Lecture 6: Graph algorithms II

- Breadth-first search and shortest paths in graphs
- Shortest paths in weighted graphs: Dijkstra’s algorithm
6.1 Breadth-first search and shortest paths

- Recall: the distance from vertex $s$ to vertex $u$ in a (di)graph $G$ is the length of the shortest path in $G$ leading from $s$ to $u$.

- Breadth-first search (BFS) started at vertex $s$:

  ```
  for $d = 0, 1, 2, \ldots, \text{max\_dist}$ :
      visit all vertices at distance $d$ from $s$.
  ```

- Implementing this simple idea requires some bookkeeping, conveniently managed by a queue $Q$ of vertices:
  - Initially $Q = [s]$.
  - When vertex $v$ is visited (ejected from front of $Q$), all its so far undiscovered neighbours are injected to end of $Q$.

- Note that in this arrangement, $Q$ contains at all times vertices from (at most) two “layers” of $G$: distance $d$ from $s$ (fully discovered, being visited) and distance $d + 1$ from $s$ (being discovered, none yet visited).
BFS: exploration and search tree

Note that all the paths starting from the root \( s \) of a BFS search tree are shortest possible, i.e. it is a shortest-path tree.
BFS: the exploration algorithm

1 function BFS(G, s);

Input: Graph $G = (V, E)$, start vertex $s$
Output: for all vertices $u$ reachable from $s$, dist[$u$] is set to the distance from $s$ to $u$

2 for all $u \in V$ do dist[$u$] $\leftarrow$ $\infty$;
3 dist[$s$] $\leftarrow$ 0;
4 $Q \leftarrow$ [$s$];
5 while $Q$ is not empty do
6 \hspace{1em} $u \leftarrow$ EJECT($Q$);
7 \hspace{1em} for all edges $(u, v) \in E$ do
8 \hspace{2em} if dist[$v$] = $\infty$ then
9 \hspace{3em} INJECT($Q$, $v$);
10 \hspace{3em} dist[$v$] $\leftarrow$ dist[$u$] + 1;
11 \hspace{1em} end
12 end
13 end
BFS: correctness and complexity

- **Correctness (by induction on the following):**
  **Claim.** In an execution of algorithm BFS($G, s$) there is for each $d = 0, 1, 2, \ldots$, some moment at which:
  
  (i) for each vertex $v$ at distance $\leq d$ from $s$, the value $\text{dist}[v]$ is correctly set;
  
  (ii) for all other vertices $u$, $\text{dist}[u] = \infty$;
  
  (iii) the queue $Q$ contains exactly the vertices at distance $d$ from $s$.

- **Complexity (similarly as in DFS):**
  The running time of BFS($G, s$) is $O(|V| + |E|)$:
  
  - Each vertex is injected in $Q$ when it is discovered and ejected from $Q$ when visiting it is completed; for a total of $2|V|$ queue operations.
  
  - Each edge is examined once (in digraphs) or twice (in graphs); for a total of $O(|E|)$ processing time related to examining the edges.
6.2 Shortest paths in weighted graphs

- BFS determines shortest paths in graphs where all edges have the same length.
- This is of course not the case in many real-life applications, e.g. actual road networks:

![Graph with weights between cities]

- How to extend BFS to this case?
Adapting BFS to weighted graphs

- Consider a weighted graph \( G = (V, E, \ell) \), where all the edge weights ("lengths") \( \ell_e \) are positive integers.\(^1\)
- Shortest paths in \( G \) can in principle be computed by replacing weighted edges by sequences of unit-length ones and then running BFS on the resulting graph \( G' \):

This is conceptually correct, but of course not efficient, in particular for graphs with large \( \ell_e \)'s.

\(^1\)The length of edge \( e = (u, v) \) is denoted alternately by \( \ell_e \), \( \ell(u, v) \), \( \ell_{uv} \).
Alarm clocks

➢ In the previous extension of BFS to weighted graphs, most of the exploration consists of uneventful traversal over the dummy nodes:

➢ The process can be speeded up (and the dummy nodes eliminated) by associating to each vertex \( v \) an “alarm clock” indicating when some activity pertinent to \( v \) may next happen.

➢ The search algorithm then proceeds in the order of increasing alarm times, and for each alarm attends to the vertex \( u \) to which the alarm was associated.
The “alarm clock” BFS algorithm

Set $\text{dist}[s] \leftarrow \infty$ for all vertices $v$
Set $\text{alarm}[s] \leftarrow 0$ and $\text{alarm}[v] \leftarrow \bot$ for $v \neq s$
Repeat until $\text{alarm}[v] = \bot$ for all vertices $v$:

Say the next alarm is $\text{alarm}[u] = T$. Then:

Set $\text{dist}[u] \leftarrow T$
Set $\text{alarm}[u] \leftarrow \bot$
For each (out-)neighbour $v$ of $u$ in $G$:

If $\text{dist}[v] = \infty$, If $\text{alarm}[v] = \bot$ or $\text{alarm}[v] > T + \ell(u, v)$,
Set $\text{alarm}[v] \leftarrow T + \ell(u, v)$
Dijkstra’s algorithm and priority queues

- The well-known shortest-path algorithm for positive-weight networks by E. Dijkstra (1959) is essentially the “alarm-clock” method, with an efficient implementation for the system of alarms.

- The right data structure for this purpose is the priority queue (usually implemented as a heap), which supports the following operations on a set $H$ of (element, key) -value pairs:
  
  - $\text{INSERT}(H, (u, x))$: Add element $u$ with key value $x$ to set $H$.
  - $\text{DECREASEKEY}(H, (u, x'))$: Update the key associated to element $u$ to a new (lower) value $x'$.
  - $\text{DELETEMIN}(H)$: Return the element $u$ with the presently lowest key value contained in $H$, and remove $u$ from $H$.
  - $\text{MAKEQUEUE}(S)$: Arrange $S$, a set of elements and their associated key values, into a priority queue structure.

\[\text{Concrete implementations of this structure will be discussed at Lect. 15.}\]
Dijkstra’s shortest path algorithm (1/2)

1 function Dijkstra(G, s);

Input: Graph or digraph $G = (V, E)$ with positive edge lengths $\ell_e$, start vertex $s$

Output: For all vertices $u$ reachable from $s$, dist[$u$] is set to the distance from $s$ to $u$

2 for all $u \in V$ do
3    dist[$u$] $\leftarrow$ $\infty$;  
4    prev[$u$] $\leftarrow$ ⊥  \{Predecessor on shortest path from $s$\};
5 end
6 dist[s] $\leftarrow$ 0;

(Continued on next slide.)
Dijkstra’s shortest path algorithm (1/2)

1. $H \leftarrow \text{MAKEQUEUE}(\langle V, \text{dist} \rangle)$;
2. while $H$ is not empty do
   3. $u \leftarrow \text{DELETEMIN}(H)$;
   4. for all edges $(u, v) \in E$ do
      5. if $\text{dist}[v] > \text{dist}[u] + \ell(u, v)$ then
         6. $\text{dist}[v] \leftarrow \text{dist}[u] + \ell(u, v)$;
         7. $\text{prev}[v] \leftarrow u$;
         8. $\text{DECREASEKEY}(H, (v, \text{dist}[v]))$;
   9. end
10. end
11. end
Dijkstra’s algorithm: example (1/2)
Dijkstra’s algorithm: example (2/2)
Dijkstra’s algorithm: an alternative derivation (1/3)

- An alternative scheme for growing shortest paths from a given start vertex $s$ in a network with positive edge lengths:
  - Maintain a region $R$ of vertices to which distances and shortest paths from $s$ are known.
  - At each expansion step, add to $R$ that vertex $v$ outside of $R$ that is closest to $s$. 

![Diagram showing the known region $R$ and vertices $s$, $u$, and $v$.]
Dijkstra’s algorithm: an alternative derivation (2/3)

- How to identify the correct $v$?
  - Consider the shortest path from $s$ to $v$, and the vertex $u$ just preceding $v$ on this path.
  - Since $\ell_{uv} > 0$, it must be the case that $\text{dist}(s, u) < \text{dist}(s, v)$, and so $u$ is already in $R$. (For otherwise $v$ would not be the closest vertex outside of $R$.)
  - Thus, the next $v$ to be added to $R$ is one of the outside-$R$ neighbours of one of the inside-$R$ vertices $u$.

- But which $u \in R$, $v \not\in R$?
  - Well, the ones that minimise $\text{dist}(s, u) + \ell_{uv}$.
  - It is namely easy to see that this defines the shortest distance to this $v$ (for otherwise there would be another $u' \in R$, $v' \not\in R$ with smaller value of $\text{dist}(s, u') + \ell_{u'v'}$), and there cannot be another $w \not\in R$ with $\text{dist}(s, w) < \text{dist}(s, v)$ (by the same argument).
Dijkstra’s algorithm: an alternative derivation (3/3)

The preceding idea leads to the following algorithm scheme:

```
for all \( u \in V \) do dist[\( u \)] \( \leftarrow \) \( \infty \);
;
dist[s] \( \leftarrow \) 0;
\( R \leftarrow \emptyset \);
while \( R \neq V \) do
    pick node \( v \notin R \) with dist[\( v \)] = min;
    \( R \leftarrow R \cup \{v\} \);
    for all edges \((v, z) \in E\) do
        if dist[\( z \)] > dist[\( v \)] + \( \ell(v, z) \) then
            dist[\( z \)] \( \leftarrow \) dist[\( v \)] + \( \ell(v, z) \);
        end
    end
end
```

A proper implementation of this scheme is again D’s algorithm.
Dijkstra’s algorithm: complexity

- At an abstract level, Dijkstra’s algorithm corresponds to BFS, and so would have linear complexity.
- However, the priority queue operations are slower than the constant-time queue inject’s and eject’s of BFS.\(^3\)
- Since MakeQueue(\(V\)) takes at most as much time as \(|V|\) Insert operations, there are at most a total of:
  - \(|V|\) Insert operations
  - \(|V|\) DeleteMin operations
  - \(|E|\) DecreaseKey operations
- Using e.g. binary heaps as an implementation structure, these give an overall running time of \(O((|V| + |E|) \log |V|)\).

\(^3\)Implementation options will be discussed at Lecture 16.