Week 08

Things to Note ...

- Mid-term exam next week

In This Lecture ...

- Minimum spanning trees, Single-source shortest paths ([S] 20-20.4,21-21.3)
- Fun quiz

Coming Up ...

- Assignment 2
- *Mid-session break*
- Search tree algorithms ([S] Ch.10)

Assignment 1

We are in the final stages of finalising the results …

Median auto-test result: 7/8

A few common issues:

- boundary cases not tested (empty list)
- *program does not compile on CSE-machine*
- program does not compile without warnings ⇒ bad style
- Magic Numbers ⇒ bad style

```
if (cr <= 2 || cr >= 400) { ... }
```

Better:

```
#define MIN_CREDITS 2
#define MAX_CREDITS 480
...
```

- The following is O(n), not O(n^2):

```
for each element in the list do
   if element is what you are looking for then
      start at head and find predecessor of element
   end if
end for
```

- use of `break,continue` in loops ⇒ unstructured programming ⇒ bad style (accepted for now but not for Assignment 2)

Mid-term Exam

Thursday next week (21 Sep)   6:15pm — 7:15pm

Last name begins with A—M :   Rex Vowels Theatre (Building F17)
Last name begins with N—Z :   Physics Theatre (Old Main Building)

Format:

- 5 multiple-choice questions (each worth 2 marks)
- 2 open questions (worth 7 and 8 marks, respectively)
- 60 minutes   + 5 mins reading time
- closed book, but you can bring one A4-sized sheet of your own notes
    - double-sided is ok

For complete instructions see:

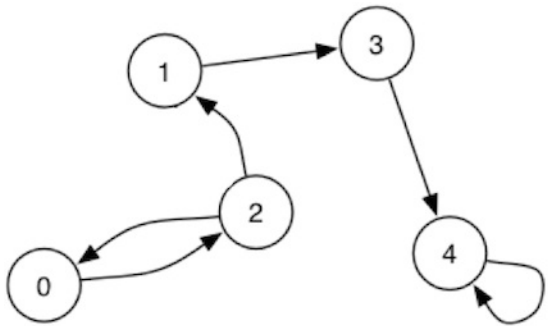
www.cse.unsw.edu.au/~cs9024/MSTinstructions.pdf

Mid-term Exam (cont)

Sample Open Questions

1. Based on an array of edges representation of a directed graph, describe an algorithm in pseudocode to compute the *indegree* (= #edges into a vertex) of every vertex.
   Determine the time complexity of your algorithm depending on *V* (= number of vertices) and *E* (= number of edges).

2. Consider the following directed graph *G*:



   - Show an adjacency matrix representation of *G*.
   - Find a *directed* Euler path of *G* (= directed path using every edge exactly once), or provide an argument why such a path does not exist.
   - Trace the execution of Warshall's algorithm to compute the transitive closure of *G*. Show **`tc[][]`** after every iteration.

Nerds You Should Know

The next in a series on famous computer scientists …

They developed one of the most useful Web tools ...

Nerds You Should Know (cont)

Sergey Brin

Larry Page

- Co-founders of Google
- Brin: BSc University of Maryland
- Page: BSc/BE University of Michigan
- Both moved to Stanford for PhD in mid-1990's
- PhD work led to new ideas on Web searching
  - use keywords like "normal" search engines
  - augment document ranking by "credibility"
  - credibility related to inbound links
- Ideas led to prototype, then to company
- Google Inc. founded in 1998
- Alphabet Inc. created in 2015

Weighted Graphs

Weighted Graphs

Graphs so far have considered

- edge = an association between two vertices/nodes
- may be a precedence in the association (directed)

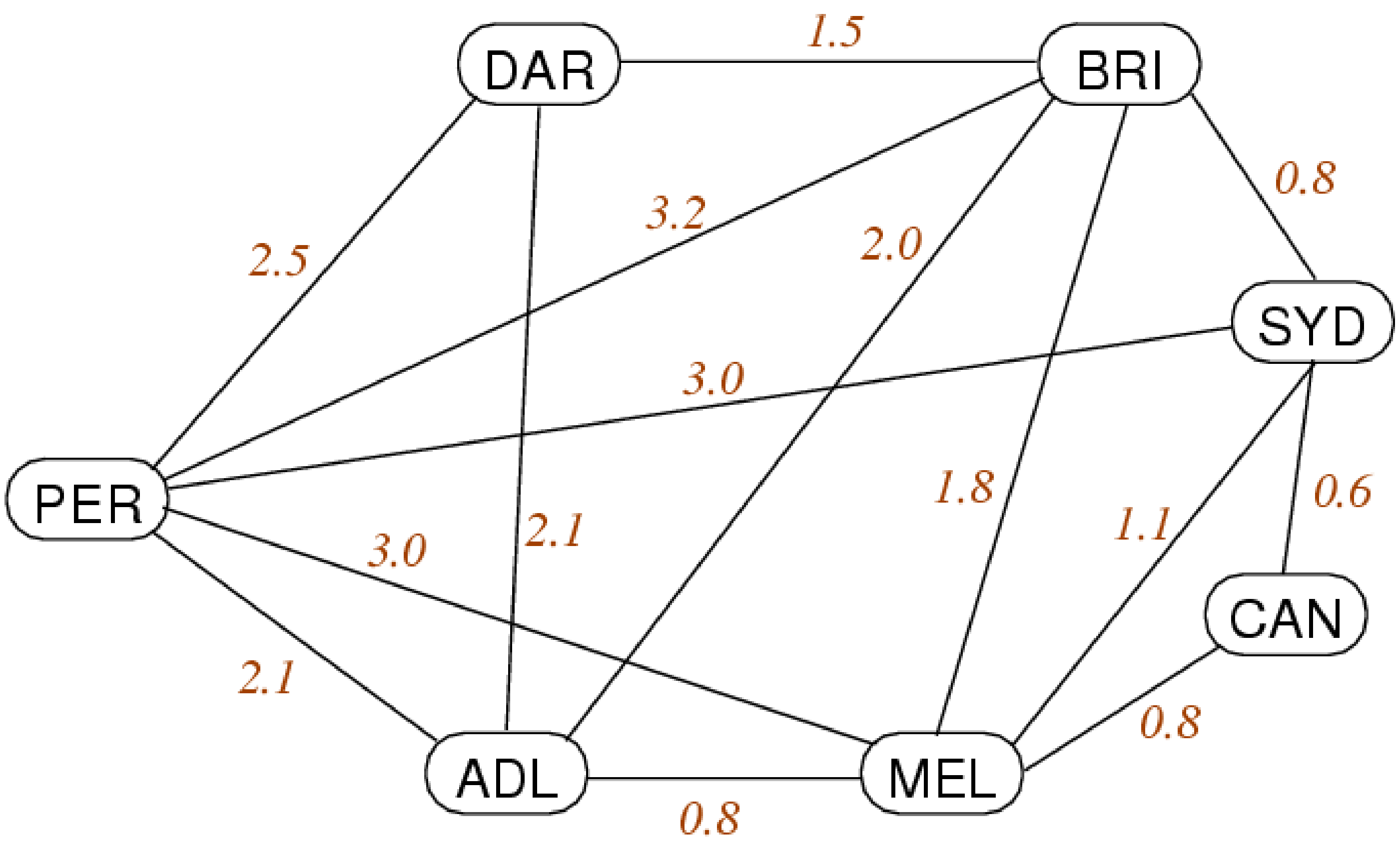
Some applications require us to consider

- a cost or weight of an association
- modelled by assigning values to edges (e.g. positive reals)

Weights can be used in both directed and undirected graphs.

Weighted Graphs (cont)

Example: major airline flight routes in Australia



Representation: edge = direct flight; weight = approx flying time (hours)

Weighted Graphs (cont)

Weights lead to minimisation-type questions, e.g.

1. Cheapest way to connect all vertices?
   - a.k.a. minimum spanning tree problem
   - assumes: edges are weighted and undirected
2. Cheapest way to get from *A* to *B*?
   - a.k.a shortest path problem
   - assumes: edge weights positive, directed or undirected

Exercise #1: Implementing a Route Finder

If we represent a street map as a graph

- what are the vertices?
- what are the edges?
- are edges directional?
- what are the weights?
- are the weights fixed?

What kind of algorithm would ...

- help us find the "quickest" way to get from A to B?

Weighted Graph Representation

Weights can easily be added to:

- adjacency matrix representation (0/1 → int or float)
- adjacency lists representation (add int/float to list node)

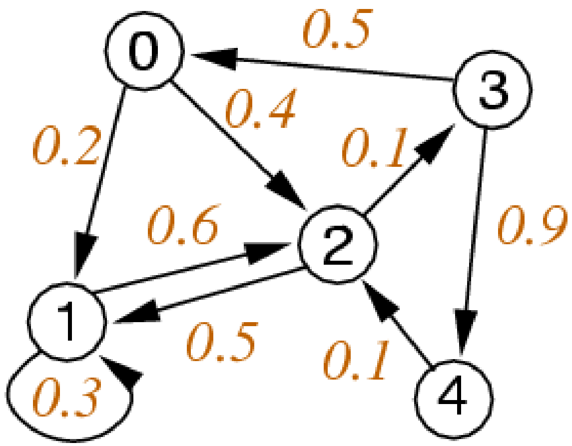
An alternative representation useful in this context:

- edge list representation (list of *(s,t,w)* triples)

All representations work whether edges are directed or not.

Weighted Graph Representation (cont)

Adjacency matrix representation with weights:



Weighted Digraph

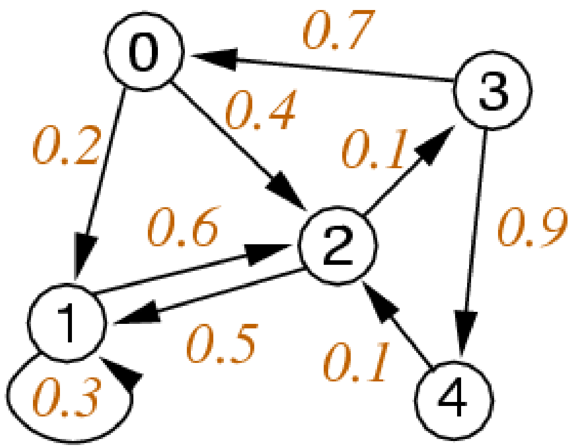
	0	1	2	3	4
0	*	0.2	0.4	*	*
1	*	0.3	0.6	*	*
2	*	0.5	*	0.1	*
3	0.5	*	*	*	0.9
4	*	*	0.1	*	*

Adjacency Matrix

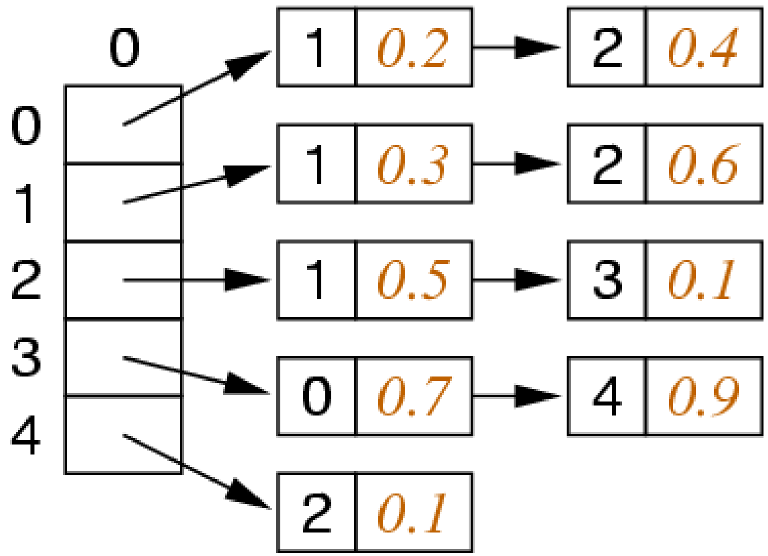
Note: need distinguished value to indicate "no edge".

Weighted Graph Representation (cont)

Adjacency lists representation with weights:



Weighted Digraph

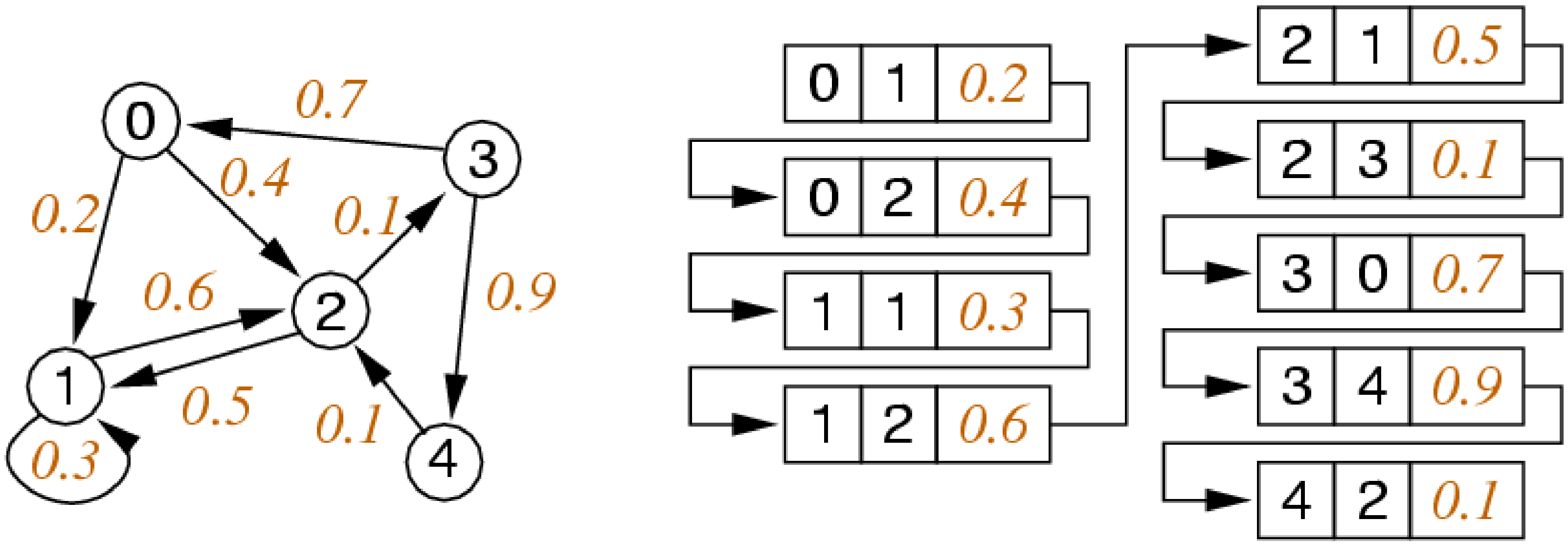


Adjacency Lists

Note: if undirected, each edge appears twice with same weight

Weighted Graph Representation (cont)

Edge array / edge list representation with weights:



Weighted Digraph *Edge List*

Note: not very efficient for use in processing algorithms, but does give a possible representation for min spanning trees or shortest paths

Weighted Graph Representation (cont)

Sample adjacency matrix implementation in C requires minimal changes to previous Graph ADT:

WGraph.h

```
// edges are pairs of vertices (end-points) plus positive weight
typedef struct Edge {
   Vertex v;
   Vertex w;
   int    weight;
} Edge;

// returns weight, or 0 if vertices not adjacent
int adjacent(Graph, Vertex, Vertex);
```

Weighted Graph Representation (cont)

WGraph.c

```
typedef struct GraphRep {
   int **edges;   // adjacency matrix storing positive weights
                  // 0 if nodes not adjacent
   int   nV;      // #vertices
   int   nE;      // #edges
} GraphRep;


void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));
   if (g->edges[e.v][e.w] == 0) {   // edge e not in graph
      g->edges[e.v][e.w] = e.weight;
      g->edges[e.w][e.v] = e.weight;
      g->nE++;
   }
}

int adjacent(Graph g, Vertex v, Vertex w) {
   assert(g != NULL && validV(g,v) && validV(g,w));
   return g->edges[v][w];
}
```

Minimum Spanning Trees

Minimum Spanning Trees

Reminder: Spanning tree *ST* of graph *G(V,E)*

- spanning = all vertices, tree = no cycles
- *ST* is a subgraph of *G* ($G'=(V,E')$ where $E' \subseteq E$)
- *ST* is connected and acyclic

Minimum spanning tree *MST* of graph *G*

- *MST* is a spanning tree of *G*
- sum of edge weights is no larger than any other ST

Applications: Computer networks, Electrical grids, Transportation networks …

Problem: how to (efficiently) find MST for graph *G*?

NB: MST may not be unique (e.g. all edges have same weight ⇒ every ST is MST)

Minimum Spanning Trees (cont)

Brute force solution:

```
findMST(G):
   Input  graph G
   Output a minimum spanning tree of G

   bestCost=∞
   for all spanning trees t of G do
      if cost(t)<bestCost then
         bestTree=t
         bestCost=cost(t)
      end if
   end for
   return bestTree
```

Example of *generate-and-test* algorithm.

Not useful because #spanning trees is potentially large (e.g. n^{n-2} for a complete graph with n vertices)

Minimum Spanning Trees (cont)

Simplifying assumption:

- edges in *G* are not directed (MST for digraphs is harder)

Kruskal's Algorithm

One approach to computing MST for graph *G* with *V* nodes:

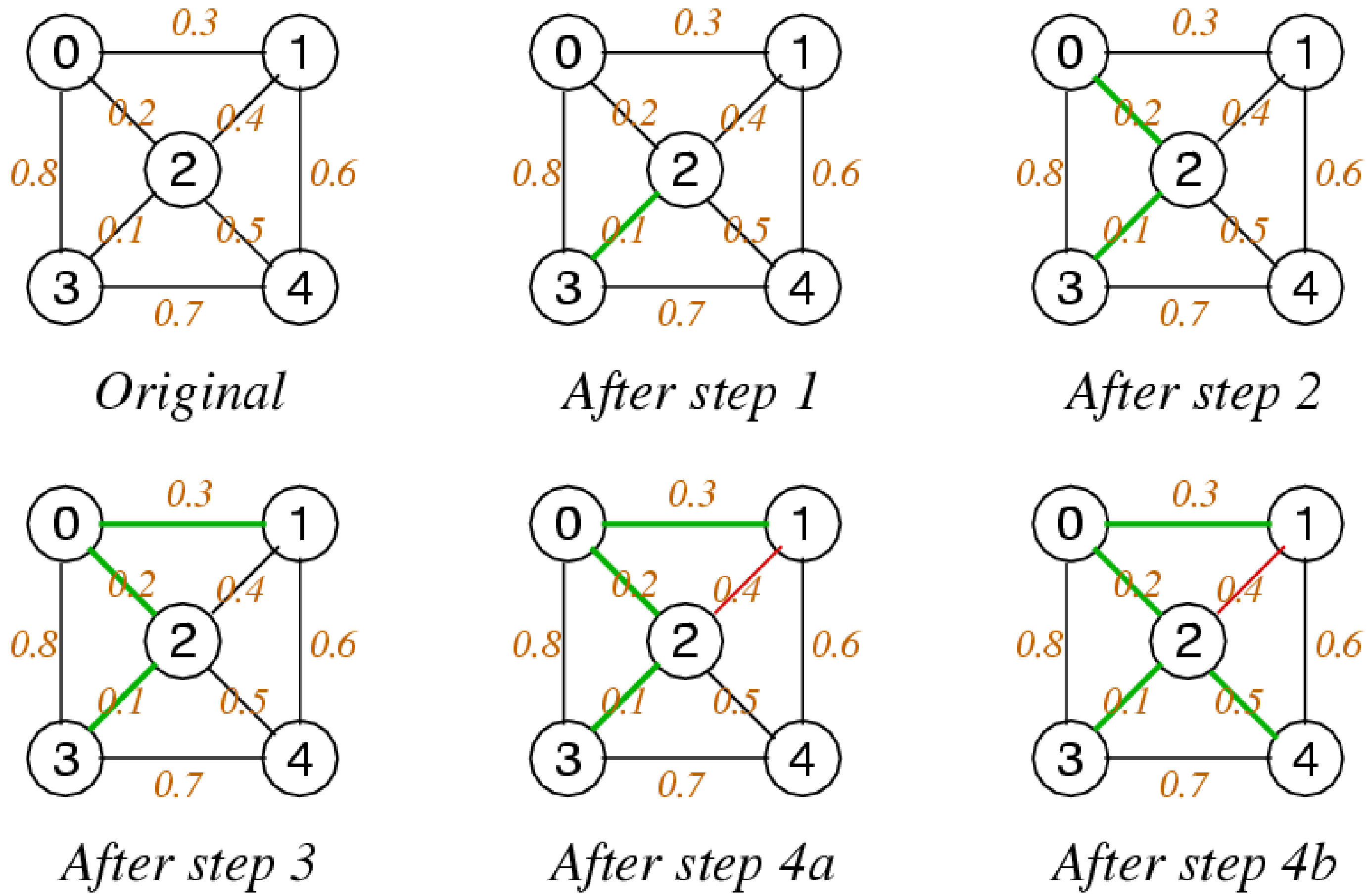
1. start with empty MST
2. consider edges in increasing weight order
    - add edge if it does not form a cycle in MST
3. repeat until *V-1* edges are added

Critical operations:

- iterating over edges in weight order
- checking for cycles in a graph

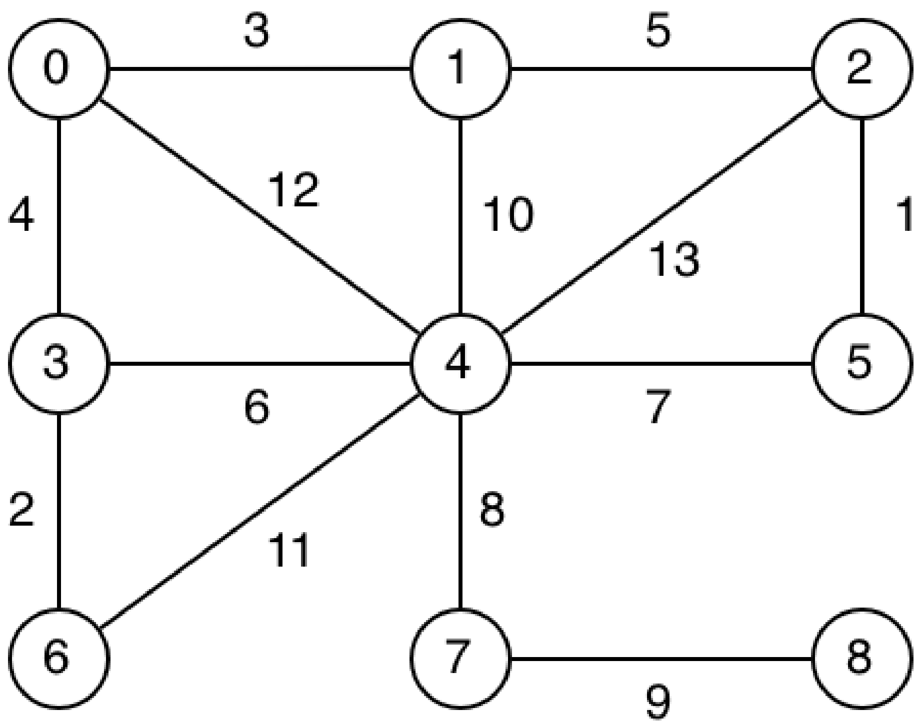
Kruskal's Algorithm (cont)

Execution trace of Kruskal's algorithm:

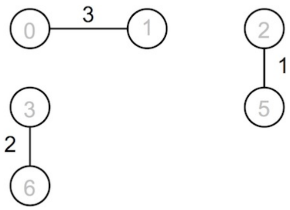


Exercise #2: Kruskal's Algorithm

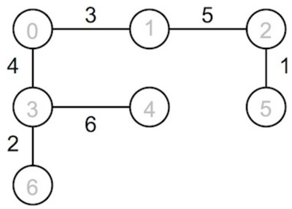
Show how Kruskal's algorithm produces an MST on:



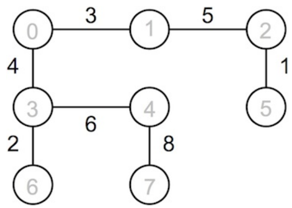
After 3rd iteration:



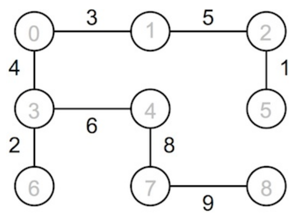
After 6th iteration:



After 7th iteration:



After 8th iteration (V-1=8 edges added):



Kruskal's Algorithm (cont)

Pseudocode:

```
KruskalMST(G):
    Input  graph G with n nodes
    Output a minimum spanning tree of G

    MST=empty graph
    sort edges(G) by weight
    for each e∈sortedEdgeList do
        MST = MST ∪ {e}
        if MST has a cyle then
            MST = MST \ {e}
        end if
        if MST has n-1 edges then
            return MST
        end if
    end for
```

Kruskal's Algorithm (cont)

Rough time complexity analysis …

- sorting edge list is $O(E \cdot \log E)$
- at least V iterations over sorted edges
- on each iteration …
  - getting next lowest cost edge is $O(1)$
  - checking whether adding it forms a cycle: cost = ??

Possibilities for cycle checking:

- use DFS … too expensive?
- could use *Union-Find data structure* (see Sedgewick Ch.1)

Prim's Algorithm

Another approach to computing MST for graph *G=(V,E)*:

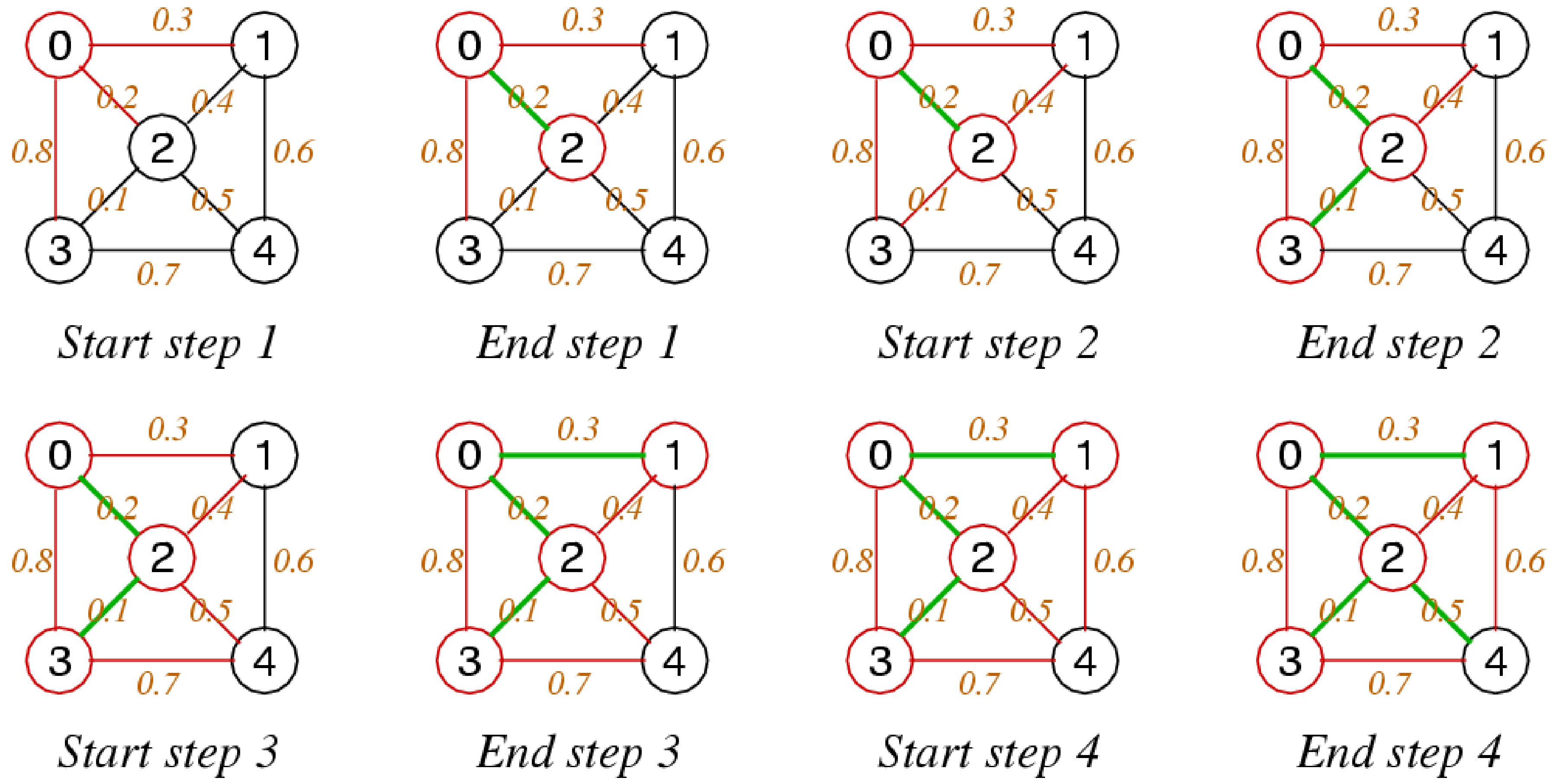
1. start from any vertex *s* and empty MST
2. choose edge not already in MST to add to MST
   - must be incident on a vertex already connected to *s* in MST
   - must have minimal weight of all such edges
3. repeat until MST covers all vertices

Critical operations:

- checking for vertex being connected in a graph
- finding min weight edge in a set of edges

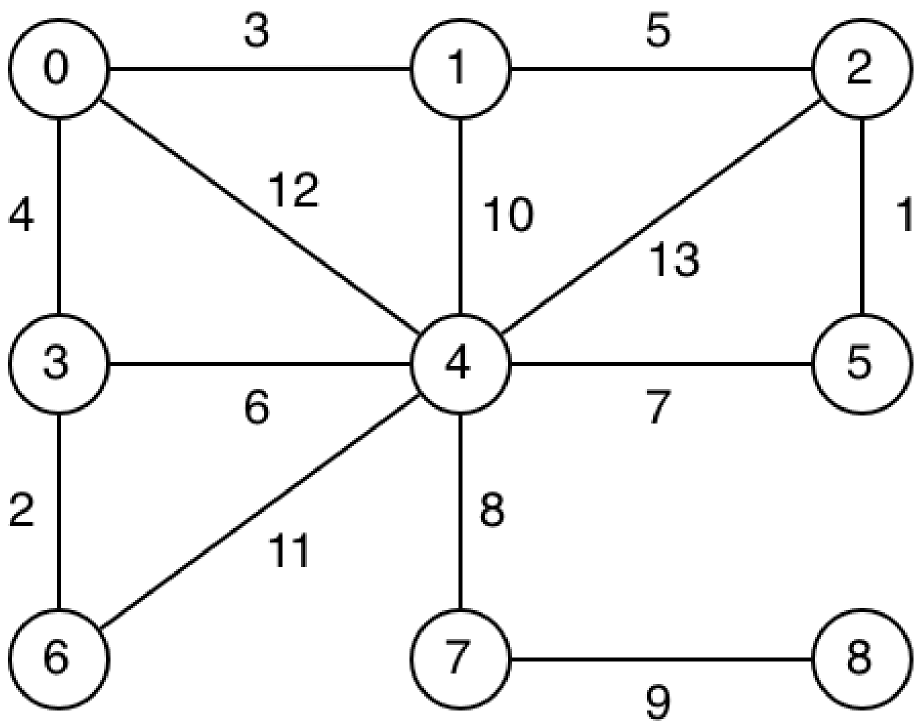
Prim's Algorithm (cont)

Execution trace of Prim's algorithm (starting at s=0):



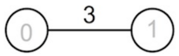
Exercise #3: Prim's Algorithm

Show how Prim's algorithm produces an MST on:

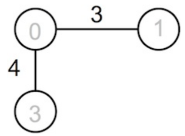


Start from vertex 0

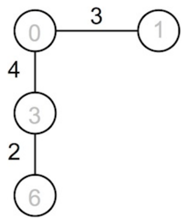
After 1st iteration:



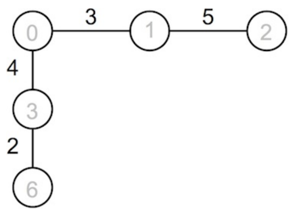
After 2nd iteration:



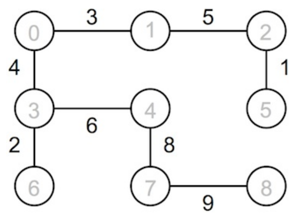
After 3rd iteration:



After 4th iteration:



After 8th iteration (all vertices covered):



Prim's Algorithm (cont)

Pseudocode:

```
PrimMST(G):
   Input  graph G with n nodes
   Output a minimum spanning tree of G

   MST=empty graph
   usedV={0}
   unusedE=edges(g)
   while |usedV|<n do
      find e=(s,t,w)∈unusedE such that {
         s∈usedV ∧ t∉usedV ∧ w is min weight of all such edges
      }
      MST = MST ∪ {e}
      usedV = usedV ∪ {t}
      unusedE = unusedE \ {e}
   end while
   return MST
```

Critical operation: finding best edge

Prim's Algorithm (cont)

Rough time complexity analysis …

- *V* iterations of outer loop
- in each iteration …
  - find min edge with set of edges is $O(E) \Rightarrow O(V \cdot E)$ overall
  - find min edge with priority queue is $O(\log E) \Rightarrow O(V \cdot \log E)$ overall

Note:

- Using a *priority queue* gives a variation of DFS (stack) and BFS (queue) graph traversal

Sidetrack: Priority Queues

Some applications of queues require

- items processed in order of "key"
- rather than in order of entry (FIFO — first in, first out)

Priority Queues (PQueues) provide this via:

- **`join`**: insert item into PQueue (replacing **`enqueue`**)
- **`leave`**: remove item with largest key (replacing **`dequeue`**)

Sidetrack: Priority Queues (cont)

Comparison of different Priority Queue representations:

	sorted array	unsorted array	sorted list	unsorted list
space usage	*MaxN*	*MaxN*	$O(N)$	$O(N)$
join	$O(N)$	$O(1)$	$O(N)$	$O(1)$
leave	$O(N)$	$O(N)$	$O(1)$	$O(N)$
is empty?	$O(1)$	$O(1)$	$O(1)$	$O(1)$

for a PQueue containing N items

Other MST Algorithms

Boruvka's algorithm ... complexity $O(E{\cdot}log\ V)$

- the oldest MST algorithm
- start with V separate components
- join components using min cost links
- continue until only a single component

Karger, Klein, and Tarjan ... complexity $O(E)$

- based on Boruvka, but non-deterministic
- randomly selects subset of edges to consider
- for the keen, here's the paper describing the algorithm

Shortest Path

Shortest Path

Path = sequence of edges in graph G $p = (v_0,v_1), (v_1,v_2), \ldots, (v_{m-1},v_m)$

cost(path) = sum of edge weights along path

Shortest path between vertices *s* and *t*

- a simple path *p(s,t)* where *s* = *first(p)*, *t* = *last(p)*
- no other simple path *q(s,t)* has *cost(q)* < *cost(p)*

Assumptions: weighted digraph, no negative weights.

Finding shortest path between two given nodes known as source-target SPP problem

Variations: single-source, all-pairs

Applications: robot navigation, routing in data networks, …

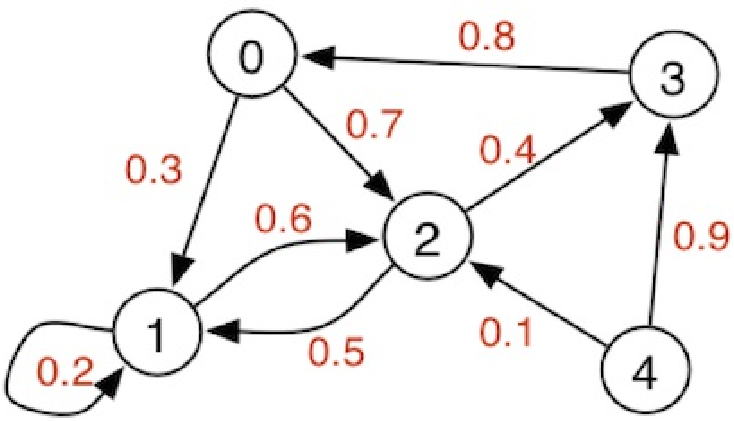
Single-source Shortest Path (SSSP)

Given: weighted digraph *G*, source vertex *s*

Result: shortest paths from *s* to all other vertices

- **`dist[]`** *V*-indexed array of cost of shortest path from *s*
- **`pred[]`** *V*-indexed array of predecessor in shortest path from *s*

Example:



	0	1	2	3	4
dist	0	0.3	0.7	1.1	inf
pred	-	0	0	2	-

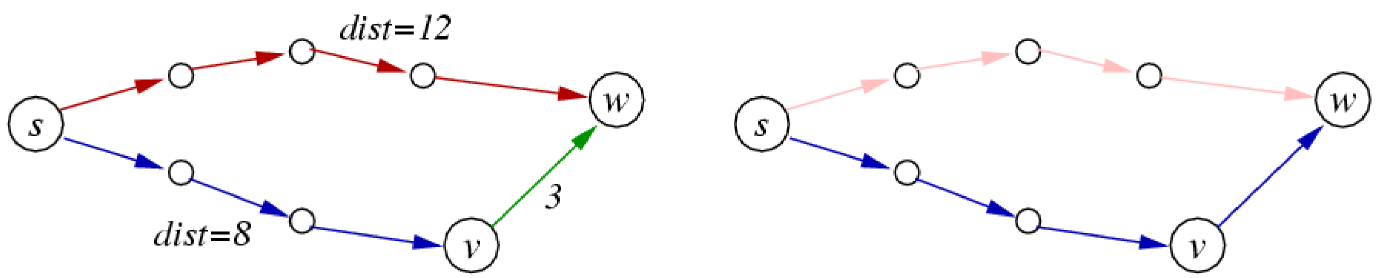
Shortest paths from s=0

Edge Relaxation

Assume: **`dist[]`** and **`pred[]`** as above (but containing data for shortest paths *discovered so far*)

`dist[v]` is length of shortest known path from *s* to *v*
`dist[w]` is length of shortest known path from *s* to *w*

Relaxation updates data for *w* if we find a shorter path from *s* to *w*:



`dist[v]=8, dist[w]=12`
`pred[v]=?, pred[w]=?`

`dist[v]=8, dist[w]=11`
`pred[v]=?, pred[w]=v`

Relaxation along edge *e*=**`(v,w,weight)`**:

- if **`dist[v]+weight < dist[w]`** then
  update **`dist[w]:=dist[v]+weight`** and **`pred[w]:=w`**

Dijkstra's Algorithm

One approach to solving single-source shortest path …

Data: *G*, *s*, **`dist[]`**, **`pred[]`** and

- *vSet*: set of vertices whose shortest path from *s* is unknown

Algorithm:

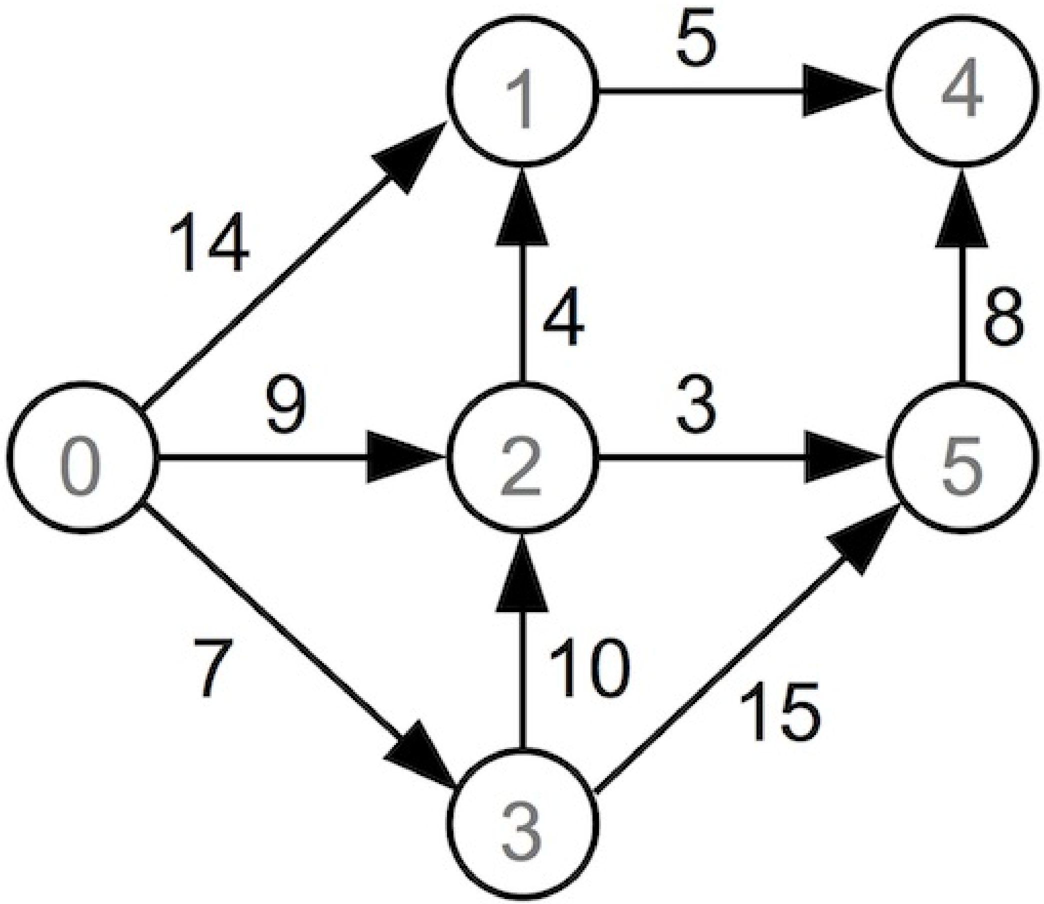
```
dist[]   // array of cost of shortest path from s
pred[]   // array of predecessor in shortest path from s

dijkstraSSSP(G,source):
|  Input graph G, source node
|
|  initialise dist[] to all ∞, except dist[source]=0
|  initialise pred[] to all -1

|  vSet=all vertices of G
|  while vSet≠∅ do
|  |  find s∈vSet with minimum dist[s]
|  |  for each (s,t,w)∈edges(G) do
|  |     relax along (s,t,w)
|  |  end for
|  |  vSet=vSet\{s}
|  end while
```

Exercise #4: Dijkstra's Algorithm

Show how Dijkstra's algorithm runs on (source node = 0):



	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	∞	∞	∞	∞	∞
pred	–	–	–	–	–	–

dist	0	14	9	7	∞	∞
pred	–	0	0	0	–	–

dist	0	14	9	7	∞	22
pred	–	0	0	0	–	3

dist	0	13	9	7	∞	12
pred	–	2	0	0	–	2

dist	0	13	9	7	20	12
pred	–	2	0	0	5	2

dist	0	13	9	7	18	12
pred	–	2	0	0	1	2

Dijkstra's Algorithm (cont)

Why Dijkstra's algorithm is correct:

Hypothesis.
(a) For visited *s* … *dist[s]* is shortest distance from source
(b) For unvisited *t* … *dist[t]* is shortest distance from source *via visited nodes*

Proof.
Base case: no visited nodes, *dist[source]=0*, *dist[s]=∞* for all other nodes

Induction step:

1. If *s* is unvisited node with minimum *dist[s]*, then *dist[s]* is shortest distance from source to *s*:
   - if ∃ shorter path via only visited nodes, then *dist[s]* would have been updated when processing the predecessor of *s* on this path
   - if ∃ shorter path via an unvisited node *u*, then *dist[u]<dist[s]*, which is impossible if *s* has min distance of all unvisited nodes
2. This implies that (a) holds for *s* after processing *s*
3. (b) still holds for all unvisited nodes *t* after processing *s*:
   - if ∃ shorter path via *s* we would have just updated *dist[t]*
   - if ∃ shorter path without *s* we would have found it previously

Dijkstra's Algorithm (cont)

Time complexity analysis …

Each edge needs to be considered once ⇒ $O(E)$.

Outer loop has $O(V)$ iterations.

Implementing "**`find s∈vSet with minimum dist[s]`**"

1. try all **`s∈vSet`** ⇒ cost = $O(V)$ ⇒ overall cost = $O(E + V^2) = O(V^2)$
2. using a PQueue to implement extracting minimum
   - can improve overall cost to $O(E + V \cdot log\ V)$ (for best-known implementation)

Summary

- Weighted graph representations
- Minimum Spanning Tree (MST)
  - Kruskal, Prim
- Single source shortest path problem
  - Dijkstra

- Suggested reading:
  - Sedgewick, Ch.19.3,20-20.4,21-21.3