**Dijkstra’s Shortest Path Algorithm**

- Initialize the cost of source to 0, and all the rest of the nodes to $\infty$.
- Initialize set $S$ to be $\emptyset$.
  - $S$ is the set of nodes to which we have a shortest path.
- While $S$ is not all vertices:
  - Select the node $A$ with the lowest cost that is not in $S$ and identify the node as now being in $S$.
  - For each node $B$ adjacent to $A$:
    - If $\text{cost}(A) + [A \to B] < B$'s currently known cost, set $\text{cost}(B) = \text{cost}(A) + [A \to B]$ and $\text{previous}(B) = A$ so that we can remember the path.

---

**Example: Initialization**

- Cost(source) = 0
- Cost(all vertices but source) = $\infty$
- Pick vertex not in $S$ with lowest cost.

**Example: Update Cost neighbors**

- Cost(B) = 2
- Cost(D) = 1
- Cost(C) = 1 + 2 = 3
- Cost(E) = 1 + 2 = 3
- Cost(F) = 1 + 8 = 9
- Cost(G) = 1 + 4 = 5

---

**Example: pick vertex with lowest cost and add it to S**

- Pick vertex not in $S$ with lowest cost, i.e., $v_4$. 

---

**Example: update neighbors**

- Cost(C) = 1 + 2 = 3
- Cost(E) = 1 + 2 = 3
- Cost(F) = 1 + 8 = 9
- Cost(G) = 1 + 4 = 5
Example (Ct’d)

Pick vertex not in S with lowest cost (v2) and update neighbors

Note: cost(v4) not updated since already in S and cost(v5) not updated since it is larger than previously computed

Example: (ct’d)

Pick vertex not in S (v5) with lowest cost and update neighbors

No updating

Example: (ct’d)

Pick vertex not in S with lowest cost (v3) and update neighbors

Previous cost
Cost(v6) = min (8, 5+1) = 6

Example: (ct’d)

Pick vertex not in S with lowest cost (v7) and update neighbors

Cost(v6) = min (8, 5+1) = 6

Example (end)

Pick vertex not in S with lowest cost (v6) and update neighbors

Data Structures

- Adjacency Lists
  - previous cost
  - priority queue pointers
  - next cost
  - Priority queue for finding and deleting lowest cost vertex and for decreasing costs (Binary Heap works)
Time Complexity

- n vertices and m edges
- Initialize data structures $O(n+m)$
- Find min cost vertices $O(n \log n)$
  - n delete mins
- Update costs $O(m \log n)$
  - Potentially m updates
- Update previous pointers $O(m)$
  - Potentially m updates
- Total time $O((n + m) \log n)$ - very fast.
  (can be reduced to $O(m \log n)$ by fib or relaxed heap)

Or... using selection-sort pq

- n vertices and m edges
- Initialize data structures $O(n+m)$
- Find min cost vertices $O(n^2)$
  - n delete mins
- Update costs $O(m)$
  - Potentially m updates
- Update previous pointers $O(m)$
  - Potentially m updates
- Total time $O(n^2+m) = O(n^2)$.

Correctness

- Dijkstra’s algorithm is an example of a greedy algorithm
- Greedy algorithms always make choices that currently seem the best
  - Short-sighted – no consideration of long-term or global issues
  - Locally optimal does not always mean globally optimal
  - In Dijkstra’s case – choose the least cost node, but what if there is another path through other vertices that is cheaper?

“Cloudy” Proof: The Idea

- If the path to G is the next shortest path, the path to P must be at least as long. Therefore, any path through P to G cannot be shorter!

Inside the Cloud (Proof)

- Everything inside the cloud has the correct shortest path
- Proof is by induction on the number of nodes in the cloud:
  - Base case: Initial cloud is just the source with shortest path 0.
  - Inductive hypothesis: Assume that a cloud of k-1 nodes all have shortest paths.
  - Inductive step: choose the least cost node G → has to be the shortest path to G (previous slide). Add k-th node G to the cloud.

All Pairs Shortest Path

- Given a edge weighted directed graph $G = (V, E)$ find for all $u, v \in V$ the length of the shortest path from $u$ to $v$. Use matrix representation.

Inside the cloud has the correct shortest path.
A (simpler) Related Problem: Transitive Closure

- Given a digraph \( G(V,E) \) the \textit{transitive closure} is a digraph \( G'(V',E') \) such that
  - \( V' = V \) (same set of vertices)
  - If \((v_i, v_{i+1}, \ldots, v_k)\) is a path in \( G \), then \((v_i, v_k)\) is an edge of \( E' \)

Unweighted Digraph Boolean Matrix Representation

- \( C \) is called the \textit{connectivity matrix}
  - \( 1 = \text{connected} \)
  - \( 0 = \text{not connected} \)

Unweighted Digraph Boolean Matrix Representation

Finding Paths of Length 2

- First initialize \( C_2 \) to all zero
- For \( k = 1 \) to \( n \)
  - For \( i = 1 \) to \( n \)
    - For \( j = 1 \) to \( n \)
      - \( C_2[i,j] := C_2[i,j] \cup (C[i,k] \cap C[k,j]) \)

- Where \( \cap \) is Boolean AND (&&) and \( \cup \) is Boolean OR (||)
- This means if there is an edge from \( i \) to \( k \)
  AND an edge from \( k \) to \( j \), then there is a path of length 2 between \( i \) and \( j \).

- Column \( k \) \((C[i,k])\) represents the predecessors of \( k \)
- Row \( k \) \((C[k,j])\) represents the successors of \( k \)

Paths of Length 2

- \( C \) is the connectivity matrix
  - \( 1 = \text{connected} \)
  - \( 0 = \text{not connected} \)

- \( C_2 \) is the matrix representing paths of length 2
  - \( 1 = \text{connected} \)
  - \( 0 = \text{not connected} \)

Transitive Closure

- Union of paths of length 0, length 1, length 2, …, length \( n-1 \)
  - Time complexity \( n \times O(n^3) = O(n^4) \)
- There exists a better \( (O(n^3)) \) algorithm: Warshall’s algorithm
**Warshall Algorithm**

TransitiveClosure {
    for k = 1 to n do  // k is the step number //
        for i = 1 to n do
            for j = 1 to n do
                C[i,j] := C[i,j] ∪ (C[i,k] ∩ C[k,j]);
        }
}

where C[i,j] starts as the original connectivity matrix and C[i,j] is updated after step k if a new path from i to j through k is found.

**Proof of Correctness**

Prove: After the k-th time through the loop, C[i,j] = 1 if there is a path from i to j that only passes through vertices numbered 1,2,...,k (except for the initial edges)

- **Base case:** k = 1. C[i,j] = 1 for the initial connectivity matrix (path of length 0) and C[i,j] = 1 if there is a path (i,1,j)

**Inductive Step**

- **Inductive Hypothesis:** Suppose after step k-1 that C[i,j] contains a 1 if there is a path from i to j through vertices 1,...,k-1.

- **Induction:** Consider step k, which does C[i,j] := C[i,j] ∪ (C[i,k] ∩ C[k,j]);

Either C[i,j] is already 1 or there is a new path through vertex k, which makes it 1.
Warshall Algorithm

Back to Weighted graphs: Matrix Representation

- \( C[i,j] \) = the cost of the edge \((i,j)\)
  - \( C[i,j] = 0 \) because no cost to stay where you are
  - \( C[i,j] = \infty \) if no edge from \( i \) to \( j \)

\[
\begin{array}{ccccccc}
  & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & 0 & 2 & 1 & - & - & - \\
2 & 0 & 3 & 10 & - & - & - \\
3 & 4 & 0 & - & 5 & - & - \\
4 & - & - & 2 & - & 8 & 4 \\
5 & - & - & 0 & - & 6 & - \\
6 & - & - & 0 & - & - & - \\
7 & - & - & 1 & 0 & - & - \\
\end{array}
\]
Floyd – Warshall Algorithm

// Start with the cost matrix C
All_Pairs_Shortest_Path {
    for k = 1 to n do
        for i = 1 to n do
            for j = 1 to n do
                C[i,j] := min(C[i,j], C[i,k] + C[k,j]);
}

Note x + ∞ = ∞ by definition

On termination C[i,j] is the length of the shortest path from i to j.

The Computation


## Time Complexity of All Pairs Shortest Path

- **n** is the number of vertices
- Three nested loops. \( O(n^3) \)
  - Shortest paths can be found too
- Repeated Dijkstra's algorithm
  - \( O(n(n + m) \log n) = O(n^3 \log n) \) for dense graphs.
  - Run Dijkstra starting at each vertex.