The challenges of implementing Dijkstra’s algorithm

Gerth Stølting Brodal
Aarhus University

(Work presented at the 11th International Conference on Fun with Algorithms, FUN 2022)
Background

- Bachelorproject = shortest paths on Open Street Map graphs
- Students have trouble implementing Dijkstra's algorithm in Java™
<node id="4606377719" visible="true" version="1" changeset="45143381" timestamp="2017-01-13T19:11:48Z" user="Dyserth" uid="1892838" lat="53.4064488" lon="-2.9660774"/>
<way id="4961017" visible="true" version="11" changeset="66454995" timestamp="2019-01-19T14:25:32Z" user="Redhochs" uid="1744940">
<nd ref="348087292"/>
<nd ref="4606377719"/>
<nd ref="268780437"/>
<nd ref="268779145"/>
<nd ref="2700274352"/>
<nd ref="268779145"/>
<tag k="highway" v="pedestrian"/>
<tag k="lit" v="yes"/>
<tag k="name" v="Ashton Street"/>
<tag k="surface" v="paving_stones"/>
</way>
<way id="826473129" visible="true" version="3" changeset="129789472" timestamp="2022-12-06T16:23:14Z" user="pmyteh" uid="81653">
<nd ref="107366213"/>
<nd ref="3468209232"/>
<nd ref="3476406048"/>
<nd ref="4112993338"/>
<nd ref="2188828954"/>
<nd ref="534390991"/>
<nd ref="7839585527"/>
<nd ref="348087295"/>
<tag k="bicycle" v="yes"/>
<tag k="foot" v="yes"/>
<tag k="highway" v="tertiary"/>
<tag k="horse" v="yes"/>
<tag k="lanes" v="3"/>
<tag k="lit" v="yes"/>
<tag k="maxspeed" v="30 mph"/>
<tag k="name" v="Brownlow Hill"/>
<tag k="oneway" v="no"/>
<tag k="postal_code" v="L3"/>
<tag k="sidewalk" v="both"/>
<tag k="surface" v="asphalt"/>
</way>
Dijkstra’s algorithm (1956)

- Non-negative edge weights
- Visits nodes in increasing distance from source

```
proc Dijkstra_1(V, E, δ, s)
    dist[v] = +∞ for all v ∈ V \ {s}
    dist[s] = 0
    Insert(Q, ⟨dist[s], s⟩)
    while Q ≠ ∅ do
        ⟨d, u⟩ = ExtractMin(Q)
        for (u, v) ∈ E ∩ (⟨u⟩ × V) do
            if dist[u] + δ(u, v) < dist[v] then
                dist[v] = dist[u] + δ(u, v)
                if v ∈ Q then
                    DecreaseKey(Q, v, dist[v])
                else
                    Insert(Q, ⟨v, dist[v]⟩)
        return dist
```

```
proc Dijkstra_2(V, E, δ, s)
    dist[v] = +∞ for all v ∈ V \ {s}
    dist[s] = 0
    Insert(Q, ⟨dist[s], s⟩)
    while Q ≠ ∅ do
        ⟨d, u⟩ = ExtractMin(Q)
        for (u, v) ∈ E ∩ (⟨u⟩ × V) do
            if dist[u] + δ(u, v) < dist[v] then
                dist[v] = dist[u] + δ(u, v)
                if v ∈ Q then
                    Remove(Q, v)
                    Insert(Q, ⟨v, dist[v]⟩)
        return dist
```

Fibonacci heaps (Fredman, Tarjan 1984) ⇒ O(m + n \cdot log n)

Relax

Remove ⇒ O(m \cdot log n)
The Challenge - Java's builtin binary heap

- no decreasekey
- remove $O(n)$ time
  $\Rightarrow$ Dijkstra $O(m \cdot n)$

- comparator function
Repeated insertions

- Relax inserts new copies of item
- Skip outdated items

```plaintext
proc Dijkstra(V, E, δ, s)
    dist[v] = +∞ for all v ∈ V \ {s}
    dist[s] = 0
    Insert(Q, ⟨dist[s], s⟩)
    while Q ≠ ∅ do
        ⟨d, u⟩ = ExtractMin(Q)
        if d = dist[u] then
            for (u, v) ∈ E ∩ ({u} × V) do
                if dist[u] + δ(u, v) < dist[v] then
                    dist[v] = dist[u] + δ(u, v)
                    Insert(Q, ⟨dist[v], v⟩)
        return dist
```
Using a visited set

```c
proc Dijkstra_4(V, E, δ, s)
    dist[v] = +∞ for all v ∈ V \ {s}
    dist[s] = 0
    visited = ∅
    Insert(Q, ⟨dist[s], s⟩)
    while Q ≠ ∅ do
        ⟨d, u⟩ = ExtractMin(Q)
        if u ∉ visited then
            visited = visited ∪ {u}
            for (u, v) ∈ E ∩ ({u} × V) do
                if dist[u] + δ(u, v) < dist[v] then
                    dist[v] = dist[u] + δ(u, v)
                    Insert(Q, ⟨dist[v], v⟩)
        return dist
```
A shaky idea...

proc Dijkstra\(_4\)(V, E, \(\delta\), s)

\[
dist[v] = +\infty \text{ for all } v \in V \setminus \{s\}
\]

\[
dist[s] = 0
\]

\(visited = \emptyset\)

Insert\((Q, \langle dist[s], s \rangle)\)

while \(Q \neq \emptyset\) do

\(\times u) = \text{ExtractMin}(Q)\)

if \(u \not\in visited\) then

\(visited = visited \cup \{u\}\)

for \((u, v) \in E \cap (\{u\} \times V)\) do

if \(dist[u] + \delta(u, v) < dist[v]\) then

\(dist[v] = dist[u] + \delta(u, v)\)

Insert\((Q, \langle dist[v], v \rangle)\)

return \(dist\)

- \(Q\) only store nodes (save space)
- Comparator
- Key = current distance \(dist\)

Heap invariants break
Experimental study

- Implemented Dijkstra in Python
- Stress test on random cliques
- Binary heaps  \[ \text{failed (default priority queue in Java and Python)} \]
- Skew heaps  \[ \text{worked} \]
- Leftist heaps  \[ \text{worked} \]
- Pairing heaps  \[ \text{worked} \]
- Binomial queues  \[ \text{worked} \]
- Post-order heaps  \[ \text{worked} \]
- Binary heaps with top-down insertions  \[ \text{worked} \]
Binary heap insertions – bottom-up vs top-down

Insert(7)
Binary heaps using \textit{dist} in a comparator fails

- outdated
- wrong placement
- not smallest key
- ignored since visited
Definition

Priority Queues supporting Decreasing Keys

- Items = \langle \text{key, value} \rangle
- Over time keys can decrease – *priority queue is not informed*
- Items are compared w.r.t. their *current keys*
- The *original key* of an item is the key when it was inserted

**Insert**(item)

**ExtractMin()** returns an item with *current key less than or equal to all original keys* in the priority queue
**Theorem 1**

$\text{Dijkstra}_{4}$ correctly computes shortest paths when using $\text{dist}$ as current key and a priority queue supporting decreasing keys.

**Theorem 2**

The following priority queues support decreasing keys (out of the box):

- binary heaps with top-down insertions ([Williams 1964])
- leftist heaps ([Crane 1972])
- binomial queues ([Vuillemin, 1978])
- skew heaps ([Sleator, Tarjan 1986])
- pairing heaps ([Fredman, Sedgewick, Sleator, Tarjan 1986])
- post-order heaps ([Harvey, Zatloukal 2004])
Proof of Theorem 2 - Basic idea

- Decreased heap order
  \[ u \text{ ancestor of } v \Rightarrow \text{ current key } u \leq \text{ original key } v \]

- Root valid item to extract

- Top-down merging two paths preserves decreased heap order

⇒ skew heaps and leftist heaps support decreasing keys
Experimental evaluation of various heaps

- Cliques with uniform random weights
- With decreasing keys less comparisons (outdated items removed earlier)
Reduction in comparisons

comparisons decreasing keys / comparisons (key, value) pairs
- **Postorder heap** [Harvey and Zatloukal, FUN 2004]

- Insert **amortized O(1)**, ExtractMin **amortized O(log n)**
- Implicit (space efficient)
- Best implicit comparison performance (and good time performance)
Conclusion

- Introduced notion of priority queues with decreasing keys ... as an approach to deal with outdated items in Dijkstra’s algorithm
- Experiments identified priority queues supporting decreasing keys ... just had to prove it
- Built-in priority queues in Java and Python are binary heaps ... do not support decreasing keys
- Binary heaps with top-down insertions do support decreasing keys ... and also
  skew heaps, leftist heaps, pairing heaps, binomial queues, post-order heaps