Algorithms & Models of Computation CS/ECE 374, Fall 2020

20.6.2 Implementing Prim's Algorithm

Implementing Prim's Algorithm

```
Prim_ComputeMST

E is the set of all edges in G

S = \{1\}

T is empty (* T will store edges of a MST *)

while S \neq V do

pick e = (v, w) \in E such that

v \in S and w \in V - S

e has minimum cost

T = T \cup e

S = S \cup w

return the set T
```

Analysis

- 1 Number of iterations = O(n), where n is number of vertices
- 2 Picking e is O(m) where m is the number of edges
- Total time O(nm)

Implementing Prim's Algorithm

```
Prim_ComputeMST

E is the set of all edges in G

S = \{1\}

T is empty (* T will store edges of a MST *)

while S \neq V do

pick e = (v, w) \in E such that

v \in S and w \in V - S

e has minimum cost

T = T \cup e

S = S \cup w

return the set T
```

Analysis

- Number of iterations = O(n), where n is number of vertices
- Picking e is O(m) where m is the number of edges
- Total time O(nm)

Implementing Prim's Algorithm

```
Prim_ComputeMST

E is the set of all edges in G

S = \{1\}

T is empty (* T will store edges of a MST *)

while S \neq V do

pick e = (v, w) \in E such that

v \in S and w \in V - S

e has minimum cost

T = T \cup e

S = S \cup w

return the set T
```

Analysis

- **1** Number of iterations = O(n), where *n* is number of vertices
- 2 Picking e is O(m) where m is the number of edges

Total time O(nm)

Implementing Prim's Algorithm

```
Prim_ComputeMST

E is the set of all edges in G

S = \{1\}

T is empty (* T will store edges of a MST *)

while S \neq V do

pick e = (v, w) \in E such that

v \in S and w \in V - S

e has minimum cost

T = T \cup e

S = S \cup w

return the set T
```

Analysis

- **1** Number of iterations = O(n), where *n* is number of vertices
- 2 Picking e is O(m) where m is the number of edges
- Total time O(nm)

More Efficient Implementation

```
Prim_ComputeMST
     E is the set of all edges in G
     S = \{1\}
     T is empty (* T will store edges of a MST *)
     for \mathbf{v} \notin \mathbf{S}, \mathbf{a}(\mathbf{v}) = \min_{\mathbf{w} \in \mathbf{S}} \mathbf{c}(\mathbf{w}, \mathbf{v})
     for v \notin S, e(v) = w such that w \in S and c(w, v) is minimum
     while S \neq V do
           pick v with minimum a(v)
           T = T \cup \{(e(v), v)\}
           S = S \cup \{v\}
           update arrays a and e
     return the set T
```

Maintain vertices in $V \setminus S$ in a priority queue with key a(v).

More Efficient Implementation

```
Prim_ComputeMST
     E is the set of all edges in G
     S = \{1\}
     T is empty (* T will store edges of a MST *)
     for \mathbf{v} \notin \mathbf{S}, \mathbf{a}(\mathbf{v}) = \min_{\mathbf{w} \in \mathbf{S}} \mathbf{c}(\mathbf{w}, \mathbf{v})
     for v \notin S, e(v) = w such that w \in S and c(w, v) is minimum
     while S \neq V do
           pick v with minimum a(v)
           T = T \cup \{(e(v), v)\}
           S = S \cup \{v\}
           update arrays a and e
     return the set T
```

Maintain vertices in $V \setminus S$ in a priority queue with key a(v).

More Efficient Implementation

```
Prim_ComputeMST
     E is the set of all edges in G
     S = \{1\}
     T is empty (* T will store edges of a MST *)
     for \mathbf{v} \notin \mathbf{S}, \mathbf{a}(\mathbf{v}) = \min_{\mathbf{w} \in \mathbf{S}} \mathbf{c}(\mathbf{w}, \mathbf{v})
     for v \notin S, e(v) = w such that w \in S and c(w, v) is minimum
     while S \neq V do
           pick v with minimum a(v)
           T = T \cup \{(e(v), v)\}
           S = S \cup \{v\}
          update arrays a and e
     return the set T
```

Maintain vertices in $V \setminus S$ in a priority queue with key a(v).

Algorithms & Models of Computation CS/ECE 374, Fall 2020

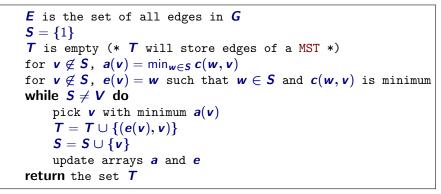
20.6.3 Implementing Prim's algorithm with priority queues

Priority Queues

Data structure to store a set S of n elements where each element $v \in S$ has an associated real/integer key k(v) such that the following operations

- makeQ: create an empty queue
- IndMin: find the minimum key in S
- **§** extractMin: Remove $v \in S$ with smallest key and return it
- add(v, k(v)): Add new element v with key k(v) to S
- **Delete**(v): Remove element v from S
- decreaseKey (v, k'(v)): decrease key of v from k(v) (current key) to k'(v) (new key). Assumption: $k'(v) \le k(v)$
- Image merge two separate priority queues into one

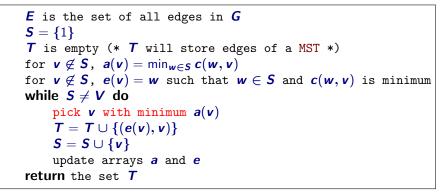
Prim's using priority queues



Maintain vertices in $V \setminus S$ in a priority queue with key a(v)

- **Organization** Requires O(n) extractMin operations
- Requires O(m) decreaseKey operations

Prim's using priority queues

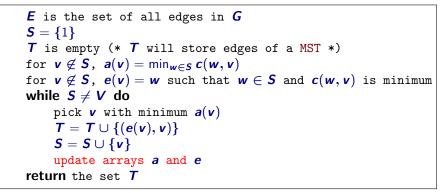


Maintain vertices in $V \setminus S$ in a priority queue with key a(v)

• Requires O(n) extract Min operations

Requires O(m) decreaseKey operations

Prim's using priority queues



Maintain vertices in $V \setminus S$ in a priority queue with key a(v)

- **Q** Requires O(n) extract Min operations
- **2** Requires O(m) decreaseKey operations

Running time of Prim's Algorithm

O(n) extractMin operations and O(m) decreaseKey operations

- Using standard Heaps, extractMin and decreaseKey take $O(\log n)$ time. Total: $O((m + n) \log n)$
- **2** Using Fibonacci Heaps, $O(\log n)$ for extractMin and O(1) (amortized) for decreaseKey. Total: $O(n \log n + m)$.
- Prim's algorithm and Dijkstra's algorithms are similar. Where is the difference?
- Prim's algorithm = Dijkstra where length of a path π is the weight of the heaviest edge in π. (Bottleneck shortest path.)

Running time of Prim's Algorithm

O(n) extractMin operations and O(m) decreaseKey operations

- Using standard Heaps, extractMin and decreaseKey take $O(\log n)$ time. Total: $O((m + n) \log n)$
- **2** Using Fibonacci Heaps, $O(\log n)$ for extractMin and O(1) (amortized) for decreaseKey. Total: $O(n \log n + m)$.
- Prim's algorithm and Dijkstra's algorithms are similar. Where is the difference?
- Prim's algorithm = Dijkstra where length of a path π is the weight of the heaviest edge in π. (Bottleneck shortest path.)

Running time of Prim's Algorithm

O(n) extractMin operations and O(m) decreaseKey operations

- Using standard Heaps, extractMin and decreaseKey take $O(\log n)$ time. Total: $O((m + n) \log n)$
- **2** Using Fibonacci Heaps, $O(\log n)$ for extractMin and O(1) (amortized) for decreaseKey. Total: $O(n \log n + m)$.
- Prim's algorithm and Dijkstra's algorithms are similar. Where is the difference?
- Prim's algorithm = Dijkstra where length of a path π is the weight of the heaviest edge in π. (Bottleneck shortest path.)

THE END

(for now)

. . .