CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 22

April 29, 2014

Graph Traversal

- DFS (Depth-first search) – scanning a graph depth-first using a pre-order like traversal.
- BFS (Breadth-first-search) – scanning a graph by going over the immediate neighbors of the source node, then their immediate neighbors etc. Finds the hop distance from the source node to any node in the graph (infinity if no path exists).
- Unweighted single source shortest path – using BFS to find the minimum number of hops from a given source node to any node in the graph.

Unweighted Single-Source Shortest-path Problem (SSSP)

- Find the shortest path (measured by number of edges) from a designated vertex S to every vertex.
- A special case of the weighted shortest-path problem with all weights=1.
- Has a more efficient solution.

Theory

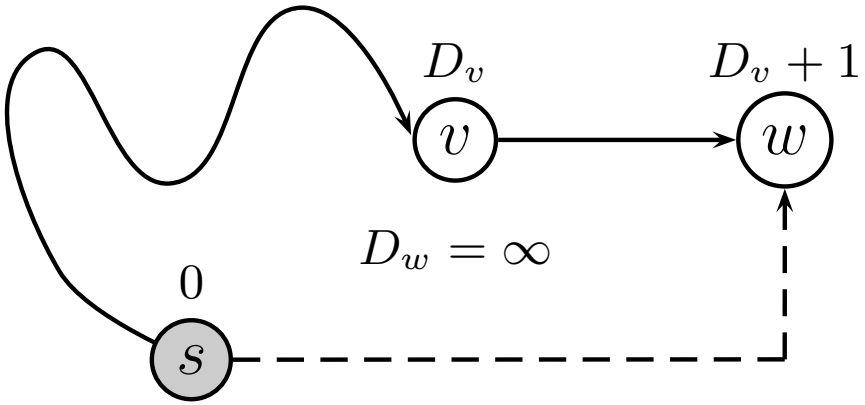
- Use a breadth-first search (BFS) strategy
  - Process vertices in layers
  - Closest to the start are evaluated first
  - Those most distant are evaluated last
- The shortest path from S to V_2 is a path of length 0.
- Start looking for all vertices that are distance 1 from S.
- Then we find each vertex whose shortest path from S is 2 ...

Single Source Shortest Path (SSSP)

- We don't want to search the whole graph looking for the next nodes to work on.
- It's clear that the nodes marked for the first time are the ones that soon should be re-examined for the next hop.
- To be efficient, we can save them on a queue, and then pull them off the queue for the next hop
- We will see these are just the nodes in BFS visitation order.

Search for the Shortest Path

- D_v = cost of a path to vertex v
- $D_w = D_v + 1$ if v is adjacent to w
- Starting with $Dw = \infty$, we can get to w by following a path to v and extending the path by the edge (v,w)

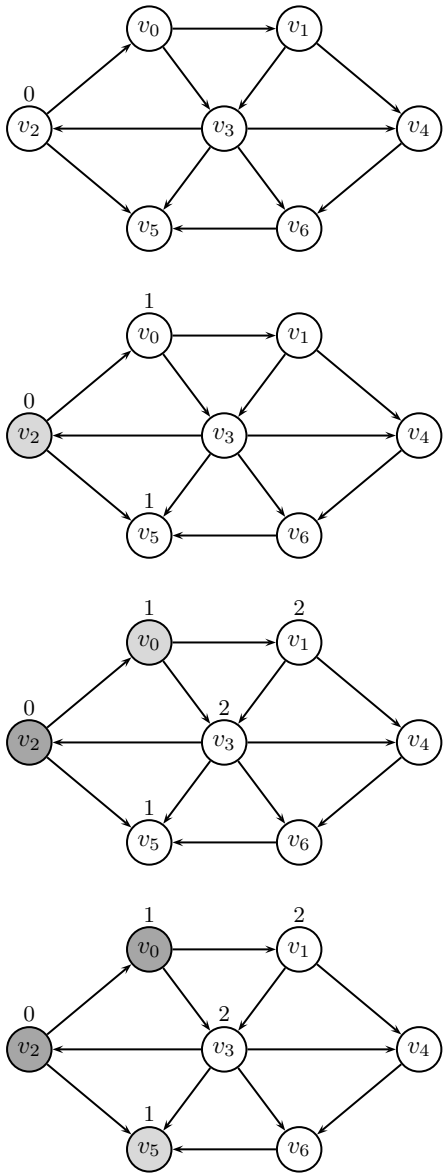


Pseudocode for Unweighted SSSP Algorithm

```
 Given a directed graph G and source node S.
Set up storage for a distance for each node, Dn:
For each node n, make Dn be INFINITY.
Set D_S = 0
Create queue q.
Enqueue source node S on q.
while the q is not empty
    Dequeue a node from q and call it v
    Loop though nodes w adjacent to v
        if D_w is INFINITY (not yet marked)
            set D_w to D_v + 1
            enqueue w on q
```

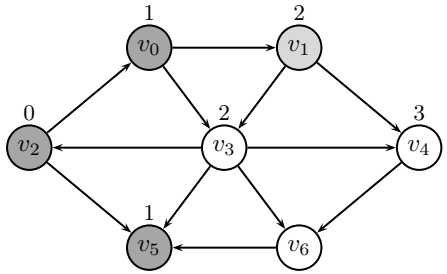
Running Example

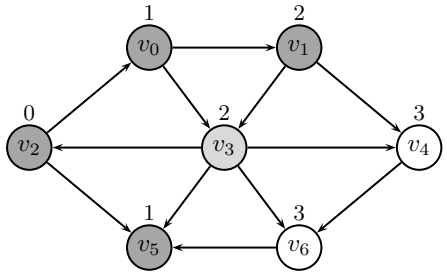
- In panel 1, the eyeball node v is not yet set, and $q = (V_2)$ from initialization.
- $D_0 = \infty$, $D_1 = \infty$, $D_2 = 0$, $D_3 = \infty$, $D_4 = \infty$, $D_5 = \infty$, $D_6 = \infty$,
- In panel 2, V_2 is dequeued and becomes the eyeball node v, and V_0 and V_5 are enqueued, so $q = (V_0, V_5)$
- When V_0 was enqueued, $D_w = D_0$ and $D_v = D_2 = 0$, so $D_0 = 0 + 1 = 1$

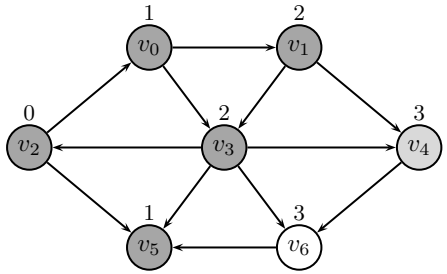


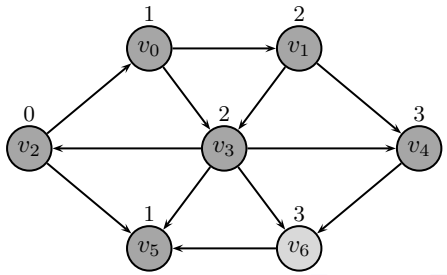
Running Example

- When V_5 was enqueued, $D_w = D_5$ and $D_v = D_2 = 0$, so $D_5 = 0 + 1 = 1$
- In panel 3, V_0 is dequeued and becomes the eyeball node, and V_1 and V_3 are enqueued, so q = (V_5, V_1, V_3)
- When V_1 was enqueued, $D_w = D_1$ and $D_v = D_0 = 1$, so $D_0 = 1 + 1 = 2$
- When V_3 was enqueued, $D_w = D_3$ and $D_v = D_0 = 1$, so $D_3 = 1 + 1 = 2$
- In 4, V_5 is dequeued and becomes the eyeball node, and nothing is enqueued, so q = (V_1, V_3)
- and so on.









Running Example

- V_0 and V_5 end up with distances 1 from V_2, as previously claimed from looking at the BFS.
- Similarly, V_1 and V_3 end up with distances 2 from V_2.
- This algorithm finds the distances fine, but what about the paths that provide these shortest distances, source to destination?
- Crucial idea: Every time we mark a node with a (final) distance, we are also performing the last step of the shortest path.
- So all we need to do is save the previous node along with the distance.
- Then we can get the penultimate step from the previous node the same way, and so on.

Pseudo-Code

```
 Given a directed graph G and source node S.
Set up storage for a distance for each node, Dn:
For each node n, make Dn be INFINITY.
Set D_S = 0
Create queue q.
Enqueue source node S on q.
while the q is not empty
    Dequeue a node from q and call it v
    Loop though nodes w adjacent to v
        if D_w is INFINITY (not yet marked)
            set D_w to D_v + 1
            set prev-of-w to v <--added for path
            enqueue w on q
```

Example

- when V_0 is enqueued above, now we also save the fact that prev-of-V_0 is the current v, V_2.
- when V_5 is enqueued above, now we also save the fact that prev-of-V_5 is the current v, V_2.
- when V_1 is enqueued above, now we also save the fact that prev-of-V_1 is the current v, V_0.
- when V_3 is enqueued above, now we also save the fact that prev-of-V_3 is the current v, V_0.
- So with these 4 facts, we can construct the best path from V_2 to V_3:
  - prev-of-V_3 is V_0
  - prev-of-V_0 is V_2
- Thus the best path from V_2 to V_3 is $V_2 \rightarrow V_0 \rightarrow V_3$.

Implementation

- Our challenge is to write this with our Graph API.
- The result of the algorithm is a distance and previous node for each node in the graph.
- Unlike Weiss's setup, we can't store the distances, etc. in the Graph itself.
- Again we set up a Map from V to what we need, in this case the combo of distance and prev-of Node.

Distinfo Class

```
static public class DistInfo {
    public final static int INFINITY =
        Integer.MAX_VALUE;
    private int dist = INFINITY;
    private Object prev = null; // a vertex object
    public int getDist() { return dist; }
    public Object getPrev() { return prev; }
    public String toString() {
        return "" + (dist == INFINITY ? ``INF'' :
            "" + dist)
        + `` (prev = ''
        + (prev == null ?  ``null'' :  prev) + ``)'';
    }
}
```

Distinfo Class

- This is a simple class, grouping two numbers, and defining one constant.
- The constant is needed to express the idea that a node is unreachable, that is, it's a special value for "dist".
- This class doesn't even have a constructor defined, so it gets a default constructor.
- The initializers of the fields (INFINITY, null) are set by that default constructor.
- This will be in the generic class for Unweighted<V,E>, and we could have V instead of Object, DistInfo<V> instead of DistInfo, etc.

Distinfo Class

- We will be using this DistInfo class outside Unweighted as well as inside, defining DistInfo objects that get passed back from the unweighted method to its caller.
- Here they are the values in a Map, where the keys are the vertices. Since the caller needs to interpret them, the class needs to be public.
- DistInfo is a class that is a helper to Unweighted, so we make it a public static class to express its relationship.
- The client calls it Unweighted.DistInfo. DistInfo is "static" because its objects are independent objects, not tied to any outer class object.

Distinfo Class

- There is no outer class object in this case anyway.
- Instead of a generic class, Unweighted has a generic method in a class that's never instantiated.
- We see "getters" here, getDist() and getPrev(), but no "setters" such as setDist() – why?
- Answer: because Unweighted will be setting these fields, and as an outer class, can access private fields of its nested classes.
- This way, the outside classes can't set anything, but the right class can, so it's nicely encapsulated.

Distinfo Class

- Look at cs310.Unweighted.java. Note that Unweighted is itself not a generic class, because there's no $< .. >$ in "public class Unweighted ..."
- Instead, the generic types are introduced at the method level.
- See pg. 152-153 for discussion of this use of generics.
- Note that Unweighted is a pure app of Graph, unlike Weiss's code, which needs to be a method of his Graph class to access vertexMap.
- Our Graph API is much more encapsulated.

Positive-Weighted, Shortest-path Problem

- Many times the edges have weights – indicating that the relations between nodes are not always equal.
- Example – travel distance between cities.
- In this case the algorithm from last class wouldn't work.
- Goal: Find the shortest path (measured by total cost) from a designated vertex S to every vertex in a weighted graph.
- The weights (costs) are assigned to edges.
- All edge costs are positive.

Dijkstra's Algorithm

- A greedy technique that actually works well
- Starting from a certain vertex v_0, we set up a Map to hold all the running min costs D_i from v_0 to various destinations v.
- Like the unweighted case, we have an "eyeball" that indicates what node we are visiting.
- We move the eyeball to a new node v, the one with the least cost from v0 that has not been visited by the eyeball.

Dijkstra's Algorithm

- We compute the min cost D_w from v_0 to various destinations w following any path through the visited node v
  - $D_w = \min\{D_w, D_v + c_{vw}\}$
  - c_{vw} = cost for edge $v \rightarrow w$ (these are all constant positive numbers)
- Repeat by moving the eyeball to another node which has the lowest cost and has not been visited.

Adjust D_w

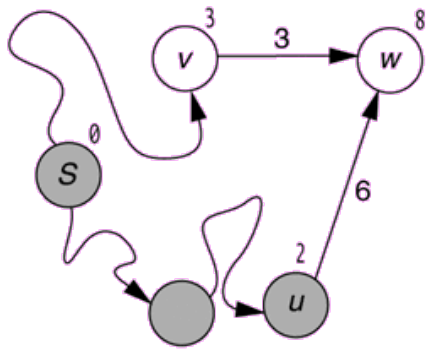
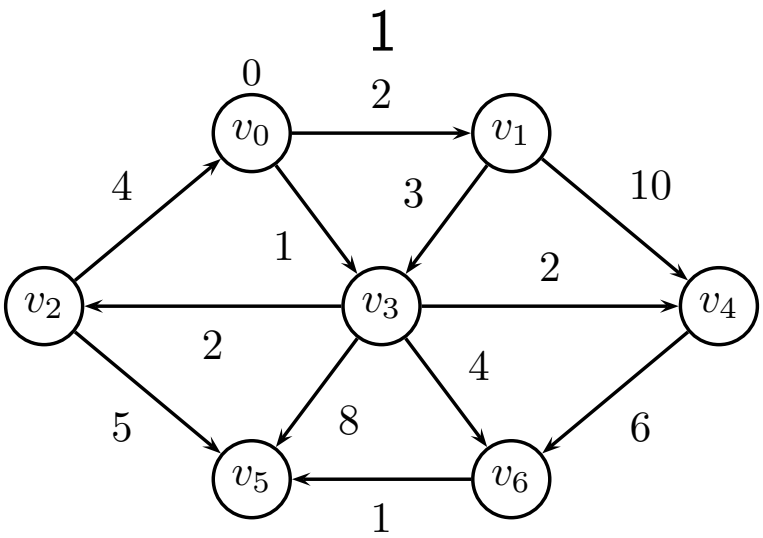


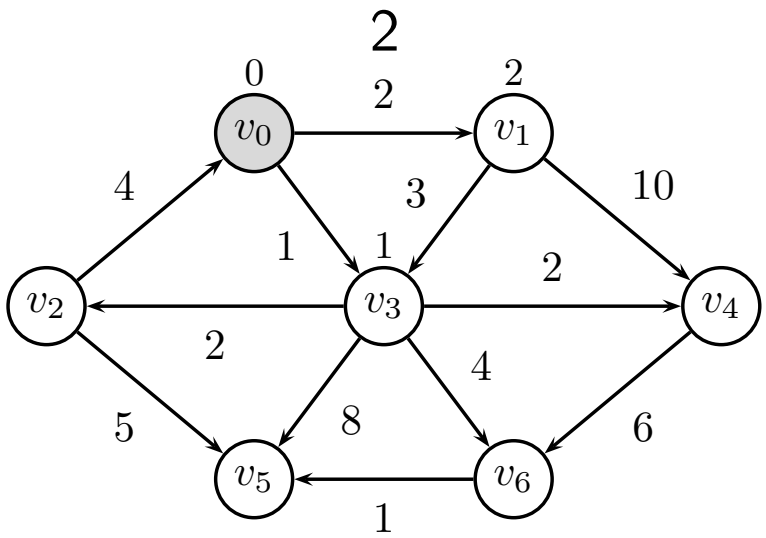
figure 14.23

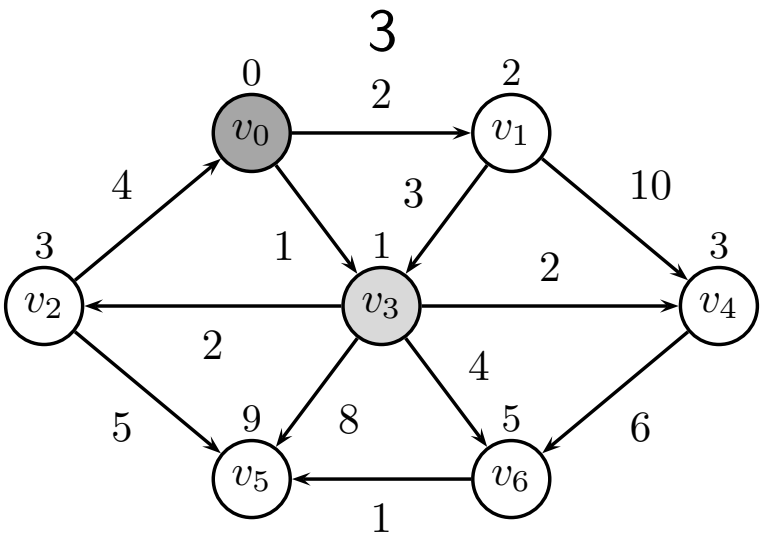
The eyeball is at v and w is adjacent, so D_w should be lowered to 6.

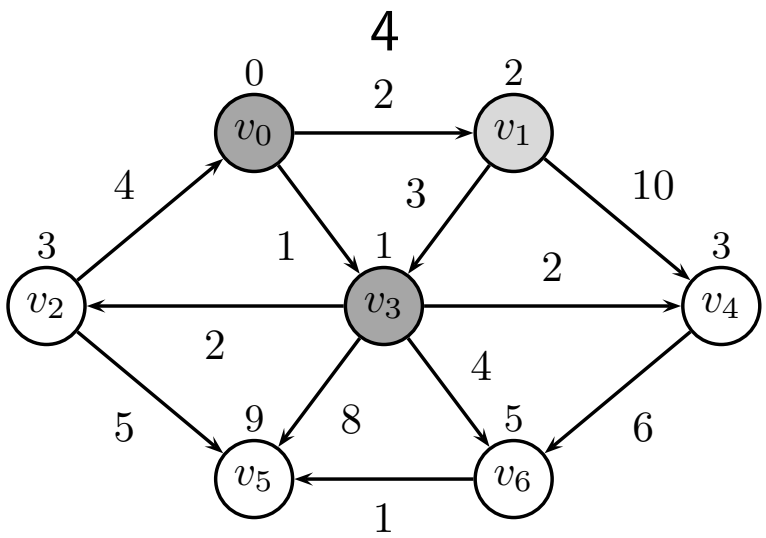
The eyeball is at v and w is adjacent to v, so D_w should be lowered from 8 to 6.

Running Example, Starting From V_0

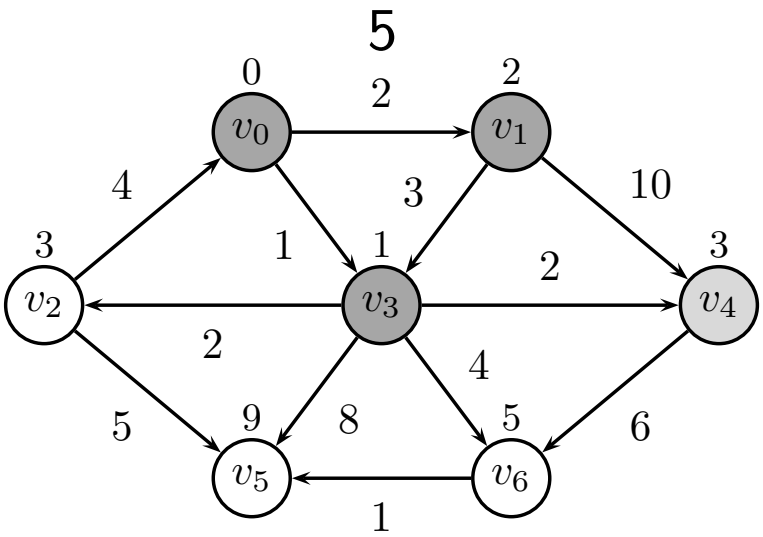


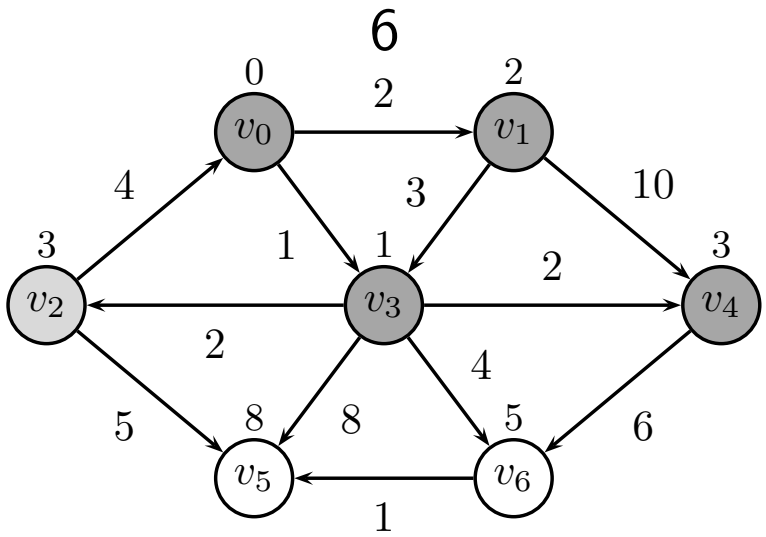


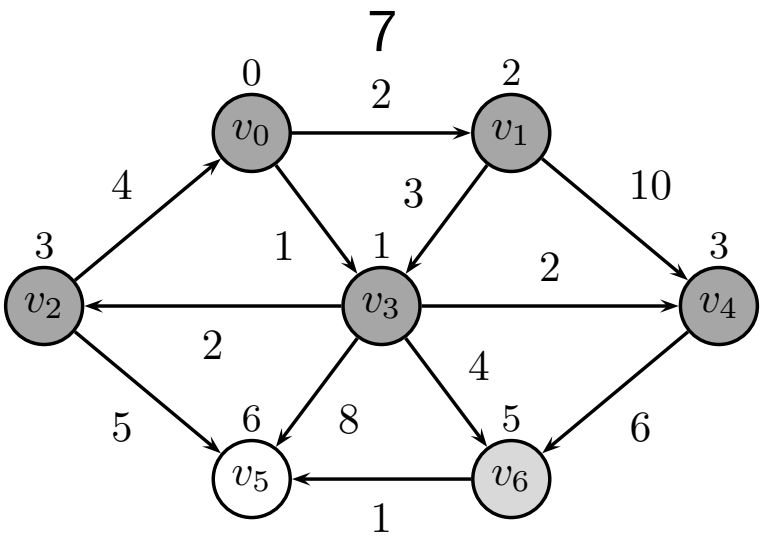


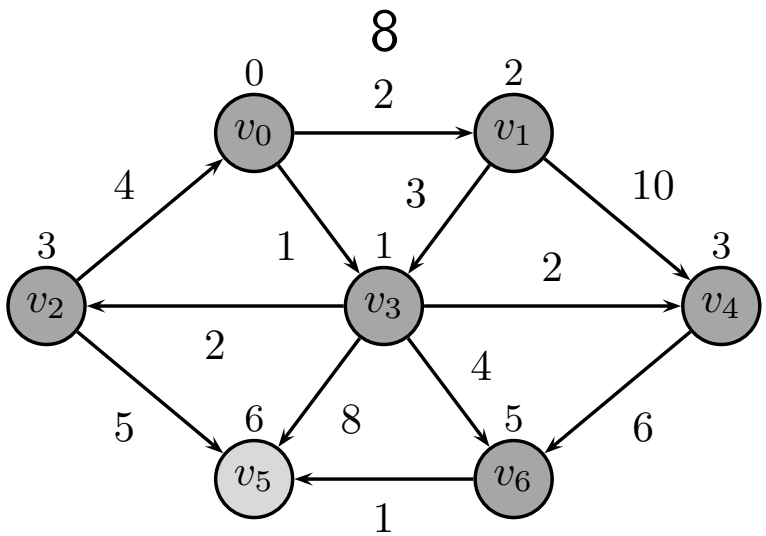


Running Example, Starting From V_0

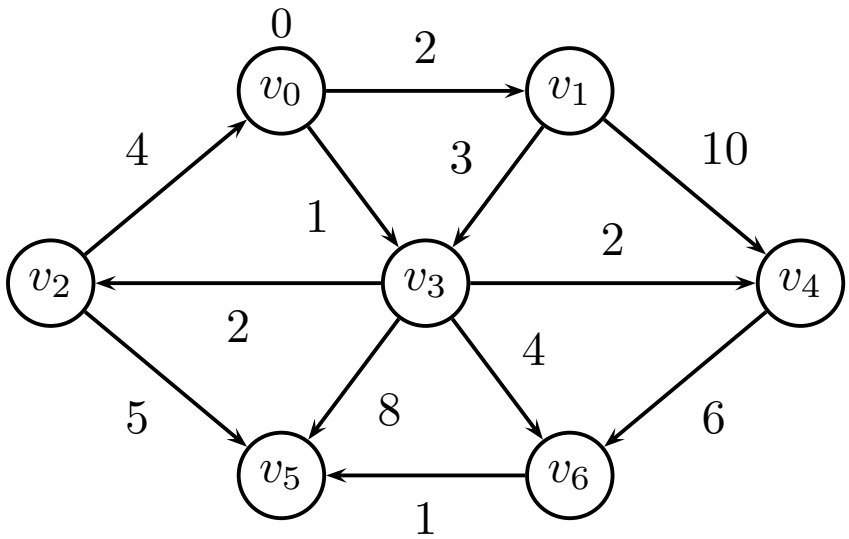








Path with Minimum Weights from V_0



eyeball	D_0	D_1	D_2	D_3	D_4	D_5	D_6
–	0	∞	∞	∞	∞	∞	∞
V_0	—	2	∞	1	∞	∞	∞
V_3	—	2	3	—	3	9	5
V_1	—	—	3	—	3	9	5
V_4	—	—	3	—	—	9	5
V_2	—	—	—	—	—	8	5
V_6	—	—	—	—	—	6	—
V_5	—	—	—	—	—	—	—
final:	0	2	3	1	3	6	5

Pseudocode for Dijkstra's Algorithm

Given a directed graph G and source node S.
Set up storage for a distance for each node, D_i:
    For each node i, set $D_i = c_{S,i}$
    (cost of the edge from S to i)
Set up a data structure P for unvisited nodes and put all nodes except S in it.
Loop while P is not empty
    Choose a node v in P with minimum D_i
    Delete v from P
    (it's now visited, and now the ``eyeball'' node)
    Loop though nodes w adjacent to v and in P
        $D_w = min(D_w, D_v + c_{vw})$

Proof of Dijkstra's Algorithm

- An eyeball-visit is a "stage", i.e. the body of one pass of the outer loop.
- Nodes are either "visited" or "unvisited" at the end of a certain stage.
- A node is "visited" after it has been an "eyeball node".
- In Figure 14.25, there is one panel for each stage. In a panel, the shaded nodes, light-shaded or dark-shaded, are the visited nodes.
- The light-shaded one has just been visited in this stage.
- Other (unvisited) nodes have been "seen" and marked with tentative D values.

Proof of Dijkstra's Algorithm

Pg. 548:

- The algorithm can be thought of as a sequence of stages. Each stage has two parts:
  1. finding the min D among unvisited nodes, to determine new eyeball node v, now visited.
  2. Updating D's for unvisited nodes w adjacent to v: old D_w vs. D_v + v-w-cost.
- The proof that this works is clearly a candidate for an inductive proof, but what is the inductive hypothesis?

Proof of Dijkstra's Algorithm

- Inductive hypothesis: After any stage, the values of D for vertices already visited are the final shortest distances and the values of D for the unvisited vertices represent the shortest distance using paths with visited nodes as intermediate path nodes.
- Base case: trivial for first stage.
- Induction step: Assume true for stage k, prove for k+1, with eyeball node v.

Proof of Dijkstra's Algorithm

- v is newly an eyeball, so its D is only known to be (by stage k setup) the shortest path with other eyeball-visited nodes as intermediate nodes.
- However, there might be another path to v that wanders outside the visited set and still gets to v with less total cost.
- Suppose, by contradiction, that there is a path from S to v with length less than D_v
- Let u be the first intermediate outside the visited set.
- By induction, the path to U is optimal for visited vertices.
- Then $D_v > D_u$ along that path, since costs only rise as edges are added (positive weight requirement).

Proof of Dijkstra's Algorithm

- But D_v was chosen by the algorithm to be the smallest D for unvisited nodes, a contradiction.
- Thus D_v is the best value.
- We have to show it is true for unvisited vertices.
- Consider an unvisited vertex i adjacent to v.
- We showed that the old D_i (at stage k) was the shortest path using the old visited nodes as intermediate nodes.
- Now we need to show the new $D_i = \min(oldD_i, D_v + C_{v,i})$ is the shortest using the new set of visited nodes, that is, with just the one additional node v.

Proof of Dijkstra's Algorithm

- Clearly the formula takes into account all the new possibilities if we are sure that v is the last intermediate node of a possible new path.
- But what if the best new path has v, but not as the last intermediate node? We need to show this is impossible.
- Suppose it could happen, with last intermediate node u, a visited node other than v. Then $S \rightarrow v \rightarrow u \rightarrow i$ is best to get to i.
- That means it's better than $S \rightarrow u \rightarrow i$ or any other path.
- Since the last edge is the same, we can say that $S \rightarrow v \rightarrow$ u is better than $S \rightarrow u$, but that means that u did not have a final best cost at step k, a contradiction.

Implementation of Dijkstra

- A simple implementation would mimic our manual processing of the D's earlier:
- scan the D's to choose a new eyeball, $O(|V|)$ for each node, or $O(|V|^2)$ in all, and then scan the edges from the eyeball node for D-updates, for each eyeball node, $|E|$ in all, so total $O(|V|^2)$, since $|E| < |V|^2$.
- Not too bad, but not great.
- But we need the minimal D each time. How can we do better?

Implementation of Dijkstra

- Problem - the D's change during the running of the algorithm, so a simple priority queue would seem to be useless.
- Weiss has a trick to allow us to use a simple priority queue even so.
- Finding the path can be done with the same trick as the simpler case.
- Every time you find a new better final edge for a path, you save the node it came from as "prev".

PQ for Elements That Change Priorities

- A trick that works in general for objects whose priority changes occasionally.
- Given an example of support messages with priorities. Now we add a valid field in the objects.
- PQ = {(Joe, pri 3, valid), (Sue, pri 4, valid), (Tom, pri 5, valid)}
- A manager has a ref to Sues message, wants to change it to pri 1, to be processed next. Just make a copy of it, change priority in the copy and set valid to invalid in the old one:
- PQ = {(Sue, pri 1, valid), (Joe, pri 3, valid), (Sue, pri 4, invalid), (Tom, pri 5, valid)}

PQ for Elements That Change Priorities

- This is perfectly proper use of the ordinary priority queue.
- We can't change the priority of an element in a simple PQ, but nothing is stopping us from changing other fields.
- When we deleteMin a message that's marked invalid, we just throw it away and deleteMin again.

PQ for Elements That Change Priorities

- Of course we are carrying around extra baggage in the PQ, so this is only a good solution for occasional changes in working priority.
- It works particularly well for changes that lower the priority number, because the new valid entry then always precedes the newly-invalid entry, so the invalid ones aren't a surprise when they come off the PQ.
- This is the case in Dijkstra's algorithm.