# 1 Breadth-First Search

*Breadth-first search* is the variant of search that is guided by a *queue*, instead of depth-first search's stack (remember, depth-first search *does* use a stack, the one implicit in its recursion). There is one stylistic difference: One does not restart breadth-first search, because breadth-first searchonly makes sense in the context of exploring the part of the graph that is reachable from a particular node ($s$ in the algorithm below). Also, although BFS does not have the wonderful and subtle properties of depth-first search, it does provide useful information of another kind: Since it tries to be "fair" in its choice of the next node, it visits nodes *in order of increasing distance from s.* In fact, our breadth-first searchalgorithm below labels each node with the shortest distance from $s$, that is, the number of edges in the shortest path from $s$ to the node. The algorithm is this:

```
Algorithm dfs(G=(V,E): graph, s:  node);
v, w:  nodes; Q: queue of nodes, initially {s};
  dist:  array[V] of integer, initially ∞
dist[s];=0
while Q is not empty do
  {v:= eject(Q),
  for all edges (v,w) out of v do
  {if dist[w] = ∞ then
    {inject(w,Q), dist[w]:=dist[v]+1}}}
```

For example, applied to the graph in Figure 1, this algorithm labels the nodes (by the array `dist`) as shown. Why are we sure that the `dist[v]` is the shortest-path distance of $v$ from $s$? It is certainly true if `dist[v]` is zero (this happens only at $s$). And, if it is true for `dist[v]` $= d$, then it can be easily shown to be true for values of dist equal to $d+1$ —any node that receives this value has an edge from a node with `dist` $d$, and from no node with lower `dist`. Notice that nodes not reachable from $s$ will not be visited or labeled.

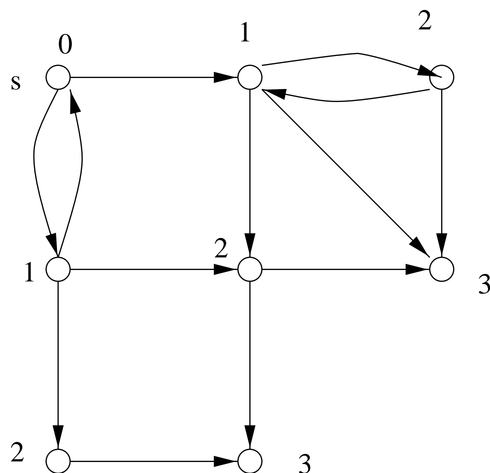


Figure 1: BFS of a directed graph

Breadth-first search runs, of course, in linear time, $O(|E|)$ (recall that we assume that $|E| \geq |V|$. The reason is the same as with depth-first search: breadth-first search visits each edge exactly once, and does a constant amount of work per edge.

## 2 Dijkstra's algorithm

What if each edge $(v, w)$ of our graph has a *length*, a positive integer denoted $\text{length}(v, w)$, and we wish to find the shortest from $s$ to all nodes reachable from it?[1] breadth-first search is still of help: We can subdivide each edge $(u, v)$ into $\text{length}(u, v)$ edges, by inserting $\text{length}(u, v) - 1$ "dummy" nodes, and then apply breadth-first search to the new graph. This algorithm solves the shortest-path problem in tom $O(\sum_{(u,v)\in E} \text{length}(u, v))$. But, of course, this can be very large —lengths could be in the thousands or millions.

This breadth-first search algorithm will most of the time visit "dummy" nodes; only occasionally will it do something truly interesting, like visit a node of the original graph. There is a way to simulate it so that we only take notice of these "interesting" steps. We need to guide breadth-first search, instead of by a queue, by a *heap* or *priority queue* of nodes. Each entry in the heap will stand for a projected future "interesting event" —breadth-first searchvisiting for the first time a node of the original graph. The priority of each node will be the projected time at which breadth-first search will reach it. These "projected events" are in general unreliable, because other future events may "move up" the true time at which breadth-first search will reach the node (see node $b$ in Figure 2). But one thing is certain: *The most imminent future scheduled event is going to happen at precisely the projected time* —because there is no intermediate event to invalidate it and "move it up." And the heap conveniently delivers this most imminent event to us.

As in all shortest path algorithms we shall see, we maintain two arrays indexed by $V$. The first array, `dist[v]`, will eventually contain the true distance of $v$ from $s$. The other array, `prev[v]`, will contain the last node before $v$ in the shortest path from $s$ to $v$. *At all times* `dist[v]` *will contain a conservative over-estimate of the true shortest distance of $v$ from $s$.* `dist[s]` is of course always 0, and all other `dist`'s are initialized to $\infty$ —the most conservative overestimate of all...

The algorithm is this:

```
algorithm Dijkstra(G=(V, E, length): graph with positive weights; s: node)
v,w: nodes, dist: array[V] of integer; prev: array[V] of nodes;
  H: heap of nodes prioritized by dist;
for all v ∈ V do {dist[v] := ∞, prev[v] :=nil}
H:={s} , dist[s] :=0
while H is not empty do
  {v := deletemin(H),
  for each edge (v,w) in E out of v do
    {if dist[w] > dist[v] + length[v,w] then
    {dist[w] := dist[v] + length[v,w], prev[w] := v, insert(w,H)}}}
```

The algorithm, run on the graph in Figure 2, will yield the following heap contents (node: dist/priority pairs) at the beginning of the while loop: $\{s : 0\}$, $\{a : 2, b : 6\}$, $\{b : 5, c : 3\}$, $\{b : 4, e : 7, f : 5\}$, $\{e : 7, f : 5, d : 6\}$, $\{e : 6, d : 6\}$, $\{e : 6\}$, $\{\}$. The final distances from $s$ are shown in Figure 2, together with the *shortest path tree from $s$*, the rooted tree defined by the pointers prev.

What is the complexity of this algorithm? The algorithm involves $|E|$ insert operations and $|V|$ deletemin operations on $H$, and so the running time depends on the implementation of the heap $H$, so let us discuss this implementation. There are many ways to implement a

[1] What if we are interested only in the shortest path from $s$ to a specific node $t$? As it turns out, all algorithms known for this problem also give us, as a free byproduct, the shortest path from $s$ to all nodes reachable from it.

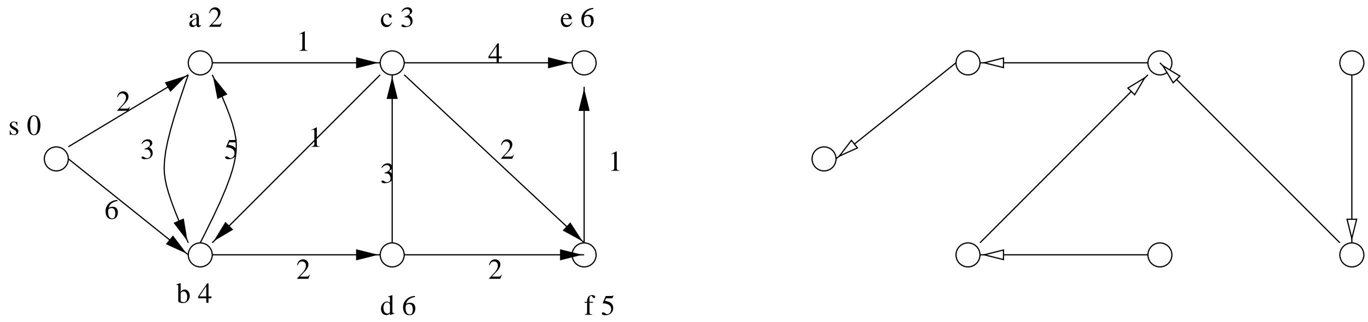


Figure 2: Shortest paths

heap.[2] Even the most unsophisticated one (an amorphous set, say an array or linked list of node/priority pairs) yields an interesting time bound, $O(n^2)$ (see first line of the table below). A binary heap gives $O(|E|\log|V|)$.

Which of the two should we use? The answer depends on how *dense* or *sparse* our graphs are. In all graphs, $|E|$ is between $|V|$ and $|V|^2$. If it is $\Omega(|V|^2)$, then we should use the lined list version. If it is anywhere below $\frac{|V|^2}{\log|V|}$, we should use binary heaps.

| heap implementation | deletemin | insert | $\|V\|\times$deletemin$+\|E\|\times$insert |
|---|---|---|---|
| linked list | $O(\|V\|)$ | $O(1)$ | $O(\|V\|^2)$ |
| binary heap | $O(\log\|V\|)$ | $O(\log\|V\|)$ | $O(\|E\|\log\|V\|)$ |
| $d$-ary heap | $O(\frac{d\log\|V\|}{\log d})$ | $O(\frac{\log\|V\|}{\log d})$ | $O((\|V\|\cdot d+\|E\|)\frac{\log\|V\|}{\log d})$ |
| Fibonacci heap | $O(\log\|V\|)$ | $O(1)$ (amortized) | $O(\|V\|\log\|V\|+\|E\|)$ |

A more sophisticated data structure, the $d$-ary heap, performs even better. A $d$-ary heap is just like a binary heap, except that the fan-out of the tree is $d$, instead of 2. In an array implementation, the child pointers of node $i$ are implemented as $d\cdot i\ldots\cdot i+d-1$, while the parent pointer as $\lfloor\frac{i}{d}\rfloor$. Since the depth of any such tree with $|V|$ nodes is $\frac{\log|V|}{\log d}$, it is easy to see that inserts take this amount of time, while deletemins take $d$ times that —because deletemins go down the tree, and must look at the children of all nodes visited.

The complexity of our algorithm is therefore a function of $d$. We must choose $d$ to minimize it. The right choice is $d=\frac{|E|}{|V|}$ —the average degree! It is easy to see that it is the right choice because it equalizes the two terms of $|E|+|V|\cdot d$. This yields an algorithm that is good for both sparse and dense graphs. For dense graphs, its complexity is $O(|V|^2)$. For sparse graphs with $|E|=O(|V|)$, it is $|V|\log|V|$. Finally, for graphs with intermediate density, such as $|E|=|V|^{1+\delta}$, where $\delta$ is the *density exponent* of the graph, the algorithm is *linear!*

The fastest known implementation of Dijkstra's algorithm uses a more sophisticated data structure called *Fibonacci heap*, which we shall not cover; see Chapter 21 of CLR. The Fibonacci heap has a separate *decrease-key* operation, and it is this operation that can be carried out in $O(1)$ *amortized* time. By "amortized time" we mean that, although each decrease-key operation may take in the worst case more than constant time, all such operations taken together have a constant average (*not* expected) cost.

[2] In all heap implementations we assume that we have an array of pointers that give, for each node, its position in the heap, if any. This allows us to always have at most one copy of each node in the heap. Each time `dist[w]` is decreased, the `insert(w,H)` operation finds $w$ in the heap, changes its priority, and possibly moves it up in the heap.

## 3 Negative lengths

Our argument of correctness of our shortest path algorithm was based on the "time metaphor:" The most imminent event cannot be invalidated, exactly because it is the most imminent. This however would not work if we had *negative edges:* If the length of edge $(b, a)$ in Figure 2 were -5, instead of 5, the first event (the arrival of BFS at node $a$ after 2 "time units") would not be suggesting the correct value of the shortest path from $s$ to $a$. Obviously, with negative lengths we need more involved algorithms, which repeatedly update the values of dist.

The basic information updated by the negative edge algorithms is the same, however. They rely on arrays `dist` and `prev`, of which `dist` is always a conservative overestimate of the true distance from $s$ (and is initialized to $\infty$ for all nodes, except for $s$ for which it is 0). The algorithms maintain `dist` so that it is always such a conservative overestimate. This is done by the same scheme as in our previous algorithm: Whenever "tension" is discovered between nodes $v$ and $w$ in that dist$(w)$ >dist$(v)$+length$(v, w)$ —that is, when it is discovered that dist$(v)$ is a more conservative overestimate than it has to be— then this "tension" is"relieved" by this code:

```
procedure update(v,w: edge)
if dist[w] > dist[w] + length[v,w] then
dist[w] := dist[v] + length[v,w], prev[w] := v
```

One crucial observation is that this procedure is *safe*, it never invalidates our "invariant" that dist is a conservative overestimate. Most shortest paths algorithms consist of many updates of the edges, performed in some clever order. For example, Dijkstra's algorithm updates each edge in the order in which the "wavefront" first reaches it. This works only when we have nonnegative lengths.

A second crucial observation is the following: Let $a \neq s$ be a node, and consider the shortest path from $s$ to $a$, say $s, v_1, v_2, \ldots, v_k = a$ for some $k$ between 1 and $|V| - 1$. If we perform update first on $(s, v_1)$, later on $(v_1, v_2)$, and so on, and finally on $(v_{k-1}, a)$, then we are sure that dist$(a)$ contains the true distance from $s$ to $a$ —and that the true shortest path is encoded in prev. We must thus find a sequence of updates that guarantee that these edges are updated in this order. We don't care if these or other edges are updated several times in between, all we need is to have a sequence of updates that contains this particular subsequence.

And there is a very easy way to guarantee this: *Update all edges $|V| - 1$ times in a row!* Here is the algorithm:

```
algorithm shortest paths(G=(V, E, length): graph with weights; s: node)
dist: array[V] of integer; prev: array[V] of nodes
for all v ∈ V do {dist[v] := ∞, prev[v] := nil}
for i := 1,...,|V|-1 do
  {for each edge (v,w) ∈ E do update[v,w]}
```

This algorithm solves the general single-source shortest path problem in $O(|V| \cdot |E|)$ time.

## 4 Negative Cycles

In fact, if the length of edge $(b, a)$ in Figure 2 were indeed changed to -5, then there would be a bigger problem with the graph of Figure 2: It would have a *negative cycle* (from $a$ to $b$ and back). On such graphs, it does not make sense to even *ask* the shortest path question. What is the shortest path from $s$ to $c$ in the modified graph? The one that goes directly from $s$ to $a$ to $c$ (cost: 3), or the one that goes from $s$ to $a$ to $b$ to $a$ to $c$ (cost: 1), or the one that takes the cycle twice (cost: -1)? And so on.

*The shortest path problem is ill-posed in graphs with negative cycles,* it makes no sense and deserves no answer. Our algorithm in the previous section works only in the absence of negative cycles. (Where did we assume no negative cycles in our correctness argument? Answer: When we asserted that a shortest path from $s$ to $a$ exists...) But it would be useful if our algorithm were able to *detect* whether there is a negative cycle in the graph, and thus to report reliably on the meaningfulness of the shortest path answers it provides.

This is easily done as follows: After the $|V| - 1$ rounds of updates of all edges, do a last round. If anything is changed during this last round of updates —if, that is, there is still "tension" in some edges— this means that there is no well-defined shortest path (because, if there were, $|V| - 1$ rounds would be enough to relieve all tension along it), and thus there is a negative cycle reachable from $s$.

## 5 Shortest paths in dags

There are two subclasses of weighted graphs that "automatically" exclude the possibility of negative cycles: Graphs with *nonnegative weights* —and we know how to handle this special case faster— and *dags* —if there are no cycles, then there are certainly no negative cycles... Here we will give a *linear* algorithm for single-source shortest paths in dags.

Our algorithm is based on the same principle: We are trying to find a sequence of updates, such that all shortest paths are its subsequences. But in a dag we know that all shortest paths from $s$ go "from left to right" in the topological order of the dag. All we have to do then is first topologically sort the dag by depth-first search, and then visit all edges coming out of nodes in the topological order:

```
algorithm shortest paths(G = (V, E, length): dag with lengths; s: node)
dist: array[V] of integer; prev: array[V] of nodes.
for all v ∈ V do {dist[v] := ∞, prev[v] := nil}
dist[s] := 0
Step 1: topologically sort G by depth-first search
for each v ∈ V in the topological order found in Step 1 do
for each edge (v,w) out of v do update(v,w)
```

This algorithm solves the general single-source shortest path problem for dag's in $O(|E|)$ time. Two more observations: Step 1 is not really needed, we could just update the edges of $G$ breadth-first. Second, since this algorithm works for any lengths, we can use it to find *longest paths in a dag:* Just make all edge lengths equal to $-1$.