Ian Piper
CSCI203

Algorithms and Data Structures

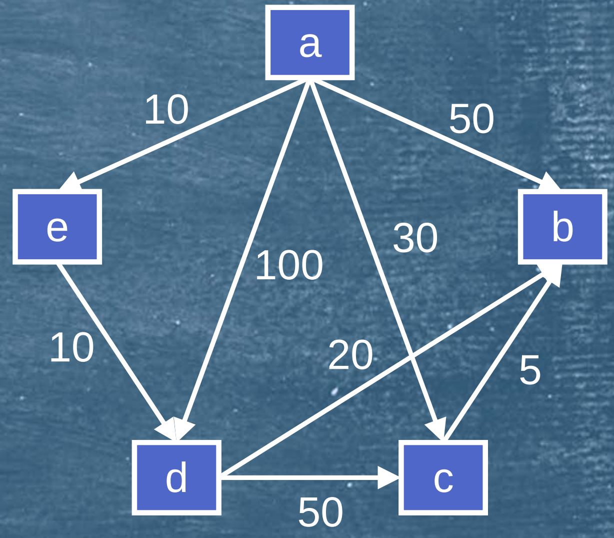
Week 9 – Lecture A

Weighted Graphs

- Frequently, we find that travelling along an edge in a graph has some associated cost (or profit) associated with it:
  - The distance along the edge;
  - The cost of petrol;
  - The time of travel;
  - Etc.
- We call these *Weighted Graphs*.
- We call the edge values *weights*.

Representation

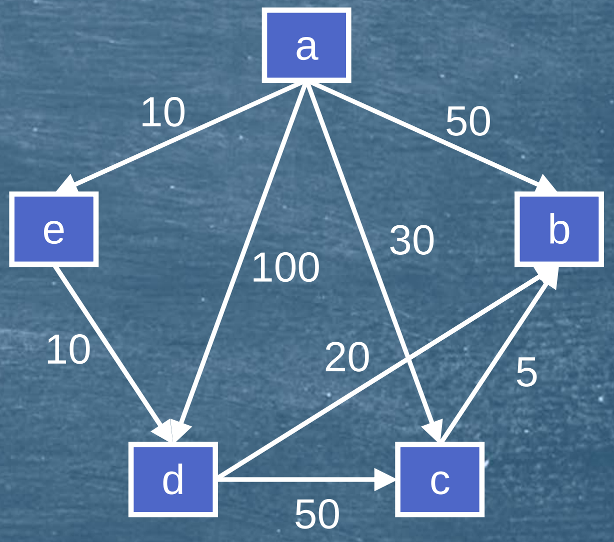
- We extend our previous graph definition as follows:
  - A weighted graph, *G*, consists of the ordered sequence, (*V*, *E*, *W*) where *V* and E are the vertices and edges and *W* are the edge weights.
- Consider the weighted graph shown to the right:
- *V*={a, b, c, d, e}
- *E*={(a, b), (a, c), (a, d), (a, e), (c, b), (d, b), (d, c), (e, d)}
- *W* is a function that maps edges to weights:
  - E.g. *W*((a, b))=50.



Representation

- We can extend our adjacency matrix representation of a graph by replacing the zero-one existence value with the edge weight.
- Thus the graph:
- Is written:

—	50	30	100	10
—	—	—	—	—
—	5	—	—	—
—	20	50	—	—
—	—	—	10	—



Representation

- In this example, "—" indicates that no edge exists.
- The actual value used to indicate this fact will depend on the nature of the weights:
  - E.g. if all weights are non-zero use 0.
  - We often use ∞ to represent missing edges.
- We can also use the adjacency list representation:
  - We just need to pair each edge with its corresponding weight.

Shortest Path

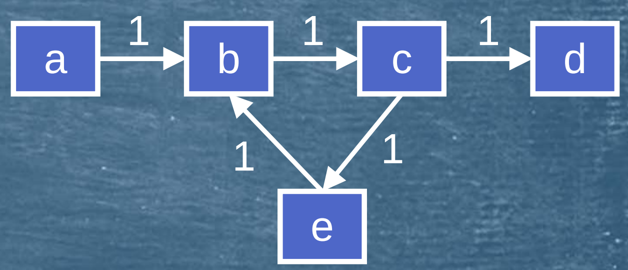
- A common problem associated with weighted graphs is finding the shortest path between vertices.
- There are several versions of this problem:
  - Single Source—All destinations;
  - Single Source—Single Destination;
  - All Sources—All Destinations;
  - All Sources—Single Destination.
- Each has applications in the real world.
- We will start by looking at the first of these types.

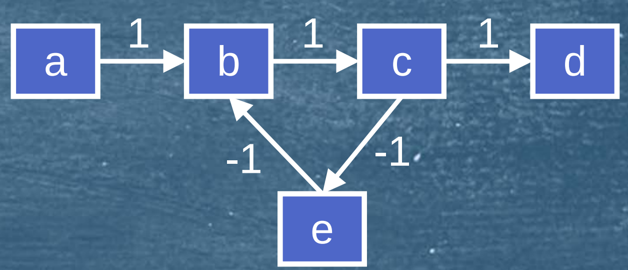
Single Source—All Destinations

- This problem is stated as follows:
  - Starting at some source vertex, *s*, find the shortest path from *s* to each other reachable vertex in the graph.
- As we shall see later, the solution to this problem can be used as a basis for solving all of the other shortest path formulations.
- We will examine two algorithms for solving this problem:
  - Dijkstra.
  - Bellman Ford.
- Each has advantages in certain cases.

Negative Weights

- There is no *a priori* reason why the edge weights in a graph must be positive, but negative edge weights can cause problems.
- Consider the following graph:
  - Clearly the length of the shortest path from a to d is 3.
- But what if we change the weights?
  - Now what is the shortest path?
- The problem is that we now have a negative cost cycle.





What is a Path?

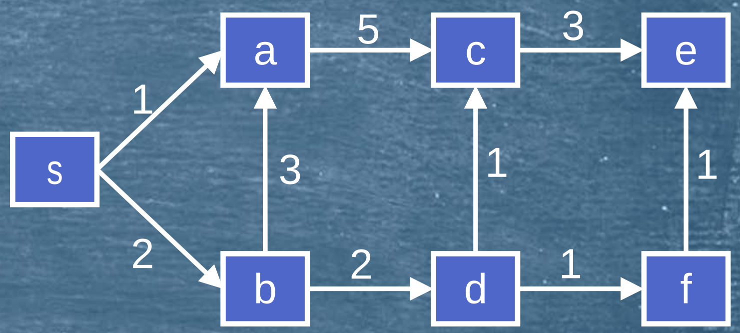
- While it is obvious what a path is, we should define it formally.
- A path p from vertex v_0 to vertex v_k is an ordered sequence $(v_0, v_1, \ldots, v_{k-1}, v_k)$ where each pair of vertices $(v_i, v_{i+1}) \in E$.
- The weight of path p, $W(p)$ is the sum of the edge weights:
  - $W(p) = \sum_{i=0}^{k-1} W((v_i, v_{i+1}))$

What doesn't Work?

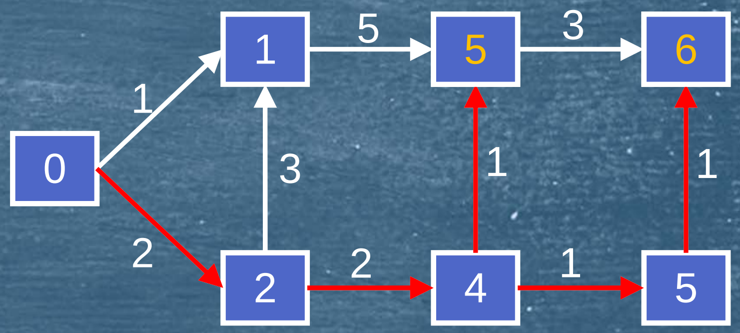
- We might be tempted to try using a technique we already know for traversing a graph, breadth first search, in our search for shortest paths.
- Unfortunately, this does not always work.
- The two definitions of shortest path:
  - Fewest edges;
  - Smallest weight;
- May not always coincide.

BFS: an Example

- Consider the following graph:



- If we conduct a BFS traversal we get the following path weights:



- Is this the best we can do?
- No!
- We need a different approach.

Dijkstra's Algorithm

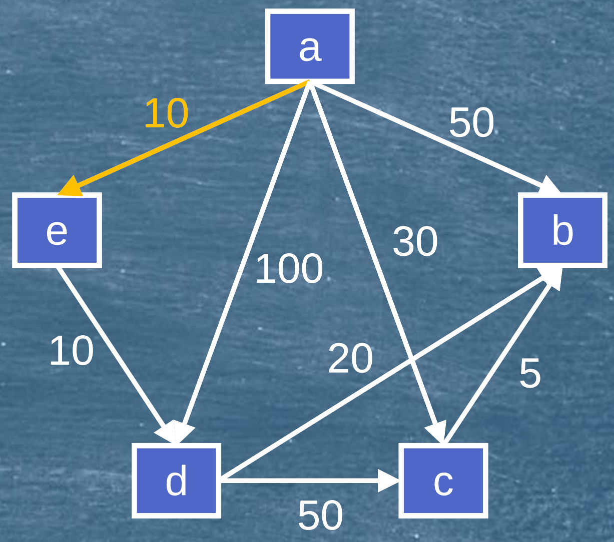
- This algorithm works by dividing the vertices into two sets, *S* and *C*.
- At each iteration; *S* contains the set of nodes that have already been chosen.
  - This is the *selected* set.
- At each iteration; *C* contains the set of nodes that have not yet been chosen:
  - This is the *candidate* set.
- At each step we move the node which is cheapest to reach from *C* to *S*.

Dijkstra's Algorithm

- We also need a function *D* such that $D(c_i)$ is the shortest distance we have so far found from vertex *s* to vertex c_i in the candidate set *C*.
- Initially:
  - The selected set, *S*, just contains the start vertex;
  - The candidate set, *C*, contains all the other vertices;
  - The distance function, *D*() has value 0 for vertex *s* and is infinite for all other vertices.
- We start by re-evaluating *D* for each vertex directly reachable from vertex *s*.

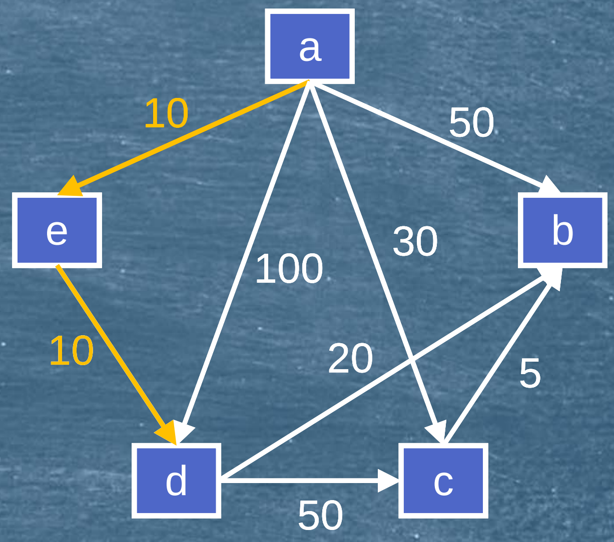
Dijkstra's Algorithm: an Example

- Step 0:
- S={a}
- C={b, c, d, e}
- D()= 50, 30, 100, 10
- We now select the minimum value of D, D(e)=10.



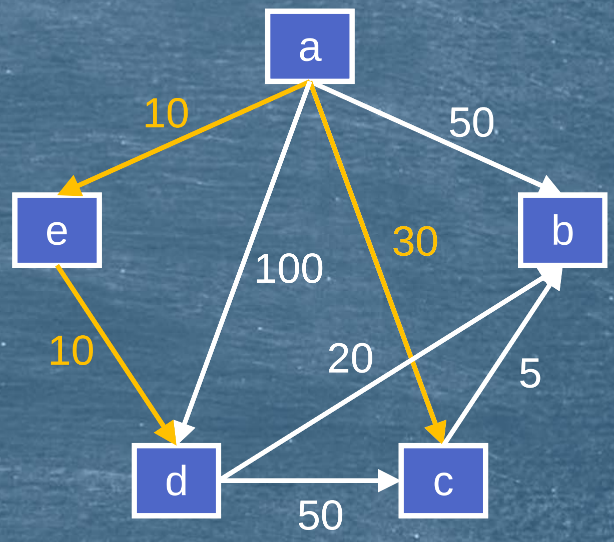
Dijkstra's Algorithm: an Example

- Step 1: move vertex e from *C* to *S*.
- *S*={a, e}
- *C*={b, c, d}
- We now update *D* by looking at vertices we can reach from vertex e.
- *D*()= 50, 30, 20
- We now select the minimum value of *D*, *D*(d)=20.



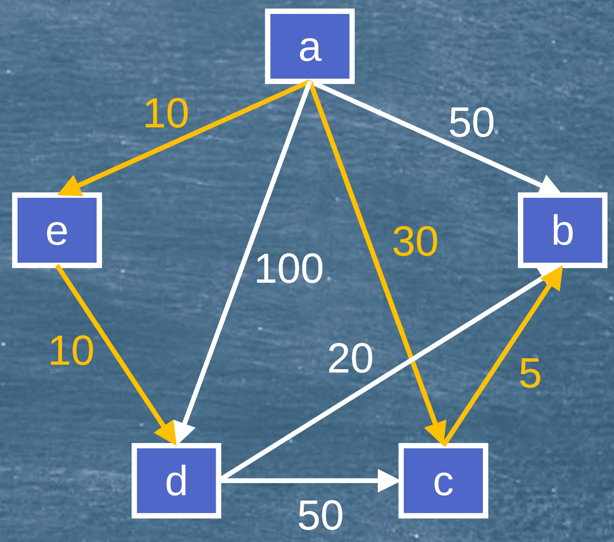
Dijkstra's Algorithm: an Example

- Step 2: move vertex d from *C* to *S*.
- *S*={a, e, d}
- *C*={b, c}
- We now update *D* by looking at vertices we can reach from vertex d.
- *D*()= 40, 30
- We now select the minimum value of *D*, *D*(c)=30.



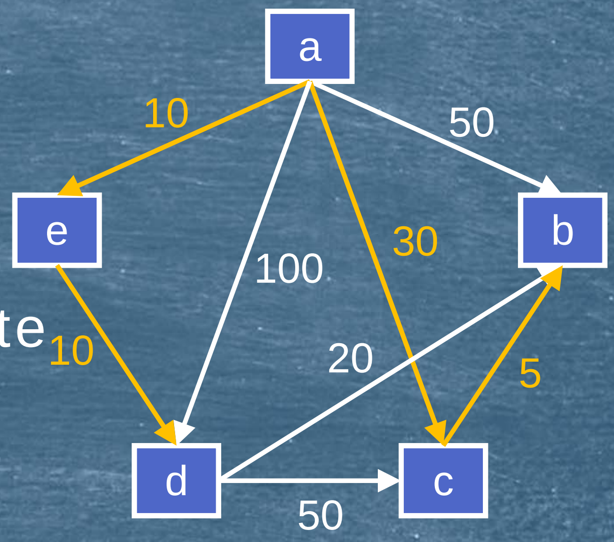
Dijkstra's Algorithm: an Example

- Step 3: move vertex c from *C* to *S*.
- *S*={a, e, d, c}
- *C*={b}
- We now update *D* by looking at vertices we can reach from vertex c.
- *D*()= 35
- We now select the minimum value of *D*, *D*(b)=35.



Dijkstra's Algorithm: an Example

- Step 4: move vertex b from *C* to *S*.
- *S*={a, e, d, c, b}
- *C*={}
- We now have no remaining candidate vertices; we have finished.
- *W*(a)=0, *W*(e)=10, *W*(d)=20, *W*(c)=30, *W*(b)=35.



Dijkstra's Algorithm: Pseudocode

```
Procedure Dijkstra(G: array[1..n, 1..n]): array [2..n]
    D: array[2..n]
    C: set = {2, 3, …, n}
    for i = 2 to n do
        D[i] = G[1, i]
    rof
    repeat
        v = the index of the minimum D[v] not yet selected
        remove v from C // and implicitly add v to S
        for each u ∈ C do
            if D[u] > D[v] + G[v, u] then
                D[u] = D[v] + G[v, u]
            fi
        rof
    until C contains no reachable nodes
    return D
end Dijkstra
```

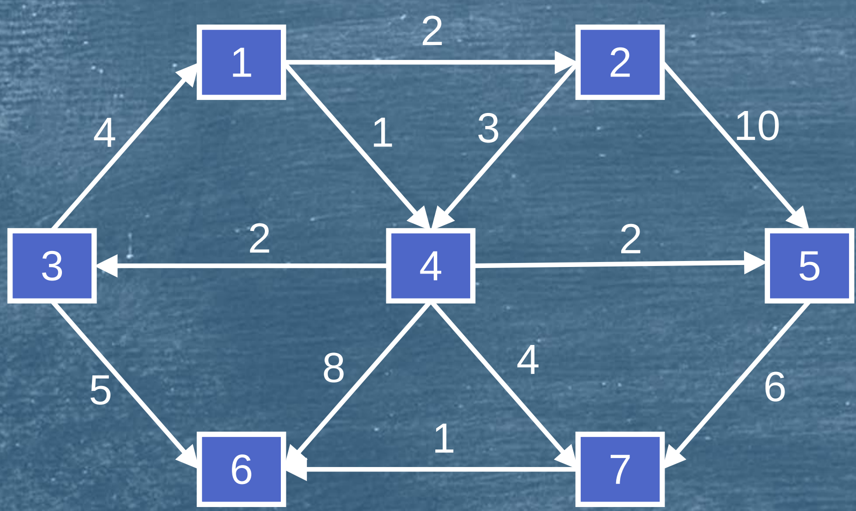
Recording Paths

- Like the basic DFS, Dijkstra's algorithm does not record the shortest path to each vertex, just its total weight.
- Also, like DFS, we can use a parent record, *p*, to keep track of how we reach each vertex.
- This entails a couple of minor changes to the algorithm…

Dijkstra's Algorithm: Path Recording

```
Procedure Dijkstra_Path(G: array[1..n, 1..n]): array [2..n]
    D: array[2..n], P: array[2..n]
    C: set = {2, 3, …, n}
    for i = 2 to n do
        D[i] = G[1, i]
        P[i] = 1
    rof
    repeat
        v = the index of the minimum D[v] not yet selected
        remove v from C // and implicitly add v to S
        for each u ∈ C do
            if D[u] > D[v] + G[v, u] then
                D[u] = D[v] + G[v, u]
                P[u] = v
            fi
        rof
    until C contains no reachable nodes
    return D
end Dijkstra_Path
```

A Larger Example



from

to

0	∞	4	∞	∞	∞	∞
2	0	∞	∞	∞	∞	∞
∞	∞	0	2	∞	∞	∞
1	3	∞	0	∞	∞	∞
∞	10	∞	2	0	∞	∞
∞	∞	5	8	∞	0	1
∞	∞	∞	4	6	∞	0

- At start:
  - G =

0	∞	4	∞	∞	∞	∞
2	0	∞	∞	∞	∞	∞
∞	∞	0	2	∞	∞	∞
1	3	∞	0	∞	∞	∞
∞	10	∞	2	0	∞	∞
∞	∞	5	8	∞	0	1
∞	∞	∞	4	6	∞	0

P =

1
4
1
4
4
4

D =

2
3
1
3
9
5

C = {2, 3, 4, 5, 6, 7}

S = {1}

v = 4

D(v) = 1

- After step 1:
  - G =

0	∞	4	∞	∞	∞	∞
2	0	∞	∞	∞	∞	∞
∞	∞	0	2	∞	∞	∞
1	3	∞	0	∞	∞	∞
∞	10	∞	2	0	∞	∞
∞	∞	5	8	∞	0	1
∞	∞	∞	4	6	∞	0

P =

1
4
1
4
4
4

D =

2
3
1
3
9
5

C = {2, 3, 5, 6, 7}

S = {1, 4}

v = 2

D(v) = 2

- After step 2:
  - G =

0	∞	4	∞	∞	∞	∞
2	0	∞	∞	∞	∞	∞
∞	∞	0	2	∞	∞	∞
1	3	∞	0	∞	∞	∞
∞	10	∞	2	0	∞	∞
∞	∞	5	8	∞	0	1
∞	∞	∞	4	6	∞	0

P =

1
4
1
4
3
4

D =

2
3
1
3
8
5

C = {3, 5, 6, 7}

S = {1, 4, 2}

v = 3

D(v) = 3

- After step 3:
  - G =

0	∞	4	∞	∞	∞	∞
2	0	∞	∞	∞	∞	∞
∞	∞	0	2	∞	∞	∞
1	3	∞	0	∞	∞	∞
∞	10	∞	2	0	∞	∞
∞	∞	5	8	∞	0	1
∞	∞	∞	4	6	∞	0

P =

1
4
1
4
3
4

D =

2
3
1
3
8
5

C = {5, 6, 7}
S = {1, 4, 2, 3}
v = 5
D(v) = 3

- After step 4:
  - G =

0	∞	4	∞	∞	∞	∞
2	0	∞	∞	∞	∞	∞
∞	∞	0	2	∞	∞	∞
1	3	∞	0	∞	∞	∞
∞	10	∞	2	0	∞	∞
∞	∞	5	8	∞	0	1
∞	∞	∞	4	6	∞	0

P =

1
4
1
4
7
4

D =

2
3
1
3
6
5

C = {6, 7}

S = {1, 4, 2, 3, 5}

v = 7

D(v) = 5

- After step 5:
  - G =

0	∞	4	∞	∞	∞	∞
2	0	∞	∞	∞	∞	∞
∞	∞	0	2	∞	∞	∞
1	3	∞	0	∞	∞	∞
∞	10	∞	2	0	∞	∞
∞	∞	5	8	∞	0	1
∞	∞	∞	4	6	∞	0

P =

1
4
1
4
7
4

D =

2
3
1
3
6
5

C = {6}

S = {1, 4, 2, 3, 5, 7}

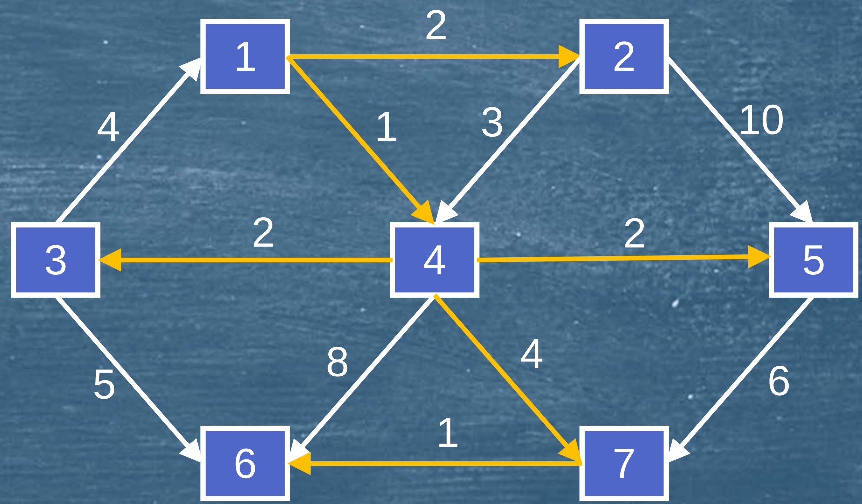
v = 6

D(v) = 6

- After step 6:

  - Paths from vertex 1:
    - To vertex 2
      - Path = (1, 2); W = 2
    - To vertex 3
      - Path = (1, 4, 3); W=3
    - To vertex 4
      - Path = (1, 4); W=1
    - To vertex 5
      - Path = (1, 4, 5); W=3
    - To vertex 6
      - Path = (1, 4, 7, 6); W = 6
    - To vertex 7
      - Path = (1, 4, 7); W = 5

$$P = \begin{bmatrix} 1 \\ 4 \\ 1 \\ 4 \\ 7 \\ 4 \end{bmatrix} \qquad D = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 3 \\ 6 \\ 5 \end{bmatrix}$$



Analysis

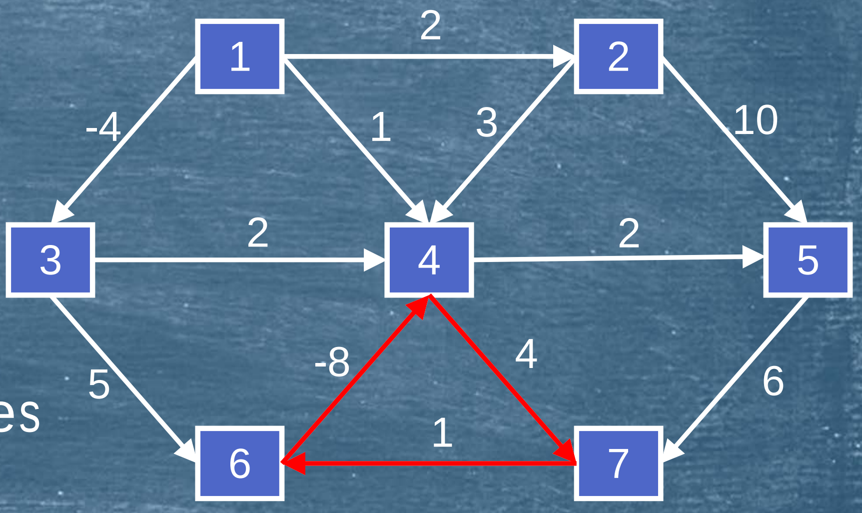
- The complexity of Dijkstra's Algorithm is $\Theta(V \times \log V + E)$
  - How?
- Usually, $E > V$, in fact, in the worst case…
  - …$E \in \Theta(V^2)$—for a complete graph.
- This is about as good as you will get.
- There is one disadvantage:
  - The algorithm only works if all edge weights are non-negative.
- If we have negative edge weights, and especially negative edge cycles, we need a different algorithm.

Bellman-Ford

- Independently invented by both Bellman and Ford.
- Works with graphs that have negative edge weights.
- Identifies negative cycles and vertices with a negative cycle on their path.
- Finds correct path and path length for all other vertices.
- Let us look at an example graph…

A Graph with a negative cycle.

- Consider the graph shown:
- Because the edges between vertices 4, 7 and 6 form a cycle whose total weight is -3, we can reduce the cost of any vertex with a path through any of these vertices as much as we like.



- Note that some vertices, namely 2 and 3, still have well-defined minimum costs of 2 and -4 respectively.
- All other vertices have undefined minimum costs.

The Algorithm

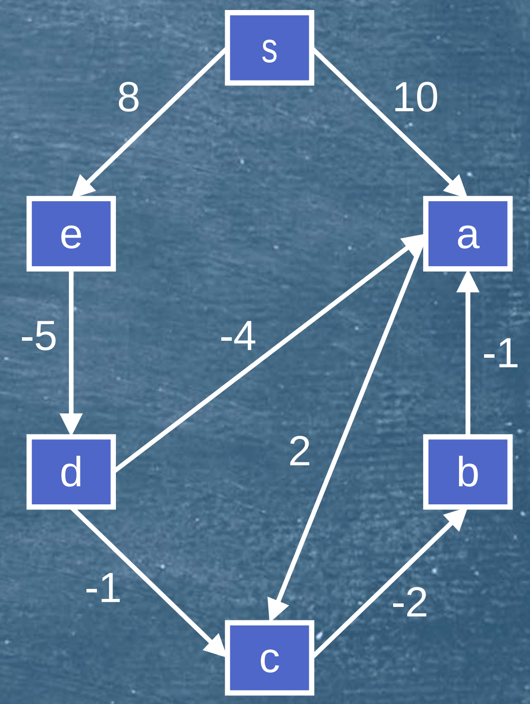
```
Procedure Bellman_Ford(G: graph(V, E), W(): weight, s:vertex):
                                                     distances()
    for all v in V
        D(v) = ∞
        P(v) = nil
    rof
    D(s) = 0
    repeat
        for each edge (u,v) in E
            D(v)=min(D(v), D(u)+W(u,v))
        rof
    |V|-1 times
    for each edge (u,v) in E
        if D(v)>D(u)+W(u,v) then
            D(v)=undefined
        fi
    rof
    return D
end Bellman_ford
```

Different from Dijkstra

- Unlike Dijkstra's algorithm, in which we update only the most promising (next lowest cost) vertex at each iteration, Bellman Ford updates every vertex at each iteration.
- This means that each iteration of Bellman-Ford involves more work than the corresponding iteration of Dijkstra.

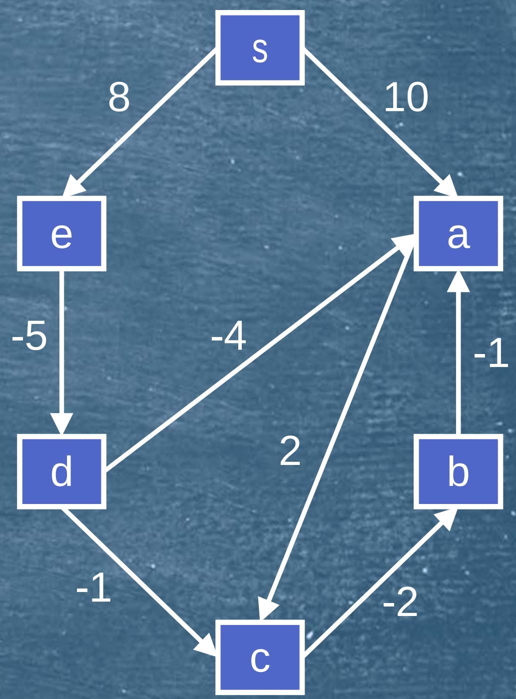
An Example

- Consider the following graph:
- It has 6 vertices so we will run through the main loop 5 times.
- Let’s do it…



Initialization

- We set our initial values for D:
- We then set D(s) to zero.

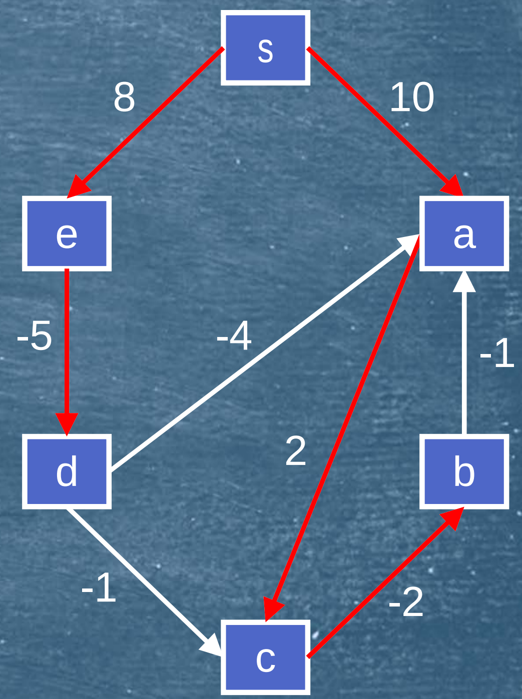


s	a	b	c	d	e
0	∞	∞	∞	∞	∞

Iteration 1

- Vertex s. We can reach a and e. Update.
- Vertex a. We can reach c. Update.
- Vertex b. We can't reach it yet. Skip.
- Vertex c. We can reach b. Update.
- Vertex d. We can't reach it yet. Skip.
- Vertex e. We can reach d. Update.

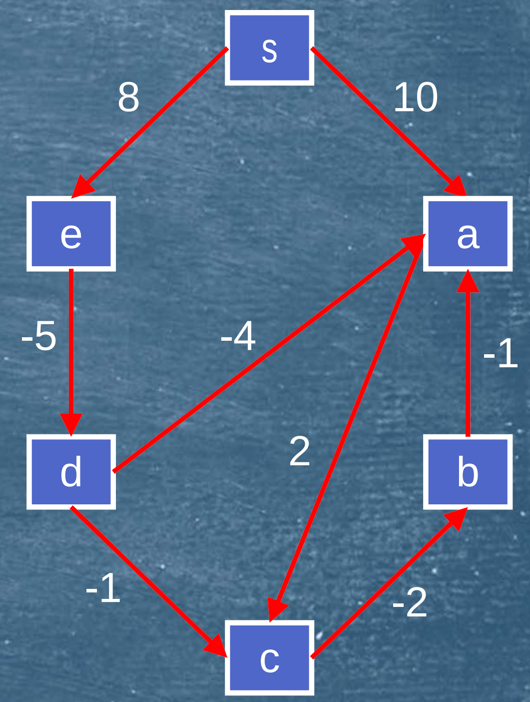
s	a	b	c	d	e
0	10	10	12	3	8



Iteration 2

- Vertex s. We can reach a and e. No change.
- Vertex a. We can reach c. No change
- Vertex b. We can reach a. Update.
- Vertex c. We can reach b. No change.
- Vertex d. We can reach a and c. Update.
- Vertex e. We can reach d. No change.

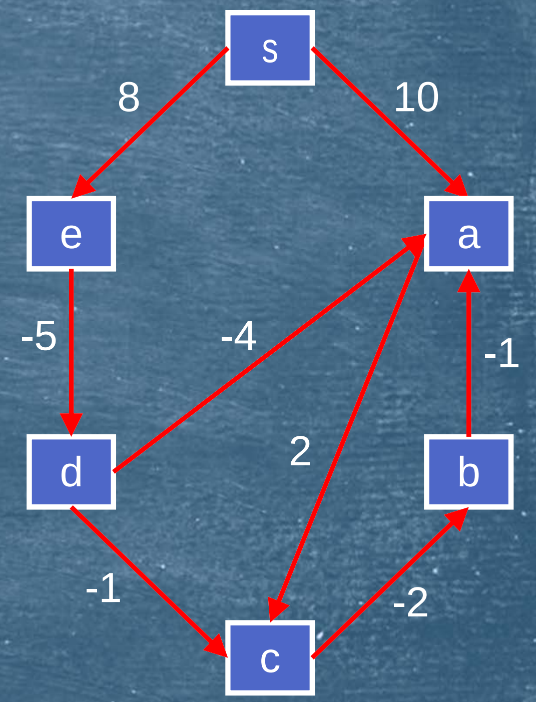
s	a	b	c	d	e
0	-1	10	2	3	8



Iteration 3

- Vertex s. We can reach a and e. No change.
- Vertex a. We can reach c. Update.
- Vertex b. We can reach a. No change.
- Vertex c. We can reach b. Update.
- Vertex d. We can reach a and c. No change.
- Vertex e. We can reach d. No change.

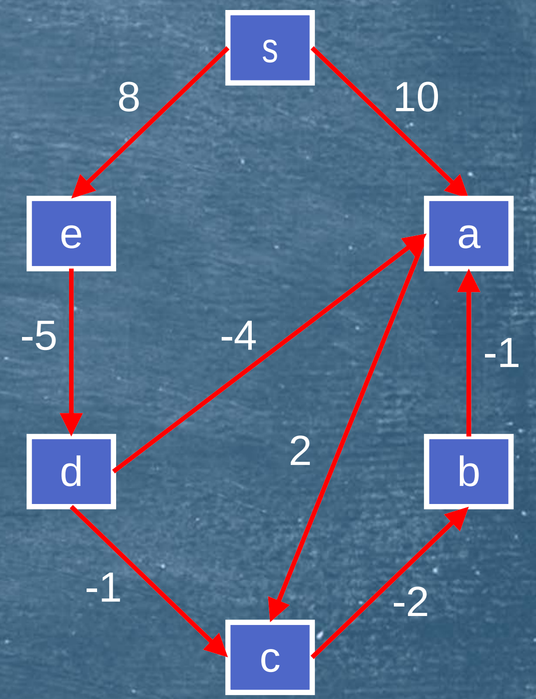
s	a	b	c	d	e
0	-1	-1	1	3	8



Iteration 4

- Vertex s. We can reach a and e. No change.
- Vertex a. We can reach c. No change.
- Vertex b. We can reach a. Update.
- Vertex c. We can reach b. No change.
- Vertex d. We can reach a and c. No change.
- Vertex e. We can reach d. No change.

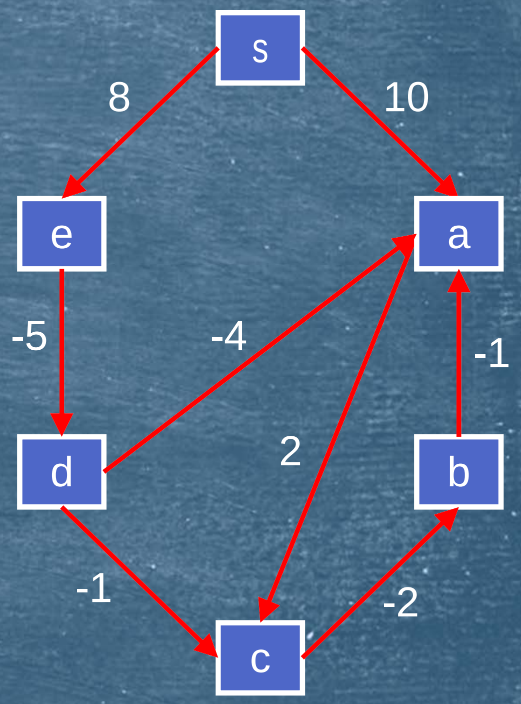
s	a	b	c	d	e
0	-2	-1	1	3	8



Iteration 5

- Vertex s. We can reach a and e. No change.
- Vertex a. We can reach c. Update.
- Vertex b. We can reach a. No change.
- Vertex c. We can reach b. Update.
- Vertex d. We can reach a and c. No change.
- Vertex e. We can reach d. No change.

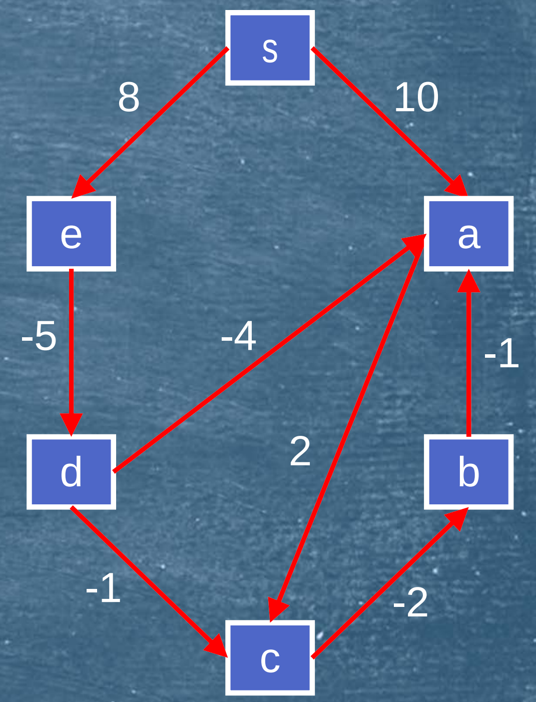
s	a	b	c	d	e
0	-2	-2	0	3	8



Check

- Vertex s. We can reach a and e. No change.
- Vertex a. We can reach c. No change.
- Vertex b. We can reach a. Mark a as bad.
- Vertex c. We can reach b. No change.
- Vertex d. We can reach a and c. No change.
- Vertex e. We can reach d. No change.

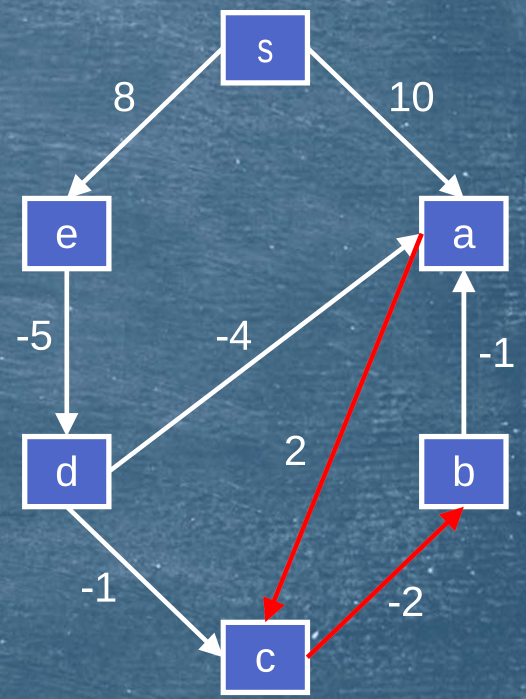
s	a	b	c	d	e
0	X	-2	0	3	8



Finishing Up

- At this point we can conclude that the graph contains a negative cost cycle that involves vertex a.
- If we conduct a DFS (or BFS) starting at vertex a, marking all visited vertices as bad, we get the following result:

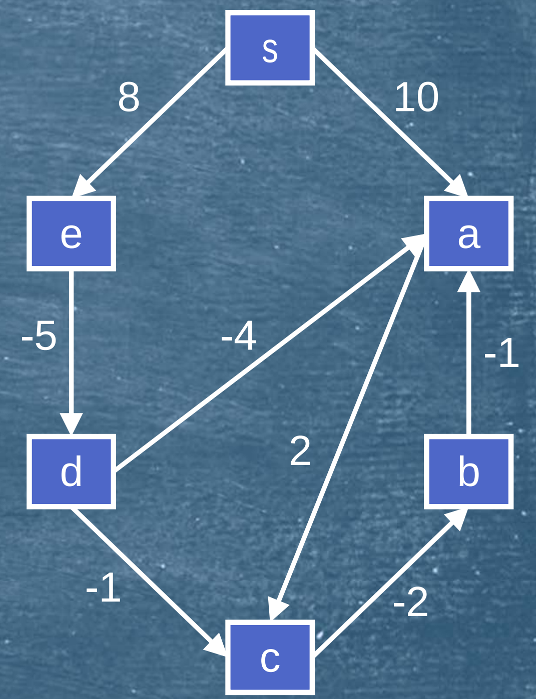
s	a	b	c	d	e
0	X	X	X	3	8



At Last

- Now we have all the information about the graph:
  - Vertex a is reachable via a cycle;
  - Vertex b is reachable via a cycle;
  - Vertex c is reachable via a cycle;
  - Vertex d is reachable at a cost of 3;
  - Vertex e is reachable at a cost of 8.

s	a	b	c	d	e
0	X	X	X	3	8



Analysis

- Bellman-Ford performs the major loop $|V\text{-}1|$ times.
- Inside this loop it checks every edge; $|E|$ operations.
- Finally, it does another $|E|$ checks for potential cycles.
- Overall, Bellman-Ford has $\Theta(|V| \times |E|)$ complexity.