

BFS

Time	$ V + E $
Space	$ V $

Data Structures

Name	Type	Initial Value
front	Queue<Vertex>	[start]
seen	Set<Vertex>	{start}

Algorithm

```
while (!front.empty()) {
  Vertex u = front.top();
  front.pop();

  // Visit u

  for (Vertex v : E[u]) {
    if (seen.has(v)) continue;
    seen.add(v);

    // See u → v

    front.push(v);
  }
}
```

Results

- seen is the set of vertices connected to start.

Prim's Algorithm

Time	$(E + V) \log V $
Space	$ V ^2$

Data Structures

Name	Type	Initial Value
front	PriorityQueue<(Weight, Vertex)>	[(0, start)]
visited	Set<Vertex>	{}
parent	Map<Vertex, Vertex>	{}
dist	Map<Vertex, Weight>	{}
tree	Map<Vertex, List<(Weight, Vertex)>>	{}

Algorithm

```
while (!front.empty()) {
    (Weight w, Vertex u) = front.top();
    front.pop();

    if (visited.has(u)) continue;
    visited.add(u);

    // Visit u

    if (parent.has(u)) {
        tree[u].push((w, parent[u]));
        tree[parent[u]].push((w, u));

        // Connect parent[u] to u
    }

    for ((Vertex v, Weight x) : E[u]) {
        if (!dist.has(v) || dist[v] > x) {
            dist[v] = x;
            parent[v] = u;

            // Relax u → v

            front.push((x, v));
        }
    }
}
```

Results

- `tree` is **some** MST of `start`'s connected component.

Notes

- In C++, `set` can be used as a `PriorityQueue` with an efficient `update_priority`.

DFS

Time	$ V + E $
Space	$ V $

Data Structures

Name	Type	Initial Value
backtrack	Stack<Vertex>	[start]
visited	Set<Vertex>	{}

Algorithm

```
while (!backtrack.empty()) {
  Vertex u = backtrack.top();

  if (!visited.has(u)) {
    visited.add(u);

    // Start visiting u
  } else {
    // Backtrack to u
  }

  bool follow = false;
  for (Vertex v : E[u]) {
    if (visited.has(v)) continue;

    // Follow u → v

    backtrack.push(v);
    follow = true;
    break;
  }
  if (follow) continue;

  // Finish visiting u

  backtrack.pop();
}
```

Results

- `visited` is the set of vertices connected to `start`.

Floyd-Warshall

Time	$ V ^3$
Space	$ V ^2$

Data Structures

Name	Type	Initial Value
next?	Map<(Vertex, Vertex), Vertex>	{E(u, v): v}
dist	Map<(Vertex, Vertex), Distance>	{E(u, v): w, V(v, v): 0}

Algorithm

```
for (Vertex m : V) {
  for (Vertex u : V) {
    for (Vertex v : V) {
      if (!dist.has((u, m)) || !dist.has((m, v))) continue;

      if (dist[u, v] > dist[u, m] + dist[m, v]) {
        dist[u, v] = dist[u, m] + dist[m, v];
        next[u, v] = next[u, m];

        // Relax u → v through m
      }
    }
  }
}
```

Results

- `dist[u, v]` is the distance from `u` to `v` (if they are connected).
- `next[u, v]` is the second vertex on **some** shortest path from `u` to `v` (if they are connected and distinct).

Notes

- Johnson’s Algorithm is faster for sparse graphs.
- Fails on graphs with negative cycles.

Bellman-Ford

Time	$ V \cdot E $
Space	$ V $

Data Structures

Name	Type	Initial Value
prev?	Map<Vertex, Vertex>	{}
dist	Map<Vertex, Distance>	{start: 0}

Algorithm

```
for (|V| - 1) {
  for ((Vertex u, Vertex v, Weight w) : E) {
    if (!dist.has(u)) continue;

    if (!dist.has(v) || dist[v] > dist[u] + w) {
      dist[v] = dist[u] + w;
      prev[v] = u;

      // Relax u → v
    }
  }
}

// Extra iteration
for ((Vertex u, Vertex v, Weight w) : E) {
  if (dist.has(u) && dist[v] > dist[u] + w) {
    return false; // Negative cycle detected
  }
}

return true;
```

Results

- `dist[v]` is the distance from `start` to `v` (if they are connected).
- `prev[v]` is the penultimate vertex on **some** shortest path from `start` to `v` (if they are connected).

Notes

- The extra iteration will relax some vertex iff a negative cycle is reachable from `start`.
- $|V| - 1$ extra iterations will relax `v` iff there is a negative cycle between `start` and `v`.

Johnson's Algorithm

Time	$(E + V) V \log V $
Space	$ V ^2$

Data Structures

Name	Type	Initial Value
adjusted	Map<Vertex, List<(Weight, Vertex)>>	G + {q: [V(0, v)]}
prev?	Map<Vertex, Map<Vertex, Vertex>>	{}
dist	Map<Vertex, Map<Vertex, Distance>>	{}

Algorithm

```
Map<Vertex, Distance> height = BellmanFord(adjusted, q);
adjusted.remove(q);

// Reweighting
for (Vertex u : V) {
    for ((Weight w, Vertex v) : adjusted[u]) {
        w += height[u] - height[v];
    }
}

// Repeated Dijkstra
for (Vertex v : V) {
    (dist[v], prev[v]) = Dijkstra(adjusted, v);
}
```

Results

- `dist[u][v]` is the distance from `u` to `v` (if they are connected).
- `prev[u][v]` is the penultimate vertex on **some** shortest path from `u` to `v` (if they are connected).

Notes

- Bellman-Ford & reweighting can be skipped for graphs with non-negative edges.
- Can detect negative cycles during Bellman-Ford.

Dijkstra's Algorithm

Time	$(E + V) \log V $
Space	$ V ^2$

Data Structures

Name	Type	Initial Value
front	PriorityQueue<(Distance, Vertex)>	[(0, start)]
visited	Set<Vertex>	{}
prev?	Map<Vertex, Vertex>	{}
dist	Map<Vertex, Distance>	{start: 0}

Algorithm

```
while (!front.empty()) {
  (Distance d, Vertex u) = front.top();
  front.pop();

  if (visited.has(u)) continue;
  visited.add(u);

  // Visit u

  for ((Vertex v, Weight w) : E[u]) {
    Distance r = d + w;
    if (!dist.has(v) || dist[v] > r) {
      dist[v] = r;
      prev[v] = u;

      // Relax u → v

      front.push((r, v));
    }
  }
}
```

Results

- `dist[v]` is the distance from `start` to `v` (if they are connected).
- `prev[v]` is the penultimate vertex on **some** shortest path from `start` to `v` (if they are connected).

Notes

- In C++, `set` can be used as a `PriorityQueue` with an efficient `update_priority`.
- Fails on graphs with negative edges.

Topological Sort

Time	$ V + E $
Space	$ V $

Data Structures

Name	Type	Initial Value
sorted	Set<Vertex>	{}
topo	List<Vertex>	[]

Algorithm

```
for (Vertex start : V) {
    if (sorted.has(start)) continue;

    Set<Vertex> visited = {};
    Stack<Vertex> backtrack = [start];

    while (!backtrack.empty()) {
        Vertex u = backtrack.top();
        visited.add(u);

        bool follow = false;
        for (Vertex v : E[u]) {
            if (sorted.has(v)) continue;
            if (visited.has(v)) return false; // Cycle detected

            backtrack.push(v);
            follow = true;
            break;
        }
        if (follow) continue;

        sorted.add(u);
        topo.push(u);
        backtrack.pop();
    }
}

return true;
```

Results

- `i < j` implies there is no path from `topo[i]` to `topo[j]` (reverse topological order).

Notes

- Fails on graphs with cycles.

C++ Prelude

```
#include <bits/stdc++.h>
using namespace std;
#define DEBUG 1
#define dbg(x) (DEBUG ? _show((#x), (x)) : (x))
#define mod(x, m) (((x) % (m)) + (m)) % (m))
template <class T> typename
enable_if<!is_compound<typename remove_reference<T>::type>::value, string>::type _str(T& x) {
    return to_string(x); }
template <class T> typename
enable_if< is_compound<typename remove_reference<T>::type>::value, string>::type _str(T& x) {
    stringstream s;
    auto b = begin(x), e = end(x);
    while (b != e) s << _str(*b++) << (b != e ? ", " : "");
    return '[' + s.str() + ']'; }
template <> string _str(string& x) { return "'" + x + "'"; }
template <class T> T& _show(string s, T&& x) { cout << s + " = " + _str(x) + '\n'; return x; }
typedef pair<int, int> ii;
typedef int64_t i64;
typedef uint64_t u64;
const double pi = 2 * acos(0.0);
const double dinf = 1.0 / 0.0;
const int inf = numeric_limits<int>::max() >> 2;
const long linf = numeric_limits<i64>::max() >> 2;
```

Java Prelude

Speed Estimates

Sample Graphs

```
10
0 0 (no vertices)
9 0 (no edges)
1 1 (self-loop)
0 0 5
2 2 (parallel edges)
0 1 0
0 1 2
3 3 (cycle)
0 1 0
1 2 2
2 0 4
4 2 (disconnected)
0 1 0
2 3 2
3 3 (non-consecutive)
9 3 0
7 3 2
7 9 4
3 3 (negative edge)
0 1 -2
1 2 4
2 0 0
3 3 (negative cycle)
0 1 -4
1 2 2
2 0 0
5 10 (complete graph)
0 1
0 2
0 3
0 4
1 2
1 3
1 4
1 5
2 3
2 4
2 5
3 4
3 5
4 5
```

Other possibilities

- Dense or sparse
- Multiple solutions (e.g., shortest paths, MSTs, topological sorts)