Shortest Paths in Weighted Graphs

Shortest Path $0 \to 5$

By least total weight

$0$:
- Runners $0 \to 1$ (arrives @ 10) 
  - (arrives @ 30)

$10$:
- Runner $0 \to 1$ arrives
- Dispatch $1 \to 3$ (arrives @ 40)
  - $1 \to 2$ (arrives @ 20) bents runner $0 \to 2$

Keep track for each city of earliest arrival
Source of earliest arrival
Whether runners sent
Priority Queue:

- maintain keys and priorities
- vertices = arrival times
  (distances)
- build queue: given keys and initial priorities, initialize
  vertices 0 for all except source
  0 for source
- decrease priority: given key, decrease its priority
- extract minimum: get key with lowest priority and remove it

Simple implementation:
unsorted array

integer items are indices
value @ index is priority
BFS

BFS(V,E,s)

for each vertex u in V
    color[u] <- WHITE
    dist[u] <- infinity
    pred[u] <- NULL

color[s] <- GRAY
dist[s] <- 0
pred[s] <- NULL
Q <- {[s]

while not Q.is_empty()
    u <- Q.dequeue()
    for each v such that (u, v) is an edge
        if color[v] = WHITE
            pred[v] <- u
            dist[v] <- dist[u] + 1 w(u,v)
            color[v] <- GRAY
            Q.enqueue(v)
    color[u] <- BLACK
<table>
<thead>
<tr>
<th>Operation</th>
<th>Unsorted Array</th>
<th>Balanced BST</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Queue</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Add All Priorities</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Decrease Priority</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Extract Minimum</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Simple implementation: unsorted array

- Integer items are indices
- Value @ index is priority

```
0 1 2 3 4 5
0 10 20 0 0 0
```
Dijkstra(V,E,s)

for each vertex u in V
    color[u] <- WHITE
    dist[u] <- infinity
    pred[u] <- NULL

color[s] <- GRAY
dist[s] <- 0
pred[s] <- NULL
Q <- pq_build(n, dist)

while not Q.is_empty() do
    u <- Q.extract_min()
    for each v such that (u, v) is an edge
        if color[v] != BLACK and dist[u] + w(u, v) < dist[v]
            pred[v] <- u
            dist[v] <- dist[u] + w(u, v)
            color[v] <- GRAY
            Q.decrease_priority(v, dist[v])
    color[u] <- BLACK

Dijkstra : for least-cost paths in graphs w/ non neg. weight edges
Heaps: shape: complete binary tree
(Min-heap)  
\[\Rightarrow\] every level except last is full
all leaves as far left as possible

order: value at node \(\leq\) value in children
(so min item is in root)

\[
\begin{align*}
\text{curr: } 5 & \quad \text{left: } 2 \cdot \text{curr} + 1 \\
\text{right: } 2 \cdot \text{curr} + 2 & \quad \text{parent: } \left\lfloor \frac{\text{curr} - 1}{2} \right\rfloor
\end{align*}
\]

\[
0 \quad 1 \\
1 \quad 2 \\
3 \quad 4 \\
5 \quad 6
\]

1 6 4 9 14 7 5 13 12 20 14 10 11 9 0 16 15 14 29 94

\[
\begin{array}{c}
1 \\
1 \\
1 \\
1 \\
1
\end{array}
\]

1 7 3 9 10 5

1

1 3

1 10 5
**HEAP-ADD**

```
add (item, 10)
```

```
7
/
13 / 14
/
62 / 46 / 28 / 17
/
100 / 89 / 10
```

```
7
/
13 / 14
/
62 / 10 / 28 / 17
/
100 / 89 / 46
```

```
10
/
13 / 14
/
62 / 10 / 28 / 17
/
100 / 89 / 46
```

**HEAP-ADD** \((H, k)\)

\[ \begin{align*}
A[\text{heap-size}] & \leftarrow A[\text{heap-size}] + 1 \\
A[k] & \leftarrow k \\
\text{REHEAP-UP}(A[\text{heap-size}])
\end{align*} \]

**REHEAP-UP** \((A[i])\)

```
while \( (i > 0 \text{ and } A[i] < A[\text{parent}(i)]) \) do
- order violation / parent
- \( O(h) = O(\log n) \) iterations
- \( O(1) \) per iteration
- swap \( A[i] \) and \( A[\text{parent}(i)] \)
- \( i \leftarrow \text{parent}(i) \) repeat at loc new item swapped into so \( O(\log n) \) overall
```
**EXTRACT-MW**

```
EXTRACT-MW(A)

min ← A[0]  // get min from root
swap (A, 0, A[heap-size - 1])  // swap last node into root
A.heap-size ← A.heap-size - 1
HEAPIFY(A, 0)  // restore order property

O(1) = O(log n)  // overall

HEAPIFY(A, i)

while (left(i) < A.heap-size)
    smallest ← left(i) if A[left(i)] ≤ A[right(i)]  // get index of
    or right(i) ≥ A.heap-size
    if A(i) > A[smallest]
        swap A(i), A[smallest]
    i ← smallest
    if A(i) < A[smallest]
        repeat until done
```
**Change Priority**

![Diagram of priority changes]

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NBO</td>
<td>1</td>
</tr>
<tr>
<td>BTV</td>
<td>4</td>
</tr>
<tr>
<td>LIM</td>
<td>0</td>
</tr>
<tr>
<td>BHM</td>
<td>3</td>
</tr>
<tr>
<td>MKE</td>
<td>6</td>
</tr>
<tr>
<td>CGK</td>
<td>2</td>
</tr>
<tr>
<td>LAX</td>
<td>7</td>
</tr>
</tbody>
</table>

**Map**

- Get index of key from map: $O(1)$
- Update priority if < current priority: $O(1)$
- REHEAP_UP (modify and update map on each swap): $O(\log n)$ overall
- Get/update map (expected if keys are not $0...n-1$ so using hash table): $O(1)$
**BUILD-HEAP (A)**

For $i = \left\lceil \frac{A.\text{heap-size} - 1}{2} \right\rceil$ down to 0

- $O(n)$ iterations
- $O(\log n)$ per iteration
- $O(n \log n)$ overall

(can show $O(n)$ overall w/ more careful math)

- ~$\frac{n}{4}$ nodes at next-to-last level, 1 iteration inside `HEAPIFY`
- ~$\frac{n}{8}$ nodes at next level up, 2 iterations
- ~$\frac{n}{16}$ nodes at next level up, 3 iterations

Total iterations in `HEAPIFY` = $\frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \ldots$

= $n \cdot \sum_{i=1}^{\infty} (\frac{1}{4})^i$

For $0 < r < 1$

\[ \sum_{i=0}^{\infty} r^i = \frac{1}{1-r} \]

\[ \frac{\frac{d}{dr}}{\frac{d}{dr}} \text{ both sides} \left( \sum_{i=0}^{\infty} i \cdot r^i = \frac{1}{(1-r)^2} \right) \]

= $\frac{1}{4} \cdot n \cdot \sum_{i=0}^{\infty} i \cdot (\frac{1}{4})^i$

= $O(n)$

O(1) work per iteration, so $O(n)$ overall