- Graph representation
- BFS
- DFS
- Applications
  - Cycle Detection
  - Topological Sort

Graph Representation

Definition: G = (V, E)

V = set of vertices

E = set of edges

Directed graph & Undirected graph

Two representations:

Adjacency matrix: edge test

space: $O(n^2)$, bad for sparse graph

Adjacency list: fast to traverse neighbors of a vertex

space: $O(n+m)$

How to choose efficient representation:

The type of operation

Space complexity

BFS

```
BFS(G, S) {
  Queue q;
  visited[1...n];
  q.enqueue(S);
  visited[S] = T;
  while (!q.isEmpty()) {
    v = q.dequeue();

    // do operation for different applications

    for (p in neighbor(v)) {
      if (!visited[p]) {
        q.enqueue(p);
        visited[p] = T;
      }
    }
  }
}
```

Time: $O(n+m)$

Space: $O(n+m)$

In this case it's better to use adjacency list, since we need to traverse all the neighbors of each vertex

```
BFS_ALL(G) {
  visited[1...n];
  for (s = 1...n) {
    if (!visited[s]) {
      BFS(G,s, visited);
    }
  }
}
```

Time: $O(n+m)$

NOTE: Pay attention to this time complexity.

DFS

```
DFS(G, S, visited) {
  visited[S] = T;

  // processing vertex S

  for (v in neighbor(S)) {
    if (!visited[v]) {
      DFS(G, v, visited);
    }
  }
}
```

Time: $O(n+m)$

Application

Cycle Detectoin

input: directed G = (V, E)

output: True if there is a cycle in the graph, false otherwise

Thought process: Obviously, dfs should be used. But basic dfs is not enough to solve the problem.

There is a cycle when during one bfs process, the neighbor of current vertex is visited but not finished (finished means dfs all neighbor).

Use color[1...n] to replace visited[1...n].

color[u] = N, not visited

color[u] = V, visited but not finished

color[u] = F, visited and finished

```
DFS(G, S) {
  color[S] = V;
  for (p in neighbor(S)) {
    if (color[p] = N) {
      DFS(G, p);
    } else if (color[p] = V) {
      exists cycle;
    }
  }
  color[S] = F;
}
```

Topological Sort

The definition of ***topological sort*** is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

The purpose of this algorithm is to produce a list of vertices in topological ordering.

Approach 1: Non-DFS

The idea is, u must comes before v in the ordering if there is a edge uv. Therefore, vertices without incoming edge must come before other vertices. We use arary indegree[n] to record the incoming edges of every vertices.

```
// O(n+m)
for (u = 1...n) {
  for (v in neighbors(u)) {
    indegree[v]++;
  }
}
// O(n)
for (u = 1...n) {
  if (indegree[u] == 0) {
    q.enqueue(u);
  }
}
// O(n+m)
while(!q.isEmpty()) {
  u = q.dequeue();
  // add u to the list
  for (p in neighbor(u)) {
    indegree[p]--;
    if (indegree[p] == 0) {
      q.enqueue(p);
    }
  }
}
```

Approach 2: DFS

For edge uv from u to v, the finish time of dfs(u) must be later than the finish time of dfs(v). Therefore, we can add the vertex to the list after its dfs process. Finally, reverse the whole list.

```
DFS(G, S, L) {
  visited[S] = T;
  for (u in neighbor(S)) {
    DFS(G, u, L);
  }
  L.append(S);
}

DFS_ALL(G) {
  L = [];
  for (s = 1...n) {
    if (!visited[s]) {
      DFS(G, s, L);
    }
  }
  reverse(L);
  return L;
}
```