CSE221 Data Structures
Lecture 21: Directed Graphs

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

November 29, 2021

Introduction

- Final exam is on Wednesday 15 December, 20:00–22:00.
- Assignment 4 will be posted tomorrow, due on Thursday next week.
- Assignment 3 will be graded by Seonghyeon Jue (`shjue@unist.ac.kr`)
- Today's lecture is on algorithms for *directed* graphs.
- Reference for this lecture: Textbook Chapter 13.4 and 13.5

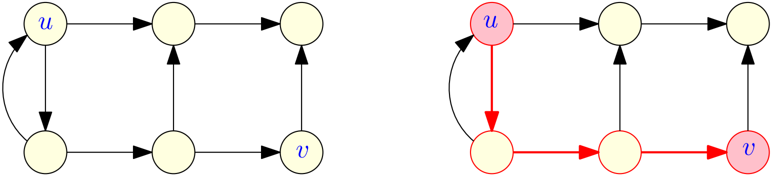
Data Structure

- In addition to the standard ADT functions, we will need the following:

- e.**isDirected**(): Test whether edge e is directed.
- e.**origin**(): Return the origin vertex of edge e.
- e.**dest**(): Return the destination vertex of edge e.
- **insertDirectedEdge**(v, w, x): Insert and return a new directed edge with origin v and destination w and storing element x.

- It allows us to deal with *directed graphs*, or *mixed graphs* (that have undirected and directed edges).

Reachability



Definition

Given two vertices u, v in a graph G, we say that u *reaches* v if there is a directed path from u to v. We also say that v is *reachable* from u. The *reachability* problem is to decide whether u reaches v.

Example

In the graph above, u reaches v, but u is no reachable from v.

Reachability

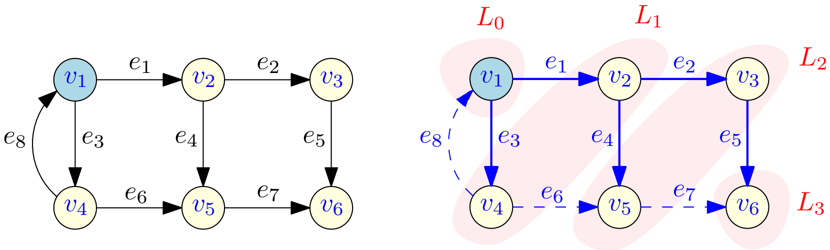
- This problem can be solved by depth-first search.

Pseudocode

```
procedure DirectedDFS(v)
    mark v as visited
    for each outgoing edge (v, w) of v do
        if vertex w has not been visited then
            recursively call DirectedDFS(w)
```

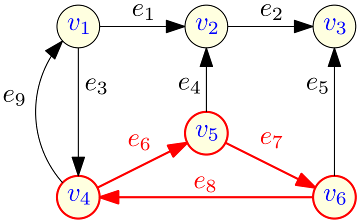
- DFS visits *all* the vertices that are reachable from v.
- It is the same algorithms as for undirected graphs, except that we only move from the origin to the destination of an edge.
- It still runs in $O(n + m)$ time where n is the number of vertices and m is the number of edges. So we can solve the reachability problem in $O(n + m)$ time.

Reachability



- We can also run BFS on a directed graph in $O(n + m)$ time.
- It creates three types of edges: Tree edges, cross edges and back edges. Cross edges connect a vertex to a vertex that is neither its ancestor nor its descendent Back edges that connect a vertex to one of its ancestors.
- In the example above, BFS produces one back edge (e_8) and two cross edges (e_6 and e_7).

Cycle Detection



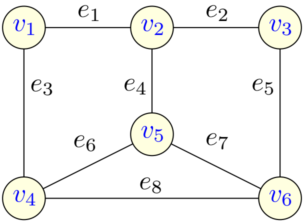
Problem

Given a directed graph G, the cycle detection problem is to find a directed cycle in G, if there is at least one.

Example

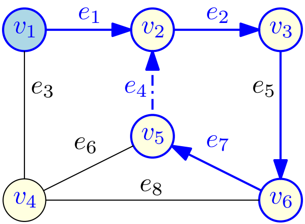
In the graph above, $(v_4, e_6, v_5, e_7, v_6, e_8, v_4)$ is a cycle, so a cycle detection algorithm could just return this cycle. There is another cycle $(v_1, e_3, v_4, e_9, v_1)$, so the algorithm could have returned this one instead.

Cycle Detection



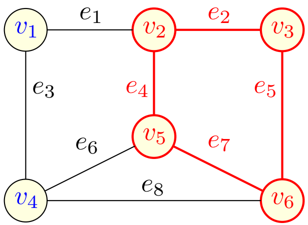
- For *undirected* graphs, we saw in the previous lecture that it suffices to run DFS, and if we encounter a previously visited vertex at Line 4, Slide 6, then we the edge under consideration closes a cycle.

Cycle Detection



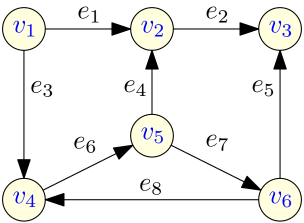
- For *undirected* graphs, we saw in the previous lecture that it suffices to run DFS, and if we encounter a previously visited vertex at Line 4, Slide 6, then we the edge under consideration closes a cycle.

Cycle Detection



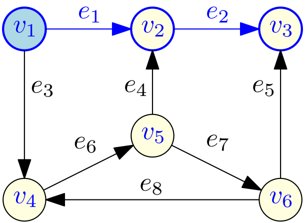
- For *undirected* graphs, we saw in the previous lecture that it suffices to run DFS, and if we encounter a previously visited vertex at Line 4, Slide 6, then we the edge under consideration closes a cycle.

Cycle Detection



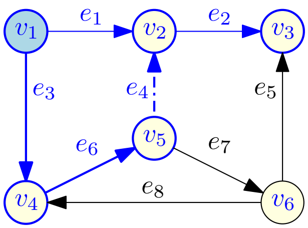
- This approach does not work for *directed* graphs.

Cycle Detection



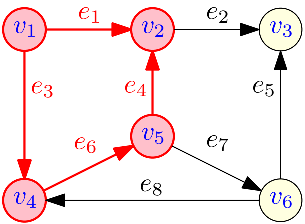
- This approach does not work for *directed* graphs.

Cycle Detection



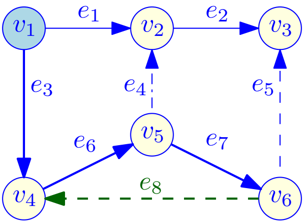
- This approach does not work for *directed* graphs.

Cycle Detection



- This is not a directed cycle.

Cycle Detection

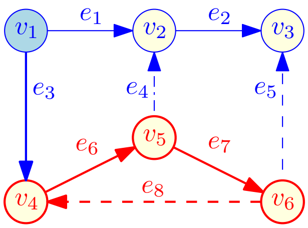


- A correct approach: Perform directed DFS.
- If a *back edge* is found, then it closes a cycle. (A back edge is an edge whose destination is an ancestor of its origin in the spanning tree.)
- If no back edge is found, then there is no directed cycle.

Example

In the DFS execution above, e_8 is a back edge, and it closes the cycle (e_6, e_7, e_8).

Cycle Detection

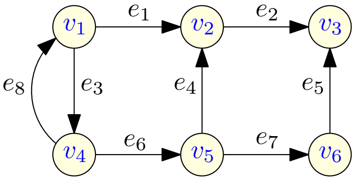


- A correct approach: Perform directed DFS.
- If a *back edge* is found, then it closes a cycle. (A back edge is an edge whose destination is an ancestor of its origin in the spanning tree.)
- If no back edge is found, then there is no directed cycle.

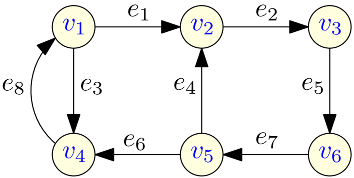
Example

In the DFS execution above, e_8 is a back edge, and it closes the cycle (e_6, e_7, e_8).

Strong Connectivity



G



H

Definition

A directed graph $G = (V, E)$ is strongly connected if u reaches v and v reaches u for every $u, v \in V$.

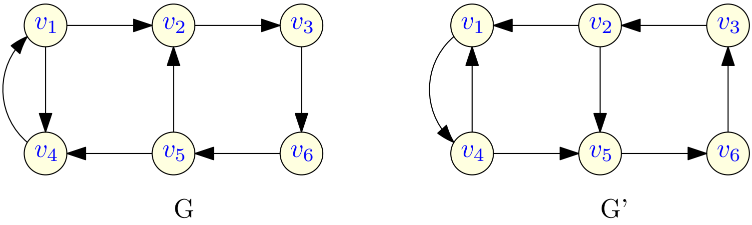
Examples

G is *not* strongly connected because v_1 is not reachable from v_2.
H is strongly connected.

Strong Connectivity

- How can we check whether a graph $G(V, E)$ is strongly connected?
- Check whether DFS(v) reaches all the vertices in V for each $v \in V$.
- Running time: $O(n(m + n))$.
- A better approach: (see next slide)

Strong Connectivity



Strong Connectivity Algorithm

- Choose a vertex $s \in V$
- If DFS(G, s) does not reach all the nodes, return FALSE.
- Let G' be the graph obtained by reversing all the edges in G.
- If DFS(G', s) does not reach all the nodes, return FALSE.
- Otherwise return TRUE.

- This algorithm runs in time $O(m + n)$.

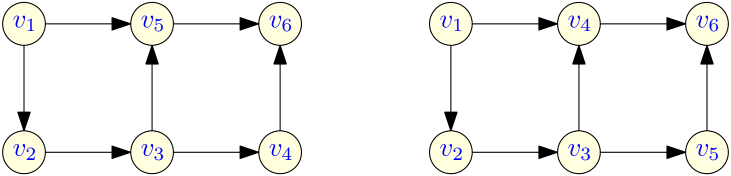
Directed Acyclic Graphs

Definition

A *directed acyclic graph (DAG)* is a directed graph with no directed cycle.

Definition

A *topological ordering* of a directed graph $G = (V, E)$ is an ordering $\{v_1, \dots, v_n\}$ of its vertices such that $i < j$ whenever $(v_i, v_j) \in E$.



Two topological orderings of the same DAG.

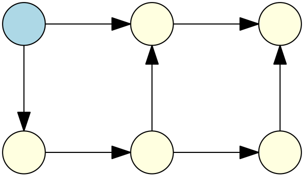
Directed Acyclic Graphs

Proposition

A directed graph G has a topological ordering if and only if it is acyclic.

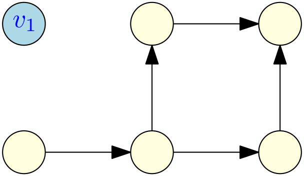
- We need to prove two statements:
  - $\Leftarrow$ If G has a topological ordering, then it is acyclic.
  - $\Rightarrow$ If G is acyclic, then it has a topological ordering.
- Proof of $\Leftarrow$: done in class.
- Proof of $\Rightarrow$: We make a constructive proof by showing how to construct a topological ordering of G.

Directed Acyclic Graphs

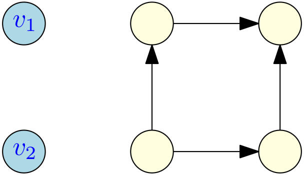


- G must have at least one vertex of indegree 0 because:
- If it were not the case, we could trace a path backwards from a starting vertex, and we would visit the same vertex twice. So G would have a directed cycle.
- We let v_1 be this vertex, and we delete this vertex from G.

Directed Acyclic Graphs

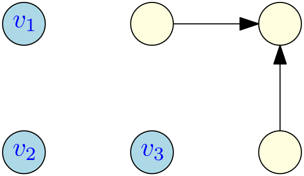


- We remove v_1 from G, and the resulting graph must still be acyclic.

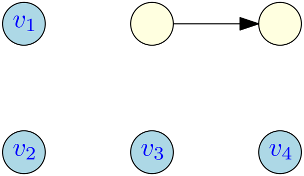


- We find a vertex v_2 of indegree 0 in this new graph, and repeat the process.

Directed Acyclic Graphs

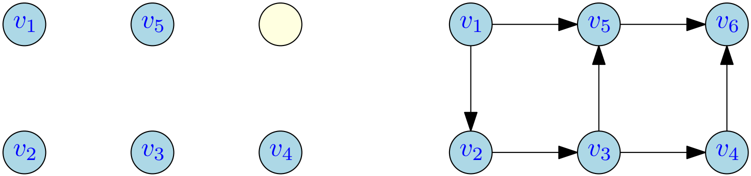


- Now there are two choices for v_4. We choose the bottom vertex.



- We find a vertex v_2 of indegree 0 in this new graph, and repeat the process.

Directed Acyclic Graphs



- In the end, we find a topological ordering of G.
- We just showed how to find a topological ordering for any graph G.
- It also gives us an algorithm to construct it.
- We call this algorithm *topological sorting*.

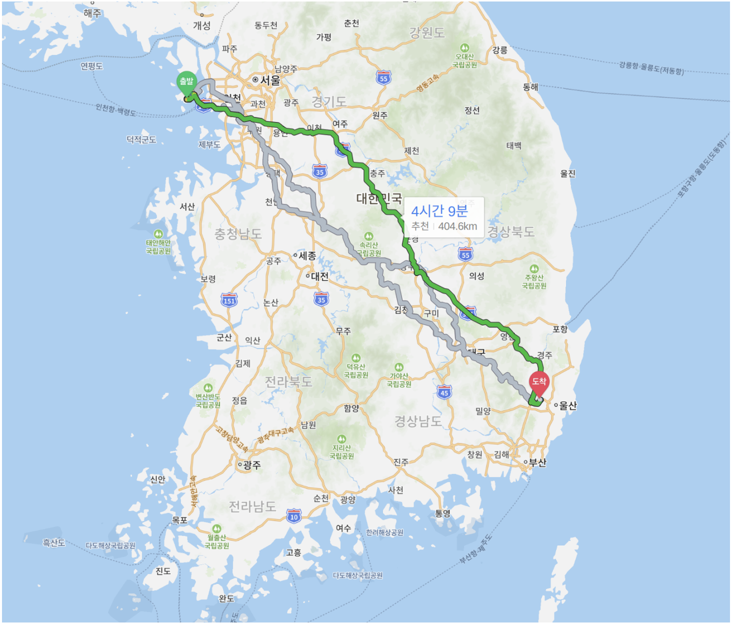
Topological Sorting

procedure $\text{TopologicalSort}(G)$
    $S \leftarrow$ an empty stack
    for all u in G.vertices() **do**
        incounter(u) $\leftarrow \text{indeg}(u)$
        if incounter(u)$=0$ **then**
            S.push(u)
    $i \leftarrow 1$
    while $S \neq \emptyset$ **do**
        $u \leftarrow S$.pop()
        $v_i \leftarrow u$
        $i \leftarrow i + 1$
        for all outgoing edges (u, w) of u **do**
            incounter(w) $\leftarrow$ incounter(w)-1
            if incounter(w)$=0$ **then**
                S.push(w)

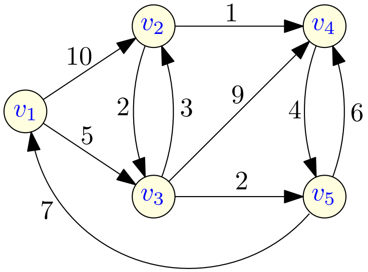
Topological Sorting

- We can initialize the incounter() variables in $O(n+m)$ time overall by performing a traversal of G.
- Other than that, we make $O(m+n)$ stack operations.
- So TopologicalSort runs in $O(m+n)$ time.
- Remark: This algorithm also allows us to decide whether G is acyclic. Indeed, if it fails to sort all the vertices, then the graph must have a cycle.
- Application: A number of tasks have to be performed.
- We represent them as a graph, with an edge (i,j) if task i needs to be completed before task j.
- Then a topological ordering gives you a possible order in which the tasks can be performed.

Shortest Paths

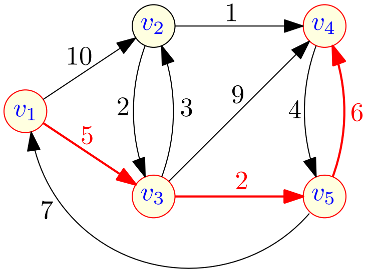


Weighted Graphs



- A *weighted graph* is a graph that has a numeric (for example, integer) label $w(e)$ associated with each edge e, called the *weight* of edge e.

Weighted Graphs

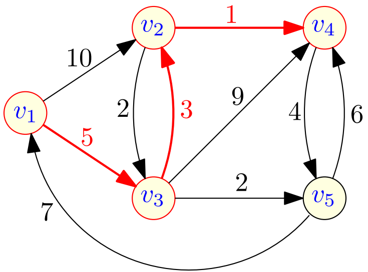


- The *length* of a path is the sum of the lengths of the edges of the path.

Example

The path shown in the graph above has length 13.

Shortest Paths

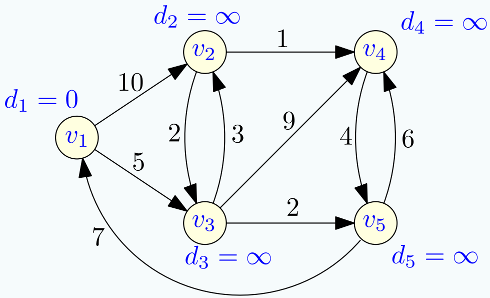


- The *distance* from a vertex v to a vertex u in G, denoted $d(v, u)$, is the length of a minimum length path (also called *shortest path*) from v to u, if such a path exists.

Example

In the graph above, $d(v_1, v_4) = 9$.

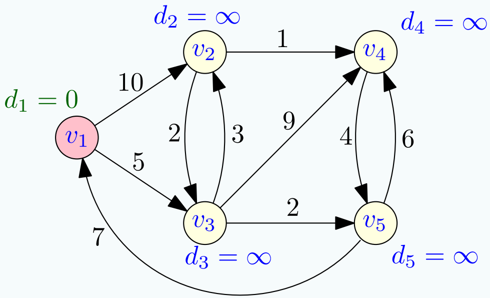
Dijkstra's Algorithm



$Q = (v_1, v_2, v_3, v_4, v_5)$

priority queue with d-values as keys

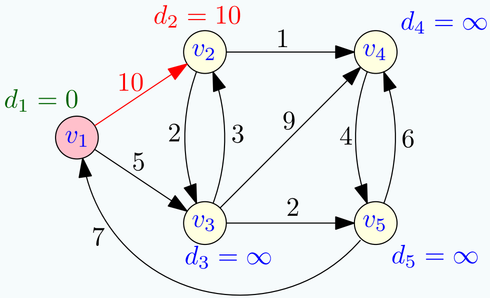
Dijkstra's Algorithm



$Q = (v_2, v_3, v_4, v_5)$

remove the minimum v_1 from Q

Dijkstra's Algorithm

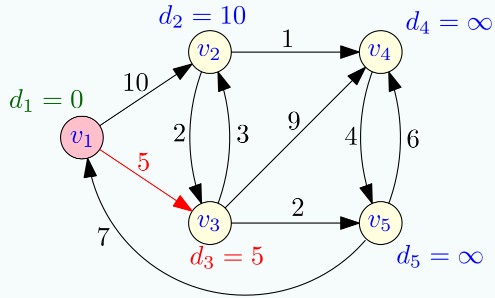


$Q = (v_2, v_3, v_4, v_5)$

relax the edge (v_1, v_2):

$d_2 \leftarrow \min(d_2, d_1 + 10)$

Dijkstra's Algorithm

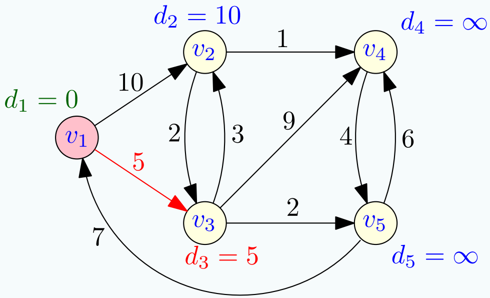


$Q = (v_2, v_3, v_4, v_5)$

relax the edge (v_1, v_3):

$d_3 \leftarrow \min(d_3, d_1 + 5)$

Dijkstra's Algorithm

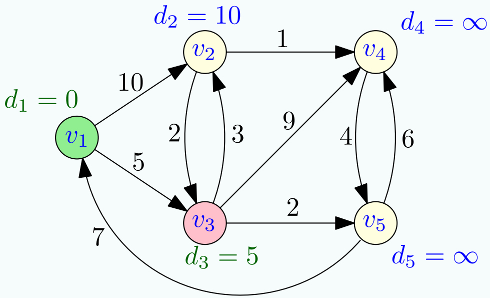


$Q = (v_3, v_2, v_4, v_5)$

relax the edge (v_1, v_3):

$d_3 \leftarrow \min(d_3, d_1 + 5)$

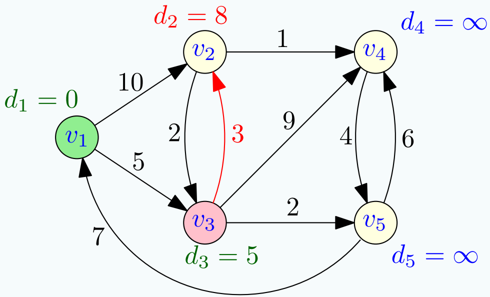
Dijkstra's Algorithm



$Q = (v_2, v_4, v_5)$

remove the minimum v_3 from Q

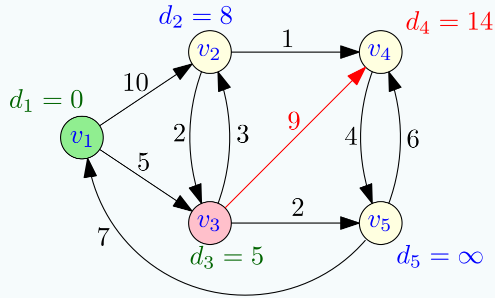
Dijkstra's Algorithm



$Q = (v_2, v_4, v_5)$

relax edge (v_2, v_3)

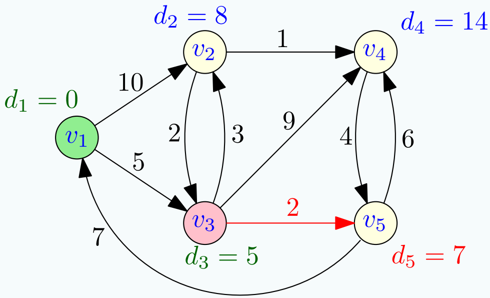
Dijkstra's Algorithm



$Q = (v_2, v_4, v_5)$

relax edge (v_3, v_4)

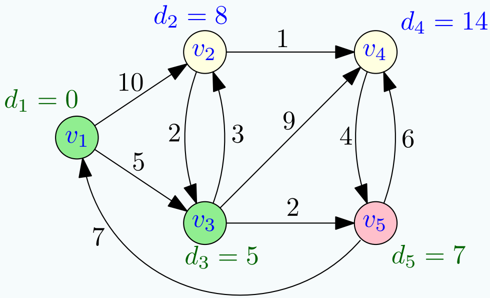
Dijkstra's Algorithm



$Q = (v_5, v_2, v_4)$

relax edge (v_3, v_5)

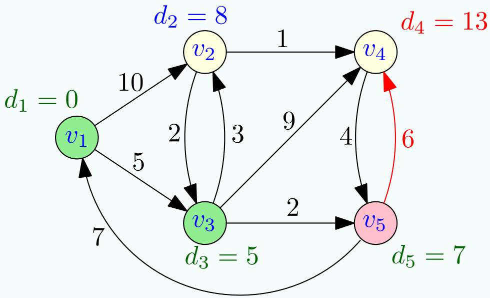
Dijkstra's Algorithm



$Q = (v_2, v_4)$

remove minimum v_5 from Q

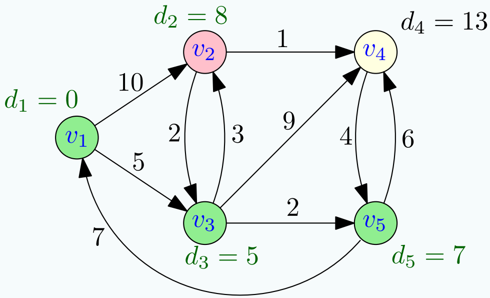
Dijkstra's Algorithm



$Q = (v_2, v_4)$

relax edge (v_5, v_4)

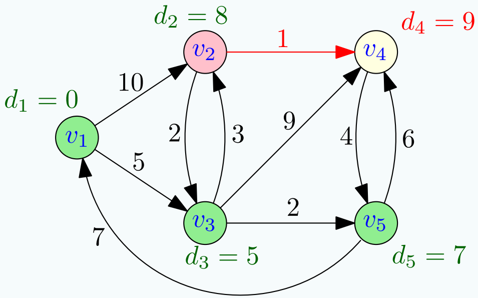
Dijkstra's Algorithm



$Q = (v_4)$

remove minimum v_2 from Q

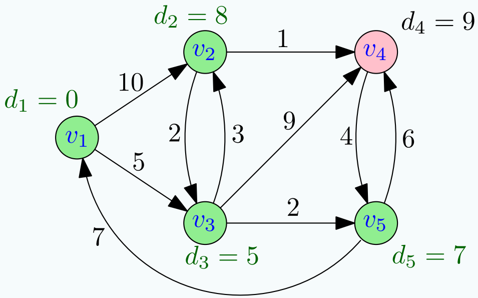
Dijkstra's Algorithm



$Q = (v_4)$

relax edge (v_2, v_4)

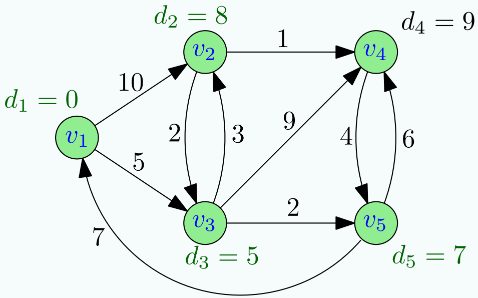
Dijkstra's Algorithm



$Q = ()$

remove minimum v_4 from Q

Dijkstra's Algorithm



$Q = ()$

All the distances from v_1 have been computed.

Dijkstra's Algorithm

- *Dijkstra's algorithm* solves the *single-source* shortest path problem: It computes the distance from a vertex s to all the other vertices in G.
- It requires the weights to be non-negative. It works for both directed and undirected graphs.
- For each vertex u, it maintains the length $D[u]$ of the shortest path to u that we have found *so far*. Initially, $D[s] = 0$ and $D[u] = \infty$ for all $u \neq s$.
- We maintain a set C of vertices (called the *cloud*) which is initially empty. At each iteration of the main loop, we insert the vertex $u \notin C$ with the smallest label $D[u]$ into C.

Dijkstra's Algorithm

- Then we update the labels of the vertices z adjacent to u and that are not in C using an *edge relaxation* operation:

Relaxation of edge (u, z)

if $D[u] + w((u, z)) < D[z]$ **then**
$D[z] \leftarrow D[u] + w((u, z))$

- The vertices that are not in the cloud C are maintained in a priority queue Q, with the labels $D[u]$ as keys.

Dijkstra's Algorithm

Pseudocode

```
procedure ShortestPath(G,s)
    set D[s] ← 0 and D[u] ← ∞ for all u ≠ s
    let Q be a priority queue recording all the vertices u with keys D[u]
    while Q ≠ ∅ do
        u ← Q.removeMin()
        for each edge (u, z) such that z ∉ Q do
            if D[u] + w((u, z) ⩽ D[z] then
                D[z] ← D[u] + w((u, z))
                change to D[z] the key of z in Q
    return the label D[u] of each vertex u
```

Analysis

- Each vertex is removed exactly once from Q, so there are n iterations of the **while** loop.
- So we make n removeMin() operations, which takes $O(n \log n)$ time overall if Q is implemented by a heap.
- Each edge is relaxed at most once, so there are m iterations of the **for** loop.
- Each iteration of the for loop takes $O(\log n)$ time for updating the key of z.

Proposition

Dijkstra's algorithm solves the single-source shortest path problem for a graph with non-negative weights in time $O((m+n)\log n)$.

Remarks

- At each iteration, Dijkstra's algorithm picks the node that is closest to s.
- This is an example of a *greedy algorithm*.
- More generally, greedy algorithms repeatedly select the best choice available at each iteration.
- The proof of correctness is non-trivial. I do not give it due to lack of time. See CSE331: Introduction to Algorithms.
- The pseudocode above does not show how to compute a shortest path: It only computes the distance from s to u.
- In order to find a shortest path from s to v, we can simply remember after each relaxation operation the node u from which the shortest path to z came. (See next slide.)
- Then we can recover the path by following the references prev[z] backwards from z.

Dijkstra's Algorithm

Pseudocode

procedure SHORTESTPATH(G,s)
  set $D[s] \leftarrow 0$ and $D[u] \leftarrow \infty$ for all $u \in V \setminus \{s\}$
  set prev[u] $\leftarrow$ NULL for all $u \in V$
  let Q be a priority queue recording all the vertices u with keys $D[u]$
  while $Q \neq \emptyset$ **do**
    $u \leftarrow Q$.removeMin()
    for each edge (u, z) such that $z \notin Q$ **do**
      if $D[u] + w((u, z)) \leqslant D[z]$ **then**
        $D[z] \leftarrow D[u] + w((u, z))$
        change to $D[z]$ the key of z in Q
        prev[z] $\leftarrow u$
  return $D[u]$ and prev[u] for each vertex u