17.3.8
Dijkstra using 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:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in S.
3. **extractMin**: Remove $v \in S$ with smallest key and return it.
4. **insert**$(v, k(v))$: Add new element v with key $k(v)$ to S.
5. **delete**(v): Remove element v from S.
6. **decreaseKey**$(v, k'(v))$: decrease key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
7. **meld**: merge two separate priority queues into one.

All operations can be performed in $O(\log n)$ time.
decreaseKey is implemented via **delete** and **insert**.

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:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in S.
3. **extractMin**: Remove $v \in S$ with smallest key and return it.
4. **insert**$(v, k(v))$: Add new element v with key $k(v)$ to S.
5. **delete**(v): Remove element v from S.
6. **decreaseKey**$(v, k'(v))$: decrease key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
7. **meld**: merge two separate priority queues into one.

All operations can be performed in $O(\log n)$ time.
decreaseKey is implemented via **delete** and **insert**.

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:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in S.
3. **extractMin**: Remove $v \in S$ with smallest key and return it.
4. **insert**$(v, k(v))$: Add new element v with key $k(v)$ to S.
5. **delete**(v): Remove element v from S.
6. **decreaseKey**$(v, k'(v))$: decrease key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
7. **meld**: merge two separate priority queues into one.

All operations can be performed in $O(\log n)$ time.
decreaseKey is implemented via **delete** and **insert**.

Dijkstra's Algorithm using Priority Queues

$Q \leftarrow$ makePQ()
insert(Q, $(s, 0)$)
for each node $u \neq s$ **do**
    insert(Q, (u, ∞))
$X \leftarrow \emptyset$
for $i = 1$ to $|V|$ **do**
    $(v, \text{dist}(s, v)) = extractMin(Q)$
    $X = X \cup \{v\}$
    for each u in $\text{Adj}(v)$ **do**
        decreaseKey$\Big(Q, \big(u, \min\big(\text{dist}(s, u),\ \text{dist}(s, v) + \ell(v, u)\big)\big)\Big)$.

Priority Queue operations:

1. $O(n)$ **insert** operations
2. $O(n)$ **extractMin** operations
3. $O(m)$ **decreaseKey** operations

Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

1. All operations can be done in $O(\log n)$ time

Dijkstra's algorithm can be implemented in $O((n + m) \log n)$ time.

Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

1. All operations can be done in $O(\log n)$ time

Dijkstra's algorithm can be implemented in $O((n + m)\log n)$ time.

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ amortized time: ℓ **decreaseKey** operations for $\ell \geq n$ take together $O(\ell)$ time
3. Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.
2. Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps, for example.
3. Boost library implements both Fibonacci heaps and rank-pairing heaps.

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ amortized time: ℓ **decreaseKey** operations for $\ell \geq n$ take together $O(\ell)$ time
3. Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.
2. Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps, for example.
3. Boost library implements both Fibonacci heaps and rank-pairing heaps.

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ amortized time: ℓ **decreaseKey** operations for $\ell \geq n$ take together $O(\ell)$ time
3. Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.
2. Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps, for example.
3. Boost library implements both Fibonacci heaps and rank-pairing heaps.

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ amortized time: ℓ **decreaseKey** operations for $\ell \geq n$ take together $O(\ell)$ time
3. Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.
2. Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps, for example.
3. Boost library implements both Fibonacci heaps and rank-pairing heaps.

THE END

...

(for now)