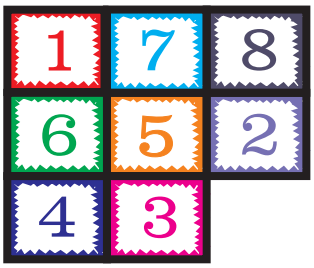
Парамонов Сергей Валерьевич
Студент Московского физико-технического института (МФТИ), II курс, факультет управления и прикладной математики

«Восьминашки» и поиск в ширину

Современные компьютеры имеют огромную память и мощные процессоры. В данной статье предлагается рассмотреть задачу, которая может занять даже очень современный компьютер на несколько дней – это решение головоломки «Пятнашки». С упрощённым вариантом этой головоломки – «Восьминашками» – компьютер вполне справляется – на вычисления уходит несколько секунд. Интересующийся читатель без труда переделает приведённые здесь коды и сможет узнать, по зубам ли его компьютеру серьёзные вычислительные задачи. В статье на элементарном уровне рассказывается про метод поиска в ширину, так что алгоритм решения будет понятен даже неискушённым в дискретной математике читателям.



Пожалуй, игра пятнашки известна всем, здесь мы рассмотрим её уменьшенный вариант. На поле 3x3 находятся 8 пронумерованных фишек размера 1x1. Одна клетка поля всегда пустая, и за один ход можно передвинуть одну из соседних фишек на это поле.

Необходимо из некоторой данной позиции получить исходную позицию, изображённую на рисунке 1.

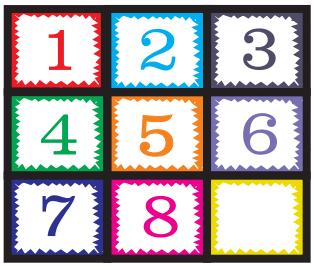


Рис 1. Исходная позиция

Человек без труда решает эту задачу. Алгоритм его действий, в принципе, прост – сначала необходимо поместить фишки 1, 2, 3 в верхний ряд, а потом разобраться с двумя нижними. Но давайте попробуем поручить эту задачу ЭВМ. Напишем

программу, которая решит эту задачу за человека. При этом усложним задание – потребуем, чтобы число ходов, приводящих к исходной позиции, было минимальным (именно эта задача предлагается на онлайн контесте acm.mipt.ru; её номер 89).

Начнём со следующих моментов.

1. Обозначим пустую клетку цифрой 0.

2. Каждой позиции поставим в соответствие перестановку цифр от 0 до 8; чтобы получить из позиции перестановку цифр, нужно просто прочитать номера фишек на поле по строчкам сверху вниз, а в каждой строке – слева направо; например, позиции, изображённой на рисунке в начале статьи, соответствует перестановка (1,7,8,6,5,2,4,3,0).

3. Для каждой перестановки необходимо научиться находить перестановки, в которые можно попасть из данной за один ход.

На рисунке 2 показано, в какие позиции можно попасть из позиции (2,0,3,1,8,4,7,6,5) за один ход. Назовём такие позиции *смежными* (соседними). Смежные позиции получаются в результате обмена нуля с одной из цифр, находящейся в прилегающей к нулю клетке. Давайте составим таблицу, в которой для каждой клетки укажем места, в которые нуль может «перекочевать» из этой клетки.

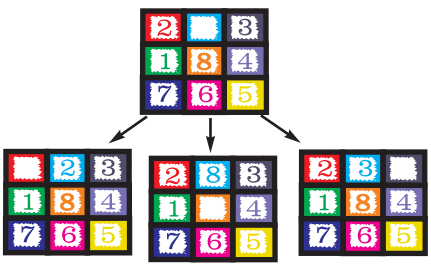


Рис. 2

Число соседей у нуля (пустой клетки) не более 4. Число возможных позиций пустой клетки – 9 штук. Поэтому размер таблицы будет 9x4. Занумеруем клетки, начиная с нуля. На рисунке 3 указаны индексы клеток поля. А в таблице 1 для каждой клетки дан список индексов клеток, в которые может перейти 0 из данной клетки. Каждый список завершается числом -1. Теперь, имея такую таблицу, несложно получать перестановки, смежные данной.

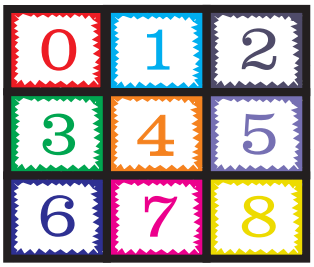


Рис 3. Нумерация индексов

Таблица 1

Возможные переходы

Текущий индекс нуля	Новый индекс нуля			
0	1	3	-1	
1	0	2	4	-1
2	1	5	-1	
3	0	4	6	-1
4	1	3	5	7
5	2	4	8	-1
6	3	7	-1	
7	6	4	8	-1
8	7	5	-1	

Для поиска минимального числа ходов от исходной позиции до данной мы используем метод «поиска в ширину» (breadth-first search).

Возьмём исходную позицию, в которую нам нужно попасть. Пометим её числом 0. Найдём множество смежных с ней позиций. Пометим

эти позиции числом 1 – из них можно попасть в исходную позицию за 1 ход. Рассмотрим далее все позиции, смежные с каждой из этих позиций, и отметим их числом 2 – из них можно попасть в исходную позицию за 2 хода. При этом не будем рассматривать уже отмеченные позиции. Будем действовать пошагово, на шаге *K* помечая очередную группу позиций, до которых можно добраться за *K* ходов.

Можно представлять себе это как лабиринт. Роль комнат в этом лабиринте играют наши перестановки. На каждом шаге у вас есть текущий фронт – комнаты, которые были обнаружены на предыдущем шаге. Проверяем, какие есть смежные комнаты, в которых мы ещё не бывали. Все эти комнаты будут фронтом на следующем шаге.

При реализации этой идеи удобно завести некоторое хранилище для комнат, в которые уже сделан шаг, но смежные комнаты которых ещё не изучены. Этим хранилищем будет структура данных *очередь*.

Очередь – это такая структура данных, в которую можно добавлять объекты и извлекать объекты. При этом объект, который был положен первее, и будет извлечён первее (First In First Out). Про очередь можно думать как про трубу, в один конец которой помещаются объекты, а из другого конца извлекаются.

При решении нам необходимо будет хранить статусы позиций, а именно, каждой позиции сопоставлять минимальное число ходов, за которое из неё можно попасть в исходную позицию. В языках программирования для хранения отображений используется ассоциативный массив. В языке C++ – это шаблон класса map, а в языке Ruby – класс Hash. Назовем это хранилище pos_to_step (позиция отображается в номер шага).

Читатели, заинтересовавшиеся методом поиска в ширину, могут прочитать о нём подробнее в [1]. Интересно заметить, что если вместо очереди использовать стек (первый вошёл – последний вышел), то алгоритм из обхода в ширину превратится в обход в глубину. Обход в глубину уже не будет давать кратчайшего пути. Но, тем не менее, его используют в некоторых задачах, так как он меньше требует памяти и может быть быстро реализован с помощью метода рекурсии. При обходе в ширину рекурсию не используют.

Приведём текст программ на языке C++ и Ruby, которые решают задачу поиска минимального числа ходов, необходимых для сборки головоломки «Восьминашки»:

Код на C++ :

```
#include <iostream>
#include <queue>     /* будем использовать очередь */
#include <map>       /* будем использовать map для хранения
                            pos_to_step
                     */
#include <string>
#include <algorithm>
using namespace std;
int table[9][4] = {
    {1, 3, -1, -1},
    {0, 2,  4, -1},
    {1, 5, -1, -1},
    {0, 4,  6, -1},
    {1, 3,  5,  7},
    {2, 4,  8, -1},
    {3, 7, -1, -1},
    {6, 4,  8, -1},
    {7, 5, -1, -1}
};

map <string, int> pos_to_step; /* отображение позиции -> номер хода*/

queue < string, int > queue;    /* очередь для новых найденных позиций */

int main()
{
   int zero_idx; /* индекс нуля */
   int count;    /* количество ходов до исходной позиции */
   string start("123456780"); /* исходная позиция */
   string position; /* исследуемая позиция */
   cin >> position; /* считаем позицию (строка из 9 цифр) */

   /* помещаем исходную позицию в очередь */
   pos_to_step[start] = 0;
   queue.push( start );

   /* запускаем BFS */
   /* будем работать, пока очередь не пуста */
   while ( ! queue.empty() ){
      string q = queue.pop() /* извлекли очередную позицию q
                                из очереди queue */
      count = pos_to_step[q]; /* число ходов из q
                                 до исходной  позиции */
      zero_idx = q.find("0");  /* найдём индекс нуля в q */
      for ( int i = 0; i < 4 && table[zero_idx][i] != -1; i++){
         /* перебираем смежные вершины */
         string tmp = q;
         /* перейдём в смежную позицию */
```

```
            swap( tmp[zero_idx], tmp[ table[zero_idx][i] ] );
            if ( pos_to_step.find(tmp) == pos_to_step.end() ){
               // нашли непомеченную смежную позицию tmp
               pos_to_step[tmp] = count + 1;
               queue.push( tmp );
            }
         }
      }
      /* вывод ответа */
      if ( pos_to_step.find(position) == pos_to_step.end() )
         cout << "Исходную позицию получить нельзя\n";
      else // выведем число ходов, если путь есть
         cout << pos_to_step[ position] << '\n';

      return 0;
}
```

Код на Ruby:

```
# считываем позицию, до которой нужно добраться
position = gets.gsub(/\s/,'')
# инициализируем пустую очередь -- это просто массив;
# в Ruby массивы поддерживают интерфейс очереди,
# для этого используются методы push и shift.
queue = []
# создаём хэш: позиция -> минимальное число ходов до исходной
pos_to_step = {}
# создаём таблицу возможных перемещений пустой клетки (нуля)
table = [
    [1, 3],
    [0, 2, 4],
    [1, 5],
    [0, 4, 6],
    [1, 3, 5, 7],
    [2, 4, 8],
    [3, 7],
    [6, 4, 8],
    [7, 5]
]
start = '123456780'     # исходная позиция
pos_to_step[start] = 0 # от исходной до исходной 0 ходов
queue.push  start       # помещаем исходную позицию в очередь
# запускаем BFS
while not queue.empty? do
  q = queue.shift
  zero_idx = q.index('0')
  # для всех возможных ходов в смежные позиции делаем …
  table[zero_idx].each do |i|
    tmp = q.dup # создаём копию q (duplicate)
    # сделаем обмен местами (swap);
    # В Ruby это делается с помощью multiple assignment:
```

```
    tmp[zero_idx],tmp[i] = tmp[i],tmp[zero_idx]
    if  not pos_to_step.has_key?(tmp)
      # если вершина tmp ещё не помечена,
      # то метим её и помещаем в очередь
      pos_to_step[tmp] = pos_to_step[q] + 1
      queue.push tmp
    end
  end
end
if pos_to_step.has_key? position
   puts 'Исходную позицию получить нельзя'
else
   puts pos_to_step[position]
end
```

Код на Ruby можно существенно ускорить, если позициям сопоставлять не строку, а девятизначное целое число. Кроме того, из цикла можно выходить сразу, как только мы дойдём до позиции position.

Для углублённого изучения теории алгоритмов рекомендуется обращаться к следующим источникам:

[1] Т. Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. 1-е изд., С.436-445.

[2] Р. Седжвик. Фундаментальные алгоритмы на C++. Часть 5. Алгоритмы на графах.

Юмор Юмор Юмор Юмор Юмор Юмор

- Выпавший из рук инструмент падает именно туда, где он может нанести максимум повреждений.
- После демонтажа и сборки какого-либо прибора несколько деталей остаются лишними.
- Количество лишних деталей обратно пропорционально их необходимости.
- Все герметично соединённые трубы протекают.