Testing Bipartiteness and Dijkstra's Algorithm

September 23, 2014

Computing All Connected Components

1. Pick an arbitrary node s in G.
2. Compute its connected component using BFS (or DFS).
3. Find a node (say v, not already visited) and repeat the BFS from v.
4. Repeat this process until all nodes are visited.

- Time spent to compute each component is

Computing All Connected Components

1. Pick an arbitrary node s in G.
2. Compute its connected component using BFS (or DFS).
3. Find a node (say v, not already visited) and repeat the BFS from v.
4. Repeat this process until all nodes are visited.

- Time spent to compute each component is linear in the size of the *component*.
- Running time of the algorithm is

Computing All Connected Components

1. Pick an arbitrary node s in G.
2. Compute its connected component using BFS (or DFS).
3. Find a node (say v, not already visited) and repeat the BFS from v.
4. Repeat this process until all nodes are visited.

- Time spent to compute each component is linear in the size of the *component*.
- Running time of the algorithm is linear in the total sizes of the components, i.e., $O(m + n)$.

Bipartite Graphs

- A graph $G = (V, E)$ is *bipartite* if V can be partitioned into two subsets X and Y such that every edge in E has one endpoint in X and one endpoint in Y.
  - $(X \times X) \cap E = \emptyset$ and $(Y \times Y) \cap E = \emptyset$.
  - Colour the nodes in X red and the nodes in Y blue. Then no edge in E connects nodes of the same colour.
- Examples of bipartite graphs:

Bipartite Graphs

- A graph $G = (V, E)$ is *bipartite* if V can be partitioned into two subsets X and Y such that every edge in E has one endpoint in X and one endpoint in Y.
  - $(X \times X) \cap E = \emptyset$ and $(Y \times Y) \cap E = \emptyset$.
  - Colour the nodes in X red and the nodes in Y blue. Then no edge in E connects nodes of the same colour.
- Examples of bipartite graphs: medical residents and hospitals, jobs and processors they can be scheduled on, professors and courses they can teach.

TestBipartiteness

INSTANCE: An undirected graph $G = (V, E)$

QUESTION: Is G bipartite?

Bipartite Graphs

- A graph $G = (V, E)$ is *bipartite* if V can be partitioned into two subsets X and Y such that every edge in E has one endpoint in X and one endpoint in Y.
  - $(X \times X) \cap E = \emptyset$ and $(Y \times Y) \cap E = \emptyset$.
  - Colour the nodes in X red and the nodes in Y blue. Then no edge in E connects nodes of the same colour.
- Examples of bipartite graphs: medical residents and hospitals, jobs and processors they can be scheduled on, professors and courses they can teach.

TestBipartiteness

INSTANCE: An undirected graph $G = (V, E)$

QUESTION: Is G bipartite?

- Is a triangle bipartite?

Bipartite Graphs

- A graph $G = (V, E)$ is *bipartite* if V can be partitioned into two subsets X and Y such that every edge in E has one endpoint in X and one endpoint in Y.
  - $(X \times X) \cap E = \emptyset$ and $(Y \times Y) \cap E = \emptyset$.
  - Colour the nodes in X red and the nodes in Y blue. Then no edge in E connects nodes of the same colour.
- Examples of bipartite graphs: medical residents and hospitals, jobs and processors they can be scheduled on, professors and courses they can teach.

TestBipartiteness

INSTANCE: An undirected graph $G = (V, E)$

QUESTION: Is G bipartite?

- Is a triangle bipartite? No.
- Generalisation: No cycle of odd length is bipartite.
- Claim: If a graph is bipartite, then it cannot contain a cycle of odd length.

Algorithm for Testing Bipartiteness

- Assume G is connected. Otherwise, apply the algorithm to each connected component separately.
- Idea: Pick an arbitrary node s and colour it red.

Algorithm for Testing Bipartiteness

- Assume G is connected. Otherwise, apply the algorithm to each connected component separately.
- Idea: Pick an arbitrary node s and colour it red. Colour all its neighbours blue.

Algorithm for Testing Bipartiteness

- Assume G is connected. Otherwise, apply the algorithm to each connected component separately.
- Idea: Pick an arbitrary node s and colour it red. Colour all its neighbours blue. Colour the uncoloured neighours of *these* nodes red, and so on till all nodes are coloured.

Algorithm for Testing Bipartiteness

- Assume G is connected. Otherwise, apply the algorithm to each connected component separately.
- Idea: Pick an arbitrary node s and colour it red. Colour all its neighbours blue. Colour the uncoloured neighours of *these* nodes red, and so on till all nodes are coloured. Check if very edge has endpoints of different colours.

Algorithm for Testing Bipartiteness

- Assume G is connected. Otherwise, apply the algorithm to each connected component separately.
- Idea: Pick an arbitrary node s and colour it red. Colour all its neighbours blue. Colour the uncoloured neighours of *these* nodes red, and so on till all nodes are coloured. Check if very edge has endpoints of different colours. Algorithm is just like BFS!

Algorithm for Testing Bipartiteness

- Assume G is connected. Otherwise, apply the algorithm to each connected component separately.
- Idea: Pick an arbitrary node s and colour it red. Colour all its neighbours blue. Colour the uncoloured neighours of *these* nodes red, and so on till all nodes are coloured. Check if very edge has endpoints of different colours. Algorithm is just like BFS!
- Algorithm:
  1. Run BFS on G. Maintain an additional array `Colour`.
  2. When we add a node v to a layer i, set `Colour`[v] to red if i is even, otherwise to blue.
  3. At the end of BFS, scan all the edges to check if there is any edge both of whose endpoints received the same colour.

Algorithm for Testing Bipartiteness

- Assume G is connected. Otherwise, apply the algorithm to each connected component separately.
- Idea: Pick an arbitrary node s and colour it red. Colour all its neighbours blue. Colour the uncoloured neighours of *these* nodes red, and so on till all nodes are coloured. Check if very edge has endpoints of different colours. Algorithm is just like BFS!
- Algorithm:
    1. Run BFS on G. Maintain an additional array `Colour`.
    2. When we add a node v to a layer i, set `Colour`$[v]$ to red if i is even, otherwise to blue.
    3. At the end of BFS, scan all the edges to check if there is any edge both of whose endpoints received the same colour.
- Running time of this algorithm is

Algorithm for Testing Bipartiteness

- Assume G is connected. Otherwise, apply the algorithm to each connected component separately.
- Idea: Pick an arbitrary node s and colour it red. Colour all its neighbours blue. Colour the uncoloured neighours of *these* nodes red, and so on till all nodes are coloured. Check if very edge has endpoints of different colours. Algorithm is just like BFS!
- Algorithm:
  1. Run BFS on G. Maintain an additional array `Colour`.
  2. When we add a node v to a layer i, set `Colour`$[v]$ to red if i is even, otherwise to blue.
  3. At the end of BFS, scan all the edges to check if there is any edge both of whose endpoints received the same colour.
- Running time of this algorithm is $O(n + m)$, since we do a constant amount of work per node in addition to the time spent by BFS.

Correctness of the Algorithm

1. If G is bipartite, the algorithm correctly says so.

Correctness of the Algorithm

1. If G is bipartite, the algorithm correctly says so.
2. If G is not bipartite, what is the proof?

Correctness of the Algorithm

1. If G is bipartite, the algorithm correctly says so.
2. If G is not bipartite, what is the proof? The algorithm can find a cycle of odd length in G.

Correctness of the Algorithm

1. If G is bipartite, the algorithm correctly says so.
2. If G is not bipartite, what is the proof? The algorithm can find a cycle of odd length in G.

- Let G be a graph and let $L_0, L_1, L_2, \ldots L_k$ be the layers produced by BFS, starting at node s. Then exactly one of the following statements is true:
  1. No edge of G joins two nodes in the same layer: then G is bipartite and nodes in even layers can be coloured red and nodes in odd layers can be coloured blue.
  2. There is an edge of G that joins two nodes in the same layer: then G contains a cycle of odd length and cannot be bipartite.

Correctness of the Algorithm

1. If G is bipartite, the algorithm correctly says so.
2. If G is not bipartite, what is the proof? The algorithm can find a cycle of odd length in G.

- Let G be a graph and let $L_0, L_1, L_2, \ldots L_k$ be the layers produced by BFS, starting at node s. Then exactly one of the following statements is true:
  1. No edge of G joins two nodes in the same layer: then G is bipartite and nodes in even layers can be coloured red and nodes in odd layers can be coloured blue.
  2. There is an edge of G that joins two nodes in the same layer: then G contains a cycle of odd length and cannot be bipartite.

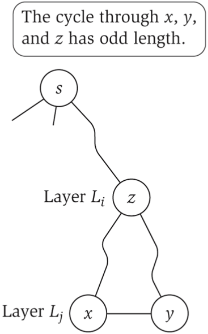

Figure 3.6 If two nodes x and y in the same layer are joined by an edge, then the cycle through x, y, and their lowest common ancestor z has odd length, demonstrating that the graph cannot be bipartite.

Shortest Path Problem

- $G(V, E)$ is a connected directed graph. Each edge e has a length $l_e \geq 0$.
- V has n nodes and E has m edges.
- *Length of a path* P is the sum of the lengths of the edges in P.
- Goal is to determine the shortest path from a specified start node s to each node in V.
- Aside: If G is undirected, convert to a directed graph by replacing each edge in G by two directed edges.

Shortest Path Problem

- $G(V, E)$ is a connected directed graph. Each edge e has a length $l_e \geq 0$.
- V has n nodes and E has m edges.
- *Length of a path* P is the sum of the lengths of the edges in P.
- Goal is to determine the shortest path from a specified start node s to each node in V.
- Aside: If G is undirected, convert to a directed graph by replacing each edge in G by two directed edges.

Shortest Paths

INSTANCE: A directed graph $G(V, E)$, a function $l : E \to R^+$, and a node $s \in V$

SOLUTION: A set $\{P_u, u \in V\}$, where P_u is the shortest path in G from s to u.

Example of Dijkstra's Algorithm

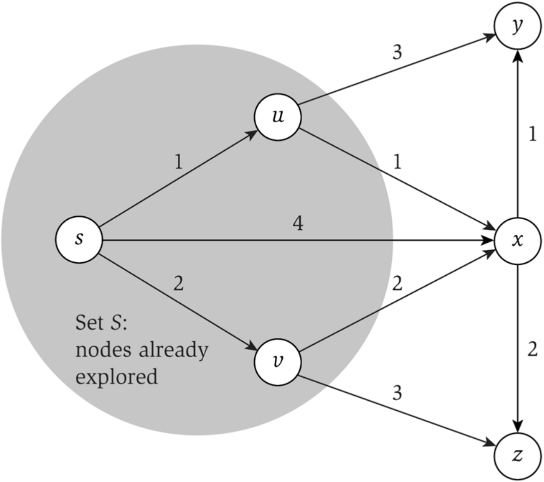

Figure 4.7 A snapshot of the execution of Dijkstra's Algorithm. The next node that will be added to the set S is x, due to the path through u.

Dijkstra's Algorithm

- Maintain a set S of explored nodes.
- For each node $u \in S$, we store a distance $d(u)$, which (we will prove) is the length of the shortest path from s to u.

Dijkstra's Algorithm

- Maintain a set S of explored nodes.
- For each node $u \in S$, we store a distance $d(u)$, which (we will prove) is the length of the shortest path from s to u.
- For each node $v \notin S$, we store a value $d'(v)$, which is the length of the shortest path from s to v using only nodes in S (and v, of course).
- "Greedily" add a node v to S that is closest to s.

Dijkstra's Algorithm

- Maintain a set S of explored nodes.
- For each node $u \in S$, we store a distance $d(u)$, which (we will prove) is the length of the shortest path from s to u.
- For each node $v \notin S$, we store a value $d'(v)$, which is the length of the shortest path from s to v using only nodes in S (and v, of course).
- "Greedily" add a node v to S that is closest to s.

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
      For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
      Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
      Add v to S and set d(v) = d'(v)
  Endwhile
```

- $v = \arg\min_{x \notin S} d'(x)$ means v is the node that minimises the distance $d'()$ over all nodes $x \notin S$.

Dijkstra's Algorithm

- Maintain a set S of explored nodes.
- For each node $u \in S$, we store a distance $d(u)$, which (we will prove) is the length of the shortest path from s to u.
- For each node $v \notin S$, we store a value $d'(v)$, which is the length of the shortest path from s to v using only nodes in S (and v, of course).
- "Greedily" add a node v to S that is closest to s.

Dijkstra's Algorithm(G, l):
  Initialize $S = \{s\}$ and $d(s) = 0$
  While $S \neq V$
    For each node $x \notin S$ compute $d'(x) = \min_{e=(u,x):u \in S}(d(u) + l_e)$
    Select a node $v \notin S$ such that $v = \arg\min_{x \notin S} d'(x)$
    Add v to S and set $d(v) = d'(v)$
  Endwhile

- $v = \arg\min_{x \notin S} d'(x)$ means v is the node that minimises the distance $d'()$ over all nodes $x \notin S$.
- To compute the shortest paths:

Dijkstra's Algorithm

- Maintain a set S of explored nodes.
- For each node $u \in S$, we store a distance $d(u)$, which (we will prove) is the length of the shortest path from s to u.
- For each node $v \notin S$, we store a value $d'(v)$, which is the length of the shortest path from s to v using only nodes in S (and v, of course).
- "Greedily" add a node v to S that is closest to s.

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
      For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
      Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
      Add v to S and set d(v) = d'(v)
  Endwhile
```

- $v = \arg\min_{x \notin S} d'(x)$ means v is the node that minimises the distance $d'()$ over all nodes $x \notin S$.
- To compute the shortest paths: when we add v to S, store the predecessor u that minimises $d'(v)$.

Proof of Correctness

- Let P_u be the shortest path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.

Proof of Correctness

- Let P_u be the shortest path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive hypothesis:

Proof of Correctness

- Let P_u be the shortest path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive hypothesis: $d(u)$ is correct for all nodes $u \in S$.

Proof of Correctness

- Let P_u be the shortest path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive hypothesis: $d(u)$ is correct for all nodes $u \in S$.
  - Inductive step: we add the node v to S. Let u be the v's predecessor on the path P_v. Could there be a shorter path P from s to v?

Proof of Correctness

- Let P_u be the shortest path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive hypothesis: $d(u)$ is correct for all nodes $u \in S$.
  - Inductive step: we add the node v to S. Let u be the v's predecessor on the path P_v. Could there be a shorter path P from s to v?

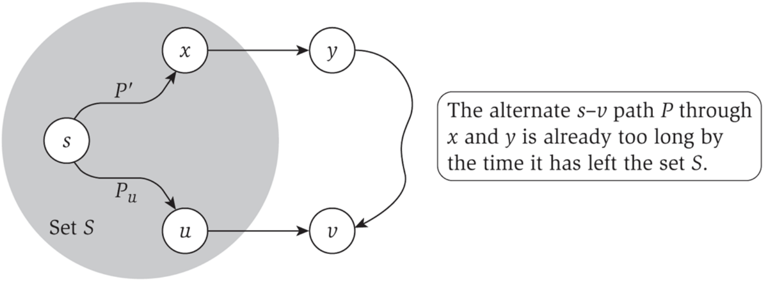

Figure 4.8 The shortest path P_v and an alternate s-v path P through the node y.

Comments about Dijkstra's Algorithm

- Algorithm cannot handle negative edge lengths. We will discuss the Bellman-Ford algorithm in a few weeks.
- Union of shortest paths output by Dijkstra's algorithm forms a tree. Why?

Comments about Dijkstra's Algorithm

- Algorithm cannot handle negative edge lengths. We will discuss the Bellman-Ford algorithm in a few weeks.
- Union of shortest paths output by Dijkstra's algorithm forms a tree. Why?
- Union of shortest paths from a fixed source s forms a tree; paths not necessarily computed by Dijkstra's algorithm.

Comments about Dijkstra's Algorithm

- Algorithm cannot handle negative edge lengths. We will discuss the Bellman-Ford algorithm in a few weeks.
- Union of shortest paths output by Dijkstra's algorithm forms a tree. Why?
- Union of shortest paths from a fixed source s forms a tree; paths not necessarily computed by Dijkstra's algorithm.
  - P_v: shortest path from s to a node v, $d(v)$: length of P_v.

Comments about Dijkstra's Algorithm

- Algorithm cannot handle negative edge lengths. We will discuss the Bellman-Ford algorithm in a few weeks.
- Union of shortest paths output by Dijkstra's algorithm forms a tree. Why?
- Union of shortest paths from a fixed source s forms a tree; paths not necessarily computed by Dijkstra's algorithm.
  - P_v: shortest path from s to a node v, $d(v)$: length of P_v.
  - If u is the second-to-last node on P_v, then $d(v) = d(u) + l_{(u,v)}$.

Comments about Dijkstra's Algorithm

- Algorithm cannot handle negative edge lengths. We will discuss the Bellman-Ford algorithm in a few weeks.
- Union of shortest paths output by Dijkstra's algorithm forms a tree. Why?
- Union of shortest paths from a fixed source s forms a tree; paths not necessarily computed by Dijkstra's algorithm.
  - P_v: shortest path from s to a node v, $d(v)$: length of P_v.
  - If u is the second-to-last node on P_v, then $d(v) = d(u) + l_{(u,v)}$.
  - If u precedes w on P_v, then $d(w) = d(u) + l_{(u,w)}$, i.e., $d(w) - d(u) = l_{(u,w)}$.

Comments about Dijkstra's Algorithm

- Algorithm cannot handle negative edge lengths. We will discuss the Bellman-Ford algorithm in a few weeks.
- Union of shortest paths output by Dijkstra's algorithm forms a tree. Why?
- Union of shortest paths from a fixed source s forms a tree; paths not necessarily computed by Dijkstra's algorithm.
  - P_v: shortest path from s to a node v, $d(v)$: length of P_v.
  - If u is the second-to-last node on P_v, then $d(v) = d(u) + l_{(u,v)}$.
  - If u precedes w on P_v, then $d(w) = d(u) + l_{(u,w)}$, i.e., $d(w) - d(u) = l_{(u,w)}$.
  - Suppose union of shortest paths from s contains a cycle involving nodes $v_1, v_2, \ldots v_k$ in that order around the cycle.

Comments about Dijkstra's Algorithm

- Algorithm cannot handle negative edge lengths. We will discuss the Bellman-Ford algorithm in a few weeks.
- Union of shortest paths output by Dijkstra's algorithm forms a tree. Why?
- Union of shortest paths from a fixed source s forms a tree; paths not necessarily computed by Dijkstra's algorithm.
  - P_v: shortest path from s to a node v, $d(v)$: length of P_v.
  - If u is the second-to-last node on P_v, then $d(v) = d(u) + l_{(u,v)}$.
  - If u precedes w on P_v, then $d(w) = d(u) + l_{(u,w)}$, i.e., $d(w) - d(u) = l_{(u,w)}$.
  - Suppose union of shortest paths from s contains a cycle involving nodes $v_1, v_2, \ldots v_k$ in that order around the cycle.

$$d(v_i) - d(v_{i-1}) = l(v_{i-1}, v_i), \text{ for each } 2 \leq i \leq k$$
$$d(v_1) - d(v_k) = l(v_k, v_1)$$

Comments about Dijkstra's Algorithm

- Algorithm cannot handle negative edge lengths. We will discuss the Bellman-Ford algorithm in a few weeks.
- Union of shortest paths output by Dijkstra's algorithm forms a tree. Why?
- Union of shortest paths from a fixed source s forms a tree; paths not necessarily computed by Dijkstra's algorithm.
  - P_v: shortest path from s to a node v, $d(v)$: length of P_v.
  - If u is the second-to-last node on P_v, then $d(v) = d(u) + l_{(u,v)}$.
  - If u precedes w on P_v, then $d(w) = d(u) + l_{(u,w)}$, i.e., $d(w) - d(u) = l_{(u,w)}$.
  - Suppose union of shortest paths from s contains a cycle involving nodes $v_1, v_2, \ldots v_k$ in that order around the cycle.

$$d(v_i) - d(v_{i-1}) = l(v_{i-1}, v_i), \text{ for each } 2 \leq i \leq k$$

$$d(v_1) - d(v_k) = l(v_k, v_1)$$

$$\sum_{i=2}^{k} \left(d(v_i) - d(v_{i-1})\right) + d(v_1) - d(v_k) = \sum_{i=2}^{k} l(v_{i-1}, v_i) + l(v_k, v_1)$$

Comments about Dijkstra's Algorithm

- Algorithm cannot handle negative edge lengths. We will discuss the Bellman-Ford algorithm in a few weeks.
- Union of shortest paths output by Dijkstra's algorithm forms a tree. Why?
- Union of shortest paths from a fixed source s forms a tree; paths not necessarily computed by Dijkstra's algorithm.
  - P_v: shortest path from s to a node v, $d(v)$: length of P_v.
  - If u is the second-to-last node on P_v, then $d(v) = d(u) + l_{(u,v)}$.
  - If u precedes w on P_v, then $d(w) = d(u) + l_{(u,w)}$, i.e., $d(w) - d(u) = l_{(u,w)}$.
  - Suppose union of shortest paths from s contains a cycle involving nodes $v_1, v_2, \ldots v_k$ in that order around the cycle.

$$d(v_i) - d(v_{i-1}) = l(v_{i-1}, v_i), \text{ for each } 2 \leq i \leq k$$

$$d(v_1) - d(v_k) = l(v_k, v_1)$$

$$\sum_{i=2}^{k} \left(d(v_i) - d(v_{i-1})\right) + d(v_1) - d(v_k) = \sum_{i=2}^{k} l(v_{i-1}, v_i) + l(v_k, v_1)$$

$$0 = \sum_{i=2}^{k} l(v_{i-1}, v_i) + l(v_k, v_1)$$

Implementing Dijkstra's Algorithm

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
      For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
      Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
      Add v to S and set d(v) = d'(v)
  Endwhile
```

- How many iterations are there of the while loop?

Implementing Dijkstra's Algorithm

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
     For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
     Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
     Add v to S and set d(v) = d'(v)
  Endwhile
```

- How many iterations are there of the while loop? $n - 1$.

Implementing Dijkstra's Algorithm

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
      For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
      Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
      Add v to S and set d(v) = d'(v)
  Endwhile
```

- How many iterations are there of the while loop? $n-1$.
- In each iteration, for each node $x \notin S$, compute

$$d'(x) = \min_{e=(u,x), u\in S} d(u) + l_e$$

Implementing Dijkstra's Algorithm

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
     For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
     Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
     Add v to S and set d(v) = d'(v)
  Endwhile
```

- How many iterations are there of the while loop? $n-1$.
- In each iteration, for each node $x \notin S$, compute

$$d'(x) = \min_{e=(u,x), u \in S} d(u) + l_e$$

- Running time per iteration is

Implementing Dijkstra's Algorithm

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
     For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
     Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
     Add v to S and set d(v) = d'(v)
  Endwhile
```

- How many iterations are there of the while loop? $n - 1$.
- In each iteration, for each node $x \notin S$, compute

$$d'(x) = \min_{e=(u,x), u \in S} d(u) + l_e$$

- Running time per iteration is $O(m)$, yielding an overall running time of $O(nm)$.

A Faster implementation of Dijkstra's Algorithm

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
      For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
      Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
      Add v to S and set d(v) = d'(v)
  Endwhile
```

- Observation: If we add v to S, $d'(x)$ changes only if (v, x) is an edge in G.

A Faster implementation of Dijkstra's Algorithm

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
     For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
     Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
     Add v to S and set d(v) = d'(v)
  Endwhile
```

- Observation: If we add v to S, $d'(x)$ changes only if (v, x) is an edge in G.
- Store the pairs $(x, d'(x))$ for each node $x \in V - S$ in a priority queue, with $d'(x)$ as the key.
- Determine the next node v to add to S using ExtractMin.
- After adding v to S, for each node w such that (v, w) is an edge in G, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$,
  1. Set $d'(w) = d(v) + l_{(v,w)}$.
  2. Update w's key to the new value of $d'(w)$ using ChangeKey.

A Faster implementation of Dijkstra's Algorithm

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
     For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
     Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
     Add v to S and set d(v) = d'(v)
  Endwhile
```

- Observation: If we add v to S, $d'(x)$ changes only if (v, x) is an edge in G.
- Store the pairs $(x, d'(x))$ for each node $x \in V - S$ in a priority queue, with $d'(x)$ as the key.
- Determine the next node v to add to S using ExtractMin.
- After adding v to S, for each node w such that (v, w) is an edge in G, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$,
  1. Set $d'(w) = d(v) + l_{(v,w)}$.
  2. Update w's key to the new value of $d'(w)$ using ChangeKey.
- How many times are ExtractMin and ChangeKey invoked?

A Faster implementation of Dijkstra's Algorithm

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
      For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
      Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
      Add v to S and set d(v) = d'(v)
  Endwhile
```

- Observation: If we add v to S, $d'(x)$ changes only if (v, x) is an edge in G.
- Store the pairs $(x, d'(x))$ for each node $x \in V - S$ in a priority queue, with $d'(x)$ as the key.
- Determine the next node v to add to S using ExtractMin.
- After adding v to S, for each node w such that (v, w) is an edge in G, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$,
  1. Set $d'(w) = d(v) + l_{(v,w)}$.
  2. Update w's key to the new value of $d'(w)$ using ChangeKey.
- How many times are ExtractMin and ChangeKey invoked? $n - 1$ and m

A Faster implementation of Dijkstra's Algorithm

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  While S ≠ V
      For each node x ∉ S compute d'(x) = min_{e=(u,x):u∈S}(d(u) + l_e)
      Select a node v ∉ S such that v = arg min_{x∉S} d'(x)
      Add v to S and set d(v) = d'(v)
  Endwhile
```

- Observation: If we add v to S, $d'(x)$ changes only if (v, x) is an edge in G.
- Store the pairs $(x, d'(x))$ for each node $x \in V - S$ in a priority queue, with $d'(x)$ as the key.
- Determine the next node v to add to S using ExtractMin.
- After adding v to S, for each node w such that (v, w) is an edge in G, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$,
  1. Set $d'(w) = d(v) + l_{(v,w)}$.
  2. Update w's key to the new value of $d'(w)$ using ChangeKey.
- How many times are ExtractMin and ChangeKey invoked? $n - 1$ and m

Faster Dijkstra's Algorithm

```
Dijkstra's Algorithm(G, l):
  Initialize S = {s} and d(s) = 0
  Insert (s, 0) into a priority queue Q.
  While S ≠ V
      (v, k) = ExtractMin(Q)
      Add v to S and set d(v) = k
      For each node w such that e = (v, w) is an edge in G
         If d(v) + l_e < d'(w),
         Set d'(w) = d(v) + l_e)
         ChangeKey(Q, w, d'(w)).
  Endwhile
```