Single-Source Shortest Paths

Note: This is a revised version of Handout #37, which was originally handed out on December 5.

Reading: *CLRS* Section 22.2, Chapter 24 *CLR* Section 23.2, Chapter 25

Weighted Graphs and Paths

A **weighted graph** is a graph (V, E) together with a weighting function $w : E \rightarrow Real$.

In a weighted, directed graph (G, w) the weight of a path $p = [v_0, v_1, \ldots, v_k]$ is $\sum_{i=1}^{k} w(v_{i-1}, v_i)$.

The **shortest-path weight** from a to b is $\delta(a, b) = \min\{w(p) | p \in paths(a, b)\}$.
Note that $\min\{\} = \infty$.

A **shortest path** from a to b is any path p such that $w(p) = \delta(a, b)$. (It may not be unique.)

Single-Source Shortest Paths Problem

The **single-source shortest path problem**: given a weighted directed graph $((V, E), w)$ and a source vertex s in V, find a shortest path from s to every vertex of V.

Notes:

- If a path has negative weight edges in a cycle, then a shortest path is not defined. Some algorithms (like the Dijkstra algorithm we will study) assume non-negative weights. Other algorithms (such as the Bellman-Ford algorithm, which we will not study) can handled negative weight edges as long as they don't appear in cycles.
- The problem of finding the shortest path between two particular vertices (the **single-pair shortest path problem**) may seem easier that the single-source shortest path problem, but no solution for the single-pair problem is known that is asymptotically faster than a solution for the single-source problem!
- The **all-pairs shortest path problem** finds the shortest path between every pair of vertices. We shall not study this problem this semester; see *CLRS* Chapter 25/*CLR* Chapter 26 for details.

Breadth First Search

In the simple case where all weights = 1, the single source shortest path problem is known as **breadth-first search**.

Idea: Expand a "frontier" outward from the source node s, level by level. A queue (FIFO queue, not priority queue) is used to manage the order in which the nodes are processed. The algorithm uses the following fields:

- `color[`v`]` maintains the current color of the node, which is one of:
  - white = unexplored;
  - gray = frontier node = discovered node whose edges have not been processed
  - black = fully processed = discovered node directly connected; only to other discovered nodes.
- $\mathtt{d}_s$`[`v`]` maintains the length of the shortest discovered path from source node s to v.
- `parent[`v`]` maintains the parent of node v in the breadth-first tree rooted at source node s.

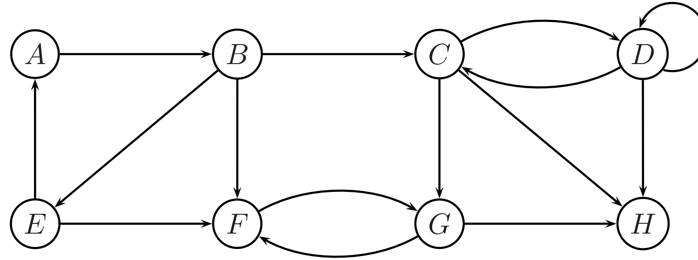
```
BFS(G,s)
  ▷ Initialization
  for v in vertices(G) do
    color[v] ← white ▷ All nodes originally unexplored
    d_s[v] ← ∞
    parent[v] ← nil
  color[s] ← gray
  d_s[s] ← 0
  Q ← Enq(s,Empty-Queue)
  ▷ Loop invariants:
  ▷   (1) Q contains only gray nodes.
  ▷   (2) d_s[v] is shortest path length from s to v for every non-white v.
  ▷   (3) (parent[v],v) is an edge in the breadth first tree rooted at s for every non-white v.
  while not Empty-Queue?(Q) do
    f ← Deq(Q)   ▷ Next frontier node to process.
    for g in Adj[f] do
      if color[g] = white then
        color[g] ← gray
        d_s[g] ← d_s[f] + 1
        parent[g] ← f
        Enq(g, Q)
    color[f] ← black   ▷ Color node black when completely processed.
```

Note that `BFS` is similar to `DFS-Visit` except that it uses a FIFO queue rather than a stack to process vertices.
Analysis:

- Each vertex enqueued and dequeued at most once at $O(1)$ time per operation: $O(V)$.
- Each edge scanned once: $O(E)$.
- Total = $O(V + E)$ (= $O(E)$ for a connected graph).

BFS Example

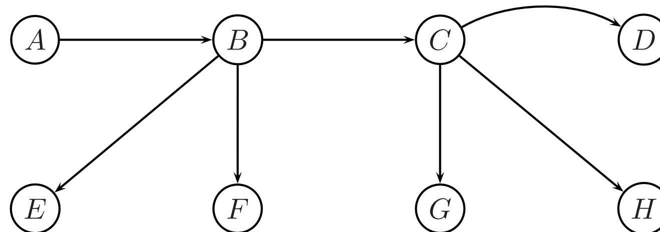
Consider the following graph:



The following is a trace of the FIF0 queue and induced tree during in the bread-first search starting at `A`. Each vertex v in the queue is annotated with a pair d/p, where d is the distance $\mathtt{d}_A\mathtt{[}v\mathtt{]}$ from `A` and p is the parent pointer $\mathtt{parent[}v\mathtt{]}$.

FIFO Queue	Induced Tree
[A(0/nil)]	{}
[B(1/A)]	{(A,B)}
[C(2/B), E(2/B), F(2/B)]	{(A,B), (B,C), (B,E), (B,F)}
[E(2/B), F(2/B), D(3/C), G(3/C), H(3/C)]	{(A,B), (B,C), (B,E), (B,F), (C,D), (C,E), (C,F)}
[F(2/B), D(3/C), G(3/C), H(3/C)]	{(A,B), (B,C), (B,E), (B,F), (C,D), (C,E), (C,F)}
[D(3/C), G(3/C), H(3/C)]	{(A,B), (B,C), (B,E), (B,F), (C,D), (C,E), (C,F)}
[G(3/C), H(3/C)]	{(A,B), (B,C), (B,E), (B,F), (C,D), (C,E), (C,F)}
[H(3/C)]	{(A,B), (B,C), (B,E), (B,F), (C,D), (C,E), (C,F)}
[]	{(A,B), (B,C), (B,E), (B,F), (C,D), (C,E), (C,F)}

Here is the tree induced by BFS for the example:



Relaxation

General shortest path algorithms maintain for each vertex v in the graph:

- a **shortest-path estimate** $d_s[v] \geq \delta(s, v)$;
- a **shortest-path parent** `parent[v]` such that d_s`[v] =` d_s`[parent[v]] + w(parent[v], v)`.

The following is an initialization routine for a shortest path algorithm:

```
Initialize-Single-Source(G,s)
  for v ←_vertices(G) do
    d_s[v] ← ∞
    parent[v] ← nil
  d_s[s] ← 0
```

Relaxation on edges (a, b) attempts to reduce the shortest-path estimate for b:

```
Relax((a, b), w)
  if (d_s[a] + w(a,b)) < d_s[b] then
    d_s[b] ← (d_s[a] + w(a,b))
    parent[b] ← a
```

Shortest path algorithms work by initializing d_s[v] as above and then repeatedly relaxing edges until $d_s[v] = \delta(s, v)$. A **shortest-path tree** rooted at the source s is induced by the `parent` fields.

`PQ-Decrease-Key`

Sometimes it is necessary to modify the key of an element stored in a priority queue. This modification can change the priority of the element relative to the priority of the other elements in the priority queue.

In particular, Dijkstra's algorithm maintains a priority queue of vertices v ordered from low to high by their estimated distances d_s[v] from a source vertex s. The relaxation process described above can sometimes decrease this estimated distance for a vertex, increasing its priority relative to other vertices.

For this purpose, we assume an operation `PQ-Decrease-Key(`Q`,` e`,` k_{new}`)`. Let k_{old} be the current value of the key of element e in a priority queue Q where lower values have higher priority. If $k_{new} \leq k_{old}$, then `PQ-Decrease-Key` decreases the key of element `e` from k_{old} to k_{new}; otherwise, the key of e remains unchanged.

- In a complete heap, `PQ-Decrease-Key` can be implemented by applying a `Bubble-Up` operation on e in time $O(\lg(n))$.
- In a leftist heap, `PQ-Decrease-Key` can be implemented by deleting the element e from the heap and reinserting it, which costs $O(\lg(n))$ time.
- There is a kind of heap we have not studied — Fibonacci heaps – in which `PQ-Insert` and `PQ-Delete-Min` take $O(\lg(n))$ amortized time, but `PQ-Decrease-Key` takes only $O(1)$ amortized time.

Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest path problem in the case where there are no negative weights.

Idea: Grow a shortest-path tree from the source vertex s. Every vertex v maintains a shortest-path estimate $d_s[v]$ and `parent` field that indicates the final edge of a path with this estimate. At each step, add the vertex v_{min} with the smallest shortest-path estimate to the tree via the edge to its parent. A priority queue can be used to manage extracting the node with the smallest shortest-path estimate.

The algorithm works because the lack of negative edges guarantees than any path from s to v_{min} that passes through other vertices not yet in the shortest-path tree must have a weight greater than that of the path from s to v_{min} via `parent`$[v]$.

This algorithm is greedy in the sense that, at each step, it adds the "best" node (node with smallest shortest-path estimate) to the growing shortest-path tree.

```
Dijkstra(G, w, s)
  Initialize-Single-Source(G,s)
  shortest ← {} ▷ vertices at which ds[v] = δ(s,v)
  PQ ← Build-PQ (vertices[G])
  ▷ Loop Invariants:
  ▷   (1) Q contains (V - shortest) ordered by ds[v].
  ▷   (2) ds[v] = δ(s,v) for every v in shortest.
  ▷   (3) parent[v] is either nil or in shortest.
  ▷   (4) ds[v] = ds[parent[v]] + w(parent[v],v) for all v with non-nil parent.
  while not PQ-Empty?(Q) do
    a ← PQ-Delete-Min(Q)
    shortest ← shortest ∪ {a}
    for b in Adj[a] do
      Relax((a,b), w)
      PQ-Decrease-Key(Q, b, ds[b])
```

Analysis:

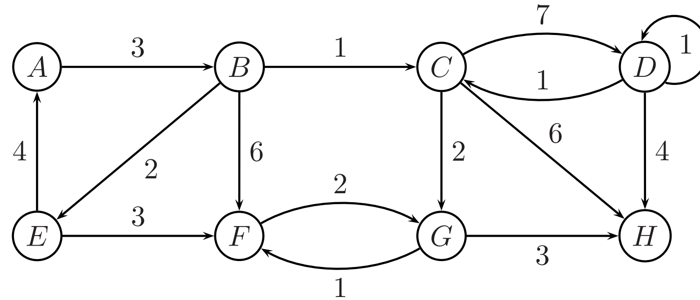
- `Build-PQ` called once on $|V|$ elements
- `PQ-Delete-Min` called $|V|$ times
- `Relax` and `PQ-Decrease-Key` called $|E|$ times.

Priority Queue Implementation	`Build-PQ`	$\lvert V\rvert\cdot$`PQ-Extract-Min`	$\lvert E\rvert\cdot$`PQ-Decrease-Key`	**Total**
Unsorted array/list	$O(V)$	$O(V^2)$	$O(E)$	$O(V^2)$
Complete/leftist heap	$O(V)$	$O(V \cdot \lg(V))$	$O(E \cdot \lg(V))$	$O(E \cdot \lg(V))$
Fibonacci heap	$O(V)$	$O(V \cdot \lg(V))$	$O(E)$	$O(V \cdot \lg(V) + E)$

The $O(E \cdot \lg(V))$ time for complete/leftist heaps assumes that $O(V) \leq O(E)$, which is true for connected graphs and even for most unconnected graphs. Note that a final distance of ∞ indicates a vertex that is in a different connected component from the source vertex s.

Dijkstra Example

Consider the following weighted graph G:



Below is a table that summarizes the steps of Dijkstra's algorithm for the graph G and source vertex `A`. Each entry shows the shortest-path estimate and the shortest-path parent at each step of the algorithm. Those entries in bold are members of the set `shortest`, whose estimates are known to be exact. Estimates are lowered by relaxing an edge. In the table, a relaxation of an edge (s, t) is indicated in a row r wherever s is italicized and t is an entry whose estimate is reduced from row r to row $r + 1$.

Step	A	B	C	D	E	F	G	H
1	**0/Nil**	∞/Nil	∞/Nil	∞/Nil	∞/Nil	∞/Nil	∞/Nil	∞/Nil
2	**0/Nil**	**3/A**	∞/Nil	∞/Nil	∞/Nil	∞/Nil	∞/Nil	∞/Nil
3	**0/Nil**	**3/A**	**4/B**	∞/Nil	5/B	9/B	∞/Nil	∞/Nil
4	**0/Nil**	**3/A**	**4/B**	11/C	**5/B**	9/B	6/G	10/H
5	**0/Nil**	**3/A**	**4/B**	11/C	**5/B**	8/E	**6/G**	10/H
6	**0/Nil**	**3/A**	**4/B**	11/C	**5/B**	**7/G**	**6/G**	9/G
7	**0/Nil**	**3/A**	**4/B**	11/C	**5/B**	**7/G**	**6/G**	**9/G**
8	**0/Nil**	**3/A**	**4/B**	**11/C**	**5/B**	**7/G**	**6/G**	**9/G**

Below is the shortest path tree induced for Dijkstra's algorithm: