Lecture 8
Topological Sort, Critical Path, Shortest Path

陆清怡

2022.12.3

991《数据结构与算法》考纲

7、图

（1）图的基本概念。

（2）图的存储，包括邻接矩阵法、邻接表法。

（3）图的遍历操作，包括深度优先搜索、广度优先搜索。

（4）最小生成树，最短路径，关键路径、拓扑排序算法的原理与实现。

Topological Sort

AOV 网：若用 DAG 图表示一个工程，其顶点表示活动，用有向边<V_i, V_j>表示活动 V_i 必须先于活动 V_j 进行的这样一种关系，则将这种有向图称为顶点表示活动的网络，记为 AOV 网。在 AOV 网中，活动 V_i 是活动 V_j 的直接前驱，活动 V_j 是活动 V_i 的直接后继，这种前驱和后继关系具有传递性，且任何活动 V_i 不能以它自己作为自己的前驱或后继。

拓扑排序：在图论中，由一个有向无环图的顶点组成的序列，当且仅当满足下列条件时，称为该图的一个拓扑排序：

① 每个顶点出现且只出现一次。

② 若顶点 A 在序列中排在顶点 B 的前面，则在图中不存在从顶点 B 到顶点 A 的路径。

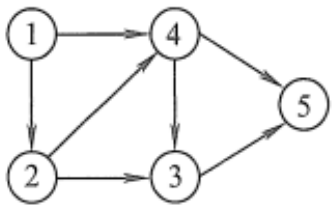
或定义为：拓扑排序是对有向无环图的顶点的一种排序，它使得若存在一条从顶点 A 到顶点 B 的路径，则在排序中顶点 B 出现在顶点 A 的后面。每个 AOV 网都有一个或多个拓扑排序序列。

对一个 AOV 网进行拓扑排序的算法有很多，下面介绍比较常用的一种方法的步骤：

① 从 AOV 网中选择一个没有前驱的顶点并输出。

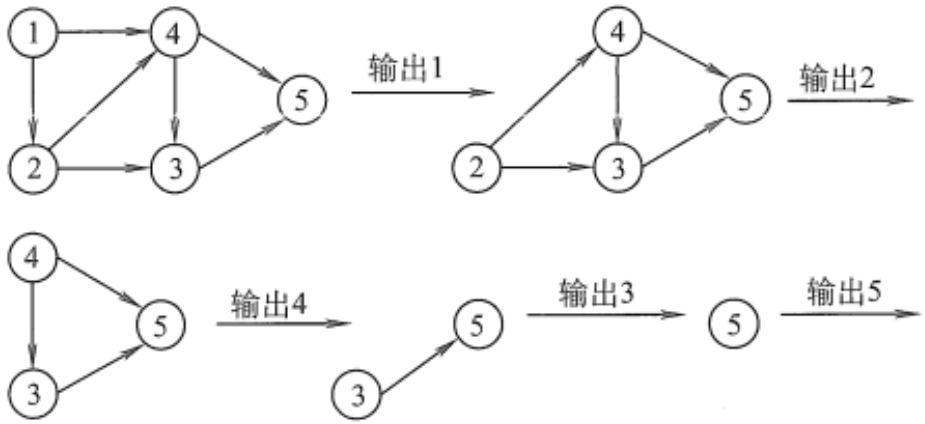
② 从网中删除该顶点和所有以它为起点的有向边。

③ 重复①和②直到当前的 AOV 网为空或当前网中不存在无前驱的顶点为止。后一种情况说明有向图中必然存在环。



结点号	1	2	3	4	5
初始入度	0	1	2	2	2
第一轮		0	2	1	2
第二轮			1	0	2
第三轮			0		1
第四轮					0
第五轮					

(a)



(b)

```
bool TopologicalSort(Graph G){
    InitStack(S);                    //初始化栈，存储入度为 0 的顶点
    for(int i=0;i<G.vexnum;i++)
        if(indegree[i]==0)
            Push(S,i);               //将所有入度为 0 的顶点进栈
    int count=0;                     //计数，记录当前已经输出的顶点数
    while(!IsEmpty(S)){              //栈不空，则存在入度为 0 的顶点
        Pop(S,i);                    //栈顶元素出栈
        print[count++]=i;            //输出顶点 i
        for(p=G.vertices[i].firstarc;p;p=p->nextarc){
        //将所有 i 指向的顶点的入度减 1，并且将入度减为 0 的顶点压入栈 S
            v=p->adjvex;
            if(!(--indegree[v]))
                Push(S,v);           //入度为 0，则入栈
        }
    }//while
    if(count<G.vexnum)
        return  false;               //排序失败，有向图中有回路
    else
        return  true;                //拓扑排序成功
}
```

由于输出每个顶点的同时还要删除以它为起点的边，故拓扑排序的时间复杂度为 $O(|V|+|E|)$。

用拓扑排序算法处理 AOV 网时，应注意以下问题：

① 入度为零的顶点，即没有前驱活动的或前驱活动都已经完成的顶点，工程可以从这个顶点所代表的活动开始或继续。

② 若一个顶点有多个直接后继，则拓扑排序的结果通常不唯一；但若各个顶点已经排在一个线性有序的序列中，每个顶点有唯一的前驱后继关系，则拓扑排序的结果是唯一的。

③ 由于 AOV 网中各顶点的地位平等，每个顶点编号是人为的，因此可以按拓扑排序的结果重新编号，生成 AOV 网的新的邻接存储矩阵，这种邻接矩阵可以是三角矩阵；但对于一般的图来说，若其邻接矩阵是三角矩阵，则存在拓扑序列；反之则不一定成立。

Critical Path

在带权有向图中，以顶点表示事件，以有向边表示活动，以边上的权值表示完成该活动的开销（如完成活动所需的时间），称之为用边表示活动的网络，简称 AOE 网。AOE 网和 AOV 网都是有向无环图，不同之处在于它们的边和顶点所代表的含义是不同的，AOE 网中的边有权值；而 AOV 网中的边无权值，仅表示顶点之间的前后关系。

AOE 网具有以下两个性质：

① 只有在某顶点所代表的事件发生后，从该顶点出发的各有向边所代表的活动才能开始；

② 只有在进入某顶点的各有向边所代表的活动都已结束时，该顶点所代表的事件才能发生。

在 AOE 网中仅有一个入度为 0 的顶点，称为开始顶点（源点），它表示整个工程的开始；网中也仅存在一个出度为 0 的顶点，称为结束顶点（汇点），它表示整个工程的结束。

在 AOE 网中，有些活动是可以并行进行的。从源点到汇点的有向路径可能有多条，并且这些路径长度可能不同。完成不同路径上的活动所需的时间虽然不同，但是只有所有路径上的活动都已完成，整个工程才能算结束。因此，从源点到汇点的所有路径中，具有最大路径长度的路径称为关键路径，而把关键路径上的活动称为关键活动。

完成整个工程的最短时间就是关键路径的长度，即关键路径上各活动花费开销的总和。这是因为关键活动影响了整个工程的时间，即若关键活动不能按时完成，则整个工程的完成时间就会延长。因此，只要找到了关键活动，就找到了关键路径，也就可以得出最短完成时间。

1．事件 v_k 的最早发生时间 $ve(k)$

它是指从源点 v_1 到顶点 v_k 的最长路径长度。事件 v_k 的最早发生时间决定了所有从 v_k 开始的活动能够开工的最早时间。可用下面的递推公式来计算：

ve(源点) = 0

$ve(k) = \text{Max}\{ve(j) + \text{Weight}(v_j, v_k)\}$，$v_k$ 为 v_j 的任意后继，Weight (v_j, v_k)表示<v_j, v_k>上的权值

计算 $ve()$值时，按从前往后的顺序进行，可以在拓扑排序的基础上计算：

① 初始时，令 $ve[1...n] = 0$。

② 输出一个入度为 0 的顶点 v_j 时，计算它所有直接后继顶点 v_k 的最早发生时间，若 $ve[j] + \text{Weight}(v_j, v_k) > ve[k]$，则 $ve[k] = ve[j] + \text{Weight}(v_j, v_k)$。以此类推，直至输出全部顶点。

2．事件 v_k 的最迟发生时间 $vl(k)$

它是指在不推迟整个工程完成的前提下，即保证它的后继事件 v_j 在其最迟发生时间 $vl(j)$能够发生时，该事件最迟必须发生的时间。可用下面的递推公式来计算：

vl(汇点) = ve(汇点)

$vl(k) = \text{Min}\{vl(j) - \text{Weight}(v_k, v_j)\}$，$v_k$ 为 v_j 的任意前驱

注意：在计算 $vl(k)$时，按从后往前的顺序进行，可以在逆拓扑排序的基础上计算。

计算 $vl()$值时，按从后往前的顺序进行，在上述拓扑排序中，增设一个栈以记录拓扑序列，拓扑排序结束后从栈顶至栈底便为逆拓扑有序序列。过程如下：

① 初始时，令 $vl[1...n] = ve[n]$。

② 栈顶顶点 v_j 出栈，计算其所有直接前驱顶点 v_k 的最迟发生时间，若 $vl[j] - \text{Weight}(v_k, v_j) < vl[k]$，则 $vl[k] = vl[j] - \text{Weight}(v_k, v_j)$。以此类推，直至输出全部栈中顶点。

3．活动 a_i 的最早开始时间 $e(i)$

它是指该活动弧的起点所表示的事件的最早发生时间。若边$<v_k, v_j>$表示活动 a_i，则有 $e(i) = ve(k)$。

4．活动 a_i 的最迟开始时间 $l(i)$

它是指该活动弧的终点所表示事件的最迟发生时间与该活动所需时间之差。若边$<v_k, v_j>$表示活动 a_i，则有 $l(i) = vl(j) - \text{Weight}(v_k, v_j)$。

5．一个活动 a_i 的最迟开始时间 $l(i)$和其最早开始时间 $e(i)$的差额 $d(i) = l(i) - e(i)$

它是指该活动完成的时间余量，即在不增加完成整个工程所需总时间的情况下，活动 a_i 可以拖延的时间。若一个活动的时间余量为零，则说明该活动必须要如期完成，否则就会拖延整个工程的进度，所以称 $l(i) - e(i) = 0$ 即 $l(i) = e(i)$的活动 a_i 是关键活动。

求关键路径的算法步骤如下：

1）从源点出发，令 $ve(源点)=0$，按拓扑有序求其余顶点的最早发生时间 $ve()$。

2）从汇点出发，令 $vl(汇点)=ve(汇点)$，按逆拓扑有序求其余顶点的最迟发生时间 $vl()$。

3）根据各顶点的 $ve()$值求所有弧的最早开始时间 $e()$。

4）根据各顶点的 $vl()$值求所有弧的最迟开始时间 $l()$。

5）求 AOE 网中所有活动的差额 $d()$，找出所有 $d()=0$ 的活动构成关键路径。

1）求 $ve()$：初始 $ve(1)=0$，在拓扑排序输出顶点过程中，求得 $ve(2)=3$，$ve(3)=2$，$ve(4)=\max\{ve(2)+2, ve(3)+4\}=\max\{5,6\}=6$，$ve(5)=6$，$ve(6)=\max\{ve(5)+1, ve(4)+2, ve(3)+3\}=\max\{7,8,5\}=8$。

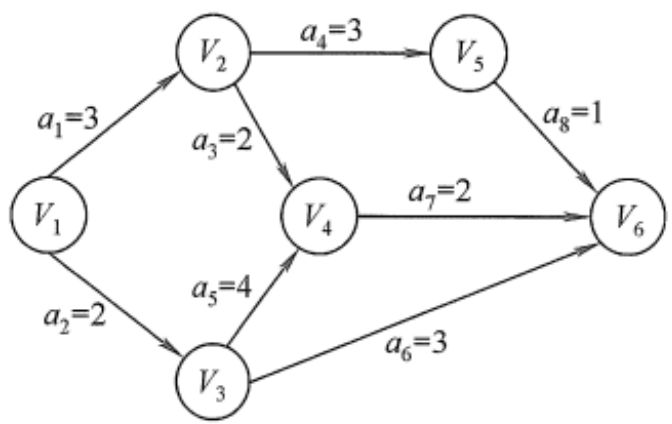
如果这是一道选择题，根据上述求 $ve()$的过程就已经能知道关键路径。

2）求 $vl()$：初始 $vl(6)=8$，在逆拓扑排序出栈过程中，求得 $vl(5)=7$，$vl(4)=6$，$vl(3)=\min\{vl(4)-4, vl(6)-3\}=\min\{2,5\}=2$，$vl(2)=\min\{vl(5)-1, vl(4)-2\}=\min\{4,4\}=4$，$vl(1)$必然为 0 而无须再求。

3）弧的最早开始时间 $e()$等于该弧的起点的顶点的 $ve()$，求得结果如下表所示。

4）弧的最迟开始时间 $l(i)$等于该弧的终点的顶点的 $vl()$减去该弧持续的时间，求得结果如下表所示。

5）根据 $l(i)-e(i)=0$ 的关键活动，得到的关键路径为(v_1, v_3, v_4, v_6)。



	v_1	v_2	v_3	v_4	v_5	v_6
$ve(i)$	0	3	2	6	6	8
$vl(i)$	0	4	2	6	7	8

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
$e(i)$	0	0	3	3	2	2	6	6
$l(i)$	1	0	4	4	2	5	6	7
$l(i)-e(i)$	1	0	1	1	0	3	0	1

对于关键路径，需要注意以下几点：

1）关键路径上的所有活动都是关键活动，它是决定整个工程的关键因素，因此可通过加快关键活动来缩短整个工程的工期。但也不能任意缩短关键活动，因为一旦缩短到一定的程度，该关键活动就可能会变成非关键活动。

2）网中的关键路径并不唯一，且对于有几条关键路径的网，只提高一条关键路径上的关键活动速度并不能缩短整个工程的工期，只有加快那些包括在所有关键路径上的关键活动才能达到缩短工期的目的。

Shortest Path

Shortest Path

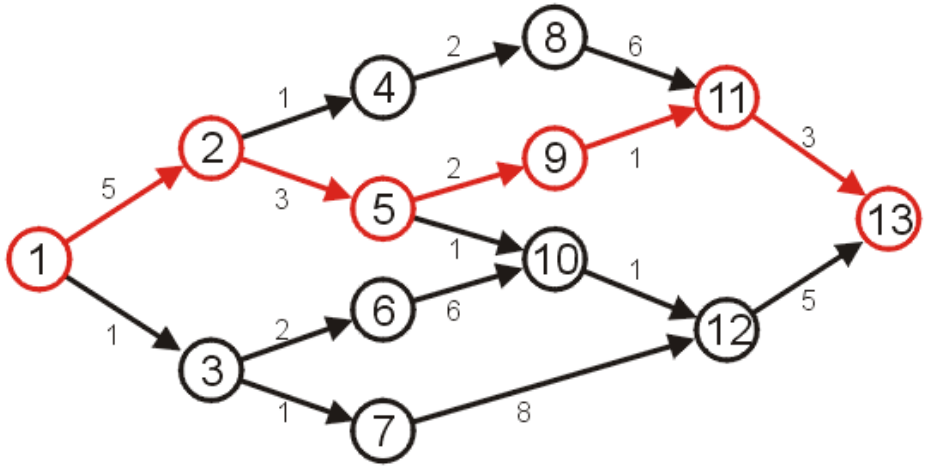
Given a weighted directed graph, one common problem is finding the shortest path between two given vertices

- Recall that in a weighted graph, the *length* of a path is the sum of the weights of each of the edges in that path

Shortest Path

Given the graph, suppose we wish to find the shortest path from vertex 1 to vertex 13

After some consideration, we may determine that the shortest path is as follows, with length 14



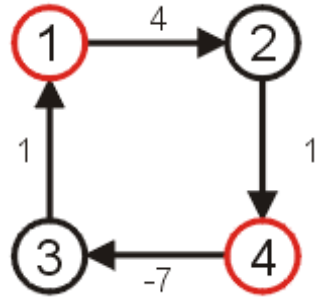
Other paths exists, but they are longer

Shortest Path

The goal is to find the shortest path and its length

We will make the assumption that the weights on all edges is a positive number

- Why this assumption?
- If we have negative weights, it may be possible to end up in a cycle whereby each pass through the cycle decreases the total *length*
- Thus, a shortest length would be undefined for such a graph
- Consider the shortest path from vertex 1 to 4...



Algorithms

Algorithms for finding the shortest path include:

- Dijkstra’s algorithm
- A* search algorithm
- Bellman-Ford algorithm
- Floyd-Warshall algorithm

Shortest Path -Dikjstra

Dijkstra’s algorithm

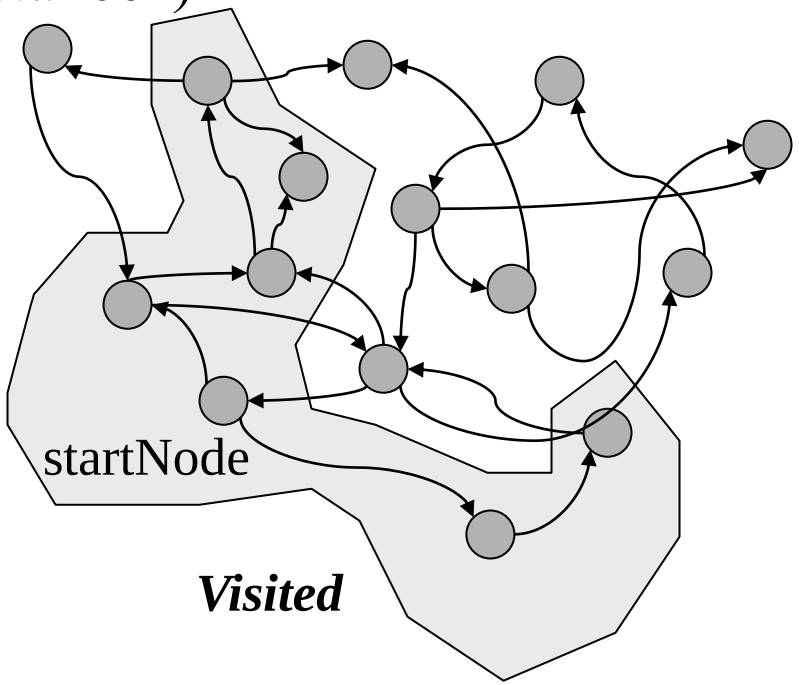
Dijkstra’s algorithm solves the single-source shortest path problem

- It is very similar to Prim’s algorithm
- Assumption: all the weights are positive

Strategy

In general

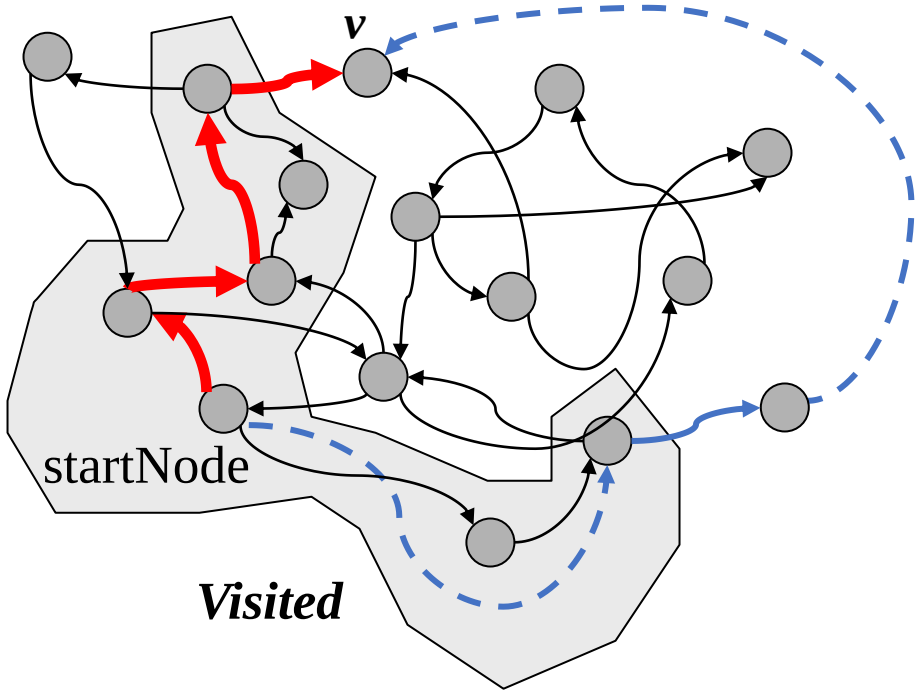
- We know the shortest distance to some of the vertices (marked as visited)
- We also know the shortest distance to each unvisited vertex **through visited vertices** (call this the “known distance”)



Strategy

Consider the unvisited vertex ***v*** that has the shortest known distance

- We are guaranteed that the known distance to ***v*** is the shortest distance from the start node to it
- Proof by contradiction



Dijkstra's algorithm

We need to track the known shortest distance to each vertex

- We require an array of distances, all initialized to infinity except for the source vertex, which is initialized to 0

Do we need to track the shortest path to each vertex?

- Ex: do I have to store (A, B, F) as the shortest path to vertex F?
- No. We only have to record that the shortest path to vertex F came from vertex B
  - The shortest path to F is the shortest path to B followed by the edge (B, F)
- Thus, we need an array of previous vertices, all initialized to null

We need to track visited vertices whose shortest paths have been found

- a Boolean table of size $|V|$

Dijkstra's algorithm

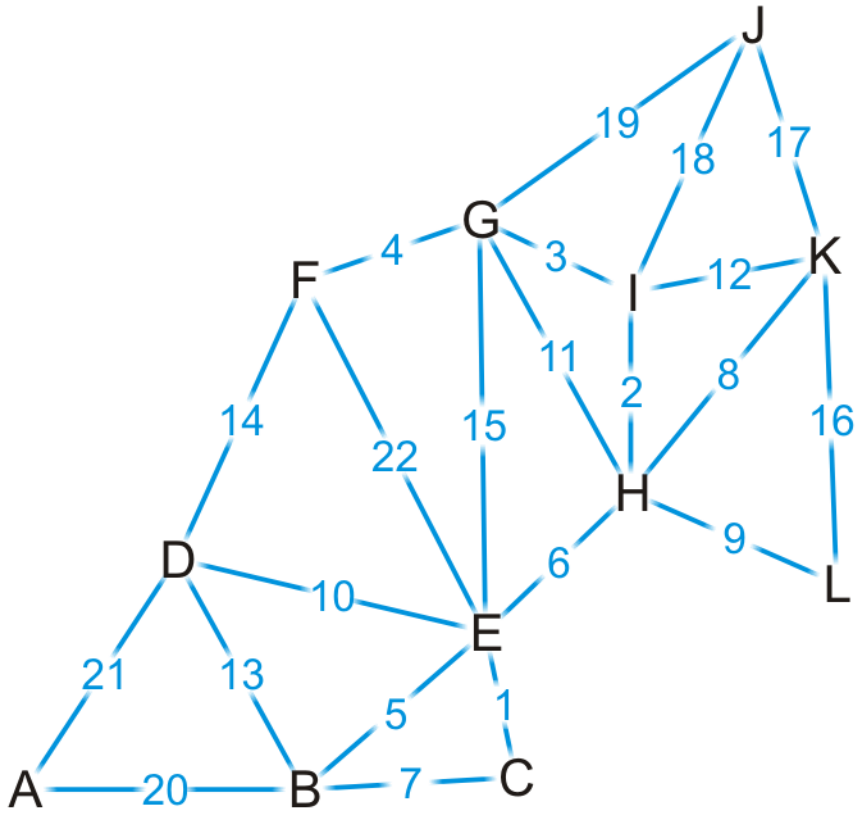
We will iterate $|V|$ times:

- Find the unvisited vertex ***v*** that has a minimum distance to it
- Mark it as visited
- Consider its every adjacent vertex ***w*** that is unvisited:
  - Is the distance to ***v*** plus the weight of the edge (***v***, ***w***) less than our currently known shortest distance to ***w*** ?
  - If so, update the shortest distance to ***w*** and record ***v*** as the previous pointer

Continue iterating until all vertices are visited or all remaining vertices have a distance of infinity

Example

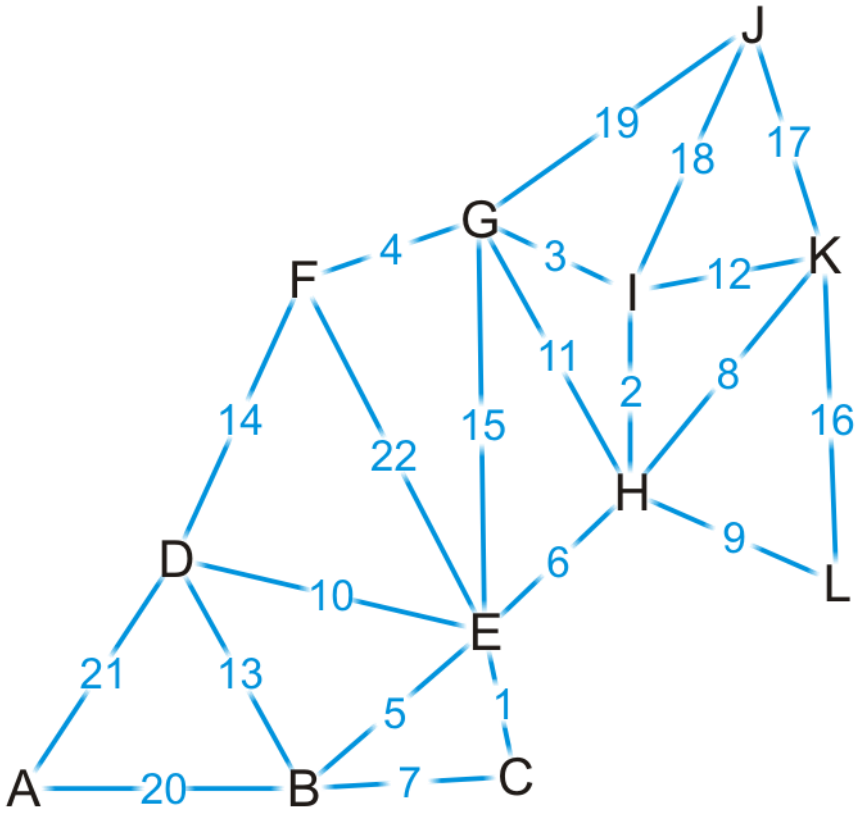
Find the shortest distance from K to every other vertex (BFS?)



Example

We set up our table

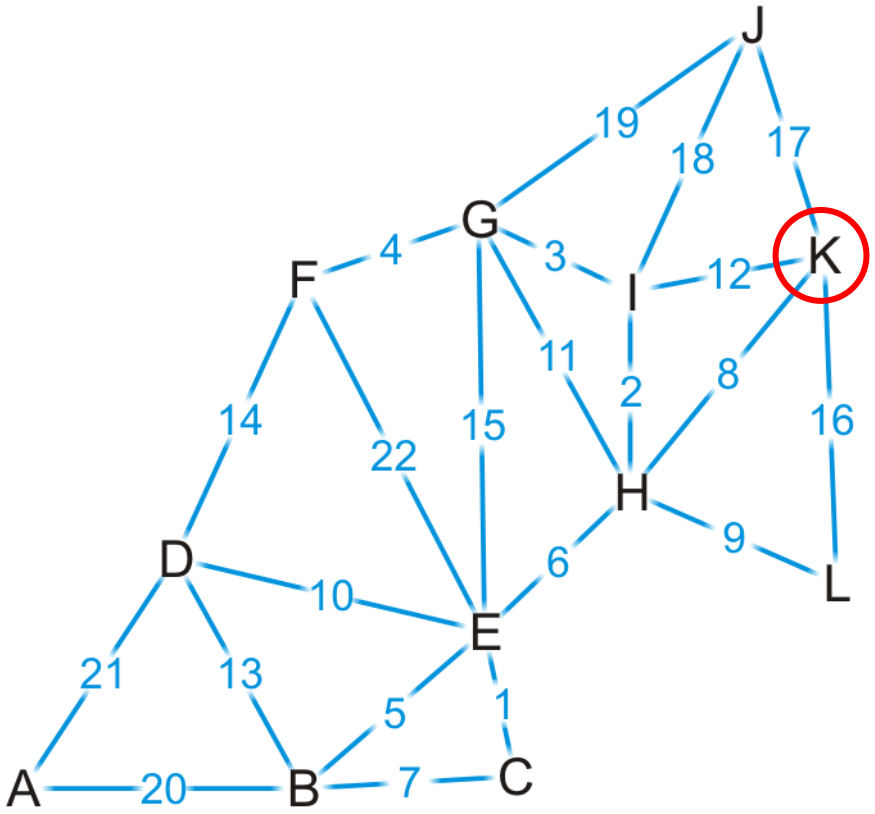
- Which unvisited vertex has the minimum distance to it?



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	∞	Ø
I	F	∞	Ø
J	F	∞	Ø
K	F	0	Ø
L	F	∞	Ø

Example

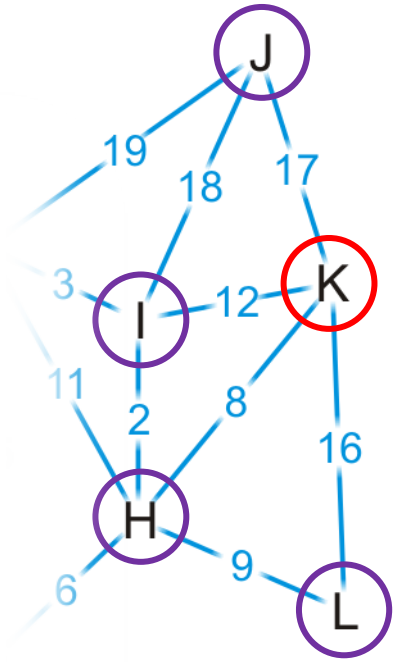
We visit vertex K



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	∞	Ø
I	F	∞	Ø
J	F	∞	Ø
K	**T**	**0**	**Ø**
L	F	∞	Ø

Example

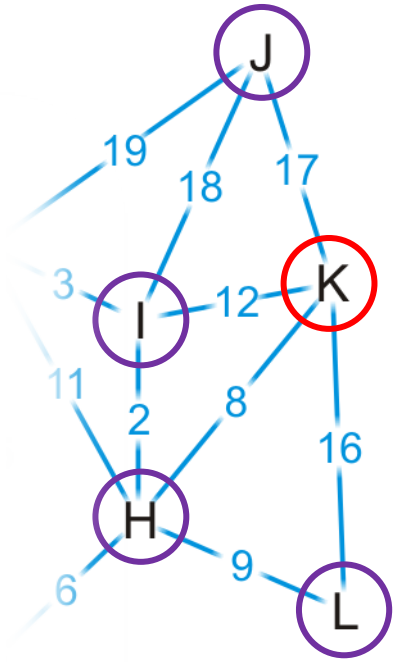
Vertex K has four neighbors: H, I, J and L



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	∞	Ø
I	F	∞	Ø
J	F	∞	Ø
K	**T**	**0**	**Ø**
L	F	∞	Ø

Example

We have now found at least one path to each of these vertices

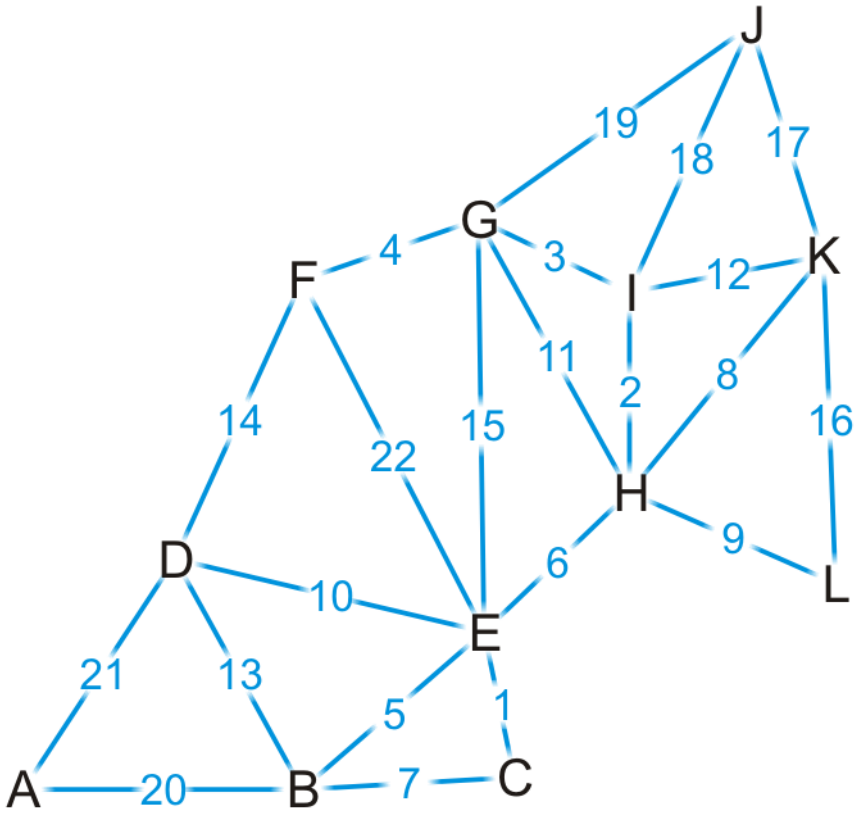


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	**F**	**8**	**K**
I	**F**	**12**	**K**
J	**F**	**17**	**K**
K	**T**	**0**	**Ø**
L	**F**	**16**	**K**

Example

We’re finished with vertex K

- To which vertex are we now guaranteed we have the shortest path?

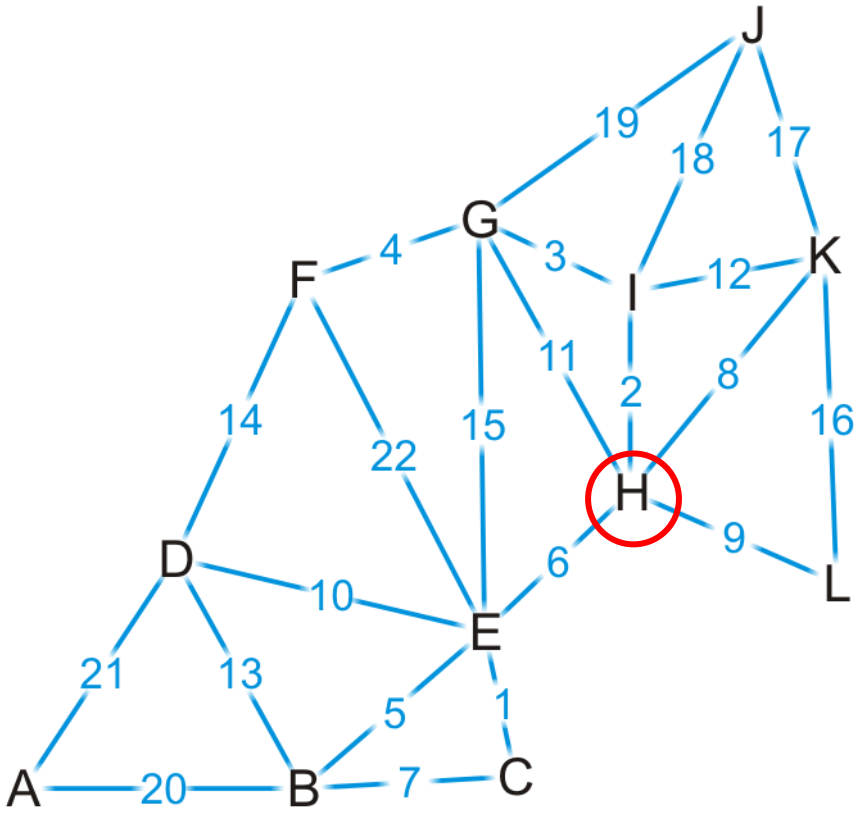


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	8	K
I	F	12	K
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We visit vertex H: the shortest path is (K, H) of length 8

- Vertex H has four unvisited neighbors: E, G, I, L



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	**T**	**8**	**K**
I	F	12	K
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Consider these paths:

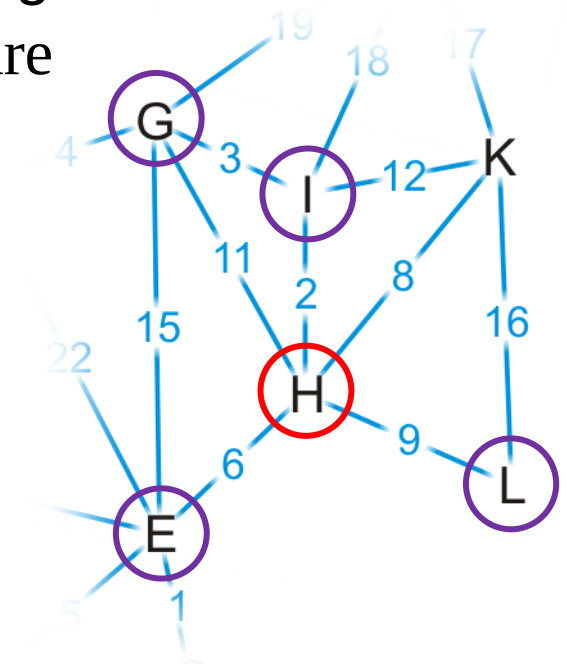
(K, H, E) of length 8 + 6 = 14

(K, H, I) of length 8 + 2 = 10

(K, H, G) of length 8 + 11 = 19

(K, H, L) of length 8 + 9 = 17

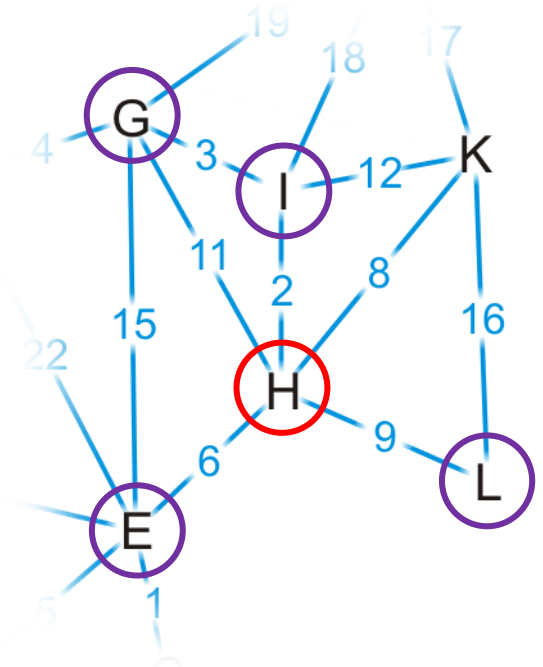
- Which of these are shorter than any known path?



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	**T**	**8**	**K**
I	F	12	K
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We already have a shorter path (K, L), but we update the other three

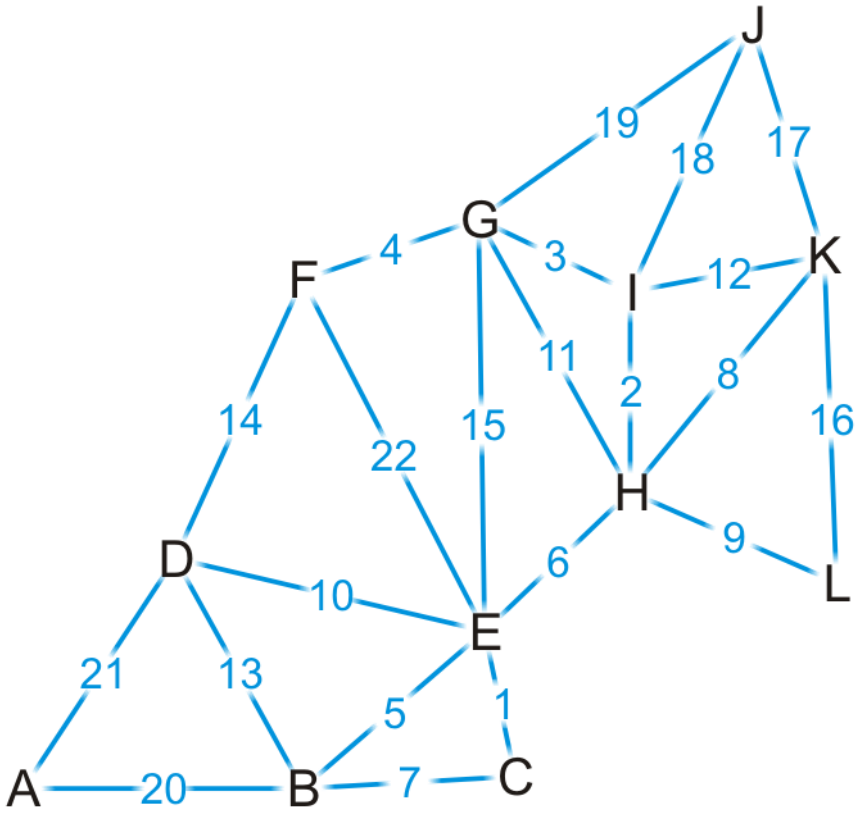


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	**14**	**H**
F	F	∞	Ø
G	F	**19**	**H**
H	**T**	**8**	**K**
I	F	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We are finished with vertex H

- Which vertex do we visit next?

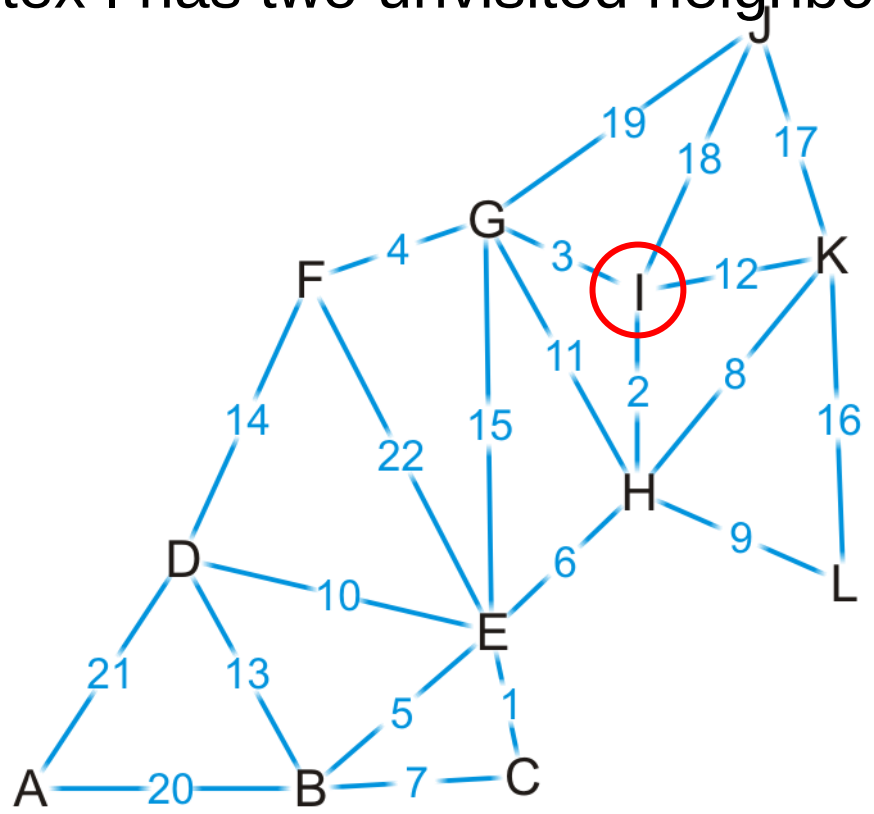


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	F	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

The path (K, H, I) is the shortest path from K to I of length 10

- Vertex I has two unvisited neighbors: G and J

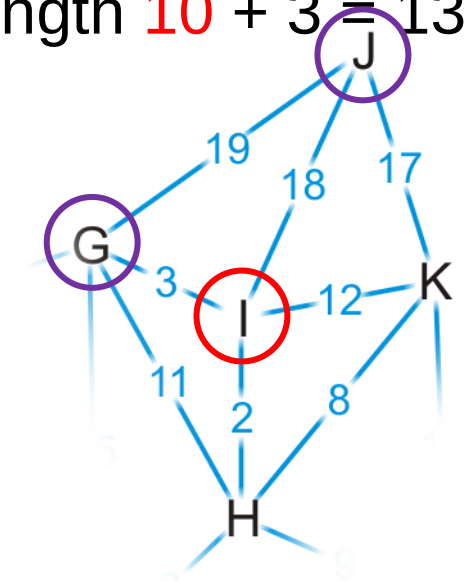


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	**T**	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Consider these paths:

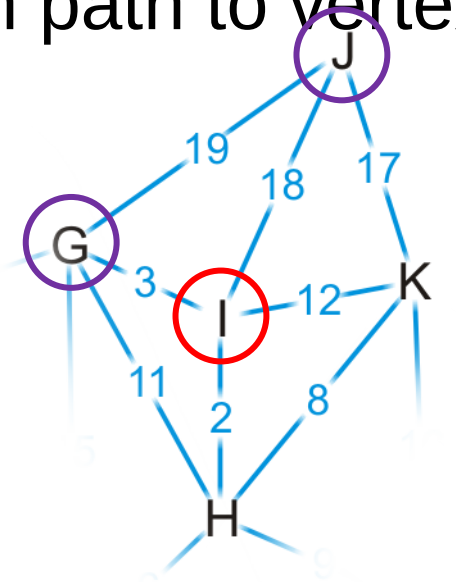
(K, H, I, G) of length 10 + 3 = 13 (K, H, I, J) of length 10 + 18 = 28



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	**T**	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

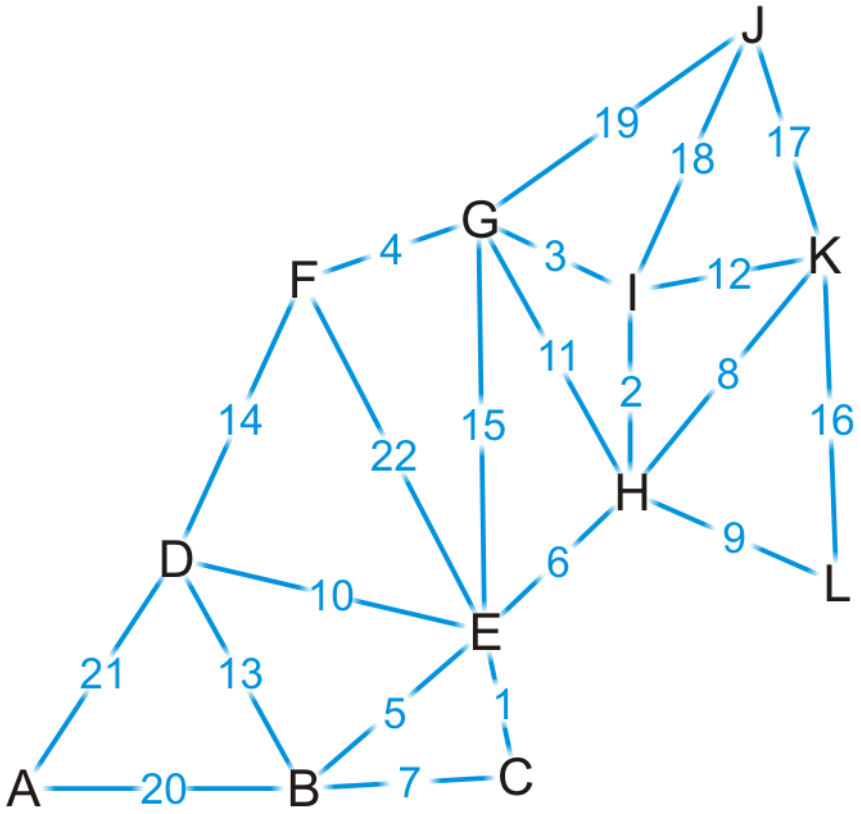
We have discovered a shorter path to vertex G, but (K, J) is still the shortest known path to vertex J



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	**13**	**I**
H	T	8	K
I	**T**	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Which vertex can we visit next?

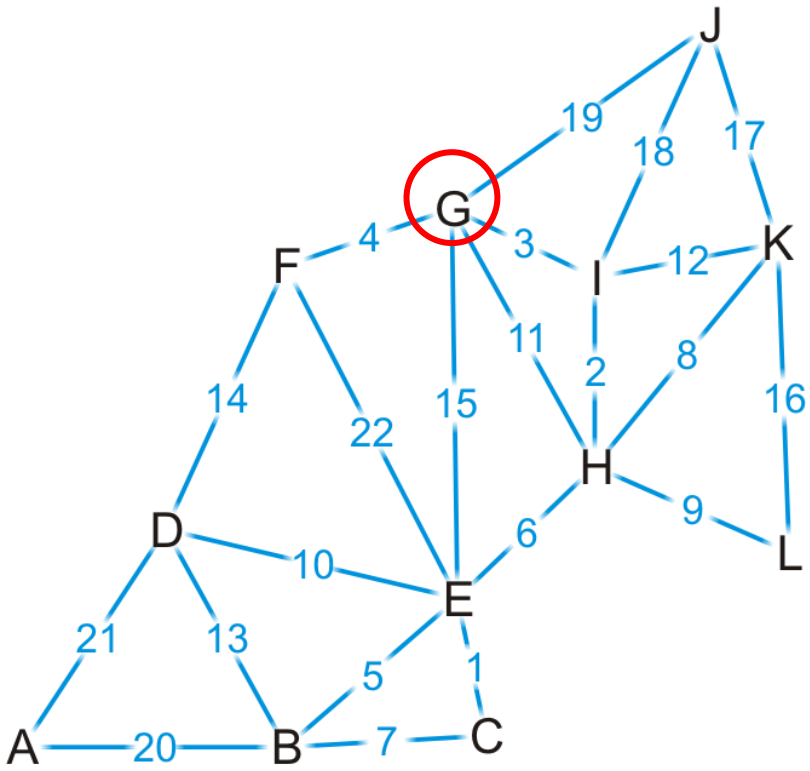


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

The path (K, H, I, G) is the shortest path from K to G of length 13

- Vertex G has three unvisited neighbors: E, F and J



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	**T**	**13**	**I**
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

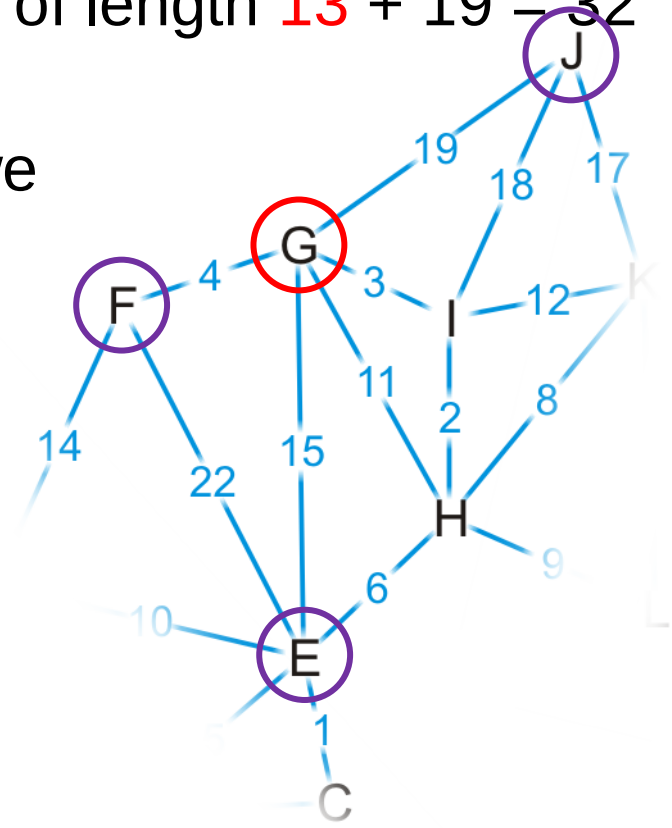
Example

Consider these paths:

(K, H, I, G, E) of length 13 + 15 = 28 (K, H, I, G, F) of length 13 + 4 = 17

(K, H, I, G, J) of length 13 + 19 = 32

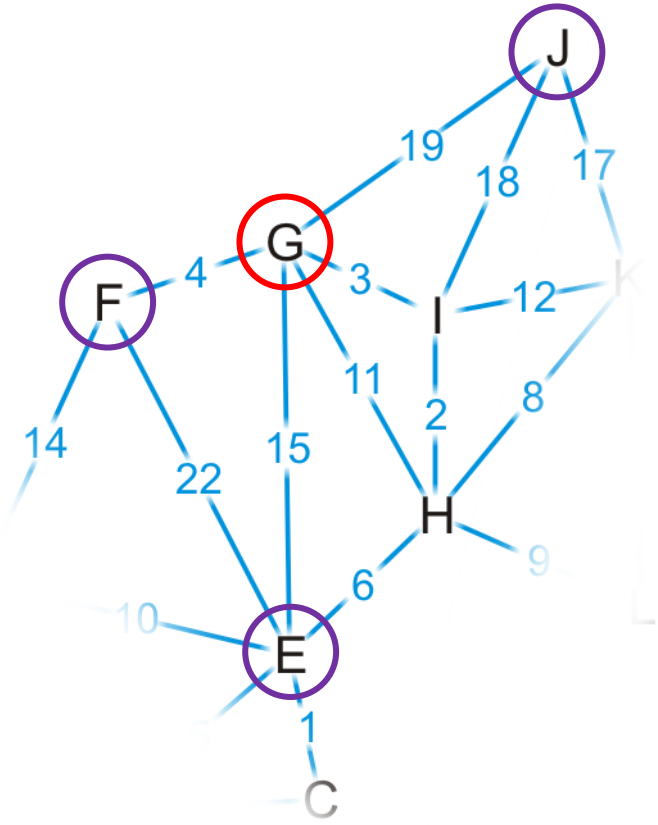
- Which do we update?



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	**T**	**13**	**I**
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

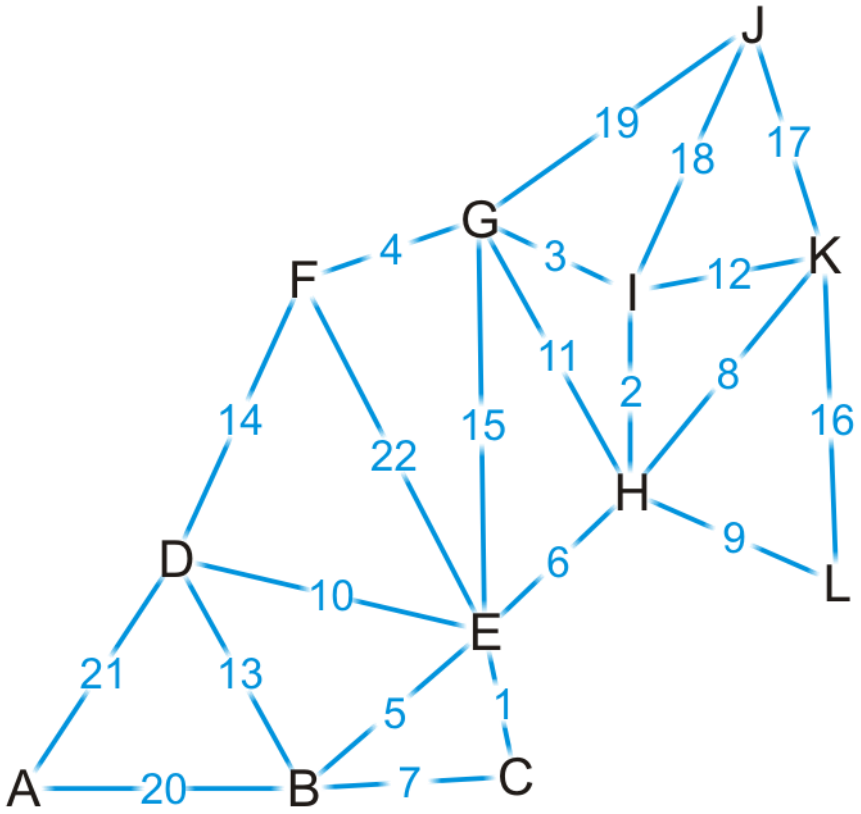
We have now found a path to vertex F



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	**17**	**G**
G	**T**	**13**	**I**
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Where do we visit next?

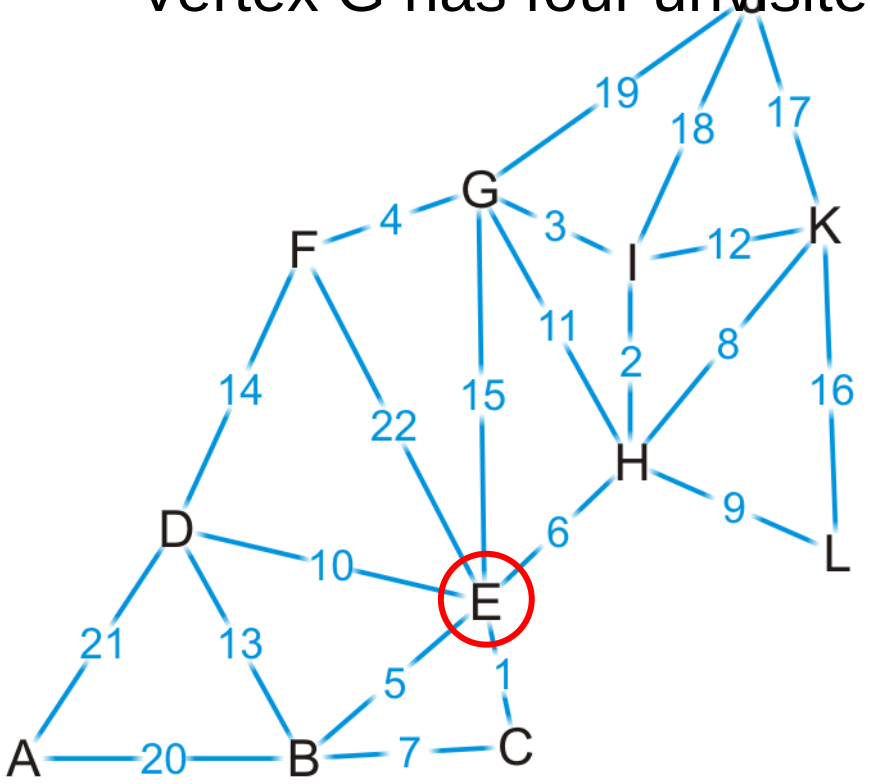


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

The path (K, H, E) is the shortest path from K to E of length 14

- Vertex G has four unvisited neighbors: B, C, D and F

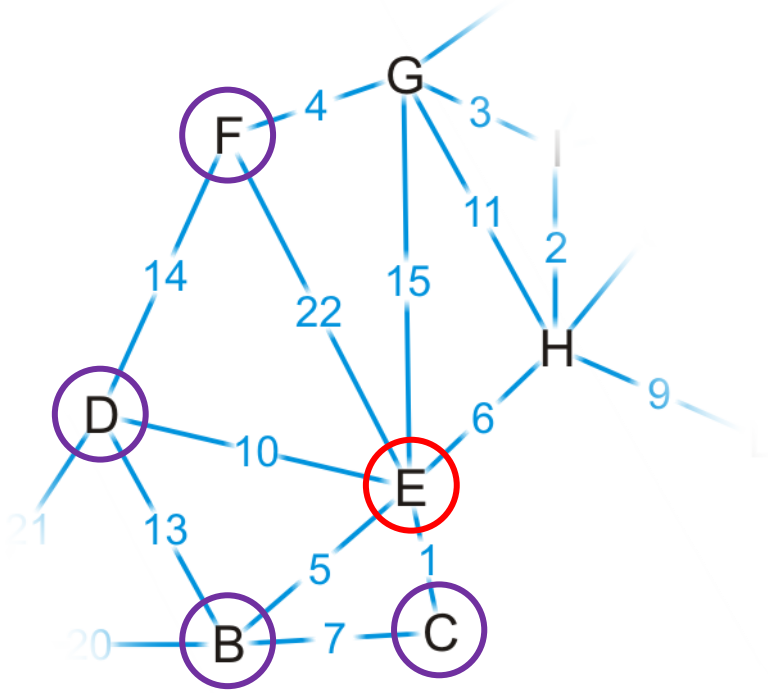


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

The path (K, H, E) is the shortest path from K to E of length 14

- Vertex G has four unvisited neighbors: B, C, D and F



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Consider these paths:

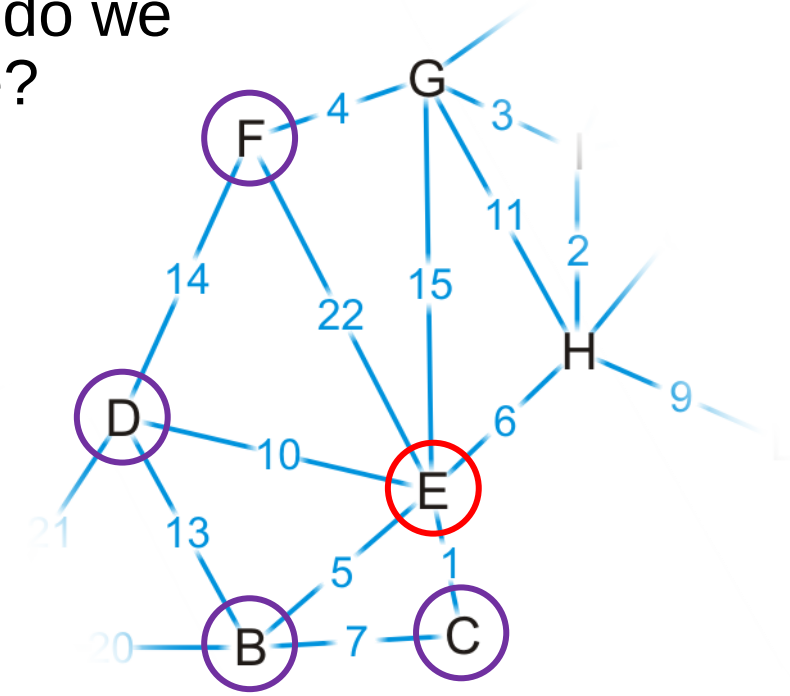
(K, H, E, B) of length 14 + 5 = 19

(K, H, E, D) of length 14 + 10 = 24

(K, H, E, C) of length 14 + 1 = 15

(K, H, E, F) of length 14 + 22 = 36

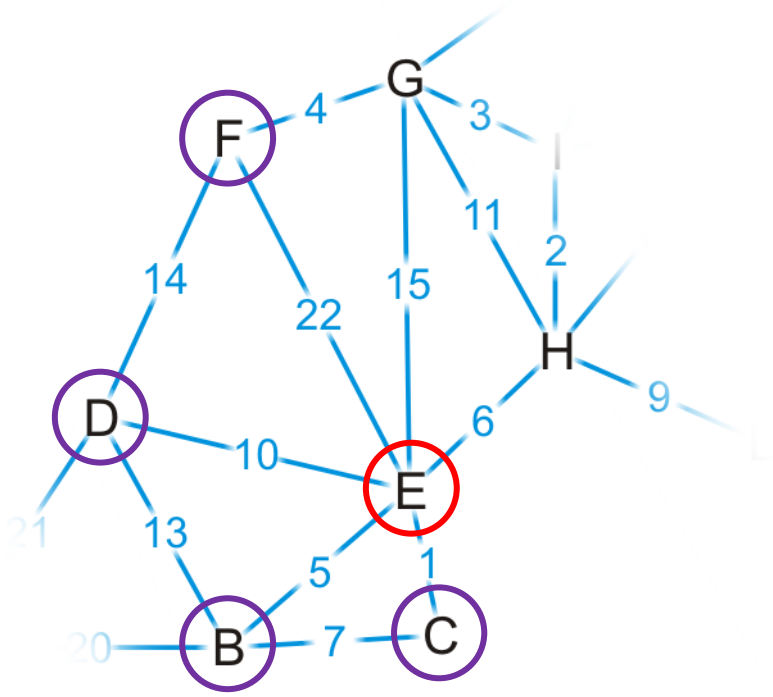
- Which do we update?



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

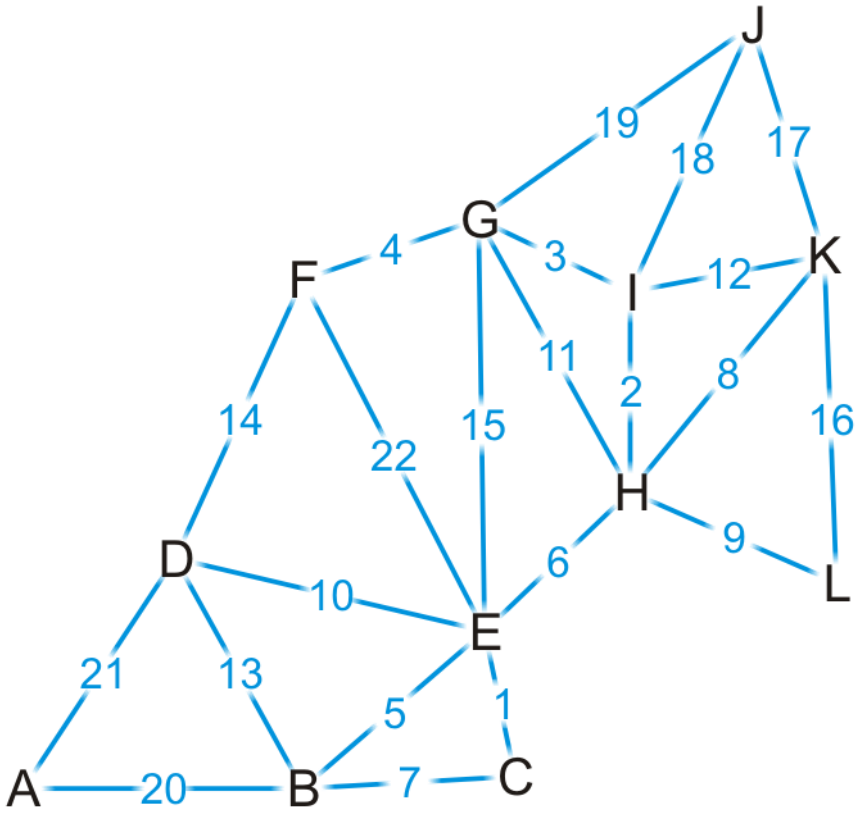
We've discovered paths to vertices B, C, D



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	**19**	**E**
C	F	**15**	**E**
D	F	**24**	**E**
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Which vertex is next?

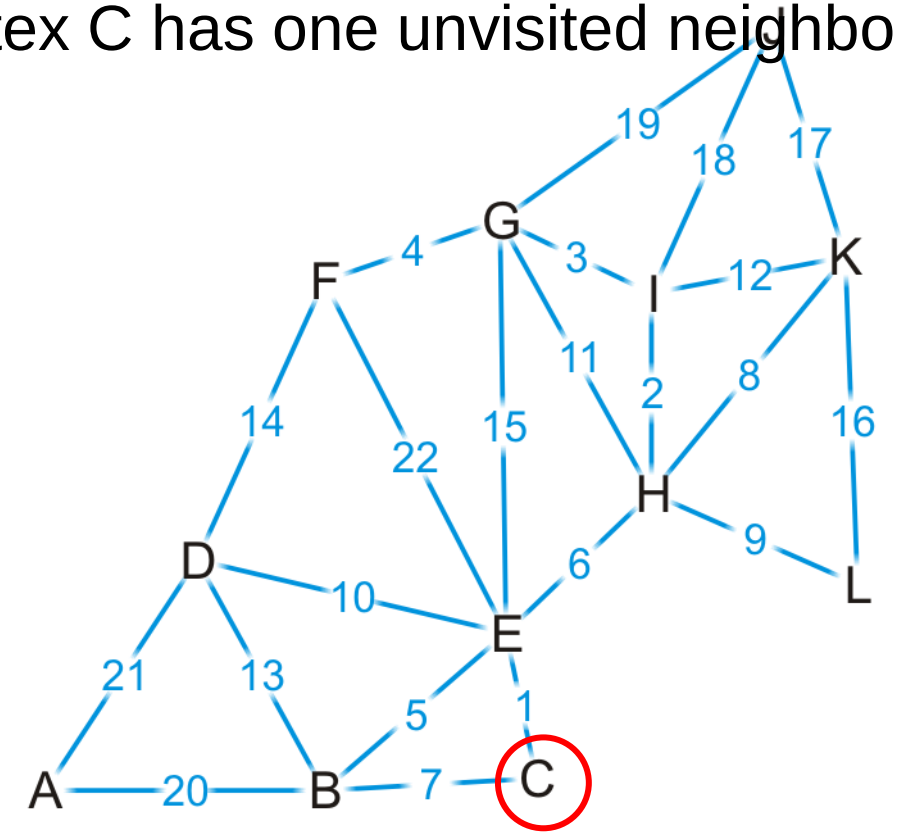


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	F	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We've found that the path (K, H, E, C) of length 15 is the shortest path from K to C

- Vertex C has one unvisited neighbor, B

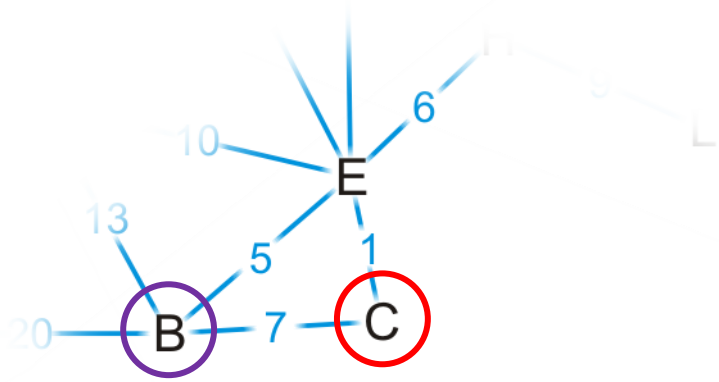


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	**T**	**15**	**E**
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

The path (K, H, E, C, B) is of length 15 + 7 = 22

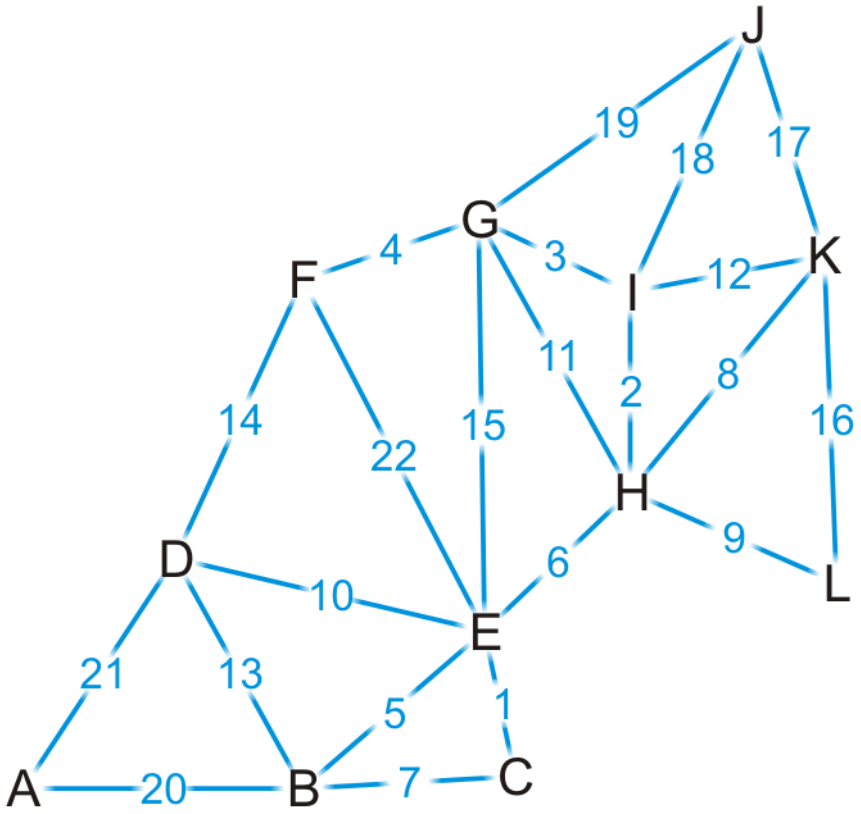
- We have already discovered a shorter path through vertex E



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	**T**	**15**	**E**
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Where to next?

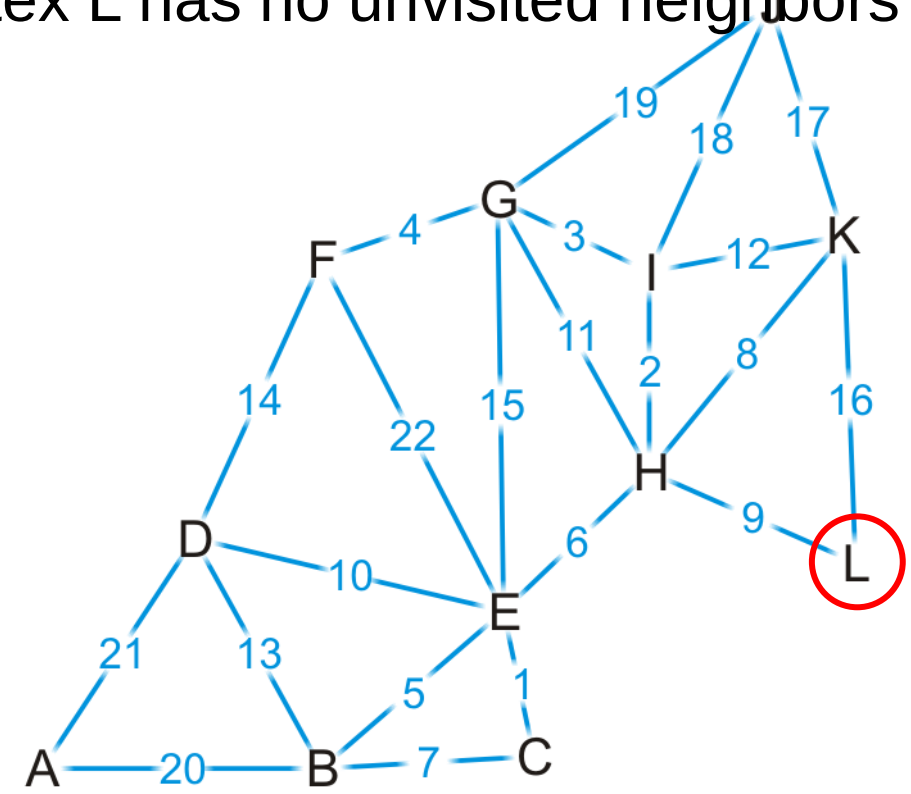


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We now know that (K, L) is the shortest path between these two points

- Vertex L has no unvisited neighbors

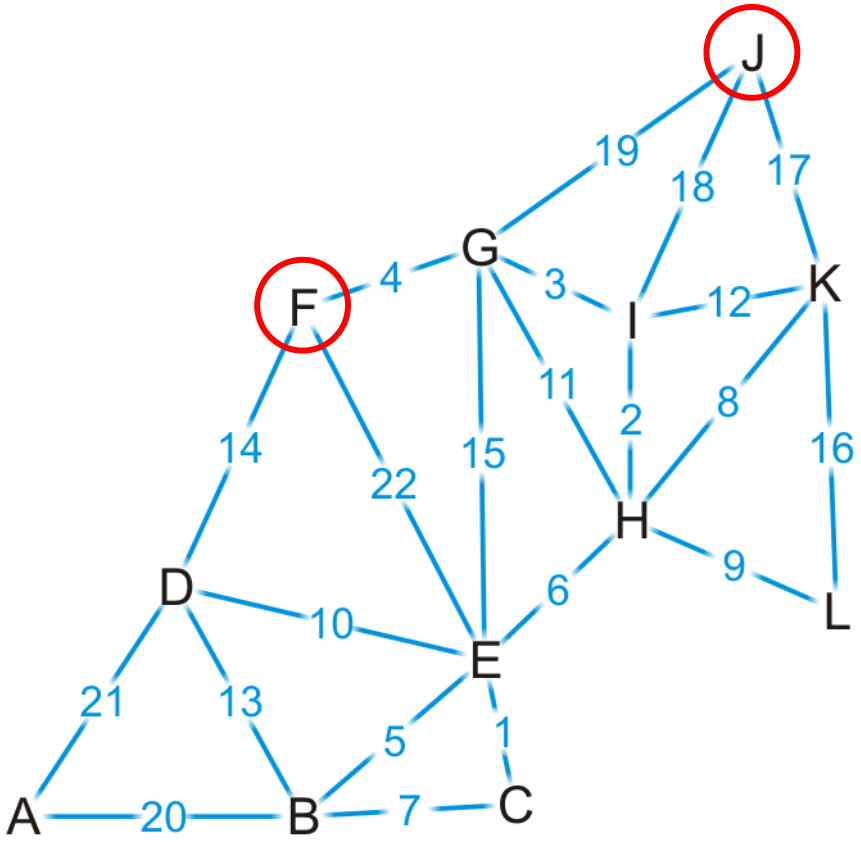


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	**T**	**16**	**K**

Example

Where to next?

- Does it matter if we visit vertex F first or vertex J first?

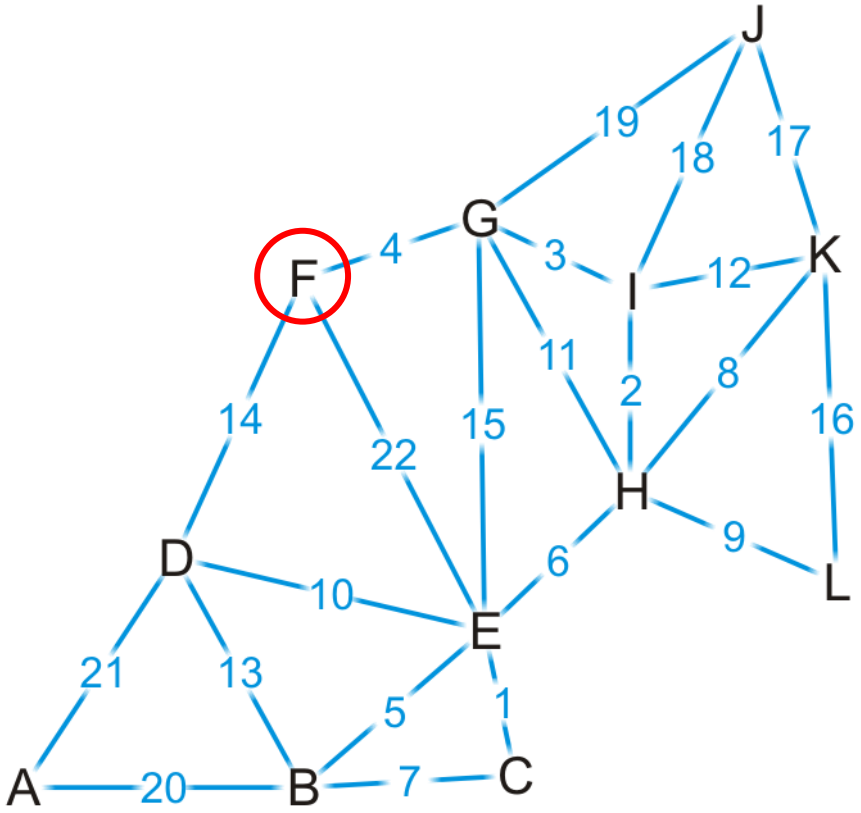


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

Example

Let’s visit vertex F first

- It has one unvisited neighbor, vertex D

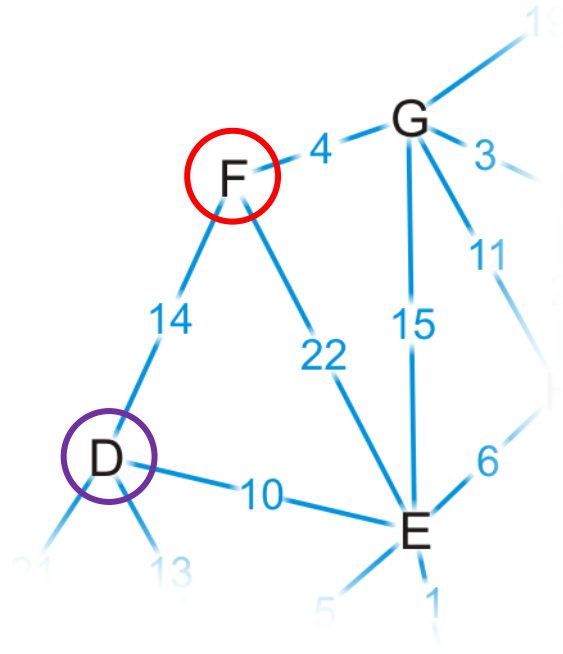


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	**T**	**17**	**G**
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

Example

The path (K, H, I, G, F, D) is of length 17 + 14 = 31

- This is longer than the path we've already discovered

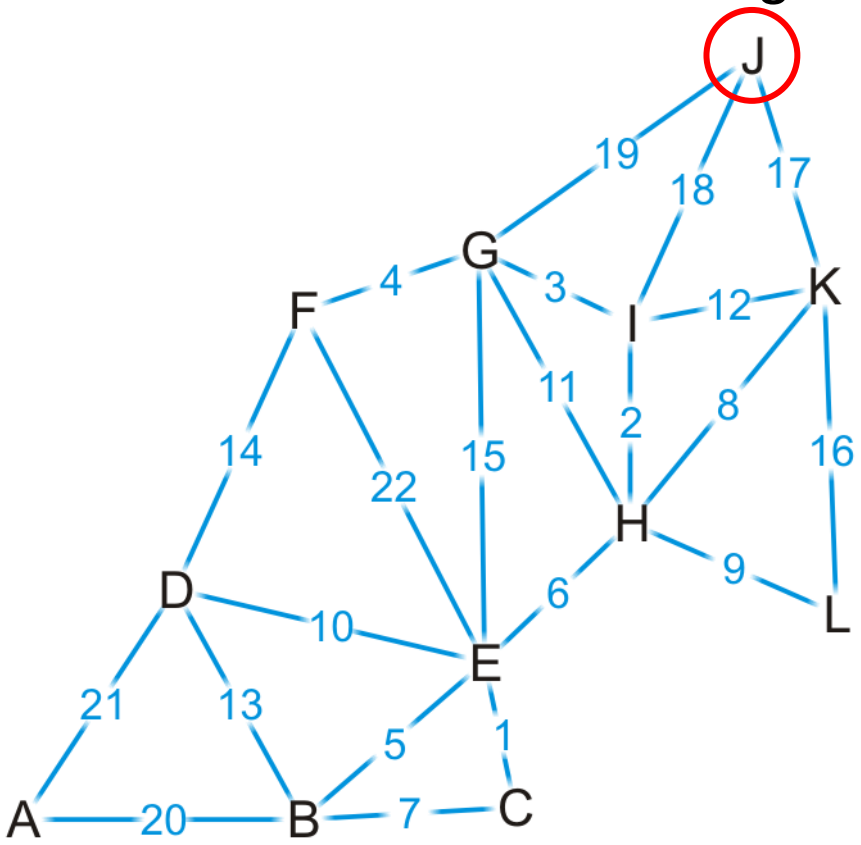


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	**T**	**17**	**G**
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

Example

Now we visit vertex J

- It has no unvisited neighbors



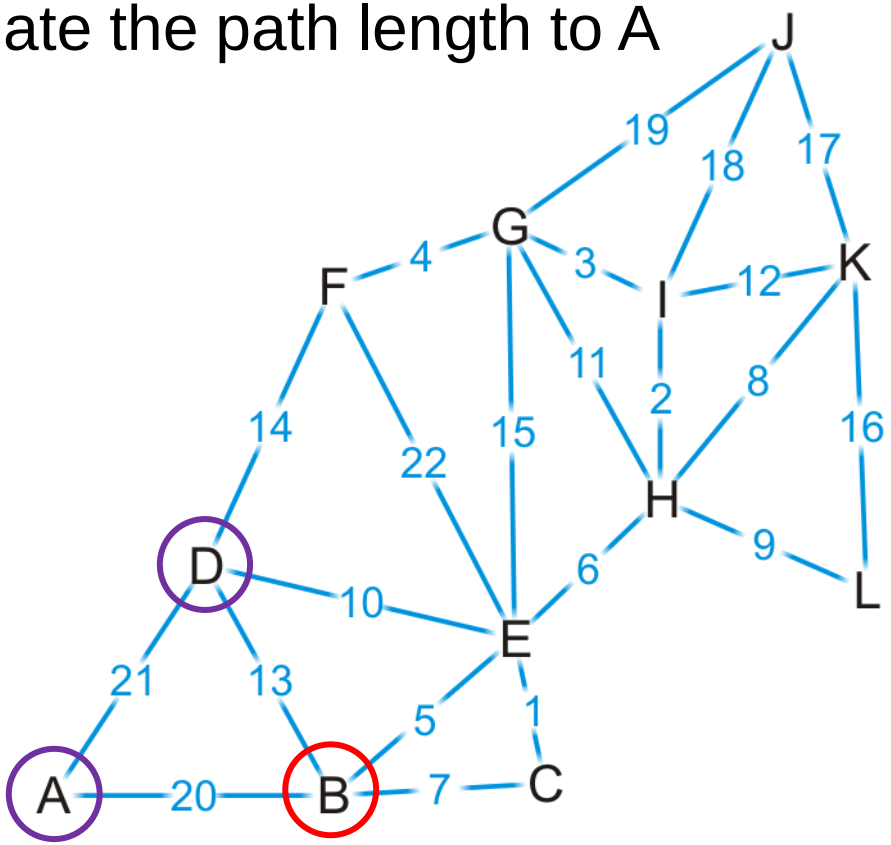
Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	**T**	**17**	**K**
K	T	0	Ø
L	T	16	K

Example

Next we visit vertex B, which has two unvisited neighbors:

(K, H, E, B, A) of length 19 + 20 = 39 (K, H, E, B, D) of length 19 + 13 = 32

- We update the path length to A

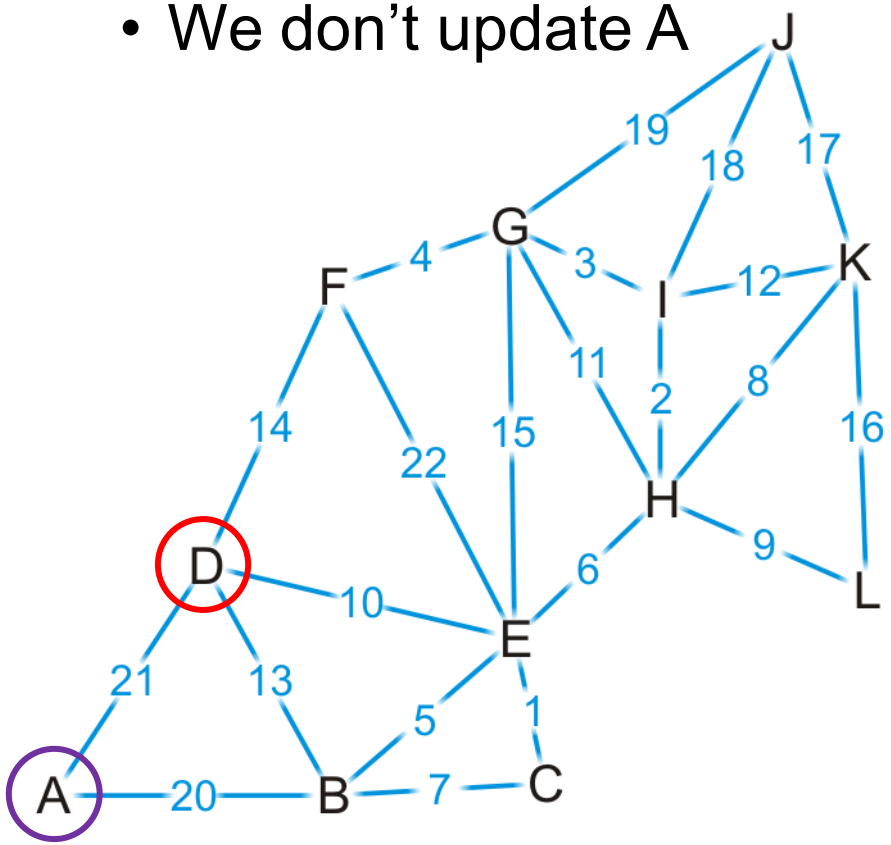


Vertex	Visited	Distance	Previous
A	F	**39**	**B**
B	**T**	**19**	**E**
C	T	15	E
D	F	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Next we visit vertex D

- The path (K, H, E, D, A) is of length 24 + 21 = 45
- We don't update A

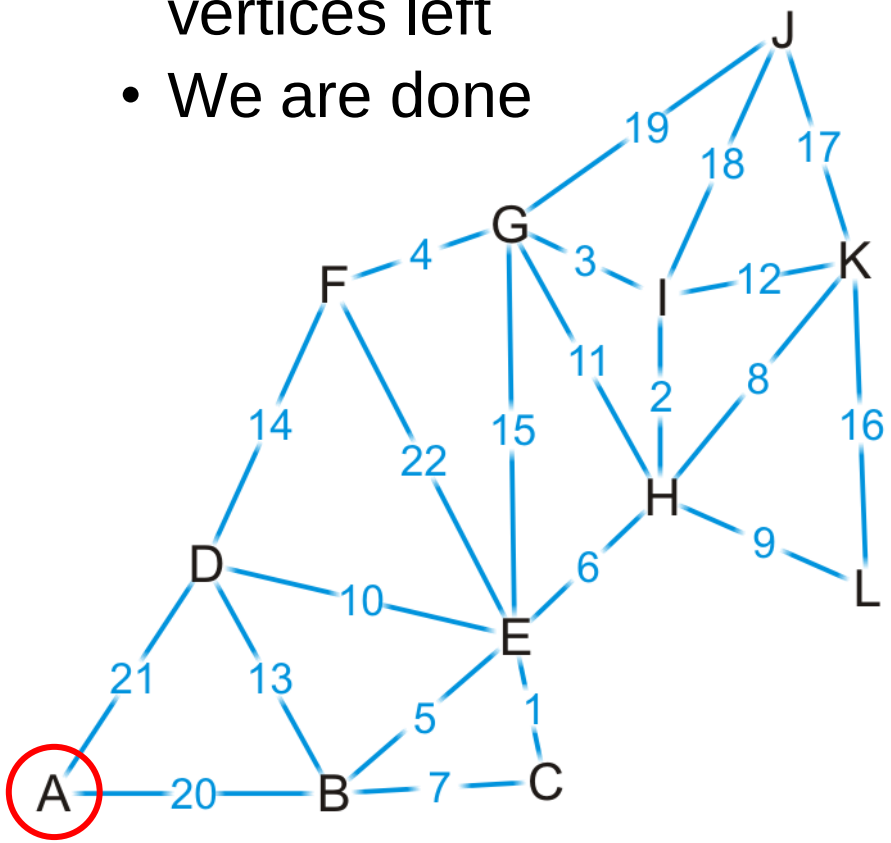


Vertex	Visited	Distance	Previous
A	F	39	B
B	T	19	E
C	T	15	E
D	**T**	**24**	**E**
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Finally, we visit vertex A

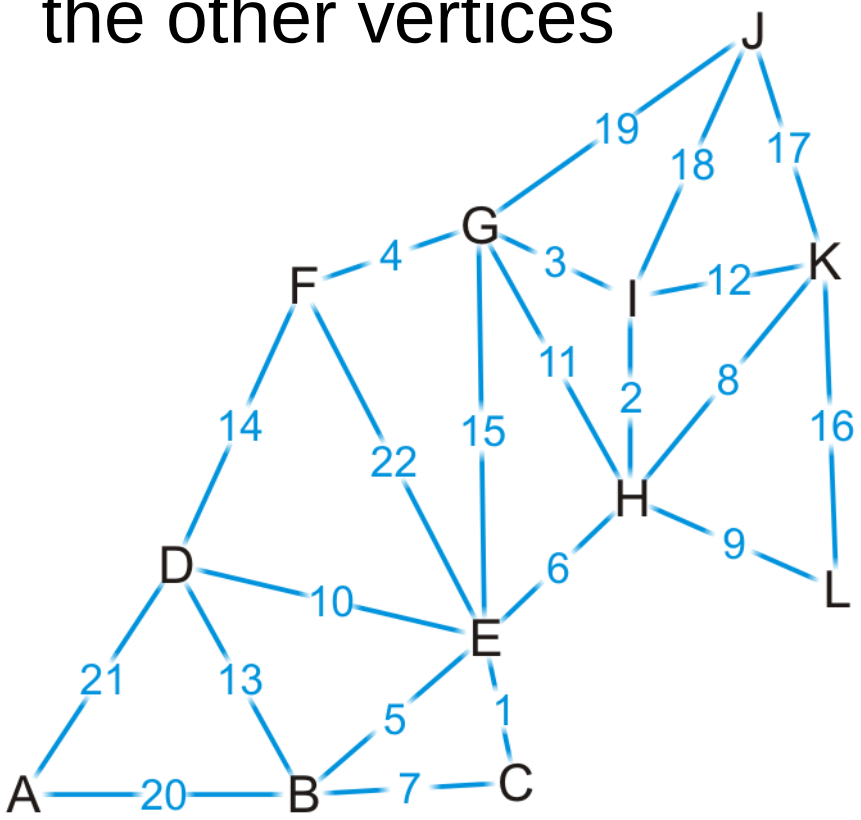
- It has no unvisited neighbors and there are no unvisited vertices left
- We are done



Vertex	Visited	Distance	Previous
A	**T**	**39**	**B**
B	T	19	E
C	T	15	E
D	T	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

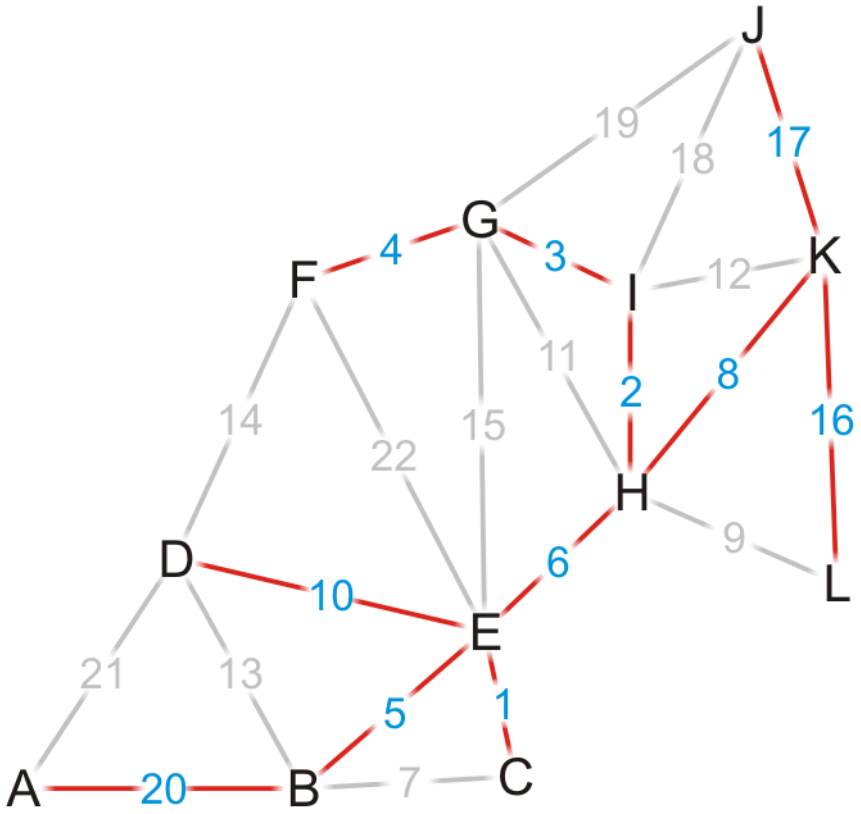
Thus, we have found the shortest path from vertex K to each of
the other vertices



Vertex	Visited	Distance	Previous
A	T	39	B
B	T	19	E
C	T	15	E
D	T	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Using the *previous* pointers, we can reconstruct the paths

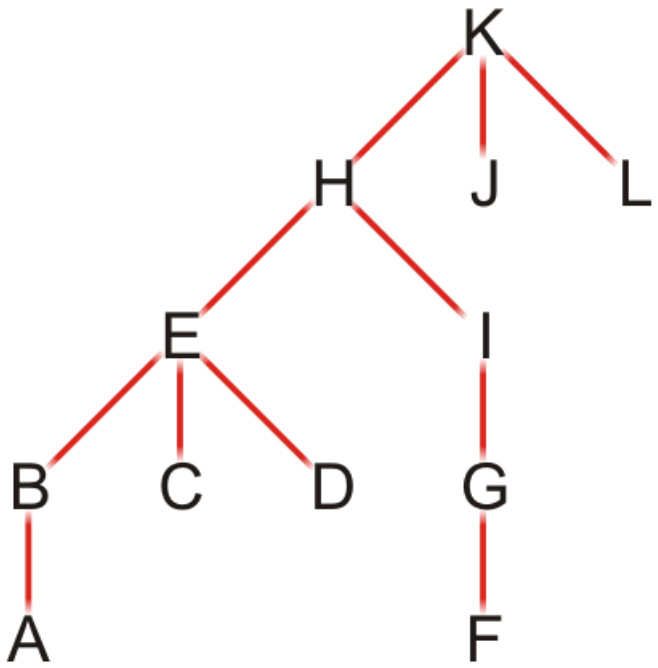


Vertex	Visited	Distance	Previous
A	T	39	B
B	T	19	E
C	T	15	E
D	T	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Note that this table defines a rooted parental tree (is it also a minimum spanning tree?)

- The source vertex K is at the root
- The previous pointer is the *parent* of the vertex in the tree



Vertex	Previous
A	B
B	E
C	E
D	E
E	H
F	G
G	I
H	K
I	H
J	K
K	Ø
L	K

Comments on Dijkstra's algorithm

Questions:

- What if at some point, all unvisited vertices have a distance ∞?
  - This means that the graph is unconnected
  - We have found the shortest paths to all vertices in the connected subgraph containing the source vertex
- What if we just want to find the shortest path between vertices v_j and v_k?
  - Apply the same algorithm, but stop when we are visiting vertex v_k
- Does the algorithm change if we have a directed graph?
  - No

Implementation and analysis

The initialization requires $\Theta(|V|)$ memory and run time

We iterate $|V|$ times, each time finding next closest vertex to the source

- Iterating through the table requires $\Theta(|V|)$ time
- Each time we find a vertex, we must check all of its neighbors
  - With an adjacency matrix, the run time is $\Theta(|V|(|V| + |V|)) = \Theta(|V|^2)$
  - With an adjacency list, the run time is $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$ as $|E| = O(|V|^2)$

Can we do better?

- How about using a priority queue to find the closest vertex?
  - Assume we are using a binary heap

Implementation and analysis

The initialization still requires $\Theta(|V|)$ memory and run time
- The priority queue will also requires $O(|V|)$ memory
- We must use an adjacency list, not an adjacency matrix

We iterate $|V|$ times, each time finding the *closest* vertex to the source
- Place the distances into a priority queue
- The size of the priority queue is $O(|V|)$
- Thus, the work required for this is $O(|V| \ln(|V|))$

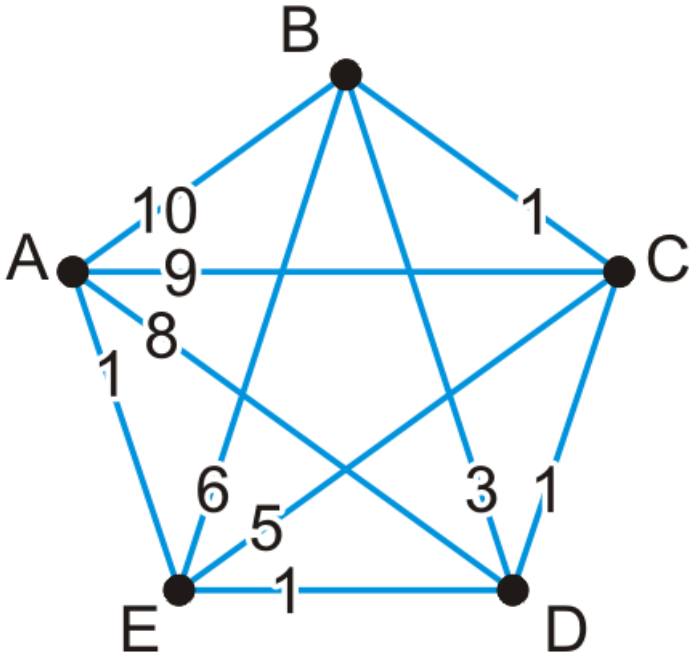
Is this all the work that is necessary?
- Recall that each edge visited may result in a distance being updated
- Thus, the work required for this is $O(|E| \ln(|V|))$

Thus, the total run time is $O(|V| \ln(|V|) + |E| \ln(|V|)) = O(|E| \ln(|V|))$?

Implementation and analysis

Here is an example of a worst-case scenario:

- Immediately, all of the vertices are placed into the queue
- Each time a vertex is visited, all the remaining vertices are checked, and in succession, each is pushed to the top of the binary heap



Implementation and analysis

We could use a different heap structure:

- A Fibonacci heap is a node-based heap
- Pop is still $O(\ln(|V|))$, but inserting and moving a key is $\Theta(1)$
- Thus, because we are only calling pop $|V| - 1$ times, the overall run-time reduces to $O(|E| + |V| \ln(|V|))$

Implementation and analysis

Thus, we have two run times when using

- A binary heap: $O(|E| \ln(|V|))$
- A Fibonacci heap: $O(|E| + |V| \ln(|V|))$

Questions: Which is faster if $|E| = \Theta(|V|)$? How about if $|E| = \Theta(|V|^2)$?

Summary

We have seen an algorithm for finding single-source shortest paths

- Start with the initial vertex
- Continue finding the next vertex that is closest

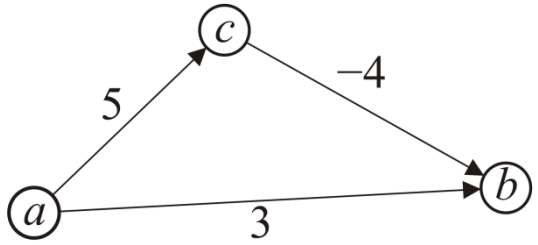
Dijkstra's algorithm always finds the next closest vertex

- It solves the problem in $O(|E| + |V| \ln(|V|))$ time

Shortest Path – Bellman Ford

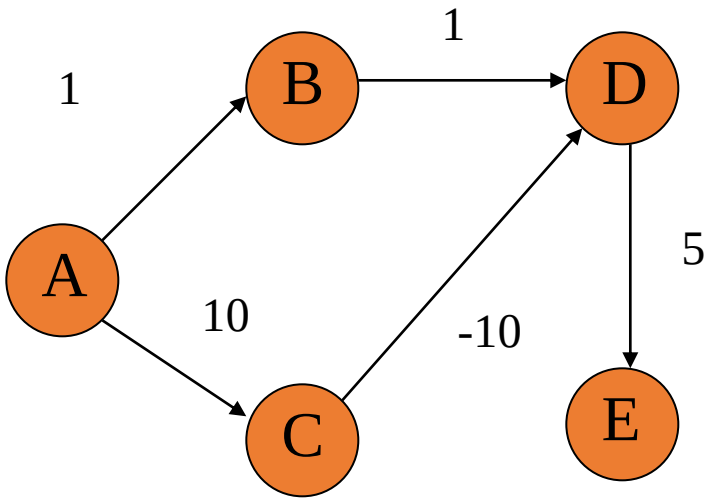
Negative Weights

If some of the edges have negative weight, so long as there are no cycles with negative weight, the Bellman-Ford algorithm will find the minimum distance

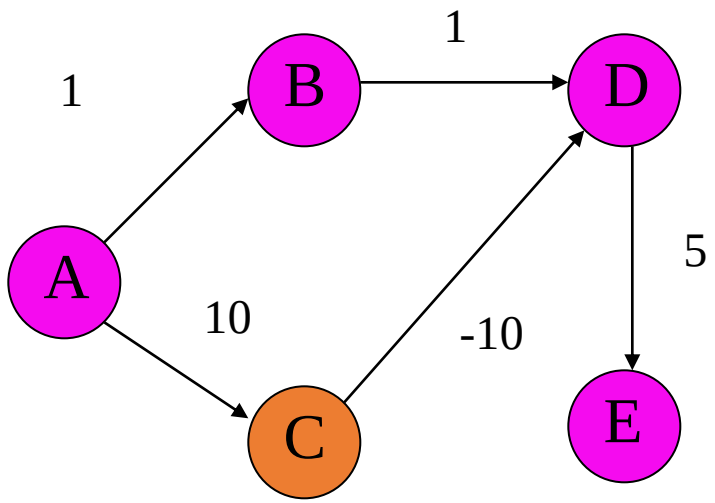


- It is slower than Dijkstra's algorithm

What about Dijkstra's on…?

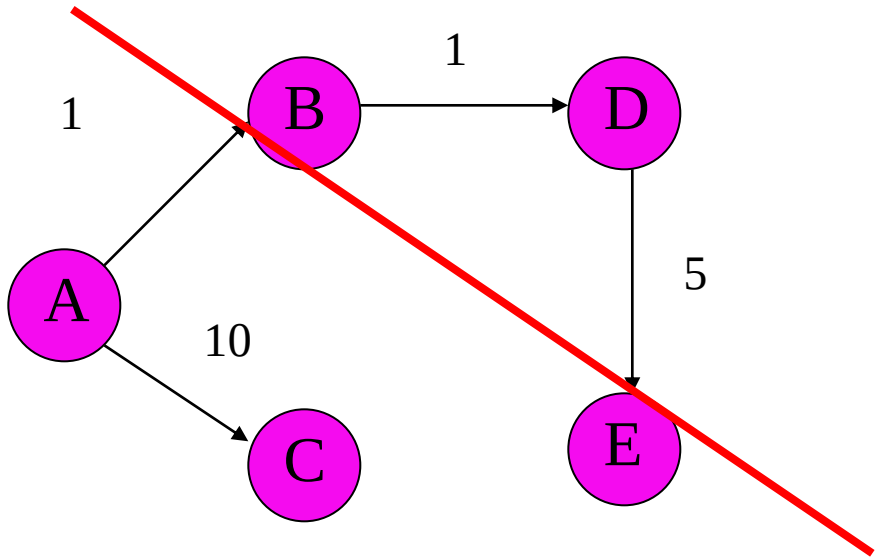


What about Dijkstra's on…?



What about Dijkstra's on…?

Dijkstra's algorithm only works for positive edge weights



Bounding the distance

- Another invariant: For each vertex v, dist[v] is an upper bound on the actual shortest distance
  - start of at ∞
  - only update the value if we find a shorter distance
- An update procedure

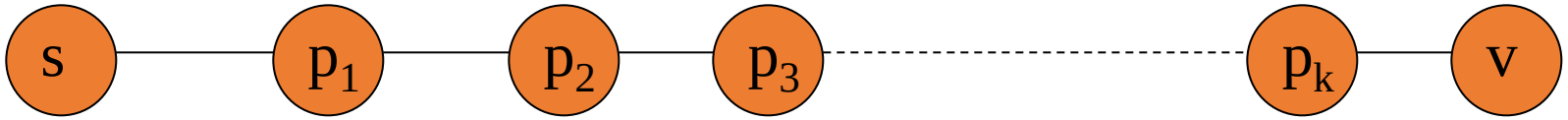
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- Can we ever go wrong applying this update rule?
  - We can apply this rule as many times as we want and will never underestimate dist[v]
- When will dist[v] be right?
  - If u is along the shortest path to v and dist[u] is correct

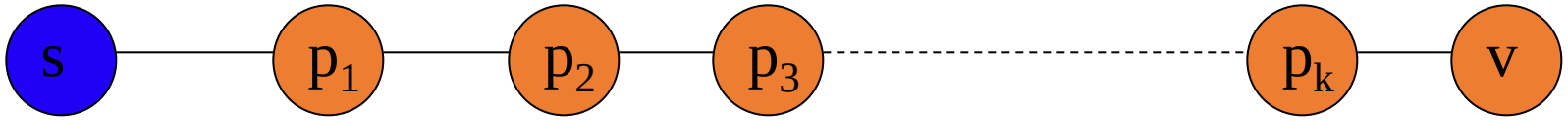
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- Consider the shortest path from s to v
- What happens if we update all of the vertices with the above update?



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

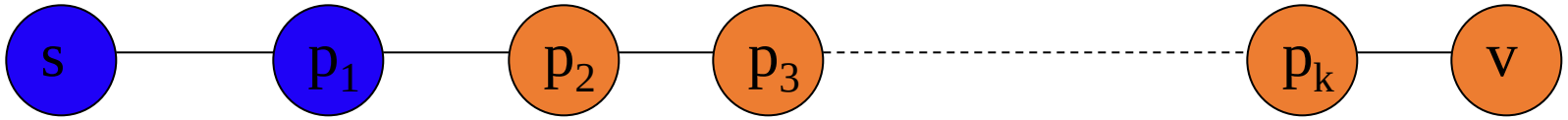
- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What happens if we update all of the vertices with the above update?



correct

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

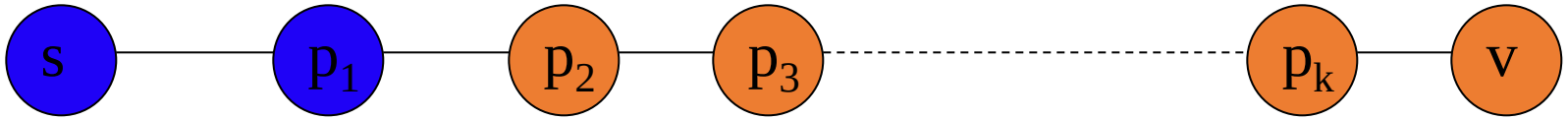
- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What happens if we update all of the vertices with the above update?



correct correct

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

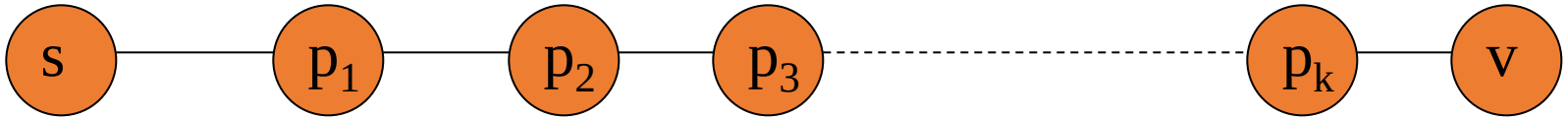
- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- Does the order that we update the vertices matter?



correct correct

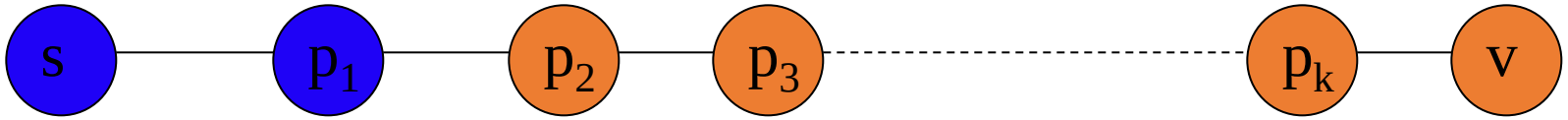
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

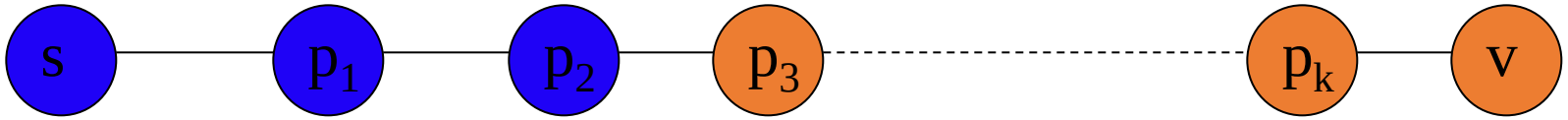
- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



correct correct

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

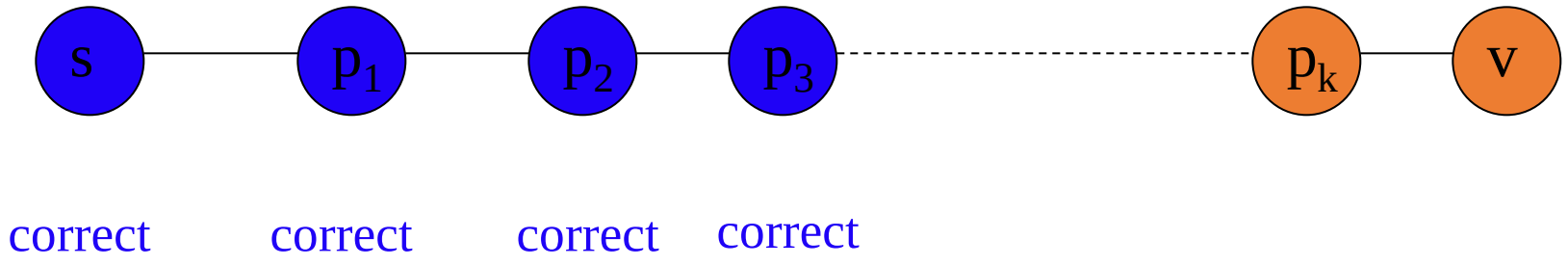
- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



correct correct correct

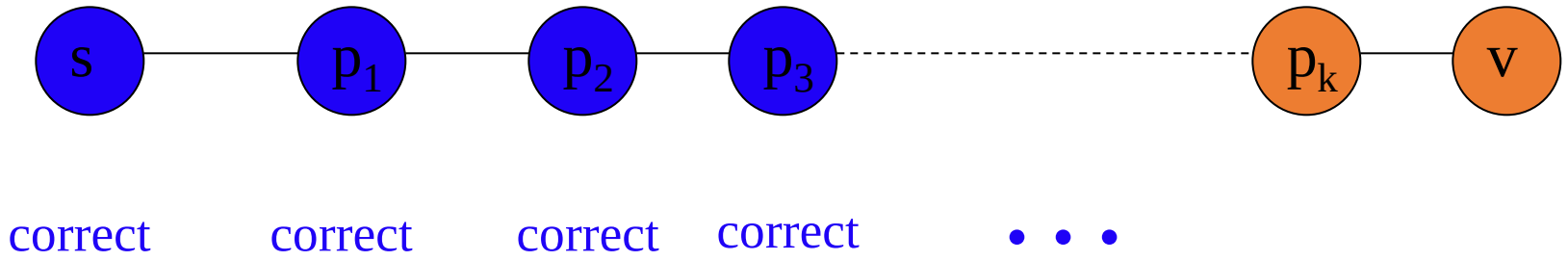
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



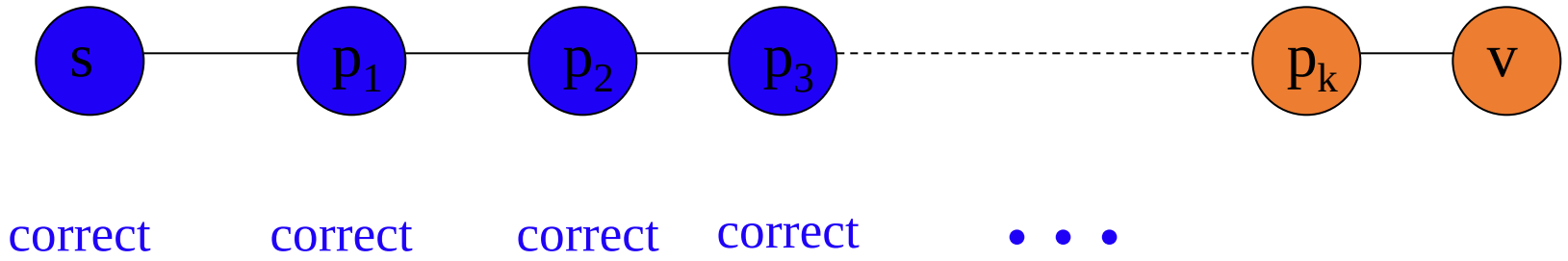
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What is the longest (vetex-wise) the path from s to any node v can be?
  - |V| - 1 edges/vertices



Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
 1  for all v ∈ V
 2          dist[v] ← ∞
 3          prev[v] ← null
 4  dist[s] ← 0
 5  for i ← 1 to |V| − 1
 6          for all edges (u, v) ∈ E
 7                  if dist[v] > dist[u] + w(u, v)
 8                          dist[v] ← dist[u] + w(u, v)
 9                          prev[v] ← u
10  for all edges (u, v) ∈ E
11          if dist[v] > dist[u] + w(u, v)
12                  return false
```

O(|V| |E|)

Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
1   for all v ∈ V
2           dist[v] ← ∞
3           prev[v] ← null
4   dist[s] ← 0
5   for i ← 1 to |V| − 1
6           for all edges (u, v) ∈ E
7                   if dist[v] > dist[u] + w(u, v)
8                           dist[v] ← dist[u] + w(u, v)
9                           prev[v] ← u
10  for all edges (u, v) ∈ E
11          if dist[v] > dist[u] + w(u, v)
12                  return false
```

Initialize all the distances

Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
 1  for all v ∈ V
 2          dist[v] ← ∞
 3          prev[v] ← null
 4  dist[s] ← 0
 5  for i ← 1 to |V| − 1
 6          for all edges (u, v) ∈ E
 7                  if dist[v] > dist[u] + w(u, v)
 8                          dist[v] ← dist[u] + w(u, v)
 9                          prev[v] ← u
10  for all edges (u, v) ∈ E
11          if dist[v] > dist[u] + w(u, v)
12                  return false
```

iterate over all edges/vertices and apply update rule

Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
 1  for all v ∈ V
 2          dist[v] ← ∞
 3          prev[v] ← null
 4  dist[s] ← 0
 5  for i ← 1 to |V| − 1
 6          for all edges (u, v) ∈ E
 7                  if dist[v] > dist[u] + w(u, v)
 8                          dist[v] ← dist[u] + w(u, v)
 9                          prev[v] ← u
10  for all edges (u, v) ∈ E
11          if dist[v] > dist[u] + w(u, v)
12                  return false
```

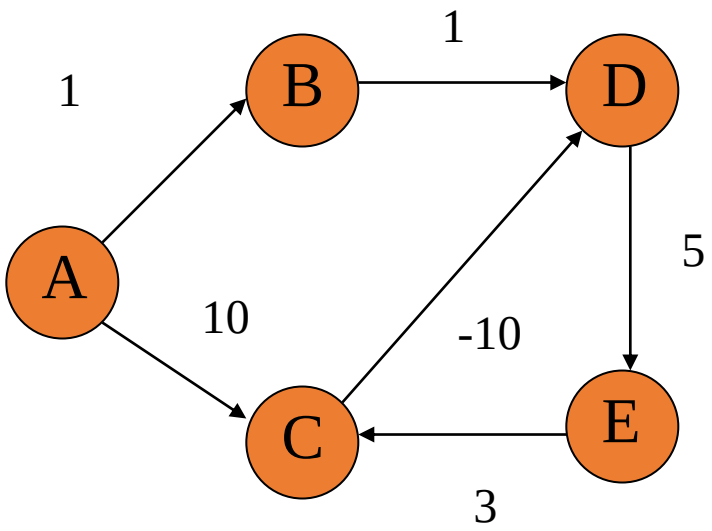
Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
 1  for all v ∈ V
 2          dist[v] ← ∞
 3          prev[v] ← null
 4  dist[s] ← 0
 5  for i ← 1 to |V| − 1
 6          for all edges (u, v) ∈ E
 7                  if dist[v] > dist[u] + w(u, v)
 8                          dist[v] ← dist[u] + w(u, v)
 9                          prev[v] ← u
10  for all edges (u, v) ∈ E
11          if dist[v] > dist[u] + w(u, v)
12                  return false
```

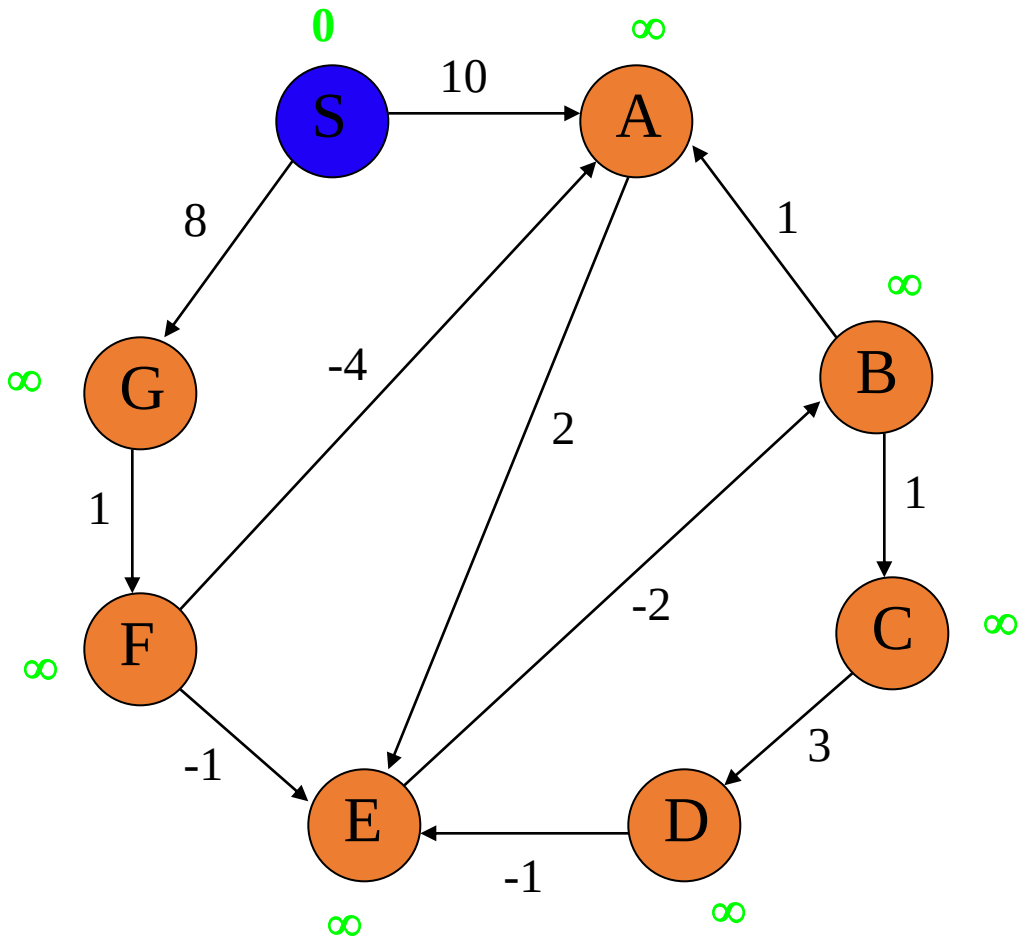
check for negative cycles

Negative cycles

What is the shortest path from a to e?

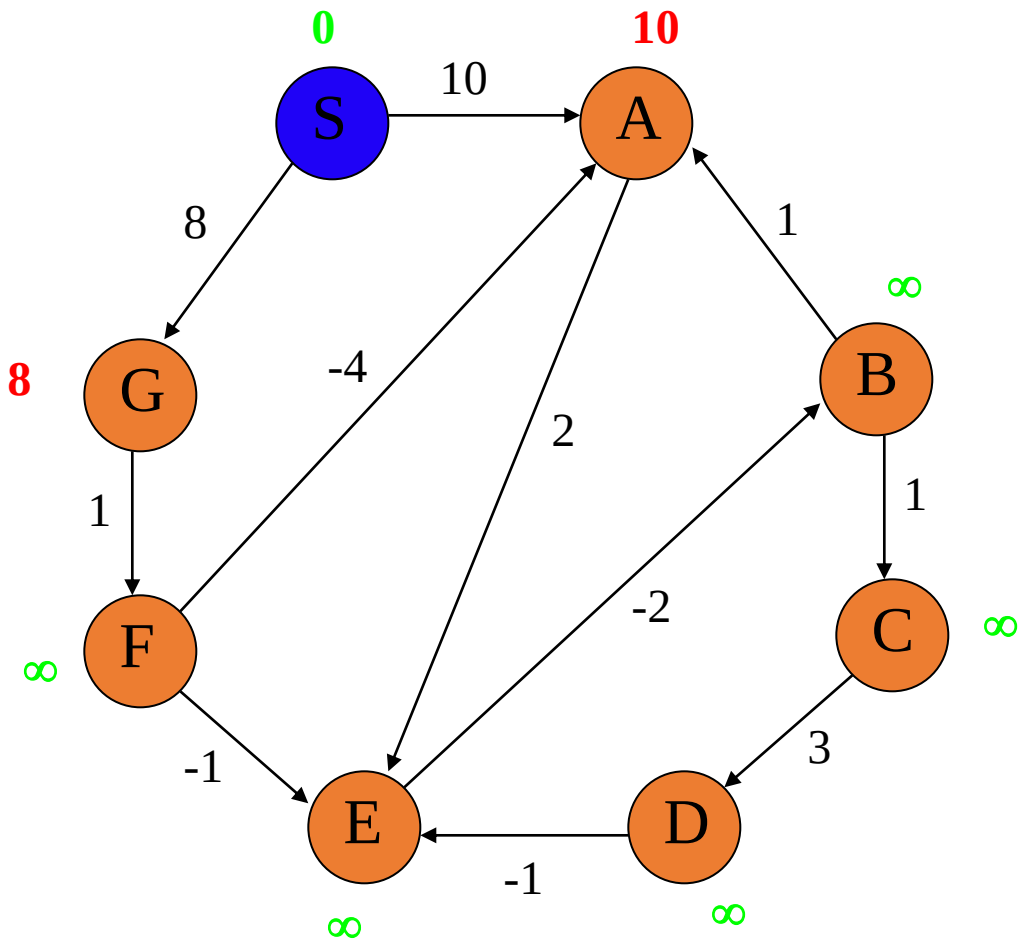


Bellman-Ford algorithm



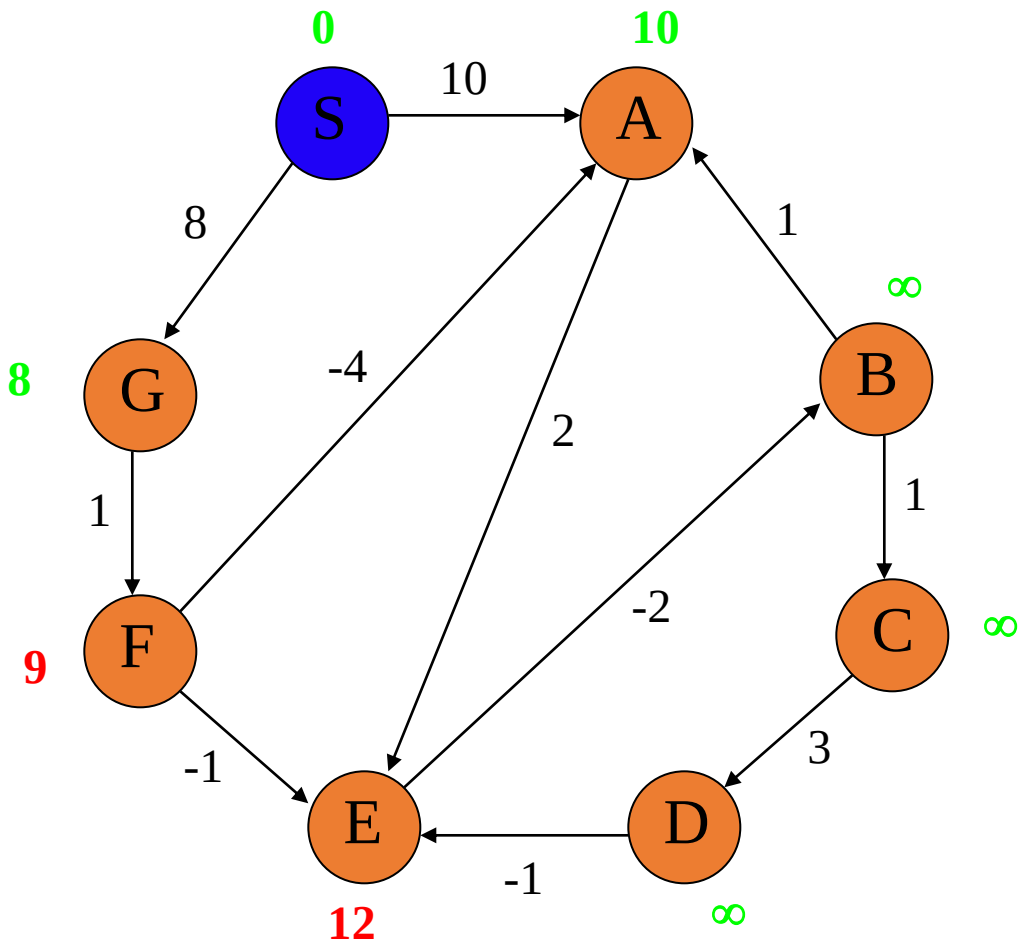
Iteration: 0

Bellman-Ford algorithm



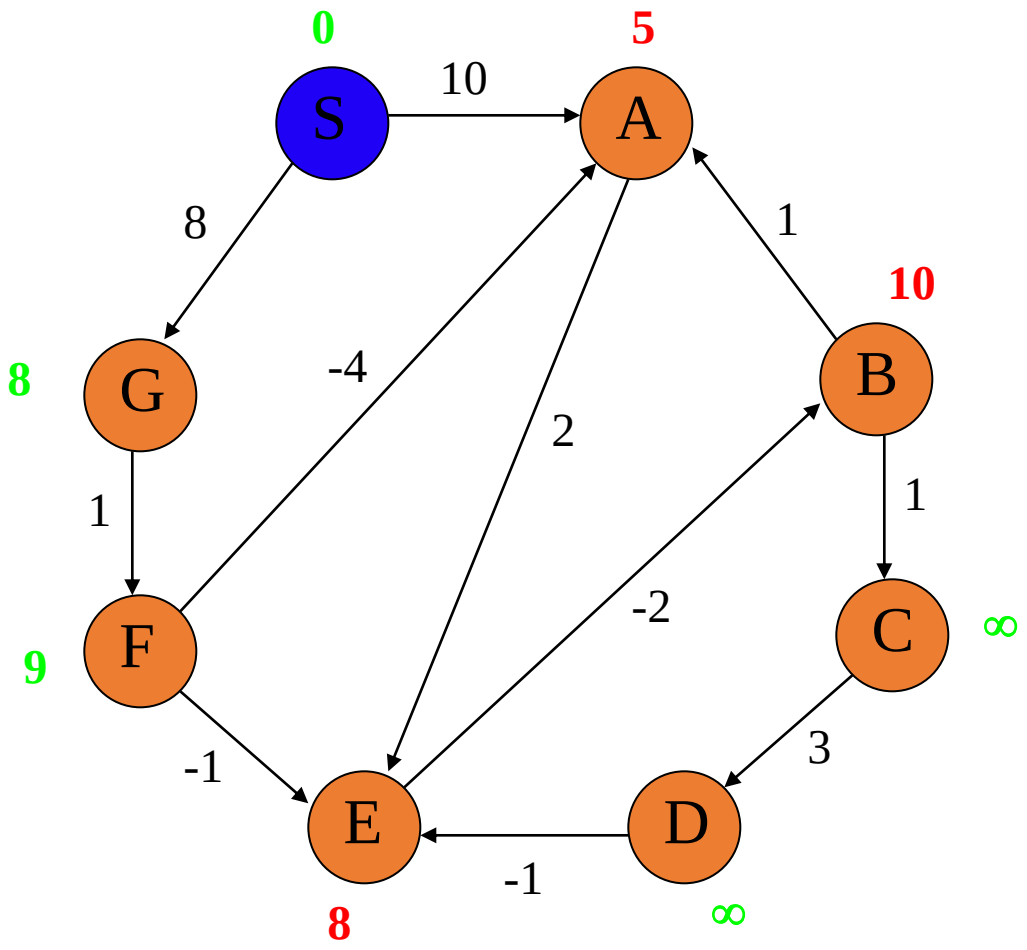
Iteration: 1

Bellman-Ford algorithm



Iteration: 2

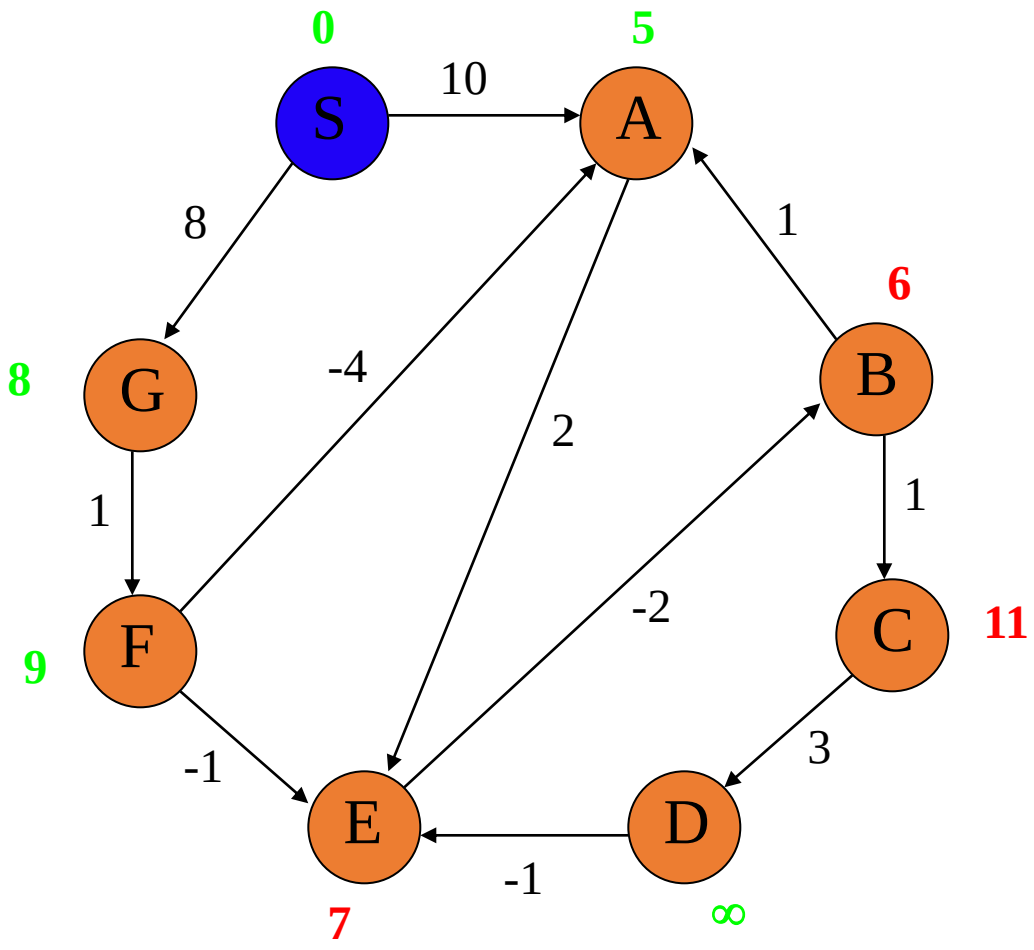
Bellman-Ford algorithm



Iteration: 3

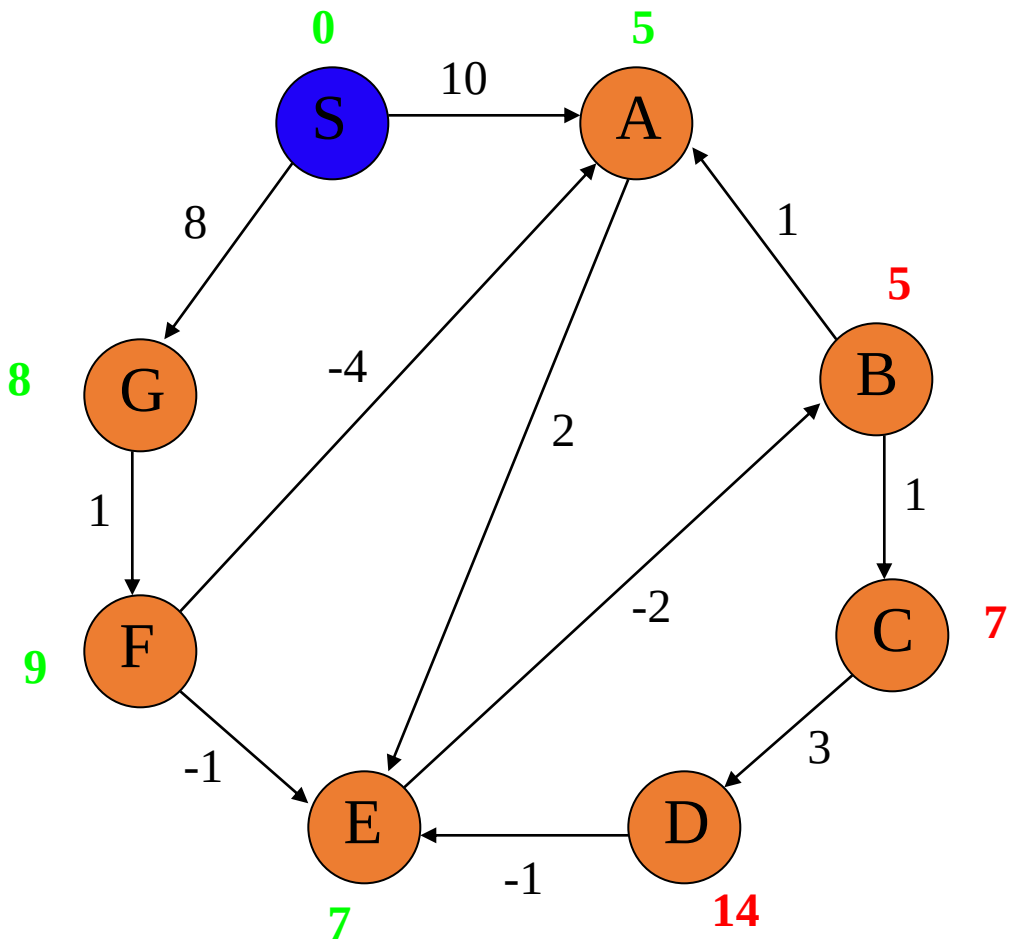
A has the correct distance and path

Bellman-Ford algorithm



Iteration: 4

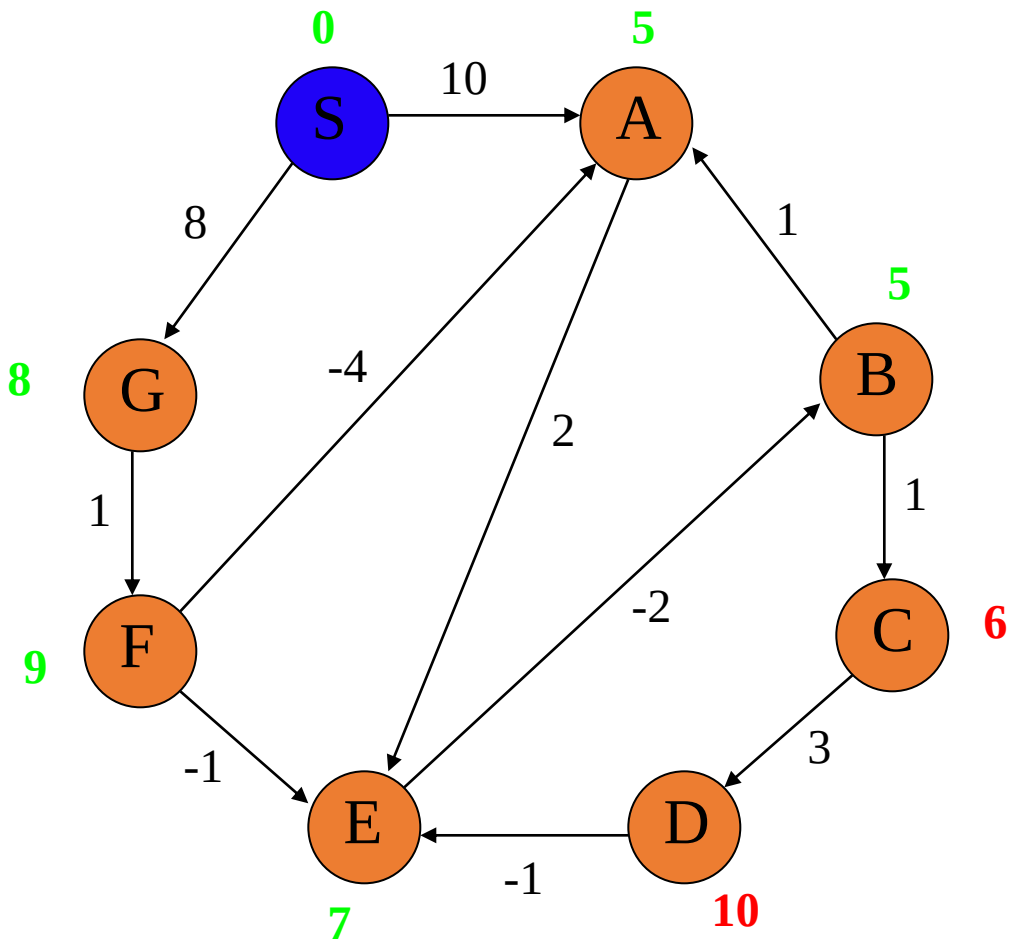
Bellman-Ford algorithm



Iteration: 5

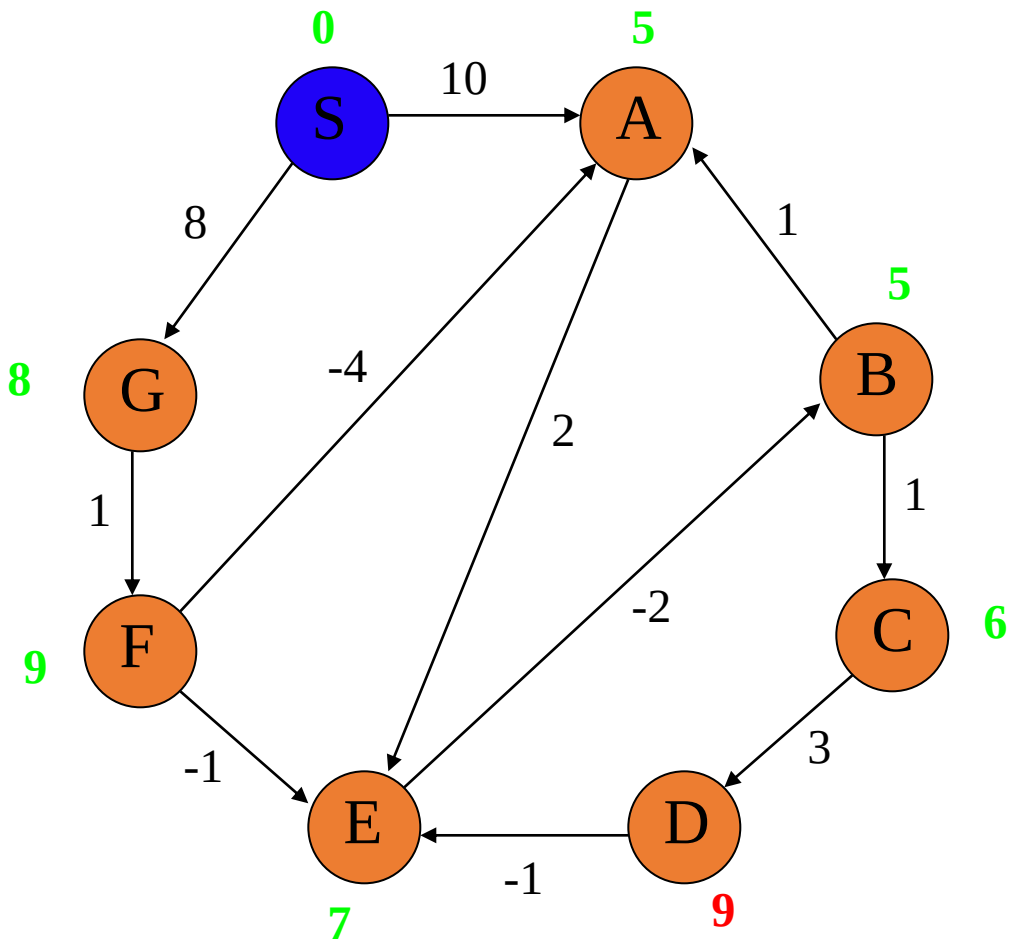
B has the correct distance and path

Bellman-Ford algorithm



Iteration: 6

Bellman-Ford algorithm



Iteration: 7

D (and all other nodes) have the correct distance and path

Summary

- Dijkstra’s algorithm
  - Single source shortest distance (non-negative weights)
- Bellman-Ford algorithm
  - For graphs with negative weights (but no cycles with negative weight)

Shortest Path – Floyd Warshall

Background

Dijkstra's algorithm finds the shortest path between two nodes

- Run time: $O(|E| \ln(|V|))$

If we wanted to find the shortest path between all pairs of nodes, we could apply Dijkstra's algorithm to each vertex:

- Run time: $O(|V| \, |E| \ln(|V|))$

Background

Now, Dijkstra's algorithm has the following run times:

- Best case:

  If $|E| = \Theta(|V|)$, running Dijkstra for each vertex is $O(|V|^2 \ln(|V|))$

- Worst case:

  If $|E| = \Theta(|V|^2)$, running Dijkstra for each vertex is $O(|V|^3 \ln(|V|))$

Problem

Question: for the worst case, can we find a $o\left(|V|^3 \ln |V|\right)$ algorithm?

We will look at the Floyd-Warshall algorithm

- It works with positive or negative weights with no negative cycle

Strategy

First, let's consider only edges that connect vertices directly:

$$d_{i,j}^{(0)} = \begin{cases} 0 & \text{If } i = j \\ w_{i,j} & \text{If there is an edge from } i \text{ to } j \\ \infty & \text{Otherwise} \end{cases}$$

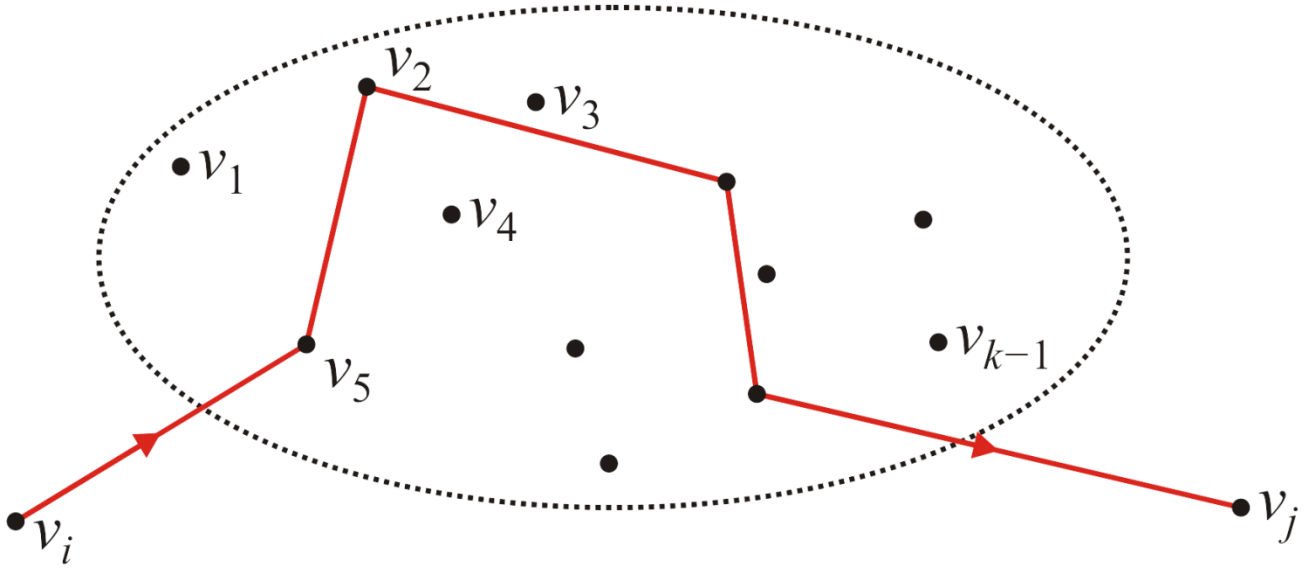
Here, $w_{i,j}$ is the weight of the edge connecting vertices i and j

- Note, this can be a directed graph; *i.e.*, it may be that $d_{i,j}^{(0)} \neq d_{j,i}^{(0)}$

The General Step

Define $d_{i,j}^{(k-1)}$ as the shortest distance, but only allowing intermediate visits to vertices $v_1, v_2, \ldots, v_{k-1}$

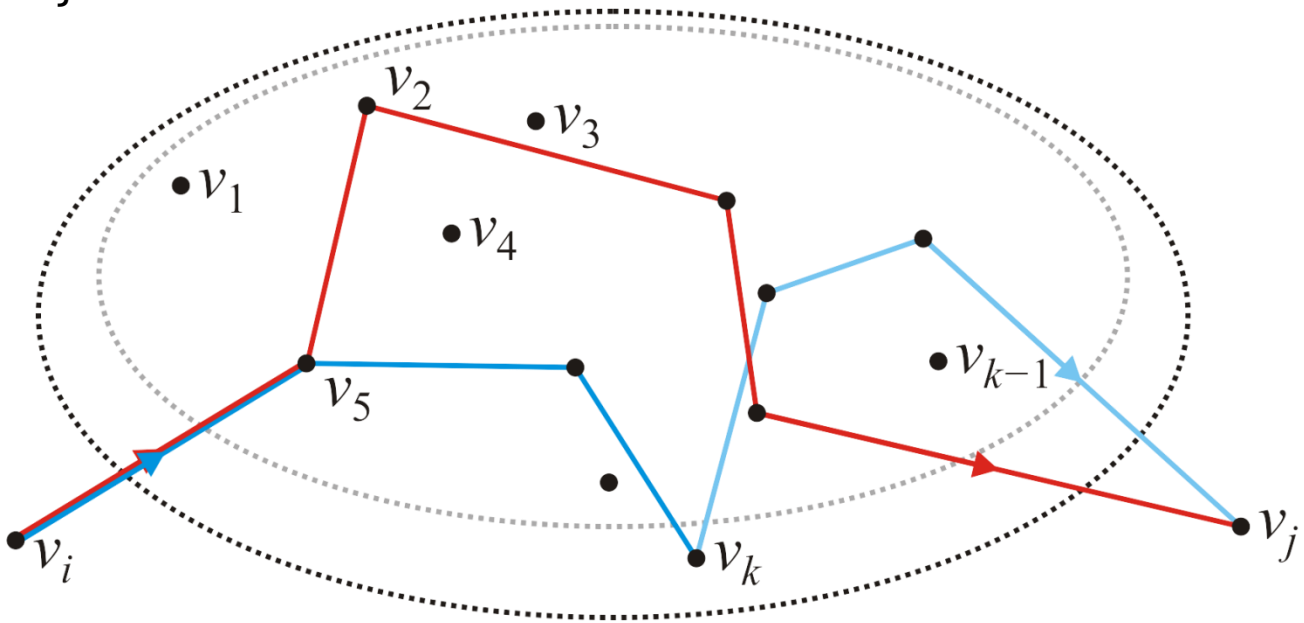
- Suppose we have an algorithm that has found these values for all pairs



The General Step

The calculation is straight forward:

```
for ( int i = 0; i < num_vertices; ++i ) {
    for ( int j = 0; j < num_vertices; ++j ) {
        d[i][j] = std::min( d[i][j], d[i][k-1] + d[k-1][j] );
    }
}
```



The Floyd-Warshall Algorithm

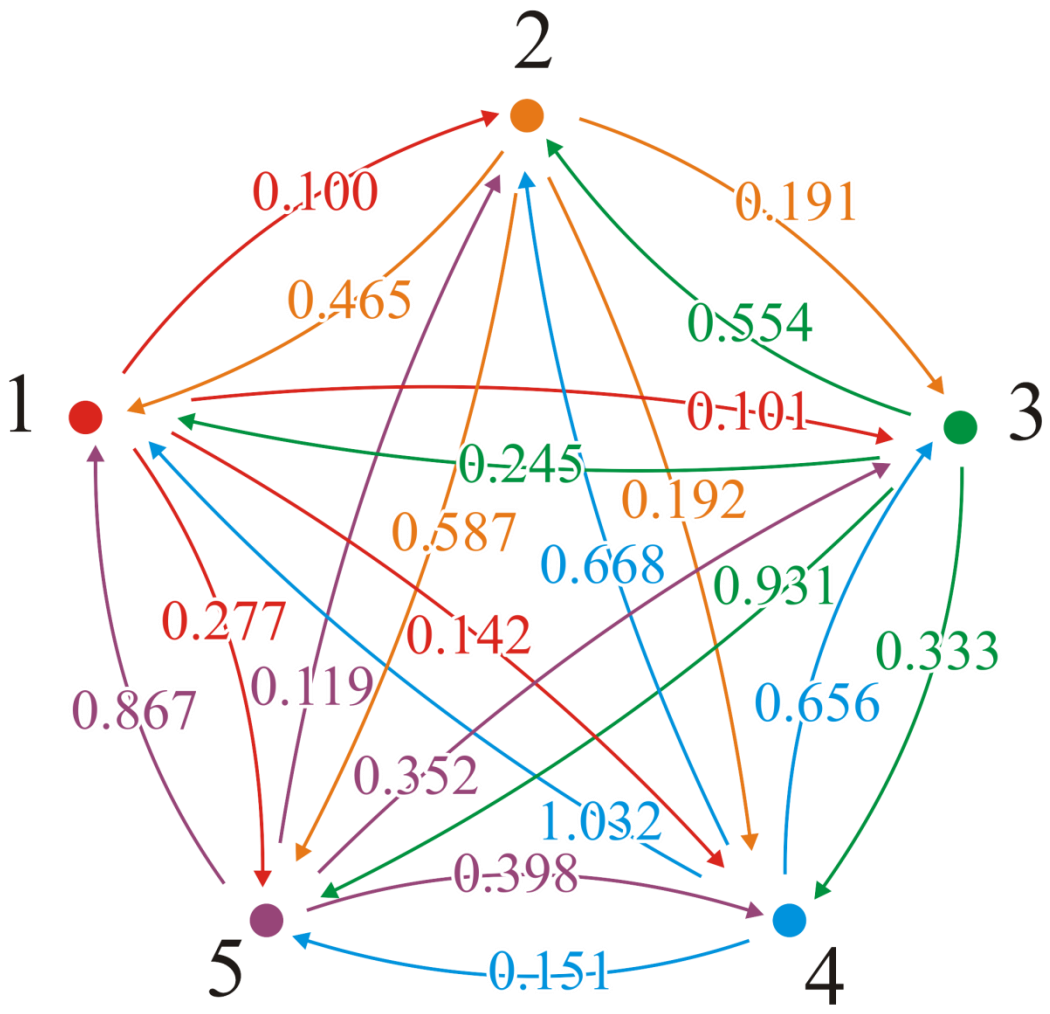
```
// Initialize the matrix d
//   ...

for ( int k = 0; k < num_vertices; ++k ) {
    for ( int i = 0; i < num_vertices; ++i ) {
        for ( int j = 0; j < num_vertices; ++j ) {
            d[i][j] = std::min( d[i][j], d[i][k] + d[k][j] );
        }
    }
}
```

Run time? $\Theta\left(|V|^3\right)$

Example

Consider this graph

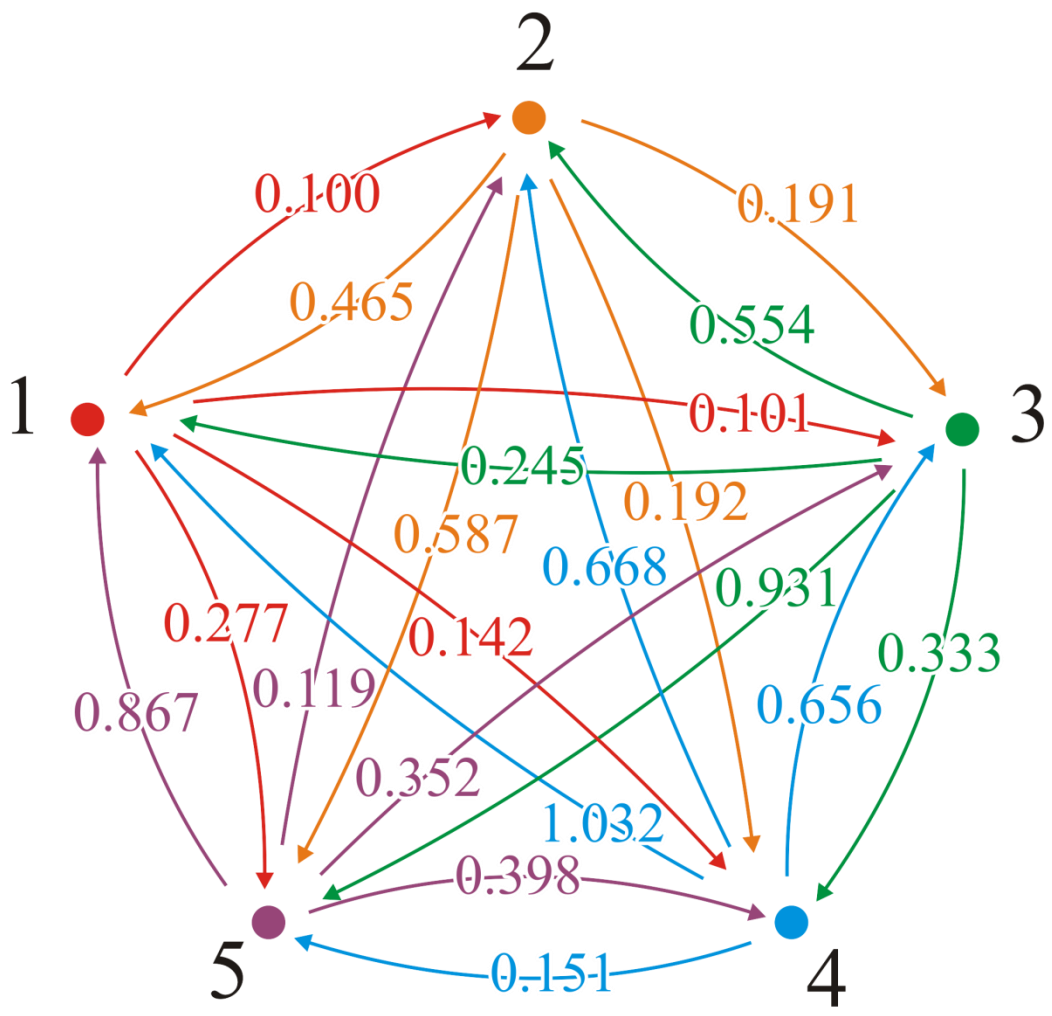


Example

The adjacency matrix is

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.554 & 0 & 0.333 & 0.931 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.867 & 0.119 & 0.352 & 0.398 & 0 \end{pmatrix}$$

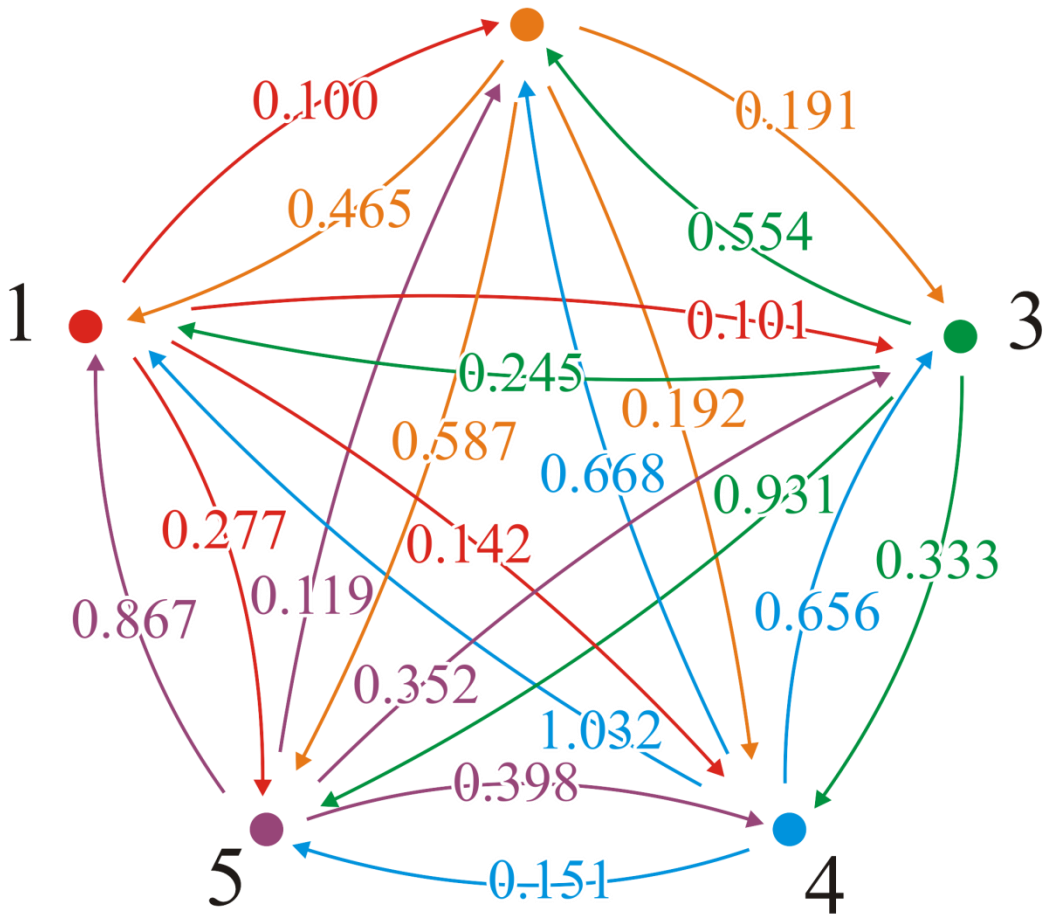
This would define our matrix $\mathbf{D} = (d_{ij})$



Example

With the first pass, $k = 1$, we attempt passing through vertex v_1

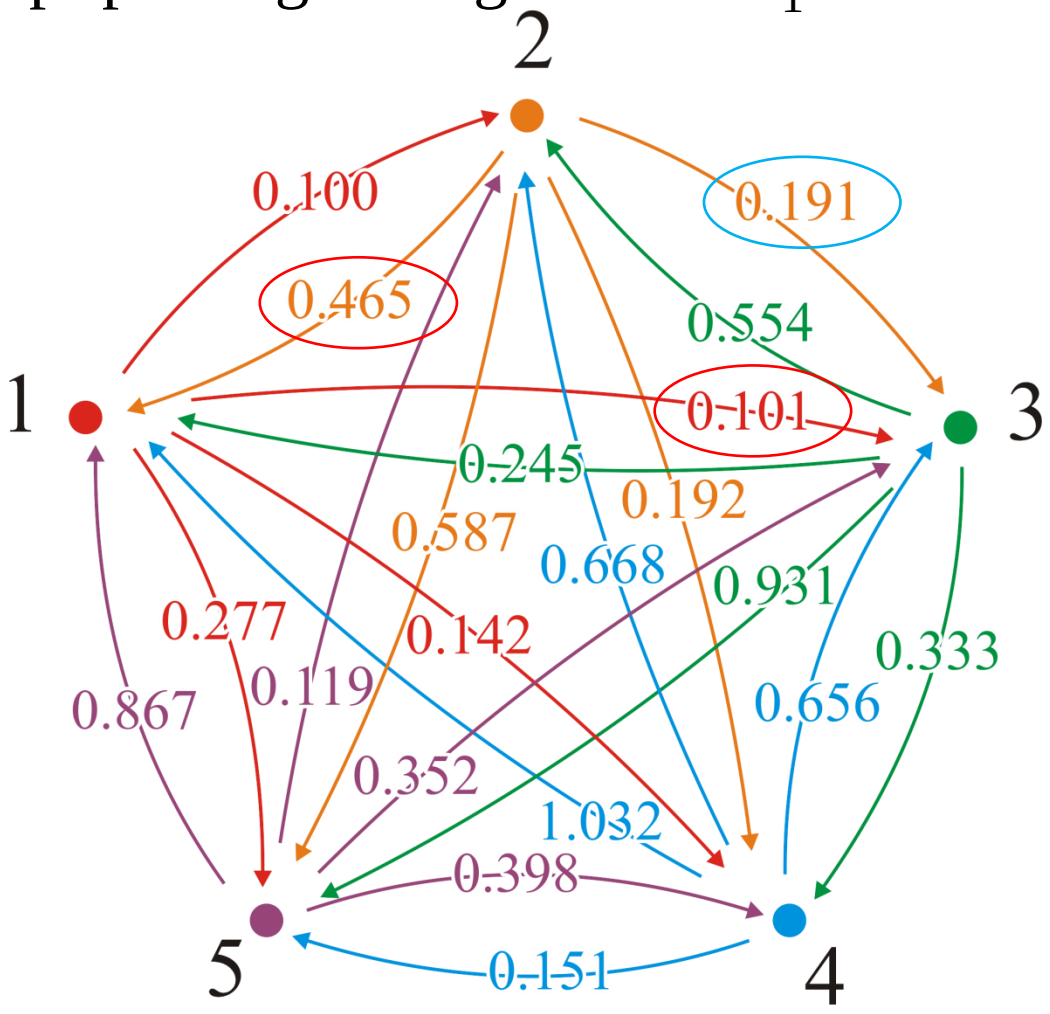
$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.554 & 0 & 0.333 & 0.931 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.867 & 0.119 & 0.352 & 0.398 & 0 \end{pmatrix}$$



Example

With the first pass, $k = 1$, we attempt passing through vertex v_1

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.554 & 0 & 0.333 & 0.931 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.867 & 0.119 & 0.352 & 0.398 & 0 \end{pmatrix}$$



We would start:

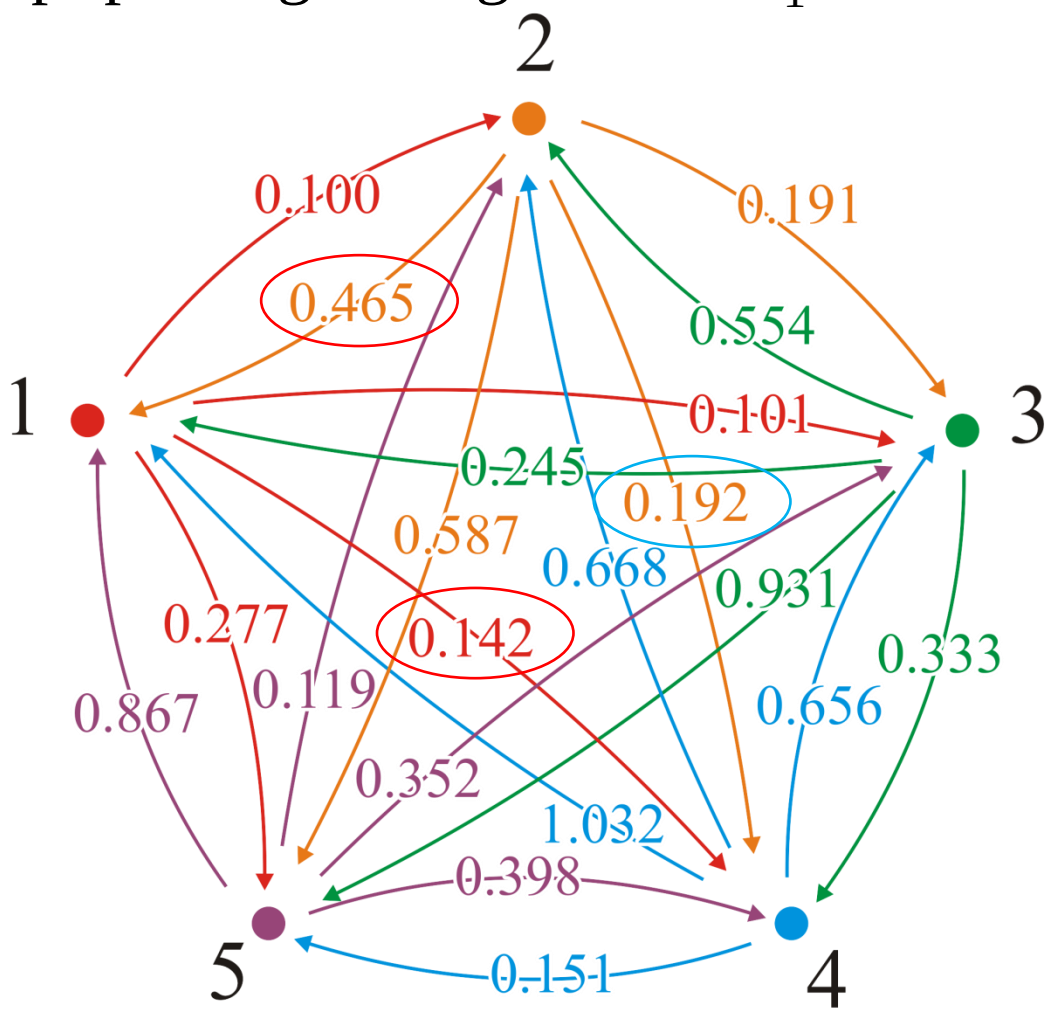
$(2, 3) \rightarrow (2, 1, 3)$

$0.191 \not> 0.465 + 0.101$

Example

With the first pass, $k = 1$, we attempt passing through vertex v_1

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.554 & 0 & 0.333 & 0.931 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.867 & 0.119 & 0.352 & 0.398 & 0 \end{pmatrix}$$



We would start:

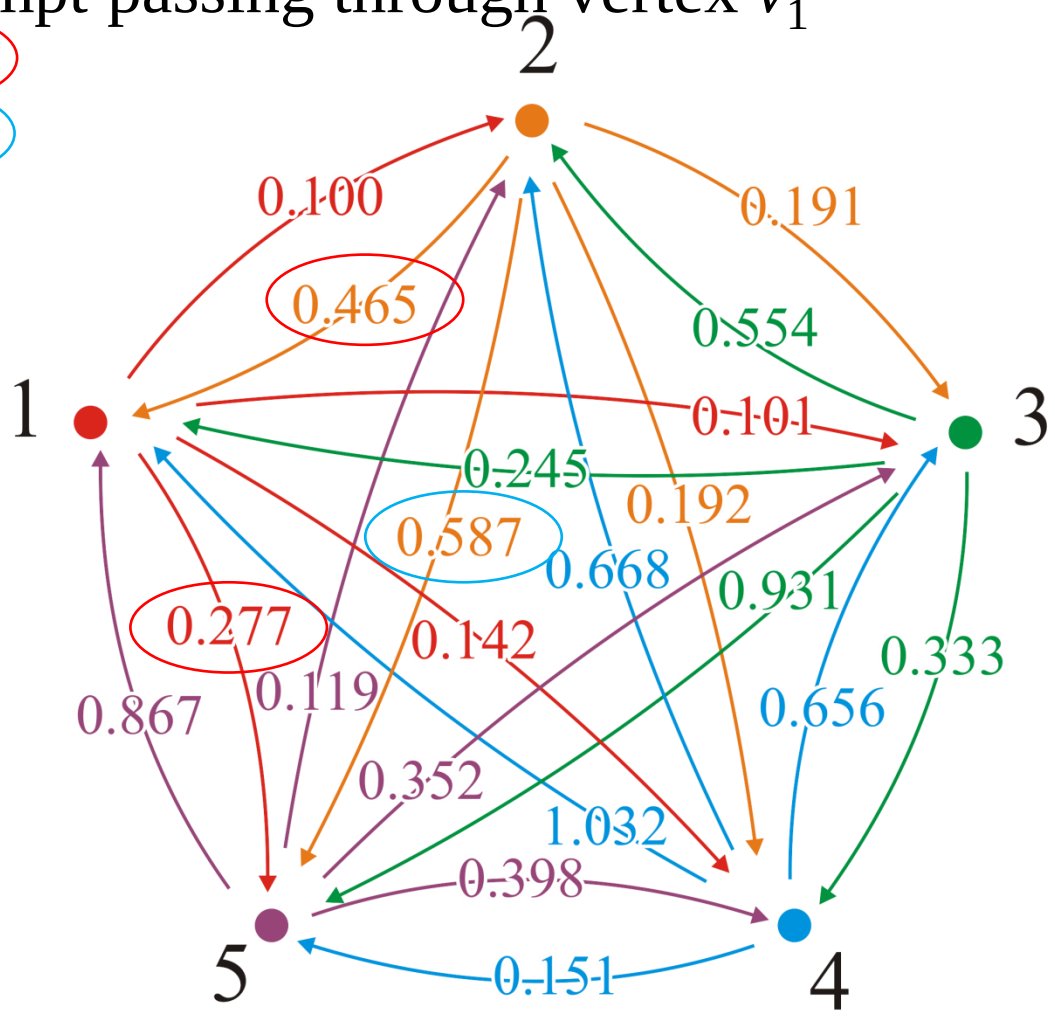
$(2, 4) \rightarrow (2, 1, 4)$

$0.192 \not> 0.465 + 0.142$

Example

With the first pass, $k = 1$, we attempt passing through vertex v_1

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.554 & 0 & 0.333 & 0.931 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.867 & 0.119 & 0.352 & 0.398 & 0 \end{pmatrix}$$



We would start:

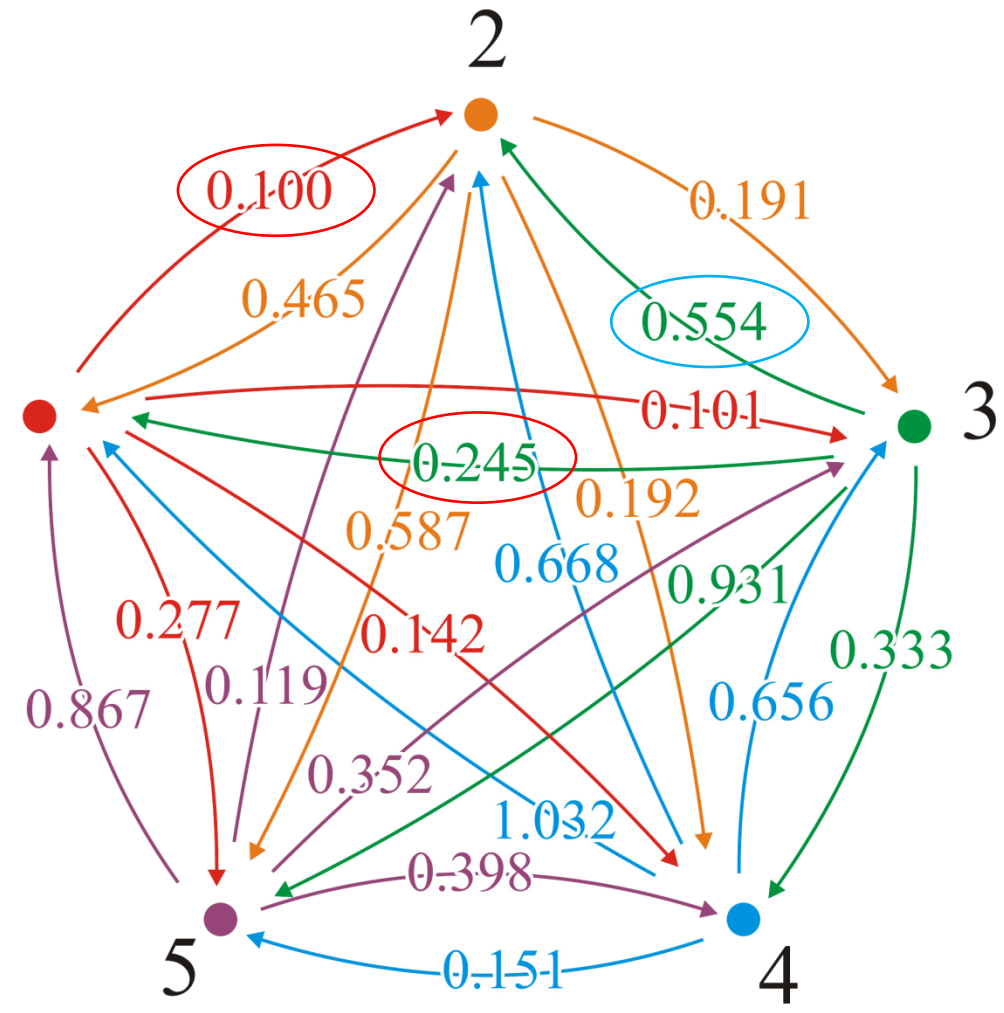
$(2, 5) \rightarrow (2, 1, 5)$

$0.587 \not\succ 0.465 + 0.277$

Example

With the first pass, $k = 1$, we attempt passing through vertex v_1

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.554 & 0 & 0.333 & 0.931 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.867 & 0.119 & 0.352 & 0.398 & 0 \end{pmatrix}$$



Here is a shorter path:

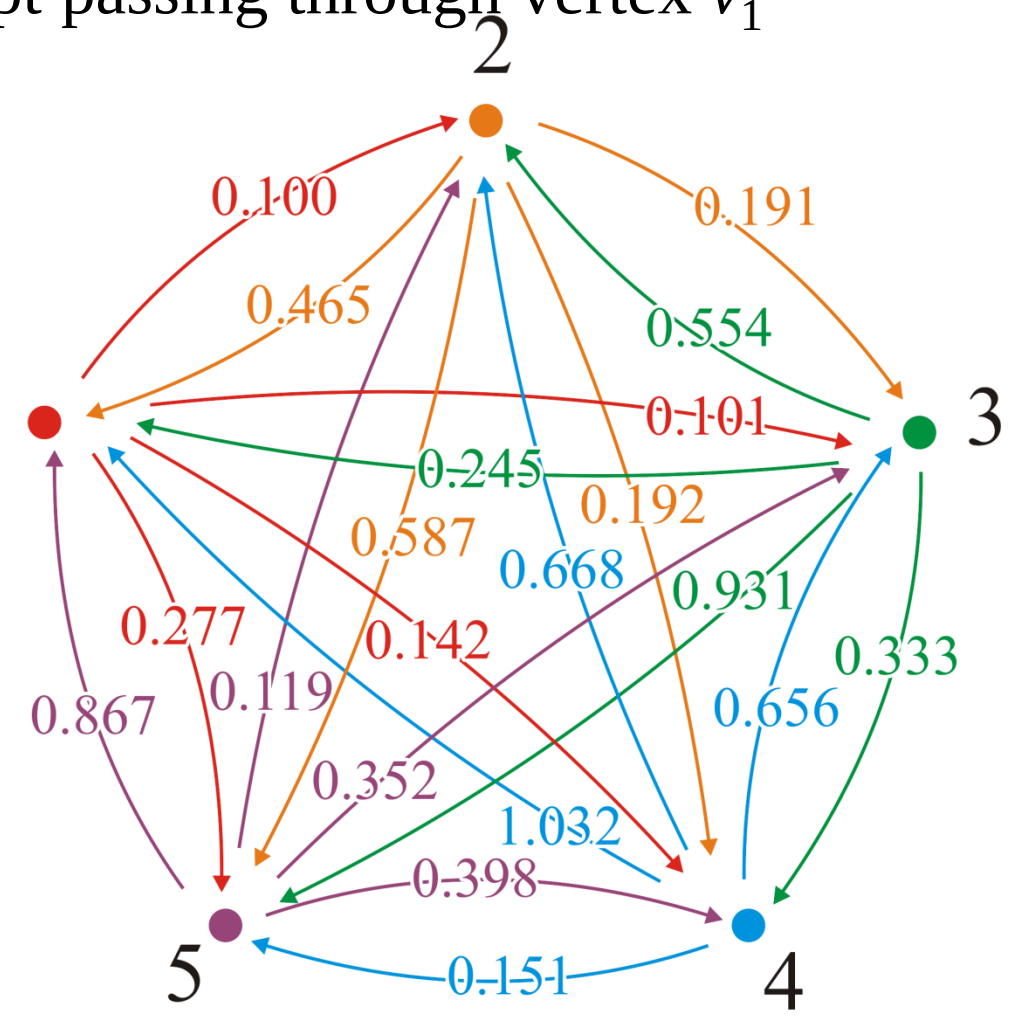
$(3, 2) \rightarrow (3, 1, 2)$

$0.554 > 0.245 + 0.100 = 0.345$

Example

With the first pass, $k = 1$, we attempt passing through vertex v_1

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.345 & 0 & 0.333 & 0.931 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.867 & 0.119 & 0.352 & 0.398 & 0 \end{pmatrix}$$



We update the table

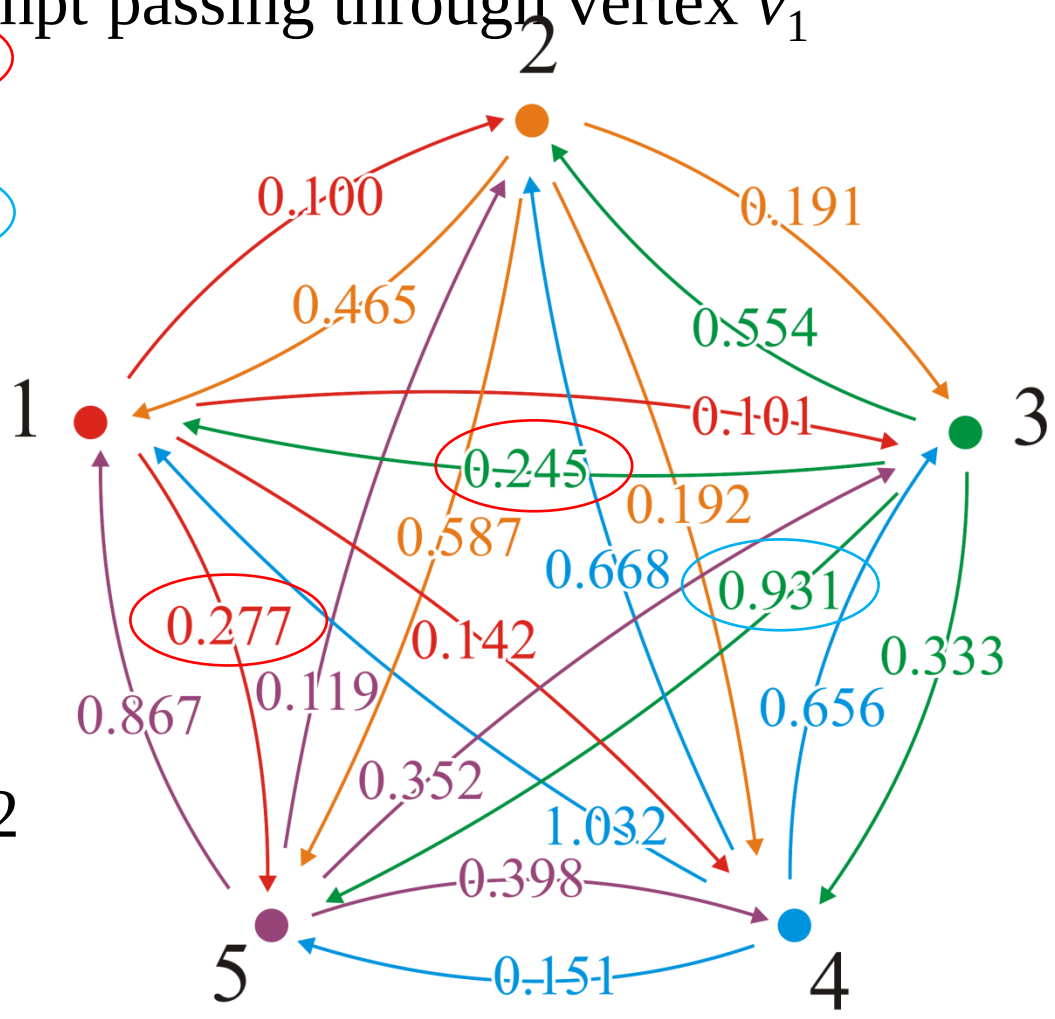
$(3, 2) \rightarrow (3, 1, 2)$

$0.554 > 0.245 + 0.100 = 0.345$

Example

With the first pass, $k = 1$, we attempt passing through vertex v_1

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.345 & 0 & 0.333 & 0.931 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.867 & 0.119 & 0.352 & 0.398 & 0 \end{pmatrix}$$



And a second shorter path:

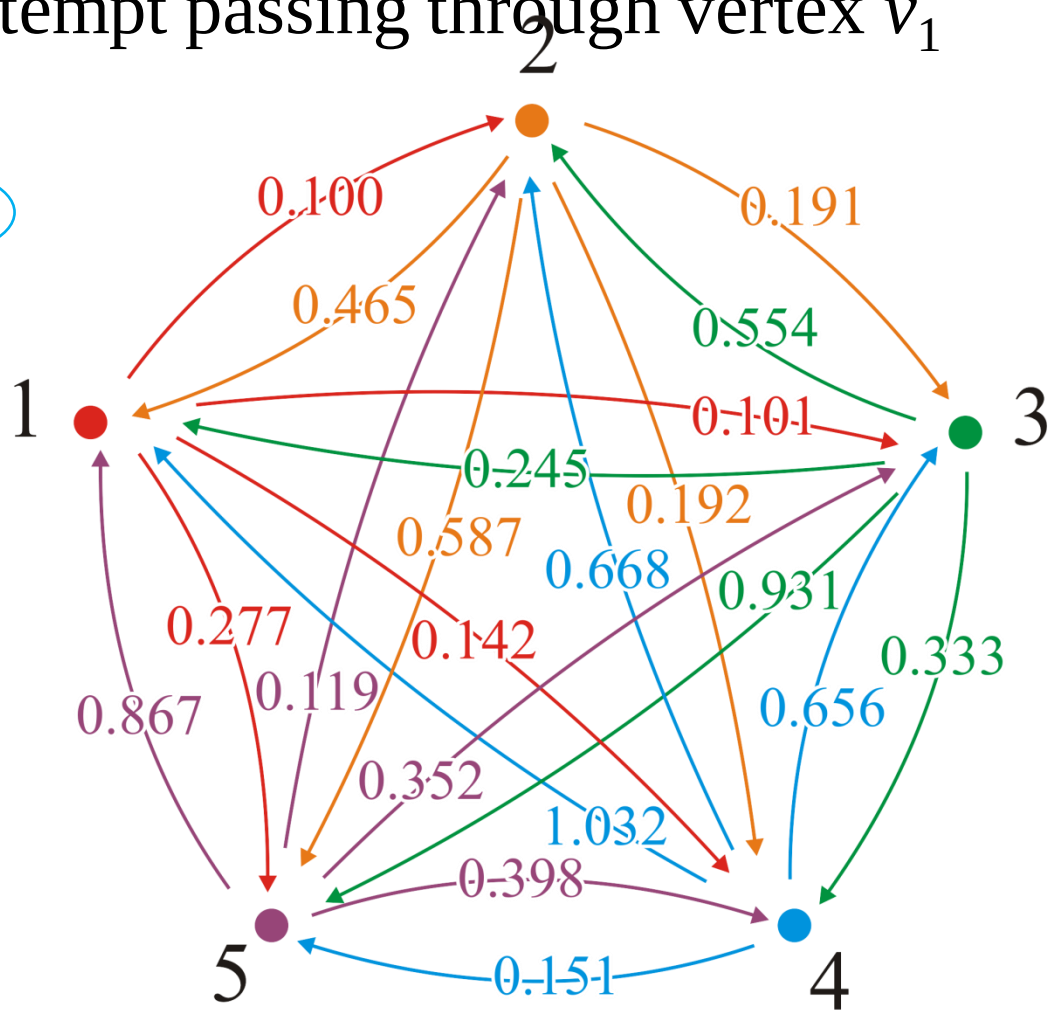
$(3, 5) \rightarrow (3, 1, 5)$

$$0.931 > 0.245 + 0.277 = 0.522$$

Example

With the first pass, $k = 1$, we attempt passing through vertex v_1

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.345 & 0 & 0.333 & 0.522 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.867 & 0.119 & 0.352 & 0.398 & 0 \end{pmatrix}$$

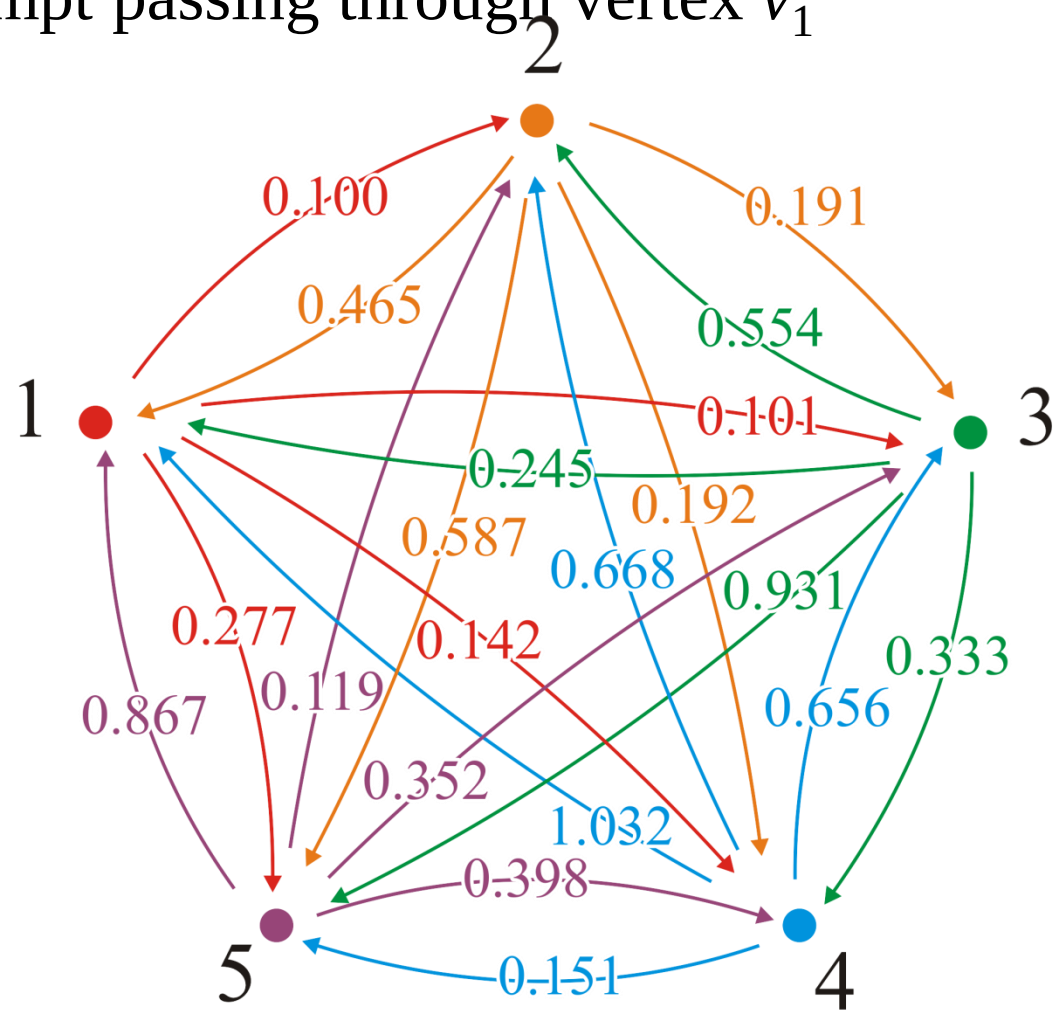


We update the table

Example

With the first pass, $k = 1$, we attempt passing through vertex v_1

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.345 & 0 & 0.333 & 0.522 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.867 & 0.119 & 0.352 & 0.398 & 0 \end{pmatrix}$$



Continuing…
We find that no other shorter paths through vertex v_1 exist

Example

With the next pass, $k = 2$, we attempt passing through vertex v_2

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.345 & 0 & 0.333 & 0.522 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.867 & 0.119 & 0.352 & 0.398 & 0 \end{pmatrix}$$

There are three shorter paths:

(5, 1) → (5, 2, 1)

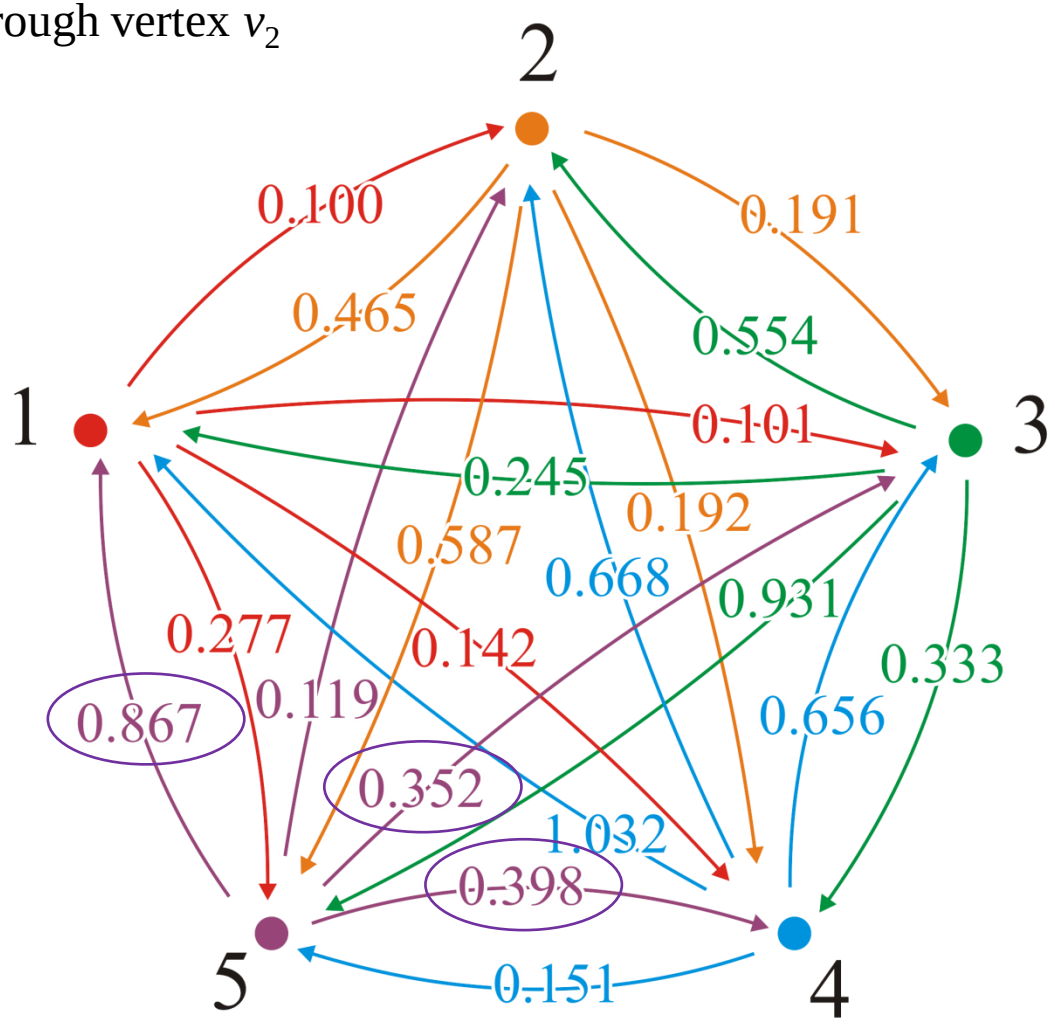
0.867 > 0.119 + 0.465 = 0.584

(5, 3) → (5, 2, 3)

0.352 > 0.119 + 0.191 = 0.310

(5, 4) → (5, 2, 4)

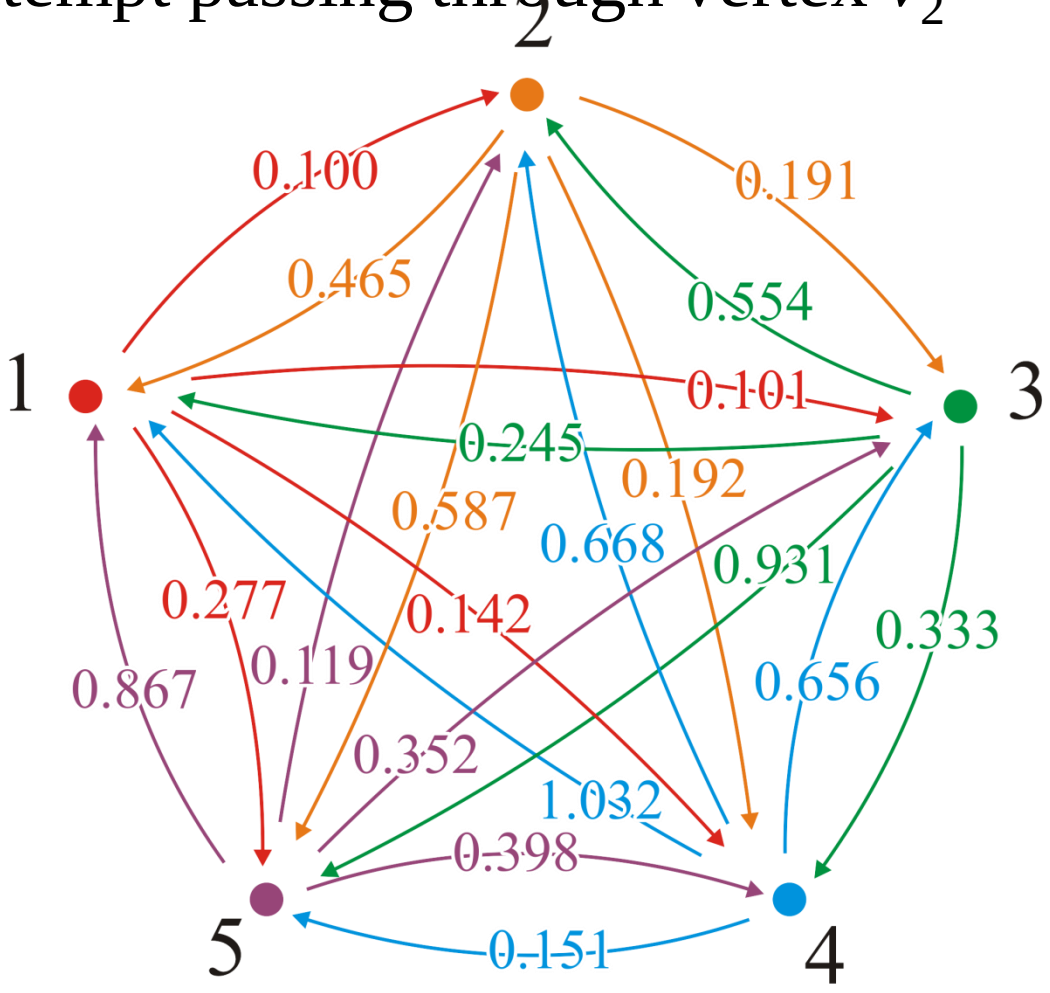
0.398 > 0.119 + 0.192 = 0.311



Example

With the next pass, $k = 2$, we attempt passing through vertex v_2

0	0.100	0.101	0.142	0.277
0.465	0	0.191	0.192	0.587
0.245	0.345	0	0.333	0.522
1.032	0.668	0.656	0	0.151
0.584	0.119	0.310	0.311	0



We update the table

Example

With the next pass, $k = 3$, we attempt passing through vertex v_3

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.465 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.345 & 0 & 0.333 & 0.522 \\ 1.032 & 0.668 & 0.656 & 0 & 0.151 \\ 0.584 & 0.119 & 0.310 & 0.311 & 0 \end{pmatrix}$$

There are three shorter paths:

(2, 1) → (2, 3, 1)

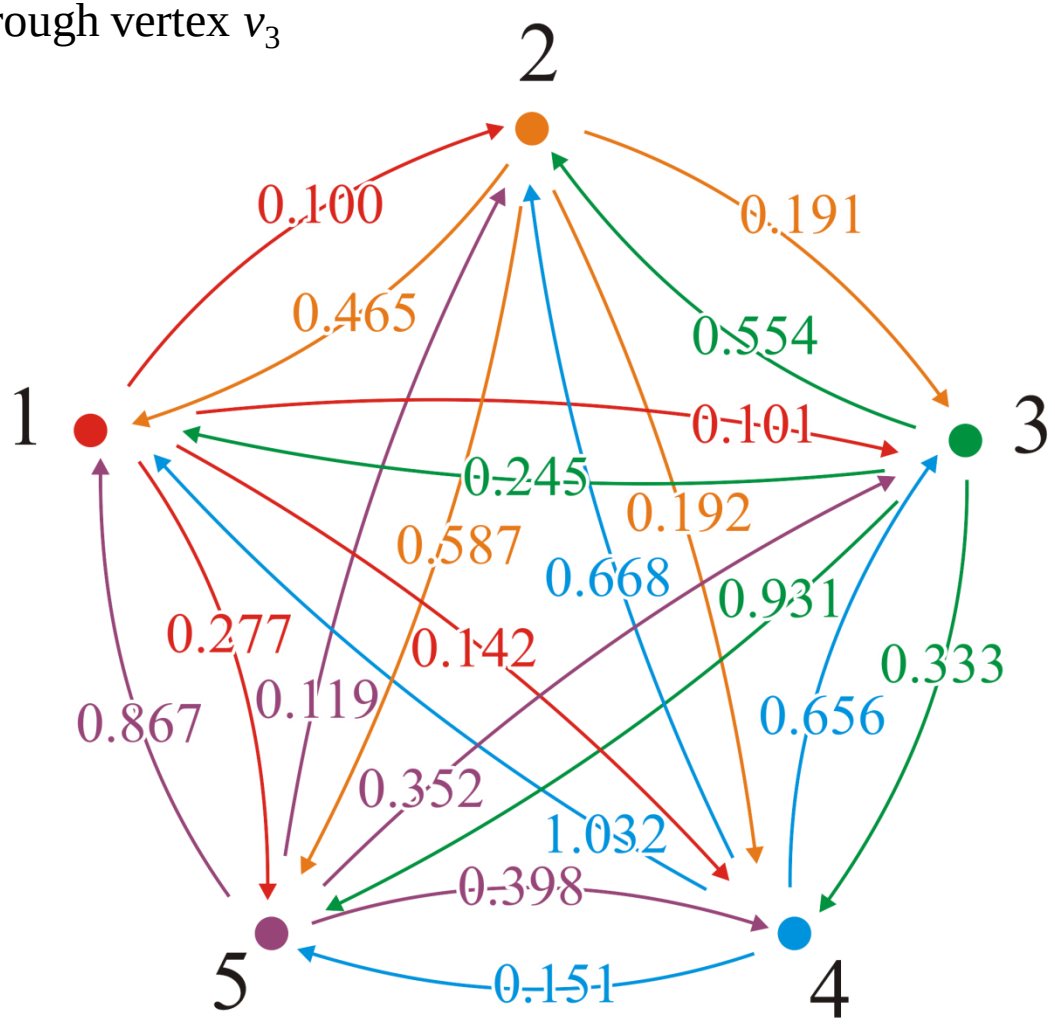
0.465 > 0.191 + 0.245 = 0.436

(4, 1) → (4, 3, 1)

1.032 > 0.656 + 0.245 = 0.901

(5, 1) → (5, 3, 1)

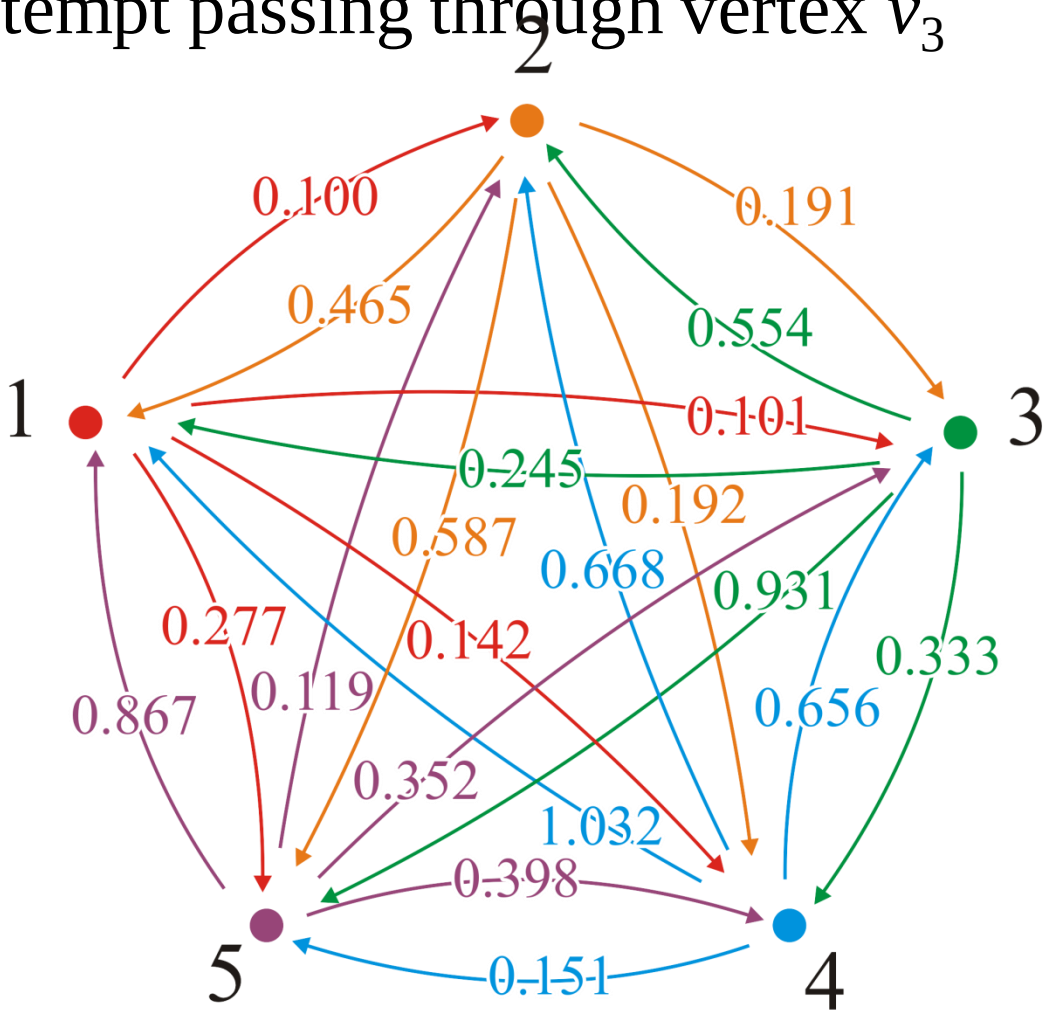
0.584 > 0.310 + 0.245 = 0.555



Example

With the next pass, $k = 3$, we attempt passing through vertex v_3

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.436 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.345 & 0 & 0.333 & 0.522 \\ 0.901 & 0.668 & 0.656 & 0 & 0.151 \\ 0.555 & 0.119 & 0.310 & 0.311 & 0 \end{pmatrix}$$



We update the table

Example

With the next pass, $k = 4$, we attempt passing through vertex v_4

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.436 & 0 & 0.191 & 0.192 & 0.587 \\ 0.245 & 0.345 & 0 & 0.333 & 0.522 \\ 0.901 & 0.668 & 0.656 & 0 & 0.151 \\ 0.555 & 0.119 & 0.310 & 0.311 & 0 \end{pmatrix}$$

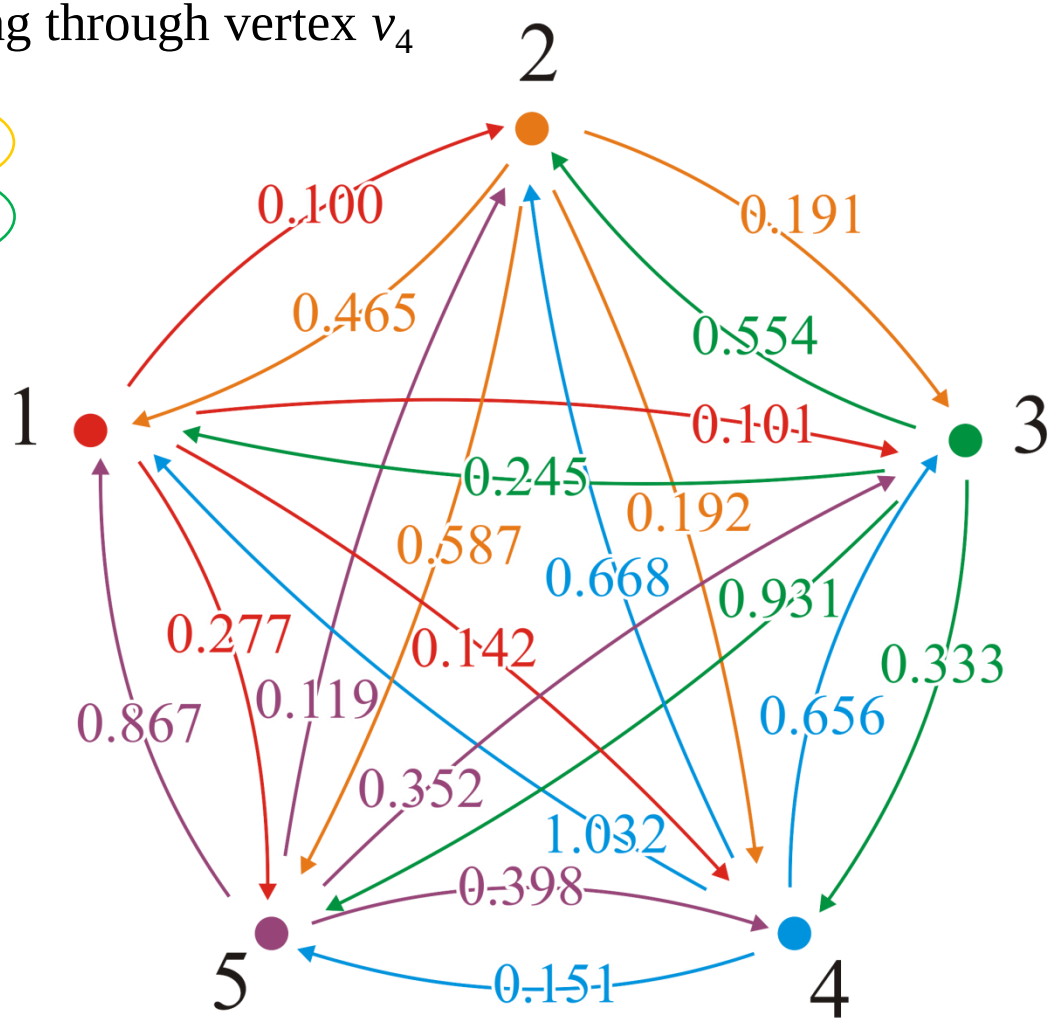
There are two shorter paths:

$(2, 5) \rightarrow (2, 4, 5)$

$0.587 > 0.192 + 0.151$

$(3, 5) \rightarrow (3, 4, 5)$

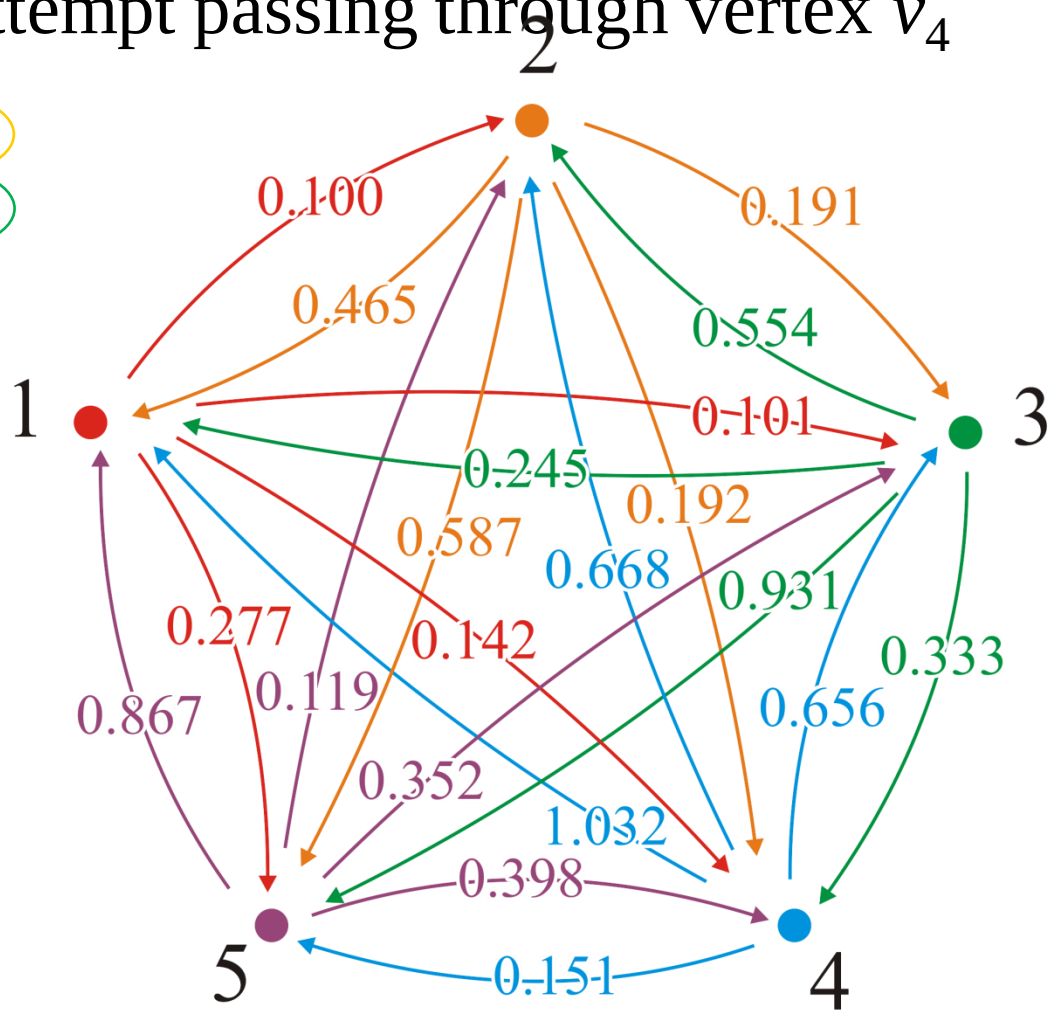
$0.522 > 0.333 + 0.151$



Example

With the next pass, $k = 4$, we attempt passing through vertex v_4

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.436 & 0 & 0.191 & 0.192 & 0.343 \\ 0.245 & 0.345 & 0 & 0.333 & 0.484 \\ 0.901 & 0.668 & 0.656 & 0 & 0.151 \\ 0.555 & 0.119 & 0.310 & 0.311 & 0 \end{pmatrix}$$



We update the table

Example

With the last pass, $k = 5$, we attempt passing through vertex v_5

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.436 & 0 & 0.191 & 0.192 & 0.343 \\ 0.245 & 0.345 & 0 & 0.333 & 0.484 \\ 0.901 & 0.668 & 0.656 & 0 & 0.151 \\ 0.555 & 0.119 & 0.310 & 0.311 & 0 \end{pmatrix}$$

There are three shorter paths:

(4, 1) → (4, 5, 1)

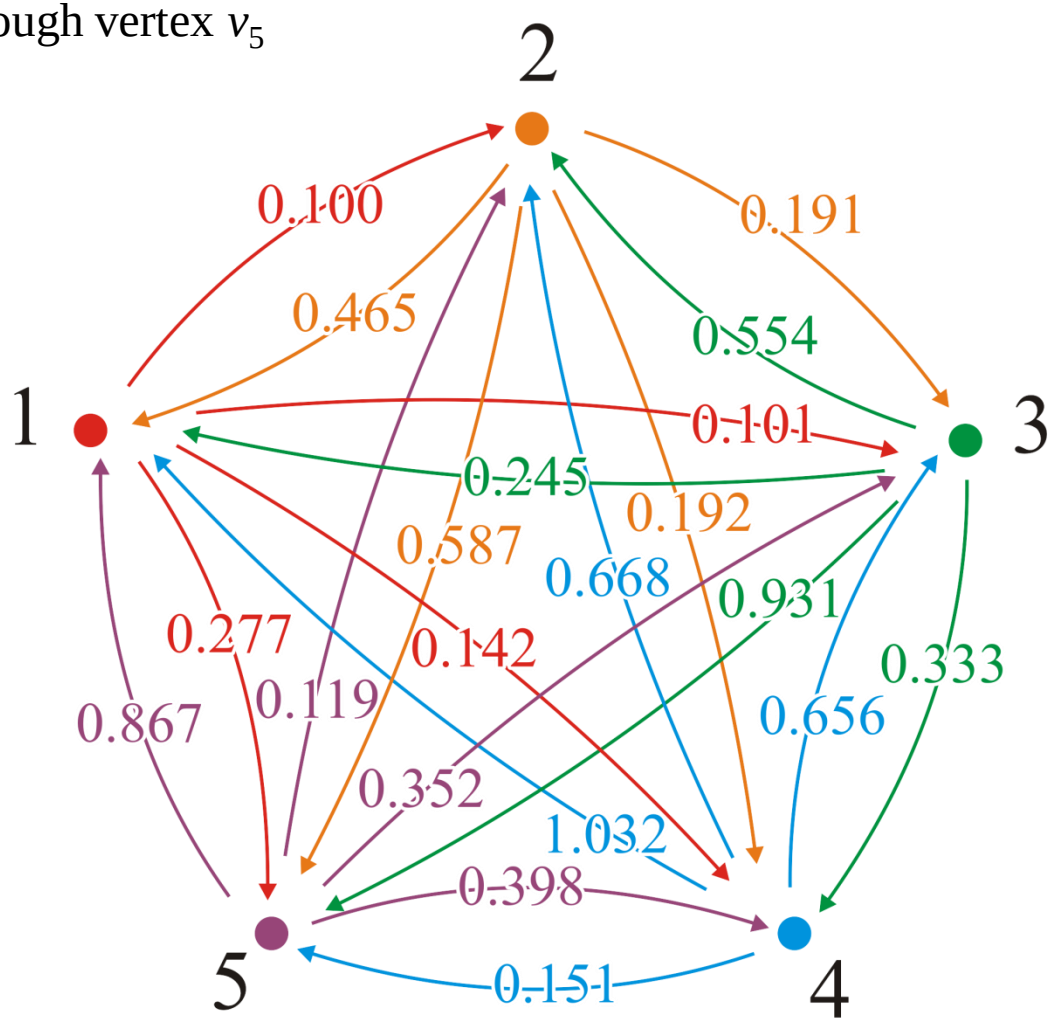
0.901 > 0.151 + 0.555 = 0.706

(4, 2) → (4, 5, 2)

0.668 > 0.151 + 0.119 = 0.270

(4, 3) → (4, 5, 3)

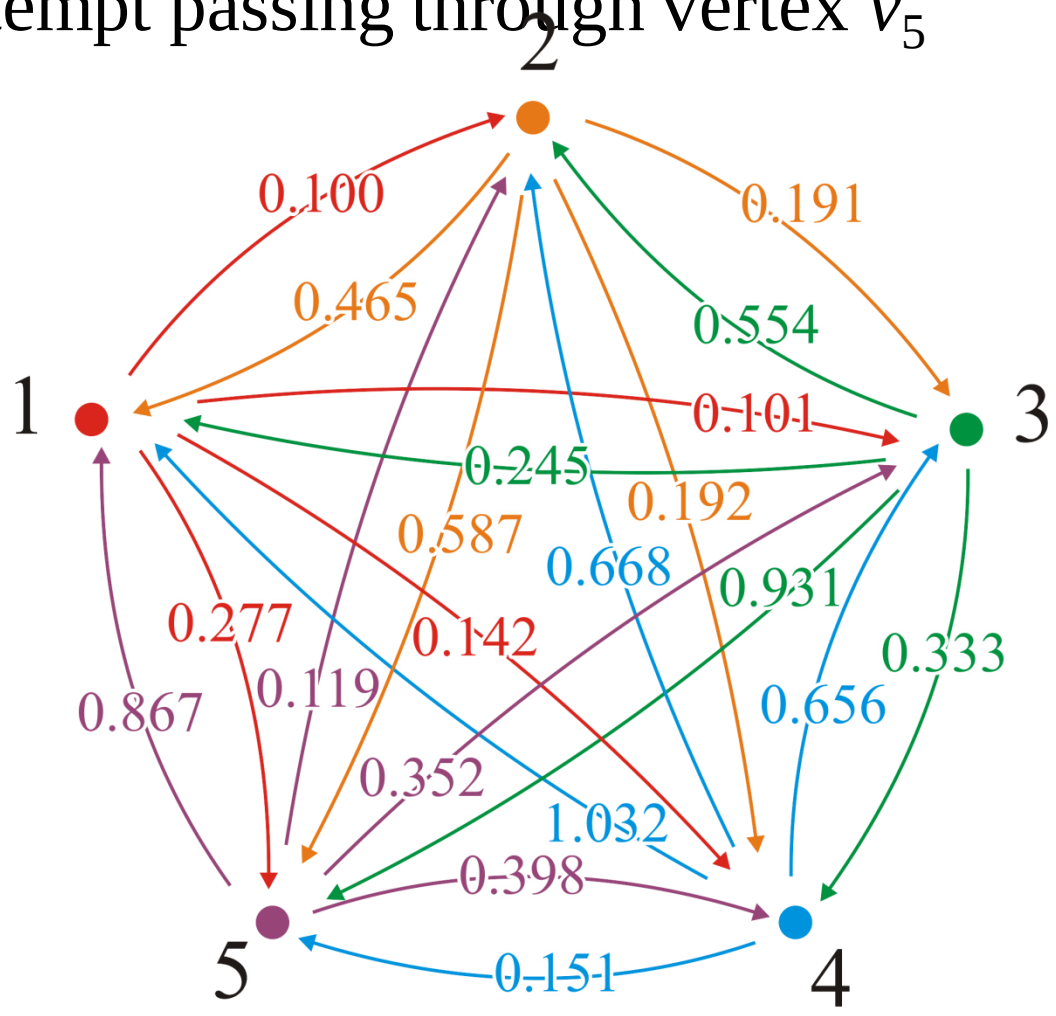
0.656 > 0.151 + 0.310 = 0.461



Example

With the last pass, $k = 5$, we attempt passing through vertex v_5

0	0.100	0.101	0.142	0.277
0.436	0	0.191	0.192	0.343
0.245	0.345	0	0.333	0.484
0.706	0.270	0.461	0	0.151
0.555	0.119	0.310	0.311	0

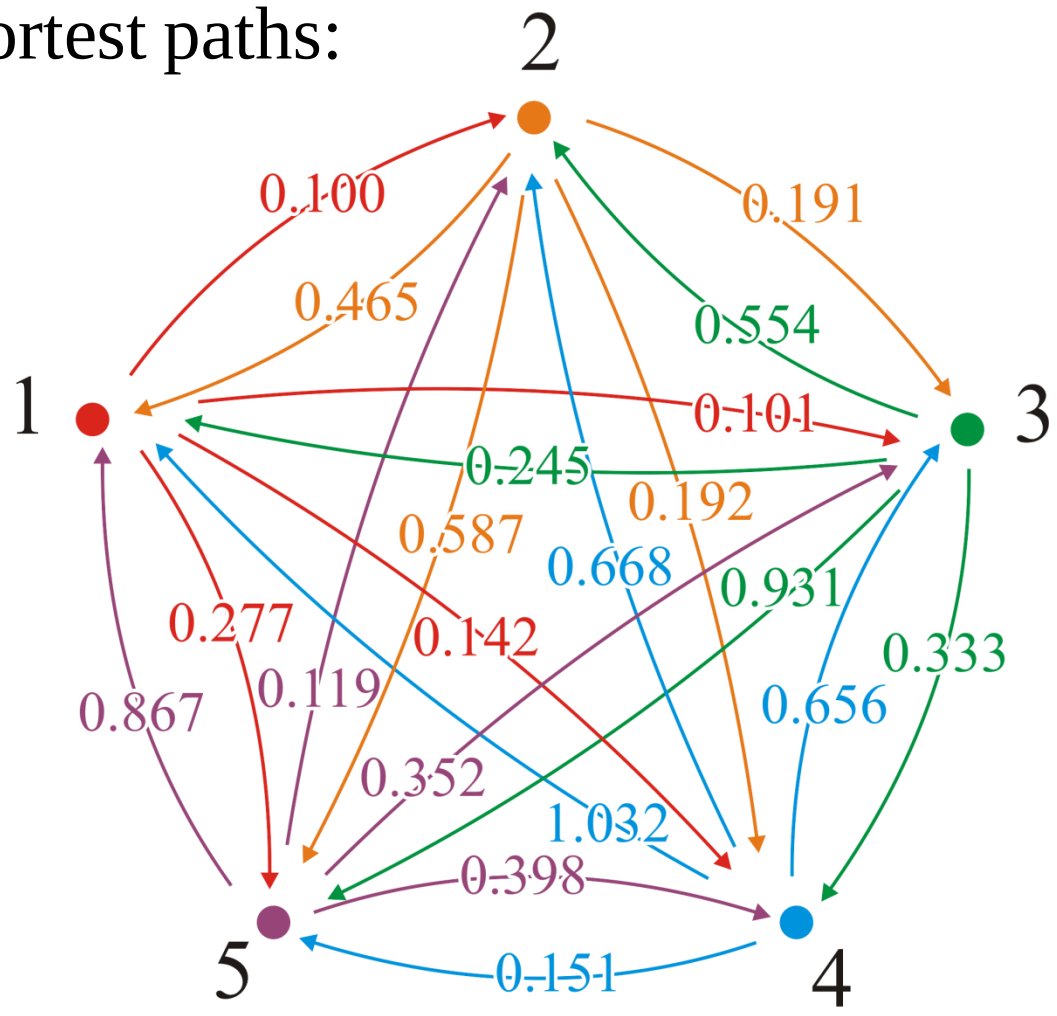


We update the table

Example

Thus, we have a table of all shortest paths:

$$\begin{pmatrix} 0 & 0.100 & 0.101 & 0.142 & 0.277 \\ 0.436 & 0 & 0.191 & 0.192 & 0.343 \\ 0.245 & 0.345 & 0 & 0.333 & 0.484 \\ 0.706 & 0.270 & 0.461 & 0 & 0.151 \\ 0.555 & 0.119 & 0.310 & 0.311 & 0 \end{pmatrix}$$



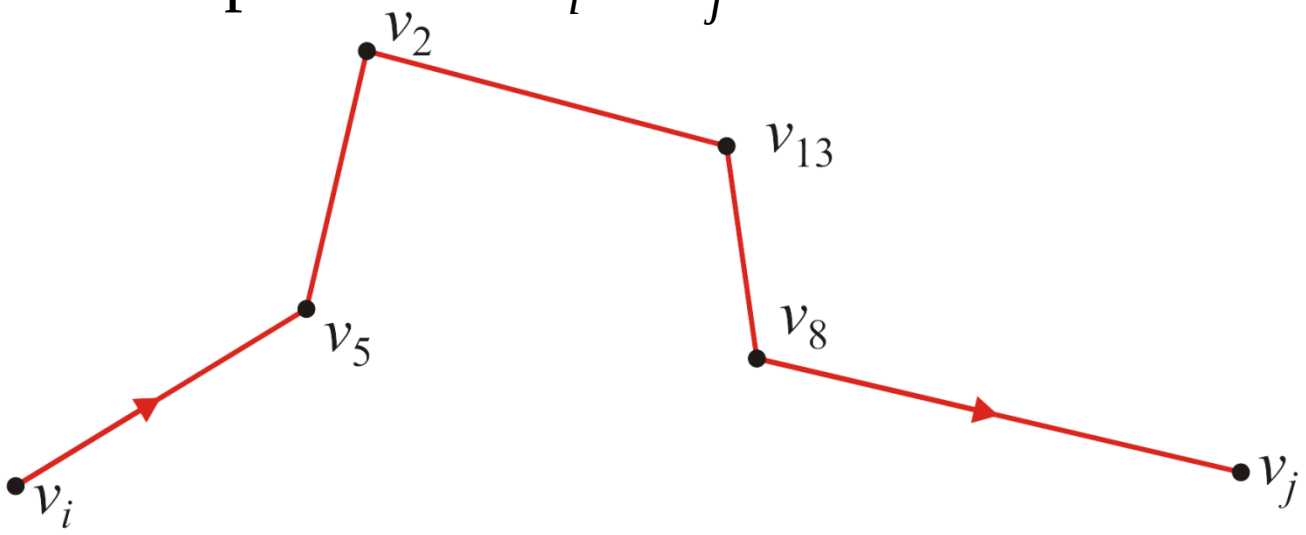
What Is the Shortest Path?

This algorithm finds the shortest distances, but what are the paths corresponding to those shortest distances?

- Recall that with Dijkstra's algorithm, we could find the shortest paths by recording the previous node
- Here we use a similar approach, but we choose to store the next node instead of the previous node

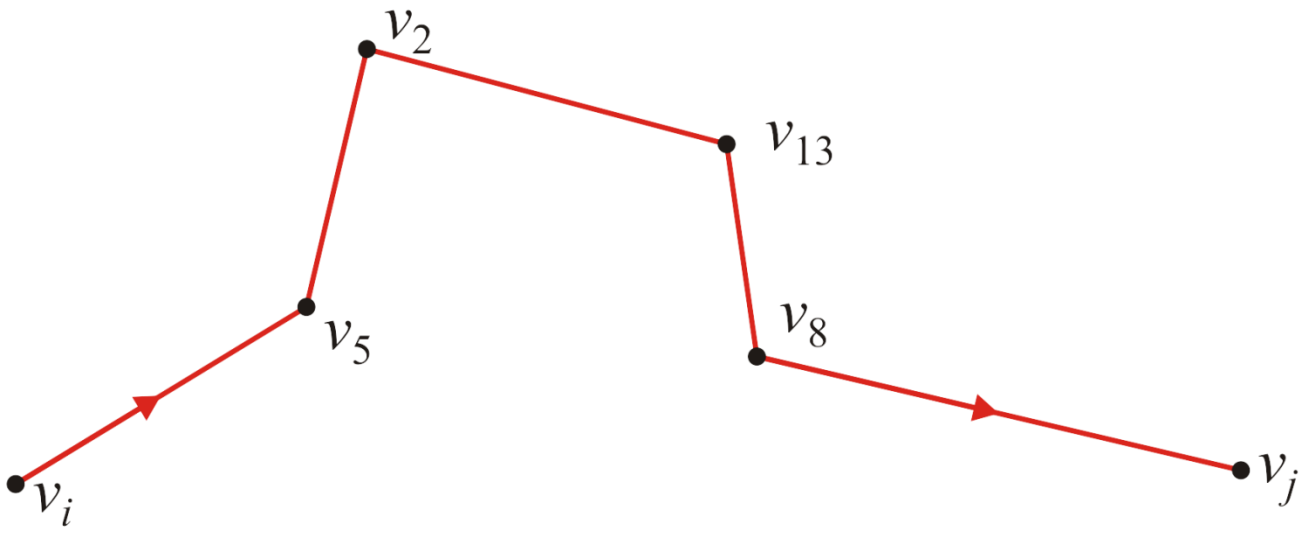
What Is the Shortest Path?

Suppose the shortest path from v_i to v_j is as follows:



What Is the Shortest Path?

Does this path consist of (v_i, v_5) and the shortest path from v_5 to v_j?

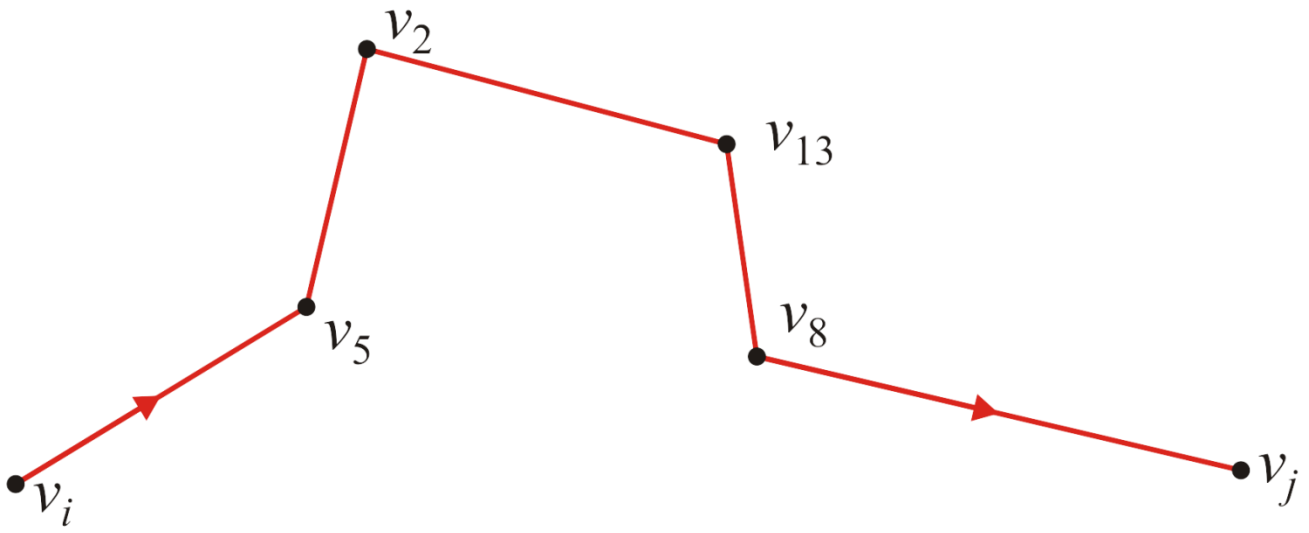


Yes

- If there was a shorter path from v_5 to v_j, then we would also find a shorter path from v_i to v_j

What Is the Shortest Path?

Does this path consist of (v_i, v_5) and the shortest path from v_5 to v_j?

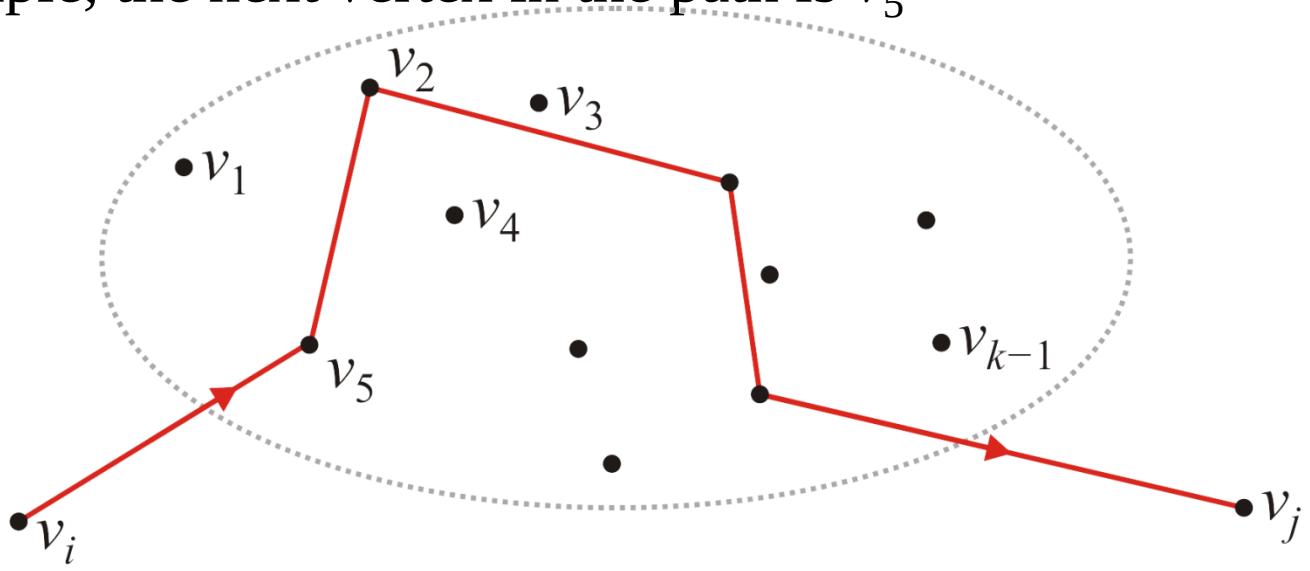


To find the shortest path from v_i to v_j, we only need to know that v_5 is the next vertex in the path — the rest of the path would be recursively recovered as the shortest path from v_5 to v_j

What Is the Shortest Path?

Now, suppose we have the shortest path from v_i to v_j which passes through the vertices $v_1, v_2, \ldots, v_{k-1}$

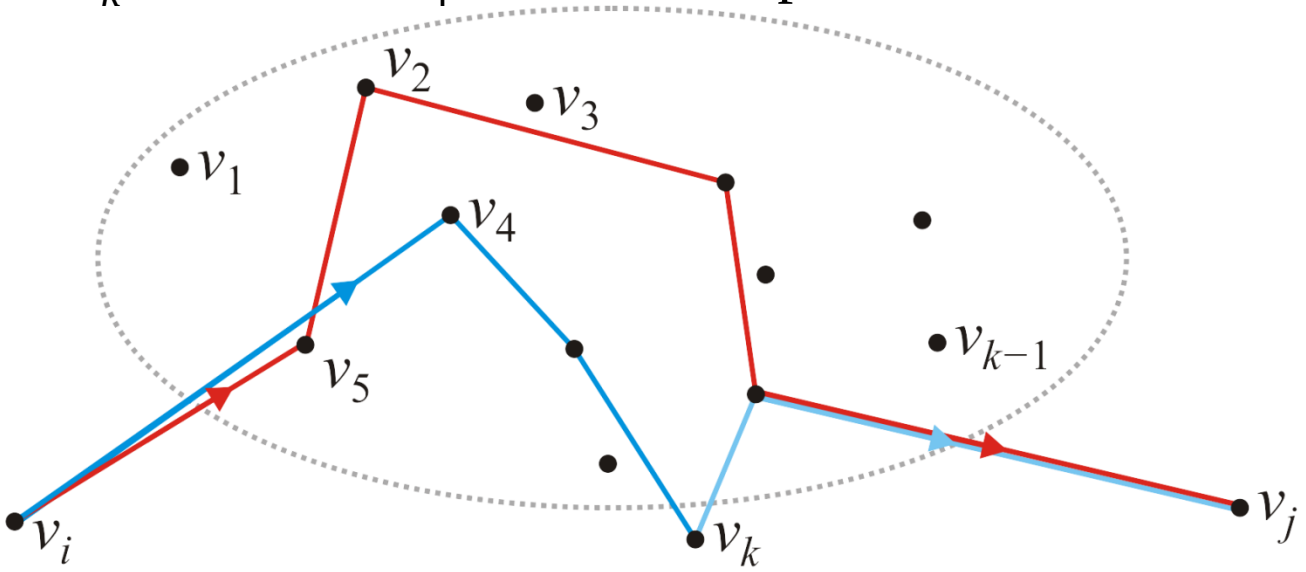
- In this example, the next vertex in the path is v_5



What Is the Shortest Path?

What if we find a shorter path passing through v_k?

- Now the next vertex in the new path should be the next vertex in the shortest path from v_i to v_k , which is v_4 in this example



What Is the Shortest Path?

Let us store the next vertex in the shortest path. Initially:

$$p_{i,j} = \begin{cases} \varnothing & \text{If } i = j \\ j & \text{If there is an edge from } i \text{ to } j \\ \varnothing & \text{Otherwise} \end{cases}$$

What Is the Shortest Path?

When we find a shorter path, update the next node: $p_{i,j} = p_{i,k}$

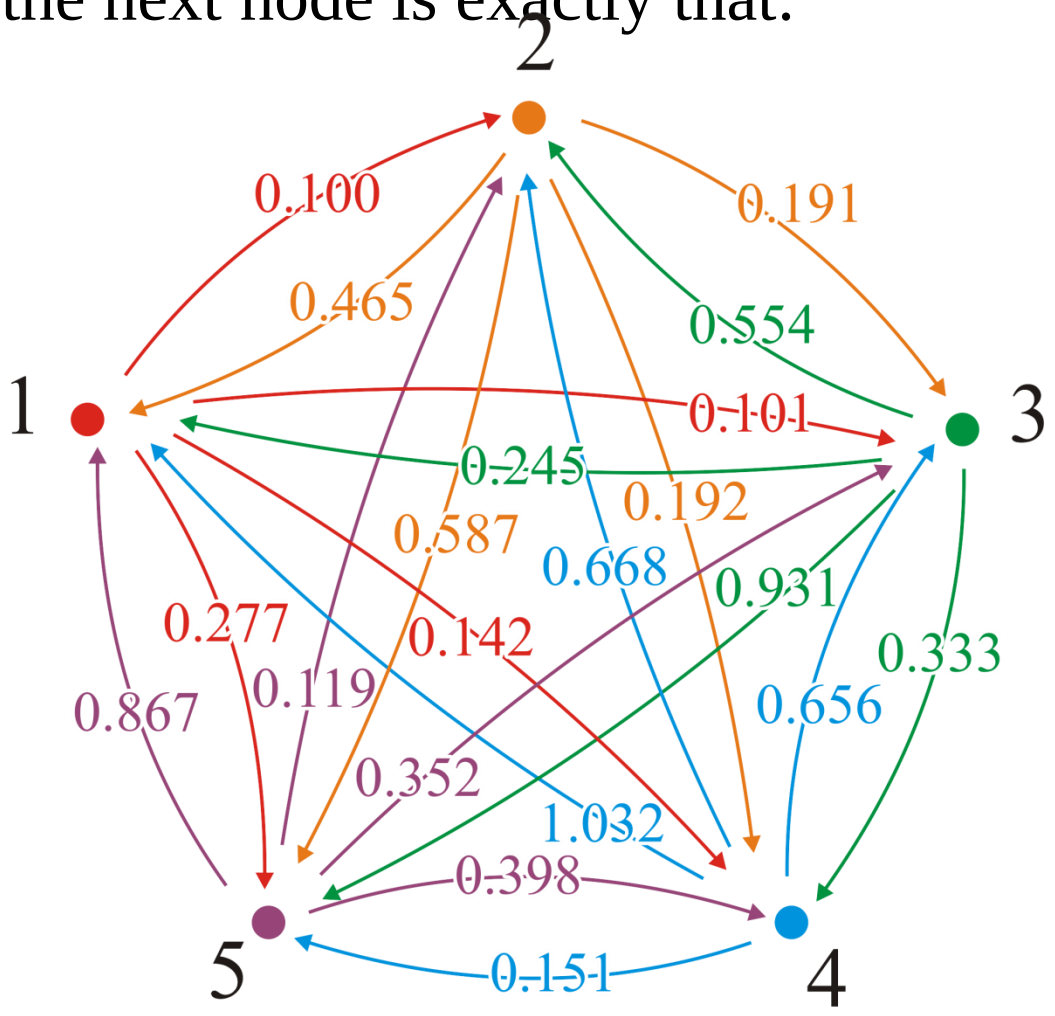
```
// Initialize the matrix p
//   ...

for ( int k = 0; k < num_vertices; ++k ) {
    for ( int i = 0; i < num_vertices; ++i ) {
        for ( int j = 0; j < num_vertices; ++j ) {
            if ( d[i][j] > d[i][k] + d[k][j] ) {
                p[i][j] = p[i][k];
                d[i][j] = d[i][k] + d[k][j];
            }
        }
    }
}
```

Example

In our original example, initially, the next node is exactly that:

$$\begin{pmatrix} - & 2 & 3 & 4 & 5 \\ 1 & - & 3 & 4 & 5 \\ 1 & 2 & - & 4 & 5 \\ 1 & 2 & 3 & - & 5 \\ 1 & 2 & 3 & 4 & - \end{pmatrix}$$

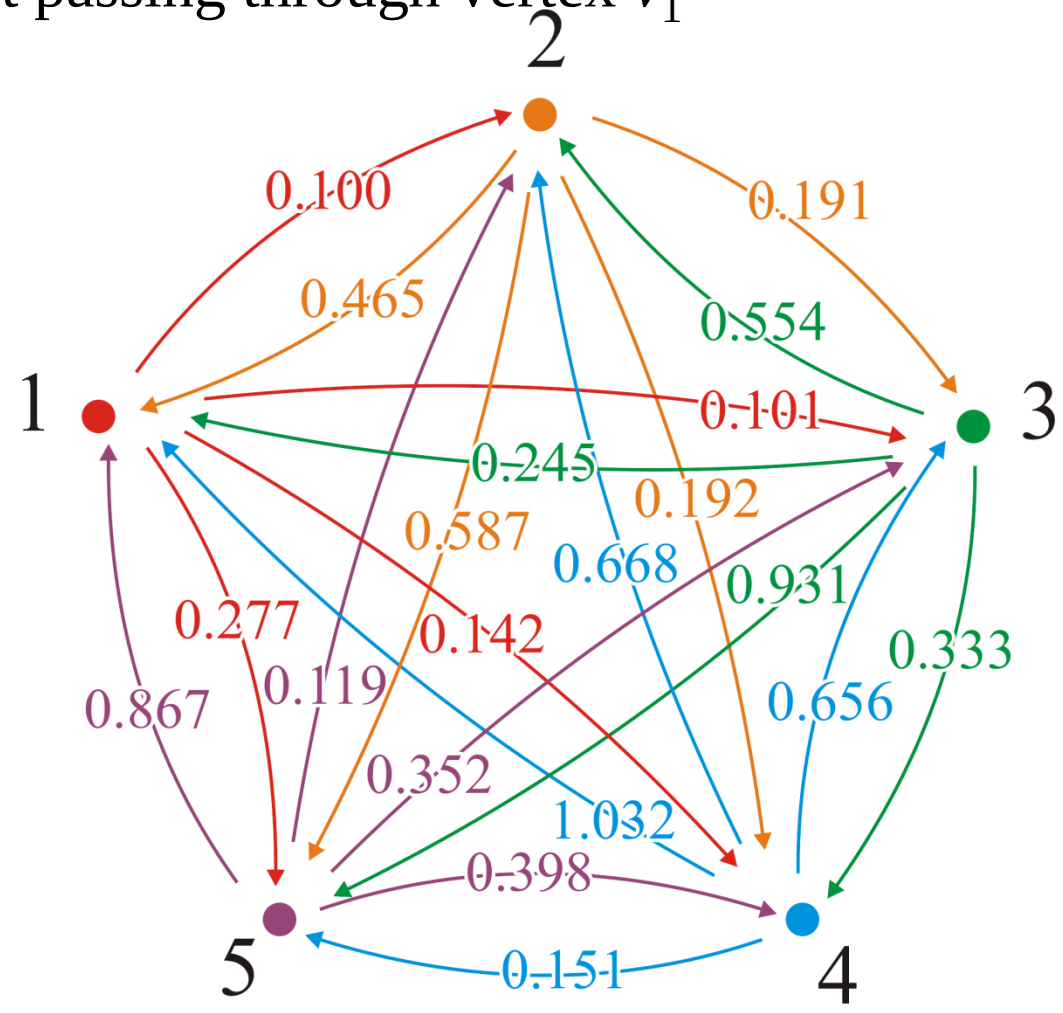


This would define our matrix $\mathbf{P} = (p_{ij})$

Example

With the first pass, $k = 1$, we attempt passing through vertex v_1

$$\begin{pmatrix} - & 2 & 3 & 4 & 5 \\ 1 & - & 3 & 4 & 5 \\ 1 & 2 & - & 4 & 5 \\ 1 & 2 & 3 & - & 5 \\ 1 & 2 & 3 & 4 & - \end{pmatrix}$$



There are two shorter paths:

$(3, 2) \rightarrow (3, 1, 2)$

$0.554 > 0.245 + 0.100$

$(3, 5) \rightarrow (3, 1, 5)$

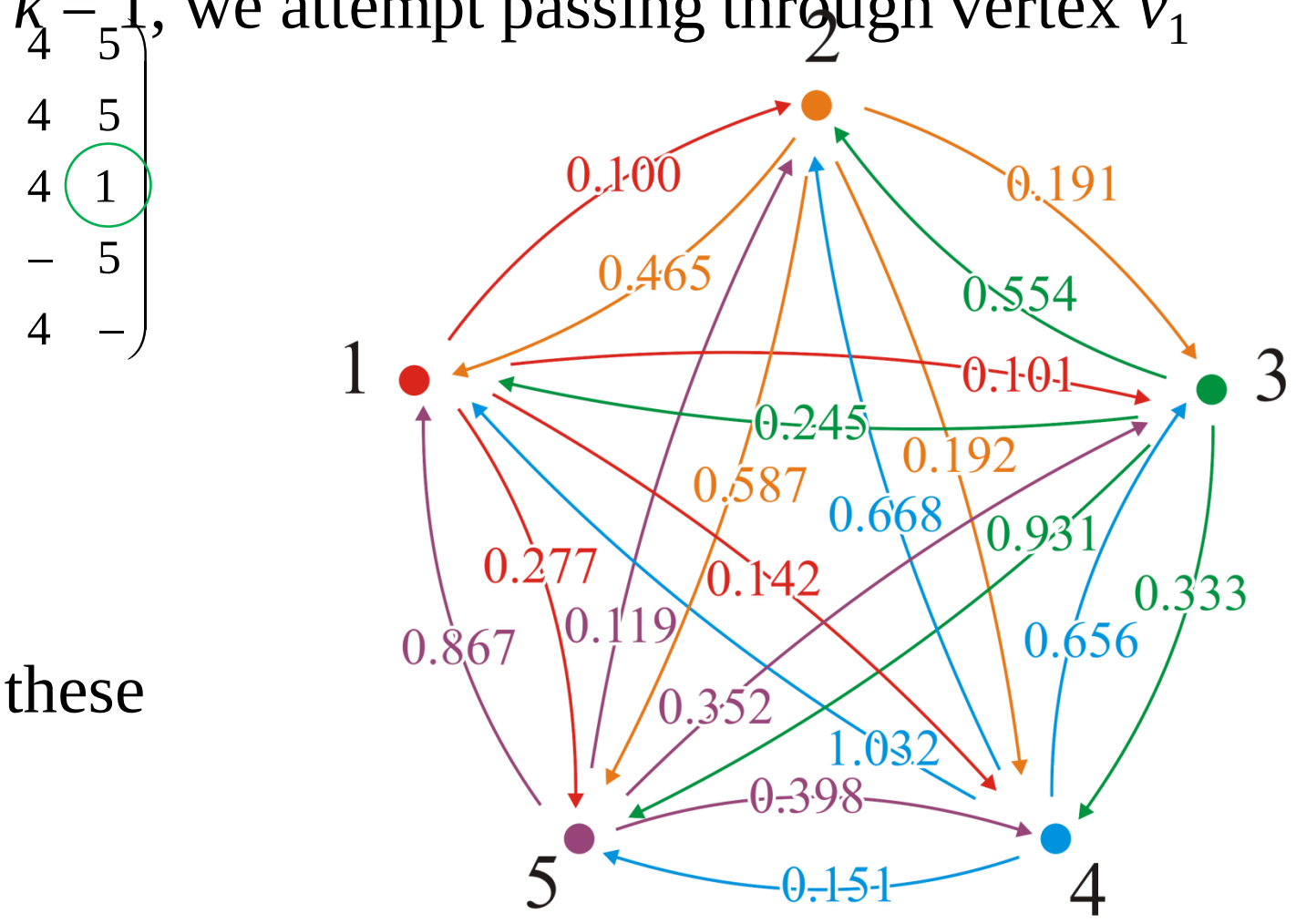
$0.931 > 0.245 + 0.277$

Example

With the first pass, $k = 1$, we attempt passing through vertex v_1

$$\begin{pmatrix} - & 2 & 3 & 4 & 5 \\ 1 & - & 3 & 4 & 5 \\ 1 & 1 & - & 4 & 1 \\ 1 & 2 & 3 & - & 5 \\ 1 & 2 & 3 & 4 & - \end{pmatrix}$$

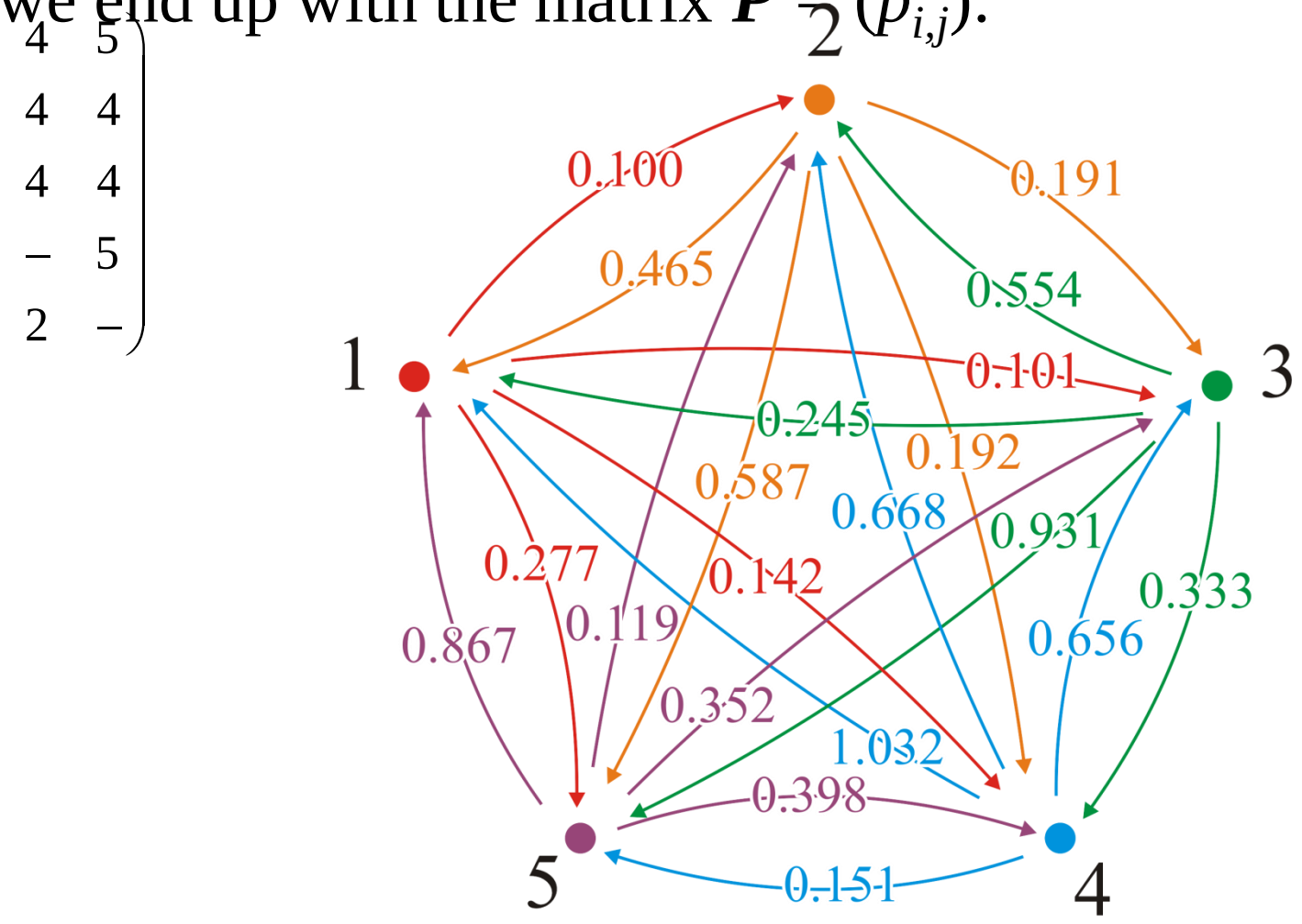
We update each of these



Example

After all the steps, we end up with the matrix $\boldsymbol{P} = (p_{i,j})$:

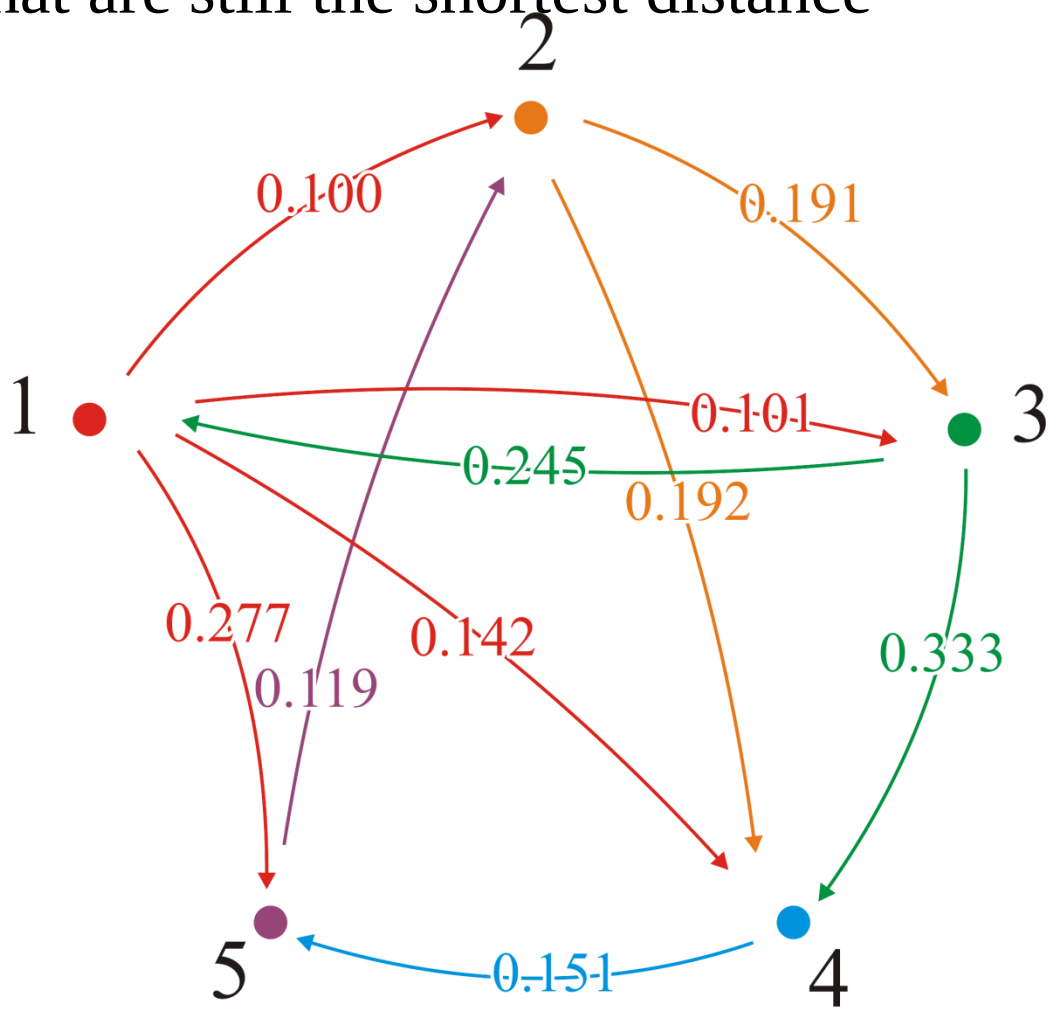
$$\begin{pmatrix} - & 2 & 3 & 4 & 5 \\ 3 & - & 3 & 4 & 4 \\ 1 & 1 & - & 4 & 4 \\ 5 & 5 & 5 & - & 5 \\ 2 & 2 & 2 & 2 & - \end{pmatrix}$$



Example

These are all the adjacent edges that are still the shortest distance

$$\begin{pmatrix} - & 2 & 3 & 4 & 5 \\ 3 & - & 3 & 4 & 4 \\ 1 & 1 & - & 4 & 4 \\ 5 & 5 & 5 & - & 5 \\ 2 & 2 & 2 & 2 & - \end{pmatrix}$$



For each of these, $p_{i,j} = j$

In all cases, the shortest distance from vertex 1 is the direct edge

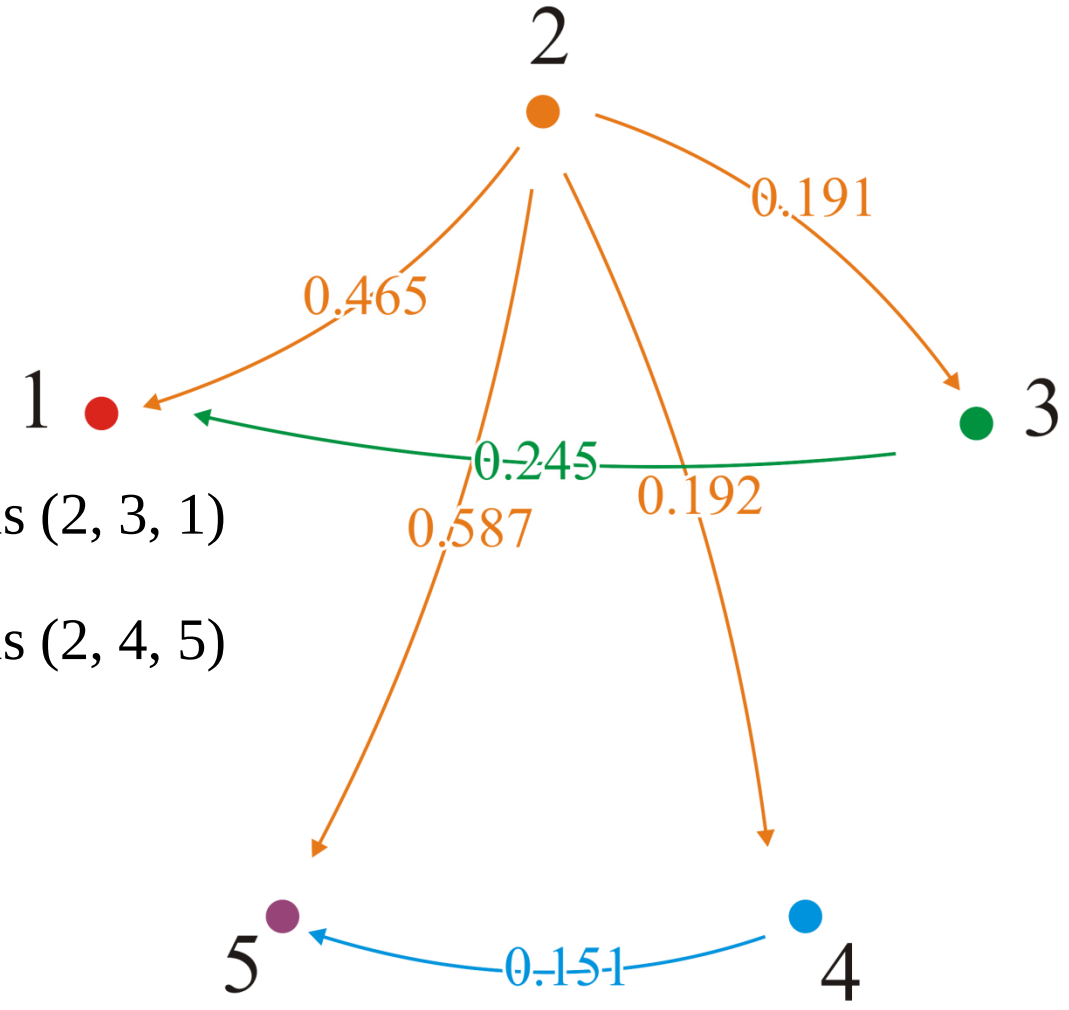
Example

From vertex v_2, $p_{2,3} = 3$ and $p_{2,4} = 4$; we go directly to vertices v_3 and v_4

$$\begin{pmatrix} - & 2 & 3 & 4 & 5 \\ 3 & - & 3 & 4 & 4 \\ 1 & 1 & - & 4 & 4 \\ 5 & 5 & 5 & - & 5 \\ 2 & 2 & 2 & 2 & - \end{pmatrix}$$

But $p_{2,1} = 3$ and $p_{3,1} = 1$;
the shortest path to v_1 is (2, 3, 1)

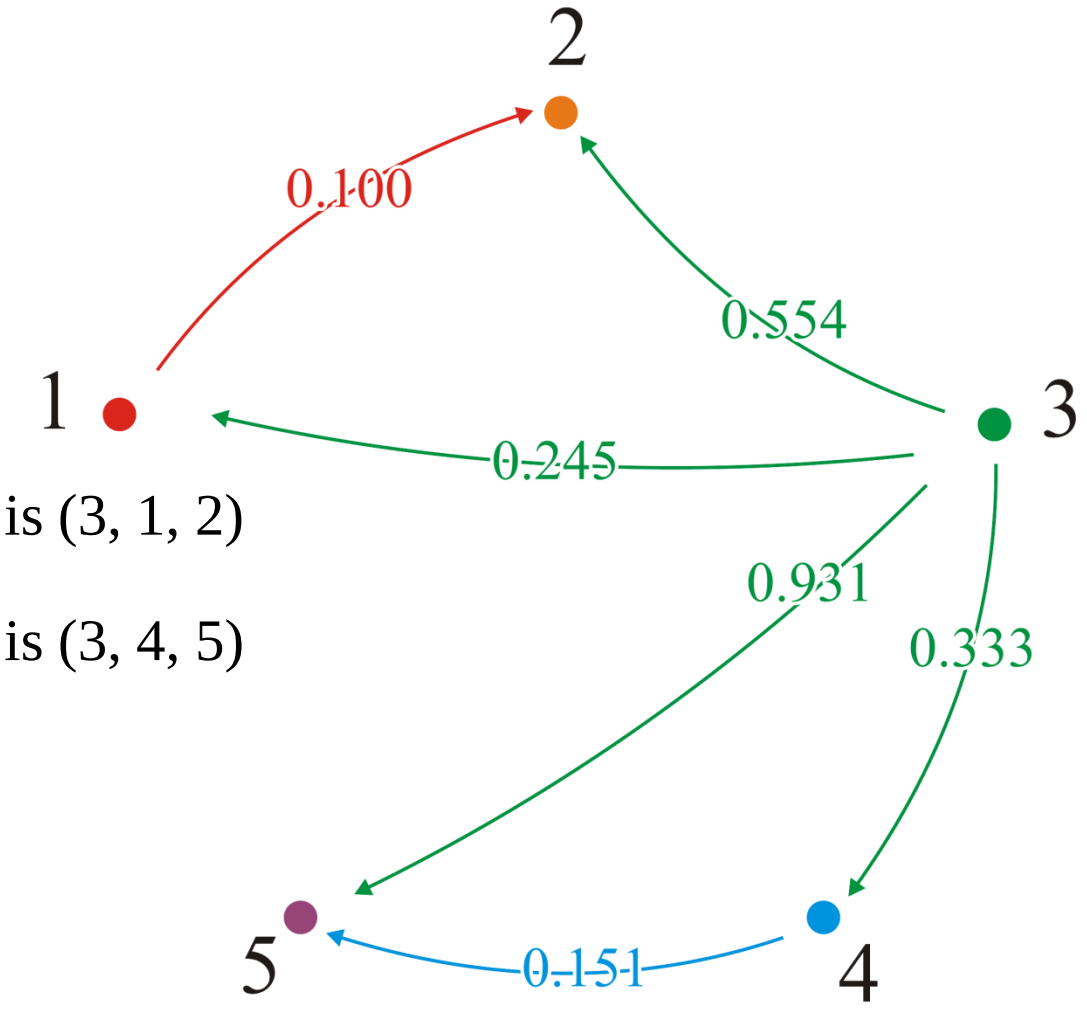
Also, $p_{2,5} = 4$ and $p_{4,5} = 5$;
the shortest path to v_5 is (2, 4, 5)



Example

From vertex v_3, $p_{3,1} = 1$ and $p_{3,4} = 4$; we go directly to vertices v_1 and v_4

$$\begin{pmatrix} - & 2 & 3 & 4 & 5 \\ 3 & - & 3 & 4 & 4 \\ 1 & 1 & - & 4 & 4 \\ 5 & 5 & 5 & - & 5 \\ 2 & 2 & 2 & 2 & - \end{pmatrix}$$



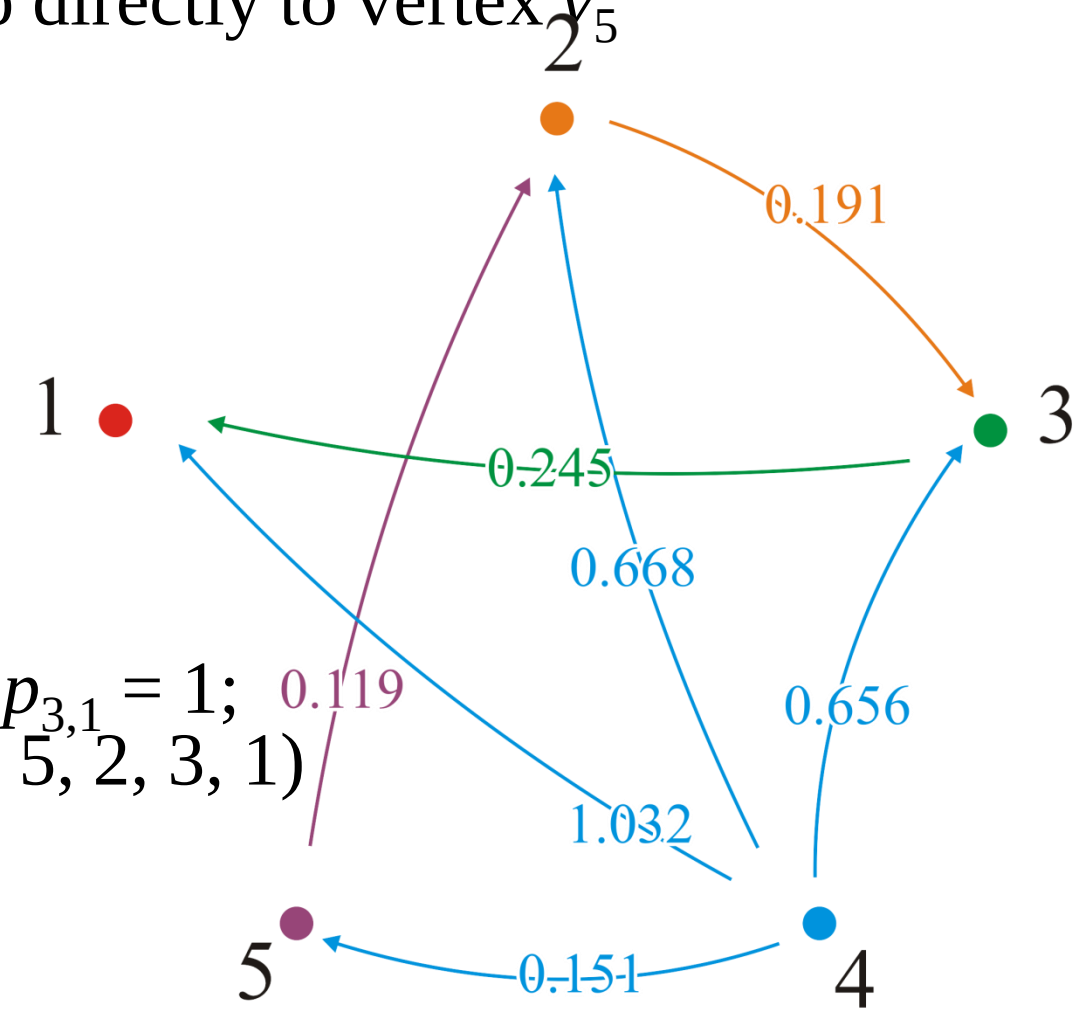
But $p_{3,2} = 1$ and $p_{1,2} = 2$;
the shortest path to v_2 is (3, 1, 2)

Also, $p_{3,5} = 4$ and $p_{4,5} = 5$;
the shortest path to v_5 is (3, 4, 5)

Example

From vertex v_4, $p_{4,5} = 5$; we go directly to vertex v_5

–	2	3	4	5
3	–	3	4	4
1	1	–	4	4
5	5	5	–	5
2	2	2	2	–

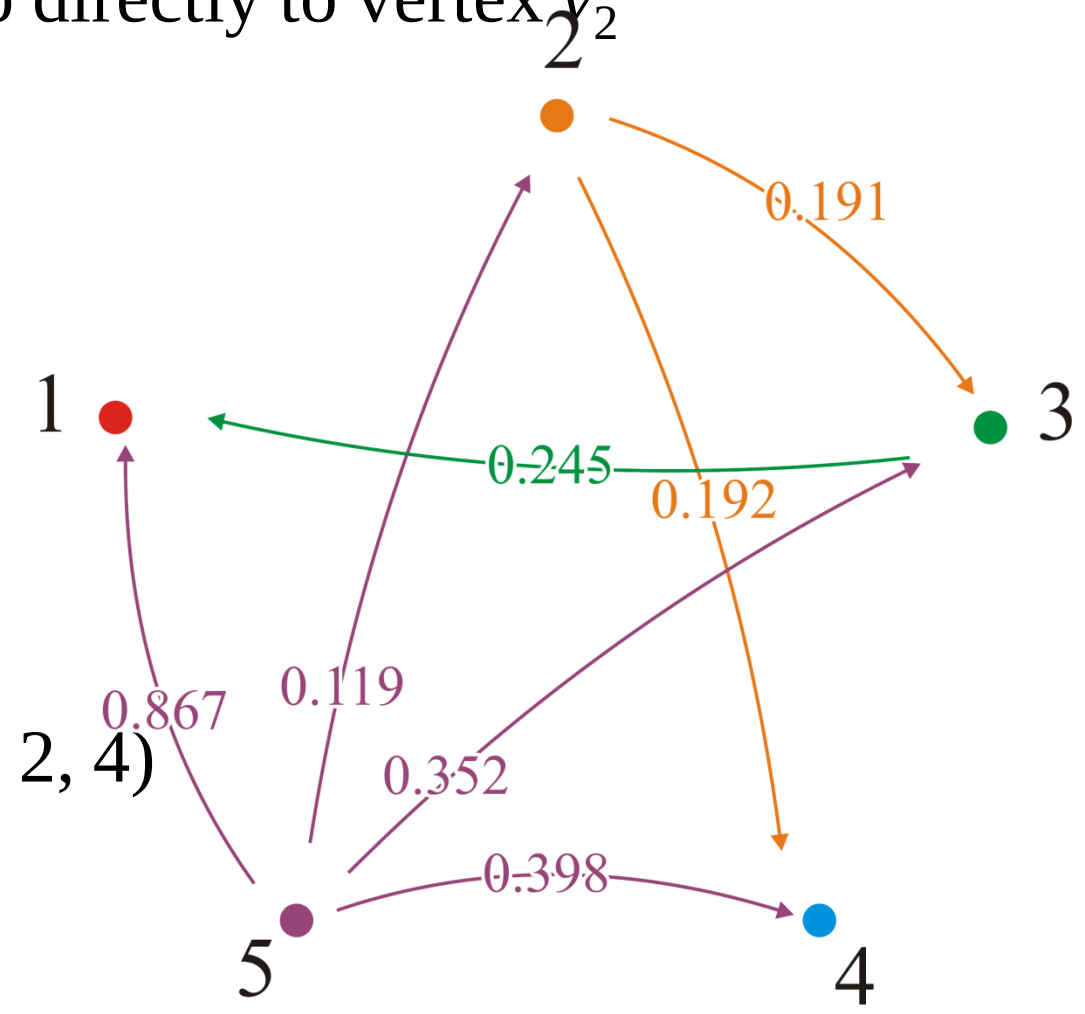


But $p_{4,1} = 5$, $p_{5,1} = 2$, $p_{2,1} = 3$, $p_{3,1} = 1$;
the shortest path to v_1 is (4, 5, 2, 3, 1)

Example

From vertex v_5, $p_{5,2} = 2$; we go directly to vertex v_2

–	2	3	4	5
3	–	3	4	4
1	1	–	4	4
5	5	5	–	5
2	2	2	2	–



But $p_{5,4} = 2$ and $p_{2,4} = 4$;
the shortest path to v_4 is (5, 2, 4)

Which Vertices are Connected?

Finally, what if we only care if a connection exists?

- Recall that with Dijkstra's algorithm, we could find the shortest paths by recording the previous node
- In this case, can make the observation that:

A path from v_i to v_j exists if either:

A path exists through the vertices from v_1 to v_{k-1}, or

A path, through those same nodes, exists from v_i to v_k and a path exists from v_k to v_j

Which Vertices are Connected?

The *transitive closure* is a Boolean graph:

```
bool tc[num_vertices][num_vertices];

// Initialize the matrix tc:  Theta(|V|^2)
//   ...

// Run Floyd-Warshall
for ( int k = 0; k < num_vertices; ++k ) {
    for ( int i = 0; i < num_vertices; ++i ) {
        for ( int j = 0; j < num_vertices; ++j ) {
            tc[i][j] = tc[i][j] || (tc[i][k] && tc[k][j]);
        }
    }
}
```

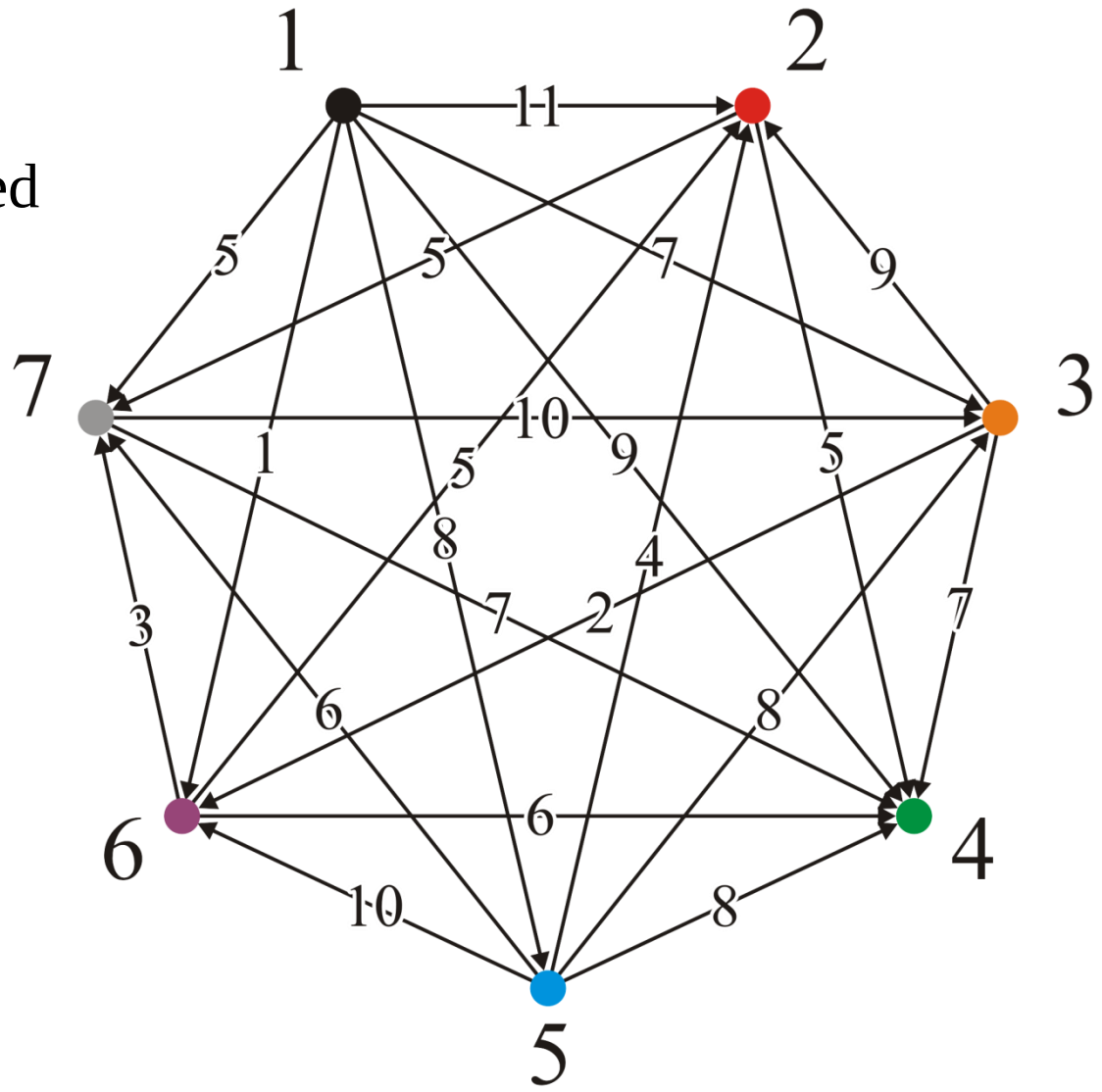
Example

Consider this directed graph

- Each pair has only one directed edge
- Vertex v_1 is a source and v_4 is a sink

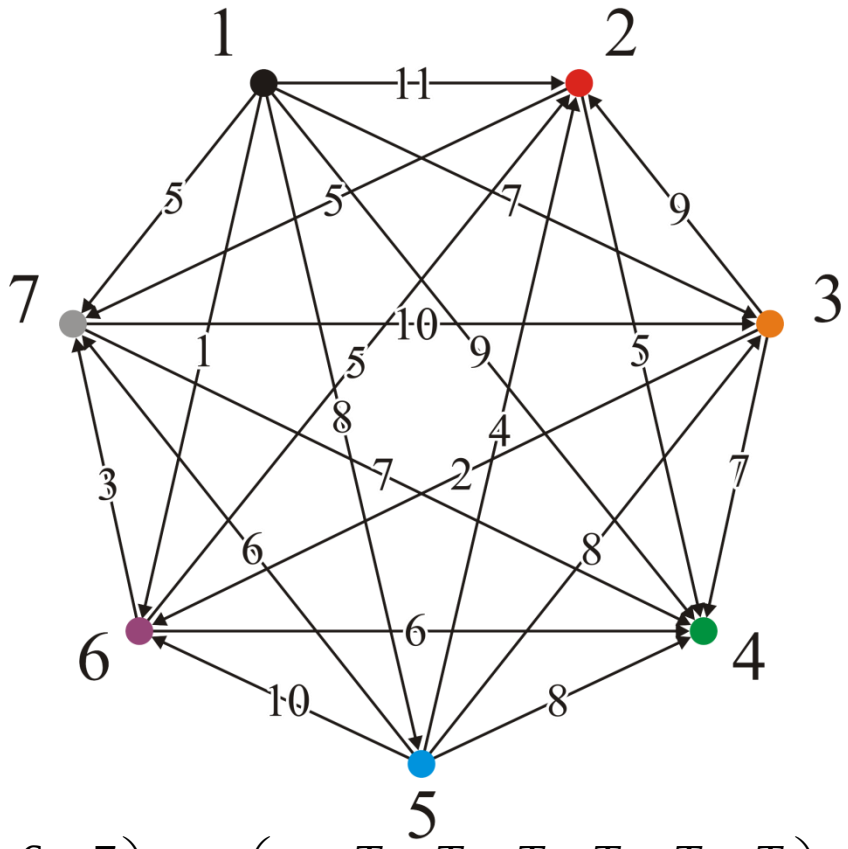
We will apply all three matrices

- Shortest distance
- Paths
- Transitive closure



Example

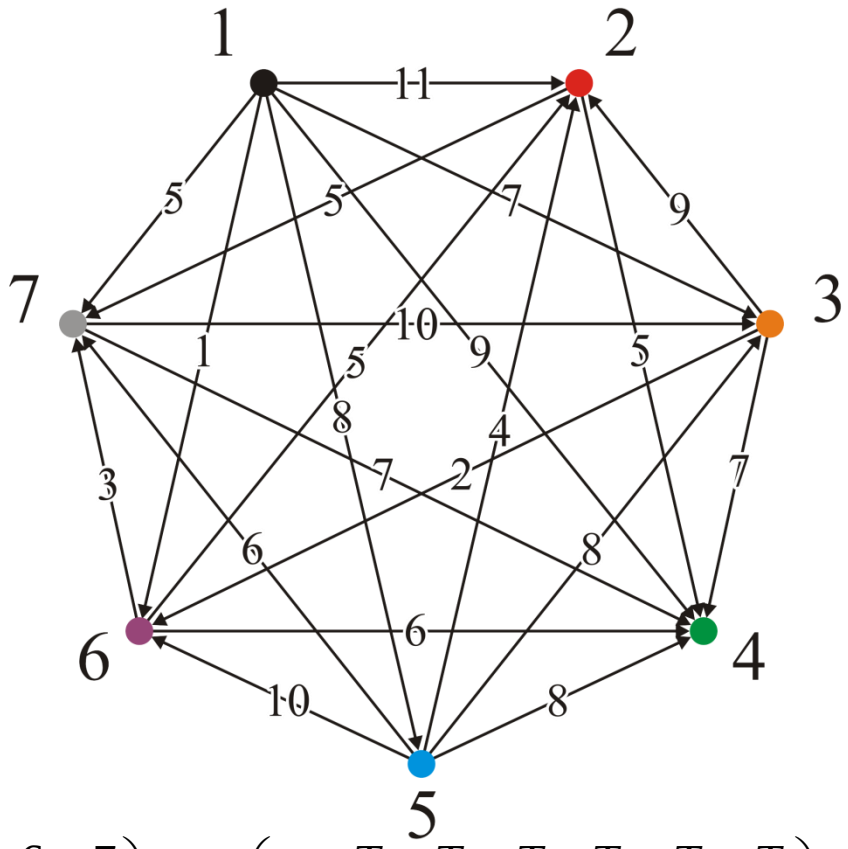
We set up the three initial matrices



$$\begin{pmatrix} 0 & 11 & 7 & 9 & 8 & 1 & 5 \\ \infty & 0 & \infty & 5 & \infty & \infty & 5 \\ \infty & 9 & 0 & 7 & \infty & 2 & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & \infty & 6 & \infty & 0 & 3 \\ \infty & \infty & 10 & 7 & \infty & \infty & 0 \end{pmatrix} \quad \begin{pmatrix} - & 2 & 3 & 4 & 5 & 6 & 7 \\ - & - & - & 4 & - & - & 7 \\ - & 2 & - & 4 & - & 6 & - \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & - & 4 & - & - & 7 \\ - & - & 3 & 4 & - & - & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & F & T & F & F & T \\ F & T & - & T & F & T & F \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & F & T & F & - & T \\ F & F & T & T & F & F & - \end{pmatrix}$$

Example

At step 1, no path leads to v_1, so no shorter paths could be found passing through v_1

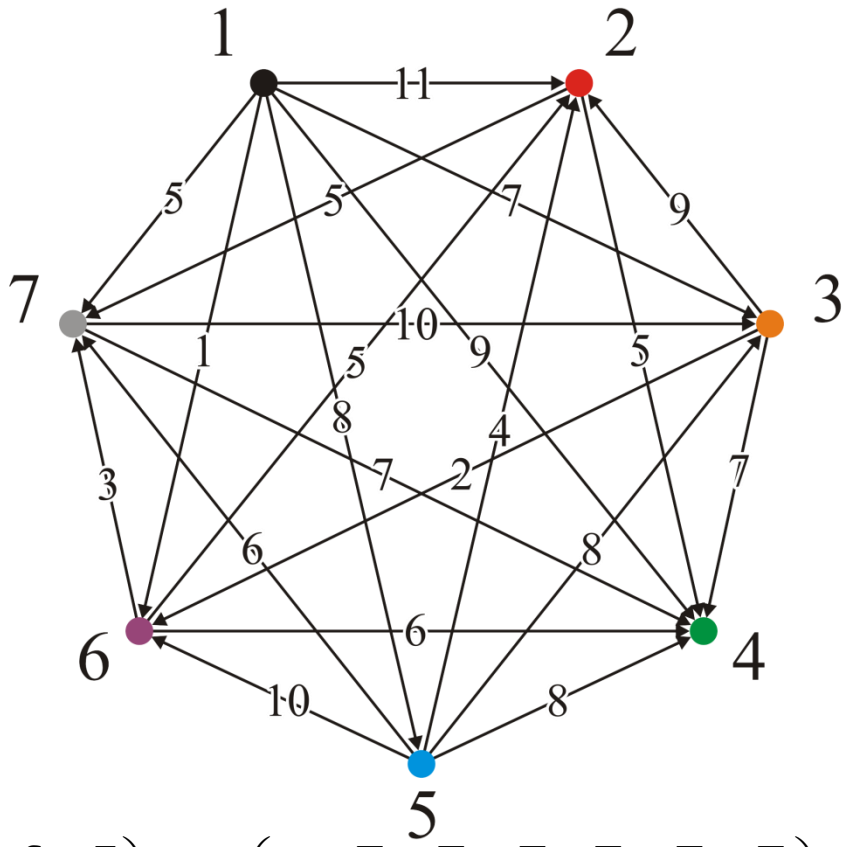


$$\begin{pmatrix} 0 & 11 & 7 & 9 & 8 & 1 & 5 \\ \infty & 0 & \infty & 5 & \infty & \infty & 5 \\ \infty & 9 & 0 & 7 & \infty & 2 & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & \infty & 6 & \infty & 0 & 3 \\ \infty & \infty & 10 & 7 & \infty & \infty & 0 \end{pmatrix} \quad \begin{pmatrix} - & 2 & 3 & 4 & 5 & 6 & 7 \\ - & - & - & 4 & - & - & 7 \\ - & 2 & - & 4 & - & 6 & - \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & - & 4 & - & - & 7 \\ - & - & 3 & 4 & - & - & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & F & T & F & F & T \\ F & T & - & T & F & T & F \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & F & T & F & - & T \\ F & F & T & T & F & F & - \end{pmatrix}$$

Example

At step 2, we find:

- A path (3, 2, 7) of length 14



$$\begin{pmatrix} 0 & 11 & 7 & 9 & 8 & 1 & 5 \\ \infty & 0 & \infty & 5 & \infty & \infty & 5 \\ \infty & 9 & 0 & 7 & \infty & 2 & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & \infty & 6 & \infty & 0 & 3 \\ \infty & \infty & 10 & 7 & \infty & \infty & 0 \end{pmatrix} \quad \begin{pmatrix} - & 2 & 3 & 4 & 5 & 6 & 7 \\ - & - & - & 4 & - & - & 7 \\ - & 2 & - & 4 & - & 6 & - \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & - & 4 & - & - & 7 \\ - & - & 3 & 4 & - & - & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & F & T & F & F & T \\ F & T & - & T & F & T & F \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & F & T & F & - & T \\ F & F & T & T & F & F & - \end{pmatrix}$$

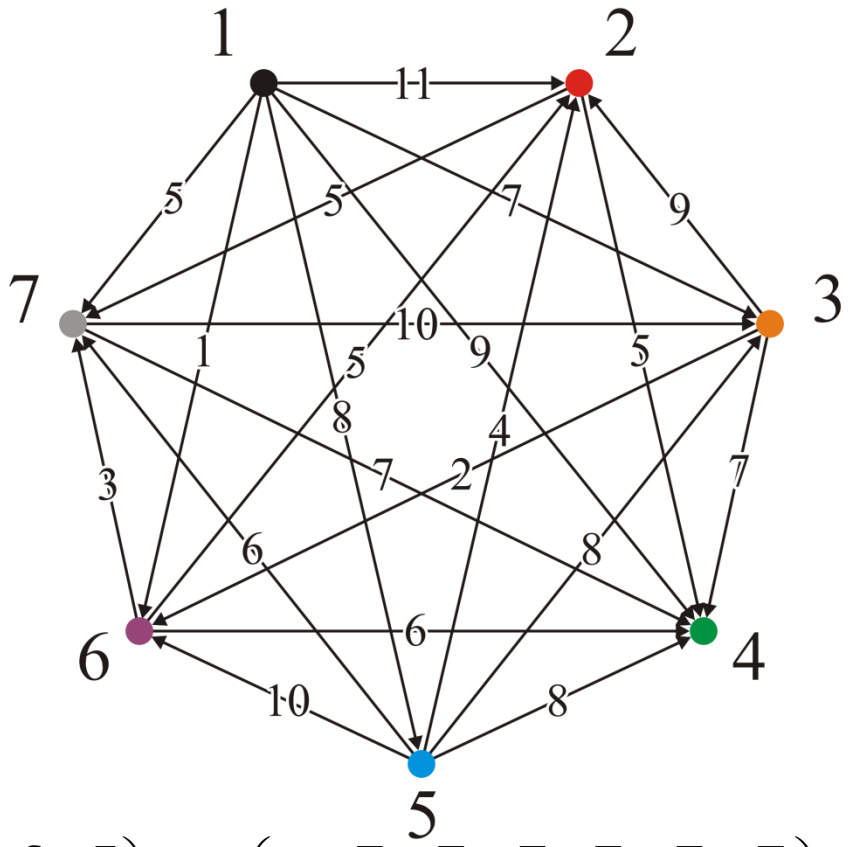
Example

At step 2, we find:

- A path (3, 2, 7) of length 14

We update

$$d_{3,7} = 14,\ p_{3,7} = 2 \text{ and } c_{3,7} = T$$

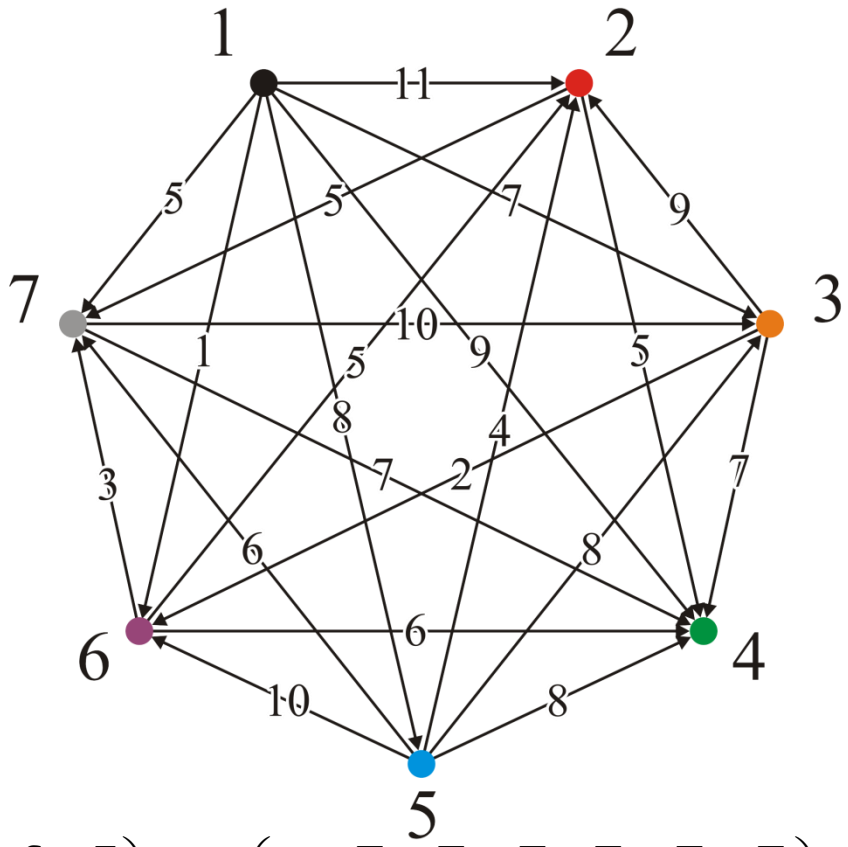


$$\begin{pmatrix} 0 & 11 & 7 & 9 & 8 & 1 & 5 \\ \infty & 0 & \infty & 5 & \infty & \infty & 5 \\ \infty & 9 & 0 & 7 & \infty & 2 & 14 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & \infty & 6 & \infty & 0 & 3 \\ \infty & \infty & 10 & 7 & \infty & \infty & 0 \end{pmatrix} \quad \begin{pmatrix} - & 2 & 3 & 4 & 5 & 6 & 7 \\ - & - & - & 4 & - & - & 7 \\ - & 2 & - & 4 & - & 6 & 2 \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & - & 4 & - & - & 7 \\ - & - & 3 & 4 & - & - & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & F & T & F & F & T \\ F & T & - & T & F & T & T \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & F & T & F & - & T \\ F & F & T & T & F & F & - \end{pmatrix}$$

Example

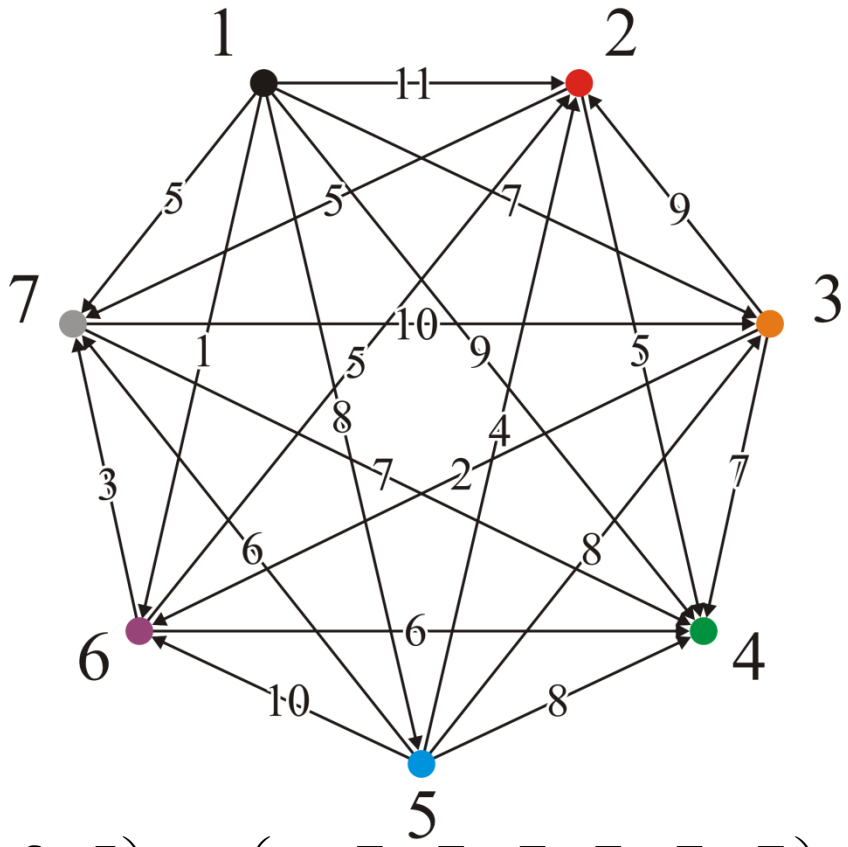
At step 3, we find:

- A path (7, 3, 2) of length 19
- A path (7, 3, 6) of length 12



$$\begin{pmatrix} 0 & 11 & 7 & 9 & 8 & 1 & 5 \\ \infty & 0 & \infty & 5 & \infty & \infty & 5 \\ \infty & 9 & 0 & 7 & \infty & 2 & 14 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & \infty & 6 & \infty & 0 & 3 \\ \infty & \infty & 10 & 7 & \infty & \infty & 0 \end{pmatrix} \quad \begin{pmatrix} - & 2 & 3 & 4 & 5 & 6 & 7 \\ - & - & - & 4 & - & - & 7 \\ - & 2 & - & 4 & - & 6 & 2 \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & - & 4 & - & - & 7 \\ - & - & 3 & 4 & - & - & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & F & T & F & F & T \\ F & T & - & T & F & T & T \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & F & T & F & - & T \\ F & F & T & T & F & F & - \end{pmatrix}$$

Example



At step 3, we find:

- A path (7, 3, 2) of length 19
- A path (7, 3, 6) of length 12

We update

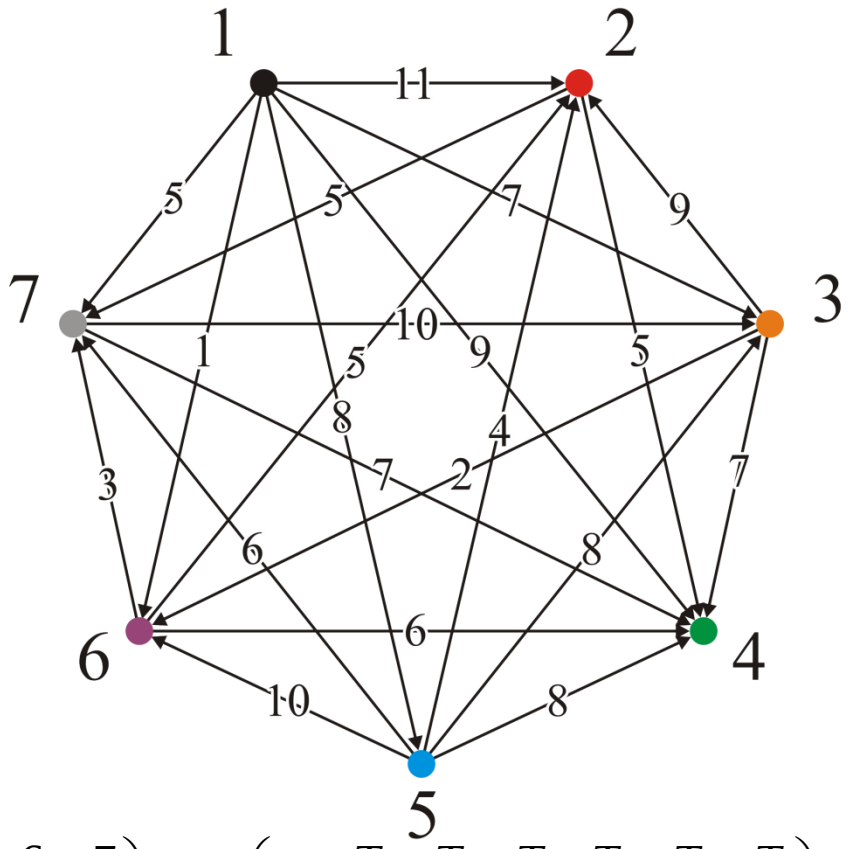
$d_{7,2} = 19$, $p_{7,2} = 3$ and $c_{7,2} = T$

$d_{7,6} = 12$, $p_{7,6} = 3$ and $c_{7,6} = T$

$$\begin{pmatrix} 0 & 11 & 7 & 9 & 8 & 1 & 5 \\ \infty & 0 & \infty & 5 & \infty & \infty & 5 \\ \infty & 9 & 0 & 7 & \infty & 2 & 14 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & \infty & 6 & \infty & 0 & 3 \\ \infty & 19 & 10 & 7 & \infty & 12 & 0 \end{pmatrix} \quad \begin{pmatrix} - & 2 & 3 & 4 & 5 & 6 & 7 \\ - & - & - & 4 & - & - & 7 \\ - & 2 & - & 4 & - & 6 & 2 \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & - & 4 & - & - & 7 \\ - & 3 & 3 & 4 & - & 3 & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & F & T & F & F & T \\ F & T & - & T & F & T & T \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & F & T & F & - & T \\ F & T & T & T & F & T & - \end{pmatrix}$$

Example

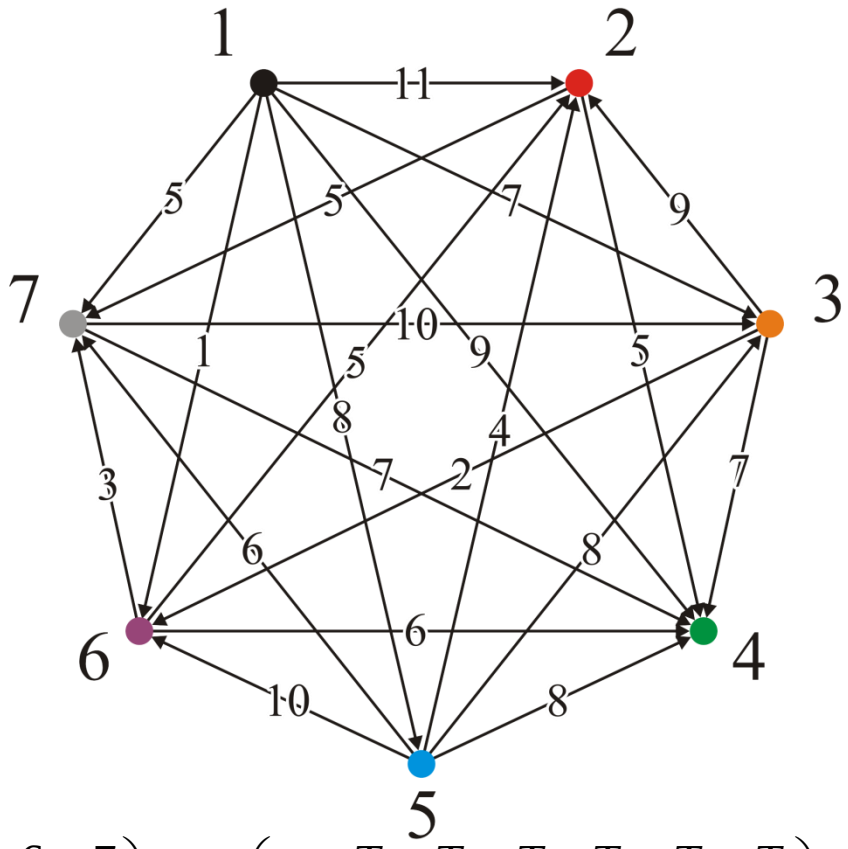
At step 4, there are no paths out of vertex v_4, so we are finished



$$\begin{pmatrix} 0 & 11 & 7 & 9 & 8 & 1 & 5 \\ \infty & 0 & \infty & 5 & \infty & \infty & 5 \\ \infty & 9 & 0 & 7 & \infty & 2 & 14 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & \infty & 6 & \infty & 0 & 3 \\ \infty & 19 & 10 & 7 & \infty & 12 & 0 \end{pmatrix} \quad \begin{pmatrix} - & 2 & 3 & 4 & 5 & 6 & 7 \\ - & - & - & 4 & - & - & 7 \\ - & 2 & - & 4 & - & 6 & 2 \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & - & 4 & - & - & 7 \\ - & 3 & 3 & 4 & - & 3 & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & F & T & F & F & T \\ F & T & - & T & F & T & T \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & F & T & F & - & T \\ F & T & T & T & F & T & - \end{pmatrix}$$

Example

At step 5, there is one incoming edge from v_1 to v_5, and it doesn't make any paths out of vertex v_1 any shorter...

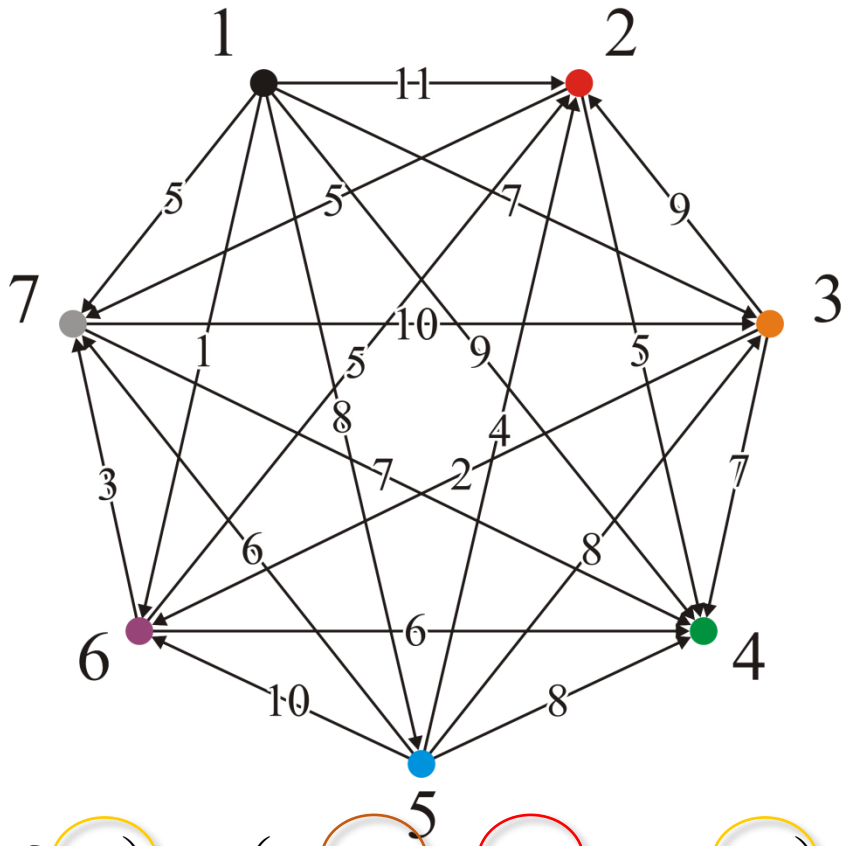


$$\begin{pmatrix} 0 & 11 & 7 & 9 & 8 & 1 & 5 \\ \infty & 0 & \infty & 5 & \infty & \infty & 5 \\ \infty & 9 & 0 & 7 & \infty & 2 & 14 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & \infty & 6 & \infty & 0 & 3 \\ \infty & 19 & 10 & 7 & \infty & 12 & 0 \end{pmatrix} \quad \begin{pmatrix} - & 2 & 3 & 4 & 5 & 6 & 7 \\ - & - & - & 4 & - & - & 7 \\ - & 2 & - & 4 & - & 6 & 2 \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & - & 4 & - & - & 7 \\ - & 3 & 3 & 4 & - & 3 & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & F & T & F & F & T \\ F & T & - & T & F & T & T \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & F & T & F & - & T \\ F & T & T & T & F & T & - \end{pmatrix}$$

Example

At step 6, we find:

- A path (1, 6, 2) of length 6
- A path (1, 6, 4) of length 7
- A path (1, 6, 7) of length 4
- A path (3, 6, 2) of length 7
- A path (3, 6, 5) of length 5
- A path (7, 3, 6, 2) of length 17

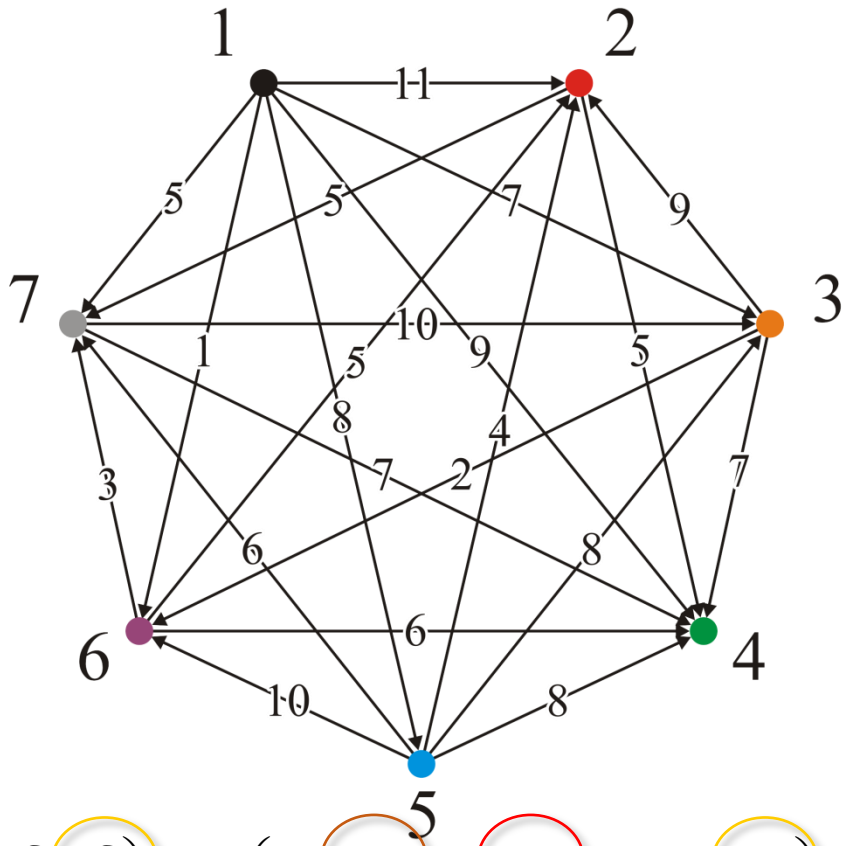


$$\begin{pmatrix} 0 & 11 & 7 & 9 & 8 & 1 & 5 \\ \infty & 0 & \infty & 5 & \infty & \infty & 5 \\ \infty & 9 & 0 & 7 & \infty & 2 & 14 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & \infty & 6 & \infty & 0 & 3 \\ \infty & 19 & 10 & 7 & \infty & 12 & 0 \end{pmatrix} \quad \begin{pmatrix} - & 2 & 3 & 4 & 5 & 6 & 7 \\ - & - & - & 4 & - & - & 7 \\ - & 2 & - & 4 & - & 6 & 2 \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & - & 4 & - & - & 7 \\ - & 3 & 3 & 4 & - & 3 & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & F & T & F & F & T \\ F & T & - & T & F & T & T \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & F & T & F & - & T \\ F & T & T & T & F & T & - \end{pmatrix}$$

Example

At step 6, we find:

- A path (1, 6, 2) of length 6
- A path (1, 6, 4) of length 7
- A path (1, 6, 7) of length 4
- A path (3, 6, 2) of length 7
- A path (3, 6, 5) of length 5
- A path (7, 3, 6, 2) of length 17

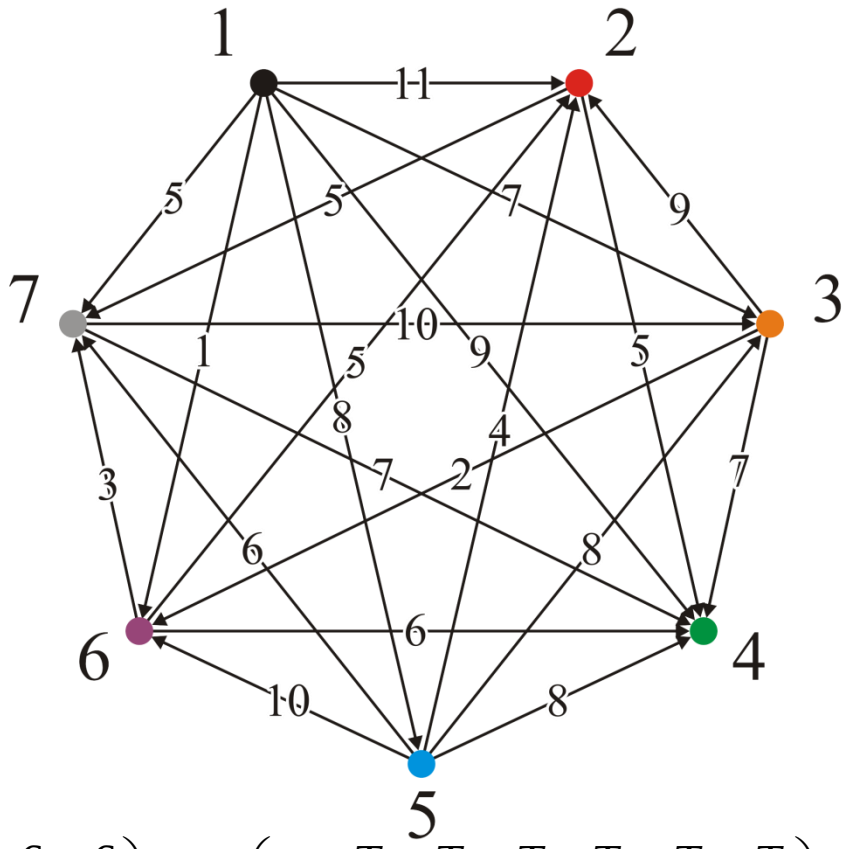


$$\begin{pmatrix} 0 & 6 & 7 & 7 & 8 & 1 & 4 \\ \infty & 0 & \infty & 5 & \infty & \infty & 5 \\ \infty & 7 & 0 & 7 & \infty & 2 & 5 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & \infty & 6 & \infty & 0 & 3 \\ \infty & 17 & 10 & 7 & \infty & 12 & 0 \end{pmatrix} \quad \begin{pmatrix} - & 6 & 3 & 6 & 5 & 6 & 6 \\ - & - & - & 4 & - & - & 7 \\ - & 6 & - & 4 & - & 6 & 6 \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & - & 4 & - & - & 7 \\ - & 3 & 3 & 4 & - & 3 & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & F & T & F & F & T \\ F & T & - & T & F & T & T \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & F & T & F & - & T \\ F & T & T & T & F & T & - \end{pmatrix}$$

Example

At step 7, we find:

- A path (2, 7, 3) of length 15
- A path (2, 7, 6) of length 17
- A path (6, 7, 3) of length 13



$$\begin{pmatrix} 0 & 6 & 7 & 7 & 8 & 1 & 4 \\ \infty & 0 & \infty & 5 & \infty & \infty & 5 \\ \infty & 7 & 0 & 7 & \infty & 2 & 5 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & \infty & 6 & \infty & 0 & 3 \\ \infty & 17 & 10 & 7 & \infty & 12 & 0 \end{pmatrix}$$

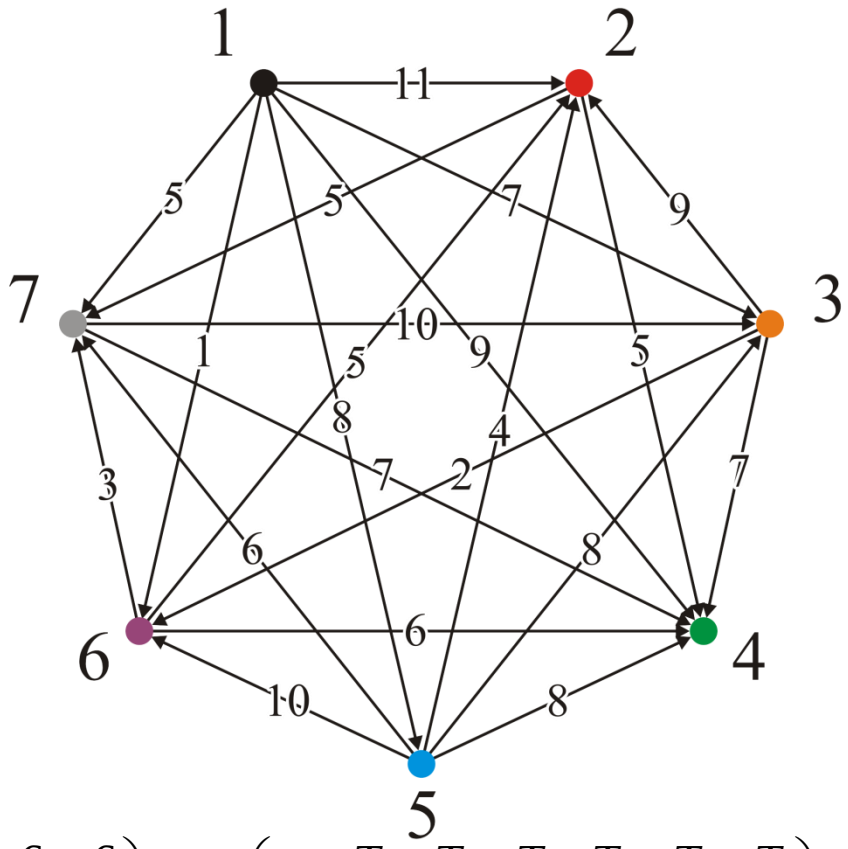
$$\begin{pmatrix} - & 6 & 3 & 6 & 5 & 6 & 6 \\ - & - & - & 4 & - & - & 7 \\ - & 6 & - & 4 & - & 6 & 6 \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & - & 4 & - & - & 7 \\ - & 3 & 3 & 4 & - & 3 & - \end{pmatrix}$$

$$\begin{pmatrix} - & T & T & T & T & T & T \\ F & - & F & T & F & F & T \\ F & T & - & T & F & T & T \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & F & T & F & - & T \\ F & T & T & T & F & T & - \end{pmatrix}$$

Example

Finally, at step 7, we find:

- A path (2, 7, 3) of length 15
- A path (2, 7, 6) of length 17
- A path (6, 7, 3) of length 13

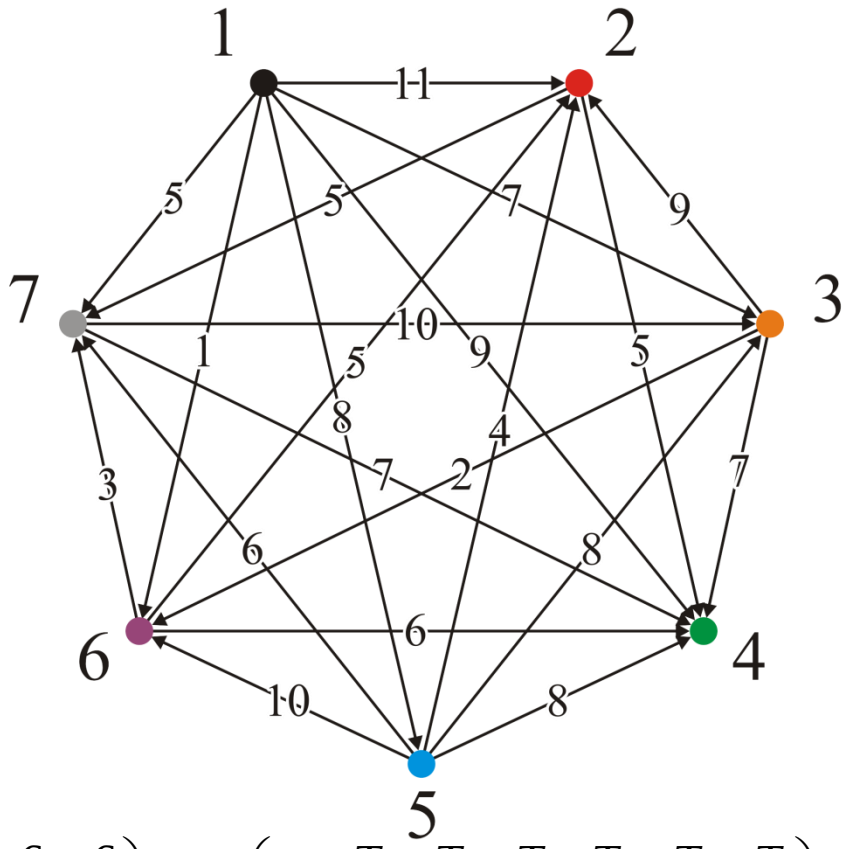


$$\begin{pmatrix} 0 & 6 & 7 & 7 & 8 & 1 & 4 \\ \infty & 0 & 15 & 5 & \infty & 17 & 5 \\ \infty & 7 & 0 & 7 & \infty & 2 & 5 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & 13 & 6 & \infty & 0 & 3 \\ \infty & 17 & 10 & 7 & \infty & 12 & 0 \end{pmatrix} \quad \begin{pmatrix} - & 6 & 3 & 6 & 5 & 6 & 6 \\ - & - & 7 & 4 & - & 7 & 7 \\ - & 6 & - & 4 & - & 6 & 6 \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & 7 & 4 & - & - & 7 \\ - & 3 & 3 & 4 & - & 3 & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & T & T & F & T & T \\ F & T & - & T & F & T & T \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & T & T & F & - & T \\ F & T & T & T & F & T & - \end{pmatrix}$$

Example

Note that:

- From v_1 we can go anywhere
- From v_5 we can go anywhere but v_1
- We go between any of the vertices in the set $\{v_2, v_3, v_6, v_7\}$
- We can't go anywhere from v_4



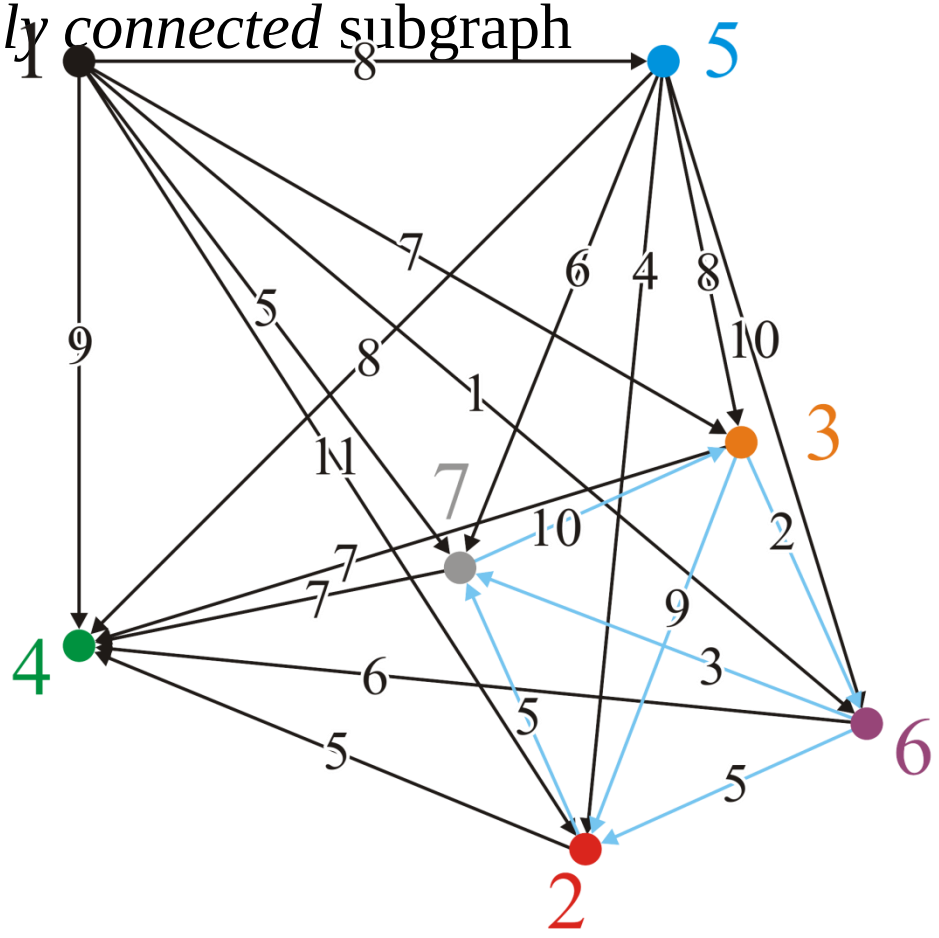
$$\begin{pmatrix} 0 & 6 & 7 & 7 & 8 & 1 & 4 \\ \infty & 0 & 15 & 5 & \infty & 17 & 5 \\ \infty & 7 & 0 & 7 & \infty & 2 & 5 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & 13 & 6 & \infty & 0 & 3 \\ \infty & 17 & 10 & 7 & \infty & 12 & 0 \end{pmatrix} \quad \begin{pmatrix} - & 6 & 3 & 6 & 5 & 6 & 6 \\ - & - & 7 & 4 & - & 7 & 7 \\ - & 6 & - & 4 & - & 6 & 6 \\ - & - & - & - & - & - & - \\ - & 2 & 3 & 4 & - & 6 & 7 \\ - & 2 & 7 & 4 & - & - & 7 \\ - & 3 & 3 & 4 & - & 3 & - \end{pmatrix} \quad \begin{pmatrix} - & T & T & T & T & T & T \\ F & - & T & T & F & T & T \\ F & T & - & T & F & T & T \\ F & F & F & - & F & F & F \\ F & T & T & T & - & T & T \\ F & T & T & T & F & - & T \\ F & T & T & T & F & T & - \end{pmatrix}$$

Example

We could reinterpret this graph as follows:

- Vertices $\{v_2, v_3, v_6, v_7\}$ form a *strongly connected* subgraph
- You can get from any one vertex to any other
- With the transitive closure graph, it is much faster finding such strongly connected components

$$\begin{pmatrix} 0 & 6 & 7 & 7 & 8 & 1 & 4 \\ \infty & 0 & 15 & 5 & \infty & 17 & 5 \\ \infty & 7 & 0 & 7 & \infty & 2 & 5 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 4 & 8 & 8 & 0 & 10 & 6 \\ \infty & 5 & 13 & 6 & \infty & 0 & 3 \\ \infty & 17 & 10 & 7 & \infty & 12 & 0 \end{pmatrix}$$



Summary

- Dijkstra’s algorithm
  - Single source shortest distance (non-negative weights)
- Bellman-Ford algorithm
  - For graphs with negative weights (but no cycles with negative weight)
- Floyd-Warshall algorithm
  - All-pairs shortest distance

Shortest Path – A*

Outline

In this topic, we will look at the A* search algorithm:

- It solves the single-source shortest path problem
- Restricted to *physical* environments
- First described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael
- Similar to Dijkstra’s algorithm
- Uses a hypothetical shortest distance to weight the paths

Background

Assume we have a heuristic lower bound for the length of a path between any two vertices

E.g., a graph embedded in a plane

- the shortest distance is the Euclidean distance

Algorithm Description

Suppose we are finding the shortest path from vertex a to a vertex z

The A* search algorithm initially:

- Marks each vertex as unvisited
- Starts with a priority queue containing only the initial vertex
  - The priority of any vertex v in the queue is the weight $w(v)$ which assumes we have found the a shortest path to v
  - Shortest weights have highest priority

Algorithm Description

The algorithm then iterates:

- Pops the vertex u with highest priority
  - Mark the vertex u of the path as visited
- For each remaining unenqueued adjacent vertex v:
  - Push v with $\mathrm{w}(v) = \mathrm{d}(a, u) + \mathrm{d}(u, v) + \mathrm{h}(v, z)$
- For each enqueued adjacent vertex v:
  - Determine if $\mathrm{w}(v) = \mathrm{d}(a, u) + \mathrm{d}(u, v) + \mathrm{h}(v, z)$ is less than the current weight/priority of v, and if so, update the path leading to v and its priority

Continue iterating until the item popped from the priority queue is the destination vertex z

Comparison with Dijkstra’s Algorithm

This differs from Dijkstra’s algorithm which gives weight only to the known path

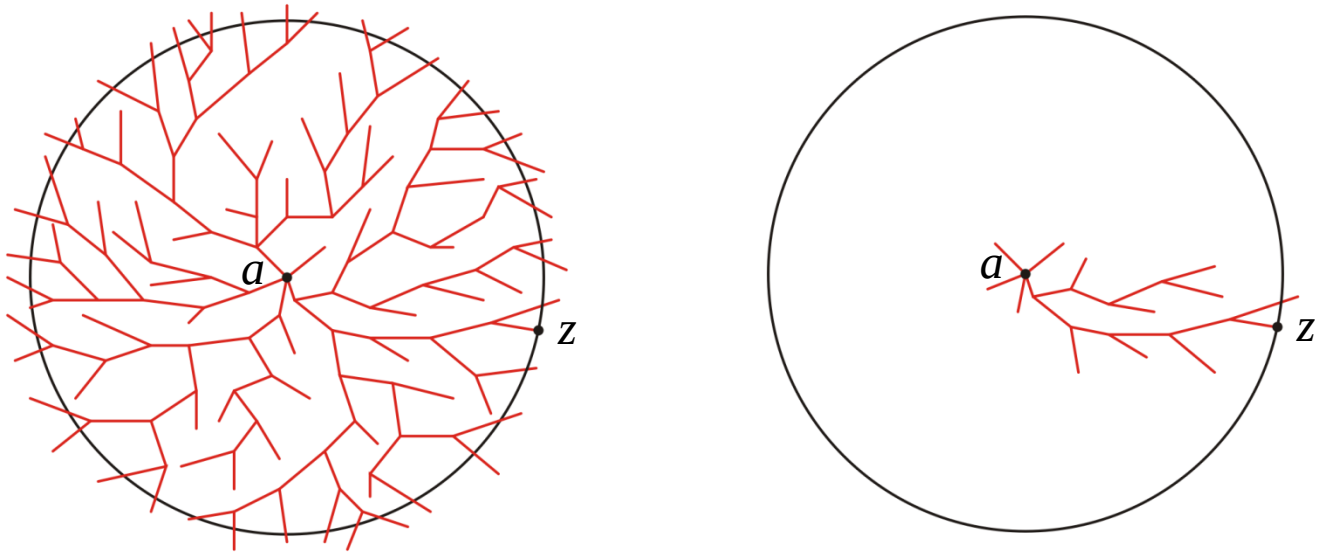
Difference:

- Dijkstra’s algorithm radiates out from the initial vertex
- The A* search algorithm directs its search towards the destination

Comparison with Dijkstra's Algorithm

Graphically, we can suggest the behaviour of the two algorithms as follows:

- Suppose we are moving from a to z:



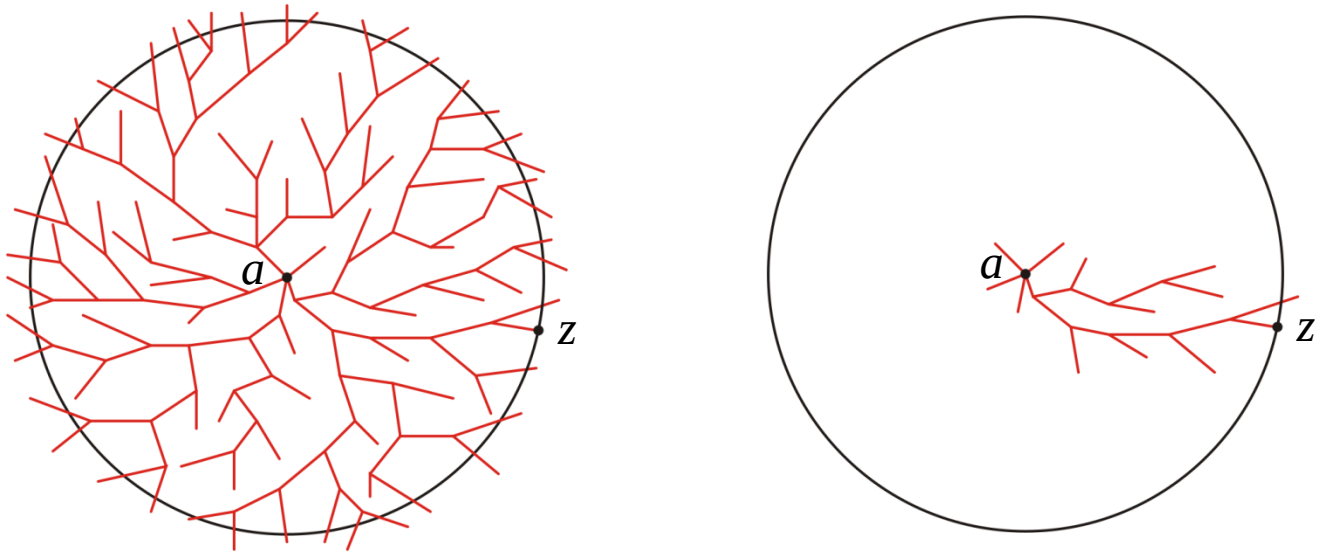
Representative search patterns for Dijkstra's and the A* search algorithms

Comparison with Dijkstra's Algorithm

Dijkstra's algorithm is the A* search algorithm when using the discrete distance

- No vertex is better than any other vertex

$$h(u,v) = \begin{cases} 0 & u = v \\ 1 & u \neq v \end{cases}$$

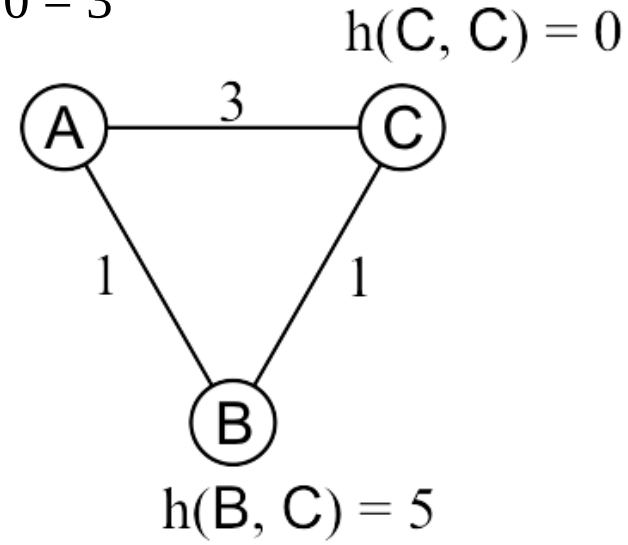


Representative search patterns for Dijkstra's and the A* search algorithms

Optimally Guarantees?

The A* search algorithm will **not** always find the optimal path with a poor heuristic distance

- Find the shortest path from A to C:
  - B is enqueued with weight w(B) = 1 + 5 = 6
  - C is enqueued with weight w(C) = 3 + 0 = 3
- Therefore, C is dequeued next and as it is the destination, we are finished

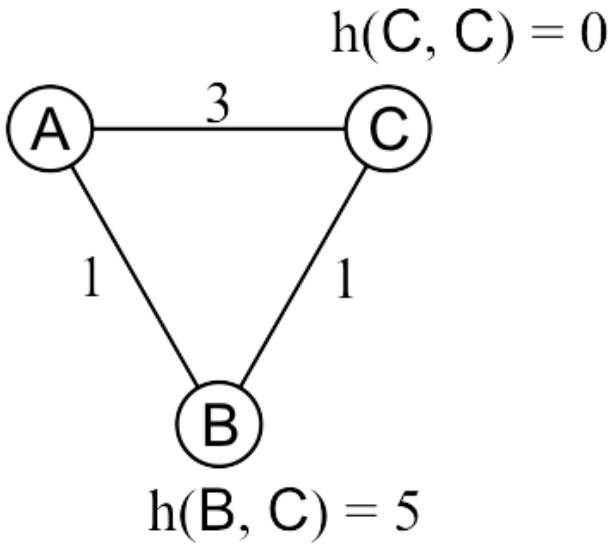


Reference: Andrew Moore, Carnegie Mellon University

Admissible Heuristics

This heuristic overestimates the actual distance from B to C

- The Euclidean distance doesn't suffer this problem:



Reference: Andrew Moore, Carnegie Mellon University

Admissible Heuristics

Admissible heuristics h must always be optimistic:

- Let $d(u, v)$ represent the actual shortest distance from u to v
- A heuristic $h(u, v)$ is *admissible* if $h(u, v) \leq d(u, v)$
- The heuristic is *optimistic* or a *lower bound* on the distance

Using the Euclidean distance between two points on a map is clearly an admissible heuristic

- The flight of the crow is shorter than the run of the wolf

Theorem: If *h(n)* is admissible, A* using `TREE-SEARCH` is optimal

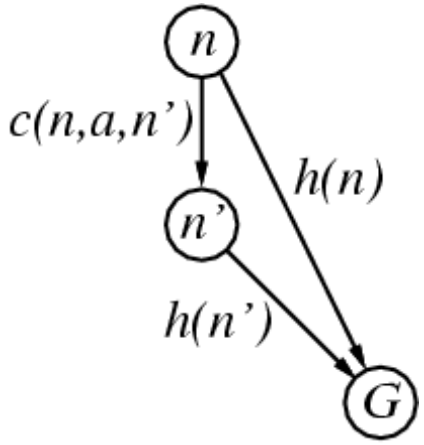
Consistent Heuristics

A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*,

$$h(n) \leq c(n,a,n') + h(n')$$

If *h* is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') && \text{(by def.)} \\ &= g(n) + c(n,a,n') + h(n') && (g(n')=g(n)+c(n.a.n')) \\ &\geq g(n) + h(n) = f(n) && \text{(consistency)} \\ f(n') &\geq f(n) \end{aligned}$$



It's the triangle inequality !

i.e., *f(n)* is non-decreasing along any path.

keeps all checked nodes
in memory to avoid repeated states

Theorem:

If *h(n)* is consistent, A* using `GRAPH-SEARCH` is optimal

Time Complexity

Exponential: $O(b^d)$ where b is the branching factor (the average number of successors per state) and d is the depth of the solution

Can be shown to run in polynomial time if

$$|h(u, v) - d(u, v)| = \mathbf{O}(\ln(d(u, v)))$$

where $d(u, v)$ is the length of the actual shortest path[1]

- *I.e.*, doubling the length of the optimal solution only increases the error by a constant
- Not likely with a road map and the Euclidean distance
- E.g. when $h(u, v) = d(u, v)$

[1]Pearl, Judea (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley.

Summary

This topic has presented the A* search algorithm

- Assumes a hypothetical lower bound on the length of the path to the destination
- Requires an appropriate heuristic
  - Useful for Euclidean spaces (vector spaces with a norm)
- Faster than Dijkstra’s algorithm in many cases
  - Directs searches towards the solution

Shortest Path - Backtracking

Backtracking

With Dijkstra's algorithm, we keep track of all current best paths

- There are at most $|V| - 1$ paths we could extend at any one time
- These can be tracked with a relatively small table

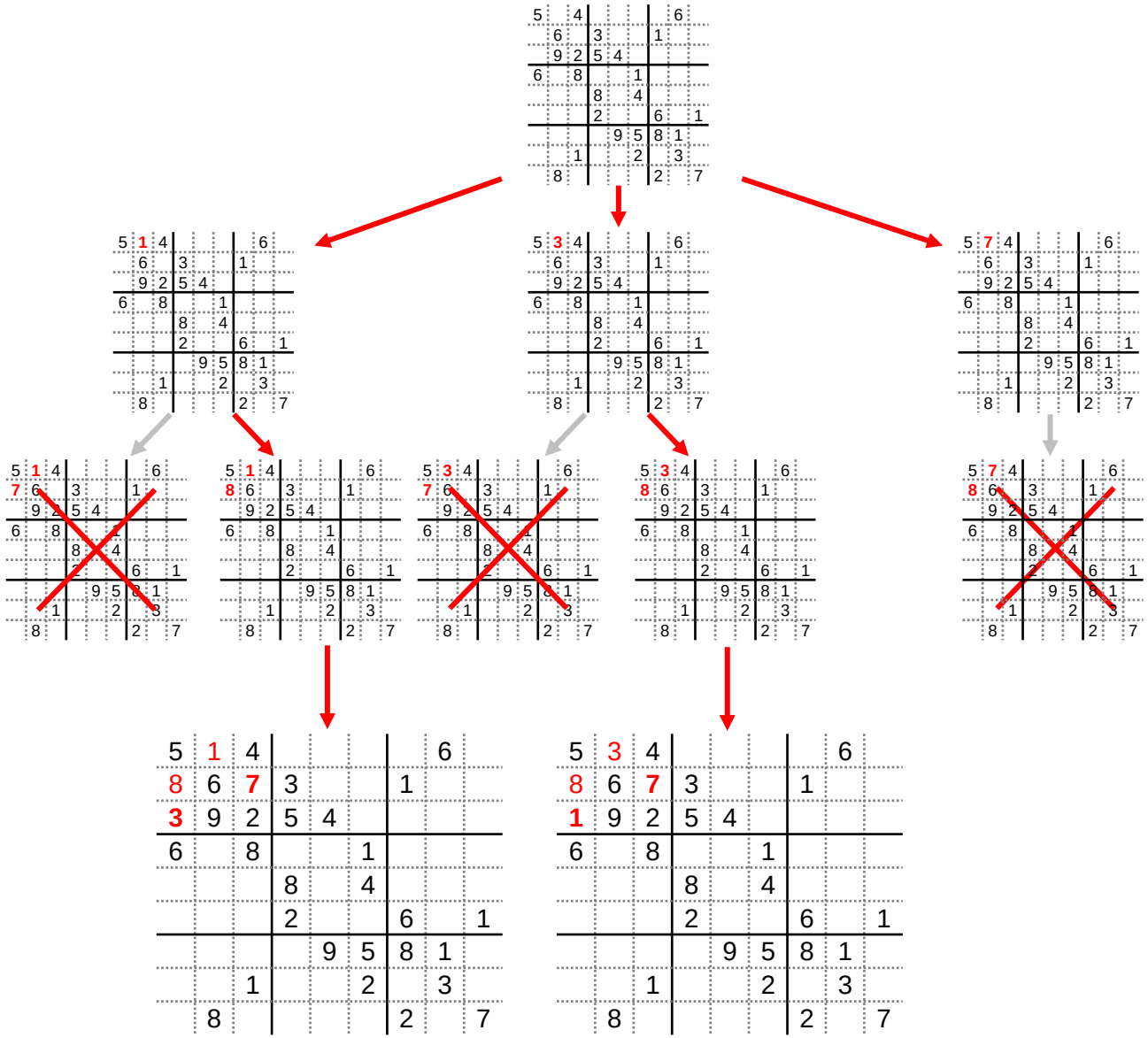
Suppose we cannot evaluate the relative fitness of solutions

- There may just be too many to record efficiently
- Are we left with a brute-force search?

Suppose there are just too many partial solutions at any one time to keep track of

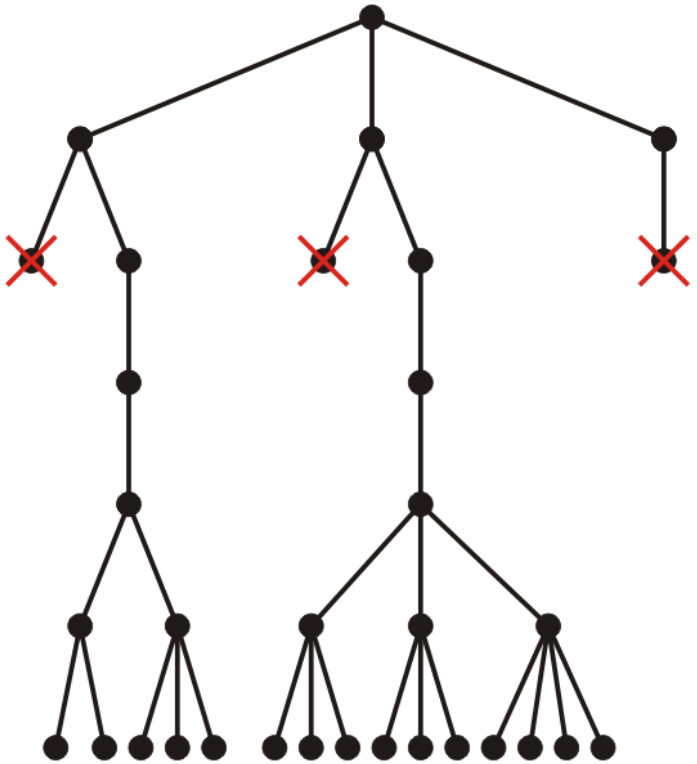
- At any point in time in a game of chess or Go (围棋), there are a plethora of moves, each valid, but the usefulness of each will vary

Sudoku



Sudoku

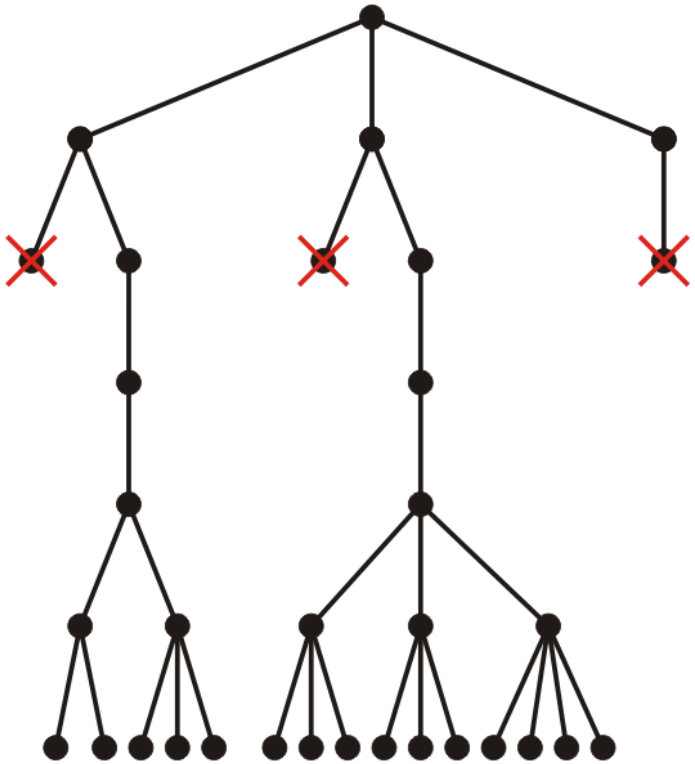
It may seem that this is a reasonably straight-forward method; however, the decision tree continues to branch quick once we start filling the second



Sudoku

A binary tree of this height would have around $2^{54} - 1$ nodes

- Fortunately, as we get deeper into the tree, more get cut



Implementation

Our straight-forward implementation takes a 9×9 matrix

- Default entries are values from 1 to 9, empty cells are `0`
- Two helper functions:

  `bool next_location( int[9][9], int &i, int &j )`
  - Finds the next location empty location returning false if none is found

  `bool is_valid( int[9][9], int i, int j, int value )`
  - Checks if there are any conflicts created if `matrix[i][j]` is assigned `value`
- The `backtracing` function:
  - Finds the next unoccupied cell
  - For each value from 1 to 9, it checks if it is valid to insert that it there
    - If so, backtracking is called recursively on the matrix with that entry set

The main function creates the initial matrix and calls backtrack

Implementation

```
// Find the next empty location in 'matrix'
// If one is found, assign 'i' and 'j' the indexes of that entry
// Otherwise, return false
//  - In this case, the values of 'i' and 'j' are undefined
bool next_location( int matrix[9][9], int &i, int &j ) {
    for ( int i1 = 0; i1 < 3; ++i1 ) {
        for ( int j1 = 0; j1 < 3; ++j1 ) {
            for ( int i2 = 0; i2 < 3; ++i2 ) {
                for ( int j2 = 0; j2 < 3; ++j2 ) {
                    i = 3*i1 + i2;
                    j = 3*j1 + j2;

                    // return 'true' if we find an
                    // unoccupied entry
                    if ( matrix[i][j] == 0 )
                        return true;
                }
            }
        }
    }

    return false;  // all the entries are occupied
}
```

```
// If 'value' already appears in
//  - the row 'm'
//  - the column 'n'
//  - the 3x3 square of entries it (m, n) appears in
// return false, otherwise return true
bool is_valid( int matrix[9][9], int m, int n, int value ) {
    // Check if 'value' already appears in either a row or column
    for ( int i = 0; i < 9; ++i ) {
        if ( (matrix[m][i] == value) || ( matrix[i][n] == value ) )
            return false;
    }

    // Check if 'value' already appears in either a row or column
    int ioff = 3*(m/3);
    int joff = 3*(n/3);

    for ( int i = 0; i < 3; ++i ) {
        for ( int j = 0; j < 3; ++j ) {
            if ( matrix[ioff + i][joff + j] == value )
                return false;  // 'value' already in the 3x3 square
        }
    }

    return true;  // 'value' could be added
}
```

Implementation

```
bool backtrack( int matrix[9][9] ) {
    int i, j;

    // If the matrix is full, we are done
    if ( !next_location( matrix, i, j ) ) {
        return true;
    }

    for ( int value = 1; value <= 9; ++value ) {
        if ( is_valid( matrix, i, j, value ) ) {
            // Assume this entry is part of the
            // solution--recursively call backtrack
            matrix[i][j] = value;

            // If we found a solution, return
            // otherwise, reset the entry to 0
            if ( backtrack( matrix ) ) {
                return true;
            } else {
                matrix[i][j] = 0;
            }
        }
    }

    // No solution found--reset the entry to 0
    return false;
}
```

Implementation

```
int main() {
    int matrix[9][9] = {
        {5, 0, 4, 0, 0, 0, 0, 6, 0},
        {0, 6, 0, 3, 0, 0, 1, 0, 0},
        {0, 9, 2, 5, 4, 0, 0, 0, 0},
        {6, 0, 8, 0, 0, 1, 0, 0, 0},
        {0, 0, 0, 8, 0, 4, 0, 0, 0},
        {0, 0, 0, 2, 0, 0, 6, 0, 1},
        {0, 0, 0, 0, 9, 5, 8, 1, 0},
        {0, 0, 1, 0, 0, 2, 0, 3, 0},
        {0, 8, 0, 0, 0, 0, 2, 0, 7}
    };

    // If found, print out the resulting matrix
    if ( backtrack( matrix ) ) {
        for ( int i = 0; i < 9; ++i ) {
            for ( int j = 0; j < 9; ++j ) {
                std::cout << matrix[i][j] << " ";
            }

            std::cout << std::endl;
        }
    }

    return 0;
}
```

Implementation

In this case, the traversal:

- Recursively calls backtrack 874 times
  - The last one determines that there are no unoccupied entries
- Checks if a placement is valid 7658 times

5	3	4	1	7	8	9	6	2
8	6	7	3	2	9	1	4	5
1	9	2	5	4	6	3	7	8
6	7	8	9	3	1	5	2	4
2	1	5	8	6	4	7	9	3
3	4	9	2	5	7	6	8	1
4	2	3	7	9	5	8	1	6
7	5	1	6	8	2	4	3	9
9	8	6	4	1	3	2	5	7

Backtracking

This should give us an idea, however:

- Perform a traversal
- Do not continue traversing if a current node indicates all descendants are infeasible solutions

Classical applications

Classic applications of this algorithm technique include:

- Eight queens puzzle
- Knight’s tour
- Logic programming languages
- Crossword puzzles

历年真题

9) 寻找最短路径的迪杰斯特拉（Dijkstra）算法**不属于**贪心法。（　　）

7. 图（6 题，共 15 分）

给定如下带权无向图（Weighted Undirected Graph）如图-3 所示：

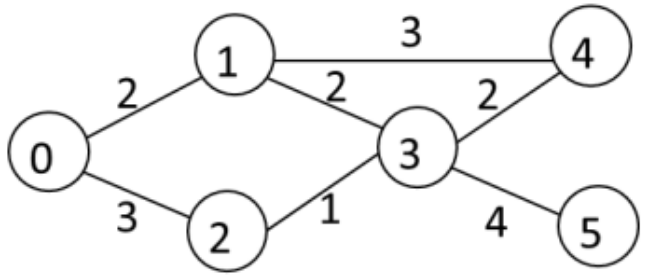


图 3-带权无向图

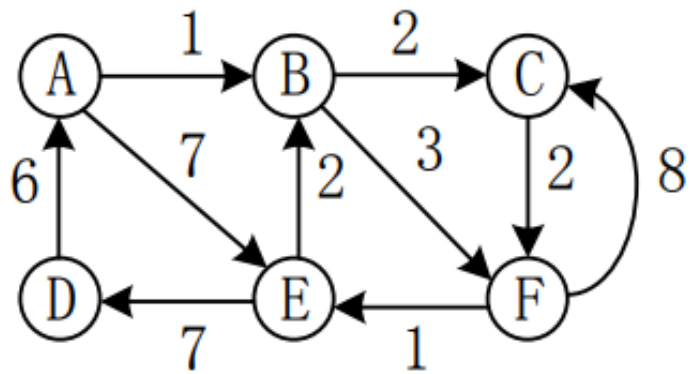
请在答题纸上回答如下问题或者画出相应答案：

4) 求从顶点 0 出发到各个顶点的最短路径，按迪杰斯特拉（Dijkstra）算法，写出逐次选择的顶点。（3 分）
5) 画出题 4）中由顶点 0 到各个顶点的最短路径。（2 分）
6) 将图 3 中连接顶点 i 和顶点 j 的边记为（i, j）。边（1, 4）在原始图中权值为 3，现对其权值进行改变，其它边的权值不变。假设边（1, 4）的权值为 a（a>0 且为整数），求使得顶点 0 到顶点 4 的最短路径通过边（3, 4）的 a 的取值范围。（3 分）

6. Graph (25 points) 图（25 分）

Given the following weighted directed graph,

给定如下带权有向图，



4) draw one shortest-paths tree of the graph rooted at the source A. （10 points）

画出该图的以 A 为出发节点的最短路径树。（10 分）

Q&A