CHƯƠNG 2
BÀI TOÁN, KHÔNG GIAN BÀI TOÁN VÀ TÌM KIẾM

Chương này trình bày một phương pháp *giải quyết vấn đề bằng tìm kiếm* (solving problems by searching). Một hệ thống giải quyết vấn đề bằng tìm kiếm sử dụng *biểu diễn nguyên tố* (atomic representation). Trong biểu diễn nguyên tố, mỗi *trạng thái* (state) của bài toán được xem như một khối nguyên tố không thể phân rã. Các giải thuật giải quyết vấn đề không truy xuất được các thành phần bên trong của trạng thái. Các hệ thống tìm kiếm sử dụng biểu diễn nâng cao hơn như *biểu diễn cấu trúc* (structured representation) thường được gọi là các *hệ lập kế hoạch* (planning systems hay planners).

Phần đầu chương trình bày phương pháp mô tả, biểu diễn *bài toán*/*vấn đề* (problems) và các *lời giải*/*giải pháp* (solutions) của nó cùng một số ví dụ minh hoạ cho các cách biểu diễn này. Phần kế tiếp mô tảmột số giải thuật tìm kiếm có thể sử dụng để giải quyết các vấn đề. Chúng ta sẽ thảo luận về các các giải thuật *tìm kiếm không có thông tin bổ sung* (uninformed search). Phần cuối chương trình bày một số điểm cần chú ý khi giải quyết vấn đề bao gồm: phân tích vấn đề, phân lập và biểu diễn tri thức nhiệm vụ cần thiết cho giải quyết vấn đề, chọn lựa kỹ thuật giải quyết vấn đề tốt nhất và áp dụng nó cho các vấn đề cụ thể.

2.1 Xác định bài toán như tìm kiếm trong không gian trạng thái

Bài toán đong dầu: một cửa hàng bán dầu có 3 bình dung tích lần lượt là 7 lít, 5 lít và 3 lít (không phân độ). Ban đầu bình 7 lít chứa đầy đầu, hai bình còn lại rỗng. Thao tác đong dầu có thể là *chuyên dầu*[1] từ một bình (có dầu) sang một bình (chưa đầy). Để xác định chính xác số dầu được chứa trong các bình, việc chuyên dầu chỉ có thể thực hiện theo kiểu *chuyên đầy*: "rót dầu từ bình A sang bình B đến khi bình B đầy" hoặc*chuyên rỗng*: "chuyên dầu từ bình A sang bình B đến khi bình A rỗng". Một người đến mua 1 lít dầu, người bán dầu phải làm thế nào để đong được đúng 1 lít dầu bán cho khách?

[1] Chuyên dầu: rót dầu từ bình này sang bình một bình khác.

Chúng ta xem xét một quá trình "phân tích, thiết kế" chương trình giải bài toán.

Trạng thái của bài toán (state) phản ánh hiện trạng của bài toán. Đối với bài toán đang xét một trạng thái có thể được biểu diễn bởi bộ 3 số (x, y, z); với x, y z lần lượt là số dầu hiện có trong các bình 7, 5 và 3 lít. Hoạtđộng chuyên dầu làm biến đổi trạng thái (x, y, z) sang trạng thái (x', y', z'). Ví dụ, thao tác chuyên dầu từ bình 7 lít sang bình 5 lít sẽ chuyển từ trạng thái (7, 0, 0) sang trạng thái (2, 5, 0).

Thông thường, để lưu vết người ta sử dụng một *con trỏ* (pointer) trỏ từ trạng thái đáng xét về *trạng thái cha* (parent state) – trạng thái sinh ra nó sau một thao tác chuyên dầu.

Như vậy, ta có thể biểu diễn trạng thái như sau:

```
typedef struct State *PState;
typedef struct State {
   int x, y, z;
   PState parent; /* con trỏ về trạng thái "cha" */
};
```

Trạng thái ban đầu (initial state): (7, 0, 0)

```
PState pStart = new State;
pStart->x = 7; pstart->y = 0; pstart->z = 0;
pStart->parent = NULL;
```

Trạng thái đích (goal state) có một trong các dạng:

(1, y, z), (x, 1, z) hay (x, y, 1)

Thủ tục kiểm tra trạng thái đích (goal test):

```
bool goalcheck(PState s) {
   if (( s->x == 1) || (s->y == 1) || (s->z == 1))
      return true;
   return false;
}
```

Các thao tác chuyên dầu:

Cxy: chuyên dầu từ bình 7 lít sang bình 5 lít. Thao tác này chuyển trạng thái (x, y, z) sang trạng thái (x', y', z')

Điều kiện: bình 7 lít phải không rỗng: x > 0, bình 5 lít phải chưa đầy hay y < 5.

Kết quả: x' = max(x + y – 5, 0), y' = min(x + y , 5), z' = z

Thao tác có thể được xem như toán tử biến đổi từ trạng thái (x, y, z) sang trạng thái (x', y', z'). Ta diễn tả phép biến đổi này bởi một hàm nhận vào một trạng thái (trạng thái bị biến đổi) và trả về trạng thái kết quả:

```
PState cxy(PState s) {
   if ((s->x > 0) && (s->y < 5)) {
      PState result = new State;
      result->x=max(s->x + s->y – 5, 0);
      result->y=min(s->x + s->y, 5) ;
      result->z = s->z;
      result->parent = s;
      return result;
   }
   return NULL;
}
```

Cxz: chuyên dầu từ bình 7 lít sang bình 3 lít. Thao tác này chuyển trạng thái (x, y, z) sang trạng thái (x', y', z').

Điều kiện: bình 7 lít phải không rỗng: x > 0, bình 3 lít phải chưa đầy hay z < 3

Kết quả: x' = max(x + z – 3, 0), y' = y, z' = min(x + z, 3)

Hàm tương ứng là:

```
PState cxz(PState s) {
   if ((s->x > 0) && (s->z<3)) {
      PState result = new State;
      result->x = max(s->x + s->z – 3, 0);
      result->y = s->y;
      result->z = min(s->x + s->z, 3);
      result->parent = s;
      return result;
   }
   return NULL;
}
```

Cyx: chuyên dầu từ bình 5 lít sang bình 7 lít. Thao tác này chuyển trạng thái (x, y, z) sang trạng thái (x', y', z')

Điều kiện: bình 5 lít phải không rỗng: y > 0

Kết quả: thu được là x' = x + y, y' = 0, z' = z (vì tổng số dầu chỉ có 7 lít)

Hàm tương ứng:

```
PState cyx(PState s) {
   if ((s->y> 0) && (s->x < 7)) {
      PState result = new State;
      result->x=s->y + s->x;
      result->y = 0;
      result->z = s->z;
      result->parent = s;
      return result;
   }
   return NULL;
}
```

Cyz: chuyên dầu từ bình 5 lít sang bình 3 lít. Thao tác này chuyển trạng thái (x, y, z) sang trạng thái (x', y', z')

Điều kiện: bình 5 lít phải không rỗng: y > 0, bình 3 lít phải chưa đầy hay x < 3

Kết quả: thu được là x' = x, y' = max(y + z – 3, 0), z' = min(y + z, 3)

Hàm tương ứng:

```
PState cyz(PState s) {
   if ((s->y > 0) && (s->z < 3)) {
      PState result = new State;
      result->x = s->x;
      result->y = max(s->y + s->z - 3, 0);
      result->z = min(s->y + s->z, 3);
      result->parent = s;
      return result;
   }
   return NULL;
}
```

Czx:chuyên dầu từ bình 3 lít sang bình 7 lít. Thao tác này chuyển trạng thái (x, y, z) sang trạng thái (x', y', z')

Điều kiện: bình 3 lít phải không rỗng: z > 0

Kết quả: thu được là x' = x + z, y' = y, z' = 0

Hàm tương ứng là:

```
PState czx(PState s) {
   if (s->z > 0) {
      PState result = new State;
      result->x = s->x + s->z;
      result->y = y;
      result->z = 0;
      result->parent = s;
      return result;
   }
   return NULL;
}
```

Czy: chuyên dầu từ bình 3 lít sang bình 5 lít. Thao tác này chuyển trạng thái (x, y, z) sang trạng thái (x', y', z')

Điều kiện: bình 3 lít phải không rỗng: z > 0, bình 5 lít phải chưa đầy hay y < 5

Kết quả: thu được là x' = x, y' = min(y + z, 5), z' = max(y + z – 5, 0)

Hàm tương ứng:

```
PState czy(PState s) {
   if((s->z > 0)&&(s->y < 5)) {
      PStateresult = new State;
      result->x = s->x;
      result->y = min(s->y + s->z, 5);
      result->z = max(s->y + s->z - 5, 0);
      result->parent = s;
      return result;
   }
   return NULL;
}
```

Quá trình "phân tích, thiết kế" trên dựa trên phương pháp tiếp cận tổng quát giải quyết vấn đề (phương pháp tiếp cận có thể áp dụng lên một lớp rất rộng các bài toán):

1. Biểu diễn bài toán trong không gian các trạng thái. Mỗi trạng thái phản ánh một "cấu hình" của thế giới bài toán.
2. Xác định các trạng thái khởi đầu, các trạng thái đích.
3. Xác định các thao tác cho phép và biểu diễn chúng: các thao tác được biểu diễn bởi các toán tử biến đổi trạng thái (dưới góc nhìn lập trình, một toán tử thường được biểu diễn như một thủ tục/một hàm).

2.1.1 Định nghĩa hình thức không gian trạng thái

Không gian trạng thái (state space) là một bộ bốn (N, S, GD, Op), trong đó N là tập các trạng thái (được xem như tập các nút dưới góc nhìn đồ thị), S là tập không rỗng các trạng thái ban đầu, GD (Goal description – Bộ mô tả mục tiêu) là tập không rỗng các trạng thái đích và Op là tập các toán tử biến đổi trạng thái (dưới góc nhìn đồ thị, mỗi toán tử được xem như một cung từ trạng thái trước biến đổi đến trạng thái sau biến đổi).

Đường đi lời giải (solution path) là một đường đi trong đồ thị trạng thái dẫn từ một trạng thái bắt đầu đến một trạng thái thỏa mãn mục tiêu/trạng thái đích (tương đương với một dãy các toán tử nếu áp dụng nó bắt đầu từ một trạng thái bắt đầu, kết quả của dãy áp dụng là một trạng thái đích).

Trong nhiều trường hợp, tập các trạng thái đích không thể biểu diễn bằng cách liệt kê mà chỉ có thể biểu diễn bằng một tập các tính chất. Trạng thái / đường đi thoả mãn các tính chất đó là trạng thái đích / đường đi tới đích.

Một toán tử có thể không áp dụng được trên một số trạng thái. Nó chỉ có thể áp dụng lên các trạng thái thoả mãn một số điều kiện nhất định. Toán tử thường được viết dưới dạng:

<điều kiện> $\rightarrow$ <hành động>

với ý nghĩa là nếu trạng thái làm thoả mãn <điều kiện> thì <hành động> có thể được thực thi.

Khi kết quả hành động được xác định tường minh, toán tử được viết:

<điều kiện> → <kết quả của hành động>

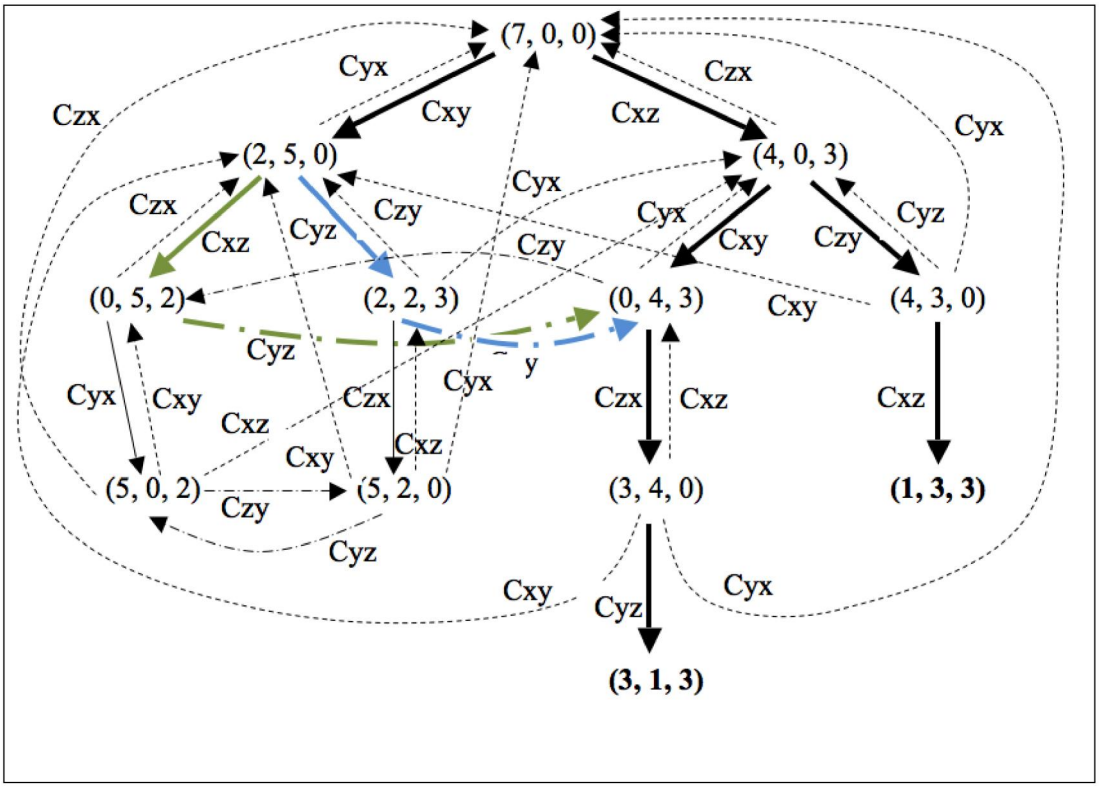
Trạng thái s làm thoả mãn điều kiện của toán tử o, toán tử o được gọi là có thể áp dụng lên s.

Nếu tồn tại toán tử có thể áp dụng lên s_1 và cho ra kết quả là s_2, trạng thái s_1 được gọi là có thể đi trực tiếp đến s_2, s_2 được gọi là có thể đạt tới trực tiếp từ s_1

Nếu tồn tại dãy toán tử khi áp dụng bắt đầu từ s_1 cho ra kết quả là s_2, s_1 được gọi là có thể đi tới s_2, s_2 được gọi là có thể đạt tới từ s_1

Khi bài toán đã được biểu diễn bởi không gian trạng thái, câu hỏi đặt ra là "làm thế nào giải quyết bài toán thông qua cách biểu diễn này?"

Mục tiêu của bài toán có thể là tìm một đường đi lời giải hoặc tìm trạng thái đích thoả mãn các điều kiện bổ sung. Điều này dẫn đến việc tìm kiếm trong đồ thị trạng thái một đường đi lời giải / một trạng thái đích mong ước.



Hình 2.1: Đồ thị trạng thái của bài toán đong dầu.

2.1.2 Một số ví dụ biểu diễn bài toán trong không gian trạng thái

Ví dụ 1: Xét bài toán 15-puzzle: một lưới 4×4 ô vuông, 15 ô chứa các số nguyên 1, 2, …, 15 (ô số), ô còn lại trống (ô trống). Quy tắc chơi: đổi chỗ một ô số với ô trống kề cạnh với nó. Xuất phát từ một cấu hình đã cho, áp dụng quy tắc chơi để đạt tới một cấu hình cho trước.

Giả sử cấu hình xuất phát là:

11	14	4	7
10	6		5
1	2	13	15
9	12	8	3

cấu hình đích là:

1	2	3	4
12	13	14	5
11		15	6
10	9	8	7

Biểu diễn "bàn cờ"

Bàn cờ có thể biểu diễn như một mảng 4×4, các phần tử của nó chứa các số từ 0 đến 15. Phần tử chứa giá trị 0 biểu diễn ô trống. Trong cách biểu diễn này, cấu hình xuất phát và cấu hình đích khá rõ ràng và trực tiếp.

Biểu diễn quy tắc chơi

Ô trống kề cạnh với 4 ô số khi nó nằm bên trong bàn cờ, kề cạnh với 3 ô số nếu nó nằm ở cạnh bàn cờ (nhưng không ở góc) và kề cạnh với 2 ô số nếu nó nằm ở góc. Như vậy quy tắc chơi có thể biểu diễn bởi bốn phép di chuyển, ta gọi là:

- UP : đổi chố ô trống với ô bên trên nó (đưa ô trống lên trên)
- DOWN : đổi chỗ ô trống với ô bên dưới nó

- LEFT : đổi chỗ ô trống với ô bên trái nó
- RIGHT : đổi chỗ ô trống với ô bên phải nó

Chú ý rằng, các phép di chuyển này không phải lúc nào cũng có thể áp dụng vào một cấu hình, ví dụ nếu cấu hình có ô trống nằm ở dòng đầu tiên, phép di chuyển UP không áp dụng được. Ta tinh chế các phép dịch chuyển dưới dạng hàm trong giả ngôn ngữ C/C++,

Giả sử ta có các hàm:

- `row(c)` trả lại chỉ số của dòng chứa ô trống trong cấu hình c
- `col(c)` trả lại chỉ số của cột chứa ô trống trong cấu hình c

Các phép di chuyển có thể biểu diễn như sau:

```
UP(c): if(row(c) > 0) {
     int cc[4][4], r=row(c), c=col(c);
     for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
           cc[i][j] = c[i][j];
     temp = cc[r][c];
     cc[r][c] = cc[r-1][c];
     cc[r-1][c] = temp;
   return cc;
}
```

```
DOWN(c): if(row(c) < 3) {
     int cc[4][4], r=row(c), c=col(c);
     for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
           cc[i][j] = c[i][j];
     temp = cc[r][c];
     cc[r][c] = cc[r+1][c];
     cc[r+1][c] = temp;
   return cc;
}
```

```
LEFT(c): if(col(c) > 0) {
     int cc[4][4], r=row(c), c=col(c);
     for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
           cc[i][j] = c[i][j];
     temp = cc[r][c];
     cc[row][c] = cc[r][c-1];
     cc[r][c-1] = temp;
   return cc;
}
```

```
RIGHT(c): if(col(c) < 3) {
     int cc[4][4], r=row(c), c=col(c);
     for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
           cc[i][j] = c[i][j];
     temp = cc[r][c];
     cc[r][c] = cc[r][c+1];
     cc[r][c+1] = temp;
   return cc;
}
```

Ví dụ 2: Trong phòng có một con khỉ (đang thèm ăn), một quả chuối được treo ở trên trần, một cái bàn, một cái gậy. Con khỉ muốn lấy được quả chuối, nó phải đứng trên cái bàn ở vị trí quả chuối, cầm cây gậy chọc quả chuối cho rơi và nhặt ăn. Con khỉ có thể thực hiện các thao tác: đi từ một vị trí đến một vị trí, đẩy cái bàn từ một vị trí đến một vị trí, nhặt một vật nhỏ (cây gậy, quả chuối) lên với điều kiện trên tay không cầm vật gì, bỏ vật nhỏ đang cầm trên tay xuống, leo lên bàn, leo xuống bàn, chọc quả chuối cho rơi.

Ta sử dụng logic vị từ để biểu diễn các trạng thái. Trước tiên ta định nghĩa các vị từ cơ sở cần thiết cho việc mô tả bài toán. Một trạng thái khi đó có thể được mô tả bằng hội của các vị từ cơ sở.

- at(X, Y) = "X ở vị trí Y"
- on(X, Y) = "X ở trên Y"
- armempty = "tay con khỉ không cầm gì"
- holding(X) = "con khỉ cầm vật X"
- large(X) = "X là vật lớn"
- small(X) = "X là vật nhỏ"

Ta sẽ sử dụng ký hiệu dấu phẩy ',' thay cho ký hiệu $\wedge$

Tráng thái ban đầu: "con khỉ, cái bàn, cây gậy, quả chuối ở vị trí tương ứng của chúng, con khỉ ở trên sàn, cây gậy ở trên sàn, cái bàn ở trên sàn, quả chuối ở trên trần, tay con khỉ không cầm gì":

```
at(khỉ, k), at(bàn, b), at(gậy, g), at(chuối, c), on(khỉ,
sàn), on(gậy, sàn), on(bàn, sàn), on(chuối, trần),
large(bàn), small(gậy), small(chuối), armempty.
```

Ta xét trạng thái kết thúc là trạng thái con khi cầm quả chuối, các mô tả khác không quan trọng:

```
…,holding(chuối), …
```

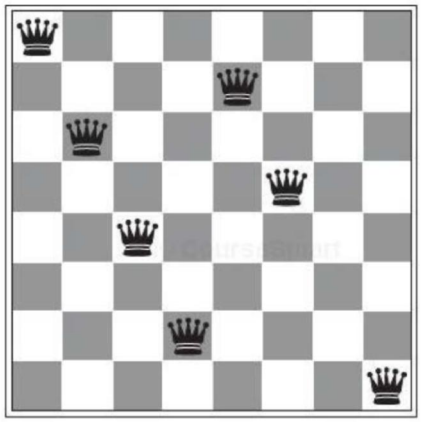
Bảng 2.1 mô tả các phép toán với tiền điều kiện và kết quả. Quy ước, trong các cột kết quả chỉ ghi các thuộc tính bị thay đổi, các thuộc tính không đươc nhắc lại là các thuộc tính không thay đổi.

Bảng 2.1: Các toán tử biến đổi trạng thái trong bài toán con khỉ và quả chuối.

Tên toán tử	**Giải thích**	**Điều kiện**	**Kết quả**
walk(p)	Con khỉ đi đến vị trí p	on(khỉ, sàn)	at(khỉ, p)
push(X, p)	Con khỉ đẩy vật lớn X đến vị trí p	*at(khỉ, V)*, *at(X, V)*, large(X), armempty	at(khỉ, p), at(X, p)
pick-up(X)	Con khỉ nhặt lên vật nhỏ X	at(khỉ, V), at(X, V), small(X), *armempty*	holding(X)
put-down(X)	Con khỉ đặt vật nhỏ X xuống	holding(X)	armempty
climb-up	Con khỉ leo lên bàn	at(khỉ, V), at(bàn, V), *on(khỉ, sàn)*	on(khỉ, bàn)
climb-down	Con khỉ leo xuống bàn	on(khỉ, bàn)	on(khỉ, sàn)
thrust	Con khỉ chọc quả chuối	at(bàn, V), at(chuối, V), on(khỉ, bàn), holding(gậy), *on(chuối, trần)*	on(chuối, sàn)

Ví dụ 3. Bài toán 8 quân hậu: hãy tìm cách đặt 8 quân hậu lên bàn cờ vua sao cho không có quân nào bị khống chế.

Hình 2.2 minh hoạ một phần (7 quân hậu) lời giải của bài toán này.



Hình 2.2: Bài toán 8 quân hậu.

Có nhiều cách để biểu diễn bài toán 8 quân hậu.

Cách 1:

Trạng thái:

- Mỗi trạng thái là vị trí của kquân hậu được đặt trên bàn cờ, mỗi quân hậu được đặt ở một ô (k = 0, 1, …, 8)

Trạng thái ban đầu:

- Bàn cờ không có quân hậu nào.

Phép toán:

- Chọn 1 ô trống bất kỳ trên bàn cờ và đặt thêm một quân hậu lên bàn cờ

Hàm kiểm tra mục tiêu:

- Kiểm tra xem có đủ 8 quân hậu trên bàn cờ chưa và có quân hậu nào bị khống chế không.

Rõ ràng cách này sẽ có nhiều bất lợi vì số đường đi phải xem xét lên đến !

Nhận xét:

- Do quân hậu sẽ khống chế hàng, cột và 2 đường chéo, nên mỗi cột/hàng có đúng một quân hậu. Một cột đã có quân hậu rồi sẽ không đặt quân hậu mới vào cột đó nữa.

Với nhận xét trên, ta có thể cải tiến một chút mô hình.

Cách 2:

Trạng thái:

- Mỗi trạng thái là vị trí của k quân hậu được đặt trên k cột đầu tiên củabàn cờ, mỗi quân hậu trên 1 cột, k = 0, 1, …, 8. Các quân hậu này không khống chế lẫn nhau.

Trạng thái ban đầu:

- Bàn cờ không có quân hậu nào

Phép toán:

- Chọn 1 vị trí trên cột trống đầu tiên (xét từ trái sang phải) sao cho nó không cùng hàng và đường chéo với các hậu được đặt trước đó và đặt thêm một quân hậu lên vị trí đó

Hàm kiểm tra mục tiêu:

- Kiểm tra xem có đủ 8 quân hậu trên bàn cờ chưa.

Tổng số đường đi phải xem xét giảm từ 1.8×10^{14} xuống chỉ còn 2.057.

Hàm kiểm tra mục tiêu cũng đơn giản hơn: ta chỉ cần kiểm tra xem đủ 8 quân hậu chưa thôi vì mỗi khi đặt quân hậu mới ta đã kiểm tra nó không bị các quân hậu trước đó khống chế rồi.

Cách 3:

Trạng thái:

- Mỗi trạng thái là vị trí của 8 quân hậu được đặt trên bàn cờ, mỗi quân hậu trên 1 cột.

Phép toán:

- Chọn một quân hậu bị khống chế và di chuyển đến một vị trí khác trong cùng cột sao cho nó không bị khống chế nữa.

Hàm kiểm tra mục tiêu:

- Kiểm tra xem có quân hậu nào còn bị khống chế không.

Ví dụ 4: Bài toán tìm đường đi từ thành phố Arad đến thành phốBucharest, ở Rumani. Bài toán này được trích từ quyển "Artificial Intelligence: A Modern Approach" của [Russel & Norvig, 2009].Hình 2.3mô tả bản đồ đơn giản của những con đường nối các thành phố của Rumani. Các con số trên mỗi con đường mô tả chiều dài của con đường (tính bằng km). Giả sử ta đang ở thành phố Arad và muốn tìm đường đi để đến thành phố Bucharest.

Trạng thái:

- Các thành phố của Rumani. Ta có thể sử dụng tên của các thành phố để mô tả trạng thái. Ta cũng có thể sử dụng vị từ `in(X)` để mô tả ta đang ở thành phố X.

Trạng thái ban đầu:

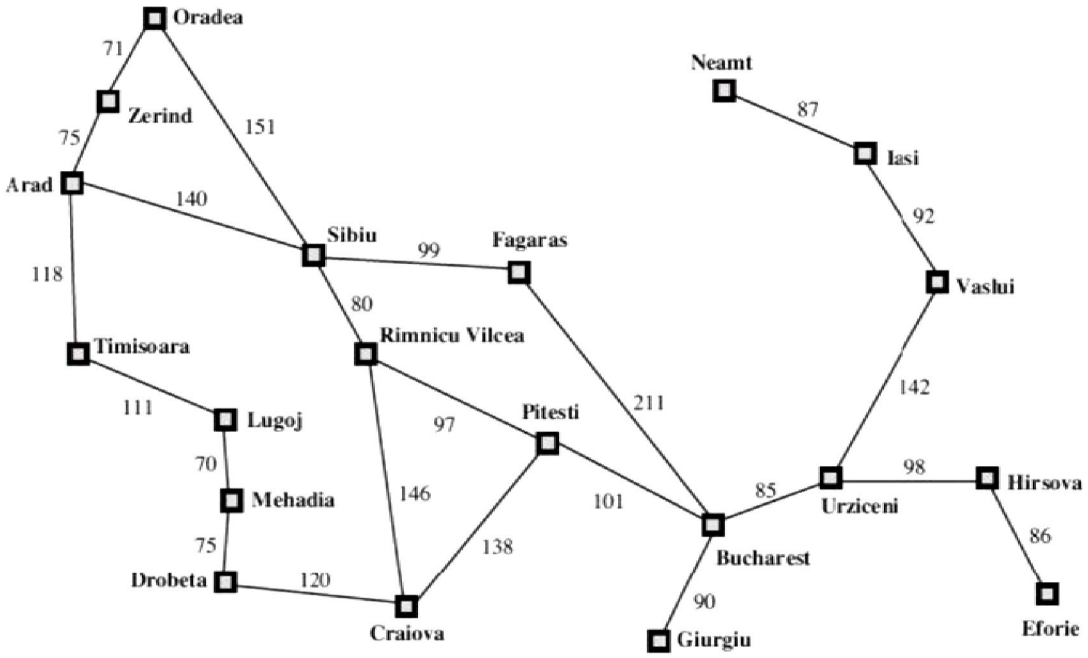
- Ta đang ở Arad: Arad hay `in(Arad)` nếu sử dụng vị từ để mô tả trạng thái.

Phép toán:

- Đi từ thành phố hiện tại đến thành phố khác có đường nối trực tiếp. Nếu có đường đi trực tiếp từ thành phố X đến thành phố khi, thì ta có phép toánbiến đổi trạng thái X thành trạng thái Y. Ví dụ nếu X = Arad, ta có thể đi đến 1 trong 3 thành phố: Zerind, Sibiu hay Timisoara.

Hàm kiểm tra mục tiêu:

- Thành phố đang xét có phải là Bucharest không

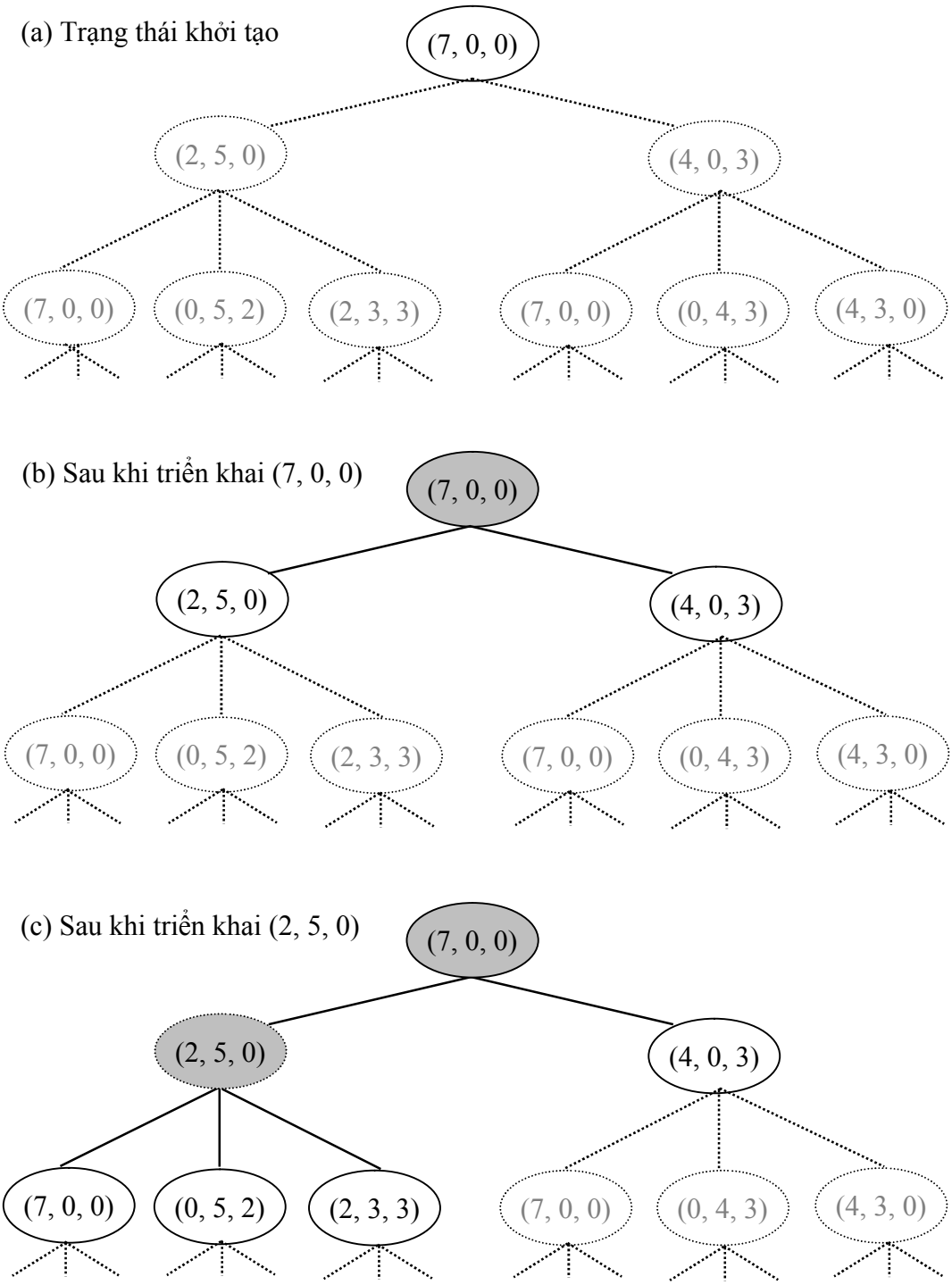


Hình 2.3: Bản đồ các con đường nối các thành phố của Rumani.

2.2 Các kỹ thuật tìm kiếm

Sau khi đã mô hình hoá bài toán, vấn đề kế tiếp là cần phải giải nó. Một *lời giải* (solution) là một *chuỗi các hành động* (action sequence), vì thế các *giải thuật tìm kiếm* làm việc bằng cách xem xét các chuỗi hành động khả dĩ. Các chuỗi các hành động khả dĩ bắt đầu từ trạng thái ban đầu tạo nên một *cây tìm kiếm* (search tree) có nút gốc là trạng thái ban đầu; nhánh là các hành động và *nút* (nodes) tương ứng với các trạng thái trong không gian trạng thái của vấn đề.

Hình 2.4 minh hoạ một số bước trong quá trình tìm kiếm lời giải trong bài toán đong dầu. Nút gốc của cây tương ứng với trạng thái bắt đầu, (7,0,0). Bước đầu tiên là kiểm tra xem trạng thái hiện tại có phải là trạng thái kết thúc chưa. Sau đó, ta xem xét các hành động có thể thực hiện được lên trạng thái hiện tại, và *sinh ra* (generate) một tập các trạng thái mới. trong trường hợp này, ta có 2 nhánh xuất phát từ *nút cha* (parent node) (7,0,0) dẫn đến 2 *nút con* (child node): (2,5,0) và (4,0,3). Bây giờ đến lượt ta phải chọn nút con nào để xem xét xem có thể sinh ra các nút "cháu" nào nữa không.



Hình 2.4: Cây tìm kiếm được sinh ra trong quá trình tìm kiếm lời giải của bài toán đong dầu.

Vấn đề chọn nút kế tiếp để xem xét là vấn đề quan trọng nhất của các giải thuật tìm kiếm: phải chọn nút nào để xem xét trước. Giả sử ta chọn (2,5,0) để xem xét trước. Ta sẽ kiểm tra xem nó có phải là trạng thái kết

thúc chưa và *triển khai/mở rộng* (expand) nó để được các nút mới: (7, 0, 0), (0, 5, 2), và (2, 3, 3).Sau đó, ta có thể chọn 1 trong 4 nút này để xem xét hoặc ta cũng có thể để 3 nút đó lại và quay chọn (4, 0, 3). Tất cả 4 nút này đều là *nút lá* (leaf node) – nút không có nút con. Tập các nút lá sẵn sàng để triển khai tại một thời điểm nào đó được gọi là *đường biên* (frontier) của cây tìm kiếm. Một số tác giả khác gọi là *danh sách mở* (open list). Tuy nhiên cách gọi này không chính xác cả về mặt địa lý và ý nghĩa vì ta có thể sử dụng cấu trúc dữ liệu khác hiệu quả hơn danh sách (list). Trong hình2.4đường biên của cây là các nút nền trắng viền đen.

Quy trình triển khai các nút trên đường biên cứ tiếp tục cho đến khi tìm thấy một lời giải hoặc không thể triển khai được nữa. Giải thuật tìm kiếm tổng quát `treeSearch` được trình bày trong hình 2.5. Các giải thuật tìm kiếm đều có chung cấu trúc này. Khác biệt chính giữa các giải thuật là cách chọn nút kế tiếp để triển khai – vấn đề này gọi là *chiến lược tìm kiếm* (search strategy).

```
PNode treeSearch(PState initial_state) {
   //Nút gốc biểu diễn trạng thái bắt đầu
   PNode root = new Node;
   root->state = initial_state;
   root->parent = NULL;
   //Khởi tạo đường biên chứa nút gốc
   insert(root, frontier);
   while (!frontier.isEmpty()) {
      //Lấy 1 nút từ đường biên và loại bỏ nó
      //ra khỏi đường biên
      PNode node = pop(frontier);
      //Nếu node là một nút mục tiêu, trả về node
      //Lần vết từ node theo con trỏ parent để tìm
      //đường đi từ gốc đến node.
      if (node là nút mục tiêu)
         return node;
      for each (child là nút con của node)
         if (child->state không thuộc frontier) {
            child->parent = node;
            insert(child, frontier);
         }
   }
   //Thất bại, không tìm thấy lời giải
   return NULL;
}
```

Hình 2.5: Giải thuật treeSearch

Nếu tinh mắt một chút ta có thể nhận thấy có một chỗ đặc biệt của cây tìm kiếm trong hình 2.2: nó gồm cả đường đi từ (7, 0, 0) đến (2, 5, 0) và trở lại (7, 0, 0). Ta nói rằng trạng thái (7, 0, 0) là *trạng thái bị lặp lại* (repeated state) trong cây tìm kiếm, và sinh ra *đường đi lặp* (loopy path). Xem xét các đường đi lặp như thế có nghĩa là cây tìm kiếm đầy đủ sẽ *vô hạn* (infinite) vì không có ràng buộc nào về số lần lặp. Mặt khác, trong ví dụ này không gian trạng thái ít hơn 20 trạng thái. Như sẽ phân tích ở phần sau, các vòng lặp có thể làm cho các giải thuật thất bại (lặp vô tận), làm một số vấn đề có thể giải được trở nên không giải được! May thay, ta không cần phải xem xét các đường đi lặp. Ta có thể dựa trên một nhận xét trực quan như sau: *vì chi phí đường đi có tính cộng (additive) và chi*

phí từng bước không âm nên một đường đi lặp không bao giờ tốt hơn một đường đi đã được khử lặp.

Các đường đi lặp là một trường hợp đặc biệt của một khái niệm tổng quát hơn: *đường đi thừa* (redundant paths). Đường đi thừa tồn tại khi có nhiều hơn một cách để đi từ một trạng thái này đến một trạng thái khác. Xem xét đường đi (7, 0, 0) – (4, 0, 3) và (7, 0, 0) – (2, 5, 0) – (2, 2 3) – (4, 0, 3). Rõ ràng là đường đi thứ hai là thừa, đường đi dài nhất để đến cùng một trạng thái (trường hợp xấu nhất). Nếu ta chỉ quan tâm đến việc đi đến trạng thái đích, không có lý do gì phải giữ lại nhiều hơn một đường đi.

Trong một số trường hợp, ta có thể định nghĩa vấn đề để bản thân nó đã loại bỏ được các đường đi thừa. Ví dụ, Nếu ta mô hình hoá bài toán 8 quân hậu để mỗi quân hậu có thể được đặt trên bất cứ cột nào, thì mỗi trạng thái n quân hậu có thể có đến $n!$ đường đi khác nhau để đến nó. Nhưng nếu ta mô hình hoá bài toán sao cho mỗi quân hậu mới chỉ có thể đặt trên cột trống đầu tiên thì chỉ có duy nhất một đường đi đến một trạng thái bất kỳ.

Trong các trường hợp khác, vấn đề đường đi thừa là không thể tránh khỏi. Điều này bao gồm cả các vấn đề trong đó các hành động đều *khả đảo* (reversible), ví dụ như bài toán tìm đường đi trên bản đồ và bài toán 15-puzzles. Bài toán tìm đường đi trên một lưới chữ nhật là một ví dụ quan trọng trong các trò chơi máy tính. Trong một lưới như thế, mỗi trạng thái có 4 trạng thái con, vì thế một cây có độ sâu d gồm luôn cả các trạng thái bị lặp lại có 4^d nút lá, nhưng chỉ có $2d^2$ trạng thái phân biệt trong d bước. Với $d = 20$, điều này có nghĩa là khoảng 1 tỷ nút nhưng chỉ có 800 trạng thái khác nhau. Vì thế, đi theo các đường đi thừa có thể làm một vấn đề *có thể giải được* (tractable) trở nên *không thể giải được* (untractable)! Điều này vẫn còn đúng ngay cả khi giải thuật biết cách tránh các vòng lặp vô tận.

Nói một cách khác, nếu giải thuật không lưu lại lịch sử thì có thể sinh ra đường đi lặp. Cách tránh triển khai các đường đi thừa là phải nhớ (lưu trữ) những nơi đã đi qua. Để làm điều này, ta tăng cường thêm cho giải thuật `treeSearch` một cấu trúc dữ liệu gọi là *tập các nút đã triển khai* (explored set, cũng được gọi là *danh sách đóng* – closed list) lưu trữ

các nút đã triển khai[1]. Khi một nút mới sinh ra, nếu trạng thái của nó đã có mặt trong tập các trạng thái đã triển khai hoặc nằm trên đường biên, ta có thể bỏ qua thay vì thêm nó vào đường biên. Giải thuật mới, `graphSearch`, được trình bày trong hình 2.4.

Rõ ràng, cây tìm kiếm xây dựng bằng giải thuật `graphSearch` chứa nhiều nhất một bản sao của mỗi trạng thái. Giải thuật này có một tính chất thú vị: ***đường biên*** *phân chia đồ thị không gian trạng thái thành 2 vùng:* ***đã khám phá*** *(explored) và* ***chưa khám phá*** *(unexplored), vì thế mỗi đường đi từ trạng thái bắt đầu đến một trạng thái trong vùng chưa khám phá phải đi qua một trạng thái trên đường biên*. Mỗi bước lặp trong vòng lặp **`while`** chuyển một trạng thái từ đường biên sang vùng đã phám phá và chuyển một số trạng thái từ vùng chưa khám phá sang đường biên. Ta có thể thấy rằng giải thuật xem xét các trạng thái một cách có hệ thống trong không gian trạng thái cho đến khi tìm được lời giải.

[1] Thực chất ta chỉ cần lưu các *trạng thái* của các nút đã triển khai.

```
PNode graphSearch(PState init_state) {
   //Nút gốc biểu diễn trạng thái ban đầu
   PNode root = new Node;
   root->state = init_state;
   root->parent = NULL;
   //Khởi tạo đường biên chứa nút gốc
   insert(root, frontier);
   //Khởi tạo tập nút đã triển khai bằng rỗng
   clear(explored);
   while (!empty(frontier)) {
      //Lấy 1 nút từ đường biên và loại bỏ nó
      //ra khỏi đường biên
      Node* node = frontier.pop();
      //Nếu node là một nút mục tiêu, trả về node
      //Lần vết từ node theo con trỏ parent để tìm
      //đường đi từ gốc đến node.
      if (node là nút mục tiêu)
         return node;
      //node là nút đã triển khai, đưa nó vào explored
      insert(node, explored);
      for (child là nút con của node)
         //Đưa child vào đường biên nếu nó
         //không nằm trên đường biên và cũng
         //không nằm trongtập đã triển khai
         if (child->state không thuộc frontier và
            child->state không thuộc explored) {
            child->parent = node;
            insert(child, frontier);
         }
   }
   //Thất bại, không tìm thấy lời giải
   return NULL;
}
```

Hình 2.6: Giải thuật graphSearch

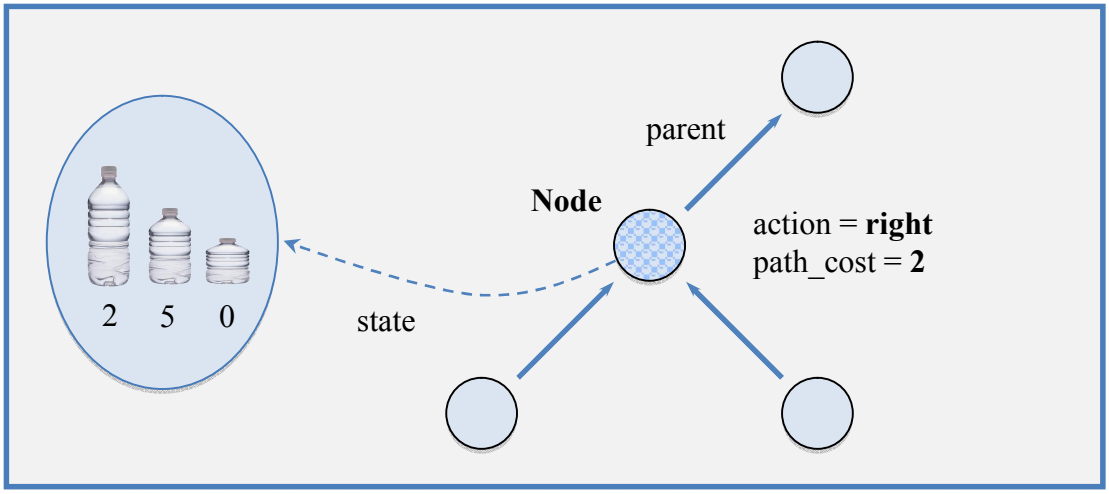
2.2.1 Cấu trúc dữ liệu cho các giải thuật tìm kiếm

Các giải thuật tìm kiếm cần một cấu trúc dữ liệu để lưu vết cây tìm kiếm đang được xây dựng.

Mỗi nút **n** trên cây tìm kiếm gồm 4 thành phần:

- `state`: trạng thái trong không gian trạng thái tương ứng với nút này;
- `parent`: nút cha (nút chứa trạng thái sinh ra trạng thái của nút n);
- `action`: hành động áp dụng lên trạng thái của nút cha để sinh ra n;
- `path_cost`: chi phí, thường được ký hiệu g(n), chi phí đường đi từ nút gốc đến nút n.

Cấu trúc của một nút được mô tả trong hình 2.7. Chú ý rằng nhờ vào con trỏ trỏ về nút cha mà ta có thể lần ngược để tìm ra đường đi của lời giải. Cho trước các thuộc tính của một nút cha, ta dễ dàng tính được các thuộc tính của nút con. Cho đến lúc này ta vẫn chưa phân biệt rõ ràng giữa *nút* và *trạng thái*. Nhưng khi viết giải thuật chi tiết ta cần phải phân biệt rõ hai khái niệm này. Nút là một cấu trúc dữ liệu dùng để biểu diễn cây tìm kiếm. Trạng thái tương ứng với một *cấu hình* (configuration) của bài toán. Vì vậy, nút nằm trên *đường đi* (path) còn trạng thái thì không. Hơn nữa hai nút khác nhau có thể chứa hai trạng thái y hệt nhau nếu trạng thái đó được sinh ra từ hai đường đi khác nhau.



Hình 2.7: Cấu trúc dữ liệu cho nút trong cây tìm kiếm.

Cấu trúc dữ liệu của nút được khai báo tổng quát như trong hình 2.8.

```
typedef struct Node* PNode;
typedef struct {
   PState state;
   PNode parent;
   Action action;
   int path_cost;
} Node;
```

Hình 2.8: Khai báo cấu trúc dữ liệu cho nút trong cây tìm kiếm.

Hàm `childNode` nhận vào hai tham số – nút **n** và hành động **action** – và trả về một nút con của **n** (xem Hình 2.9). Hàm `result` nhận vào một trạng thái và một hành động, trả về trạng thái mới được sinh ra nếu áp dụng hành động action lên trạng thái hiện tại. Hàm `stepCost` trả về chi phí của hành động **action** áp dụng lên trạng thái hiện tại.

```
PNode childNode(PNode parent, Action action) {
   PNode n = new Node;
   n->state= result(parent->state, action);
   if (n->state == NULL)
      return NULL;
   n->parent = parent;
   n->action = action;
   n->path_cost = parent->path_cost +
          stepCost(parent->state, action);
   return n;
}
```

Hình 2.9: Hàm childNode sinh ra nút con từ nút cha và hành động.

Trở lại bài toán đong dầu, mỗi trạng thái là bộ 3 (x, y, z) với x, y, z: là số dầu hiện có trong các bình 7, 5 và 3 lít. Ta có thể khai báo cấu trúc dữ liệu cho trạng thái và nút cùng với các hàm cần thiết như trong hình 2.10.

```
typedef struct {
   int x, y, z;
} State;
typedef State* PState;

typedef struct Node* PNode;

typedef int Action;
typedef struct {
   PState state;
   PNode parent;
   Action action;
   int path_cost;
} Node;
```

Hình 2.10: Khai báo cấu trúc dữ liệu cho bài toán đong dầu.

Con trỏ trỏ đến nút cha (parent) của một nút được lưu trữ trong cấu trúc `Node` thay vì cấu trúc `State` như trước đây. Mỗi hành động chuyên dầu được mã hoá bằng một số nguyên (trường action trong cấu trúc State).Mã và ý nghĩa các hành động được cho trong Bảng 2.1.

Bảng 2.2: Các hành động chuyên dầu trong bài toán đong dầu.

Mã hành động	**Tên hành động**	**Ý nghĩa**
1	cxy	Chuyên dầu từ bình 7 lít sang bình 5 lít
2	cxz	Chuyên dầu từ bình 7 lít sang bình 3 lít
3	cyx	Chuyên dầu từ bình 5 lít sang bình 7 lít
4	cyz	Chuyên dầu từ bình 5 lít sang bình 3 lít
5	czx	Chuyên dầu từ bình 3 lít sang bình 7 lít
6	czy	Chuyên dầu từ bình 3 lít sang bình 5 lít

Hàm `result` áp dụng hành động action lên trạng thái hiện tại để sinh ra một trạng thái mới hoặc `NULL` nếu không thể áp dụng.

```
PState result(PState state, Action action) {
   switch (action) {
      case 1:
         return cxy(state);
      case 2:
         return cxz(state);
      case 3:
         return cyx(state);
      case 4:
         return cyz(state);
      case 5:
         return czx(state);
      case 6:
         return czy(state);
      default:
         return NULL;
   }
}
```

Hàm `stepCost` luôn trả về 1.

```
int stepCost(PState state, Action action) {
   return 1;
}
```

Giờ đây khi có nút, ta cần nơi để lưu trữ chúng. Đường biên cần phải được lưu trữ sao cho giải thuật tìm kiếm có thể dễ dàng chọn nút kế tiếp để triển khai tuỳ thuộc vào chiến lược tìm kiếm được sử dụng. Cấu trúc dữ liệu thường dùng để lưu trữ đường biên là *hàng đợi* (queue). Các phép toán trên hàng đợi bao gồm:

- `empty(queue)`: kiểm tra hàng đợi rỗng
- `pop(queue)`: loại bỏ phần tử đầu tiên trong hàng đợi và trả về phần tử đó.
- `insert(element, queue)`: thêm phần tử `element` vào hàng đợi `queue` và trả về hàng đợi sau khi đã thêm phần tử.

Các hàng đợi có thể được phân biệt bằng *thứ tự lưu trữ* các nút mới. Ba kiểu hàng đợi thường gặp là: FIFO (vào trước ra trước), LIFO (vào sau ra trước) và *hàng đợi ưu tiên* (priority queue): lấy phần tử có độ ưu tiên cao nhất ra trước theo một hàm ưu tiên nào đó.

Tập hợp các nút đã khám phá/triển khai có thể được cài đặt dưới dạng một bảng băm để giúp kiểm tra trạng thái lặp một cách hiệu quả. Nếu cài đặt tốt, độ phức tạp thời gian của phép toán xen và tìm kiếm một nút (thực ra là tìm trạng thái) có thể là hằng số (không cần quan tâm đến số lượng trạng thái đã được khám phá).

2.2.2 Đánh giá hiệu quả giải quyết vấn đề

Trước khi đi vào thiết kế các giải thuật tìm kiếm cụ thể, ta cần xem xét các tiêu chí có thể dùng để lựa chọn giải thuật. Có thể đánh giá hiệu quả của giải thuật theo 4 tiêu chí sau:

- *Tính đầy đủ* (competeness): giải thuật có đảm bảo tìm kiếm được lời giải nếu nó tồn tại không?
- *Tính tối ưu* (Optimality): chiến lược tìm kiếm có cho ra lời giải tối ưu không?
- *Độ phức tạp thời gian* (Time complexity): mất bao lâu để tìm ra lời giải?
- *Độ phức tạp không gian* (Space complexity): cần bao nhiêu bộ nhớ để thực hiện quá trình tìm kiếm?

Độ phức tạp thời gian và không gian luôn có thể tính được theo một số độ đo về độ khó của bài toán. Trong khoa học máy tính, độ đo tiêu biểu là kích thước của đồ thị không gian trạng thái, |V| + |E|, trong đó V là tập các nút và E là tâp các cạnh/cung (edges/links) của đồ thị. Độ đo này phù hợp khi đồ thị là một cấu trúc dữ liệu tường minh được xem như đầu vào của giải thuật tìm kiếm. Trong trí tuệ nhân tạo, đồ thị thường được biểu diễn không tường minh (implicitly) bằng trạng thái bắt đầu, các hành động và mô hình biến đổi. Các đồ thị thường vô hạn. Vì các lý do này mà độ phức tạp được thể hiện dưới dạng 3 tham số: b – *hệ số phân nhánh* (branching factor) hay số con tối đa của các nút, d – *độ sâu của nút mục tiêu cạn nhất* (số bước đi từ gốc đến nút mục tiêu cạn nhất) và m – chiều dài lớn nhất của các đường đi trong không gian trạng thái. Độ phức tạp thời gian thường được đo theo số nút được sinh ra trong quá

trình tìm kiếm trong khi độ phức tạp không gian được đo theo số lượng nút được lưu trữ trong bộ nhớ. Đa số các trường hợp, ta mô tả độ phức tạp thời gian và không gian cho phiên bản tìm kiếm trên cây (treeSearch). Đối với phiên bản tìm kiếm trên đồ thị (graphSearch), câu trả lời còn phụ thuộc vào các đường đi thừa trong không gian trạng thái như thế nào nữa.

Để đánh giá tính hiệu quả của một giải thuật tìm kiếm, chúng ta có thể chỉ xem xét *chi phí tìm kiếm* (search cost) – phụ thuộc vào độ phức tạp thời gian nhưng cũng có thể phụ thuộc vào bộ nhớ sử dụng – hoặc ta có thể dùng *chi phí tổng* (total cost): kết hợp chi phí tìm kiếm và chi phí đường đi của lời giải. Trong bài toán tìm đường đi trên bản đồ, chi phí tìm kiếm là thời gian thực hiện tìm kiếm (đo bằng mili giây chẳng hạn) và chi phí lời giải là tổng chiều dài của đường (tính theo đơn vị km chẳng hạn). Như thế, để tính chi phí tổng ta cần phải cộng *mili giây* và *km* ! Không có một cách chính thức nào để đổi *km* ra giây. Tuy nhiên ta ca cũng có thể đổi *km* sang mili giây bằng cách ước lượng vận tốc trung bình của xe ô-tô. Điều này cho phép dung hoà giữa chi phí tính toán để tìm lời giải và chi phí của lời giải.

2.2.3 Các chiến lược tìm kiếm

Các chiến lược tìm kiếm trong trong không gian trạng thái có thể chia thành ba loại :

Tìm kiếm tiến (forward chaining) - Tìm kiếm trạng thái đích. Quá trình tìm kiếm xuất phát từ tập các trạng thái bắt đầu. Đầu tiên tìm kiếm trạng thái đích trong tập các trạng thái bắt đầu, nếu tìm thấy, quá trình tìm kiếm dừng, nếu không xây dựng một tập các trạng thái có thể đạt tới từ các trạng thái bắt đầu và xét nó như tập các "trạng thái bắt đầu" mới, lặp lại quá trình tìm kiếm với tập các "trạng thái bắt đầu" mới.

Quá trình tìm kiếm có thể được tiến hành theo hai kiểu :

- Đồng thời: tìm kiếm trong toàn bộ tập các "trạng thái bắt đầu" trước khi chuyển sang tìm kiếm trong tập các "trạng thái bắt đầu"mới.
- Chọn lọc: chọn ra một trạng thái trong tập các "trạng thái bắt đầu", xem xét nó, tạm thời bỏ qua các "trạng thái ban đầu" còn lại, xây

dựng tập các ”trạng thái bắt đầu” mới , chuyển sang tìm kiếm trong tập các “trạng thái bắt đầu” mới, theo kiểu “chọn lọc”.

Tìm kiếm lùi (backward chaining) - Tìm kiếm trạng thái bắt đầu. Quá trình tìm kiếm xuất phát từ tập các trạng thái đích. Đầu tiên xem xét có trạng thái đích nào là trạng thái bắt đầu không? Nếu đúng quá trình tìm kiếm dừng nếu không, xác định các trạng thái có thể đi trực tiếp tới trạng thái đích, tập các trạng thái này được xem như tập các “trạng thái đích” mới, lặp lại quá trình tìm kiếm với tập các “trạng thái đích” mới.

Tìm kiếm hai hướng - Kết hợp tìm kiếm dây chuyền tiến và tìm kiếm dây chuyền lùi. Quá trình tìm kiếm dừng lại khi tập các “trạng thái ban đầu” trong tìm kiếm dây chuyền tiến và tập các “trạng thái đích” trong tìm kiếm dây chuyền lùi có phần tử chung.

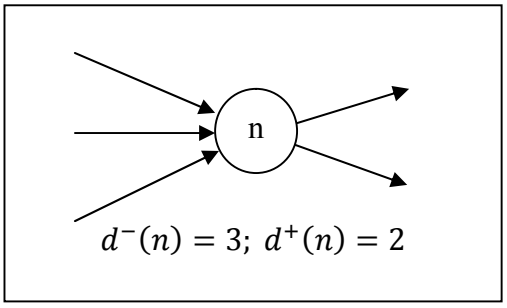
Một số gợi ý cho chọn lựa hướng tìm kiếm :

Tìm kiếm dây chuyền tiến phù hợp khi:

- Tập các trạng thái bắt đầu xác định tường minh
- Có nhiều đích, nhưng chỉ có một số nhỏ các phép toán có thể áp dụng vào một trạng thái (bặc cộng của mỗi nút nhỏ : d^+ nhỏ).

Tìm kiếm dây chuyền lùi phù hợp khi :

- Tập các trạng thái đích xác định tường minh, tập các trạng thái bắt đầu không được cho trước.
- Có nhiều toán tử có thể áp dụng lên một trạng thái (d^+ lớn), nhưng có một số nhỏ hơn các cung đến một nút (d^- nhỏ)



Tuy nhiên, đối với rất nhiều bài toán, khó xác định được từ ban đầu chiến lược tìm kiếm nào hiệu quả hơn !

Phần tiếp theo đây sẽ trình bày cụ thể các chiến lược tìm kiếm chỉ dựa vào thông tin mô tả của bài toán. Vì thế họ các giải thuật này còn có tên gọi là *tìm kiếm không có thông tin bổ sung* (uninformed search) hay *tìm kiếm mù* (blind search). Thuật ngữ tìm kiếm không có thông tin bổ sung có nghĩa là các chiến lược tìm kiếm không có thông tin gì khác ngoài các trạng thái được cung cấp trong *định nghĩa vấn đề* (problem definition). Tất cả những gì có thể làm là sinh ra các *trạng thái tiếp theo* (successors) từ *trạng thái hiện tại* (current state) và phân biệt được đâu là *trạng thái mục tiêu* (goal state) đâu không phải là trạng thái mục tiêu (no-goal state). Người ta phân biệt các chiến lược tìm kiếm dựa trên thứ tự triển khai các nút trong quá trình tìm kiếm. Các chiến lược tìm kiếm nào "biết" được trạng thái (không phải trạng thái đích) này "nhiều hứa hẹn" hơn các trạng thái khác gọi là *chiến lược tìm kiếm có thông tin bổ sung* (informed search) hay *chiến lược tìm kiếm heuristic* (heuristic search); các chiến lược này được trình bày trong chương 3.

2.2.4 Tìm kiếm theo chiều rộng (Breadth-first search)

Tìm kiếm theo chiều rộng là một chiến tìm kiếm đơn giản trong đó nút gốc (tương ứng với trạng thái bắt đầu) được triển khai trước, kế tiếp tất cả các nút con của nút gốc được triển khai, sau đó đến lượt các *nút con của các nút con này*, và cứ tiếp tục như thế. Nói một cách tổng quát là tất cả các nút được triển khai theo *độ sâu* (depth) của chúng trong cây tìm kiếm.

2.2.4.1 Giải thuật tìm kiếm theo chiều rộng

Tìm kiếm theo chiều rộng là một thể hiện của *giải thuật tìm kiếm tổng quát trên đồ thị* (general graph-search algorithm) trong đó *nút chưa triển khai có độ sâu nhỏ nhất* (shallowest node) sẽ được chọn để triển khai trước. Chiến lược tìm kiếm này có thể cài đặt dễ dàng bằng cách sử dụng một hàng đợi FIFO (vào trước ra trước) để lưu trữ các nút sắp sửa triển khai. Như thế, các nút mới (có độ sâu lớn hơn độ sâu của cha nó) sẽ nằm cuối hàng đợi và các nút cũ (có độ sâu nhỏ hơn nút mới), sẽ được triển khai trước. Chỉ có một sự khác biệt nhỏ so với giải thuật tìm kiếm tổng quát trên đồ thị là việc kiểm tra một nút có phải là trạng thái kết thúc không được thực hiện *khi nó được sinh ra chứ không phải khi chọn nó để triển khai*. Lý do cho sự quyết định này được giải thích trong phần phần bên dưới khi ta thảo luận về độ phức tạp thời gian. Cũng cần phải chú ý rằng: theo mô hình tìm kiếm tổng quát, giải thuật sẽ bỏ qua tất cả các đường đi mới đi đến một trạng thái đã có trong hàng đợi hoặc đã

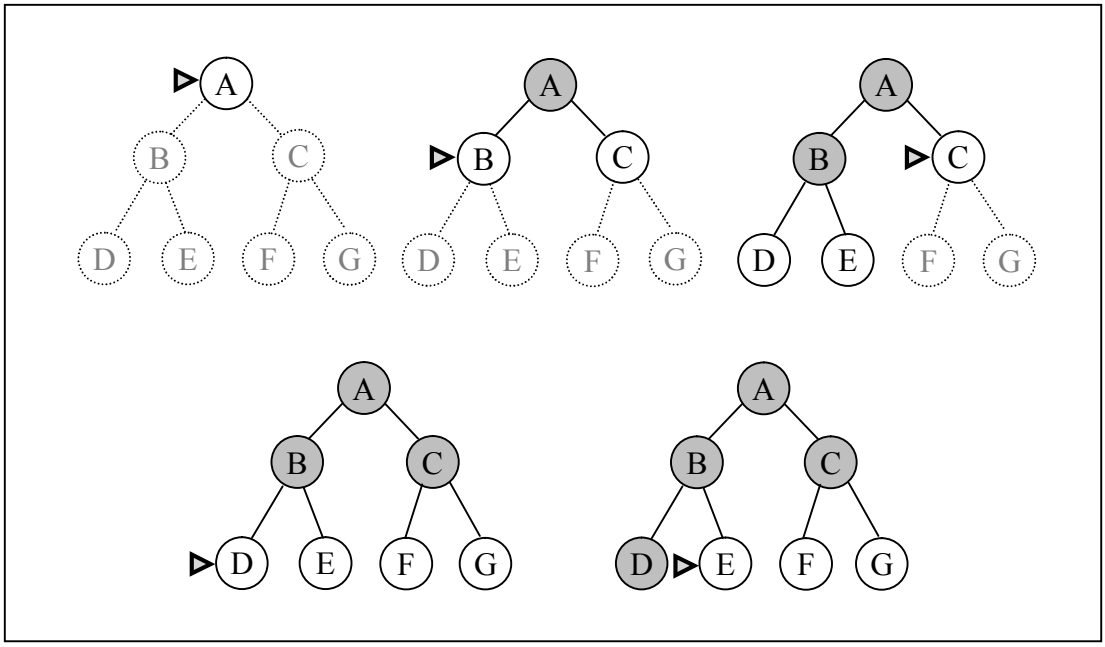
được triển khai rồi (các nút nằm trong hàng đợi là các nút nằm trên đường biên phân chia các nút đã được triển khai và các nút chưa được duyệt đến). Dễ dàng thấy rằng các đường đi mới này luôn có chiều dài ít nhất là bằng với chiều dài đường đi đã tìm thấy. Vì thế, chiến lược tìm kiếm theo chiều rộng luôn có đường đi ngắn nhất (có số cung đi qua ít nhất) từ nút gốc đi đến các nút nằm trên đường biên. Mã giả cho giải thuật tìm kiếm theo chiều rộng được cho trong Hình 2.11. Hình 2.12 minh hoạ quá trình tìm kiếm trên một cây nhị phân đơn giản.

2.2.4.2 Đánh giá tìm kiếm theo chiều rộng

Có thể dễ dàng thấy rằng tìm kiếm theo chiều rộng là đầy đủ. Nếu nút mục tiêu cạn nhất (có độ sâu nhỏ nhất) có độ sâu d, chiến lược tìm kiếm theo chiều rộng sẽ tìm thấy nó sau khi đã sinh ra tất cả các nút cạn (nông) hơn nó (với điều kiện hệ số phân nhánh b hữu hạn). Chú ý rằng ngay khi nút mục tiêu được sinh ra, ta luôn biết rằng nó là nút cạn nhất vì tất cả các nút cạn hơn đều đã được sinh ra và đã được kiểm tra rồi. Nút mục tiêu cạn nhất chưa chắc là nút tối ưu. Về mặt kỹ thuật, tìm kiếm theo chiều rộng sẽ tối ưu nếu chi phí đường đi là một hàm không giảm theo độ sâu của nút. Trường hợp dễ gặp nhất là tất cả các hành động đều có chi phí như nhau.

```
PNode breadthFirstSearch(PState initial_state) {
   //Nút gốc biểu diễn trạng thái bắt đầu
   PNode root = (PNode)malloc(sizeof(Node));
   root->state = initial_state;
   root->path_cost = 0;
   root->parent = NULL;
   //Kiểm tra mục tiêu
   if (goalCheck(root->state))
      return root;
   //Đường biên là hàng đợi FIFO
   insert(root, frontier);
   //Khởi tạo tập nút đã triển khai bằng rỗng
   while (!empty(frontier)) {
      //Lấy nút cạn nhất từ đường biên và
      //loại bỏ nó ra khỏi đường biên
      //ra khỏi đường biên, gán vào node.
      Node* node = pop(frontier);
      //node là nút đã triển khai
      //đưa nó vào explored
      insert(node, explored);
      for each action{
         child = childNode(node, action)
         if (chils != NUL và
             child->state không có trong frontier và
            child->state không có trong explored) {
            if (goalCheck(child->state))
               returnchild; //tìm thấy
            insert(child, frontier);
         }
      }
      //Thất bại, không tìm thấy lời giải
      return NULL;
```

Hình 2.11: Giải thuật tìm kiếm theo chiều rộng.



Hình 2.12: Minh hoạ quá trình tìm kiếm theo chiều rộng trên cây nhị phân.

Giả sử ta đang thực hiện tìm kiếm trên một cây đồng nhất với tất cả các nút đều có đúng b nút con. Nút gốc của cây tìm kiếm sinh ra b nút con có độ sâu 1, mỗi nút này lại sinh ra b nút con có độ sâu 2. Tổng số nút có độ sâu 2 là b^2. Mỗi nút ở độ sâu 2 sinh ra b nút có độ sâu 3, dẫn đến tổng số nút có độ sâu 3 là b^3. Tiếp tục như thế, tổng số nút có độ sâu d là b^d. Giả sử, nút mục tiêu nằm ở độ sâu d. Trong trường hợp xấu nhất, nút mục tiêu sẽ là nút cuối cùng được sinh ra tại độ sâu này. Như thế tổng số nút được sinh ra là:

$$b^0 + b^1 + b^2 + \cdots + b^d \tag{2.1}$$

Nếu giải thuật tìm kiếm thực hiện việc kiểm tra mục tiêu khi nút được chọn triển khai thay vì kiểm tra khi nút được sinh ra, thì tất cả các nút có độ sâu d phải được triển khai trước khi tìm thấy nút mục tiêu. Điều này dẫn đến độ phức tạp thời gian là $O(b^{d+1})$.

Độ phức tạp không gian: với bất kỳ chiến lược tìm kiếm trên đồ thị nào mà lưu các nút đã được triển khai, độ phức tạp không gian luôn là b lần độ phức tạp thời gian. Trong trường hợp tìm kiếm theo chiều rộng, tất cả các nút đã sinh ra đều nằm trong bộ nhớ. Có $O\left(b^{d-1}\right)$ nút trong tập hợp các nút đã được triển khai và $O(b^d)$ nút nằm trên đường biên. Vì

thế độ phức tạp không gian là $O(b^d)$ có nghĩa là độ phức tạp không gian phụ thuộc vào số nút trên đường biên.

Một độ phức tạp có cận luỹ thừa như $O(b^d)$ là rất lớn. Bảng 2.3 mô tả thời gian và không gian cần thiết tại một số giá trị độ sâu trong tìm kiếm theo chiều rộng với hệ số phân nhánh $b = 10$. Trong bảng này ta giả sử rằng trong một giây có thể sinh ra 1 triệu nút và mỗi nút cần 1000 bytes để lưu trữ.

Bảng 2.3: Thời gian và bộ nhớ cần thiết cho chiến lược tìm kiếm theo chiều.

Độ sâu	**Số nút**	**Thời gian**	**Bộ nhớ**
2	110	0.11 mili giây	107 kilobytes
4	11,110	11 mili giây	10.6 megabytes
6	10^6	1.1 giây	1 gigabytes
8	10^8	2 phút	103 gigabytes
10	10^{10}	3 giờ	10 terabytes
12	10^{12}	13 ngày	1 petabytes
14	10^{14}	3.5 năm	99 petabytes
16	10^{16}	350 năm	10 exabytes

Có hai bài học được rút ra từ bảng 2.3. Trước hết ta có thể thấy rằng yêu cầu về bộ nhớ là vấn đề lớn hơn là thời gian thực thi. Ta có thể chờ 13 ngày để nhận được kết quả cho một vấn đề quan trọng tại độ sâu 12, nhưng không một máy tính nào có đủ 1 petabyte bộ nhớ. May mắn thay, các chiến lược tìm kiếm khác đòi hỏi bộ nhớ ít hơn.Bài học thứ hai là thời gian thực thi vẫn là một vấn đề quan trọng. Nếu bài toán của chúng ta có lời giải nằm ở độ sâu 16, cần khoảng 360 năm chiến lược tìm kiếm theo chiều rộng (và các chiến lược tìm kiếm không có thông tin khác) mới có thể tìm thấy nó. Tóm lại, các vấn đề tìm kiếm có độ phức tạp luỹ thừa không thể giải quyết được bằng các phương pháp thiếu thông tin ngoại trừ các vấn đề rất nhỏ.

2.2.5 Tìm kiếm chi phí đồng nhất (Uniform-cost search)

Khi tất cả các bước (hoặc các hành động) đều có chi phí như nhau, tìm kiếm theo chiều rộng sẽ tối ưu vì nó luôn triển khai các nút cạn nhất. Mở rộng chiến lược này một chút, ta có thể tìm một được giải thuật tối ưu với bất kỳ hàm chi phí nào. Thay vì triển khai các nút cạn nhất, chiến lược tìm kiếm chi phí phí đồng nhất sẽ triển khai các nút có *chi phí đường đi thấp nhất từ gốc đến nó,* $g(n)$. Điều này có thể thực hiện bằng cách lưu trữ đường biên như một hàng đợi ưu tiên sắp xếp theo $g(n)$.

Để có thể duy trì thứ tự của các nút trong hàng đợi theo chi phí đường đi của chúng, có hai sự khác biệt lớn so với tìm kiếm theo chiều rộng. Đầu tiên là việc kiểm tra mục tiêu được thực hiện *khi chọn nút để triển khai chứ không phải khi nút được sinh ra*. Lý do là vì nút mục tiêu đầu tiên được sinh ra có thể chỉ là nút *gần tối ưu* (suboptimal) chứ chưa phải phải nút tối ưu. Có thể còn đường đi khác đi đến nút này với chi phí đường đi nhỏ hơn. Khác biệt thứ hai là thêm một phép kiểm tra nữa trong trường hợp tìm được một đường đi tốt hơn đến một nút đang nằm trên đường biên.

Dễ dàng thấy rằng tìm kiếm chi phí đồng nhất luôn tối ưu. Để chứng minh điều này, trước hết, ta nhận thấy rằng mỗi khi chọn một nút n để triển khai, ta đã tìm được đường đi tối ưu từ nút gốc đến nó.

Chứng minh điều này bằng phản chứng như sau: *Giả sử đường đi hiện tại đến* n *chưa phải là đường đi tối ưu*. Như thế phải tồn tại một đường đi khác đến n chứa nút n' (n' nằm trên đường biên) có chi phí:

$$g^*(n) < g(n) \tag{2.2}$$

Ta có thể tách chi phí $g^*(n)$ thành hai thành phần:

$$g^*(n) = g(n') + d(n', n) \tag{2.3}$$

trong đó, $g(n')$ là chi phí đường đi từ nút gốc đến n' và $d(n', n)$ là tổng chi phí các bước từ nút n' đến nút n trên đường đi mới.

Mặt khác, theo quy tắc *chọn nút có chi phí g(n) nhỏ nhất để triển khai*, với mọi *n*’ nằm trên đường biên ta có:

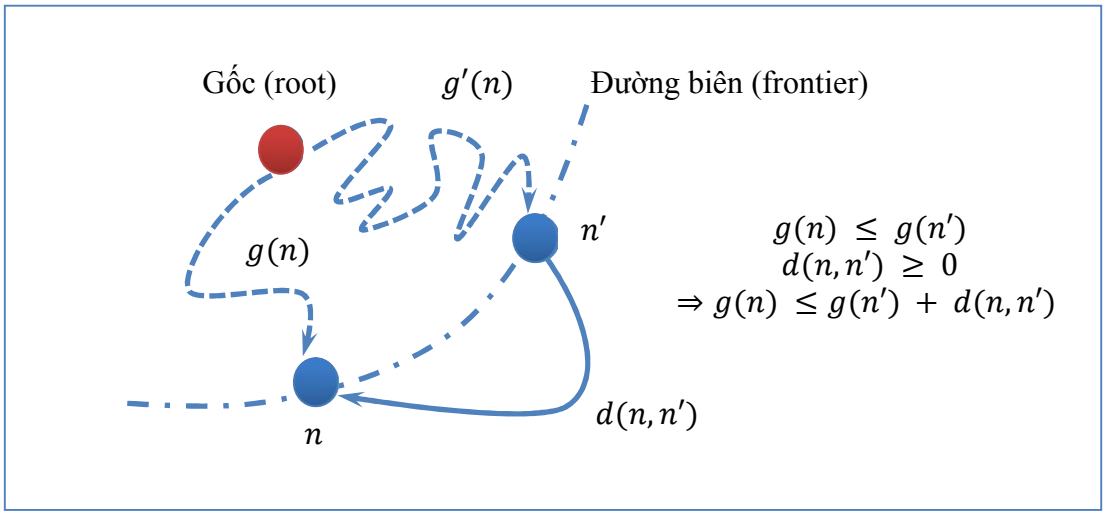
$$g(n) \leq g(n') \tag{2.4}$$

Hơn nữa, vì chi phí các bước đều không âm, tổng chi phí từ n' đến n trên đường đi mới phải không âm hay:

$$d(n', n) \geq 0 \tag{2.5}$$

Kết hợp (2.3), (2.4) và (2.5) ta có $g^*(n) \geq g(n)$. Điều này mâu thuẫn với (2.2). Vì thế đường đi hiện tại đến n với chi phí $g(n)$ là đường đi tối ưu. Hình 2.9 minh hoạ cho chứng minh này.

Chính vì ta luôn triển khai nút có chi phí đường đi nhỏ nhất trước, nên nếu nút mục tiêu đầu tiên được triển khai thì nó sẽ là nút tối ưu.



Hình 2.13: Chứng minh tính chất "đường đi đến nút được chọn để triển khai là đường đi tối ưu".

Trong khi chiến lược tìm kiếm theo chiều rộng triển khai các nút theo thứ tự độ sâu của nút, chiến lược tìm kiếm theo chi phí đồng nhất sẽ triển khai các nút theo thứ tự của đường đi tối ưu đi đến nó.

Chiến lược tìm chi phí đồng nhất không quan tâm đến *số bước trên đường đi* mà chỉ quan tâm đến *tổng chi phí của các bước trên đường đi*. Vì thế nó có thể rơi vào vòng lặp vô tận nếu có đường đi chứa một chuỗi vô hạn các hành động với chi phí bằng không, ví dụ một chuỗi các hành động *NoOp* (NoOp có nghĩa là không làm gì cả). Tìm kiếm chi phí đồng nhất sẽ đầy đủ nếu chi phí của tất cả các bước/hành động đều lớn hơn một hằng số dương nhỏ ε nào đó.

Tìm kiếm theo chi phí đồng nhất sẽ được điều khiển bằng chi phí đường đi đến nút chứ không phải độ sâu của nút. Vì thế rất khó biểu diễn độ phức tạp theo hệ số phân nhánh b và độ sâu d. Gọi C^* là chi phí của lời giải tối ưu (chi phí của đường đi từ gốc đến nút tối ưu), và giả sử rằng chi phí của mỗi hành động ít nhất là bằng ε (dương) thì độ phức tạp thời gian trong trường xấu nhất và độ phức tạp không gian là $O\left(b^{1+C*/\varepsilon}\right)$. Độ phức tạp này có thể lớn hơn $O(b^d)$ rất nhiều. Đó là bởi vì tìm kiếm chi phí đồng nhất có thể mở rộng cây tìm kiếm theo nhánh có nhiều bước với chi phí nhỏ trước khi mở rộng nhánh có ít bước nhưng chi phí lớn hơn. Tuy nhiên nhánh này lại có thể hữu ích hơn. Trong trường hợp chi phí các bước là như nhau, $b^{1+C*/\varepsilon}$ chính là b^{1+d}. Khi đó, tìm kiếm chi phí đồng nhất chính là tìm kiếm theo chiều rộng, ngoại trừ một điểm khác biệt duy nhất: *phép kiểm tra mục tiêu được thực hiện khi chọn nút để triển khai*.

2.2.6 Tìm kiếm theo chiều sâu (Depth-first search)

Tìm kiếm theo chiều sâu sẽ luôn chọn nút sâu nhất trên đường biên cây tìm kiếm để triển khai. Giải thuật tìm kiếm theo chiều sâu là một thể hiện của giải thuật tìm kiếm trên đồ thị; trong khi tìm kiếm theo chiều rộng sử dụng hàng đợi FIFO để lưu trữ các nút trên đường biên, tìm kiếm theo chiều sâu sử dụng một ngăn xếp LIFO (Last In First Out, vào sau ra trước). Ngoài phương pháp cài đặt tìm kiếm theo chiều sâu bằng phương pháp tìm kiếm tổng quát trên đồ thị, ta có thể cài đặt tìm kiếm theo chiều sâu bằng phương pháp đệ quy.

Các tính chất của tìm kiếm theo chiều sâu phụ thuộc rất nhiều vào phiên bản *tìm kiếm trên đồ thị* (graph search) hay *tìm kiếm trên cây* (tree search). Nếu sử dụng phương pháp tìm kiếm trên đồ thị (có sử dụng tập explored), tìm kiếm theo chiều sâu sẽ đầy đủ trong không gian hữu hạn các trạng thái. Phương pháp tìm kiếm trên cây sẽ không đầy đủ vì có thể rơi vòng lặp vô tận. Tìm kiếm trên cây theo độ sâu có thể thay đổi chút ít để tránh trường hợp này: mỗi khi sinh ra một nút mới ta kiểm tra nó đã tồn tại trên đường đi từ gốc đến nút hiện tại chưa. Làm như thế giúp ta có thể tránh được vòng lặp vô tận trong không gian trạng thái hữu hạn. Tuy nhiên, vẫn không tránh khỏi được các đi dư thừa. Nếu không gian trạng thái vô hạn, cả hai phương pháp đều thất bại nếu ta rời vào một đường đi vô hạn mà không dẫn đến mục tiêu.Cũng vì các lý do giống nhau, cả hai phương pháp đều không tối ưu.

Độ phức tạp thời gian của tìm kiếm trên đồ theo chiều sâu bị chặn trên bằng kích thước của không gian trạng thái (có thể vô hạn) trong khi độ phức tạp của tìm kiếm trên cây theo chiều sâu có thể sinh ra $O(b^m)$ nút trong cây tìm kiếm với m là độ sâu của nút sâu nhất. Độ phức tạp này có thể lớn hơn rất nhiều so với kích thước của không gian trạng thái. Chú ý là m có thể lớn hơn rất nhiều so với d (độ sâu của nút mục tiêu cạn nhất) và là vô hạn nếu cây không bị chặn.

Như thế, tìm kiếm theo chiều sâu dường như không có ưu điểm nào so với tìm kiếm theo chiều rộng, vậy tại sao ta sử dụng nó ? Câu trả lời là do độ phức tạp không gian. Với phương pháp tìm kiếm trên đồ thị, ta không có thuận lợi nào nhưng với phương pháp tìm kiếm trên cây ta chỉ cần lưu trữ một đường đi từ gốc đến nút lá, kèm theo các nút anh em của các nút trên đường đi này. Khi một nút được triển khai, nó sẽ được loại khỏi bộ nhớ nếu các nút hậu duệ của nó đã được triển khai hoàn toàn. Với một không gian trạng thái có hệ số phân nhánh b và độ sâu tối đa m, tìm kiếm theo chiều sâu chỉ cần lưu trữ $O(bm)$ nút.

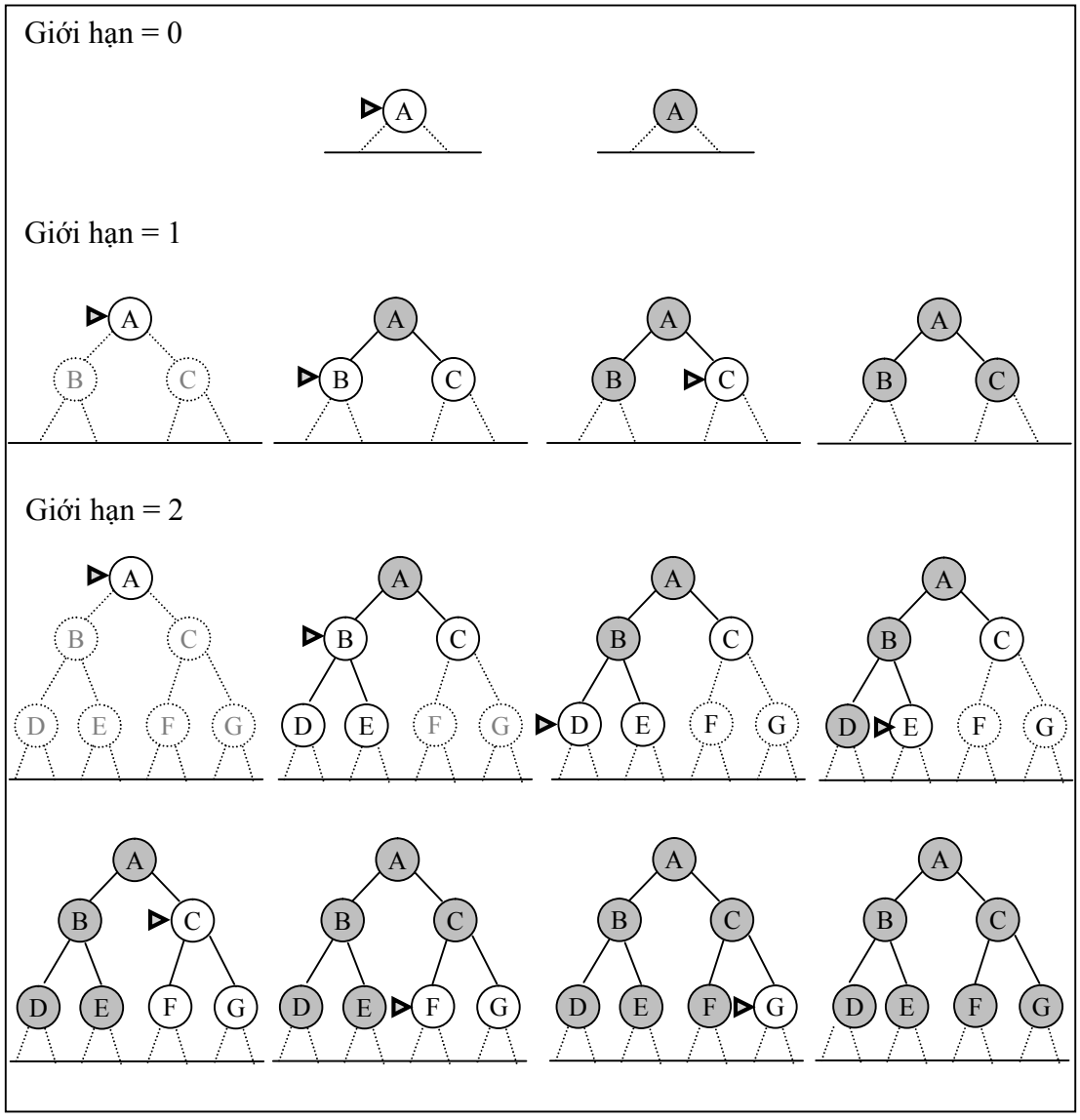
2.2.7 Tìm kiếm với độ sâu giới hạn (Depth-limited search)

Thất bại của tìm kiếm theo chiều sâu trong không gian trạng thái vô hạn có thể được tránh bằng cách giới hạn độ sâu tối đa l. Có nghĩa là nút có độ sâu l được xem như nút lá (không có hậu nút duệ). Tiếp cận này gọi là tìm kiếm với độ sâu giới hạn. Giới hạn về độ sâu giúp giải quyết được vấn đề đường đi vô hạn. Nhưng cũng thật không may, đây cũng là nguyên nhân làm cho phương pháp này không đầy đủ nếu ta chọn $l < d$, có nghĩa là mục tiêu cạn nhất vẫn nằm ngoài phạm vi tìm kiếm. Tìm kiếm với độ sâu giới hạn có thể không tối ưu nếu $l > d$. Độ phức tạp thời gian là $O(b^l)$ và độ phức tạp không gian là $O(bl)$. Tìm kiếm theo chiều sâu có thể được xem là trường hợp đặc biệt của tìm kiếm với độ sâu giới hạn với $l = \infty$.

2.2.8 Tìm kiếm sâu dần (Iterative deepening depth-first search)

Tìm kiếm sâu dần là một chiến lược tìm kiếm tổng quát thường dùng với tìm kiếm (trên cây) theo chiều sâu cho phép tìm được độ sâu giới hạn tốt nhất. Thực hiện điều này bằng cách tăng dần độ sâu giới hạn – đầu tiên là 0, sau đó là 1, 2, 3, … cho đến khi tìm thấy mục tiêu. Ta sẽ tìm thấy mục tiêu khi độ sâu giới hạn đạt đến d, độ sâu của nút mục tiêu cạn nhất (gần với gốc nhất). Chiến lược tìm kiếm sâu dần tận dụng được thế mạnh của cả hai chiến lược tìm kiếm theo chiều rộng và tìm kiếm theo

chiều sâu. Giống như tìm kiếm theo chiều sâu, bộ nhớ yêu cầu ít nhất $O(bd)$. Giống như tìm kiếm theo chiều rộng, chiến lược này sẽ đầy đủ khi hệ số phân nhánh hữu hạn và sẽ tối ưu khi chi phí đường đi là một hàm không giảm theo độ sâu của nút. Hình 2.14 trình bày 4 bước lặp của chiến lược tìm kiếm sâu dần trên một cây nhị phân, lời giải được tìm thấy trong bước lặp 4.



Hình 2.14: Tìm kiếm sâu dần trên cây nhị phân.

Tìm kiếm sâu dần có vẻ như rất lãng phí vì ta phải bắt đầu việc tìm kiếm từ nút gốc sau mỗi bước lặp. Tuy nhiên, chi phí cho việc lặp lại này không quá cao. Lý do là vì trong một cây tìm kiếm có cùng hệ số phân nhánh tại các nút (hoặc hệ số phân nhánh của các nút gần như bằng

nhau), hầu hết các nút đều nằm ở mức cuối cùng (độ sâu d). Như vậy, ta không cần quan tâm đến việc các nút ở trên được sinh ra nhiều lần. Trong tìm kiếm sâu dần, các nút có độ sâu d chỉ được sinh ra đúng 1 lần. Các nút ở độ sâu $d-1$ được sinh ra 2 lần, và cứ như thế cho đến các nút con của nút gốc. Và như thế trong trường hợp xấu nhất, tổng số nút được sinh ra là:

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

và độ phức tạp thời gian là $O(b^d)$ tương đương với tìm kiếm theo chiều rộng. Thực ra tìm kiếm sâu dần có thêm ít chi phí do việc các nút được sinh ra nhiều lần, tuy nhiên chi phí này không đáng kể. Ví dụ, với $b = 10$ và $d = 5$

N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

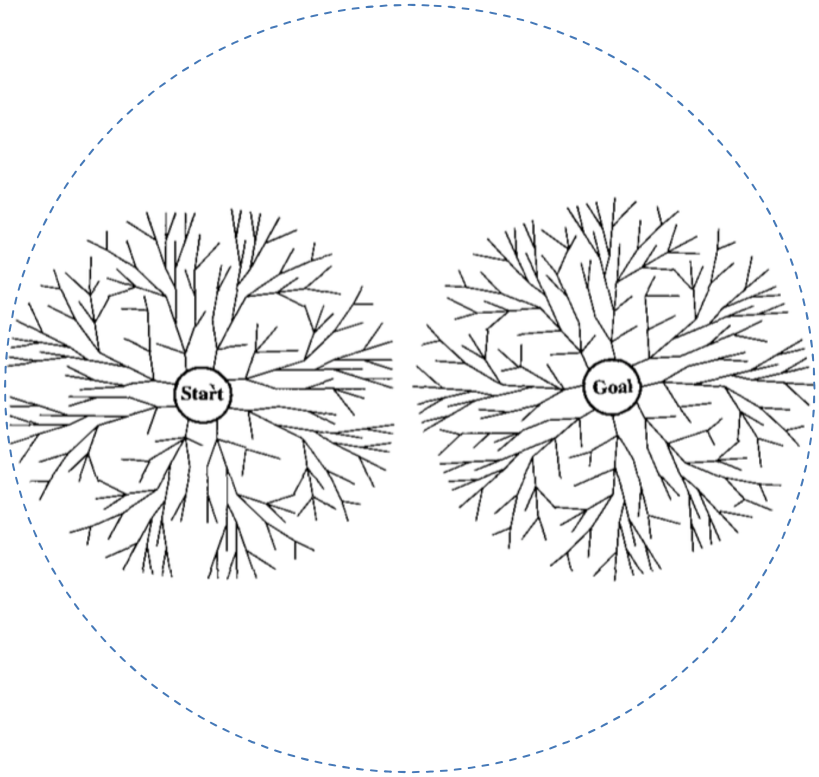
N(BFS) = 10 + 100 + 1,000 + 10,000+ 100,000 = 111,110

Nếu quan tâm đến chi phí này, ta có thể sử dụng tiếp cận lai: chạy tìm kiếm theo chiều rộng cho đến khi hết bộ nhớ và sau đó thực hiện tìm kiếm sâu dần từ các nút nằm trên đường biên. *Nói chung, tìm kiếm sâu dần là phương pháp tìm kiếm không có thông tin bổ sung được ưa thích khi không gian trạng thái lớn và không biết trước độ sâu của lời giải*.

Tìm kiếm sâu dần tương tự như tìm kiếm theo chiều rộng ở chỗ nó triển khai đầy đủ các nút theo từng mức trước khi đi xuống mức kế tiếp. Ta có thể phát triển chiến lược này cho tìm kiếm với chi phí đồng nhất. Ý tưởng chính là sử dụng giới hạn về chi phí đường đi thay vì giới hạn về độ sâu. Kết quả cho ta *giải thuật tìm kiếm xa dần* (iterative lengthening search). Tuy nhiên, khi so sánh với giải thuật tìm kiếm chi phí đồng nhất, tìm kiếm xa dần có độ phức tạp thời gian lớn hơn.

2.2.9 Tìm kiếm hai chiều (Bidirectional search)

Ý tưởng phía sau chiến lược tìm kiếm hai chiều là tìm kiếm đồng thời từ 2 phía: tìm kiếm xuôi từ trạng thái bắt đầu và tìm kiếm ngược từ trạng thái mục tiêu với hy vọng rằng sẽ gặp nhau ở giữa đường. Động lực chính của ý tưởng này là $b^{d/2} + b^{d/2}$ nhỏ hơn b^d rất nhiều. Trên hình 2.15 tổng diện tích hai hình tròn nhỏ nhỏ hơn diện tích của hình tròn lớn với tâm là start và bán kính là khoảng cách từ start đến goal.



Hình 2.15: Tìm kiếm hai chiều.

Tìm kiếm hai chiều được cài đặt bằng cách thay phép kiểm tra mục tiêu bằng việc kiểm tra hai đường biên có giao nhau không. Nếu giao nhau, ta sẽ tìm thấy lời giải. Cần phải chú ý rằng lời giải đầu tiên tìm theo phương pháp này có thể không tối ưu, ngay cả khi cả hai tìm kiếm đều là tìm kiếm theo chiều rộng. Việc kiểm tra có thể được thực hiện ngay khi mỗi nút được sinh ra hoặc khi được chọn để triển khai. Với một bảng băm độ phức tạp thời gian của phép kiểm tra này là hằng số. Ví dụ, nếu bài toán có lời giải ở độ sâu 6, và mỗi tìm kiếm sử dụng tìm kiếm theo chiều rộng, thì trong trường hợp xấu nhất hai giải thuật tìm kiếm sẽ gặp nhau khi chúng sinh ra tất cả các nút ở độ sâu 3. Với $b = 10$, có tất cả 2,220 nút được sinh ra, so với 1,111,110 nút trong trường hợp tìm kiếm theo chiều rộng chuẩn. Vì thế, độ phức tạp thời gian của tìm kiếm hai chiều sử dụng tìm kiếm theo chiều rộng theo hai hướng là $O\left(b^{d/2}\right)$. Độ phức tạp không gian cũng là $O\left(b^{d/2}\right)$. Ta có thể giảm bớt độ phức tạp không gian đi một nửa nếu cả hai chiều tìm kiếm sử dụng chiến lược tìm kiếm sâu dần, nhưng phải lưu trữ ít nhất một đường biên để kiểm tra giao nhau. Yêu cầu về bộ nhớ không phải là điểm yếu chính của tìm kiếm hai chiều.

Giảm bớt đáng kể độ phức tạp thời gian trong tìm kiếm hai quả là hấp dẫn. Tuy nhiên làm thế nào để tìm kiếm theo chiều ngược ? Không dễ dàng như ta nghĩ chút nào. Gọi các *tiền bối* (predecessors) của trạng thái x là tất cả các trạng thái có *hậu bối* (successors) là x. tìm kiếm hai chiều đòi hỏi phải có cách nào đó tính các tiền bối của một trạng thái. Nếu các hành động trong không gian trạng thái đều *khả đảo* (reversible), các tiền bối của x cũng chính là các hậu bối của nó. Trong các trường hợp khác, đòi hỏi phải có cách nào đó định nghĩa các hành động đảo.

Xét câu hỏi "*mục tiêu* là gì ?" trong tìm kiếm ngược bắt đầu từ trạng thái mục tiêu. Đối với bài toán 15-puzzles hay bài toán tìm đường đi ở Rumani, câu trả lời đơn giản là trạng thái mong muốn. Nếu tồn tại nhiều trạng thái mục tiêu tường minh, ta có thể xây dựng một *trạng thái mục tiêu giả* (dummy goal) có tiền bối là các trạng thái mục tiêu này. Nếu mục tiêu chỉ là một mô tả trừu tượng như: "*không có quân hậu nào khống chế quân hậu nào*" trong bài toán 8 quân hậu, thì rất khó sử dụng chiến lược tìm kiếm hai chiều.

2.2.10 So sánh các chiến lược tìm kiếm không có thông tin bổ sung

Bảng 2.4 So sánh các chiến lược tìm kiếm theo bốn tiêu chí được trình bày trong phần trên. Các so sánh này được thực hiện trên phiên bản tìm kiếm trên cây. Với tìm kiếm trên đồ thị, khác nhau chính là tìm kiếm theo chiều sâu sẽ đầy đủ với không gian trạng thái hữu hạn và độ phức tạp không gian và thời gian bị chặn bởi kích thước của không gian trạng thái. Ý nghĩa các ký tự [a], [b], [d] trong dấu ngoặc (trong hai tiêu chí đầy đủ và tối ưu) như sau: [a] đầy đủ nếu hệ số phân nhánh b hữu hạn; [b] đầy đủ nếu chí phí từng bước $\geq \varepsilon$; [c] tối ưu nếu chi phí từng bước như nhau; [d] nếu cả hai hướng đều dùng tìm kiếm theo chiều rộng.

Bảng 2.4: Đánh giá các chiến lược tìm kiếm không có thông tin bổ sung.

Tiêu chí	**Tìm kiếm theo chiều rộng**	**Tìm kiếm chi phí đồng nhất**	**Tìm kiếm theo chiều sâu**	**Tìm kiếm độ sâu giới hạn**	**Tìm kiếm sâu dần**	**Tìm kiếm hai chiều**
Đầy đủ ?	Có [a]	Có [a, b]	Không	Không	Có [a]	Có [a, d]
Tối ưu ?	Có [c]	Có	Không	Không	Có [c]	Có [c, d]
Thời gian	$O(b^d)$	$O(b^{1+\lfloor C*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Không gian	$O(b^d)$	$O(b^{1+\lfloor C*/\varepsilon \rfloor})$	$O(bm)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$

2.3 Các hệ thống luật sinh

Tìm kiếm là cốt lõi của nhiều quá trình trí tuệ. Các *hệ thống luật sinh* (production system hay production rule system) còn gọi là *hệ sinh*cung cấp các cấu trúc tạo thuận lợi cho việc mô tả và thực hiện quá trình tìm kiếm.

2.3.1 Định nghĩa hệ thống luật sinh

Luật sinh (production rule) làm một cấu trúc gồm hai phần có dạng:

<điều kiện> $\rightarrow$ <hành động>

vế trái xác định điều kiện áp dụng luật (với điều kiện nào thì có thể áp dụng luật này), vế phải mô tả hành động được thực hiện khi luật được áp dụng.

Một hệ thống luật sinh bao gồm:

1. *Một tập các luật sinh (production rules)* bao gồm tất cả các luật sinh có thể áp dụng được đối với bài toán đang xét. Các luật sinh được lưu trữ trong một vùng bộ nhớ được gọi là *bộ nhớ luật sinh* (production memory).

2. *Một tập các tri thức/cơ sở dữ liệu* chứa thông tin thích hợp cho nhiệm vụ xác định. Thông tin trong cở sở dữ liệu này có thể được cấu trúc một cách thích hợp. Một số phần của cơ sở dữ liệu liên quan đến giải quyết bài toán hiện hành được lưu trữ trong một vùng bộ nhớ được gọi là *bộ nhớ làm việc* (working memory), các phần còn lại có thể được lưu trữ trong bộ nhớ ngoài.
3. *Một chiến lược điều khiển*, xác định thứ tự các luật được so sánh với cơ sở dữ liệu và phương pháp giải quyết xung đột khi một vài luật cùng tương hợp (các luật có vế trái được làm thoả mãn bởi trạng thái hiện hành).
4. *Một bộ áp dụng luật*: thực thi luật do chiến lược điều khiển chọn ra.

2.3.2 Chiến lược điều khiển

Trong quá trình tìm kiếm lời giải cho một bài toán, thông thường có nhiều luật có thể áp dụng lên cùng một trạng thái, chọn lựa luật nào trong chúng để có thể đẩy nhanh tới trạng thái đích? Chiến lược điều khiển được xem như bộ quyết định trong chọn lựa luật. Một chiến lược điều khiển tốt sẽ mang lại hiệu quả tìm kiếm cao.

Một chiến lược điều khiển tốt phải thoả mãn hai tiêu chuẩn:

1. ***Tạo ra sự vận động***. Trong bài toán đong dầu, nếu chiến lược điều khiển là áp dụng luật tương hợp vớitrạngtháihiện hành gặp đầu tiên trong dãy Cxy, Cyx, Cxz, Czx, Cyz, Czy.Trạng thái ban đầu (7, 0, 0) luật đầu tiên tương hợp là Cxy, áp dụng luật cho ra trạng thái (2, 5, 0) luật đầu tiên trong dãy tương hợp với trạng thái này là Cyx, áp dụng luật cho ra trạng thái (7, 0, 0) … chiến lược điều khiển này không tạo ra sự vận động!
2. ***Có hệ thống***. Trong bài toán đong dầu, nếu chiến lược điều khiển là chọn ngẫu nhiên một quy tắc tương hợp với trạng thái hiện hành, chiến lược này tốt hơn chiến lược trên vì nó tạo ra sự vận động. Tuy nhiên, việc chọn lựa ngẫu nhiên này có thể dẫn đến việc "*gặp lại một trạng thái nhiều lần*" hoặc "*không gặp một trạng thái nào đó một lần nào cả*" trong quá trình tìm kiếm do chiến lược chọn ngẫu nhiên thiếu tính hệ thống. Một chiến lược điều khiển

phải cho phép mọi luật đều có khả năng được áp dụng. Hơn nữa, mục tiêu là tìm kiếm trạng thái đích, do vậy những luật đưa đến một trạng thái đã được xét đến không cần thiết được áp dụng.

2.3.3 Tìm kiếm heuristic

Heuristic là một kỹ thuật cải thiện tính hiệu quả của quá trình tìm kiếm, có thể phải hy sinh đòi hỏi về tính đầy đủ. Heuristic giống như người hướng dẫn du lịch. Heuristic tốt có thể hướng dẫn tìm thấy một lời giải tốt cho một bài toán khó trong một thời gian ngắn hơn. Heuristic có ưu điểm là chỉ ra hướng có lợi trong toàn thể nhưng cũng có thể bỏ sót những lợi thế của các cá nhân đặc biệt.

Có heuristic tốt chung cho nhiều lĩnh vực bài toán, chẳng hạn heuristic *lân cận gần nhất (nearest neighbor)* làm việc bằng cách ở mỗi bước chọn ra ứng viên nổi trội cục bộ. Có thể xây dựng các heuristic khai thác tri thức lĩnh vực để giải quyết các bài toán cá biệt.Không có heuristics, giải quyết vấn đề bùng nổ tổ hợp trở nên vô vọng.

Tìm kiếm heuristics hiếm khi cho ra lời giải tối ưu mà thường chỉ là một lời giải tốt. Tuy nhiên, trong thực tế hiếm khi bài toán đòi hỏi một lời giải tối ưu, một lời giải xấp xỉ tốt thường được chấp nhận. Con người thường chỉ tìm kiếm một giải pháp tốt cho vấn đề hơn là tìm kiếm giải pháp tốt nhất.

Ví dụ, giải phương trình sin3x + sin6x + sin9x = 0. Làm thế nào để giải phương trình này? Với "tri thức kinh nghiệm" thu nhận được trong khi giải các phương trình "dạng tương tự", định hướng ta đi theo con đường biến đổi thành tích: sin3x + sin9x + sin6x = 2sin6xcos3x + sin6x = sin6x(2cos3x + 1). Tri thức định hướng tìm kiếm lời giải này chính là tri thức heuristic.

Tri thức heuristic thường được kết hợp với thủ tục tìm kiếm dựa trên quy tắc thông qua:

1. ***Các luật***, chỉ ra luật nào hứa hẹn hơn các luật khác trong việc hướng tới đích
2. ***Hàm heuristic*** đánh giá các trạng thái và chỉ ra tính "hứa hẹn nằm trên đường đi lời giải" / "hứa hẹn sắp tới đích" của chúng. Hàm heuristic là một ánh xạ từ tập các trạng thái đến tập độ đo tính hứa hẹn (thường được biểu diễn bởi số)

Xét ví dụ, bài toán 15-puzzle:

Giả sử cấu hình xuất phát là:

11	14	4	7
10	6		5
1	2	13	15
9	12	8	3

cấu hình đích là:

1	2	3	4
12	13	14	5
11		15	6
10	9	8	7

Có thể biểu diễn trạng thái bài toán như một mảng hai chiều 4×4 nhận giá trị nguyên, với quy ước phần tử mảng có giá trị 0 là ô trống. Ta "nhận thấy" rằng trạng thái có số các ô sai vị trí so với trạng thái đích càng nhỏ càng hứa hẹn nó càng ở gần đích, như vậy trước tiên ta sẽ khai thác các trạng thái có ít ô sai vị trí với hy vọng sẽ mau chóng hơn đạt tới đích. Ta xây dựng được một hàm heuristic:

h_1(trạng thái) = số các ô sai vị trí so với đích

ta có h_1(trạng thái ban đầu) = 14

ô 0: sai,ô 1: sai, ô 2: sai, ô 3: sai, ô 4: sai, ô 5: đúng, ô 6: sai, ô 7: sai, ô 8:đúng, ô 9: sai, ô 10: sai, ô 11: sai, ô 12: sai, ô 13: sai, ô 14: sai, ô 15:sai.

Tìm kiếm theo hướng "khai thác" trạng thái có giá trị h_1 nhỏ.

"Hiểu biết về luật di chuyển các ô" cho ta nhận thức "một ô ở toạ độ (x, y) muốn di chuyển đến ô ở toạ độ (x', y') cần ít nhất$|x - x'| + |y - y'|$ bước chuyển chỗ".

Hiểu biết này cho phép ta xây dựng hàm heuristic h_2:

h_2(trạng thái) = tổng d_m(vị trí của ô trong trạng thái, vị trí của ô tương ứng trong trạng thái đích)

khoảng cách Manhattan từ điểm A(x, y) đến điểm B(x', y') ký hiệu d_m và được tính bởi công thức: $d_m(A, B) = |x - x'| + |y - y'|$. ta ký hiệu khoảng cách mahattan của vị trí ô số i đến vị trí của nó trong trạng thái đích là md(i).

h_2(trạng thái ban đầu) =$\sum_{i=0}^{15} md(i) = 30$

md(ô số 0) = 2, md(ô số 1) = 2, md(ô số 2) = 2, md(ô số 3) = 4, md(ô số 4) = 1, md(ô số 5)= 0, md(ô số 6) = 3, md(ô số 7) = 3, md(ô số 8) = 0, md(ô số 9) = 1, md(ô số 10) = 2, md(ô số 11) = 2, md(ô số 12) = 3, md(ô số 13) = 2, md(ô số 14) = 2, md(ô số 15) = 1.

Tìm kiếm theo hướng "khai thác" trạng thái có h_2 nhỏ.

h_2 chứa "lượng hiểu biết" về bài toán nhiều hơn h_1 vì nó bổ sung hiểu biết về luật dịch chuyển vào hiểu biết về sai khác giữa hai trạng mà h_1 khai thác. Câu hỏi đặt ra là heuristic nào tốt hơn,h_1 hay h_2? Heuristic chứa lượng tri thức về lĩnh vực nhiều hơn thường là heuristic tốt hơn: h_2 tốt hơn h_1.

Một hàm heuristic được thiết kế tốt có thể giữ một phần quan trọng trong việc định hướng một quá trình tìm kiếm hướng tới lời giải. Mục tiêu của hàm heuristic là định hướng quá trình tìm kiếm theo hướng có lợi nhất bằng cách gợi ý con đường phải đi theo đầu tiên khi có nhiều hơn một con đường.

Sử dụng hàm heuristic giúp ngắn thời gian tìm kiếm nhưng cũng làm nẩy sinh phí tổn "thời gian tính giá trị heuristic của các trạng thái". Nếu hàm heuristic quá phức tạp, thời gian cho tính các giá trị trở nên quan trọng. Cần có một sự thoả hiệp giữa thời gian tính giá trị hàm heuristic và sự tiết kiệm thời gian tìm kiếm do hàm heuristic cung cấp. Việc sử dụng heuristic trong tìm kiếm sẽ được trình bày chi tiết hơn trong chương 3.

2.4 Các đặc trưng của bài toán

Tìm kiếm heuristic là một phương pháp chung có thể áp dụng cho một lớp rộng các bài toán, bao gồm các kỹ thuật đặc trưng, mỗi một kỹ thuật có hiệu quả riêng biệt đối với một lớp nhỏ các bài toán. Để lựa

chọn phương pháp thích hợp nhất cho một bài toán cụ thể, cần phân tích bài toán theo một số hướng chủ đạo:

1. Bài toán có thể phân tích thành các bài toán nhỏ hơn hoặc dễ hơn không?
2. Các bước giải có thể bỏ lơ hay ít nhất có thể huỷ bỏ nếu chúng tỏ ra thiếu triển vọng?
3. Vũ trụ của bài toán có thể tiên đoán?
4. Bài toán đòi hỏi tìm lời giải tốt nhất hay chỉ cần tìm một lời giải?
5. Yêu cầu của bài toán là tìm một đường đi lời giải hay một trạng thái?
6. Lượng tri thức cần thiết để giải quyết bài toán?
7. Để giải quyết bài toán có đòi hỏi sự tương tác với con người?

2.4.1 Bài toán có thể phân tích?

Ví dụ, ta muốn tìm nguyên hàm

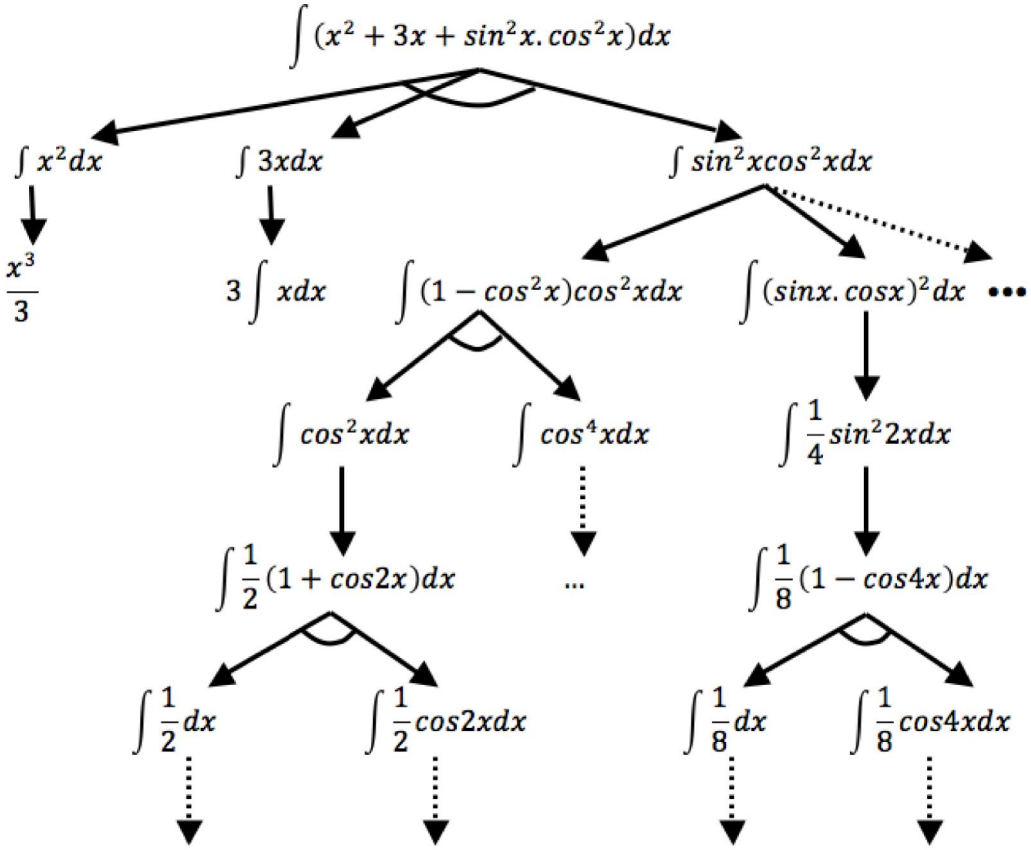
$$I = \int (x^2 + 3x + sin^2x.cos^2x)dx$$

Bài toán có thể được giải bằng cách phân tích thành ba bài toán nhỏ hơn:

$I = \int x^2dx + \int 3xdx + \int sin^2xcos^2xdx$ bài toán tìm nguyên hàm $\int x^2dx$ được xem là có thể giải được, bài toán $\int 3xdx$ có thể được phân tích thành$3\int xdx$với$\int xdx$ được xem là giải được. Bài toán $\int sin^2xcos^2xdx$ có thể phân tích thành $\int(1 - cos^2x)cos^2xdx$ hoặc $\int(sinx.cosx)^2dx$ …

Mỗi một bài toán tạo ra một đỉnh của đồ thị, sự phân tích tạo ra các cung nối bài toán "lớn" với các bài toán "con". Có hai loại cung nối, cung nối "đơn" chỉ ra rằng bài toán phân tích thành bài toán "dễ hơn", cung nối bội chỉ ra bài toán được phân tích thành một vài bài toán con với nghĩa là "bài toán lớn sẽ được giải quyết nếu tất cả các bài toán con là được giải quyết". Cung nối bội được nối lại với nhau bởi một "cung liên kết".

Quá trình phân tích dừng lại khi đạt tới các bài toán "con" có thể nhận biết là "giải được" hoặc "không giải được".



Quá trình phân tích như vậy xây dựng nên một đồ thị

Đồ thị AND-OR: AO-Graph là cặp < V, E >, trong đó

- V là tập các đỉnh
- Tập các cung E là hợp rời rạc của hai kiểu cung, các "cung AND" A và các "cung OR" O : E = A ∪ O; A ∩ O = ∅
- A, tập các cung AND, mỗi cung AND có nhiều hơn một successor (được biểu diễn bởi "liên kết" các đường nối từ nút xuất đến các successors của nó bởi một "cung tròn"), ý nghĩa là: *nút xuất của một cung AND là "giải được" nếu và chỉ nếu tất cả các successors của nó là "giải được"*
- O, tập các cung OR, mỗi cung OR chỉ có đúng một successor, ý nghĩa là: nút xuất của một cung OR là "giải được" nếu successor của nó là "giải được"

Một bộ các quy tắc phân tích được sử dụng trong quá trình phân tích. Quá trình phân tích có tính đệ quy: ở mỗi bước, xét bài toán đang được phân tích, nếu bài toán được nhận biết là giải được hay không giải được việc phân tích bài toán dừng, nếu không lặp lại việc sử dụng quy tắc phân tích để phân tích bài toán. Sử dụng kỹ thuật phân tích bài toán, ta có thể giải quyết dễ dàng các bài toán lớn.

2.4.2 Các bước giải có thể bỏ lơ hay có thể huỷ bỏ?

Giả sử chúng ta đang chứng minh một định lý toán học, đầu tiên chúng ta tiến hành bằng chứng minh một bổ đề mà chúng ta nghĩ là hữu dụng, nhưng thực tế bổ đề này không giúp ích gì cho việc chứng minh định lý. Điều này có gây phiền phức gì cho việc chứng minh định lý ? Không. "hoạt động chứng minh bổ đề" được gọi là bước có thể bỏ lơ, sự hiện diện của nó không làm ảnh hưởng tới việc tìm chứng minh cho định lý.

Xét bài toán 15-puzzle, sau khi thực hiện một (một vài) bước di chuyển, ta nhận thấy bước di chuyển này không mang lại lợi ích, ta có thể "huỷ bỏ" bước đi này bằng (các) di chuyển ngược lại để nhận lại trạng thái trước khi thực hiện (các) bước di chuyển vô dụng. Cơ chế điều khiển có thể áp dụng cho tình huống này là sử dụng kỹ thuật "lưu vết" : các trạng thái sinh ra của một vài bước di chuyển được lưu lại theo đúng thứ tự sinh ra của chúng, muốn huỷ bỏ một số di chuyển ta phục hồi lại trạng thái trước khi các bước di chuyển này được thực hiện.

Xét bài toán điều khiển robot, sau một bước điều khiển "sai lầm", giả sử robot bị phá huỷ. Bước đi này trở thành "không thể cứu vãn" được. Kỹ thuật được áp dụng cho bài toán "không được phép sai lầm" là "lập kế hoạch"

Một kế hoạch là công thức hoá chi tiết và hệ thống các hành động. Nó là một tập các hành động (toán tử) nhóm lại trong một cấu trúc mệnh lệnh xác định. Cấu trúc có thể là dãy, chọn lựa, lặp, đệ quy, song song, và ở dạng không xác định trước. Một kế hoạch có thể được phân tích thành các kế hoạch con. Kế hoạch con lại có thể được phân tích thành các kế hoạch con nhỏ hơn, …

2.4.3 Vũ trụ có thể tiên đoán?

Trong bài toán đong dầu, mỗi khi thực hiện một toán tử, trạng thái kết quả là xác định chính xác. Điều đó có nghĩa là có thể lập một kế

hoạch tổng thể một dãy các toán tử và biết chắc chắn rằng kết quả của nó sẽ là trạng thái gì. Lập kế hoạch nhằm tránh "huỷ bỏ, làm lại". Tuy nhiên trong quá trình lập kế hoạch có thể không tránh khỏi "quay lui". Cấu trúc điều khiển cho phép quay lui là cần thiết.

Xét trò chơi bài poker, người chơi phải quyết định bỏ đi một số quân bài, thay thế bởi các quân bài mới được rút ngẫu nhiên trong tập các quân bài còn lại với "hy vọng" sẽ được bài tốt hơn. "Toán tử" này không xác định chính xác trạng thái kết quả và do vậy không thể lập được một kế hoạch chính xác, tuy nhiên có thể lập ra một vài kế hoạch và chọn kế hoach có kết quả với ước lượng xác suất cao nhất. Trở ngại lớn nhất là phải trả giá cho việc xây dựng nhiều kế hoạch.

2.4.4 Lời giải tốt nhất hay tương đối?

Xét bài toán trả lời câu hỏi dựa trên một cơ sở dữ liệu các sự kiện sau:

1. Marcus là một con người
2. Marcus là một người Pompei
3. Marcus sinh năm 40 sau công nguyên
4. Mọi người đều phải chết
5. Mọi người Pompei đều bị chết vào năm 79 sau công nguyên do núi lửa phun trào
6. Không người nào sống lâu hơn 150 năm
7. Năm nay là năm 2000 sau công nguyên

Câu hỏi đặt ra "Marcus còn sống không ?"

Ta có thể suy diễn :

1. Marcus sinh năm 40 sau công nguyên (Giả thiết 3.)
2. Năm nay là năm 2000 sau công nguyên (Giả thiết 7.)
3. Tuổi của Marcus bây giờ là 1960 năm (1&2)
4. Không người nào sông lâu hơn 150 (Giả thiết 6.)
5. Marcus không còn sống (3 & 4)

Hoặc

1. Mọi người Pompei đều bị chết vào năm 79 sau CN (Giả thiết 5.)
2. Marcus là nguòi Pompei (Giả thiết 2.)

3. Marcus chết vào năm 79 sau công nguyên (1 & 2)
4. Năm nay là năm 2000 sau công nguyên (Giả thiết 7.)
5. Marcus không còn sống (3 & 4)

Mục tiêu là trả lời câu hỏi nên sau một dãy lập luận, câu hỏi được trả lời. Không có lý do gì phải tìm một dãy lập luận khác. Ví dụ này minh hoạ cho bài toán tìm **một** lời giải.

Xét bài toán TSP (Traveling Saleman Problem): Một người giao hàng cho n cơ sở tại n thành phố. Ông ta xuất phát từ thành phố nơi ông cư trú, đi đến các thành phố để giao hàng (mỗi thành phố đúng một lần) xong việc ông ta quay về thành phố cư trú. Chi phí di chuyển giữa hai thành phố bất kỳ đã biết. Hành trình của người giao hàng phải diễn ra như thế nào để chi phí cho chuyến đi của ông ta là nhỏ nhất?

Giả sử ta tìm được một hành trình. Điều gì đảm bảo là hành trình đó có chi phí nhỏ nhất? Ta chỉ có thể kết luận hành trình đó là tốt nhất, sau khi so sánh nó với tất cả các hành trình có thể khác. Ví dụ này minh hoạ cho bài toán tìm lời giải tối ưu.

Nói chung, bài toán tìm lời giải tối ưu đòi hỏi tính toán nhiều hơn bài toán tìm một lời giải. Bài toán tìm một lời giải có thể sử dụng heuristic để giảm thiểu thời gian tìm kiếm. Không có một heuristic nào đảm bảo tìm thấy lời giải tốt nhất.

2.4.5 Lời giải là một trạng thái hay một đường đi?

Xét bài toán đong dầu, mục tiêu là tìm một dãy các bước chuyên dầu để đạt được một trạng thái đích. Bài toán như vậy đòi hỏi tìm một đường đi lời giải. Trong trường hợp này không tránh khỏi sử dụng kỹ thuật lưu vết.

Xét bài toán dịch từ tiếng anh sang tiếng việt. Nhiệm vụ là tìm từ / cụm từ trong tiếng việt phù hợp với từ/cụm từ trong tiếng anh (phụ thuộc vào phạm trù bài viết, ngữ cảnh, …), rõ ràng con đường để tìm ra từ/cụm từ không giữ vai trò quan trọng trong quá trình dịch. Đòi hỏi của bài toán là một trạng thái. Kỹ thuật lưu vết không cần thiết sử dụng trong trường hợp này.

2.4.6 Vai trò của tri thức?

Xét bài toán đong dầu, lượng tri thức cần thiết cho giải quyết bài toán chỉ là hiểu biết về luật chuyên dầu: để chuyên được dầu từ bình a sang bình b thì bình a phải có dầu và bình b phải chưa đầy, để biết chính xác mỗi bình chứa bao nhiêu lít dầu thì việc chuyên phải được tiến hành theo kiểu hoặc là chuyên đến khi bình b đầy hoặc là chuyên đến khi bình a rỗng.

Xét bài toán dịch một bài viết từ tiếng Anh sang tiếng Việt. Một từ (cụm từ) trong tiếng Anh có thể tương ứng với nhiều từ (cụm từ) trong tiếng Việt, với ý nghĩa rất khác nhau tuỳ theo ngữ cảnh. Để có được bản dịch sát nghĩa, ít nhất cần các kiến thức : hiểu biết về việc gom các từ đơn thành một cụm từ (ví dụ *in order to* nếu tách riêng từng từ mà dịch sẽ không sát nghĩa !), hiểu biết về lĩnh vực mà bài viết thuộc vào (ví dụ trong lĩnh vực toán học, tin học từ *function* thường mang nghĩa là "hàm" chứ không mang nghĩa "chức năng" !), hiểu biết về lịch sử và hoàn cảnh ra đời của bài viết, hiểu biết về cách "chơi chữ" của tác giả bài viết, … một lượng rất lớn tri thức là cần thiết để có được một bài dịch sát nghĩa.

2.4.7 Nhiệm vụ đòi hỏi tương tác với con người?

Xét bài toán đong dầu, sau khi thiết kế chương trình giải bài toán, cho chạy chương trình và nhận được lời giải của bài toán, không cần sự tương tác, trao đổi với con người.

Xét bài toán bán vé tự dộng: Xây dựng một chương trình máy tính bán vé vận chuyển. các thông tin cần thiết là nơi đi, nơi đến của khách hàng, phương tiện khách chọn lựa, thời gian khởi hành, số lượng vé, … Một khách hàng mua vé có thể không cung cấp đầy đủ các thông tin cần thiết, ví dụ khách đưa ra yêu cầu "tôi muốn đi Hà nội", chương trình cần phải hỏi lại khách hàng về nơi xuất phát, phương tiện lựa chọn (máy bay, tàu hoả, tàu thuỷ, xe bus…), về thời gian khởi hành, về số lượng vé cần mua…Trong bài toán này, sự tương tác, trao đổi với con người là không thể tránh khỏi.

2.5 Các đặc trưng của hệ sản xuất

Các hệ sản xuất cung cấp cấu trúc tốt để mô tả và thực hiện quá trình tìm kiếm lời giải cho bài toán. Một bài toán có lời giải, có một tập vô hạn các hệ sản xuất mô tả cách tìm kiếm lời giải. Một số hệ sản xuất bản chất hơn hoặc hiệu quả hơn. Các hệ sản xuất có thể được mô tả bởi một tập

các đặc trưng, các đặc trưng này có thể làm hé lộ chiến lược thực thi thích hợp.

Các định nghĩa về một số đặc trưng của các hệ sản xuất

Hệ sản xuất đơn điệu là hệ sản xuất thoả mãn tính chất "nếu có nhiều quy tắc có thể áp dụng, việc áp dụng một quy tắc được lựa chọn khôngngăn cản việc áp dụng muộn hơn của các quy tắc còn lại".Nếu tính chất này không được thoả mãn, hệ sản xuất được gọi là không đơn điệu.

Hệ sản xuất giao hoán bộ phận là hệ sản xuất thoả mãn tính chất "một dãy các quy tắc, nếu áp dụng nó sẽ làm biến đổi trạng thái x thành trạng thái y, thì mọi hoán vị cho phép của dãy quy tắc này (hoán vị mà các điều kiện của các quy tắc trong dãy hoán vị đều được thoả mãn) khi được áp dụng cũng biến đổi trạng thái x thành trạng thái y".

Hệ sản xuất giao hoán là hệ sản xuất đơn điệu và giao hoán bộ phận.

Đối với các bài toán giải được bất kỳ, có một số vô hạn các hệ sản xuất mô tả cách tìm lời giải. Trong đó có một số hệ tự nhiên hơn hoặc hiệu quả hơn, xác định hệ sản xuất như vậy làm tăng hiệu quả của quá trình tìm kiếm.

Các hệ sản xuất đơn điệu và giao hoán bộ phận là đáng quan tâm dưới quan điểm thực thi vì chúng có thể thực thi không cần quay lui lại các trạng thái trước đó khi phát hiện ra con đường đang đi theo là không đúng. Tuy nhiên, thực thi các hệ thống như vậy với quay lui vẫn là hữu dụng trong việc đảm bảo một sự tìm kiếm hệ thống, cơ sở dữ liệu hiện hành biểu diễn trạng thái bài toán không cần thiết được lưu trữ. Điều này làm tăng đáng kể hiệu quả.

Các hệ sản xuất không đơn điệu, giao hoán bộ phận có thể được dùng cho các bài toán trong đó các thay đổi xảy ra có thể được bảo tồn và thứ tự các phép toán là không quan trọng. Các bài toán thao tác vật lý thường thoả mãn các tính chất này.

Các hệ sản xuất không giao hoán bộ phận có thể được dùng cho những bài toán trong đó các thay đổi không thể đảo ngược được. Ví dụ, bài toán xác định quá trình sản xuất một hợp chất hoá học. các toán tử có thể là "thêm hoá chất x vào bình" hoặc "thay đổi nhiệt độ t độ". các toán tử này gây ra các thay đổi không thể đảo ngược được (hoá chất trong

bình đã bị thay đổi thành chất khác). Lập kế hoạch có thể được dùng để giải quyết các bài toán với các thay đổi không thể đảo ngược và vũ trụ có thể tiên đoán.

2.6 Các vấn đề trong thiết kế chương trình tìm kiếm

Quá trình tìm kiếm có thể được xem như một phép duyệt của một cấu trúc cây trong đó mỗi nút biểu diễn một trạng thái bài toán, mỗi cung biểu diễn toán tử thay đổi trạng thái. Về phương diện lý thuyết, cây này có thể được xây dựng tường minh, và sự tìm kiếm được tiến hành trên nó. Tuy nhiên, cây tìm kiếm tường minh thường rất lớn và hầu như không cần khảo sát. Về phương diện thực tế, các chương trình tìm kiếm, tại một thời điểm của quá trình tìm kiếm, chỉ khảo sát một bộ phận tường minh của cây, người ta gọi nó là cây lời giải tiềm năng. Cây lời giải tiềm năng lớn dần lên theo quá trình tìm kiếm.

Trước khi xây dựng một chương trình tìm kiếm, cần phải xét đến một số vấn đề quan trọng sau :

- Hướng tìm kiếm (suy diễn tiến/lui)
- Chọn quy tắc áp dụng : những quy tắc nào có thể áp dụng vào trạng thái hiện hành ?(tương hợp - matching). Trong các quy tắc có thể áp dụng chọn quy tắc nào là thích hợp ? (giải quyết xung đột – conflict solution)
- Biểu diễn các nút của quá trình tìm kiếm (biểu diễn tri thức và bài toán khung).

Quá trình tìm kiếm được thực hiện trên đồ thị biểu diễn không gian trạng thái. Điều này nảy sinh hai vấn đề, thứ nhất một nút có thể được xét hơn một lần, thứ hai tìm kiếm rơi vào chu trình.

Kỹ thuật để tránh xét lặp lại và chu trình: mỗi khi một nút mới được sinh ra, kiểm tra nút đó có nằm trong tập hợp các nút đã được sinh ra chưa, nếu chưa nó mới được đưa vào tập các nút sẽ được xem xét (kỹ thuật sử dụng Open và Closed). Chú ý rằng kỹ thuật này giúp tránh xét lặp lại và rơi vào chu trình, nhưng phải trả giá bằng phí tổn "kiểm tra mỗi nút mới được sinh ra có thuộc tập các nút đã được sinh ra ?".

Câu hỏi ôn tập và bài tập

1) Cho một ví dụ trong đó tìm kiếm rộng làm việc tốt hơn tìm kiếm sâu.
2) Cho một ví dụ trong đó tìm kiếm sâu làm việc tốt hơn tìm kiếm rộng.
3) Viết chương trình giải quyết bài toán đong dầu, sử dụng tìm kiếm rộng/sâu.
4) Viết chương trình giải quyết bài toán 8-puzzle, sử dụng tìm kiếm rộng/sâu.

Với mỗi bài toán bên dưới,biểu diễn bài toán trong không gian trạng thái và tìm heuristic tốt đối với mỗi bài toán. Chú ý: khi phân tích, cần tôn trọng bảy đặc trưng bài toán.

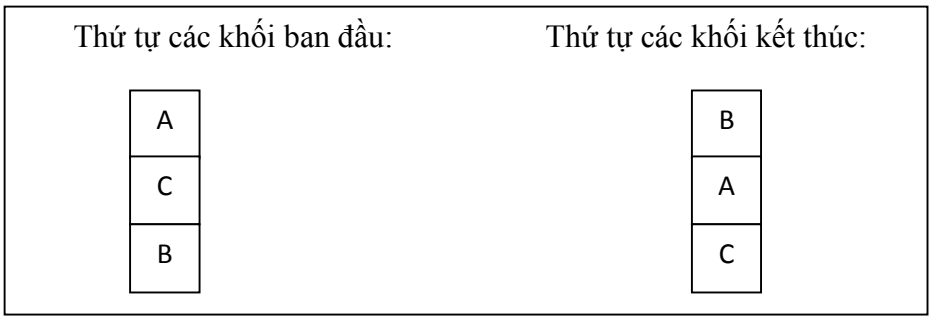
5) Bài toán "đong nước" : Có hai bình không phân độ, một bình dung tích 4 lít, một bình dung tích 3 lít. Ta có thể làm đầy nước trong một bình bằng một vòi nước, ta có thể làm rỗng một bình bằng cách đổ hết nước trong bình đó đi, ta cũng có thể chuyên nước từ một bình sang bình khác theo kiểu "chuyên đầy" hoặc "chuyên rỗng" như trong bài toán "đong dầu". Làm thế nào để nhận được một bình chứa đúng 2 lít nước.
6) Bài toán "Con sói, con dê và cái bắp cải" : Một người nông dân dắt theo ba vật : một con sói, một con dê và một cái bắp cải. Đến một bờ sông, có một con thuyền nhỏ, người nông dân sử dụng con thuyền đó để chuyên chở các vật đi theo mình sang bên kia sông. Mỗi lần chuyên chở, ông chỉ có thể chuyên chở được một vật. Nhưng khó khăn cho ông là, nếu bên một bờ có con sói và con dê / con dê và cái bắp cải mà không có ông ở đó thì con sói ăn thịt con dê / con dê ăn cái bắp cải. Làm thế nào để người nông dân chuyên chở được cả ba vật qua sông an toàn ?
7) Bài toán "Các thày tu và các kẻ ăn thịt người" : ba ông thày tu và ba kẻ ăn thịt người ở trên một bờ sông cùng muốn qua sông. Có một con thuyền nhỏ, một chuyến chỉ có thể chở được hai người.

Nếu trên một bờ nào đó, số kẻ ăn thịt người nhiều hơn số thày tu, chúng sẽ xâu xé các ông thày tu. Làm thế nào để chuyên chở mọi người qua sông mà các ông thày tu không ai bị ăn thịt ?

8) Bài toán "Tháp hà nội" :Trên một cọc A, có ba chiếc đĩa kích thước đôi một khác nhau, được xếp chồng lên nhau theo thứ tự kích thước từ lớn đến nhỏ. Có hai cọc trống B và C. Người ta muốn di rời cọc đĩa từ A sang B. Thao tác di rời : mỗi làn chỉ di rời được một đĩa nằm trên đỉnh của một cọc. khi sắp xếp đĩa lên một cọc chỉ được phép đặt đĩa kích thước nhỏ hơn lên trên đĩa có kích thước lớn hơn. Làm thế nào để thực thi công việc di rời, cho phép sử dụng cọc C làm trung gian ?

9) Bài toán "cờ ca-rô" : Một lưới ô vuông kích thước nxm. Hai người chơi luân phiên đặt vào một ô trống một ký hiệu của mình (chẳng hạn, một người sử dụng ký hiệu X, người kia sử dụng ký hiệu O). Một người chơi sắp được liên tiếp và thẳng hàng (ngang, dọc, chéo) ít nhất năm ký hiệu của mình là người thắng.

10) Bài toán "Thế giới các khối" : Trên bàn có một chồng ba khối gỗ lập phương, mỗi khốiđược ký hiệu bởi các ký tự A, B, C. Một cánh tay robot có thể gắp một khối trên đỉnh của một chồng, đặt nó lên bàn hoặc chồng nó lên đỉnh của một chồng khác. Nhiệm vụ của cánh tay robot là sắp xếp lại thứ tự các khối gỗ theo một thứ tự đã cho. Ví dụ



11) Bài toán "Số học mật mã" :Gán các chữ các bởi các chữ số, thoả mãn điều kiện : hai chữ cái khác nhau được gán với hai chữ số

khác nhau, một chữ cái được gán với đúng một chữ số, sao cho phép tính đã cho là đúng. Ví dụ :

```
  S E N D        D O N A L D        C R O S S
+ M O R E    +   G E R A L D    +   R O A D S
-----------    -------------    -------------
M O N E Y      R O B E R T      D A N G E R
```

TÀI LIỆU THAM KHẢO

[1] Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press.

[2] Newell, A. and Simon, H. A. (1961). GPS, a program that simulates human thought. In Billing, H. (Ed.), *Lernende Automaten*, pp. 109–124. R. Olden- bourg.

[3] Nilsson, N. J. (1971). Problem-Solving Methods in Artificial Intelligence. McGraw-Hill.

[4] Russell, S.and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach,*(third edition), Prentice Hall.

[5] Slate, D. J. and Atkin, L. R. (1977). CHESS 4.5— Northwestern University chess program. In Frey, P. W. (Ed.), *Chess Skill in Man and Machine*, pp. 82–118. Springer-Verlag.

[6] Slocum, J. and Sonneveld, D. (2006). *The 15 Puzzle*. Slocum Puzzle Foundation.

CHƯƠNG 3
KỸ THUẬT TÌM KIẾM NÂNG CAO

Trong chương hai chúng ta đã tìm hiểu phương pháp giải quyết vấn đề bằng phương pháp tìm kiếm. Các kỹ thuật tìm kiếm ở chương hai chỉ sử dụng duy nhất thông tin của bài toán. Trong chương này chúng ta sẽ tiếp tục phân tích các kỹ thuật tìm kiếm nâng cao. Chúng ta bắt đầu bằng kỹ thuật tìm kiếm với thông tin liên quan đến bài toán nhưng không được định nghĩa trong bài toán. Phương pháp tìm kiếm cục bộ sẽ được đề cập ngay sau đó. Chương này được kết thúc bằng việc mở rộng kỹ thuật tìm kiếm bằng cách xem xét cấu trúc của các trạng thái. Tiêu biểu cho kỹ thuật này là giải thuật giải bài toán thoả mãn ràng buộc.

3.1 Tìm kiếm với thông tin bổ sung

Tìm kiếm với thông tin bổ sung (informed search) hay tìm kiếm heuristic sử dụng các kiến thức có liên quan đến vấn đề nhưng không được định nghĩa trong bài toán.

Tiếp cận tổng quát mà ta xem xét có tên gọi *tìm kiếm tốt nhất đầu tiên* (best-first search). Tìm kiếm tốt nhất đầu tiên có thể là một thể hiện của giải thuật tìm kiếm trên cây (`treeSearch`) hay tìm kiếm trên đồ thị (`graphSearch`) tổng quát. Việc chọn nút để triển khai dựa trên một *hàm lượng giá* (evaluation function): $f(n)$. Hàm lượng giá được xây dựng dựa trên việc ước lượng chi phí và nút có giá ước lượng nhỏ nhất được ưu tiên triển khai trước. Cài đặt giải thuật tìm kiếm tốt nhất đầu tiên trên đồ thị y hệt như tìm kiếm chi phí đồng nhất ngoại trừ việc sử dụng $f(n)$ thay vì $g(n)$ để sắp xếp thứ tự các nút trong hàng đợi ưu tiên.

Giải thuật tìm kiếm tốt nhất đầu tiêu được trình bày trong hình 3.1 Đường biên `frontier` là một hàng đợi ưu tiên. Các phần tử trong đường biên được sắp xếp theo $f(n)$. Giải thuật bắt đầu từ nút gốc chứa trạng thái khởi tạo. Mỗi lần lặp, giải thuật chọn nút n có $f(n)$ nhỏ nhất trên đường biên và triển khai nó. Với mỗi nút con $child$, nếu trạng thái của nó, `child->state`, chưa có trong `frontier` lẫn `explorer`, ta sẽ đưa child vào đường biên. Nếu `child->state` đã tồn tại trên đường biên và nút chứa nó có giá trị f lớn hơn giá trị f của `child`, ta cập nhật giá trị f của nút tương ứng trên đường biên (trường hợp (*) của giải thuật trong hình 3.1). Nếu `child->state` nằm trong `explored`