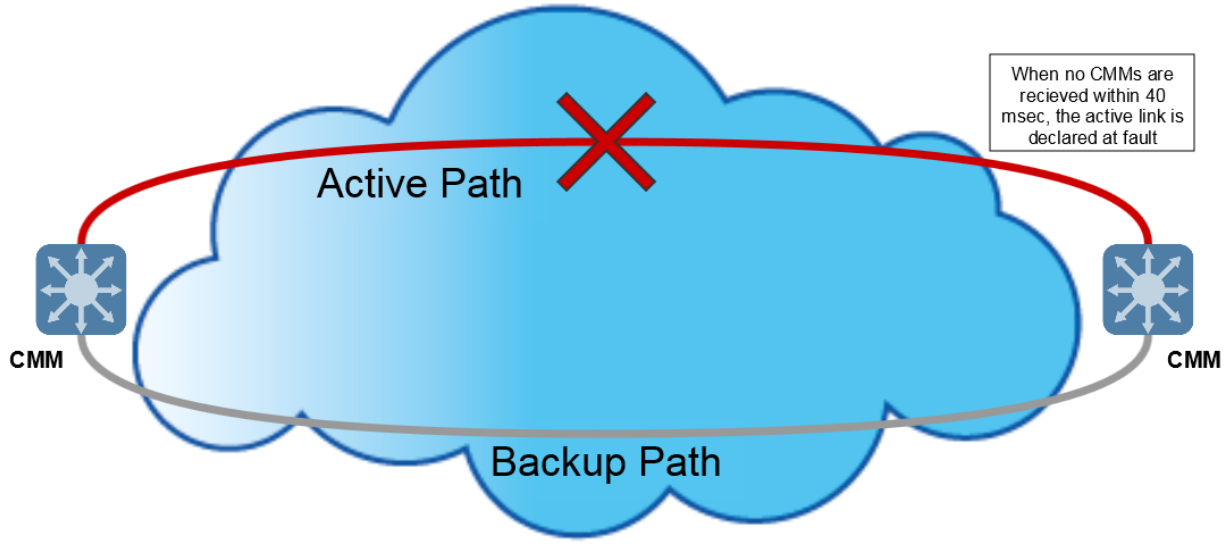
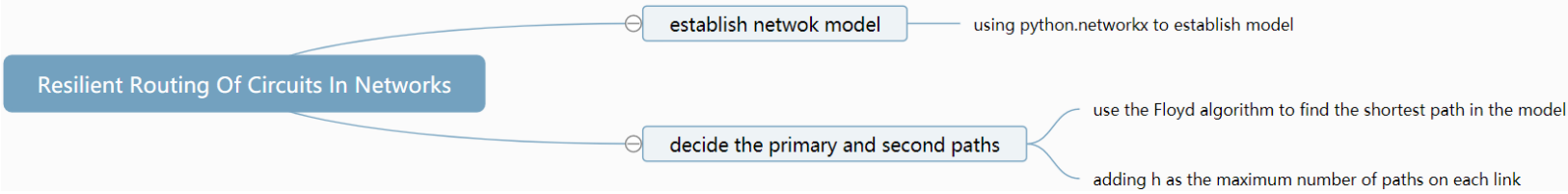
12. RESILIENT ROUTING OF CIRCUITS IN NETWORKS

Consider a connected network G(V,E) in which every edge $e \in E$ has an associated cost K(e). Given a set of source-destination pairs (s_i,d_i), for every pair (s_i,d_i), find a pair of edge-disjoint paths connecting s_i to d_i, such that the total path cost (the cost of a path is the sum of its edge costs) is minimized. Observe that one path (the primary) is used to route the connection, the other (secondary) is used as backup path. (used only in case of failure of the main path). Reconsider the problem by adding the following constraint: at most h primary paths can share the same link.



Questions:

- How much the parameters influence the solution?



Inputs:

N: the number of nodes 1*N array

e: the number of edges 1*N array

$G(V,E)$: the matrix of the network N*N matrix

$k(e)$: the cost of each edge 1*N array

s_i: the source of a path

d_i: the destination of a path

h: the maximum number of paths which can share the same link

Outputs:

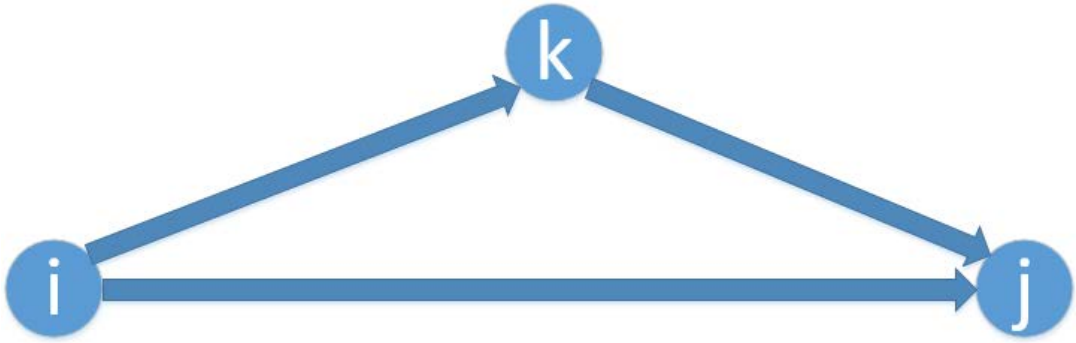
A graph showing the two shortest paths in different color

Primary path(N,N) : the shortest path of nodes N*N matrix

Secondary path(N,N):the backup path which is edge-disjoint from primary path N*N matrix

Cost sum: the cost of each path 1*N array

Objective Function:



$d_{ij}(k)$ = cost (i, j) when k=0;

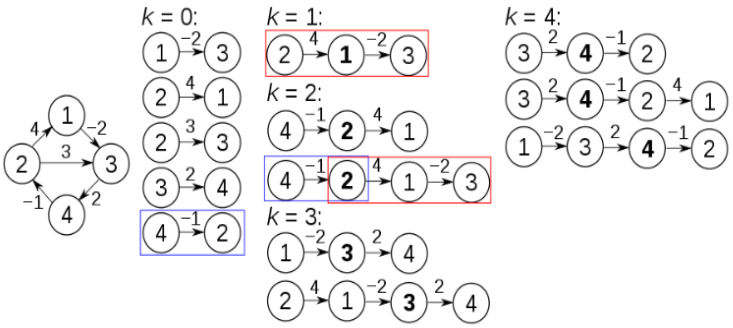
$d_{ij}(k)$ = min($d_{ij}(k-1)$, $d_{ik}(k-1)+d_{kj}(k-1)$) when k>0;

Constrain:

at most h primary paths can share the same link, every time we use this link as primary path, h is decreased by 1 until it goes to 0, this link can not be used again, we delete this edge from G(V,E).

Algorithm:

The algorithm above is executed on the graph on the left below:



Prior to the first recursion of the outer loop, labeled $k = 0$ above, the only known paths correspond to the single edges in the graph. At $k = 1$, paths that go through the vertex 1 are found: in particular, the path [2,1,3] is found, replacing the path [2,3] which has fewer edges but is longer (in terms of weight). At $k = 2$, paths going through the vertices {1,2} are found. The red and blue boxes show how the path [4,2,1,3] is assembled from the two known paths [4,2] and [2,1,3] encountered in previous iterations, with 2 in the intersection. The path [4,2,3] is not considered, because [2,1,3] is the shortest path encountered so far from 2 to 3. At $k = 3$, paths going through the vertices {1,2,3} are found. Finally, at $k = 4$, all shortest paths are found.

The distance matrix at each iteration of k, with the updated distances in **bold**, will be:

$k = 0$

i \ j	1	2	3	4
1	0	∞	−2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	∞	−1	∞	0

$k = 1$

i \ j	1	2	3	4
1	0	∞	−2	∞
2	4	0	**2**	∞
3	∞	∞	0	2
4	∞	−1	∞	0

$k = 2$

i \ j	1	2	3	4
1	0	∞	−2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	**3**	−1	**1**	0

$k = 3$

i \ j	1	2	3	4
1	0	∞	−2	**0**
2	4	0	2	**4**
3	∞	∞	0	2
4	3	−1	1	0

$k = 4$

i \ j	1	2	3	4
1	0	**−1**	−2	0
2	4	0	2	4
3	**5**	**1**	0	2
4	3	−1	1	0

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u, v) do
    dist[u][v] ← w(u, v)    // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```