Undirected Graph

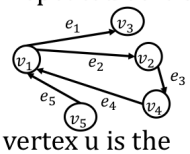
- An undirected graph is a pair of (V, E) where:
  - V is a set of elements, each of which called a node
  - E is a set of unordered pairs{u,v} such that u and v are nodes

Directed Graph

- An directed graph is a pair of (V, E) where:
  - V is a set of elements, each of which called a node
  - E is a set of unordered pairs{u,v} where u and v are nodes, we say there is a directed edge from u to v.
- A directed edge (u,v) is said to be an outgoing edge of u, and incoming edge of v. Accordingly, v is an out-neighbor of u, and u is in-neighbor of v.

Definitions of Graph

- Let G = (V, E) be a graph. A path in G is a sequence of nodes $(v_1, v_2, \dots, v_k)$ such that
  - For every $i \in [1, k-1]$, there is an edge between v_i and v_{i+1}.
- A cycle in G is a trail in which the only repeated vertices are the first and last vertices.
- Example:
  - Cycle: (v_1, v_2, v_4, v_1); Path: (v_5, v_1, v_2, v_4)



- In an undirected graph, the degree of vertex u is the number of edges of u
- In a directed graph, the out-degree of a vertex u is the number of outgoing edges of u, and its in-degree is the number of its incoming edges

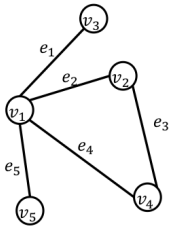
In degree: $d^+(v_1)=2$
out degree: $d^-(v_1)=2$

在无向图中,
$\sum_{u \in V} d(u) = 2|E|$

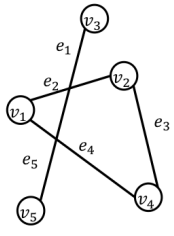
在有向图中,
$\sum_{u \in V} d^+(u) = |E|$
$\sum_{u \in V} d^-(u) = |E|$

Connected Graph (连通图)

- An undirected graph G=(V,E) is connected if, for any two distinct vertices u and v, G has a path from u to v.



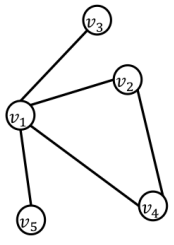
connected



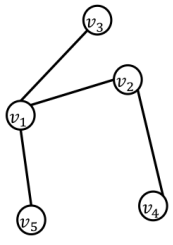
not connected

Graph, Tree, Forest

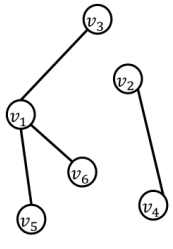
- A tree is a connected undirected graph contains no cycles.
- Forest is a set of disjoint trees.



Graph, not tree



Graph, tree

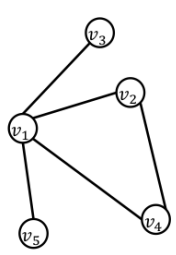


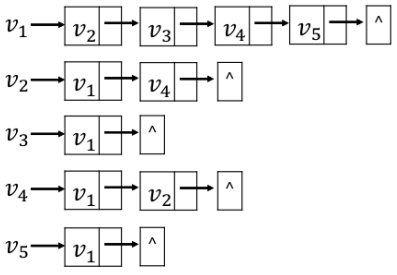
Graph, forest

Adjacency List：邻接表

Undirected Graph

- Each vertex u ∈ V is associated with a linked list that enumerates all the vertices that are connected to u.

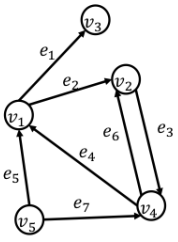


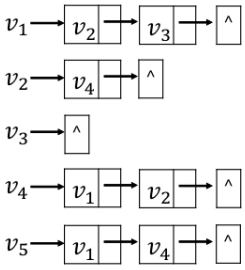


- Space = O(|V|+|E|)

Directed Graph

- Each vertex u ∈ V is associated with a linked list that enumerates all the vertices v ∈ V that there is an edge from u to v.



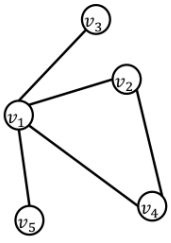


- Space = O(|V| + |E|)

Adjacency Matrix：邻接矩阵

Undirected Graph

- A |V|*|V| matrix A where A[u,v] = 1 if (u, v) ∈ E, or 0 otherwise

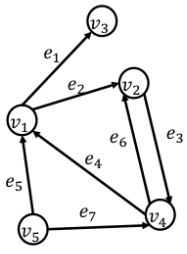


	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	1	1
v_2	1	0	0	1	0
v_3	1	0	0	0	0
v_4	1	1	0	0	0
v_5	1	0	0	0	0

- A must be symmetric
- Space = O(|V|²)

Directed Graph

- Defined in the same way as in the undirected graph



	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	0	0
v_2	0	0	0	1	0
v_3	0	0	0	0	0
v_4	1	1	0	0	0
v_5	1	0	0	1	0

- A may not be symmetric.
- Space = O(|V|²)

Graph Traversal

Tree		Graph
level	→	BFS
inorder preorder postorder	→	DFS

Shortest Path

- Let G = (V, E) be a directed graph. A path in G is a sequence of nodes $(v_1, v_2, \dots, v_k)$ such that
  - For every $i \in [1, k-1]$, there is an edge between v_i and v_{i+1}.
  - E.g., (v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k)
  - Sometimes, we also denote the path as $v_1 \to v_2 \to \cdots \to v_k$
- The path is said to be from v_1 to v_k, the length of the path is $k-1$.
- Given two vertices $u, v \in V$, a shortest path from u to v is a path from u to v that has the minimum length among all the paths from u to v .
- If there is no path from u to v, then v is said to be unreachable from u.

Breadth First Search (BFS)

- At the beginning, color all vertices in graph white. And create an empty BFS tree T.
- Create a queue Q. Insert the source vertex s into Q, and color it yellow (which means "in the queue")
- Make s the root of T.
- Repeat the following until Q is empty
  - De-queue from Q the first vertex v
  - For every out-neighbor u of v that is still white
    - 2.1 Enqueue u into Q, and color u yellow
    - 2.2 Make u a child of v in the BFS tree T.
  - Color v red (meaning v is visited)

初始化graph G全部节点为白色. 创建一个空的BFS树T.
创建一个队列Q. 往Q中加入源节点s. 并将其变成黄色.
令s为T的根节点
while(!Q.isEmpty())
    v = Q.dequeue() v ← red
    for(v's neighbour u)
        if(u is still white)
            Q.enqueue(u) u ← yellow
            T(u is v's child)
        endif
    endfor
endwhile
事实上黄/红相当于一种颜色

Single Source Shortest Path (sssp)

从一个源节点出发, 到其他节点的最短路径.
BFS树中源a到任意节点x的路径即最短路径

Complexity Analysis

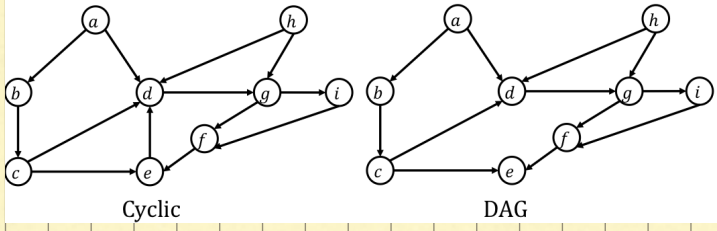
- When a vertex v is dequeued, we spend $O(1+d^+(v))$ time processing it, where $d^+(v)$ is the out-degree of v.
- Clearly, every vertex enters the queue at most once.
- The total running time of BFS is therefore:

$$O\left(\sum_{v \in V}(1 + d^+(v))\right) = O(|V| + |E|)$$

$O(1+d^+(v))$ = 出一个节点 + 该节点的 in degree

Directed Acyclic/ Cyclic Graph

- If a directed graph contains no cycles, we say that it is a directed acyclic graph (DAG). Otherwise, G is Cyclic.
- DAG is extremely important concept in Computer Science, e.g., spark, tensorflow
- Example



Cyclic DAG

有向无环图 DAG

Depth First Search (DFS)

- At the beginning, color all vertices in the graph white, and create an empty DFS tree T.
- Create a stack S. Pick an arbitrary vertex v. Push v into S, and color it yellow (which means "in the stack")
  - What is the difference between BFS and DFS underlying data structure?
  - BFS → Queue, DFS → Stack
- Make v the root of T
- Repeat the following until S is empty
  - Let v be the vertex that currently tops the stack S (do not remove v from S)
  - Does v still have a white out-neighbor
    - 2.1 If yes: let it be u.
      - Push u into S, and color u yellow
      - Make u a child of v in the DFS-tree T
    - 2.2 If no, pop v from S, and color v red (meaning v is visited)
  - If there are still white vertices, repeat the above by restarting from an arbitrary white vertex v', creating a new DFS tree rooted at v'.

初始化graph G全部节点白色，创建空DFS树T.
创建栈S. 取任一节点v，加入S，并改成黄色.
令v为T的根节点.
while(!S.isEmpty())
  u = S.top()
  for(v's neighbour u)
    if(u is still white)
      S.push(u) u ← yellow
      T(u is v's child)
    endif
  endfor
  S.pop() (将节点v pop出去) v ← red
endwhile
若图中仍有白色节点，则任取一个加入S中继续DFS.

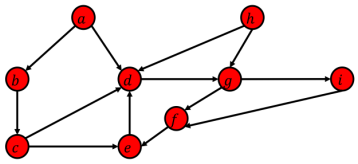
DFS最终也会生成一棵tree(或者forest)

Complexity Analysis

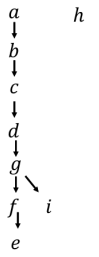
- DFS can be implemented efficiently as follows.
  - Store G in the adjacency list format
  - For every vertex v, remember the out-neighbor to explore next
  - O(|V|+|E|) stack operations
  - Use an array to remember the colors of all vertices
- Hence, the total running time is O(|V|+|E|).

Edge classification

- Suppose we have already built a DFS-forest T.
- Let (u,v) be an edge in G (remember that the edge is directed from u to v). It can be classified into:
  - Forward edge: u is a proper ancestor of v in a DFS-tree of T.
  - Backward edge: u is a descendant of v in a DFS-tree of T.
  - Cross edge: if neither of the above applies.



DFS Forest



- Forward edge:
  - (a,b),(a,d),(b,c),(c,d),(c,e),(d,g),(g,f),(g,i),(f,e)
- Backward edge: (e,d)
- Cross edge: (i,f),(h,d),(h,g)

- I(a)=[1,16], I(b)=[2,15], I(c)=[3,14]
- I(d)=[4,13], I(g)=[5,12], I(f)=[6,9]
- I(e)=[7,8], I(i)=[10,11],I(h)=[17,18]

O(1)确定edge(u,v)的类型

- Maintain a counter c, which is initially 0. Every time a push or pop is performed on the stack, we increment c by 1.
- For every vertex v, define:
  - Its discovery time d-tm(v) to be the value of c right after v is pushed into the stack
  - Its finish time f-tm(v) to be the value of c right after v is popped from the stack
  - Define I(v) = [d-time(v), f-tm(v)]
- It is straight forward to obtain I(v) for all $v \in V$ by paying O(|V|) extra time on top of DFS's running time.

设定一个计数器 c=0. 每当有节点a被push或pop出stack中时, c++.

对每个节点v, 有:

d-tm(v): v被push入stack时的c值.

f-tm(v): v被pop出stack时的c值

定义 I(v) = [d-tm(v) f-tm(v)]

Theorems

- **Parenthesis Theorem:** all the following are true:
  - If u is a proper ancestor of v in DFS-tree of T, then I(u) contains I(v).
  - If u is a proper descendant of v in DFS-tree of T, then I(u) is contained in I(v).
  - Otherwise, I(u) and I(v) are disjoint.
- Proof: Follows directly from the first-in-last-out property of the stack.
- **Cycle Theorem:** let T be an arbitrary DFS-forest. G contains a cycle if and only if there is a backward edge with respect to T.
- Proof: will left as exercise.

u是v的真祖先 $\Leftrightarrow$ $I(v) \in I(u)$

u是v的真后代 $\Leftrightarrow$ $I(u) \in I(v)$

否则, I(u)与I(v)不相交.

G有环 $\Leftrightarrow$ T中有backward edge.

Cycle Detection

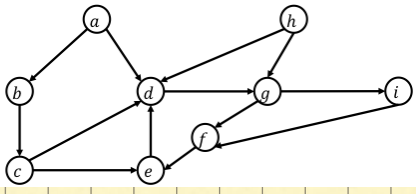
- Equipped with the cycle theorem, we know that we can detect whether G has a cycle easily after having obtained a DFS-forest T:
  - For every edge (u,v), determine whether it is a backward edge in O(1) time.
- If no backward edges are found, decide G to be a DAG; otherwise, G has at least a cycle.
- Only O(|E|) extra time is needed
- We now conclude that the cycle detection problem can be solved in O(|V|+|E|) time.

利用 cycle Theorem.

Topological Sort

- Let G=(V,E) be a directed acyclic graph (DAG).
- A topological order of G is an ordering of the vertices in V such that, for any edge (u,v), it must hold that u precedes v in the ordering.
- Example: two possible topological orders:
  - h, a, b, c, d, g, i, f, e
  - a, h, b, c, d, g, i, f, e
- a, h, d, b, c, g, i, f, e

is not topological order, because of edge (c,d).



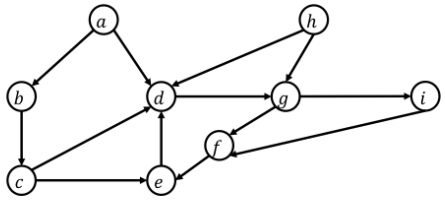
拓扑序是对任意边(u,v), u都是v的前趋.
拓扑排序的第一个节点必须没有进边,
最后一个节点必须没有出边.

拓扑序不只一个.
(因为源节点不一定一样)

- Let G=(V,E) be a directed acyclic graph (DAG). The goal of topological sort is to produce a topological order of G.
- Topological Sort Algorithm
  - Create an empty list L
  - Run DFS on G, whenever a vertex v turns red (i.e., it is popped from the stack), append it to L.
  - Output the reverse order of L
- The total running time is clearly O(|V|+|E|)

每一个DAG都有topological order

创建空列表L
对G跑DFS. 当有节点v出栈时, 加在L末尾
最终L的逆序即拓扑序.



- Suppose we run DFS starting from a. The following is one possible order by which the vertices turn red:
  - e, f, i, g, d, c, b, a, h
- Therefore, we output h, a, b, c, d, g, i, f, e as a topological order.
- Suppose we run DFS starting from d, then restarting from h, then from a. The following is one possible order by which the vertices turn red:
  - e, f, i, g, d, h, c, b, a
- Therefore, we output a, b, c, h, d, g, i, f, e as a topological order.

63

Correctness Analysis

- We now prove that the algorithm is correct.
- Proof. Take any edge (u,v). We will show that u turns red after v, which will complete the proof.
  - Consider the moment when u enters the stack, We argue that that currently v cannot be in the stack. Suppose that v was in the stack. As there must be a path chaining up all the vertices in the stack bottom up, we know that there is a path from v to u. Then, adding the edge (u,v) forms a cycle, contradicting the fact that G is a DAG.
  - v is red at this moment then obviously u will turn red after v.
  - v is white: then by the white path theorem of DFS, we know that v will become a proper descendant of u in the DFS-forest. Therefore, u will turn red after v.
- Every DAG has a topological order!

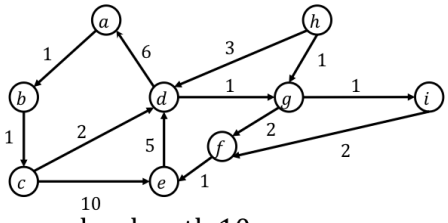
64

Shortest Path (最短路径问题)

- Single source shortest path (SSSP)
  - BFS algorithm
  - All the edges have the same weight
- SSSP with arbitrary positive path (SP)
- Weight graph
  - Let G=(V,E) be a directed graph. Let w be a function that maps each edge in E to a positive integer value. Specifically, for each e $\in E$, w(e) is a positive integer value, which we call the weight of e.
  - A directed weighted graph is defined as the pair (G,w).
- Consider a directed weighted graph defined by a directed graph G=(V,E) and function w.
- Consider a path in G: (v_1, v_2), (v_2, v_3), ..., (v_l, v_{l+1}), for some integer $l \geq 1$. We define the length of the path as: $\sum_{i=1}^{l} w(v_i, v_{i+1})$.
- Recall that we may also denote the path as: $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{l+1}$.
- Give two vertices $u, v \in V$, a shortest path from u to v is a path from u to v that has the minimum length among all the paths from u to v.
- If v is unreachable from u, then the shortest path distance from u to v is ∞.

SSSP with Positive Weights

- Let (G,w) with G=(V, E) be a directed weighted graph, where w maps every edge of E to a positive value.
- Give a vertex s in V, the goal of the SSSP problem is to find, for every other vertex $t \in V \setminus \{s\}$, a shortest path from s to t, unless t is unreachable from s.
- A subsequence property
  - Lemma: if $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{l+1}$ is a shortest path from v_1 to v_{l+1}, then for every i, j satisfying $1 \leq i \leq j \leq l+1$, $v_i \rightarrow v_{i+1} \rightarrow \cdots \rightarrow v_j$ is shortest path from v_i to v_j.
  - Proof: suppose that this is not true, then we can find a shorter path from v_i to v_j. Using that path to replace the original path from v_1 to v_{l+1}, which contradicts the fact that $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{l+1}$ is a shortest path.



- The path $c \rightarrow e$ has length 10
- The path $c \rightarrow d \rightarrow g \rightarrow f \rightarrow e$ has length 6
- The second path is the shortest path from c to e
- We know that any subsequence of this path is also a shortest path. For example, $c \rightarrow d \rightarrow g \rightarrow f$ must be a shortest path from c to f.

Dijkstra's Algorithm

edge relaxation

- The edge relaxation idea
  - For every vertex $v \in V$, we will maintain a value dist(v) that represents the length of the shortest path from s to v found so far.
  - At the end of the algorithm, we will ensure that every dist(v) equal to the precise shortest path from s to v
  - A core operation in our algorithm is called edge relaxation. Given an edge (u,v), we relax it as follows:
    - If dist(v) < dist(u) + w(u,v), do nothing
    - Otherwise, reduce dist(v) to dist(u) + w(u,v)

71

对每个节点v，都有到节源点s的最短距离dist(v)

对某条边edge(u,v)，若dist(v) > dist(u) + w(u,v)就令dist(v)等于dist(u)+w(u,v)

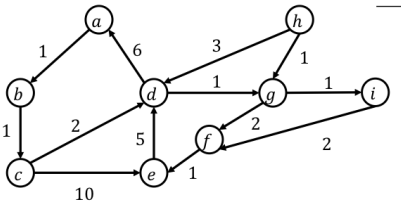
- Set parent(v) = nil for all vertices $v \in V$
- Set dist(s) =0 and dist(v)= ∞ for all other vertices $v \in V$
- Set S = V
- Repeat the following until S is empty
  - Remove from S the vertex u with the smallest dist(u).
    /* next we relax all the outgoing edges of u*/
  - For every outgoing edge (u,v) of u
  - If dist(v) > dist(u) + w(u,v) then
    - Set dist(v) = dist(u) + w(u,v), and parent (v)=u

初始化时将所有节点v的父节点parent(v)设为null.
对源点s设dist(s)=0，其他点dist(v)=∞.
设S=V.

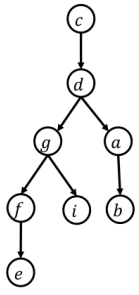
```
while(!S.isEmpty())
    u = smallest dist(u) in S
    for out edge(u,v) of u
        if dist(v) > dist(u)+w(u,v)
            dist(v) = dist(u)+w(u,v)
            parent(v) = u.
        endif
    endfor
endwhile
```

Constructing the SP tree

- For every vertex v, if u = parent(v) is not nil, the make v a child of u.



Vertex v	parent(v)
a	d
b	a
c	nil
d	c
e	f
f	g
g	d
h	nil
i	g



Correctness and Running Time

- It will be left as an exercise for you to prove that Dijkstra's algorithm is correct
- Just as equally instructive is an exercise for you to implement Dijkstra's algorithm in O((|V|+|E|)*log|V|) time. Why?
- You have already learned all the data structure for this purpose. Now it is time to practice using them.

时间复杂度：$O((|V|+|E|)*\log|V|)$

$$\sum_{u \in V}[\log|V| + d^+(u)\cdot 2\log|V|)$$

$$= \log|V| \sum_{u \in V}(1+2d^+(u))$$

$$= \log|V| * (|V|+2|E|)$$

Minimum Spanning tree (MST)

Undirected Weighted Graph

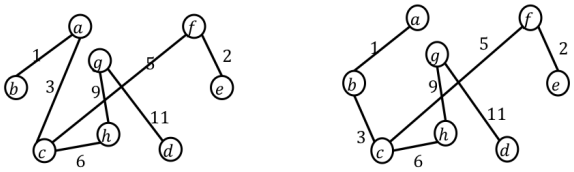
- Let G=(V, E) be an undirected graph. Let w be a function that maps each edge of G to a positive integer value. Specifically, for each edge e, w(e) is a positive integer value, which we call the weight of e.
- An undirected weighted graph is defined as the pair (G,w)
- We will denote an edge between vertices u and v in G as {u,v}, instead of (u,v), to emphasize that the ordering of u, v does not matter
- We consider that G is connected, namely, there is a path between any two vertices in V.

Spanning Tree : connected. undirected graph without cycles

- Given a connected undirected weighted graph (G,w) with G=(V,E), a spanning tree T is a tree satisfying the following conditions:
  - The vertex set of T is V.
  - Every edge of T is an edge of G.
- The cost of T is defined as the sum of the weights of all the edges in T (note that T must have |V|-1 edges)

Minimum Spanning Tree

- The minimum spanning tree problem
- Given a connected undirected weighted graph (G,w) with G=(V,E), the goal of the minimum spanning tree (MST) problem is to find a spanning tree of the smallest cost.
- Such a tree is called an MST of (G, w)



- Both trees are MSTs. This means that MSTs may not be unique.

给定一个连通的无向有权图 (G=(V,E),w). 最小生成树就是建树时花费最小.

Prim's Algorithm

- The algorithm grows a tree T_{mst} by including one vertex at a time, at any moment, it divides the vertex set V into two parts:
  - The set S of vertices that are already in T_{mst}
  - The set of other vertices: V \ S
- at the end of the algorithm, S = V
- If an edge connects a vertex in S and a vertex in V \ S, we call it an extension edge.
- At all times, the algorithm enforces the following lightest extension principle:
  - For every vertex v ∈ V \ S, it remembers which extension edge of v has the smallest weight, referred to as the lightest extension edge of v, and denoted as *best-ext(v).*

生成 T_{mst} 时每次加入一个新节点, 最终全部节点都加入 T_{mst} 中.

对某边, 若其一端在S中, 另一端在V\S中, 则该边被称为外延边.

外延边中有最轻权值的记为 best-ext(v).

- 1. Let {u,v} be an edge with the smallest weight among all edges
- 2. Set S={u,v}. Initialize a tree T_{mst} with only one edge {u,v}.
- 3. Enforce the lightest extension principle:
  - For every vertex z of V \ S
    - If z is a neighbor of u, but not of v
      - *best-ext(z)* = edge {z, u}
    - If z is a neighbor of v, but not of u
      - *best-ext(z)* = edge {z, v}
    - If z is a neighbor of both u and v
      - *best-ext(z)* = the lighter edge between {z, u} and {z, v}
- 4. Repeat the following until S = V:
  - 5. Get an extension edge of {u, v} with the smallest weight

/* Without loss of generality, suppose u ∈ S, and */

  - 6. Add v to S, and add edge {u, v} into T_{mst}

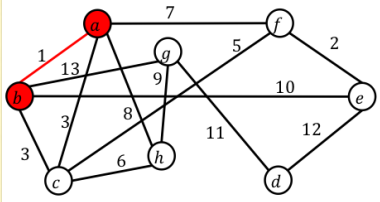
/* Next, we restore the lightest extension principle. */

  - For every edge {v, z} of v:
    - If z ∉ S then
      - If best-ext(z) is heavier than edge {v, z} then
        - Set best-ext(z) = edge {v, z}

以最轻权值边(u,v)开始，令S={u,v}，Tmst中含有一边(u,v)。

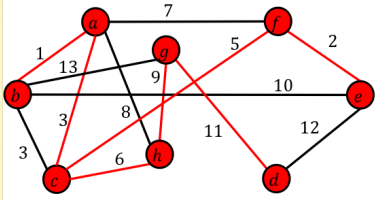
```
for z in V\S
  if z in u.neighbour but not of v:
     best-ext(z) = edge(z,u)
  if z in v.neighbour but not of u:
     best-ext(z) = edge(z,v)
  if z in u.neighbour && z in v.neighbour:
     best-ext(z) = min_edge(w(z,u), w(z,v)).

while(S!=V)
  得到S中有最小权值的外延边(k,w).
  S.add(w); Tmst.add(|k,w|).
  for every edge (w,z) of w:
    if z ∉ S:
      if best-ext(z) > w(w,z):
        best-ext(z)=(w,z)
      endif
    endif
  end for
endwhile
```

- Edge {a,b} is the lightest of all. So, at the beginning S = {a, b}. The MST we are growing now has one edge {a,b}



Vertex v	best-ext(v) and weight
a	n/a
b	n/a
c	{c,a}, 3
d	nil, ∞
e	{e,b}, 10
f	{a,f}, 7
g	{g,b}, 13
h	{a,h}, 8

- Note: edge {c,a} and {c,b} have the same weight. Either of them can be best-ext(c).
- Finally, edge {d,g} is the lightest extension edge. So, we add d to S, which now S = {a,b,c,f,e,h,g,d}, add edge {d,g} into MST



Vertex v	best-ext(v) and weight
a	n/a
b	n/a
c	n/a
d	n/a
e	n/a
f	n/a
g	n/a
h	n/a

- We have obtained our final MST.

Time Complexity Analysis

- A priority queue Q (min-heap) was employed in Prim's algorithm, what is the key of node in Q?
- Line 1 & 2: O(1)
- Line 3: O(|E|)
- Line 4: O(|V|)
- Line 5: O(|V| log |V|)
- Line 6: O(|V|)
- Line 7: O(|E| log |V|), Total: O((|V|+|E|) log |V|)
- Remark: Using the Fibonacci Heap, will not cover in this course, we can improve the running time to O(|V| log |V| + |E|)

Correctness Proof

- **Claim:** For any i ∈ [1, |V|-1] , there must be an MST containing all the first i edges chosen by the algorithm
- Then the algorithm's correctness follows from the above claim at i = |V|-1
- We prove it by induction the sequence of the edges added to the tree
- Base case: i=1, let {u,v} be the edge with the smallest weight in the graph, the edge must exist in some MST
- Inductive case: the claim holds for i<=k-1
- We prove it also hold for i=k

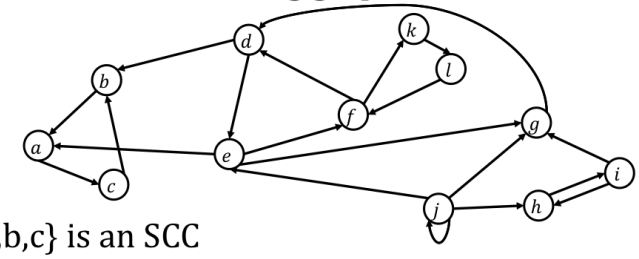
Strongly Connected Components (SCC)

- Let G=(V,E) be a directed graph.
- A strongly connected component (SCC) of G is a subset S of V such that:
  - For any two vertices u, v ∈ S, it must hold that:
    - There is a path from u to v
    - There is a path from v to u
  - S is maximal in the sense that we cannot put any more vertex into S without violating the above property
- It seems to be rather difficult at first glance, the algorithm is once again very simple, run DFS only twice.

给定一个有向图 G=(V,E).
强连通分量即是 S⊆V，S中的每个节点之间都有路径，且新加任何节点都会破坏这个性质.

只需跑两次DFS.

- Consider the following graph:



- {a,b,c} is an SCC
- {a,b,c,d} is not an SCC
- {d,e,f,k,l} is not an SCC (why?)
- {e,d,f,k,l,g} is an SCC

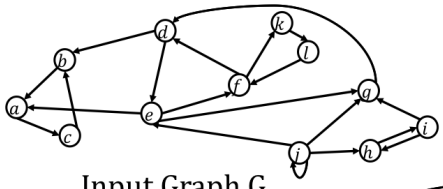
SCCs are Disjoint

- Theorem: Suppose that S_1 and S_2 are both SCCs of G, Then $S_1 \cap S_2 = \emptyset$
- Proof: Assume that there is a vertex v in both S_1 and S_2 . Then, for any vertex $u_1 \in S_1$ and any vertex $u_2 \in S_2$:
  - There is a path from u_1 to u_2 : we can first go from u_1 to v within S_1 , and then from v to u_2 within S_2 .
  - Likewise, there is also a path from u_2 to u_1 .

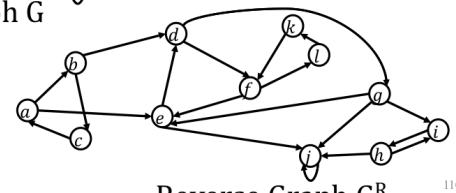
  Hence, neither S_1 and S_2 is maximal, contradicting the fact that they are SCCs.

Finding SCCs Algorithm

- Step 1: obtain the reverse graph G^R by reversing the directions of all the edges in G.



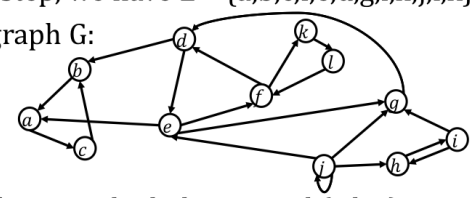
Input Graph G



Reverse Graph G^R

- Step 2: Perform DFS on G^R , and obtain the sequence L^R that the vertices in G^R turn red (i.e., whenever a vertex is popped out of the stack, append it to L^R)
- Obtain L as the reverse order of L^R
- We may perform DFS starting from any vertex. The following is a possible order that the vertices are discovered: f,l,k,e,j,d,g,i,h,a,b,c
- The corresponding turn-red sequence is
- L^R = {k,l,j,h,i,g,d,e,f,c,b,a}
- Hence L = {a,b,c,f,e,d,g,i,h,j,l,k}
- Step 3: Perform DFS on the original graph G by obeying the following rules:
  - Rule 1: start the DFS at the first vertex of L
  - Rule 2: whenever a restart is needed, start from the first vertex of L that is still white.
- Output the vertices in each DFS-tree as an SCC

- From the last step, we have L = {a,b,c,f,e,d,g,i,h,j,l,k}
- The original graph G:



- Starting DFS from a, which discovered {a,b,c}
- Restart from f, which discovered {f,k,l,d,e,g}
- Restart from i, which discovered {i,h}
- Restart from j, which discovered {j}
- The DFS returns 4 DFS-tree, whose vertex sets are as above, Each vertex set constitutes an SCC.

Running Time Analysis

- Steps 1 and 2 obviously require only O(|V|+|E|) time.
- Regarding Step 3, the DFS itself takes O(|V|+|E|), but how about the cost of implement Rule 2.
- Namely, whenever, DFS needs a restart, how do we find the first white vertex in L efficiently?
- It can be done in O(|V|) total time.
- Hence, the overall execution time is O(|V|+|E|)

Correctness Proof

- Let G be the input directed graph, with SCCs S_1, S_2, ..., S_t for some t >= 1
- Let us define a SCC graph G^{SCC} as follows:
  - Each vertex in G^{SCC} is a distinct SCC in G.
  - Consider two vertices S_i and S_j , G^{SCC} has an edge from S_i to S_j if and only if:
    - i !=j
    - There is a path in G from a vertex in S_i to a vertex in S_j
- G^{SCC} is a DAG, define an SCC as a sink SCC if it has no outgoing edge in G^{SCC}
- Lemma: There must be at least one sink SCC in G^{SCC}

- Let S be a sink SCC in G^{SCC} . Suppose that we perform a DFS starting from any vertex in S. Then the first DFS-tree output must include all and only the vertex in S.
- Finding SCC: The strategy
  - 1. Performing DFS from any vertex in a sink SCC S
  - 2. Delete all vertices of S from G, as well as their edges
  - 3. Accordingly, delete S from G^{SCC} , as well as its edges.
  - 4. Repeat from Step 1, until G is empty.
- Lemma: Let S_1, S_2 be SCCs such that there is a path from S_1 to S_2 in G^{SCC}. In the ordering of L, the earliest vertex in S_2 must come before the earliest vertex in S_1