Shortest Path Problems: Dijkstra, Bellman-Ford, and Floyd-Warshall
Duke COMPSCI 309s

Siyang Chen

Spring 2014
General Graph Search

Let $q$ be some sort of \textit{abstract queue object}, which supports the following two operations:

1. \textit{add}, which adds a node into $q$
2. \textit{popFirst}, which pops the ‘first’ node from $q$

Here the definition of ‘first’ depends on the specific queue implementation.
General Graph Search

Let $q$ be some sort of abstract queue object, which supports the following two operations:

1. $add$, which adds a node into $q$
2. $popFirst$, which pops the ‘first’ node from $q$

Here the definition of ‘first’ depends on the specific queue implementation.

Then we can formulate all graph search algorithms in the following manner:

- While $q$ is not empty:
  - $v \leftarrow q.popFirst()$
  - For all neighbours $u$ of $v$ such that $u \notin q$:
    - Add $u$ to $q$
General Graph Search

- While $q$ is not empty:
  - $v \leftarrow q.popFirst()$
  - For all neighbours $u$ of $v$ such that $u \not\in q$:
    - Add $u$ to $q$

By changing the behaviour of $q$, we recreate all the classical graph search algorithms:
General Graph Search

- While \( q \) is not empty:
  - \( v \leftarrow q.popFirst() \)
  - For all neighbours \( u \) of \( v \) such that \( u \not\in q \):
    - Add \( u \) to \( q \)

By changing the behaviour of \( q \), we recreate all the classical graph search algorithms:

- If \( q \) is a stack, then the algorithm becomes DFS.
General Graph Search

While $q$ is not empty:

$\quad v \leftarrow q.popFirst()$
$\quad$ For all neighbours $u$ of $v$ such that $u \notin q$:

$\quad$ Add $u$ to $q$

By changing the behaviour of $q$, we recreate all the classical graph search algorithms:

$\quad$ If $q$ is a stack, then the algorithm becomes DFS.
$\quad$ If $q$ is a standard FIFO queue, then the algorithm is BFS.
General Graph Search

- While \( q \) is not empty:
  - \( v \leftarrow q.popFirst() \)
  - For all neighbours \( u \) of \( v \) such that \( u \not\in q \):
    - Add \( u \) to \( q \)

By changing the behaviour of \( q \), we recreate all the classical graph search algorithms:

- If \( q \) is a stack, then the algorithm becomes DFS.
- If \( q \) is a standard FIFO queue, then the algorithm is BFS.
- If \( q \) is a priority queue, then the algorithm is Dijkstra.
- If \( q \) is a priority queue with a heuristic, then the algorithm is A*.
General Graph Search

- While \( q \) is not empty:
  - \( v \leftarrow q.popFirst() \)
  - For all neighbours \( u \) of \( v \) such that \( u \notin q \):
    - Add \( u \) to \( q \)

By changing the behaviour of \( q \), we recreate all the classical graph search algorithms:

- If \( q \) is a stack, then the algorithm becomes DFS.
- If \( q \) is a standard FIFO queue, then the algorithm is BFS.
- If \( q \) is a priority queue, then the algorithm is Dijkstra.
- If \( q \) is a priority queue with a heuristic, then the algorithm is A*.

Today we're going to focus on Dijkstra.
General Graph Search

- While $q$ is not empty:
  - $v \leftarrow q$.popFirst()
  - For all neighbours $u$ of $v$ such that $u \notin q$:
    - Add $u$ to $q$

By changing the behaviour of $q$, we recreate all the classical graph search algorithms:

- If $q$ is a stack, then the algorithm becomes DFS.
- If $q$ is a standard FIFO queue, then the algorithm is BFS.
- If $q$ is a priority queue, then the algorithm is Dijkstra.
- If $q$ is a priority queue with a heuristic, then the algorithm is A*.

Today we’re going to focus on Dijkstra.
Dijkstra’s Algorithm

Given a graph $G = (V, E)$ where edges have nonnegative lengths, and a source node $s \in V$, Dijkstra’s algorithm finds the shortest path from $s$ to every other node.
Dijkstra’s Algorithm

Given a graph $G = (V, E)$ where edges have nonnegative lengths, and a source node $s \in V$, Dijkstra’s algorithm finds the shortest path from $s$ to every other node.

A standard implementation of Dijkstra’s algorithm is the following:

- For all $v \in V$, $d_v \leftarrow \infty$
- $d_s \leftarrow 0$
- $q\text{.add}(s)$
- While $q$ is not empty:
  - $v \leftarrow q\text{.popFirst}()$
  - For all neighbours $u$ of $v$, such that $d_v + e(v, u) \leq d_u$:
    - $q\text{.remove}(u)$
    - $d_u \leftarrow d_v + e(v, u)$
    - $q\text{.add}(u)$

Runtime? $O(|E| + |V| \log |V|)$

We can also adapt the algorithm to store the shortest path itself.
Dijkstra’s Algorithm

Given a graph $G = (V, E)$ where edges have nonnegative lengths, and a source node $s \in V$, Dijkstra’s algorithm finds the shortest path from $s$ to every other node.

A standard implementation of Dijkstra’s algorithm is the following:

- For all $v \in V$, $d_v \leftarrow \infty$
- $d_s \leftarrow 0$
- $q.add(s)$
- While $q$ is not empty:
  - $v \leftarrow q.popFirst()$
  - For all neighbours $u$ of $v$, such that $d_v + e(v, u) \leq d_u$:
    - $q.remove(u)$
    - $d_u \leftarrow d_v + e(v, u)$
    - $q.add(u)$

Runtime?
Dijkstra’s Algorithm

Given a graph $G = (V, E)$ where edges have nonnegative lengths, and a source node $s \in V$, Dijkstra’s algorithm finds the shortest path from $s$ to every other node.

A standard implementation of Dijkstra’s algorithm is the following:

- For all $v \in V$, $d_v \leftarrow \infty$
- $d_s \leftarrow 0$
- $q.add(s)$
- While $q$ is not empty:
  - $v \leftarrow q.popFirst()$
  - For all neighbours $u$ of $v$, such that $d_v + e(v, u) \leq d_u$:
    - $q.remove(u)$
    - $d_u \leftarrow d_v + e(v, u)$
    - $q.add(u)$

Runtime? $O(|E| + |V| \log |V|)$
Dijkstra’s Algorithm

Given a graph $G = (V, E)$ where edges have nonnegative lengths, and a source node $s \in V$, Dijkstra’s algorithm finds the shortest path from $s$ to every other node.
A standard implementation of Dijkstra’s algorithm is the following:

- For all $v \in V$, $d_v \leftarrow \infty$
- $d_s \leftarrow 0$
- $q.add(s)$
- While $q$ is not empty:
  - $v \leftarrow q.popFirst()$
  - For all neighbours $u$ of $v$, such that $d_v + e(v, u) \leq d_u$:
    - $q.remove(u)$
    - $d_u \leftarrow d_v + e(v, u)$
    - $q.add(u)$

Runtime? $O(|E| + |V| \log |V|)$
We can also adapt the algorithm to store the shortest path itself.
Bellman-Ford

In the case when the graph may have negative edges, we must use the Bellman-Ford algorithm:

- For all \( v \in V \), \( d_v \leftarrow \infty \)
- \( d_s \leftarrow 0 \)
- For \( t \in \{1, \ldots, |V|\} \):
  - For all edges \((v, u)\):  
    - \( d_u \leftarrow \min\{d_u, d_v + e(v, u)\} \)
- If there is an edge \((v, u)\) such that \( d_v + e(v, u) < d_u \):
  - Throw \texttt{NEGATIVE\_CYCLE\_FOUND}
Bellman-Ford

In the case when the graph may have negative edges, we must use the Bellman-Ford algorithm:

- For all $v \in V$, $d_v \leftarrow \infty$
- $d_s \leftarrow 0$
- For $t \in \{1, \ldots, |V|\}$:
  - For all edges $(v, u)$:
    - $d_u \leftarrow \min\{d_u, d_v + e(v, u)\}$
  - If there is an edge $(v, u)$ such that $d_v + e(v, u) < d_u$:
    - Throw `NEGATIVE_CYCLE_FOUND`

Runtime?

\[ O(|E||V|) \]
Bellman-Ford

In the case when the graph may have negative edges, we must use the Bellman-Ford algorithm:

- For all $v \in V$, $d_v \leftarrow \infty$
- $d_s \leftarrow 0$
- For $t \in \{1, \ldots, |V|\}$:
  - For all edges $(v, u)$:
    - $d_u \leftarrow \min\{d_u, d_v + e(v, u)\}$
  - If there is an edge $(v, u)$ such that $d_v + e(v, u) < d_u$:
    - Throw `NEGATIVE_CYCLE_FOUND`

Runtime? $O(|E| \cdot |V|)$
In the case when the graph may have negative edges, we must use the Bellman-Ford algorithm:

1. For all \( v \in V \), \( d_v \leftarrow \infty \)
2. \( d_s \leftarrow 0 \)
3. For \( t \in \{1, \ldots, |V|\} \):
   - For all edges \((v, u)\):
     - \( d_u \leftarrow \min\{d_u, d_v + e(v, u)\} \)
   - If there is an edge \((v, u)\) such that \( d_v + e(v, u) < d_u \):
     - Throw `NEGATIVE_CYCLE_FOUND`

Runtime? \( O(|E| \cdot |V|) \)

One additional reason to use Bellman-Ford over Dijkstra is that it’s much easier to implement.
Lastly, suppose we want to find the shortest paths between each pair of nodes in graph:

- For all \( v \in V \), \( d_{u,v} \leftarrow \infty \)
- For all \((u, v) \in E\), \( d_{u,v} \leftarrow e(u, v) \)
- For \( w \in \{1, \ldots, |V|\} \):
  - For all pairs of nodes \((u, v)\):
    - \( d_{u,v} \leftarrow \min\{d_{u,v}, d_{u,w} + d_{w,v}\} \)
Floyd-Warshall

Lastly, suppose we want to find the shortest paths between each pair of nodes in graph:

1. For all $v \in V$, $d_{u,v} \leftarrow \infty$
2. For all $(u, v) \in E$, $d_{u,v} \leftarrow e(u, v)$
3. For $w \in \{1, \ldots, |V|\}$:
   - For all pairs of nodes $(u, v)$:
     - $d_{u,v} \leftarrow \min\{d_{u,v}, d_{u,w} + d_{w,v}\}$

Runtime?
Lastly, suppose we want to find the shortest paths between each pair of nodes in graph:

- For all $v \in V$, $d_{u,v} \leftarrow \infty$
- For all $(u, v) \in E$, $d_{u,v} \leftarrow e(u, v)$
- For $w \in \{1, \ldots, |V|\}$:
  - For all pairs of nodes $(u, v)$:
    - $d_{u,v} \leftarrow \min\{d_{u,v}, d_{u,w} + d_{w,v}\}$

Runtime? $O(|V|^3)$
Lastly, suppose we want to find the shortest paths between each pair of nodes in graph:

- For all $v \in V$, $d_{u,v} \leftarrow \infty$
- For all $(u, v) \in E$, $d_{u,v} \leftarrow e(u, v)$
- For $w \in \{1, \ldots, |V|\}$:
  - For all pairs of nodes $(u, v)$:
    - $d_{u,v} \leftarrow \min\{d_{u,v}, d_{u,w} + d_{w,v}\}$

Runtime? $O(|V|^3)$
Again, Floyd-Warshall is less efficient, but much easier to implement than all-pairs Dijkstra.