EE441- Programming Assignment 3

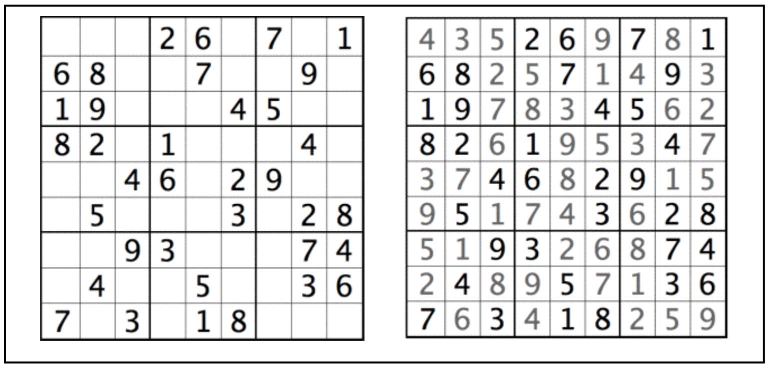
Due Date: 30.12.2024, 23:59

For your questions: Utkucan Doğan – utkucan@metu.edu.tr

This assignment consists of one part. You are going to create a makefile project for this part. Don't forget to write comments to your code as they are also graded. **Use the given code templates from ODTUClass, otherwise your answers will not be graded. You are NOT allowed to use existing data structures provided by the standard library such as std::vector or std::list, and algorithms such as std::sort.**

Sudoku Solver [100 points]

Sudoku is a game where you fill a 9x9 grid with numbers from 1 to 9 such that there will be no repetition of numbers in each row, column and 3x3 subgrids. A sample sudoku puzzle and its solution is given in Figure 1.



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

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

Figure 1: Example sudoku and its solution

Graph coloring problem is a process where each vertex of a graph is colored (or labeled) such that no two adjacent vertices share the same color, with the goal of minimizing the total numbers of colors used. An example coloring with optimal and sub-optimal solutions is given in Figure 2. The maximum number of colors for an optimal solution is $\Delta + 1$ where Δ is the maximum degree in the graph.

A game of sudoku can be considered as a graph coloring problem when the sudoku grid is converted into a graph where each cell in the sudoku grid is a vertex and these vertices are adjacent to the other vertices within the same row, column and subgrid. A simplified version of this representation is given in Figure 3. For sudoku, the number of colors used are exactly 9.

During this homework, you are going to implement a graph, a doubly linked list for auxiliary storage and a recursive brute-force graph coloring algorithm.

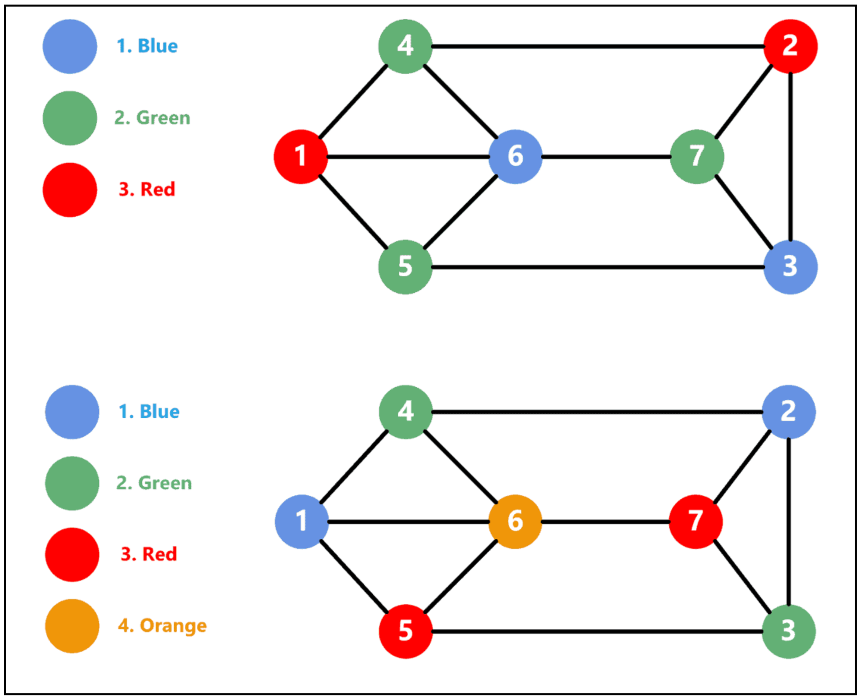


Figure 2: Example graph coloring with optimal (top) and sub-optimal (bottom) solutions

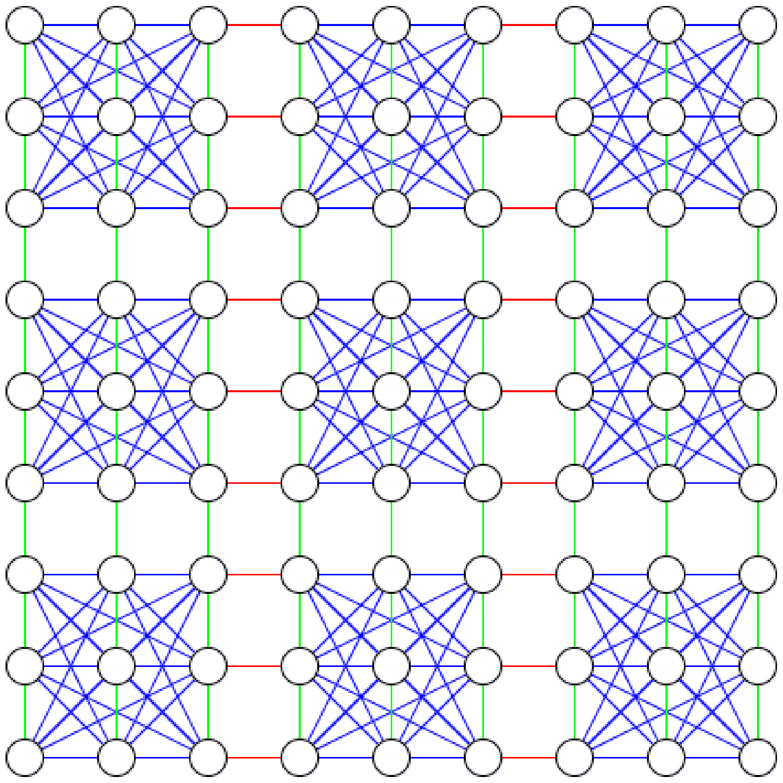


Figure 3: Sudoku grid as a graph

Preliminary Information

During this homework, you are going to be introduced two new concepts that are not discussed in the lectures yet are very useful and powerful for C++ programming.

Iterators

Iterators are simple classes that are used during the iteration process over a list-like class. In terms of linked lists, they are a wrapper class around the node pointers. An iterator class must have an increment, an indirection (dereference) and an equality operator. The increment operator is used for obtaining a new iterator for the next element, the indirection operator is used for getting the element reference from the iterator and the equality operator is used for comparing two iterators.

The list-like class should provide a `begin` and an `end` function that returns appropriate iterators. The iterator that is returned by the `end` function is for checking if the end of the list is reached and should not be dereferenced.

A list-like class should implement both a normal iterator and a constant iterator which is used when the list-like class is constant. This constant iterator holds and returns constant references to the elements, but the constant iterator itself is not constant and therefore can be incremented.

If an iterator is implemented correctly, the following syntax can be used for iterating over an entire list:

```
for (List<int>::Iterator iter = list.begin() ; iter != list.end() ; ++iter) {
    int value = *iter;
    std::cout << value << std::endl;
}
```

An equivalent but simple syntax is provided by C++ if you do not need the iterator itself:

```
for (int value : list) {
    std::cout << value << std::endl;
}
```

This syntax is converted to the previous one during compilation and thus requires the iterator implementation.

Copy vs Move Constructors/Assignment Operators

A copy constructor is a constructor for a class that copies the entire object. For a complex data structure, it needs to allocate new memory and copy the existing data from the other object. The conventional prototype for a copy constructor for class List is:

```
List::List(const List& other);
```

On the other hand, a move constructor moves the allocated memory and all the data from the other object into the newly created one. After a move operation, the other object becomes invalid and should not be used. Move constructors are more efficient than the copy constructors as they require less memory allocation. The prototype for a move constructor for class List is:

```
List::List(List&& other);
```

Notice the use of double ampersand in the type which is a special syntax and differs from the ordinary reference. However, it is still a type of reference and changes done to the other object in this function affects the object outside. The other object should be modified to prevent an early destruction of the values moved. To invoke a move constructor, `std::move` is used when passing the other object as parameter. Move assignment operator, just like the copy assignment operator, is the assignment variant of the constructor.

If a function returns a newly generated class, its return value does not require to be copied or moved. This concept in C++ is known as "copy elision".

The following code shows the syntax for calling the copy and move constructors and which function is called for each syntax.

```
List create();

int main()
{
    List list1;
    List list2(list1);           // copy constructor is called
    List list3 = list1;          // copy constructor is called
    list2 = list3;               // copy assignment is called

    List list4(std::move(list2));  /* move constructor is called if exists,
                                      otherwise copy constructor is called */
    List list5 = std::move(list3); /* move constructor is called if exists,
                                      otherwise copy constructor is called */
    list4 = std::move(list5);      /* move assignment is called if exists,
                                      otherwise copy assignment is called */

    List list6(create());        // copy elision
    List list7 = create();       // copy elision
    list7 = create();            /* move assignment is called if exists,
                                    otherwise copy assignment is called */
}
```

Important: The template project for this homework also includes a memory leak detector unit which warns you when an allocated memory is not deleted. Your implementations should not cause a memory leak, otherwise you may lose points.

Part 1 – Doubly Linked List [50 points]

Implement a doubly linked list class by completing the implementation given in "`list.hpp`".

1. **[15 points]** Implement a destructor that deletes all the nodes. Implement a proper copy constructor, move constructor, copy assignment operator and move assignment operator. Move assignment should swap the pointers to allow the destruction of the already existing elements.
2. **[5 points]** Implement the helper functions `initiate`, `deplete` and `hook`.
    a. `initiate` is used for inserting to an empty list.
    b. `deplete` is used for removing from a list with size one.
    c. `hook` is used for connecting two nodes.
3. **[5 points]** Implement the functions `push_back`, `push_front`, `pop_back` and `pop_front`.
    a. `push_back` is used for pushing an element to the end of the list.
    b. `push_front` is used for pushing an element to the start of the list.
    c. `pop_back` is used for getting and removing the last element of the list. It should throw `std::logic_error` if the list is empty.
    d. `pop_front` is used for getting and removing the first element of the list. It should throw `std::logic_error` if the list is empty.
4. **[5 points]** Complete the implementation of the `Iterator` and `ConstIterator`.
    a. `operator==` and `operator!=` is used for comparing two iterators.
    b. `operator*` is used for accessing the data.
    c. `next` is used for obtaining a new iterator that comes after this iterator. It should throw `std::range_error` if next iterator cannot be obtained.
    d. Prefix increment operator is used for updating this iterator to the next one and returns the new value. It should throw `std::range_error` if next iterator cannot be obtained.
    e. Postfix increment operator is used for updating this iterator to the next one and returns the old value. It should throw `std::range_error` if next iterator cannot be obtained.
5. **[10 points]** Implement the function `insert` which inserts a new node to the point before the given iterator.

    Note: `list.insert(list.begin(), data)` is equivalent to `list.push_front(data)` and `list.insert(list.end(), data)` is equivalent to `list.push_back(data)`
6. **[10 points]** Implement the function `remove` which removes the node corresponds to the given iterator. It should throw `std::logic_error` if the list is empty or the end iterator is tried to be removed.

Part 2 – Graph [40 points]

Implement an undirected graph by completing the implementation given in "`graph.cpp`".

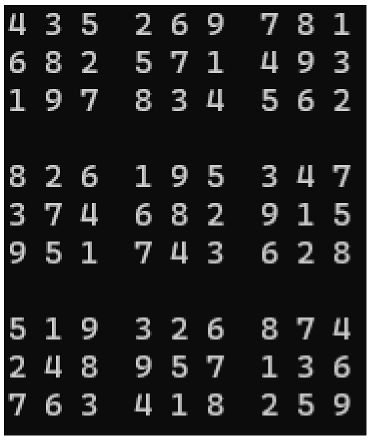
1. **[1 point]** Implement the function `Graph::Vertex::color(int color)` which updates the color to the given input if none of its neighbors uses that color. Returns true if updated successfully, otherwise returns false. Note that the color value zero is used as unassigned color and the color should always be updated if the input is zero.
2. **[5 points]** Implement the function `Graph::Vertex::add_neighbor` which adds the other vertex into the neighbor list if it is not already in it. It should also add this vertex to the other vertex's neighbor list since the graph is undirected. This function should do nothing if the input is `nullptr`.
3. **[5 points]** Implement the function `Graph::Vertex::remove_neighbor` which removes the other vertex from the neighbor list if it is in it. It should also remove this vertex from the other vertex's neighbor list since the graph is undirected. This function should do nothing if the input is `nullptr`.
4. **[5 points]** Implement a destructor that deletes all the vertices. Implement a proper copy constructor, move constructor, copy assignment operator and move assignment operator.
5. **[1 point]** Implement the function `Graph::add_vertex` which adds a new vertex with incremental ID. ID of the vertices should start from one.
6. **[1 point]** Implement the function `Graph::operator[]` which gives the vertex pointer for the given ID. It should throw `std::domain_error` if the vertex with given id does not exist.
7. **[1 point]** Implement the function `Graph::connect` which connects two vertices given by their ID.
8. **[1 point]** Implement the function `Graph::max_color` which gives the maximum possible color. Please note that this value is equal to the maximum degree in the graph plus one.
9. **[20 points]** Implement a recursive graph coloring algorithm by completing the functions `Graph::color` and `Graph::color_helper`. `Graph::color_helper` should follow these steps:
    a. If the vertices are depleted, return true
    b. If this vertex is already colored, skip to the next one
    c. If this vertex is not colored, try the possible colors. If coloring for the rest of the graph is done with a selected color for this vertex, return true.
    d. If no color selected allows the correct coloring of the rest of the graph, reset the color (to zero) and return false.

    `Graph::color` should call `Graph::color_helper` for every vertex. If coloring is not done even for one vertex, return false. (Note: for a connected graph, calling `Graph::color_helper` only for one vertex is enough but if the graph is disconnected `Graph::color_helper` should be called for one vertex from each disconnected subgraph. Calling for every vertex ensures this.)

Part 3 – Sudoku and Main Function [10 points]

Sudoku class inherits the graph class for solving the game.

1. Complete the implementation of the Sudoku class. Constructor should call `create("data/sudoku.txt")` to obtain the correct connectivity (`create` and `set_clues` functions are provided to you in "parser.cpp"). `Sudoku::max_color` should return the correct value for the sudoku puzzle. `Sudoku::print` should print the board like in the Figure 4.
2. Run the given main function to check your implementation. If your implementation is <u>incomplete</u>, modify the main function so that it can demonstrate all the features you have implemented.



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

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

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

Figure 4: Expected output of the print function