**Breadth First Search → Dijkstra**

\[ R = \emptyset \cup \{ s \} \]
\[ Q = \emptyset \cup \{ s \} \]

while Q:
  \[ u = Q . \text{get}() \]
  for v in Adj[u]:
    if v not in R:
      R.add(v)
      Q.add(v)

we want:
- guaranteed shortest paths
- edges have non-negative weight
- runtime: examine each node/edge once.

\[ O(VA + EB) \]
- A: vertex-exam time
- B: edge-exam time

⇒ make Q a priority queue!

**The natural extension!**

- guaranteed shortest paths (edges all have weight 1)
- runtime: vertex added once
  
  - edge examined once

  \[ O(V + E) \]

- Q is **FIFO**

\[ d = [\infty \times [V]] \]
\[ d[s] = 0 \]
\[ Q = \text{build}(\text{size}=|V|) \]
while (Q):

  \[ u = Q . \text{get min}() \]
  for v in Adj[u]:
    if \[ d[v] > d[u] + w(u,v) \]:
      \[ d[v] = d[u] + w(u,v) \]
      Q. decrease_key(v, d[v])

**differences:**
- Priority Q
- while Q loop can be \(|V|\) or \(|V| - 1\)
- v may be relaxed many times (in BFS only once).
- runtime: \( A = \text{extract-min} \)
  \( B = \text{decrease-key} \)

**notes:**
- \( R \) from BFS: now the set of 'done' nodes, i.e. \( u \in R \Rightarrow d[u] = S(u) \)

(check out Fibonacci heaps!)
Correctness.

Suppose $u$ added to $S$ and $d[u] \neq \delta(su)$ and first violation.

Then $d[y] = \delta(sy)$ when $u$ added.

$d[x] = \delta(sx)$ b/c $u$ is first violation.
(x,y) relaxed at that time.
and $y$ on a shortest path to $u$ so is also a shortest.

$\delta(sy) \leq \delta(su)$

$d[y] = \delta(sy) \leq \delta(su) \leq d[u].$

but $u$ added first $\iff$

Even further improvements.

A* & heuristic functions: pairwise shortest paths.

Idea: sometimes we have additional info that can help.

2-way Dijkstra:

What if?

Heuristic function: (also called potential)

$h(v) \leq \min [w(p(v, end))]$ over all possible paths.

In geometry euclidean: use dist between points.

Each vertex is a point in space.

A*: add [edge weight + $h(v)$] as new key.
It's Halloween and you're a very practical trick-or-treater. You've found a map of the neighborhood and have labeled each house by their likely candy generosity.

example: 

<table>
<thead>
<tr>
<th>Home</th>
<th>2</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

you've decided that if you can't get a piece of candy for every 0.1 miles you walk, you won't go trick or treating.

- how do you find out if you will go trick-or-treating efficiently?
  - remember: once you've visited a house it will no longer give you candy.
  - hint: can you make a new graph that is easier to work with?

you realize that at the end of the day you should end up back home. how does this alter your computation?