Graph Search

A problem: We have a graph $G = (V, E)$ and a node $s \in V$.

- Write $u \rightarrow v$ if node u is reachable in one step. "v is a successor of u".
  - $*$ If the graph is directed $u \rightarrow v$ means $u = \overleftarrow{e}$ and $v = \overrightarrow{e}$ for some $e \in E$
  - $*$ If the graph is undirected $u \rightarrow v$ means $\{u, v\} = \overleftrightarrow{e}$ for some $e \in E$
- Write $u \xrightarrow{*} v$ to mean there is a path from u to v, i.e. v **is reachable from** u.
  - $*$ I.e. there is a sequence of one or more nodes $[v_0, v_1, ..., v_n]$ such that

$$u = v_0 \rightarrow v_1 \rightarrow ... \rightarrow v_n = v$$

Reachability: Given a graph G and a node s, find all nodes reachable from s.

Use the following colour scheme

- H Black nodes. Found and processed. (**H**andled)
- W Grey nodes. Found but not processed yet. (**W**ork set)
- White nodes. Not yet found.

The 'flood' strategy.

- Colour s grey and all other nodes white.
- Until there are no grey nodes:
  - $*$ Pick a grey node u.
  - $*$ Colour u black.
  - $*$ Colour all of u's white successors grey.

When there are no more grey nodes, all black nodes are reachable and all white nodes are not.

Invariants:

- LI1: All black or grey nodes are reachable.
- LI2: All successors of a black node are black or grey.
- LI3: s is black or grey.

If LI2, and LI3 are true and, furthermore, no node is grey, then all nodes reachable from s must be black.

If LI1, LI2, and LI3 are true and no node is grey, the black nodes are exactly the nodes reachable from s.

The flood algorithm for reachability

Inputs: a graph $G = (V, E)$ and a node s

Output: a set $H \subseteq V$

Precondition $s \in V$

Postcondition $H = \{v \in V \mid s \xrightarrow{*} v\}$

$H := \emptyset$ // Handled (black) nodes

var $W := \{s\}$ // Work set (grey nodes)

invariant

- LI1: All nonwhite nodes are reachable:
$$\forall v \in H \cup W \cdot s \xrightarrow{*} v$$
- LI2: If u is black, all its successors have been found: $\forall u \in H, v \in V \cdot (u \to v) \Rightarrow (v \in H \cup W)$
- LI3: s is grey or black: $s \in H \cup W$
- LI4: $H \cap W = \emptyset$

while $W \neq \emptyset$ do
    val $u \in W$ // let u be any value in W
    $W := W - \{u\}$; $H := H \cup \{u\}$
    for $v \mid u \to v$ do
        if $v \notin H \cup W$ then $W := W \cup \{v\}$ end if
    end for
end while

Does it work?

Recall the invariant is

- LI1: All nodes found are reachable $\forall v \in H \cup W \cdot s \xrightarrow{*} v$
- LI2: If u has been handled, all its successors have been found $\forall u \in H, v \in V \cdot (u \to v) \Rightarrow (v \in H \cup W)$
- LI3: s is grey or black $s \in H \cup W$.
- LI4: $H \cap W = \emptyset$

We need to show:

- Termination: $|V| - |H|$ is a variant.
- The invariant is established: Exercise.
- The invariant is preserved: Exercise
- The postcondition $H = \{v \in V \mid s \xrightarrow{*} v\}$ is established by the loop terminating:
  - ∗ From LI1 and $W = \emptyset$, $\forall v \in H \cdot s \xrightarrow{*} v$ and so $H \subseteq \{v \in V \mid s \xrightarrow{*} v\}$
  - ∗ It remains to show $\{v \in V \mid s \xrightarrow{*} v\} \subseteq H$.
    - · Let v be any reachable node $s \xrightarrow{*} v$
    - · So, there is a path $s = v_0 \to v_1 \to ... \to v_n = v$
    - · By LI3 and $W = \emptyset$, s is in H.
    - · By LI2 and $W = \emptyset$, $\forall u \in H, v \in V \cdot (u \to v) \Rightarrow v \in H$
    - · So, by induction, each v_i is in H and $v \in H$ QED

Leaving a trail of bread crumbs

We will mark each node reached with the node that was used to reach it.

LI5: For any black or grey node u there is a path from s,

$$s = \underbrace{\pi(...\pi(}_{\geq 0}u)...) \to ... \to \pi(\pi(u)) \to \pi(u) \to u$$

all nodes of which, apart from possibly the last, are black.

Use a function valued state variable $\pi : V \to V \cup \{\text{null}\}$ (π for πarent). When a node turns grey, update π

The flood algorithm for reachability with paths

for $v \leftarrow V$ do $\pi(v) := \text{null}$ end for
$H := \emptyset$ var $W := \{s\}$
{ Inv: LI1and LI2 and LI3 and L4 and LI5 }
while $W \neq \emptyset$ do
  val $u \in W$ // let u be any value in W
  $W := W - \{u\}$ $H := H \cup \{u\}$
  for $v \mid u \to v$ do
    if $v \notin H \cup W$ then $W := W \cup \{v\}$; $\pi(v) := u$ end if
  end for end while

The π function defines a tree with s at its root.

It has the result of classifying each reachable edge as a

- Tree edge. Tree edges form a tree defined by π
- Back edge. From descendant to ancestor.
- Forward edge. From ancestor to descendant. (Other than tree edges.)
- Cross edge. All others

Tracking the colour

To make expressions like $v \notin H \cup W$ faster, we can track the colour of each node with an array colour with a linking invariant that, for all $v \in V$,

$$\begin{aligned} & (\text{colour}(v) = \text{grey} \Leftrightarrow v \in W) \\ \wedge\ & (\text{colour}(v) = \text{black} \Leftrightarrow v \in H) \\ \wedge\ & (\text{colour}(v) = \text{white} \Leftrightarrow v \notin H \cup W) \end{aligned}$$

H is no longer needed.

W is still useful for finding the next node to process.

- LI1: $\forall v \cdot \text{colour}(v) \in \{\text{grey}, \text{black}\} \Rightarrow s \xrightarrow{*} v$
- LI2: $\forall u, v \cdot\ \text{colour}(u) = \text{black} \wedge (u \rightarrow v)$
  $\Rightarrow \text{colour}(v) \in \{\text{grey}, \text{black}\}$
- LI3: $\text{colour}(s) \in \{\text{grey}, \text{black}\}$
- LI4: $\forall v \cdot \text{colour}(v) = \text{grey} \Leftrightarrow v \in W$

The flood algorithm for reachability with colour array

for $v \leftarrow V$ do $\pi(v) := \text{null}$ $\text{colour}(v) := \text{white}$ end for
var $W := \{s\}$
$\text{colour}(s) := \text{grey}$
{ Inv: LI1and LI2 and LI3 and L4 and LI5 }
while $W \neq \emptyset$ do
    val $u \in W$ // let u be any value in W
    $W := W - \{u\}$
    $\text{colour}(u) := \text{black}$
    for $v \mid u \rightarrow v$ do
        if $\text{colour}(v) = \text{white}$ then
            $W := W \cup \{v\}$; $\text{colour}(v) := \text{grey}$
            $\pi(v) := u$ end if end for end while

Data refining W

We can keep track of the set of grey nodes with any kind of collection data structure: Set, FIFO queue, stack.

A FIFO queue Q

Replace W with a FIFO queue Q

LI4 becomes $\forall v \cdot \text{colour}(v) = \text{grey} \;\Leftrightarrow\; Q.\text{contains}(v)$

Nodes are visited in a "breadth" first order.

Nodes closer to s are handled earlier.

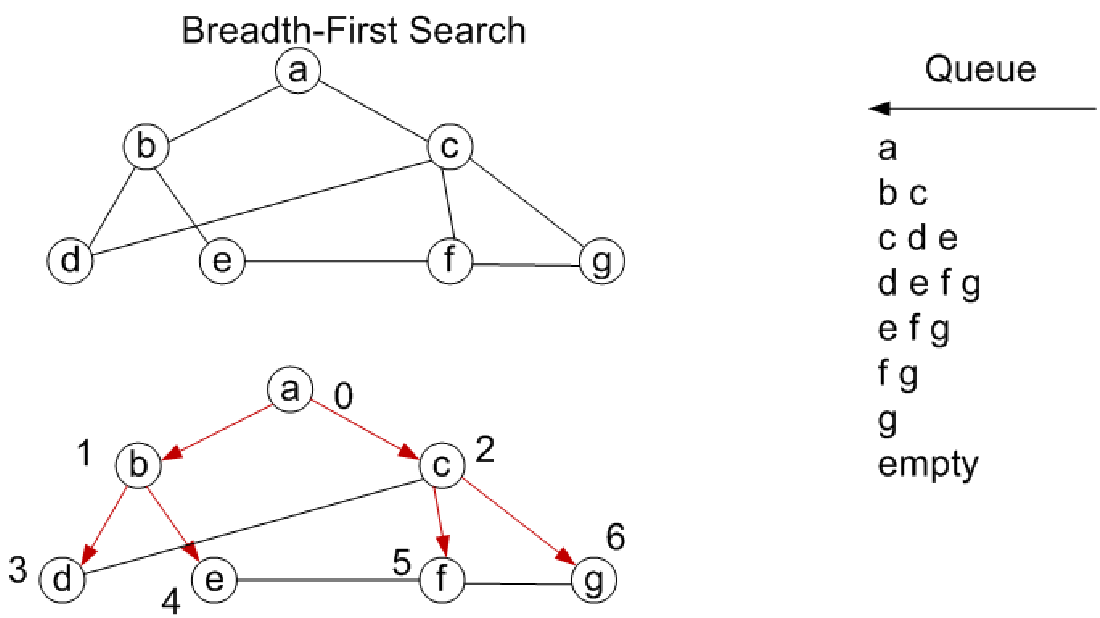
Each path found has as few edges as possible.

Breadth first search

for $v \leftarrow V$ do $\pi(v) := \text{null}$ $\text{colour}(v) := \text{white}$ end for
var $Q : \text{Queue} := \text{new Queue}$
$Q.\text{add}(s)$
$\text{colour}(s) := \text{grey}$
{ Inv: LI1and LI2 and LI3 and L4 and LI5 }
while $\neg Q.\text{isEmpty}$ do
  val $u := Q.\text{remove}()$
  $\text{colour}(u) := \text{black}$
  for $v \mid u \rightarrow v$ do
    if $\text{colour}(v) = \text{white}$ then
      $Q.\text{add}(v)$; $\text{colour}(v) := \text{grey}$
      $\pi(v) := u$
    end if
  end for
end while

Efficiency

At this point, we can see that, if we can quickly find the successors of a node, then processing each edge is $\Theta(1)$. Each edge is processed twice. Hence $\Theta(|V| + |E|)$. An adjacency list representation for the graph will do the trick.



A LIFO stack S

LI4 becomes $\forall v \cdot \text{colour}(v) = \text{grey} \;\Leftrightarrow\; S.\text{contains}(v)$

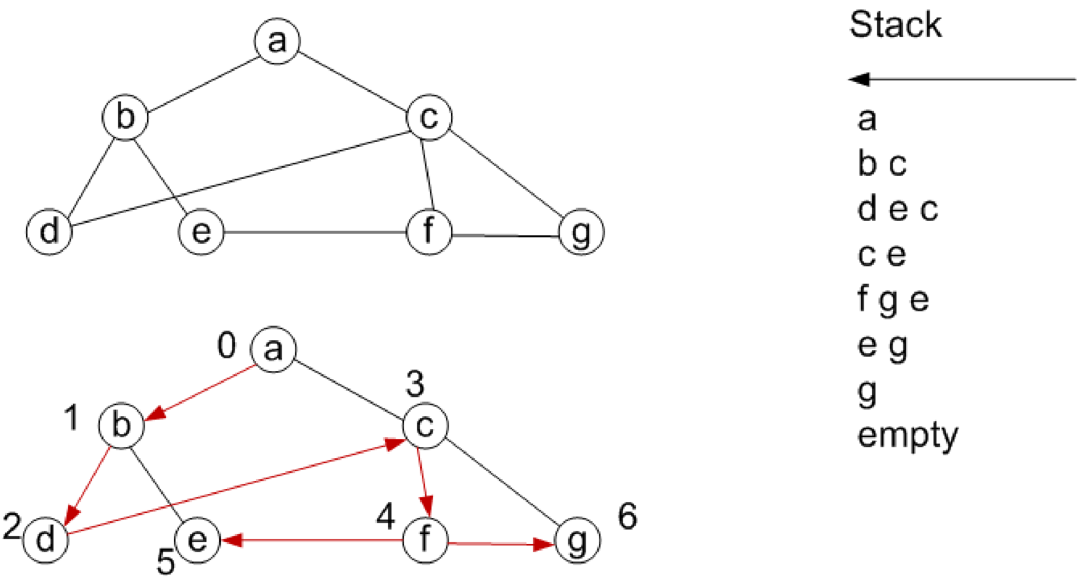
If a grey node is found a second (etc) time, it is moved to the top of the stack.

Depth-first search

for $v \leftarrow V$ do $\pi(v) := \text{null} \quad \text{colour}(v) := \text{white}$ end for
var $S : \text{Stack} := \text{new Stack}$
$S.\text{push}(s)$
$\text{colour}(s) := \text{grey}$
{ Inv: LI1and LI2 and LI3 and L4 and LI5 }
while $\neg S.\text{isEmpty}$ do
  val $u := S.\text{pop}()$
  $\text{colour}(u) := \text{black}$
  for $v \mid u \rightarrow v$ do
    if $\text{colour}(v) \neq \text{black}$ then // Note change!
      if $\text{colour}(v) = \text{grey}$ then
          // Move v to the top of the stack.
          $S.\text{remove}(v)$ end if
      $S.\text{push}(v)$; $\text{colour}(v) := \text{grey}$
      $\pi(v) := u$ // If v is grey, overwrites earlier assignment!
    end if
  end for
end while

We need to implement the stack so that an arbirary node can be removed in constant time. A doubly-linked list implemented with arrays will do it.

This is a depth-first search. It follows paths leading away from s as far as possible before backtracking to find other paths.



Dijkstra's algorithm

Let's revisit the breadth first search

Breadth first search

for $v \leftarrow V$ do $\pi(v) := \text{null colour}(v) := \text{white}$ **end for**

var $Q : \text{Queue} := \text{new Queue}$

$Q.\text{add}(s)$; $\text{colour}(s) := \text{grey}$

{ Inv: LI1and LI2 and LI3 and L4 and LI5 }

while $\neg Q.\text{isEmpty}$ do

  val $u := Q.\text{remove}()$

  $\text{colour}(u) := \text{black}$

  for $v \mid u \rightarrow v$ do

    if $\text{colour}(v) = \text{white}$ then

      $Q.\text{add}(v)$; $\text{colour}(v) := \text{grey}$

      $\pi(v) := u$ end if end for end while

This finds the shortest path from s to each reachable node, counting each edge as costing 1.

Suppose that each edge e is associated with a nonnegative distance $w(\overleftarrow{e}, \overrightarrow{e})$.

We want to find the *shortest path* from s to each reachable node.

Applications are ubiquitous, e.g. in robotics, navigation, and planning.

Let $t(u)$ be the length of the shortest path from s to u.

$$t(u) = \min_{p \mid s \xrightarrow{p} u} \text{distance}(p)$$

where $s \xrightarrow{p} u$ means that p is a path from s to u and

$$\text{distance}([u_0, e_0, u_1, e_1, ..., e_{n-1}, u_n]) = \Sigma_{i \in \{0,..n\}} w(u_i, u_{i+1})$$

Use array item $d(v)$ to track the distance of the shortest path from s to v handled so far. (I.e., that either consists of all black nodes, or is all black except for the final item.)

Since we stop as soon as all reachable nodes are black, we need

- DI1: For each black node, v, $d(v) = t(v)$.

To ensure that the grey node with the smallest d value also has the true distance, we need

- DI2: For each grey node, v, $d(v)$ is the distance of some path from s to v.

We data refine W with a priority queue PQ.

- A priority queue associates each item with a priority value.
- $PQ.\mathrm{add}(v, x)$ adds node v with priority x or updates the priority of v to x.
- $PQ.\mathrm{removeLeast}()$ removes and returns a node with the lowest priority.

Invariants about PQ

- LI4: $\forall v \cdot \mathrm{colour}(v) = \mathrm{grey} \Leftrightarrow PQ.\mathrm{contains}(v)$
- DI3: The priority of each node v on PQ is $d(v)$.

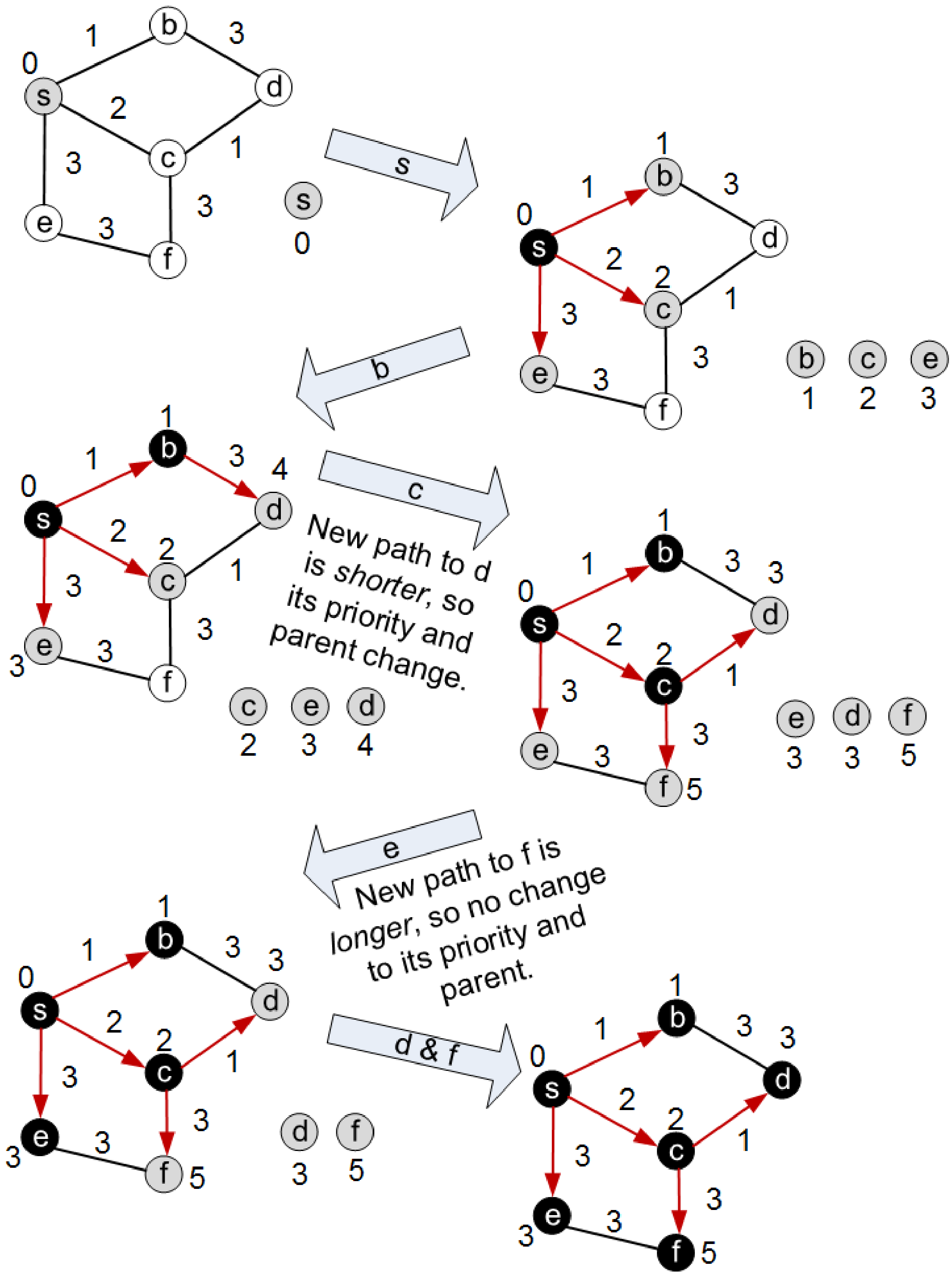
- DI1: For each black node, v, $d(v) = t(v)$.
- DI2: For each grey node, v, $d(v)$ is the distance of some path from s to v.
- DI3: The priority of each node v on PQ is $d(v)$.

As with DFS, grey nodes may be found more than once, so we might need to improve a $d(v)$

Dijkstra's algorithm

for $v \leftarrow V$ do
    $\pi(v) := \text{null}$ $\text{colour}(v) := \text{white}$ $d(v) := \infty$
end for
var $PQ : \text{PriorityQueue} := \text{new PriorityQueue}$
$PQ.\text{add}(s, 0)$
$\text{colour}(s) := \text{grey}$ $d(s) := 0$
{ Inv: LI1 and ... and LI5 and DI1 and DI2 and DI3 }
while $\neg PQ.\text{isEmpty}$ do
    val $u := PQ.\text{removeLeast}()$
    { u has the smallest d value of all grey nodes }
    $\text{colour}(u) := \text{black}$
    for $v \mid u \rightarrow v$ do
        if $d(v) > d(u) + w(u, v)$ then
            { v is not black, by DI1 }
            $d(v) := d(u) + w(u, v)$
            $PQ.\text{add}(v, d(v))$; $\text{colour}(v) := \text{grey}$
            $\pi(v) := u$
        end if end for end while

Note: when $PQ.\text{add}(v, d(v))$ is executed, v may already be on the queue (grey). In this case, its priority is updated.


s
b
c
New path to d is shorter, so its priority and parent change.
e
New path to f is longer, so no change to its priority and parent.
d & f

We need to see that the invariants are preserved.

- DI1: For each black node, v, $d(v) = t(v)$.
- DI2: For each grey node, v, $d(v)$ is the distance of some path from s to v.

Lemma: If DI1 and DI2 hold, then, for any w and any optimal path from s to w, the first grey node v on the path (if any) has $d(v) = t(v)$.

Proof. Let v be grey and the first grey node on some optimal path. If v is s, then $d(v) = 0 = t(s)$.

If v is not s, then s is black. Since the successor of a black node must be black or grey, the first grey node on any path starting at s will be preceded by a black node.

Let u be the predecesor of v on the path, as u is black, by DI1 $d(u) = t(u)$.

Furthermore, when u was visited, the edge from u to v would have been considered and so $d(v) \leq t(u)+w(u,v)$.

By DI2, $d(v) \geq t(v)$, so $d(v) = t(u) + w(u,v)$, and, since (u,v) is on an optimal path, $t(v) = t(u) + w(u,v)$.

So $d(v) = t(v)$.

DI1 is preserved. Suppose that DI1 and DI2 hold, but, at line $\mathrm{colour}(u) := \mathrm{black}$, u does not have a "true value" ($t(u) < d(u)$) i.e. DI1 is about to be broken.

Then there is an optimal path p' from u to s with a shorter distance than $d(u)$.

Consider the first grey node u' on this optimal path. By the lemma, it must be that $d(u') = t(u')$

Since $u' = u$ or u' is before u on an optimal path, $t(u') \leq t(u)$. Altogether

$$d(u') = t(u') \leq t(u) < d(u)$$

But this is impossible since u' would have priority over u and u wouldn't have been picked on the previous line.

DI2 is preserved. Exercise.

Implementation note

The colour array is no longer being used. We can demote it to a ghost variable.

Animation

See the Algorithms Animated site.

Other algorithms

There are many other algorithms for finding shortest paths. E.g. the Bellman-Ford algorithm and Floyd's algorithm.

Other problems

We can replace $>$ and + with other suitable operators.

E.g. If weights represent (independent) probabilities of success, replace

$>$ with $<$,
$+$ with $\times$,
0 with 1, and
∞ with 0

to find the most reliable path.

Efficiency

Assume the priority queue operations add and remove can be done in $\Theta(\log n)$ time where n is the size of the queue:

- We may need $\Theta(|V|)$ items on the queue, so the algorithm takes $\Theta(|E| \times \log |V|)$ time.

Dijkstra's algorithm has the property that it can be modified to print out all the nodes in order of their distance from s. Can you show that any shortest path algorithm with this property takes $\Omega(|E| \times \log |V|)$ time?

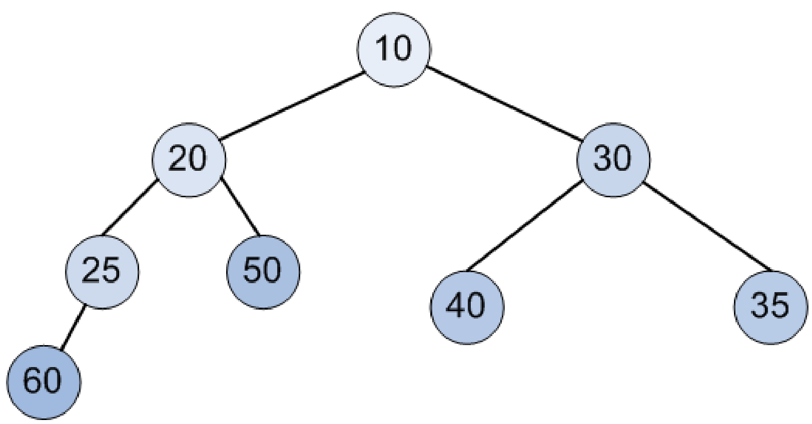
Priority Queue Representation

An efficient priority queue can be built from a balanced heap.

- A heap is a labelled binary tree in which each node is labeled with a data item and a priority
- The priority of each parent is less than or equal to the priority of its children.
- We store the items and priorities in the first n items of an array a. Invariant: $\forall i \in \{0, ..n\} \cdot$

$$(\mathit{leftExists}(i) \quad \Rightarrow \quad a(i).\text{priority} \leq a(\mathit{left}(i)).\text{priority})$$
$$\wedge \ (\mathit{rightExists}(i) \quad \Rightarrow \quad a(i).\text{priority} \leq a(\mathit{right}(i)).\text{priority})$$

- E.g.,



- Note: Only the priorities are shown in the pictures.
- We also need a function mapping each item to its location in a. If each item is represented by a unique small number in $\{0, ..m\}$, we can use an array loc so that

$$\forall i \in \{0, ..n\} \cdot loc(a(i).\text{item.number}) = i$$
$$\forall j \in \{0, ..m\} \cdot loc(j) = -1 \vee a(loc(j)).\text{item.number} = j$$

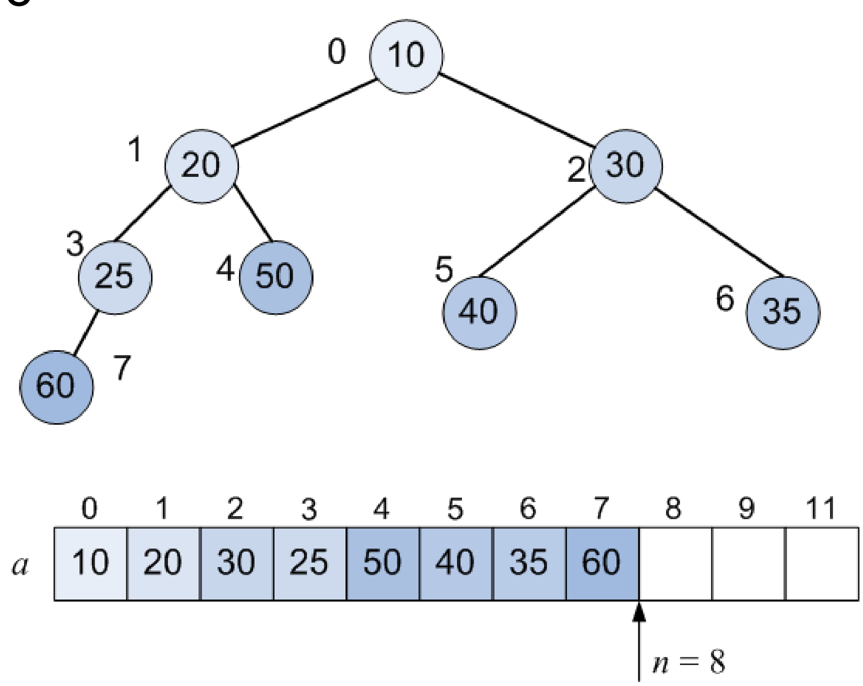
Array representation

We can build a balanced heap of size n by using the first n items of an array a.

- Use breadth-first numbering.
- The root is at location 0.
- Invariant: $\forall i \in \{0, ..n\} \cdot$

$$\begin{array}{ll} & \mathit{left}(i) = 2i + 1 \\ \wedge & \mathit{right}(i) = 2i + 2 \\ \wedge & \mathit{leftExists}(i) = (2i + 1 < n) \\ \wedge & \mathit{rightExists}(i) = (2i + 2 < n) \end{array}$$

- In a picture

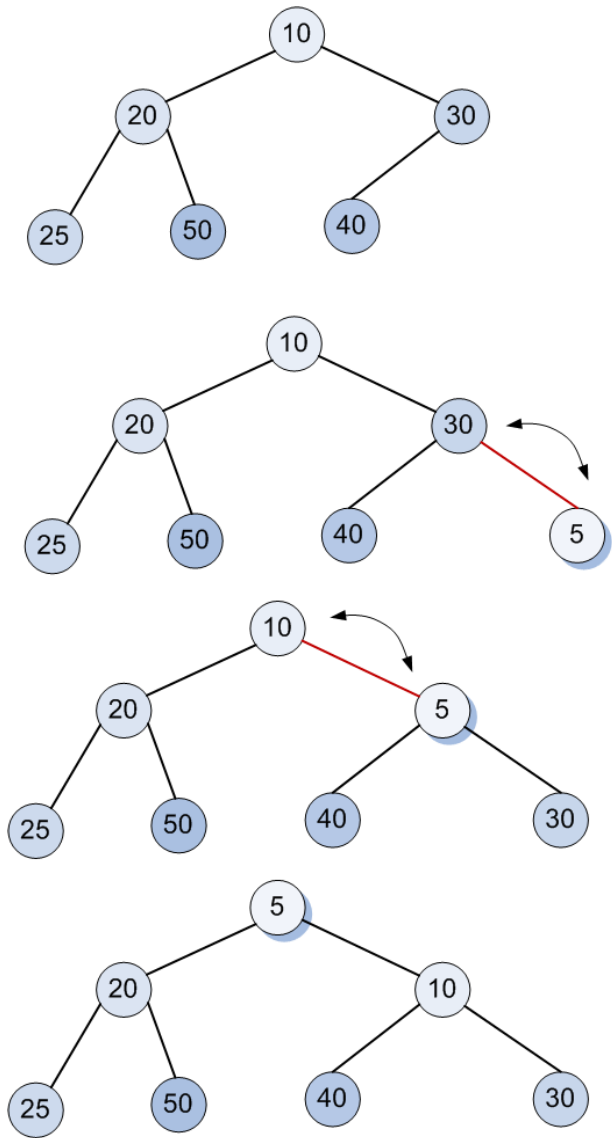


- The height (counting branches) of the tree is $\lfloor \log_2 n \rfloor$, which is in $\Theta(\log n)$

Inserting into a heap

Put the new item at $a(n)$; increment n. Then swap the element upwards until its priority is larger or equal to its parent's (or at the root) (+ corresponding changes to loc)

E.g.



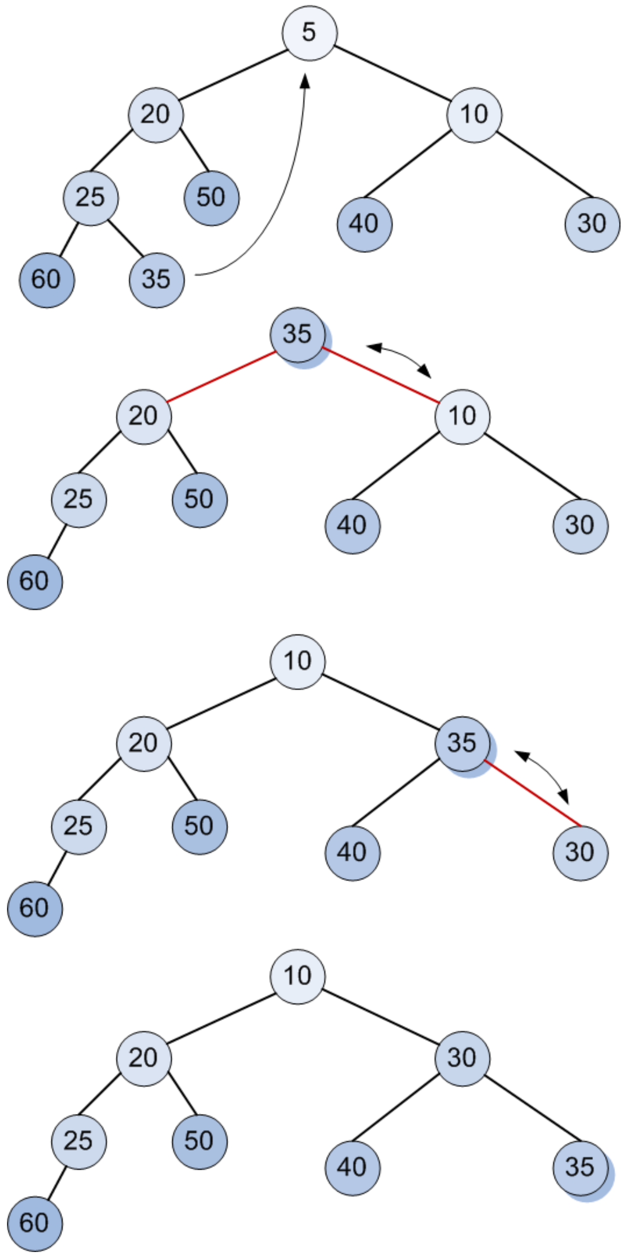
The worst case is $\Theta(\log n)$

Reducing priority of an item

Reduce the priority and then swap it upwards, just as in insert.

Removing the lowest priority item

- Decrement n; then $a(0) := a(n)$ (+ corresponding changes to loc).
- Swap the item now at the root down until its priority is less than or equal to that of all its children.
  - ∗ Swap only with a lowest child.



  - ∗ The number of swaps is limited to the height of the tree. $\Theta(\log n)$

Another application of heaps

Heap Sort:

Input: an array a such that $a.\mathrm{length} > 0$

Output: the same array

Postcondition a is a sorted permutation of a_0

var $n := 1$

inv a is a permutation of a_0 and $a[0, ..n]$ is a heap

while $n < a.\mathrm{length}$ do

floatUp(n) $n := n + 1$

end while

inv a is a permutation of a_0 and $a[0, ..n]$ is a heap.

inv $a\{0, ..n\} \leq^* a\{n, ..a.\mathrm{length}\}$

inv $a[n, ..a.\mathrm{length}]$ is sorted largest to smallest

while $n > 0$ do

$n := n - 1$ swap(a, 0, n) sinkDown(0)

end while

where

- floatUp restores the heap invariant by swapping an item upward from a leaf position
- sinkDown restores the heap invariant by swapping an item downward from the root position.

Since floatUp and sinkDown are both $\Theta(\log n)$ time (where n is the size of the heap), Heap Sort is $\Theta(n \log n)$ time (where n is the size of the array).

(No loc array is needed. We only needed it before to reduce the priority of an item.)