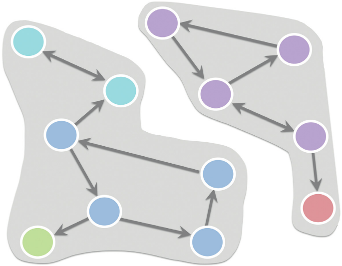
CS 4884: Components and Shortest Paths

T. M. Murali

February 15 and 17, 2022



Results of Poll

Would you like me to discuss the algorithm to compute shortest paths in ***unweighted*** graphs?

True	9 respondents	**69 %**
False	4 respondents	31 %

Would you like me to discuss the algorithm to compute shortest paths in ***weighted*** graphs?

True	11 respondents	**85 %**
False	2 respondents	15 %

Results of Poll

We can use **depth-first** search to compute the shortest path from one node to all nodes in an ***unweighted directed*** graph in time proportional to the size of the graph.

True	8 respondents	**62 %**	✓
False	5 respondents	38 %	

Results of Poll

We can use [algorithmname] to compute the shortest path from one node to all nodes in a ***weighted directed*** graph in [runningtime] time. You can assume that the graph has n nodes and m edges.

algorithmname runningtime

BFS	1 respondent	8 %	✓
Dijkstra's algorithm	2 respondents	15 %	✓
Prim's algorithm		0 %	✓
DFS	1 respondent	8 %	✓
Something Else	8 respondents	62 %	
No Answer	1 respondent	8 %	

We can use [algorithmname] to compute the shortest path from one node to all nodes in a ***weighted directed*** graph in [runningtime] time. You can assume that the graph has n nodes and m edges.

algorithmname **runningtime**

O(m^2)		0 %	✓
O(mn)		0 %	✓
O(m log n)		0 %	✓
O(log m)		0 %	✓
Something Else	11 respondents	85 %	
No Answer	2 respondents	15 %	

Summary of Course Thus Far

- History of neuroscience
- Graphs (Definitions, basic concepts, Euler tours)
- Brain graphs (types of nodes and edges, experimental methods, Chapter 2)
- Brain connectivity matrices and node degrees (Chapters 3 and 4)
- Clustering coefficient and small world networks (Chapter 8.2)

Plan till Spring Break

- Clustering coefficient is a local measure of graph density.
- Small world measures capture global features of graphs.

Plan till Spring Break

- Clustering coefficient is a local measure of graph density.
- Small world measures capture global features of graphs.

Are there intermediate notions of graph density?

- Subgraphs that represent backbones of network topology (components, shortest paths, Chapter 6.1, 7.1, 7.2, February 15 and 17)
- Cores and Modularity (Chapter 6.2, 9.1, February 22, 24, March 1)

Plan till Spring Break

- Clustering coefficient is a local measure of graph density.
- Small world measures capture global features of graphs.

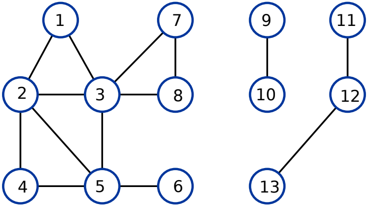
Are there intermediate notions of graph density?

- Subgraphs that represent backbones of network topology (components, shortest paths, Chapter 6.1, 7.1, 7.2, February 15 and 17)
- Cores and Modularity (Chapter 6.2, 9.1, February 22, 24, March 1)
- Describe group projects (March 3).

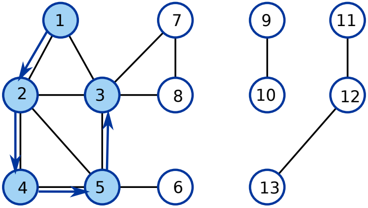
Plan after Spring Break

- Schedule meetings with project groups during class time in my office.
- Number of meetings will depend on number of groups.
- Poster preparation for VTURCS Symposium on April ??.

Paths and Connectivity

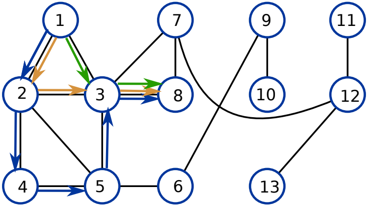


Paths and Connectivity



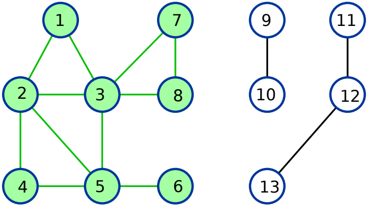
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.

Paths and Connectivity



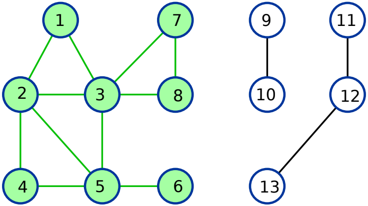
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- *Distance $d(u, v)$* between two nodes u and v is the minimum number of edges in any u-v path. Abuse of notation: d for both degree and distance.

Paths and Connectivity



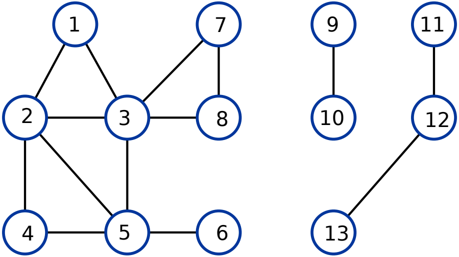
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- *Distance $d(u, v)$* between two nodes u and v is the minimum number of edges in any u-v path. Abuse of notation: d for both degree and distance.
- A *connected component* of G is a subgraph $H = (V', E')$ of G such
  - for every pair of nodes u, v in V' there is a u-v path in H, i.e., that uses only the edges in E' and

Paths and Connectivity



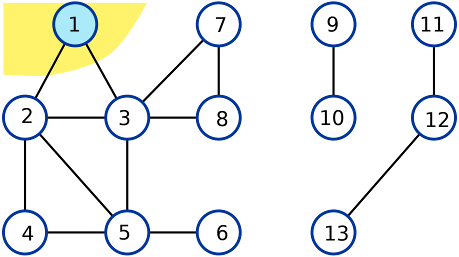
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- *Distance $d(u, v)$* between two nodes u and v is the minimum number of edges in any u-v path. Abuse of notation: d for both degree and distance.
- A *connected component* of G is a subgraph $H = (V', E')$ of G such
  - for every pair of nodes u, v in V' there is a u-v path in H, i.e., that uses only the edges in E' and
  - H is *maximal*, i.e., for every node $x \in V - V'$, there is no path in G between x and any node in V'.

Breadth-First Search (BFS)



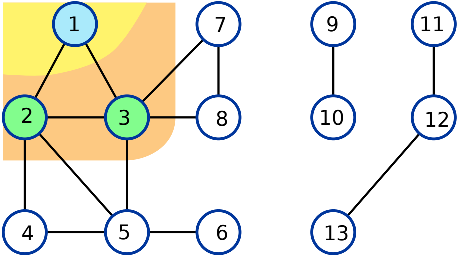
- Use BFS to compute connected component containing a node s.
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.

Breadth-First Search (BFS)



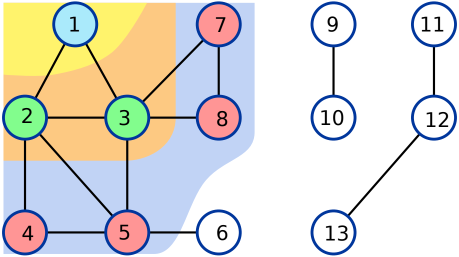
- Use BFS to compute connected component containing a node s.
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.

Breadth-First Search (BFS)



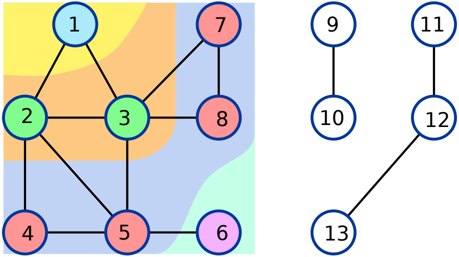
- Use BFS to compute connected component containing a node s.
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.
- Layer L_1 contains all neighbours of s.

Breadth-First Search (BFS)



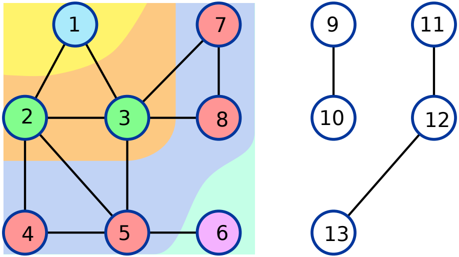
- Use BFS to compute connected component containing a node s.
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.
- Layer L_1 contains all neighbours of s.
- Given layers $L_0, L_1, \ldots, L_j$, layer L_{j+1} contains all nodes that
  1. do not belong to an earlier layer and
  2. are connected by an edge to a node in layer L_j.

Breadth-First Search (BFS)



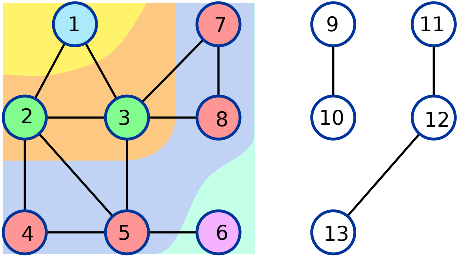
- Use BFS to compute connected component containing a node s.
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.
- Layer L_1 contains all neighbours of s.
- Given layers $L_0, L_1, \ldots, L_j$, layer L_{j+1} contains all nodes that
  1. do not belong to an earlier layer and
  2. are connected by an edge to a node in layer L_j.

Properties of BFS



- For each $j \geq 1$, layer L_j consists of all nodes

Properties of BFS

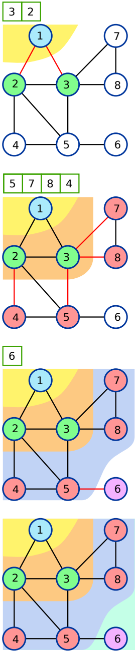


- For each $j \geq 1$, layer L_j consists of all nodes exactly at distance j from S.
- There is a path from s to t if and only if t is a member of some layer.

Implementing BFS

- Maintain an array Discovered and set Discovered$[v]$ $=$ *true* as soon as the algorithm sees v.

```
BFS(s):
  Set Discovered[s] = true and Discovered[v] = false for all other v
  Initialize L[0] to consist of the single element s
  Set the layer counter i = 0
  Set the current BFS tree T = Ø
  While L[i] is not empty
    Initialize an empty list L[i+1]
    For each node u ∈ L[i]
      Consider each edge (u, v) incident to u
      If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Add v to the list L[i+1]
      Endif
    Endfor
    Increment the layer counter i by one
  Endwhile
```



Using a Queue in BFS

- Instead of storing each layer in a different list, maintain all the layers in a single queue L.
- We can guarantee that all nodes in layer i will be put in the queue after every node in layer $i-1$ and before every node in layer $i+1$.

```
BFS(s):
    Set Discovered[s] = true
    Set Discovered[v] = false, for all other nodes v
    Initialize L to consist of the single element s
    While L is not empty
        Pop the node u at the head of L
        Consider each edge (u, v) incident on u
        If Discovered[v] = false then
            Set Discovered[v] = true
            Add edge (u, v) to the tree T
            Push v to the back of L
        Endif
    Endwhile
```

Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- How many times is each node popped from L?

Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- How many times is each node popped from L? Exactly once.

Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- How many times is each node popped from L? Exactly once.
- Time used by `for` loop for a node u:

Analysis of BFS Implementation

```
BFS(s):
  Set Discovered[s] = true
  Set Discovered[v] = false, for all other nodes v
  Initialize L to consist of the single element s
  While L is not empty
     Pop the node u at the head of L
     Consider each edge (u, v) incident on u
     If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Push v to the back of L
     Endif
  Endwhile
```

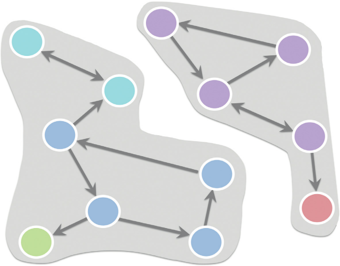
- How many times is each node popped from L? Exactly once.
- Time used by `for` loop for a node u: $O(d(u))$ time.

Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

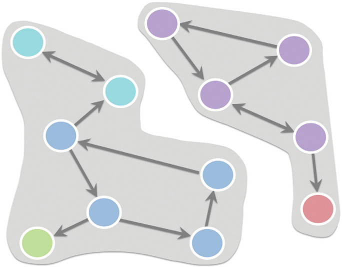
- How many times is each node popped from L? Exactly once.
- Time used by `for` loop for a node u: $O(d(u))$ time.
- Total time for all `for` loops: $\sum_{u \in G} O(d(u)) = O(m)$ time.
- Total time is $O(n + m)$.

Connected Components in Directed Graphs



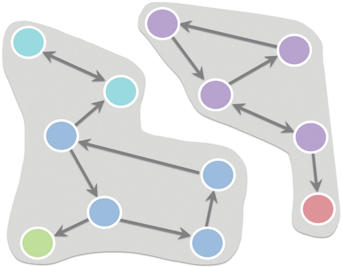
- In directed graphs, connectivity is not symmetric.

Connected Components in Directed Graphs



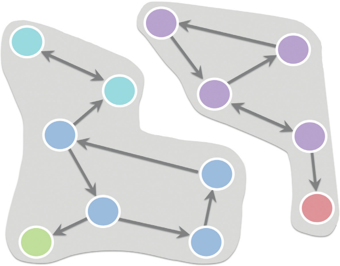
- In directed graphs, connectivity is not symmetric.
- A *weakly connected component* of a directed graph G is a connected component of the undirected graph G' obtained by replacing every edge in G by an undirected edge.

Connected Components in Directed Graphs



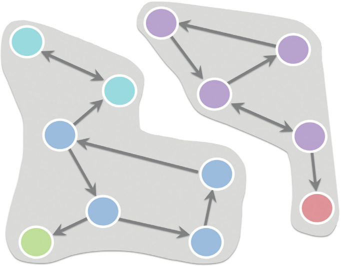
- In directed graphs, connectivity is not symmetric.
- A *weakly connected component* of a directed graph G is a connected component of the undirected graph G' obtained by replacing every edge in G by an undirected edge.
- We can compute all weakly connected components in linear time.

Connected Components in Directed Graphs



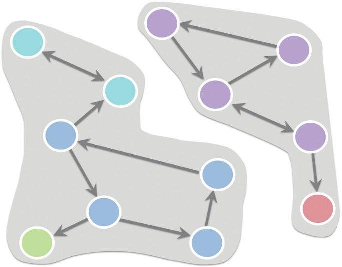
- In directed graphs, connectivity is not symmetric.
- A *strongly connected component* of a directed graph $G = (V, E)$ is a subgraph $H = (V', E')$ of G such

Connected Components in Directed Graphs



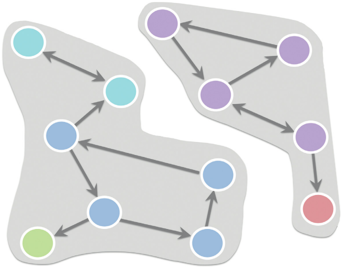
- In directed graphs, connectivity is not symmetric.
- A *strongly connected component* of a directed graph $G = (V, E)$ is a subgraph $H = (V', E')$ of G such
  - for every pair of nodes u, v in V' there is a u-to-v path and a v-to-u path in H, i.e., that use only the edges in E' and

Connected Components in Directed Graphs



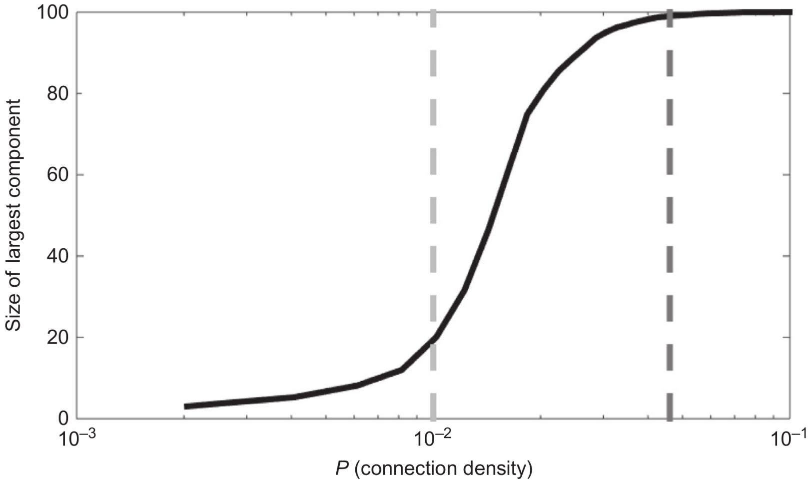
- In directed graphs, connectivity is not symmetric.
- A *strongly connected component* of a directed graph $G = (V, E)$ is a subgraph $H = (V', E')$ of G such
  - for every pair of nodes u, v in V' there is a u-to-v path and a v-to-u path in H, i.e., that use only the edges in E' and
  - H is *maximal*, i.e., for every node $x \in V - V'$, there is at least one node $y \in V'$ such that there is no path in G from x to y or from y to x.

Connected Components in Directed Graphs



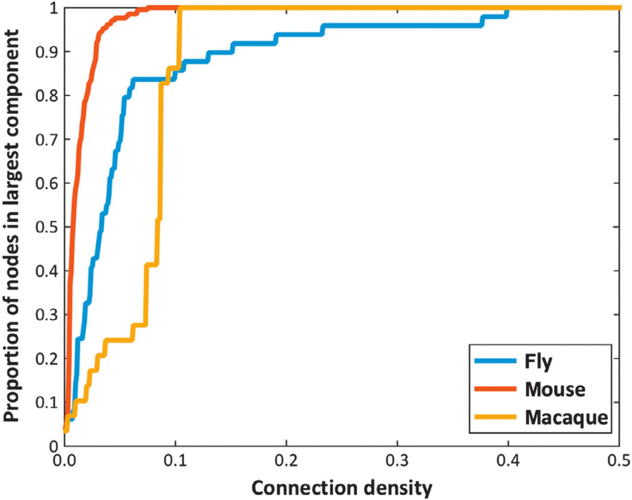
- In directed graphs, connectivity is not symmetric.
- A *strongly connected component* of a directed graph $G = (V, E)$ is a subgraph $H = (V', E')$ of G such
  - for every pair of nodes u, v in V' there is a u-to-v path and a v-to-u path in H, i.e., that use only the edges in E' and
  - H is *maximal*, i.e., for every node $x \in V - V'$, there is at least one node $y \in V'$ such that there is no path in G from x to y or from y to x.
- We can compute all strongly connected components in linear time using DFS with some tricks.

Largest Component in Brain Graphs



- Phase transition for appearance of large component in E-R graphs.

Largest Component in Brain Graphs



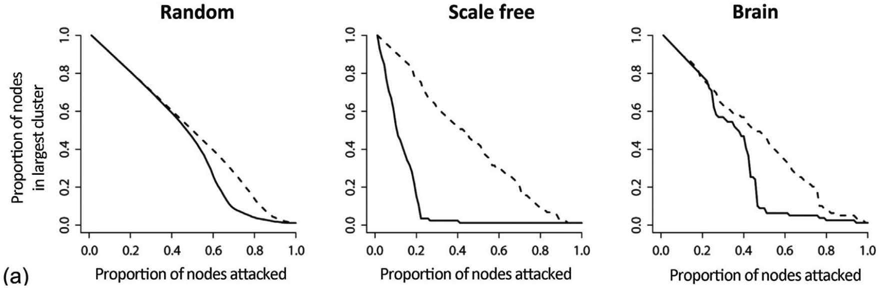
- Add edges in decreasing order of weight.
- Plot the size of the largest weakly connected component.

Random and Targeted Attack on Brain Networks

- Remove nodes randomly.
- Targeted attack: Remove nodes in decreasing order of degree.

Random and Targeted Attack on Brain Networks

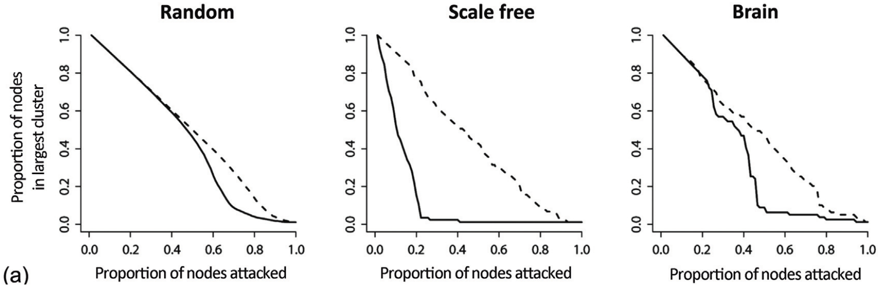
- Remove nodes randomly.
- Targeted attack: Remove nodes in decreasing order of degree.



$\Pr\{\text{degree } = k\} \sim k^{-\gamma}$ $\sim k^{-\gamma} e^{-k/k_c}$

Random and Targeted Attack on Brain Networks

- Remove nodes randomly.
- Targeted attack: Remove nodes in decreasing order of degree.



$\Pr\{\text{degree } = k\} \sim k^{-\gamma}$ $\sim k^{-\gamma} e^{-k/k_c}$

- Degree distribution of the brain is broad-scale: characterized by an exponentially-truncated power law.
- Concentration of links on hub nodes is weaker in a broad-scale network compared to a scale-free network.

Shortest Paths Problem

- $G(V, E)$ is a 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 Problem

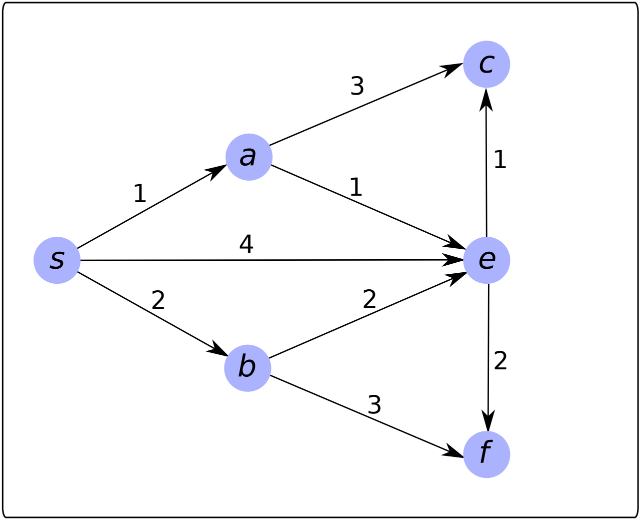
- $G(V, E)$ is a directed graph. Each edge e has a length $l(e) \geq 0$.
- V has n nodes and E has m edges.
- *Length of a path* P is the sum of the lengths of the edges in P.
- Goal is to determine the shortest path from a specified start node s to each node in V.
- Aside: If G is undirected, convert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

SHORTEST PATHS

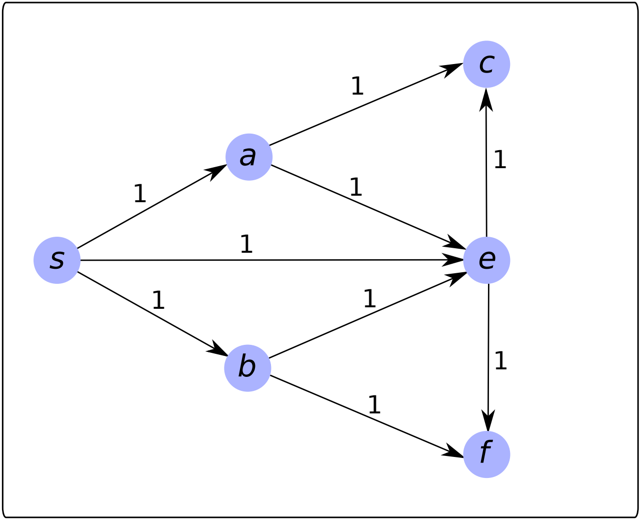
Given a directed graph $G(V, E)$, a function $l : E \rightarrow \mathbb{R}^+$, and a node $s \in V$,

compute a set $\{P(u), u \in V\}$, where $P(u)$ is the shortest path in G from s to u.

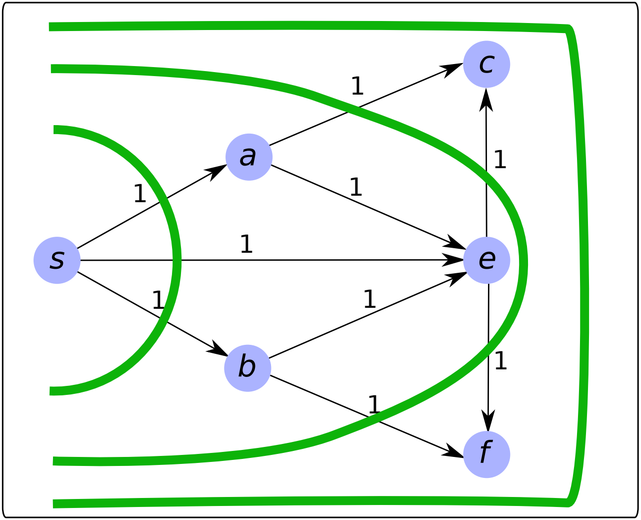
Shortest Paths Problem Instance



Generalizing BFS

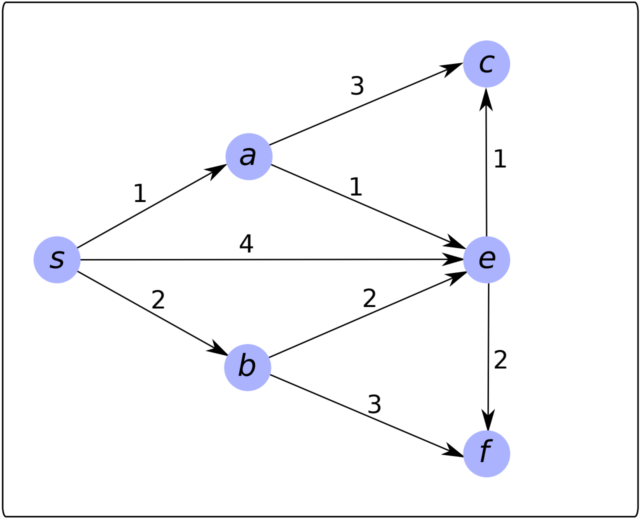


Generalizing BFS



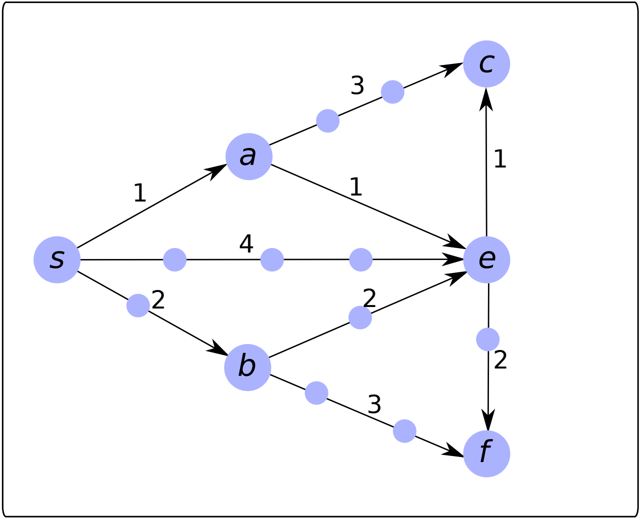
Unweighted graph: Use BFS. Process nodes in non-decreasing order of distance.

Generalizing BFS



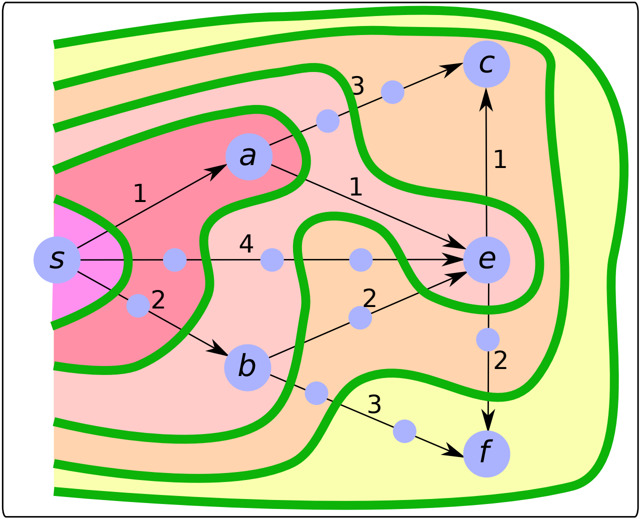
Weighted graph: Edge weights are integers. Can we make the graph unweighted?

Generalizing BFS



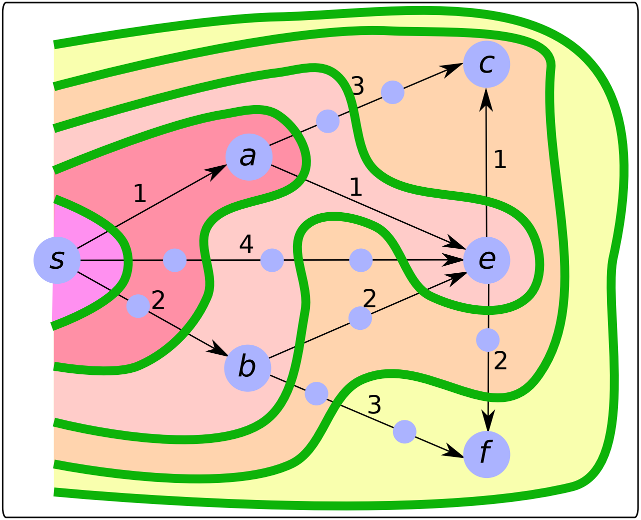
Add dummy nodes: Edge of weight w gets $w-1$ nodes.

Generalizing BFS



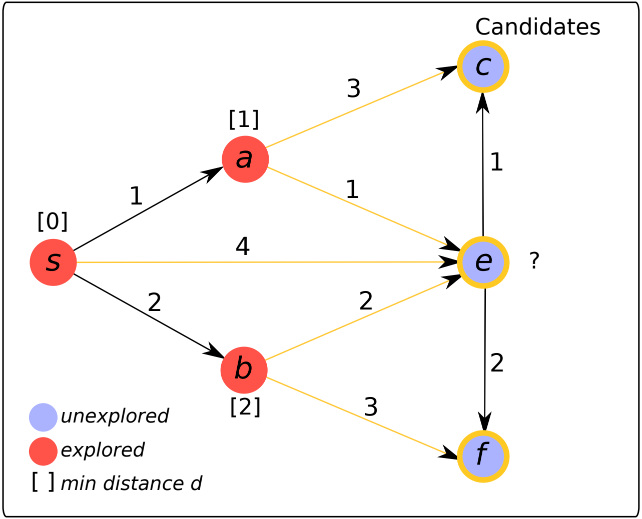
Dummy nodes: BFS computes shortest paths correctly. Running time is

Generalizing BFS



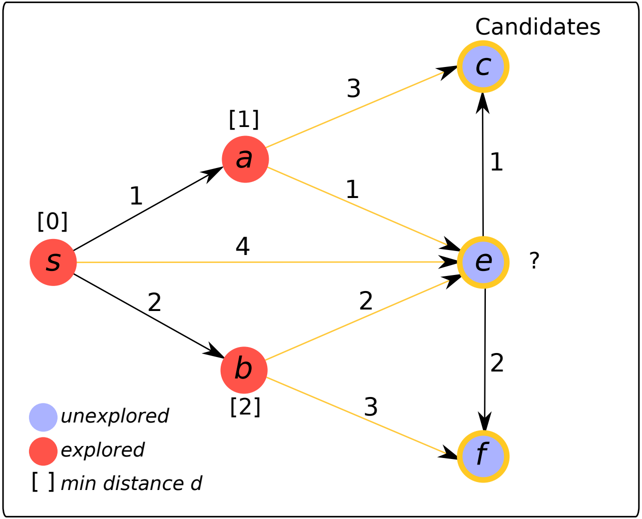
Dummy nodes: BFS computes shortest paths correctly. Running time is $O(m + n + \sum_{e \in E} l(e))$. *Pseudo-polynomial time*: depends on input *values*.

Generalizing BFS to Dijkstra's Algorithm



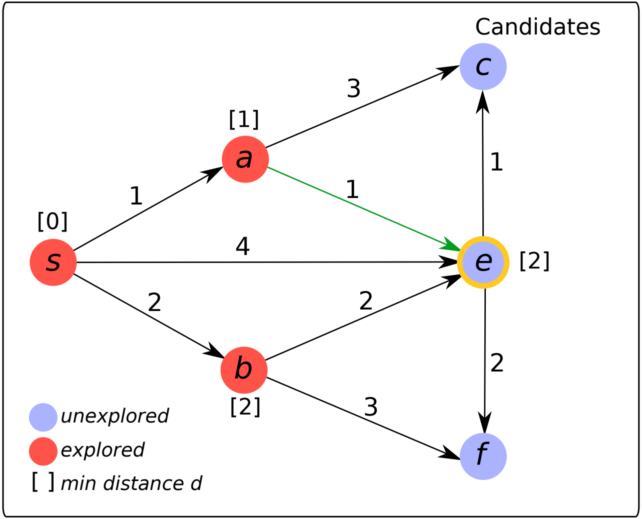
Like BFS: explore nodes in non-increasing order of distance from s. Once a node is explored, its distance is fixed.

Generalizing BFS to Dijkstra's Algorithm



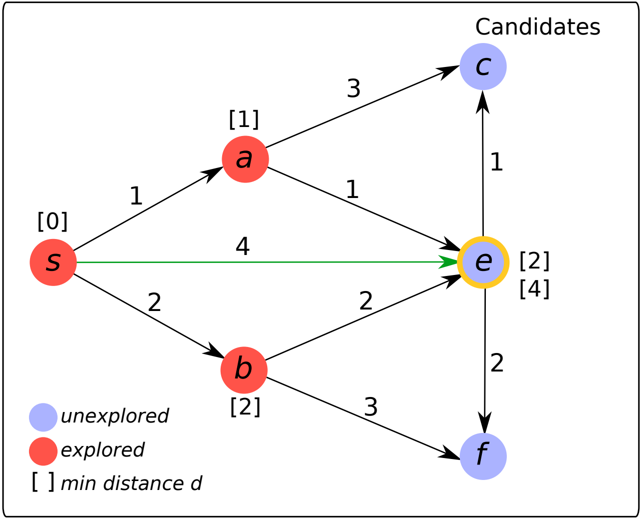
Unlike BFS: Layers are not uniform. Which node to process next?
Candidates are nodes with an edge from a explored node.

Generalizing BFS to Dijkstra's Algorithm



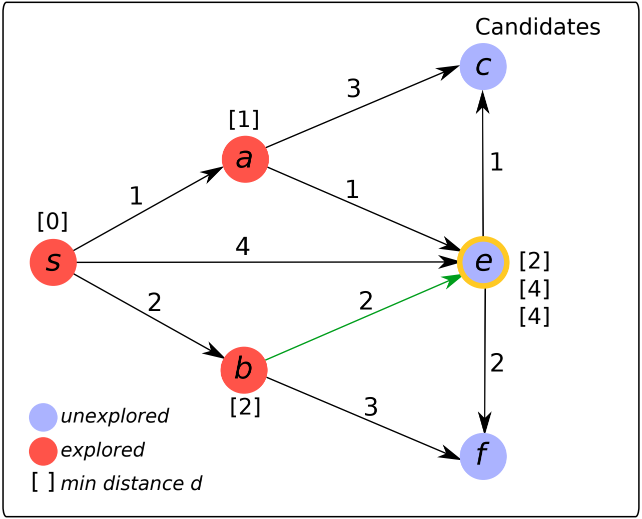
For each unexplored node, determine "best" preceding explored node.

Generalizing BFS to Dijkstra's Algorithm



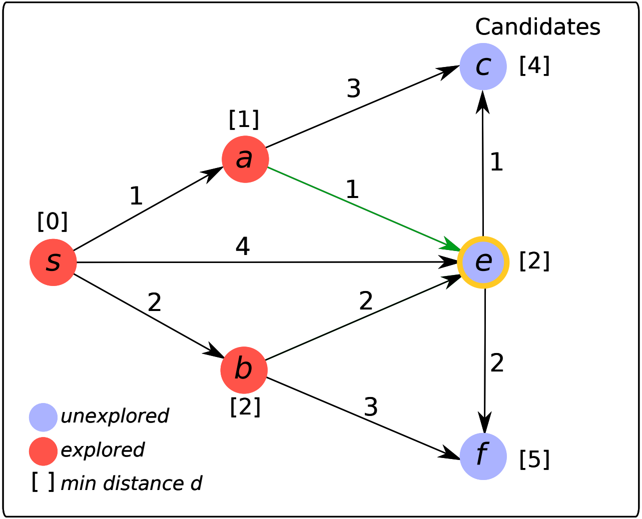
For each unexplored node, determine "best" preceding explored node.

Generalizing BFS to Dijkstra's Algorithm



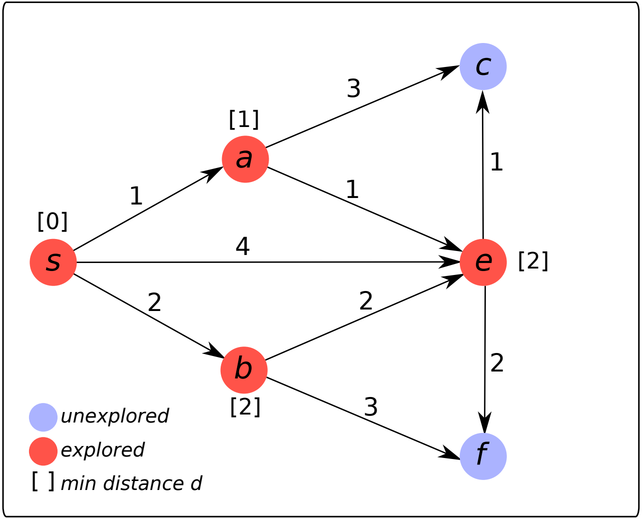
For each unexplored node, determine "best" preceding explored node.

Generalizing BFS to Dijkstra's Algorithm



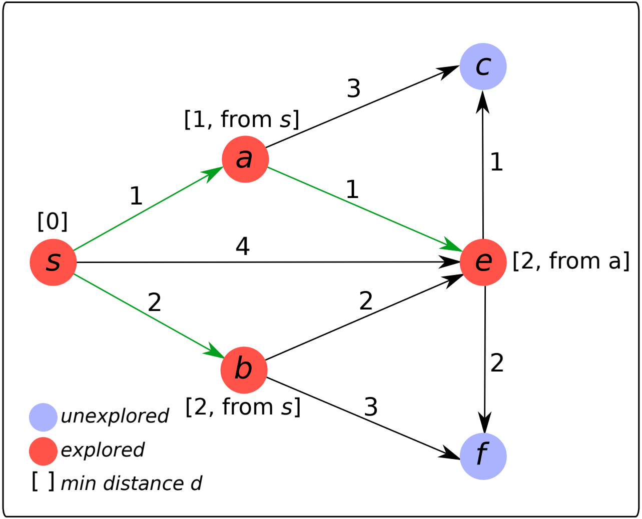
For each unexplored node, determine "best" preceding explored node.
Record shortest path length only through explored nodes.

Generalizing BFS to Dijkstra's Algorithm



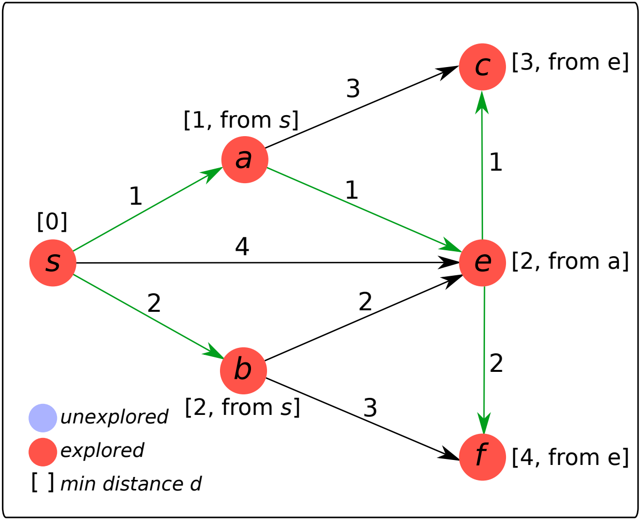
Explore node with smallest path length only through explored nodes.

Generalizing BFS to Dijkstra's Algorithm



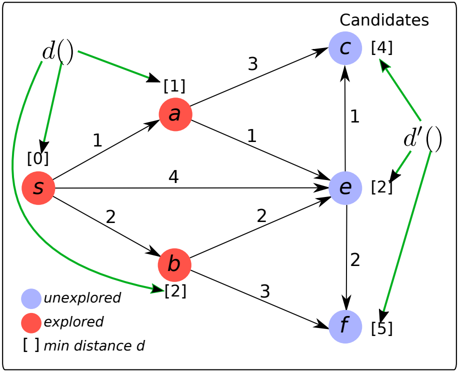
Like BFS: Record previous node in the computed path.

Generalizing BFS to Dijkstra's Algorithm



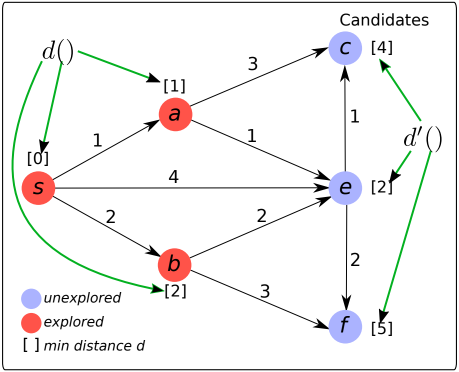
Follow previous nodes to compute shortest path. Like BFS: these edges form a tree.

Idea Underlying Dijkstra's Algorithm



- Maintain a set S of explored nodes.
  - For each node $u \in S$, compute a value $d(u)$, which (we will prove) is the length of the shortest path from s to u.
  - For each node $x \notin S$, maintain a value $d'(x)$, which is the length of the shortest path from s to x using only the nodes in S (and x, of course).

Idea Underlying Dijkstra's Algorithm

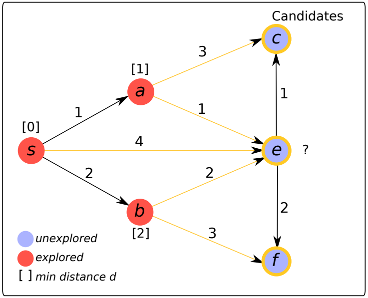


- Maintain a set S of explored nodes.
  - For each node $u \in S$, compute a value $d(u)$, which (we will prove) is the length of the shortest path from s to u.
  - For each node $x \notin S$, maintain a value $d'(x)$, which is the length of the shortest path from s to x using only the nodes in S (and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

Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

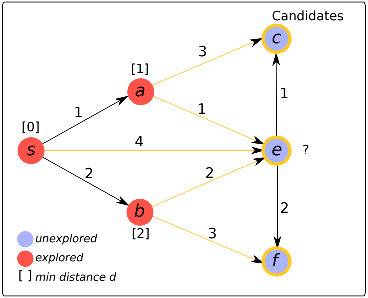
1: $S = \{s\}$ and $d(s) = 0$
2: **while** $S \neq V$ **do**
3: **for** every node $x \in V - S$ **do**
4: Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
5: Set $v = \arg\min_{x \in V-S} d'(x)$
6: Add v to S and set $d(v) = d'(v)$



Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

```
S = {s} and d(s) = 0
while S ≠ V do
    for every node x ∈ V − S do
        Set d'(x) = min_{(u,x): u∈S} (d(u) + l(u, x))
    Set v = arg min_{x∈V−S} d'(x)
    Add v to S and set d(v) = d'(v)
```

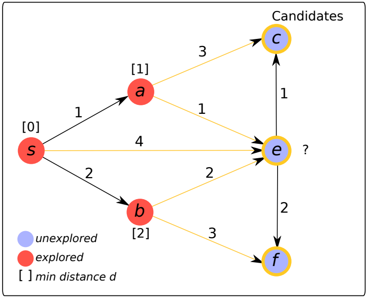


- How do we parse $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$?

Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

1: $S = \{s\}$ and $d(s) = 0$
2: **while** $S \neq V$ **do**
3: **for** every node $x \in V - S$ **do**
4: Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
5: Set $v = \arg\min_{x \in V-S} d'(x)$
6: Add v to S and set $d(v) = d'(v)$

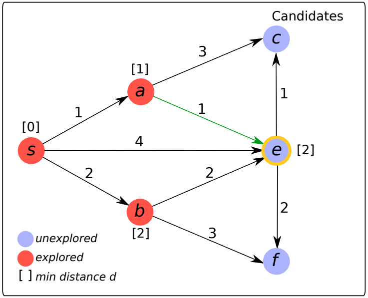


- How do we parse $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$?
  - The algorithm is examining a particular (unexplored) node x in $V - S$.

Dijkstra's Algorithm

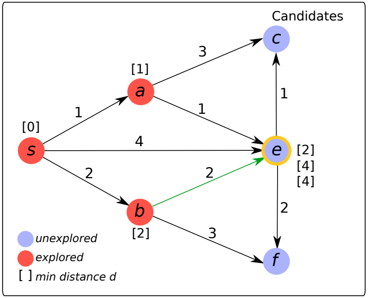
DIJKSTRA'S ALGORITHM(G, l, s)

1: $S = \{s\}$ and $d(s) = 0$
2: **while** $S \neq V$ **do**
3: **for** every node $x \in V - S$ **do**
4: Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
5: Set $v = \arg\min_{x \in V-S} d'(x)$
6: Add v to S and set $d(v) = d'(v)$



- How do we parse $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$?
  - The algorithm is examining a particular (unexplored) node x in $V - S$.
  - Argument of min runs over all edges of the type (u, x), where u is in S (i.e., u is explored).

Dijkstra's Algorithm



DIJKSTRA'S ALGORITHM(G, l, s)

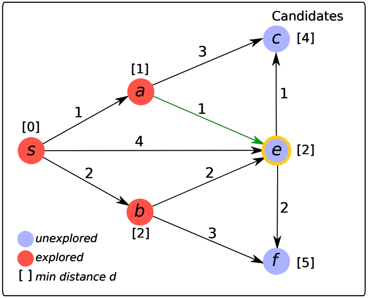
1: $S = \{s\}$ and $d(s) = 0$
2: **while** $S \neq V$ **do**
3: **for** every node $x \in V - S$ **do**
4: Set $d'(x) = \min_{(u,x):u\in S}(d(u) + l(u,x))$
5: Set $v = \arg\min_{x \in V-S} d'(x)$
6: Add v to S and set $d(v) = d'(v)$

- How do we parse $d'(x) = \min_{(u,x):u\in S}(d(u) + l(u,x))$?
  - The algorithm is examining a particular (unexplored) node x in $V - S$.
  - Argument of min runs over all edges of the type (u, x), where u is in S (i.e., u is explored).
  - For each such edge, we compute the length of the shortest path from s to x via u, which is $d(u) + l(u, x)$.

Dijkstra's Algorithm

Dijkstra's Algorithm(G, l, s)

$S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  for every node $x \in V - S$ **do**
    Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
  Set $v = \arg\min_{x \in V - S} d'(x)$
  Add v to S and set $d(v) = d'(v)$

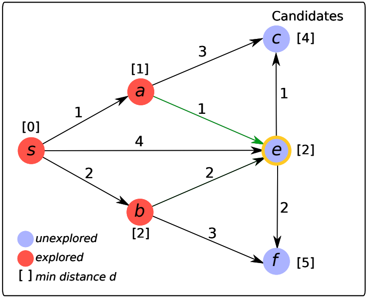


- How do we parse $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$?
  - The algorithm is examining a particular (unexplored) node x in $V - S$.
  - Argument of min runs over all edges of the type (u, x), where u is in S (i.e., u is explored).
  - For each such edge, we compute the length of the shortest path from s to x via u, which is $d(u) + l(u, x)$.
  - We store the smallest of these values in $d'(x)$.

Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

1: $S = \{s\}$ and $d(s) = 0$
2: **while** $S \neq V$ **do**
3: **for** every node $x \in V - S$ **do**
4: Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
5: Set $v = \arg\min_{x \in V-S} d'(x)$
6: Add v to S and set $d(v) = d'(v)$

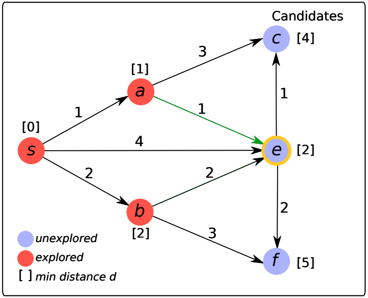


- How do we parse $v = \arg\min_{x \in V-S} d'(x)$?

Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

1: $S = \{s\}$ and $d(s) = 0$
2: **while** $S \neq V$ **do**
3: **for** every node $x \in V - S$ **do**
4: Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
5: Set $v = \arg\min_{x \in V-S} d'(x)$
6: Add v to S and set $d(v) = d'(v)$

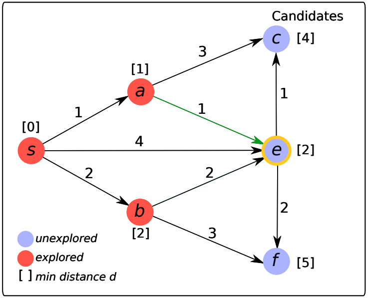


- How do we parse $v = \arg\min_{x \in V-S} d'(x)$?
  - Run over all (unexplored) nodes x in $V - S$.

Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

1: $S = \{s\}$ and $d(s) = 0$
2: **while** $S \neq V$ **do**
3: **for** every node $x \in V - S$ **do**
4: Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
5: Set $v = \arg\min_{x \in V-S} d'(x)$
6: Add v to S and set $d(v) = d'(v)$

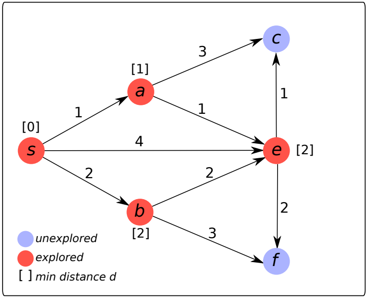


- How do we parse $v = \arg\min_{x \in V-S} d'(x)$?
  - Run over all (unexplored) nodes x in $V - S$.
  - Examine the d' values for these nodes.

Dijkstra's Algorithm

Dijkstra's Algorithm(G, l, s)

$S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  for every node $x \in V - S$ **do**
    Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
  Set $v = \arg\min_{x \in V-S} d'(x)$
  Add v to S and set $d(v) = d'(v)$

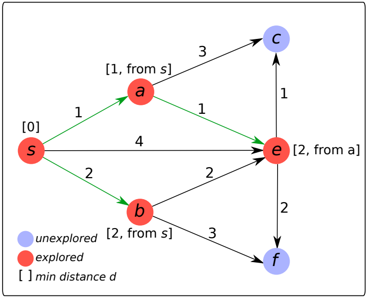


- How do we parse $v = \arg\min_{x \in V-S} d'(x)$?
  - Run over all (unexplored) nodes x in $V - S$.
  - Examine the d' values for these nodes.
  - Return the *argument* (i.e., the node) that has the smallest value of $d'(x)$.

Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

$S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  for every node $x \in V - S$ **do**
    Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
  Set $v = \arg\min_{x \in V-S} d'(x)$
  Add v to S and set $d(v) = d'(v)$



- How do we parse $v = \arg\min_{x \in V-S} d'(x)$?
  - Run over all (unexplored) nodes x in $V - S$.
  - Examine the d' values for these nodes.
  - Return the *argument* (i.e., the node) that has the smallest value of $d'(x)$.
- To compute the shortest paths: when adding a node v to S, store the predecessor u that minimises $d'(v)$.

Proof of Correctness

- Let $P(u)$ be the path computed by the algorithm for a node u.
- Claim: $P(u)$ is the shortest path from s to u.
- Prove by induction on the size of S, i.e., follow the algorithm.

Proof of Correctness

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

Proof of Correctness

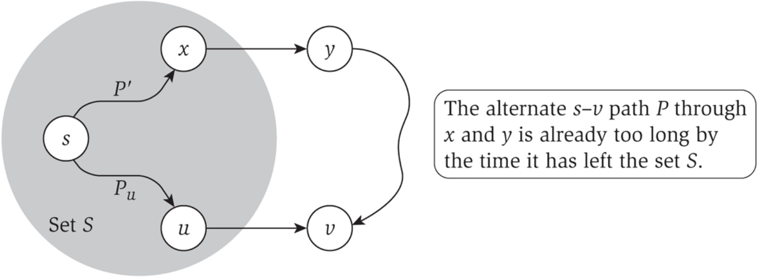
- Let $P(u)$ be the path computed by the algorithm for a node u.
- Claim: $P(u)$ is the shortest path from s to u.
- Prove by induction on the size of S, i.e., follow the algorithm.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive hypothesis: The algorithm has correctly computed $P(t)$ for all nodes $t \in S$.

Proof of Correctness

- Let $P(u)$ be the path computed by the algorithm for a node u.
- Claim: $P(u)$ is the shortest path from s to u.
- Prove by induction on the size of S, i.e., follow the algorithm.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive hypothesis: The algorithm has correctly computed $P(t)$ for all nodes $t \in S$.
  - 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 R from s to v? We must prove this cannot be the case.

Proof of Correctness

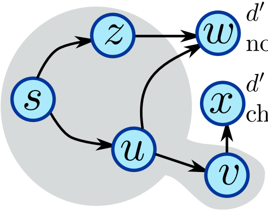
- Let $P(u)$ be the path computed by the algorithm for a node u.
- Claim: $P(u)$ is the shortest path from s to u.
- Prove by induction on the size of S, i.e., follow the algorithm.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive hypothesis: The algorithm has correctly computed $P(t)$ for all nodes $t \in S$.
  - 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 R from s to v? We must prove this cannot be the case.



A Faster implementation of Dijkstra's Algorithm

Dijkstra's Algorithm(G, l, s)

1: $S = \{s\}$ and $d(s) = 0$
2: **while** $S \neq V$ **do**
3: **for** every node $x \in V - S$ **do**
4: Set $d'(x) = \min_{(u,x): u \in S}(d(u) + l(u,x))$
5: Set $v = \arg\min_{x \in V - S} d'(x)$
6: Add v to S and set $d(v) = d'(v)$



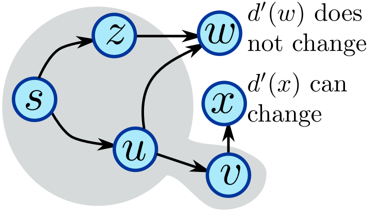
$d'(w)$ does not change

$d'(x)$ can change

A Faster implementation of Dijkstra's Algorithm

Dijkstra's Algorithm(G, l, s)

```
S = {s} and d(s) = 0
while S ≠ V do
    for every node x ∈ V − S do
        Set d'(x) = min_{(u,x):u∈S} (d(u) + l(u, x))
    Set v = arg min_{x∈V−S} d'(x)
    Add v to S and set d(v) = d'(v)
```

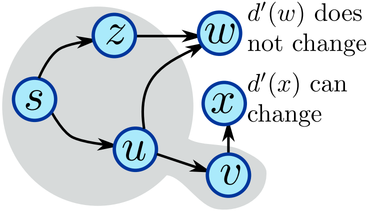


- Observation: If we add v to S, $d'(x)$ changes only if (v, x) is an edge in G.

A Faster implementation of Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

1: $S = \{s\}$ and $d(s) = 0$
2: **while** $S \neq V$ **do**
3: **for** every node $x \in V - S$ **do**
4: Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
5: Set $v = \arg\min_{x \in V - S} d'(x)$
6: Add v to S and set $d(v) = d'(v)$

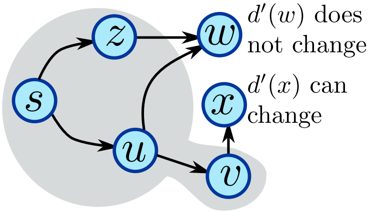


- Observation: If we add v to S, $d'(x)$ changes only if (v, x) is an edge in G.
- Idea: For each node $x \in V - S$, store the current value of $d'(x)$. Upon adding a node v to S, update $d'()$ only for neighbours of v.

A Faster implementation of Dijkstra's Algorithm

Dijkstra's Algorithm(G, l, s)

$S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  for every node $x \in V - S$ **do**
    Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
  Set $v = \arg\min_{x \in V - S} d'(x)$
  Add v to S and set $d(v) = d'(v)$

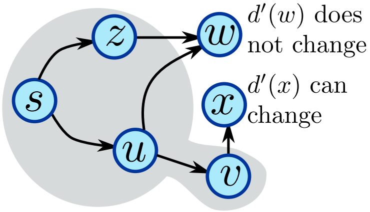


- Observation: If we add v to S, $d'(x)$ changes only if (v, x) is an edge in G.
- Idea: For each node $x \in V - S$, store the current value of $d'(x)$. Upon adding a node v to S, update $d'()$ only for neighbours of v.
- How do we efficiently compute $v = \arg\min_{x \in V - S} d'(x)$?

A Faster implementation of Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

$S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
    for every node $x \in V - S$ **do**
        Set $d'(x) = \min_{(u,x): u \in S}(d(u) + l(u,x))$
    Set $v = \arg\min_{x \in V - S} d'(x)$
    Add v to S and set $d(v) = d'(v)$



- Observation: If we add v to S, $d'(x)$ changes only if (v, x) is an edge in G.
- Idea: For each node $x \in V - S$, store the current value of $d'(x)$. Upon adding a node v to S, update $d'()$ only for neighbours of v.
- How do we efficiently compute $v = \arg\min_{x \in V - S} d'(x)$?
- Use a priority queue!

Faster Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

```
1: INSERT(Q, s, 0).
2: while S ≠ V do
3:    (v, d'(v)) = EXTRACTMIN(Q)
4:    Add v to S and set d(v) = d'(v)
5:    for every node x ∈ V − S such that (v, x) is an edge in G do
6:       if d(v) + l(v, x) < d'(x) then
7:          d'(x) = d(v) + l(v, x)
8:          CHANGEKEY(Q, x, d'(x))
```

- For each node $x \in V - S$, store the pair $(x, d'(x))$ in a priority queue Q with $d'(x)$ as the key.
- Determine the next node v to add to S using EXTRACTMIN (line 3).
- After adding v to S, for each node $x \in V - S$ such that there is an edge from v to x, check if $d'(x)$ should be updated, i.e., if there is a shortest path from s to x via v (lines 5–8).
- In line 8, if x is not in Q, simply insert it.

Running Time of Faster Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

INSERT$(Q, s, 0)$.
while $S \neq V$ **do**
  $(v, d'(v)) =$ EXTRACTMIN(Q)
  Add v to S and set $d(v) = d'(v)$
  for every node $x \in V - S$ such that (v, x) is an edge in G **do**
    if $d(v) + l_{(v,x)} < d'(x)$ **then**
      $d'(x) = d(v) + l_{(v,x)}$
      CHANGEKEY$(Q, x, d'(x))$

- How many invocations of EXTRACTMIN?

Running Time of Faster Dijkstra's Algorithm

$\textsc{Dijkstra's Algorithm}(G, l, s)$

$\textsc{Insert}(Q, s, 0)$.
while $S \neq V$ **do**
    $(v, d'(v)) = \textsc{ExtractMin}(Q)$
    Add v to S and set $d(v) = d'(v)$
    for every node $x \in V - S$ such that (v, x) is an edge in G **do**
        if $d(v) + l_{(v,x)} < d'(x)$ **then**
            $d'(x) = d(v) + l_{(v,x)}$
            $\textsc{ChangeKey}(Q, x, d'(x))$

- How many invocations of $\textsc{ExtractMin}$? $n - 1$.

Running Time of Faster Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

1: INSERT$(Q, s, 0)$.
2: **while** $S \neq V$ **do**
3: $\quad (v, d'(v)) =$ EXTRACTMIN(Q)
4: $\quad$ Add v to S and set $d(v) = d'(v)$
5: $\quad$ **for** every node $x \in V - S$ such that (v, x) is an edge in G **do**
6: $\quad\quad$ **if** $d(v) + l_{(v,x)} < d'(x)$ **then**
7: $\quad\quad\quad d'(x) = d(v) + l_{(v,x)}$
8: $\quad\quad\quad$ CHANGEKEY$(Q, x, d'(x))$

- How many invocations of EXTRACTMIN? $n - 1$.
- For every node v, what is the running time of step 5?

Running Time of Faster Dijkstra's Algorithm

Dijkstra's Algorithm(G, l, s)

1: Insert$(Q, s, 0)$.
2: **while** $S \neq V$ **do**
3: $(v, d'(v)) =$ ExtractMin(Q)
4: Add v to S and set $d(v) = d'(v)$
5: **for** every node $x \in V - S$ such that (v, x) is an edge in G **do**
6: **if** $d(v) + l_{(v,x)} < d'(x)$ **then**
7: $d'(x) = d(v) + l_{(v,x)}$
8: ChangeKey$(Q, x, d'(x))$

- How many invocations of ExtractMin? $n - 1$.
- For every node v, what is the running time of step 5? $O(d_{\mathrm{out}}(v))$, the number of *outgoing* neighbours of v.

Running Time of Faster Dijkstra's Algorithm

$\textsc{Dijkstra's Algorithm}(G, l, s)$

1: $\textsc{Insert}(Q, s, 0)$.
2: **while** $S \neq V$ **do**
3: $\quad(v, d'(v)) = \textsc{ExtractMin}(Q)$
4: $\quad$Add v to S and set $d(v) = d'(v)$
5: $\quad$**for** every node $x \in V - S$ such that (v, x) is an edge in G **do**
6: $\quad\quad$**if** $d(v) + l_{(v,x)} < d'(x)$ **then**
7: $\quad\quad\quad d'(x) = d(v) + l_{(v,x)}$
8: $\quad\quad\quad \textsc{ChangeKey}(Q, x, d'(x))$

- How many invocations of $\textsc{ExtractMin}$? $n - 1$.
- For every node v, what is the running time of step 5? $O(d_{\text{out}}(v))$, the number of *outgoing* neighbours of v.
- What is the total running time of step 5?

Running Time of Faster Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

1: INSERT$(Q, s, 0)$.
2: **while** $S \neq V$ **do**
3: $(v, d'(v)) = \text{EXTRACTMIN}(Q)$
4: Add v to S and set $d(v) = d'(v)$
5: **for** every node $x \in V - S$ such that (v, x) is an edge in G **do**
6: **if** $d(v) + l_{(v,x)} < d'(x)$ **then**
7: $d'(x) = d(v) + l_{(v,x)}$
8: CHANGEKEY$(Q, x, d'(x))$

- How many invocations of EXTRACTMIN? $n - 1$.
- For every node v, what is the running time of step 5? $O(d_{\text{out}}(v))$, the number of *outgoing* neighbours of v.
- What is the total running time of step 5? $\sum_{v \in V} O(d_{\text{out}}(v)) = O(m)$.

Running Time of Faster Dijkstra's Algorithm

$\textsc{Dijkstra's Algorithm}(G, l, s)$

1: $\textsc{Insert}(Q, s, 0)$.
2: **while** $S \neq V$ **do**
3: $\quad (v, d'(v)) = \textsc{ExtractMin}(Q)$
4: $\quad$ Add v to S and set $d(v) = d'(v)$
5: $\quad$ **for** every node $x \in V - S$ such that (v, x) is an edge in G **do**
6: $\quad\quad$ **if** $d(v) + l_{(v,x)} < d'(x)$ **then**
7: $\quad\quad\quad d'(x) = d(v) + l_{(v,x)}$
8: $\quad\quad\quad \textsc{ChangeKey}(Q, x, d'(x))$

- How many invocations of $\textsc{ExtractMin}$? $n - 1$.
- For every node v, what is the running time of step 5? $O(d_{\text{out}}(v))$, the number of *outgoing* neighbours of v.
- What is the total running time of step 5? $\sum_{v \in V} O(d_{\text{out}}(v)) = O(m)$.
- How many times does the algorithm invoke $\textsc{ChangeKey}$?

Running Time of Faster Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

1: INSERT$(Q, s, 0)$.
2: **while** $S \neq V$ **do**
3: $\quad (v, d'(v)) =$ EXTRACTMIN(Q)
4: $\quad$ Add v to S and set $d(v) = d'(v)$
5: $\quad$ **for** every node $x \in V - S$ such that (v, x) is an edge in G **do**
6: $\quad\quad$ **if** $d(v) + l_{(v,x)} < d'(x)$ **then**
7: $\quad\quad\quad d'(x) = d(v) + l_{(v,x)}$
8: $\quad\quad\quad$ CHANGEKEY$(Q, x, d'(x))$

- How many invocations of EXTRACTMIN? $n - 1$.
- For every node v, what is the running time of step 5? $O(d_{\text{out}}(v))$, the number of *outgoing* neighbours of v.
- What is the total running time of step 5? $\sum_{v \in V} O(d_{\text{out}}(v)) = O(m)$.
- How many times does the algorithm invoke CHANGEKEY? $\leq m$.

Running Time of Faster Dijkstra's Algorithm

Dijkstra's Algorithm(G, l, s)

1: Insert$(Q, s, 0)$.
2: **while** $S \neq V$ **do**
3: $(v, d'(v)) =$ ExtractMin(Q)
4: Add v to S and set $d(v) = d'(v)$
5: **for** every node $x \in V - S$ such that (v, x) is an edge in G **do**
6: **if** $d(v) + l_{(v,x)} < d'(x)$ **then**
7: $d'(x) = d(v) + l_{(v,x)}$
8: ChangeKey$(Q, x, d'(x))$

- How many invocations of ExtractMin? $n - 1$.
- For every node v, what is the running time of step 5? $O(d_{\mathrm{out}}(v))$, the number of *outgoing* neighbours of v.
- What is the total running time of step 5? $\sum_{v \in V} O(d_{\mathrm{out}}(v)) = O(m)$.
- How many times does the algorithm invoke ChangeKey? $\leq m$.
- What is total running time of the algorithm?

Running Time of Faster Dijkstra's Algorithm

$\text{Dijkstra's Algorithm}(G, l, s)$

1: $\text{Insert}(Q, s, 0)$.
2: **while** $S \neq V$ **do**
3: $(v, d'(v)) = \text{ExtractMin}(Q)$
4: Add v to S and set $d(v) = d'(v)$
5: **for** every node $x \in V - S$ such that (v, x) is an edge in G **do**
6: **if** $d(v) + l_{(v,x)} < d'(x)$ **then**
7: $d'(x) = d(v) + l_{(v,x)}$
8: $\text{ChangeKey}(Q, x, d'(x))$

- How many invocations of ExtractMin? $n - 1$.
- For every node v, what is the running time of step 5? $O(d_{\text{out}}(v))$, the number of *outgoing* neighbours of v.
- What is the total running time of step 5? $\sum_{v \in V} O(d_{\text{out}}(v)) = O(m)$.
- How many times does the algorithm invoke ChangeKey? $\leq m$.
- What is total running time of the algorithm? $O(m \log n)$.

Graph Measures Based on Shortest Paths

- *Characteristic path length* $l(G)$ is the average shortest path length between all pairs of nodes in G. $\delta(u, v) =$ shortest path length from u to v.

$$l(G) = \frac{1}{n(n-1)} \sum_{u,v \in V, u \neq v} \delta(u, v)$$

Graph Measures Based on Shortest Paths

- *Characteristic path length* $l(G)$ is the average shortest path length between all pairs of nodes in G. $\delta(u, v)$ = shortest path length from u to v.

$$l(G) = \frac{1}{n(n-1)} \sum_{u,v \in V, u \neq v} \delta(u, v)$$

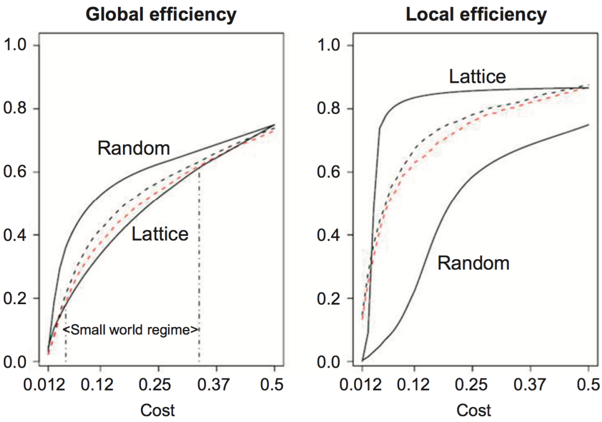
- *Global efficiency* $e_{\mathrm{glob}}(G)$ is the average of the reciprocal of the shortest path length between all pairs of nodes in G.

$$e_{\mathrm{glob}}(G) = \frac{1}{n(n-1)} \sum_{u,v \in V, u \neq v} \frac{1}{\delta(u, v)}$$

- *Local efficiency* $e_{\mathrm{loc}}(v)$ of a node v is the average of the reciprocal of the shortest path length between all pairs of neighbours of v in G.

$$e_{\mathrm{loc}}(v) = \frac{1}{d(v)(d(v)-1)} \sum_{\substack{u,v \in N(v) \\ u \neq v}} \frac{1}{\delta(u, v)}$$

Efficiency in Brain Networks



- Functional connectivity networks from fMRI data in young (black) and old (orange) human volunteers.
- x-axis is fraction of possible edges as threshold on edge weight varies.
- y-axis is global (left) and local (right) efficiency.
- Small world networks are both locally and globally efficient.