# Graph Algorithms

## COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Sydney

Term 2, 2025

## Table of Contents

## Graphs

- A graph is a collection of *vertices* and *edges* connecting pairs of vertices.
- Generally, graphs can be thought of as abstract representations of objects and connections between those objects, e.g. intersections and roads, people and friendships, variables and equations.
- Many unexpected problems can be solved with graph techniques.

## Graphs

- Many different types of graphs
    - Directed graphs
    - Acyclic graphs (i.e: trees, forests, DAGs)
    - Weighted graphs
    - Flow graphs
    - Other labels for the vertices and edges

## Graph Representations

- Mostly you'll want to use an *adjacency list*, occasionally an *adjacency matrix* to store your graph.
- An adjacency matrix is just a table (usually implemented as an array) where the $j$th entry in the $i$th row represents the edge from $i$ to $j$, or lack thereof.
  - Useful for dense graphs or when you want to know about specific edges.
- An adjacency list is a vector for every vertex containing a list of adjacent edges.
  - Much better for traversing sparse graphs.

## Adjacency List

```
#include <iostream>
#include <vector>

int N = 1001001; // number of vertices in graph
vector<int> edges[N]; // each vertex has a list of connected vertices

void add(int u, int v) {
  edges[u].push_back(v);
  // Warning: If the graph has self-loops, you need to check this.
  if (v != u) { edges[v].push_back(u); }
}

...

// iterate over edges from u (since C++11)
for (int v : edges[u]) { cout << v << '\n'; }

// iterate over edges from u (before C++11)
vector<int>::iterator it = edges[u].begin();
for (; it != edges[u].end(); ++it) {
  int v = *it;
  cout << v << '\n';
}

// or just a regular for loop will work too
for (unsigned int i = 0; i < edges[u].size(); i++) {
    cout << edges[u][i] << '\n';
}
```

## Table of Contents

## Graph Traversals

- There are two main ways to traverse a graph, which differ in the order in which they visit new vertices:
    - Breadth-first search (BFS): visit the entire adjacency list of some vertex, then recursively visit every unvisited vertex in the adjacency list of those vertices.
    - Depth-first search (DFS): visit the first vertex in some vertex's adjacency list, and then recursively DFS on that vertex, then move on;
- Both can be implemented in $O(|V| + |E|)$ time.

## Breadth-First Search

- Visits vertices starting from $u$ in increasing distance order.
- Use this to find shortest path from $u$ to every other vertex.
- Not much other reason to use BFS over DFS.

## Breadth-First Search

- **Implementation**

```
vector<int> edges[N];
// dist from start. -1 if unreachable.
int dist[N];
// previous node on a shortest path to the start
// Useful for reconstructing shortest paths
int prev[N];

void bfs(int start) {
    fill(dist, dist+N, -1);
    dist[start] = 0;
    prev[start] = -1;

    queue<int> q;
    q.push(start);
    while (!q.empty()) {
        int c = q.front();
        q.pop();
        for (int nxt : edges[c]) {
            // Push if we have not seen it already.
            if (dist[nxt] == -1) {
                dist[nxt] = dist[c] + 1;
                prev[nxt] = c;
                q.push(nxt);
            }
        }
    }
}
```

## Depth-First Search

- Depth-first search is a simple idea that can be extended to solve a huge amount of problems.
- Basic idea: for every vertex, recurse on everything it's adjacent to that hasn't already been visited.

## Depth-First Search

- **Implementation**

```
// global arrays are initialised to zero for you
bool seen[N];

void dfs(int u) {
  if (seen[u]) { return; }
  seen[u] = true;
  for (int v : edges[u]) { dfs(v); }
}
```

## Depth-First Search

- In its simple form, it can already be used to solve several problems - undirected cycle detection, connectivity, flood fill, etc. In short, it should be your default choice for traversing a graph.
- However, its true power comes from the fact DFS has nice invariants and the tree it creates has nice structure.
- **Main Invariant:** By the time we return from a vertex $v$ in our DFS, we have visited every vertex $v$ can reach that does not require passing through an already visited vertex.

## DFS Tree

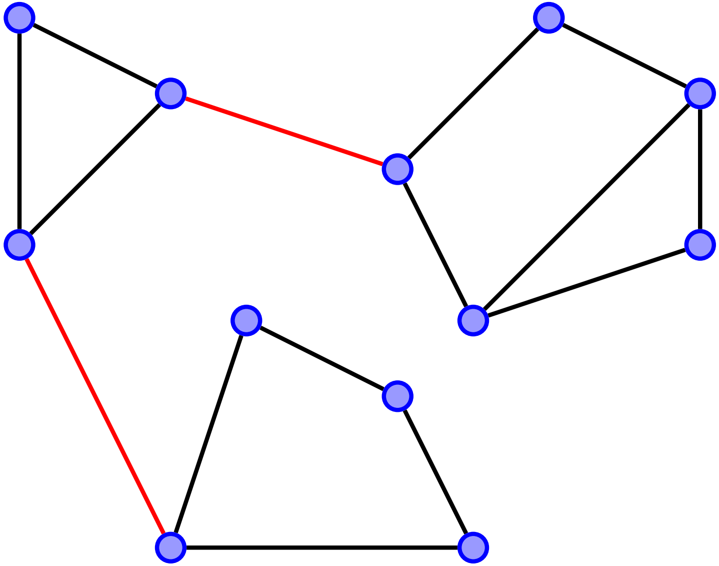
For now, we restrict ourselves to undirected graphs.

- In our DFS, if we only consider edges that visit a vertex for the first time, these edges form a tree. All other edges are called "back edges".
- See this DFS Tree Tutorial for an illustration.
- **Main Structure:** Back edges always go directly upwards to an ancestor in the DFS tree.
- A not difficult consequence of the **Main Invariant**.
- This is an abstract but really powerful tool for analyzing a graph's structure.
- Sometimes it is useful to explicitly construct this tree but often we just implicitly consider it in our DFS.

## Example problem: Bridge Finding

- **Problem statement** Consider $G$ an undirected, simple (loopless and with no multi-edges), connected graph. A bridge of $G$ is an edge $e$ whose removal disconnects $G$. Output all bridges of $G$.
- **Input**
  - First line, 2 integers $V, E$, the number of vertices and number of edges respectively.
  - Next $E$ lines, each a pair, $u_i v_i$. Guaranteed $u_i \neq v_i$ and no unordered pair appears twice.
  - $1 \leq V, E, \leq 100,000$.
- **Output** Output all bridges, each on a single line as the two vertices the bridge connects.

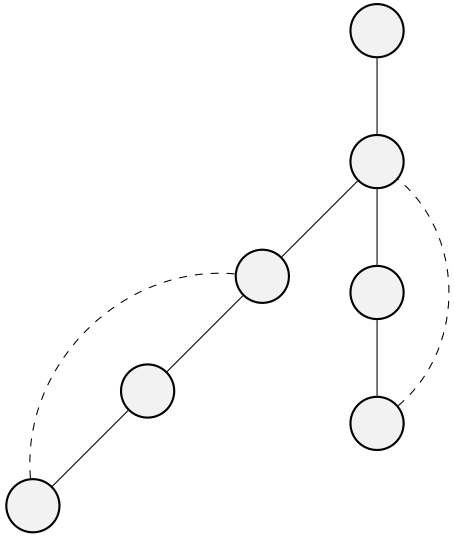
## Example problem: Bridge Finding



## Example problem: Bridge Finding

- A graph is a pretty chaotic thing.
- Let us introduce some structure by looking at the DFS tree.

## Example problem: Bridge Finding



## Example problem: Bridge Finding

- **Claim 1:** Back edges can not be bridges.
- **Claim 2:** A tree edge is a bridge iff there is no back edge going "past it".
- More formally, it is enough to know within each subtree of the DFS tree, what the highest node a back edge in this subtree can reach.
- Not hard to compute this recursively in our DFS.
- As a minor technical note: our code will use pre-order indices instead of computing depth.

# Example problem: Bridge Finding

- **Implementation**

```
int preorder[N]; // order of pushing to the DFS stack
                 // initialise to -1
int T = 0; // counter for preorder sequence
int reach[N]; // reach[u] is the smallest preorder index
            // of any vertex reachable from u
vector<pair<int,int>> bridges;

void dfs(int u, int from = -1) {
  preorder[u] = T++;
  reach[u] = preorder[u];

  for (int v : edges[u]) if (v != from) {
    if (preorder[v] == -1) { // u--v is a tree edge
      // v hasn't been seen before, so recurse
      dfs(v, u);
      // if v can't reach anything earlier than itself
      // then u--v is a bridge
      if (reach[v] == preorder[v]) { bridges.emplace_back(u,v); }
    }
    // anything reachable from v is reachable from u
    reach[u] = min(reach[u], reach[v]);
  }
}
```

## Example problem: Bridge Finding

- **Complexity?** $O(V + E)$, just one DFS.
- Bridges have broader relevance. A 2-edge connected component is one with no bridges. Compressing these turns any graph into a tree.
- Vertices whose removal disconnects the graph are called *articulation vertices*. There is a similar algorithm for finding them.
- But we won't talk about this more.
- **Moral:** DFS trees are cool, especially on undirected graphs.

## Example problem: Cycle Detection

- **Problem Statement** Given a directed graph, determine if there is a simple cycle.
- **Input**
  - First line, 2 integers $V, E$, the number of vertices and number of edges respectively.
  - Next $E$ lines, each a pair, $u_i v_i$.
  - $1 \leq V, E, \leq 100,000$.
- **Output** YES if there is a cycle, NO otherwise.

## Example problem: Cycle Detection

- If the graph is undirected, we can simply run a DFS on the graph, and return true if any vertex marked seen is visited again.

- However, this doesn't work for directed graphs, such as the diamond graph $(1 \to 2 \to 3 \leftarrow 4 \leftarrow 1)$.

- DFS on directed graphs is not as nice as on undirected graphs, just because $u$ can reach a visited node $v$ does not mean $v$ can reach $u$.

## Directed Graph Cycle Detection

- However, we can see that the only way a cycle can exist is if the DFS tree has a back-edge that goes up the tree.
- If there is a cycle $C$ and $u \in C$ is the first vertex our DFS visits in the cycle then all vertices in the cycle will be in the subtree of $u$ in the DFS tree. Hence this subtree must have some backedge to $u$.
- We can rephrase this algorithm as checking if any edge visits a vertex we are still recursing from. This means we reach a vertex $v$ that we are still trying to build the subtree for. So $v$ is an ancestor.
- It turns out this is easy to do — just mark each vertex "active" in a table during its preorder step (when we first reach $u$), and unmark it during its postorder step (when we return from $u$).

## Directed Graph Cycle Detection

- **Implementation**

```
const int UNSEEN = 0, ACTIVE = 1, COMPLETE = 2;
int status[N]; // initially UNSEEN

bool has_cycle(int u) {
  if (status[u] != UNSEEN) { return false; }
  status[u] = ACTIVE; // preorder (stack push)
  for (int v : edges[u]) {
    // if v is ACTIVE then it is lower down in the stack
    // so the cycle is from v to the top of the stack
    if (status[v] == ACTIVE || has_cycle(v)) { return true; }
  }
  status[u] = COMPLETE; // postorder (stack pop)
  return false;
}
```

## Table of Contents

## Special Classes of Graphs

- General graphs are quite hard to do many things on.
- Certain tasks are much more suited to specific classes of graphs.
    - Directed Acyclic Graphs (DAGs) are well suited for DP since you have a natural order to build up your recurrence.
    - Trees are well suited for like everything since from any given node, its subtrees should behave independently.

## Tree Representation

A *tree* is an undirected, connected graph ...

- with a unique simple path between any two vertices.
- where $E = V - 1$.
- with no cycles.
- where the removal of any edge disconnects the graph.

## Tree Representation

- We usually represent them as if they have a root.
- Hence each node naturally has a subtree associated to it.
- To represent a tree, we generally like to know for each node:
    - its parent,
    - its children (if any), and
    - additional problem-specific metadata on its subtree (e.g. size).

## Tree Representation

```
const int N = 1001001;

// We need the list of edges to construct our representation
// But we don't use it afterwards.
vector<int> edges[N];

int par[N]; // Parent. -1 for the root.
vector<int> children[N]; // Your children in the tree.
int size[N]; // As an example: size of each subtree.

void constructTree(int c, int cPar = -1) {
    par[c] = cPar;
    size[c] = 1;
    for (int nxt : edges[c]) if (nxt != par[c]) {
        constructTree(nxt, c);
        children[c].push_back(nxt);
        size[c] += size[nxt];
    }
}
```

## Tree Representation

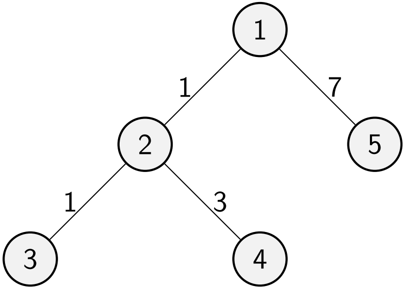
- Now that we have our representation, we can do most of what we want by just recursing using the `children` array.
- In some sense, as close to a line as we can get, and lines are very nice to work with.
- Many of the techniques you like for lines still work on a tree.
  - Linear Sweep *is to* ... DFS
  - DP *is to* ... DP on a tree
  - Range Tree *is to* ... Path Queries or Range tree over a tree
  - Divide and Conquer *is to* ... Centroid Decomposition

We'll talk about the first 3.

## Shortest Distance on a Tree

- **Problem Statement** Given a weighted tree, answer $Q$ queries of shortest distance between vertex $u_i$ and $v_i$.
- **Input** A tree described as $|V| - 1$ edges. Followed by $Q$ queries. $1 \leq |V|, Q \leq 100,000$.
- **Output** For each query, an integer, the shortest distance from $u_i$ to $v_i$.

## Shortest Distance on a Tree

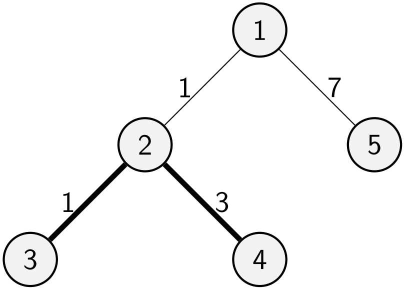


**Sample Queries:**

- **1 3**: 2
- **3 4**: 4
- **4 5**: 11

## Shortest Distance on a Tree

- As usual, assume you've run your tree representation DFS so the tree is now arbitrarily rooted.
- Well, the hard part seems to be figuring out what the path actually is.



- And for this it suffices to find the Lowest Common Ancestor (LCA)!

## Lowest common ancestor

- **Problem statement** You are given a labelled rooted tree, $T$, and $Q$ queries of the form, "What is the vertex furthest away from the root in the tree that is an ancestor of vertices labelled $u$ and $v$?"
- **Input** A rooted tree $T$ $(1 \leq |T| \leq 1,000,000)$, as well as $Q$ $(1 \leq Q \leq 1,000,000)$ pairs of integers $u$ and $v$.
- **Output** A single integer for each query, the label for the vertex that is furthest away from the root that is an ancestor of $u$ and $v$

## Lowest common ancestor

- **Algorithm 1** The most straightforward algorithm to solve this problem involves starting with pointers to the vertices $u$ and $v$, and then moving them upwards towards the root until they're both at the same depth in the tree, and then moving them together until they reach the same place
- This is $O(n)$ per query, since it's possible we need to traverse the entire height of the tree, which is not bounded by anything useful

## Lowest common ancestor

- The first step we can take is to try to make the “move towards root” step faster
- Since the tree doesn’t change, we can pre-process the tree somehow so we can jump quickly

## Binary function composition

- Let’s examine the parent relation parent[u] in the tree
- Our “move towards root” operation is really just repeated application of this parent relation
- The vertex two steps above u is parent[parent[u]], and three steps above is parent[parent[parent[u]]]

## Binary function composition

- Immediately, we can precompute the values parent[u][k], which is parent[u] applied k times
- This doesn't have an easy straightforward application to our problem, nor is it fast enough for our purposes

## Binary function composition

- If we only precompute parent[u][k] for each $k = 2^\ell$, we only need to perform $O(\log n)$ computations.
- Then, we can then compose up to $\log n$ of these precomputed values to obtain parent[u][k] for arbitrary $k$
- To see this, write out the binary expansion of $k$ and keep greedily striking out the most significant set bit — there are at most $\log n$ of them.

## Lowest common ancestor

- **Algorithm 2** Instead of walking up single edges, we use our precomputed parent[u][k] to keep greedily moving up by the largest power of 2 possible until we're at the desired vertex
- How do we find the LCA of $u$ and $v$ given our precomputation?
- First, move both $u$ and $v$ to the same depth.
- Binary Search! You are binary searching for the maximum amount you can jump up without reaching the same vertex. Then the parent of that vertex is the LCA.
- To implement this, we try jumping up in decreasing power of 2 order. We reject any jumps that result in $u$ and $v$ being at the same vertex.

## Lowest common ancestor

- **Implementation (preprocessing)**

```
// parent[u][k] is the 2^k-th parent of u
void preprocess() {
  for (int i = 0; i < n; i++) {
    // assume parent[i][0] (the parent of i) is already filled in
    for (int j = 1; (1<<j) < n; j++) { parent[i][j] = -1; }
  }

  // fill in the parent for each power of two up to n
  for (int j = 1; (1<<j) < n; j++) {
    for (int i = 0; i < n; i++) {
      if (parent[i][j-1] != -1) {
        // the 2^j-th parent is the 2^(j-1)-th parent of the 2^(j-1)-th
              parent
        parent[i][j] = parent[parent[i][j-1]][j-1];
      }
    }
  }
}
```

## Lowest common ancestor

- **Implementation (querying)**

```
int lca (int u, int v) {
  // make sure u is deeper than v
  if (depth[u] < depth[v]) { swap(u,v); }
  // log2s[i] holds the largest k such that 2^k <= i
  // precompute by DP: log2s[i] = log2s[i/2] + 1
  for (int i = log2s[depth[u]]; i >= 0; i--) {
    // repeatedly raise u by the largest possible power of two until it is
          the same depth as v
    if (depth[u] - (1<<i) >= depth[v]) { u = parent[u][i]; }
  }

  if (u == v) return u;

  for (i = log2s[depth[u]]; i >= 0; i--)
    if (parent[u][i] != -1 && parent[u][i] != parent[v][i]) {
      // raise u and v as much as possible without having them coincide
      // this is important because we're looking for the lowest common
            ancestor, not just any
      u = parent[u][i];
      v = parent[v][i];
    }

  // u and v are now distinct but have the same parent, and that parent is
        the LCA
  return parent[u][0];
}
```

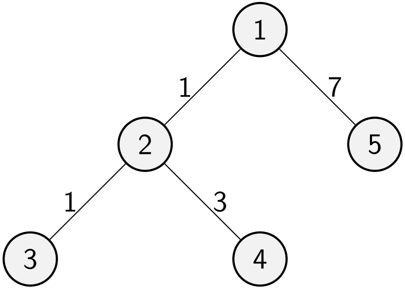
## Lowest common ancestor

- **Complexity?** $O(n \log n)$ time and memory preprocessing, $O(\log n)$ time per query.
- **Trap:** You **must** do the jumps from largest power of 2 to lowest. Otherwise it's just completely wrong.
- You can use this to support a bunch of path queries if there are no updates. Think of it as the range tree of paths in trees.
- Surprisingly you can do LCA in $O(n)/O(1)$ preprocessing/per query time.
- Even more surprisingly, one can use this to do Range Minimum Queries with no updates in $O(n)/O(1)$.

## Shortest Distance on a Tree

- **Problem Statement** Given a weighted tree, answer $Q$ queries of shortest distance between vertex $u_i$ and $v_i$.
- **Input** A tree described as $|V| - 1$ edges. Followed by $Q$ queries. $1 \leq |V|, Q \leq 100,000$.
- **Output** For each query, an integer, the shortest distance from $u_i$ to $v_i$.

## Shortest Distance on a Tree



**Sample Queries:**

- **1 3**: 2
- **3 4**: 4
- **4 5**: 11

## Shortest Distance on a Tree

- Now we know what the path between $u$ and $v$ looks like, it's $u \to lca$ followed by $lca \to v$. What else do we need to answer distance queries?
- Need to know lengths of certain ranges, like in a range tree.
- Generally, you would compute lengths and store it in the binary composition data structure you are using, like a range tree.
- But since sum has an inverse, we can be a bit lazier and use a cumulative sum like data structure instead.
- We will store $\text{dist}(root, u)$ for all $u$. Then $\text{dist}(u, v) = \text{dist}(root, u) + \text{dist}(root, v) - 2 \cdot \text{dist}(root, lca)$.

## Shortest Distance on a Tree

```
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 100100, LOGN = 20;
struct edge { int nd; long long d; };
int parent[MAXN][LOGN];
long long distToRoot[MAXN];
vector<edge> children[MAXN];
// Code to set up LCA and tree representation
void construct_tree(int c, int cPar = -1);
int lca(int a, int b);

void calc_dists_to_root(int c) {
    for (auto edg : children[c]) {
        distToRoot[edg.nd] = distToRoot[c] + edg.d;
        calc_dists_to_root(edg.nd);
    }
}

long long find_tree_dist(int a, int b) {
    int cLca = lca(a, b);
    return distToRoot[a] + distToRoot[b] - 2 * distToRoot[cLca];
}
```

## DAGs

- A DAG is a directed, acyclic graph.
- **Key Property 1:** Every DAG has a maximal vertex, one with no incoming edges.
- **Key Property 2:** Every DAG can be linearly ordered, i.e. there is some ordering of vertices such that edges only go from $v_i \to v_j$ where $i < j$.
- **Proof of (1):** Pick any vertex and keep arbitrarily following an incoming edge backwards if one exists. This either terminates or results in a cycle.
- **Proof of (2):** Induction with **(1)**.

## Topological Sort

- An order satisfying (2) is called a topological order or sort. It is an ordering of the vertices that has the property that if some vertex $u$ has a directed edge pointing to another vertex $v$, then $v$ comes after $u$ in the ordering.

- Clearly, if the graph has a cycle, then there does not exist a valid topological ordering of the graph.

## Topological Sort

- How do we compute a topological ordering?
- **Observation!** The key invariant of a DFS tells us that in acyclic graphs, every vertex $v$ can reach has been seen by the time we return from $v$.
- For an *acyclic* graph, this means every vertex after $v$ in the topsort order is popped before $v$ is popped.
- We can directly use the reverse of the postorder sequence of the graph.
- The postorder sequence of the graph is an ordering of the vertices of the graph in the order that each vertex reaches its postorder procedure (i.e. the order of stack pops).
- A vertex is only added after its children have been visited (and thus added), so the reverse order is a valid topological ordering.

## Topological Sort

- **Implementation**

```
// if the edges are in ASCENDING order of node number,
// this produces the lexicographically GREATEST ordering

void dfs(int u, vector<int>& postorder) {
  if (seen[u]) return;
  seen[u] = true;
  for (int v : edges[u]) { dfs(v, postorder); }
  postorder.push_back(u);
}

vector<int> topsort() {
  vector<int> res;
  for (int i = 0; i < n; i++) { dfs(i, res); }
  reverse(res.begin(), res.end()); // #include <algorithm>
  return res;
}
```

## Example problem: Water Falls

- **Problem Statement** A certain village is surrounded by $n$ mountain peaks. There are $m$ trails connecting pairs of mountain peaks.
  Every night rain will fall on a single mountain peak. The rain will then flow down trails to **strictly lower** mountain peaks until it reaches a mountain peak with no trail to any lower mountain peak.
  What is the maximum distance the water can flow?
- **Input** First line, $n$ $m$, $1 \leq n, m \leq 10^5$. Following this, $n$ integers, $h_i$, the heights of the mountain peaks.
  Following this, $m$ lines, each with a triple $u_i$ $v_i$ $d_i$, $0 \leq u_i, v_i < n$, $u_i \neq v_i$, $0 \leq d_i \leq 10^9$. This denotes a trail from mountain peak $u_i$ to mountain peak $v_i$ of length $d_i$.
- **Output** A single number, the maximum distance water can flow for before becoming stationary.

## Example problem: Water Falls

- **Example Input**

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

- **Example Output** 7

- **Explanation**: The longest path is $2 \to 0 \to 1$.

## Example problem: Water Falls

- **Observation:** The trails are directed edges (it is impossible $u_i$ is lower than $v_i$ AND $v_i$ is lower than $u_i$). Furthermore, it describes a DAG!
- Focus on one peak at a time. What is the longest path starting at peak 1? What information do I need to answer this?
- I need to know the longest path starting at each peak that peak 1 can reach.
- But since it is a DAG there is a natural order to process the peaks! The topsort order!
- In this case, it's even more natural, it's just increasing height order.

## Example problem: Water Falls

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAXV = 100100;
int V, E, h[MAXV];
vector<pair<int, long long>> allE[MAXV];
long long longestPath[MAXV], ans;
// Returns indices in topsort order (or dec. height order, reversed later).
vector<int> topsort() {}

int main() {
    cin >> V >> E;
    for (int i = 0; i < V; i++) cin >> h[i];
    for (int i = 0; i < E; i++) {
        int a, b; long long w; cin >> a >> b >> w;
        if (h[b] < h[a]) allE[a].emplace_back(b, w);
        if (h[a] < h[b]) allE[b].emplace_back(a, w);
    }
    vector<int> order = topsort();
    reverse(order.begin(),order.end());
    for (auto ind : order) {
        for (auto edge : allE[ind]) {
            longestPath[ind] = max(longestPath[ind],
                    longestPath[edge.first] + edge.second);
        }
        ans = max(ans, longestPath[ind]);
    }
    cout << ans << '\n';
}
```

## Summary

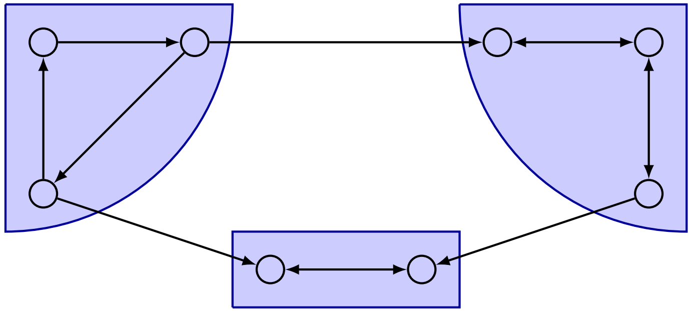
- More generally, known as ‘Longest Path in a DAG‘.
- In some sense, the prototypical example of Dynamic Programming.
    - Representing subproblems as vertices and the recurrence dependencies as edges, we *must* have a DAG in order to use DP.
- There is a way to reduce general graphs to DAGs called Strongly Connected Components (SCCs).

## Table of Contents

## Strongly Connected Components

- A *strongly connected component* (SCC) is a maximal subset of the vertices of a directed graph such that every vertex in the subset can reach every other vertex in that component.
- Condensing every strongly connected component to a single vertex results in a directed acyclic graph (DAG).

# Strongly Connected Components



## Finding Strongly Connected Components

- There are a few linear time algorithms known to compute the strongly connected components of a graph based on DFS.
- Kosaraju's algorithm is simple to implement but hard to understand.
- Another popular choice in these contests is Tarjan's algorithm.

## Tarjan's Algorithm

- Do a regular DFS on the graph, but with an explicit stack.
- When an item is pushed onto the stack, mark it as "in-stack", and unmark it as such when it is popped.
- If we want to push a vertex that is already "in-stack", then we've found a strongly connected component.
- Each item on the stack after this vertex can be reached from it, and can also reach that vertex.
- Simply pop everything off the stack including that vertex, and combine it into an SCC.
- The actual algorithm is slightly more complicated because some bookkeeping is required.

# Tarjan's Algorithm

## Implementation

```
// we will number the vertices in the order we see them in the DFS
int dfs_index[MAX_VERTICES];
// for each vertex, store the smallest number of any vertex we see
// in its DFS subtree
int lowlink[MAX_VERTICES];

// explicit stack
stack<int> s; // #include <stack>
bool in_stack[MAX_VERTICES];

// arbitrarily number the SCCs and remember which one things are in
int scc_counter;
int which_scc[MAX_VERTICES];

void connect(int v) {
  // a static variable doesn't get reset between function calls
  static int i = 1;
  // set the number for this vertex
  // the smallest numbered thing it can see so far is itself
  lowlink[v] = dfs_index[v] = i++;
  s.push(v);
  in_stack[v] = true;

  // continued
```

## Tarjan's Algorithm (continued)

- **Implementation**

```
  for (auto w : edges[v]) { // for each edge v -> w
    if (!dfs_index[w]) {  // w hasn't been visited yet
      connect(w);
      // if w can see something, v can too
      lowlink[v] = min(lowlink[v], lowlink[w]);
    }
    else if (in_stack[w]) {
      // w is already in the stack, but we can see it
      // this means v and w are in the same SCC
      lowlink[v] = min(lowlink[v], dfs_index[w]);
    }
  }
  // v is the root of an SCC
  if (lowlink[v] == dfs_index[v]) {
    ++scc_counter;
    int w;
    do {
      w = s.top(); s.pop();
      in_stack[w] = false;
      which_scc[w] = scc_counter;
    } while (w != v);
  }
}

// call connect for each vertex once
for (int v = 0; v < n; ++v) if (!dfs_index[v]) connect(v);
```

## Kosaraju's Algorithm

- Compute a reverse postordering of the graph, using depth-first search.
- Compute the reverse graph, by reversing all of the edges in the input graph.
- For every vertex in the reverse postordering, start a depth-first search on the reverse graph, ignoring any vertices that have already been visited. All vertices visited during this DFS will be part of the same SCC.

## (Partial) Proof of Kosaraju's Algorithm

- Why does this algorithm work?
- We need to show that if:
  - $u$ is before $v$ in a reverse postorder sequence, i.e. $u$ is popped *after* $v$ in the original DFS, and
  - $u$ can reach $v$ in the reverse graph, i.e. $v$ can reach $u$,

  then $u$ and $v$ belong to the same strongly connected component, i.e.
  - $u$ can reach $v$, and
  - $v$ can reach $u$ (already satisfied).
- It will still remain to prove that the identified components are maximal. This is left as an exercise.

## (Partial) Proof of Kosaraju's Algorithm

**Assumptions**

$u$ is popped after $v$ in the original DFS, and $v$ can reach $u$.

**Lemma**

$u$ is pushed before $v$ in the original DFS.

**Proof**

Suppose the opposite, i.e. $v$ is pushed before $u$. Since there is a path from $v$ to $u$, the DFS would pop $u$ *before* it pops $v$. This is a contradiction!

## (Partial) Proof of Kosaraju's Algorithm

**Assumptions**

$u$ is pushed before and popped after $v$ in the original DFS, and $v$ can reach $u$.

**Theorem**

$u$ can reach $v$.

**Proof**

Suppose the opposite, i.e. $u$ cannot reach $v$. Since $u$ was pushed before $v$, the DFS would pop $u$ *before* it pops (or even pushes) $v$. This is a contradiction!

## Kosaraju's Algorithm

```
void dfs(int u) {
        if (seen[u]) { return; }
        seen[u] = true;
        for (int v : edges[u]) { dfs(v); }
        postorder[p++] = u;
}

void dfs_r(int u, int mark) {
        if (seen_r[u]) { return; }
        seen_r[u] = true;
        scc[u] = mark;
        for (int v : edges_r[u]) { dfs_r(v, mark); }
}

int compute_sccs() {
        int sccs = 0;
        for (int i = 0; i < n; i++) if (!seen[i]) { dfs(i); }
        for (int i = p - 1; i >= 0; i--) {
                int u = postorder[i];
                // ignore visited vertices
                if (!seen_r[u]) { dfs_r(u, sccs++); }
        }
        return sccs;
}
```

## Table of Contents

## Satisfiability

- Satisfiability (SAT) is the problem of determining, given some Boolean formula, if there exists some truthiness assignment of variables which would result in the formula evaluating to true.
- Satisfiability is NP-hard in the general case.

## 2SAT

- 2-satisfiability (2SAT) is the problem of determining, given a set of constraints on pairs of Boolean variables, if there exists some truthiness assignment of variables which would result in the conjunction of all the constraints evaluating to true.
- Unlike general satisfiability, 2-satisfiability can be solved in linear time (number of variables plus number of constraints).

## 2SAT

- The inputs to a 2SAT problem are a set of constraints on Boolean variables with standard Boolean operators.
- Each clause is typically a disjunction: $x_1 \vee x_2$.
- Clauses with only a single variable (e.g. $\neg x_1$) can be written as a self-disjunction (e.g. $\neg x_1 \vee \neg x_1$).
- We can write disjunctions in equivalent implicative normal form:

$$x_1 \vee x_2 \equiv (\neg x_1 \rightarrow x_2) \wedge (\neg x_2 \rightarrow x_1)$$

## 2SAT

- These implications now form a directed graph with the Boolean variables (and their negations) as vertices, called the implication graph.

- What does it mean when some variable $x$ can reach some other variable $y$ in this implication graph?

    - If $x$ can reach $y$ in this graph, then $x \to y$.

## 2SAT

- When do we have a valid solution to our 2SAT instance?
- As long as we don't have any contradictions (i.e. $x \to \neg x$ and $\neg x \to x$, we can solve our 2SAT instance.
- In our implication graph, this is exactly the same as checking to see if $x$ and $\neg x$ are in the same strongly connected component!

## 2SAT

- We can then easily construct an actual solution to our 2SAT instance after computing the strongly connected components by assigning to each variable whichever truthiness value comes second in the topological ordering of the SCC condensed graph.
- If $x$ comes after $\neg x$ in the topological ordering of the condensed implication graph, then we say $x$ is true. Otherwise, we say it's false.
- Kosaraju's algorithm happens to topologically sort the SCCs already, so the code to actually construct a solution is extremely short, after computing the SCCs.

## 2SAT

- **Implementation**

```
void add_edge (int i, int j); // append to edges[i] and edges_r[j]
int compute_sccs (void); // unused return value (#sccs)

// vertices 0..n-1 are the variables, and n..2n-1 their negatives
inline int neg (int i) { return (i < n) ? (i + n): (i - n); }

void add_or (int i, int j) { // clause (i OR j)
  add_edge(neg(i), j); // NOT i implies j
  add_edge(neg(j), i); // NOT j implies i
}

bool is_2sat_solvable (void) {
  compute_sccs();
  for (int i = 0; i < n; i++) { // impossible iff T and F together
    if (scc[i] == scc[neg(i)]) { return false; }
  }
  return true;
}

vector<bool> find_2sat_solution (void) {
  vector<bool> solution(n);
  assert(is_2sat_solvable());
  // Kosaraju finds the SCCs in topological order
  // if F before T, then T // if T before F, then F
  for (int i = 0; i < n; i++) { solution[i] = (scc[i] > scc[neg(i)]); }
  return solution;
}
```

## Table of Contents

## Minimum Spanning Trees

- A *spanning tree* for some graph $G$ is a subgraph of $G$ that is a tree, and also connects (spans) all of the vertices of $G$.
- A *minimum spanning tree* (MST) is a spanning tree with minimum sum of edge weights.
- There are several similar algorithms to solve this problem.

## Minimum Spanning Trees

- To construct a minimum spanning tree of some graph $G$, we maintain a set of spanning forests, initially composed of just the vertices of the graph and no edges, and we keep adding edges until we have a spanning tree.
- Clearly, if we add $|V| - 1$ edges and we avoid constructing any cycles, we'll have a spanning tree.

## Minimum Spanning Trees

- How do we decide which edges to add, so that we end up with a minimum spanning tree?

- We can't add any edges to our spanning forest that has its endpoints in the same connected component of our spanning forest, or we'll get a cycle.

## Minimum Spanning Trees

- We can restrict ourselves to only the edges that cross components that we haven't connected yet.
- **Key Property:** There is a greedy exchange property. If $e$ has minimum weight of edges that connect components we haven't connected yet, then there is a spanning tree containing $e$.
- **Proof:** By contradiction, consider a MST without $e$. Then the addition of $e$ to this MST would introduce a cycle. But this cycle must contain another edge with weight at least $e$'s. Replace this edge with $e$.

## Minimum Spanning Trees

- The distinction between MST algorithms is in the way that they pick the next components to join together, and how they handle the joining.
- Kruskal's algorithm maintains multiple components at once and connects the two components that contain the next globally minimum edge.
- Prim's algorithm only ever connects one large connected component to single disconnected vertices in the spanning forest.

## Kruskal's Algorithm

- Kruskal's algorithm is generally simpler to implement, and more directly mirrors the mathematical properties of MSTs.
- Kruskal's algorithm:
    - For each edge $e$ in increasing order of weight, add $e$ to the MST if the vertices it connects are not already in the same connected component.
    - Maintain connectedness with union-find.
    - This takes $O(|E| \log |E|)$ time to run, with the complexity dominated by the time needed to sort the edges in increasing order.

## Kruskal's Algorithm

- **Implementation**

```
struct edge { int u, v, w; }; // endpoint, endpoint, weight
bool operator< (const edge& a, const edge& b) { return a.w < b.w; }

edge edges[N];

int root (int u); // union-find root with path compression
void join (int u, int v); // union-find join with size heuristic

long long kruskal() {
  sort(edges, edges+m); // sort by increasing weight
  long long total_weight = 0;
  for (int i = 0; i < m; i++) {
    edge& e = edges[i];
    // if the endpoints are in different trees, join them
    if (root(e.u) != root(e.v)) {
      total_weight += e.w;
      join(e.u, e.v);
    }
  }
  return total_weight;
}
```

## Prim's Algorithm

- An alternative is Prim's Algorithm.
- Instead of considering all components, instead pick a start vertex, say $v$ and consider 2 components: $\{v\}$ and $V \setminus \{v\}$.
- Now our property tells us there is a MST using the lowest weight edge between $\{v\}$ and $V \setminus \{v\}$.
- If this edge is $e : v \to w$, then add $e$ to our MST. Now repeat with components $\{v, w\}$ and $V \setminus \{v, w\}$.

## Prim's Algorithm

- **Implementation**

```
typedef pair<int,int> ii;

vector<ii> edges[N]; // (weight, destination)
bool in_tree[N];
// use 'greater' comparator for min-heap
priority_queue<ii, vector<ii>, greater<ii>> pq;

long long prim() {
  long long total_weight = 0;
  pq.emplace(0,0); // start at vertex 0 with total 0
  while (!pq.empty()) {
    auto cur = pq.top();
    pq.pop();
    int weight = cur.first, v = cur.second;
    // if this vertex was already seen before
    // it must have been seen at a better distance
    if (in_tree[v]) { continue; }
    in_tree[v] = true;
    total_weight += weight;
    // add all edges from this vertex
    for (auto nxt : edges[v]) { pq.push(nxt); }
  }
  return total_weight;
}
```