<https://github.com/TT00FE39-3001/lecture7>

<https://github.com/seppotk/Datastructures_and_algorithms.git>

courses:

<https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/>

<https://www.coursera.org/specializations/algorithms>

<https://www.coursera.org/learn/algorithms-part1>

<https://www.coursera.org/learn/algorithms-part2>

<https://www.coursera.org/learn/analysis-of-algorithms>

<https://www.coursera.org/specializations/data-structures-algorithms>

<https://www.youtube.com/@WilliamFiset-videos/playlists>

katso tätä

REVIEW

**# Review**

**### Algorithms: Techniques/Classification**

- Brute Force

  - Linear search

  - Bubble sort

  - Selection sort

- Decrease and conquer

  - Binary Search

  - Insertion sort

- Divide-and-Conquer

  - Quick Sort

  - Merge Sort

- Dynamic Programming

  - Bottom Up: Tabulation

  - Top Down: Memoization

    - 0/1 Knapsack & Staircase Problems

- **\*\*Greedy Method\*\***

**### Data Structures & Abstract data types (ADT)**

- Data Structures (Physical)

  - Arrays

  - Linked Lists

- ADT (Logical)

  - Linear

    - Queues

    - Stacks

    - Hash Tables

  - Non Linear

    - Binary Tree

    - Binary Search Tree

    - Heaps

    - **\*\*Graphs\*\***

**### Analysis of Algorithm Efficiency**

- Big O Complexity

- Average case vs worst case

- Space vs Time

**### Misc**

- FIFO vs LIFO

- Recursion vs Iteration

  - The Top-Down Thought Process

- Logarithms vs Exponential

README

**# Outline**

**## Topics**

- [Review](./review.md)

- Graphs

- Greedy Method

**## This Week in Points**

- Group Activities (Max 9 points)

- Homework (Max 9 points)

- Peer reviews (Max 7 points)

**## Part 1: Graphs**

- Graph Terminology

- Graph Representations

- Graph Traversals: Breadth-First vs Depth-First

- [Activity 1](./activity1)

**## Part 2: Greedy Approach**

- Characteristics of Greedy approach

- Advantages of the Greedy Approach

- Greedy approach vs Dynamic programming

- [Activity 2](./activity2)

**## Part 3**

- Minimal Cost Spanning Trees

- Dijkstra’s Shortest Path Algorithm

- [Activity 3](./activity3)

HOMEWORK:

**# Homework**

**## Task 1/2: Reading**

- [What is Greedy Algorithm: Example, Applications, Limitations and More]([**https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm**](https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm)**)**

**- [Breadth First Search (BFS) C++ Program to Traverse a Graph Or Tree](**[**https://www.softwaretestinghelp.com/cpp-bfs-program-to-traverse-graph/**](https://www.softwaretestinghelp.com/cpp-bfs-program-to-traverse-graph/)**)**

**- [Depth First Search (DFS) C++ Program To Traverse A Graph Or Tree](**[**https://www.softwaretestinghelp.com/cpp-dfs-program-to-traverse-graph/**](https://www.softwaretestinghelp.com/cpp-dfs-program-to-traverse-graph/)**)**

**## Task 2/2: Pre-Lecture**

**- [Standard Template Library (STL): A Brief Introduction](**[**https://www.softwaretestinghelp.com/standard-template-library-stl/**](https://www.softwaretestinghelp.com/standard-template-library-stl/)**)**

**ANSWERS:**

- [What is Greedy Algorithm: Example, Applications, Limitations and More]([**https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm**](https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm)**)**

# **What is Greedy Algorithm: Example, Applications, Limitations and More**

## Table of Contents

[What Is Greedy Algorithm?](https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm#what_is_greedy_algorithm)

[Steps for Creating a Greedy Algorithm](https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm#steps_for_creating_a_greedy_algorithm)

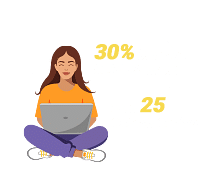
[Example of Greedy Algorithm](https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm#example_of_greedy_algorithm)

[Limitations of Greedy Algorithm](https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm#limitations_of_greedy_algorithm)

[Applications of Greedy Algorithm](https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm#applications_of_greedy_algorithm)

View More

Do you know, during Huffman data compression, greedy algorithm is utilized to build a Huffman tree that compresses a given image, spreadsheet, or video into a lossless compressed file? Also, this greedy [coding](https://www.simplilearn.com/pgp-full-stack-web-development-certification-training-course?source=GhPreviewCoursepages) paradigm encapsulates algorithmic problems where selecting the most obvious answer for the current subproblem leads to the solution of a more complex problem. So, in this article, we will discover a greedy algorithmic paradigm in detail.



## What Is Greedy Algorithm?

A Greedy algorithm is an approach to solving a problem that selects the most appropriate option based on the current situation. This algorithm ignores the fact that the current best result may not bring about the overall optimal result. Even if the initial decision was incorrect, the algorithm never reverses it.

This simple, intuitive algorithm can be applied to solve any optimization problem which requires the maximum or minimum optimum result. The best thing about this algorithm is that it is easy to understand and implement.

The runtime complexity associated with a greedy solution is pretty reasonable. However, you can implement a greedy solution only if the problem statement follows two properties mentioned below:

* Greedy Choice Property: Choosing the best option at each phase can lead to a global (overall) optimal solution.
* Optimal Substructure: If an optimal solution to the complete problem contains the optimal solutions to the subproblems, the problem has an optimal substructure.

Moving forward, we will learn how to create a greedy solution for a problem that adheres to the principles listed above.

## Steps for Creating a Greedy Algorithm

By following the steps given below, you will be able to formulate a greedy solution for the given problem statement:

* Step 1: In a given problem, find the best substructure or subproblem.
* Step 2: Determine what the solution will include (e.g., largest sum, shortest path).
* Step 3: Create an iterative process for going over all subproblems and creating an optimum solution

**- [Breadth First Search (BFS) C++ Program to Traverse a Graph Or Tree](**[**https://www.softwaretestinghelp.com/cpp-bfs-program-to-traverse-graph/**](https://www.softwaretestinghelp.com/cpp-bfs-program-to-traverse-graph/)**)**

**Breadth First Search (BFS) C++ Program To Traverse A Graph Or Tree**

**This Tutorial Covers Breadth First Search in C++ in Which The Graph or Tree is Traversed Breadthwise. You will Also Learn BFS Algorithm & Implementation:**

This explicit C++ tutorial will give you a detailed explanation of traversal techniques that can be performed on a tree or graph.

Traversal is the technique using which we visit each and every node of the graph or a tree. **There are two standard methods of traversals.**

* Breadth-first search(BFS)
* Depth-first search(DFS)

## Breadth First Search (BFS) Technique In C++

In this tutorial, we will discuss in detail the breadth-first search technique.

In the breadth-first traversal technique, the graph or tree is traversed breadth-wise. This technique uses the queue data structure to store the vertices or nodes and also to determine which vertex/node should be taken up next.

Breadth-first algorithm starts with the root node and then traverses all the adjacent nodes. Then, it selects the nearest node and explores all the other unvisited nodes. This process is repeated until all the nodes in the graph are explored.

### Breadth-First Search Algorithm

Given below is the algorithm for BFS technique.

Consider G as a graph which we are going to traverse using the BFS algorithm.

Let S be the root/starting node of the graph.

* **Step 1:** Start with node S and enqueue it to the queue.
* **Step 2:** Repeat the following steps for all the nodes in the graph.
* **Step 3:** Dequeue S and process it.
* **Step 4:** Enqueue all the adjacent nodes of S and process them.
* [END OF LOOP]
* **Step 6:** EXIT

#### **Pseudocode**

**The pseudo-code for the BFS technique is given below.**

Procedure BFS (G, s)

G is the graph and s is the source node

begin

let q be queue to store nodes

q.enqueue(s) //insert source node in the queue

mark s as visited.

while (q is not empty)

//remove the element from the queue whose adjacent nodes are to be processed

n = q.dequeue( )

//processing all the adjacent nodes of n

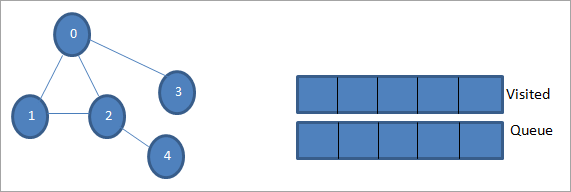
for all neighbors m of n in Graph G if w is not visited

q.enqueue (m) //Stores m in Q to in turn visit its adjacent nodes

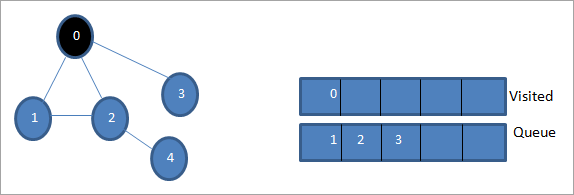
mark m as visited.

end

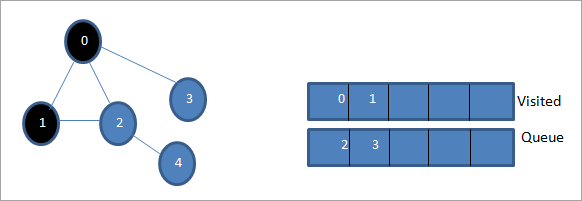
### Traversals With Illustrations

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-1.png)

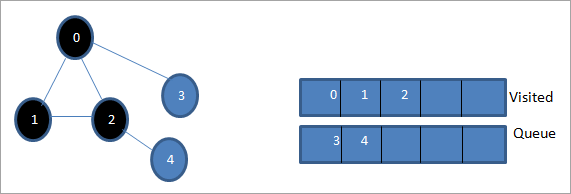
Let 0 be the starting node or source node. First, we enqueue it in the visited queue and all its adjacent nodes in the queue.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-2.png)

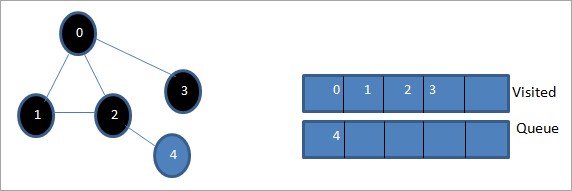
Next, we take one of the adjacent nodes to process i.e. 1. We mark it as visited by removing it from the queue and put its adjacent nodes in the queue (2 and 3 already in queue). As 0 is already visited, we ignore it.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-3.png)

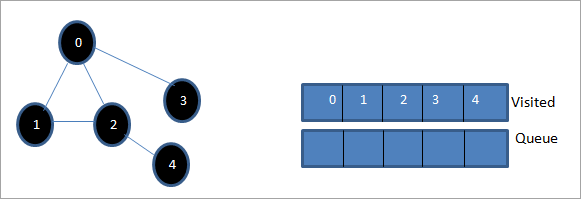
Next, we dequeue node 2 and mark it as visited. Then, its adjacent node 4 is added to the queue.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-4.png)

Next, we dequeue 3 from the queue and mark it as visited. Node 3 has only one adjacent node i.e. 0 which is already visited. Hence, we ignore it.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-5.png)

At this stage, only node 4 is present in the queue. Its adjacent node 2 is already visited, hence we ignore it. Now we mark 4 as visited.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-6.png)

Next, the sequence present in the visited list is the breadth-first traversal of the given graph.

If we observe the given graph and the traversal sequence, we can notice that for the BFS algorithm, we indeed traverse the graph breadth-wise and then go to the next level.

### BFS Implementation

#include <iostream>

#include <list>

using namespace std;

// a directed graph class

class DiGraph

{

    int V;    // No. of vertices

    // Pointer to an array containing adjacency lists

    list<int> \*adjList;

public:

    DiGraph(int V);  // Constructor

    // add an edge from vertex v to w

    void addEdge(int v, int w);

    // BFS traversal sequence starting with s -&gt;starting node

    void BFS(int s);

};

DiGraph::DiGraph(int V)

{

    this->V = V;

    adjList = new list <int>[V];

}

 void DiGraph::addEdge(int v, int w)

{

    adjList[v].push\_back(w); // Add w to v’s list.

}

void DiGraph::BFS(int s)

{

    // initially none of the vertices is visited

    bool \*visited = new bool[V];

    for(int i = 0; i < V; i++)

        visited[i] = false;

    // queue to hold BFS traversal sequence

    list<int> queue;

    // Mark the current node as visited and enqueue it

    visited[s] = true;

    queue.push\_back(s);

    // iterator 'i' to get all adjacent vertices

    list<int>::iterator i;

    while(!queue.empty())

    {

        // dequeue the vertex

        s = queue.front();

        cout << s << " ";

        queue.pop\_front();

        // get all adjacent vertices of popped vertex and process each if not already visited

        for (i = adjList[s].begin(); i != adjList[s].end(); ++i)

        {

            if (!visited[\*i])

            {

                visited[\*i] = true;

                queue.push\_back(\*i);

            }

        }

    }

}

// main program

int main()

{

    // create a graph

    DiGraph dg(5);

    dg.addEdge(0, 1);

    dg.addEdge(0, 2);

    dg.addEdge(0, 3);

    dg.addEdge(1, 2);

    dg.addEdge(2, 4);

    dg.addEdge(3, 3);

    dg.addEdge(4, 4);

    cout << "Breadth First Traversal for given graph (with 0 as starting node): "<< endl;

    dg.BFS(0);

    return 0;

}

Breadth First Traversal for given graph (with 0 as starting node):

0 1 2 3 4

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs>.\BFS

**Output:**

Breadth-First Traversal for the given graph (with 0 as starting node):

0 1 2 3 4

We have implemented the BFS in the above program. Note that the graph is in the form of an adjacency list and then we use an iterator to iterate through the list and perform BFS.

We have used the same graph that we used for illustration purposes as an input to the program to compare the traversal sequence.

### Runtime Analysis

If V is the number of vertices and E is the number of edges of a graph, then the time complexity for BFS can be expressed as **O (|V|+|E|)**. Having said this, it also depends on the data structure that we use to represent the graph.

If we use the adjacency list (like in our implementation), then the time complexity is **O (|V|+|E|).**

If we use the adjacency matrix, then the time complexity is **O (V^2)**.

Apart from the data structures used, there is also a factor of whether the graph is densely populated or sparsely populated.

When the number of vertices exceeds the number of edges, then the graph is said to be sparsely connected as there will be many disconnected vertices. In this case, the time complexity of the graph will be O (V).

On the other hand, sometimes the graph may have a higher number of edges than the number of vertices. In such a case, the graph is said to be densely populated. The time complexity of such a graph is O (E).

To conclude, what the expression O (|V|+|E|) means is depending on whether the graph is densely or sparsely populated, the dominating factor i.e. edges or vertices will determine the time complexity of the graph accordingly.

## Applications Of BFS Traversal

* **Garbage Collection:**The garbage collection technique, “Cheney’s algorithm” uses breadth-first traversal for copying garbage collection.
* **Broadcasting In Networks:**A packet travels from one node to another using the BFS technique in the broadcasting network to reach all nodes.
* **GPS Navigation:**We can use BFS in GPS navigation to find all the adjacent or neighboring location nodes.
* **Social Networking Websites:**Given a person ‘P’, we can find all the people within a distance, ‘d’ from p using BFS till the d levels.
* **Peer To Peer Networks:**Again BFS can be used in peer to peer networks to find all the adjacent nodes.
* **Shortest Path And Minimum Spanning Tree In The Un-weighted Graph:**BFS technique is used to find the shortest path i.e. the path with the least number of edges in the un-weighted graph. Similarly, we can also find a minimum spanning tree using BFS in the un-weighted graph.

## Conclusion

The breadth-first search technique is a method that is used to traverse all the nodes of a graph or a tree in a breadth-wise manner.

This technique is mostly used to find the shortest path between the nodes of a graph or in applications that require us to visit every adjacent node like in networks.

### Recommended Reading

* [Binary Search Tree C++: BST Implementation And Operations With Examples](https://www.softwaretestinghelp.com/binary-search-tree-in-cpp/)
* [B Tree And B+ Tree Data Structure In C++](https://www.softwaretestinghelp.com/b-tree-data-structure-cpp/)
* [Graph Implementation In C++ Using Adjacency List](https://www.softwaretestinghelp.com/graph-implementation-cpp/)
* [Binary Tree Data Structure In C++](https://www.softwaretestinghelp.com/binary-tree-in-cpp/)
* [12 Best Line Graph Maker Tools For Creating Stunning Line Graphs [2023 RANKINGS]](https://www.softwaretestinghelp.com/line-graph-maker/)
* [AVL Tree And Heap Data Structure In C++](https://www.softwaretestinghelp.com/avl-trees-and-heap-data-structure-in-cpp/)
* [Trees In C++: Basic Terminology, Traversal Techniques & C++ Tree Types](https://www.softwaretestinghelp.com/trees-in-cpp/)
* [Cause and Effect Graph - Dynamic Test Case Writing Technique For Maximum Coverage with Fewer Test Cases](https://www.softwaretestinghelp.com/cause-and-effect-graph-test-case-writing-technique/)

- [Depth First Search (DFS) C++ Program To Traverse A Graph Or Tree](<https://www.softwaretestinghelp.com/cpp-dfs-program-to-traverse-graph/>)

# Depth First Search (DFS) C++ Program To Traverse A Graph Or Tree

February 7, 2023

**This Tutorial Covers Depth First Search (DFS) in C++ in Which A Graph or Tree is Traversed Depthwise. You will Also Learn DFS Algorithm & Implementation:**

Depth-first search (DFS) is yet another technique used to traverse a tree or a graph.

DFS starts with a root node or a start node and then explores the adjacent nodes of the current node by going deeper into the graph or a tree. This means that in DFS the nodes are explored depth-wise until a node with no children is encountered.

Once the leaf node is reached, DFS backtracks and starts exploring some more nodes in a similar fashion.

## Depth First Search (DFS) In C++

Unlike BFS in which we explore the nodes breadthwise, in DFS we explore the nodes depth-wise. In DFS we use a stack data structure for storing the nodes being explored. The edges that lead us to unexplored nodes are called ‘discovery edges’ while the edges leading to already visited nodes are called ‘block edges’.

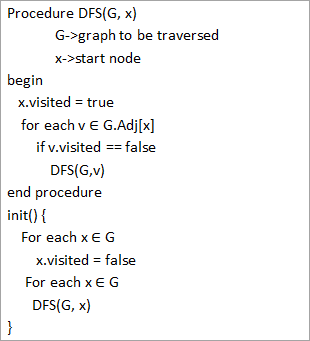
Next, we will see the algorithm and pseudo-code for the DFS technique.

### DFS Algorithm

* **Step 1:** Insert the root node or starting node of a tree or a graph in the stack.
* **Step 2:** Pop the top item from the stack and add it to the visited list.
* **Step 3:** Find all the adjacent nodes of the node marked visited and add the ones that are not yet visited, to the stack.
* **Step 4**: Repeat steps 2 and 3 until the stack is empty.

**Pseudocode**

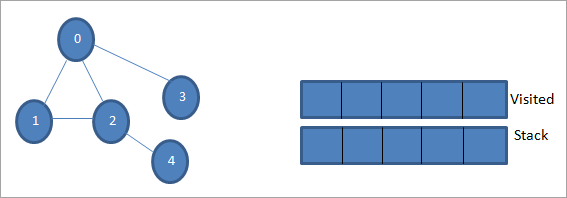
**The pseudo-code for DFS is given below.**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/09/The-pseudo-code-for-DFS.png)

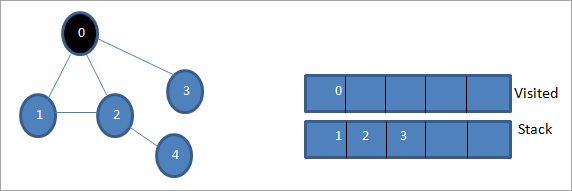
From the above pseudo-code, we notice that the DFS algorithm is called recursively on each vertex to ensure that all the vertices are visited.

### Traversals With Illustrations

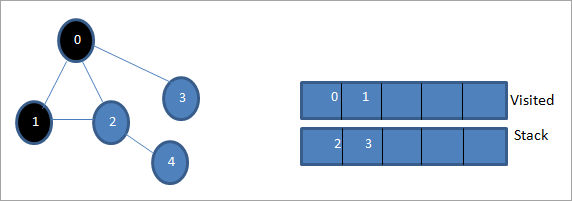
Let us now illustrate the DFS traversal of a graph. For clarity purposes, we will use the same graph that we used in the BFS illustration.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-1-1.png)

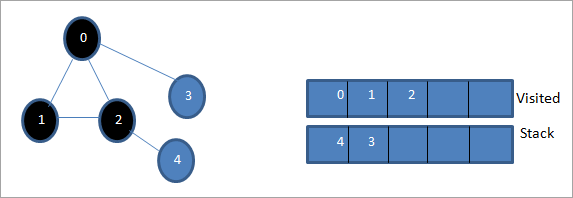
Let 0 be the starting node or source node. First, we mark it as visited and add it to the visited list. Then we push all its adjacent nodes in the stack.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-2-1.png)

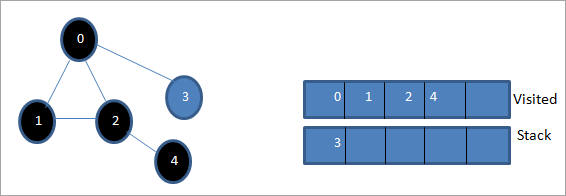
Next, we take one of the adjacent nodes to process i.e. the top of the stack which is 1. We mark it as visited by adding it to the visited list. Now look for the adjacent nodes of 1. As 0 is already in the visited list, we ignore it and we visit 2 which is the top of the stack.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-3-1.png)

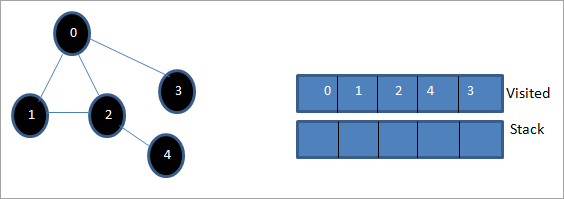
Next, we mark node 2 as visited. Its adjacent node 4 is added to the stack.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-4-1.png)

Next, we mark 4 which is the top of the stack as visited. Node 4 has only node 2 as its adjacent which is already visited, hence we ignore it.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-5-1.png)

At this stage, only node 3 is present in the stack. Its adjacent node 0 is already visited, hence we ignore it. Now we mark 3 as visited.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Illustrations-with-traversals-step-6-1.png)

Now the stack is empty and the visited list shows the sequence of the depth-first traversal of the given graph.

If we observe the given graph and the traversal sequence, we notice that for the DFS algorithm, we indeed traverse the graph depth-wise and then backtrack it again to explore new nodes.

### Depth-First Search Implementation

#include <iostream>

#include <list>

using namespace std;

//graph class for DFS travesal

class DFSGraph

{

int V;    // No. of vertices

list<int> \*adjList;    // adjacency list

void DFS\_util(int v, bool visited[]);  // A function used by DFS

public:

    // class Constructor

DFSGraph(int V)

    {

 this->V = V;

 adjList = new list<int>[V];

    }

    // function to add an edge to graph

void addEdge(int v, int w){

adjList[v].push\_back(w); // Add w to v’s list.

    }

void DFS();    // DFS traversal function

};

void DFSGraph::DFS\_util(int v, bool visited[])

{

    // current node v is visited

visited[v] = true;

cout << v << " ";

    // recursively process all the adjacent vertices of the node

list<int>::iterator i;

for(i = adjList[v].begin(); i != adjList[v].end(); ++i)

if(!visited[\*i])

DFS\_util(\*i, visited);

}

// DFS traversal

void DFSGraph::DFS()

{

    // initially none of the vertices are visited

bool \*visited = new bool[V];

for (int i = 0; i < V; i++)

visited[i] = false;

    // explore the vertices one by one by recursively calling  DFS\_util

for (int i = 0; i < V; i++)

if (visited[i] == false)

DFS\_util(i, visited);

}

int main()

{

    // Create a graph

DFSGraph gdfs(5);

gdfs.addEdge(0, 1);

gdfs.addEdge(0, 2);

gdfs.addEdge(0, 3);

gdfs.addEdge(1, 2);

gdfs.addEdge(2, 4);

gdfs.addEdge(3, 3);

gdfs.addEdge(4, 4);

cout << "Depth-first traversal for the given graph:"<<endl;

gdfs.DFS();

return 0;

}

**PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\DFS**

**Depth-first traversal for the given graph:**

**0 1 2 4 3**

We have once again used the graph in the program that we used for illustration purposes. We see that the DFS algorithm (separated into two functions) is called recursively on each vertex in the graph in order to ensure that all the vertices are visited.

### Runtime Analysis

The time complexity of DFS is the same as BFS i.e. **O (|V|+|E|)** where V is the number of vertices and E is the number of edges in a given graph.

Similar to BFS, depending on whether the graph is scarcely populated or densely populated, the dominant factor will be vertices or edges respectively in the calculation of time complexity.

### Iterative DFS

The implementation shown above for the DFS technique is recursive in nature and it uses a function call stack. We have another variation for implementing DFS i.e. “**Iterative depth-first search**”. In this, we use the explicit stack to hold the visited vertices.

We have shown the implementation for iterative DFS below. Note that the implementation is the same as BFS except the factor that we use the stack data structure instead of a queue.

#include<bits/stdc++.h>

using namespace std;

// graph class

class Graph

{

int V;    // No. of vertices

list<int> \*adjList;    // adjacency lists

public:

Graph(int V)  //graph Constructor

    {

this->V = V;

adjList = new list<int>[V];

    }

void addEdge(int v, int w) // add an edge to graph

    {

adjList[v].push\_back(w); // Add w to v’s list.

    }

void DFS();  // DFS traversal

    // utility function called by DFS

void DFSUtil(int s, vector<bool> &visited);

};

//traverses all not visited vertices reachable from start node s

void Graph::DFSUtil(int s, vector<bool> &visited)

{

    // stack for DFS

stack<int> dfsstack;

   // current source node inside stack

dfsstack.push(s);

while (!dfsstack.empty())

    {

        // Pop a vertex

        s = dfsstack.top();

       dfsstack.pop();

        // display the item or node only if its not visited

 if (!visited[s])

        {

cout << s << " ";

visited[s] = true;

        }

        // explore all adjacent vertices of popped vertex.

        //Push the vertex to the stack if still not visited

for (auto i = adjList[s].begin(); i != adjList[s].end(); ++i)

if (!visited[\*i])

dfsstack.push(\*i);

    }

}

// DFS

void Graph::DFS()

{

    // initially all vertices are not visited

vector<bool> visited(V, false);

for (int i = 0; i < V; i++)

if (!visited[i])

DFSUtil(i, visited);

}

//main program

int main()

{

    Graph gidfs(5);  //create graph

gidfs.addEdge(0, 1);

gidfs.addEdge(0, 2);

gidfs.addEdge(0, 3);

gidfs.addEdge(1, 2);

gidfs.addEdge(2, 4);

gidfs.addEdge(3, 3);

gidfs.addEdge(4, 4);

cout << "Output of Iterative Depth-first traversal:\n";

gidfs.DFS();

return 0;

}

**PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\DFS\_iterative**

**Output of Iterative Depth-first traversal:**

**0 3 2 4 1**

We use the same graph that we used in our recursive implementation. The difference in output is because we use the stack in the iterative implementation. As the stacks follow LIFO order, we get a different sequence of DFS. To get the same sequence, we might want to insert the vertices in the reverse order.

## BFS Vs DFS

So far we have discussed both the traversal techniques for graphs i.e. BFS and DFS.

**Now let us look into the differences between the two.**

| **BFS** | **DFS** |
| --- | --- |
| Stands for “Breadth-first search” | Stands for “Depth-first search” |
| The nodes are explored breadth wise level by level. | The nodes are explored depth-wise until there are only leaf nodes and then backtracked to explore other unvisited nodes. |
| BFS is performed with the help of queue data structure. | DFS is performed with the help of stack data structure. |
| Slower in performance. | Faster than BFS. |
| Useful in finding the shortest path between two nodes. | Used mostly to detect cycles in graphs. |

## Applications Of DFS

* **Detecting Cycles In The Graph:**If we find a back edge while performing DFS in a graph then we can conclude that the graph has a cycle. Hence DFS is used to detect the cycles in a graph.
* **Pathfinding:**Given two vertices x and y, we can find the path between x and y using DFS. We start with vertex x and then push all the vertices on the way to the stack till we encounter y. The contents of the stack give the path between x and y.
* **Minimum Spanning Tree And Shortest Path:**DFS traversal of the un-weighted graph gives us a minimum spanning tree and shortest path between nodes.
* **Topological Sorting:**We use topological sorting when we need to schedule the jobs from the given dependencies among jobs. In the computer science field, we use it mostly for resolving symbol dependencies in linkers, data serialization, instruction scheduling, etc. DFS is widely used in Topological sorting.

## Conclusion

In the last couple of tutorials, we explored more about the two traversal techniques for graphs i.e. BFS and DFS. We have seen the differences as well as the applications of both the techniques. BFS and DFS basically achieve the same outcome of visiting all nodes of a graph but they differ in the order of the output and the way in which it is done.

We have also seen the implementation of both techniques. While BFS uses a queue, DFS makes use of stacks to implement the technique.  With this, we conclude the tutorial on traversal techniques for graphs. We can also use BFS and DFS on trees.

***We will learn more about spanning trees and a couple of algorithms to find the shortest path between the nodes of a graph in our upcoming tutorial.***

**=>**[**See Here To Explore The Full C++ Tutorials list.**](https://www.softwaretestinghelp.com/cpp-tutorials/)

### Recommended Reading

* [Breadth First Search (BFS) C++ Program to Traverse a Graph Or Tree](https://www.softwaretestinghelp.com/cpp-bfs-program-to-traverse-graph/)
* [Binary Search Tree C++: BST Implementation And Operations With Examples](https://www.softwaretestinghelp.com/binary-search-tree-in-cpp/)
* [B Tree And B+ Tree Data Structure In C++](https://www.softwaretestinghelp.com/b-tree-data-structure-cpp/)
* [In-Depth Eclipse Tutorials For Beginners](https://www.softwaretestinghelp.com/eclipse/)
* [Binary Tree Data Structure In C++](https://www.softwaretestinghelp.com/binary-tree-in-cpp/)
* [Graph Implementation In C++ Using Adjacency List](https://www.softwaretestinghelp.com/graph-implementation-cpp/)
* [AVL Tree And Heap Data Structure In C++](https://www.softwaretestinghelp.com/avl-trees-and-heap-data-structure-in-cpp/)
* [12 Best Line Graph Maker Tools For Creating Stunning Line Graphs [2023 RANKINGS]](https://www.softwaretestinghelp.com/line-graph-maker/)

**## Task 2/2: Pre-Lecture**

- [Standard Template Library (STL): A Brief Introduction](<https://www.softwaretestinghelp.com/standard-template-library-stl/>)

# Standard Template Library (STL): A Brief Introduction

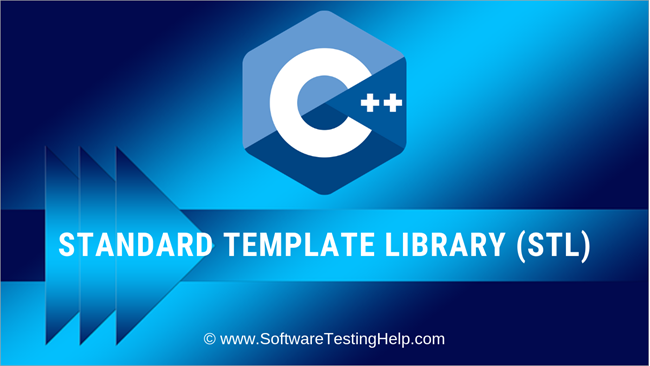
February 18, 2023

**A Complete Overview Of Standard Template Library (STL):**

Standard Template Library (STL) of C++ is a collection of template classes that provide data structures such as arrays, vectors, queue, etc. STL is a library consisting of containers, algorithms, and iterators.

As STL consists of a collection of template classes, it’s a generalized library that is independent of data types.

**=>**[**Read Through The Extensive C++ Training Tutorial Series Here.**](https://www.softwaretestinghelp.com/cpp-tutorials/)

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/Standard-Template-Library-STL.png)

01:0709:31EXPLORE MOREGraphical user interface

Description automatically generatedIntroduction to Java Virtual Machine (Tutorial #13)09:31Graphical user interface, website

Description automatically generatedAnalyze the Importance of Mocking in Spock...14:58Text

Description automatically generatedHow to Download and Install Selenium on your system25:52Graphical user interface, website

Description automatically generatedPython Conditional Statements (Tutorial # 5, Part-1)13:57Graphical user interface

Description automatically generatedGatling: Create Simulation24:06Graphical user interface, application

Description automatically generatedAn In-Depth Tutorial on Spock Selenium Integration06:09Graphical user interface, text, application, website

Description automatically generatedAll SQL Joins are Clearly Explained with Examples19:38Graphical user interface, website

Description automatically generatedWhat is Performance Testing? | Explained the Load, Volume,...26:07Graphical user interface

Description automatically generatedPython OOPs Concepts (Tutorial # 10, Part-1)15:44Text

Description automatically generatedJest Configuration12:54

**What You Will Learn:**[[hide](https://www.softwaretestinghelp.com/standard-template-library-stl/)]

* [Components Of STL](https://www.softwaretestinghelp.com/standard-template-library-stl/#Components_Of_STL)
  + [#1) Containers](https://www.softwaretestinghelp.com/standard-template-library-stl/#1_Containers)
  + [#2) Algorithms](https://www.softwaretestinghelp.com/standard-template-library-stl/#2_Algorithms)
  + [#3) Iterators](https://www.softwaretestinghelp.com/standard-template-library-stl/#3_Iterators)
* [Containers](https://www.softwaretestinghelp.com/standard-template-library-stl/#Containers)
  + [#1) Sequential Containers](https://www.softwaretestinghelp.com/standard-template-library-stl/#1_Sequential_Containers)
  + [#2) Associative Containers](https://www.softwaretestinghelp.com/standard-template-library-stl/#2_Associative_Containers)
  + [#3) Container Adopters](https://www.softwaretestinghelp.com/standard-template-library-stl/#3_Container_Adopters)
* [Iterators](https://www.softwaretestinghelp.com/standard-template-library-stl/#Iterators)
* [Algorithms](https://www.softwaretestinghelp.com/standard-template-library-stl/#Algorithms)
* [Conclusion](https://www.softwaretestinghelp.com/standard-template-library-stl/#Conclusion)
* [Recommended Reading](https://www.softwaretestinghelp.com/standard-template-library-stl/#Recommended_Reading)

### Components Of STL

STL mainly consists of the following components which are mentioned below:

#### **#1) Containers**

A container is a collection of objects of a particular type of data structure. In STL, we have various types of container classes like Array, vector, queue, deque, list, map, set, etc. These containers are generic in nature and are implemented as class templates.

Containers are dynamic in nature and can be used to hold various types of objects.

#### **#2) Algorithms**

Algorithms are the methods or functions that act on containers. By using algorithms provided by STL, we can have methods to search, sort, modify, transform or initialize the contents of container class objects.

Algorithms provided by STL have built-in functions that can directly operate on complex data structure instead of having to write the algorithms ourselves.

**For Example,** reverse() function in STL can be used to reverse the linked list.

#### **#3) Iterators**

Iterators are the very important and distinguishing feature of STL. Iterators are the constructs that are used to traverse through the container objects. Similar to indexes that we use to step through the arrays, Iterators act on container class objects and can be used to step through the data.

### Containers

Containers store objects and data. They are basically template-based generic classes.

**Containers in STL are divided into the following types:**

#### **#1) Sequential Containers**

Containers the can be accessed in a sequential or linear manner are said to be sequential containers.

Arrays, Vectors, Lists, Deques are the STL containers that store data linearly and can be accessed in a sequential manner.

#### **#2) Associative Containers**

Associative containers are containers that implement sorted data structures. These containers are fast to search. Some of the **Examples** of associative containers are Map, Set, MultiMap, Multiset, etc. These containers are usually implemented in a key/value pair fashion.

#### **#3) Container Adopters**

Container adopters are sequential containers, however, they are implemented by providing a different interface. Thus containers like a queue, deque, stack, and priority-queue are all classified as container adopters.

### Iterators

Iterators are constructs that we use to traverse or step through containers in STL. Iterators are very important in STL as they act as a bridge between algorithms and containers. Iterators always point to containers and in fact algorithms actually, operate on iterators and never directly on containers.

**Iterators are of following types:**

* **Input Iterators:** Simplest and is used mostly in single-pass algorithms.
* **Output Iterators:** Same as input iterators but not used for traversing.
* **Bidirectional Iterators:** These iterators can move in both directions.
* **Forward Iterators:** Can be used only in the forward direction, one step at a time.
* **Random Access Iterators:** Same as pointers. Can be used to access any element randomly.

### Algorithms

Algorithms are a set of functions or methods provided by STL that act on containers. These are built-in functions and can be used directly with the STL containers and iterators instead of writing our own algorithms.

**STL supports the following types of algorithms:**

* Searching algorithms
* Sorting algorithms
* Modifying or manipulating algorithms
* Non-modifying algorithms
* Numeric algorithms
* Min/Max algorithms

As each of the algorithm types suggests, these algorithms can be used to achieve different functionality in STL containers like searching, sorting, transforming the data in the containers, finding min/max value, etc.

### Conclusion

This is the brief introduction of Standard Template Library. In our upcoming tutorials, we will learn more about each of the containers, algorithms, and iterators.

**=>**[**Check Complete C++ FREE Training Series Here.**](https://www.softwaretestinghelp.com/cpp-tutorials/)

### Recommended Reading

* [Blazemeter Plugin And Jmeter Template](https://www.softwaretestinghelp.com/blazemeter-plugin-and-jmeter-template/)
* [Priority Queue In STL](https://www.softwaretestinghelp.com/priority-queue-in-stl/)
* [Arrays In STL](https://www.softwaretestinghelp.com/arrays-in-stl/)
* [Library Functions In C++](https://www.softwaretestinghelp.com/library-functions-in-cpp/)
* [Iterators In STL](https://www.softwaretestinghelp.com/iterators-in-stl/)
* [JMeter Video 1: Introduction, JMeter Download and Install](https://www.softwaretestinghelp.com/apache-jmeter-download-install/)
* [Stacks And Queues In STL](https://www.softwaretestinghelp.com/stacks-and-queues-in-stl/)
* [Sample Software Test Plan Template with Format and Contents](https://www.softwaretestinghelp.com/test-plan-template/)

**ACTIVITY 1**

**# Activities**

**## Task 1:**

**- Answer at least 5 questions from the following link. Make sure you write the question as well as the answer.**

[**https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Graph/GraphIntroSumm.html**](https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Graph/GraphIntroSumm.html)

**> You can refer to [link #2](#links) below for more info.**

**## Task 2**

**- Discuss how depth-first search works by experimenting with the following link. Try both directed and undirected graphs and write a short summary.**

[**https://opendsa-server.cs.vt.edu/OpenDSA/AV/Graph/DFSAV.html**](https://opendsa-server.cs.vt.edu/OpenDSA/AV/Graph/DFSAV.html)

**> You can refer to [link #3](#links) below for more info.**

**## Task 3**

**- Discuss how breadth-first search works by experimenting with the following link. Try both directed and undirected graphs and write a short summary.**

[**https://opendsa-server.cs.vt.edu/OpenDSA/AV/Graph/BFSAV.html**](https://opendsa-server.cs.vt.edu/OpenDSA/AV/Graph/BFSAV.html)

**> You can refer to [link #4](#links) below for more info.**

**## Task 4:**

**- Reproduce the behavior of the BFS algorithm for the following graph:**

**https://opendsa-server.cs.vt.edu/OpenDSA/AV/Graph/BFSPE.html**

**> You can refer to [link #4](#links) below.**

**## Task 5: Individual (at home)**

**- There are two traditional approaches to representing graphs: The adjacency matrix and the adjacency list. What are the main differences in term of space/time complexity. You can refer to following link:**

**https://www.baeldung.com/cs/adjacency-matrix-list-complexity**

**## Links**

**1. https://cpp.sh/**

**2.** [**https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/GraphIntro.html**](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/GraphIntro.html)

**3.** [**https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/GraphTraversal.html#depth-first-search**](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/GraphTraversal.html#depth-first-search)

**4.** [**https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/GraphTraversal.html#breadth-first-search**](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/GraphTraversal.html#breadth-first-search)

**ANSWERS:**

**## Task 1:**

**- Answer at least 5 questions from the following link. Make sure you write the question as well as the answer.**

[**https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Graph/GraphIntroSumm.html**](https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Graph/GraphIntroSumm.html)

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Diagram

Description automatically generated

The **in** degree of a vertex is the number of edges going **into** the vertex.

Graphical user interface, text, application, email

Description automatically generated

**A free tree is a connected, undirected graph with no cycles**

Graphical user interface, text, application, email

Description automatically generated

**A cycle is simple if all vertices on the cycle are distinct, with the first and last vertices being the same.**

**vertice = node**

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

**## Task 2**

- Discuss how depth-first search works by experimenting with the following link. Try both directed and undirected graphs and write a short summary.

<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Graph/DFSAV.html>

> You can refer to [link #3](#links) below for more info.

Diagram

Description automatically generated

A picture containing chart

Description automatically generated

### **19.3.1.1. Depth-First Search**

Our first method for organized graph traversal is called [**depth-first search**](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/Glossary.html#term-depth-first-search) (DFS). Whenever a vertex v� is visited during the search, DFS will recursively visit all of v� ‘s unvisited neighbors. Equivalently, DFS will add all edges leading out of v� to a stack. The next vertex to be visited is determined by popping the stack and following that edge. The effect is to follow one branch through the graph to its conclusion, then it will back up and follow another branch, and so on. The DFS process can be used to define a [**depth-first search tree**](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/Glossary.html#term-depth-first-search-tree). This tree is composed of the edges that were followed to any new (unvisited) vertex during the traversal, and leaves out the edges that lead to already visited vertices. DFS can be applied to directed or undirected graphs.

DFS processes each edge once in a directed graph. In an undirected graph, DFS processes each edge from both directions. Each vertex must be visited, but only once, so the total cost is Θ(|V|+|E|)Θ(|�|+|�|)

**## Task 3**

- Discuss how breadth-first search works by experimenting with the following link. Try both directed and undirected graphs and write a short summary.

<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Graph/BFSAV.html>

> You can refer to [link #4](#links) below for more info.

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

## A picture containing diagram Description automatically generated

## Chart Description automatically generated with low confidence

## 19.3.2. Breadth-First Search

Our second graph traversal algorithm is known as a [**breadth-first search**](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/Glossary.html#term-breadth-first-search) (BFS). BFS examines all vertices connected to the start vertex before visiting vertices further away. BFS is implemented similarly to DFS, except that a queue replaces the recursion stack. Note that if the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom.

This visualization shows a graph and the result of performing a BFS on it, resulting in a breadth-first search tree.

**## Task 4:**

- Reproduce the behavior of the BFS algorithm for the following graph:

<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Graph/BFSPE.html>

> You can refer to [link #4](#links) below.

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing map

Description automatically generated

A picture containing map

Description automatically generated

A picture containing diagram

Description automatically generated

**## Task 5: Individual (at home)**

- There are two traditional approaches to representing graphs: The adjacency matrix and the adjacency list. What are the main differences in term of space/time complexity. You can refer to following link:

<https://www.baeldung.com/cs/adjacency-matrix-list-complexity>

## 1. Overview

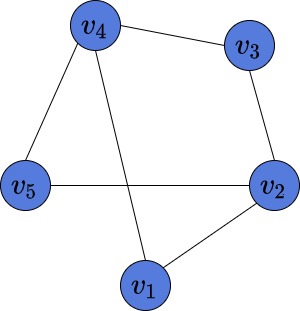
In this tutorial, we’ll learn one of the main aspects of [Graph Theory](https://www.baeldung.com/cs/graph-theory-intro) — graph representation. **The two main methods to store a graph in memory are adjacency matrix and adjacency list representation.** These methods have different time and space complexities.

Thus, to optimize any graph algorithm, we should know which graph representation to choose. The choice depends on the particular graph problem. In this article, we’ll use [Big-O](https://www.baeldung.com/cs/big-o-notation) notation to describe the time and space complexity of methods that represent a graph.

## 2. Graph Representation

It’s important to remember that the graph is a set of vertices  that are connected by edges . An edge is a pair of vertices , where . Each edge has its starting and ending vertices. If graph is undirected, . But, in directed graph the order of starting and ending vertices matters and .

Here is an example of an undirected graph, which we’ll use in further examples:



This graph consists of 5 vertices , which are connected by 6 edges , and .

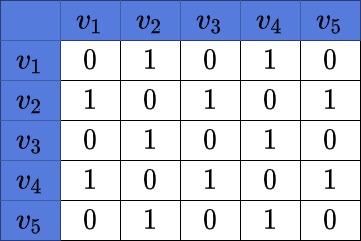
Some graphs might have many vertices, but few edges. These ones are called [sparse](https://www.baeldung.com/cs/graphs-sparse-vs-dense). On the other hand, the ones with many edges are called [dense](https://www.baeldung.com/cs/graphs-sparse-vs-dense). Our graph is neither sparse nor dense. However, in this article, we’ll see that the graph structure is relevant for choosing the way to represent it in memory.

## 3. Adjacency Matrix

The first way to represent a graph in a computer’s memory is to build an adjacency matrix. Assume our graph consists of  vertices numbered from  to .  An adjacency matrix is a binary matrix of size . There are two possible values in each cell of the matrix: 0 and 1. Suppose there exists an edge between vertices  and . It means, that the value in the  row and  column of such matrix is equal to 1. **Importantly, if the graph is undirected then the matrix is**[**symmetric**](https://en.wikipedia.org/wiki/Symmetric_matrix)**.**

### 3.1. Example

Here is an example of an adjacency matrix, corresponding to the above graph:



We may notice the symmetry of the matrix. Also, we can see, there are 6 edges in the matrix. It means, there are 12 cells in its adjacency matrix with a value of 1.

### 3.2. Time and Space Complexity

**Assuming the graph has  vertices, the time complexity to build such a matrix is . The space complexity is also .** Given a graph, to build the adjacency matrix, we need to create a square  matrix and fill its values with 0 and 1. It costs us  space.

To fill every value of the matrix we need to check if there is an edge between every pair  of vertices. The amount of such pairs of  given vertices is . That is why the time complexity of building the matrix is .

### 3.3. Pros and Cons

**The advantage of such representation is that we can check in  time if there exists edge  by simply checking the value at  row and  column of our matrix.**

However, this approach has one big disadvantage. We need  space in the only case — if our graph is [complete](https://www.baeldung.com/cs/graph-theory-intro#6-the-complete-graph) and has all  edges. The matrix will be full of ones except the main diagonal, where all the values will be equal to zero. But, the complete graphs rarely happens in real-life problems. So, if the target graph would contain many vertices and few edges, then representing it with the adjacency matrix is inefficient.

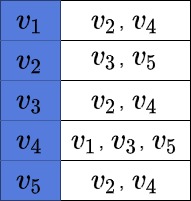
## 4. Adjacency List

The other way to represent a graph in memory is by building the adjacent list. If the graph consists of  vertices, then the list  contains  elements. Each element  is also a list and contains all the vertices, adjacent to the current vertex . **By choosing an adjacency list as a way to store the graph in memory, this may save us space.**

For instance, in the [Depth-First Search](https://www.baeldung.com/java-depth-first-search) algorithm, there is no need to store the adjacency matrix. At each algorithm step, we need to know all the vertices adjacent to the current one. This what the adjacency lists can provide us easily. We may also use the adjacency matrix in this algorithm, but there is no need to do it. Instead, we are saving space by choosing the adjacency list.

### 4.1. Example

This is the adjacency list of the graph above:



We may notice, that this graph representation contains only the information about the edges, which are present in the graph.

### 4.2. Time and Space Complexity

**If  is the number of edges in a graph, then the time complexity of building such a list is . The space complexity is .** But, in the worst case of a complete graph, which contains  edges, the time and space complexities reduce to .

### 4.3. Pros and Cons

As it was mentioned, complete graphs are rarely meet. **Thus, this representation is more efficient if space matters.** Moreover, we may notice, that the amount of edges doesn’t play any role in the space complexity of the adjacency matrix, which is fixed. But, the fewer edges we have in our graph the less space it takes to build an adjacency list.

However, there is a major disadvantage of representing the graph with the adjacency list. The access time to check whether edge  is present is constant in adjacency matrix, but is linear in adjacency list. In a complete graph with  vertices, for every vertex  the element of  would contain  element, as every vertex is connected with every other vertex in such a graph.

**Therefore, the time complexity checking the presence of an edge in the adjacency list is .** Let’s assume that an algorithm often requires checking the presence of an arbitrary edge in a graph. Also, time matters to us. Here, using an adjacency list would be inefficient.

## 5. Removing Edges and Vertices

Let’s now see the worst-case complexity of removing a vertex and an edge.

### 5.1. Removing a Vertex

Let’s suppose we want to remove .

If we use the adjacency matrix, we’ll have to set all the entries in the -th row and the -th column to zero. Doing so will delete all the edges incident to , effectively removing  from the graph. In total, we’ll iterate over  cells, so **the time complexity will be**.

**On the other hand, removing a vertex from an adjacency list will cost more.** To remove all the outgoing edges, we set to NULL the pointer to the ‘s list, . However, to delete all the occurrences of  from other nodes’ lists, we have to iterate over all the other lists. In the worst case, each node will be connected to all the other vertices, so we’ll traverse  list elements. Therefore, **removing a vertex from the list representation of a graph is an  operation.**

### 5.2. Removing an Edge

To remove an edge  from an adjacency matrix , we set  to zero. If the graph is symmetric, we do the same with . **Accessing a cell in the matrix is an  operation, so the complexity is** in the best-case, average-case, and worst-case scenarios.

**If we store the graph as an adjacency list, the complexity of deleting an edge is .** That’s because, in the worst case, we traverse the whole list  to remove  from it. If the graph is symmetric, we do the same with , removing  from it. In total, we iterate over no more than  elements, so the complexity is .

## 6. Conclusion

In this tutorial, we’ve discussed the two main methods of graph representation. We’ve learned about the time and space complexities of both methods. Moreover, we’ve shown the advantages and disadvantages of both methods.

The choice of the graph representation depends on the given graph and given problem. In some problems space matters, however, in others not. These assumptions help to choose the proper variant of graph representation for particular problems.

ACTIVITY 2

**# Activities**

**## Task 1**

- Refer to the following link. Discuss the characteristics of Greedy approach:

<https://www.geeksforgeeks.org/greedy-approach-vs-dynamic-programming/>

**## Task 2**

- Explain how the code in `./src/fkp.cp`p works. Refer to the following link:

<https://www.geeksforgeeks.org/fractional-knapsack-problem/>

**## Task 3**

- Explain how the code in `./src/asp.cpp` works. Refer to the following link:

<https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/>

**## Task 4: Individual (at home)**

- Refer to te following link. Explain the differences between Greedy Algorithm and Dynamic Programming

<https://www.geeksforgeeks.org/greedy-approach-vs-dynamic-programming/>

**## Link(s)**

- https://cpp.sh/

ANSWERS:

**## Task 1**

- Refer to the following link. Discuss the characteristics of Greedy approach:

<https://www.geeksforgeeks.org/greedy-approach-vs-dynamic-programming/>

**Greedy approach:**

*A Greedy algorithm is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to a global solution is the best fit for Greedy.*

**Example:** In Fractional Knapsack Problem the local optimal strategy is to choose the item that has maximum **value vs weight** ratio. This strategy also leads to global optimal solution because we allowed taking fractions of an item.

### **Characteristics of Greedy approach:**

A problem that can be solved using the Greedy approach follows the below-mentioned properties:

* Optimal substructure property.
* Minimization or Maximization of quantity is required.
* Ordered data is available such as data on increasing profit, decreasing cost, etc.
* Non-overlapping subproblems.

### **Below are some major differences between Greedy method and Dynamic programming:**

| **Feature** | **Greedy method** | **Dynamic programming** |
| --- | --- | --- |
| **Feasibility** | In a [greedy Algorithm](https://www.geeksforgeeks.org/greedy-algorithms/), we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution. | In [Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming/) we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution . |
| **Optimality** | In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution. | It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best. |
| **Recursion** | A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage. | A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states. |
| **Memoization** | It is more efficient in terms of memory as it never look back or revise previous choices | It requires Dynamic Programming table for Memoization and it increases it’s memory complexity. |
| **Time        complexity** | Greedy methods are generally faster. For example, [Dijkstra’s shortest path](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/) algorithm takes O(ELogV + VLogV) time. | Dynamic Programming is generally slower. For example, [Bellman Ford algorithm](https://www.geeksforgeeks.org/bellman-ford-algorithm-simple-implementation/) takes O(VE) time. |
| **Fashion** | The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices. | Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions. |
| **Example** | Fractional knapsack . | 0/1 knapsack problem |

# **Fractional Knapsack Problem**

The fractional knapsack problem is also one of the techniques which are used to solve the knapsack problem. In fractional knapsack, the items are broken in order to maximize the profit. The problem in which we break the item is known as a Fractional knapsack problem.

**This problem can be solved with the help of using two techniques:**

* Brute-force approach: The brute-force approach tries all the possible solutions with all the different fractions but it is a time-consuming approach.
* Greedy approach: In Greedy approach, we calculate the ratio of profit/weight, and accordingly, we will select the item. The item with the highest ratio would be selected first.

**There are basically three approaches to solve the problem:**

* The first approach is to select the item based on the maximum profit.
* The second approach is to select the item based on the minimum weight.
* The third approach is to calculate the ratio of profit/weight.

**## Task 2**

- Explain how the code in `./src/fkp.cp`p works. Refer to the following link:

<https://www.geeksforgeeks.org/fractional-knapsack-problem/>

ANSWER:

## Fractional Knapsack Problem using [Greedy algorithm](https://www.geeksforgeeks.org/greedy-algorithms/):

An efficient solution is to use the Greedy approach.

*The basic idea of the greedy approach is to calculate the ratio****value/weight****for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them until we can’t add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.*

Follow the given steps to solve the problem using the above approach:

* Calculate the ratio(value/weight) for each item.
* Sort all the items in decreasing order of the ratio.
* Initialize **res =0**, curr\_cap = given\_cap.
* Do the following for every item “i” in the sorted order:
  + If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result
  + Else add the current item as much as we can and break out of the loop.
* Return **res**.

Below is the implementation of the above approach:

// C++ program to solve fractional Knapsack Problem

#include <bits/stdc++.h>

#include <iostream>

using namespace std;

// Structure for an item which stores weight and

// corresponding value of Item

struct Item

{

    int value, weight;

    // Constructor

    Item(int value, int weight)

    {

        this->value = value;

        this->weight = weight;

    }

};

// Comparison function to sort Item

// according to val/weight ratio

static bool cmp(struct Item a, struct Item b)

{

    double r1 = (double)a.value / (double)a.weight;

    double r2 = (double)b.value / (double)b.weight;

    return r1 > r2;

}

// Main greedy function to solve problem

double fractionalKnapsack(int W, struct Item arr[], int N)

{

    // Sorting Item on basis of ratio

    sort(arr, arr + N, cmp);

    double finalvalue = 0.0;

    // Looping through all items

    for (int i = 0; i < N; i++)

    {

        // If adding Item won't overflow,

        // add it completely

        if (arr[i].weight <= W)

        {

            W -= arr[i].weight;

            finalvalue += arr[i].value;

        }

        // If we can't add current Item,

        // add fractional part of it

        else

        {

            finalvalue += arr[i].value \* ((double)W / (double)arr[i].weight);

            break;

        }

    }

    // Returning final value

    return finalvalue;

}

// Driver code

int main()

{

    int W = 50;

    Item arr[] = {{60, 10}, {100, 20}, {120, 30}};

    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call

    cout << fractionalKnapsack(W, arr, N);

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture7-main\activity2\src> .\fkp

240

**Time Complexity:** O(N \* log N)  
**Auxiliary Space:** O(N)

This article is contributed by Utkarsh Trivedi.  
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Related Articles

**1.**[Difference between 0/1 Knapsack problem and Fractional Knapsack problem](https://www.geeksforgeeks.org/difference-between-0-1-knapsack-problem-and-fractional-knapsack-problem/?ref=rp)

**2.**[C++ Program for the Fractional Knapsack Problem](https://www.geeksforgeeks.org/c-program-for-the-fractional-knapsack-problem/?ref=rp)

**3.**[Unbounded Fractional Knapsack](https://www.geeksforgeeks.org/unbounded-fractional-knapsack/?ref=rp)

**4.**[Fractional Knapsack Queries](https://www.geeksforgeeks.org/fractional-knapsack-queries/?ref=rp)

**5.**[A Space Optimized DP solution for 0-1 Knapsack Problem](https://www.geeksforgeeks.org/space-optimized-dp-solution-0-1-knapsack-problem/?ref=rp)

**6.**[Python Program for 0-1 Knapsack Problem](https://www.geeksforgeeks.org/python-program-for-dynamic-programming-set-10-0-1-knapsack-problem/?ref=rp)

**7.**[Java Program 0-1 Knapsack Problem](https://www.geeksforgeeks.org/java-program-for-dynamic-programming-set-10-0-1-knapsack-problem/?ref=rp)

**8.**[0/1 Knapsack Problem](https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/?ref=rp)

**9.**[Extended Knapsack Problem](https://www.geeksforgeeks.org/extended-knapsack-problem/?ref=rp)

**10.**[0/1 Knapsack Problem to print all possible solutions](https://www.geeksforgeeks.org/0-1-knapsack-problem-to-print-all-possible-solutions/?ref=rp)

**## Task 3**

- Explain how the code in `./src/asp.cpp` works. Refer to the following link:

<https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/>

# Activity Selection Problem | Greedy Algo-1

* **Difficulty Level :** [Easy](https://www.geeksforgeeks.org/easy/)
* **Last Updated :** 23 Dec, 2022

 Read

 Discuss(20+)

 Courses

 Practice

 Video

*Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems.*

*An optimization problem can be solved using Greedy if the problem has the following property:*

* *At every step, we can make a choice that looks best at the moment, and we get the optimal solution to the complete problem.*

If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. But Greedy algorithms cannot always be applied. For example, the [Fractional Knapsack](https://www.geeksforgeeks.org/fractional-knapsack-problem/) problem can be solved using Greedy, but [0-1 Knapsack](https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/) cannot be solved using Greedy.

## Following are some standard algorithms that are Greedy algorithms:

### **1)**[**Kruskal’s Minimum Spanning Tree (MST)**](https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/)**:**

In Kruskal’s algorithm, we create an MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn’t cause a cycle in the MST constructed so far

### **2)**[**Prim’s Minimum Spanning Tree**](https://www.geeksforgeeks.org/prims-algorithm-using-priority_queue-stl/)**:**

In Prim’s algorithm also, we create a MST by picking edges one by one. We maintain two sets: a set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets

### **3)**[**Dijkstra’s Shortest Path**](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/)**:**

Dijkstra’s algorithm is very similar to Prim’s algorithm. The shortest-path tree is built up, edge by edge. We maintain two sets: a set of the vertices already included in the tree and a set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from the source to the set that contains not yet included vertices

### **4)**[**Huffman Coding**](https://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/)**:**

Huffman Coding is a loss-less compression technique. It assigns variable-length bit codes to different characters. The Greedy Choice is to assign the least bit length code to the most frequent character.

The greedy algorithms are sometimes also used to get an approximation for Hard optimization problems. For example, [Traveling Salesman Problem](https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/) is an NP-Hard problem. A Greedy choice for this problem is to pick the nearest unvisited city from the current city at every step. These solutions don’t always produce the best optimal solution but can be used to get an approximately optimal solution.

Here let us see one such problem that can be solved using Greedy algorithm

### **Problem:**

You are given **n** activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

**Examples:**

***Input:****start[]  =  {10, 12, 20}, finish[] =  {20, 25, 30}****Output:****0 2****Explanation:****A person can perform at most two activities. The   
maximum set of activities that can be executed   
is {0, 2} [ These are indexes in start[] and finish[] ]*

***Input:****start[]  =  {1, 3, 0, 5, 8, 5}, finish[] =  {2, 4, 6, 7, 9, 9};****Output:****0 1 3 4****Explanation:****A person can perform at most four activities. The   
maximum set of activities that can be executed   
is {0, 1, 3, 4} [ These are indexes in start[] and finish[]*

Recommended Problem

Activity Selection

[Dynamic Programming](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Dynamic%20Programming&sortBy=submissions)

[Binary Search](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Binary%20Search&sortBy=submissions)

+1 more

[Flipkart](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Flipkart&sortBy=submissions)

[Morgan Stanley](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Morgan%20Stanley&sortBy=submissions)

+1 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/activity-selection-1587115620/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article" \o "Permalink to Activity Selection)

Submission count: 93.2K

**Approach:** To solve the problem follow the below idea:

*The greedy choice is to always pick the next activity whose finish time is the least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as the minimum finishing time activity*

Follow the given steps to solve the problem:

* Sort the activities according to their finishing time
* Select the first activity from the sorted array and print it
* Do the following for the remaining activities in the sorted array
  + If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it

**Note:**In the implementation, it is assumed that the activities are already sorted according to their finish time

Below is the implementation of the above approach.

// C++ program for activity selection problem.

// The following implementation assumes that the activities

// are already sorted according to their finish time

#include <iostream>

using namespace std;

// Prints a maximum set of activities that can be done by a

// single person, one at a time.

void printMaxActivities(int s[], int f[], int n)

{

    int i, j;

    cout << "Following activities are selected" << endl;

    // The first activity always gets selected

    i = 0;

    cout << i << " ";

    // Consider rest of the activities

    for (j = 1; j < n; j++)

    {

        // If this activity has start time greater than or

        // equal to the finish time of previously selected

        // activity, then select it

        if (s[j] >= f[i])

        {

            cout << j << " ";

            i = j;

        }

    }

}

// Driver code

int main()

{

    int s[] = {1, 3, 0, 5, 8, 5};

    int f[] = {2, 4, 6, 7, 9, 9};

    int n = sizeof(s) / sizeof(s[0]);

    // Function call

    printMaxActivities(s, f, n);

    return 0;

}

// this code contributed by shivanisinghss2110

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture7-main\activity2\src> .\asp

Following activities are selected

0 1 3 4

**## Task 4: Individual (at home)**

- Refer to te following link. Explain the differences between Greedy Algorithm and Dynamic Programming

<https://www.geeksforgeeks.org/greedy-approach-vs-dynamic-programming/>

### **Below are some major differences between Greedy method and Dynamic programming:**

| **Feature** | **Greedy method** | **Dynamic programming** |
| --- | --- | --- |
| **Feasibility** | In a [greedy Algorithm](https://www.geeksforgeeks.org/greedy-algorithms/), we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution. | In [Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming/) we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution . |
| **Optimality** | In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution. | It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best. |
| **Recursion** | A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage. | A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states. |
| **Memoization** | It is more efficient in terms of memory as it never look back or revise previous choices | It requires Dynamic Programming table for Memoization and it increases it’s memory complexity. |
| **Time        complexity** | Greedy methods are generally faster. For example, [Dijkstra’s shortest path](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/) algorithm takes O(ELogV + VLogV) time. | Dynamic Programming is generally slower. For example, [Bellman Ford algorithm](https://www.geeksforgeeks.org/bellman-ford-algorithm-simple-implementation/) takes O(VE) time. |
| **Fashion** | The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices. | Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions. |
| **Example** | Fractional knapsack . | 0/1 knapsack problem |

**Related Articles:**

* [Greedy Algorithm](http://www.geeksforgeeks.org/greedy-algorithms/)
* [Dynamic programming](http://www.geeksforgeeks.org/dynamic-programming/)
* [Optimal substructure](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)
* [Overlapping subproblem](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)

ACTIVITY 3

**# Activities**

**## Task 1**

- Refer to te following link. Explain the Dijkstra's Shortest Path Algorithm.

  https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/

**## Task 2**

- Explain how the code in `./src/dspa.cpp` works. Refer to the following link:

  https://www.geeksforgeeks.org/c-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/

**## Task 3**

- Explain how the code in `./src/mspt.cpp` works. Refer to the following link:

  https://www.tutorialspoint.com/kruskal-s-minimum-spanning-tree-algorithm-greedy-algorithm-in-cplusplus#

**## Task 4: Individual (at home)**

- Refer to te following link. Explain when to use the greedy methods and when to avoid them.

  https://www.freecodecamp.org/news/when-to-use-greedy-algorithms/

**## Link(s)**

- https://cpp.sh/

ANSWERS:

**## Task 1**

- Refer to te following link. Explain the Dijkstra's Shortest Path Algorithm.

<https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/>

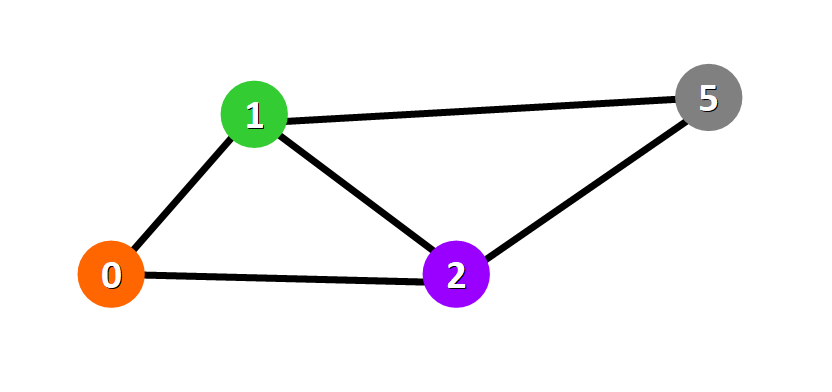
**Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction**

### Basic Concepts

Graphs are data structures used to represent "connections" between pairs of elements.

* These elements are called **nodes**. They represent real-life objects, persons, or entities.
* The connections between nodes are called **edges**.

This is a graphical representation of a graph:

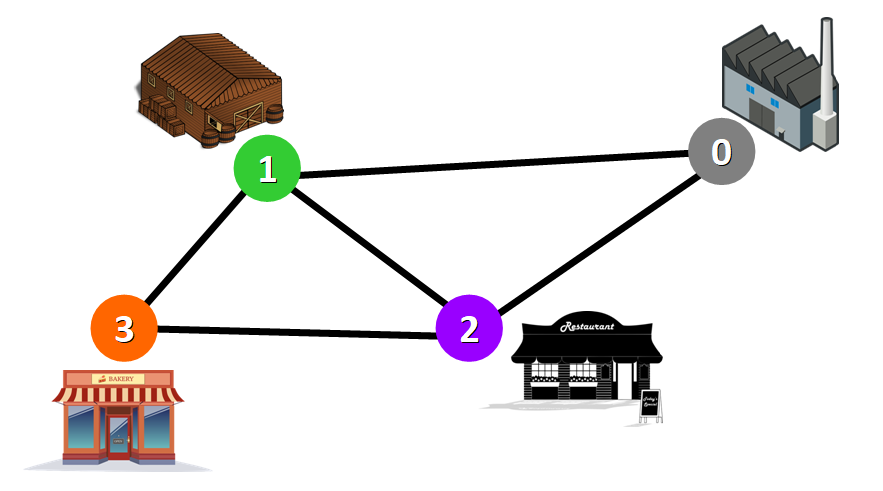


**Nodes**are represented with colored circles and **edges**are represented with lines that connect these circles.

**💡 Tip:**Two nodes are connected if there is an edge between them.

### Applications

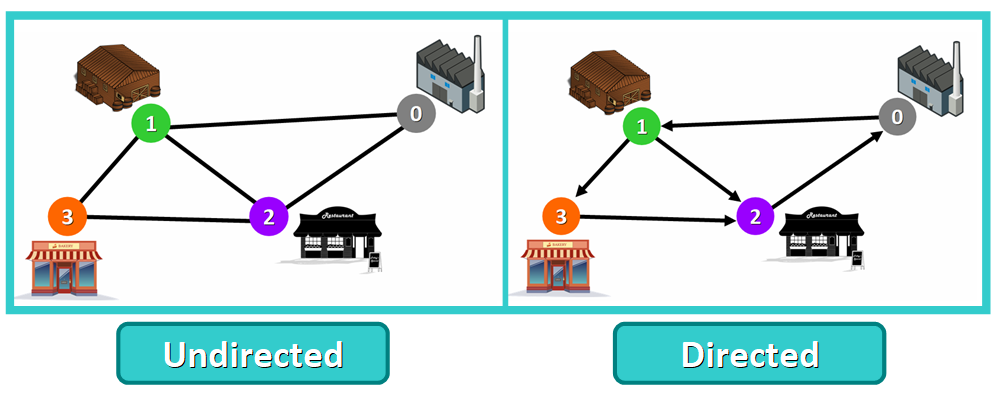
Graphs are directly applicable to real-world scenarios. For example, we could use graphs to model a transportation network where nodes would represent facilities that send or receive products and edges would represent roads or paths that connect them (see below).

Network represented with a graph

### Types of Graphs

Graphs can be:

* **Undirected:**if for every pair of connected nodes, you can go from one node to the other in both directions.
* **Directed:**if for every pair of connected nodes, you can only go from one node to another in a specific direction. We use arrows instead of simple lines to represent directed edges.

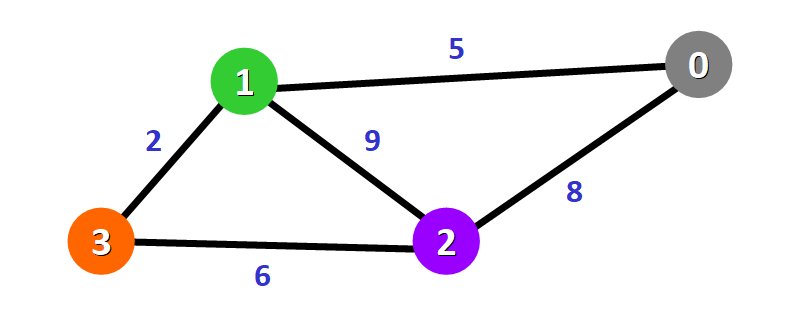


**💡 Tip:** in this article, we will work with **undirected**graphs.

### Weighted Graphs

A **weight graph** is a graph whose edges have a "weight" or "cost". The weight of an edge can represent distance, time, or anything that models the "connection" between the pair of nodes it connects.

For example, in the weighted graph below you can see a blue number next to each edge. This number is used to represent the weight of the corresponding edge.



**💡 Tip:** These weights are essential for Dijkstra's Algorithm. You will see why in just a moment.

## 🔸 Introduction to Dijkstra's Algorithm

Now that you know the basic concepts of graphs, let's start diving into this amazing algorithm.

* Purpose and Use Cases
* History
* Basics of the Algorithm
* Requirements

### Purpose and Use Cases

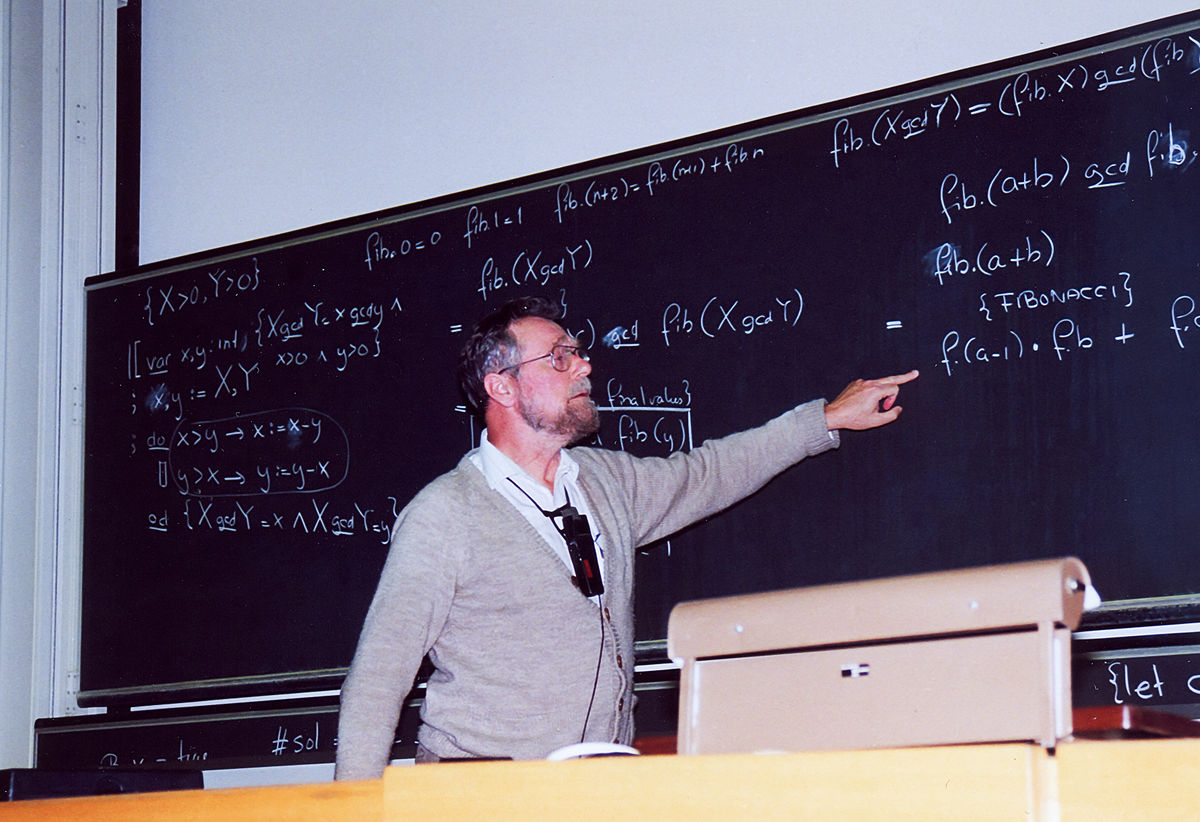
With Dijkstra's Algorithm, you can find the shortest path between nodes in a graph. Particularly, you can **find the shortest path from a node (called the "source node") to all other nodes in the graph**, producing a shortest-path tree.

This algorithm is used in GPS devices to find the shortest path between the current location and the destination. It has broad applications in industry, specially in domains that require modeling networks.

### History

This algorithm was created and published by [Dr. Edsger W. Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra), a brilliant Dutch computer scientist and software engineer.

In 1959, he published a 3-page article titled "A note on two problems in connexion with graphs" where he explained his new algorithm.

[Dr. Edsger Dijkstra](https://commons.wikimedia.org/wiki/File:Edsger_Dijkstra_1994.jpg) at [ETH Zurich](https://en.wikipedia.org/wiki/ETH_Zurich) in 1994 (image by Andreas F. Borchert)

During an interview in 2001, Dr. Dijkstra revealed how and why he designed the algorithm:

What’s the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. In fact, it was published in 1959, three years later. The publication is still quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. Without pencil and paper you are almost forced to avoid all avoidable complexities. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame. — As quoted in the article [Edsger W. Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra) from [An interview with Edsger W. Dijkstra](https://dl.acm.org/doi/pdf/10.1145/1787234.1787249).

⭐ **Unbelievable, right?** In just 20 minutes, Dr. Dijkstra designed one of the most famous algorithms in the history of Computer Science.

### Basics of Dijkstra's Algorithm

* Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
* The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
* Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
* The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

### Requirements

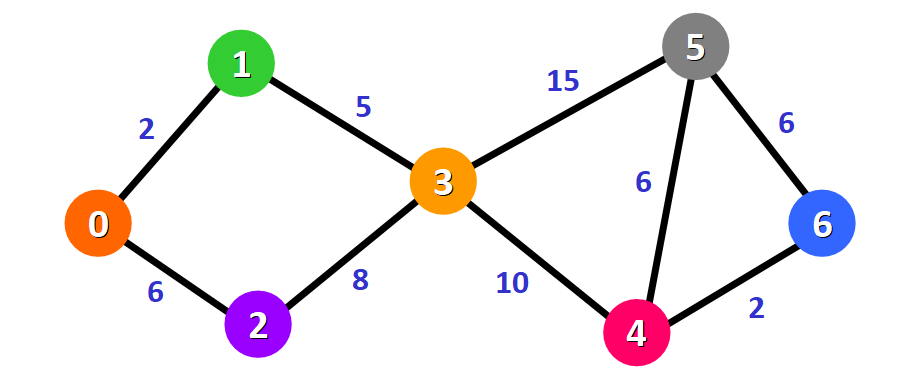
Dijkstra's Algorithm can only work with graphs that have **positive** weights. This is because, during the process, the weights of the edges have to be added to find the shortest path.

If there is a negative weight in the graph, then the algorithm will not work properly. Once a node has been marked as "visited", the current path to that node is marked as the shortest path to reach that node. And negative weights can alter this if the total weight can be decremented after this step has occurred.

## 🔹 Example of Dijkstra's Algorithm

Now that you know more about this algorithm, let's see how it works behind the scenes with a a step-by-step example.

We have this graph:



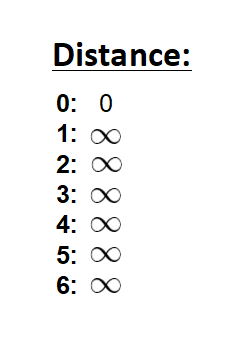
The algorithm will generate the shortest path from node 0 to all the other nodes in the graph.

**💡 Tip:**For this graph, we will assume that the weight of the edges represents the distance between two nodes.

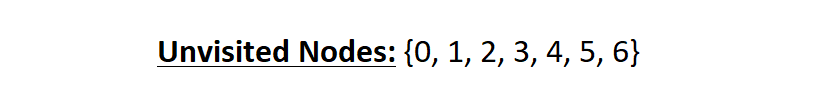
We will have the shortest path from node 0 to node 1, from node 0 to node 2, from node 0 to node 3, and so on for every node in the graph.

Initially, we have this list of distances (please see the list below):

* The distance from the source node to itself is 0. For this example, the source node will be node 0 but it can be any node that you choose.
* The distance from the source node to all other nodes has not been determined yet, so we use the infinity symbol to represent this initially.

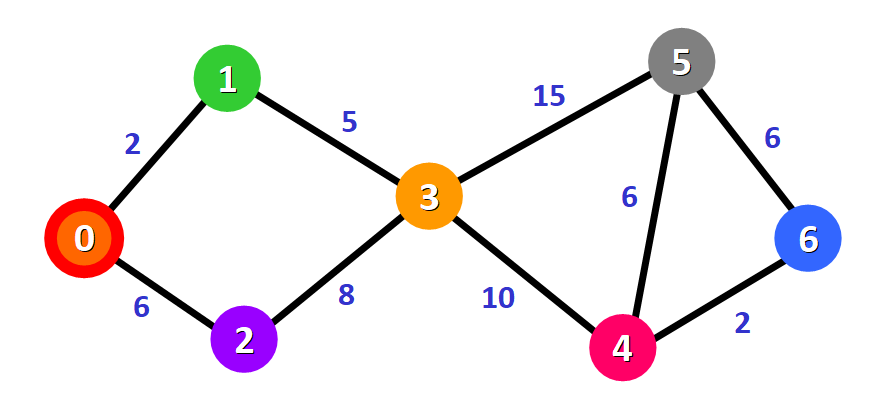


We also have this list (see below) to keep track of the nodes that have not been visited yet (nodes that have not been included in the path):

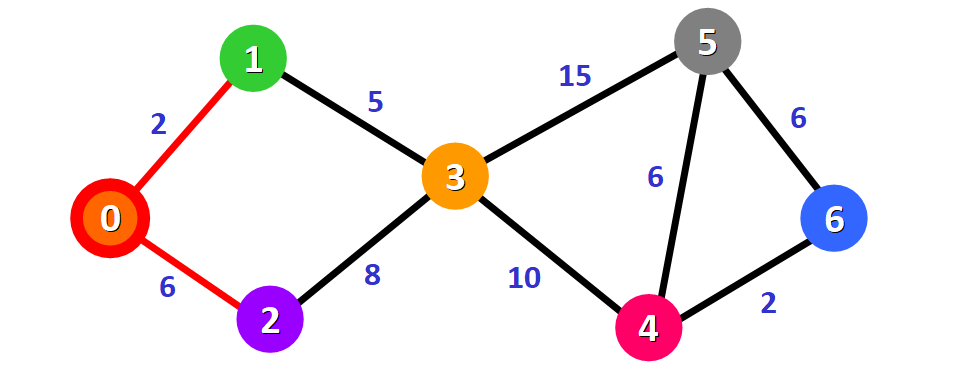


**💡 Tip:**Remember that the algorithm is completed once all nodes have been added to the path.

Since we are choosing to start at node 0, we can mark this node as visited. Equivalently, we cross it off from the list of unvisited nodes and add a red border to the corresponding node in diagram:

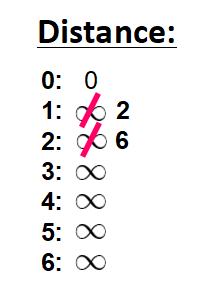
image-87

Now we need to start checking the distance from node 0 to its adjacent nodes. As you can see, these are nodes 1 and 2 (see the red edges):



**💡 Tip:** This doesn't mean that we are immediately adding the two adjacent nodes to the shortest path. Before adding a node to this path, we need to check if we have found the shortest path to reach it. We are simply making an initial examination process to see the options available.

We need to update the distances from node 0 to node 1 and node 2 with the weights of the edges that connect them to node 0 (the source node). These weights are 2 and 6, respectively:

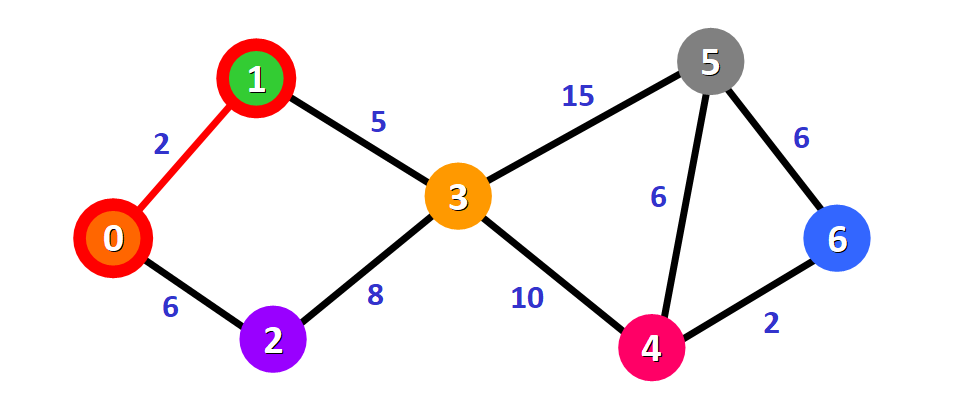


After updating the distances of the adjacent nodes, we need to:

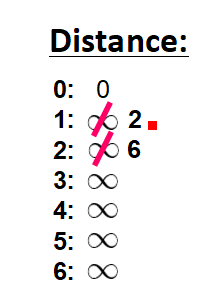
* Select the node that is closest to the source node based on the current known distances.
* Mark it as visited.
* Add it to the path.

If we check the list of distances, we can see that node 1 has the shortest distance to the source node (a distance of 2), so we add it to the path.

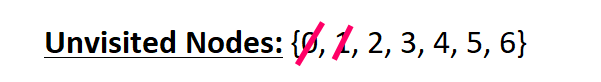
In the diagram, we can represent this with a red edge:



We mark it with a red square in the list to represent that it has been "visited" and that we have found the shortest path to this node:

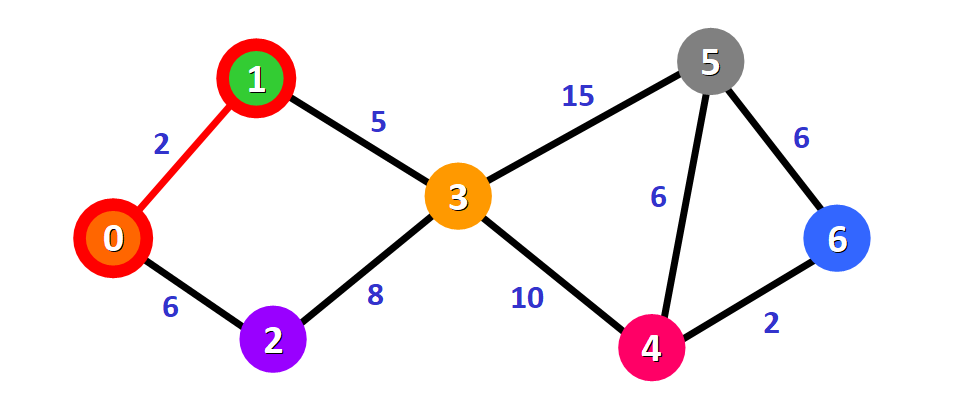


We cross it off from the list of unvisited nodes:

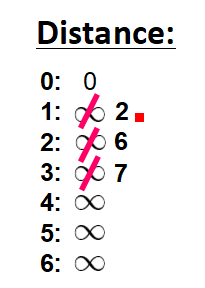


Now we need to analyze the new adjacent nodes to find the shortest path to reach them. We will only analyze the nodes that are adjacent to the nodes that are already part of the shortest path (the path marked with red edges).

Node 3 and node 2 are both adjacent to nodes that are already in the path because they are directly connected to node 1 and node 0, respectively, as you can see below. These are the nodes that we will analyze in the next step.



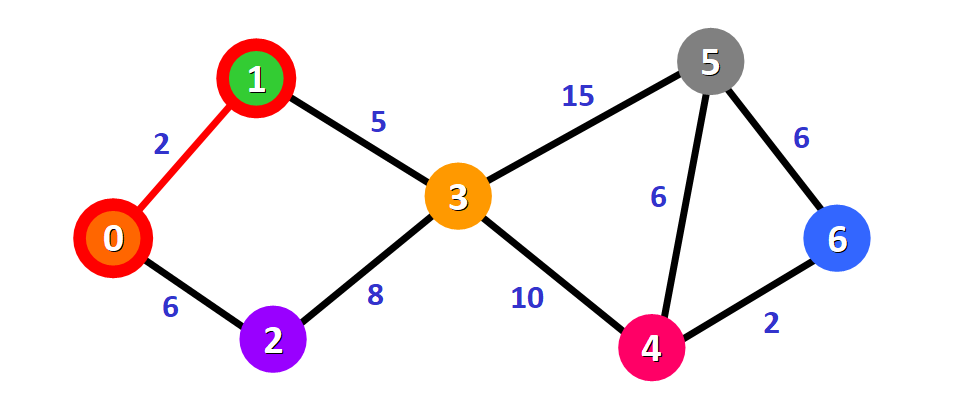
Since we already have the distance from the source node to node 2 written down in our list, we don't need to update the distance this time. We only need to update the distance from the source node to the new adjacent node (node 3):



This distance is **7**. Let's see why.

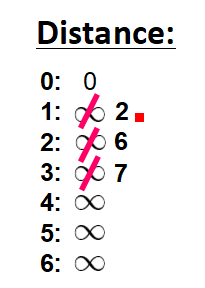
To find the distance from the source node to another node (in this case, node 3), we add the weights of all the edges that form the shortest path to reach that node:

* **For node**3**:** the total distance is **7** because we add the weights of the edges that form the path 0 -> 1 -> 3 (2  for the edge 0 -> 1 and 5 for the edge 1 -> 3).

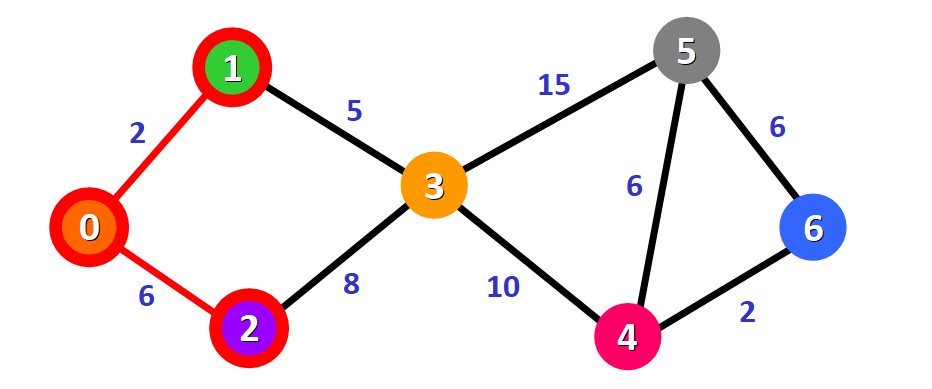


Now that we have the distance to the adjacent nodes, we have to choose which node will be added to the path. We must select the **unvisited**node with the shortest (currently known) distance to the source node.

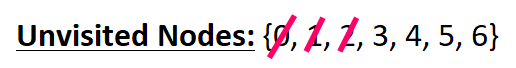
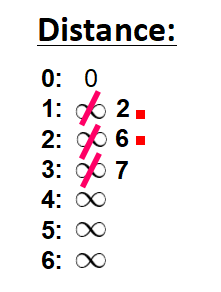
From the list of distances, we can immediately detect that this is node 2 with distance **6**:



We add it to the path graphically with a red border around the node and a red edge:

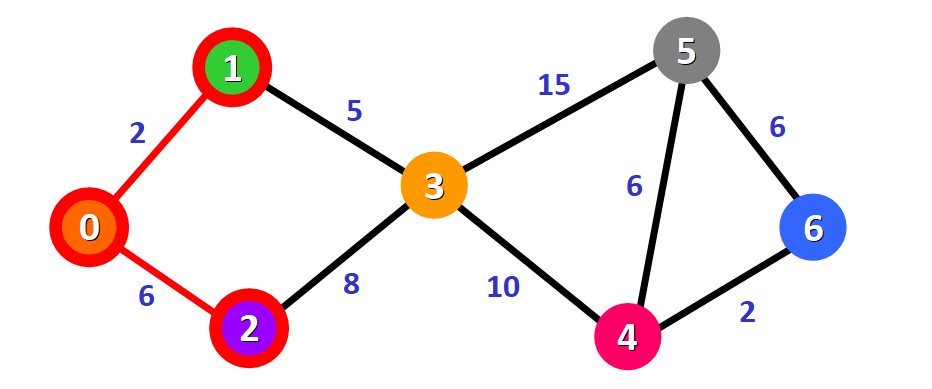


We also mark it as visited by adding a small red square in the list of distances and crossing it off from the list of unvisited nodes:



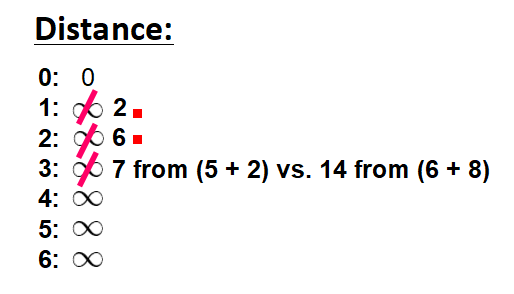
Now we need to repeat the process to find the shortest path from the source node to the new adjacent node, which is node 3.

You can see that we have two possible paths 0 -> 1 -> 3 or 0 -> 2 -> 3. Let's see how we can decide which one is the shortest path.



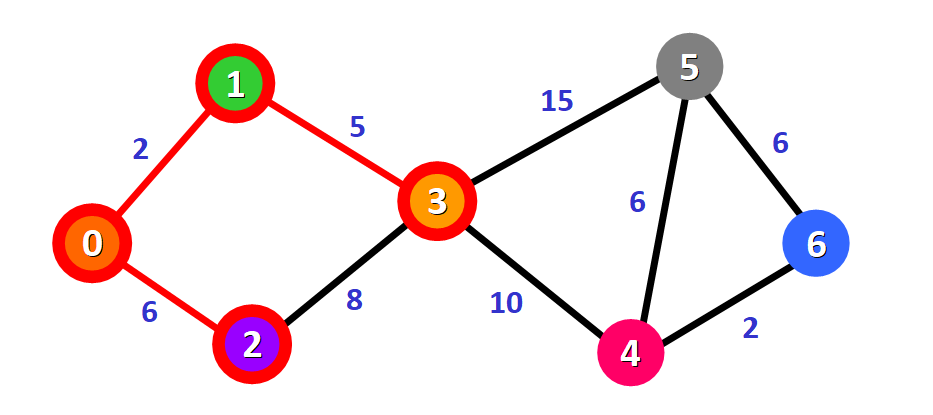
Node 3 already has a distance in the list that was recorded previously (**7,**see the list below). This distance was the result of a previous step, where we added the weights 5 and 2 of the two edges that we needed to cross to follow the path 0 -> 1 -> 3.

But now we have another alternative. If we choose to follow the path 0 -> 2 -> 3, we would need to follow two edges 0 -> 2 and 2 -> 3 with weights **6** and **8**,respectively,which represents a total distance of **14**.

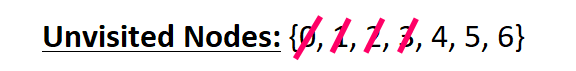
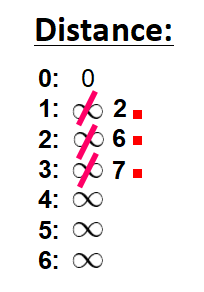


Clearly, the first (existing) distance is shorter (7 vs. 14), so we will choose to keep the original path 0 -> 1 -> 3. **We only update the distance if the new path is shorter.**

Therefore, we add this node to the path using the first alternative: 0 -> 1 -> 3.

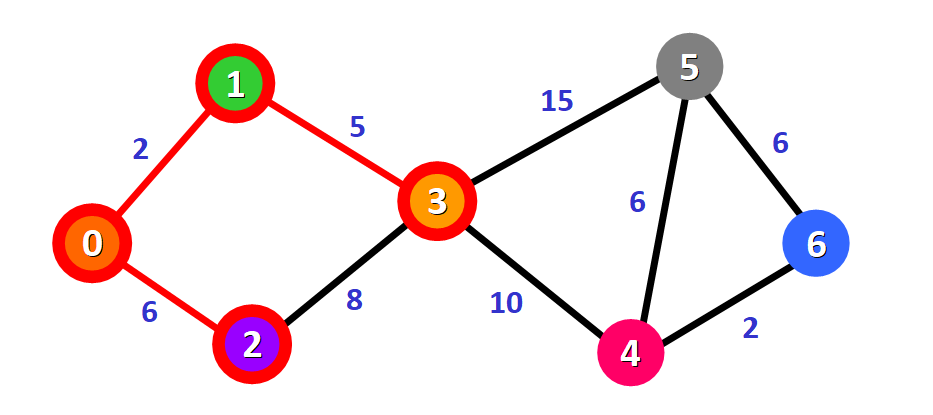


We mark this node as visited and cross it off from the list of unvisited nodes:



Now we repeat the process again.

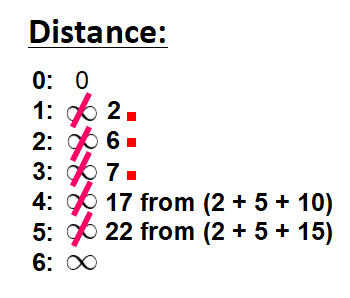
We need to check the new adjacent nodes that we have not visited so far. This time, these nodes are node 4 and node 5 since they are adjacent to node 3.



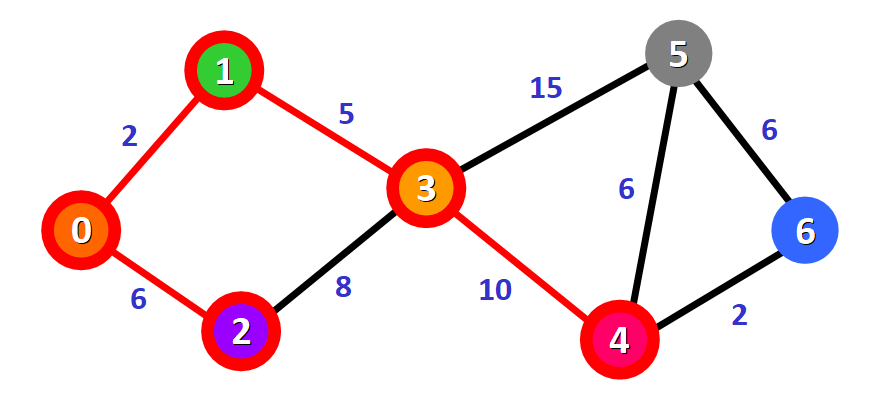
We update the distances of these nodes to the source node, always trying to find a shorter path, if possible:

* **For node**4**:** the distance is **17** from the path  0 -> 1 -> 3 -> 4.
* **For node**5**:** the distance is **22** from the path 0 -> 1 -> 3 -> 5.

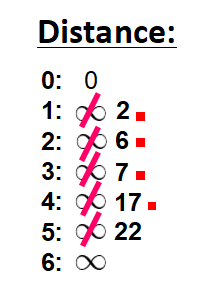
**💡 Tip:** Notice that we can only consider extending the shortest path (marked in red). We cannot consider paths that will take us through edges that have not been added to the shortest path (for example, we cannot form a path that goes through the edge 2 -> 3).



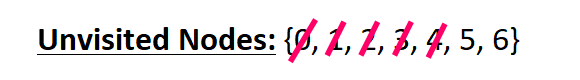
We need to choose which unvisited node will be marked as visited now. In this case, it's node 4 because it has the shortest distance in the list of distances. We add it graphically in the diagram:



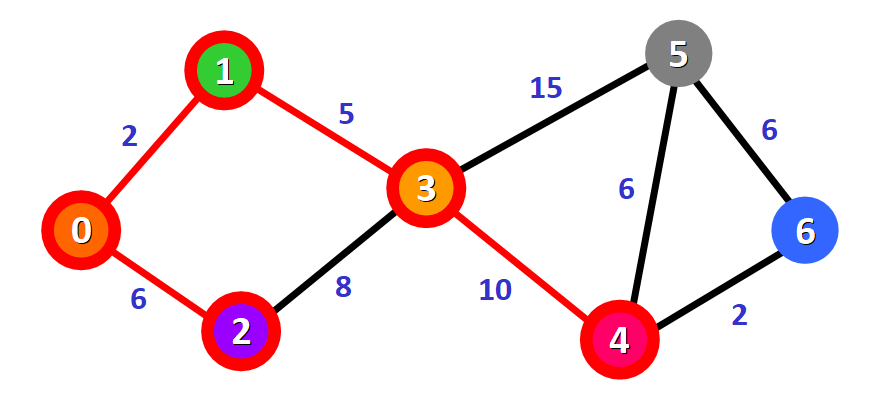
We also mark it as "visited" by adding a small red square in the list:



And we cross it off from the list of unvisited nodes:



And we repeat the process again. We check the adjacent nodes: node 5 and node 6. We need to analyze each possible path that we can follow to reach them from nodes that have already been marked as visited and added to the path.



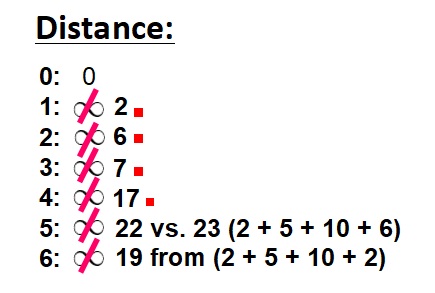
**For node**5**:**

* The first option is to follow the path 0 -> 1 -> 3 -> 5, which has a distance of **22**from the source node (2 + 5 + 15). This distance was already recorded in the list of distances in a previous step.
* The second option would be to follow the path 0 -> 1 -> 3 -> 4 -> 5, which has a distance of **23**from the source node (2 + 5 + 10 + 6).

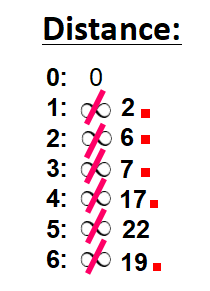
Clearly, the first path is shorter, so we choose it for node 5.

**For node**6**:**

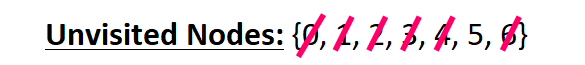
* The path available is 0 -> 1 -> 3 -> 4 -> 6, which has a distance of **19** from the source node (2 + 5 + 10 + 2).



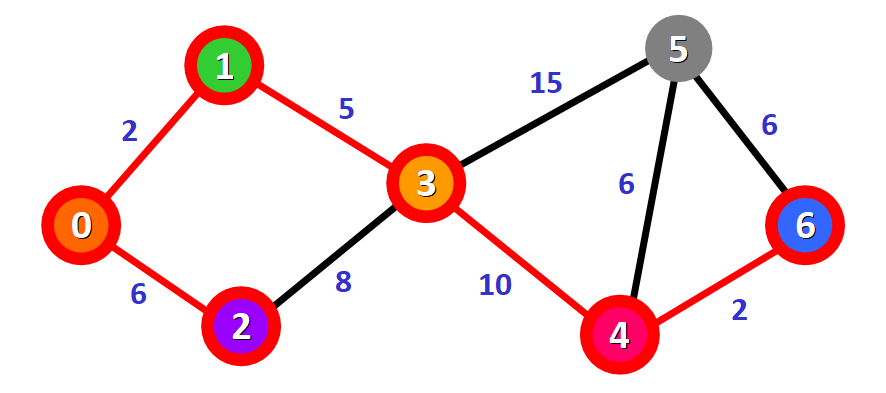
We mark the node with the shortest (currently known) distance as visited. In this case, node 6.



And we cross it off from the list of unvisited nodes:



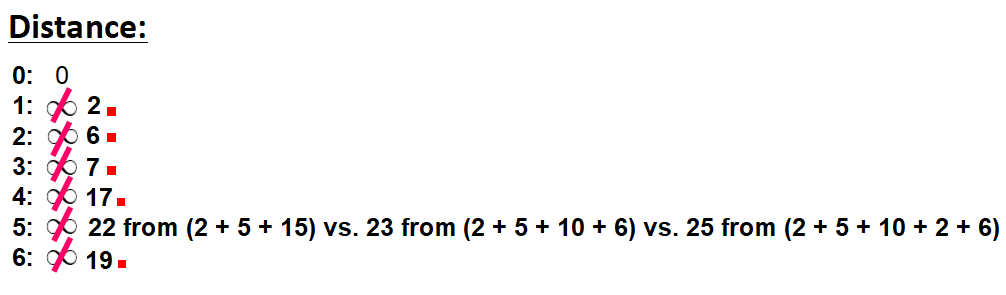
Now we have this path (marked in red):



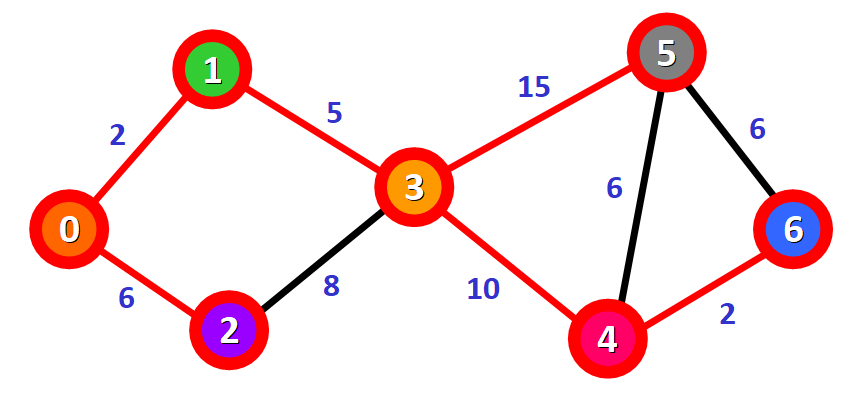
Only one node has not been visited yet, node 5. Let's see how we can include it in the path.

There are three different paths that we can take to reach node 5 from the nodes that have been added to the path:

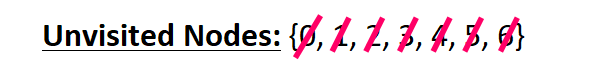
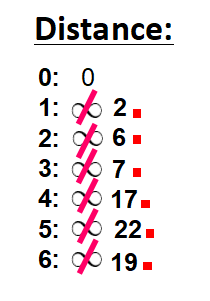
* **Option 1:**0 -> 1 -> 3 -> 5 with a distance of **22**(2 + 5 + 15).
* **Option 2:**0 -> 1 -> 3 -> 4 -> 5 with a distance of **23** (2 + 5 + 10 + 6).
* **Option 3:** 0 -> 1 -> 3 -> 4 -> 6 -> 5 with a distance of **25**(2 + 5 + 10 + 2 + 6).



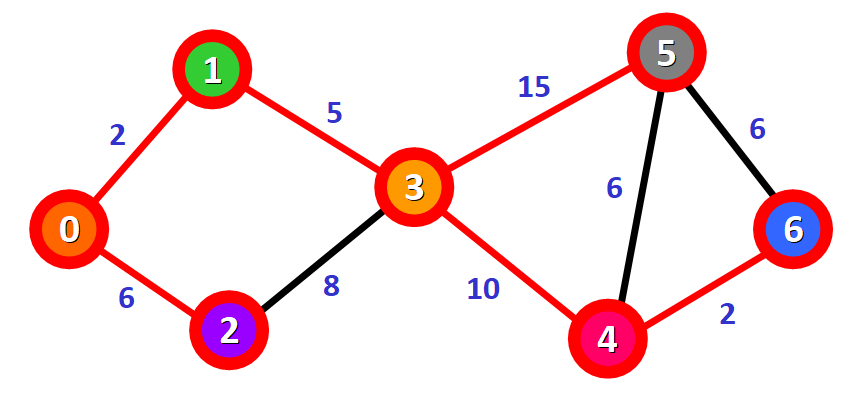
We select the shortest path: 0 -> 1 -> 3 -> 5 with a distance of **22**.



We mark the node as visited and cross it off from the list of unvisited nodes:



**And voilà!** We have the final result with the shortest path from node 0 to each node in the graph.



In the diagram, the red lines mark the edges that belong to the shortest path. You need to follow these edges to follow the shortest path to reach a given node in the graph starting from node 0.

For example, if you want to reach node 6 starting from node 0, you just need to follow the red edges and you will be following the shortest path 0 -> 1 -> 3 -> 4 - > 6 automatically.

## 🔸 In Summary

* Graphs are used to model connections between objects, people, or entities. They have two main elements: nodes and edges. Nodes represent objects and edges represent the connections between these objects.
* Dijkstra's Algorithm finds the shortest path between a given node (which is called the "source node") and all other nodes in a graph.
* This algorithm uses the weights of the edges to find the path that minimizes the total distance (weight) between the source node and all other nodes.

**## Task 2**

- Explain how the code in `./src/dspa.cpp` works. Refer to the following link:

<https://www.geeksforgeeks.org/c-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/>

**C / C++ Program for Dijkstra’s shortest path algorithm | Greedy Algo-7**

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.  
Dijkstra’s algorithm is very similar to [Prim’s algorithm for minimum spanning tree](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/). Like Prim’s MST, we generate a*SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.  
Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.   
Algorithm   
**1)** Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.   
**2)** Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.   
**3)** While *sptSet* doesn’t include all vertices   
….**a)** Pick a vertex u which is not there in *sptSet* and has minimum distance value.   
….**b)** Include u to *sptSet*.   
….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

// A C++ program for Dijkstra's single source shortest path algorithm.

// The program is for adjacency matrix representation of the graph

#include <limits.h>

#include <stdio.h>

// Number of vertices in the graph

#define V 9

// A utility function to find the vertex with minimum distance value, from

// the set of vertices not yet included in shortest path tree

int minDistance(int dist[], bool sptSet[])

{

    // Initialize min value

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < V; v++)

        if (sptSet[v] == false && dist[v] <= min)

            min = dist[v], min\_index = v;

    return min\_index;

}

// A utility function to print the constructed distance array

void printSolution(int dist[], int n)

{

    printf("Vertex Distance from Source\n");

    for (int i = 0; i < V; i++)

        printf("%d \t\t %d\n", i, dist[i]);

}

// Function that implements Dijkstra's single source shortest path algorithm

// for a graph represented using adjacency matrix representation

void dijkstra(int graph[V][V], int src)

{

    int dist[V]; // The output array. dist[i] will hold the shortest

    // distance from src to i

    bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest

    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false

    for (int i = 0; i < V; i++)

        dist[i] = INT\_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0

    dist[src] = 0;

    // Find shortest path for all vertices

    for (int count = 0; count < V - 1; count++)

    {

        // Pick the minimum distance vertex from the set of vertices not

        // yet processed. u is always equal to src in the first iteration.

        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed

        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the picked vertex.

        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an edge from

            // u to v, and total weight of path from src to v through u is

            // smaller than current value of dist[v]

            if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v])

                dist[v] = dist[u] + graph[u][v];

    }

    // print the constructed distance array

    printSolution(dist, V);

}

// driver program to test above function

int main()

{

    /\* Let us create the example graph discussed above \*/

    int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},

                       {4, 0, 8, 0, 0, 0, 0, 11, 0},

                       {0, 8, 0, 7, 0, 4, 0, 0, 2},

                       {0, 0, 7, 0, 9, 14, 0, 0, 0},

                       {0, 0, 0, 9, 0, 10, 0, 0, 0},

                       {0, 0, 4, 14, 10, 0, 2, 0, 0},

                       {0, 0, 0, 0, 0, 2, 0, 1, 6},

                       {8, 11, 0, 0, 0, 0, 1, 0, 7},

                       {0, 0, 2, 0, 0, 0, 6, 7, 0}};

    dijkstra(graph, 0);

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture7-main\activity3\src> .\dspa

Vertex Distance from Source

0 0

1 4

2 12

3 19

4 21

5 11

6 9

7 8

8 14

**Time Complexity:** The time complexity of Dijkstra’s algorithm is O(V^2). This is because the algorithm uses two nested loops to traverse the graph and find the shortest path from the source node to all other nodes.

**Space Complexity:**The space complexity of Dijkstra’s algorithm is O(V), where V is the number of vertices in the graph. This is because the algorithm uses an array of size V to store the distances from the source node to all other nodes.

**## Task 3**

- Explain how the code in `./src/mspt.cpp` works. Refer to the following link:

<https://www.tutorialspoint.com/kruskal-s-minimum-spanning-tree-algorithm-greedy-algorithm-in-cplusplus#>

# **Kruskal's Minimum Spanning Tree Algorithm-Greedy algorithm in C++**

[C++](https://www.tutorialspoint.com/articles/category/Cplusplus)[Server Side Programming](https://www.tutorialspoint.com/articles/category/Server-Side-Programming)[Programming](https://www.tutorialspoint.com/articles/category/Programming)

A spanning tree is a linked and undirected graph subgraph that connects all vertices. Many spanning trees can exist in a graph. The minimum spanning tree (MST) on each graph is the same weight or less than all other spanning trees. Weights are assigned to edges of spanning trees and the sum is the weight assigned to each edge. As V is the number of vertices in the graph, the minimum spanning tree has edges of (V - 1), where V is the number of edges.

## Finding minimum spanning tree using Kruskal’s algorithm

* All of the edges should be arranged in a non-descending sequence of weight.
* Choose the smallest edge. This edge is included if the cycle is not formed.
* Step 2 should be performed until the spanning tree has (V-1) edges.

In this scenario, we are told to use a greedy method. The greedy option is to select the edge with the least amount of weight. As an illustration: The minimum spanning tree for this graph is (9-1)= 8 edges.

Diagram

Description automatically generated

After sorting:

Weight  Src    Dest

21       27    26

22       28    22

22       26    25

24       20    21

24       22    25

26       28    26

27       22    23

27       27    28

28       20    27

28       21    22

29       23    24

30       25    24

31       21    27

34       23    25

**Now we need to pick all the edges according to the sort.**

Edge 26-27-> included because no cycle is formed

Edge 28-22-> included because no cycle is formed

Edge 26-25-> included because no cycle is formed.

Edge 20-21-> included because no cycle is formed

Edge 22-25-> included because no cycle is formed.

Edge 28-26-> discarded as cycle is formed

Edge 22-23-> included because no cycle is formed

Edge 27-28-> discarded as cycle is formed

Edge 20-27-> included because no cycle is formed

Edge 21-22-> discarded as cycle is formed

Edge 23-24-> included because no cycle is formed

As the number of edges is (V-1), so the algorithm ends here.

## Example

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <iostream>

using namespace std;

struct Edge {

   int src, dest, weight;

};

struct Graph {

   int V, E;

   struct Edge\* edge;

};

struct Graph\* createGraph(int V, int E){

   struct Graph\* graph = (struct Graph\*)(malloc(sizeof(struct Graph)));

   graph->V = V;

   graph->E = E;

   graph->edge = (struct Edge\*)malloc(sizeof( struct Edge)\*E);

   return graph;

}

struct subset {

   int parent;

   int rank;

};

int find(struct subset subsets[], int i){

   if (subsets[i].parent != i)

      subsets[i].parent

   = find(subsets, subsets[i].parent);

   return subsets[i].parent;

}

void Union(struct subset subsets[], int x, int y){

   int xroot = find(subsets, x);

   int yroot = find(subsets, y);

   if (subsets[xroot].rank < subsets[yroot].rank)

      subsets[xroot].parent = yroot;

   else if (subsets[xroot].rank > subsets[yroot].rank)

      subsets[yroot].parent = xroot;

   else{

      subsets[yroot].parent = xroot;

      subsets[xroot].rank++;

   }

}

int myComp(const void\* a, const void\* b){

   struct Edge\* a1 = (struct Edge\*)a;

   struct Edge\* b1 = (struct Edge\*)b;

   return a1->weight > b1->weight;

}

void KruskalMST(struct Graph\* graph){

   int V = graph->V;

   struct Edge

   result[V];

   int e = 0;

   int i = 0;

   qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

   struct subset\* subsets

   = (struct subset\*)malloc(V \* sizeof(struct subset));

   for (int v = 0; v < V; ++v) {

      subsets[v].parent = v;

      subsets[v].rank = 0;

   }

   while (e < V - 1 && i < graph->E) {

      struct Edge next\_edge = graph->edge[i++];

      int x = find(subsets, next\_edge.src);

      int y = find(subsets, next\_edge.dest);

      if (x != y) {

         result[e++] = next\_edge;

         Union(subsets, x, y);

      }

   }

   printf("Following are the edges in the constructed MST\n");

   int minimumCost = 0;

   for (i = 0; i < e; ++i){

      printf("%d -- %d == %d\n", result[i].src,

      result[i].dest, result[i].weight);

      minimumCost += result[i].weight;

   }

   printf("Minimum Cost Spanning tree : %d",minimumCost);

   return;

}

int main(){

   /\* Let us create the following weighted graph

   30

   0--------1

   | \       |

   26| 25\ |15

   | \ |

   22--------23

   24 \*/

   int V = 24;

   int E = 25;

   struct Graph\* graph = createGraph(V, E);

   graph->edge[0].src = 20;

   graph->edge[0].dest = 21;

   graph->edge[0].weight = 30;

   graph->edge[1].src = 20;

   graph->edge[1].dest = 22;

   graph->edge[1].weight = 26;

   graph->edge[2].src = 20;

   graph->edge[2].dest = 23;

   graph->edge[2].weight = 25;

   graph->edge[3].src = 21;

   graph->edge[3].dest = 23;

   graph->edge[3].weight = 35;

   graph->edge[4].src = 22;

   graph->edge[4].dest = 23;

   graph->edge[4].weight = 24;

   KruskalMST(graph);

   return 0;

}

Following are the edges in the constructed MST

22 -- 23 == 24

20 -- 23 == 25

20 -- 21 == 30

Minimum Cost Spanning tree : 79

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture7-main\activity3\src> .\mspt

Following are the edges in the constructed MST

22 -- 23 == 24

20 -- 23 == 25

20 -- 21 == 30

Minimum Cost Spanning tree : 79

**## Task 4: Individual (at home)**

- Refer to te following link. Explain when to use the greedy methods and when to avoid them.

<https://www.freecodecamp.org/news/when-to-use-greedy-algorithms/>

Greedy algorithms try to find the optimal solution by taking the best available choice at every step.

For example, you can greedily approach your life. You can always take the path that maximizes your happiness today. But that doesn't mean you'll be happier tomorrow.

Similarly, there are problems for which greedy algorithms don't yield the best solution. Actually, they might yield the worst possible solution.

But there are other cases in which we can obtain a solution that is good enough by using a greedy strategy.

In this article, I'll write about greedy algorithms and the use of this strategy even when it doesn't guarantee that you'll find an optimal solution.

The first section is an introduction to greedy algorithms and well-known problems that are solvable using this strategy. Then I'll talk about problems in which the greedy strategy is a really bad option. And finally, I'll show you an example of a good approximation through a greedy algorithm.

***Note****: Most of the algorithms and problems I discuss in this article include graphs. It would be good if you are familiar with graphs to get the most out of this post.*

## How greedy algorithms work

Greedy algorithms always choose the best available option.

In general, they are computationally cheaper than other families of algorithms like dynamic programming, or brute force. This is because they don't explore the solution space too much. And, for the same reason, they don't find the best solution to a lot of problems.

But there are lots of problems that are solvable with a greedy strategy, and in those cases that strategy is precisely the best way to go.

One of the most popular greedy algorithms is Dijkstra's algorithm that finds the path with the minimum cost from one vertex to the others in a graph.

This algorithm finds such a path by always going to the nearest vertex. That's why we say it is a greedy algorithm.

This is pseudocode for the algorithm. I denote with G the graph and with s the source node.

Dijkstra(G, s):

distances <- list of length equal to the number of nodes of the graph, initially it has all its elements equal to infinite

distances[s] = 0

queue = the set of vertices of G

while queue is not empty:

u <- vertex in queue with min distances[u]

remove u from queue

for each neighbor v of u:

temp = distances[u] + value(u,v)

if temp < distances[v]:

distances[v] = temp

return distances

After running this algorithm, we get a list of distances such that distances[u] is the minimum cost to go from node s to node u.

This algorithm is guaranteed to work only if the graph doesn't have edges with negative costs. A negative cost in an edge can make the greedy strategy choose a path that is not optimal.

Another example that is used to introduce the concepts of the greedy strategy is the Fractional Knapsack.

In this problem, we have a collection of items. Each item has a weight Wi greater than zero, and a profit Pi also greater than zero.

We have a knapsack with a capacity W and we want to fill it in such a way that we get the maximum profit. Of course, we cannot exceed the capacity of the knapsack.

In the fractional version of the knapsack problem, we can take either the entire object or only a fraction of it. When taking a fraction 0 <= X <= 1 of the i-th object, we obtain a profit equal to X\*Pi and we need to add X\*Wi to the bag.

We can solve this problem by using a greedy strategy. I won't discuss the solution here. If you don't know it, I recommend that you try to solve it by yourself and then look for the solution online.

The number of problems that we can solve by using greedy algorithms is huge. But the number of problems that we cannot solve this way is even bigger. The next section is about the latter problems - those we shouldn't solve this way.

## When being greedy is the worst

In the previous section, we saw two examples of problems that are solvable using a greedy strategy. This is great because those are pretty fast algorithms.

But, as I said, Dijkstra's algorithm doesn't work in graphs with negative edges.

And the problem is even bigger. I can always build a graph with negative edges in a way that Dijkstra's solution would be as bad as I wanted! Consider the following example that was extracted from [Stackoverflow](https://stackoverflow.com/questions/6799172/negative-weights-using-dijkstras-algorithm/6799344#6799344)

Diagram

Description automatically generated

Dijkstra's algorithm fails to find the distance between A and C. It finds d(A, C) = 0 when it should be -200. And if we decrease the value of the edge D -> B, we'll obtain a distance that we'll be even farther from the actual minimum distance.

Similarly, when we can't break objects in the knapsack problem (the 0-1 Knapsack Problem), the solution that we obtain when using a greedy strategy can be pretty bad, too. We can always build an input to the problem that makes the greedy algorithm fail badly.

Another example is the Travelling Salesman Problem (TSP). Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

We can greedily approach the problem by always going to the nearest possible city. We select any of the cities as the first one and apply that strategy.

As happened in previous examples, we can always build a disposition of the cities in a way that the greedy strategy finds the worst possible solution.

In this section, we have seen that a greedy strategy could lead us to disaster. But there are problems in which such an approach can approximate the optimal solution quite well.

## When being greedy is not that bad

We have seen that a greedy strategy can become as bad as we want for some problems. This means that we cannot use it to obtain the optimal solution nor even a good approximation of it.

But there are some examples in which greedy algorithms provide us with very good approximations! In these cases, the greedy approach is very useful because it tends to be cheaper and easier to implement.

The vertex cover of a graph is the minimum set of vertices such that every edge of the graph has at least one of its endpoints in the set.

This is a very hard problem. Actually, there isn't any efficient and exact solution for it. But the good news is that we can make a good approximation with a greedy algorithm.

We select any edge <u, v> from the graph, and add u and v to the set. Then, we remove all the edges that have u or v as one of their endpoints, and we repeat the previous process while the remaining graph had edges.

This might be pseudocode of the previous algorithm:

vertexCover(G):

VertexCover <- {} // empty set

E' <- edges of G

while E' is not empty:

VertexCover <- VertexCover U {u,v} where <u,v> is in E'

E' = E' - {<u, v> U edges incident to u, v}

return VertexCover

As you can see, this is a simple and relatively fast algorithm. But the best part is that the solution will always be less than or equal to two times the optimal solution! We'll never obtain a set that is bigger than two times the smaller vertex cover, no matter how the input graph was built.

I'm not going to include the demonstration of this statement in this post, but you can prove it by noticing that for every edge <u, v> that we add to the vertex cover, either u or v are in the optimal solution (that is, in the smaller vertex cover).

Many computer scientists are working to find more of these approximations. There are more examples, but I'm going to stop here.

This is an interesting and very active research field in Computer Science and Applied Mathematics. With these approximations, we can get very good solutions for very hard problems by implementing pretty simple algorithms.

## Conclusions

In this post, I gave you a shallow introduction to greedy algorithms. We saw examples of problems that can be solved using the greedy strategy. Then, I talk about some problems for which the greedy strategy is a bad option. And finally, we saw an example of a greedy algorithm that'll get you an approximated solution to a hard problem.

Sometimes we can solve a problem using a greedy approach but it is hard to come up with the right strategy. And demonstrating the correctness of greedy algorithms (for exact or approximated solutions) can be very difficult. So, there are a lot of things we can discuss about greedy algorithms!