Distance in a graph

Depth-first search finds vertices reachable from another given vertex. The paths are not the shortest ones.

Distance between two nodes: length of the shortest path between them.

Breadth-first search

Similar to a wave propagation
Breadth-first search

Breadth-first search

BFS algorithm

• BFS visits vertices layer by layer: 0, 1, 2, ..., d.

• Once the vertices at layer d have been visited, start visiting vertices at layer d + 1.

• Algorithm with two active layers:
  – Vertices at layer d (currently being visited).
  – Vertices at layer d + 1 (to be visited next).

• Central data structure: a queue.

BFS algorithm

function BFS(G, s)
// Input: Graph G(V, E), source vertex s.
// Output: For each vertex u, dist[u] is the distance from s to u.

for all u ∈ V: dist[u]=∞
dist[s] = 0
Q = {s} // Queue containing just s

while not Q.empty():
    u = Q.pop_front()
    for all (u, v) ∈ E:
        if dist[v] = ∞:
            dist[v] = dist[u] + 1
            Q.push_back(v)

Runtime O(|V| + |E|): Each vertex is visited once, each edge is visited once (for directed graphs) or twice (for undirected graphs).
Distances on edges

Graphs © Dept. CS, UPC


Reusing BFS

Graphs © Dept. CS, UPC

A
B
C
D
E

跟踪跑步者

图

The runners algorithm

At the beginning, we have one runner for each edge \( u \rightarrow v \) waiting at \( u \) for the arrival of the first runner. When the first runner arrives at \( u \), all runners waiting at \( u \) start running along their edge \( u \rightarrow v \). All runners run at a constant speed: \( \frac{1}{d_{\text{unit}}} \frac{\text{unit}}{\text{time unit}} \).

We just need to annotate the first arrival time at every vertex.

Algorithm:

- Set the expected arrival time at \( s \) to 0.
- No expected arrival time for the rest of vertices (\( \infty \)).
- Repeat until there are no more runners:
  - Select the earliest expected arrival time at a vertex not visited yet (vertex \( u \), time \( T \)). Annotate final distance \( s \rightarrow u \) to \( T \).
  - For each outgoing \( u \rightarrow v \):
    - If \( v \) has not been visited yet and the expected arrival time for the runner \( u \rightarrow v \) is smaller than the expected arrival time at \( v \) by any other active runner, start running along \( u \rightarrow v \) and stop the other runners going to \( v \).
    - Otherwise, do not start running.

Annotate the time of the first arrival at each node.
We need to:
- keep a list of active runners and their expected arrival times.
- select the earliest runner arriving at a vertex.
- update the active runners and their arrival times.

Dijkstra’s algorithm for shortest paths

```plaintext
function ShortestPaths(G, l, s)
// Input: Graph G(V,E), source vertex s,
//    ... > dist[u] + l(u,v):
dist[v] = dist[u] + l(u,v)
prev[v] = u
Q.decreasekey(v)
```

```
for all u ∈ V:
dist[u] = ∞
prev[u] = nil

dist[s] = 0
Q = makequeue(V)  // using dist as keys

while not Q.empty():
    u = Q.deleteMin()
    for all (u,v) ∈ E:
        if dist[v] > dist[u] + l(u,v):
            dist[v] = dist[u] + l(u,v)
            prev[v] = u
            Q.decreaseKey(v)
```
Dijkstra’s algorithm: complexity

\[
Q = \text{makequeue}(V)
\]

\[
\text{while not } Q.\text{empty}():
\]

\[
u = Q.\text{deletemin}()
\]

\[
\text{for all } (u, v) \in E:
\]

\[
\text{if } \text{dist}[v] > \text{dist}[u] + l(u, v):
\]

\[
\text{dist}[v] = \text{dist}[u] + l(u, v)
\]

\[
\text{prev}[v] = u
\]

\[
Q.\text{decreasekey}(v)
\]

\[|V| \text{ times} \]

\[|E| \text{ times} \]

The skeleton of Dijkstra’s algorithm is based on BFS, which is \(O(|V| + |E|)\).

We need to account for the cost of:

- makequeue: insert |V| vertices to a list.
- deletemin: find the vertex with min dist in the list (|V| times)
- decreasekey: update dist for a vertex (|E| times)

Let us consider two implementations for the list: vector and binary heap

### Graphs with negative edges

- Dijkstra’s algorithm does not work:

  ![Graph](image)

  Dijkstra would say that the shortest path \(A \rightarrow B\) has length=3.

- Dijkstra is based on a safe update each time an edge \((u, v)\) is treated:

  \[
  \text{dist}(v) = \min\{\text{dist}(v), \text{dist}(u) + l(u, v)\}
  \]

- Problem: non-promising updates are not done.

- Solution: let us not abort updates that early.

### Graphs with negative edges

- The shortest path from \(s\) to \(t\) can have at most \(|V| - 1\) edges:

  \[
  s \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \ldots \rightarrow u_k \rightarrow t
  \]

- If the sequence of updates includes

  \[(s, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_k, t),\]

  in that order, the shortest distance from \(s\) to \(t\) will be computed correctly (updates are always safe). Note that the sequence of updates does not need to be consecutive.

- Solution: update all edges \(|V| - 1\) times!

- Complexity: \(O(|V| \cdot |E|)\).

---

**Implementation**

<table>
<thead>
<tr>
<th></th>
<th>deletemin</th>
<th>insert/decreasekey</th>
<th>Dijkstra’s complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>(O(</td>
<td>V</td>
<td>))</td>
</tr>
<tr>
<td>Binary heap</td>
<td>(O(\log</td>
<td>V</td>
<td>))</td>
</tr>
</tbody>
</table>

**Binary heap:**

- The elements are stored in a complete (balanced) binary tree.
- **Insertion:** place element at the bottom and let it **bubble up** swapping the location with the parent (at most \(\log_2 |V|\) levels).
- **Deletemin:** Remove element from the root, take the last node in the tree, place it at the root and let it **sift down** (at most \(\log_2 |V|\) levels).
- **Decreasekey:** decrease the key in the tree and let it bubble up (same as insertion). A data structure might be required to known the location of each vertex in the heap (table of pointers).
Bellman-Ford algorithm

function ShortestPaths(G, l, s)
// Input: Graph G(V, E), source vertex s,
// edge lengths {l_e ∈ E}, no negative cycles.
// Output: dist[u] has the distance from s,
// prev[u] has the predecessor in the tree

for all u ∈ V:
    dist[u] = ∞
    prev[u] = nil

dist[s] = 0

repeat |V| − 1 times:
    for all (u, v) ∈ E:
        if dist[v] > dist[u] + l(u, v):
            dist[v] = dist[u] + l(u, v)
            prev[v] = u

Graphs © Dept. CS, UPC

Bellman-Ford: example

Negative cycles

• What is the shortest distance between S and A?

Bellman-Ford does not work as it assumes that the shortest path will not have more than |V| − 1 edges.

• A negative cycle produces −∞ distances by endlessly applying rounds to the cycle.

• How to detect negative cycles?
  – Apply Bellman-Ford (update edges |V| − 1 times)
  – Perform an extra round and check whether some distance decreases.

Shortest paths in DAGs

• DAG’s property:

In any path of a DAG, the vertices appear in increasing topological order.

• Any sequence of updates that preserves the topological order will compute distances correctly.

• Only one round visiting the edges in topological order is sufficient: O(|V| + |E|).

• How to calculate the longest paths?
  – Negate the edge lengths.
  – Compute the shortest paths.
DAG shortest paths algorithm

```plaintext
function DagShortestPaths(G, l, s)
    // Input: DAG G(V,E), source vertex s,
    //        edge lengths {l(e) ∈ E}.
    // Output: dist[u] has the distance from s,
    //        prev[u] has the predecessor in the tree
    for all u ∈ V:
        dist[u] = ∞
        prev[u] = nil
    dist[s] = 0
    Linearize G
    for all u ∈ V in linearized order:
        for all (u, v) ∈ E:
            if dist[v] > dist[u] + l(u, v):
                dist[v] = dist[u] + l(u, v)
                prev[v] = u
```

Minimum Spanning Trees

- Nodes are computers
- Edges are links
- Weights are maintenance cost
- Goal: pick a subset of edges such that
  - the nodes are connected
  - the maintenance cost is minimum

The solution is not unique.
Find another one!

Property:
An optimal solution cannot contain a cycle.

Properties of trees

- A tree is an undirected graph that is connected and acyclic.
- Property: A tree on n nodes has n − 1 edges.
  - Start from an empty graph. Add one edge at a time making sure that it connects two disconnected components.
- Property: Any connected, undirected graph G = (V, E) with |E| = |V| − 1 is a tree.
  - It is sufficient to prove that G is acyclic. If not, we can always remove edges from cycles until the graph becomes acyclic.
- Property: Any undirected graph is a tree iff there is a unique path between any pair of nodes.
  - If there would be two paths between two nodes, the union of the paths would contain a cycle.

Minimum Spanning Tree

- Given an undirected graph G = (V, E) with edge weights \( w_e \), find a tree \( T = (V, E') \), with \( E' \subseteq E \), that minimizes
  \[
  \text{weight}(T) = \sum_{e \in E'} w_e.
  \]
- Kruskal's algorithm: repeatedly add the next lightest edge that does not produce a cycle.
The cut property

Suppose edges \( X \) are part of an MST of \( G = (V, E) \). Pick any subset of nodes \( S \) for which \( X \) does not cross between \( S \) and \( V - S \), and let \( e \) be the lightest edge across this partition. Then \( X \cup \{ e \} \) is part of some MST.

Proof (sketch): Let \( T \) be an MST and assume \( e \) is not in \( T \). If we add \( e \) to \( T \), a cycle will be created with another edge \( e' \) across the cut \((S, V - S)\). We can now remove \( e' \) and obtain another tree \( T' \) with weight \( T' \leq \text{weight}(T) \). Since \( T \) is an MST, then the weights must be equal.

Kruskal’s algorithm

- Informal algorithm:
  - Sort edges by weight
  - Visit edges in ascending order of weight and add them as long as they do not create a cycle

- How do we know whether adding new edge will create a new cycle? How much does it cost to know it?

Disjoint sets

- A data structure to store a collection of disjoint sets.

- Operations:
  - \( \text{makeset}(x) \): creates a singleton set containing just \( x \).
  - \( \text{find}(x) \): returns the identifier of the set containing \( x \).
  - \( \text{union}(x, y) \): merges the sets containing \( x \) and \( y \).

- Kruskal’s algorithm uses disjoint sets and calls
  - \( \text{makeset}: |V| \) times
  - \( \text{find}: 2 \cdot |E| \) times
  - \( \text{union}: |V| - 1 \) times
Kruskal’s algorithm

function Kruskal(G, w)
// Input: A connected undirected Graph G(V,E)
// with edge weights w_e.
// Output: An MST defined by the edges in X.
for all u ∈ V: makeset(u)
X = {}
sort the edges in E by weight
for all (u,v) ∈ E, in ascending order of weight:
    if (find(u) ≠ find(v)):
        X = X ∪ {(u,v)}
        union(u,v)

Disjoint sets

• The nodes are organized as a set of trees. Each tree represents a set.

• Each node has two attributes:
  – parent (π): ancestor in the tree
  – rank: height of the subtree

• The root element is the representative for the set: its parent pointer is itself (self-loop).

• The efficiency of the operations depends on the height of the trees.

function makeset(x):
π(x) = x
rank(x) = 0

function find(x):
while x ≠ π(x): x = π(x)
return x

function union(x, y):
r_x = find(x)
r_y = find(y)
if r_x = r_y: return
if rank(r_x) > rank(r_y):
    π(r_y) = r_x
else:
    π(r_y) = r_x
    if rank(r_x) = rank(r_y):
        rank(r_y) = rank(r_y) + 1
        return x

Property: Any root node of rank k has at least $2^k$ nodes in its tree.
Property: If there are n elements overall, there can be at most $n/2^k$ nodes of rank k. Therefore, all trees have height $\leq \log n$. 
Disjoint sets: path compression

- Complexity of Kruskal’s algorithm: $O(|E| \log |V|)$.
  - Sorting edges: $O(|E| \log|E|) \approx O(|E| \log |V|)$.
  - Find + union (2 · |E| times): $O(|E| \log |V|)$.

- How about if the edges are already sorted or sorting can be done in linear time (weights are small)?

- Path compression:

```
function find(x):
if x ≠ π x:
    π(x) = find(π x)
return π(x)
```

Amortized cost of find: $O(1)$
Kruskal’s cost: $O(|E|)$
(if sorting has linear cost)

Minimum Spanning Tree

Any scheme like this works (because of the properties of trees):

```
X = {}
repeat |V| − 1 times:
    pick a set $S \subset V$ for which $X$ has no edges between $S$ and $V − S$
    let $e \in E$ be the minimum-weight edge between $S$ and $V − S$
    $X = X \cup \{e\}$
```

**Prim’s algorithm**: $X$ always forms a subtree and $S$ is the set of $X$’s nodes:
Prim’s algorithm

Graphs © Dept. CS, UPC

Max-flow/min-cut problems

How much water can you pump from source to target?

OpenValve, by JAE HYUN LEE

https://brilliant.org/wiki/max-flow-min-cut-algorithm/

Max-flow/min-cut problems: applications

Model:
- a directed graph $G = (V, E)$.
- Two special nodes $s, t \in V$.
- Capacities $c_e > 0$ on the edges.

Goal: assign a flow $f_e$ to each edge $e$ of the network satisfying:
- $0 \leq f_e \leq c_e$ for all $e \in E$ (edge capacity not exceeded)
- For all nodes $u$ (except $s$ and $t$), the flow entering the node is equal to the flow exiting the node:

\[
\sum_{(u,v) \in E} f_{wu} = \sum_{(u,v) \in E} f_{vu}.
\]

Size of a flow: total quantity sent from $s$ to $t$ (equal to the quantity leaving $s$):

\[
\text{size}(f) = \sum_{(s,u) \in E} f_{su}.
\]
Given a flow, an augmenting path represents a feasible additional flow from $s$ to $t$. An augmenting path can have forward and backward edges.

Augmenting paths can have forward and backward edges.

Given a flow $f$, an augmenting path is a directed path from $s$ to $t$, which consists of edges from $E$, but not necessarily in the same direction. Each of these edges $e$ satisfies exactly one of the following two conditions:

- $e$ is in the same direction as in $E$ (forward) and $f_e < c_e$. The difference $c_e - f_e$ is called the slack of the edge.

- $e$ is in the opposite direction (backward) and $f_e > 0$. It represents the fact that some flow can be borrowed from the current flow.
Ford-Fulkerson algorithm: example

**Graphs © Dept. CS, UPC**

![](flow_graph.png)

```
function Ford-Fulkerson(G,s,t)
    // Input: A directed Graph G(V,E) with edge capacities c_e.
    // s and t and the source and target of the flow.
    // Output: A flow f that maximizes the size of the flow.
    // For each (u,v) ∈ E, f(v,u) represents its flow.
    for all (u,v) ∈ E:
        f(u,v) = c(u,v) // Forward edges
        f(v,u) = 0 // Backward edges
    while there exists a path p = s ⇝ t in the residual graph:
        f(p) = min{f(u,v): (u,v) ∈ p}
        for all (u,v) ∈ p:
            f(u,v) = f(u,v) - f(p)
            f(v,u) = f(v,u) + f(p)
```

Ford-Fulkerson algorithm: complexity

- Finding a path in the residual graph requires $O(|E|)$ time (using BFS or DFS).

- How many iterations (augmenting paths) are required?
  - The worst case is really bad ($C$ being the largest capacity of an edge if only integral values are used).
  - By carefully selecting *fat* augmenting paths (using some variant of Dijkstra's algorithm), the number of iterations is at most $O(|V| \cdot |E|)$.

- Ford-Fulkerson algorithm is $O(|V| \cdot |E|^2)$.
**Max-flow problem**

**Cut:** An \((s, t)\)-cut partitions the nodes into two disjoint groups, \(L\) and \(R\), such that \(s \in L\) and \(t \in R\).

For any flow \(f\) and any \((s, t)\)-cut \((L, R)\):

\[\text{size}(f) \leq \text{capacity}(L, R).\]

**The max-flow min-cut theorem:**

The size of the maximum flow equals the capacity of the smallest \((s, t)\)-cut.

**The augmenting-path theorem:**

A flow is maximum if it admits no augmenting path.

---

**Bipartite matching**

**BOYS**

Aleix
Bernat
Carles
David

**GIRLS**

Aida
Berta
Cristina
Duna

There is an edge between a boy and a girl if they like each other.

Can we pick couples so that everyone has exactly one partner that he/she likes?

Bad matching: if we pick (Aleix, Aida) and (Bernat, Cristina), then we cannot find couples for Berta, Duna, Carles and David.

A **perfect matching** would be: (Aleix, Berta), (Bernat, Duna), (Carles, Aida) and (David, Cristina).

**Question:** can we always guarantee an integer-valued flow?

**Property:** if all edge capacities are integer, then the optimal flow found by Ford-Fulkerson’s algorithm is integral. It is easy to see that the flow of the augmenting path found at each iteration is integral.

---

**Min-cut algorithm**

**Finding a cut with minimum capacity:**

1. Solve the max-flow problem with Ford-Fulkerson.
2. Compute \(L\) as the set of nodes reachable from \(s\) in the residual graph.
3. Define \(R = V - L\).
4. The cut \((L, R)\) is a min-cut.

**Reduced to a max-flow problem with** \(c_e = 1\).