Greedy Strategy

Algorithm : Design & Analysis

[14]

In the last class…

- Undirected and Symmetric Digraph
- UDF Search Skeleton
- Biconnected Components
  - Articulation Points and Biconnectedness
  - Biconnected Component Algorithm
  - Analysis of the Algorithm

Greedy Strategy

- Optimization Problem
- MST Problem
  - Prim's Algorithm
  - Kruskal's Algorithm
- Single-Source Shortest Path Problem
  - Dijstra's Algorithm
- Greedy Strategy

Optimizing by Greedy

- Coin Change Problem
  - [candidates] A finite set of coins, of 1, 5, 10 and 25 units, with enough number for each value
  - [constraints] Pay an exact amount by a selected set of coins
  - [optimization] a smallest possible number of coins in the selected set
- Solution by greedy strategy
  - For each selection, choose the highest-valued coin as possible.

Greedy Fails Sometimes

- If the available coins are of 1,5,12 units, and we have to pay 15 units totally, then the smallest set of coins is {5,5,5}, but not {12,1,1,1}
- However, the correctness of greedy strategy on the case of {1,5,10,25} is not straightly seen.

Greedy Strategy

- Constructing the final solution by expanding the partial solution step by step, in each of which a selection is made from a set of candidates, with the choi[...] made **must** be:
  - [feasible] it has to s[...] problem's constrain[...]
  - [locally optimal] it has[...] the best local choice am[...] feasible choices on the st[...]p
  - [irrevocable] the candidate selected can never be de-selected on subsequent steps

set greedy(**set** c[...]ndidate)

set S=∅[...]

[...](S) **and**

[...]optimizing *x*

[...]candidate;

[...]ndidate=candidate-{*x*};

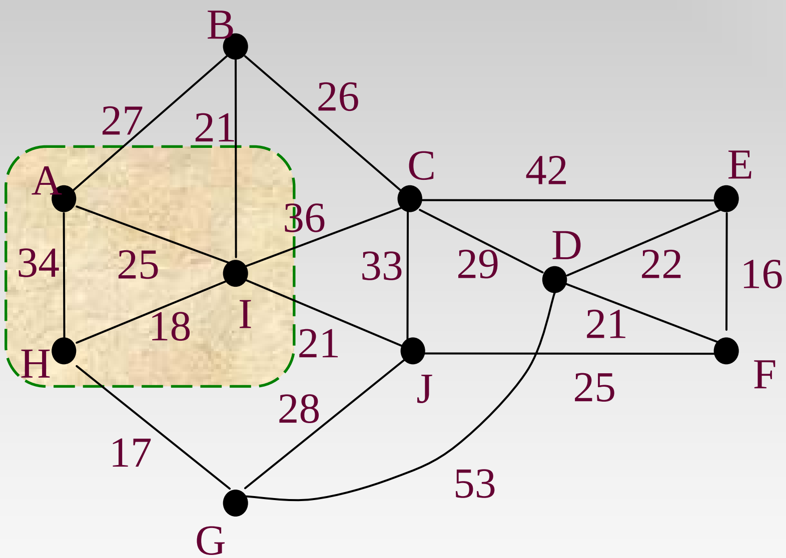
if *feasible*(*x*) **then** S=S∪{*x*};

if *solution*(S) **then return** S

else return ("no solution")

Key: trading off on "local optimization" and "feasibility"

Weighted Graph and MST

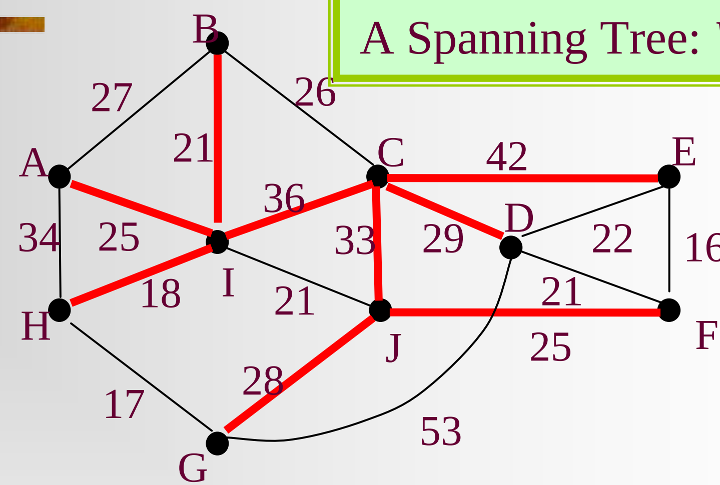


A weighted graph

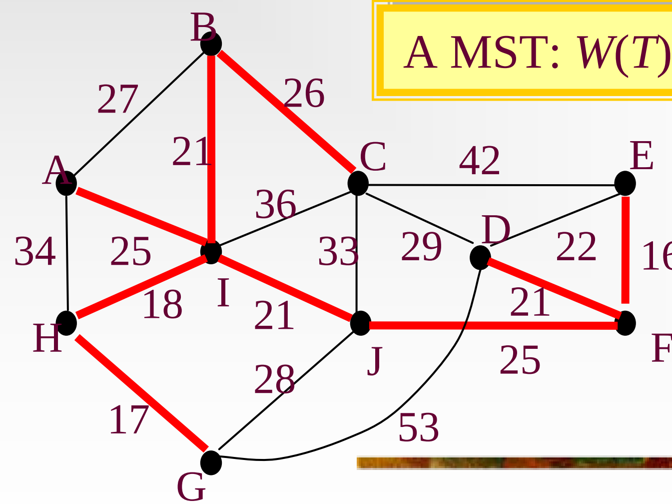
The nearest neighbor of vertex ***I*** is ***H***

The nearest neighbor of shaded subset of vertex is ***G***

A Spanning Tree: $W(T)$=257



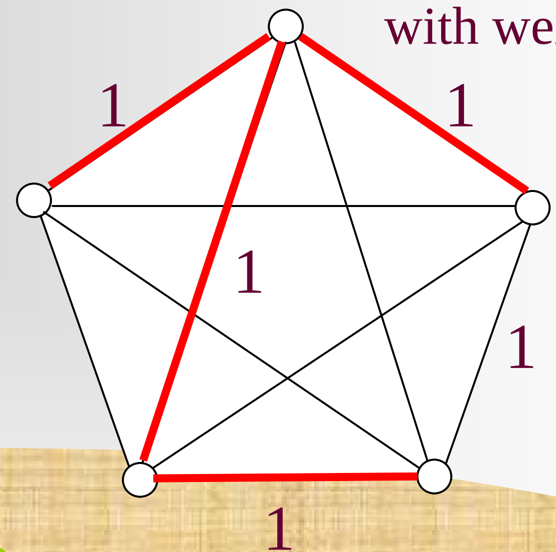
A MST: $W(T)$=190

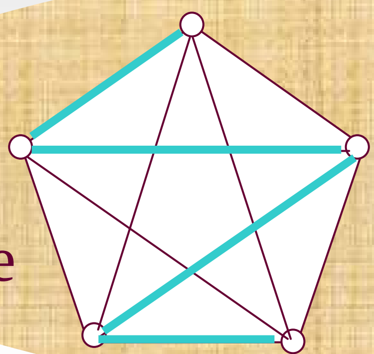


Graph Traversal and MST

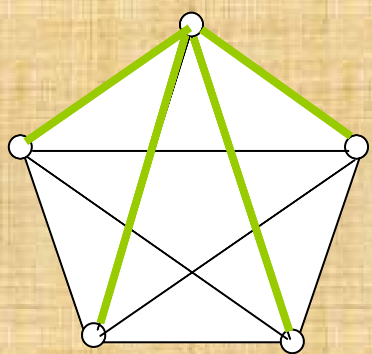
There are cases that graph traversal tree **cannot** be minimum spanning tree, with the vertices explored in any order.

All other edges with weight 5





DFS tree



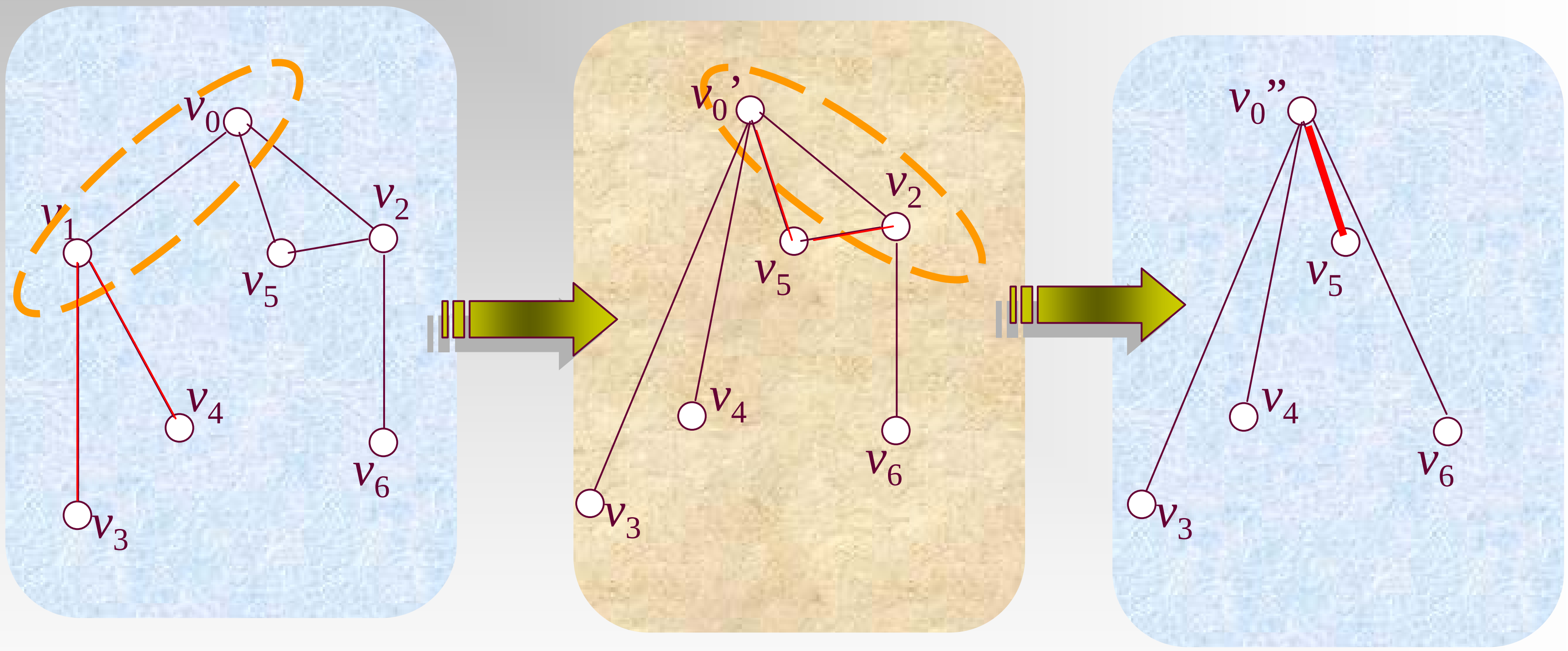
BFS tree

in any ordering of vertex

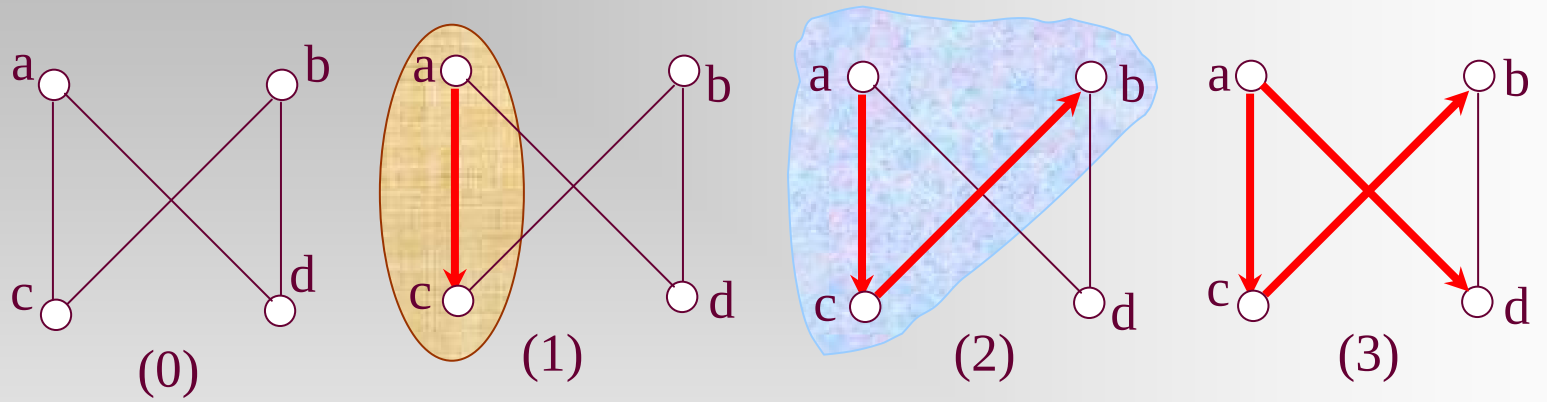
Greedy Algorithms for MST

- Prim’s algorithm:
  - Difficult selecting: “best local optimization means **no cycle and small weight under limitation**.
  - Easy checking: doing nothing
- Kruskal’s algorithm:
  - Easy selecting: smallest in primitive meaning
  - Difficult checking: **no cycle**

Merging Two Vertices



Constructing a Spanning Tree



0. Let a be the starting vertex, selecting edges one by one in original graph

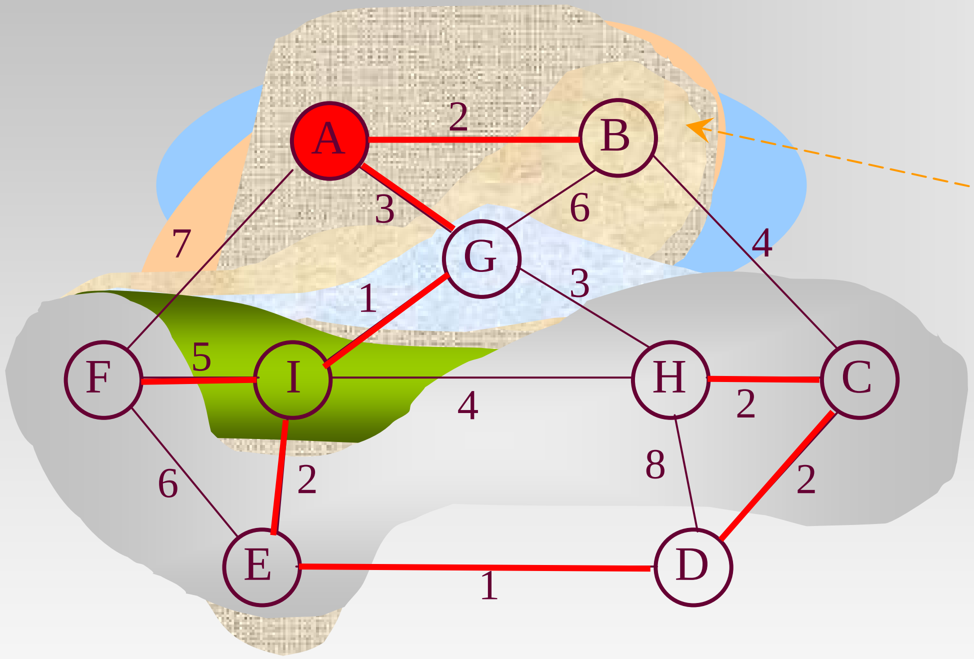
1. Merging a and c into a'({a,c}), selecting (a,c)

2. Merging a' and b into a"({a,c,b}), selecting (c,b)

3. Merging a" and d into a"'({a,c,b,d}), selecting (a,d) or (d,b)

Ending, as only one vertex left

Prim's Algorithm for MST

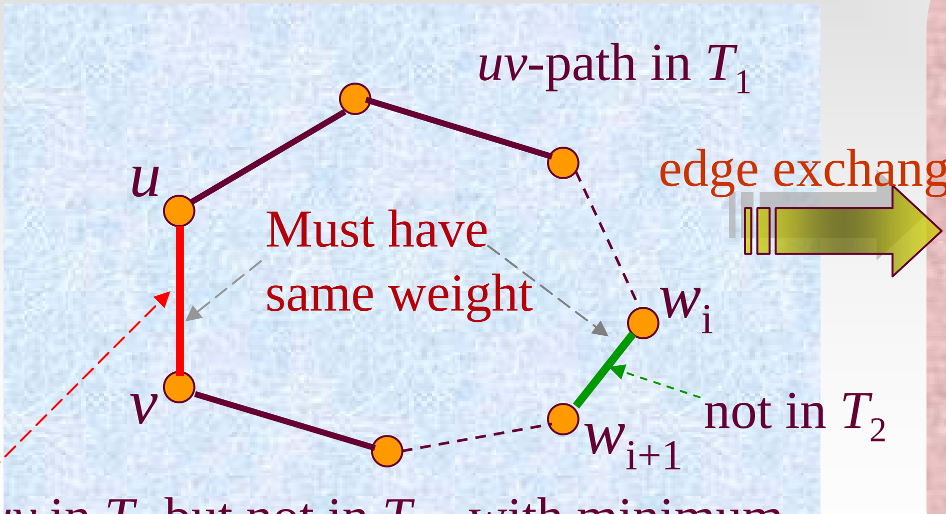


edges included in the MST

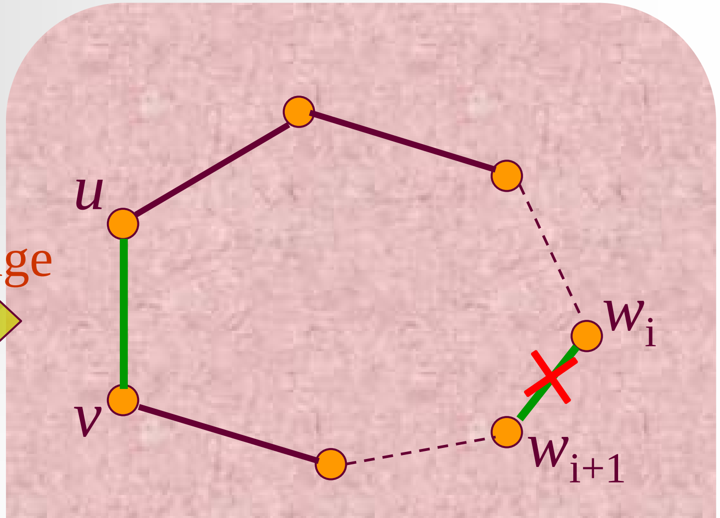
Greedy strategy:
For each set of fringe vertex, select the edge with the minimal weight, that is, local optimal.

Minimum Spanning Tree Property

- A spanning tree T of a connected, weighted graph has MST property if and only if for any non-tree edge uv, $T \cup \{uv\}$ contain a cycle in which uv is one of the maximum-weight edge.
- **All the spanning trees having MST property have the same weight.**



edge uv in T_2 but not in T_1 , with minimum weight among all different edges



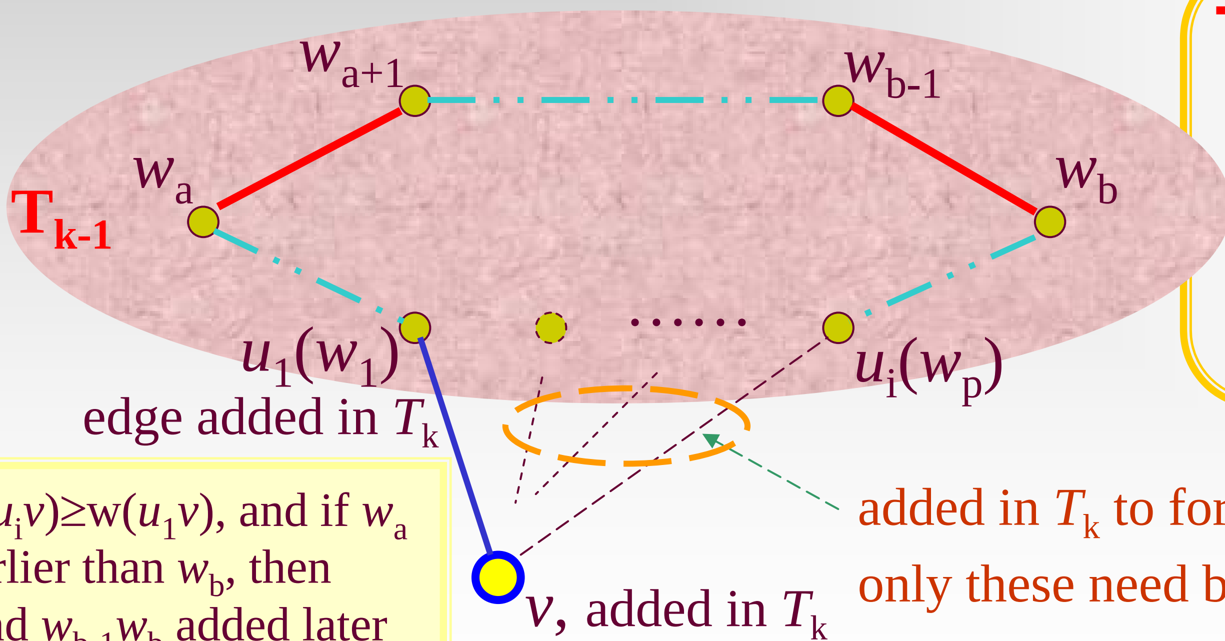
a new spanning tree: same weight as T_1, less different edges from that of T_2

MST Property and Minimum Spanning Tree

- In a connected, weighted graph $G=(V,E,W)$, a tree T is a minimum spanning tree if and only if T has the MST property.
- Proof
  - $\Rightarrow$ For a minimum spanning tree T, if it doesn't has MST property. So, there is a non-tree edge uv, and $T\cup\{uv\}$ contain an edge xy with weight larger than that of uv. Substituting uv for xy results a spanning tree with less weight than T. Contradiction.
  - $\Leftarrow$ As claimed above, any minimum spanning tree has the MST property. Since T has MST property, it has the same weight as any minimum spanning tree, i.e. T is a minimum spanning tree as well.

Correctness of Prim's Algorithm

- Let T_k be the tree constructed after the kth step of Prim's algorithm is executed, then T_k has the MST property in G_k, the subgraph of G induced by vertices of T_k.



assumed first and last edges with larger weight than $w(u_i v)$, resulting contradictions.

added in T_k to form a cycle, only these need be considered

Note: $w(u_i v) \geq w(u_1 v)$, and if w_a added earlier than w_b, then $w_a w_{a+1}$ and $w_{b-1} w_b$ added later than any edges in $u_1 w_a$-path, and v as well

Key Issue in Implementation

- Maintaining the set of fringe vertices
  - Create the set and update it after each vertex is “selected” (***deleting*** the vertex having been selected and ***inserting*** new fringe vertices)
  - Easy to decide the vertex with “highest priority”
  - Changing the priority of the vertices (***decreasing key***).
- The choice: priority queue

Implementing Prim's Algorithm

Main Procedure

primMST(*G*,*n*)

  Initialize the priority queue *pq* as empty;

  Select vertex *s* to start the tree;

  Set its candidate edge to (-1,*s*,0);

  insert(*pq*,*s*,0);

  while (*pq* is not empty)

    v=getMin(*pq*); deleteMin(*pq*);

    add the candidate edge of *v* to the tree;

    updateFringe(*pq*,*G*,*v*);

  return

ADT operation executions:

insert, getMin, deleteMin: *n* times

decreaseKey: *m* times

Updating the Queue

updateFringe(*pq*,*G*,*v*)

  For all vertices *w* adjcent to *v* //2*m* loops

    newWgt=w(*v*,*w*);

    if *w*.status is unseen **then**

      Set its candidate edge to (*v*,*w*,*newWgt*);

      insert(*pq*,*w*,*newWgt*)

    else

      if *newWgt*<getPriorty(*pq*,*w*)

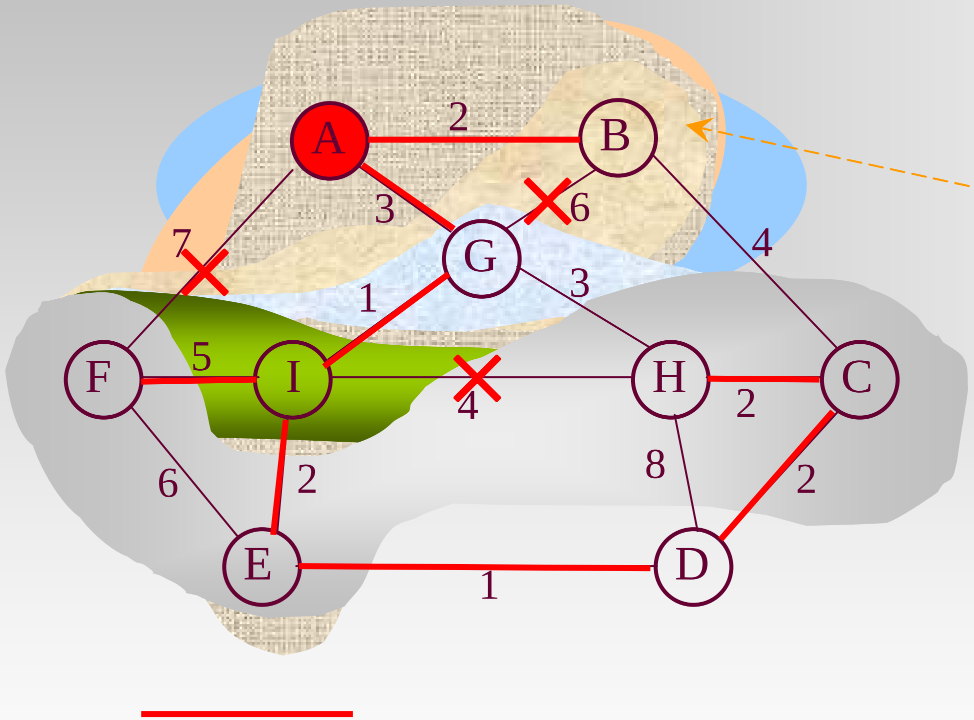
        Revise its candidate edge to (*v*,*w*,newWgt);

        decreaseKey(*pq*,*w*,*newWgt*)

  return

getMin(*pq*) always be the vertex with the smallest key in the fringe set.

Prim's Algorithm for MST



edges included in the MST

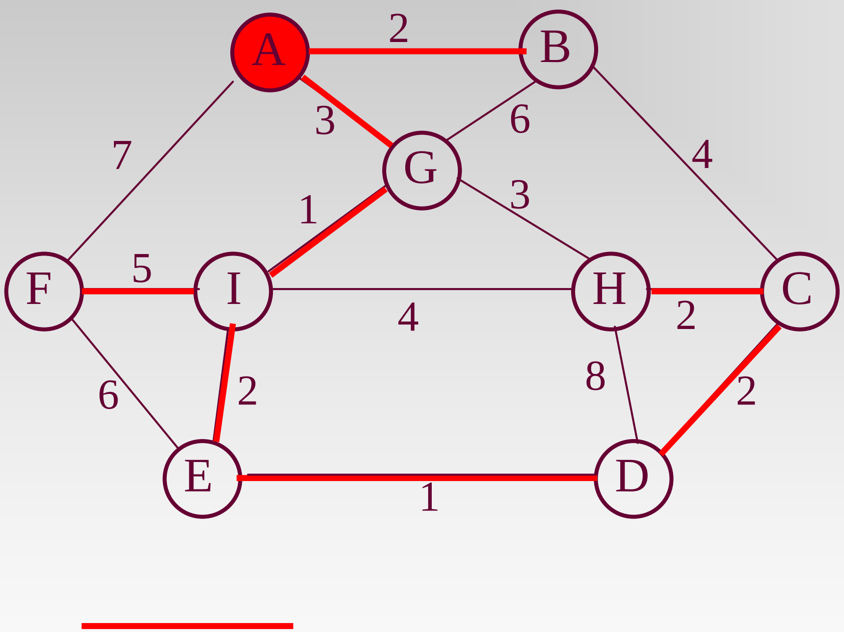
Greedy strategy:
For each set of fringe vertex, select the edge with the minimal weight, that is, local optimal.

Complexity of Prim's Algorithm

- Operations on ADT priority queue: (for a graph with n vertices and m edges)
  - insert: n
  - getMin: n
  - deleteMin: n
  - decreasKey: m (appears in $2m$ loops, but execute at most m)
- So,

  $T(n,m) = O(nT(\text{getMin})+nT(\text{deleteMin+insert})+mT(\text{decreaseKey}))$
- Implementing priority queue using heap, we can get $\boldsymbol{\Theta(n^2+m)}$

Kruskal's Algorithm for MST



edges included in the MST

Also Greedy strategy:
From the set of edges not yet included in the partially built MST, select the edge with the minimal weight, that is, local optimal, in another sense.

Key Issue in Implementation

- How to know an insertion of edge will result in a cycle ***efficiently***?
- For correctness: the two endpoints of the selected edge ***can not*** be in the same connected components.
- For the efficiency: connected components are implemented as dynamic equivalence classes using union-find.

Kruskal's Algorithm: the Procedure

- kruskalMST(G,n,F) //outline
- **int** count;
- Build a minimizing priority queue, pq, of edges of G, prioritized by weight.
- Initialize a Union-Find structure, sets, in which each vertex of G is in its own set.
-
- F=ϕ;
- **while** (isEmpty(pq) == **false**)
- vwEdge = getMin(pq);
- deleteMin(pq);
- **int** vSet = find(sets, vwEdge.from);
- **int** wSet = find(sets, vwEdge.to);
- **if** (vSet ≠ wSet)
- Add vwEdge to F;
- union(sets, vSet, wSet)
- **return**

Simply sorting, the cost will be $\Theta(m\log m)$

Prim vs. Kruskal

- Lower bound for MST
  - For a correct MST, each edge in the graph should be examined at least once.
  - So, the lower bound is $\Omega(m)$
- $\Theta(n^2+m)$ and $\Theta(m\log m)$, which is better?
  - Generally speaking, depends on the density of edge of the graph.

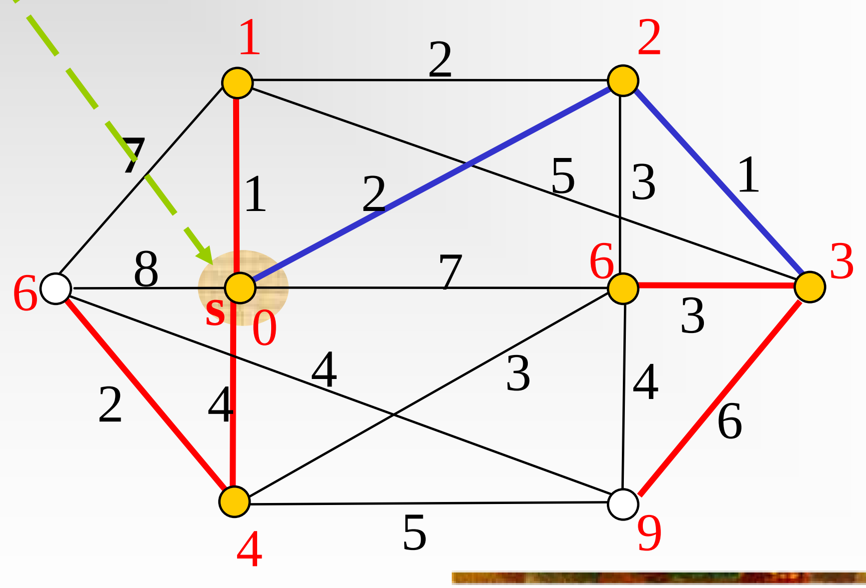
Single Source Shortest Paths

The single source

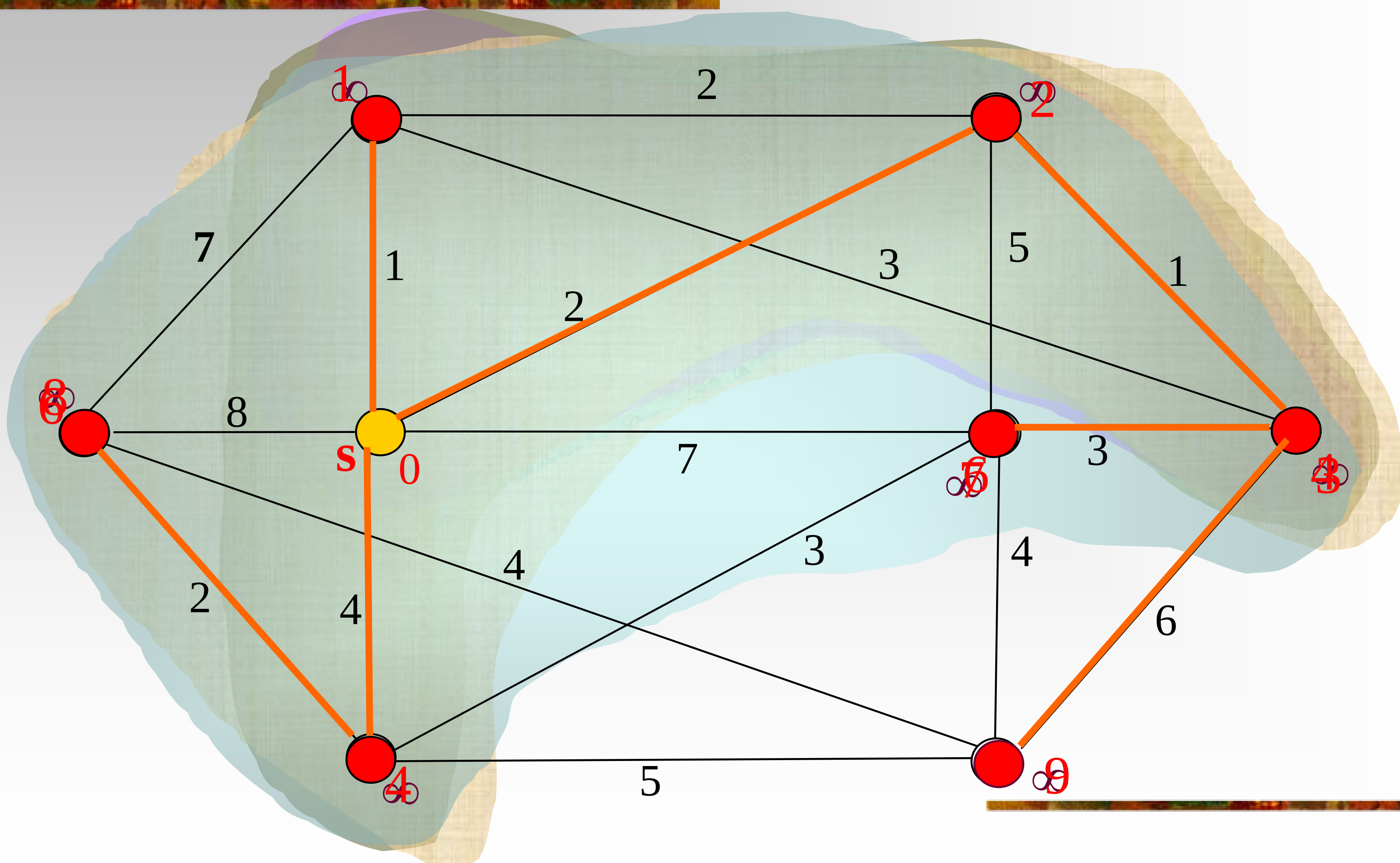
Red labels on each vertex is the length of the shortest path from s to the vertex.

Note:

The shortest [0, 3]-path doesn't contain the shortest edge leaving s, the edge [0,1]



Dijstra's Algorithm: an Example



Home Assignment

- pp.416-:
  - 8.7-8.9
  - 8.14-15
  - 8.25