Data Structures

Graphs

Hikmat Farhat

Introduction

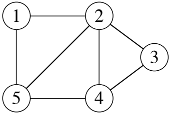
- **Note Most Figures are from Cormen et. al.**
- A **graph** $G = (V, E)$ is a set of vertices V and a set of edges E.
- Each element in E is a pair (v, w) with $v, w \in V$.
- If the pairs are **ordered** then the graph is **directed** (sometimes called **digraph**).
- if $(v, w) \in E$ then we say w is **adjacent** to v
- Usually we associate a **weight** (or **cost**) with each edge.
- A **path** is a sequence of vertices $w_1, \ldots, w_n$ such that $(w_i, w_{i+1}) \in E$.
- the **length** of a path is the number of edges in it

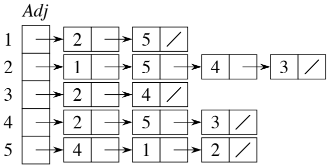
- A path is said to be **simple** if all vertices, except possibly the first and last, are **distinct**.
- A **cycle** is a path such that $w_1 = w_n$.
- in an undirected graph we require that the edges be distinct to have a cycle.
- for example v, w, v should not be considered a cycle since (v, w) and (w, v) are the same edge.
- A graph is said to be **acyclic** if it contains no cycles.
- A graph in which from every vertex there is path to every other vertex is called **connected**.

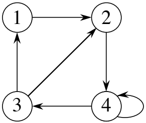
Graph representation

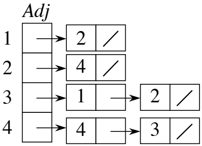
- There are essentially two ways to represent a graph
  - Adjacency matrix.
  - Adjacency list.
- Most of the time adjacency list is better since it is $O(|E|+|V|)$ in memory requirement.
- This is the preferred representation when the graph is sparse, $|E| \ll |V^2|$.
- The adjacency matrix is $O(|V^2|)$ in memory requirement and it is preferred when the graph is **dense**, $|E| \approx |V^2|$.
- In the adjacency matrix representation it is much faster to check whether two vertices are adjacent.

Examples









Topological Sort

- Topological sort is an ordering of **directed acyclic** graphs.
- The idea is that if there is a path from node u to node v then v appears **after** u in the ordering.
- As an example, we use topological sort to list the **valid** sequence of courses that are consistent with prerequisites.

Example

- A simple algorithm to perform topological sort is to find a node with no **incoming** edges.
- We can print that edge then follow the adjacency list.
- Define the **indegree** of a node v as the number of edges (u, v).
- Suppose that for each node in the graph we have the indegree and the adjacency list then a simple algorithm would be

for $i = 1$ *to* n **do**
    u=findIndegreeZero()
    print u
    foreach $v \in Adj[u]$ **do**
        $v.indegree \leftarrow v.indegree - 1$

- The complexity of the above algorithm is $O(|V|^2)$ because findIndegreeZero has to scan all nodes every time which is $O(|v|)$
- since we do it $O(|V|))$ times then the total is $O(|V|^2)$.
- Not counting the cost of computing the indegree of all nodes initially.

Breadth First Search

- As we will see later many algorithms depend on **breadth first search** (BFS).
- Given a graph $G = (V, E)$ and a **source** node s, BFS systematically "discovers" all vertices that can be reached from s.
- It is breadth first because all vertices at distance k from s are discovered **before** any vertex at distance $k + 1$ is discovered.
- BFS works by coloring nodes with two different colors: **white** and **black**.
- A white node means it has not been discovered. Black means it has been discovered.

- The algorithm starts by coloring all nodes white except the source s is colored black.
- It then proceed with the discovery of all of s neighbors.
- Given a node v
  - $v.d$ is the distance (number of links) from s to v.
  - $adj[v]$ is the list of v's neighbors.
  - $v.p$ is the predecessor of v in the path from s to v.

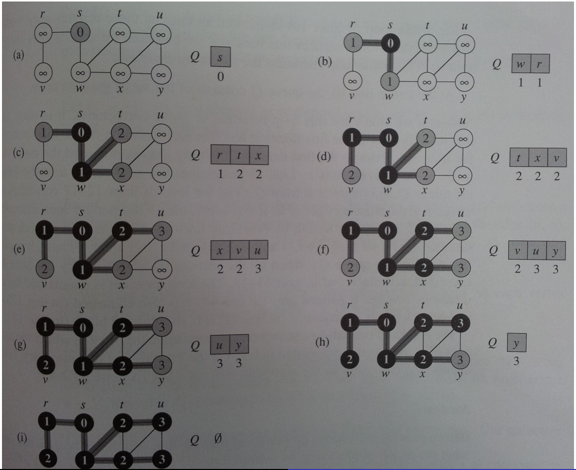
BFS Initialization

BFS(G,v)
foreach $v \in V - \{s\}$ **do**
  $v.color \leftarrow WHITE$
  $v.d \leftarrow 0$
  $v.p \leftarrow NULL$
$s.color \leftarrow BLACK$
$s.d \leftarrow 0$
$s.p \leftarrow NULL$
$Q \leftarrow \emptyset$
ENQUEUE(Q,s)

BFS Pseudo Code

```
BFS(G,v)
while Q ≠ ∅ do
    u ← DEQUEUE(Q)
    foreach v ∈ Adj[u] do
        if v.color = WHITE then
            v.color ← BLACK
            v.d ← u.d + 1
            v.p ← u
            ENQUEUE(Q,v)
```

Example


(a)
Q s
0
(b)
Q w r
1 1
(c)
Q r t x
1 2 2
(d)
Q t x v
2 2 2
(e)
Q x v u
2 2 3
(f)
Q v u y
2 3 3
(g)
Q u y
3 3
(h)
Q y
3
(i)
Q ∅

Complexity of BFS

- To analyze the complexity of BFS first we note that after initialization no vertex color is set to white.
- The above implies that each vertex is enqueued (and dequeued) only once.
- Since the enqueue/dequeue operations are $O(1)$ then for all nodes it is $O(|V|)$.
- When a vertex is dequeued we scan the adjacency list and the sum of all adjacency list is just $|E|$
- Therefore the total cost of BFS is $O(|V| + |E|)$.

Shortest Paths

- Given a graph $G = (V, E)$ and a source node $s \in V$. We define the **shortest-path** distance $\delta(s, v)$ from s to $v \in V$ to be the minimum number of edges in any path from s to v.
- BFS not only discovers every vertex $v \in V$ reachable from a source s
- But also $v.d = \delta(s, v)$ and
- The shortest-path from s to v is **composed** of the shortest-path from s to $v.p$ **followed** by the edge $(v.p, v)$.
- The above observation allows us to determine not only the cost $\delta(s, v)$ but also the exact path by iterating backwards over $v.p$.

Depth First Search

- In a **depth first search** DFS edges are explored out of the most recently discovered node.
- As the name implies we go "deeper" whenever it is possible.
- When all the neighbors of a node v are discovered we "backtrack" to the parent of v and explore other nodes.
- When we are done discovering the descendants of some source s and some nodes remain undiscovered then one of them is selected as source and the process is repeated.
- When the algorithm is done with a certain node, it records the **discovery time** and **finishing time**

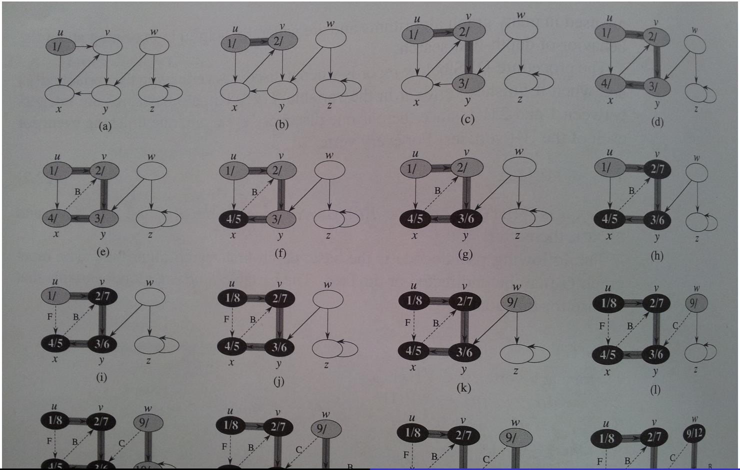
DFS Pseudo Code

```
DFS(G)
foreach v ∈ V do
    v.color ← WHITE
    v.p ← NULL
time ← 0
foreach v ∈ V do
    if v.color = WHITE then
        DFS-VISIT(v)
```

DFS-VISIT Pseudo Code

```
DFS-VISIT(u)
u.color ← GRAY
time ← time + 1
u.d ← time
foreach v ∈ adj[u] do
    if v.color = WHITE then
        DFS-VISIT(v)
u.color ← BLACK
times ← time + 1
u.f ← time
```

DFS Example


u v w x y z 1/ (a)
u v w x y z 1/ 2/ (b)
u v w x y z 1/ 2/ 3/ (c)
u v w x y z 1/ 2/ 4/ 3/ (d)
u v w x y z 1/ 2/ B 4/ 3/ (e)
u v w x y z 1/ 2/ B 4/5 3/ (f)
u v w x y z 1/ 2/ B 4/5 3/6 (g)
u v w x y z 1/ 2/7 B 4/5 3/6 (h)
u v w x y z 1/ 2/7 F B 4/5 3/6 (i)
u v w x y z 1/8 2/7 F B 4/5 3/6 (j)
u v w x y z 1/8 2/7 9/ F B 4/5 3/6 (k)
u v w x y z 1/8 2/7 9/ F B C 4/5 3/6 (l)
u v w 1/8 2/7 9/ F B C
u v w 1/8 2/7 9/ F B C B
u v w 1/8 2/7 9/ F B C
u v w 1/8 2/7 9/12 F B C B

Complexity

- The initialization to WHITE is $O(|V|)$
- Then DFS is called $O(|V|)$ times.
- Each time DFS-VISIT is called **only once** for each node because it is called on WHITE nodes only.
- The cost of DFS-VISIT(v) is $O(|adj[v]|)$.
- Thus the cost of all calls to DFS-VISIT is

$$\sum_{v \in V} |adj[v]| = O(|E|)$$

- Therefore the total cost is

$$O(|E| + |V|)$$

Topological Sort Revisited

- We can implement an efficient topological sort using DFS as follows
    1. Call DFS on the graph.
    2. Every time a node is finished add it to the front of a linked list
    3. When done the resulting list is the topological sort.

DFS Topological Sort Example

Transitive Closure

- Given a graph $G =< V, E >$ the transitive closure is a two dimensional array (a relation) $tc[][]$ such that $t[u][v] = 1$ if v can be reached from u and 0 otherwise.
- The transitive closure closure can be computed with a slight modification of DFS shown below.

```
foreach s ∈ V do
    SEARCH(s,s);
SEARCH(s,u)
tc[s][u] ← 1
foreach v ∈ adj[u] do
    if tc[s][v] = 0 then
        SEARCH(s,v)
```

Minimum Spanning Trees

- In many application, when the system is represented by a graph we need to find a **Minimum Spanning Tree** (MST).
- As the name suggest this collection of nodes is
  1. **A tree**.
  2. **Spanning**. meaning includes all the nodes of the graph.
  3. It has the **least total cost** of all such trees.
- First we need to introduce some preliminary operations.

Disjoint Sets Data Structures

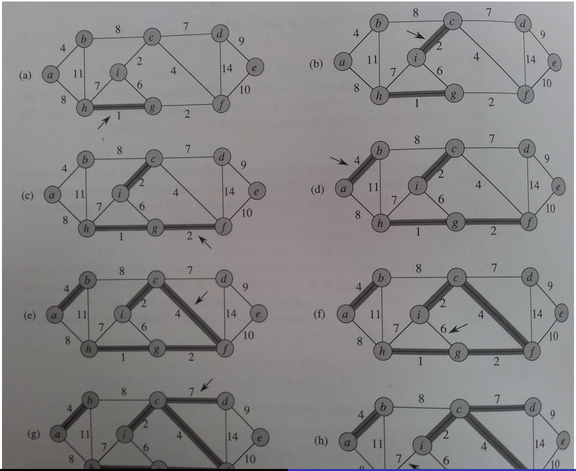
- We introduce some operations on **disjoint sets**. Any element is contained in **only one set**.
- MAKE-SET(x): create a new set whose only member is x.
- FIND-SET(x):returns a pointer to the representative of the set containing x.
- UNION(x,y):combine the sets containing x and y into a new set.

Kruskal's Algorithm

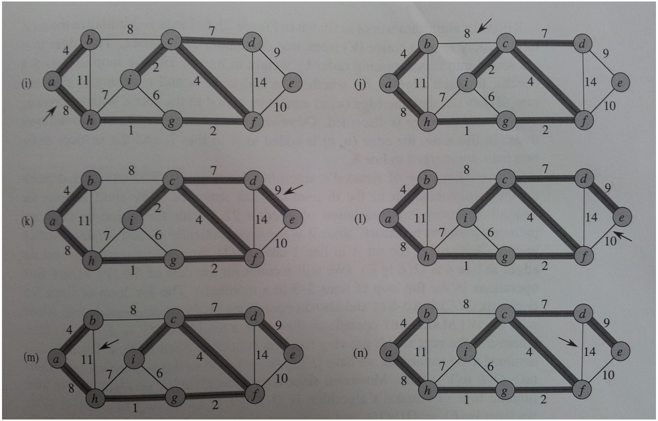
- Kruskal's algorithm computes a MST of a given graph.
- Every edge has an associated weight or cost.
- The idea is to build the MST by adding an edge every iteration.
- The edges are considered by increasing order.
- An edge is added if it doesn't create a cycle.
- The algorithm stops when there are no more edges to consider.

`MST-KRUSKAL(G)`
$A \leftarrow \emptyset$
foreach $v \in V$ **do**
    `MAKE-SET(v)`
$F \leftarrow$ `SORT-EDGES(E)`
foreach $(u, v) \in F$ **do**
    if $\texttt{FIND-SET}(u) \neq \texttt{FIND-SET}(v)$ **then**
        $A \leftarrow A \cup \{(u, v)\}$
        $\texttt{UNION}(u, v)$

Example



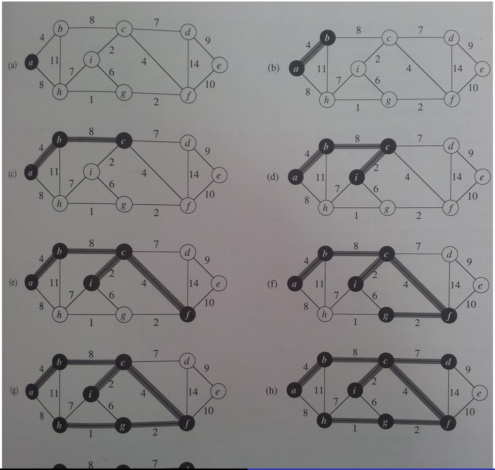
Example



Prim's Algorithm

MST-PRIM(G,r)
foreach $v \in V$ **do**
    $v.key \leftarrow \infty$
    $v.p \leftarrow NULL$
$r.key \leftarrow 0$
$Q \leftarrow V$
while $Q \neq \emptyset$ **do**
    $u \leftarrow$ DELETE-MIN(Q)
    foreach $v \in Adj[u]$ **do**
        if $w(u, v) < v.key$ *and* $v \in Q$ **then**
            $v.key \leftarrow w(u, v)$
            $v.p \leftarrow u$

Example



Why does it work?

- Both Kruskal's and Prim's algorithms are special cases of a general method to obtain a minimum spanning tree.
- The basic idea is based on the following:
- Maintain a set of edges A.
- Before every iteration A is a subset of some minimum spanning tree.
- At each step we add an edge to A such that A is **still** a subset of some MST.
- An edge having that property is called **safe** for A.

```
MST(G)
A ← ∅
while A is not MST do
    find edge (u, v) safe for A
    A ← A ∪ {(u, v)}
return A
```

- The above algorithm looks easy.
- But how do we find a safe edge?

Some Definitions

- Let $G = (V, E)$ be a graph with some real-valued weight function $w : E \to R$.
- A **cut** $(S, V - S)$ of the graph G is a **partition** of V.
- We say a cut $(S, V - S)$ **respects** $A \subseteq E$ if no edge in A crosses the cut.
- An edge is said to be a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.

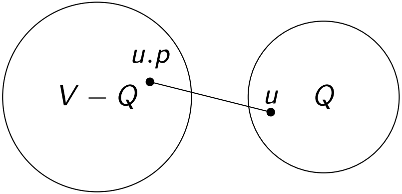
This is why it works

- The reason why both algorithms work is the following theorem

Theorem

Let A be a set of edges included in some minimum spanning tree, $(S, V - S)$ a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

Correctness of Prim's Algorithm



- At the beginning of every iteration (except the first) Prim's algorithm starts by removing u where $u.key$ is minimum. This means that $(u.p, u)$ is a light edge for the cut $(Q, V - Q)$
- Therefore Prim's algorithm is correct.

Correctness of Kruskal's Algorithm

- Prior to every iteration of Kruskal's algorithm we have
  1. A forest (a collection of trees) $G_A = (V, A)$. (initially is A is empty)
  2. Select an edge $(u, v) \in E - A$ with
     2.1 $w(u, v)$ is minimal.
     2.2 $u \in T_u$ and $v \notin T_u$ where T_u is a tree in G_A that contains u.
  3. From the above we have that: $(T_u, V - T_u)$ is a cut that respects A and (u, v) is a light edge crossing that cut.
- From the theorem we know that (u, v) is a safe edge for A.

Complexity

- **Kruskal**: we use the union find operations we learned in the beginning of the semester. Let $\mid V \mid = n$ and $\mid E \mid = m$.
- Recall that we use an array *id* to specify the parent of node in the (logical) tree that represents a given group.
- e.g. node $id[i]$ is the parent of i. Initially each node is its own parent: $id[i] = i$ thus the first **for** loop is $\Theta(n)$.
- Sorting is $\Theta(m \log m)$.
- In our implementation, Union is $\Theta(1)$ and FIND-SET is $\Theta(\log n)$. Therefore the foreach loop is $\Theta(m \log n)$.
- Adding all the contributions we get: $\Theta(n + m \log m + m \log n)$.

Strongly Connected Components

- Given a graph $G =< V, E >$ we say that the set of vertices $C \subseteq V$ is a **strongly connected component** if
- for every pair $u, v \in C$ we have: $u \rightsquigarrow v$ and $v \rightsquigarrow u$
- We can print all strongly connected components in a graph by doing DFS twice. The first over the graph and the second over the transpose of the graph.

Kosaraju Algorithm

foreach $v \in V$ **do**
    if $v.color = WHITE$ **then**
        DFS-VISIT(v)
Reverse all the edges of G and reset all colors
foreach $v \in V$ *in decreasing finish time* **do**
    if $v.color = WHITE$ **then**
        DFS-VISIT(v)

Single Source Shortest Path

- Given a graph $G = (V, E)$ with a real-valued weight function w we often as the question:
- What is the minimal cost (shortest) path from $s \in V$ to all other vertices of the graph.
- We will look at two algorithms that perform that taks
  1. Bellman-Ford.
  2. Dijkstra.
- First we need some definitions and theorems.

- Given a graph $G = (V, E)$ and a real-valued weight function $w : E \to R$.
- weight of path $p = (v_0, \ldots, v_k)$ sometimes written as

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- The shortest path cost δ

$$\delta(u, v) = \begin{Bmatrix} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from u to v} \\ \infty & \text{otherwise} \end{Bmatrix}$$

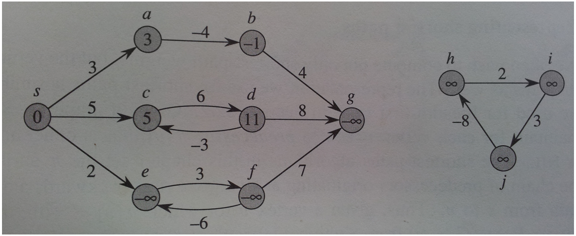
Properties of Shortest Path

- Subpaths of shortest path are subpath: Given a graph $G = (V, E)$ and weight function $w : E \to \mathbf{R}$ let $p = (v_1, \ldots v_k)$ be a shortest path from v_1 to v_k then for any $1 \leq i, j \leq k$, $p_{ij} = (v_i, \ldots, v_j)$ is a shortest path from v_i to v_j.
- **Proof**: we write $v_1 \overset{p}{\rightsquigarrow} v_k$ which can be decomposed into $v_1 \overset{p_i}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_j}{\rightsquigarrow} v_k$
- Then $w(p) = w(p_i) + w(p_{ij}) + w(p_j)$ so if p_{ij} is not the shortest path then $\exists p'_{ij}$ with $w(p'_{ij}) < w(p_{ij})$ then we can write
- $w(p') = w(p_i) + w(p'_{ij}) + w(p_j) < w(p)$ a contradiction since p is the shortest path from v_1 to v_k.

Negative weight

- Even if a path contains edges with negative weight a shortest path can still be defined.
- It is undefined if the path contains a negative weight **cycle**.
- This is because we can "cross" the cycle as many times as we want, every time lower the cost.
- Therefore in the case when there is a negative cycle on a path from u to v then we set $\delta(u, v) = -\infty$ where $\delta(a, b)$ is the shortest path cost from a to b.

Example of Negative Cycles



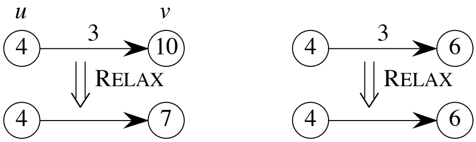
- $\delta(s,a)=3$,$\delta(s,b)=-1$,$\delta(s,c)=5$,$\delta(s,d)=11$.
- (e,f) form a negative cycle therefore any node reachable from s through this cycle has $\delta=-\infty$
  $\delta(s,e)=\delta(s,f)=\delta(s,g)=-\infty$
- h,i,j are not reachable from s thus
  $\delta(s,h)=\delta(s,i)=\delta(s,j)=\infty$

Representation of Shortest Paths

- In all the algorithms that we will deal with, we maintain for every vertex v its predecessor $v.p$ (which could be NULL)
- At **termination** $v.p$ will be the predecessor of v on a shortest path from source s to v.
- We also maintain a value $v.d$ which at termination will be the value of the shortest path cost from source s to v.
- During the execution of the algorithm $v.d$ will be **an upper bound** on the value of the shortest path cost.

Relaxation

- **Relaxing** an edge (u, v) means testing if we can improve the shortest path cost of v by using the edge (u, v).
- If we can then we update $v.d$ and $v.p$.



- In the figure to the left the cost of v was changed to the new cost (7) whereas to the right it was not changed since the new cost (7) is bigger than the current (6).
- What is NOT shown is the change to $v.p$ in the first case.

Initialization and Relaxation

- Initially all vertices (except the source) have cost ∞ and no predecessors (including the source).

```
INITIALIZE(G,s)
foreach v ∈ V do
    v.d ← ∞
    v.p ← NULL
s.d ← 0
```

```
RELAX(u,v)
if v.d > u.d + w(u,v) then
    v.d ← u.d + w(u,v)
    v.p ← u
```

Properties of Relaxation

Relaxation has the following properties

Path relaxation If $p = (v_0, \ldots, v_k)$ is the shortest path from $s = v_0$ to $v = v_k$ and the edges of p are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ then $v.d = \delta(s, v)$. (note that this is true regardless of any other relaxations)

Predecessor subgraph If $v.d = \delta(s, v)$ for all $v \in V$ then the predecessor subgraph is a shortest-paths tree rooted at s.

Upper Bound We always have $v.d \geq \delta(s, v)$ and once $v.d = \delta(s, v)$ it never changes.

Bellman-Ford Algorithm

- The Bellman-Ford algorithm computes the shortest path from a given source to all other nodes in the graph.
- It works with negative weights.
- It can detect negative cycles.
- It uses the previously defined procedure RELAX to compute the shortest path.

Bellman-Ford Pseudo Code

```
BELLMAN-FORD(G,s);
INITIALIZE(G,s)
for i ← 1 To V − 1 do
    foreach (u, v) ∈ E do
        RELAX(u,v)

foreach (u, v) ∈ E do
    if v.d > u.d + w(u, v) then
        return FALSE
return TRUE
```

Example

Correctness of Bellman-Ford

- If the graph has no negative cycles then the shortest path cannot contain a cycle since remove it "shortens" (at least the same for 0 cost cycle) the path
- Therefore if we have n vertices a shortest path cannot visit more that n of them and thus it contains at most $n-1$ edges.
- Bellman-Ford is iterated $n-1$ times and each time ALL the edges are relaxed.
- So if $p_1, \ldots p_k$ is a shortest path, iteration i relaxes all edges INCLUDING p_{i-1}, p_i.
- This means among ALL relaxations the edges of the path are relaxed in the order $(p_1, p_2), \ldots, (p_{k-1}, p_k)$
- By the path-relaxation property $d[p_k] = \delta(s, p_k)$

Complexity of Bellman-Ford

- The initialization is $O(|V|)$.
- the double loop is $O(|V| \cdot |E|)$.
- Therefore the total cost of the Bellman-Ford is $O(|V| \cdot |E|)$.

Dijkstra's Algorithm

- Dijkstra's algorithm is another single source shortest path.
- It works when all weights are **positive**.
- We will see that it is faster than the Bellman-Ford algorithm.
- It maintains a set S of nodes whose shortest paths have been determined
- All other nodes are kept in a min-priority queue to keep track of the next node to process.

Dijkstra Pseudo Code

DIJKSTRA(G,s);
INITIALIZE(G,s)
$S \leftarrow \emptyset$
$Q \leftarrow V$
while $Q \neq \emptyset$ **do**
    $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    foreach $v \in Adj[u]$ **do**
        RELAX(u,v)

Example

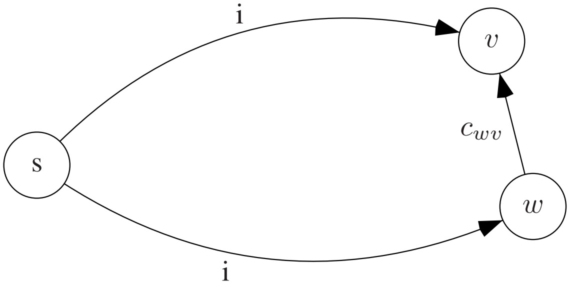
Complexity

- The running time of Dijkstra's algorithm depends on the implementation of the queue.
- Using a min-heap on a sparse graph gives complexity of $O((V+E)\log V)$.
- This is because the while loop executes V times. The extract-min is $O(\log V)$ for a cost of $V \log V$. The relax includes an key update which means $\log V$. Since each edge is relaxed at most once then the total is E with a cost of $E \log V$.

Bellman-Ford Revisited

- We will take a look at a variation of the Bellman-Ford discussed earlier.
- The basic idea is that with n nodes the shortest path from any two nodes can have at most $n-1$ edges.
- Let s be the source node. We need to compute the shortest path from s to all other nodes.
- For any v let $d[i,v]$ be the cost of the shortest path from s to v that contains **at most** i edges. Then (see figure)

$$d[i+1,v] = \min(d[i,v], \min_{w\in V}(d[i,w] + c_{wv}))$$


i
v
c_{wv}
s
w
i

- From the previous information we have
- Since we are guaranteed that the shortest path is at most $n-1$ edges the above recursive equation gives us an algorithm to compute the shortest path by iterating of the length.
- Note that the values for step i is saved to be used later, namely in step $i+1$.
- This strategy of saving values instead of recomputing is called Dynamic Programming.

`BELLMAN-FORD(G,s);`
foreach $v \in V$ **do**
  $d[0, v] = \infty$
$d[0, s] = 0$
for $i = 1, \ldots, n$ **do**
  $d[i, v] = \min(d[i-1, v], \min_{w \in V}(d[i-1, w] + c_{vw}))$

Eulerian cycles

- A Eulerian path in a graph is a path from vertex u to vertex v that uses every edge exactly once.
- A Eulerian cycle is a closed (i.e. $u = v$ Eulerian path)
- Formally, a path $v_1, \ldots, v_k$ in a graph $G = (V, E)$ is said to be Eulerian iff
  1. $\forall e \in E, \exists i$ such that $(v_{i-1}, v_i) = e$.
  2. $\forall i, j$ we have $i \neq j \Rightarrow (v_{i-1}, v_i) \neq (v_{j-1}, v_j)$.

Theorem

A graph $G = (V, E)$ has a Eulerian cycle iff every vertex has even degree

Proof.

- ($\Rightarrow$) Assume that a Eulerian cycle, $v_1 \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_k$ exists. Consider an arbitrary vertex $v_i \neq 1, k$. that occurs l times in the path. Every time v_i occurs it is of the form v_{i-1}, v_i, v_{i+1} where $(v_{i-1}, v_i) \in E$ and $(v_i, v_{i+1}) \in E$ which means for every occurrence of v_i in the path, two edges (distinct by definition) are "used". The same reasoning applies to v_1 and v_k since $v_1 = v_k$.
- ($\Leftarrow$)Assume that every vertex has an even degree. We construct a Eulerian cycle as follows.
  - Start at an arbitrary vertex u, and choose an unused edge every time until you get back to u and there are no more unused edges to choose from.
  - Next we select a vertex v included in the previous "walk" and repeat until we get back to v.

- We still need to prove that when starting at vertex u and choosing previously unused edges we get back to u.
- By way of contradiction assume that starting with vertex u we get "stuck" in vertex $v \neq u$. Let the followed path be $u, x_1, \ldots, x_k, v$.
- Every time v is visited (except the last) two edges of v are used therefore an odd number of edges of v are used which is a contradiction because every vertex was assumed to have an even number of edges.