

DSC 40B

Theoretical Foundations II

Dijkstra's Algorithm

Shortest Path Algorithms

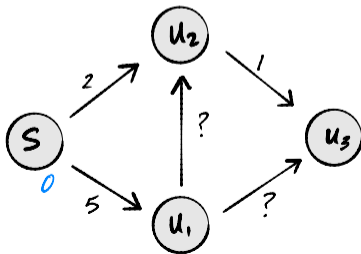
- ▶ **Bellman-Ford** and **Dijkstra's** are shortest path algorithms:
 - INPUT: weighted graph, source vertex s .
 - OUTPUT: shortest paths from s to every other node.
- ▶ Both work by:
 - ▶ keeping estimates of shortest path distances;
 - ▶ iteratively **updating** estimates until they're correct.

Shortest Path Algorithms

- ▶ We saw Bellman-Ford last time; takes time $\Theta(VE)$.
- ▶ Dijkstra's will be faster, but can't handle negative weights.

Dijkstra's Algorithm

- ▶ On every iteration, Bellman-Ford updates all edges – many don't need to be updated.
- ▶ If we **assume** all edge weights are positive, we can rule out some paths immediately:



Dijkstra's Idea

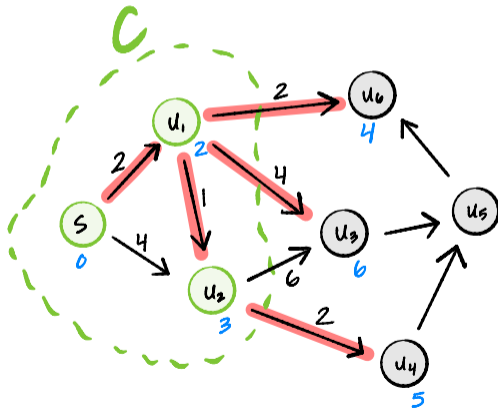
- ▶ Keep track of set C of correct nodes.
- ▶ At every step, add node outside of C with smallest estimated distance; update its neighbors.

Outline of Dijkstra's Algorithm

```
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
```

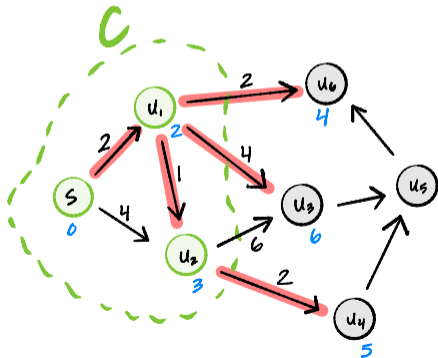
Proof Idea

- Claim: at beginning of any iteration of Dijkstra's, if u is node $\notin C$ with smallest estimated distance, the shortest path to u has been correctly discovered.



Proof Idea

- ▶ Let u be node outside of C for which $\text{est}[u]$ is smallest.
- ▶ We've discovered a path from s to u of length $\text{est}[u]$.
- ▶ Any path from s to u has to exit C somewhere.
- ▶ Any path from s to u will cost at least $\text{est}[u]$ just to exit C .



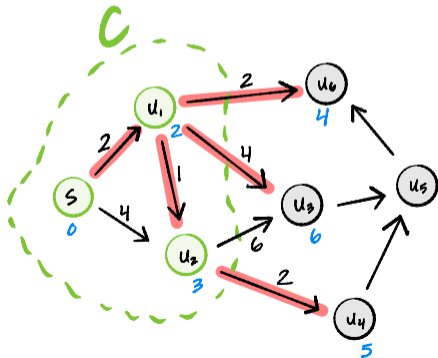
DSC 40B

Theoretical Foundations II

Analysis

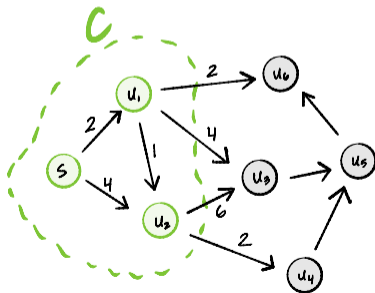
Proof Idea

- ▶ Let u be node outside of C for which $\text{est}[u]$ is smallest.
- ▶ We've discovered a path from s to u of length $\text{est}[u]$.
- ▶ Any path from s to u has to exit C somewhere.
- ▶ Any path from s to u will cost at least $\text{est}[u]$ just to exit C .



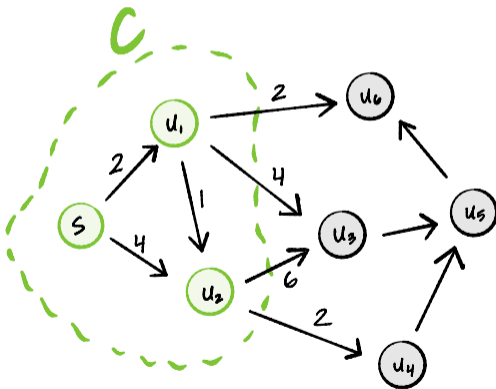
Exit Paths

- ▶ An **exit path from s through C** is a path for which:
 - ▶ the first node is s ;
 - ▶ the last node (a.k.a., the **exit node**) is **not** in C ;
 - ▶ all other nodes **are** in C .
- ▶ Example:



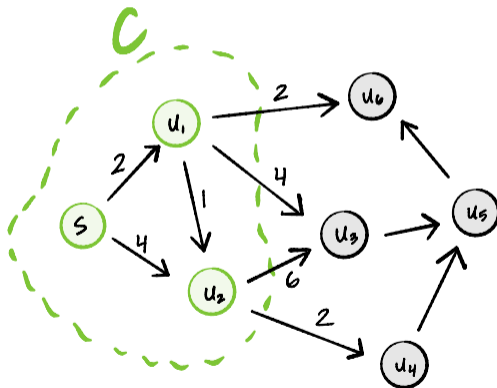
Exit Paths

- True or False: this is an exit path from s through C .



Exit Paths

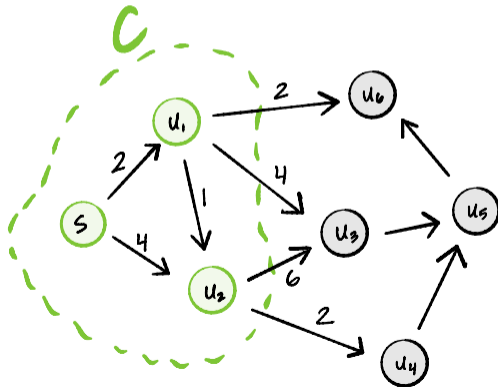
- True or False: this is an exit path from s through C .



Path Decomposition

- Any path from s to a node u outside of C can be broken into two parts:

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



Path Decomposition

- ▶ Consider any path from s to $u \notin C$.

- ▶ Suppose e is the path's exit node.

- ▶ We have:

(length of the path)

= (length of exit path to e) + (length of path from e to u)

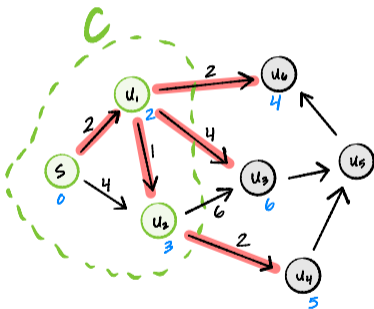
\geq (length of shortest exit path to e) + (length of path from e to u)

- ▶ Since edge weights are positive, all path lengths ≥ 0 :

\geq (length of shortest exit path to e) + 0

Shortest Exit Paths

- Example: What is the shortest exit path with exit node u_3 ?



- If u is outside of C , then the length of the shortest exit path with exit node e is $\text{est}[e]$.

Proof Idea

- ▶ Suppose u is a node outside of C for which $\text{est}[u]$ is smallest.
- ▶ Consider any path from s to u , and let e be the path's exit node.
- ▶ We have:
$$\begin{aligned} & (\text{length of this path from } s \text{ to } u) \\ & \geq (\text{length of shortest exit path to } e) + 0 \\ & = \text{est}[e] \\ & \geq \text{est}[u] \end{aligned}$$
- ▶ That is, any path from s to u has length $\geq \text{est}[u]$.
- ▶ We've already found one with length $\text{est}[u]$; this proves that it is the shortest.

Outline of Dijkstra's Algorithm

```
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
```

Dijkstra's Algorithm: Naïve Implementation

```
1 def dijkstra(graph, weights, source):
2     est = {node: float('inf') for node in graph.nodes}
3     est[source] = 0
4     pred = {node: None for node in graph.nodes}
5
6     outside = set(graph.nodes)
7
8     while outside:
9         # find smallest with linear search
10        u = min(outside, key=est)
11        outside.remove(u)
12        for v in graph.neighbors(u):
13            update(u, v, weights, est, pred)
14
15    return est, pred
```

Priority Queues

- ▶ A **priority queue** allows us to store (key, value) pairs, efficiently return key with lowest value.
- ▶ 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 key with smallest value.
 - ▶ The `.change_priority(key, value)` method changes key's value (only if key in queue).

Example

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

Dijkstra's Algorithm: 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
```

Heaps

- ▶ A priority queue can be implemented using a **heap**.
- ▶ If a **binary min-heap** is used:
 - ▶ `PriorityQueue(est)` takes $\Theta(V)$ time.
 - ▶ `.extract_min()` takes $O(\log V)$ time.
 - ▶ `.change_priority()` takes $O(\log V)$ time.

Time Complexity Using Min Heap

```
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: _____

