Ayudantía 12

BFS, Dijsktra y Prim

Ignacio Porte - ignacio.porte@uc.cl

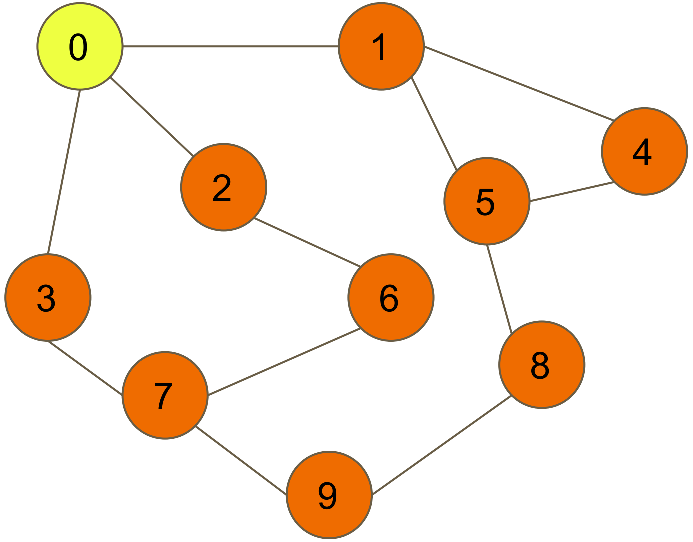
Nicholas Mc-Donnell - namcdonnell@uc.cl

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```

BFS

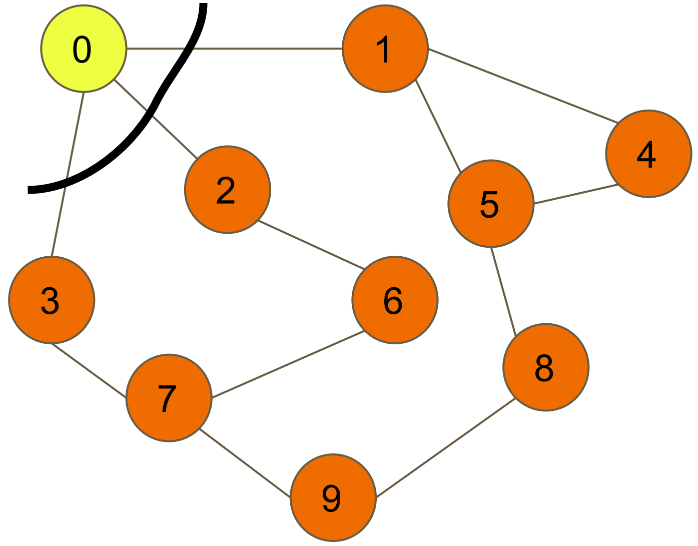
```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



🟨	Current Node
🟪	Visited Node
🟩	Done Node

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



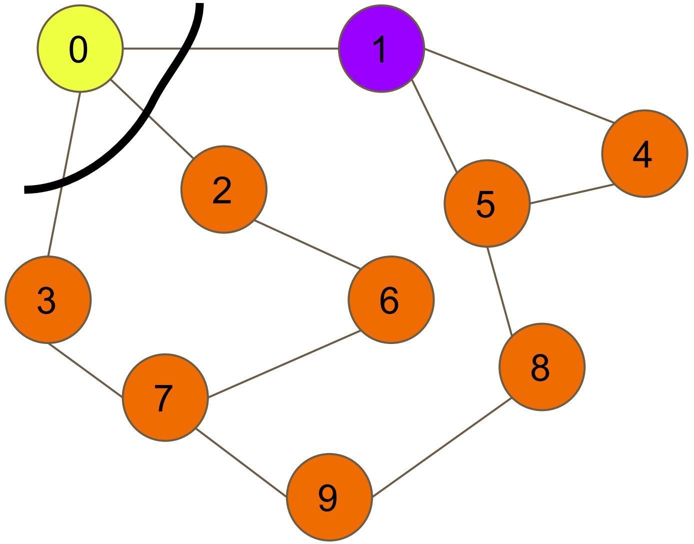
🟡 Current Node

🟣 Visited Node

🟢 Done Node

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



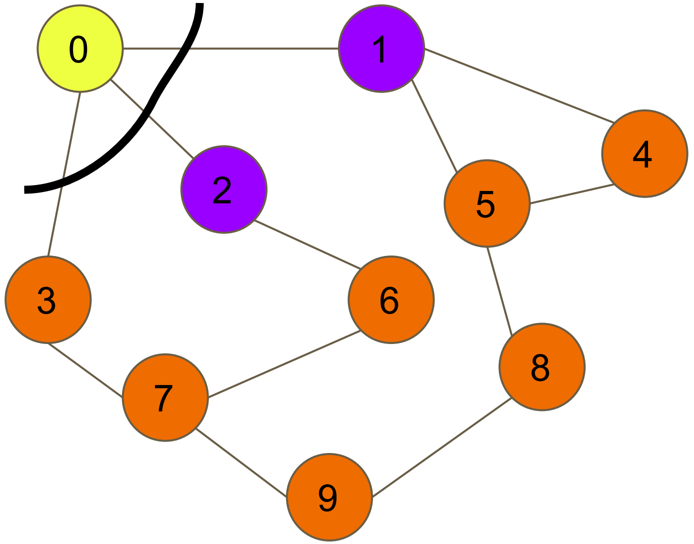
🟨 Current Node

🟪 Visited Node

🟦 Done Node

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



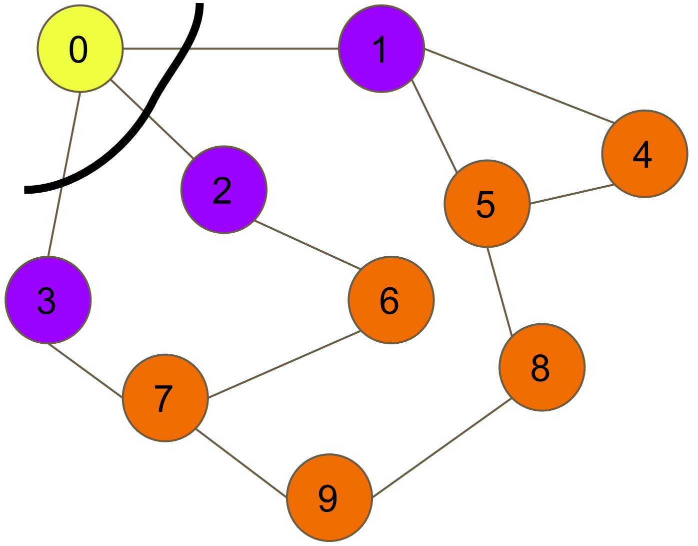
🟨 Current Node

🟪 Visited Node

🟦 Done Node

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



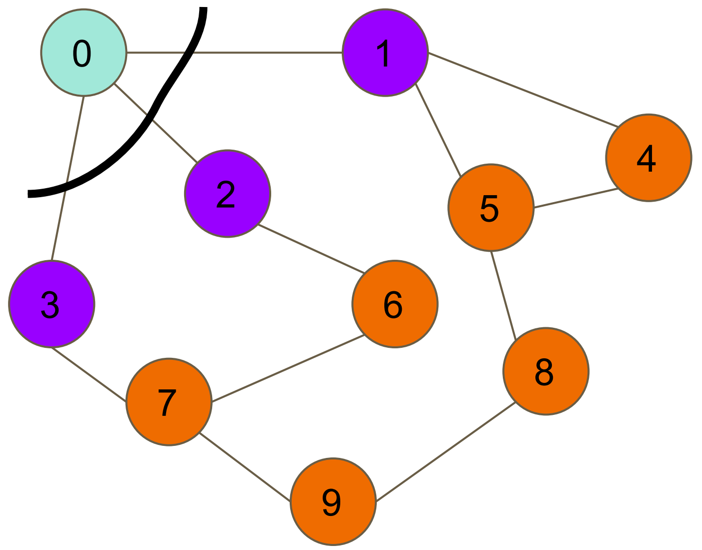
🟡 Current Node

🟣 Visited Node

🟢 Done Node

BFS

```
bfs(graph G, node start)
    Queue q
    q.push(start)
    mark start as visited
    while not q.empty()
        u = q.pop()
        for v in G.adjacent[u]
            if v not visited
                mark v as visited
                q.push(v)
        mark u as Done
```



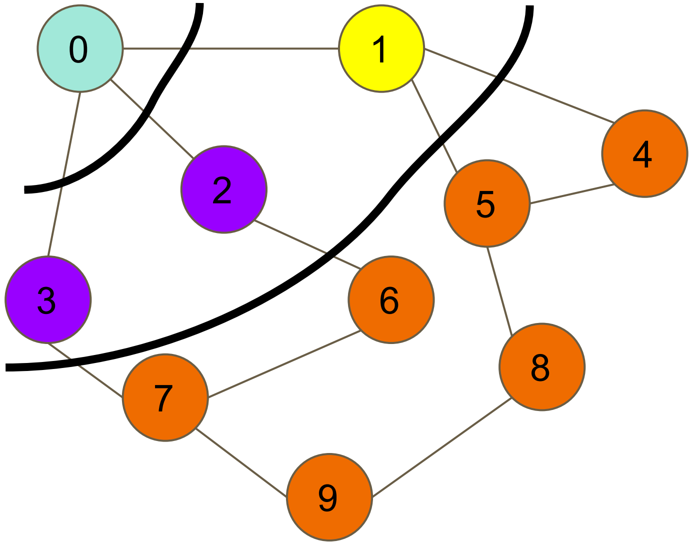
🟡 Current Node

🟣 Visited Node

🟢 Done Node

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



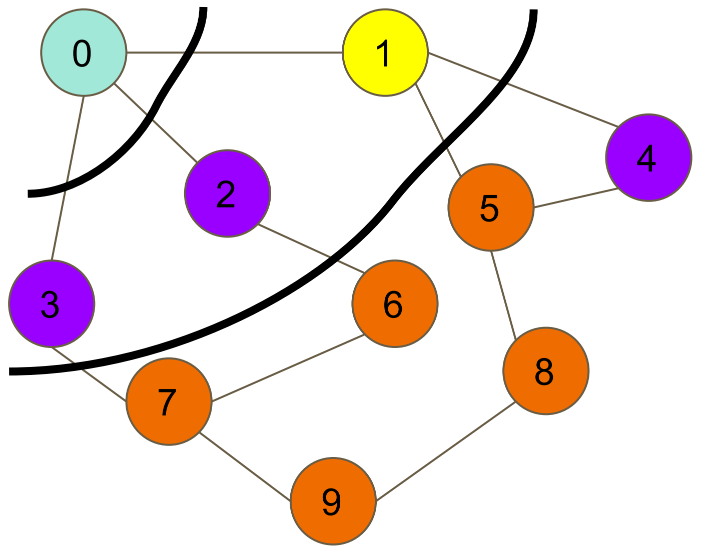
🟡 Current Node

🟣 Visited Node

🟢 Done Node

BFS

```
bfs(graph G, node start)
    Queue q
    q.push(start)
    mark start as visited
    while not q.empty()
        u = q.pop()
        for v in G.adjacent[u]
            if v not visited
                mark v as visited
                q.push(v)
        mark u as Done
```



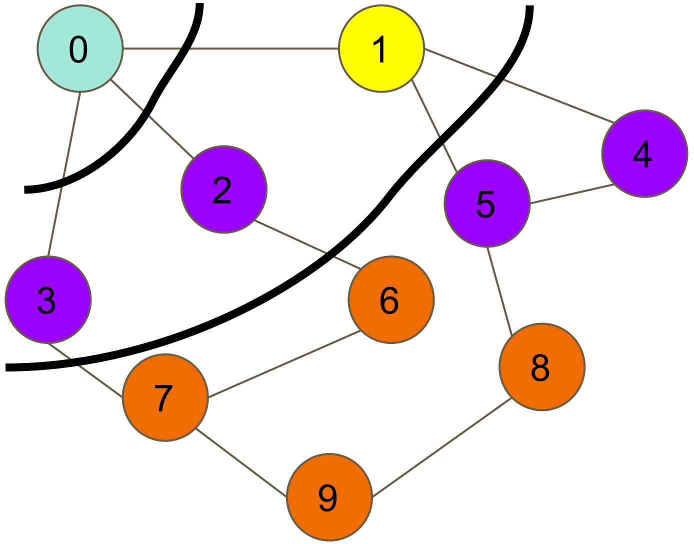
🟨 Current Node

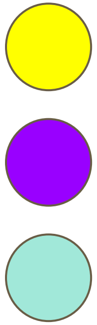
🟪 Visited Node

🟦 Done Node

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```





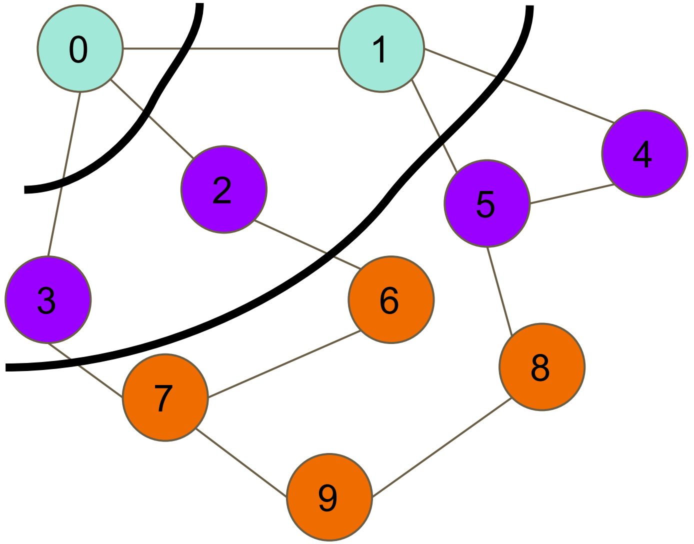
Current Node

Visited Node

Done Node

BFS

```
bfs(graph G, node start)
    Queue q
    q.push(start)
    mark start as visited
    while not q.empty()
        u = q.pop()
        for v in G.adjacent[u]
            if v not visited
                mark v as visited
                q.push(v)
        mark u as Done
```



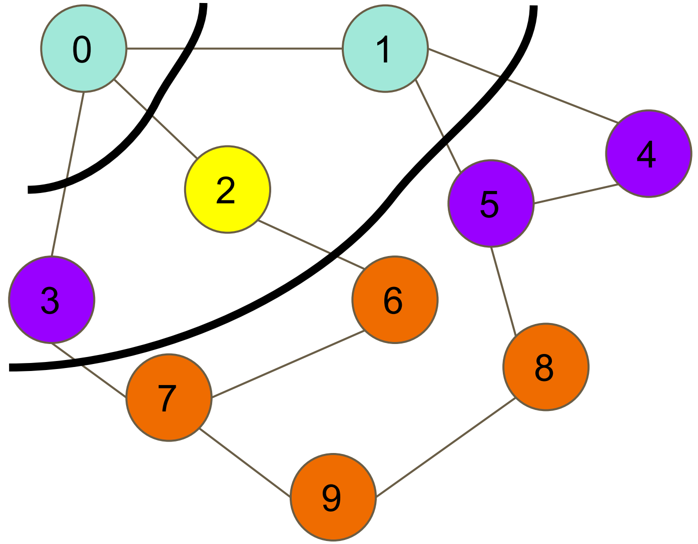
🟡 Current Node

🟣 Visited Node

🟢 Done Node

BFS

```
bfs(graph G, node start)
    Queue q
    q.push(start)
    mark start as visited
    while not q.empty()
        u = q.pop()
        for v in G.adjacent[u]
            if v not visited
                mark v as visited
                q.push(v)
        mark u as Done
```



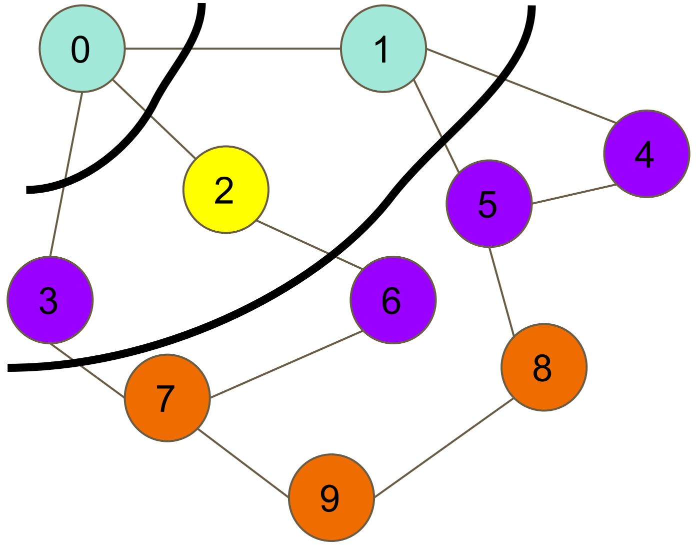
🟡 Current Node

🟣 Visited Node

🟢 Done Node

BFS

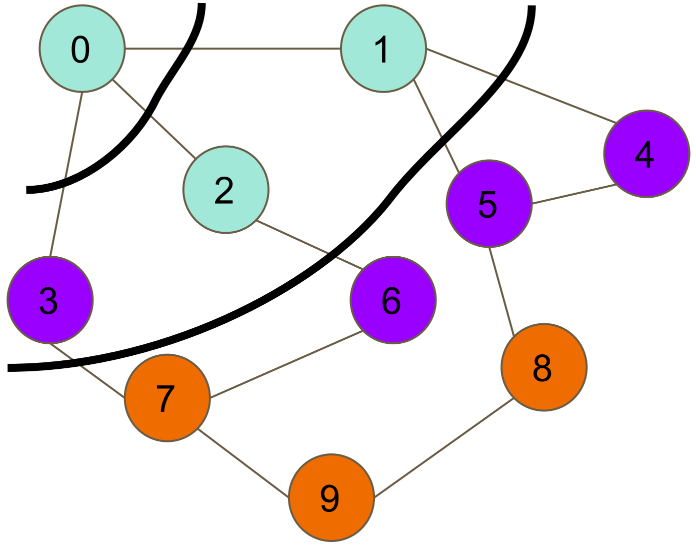
```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



🟨	Current Node
🟪	Visited Node
🟦	Done Node

BFS

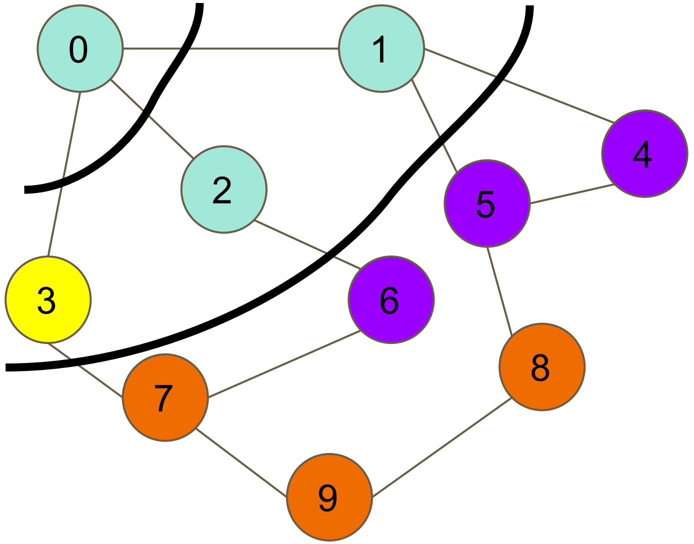
```
bfs(graph G, node start)
    Queue q
    q.push(start)
    mark start as visited
    while not q.empty()
        u = q.pop()
        for v in G.adjacent[u]
            if v not visited
                mark v as visited
                q.push(v)
        mark u as Done
```

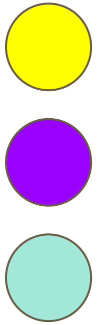


- 🟡 Current Node
- 🟣 Visited Node
- 🟢 Done Node

BFS

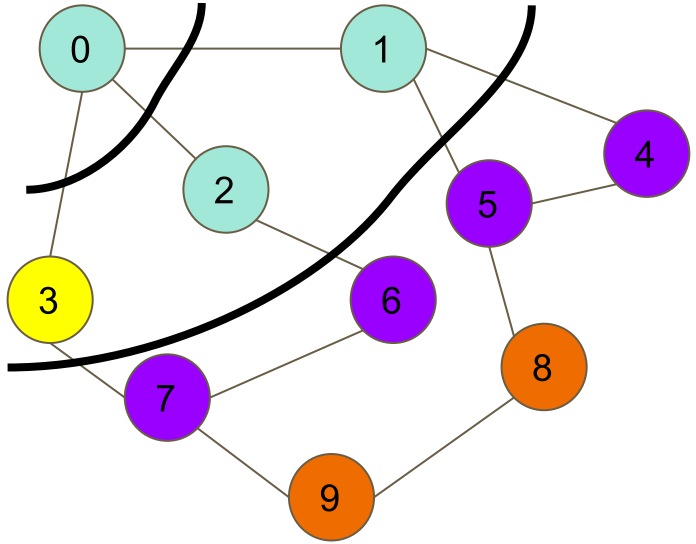
```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```





BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



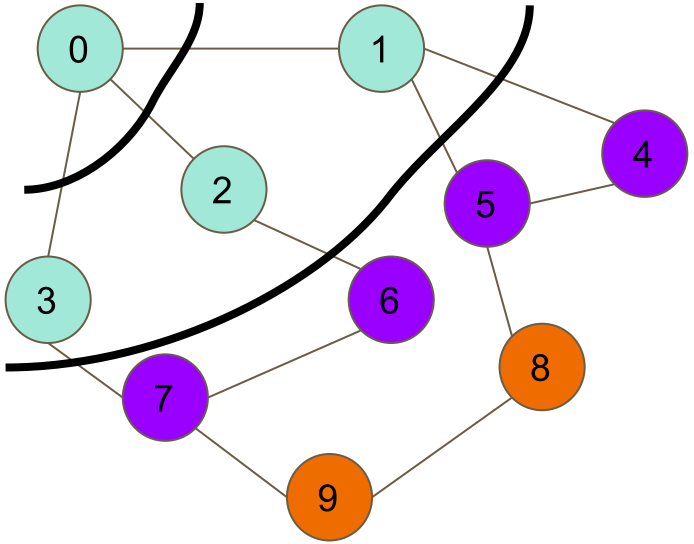
🟡 Current Node

🟣 Visited Node

🟢 Done Node

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



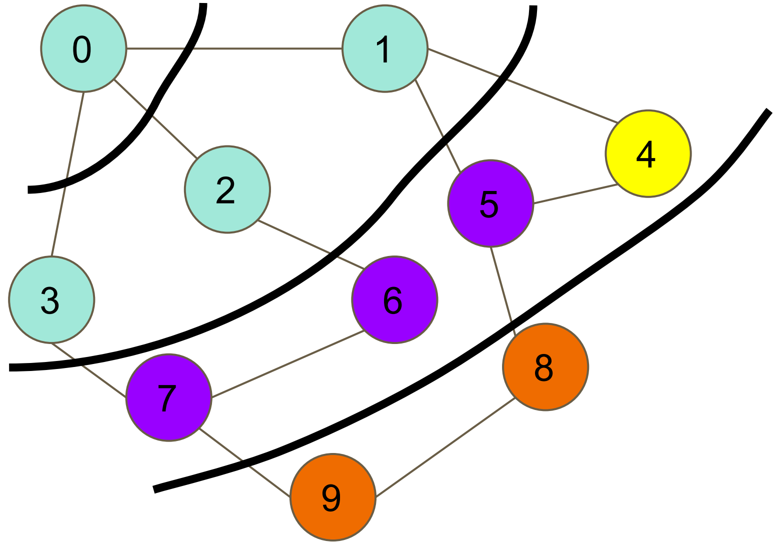
🟡 Current Node

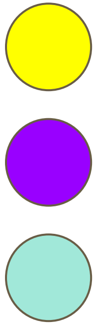
🟣 Visited Node

🟢 Done Node

BFS

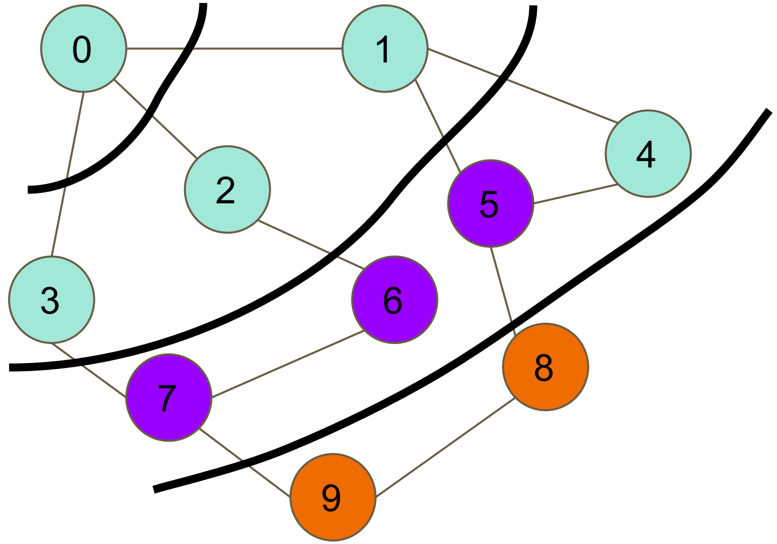
```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```





BFS

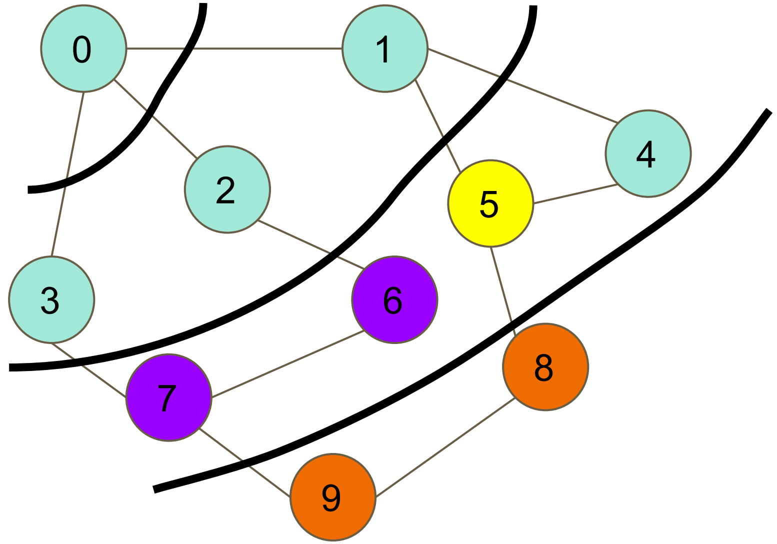
```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



🟡	Current Node
🟣	Visited Node
🟢	Done Node

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



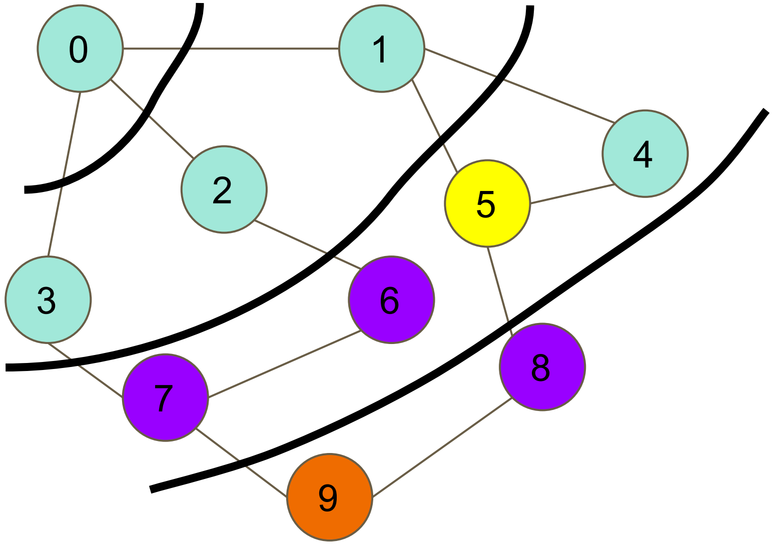
🟡 Current Node

🟣 Visited Node

🟢 Done Node

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



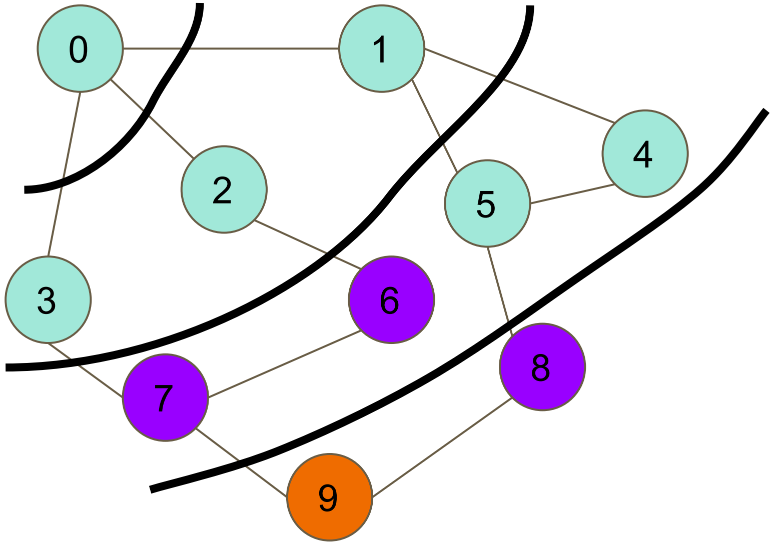
🟨 Current Node

🟪 Visited Node

🟩 Done Node

BFS

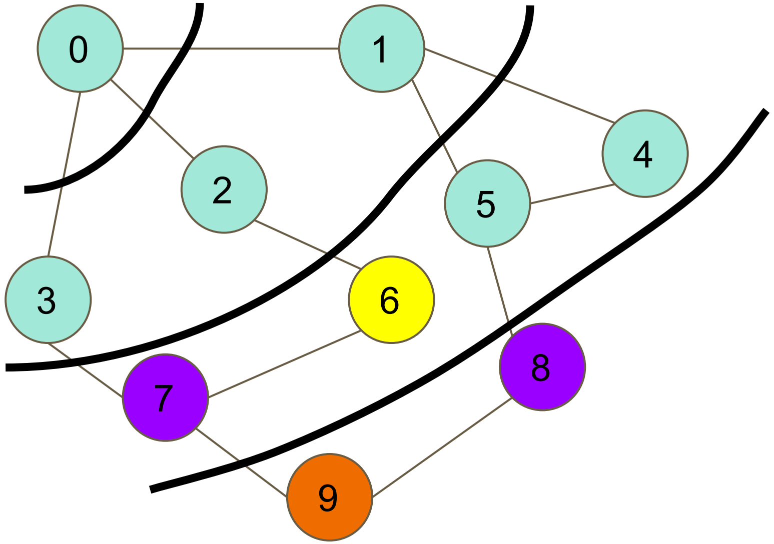
```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



🟡	Current Node
🟣	Visited Node
🟢	Done Node

BFS

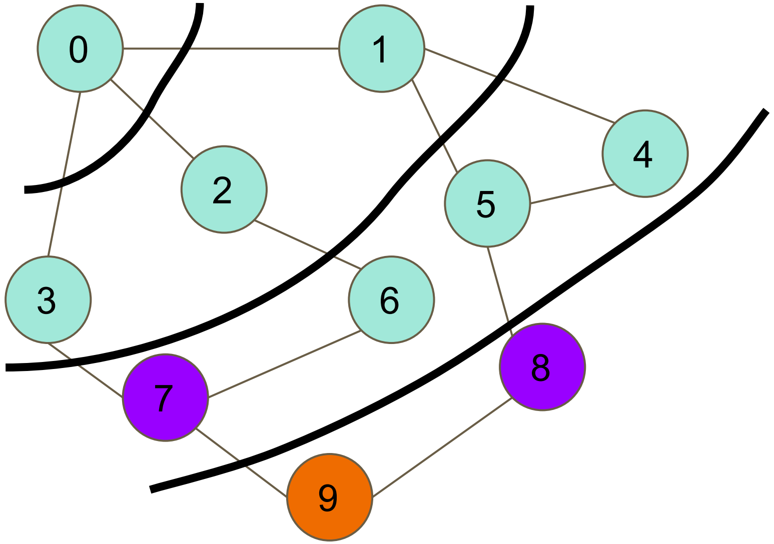
```
bfs(graph G, node start)
    Queue q
    q.push(start)
    mark start as visited
    while not q.empty()
        u = q.pop()
        for v in G.adjacent[u]
            if v not visited
                mark v as visited
                q.push(v)
        mark u as Done
```



🟨	Current Node
🟪	Visited Node
🟩	Done Node

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



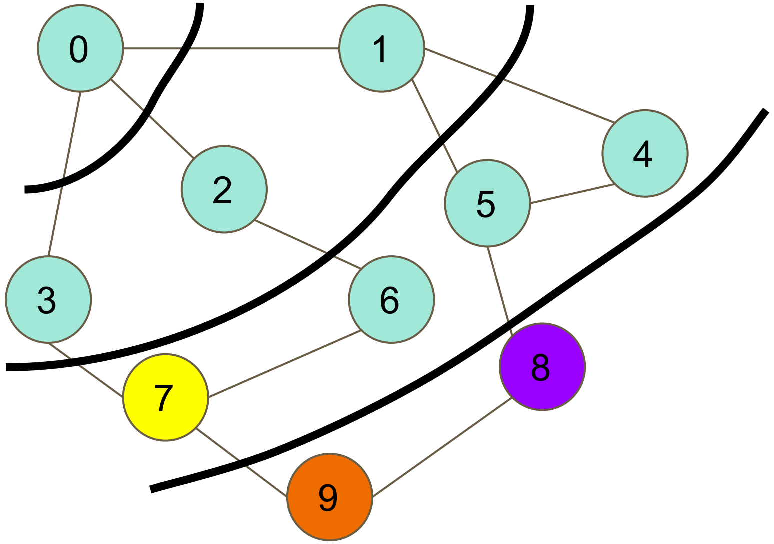
🟨 Current Node

🟪 Visited Node

🟩 Done Node

BFS

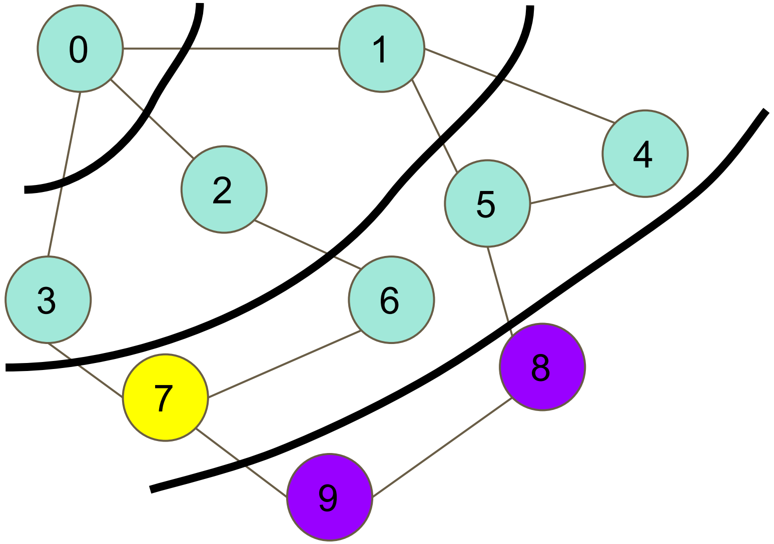
```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```

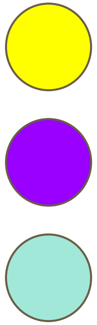


🟨	Current Node
🟪	Visited Node
🟩	Done Node

BFS

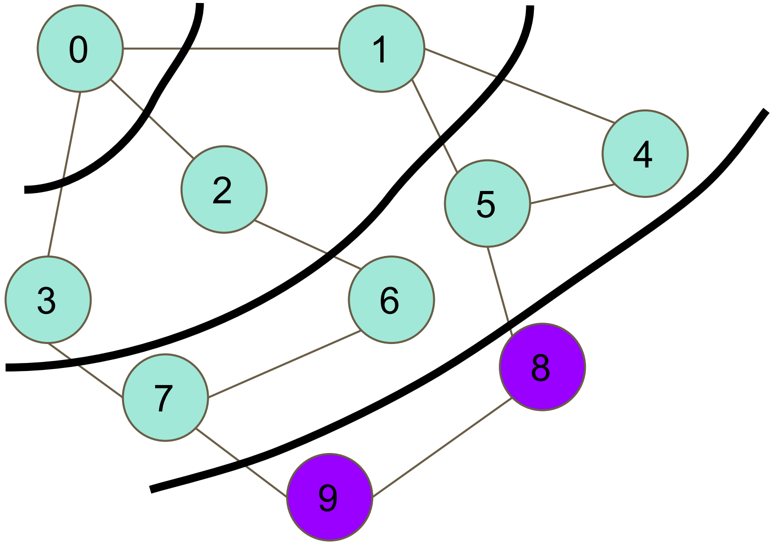
```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```





BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



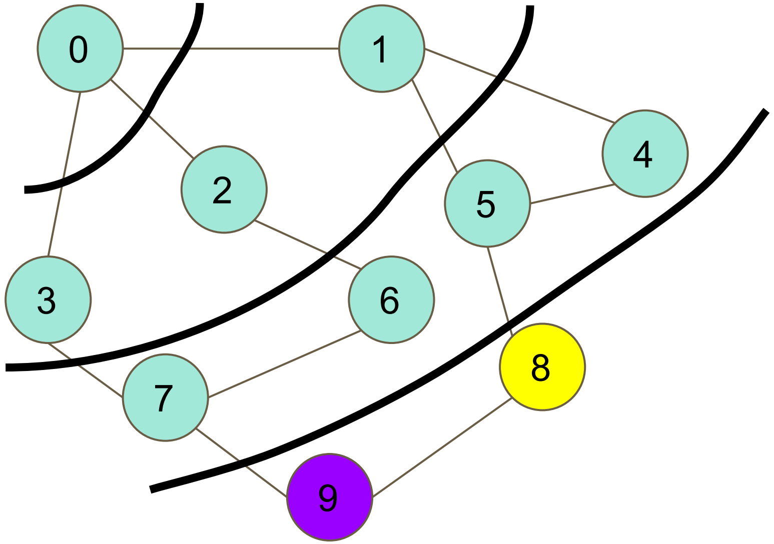
🟡 Current Node

🟣 Visited Node

🟢 Done Node

BFS

```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



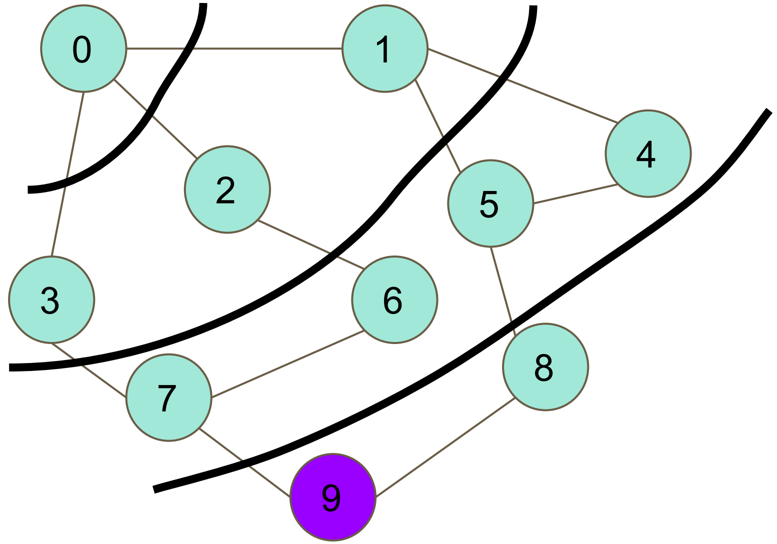
🟨 Current Node

🟪 Visited Node

🟩 Done Node

BFS

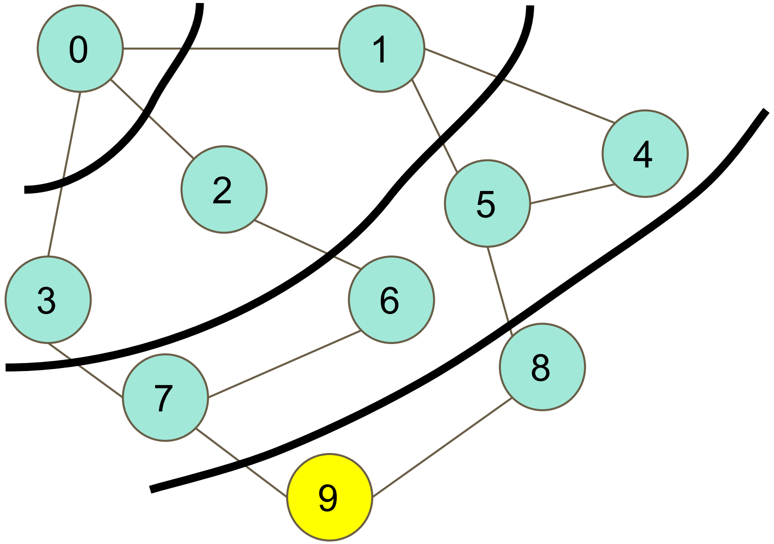
```
bfs(graph G, node start)
    Queue q
    q.push(start)
    mark start as visited
    while not q.empty()
        u = q.pop()
        for v in G.adjacent[u]
            if v not visited
                mark v as visited
                q.push(v)
        mark u as Done
```



🟨	Current Node
🟪	Visited Node
🟩	Done Node

BFS

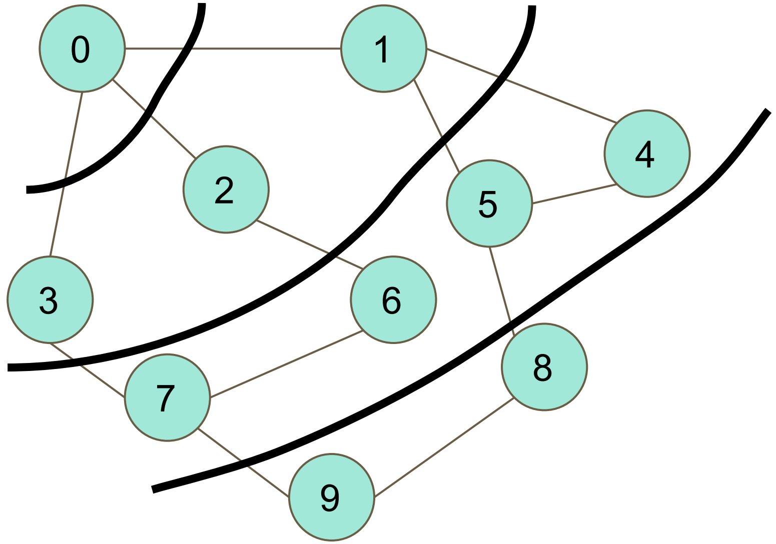
```
bfs(graph G, node start)
     Queue q
     q.push(start)
     mark start as visited
     while not q.empty()
          u = q.pop()
          for v in G.adjacent[u]
               if v not visited
                    mark v as visited
                    q.push(v)
          mark u as Done
```



- 🟡 Current Node
- 🟣 Visited Node
- 🟢 Done Node

BFS

```
bfs(graph G, node start)
    Queue q
    q.push(start)
    mark start as visited
    while not q.empty()
        u = q.pop()
        for v in G.adjacent[u]
            if v not visited
                mark v as visited
                q.push(v)
        mark u as Done
```



🟡	Current Node
🟣	Visited Node
🟢	Done Node

Problema 1

Sea T un grafo conexo no dirigido sin ciclos, se dice que T es un árbol. El diámetro del árbol corresponde al largo del camino simple de largo máximo en el grafo. El problema es calcular el diámetro de un árbol.

Problema 1

Idea: encontrar el camino más largo en T usando BFS.

Problema 1

Notar que como T es un árbol, para cada par de nodos u,v existe un único camino que los une, más aún cada camino en T corresponde a un par de nodos u,v.

Notar además que dado un camino de largo máximo entre dos nodos u,v y un tercer nodo w, se tiene que u está a distancia máxima de w o que v está a distancia máxima de w.

Problema 1

Solución: Correr 2 bfs, uno desde cualquier nodo, y otro desde el último nodo visitado por el bfs anterior. El diámetro del árbol es la distancia entre el último nodo del primer bfs y el último nodo del segundo bfs.

Dijsktra

```
dijkstra(graph G, node start)
     priority_queue q
     q.push((0, start))
     distance[start] = 0 //distance es infty al comienzo
     while not q.empty()
          d,u = q.pop()
          for v,w in G.adjacent[u]
               if distance[v]>d+w
                    distance[v]=d+w
                    q.push((distance[v],v))
```

Problema 2

Hay un juego donde el héroe quiere llegar al último nivel con el menor costo, para ganar cada nivel i el héroe debe gastar una cantidad de energía E_i, y en algunos niveles hay unas tiendas, cada tienda t fija la energía de nuestro héroe a E_t por un costo C_t. El héroe quiere tu ayuda para lograr esto.

Problema 2

Idea: Usar Dijsktra para ayudar a nuestro héroe.

Problema 2

Notemos que si ponemos que cada nivel es un nodo tenemos un grafo con pesos, donde cada arista corresponde a una tupla (u,v,w) donde en el nivel u hay una tienda con costo w tal que fija suficiente energía para llegar a v.

Problema 2

Correr Dijsktra en el grafo descrito anteriormente comenzando en el nivel “0” para llegar al último nivel.

Prim

```
prim(Grafo G)
    set Q //de nodos ordenados por C[u]
    //C[u] es infty en un comienzo
    forest F
    while not Q.empty()
        u = min(Q)
        F.add(u)
        if E[u]≠null
            F.add(E[u])
        for w,v in G.adjacent[u]
            C[v] = w
            E[v] = (u,v)
```

Kruskal

```
kruskal(Grafo G)
     UnionFind T
     E = G.edges
     sort(E) //Por el peso
     for w,u,v in E
          if not T.same_set(u,v)
               T.join(u,v)
               //añadir arista u,v
```

Problema 3

Dado uno matriz A de 1 y 0 donde se perdieron algunos de sus valores, pero se tiene el checksum (xor) de cada columna y cada fila, y recuperar cada valor tiene un costo (representado con un matriz B), ¿cúal es el costo mínimo para recuperar todos los valores?

E.g. A = [[1,0],[-1,0]] y B = [[10,10],[10,10]]

Problema 3

Veamos que si falta exactamente 1 valor en alguna columna o fila se puede recuperar sin costo usando el checksum.

Podemos representar la matriz como un grafo bipartito, donde cada entrada A_ij=-1 de la matriz corresponde a una arista entre la fila i y la columna j con peso B_ij.

Problema 3

Vemos que si hay algún nodo desconectado del resto, entonces la columna/fila está completa.

Vemos que si el grafo no tiene ciclos, entonces todo se puede recuperar sin costo.

Problema 3

Entonces si tenemos un grafo con ciclos queremos el subconjunto de aristas de peso mínimo que rompa todos los ciclos.

Pero podemos ver el dual de este problema, encontrar el árbol de peso máximo.

Problema 3

Solución: usar Prim (o Kruskal) para encontrar el árbol de peso máximo del grafo bipartito, y la respuesta corresponde a la suma de los pesos de las aristas que no están en el árbol.

Fin!

Nicholas Mc-Donnell - namcdonnell@uc.cl   Ignacio Porte - ignacio.porte@uc.cl