Course C^{++}, Exercise Number 9

Date: 16.05.2013

The 15-Puzzle

The fifteen puzzle `http://en.wikipedia.org/wiki/Fifteen_puzzle`) was introduced by Noyes Palmer Chapman in 1875. In the beginning of 1880, the puzzle became a craze, that lasted approximately half a year. (In 1981, the same effect was obtained by Rubik's cube.)

The aim of this task is to solve the 15-puzzle, and to learn how to use data structures in the STL (Standard Template Library).

We will solve the 15-puzzle by *exhaustive search*. This means that we fill a container with all the states that we can reach from the state that we try to solve, until we have reached the solved state.

In order to implement this algorithm one needs two sets of states, called R and U (*reached* and *unchecked*). The set R is the set of all states that we have reached. The set U is the set of states that we have reached, but for which we didn't check yet, which states are reachable from it. More precisely, we have $U \subseteq R$, and for every $s \in R \backslash U$, if state s' is reachable from s in a single move, then $s' \in S$. This is the structure of the search algorithm:

```
solve( startingstate )
{
   R = { startingstate }; U = { startingstate };
   while( U is non-empty )
   {
      select a state fnext from U, and remove it from U.
      if fnext is solved then we found the solution!

      for every f' that can be reached from fnext in a single move,
         if( f' is not in R ) then
         {
            R = R union {f'};  U = U union {f'};
         }
   }
   We did not find a solution :-(
}
```

1. Download the file `fifteen.cpp` from my homepage.

Complete class `class fifteen`, and the printing function for `class fifteen`.

It is better to make separate files `move.h`, `fifteen.h` and `fifteen.cpp`.

The function `fifteen::distance( )` is very important, because it will decide which element from U will be selected in the search algorithm. Wikipedia has some suggestions.

2. For the set R, we will use an `std::map`. [1] Map is implemented by a search tree, which means that we need an order on `class fifteen`, so that it can be sorted. Map allows two ways to define the order. The first is by defining an operator `<`, the second is by defining a comparator class. The first method should only be used when the order is fundamental to the data type (Like `<` on the natural numbers, reals or strings). The second method should be used when the only reason that we introduce the order, is the map. This is the case here, so we introduce a class `fifteen_cmp`.

   Complete the method `bool fifteen_cmp::operator( )` . It can be any order on `fifteen`s that sorts them completely, for the rest there are no requirements. See `http://www.cplusplus.com/reference/stl/map/`.

   An element can be put in the map by assigning `reached[f] = true`. Membership can be checked by checking `reached[f]`.

3. Complete the function `solve`. For set U, we use an `std::priority_queue`. When selecting an unchecked state, we do not pick an arbitray one, but we pick one that is closest to the solution. In this way, we hope to find a solution quicker.

   Priority queue needs an order, but the purpose of the order is completely different from the order of map. The order of map must be total, but for the rest there are no requirements. It doesn't have to be meaningful. Its only purpose is to sort, and it doesn't matter how.

   The order of priority queue must be a preference order. It will determine which state is checked next in the main loop of the search algorithm. The second order should be carefully chosen, because it determines the efficiency of the search algorithm, and which solution will be reported to the user, when there is more than one solution.

   In our case, we use a `fifteen_better` object for selecting. The `fifteen_better` object simply calls `fifteen::distance( )`, so in the end, it is this function which makes the decisions. See `http://www.cplusplus.com/reference/stl/priority_queue/`, for explanation of priority queue.

   Note that not all states are solvable. If you randomly pick a starting state, you have a chance of one half that it is solvable.

[1] You may also use **std::unordered_map**