There are two code files that you need to concern yourself width: `MySolver.h` and `MySolver.cpp`. You might also want to look at the definition of the Node class in `Node.h`.

Node.h

Notice that the Node class is designed to store two things: the state of the puzzle and a path that represents the moves it takes to get to that state. The state is represented as a vector of integers which are the numbers of each tile read left to right from top to bottom with the blank tile using the number 0. Each move is represented by a single character U, D, L or R and a sequence of moves as a string using these letters. The Node class has a constructor that sets the state and path and accessor functions to access each of these. You should not edit this file, but it is a useful reference.

```
#pragma once

#include <memory>
#include <string>

class Node {
public:
  Node(std::vector<int> state, std::string path);

  const std::vector<int> &getState() const;
  std::string getPath() const;

private:
  std::vector<int> _state;
  std::string _path;
};
```

MySolver.h

The class MySolver is the one that you will be modifying to implement your solution. The class has a handle full of methods:

MySolver	The constructor takes a puzzle size (3, 4, 5, etc.) and the puzzle that needs to solve. You will not be modifying this method.
init	You can use this function to do any precomputation you need for your final version of the project. For example, there are some lookup tables that can significantly speed up your solution.
mySolution	This method actually find a solution to the puzzle are returns a string representing the sequence of moves that solves the puzzle. You will be changing this function.
_expand	This method finds that states reachable from the current state and adds them to the frontier. You will be making minor changes to this function.
_heurstic	You will need to implement this function for any informed search strategy you perform.

We will discuss how each of these work in the next section.

```cpp
#pragma once

#include <vector>

#include "Node.h"
#include "Solver.h"

class MySolver : public Solver {
public:
  MySolver(int size, std::vector<int> puzzle);
  void init();
  std::string mySolution();

private:
  void _expand(Node &node);
  int _heuristic(const std::vector<int> state) const;

private:
  // Change this to whatever data stucture you need.  However make
  // sure to keep the type mutable to do interfere with const-correctness.
  mutable std::vector<Node> _frontier;
};
```

MySolver.cpp

This implementation of MySolver stores the frontier in a vector (of Nodes) and does a version of the graph search discussed in the book. It selects nodes to expand from the frontier randomly (not a very good strategy.)

The algorithm works as follows. In line 17, an Node instance is created that stores the initial state of the puzzle. Note that the path for this Node is represented by an empty string since it takes no moves from the initial state to get to this state. In line 24, this Node is added to the frontier (the push_back method adds the node to the end of the vector). The code in lines 21–22 initializes the random number generator (you will not need this for you implementation.

The while loop in lines 26–47 is the main loop for the graph search algorithm. It will continue as long as there is a node in the frontier. In lines 31–35, a random node is selected from the frontier, stored in the variable current and removed from frontier. In lines 37–38, the algorithm checks if the solved state has been found and, if so, returns the path. In line 46, the selected node is expanded. We will look at the implementation of this function to see what is does. The other portions of this function are there to display periodic updates on the algorithms progress.

The _expand method finds all states reachable from the current state and adds them to the frontier if they have not been seen. In line 58, it stores the state stored by the current node and, in line 59, it only continues only if they state has not been expanded previously. Assuming the state has not been previously expanded, it is added to the set of expanded nodes in line 62. In line 64, the algorithm loops through all valid actions that can be applied to the current state. If that state has not been previous expanded (which is checked in line 67), a new node is created storing the new state in line 68. Note that this also creates a path of how to get to the new state from the initial one: follow the same path the current state and then apply the single move. This node is added to the frontier in line 73.

```cpp
1 #include <algorithm>
2 #include <iostream>
3 #include <random>
```

```

#include "MySolver.h"

MySolver::MySolver(int size, std::vector<int> puzzle) : Solver(size, puzzle) {}

void MySolver::init() {
  // Put any initialization code here
}

std::string MySolver::mySolution() {
  // You can rewrite this to use whatever algorithm you need to.
  // Right now it randomly selects frontier nodes (not a good algorithm)

  Node initial(_puzzle.getInitial(), "");

  // Initialize random number generator, only needed because this
  // solver uses randomized behavior.
  std::random_device rd;
  std::mt19937 gen(rd());

  _frontier.push_back(initial);
  int count = 0;
  while (!_frontier.empty()) {
    count++;
    // Select a random element of the frontier andr emove it from the
    // frontier.  Note you will need to change this since you will
    // probablby not be using a vector to store your frontier.
    std::uniform_int_distribution<> dis(0, _frontier.size() - 1);
    int randomIndex = dis(gen);
    Node current = _frontier[randomIndex];
    _frontier[randomIndex] = _frontier[_frontier.size() - 1];
    _frontier.pop_back();

    // Check if we have reached the goal
    if (_puzzle.isGoal(current.getState()))
      return current.getPath();

    if (count % 100000 == 0)
      std::cout << _numExpansions << " nodes examined.  " << _frontier.size()
                << " nodes on the frontier" << std::endl;

    // _expand adds unexpanded neighbors to the frontier
    _expand(current);
  }

  return "";
}

int MySolver::_heuristic(const std::vector<int> state) const {
  // Implement this for use in your informed search your informed search
  return 0;
55 }
56
57 void MySolver::_expand(Node &node) {
58   std::vector<int> state = node.getState();
59   if (_expanded.find(state) != _expanded.end())
60     return;
61   _numExpansions++;
62   _expanded.insert(state);
63
64   for (auto action : _puzzle.actions(state)) {
65     auto newState = _puzzle.result(state, action);
66
67     if (_expanded.find(newState) == _expanded.end()) {
68       Node newNode(newState, node.getPath() + action);
69
70       // The following line will probably need to be changed
71       // depending on the data structure you use for _frontier.
72       // Do not make any other changes to this function.
73       _frontier.push_back(newNode);
74     }
75   }
76 }
```

What you will need to change

- _frontier data structure in `MySolver.h`. You will want to change this to be a stack, queue or (in your final version) a priority queue.
- Replace line 24 with the appropriate method to add the initial node to the frontier.
- Replace lines 31–35 with code that selects a Node from the frontier and removes it from the frontier.
- Replace line 73 with appropriate code to add the new node to the frontier.
- When you utilize informed search you will need to implement _heuristic.

For reference on the possible data structures, I suggest the website cppreference.com. Using a stack or queue should be relatively straightforward. Utilizing a priority queue will be more complicated. We will discuss this in class.