CSE 101, Winter 2018

Design and Analysis of Algorithms

Lecture 6: BFS, SPs, Dijkstra

Class URL: http://vlsicad.ucsd.edu/courses/cse101-w18/
Distance in a Graph

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

E.g., \( d(S,B) = 2; \ d(S,D) = 1; \) etc.

Figure 4.2 A physical model of a graph.
Limitations of DFS

- From given starting node s, DFS finds
  - All nodes reachable from s
  - Explicit paths to these nodes (= a DFS tree)

- So, DFS determines whether a path exists between nodes in a graph
  - But it does not always find the shortest path

- DFS finds a path of length 5 from A to F, but the shortest A-F path has length 1.

*Explore(G,A)*
Breadth-First Search: The Idea

• Imagine vertices of the graph as balls and edges as strings tied to the balls.
• To find shortest paths: “lift the start vertex off the ground” → get layered view of the graph
• IDEA: Find vertices at distance 0, then 1, 2, etc.
• Suppose we’ve found all nodes at distance \( \leq d \)
• A node is at distance \( d+1 \) if:
  • it is adjacent to a node at distance \( d \)
  • it hasn’t been seen yet
Breadth-First Search Algorithm

procedure \text{BFS}(G,s)

Input: Graph $G = (V,E)$, (starting) vertex $s \in V$
Output: for each $u \in V$ reachable from $s$, $\text{dist}[u]$ is set to $u$'s distance from $s$

for each $u \in V$
    \text{dist}[u] \leftarrow \infty

\text{dist}[s] \leftarrow 0

Q \leftarrow [s] // queue containing just $s$

while Q is not empty
    $u \leftarrow \text{eject}(Q)$
    for each edge $(u,w) \in E$
        if \text{dist}[w] = \infty: // i.e., $w$ not seen yet
            \text{inject}(Q,w)
            \text{dist}[w] \leftarrow \text{dist}[u] + 1
## BFS Example

### BFS Tree

<table>
<thead>
<tr>
<th>Queue</th>
<th>d(A)</th>
<th>d(B)</th>
<th>d(C)</th>
<th>d(D)</th>
<th>d(E)</th>
<th>d(F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A]</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td><em>[B D F]</em></td>
<td>0</td>
<td>1</td>
<td>∞</td>
<td>1</td>
<td>∞</td>
<td>1</td>
</tr>
<tr>
<td>[D F C]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>∞</td>
<td>1</td>
</tr>
<tr>
<td>[F C E]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><em>[C E]</em></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>[E]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>*[ ]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

* = All nodes at level 0, 1, 2 (respectively, per each *) have been processed
Correctness of BFS by Induction

• **Statement:** For any distance \( d = 0, 1, 2, \ldots \) there is a point in time at which 3 conditions hold:
  – BFS has correctly set \( \text{dist}[ ] \) for all nodes at distance \( \leq d \)
  – All other nodes have \( \text{dist}[ ] = \infty \)
  – \( Q \) contains precisely the nodes at distance \( d \)

• **Base Case:** \( d = 0 \)
  – This holds immediately before the first iteration

• **General Case:** Assume true for \( d \); will show for \( d+1 \)
  – By induction hypothesis, there is a point in time at which our 3 conditions hold for level \( d \). Let \( Q \) be the state of the queue at that time.
  – Then the 3 conditions hold for level \( d+1 \) when all level \( d \) nodes have been ejected from the queue.
BFS Runtime Analysis

- Initializing the `dist[ ]` array takes time $O(|V|)$
- The outer while loop runs $|V|$ times, once per node $u \in V \text{ each node enqueued, dequeued exactly once}$
- The time taken on node $u$ is proportional to the degree of $u = \text{the number of edges out of } u$
- Therefore, the total runtime of the algorithm is

$$O(|V| + \sum_{u \in V} \text{deg}(u)) = O(|V| + |E|)$$

Food for Thought (already asked in Lecture 1): What would a graph search look like that is “in between” DFS and BFS?
Edge Lengths: Toward Dijkstra’s Algorithm

- BFS treats all edges as having the same length
  - But this is not true in many applications
- Notation: length or weight of $e = (u,w) \equiv l(e)$ or $l(u,w)$
- Sometimes, edge lengths / weights can be negative
  - Can you think of examples?
Extending BFS

• Suppose graph G has positive, integral edge lengths
• Simple trick: add dummy nodes to make G’ with unit-length edges
  – For ‘real nodes’, distances in G = distances in G’
  – We know how to run BFS on G’
• But BFS on G’ could waste a lot of time

→ Can we adapt BFS to ‘work’ only when there is something interesting to do?
“Alarm Clocks”

• Idea
  – Snooze through visits to dummy nodes
  – ‘Alarm’ should wake us up whenever something interesting is happening, i.e., when BFS encounters a **real** node

• Alarm for each **real** node = estimated time of arrival based on edges currently being traversed
  – \( T = 0 \): set alarms for A (100), B (200); snooze
  – \( T = 100 \): wake up, BFS is at A; set alarm for B (150); snooze
  – \( T = 150 \): wake up; done.
“Alarm Clock Algorithm”

• Set an alarm for node $s$ at time $T = 0$
• If the next alarm goes off at time $T$, for node $u$, then:
  – The shortest path distance to $u$ is $T$
  – For each edge $(u,w) \in E$
    • If no alarm has been set for $w$, then set an alarm at time $T + l(u,w)$
    • If an alarm has been set, but at a time later than $T + l(u,w)$, then move it to this earlier time

(1) Exactly simulates BFS on $G'$
(2) This is also known as Dijkstra’s algorithm
(3) What data structure helps implementation?
An Excursion on Shortest Paths
Algorithm Design Key Ideas (Shortest Paths)

• For SOME problems, the following property enables an efficient solution:
  
  “Principle of Optimality”: Any subsolution of an optimal solution is itself an optimal solution

• For such problems, solution strategies can include:

  “Tabulate (cache) Subproblem Solutions”: Avoid recomputation by creating a table of subproblem solutions

  “Relaxation / Successive Approximation”: Make multiple passes, each time solving a less restricted version of the original problem
  • Eventually, solve completely unrestricted = original problem instance
Shortest Path Problem Types

- Given a graph $G=(V,E)$ and $w: E \rightarrow \mathbb{R}$
  - (1 to 2) “s-t”: Find a shortest path from $s$ to $t$
  - (1 to all) “single-source” or “SSSP”: Find a shortest path from a source $s$ to every other vertex $v \in V$
  - (All to all) “all-pairs” or “APSP”: Find a shortest path from every vertex to every other vertex

- The weight or cost of path $v_i, ..., v_k = \sum l(v_i, v_{i+1})$

- Sometimes: no negative edges
  - Examples of “negative edges”: travel cost incentives, exothermic chemical reactions, unprofitable transactions in arbitrage, …

- Always: no negative cycles
  - Otherwise, the shortest-path problem isn’t well-defined
SSSP When Edges Have Positive Length

• **Condition 1:** $d_{i,j} > 0$
  - All edges have positive length
  - (Notation: $d_{i,j} =$ length of $(v_i,v_j)$ edge)

• **Condition 2:** $d_{i,j} + d_{j,k} < d_{i,k}$ for some $i,j,k$
  - Otherwise, shortest-path problem is trivial !!!
  - (All shortest paths would be direct edges from the source)
SSSP When Edges Have Positive Length

- **Condition 1:** $d_{i,j} > 0$
  - All edges have positive length
  - (Notation: $d_{i,j}$ = length of $(v_i, v_j)$ edge)

- **Condition 2:** $d_{i,j} + d_{j,k} < d_{i,k}$ for some $i,j,k$
  - Otherwise, shortest-path problem is trivial !!!
  - (All shortest paths would be direct edges from the source)

- **Fact 1:** Length of a path > length of any of its subpaths
  - (Because all edges have positive length)

Length($P_1$) > Length($P_x$), $x = 2, 3, 4$
SSSP When Edges Have Positive Length

- **Condition 1**: \( d_{i,j} > 0 \)
  - All edges have positive length
  - (Notation: \( d_{i,j} = \) length of \((v_i, v_j)\) edge)

- **Condition 2**: \( d_{i,j} + d_{j,k} < d_{i,k} \) for some \( i,j,k \)
  - Otherwise, shortest-path problem is trivial !!!
  - (All shortest paths would be direct edges from the source)

- **Fact 1**: Length of a path > length of any of its subpaths

- **Fact 2**: Any subpath of a shortest path is itself a shortest path

\[ \text{“Principle of Optimality”} \]

If there is a shorter path from \( v_i \) to \( v_j \), then we wouldn’t have had a shortest s-t path
SSSP When Edges Have Positive Length

- **Condition 1:** $d_{i,j} > 0$
  - All edges have positive length
  - (Notation: $d_{i,j} =$ length of $(v_i, v_j)$ edge)

- **Condition 2:** $d_{i,j} + d_{j,k} < d_{i,k}$ for some $i,j,k$
  - Otherwise, shortest-path problem is trivial !!!
  - (All shortest paths would be direct edges from the source)

- Fact 1: Length of a path > length of any of its subpaths
- Fact 2: Any subpath of a shortest path is itself a shortest path
- Fact 3: Any shortest path contains $\leq |V| - 1$ edges

By the Pigeonhole Principle
(Note: requires that there are no negative or zero-cost cycles)
Back to Dijkstra’s Algorithm
Shortest Paths – Dijkstra’s Algorithm

- **Scenario:** All shortest paths from $v_0 = \text{source}$ to other nodes are ordered by increasing length:
  - $|P_1| \leq |P_2| \leq \ldots \leq |P_{n-1}|
  - Index the nodes accordingly

- **What Dijkstra’s Algorithm Does:** Find $P_1$, then find $P_2$, etc.

- **Question:** How many edges are there in $P_1$?
  - **Exactly** 1 edge, else can find a subpath that is shorter

- **Question:** How many edges are there in $P_k$?
  - **At most** $k$ edges, else can find $k$ (shorter) subpaths, which would contradict the definition of $P_k$
Shortest Paths – Dijkstra’s Algorithm

- **Scenario:** All shortest paths from $v_0 = \text{source}$ to other nodes are ordered by increasing length:
  - $|P_1| \leq |P_2| \leq \ldots \leq |P_{n-1}|$
  - Index nodes accordingly

- **Algorithm Idea:** Find $P_1$, then find $P_2$, etc.

- **Fact 4:** $P_k$ contains $\leq k$ edges

- To find $P_1$: only look at 1-edge paths ($\min = P_1$)
- To find $P_2$: only look at 1- and 2-edge paths
  - BUT: need only consider 2-edge paths of form $d_{01} + d_{1i}$
  - Else more than one path shorter than $P_2$, a contradiction

Why is this picture impossible?
Section 4.4.2: Dijkstra’s Algorithm

- Terminology
  - **Permanent label**: Know **true** SP distance from \( v_0 \) to \( v_i \)
  - **Temporary label**: Know **restricted** SP distance from \( v_0 \) to \( v_i \) (going through only existing permanently-labeled nodes)

Set of **permanently labeled nodes** = “R” in the textbook

- Dijkstra’s Algorithm
  0. All vertices \( v_i, i = 1, \ldots, n-1 \), receive temporary labels \( l_i \) with value \( d_{0i} \)

**LOOP**:
  1. Among all temporary labels, pick \( l_k = \min_i l_i \) and change \( l_k \) to \( l_k^* \) (i.e., make vertex \( v_k \)’s label permanent)
     // stop if no temporary labels left
  2. Replace all temporary labels of \( v_k \)’s neighbors, using \( l_i \leftarrow \min (l_i, l_k^* + d_{ki}) \)
Example

Execute Dijkstra with S as source

LABELS: A B C D
Example

**LABELS:**

A  8  $\min(8, 2+\infty) = 8$  $\min(8, 3+4) = 7$  $\min(7, 4+1) = 5^*$

B  3  $\min(3, 2+\infty) = 3^*$

C  8  $\min(\infty, 2+2) = 4$  $4^*$

D  $2^*$  $(2^*)$
Figure 4.10: Single-Edge Extensions of SPs

- Find shortest path to \( v_{k+1} \) by extending by a single edge the shortest path to one of \( v_0, v_1, \ldots, v_k \)
Proof of Dijkstra Correctness
1-Slide Proof of Dijkstra

From Kevin Wayne slides for Kleinberg/Tardos Algorithm Design textbook

- Invariant. For each node \( u \in R \), \( d(u) \) is the length of the shortest \( s-u \) path.
- Let \( \pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e \) = shortest path cost to some \( u \) in the explored part (R), followed by a single edge.
- Proof. (by induction on \(|R|\))
- Base case: \(|R| = 1\) is trivial.
- Inductive hypothesis: Assume true for \(|R| = k \geq 1\).
  - Let \( v \) be next node added to \( R \), and let \( u-v \) be the chosen edge.
  - The shortest \( s-u \) path plus \( (u, v) \) is an \( s-v \) path of length \( \pi(v) \).
  - Consider any \( s-v \) path \( P \). We'll see that it's no shorter than \( \pi(v) \).
  - Let \( x-y \) be the first edge in \( P \) that leaves \( R \), and let \( P' \) be the subpath to \( x \).
  - Essentially, \( P \) is already too long, by the inequalities below.

\[
\ell(P) \geq \ell(P') + \ell(x,y) \geq d(x) + \ell(x,y) \geq \pi(y) \geq \pi(v)
\]

- nonnegative weights
- inductive hypothesis
- defn of \( \pi(y) \)
- Dijkstra chose \( v \) instead of \( y \)
Prim’s Algorithm vs. Dijkstra’s Algorithm

- **Prim:** Iteratively add edge \( e(u,v) \) to \( T \), such that \( u \in T, v \notin T \), and \( d_{u,v} \) is minimum
- **Dijkstra:** Iteratively add edge \( e(u,v) \) to \( T \), such that \( u \in T, v \notin T \), and \( l_u + d_{u,v} \) is minimum

- Both are building trees, in similar ways!
  - **Prim:** Minimum Spanning Tree
  - **Dijkstra:** Shortest Path Tree

- **INTERESTING QUESTION:** What kind of tree does the following algorithm build?

- “**Prim-Dijkstra**”: Iteratively add edge \( e(u,v) \) to \( T \), such that \( u \in T, v \notin T \), and \( c \cdot l_u + d_{u,v} \) is minimum \( \quad // 0 \leq c \leq 1 \)