TRƯỜNG ĐẠI HỌC SƯ PHẠM HÀ NỘI

KHOA CÔNG NGHỆ THÔNG TIN

BÀI TẬP LỚN

BÀI TOÁN 8-PUZZLE

Nhóm: **Trần Thị Hà An (K60C)**

Phan Tuấn Anh (K60C)

Nguyễn Phương Dung (K60C)

Tạ Chí Hiếu (K60C)

Bùi Hải Lý (K60B)

Lê Hà Minh (K60B)

Nguyễn Thị Tươi (K60C)

Hà Nội, 8/5/2013

Lời mở đầu

Trong ngành khoa học máy tính, một giải thuật tìm kiếm là một thuật toán lấy đầu vào là một bài toán và trả về kết quả là một lời giải cho bài toán đó, thường là sau khi cân nhắc giữa một loạt các lời giải có thể. Tập hợp tất cả các lời giải có thể cho bài toán được gọi là không gian tìm kiếm. Có những thuật toán tìm kiếm "sơ đẳng" không có thông tin, đây là những phương pháp đơn giản và trực quan, trong khi đó các thuật toán tìm kiếm có thông tin sử dụng hàm đánh giá heuristic giúp ta giảm đáng kể thời gian cần thiết cho việc tìm kiếm lời giải.

Để áp dụng được các giải thuật tìm kiếm, ta cần chuyển không gian tìm kiếm về dạng đồ thị. Với dạng đồ thị ta sẽ nắm bắt những mối liên hệ, những ảnh hưởng giữa các trạng thái của bài toán một cách nhanh chóng và ngắn gọn. Trong phạm vi bài báo cáo, chúng em xin trình bày ba thuật tìm kiếm toán cơ bản và tiêu biểu với lý thuyết đồ thị đó là*: Tìm kiếm theo chiều rộng, Tìm kiếm theo chiều sâu* và *Tìm kiếm A**. Qua đó, chúng em sẽ áp dụng giải thuật tìm kiếm A* để giải bài toán 8 puzzle, một bài toán quen thuộc với những người lập trình, chúng em sẽ đưa ra cơ chế của thuật toán, ưu nhược điểm cũng như độ phức tạp của những thuật toán trên. Bài báo cáo gồm các nội dung chính sau:

Phần 1 – Tổng quan*: Giới thiệu về bài toán 8 puzzle. Miêu tả bài toán, nêu đặc điểm và phân tích để hình dung ra hướng đi cho thuật toán áp dụng cho bài toán,*

Phần 2 – Các thuật toán*: Trình bày về 3 thuật toán chúng em tìm hiểu, nêu ra cơ chế và ưu nhược điểm của mỗi thuật toán.*

Phần 3 – Tổng kết

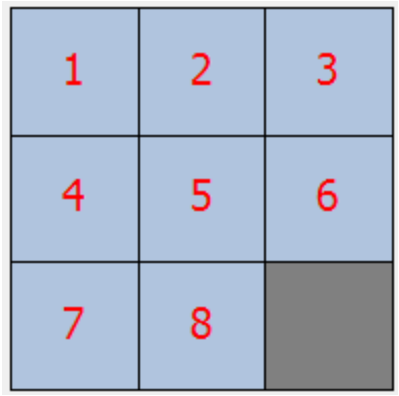
Báo cáo có thể còn có nhiều thiếu xót, chúng em rất mong nhận được sự góp ý của thầy và các bạn. Chúng em xin chân thành cảm ơn!

I. TỔNG QUAN:

1. Giới thiệu bài toán:

Bài toán 8-puzzle (hay còn gọi là 8 số) là một bài toán quen thuộc với những người bắt đầu tiếp cận với môn Trí tuệ nhân tạo. Bài toán có nhiều phiên bản khác nhau dựa theo số ô, như 8-puzzle, 15-puzzle,… ở mức độ đơn giản nhất, chúng em xem xét dạng bài toán 8-puzzle.

Bài toàn gồm một bảng ô vuông kích thước 3x3, có tám ô được đánh số từ 1 tới 8 và một ô trống. Trạng thái ban đầu, các ô được sắp xếp một cách ngẫu nhiên, nhiệm vụ của người chơi là tìm cách đưa chúng về đúng thứ tự như hình dưới:



Trong quá trình giải bài toàn, tại mỗi bước, ta giả định chỉ có ô trống là di chuyển, như vậy, tối đa ô trống có thể có 4 khả năng di chuyển (lên trên, xuống dưới, sang trái, sang phải).

2. Điều kiện của trạng thái đầu:

Có những trạng thái của bảng số không thể chuyển về trạng thái đích. Người ta chứng minh được rằng, để có thể chuyển từ trạng thái đầu tới trạng thái đích, thì trạng thái đầu này phải thỏa mãn điều kiện được xác định như sau:

Ta xét lần lượt từ trên xuống dưới, từ trái sang phải, với mỗi ô số đang xét (giả sử là ô thứ i), ta kiểm tra xem phía sau có bao nhiêu ô số có giá trị nhỏ hơn ô đó. Sau đó ta tính tổng $N = n_1 + n_2 + \ldots + n_8$.

Ta có quy tắc chung sau cho bài toán n-puzzle:

- **Nếu số ô vuông lẻ:**
  N mod 2 = 0 (1)

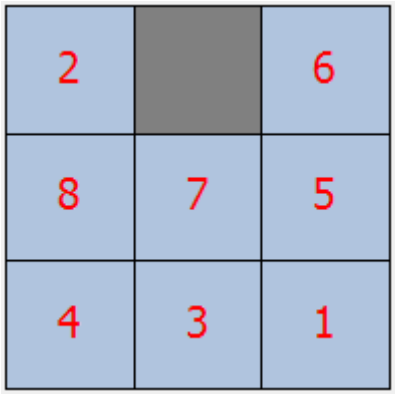
- **Nếu số ô vuông chẵn:**

  N mod 2 = 0 và ô trống phải nằm ở hàng chẵn xét từ trên xuống. (2)

  N mod 2 = 1 và ô trống phải nằm ở hàng lẻ xét từ trên xuống. (3)

Cụ thể, ta đang xét bài toán 8-puzzle – có 9 ô vuông nên trạng thái đầu phải thỏa mãn điều kiện (1).

Ví dụ: Cho trạng thái đầu sau 2-0-6-8-7-5-4-3-1



Xét ô thứ nhất có giá trị 2: Phía sau có *1* ô nhỏ hơn (1) => $n_1 = 1$

Xét ô thứ hai có giá trị 6: Phía sau có *4* ô nhỏ hơn (5,4,3,1) => $n_2 = 4$

Xét ô thứ ba có giá trị 8: Phía sau có *5* ô nhỏ hơn (7,5,4,3,1) => $n_3 = 5$

Xét ô thứ tư có giá trị 7: Phía sau có *4* ô nhỏ hơn (5,4,3,1) => $n_4 = 4$

Xét ô thứ năm có giá trị 5: Phía sau có *3* ô nhỏ hơn (4,3,1) => $n_5 = 3$

Xét ô thứ sáu có giá trị 4: Phía sau có *2* ô nhỏ hơn (3,1) => $n_6 = 2$

Xét ô thứ bảy có giá trị 3: Phía sau có *1* ô nhỏ hơn (1) => $n_7 = 1$

Xét ô thứ tám có giá trị 1: Phía sau không còn ô nào nhỏ hơn => $n_8 = 0$

$$N = 1 + 4 + 5 + 4 + 3 + 2 + 1 + 0 = 20$$

Ta có 20 mod 2 = 0 => Thỏa mãn.

Những trạng thái của bảng số mà có thể chuyển về trạng thái đích gọi là cấu hình hợp lệ, ngược lại gọi là cấu hình không hợp lệ.

Với bảng số kích thước *mxm* (m là cạnh) thì ta có không gian trạng thái là *(mxm)!* Với bài toán 8-puzzle, các trạng thái có thể có của bảng số là *(3x3)! = 362880*. Nếu m tăng lên 1, sẽ làm không gian trạng thái tăng lên rất lớn, do vậy các phiên bản bài toán với m>3 ít khi được áp dụng.

Trong phạm vi bài báo cáo, chúng em tìm hiểu về 3 phương pháp tìm kiếm lời giải cho bài toán 8-puzzle, đó là: Tìm kiếm theo chiều rộng, tìm kiếm theo chiều sâu, và tìm kiếm A*. Hai phương pháp đầu là những phương pháp tìm kiếm không có thông tin, A* là phương pháp tìm kiếm có thông tin. Cụ thể mỗi phương pháp sẽ được trình bày ngay sau đây.

II. THUẬT TOÁN TÌM KIẾM KHÔNG CÓ THÔNG TIN:

Thuật toán tìm kiếm theo chiều sâu và chiều rộng là hai thuật toán tìm kiếm mù phổ biến, thường được sử dụng trong lý thuyết đồ thị. Chúng ta sẽ đi vào từng thuật toán từ tư tưởng của thuật toán cho tới giả mã và bước đi trong thuật toán để làm rõ hơn cách thức hoạt động của thuật toán. Trong quá trình tìm hiểu ta sẽ nhận thấy chúng có nhiều điểm tương đồng trong cách thực hiện, nhưng cách tổ chức thì khác nhau. Với mỗi thuật toán ta sẽ đưa ra ưu nhược điểm của chúng để có thể sử dụng chúng phù hợp hơn theo những yêu cầu riêng của bài toán đầu vào.

1. Thuật toán tìm kiếm theo chiều sâu (Depth First Sreach):

Trước tiên ta sẽ đi vào tìm hiểu về thuật toán *tìm kiếm theo chiều sâu (DFS)* . Thuật toán DFS là một quá trình duyệt hay tìm kiếm trên một cây hoặc một đồ thị. Thuật toán DFS sẽ bắt đầu với một đỉnh gốc và phát triển sâu và xa nhất có thể của mỗi nhánh. Để hiểu hơn ta sẽ đi vào từng phần trong thuật toán:

a) Tư tưởng của thuật toán:

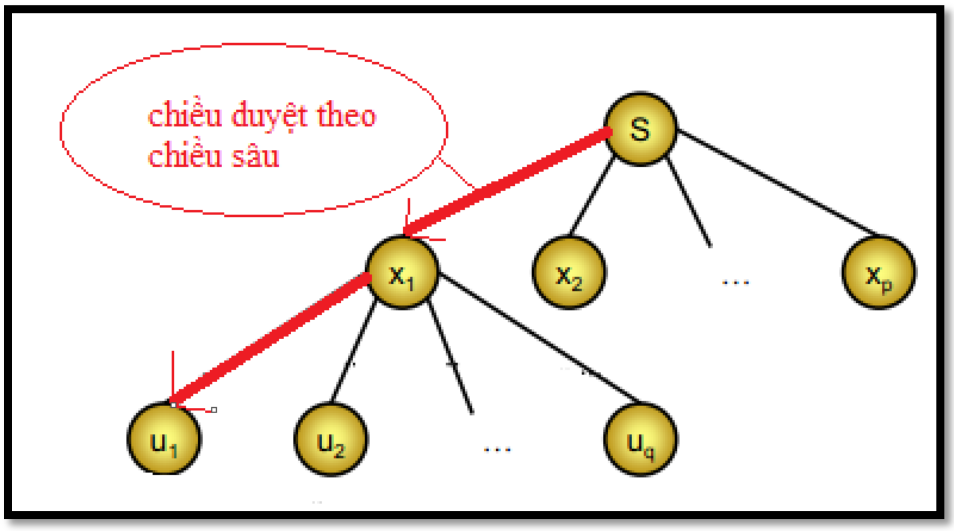
Với tư tưởng đi sâu vào từng nhánh, ta giả sử đầu vào của thuật toán là một đồ thị G = (V,E). Coi s là đỉnh gốc của V, ta sẽ bắt đầu quá trình duyệt với s.Từ s ta sẽ đi

thăm tới đỉnh kề với s (giả sử ở đây là u_0), từ u_0 ta tiếp tục quá trình duyệt với các đỉnh kề u_0 (trừ các đỉnh đã thăm). Quá trình sẽ tiếp tục tới khi gặp đỉnh cần tìm hoặc đi hết nhánh thì thực hiện lùi lại đỉnh trước đó.

Xét một cách tổng quát thì khi xét một đỉnh u_0 ta sẽ có hai khả năng xảy ra:

- Nếu như tồn tại đỉnh v_0 kề với u_0 mà chưa được thăm thì đỉnh v_0 đó sẽ được đánh dấu để trở thành đỉnh đã thăm và quá trình tìm kiếm sẽ bắt đầu từ đỉnh v_0 đó.
- Ngược lại, nếu mọi đỉnh kề với u_0 đều đã thăm thì ta sẽ quay lại đỉnh mà trước đó ta đến đỉnh u_0 để tiếp tục quá trình tìm kiếm

Như vậy trong quá trình thăm đỉnh bằng thuật toán tìm kiếm theo chiều sâu, ta nhận thấy ngay đỉnh càng thăm muộn thì sớm được duyệt trước (đây là cơ chế Last In First Out) . Do đó ta có thể sử dụng thủ tục đệ quy hoặc sử dụng một danh sách kiểu ngăn xếp để tổ chức cho quá trình duyệt của thuật toán. Dưới dây là minh họa cho quá trình duyệt với một cây:



Hình trên minh họa thứ tự duyệt của thuật toán tìm kiếm theo chiều sâu: ta nhận thấy từ S sẽ thăm X1, tiếp tục tới u1 là kề với X1. Nếu u1 không phải đỉnh tìm và

hết nhánh thì sẽ lùi về X1 để thăm u2. Do đó quá trình duyệt sẽ là: S→x1→u1→u2 →... →uq →x2 →...

b) Giải thuật của thuật toán:

Xác định bài toán ta cần lấy ra input và output của bài toán như sau:

- ***Input:*** *đồ thị vào G=(V,E) với đỉnh gốc là s_0(Trạng thái đầu)*

  Tập đích Goals

- ***Output:*** *một đường đi p từ s đến một đỉnh ftrong tập đích Goals*

Thuật toán DFS có 2 cách để duyệt những đỉnh trong quá trình tìm kiếm đó là sử dụng thủ tục đệ quy hoặc sử dụng ngăn xếp để lưu trữ các đỉnh sẽ duyệt tiếp đó. Ta sẽ đi vào cách sử dụng ngăn xếp để lưu trữ các đỉnh.Ta có các bước cho quá trình thực hiện thuật toán như sau:

Bước 1: Khởi tạo

- Các đỉnh đều ở trạng thái chưa đánh dấu, trừ đỉnh xuất phát s là đã đánh dấu.
- Một ngăn xếp S (Stack)ban đầu chỉ đưa vào có một phần tử là s. Bằng việc sử dụng ngăn xếp lưu các đỉnh ta sẽ duyệt sâu vào từng nhánh của đồ thị.

Bước 2: Lặp lại các bước sau cho đến khi ngăn xếp rỗng:

- Nếu ngăn xếp rỗng, không thấy đỉnh đích, thông báo "*không tìm thấy*", dừng.
- Ngăn xếp không rỗng, lấy u ra khỏi ngăn xếp, thông báo thăm u (bắt đầu duyệt đỉnh u, nếu lần đầu thì là u chính là s).
- Kiểm tra u có phải đỉnh đích t không:
  - Nếu đúng trả về u, dừng vòng lặp, chuyển sang bước 3.

- Nếu không đúng thì tiếp tục duyệt.

- Xét tất cả các đỉnh v kề với u mà chưa được đánh dấu, với mỗi đỉnh v đó:
  - Đánh dấu v
  - Ghi nhận đường đi từ v đến u
  - Đẩy v vào ngăn xếp (v sẽ chờ được duyệt tại những bước sau)

Bước 3: Truy ngược lại đường đi (nếu có)

Giả mã:

For (mọi v thuộc V) **do** Free[v] := **True**;

Free[s] :=**False** {khởi tạo ban đầu chỉ có đỉnh s là bị đánh dấu}

Stack := rỗng; Push(s); { Khởi tạo ngăn xếp ban đầu chỉ gồm một đỉnh s}

Repeat {lặp tới khi ngăn xếp rỗng rỗng}

u := Pop; {lấy từ ngăn xếp ra một đỉnh u}

For (với mọi v thuộc V : Free[v] **and** ((u. v) thuộc E)) **do**

{xét những đỉnh v kề u chưa vào ngăn xếp}

Begin

Trace[v] := u; { Lưu vết đường đi}

Free[v] := False ; {đánh dấu v}

Push(v); {đẩy v vào ngăn xếp}

End;

Until Stack= rỗng;

{thông báo từ s có thăm được tới đỉnh v mà Free[v] = False};

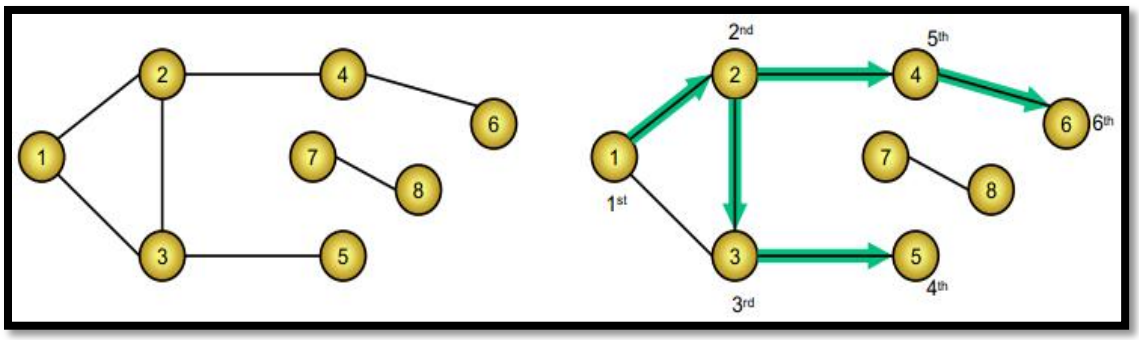
If Free[f] **then** {s đi tới được f}

{truy theo vết từ f để tìm đường từ s → f};

c) Nhận xét:

– Có thể có nhiều đường đi từ s→ f, nhưng thuật toán DFS luôn trả về một đường đi có thứ tự từ điển nhỏ nhất.

– Quá trình tìm kiếm theo chiều sâu cho ta một cây DFS gốc s. Quan hệ cha-con trên cây được định nghĩa là: nếu từ đỉnh u tới thăm đỉnh v thì u là nút cha của nút v. Hình dưới sẽ minh họa cho cây DFS tương ứng với đỉnh xuất phát s=1



Ưu điểm

- Nếu bài toán có lời giải, phương pháp tìm kiếm theo chiều sâu đảm bảo tìm ra lời giải.
- Ký thuật tìm kiếm sâu tập trung vào đích, con người cảm thấy hài lòng khi các câu hỏi tập trung vào vấn đề chính.
- Do cách tìm của kỹ thuật này, nếu lời giải ở rất sâu, kỹ thuật sâu sẽ tiết kiệm thời gian

Nhược điểm

- Tìm sâu khai thác không gian bài toán để tìm lời giải theo thuật toán đơn giản một cách cứng nhắc. Trong quá trình tìm nó không có thông tin nào để phát hiện lời giải. Nếu cọn nút ban đầu không thích hợp có thể không dẫn tới đích của bài toán.
- Không phù hợp với không gian bài toán lớn, kỹ thuật tìm kiếm sâu có thể không đi đến lời giải trong khoảng thời gian vừa phải (nếu cố định thời gian).

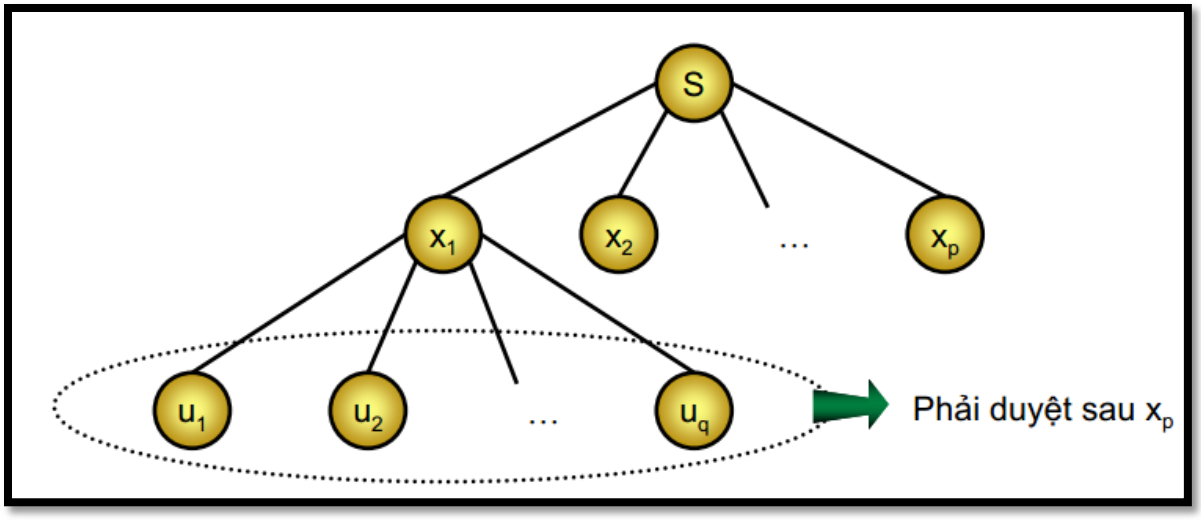
2. Thuật toán tìm kiếm theo chiều rộng (Breadth First Sreach):

Tương tự thuật toán tìm kiếm DFS thì thuật toán BFS cũng là một thuật toán phổ biến trong việc tìm kiếm trong đồ thị. Nhưng có đôi chút khác biệt về cách tổ chức các đỉnh để duyệt so với thuật toán DFS. Do đó cách duyệt của BFS cũng trở nên khác DFS. Để thấy sự khác nhau này ta sẽ đi vào tìm hiểu hơn vào thuật toán.

a) Tư tưởng của thuật toán

Ta giả sử đầu vào thuật toán là một đồ thị G = (V, E), ta sẽ phải thực hiện lập lịch duyệt cho các đỉnh của đồ thị G. Việc duyệt các đỉnh sẽ được ưu tiên sao cho đỉnh nào gần với nó nhất sẽ được duyệt trước. Tức là nó bắt đầu từ mức thấp nhất của không gian bài toán, sẽ duyệt theo chiều từ trái sang phải hoặc ngược lại ở mức tiếp theo, nếu không thấy lời giải ở mức này nó sẽ chuyển xuống mức kế để tiếp tục...cứ như vậy đến khi tìm được lời giải (nếu có). Ta xét ví dụ sau:

Ví dụ : Bắt đầu ta thăm đỉnh S. Việc thăm đỉnh S sẽ phát sinh thứ tự duyệt những đỉnh (x[1], x[2], .. ,x[p]) kề với S (những đỉnh gần S nhất). Khi thăm đỉnh x[1] sẽ lại phát sinh yêu cầu duyệt những đỉnh (u[1], u[2], ..., u[q]) kề với x[1]. Nhưng rõ ràng các đỉnh u này xa S hơn những đỉnh x nên chúng chỉ được duyệt khi tất cả các đỉnh x đã được duyệt xong. Tức là thứ tự duyệt đỉnh sau khi đã thăm x[1] sẽ là : (x[2], x[3], ..., x[p], u[1], u[2],..., u[q])



Hình trên minh họa thứ tự duyệt của thuật toán tìm kiếm theo chiều rộng: ta nhận thấy quá trình duyệt của đồ thị sẽ là S→x1 →x2 →... →xp →u1 → u2 →...

b) Giải thuật thuật toán:

- ***Input:*** *cây đồ thị G= (V,E) với đỉnh gốc là* s_0 ***(*** *trạng thái đầu****)***

  Tập đích Goals.

- ***Output:*** *một đường đi p từ* n_0 *đến 1 đỉnh f trong tập Goals.*

Thuật toán sử dụng một cấu trúc dữ liệu là *hàng đợi* (Queue) để lưu trữ thông tin trung gian trong quá trình tìm kiếm (ở đây dễ hiểu là các đỉnh kế tiếp đợi được duyệt). Tương tự với tìm kiếm theo chiều sâu, các bước cho giải thuật tìm kiếm theo chiều rộng như sau :

Bước 1: Khởi tạo

- Các đỉnh đều ở trạng thái chưa đánh dấu, trừ đỉnh xuất phát s là đã đánh dấu.
- Một hàng đợi Q (tổ chức dạng hàng đợi Queue), ban đầu chỉ có một phần tử là s. Hàng đợi dùng để chứa các đỉnh sẽ được duyệt theo thứ tự ưu tiên chiều rộng.

Bước 2: Lặp lại các bước sau cho đến khi hàng đợi rỗng:

- Nếu hàng đợi rỗng, không thấy đỉnh đích, thông báo " *không tìm thấy"* , dừng.
- Hàng đợi không rỗng, lấy u ra khỏi hàng đợi , thông báo thăm u (bắt đầu duyệt đỉnh u, nếu là lần duyệt đầu thì u ở đây là s).
- Kiếm tra u có phải đỉnh đích t không
    - Nếu đúng trả về u, dừng vòng lặp, sang bước 3
    - Nếu sai tiếp tục chương trình.
- Xét tất cả các đỉnh v kề với u mà chưa được đánh dấu, với mỗi đỉnh v đó:
    - Đánh dấu v
    - Ghi nhận đường đi từ v đến u
    - Đấy v vào hàng đợi (v sẽ chờ được duyệt tại những bước sau).

Bước 3: Truy ngược lại đường đi

<u>Giả mã:</u>

For (mọi v thuộc V) **do** Free[v] := **True**;

Free[s] :=**False** {khởi tạo ban đầu chỉ có đỉnh s là bị đánh dấu}

Queue := rỗng; Push(s); { Khởi tạo hàng đợi ban đầu chỉ gồm một đỉnh s}

Repeat {lặp tới khi hàng đợi rỗng}

u := Pop; {lấy từ hàng đợi ra một đỉnh u}

For (với mọi v thuộc V : Free[v] **and** ((u. v) thuộc E)) **do**

{xét những đỉnh v kề u chưa vào hàng đợi}

Begin

Trace[v] := u; { Lưu vết đường đi}

Free[v] := False ; {đánh dấu v}

Push(v); {đẩy v vào hàng đợi}

End;

Until Queue = rỗng;

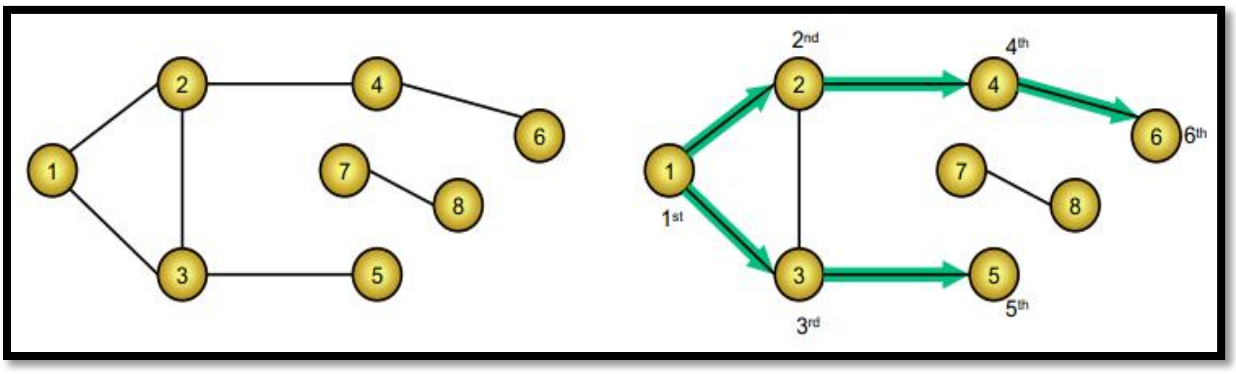
{thông báo từ s có thăm được tới đỉnh v mà Free[v] = False};

If Free[f] **then** {s đi tới được f}

{truy theo vết từ f để tìm đường từ s → f};

c) Nhận xét :

- Có thể có nhiều đường đi từ s tới f nhưng thuật toán BFS luôn trả về một đường đi ngắn nhất (theo nghĩa đi qua ít cạnh nhất).
- Quá tình tìm kiếm theo chiều rộng cho ta một cây BFS gốc s. Quan hệ cha – con trên cây được định nghĩa là : nếu từ đỉnh u tới thăm đỉnh v thì u là nút cha của nút v. Hình biểu diễn về cây BFS:



Ưu điểm

- Kỹ thuật tìm kiếm rộng là kỹ thuật vét cạn không gian trạng thái bài toán vì vậy sẽ tìm được lời giải nếu có
- Đường đi tìm được thỏa mãn đi qua ít đỉnh nhất.

Nhược điểm

- Tìm kiếm lời giải theo thuật toán đã định trước, do vậy tìm kiếm một cách máy móc; khi không có thông tin bổ trợ cho quá trình tìm kiếm, không nhận ra ngay lời giải.
- Không phù hợp với không gian bài toán có kích thước lớn. Đối với loại bài toán này thì phương pháp tìm kiếm chiều rộng đối diện với các khó khăn về nhu cầu:
    - Cần nhiều bộ nhớ theo số nút cần lưu trữ.
    - Cần nhiều công sức xử lý các nút, nhất là khi các nhánh cây dài, số nút tăng.
    - Dễ thực hiện các thao tác không thích hợp , thừa, đưa đến việc tăng đáng kể số nút phải xử lý.
- Không hiệu quả nếu lời giải ở sâu. Phương pháp này không phù hợp cho trường hợp có nhiều đường dẫn đến kết quả nhưng đều sâu.
- Giao tiếp với người dùng không thân thiện. Do duyệt qua tất cả các nút, việc tìm kiếm không tập trung vào một chủ đề.

3. Độ phức tạp của BFS và DFS:

Quá trình tìm kiếm trên đồ thị bắt đầu từ một đỉnh có thể thăm tất cả các đỉnh còn lại, khi đó cách biểu diễn đồ thị có ảnh hưởng lớn tới chi phí về thời gian thực hiện giải thuật:

- ❖ Trong các trường hợp ta biểu diễn đồ thị bằng danh sách kề, cả hai thuật toán BFS và DFS đều có độ phức tạp tính toán là O(n+m) = O(max(n,m)). Đây là cách cài đặt tốt nhất.
- ❖ Nếu ta biểu diễn đồ thị bằng ma trận kề thì độ phức tạp tính toán trong trường hợp này là $O(n+n^2) = O(n^2)$.
- ❖ Nếu ta biểu diễn đồ thị bằng danh sách cạnh, thao tác duyệt những đỉnh kề với đỉnh u sẽ dẫn tới việc duyệt toàn bộ danh sách cạnh, đây là cách cài đặt tồi tệ nhất, nó có độ phức tạp tính toán là O(n.m).

III. TÌM KIẾM A*:

1. Ý tưởng chung:

Hai phương pháp tìm kiếm theo chiều rộng và theo chiều sâu, mặc dù đều có khả năng cho ta kết quả của bài toán, tuy nhiên đây là dạng tìm kiếm không có thông tin, đi theo một lối đi định sẵn vì thế thời gian tìm kiếm lời giải có thể rất lâu. Ta sẽ cải tiến việc giải bài toán bằng cách sử dụng phương pháp tìm kiếm có thông tin A*. Tìm kiếm A* là một ví dụ của tìm kiếm theo lựa chọn tốt nhất (Best-first search) – một thuật toán tìm kiếm tối ưu hóa theo độ sâu bằng cách mở rộng nút "hứa hẹn" nhất có thể dẫn đến đích.

Triển vọng của một nút (mỗi nút tương ứng với một trạng thái của bài toán) được ước lượng bằng một hàm đánh giá heuristic. Hàm này có thể là số lượng ô đặt sai vị trí (h_1) hoặc là tổng khoảng cách Manhattan (h_2).

Ví dụ: Có trạng thái bài toán sau:

1	3	4
8	5	
7	6	2

1	2	3
4	5	6
7	8	

Trạng thái đầu *Trạng thái đích*

- Có 5 ô số nằm sai vị trí so với trạng thái đích

  => $h_1 = 5$

- Tính khoảng cách Mahattan, xác định tọa độ (dòng, cột) của ô khi ở sai vị trí và khi ở đúng vị trí:
  - Ô số 3 khi ở sai vị trí có tọa độ (0,1)
  - Ô số 3 khi ở đúng vị trí có tọa độ (0,2)

  => Khoảng cách Mahattan của ô số 3 bằng $|0 - 0| + |2 - 1| = 1$

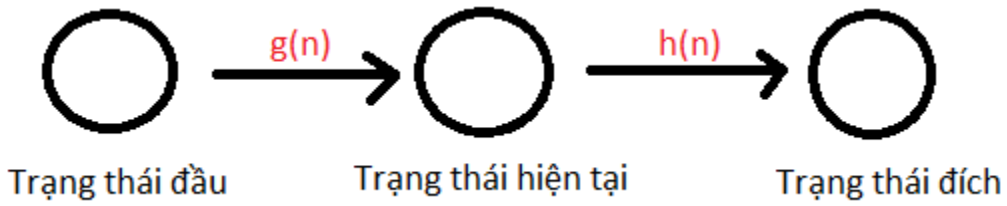
  Tương tự, ô số 4 có khoảng cách Mahattan là 3, ô số 8 có khoảng cách Mahattan là 2, ô số 6 có khoảng cách Mahattan là 2, ô số 2 có khoảng cách Mahattan là 3. Những ô số 1, 5, 7 đã ở đúng vị trí nên có khoảng cách Mahattan là 0.

  $=> h_2 = 0 + 1 + 3 + 2 + 0 + 0 + 2 + 3 = 11$

Điểm khác biệt của A* so với tìm kiếm theo lựa chọn tốt nhất đó là nó còn xét tới đoạn đường đã đi qua. Ta có:

$$\mathbf{f(n) = g(n) + h(n)}$$

Trong đó, g(n) là chi phí đi từ trạng thái đầu đến trạng thái đang xét, h(n) là hàm heuristic ước lượng chi phí đi từ trạng thái hiện tại tới đích. Hàm f(n) có giá trị càng thấp thì trạng thái đó càng có độ ưu tiên và triển vọng cao.



2. Giả thuật bài toán:

Ban đầu ta có Open là tập chứa các trạng thái chưa được xét (sắp xếp theo f tăng dần), Close là tập các trạng thái đã được xét. Ban đầu Open chỉ chứa trạng thái ban đầu, tập Close rỗng.

Begin

Open:={Start};

Close:=∅;

While (Open <> ∅) **do**

Begin

X=Retrieve(Open); {Chọn X sao cho f(X) đạt là nhỏ nhất}

If (X=Goal) **then return** True

Else

Begin

Sinh ra các trạng thái con của X;

For mỗi nút con Y của X **do**

If (Y ∉ Open) và (Y ∉ Close)

Begin

Tính f(Y);

Open = Open ∪ {Y};

End;

If (Y ∈ Open)

If (g(Y) < g(Y')) cập nhật lại giá trị f(Y'), đặt cha của Y' là X;

If (Y ∈ Close)

If (g(Y) < g(Y')) cập nhật lại giá trị f(Y') , đặt cha của Y' là X, cập nhật lại giá trị f và g của tất cả các con của Y đã có trong Open và Close.

End;

Close = Close ∪ {X};

End;

Return False;

End.

3. Nhận xét:

Cũng giống như 2 thuật toán tìm kiếm trước, A* có tính "đầy đủ" theo – có nghĩa nó sẽ luôn tìm thấy lời giải nếu bài toán đó có lời giải.

Bên cạnh đó A* còn "tối ưu". Muốn A* tối ưu thì hàm h(n) phải có tính chấp nhận được – tức là nó không bao giờ đánh giá cao hơn chi phí nhỏ nhất thực sự của việc đi tới đích (trong trường hợp tập đóng). Nếu không dùng tập đóng, thì h(n) phải có tính đơn điệu – tức là nó không bao giờ đánh giá chi phí đi từ một nút tới một nút kề nó cao hơn chi phí thực. Một cách hình thức, nếu m là nút kế tiếp của n thì:

$$\mathbf{h(n) \leq g(m) - g(n) + h(m)}$$

Không có thuật toán nào cũng sử dụng hàm đánh giá heuristic đó mà chỉ phải mở rộng ít nút hơn A*, A* có tính hiệu quả một cách tối ưu. Tuy nhiên, ta không đảm bảo A* luôn chạy nhanh hơn các thuật toán khác.

4. Độ phức tạp:

Độ phức tạp của A* phụ thuộc vào hàm đánh giá h(n). Độ phức tạp đó sẽ là hàm đa thức nếu h(n) thỏa mãn điều kiện sau:

$$|h(n) - h^*(n)| \leq O(\log h^*(n))$$

Với h*(n) là hàm cho kết quả là chi phí chính xác để đi từ nút đang xét n tới đích. Cụ thể, sai số của h(n) không nên tăng nhanh hơn log h*(n). Trong trường hợp xấu nhất, độ phức tạp sẽ lên đến hàm mũ, vấn đề sử dụng bộ nhớ của A* khá rắc rối, nó phải ghi nhớ số lượng nút tăng theo hàm mũ. Một số biến thể của A* được phát triển để giải quyết vấn đề này.

IV. TỔNG KẾT

Như vậy, mỗi một phương pháp đều có ưu và nhược điểm riêng, tuy nhiên giải thuật A* áp dụng cho bài toán 8-puzzle là hiệu quả hơn cả. Giải thuật A* là thuật toán tìm kiếm trong đồ thị có thông tin phản hồi, sử dụng đánh giá heuristic để xếp loại từng nút và duyệt nút theo hàm đánh giá này. Với các dạng bài toán tìm kiếm trên không gian trạng thái, có hai trường hợp cần tới heuristic:

- Những vấn đề không thể có nghiệm chính xác do các mệnh đề không phát biểu chặt chẽ hay do thiếu dữ liệu để khẳng định kết quả.
- Những vấn đề có nghiệm chính xác nhưng chi phí tính toán để tìm ra nghiệm là quá lớn (dẫn đến bùng nổ tổ hợp).

⇨ Heuristic giúp ta tìm kiếm đạt kết quả với chi phí thấp hơn.

Tài liệu tham khảo

http://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u

http://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_r%E1%BB%99ng

http://vi.wikipedia.org/wiki/Gi%E1%BA%A3i_thu%E1%BA%ADt_t%C3%ACm_ki%E1%BA%BFm_A*

http://en.wikipedia.org/wiki/Taxicab_geometry

http://aptech.vn/kien-thuc-tin-hoc/n-puzzle-tim-hieu-ve-cach-giai-bai-toan.html