Bài tập lập trình

Đối tượng: Lớp K57 Máy tính & Khoa học Thông tin,

Đại học Khoa học Tự nhiên Hà Nội, 2014

Bài toán

Viết chương trình giải bài toán *n-puzzle* như sau. Có *n*n -1* mảnh ghép được xếp trong một hình vuông *n * n* để lập thành một mẫu hình nào đó, một mảnh trống. Ban đầu các mảnh ghép được trộn ngẫu nhiên. Trong mỗi bước di chuyển, ta có thể hoán vị mảnh trống với một mảnh kề nó bất kì. Mục tiêu là khôi phục lại mẫu hình với số lần di chuyển ít nhất.

Ta mô hình bài toán dưới dạng ma trận số nguyên *n*n*, các ô được đánh số từ 1 tới *n * n – 1*, riêng ô trống được đánh số 0.

Dưới đây là một ví dụ các bước di chuyển từ ma trận ban đầu về ma trận đích:

0 1 3 4 2 5 7 8 6	→	1 0 3 4 2 5 7 8 6	→	1 2 3 4 0 5 7 8 6	→	1 2 3 4 5 0 7 8 6	→	1 2 3 4 5 6 7 8 0
Ban đầu		1 sang trái		2 lên trên		5 sang trái		Đích

Thuật toán A*

Ta sử dụng thuật toán tìm kiếm A* để giải bài toán nêu trên. A* là thuật toán tìm kiếm heuristic được sử dụng rộng rãi trong ngành trí tuệ nhân tạo. Ta gọi mỗi *nút tìm kiếm* trong bài toán gồm ma trận, số bước đã dịch chuyển để đến ma trận này và nút tìm kiếm trước đó. Trước tiên, ta thêm nút tìm kiếm đầu tiên (gồm ma trận ban đầu, 0 bước dịch chuyển, nút tìm kiếm trước đó là rỗng) vào một hàng đợi ưu tiên. Sau đó ta xóa nút tìm kiếm có độ ưu tiên thấp nhất trong hàng đợi, và thêm vào hàng đợi tất cả các nút tìm kiếm hàng xóm của nó – tức là thêm các nút tìm kiếm có thể đạt tới từ nút hiện tại với một bước chuyển. Lặp lại thủ tục này cho tới khi nút tìm kiếm được xóa tương ứng với nút đích.

Ta sử dụng *hàm ưu tiên* cho mỗi nút tìm kiếm là hàm Manhattan, xác định như sau: hàm Manhattan của một nút tìm kiếm là tổng các khoảng cách Manhattan (tổng các khoảng cách ngang và dọc) từ các ô tới các vị trí đích của nó, cộng với số bước dịch chuyển đã thực hiện để tới nút tìm kiếm hiện tại. Ví dụ, hàm ưu tiên Manhattan của nút tìm kiếm đầu tiên cho giá trị 10:

8 1 3 4 0 2 7 6 5	1 2 3 4 5 6 7 8 0	1 2 3 4 5 6 7 8 --------- 1 2 0 0 2 2 0 3
Ban đầu	Đích	Manhattan = 10 + 0

Ta có quan sát sau: Để giải bài toán từ một nút tìm kiếm cho trước, tổng số lần dịch chuyển ta cần thực hiện (gồm cả những lần đã dịch chuyển) nhỏ nhất chính là độ ưu tiên của nút. Chú ý rằng ta không xét ô trắng khi tính độ ưu tiên. Do đó, khi nút đích được lấy ra khỏi hàng đợi thì ta sẽ tìm được dãy dịch chuyển từ nút ban đầu tới nút đích và đây cũng chính là dãy có số lần dịch chuyển ít nhất.

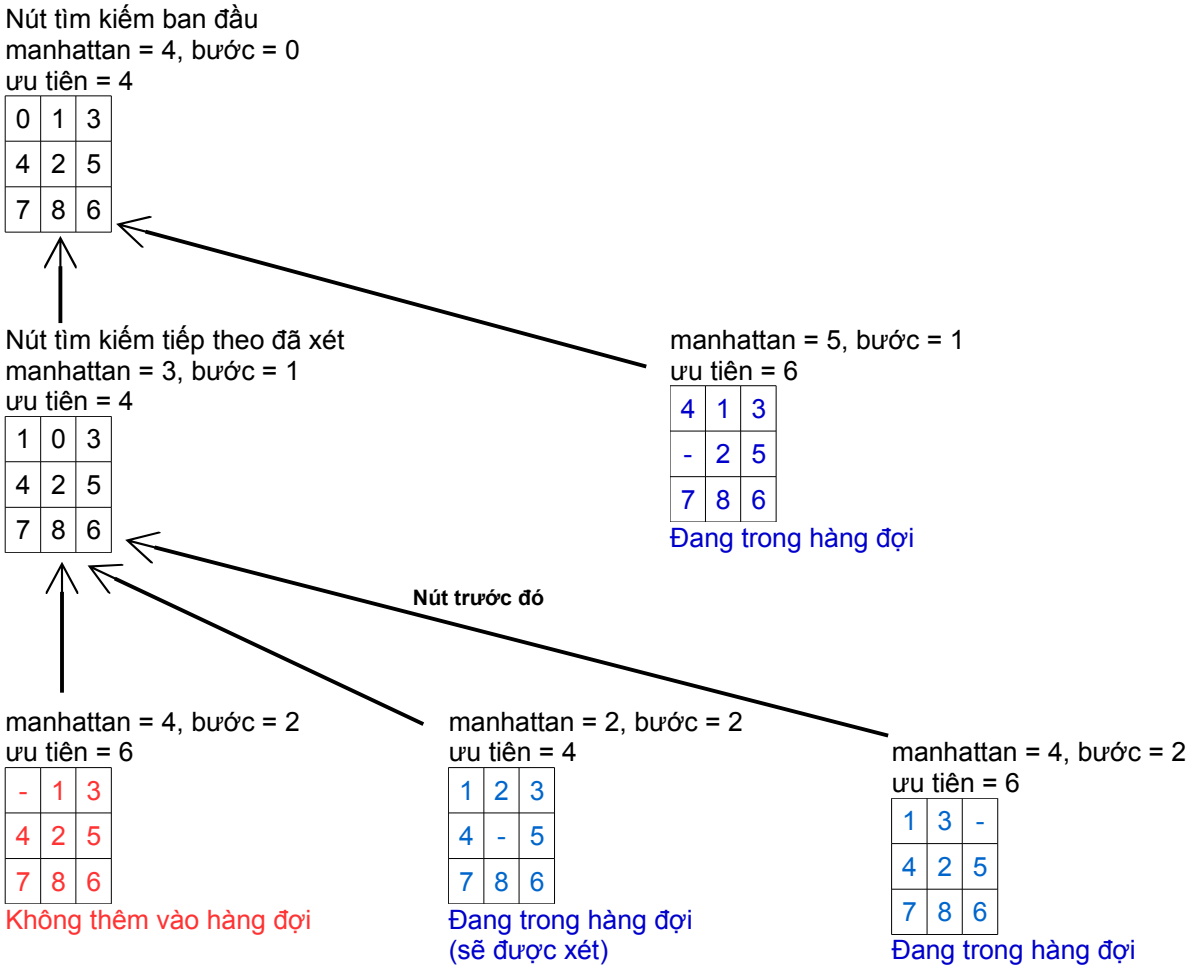
Tối ưu không gian tìm kiếm

Thuật toán tìm kiếm A* nêu trên có nhược điểm là các nút tìm kiếm ứng với ma trận giống nhau được đưa vào hàng đợi nhiều lần. Để giảm việc phải tìm kiếm trên nhiều nút vô ích, khi xét các nút hàng xóm, ta không thêm nút hàng xóm nào có ma trận giống với ma trận của nút tìm kiếm trước đó. Ví dụ:

8 1 3 4 0 2 7 6 5	**8 1 3** **4 2 0** **7 6 5**	8 1 0 4 2 3 7 6 5	8 1 3 4 0 2 7 6 5	8 1 3 4 2 5 7 6 0
Trước đó	**Nút tìm kiếm**	Hàng xóm	~~Hàng xóm~~ (Không xét)	Hàng xóm

Cây trò chơi

Ta có thể biểu diễn quá trình tìm kiếm lời giải dưới dạng một cây, thường được gọi là *cây trò chơi*, trong đó mỗi nút của cây ứng với một nút tìm kiếm và các nút con của nó ứng với các nút tìm kiếm hàng xóm. Nút gốc của cây là nút ban đầu; các nút trong là các nút tìm kiếm đã xét; các nút lá là các nút nằm trong hàng đợi. Tại mỗi bước, thuật toán A* lấy nút có độ ưu tiên thấp nhất ra khỏi hàng đợi và xử lí nút này (thêm các nút con của nó vào cây trò chơi và vào hàng đợi).



Phát hiện bài toán không có lời giải

Không phải ma trận ban đầu nào cũng cho lời giải. Ví dụ:

1	2	3
4	5	6
8	7	0

Để phát hiện ma trận ban đầu có giải được hay không, ta sử dụng quan sát sau. Tập các ma trận được chia làm hai lớp tương đương tương ứng có giải được hay không: 1) tập các ma trận giải được và 2) tập

các ma trận không giải được nếu ta sửa đổi ma trận ban đầu bằng cách hoán đổi hai ô khác trống bất kì trên cùng một hàng. Từ đó, để phát hiện ma trận có giải được hay không, ta chạy thuật toán A* đồng thời trên hai ma trận – một với ma trận ban đầu và một với ma trận đã được sửa bằng cách hoán đổi hai ô khác trống trên một hàng bất kì. Chỉ có một và chỉ một ma trận dẫn tới ma trận đích. Nếu ma trận sửa đổi dẫn tới ma trận đích thì ma trận ban đầu là không giải được.

API

Tổ chức chương trình theo API dưới đây.

Board

```
public class Board {

  public Board(int[][] blocks)       // xây dựng ma trận kích thước n*n

  public int dimension()             // lấy kích thước n của ma trận

  public int manhattan()             // tính khoảng cách manhattan tới ma trận đích

  public boolean isGoal()            // kiểm tra ma trận này đã là đích chưa?

  public Board twin()                // lấy ma trận sửa đổi bằng cách hoán vị hai ô khác 0

  public boolean equals(Object y)    // ma trận này có bằng ma trận y?

  public Iterable<Board> neighbors() // lấy các ma trận hàng xóm

  public String toString()           // lấy biểu diễn xâu của ma trận

}
```

Solver

```
public class Solver {

    public Solver(Board initial)      // tìm lời giải (sử dụng thuật toán A*)

    public boolean isSolvable()       // ma trận ban đầu có giải được không?

    public int moves()                // số bước dịch chuyển ít nhất để giải; -1 nếu không giải được

    public Iterable<Board> solution() // dãy ma trận trong lời giải; null nếu không giải được

    public static void main(String[] args)

}
```

Hàm main() của lớp Solver có dạng như sau:

```
public static void main(String[] args) {
    // create initial board from a text file
    int[][] blocks = ...;
    Board initial = new Board(blocks);

    // solve the puzzle
    Solver solver = new Solver(initial);

    // print solution to standard output
    if (!solver.isSolvable())
        System.out..println("No solution possible");
    else {
        System.out.println("Minimum number of moves = " + solver.moves());
        for (Board board : solver.solution())
            System.out.println(board);
    }
}
```

Để cài đặt chương trình, bạn cần sử dụng cấu trúc dữ liệu MinPriorityQueue.

Định dạng dữ liệu vào/ra

Ma trận ban đầu được đọc từ một tệp văn bản có định dạng sau. Dòng đầu tiên là kích thước *n*. Tiếp theo là *n* dòng, mỗi dòng chứa *n* số. Ví dụ:

puzzle04.txt

```
3
0  1  3
4  2  5
7  8  6
```

```
java Solver puzzle04.txt
Minimum number of moves = 4
```

```
3
0  1  3
4  2  5
7  8  6
```

```
3
1  0  3
4  2  5
7  8  6
```

```
3
1  2  3
4  0  5
7  8  6
```

```
3
1  2  3
4  5  0
7  8  6
```

```
3
1  2  3
4  5  6
7  8  0
```