Data Structures – LECTURE 15

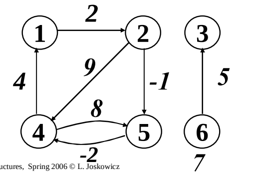
Shortest paths algorithms

- Properties of shortest paths
- Bellman-Ford algorithm
- Dijsktra's algorithm

Chapter 24 in the textbook (pp 580–599).

Weighted graphs -- reminder

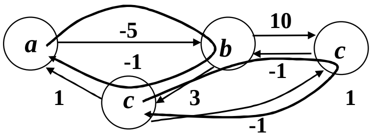
- A *weighted graph* is graph in which edges have *weights* (*costs*) $w(v_i, v_j)$.
- A graph is a weighted graph in which all costs are 1. Two vertices with no edge (path) between them can be thought of having an edge (path) with weight ∞.



The cost of a path is the sum of the costs of its edges:

Negative-weight edges

- Shortest paths are well-defined as long as there are no negative-weight cycles.
- In negative cycles, the longer the path, the lower the value → shortest path has infinite number of edges!



- Allow negative-weight edges, but disallow (or detect) negative-weight cycles!

Two basic properties of shortest paths

Triangle inequality

Let $G=(V,E)$ be a weighted directed graph, $w: E \rightarrow R$ a weight function and $s \in V$ be a source vertex. Then, for all edges $e=(u,v) \in E$:

$$\delta(s,v) \leq \delta(s,u) + w(u,v)$$

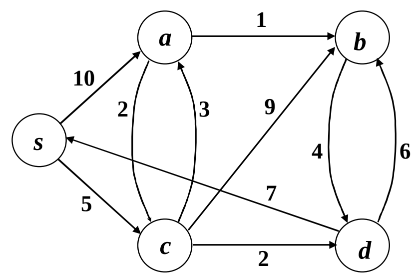
Optimal substructure of a shortest path

Let $p = <v_1, .. v_k>$ be the shortest path between v_1 and v_k. The sub-path between v_i and v_j, where $1 \leq i,j \leq k$, $p_{ij} = <v_i, .. v_j>$ is a shortest path.

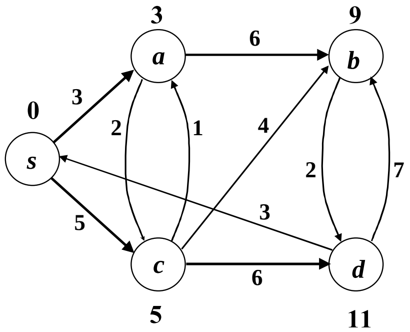
Shortest paths and cycles

- The shortest path between any two vertices has no negative-weight cycles.
- The representation for shortest paths between a vertex and all other vertices is the same as the one used in the unweigthed BFS: *breath-first tree:*
  $G_\pi = (V_\pi, E_\pi)$ such that $V_\pi = \{v \in V: \pi[v] \neq \textbf{\textit{null}}\} \cup \{s\}$
  and $E_\pi = \{(\pi[v],v), v \in V - \{s\}\}$
- We will prove that a breath-first tree is a shortest-path tree for its root s in which vertices reachable from s are in it and the unique simple path from s to v is shortest.

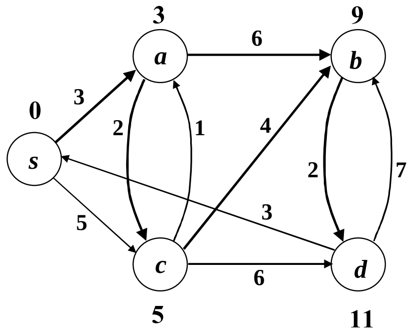
Example: weighted graph



Example: shortest-path tree (1)



Example: shortest-path tree (2)



Estimated distance from source

- As for BFS on unweighted graphs, we keep a label which is the current best estimate of the shortest distance between s and v.
- Initially, $dist[s] = 0$ and $dist[v] = \infty$ for all $v \neq s$, and $\pi[v] =$ ***null.***
- At all times during the algorithm, $dist[v] \geq \delta(s,v)$.
- At the end, $dist[v] = \delta(s,v)$ and $(\pi[v],v) \in E_\pi$

Edge relaxation

- The process of *relaxing an edge* (u,v) consists of testing whether it can improve the shortest path from s to v so far by going through u.

Relax(u,v)
    if $dist[v] > dist[u] + w(u,v)$
      then $dist[v] \leftarrow dist[u] + w(u,v)$
          $\pi[v] \leftarrow u$

Properties of shortest paths and relaxation

1. Triangle inequality
   $\forall e = (u,v) \in E$: $\delta(s,v) \leq \delta(s,u) + w(u,v)$
2. Upper-boundary property
   $\forall v \in V$: $dist[v] \geq \delta(s,v)$ at all times.
   $dist[v]$ is monotonically decreasing.
3. No-path property
   if there is no path from s to v, then $dist[v]= \delta(s,v) = \infty$

Properties of shortest paths and relaxation

4. Convergence property
   if $s \rightarrow u \rightarrow v$ is a shortest path in G for some u and v, and $dist[u]= \delta(s,u)$ at any time prior to relaxing edge (u,v), then $dist[v]= \delta(s,v)$ at all times afterwards.

5. Path-relaxation property
   Let $p = \langle v_0, .. v_k \rangle$ be shortest path between v_0 and v_k. when edges are relaxed in order
   $$(v_0,v_1),(v_1,v_2),\ldots,(v_{k-1}, v_k)$$
   then $dist[v_k]= \delta(s,v_k)$.

Properties of shortest paths and relaxation

6. Predecessor sub-graph property

Once $dist[v] = \delta(s,v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Two shortest-path algorithms

1. Bellmann-Ford algorithm
   Handles and detects negative cycles
1. Dijkstra's algorithm – Generalization of BFS
   Requires non-negative weights

Assumptions:

1. Adjacency list representation
2. $n + \infty = \infty$

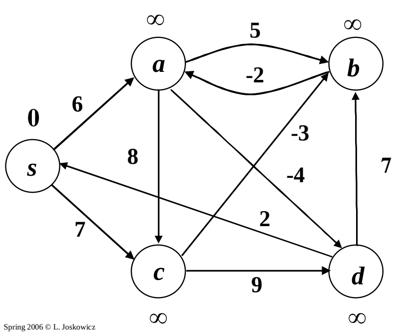
Bellman-Ford's algorithm: overview

- Allows negative weights. If there is a negative cycle, returns "a negative cycle exists".
- The idea:
  - There is a shortest path from s to any other vertex that does not contain a non-negative cycle, it can be eliminated to produce a shorter path.
  - The maximal number of edges in such a path with no cycles is $|V| - 1$, because it can have at most $|V|$ nodes on the path if there is no cycle.
  - $\Rightarrow$ it is enough to check paths of up to $|V| - 1$ edges.

Bellman-Ford's algorithm

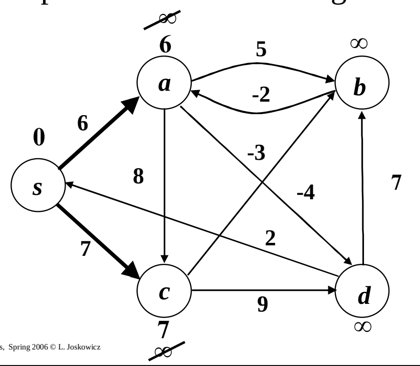
```
Bellman-Ford(G, s)
Initialize(G, s)
for i ← 1 to |V| − 1
   for each edge (u, v) ∈ E
      do if dist[v] > dist[u] + w(u, v)
         dist[v] ← dist[u] + w(u, v)
         π[v] ← u
for each edge (u, v) ∈ E
   if dist[v] > d[u] + w(u, v) return "negative cycle"
```


Example: Bellman-Ford's algorithm (0)



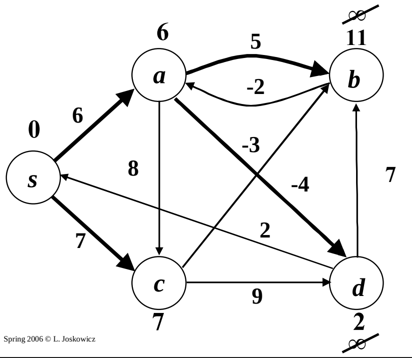
Edge order
(a,b)
(a,c)
(a,d)
(b,a)
(c,b)
(c,d)
(d,s)
(d,b)
(s,a)
(s,b)

Example: Bellman-Ford's algorithm (1)



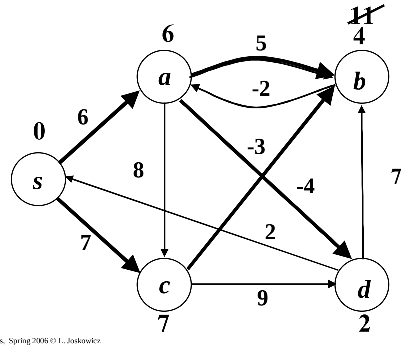
Edge order
(a,b)
(a,c)
(a,d)
(b,a)
(c,b)
(c,d)
(d,s)
(d,b)
(s,a)
(s,c)

Example: Bellman-Ford’s algorithm (2)



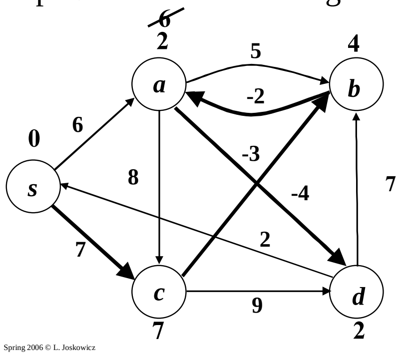
Edge order
(a,b)
(a,c)
(a,d)
(b,a)
(c,b)
(c,d)
(d,s)
(d,b)
(s,a)
(s,c)

Example: Bellman-Ford’s algorithm (2)



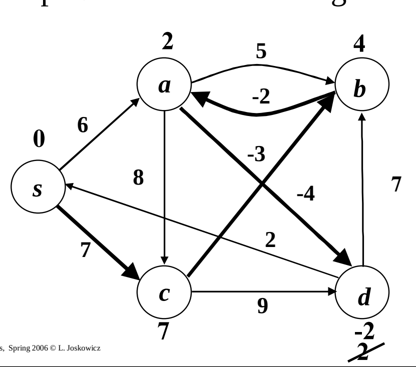
Edge order
(a,b)
(a,c)
(a,d)
(b,a)
(c,b)
(c,d)
(d,s)
(d,b)
(s,a)
(s,c)

Example: Bellman-Ford’s algorithm (3)



Edge order
(a,b)
(a,c)
(a,d)
(b,a)
(c,b)
(c,d)
(d,s)
(d,b)
(s,a)
(s,b)

Example: Bellman-Ford’s algorithm (4)



Edge order
(a,b)
(a,c)
(a,d)
(b,a)
(c,b)
(c,d)
(d,s)
(d,b)
(s,a)
(s,b)

Bellman-Ford’s algorithm: properties

- The first pass over the edges – only neighbors of s are affected (1-edge paths). All shortest paths with one edge are found.
- The second pass – shortest 2-edge paths are found
- After $|V|$–1 passes, all possible paths are checked.
- Claim: we need to update any vertex in the |V| pass iff there is a negative cycle reachable from s in G.

Bellman Ford algorithm: proof (1)

- => if we need to update an edge in the last iteration then there is a negative cycle, because we proved before that if there are no negative cycles, and the shortest paths are well defined, we find them in the $|V|$–1 iteration.
- <= if there is a negative cycle, we will discover a problem in the last iteration. Suppose there is a negative cycle and the algorithm does not find any problem in the last iteration. This means that for all edges, we have that

for all edges in the cycle.

Bellman Ford algorithm: proof (2)

- Proof by contradiction: for all edges in the cycle

- After summing up over all edges in the cycle, we discover that the term on the left is equal to the first term on the right (just a different order of summation). We can subtract them, and we get that the cycle is actually positive, which is a contradiction.

Bellman-Ford's algorithm: complexity

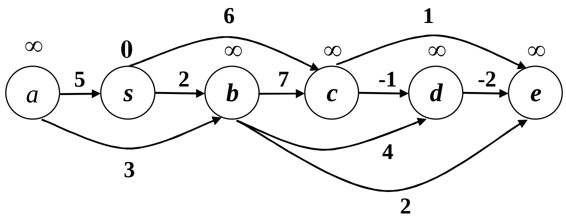
- Visits $|V|–1$ vertices → $O(|V|)$
- Performs vertex relaxation on all edges → $O(|E|)$
- Overall, $O(|V|.|E|)$ time and $O(|V|+|E|)$ space.

Bellman-Ford on DAGs

For Directed Acyclic Graphs (DAG), $O(|V|+|E|)$ relaxations are sufficient when the vertices are visited in topologically sorted order:

DAG-Shortest-Path(*G*)
1. Topologically sort the vertices in *G*
2. Initialize *G* (*dist*[*v*] and π(*v*)) with *s* as source.
3. **for** each vertex *u* in topologically sorted order **do**
4.   **for** each vertex *v* incident to *u* **do**
5.   Relax(*u*,*v*)

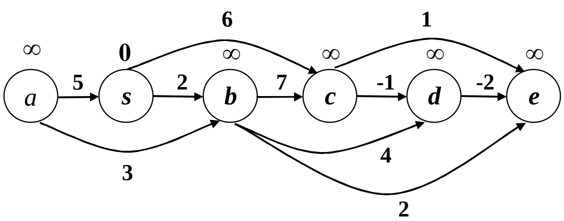
Example: Bellman-Ford on a DAG (0)



E = (*a*,*s*) (*a*,*b*) (*s*,*b*) (*s*,*c*) (*b*,*c*) (*b*,*d*) (*b*,*e*) (*c*,*d*) (*c*,*e*) (*d*,*e*)

Vertices sorted from left to right

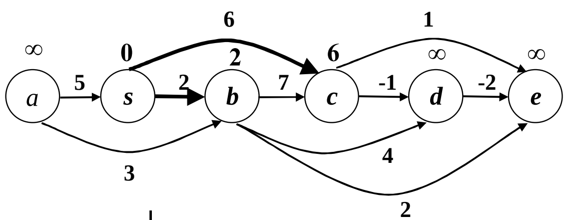
Example: Bellman-Ford on a DAG (1)



E = (*a*,*s*) (*a*,*b*) (*s*,*b*) (*s*,*c*) (*b*,*c*) (*b*,*d*) (*b*,*e*) (*c*,*d*) (*c*,*e*) (*d*,*e*)

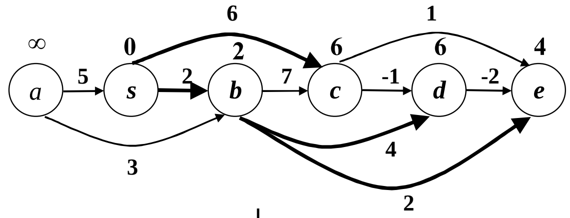
Vertices sorted from left to right

Example: Bellman-Ford on a DAG (2)



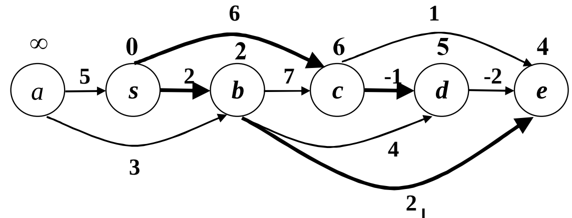
E = (*a*,*s*) (*a*,*b*) | (*s*,*b*) (*s*,*c*) (*b*,*c*) (*b*,*d*) (*b*,*e*) (*c*,*d*) (*c*,*e*) (*d*,*e*)

Example: Bellman-Ford on a DAG (3)



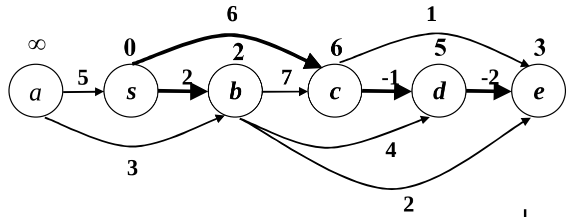
E = (a,s) (a,b) (s,b) (s,c) | (b,c) (b,d) (b,e) (c,d) (c,e) (d,e)

Example: Bellman-Ford on a DAG (4)



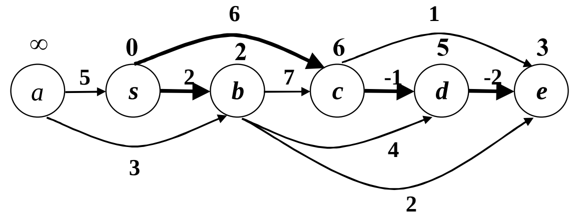
E = (a,s) (a,b) (s,b) (s,c) (b,c) (b,d) (b,e) | (c,d) (c,e) (d,e)

Example: Bellman-Ford on a DAG (5)



E = (a,s) (a,b) (s,b) (s,c) (b,c) (b,d) (b,e) (c,d) (c,e) | (d,e)

Example: Bellman-Ford on a DAG (6)



Bellman-Ford on DAGs: correctness

Path-relaxation property

Let $p = <v_0, .. v_k>$ be the shortest path between v_0 and v_k. When the edges are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots (v_{k-1}, v_k)$, then $dist[v_k] = \delta(s,v_k)$.

In a DAG, we have the correct ordering!

Therefore, the complexity is $O(|V|+|E|)$.

Dijkstra's algorithm: overview

Idea: Do the same as BFS for unweighted graphs, with two differences:

– use the cost as the distance function

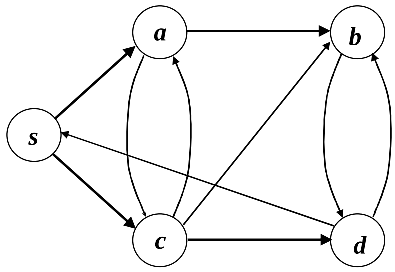
– use a minimum priority queue instead of a simple queue.

The BFS algorithm

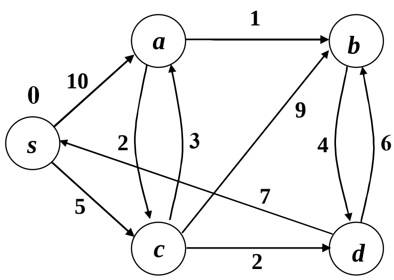
<u>BFS(*G*, *s*)</u>

label[*s*] ← *current; dist*[*s*] = 0*;* π[*s*] = ***null***
for all vertices *u* in *V* – {*s*} **do**
  label[*u*] ← *not_visited; dist*[*u*] = ∞*;* π[*u*] = ***null***
EnQueue(*Q*,*s*)
while *Q* is not empty **do**
  u ← DeQueue(*Q*)
  for each *v* that is a neighbor of *u* **do**
    if *label*[*v*] = *not_visited* **then** *label*[*v*] ← *current*
    dist[*v*] ← *dist*[*u*] + 1; π[*v*] ← *u*
    EnQueue(*Q*,*v*)
  label[*u*] ← *visited*

Example: BFS algorithm



Example: Dijkstra's algorithm

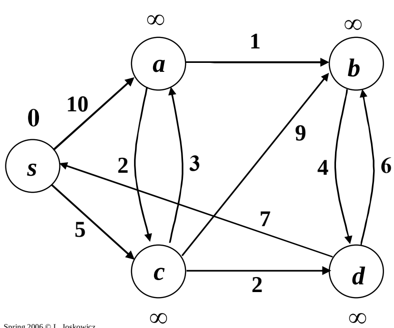


Dijkstra's algorithm

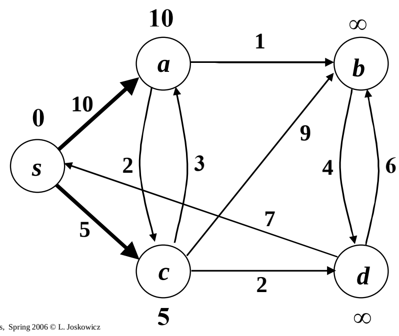
<u>Dijkstra(*G*, *s*)</u>

label[*s*] ← *current; dist*[*s*] = 0*;* π[*u*] = ***null***
for all vertices *u* in *V* – {*s*} **do**
  label[*u*] ← *not_visited; dist*[*u*] = ∞*;* π[*u*] = ***null***
Q ← *s*
while *Q* is not empty **do**
  u ← ~~DeQueue(*Q*)~~ Extract-Min(*Q*)
  for each *v* that is a neighbor of *u* **do**
    ~~**if** *label*[*v*] = *not_visited* **then** *label*[*v*] ← *current*~~
    if *d*[*v*] > *d*[*u*] + *w*(*u*,*v*)
      then *d*[*v*] ← *d*[*u*] + *w*(*u*,*v*); π[*v*] = *u*
    Insert-Queue(*Q*,*v*)
  ~~*label*[*u*] = *visited*~~

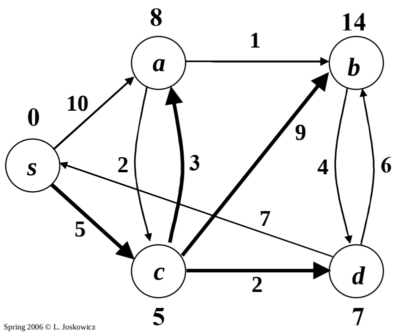
Example: Dijkstra's algorithm (1)



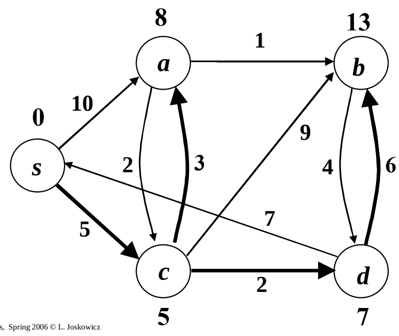
Example: Dijkstra's algorithm (2)



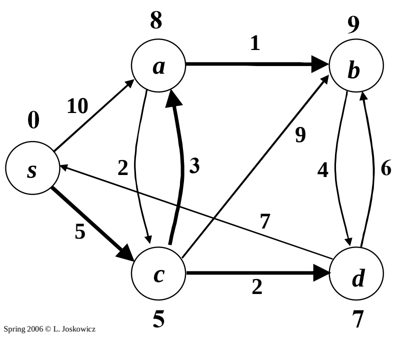
Example: Dijkstra's algorithm (3)



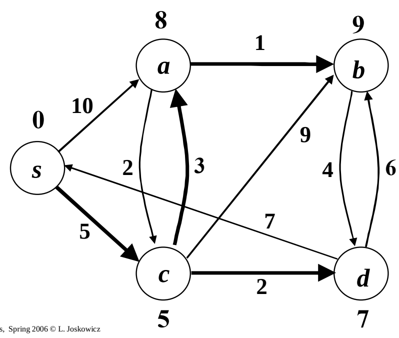
Example: Dijkstra's algorithm (4)



Example: Dijkstra's algorithm (5)



Example: Dijkstra's algorithm (6)



Dijkstra's algorithm: correctness (1)

<u>Theorem:</u> Upon termination of Dijkstra's algorithm $dist[v] = \delta(s,v)$ for each vertex $v \in V$

<u>Definition</u>: a path from s to v is said to be a *special* path if it is the shortest path from s to v in which all vertices (except maybe for v) are inside S.

<u>Lemma</u>: At the end of each iteration of the **while** loop, the following two properties hold:

1. For each $w \in S$, $dist[w]$ is the length of the <u>shortest path</u> from s to w which stays inside S.
2. For each $w \in (V–S)$, $dist(w)$ is the length of the <u>shortest special path</u> from s to w.

The theorem follows when $S = V$.

Dijkstra's algorithm: correctness (2)

<u>Proof</u>: by induction on the size of S.

- For $|S|=1$, it is clearly true: $dist[v] = \infty$ except for the neighbors of s, which contain the length of the shortest special path.
- <u>Induction step</u>: suppose that in the last iteration node v was added added to S. By the induction assumption, $dist[v]$ is the length of the <u>shortest special path</u> to v. It is also the length of the <u>general shortest path</u> to v, since if there is a shorter path to v passing through nodes of S, and x is the first node of S in that path, then x would have been selected and not v. So the first property still holds.

Dijkstra's algorithm: correctness (3)

Property 2: Let $x \in S$. Consider the shortest new special path to w

If it doesn't include v, $dist[x]$ is the length of that path by the induction assumption from the last iteration since $dist[x]$ did not change in the final iteration.

If it does include v, then v can either be a node in the middle or the last node before x. Note that v cannot be a node in the middle since then the path would pass from s to v to y in S, but by Property 1, the shortest path to y would have been inside S → v need not be included.

If v is the last node before x on the path, then $dist[x]$ contains the distance of that path, by the substitution $dist[x] = dist[v] + w(v,x)$ in the algorithm.

Dijkstra's algorithm: complexity

- The algorithm performs $|V|$ Extract-Min operations and $|E|$ Insert-Queue operations.
- When the priority queue is implemented as a heap, insert takes $O(\lg|V|)$ and Extract-Min takes $O(\lg(|V|)$. The total time is $O(|V|\lg|V|) + O(|E|\lg|V|) = O(|E|\lg|V|)$
- When $|E|=O(|V|^2)$, this is not optimal. In this case, there are many more insert than extract operations.
- Solution: Implement the priority queue as an array! Insert will take $O(1)$ and Extract-Min $O(|V|)$ → $O(|V|^2) + O(|E|) = O(|V|^2)$, better than the heap when $|E|$ is $O(|V|^2/\lg(|V|))$.

Summary

- Solving the shortest-path problem on weighted graphs is performed by relaxation, based on the path triangle inequality:

  $\forall e = (u,v) \in E: \quad \delta(s,v) \leq \delta(s,u) + w(u,v)$
- Two algorithms for solving the problem:
  - Bellman Ford: for each vertex, relaxation on all edges. Takes $O(|E|.|V|)$ time for graphs with non-negative cycles.
  - Dijkstra: BFS-like, takes $O(|E|\lg|V|)$ time.

ERROR: undefined
OFFENDING COMMAND:

STACK: