7SENG010W Data Structres & Algorithms

Week 10 Lecture

Graph Shortest Path Algorithms

Overview of Week 10 Lecture: Graph Shortest Path Algorithms

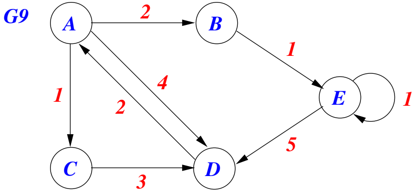
- *Concepts of Graphs*
- *Shortest Path Algorithms*
- *Dijkstra's Shortest Path Algorithm*

PART I

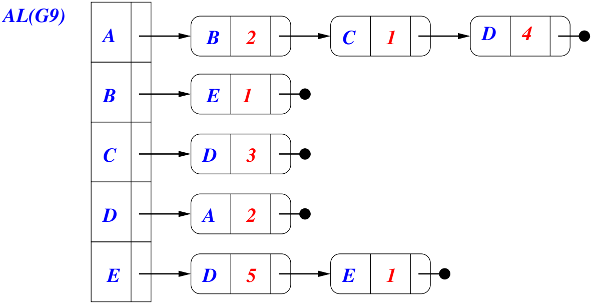
Concepts of Graphs

Adjacency Lists, Paths & Breadth First Search

Adjacency List for Weighted Directed Graph *G9*



Adjacency list representation of *G9*, using an *array* for the vertices headers.



Paths in a Graph

Definition: Paths

A **path** in a graph is a sequence of vertices connected by edges.

A **simple path** is a path with no repeated vertices.

Examples: In *G9*: $A \to C \to D \to A \to D$ is a path.

And $A \to B \to E \to D$ is a *simple path*.

Graph Traversal

- A graph **traversal** (or **search**) algorithm is:
  1. starting from some chosen vertex,
  2. a "**walk**" around a graph in a systematic manner,
  3. in such a way that every vertex reachable from that starting vertex is visited **exactly once**.

- There are two traversal algorithms: *Depth-First Search* (DFS) & *Breadth-First Search* (BFS).

- The *breadth-first* traversal, or search, algorithm (BFS):
  - visits all vertices adjacent to the starting vertex,
  - then visits all vertices adjacent to those vertices, and so on.

- Since all adjacent vertices are visited before probing further, the search is broad rather than deep, hence the term "*breadth-first*".

Pseudo Code for Breadth First Search (BFS) Graph Traversal

Pseudo code uses two methods: `BreadthFirstSearch` & `bfs(v)`.

- BFS algorithm pseudo code performs *breadth-first* traversal (or search), of a graph stored in an adjacency matrix `AM`, starting from vertex 1.

  (Alternatively use an adjacency list.)

- It uses a FIFO queue `vertexQueue` to store `sv`'s *visited vertex neighbours*, i.e. adjacent vertices of vertex `u`.

- `vertexQueue` is also known as the "*open list*", as it plays the rôle of `bfs`'s "*to-do-list*".

- A vertex is in this list if it has been *found*, but not yet "*explored*", i.e. its neighbour vertices have not yet been found.

- `ProcessVertex( u )` can add vertex `u` to the list of *fully explored* vertices, this list is known as the "*closed list*", since it has been *fully explored* & plays no further role in the traversal.

- Vertices are added to this *closed list* in the order in which they have been *fully explored*, i.e. *Breadth-First* traversal order.

Pseudo Code of Breadth First Search Algorithm (1/2)

```
BreadthFirstSearch:

    /* Initialisation: NO vertex has been visited */
    FOR v <-- 1 TO numberOfVertices
    DO
           visitedVertex[ v ] <-- NO
    ENDFOR

    // Perform BFS from each vertex in the graph
    FOR v <-- 1 TO numberOfVertices
    DO
        IF    visitedVertex[ v ] == NO
        THEN
              // performs BFS starting from vertex v
              bfs( v )
        ENDIF

    ENDFOR
```

Pseudo Code of Breadth First Search Algorithm (2/2)

```
bfs( vertex sv ):

   vertexQueue <-- emptyQueue    // create an empty FIFO queue

   visitedVertex[sv] <-- YES     // YES vertex has been visited

   vertexQueue.insert( sv )      // add sv to rear of queue

   WHILE ( vertexQueue is not empty )
   DO
      u <-- vertexQueue.remove()  /* remove 1st visited neighbour
                                     vertex from queue */

      FOR  dv <-- 1 TO numberOfVertices
      DO
         IF ( (AM[u][dv] == TRUE) AND (visitedVertex[dv] == NO) )
         THEN
               visitedVertex[dv] <-- YES

               vertexQueue.insert( dv )  // add dv to rear of queue
         ENDIF
      ENDFOR

      ProcessVertex( u )          // Process vertex u as required

   ENDWHILE
```

PART II

Shortest Path Algorithms

What is the "Shortest Path" Problem

Problem:

Is to find the shortest path in a weighted edge digraph, from a source (start) vertex s to a sink (destination) vertex t.

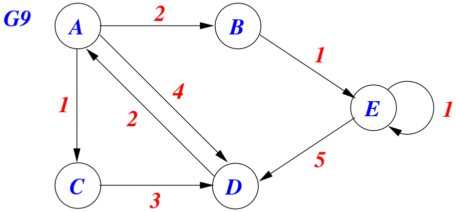
Where the *shortest path between s & t* is defined as a path that:

- starts at *s* & ends at *t*,
- follows the *direction of the edges*,
- is a *simple* path, i.e. no repeated vertices,
- the *sum of the path's edge weights* is less than or equal to that of all other paths that exist between *s* & *t*.

NOTE: that the *number of edges traversed is irrelevant*, it is only the sum of the edge weights that matter.

Example of a Shortest Path

For example in *G9* with *s = A* & *t = D*:



Here both of:

$A \xrightarrow{4} D$ (path weight = *4*)

&

$A \xrightarrow{1} C \xrightarrow{3} D$ (path weight = *4*)

are *shortest paths*, but

$A \xrightarrow{2} B \xrightarrow{1} E \xrightarrow{5} D$ (path weight = *8*)

is a path but **not** a *shortest path*.

Shortest Path Algorithm Variants

- There are several variants of the basic Shortest Path problem algorithm.
- *Shortest Path algorithm variants:*
  - *Single source:* from one vertex *s* to every other vertex in a graph.
  - *Single sink:* from *every* vertex in the graph to one vertex *t*.
  - *Source-sink:* from one vertex *s* to another vertex *t*.
  - *All pairs:* between all pairs of vertices in a graph.
- Restrict edge weights to be *non-negative*, i.e. only allow $0 \leq$ weights. (There are algorithms that can deal with *negative* edge weights.)
- We will only consider the *Single source* variant with *non-negative* edge weights in this lecture — *Dijkstra's Shortest Path Algorithm*.

Data Structures for Single-Source Shortest Paths (1/2)

Goal: is to find the *shortest path* from *s* to every other vertex, therefore we need to be able to represent a *shortest path*.

We can represent a graph's *shortest paths* from *s* to every other vertex with two vertex indexed arrays:

- `distTo`[*v*] – is the *length* of the shortest path from *s* to *v* currently known.

- `edgeTo`[*v*] – is the *last edge* in shortest path from *s* to *v*, currently known.

Data Structures for Single-Source Shortest Paths (2/3)

Example: Assume that the *current shortest path* found so far from s to v_k is the following:

$$CSP(s, v_k) \;=\; s \xrightarrow{w_1} v_1 \xrightarrow{w_2} v_2 \xrightarrow{w_3} v_3 \quad \ldots \quad v_{k-1} \xrightarrow{w_k} v_k$$

$$= \langle\, (s,\; v_1,\; w_1),\; (v_1,\; v_2,\; w_2),\; (v_2,\; v_2,\; w_3), \quad \ldots \quad (v_{k-1}, v_k,\; w_k)\,\rangle$$

Then:

- `distTo[`v_k`]` is the *sum of the weights* of $CSP(s, v_k)$:

  $$\texttt{distTo}[v_k] \;=\; w_1 + w_2 + w_3 + \ldots + w_k$$

- `edgeTo[`v_k`]` is the *last edge*:

  $$v_{k-1} \xrightarrow{w_k} v_k$$

  that connects to vertex v_k in $CSP(s, v_k)$:

  $$\texttt{edgeTo}[v_k] \;=\; (\; v_{k-1},\; v_k,\; w_k \;)$$

Data Structures for Single-Source Shortest Paths (3/3)

It is also necessary to use a *queue* to record the vertices that have been *found* but not yet *fully explored*.

- This plays the same rôle as the `vertexQueue` in Breadth-First search.
- However, unlike `vertexQueue`, the queue required for finding the shortest paths is more complicated.
- For this queue the items in this queue are a *pair of values*:
  - a vertex *v* that has been *found*, but *not yet explored*, &
  - `distTo`[*v*] the *length of* its currently known *shortest path* from *s*.

  That are both inserted into the queue.
- The *length of its shortest path*, i.e. `distTo`[*v*], is then used to *sort the vertices into ascending order*.
- Result is the vertices in the queue[3] are in "*distance from the source vertex s*" order, & that the *first vertex* in the queue is the **nearest to** ***s***.

[3] This type of queue is an example of a data structure known as a *priority-queue*; here the value of `distTo`[*v*] is the "*priority*".

Edge “Relaxation” (1/2)

Process of *finding a shortest path* relies on traversing a graph starting from the source vertex *s* searching for shorter paths from it to all the other vertices, when one is found, it updates the shortest paths accordingly.

- From each vertex `sv`, search for all of its adjacent destination vertices `dv`.
- Then check if a shorter distance exists from the source vertex ***s*** to `dv` via `sv`, if it does then update the shortest path for `dv`.
- This checking operation is known as *“Edge Relaxation”*, & it is applied to each adjacent destination vertex `dv` in turn.
- So *“Edge Relaxation”* is applied to an *edge* `(sv, dv, w)` that connects:
  - the vertex `sv` that is being *searched from*
  - to an adjacent destination vertex `dv`,
  - via the edge of weight `w`.

Edge "Relaxation" (2/2)

The process of *Edge "Relaxation"* involves:

- Checking if this edge `(sv, dv, w)` results in a *shorter path* for the edge's destination vertex `dv`.

- This is done by checking if the following condition is true:

  ```
  distTo[ dv ] > distTo[ sv ] + w
  ```

  i.e. the current known shortest distance to `dv` is longer than the distance via `sv`.

- If it is then update `dv`'s shortest path as follows:

  ```
  distTo[ dv ]  <--  distTo[ sv ] + w

  edgeTo[ dv ]  <--  (sv, dv, w)
  ```

- Insert `dv` & its new *shorter path distance* into the priority queue:

  ```
  PriQueue.Enqueue( dv, distTo[dv] )
  ```

PART III

Dijkstra’s Shortest Path Algorithm

Dijkstra's Shortest Path Algorithm (1/2)

- Dijkstra's *Shortest Path* algorithm is based on *Breadth-First* traversal.
- As in BFS, it finds all graph vertices reachable from a particular *starting vertex s*, before proceeding to explore any of the ***found vertices*** further, i.e. *"broad rather than deep"*.
- As with BFS the ***found vertices*** are inserted into a queue, but the vertices are *sorted by ascending path length* (distance) from the starting vertex *s*, not just inserted at the end of the queue as in BFS.
- The important difference is that after performing this search, the *selection criteria* for choosing the next ***found vertex*** to explore is (semantically) different, due to the *sorted* queue.
- In the *Shortest Path* algorithm as in BFS it still selects the vertex at the front of the queue to explore next.
- **But** now the first vertex is not just some random ***found vertex*** as in BFS, but the one with the **shortest path** from the staring vertex *s*, i.e. the nearest to *s*.

Dijkstra's Shortest Path Algorithm (2/2)

- Using this "*nearest to s*" vertex ***sv*** it repeats the process of searching for undiscovered vertices that are adjacent to it.

- When it finds one, e.g. ***dv***, it performs an *edge relaxation* on the connecting edge (***sv***, ***dv***, w) by:
  - checking the length of vertex ***dv***'s path from the source vertex *s*, via the search vertex ***sv***;
  - if it is shorter, it updates the found vertex ***dv***'s path;
  - inserts the found vertex ***dv*** into the queue in the appropriate position based on its path length.

The following pseudo code for Dijkstra's Shortest Path algorithm is based on *breadth-first* traversal.

The *weighted digraph* it is applied to is represented as an adjacency list, but it could have used an adjacency matrix.

Dijkstra's Shortest Path Algorithm Pseudo Code Data Structures (1/3)

The algorithm uses several data structures to represent information about the graph & the state of the algorithm:

- a composite data type to represent a *weighted directed edge*;
- an *adjacency list* of *edge*s to represent a graph;
- a *vertex indexed array of edges*, i.e. `edgeTo[dv] = (sv, dv, w)`, All initialised to none.
- a *vertex indexed array of distances*, i.e. from the vertex to **s**, `distTo[dv]`;
  All initialised to *infinity* ∞, so a *real path* will always be shorter.
- a *priority queue* containing a list of *(key, value)* pairs sorted in *ascending key* order.
  where:
  - *key* – is the *distance of the vertex from the source vertex s*,
  - *value* – is the vertex.

  Initialised with the source vertex *s* & *zero distance*.

Dijkstra’s Shortest Path Algorithm Pseudo Code Data Structures (2/3)

```
Edge:
     Vertex source              // represents edge:
     Vertex destination         // ( sv, dv, weight(sv, dv) )
     int    weight
END

PriorityQueue:
   KEY   int
   VALUE Vertex

   OPERATIONS:

      Enqueue( Vertex, int )  // insert vertex & distance
                              // into queue

      Dequeue()               // remove & return first item in
                              // queue, i.e. vertex with the
                              // minimum distance from source vertex
END
```

Dijkstra's Shortest Path Algorithm Pseudo Code (1/3)

```
DijkstraShortestPath( Vertex startVertex )
BEGIN

   // Initialise all the data structures

   Edge[] edgeTo        // edgeTo[dv] = (sv, dv, w)

   int[]  distTo        // distance from startVertex to vertex

   // initialise all vertex distances to infinity
   FOR vertex <-- 0 TO numberOfVertices - 1
   DO
        distTo[vertex] <-- INFINITY
   ENDFOR

   // initialise startVertex distances to 0
   distTo[ startVertex ] <-- 0


   PriorityQueue PriQueue  <-- EMPTY_QUEUE  // empty priority queue

   PriQueue.Enqueue( startVertex, 0 )       // add  start Vertex,
                                            // with distance 0
```

Dijkstra's Shortest Path Algorithm Pseudo Code (2/3)

`DijkstraShortestPath( Vertex startVertex )` continued:

```
    // Run algorithm, until all vertices have been processed

    WHILE ( PriQueue is not empty )
    DO
        // "relax" vertices in order of distance from startVertex

        // get nearest vertex to startVertex
        nearestVertex  <-- PriQueue.Dequeue()

        // for all of nearest vertex's adjacent vertices
        // check if a shorter path exists via the nearest vertex

        FORALL edge IN AdjacencyList[ nearestVertex ]
        DO
                   RelaxEdge( edge )
        ENDFORALL

       ENDWHILE

END // DijkstraShortestPath
```

Dijkstra’s Shortest Path Algorithm Pseudo Code (3/3)

```
RelaxEdge( Edge edge )
BEGIN

   sv <-- edge.source
   dv <-- edge.destination

   // Check if found a shorter path for dv
   IF ( distTo[dv] > distTo[sv] + edge.weight )
   THEN
        // found a shorter path to dv via edge: (sv, dv, w)
        // update dv's shortest path

        distTo[dv] <-- distTo[sv] + edge.weight

        edgeTo[dv] <-- edge

        // insert dv & its shorter distance into the queue
        PriQueue.Enqueue( dv, distTo[dv] )

     ELSE
          // did not find a shorter path to dv
          // via edge: (sv, dv, w), nothing to update
          SKIP ;
     ENDIF

END // RelaxEdge
```

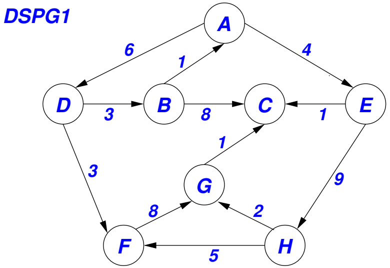
PART IV

Example of

Dijkstra's Shortest Path Algorithm

Example of Dijkstra's Shortest Path Algorithm

For our example of Dijkstra's Shortest Path algorithm we shall use the following *weighted digraph DSPG1*, using *A* as the starting vertex:



Graph *DSPG1* = (V, E), where V & F are:

$$V = \{\ A,\ B,\ C,\ D,\ E,\ F,\ G,\ H\ \}$$

$$E = \{\ (A, D, 6),\ (A, E, 4),\ (B, A, 1),\ (B, C, 8),\ (D, B, 3),\ (D, F, 3),$$
$$(E, C, 1),\ (E, H, 9),\ (F, G, 8),\ (G, C, 1),\ (H, F, 5),\ (H, G, 2)\ \}$$

Representation of Graph DSPG1 using an Adjacency List

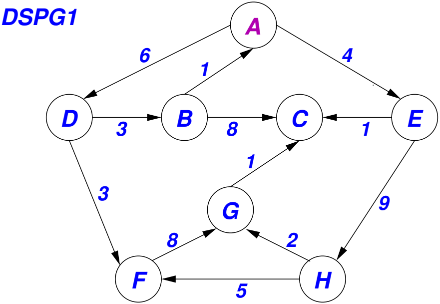
Vertex		
$\boldsymbol{A}$	$(A, \boldsymbol{D}, 6)$	$(A, \boldsymbol{E}, 4)$
$\boldsymbol{B}$	$(B, \boldsymbol{A}, 1)$	$(B, \boldsymbol{C}, 8$
$\boldsymbol{C}$		
$\boldsymbol{D}$	$(D, \boldsymbol{B}, 3)$	$(D, \boldsymbol{F}, 3)$
$\boldsymbol{E}$	$(E, \boldsymbol{C}, 1)$	$(E, \boldsymbol{H}, 9)$
$\boldsymbol{F}$	$(F, \boldsymbol{G}, 8)$	
$\boldsymbol{G}$	$(G, \boldsymbol{C}, 1)$	
$\boldsymbol{H}$	$(H, \boldsymbol{F}, 5)$	$(H, \boldsymbol{G}, 2)$

DSP Step 0: Call `DijkstraShortestPath` (1/2)

- Call **`DijkstraShortestPath( A )`**
- Initialise the data structures by:
  - setting all `edgeTo[v]` to none,
  - setting all `distTo[v]` to ∞,
  - create an empty priority queue,
- Start search from the start vertex *A*, by:
  - setting its distance from itself to *0* &
  - inserting it & its distance into the queue, i.e. *(A, 0)*.

DSP Step 0: Call `DijkstraShortestPath` (2/2)

The state of the graph *DSPG1* & algorithm:



	A	B	C	D	E	F	G	H
`edgeTo[v]`	–	–	–	–	–	–	–	–
`distTo[v]`	*0*	∞	∞	∞	∞	∞	∞	∞
`PriQueue`	⟨ *(A, 0)* ⟩							
`nearestVertex`	–							

DSP Step 1: Search from Nearest Vertex *A* (1/2)

- `PriQueue` equals ⟨*(A, 0)*⟩, so enter `WHILE`-loop.
- Get: **`nearestVertex`** `<-- PriQueue.Dequeue()` = *A*;
  then `PriQueue` is ⟨ ⟩.
- Find *A*'s adjacent vertices: *D* & *E*, using edges: (*A*, *D*, 6), (*A*, *E*, 4).
- Found *shorter path* to *D*, via (*A*, *D*, 6):

```
distTo(D) > distTo(A) + weight(A, D) // INFINITY > 6

edgeTo[D] <-- (A, D, 6)
distTo[D] <-- distTo(A) + weight(A, D) = 0 + 6 = 6
PriQueue.insert( (D, 6) )
```

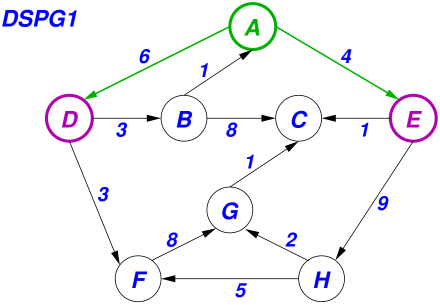
- Found *shorter path* to *E*, via (*A*, *E*, 4):

```
distTo(E) > distTo(A) + weight(A, E) // INFINITY > 4

edgeTo[E] <-- (A, E, 4)
distTo[E] <-- distTo(A) + weight(A, E) = 0 + 4 = 4
PriQueue.insert( (E, 4) )
```

DSP Step 1: Search from Nearest Vertex *A* (2/2)

The state of the graph *DSPG1* & algorithm:



	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
`edgeTo[v]`	–	–	–	(*A*, *D*, 6)	(*A*, *E*, 4)	–	–	–
`distTo[v]`	*0*	∞	∞	*6*	*4*	∞	∞	∞
`PriQueue`	⟨ *(E, 4), (D, 6)* ⟩							
`nearestVertex`	*A*							

DSP Step 2: Search from Nearest Vertex *E* (1/2)

- `PriQueue` equals ⟨*(E, 4)*, (D, 6)⟩, so enter `WHILE`-loop.
- Get: **`nearestVertex`** `<-- PriQueue.Dequeue()` = *E*; then `PriQueue` is ⟨(D, 6)⟩.
- Find *E*'s adjacent vertices: *C* & *H*, using edges: (*E*, *C*, 1), (*E*, *H*, 9).
- Found *shorter path* to *C*, via (*E*, *C*, 1):

```
distTo[C] > distTo[E] + weight(E, C) // INFINITY > 5

edgeTo[C] <-- (E, C, 1)
distTo[C] <-- distTo(E) + weight(E, C) = 4 + 1 = 5
PriQueue.insert( (C, 5) )
```

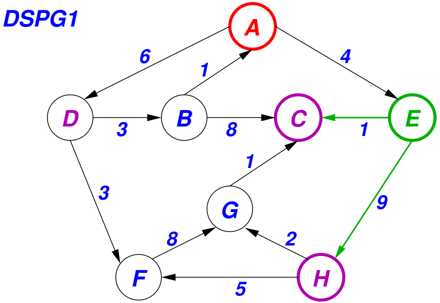
- Found *shorter path* to *H*, via (*E*, *H*, 9):

```
distTo[H] > distTo[E] + weight(E, H) // INFINITY > 13

edgeTo[H] <-- (E, H, 9)
distTo[H] <-- distTo[E] + weight(E, H) = 4 + 9 = 13
PriQueue.insert( (H, 13) )
```

DSP Step 2: Search from Nearest Vertex *E* (2/2)

The state of the graph *DSPG1* & algorithm:



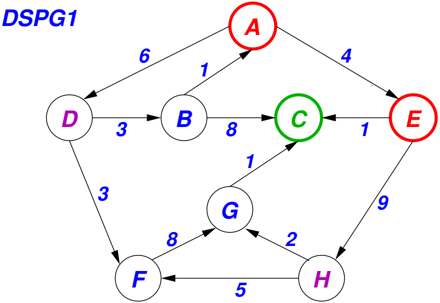
	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
`edgeTo[v]`	–	–	(*E*, *C*, 1)	(*A*, *D*, 6)	(*A*, *E*, 4)	–	–	(*E*, *H*, 9)
`distTo[v]`	*0*	∞	*5*	*6*	*4*	∞	∞	*13*
`PriQueue`	$\langle$ *(C, 5)*, (*D*, 6), *(H, 13)* $\rangle$							
`nearestVertex`	*E*							

DSP Step 3: Search from Nearest Vertex *C* (1/2)

- `PriQueue` equals ⟨*(C, 5)*, *(D, 6), (H, 13)*⟩, so enter `WHILE`-loop.
- Get: **`nearestVertex`** `<-- PriQueue.Dequeue()` = *C*;
- Then: `PriQueue` is ⟨*(D, 6), (H, 13)*⟩.
- Find *C*'s adjacent vertices: **NONE**.
- **No** new *shorter paths* found from *C*, since it has no adjacent vertices.

DSP Step 3: Search from Nearest Vertex *C* (2/2)

The state of the graph *DSPG1* & algorithm:



	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
`edgeTo[v]`	–	–	(*E*, *C*, 1)	(*A*, *D*, 6)	(*A*, *E*, 4)	–	–	(*E*, *H*, 9)
`distTo[v]`	*0*	∞	*5*	*6*	*4*	∞	∞	*13*
`PriQueue`	$\langle$ (*D*, 6), (*H*, 13) $\rangle$							
`nearestVertex`	*C*							

DSP Step 4: Search from Nearest Vertex *D* (1/2)

- `PriQueue` equals ⟨*(D, 6)*, *(H, 13)*⟩, so enter `WHILE`-loop.

- Get: **`nearestVertex`** `<-- PriQueue.Dequeue()` = *D*;
  then `PriQueue` is ⟨*(H, 13)*⟩.

- Find *D*'s adjacent vertices: *B* & *F*, using edges: (*D*, *B*, 3), (*D*, *F*, 3).

- Found *shorter path* to *B*, via (*D*, *B*, 3):

```
distTo[B] > distTo[D] + weight(D, B) // INFINITY > 9

edgeTo[B] <-- (D, B, 3)
distTo[B] <-- distTo(D) + weight(D, B) = 6 + 3 = 9
PriQueue.insert( (B, 9) )
```

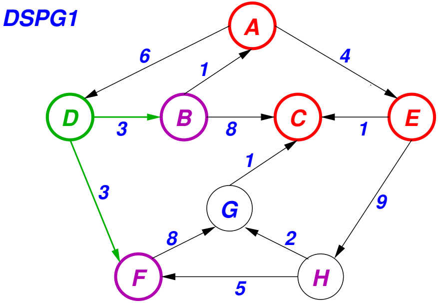
- Found *shorter path* to *F*, via (*D*, *F*, 3):

```
distTo[F] > distTo[D] + weight(D, F) // INFINITY > 9

edgeTo[F] <-- (D, F, 3)
distTo[F] <-- distTo[D] + weight(D, F) = 6 + 3 = 9
PriQueue.insert( (F, 9) )
```

DSP Step 4: Search from Nearest Vertex *D* (2/2)

The state of the graph *DSPG1* & algorithm:



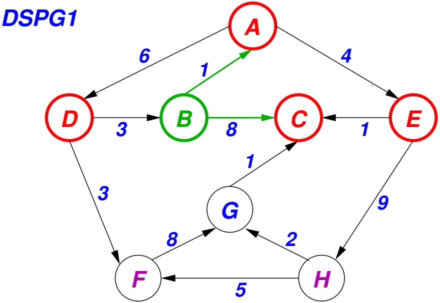
	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
`edgeTo[v]`	–	(*D*, *B*, 3)	(*E*, *C*, 1)	(*A*, *D*, 6)	(*A*, *E*, 4)	(*D*, *F*, 3)	–	(*E*, *H*, 9)
`distTo[v]`	*0*	*9*	*5*	*6*	*4*	*9*	∞	*13*
`PriQueue`	⟨ *(B, 9)*, *(F, 9)*, (*H*, 13) ⟩							
`nearestVertex`	*D*							

DSP Step 5: Search from Nearest Vertex *B* (1/2)

- `PriQueue` equals ⟨*(B, 6)*, *(F, 9)*, *(H, 13)*⟩, so enter `WHILE`-loop.
- Get: **`nearestVertex`** `<-- PriQueue.Dequeue()` = *B*;
  then `PriQueue` is ⟨*(F, 9)*, *(H, 13)*⟩.
- Find *B*'s adjacent vertices: *A* & *C*, using edges: (*B*, *A*, 1), (*B*, *C*, 8).
- **No** *shorter path* found to *A*, via (*B*, *A*, 1):
  `distTo[A] > distTo[B] + weight(B, A) // 0 > 10`
  So nothing to update.
- **No** *shorter path* found to *C*, via (*B*, *C*, 8):
  `distTo[C] > distTo[B] + weight(B, C) // 5 > 17`
  So nothing to update.
- So **no** new shortest paths were found via vertex *B* for vertices *A* & *C*, only *longer paths*, i.e. for *A* – 0 vs. 10, & for *C* – 5 vs. 17.
- Also **no** new vertices were found, i.e. nothing to insert into `PriQueue`.

DSP Step 5: Search from Nearest Vertex *B* (2/2)

The state of the graph *DSPG1* & algorithm:



	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
`edgeTo[v]`	–	(*D*, *B*, 3)	(*E*, *C*, 1)	(*A*, *D*, 6)	(*A*, *E*, 4)	(*D*, *F*, 3)	–	(*E*, *H*, 9)
`distTo[v]`	*0*	*9*	*5*	*6*	*4*	*9*	∞	*13*
`PriQueue`	⟨ (*F*, 9), (*H*, 13) ⟩							
`nearestVertex`	*B*							

DSP Step 6: Search from Nearest Vertex *F* (1/2)

- `PriQueue` equals ⟨*(F, 9)*, *(H, 13)*⟩, so enter `WHILE`-loop.
- Get: **`nearestVertex`** `<-- PriQueue.Dequeue()` = *F*;
  then `PriQueue` is ⟨*(H, 13)*⟩.
- Find *F*'s adjacent vertex: *G*, using edges: (*F*, *G*, 8).
- Found *shorter path* to *G*, via (*F*, *G*, 8):

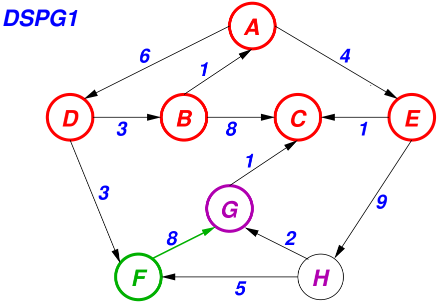
```
distTo[G] > distTo[F] + weight(F, G) // INFINITY > 17

edgeTo[G] <-- (F, G, 8)
distTo[G] <-- distTo(F) + weight(F, G) = 9 + 8 = 17
PriQueue.insert( (G, 17) )
```

- So a new shortest path was found via vertex *F* for vertex *G* of 17.

DSP Step 6: Search from Nearest Vertex *F* (2/2)

The state of the graph *DSPG1* & algorithm:



	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
`edgeTo[v]`	–	(*D*, *B*, 3)	(*E*, *C*, 1)	(*A*, *D*, 6)	(*A*, *E*, 4)	(*D*, *F*, 3)	(*F*, *G*, 8)	(*E*, *H*, 9)
`distTo[v]`	*0*	*9*	*5*	*6*	*4*	*9*	*17*	*13*
`PriQueue`	⟨ (*H*, 13), *(G, 17)* ⟩							
`nearestVertex`	*F*							

DSP Step 7: Search from Nearest Vertex *H* (1/2)

- `PriQueue` equals ⟨*(H, 13)*, *(G, 17)*⟩, so enter `WHILE`-loop.
- Get: **`nearestVertex`** `<-- PriQueue.Dequeue()` = *H*;
  then `PriQueue` is ⟨*(G, 17)*⟩.
- Find *H*'s adjacent vertices: *F* & *G*, using edges: (*H*, *F*, 5), (*H*, *G*, 2).
- **No** *shorter path* found to *F*, via (*H*, *F*, 5):

  ```
  distTo[F] > distTo[H] + weight(H, F) // 9 > 18
  ```

  So only a *longer path* found for vertex *F* – 9 vs. 18, so nothing to update.
- **But** found a *shorter path* to *G*, via (*H*, *G*, 2):

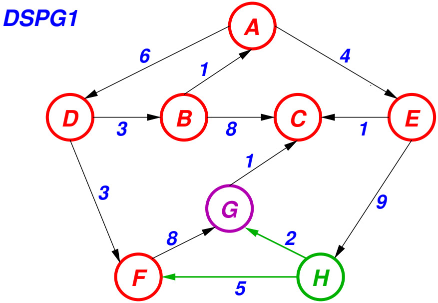
  ```
  distTo[G] > distTo[H] + weight(H, G) // 17 > 15

  edgeTo[G] <-- (H, G, 2)
  distTo[G] <-- distTo(H) + weight(H, G) = 13 + 2 = 15
  PriQueue.insert( (G, 15) )
  ```

- So a new *shorter path* was found via vertex *H* for vertex *G* – 17 vs. 15,
  so replace (*G*, 17) with (*G*, 15) in `PriQueue`.

DSP Step 7: Search from Nearest Vertex *H* (2/2)

The state of the graph *DSPG1* & algorithm:



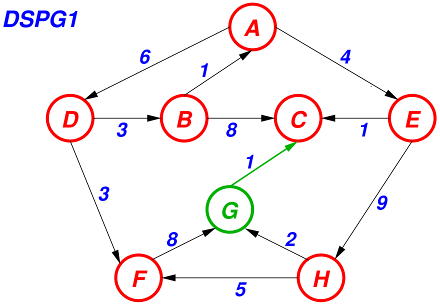
	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
`edgeTo[v]`	–	(*D*, *B*, 3)	(*E*, *C*, 1)	(*A*, *D*, 6)	(*A*, *E*, 4)	(*D*, *F*, 3)	**(*H*, *G*, 2)**	(*E*, *H*, 9)
`distTo[v]`	*0*	*9*	*5*	*6*	*4*	*9*	***15***	*13*
`PriQueue`	⟨ ***(G, 15)*** ⟩							
`nearestVertex`	*H*							

DSP Step 8: Search from Nearest Vertex *G* (1/2)

- `PriQueue` equals ⟨*(G, 15)*⟩, so enter `WHILE`-loop.
- Get: **`nearestVertex`** `<-- PriQueue.Dequeue() =` *G*;
  then `PriQueue` is ⟨ ⟩.
- Find *G*'s adjacent vertices: *C*; using edges: (*G*, *C*, 1).
- **No** *shorter path* found to *C*, via (*G*, *C*, 1):
  `distTo[C] > distTo[G] + weight(G, C) // 5 > 16`
  So nothing to update.
- So **no** new shortest path was found via vertex *G* for vertex *C*, only a *longer path*, i.e. for *C* – 5 vs. 16.
- Also **no** new vertices were found, i.e. nothing to insert into `PriQueue`.

DSP Step 8: Search from Nearest Vertex *G* (2/2)

The state of the graph *DSPG1* & algorithm:



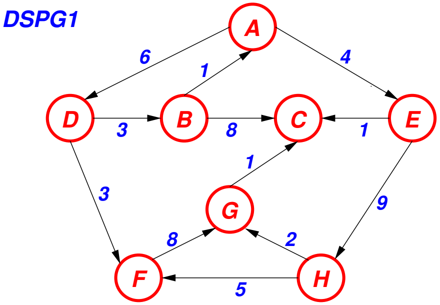
	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
`edgeTo[v]`	–	(*D*, *B*, 3)	(*E*, *C*, 1)	(*A*, *D*, 6)	(*A*, *E*, 4)	(*D*, *F*, 3)	(*H*, *G*, 2)	(*E*, *H*, 9)
`distTo[v]`	*0*	*9*	*5*	*6*	*4*	*9*	*15*	*13*
`PriQueue`	⟨ ⟩							
`nearestVertex`	*G*							

DSP Step 9: Termination (1/2)

- `PriQueue` equals $\langle\,\rangle$, so **exit** the `WHILE`-loop.
- So the queue is empty & exited WHILE loop, therefore have processed all the vertices,
- **`DijkstraShortestPath( A )`** **terminates**.

DSP Step 9: Termination (2/2)

The state of the graph *DSPG1* & algorithm:



	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
`edgeTo[v]`	–	(*D*, *B*, 3)	(*E*, *C*, 1)	(*A*, *D*, 6)	(*A*, *E*, 4)	(*D*, *F*, 3)	(*H*, *G*, 2)	(*E*, *H*, 9)
`distTo[v]`	*0*	*9*	*5*	*6*	*4*	*9*	*15*	*13*
`PriQueue`	⟨ ⟩							
`nearestVertex`	–							

DSP Step 10: Final Shortest Paths from A

The final state with the **shortest paths** from *A* to all other vertices *B* to *H*.

To construct a *shortest path* from the source vertex *A* to another vertex, e.g. *F*, start at *F*'s `edgeTo`[*F*] & link backwards to *D* then to *A*.

Vertex	`edgeTo[v]`	`distTo[v]`	Path Edges	Path
A	–	**0**	⟨ ⟩	⟨*A*⟩
B	(*D*, *B*, 3)	**9**	⟨ (*A*, *D*, 6), (*D*, *B*, 3) ⟩	⟨ *A*, *D*, *B* ⟩
C	(*E*, *C*, 1)	**5**	⟨ (*A*, *E*, 4), (*E*, *C*, 1) ⟩	⟨ *A*, *E*, *C* ⟩
D	(*A*, *D*, 6)	**6**	⟨ (*A*, *D*, 6) ⟩	⟨ *A*, *D* ⟩
E	(*A*, *E*, 4)	**4**	⟨ (*A*, *E*, 4) ⟩	⟨ *A*, *E* ⟩
F	(*D*, *F*, 3)	**9**	⟨ (*A*, *D*, 6), (*D*, *F*, 3) ⟩	⟨ *A*, *D*, *F* ⟩
G	(*H*, *G*, 2)	**15**	⟨ (*A*, *E*, 4), (*E*, *H*, 9), (*H*, *G*, 2) ⟩	⟨ *A*, *E*, *H*, *G* ⟩
H	(*E*, *H*, 9)	**13**	⟨ (*A*, *E*, 4), (*E*, *H*, 9) ⟩	⟨ *A*, *E*, *H* ⟩