Algorithm Design VII

Path in Graphs

Guoqiang Li
School of Software
SHANGHAI JIAO TONG UNIVERSITY

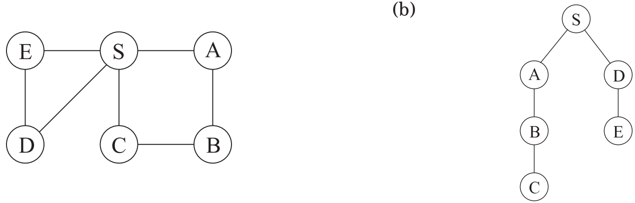
Distances

Distances

Definition

The distance between two nodes is the length of the shortest path between them.

(a) (b)



Breadth-First Search

SHANGHAI JIAO TONG UNIVERSITY

```
BFS(G, v)
input : Graph G = (V, E), directed or undirected; Vertex v ∈ V
output: For all vertices u reachable from v, dist(u) is the set to the distance from v to u

for all u ∈ V do
    dist(u) = ∞;
end
dist[v] = 0;
Q = [v] queue containing just v;
while Q is not empty do
    u=Eject(Q);
    for all edge (u, s) ∈ E do
        if dist(s) = ∞ then
            Inject(Q,s); dist[s] = dist[u] + 1;
        end
    end
end
```

Correctness

Lemma

For each $d = 0, 1, 2, \ldots$ there is a moment at which,

1. *all nodes at distance $\leq d$ from s have their distances correctly set;*
2. *all other nodes have their distances set to ∞; and*
3. *the queue contains exactly the nodes at distance d.*

Lengths on Edges

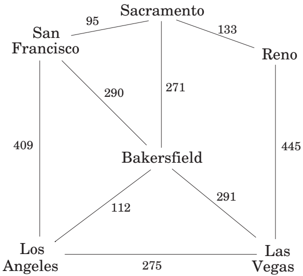
BFS treats all edges as having the same length.

It is rarely true in applications where shortest paths are to be found.

Every edge $e \in E$ with a length l_e.

If $e = (u, v)$, we will sometimes also write

$$l(u, v) \qquad \text{or} \qquad l_{uv}$$



Dijkstra’s Algorithm

An Adaption of Breadth-First Search

BFS finds shortest paths in any graph whose edges have unit length.

An Adaption of Breadth-First Search

BFS finds shortest paths in any graph whose edges have unit length.

Q: Can we adapt it to a more general graph $G = (V, E)$ whose edge lengths l_e are positive integers?

An Adaption of Breadth-First Search

BFS finds shortest paths in any graph whose edges have unit length.

Q: Can we adapt it to a more general graph $G = (V, E)$ whose edge lengths l_e are positive integers?

A simple trick: For any edge $e = (u, v)$ of E, replace it by l_e edges of length 1, by adding $l_e - 1$ dummy nodes between u and v. It might take time

$$O(|V| + \sum_{e \in E} l_e)$$

An Adaption of Breadth-First Search

BFS finds shortest paths in any graph whose edges have unit length.

Q: Can we adapt it to a more general graph $G = (V, E)$ whose edge lengths l_e are positive integers?

A simple trick: For any edge $e = (u, v)$ of E, replace it by l_e edges of length 1, by adding $l_e - 1$ dummy nodes between u and v. It might take time

$$O(|V| + \sum_{e \in E} l_e)$$

It is bad in case we have edges with high length.

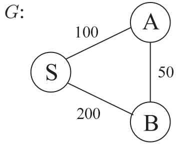
Alarm Clocks

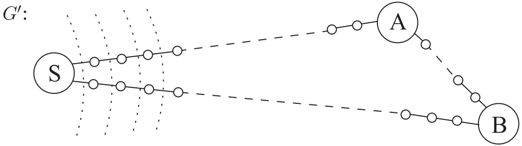
Set an alarm clock for node s at time 0.

Repeat until there are no more alarms:

The next alarm goes off at time T, for node u. Then:

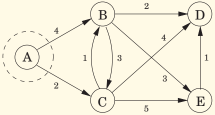
- The distance from s to u is T.
- For each neighbor v of u in G:
  - If there is no alarm yet for v, set one for time $T + l(u, v)$.
  - If v's alarm is set for later than $T + l(u, v)$, then reset it to this earlier time.





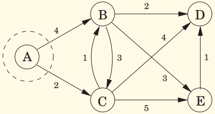
An Example

An Example

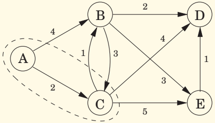


A: 0	D: ∞
B: 4	E: ∞
C: 2	

An Example

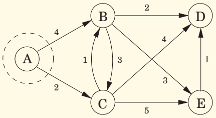


A: 0	D: ∞
B: 4	E: ∞
C: 2	

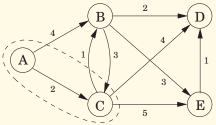


A: 0	D: 6
B: 3	E: 7
C: 2	

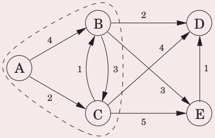
An Example



A: 0	D: ∞
B: 4	E: ∞
C: 2	

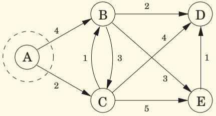


A: 0	D: 6
B: 3	E: 7
C: 2	

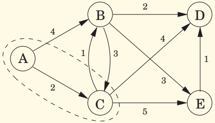


A: 0	D: 5
B: 3	E: 6
C: 2	

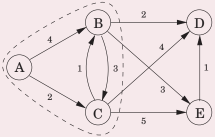
An Example



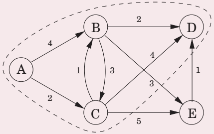
A: 0	D: ∞
B: 4	E: ∞
C: 2	



A: 0	D: 6
B: 3	E: 7
C: 2	

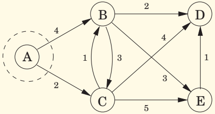


A: 0	D: 5
B: 3	E: 6
C: 2	

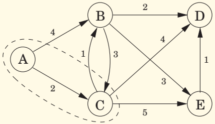


A: 0	D: 5
B: 3	E: 6
C: 2	

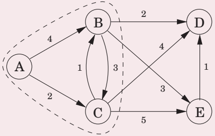
An Example



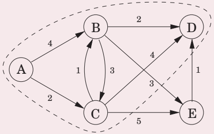
A: 0	D: ∞
B: 4	E: ∞
C: 2	



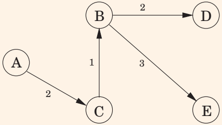
A: 0	D: 6
B: 3	E: 7
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



Dijkstra's Shortest-Path Algorithm


```
DIJKSTRA(G, l, s)
input  : Graph G = (V, E), directed or undirected; positive edge length
         {l_e | e ∈ E}; Vertex s ∈ V
output: For all vertices u reachable from s, dist(u) is the set to the distance
         from s to u

for all u ∈ V do
    dist(u) = ∞; prev(u) = nil;
end
dist(s) = 0;
H =makequeue(V)\\ using dist-values as keys;
while H is not empty do
    u=deletemin(H);
    for all edge (u, v) ∈ E do
        if dist(v) > dist(u) + l(u, v) then
            dist(v) = dist(u) + l(u, v);   prev(v) = u;
            decreasekey (H,v);
        end
    end
end
```

Data Structure Trailer: Priority Queue

Priority queue is a data structure usually implemented by heap.

Data Structure Trailer: Priority Queue

Priority queue is a data structure usually implemented by heap.

- Insert: Add a new element to the set.

Data Structure Trailer: Priority Queue

Priority queue is a data structure usually implemented by heap.

- Insert: Add a new element to the set.
- Decrease-key: Accommodate the decrease in key value of a particular element.

Data Structure Trailer: Priority Queue

Priority queue is a data structure usually implemented by heap.

- Insert: Add a new element to the set.
- Decrease-key: Accommodate the decrease in key value of a particular element.
- Delete-min: Return the element with the smallest key, and remove it from the set.

Data Structure Trailer: Priority Queue

Priority queue is a data structure usually implemented by heap.

- Insert: Add a new element to the set.
- Decrease-key: Accommodate the decrease in key value of a particular element.
- Delete-min: Return the element with the smallest key, and remove it from the set.
- Make-queue: Build a priority queue out of the given elements, with the given key values. (In many implementations, this is significantly faster than inserting the elements one by one.)

Data Structure Trailer: Priority Queue

Priority queue is a data structure usually implemented by heap.

- Insert: Add a new element to the set.
- Decrease-key: Accommodate the decrease in key value of a particular element.
- Delete-min: Return the element with the smallest key, and remove it from the set.
- Make-queue: Build a priority queue out of the given elements, with the given key values. (In many implementations, this is significantly faster than inserting the elements one by one.)

The first two let us set alarms, and the third tells us which alarm is next to go off.

Running Time

Since `makequeue` takes at most as long as $|V|$ insert operations, we get a total of $|V|$ `deletemin` and $|V| + |E|$ `insert/decreasekey` operations.

Shortest Paths in the Presence of Negative Edges

Negative Edges

Shanghai Jiao Tong University

Dijkstra's algorithm works because the shortest path from the starting point s to any node v must pass exclusively through nodes that are closer than v.

Negative Edges

Dijkstra's algorithm works because the shortest path from the starting point s to any node v must pass exclusively through nodes that are closer than v.

This no longer holds when edge lengths can be negative.

Negative Edges

Dijkstra's algorithm works because the shortest path from the starting point s to any node v must pass exclusively through nodes that are closer than v.

This no longer holds when edge lengths can be negative.

Q: What needs to be changed in order to accommodate this new complication?

Negative Edges

Dijkstra's algorithm works because the shortest path from the starting point s to any node v must pass exclusively through nodes that are closer than v.

This no longer holds when edge lengths can be negative.

Q: What needs to be changed in order to accommodate this new complication?

A crucial invariant of Dijkstra's algorithm is that the $dist$ values it maintains are always either overestimates or exactly correct.

Negative Edges

Dijkstra's algorithm works because the shortest path from the starting point s to any node v must pass exclusively through nodes that are closer than v.

This no longer holds when edge lengths can be negative.

Q: What needs to be changed in order to accommodate this new complication?

A crucial invariant of Dijkstra's algorithm is that the $dist$ values it maintains are always either overestimates or exactly correct.

They start off at ∞, and the only way they ever change is by updating along an edge:

$\text{UPDATE}\ ((u, v) \in E)$

$$dist(v) = min\{dist(v), dist(u) + l(u, v)\};$$

Update

$\texttt{UPDATE}\ ((u, v) \in E)$

$dist(v) = min\{dist(v), dist(u) + l(u, v)\};$

This UPDATE operation expresses that the distance to v cannot possibly be more than the distance to u, plus $l(u, v)$. It has the following properties,

1. It gives the correct distance to v in the particular case where u is the second-last node in the shortest path to v, and $dist(u)$ is correctly set.
2. It will never make $dist(v)$ too small, and in this sense it is safe.

Update

$\texttt{UPDATE}\ ((u, v) \in E)$

$$dist(v) = min\{dist(v), dist(u) + l(u, v)\};$$

Update

$\texttt{UPDATE}\ ((u, v) \in E)$

$dist(v) = min\{dist(v), dist(u) + l(u, v)\};$

Let

$$s \to u_1 \to u_2 \to u_3 \to \ldots \to u_k \to t$$

be a shortest path from s to t.

Update

$\text{UPDATE}\ ((u, v) \in E)$

$dist(v) = min\{dist(v), dist(u) + l(u, v)\};$

Let

$$s \to u_1 \to u_2 \to u_3 \to \ldots \to u_k \to t$$

be a shortest path from s to t.

This path can have at most $|V| - 1$ edges (why?).

Update

$\texttt{UPDATE}\ ((u, v) \in E)$

$dist(v) = min\{dist(v), dist(u) + l(u, v)\};$

Let

$$s \to u_1 \to u_2 \to u_3 \to \ldots \to u_k \to t$$

be a shortest path from s to t.

This path can have at most $|V| - 1$ edges (why?).

If the sequence of updates performed includes $(s, u_1), (u_1, u_2), \ldots, (u_k, t)$, in that order, then by rule 1 the distance to t will be correctly computed.

Update

UPDATE $((u, v) \in E)$

$dist(v) = min\{dist(v), dist(u) + l(u, v)\};$

Let

$$s \to u_1 \to u_2 \to u_3 \to \ldots \to u_k \to t$$

be a shortest path from s to t.

This path can have at most $|V| - 1$ edges (why?).

If the sequence of updates performed includes $(s, u_1), (u_1, u_2), \ldots, (u_k, t)$, in that order, then by rule 1 the distance to t will be correctly computed.

It doesn't matter what other updates occur on these edges, or what happens in the rest of the graph, because updates are safe (by rule 2).

Bellman-Ford Algorithm

But still, if we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order?

Bellman-Ford Algorithm

But still, if we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order?

We simply update all the edges, $|V| - 1$ times!

Bellman-Ford Algorithm

SHANGHAI JIAO TONG UNIVERSITY

SHORTEST-PATHS(G, l, s)

input : Graph $G = (V, E)$, edge length$\{l_e \mid e \in E\}$; Vertex $s \in V$

output: For all vertices u reachable from s, $dist(u)$ is the set to the distance from s to u

for *all* $u \in V$ **do**

  $dist(u) = \infty$;

  $prev(u) = nil$;

end

$dist[s] = 0$;

repeat $|V| - 1$ times: **for** $e \in E$ **do**

  UPDATE(e);

end

Bellman-Ford Algorithm

SHORTEST-PATHS (G, l, s)

input : Graph $G = (V, E)$, edge length$\{l_e \mid e \in E\}$; Vertex $s \in V$

output: For all vertices u reachable from s, $dist(u)$ is the set to the distance from s to u

for *all* $u \in V$ **do**

  $dist(u) = \infty$;

  $prev(u) = nil$;

end

$dist[s] = 0$;

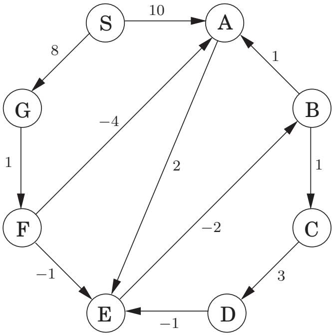
repeat $|V| - 1$ times: **for** $e \in E$ **do**

  UPDATE(e);

end

Running time: $O(|V| \cdot |E|)$

Bellman-Ford Algorithm



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Negative Cycles

If the graph has a negative cycle, then it doesn't make sense to even ask about shortest path.

Negative Cycles

If the graph has a negative cycle, then it doesn't make sense to even ask about shortest path.

Q: How to detect the existence of negative cycles:

Negative Cycles

If the graph has a negative cycle, then it doesn't make sense to even ask about shortest path.

Q: How to detect the existence of negative cycles:

Instead of stopping after $|V| - 1$, iterations, perform one extra round.

Negative Cycles

If the graph has a negative cycle, then it doesn't make sense to even ask about shortest path.

Q: How to detect the existence of negative cycles:

Instead of stopping after $|V| - 1$, iterations, perform one extra round.

There is a negative cycle if and only if some $dist$ value is reduced during this final round.

Shortest Paths in DAGs

Graphs without Negative Edges

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

- graphs without negative edges,

Graphs without Negative Edges

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

- graphs without negative edges,
- and graphs without cycles.

Graphs without Negative Edges

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

- graphs without negative edges,
- and graphs without cycles.

We will now see how the single-source shortest-path problem can be solved in just linear time on directed acyclic graphs.

Graphs without Negative Edges

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

- graphs without negative edges,
- and graphs without cycles.

We will now see how the single-source shortest-path problem can be solved in just linear time on directed acyclic graphs.

As before, we need to perform a sequence of updates that includes every shortest path as a subsequence.

Graphs without Negative Edges

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

- graphs without negative edges,
- and graphs without cycles.

We will now see how the single-source shortest-path problem can be solved in just linear time on directed acyclic graphs.

As before, we need to perform a sequence of updates that includes every shortest path as a subsequence.

- In any path of a DAG, the vertices appear in increasing linearized order.

A Shortest-Path Algorithm for DAG

SHANGHAI JIAO TONG UNIVERSITY

DAG-SHORTEST-PATHS(G, l, s)
input : Graph $G = (V, E)$, edge length$\{l_e \mid e \in E\}$; Vertex $s \in V$
output: For all vertices u reachable from s, $dist(u)$ is the set to the distance from s to u

for *all* $u \in V$ **do**
  $dist(u) = \infty$;
  $prev(u) = nil$;
end
$dists = 0$;
linearize G;
for *each* $u \in V$ *in linearized order* **do**
  for *all* $e \in E$ **do**
    UPDATE(e);
  end
end

A Shortest-Path Algorithm for DAG

SHANGHAI JIAO TONG UNIVERSITY

A Shortest-Path Algorithm for DAG

SHANGHAI JIAO TONG UNIVERSITY

The scheme does not require edges to be positive.

A Shortest-Path Algorithm for DAG

The scheme does not require edges to be positive.

Even can find longest paths in a DAG by the same algorithm: just negate all edge lengths.

Exercises

Exercises 1

Professor Fake suggests the following algorithm for finding the shortest path from node s to node t in a directed graph with some negative edges: add a large constant to each edge weight so that all the weights become positive, then run Dijkstra's algorithm starting at node s, and return the shortest path found to node t.

Exercises 2

You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $v_0 \in V$. Give an efficient algorithm for finding shortest paths between *all pairs of nodes*, with the one restriction that these paths must all pass through v_0.