Dijkstra's Algorithm

Sadatul Islam Sadi Irtiaz Kabir Wahid Al Azad Navid

Bangladesh University of Engineering and Technology

February 20, 2024

Problem Statement

First Route:

Problem Statement

First Route:

 $\longrightarrow$ 

Problem Statement

First Route:

 $\longrightarrow$  $\longrightarrow$ 

Problem Statement

First Route:

 $\longrightarrow$  $\longrightarrow$ 

$\longrightarrow$ 

Problem Statement

First Route:

Second Route:

Problem Statement

First Route:

 ⟶  ⟶ 

⟶ 

Second Route:

 ⟶ 

Problem Statement

First Route:

 →  → 

→ 

Second Route:

 →  → 

Problem Statement

First Route:

 ⟶  ⟶ 

⟶ 

Second Route:

 ⟶  ⟶  ⟶ 

Problem Statement

First Route:

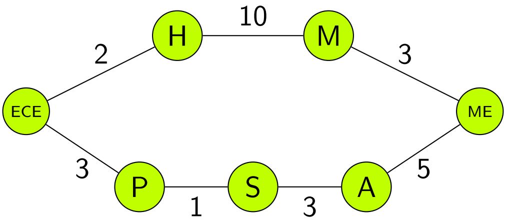
 ⟶  ⟶ 

⟶ 

Second Route:

 ⟶  ⟶  ⟶  ⟶ 

Problem Statement



A Graph Representation of the previous problem

Dijkstra's Algorithm

Definition

Dijkstra's algorithm is an greedy algorithm for finding the shortest paths between nodes in a weighted graph.

Graph

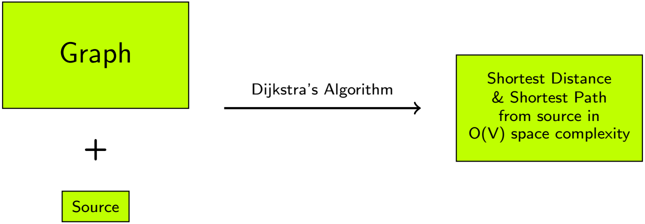
+

Source

Dijkstra's Algorithm

Definition

Dijkstra's algorithm is an greedy algorithm for finding the shortest paths between nodes in a weighted graph.



History

O(V^2) Algorithm

What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. [1]

History

O(V^2) Algorithm

What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. [1]

O(E + VlogV) Algorithm

In 1984, Fredman & Tarjan proposed use of fibonacci heap to optimize the running time of the algorithm to O($\|E\| + \|V\| logV$)[2]

Pseudocode

```
def Dijkstra(Graph, source):

    Q = PrioriyQueue()
    dist ← ϕ
    prev ← ϕ

    for each vertex v in Graph.Vertices:
        dist[v] ← ∞ if v is not source else 0
        prev[v] ← None
        Q.enqueue(v, dist[v])

    while Q is not empty:
        u ← Q.extract_min()

        for each neighbor v of u:
            alt ← dist[u] + Graph.Edges(u, v)

            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)

    return dist, prev
```

Pseudocode

```
def Dijkstra(Graph, source):

Q = PrioriyQueue()
dist ← ϕ
prev ← ϕ

for each vertex v in Graph.Vertices:
    dist[v] ← ∞ if v is not source else 0
    prev[v] ← None
    Q.enqueue(v, dist[v])

while Q is not empty:
    u ← Q.extract_min()

    for each neighbor v of u:
        alt ← dist[u] + Graph.Edges(u, v)

        if alt < dist[v]:
            dist[v] ← alt
            prev[v] ← u
            Q.decrease_priority(v, alt)

return dist, prev
```

Pseudocode

```
def Dijkstra(Graph, source):

    Q = PrioriyQueue()
    dist ← ϕ
    prev ← ϕ

    for each vertex v in Graph.Vertices:
        dist[v] ← ∞ if v is not source else 0
        prev[v] ← None
        Q.enqueue(v, dist[v])

    while Q is not empty:
        u ← Q.extract_min()

        for each neighbor v of u:
            alt ← dist[u] + Graph.Edges(u, v)

            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)

    return dist, prev
```

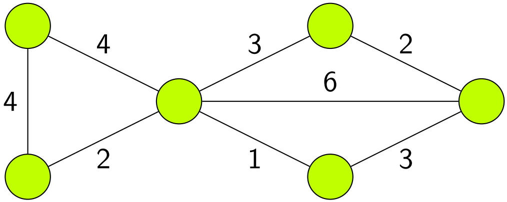
Pseudocode

```
def Dijkstra(Graph, source):

    Q = PrioriyQueue()
    dist ← ϕ
    prev ← ϕ

    for each vertex v in Graph.Vertices:
        dist[v] ← ∞ if v is not source else 0
        prev[v] ← None
        Q.enqueue(v, dist[v])

    while Q is not empty:
        u ← Q.extract_min()

        for each neighbor v of u:
            alt ← dist[u] + Graph.Edges(u, v)

            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)

    return dist, prev
```

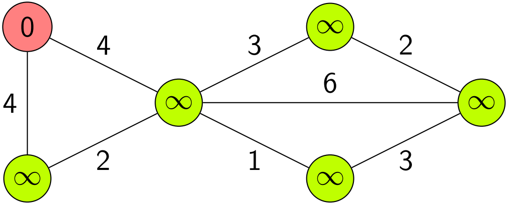
Pseudocode

```
def Dijkstra(Graph, source):

Q = PrioriyQueue()
dist ← ϕ
prev ← ϕ

for each vertex v in Graph.Vertices:
    dist[v] ← ∞ if v is not source else 0
    prev[v] ← None
    Q.enqueue(v, dist[v])

while Q is not empty:
    u ← Q.extract_min()

    for each neighbor v of u:
        alt ← dist[u] + Graph.Edges(u, v)

        if alt < dist[v]:
            dist[v] ← alt
            prev[v] ← u
            Q.decrease_priority(v, alt)

return dist, prev
```

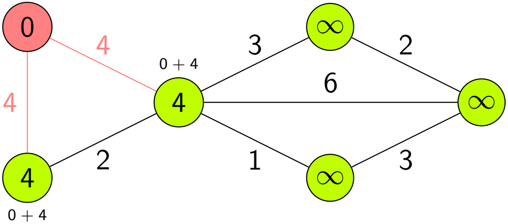
Simulation



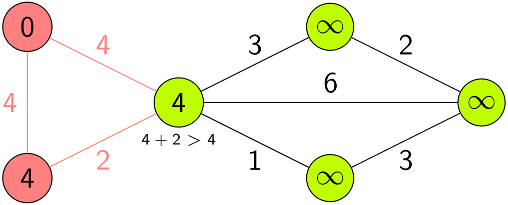
Simulation



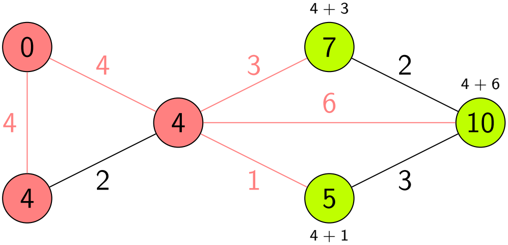
Simulation



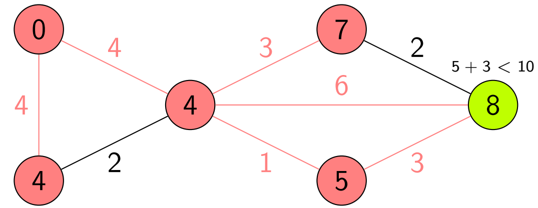
Simulation



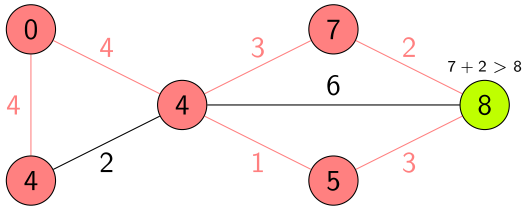
Simulation



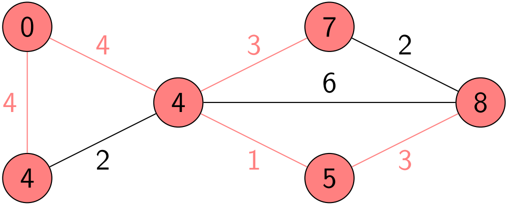
Simulation



Simulation



Simulation



Running time - revisiting the pseudocode

```
def Dijkstra(Graph, source):

    Q = PrioriyQueue()
    dist ← ϕ
    prev ← ϕ

    for each vertex v in Graph.Vertices:
        dist[v] ← ∞ if v is not source else 0
        prev[v] ← None
        Q.enqueue(v, dist[v])

    while Q is not empty:
        u ← Q.extract_min()

        for each neighbor v of u:
            alt ← dist[u] + Graph.Edges(u, v)

            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)

    return dist, prev
```

Running time

Can you identify the costly operations performed here?

Running time

```
def Dijkstra(Graph, source):

    Q = PrioriyQueue()
    dist ← ϕ
    prev ← ϕ

    for each vertex v in Graph.Vertices:
        dist[v] ← ∞ if v is not source else 0
        prev[v] ← None
        Q.enqueue(v, dist[v])

    while Q is not empty:
        u ← Q.extract_min()

        for each neighbor v of u:
            alt ← dist[u] + Graph.Edges(u, v)

            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)

    return dist, prev
```

Running time

Number of calls

Extract min (T_{em}) $\Theta(|V|)$

Decrease priority (T_{dp}) $\Theta(|E|)$

Running time

Number of calls

Extract min (T_{em}) $\Theta(|V|)$

Decrease priority (T_{dp}) $\Theta(|E|)$

Total running time

$$T = \Theta(|E|.T_{dp} + |V|.T_{em})$$

Running time

$$T = \Theta(|E|.T_{dp} + |V|.T_{em})$$

Implementation	$\boldsymbol{T_{dp}}$	$\boldsymbol{T_{em}}$	$\boldsymbol{T}$
Singly Linked List	$\Theta(1)$	$\Theta(\lvert V\rvert)$	$\Theta(V^2)$

Running time

$$T = \Theta(|E|.T_{dp} + |V|.T_{em})$$

Implementation	$\boldsymbol{T_{dp}}$	$\boldsymbol{T_{em}}$	$\boldsymbol{T}$
Singly Linked List	$\Theta(1)$	$\Theta(\|V\|)$	$\Theta(V^2)$
Binary Heap	$\Theta(log\|V\|)$	$\Theta(log\|V\|)$	$\Theta\left((E+V)log\|V\|\right)$

Running time

$$T = \Theta(|E|.T_{dp} + |V|.T_{em})$$

Implementation	$\boldsymbol{T_{dp}}$	$\boldsymbol{T_{em}}$	$\boldsymbol{T}$
Singly Linked List	$\Theta(1)$	$\Theta(\lvert V \rvert)$	$\Theta(V^2)$
Binary Heap	$\Theta(log\lvert V \rvert)$	$\Theta(log\lvert V \rvert)$	$\Theta\left((E+V)log\lvert V \rvert\right)$
Fibonacci Heap	$\Theta(1)$	$\Theta(log\lvert V \rvert)$	$\Theta(E+Vlog\lvert V \rvert)$

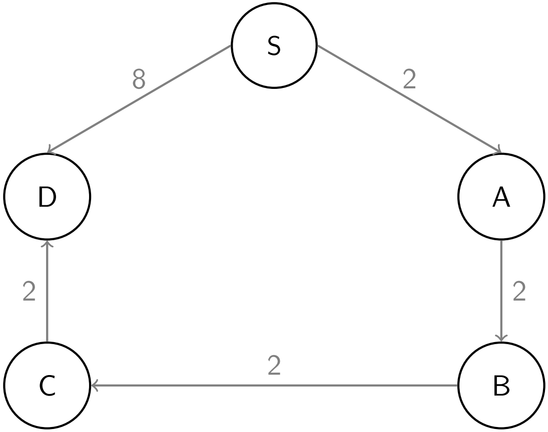
Drawbacks

Traversing unnecessarily

When we are eager to know the distance from the source to a particular node, Dijkstra's algorithm may result in a longer run-time.

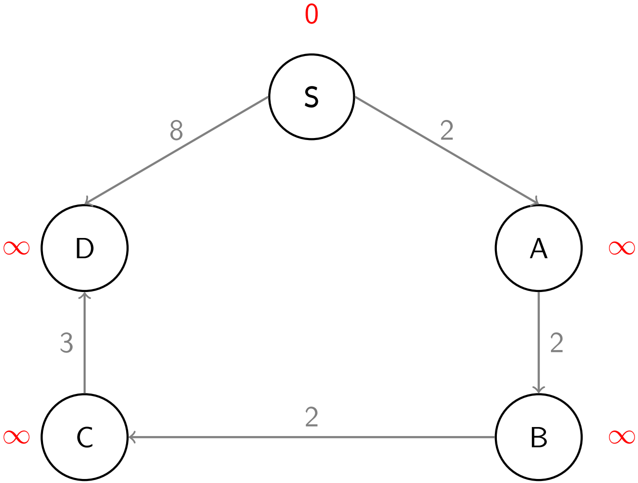
Traversing unnecessarily

- Find the shortest distance from S to D



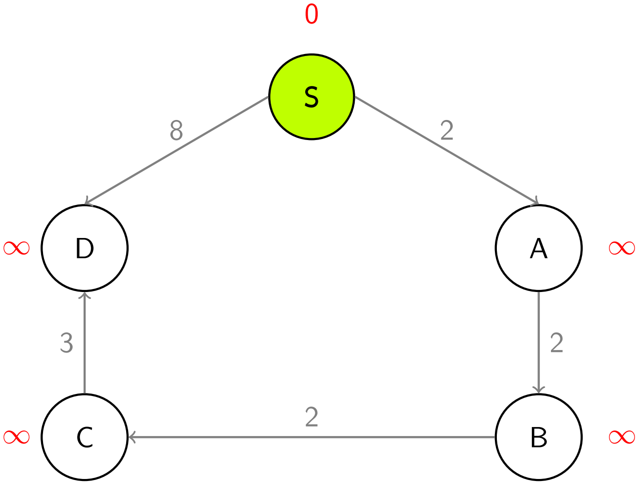
Traversing unnecessarily

- Find the shortest distance from S to D



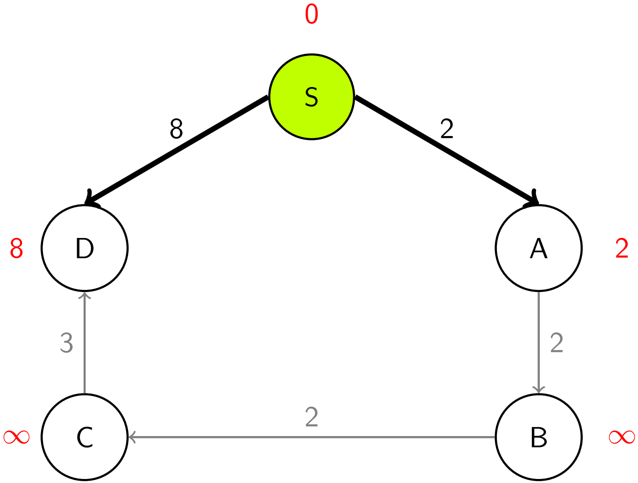
Traversing unnecessarily

- Find the shortest distance from S to D



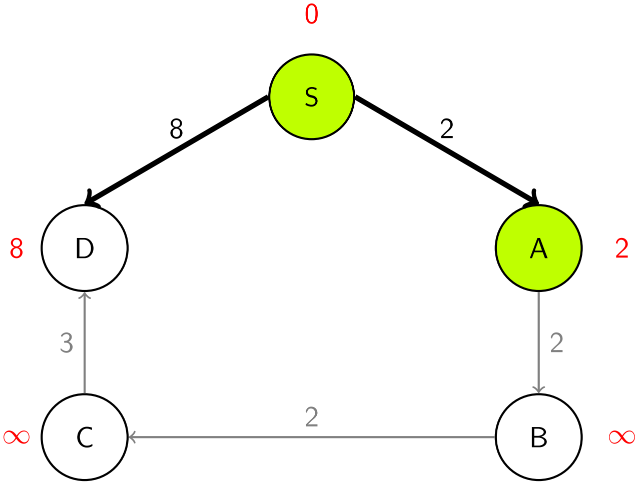
Traversing unnecessarily

- Find the shortest distance from S to D



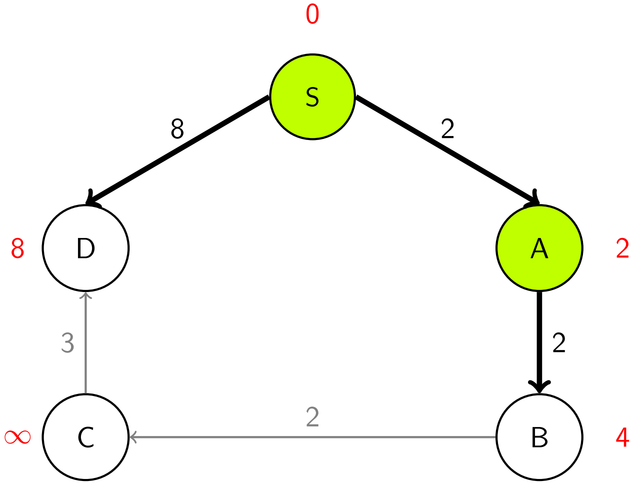
Traversing unnecessarily

- Find the shortest distance from S to D



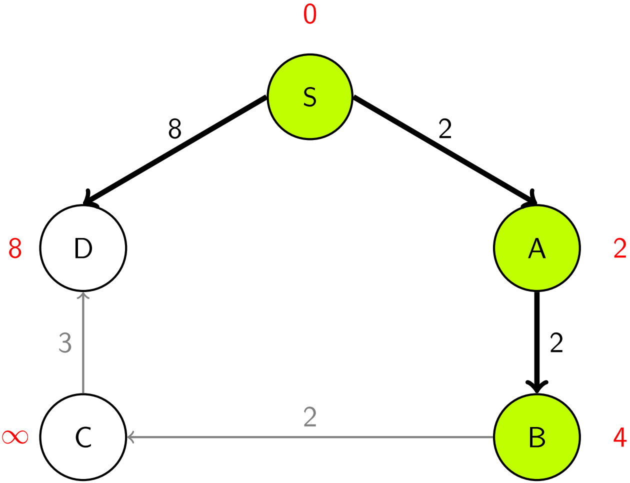
Traversing unnecessarily

- Find the shortest distance from S to D



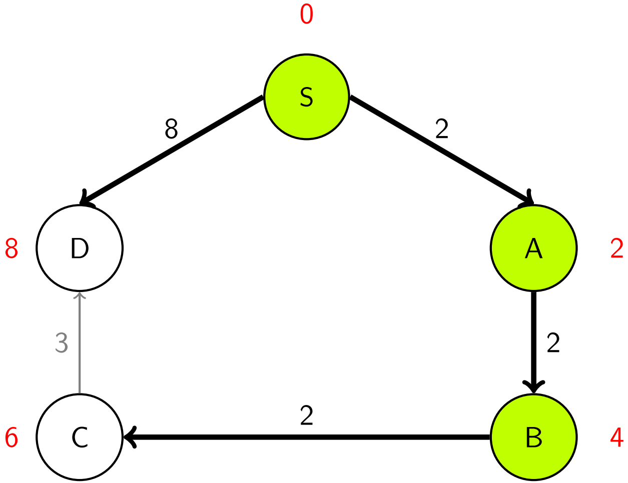
Traversing unnecessarily

- Find the shortest distance from S to D



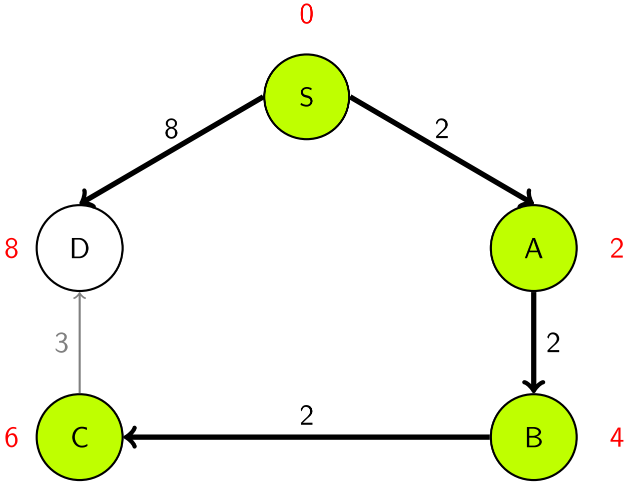
Traversing unnecessarily

- Find the shortest distance from S to D



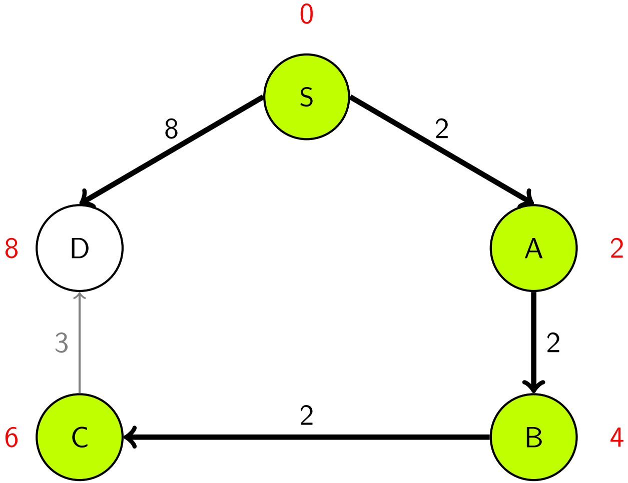
Traversing unnecessarily

- Find the shortest distance from S to D



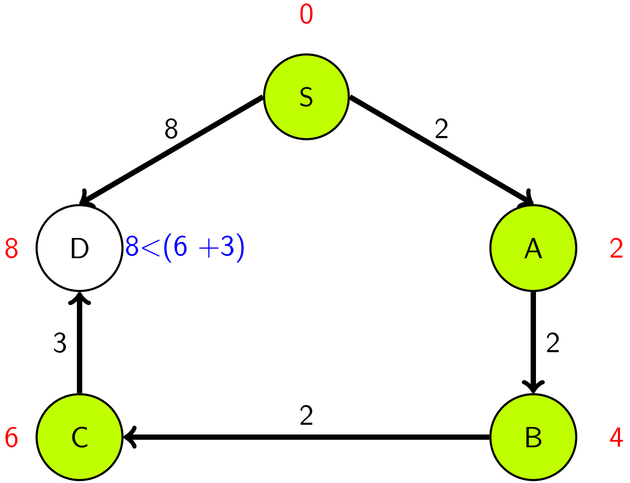
Traversing unnecessarily

- Find the shortest distance from S to D



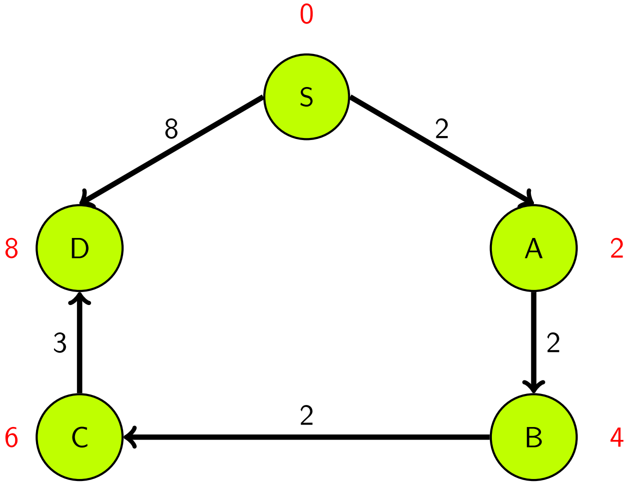
Traversing unnecessarily

- Find the shortest distance from S to D



Traversing unnecessarily

- Find the shortest distance from S to D



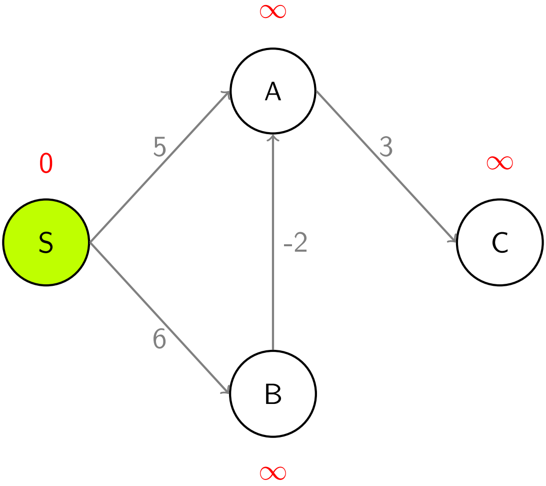
Drawbacks

Fails in case of negative weight edge

If a graph contains negative weight edge(s), the algorithm may often produce wrong results.

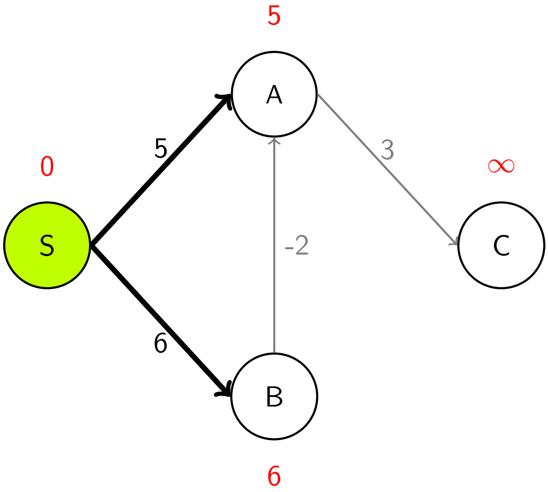
Negative Weight Issue

- Find the shortest distance from S to C



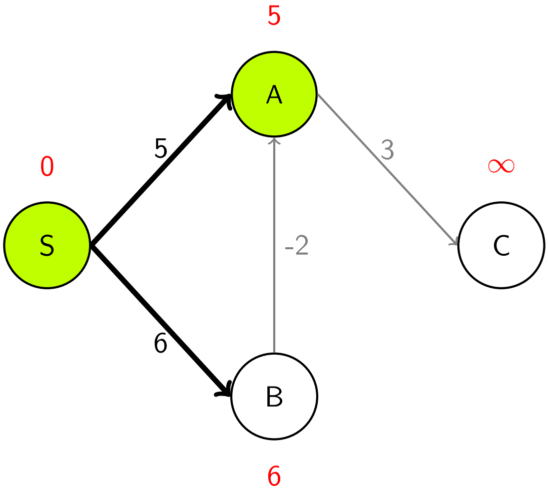
Negative Weight Issue

- Find the shortest distance from S to C



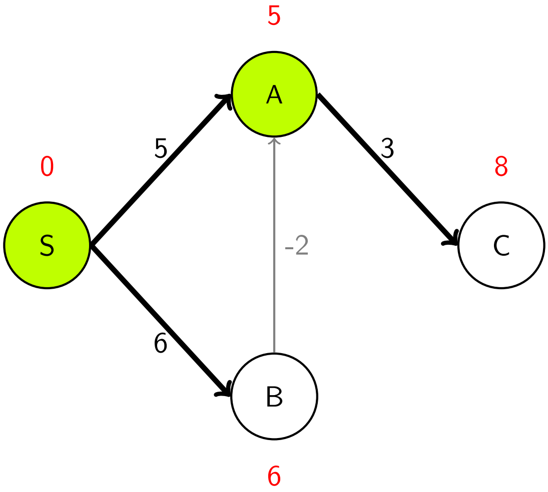
Negative Weight Issue

- Find the shortest distance from S to C



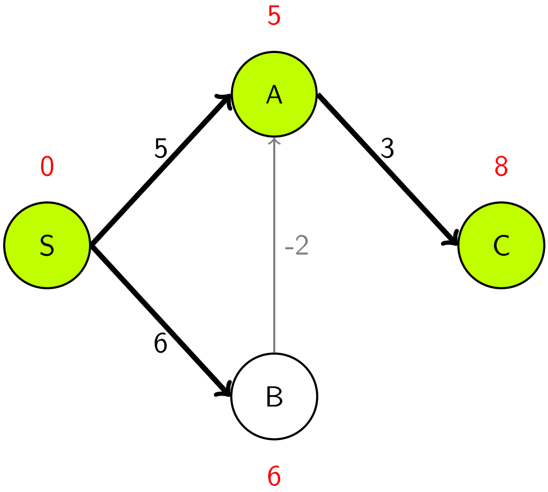
Negative Weight Issue

- Find the shortest distance from S to C



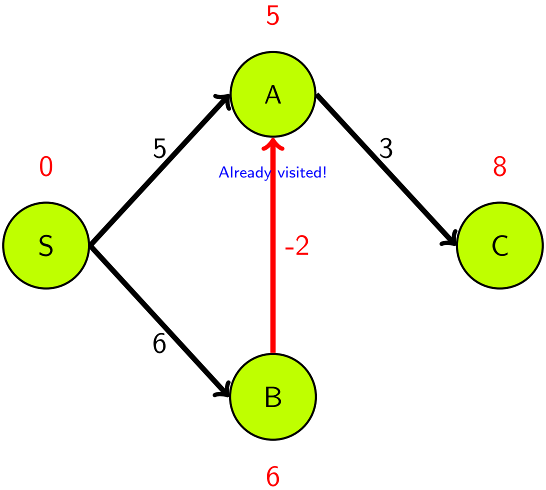
Negative Weight Issue

- Find the shortest distance from S to C

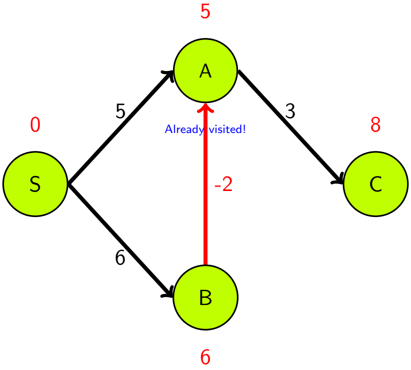


Negative Weight Issue

- Find the shortest distance from S to C

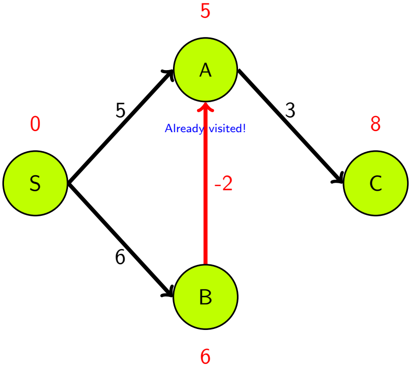


Negative Weight Issue



- Thus, the algorithm gives us the shortest distance of 8 by taking the path $S \to A \to C$

Negative Weight Issue



- However, taking the path $S \rightarrow B \rightarrow A$ would result in a shorter distance of $6 - 2 + 3 = 7$

Specialized variants

- When arc weights are small integers (bounded by a parameter C), specialized queues which take advantage of this fact can be used to speed up Dijkstra's algorithm.

Name	Data Structure	Running Time
Dial's algorithm	Bucket queue	$O(\|E\| + \|V\| * C)$

Specialized variants

- When arc weights are small integers (bounded by a parameter C), specialized queues which take advantage of this fact can be used to speed up Dijkstra's algorithm.

Name	Data Structure	Running Time
Dial's algorithm	Bucket queue	$O(\|E\| + \|V\| * C)$
Van Embde Boas Tree	Priority Queue	$O(\|E\| loglogC)$

Specialized variants

- When arc weights are small integers (bounded by a parameter C), specialized queues which take advantage of this fact can be used to speed up Dijkstra's algorithm.

Name	Data Structure	Running Time
Dial's algorithm	Bucket queue	$O(\|E\| + \|V\| * C)$
Van Embde Boas Tree	Priority Queue	$O(\|E\| loglogC)$
Ahuja *et al.*	Fibonacci Heap	$O(\|E\| + \|V\|\sqrt{logC})$

Frame Title

- **Routing Protocols in Computer Networks:**
  Dijkstra's algorithm is often used in routing protocols such as OSPF (Open Shortest Path First) and IS-IS (Intermediate System to Intermediate System)

Frame Title

- **GPS Navigation Systems:**
  In GPS navigation, Dijkstra's algorithm is employed to find the shortest route between two locations, considering factors like distance, traffic conditions, and road types.

Frame Title

- **Game Development:**
  In video games, especially those involving maps and navigation, Dijkstra's algorithm can be employed to create intelligent and efficient path-finding for characters or objects.

Frame Title

- **Resource Management in Operating Systems:**
  Dijkstra's algorithm is applied in resource management to allocate resources optimally.

Thank You

References

Philip L. Frana and Thomas J. Misa.
An interview with edsger w. dijkstra.
Commun. ACM, 53(8):41–47, aug 2010.

M.L. Fredman and R.E. Tarjan.
Fibonacci heaps and their uses in improved network optimization algorithms.
In *25th Annual Symposium onFoundations of Computer Science, 1984.*, pages 338–346, 1984.