên cây nhị phân đơn giản, với giới hạn độ sâu giới hạn thay đổi từ 0 đến 3. Hình tam giác đánh dấu nút để mở rộng tiếp theo.

Tính chất của tìm sâu dần:

- Thuật toán đầy đủ, tức là đảm bảo tìm ra lời giải nếu có. Thật vậy, tương tự tìm theo chiều rộng, tìm sâu dần xem xét toàn bộ các nút ở một độ sâu, bắt đầu từ độ sâu 0, trước khi chuyển sang xem xét toàn bộ nút ở độ sâu tiếp theo. Do vậy, thuật toán sẽ tìm được lời giải khi xem xét hết các nút ở độ sâu *d* (*d* là độ sâu lời giải).
- Tương tự tìm theo chiều rộng, *thuật toán tối ưu nếu giá thành đường đi giữa hai nút giống nhau*, tức là nếu có nhiều lời giải, có thể tìm ra được lời giải nông nhất. Thật vậy, thuật toán sẽ xem xét toàn bộ các nút có độ sâu bằng 0, trước khi xem xét các nút có độ sâu bằng 1 v.v. Do vậy, trong các lời giải có độ sâu khác nhau, thuật toán sẽ xem xét lời giải nông hơn trước.



Hình 10. Cây tìm kiếm theo thuật toán tìm kiếm sâu dần trên cây nhị phân đơn giản

- Yêu cầu bộ nhớ nhỏ. Cụ thể, yêu cầu bộ nhớ là O(bd). Thực chất, tại mỗi bước lặp, thuật toán thực hiện tìm kiếm theo chiều sâu nên yêu cầu bộ nhớ tương tự tìm theo chiều sâu, tức là O(bd).
- Độ phức tạp tính toán O(b^d).Trong tìm kiếm sâu lặp, ta phải phát triển lặp lại nhiều lần cùng một trạng thái. Điều đó làm cho ta có cảm giác rằng, tìm kiếm sâu lặp lãng phí nhiều thời gian. Thực ra thời gian tiêu tốn cho phát triển lặp lại các trạng thái là không đáng kể so với thời gian tìm kiếm theo bề rộng. Thật vậy, mỗi lần gọi thủ tục tìm kiếm sâu hạn chế tới mức d, nếu cây tìm kiếm có nhân tố nhánh là b, thì số đỉnh cần phát triển là:

$$1 + b + b^2 + ... + b^d$$

Nếu lời giải ở độ sâu d, thì trong tìm kiếm sâu lặp, ta phải gọi thủ tục tìm kiếm sâu hạn chế với độ sâu lần lượt là 0, 1, 2, ..., d. Do đó các đỉnh ở mức 1 phải phát triển lặp d lần, các đỉnh ở mức 2 lặp d-1 lần, ..., các đỉnh ở mức d lặp 1 lần. Như vậy tổng số đỉnh cần phát triển trong tìm kiếm sâu lặp là:

$$(d + 1) + db + (d - 1)b^2 + ... + 2b^{d-1} + b^d$$

Do đó thời gian tìm kiếm sâu dần là O(b^d) tương tự tìm kiếm theo chiều rộng. Trên

thực tế, do phải xem xét nhiều lần các nút ở độ sâu nhỏ, nên độ phức tạp tính toán của tìm sâu dần lớn hơn tìm kiếm theo chiều rộng nhưng không lớn hơn quá nhiều.

4.5 Tìm theo hai hướng

Trong các phương pháp tìm kiếm ở trên, quá trình tìm kiếm bắt đầu từ nút xuất phát và kết thúc khi đạt tới nút đích. Do tính chất đối xứng của đường đi, quá trình tìm kiếm cũng có thể bắt đầu từ nút đích và tìm tới nút xuất phát. Ngoài ra, quá trình tìm kiếm có thể xuất phát đồng thời từ cả nút xuất phát và nút đích, xây dựng đồng thời hai cây tìm kiếm. Quá trình tìm kiếm kết thúc khi hai cây tìm kiếm có một nút chung.

Tìm theo hai hướng (Bidirectional Search) là phương pháp tìm kiếm trong đó ta đồng thời xây dựng hai cây tìm kiếm có nút gốc là trạng thái xuất phát và trạng thái đích.

Phương pháp. Tìm kiếm bắt nguồn từ nút xuất phát và nút đích. Như vậy, sẽ tồn tại hai cây tìm kiếm, một cây có gốc là nút xuất phát và một cây có gốc là nút đích. Tìm kiếm kết thúc khi có nút lá của cây này trùng với nút lá của cây kia.

Chú ý:

- Khi tìm theo hai hướng cần sử dụng tìm theo chiều rộng. Việc tìm theo chiều sâu có thể không cho phép tìm ra lời giải nếu hai cây tìm kiếm phát triển theo hai nhánh không gặp nhau.
- Để tìm theo hai hướng cần viết thêm một hàm chuyển động ngược là D(x): tập các nút cha của x. Nút càng gần nút xuất phát càng được coi là tổ tiên

Tính chất của tìm theo hai hướng:

- Việc kiểm tra xem nút lá này có trùng với nút lá kia đòi hỏi tương đối nhiều thời gian. Thật vậy, ở độ sâu d, mỗi cây tìm kiếm sẽ sinh ra b^d nút lá. Như vậy cần so sánh mỗi nút lá của cây theo hướng này với b^d nút lá của hướng ngược lại.
- Độ phức tạp tính toán: do gặp nhau ở giữa nên chiều sâu mỗi cây là d/2. Theo tính toán đối với tìm theo chiều rộng, độ phức tạp tính toán khi đó là O($b^{d/2}$). Như vậy mặc dù việc kiểm tra các nút trùng nhau gây tốn thời gian nhưng số lượng nút cần mở rộng của cả hai cây giảm đáng kể so với tìm theo một chiều.

5 Tìm kiếm có thông tin

Đối với tìm kiếm mù, các nút biên lần lượt được mở rộng theo một thứ tự nhất định mà không tính tới việc ưu tiên các nút có khả năng dẫn tới lời giải nhanh hơn. Kết quả của việc tìm kiếm như vậy là việc di chuyển trong không gian tìm kiếm không có định hướng, dẫn tới phải xem xét nhiều trạng thái. Đối với những bài toán thực tế có không gian trạng thái lớn, tìm kiếm mù thường không thực tế do có độ phức tạp tính toán và yêu cầu bộ nhớ lớn.

Để giải quyết vấn đề trên, *chiến lược tìm kiếm có thông tin (informed search) hay còn được gọi là tìm kiếm heuristic sử dụng thêm thông tin từ bài toán để định hướng tìm kiếm, cụ thể là lựa chọn thứ tự mở rộng nút theo hướng mau dẫn tới đích hơn*.

Nguyên tắc chung của tìm kiếm có thông tin là sử dụng một hàm $f(n)$ để đánh giá độ "tốt" tiềm năng của nút n, từ đó chọn nút n có hàm f tốt nhất để mở rộng trước. Thông thường, độ tốt được đo bằng giá thành đường đi tới đích, do vậy nút có hàm $f(n)$ nhỏ được

ưu tiên mở rộng trước.

Trên thực tế, việc xây dựng hàm $f(n)$ phản ánh chính xác độ tốt của nút thường không thực hiện được, thay vào đó ta chỉ có thể ước lượng hàm $f(n)$ dựa vào thông tin có được từ bài toán. Như sẽ thấy trong các phần sau, hàm $f(n)$ thường chứa một thành phần là hàm heuristic $h(n)$, là hàm ước lượng khoảng cách từ nút n tới đích.

Trong phần này ta sẽ xem xét hai chiến lược tìm kiếm có thông tin, đó là *tìm kiếm tham lam* và *tìm kiếm A**.

5.1 Tìm kiếm tham lam

Ý tưởng của *tìm kiếm tham lam* (Greedy Search) là chọn trong tập nút biên nút có khoảng cách tới đích nhỏ nhất để mở rộng. Lý do làm như vậy là việc mở rộng nút gần đích có xu hướng dẫn tới lời giải nhanh hơn.

Trong phương pháp này, để đánh giá độ tốt của một nút, ta sử dụng hàm đo giá thành đường đi từ nút đó tới đích. Tuy nhiên, do không biết được chính xác giá thành đường đi từ một nút tới đích, ta chỉ có thể ước lượng giá trị này. Hàm ước lượng độ tốt, hay giá thành đường đi từ một nút n tới địch gọi là *hàm heuristic* và ký hiệu $h(n)$. Như vậy, đối với thuật toán tham lam, ta có $f(n) = h(n)$.

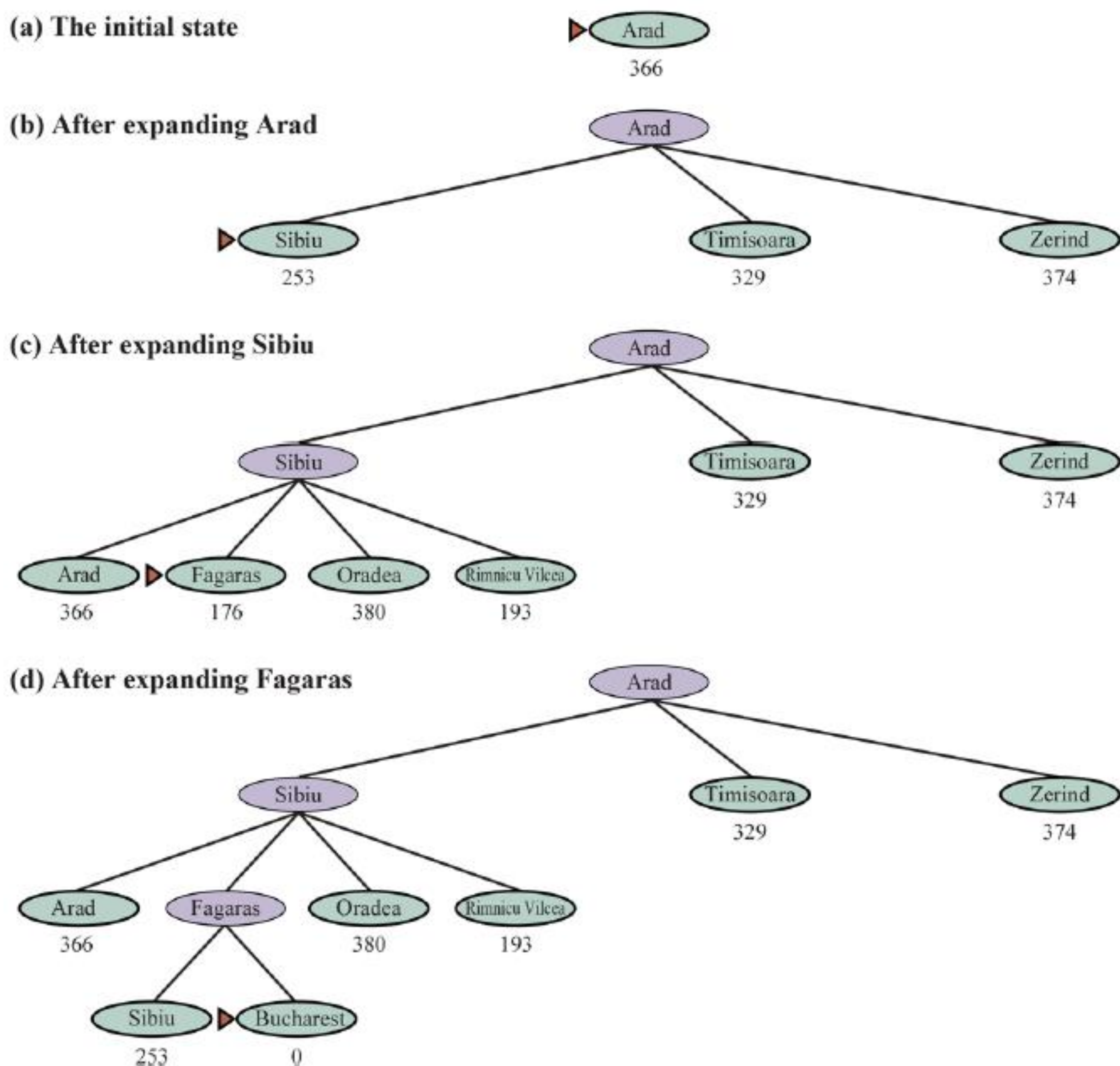
Do hàm $h(n)$ chỉ là hàm ước lượng giá thành đường đi tới đích nên có thể nói rằng tìm kiếm tham lam mở rộng nút trông có vẻ gần đích nhất trước các nút khác. Thuật toán được gọi là "tham lam" do tại mỗi bước, thuật toán cố gắng tiến về đích nhiều nhất có thể. Như ta sẽ thấy dưới đây, do thuật toán chỉ cố gắng làm tốt nhất tại mỗi bước mà không tính tới tổ hợp các bước, đường đi tìm được có thể không phải là đường đi ngắn nhất.

Hàm heuristic được xây dựng dựa trên thông tin có được về bài toán. Hàm này phải thỏa mãn hai điều kiện: thứ nhất, đây là hàm không âm ($h(n) \geq 0$), và thứ hai, nếu n là nút đích thì $h(n) = 0$.

Như vậy, *tìm kiếm tham lam sử dụng hàm heuristic h(n) để ước lượng khoảng cách từ các nút tới đích và thuật toán luôn mở rộng nút n có hàm h(n) nhỏ nhất trong số các nút biên.*

Ví dụ. Khi tìm đường đi giữa hai nút bản đồ, hàm heuristic cho một nút có thể tính bằng khoảng cách theo đường chim bay giữa thành phố đó với nút đích cần đến. Để minh họa, ta xét bài toán tìm đường đi từ nút Arad tới nút Bucharest trên đồ thị ở Hình 2. Trên đồ thị này, khoảng cách thực giữa hai nút bất kỳ được cho dưới dạng số bên cạnh cung nối hai nút.

Hình 11 minh họa hoạt động của thuật toán tìm kiếm tham lam tìm đường đi trên đồ thị từ Arad tới nút Bucharest. Khoảng cách tính theo đường chim bay từ mỗi nút tới nút đích Bucharest được cho bằng số bên cạnh các nút.



Hình 11. Minh họa hoạt động của thuật toán tìm kiếm tham lam

Đặc điểm của tìm kiếm tham lam:

- Không có tính đầy đủ do có khả năng tạo thành vòng lặp vô hạn ở một số nút.
- Độ phức tạp về thời gian: trong trường hợp xấu nhất là O(b^m). Độ phức tạp này có thể giảm rất nhiều nếu tồn tại heuristic tốt. Tuy nhiên trong trường hợp heuristic không tốt thì thuật toán sẽ đi sai hướng và do vậy gần giống với tìm sâu.
- Độ phức tạp về không gian lưu trữ: trong trường hợp xấu nhất thuật toán lưu O(b^m) nút với m là độ sâu cây tìm kiếm, tuy nhiên nếu heuristic tốt thì số nút cần lưu giảm đi rất nhiều như trong ví dụ trên.
- Thuật toán không tối ưu: trong ví dụ ở trên, đường đi Arad→Sibiu→Faragas→ Bucharest tìm được có độ dài dài hơn đường đi ngắn nhất Arad→Sibiu →Rimnicu→Pitesti→ Bucharest 32 dặm.

5.2 Thuật toán A*

Một trong những nhược điểm của tìm kiếm tham lam là không cho lời giải tối ưu, tức là lời giải với đường đi ngắn nhất. Lý do tìm kiếm tham lam không đảm bảo tìm ra đường đi ngắn nhất là do thuật toán chỉ quan tâm tới khoảng cách ước lượng từ một nút tới đích

mà không quan tâm tới quãng đường đã đi từ nút xuất phát tới nút đó. Trong trường hợp khoảng cách từ nút xuất phát tới nút đang xét lớn sẽ làm tổng độ dài đường đi từ xuất phát tới đích qua nút hiện thời lớn lên.

Để khắc phục nhược điểm này, thuật toán A* sử dụng hàm đánh giá $f(n)$ với hai thành phần, thành phần thứ nhất là đường đi từ nút đang xét tới nút xuất phát, thành phần thứ hai là khoảng cách ước lượng tới đích.

Phương pháp: khắc phục nhược điểm của thuật toán tham lam, thuật toán A* sẽ sử dụng hàm $f(n) = g(n) + h(n)$. Trong đó:

- $g(n)$ là giá thành đường đi từ nút xuất phát đến nút n
- $h(n)$ là giá thành ước lượng đường đi từ nút n đến nút đích; h(n) là hàm heuristic.

Trong phần tìm kiếm mù, ta đã xem xét thuật toán tìm kiếm với giá thành thống nhất, trong đó nút biên có hàm $g(n)$ nhỏ nhất được chọn để mở rộng. Như vậy, thuật toán A* khác tìm kiếm với giá thành thống nhất ở chỗ sử dụng thêm hàm $h(n)$ để ước lượng khoảng cách tới đích.

Thuật toán A* yêu cầu hàm $h(n)$ là hàm *chấp nhận được* (admissible) theo định nghĩa sau.

Định nghĩa: Hàm h(n) được gọi là chấp nhận được nếu h(n) không lớn hơn độ dài đường đi thực ngắn nhất từ n tới nút đích.

Thuật toán A* được thể hiện trên Hình 12.

Thuật toán: $A^*(Q, S, G, P, c, h)$

- Đầu vào: bài toán tìm kiếm, hàm heuristic h
- Đầu ra: đường đi ngắn nhất từ nút xuất phát đến nút đích
- Khởi tạo: tập các nút biên (nút mở) $O \leftarrow S$

While(O không rỗng) do

1. Lấy nút n có $f(n)$ nhỏ nhất ra khỏi O
2. Nếu n thuộc G, return(đường đi tới n)
3. Với mọi $m \in P(n)$
   i. $g(m) = g(n) + c(m, n)$
   ii. $f(m) = g(m) + h(m)$
   iii. Thêm m vào O cùng giá trị $f(m)$

Return: không tìm được đường đi

*Hình 12. Thuật toán A**

Với thuật toán A*, có thể sử dụng phiên bản Graph-search (tức là có danh sách

nút đóng) hoặc không. Trong trường hợp không sử dụng nút đóng, tất cả các nút mới sinh ra đều được thêm vào tập biên. Trong trường hợp sử dụng phiên bản Graph-search, một nút đã mở rộng sẽ không được thêm vào danh sách biên. Nếu một nút đã có một bản sao trong tập biên thì bản sao với giá trị hàm *f*(*n*) nhỏ hơn sẽ được giữ lại.

Đặc điểm của thuật toán A*:

- Thuật toán cho *kết quả tối ưu* nếu hàm heuristic *h* là hàm *chấp nhận được*.
- *Thuật toán đầy đủ*, trừ trường hợp có vô số các nút với hàm *f* có giá trị rất nhỏ nằm giữa node xuất phát và node đích.
- Độ phức tạp: trong trường hợp xấu nhất, khi hàm heuristic không có nhiều thông tin, độ phức tạp tính toán và yêu cầu bộ nhớ của A* đều là O(b^m).
- Trong tất cả các thuật toán tìm kiếm tối ưu sử dụng cùng hàm heuristics thì thuật toán A* có độ phức tạp tính toán nhỏ nhất, tức là yêu cầu sinh ra ít nút nhất trước khi tìm ra lời giải.

Yêu cầu bộ nhớ lớn là nhược điểm lớn nhất của A*. Do nhược điểm này, A* thường không thể sử dụng để giải các bài toán có kích thước lớn. Trong phần sau, ta sẽ xem xét một thuật toán với yêu cầu bộ nhớ nhỏ, trong khi vẫn cho phép tìm ra lời giải tối ưu và có độ phức tạp tính toán gần tốt như A*.

5.3 Hàm heuristic

Trong tìm kiếm có thông tin, các hàm heuristic đóng vai trò rất quan trọng. Hàm heuristic tốt đảm bảo tính tối ưu cho những thuật toán như A*. Bên cạnh đó, hàm heuristic tốt cho phép hướng thuật toán theo phía lời giải, nhờ vậy giảm số trạng thái cần khảo sát và độ phức tạp của thuật toán.

Các hàm heuristic *h*(*n*) được xây dựng tùy thuộc vào bài toán cụ thể. Cùng một loại bài toán chúng ta có thể có rất nhiều hàm heuristic khác nhau. Chất lượng hàm heuristic ảnh hưởng rất nhiều đến quá trình tìm kiếm.

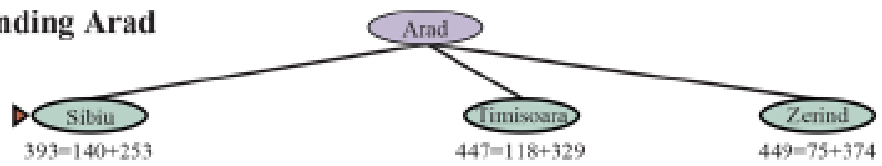
Hàm heuristic *h*(*n*) được gọi là chấp nhận được khi: $h(n) \leq h^*(n)$ trong đó h*(n) là giá thành đường đi thực tế từ *n* đến nút đích. Lưu ý rằng hàm $h(n)=0$ với mọi *n* là hàm chấp nhận được. Để minh hoạ cho hàm heuristic, ta xét một số ví dụ sau.

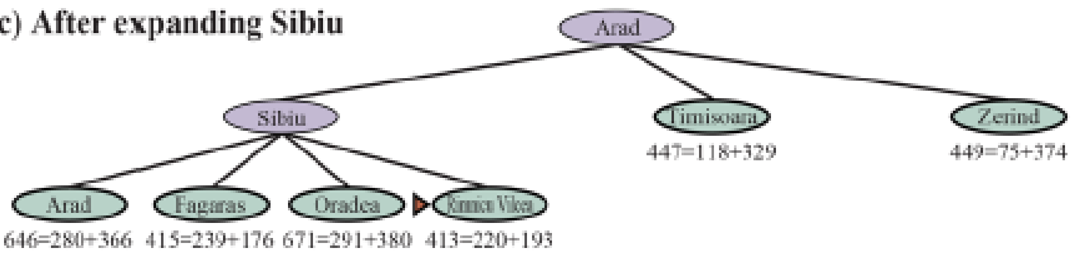
Ví dụ:

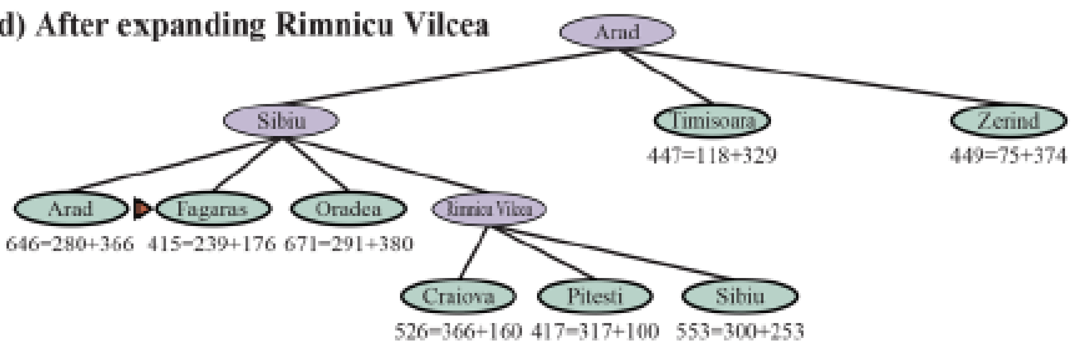
Ví dụ 1: Trong bài toán tìm đường, khoảng cách theo đường chim bay như nhắc tới ở trên là một ví dụ của hàm heuristic chấp nhận được. Hình 13 thể hiện các bước của thuật toán A* cho bài toán tìm đường đi đến Bucharest, trong đó nút được đánh dấu bằng hình tam giác là nút được chọn để mở rộng ở bước tiếp theo. Các nút được gắn nhãn $f=g+h$, với các giá trị *h* là khoảng cách theo đường thẳng đến Bucharest được lấy từ Hình 11.

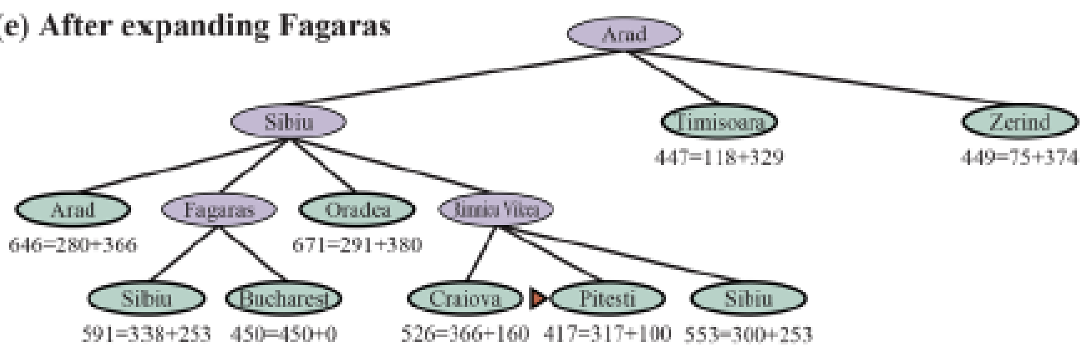
(a) The initial state

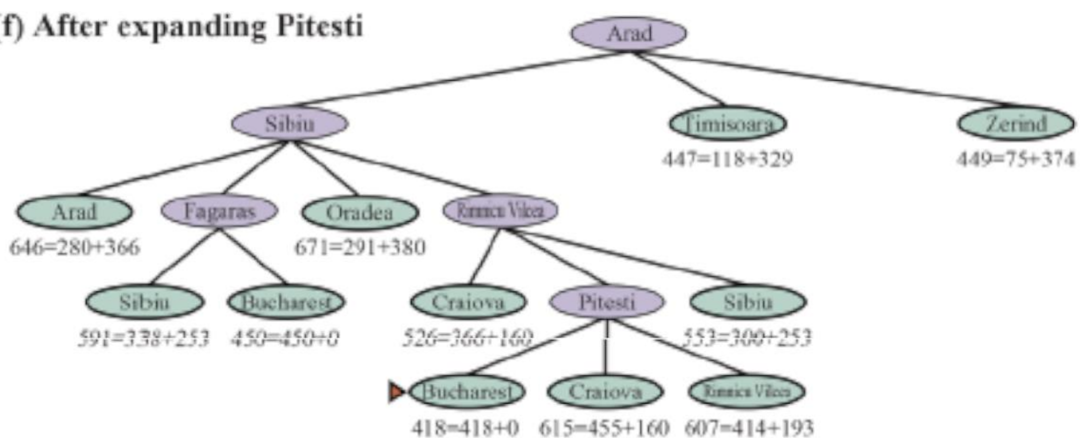
(b) After expanding Arad











Hình 13. Thuật toán A cho bài toán tìm đường đi từ Arad đến Bucharest*

Ví dụ 2: Xét bài toán 8 ô như trong (a) Trạng thái khởi đầu
(b) Trạng thái đích

Hình 14. Mục tiêu của bài toán là trượt các ô (có số) theo chiều ngang hoặc chiều dọc vào chỗ trống cho đến khi khớp với trạng thái đích.

7	2	4
5		6
8	3	1

(a) Trạng thái khởi đầu

	1	2
3	4	5
6	7	8

(b) Trạng thái đích

Hình 14. Bài toán 8 ô

Đối với bài toán này, có thể xây dựng một số hàm heuristic. Dưới đây, ta sẽ xem xét hai trong số các hàm như vậy.

- $h_1(n)$: số ô đặt sai chỗ. Với hình bên phải là trạng thái đích và hình bên trái là trạng thái hiện thời (u) thì trạng thái bên trái có $h_1(u) = 8$ do có tất cả 8 ô nằm sai vị trí. Có thể nhận thấy h_1 là hàm chấp nhận được do muốn di chuyển từ trạng thái bên trái sang trạng thái đích ta phải chuyển vị trí tất cả những ô đứng sai, trong khi để di chuyển mỗi ô sai, ta cần ít nhất một nước đi. Như vậy độ dài đường đi luôn lớn hơn hoặc bằng giá trị của h_1.
- $h_2(n)$: tổng khoảng cách Manhattan giữa vị trí hiện thời của mỗi ô tới vị trí đúng của ô đó ở trạng thái đích. Khoảng cách Manhattan được tính bằng số ít nhất các dịch chuyển theo hàng hoặc cột để đưa một ô tới vị trí của nó trong trạng thái đích. Ví dụ, để đưa ô số 1 tới vị trí ở trạng thái đích ta cần 3 dịch chuyển và do vậy khoảng cách Manhattan của ô 1 tới đích là 3. Giá trị h_2 của trạng thái u trên hình bên trái sẽ bằng $h_2(u) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2$. Hàm h_2 cũng là hàm chấp nhận được. Thật vậy, để di chuyển một ô tới vị trí đích, ta cần ít nhất số nước đi bằng khoảng cách Manhattan từ ô đó tới đích. Như vậy, số nước để di chuyển toàn bộ các ô đứng sai sẽ lớn hơn hoặc bằng tổng khoảng cách Manhattan như cách tính h_2.

Hàm heuristic trội

Ví dụ trên cho thấy, với cùng một bài toán, ta có thể xây dựng đồng thời nhiều hàm heuristic chấp nhận được. Vấn đề đặt ra khi đó là nên dùng hàm nào trong thuật toán tìm kiếm, hàm được chọn phải là hàm tốt hơn, tức là hàm nhanh dẫn tới kết quả hơn.

Giả sử có hai hàm heuristic chấp nhận được $h_1(n)$ và $h_2(n)$. Nếu $h_1(n) \leq h_2(n)$ với mọi n thì ta nói rằng $h_2(n)$ trội hơn so với $h_1(n)$. Rõ ràng hàm $h_2(n)$ sẽ gần với khoảng cách thực tế hơn, mang nhiều thông tin hơn và do vậy là hàm tốt hơn do dẫn tới kết quả nhanh hơn.

Trong trường hợp trong hai hàm $h_1(n)$ và $h_2(n)$ không có hàm trội hơn thì ta có thể tạo ra hàm $h'(n) = \max(h_1(n), h_2(n))$ với mọi n. Rõ ràng, $h'(n)$ là hàm trội hơn hai hàm ban đầu.

Cách xây dựng hàm heuristic

Việc lựa chọn hàm heuristic phụ thuộc vào bài toán cụ thể. Nguyên tắc chung để xây dựng hàm heuristic cho một bài toán là nới lỏng các ràng buộc của bài toán đó khi ước lượng đường đi tới đích. Ví dụ, đối với hàm $h_1(n)$ trong trò đố 8 ô, ta đã bỏ ràng buộc về việc các ô phải di chuyển từng bước để đến được vị trí đích. Hay đối với heuristic đường chim bay, ta đã nới rỏng ràng buộc về việc phải di chuyển trên đường bộ (thường không phải là đường thẳng) và giả sử có thể di chuyển thẳng từ một điểm tới đích.

6 Tìm kiếm cục bộ

Các thuật toán tìm kiếm ở trên đều dựa trên việc khảo sát không gian tìm kiếm một cách hệ thống bằng cách di chuyển theo một số đường đi. Trong quá trình di chuyển, các thuật toán này lưu lại thông tin về đường đi đã qua cùng với thông tin về phương án đã hoặc chưa được xem xét tại mỗi trạng thái trên đường đi. Vấn đề của thuật toán như vậy là việc sử dụng đường đi để khảo sát không gian tìm kiếm một cách hệ thống làm tăng số lượng trạng thái cần xem xét đồng thời đòi hỏi ghi nhớ nhiều trạng thái và do vậy không thích hợp với bài toán có không gian trạng thái lớn.

Trong phần này ta sẽ xem xét các thuật toán *tìm kiếm cục bộ* (local search), còn được gọi là tìm kiếm *cải thiện dần* (iterative improvement). Đặc điểm chính của thuật toán tìm kiếm cục bộ là tại mỗi thời điểm, thuật toán chỉ sử dụng một trạng thái hiện thời, và chỉ xem xét các láng giềng của trạng thái đó. Thuật toán không lưu đường đi hoặc các trạng thái đã khảo sát, do vậy không đòi hỏi nhiều bộ nhớ như các thuật toán đã trình bày ở trên. Mặc dù tìm kiếm cục bộ thường không cho phép tìm được lời giải tối ưu, thuật toán loại này cho phép tìm được lời giải tương đối tốt với thời gian và bộ nhớ nhỏ hơn nhiều so với tìm kiếm có hệ thống, và do vậy có thể sử dụng cho các bài toán có kích thước lớn.

Tìm kiếm cục bộ thường được sử dụng cho những bài toán **tối ưu hóa tổ hợp** hoặc tối ưu hóa rời rạc, tức là những bài toán trong đó cần tìm trạng thái tối ưu hoặc tổ hợp tối ưu trong không gian rời rạc các trạng thái, và *không quan tâm tới đường đi dẫn tới trạng thái đó*. Ví dụ, trong bài toán 8 con hậu, ta chỉ cần tìm vị trí để xếp 8 con hậu vào bàn cờ để không có cặp hậu nào ăn nhau mà không cần quan tâm tới đường di dẫn tới trạng thái đó.

Bài toán tối ưu hóa tổ hợp (tối ưu hóa rời rạc) có những đặc điểm sau.

- Tìm trạng thái tối ưu cực đại hóa hoặc cực tiểu hóa hàm mục tiêu. Không quan tâm tới đường đi.
- Không gian trạng thái rất lớn.
- Không thể sử dụng các phương pháp tìm kiếm trước để xem xét tất cả không gian trạng thái
- Thuật toán cho phép tìm lời giải tốt nhất với độ phức tạp tính toán nhỏ. Thuật toán cũng chấp nhận lời giải tương đối tốt.

Ví dụ: tối ưu hóa tổ hợp là lớp bài toán có nhiều ứng dụng trên thực tế. Có thể kể ra một số ví dụ sau: bài toán lập lịch, lập thời khóa biểu, thiết kế vi mạch, …

Tìm kiếm cục bộ được thiết kế cho bài toán tìm kiếm với không gian trạng thái rất lớn và cho phép tìm kiếm trạng thái tương đối tốt với thời gian tìm kiếm chấp nhận được.

Ý tưởng: tìm kiếm cục bộ dựa trên ý tưởng chung như sau:

- Chỉ quan tâm đến trạng thái đích, không quan tâm đến đường đi.
- Mỗi trạng thái tương ứng với một lời giải (chưa tối ưu) → cải thiện dần bằng cách chỉ quan tâm tới một trạng thái hiện thời, sau đó xem xét để để chuyển sang trạng thái hàm xóm của trạng thái hiện thời (thường là trạng thái có hàm mục tiêu tốt hơn).
- Thay đổi trạng thái bằng cách thực hiện các chuyển động (trạng thái nhận được từ trạng thái *n* bằng cách thực hiện các chuyển động được gọi là hàng xóm của *n*).

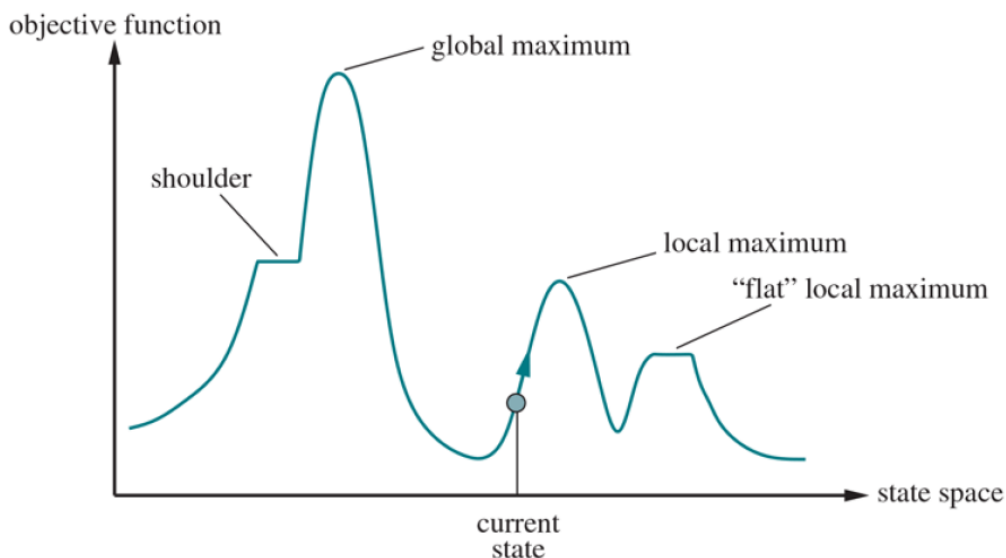
Do tìm kiếm cục bộ chỉ quan tâm tới trạng thái hiện thời và hàng xóm nên cần ít bộ nhớ hơn nhiều so với các phương pháp tìm kiếm có hệ thống ở trên. Tìm kiếm cục bộ thường cho phép tìm được lời giải chấp nhận được kể cả khi bài toán lớn đến mức không dùng được những phương pháp tìm kiếm có hệ thống.

Phát biểu bài toán:

Bài toán tìm kiếm cục bộ được cho bởi những thành phần sau:

- Không gian trạng thái X
- Tập chuyển động để sinh ra hàng xóm
- Hàm mục tiêu Obj: X → R
- Yêu cầu: Tìm trạng thái X* sao cho Obj(X*) là lớn nhất hoặc nhỏ nhất. Lưu ý: có thể dễ dàng biến đổi bài toán tìm trạng thái với hàm mục tiêu lớn nhất thành nhỏ nhất hoặc ngược lại bằng cách nhân hàm mục tiêu với –1.

Để minh họa cho bài toán tìm kiếm cục bộ, ta xét ví dụ trên Hình 15. Trục hoành trên hình vẽ thể hiện không gian các trạng thái (để cho đơn giản, không gian trạng thái ở đây được thể hiện trong không gian một chiều dưới dạng các điểm trên trục hoành), trục tung là độ lớn của hàm mục tiêu. Yêu cầu bài toán tối ưu hóa tổ hợp là tìm được trạng thái (điểm trên trục hoành) có hàm mục tiêu lớn nhất. Lưu ý, hình vẽ minh họa trường hợp cần tìm trạng thái với hàm mục tiêu lớn nhất, tuy nhiên trong bài toán khác có thể yêu cầu tìm trạng thái với hàm mục tiêu nhỏ nhất.



Hình 15. Bài toán tìm kiếm cục bộ với không gian trạng thái và hàm mục tiêu

6.1 Thuật toán leo đồi

Leo đồi (Hill climbing) là tên chung để chỉ một họ các thuật toán có nguyên lý giống nhau. Thuật toán thực hiện bằng cách tạo ra hàng xóm cho trạng thái hiện thời và di chuyển sang hàng xóm có hàm mục tiêu tốt hơn, tức là di chuyển lên cao đối với trường hợp cần cực đại hóa hàm mục tiêu. Thuật toán dừng lại khi đạt tới một đỉnh cực đại (toàn cục hoặc địa phương) của đồ thị hàm mục tiêu, tương ứng với trạng thái không có hàng xóm nào tốt hơn. Đỉnh này có thể là đỉnh cao nhất (cực đại toàn cục), hoặc cũng là những đỉnh thấp hơn (cực đại địa phương) Hình 15. Thuật toán leo đồi không lưu lại những trạng thái đã qua, đồng thời không nhìn xa hơn hàng xóm của trạng thái hiện thời.

Thuật toán leo đồi còn được gọi là thuật toán tìm cục bộ tham lam do tại mỗi thời điểm thuật toán này chỉ xét một láng giềng tốt mà không quan tâm tới tương lai xa hơn.

6.1.1 Di chuyển sang trạng thái tốt nhất

Có nhiều phiên bản khác nhau của thuật toán leo đồi. Một trong những phiên bản thông dụng nhất có tên là leo đồi *di chuyển sang trạng thái tốt nhất* (best-improvement hill climbing). Phiên bản này của leo đồi lựa chọn trong số hàng xóm hiện thời hàng xóm có hàm mục tiêu tốt nhất. Nếu hàng xóm đó tốt hơn trạng thái hiện thời thì di chuyển sang hàm xóm đó. Nếu ngược lại thì kết thúc và trả về trạng thái hiện thời. Thuật toán đầy đủ được thể hiện trên Hình 16. Bước 1 là bước khởi tạo, trong đó ta chọn ngẫu nhiên trạng thái xuất phát. Bước 2 sinh ra các láng giềng của trạng thái hiện thời. Ở bước 3, thuật toán kiểm tra các láng giềng, nếu không có láng giềng nào tốt hơn thuật trạng thái hiện thời (tất cả láng giềng có giá trị hàm mục tiêu Obj không lớn hơn Obj của trạng thái hiện thời) thì kết thúc và trả về trạng thái hiện thời là kết quả. Trong trường hợp ngược lại, ở bước 4, thuật toán chọn láng giềng có giá trị hàm mục tiêu Obj lớn nhất và chuyển sang trạng thái đó, sau đó lặp lại từ bước 2.

Đầu vào: bài toán tối ưu tổ hợp

Đầu ra: trạng thái với hàm mục tiêu lớn nhất (hoặc cực đại địa phương)

----

1. Chọn ngẫu nhiên trạng thái x
2. Gọi Y là tập các trạng thái hàng xóm của x
3. If $\forall y_i \in Y$: Obj (y_i) < Obj (x) then

   Kết thúc và trả lại x là kết quả
4. $x \leftarrow y_i$, trong đó $i = \text{argmax}_i$ (Obj (y_i))
5. Go to 2

Hình 16. Thuật toán leo đồi di chuyển sang trạng thái tốt nhất

Đặc điểm của leo đồi

- Đơn giản, dễ lập trình. Trong các thuật toán trình bày ở phần trước, khi lập trình cần sử dụng các cấu trúc dữ liệu để biểu diễn các nút, nhớ tập biên.

Thuật toán leo đồi không đòi hỏi phải lập trình với các cấu trúc như vậy.

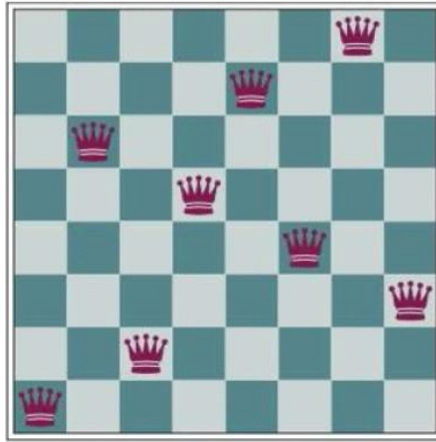
- Như các giải thuật tìm kiếm cục bộ khác, leo đồi sử dụng ít bộ nhớ do không phải lưu lại các trạng thái. Tại mỗi thời điểm, thuật toán chỉ cần lưu lại trạng thái hiện thời và một tráng thái láng giềng.
- Bên cạnh hai ưu điểm trên, leo đồi dễ bị lời giải tối ưu cục bộ (cực trị địa phương) tương ứng với đỉnh các "đồi" thấp trong hình 2.17. Nếu bắt đầu từ một trạng thái nằm trong phạm vi của các vùng "đồi" thấp, thuật toán sẽ di chuyển về trạng thái tương ứng với đỉnh đồi, sau đó dừng lại ở đỉnh do các trạng thái xung quanh đều không tốt bằng. Mặc dù đỉnh đồi thấp tương ứng với trạng thái tốt nhất trong một vùng, trạng thái đó không phải là trạng thái tối ưu trong toàn bộ không gian trạng thái. Để khắc phục phần nào vấn đề này, thuật toán được thực hiện nhiều lần, mỗi lần sử dụng một trạng thái xuất phát sinh ngẫu nhiên khác với trạng thái xuất phát trong những lần trước đó. Nếu số lần thực hiện với trạng thái xuất phát ngẫu nhiên như vậy được thực hiện đủ nhiều thì xác suất tìm ra lời giải tối ưu sẽ tiến tới 1.

Khi thiết kế thuật toán leo đồi với cách chuyển động sang trạng thái láng giềng tốt nhất, việc lựa chọn chuyển động rất quan trọng. Nếu nhiều chuyển động, từ trạng thái hiện thời có thể sinh ra nhiều láng giềng. Do thuật toán cần so sánh tất các láng giềng để tìm ra trạng thái tốt nhất, mỗi bước của vòng lặp sẽ đòi hỏi nhiều thời gian do phải tính hàm mục tiêu cho tất cả láng giềng. Ngược lại, nếu sinh ra tập láng giềng nhỏ sẽ dễ dẫn tới cực trị địa phương do không vượt qua được những "hố" nhỏ trên đường đi.

Ví dụ minh hoạ 1: bài toán 8 con hậu: sắp xếp 8 con hậu vào một bài cờ vua để không có hai con hậu nào có thể ăn được nhau (quân hậu ăn theo hàng, cột hoặc đường chéo). Ta sẽ quy định mỗi trạng thái là sắp xếp của cả 8 con hậu trên bàn cờ, sao cho mỗi cột chỉ gồm 1 con. Trạng thái xuất phát được khởi tạo ngẫu nhiên. Hàm mục tiêu là hàm heuristic được tính bằng số các đôi hậu đang đe dọa nhau. Như vậy hàm mục tiêu càng nhỏ thì càng gần với trạng thái đích cần tìm. Trạng thái đích là trạng thái có hàm mục tiêu nhỏ nhất (bằng 0). Để minh họa cho ảnh hưởng của việc lựa chọn hàm chuyển động , ta xét hai cách chuyển động sau.

Cách 1: chọn một cột, dịch chuyển quân hậu trong cột đó sang dòng khác. Với cách chuyển động như vậy, từ mỗi trạng thái có thể sinh ra 7 x 8 = 56 láng giềng. Hình 17b là ví dụ một trạng thái với hàm mục tiêu bằng 17, đồng thời trên các ô trống là giá trị hàm mục tiêu (số đôi hậu đe dọa nhau) ứng với chuyển động di chuyển quân hậu trong cùng cột tới ô đó. Trạng thái này có 8 láng giềng cùng là tốt nhất với hàm mục tiêu bằng 12. Với số lượng láng giềng tương đối ít, cách này dễ dẫn tới cực trị địa phương, chẳng hạn trạng thái ở Hình 17a có hàm mục tiêu bằng 1 nhưng tất cả láng giềng đều có hàm mục tiêu lớn hơn hoặc bằng và do vậy không tìm được lời giải.

Cách 2: chọn hai cột, dịch chuyển đồng thời mỗi quân hậu trong hai cột đó sang dòng khác với dòng hiện thời. Cách này tạo ra 7 x 8 + 7 x 7 láng giềng cho mỗi trạng thái. Mặc dù số lượng láng giềng lớn đòi hỏi mỗi bước lặp phải tính toán nhiều gần gấp đôi so với cách 1 nhưng sẽ ít khả năng dẫn tới cực trị địa phương như trạng thái trên Hình 17b hơn.



18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14		13	16	13	16
	14	17	15		14	16	16
17		16	18	15		15	
18	14		15	15	14		16
14	14	13	17	12	14	12	18

(a) (b)

Hình 17. Bài toán 8 hậu với giá trị hàm mục tiêu cho mỗi trạng thái láng giềng (b) và trạng thái cực trị địa phương (a)

6.1.2 Leo đồi ngẫu nhiên

Leo đồi ngẫu nhiên (stochastic hill climbing) là một phiên bản khác của leo đồi. Thay vì tìm ra hàng xóm tốt nhất, phiên bản này lựa chọn ngẫu nhiên một hàng xóm. Nếu hàng xóm đó tốt hơn trạng thái hiện thời, hàng xóm đó sẽ được chọn làm trạng thái hiện thời và thuật toán lặp lại. Ngược lại, nếu hàng xóm được chọn không tốt hơn, thuật toán sẽ chọn ngẫu nhiên một hàng xóm khác và so sánh. Thuật toán kết thúc và trả lại trạng thái hiện thời khi đã hết "kiên nhẫn". Thông thường, độ kiên nhẫn được cho bằng số lượng tối đa hàng xóm mà thuật toán xem xét trong mỗi bước lặp hoặc trong toàn bộ thuật toán.

Thuật toán leo đồi ngẫu nhiên được thể hiện trong Hình 18.

Đầu vào: bài toán tối ưu tổ hợp

Đầu ra: trạng thái với hàm mục tiêu lớn nhất (hoặc cực đại địa phương)

----

1. Chọn ngẫu nhiên trạng thái x
2. Gọi Y là tập các trạng thái hàng xóm của x
3. Chọn ngẫu nhiên $y_i \in Y$
4. Nếu Obj (yi) > Obj (x) thì

   $x \leftarrow yi$
5. Go to 2 nếu chưa hết kiên nhẫn

Hình 18. Thuật toán leo đồi ngẫu nhiên

Các thực nghiệm trên nhiều bài toán cho thấy, trong trường hợp mỗi trạng thái có nhiều láng giềng, leo đồi ngẫu nhiên cho kết quả nhanh hơn do thuật toán có thể chuyển sang trạng thái tiếp theo mà không cần khảo sát toàn bộ tập láng giềng. Ngoài ra, leo đồi ngẫu nhiên cũng ít gặp phải cực trị địa phương hơn leo đồi chuyển sang trạng thái tốt nhất.

Nhìn chung, khả năng tìm được lời giải tối ưu của các thuật toán leo đồi phụ thuộc nhiều vào không gian trạng thái. Đối với những không gian có ít cực trị địa phương, leo đồi thường tìm được lời giải khá nhanh. Trong trường hợp không gian trạng thái phức tạp, thuật toán thường chỉ tìm được cực trị địa phương. Tuy nhiên, bằng cách thực hiện nhiều lần với trạng thái xuất phát ngẫu nhiên, leo đồi thường tìm được cực trị địa phương khá tốt.

6.2 Thuật toán tôi thép

Một vấn đề lớn với leo đồi là thuật toán không có khả năng "đi xuống" và do vậy không thoát khỏi được cực trị địa phương khi đã rơi vào. Ngược lại, cách di chuyển hoàn toàn ngẫu nhiên (random walk) có thể khảo sát toàn bộ không gian trạng thái nhưng không hiệu quả. *Thuật toán tôi thép* hay *mô phỏng luyện kim* (simulated annealing) là một phương pháp tìm kiếm cục bộ cho phép giải quyết phần nào vấn đề cực trị địa phương một cách tương đối hiệu quả.

Có thể coi tôi thép là phiên bản của thuật toán leo đồi ngẫu nhiên, trong đó thuật toán chấp nhận cả những trạng thái kém hơn trạng thái hiện thời với một xác suất *p* nào đó. Cụ thể là khi lựa chọn ngẫu nhiên một hàng xóm, nếu hàng xóm đó kém hơn trạng thái hiện thời, thuật toán có thể quyết định di chuyển sang đó với một xác suất *p*.

Vấn đề quan trọng đối với thuật toán là lựa chọn xác suất *p* thế nào. Nguyên tắc chung là không chọn *p* cố định, giá trị *p* được xác định dựa trên hai yếu tố sau.

- Nếu trạng thái mới kém hơn nhiều so với trạng thái hiện thời thì p phải giảm đi. Có nghĩa là xác suất chấp nhận trạng thái tỷ lệ nghịch với độ kém của trạng thái đó. Gọi $\Delta(x,y)$ = Obj(x) – Obj(y) trong đó x là trạng thái hiện thời, ta cần chọn p tỷ lệ nghịch với $\Delta(x,y)$.
- Theo thời gian, giá trị của *p* phải giảm dần. Ý nghĩa của việc giảm p theo thời gian là do khi mới bắt đầu, thuật toán chưa ở vào vùng trạng thái tốt và do vậy chấp nhận thay đổi lớn. Theo thời gian, thuật toán sẽ chuyển sang vùng trạng thái tốt hơn và do vậy cần hạn chế thay đổi.

Thuật toán tôi thép được thể hiện trên Hình 19. Khác với thuật toán leo đồi, trong đó ta luôn chuyển động sang trạng thái tốt hơn và do vậy trạng thái hiện thời luôn là trạng thái tốt nhất, thuật toán tôi thép có thể di chuyển sang trạng thái kém hơn. Do vậy, thuật toán tôi thép sử dụng một biến *x** để lưu trạng thái tốt nhất đã từng khảo sát, tính tới thời điểm hiện tại. Lệnh **if** rand[0,1] < p **then** x ← y có nghĩa là gán y cho x với xác suất p, trong đó rand[0,1] là hàm trả về giá trị ngẫu nhiên trong khoảng [0,1]. Lệnh này có thể triển khai trên máy tính có hàm sinh số tự nhiên.

Thuật toán tôi thép dựa trên một hiện tượng cơ học là quá trình làm lạnh kim loại để tạo ra cấu trúc tinh thể bền vững. Hàm mục tiêu khi đó được đo bằng độ vững chắc của cấu trúc tinh thể. Khi còn nóng, mức năng lượng trong kim loại cao, các nguyên tử kim loại có khả năng di chuyển linh động hơn. Khi nhiệt độ T giảm xuống, tinh thể dần chuyển tới trạng thái ổn định và tạo ra mạng tinh thể. Bằng cách thay đổi nhiệt độ hợp lý, có thể tạo ra những mạng tinh thể rất rắn chắc.

Chính vì sự tương tự với cách tôi kim loại như vậy nên trong thuật toán, xác suất p giảm theo thời gian dựa vào một công thức gọi là sơ đồ làm lạnh C. Với sơ đồ làm lạnh này, khi số vòng lặp tăng lên, tức là t tăng, giá trị của T giảm. Tiếp theo, từ công thức tính xác suất

sử dụng trong thuật toán, có thể nhận thấy khi T tăng và tiến tới vô cùng, xác suất $p \to 1$ với mọi $\Delta(x, y)$. Khi đó, thuật toán chấp nhận bất cứ chuyển động nào bất kể trạng thái mới tốt hơn hay kém hơn, tức là thuật toán di chuyển ngẫu nhiên trong không gian trạng thái.

```
SA(X, Obj, N, m, x, C) //Obj càng nhỏ càng tốt
Đầu vào:   số bước lặp m
           trạng thái bắt đầu x (chọn ngẫu nhiên)
           sơ đồ làm lạnh C
Đầu ra:    trạng thái tốt nhất x*
Khởi tạo:  x* = x
For i = 1 to m
    1. chọn ngẫu nhiên y ∈ N(x)
       a) tính Δ(x,y) = Obj(y) – Obj(x)
       b) if Δ(x,y) < 0 then p = 1
       c) else p = exp(- Δ(x,y)/T)
       d) if rand[0,1] < p then x ← y //gán y cho x với xác suất p.
          if Obj(x) < Obj(x*) then x* ← x
    2. giảm T theo sơ đồ C
return x*     //x* là trạng thái tốt nhất trong số những trạng thái đã xem xét
```

Hình 19. Thuật toán tôi thép

Ngược lại, khi $T \to 0$, ta có $p \to 0$ với mọi $\Delta(x, y)$, tức là thuật toán không chấp nhận các trạng thái kém hơn trạng thái hiện thời và do vậy trở thành leo đồi ngẫu nhiên. Như vậy, có thể coi thuật toán tôi thép là sự kết hợp giữa leo đồi ngẫu nhiên với chuyển động ngẫu nhiên, trong đó khởi đầu xu hướng chuyển động ngẫu nhiên lớn hơn, và càng về cuối, xu hướng leo đồi càng chiếm ưu thế.

Việc lựa chọn các tham số cho sơ đồ làm lạnh thường được thực hiện bằng cách thực nghiệm với từng bài toán cụ thể.

Thuật toán tôi thép được dùng nhiều trong việc thiết kế vi mạch có độ tích hợp lớn cũng như giải quyết những bài toán tối ưu hóa tổ hợp có kích thước lớn trên thực tế. Nhiều ứng dụng cho thấy, thuật toán tôi thép có khả năng tìm được lời giải tốt hơn so với thuật toán leo đồi.

6.3 Giải thuật di truyền

Giải thuật di truyền (genetic algorithm) hay *thuật toán di truyền* là thuật toán tìm kiếm được thiết kế dựa trên sự tương tự với quá trình chọn lọc tự nhiên và thuyết tiến

hoá của Charles Darwin. Giải thuật di truyền là trường hợp riêng của một lớp giải thuật được gọi là *giải thuật tiến hoá* (evolutionary algorithms). Giải thuật di truyền cho lời giải tốt trong nhiều bài toán tối ưu và được sử dụng rộng rãi trong rất nhiều ứng dụng khác nhau.

Về mặt thuật toán, có thể xem giải thuật di truyền như một phiên bản leo đồi ngẫu nhiên với một số khác biệt chính. Thứ nhất, thuật toán đồng thời duy trì nhiều lời giải tại mỗi thời điểm thay vì một lời giải như leo đồi. Thứ hai, từng cặp lời giải có thể trao đổi thành phần với nhau để tạo ra lời giải mới.

Về mặt ý tưởng, giải thuật di truyền học tập từ quá trình sinh tồn và chọn lọc tự nhiên của sinh vật trong tự nhiên, trong đó cá thể với khả năng thích nghi cao hơn sẽ chiếm ưu thế và tồn tại. Nếu ta coi đây là bài toán tối ưu thì quá trình tiến hoá sẽ sinh ra cá thể tối ưu, tức là cá thể thích nghi cao.

6.3.1 Nguyên lý chung.

Giải thuật di truyền mô phỏng quá trình tiến hoá qua các thế hệ trong tự nhiên như sau. Mỗi thế hệ gồm một số nhất định *lời giải*, còn gọi là *cá thể*. Mỗi lời giải hay cá thể được biểu diễn bằng một *nhiễm sắc thể*, chứa các *gen*. Độ tốt của lời giải được đánh giá bằng *hàm thích nghi* (fitness). Thông thường, hàm thích nghi chính là hàm mục tiêu của bài toán tối ưu. Các lời giải trong mỗi thế hệ được biến đổi trong quá trình thích nghi để tạo ra thế hệ tiếp theo.

Nguyên bản sinh học. Trong sinh học, mỗi cá thể được mã hoá di truyền bằng bộ gen, bộ gen có thể gồm nhiều nhiễm sắc thể như của người hoặc động vật bậc cao, mỗi nhiễm sắc thể gồm nhiều gen. Khi sinh sản, hai cá thể bố mẹ trao đổi các gen với nhau, cá thể con nhận một số gen từ bố và một số gen từ mẹ.

Giải thuật di truyền sử dụng các quy tắc sau để tạo ra thế hệ lời giải tiếp theo.

- Các cá thể cạnh tranh với nhau. Cá thể với độ thích nghi cao hơn sẽ tạo ra nhiều hậu duệ hơn cá thể kém thích nghi.
- Gen từ các cá thể thích nghi tốt được kết hợp với nhau, nhờ đó có thể tạo ra hậu duệ tốt hơn tổ tiên.
- Từng cá thể cũng có thể thay đổi gen của mình.
- Nhờ vậy, thế hệ tiếp theo sẽ chứa cá thể thích nghi hơn với môi trường.

6.3.2 Biểu diễn lời giải và không gian tìm kiếm

Để sử dụng giải thuật di truyền, trước hết mỗi lời giải phải được biểu diễn bằng một nhiễm sắc thể, chứa các gen. Ở đây, nhiễm sắc thể là một vec tơ có độ dài xác định, mỗi phần tử của vec tơ là một gen. Thông thường, lời giải được biểu diễn bằng vec tơ nhị phân, tức là phần tử nhận giá trị {0, 1}, nhưng cũng có thể sử dụng phần tử là số hoặc chữ.

Mỗi lời giải được gán một *độ thích nghi* (fitness) thể hiện khả năng cạnh tranh của lời giải. Thông thường, độ thích nghi được tính từ hàm mục tiêu của bài toán, hàm mục tiêu tốt tương đương với độ thích nghi cao.

Giải thuật thực hiện qua nhiều bước lặp, mỗi bước lặp tương ứng với một thế hệ. Tại mỗi thế hệ, giải thuật duy trì một quần thể, tức là một tập hợp gồm N lời giải, trong đó N là tham số. Quần thể trong thế hệ đầu tiên được khởi tạo ngẫu nhiên bằng cách sinh ngẫu

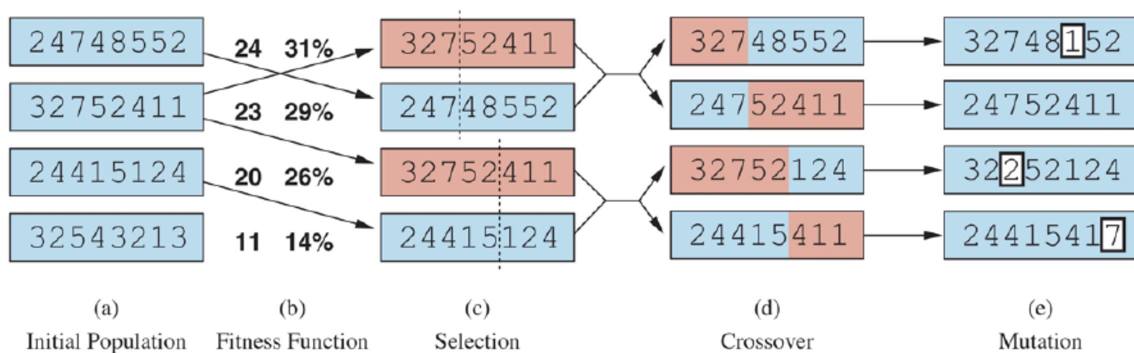
nhiên các lời giải. Từ quần thể hiện tại, từng cặp cá thể được lựa chọn dựa trên độ thích nghi và được lai ghép với nhau để tạo ra cá thể con. Cá thể với độ thích nghi cao được lựa chọn với xác suất cao hơn. Các cá thể con có thể bị đột biến ở một số gen. Các cá thể con được tạo ra như vậy sẽ thay thế các cá thể bố mẹ để tạo ra quần thể mới trong thế hệ tiếp theo.

Như vậy, thế hệ tiếp theo sẽ chứa các lời giải với các gen tốt hơn mức trung bình trong thế hệ trước. Nói cách khác, thế hệ sau chứa các lời giải thành phần tốt hơn thế hệ trước. Thuật toán lặp lại cho đến khi cá thể trong thế hệ tiếp theo không khác nhiều so với thế hệ trước. Khi đó thuật toán được gọi là *hội tụ*. Cũng có thể sử dụng các tiêu chí kết thúc khác, chẳng hạn sau khi lặp một số lần nhất định.

6.3.3 Giải thuật

Quần thể đầu tiên được khởi tạo ngẫu nhiên. Sau đó thuật toán được thực hiện qua nhiều bước lặp, tại mỗi bước lặp thuật toán sinh ra một quần thể mới. Các quần thể tiếp theo được tạo ra từ quần thể trước đó bằng cách áp dụng ba thao tác: *chọn lọc*, *lai ghép* và *đột biến*.

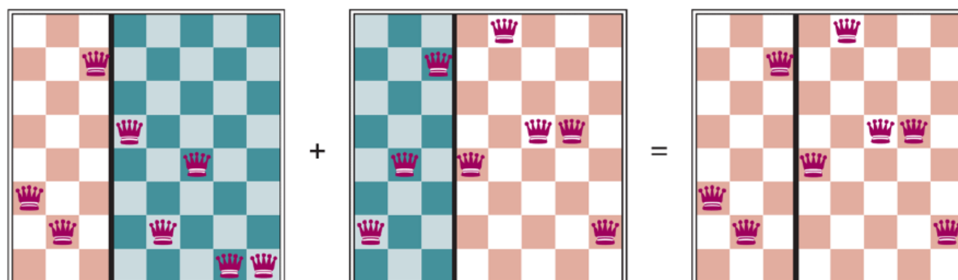
Để minh hoạ cho các bước của thuật toán, ta xét ví dụ bài toán 8 quân hậu, trong đó mỗi quân hậu được đặt trong một cột riêng. Mỗi trạng thái gồm thông tin về vị trí 8 quân hậu trong cột của mình. Nếu chọn biểu diễn dưới dạng chuỗi bit, cần 3 bit cho một vị trí, tức là 24 bit cho mỗi trạng thái. Nếu biểu diễn dưới dạng chuỗi số thập phân, cần 8 chữ số thập phân cho mỗi trạng thái. Trong ví dụ minh hoạ này, ta sẽ dùng biểu diễn dưới dạng thập phân như minh hoạ trên Hình 20. Chẳng hạn, nhiễm sắc thể 24415124 cho biết quân hậu thứ nhất nằm ở hàng thứ 2 trong cột đầu tiên bên trái, quân thứ hai nằm ở hàng 4 trong cột tiếp theo. Do trạng thái tốt ứng với giá trị hàm thích nghi lớn nên ta sẽ chọn độ thích nghi bằng *số lượng đôi hậu không đe doạ nhau*. Trạng thái lời giải là trạng thái có độ thích nghi bằng 28. Ở Hình 20, quần thể ban đầu (a) được xếp hạng theo hàm thích nghi (b) và dựa trên đó tính ra xác xuất được chọn để lai ghép (c). Chúng tạo ra con cái ở (d) và những cá thể con này có thể bị đột biến (e).



Hình 20. Minh họa giải thuật di truyền cho bài toán 8 quân hậu

Chọn lọc (selection). Chọn lọc cá thể để tham gia việc tạo ra thế hệ tiếp theo. Cá thể có độ thích nghi càng cao càng có nhiều khả năng được chọn. Có nhiều phương pháp thực hiện lựa chọn như vậy, chi tiết sẽ được trình bày ở dưới. Trong ví dụ trên Hình 20, hai cặp bố mẹ được chọn ở (c), phù hợp với xác suất trong (b). Cá thể thứ nhất được chọn 1 lần, các thể thứ hai được chọn 2 lần, cá thể thứ ba được chọn một lần và cá thể thứ tư không được chọn.

Lai ghép (crossover), còn gọi là trao đổi chéo, hay tái tổ hợp. Lai ghép là thao tác kết hợp các phần tử nhiễm sắc thể của bố và mẹ để tạo ra hai cá thể con với một xác suất nhất định. Cá thể bố và mẹ được lựa chọn bằng cách sử dụng thao tác chọn lọc ở trên. Sau đó, nhiễm sắc thể bố và mẹ được cắt ra. Phần đầu nhiễm sắc thể bố được nối với phần đuôi nhiễm sắc thể mẹ và ngược lại. Đối với mỗi cặp được chọn, vị trí cắt được chọn ngẫu nhiên. Trên Hình 20, ở phép lai ghép đầu tiên, vị trí cắt ở giữa gen thứ 3 và gen thứ 4. Hình 21 minh hoạ kết quả lai ghép giữa hai nhiễm sắc thể bố mẹ thứ nhất và thứ hai ở Hình 20.



Hình 21. Kết quả lai ghép giữa cặp bố mẹ thứ nhất và thứ hai

Cũng có thể cắt mỗi nhiễm sắc thể bố mẹ thành nhiều hơn hai phần và ghép các phần con với nhau. Việc chọn điểm cắt và số phần con phụ thuộc bài toán cụ thể. Như vậy, nhờ kết quả lai ghép, mỗi cá thể con nhận một phần gen từ bố và một phần gen từ mẹ. Lưu ý là, với mỗi cặp bố mẹ, thao tác lai ghép được thực hiện với một xác suất nhất định gọi là *xác suất lai ghép*. Xác suất này thường được chọn lớn hơn 0.5.

Đột biến (mutation). Thay đổi giá trị các gen của cá thể con vừa tạo ra với một xác suất nhất định. Cụ thể, với mỗi cá thể con, duyệt các gen và với một xác suất rất nhỏ thay đổi giá trị của gen đó (0 thành 1 và ngược lại nếu biểu diễn nhị phân). Xác suất này thường được chọn nhỏ hơn 0.1 để tránh đột biến quá nhiều. Trong ví dụ ở Hình 20, gen thứ 6 của cá thể con số 1 được thay đổi giá trị. Ngoài ra, còn có đột biến ở con thứ ba và thứ tư. Trong bài toán 8 quân hậu, điều này tương ứng với việc chọn ngẫu nhiên một quân hậu và di chuyển quân hậu đến một ô vuông ngẫu nhiên trong cột của nó.

Mục đích của thao tác đột biến là để tạo ra những đoạn gen hoàn toàn mới, chưa có trong quần thể cha mẹ. Đột biến cũng cho phép hạn chế việc hội tụ quá sớm của thuật toán. Về bản chất, đột biến có hiệu quả tương tự di chuyển ngẫu nhiên trong không gian trạng thái, có thể cho phép vượt qua các trường hợp cực trị địa phương.

Tại mỗi bước lặp, thuật toán thực hiện ba thao tác trên này để sinh ra quần thể mới. Quá trình lặp được thực hiện cho tới khi thuật toán hội tụ, tức là cá thể con không khác nhiều cá thể bố mẹ, hoặc khi thực hiện đủ một số lượng vòng lặp do người dùng quy định. Toàn bộ thuật toán được trình bày ở Hình 22.

6.3.4 Giá trị xác suất.

Xác suất lai ghép. Nếu xác suất lai ghép là 1 (100%) thì toàn bộ cá thể con sẽ được tạo ra do lai ghép. Ngược lại, nếu xác suất lai ghép là 0 thì toàn bộ cá thể con là bản sao của một số cá thể bố mẹ nhưng không nhất thiết quần thể tiếp theo trùng với quần thể cũ. Như đã nói ở trên, xác suất lai ghép được lựa chọn tương đối lớn, thường từ 0.5 trở lên.

Xác suất đột biến. Nếu xác suất đột biến là 100% thì toàn bộ cá thể sau khi lai ghép

sẽ bị thay đổi. Ngược lại, nếu xác suất này là 0 thì không cá thể nào bị thay đổi. Xác suất đột biến được lựa chọn rất nhỏ, ít khi vượt quá 0.1. Xác suất đột biến nhỏ để tránh cho thuật toán di chuyển theo kiểu ngẫu nhiên.

```
GA(X, f, N, c, m)
Đầu vào:   bài toán tối ưu với không gian trạng thái X
           hàm thích nghi f
           kích thước quần thể N
           xác suất lai ghép c
           xác suất đột biến m
Đầu ra:    lời giải (cá thể) với độ thích nghi cao
Khởi tạo:  Khởi tạo ngẫu nhiên quần thể G gồm N lời giải
---------------------------------------------
While (chưa thoả mãn điều kiện dừng) do
       1. For i = 1 to N do
          Tính giá trị hàm thích nghi f(i) cho cá thể thứ i
       2. For i = 1 to ≤⌊N/2f⌋ do
          i.   Chọn lọc: chọn 2 cá thể bố mẹ x và y từ G tuỳ theo giá trị thích nghi
          ii.  Lai ghép: với xác suất c, đổi chỗ đoạn gen trên x và y
          iii. Đột biến: với xác suất m, thay đổi giá trị các gen trên cá thể mới tạo ra
          iv.  Thêm cá thể mới vào quần thể mới G’
       3. Gán G ← G’
Return: Lời giải thuộc G với giá trị thích nghi tốt nhất
```

Hình 22. Giải thuật di truyền

6.3.5 Kích thước quần thể.

Kích thước quần thể N là số lượng cá thể được duy trì trong mỗi thế hệ. Nếu N quá nhỏ, thuật toán có ít lựa chọn để thực hiện lai ghép, dẫn tới chỉ một phần không gian tìm kiếm được khảo sát và do vậy có thể không tìm được lời giải tốt. Nếu N quá lớn, thuật toán sẽ thực hiện chậm do phải xử lý nhiều trong mỗi vòng lặp. Giá trị tốt của N phụ thuộc vào bài toán cụ thể và cách mã hoá lời giải. Tuy nhiên, nhiều kết quả thực nghiệm cho thấy, khi N tăng tới một mức độ nào đó, chất lượng lời giải không tăng, trong khi thuật toán sẽ chậm hơn.

6.3.6 Mã hoá lời giải

Lời giải hay cá thể cần được mã hoá dưới dạng nhiễm sắc thể. Cách mã hoá cụ thể phụ thuộc rất nhiều vào từng bài toán. Sau đâu là một số cách mã hoá thường gặp.

- Mã hoá dưới dạng chuỗi bit hay dạng nhị phân. Mỗi nhiễm sắc thể là một vec tơ gồm các số 0 hoặc 1. Đây là cách mã hoá thông dụng nhất. Cách mã hoá tạo ra rất

nhiều nhiễm sắc thể. Tuy nhiên, với nhiều bài toán, cách mã hoá này không tự nhiên và việc lai ghép hay đột biến có thể tạo ra nhiễm sắc thể không tương ứng với lời giải hợp lệ nào và do vậy cần xử lý để nhận được lời giải hợp lệ.

- Mã hoá dưới dạng các hoán vị. Cách mã hoá này thường sử dụng trong bài toán cần xác định thứ tự như đường đi hay thời khoá biểu. Ví dụ, trong bài toán người bán hàng, mỗi gen tương ứng với một thành phố và nhiễm sắc thể mã hoá thứ tự đường đi dưới dạng hoán vị thứ tự thành phố.

Nhiễm sắc thể	1 5 3 2 6 4 7 9 8

- Mã hoá dưới dạng các giá trị. Trong một số trường hợp, các giá trị như số thực được sử dụng để biểu diễn lời giải. Việc sử dụng trực tiếp số thực là cần thiết do việc biểu diễn bằng bit trong trường hợp này rất phức tạp. Với cách mã hoá này, nhiễm sắc thể là một chuỗi giá trị, trong đó giá trị có thể là số thực , số nguyên, chữ cái hoặc xâu ký tự. Trong bài toán 8 quân hậu ở trên, ta đã thấy vị trí các con hậu được biểu diễn bởi vec tơ các số nguyên. Một ví dụ khác là khi huấn luyện mạng nơ ron, cần xác định trọng số cho các kết nối. Nhiễm sắc thể khi đó chứa các trọng số này.

6.3.7 Các phương pháp lai ghép

Thao tác lai ghép rất đa dạng và phụ thuộc vào cấu trúc của lời giải trong bài toán cụ thể. Các lựa chọn bao gồm:

- Lựa chọn điểm cắt. Thông thường sử dụng cùng điểm cắt trên cả bố và mẹ nhưng cũng có thể sử dụng điểm cắt khác nhau. Trường hợp sau sinh ra cá thể con có độ dài thay đổi và do vậy cần xử lý riêng sau đó.
- Lựa chọn số đoạn gen. Thông thường nhiễm sắc thể được cắt làm hai phần nhưng cũng có thể nhiều hơn.

Ví dụ lai ghép với hai điểm cắt:

11|0010|**11** + 11|**0111**|11 = **11011111**

Với cách mã hoá kiểu hoán vị như trong bài toán người bán hàng, khi lai ghép với một điểm cắt, phần trước điểm cắt được lấy nguyên từ cá thể bố mẹ thứ nhất. Phần sau điểm cắt được tạo ra bằng cách lấy các số chưa được sử dụng từ cá thể bố mẹ thứ hai (thay vì lấy nguyên cả đoạn từ sau điểm cắt). Cách này đảm bảo để mỗi số chỉ xuất hiện đúng một lần (cần lưu ý là có những cách khác cho phép đảm bảo điều này). Ví dụ:

(**1 2 3 4 5** 6 7 8 9) + (4 5 3 6 8 9 7 2 1) = (**1 2 3 4 5** 6 8 9 7)

- Lai ghép đều. Trong phương pháp này, mỗi gen được sao chép từ bố hoặc từ mẹ với xác suất bằng nhau.

Ví dụ lai ghép đều:

1**11**01**0**1**1** + **11**0**11**1**0**1 = **11111111**

- Lai ghép số học. Áp dụng phép tính số học hoặc logic trên cặp bit tương ứng của bố và mẹ. Ví dụ: sử dụng phép AND

11001011 + 11011111 = 11001001

- Có thể sử dụng ba cha mẹ thay vì hai.

6.3.8 Các phương pháp tạo đột biến

Đối với cách mã hoá nhị phân, đột biến được thực hiện bằng cách đổi giá trị các bit được chọn.

Đối với mã hoá dạng hoán vị: hai số được chọn và đổi vị trí cho nhau. Ví dụ:

(1 **2** 3 4 5 6 **8** 9 7) => (1 **8** 3 4 5 6 **2** 9 7)

Đối với mã hoá dạng vec tơ các giá trị: thay đổi giá trị các gen được chọn như trong ví dụ 8 quân hậu ở trên.

6.3.9 Hiệu ứng của các thao tác

Các thao tác chọn lọc, lai ghép và đột biến có các hiệu ứng như sau khi được sử dụng riêng rẽ hoặc kết hợp với các thao tác khác.

- Nếu chỉ sử dụng thao tác chọn lọc thì thuật toán có xu hướng chọn cá thể tốt nhất từ thế hệ trước. Như vậy chỉ có thể tăng độ thích nghi trung bình và tránh lời giải ít thích nghi chứ không tạo ra lời giải tốt hơn lời giải tốt nhất của thế hệ trước.
- Kết hợp chọn lọc và lai ghép có xu hướng khiến thuật toán hội tụ ở lời giải tương đối tốt nhưng không tối ưu.
- Sử dụng một mình thao tác đột biến sinh ra chuyển động ngẫu nhiên. Nếu chỉ sử dụng một mình thao tác đột biến, thuật toán sẽ không di chuyển về hướng có lời giải tốt.
- Sử dụng chọn lọc kết hợp với đột biến tạo ra một dạng leo đồi song song ổn định hơn leo đồi thông thường.

Như vậy, giải thuật di truyền có thể coi như một biến thể mở rộng của tìm kiếm leo đồi ngẫu nhiên thực hiện song song trên một tập hợp các lời giải.

Tài liệu tham khảo

1. Russell and Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, 4th Edition. 2021
2. Từ Minh Phương. Giáo trình Nhập môn trí tuệ nhân tạo. NXB Thông tin và Truyền thông. 2016