Shortest Paths on Graphs

Module: Graphs

1 Overview

Today we talk about *shortest paths*. We have already seen a version of shortest paths, namely when the length of a path is defined as the number of edges on the path. This corresponds to considering all edges in the graph as being equal, or having the same weight. We know that in this case BFS from an arbirary vertex s computes the shortest paths from s; the levels of the vertices as they are visited by BFS correspond to the shortest distances from s; the BFS-tree represents the shortest paths from s, and in order to trace its shortest path from s, a vertex need only trace its parent pointers back to s.

Today we discuss the general version of this problem, when the edges on the graph are not equal but have arbitrary weights. A graph where the edges have weights is called a *weighted* graph. We discuss properties of the shortest path problem, and the common theme in finding shortest paths, which is the concept of *edge relaxation*. We discuss instantiation of these principles in three algorithms: the most general algorithm, Bellman-Ford; a faster algorithm that works only if the edges in the graph are all positive, Dijkstra's algorithm; and an even faster algorithm that works only on DAGs.

2 Shortest paths: problem definition

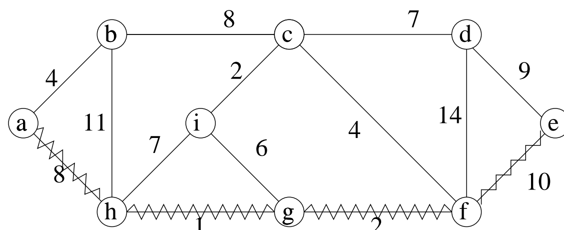
- We discussed that BFS finds shortest paths if the length of a path is defined to be the number of edges on it. This means all edges are the same, and we refer to G as being un-weighted.
- A *weighted* graph, in contrast, is a graph where edges have weights. The weight of an edge can easily be incorporated in the adjacency list/matrix representation of the graph.
- On weighted graphs we are interested in shortest paths with respect to the sum of the weights of edges on a path.
- Example: Finding shortest driving distance between two addresses (lots of www-sites with this functionality). Note that weight on an edge (road) can be more than just distance (weight can e.g. be a function of distance, road condition, congestion probability, etc).
- Formal definition of shortest path: Let $G = (V, E)$ be a weighted graph, directed or undirected.

  Weight of path $P =< v_0, v_1, v_2, \cdots, v_k >$ is $w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

  We denote by $\delta(u, v)$ the weight of the shortest path from u to v. $\delta(u, v)$ is defined as

$$\delta(u,v) = \begin{cases} \min\{w(P) : P \text{ is path from } u \text{ to } v\} & \text{If path exists} \\ \infty & \text{Otherwise} \end{cases}$$

Example: Shortest path from a to e (of length 21)



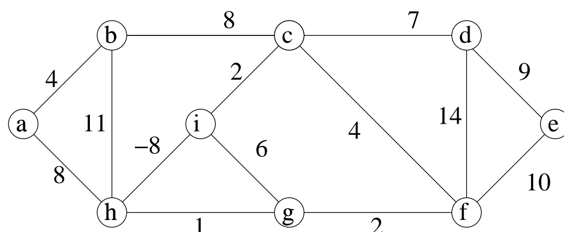
3 Properties of shortest paths

- Optimal substructure:

  Subpaths of shortest paths are shortest paths: If $P =< u = v_0, v_1, v_2, \cdots, v_k = v >$ is a shortest path from u to v, then, for all $i < k$, $P' =< u = v_0, v_1, v_2, \cdots, v_i >$ is shortest path from u to v_i.

- If graph has a cycle with negative weight, shortest paths do not exist (are not well defined)

  Example: If we change weight of edge (h, i) to -8, we have a cycle (i,h,g) with negative weight (-1). Using this we can make the weight of path between a and e arbitrarily low by going through the cycle several times

  Note: We could reframe the problem to compute shortest paths that avoid negative cycles, but that's known to be NPC.

- On the other hand, the problem is well defined if the graph has negative edges, but no negative cycles.

  Example: Edge (h, i) has weight -5 ; ok, no negative cycle

- A shortest path cannot contain cycles. Therefore a shortest path can have at most $|V| - 1$ edges.

- Shortest paths are not necessarily unique.

4 Different variants of shortest path problem

- *Single-pair shortest path*: Find shortest path from u to v

  Note: No algorithm is known for computing the shortest path (u, v) faster, in the worst-case, than computing shortest paths from u to all vertices (SSSP(u))

- *Single-source shortest path (SSSP)*: Find shortest path from source s to all vertices $v \in V$

- *All-pair shortest path (APSP)*: Find shortest path from u to v for all $u, v \in V$

  Note: APSP can be solved by running SSSP $|V|$ times

In this class we'll concentrate on the SSSP problem.

5 Representing shortest paths

When computing SSSP from a source vertex s, we often want to compute not only the length $\delta(s, v)$ of the shortest path to any vertex v, but the actual path (the vertices on the path).

Since the shortest path from s to v has optimal sub-structure, we will represent it by storing, for each vertex v, its predecessor $pred[v]$ on this path. The predecessor of the source will be set to NULL.

At the end of SSSP(s), the edges $(v, pred(v))$ define a tree, called the *shortest path tree*. This is a directed tree rooted at s that stores the shortest path to every vertex reachable from s. The shortest path from s to any vertex v can be found by traversing from v up to the root, following predecessors. Note that the SP-tree is not necessarily unique (since SP are not unique).

6 Computing SP via edge relaxation

The SSSP algorithms all use the idea of relaxation: for each vertex v, we'll maintain a shortest-path estimate $d[v]$ which will represent the length of the currently best-known path from s to v. $d[v]$ will start at ∞ and will get smaller until at the end of the SSSP algorithm $d[v]$ will represent $\delta(s, v)$.

> Initialize-SSSP(source vertex s)
>
> for each $v \in V$ then: $d[v] = \infty$, $pred[v] = NULL$
>
> $d[s] = 0$

The process of *relaxing* an edge (u, v) is the following: when we relax the edge we have a path from s to u of cost $d[u]$ and we found a new path to v through u of cost $d[u] + w_{uv}$. Relaxing the edge consists of checking whether this new path through (u, v) improves the current $d[v]$. More precisely:

> Relax(edge (u, v))
>
> if $d[v] > d[u] + w_{uv}$: $d[v] = d[u] + w_{uv}$, $pred[v] = u$

7 Effects of relaxation

The SSSP algorithm will work by calling Initialize-SSSP and then relaxing edges in a specific order, depending on the algorithm. Initially all $d[v]$ except the source are set to ∞; edge relaxation will brings them down, until the goal finally is for every $d[v]$ to represent $\delta(s, v)$. Below are a couple of properties: (Please check section 24.5 in CLRS).

- The upper-bound property: We always have that $d[v] \geq \delta(s, v)$ for all v, and once $d[v]$ achieves the value $\delta(s, v)$ it never changes.

- No-path property: If there is no path from s to v then we always have $d[v] = \delta(s, v) = \infty$.

- Convergence property: If $s \rightsquigarrow u \rightarrow v$ is a shortest path from s to v, and if $d[u] = \delta(s, u)$ at any time prior to relaxing the edge (u, v), then $d[v] = \delta(s, v)$ at all times afterwards.

- **Path-relaxation lemma:** If $p =< v_0, v_1, ..., v_k >$ is a shortest path from $s = v_0$ to v_k, and the edges of p are relaxed in order $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$, then at the end $d[v_k] = \delta(s, v_k)$. This is true no matter what other relaxations occur intermixed with the edges of p.

Proof: Initially $d[v_0] = d[s] = 0$. This is correct because $\delta(s, s) = 0$. This is maintained at all times thereafter.

When edge (v_0, v_1) is relaxed: we know that this edge is the shortest path from s to v_1, therefore when this edge is relaxed $d[v_1]$ is set to w_{sv_1} which represents $\delta(s, v_1)$. Therefore after the first edge is relaxed, $d[v_1] = \delta(s, v_1)$. This is maintained at all times thereafter.

When edge (v_1, v_2) is relaxed: we know that at this time $d[v_1] = \delta(s, v_1)$ (because the second edge is relaxed after the first edge), and that the shortest path to v_2 goes through v_1 (because of optimal substructure). Therefore after the second edge is relaxed we have $d[v_2] = d[v_1] + w_{v_1 v_2} = \delta(s, v_1) + w_{v_1 v_2} = \delta(s, v_2)$. This is maintained at all times thereafter.

....

In general, when the edge (v_{i-1}, v_i) is relaxed, we know that at this time $d[v_{i-1}] = \delta(s, v_{i-1})$ (because previosu edges have been relatex before this edge), and that the shortest path to v_i goes through v_{i-1} (optimal substructure). Therefore after this edge is relaxed we have that $d[v_i] = \delta(s, v_{i-1}) + w_{v_{i-1} v_i} = \delta(s, v_i)$. This is maintained at all times thereafter.

8 SSSP for DAG's

```
SSSP-DAG(source vertex s)

    //SSSP-initialize

    FOR each v ∈ V: d[v] = ∞, pred[v] = NULL; d[s] = 0;

    sort vertices in topological order

    //do one round of edge relaxation in topological order

    For each vertex u in topological order:

        for each outgoing edge e = (u, v) ∈ E: Relax((u, v))
```

Correctness: At the end of SSSP-DAG(s), $d[v] = \delta(s, v)$ for all v.

Proof: Consider the shortest path from s to a vertex v. The vertices along this path must come in topological order. We relax edges in topological order, therefore we must relax the edges on this path in the order in which they appear on the path. By the path-relaxation lemma, $d[v]$ is computed correctly.

Analysis: This is overall $O(V + E)$ because [..].

9 SSSP for arbitrary graphs– Bellman-Ford algorithm

BELLMAN-FORD(source vertex s)

//SSSP-initialize

FOR each $v \in V$: $d[v] = \infty, pred[v] = NULL$; $d[s] = 0$;

//do $|V| - 1$ rounds of edge relaxation

For $k = 1$ to $|V| - 1$:

for each edge $e = (u, v) \in E$: Relax((u, v))

Correctness: At the end of Bellman-Ford(s), $d[v] = \delta(s, v)$ for all v.

Proof: Consider any vertex v that is reachable from s, and let $p =< s, v_1, v_2, ..., v_k, v >$ be a shortest path from s to v. Note that p must have at most $|V| - 1$ edges, therefore $k + 2 \leq |V| - 1$. Each of the $|V| - 1$ rounds of relaxation relaxes all edges, therefore edge (s, v_1) will be relaxed in the first round, and edge (v_1, v_2) will be relaxed in the second round, ... Therefore by path relaxation lemma, after $k + 2 \leq |V| - 1$ rounds it is guaranteed that $d[v] = \delta(s, v)$.

Analysis: This is overall $O(V \cdot E)$ because [..].

10 SSSP for non-negative weights—Dijkstra's algorithm

- Input: A graph G, directed or undirected, with all edge-weights non-negative; and a source vertex s.
- Output: For each vertex v, $d[v]$ and $pred[v]$, which represent the shortest paths from s to v.
- Dijkstra's algorithm is a greedy algorithm. It improves on Bellman-Ford in the sense that instead of doing $|V| - 1$ relazation rounds, it does only one, and it chooses the edges to relax greedily.
- Its correctness crucially exploits that all edges in G are non-negative.
- As with the previous algorithms, for each vertex, we'll maintain:
  - $d[v]$ = the length of the best known path $s \rightsquigarrow v$ so far; and
  - $pred[v]$ = the predecessor of v on this path.
- As before, $d[v]$ starts at ∞ for all vertices except s. We perform edge relaxations to decrease these distances until they represent the length of shortest path.

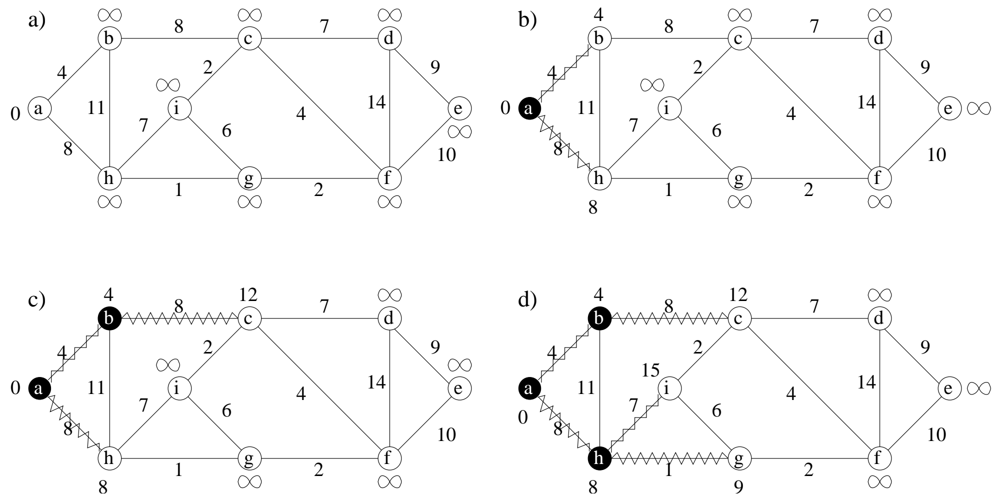
- Order for edge relaxation: The next vertex v is chosen as the vertex with the smallest $d[v]$ among all vertices left. All edges outgoing from u are relaxed.

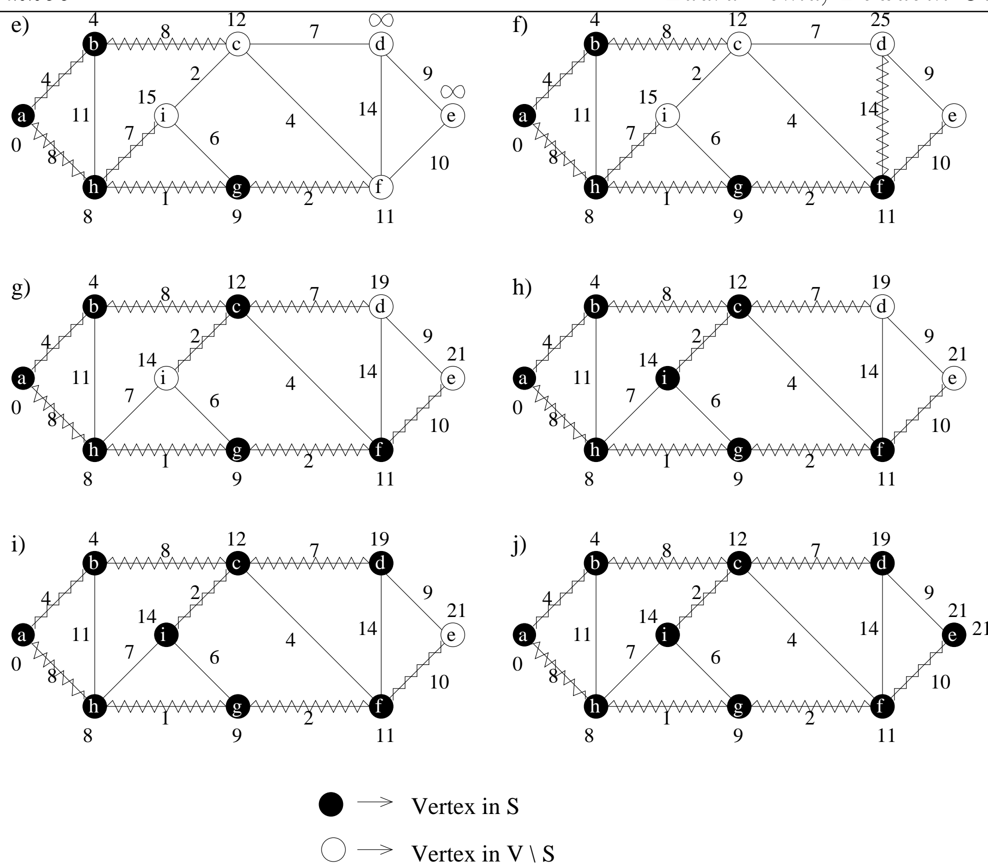
- This is implemented using priority queue on vertices in $V \setminus S$.

```
Dijkstra(source vertex s)
    //initialize
    d[s] = 0; FOR each v ∈ V, v ≠ s: d[v] = ∞
    FOR each v ∈ V: INSERT(Q, v, d[v])

    //do one round of edge relaxation in greedy order of d[]
    S = ∅
    WHILE Q not empty DO
        u = DELETEMIN(Q)
        S = S ∪ {u}
        FOR each e = (u,v) ∈ E with v ∈ V \ S DO
            IF d[v] > d[u] + w(u,v) THEN
                d[v] = d[u] + w(u,v)
                CHANGE(Q, v, d[v])
                pred[v] = u
```

- Example:





- Analysis:
  - We perform $|V|$ INSERT's
  - We perform $|V|$ DELETEMIN's
  - We perform at most one CHANGE for each of the $|E|$ edges

    $\Downarrow$

    Assuming edge-list representation and the priority queue implemented with a heap $O((|V| + |E|) \log |V|) = O(|E| \log |V|)$ running time.

- Correctness:
  - We prove correctness by induction on size of S
  - We will prove that after each iteration of the while-loop the following *invariants* hold:

    (I1) For any vertex $v \notin S$, $d[v]$ represents the length of shortest path from s to v among all possible paths from s to v that contain only vertices from S.

    (I2) For any vertex $v \in S$, $d[v]$ represents the length of the shortest path $\delta(s, v)$

    Note that when the algorithm terminates all vertices are in S, and therefore (I2) means that all vertices have found their shortest paths from s.

– Proof:

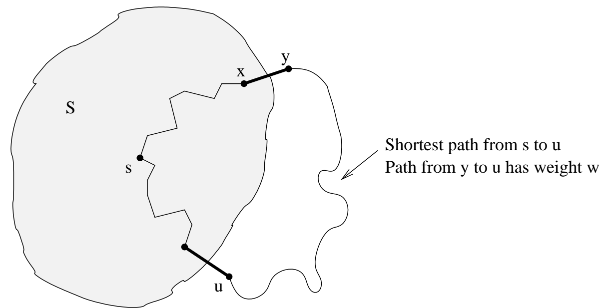
Both invariants trivially holds initially when $S = \emptyset$. To prove that each invariant holds after one iteration of while-loop, given that it holds before the iteration, we need to prove that after adding u to S:

(I1) $d[v]$ correct for all $(u, v) \in E$ where $v \notin S$

- Easily seen to be true since $d[v]$ explicitly updated by algorithm (all the new paths to v of the special type go through u)

(I2) $d[u] = \delta(s, u)$

- Assume by contradiction that the found path is not the shortest, that is, $d[u] > \delta(s, u)$.
- Consider shortest path to u and edge (x, y) on this path where $x \in S$ and $y \notin S$ (such an edge must exist since $s \in S$ and $u \notin S$)



- Vertex u comes out of the priority queue $\Rightarrow d[y] > d[u]$.
- We know that $\delta(s, u) = \delta(s, y) + \delta(y, u) = \delta(s, y) + w$
- We know that $d[y] = \delta(s, y)$ by (I1)
- Therefore $\delta(s, u) = d[y] + w$
- Therefore $\delta(s, u) = d[y] + w > d[u] + w$
- In order for $\delta(s, u)$ to be smaller than $d[u]$, this means that w must be $< 0 \Rightarrow$ contradiction since all weights are non-negative