GRAPH DATA STRUCTURE AND ALGORITHMS - 3

Shortest Path Problem

The shortest path problem is about finding a path between vertices in a graph such that the total sum of the edges weights is minimum.

This problem could be solved with different algorithms:

Some Algorithms for the Shortest Path Problem

Single-Source Shortest Paths (SSSP)

- DFS (backtracking) (Exponential)
- **BFS (unweighted graph - dynamic programming) O(V + E)**
- D´Esopo-Pape algorithm (dynamic programming, input sensitive)
- **Dijkstra (greedy) O(Elog V) (No negative edge)**
- Dial's Algorithm (greedy) O(E + WV)
- **Bellman-Ford (dynamic programming) O(VE)**

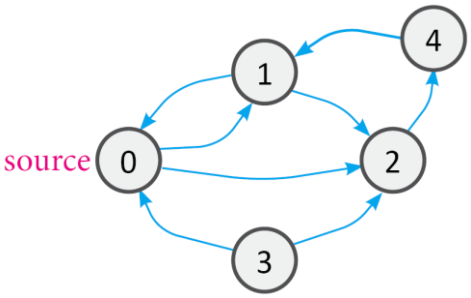
All-Pairs Shortest Paths (APSP)

- **Floyd-Warshall (dynamic programming) O(V^3)**
- Johnson's Algorithm (dynamic programming + greedy) O(V^2log V + VE)

Shortest Paths in an Unweighted Graph (BFS)

Since the graph is unweighted, we can solve this problem in O(V + E) time. Because in BFS, the first time a node is discovered during the traversal, that distance from the source would give us the shortest path.

The idea is to use a modified version of Breadth-first search in which we keep storing the **predecessor (parent)** and **distance** of a given vertex while doing the breadth first search.



Sample Input

```
5 8
3 0
0 2
4 1
2 4
1 2
3 2
0 1
1 0
```

Sample Output

```
Distances from the node 0
0: 0
1: 1
2: 1
3: not reachable
4: 2

Parents
0: 0
1: 0
2: 0
3: -1
4: 2
```

```
// BFS Shortest Path Unweighted DiGraph

#include <fstream>
```

```
#include <vector>
#include <queue>

#define BIGNUM INT_MAX

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, M;
vector<vector<int>> adjList;
vector<int> distances;
vector<int> parents;
int source = 0;
//-----------------------------------
//Create the adjacency list
void readInput()
{
	cin >> N >> M;
	adjList.resize(N);
	for (int i = 0; i < M; i++)
	{
		int u, v;
		cin >> u >> v;
		adjList[u].push_back(v);
	}
}
//-----------------------------------
//Find all shortest paths from the soruce node
//to all other nodes.
void go()
{
	distances.resize(N, BIGNUM);
	parents.resize(N, -1);
	queue<int> q; //node id
	distances[source] = 0;
	parents[source] = source;

	q.push(source);

	while (!q.empty())
	{
		int u = q.front();
		q.pop();
		for (int v : adjList[u])
		{
			if (distances[v] == BIGNUM)
			{
				distances[v] = distances[u] + 1;
				parents[v] = u;
				q.push(v);
			}
		}
	}
}
//-----------------------------------
void printDistances()
{
	cout << "Distances from the node " <<source<<endl;
	for (int i = 0; i < N; i++)
		if (distances[i] < BIGNUM)
			cout << i << ": " << distances[i] << endl;
		else
			cout << i<< ": not reachable" << endl;
	cout << endl;
}
//-----------------------------------
void printParents()
{
	cout << "Parents" << endl;
```

```
        for (int i = 0; i < N; i++)
                cout << i << ": " << parents[i] << endl;
}
//------------------------------------
int main()
{
    readInput();
        go();
        printDistances();
        printParents();
}
```

Dijkstra's Algorithm

Dijkstra's Algorithm allows you to calculate the shortest path between one node (source) and every other node in the graph.

Dijkstra's algorithm is very similar to **Prim's algorithm** for minimum spanning tree. 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, and 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.

Dijkstra runs in **O(ElogV)** time.

Data Structures

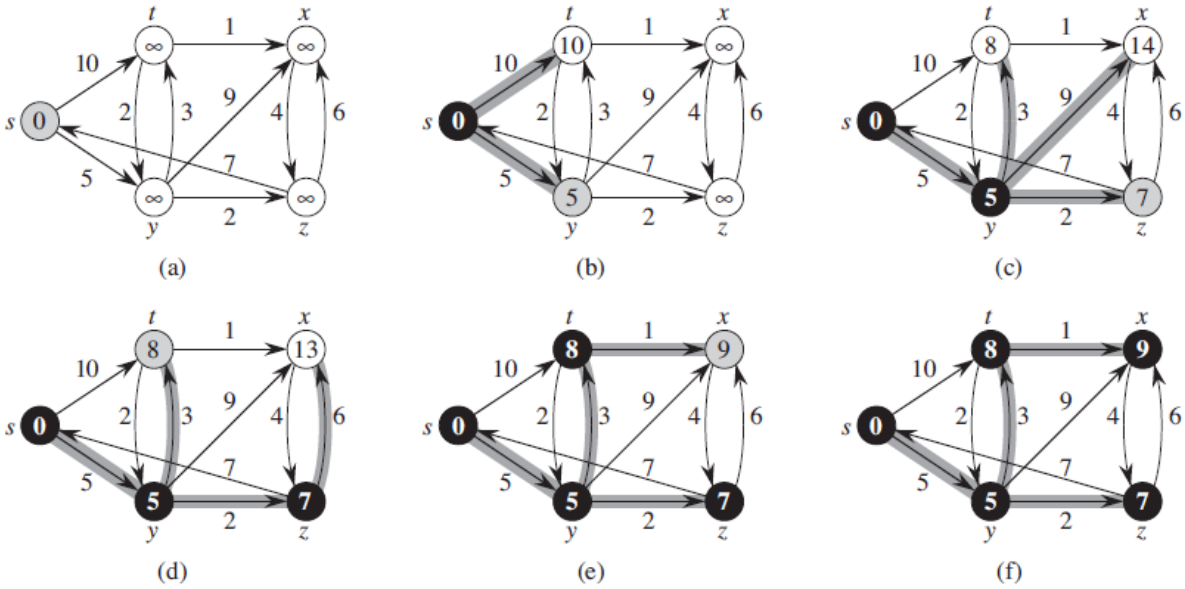
- Adjacency list
- Distance array (to track shortest distances)
- Priority Queue (Min heap to pick the next closest node)
- Parent (or Prev) array (to print the shortest paths)

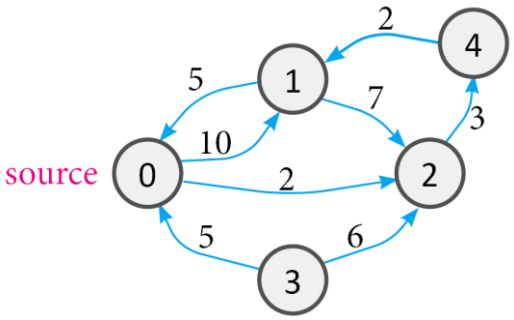
Algorithm

- Initialize distances to infinitive. Set the distance of the source node to zero. Push the source node into the priority queue.

  While Priority Queue is not empty

  - Pick next node with minimal distance (priority queue); update distances to adjacent nodes and parents.
  - Push adjacent nodes into the priority queue.





Sample Input

```
5 8
3 0 5
0 2 2
4 1 2
2 4 3
1 2 7
3 2 6
0 1 10
1 0 5
```

Sample Output

```
Distances from node 0
0: 0
1: 7
2: 2
3: not reachable
4: 5

Parents
0: 0
1: 4
2: 0
3: -1
4: 2
```

```cpp
// Dijkstra Directed Graph.cpp :
#include <fstream>
#include <vector>
#include <queue>

#define BIGNUM INT_MAX

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, M;
//first is index, second is distance
vector < vector<pair<int, int>>> adjList;
vector<int> distances, parents;
int source = 0;
//----------------------------------------
void readInput()
{
	cin >> N >> M;
	adjList.resize(N);
	for (int i = 0; i < M; i++)
	{
		int u, v, c;
		cin >> u >> v >> c;
		adjList[u].push_back({ v,c });
	}
}
//----------------------------------------
```

```
void go()
{
	distances.resize(N, BIGNUM);
	distances[source] = 0;
	parents.resize(N, -1);
	parents[source] = source;


	//first is cost, second is index, min heap
	priority_queue<pair<int, int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;
	pq.push({ 0, source });

	while (!pq.empty())
	{
		pair<int, int> cur = pq.top();
		pq.pop();

		int u = cur.second;  //current node
		int d = cur.first;   //current distance

		if (d > distances[u]) //There is a shorter path
			continue;

		for (auto p : adjList[u])
		{
			int v = p.first;
			int d2 = p.second;
			if (distances[v] > d + d2)
			{
				distances[v] = d + d2;
				parents[v] = u;
				pq.push({ distances[v], v });
			}
		}
	}
}
//----------------------------------------
void printDistances()
{
	cout << "Distances from the node " <<source<<endl;
	for (int i = 0; i < N; i++)
		if (distances[i] < BIGNUM)
			cout << i << ": " << distances[i] << endl;
		else
			cout << i<< ": not reachable" << endl;
	cout << endl;
}
//-----------------------------------
void printParents()
{
	cout << "Parents" << endl;
	for (int i = 0; i < N; i++)
		cout << i << ": " << parents[i] << endl;
}
//----------------------------------------
int main()
{
	readInput();
	go();
	printDistances()
	printParents();
}
```

Bellman-Ford Algorithm

Bellman-Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph. It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have **negative weights**. It is slower than Dijkstra's Algorithm but more versatile, as it capable of handling some of the negative weight edges.

Bellman-Ford runs in **O(VE)** time.

The main idea is to relax all the edges exactly n - 1 times. There can be maximum n-1 vertices in a shortest path.

The steps involved are:

- The outer loop traverses from 0 to N-1
- Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

This algorithm depends on the **relaxation** principle where the shortest distance for all vertices is gradually replaced by more accurate values until eventually reaching the optimum solution. In the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0, then update all the connected vertices with the new distances (source vertex distance + edge weights), then apply the same concept for the new vertices with new distances and so on.

Shortest Distances When There is no Negative Cycle

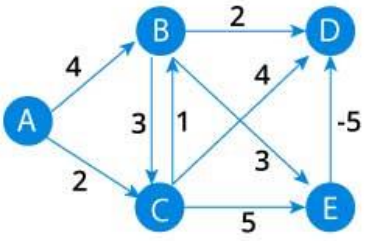
Print all shortest distances from the source node if there is no negative cycle, otherwise report the negative cycle.

Pseudocode

```
Bellman-Ford(int v)

	d[i] = inf for each vertex i
	d[v] = 0

	//n-1 relaxations of all edges to get min distances
	for step = 1 to n-1
		for all edges like e
			i = e.first   // first end, from
			j = e.second // second end, to
			w = e.weight
			if d[j] > d[i] + w
				d[j] = d[i] + w

	//One more relaxations of all edges to
	//test if there exist a negative cycle.
	for all edges like e
		i = e.first // first end
		j = e.second // second end
		w = e.weight
		if d[j] > d[i] + w
			Error: Negative Cycle Exists
```

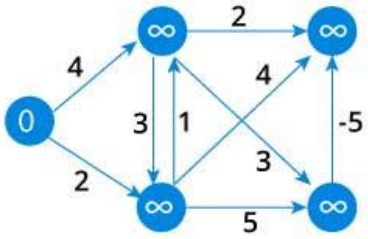
1

Start with a weighted graph



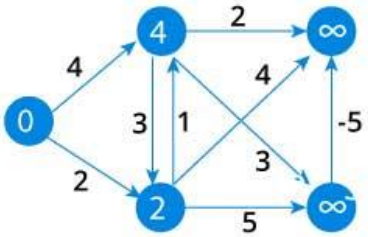
2

Choose a starting vertex and assign infinity path values to all other vertices



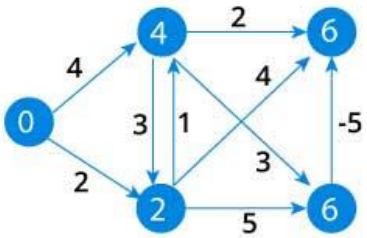
3

Visit each edge and relax the path distances if they are inaccurate



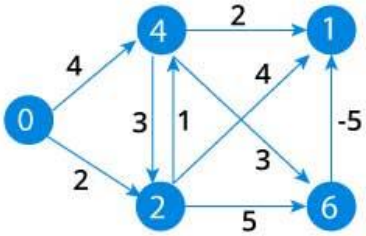
4

We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



5

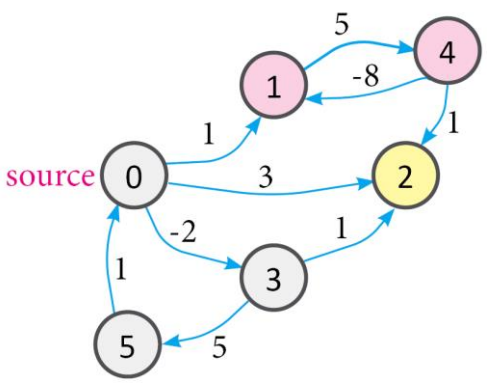
Notice how the vertex at the top right corner had its path length adjusted



6

After all the vertices have their path lengths, we check if a negative cycle is present.

A	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6



Sample Input

```
6 9
4 2 1
0 1 1
4 1 -8
3 2 1
0 2 3
3 5 5
1 4 5
5 0 1
0 3 -2
```

Sample Output

```
Negative cycle detected.
```

```
// Bellman-Ford Directed Graph.cpp:
#include <fstream>
#include <vector>

#define BIGNUM INT_MAX/2

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

struct Edge
{
	int from, to, cost;
};
int N, M;
vector<Edge> edgeList;
vector<int> distances, parents;
int source = 0;
//---------------------------------------
void readInput()
{
	cin >> N >> M;

	for (int i = 0; i < M; i++)
	{
		int from, to, cost;
		cin >> from >> to >> cost;
		edgeList.push_back({ from, to, cost });
	}
}
//---------------------------------------
bool go()
{
	distances.resize(N, BIGNUM);
	distances[source] = 0;
	parents.resize(N, -1);
	parents[source] = source;

	for (int i = 0; i < N-1 ; i++)
		for (Edge e : edgeList)
			//Relaxation
			if (distances[e.to] > distances[e.from] + e.cost)
			{
				distances[e.to] = distances[e.from] + e.cost;
				parents[e.to] = e.from;
			}

	//Detect a negative cycle. If found return false.
	for (Edge e : edgeList)
		if (distances[e.to] > distances[e.from] + e.cost)
			return false;

	return true;
}
//---------------------------------------
void printDistances()
{
	cout << "Distances from node " << source << endl;
	for (int i = 0; i < N; i++)
			cout << i << ": " << distances[i] << endl;
	cout << endl;
}
//---------------------------------------
void printParents()
{
	cout << "Parents" << endl;
	for (int i = 0; i < N; i++)
```

```
            cout << i << ": " << parents[i] << endl;
}
//--------------------------------------
int main()
{
        readInput();
        if (go())
        {
                printDistances();
                printParents();
        }
        else
                cout << "Negative cycle detected." << endl;
}
```

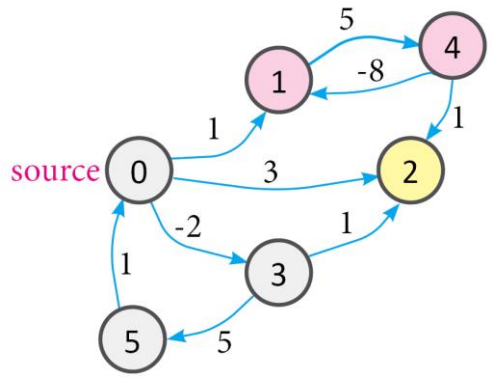
Shortest Distances to the Valid Nodes (Only in a Directed Graph)

We can slightly modify the previous implementation to calculate shortest distances from the source node to the nodes they are not in a negative cycle or not reachable from a negative cycle.

Making n-1 times relaxations of all edges calculates all shortest distances. We make other n-1 relaxations of all edges and if there are some nodes their distances getting smaller, we mark them as invalid nodes. They are either in a negative cycle or reachable from a negative cycle. In both cases their distances will continue getting decreasing forever.

Pseudocode

```
Bellman-Ford(int v)

        d[i] = inf for each vertex i
        d[v] = 0

        //n-1 relaxations of all edges to get min distances
        for step = 1 to n-1
                for all edges like e
                        i = e.first   //first end, from
                        j = e.second //second end, to
                        w = e.weight
                        if d[j] > d[i] + w
                                d[j] = d[i] + w

        //Other n-1 relaxations of all edges to mark nodes
        //that are in negative cycles or effected from a negative cycle.
        for step = 1 to n-1
                for all edges like e
                        i = e.first   //first end
                        j = e.second  //second end
                        w = e.weight
                        if d[j] > d[i] + w
                                d[j] = -inf   //invalid node
```



Sample Input

```
6 9
4 2 1
0 1 1
4 1 -8
3 2 1
0 2 3
3 5 5
1 4 5
5 0 1
0 3 -2
```

Sample Output

```
Distances from node 0
0: 0
1: invalid node
2: invalid node
3: -2
4: invalid node
5: 3

Parents
0: 0
1: -1
2: -1
3: 0
4: -1
5: 3
```

```
// Bellman-Ford Directed Graph.cpp:
#include <fstream>
#include <vector>

//BIGNUM should be bigger than abs(E*V*minCost).
#define BIGNUM INT_MAX/2

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

struct Edge
{
	int from, to, cost;
};
int N, M;
vector<Edge> edgeList;
vector<int> distances, parents;
int source = 0;
//--------------------------------------
void readInput()
{
	cin >> N >> M;

	for (int i = 0; i < M; i++)
	{
		int from, to, cost;
		cin >> from >> to >> cost;
		edgeList.push_back({ from, to, cost });
	}
}
//--------------------------------------
void go()
{
	distances.resize(N, BIGNUM);
	distances[source] = 0;
	parents.resize(N, -1);
	parents[source] = source;

	for (int i = 0; i < N - 1; i++)
		for (Edge e : edgeList)
			//Relaxation
			if (distances[e.to] > distances[e.from] + e.cost)
			{
				distances[e.to] = distances[e.from] + e.cost;
				parents[e.to] = e.from;
			}

	//Dedect the nodes in a negative cycle and
	//the nodes that are reachable from a negative cycle.
	for (int i = 0; i < N - 1; i++)
		for (Edge e : edgeList)
			if (distances[e.to] > distances[e.from] + e.cost)
```

```
                    {
                            distances[e.to] = -BIGNUM;   //marking an invalid node.
                            parents[e.to] = -1;          //set its parent as -1.
                    }
}
//-------------------------------------
void printDistances()
{
        cout << "Distances from node " << source << endl;
        for (int i = 0; i < N; i++)
                if (distances[i] > -BIGNUM)
                        cout << i << ": " << distances[i] << endl;
                else
                        cout << i << ": invalide node" << endl;
        cout << endl;
}
//-------------------------------------
void printParents()
{
        cout << "Parents" << endl;
        for (int i = 0; i < N; i++)
                cout << i << ": " << parents[i] << endl;
}
//-------------------------------------
int main()
{
        readInput();
        go();
        printDistances();
        printParents();
}
```

Floyd-Warshall All-Pairs Shortest Paths (APSP)

Floyd-Warshall computes the shortest distances between every pair of vertices in a weighted graph with positive or negative edge weights (but with no negative cycles).

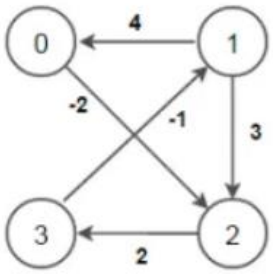
The Floyd-Warshall algorithm is an example of **dynamic programming**. It breaks the problem down into smaller sub-problems, and then combines the answers to those sub-problems to solve the big, initial problem. The idea is this: either the quickest path from *i* to *j* is the quickest path found so far from *i* to *j*, or it's the quickest path from *i* to *k* plus the quickest path from *k* to *j*.

shortestPath(i, j) = min(shortestPath(i, j), shortestPath(I ,k) + shortestPath(k, j) for all k, i, j

Floyd-Washall represent the graph with an **adjacency matrix** (distance matrix). Initialize the distance matrix same as the input graph matrix as a first step. Then update the distance matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. There are N intermediate vertices to pick and N*N pairs of vertices, thus Folyd-Warshall runs in **$O(V^3)$ time and $O(V^2)$ space**.

Running **the Dijkstra's algorithm V times** calculates all shortest paths in **O(VElogV)** time. It runs faster than Floyd-Warshall in a **sparse graph**.

```
1 let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
2 for each edge (u,v)
3    dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
4 for each vertex v
5    dist[v][v] ← 0
6 for k from 1 to |V|
7    for i from 1 to |V|
8       for j from 1 to |V|
9          if dist[i][j] > dist[i][k] + dist[k][j]
10             dist[i][j] ← dist[i][k] + dist[k][j]
11         end if
```




```
// Floyd-Warshall.cpp : All-pairs shortest paths.
// Report if there is a negative cycle.
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>

#define INF INT_MAX/2

using namespace std;

int N = 4, M = 5;
//the input graph
vector<vector<int>> dist = { { 0,   INF, -2,  INF },
                             { 4,   0,   3,   INF },
                             { INF, INF, 0,   2   },
                             { INF, -1,  INF, 0   }
                           };
//--------------------------------------------------------
void printDistances()
{
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			if (dist[i][j] == INF)
				cout << left << setw(5) << "inf";
			else
				cout << left << setw(5) << dist[i][j];
		}
		cout << endl;
	}
	cout << endl;
}
//--------------------------------------------------------
bool floydWarshall()
{
	for (int k = 0; k < N; k++)
		for (int i = 0; i < N; i++)
			for (int j = 0; j < N; j++)
			{
				dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);

				if (i == j && dist[i][j] < 0) //negative cycle
					return false;
			}
```

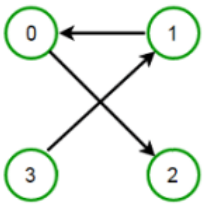
```
	return true;
}
//--------------------------------------------------------
int main()
{
	cout << "Initial distances:" << endl;
	printDistances();

	if (floydWarshall())
	{
		cout << "Shortest distances" << endl;
		printDistances();
	}
	else
		cout << "Negative cycle detected." << endl;
}
```

Output

```
Initial distances:
0    inf  -2   inf
4    0    3    inf
inf  inf  0    2
inf  -1   inf  0

Shortest distances
0    -1   -2   0
4    0    2    4
5    1    0    2
3    -1   1    0
```

Transitive Closure of a Graph

Given a directed graph, find out if a vertex *v* is reachable from another vertex *u* for all vertex pairs (*u*, *v*) in the given graph. Here reachable means that there is a path from vertex *u* to *v*. The **reachability matrix** is called transitive closure of a graph. A modified Floyd-Warshall algorithm construct the Boolean reachability matrix in $O(V^3)$ time.



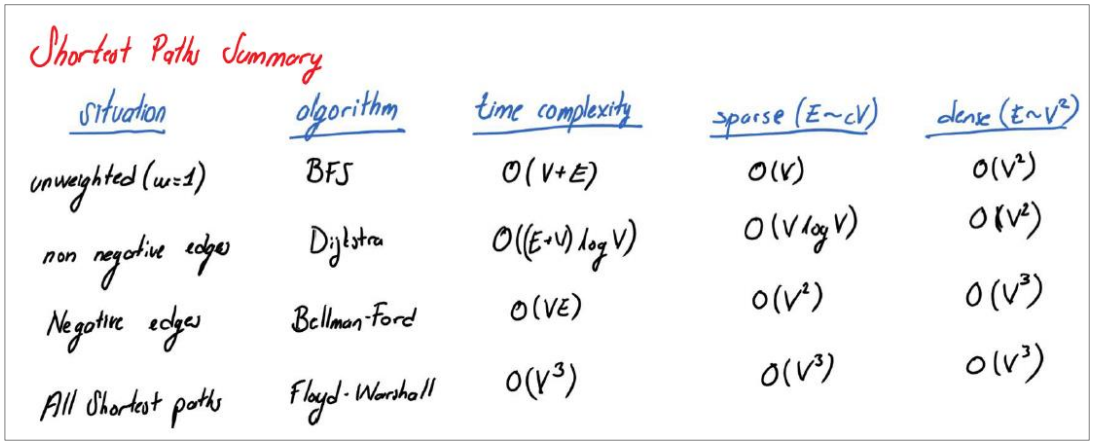
1 0 1 0
1 1 1 0
0 0 1 0
1 1 1 1

```
// TransitiveClosure.cpp :
#include <iostream>
#include <vector>

using namespace std;

int N = 4;
vector<vector<bool>> G = {   {1, 0, 1, 0},
                             {1, 1, 0, 0},
                             {0, 0, 1, 0},
                             {0, 1, 0, 1}
                         };
//-----------------------------------------------
//Reach matrix of G
void go()
{
	for (int k = 0; k < N; k++)
```

```
		for (int i = 0; i < N; i++)
		{
			if (!G[i][k]) continue; //skip next for
			for (int j = 0; j < N; j++)
				G[i][j] = G[i][j] || (G[i][k] && G[k][j]);
		}
}
//---------------------------------------------
void printG()
{
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
			cout << G[i][j] << " ";
		cout << endl;
	}

}
//---------------------------------------------
int main()
{
	go();
	printG();
	return 0;
}
```

Output

```
1 0 1 0
1 1 1 0
0 0 1 0
1 1 1 1
```



Shortest and Longest Paths in a DAG

Single Source Shortest Paths in a DAG

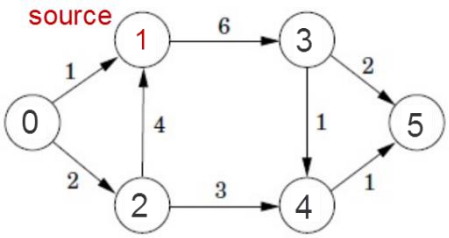
We saw how to find the shortest path in a graph with positive edges using the *Dijkstra's algorithm*. We also know how to find the shortest paths from a given source node to all other nodes even when there are negative edges using the *Bellman-Ford algorithm*. Now we'll see that there's a faster algorithm running in linear time that can find the shortest paths from a given source node to all other reachable vertices in a directed acyclic graph, also known as a DAG.

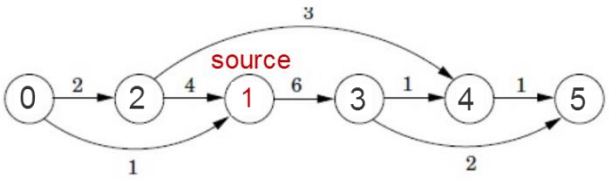
Because the DAG is acyclic we don't have to worry about negative cycles.

Algorithm:

- Find topological ordering of the DAG;
- Set the distance to the source to 0 and infinity to all other vertices
- For each vertex from the list starting from the source node pass through all its neighbors and update the shortest distances.

It's pretty much like the Dijkstra's algorithm with the main difference that in Dijkstra we used a priority queue; while this time we use the list from the topological sort.





Sample Input

```
6 8
0 1 1
1 3 6
3 4 1
2 4 3
3 5 2
0 2 2
2 1 4
4 5 1
```

Sample Output

```
Distances: INF 0 INF 6 7 8
Parents: none 1 none 1 3 3
```

Code:

```
// Shortest Paths in DAG.cpp:

#include <fstream>
#include <vector>
#include <queue>
#include <algorithm>

#define INF 99

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, M;
vector < vector<pair<int, int>>> adjList;
vector<int> topSortVec;
vector<int> distances, inDegrees, parents;
int source;
//-----------------------------------------
void readInput()
{
	cin >> N >> M;
	adjList.resize(N);
	inDegrees.resize(N, 0);
	for (int i = 0; i < M; i++)
	{
		int u, v, c;
		cin >> u >> v >> c;
		adjList[u].push_back({ v,c });
		inDegrees[v]++;
	}
```

```
}
//----------------------------------------
//Find one topological sort of DAG
void topSort()
{
	queue<int> q;
	//start with the nodes that have 0 indegree.
	for (int i = 0; i < N; i++)
		if (inDegrees[i] == 0)
			q.push(i);

	while (!q.empty())
	{
		int u = q.front();
		q.pop();

		topSortVec.push_back(u);

		for (auto p : adjList[u])
		{
			int v = p.first;
			inDegrees[v]--;
			if (inDegrees[v] == 0)
				q.push(v);
		}
	}
}
//----------------------------------------
void go()
{
	distances.resize(N, INF);
	parents.resize(N, -1);
	distances[source] = 0;
	parents[source] = source;

	//Locate index of source in topSortVec.
	int startIndex = find(topSortVec.begin(), topSortVec.end(), source) - topSortVec.begin();

	//Process nodes starting from the source node.
	for (int i = startIndex; i < N; i++)
	{
		int u = topSortVec[i];
		for (auto p : adjList[u])
		{
			int v = p.first;
			int cost = p.second;
			if (distances[v] > distances[u] + cost) //be careful for integer overflow
			{
				distances[v] = distances[u] + cost;
				parents[v] = u;
			}
		}
	}
}
//----------------------------------------
void printDistances()
{
	cout << "Distances: ";
	for (auto d : distances)
		d == INF ? (cout << "INF ") : (cout << d << " ");
	cout << endl;
}
//----------------------------------------
void printParents()
{
	cout << "Parents: ";
	for (auto p : parents)
		p == -1 ? (cout<< "none ") : (cout << p << " ");
}
//----------------------------------------
int main()
```

```
{
        readInput();
        topSort();
        source = 1;
        go();
        printDistances();
        printParents();
}
```

Single Source Longest Paths in a DAG

Given a Weighted Directed Acyclic Graph (DAG) and a source vertex s in it, find the longest distances from s to all other vertices in the given graph.

The longest path problem for a general graph is not as easy as the shortest path problem it is NP-Hard for a general graph. However, the longest path problem has a linear time solution for directed acyclic graphs. The idea is similar to linear time solution for shortest path in a directed acyclic graph. Set the initial distances to negative infinitive (NINF) and update the distances if the new distance is bigger than the current distance.

Pseudocode:

```
1) Initialize dist[] = {NINF, NINF, ….} and dist[s] = 0 where s is the source vertex.
   (Here NINF means negative infinite.)
2) Create a topological order of all vertices.
3) Do following for every vertex u in topological order.
       Do following for every adjacent vertex v of u
              if (dist[v] < dist[u] + weight(u, v))
                     dist[v] = dist[u] + weight(u, v)
```

Practice Problems

1. **0-1 BFS**. Find all shortest paths from a source vertex in a binary weighted graph using the BFS algorithm. (*Hint: Use a Deque or two queues.)*
2. **Vertex Weighted Graph**. How can you calculate shortest paths in a graph where vertices have cost? (*Hint: Use split vertex technique*)
3. **Printing Path**. Make a function to print the path from the source note to a target node using BFS, Dijkstra and Bellman-Ford shortest paths algorithms.
4. **Cycle Detection**. How can you detect a cycle (or a negative cycle) with Floyd-Warshall or BFS algorithm?
5. **Stop Earlier**. If we are interested with the shortest path from the source node to a target node. Can we stop the process as soon as we visit target node? Answer this question for BFS, Dijkstra, Bellman-Ford and Floyd-Warshall algorithms.
6. **Paths in Floyd-Warshall**. Print the path between any pair of vertices in Floyd-Warshall algorithm. (*Hint: Use a path matrix to keep the parent information. You can consider each row as a path array of a vertex. Update the path matrix any time you update the distance matrix.*)
7. **Transitive Closure in O(V(V+E)) Time**. Construct the reachability matrix of transitive closure in a quadratic time. This algorithm is preferable for sparse graphs. (*Hint: Call DFS (or BFS) for every vertex.*)

8. **Shortest Paths in a DAG with Bellman-Ford.** Calculate the single source shortest paths in a DAG using the Bellman-Fort relaxations in O(E+V) time. (*Hint: Topological sorting + one pass of Bellman-Ford relaxations.*)
9. **Transitive Reduction**. A transitive reduction of a digraph G is another digraph with the same vertices and as few edges as possible, such that if there is a (directed) path from vertex v to vertex w in G, then there is also such a path in the reduction. Modify the transitive closure algorithm to construct transitive reduction matrix.
10. **Number of Shortest Paths in a DAG**. Calculate number of shortest paths from a source node to each of other notes in a DAG.
11. **Shortest Paths in a DAG with BFS.** Find all shortest paths from a source node to all nodes with BFS in a DAG without pre-calculating the topological ordering. Compare the time complexity of this algorithm with the one with topological sorting.
12. **Longest Paths in a DAG.** Find all longest paths from a source node to all nodes in a DAG using the single source shortest paths algorithm. (*Hint: Negate the weights and find the shortest paths.*)
13. **Shortest and Longest Paths in a Tree**. Find the shortest and longest path between the node *u* and node *v* in a tree. (Hint: There is a unique path between a pair of nodes in a tree. BFS finds the path in O(E+V) time. With some pre-calculation and using Lowest Common Ancestor (LCA) of *u* and *v* it can be calculated in O(H) (height of the tree) time, or in O(VlogV), or in O(V) time.

Useful Links and References

https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/
https://www.wikizeroo.org/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvRGlqa3N0cmEnc19hbGdvcml0aG0
http://www.codebytes.in/2014/08/dijkstras-algorithm-implementation-in.html
https://www.geeksforgeeks.org/shortest-path-unweighted-graph/
https://www.programiz.com/dsa/bellman-ford-algorithm
https://www.javatpoint.com/bellman-ford-algorithm
https://iq.opengenus.org/bellman-ford-algorithm/#algorithm
https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
https://brilliant.org/wiki/floyd-warshall-algorithm/
https://www.techiedelight.com/pairs-shortest-paths-floyd-warshall-algorithm/
https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/
https://www-m9.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_en.html
https://www.geeksforgeeks.org/transitive-closure-of-a-graph/
https://www.techiedelight.com/transitive-closure-graph/
https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/
https://allaboutalgorithms.wordpress.com/2011/11/02/shortest-path-in-directed-acyclic-graphs/
https://dzone.com/articles/algorithm-week-shorted-path
https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/