# Graph Algorithms

§9.3.2 Dijkstra's

§9.5.1 Prim's

shortest path,
minimum spanning tree

- Code ends up being very similar
- Lab 12: use STL map<>
(differs from text!!!)

- Definition

$G = (V, E)$

graph G with vertices V and edges E

- each edge is a pair $(v, w) \mid v, w \in V$

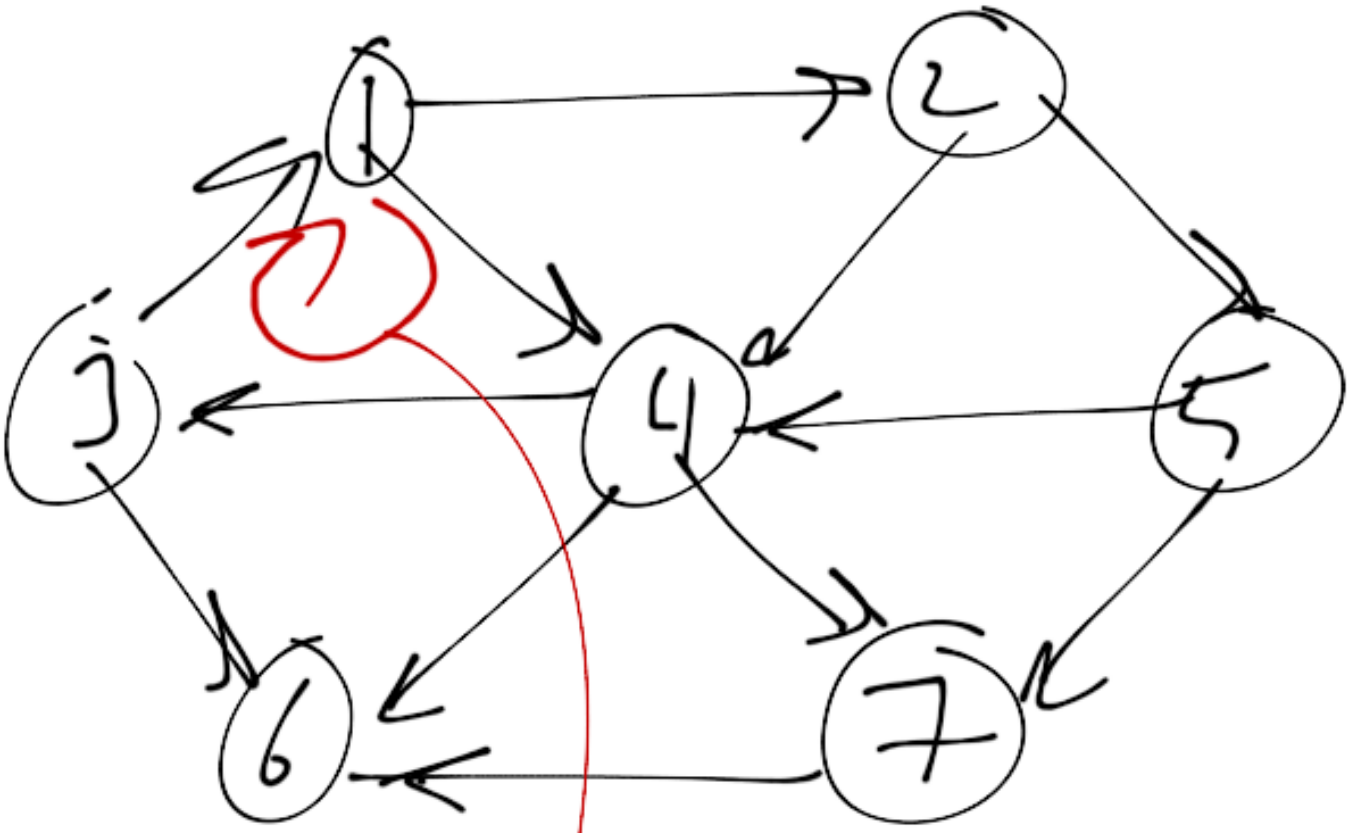
- if the pair is ordered, $(v, w) \neq (w, v)$

then graph is directed

- an edge expresses notion of adjacency

i.e. vertex w is adjacent to v iff $(v, w) \in E$

e.g. a digraph:




a cycle: graphs can have them, trees can not

- data structure (representation):

matrix

| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ | $V_7$ |
|---|---|---|---|---|---|---|---|
| $V_1$ | ∅ | 1 | 1 | 1 | ∞ | ∞ | ∞ |
| $V_2$ | ∞ | ∅ | ∞ | 1 | 1 | ∞ | ∞ |
| $V_3$ | 1 | ∞ | ∅ | ∞ | ∞ | 1 | 0 |
| $V_4$ | ∞ | ∞ | ∞ | ∅ | ∞ | 1 | 1 |
| $V_5$ | ∞ | ∞ | ∞ | ∞ | ∅ | ∞ | 1 |
| $V_6$ | ∞ | ∞ | ∞ | ∞ | ∞ | ∅ | ∞ |
| $V_7$ | ∞ | ∞ | ∞ | ∞ | ∞ | 1 | ∅ |

- edge weight or cost:
  if edge exists, assign
  unit (1) cost by
  default — often
  times, edges will have
  cost $> 1$.

- path: a sequence of
  vertices $w_1, w_2, \ldots w_n$
  s.t. $(w_i, w_{i+1}) \in E, i \in 1 \leq$
  $n$

- e.g., in graph, path from ① to ⑦ can be


$(v_1, v_4), (v_4, v_7) \in E$

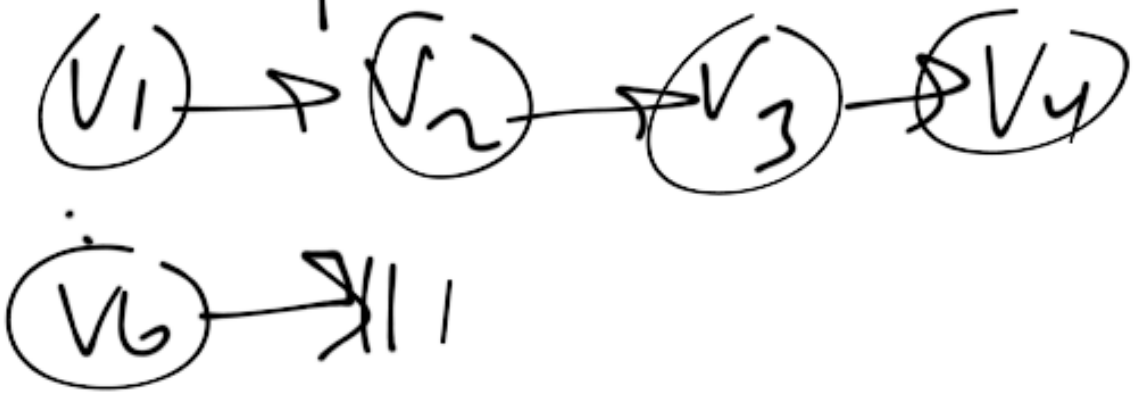
- cycle: path from some vertex $w_i$ s.t. it leads back to $w_i$: a path of length $\geq 1$ $|w_1 = w_n$

- A directed, acyclic graph is known as a DAG

- A complete graph is one where there is an edge between every pair of vertices (matrix is full)

- matrix representation is generally preferred for non-sparse graphs (many edge connections or complete graph)

- for sparse graphs, node-ptr rep. may be ok



- WARNING: text gives pseudo-code for algs in Lab 12, eg. Fig. 9.31, BUT they use previous idea of Decrkey (min heap) (cf. Fig. 9.18 where Decrkey appears to be absent)

- I will show you how to do Dijkstra's w/o t Decrkey

- Decrease changes a node's weight (key) like on minheap (not a good idea, IMO)

- I will use STL's priority_queue which has no decrease equivalent.

- STL solution relies on the use of associative array where each node's (vertice's) string

Key is used as index

— std::string

map<string, map<string, int>>

graph;

— space

each graph[string] is
an associative array,

e.g.,

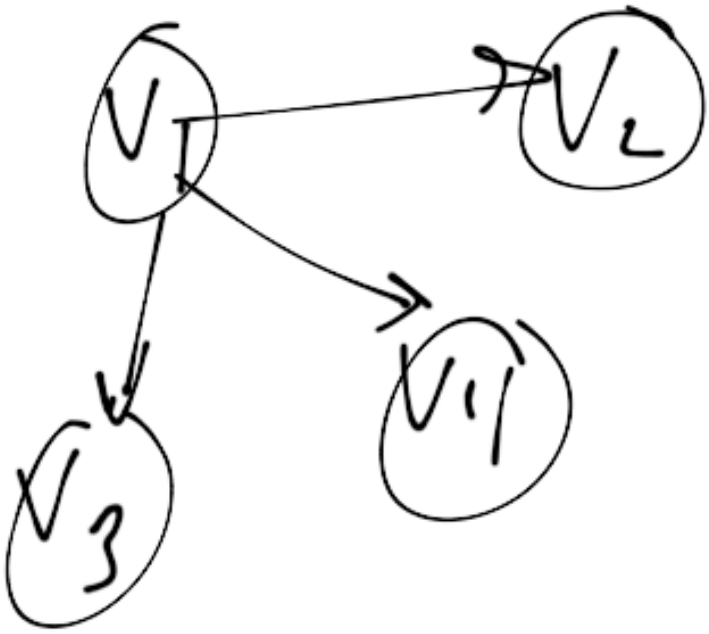
map<string, map<string, int>> g;

g["v1"]["v2"] = 1;
g["v1"]["v3"] = 1;
g["v1"]["v4"] = 1;

(think g["Greenville"]["Charleston"]
= miles)

- no need to store

  ∞ edges (edges in graph that don't exist)

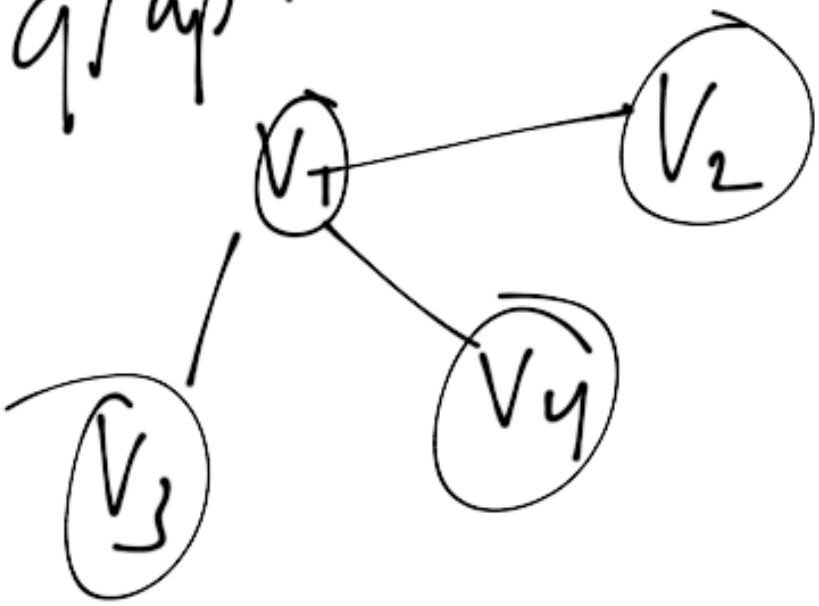



$g("v1")["v2"] = 1$

$g["v1"]["v4"] = 1$

$g("v1")("v3") = 1$

If this were an undirected graph




$g["V_1"]["V_2"] = 1$ $g["V_1"][V_1] = 1$

$g["V_1"]["V_2"] = 1$ $g[V_2][V_1] = 1$

$g["V_1"]["V_4"] = 1$

$g["V_3"]["V_1"] = 1$

- in practice we want to express graph nodes (vertices) as node_t:

```
class node_t {
    public:
        string id;
        int    cost;
        // constructor
        // operator=
```

```
bool operator<(
    const node_t & rhs)
{
    return cost < rhs.cost;
}
bool operator>(...)
  {...}
bool operator==()
  { return id == rhs.id;
  }
```

(Git

friend ostream&
operator<<( ... )
{ --- }

.

initialization: something like:
graph (node1: id) (node1. id)
= cost