

Programming Assignment: Flood-Fill

Topics and references

- Flood-fill algorithm for existence of a path searching
- Recursive and iterative implementations
- Depth-first and Breadth-first strategies

Task

This assignment helps to review what you have learned in Data Structures course and define fundamental structures and algorithms for your future assignments.

You can find the complete list of structures and functions to implement in form of pseudocode on the Moodle web page and in the assignment framework that is provided and accessible on the SVN server.

Complete instruction and explanation **will be given in the class**.

Your implementation must pass all given tests in order to get the full mark for the assignment.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions. Since submission details will remain the same for all programming assessments, this portion will be skipped in future documents detailing labs and assignments.

Source files

You have to submit the header `functions.h` and source file `functions.cpp`.

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <stack>
#include <algorithm>

#include "data.h"

#define UNUSED(expr) (void)expr;

namespace AI
{
    // Domain specific functor that returns adjacent nodes
    class GetMapAdjacents : public GetAdjacents
    {
    {
        int* map; // the map with integers where 0 means an empty cell
        int size; // width and height of the map in elements

    public:

        GetMapAdjacents(int* map=nullptr, int size=0)
```

```

        : GetAdjacents(), map{ map }, size{ size }
    {
    }

    virtual ~GetMapAdjacents()
    {
    }

    std::vector<AI::Node*> operator()(Key key)
    {
        UNUSED(key)

        std::vector<AI::Node*> list = {};

        // Find and return all empty adjacent cells

        return list;
    }
};

// Domain specific functor that returns shuffled adjacent nodes
class GetMapStochasticAdjacents : public GetMapAdjacents
{
public:

    GetMapStochasticAdjacents(int* map, int size)
        : GetMapAdjacents{ map, size }
    {
    }

    std::vector<AI::Node*> operator()(Key key)
    {
        UNUSED(key)

        std::vector<AI::Node*> list = {};

        // Find and return all empty adjacent cells
        // Use the base class operator() and then shuffle the result

        return list;
    }
};

// Wrappers that provide same interface for queue and stack

struct Interface
{
    virtual void clear() = 0;

    virtual void push(Node* pNode) = 0;

    virtual Node* pop() = 0;
};

struct Queue : Interface //...
{
    void clear()
    {

```

```

        //...
    }

    void push(Node* pNode)
    {
        UNUSED(pNode)
        //...
    }

    Node* pop()
    {
        Node* pNode = nullptr;
        //...
        return pNode;
    }
};

struct Stack : Interface //...
{
    void clear()
    {
        //...
    }

    void push(Node* pNode)
    {
        UNUSED(pNode)
        //...
    }

    Node* pop()
    {
        Node* pNode = nullptr;
        //...
        return pNode;
    }
};

// Recursive Flood Fill
class Flood_Fill_Recursive
{
    GetAdjacents* pGetAdjacents;

public:
    Flood_Fill_Recursive(GetAdjacents* pGetAdjacents)
        : pGetAdjacents{ pGetAdjacents }
    {
    }

    void run(Key key, int color)
    {
        UNUSED(key)
        UNUSED(color)

        // Implement the flood fill
    }
};

```

```

// Iterative Flood Fill
// Type T defines is it depth- or breadth-first
template<typename T>
class Flood_Fill_Iterative
{
    GetAdjacents* pGetAdjacents;
    T openlist;

public:
    Flood_Fill_Iterative(GetAdjacents* pGetAdjacents)
        : pGetAdjacents{ pGetAdjacents }, openlist{}
    {
    }

    void run(Key key, int color)
    {
        UNUSED(key)
        UNUSED(color)

        // Implement the flood fill
    }
};

} // end namespace

#endif

```

```

#include "functions.h"

namespace AI
{

} // end namespace

```

Compiling, executing, and testing

Run `make` with the default rule to bring program executable `main.out` up to date:

```
$ make
```

Or, directly test your implementation by running `make` with target `test`:

```
$ make test
```

If the `diff` command in the `test` rule is not silent, then one or more of your function definitions is incorrect and will require further work.

To make a memory leak test, run `make` with target `leak`:

```
$ make leak
```

Make sure that the output does not show any memory related issue.

File-level documentation

Every **edited by student** source and header file *must* begin with a *file-level* documentation block. This documentation serves the purpose of providing a reader the [raison d'être](#) of this source file at some later point of time (could be days or weeks or months or even years later). This module will use [Doxygen](#) to tag source and header files for generating html-based documentation. An introduction to Doxygen and a configuration file is provided on the module web page. Here is a sample for a C++ source file:

```
/*!*****  
\file    functions.cpp  
\author  Vadim Surov, <Your Name>  
\par     DP email: vsurov\@digipen.edu, <Your Email>  
\par     Course: CS380  
\par     Section: A  
\par     Programming Assignment 1  
\date    04-30-2021  
  
\brief  
    This file has declarations and definitions that are required for submission  
******/
```

Function-level documentation

Every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value. In team-based projects, this information is crucial for every team member to quickly grasp the details necessary to efficiently use, maintain, and debug the function. Certain details that programmers find useful include: what does the function take as input, what is the output, a sample output for some example input data, how the function implements its task, and importantly any special considerations that the author has taken into account in implementing the function. Although beginner programmers might feel that these details are unnecessary and are an overkill for assignments, they have been shown to save considerable time and effort in both academic and professional settings. Humans are prone to quickly forget details and good function-level documentation provides continuity for developers by acting as a repository for information related to the function. Otherwise, the developer will have to unnecessarily invest time in recalling and remembering undocumented gotcha details and assumptions each time the function is debugged or extended to incorporate additional features. Here is a sample for function `substitute_char`:

```
/*!*****  
\brief  
    Replaces each instance of a given character in a string with  
    other given characters.  
  
\param string  
    The string to walk through and replace characters in.  
  
\param old_char  
    The original character that will be replaced in the string.  
  
\param new_char  
    The character used to replace the old characters
```

```
\return
```

```
The number of characters changed in the string.
```

```
*****/
```

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit `functions.cpp` and `functions.h`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
 - **F** grade if your `functions.cpp` doesn't compile with the full suite of `g++` options.
 - **F** grade if your `functions.cpp` doesn't link to create an executable.
 - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will provide a proportional grade based on how many incorrect results were generated by your submission. **A+** grade if output of function matches correct output of auto grader.
 - A deduction of one letter grade for each missing documentation block in `functions.cpp` and `functions.h`. Your submission `functions.*` (if it was edited by you) must have **one** file-level documentation block and **one** function-level documentation blocks. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an **A+** grade and one documentation block is missing, your grade will be later reduced from **A+** to **B+**. Another example: if the automatic grader gave your submission a **C** grade and the two documentation blocks are missing, your grade will be later reduced from **C** to **E**. Likewise, your submission `functions.h` must have **one** file-level documentation block and **one** function-level documentation block.