



CSE 100:

WEIGHTED GRAPH
SHORTEST PATH

What to expect from the tutors

- The tutor's goal is to get you “un-stuck”, not to solve all your problems.
- If you need major help at the last minute, it's unlikely that the tutor will be able to provide the help you require.

SEEK HELP EARLY

- If your class design is complicated to the point where it is making it very difficult for the tutor to help you solve your specific bug, the tutor may suggest you refactor your code before they can help you. This is a perfectly reasonable suggestion. It will help you in the long run. Do not get angry with the tutor.

Respect your tutors

- Your tutors are working hard to help you. You and they are adults, and I expect you all to act professionally. This means:

- Do not yell at your tutor or insult them, either to their face, behind their back, or in e-mail
- Sexist, racist or otherwise offensive comments toward the tutors (or any student in the class) are NEVER OK.
- Do not beg your tutor to stay when they say they have to move on
- Be understanding when they can't answer *all* your questions. These are hard assignments, and there's an unbounded number of ways things can go wrong.
- Come to me with any problem you have with the tutoring or other course staff.

What is this algorithm??

Mystery(G,v) (v is the vertex where the search starts)

Stack S := {}; (start with an empty stack)

for each vertex u, set visited[u] := false;

push S, v;

while (S is not empty) do

u := pop S;

if (u is not visited) then

visited[u] := true;

for each unvisited neighbour w of u

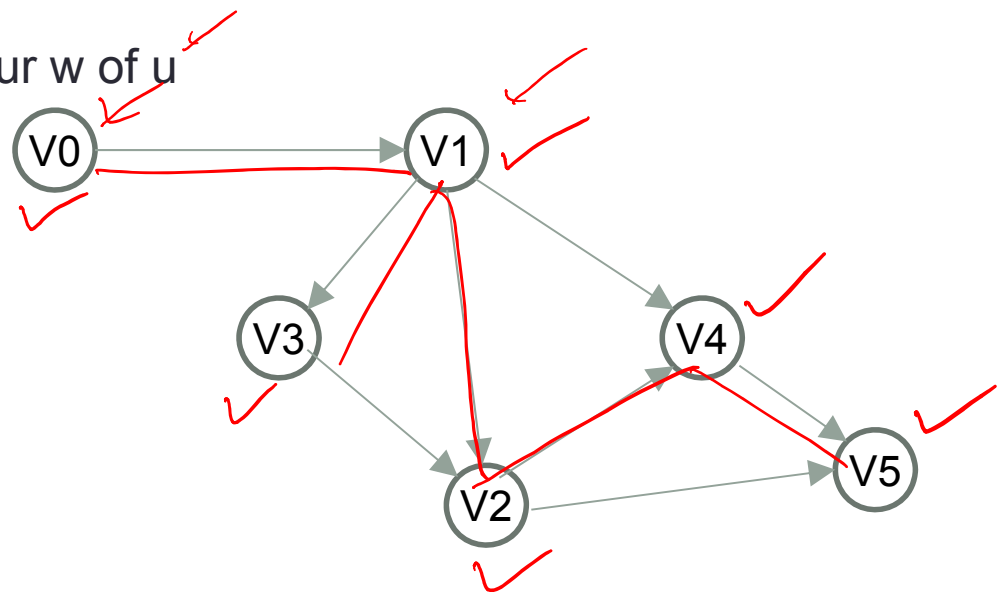
push S, w;

end if

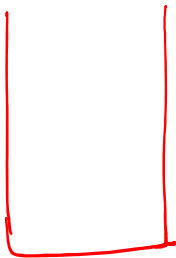
end while

END Mystery()

$V_0, V_1, V_2, V_3, V_4, V_5$



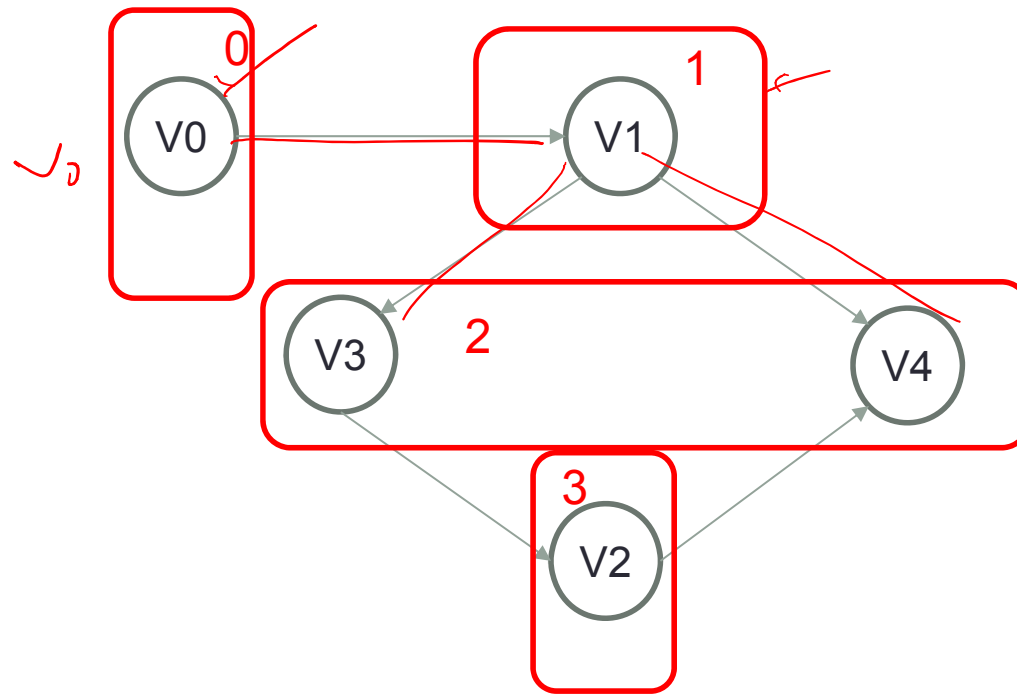
Stack:



A. BFS B. DFS C. Dijkstra's algorithm D. Nothing interesting

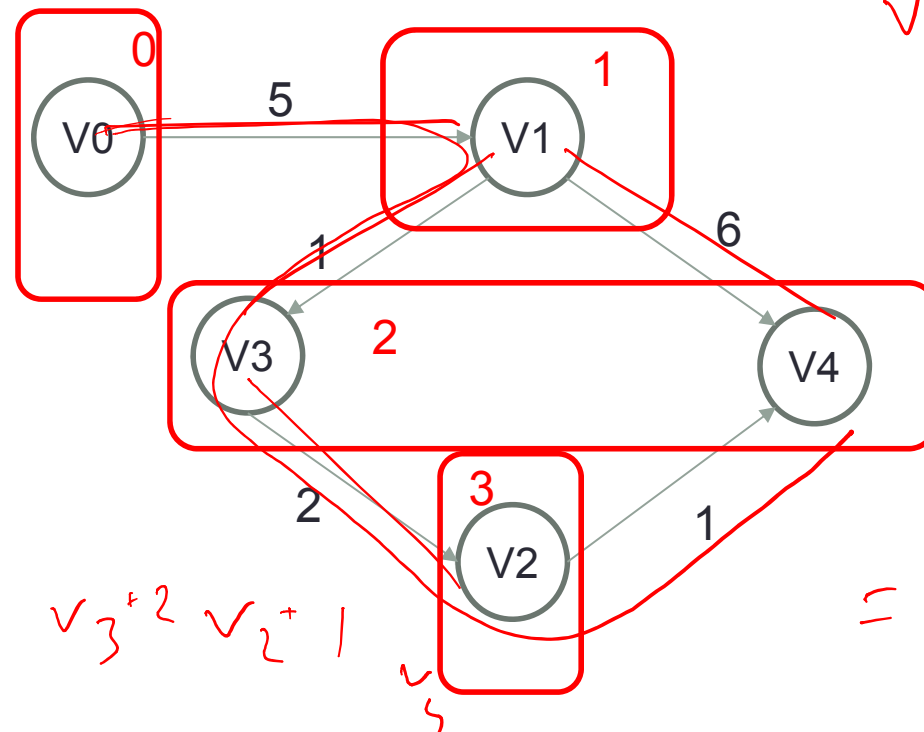
Why Did **BFS** Work for Shortest Path?

- Vertices are explored in order of their distance away from the source. So we are guaranteed that the first time we see a vertex we have found the shortest path to it.
- The queue (FIFO) assures that we will explore from all nodes in one level before moving on to the next.



BFS on weighted graphs?

- Run BFS on this weighted graph. What are the weights of the paths that you find to each vertex from v_0 ? Are these the shortest paths?

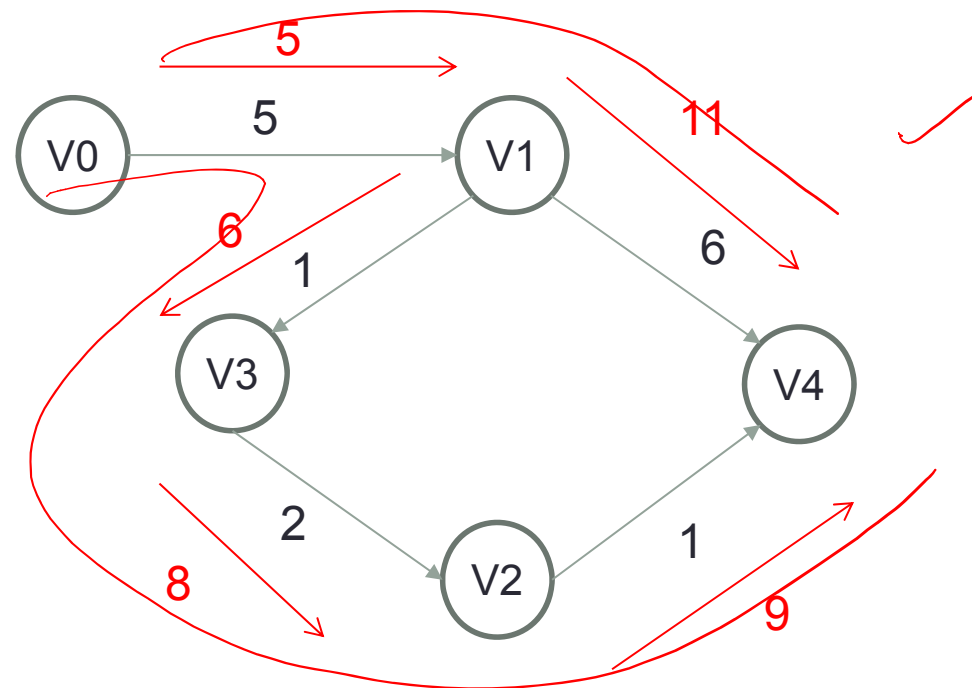


$$v_0 \xrightarrow{5} v_1 \xrightarrow{6} v_4 = 11$$

$$v_0 \xrightarrow{5} v_1 \xrightarrow{1} v_3 \xrightarrow{2} v_2 \xrightarrow{1} v_4 = 9$$

BFS on weighted graphs?

- In a weighted graph, the number of edges no longer corresponds to the length of the path. We need to decouple path length from edges, and explore paths in increasing *path length* (rather than increasing number of edges).
- In addition, the first time we encounter a vertex may, we may not have found the shortest path to it, so we need to delay committing to that path.



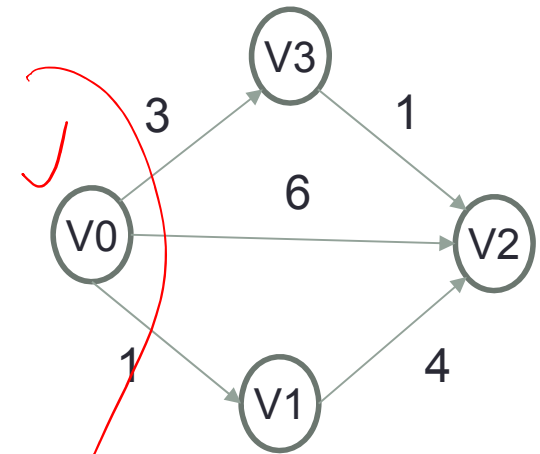
Dijkstra's Algorithm

Key ideas:

- Vertices are explored similar to previous graph search algos: start with a source and iteratively grow like a mold
- Vertices maintain a “tentative” measure of their distance from the source called:

The Dijkstra Score

Nodes have:
prev
dist
done

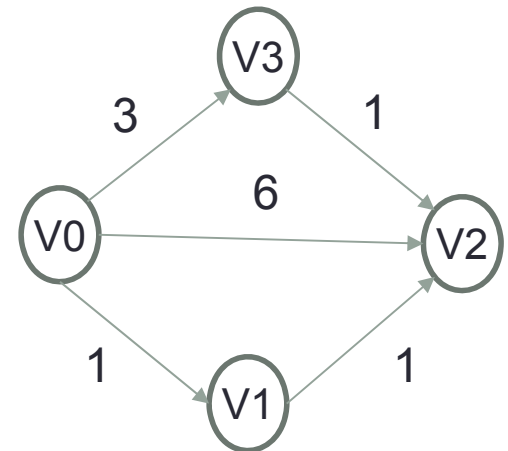


Dijkstra's Algorithm

Key ideas:

- Vertices are explored similar to previous graph search algos: start with a source and iteratively grow like a mold
- Vertices maintain a “tentative” measure of their distance from the source called:
The Dijkstra Score
- The scores of all vertices that have been explored so far is maintained in **some data structure**
- Among a choice of candidate vertices, the vertex with the minimum score is picked.
- If a new shorter route is discovered to a vertex, its score is updated and **inserted into the data structure storing all the scores**
- How do we make progress? Think about the very first iteration

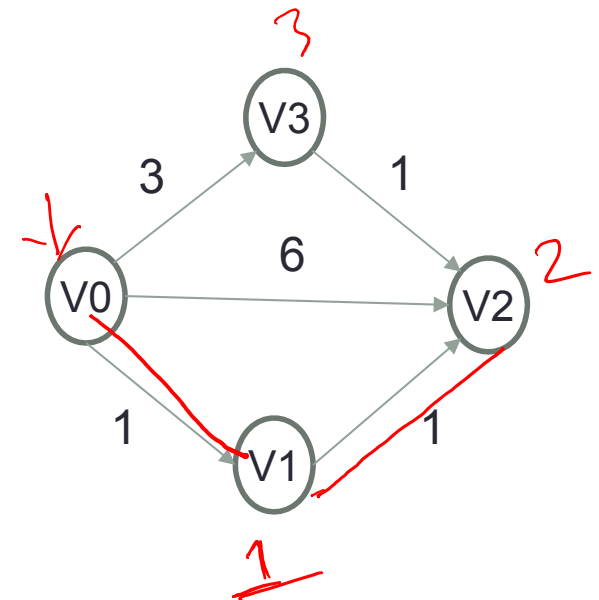
Nodes have:
prev
dist
done



Towards a fast implementation of Dijkstra's Algorithm

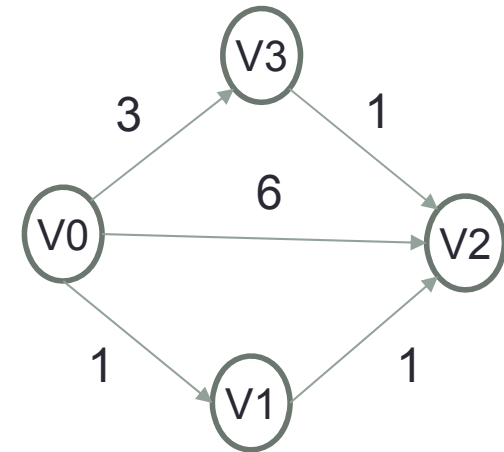
- When the shortest path to a new vertex is discovered, we need to update the Dijkstra scores of all vertices that are connected to the new vertex
- What should happen in the next iteration, once the shortest path to v1 is discovered?

- A. The score for v2 should be updated
- B. The score for v3 should be updated
- C. The score for both v2 and v3 should be updated
- D. None of the above



Towards a fast implementation of Dijkstra's Algorithm

- Which data structure should be used to maintain the Dijkstra scores?
 - A. Linked list $O(n)$
 - B. Sorted array $O(1)$ $O(n)$
 - C. Heap** $O(1)$ $O(\log n)$ ←
 - D. BST $O(n)$ $O(n)$
 - E. None of the above



Dijkstra's Algorithm

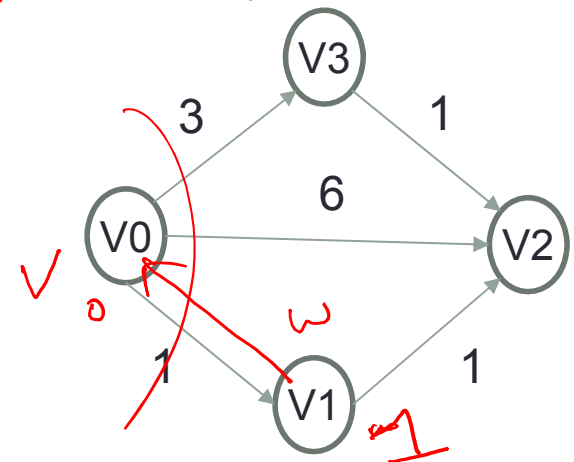
Nodes have:

- ✓ prev
- ✓ dist
- ✓ **done**

- Initialize the graph: Give all vertices a dist of INFINITY, set all “done” flags to false
- Start at s; give s dist = 0 and set prev field to -1
- Enqueue (s, 0) into a priority queue. This queue contains pairs (v, cost) where cost is the best cost path found so far from s to v. It will be ordered by cost, with smallest cost at the head.
- While the priority queue is not empty or until all shortest paths are discovered
 - Dequeue the pair (v, c) from the head of the queue.
 - If v’s “done” is true, continue
 - Else set v’s “done” to true. We have found the shortest path to v. (It’s prev and dist field are already correct).
 - For each of v’s adjacent nodes, w (*whose done flag is not true*):
 - Calculate the best path cost (also referred to as score), c, to w via v by adding the edge cost for (v, w) to v’s “dist”.
 - If c is less than w’s “dist”, replace w’s “dist” with c, replace its prev by v and enqueue (w, c)

$$c = 0 + 1 = 1$$

$$(w, c)$$



Dijkstra's Algorithm: Data Structures

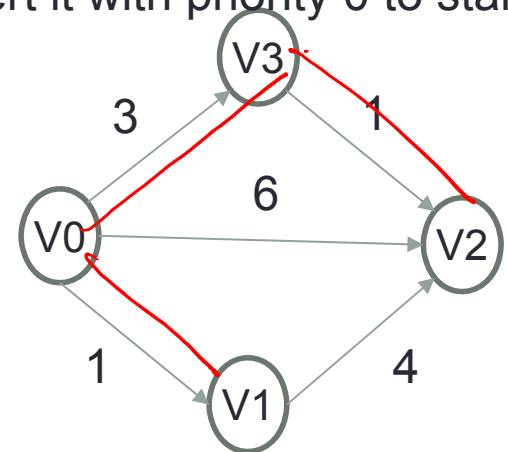
- Maintain a sequence (e.g. an array) of vertex objects, indexed by vertex number
 - Vertex objects contain these 3 fields (and others):
 - ✓ “dist”: the cost of the best (least-cost) path discovered so far from the start vertex to this vertex.
 - ✓ “prev”: the vertex number (index) of the previous node on that best path ↩
 - ✓ “done”: a boolean indicating whether the “dist” and “prev” fields contain the final best values for this vertex, or not
- Maintain a priority queue
 - The priority queue will contain (*pointer-to-vertex*, *path cost*) pairs
 - *Path cost* is priority, in the sense that low cost means high priority
 - Note: multiple pairs with the same “*pointer-to-vertex*” part can exist in the priority queue at the same time. These will usually differ in the “*path cost*” part

Your Turn

The array of vertices, which include dist, prev, and done fields (initialize dist to 'INFINITY' and done to 'false'):

V0:	dist= 0	prev= -1	done= true	adj: (V1,1), (V2,6), (V3,3)
V1:	dist= 1	prev= 0	done= true	adj: (V2,4)
V2:	dist= 4	prev= 3	done= true	adj: -
V3:	dist= 3	prev= 0	done= true	adj: <u>(V2,1)</u>

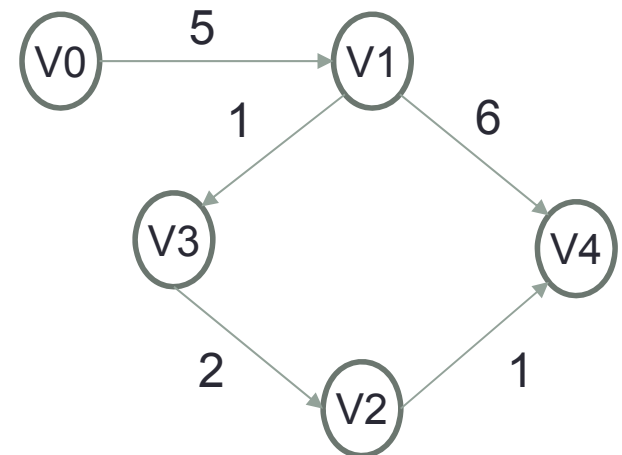
The priority queue (set start vertex dist=0, prev=-1, and insert it with priority 0 to start)



Dijkstra's Algorithm: Questions

- Initialize the graph: Give all vertices a dist of INFINITY, set all “done” flags to false
- Start at s ; give s dist = 0 and set prev field to -1
- Enqueue $(s, 0)$ into a **priority queue**. This queue contains pairs (v, cost) where cost is the best cost path found so far from s to v . It will be ordered by cost, with smallest cost at the head.
- While the priority queue is not empty:
 - Dequeue the pair (v, c) from the head of the queue.
 - If v 's “done” is true, continue
 - Else set v 's “done” to true. We have found the shortest path to v . (It's prev and dist field are already correct).
 - For each of v 's adjacent nodes, w :
 - Calculate the best path cost, c , to w via v by adding the edge cost for (v, w) to v 's “dist”.
 - If c is less than w 's “dist”, replace w 's “dist” c and enqueue (w, c)

When a node comes out of the priority queue, how do you know you've found the shortest path to the node?



Representing the graph with structs

```
#include <iostream>
#include <limits>
#include <vector>
#include <queue>
```

```
using namespace std;
```

```
struct Vertex
```

```
{
    vector<int> adj; // The adjacency list
    int dist;      // The distance from the source
    int index;     // The index of this vertex
    int prev;      // The index of the vertex previous in the path
};
```

```
vector<Vertex*> createGraph()
{ ... }
```

BFSTraverse: C++ code

```
/** Traverse the graph using BFS */  
void BFSTraverse( vector<Vertex*> theGraph, int from )  
{  
    queue<Vertex*> toExplore;  
    Vertex* start = theGraph[from];  
    start->dist = 0;  
    toExplore.push(start);  
    // finish the code...  
}
```

```
struct Vertex  
{  
    vector<int> adj;  
    int dist;  
    int index;  
    int prev  
};
```

- Initially, give all vertices in the graph a distance of INFINITY
- Start at s; give s distance = 0
- Enqueue s into a queue
- **While the queue is not empty:**
 - **Dequeue the vertex v from the head of the queue**
 - **For each of v's adjacent nodes that has not yet been visited:**
 - **Mark its distance as 1 + the distance to v**
 - **Enqueue it in the queue**

/ Traverse the graph using BFS */**

void BFSTraverse(vector<Vertex*> theGraph, int from)

```
{
  queue<Vertex*> toExplore;
  Vertex* start = theGraph[from];
  start->dist = 0;
  toExplore.push(start);
  while ( !toExplore.empty() ) {
    Vertex* next = toExplore.front();
    toExplore.pop();
    vector<int>::iterator it = next->adj.begin();
    for ( ; it != next->adj.end(); ++it ) {
      Vertex* neighbor = theGraph[*it];
      if ( neighbor->dist == numeric_limits<int>::max() ) {
        neighbor->dist = next->dist + 1;
        neighbor->prev = next->index;
        toExplore.push(neighbor);
      }
    }
  }
}
```

```
struct Vertex
{
  vector<int> adj;
  int dist;
  int index;
  int prev
};
```

While the queue is not empty:

- **Dequeue the vertex v from the head of the queue**
- **For each of v 's adjacent nodes that has not yet been visited:**
 - **Mark its distance as $1 +$ the distance to v**
 - **Enqueue it in the queue**