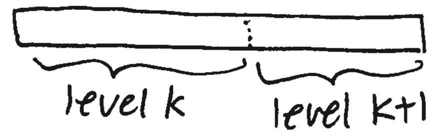
# Recitation 17

Breadth First Search → Dijkstra  The natural extension!

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

- guaranteed shortest paths (edges all have weight 1)
- runtime: vertex <u>added</u> once
  edge <u>examined</u> once ↑ const time
  O(V+E)
- Q is FIFO




⬇

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

  O(VA + EB)
  A: vertex exam time
  B: edge exam time

→ make Q a priority Queue!

```
d = [∞] * |v|
d[s] = 0
Q = build(size=|v|)
while (Q):
    u = Q.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
```

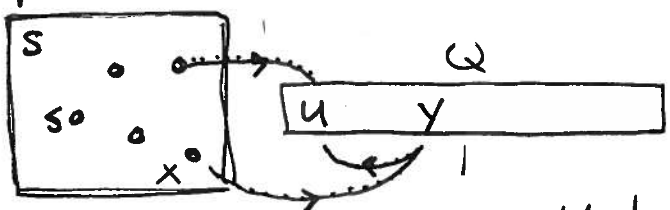
— differences:
- Priority Q
- while Q loop – can be |V| or |V|-1
- v may be relaxed many times (in BFS only once).
- runtime A = extract_min
  B = decrease_key
  } depends on implementation.
  heaps: A = log(V)
  B = log(V)

— notes:
- R from BFS: now the set of 'done' nodes. i.e. $u \in R \Rightarrow d[u] = \delta(s,u$

(check out Fibonacci heaps!)

> Correctness.

suppose $u$ added to $S$ and $d[u] \neq \delta(s\,u)$ and FIRST violation.




then $d[y] = \delta(s\,y)$ when $u$ added.

$d[x] = \delta(s\,x)$ 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.

$\delta(s\,y) \leq \delta(s\,u)$

$d[y] = \delta(s\,y) \leq \delta(s\,u) \leq d[u]$.

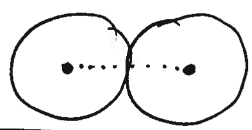
but $u$ relaxed first $\Rightarrow\Leftarrow$

> Even further improvements.

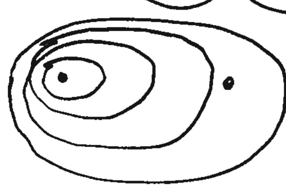
A* & hueristic functions: pairwise shortest paths.

idea: sometimes we have additional info that can help.

2-way Dijkstra:



what if?



hueristic function: (also called potential)

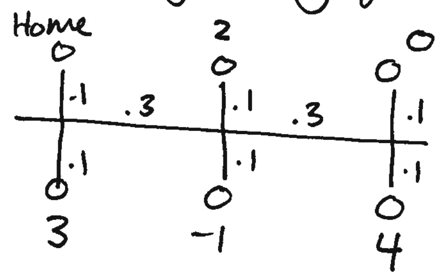
$h(v)$ [redacted] $\leq \min \left[ w(p(v, end)) \right]$ over all possible paths.

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.

ex:




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 w/?

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