Graph Algorithms

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

Graphs and Graph Representations
Graph Traversals
Special Classes of Graphs
Minimum Spanning Trees
Single Source Shortest Paths
Dijkstra's Algorithm
Bellman-Ford Algorithm
All Pairs Shortest Paths
Implicit Graphs
Trees

- 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.

- 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 jth entry in the ith 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


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

int N = 1e6 + 5; // 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.

- **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.

- **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

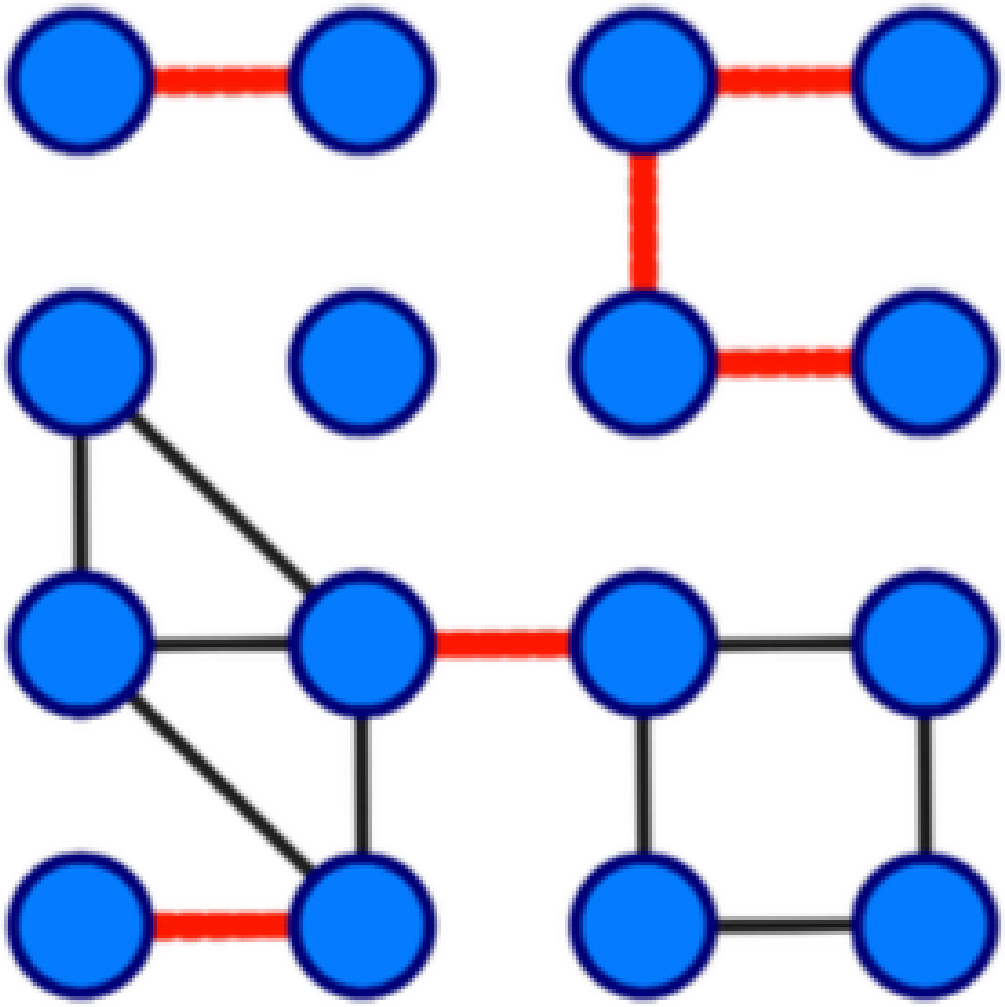
For now, we restrict ourselves to undirected graphs.

- 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.

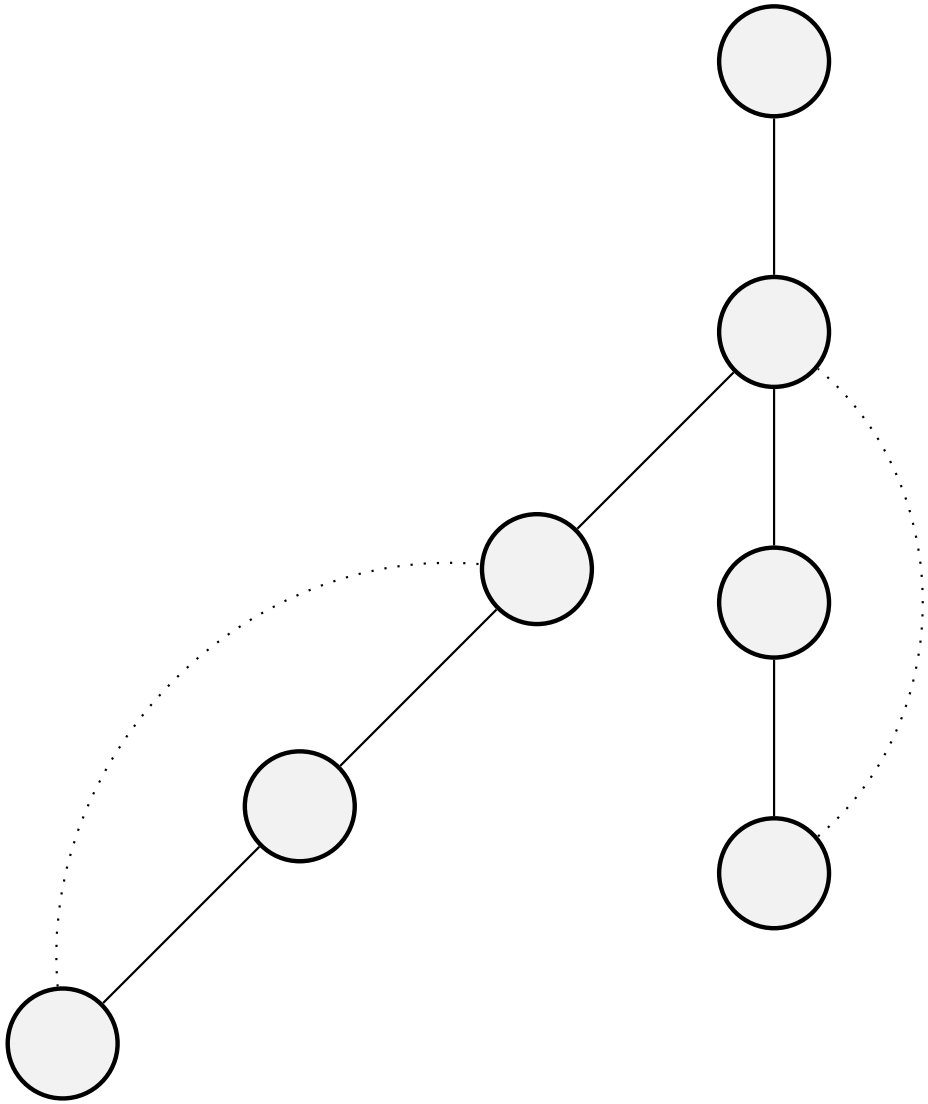
Example problem: Bridge Finding



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.

- **Implementation**

```
void dfs(int u, int from = -1) {
  low[u] = preorder[u] = T++;
  for (int v : edges[u]) {
    // ignore the edge to our parent in the dfs
    if (v == from) continue;
    // update the lowest value in the preorder sequence that we can
        reach
    if (preorder[v] != -1) low[u] = min(low[u], preorder[v]);
    else {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      // if we haven't visited v before, check to see if we have a
          bridge
      if (low[v] == preorder[v]) bridges.insert(make_pair(min(u, v),
          max(u, 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.

- 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).

- **Implementation**

```
// the vertices that are still marked active when this returns are the
       ones in the cycle we detected
bool has_cycle(int u) {
  if (seen[u]) return false;
  seen[u] = true;
  active[u] = true;
  for (int v : edges[u]) {
    if (active[v] || has_cycle(v)) return true;
  }
  active[u] = false;
  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.

DAG

- 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, there is some ordering of vertices such that edges only go from $v_i \rightarrow v_j$ 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 returned from before v is returned from.
- 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: in the order vertices return in).
- A vertex is only added after its children have been visited (and thus added), so the reverse order is a valid topological ordering.

- **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.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 E 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 E, $1 \leq N, E \leq 10^5$. Following this, N integers, h_i, the heights of the mountain peaks.
  Following this, E 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 $2 \rightarrow 0 \rightarrow 1$.

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 = 100005;
int V, E, h[MAXV];
vector<pair<int, long long>> allE[MAXV];
long long longestPath[MAXV], ans;
// Returns indices in topsort order (or inc. height order).
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';
}
```

- More generally, known as 'Longest Path in a DAG'.
- In some sense, the prototypical example of Dynamic Programming.
- There is a way to reduce general graphs to DAGs called Strongly Connected Components (SCCs).
- Interesting but likely won't have time to cover it.

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 what the parent of each node is, what the children of each node is and problem-specific additional metadata on its subtree (e.g: size).

Tree Representation


```cpp
const int N = 1e6 + 5;

// 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]) continue;
        constructTree(nxt, c);
        children[c].push_back(nxt);
        size[c] += size[nxt];
    }
}
```

- 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.
- We will see more trees in a bit.

Table of Contents

- 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.

- 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.

- 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.

- 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.

- 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.

- **Implementation**

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

edge edges[N];
int p[N];
int root (int u); // union-find root with path compression
void join (int u, int v); // union-find join with size heuristic

int mst() {
  sort(edges, edges+m); // sort by increasing weight
  int 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;
}
```

Table of Contents

Single Source Shortest Paths

- Given a weighted directed graph G with two specific vertices s and t, we want to find the shortest path that goes between s and t on the graph.
- Generally, algorithms which solve the shortest path problem also solve the single source shortest path problem, which computes shortest paths from a single source vertex to every other vertex.
- You can represent all the shortest paths from the same source as a tree.

- It's very important to distinguish between graphs where all edges are positive and graphs with negative weight edges! Why?
- Imagine a graph with a cycle whose total weight is negative.
- Even if there are no negative cycles, this may still cause problems depending on your algorithm choice!
- If the graph is acyclic, negative weight edges generally don't cause problems, but care should be taken regardless.

Single Source Shortest Paths

- Most single source shortest paths algorithms rely on the basic idea of building shortest paths iteratively. At any point, we keep what we think is the shortest path to each vertex and we update this by "relaxing" edges.
- Relax(u, v): if the currently found shortest path from our source s to a vertex v could be improved by using the edge (u, v), update it.
- For graphs with non-negative weights, we can get away with only relaxing vertices for which we know the optimal distance. But with negative weights, this becomes trickier.

- If we keep track for each v of its most recently relaxed incoming edge, we can find the actual path from the source to v. How?
- For each v, we know the vertex we would've come from if we followed the shortest path from the source.
- We can work backwards from v to the source to find the shortest path *from* the source *to* v.

- If we keep relaxing our edges until we can't anymore, then we will have a shortest path.
- How do we choose which edges to relax?

Dijkstra's Algorithm

- For now, suppose there are no negative edges.
- Visit each vertex u in turn, starting from the source s. Whenever we visit the vertex u, we relax all of the edges coming out of u.
- How do we decide the order in which to visit each vertex?
- We can do something similar to breadth-first search.
- The next vertex we process is always the unprocessed vertex with the smallest distance from the source.
- This ensures that we only need to process each vertex once: by the time we process a vertex, we have definitely found the shortest path to it. Prove this inductively!

Dijkstra's Algorithm

- To decide which vertex we want to visit next, we can either just loop over all of them, or use a priority queue keyed on each vertex's current shortest known distance from the source.
- Since we know that we have a complete shortest path to every vertex by the time we visit it in Dijkstra's algorithm, we know we only visit every vertex once.
- **Complexity** Dijkstra's Algorithm is $O(E \log V)$ using a binary heap as a priority queue, or $O(V^2)$ with a loop.

- The above only holds for graphs without negative edges!
- With negative edges, we may need to visit each vertex more than once, and it turns out this makes the runtime exponential in the worst case (and it's even worse with negative cycles).
- In short: don't use Dijkstra's if there's any negative edges! (But most graphs you see won't have them).

- **Implementation**

```
#include <queue>

typedef pair<int, int> edge; // (distance, vertex)
priority_queue<edge, vector<edge>, greater<edge>> pq;

// put the source s in the queue
pq.push(edge(0, s));
while (!pq.empty()) {
  // choose (d, v) so that d is minimal,
  // i.e. the closest unvisited vertex
  edge cur = pq.top();
  pq.pop();
  int v = cur.second, d = cur.first;
  if (seen[v]) continue;

  dist[v] = d;
  seen[v] = true;

  // relax all edges from v
  for (int i = 0; i < edges[v].size(); i++) {
    edge next = edges[v][i];
    int u = next.second, weight = next.first;
    if (!seen[u]) pq.push(edge(d + weight, u));
  }
}
```

Bellman-Ford Algorithm

- How do we handle negative edges in a more efficient way?
- How do we handle negative cycles?
- **Key Observation:** If a graph has no negative cycle then all shortest paths from u have length $\leq |V| - 1$. Conversely, a negative cycle implies there is a shortest path of length $|V|$ better than any path of length $|V| - 1$.
- So we should instead build up shortest paths by number of edges, not just from u outwards.
- Bellman-Ford involves trying to relax every edge of the graph (a *global relaxation*) $|V| - 1$ times and update our tentative shortest paths each time.
- Because every shortest path has at most $|V| - 1$ edges, if after all of these global relaxations, relaxations can still be made, then there exists a negative cycle.

Bellman-Ford Algorithm

- The time complexity of Bellman-Ford is $O(VE)$.
- However, if we have some way of knowing that the last global relaxation did not affect the tentative shortest path to some vertex v, we know that we don't need to consider edges coming out of v in our next global relaxation.
- This heuristic doesn't change the overall time complexity of the algorithm, but makes it run very fast in practice on random graphs.
- Sometimes called Shortest Path Faster Algorithm (SPFA) for some reason...

- **Implementation**

```
struct edge {
  int u, v, w;
  edge(int _u, int _v, int _w) : u(_u), v(_v), w(_w) {}
};
vector<int> dist(n);
vector<edge> edges;
// global relaxation: try to relax every edge in the graph
// Returns if any distance was updated.
bool relax() {
  bool relaxed = false;
  for (auto e = edges.begin(); e != edges.end(); ++e) {
    // we don't want to relax an edge from an unreachable vertex
    if (dist[e->u] != INF && dist[e->v] > dist[e->u] + e->w) {
      relaxed = true;
      dist[e->v] = dist[e->u] + e->w;
    }
  }
  return relaxed;
}
```

- **Implementation (continued)**

```
// Puts distances from source (n-1) in dist
// Returns true if there is a negative cycle, false otherwise.
// NOTE: You can't trust the dist array if this function returns True.
vector<int> find_dists_and_check_neg_cycle() {
  fill(dist.begin(), dist.end(), INF);
  dist[n-1] = 0;
  // |V|-1 global relaxations
  for (int i = 0; i < n - 1; i++) relax();

  // If any edge can be relaxed further, there is a negative cycle
  for (auto e = edges.begin(); e != edges.end(); ++e) {
    if (dist[e->u] != INF &&
        dist[e->v] > dist[e->u] + e->w) {
        return true;
    }
  }
  // Otherwise, no negative cycle, that condition is actually a iff
  return false;
}
```

Bellman-Ford Algorithm

- Slight technical note: Due to how we coded `relax`, after $n-1$ iterations, `dist[i]` may be the distance of a path of length $> n-1$. But this does not matter, everything still holds.
- If there is a negative cycle, you can't trust the distances computed. Call a vertex v ruined if its shortest distance from u is actually $-\infty$.
- For every negative cycle, in every relaxation round at least one of its vertices will be updated.
- Hence, to find all ruined vertices, DFS out of all vertices who were relaxed in the V-th round.
- To find a specific negative cycle, backtrack from any 'ruined' vertex.

Table of Contents

All Pairs Shortest Paths

- The all pairs shortest path problem involves finding the shortest path between every pair of vertices in the graph.
- Surprisingly, this can be found in $O(V^3)$ time and $O(V^2)$ memory.

- Let $f(u, v, i)$ be the length of the shortest path between u and v using only the first i vertices (i.e. the vertices with the i smallest labels) as intermediate vertices.
- The key is to build this up for increasing values of i.
- **Base Case:** Then $f(u, u, 0) = 0$ for all vertices u, and $f(u, v, 0) = w_e$ if there is an edge e from u to v, and infinity otherwise.

- Say we have already calculated $f(u, v, i-1)$ for all pairs u, v and some i. Then

$$f(u, v, i) = \min(f(u, v, i-1), f(u, i, i-1) + f(i, v, i-1)).$$

- The solution we already had, $f(u, v, i-1)$, definitely doesn't use i as an intermediate vertex.
- If i is the only new intermediate vertex we can use, the only new path that could be better is the shortest path $u \to i$ concatenated with the shortest path $i \to v$.

Floyd-Warshall Algorithm

- $f(u, v, i) = \min(f(u, v, i-1), f(u, i, i-1) + f(i, v, i-1))$
- Thus, $f(u, v, n)$ will give the length of the shortest path from u to v.
- Noting that to calculate the table for the next i, we only need the previous table, we see that we can simply overwrite the previous table at each iteration, so we only need $O(V^2)$ space.
- But what if $f(u, i, i-1)$ or $f(i, v, i-1)$ is overwritten in the table before we get to use it?
- Allowing the use of i as an intermediate vertex on a path to or from i is not going to improve the path, unless we have a negative-weight cycle.

- **Implementation**

```
// the distance between everything is infinity
for (int u = 0; u < n; ++u) for (int v = 0; v < n; ++v) {
  dist[u][v] = 2e9;
}

// update the distances for every directed edge
for (/* each edge u -> v with weight w */) dist[u][v] = w;

// every vertex can reach itself
for (int u = 0; u < n; ++u) dist[u][u] = 0;

for (int i = 0; i < n; i++) {
  for (int u = 0; u < n; u++) {
    for (int v = 0; v < n; v++) {
      // dist[u][v] is the length of the shortest path from u to v
           using only 0 to i-1 as intermediate vertices
      // now that we're allowed to also use i, the only new path that
           could be shorter is u -> i -> v
      dist[u][v] = min(dist[u][v], dist[u][i] + dist[i][v]);
    }
  }
}
```

What if there is a negative cycle?

- If there is a negative-weight cycle, our invariant is instead $f(u, v, i) \leq$ shortest simple path from $u \to v$ only using the first i vertices as intermediaries.
- Hence $f(u, u, n)$ will be negative for vertices in negative cycles. Also you can't trust the calculated distances, same as Bellman-Ford.
- Note that if there *are* negative-weight edges, but *no negative cycles*, Floyd-Warshall will correctly find all distances.
  - Every *undirected* graph with a negative-weight edge contains a negative cycle.

- How can we find the actual shortest path?
- As well as keeping track of a `dist` table, any time the improved path (via i) is used, note that the next thing on the path from u to v is going to be the next thing on the path from u to i, which we should already know because we were keeping track of it!
- When initialising the table with the edges in the graph, don't forget to set v as next on the path from u to v for each edge $u \rightarrow v$.
- Implementing this functionality is left as an exercise.

Table of Contents

- Although some graph interpretations are obvious (e.g. cities and highways), it's often the case that the graph you must run your algorithm on is non-obvious.
- Often this doesn't admit a clean implementation using something like an explicit adjacency list.
- In many cases like this, it may be a better idea to compute the adjacencies on the fly.
- Also, sometimes modifying the graph you consider is helpful!

Example Problem: Two Buttons

- **Problem Statement** You have found a strange device that has a red button, a blue button, and a display showing a single integer, initially n. Pressing the red button multiplies the number by two; pressing the blue button subtracts one from the number. If the number stops being positive, the device breaks. You want the display to show the number m. What is the minimum number of button presses to make this happen?
- **Input** Two space-separated integers n and m $(1 \leq n, m \leq 10^7)$.
- **Output** A single number, the smallest number of button presses required to get from n to m.

Example Problem: Two Buttons

- In this example, we should think of our button presses as transitions.
- Hence our graph has numbers as its vertices and edges representing which numbers can reach each other through button presses.
- The graph is unweighted, so we just need to do a simple BFS to find the answer.

- **Implementation**

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

const int MAXVAL=10000000;
int n, m, v[MAXVAL+5];
queue<int> q;

int main () {
  cin >> n >> m;
  fill(v, v + 200000001, 1e9);
  q.push(n);
  v[n] = 0;
  while (q.size()) {
    int i = q.front(); q.pop();
    if (i > 0 && v[i] + 1 < v[i-1]) {
      v[i-1] = v[i] + 1;
      q.push(i - 1);
    }
    if (i <= MAXVAL && v[i] + 1 < v[i*2]) {
      v[i*2] = v[i] + 1;
      q.push(i * 2);
    }
  }
  cout << v[m];
}
```

Example Problem: Rock Climbing

- **Problem Statement** You are a rock climber trying to climb a wall. On this wall, there are N rock climbing holds for you to use. Whenever you are on the wall, you must be holding on to exactly three holds, each of which can be at most D distance from the other two. To move on the wall, you can only disengage from one of the holds and move it to another hold that is within D distance of the two holds that you are still holding onto. You can move from hold to hold at a rate of 1m/s. How can you reach the highest hold in the shortest amount of time, starting from some position that includes the bottom hold?
- **Input** A set of up to N ($1 \leq N \leq 50$) points on a 2D plane, and some integer D ($1 \leq D \leq 1000$). Each point's coordinates will have absolute value less than 1,000,000.
- **Output** A single number, the least amount of time needed to move from the bottom to the top.

Example Problem: Rock Climbing

- If there was no restriction that required you to always be using three holds, then this would just be a standard shortest path problem that is solvable using Dijkstra's algorithm.
- We would just need to take the points as the vertices and the distance between points as the edge weights.

Example Problem: Rock Climbing

- However, we need to account for the fact that we must be using three holds clustered together at any time.
- But there is a natural interpretation of the hold restriction in terms of a graph: when we move from some position that uses holds $\{a, b, c\}$ to some position where we use holds $\{a, b, d\}$, we can say that we are moving from some vertex labelled $\{a, b, c\}$ to some vertex labelled $\{a, b, d\}$.
- It can be determined whether or not such a move is allowed, i.e. if there is an edge between these vertices, in constant time.

Example Problem: Rock Climbing

- Now, we have a graph where we have $O(N^3)$ vertices and $O(N^4)$ edges.
- Running our shortest path algorithm on this graph directly will give us the answer we want, by definition.
- So we can solve this problem in $O(E \log V) = O(N^4 \log N^3) = O(N^4 \log N)$ time.

- **Implementation**

```
struct state {
  int pid[3];
  int dist;
};

bool operator< (const state &a, const state &b) {
  return a.dist > b.dist;
}

priority_queue<state> pq;
pq.push(begin);
bool running = true;
while (!pq.empty() && running) {
  state cur = pq.top();
  pq.pop();
  // check if done
  for (int j = 0; j < 3; j++) {
    if (cur.pid[j] == n) {
      running = false;
      break;
    }
    // to be continued
```

Example Problem: Rock Climbing

- **Implementation (continued)**

```
      // try disengaging our jth hold
      for (int j = 0; j < 3; j++) {
        // and moving to hold number i
        for (int i = 1; i <= n; i++) {
          // can't reuse existing holds
          if (i == cur.pid[0] || i == cur.pid[1] || i == cur.pid[2])
              continue;

          state tmp = cur;
          tmp.dist += dist(cur.pid[j], i);
          tmp.pid[j] = i;
          sort(tmp.pid, tmp.pid + 3);

          // try to move if valid
          if (valid(tmp) &&
              (!seen[tmp.pid[0]][tmp.pid[1]][tmp.pid[2]] ||
               seen[tmp.pid[0]][tmp.pid[1]][tmp.pid[2]] > tmp.dist)) {
            pq.push(tmp);
            seen[tmp.pid[0]][tmp.pid[1]][tmp.pid[2]] = tmp.dist;
          }
        }
      }
    }
}
```

- **Problem Statement** There are N cities, each in one of C countries. There are two modes of travel.
  - There are A unidirectional direct flight routes connecting two cities, u_i, v_i with weight w_i.
  - There are B unidirectional inter-country flights, connecting two countries, a_i, b_i with a weight w_i. These flights can be boarded from any city in the source country and disembarked from in any city in the destination country.

  Find the shortest distance from city 1 to city N.
- **Input** First line, 4 integers N, C, A, B. $1 \leq N, C, A, B \leq 100,000$. Next line, N integers, c_i, denoting the country the i-th city is in. Next A lines each with 3 integers $u_i, v_i, w_i, 1 \leq u_i, v_i \leq N, 1 \leq w_i \leq 10^9$. Next B lines each with 3 integers $a_i, b_i, w_i, 1 \leq a_i, b_i \leq C, 1 \leq w_i \leq 10^9$.
- **Output** A single integer, shortest distance from city $1 \rightarrow N$. -1 if impossible.

Example Problem: Intercountry

- **Sample Input:**

```
4 3 2 2
1 2 2 3
1 4 100
2 4 20
1 2 50
1 3 80
```

- **Sample Output**: 70
- **Explanation**: Fly with an intercountry flight of cost 50 to city 2. Then take a direct flight with cost 20 to city 4.

- How to view this as a graph? Without inter-country flights, this is routine.
- With inter-country flights, we could just generate all edges between cities in country a_i and b_i.
- But this is too many edges.

Example Problem: Intercountry

- **Observation:** We should only consider inter-country flights originating from A the first time we reach a city in A. Similarly, we should only consider inter-country flights to B once.
- This sounds just like how we treat cities in Dijkstra's.
- To encode this, we should treat countries just like cities in our graph, they should have nodes. Be careful, we need 2 nodes per country, one to encode outgoing flights and one to encode incoming flights. What are the edges though?
- The natural ones.
  - Cities go to the "outgoing" node for their country.
  - The "incoming" node for a country goes to all cities in that country.
  - "Outgoing" country nodes connect by inter-country flights to "incoming" country nodes.
- $O(N + C)$ nodes, $O(N + A + B)$ edges. Okay!

Example Problem: Intercountry


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

const int MAXN = 100005;
const int MAXC = 100005;
int N, C, A, B;
// (dest, dist)
// "outgoing" country nodes are at MAXN + (country id)
// "incoming" country nodes are at MAXN + MAXC + (country id)
vector<pair<int, long long>> allE[MAXN+2*MAXC];

int main() {
    cin >> N >> C >> A >> B;
    for (int i = 1; i <= N; i++) {
        int cC; cin >> cC;
        allE[i].emplace_back(MAXN + cC, 0);
        allE[MAXN+MAXC+cC].emplace_back(i, 0);
    }
    for (int i = 0; i < A; i++) {
        int a, b; long long w;
        cin >> a >> b >> w;
        allE[a].emplace_back(b, w);
    }
    for (int i = 0; i < B; i++) {
        int a, b; long long w;
        cin >> a >> b >> w;
        allE[MAXN+a].emplace_back(MAXN+MAXC+b);
    }
    // Run your favorite shortest dist algo!
}
```

Example Problem: Escape From Enemy Territory88

- **Problem Statement** You are at some position on a grid and wish to reach your safe house at some other location on the grid. However, also on certain cells on the grid are enemy safe houses, which you do not want to go near. What is the maximum possible distance you can stay away from every enemy safe house, and still be able to reach your own safe house? When there are multiple paths that keep the same distance from the enemy safe houses, print the shortest one. Distance in this problem is measured by Manhattan distance.

Example Problem: Escape From Enemy Territory89

- **Input** An $N \times M$ grid ($1 \leq N, M, \leq 1000$), and the location of your starting point, your safe house, and all the enemy safe houses. There are up to 10,000 enemy safe houses.
- **Output** Two integers, the maximum distance that you can stay away from every enemy safe house and still be able to reach your safe house, and the shortest length of such a path.

Example Problem: Escape From Enemy Territory90

- If there was no restriction stating that you must stay as far away from the enemy safe houses as possible, this would be a simple shortest path problem on a grid.
- What if we already knew how far we need to stay away from each enemy safe house?

Example Problem: Escape From Enemy Territory91

- Call the distance that we know we need to stay away from the enemy safe houses X.
- We just need to BFS out from every enemy safe house to a distance of X squares, marking all of those squares as unusable. Just marking them as seen will suffice.
- Then we can find the answer with a simple BFS from the starting point. It will ignore the squares that are too close to enemy safe houses because we've marked them as seen.

Example Problem: Escape From Enemy Territory92

- How do we view our original optimisation problem in terms of this decision problem?
  - Our simpler problem is a decision problem because we answer whether or not it's possible to get from the starting point to the safe house with distance X.
  - The original problem is an optimisation problem because it requires a 'best' answer.
- Observe that if we can stay X distance away from the enemy safe houses, then any smaller distance is also feasible, and if we cannot stay X distance away, then any larger distance is also infeasible.

Example Problem: Escape From Enemy Territory93

- This monotonicity allows us to binary search for the largest X such that we can still reach our safe house from our starting point, which we check using the BFS procedure outlined earlier.
- **Complexity** Each check takes $O(NM)$ time, and we need to perform $\log X_{MAX} = \log(N + M)$ of these checks in our binary search, so this algorithm takes $O(NM \log(N + M))$ total.

Example Problem: Escape From Enemy Territory94

- **Implementation**

```
const int di[4] = { -1, 1, 0, 0 };
const int dj[4] = { 0, 0, -1, 1 };

vector<pair<int,int>> enemies;

// search from all enemy safe houses to find
// each square's minimum distance to an enemy safe house
queue<pair<int, int>> q;
for (auto it = enemies.begin(); it != enemies.end(); ++it) {
  q.push(*it);
}
while (!q.empty()) {
  pair<int, int> enemy = q.front(); q.pop();
  int i = enemy.first, j = enemy.second;
  // try all neighbours
  for (int d = 0; d < 4; ++d) {
    int ni = i + di[d], nj = j + dj[d];
    // if off board, ignore
    if (ni < 0 || ni >= N || nj < 0 || nj >= M) continue;
    if (dist_to_enemy[ni][nj] != -1) continue;
    dist_to_enemy[ni][nj] = dist_to_enemy[i][j] + 1;
    q.push(make_pair(ni, nj));
  }
}
```

Example Problem: Escape From Enemy Territory95

- **Implementation (continued)**

```
// binary search
int lo = -1, hi = min(dist_to_enemy[i1][j1], dist_to_enemy[i2][j2]),
      sol = -1;
while (lo != hi) {
  int X = (lo + hi + 1) / 2;
  // BFS, since the edges are unit weight
  vector<vector<int> > d2(N, vector<int>(M, -1));
  d2[i1][j1] = 0;
  q.push(make_pair(i1, j1));
  while (!q.empty()) {
    int i = q.front().first, j = q.front().second; q.pop();
    for (int d = 0; d < 4; ++d) {
      int ni = i + di[d], nj = j + dj[d];
      if (ni < 0 || ni >= N || nj < 0 || nj >= M) continue;
      if (dist_to_enemy[ni][nj] < X) continue;
      if (dist[ni][nj] != -1) continue;
      dist[ni][nj] = dist[i][j] + 1;
      q.push(make_pair(ni, nj));
    }
  }

  if (dist[i2][j2] == -1) hi = X - 1;
  else lo = X, sol = dist[i2][j2];
}
```

Table of Contents

Graphs and Graph Representations
Graph Traversals
Special Classes of Graphs
Minimum Spanning Trees
Single Source Shortest Paths
Dijkstra's Algorithm
Bellman-Ford Algorithm
All Pairs Shortest Paths
Implicit Graphs
Trees

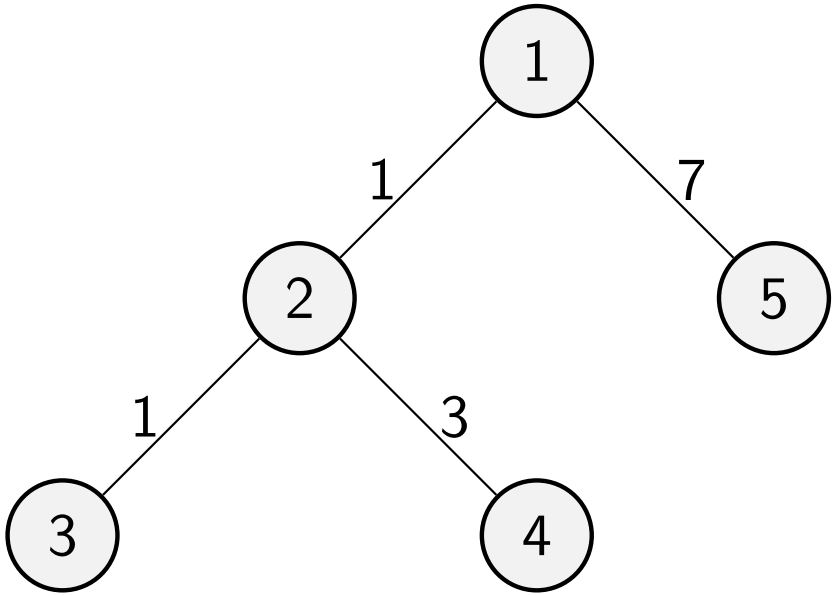
- The nicest class of graphs. In some sense, the closest to a line you can get.
- Many of the techniques you like for lines still work on a tree.
  - Linear Sweep *is to...* DFS
  - DP *is to...* DP on a tree
  - Range Tree *is to...* Path Queries or Range tree over a tree
  - Divide and Conquer *is 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.

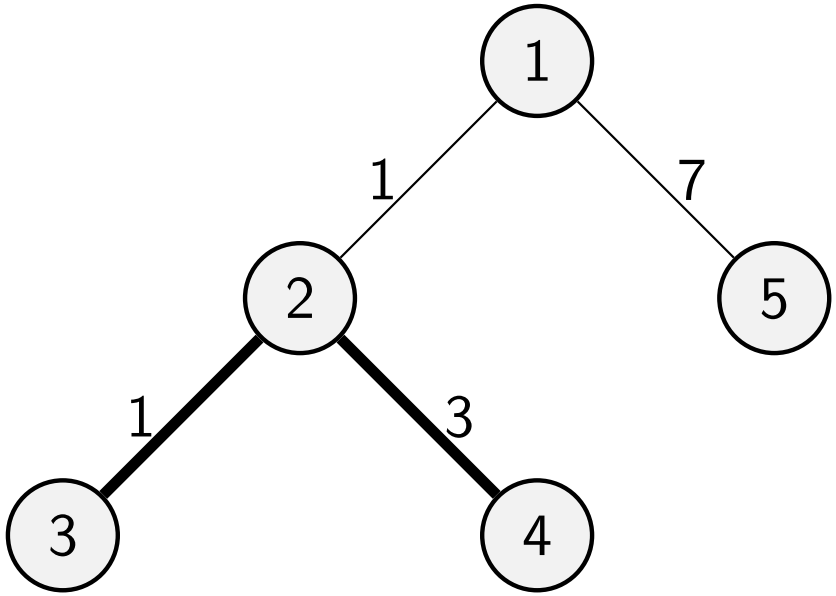
Shortest Distance on a Tree



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

- **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

- 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]]]

- 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.

- **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.

- **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];
      }
    }
  }
}
```

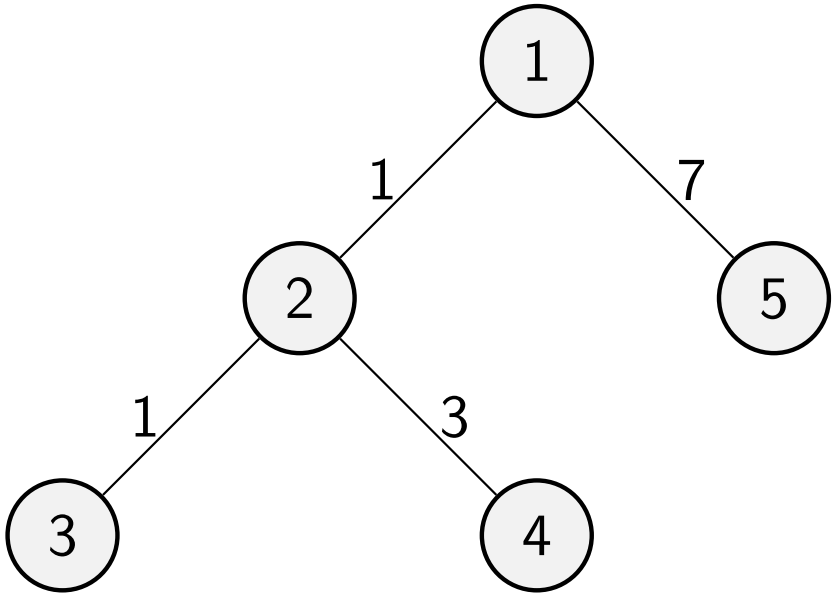
- **Implementation (querying)**

```
int lca (int u, int v) {
  // make sure u is deeper than v
  if (depth[u] < depth[v]) swap(u,v);

  // log[i] holds the largest k such that 2^k <= i
  for (int i = log[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 = log[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];
}
```

- **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 $O(n)/O(1)$.

- **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

- 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 $\mathrm{dist}(root, u)$ for all u. Then $\mathrm{dist}(u, v) = \mathrm{dist}(root, u) + \mathrm{dist}(root, v) - 2 \cdot \mathrm{dist}(root, lca)$.

Shortest Distance on a Tree


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

const int MAXN = 100000, LOGN = 18;
struct edge { int nd; long long d; };
int parent[MAXN+5][LOGN];
long long distToRoot[MAXN+5];
vector<edge> children[MAXN+5];
// 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];
}
```