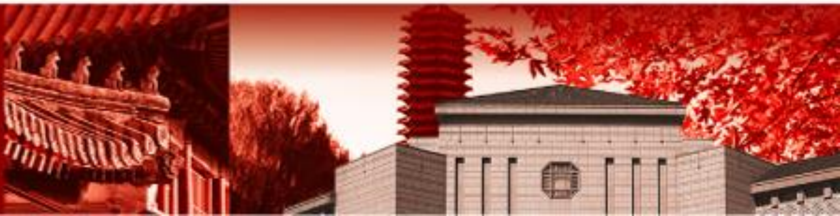
数据结构与算法B

14-最短路径算法



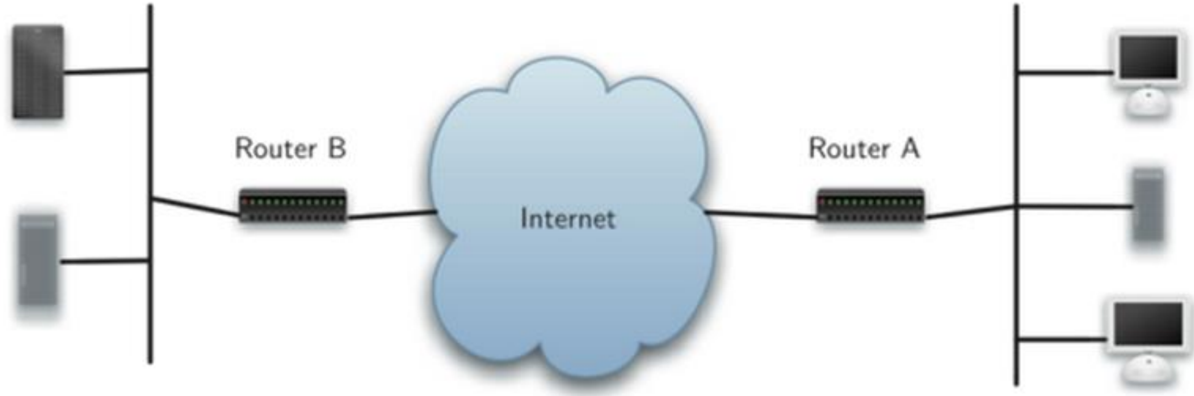
目录

- 14.1 图的最短路径问题
- 14.2 Dijkstra 算法
- 14.3 Floyd 算法

14.1 图的最短路径问题

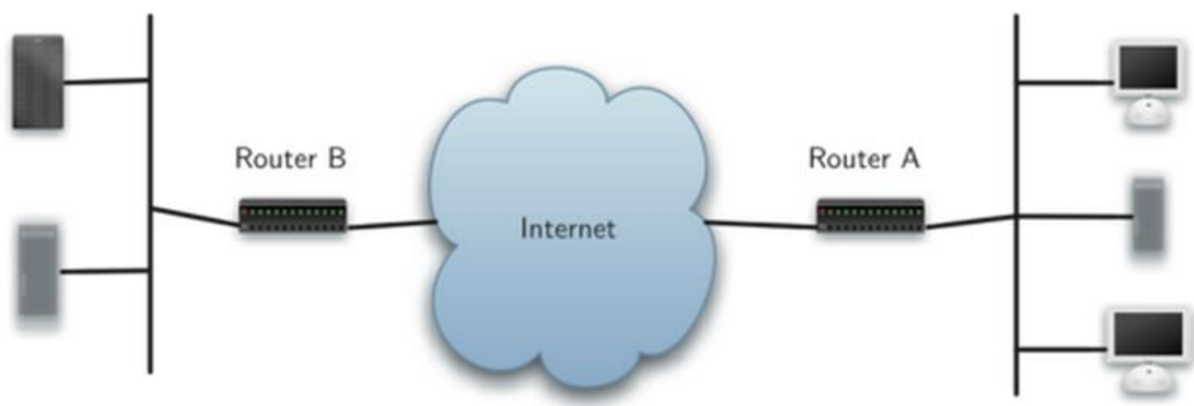
问题引入

- 当我们浏览网页、发送电子邮件，消息传输时，为了将信息从一台计算机传送到另一台计算机，后台借助互联网进行数据传输。
  – 下图大体展示了互联网的通信机制
  – 数据首先从本地局域网发出，经过路由器传送到互联网上，最终到达目标主机所在的局域网
  – 图中的“Internet”是由众多的路由器等设备组成的网络



问题引入

- 网络中任何两个结点的通信都会产生一定的代价
  - 代价大小取决于流量、时间段以及众多其他因素
- 因此，网络可以用一个带权无向图来表示
- 我们要解决的问题是为给定信息找到权重最小的路径



最短路问题

- 在带权无向或有向图上，一个顶点到另一个顶点的边权值之和最小的路径，称为最短路，边权值之和称为最短路的长度。
  - 现实问题中，顶点可能代表地址、阶段性目标、任务等
  - 边可能代表某种操作，例如从一个地方走到另一个地方，或者完成某个工作步骤
  - 边权值可能代表距离、时间、花费等
- 我们先前学过的广度优先搜索算法同样用于求最短路
  - 不过，现在需要考虑路径的总权重，而不是路径包含的边数
  - 如果所有边的权重相等，就同样可以使用广度优先搜索
- 求最短路也可以使用深度优先搜索算法
  - 但是，需要遍历的解空间往往过大，效率无法保证

最短路问题

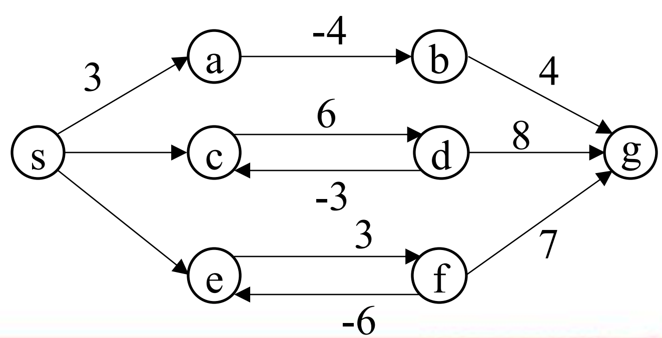
- 最短路径问题的各种变体：
- 单源最短路径问题：给定图 G=(V, E)，我们希望找到从给定源结点 s∈V 到每个结点 v∈V 的最短路径
  - Dijkstra 算法就是解决单源最短路径问题的经典算法
- 单目的地最短路径问题：找到从每个结点 v 到给定目的地结点 t 的最短路径。
  - 对于无向图，与单源最短路径问题是相同的
  - 对于有向图，将图中每条边的方向翻转过来，同样可以转换为单源最短路径问题

最短路问题

- 单结点对最短路径问题：找到从给定结点 u 到给定结点 v 的最短路径
  - 如果解决了结点 u 的单源最短路径问题，那么也就解决了这个问题
- 所有结点对的最短路径问题（多源最短路径问题）：对于每对结点 u 和 v，找到从结点 u 到结点 v 的最短路径
  - 虽然可以对每个结点都运行一遍单源最短路径算法，但通常可以更快地解决这个问题
  - Floyd 算法用于求解所有结点对的最短路径问题
- 接下来的讨论都是基于有向图。对于无向图，可以将每条边改为双向边，从而转化为有向图。

最短路中的负权边

- 某些单源最短路径问题可能包括权重为负值的边。
- 但如果图中存在着权重为负值的回路，且该回路对于结点 v 可达，则从 v 出发到部分顶点的最短路径可能不存在
  - 从源点 s 到回路上的任意结点的路径都不可能是最短路径
  - 因为我们只要沿着任何“最短”路径，再沿着权重为负的回路走一遍，就可以找到一条权重更小的路径
  - 下图中，结点 e, f 形成了一个权重为负，且从源点 s 可达的回路，这使得 s 到 e, f, g 的最短路径不存在



最短路中的负权边

- 对于单源最短路径问题，Dijkstra 算法假设输入图中所有的边权非负，不能在含有负权边的图上运行
  - 其他单源最短路径算法，例如 Bellman-Ford 算法，允许图中含有负权边，且可以识别图中可能存在的负回路，但效率低于 Dijkstra 算法
- 对于多源最短路径问题，Floyd 算法允许图中含有负权边，也可以检测图中可能存在的负回路

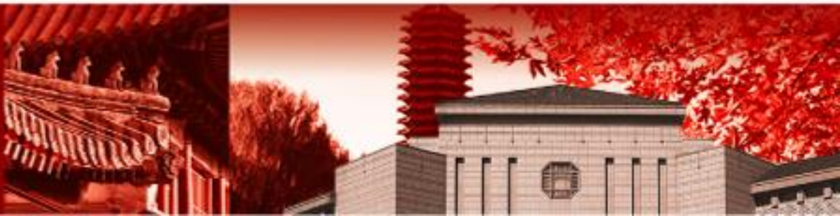
最短路中的回路

- 一条最短路径是否可以包含回路？
  - 最短路径不能包含权重为负值的回路
  - 实际上，最短路径也不能包含权重为正值的回路，因为只要将回路从路径上提出就可以得到一条权重更小的路径
- 这样，就只剩下权重为 0 的回路
  - 权重为 0 的回路不会对路径权重产生影响，我们可以从任何路径上剔除权重为 0 回路，得到另一条权重相同的路径
  - 因此，对于任何最短路径，只要其中含有权重为 0 的回路，我们就可以重复删除这些回路，直到得到一条不包含回路的最短路径
  - 不失一般性，我们在求解问题时只考虑不含回路的最短路径，即最短路径都是简单路径

14.2 Dijkstra算法

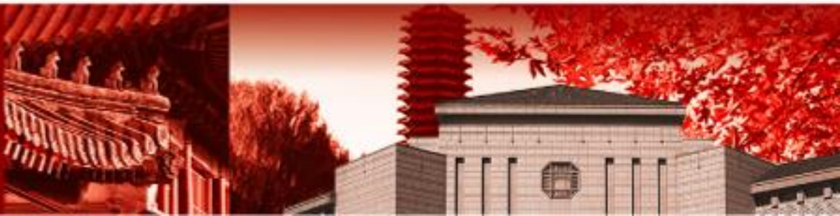
Dijkstra

- Dijkstra 其人： Edsger Wybe Dijkstra (1930-2002), 荷兰皇家艺术与科学学院的院士，美国科学院院士，英国计算协会Fellow
- 获得多种奖项：
  - 1972 Turing Award
  - 1974 AFIPS Harry Goode Award
  - 1982 IEEE Computer Pioneer Award
  - 1989 ACM SIGCSE Award for outstanding Contributions to Computer Science Education



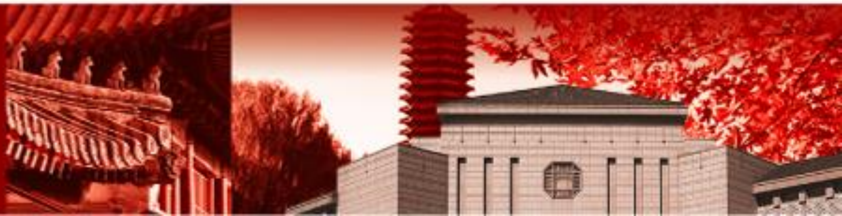
Dijkstra

- Dijkstra 是计算机科学与工程领域的许多概念、术语的缔造者：
  - 结构化程序设计、问题分解
  - 同步、死锁、哲学家就餐问题
  - 栈、向量
  - goto 语句有害论
- 关于 Dijkstra 的名字
  - 荷兰语中的 ‘ij’ 是一个特殊字母，发音类似英语中的字母 A
  - 他的名字的规范读音是 /ˈdɛıkstra/
  - 中文译名为戴克斯特拉，或较多使用的迪杰斯特拉



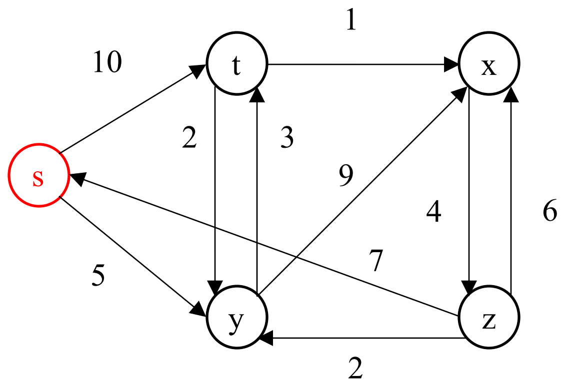
Dijkstra 算法的基本思路

- Dijkstra 算法运行时维护一组结点集合 U
  - 若结点 t 被加入集合 U，表明已经找到了由源点 s 到结点 t 最短路径
  - 初始时 U 仅包括源点
- 同时，维护每个顶点到源点 s 的最短路径长度估计列表
  - Dist[s] 初始化为 0，其余顶点i到s的Dist[i] 初始化为 ∞
- 进行重复迭代：
  - 选取 V – U 集合中最短路径长度估计最小的顶点 u，加入到集合 U 中
  - 根据顶点 u 关联的边，更新 V – U 中所有顶点的最短路径长度估计
  - 直到全部顶点都加入 U，求解完成



Dijkstra 算法示例

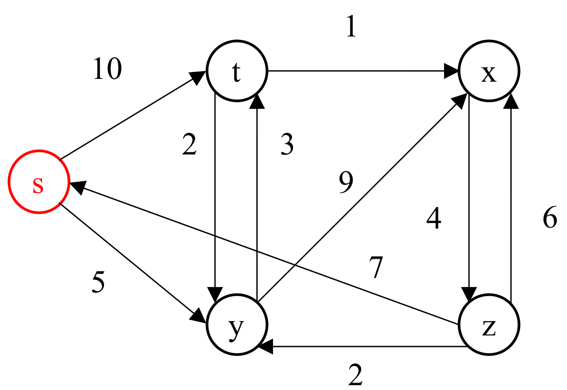
- 维护一个Dist 列表，其中Dist[i]表示目前为止发现的源点s到顶点i的最短路的长度。
- 维护一个Prev 列表，Prev[i]表示目前为止发现的，从s到i的最短路（不一定是最终结果）上的i的前驱顶点。
  - 初始时，Dist[s] = 0，其余顶点i的Dist[i] = ∞
  - U = {s}, V – U = {t, x, y, z}



Vertex	s	t	x	y	z
Dist	0	∞	∞	∞	∞
Prev	None	None	None	None	None

Dijkstra 算法示例

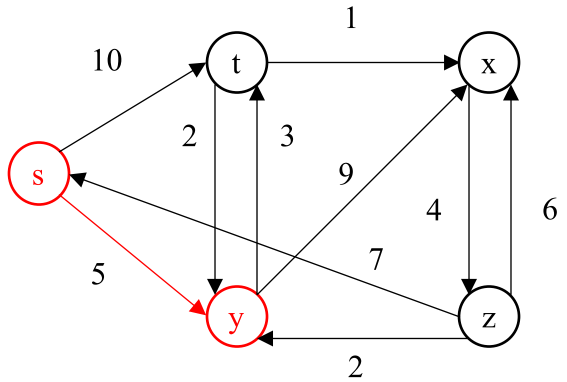
- 根据顶点 s 关联的边，更新 V – U 中顶点的 Dist 值
  - 顶点 t：Dist[t] = 0 + 10 = 10，Prev = s
  - 顶点 y：Dist[y] = 0 + 5 = 5，Prev = s



Vertex	**s**	**t**	**x**	**y**	**z**
Dist	0	10	∞	5	∞
Prev	None	s	None	s	None

Dijkstra 算法示例

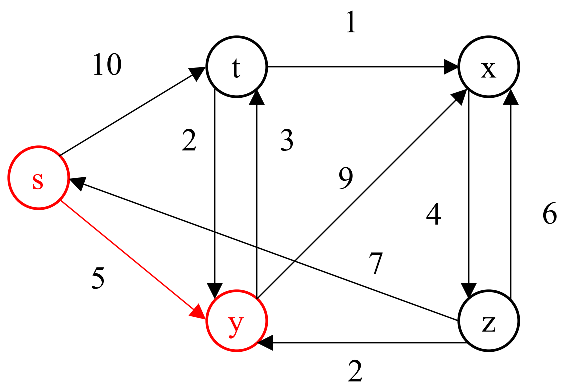
- 选取 V – U 中 Dist 值最小的顶点，即顶点 y，加入集合 U



Vertex	s	t	x	y	z
Dist	0	10	∞	5	∞
Prev	None	s	None	s	None

Dijkstra 算法示例

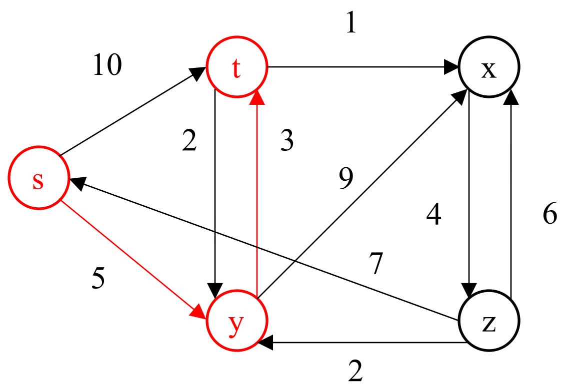
- 选取 V – U 中 Dist 值最小的顶点，即顶点 y，加入集合 U
- 根据顶点 y 关联的边，更新 V – U 中顶点的 Dist 值
  - 顶点 t：Dist[t] = 5 + 3 = 8 < 10，Prev = y
  - 顶点 x：Dist[x] = 5 + 9 = 14，Prev = y



Vertex	**s**	**t**	**x**	**y**	**z**
Dist	0	8	14	5	∞
Prev	None	y	y	s	None

Dijkstra 算法示例

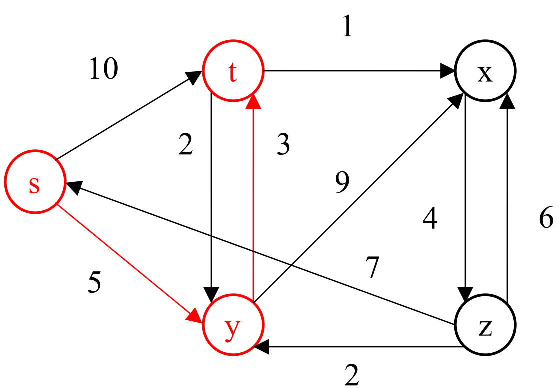
- 选取 V – U 中 Dist 值最小的顶点，即顶点 t，加入集合 U



Vertex	s	t	x	y	z
Dist	0	8	14	5	∞
Prev	None	y	y	s	None

Dijkstra 算法示例

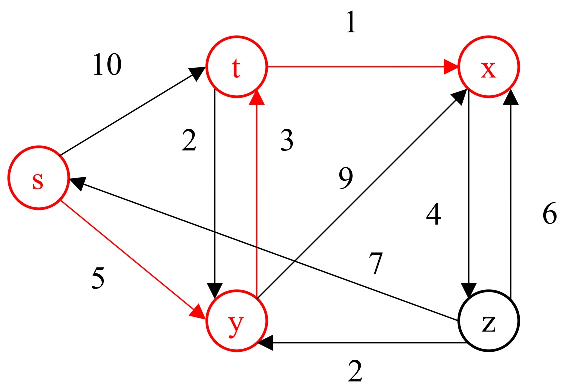
- 选取 V – U 中 Dist 值最小的顶点，即顶点 t，加入集合 U
- 根据顶点 t 关联的边，更新 V – U 中顶点的 Dist 值
  - 顶点 x：Dist[x] = 8 + 1 = 9 < 14，Prev = t



Vertex	**s**	**t**	**x**	**y**	**z**
Dist	0	8	9	5	∞
Prev	None	y	t	s	None

Dijkstra 算法示例

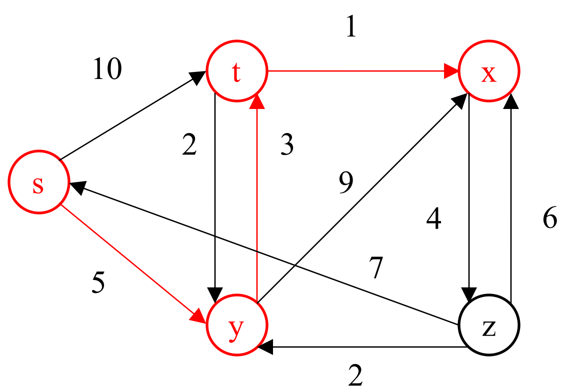
- 选取 V – U 中 Dist 值最小的顶点，即顶点 x，加入集合 U



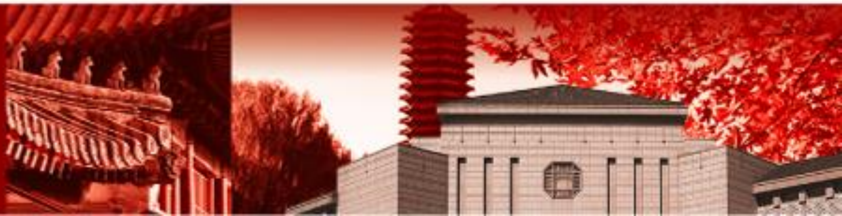
Vertex	s	t	x	y	z
Dist	0	8	9	5	∞
Prev	None	y	t	s	None

Dijkstra 算法示例

- 选取 V – U 中 Dist 值最小的顶点，即顶点 x，加入集合 U
- 根据顶点 x 关联的边，更新 V – U 中顶点的 Dist 值
  – 顶点 z：Dist[z] = 9 + 4 = 13，Prev = x

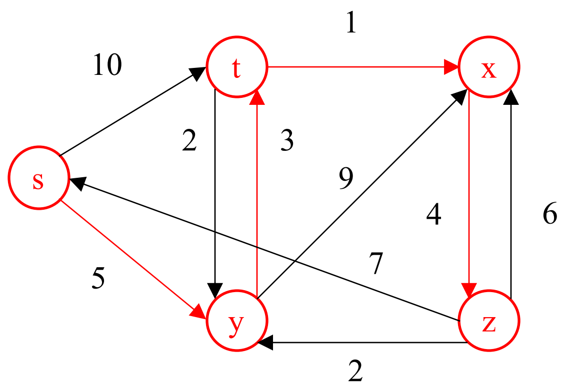


Vertex	s	t	x	y	z
Dist	0	8	9	5	13
Prev	None	y	t	s	x

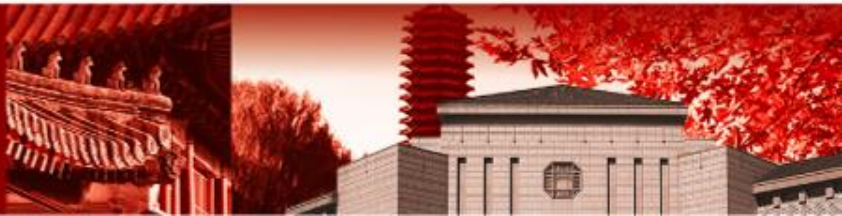


Dijkstra 算法示例

- 选取 V – U 中 Dist 值最小的顶点，即顶点 x，加入集合 U
- 根据顶点 x 关联的边，更新 V – U 中顶点的 Dist 值
  – 顶点 z：Dist = 9 + 4 = 13，Prev = x
- 最终将顶点 z 加入集合 U，算法结束

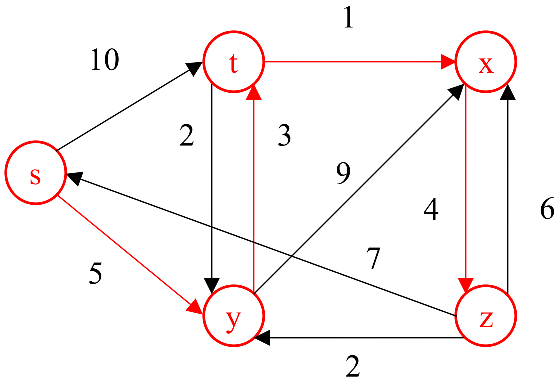


Vertex	**s**	**t**	**x**	**y**	**z**
Dist	0	8	9	5	13
Prev	None	y	t	s	x



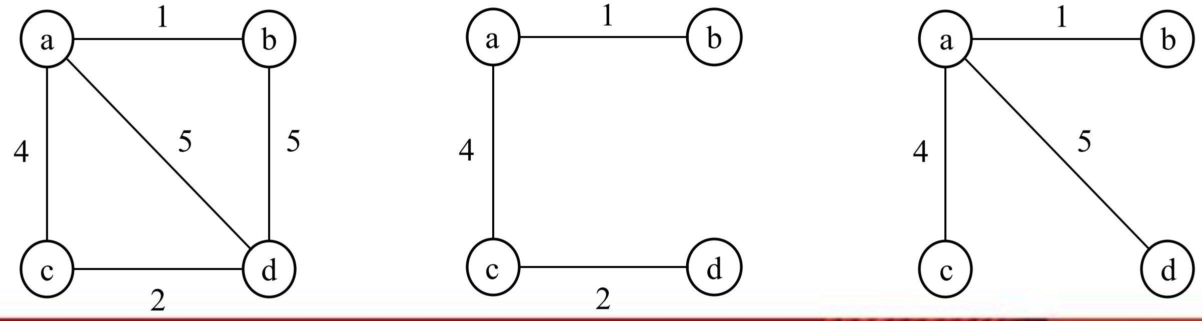
思考：Dijkstra 算法与 Prim 算法

- Dijkstra 算法与 Prim 算法的流程非常相似：
  – 从一个点出发，维护一个已确认的顶点集合，以及每个顶点的距离值；每次选取最小的点加入，并更新未加入顶点的距离值
- Dijkstra 算法同样得到了一棵生成树，该生成树是否一定是最小生成树？
- Dijkstra 算法与 Prim 算法的区别是什么?



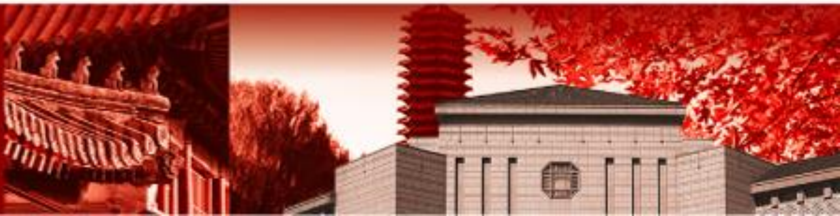
思考：Dijkstra 算法与 Prim 算法

- 适用的问题不同
  - Dijkstra 算法适用于有向图或无向图上的单源最短路问题
  - Prim 算法以及 Kruskal 算法，只适用于无向图上的最小生成树问题
- 结点距离值的定义不同
  - Dijkstra 算法中，定义为最短路径长度估计，是到源点 s 的距离
  - Prim 算法中，是到集合中相邻的点的距离
  - 因此，在无向图上，Dijkstra 算法也并不能用于求解最小生成树
  - 下图展示了在无向图上运行 Prim 算法与 Dijkstra 算法得到的结果



Dijkstra 算法的正确性

- Dijkstra 算法利用了最短路问题的最优子结构：最短路径的子路径也是最短路径
- 给定带权有向图 $G = (V, E)$，设 $P = (v_0, v_1, \cdots, v_k)$ 为从结点 v_0 到 v_k 的一条最短路径，且对于任意的 $0 \le i \le j \le k$，设 $P_{ij} = (v_i, v_{i+1}, \cdots, v_j)$ 为路径 P 中从结点 v_i 到 v_j 的子路径，则 P_{ij} 也是结点 v_i 到 v_j 的最短路径
- 利用反证法易证
  - 假设存在从结点 v_i 到 v_j 的路径 P_{ij}^* 权重小于 P_{ij}
  - 则将 P 中 P_{ij} 的对应部分替代为 P_{ij}^*，就得到从结点 v_0 到 v_k 的路径 P^*，权重小于 P，与 P 是最短路径矛盾

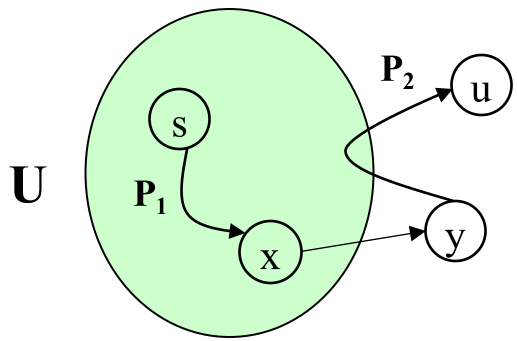


Dijkstra 算法的正确性

- Dijkstra 算法总是选择集合 V – U 中距离最小的结点加入到 U 中，该算法使用的也是贪心策略
- 可以证明：算法运行过程中，每个结点被加入集合 U 时，其 Dist 值与最短路径长度相等。因而 Dijkstra 算法能够找到所有最短路径
- 同样使用归纳证明
  - 归纳基础：该性质对于源点 s 显然成立
  - 下面需要证明：对于顶点 u，假设之前加入 U 的顶点都找到了最短路径，则 u 加入 U 时也已经找到了最短路径

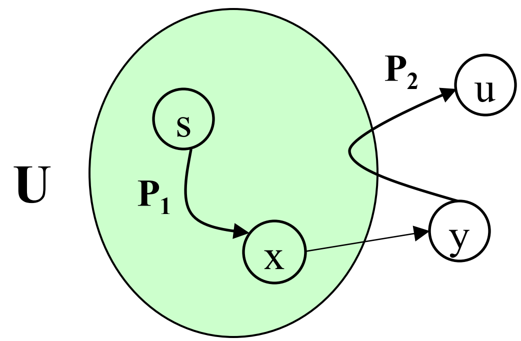
Dijkstra 算法的正确性

- 考虑源点 s 到 u 实际的最短路径。
- 由于 u 此时仍不属于 U，该路径必然跨越了集合 U 与 V – U
- 设第一次跨越为从顶点 x 到顶点 y，可以将最短路进行划分为 s – x – y – u
  - s 可能与 x 重合，u 也可能与 y 重合，此时 P_1 或 P_2 为空



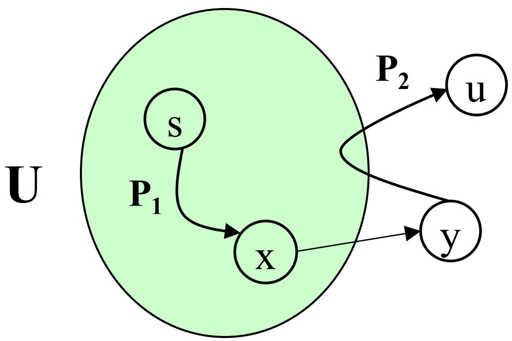
Dijkstra 算法的正确性

- 根据归纳假设将 x 加入 U 时，确认了 x 的最短路径
- 将 x 加入 U 后，必然更新了顶点 y 的距离值。
- 根据最优子结构，s – x – y 是 s 到 y 的最短路径，因此更新 y 的距离值后，其距离值已经与最短路径长度相等



Dijkstra 算法的正确性

- Dijkstra 算法假设图中所有的边权非负，因此，u 的最短路径长度不小于 y 的最短路径长度
- 而算法在下一步选择了 u，说明当前 u 的距离值不超过 y 的距离值（也等于 y 的最短路径长度）
- 从而说明，u 的最短路径长度等于 y 的最短路径长度，也等于当前 u 的距离值



Dijkstra 算法：伪代码

```
# 输入一个带权图 G 和源点 s，输出 s 到所有顶点的最小路径长度
def Dijkstra(G, s):
    初始化 Dist, Prev 列表
    创建一个最小堆 Heap，插入所有结点，键值为 Dist 值
    while 堆非空:
        取出堆顶元素 u
        依据 u 关联的所有边，更新 Dist, Prev 列表
        对于发生更新的元素，相应调整堆的结构（DECREASE-KEY）
    返回 Dist 及 Prev 值
```

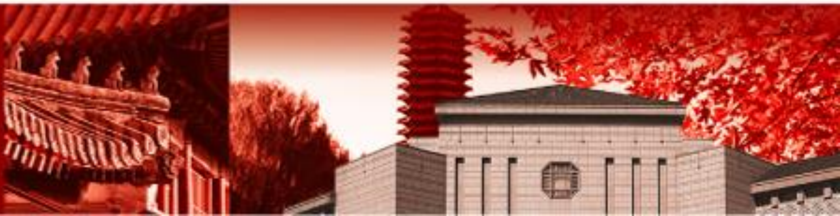
Dijkstra 算法的时间复杂度

- 记 n = |V|, e = |E|
- 算法的主要代价在于，不断从堆中取出最小元素，以及调整堆的操作
  - 要进行 O(n) 次取出最小元素操作，代价为 O(nlog n)
  - 每个结点加入 V’ 后都需要更新其全部邻居结点（DECREASE-KEY）
  - 最坏情况下，要进行 O(e) 次更新，代价为 O(elog n)
- 基于堆实现的 Dijkstra 算法复杂度为 O((n+e)log n)
- 每次使用遍历方法寻找距离最小的顶点，实现的 Dijkstra 算法复杂度为 $O(n^2)$

14.3 Floyd算法

Floyd 算法

- 为了计算图中任意源点到任意终点的最短路径：
  - 可以执行 n 次单源最短路径算法，复杂度为 $O(n*(n+e)*\log n)$ 或 $O(n^3)$
  - 还可以使用动态规划的方法，即 Floyd 算法
- Floyd 算法也称 Floyd-Warshall 算法
  - 该算法用于求解多源最短路径问题
  - 算法允许图中存在权值为负的边
  - 算法假定图中不存在负回路，但算法本身也可以用于检测负回路是否存在



Floyd 算法

- 动态规划的原问题：对于任意顶点 v_i, v_j，求出从 v_i 到 v_j 的最短路径长度
- 定义如下的子问题：对于任意顶点 v_i, v_j，求出从 v_i 到 v_j 且中途只经过 $v_1, v_2, \cdots, v_k$ 的最短路径长度（$0 \le k \le n$）
  - 这样，原问题就是取 $k = n$ 时的子问题，即“最大”的子问题
- 用最短路径权重矩阵 D 来记录子问题的答案
  - $D^k(i,j)$ 表示规模为 k 的子问题中，从 v_i 到 v_j 的最短路径长度
- 动态规划的边界条件
  - $k = 0$ 时的子问题要求路径不能途径任何顶点
  - 因此，D^0 实际上就是原图的带权邻接矩阵，不相邻的顶点之间权值为无穷

Floyd 算法

- 状态转移方程：如何由 D^k 计算 D^{k+1} ？
- 与规模为 k 的子问题相比，规模为 $k+1$ 的子问题允许路径中经过一个新的顶点 v_{k+1}
- 考虑新的子问题中，最短路径是否经过了 v_{k+1}
  - 若 $D^{k+1}(i,j)$ 对应的最短路径中没有经过 v_{k+1}：

$$D^{k+1}(i,j) = D^k(i,j)$$

  - 若 $D^{k+1}(i,j)$ 对应的最短路径中经过了 v_{k+1}，可以将该路径拆分成，只途经 $v_1, v_2, \cdots, v_k$，由 v_i 先到达 v_{k+1}，再到 v_j

$$D^{k+1}(i,j) = D^k(i,k+1) + D^k(k+1,j)$$

  - 两种情况取最小值：

$$D^{k+1}(i,j) = \min\{D^k(i,j), D^k(i,k+1) + D^k(k+1,j)\}$$

Floyd 算法

- 最短路径的构建
  - 为了得到最短路径本身，可以设置 Prev 矩阵记录最短路的前驱顶点
- 每次根据状态转移方程计算子问题时，
  - 若 $D^{k+1}(i,j)$ 对应的最短路径中没有经过 v_{k+1}：

$$Prev^{k+1}(i,j) = Prev^{k}(i,j)$$

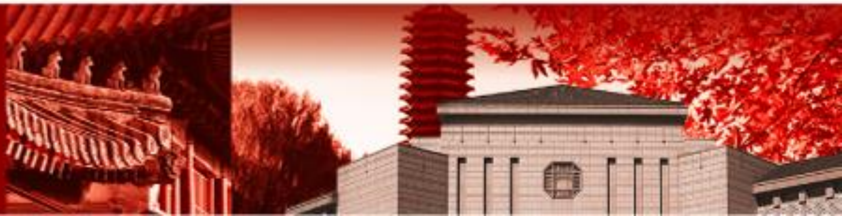
  - 若 $D^{k+1}(i,j)$ 对应的最短路径中经过了 v_{k+1}：

$$Prev^{k+1}(i,j) = Prev^{k}(k+1,j)$$

- 算法的时间复杂度：
  - 算法顺序计算 O(n) 个子问题，每个子问题进行 $O(n^2)$ 次迭代，每次迭代的代价为 O(1)
  - 总的时间复杂度为 $O(n^3)$

Floyd 算法：伪代码

```
# 输入图 G，输出任意结点对之间的最短路及其长度
def Floyd(G):
    初始化距离矩阵 D 为 G 的邻接矩阵
    按照邻接矩阵，初始化前驱矩阵 P
    for k = 1 to n:          // n 为结点个数
        for i = 1 to n:
            for j  = 1 to n:
                if D[i][j] > D[i][k] + D[k][j]:
                    D[i][j] = D[i][k] + D[k][j]
                    P[i][j] = P[k][j]
    返回 D, P 矩阵
```

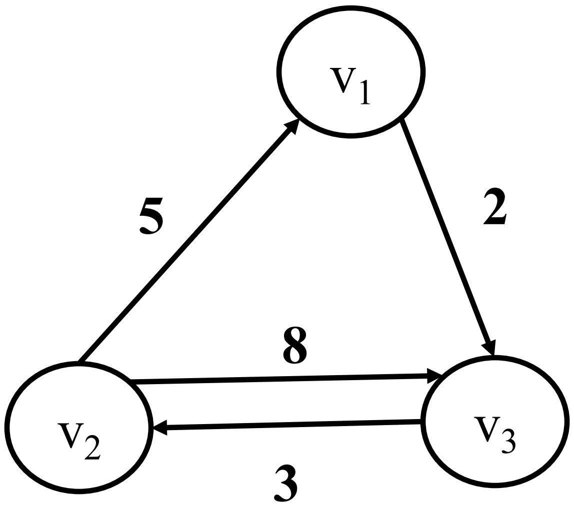


Floyd 算法：示例

$D^{(0)}$ 即为图的带权邻接矩阵

$$D^{(0)} = \begin{pmatrix} 0 & \infty & 2 \\ 5 & 0 & 8 \\ \infty & 3 & 0 \end{pmatrix}$$

$$Pred^{(0)} = \begin{pmatrix} None & None & 1 \\ 2 & None & 2 \\ None & 3 & None \end{pmatrix}$$

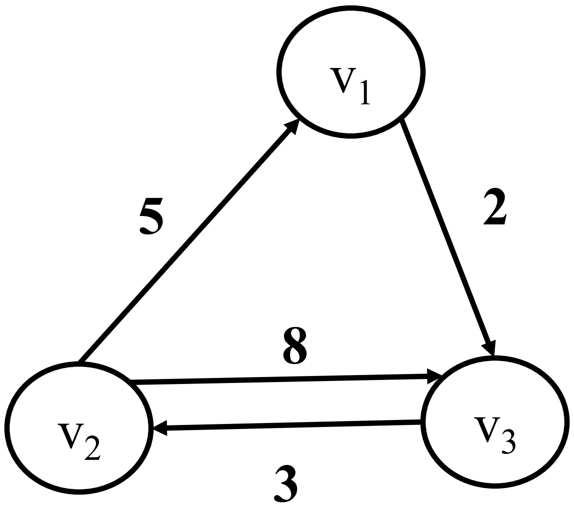


Floyd 算法：示例

$$D^{(1)}[i, j] = \min\{D^{(0)}[i, j], D^{(0)}[i, 1]+D^{(0)}[1, j]\}$$

$$D^{(0)} = \begin{pmatrix} 0 & \infty & 2 \\ 5 & 0 & 8 \\ \infty & 3 & 0 \end{pmatrix} \quad D^{(1)} = \begin{pmatrix} 0 & \infty & 2 \\ 5 & 0 & 7 \\ \infty & 3 & 0 \end{pmatrix}$$

$$Pred^{(1)} = \begin{pmatrix} None & None & 1 \\ 2 & None & 1 \\ None & 3 & None \end{pmatrix}$$

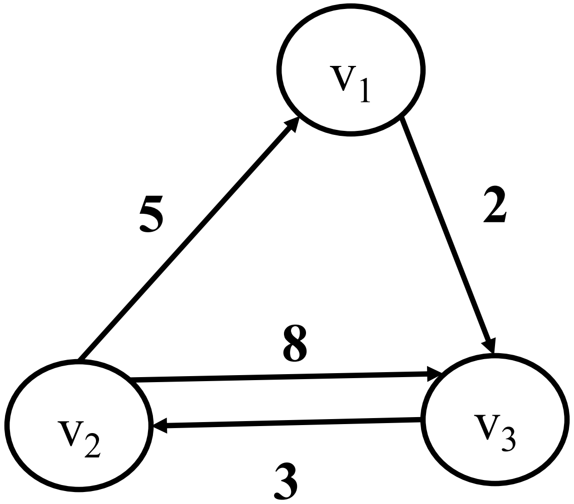


Floyd 算法：示例

$$D^{(2)}[i, j] = \min\{D^{(1)}[i, j], D^{(1)}[i, 2]+D^{(1)}[2, j]\}$$

$$D^{(1)} = \begin{pmatrix} 0 & \infty & 2 \\ 5 & 0 & 7 \\ \infty & 3 & 0 \end{pmatrix} \quad D^{(2)} = \begin{pmatrix} 0 & \infty & 2 \\ 5 & 0 & 7 \\ 8 & 3 & 0 \end{pmatrix}$$

$$Pred^{(2)} = \begin{pmatrix} None & None & 1 \\ 2 & None & 1 \\ 2 & 3 & None \end{pmatrix}$$

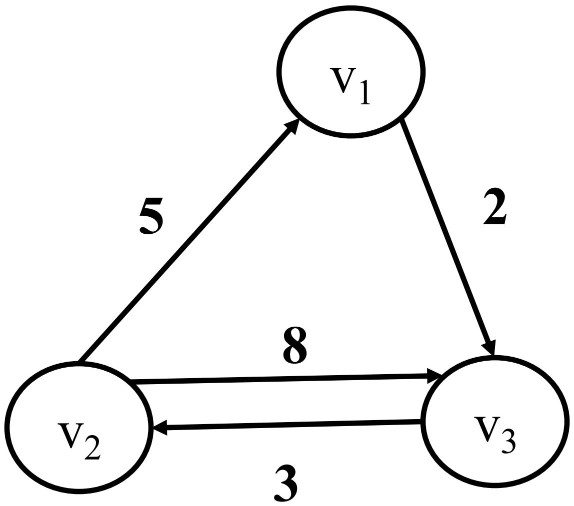


Floyd 算法：示例

$$D^{(3)}[i,j]=\min\{D^{(2)}[i,j],\ D^{(2)}[i,3]+D^{(2)}[3,j]\}$$

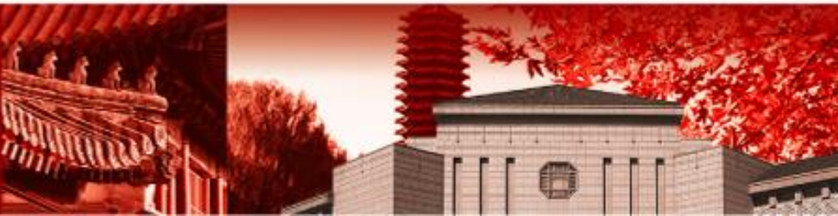
$$D^{(2)}=\begin{pmatrix}0 & \infty & 2\\ 5 & 0 & 7\\ 8 & 3 & 0\end{pmatrix}\quad D^{(3)}=\begin{pmatrix}0 & 5 & 2\\ 5 & 0 & 7\\ 8 & 3 & 0\end{pmatrix}$$

$$Pred^{(3)}=\begin{pmatrix}None & 3 & 1\\ 2 & None & 1\\ 2 & 3 & None\end{pmatrix}$$



负回路的检测

- 思考：如何利用 Floyd 算法的输出来判断，图中是否存在负回路？



负回路的检测

- 思考：如何利用 Floyd 算法的输出来判断，图中是否存在负回路？
- 只需要检查所有结点到自身的最短路径长度即可
  - 在不含负回路的图中，结点到自身的最短路径长度为 0
  - 若含有负回路，则算法得到的负回路上的点到自身的最短路径长度为负

思考

- 给定无向图 G = (V, E)，如何计算任意两个顶点之间是否连通？如果 G 为有向图，又应该如何计算连通性？

思考

- 给定无向图 G = (V, E)，如何计算任意两个顶点之间是否连通？如果 G 为有向图，又应该如何计算连通性？
- 对于无向图：
  – 可以从给定的源点开始进行 DFS/BFS 遍历，遍历到终点则说明连通
  – 可以利用并查集，依据边建立等价类，表示连通这一等价关系
  – 将所有边赋予权值为 1，运行 Floyd 算法，得到的结果中若两个顶点之间最短路长度为无穷，表示不连通；否则表示连通
- 对于有向图：
  – 仍可以使用 Floyd 算法，以及基于 DFS/BFS 的遍历方法
  – 并查集将不再适用，此时结点的连通性也有方向，不构成等价关系