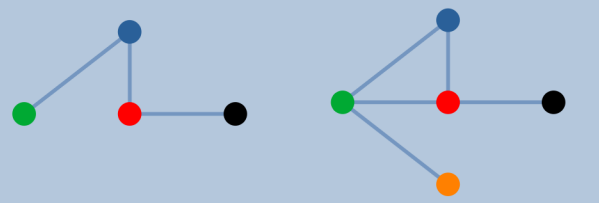
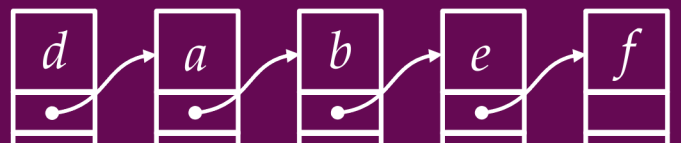
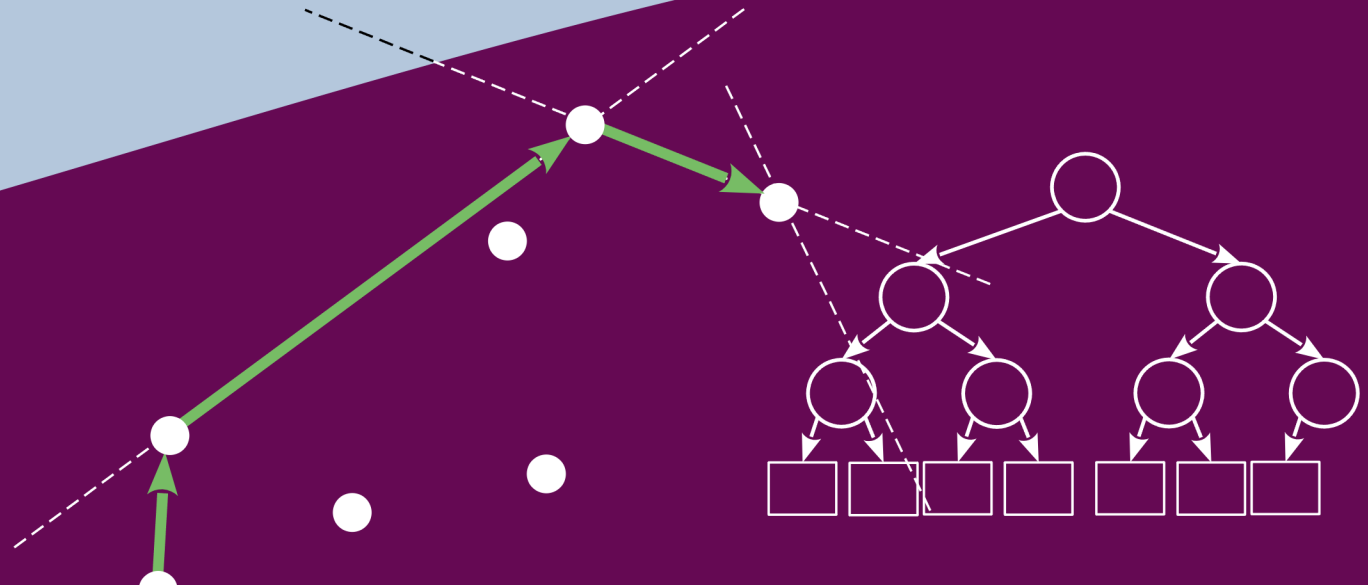
Advanced algorithms and structures data

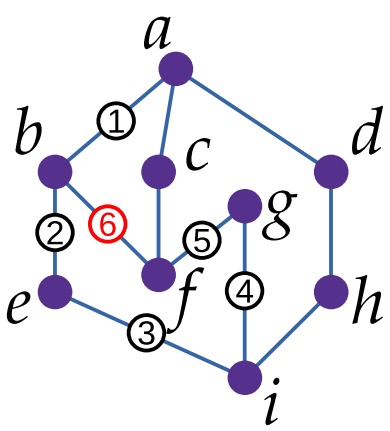
Week 10: Algorithms over graphs Cycle detection, Min. spanning graphs, Euler graphs







Cycle detection



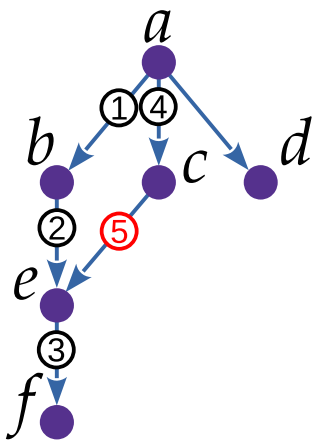
Cycle detection by DFS

- Cycle detection is an essential task for many graph algorithms: say algorithms for finding a minimum spanning tree or for detecting Hamiltonian cycles or Euler circles
- The simplest way is to use DFS or BFS to traverse the graph
  - Let's mark each peak we visit
  - If in the tour we come across an already marked peak, then we have detected a cycle in the graph

Alternative literature:

- **Thomas H, Cormen, et al. "Introduction to Algorithms." (2016). Introduction to Algorithms., Chapter VI**

Cycle detection

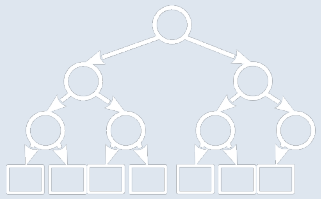


- This approach has a big drawback (say for DFS)
  - After returning from the recursion, we start descending through the graph again
  - We did not erase the marks that we visited some of the peaks
  - If in that descent we encounter a marked peak - have we detected a cycle?
  - We're never really sure
  - If we were to remove labels, this could cause a problem with finding cycles in disconnected graphs
- The solution is to introduce a stack with which we track our current path
  - When we recurse down through the graph, we put the vertices on the stack
  - When we return from the recursion, we remove the vertices from the stack
  - We know that we have returned to an already visited peak if it is on the stack

Cycle detection

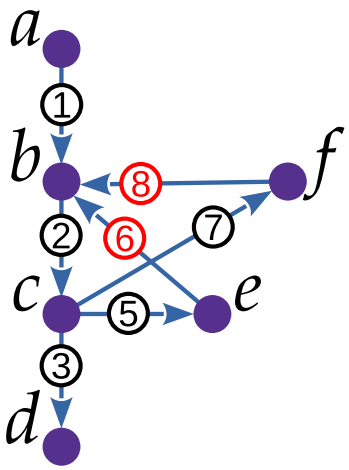
```
procedure FindCycle(G)
    initialize all vertices in G as not visited
    S ← empty stack
    while there is an unvisited vertex u_0 in G do
        FindCycle_recursive(G, u_0, S)

procedure FindCycle_recursive(G, u, S)
    mark u as visited
    S.push(u)
    for v in adjacent vertices of u do
        if v not in S then
            set predecessor of v to u
            FindCycle_recursive(G, v, S)
        else
            if predecessor of u is not v then
                initialize cycle c ← {}
                repeat
                    retrieve vertex vx backwards from the stack S
                                                ▷ Do not pop edges
                    add vertex vx to the cycle c
                until vx = v
                report the cycle c
    S.pop()
```



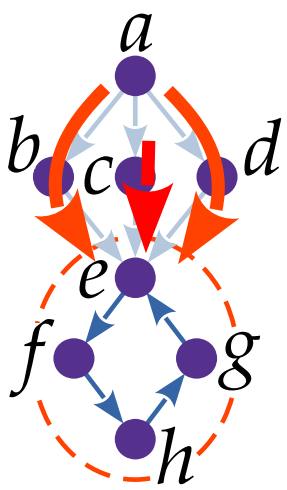
- At the same time, we use two mechanisms: we mark vertices as visited and we have a stack with which we follow the current trajectory
  - By marking vertices, we avoid having to process the same partition of vertices more than once in a disconnected graph
- By entering the recursive call, we put the top on the stack ,and when leaving, we move it from the stack
  - We take care not to return to the previous vertex in the current path in undirected graphs
  - At the moment when we see that the top on the stack , we count the cycle - all vertices on the stack backwards until the previous occurrence of the vertices
    - When calculating the cycles, we do not move the vertices from the stack to allow further progression of the algorithm

Cycle detection - example



Iteration	1	2	3	4	5	6	7	8
vertex u	a	b	c	d	c	e	c	f
vertex v	b	c	d		e	b	f	b
stack S	a	b a	c b a	d c b a	c b a	e c b a	c b a	f c b a
cycle c						e, c, b		f, c, b

Cycle detection

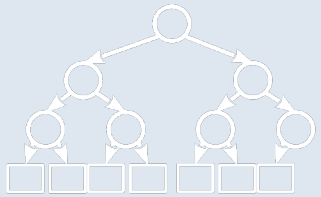


- This approach is not without problems either
  - Given that we only follow the current path, it is possible to detect one and the same cycle more than once
  - Let's say, the first trajectory in the exposed example is h
    - The last edge reveals the cycle h
  - We return from the recursion until
  - We enter the recursion again and fill the stack, up to h
    - With the last edge, we rediscover the cycle h

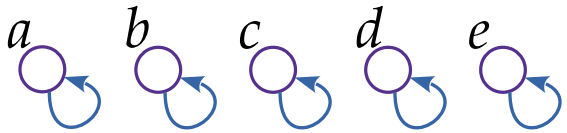
- The method is essentially a slightly modified DFS, thus increasing the complexity of the algorithm (+)

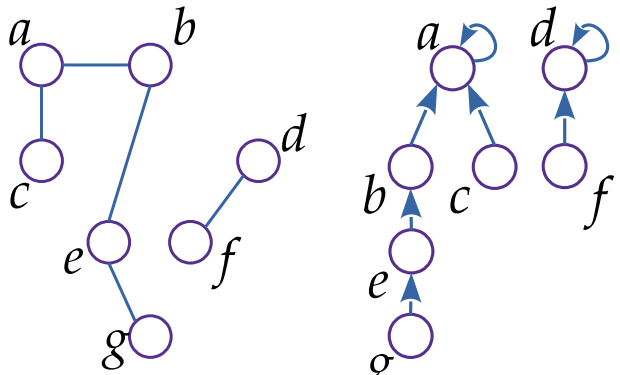
Union-Find

- It uses notation of partitions of an undirected graph to avoid cycles
- One of the simpler methods is to use graphs as a representation
- We call each graph a tree of a disconnected set(disjoint-set tree)
  - Each vertex of the tree represents a vertex of the graph
  - The top of the tree points to the parent with a pointed edge
  - The root tip in the tree points to itself
- The set of all trees of disjoint sets is called a disjoint forest meetings(disjoint-set forest)
- Let's imagine we have a graph = (,)and to process it edge by edge – which often happens in algorithms that use cycle detection

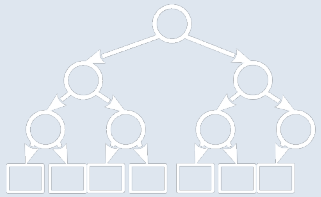


Union-Find

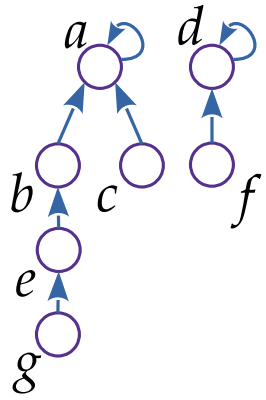


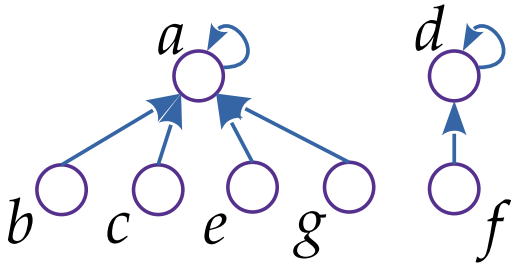


- At the beginning, we have a tree of the disconnected set for each vertex of the graph
  - Each tree obviously has one tip which is the root and points to itself
  - A forest of disconnected sets has as many trees as there are vertices in the graph
- As we get the edges of the graph, we have two situations:
  1. A new edge connects vertices located in two different trees of disjoint sets - means that the edge does not form a cycle
     - We return to the calling algorithm that the edge does not form a cycle
     - We join the two trees according to the edge we got, so that one top depends on the other
  2. A new edge connects vertices in the same tree of a disjoint set – means that the edge forms a cycle
     - We return to the calling algorithm that the edge forms a cycle

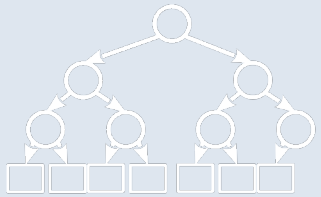


Union-Find





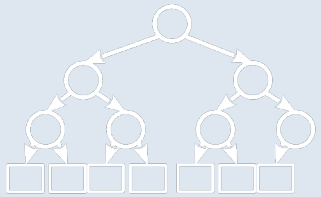
- Trees of disjoint sets can be collapsed, to allow the shortest possible search for a vertex in the tree
- By compressing, we reduce the tree search to O(1), and do not impair the functionality at all
- Merging summary sets takes more time, since restructuring of these merging trees is done



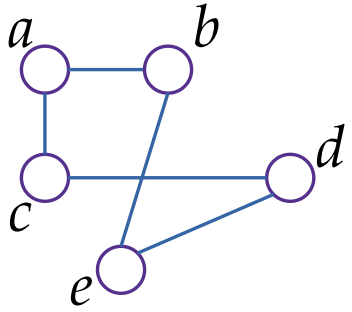
Union-Find

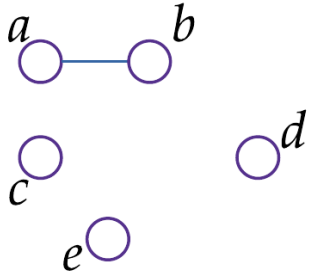
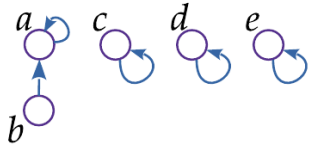
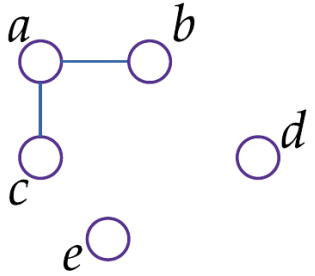
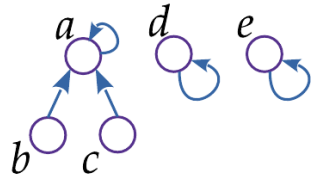
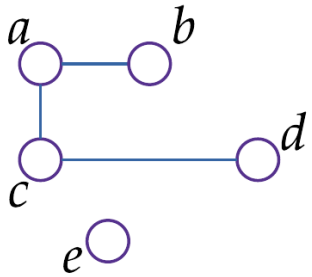
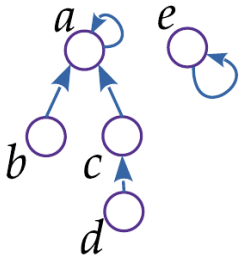
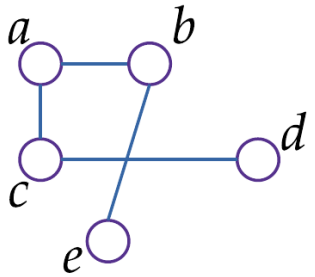
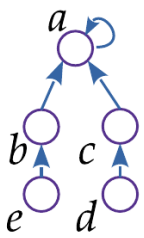
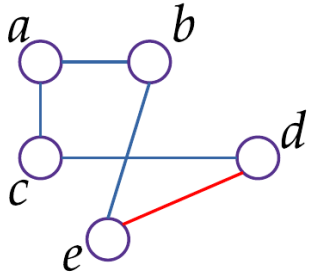
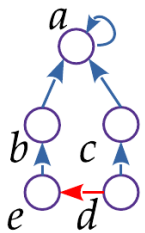
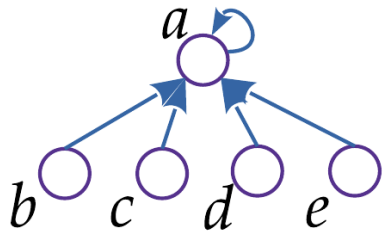
```
function MakeSet(F, x)
    if x not in F then
        x.parent = x
        add tree x to the forest F
    return F
function Find(x)
    while x ≠ x.parent do
        x.parent = x.parent.parent          ▷ The compression
        x = x.parent
    return x
procedure Union(x, y)
    x = Find(x)
    y = Find(y)
    if x ≠ y then
        x.parent = y
```

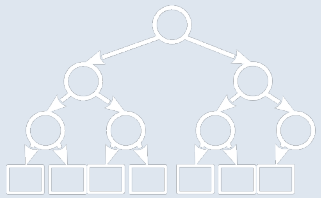
- **MakeSet**–a procedure that in a forest of unrelated sets adds the top who is his own parent
- **Find**–a procedure that searches for the parent of the vertex –at the same time it does tree compaction
- **Union**–procedure that checks if vertices and in the same tree, so if they are not, it puts them in the same tree
- A simple check whether the peaks are in the same cycle can be seen in the procedure**Union**



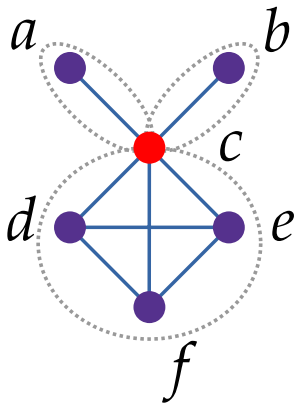
Union-Find

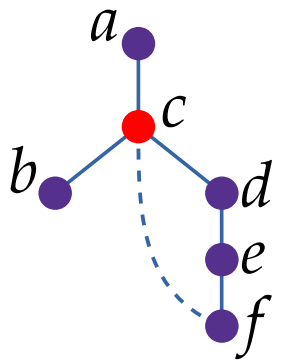


Iteration 1: Edge *ab*		**Iteration 2:** Edge *ac*	
			
Iteration 3: Edge *cd*		**Iteration 4:** Edge *be*	
			
Iteration 5: Edge *de* - **cycle detected**		**The final compressed disjoint-set forest**	
			



Detection of blocks in the graph





- It is done in undirected graphs
- Block(biconnected component)is a subgraph ′graph which does not contain breaking points(articulation points). This means that if we remove any vertex from the block, the block still remains connected.
- Finding breakpoints in a graph is the basis for block detection - the problem is similar to cycle detection
  - Blocks are obtained by interrupting the graph at a turning point
  - For example, we have blocks*a.c*,*b.c*,*cdef*
- Using DFS we search for cycles in the graph
  - When we find the cycle, the root tip represents the breaking point
- For example, we see a cycle*cdefc*, where is*c*the root tip, and thus the breaking point

Detection of blocks in the graph

- Block detection is performed on a spanning tree derived from the extended definition of a graph, which we define as an ordered triple

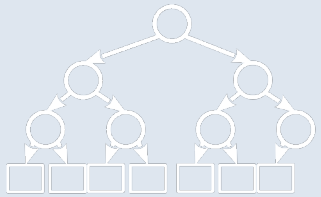
$$= (, ,)$$

  - where is mapping function

$$: \rightarrow \mathbb{N}\times\mathbb{N}$$

  - which maps each vertex of the graph to an ordered pair (,),where
    - represents the number of the vertex in the spanning tree
    - represents the highest predecessor of that peak - obviously a candidate for the turning point
- We define basic rules on such a graph
  - No two vertices in the graph have the same number

$$\nexists , \in : \neq , \quad = (())$$



Detection of blocks in the graph

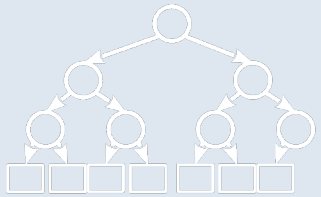
- We define basic rules on such a graph
  - The predecessor of the vertex v is defined

$$\in m)\min(\quad,\quad !,(\quad '),\ldots,(\quad(\quad \#)))\quad(\quad\quad)$$

    - where are they !, ",..., #the vertices to which descendents from the subtree of the vertex return
    - wanted**minimal**the common ancestor of the entire subtree of the vertex
  - Defined differently

$$\in m)\min(\quad,\quad !,(\quad '),\ldots,(\quad(\quad \#)))\quad(\quad\quad)$$

    - where are they !, ",..., #descendants in the vertex subtree
    - **minimal**common ancestor of the peak is either the top itself ,or the minimal predecessor in the vertex's subtree

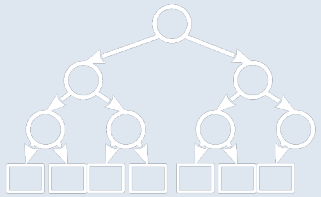


Detection of blocks in the graph

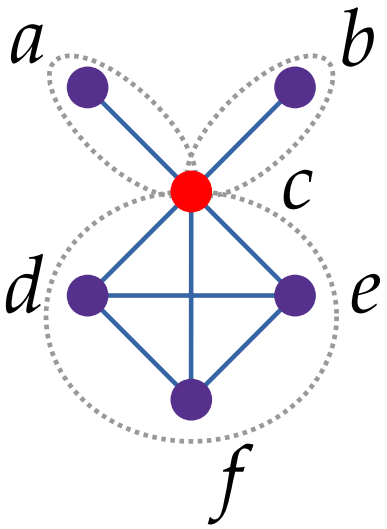
```
procedure BlockSearch(G)
    initialize all vertices in G as n(v) = 0
    step ← 1
    S ← empty stack
    while there is a vertex u in G, having n(u) = 0 do
        BlockSearch_recursive(G, u, step, S)

procedure BlockSearch_recursive(G, u, step, S)
    p(u) = n(u) = step
    step ← step + 1
    for v in adjacent vertices of u do
        if n(v) = 0 then
            if not edge uv on the stack S then
                S.push(uv)
            BlockSearch_recursive(G, v, step, S)
            if p(v) ≥ n(u) then
                pop edges until uv and form a block
            else
                p(u) = min(p(u), p(v))
        else if S is not empty and vu is not the last element then
            p(u) = min(p(u), n(v))
```

- We initialize each vertex uniquely ( ) = ( )
- We are moving towards the neighboring peaks of
  - We put the edge on the stack
  - If the adjacent vertex new, that is ( ) = 0, then we recursively call its processing
    - When returning from recursion, we check whether it is its predecessor or some descendants of
    - If it is, we form a block with a stack on which we put the edges
    - If not, we update the minimal predecessor
  - If we are an adjacent vertex already visited, it is possible that it is a return edge
    - We update the minimal predecessor of the vertex ,and which can be the top



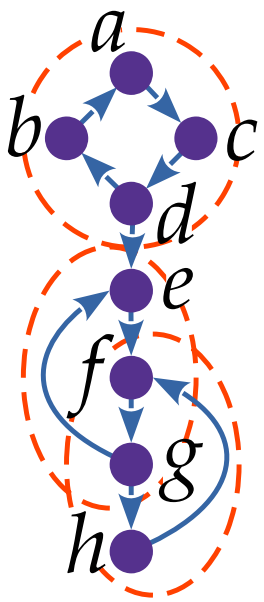
Detection of blocks in the graph



	0	1	2	3	4	5	6	7	8	9	10	11	12
u		a	c	b	c	c	d	e	f	e	d	c	a
v		c	b		b	d	e	f	c	f	e	d	c
	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p
a	0/0	1/1											pop
b	0/0			3/3									
c	0/0		2/2		pop							pop	
d	0/0						4/4				4/2		
e	0/0							5/5		5/2			
f	0/0								6/6 6/2				
stack S		ac	cb ac	cb ac	ac	cd ac	de cd ac	ef de cd ac	ef de cd ac	ef de cd ac	ef de cd ac	ac	

!= = [, "{ =}], , #= [= , { , }] [{ }]

Detection of components in a directed graph



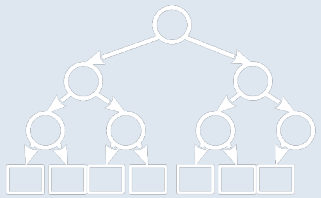
- The definition of a tightly connected component (SCC) implies that all pairs of vertices in that component are mutually accessible
- A naive attempt to solve this problem is NP-complex because we should create partitions of vertices in which all vertices are mutually reachable
- The problem can be simplified by knowing that cycles form such components
  - For example, we have three cycles: 1 =*acdba*, 2 =*efge*and 3 =*fghf*
  - The two cycles share common vertices and edges, which gives us two tightly connected components: 1and 2 ∪ 3
- Blocks in undirected graphs share a breakpoint, while tightly connected components in directed graphs cannot share vertices

Tarjan's Algorithm (SCC)

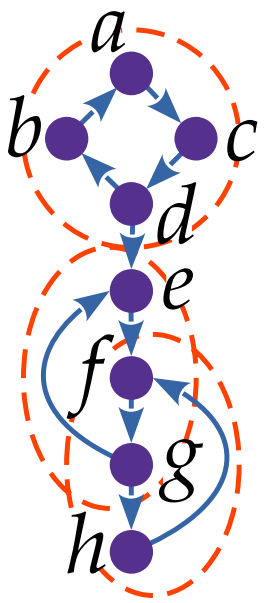
```
procedure SCCSearch(G)
    initialize all vertices in G as n(v) = 0
    step ← 1
    S ← empty stack
    while there is a vertex u in G, having n(u) = 0 do
        SCCSearch_recursive(G, u, step, S)

procedure SCCSearch_recursive(G, u, step, S)
    p(u) = n(u) = step
    step ← step + 1
    S.push(u)
    for v in adjacent vertices of u do
        if n(v) = 0 then
            SCCSearch_recursive(G, v, step, S)
            p(u) = min(p(u), p(v))
        else if n(v) < n(u) and v is on the stack S then
            p(u) = min(p(u), n(v))
    if p(u) = n(u) then                    ▷ this is the SCC root vertex
        pop vertices from the stack S until u is popped off
```

- We initialize each vertex uniquely ( ) = ( )
- We are moving towards the neighboring peaks of
  - We put the top on the stack
  - If the adjacent vertex new, that is ( ) = 0,then we recursively call its processing
    - When returning from recursion, we update the minimal predecessor of the vertex
  - If we are an adjacent vertex already visited, it is possible that it is a return edge
    - We update the minimal predecessor of the vertex ,and which can be the top
  - When returning back to the caller, we test the vertex's predecessor .If the top has no predecessor so it is ( ) < ( ),then the top we consider the root of the component
    - This step is slightly different from the block detection algorithm



Tarjan's Algorithm (SCC)



u / v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
		a	*c*	*d*	*b*	*d*	*d*	*e*	*f*	*g*	*g*	*h*	*g*	*f*	*e*	*d*	*c*	*a*
	n/p	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*	*n/p*
a	0/0	1/1																pop
b	0/0				4/4 4/1													
c	0/0		2/2														2/1	
d	0/0			3/3		3/1												
e	0/0							5/5							pop			
f	0/0								6/6					6/5				
g	0/0									7/7 7/5								
h	0/0											8/8 8/6						
stack S		*a*	*c* *a*	*d* *c* *a*	*b* *d* *c* *a*	*b* *d* *c* *a*	*b* *d* *c* *a*	*e* *b* *d* *c* *a*	*f* *e* *b* *d* *c* *a*	*g* *f* *e* *b* *d* *c* *a*	*g* *f* *e* *b* *d* *c* *a*	*h* *g* *f* *e* *b* *d* *c* *a*	*h* *g* *f* *e* *b* *d* *c* *a*	*h* *g* *f* *e* *b* *d* *c* *a*	*h* *g* *f* *e* *b* *d* *c* *a*	*b* *d* *c* *a*	*b* *d* *c* *a*	*b* *d* *c* *a*

!= [= {*h*, , , }] "= [= { , , , }]

Minimum spanning tree (MST)

- Previously, we defined a spanning tree as the product of graph traversal with various algorithms, such as BFS and DFS
  - Because of this diverse approach, a graph can have many different spanning trees
  - For the weight graph = (, ,)thus we can define the set of all spanning trees

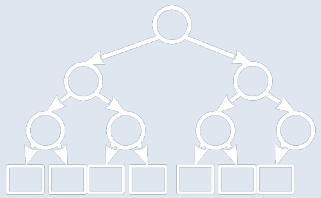
$$( ) = \{ " \quad != , (!, \ : \ !\subseteq \ )( )\}$$

  - Finding the minimum spanning tree is the result of optimization

$$= ar(g\ ) \ \#\% \qquad \$\in\%\&(\#)> \qquad ( )$$

$$) \in *\$(\#\$)^{\%}$$

  - We are looking for a spanning tree whose sum of edge weights is minimal in the set of all spanning trees
- All algorithms used for this purpose are greedy algorithms

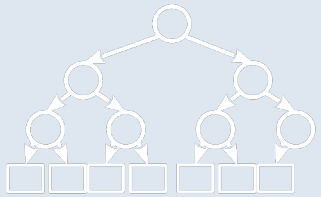


Kruskal's algorithm

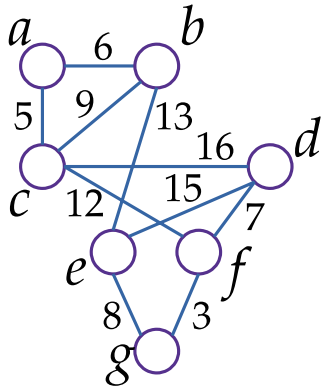
```
procedure KRUSKAL(G)
    MST ← (V = V(G), E = ∅)
    sort edges E(G) ascending by weights
    for e_i ∈ E(G) do
        if |E(MST)| < |V(G)| − 1 then
            if no cycle in G' = (V(MST), E(MST) ∪ {e_i}) then
                add e_i to E(MST)
    return MST
```

- Search complexity summarized *Union-Find* method is (1)
- Due to sorting, the complexity of Kruskal's algorithm is log(.)
- Given that is in a complete undirected graph < !, complexity *for* loops with *Union-Find* it's a method + −(1 , (which for the complete graph is less complex than sorting.

- Exclusively for undirected graphs!
- We initialize the min. spanning tree by moving all vertices and leaving the set of edges empty
- We sort the edges of the input graph in ascending order of weight(log$!)
- We pass along the edges for so long that in min. the crucifix tree has fewer edges than (−)1
  - If the edge we took from the sorted set of edges does not form a cycle in min. to the crucifying tree, we add it to the edges min. crucifixion tree
- For cycle detection we can use summary *UnionFind* method.



Kruskal's algorithm

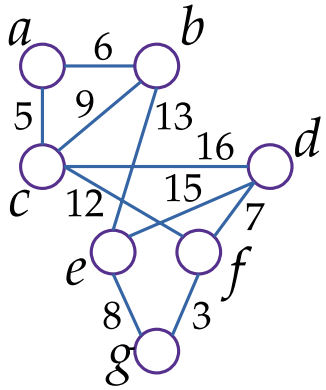


$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 & 13 & 0 & 15 & 0 & 0 & 8 \\ 0 & 0 & 12 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 8 & 3 & 0 \end{bmatrix} \end{array}$$

= 3, = 5, = 6, = 7,
= 8, = 9, = 12, = 13,
= 15, = 16

Iteration 1: Edge $fg = 3$	**Iteration 2:** Edge $ac = 5$	**Iteration 3:** Edge $ab = 6$
Iteration 4: Edge $df = 7$	**Iteration 5:** Edge $eg = 8$	**Iteration 6:** Edge $bc = 9$

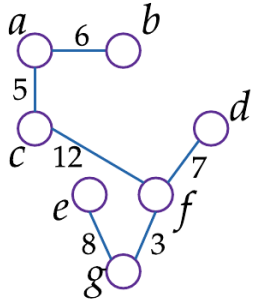
Kruskal's algorithm



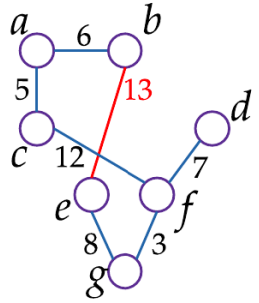
$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 & 13 & 0 & 15 & 0 & 0 & 8 \\ 0 & 0 & 12 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 8 & 3 & 0 \end{bmatrix} \end{array}$$

= 3, = 5, = 6, = 7,
= 8, = 9, = 12, = 13,
= 15, = 16

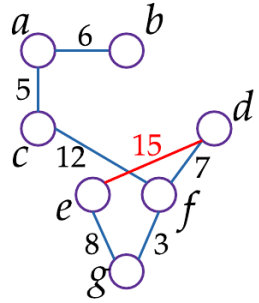
Iteration 7: Edge $cf = 12$



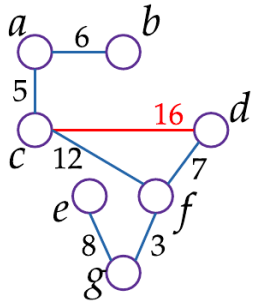
Iteration 8: Edge $be = 13$



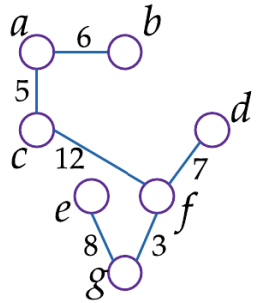
Iteration 9: Edge $de = 15$



Iteration 10: Edge $cd = 16$



Final MST



Final MST adjacency matrix

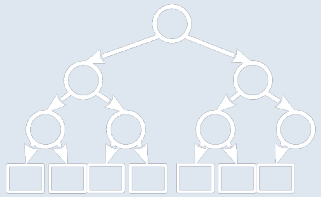
$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 12 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 8 & 3 & 0 \end{bmatrix} \end{array}$$

Dijkstra's Algorithm (MST)

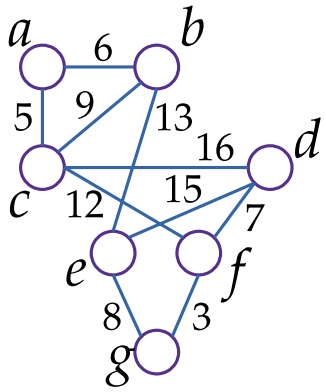
```
procedure DijkstraMST(G)
    MST ← (V = V(G), E = ∅)
    for e_i ∈ E(G) do
        add e_i to E(MST)
        if there is a cycle in MST then
            remove the maximal weight edge from the cycle
    return MST
```

- The complexity of the base iteration is ( ). DFS algorithm is (+),what is becoming (2),that is ( )for min. crucifixion tree.
- Ultimately, the complexity of Dijkstra's algorithm (*).

- The disadvantage of Kruskal's algorithm is its high complexity due to the need to sort edges
- Dijkstra's method is somewhat different
  - No sorting
  - The moment we detect a cycle, we remove the heaviest edge from the cycle
- Here to us *UnionFind* method does not work, so we have to use the method that uses extended DFS (shown in front of several displays)
  - After we add the edge ,let's take one of those two vertices as the initial vertex
  - If we return to that initial peak, then we have a cycle
  - We can also save edge weights in the stack - to detect the edge with the highest weight

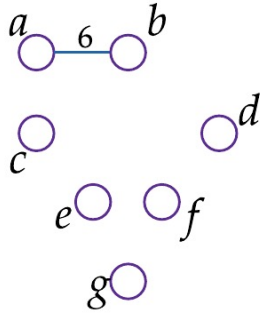
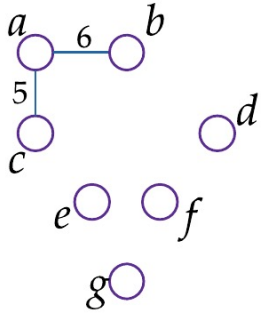
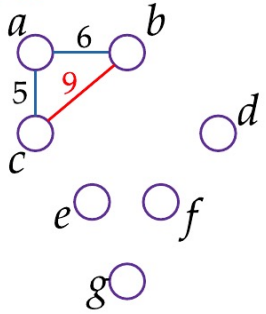
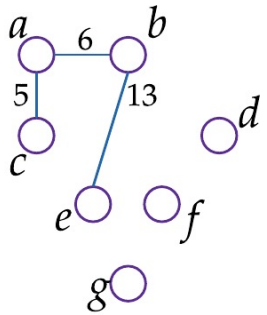
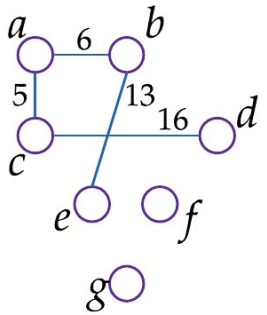
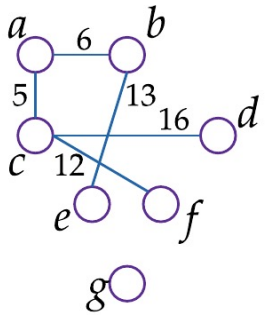


Dijkstra's Algorithm (MST)

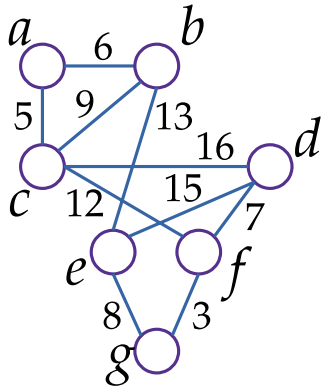


$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 & 13 & 0 & 15 & 0 & 0 & 8 \\ 0 & 0 & 12 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 8 & 3 & 0 \end{bmatrix} \end{array}$$

, , , , , , , , , ,

Iteration 1: Edge $ab = 6$ 	**Iteration 2:** Edge $ac = 5$ 	**Iteration 3:** Edge $bc = 9$, Cycle, Remove bc 
Iteration 4: Edge $be = 13$ 	**Iteration 5:** Edge $cd = 16$ 	**Iteration 6:** Edge $cf = 12$ 

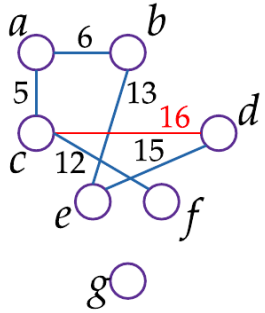
Dijkstra's Algorithm (MST)



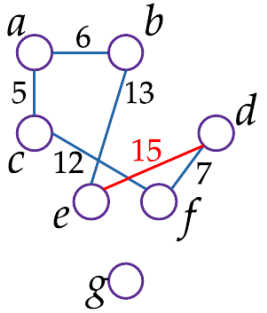
$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 & 13 & 0 & 15 & 0 & 0 & 8 \\ 0 & 0 & 12 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 8 & 3 & 0 \end{bmatrix} \end{array}$$

, , , , , , , , , ,

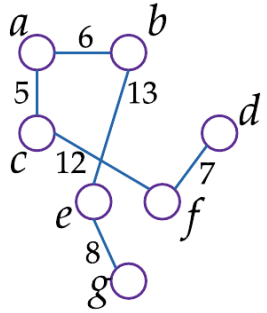
Iteration 7: Edge $de = 15$, Cycle, Remove cd



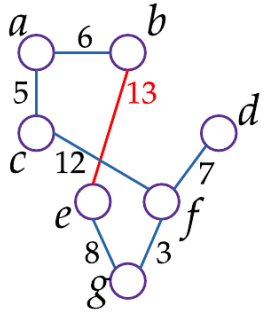
Iteration 8: Edge $df = 7$, Cycle, Remove de



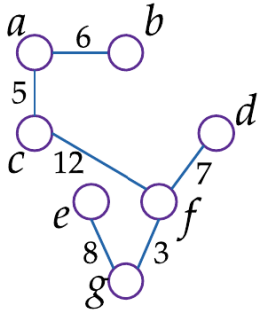
Iteration 9: Edge $eg = 8$



Iteration 10: Edge $fg = 3$, Cycle, Remove be



Final MST



Final MST adjacency matrix

$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 12 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 8 & 3 & 0 \end{bmatrix} \end{array}$$

Prim's algorithm

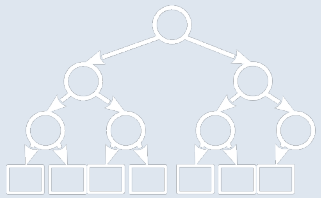
- Let's imagine that we decomposed the graph G into a set of elementary spanning trees ST(G), so that each vertex forms one tree

  $= (\{ 2): 1 \leq \leq , \ 2= \ = 2, \ = \varnothing (2\in (\{)\}\}$)

  - by adding an edge between two elementary spanning trees we get a new unique spanning tree

    $\text{AND}U \ B= (\ \text{AN}(\text{D}U)(\ B), \ (\ \text{AND}) \ U \ (\ B) \ U \{ \text{C}\})$

- The basic idea of Prim's algorithm is **growing** minimal spanning tree $(3\in)$ and the rest of the elemental crucifixion trees $4=(\ ) \setminus \{ 3\}$

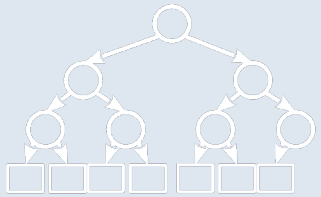


Prim's algorithm

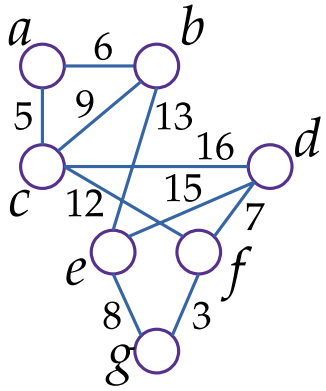
```
procedure PRIM(G, v_s)
    MST ← (V_g = {v_s}, E = ∅)
    V_r ← V(G) \ {v_s}
    while V_r ≠ ∅ do
        choose minimal weighted edge e_n = uv ∈ E(G) such that
                                    u ∈ V(MST) and v ∈ V_r
        add v to V_g(MST)
        remove v from V_r
        add e_n = uv to E(MST)
    return MST
```

- The complexity of the base iteration is ( ).
- The problem is the search for edges of minimum weight.
  - In a sequential implementation, the total complexity is (!).
  - If we use heap, then the total complexity is (+ log!).

- In the initial min. we add only the initial vertex to the spanning tree 5, without edges
- We define a set 4which initially contains all vertices except the initial vertex 5
- We take the first peak from the set 4. We do this until we have peaks in that set
  - Let's find the edge C= minimum weight between the top and one of the peaks in the growing crucifixion tree D
  - Let's add the top and the edge Cinto a growing crucifix tree D
  - Top we remove from the set of vertices E

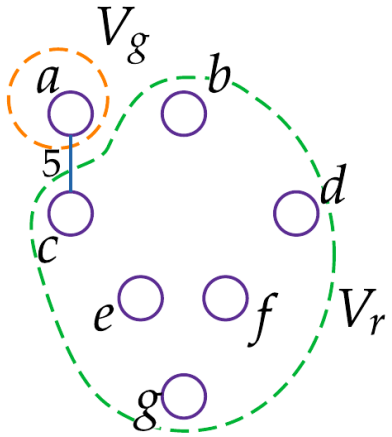


Prim's algorithm

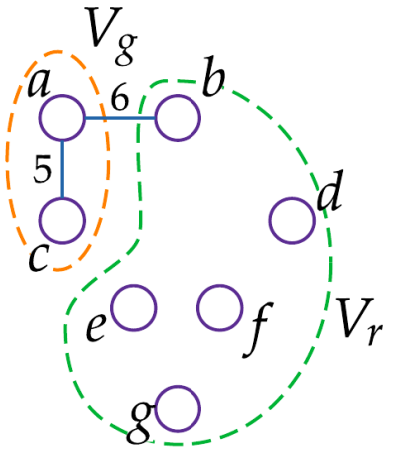


$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 & 13 & 0 & 15 & 0 & 0 & 8 \\ 0 & 0 & 12 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 8 & 3 & 0 \end{bmatrix} \end{array}$$

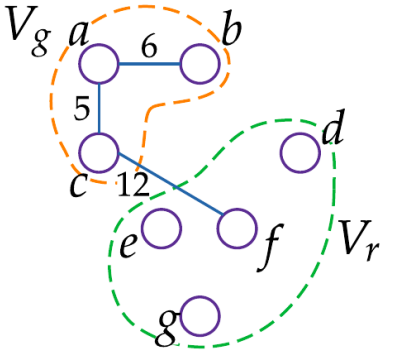
Iteration 1: Edge $ac = 5$



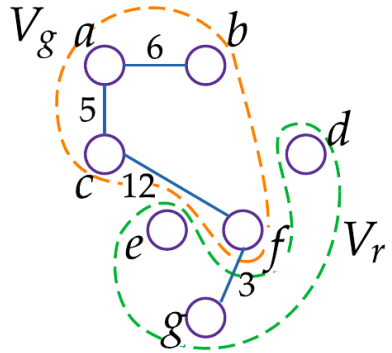
Iteration 2: Edge $ab = 6$



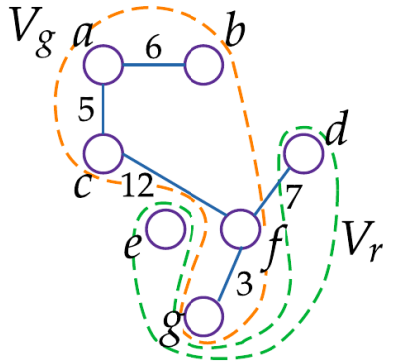
Iteration 3: Edge $cf = 12$



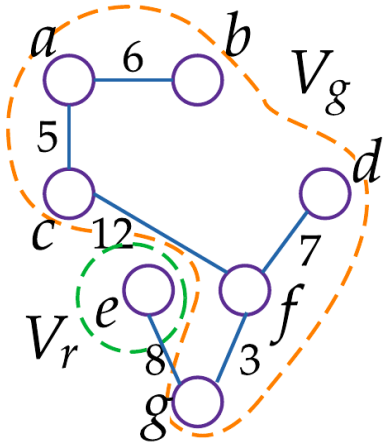
Iteration 4: Edge $fg = 3$

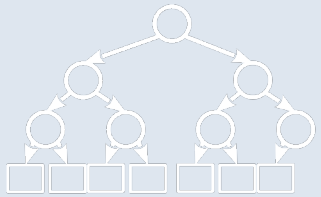


Iteration 5: Edge $fd = 7$

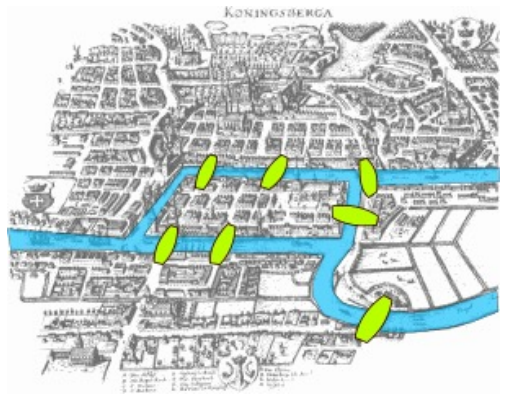


Iteration 6: Edge $eg = 8$

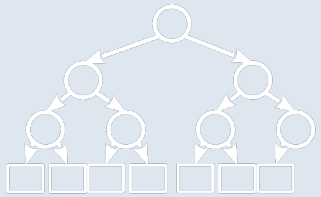




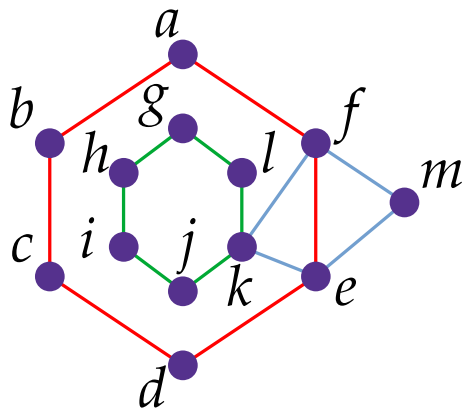
Euler graphs



- Leibniz is the first to propose a branch*position geometry*
- Euler was the first to postulate*positional geometry*on the problem of the seven bridges of Königsberg (today Kaliningrad)
- The problem was to find a route through Königsberg so that each of the seven bridges was crossed only once
  - Euler first saw through the problem because he thought it was trivial and didn't want to waste time on it
  - Later, his observations on this problem gave rise to the theory of graphs
  - Euler proved that the problem is unsolvable, but he also offered a class of problems that are solvable
- Later, on the original problem of the seven bridges of Königsberg, some generic problems are derived, such as the Chinese postman problem

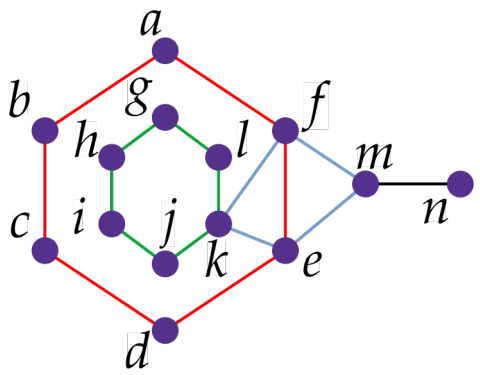


Euler graphs



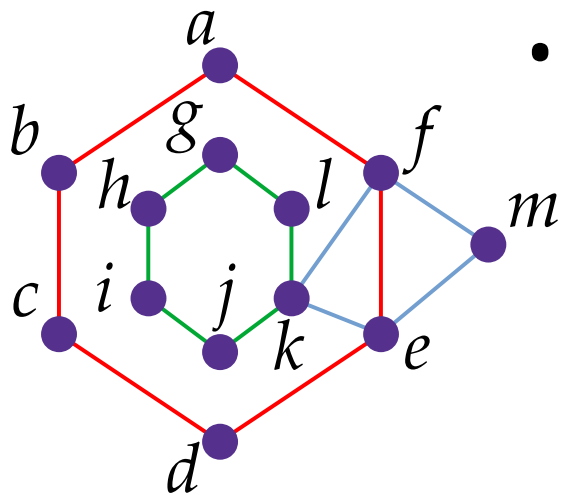
- **Euler trajectory**(*trail*) is a path through the graph that passes each edge of the graph only once
- **Euler's circle**(*circuit*) is an Euler path that starts and ends at the same vertex
- **Euler graph**–is a graph that is made up of an Euler circle or path
- **Theorem**–A connected undirected graph that has all vertices of even degree is an Euler graph
  - **Evidence**
    - Let's imagine that we are doing a tour of the graph. For each entry into the top we use one adjacent edge, while for exiting the top we use another adjacent edge.
    - We can visit each vertex more than once, which means that each vertex of the Euler graph must have*number of tours**2 adjacent edges
    - Thus, the Euler graph is made of the Euler circle

Euler graphs



- An Euler graph can also be made of an Euler path that does not have to start and end at the same vertex!?
- Unlike the Euler graph, which is made up of the Euler circle, in this case exactly two vertices in the graph may be of odd degree
  - Regardless, there is a tour that traverses each edge of the graph exactly once

Euler circles



- We can see that the Euler graph in the example consists of several non-Eulerian circles

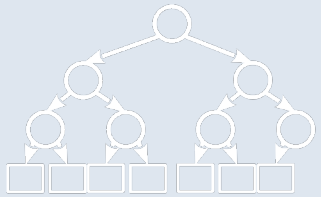
  6= , 7= *h* , 8=

  - These circles share only common vertices, and can be connected through these common vertices
  - So the Euler circle is then

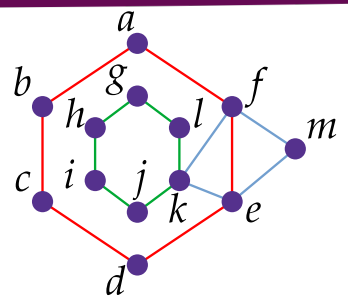
    = !∪ "∪ F

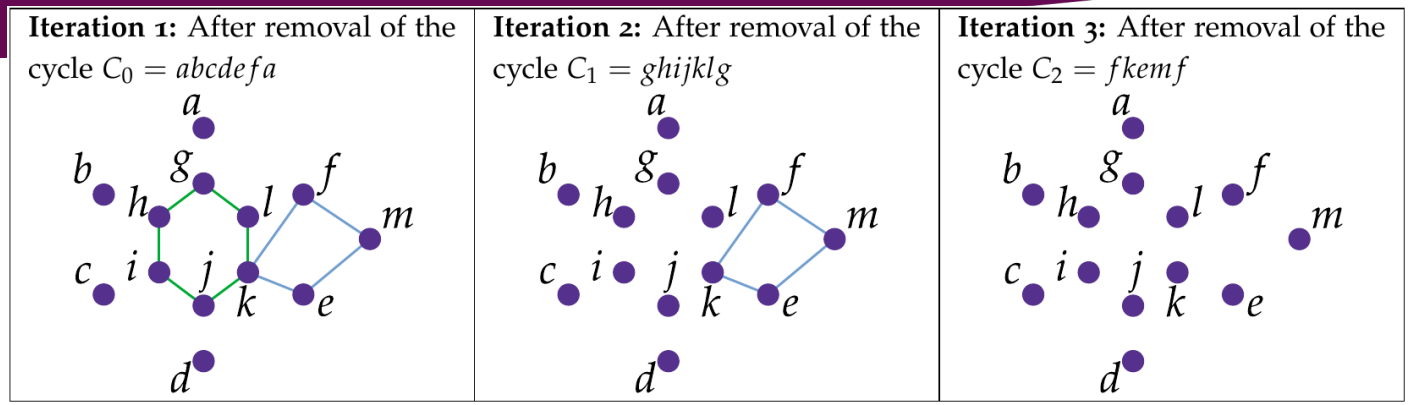
    = *h*

- The idea of detecting Euler circles is based on the traversal of the graph with the removal of edges that have been traversed



Euler circles





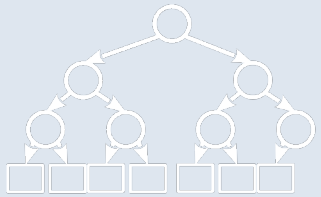
- We go around the graph and remove vertices. At the moment when we are in the initial vertex, we have either an Euler circle or one of the non-Eulerian circles
  - Since the Euler graph is connected, non-Eulerian circles share vertices
  - Shared vertices must have an even degree even after removing the non-Eulerian circle
  - We continue traversing the graph until we can detect more non-Eulerian circles in the graph and until we have removed all edges of the graph
- If we are left with vertices that do not form a non-Eulerian circle, and we still have remaining edges in the graph - then obviously some of the vertices did not have an even degree and it is not possible to find an Euler circle in the graph

Euler circles

```
procedure IsEulerCircuit(G, u)
    while true do
        u_0 ← u
        while there is an edge uv in G do
            add uv to the circuit
            remove uv from the graph G
            u ← v
        if u ≠ u_0 then
            return false
        if there are edges in G then
            pick a vertex u that has incident edges
        else
            return true
```

- The complexity of this algorithm is (+)
- Traversing the graph goes through all the edges, while searching for a new vertex that still has an edge requires going through all the vertices

- We start with some arbitrary initial vertex = MR
  - As long as we have the edge which leads from the top we create a circle
    - Remove the edge
    - Top we transfer to the top
  - If we no longer have edges leading from the top although not the initial peak $, then this graph**It's NOT**made of Euler's circle
    - At this point we know that we have either an Euler circuit or one of the non-Eulerian circuits
- If there are still edges, then we select a new starting vertex MRand we're back to circle detection again
- If there are no more edges in the graph, then we have an Euler graph made of an Euler circle!

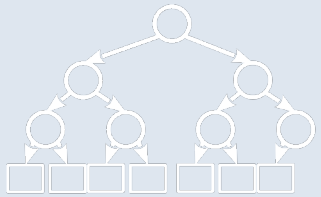


Hierholzer's algorithm

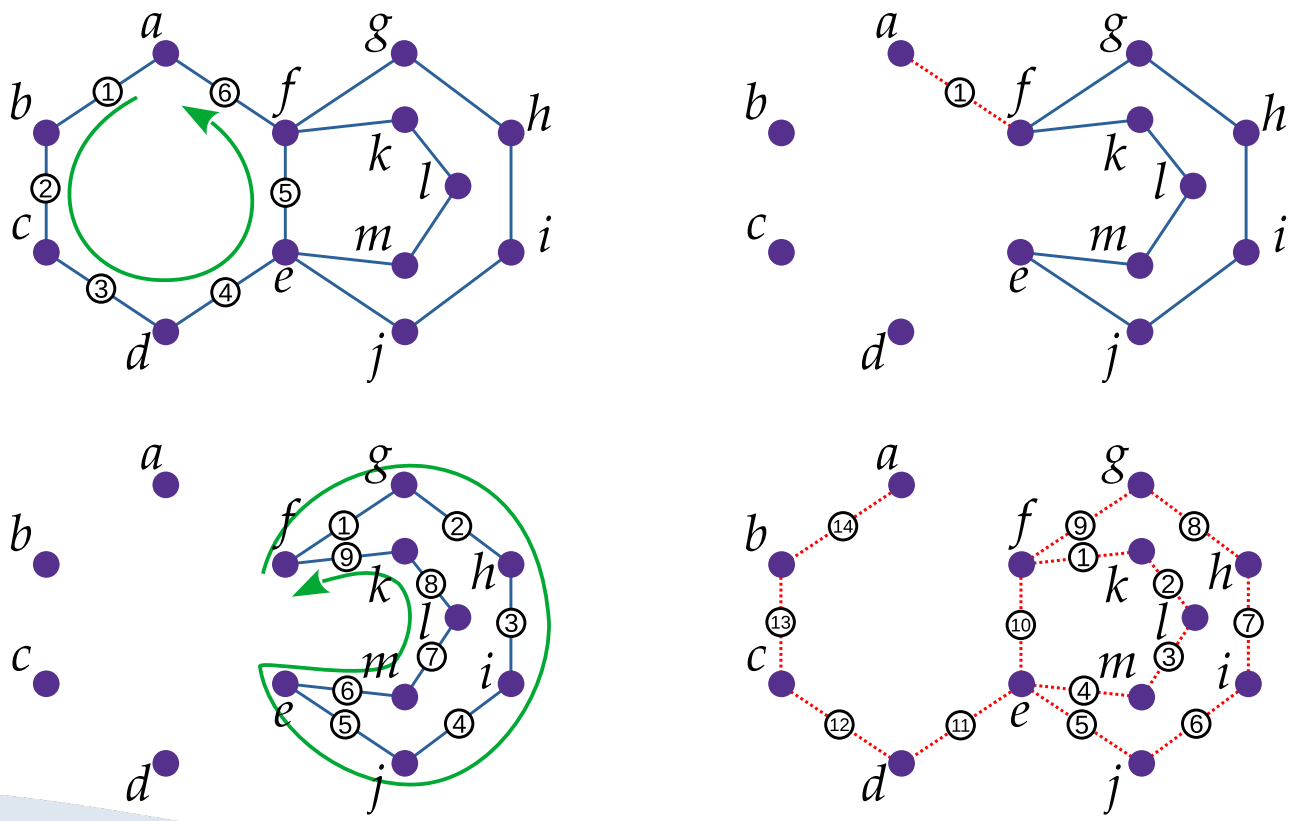
```
procedure HIERHOLZER(G, u)
    s ← new empty stack
    cycle ← ∅
    s.push(u)
    while s not empty do
        u ← last element on the stack s
        if u has adjacent vertices then
            v ← one of the adjacent vertices of u
            s.push(v)
            remove edge uv from G
        else
            e ← edge uv from last two vertices on the stack
            add e to cycle
            s.pop()
    return cycle
```

- The complexity of this algorithm is ( )

- Taking advantage of the fact that the Euler graph is connected, we can use a stack to avoid blindly searching all vertices for a new edge
  - As we progress around the circle, we put the tops on the stack
  - When we reach the end of the circuit, we return to the visited vertices by taking them from the stack
    - If the previous circle was Euler, then we return back to the initial vertex
    - If the previous circle was not Euler, and given that the Euler graph is connected, on the way back we will encounter at least one vertex that still has edges
      - We start along that edge, progressing through a new non-Eulerian circle.



Hierholzer's algorithm



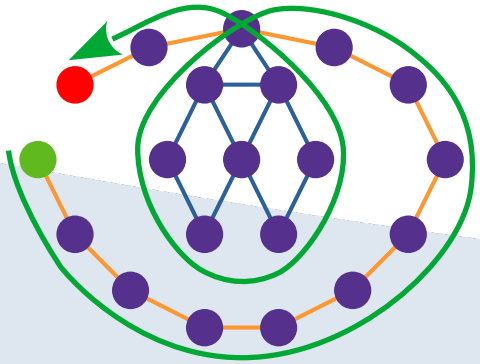
- The ultimate Euler circle is

  fa, kf, lk, ml, em, je, il, hi, gh, fg, ef, de, cd, bc, ab

Fleury's algorithm

```
function FLEURY(G, u)
    cycle ← ∅
    while there are edges in G do
        pick an edge uv from G prioritizing
            non-bridge edges over bridge edges
        add uv to the cycle
        remove uv from the graph G
        u ← v
    return cycle
```

- The complexity of this algorithm is ( )
- However, the detection of whether an edge is a bridge or not raises the complexity of the algorithm to ("), which is slower than the Hierholzer algorithm



- The concept of Fleury's algorithm is based on the selection of the next edge by which we move along the Euler circle
  - If an edge that is not a bridge leads from the current vertex (*bridge*), then we select it
  - Only when there is only a bridge leading from the top, we move along it
  - Let's remember – a bridge is an edge whose removal makes the graph disconnected
    - This can happen when we move along the Euler graph
    - If we have edges left in both partitions of the graph, we cannot go back
    - In this way, we will not detect the Euler circle
- Everything is based on instruction (*lemma*), by which we determine that by removing the first edge on the initial vertex, the degree of that vertex becomes odd and thus it becomes the last vertex that we will visit