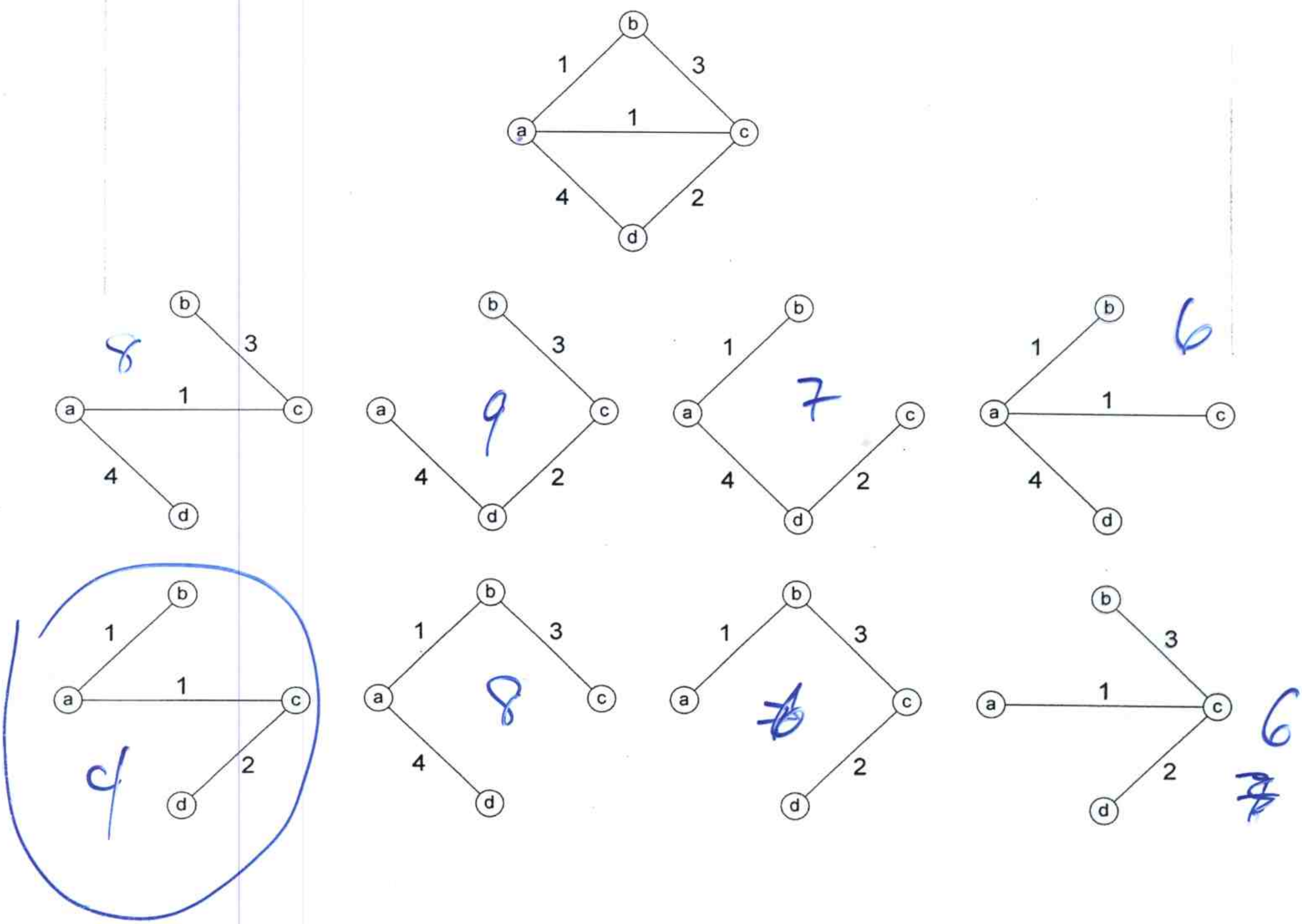
MST's (Cont)



Prim

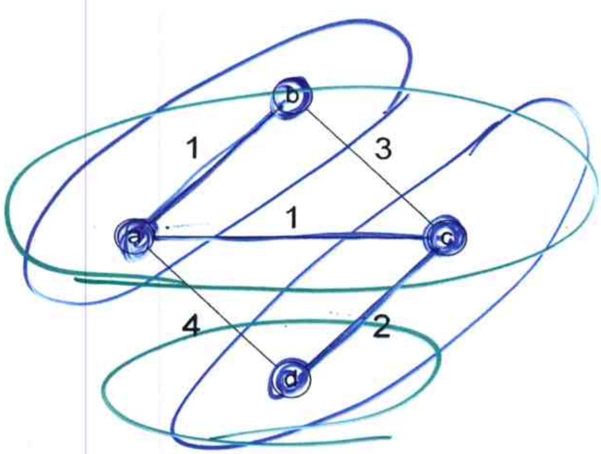
Prim's algo is a greedy approach to finding an MST

Prim (high level description):

```
Prim(G)   // return T, which is a MST of G
– add any vertex of G to the solution T
– let E be all edges of G that connect any vertex
  in T to any vertex not in T
– find an edge in E with minimal weight, and add
  it to T
– repeat the previous 2 steps until all vertices
  are in T
```

that connects a new vertex.

– from a greedy perspective we are continually adding edges such that we always add a minimum weight edge, as this is the edge that will get us closer to a solution at minimal cost



Prim (as written in your textbook)

Soln is V_T plus E_T

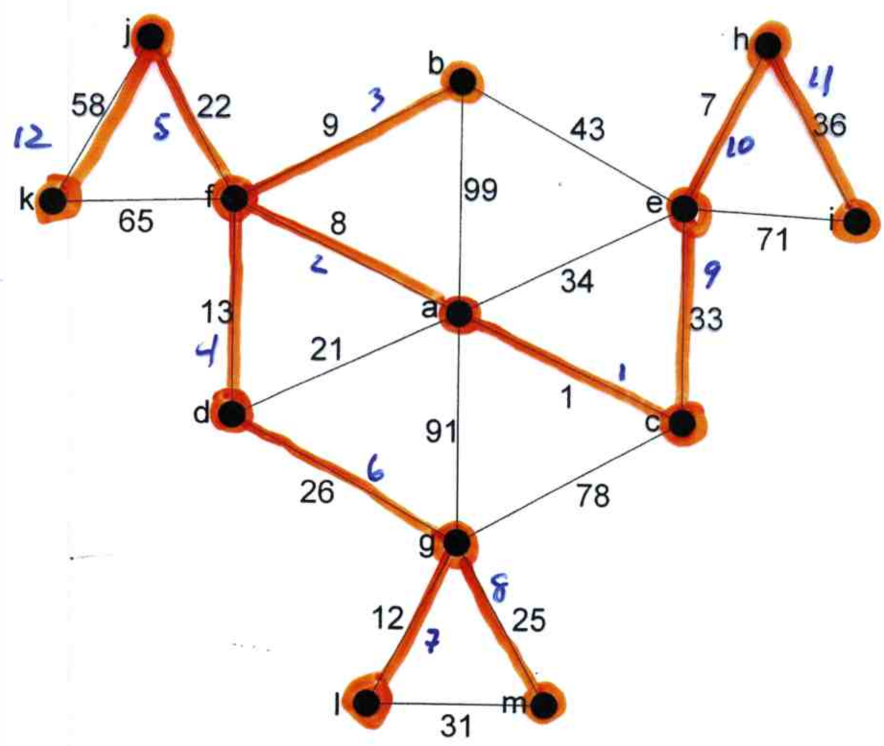
```
Prim(G)
  V_T ← {v_0}
  E_T ← ∅
  for i ← 0 to |V|-1 do
     find a min-weight edge e from the set of edges ...
   ...{u,v} where v is in V_T and u is in V-V_T
   V_T ← V_T ∪ u
   E_T ← E_T ∪ e
  return E_T
```

empty set.

magnitude of V-1 ⇒ # vertices -1

union.

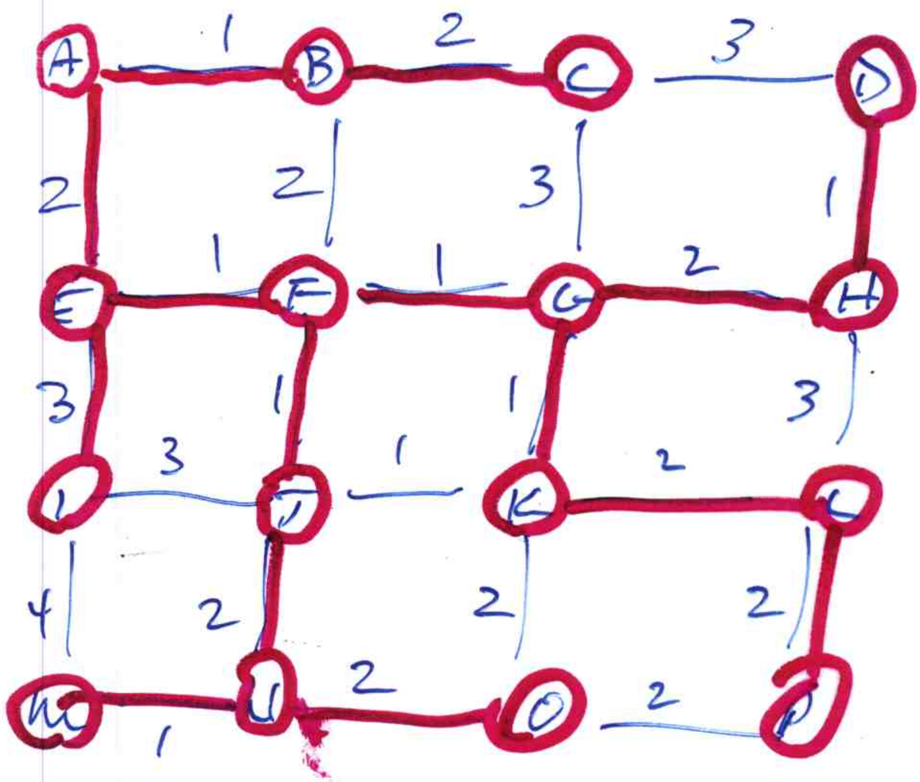
⇒ "add edge e to the soln"



Prim (as written in your textbook)

MST = minimum spanning tree

```
Prim(G)
  V_T ← {v_0}
  E_T ← ∅
  for i ← 0 to |V|-1 do
      find a min-weight edge e from the set of edges
      {u,v} where v is in V_T and u is in V-V_T
      V_T ← V_T ∪ u
      E_T ← E_T ∪ e
  return E_T
```



Note: there are many possible solutions!

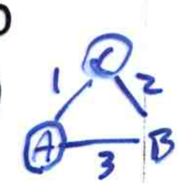
Prims (implementing)

Notice:

- we are always choosing the minimum weight edge that connects a new vertex to the tree
  - maybe we can use a keyed-min-heap?
- if we use a heap, it could contain "candidate edges"
  - but we need to maintain it , ie:
    - we will remove min edge e from heap, and add it (plus its connected v) to the solution, then
      - look for new edges adj to v to add to heap
      - look for new edges that replace edges already in heap *** this is expensive to do ***
- it would be better if the heap could contain "candidate vertices", ie, the vertices on the "fringe"
  - each vertex would be the value, and the key (in the heap) could be the weight of the connecting edge
  - would also need to know which vertex in T (call it the parent of v) the weight corresponds to
    - this way we don't "replace" heap elements, we "update" their key
  - there is a type of heap, called a fibonacci heap, that implements an efficient "key update" operation

Prim (restated)

- Now, let's restate prim so we can use a heap
  - (this is the typical way that prim is described)

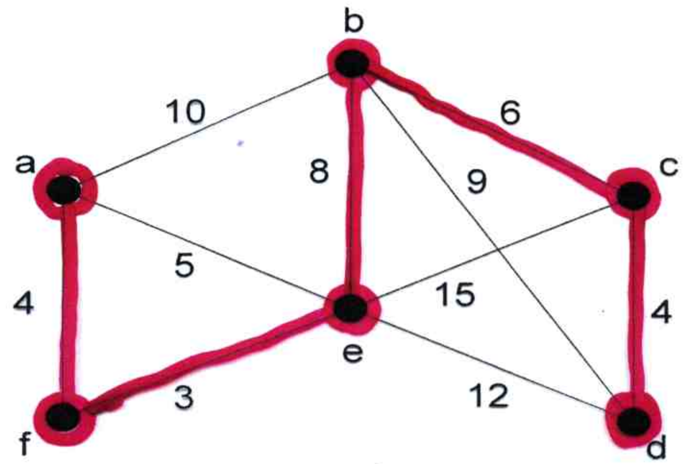


1: ~~C~~
2 ~~3~~: B

B: ~~A~~ C

```
Prim(G)
   create an empty graph S    // the solution
   create an empty keyed min heap PQ(key,value)
                                    (weight: vertex)
   create an empty map parent
   set v0 to be any vertex in G
   add v0 to PQ with key zero
   for each vertex u in graph G except v0
      add u to PQ with key ∞
      parent.put(u,nil)
   while PQ is not empty
      extract next vertex u from PQ
      add vertex u to S
      add edge (u, parent.get(u) ) to S
      for each vertex v adjacent to u do
         if (v is not in S)
            uv_key ← edge(u,v).weight
            if uv_key < key of v
               parent.update(v,u)
               update v in PQ with key uv_key
   return S
```

Another Prim Example (using the PQ)



S - red

PQ weight : vertex

~~∅ : a~~

~~8 10 ∅ : b~~

~~6 15 ∅ : c~~

~~4 9 12 ∅ : d~~

~~3 5 ∅ : e~~

~~4 ∅ : f~~

parent (map).

a : nil

b : ~~nil~~ ~~a~~ e

c : ~~nil~~ ~~e~~ ~~b~~ ~~a~~ b

e : ~~nil~~ ~~a~~ f

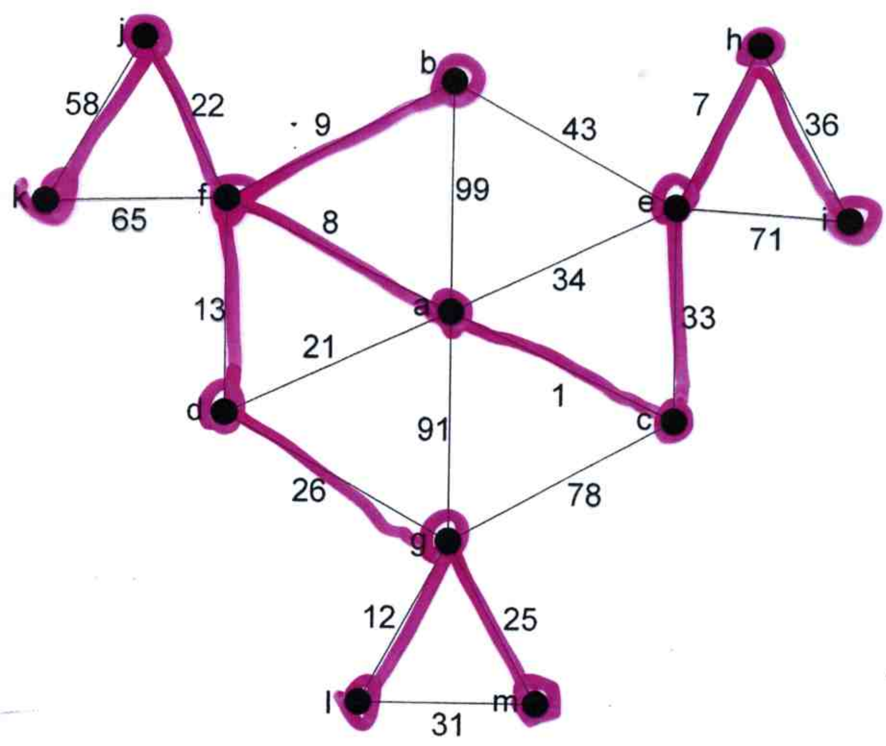
d : ~~nil~~ ~~e~~ ~~b~~ c

f : ~~nil~~ a

Kruskals (overview)

- also greedy
- repeatedly adds the minimum weight edge that does not induce a cycle

- example:



Kruskals (more detailed)

```
Kruskal(G)
    sort e ∈ E in ascending order of
   weights
   E_T ← ∅
   count ← 0
   k ← 0
   while count < |V|-1 do
      k ← k + 1
      if E_T ∪ e_k is acyclic
         E_T ← E_T ∪ e_k
         count ← count + 1
   return E_T
```

E_T: set of edges in the solution

← union e_k with set E_T

- implementation notes:
  - you need to be able to efficiently sort the edges
    - maybe use a regular min-heap?
  - need to be able to determine if adding an edge will create a cycle
    - maybe use a dfs or bfs cycle checker?
      - too slow ...