Graph

- Weighted graph – call it networks
- Shortest path between nodes s and t in a network
  - Directed simple path from s to t with the property that no other such path has a lower weight
  - Negative edges?
- Applications?
The shortest path problem has several different forms:

- **Source-sink SP:**
  - Given two nodes A and B, find the shortest path in the weighted graph from A to B.

- **Single source SP:**
  - Given a node A, find the shortest path from A to every other node in the graph.

- **All Pair SP:**
  - Find the shortest path between every pair of nodes in the graph.
Basic concept in SP

- Relaxation
  - At each step increment the SP information
  - Path relaxation
    - Test if traveling through a given vertex introduces a new shortest path between a pair of vertices
  - Edge relaxation: special case of path relaxation
    - Test if traveling through a given edge gives a new shortest path to its destination vertex
      
      If (wt[w] > wt[v] + e.wt)
      
      \[ \text{If } (\text{wt}[w] > \text{wt}[v] + e.wt) \]
      
      \[ \{ \text{wt}[w] = \text{wt}[v] + e.wt; \text{st}[w] = v; \} \]
Relaxation

Edge relaxation

Path relaxation
Shortest Path

- Property 21.1
  - If a vertex $x$ is on a shortest path from $s$ to $t$, then that path consists of a shortest path from $s$ to $x$ followed by a shortest path from $x$ to $t$

- Dijkstra’s algorithm (similar to Prim’s MST)
  - Start at source
  - Include next edge that gives the shortest path from the source to a vertex not in the SP
Shortest path

To select the next node to visit, we must choose the node in the fringe that has the shortest path to A. The shortest path from the next closest node must immediately go to a visited node.

Visited nodes form a shortest path tree

Fringe node set
Dijkstra’s algorithm

- Complexity?

![Graph with nodes and edges](image)

**Path | Length | Fringe**
--- | --- | ---
1) \(v_0v_2\) | 10 | \(v_0v_1, v_0v_2, v_0v_4\)
2) \(v_0v_2v_3\) | 25 | \(v_0v_1, v_0v_4, v_2v_3\)
3) \(v_0v_2v_3v_1\) | 45 | \(v_0v_1, v_0v_4, v_3v_1\) (\(v_3v_4\)?)
4) \(v_0v_4\) | 45 | \(v_0v_4\)
Dijkstra’s algorithm

#define GRAPHpfs GRAPHspt
#define P (wt[v] + t->wt)
void GRAPHpfs(Graph G, int s, int st[], double wt[])
{
    int v, w; link t;
    PQinit(); priority = wt;
    for (v = 0; v < G->V; v++)
    {
        st[v] = -1; wt[v] = maxWT; PQinsert(v);
    }
    wt[s] = 0.0; PQdec(s);
    while (!PQempty())
    {
        if (wt[v = PQdelmin()] != maxWT)
            for (t = G->adj[v]; t != NULL; t = t->next)
                if (P < wt[w = t->v])
                    { wt[w] = P; PQdec(w); st[w] = v; }
    }
}
All-pairs shortest path

- Use Dijkstra’s algorithm from each vertex
  - Complexity: $VE\log V$
- Floyd’s algorithm
  - Use extension of Warshall’s algorithm for transitive closure
  - Complexity: $V^3$
Floyd-Warshall Algorithm

- \( D_{ij}^{(k)} \) = length of shortest path from \( i \) to \( j \) with intermediate vertices from \{1, 2, ..., k\}:

- Dynamic Programming: recurrence
  - \( D_{ij}^{(0)} = D_{ij} \)
  - \( D_{ij}^{(k)} = \min \{ D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \} \)

intermediate nodes in \{1, 2, ..., k\}
The Floyd-Warshall algorithm

\[
d_{ij}^{(k)} = \begin{cases} 
W_{ij} & \text{if } k = 0 \\
\min( d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} ) & \text{if } k \geq 1 
\end{cases}
\]

Floyd-Warshall(\(W\)) \(\theta(n^3)\)

1. \(n \leftarrow \text{rows}[W]\)
2. \(D^{(0)} = W\)
3. for \(k \leftarrow 1\) to \(n\)
4. do for \(i \leftarrow 1\) to \(n\)
5. do for \(j \leftarrow 1\) to \(n\)
6. \(d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})\)
7. return \(D^{(n)}\)

calculate \(D^{(0)} , D^{(1)} , D^{(2)} , D^{(3)} , D^{(4)} \) and \(D^{(5)}\)
Floyd-Warshall Algorithm

```c
void GRAPHspALL(Graph G)
{
    int i, s, t;
    double **d = MATRIXdouble(G->V, G->V, maxWT);
    int **p = MATRIXint(G->V, G->V, G->V);
    for (s = 0; s < G->V; s++)
        for (t = 0; t < G->V; t++)
            if ((d[s][t] = G->adj[s][t]) < maxWT)
                p[s][t] = t;
    for (i = 0; i < G->V; i++)
        for (s = 0; s < G->V; s++)
            if (d[s][i] < maxWT)
                for (t = 0; t < G->V; t++)
                    if (d[s][t] > d[s][i] + d[i][t])
                        { p[s][t] = p[s][i];
                          d[s][t] = d[s][i] + d[i][t]; }
    G->dist = d; G->path = p;
}
```
Complexity classes

- An algorithm A is of polynomial complexity if there exists a polynomial $p(n)$ such that the computing time of A is $O(p(n))$
- Set $P$
  - set of decision problems solvable in polynomial time using deterministic algorithm
    - **Deterministic**: result of each step is uniquely defined
- Set $NP$
  - set of decision problem solvable in polynomial time using nondeterministic algorithm
    - **Non-deterministic**: result of each step is a set of possibilities
- $P \subseteq NP$
  - Problem is $P = NP$ or $P \neq NP$?
Complexity classes

- Satisfiability is in P iff P = NP
- NP-hard problems
  - Reduction L1 reduces to L2
    - Iff there is a way to solve L1 in deterministic polynomial time algorithm using deterministic algorithm that solves L2 in polynomial time
  - A problem L is NP-hard iff satisfiability reduces to L
- NP-complete
  - L is in NP
  - L is NP-hard
Showing a problem NP-complete

- Show that it is in NP
- Show that it is NP-hard
  - Pick problem L already known to be NP-hard
  - Show that the problem can be reduced to L
- Example
  - Show that traveling salesman problem is NP-complete
  - Known: directed Hamiltonian cycle problem is NP-complete