Algorithms and Theory of Computation

Lecture 7: Dijkstra Algorithm

Xiaohui Bei

MAS 714

September 3, 2018
Shortest Paths Problems

Given a (undirected or directed) weighted graph \( G = (V, E) \) with edge lengths (or weights). For edge \( e = (u, v) \), \( \ell(e) = \ell(u, v) \) is its length.

1. Given vertices \( s, t \), find a shortest path from \( s \) to \( t \).
2. Given vertex \( s \), find shortest paths from \( s \) to all other vertices.
3. Find shortest paths for all pair of vertices.

The length of a path is the sum of its edge weights.
Single-Source Shortest Path

Given a directed weighted graph $G = (V, E)$ with non-negative edge lengths (or weights). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given vertices $s, t$, find a shortest path from $s$ to $t$.
2. Given vertex $s$, find shortest paths from $s$ to all other vertices.

Focus on directed graphs.

- undirected graph problem can be reduced to directed graph problem

Uniform case: all edges lengths are 1.

- $\text{BFS}(s)$ solves the problem in $O(m + n)$ time.
Why does BFS work?

BFS\((s)\) explores nodes in increasing distance from \(s\).

Let \(\text{dist}(s, v)\) denote the shortest path length from \(s\) to \(v\).

Algorithm: \textbf{ShortestPathAlgorithm}(s):

- Initialize \(\text{dist}(s, v) = \infty\) for each vertex \(v \in V\);
- Initialize \(S = \emptyset\);
- \textbf{while} \(S \neq V\) \textbf{do}
  - Find vertex \(v \in V - S\) that is the closest to \(s\);
  - Update \(\text{dist}(s, v)\);
  - \(S = S \cup \{v\}\)

How to find the next closest vertex?
Finding the $i$th Closest Vertex

At the beginning of the $i$th iteration, $S$ already stores the $i - 1$ closest vertices to $s$.

What do we know about the $i$th closest vertex?

**Claim**

Let $P$ be a shortest path from $s$ to $v$ where $v$ is the $i$th closest vertex. Then all intermediate vertices in $P$ belong to $S$.

**Proof.**

Let $v'$ be an intermediate vertex in path $P$, then $\text{dist}(s, v') < \text{dist}(s, v)$. Thus $v' \in S$.

**Corollary**

At each step, the next closest vertex is adjacent to $S$. 
Finding the $i$th Closest Vertex

Lemma

If $s = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k = v$ is a shortest path from $s$ to $v$, then for any $1 \leq i < k$:

- $s = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_i$ is a shortest path from $s$ to $v_i$

Assume that $S$ contains the $i - 1$ closest vertices to $s$.
For each $u \in V - S$, let $\pi(u) = \min_{a \in S} (\text{dist}(s, a) + \ell(a, u))$

Lemma

If $u$ is the $i$th closest vertex to $s$, then $\pi(u) = \text{dist}(s, u)$.

Corollary

The $i$th closest vertex to $s$ is the vertex $u \in V - S$ such that $\pi(u) = \min_{v \in V - S} \pi(v)$.
Algorithm

Algorithm: ShortestPathAlgorithm(s):

Initialize $\pi(v) = \infty$ for each vertex $v \in V$;
Initialize $S = \emptyset$, $\pi(s) = 0$;
while $S \neq V$ do
  Find vertex $u \in V - S$ such that $\pi(u) = \min_{v \in V - S} \pi(v)$;
  $\text{dist}(s, u) = \pi(u)$;
  $S = S \cup \{u\}$;
  foreach vertex $v \in V - S$ do
    $\pi(v) = \min_{a \in S} (\text{dist}(s, a) + \ell(a, v))$;
Correctness: by induction using previous lemmas.
Running Time: $O(n(n + m))$ time.
  $n$ outer iterations. $O(m + n)$ time in each iteration.
Example

A network diagram with weighted edges between nodes. Each node is labeled with a letter (a, b, c, d, e, f, g, h, 0, 6, 13, 19, 25, 36, 38) and the weights on the edges are indicated next to the arrows connecting the nodes. The weights range from 6 to 30.
Critical Optimization For each unexplored vertex $v$, explicitly maintain $\pi(v)$ instead of computing directly from formula:

$$\pi(v) = \min_{u \in S} (\text{dist}(s, u) + \ell(u, v)).$$

- For each $v \notin S$, $\pi(v)$ can only decrease (because $S$ can only increase).
- More specifically, suppose $u$ is added to $S$ and there is an edge $(u, v)$ leaving $u$. Then, it suffices to update

$$\pi(v) = \min\{\pi(v), \text{dist}(s, u) + \ell(u, v)\}$$
Dijkstra’s Algorithm

1. eliminate $\pi(v)$ and let $\text{dist}(s, v)$ maintain it
2. update $\text{dist}$ values after adding $v$ by scanning edges out of $v$

Algorithm: $\text{Dijkstra}(s)$:

- Initialize $\text{dist}(s, v) = \infty$ for each vertex $v \in V$;
- Initialize $S = \emptyset, \text{dist}(s, s) = 0$;
- while $S \neq V$ do
  - Find vertex $u \in V - S$ such that $\text{dist}(s, u) = \min_{v \in V - S} \text{dist}(s, v)$;
  - $S = S \cup \{u\}$;
- foreach vertex $v \in \text{Adj}(u)$ do
  - $\text{dist}(s, v) = \min \{ \text{dist}(s, v), \text{dist}(s, u) + \ell(u, v) \}$;

How to maintain $\text{dist}$ values efficiently? Priority Queues
Priority Queues

Store a set $S$ of $n$ elements, where each element $v \in S$ has an associated real/integer key $k(v)$, with the following operations:

1. **Make-Queue**: create an empty queue
2. **Find-Min**: find the minimum key in $S$
3. **Extract-Min**: remove $v \in S$ with the smallest key and return it
4. **Add($v$, $k(v)$)**: add new element $v$ with key $k(v)$ to $S$
5. **Delete($v$)**: remove element $v$ from $S$
6. **Decrease-Key($v$, $k'(v)$)**: decrease key of $v$ from $k(v)$ to $k'(v)$

Decrease-Key is implemented via delete and insert.
Priority Queues

Applications:
- Prim’s MST algorithm
- Dijkstra’s shortest-path algorithm
- heapsort
- online median
- etc.

Implementations:
- binary heaps
- d-ary heaps
- binomial heaps
- Fibonacci heaps
Complete Binary Tree

A **binary tree** is a tree in which each vertex has at most two children.

A **complete tree** is a binary tree such that every level, except possibly the last level, is completely filled, and all vertices are as far left as possible.

### Property

The depth of a complete binary tree with $n$ vertices is $\lfloor \log_2 n \rfloor$.

**Proof.** The depth increases (by 1) only when $n$ is a power of 2.
Binary Tree in Nature
A **Binary Heap** is a complete binary tree such that for every element \( v \), at a vertex \( i \), the element \( w \) at \( i \)’s parent satisfies \( k(w) \leq k(v) \).
Explicit Implementation

**Pointer representation.** Vertex has a pointer to parent and two children.

- maintain number of elements \( n \)
- maintain pointer to root vertex

![Binary tree diagram]

Nanyang Technological University  
MAS 714  
September 3, 2018 16 / 24
Implicit Implementation

Array representation. Indices start at 1.

- take vertices in *level order*
- parent of vertex at $k$ is at $\lfloor k/2 \rfloor$
- children of vertex at $k$ are at $2k$ and $2k + 1$
Binary Heap: Insert

**Insert.** Add element in new vertex at end; repeatedly exchange new element with element in its parent until heap order is restored.

\[
\text{Time} = O(\log n)
\]

*Figure: swim up*
**Binary Heap: Extract-Min**

**Extract-Min.** Exchange element in root vertex with last vertex; repeatedly exchange element in root with its smaller child until heap order is restored.

![Binary Heap Diagram](image)

Figure: sink down

\[ \text{Time} = \mathcal{O}(\log n) \]
Delete. Exchange the element in this vertex with the last vertex; either swim up or sink down the vertex until heap order is restored.

- Time = $O(\log n)$

Decrease-Key. Repeatedly swim up the vertex until heap order is restored.

- Time = $O(\log n)$

Find-Min. Return the element in the root vertex.

- Time = $O(1)$
Binary Heap: Analysis

Dijkstra’s Algorithm with a priority queue
- $O(n)$ Insert operations
- $O(n)$ Extract-Min operations
- $O(m)$ Decrease-Key operations

Dijkstra’s Algorithm
Dijkstra’s algorithm can be implemented in $O((n + m) \log n)$ time.
Heapify. Given \( n \) elements, construct a binary heap containing them.

- Can be done in \( O(n \log n) \) time by inserting each element.

**Bottom-up Method.** For \( i = n \) to 1, repeatedly exchange the element in vertex \( i \) with its smaller child until subtree rooted at \( i \) is heap-ordered.
Theorem

Given \( n \) elements, a binary heap containing those \( n \) elements can be constructed in \( O(n) \) time.

Intuition:

- There are at most \( \lceil n/2^h + 1 \rceil \) vertices of height \( h \).
- The amount of work to sink a vertex is proportional to its height \( h \).
Dijkstra’s algorithm gives shortest paths from $s$ to all vertices in $V$.

Dijkstra’s algorithm can be implemented in $O(n \log n + m)$ time using Fibonacci Heaps. If $m = \Omega(n \log n)$, running time is linear in input size.

Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009)