Algorithm Design VII

Path in Graphs

Guoqiang Li
School of Software, Shanghai Jiao Tong University

SHANGHAI JIAO TONG UNIVERSITY

Distances

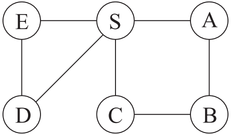
Distances

一定得是最短距离！

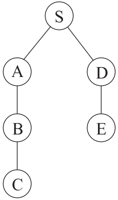
Definition

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

(a)



(b)



```
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
```

v：BFS起点
Eject：简单的出队列
∞：表示该点未被访问

Correctness

SHANGHAI JIAO TONG UNIVERSITY

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*.*

DFS与BFS：DFS更关注于"最大的深度"，即一次DFS结束是没有能到达的点了才会回溯并结束，更适用于连通性的判断；BFS更加"扁平"，因为他会在一次Search中考虑所有能直接相连的点的情况，因而更适合进行一种"比较"，即用于寻找所谓的"最短路径"。

Lengths on Edges

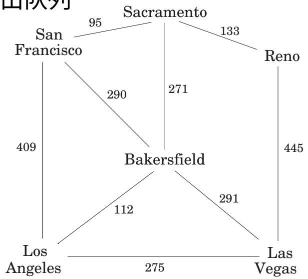
BFS treats all edges as having the same length. 所以可以无差别的直接出队列

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

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



BFS的时间复杂度为O(V+E)。因为根据上述算法实现可以知道，一个顶点最多被访问2次(一次入队列，一次出队列)，一条边最多被访问2次(无向图2次(因为没有对于变得访问次数作出限制，对于边uv，遍历u的关联边与v的关联边的时候都会访问同一边)，有向图1次)，所以为O(V+E)

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

SHANGHAI JIAO TONG UNIVERSITY

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

根据上述的O(V+E),就可以得出右面的结果
添加了Σ l_e-1个节点

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

通过添加自定义的人工节点，
来将店点与点之间的距离变为
单位长度

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

Alarm Clocks

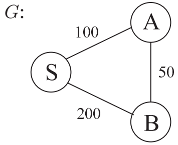
这个alarm clock实际上就是一种对于Dijkstra算法中dist量的表示

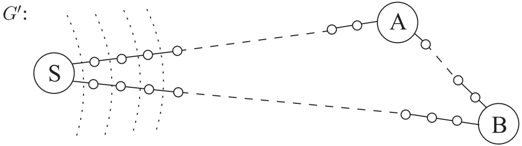
Set an alarm clock for node s at time 0.

Repeat until there are no more alarms:

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

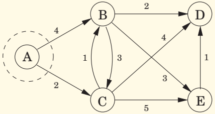
- The distance from s to u is T.
- For each neighbor v of u in G:
  - If there is no alarm yet for v, set one for time $T + l(u, v)$.
  - If v's alarm is set for later than $T + l(u, v)$, then reset it to this earlier time.





An Example

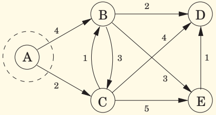
An Example



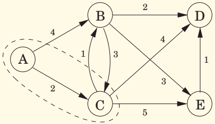
A: 0	D: ∞
B: 4	E: ∞
C: 2	

An Example

Shanghai Jiao Tong University

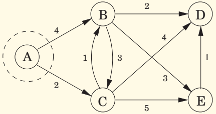


A: 0	D: ∞
B: 4	E: ∞
C: 2	

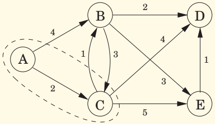


A: 0	D: 6
B: 3	E: 7
C: 2	

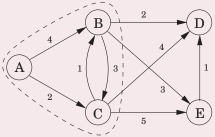
An Example



A: 0	D: ∞
B: 4	E: ∞
C: 2	

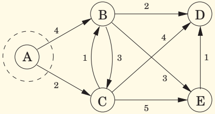


A: 0	D: 6
B: 3	E: 7
C: 2	

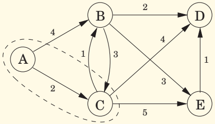


A: 0	D: 5
B: 3	E: 6
C: 2	

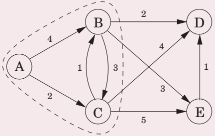
An Example



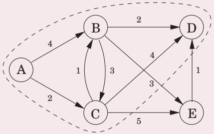
A: 0	D: ∞
B: 4	E: ∞
C: 2	



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



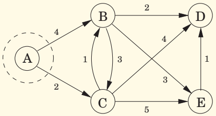
A: 0	D: 5
B: 3	E: 6
C: 2	



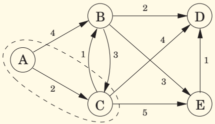
A: 0	D: 5
B: 3	E: 6
C: 2	

An Example

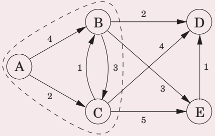
注意Dijkstra算法关注的是从起点到某个点的最短路径，所以dist的更新应该取决于从起点到这个点的最短距离，而不是从某个别的点到这个点的边的距离。



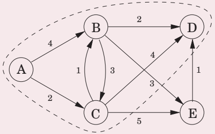
A: 0	D: ∞
B: 4	E: ∞
C: 2	



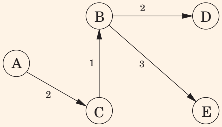
A: 0	D: 6
B: 3	E: 7
C: 2	



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算法只讨论正数权值边。而权值均为正数也保证了每一个点的最短路径找到之后不会再发生变化，即为全局的最短路径-->贪婪法

SHANGHAI JIAO TONG UNIVERSITY

```
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 Review: Priority Queue

Priority queue is a data structure usually implemented by heap.

Data Structure Review: Priority Queue

Priority queue is a data structure usually implemented by heap.

- Insert: Add a new element to the set.

Data Structure Review: 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 Review: 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 Review: 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.)

Insert：将一个节点插入优先队列中去，一般需要额外的一次重新排列
DecreaseKey：指的是一个点的dist更新之后其对应的key(就是dist)会变得更小，所以需要重新排列对应的位置。
deleteMin：由于每条边不再是等长的了，而为了获取最短路径，每次一定是要拿当前能拿到的最短边去继续处理。

Data Structure Review: 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.

1.最终要把队列变为空队列，所以一定需要V次deleteMin
2.对于E中的每一条边u->v，由于终点的v在还未被访问时
距离一定是∞，所以在初次访问的时候(若为无向图)则
一定会需要一次decreaseKey，所以共需要E次decreaseKey
3.makeQueue时候需要V次insert，所以总的时间复杂度就是
O(V+E)

Which Heap is Best

Implementation	deletemin	insert/decreasekey	$\|V\|\times\texttt{deletemin}+(\|V\|+\|E\|)\times\texttt{ insert}$
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\|+\|E\|)\log \|V\|)$
d-ary heap	$O(\frac{d\log \|V\|}{\log d})$	$O(\frac{\log \|V\|}{\log d})$	$O(\frac{(d\|V\|+\|E\|)\log \|V\|}{\log d})$
Fibonacci heap	$O(\log \|V\|)$	$O(1)$ (amortized)	$O(\|V\|\log \|V\|+\|E\|)$

d-ary heap中deletmin时候需要乘以一个d，是因为下一层是d叉树，每走一层需要额外的比较d次。
基于线性array的实现最好使用无序的数组。

Which heap is Best

A naive array implementation gives a respectable time complexity of $O(|V|^2)$, whereas with a binary heap we get $O((|V| + |E|) \log |V|)$. Which is preferable?

Which heap is Best

A naive array implementation gives a respectable time complexity of $O(|V|^2)$, whereas with a binary heap we get $O((|V| + |E|) \log |V|)$. Which is preferable?

This depends on whether the graph is sparse or dense.

Which heap is Best

SHANGHAI JIAO TONG UNIVERSITY

A naive array implementation gives a respectable time complexity of $O(|V|^2)$, whereas with a binary heap we get $O((|V| + |E|) \log |V|)$. Which is preferable?

This depends on whether the graph is sparse or dense.

- $|E|$ is less than $|V|^2$. If it is $\Omega(|V|^2)$, then clearly the array implementation is the faster.
- On the other hand, the binary heap becomes preferable as soon as $|E|$ dips below $|V|^2 / \log |V|$.
- The d-ary heap is a generalization of the binary heap and leads to a running time that is a function of d. The optimal choice is $d \approx |E|/|V|$;

Shortest Paths in the Presence of Negative Edges

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.

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 $\texttt{UPDATE}$ 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

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

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.

至多有V-1个边，是因为最多是把所有的点全部放进去，而如果存在一个路径长度大于V-1，则说明一定成环，若成环的回边>0,则一定不参与最短路径，若回边<0,只要不构成负环，就不会对回边终点的最短路径产生影响，所以直接忽略这样的回边，所以还是V-1条边。

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

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 ∈ V do
|   dist(u) = ∞;
|   prev(u) = nil;
end
dist[s] = 0;
repeat |V| − 1 times: for e ∈ E do
|   UPDATE(e);
end
```

因为对于每个点的最短路径长度最大为V-1，所以对于每条边最多更新V-1次就一定可以得到真正的最短路径以及最短路径长度。
但是还需要额外的考虑出现了负环(即成环的边权重和为负数)的情况。
而bellman-Ford算法另一个用处就是检查负环。只需要额外的迭代第V次，若各个边的w都不变则说明无负环，若变化了，说明存在负环，因为负环会导致之前已经处理过的节点的最短路径变小，从在最终的最短路径中会添加新的边。

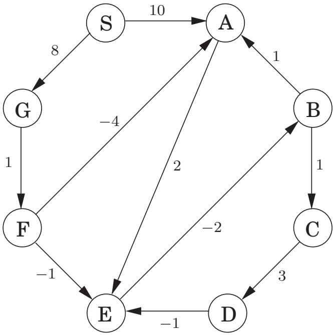
Bellman-Ford Algorithm

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.

做一次拓扑排序(recall:O(N))

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 | 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
dists = 0;
linearize G;
for each u ∈ V in linearized order do
    for all e ∈ E do
        UPDATE(e);
    end
end
```

之所以还需要每次更新所有的边，是因为这里的DAG还是允许负数权值边的

A Shortest-Path Algorithm for DAG

A Shortest-Path Algorithm for DAG

The scheme does not require edges to be positive.

A Shortest-Path Algorithm for DAG

SHANGHAI JIAO TONG UNIVERSITY

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.

Homework

Homework

Assignment 3. Exercises 3.7, 3.11, 3.28, 4.11, 4.12 and 4.16.