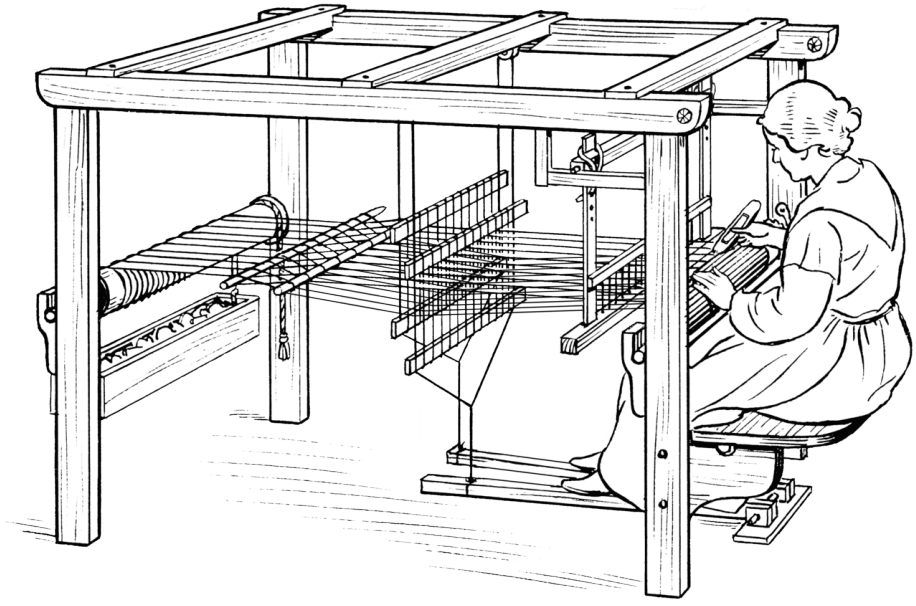
MathAlgorithm
A General Purpose Math and Algorithm Library



Introduction

The MathAlgorithm library provides commonly used math and algorithms used in Computer Science education in the C++ language. This book provides the documentation of the Library. Images, diagrams along with mathematical concepts are provided throughout.

`Algorithm` namespace

The `Algorithm` namespace provides a collection of algorithm operations that include sorting, partition, operations on sets among other operations.

Insertion Sort

The library provides an **Insertion Sort** implementation through the `insertion_sort()` function.

```
20 void insertion_sort(std::vector<Data>& arr)
21     {
22         for (std::size_t i=1; i < arr.size(); ++i) {
23             insert(arr, i, arr[i]);
24         }
25     }
```

The `insert()` function is implemented as,

```
8  void insert(std::vector<Data>& arr, std::size_t pos, Data& value)
9      {
10         std::size_t i = pos - 1;
11
12         while ( (i >= 0) && (arr[i] > value) ) {
13             arr[i + 1] = arr[i];
14             --i;
15         }
16
17         arr[i + 1] = value;
18     }
```

The **Insertion Sort** algorithm can be described as,

The complexity of the insertion sort algorithm is $O(n)$ for best case, $O(n^2)$ for average case and $O(n^2)$ for the worst case. The complexity is driven by the single while loop in the worst case,

```
12         while ( (i >= 0) && (arr[i] > value) ) {
13             arr[i + 1] = arr[i];
14             --i;
15         }
```

Graph Algorithms

Graph algorithms require data structures to represent input, intermediate, and output data. The library provides various data structures for the user to use to represent graph data. Here are the various data structures available in the library.

`Edge` Data Structure

The `Edge` data structure is used to represent an edge `(u,v)` between two vertices `u` and `v` in a graph. It is the base type that has different derived types depending on usage.

Edge
+u: int +v: int +weight: float

The starting or first vertex is identified by a unique integer value of u, and the ending or next vertex is identified as v. A weight is also provided to allow for edge weighted graph representation. A vertex can have many edges in a graph, so the value of u can appear many times. A sequence of edges could be represented by $(u_0, v_0, w_0), (u_0, v_1, w_1), (u_0, v_2, w_2)$.

`Graph` class

A graph is represented as a `Graph` class which encapsulates the graph data and provides unary operations, getters and setters.

Graph
-maxVertices: int -directed: bool -edges: list<Edge>
+Graph(int maxVertices, bool directed = false) +load(std::string_view fileName): bool +isDirected(): bool +numVertices(): int +isEdge(int u, int v): bool +edgeWeight(int u, int v): float +addEdge(int u, int v, float weight): bool +removeEdge(int u, int v): bool +friend operator<<(std::ostream& os, const Graph& graph): std::ostream&

An simple example of using the graph and an external json file to load is,

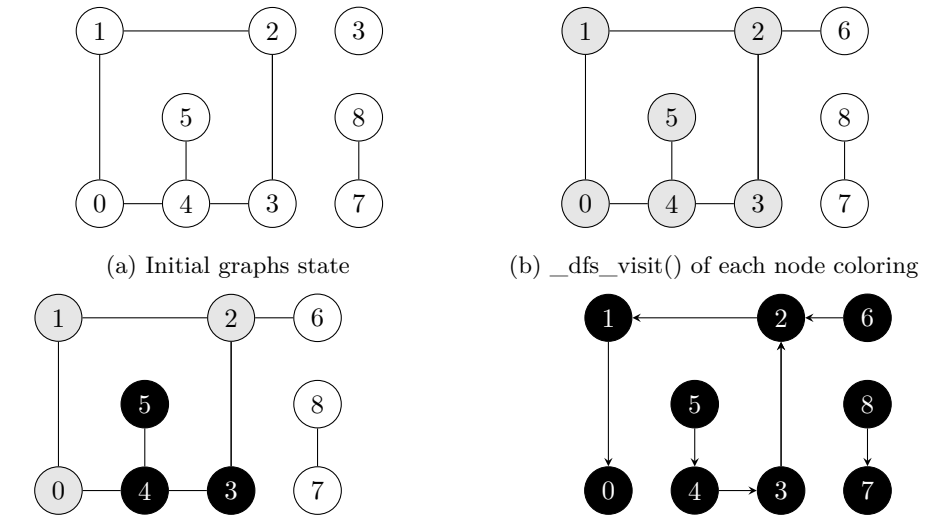
```
19
20          graph.load("data/test.json");
21          std::cout << "\tOutput: ";
22          std::cout << graph;
23
```

The output is,

```
Output: { (1,2),(2,3),(1,4), }
```

And the json file is,

```
[
    {
        "u" : 1,
        "v" : 2,
        "weight" : 1.0
    },
    {
        "u" : 2,
        "v" : 3,
        "weight" : 0.5
    },
    {
        "u" : 1,
        "v" : 4,
        "weight" : 1.0
    }
]
```



(a) Initial graphs state

(b) _dfs_visit() of each node coloring

(c) _dfs_visit() with some finished vertices

(d) _dfs_visit() final state

Figure 2: Depth-first search example

The `Algo::dfs(const Types::Graph& graph, int startId, std::vector<int>& predecessor)` function provides the `DEPTH-FIRST SEARCH` algorithm on a graph object and outputs a predecessor list to determine the paths taken back to the start for each child vertices.

```
17 void dfs(const Types::Graph& graph, int s, std::vector<int>& predecessor) {
18         std::vector<int> vertices = graph.getVertices();
19         std::vector<Types::VertexColor> color(vertices.size());
20         std::vector<int> discovered(vertices.size());
21         std::vector<int> finished(vertices.size());
22         std::size_t counter = 0;
23 
24         std::fill(color.begin(), color.end(), Types::VertexColor::White);
25         std::fill(discovered.begin(), discovered.end(), -1);
26         std::fill(finished.begin(), finished.end(), -1);
27 
28         _dfs_visit(graph, discovered, finished, predecessor, color, s, counter);
29 
30         for (const auto v : vertices) {
31             if (color[v] == Types::VertexColor::White) {
32                 _dfs_visit(graph, discovered, finished, predecessor, color, v, counter);
33             }
34         }
35     }
```

The `Algo::_dfs_visit(const Types::Graph& graph, std::vector<int>& discovered, std::vector<int>& finished, std::vector<int>& predecessor, std::vector<Types::VertexColor>& color, int id, std::size_t counter)` provides the recursive calls to walk the entire graph. The `discovered`, `finished`, `color` and the final ouput `predecessor` lists are passed through each `_dfx_visit(...)` call. These are free functions and not a class object. This implementation is for an explicit example of where the affected data structures are used.

The pseudo code for the `DEPTH-FIRST SEARCH` is shown in Algorithm 1,

Algorithm 1: DEPTH-FIRST SEARCH

Data: Graph V, start vertex s
Result: Arrays of d, discovered, *pred*, predecessor vertices, and f, finished counter

```
begin
    foreach v ∈ V do
        d[v] = f[v] = pred[v] = -1
        color[v] = White
    end
    counter = 0
    dfs_visit(s)
    foreach v ∈ V do
        if color[v] = White then
            dfs_visit(v)
        end
    end
end

dfs_visit(u)
begin
    color[u] = Gray
    d[u] = ++counter
    for each neighbor v of u do
        if color[v] = White then
            pred[v] = u
            dfs_visit(v)
        end
    end
    color[u] = black
    f[u] = ++counter
end
```

An easier method of supplying a depth first search algorithm on a graph object is to wrap the operation in a class object to hide the implementation details. That can be seen in the `Algo::DepthFirstSearch` class that takes in a graph

object and returns the predecessor list in a search function.

The complexity of the DEPTH-FIRST SEARCH algorithm is show in Table 1.

Best	Average	Worst
$O(V+E)$	$O(V+E)$	$O(V+E)$

Table 1: DEPTH-FIRST SEARCH complexity

`TaskManager` helper class

The `TaskManager` class provides a simple task management to allow tasks to use an underlying thread pool within the class implementation to run algorithms in a parallel manner.

The sequence diagram in Figure 1 shows the interaction between the task build and the thread pool implementation.

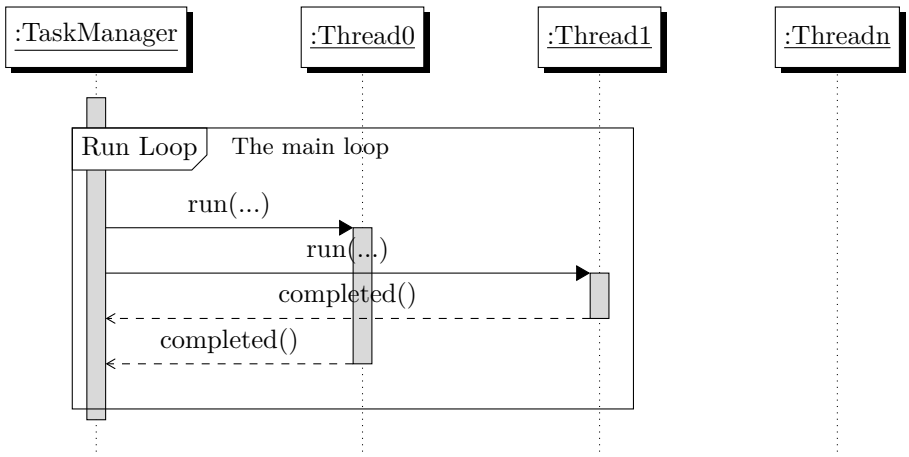


Figure 3: TaskManager `run` sequence diagram

The `TaskManager` class has a `run(...)` function that takes in both a task as a `std::function<void()>` and a completed callback that is used to examine a `TaskManager::Result` enumeration and execute code as a result. The implementation is,

```
19 void TaskManager::run(
20     std::string_view label,
21     std::function<Result()>&& task,
22     std::function<void(Result&)> completed)
23 {
```

```
    auto thread_entry_point = [](
        std::function<Result()>&& func,
        std::function<void(Result&)> completed)
    {
        Result result = func();
        completed(result);
    };
    std::jthread worker(thread_entry_point, std::move(task), completed);
    worker.detach();
}

// This is a comment that appears in the documentation only

void TaskManager::run(
```

This is regular body text description, the `std::string_view` is used as a label as the underlying string is in static storage as a literal. We should use `std::string_view` over `const std::string&` where we can.

```
    std::string_view label,
```