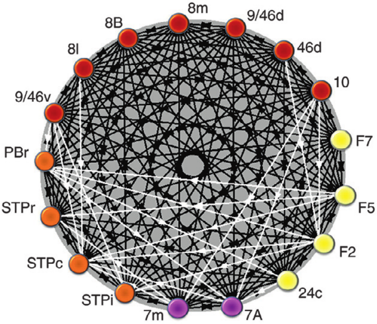
CS 6824: Components, Cliques, and Cores

T. M. Murali

February 15 and 20, 2018



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)
- Shortest paths (Chapter 7.1 and 7.2)
- Clustering coefficient and small world networks (Chapter 8.1 and 8.2)

Plan till Spring Break

- Clustering coefficient is a local measure of graph density.
- Small world property captures global features of graph density.

Plan till Spring Break

- Clustering coefficient is a local measure of graph density.
- Small world property captures global features of graph density.

Are there intermediate notions of graph density?

- Subgraphs that represent backbones of network topology (components, shortest paths, spanning trees, cores, Chapter 6.1, 6.2, 7.1, February 15 and 20)
- Modularity (Chapter 9, February 22, 27, March 1)

Student Presentations

- I have provided a list of topics (roughly corresponding to textbook sections) for student presentations on the course website.
- Each group should give me its top three choices by 5pm on Tuesday, February 20.
- I will assign one topic to each group by February 22.
- I will also add the topics to the course schedule.

Student Presentations

- I have provided a list of topics (roughly corresponding to textbook sections) for student presentations on the course website.
- Each group should give me its top three choices by 5pm on Tuesday, February 20.
- I will assign one topic to each group by February 22.
- I will also add the topics to the course schedule.
- Each group meets me for 60–90 minutes about two weeks before practice presentation.
  - I will announce office hours and a schedule for these meetings.
  - Goal is to discuss details of presentation.
  - Come prepared: read your section, find relevant papers, have a talk outline, ask me quesitons.
- Each group meets me for 60–90 minutes about two weeks before actual presentation.

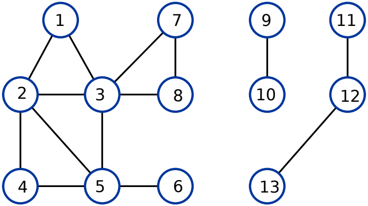
Student Presentations

- I have provided a list of topics (roughly corresponding to textbook sections) for student presentations on the course website.
- Each group should give me its top three choices by 5pm on Tuesday, February 20.
- I will assign one topic to each group by February 22.
- I will also add the topics to the course schedule.
- Each group meets me for 60–90 minutes about two weeks before practice presentation.
  - I will announce office hours and a schedule for these meetings.
  - Goal is to discuss details of presentation.
  - Come prepared: read your section, find relevant papers, have a talk outline, ask me quesitons.
- Each group meets me for 60–90 minutes about two weeks before actual presentation.
- Projects to be announced before spring break.

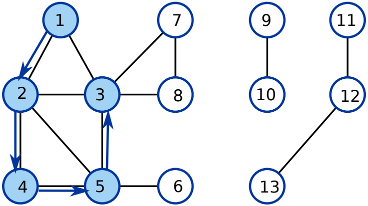
Plan after Spring Break

- Two invited presentations by Heidi Theussen from Smith Career Center (March 15 and 17)
- Practice presentations (March 20 to April 5, with one practice presentation held outside class hours)
- Presentations (April 10 to May 1)

Paths and Connectivity

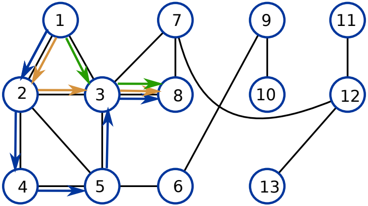


Paths and Connectivity



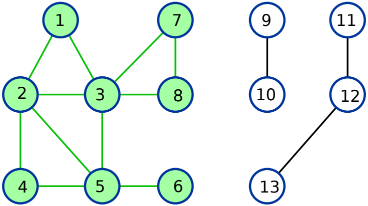
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.

Paths and Connectivity



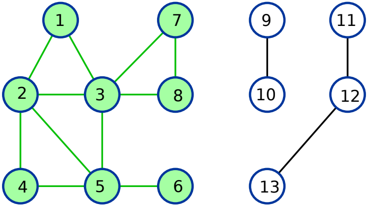
- A v_1-v_k *path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- *Distance* $d(u, v)$ between two nodes u and v is the minimum number of edges in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

Paths and Connectivity



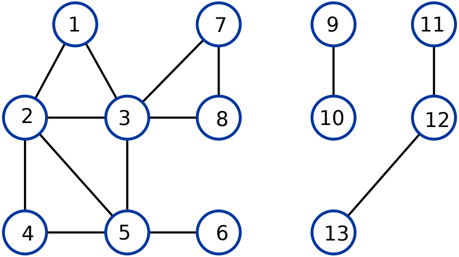
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- *Distance $d(u, v)$* between two nodes u and v is the minimum number of edges in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



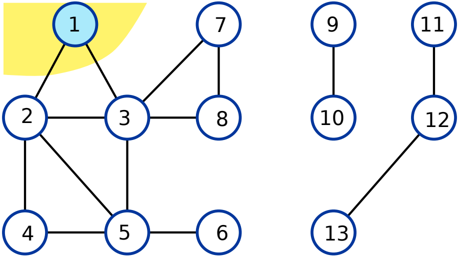
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- *Distance $d(u, v)$* between two nodes u and v is the minimum number of edges in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



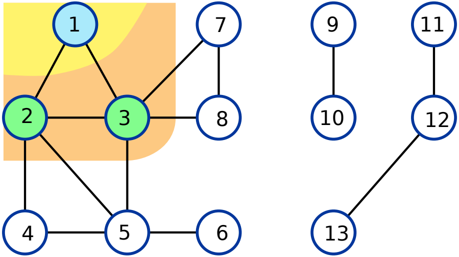
- Use BFS to compute connected component containing a node s.
- Idea: explore G starting at s and going “outward” in all directions, adding nodes one layer at a time.

Breadth-First Search (BFS)



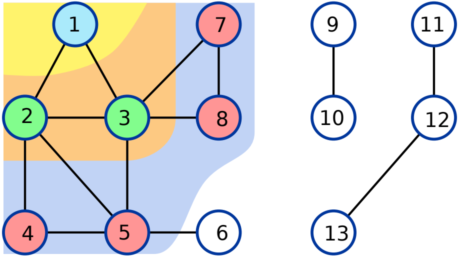
- Use BFS to compute connected component containing a node s.
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.

Breadth-First Search (BFS)



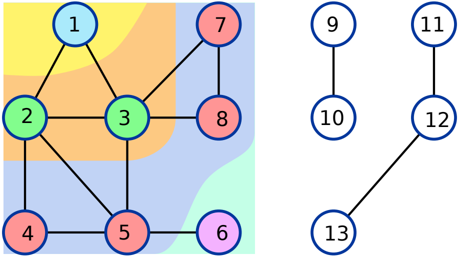
- Use BFS to compute connected component containing a node s.
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.
- Layer L_1 contains all neighbours of s.

Breadth-First Search (BFS)



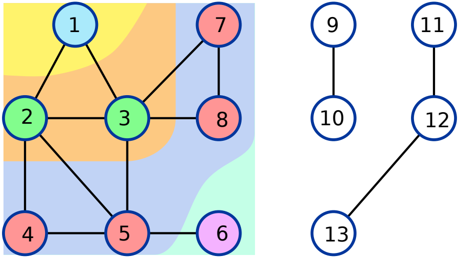
- Use BFS to compute connected component containing a node s.
- Idea: explore G starting at s and going “outward” in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.
- Layer L_1 contains all neighbours of s.
- Given layers $L_0, L_1, \ldots, L_j$, layer L_{j+1} contains 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)



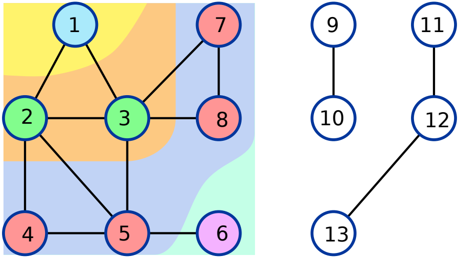
- Use BFS to compute connected component containing a node s.
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.
- Layer L_1 contains all neighbours of s.
- Given layers $L_0, L_1, \ldots, L_j$, layer L_{j+1} contains 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

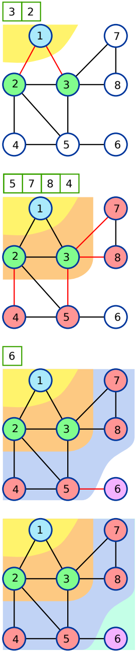


- For each $j \geq 1$, layer L_j consists of all nodes exactly at distance j from S.
- There is a path from s to t if and only if t is a member of some layer.

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
```

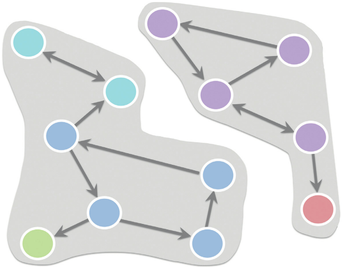
- How many times is each node popped from L? Exactly once.
- Time used by `for` loop for a node u: $O(d(u))$ time.

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
```

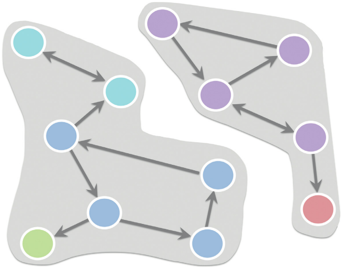
- How many times is each node popped from L? Exactly once.
- Time used by `for` loop for a node u: $O(d(u))$ time.
- Total time for all `for` loops: $\sum_{u \in G} O(d(u)) = O(m)$ time.
- Total time is $O(n + m)$.

Connected Components in Directed Graphs



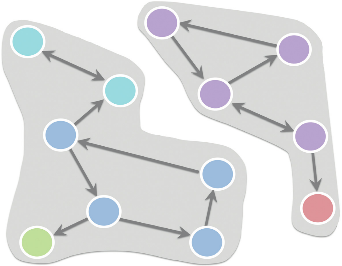
- In directed graphs, connectivity is not symmetric.

Connected Components in Directed Graphs



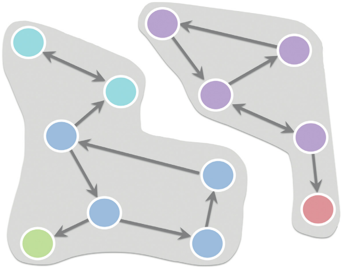
- In directed graphs, connectivity is not symmetric.
- A *weakly connected component* of a directed graph G is a connected component of the undirected graph G' obtained by replacing every edge in G by an undirected edge.

Connected Components in Directed Graphs



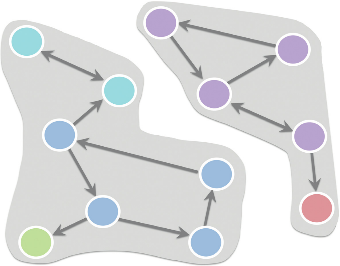
- In directed graphs, connectivity is not symmetric.
- A *weakly connected component* of a directed graph G is a connected component of the undirected graph G' obtained by replacing every edge in G by an undirected edge.
- We can compute all weakly connected components in linear time.

Connected Components in Directed Graphs



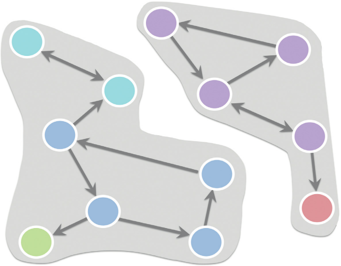
- In directed graphs, connectivity is not symmetric.
- A *strongly connected component* of a directed graph $G = (V, E)$ is a subgraph $H = (V', E')$ of G such

Connected Components in Directed Graphs



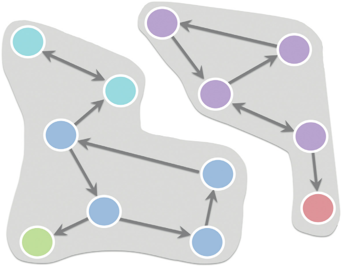
- In directed graphs, connectivity is not symmetric.
- A *strongly connected component* of a directed graph $G = (V, E)$ is a subgraph $H = (V', E')$ of G such
  - for every pair of nodes u, v in V' there is a u-to-v path and a v-to-u path in H, i.e., that use only the edges in E' and

Connected Components in Directed Graphs



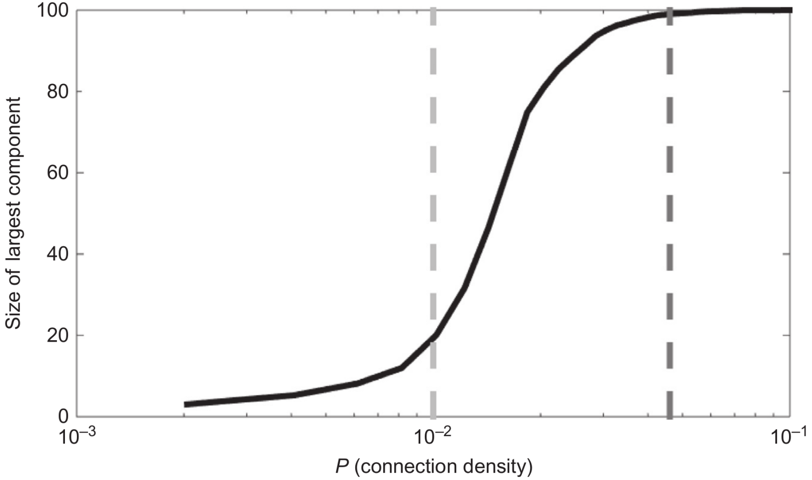
- In directed graphs, connectivity is not symmetric.
- A *strongly connected component* of a directed graph $G = (V, E)$ is a subgraph $H = (V', E')$ of G such
  - for every pair of nodes u, v in V' there is a u-to-v path and a v-to-u path in H, i.e., that use only the edges in E' and
  - H is *maximal*,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



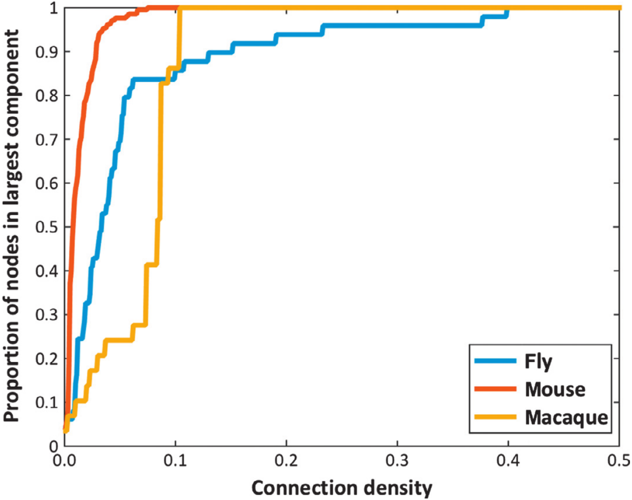
- In directed graphs, connectivity is not symmetric.
- A *strongly connected component* of a directed graph $G = (V, E)$ is a subgraph $H = (V', E')$ of G such
  - for every pair of nodes u, v in V' there is a u-to-v path and a v-to-u path in H, i.e., that use only the edges in E' and
  - H is *maximal*,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

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

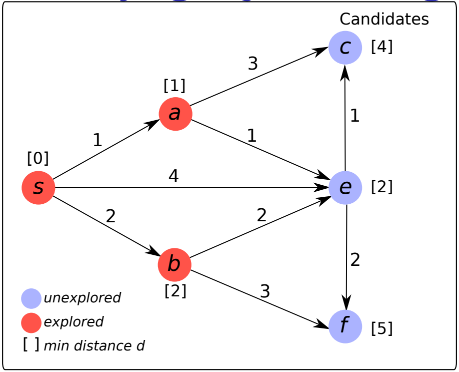
- $G(V, E)$ is a directed graph. Each edge e has a length $l(e) \geq 0$.
- V has n nodes and E has m edges.
- *Length of a path* P is the sum of the lengths of the edges in P.
- Goal is to determine the shortest path from a specified start node s to each node in V.
- Aside: If G is undirected, convert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

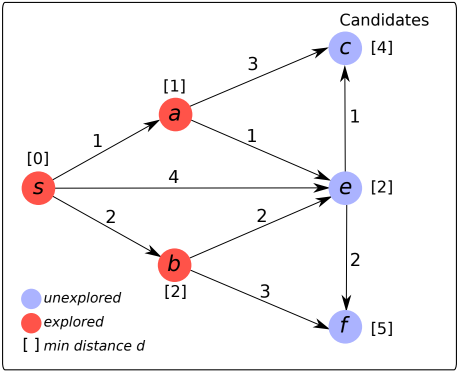
compute a set $\{P(u), u \in V\}$, where $P(u)$ is the shortest path in G from s to u.

Idea Underlying Dijkstra's Algorithm



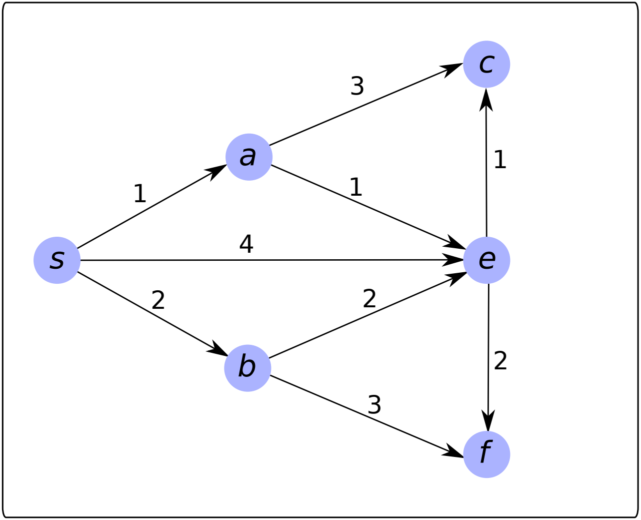
- Maintain a set S of explored nodes.
  - For each node $u \in S$, compute a value $d(u)$, which (we will prove) is the length of the shortest path from s to u.
  - For each node $x \notin S$, maintain a value $d'(x)$, which is the length of the shortest path from s to x using only the nodes in S (and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

Idea Underlying Dijkstra's Algorithm

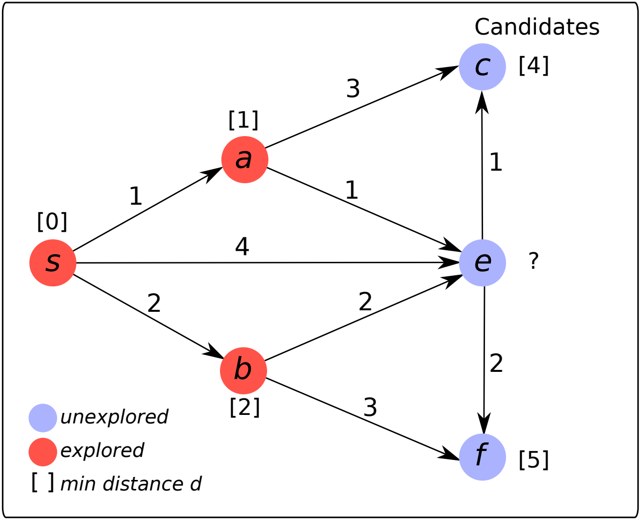


- Maintain a set S of explored nodes.
- "Greedily" add a node v to S that has the smallest value of $d'(v)$ (is closest to s using only nodes in S).
- Prove that at the moment we add v to S, $d(v) = d'(v)$.

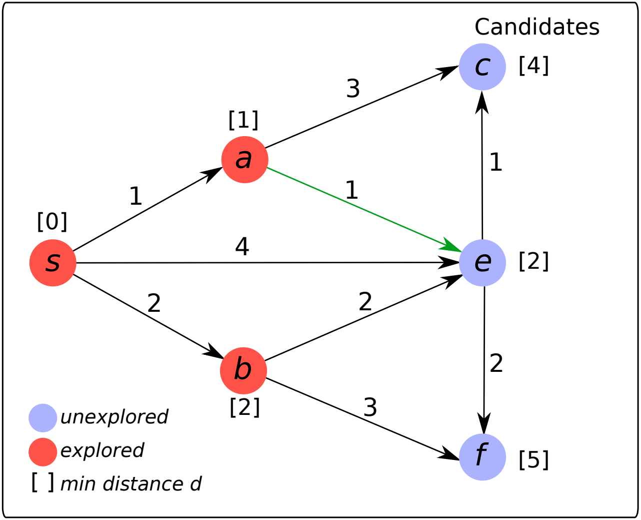
Example of Dijkstra's Algorithm



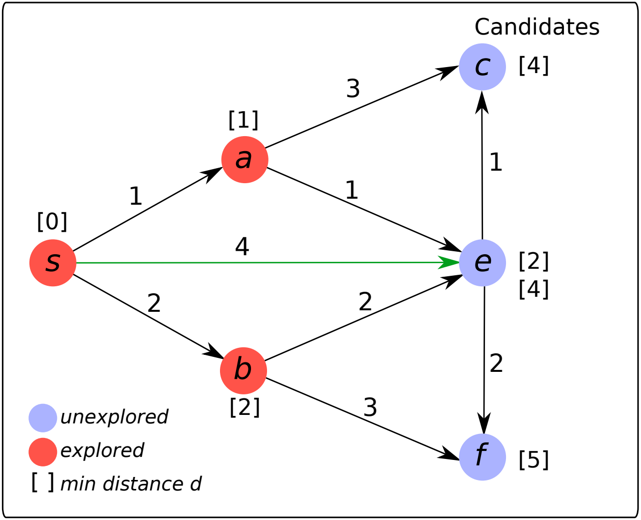
Example of Dijkstra's Algorithm



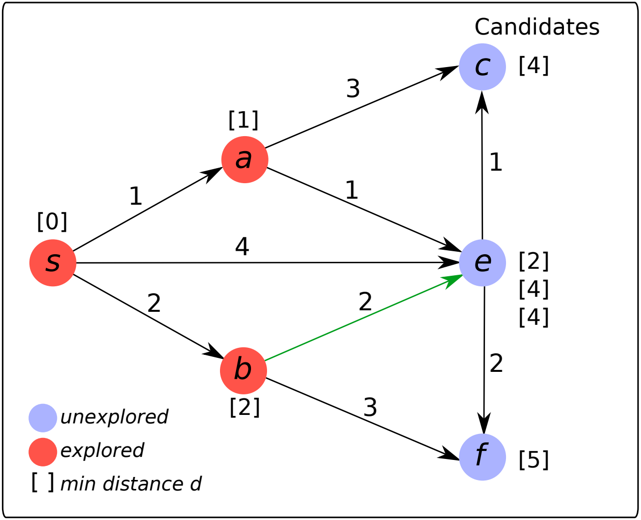
Example of Dijkstra's Algorithm



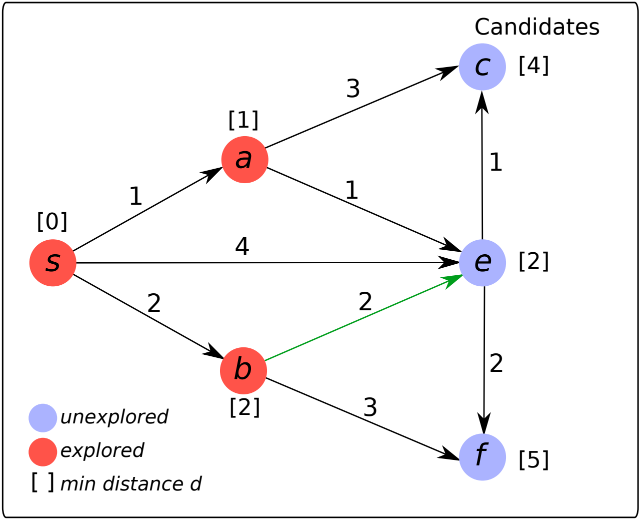
Example of Dijkstra's Algorithm



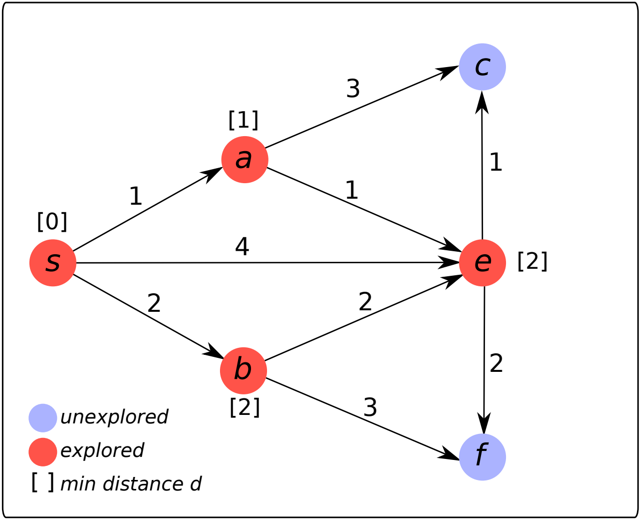
Example of Dijkstra's Algorithm



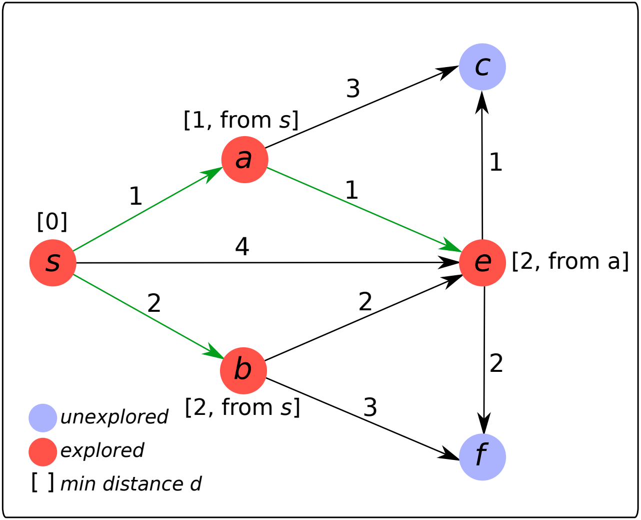
Example of Dijkstra's Algorithm



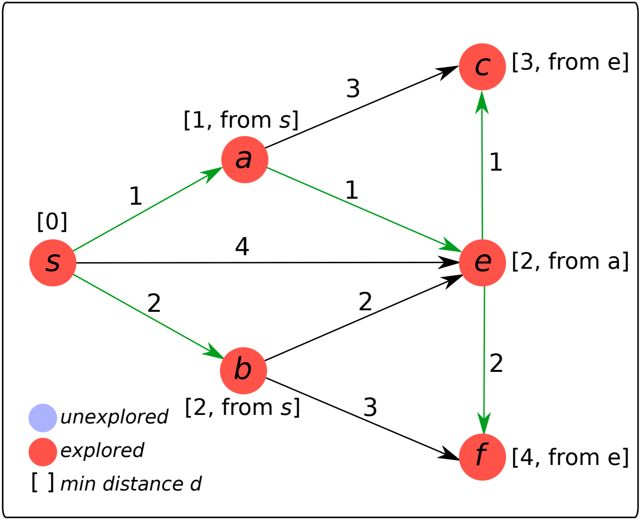
Example of Dijkstra's Algorithm



Example of Dijkstra's Algorithm



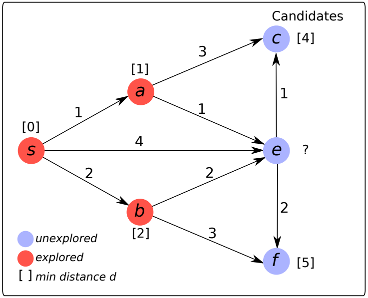
Example of Dijkstra's Algorithm



Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

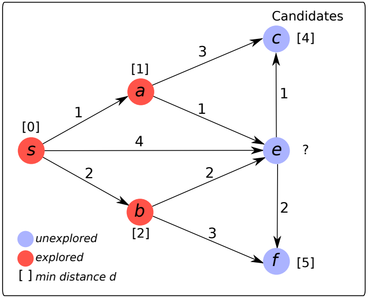
1: $S = \{s\}$ and $d(s) = 0$
2: **while** $S \neq V$ **do**
3: **for** every node $x \in V - S$ **do**
4: Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$
5: Set $v = \arg\min_{x \in V-S} d'(x)$
6: Add v to S and set $d(v) = d'(v)$



Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

$S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **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 $d(v) = d'(v)$

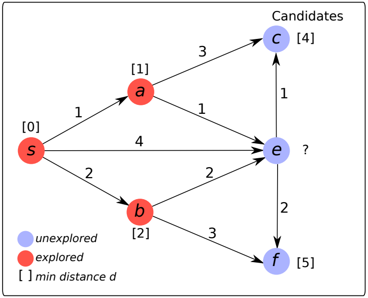


- How do we parse $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$?

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)
```

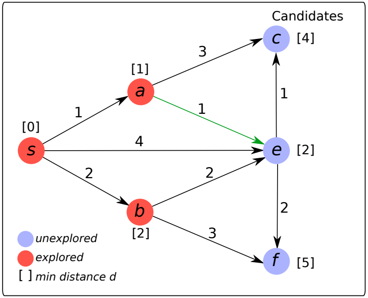


- How do we parse $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$?
  - The algorithm is examining a particular (unexplored) node x in $V - S$.

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)
```

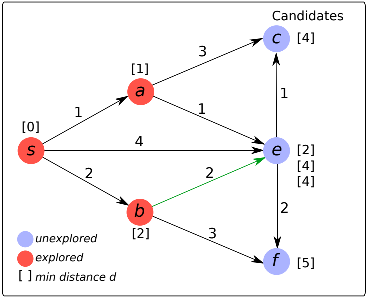


- How do we parse $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$?
  - The algorithm is examining a particular (unexplored) node x in $V - S$.
  - Argument of min runs over all edges of the type (u, x), where u is in S (i.e., u is explored).

Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

$S = \{s\}$ 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 $d(v) = d'(v)$

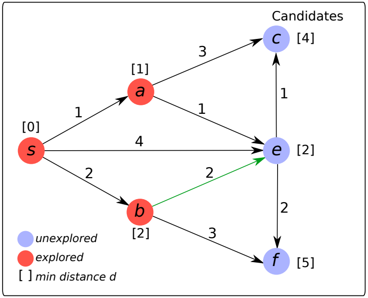


- How do we parse $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$?
  - The algorithm is examining a particular (unexplored) node x in $V - S$.
  - Argument of min runs over all edges of the type (u, x), where u is in S (i.e., u is explored).
  - For each edge (u, v), we compute the length of the shortest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

Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

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 $d(v) = d'(v)$

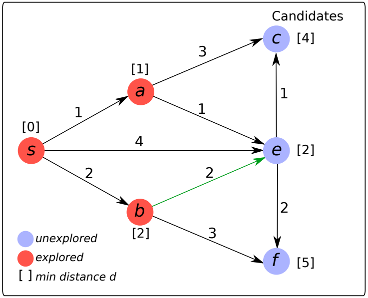


- How do we parse $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u,x))$?
  - The algorithm is examining a particular (unexplored) node x in $V - S$.
  - Argument of min runs over all edges of the type (u, x), where u is in S (i.e., u is explored).
  - For each edge (u, v), we compute the length of the shortest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

```
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)
```

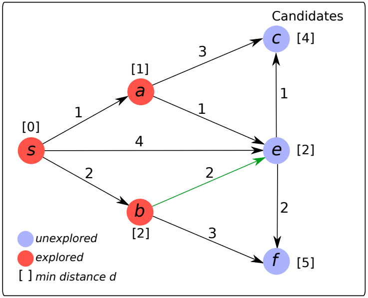


- How do we parse $v = \arg\min_{x \in V-S} d'(x)$?

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 $d(v) = d'(v)$

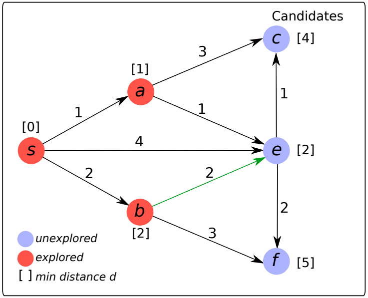


- How do we parse $v = \arg\min_{x \in V-S} d'(x)$?
  - Run over all (unexplored) nodes x in $V - S$.

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 $d(v) = d'(v)$

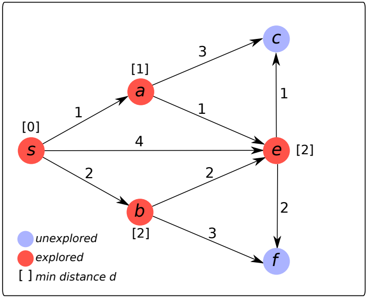


- How do we parse $v = \arg\min_{x \in V-S} d'(x)$?
  - Run over all (unexplored) nodes x in $V - S$.
  - Examine the d' values for these nodes.

Dijkstra's Algorithm

Dijkstra's Algorithm(G, l, s)

```
S = {s} and d(s) = 0
while S ≠ 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)
```

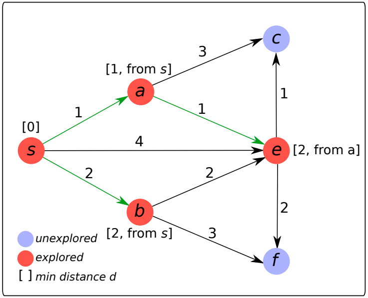


- How do we parse $v = \arg\min_{x \in V-S} d'(x)$?
  - Run over all (unexplored) nodes x in $V - S$.
  - Examine the d' values for these nodes.
  - Return the *argument* (i.e., the node) that has the smallest value of $d'(x)$.

Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM(G, l, s)

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.
  - 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

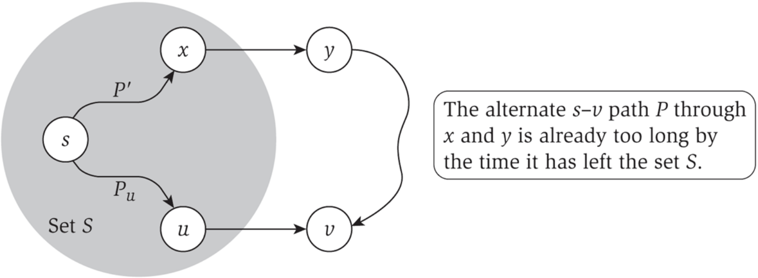
- Let $P(u)$ be the path computed by the algorithm for a node u.
- Claim: $P(u)$ is the shortest path from s to u.
- Prove by induction on the size of S, i.e., follow the algorithm.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive hypothesis: The algorithm has correctly computed $P(t)$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

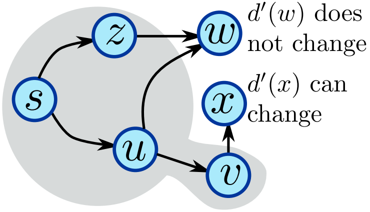
- Let $P(u)$ be the path computed by the algorithm for a node u.
- Claim: $P(u)$ is the shortest path from s to u.
- Prove by induction on the size of S, i.e., follow the algorithm.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive hypothesis: The algorithm has correctly computed $P(t)$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

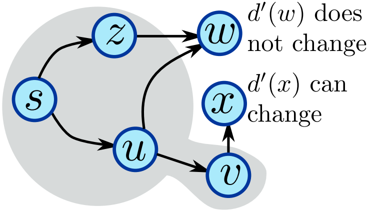
1: $S = \{s\}$ and $d(s) = 0$
2: **while** $S \neq V$ **do**
3: **for** every node $x \in V - S$ **do**
4: Set $d'(x) = \min_{(u,x):u \in S}(d(u) + l(u, x))$
5: Set $v = \arg\min_{x \in V-S} d'(x)$
6: Add v to S and set $d(v) = d'(v)$



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 $d(v) = d'(v)$

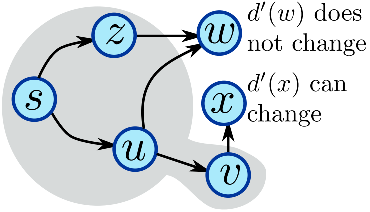


- Observation: If we add v to S, $d'(x)$ changes only if (v, x) is an edge in G.

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 $d(v) = d'(v)$

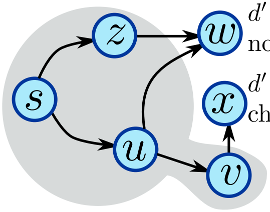


- Observation: If we add v to S, $d'(x)$ changes only if (v, x) is an edge in G.
- Idea: For each node $x \in V - S$, store the current value of $d'(x)$. Upon adding a node v to S, update $d'()$ only for neighbours of v.

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

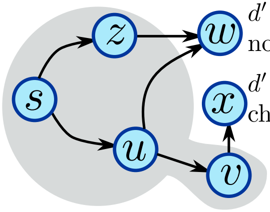
$d'(x)$ can change

- 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)

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.
- 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** $S \neq V$ **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$(Q, x, d'(x))$

- 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 ≠ V do
    (v, d'(v)) = EXTRACTMIN(Q)
    Add v to S and set d(v) = d'(v)
    for every node x ∈ 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

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? $n - 1$.

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: $\quad (v, d'(v)) = \text{EXTRACTMIN}(Q)$
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: $\qquad$ **if** $d(v) + l_{(v,x)} < d'(x)$ **then**
7: $\qquad\quad d'(x) = d(v) + l_{(v,x)}$
8: $\qquad\quad \text{CHANGEKEY}(Q, x, d'(x))$

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

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? $O(d_{\text{out}}(v))$, the number of *outgoing* neighbours of v.

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? $O(d_{\text{out}}(v))$, 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: $\quad (v, d'(v)) =$ 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? $\sum_{v \in V} O(d_{\text{out}}(v)) = O(m)$.

Running Time of 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))
```

- 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?

Running Time of 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))
```

- 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? $\leq m$.

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

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? $O(d_{\text{out}}(v))$, 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.

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

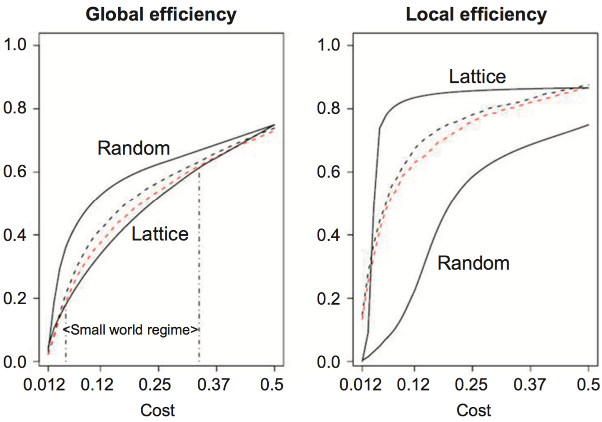
- *Global efficiency* $e_{\mathrm{glob}}(G)$ is the average of the reciprocal of the shortest path length between all pairs of nodes in G.

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

- *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.

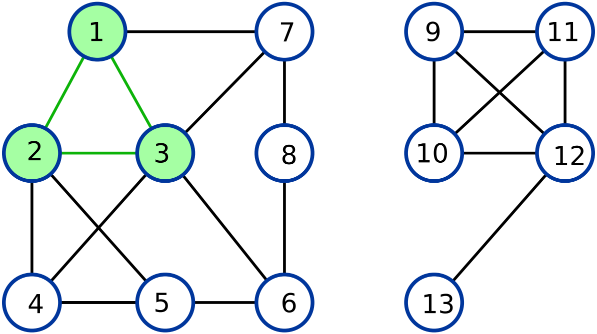
$$e_{\mathrm{loc}}(v) = \frac{1}{d(v)(d(v)-1)} \sum_{\substack{u,v \in N(v) \\ u \neq v}} \frac{1}{\delta(u, v)}$$

Efficiency in Brain Networks



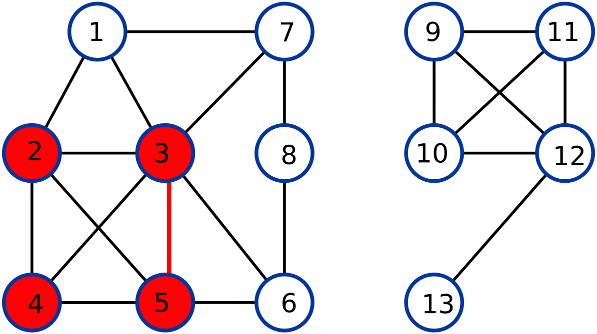
- Functional connectivity networks from fMRI data in young (black) and old (orange) human volunteers.
- *x*-axis is fraction of possible edges as threshold on edge weight varies.
- *y*-axis is global (left) and local (right) efficiency.
- Small world networks are both locally and globally efficient.

Defining Modules



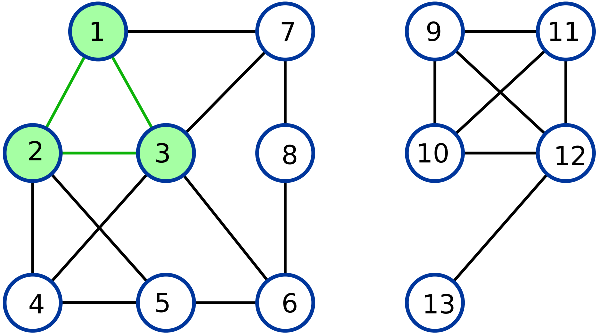
- How do we define a module in an undirected graph?
- In an undirected graph $G = (V, E)$, a subset of nodes $C \subseteq V$ is a *clique* or *complete subgraph* if for every pair of nodes $u, v \in C$, (u, v) is an edge in E.

Defining Modules



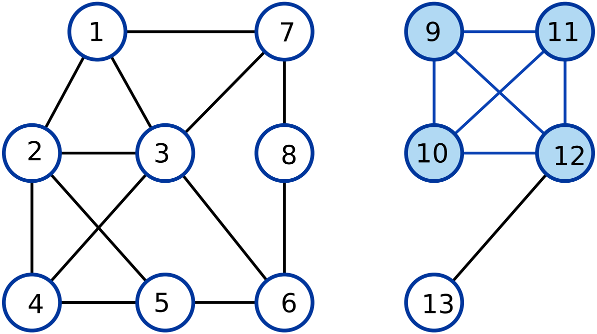
- How do we define a module in an undirected graph?
- In an undirected graph $G = (V, E)$, a subset of nodes $C \subseteq V$ is a *clique* or *complete subgraph* if for every pair of nodes $u, v \in C$, (u, v) is an edge in E.

Defining Modules



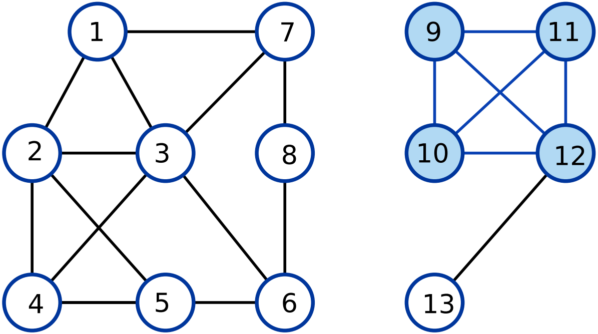
- How do we define a module in an undirected graph?
- In an undirected graph $G = (V, E)$, a subset of nodes $C \subseteq V$ is a *clique* or *complete subgraph* if for every pair of nodes $u, v \in C$, (u, v) is an edge in E.
  - A clique C is *maximal* if no node outside C can be added to it, i.e., for every node $x \in V - C$, x is not connected to at least one node in C.

Defining Modules



- How do we define a module in an undirected graph?
- In an undirected graph $G = (V, E)$, a subset of nodes $C \subseteq V$ is a *clique* or *complete subgraph* if for every pair of nodes $u, v \in C$, (u, v) is an edge in E.
  - A clique C is *maximal* if no node outside C can be added to it, i.e., for every node $x \in V - C$, x is not connected to at least one node in C.
  - A clique C is *maximum* if there is no clique C' in G with more nodes than C.

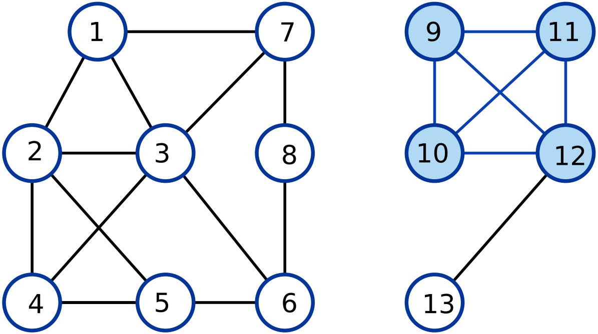
Computing a Maximum Clique



MAXIMUM CLIQUE

Given an undirected, unweighted graph $G(V, E)$, compute the largest clique in G.

Computing a Maximum Clique

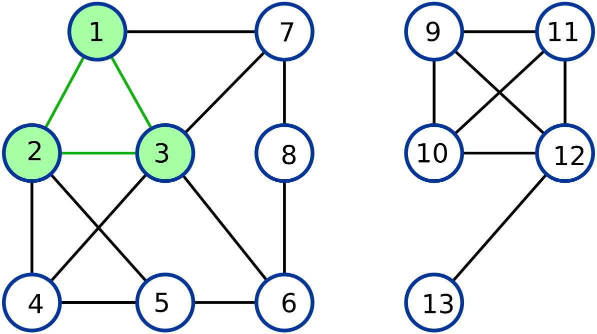


Maximum Clique

Given an undirected, unweighted graph $G(V, E)$, compute the largest clique in G.

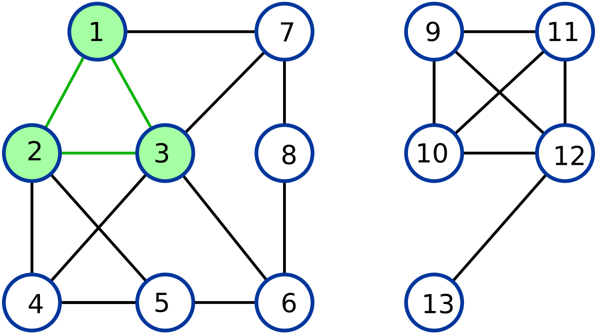
- Computing a maximum clique is NP-hard.
- Any algorithm that can provably compute the maximum clique is likely to have a running time that is exponential in the size of the graph.

Computing a Maximal Clique



MAXIMAL CLIQUE
Given an undirected, unweighted graph $G(V, E)$, compute a maximal clique in G.

Computing a Maximal Clique

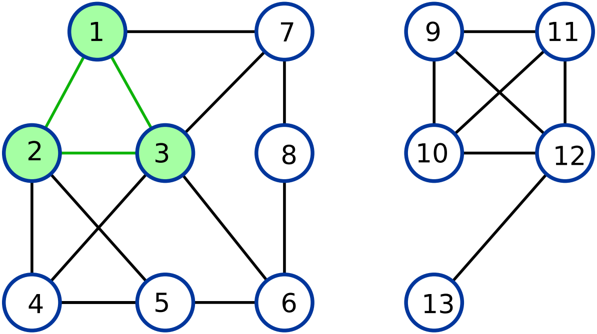


MAXIMAL CLIQUE

Given an undirected, unweighted graph $G(V, E)$, compute a maximal clique in G.

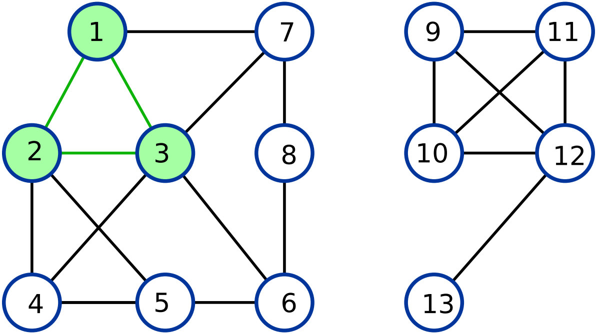
1. Select an arbitrary node v and add it to S (the clique we will output).
2. If there is a node u in $V - S$ that is connected to every node in S, add u to S.
3. Repeat the previous step until no such node u is found.

Running Time to Compute a Maximal Clique



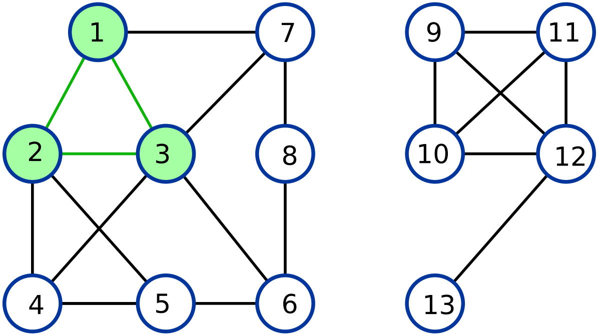
1. Select an arbitrary node v and add it to S (the clique we will output).
2. If there is a node u in $V - S$ that is connected to every node in S, add u to S.
3. Repeat the previous step until no such node u is found.

Running Time to Compute a Maximal Clique



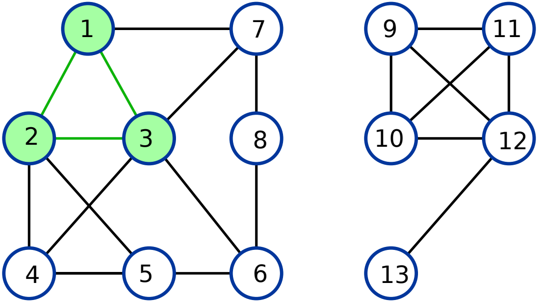
1. Select an arbitrary node v and add it to S (the clique we will output).
2. If there is a node u in $V - S$ that is connected to every node in S, add u to S. $O(n|S|)$ checks for edge existence.
3. Repeat the previous step until no such node u is found.

Running Time to Compute a Maximal Clique



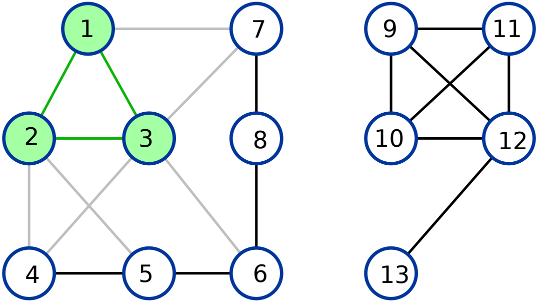
1. Select an arbitrary node v and add it to S (the clique we will output).
2. If there is a node u in $V - S$ that is connected to every node in S, add u to S. $O(n|S|)$ checks for edge existence.
3. Repeat the previous step until no such node u is found. $O(n|S|^2)$ checks for edge existence.

Clique Decomposition



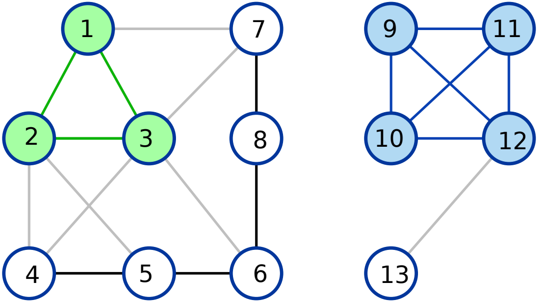
- What do we do after computing a maximal clique?

Clique Decomposition



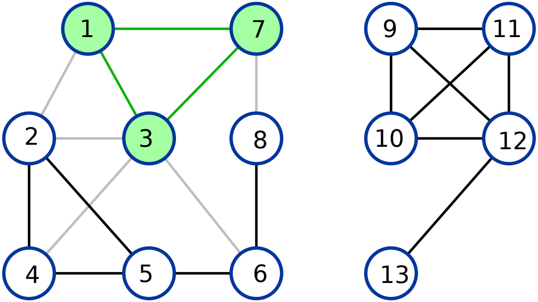
- What do we do after computing a maximal clique?
- Delete nodes in that clique from the graph and repeat.

Clique Decomposition



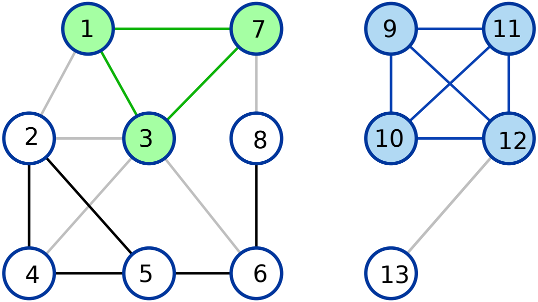
- What do we do after computing a maximal clique?
- Delete nodes in that clique from the graph and repeat.
- The resulting set of cliques forms a *clique decomposition* of G.

Clique Decomposition



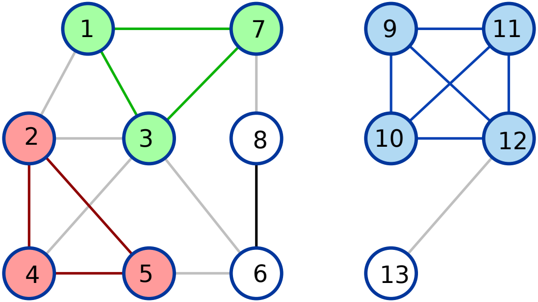
- What do we do after computing a maximal clique?
- Delete nodes in that clique from the graph and repeat.
- The resulting set of cliques forms a *clique decomposition* of G.
- Sequence of cliques found depends on order of processing nodes.

Clique Decomposition



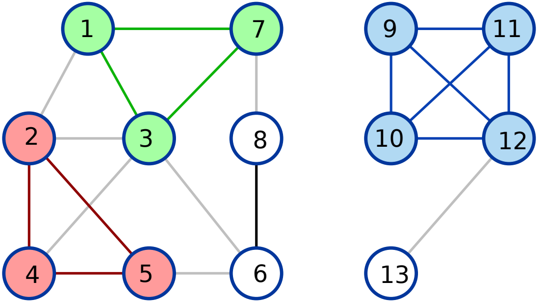
- What do we do after computing a maximal clique?
- Delete nodes in that clique from the graph and repeat.
- The resulting set of cliques forms a *clique decomposition* of G.
- Sequence of cliques found depends on order of processing nodes.

Clique Decomposition



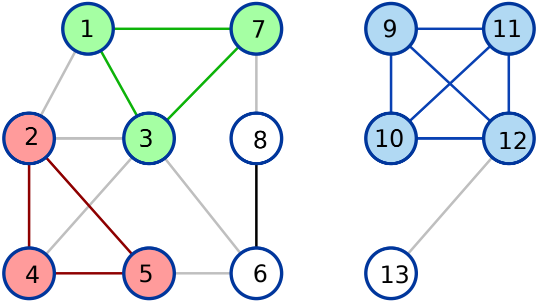
- What do we do after computing a maximal clique?
- Delete nodes in that clique from the graph and repeat.
- The resulting set of cliques forms a *clique decomposition* of G.
- Sequence of cliques found depends on order of processing nodes.

Clique Decomposition



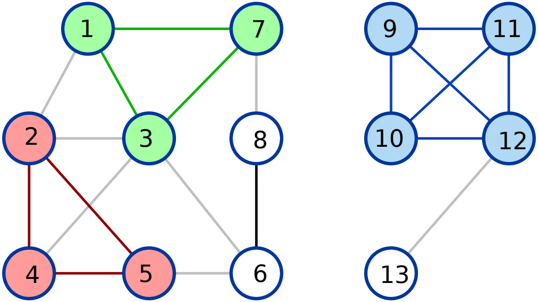
- What do we do after computing a maximal clique?
- Delete nodes in that clique from the graph and repeat.
- The resulting set of cliques forms a *clique decomposition* of G.
- Sequence of cliques found depends on order of processing nodes.
- There is no notion of correctness here since we defined what we compute (the clique decomposition) based on an algorithm we specified.

Clique Decomposition



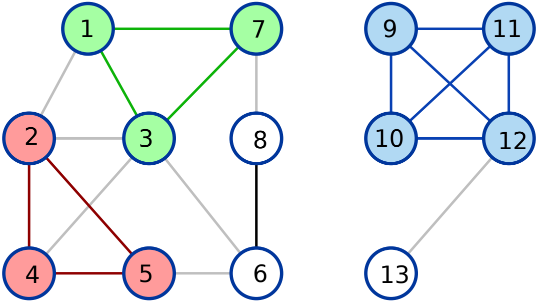
- What do we do after computing a maximal clique?
- Delete nodes in that clique from the graph and repeat.
- The resulting set of cliques forms a *clique decomposition* of G.
- Sequence of cliques found depends on order of processing nodes.
- There is no notion of correctness here since we defined what we compute (the clique decomposition) based on an algorithm we specified.
- Will every edge in G be in some clique in the decomposition? Can a node be in multiple cliques?

Clique Decomposition



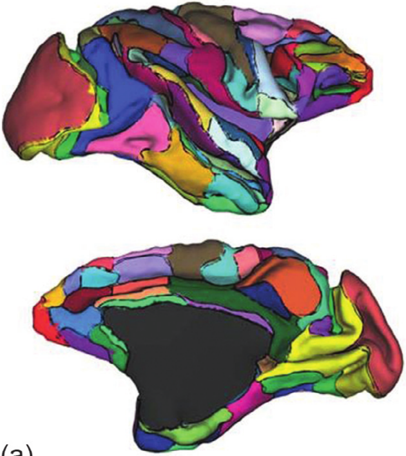
- What do we do after computing a maximal clique?
- Delete nodes in that clique from the graph and repeat.
- The resulting set of cliques forms a *clique decomposition* of G.
- Sequence of cliques found depends on order of processing nodes.
- There is no notion of correctness here since we defined what we compute (the clique decomposition) based on an algorithm we specified.
- Will every edge in G be in some clique in the decomposition? Can a node be in multiple cliques? No, to both questions.

Clique Decomposition



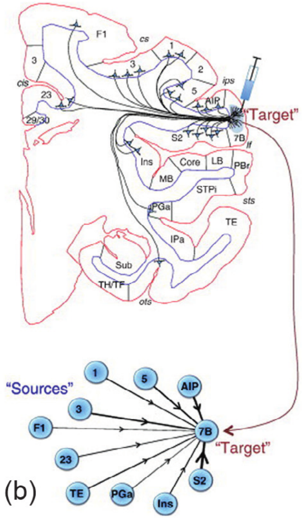
- What do we do after computing a maximal clique?
- Delete nodes in that clique from the graph and repeat.
- The resulting set of cliques forms a *clique decomposition* of G.
- Sequence of cliques found depends on order of processing nodes.
- There is no notion of correctness here since we defined what we compute (the clique decomposition) based on an algorithm we specified.
- Will every edge in G be in some clique in the decomposition? Can a node be in multiple cliques? No, to both questions.
- Modification: After finding a clique, delete only the edges in it.

Structural Connectivity at the Mesoscale


(a)

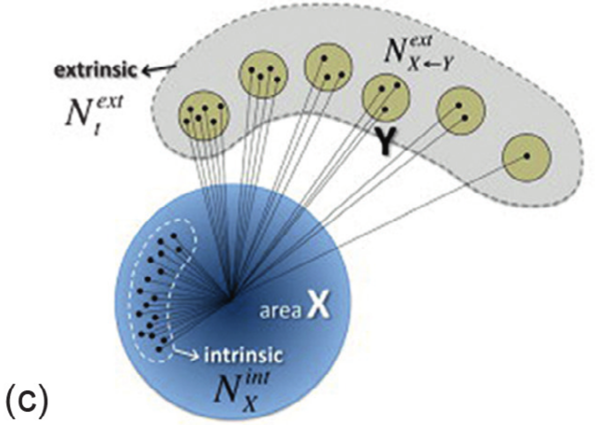
Parcellate the macaque cortex into 91 areas, defined according to cytoarchitecture and sulco-gyral landmarks.

Structural Connectivity at the Mesoscale



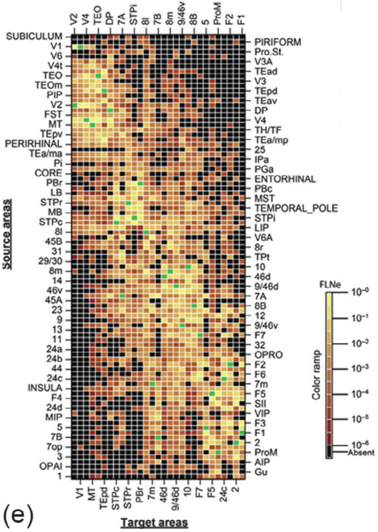
Use retrograde tract tracing. Determine edges coming into node representing area of injection from "labelled" nodes representing neurons that the tracer reaches.

Structural Connectivity at the Mesoscale



Injection is at X: $w(Y, X) = \frac{\text{number of neurons labelled in } Y}{\text{total number of labelled neurons}}$

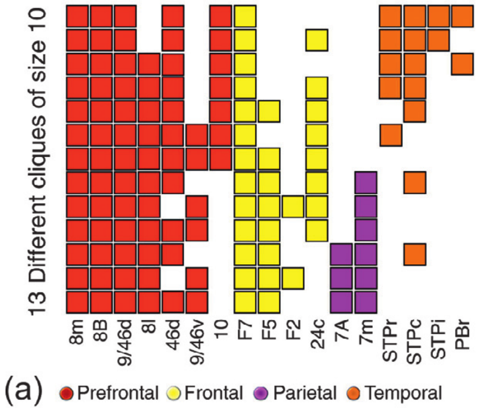
Structural Connectivity at the Mesoscale



Example of connectivity matrix.

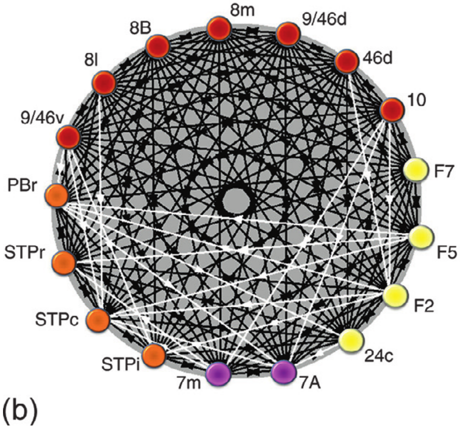
Edge weights range over six orders of magnitude.

Cliques in Macaque Cerebral Cortex Connectome



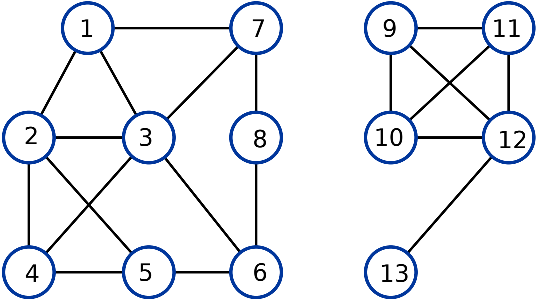
- 29-node directed graph representing connectome of the cerebral cortex of the macaque; only considering nodes with tracer injection points.
- Computed all 13 maximum cliques, each of which had 10 nodes.

Cliques in Macaque Cerebral Cortex Connectome



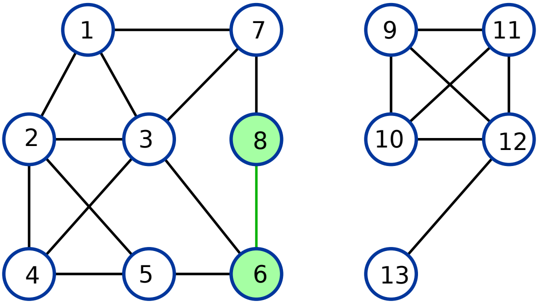
- 29-node directed graph representing connectome of the cerebral cortex of the macaque; only considering nodes with tracer injection points.
- Computed all 13 maximum cliques, each of which had 10 nodes.
- Union of cliques formed a dense subgraph among 17 nodes.

k-Cores



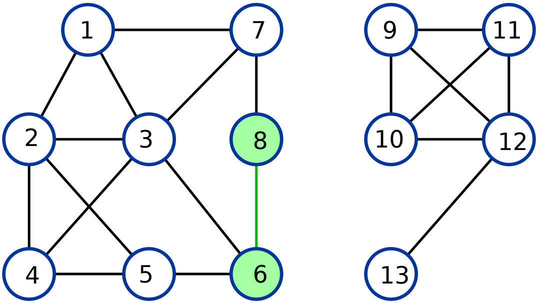
- In an undirected graph $G = (V, E)$, a subset of nodes $C \subseteq V$ is a *k-core* if every node $u \in C$ is connected in G to at least k nodes in C.

k-Cores



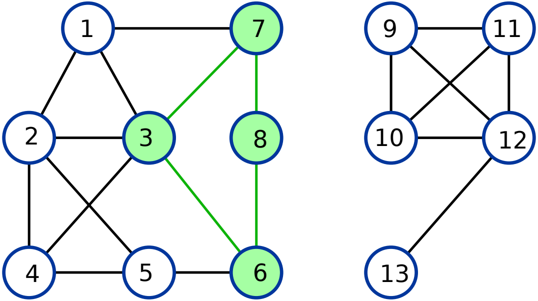
- In an undirected graph $G = (V, E)$, a subset of nodes $C \subseteq V$ is a *k-core* if every node $u \in C$ is connected in G to at least k nodes in C.
- What is largest the 1-core of G?

k-Cores



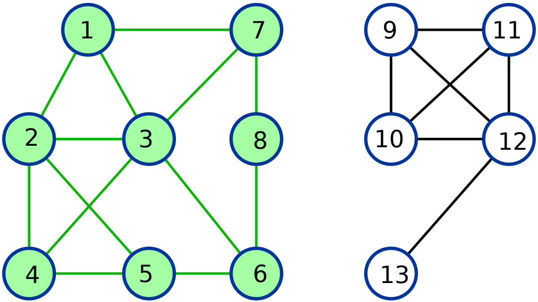
- In an undirected graph $G = (V, E)$, a subset of nodes $C \subseteq V$ is a *k-core* if every node $u \in C$ is connected in G to at least k nodes in C.
- What is largest the 1-core of G? G itself (without any nodes of degree zero).

k-Cores



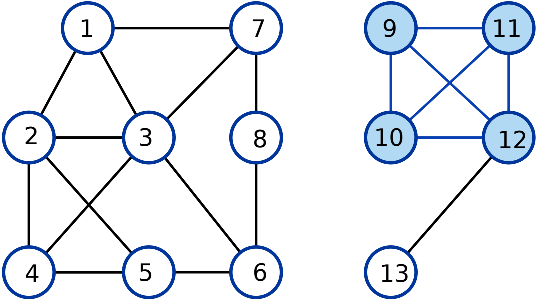
- In an undirected graph $G = (V, E)$, a subset of nodes $C \subseteq V$ is a *k-core* if every node $u \in C$ is connected in G to at least k nodes in C.
- What is largest the 1-core of G? G itself (without any nodes of degree zero).

k-Cores



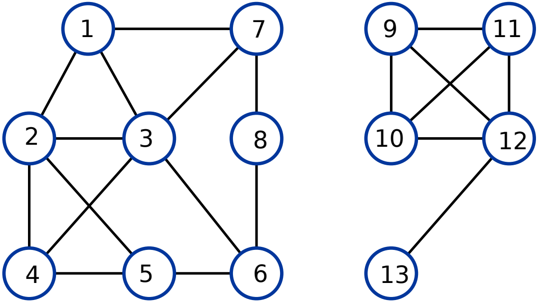
- In an undirected graph $G = (V, E)$, a subset of nodes $C \subseteq V$ is a *k-core* if every node $u \in C$ is connected in G to at least k nodes in C.
- What is largest the 1-core of G? G itself (without any nodes of degree zero).

k-Cores



- In an undirected graph $G = (V, E)$, a subset of nodes $C \subseteq V$ is a *k-core* if every node $u \in C$ is connected in G to at least k nodes in C.
- What is largest the 1-core of G? G itself (without any nodes of degree zero).
- Does this graph have a 4-core?

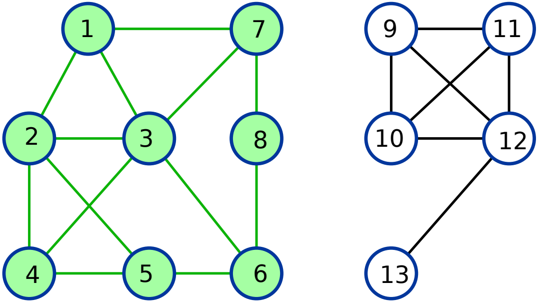
Problems related to k-cores



k-CORE EXISTENCE

Given an undirected, unweighted graph $G(V, E)$ and an integer k, compute the k-core with the largest number of nodes in G.

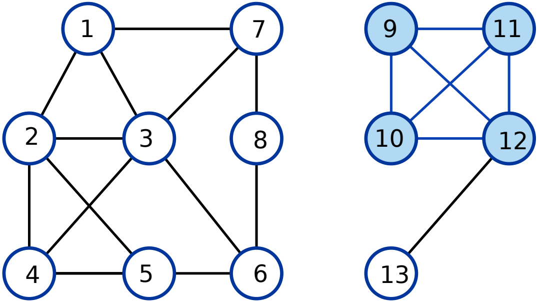
Problems related to k-cores



k-core Existence

Given an undirected, unweighted graph $G(V, E)$ and an integer k, compute the k-core with the largest number of nodes in G.

Problems related to k-cores



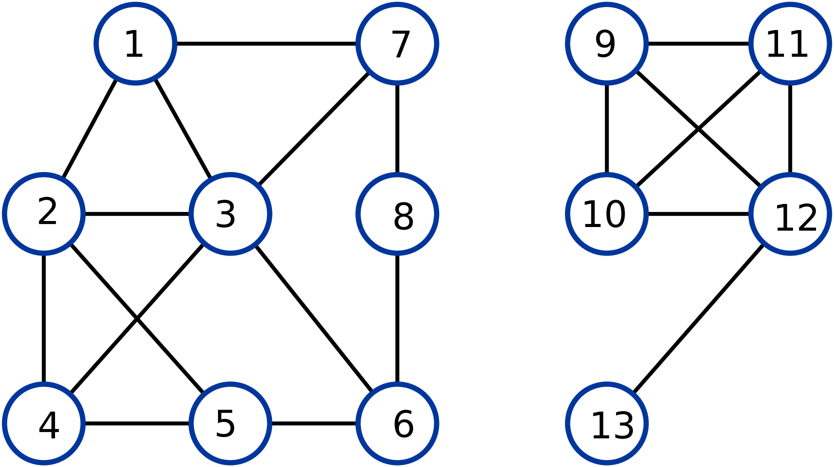
k-core Existence

Given an undirected, unweighted graph $G(V, E)$ and an integer k, compute the k-core with the largest number of nodes in G.

Largest k-core

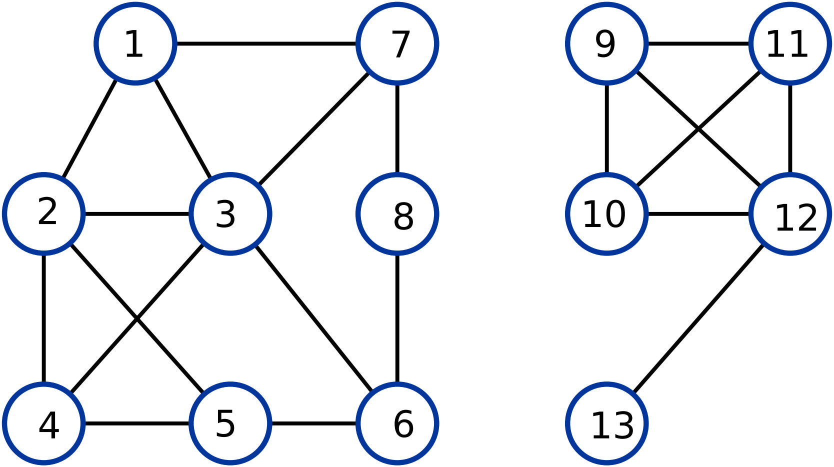
Given an undirected, unweighted graph $G(V, E)$, compute the largest value of k for which G contains a k-core.

Algorithm for k-Core Existence



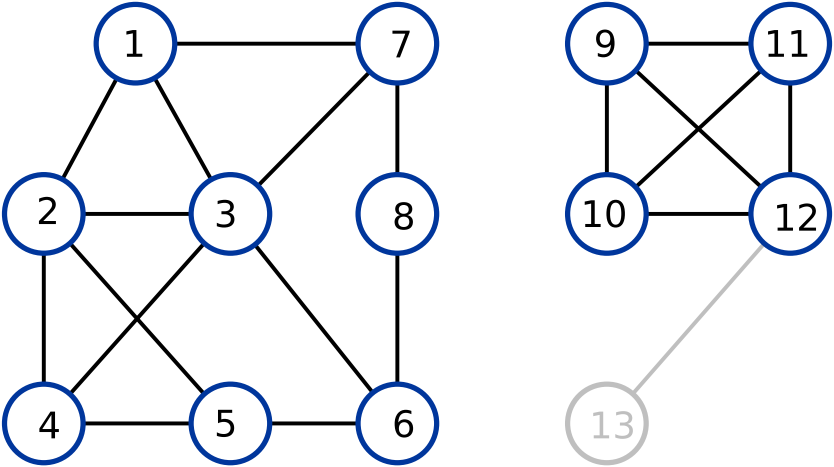
- Repeatedly delete all nodes of degree $< k$ until

Algorithm for k-Core Existence



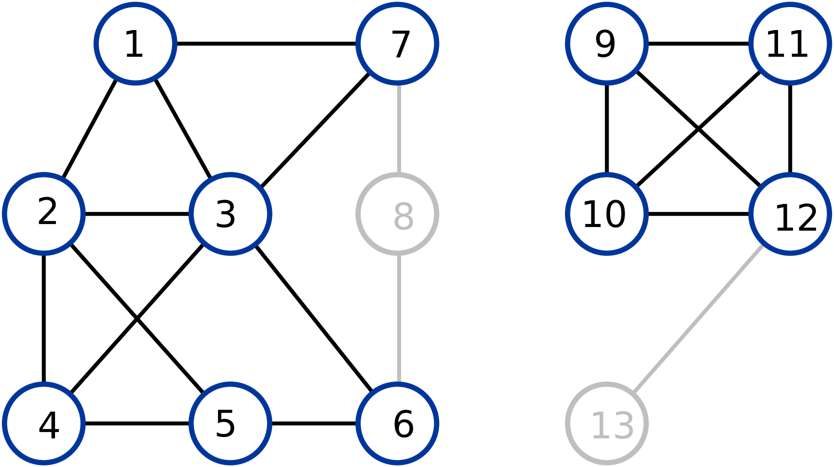
- Repeatedly delete all nodes of degree $< k$ until every remaining node has degree $\geq k$.
- Resulting graph is the largest k-core.

Algorithm for k-Core Existence



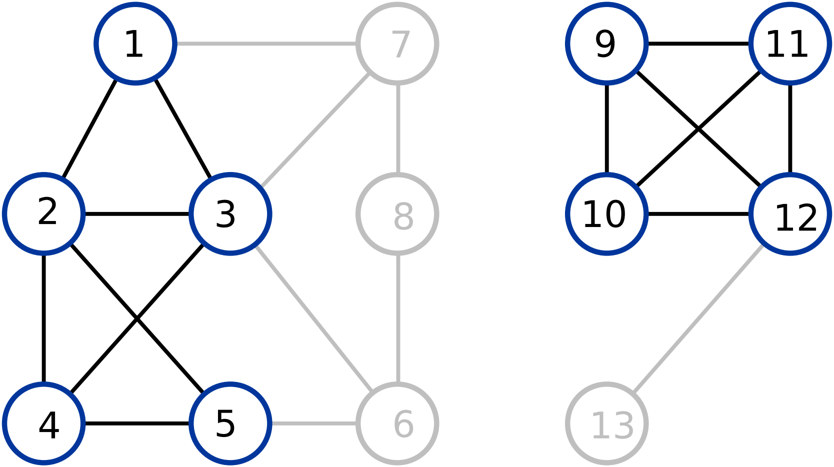
- Repeatedly delete all nodes of degree $< k$ until every remaining node has degree $\geq k$.
- Resulting graph is the largest k-core.

Algorithm for k-Core Existence



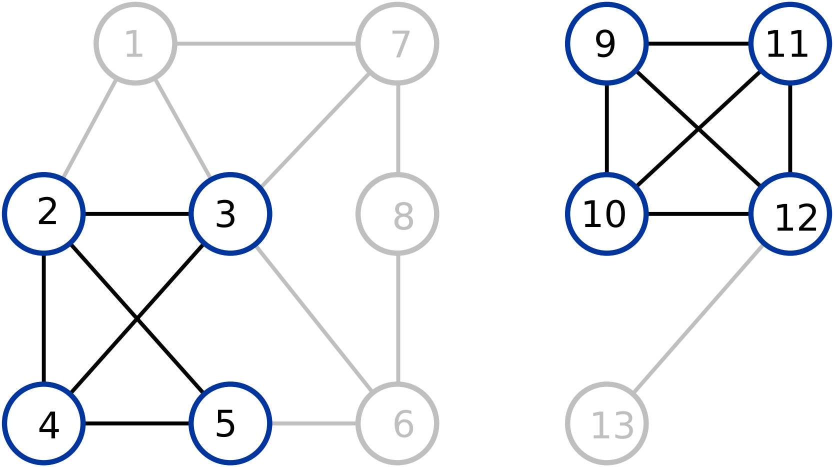
- Repeatedly delete all nodes of degree $< k$ until every remaining node has degree $\geq k$.
- Resulting graph is the largest k-core.

Algorithm for k-Core Existence



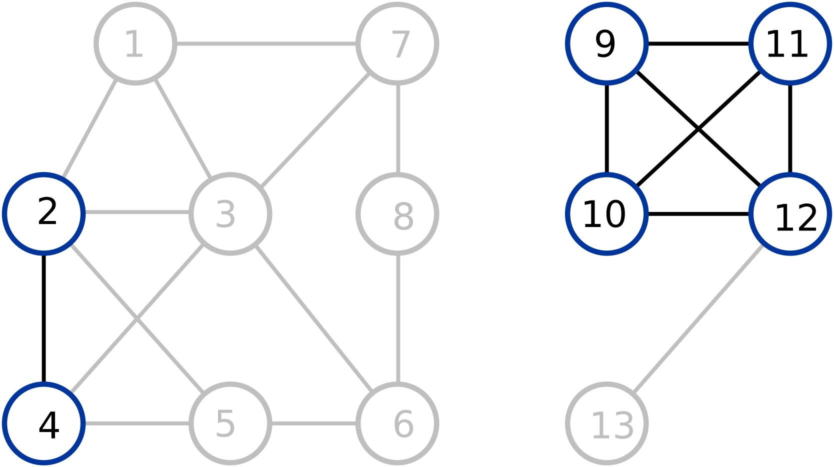
- Repeatedly delete all nodes of degree $< k$ until every remaining node has degree $\geq k$.
- Resulting graph is the largest k-core.

Algorithm for k-Core Existence



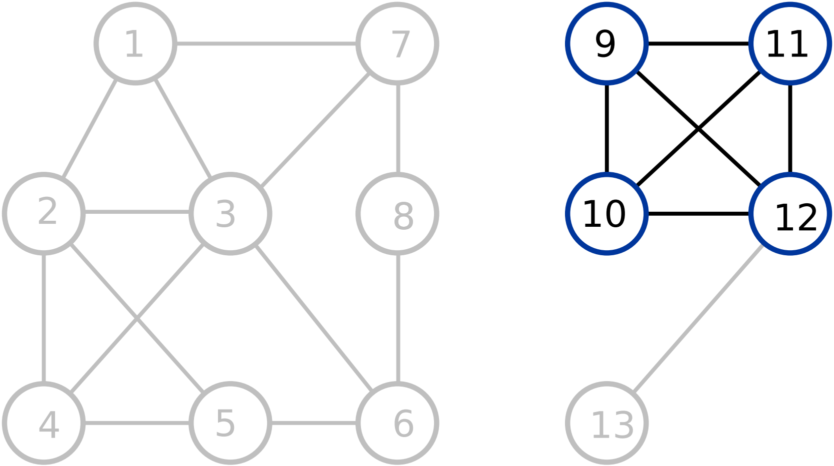
- Repeatedly delete all nodes of degree $< k$ until every remaining node has degree $\geq k$.
- Resulting graph is the largest k-core.

Algorithm for k-Core Existence



- Repeatedly delete all nodes of degree $< k$ until every remaining node has degree $\geq k$.
- Resulting graph is the largest k-core.

Algorithm for k-Core Existence



- Repeatedly delete all nodes of degree $< k$ until every remaining node has degree $\geq k$.
- Resulting graph is the largest k-core.

Correctness of k-Core Existence Algorithm

- Repeatedly delete all nodes of degree $< k$ until every remaining node has degree $\geq k$.
- Why should the resulting graph H be a k-core?
- Why should the resulting graph H be the k-core with the largest number of nodes?

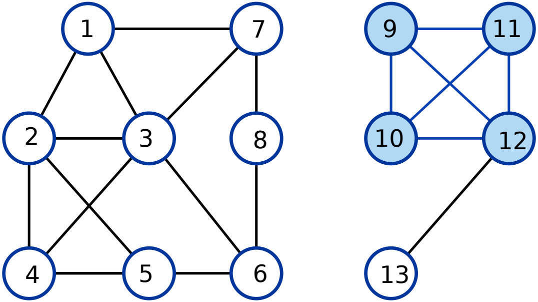
Correctness of k-Core Existence Algorithm

- Repeatedly delete all nodes of degree $< k$ until every remaining node has degree $\geq k$.
- Why should the resulting graph H be a k-core?
- Why should the resulting graph H be the k-core with the largest number of nodes?
- Proof by contradiction.
  - Suppose there is a k-core H' with more nodes than H.
  - Then $H \cup H'$ is also a k-core.
  - Moreover, no node in H' will be deleted by the algorithm.

Correctness of k-Core Existence Algorithm

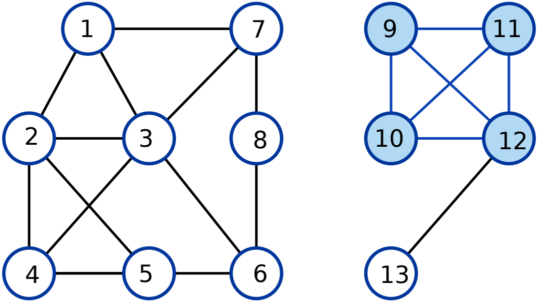
- Repeatedly delete all nodes of degree $< k$ until every remaining node has degree $\geq k$.
- Why should the resulting graph H be a k-core?
- Why should the resulting graph H be the k-core with the largest number of nodes?
- Proof by contradiction.
  - Suppose there is a k-core H' with more nodes than H.
  - Then $H \cup H'$ is also a k-core.
  - Moreover, no node in H' will be deleted by the algorithm.
- How do we implement k-core algorithm efficiently?

Cores vs. Cliques



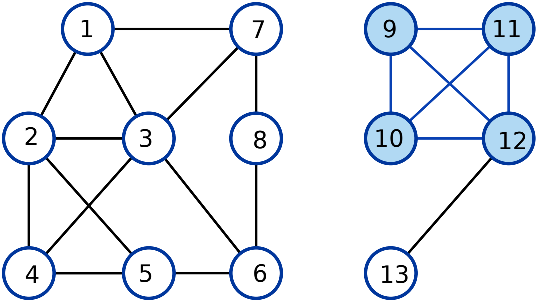
- A clique with k nodes is a $(k-1)$-core.
- Can we use the k-core algorithm to find maximum cliques?

Cores vs. Cliques



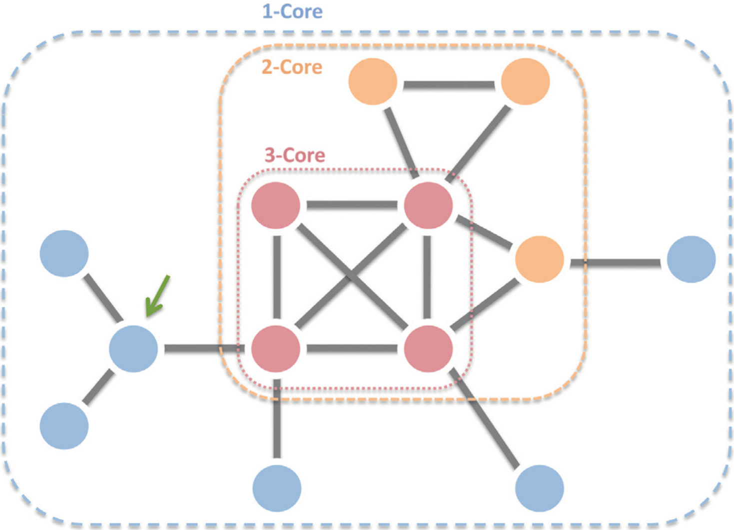
- A clique with k nodes is a $(k-1)$-core.
- Can we use the k-core algorithm to find maximum cliques?
- Idea: Compute the largest value of k for which a k-core H exists. If H is a clique, it must be the largest clique (of size $k+1$) in the graph.

Cores vs. Cliques



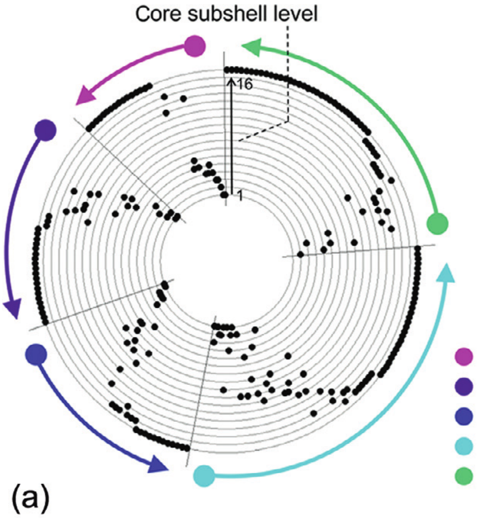
- A clique with k nodes is a $(k-1)$-core.
- Can we use the k-core algorithm to find maximum cliques?
- Idea: Compute the largest value of k for which a k-core H exists. If H is a clique, it must be the largest clique (of size $k+1$) in the graph.
- Flaw is that H may not be a clique, in general. The largest clique may be disjoint from H or be a subgraph of H.
- Moreover, the maximum clique may have l nodes while there may be a k-core where $k > l - 1$, e.g., $k = 3$ and $l = 3$. Create such an example.

k-Core Decomposition



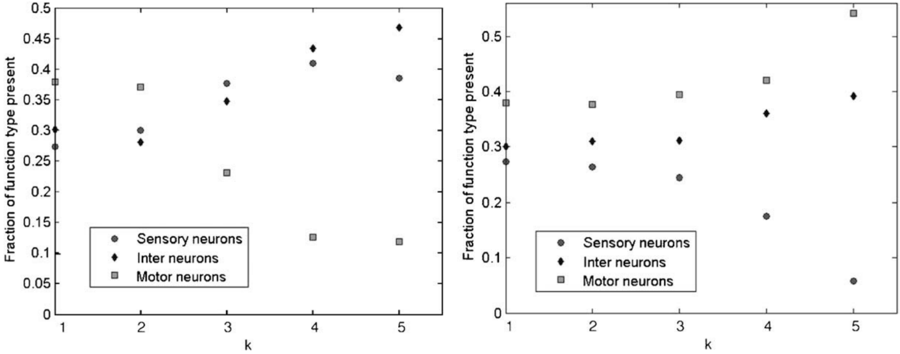
- Label each node by the k-core to which it belongs.

k-Core Decomposition of Macaque Cortex



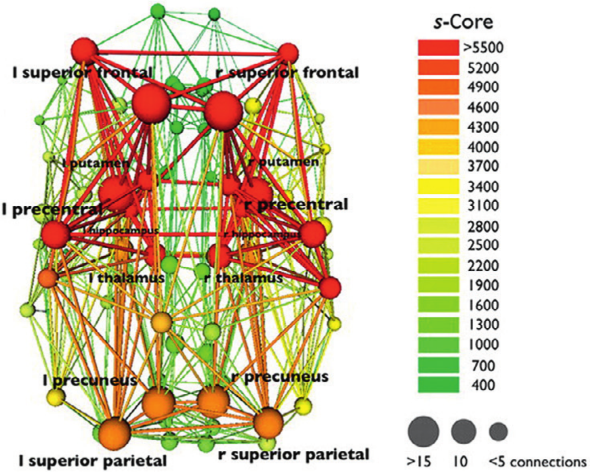
- 242-region macaque cortical connectome containing a 16-core.

k-Core Decomposition of C. Elegans Connectome



- Sensory neurons comprise the innermost cores based on out-degree.
- Motor neurons comprise the inner-most cores based on in-degree.

s-Core Decomposition of Human Connectome



- Structural connectivity from diffusion tensor imaging.
- Connectome is the average of 21 individuals.
- Extend k-core algorithm to weighted networks.