Finlay Morrison - Electronic Engineering with Artificial Intelligence - 17/3/2021

P7 - Sudoku GUI 2

2 - Preparation

2.1 - Read Qt tutorial and design GUI

1) What are signals and slots?

A signal is something that is emitted whenever an event is triggered and a slot is something that happens as the result of a signal. The following code gives an example of this.

Header file:

```
#include <QWidget>
#include <QPushButton>

class my_class : public QWidget
{
   Q_OBJECT

public:
   my_class(QWidget *parent=nullptr);

private slots:
   void my_slot_definition();
};
```

Implementation file:

```
#include "includes/header.h"

my_class::my_class(QWidget *parent) : QWidget(parent)
{
   QPushButton* btn = new QPushButton("do something", this);
   connect(btn, &QPushButton::clicked, this, &my_class::my_slot_definition);
}

void my_class::my_slot_definition()
{
   std::terminate();
}
```

Main file:

```
#include <QApplication>

#include "includes/header.h"
```

```cpp
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);

   my_class window;

   window.show();
    return app.exec();
}
```

In this example I create a pushbutton, which is used to generate the signal `QPushButton::clicked`, this is then connected to the slot that I defined, `my_class::my_slot_definition` which is then used to handle the event. The `private slot:` syntax that is used to define a slot requires you to inform the preprocessor of this by adding `Q_OBJECT` to the start of the class declaration.

2) What should my GUI look like?

My GUI will consist of a toolbar, which will contain the following functionality:

- New: Clears the sudoku grid
- Save: Prompts the user for a filename and will serialize the sudoku board into that file.
- Open: Prompts the user for a filename and will deserialize the sudoku board stored in that file to the object storing the sudoku board.
- DIVIDER
- Solve: Solves the sudoku board.
- DIVIDER
- Quit: closes the application.

I will use the rest of the space to display the sudoku board onto. I will use a QGridLayout and then I will have 81 QLineEdits in each spot of the sudoku grid, with the maximum length set to 1. By using a QGridLayout, the board will scale to the window size.

3) What widget classes will I use to implement the user interface, and how will I nest them to create a suitable layout? Will this layout scale when the window is resized?

The toolbar will be implemented with a QToolBar, where I will use a QIcon with a QPixmap passed in to display the options. The grid will be implemented with a QGridLayout, and each of the squares of the grid will have a QLineEdit, with its max size set to 1. The choice of QWidgets here will mean that the board will scale with the window size.

4) What signals and slots should I use for the various widgets? How can I cleanly handle the events from all the 81 widgets in the 9x9 grid without writing 81 separate methods?

Each of the QLineEdit in the grid will have a `textChanged()` signal, that for each element in the grid will be used to update the board when the text is changed. In order to cleanly handle the signals I will define a class for an element on the sudoku grid, and it will then have a member of it connected to the signal as a slot. This will mean that I only need to define a single slot in the class that will be used to implement the sudoku grid part.

3 - Laboratory Work

3.1 - Create the GUI

I first went about creating the toolbar for the GUI, i have 5 icons for it, new, open, save, solve and quit, I got the icons from https://iconarchive.com/show/essential-toolbar-icons-by-fasticon.2.html. The code I used to do this was as follows.

window.h

```
#ifndef SUDOKU_SRC_INCLUDES_WINDOW_H
#define SUDOKU_SRC_INCLUDES_WINDOW_H

#include <QMainWindow>
#include <QApplication>

class window : public QMainWindow
{
   Q_OBJECT

public:
   window(QWidget *parent=nullptr);
};

#endif
```

window.cpp

```
#include <QToolBar>
#include <QIcon>
#include <QAction>

#include "includes/window.h"

window::window(QWidget *parent) :
   QMainWindow(parent)
{
   /* loading images for toolbar icons */
   QPixmap new_pix("img/new.png");
   QPixmap open_pix("img/open.png");
   QPixmap save_pix("img/save.png");
   QPixmap solve_pix("img/wizard.png");
```

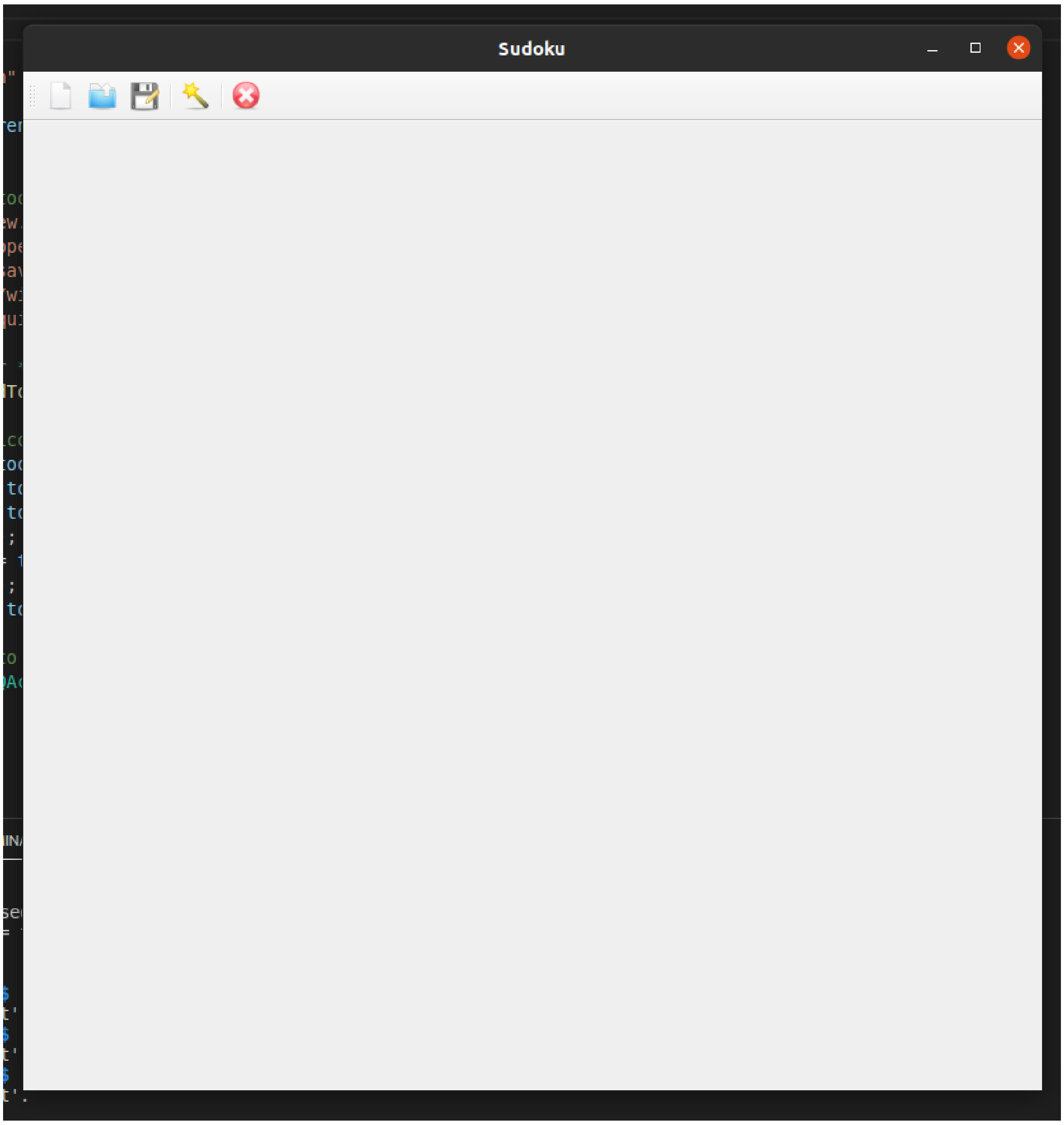
```
    QPixmap quit_pix("img/quit.png");

    /* initializing toolbar */
    QToolBar *toolbar = addToolBar("main toolbar");

    /* setting up toolbar icons */
    QAction *new_action = toolbar->addAction(QIcon(new_pix), "new file");
    QAction *open_action = toolbar->addAction(QIcon(open_pix), "open file");
    QAction *save_action = toolbar->addAction(QIcon(save_pix), "save file");
    toolbar->addSeparator();
    QAction *solve_action = toolbar->addAction(QIcon(solve_pix), "solve");
    toolbar->addSeparator();
    QAction *quit_action = toolbar->addAction(QIcon(quit_pix), "quit");

    /* connecting signals to their corresponding slots */
    connect(quit_action, &QAction::triggered, qApp, &QApplication::quit);
}
```

main.cpp

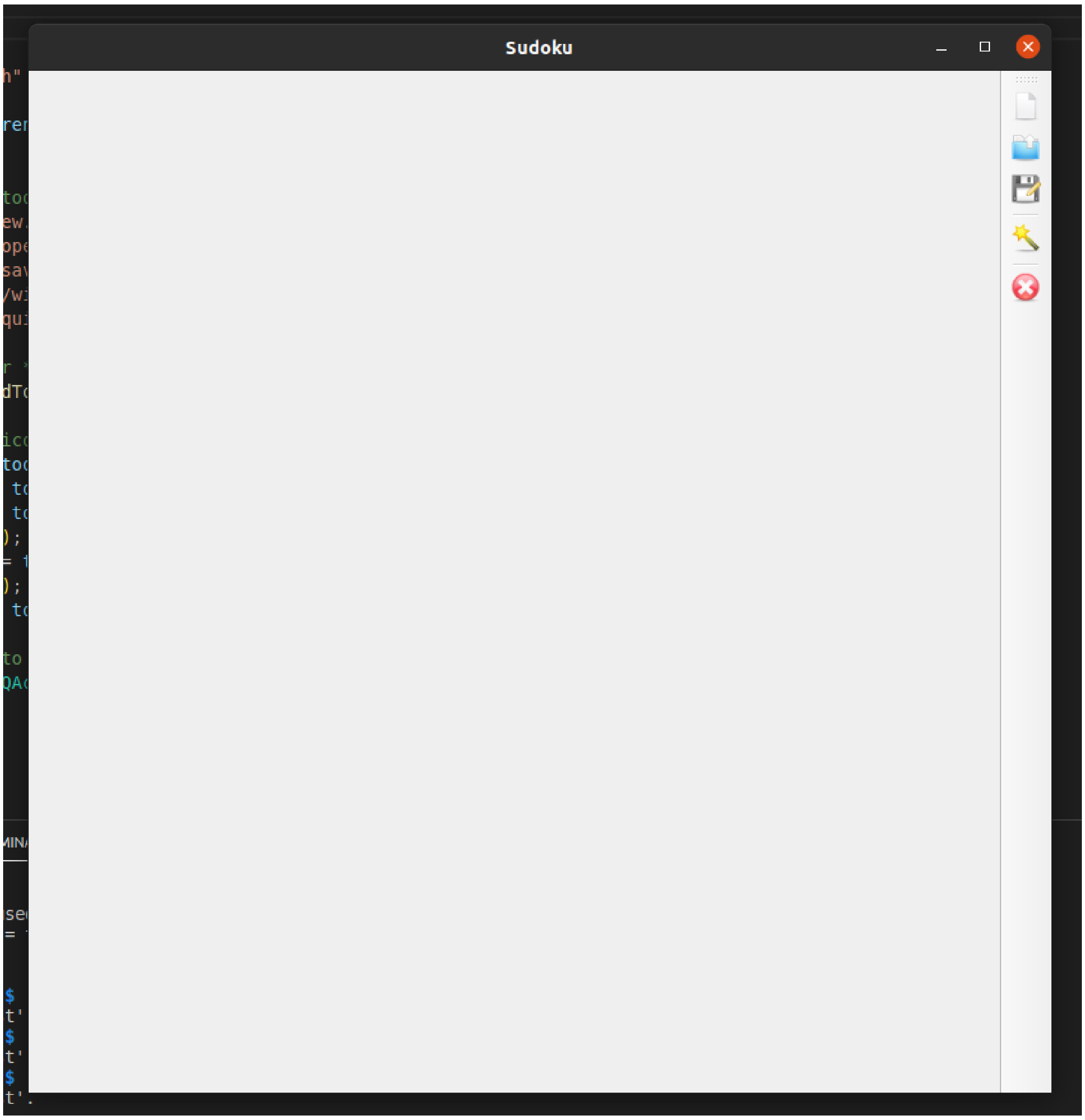
```
#include "includes/window.h"

int main(int argc, char *argv[]) {

  QApplication app(argc, argv);

  window win;

  win.resize(800, 800);
  win.setWindowTitle("Sudoku");
  win.show();

  return app.exec();
}
```

In order to use a toolbar, I must derive the window class from the QMainWindow class. I load each of the icons, I then create a toolbar and create actions for new, open, save, solve and quite. I then connect the signal from the quit button to QApplication::quit on the qApp object, which is defined in the QApplication header file (more specifically the qapplication.h header file, which is all that is included in the QApplication header file).

This code will produce the following output into a window:



The toolbar is also capable of being dragged around to any of the toolbars at runtime:



The rest of the grid will be able to account for this when it is implemented, since the grid will auto scale.

I first changed the constructor of sudoku_grid to create some UI elements to show the grid.

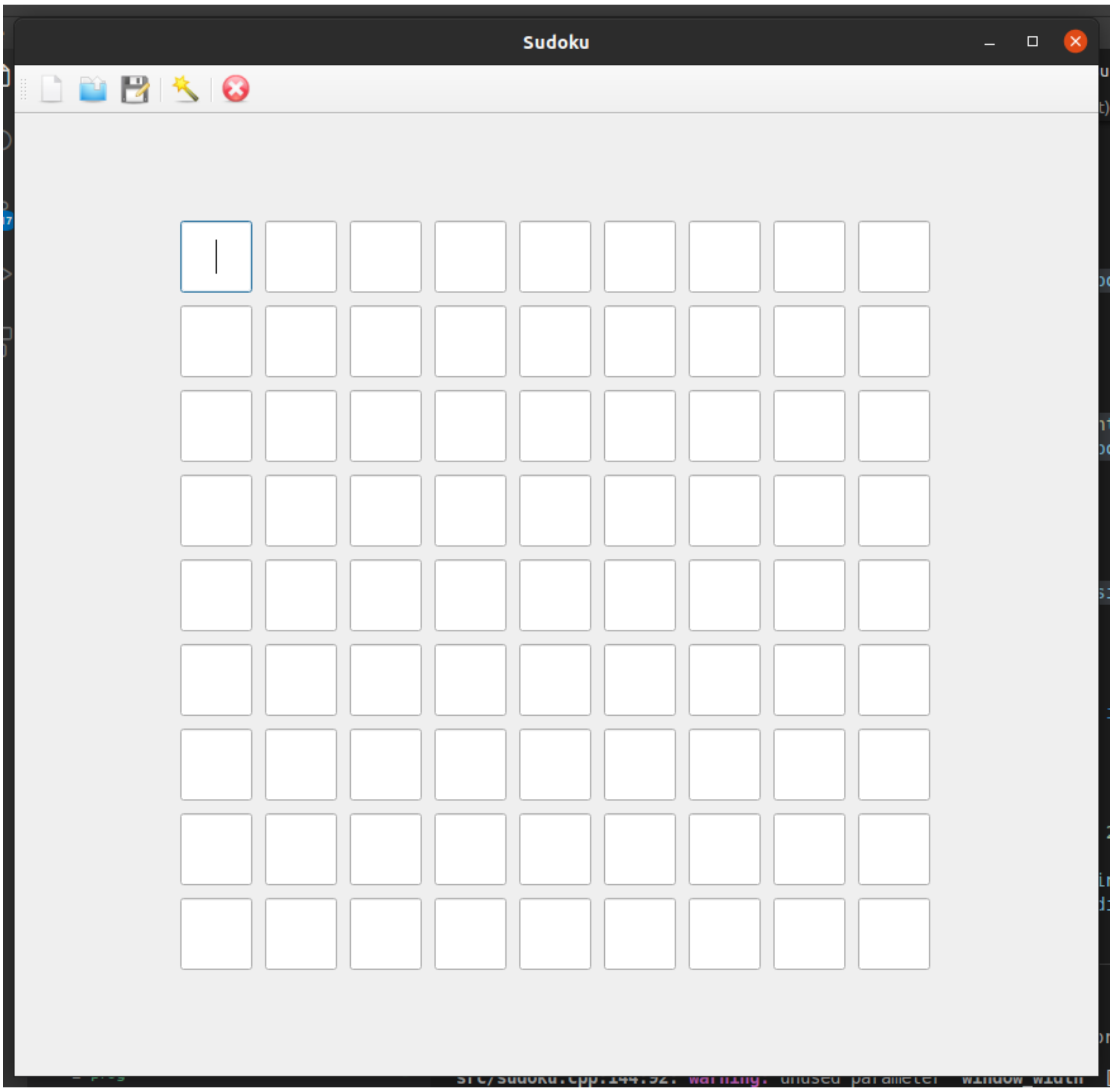
```
sudoku_grid::sudoku_grid(QMainWindow* parent, int grid_size) :
   grid_size(grid_size),
   board_size(grid_size * grid_size),
   grid(grid_size * grid_size, std::vector<int>(grid_size * grid_size)),
   grid_edit(grid_size * grid_size, std::vector<QLineEdit*>(grid_size *
grid_size)),
   parent(parent)
{
   constexpr int padding = 100;
   constexpr int spacing = 10;
   constexpr float font_scale = 0.8;

   const int board_pixel_size = std::min(parent->height(), parent->width())
- 2 * padding;
   const int input_pixel_size = (board_pixel_size - (board_size - 1) *
spacing) / board_size;

   QGridLayout *sudoku_grid = new QGridLayout;
   sudoku_grid->setSpacing(spacing);

   QFont font;
   font.setPointSize(input_pixel_size * font_scale);

   for (int i = 0; i < board_size; ++i)
   {
       for (int j = 0; j < board_size; ++j)
       {
           grid_edit[i][j] = new QLineEdit(parent);
           grid_edit[i][j]->setMaxLength(1);
           grid_edit[i][j]->setBaseSize(input_pixel_size, input_pixel_size);
           grid_edit[i][j]->setFont(font);
           grid_edit[i][j]->setAlignment(Qt::AlignCenter);
           sudoku_grid->addWidget(grid_edit[i][j], i, j);
       }
   }

   QGridLayout *screen_grid = new QGridLayout;
   for (int i = 0; i < 3; ++i)
   {
       for (int j = 0; j < 3; ++j)
       {
           if (i == 1 && j == 1)
           {
               screen_grid->addLayout(sudoku_grid, i, j);
           }
           else
           {
               QFrame* frame = new QFrame;
               frame->setMaximumSize(padding, padding);
               screen_grid->addWidget(frame, i, j);
           }
       }
   }

   QWidget* widget = new QWidget();
   widget->setLayout(screen_grid);
   parent->setCentralWidget(widget);
```

```
}
```

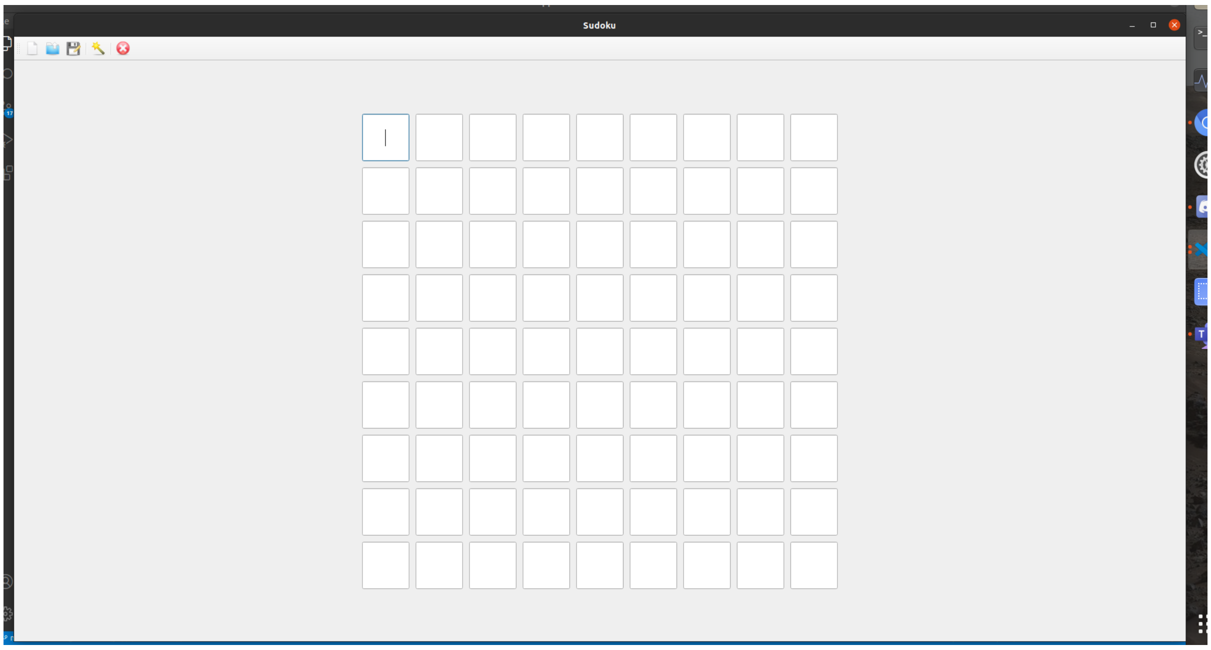
I also implemented a function that would run on every draw event to change the size of the inputs:

```
void sudoku_grid::display_grid(QPainter* painter, int border_width, int
subgrid_width, int window_width, int window_height)
{
   constexpr int padding = 100;
   constexpr int spacing = 10;
   constexpr float font_scale = 0.8;

   const int board_pixel_size = std::min(parent->height(), parent->width())
- 2 * padding;
   const int input_pixel_size = (board_pixel_size - (board_size - 1) *
spacing) / board_size;

   for (int i = 0; i < board_size; ++i)
   {
       for (int j = 0; j < board_size; ++j)
       {
           grid_edit[i][j]->setFixedSize(input_pixel_size,
input_pixel_size);
       }
   }
}
```

This produces the following output into a window:



The window also scales well:



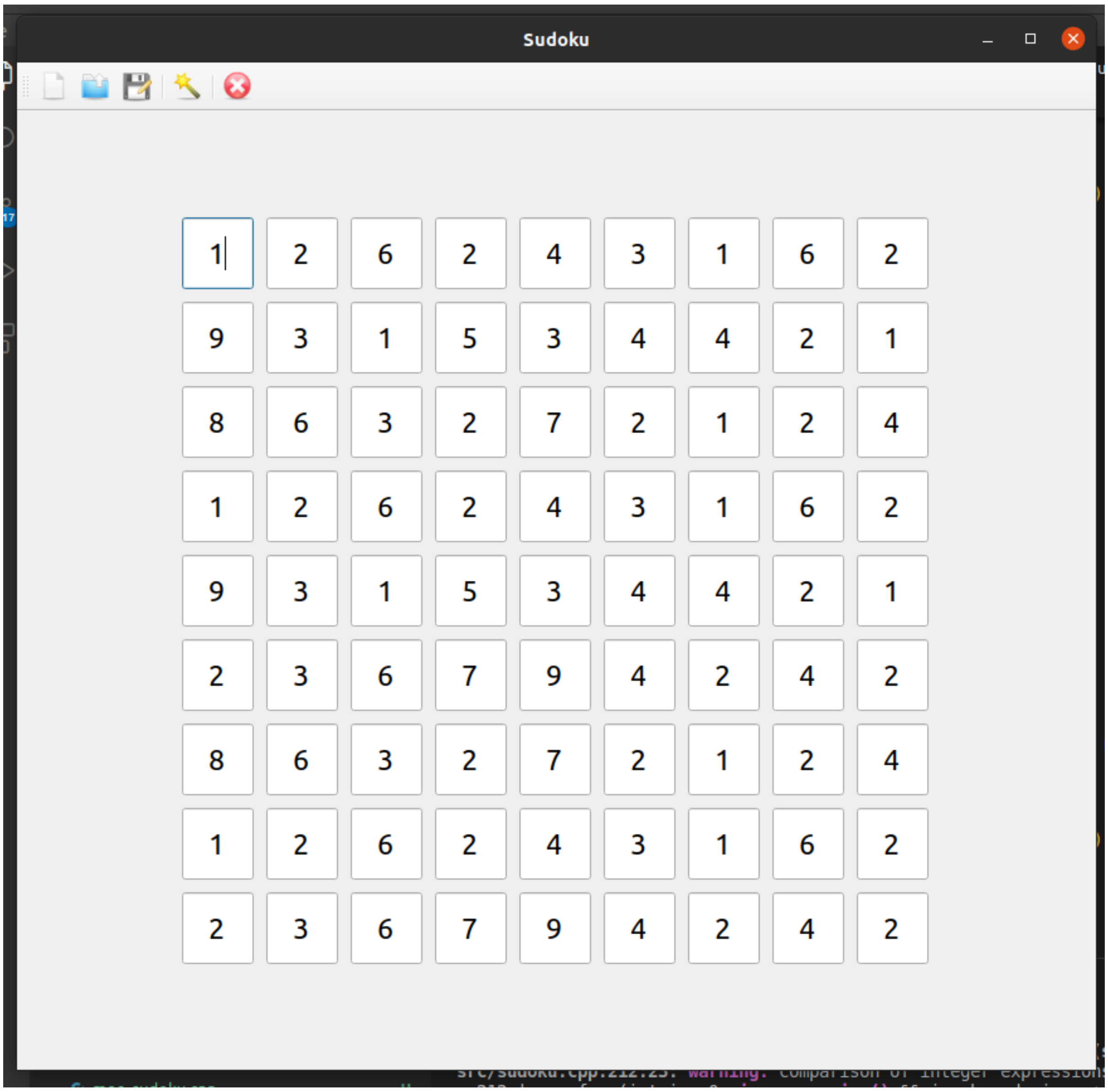
I made a function that takes in a 2D vector of integers and will set the grid to that value.

```
void sudoku_grid::load_grid(std::vector<std::vector<int>> nums)
{
   for (int i = 0; i < nums.size() && i < board_size; ++i)
   {
       for (int j = 0; j < nums[i].size() && j < board_size; ++j)
       {
           grid_edit[i][j]->setText(QString(nums[i][j] + '0'));
       }
   }
}
```

I tested this code by putting the following code at the end of the sudoku grid constructor.

```
load_grid({
       {1,2,6,2,4,3,1,6,2},
       {9,3,1,5,3,4,4,2,1},
       {8,6,3,2,7,2,1,2,4},
       {1,2,6,2,4,3,1,6,2},
       {9,3,1,5,3,4,4,2,1},
       {2,3,6,7,9,4,2,4,2},
       {8,6,3,2,7,2,1,2,4},
       {1,2,6,2,4,3,1,6,2},
       {2,3,6,7,9,4,2,4,2}
   });
```

This gives the following output:



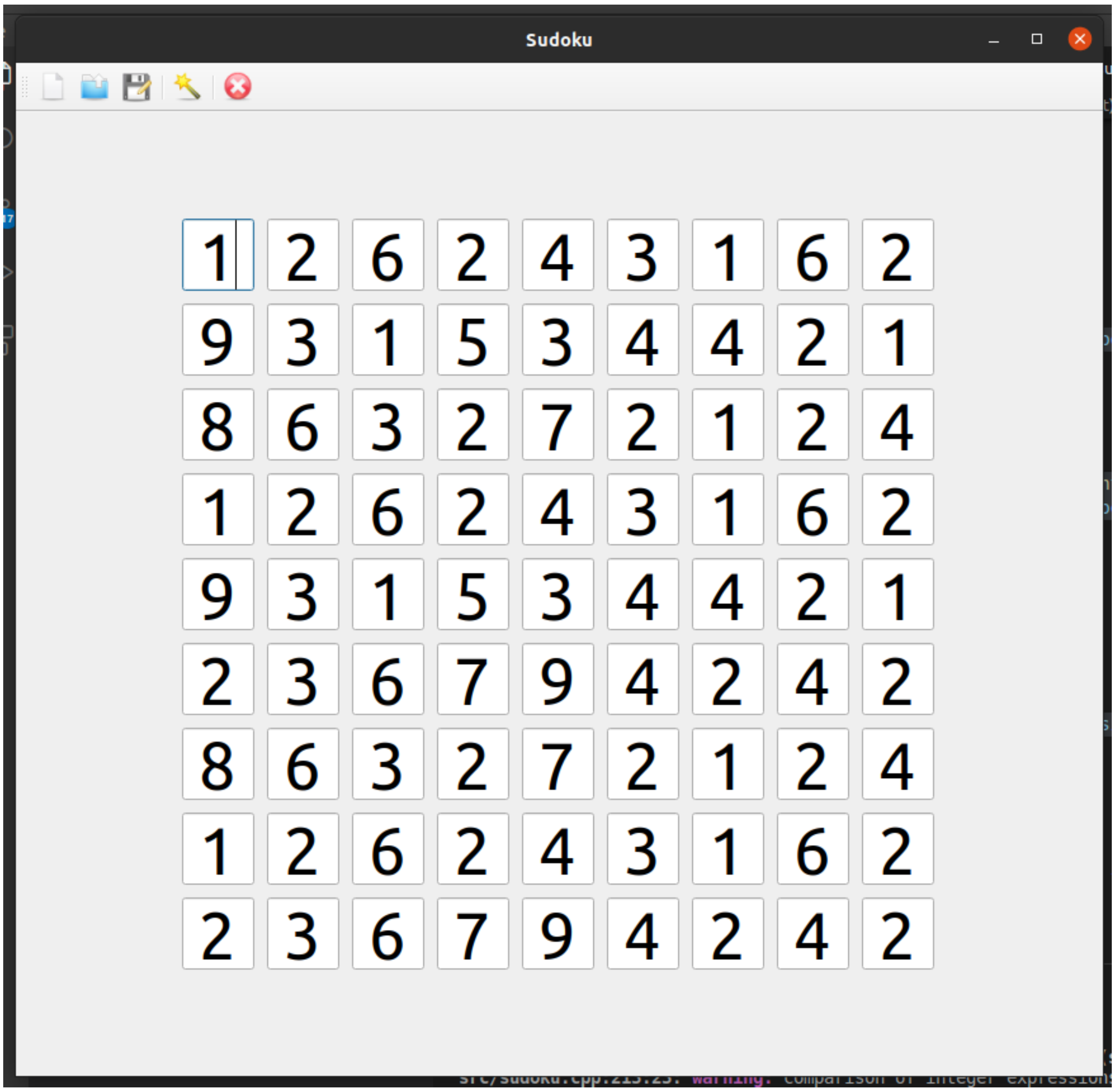
1	2	6	2	4	3	1	6	2
9	3	1	5	3	4	4	2	1
8	6	3	2	7	2	1	2	4
1	2	6	2	4	3	1	6	2
9	3	1	5	3	4	4	2	1
2	3	6	7	9	4	2	4	2
8	6	3	2	7	2	1	2	4
1	2	6	2	4	3	1	6	2
2	3	6	7	9	4	2	4	2

I noticed that the text was not scaling, so I changed the display_grid function to implement this:

```
void sudoku_grid::display_grid(QPainter* painter, int border_width, int
subgrid_width, int window_width, int window_height)
{
   constexpr int padding = 100;
   constexpr int spacing = 10;
   constexpr float font_scale = 0.7;

   const int board_pixel_size = std::min(parent->height(), parent->width())
- 2 * padding;
   const int input_pixel_size = (board_pixel_size - (board_size - 1) *
spacing) / board_size;
```

```
    QFont font;
    font.setPointSize(input_pixel_size * font_scale);

    for (int i = 0; i < board_size; ++i)
    {
        for (int j = 0; j < board_size; ++j)
        {
            grid_edit[i][j]->setFixedSize(input_pixel_size,
input_pixel_size);
            grid_edit[i][j]->setFont(font);
        }
    }
}
```

This now gives the following output:



1	2	6	2	4	3	1	6	2
9	3	1	5	3	4	4	2	1
8	6	3	2	7	2	1	2	4
1	2	6	2	4	3	1	6	2
9	3	1	5	3	4	4	2	1
2	3	6	7	9	4	2	4	2
8	6	3	2	7	2	1	2	4
1	2	6	2	4	3	1	6	2
2	3	6	7	9	4	2	4	2

I declared some slots in the window class:

```
public slots:
   void new_sudoku();
   void save_sudoku();
   void open_sudoku();
   void solve_sudoku();
```

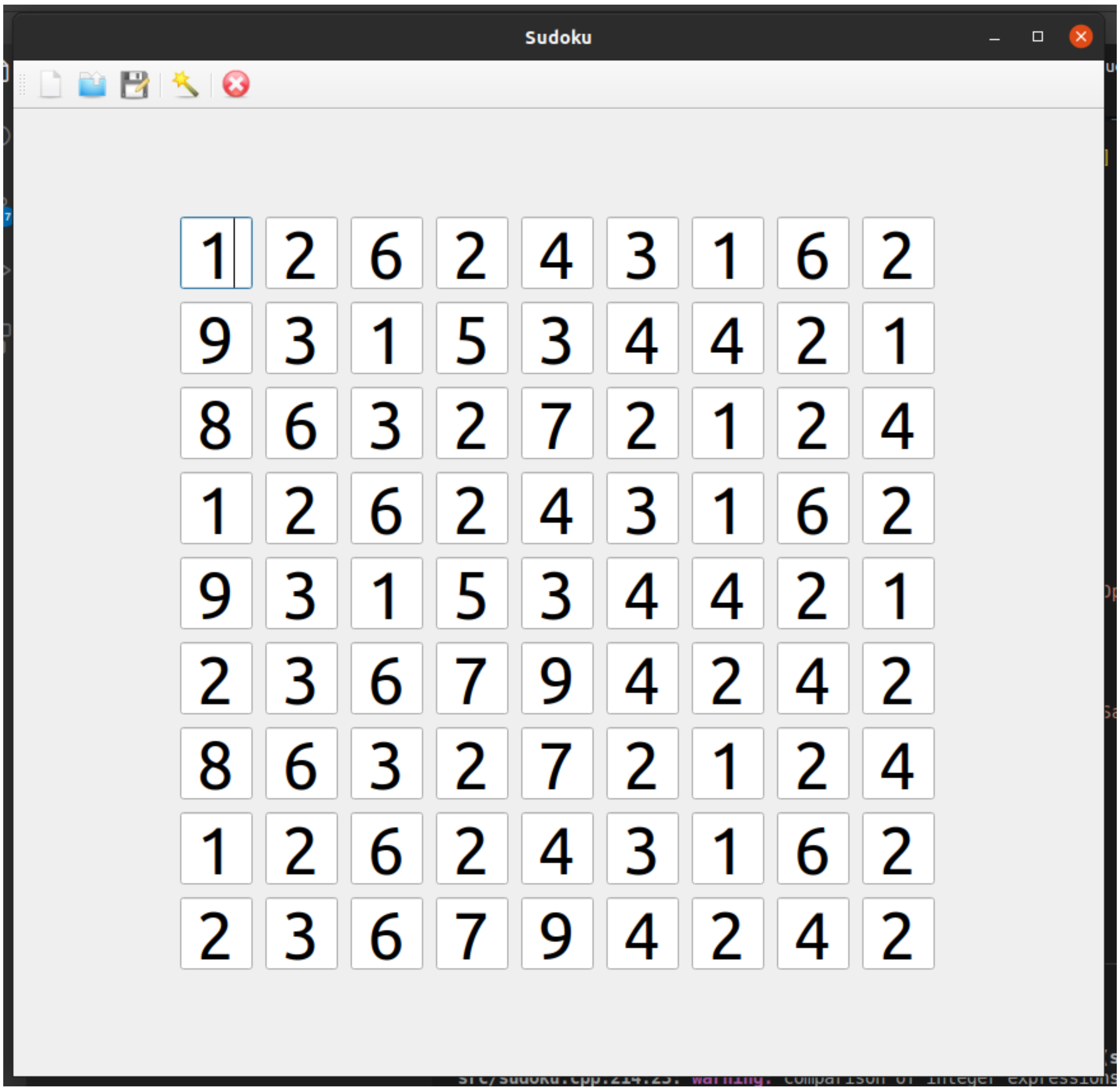
I then connected the rest of the toolbar signals:

```
// connecting signals to their corresponding slots
   connect(new_action, &QAction::triggered, &board,
&sudoku_grid::new_sudoku);
   connect(open_action, &QAction::triggered, &board,
&sudoku_grid::open_sudoku);
   connect(save_action, &QAction::triggered, &board,
&sudoku_grid::save_sudoku);
   connect(solve_action, &QAction::triggered, &board,
&sudoku_grid::solve_sudoku);
   connect(quit_action, &QAction::triggered, qApp, &QApplication::quit);
```

I implemented the new_sudoku slot as follows:

```
void sudoku_grid::new_sudoku()
{
   for (int i = 0; i < board_size; ++i)
   {
       for (int j = 0; j < board_size; ++j)
       {
           grid_edit[i][j]->setText("");
       }
   }
}
```

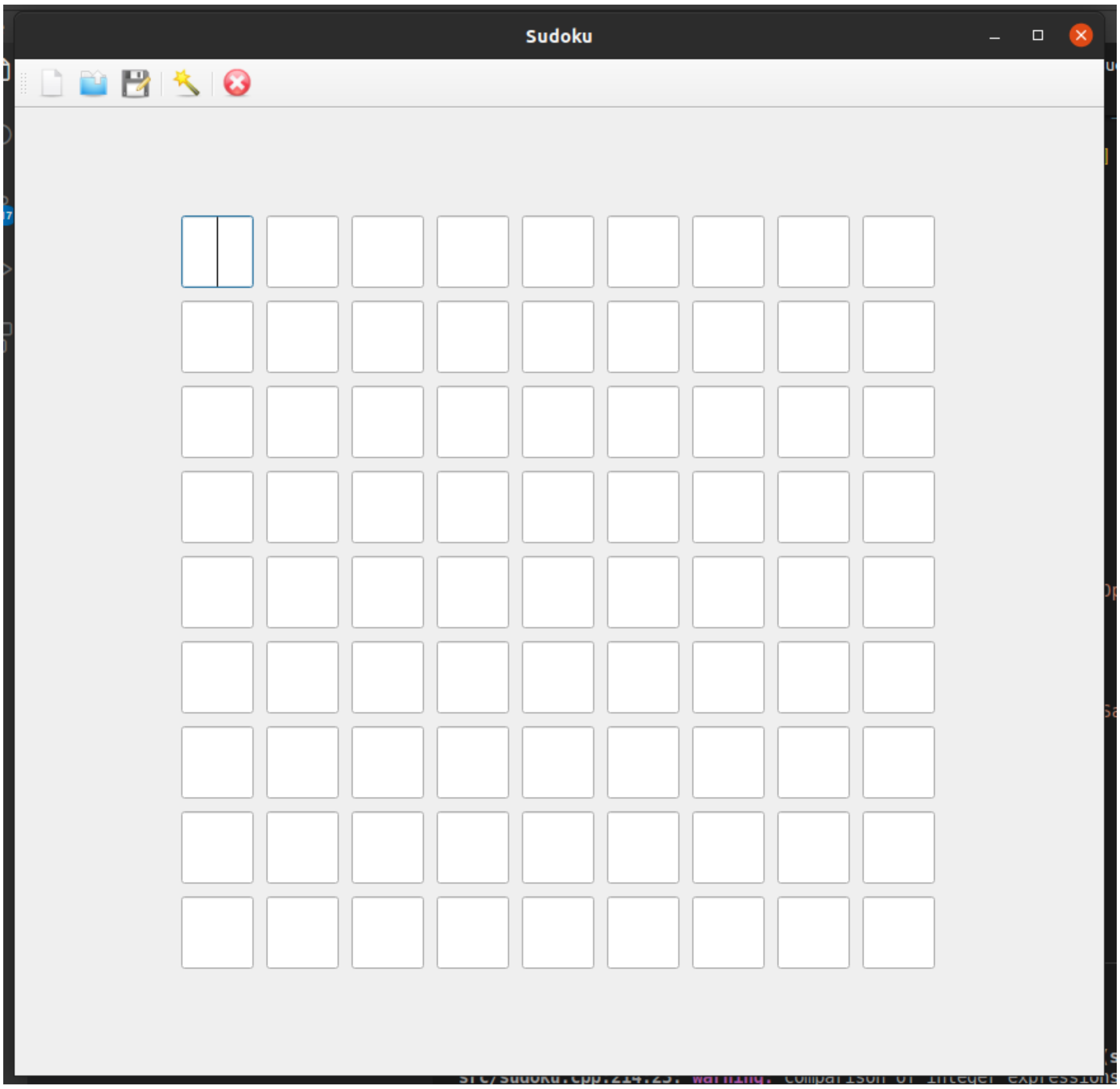
This works as expected:



Sudoku

1	2	6	2	4	3	1	6	2
9	3	1	5	3	4	4	2	1
8	6	3	2	7	2	1	2	4
1	2	6	2	4	3	1	6	2
9	3	1	5	3	4	4	2	1
2	3	6	7	9	4	2	4	2
8	6	3	2	7	2	1	2	4
1	2	6	2	4	3	1	6	2
2	3	6	7	9	4	2	4	2

```
src/sudoku.cpp:214:23: warning: comparison of integer expressions
```

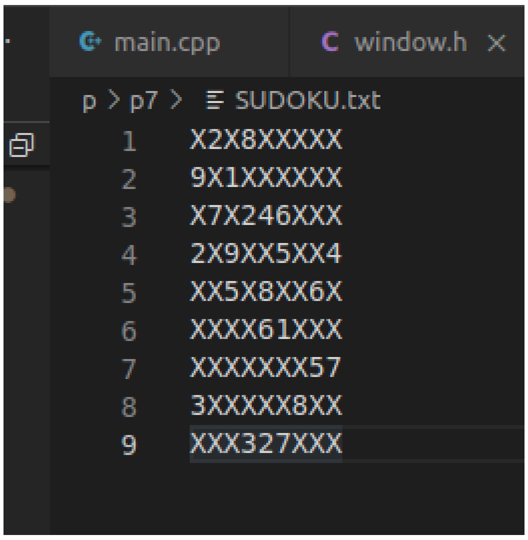


I implemented the open_sudoku slot as follows:

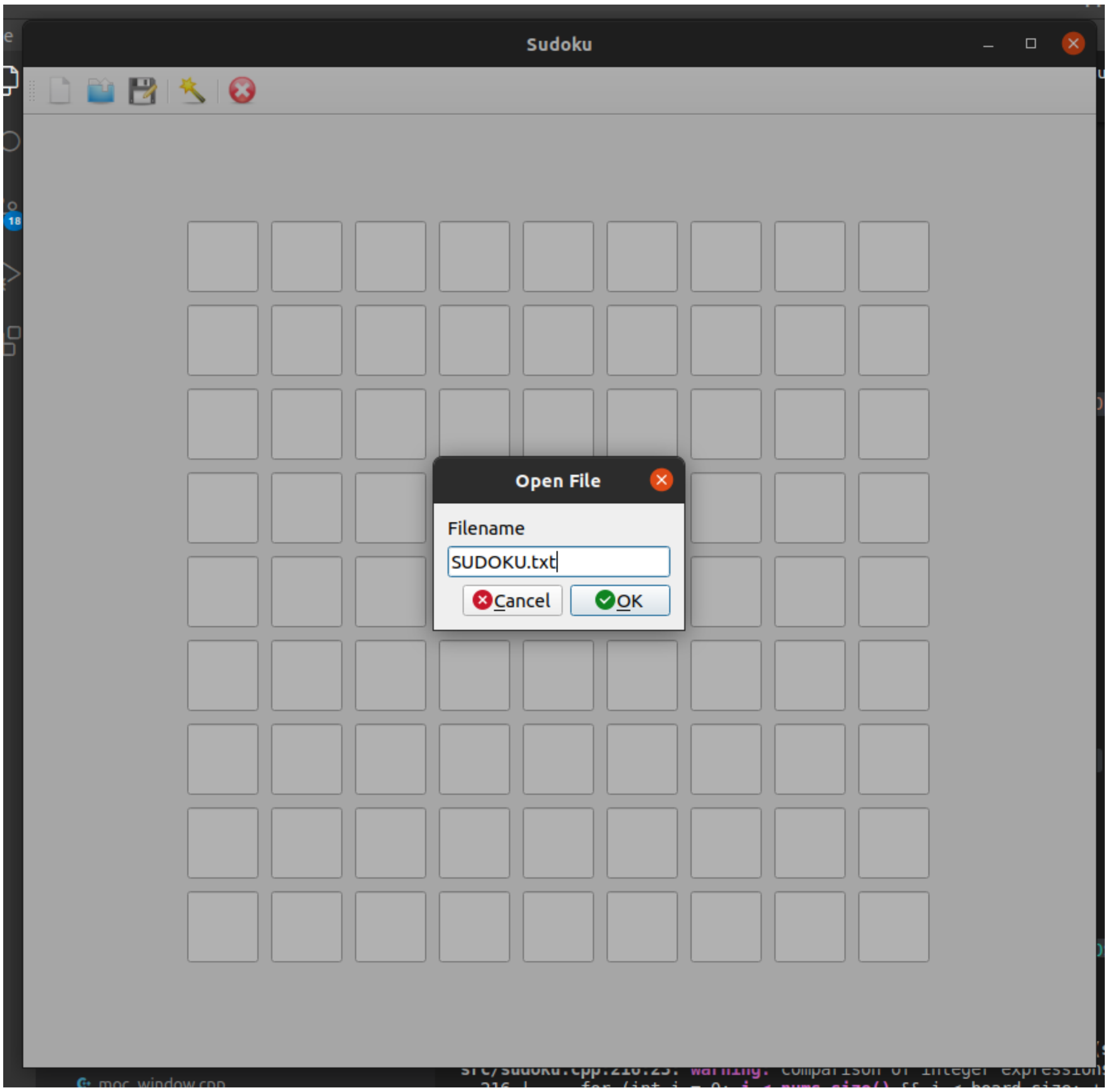
```
void sudoku_grid::open_sudoku()
{
   QString filename = QInputDialog::getText(parent, "Open File",
"Filename");

   QFile file(filename);
   if (!file.open(QIODevice::ReadOnly))
   {
       qWarning("Cannot open file for reading");
   }
   QTextStream stream(&file);

   int line_count = 0;
   while (!stream.atEnd())
```

```
    {
        QString line = stream.readLine();
        if (line.size() != board_size)
        {
            qWarning("Line in file incorrect length");
        }
        else
        {
            for (int i = 0; i < board_size; ++i)
            {
                if (line[i] != 'X')
                {
                    grid_edit[line_count][i]->setText(QString(line[i]));
                }
                else
                {
                    grid_edit[line_count][i]->setText("");
                }
            }
        }
        ++line_count;
    }
    if (line_count != board_size)
    {
        qWarning("Incorrect line count in file");
    }

}
```

The output of this when I load a file:

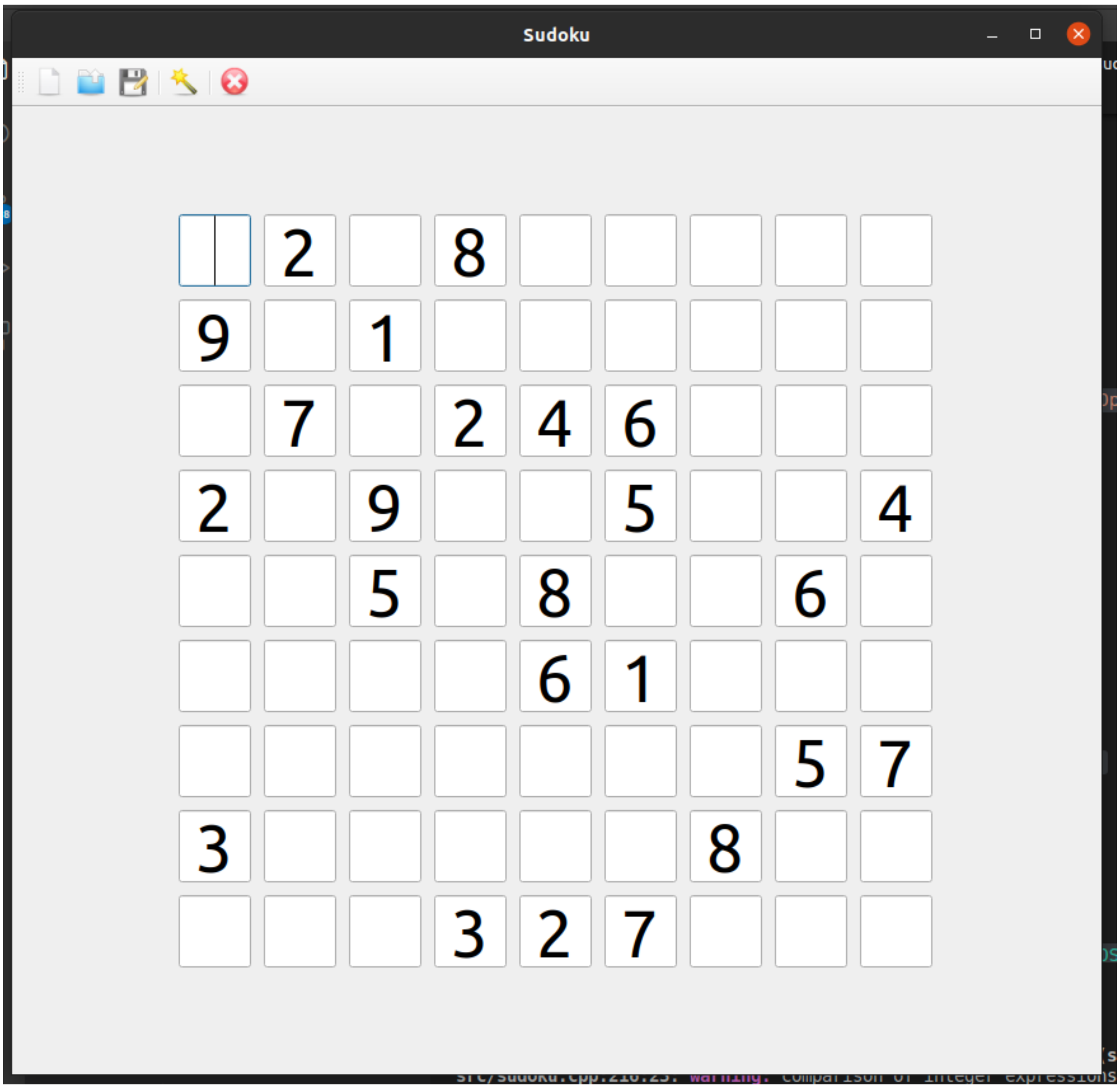


```
X2X8XXXXX
9X1XXXXXX
X7X246XXX
2X9XX5XX4
XX5X8XX6X
XXXX61XXX
XXXXXXX57
3XXXXX8XX
XXX327XXX
```



Filename

SUDOKU.txt

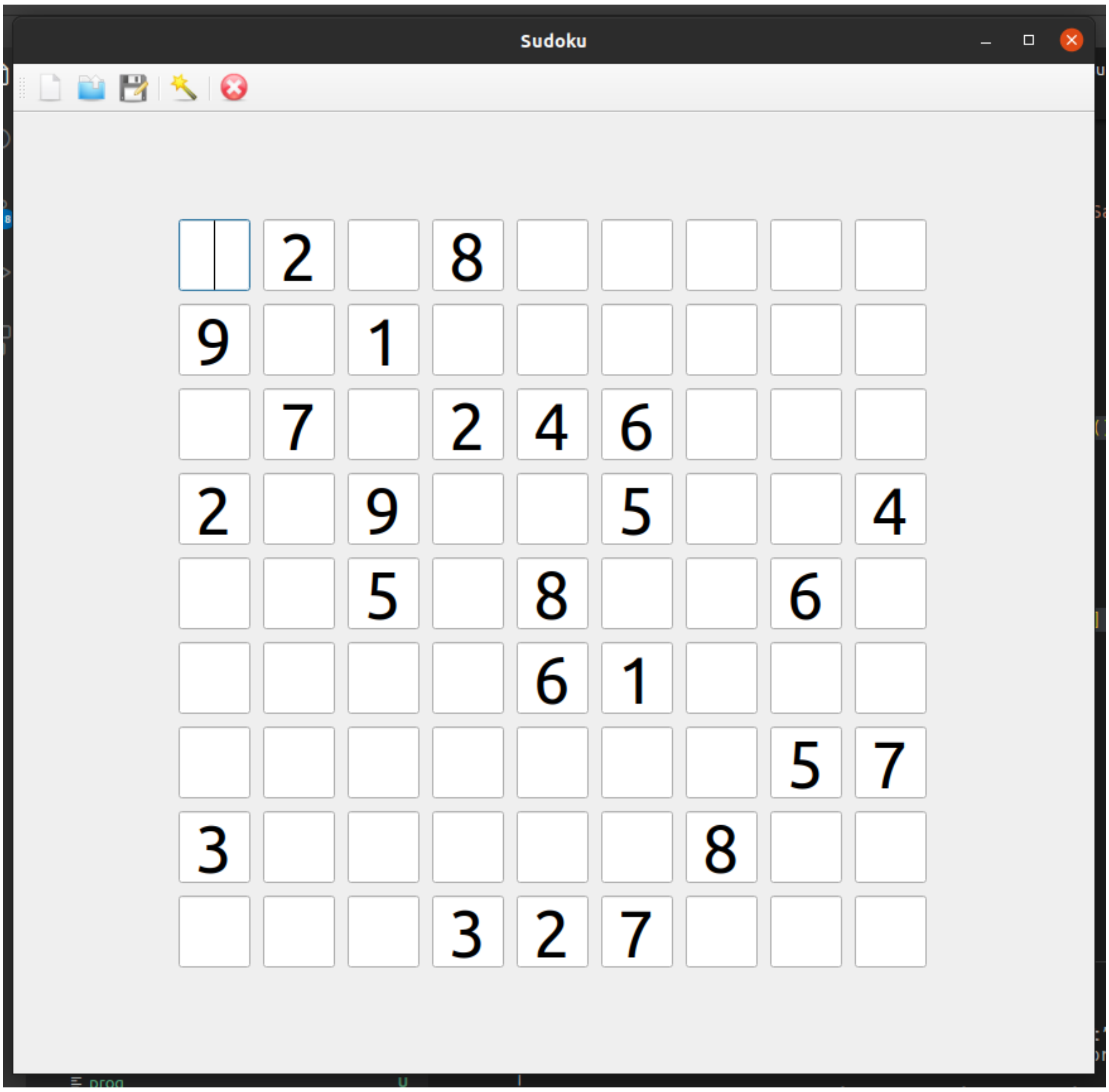


	2		8					
9		1						
	7		2	4	6			
2		9			5			4
		5		8			6	
				6	1			
							5	7
3						8		
			3	2	7			

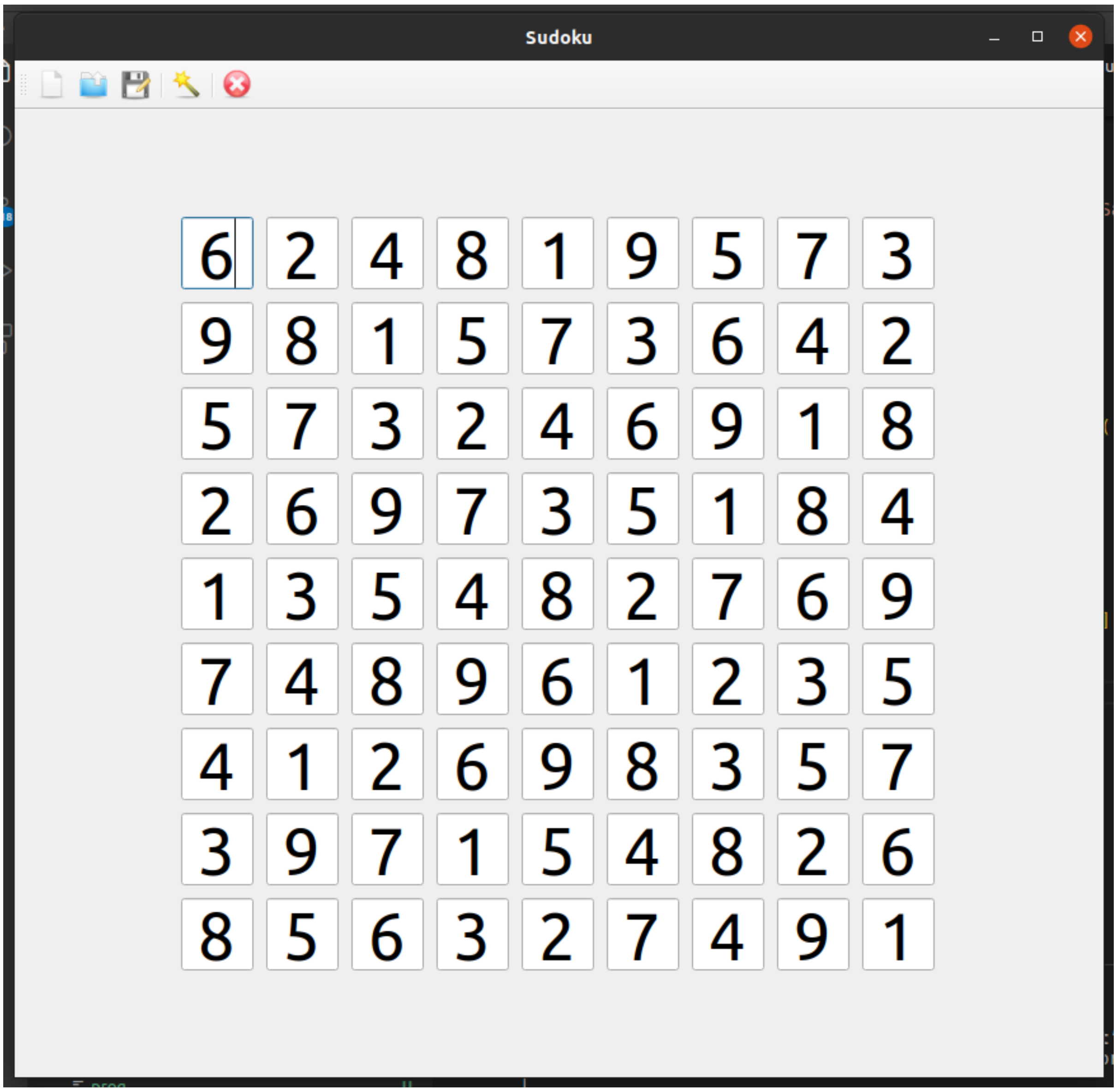
I implemented the solving slot as follows:

```cpp
void sudoku_grid::solve_sudoku()
{
   for (int i = 0; i < board_size; ++i)
   {
       for (int j = 0; j < board_size; ++j)
       {
           grid[i][j] = grid_edit[i][j]->text().toInt();
       }
   }
   solve();
   for (int i = 0; i < board_size; ++i)
   {
       for (int j = 0; j < board_size; ++j)
       {
           grid_edit[i][j]->setText(QString(grid[i][j] + '0'));
       }
   }
}
```

This correctly solves the sudoku:



	2		8					
9		1						
	7		2	4	6			
2		9			5			4
		5		8			6	
				6	1			
							5	7
3						8		
			3	2	7			



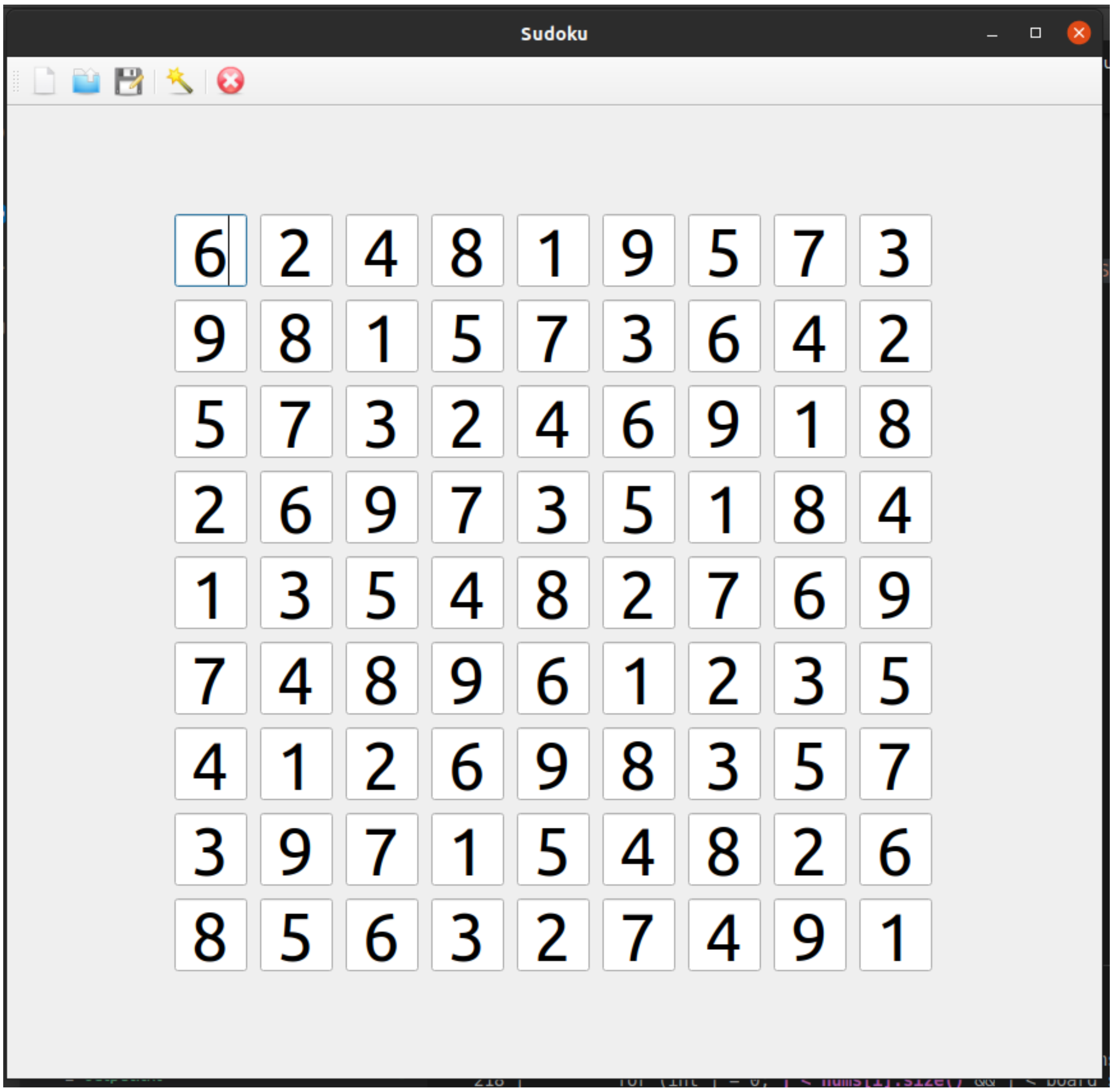
6	2	4	8	1	9	5	7	3
9	8	1	5	7	3	6	4	2
5	7	3	2	4	6	9	1	8
2	6	9	7	3	5	1	8	4
1	3	5	4	8	2	7	6	9
7	4	8	9	6	1	2	3	5
4	1	2	6	9	8	3	5	7
3	9	7	1	5	4	8	2	6
8	5	6	3	2	7	4	9	1

I implemented the sudoku saving as follows:

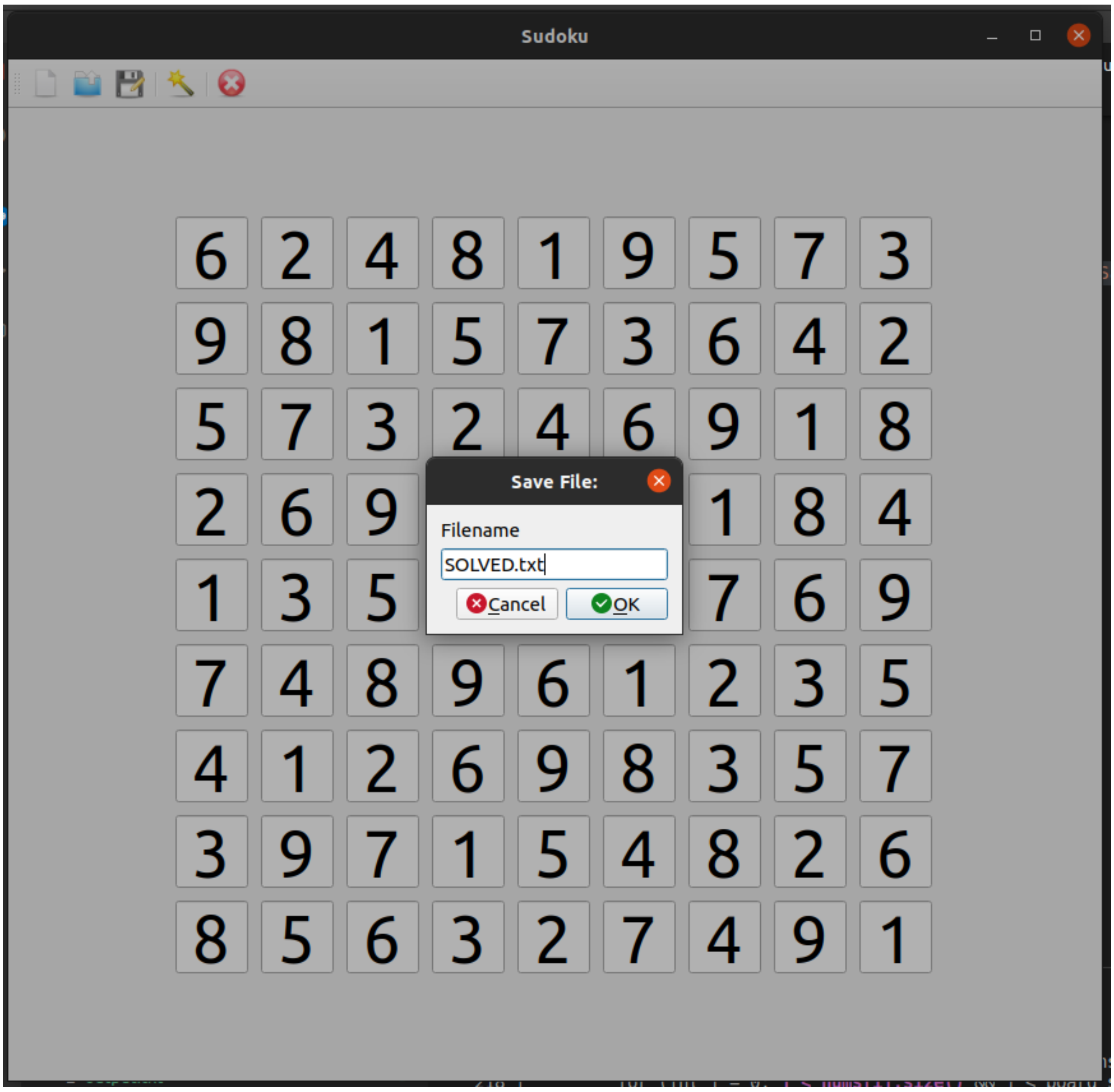
```
void sudoku_grid::save_sudoku()
{
   QString filename = QInputDialog::getText(parent, "Save File:",
"Filename");

   QFile file(filename);
   if (!file.open(QIODevice::WriteOnly))
   {
       qWarning("Cannot open file for writing");
   }
   QTextStream stream(&file);

   for (int i = 0; i < board_size; ++i)
```

```
    {
        QString line;
        for (int j = 0; j < board_size; ++j)
        {
            if (grid_edit[i][j]->text() != "")
            {
                line.append(grid_edit[i][j]->text());
            }
            else
            {
                line.append("X");
            }
        }
        stream << line << '\n';
    }
}
```

I used this so open the SUDOKU.txt file, solve it, and then save it to SOLVED.txt.

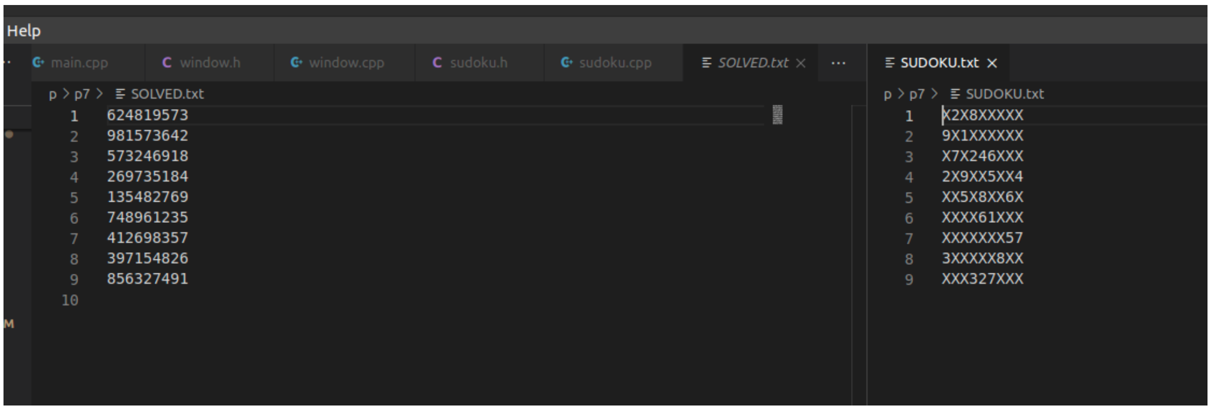


6	2	4	8	1	9	5	7	3
9	8	1	5	7	3	6	4	2
5	7	3	2	4	6	9	1	8
2	6	9	7	3	5	1	8	4
1	3	5	4	8	2	7	6	9
7	4	8	9	6	1	2	3	5
4	1	2	6	9	8	3	5	7
3	9	7	1	5	4	8	2	6
8	5	6	3	2	7	4	9	1



```
6 2 4 8 1 9 5 7 3
9 8 1 5 7 3 6 4 2
5 7 3 2 4 6 9 1 8
2 6 9       1 8 4
1 3 5       7 6 9
7 4 8 9 6 1 2 3 5
4 1 2 6 9 8 3 5 7
3 9 7 1 5 4 8 2 6
8 5 6 3 2 7 4 9 1
```

Filename
SOLVED.txt



```
624819573
981573642
573246918
269735184
135482769
748961235
412698357
397154826
856327491
```

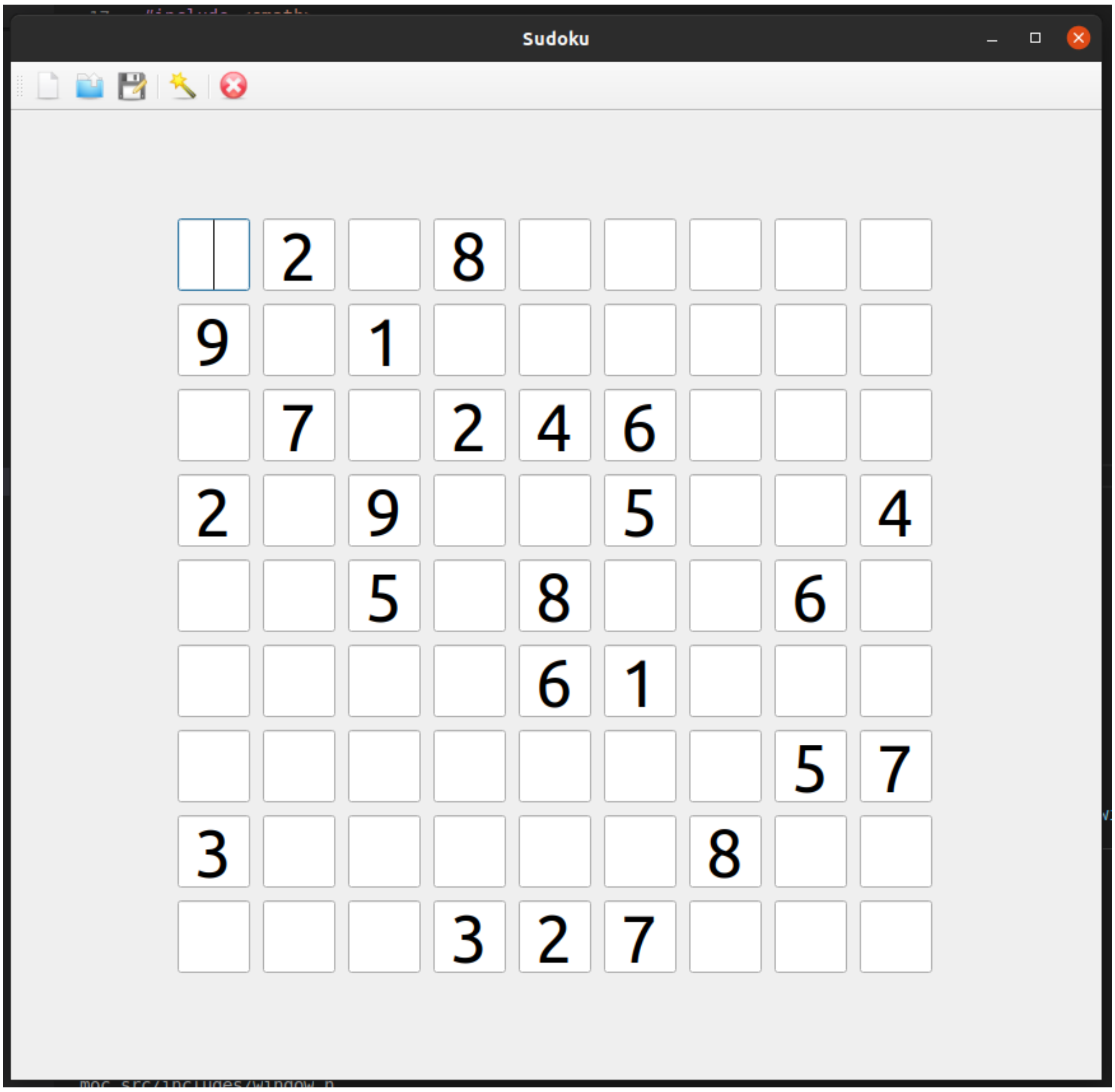
```
X2X8XXXXX
9X1XXXXXX
X7X246XXX
2X9XX5XX4
XX5X8XX6X
XXXX61XXX
XXXXXXX57
3XXXXX8XX
XXX327XXX
```

4 - Optional Additional Work

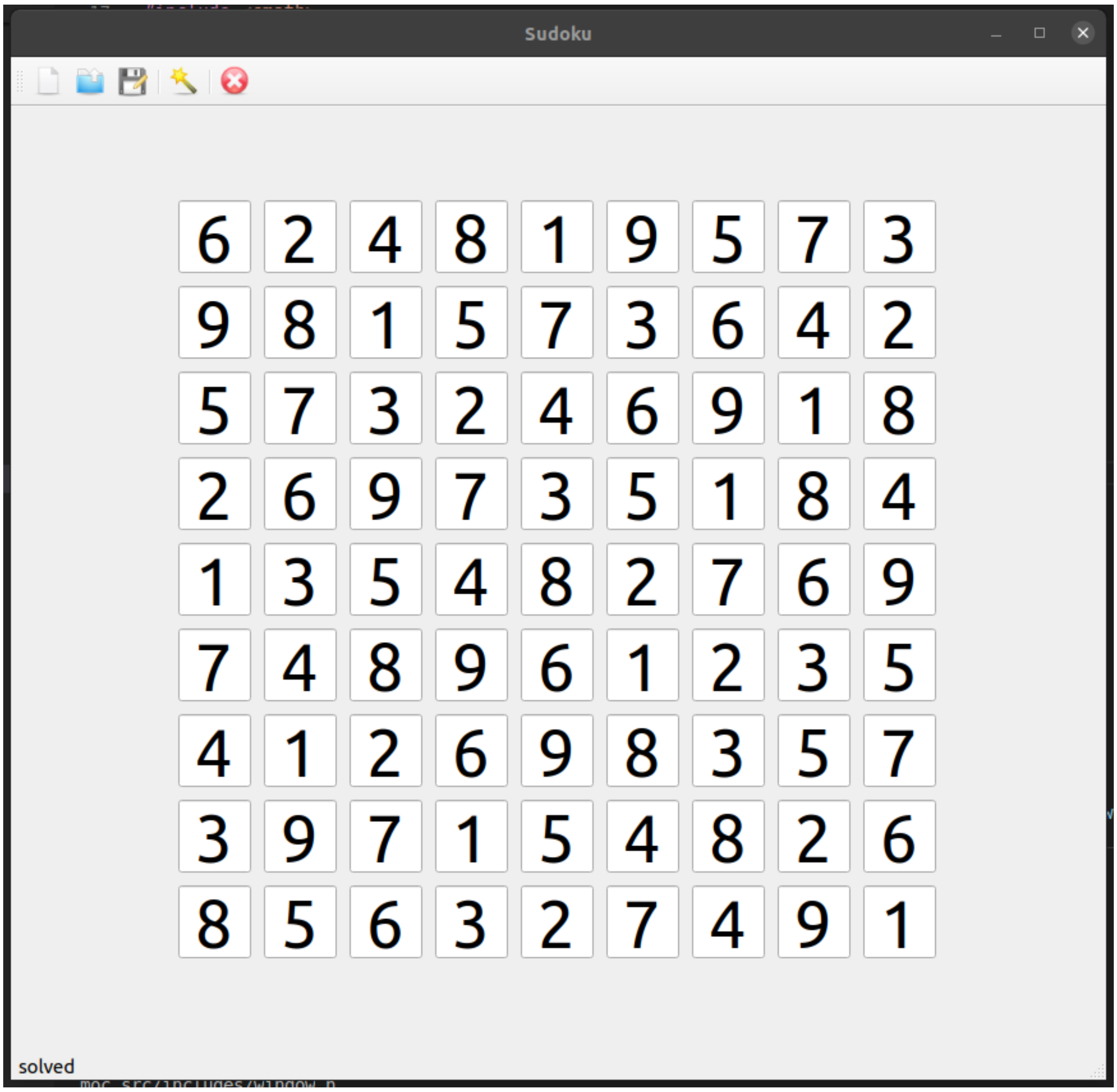
(Puzzle saving was implemented in the previous section)

I implemented the following code to react to the text in one of the input boxes changing to test if the board has been solved and will print in the statusbar in the bottom left if it is finished.

This works as intended:



	2		8					
9		1						
	7		2	4	6			
2		9			5			4
		5		8			6	
				6	1			
							5	7
3						8		
			3	2	7			



```
6 2 4 8 1 9 5 7 3
9 8 1 5 7 3 6 4 2
5 7 3 2 4 6 9 1 8
2 6 9 7 3 5 1 8 4
1 3 5 4 8 2 7 6 9
7 4 8 9 6 1 2 3 5
4 1 2 6 9 8 3 5 7
3 9 7 1 5 4 8 2 6
8 5 6 3 2 7 4 9 1
```

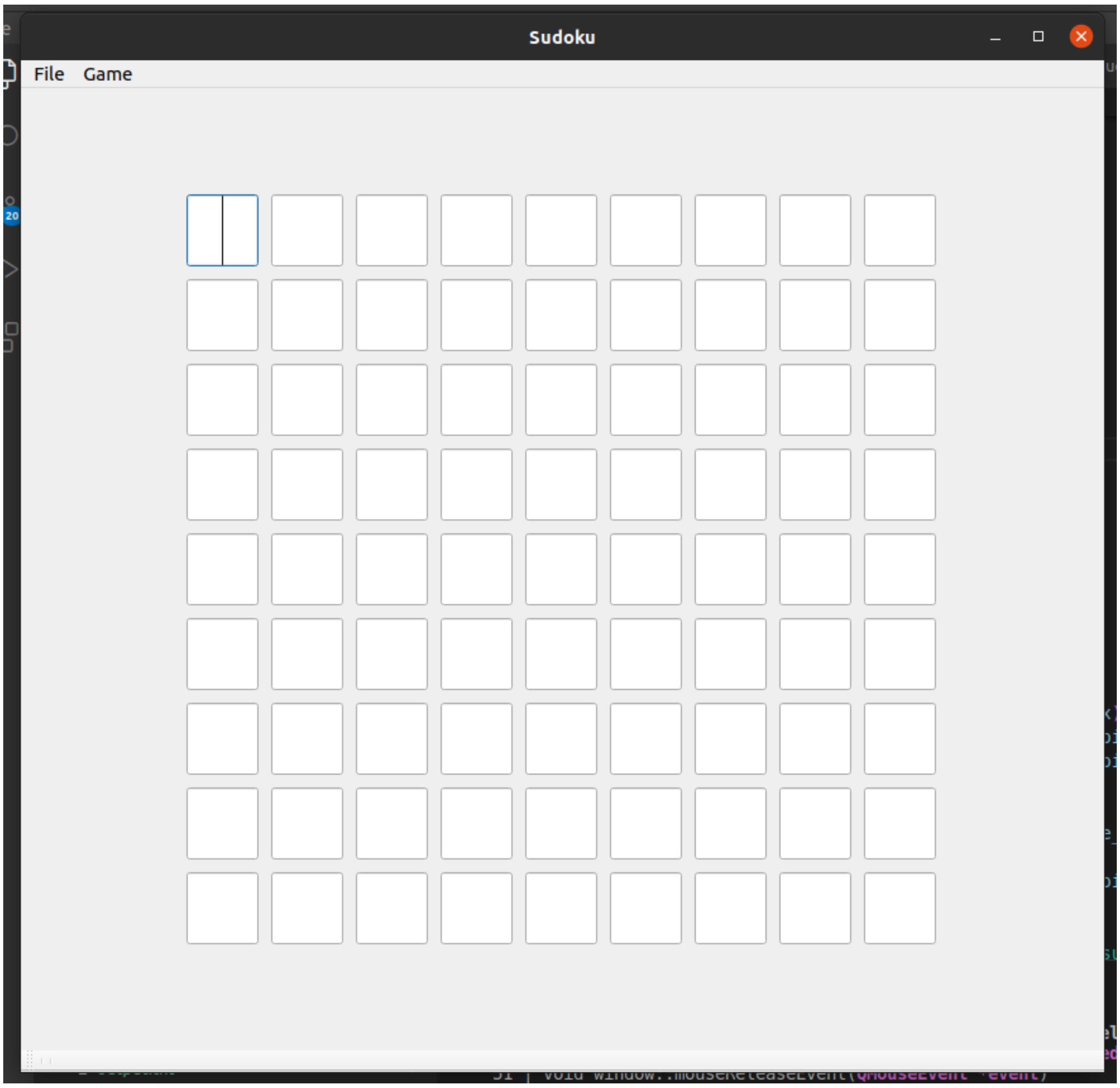
solved

If I edit the constructor of the window to the following, I can put the taskbar into a menu bar.

```
window::window(QWidget *parent) :
   QMainWindow(parent), board(this, 3)
{
   // loading images for toolbar icons
   QPixmap new_pix("img/new.png");
   QPixmap open_pix("img/open.png");
   QPixmap save_pix("img/save.png");
   QPixmap solve_pix("img/wizard.png");
   QPixmap quit_pix("img/quit.png");

   // initializing toolbar
   toolbar = addToolBar("main toolbar");
```

```
    QMenu *file = menuBar()->addMenu("File");
    QMenu *game = menuBar()->addMenu("Game");

    // setting up toolbar icons
    QAction *new_action = file->addAction(QIcon(new_pix), "new file");
    QAction *open_action = file->addAction(QIcon(open_pix), "open file");
    QAction *save_action = file->addAction(QIcon(save_pix), "save file");
    file->addSeparator();
    toolbar->addSeparator();
    QAction *solve_action = game->addAction(QIcon(solve_pix), "solve");
    toolbar->addSeparator();
    QAction *quit_action = file->addAction(QIcon(quit_pix), "quit");

    // connecting signals to their corresponding slots
    connect(new_action, &QAction::triggered, &board,
&sudoku_grid::new_sudoku);
    connect(open_action, &QAction::triggered, &board,
&sudoku_grid::open_sudoku);
    connect(save_action, &QAction::triggered, &board,
&sudoku_grid::save_sudoku);
    connect(solve_action, &QAction::triggered, &board,
&sudoku_grid::solve_sudoku);
    connect(quit_action, &QAction::triggered, qApp, &QApplication::quit);
}
```

I was not able to get an image of this working, since when I click onto my software that I use to screenshot, the menu would close again, however I can get an image of it without the menus opening:

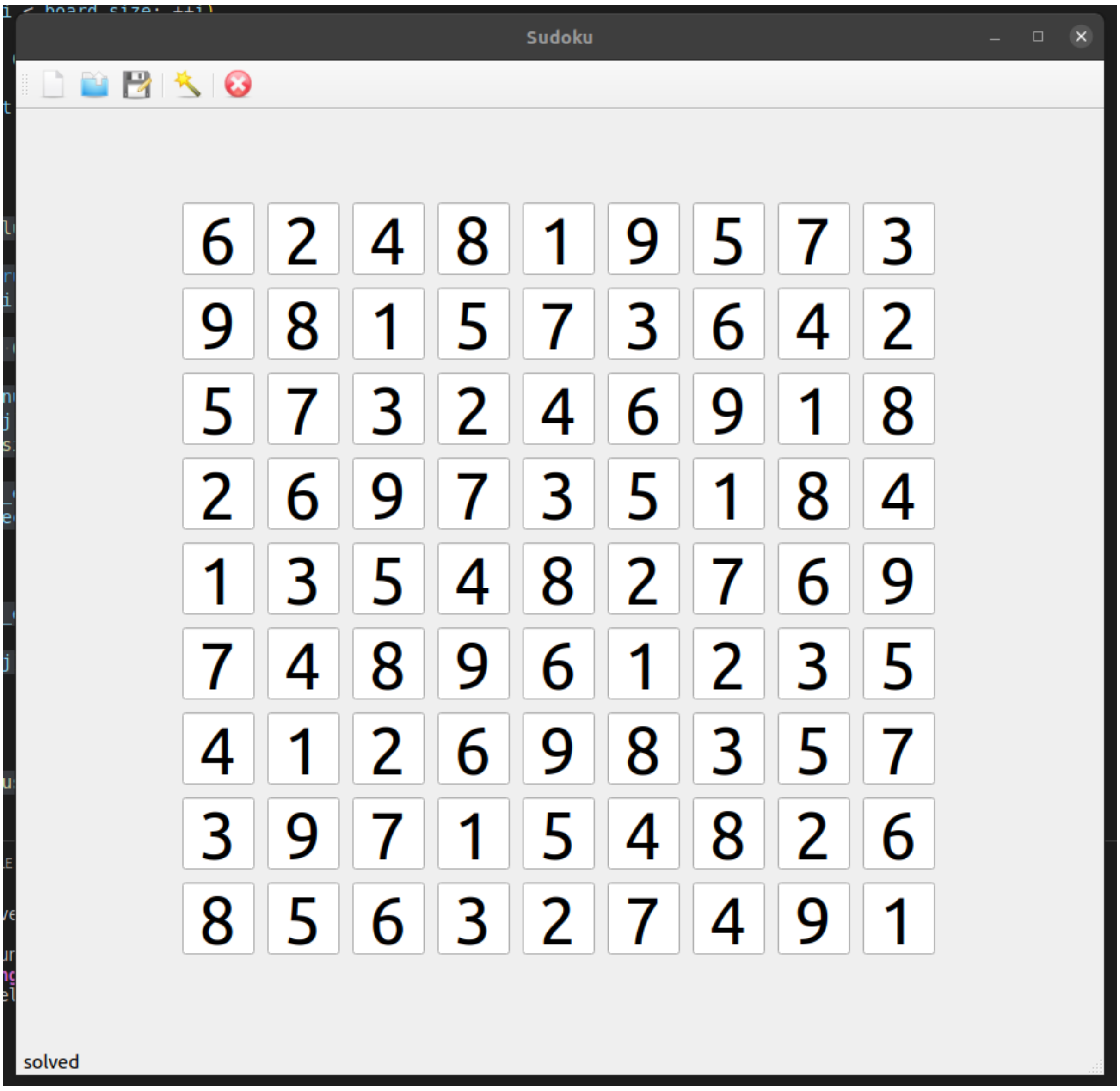


I then implemented some code to tell the user about errors:

```
void sudoku_grid::value_edit()
{
   bool correct = true;
   for (int i = 0; i < board_size; ++i)
   {
       for (int j = 0; j < board_size; ++j)
       {
           int tempnum = grid[i][j];
           grid[i][j] = 0;
           if (!possible_placement(i, j, grid_edit[i][j]->text().toInt()))
           {
               grid_edit[i][j]->setStyleSheet("Color: red;");
               correct = false;
           }
```

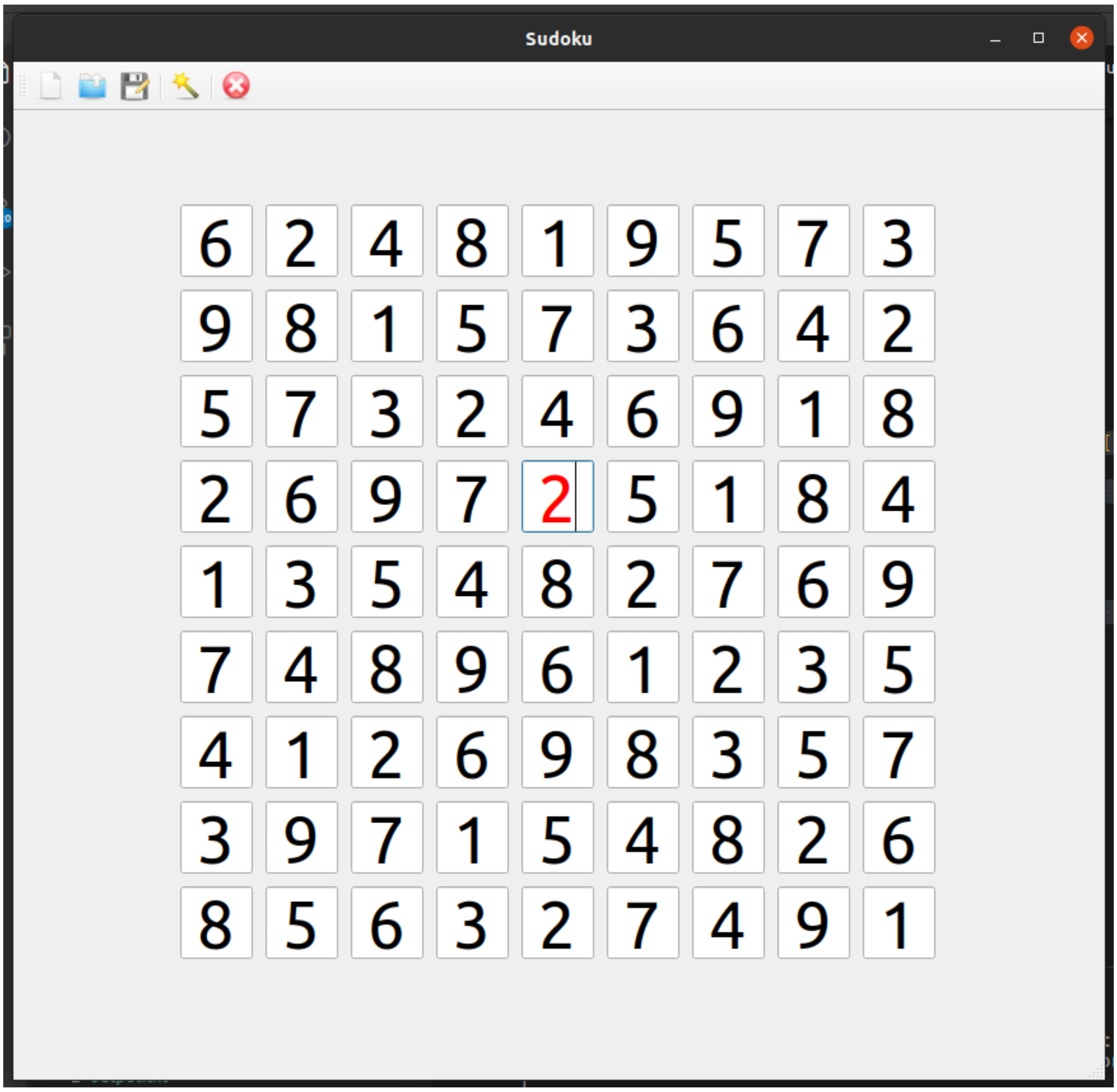
```
            else
            {
                grid_edit[i][j]->setStyleSheet("Color: black;");
            }
            grid[i][j] = tempnum;
        }
    }
    if (correct)
    {
        parent->statusBar()->showMessage("solved");
    }
    else
    {
         parent->statusBar()->clearMessage();
    }
}
```

This produces the following output:



6	2	4	8	1	9	5	7	3
9	8	1	5	7	3	6	4	2
5	7	3	2	4	6	9	1	8
2	6	9	7	3	5	1	8	4
1	3	5	4	8	2	7	6	9
7	4	8	9	6	1	2	3	5
4	1	2	6	9	8	3	5	7
3	9	7	1	5	4	8	2	6
8	5	6	3	2	7	4	9	1

solved



6	2	4	8	1	9	5	7	3
9	8	1	5	7	3	6	4	2
5	7	3	2	4	6	9	1	8
2	6	9	7	2	5	1	8	4
1	3	5	4	8	2	7	6	9
7	4	8	9	6	1	2	3	5
4	1	2	6	9	8	3	5	7
3	9	7	1	5	4	8	2	6
8	5	6	3	2	7	4	9	1