SSSP (Dijkstra) - assumption: no negative length edges.

Idea: explore outward from source

Repeatedly expand a circle about s so as to find all vertices within distance s.

Example:

Note: \( l(v, w) \) is length of edge \( (v, w) \)
One way to think about this:

The edges are fuses that burn at the same constant rate (one lb).

Start a fire at the source $s$.

This lights the fuse exiting $s$.

When a fuse burns through it into its far endpoint, on fire which lights the fuse exiting $w$.

The time a vertex catches fire is its shortest distance from $s$.
- For the shortest path of length $d$, will burn in time $d$.
- No other path, necessarily of length $>d$, can burn in time $<d$.

To model this process, partition the vertices into 3 sets:

- On Fire
- Endangered - vertices with a burning incoming fuse
- Still Safe - the rest
Core of the algorithm:

**Repeat**

1. Let \( v \) be the next vertex to ignite;
2. \( \text{IT}[v] \leftarrow v \text{'s ignition time}; \)
3. For each edge \((v, w)\) do
   - Light the fuse \((v, w)\);
   - If we still safe then move \( w \) to \( \text{Endangered} \)

**Until** \( (\text{Endangered} = \emptyset) \).

**Getting started:**

- Add \( s \) to \( \text{Endangered} \).
- \( \text{IT}[s] \) will be \( 0 \).

**How do we figure out \( w \text{'s ignition time} \)?**

- Maintain a current (over) estimate of \( w \text{'s ignition time} \):

  - \( w \) added to \( \text{Endangered} \):
    - \( \text{EIT}[cw] \leftarrow \text{IT}[v] + l(v, w) \)
  - \( w \) \& \( \text{Endangered} \):
    - \( \text{EIT}[cw] \leftarrow \min \{ \text{EIT}[cw], \text{IT}[v] + l(v, w) \} \)

---

[Diagram of a graph with a vertex \( v \) on fire at time \( \text{IT}[v] \), with an edge \((v, w)\) and a fuse \((v, w)\).]
Which is the next vertex to ignite?

The one with smallest \( EIT[v] \) among all \( v \in \text{Endangered} \).

Why? Because the next vertex to ignite must be at the end of some currently burning fuse and the lengths of the paths from start to vertices reflected in \( EIT[v] \).

Want a data structure to store pairs \((v, EIT[v])\):

- \( \text{DeleteMin} (Q) \) returns \( v \) with smallest \( EIT[v] \)
- \( \text{DeleteKey} (v, h') \) sets \( EIT[v] \) to \( h' \) and guarantees \( EIT[v] \geq h' \)
- \( \text{Insert} (v, h, q) \)

Called a priority queue. (can have DeleteMin, DeleteMax versions).

Further simplifications:

- Set \( EIT[v] = \infty \) for all \( v \in \text{Safe} \).
- \( Q \) will store the vertices in \( \text{Safe} \cup \text{Endangered} \), so no need to track these sets explicitly.

Note that \( J[v] = EIT[v] \) at ignite time and \( EIT[v] \) cannot be reduced further, so no need for separate \( J \) array.
Initialize:

\[
\text{for } v \in V - \{s, t\} \text{ do } EIT(v) = \infty \text{ end;}
\]
\[
EIT(s) = 0;
\]
\[
\text{Insert } (v, EIT(v)) \text{ in } Q \text{ for all } v \in V;
\]

repeat

\[
v \leftarrow \text{DeleteMin}(Q); \quad (v \text{ is the next vertex to visit})
\]

for each edge \((v, w)\) do

\[
\text{if } EIT(w) > EIT(v) + l(v, w) \quad \text{then do}
\]

\[
EIT(w) \leftarrow EIT(v) + l(v, w);
\]

\[
\text{Redeclare } (w, EIT(w));
\]

end

until \((Q = \emptyset \text{ or minkey in } Q \text{ is } \infty)\)

Runtime

**Initialization** \(Q : O(V)\)

\(\leq V \text{ delete min} : O(V \log V)\)

\(\leq E \text{ Redeclare} : O(E \log V) \quad \text{(can reduce to } O(E)).\)

\(O(V + E) \text{ after operations.}\)

**Total runtime**: \(O((E + V) \log V)\).
Priority Queue Implementation

Use a min heap.

Reduction implementation.

\[ Q \rightarrow \begin{array}{c} v3 \ 4 \\ \end{array} \]

\[ \begin{array}{c} v1 \ 7 \\ v6 \ 11 \\ v3 \ 4 \\ \end{array} \]

\[ \begin{array}{c} v2 \ 8 \\ v4 \ 14 \\ v5 \ 9 \\ v1 \ 7 \\ v2 \ 8 \end{array} \]

Reduce key \((v4, 6)\):

\[ Q \rightarrow \begin{array}{c} v3 \ 4 \\ \end{array} \]

\[ \begin{array}{c} v1 \ 7 \\ v6 \ 11 \\ v3 \ 4 \\ v4 \ 6 \\ v2 \ 8 \end{array} \]

\[ v5 \ 9 \]

\[ O(\log v) \] cost if \( v \) items in heap.
How does one find $v_4$ (or any other vertex) quickly?

Cross-link nodes in heap with vertex array.

Key values will be stored with the vertices.

Entries in heap are indices of corresponding vertices in the array indexed by vertex number.

Remember, pointers in heap are implicit.

When a node $p$ moves nodes in heap, i.e., changes entries, e.g., for node $p$, need to do corresponding update to vertex array:

$p\_\text{uvalue.\_link} \gets p$.

Really nice.
Output Shortest Paths

Ekham's Dijsktra's algorithm:

We define the choice vertex, Prev[v], here.

Initialization: Prev[v] = nil for all v

Modify the algorithm as follows:

if EIT[v] > EIT[u] + l(u,v)
    then do
        Reduce key (u, EIT[v] + l(u,v));
        Prev[v] = u
    end

SPath(v); (* Prints a shortest path to v *)
if v = s then Print(s)
else if Prev[v] = nil then Print (no path from s to v)
else do
    SPath(Prev[v]); (* Prints a shortest path to Prev[v] *)
    Print(v)
end