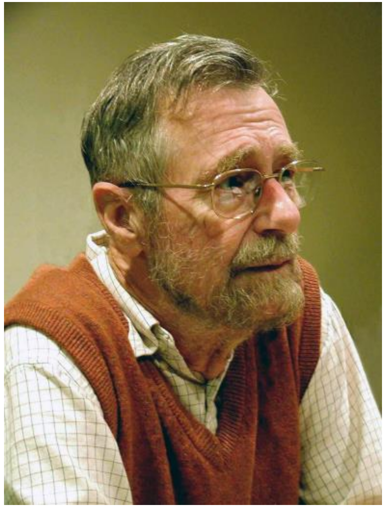
Shortest Paths: Dijkstra’s Algorithm

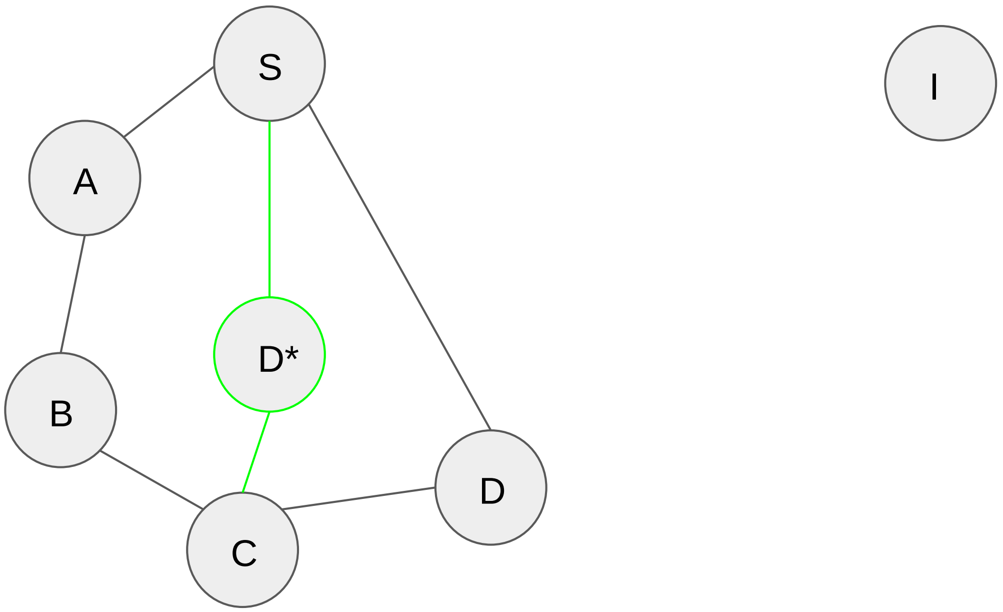


SSSP: Context

- Finding the shortest path from a source to goal, or source to every node
- Google/Apple maps

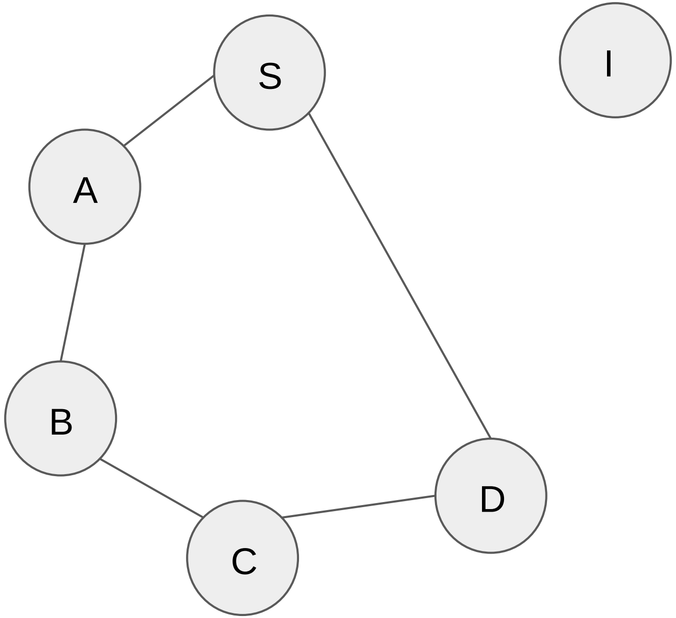
Recall: Breadth-First Search

- Finds **a** path of fewest edges from a source to every reachable node in the graph



Recall: Breadth-First Search

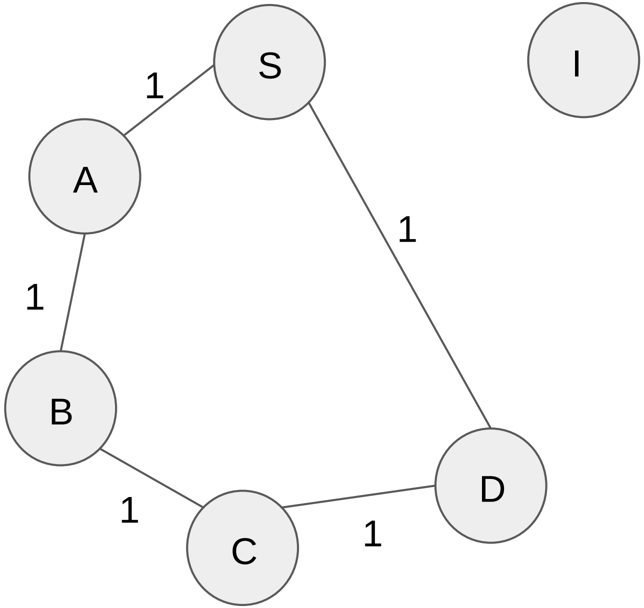
- Finds **a** path of fewest edges from a source to every reachable node in the graph



Destination	Path	Path length
S	[S]	0
A	[S, A]	1
B	[S, A, B]	2
C	[S, D, C]	2
D	[S, D]	1
I	none	∞

Introduce: Weighted Graphs

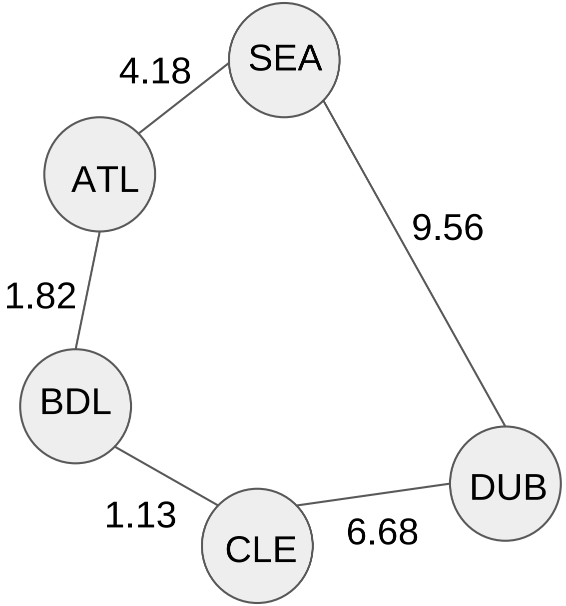
- Each edge has a **weight**, or **cost**, associated with it



Destination	Path	Path **cost**
S	[S]	0
A	[S, A]	1
B	[S, A, B]	2
C	[S, D, C]	2
D	[S, D]	1
I	none	∞

Introduce: Weighted Graphs

- Edge weight could represent a time cost, financial cost, or a relation between two nodes

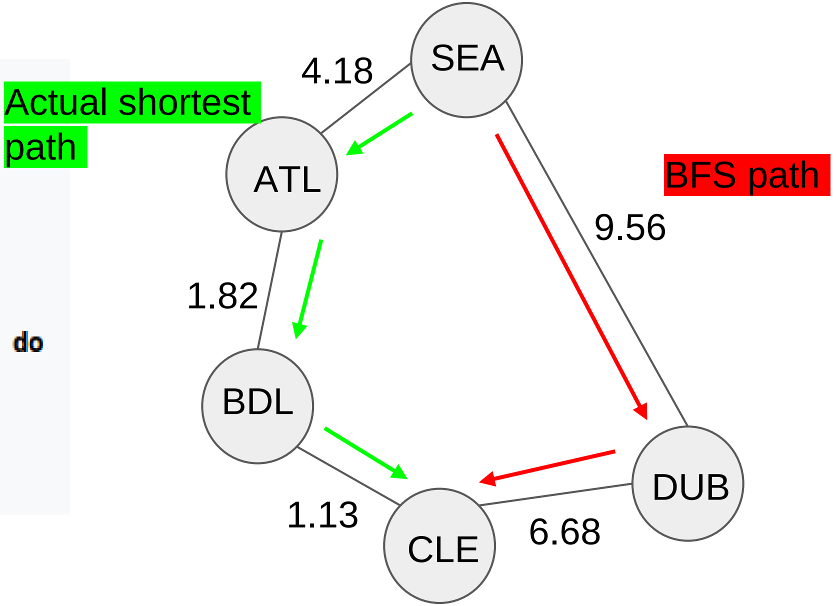


Destination	Path	Path **cost**
SEA	[S]	0
ATL	[S, A]	4.18
BDL	[S, A, B]	6.00
CLE	[S, A, B C]	7.13
DUB	[S, D]	9.56
I	none	∞

Problem: Breadth-First Search on Weighted Graphs

- BFS is not designed to consider edge weights!

```
procedure BFS(G,start_v):
    let S be a queue
    S.enqueue(start_v)
    while S is not empty
        v = S.dequeue()
        if v is the goal:
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered:
                label w as discovered
                w.parent = v
                S.enqueue(w)
```



Solution: Dijkstra’s Algorithm

- Explore nodes in a **greedy, shortest-path-first** manner
- Key feature: explore (dequeue/pop) the closest node not yet explored.

Pseudocode: With Inefficiencies

- Overview of pseudocode

```
Dijkstra(Graph, source):

    create vertex list Q

    for each vertex v in Graph:
        v.dist = INFINITY   //dist represents distance from source
        v.parent = null
        add v to Q
    source.dist = 0

    while Q is not empty:
        u = vertex in Q with smallest "dist"

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt = u.dist + weight(u, v)
            if alt < v.dist:
                v.dist = alt
                v.parent = u

    // return depends on the application
```

Pseudocode: Addressing Inefficiencies

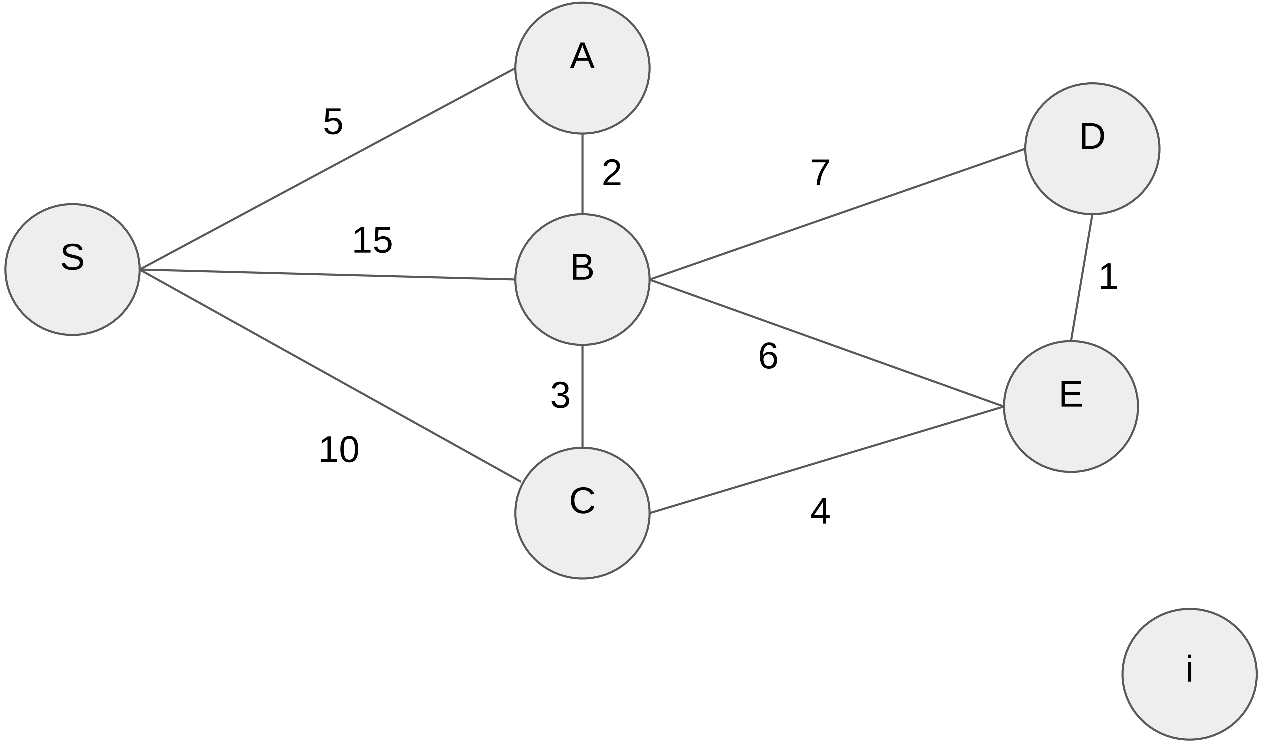
- Use your Data Structures…

```
function Dijkstra(Graph, source):

    create vertex set Q

    source.dist = 0
    for each vertex v in Graph:
        v.dist = INFINITY
        v.prev = null
        Q.addWithPriority(v, v.dist)

    while Q is not empty:

        u = Q.extractMin() // O(log(n))


        for each neighbor v of u:           // only v that are still in Q
            alt = u.dist + weight(u, v)
            if alt < v.dist:
                v.dist = alt
                v.parent = u
                Q.decreasePriority(v, v.dist) //O(log(n)) if we can find it quickly

    // return depends on the application
```

Recall: PriorityQueue (Min-Heap Implementation)

- add(Object key, int priority)^ - O(log(n))
- extractMin() - O(log(n))
- decreaseKey(Object key, int newPriority)^ - O(log(n))*

GYHD: Example



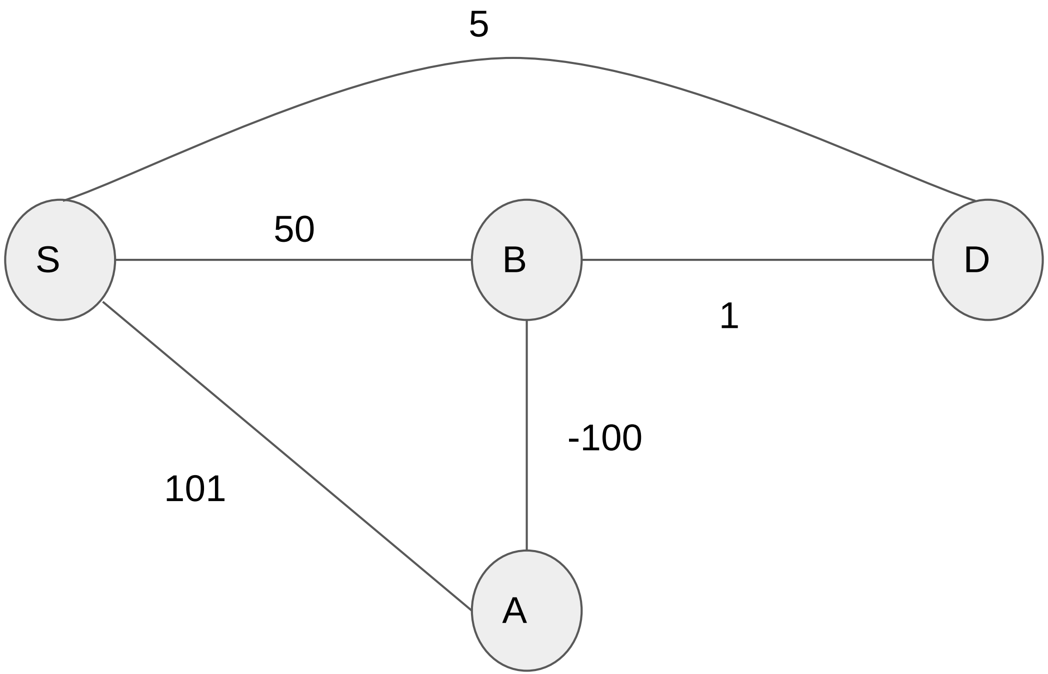
Assuming S is the source

Destination	Cost	Parent
S	0	null
A		
B		
C		
D		
E		
i		

Conditions for Dijkstra’s

- No negative cycles; a negative cycle will **always** be a problem
- In fact, no negative edges; a negative edge will **sometimes** be a problem
- GYHD: come up with a graph that has no negative cycles, has a negative edge, where Dijkstra’s algorithm would not find the shortest path from a source to a destination

Counterexample



If Time: Java implementation

- `compareTo(Node other)` vs. `compare(Node a, Node b)`
- Cats
- Time complexity and ease of implementation

Finally

- Thank you!
- PLEASE take survey on teaching feedback - will send to you shortly