CSE 373

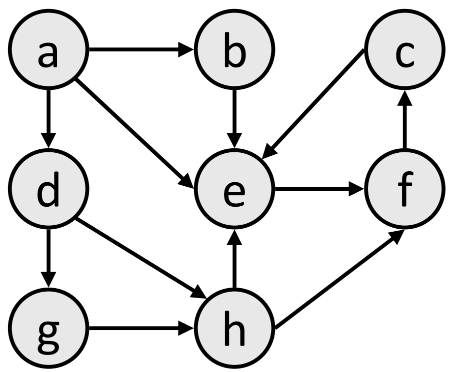
Graphs 2: Dijkstra's Algorithm

reading: Weiss 9.3

slides created by Marty Stepp
http://www.cs.washington.edu/373/

Recall: DFS, BFS

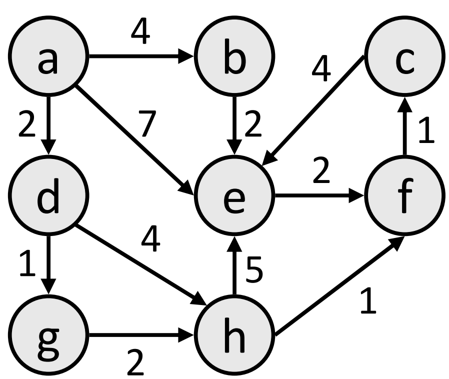
- **depth-first search** (DFS): Explore each possible path as far as possible before backtracking.
  - Often implemented recursively.
  - DFS paths from *a* to all vertices (assuming ABC edge order):
    - to b: {a, b}
    - to c: {a, b, e, f, c}
    - to d: {a, d}
    - to e: {a, b, e}
    - to f: {a, b, e, f}
    - to g: {a, d, g}
    - to h: {a, d, g, h}



- **breadth-first search** (BFS): Take one step down all paths and then immediately backtrack.
  - A queue of vertices to visit.
  - Always returns shortest path (one with fewest edges):
    - to b: {a, b}
    - to c: **{a, e, f, c}**
    - to d: {a, d}
    - to e: **{a, e}**
    - to f: **{a, e, f}**
    - to g: {a, d, g}
    - to h: **{a, d, h}**

DFS/BFS and weight

- DFS and BFS do not consider edge weights.
  - The minimum weight path is not necessarily the shortest path.
  - Sometimes weight is more important:
    - example: plane flight costs, network transmission (latency btwn servers)
    - BFS(a,f) yields [a,e,f], but [a,d,g,h,f] has lower cost (6 vs. 9)



Dijkstra's Algorithm

- **Dijkstra's algorithm**: Finds the minimum-weight path between a pair of vertices in a weighted directed graph.
  - Solves the "one vertex, shortest path" problem in weighted graphs.
  - Made by famous computer scientist Edsger Dijkstra (look him up!)
  - *basic algorithm concept*: Create a table of information about the currently known best way to reach each vertex (cost, previous vertex), and improve it until it reaches the best solution.
- *Example:* In a graph where vertices are cities and weighted edges are roads between cities, Dijkstra's algorithm can be used to find the shortest route from one city to any other.

Dijkstra pseudocode

```
function dijkstra(v1, v2):
    for each vertex v:                    // Initialize vertex info
        v's cost := infinity.
        v's previous := none.
    v1's cost := 0.
    pqueue := {all vertices, ordered by distance}.

    while pqueue is not empty:
        v := remove vertex from pqueue with minimum cost.
        mark v as visited.
        for each unvisited neighbor n of v:
            cost := v's cost + weight of edge (v, n).

            if cost < n's cost:
                n's cost := cost.
                n's previous := v.

    reconstruct path from v2 back to v1, following previous pointers.
```

Dijkstra example

- dijkstra(A, F);

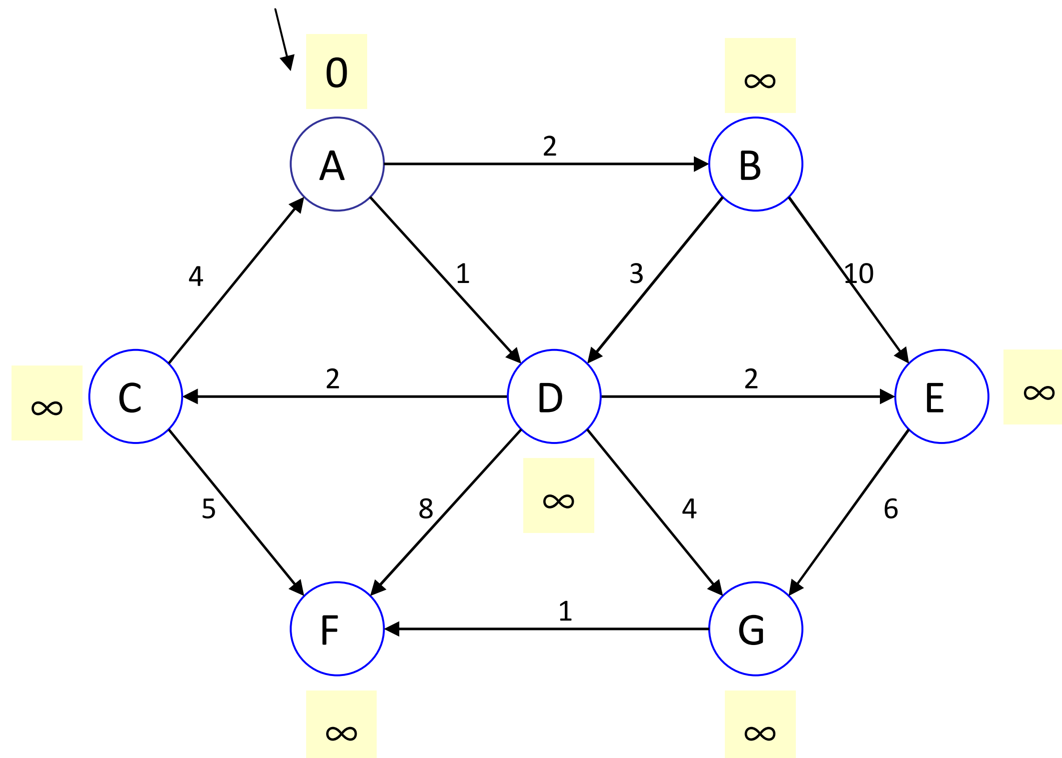
```
function dijkstra(v1, v2):
   for each vertex v:   // Initialize vertex info
      v's cost := infinity.
      v's previous := none.
   v1's cost := 0.
   pqueue := {all vertices,
              by distance}.

   while pqueue is not empty:
      v := pqueue.removeMin().
      mark v as visited.
      for each unvisited neighbor n of v:
         cost := v's cost + edge(v, n)'s weight.

         if cost < n's cost:
            n's cost := cost.
            n's previous := v.

   reconstruct path from v2 back to v1,
    following previous pointers.
```

v_1's distance := 0.   all other distances := ∞.



pqueue = [**A:0**, B:∞, C:∞, D:∞, E:∞, F:∞, G:∞]

Dijkstra example

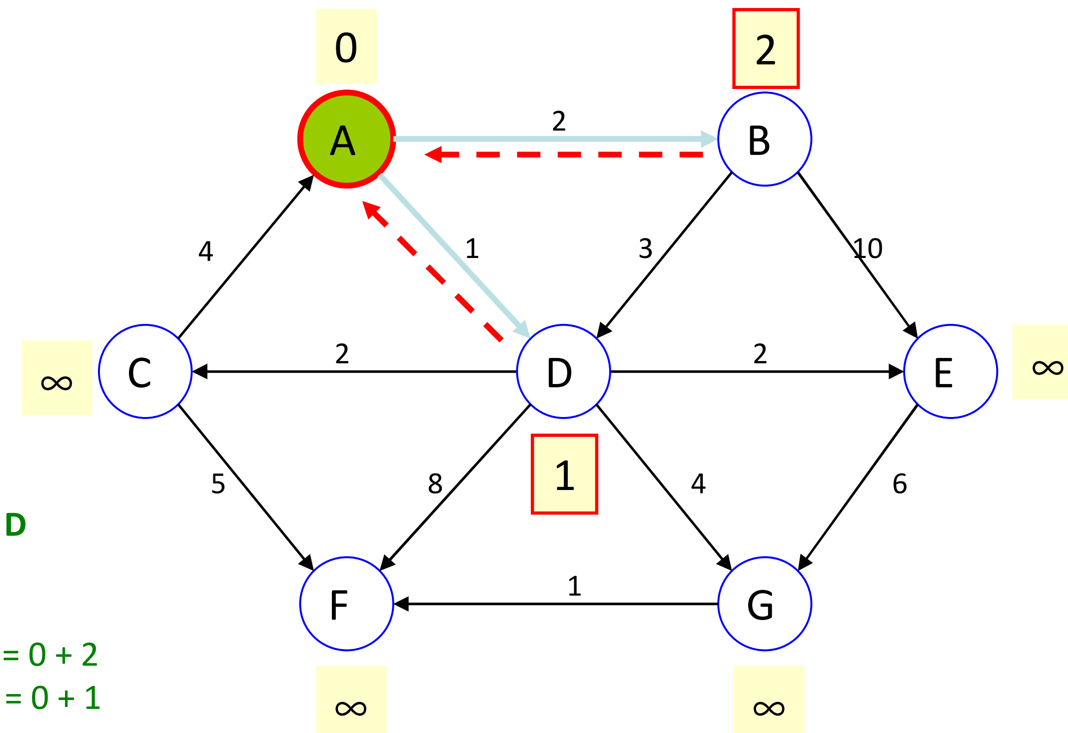
- dijkstra(A, F);

function **dijkstra**(v_1, v_2):
  for each vertex v: // Initialize vertex info
    v's cost := infinity.
    v's previous := none.
  v_1's cost := 0.
  pqueue := {all vertices,
          by distance}.

  while *pqueue* is not empty:
    v := *pqueue*.removeMin(). **// A**
    mark *v* as visited.
    for each unvisited neighbor *n* of *v*:   **// B, D**
      cost := *v*'s cost + edge(*v*, *n*)'s weight.

      if *cost* < *n*'s cost: // B's cost = 0 + 2
        n's cost := *cost*.   // D's cost = 0 + 1
        n's previous := *v*.

  reconstruct path from v_2 back to v_1,
  following previous pointers.



pqueue = [**D:1**, **B:2**, C:∞, E:∞, F:∞, G:∞]

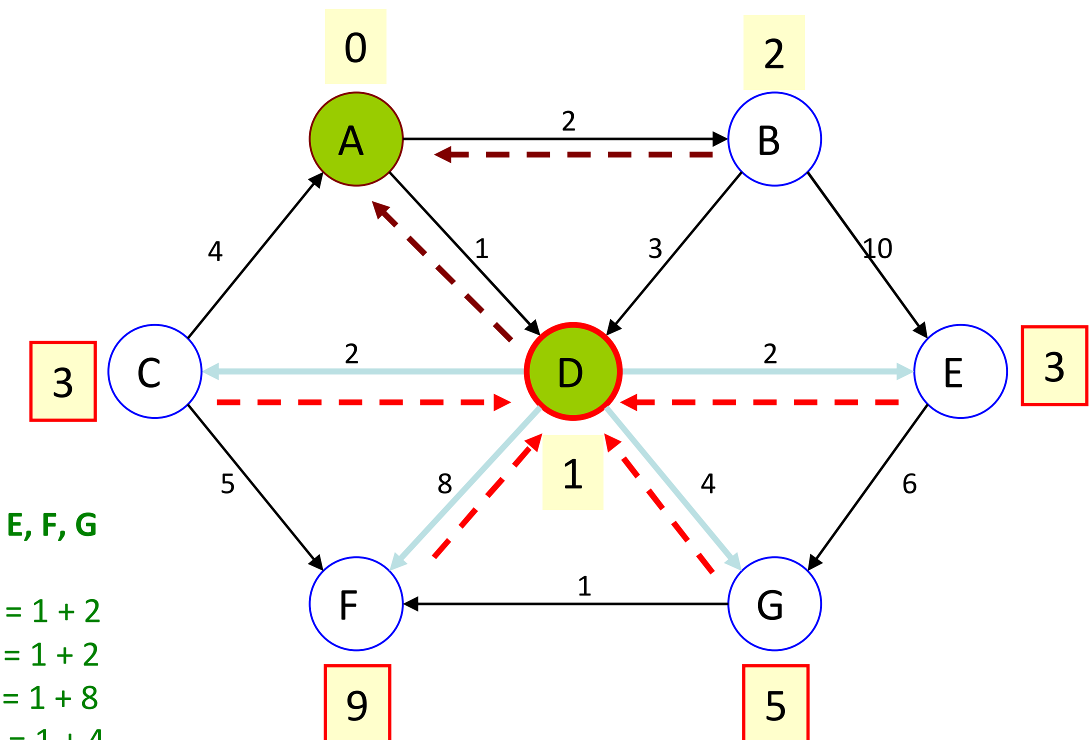
Dijkstra example

- dijkstra(A, F);

```
function dijkstra(v1, v2):
   for each vertex v:   // Initialize vertex info
      v's cost := infinity.
      v's previous := none.
   v1's cost := 0.
   pqueue := {all vertices,
                 by distance}.

   while pqueue is not empty:
      v := pqueue.removeMin().  // D
      mark v as visited.
      for each unvisited neighbor n of v:   // C, E, F, G
         cost := v's cost + edge(v, n)'s weight.
                                       // C's cost = 1 + 2
         if cost < n's cost:           // E's cost = 1 + 2
            n's cost := cost.          // F's cost = 1 + 8
            n's previous := v.         // G's cost = 1 + 4

   reconstruct path from v2 back to v1,
    following previous pointers.
```



pqueue = [B:2, **C:3, E:3, G:5, F:9**]

Dijkstra example

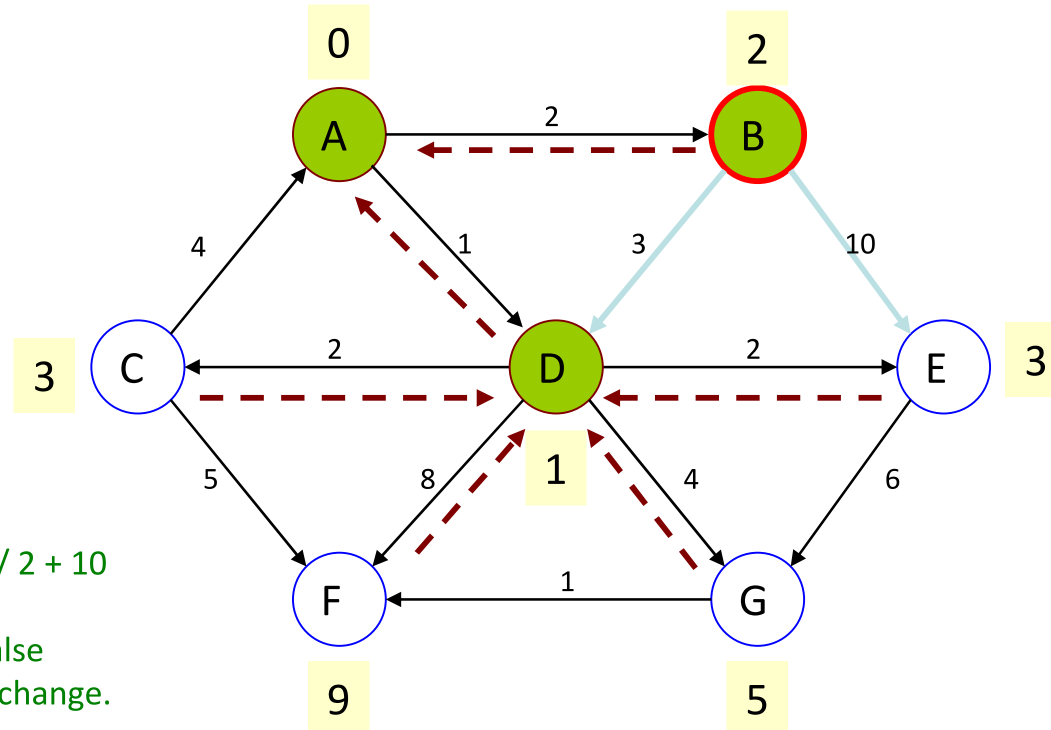
- dijkstra(A, F);

function **dijkstra**(v_1, v_2):
    for each vertex v: // Initialize vertex info
        v's cost := infinity.
        v's previous := none.
    v_1's cost := 0.
    pqueue := {all vertices,
                    by distance}.

    while *pqueue* is not empty:
        v := *pqueue*.removeMin().   **// B**
        mark *v* as visited.
        for each **unvisited** neighbor *n* of *v*:   **// E**
            cost := *v*'s cost + edge(*v*, *n*)'s weight.   // 2 + 10

            if *cost* < *n*'s cost: // 12 > 3; false
                n's cost := *cost*.   // no costs change.
                n's previous := *v*.

    reconstruct path from v_2 back to v_1,
     following previous pointers.



pqueue = [C:3, E:3, G:5, F:9]

Dijkstra example

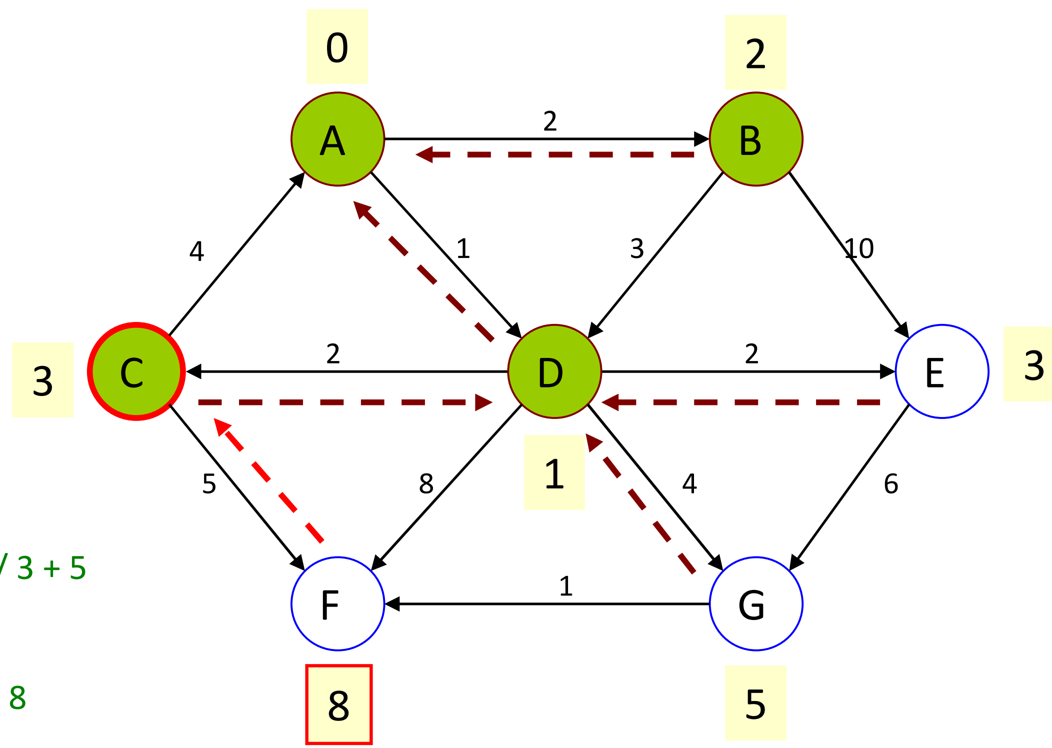
- dijkstra(A, F);

function **dijkstra**(v_1, v_2):
  for each vertex v: // Initialize vertex info
    v's cost := infinity.
    v's previous := none.
  v_1's cost := 0.
  pqueue := {all vertices,
          by distance}.

  while *pqueue* is not empty:
    v := *pqueue*.removeMin(). **// C**
    mark *v* as visited.
    for each unvisited neighbor *n* of *v*:   **// F**
      cost := *v*'s cost + edge(*v*, *n*)'s weight.   // 3 + 5

      if *cost* < *n*'s cost: // 8 < 9
        n's cost := *cost*.   // F's cost = 8
        n's previous := *v*.

  reconstruct path from v_2 back to v_1,
   following previous pointers.



pqueue = [E:3, G:5, **F:8**]

Dijkstra example

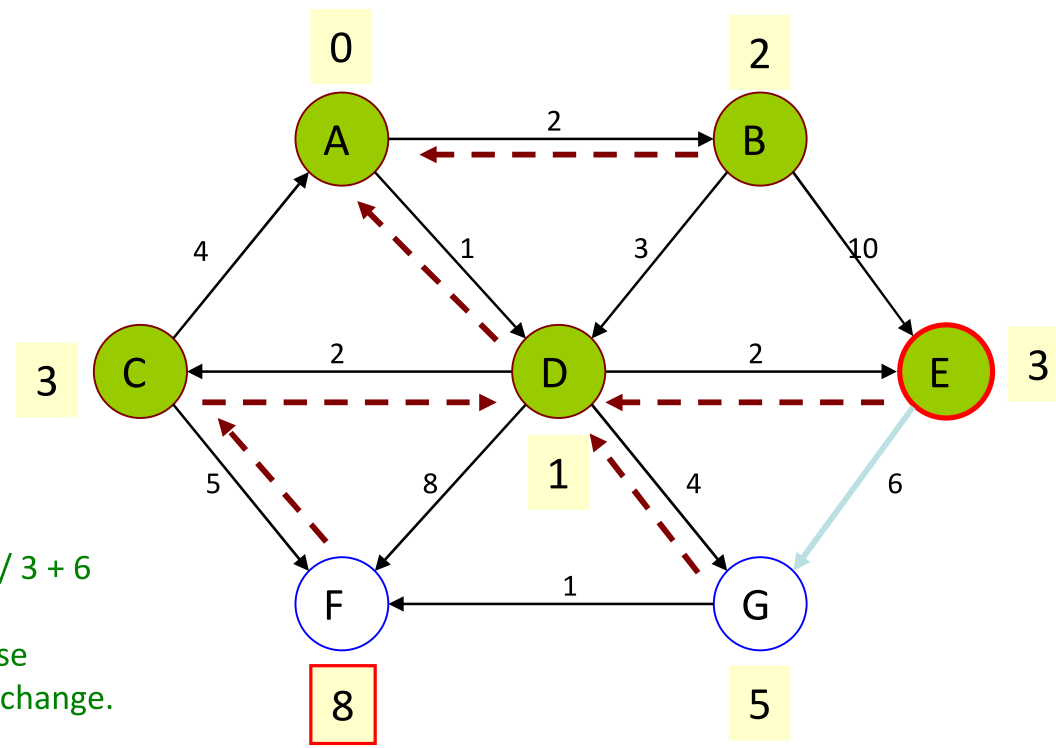
- dijkstra(A, F);

```
function dijkstra(v1, v2):
   for each vertex v:   // Initialize vertex info
      v's cost := infinity.
      v's previous := none.
   v1's cost := 0.
   pqueue := {all vertices,
              by distance}.

   while pqueue is not empty:
      v := pqueue.removeMin().  // E
      mark v as visited.
      for each unvisited neighbor n of v:   // G
         cost := v's cost + edge(v, n)'s weight.   // 3 + 6

         if cost < n's cost:            // 9 > 5; false
            n's cost := cost.           // no costs change.
            n's previous := v.

   reconstruct path from v2 back to v1,
    following previous pointers.
```



pqueue = [G:5, F:8]

Dijkstra example

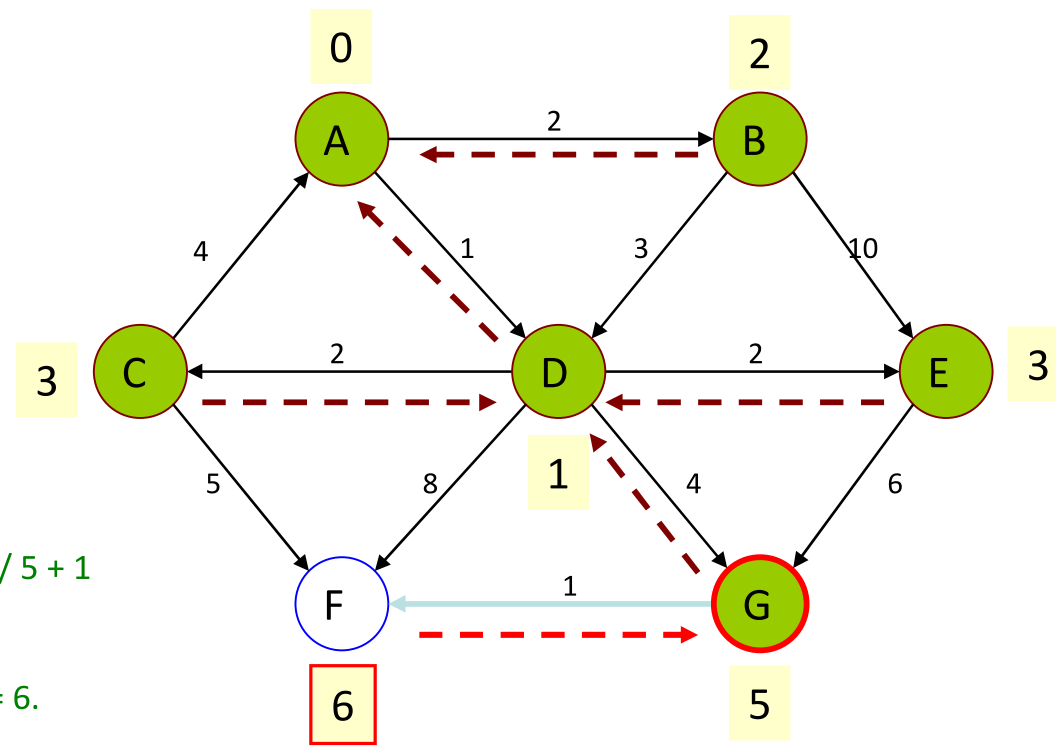
- dijkstra(A, F);

function **dijkstra**(v_1, v_2):
  for each vertex v: // Initialize vertex info
    v's cost := infinity.
    v's previous := none.
  v_1's cost := 0.
  pqueue := {all vertices,
            by distance}.

  while *pqueue* is not empty:
    v := *pqueue*.removeMin(). **// G**
    mark *v* as visited.
    for each unvisited neighbor *n* of *v*:   **// F**
      cost := *v*'s cost + edge(*v*, *n*)'s weight.   // 5 + 1

      if *cost* < *n*'s cost: // 6 < 8
        n's cost := *cost*.   // F's cost = 6.
        n's previous := *v*.

  reconstruct path from v_2 back to v_1,
   following previous pointers.



pqueue = [**F:6**]

Dijkstra example

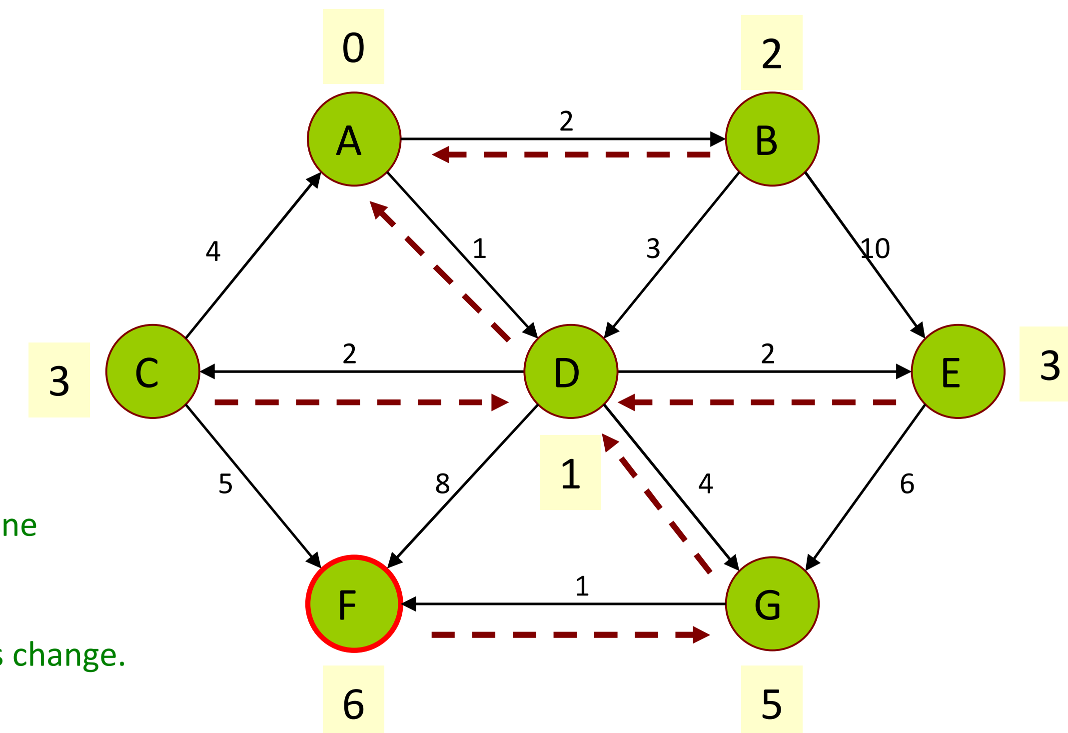
- dijkstra(A, F);

function **dijkstra**(v_1, v_2):
  for each vertex v: // Initialize vertex info
    v's cost := infinity.
    v's previous := none.
  v_1's cost := 0.
  pqueue := {all vertices,
      by distance}.

  while *pqueue* is not empty:
    v := *pqueue*.removeMin(). **// F**
    mark *v* as visited.
    for each unvisited neighbor *n* of *v*:   // none
      cost := *v*'s cost + edge(*v*, *n*)'s weight.

      if *cost* < *n*'s cost: // no costs change.
        n's cost := *cost*.
        n's previous := *v*.

  reconstruct path from v_2 back to v_1,
   following previous pointers.



pqueue = []

Dijkstra example

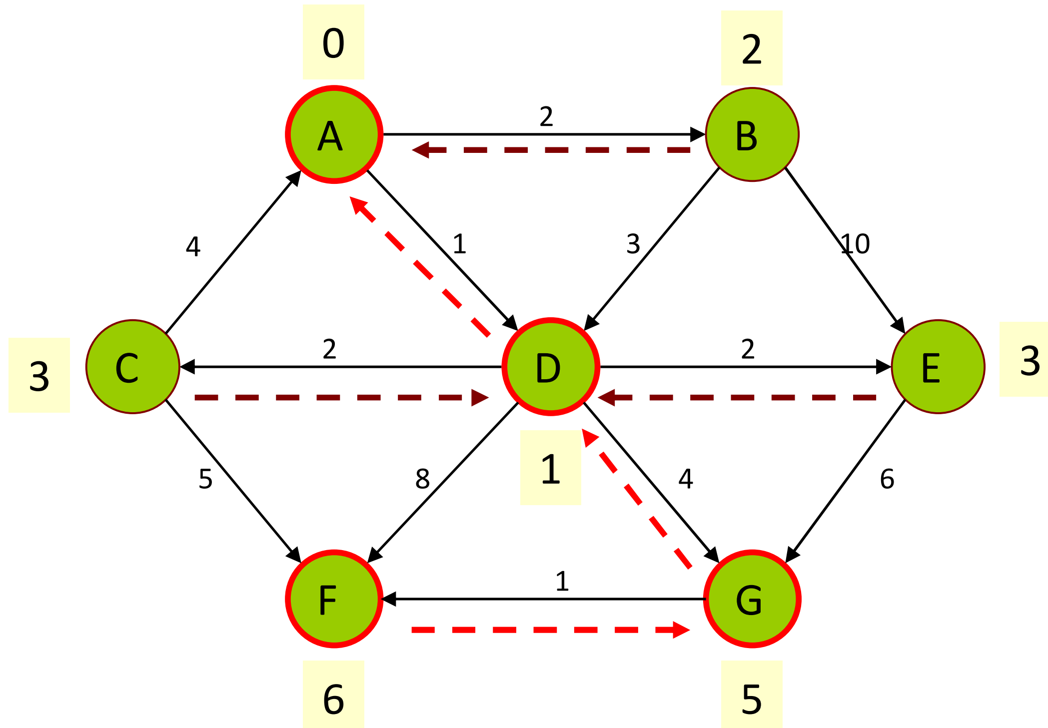
- dijkstra(A, F);

function **dijkstra**(v_1, v_2):
  for each vertex v: // Initialize vertex info
    v's cost := infinity.
    v's previous := none.
  v_1's cost := 0.
  pqueue := {all vertices,
          by distance}.

  while *pqueue* is not empty:
    v := *pqueue*.removeMin().
    mark *v* as visited.
    for each unvisited neighbor *n* of *v*:
      cost := *v*'s cost + edge(*v*, *n*)'s weight.

      if *cost* < *n*'s cost:
        n's cost := *cost*.
        n's previous := *v*.

  reconstruct path from v_2 back to v_1,
  following previous pointers.



// path = [A, D, G, F]

Algorithm properties

- Dijkstra's algorithm is a *greedy algorithm*:
  - Make choices that currently seem the best.
  - Locally optimal does not always mean globally optimal.

- It is correct because it maintains the following two properties:
  - 1) for every marked vertex, the current recorded cost is the lowest cost to that vertex from the source vertex.
  - 2) for every unmarked vertex *v*, its recorded distance is shortest path distance to *v* from source vertex, considering only currently known vertices and *v*.

Dijkstra's runtime

- For sparse graphs, (i.e. graphs with much less than $|V|^2$ edges) Dijkstra's is implemented most efficiently with a priority queue.
  - initialization: $O(|V|)$
  - while loop: $O(|V|)$ times
    - remove min-cost vertex from *pq*: $O(\log |V|)$
    - potentially perform $|E|$ updates on cost/previous
    - update costs in *pq*: $O(\log |V|)$
  - reconstruct path: $O(|E|)$
  - Total runtime: $O(|V| \log |V| + |E| \log |V|)$
    - = $\mathbf{O(|E| \log |V|)}$, because $|V| = O(|E|)$ if graph is connected
    - if a list is used instead of a *pq*: $O(|V^2| + |E|) = O(|V|^2)$

Dijkstra exercise

- Use Dijkstra's algorithm to determine the lowest cost path from vertex *A* to all of the other vertices in the graph.
  - Keep track of previous vertices so that you can reconstruct the path.