DSC 40B

Theoretical Foundations II

Lecture 15 | Part 1

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.

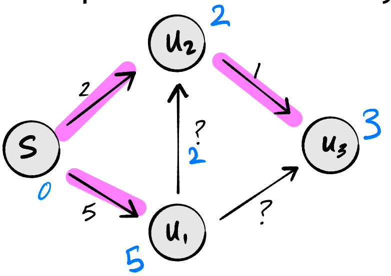
Shortest Path Algorithms

$\Theta(V + V \cdot E)$

- 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

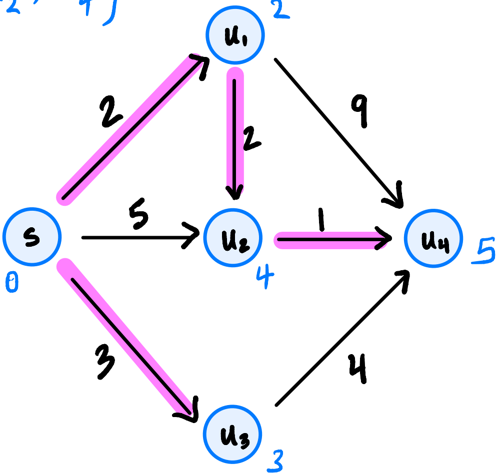
- Keep track of set C of “correct” nodes.
  - Nodes whose distance estimate is correct.
- At every step, add node outside of C with smallest estimated distance; update only its neighbors.
- A “greedy” algorithm.

Outline of Dijkstra’s Algorithm

```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}

    # 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
```

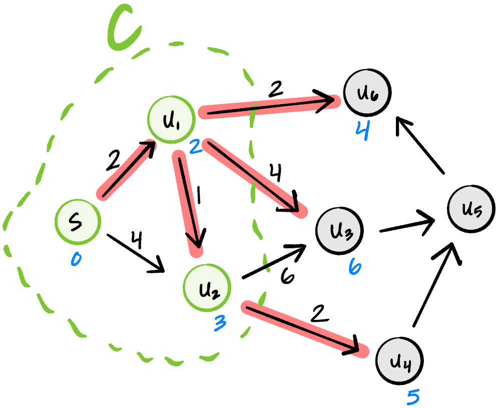
Example

$C = \{s, u_1, u_3, u_2, u_4\}$



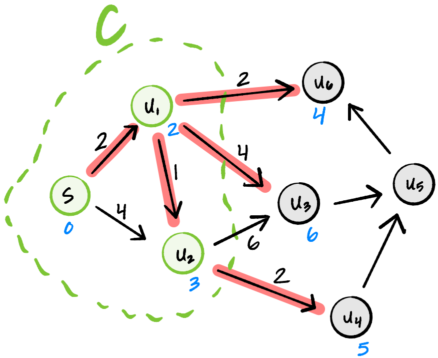
Proof Idea

- 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 `est[u]` is smallest.
- We've discovered a path from s to u of length `est[u]`.
- Any path from s to u has to exit C somewhere.
- Any path from s to u will cost at least `est[u]` just to exit C.

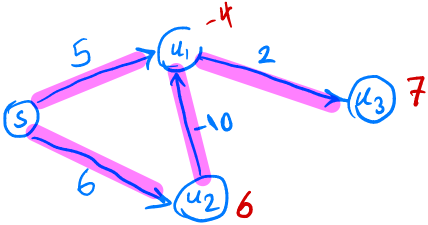


Exercise

Why do the edge weights need to be positive? Come up with a simple example graph with some negative edge weights where Dijkstra's fails to compute the correct shortest path.

Exercise

Why do the edge weights need to be positive? Come up with a simple example graph with some negative edge weights where Dijkstra's fails to compute the correct shortest path.



$C = \{s, u_1, u_2, u_3\}$

DSC 40B

Theoretical Foundations II

Lecture 15 | Part 2

Implementation

Outline of Dijkstra's Algorithm

```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}

    # 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

```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
```

$\Theta(|V|)$ → while outside:

$\Theta(|V|)$ → u = min(outside, key=est)

$\Theta(|E|)$ [for v in graph.neighbors(u): update(u, v, weights, 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.

Example

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

'w'

{'w': 5, 'x': 4, 'z': 3}

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
```

extract_min() → T_{em}

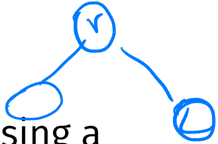
change_priority(v, est[v]) → T_{cp}

$\Theta(V \cdot T_{em})$

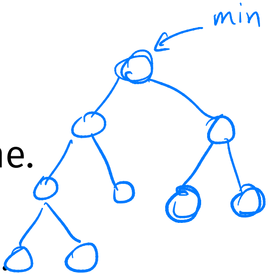
$\Theta(E \cdot T_{cp})$

Total: $\Theta(V \cdot T_{em} + E \cdot T_{cp})$

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 $\Theta(\log V)$ time.
  - `.change_priority()` takes $\Theta(\log V)$ 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
```

$\Theta(V)$ ← priority_queue = PriorityQueue(est)

extract_min → $\Theta(\log(V)) = T_{em}$

change_priority: $\Theta(\log V) = T_{cp}$

$\Theta(V + V \cdot \log V + E \cdot \log V)$

$\Theta((V+E)\log V) =$

- Time complexity: $\Theta(V \log V + E \cdot \log V)$

DSC 40B

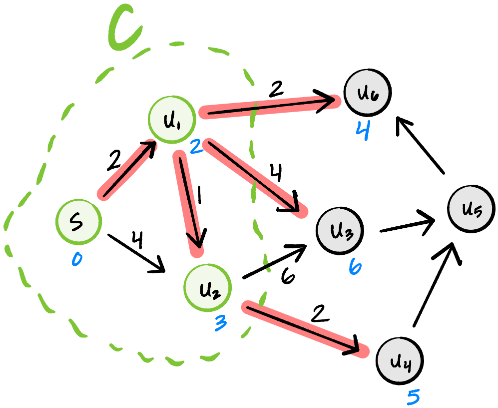
Theoretical Foundations II

Lecture 15 | Part 3

Proof

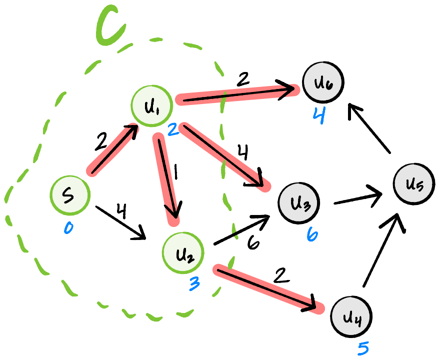
Proof Idea

- 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

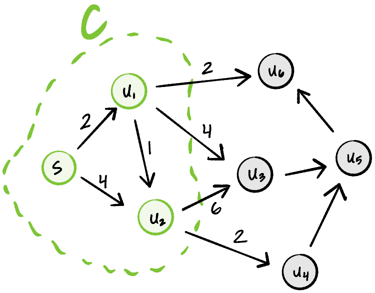
- Let u be node outside of C for which `est[u]` is smallest.
- We've discovered a path from s to u of length `est[u]`.
- Any path from s to u has to exit C somewhere.
- Any path from s to u will cost at least `est[u]` just to exit C.



Exit Paths

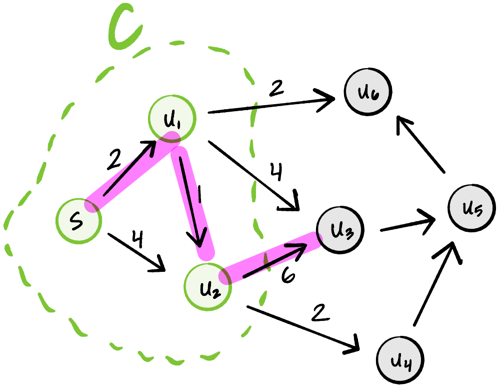
- An **exit path from s through C** is a path for which:
  - the first node is s;
  - the last node (a.k.a., the **exit node**) is **not** in C;
  - all other nodes **are** in C.

- Example:



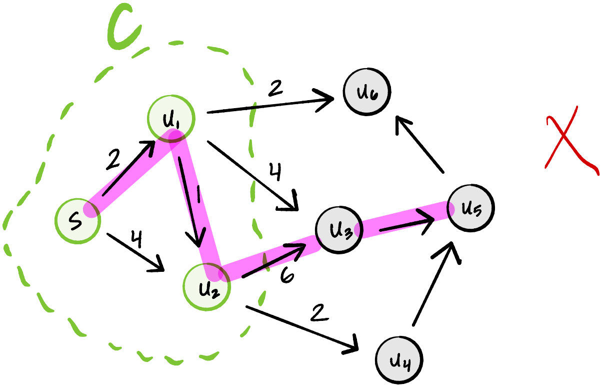
Exit Paths

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



Exit Paths

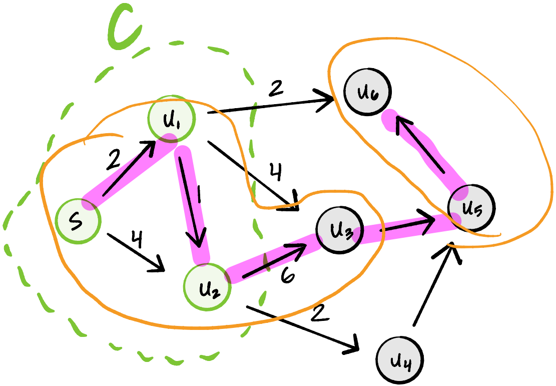
- True or False: this is an exit path from s through C.



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)

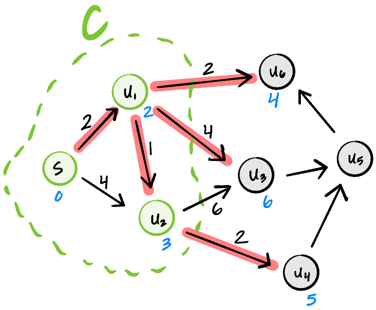
  ≥ (length of shortest exit path to e) + (length of path from e to u)

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

  ≥ (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 `est[e]`.

Proof Idea

- Suppose u is a node outside of C for which `est[u]` is smallest.
- Consider any path from s to u, and let e be the path's exit node.
- We have:

  (length of this path from s to u)

  ≥ (length of shortest exit path to e) + 0

  = `est[e]`

  ≥ `est[u]`

- That is, any path from s to u has length ≥ `est[u]`.
- We've already found one with length `est[u]`; this proves that it is the shortest.