COMP26120: Algorithms and Imperative Programming

Graph Algorithms
Lecture 3: Shortest paths
Lecture outline

- Single source shortest paths in a weighted graph;
- Dijkstra’s algorithm;
- The Bellman-Ford algorithm;
- Shortest path in directed acyclic graphs;
- All-pairs shortest paths;
Many systems in real world can be represented as weighted graphs (road/railway network, computer network, social networks, complex systems).

In general, a weighted graph is a graph that has a numeric label $w(e)$ associated with each edge $e$.

Edge weights represent a concept such as distance, connection costs, or affinity.
Single source shortest paths

![Graph with nodes and edges labeled with values]

Nodes: A, B, C, D, E, F, G

Edges with values:
- A to B: 4
- B to C: 5
- B to D: 6
- C to E: 7
- D to E: 8
- E to F: 2
- F to G: 1
- G to A: 0.5
Single source shortest paths

- Let $G$ be a weighted graph. The length $w(p)$ of a path $p = \{e_0, e_1, \ldots, e_n\}$ is defined as
  \[
  w(p) = \sum_{k=0}^{n} w(e_i)
  \]

- We need to ensure that all the weights in the graph are non-negative, so that no negative cycles in the graph exist.
- **The problem:** For a given weighted graph $G$ find the shortest paths from a selected node $v$ to all other nodes.
- For the case with no negative edge weights these exists a greedy strategy referred to as **Dijkstra’s algorithm**.
- Dijkstra’s algorithm is performing a weighted BFS starting at the node $v$. 
Dijkstra’s algorithm

- The greedy method iteratively grows a “cloud” of vertices out of $v$, with the vertices entering the cloud in based on increasing distances from $v$.
- At each iteration, the next chosen vertex the one outside the cloud that is closest to $v$.
- The algorithm terminates when no more vertices are outside the cloud, at which point we have a shortest path from $v$ to every other vertex of $G$.
- Our presentation will be for a graph that is undirected and has no self loops or parallel edges.
- We use the approximation to the true distance between the two nodes which is iteratively improved until its true value is obtained.
Dijkstra’s algorithm

• Greedy procedure deploys **edge relaxation**, where for each node \( u \neq v \) a label \( D[u] \) is assigned with an initial value \(+\infty\). During the course of the algorithm, this label holds current approximation of the shortest path length between \( u \) and \( v \).

• We define a cloud \( C \) of visited nodes to be initially an empty set.

• At each iteration we pull out a node with the smallest \( D[u] \) and update those \( D[z] \) for the nodes \( z \) that are adjacent to \( u \). This update should reflect the fact that there is a better way to get to \( z \) via the node \( u \) than previously determined.

\[
\text{if } D[u] + w(u, z) < D[z] \text{ then } \\
D[u] + w(u, z) \to D[z]
\]
Dijkstra’s algorithm

**Algorithm** DijkstraShortestPaths($G, v$);

**Input:** A simple undirected weighted graph $G$ with nonnegative edge weights, and a distinguished vertex $v$ of $G$;

**Output:** Labels $D[u]$ for all $u \in G$, representing the distances from $v$ to $u$;

- $D[v] = 0$; $D[u] = +\infty$ for all $u \neq v$ in $G$;
- Set a **priority queue** $Q$ to contain all vertices $u \in G$ using $D[u]$ as keys;

**while** (Q is not empty) **do**

- $u \leftarrow Q\text{.removeMin}()$; // pull a new vertex into the cloud

  **for** (each vertex $z$ adjacent to $u$ && $z \in Q$) **do**

  - if $D[u] + w(u, z) < D[z]$ **then**
    - $D[u] + w(u, z) \rightarrow D[z]$; // perform the relaxation on edge $(u, z)$

  **end if**

**end for**

**end while**

**return** the labels $D[u]$ of each vertex $u$
Dijkstra’s algorithm

- **Complexity analysis:** Denote by $n$ and $e$ the number of nodes and the number of edges in a graph $G$, respectively.
- The assumption is that edge weight additions and comparisons are the basic operations that take $O(1)$ time.
- We also assume that the graph $G$ is represented by an adjacency list. The cost of sweeping the vertices This data structure allows us to step through the vertices adjacent to $u$ during the relaxation step is proportional to their number.
- **How is the priority queue $Q$ implemented?** We can implement a PQ either as a sorted list or as a heap. In the latter case the cost of performing the method $Q\text{.removeMin()}$ is $O(\log(n))$ -- recall up/down heap bubbling.
Dijkstra’s algorithm

- The cost of inserting $n$ vertices in $Q$ with their initial key value is $O(n \cdot \log(n))$.
- At each of the $n$ iterations of the while loop we spend $O(\log(n))$ time performing $Q.removeMin()$ and $O(\deg(u) \cdot \log(n))$ time to perform the relaxation procedure on the edges incident on $u$.
- The overall running time of the while loop is
  \[
  \sum_{u \in G} (1 + \deg(u)) \cdot \log(n) = O((n + e) \cdot \log(n))
  \]
  since $\sum_{u \in G} \deg(u) = 2e$.
- If the PQ is implemented as an unsorted doubly linked list. Such implementation would result in $O(n^2)$ complexity (Homework: work out the details in Goodrich & Tamasia, p.406).
Dijkstra’s algorithm

Step 1: Remove A from PQ

Step 2: Remove E from PQ
Dijkstra’s algorithm

\[ D = \{0, 10\frac{1}{2}, 19\frac{1}{2}, 13\frac{1}{2}, 4, 5\frac{1}{2}, 5\} \]

Step 3: Remove G from PQ

\[
\begin{array}{cccc}
B & C & D & F \\
\hline
11 & +\infty & +\infty & 5.5
\end{array}
\]

Step 4: Remove F from PQ

\[
\begin{array}{ccc}
B & C & D \\
\hline
10.5 & +\infty & 13.5
\end{array}
\]

Step 5: Remove B from PQ

\[
\begin{array}{cc}
C & D \\
\hline
+\infty & 13.5
\end{array}
\]

Step 6: Remove D from PQ

\[
\begin{array}{c}
C \\
\hline
19.5
\end{array}
\]
The Bellman-Ford algorithm

- This algorithm finds the shortest paths in directed graphs which have negative edge weights.
- The Bellman-Ford algorithm shares the edge relaxation with Dijkstra’s algorithm, but does not use it in conjunction with the greedy method. Instead, it performs the relaxation of each edge in a digraph \(n - 1\) times, where \(n\) is the number of nodes.
- The shortest paths between the nodes are calculated in a bottom-up manner, where after each iteration \(k = 1, ..., n - 1\) we have the shortest paths of the length at most \(k\) between all pairs of nodes.
The Bellman-Ford algorithm

**Algorithm** BellmanFordShortestPaths \((G, v)\);

**Input:** A weighted directed graph \(G\) with \(n\) vertices, and a distinguished vertex \(v \in G\);

**Output:** Labels \(D[u]\) for all \(u \in G\), representing the distances from \(v\) to \(u\) or an indication that \(G\) has a negative-weight cycle;

\[D[v] = 0; \quad D[u] = +\infty \text{ for all } u \neq v \text{ in } G;\]

\[\text{for } i = 1: n - 1 \text{ do}\]

\[\text{for each directed edge } (u, z) \text{ outgoing from } u \text{ do}\]

\[\text{if } D[u] + w(u, z) < D[z] \text{ then}\]

\[D[u] + w(u, z) \rightarrow D[z]; \quad \text{// perform the relaxation on edge } (u, z)\]

\[\text{end if}\]

\[\text{end for}\]

\[\text{end for}\]

\[\text{if for any pair of nodes } (u, v) \text{ we have } D[v] > D[u] + w(u, v) \text{ then}\]

\[\text{return } G \text{ contains a negative-weight cycle};\]

\[\text{else}\]

\[\text{return } D[u] \text{ for all } u \in G;\]

\[\text{end if}\]
The Bellman-Ford algorithm

We process the edges in the following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)

Iteration 1:

After processing (B,E), (D,B), (B,D), (A,B)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>−1</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
</tr>
</tbody>
</table>

After processing (A,C)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>−1</td>
<td>4</td>
<td>+∞</td>
<td>+∞</td>
</tr>
</tbody>
</table>

After processing (D,C), (B,C), (E,D)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>−1</td>
<td>2</td>
<td>+∞</td>
<td>+∞</td>
</tr>
</tbody>
</table>

\( n = 5 \), the edges need to be processed 4 times
The Bellman-Ford algorithm

Iteration 2:

After processing (B,E), (D,B), (B,D), (A,B)

\[
\begin{array}{cccccc}
A & B & C & D & E \\
0 & -1 & 2 & 1 & 1 \\
\end{array}
\]

After processing (A,C), (D,C), (B,C), (E,D)

\[
\begin{array}{cccccc}
A & B & C & D & E \\
0 & -1 & 2 & -2 & 1 \\
\end{array}
\]

Iterations 3 and 4 do not produce any changes (check)

Homework: Change the weight -3 in the graph into -4 and rerun the algorithm.

\( n = 5 \), the edges need to be processed 4 times
The Bellman-Ford algorithm

- The complexity of the Bellman-Ford algorithm:
  - The main loop is performed \( n - 1 \) times.
  - During the loop all \( e \) edges are visited.
  - Each edge is processed in \( O(1) \) time.
- The complexity time for this algorithm is \( O(n \cdot e) \).