# DSC40B: Theoretical Foundations of Data Science II

## Lecture 15: *Shortest Path in Weighted Graphs – part II*

Instructor: Yusu Wang

# Prelude

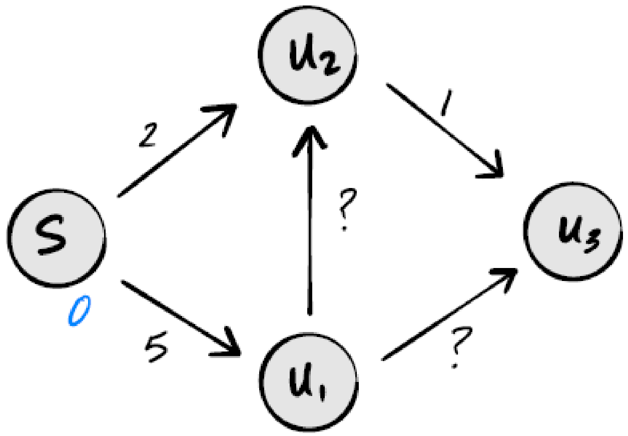
- Previously
  - SSSP in weighted graphs
  - Properties of shortest paths in weighted graphs
  - Edge update
  - Bellman-Ford algorithm to solve SSSP for any weighted graphs
- Today: Dijkstra algorithm
  - A much more efficient algorithm for SSSP for positively weighted graphs

# Dijkstra shortest path algorithm

# Dijkstra Algorithm

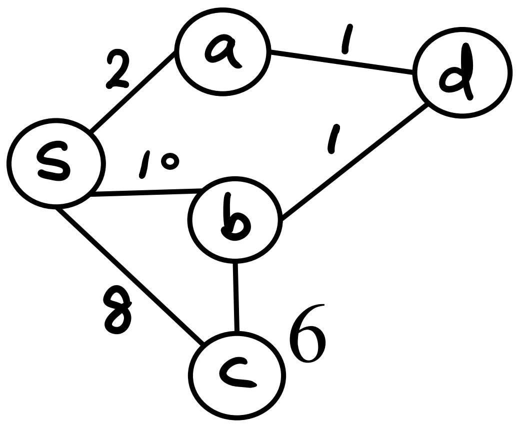
- Dijkstra has some similarity to Bellman-Ford
  - In the sense that both will repeatedly perform update(edge) operations to improve shortest path estimates
    - different in the order of these update operations, where Dijkstra does so more intelligently for positively weighted graphs to reduce redundancy.

- In particular,
  - Bellman-Ford updates all edges in each iteration – many of them don't need to be updated
  - If we assume all edge weights are positive, then we can rule out some paths immediately:



# Dijkstra Algorithm

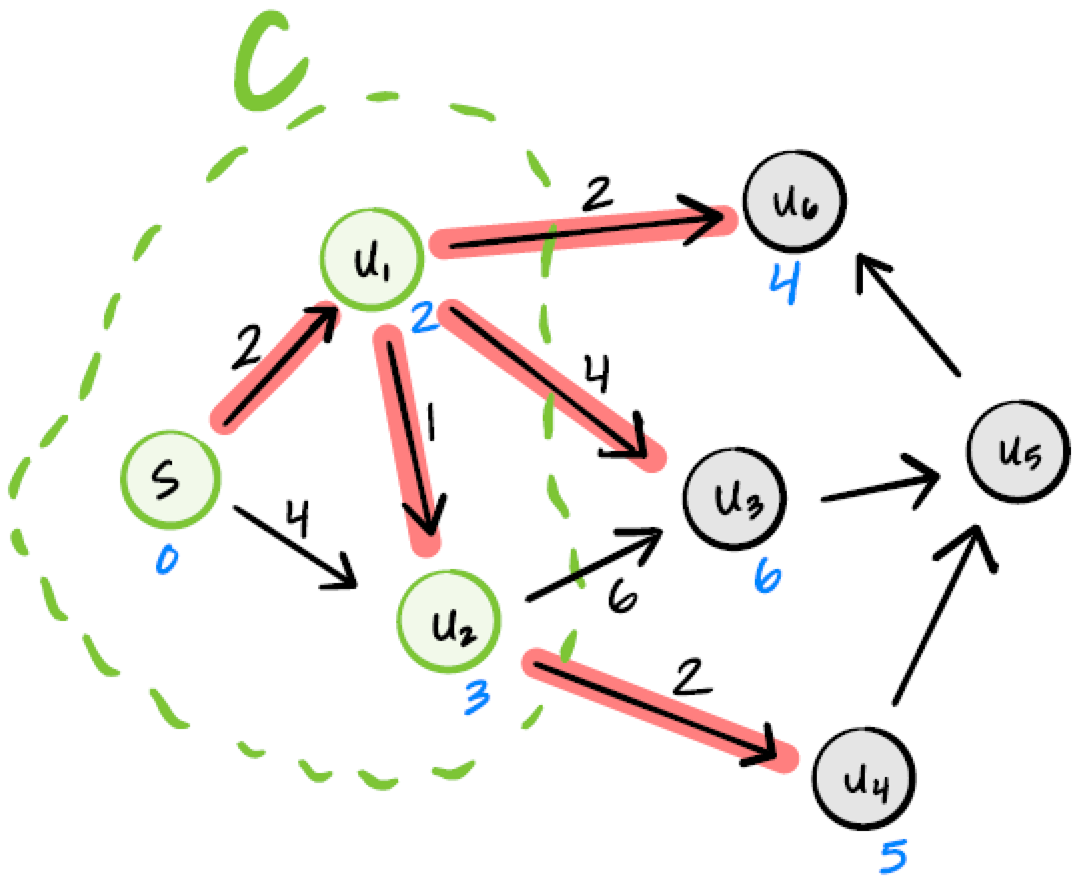
- High level idea also similar to BFS
  - for each node, we will maintain an estimate of shortest distance to the source
  - this estimate will be iteratively updated
  - the algorithm will explore the nodes in a greedy manner, in increasing distance to the source
    - by the time we start to explore a node, the algorithm will be guaranteed to have already computed correct shortest path distance from the source to this node



# Estimated shortest path

- Fix the source node to be $s$
- Similar to BFS, Dijkstra algorithm keeps track of the estimated shortest paths found so far, together with $u$.est (estimated distance from $s$ to $u$ )

- At the beginning, $u$.est = $\infty$ for all nodes other than the source $s$
- Keep track of a set $C$ of correct nodes
- At every step, add node outside of $C$ with smallest estimated distance to $C$; update estimated distances to its neighbors.



C
S
0
u1
2
u2
3
u3
6
u4
5
u5
u6
4
2
4
1
4
2
6
2

# Outline of Dijkstra Alg (not code)

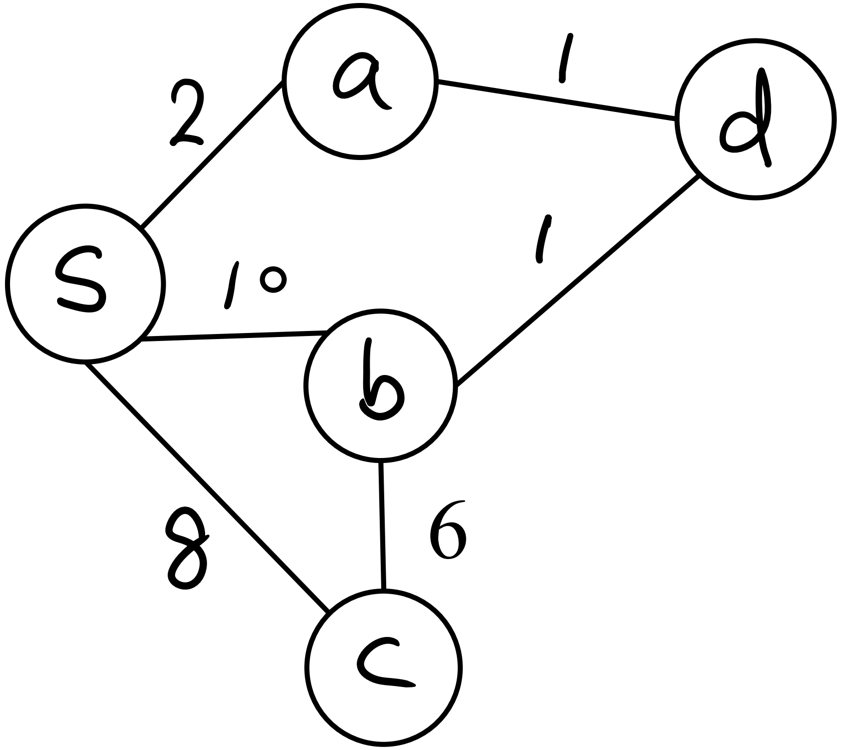
```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
        C.add(u)
        for v in graph.neighbors(u):
            update(u, v, weights, est, pred)

    return est, pred
```

# Example



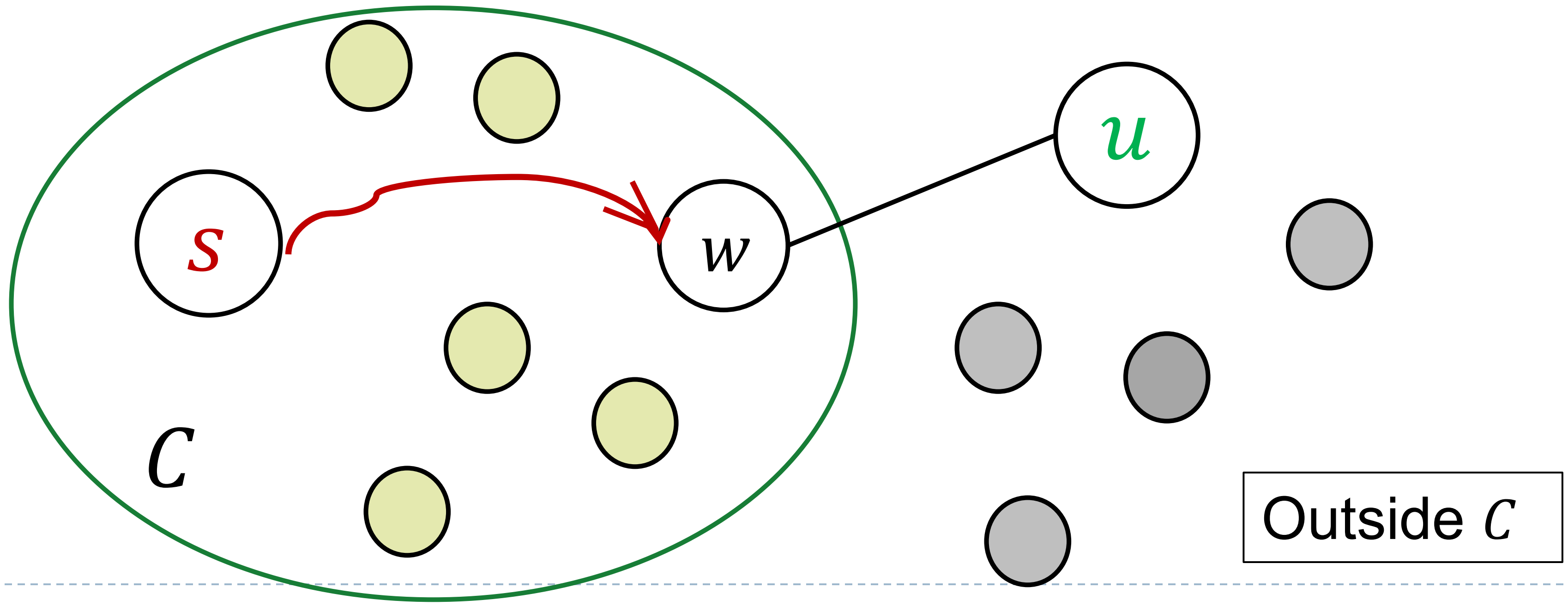

- Outside

- Set C

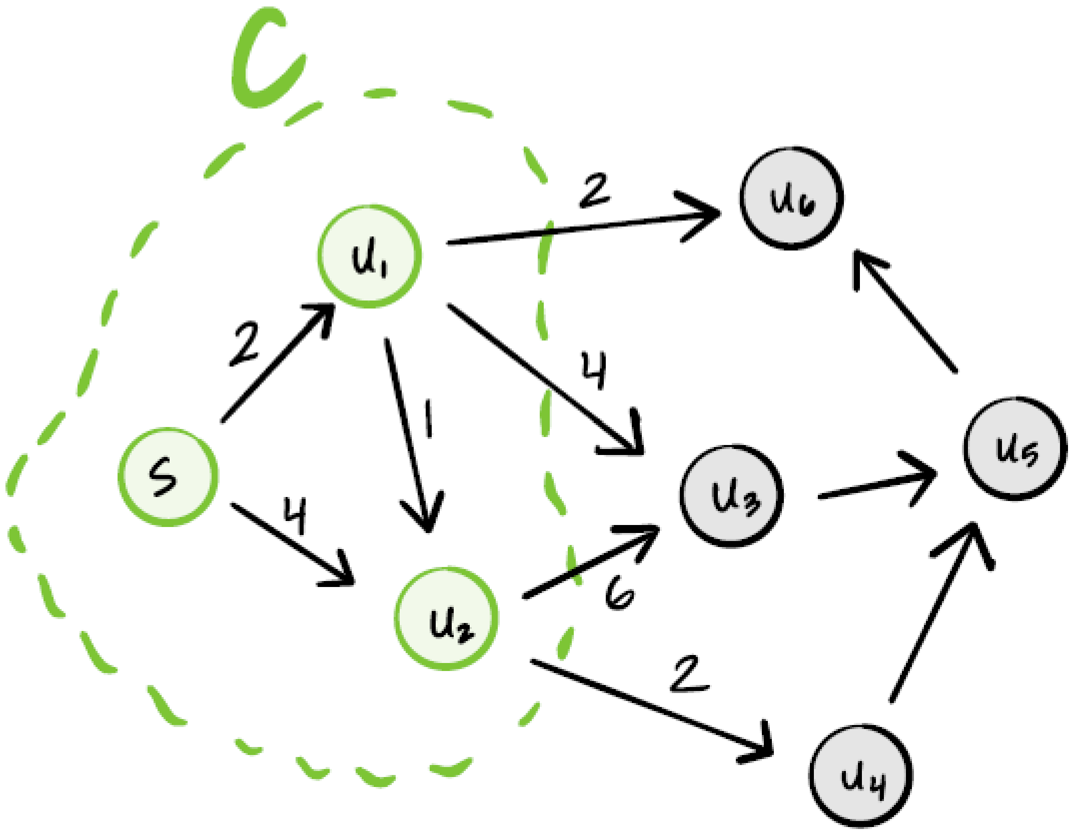
# Correctness of Dijkstra

# Exit paths

- An exit path through $C$ is a path $\pi$: $s \rightsquigarrow u$ from the source $s$ to some node $u \notin C$, called exit node, such that $\pi$ consists of
  - first a path in $C$ from $s$ to some node $w$
  - followed by an edge $(w, u)$ (called exit edge) to reach exit node $u$

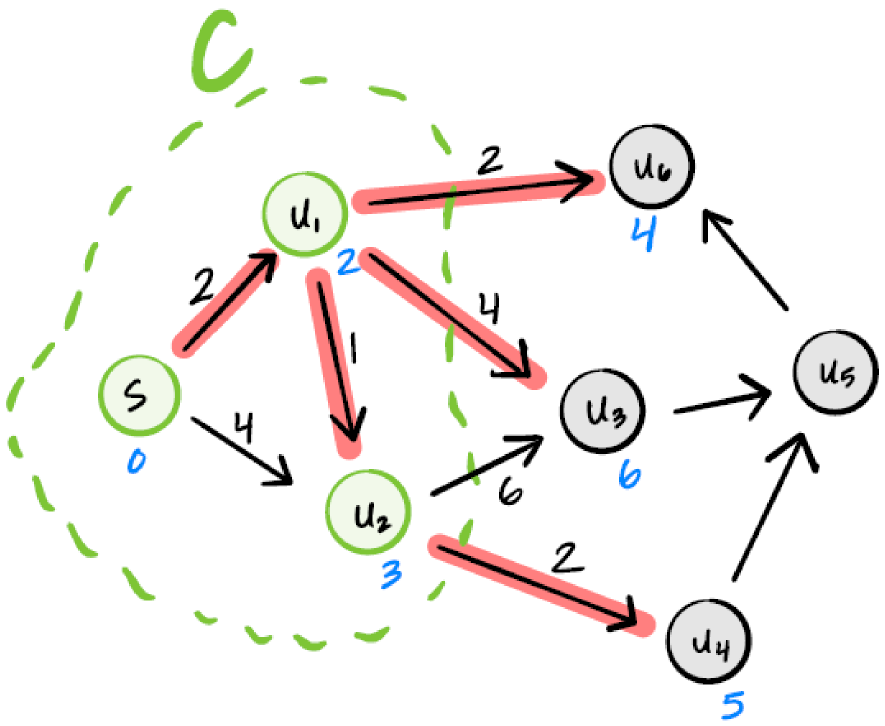


# Examples



# Shortest Exist-paths

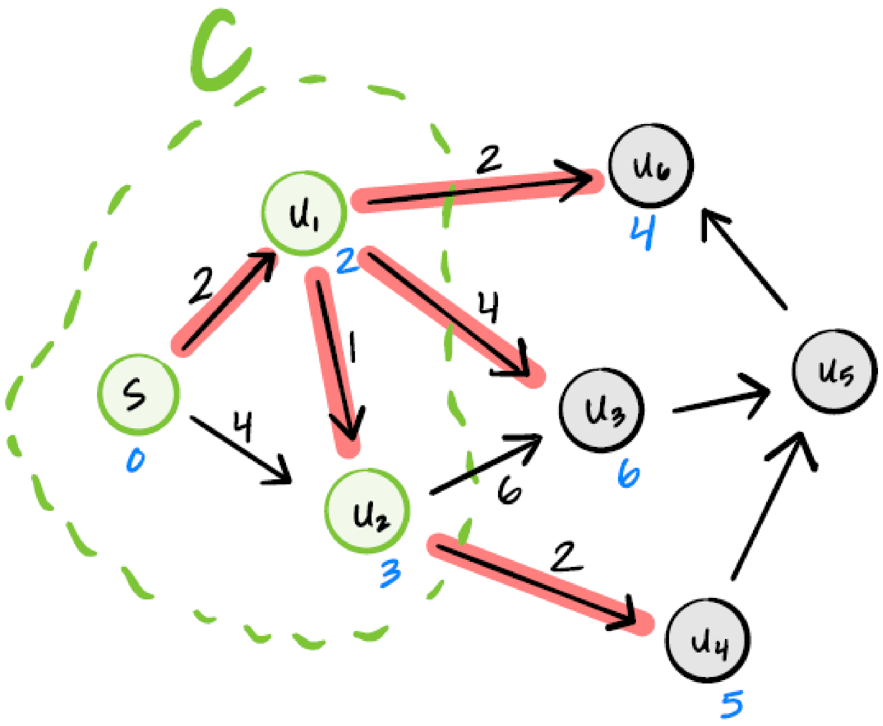
- Assume all nodes in $C$ has correct shortest path distance.
- What is the length of the shortest exit path to exist node
  - $u_3$? $u_6$? $u_5$ ?



# Shortest Exist-paths

- **Observation A.**
  - Assume all nodes in $C$ has correct shortest path distance.
  - For any node $u$ outside $C$, the shortest exist-path with exist node $u$ has length $u$.est !

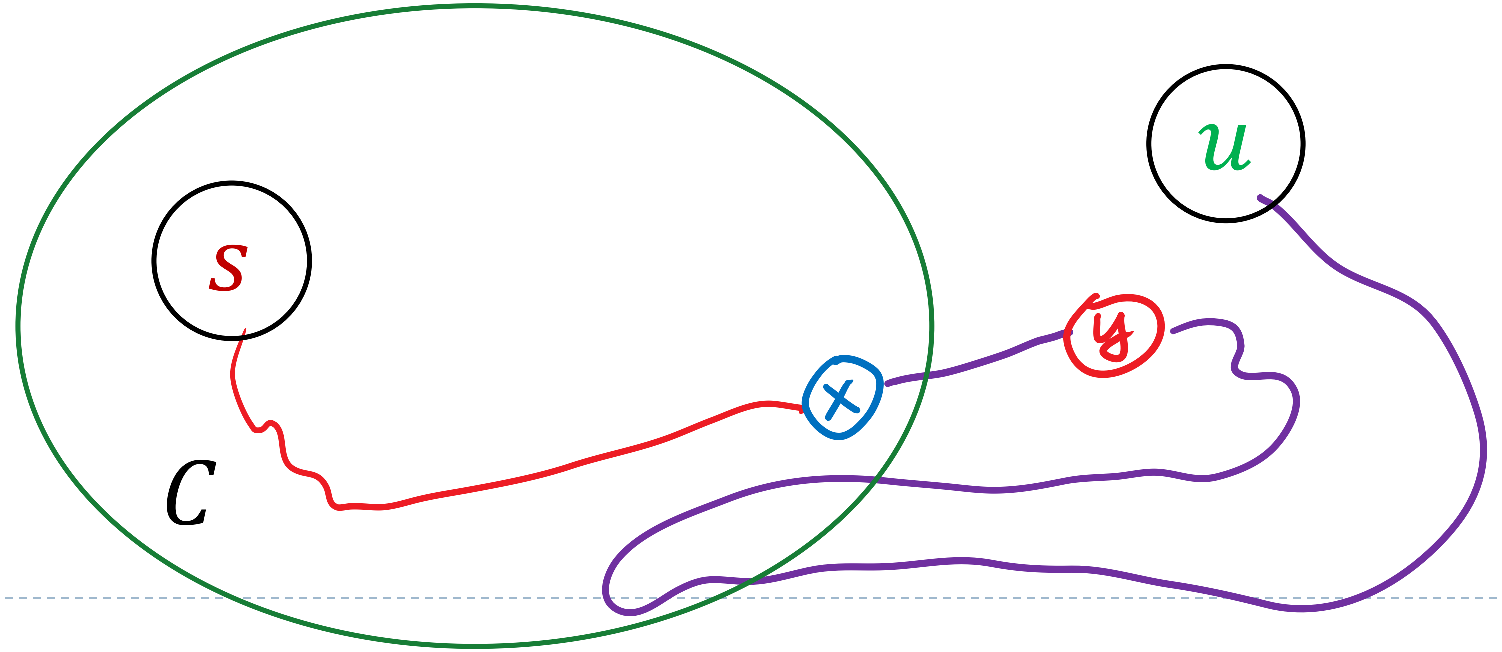
This follows from the update() operations that we do for each node in $C$ after we add it to $C$.



# Exit-path Decomposition

- Observation B:
  - Any path from $s$ to a node $u$ outside $C$ **starts with** an exist path (as it has to leave the set $C$ at some point!).
  - That is, this path can be decomposed to

    *(an exit path from $s$) + (path from exit node to $u$)*



# Correctness of Dijkstra

- **Loop invariant:**

  (i) At the beginning of each While-loop, the distance estimates already computed in set $C$ are correct.

  (ii) For each node $u$ outside set $C$, $u$.est stores the length of shortest exit path to $u$.

- **Base case:**
  - At the beginning, $C$ is empty so this holds.
- **Inductively:**
  - If this holds so far, we want to argue that after we process the next node via one While-loop iteration, it still holds.

# Outline of Dijkstra Alg (not code)

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
        C.add(u)
        for v in graph.neighbors(u):
            update(u, v, weights, est, pred)

    return est, pred
```

# Proof of Loop Invariant (i)

- Suppose $u \notin C$ is the node outside $C$ with smallest $u$.est.
- Claim: $u$.est must be the length of the shortest path distance from $s$ to node $u$.

- Proof sketch:
  - Consider any path $\pi$ from $s$ to $u$. Let $y$ be the exit node of this path.

    (length of this path $\pi$ from $s$ to $u$)

    $\geq$ (length of subpath from $s$ to $y$) + (length of subpath from $y$ to $u$)
  - Since all edge weights are positive, (length of subpath from $y$ to $u$) $\geq 0$
  - Hence we have:

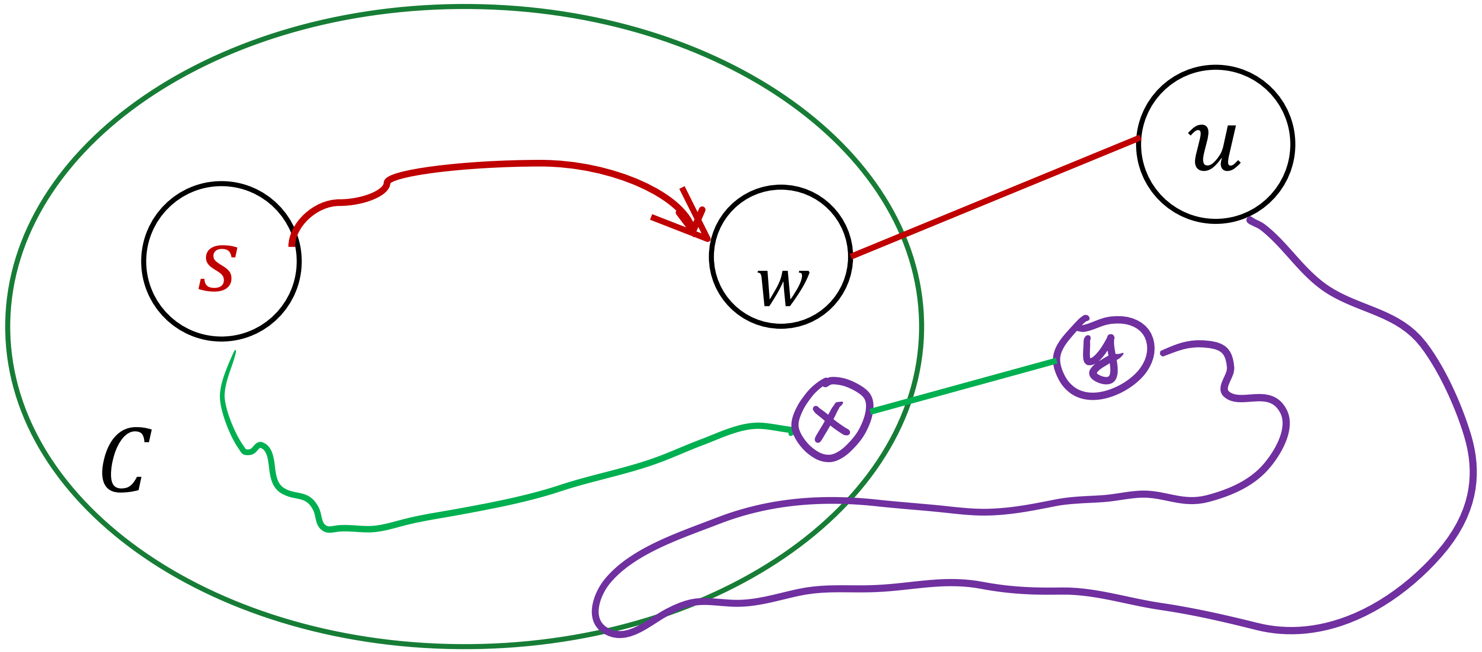
    (length of this path $\pi$ from $s$ to $u$)

    $\geq$ (length of subpath from $s$ to $y$) + $0$

    $\geq$ (length of shortest exit path from $s$ to $y$)

    $=$ $y$.est $\geq$ $u$.est $\Rightarrow$ $u$.est must be the shortest path distance.

# Illustration



# Proof of Loop invariant (ii)

- Before while-loop, set $C$
- After while-loop, set $C' = C \cup \{u\}$

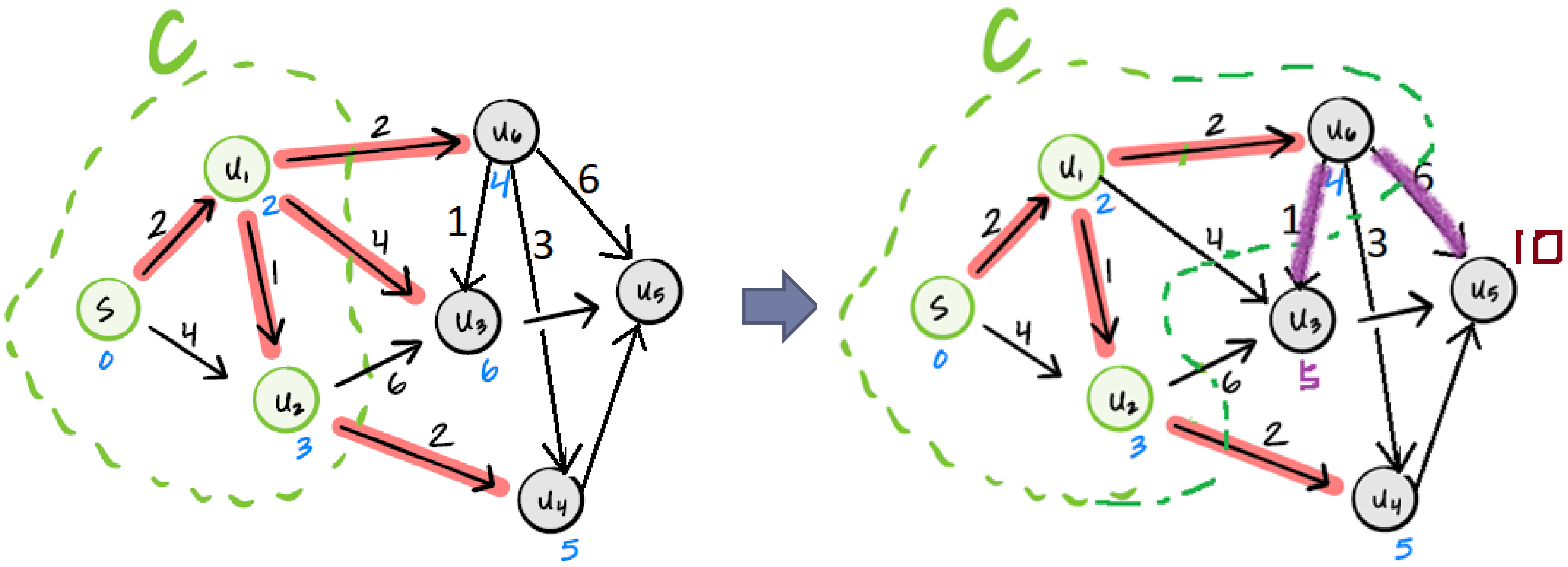
- For any node $w$ outside $C'$, $w$.est already stores the shortest exit path length through $C$

- Now we add a new node $u$ to $C$, only neighbors of $u$ may have their exist paths potentially affected

- Hence we perform update operation on each neighbor of $u$
- After that, all neighbors of $u$ finds length of shortest exit path.

# Illustrations

- Note, our algorithm will choose $u_6$ in this iteration, and afterwards, $C$ will be updated to $C \cup \{u_6\}$



# Correctness of Dijkstra

- **Loop invariant:**

  (i) At the beginning of each While-loop, the distance estimates already computed in set $C$ are correct.

  (ii) For each node $u$ outside set $C$, $u$.est stores the length of shortest exit path to $u$.

- **Base case:**
  - At the beginning, $C$ is empty so this holds.
- **Inductively:**
  - If this holds so far, then after we process the next node via one While-loop iteration, it still holds.

- To think:
  - Why do we need that all edge weights are positive in order to Dijkstra Algorithm to work?

- Exercise:
  - Give an example of a weighted graph $G$ and a source node $s$ where running Dijkstra$(G, s)$ fails to compute correct shortest path distance to some node(s).

# Implementation of Dijkstra

# Outline of Dijkstra Alg (not code)

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
        C.add(u)
        for v in graph.neighbors(u):
            update(u, v, weights, est, pred)

    return est, pred
```

# Naïve implementation of Dijkstra

```python
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    outside = set(graph.nodes)

    while outside:
        # find smallest with linear search
        u = min(outside, key=est)
        outside.remove(u)
        for v in graph.neighbors(u):
            update(u, v, weights, est, pred)

    return est, pred
```

# Time complexity of Naïve implementation

- Each while-loop takes
  - $\Theta(V)$ for finding min distance node outside
  - $\Theta(\deg(u)) = O(V)$ for Update operation
  - Hence total $\Theta(V)$ for each while-loop iteration
- Each node can only be processed once
  - Hence there are $V$ iterations of the while-loop
- Initialization takes $\Theta(V)$ time
- Total time complexity:
  - $\Theta(V) + \Theta(V) \times V = \Theta(V^2)$

Can we do better?

- Bottleneck is that we have to repeatedly perform linear-scan to find the node outside with smallest distance estimate

- We need a data structure to do the following:
  - For each outside node, maintain estimated distance
  - Extract (i.e., identify and delete) the node with smallest estimated distance
  - Update the estimated distance for a given node (in fact, decrease the estimated distance)

- We need a priority-queue data structure!

# Priority queues

- A priority queue is a data structure that allows us to store (key, value) pairs, extract the key with lowest value, and to decrease the value
  - These are exactly what we need!

- Suppose we have a priority queue class:
  - PriorityQueue(priorities) will create a priority queue from a dictionary whose values are priorities
  - The .extract_min() method removes and returns (i.e., extract) key with smallest value
  - The .change_priority(key, value) method changes key's value

# Example

```
>>> pq = PriorityQueue({
        'w': 5,
        'x': 4,
        'y': 1,
        'z': 3
        })
>>> pq.extract_min()
        'y'
>>> pq.change_priority('w', 2)
>>> pq.extract_min()
________________?
```

# Heap implementation of priority queue

- A priority queue can be implemented using a (min) heap
- min-heap implementation of priority queue:
  - PriorityQueue(priorities): takes $\Theta(n)$ time for $n = |\text{priorities}|$
  - .extract_min() : takes $\Theta(\log n)$ time where $n$ is the size of priority queue
  - .change_priority(key, value) : takes $\Theta(\log n)$ time where $n$ is the size of priority queue

# Dijkstra using priority queue

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

# Time Complexity using heap implementation of priority queue

- Creating priority queue:
  - $\Theta(V)$
- Number of .extract_min()
  - $V$
- Total costs of .extract_min()
  - $\Theta(V \lg V)$
- Number of .change_priority()
  - $\sum_{v \in V} \deg(v) = \Theta(E)$ *[deg(v) should become outdeg(v) for directed graphs]*
- Total costs of .change_priority()
  - $\Theta(E \lg V)$
- Total time complexity:
  - $\Theta((V + E) \lg V)$

- Using Fibonacci heap, one can improve the time complexity of Dijkstra algorithm to
  - $\Theta(E + V \lg V)$

# Summary

- Graph traversal / search strategy (BFS/DFS)
  - $\Theta(V + E)$
  - BFS can be used to compute single source shortest path for unweighted graphs, or for graphs where all edges having the same weight.

- Graph single source shortest path
  - Bellman-Ford for arbitrary graphs: $\Theta(V \cdot E)$
  - Dijkstra for positively-weighted graphs: $\Theta\big((V + E)\lg V\big)$
    - Can be improved to $\Theta(E + V \lg V)$

FIN