04-630
Data Structures and Algorithms for Engineers

Lecture 18: Graph Algorithms

Graphs: Shortest Path Algorithms

Outline

- Shortest Path algorithms:
  - Dijkstra’s,
  - Floyd’s
- Applications

Time Complexity

Kruskal

- Create a priority Queue of Edges

- For each edge
  - check if it creates a cycle between vertices

O(ELogE + ELogV)

Prims

- Create a list of vertices called Vertexlist
- Create list for storing the graph called SPT

- Pick a random vertex
  - Add edges to Edgelist

- While vertices in list
  - Search list for minimum weight edge
    - Add to SPT if edge creates no cycles and connects new vertex
  - Add edges from new vertex to Edgelist
  - Remove vertex from Vertexlist

O(V*V)

Shortest paths: preliminaries

- **Path**: sequence of edges connecting two vertices.

- **BFS** returns shortest path in an ***unweighted graph***.
  - **BFS** also returns shortest path if all ***weights are the same*** in a ***weighted graph***.

- In general, the shortest path in a ***weighted graph*** may pass through many intermediate vertices.
  - BFS won't work in such a case.

Shortest paths: preliminaries

Two main algorithms:

- Dijkstra's algorithm:
  - Takes as input start and destination vertices and finds the shortest path between them.
  - Other implementations find the shortest path from a ***start vertex*** and all other vertices, i.e., *a shortest path spanning tree rooted in the start vertex.*
- Floyd's algorithm:
  - Finds the shortest path between ***all pairs*** of vertices in a graph
- We assume ***positive weights*** to avoid ***looping***.

Dijkstra’s algorithm (intuition)

- A greedy algorithm.
- Given (s,…,x,….t) is the shortest part from s to t, then (s,…,x) should be the shortest path from s to x.
- Incrementally build (can be suboptimal)

Dijkstra's algorithm

- A greedy algorithm.
- Uses ***distance/weight/cost*** to determine shortest path from a vertex ***s***.
  - Repeatedly
    - selects the smallest distance/weight/cost,
    - extend the path one edge at a time,
    - until all vertices are included.
- Given (s,…,x,….t) is the shortest part from s to t, then (s,…,x) should be the shortest path from s to x.
- Comparison to Prim's algorithm- similar except:
  - Instead of just considering the weight of the potential edge, it also *considers the distance from the start edge to the vertex from which the edge emanates*.

Size of G – assume vertexes are Numbered with IDS

Dijkstra's algorithm: pseudocode

3 Data structures:
- ProcessedVertex
- Adjacent List
- Distance

- **Dijkstra(G,s,t): //shortest path from s to t**

```
path={s}
for i=1 to n
    distance[i]= ∞
for each edge (s,v)
    distance[v]=w(s,v) #initially, the distances are just weights
last=s //set last vertex to s
while(last!=t)
    select v_next,  such that v_next is the unknown vertex minimizing distance[v]
    for each edge (v_next,x)
        distance[x]=min(distance[x],distance[v_next]+w(v_next,x))
    last=v_next
    path=path U {v_next}
```

Dijkstra’s algorithm: pseudocode

- Time Complexity: O(n * n)
  - Iterate through distance
    - For each iterate through Adjacency List
  - Cost to initialize is small

3 Data structures:
- ProcessedVertex
- Adjacent List
- Distance

Dijkstra’s algorithm: implementation

```
dijkstra(graph *g, int start)
{
    node *temp;
    bool intree[MAXV+1] ;//marks status if vertex is in tree yet
    int distance[MAXV+1];//cost of adding vertex to tree
    int parent[MAXV+1]; //parent vertex
    int current_vertex;// current vertex being processed
    int candidate_vertex; //potential next vertex
    int dist=0;//cheapest cost to enlarge tree
    int weight=0 ; //tree weight
    for(int i=1;i<=nvertices;i++)
    {
        intree[i]=false;
        distance[i]=INT_MAX;
        parent[i]=-1;
    }
    distance[start]=0;
    current_vertex=start;
```

Dikstra’s algorithm: implementation

```
while(!intree[current_vertex])
{
      intree[current_vertex]=true;
      if(current_vertex!=start)
      {
            cout<<"\n\tedge("<<parent[current_vertex]<<","<<current_vertex<<") in tree\n";
            weight=weight+dist;

      }
      temp=adjLists[current_vertex].head;
      while(temp) //get all adjacent vertices
      {
            candidate_vertex=temp->dest;
            if(distance[candidate_vertex]>(distance[current_vertex]+ temp->weight))//difference to Prim’s
            {
                  distance[candidate_vertex]= distance[current_vertex]+ temp->weight;//difference to Prim’s
                  parent[candidate_vertex]=current_vertex;
```

Dikstra’s algorithm: implementation

```
                }
                temp=temp->next;//obtain next adjacent node.
            }//end of while loop accessing the vertices
            current_vertex=1;
            dist=INT_MAX;
            //now pick node with lowest distance
            for(int i=1;i<=nvertices;i++)
            {
                if((!intree[i])&&(dist>distance[i]))
                {
                    dist=distance[i];
                    current_vertex=i;
                }
            }//end for
        }//end loop for intree
        return weight;
}
```

All-pairs shortest path: Floyd’s algorithm

- Suitable for applications like finding the ***center*** or ***diameter*** of a graph, which requires finding shortest path between all pairs of vertices.

- If we run Dijkstra’s n times (once for each start vertex), we achieve this in $O(n^3)$

All-pairs shortest path: Floyd's algorithm

- Find center of graph:
  - Minimize longest and average distance to all other vertices.
  - ***Application***: optimal location for an outlet to serve the greatest number of people.
- Find diameter of a graph:
  - Minimize longest shortest-path distance over all pairs of vertices.
  - ***Application***: communication- determine the longest possible time for a network packet to be delivered.
- Compute the shortest path between all pairs of vertices using an ***nxn distance matrix***.

Floyd’s Algorithm: solution approa

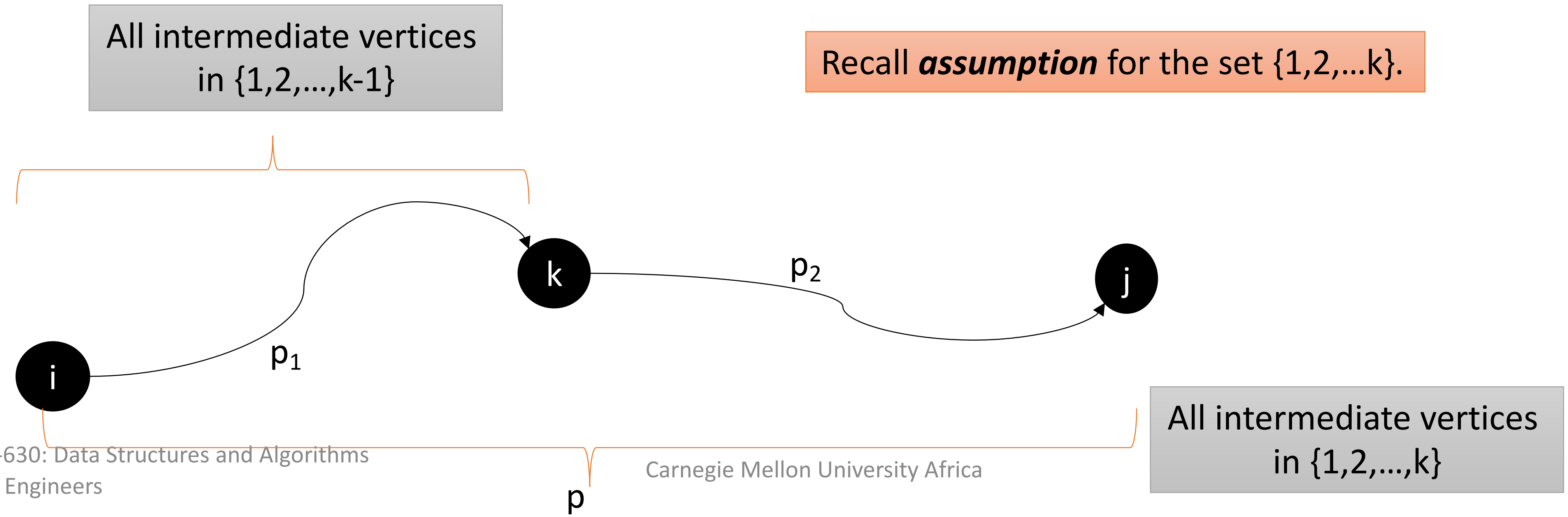
3 Data structures of Dijsktra:
- ProcessedVertex
- AdjacencyList
- Distance

- Simple solution:
  - Call Dijkstra’s algorithm from each of the n possible starting vertices.
  - Takes $O(n^3)$

- Floyd-Warshall algorithm:
  - Construct a n x n shortest path distance matrix directly from n x n weight matrix.
  - Implement using adjacency matrix, instead of adjacency list data structure.

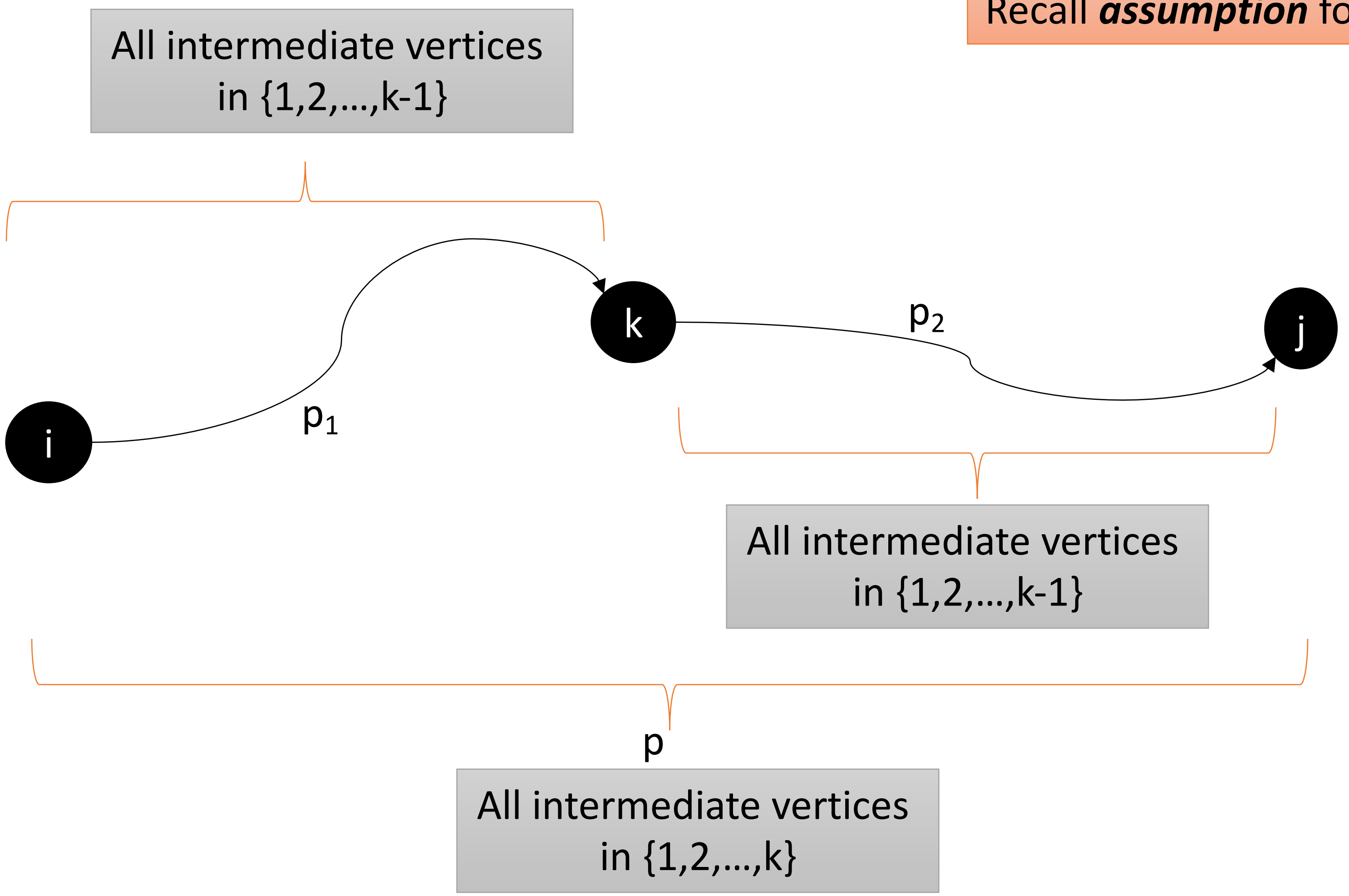
Floyd-Washall algorithm: formulation

- A ***dynamic-programming*** algorithm.
- Considers the ***intermediate vertices*** of a shortest path.
  - ***Intermediate vertex***: An intermediate vertex of a simple path $p=<v_1,v_2,\ldots,v_j>$ is any vertex of p other than v_1 or v_j, i.e. any vertex in the set $\{v_2,v_3,\ldots,v_{j-1}\}$.
- **Assumption**: Assuming the vertices of G are V={1,2,……,n}, let's consider a subset {1,2,…..,k} for some k.
- For any pair of vertices (i,j) ϵ V, considering all paths from i to j, where the intermediate vertices are all drawn from {1,2,….,k}, let p be a *minimum-weight path* from among them.



Floyd-Washall algorithm: formulation

Recall ***assumption*** for the set {1,2,…k}

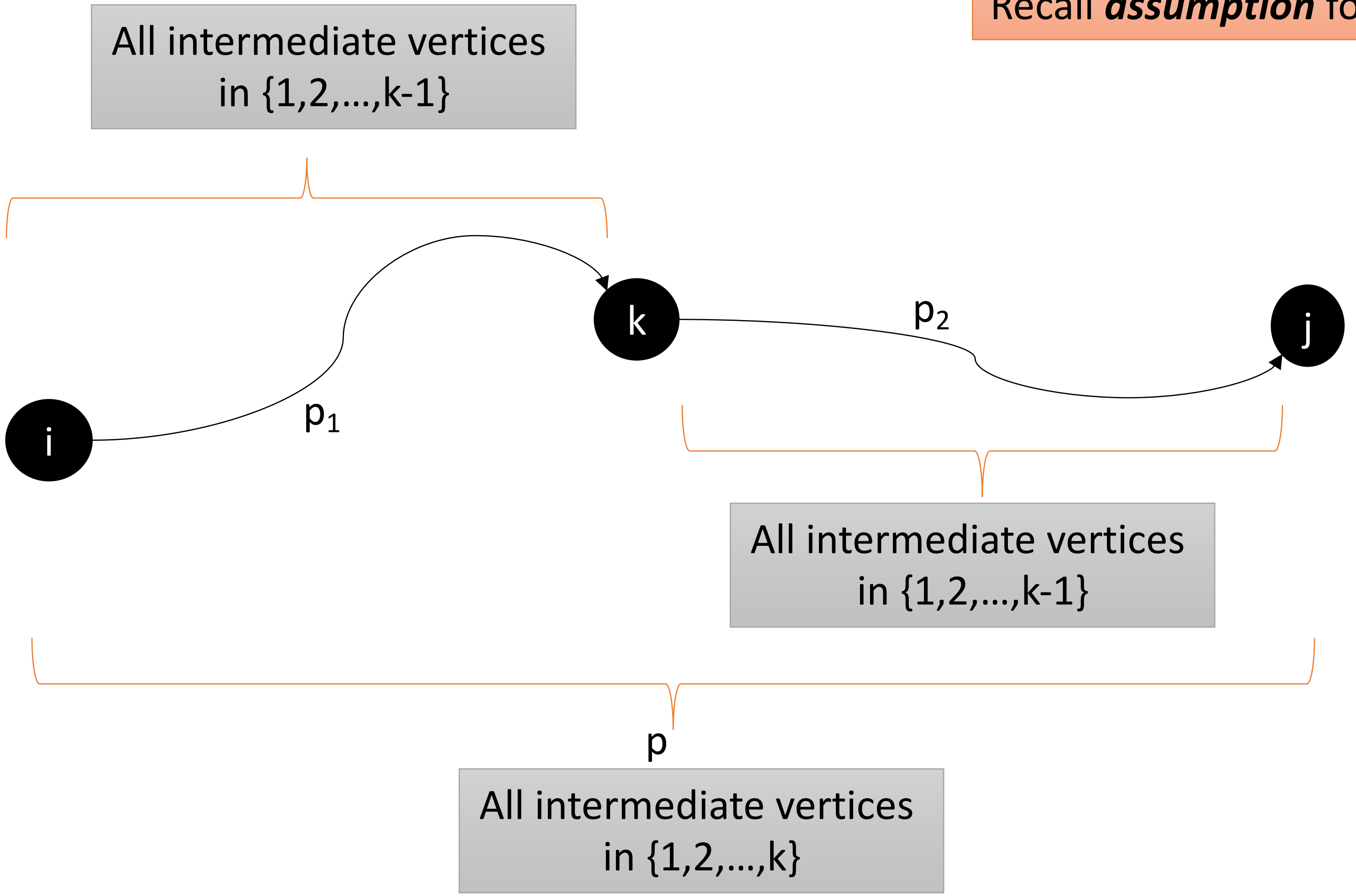


Floyd-Washall algorithm: formulation

- A ***dynamic-programming*** algorithm.
- Considers the ***intermediate vertices*** of a shortest path.
  - ***Intermediate vertex***: An intermediate vertex of a simple path p=<v_1,v_2,….,v_j> is any vertex of p other than v_1 or v_j, i.e. any vertex in the set {v_2,v_3,….v_{j-1}}.
- **Assumption**: Assuming the vertices of G are V={1,2,……,n}, let's consider a subset {1,2,…..,k} for some k.
- For any pair of vertices (i,j) ϵ V, considering all paths from i to j, where the intermediate vertices are all drawn from {1,2,….,k}, let p be a *minimum-weight path* from among them.
- Exploits a relationship between path **p** and shortest paths ***from i to j*** with all intermediate vertices in the set ***{1,2,…,k-1}.***
  - If *k* is **not an** intermediate vertex of path p, then all intermediate vertices of path p are in the set ***{1,2,…,k-1}.***
    - Thus, a shortest path from vertex ***i to vertex j*** with all intermediate vertices in the set ***{1,2,…,k-1}*** is also a shortest path ***from i to j*** with all intermediate vertices in the set ***{1,2,…,k}*** , since ***{1,2,…,k-1}*** $\subseteq$ ***{1,2,…,k}***
  - If k **is an** intermediate vertex of path p, then we decompose **p** into $\mathbf{p_1}$(*from i to k*) and $\mathbf{p_2}$(*from k to j*), with both *p1 and p2 deriving their intermediate vertices* from **{1,2,…,k}-{k}**, i.e. **{1,2,…,k-1}.**

Floyd-Washall algorithm: formulation

Recall ***assumption*** for the set {1,2,…k} two slides back.



Floyd-Washall algorithm: pseudoco

1 Data structures of floyd:

- AdjacencyMatrixes

Floyd-Warshall(W):#W: nxn weight matrix

set n: number of vertices in W

initialize distance matrix $D^{(0)=}$W #initial distance matrix

for k=1 **to** n

let $D^{(k)}=(d_{ij}^{(k)})$ be a new **n x n matrix** *#We will have matrices*

#$D^{(1)}$, $D^{(2)}$….$D^{(n)}$. The final matrix $D^{(n)}$ is returned.

for i=1 **to** n

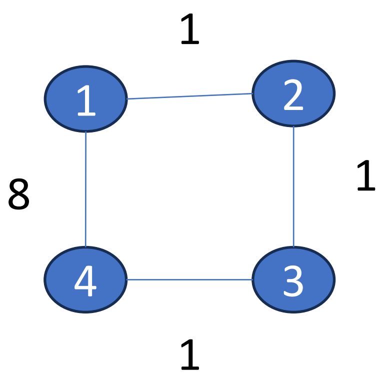
for j=1 **to** n

$d_{ij}^{(k)}= \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)})$

return $D^{(n)}$ #at this point ***k=n***, the final matrix.

initialize distance matrix

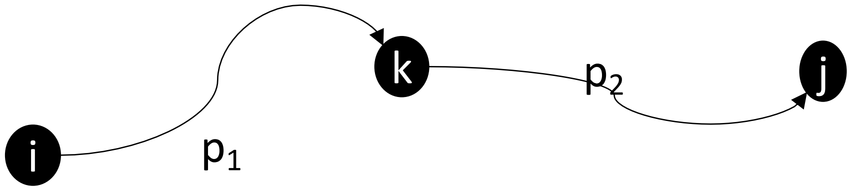
$D^{(0)}=W$ #initial distance matrix



	1	2	3	4
1	0	1	--	8
2	1	0	1	--
3	--	1	0	1
4	8	--	1	0

D(0)

$$d_{ij}^{(k)}= \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)})$$



for k=1 **to** n

D(1)

	1	2	3	4
1	0	1	--	8
2	1	0	1	--
3	--	1	0	
4	8	--		0

$d_{12}^{(1)}= \min(d_{12}^{(0)}, d_{11}^{(0)}+d_{12}^{(0)})$

$d_{13}^{(1)}= \min(d_{13}^{(0)}, d_{11}^{(0)}+d_{13}^{(0)})$

….

$d_{23}^{(1)}= \min(d_{23}^{(0)}, d_{21}^{(0)}+d_{13}^{(0)})$

$d_{24}^{(1)}= \min(d_{24}^{(0)}, d_{21}^{(0)}+d_{14}^{(0)})$

..

$d_{34}^{(1)}= \min(d_{34}^{(0)}, d_{31}^{(0)}+d_{14}^{(0)})$

	1	2	3	4
1	0	1	--	8
2	1	0	1	--
3	--	1	0	
4	8	--		0

	1	2	3	4
1	0	1	--	8
2	1	0	1	--
3	--	1	0	1
4	8	--	1	0

Floyd-Washall algorithm: formulation

- Let $d_{ij}^{(k),}$ be the weight/cost/distance of a shortest path from vertex i to j with all intermediate vertices in {1,2,…,k}.
- For k=0,
- A recursive formulation of shortest path estimates is defined as:
  - $d_{ij}^{(k)=}w_{ij}$, if k=0 {*path with at most one edge; no intermediate vertices*}

  - $d_{ij}^{(k)}=\min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)})$, if k≥1 {*path has intermediate vertices*}.
- Given that for any path, all intermediate vertices are in the set {1,2,…,n}, the matrix $D^{(n)}=(d_{ij}^{(n)})$, gives all shortest pairs.

Floyd’s algorithm: implementation

```
#define MAXV 100

struct adjacency_matrix
{
int weight[MAXV+1][MAXV+1]; //for adjacency or weight information
int nvertices; //number of vertices in graph
};
```

Floyd’s algorithm: implementation

```
void floyd(adjacency_matrix *g){
	int i,j;//counters
	int k; //intermediate vertex counter
	int through_k;//distance through vertex k
	for(k=1;k<=g->nvertices;k++)
	{
		for(i=1;i<=g->nvertices;i++)
		{
			for(j=1;j<=g->nvertices;j++)
			{
				through_k=g->weight[i][k]+g->weight[k][j];
				if(through_k<g->weight[i][j])
				{
						g->weight[i][j]=through_k;
				}
			}
		}
	}
}
```

Comments on Performance

- Dijkstra's:
  - $O(n^2)$ with simple data structures.
  - Constructs actual shortest path between any given pair of vertices.
- Floyd's: $O(n^3)$.
  - No better than ***n*** calls to Dijkstra's but performs better in practice.
  - Does not construct actual shortest path between any given pair of vertices.

Acknowledgement

Adapted from material by Prof. David Vernon

Augmented by material from:

The Algorithm Design Manual 2nd Edition: by Steven Skiena

Introduction to Algorithms, 3rd Edition, Thomas H. Cormen et al. (2009)