CS2040S
Data Structures and Algorithms

BFS, DFS, and Directed Graphs!

Roadmap

Last time: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)
- Searching graphs: BFS

What is a graph?

Graph $G = \langle V, E \rangle$

- V is a set of nodes
  - At least one: $|V| > 0$.
- E is a set of edges:
  - $E \subseteq \{ (v,w) : (v \in V), (w \in V) \}$
  - $e = (v,w)$
  - For all $e_1, e_2 \in E : e_1 \neq e_2$

2 x 2 x 2 Rubik's Cube

Configuration Graph

- Vertex for each possible state
- Edge for each basic move
  - 90 degree turn
  - 180 degree turn

Puzzle: given initial state, find a path to the solved state.

Trade-offs

Adjacency Matrix:

- Fast query: are v and w neighbors?
- Slow query: find me any neighbor of v.
- Slow query: enumerate all neighbors.

Adjacency List:

- Fast query: find me any neighbor.
- Fast query: enumerate all neighbors.
- Slower query: are v and w neighbors?

Searching a Graph

Goal:

- Start at some vertex **s** = start.
- Find some other vertex **f** = finish.

  Or: visit **all** the nodes in the graph;

Two basic techniques:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

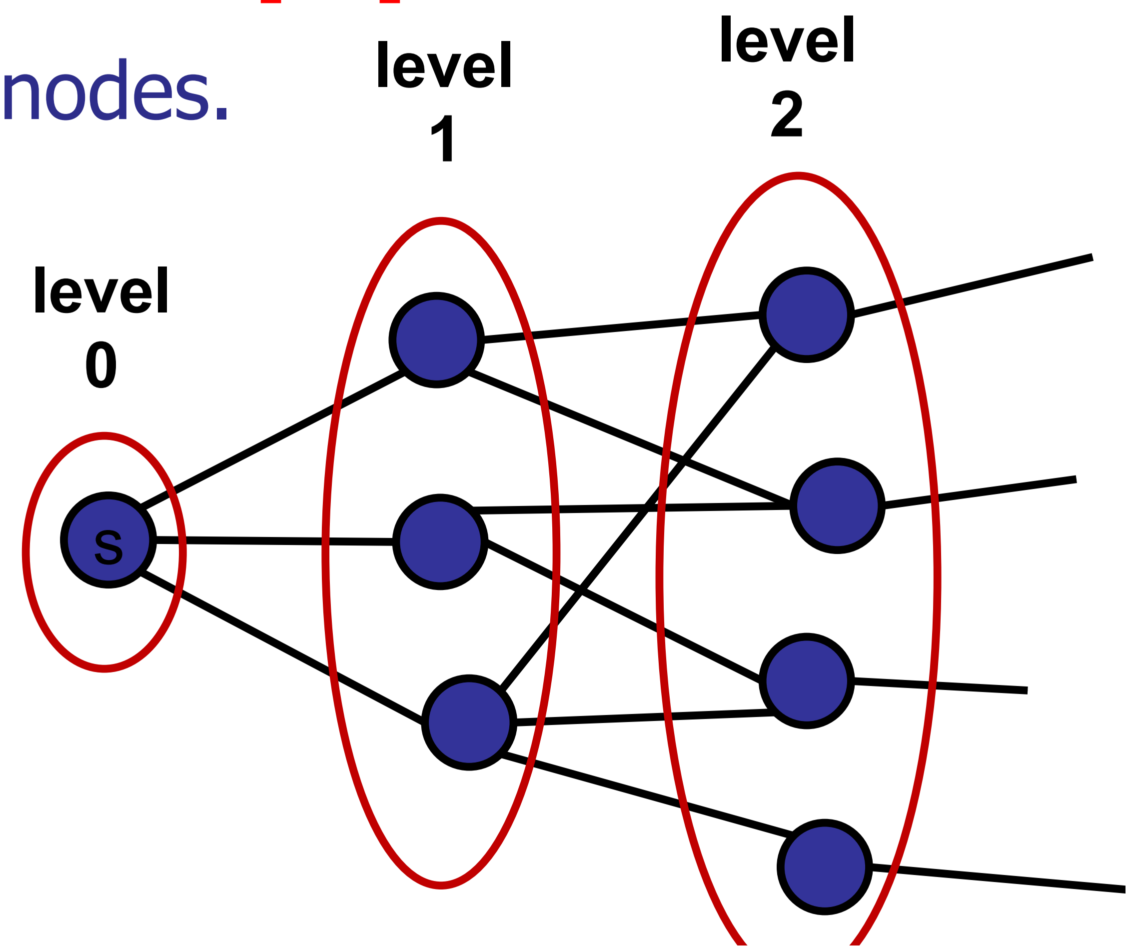
Graph representation:

- Adjacency list

Searching a graph

Breadth-First Search:

- Explore graph level by level.
- Calculate level[i] from level[i-1]
- Skip already visited nodes.

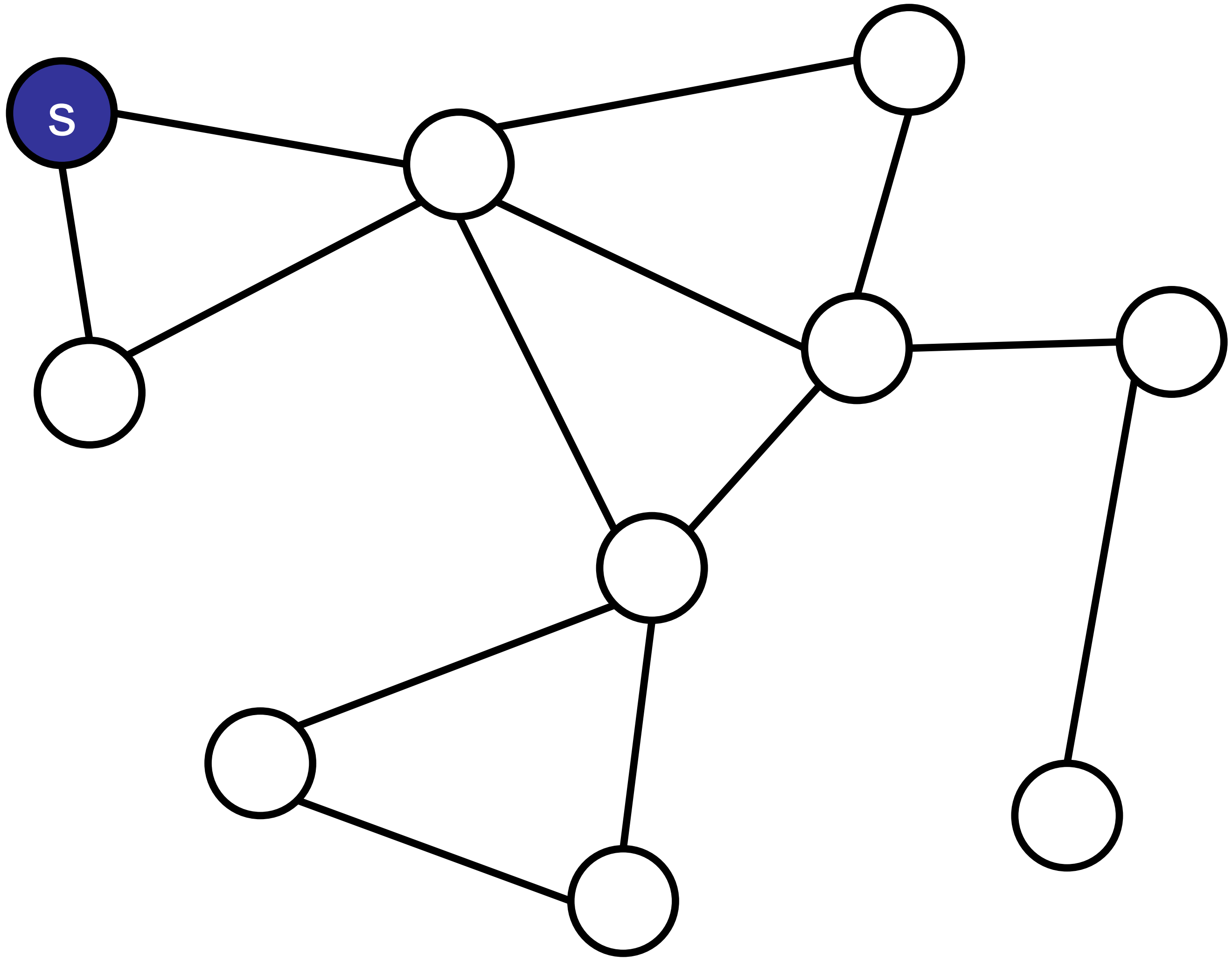


Searching a graph

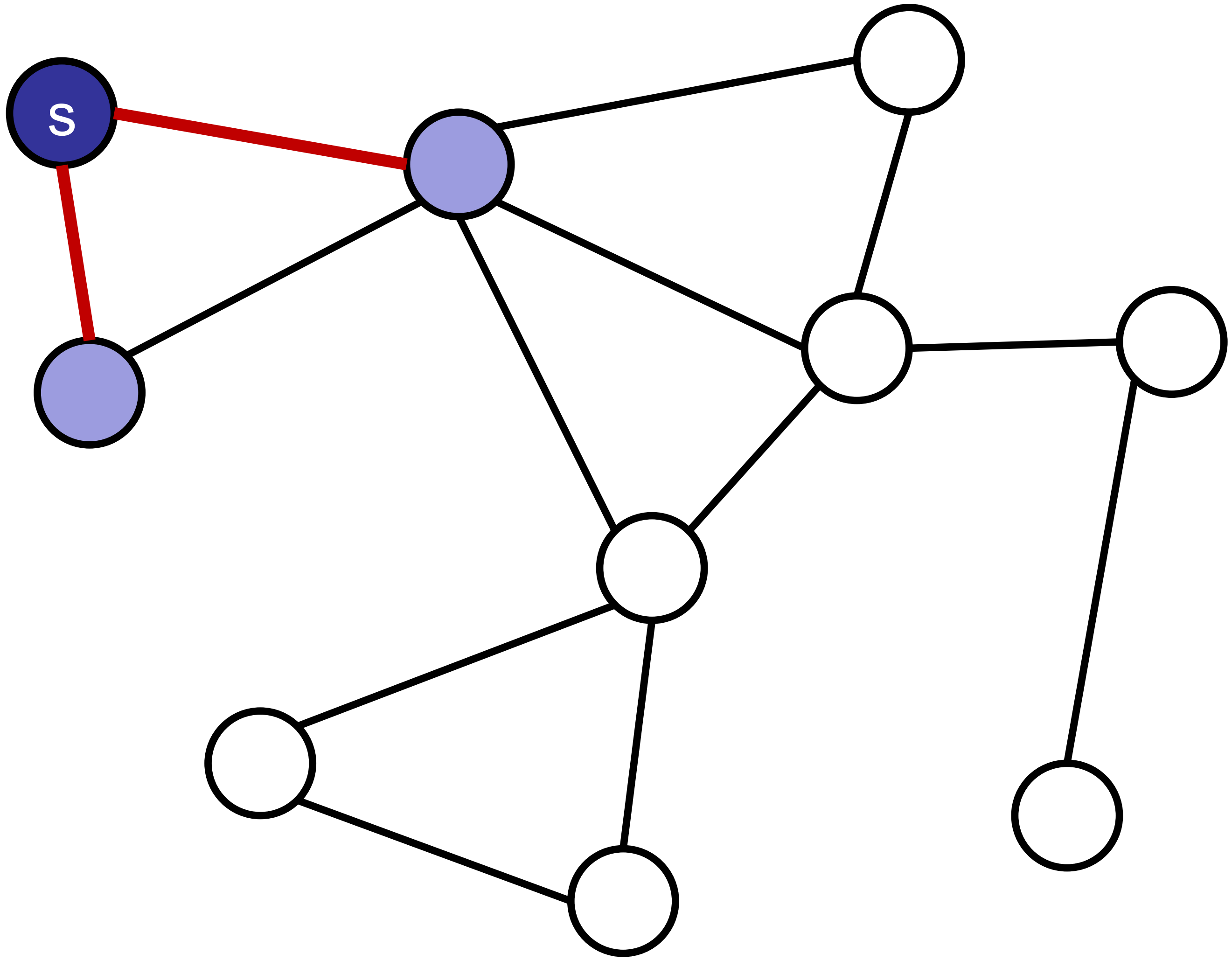
Breadth-First Search:

```
frontier = {s}
while frontier is not empty:
    next-frontier = {}
    for each node u in the frontier:
       for each edge (u,v) in the graph:
           if v is not marked visited, add v to next-frontier
           mark v as visited.
    frontier = next-frontier
```

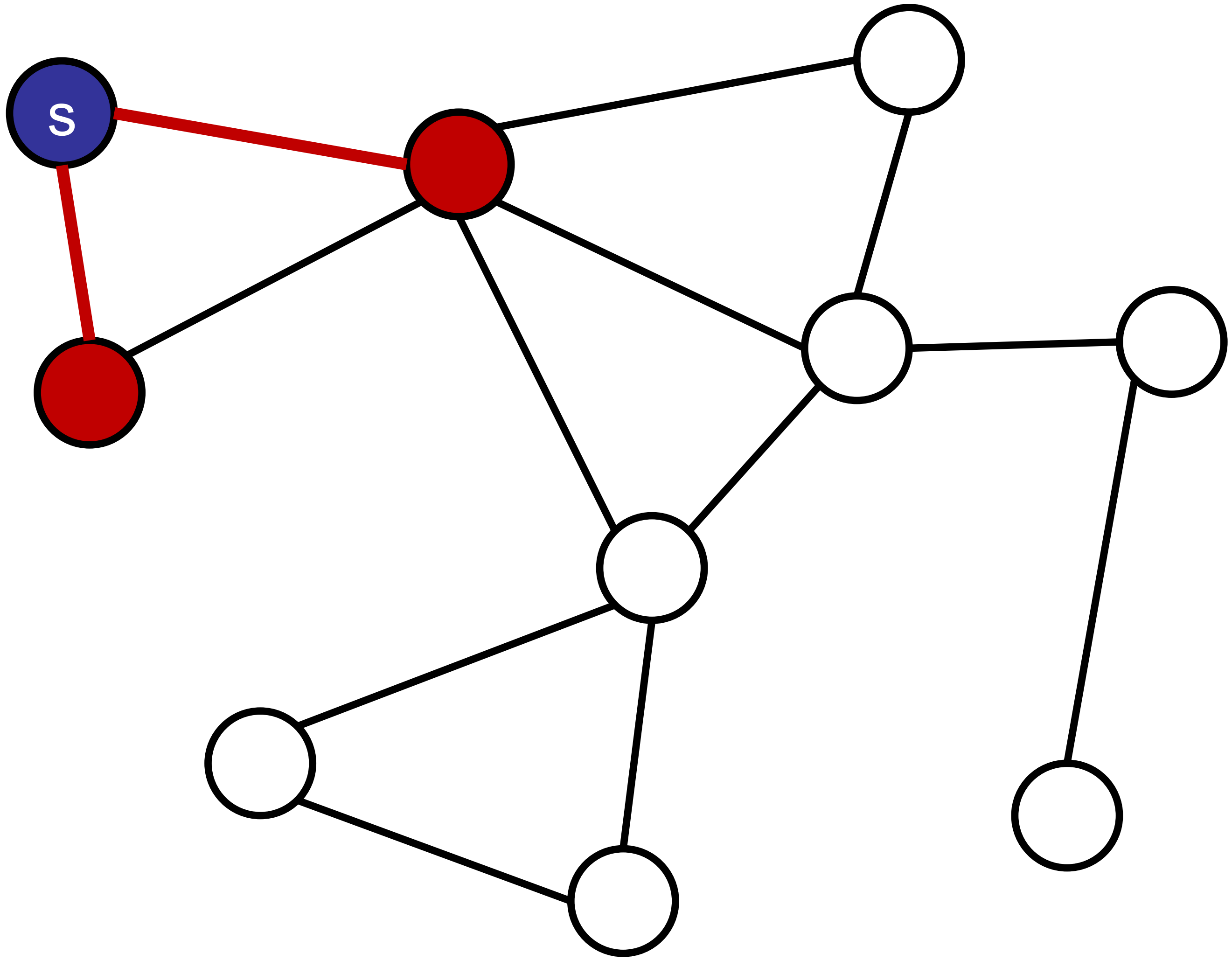
Breadth First Search



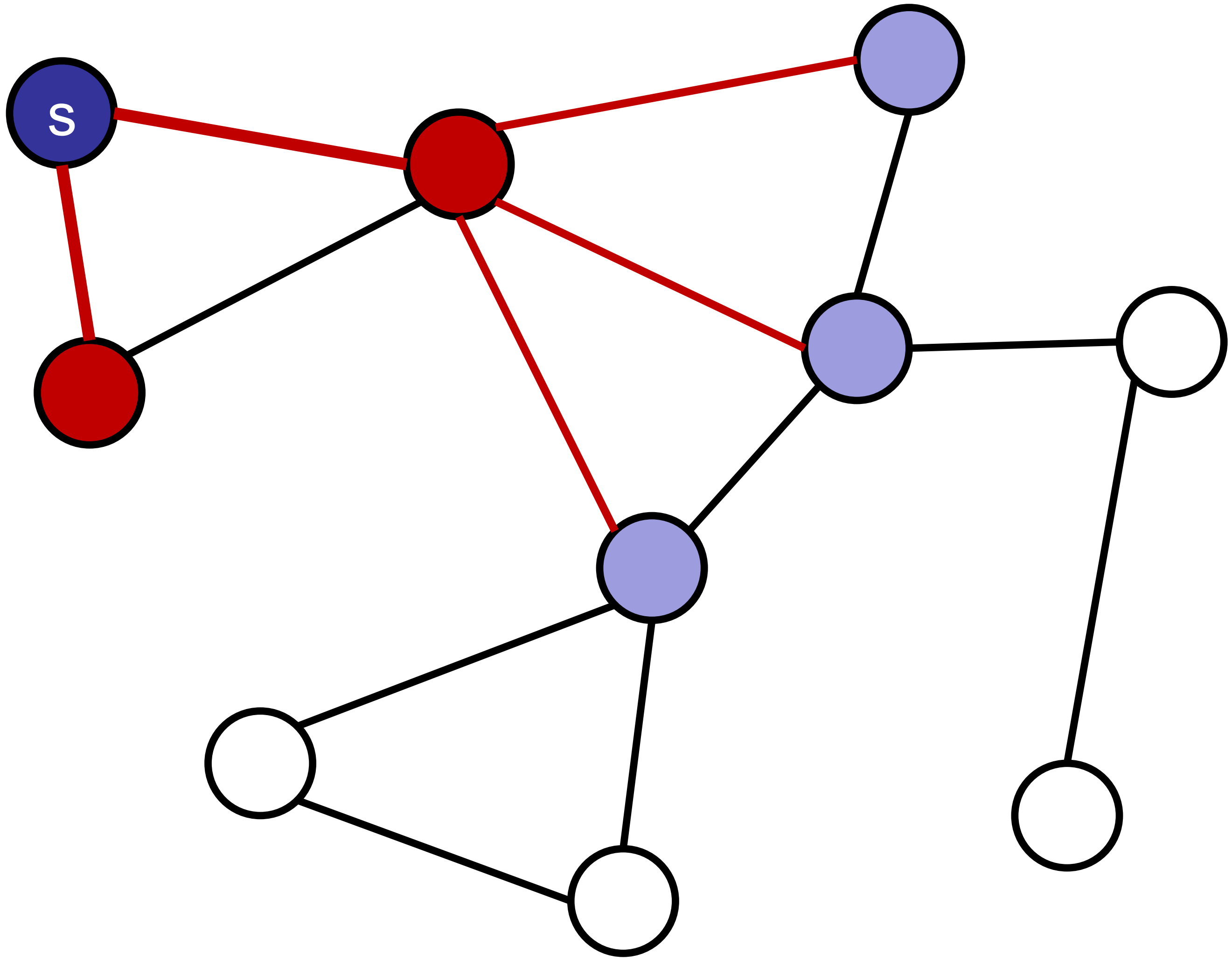
Breadth First Search



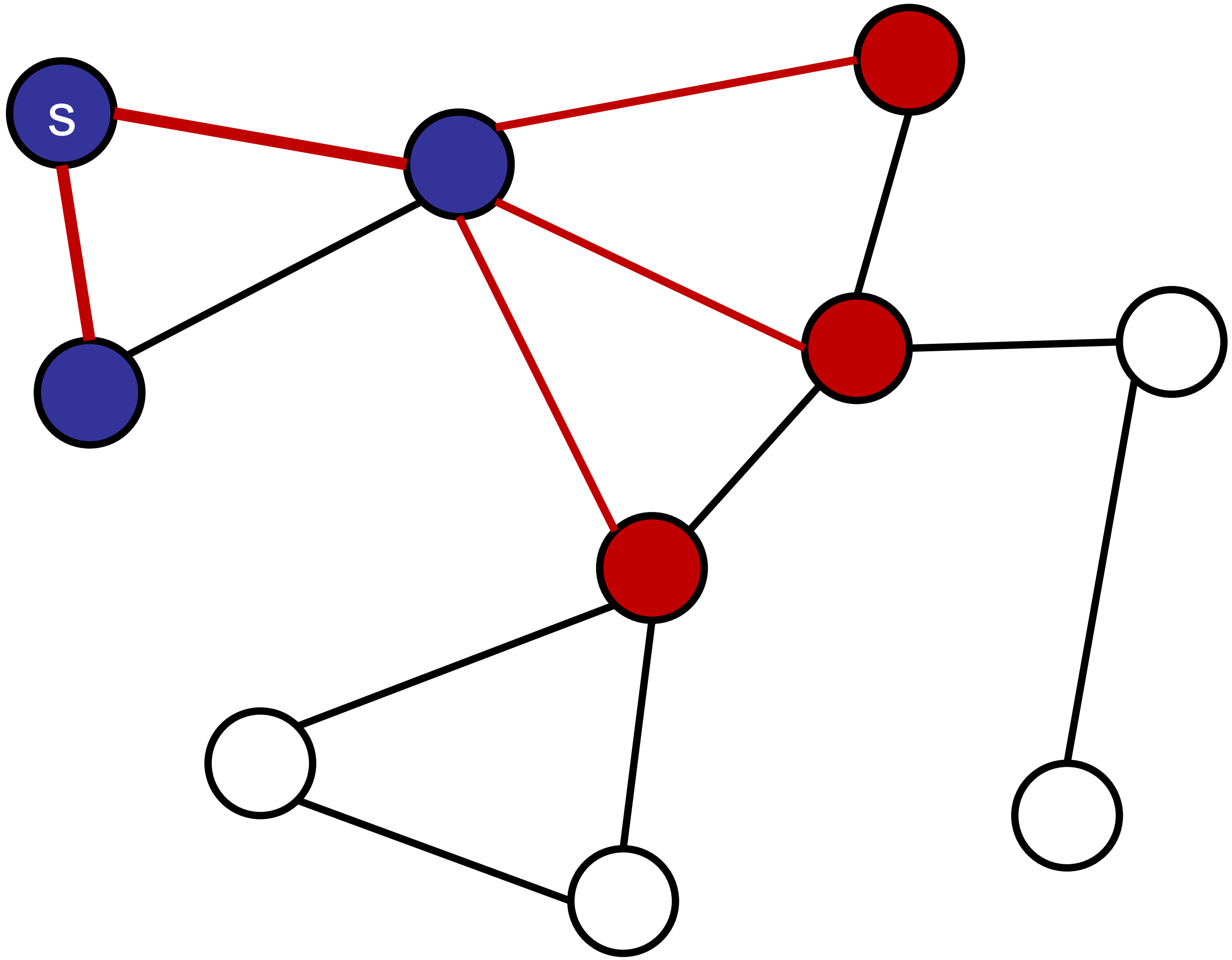
Breadth First Search



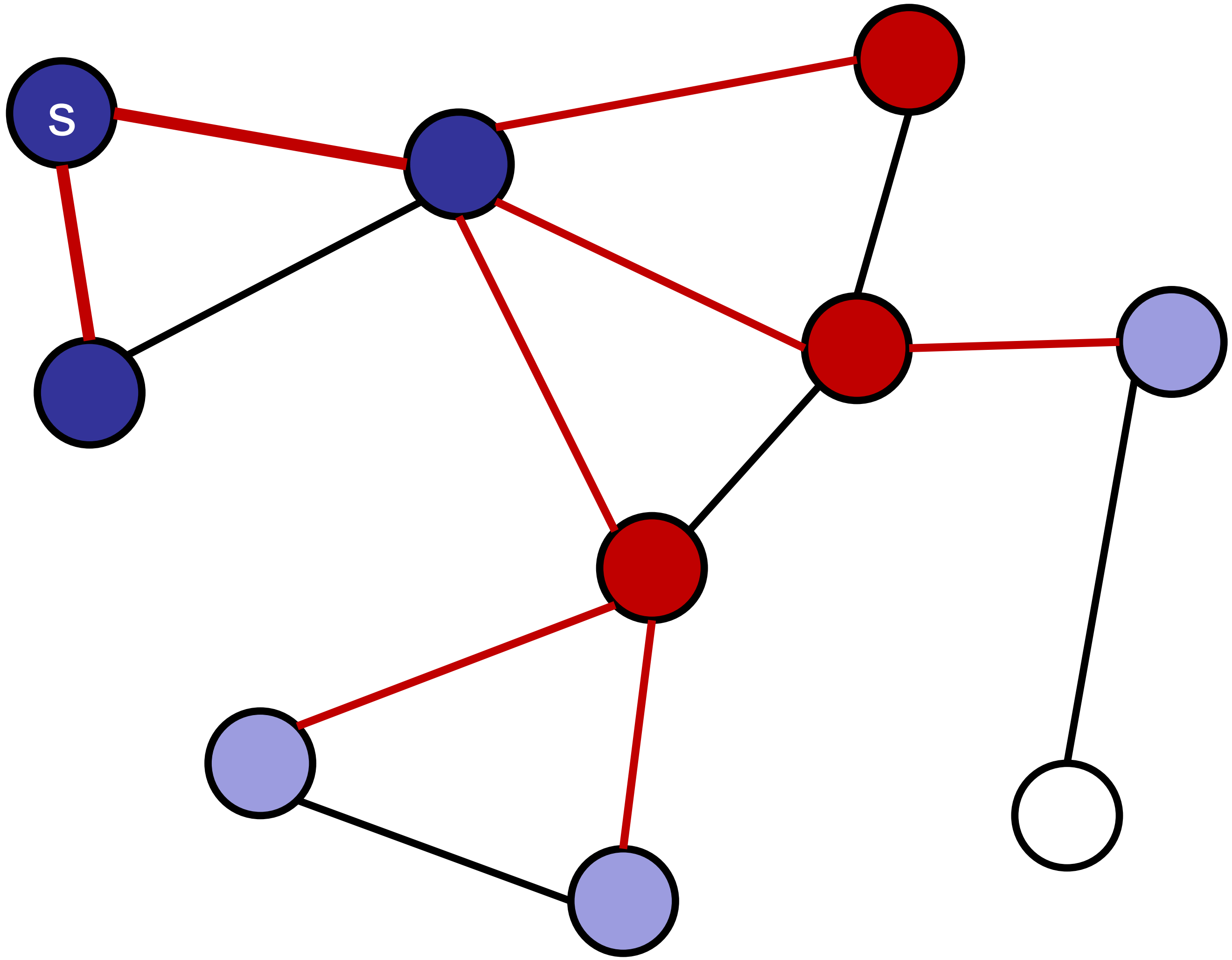
Breadth First Search



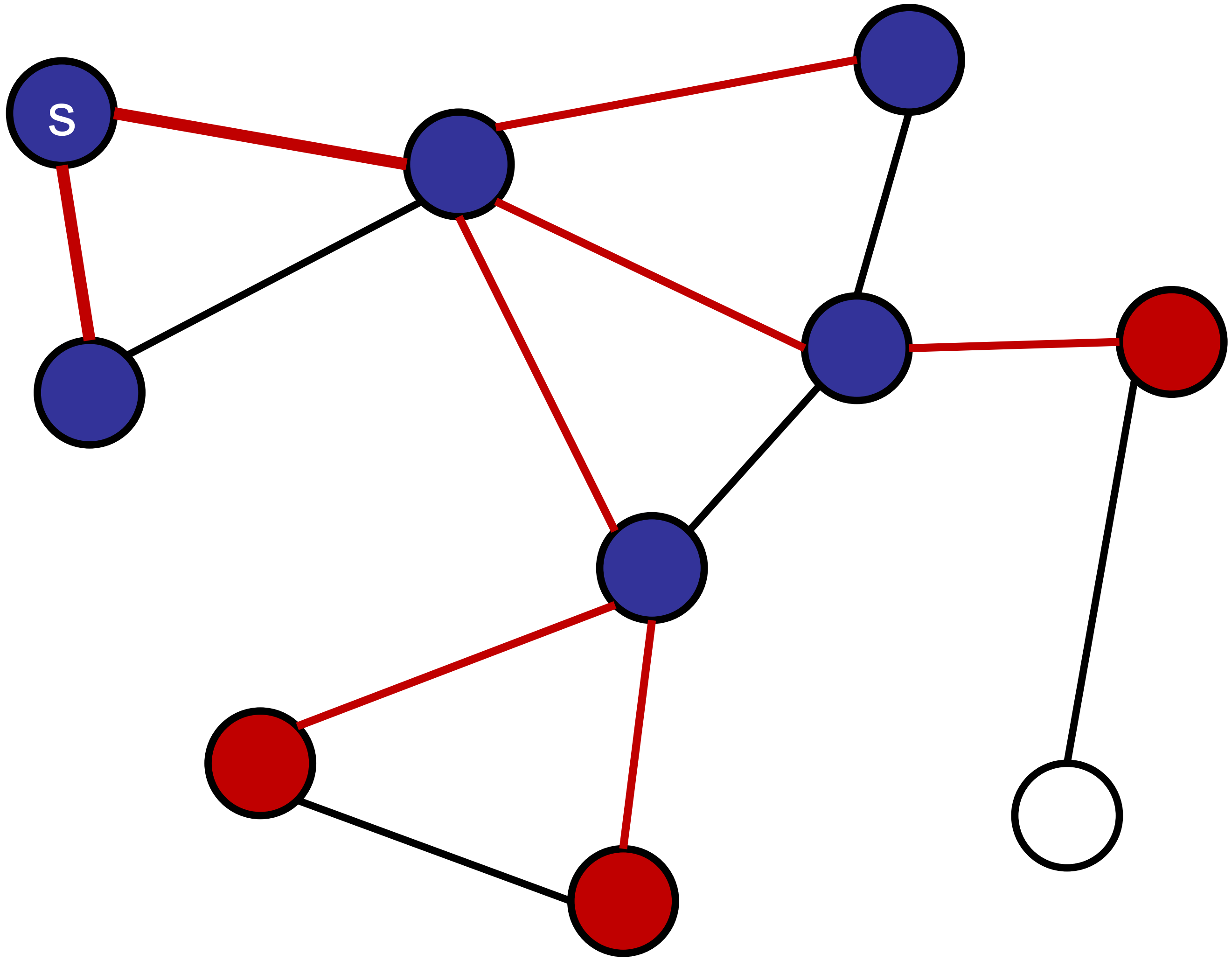
Breadth First Search



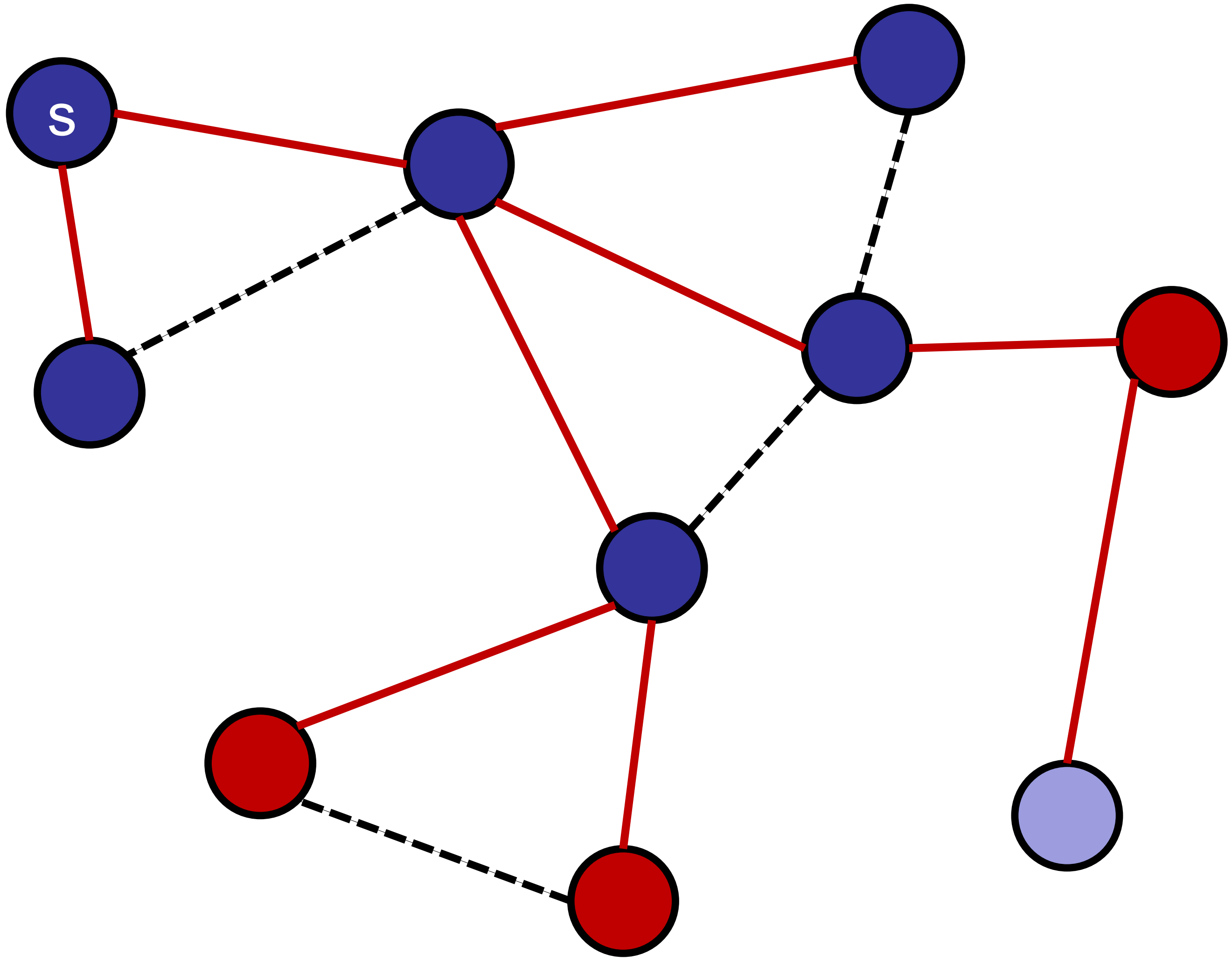
Breadth First Search



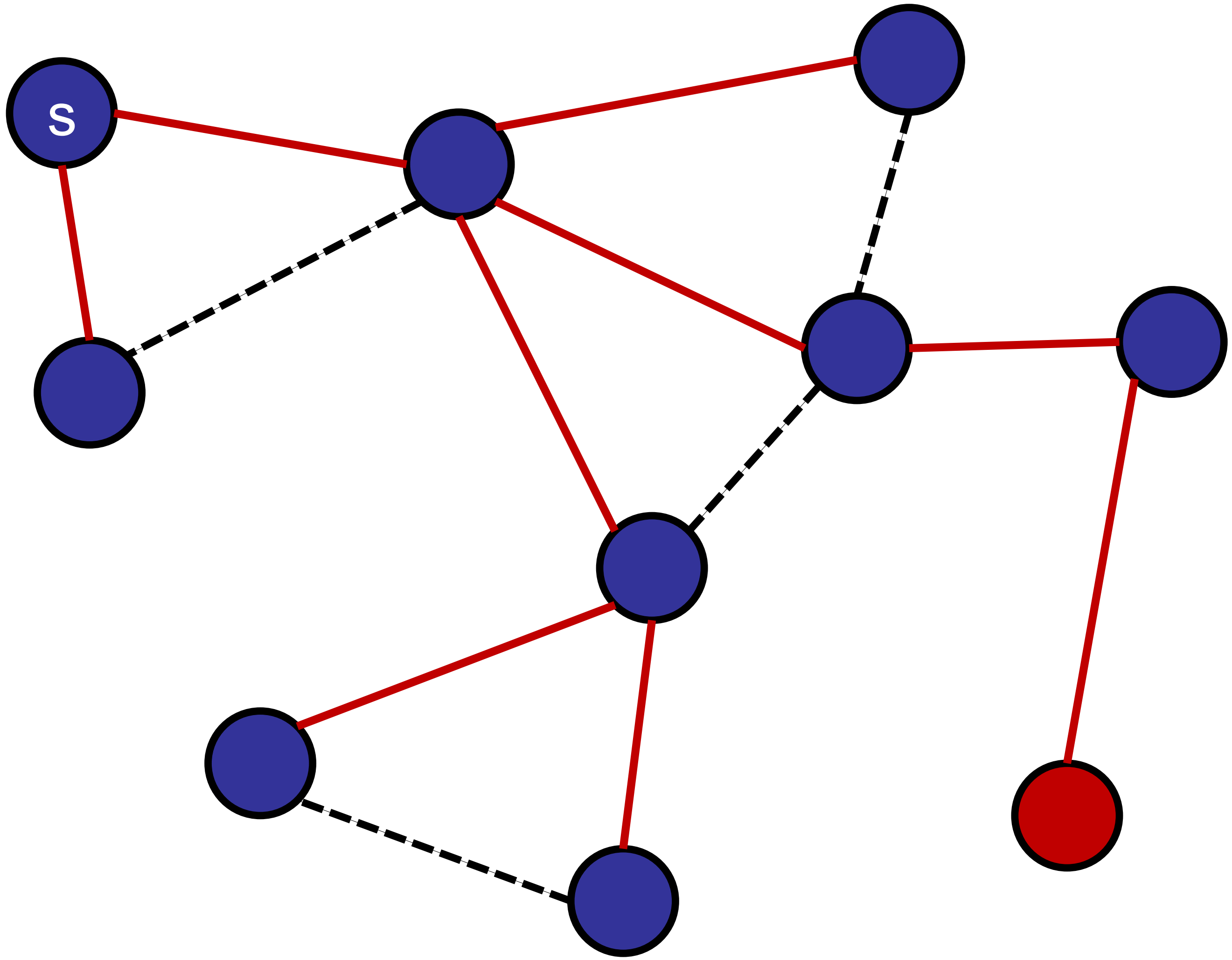
Breadth First Search



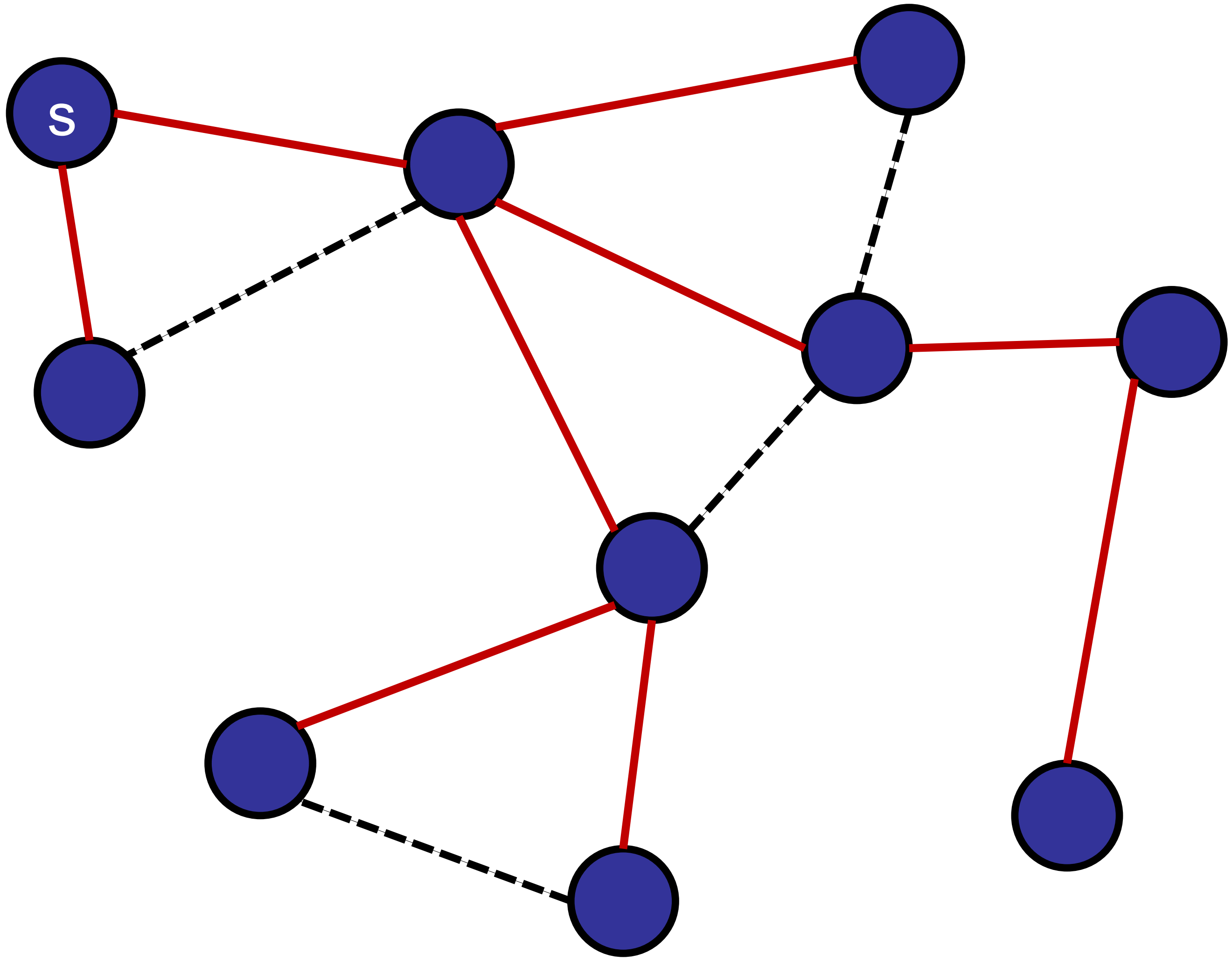
Breadth First Search



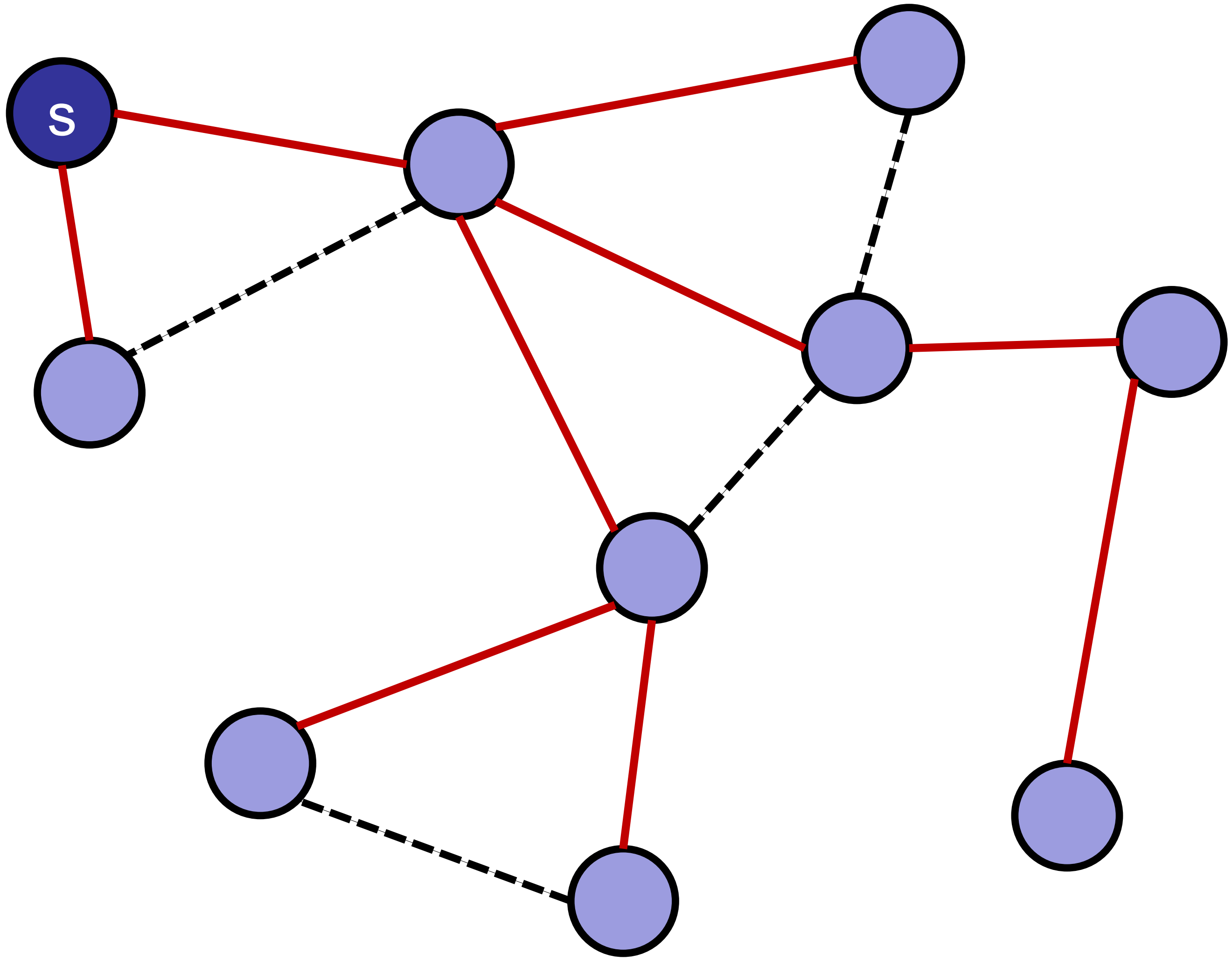
Breadth First Search



Breadth First Search



Breadth First Search

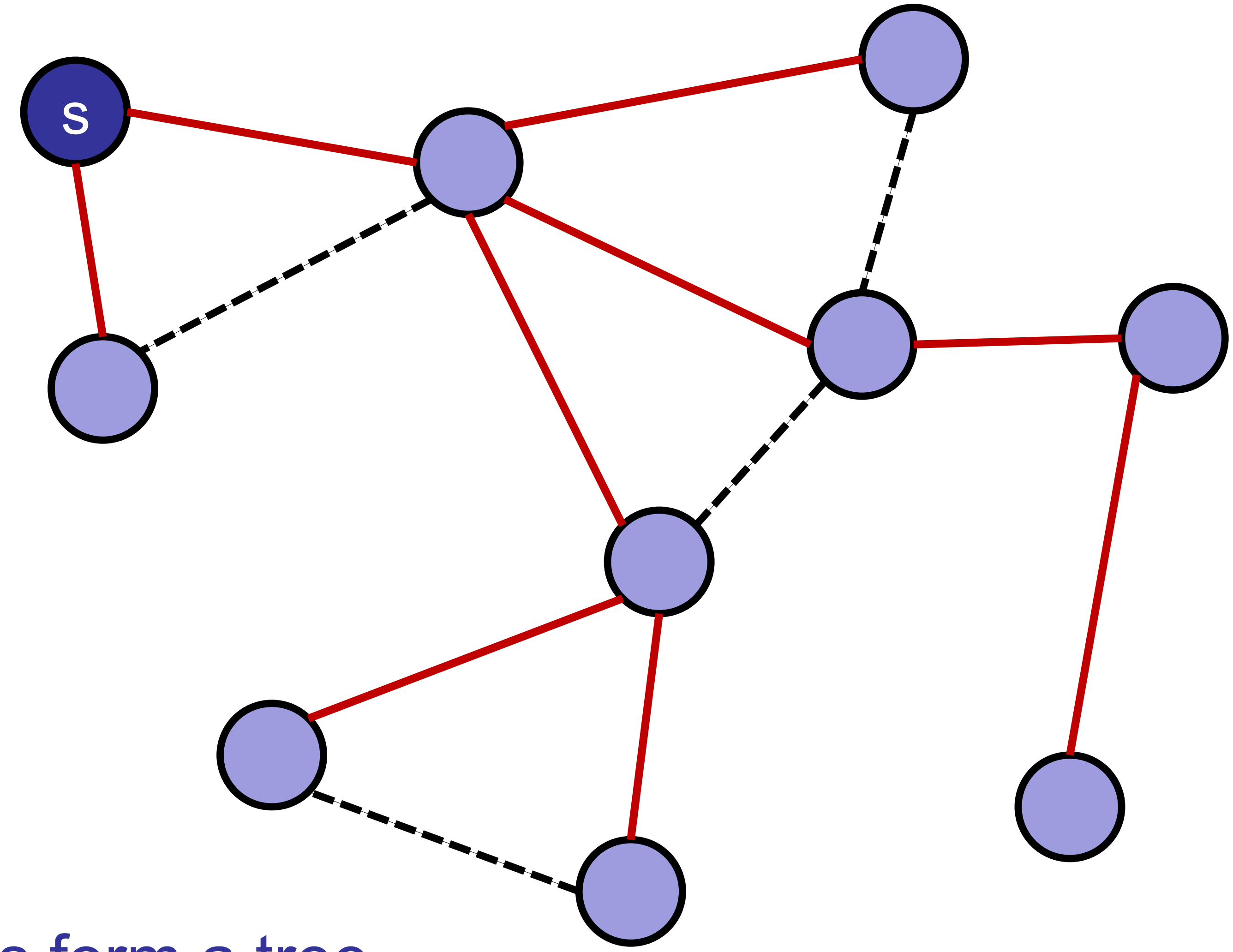


ARCHIPELAGO

is open

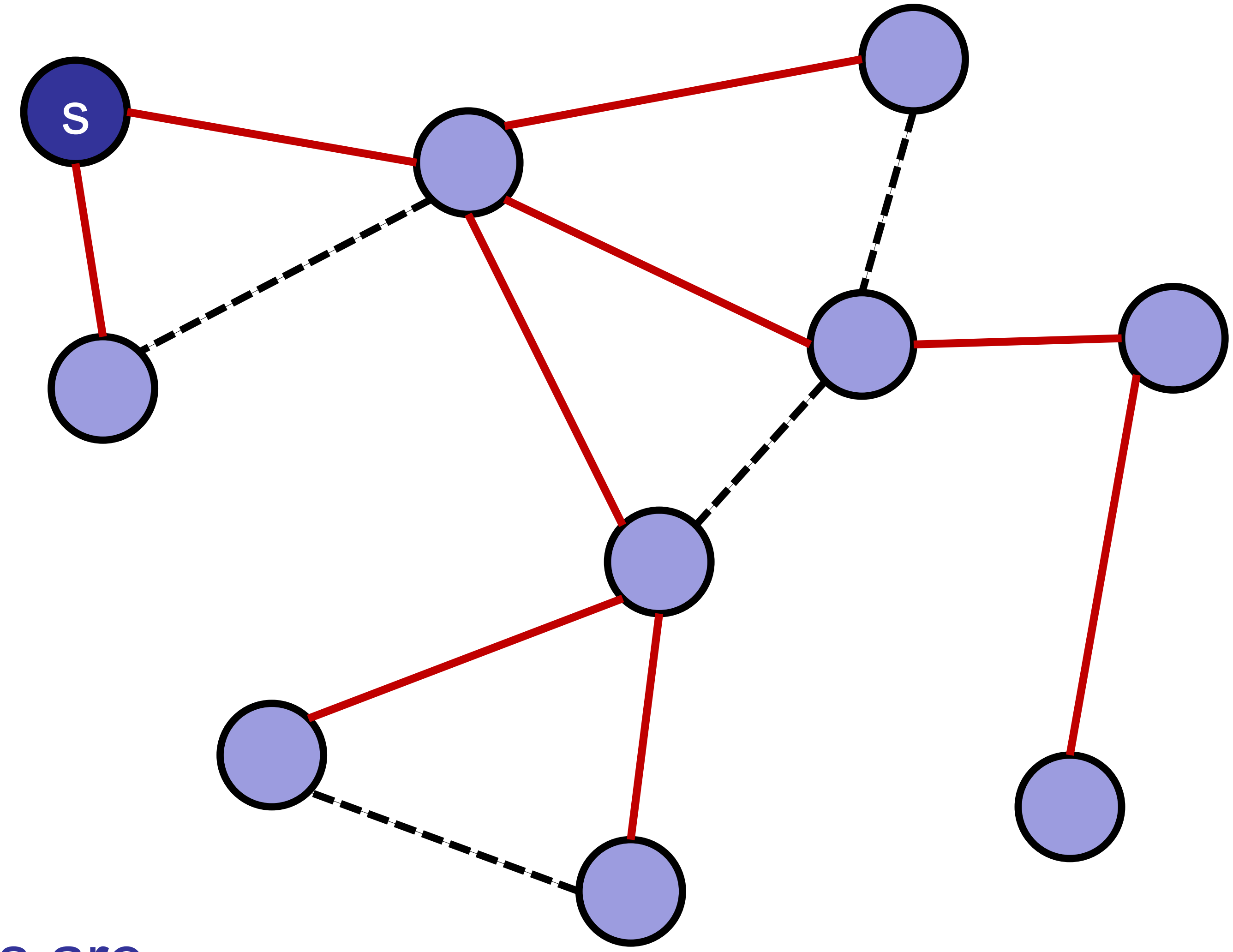
What are the properties of the parent edges?

Breadth First Search



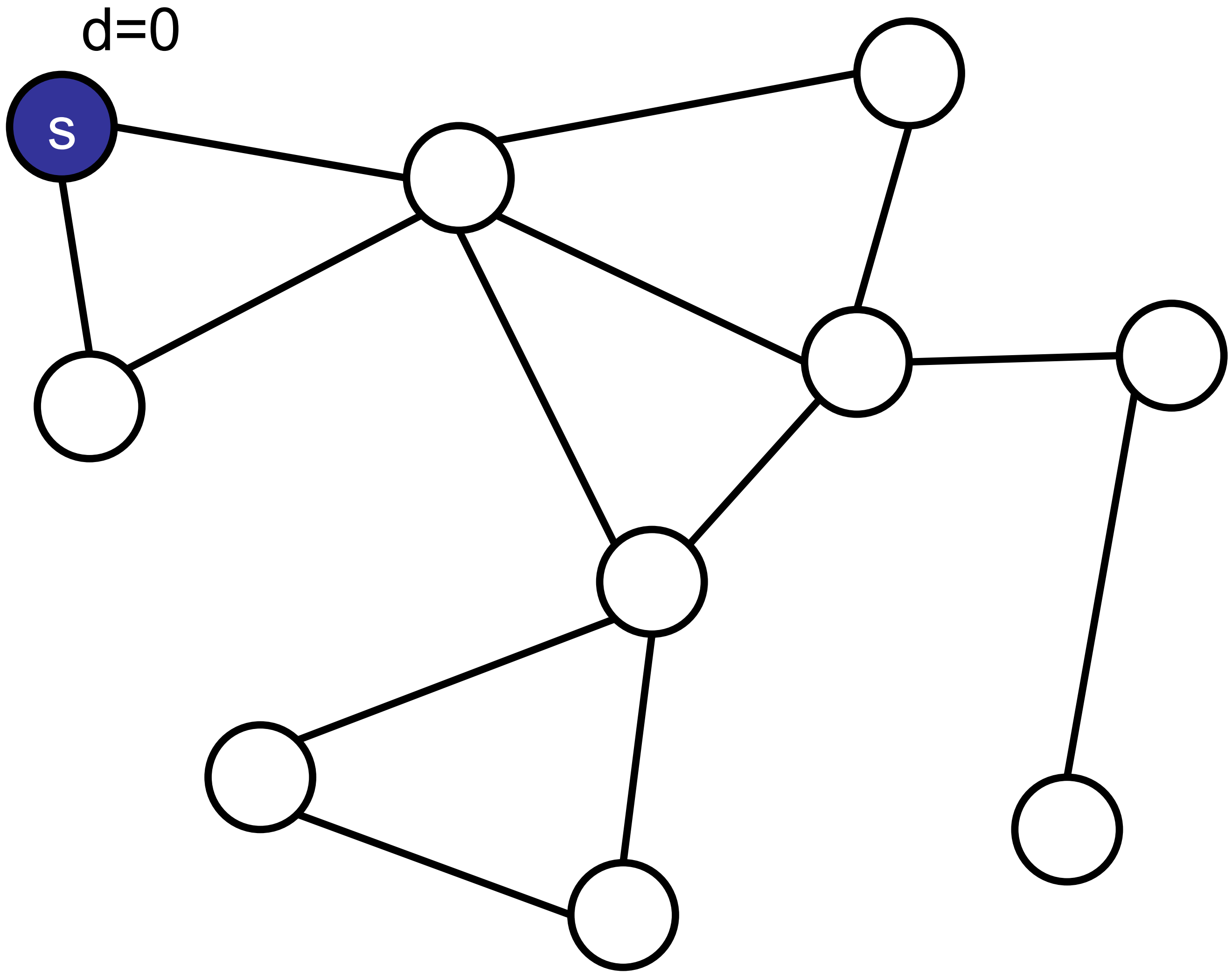
1. Parent edges form a tree
   (No cycles.)

Breadth First Search

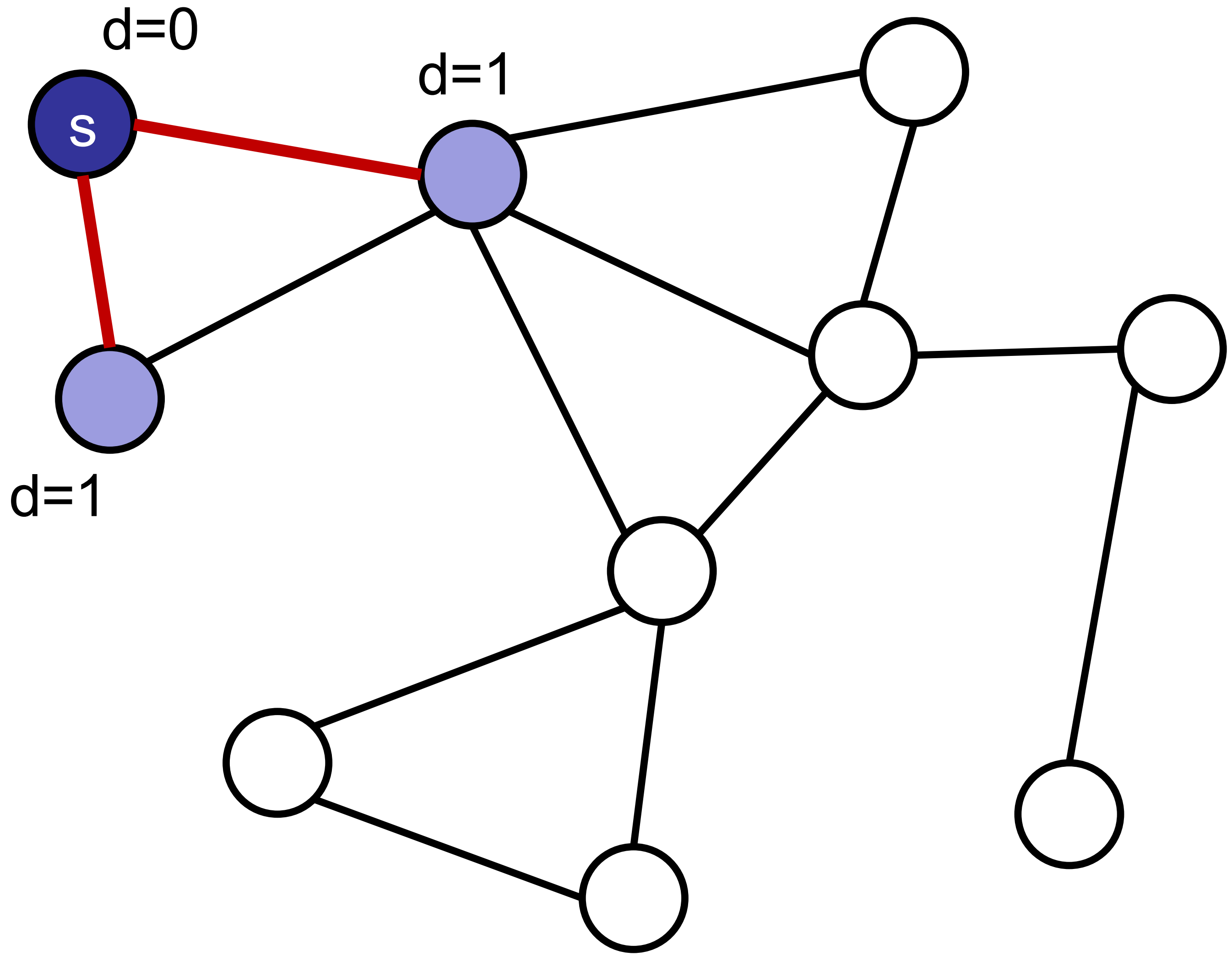


2. Parent edges are shortest paths from s.

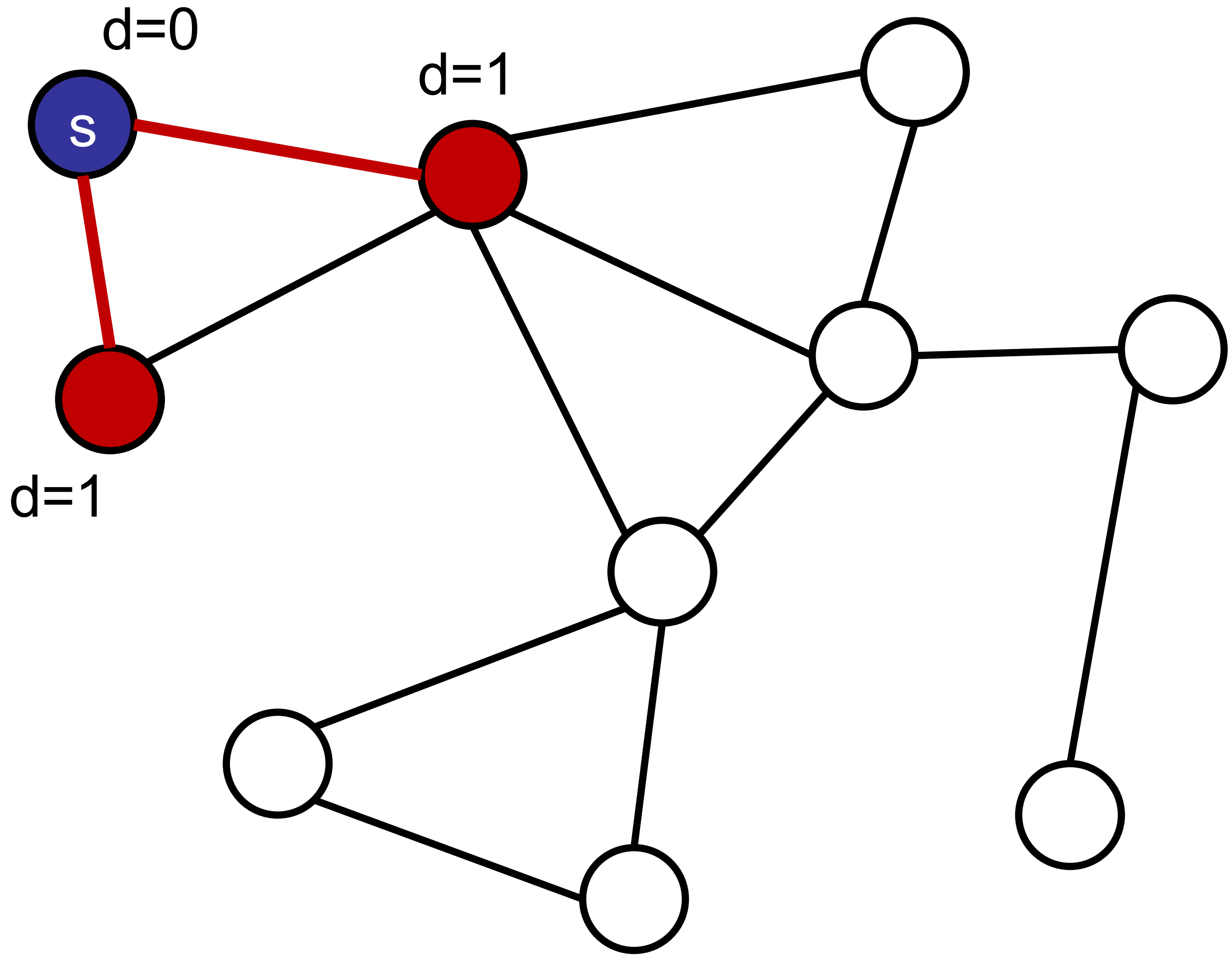
Breadth First Search



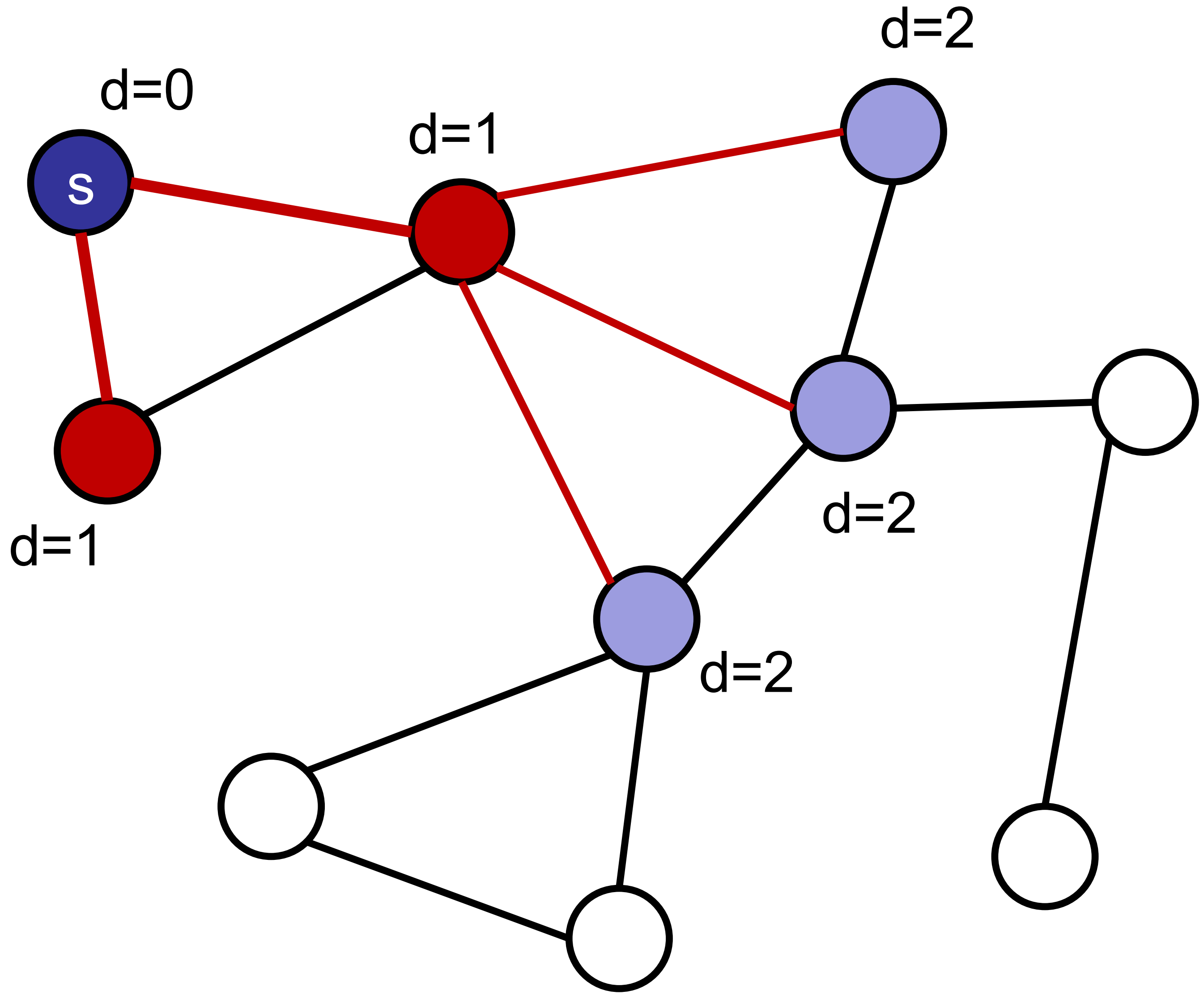
Breadth First Search



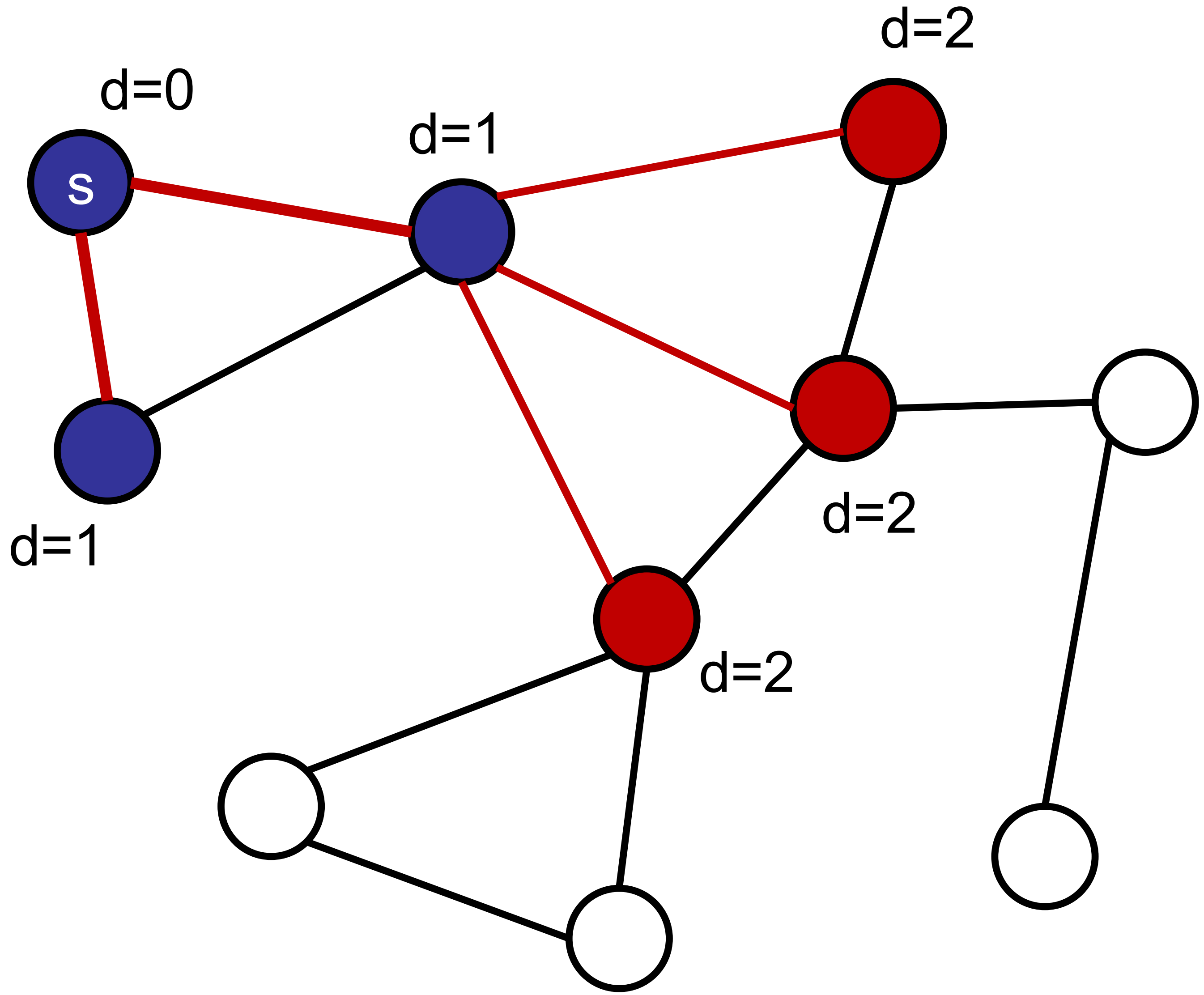
Breadth First Search



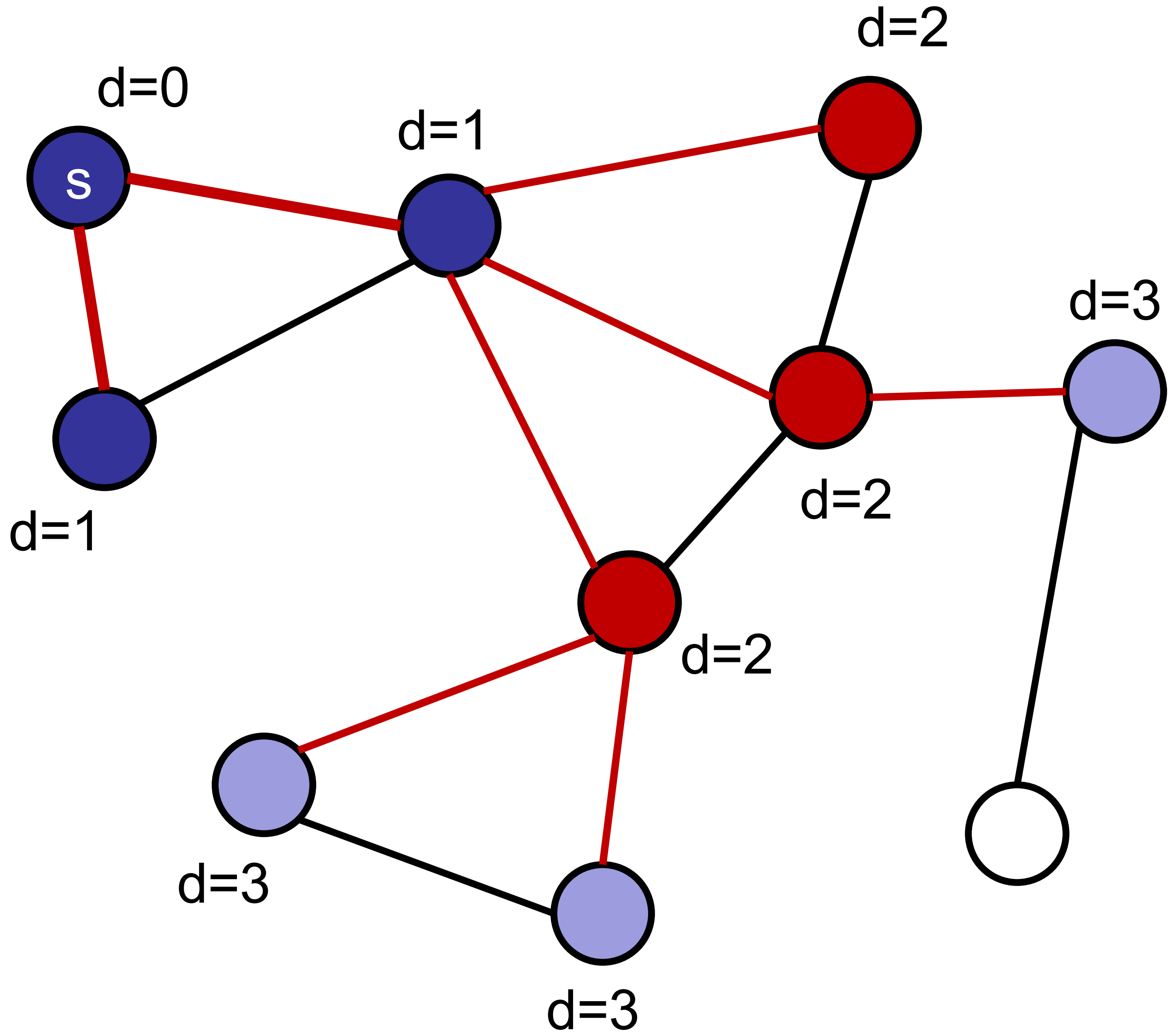
Breadth First Search



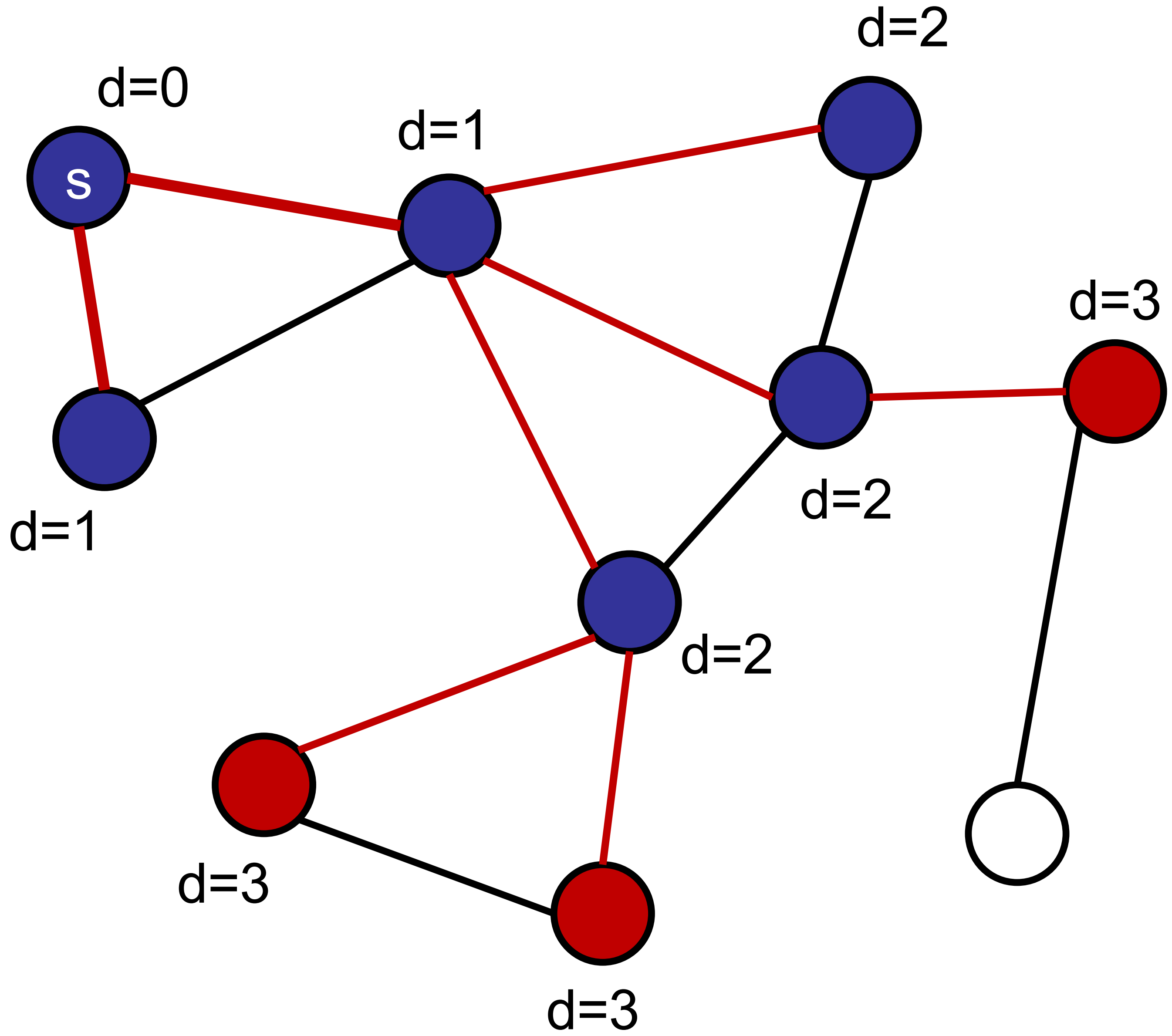
Breadth First Search



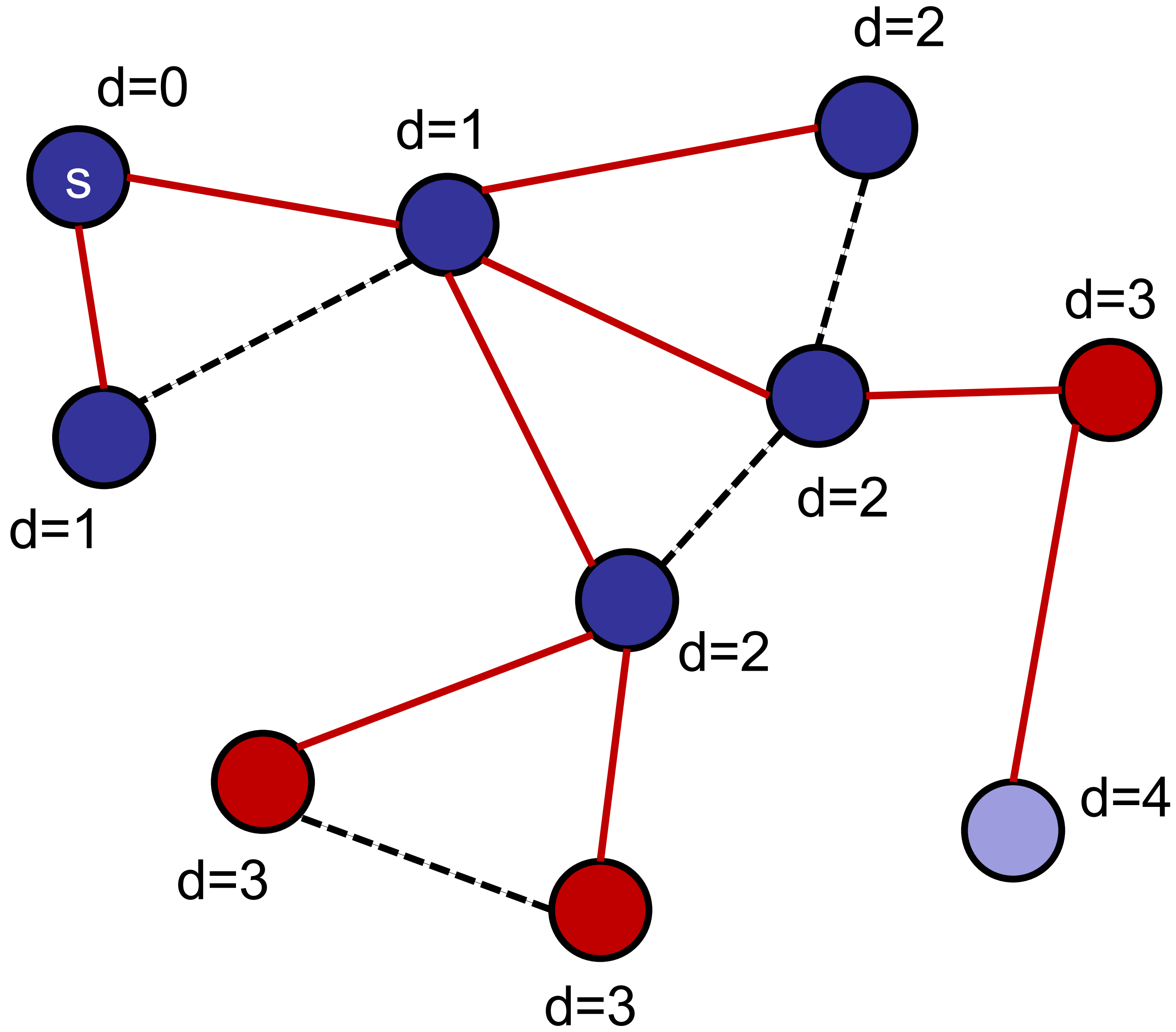
Breadth First Search



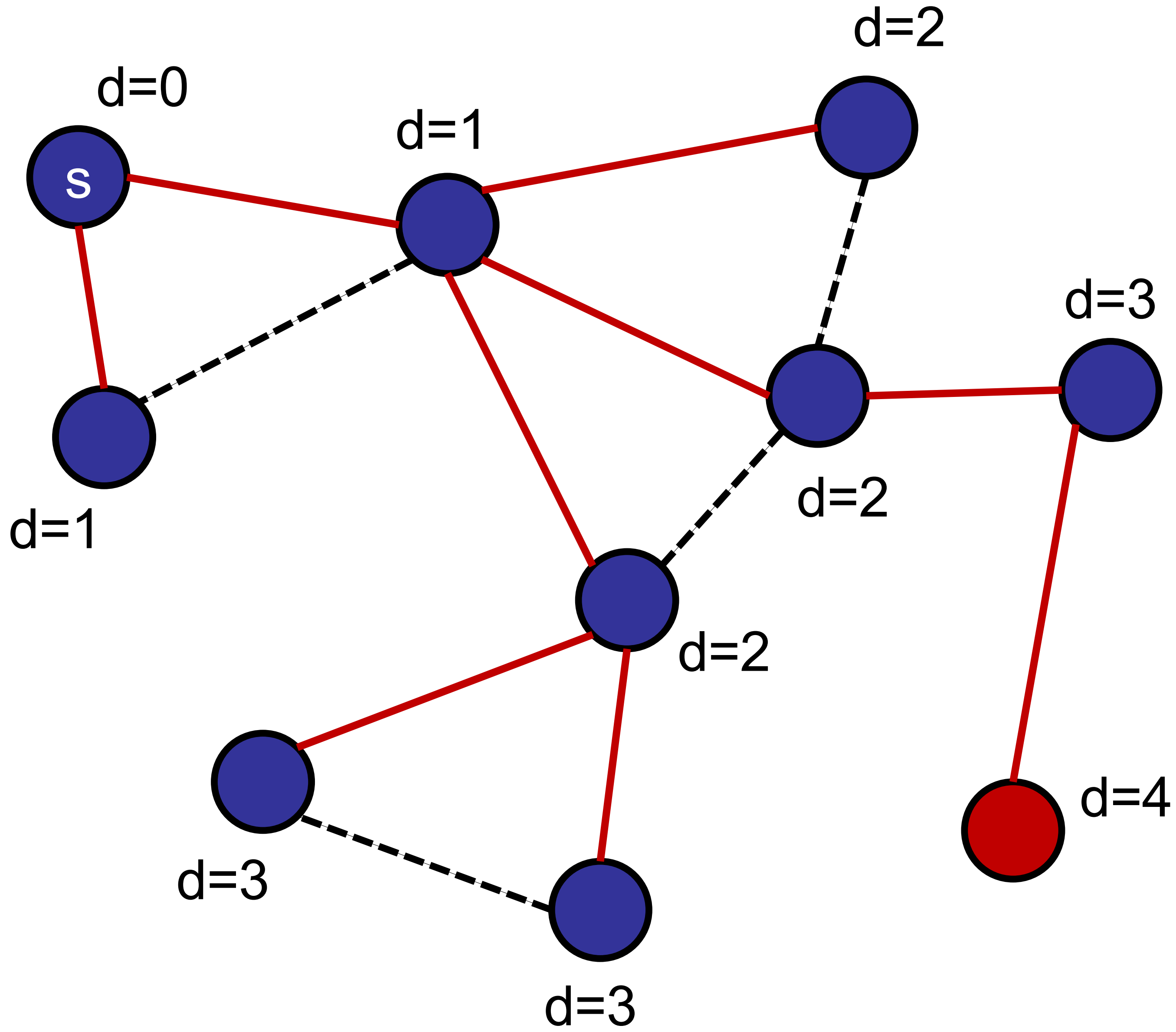
Breadth First Search



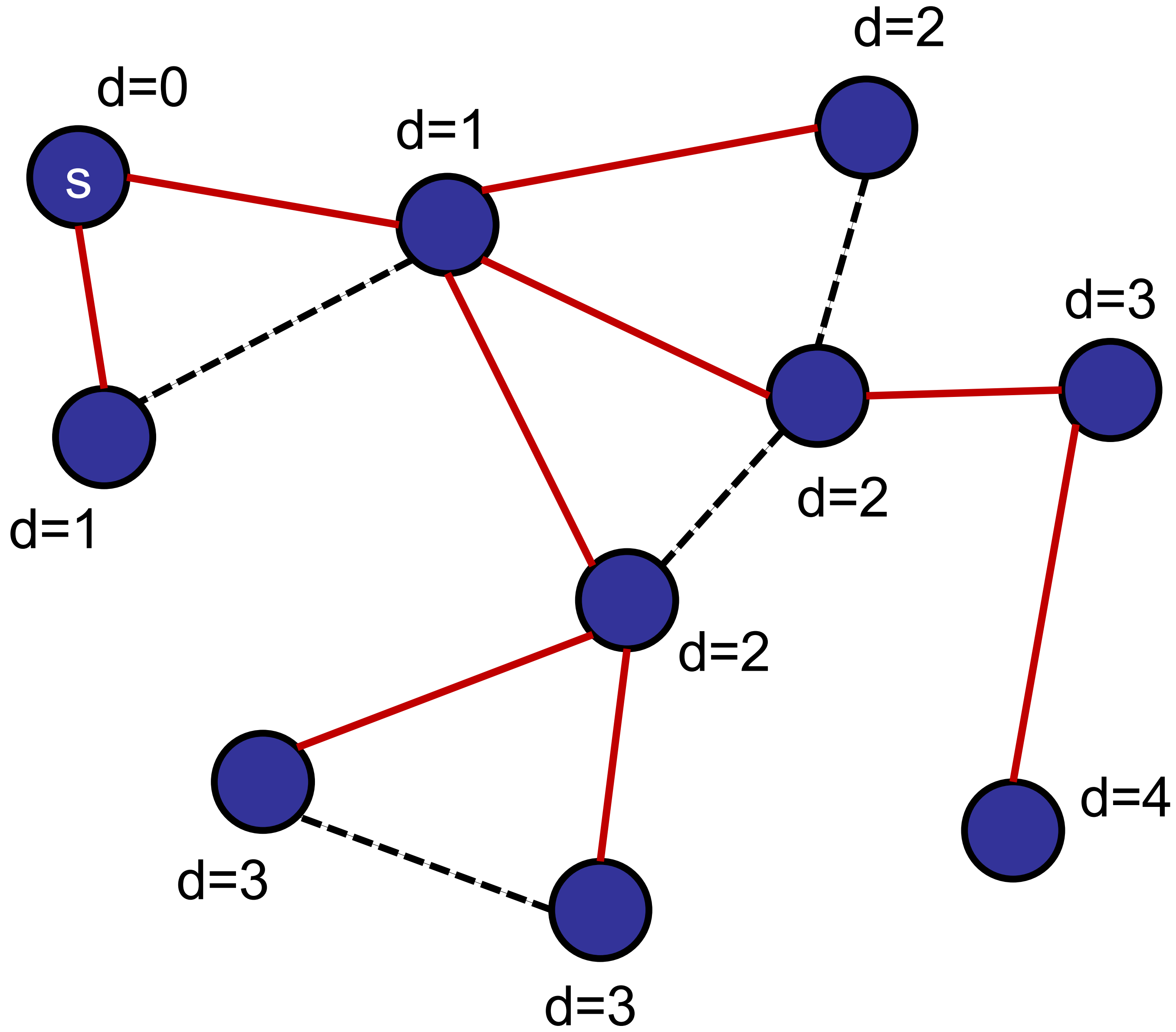
Breadth First Search



Breadth First Search



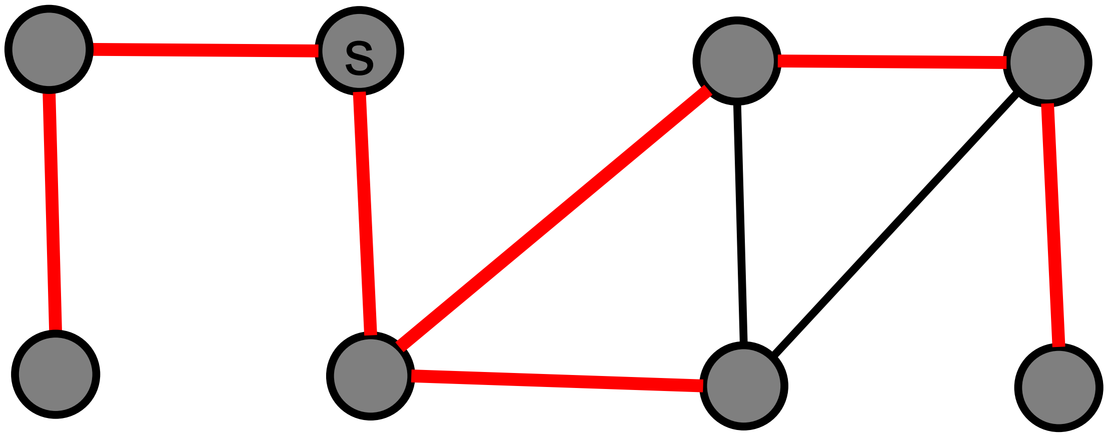
Breadth First Search



Breadth-First Search

Shortest paths:

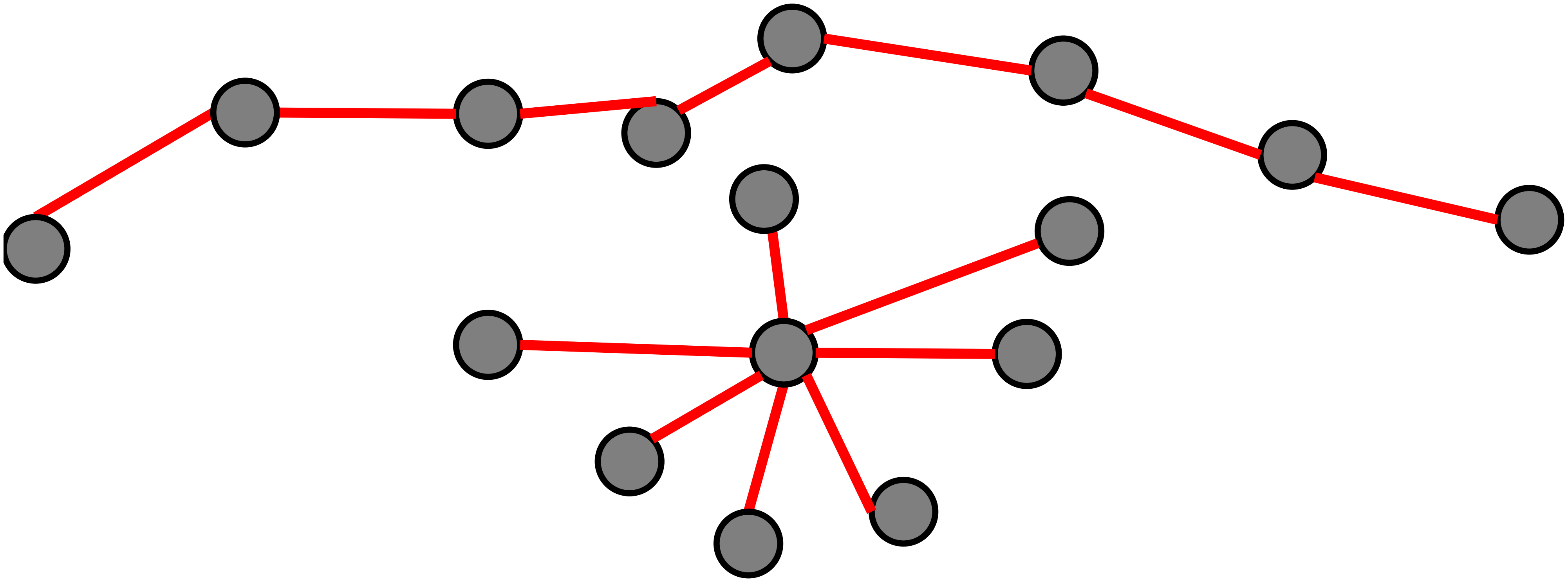
- Parent pointers store shortest path.
- Shortest path is a tree.
- (Possibly high degree; possibly high diameter.)



Breadth-First Search

Shortest paths:

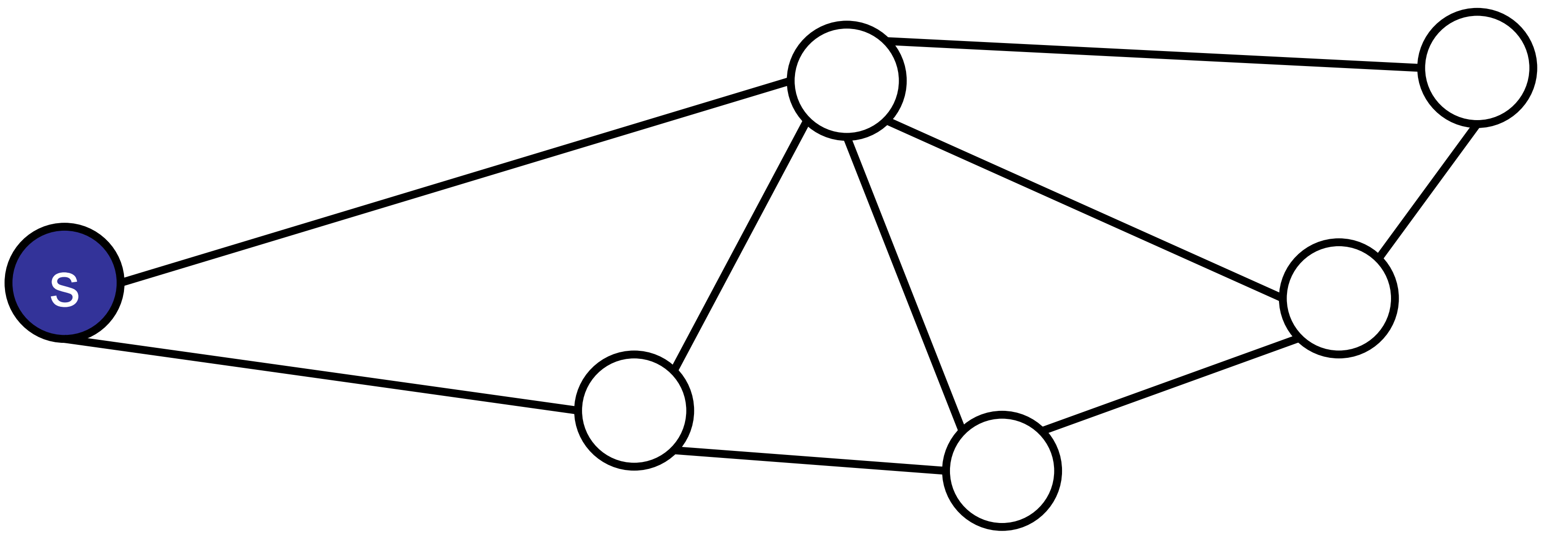
- Parent pointers store shortest path.
- Shortest path is a tree.
- (Possibly high degree; possibly high diameter.)



Breadth-First Search

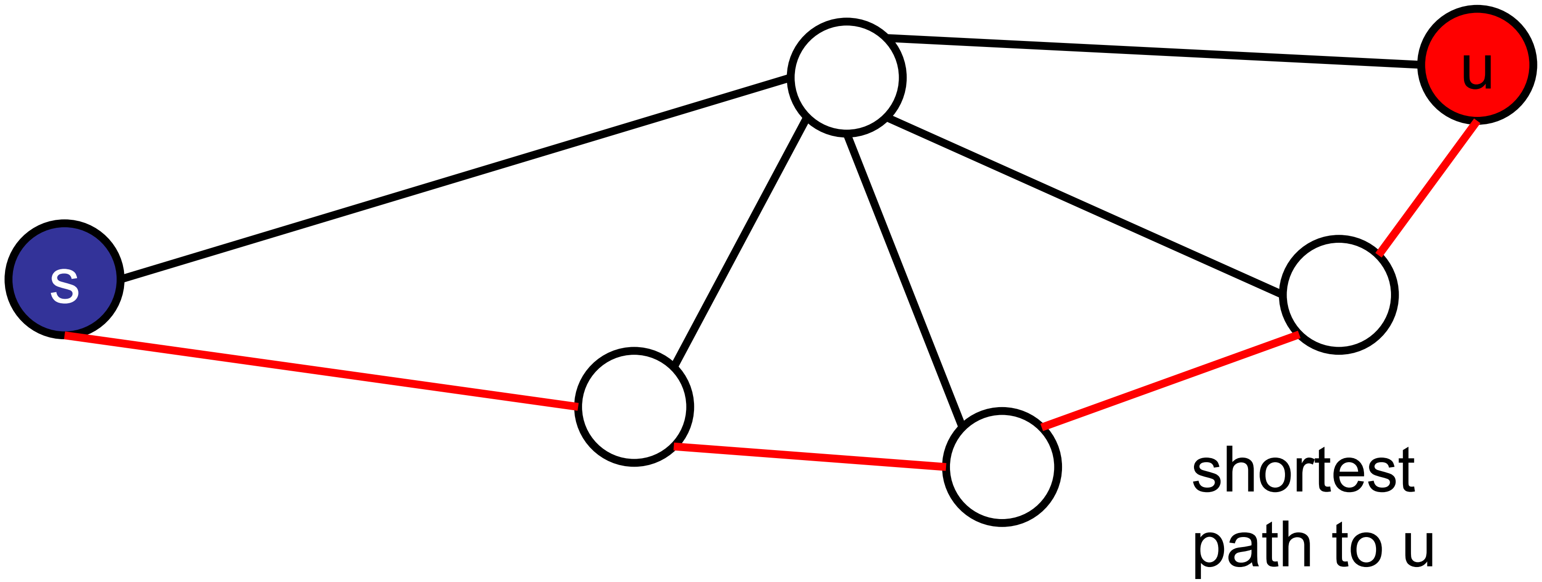
Claim:

- Let P be the set of "shortest paths from s to each node u."
- Shortest paths *always* form a tree
- Can never have a cycle of shortest paths



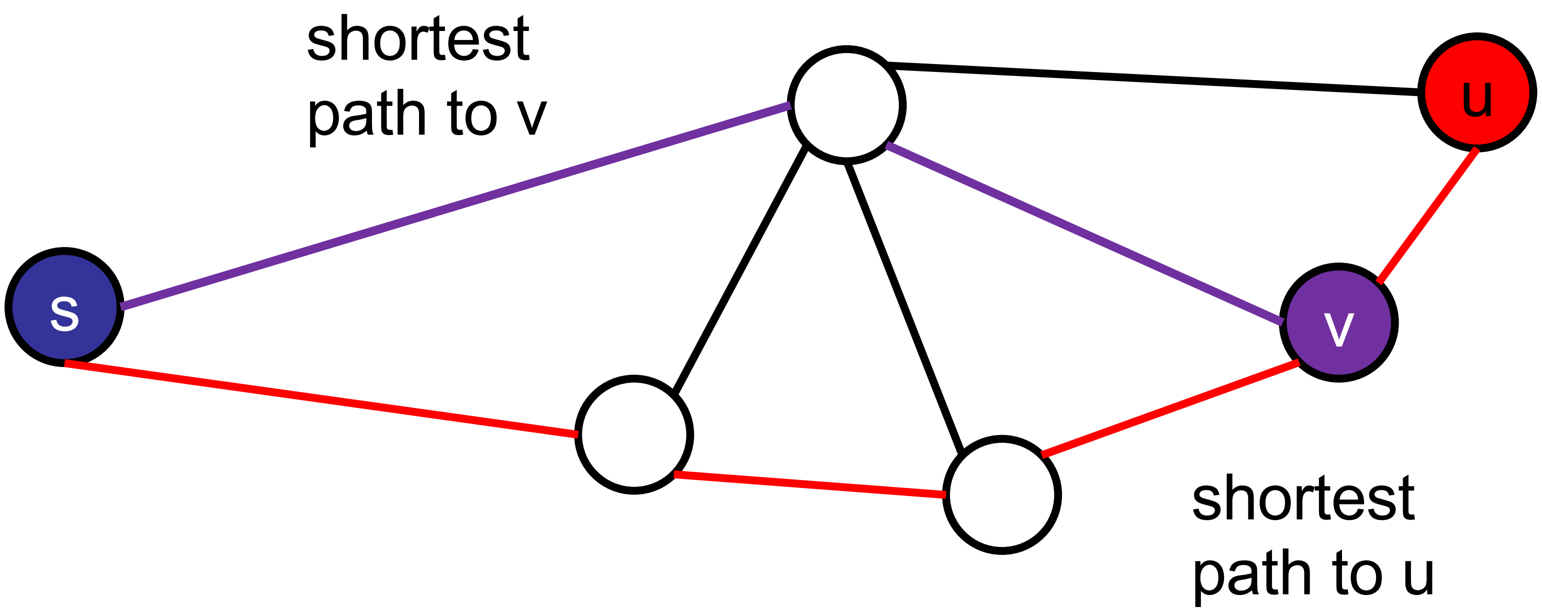
Breadth-First Search

Assume we have a cycle…



Breadth-First Search

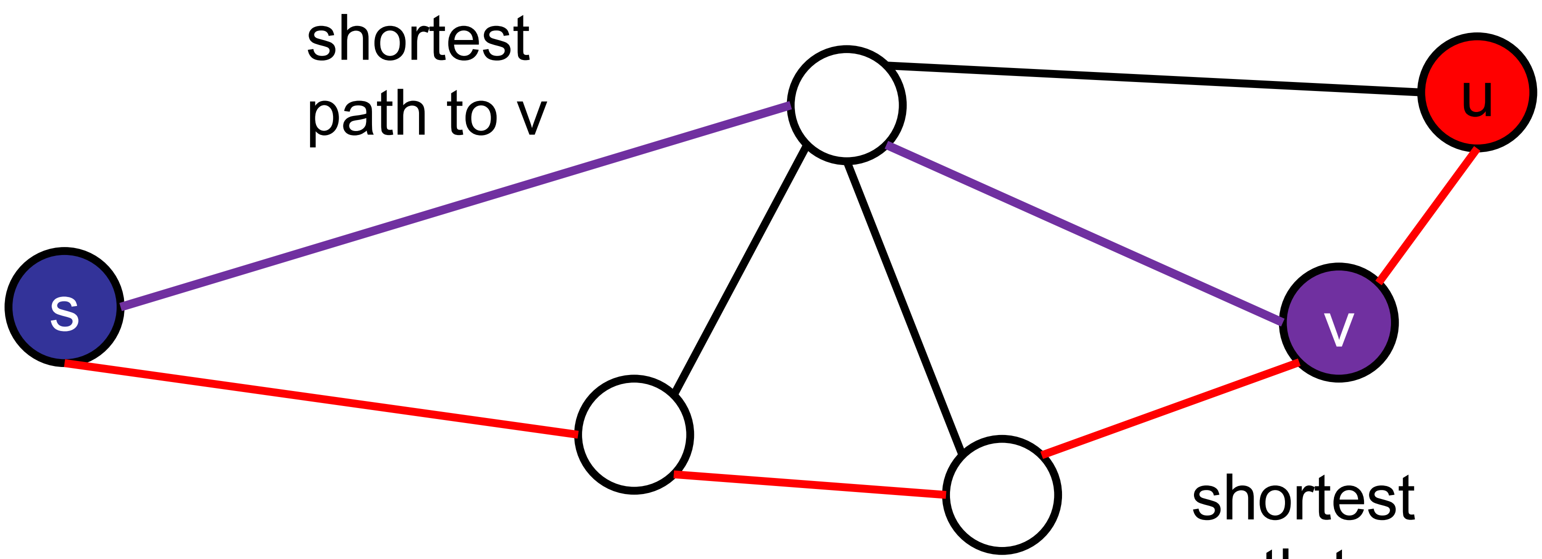
Assume we have a cycle…



Breadth-First Search

Assume we have a cycle…

Purple path to v is shorter than red path to v.

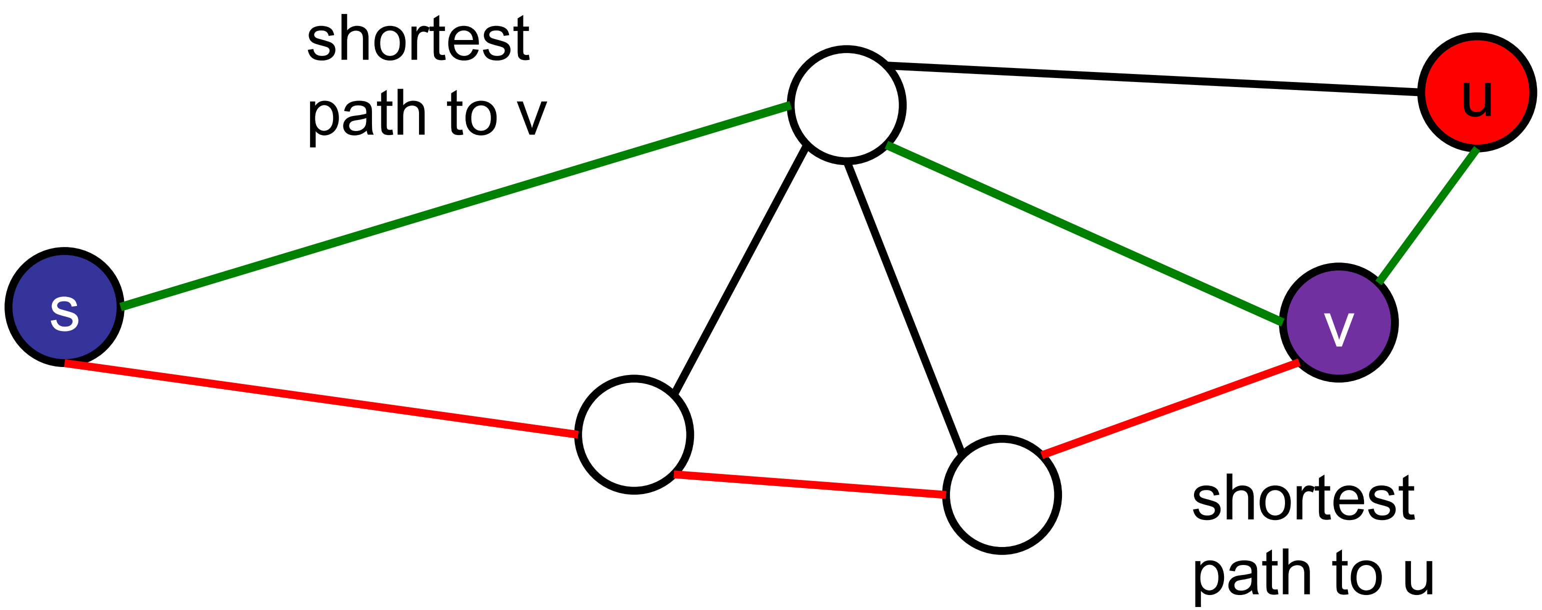


Breadth-First Search

Assume we have a cycle…

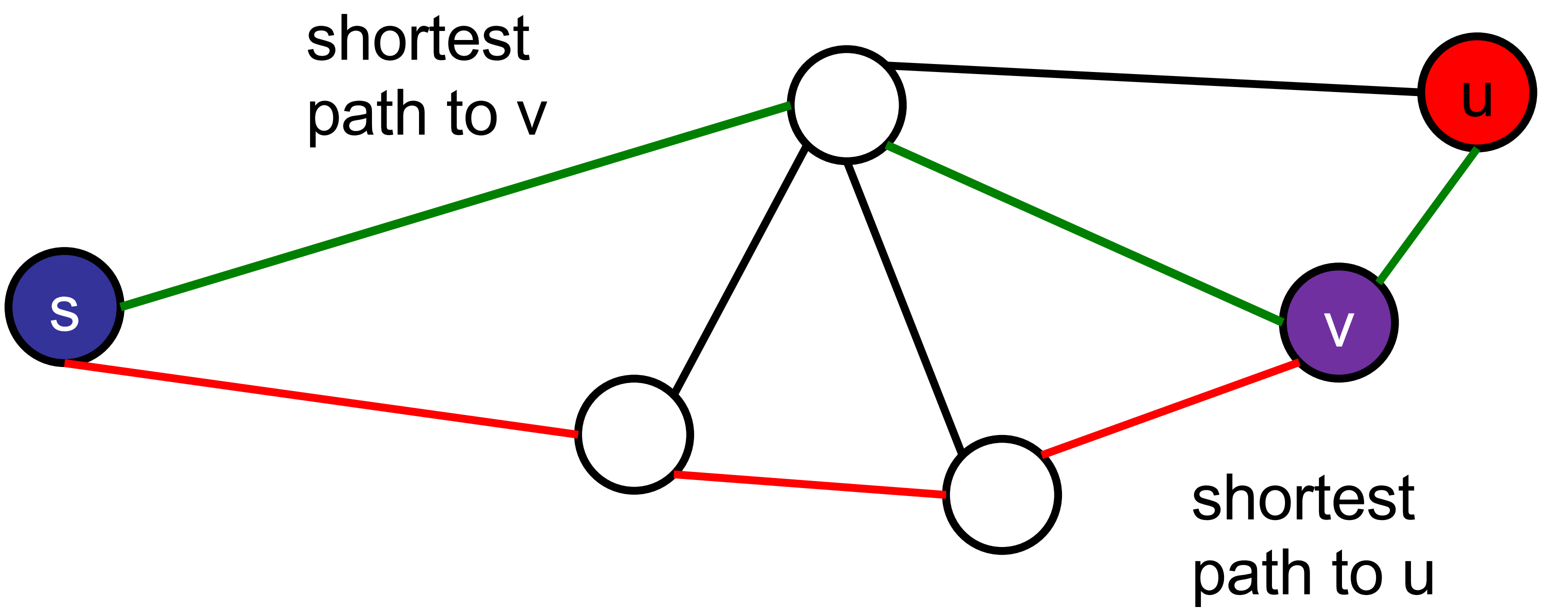
Purple path to v is shorter than red path to v.

So green path to u is shorter than red path to u.



Breadth-First Search

Problem: what if two paths have the same length?



Breadth-First Search

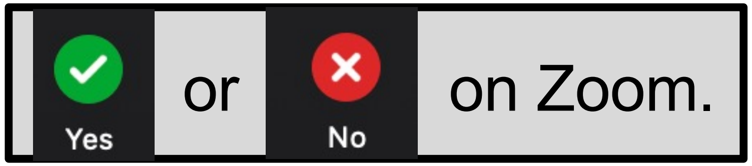
Claim:

- Assume each node has a unique shortest path.
- Let P be the set of “shortest paths from s to each node u.”
- Shortest paths *always* form a tree
- Can never have a cycle of shortest paths

Breadth-First Search

What if non-unique shortest paths?

Will BFS ever construct a cycle?



Breadth-First Search

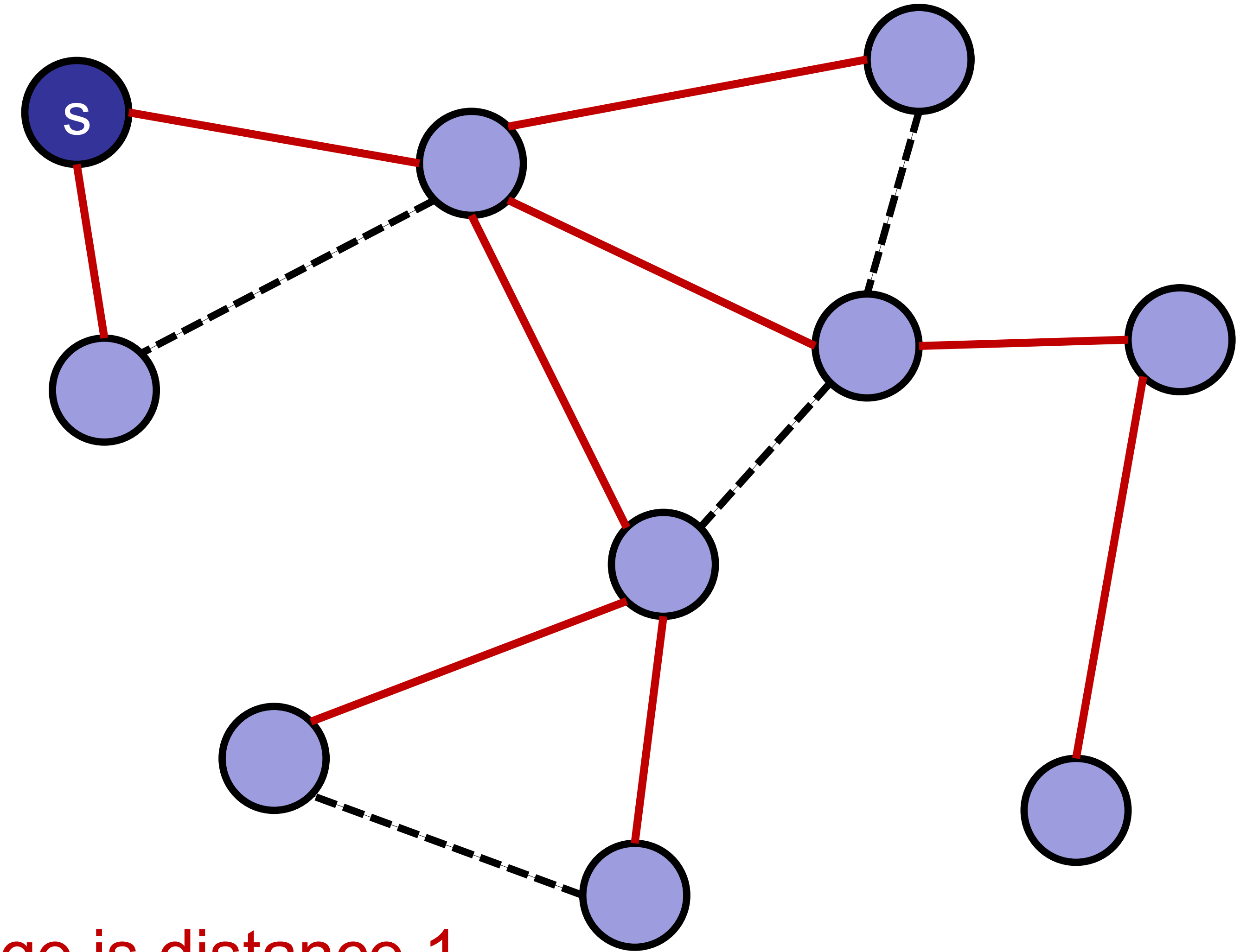
What if non-unique shortest paths?

Will BFS ever construct a cycle?

NO!   Because of the ”visited” check.

```
frontier = {s}
while frontier is not empty:
   next-frontier = {}
   for each node u in the frontier:
     for each edge (u,v) in the graph:
         if v is not marked visited, add v to next-frontier
         mark v as visited.
   frontier = next-frontier
```

Breadth First Search



Beware: each edge is distance 1.

Next week: graphs with distances on the edges.

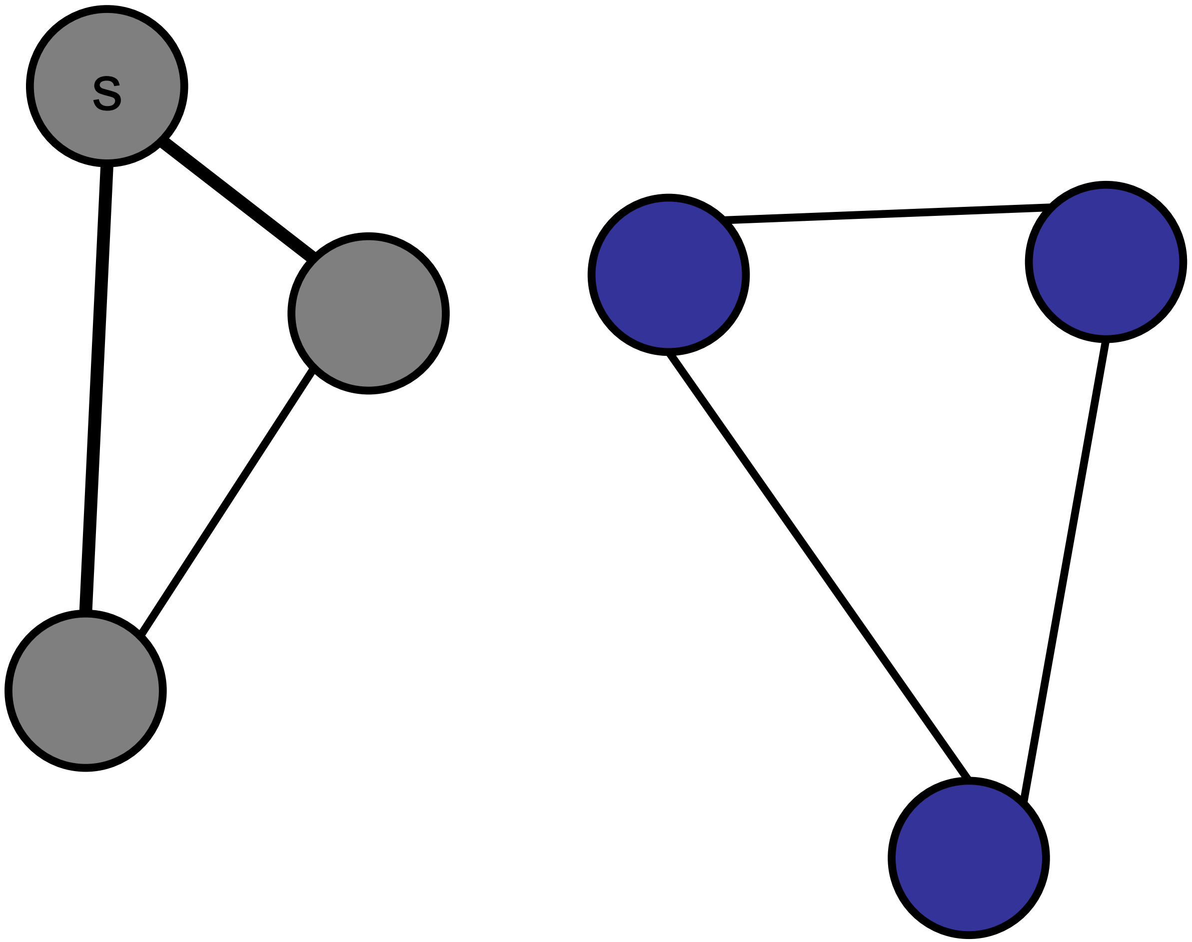
When does BFS fail to visit every node?

1. In a clique.
2. In a cycle.
3. In a graph with two components.
4. In a sparse graph.
5. In a dense graph.
6. Never.

ARCHIPELAGO
is open

BFS on Disconnected Graph

Example:



Visiting every component

Breadth-First Search:

```
for each node u in the graph:
    if u is not marked visited:
        frontier = {u}
        while frontier is not empty:
            next-frontier = {}
            for each node u in the frontier:
                for each edge (u,v) in the graph:
                    if v is not marked visited, add v to next-frontier
                    mark v as visited.
            frontier = next-frontier
```

Important if need to visit *every* node.
Important if searching for something.
NOT good for measuring distances.

The running time of BFS (using adjacency list) is:

1. O(V)
2. O(E)
3. O(V+E)
4. O(VE)
5. (V^2)
6. I have no idea.

ARCHIPELAGO
is open

The running time of BFS (using adjacency list) is:

1. O(V)
2. O(E)
3. O(V+E)
4. O(VE)
5. (V^2)
6. I have no idea.

Depends on adjacency list vs. adjacency matrix.

Here: assume adjacency list.

Breadth-First Search

Analysis:

– Vertex v = "start" once.

O(V)

– Vertex v added to nextFrontier (and frontier) once.

  - After visited, never re-added.

– Each v.nbrlist is enumerated once.

  - When v is removed from frontier.

O(E)

Running time

Breadth-First Search:

```
for each node u in the graph:
    if u is not marked visited:
        frontier = {u}
        while frontier is not empty:
            next-frontier = {}
            for each node u in the frontier:
                for each edge (u,v) in the graph:
                    if v is not marked visited, add v to next-frontier
                    mark v as visited.
            frontier = next-frontier
```

Each node is only in ONE frontier.

Each edge only has two endpoints and so is examined only twice.

Searching a Graph

Goal:

- Start at some vertex **s** = start.
- Find some other vertex **f** = finish.

  Or: visit **all** the nodes in the graph;

Two basic techniques:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

Graph representation:

- Adjacency list

Depth-First Search

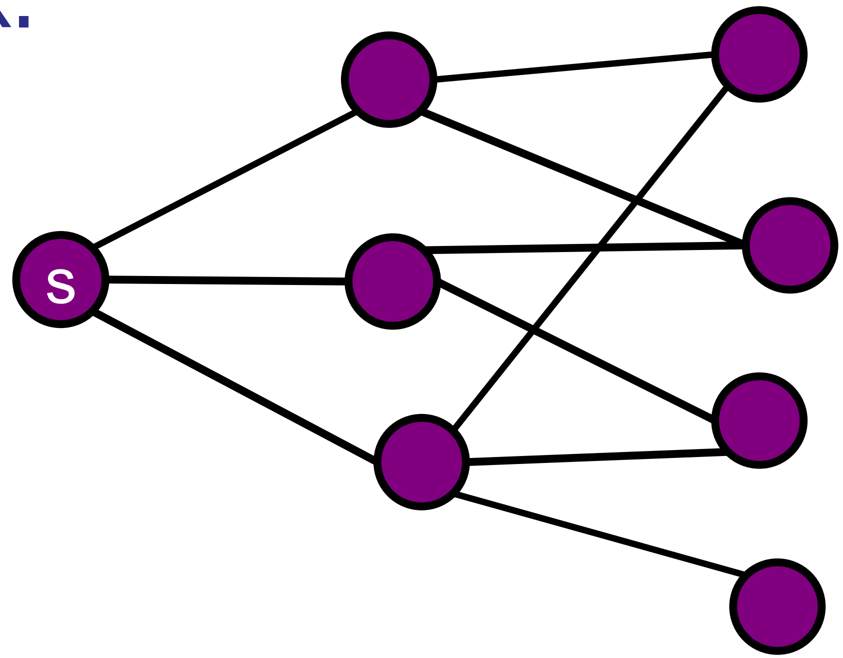
Exploring a maze:

- Follow path until stuck.
- Backtrack along breadcrumbs until reach unexplored neighbor.
- Recursively explore.

Searching a graph

Depth-First Search:

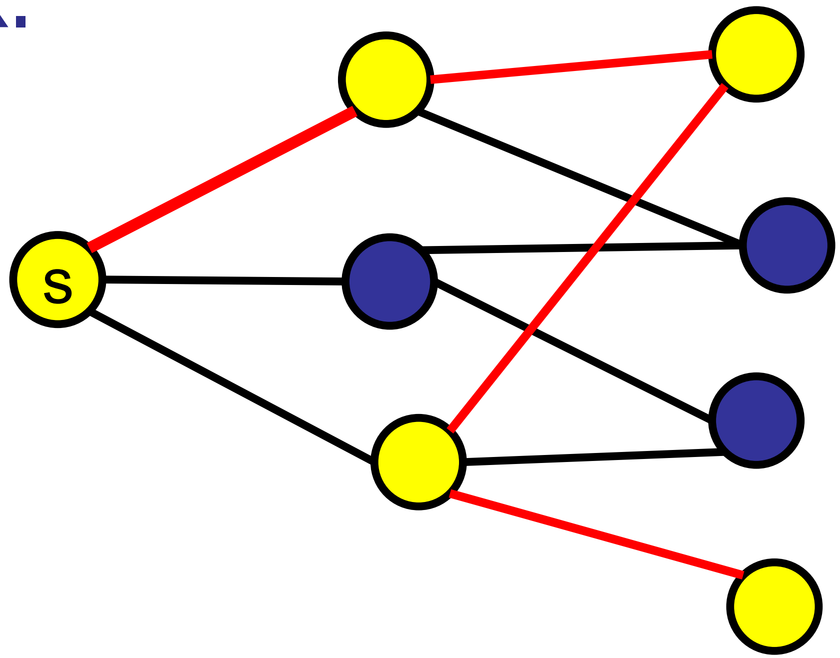
- Follow path until you get stuck
- Backtrack until you find a new edge
- Recursively explore it
- Don't repeat a vertex.



Searching a graph

Depth-First Search:

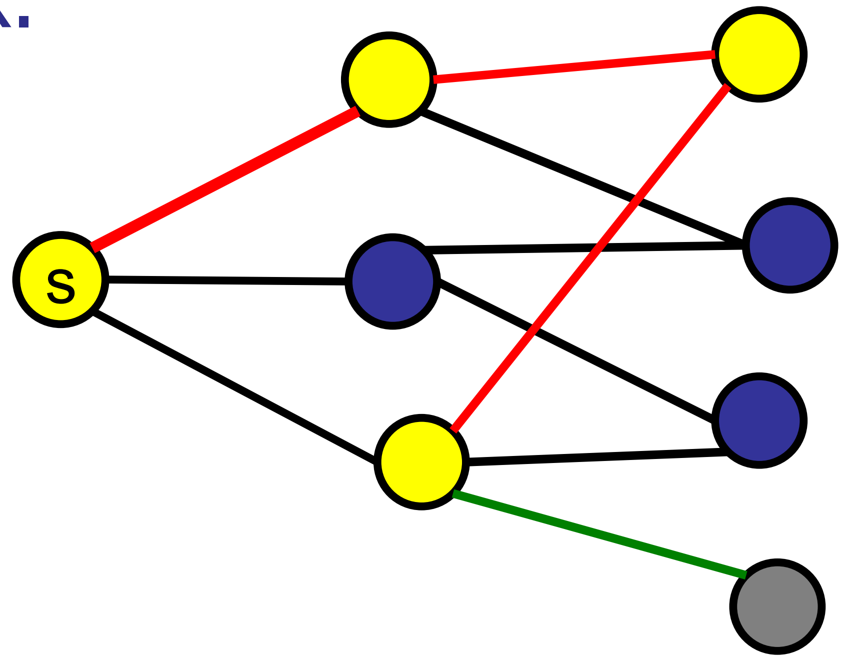
- Follow path until you get stuck
- Backtrack until you find a new edge
- Recursively explore it
- Don’t repeat a vertex.



Searching a graph

Depth-First Search:

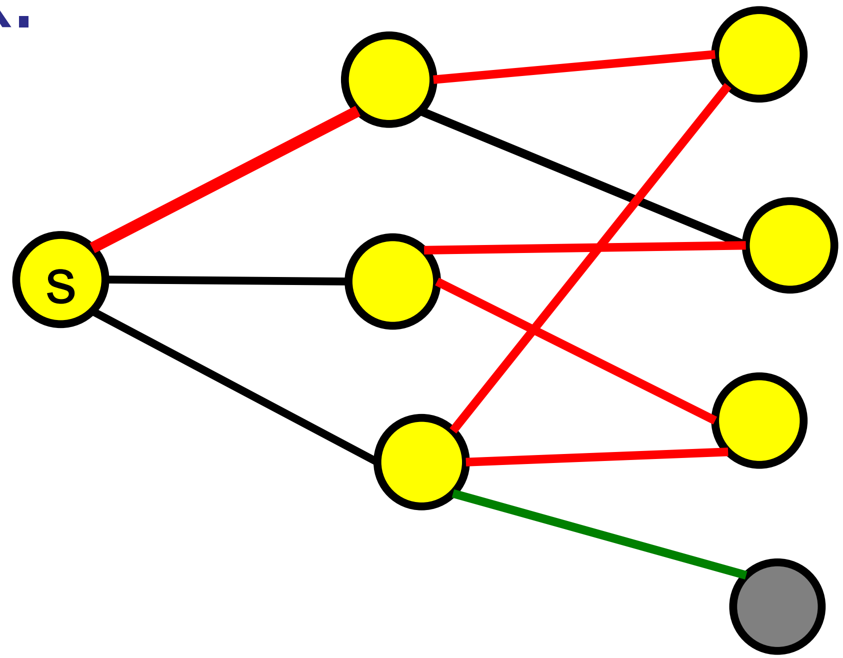
- Follow path until you get stuck
- Backtrack until you find a new edge
- Recursively explore it
- Don’t repeat a vertex.



Searching a graph

Depth-First Search:

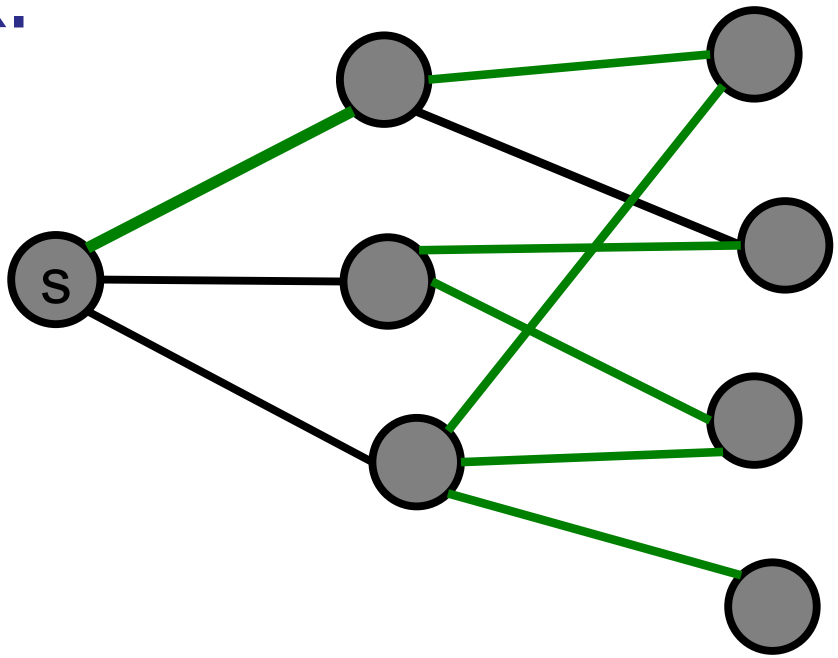
- Follow path until you get stuck
- Backtrack until you find a new edge
- Recursively explore it
- Don’t repeat a vertex.



Searching a graph

Depth-First Search:

- Follow path until you get stuck
- Backtrack until you find a new edge
- Recursively explore it
- Don't repeat a vertex.



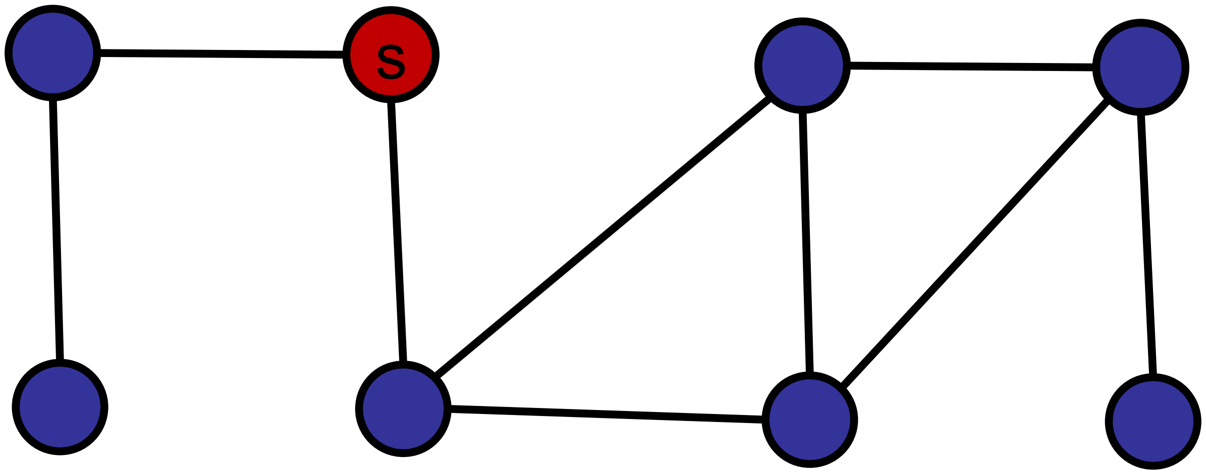
Depth-First Search

```
DFS-visit(Node[] nodeList, boolean[] visited, int startId){
  for (Integer v : nodeList[startId].nbrList) {
     if (!visited[v]){
          visited[v] = true;
          DFS-visit(nodeList, visited, v);
     }
  }
}
```

Depth-First Search

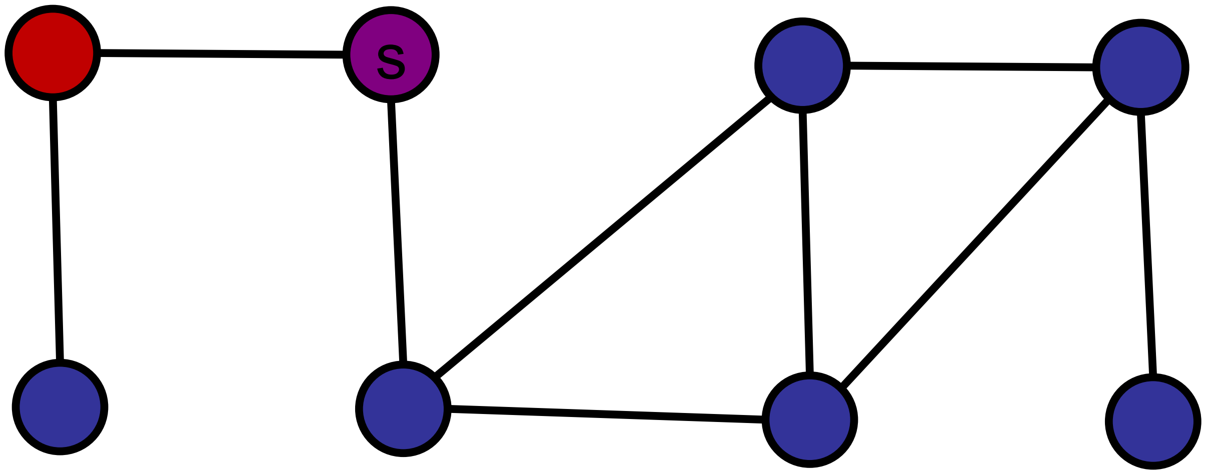
```
DFS(Node[] nodeList){
  boolean[] visited = new boolean[nodeList.length];
  Arrays.fill(visited, false);


  for (start = i; start<nodeList.length; start++) {
     if (!visited[start]){
           visited[start] = true;
           DFS-visit(nodeList, visited, start);
     }
  }
}
```

Depth-First Search Example



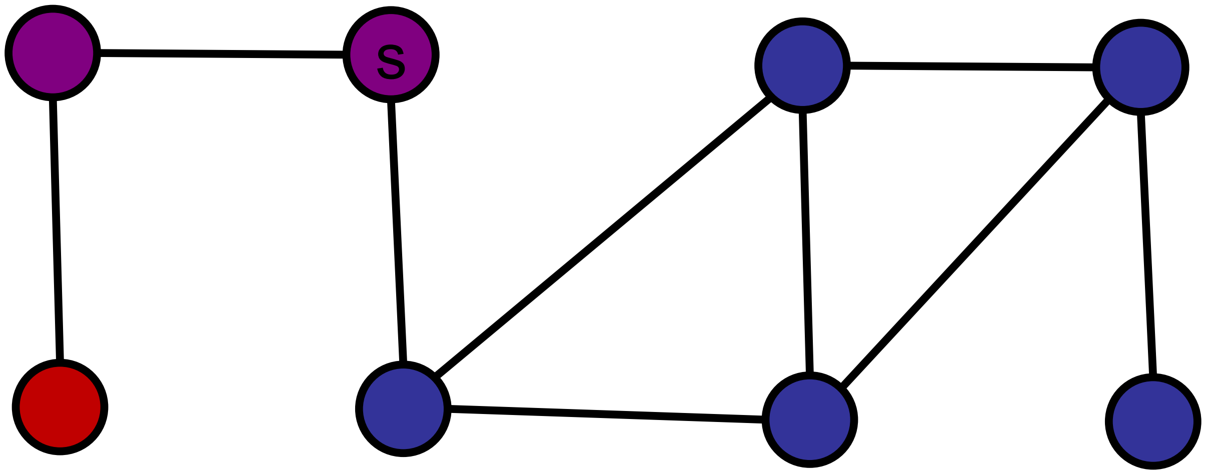
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



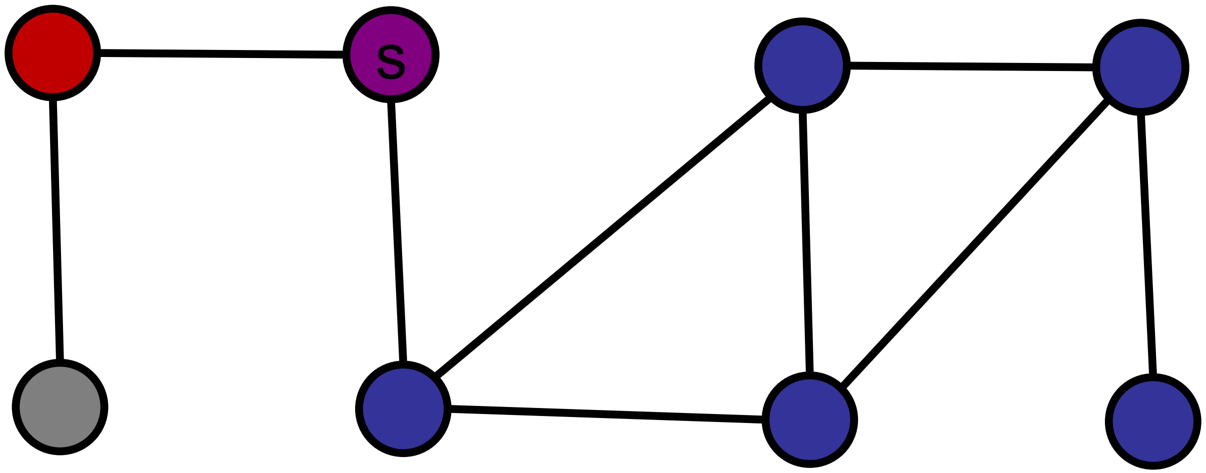
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



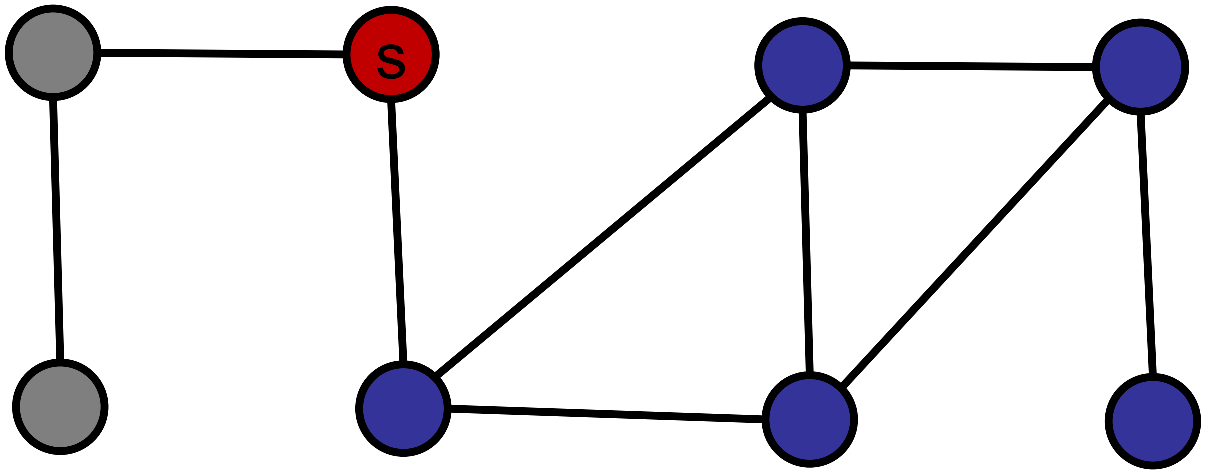
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



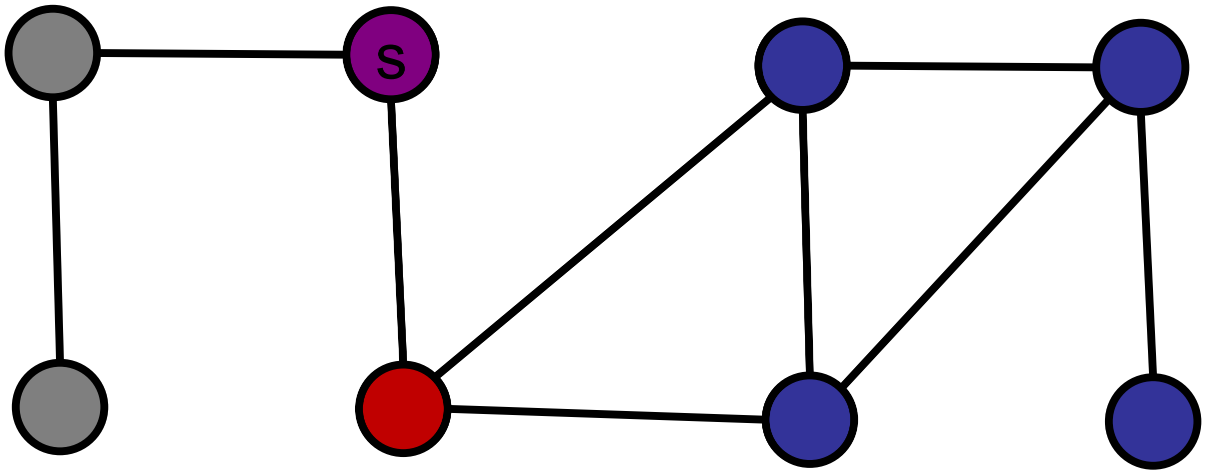
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



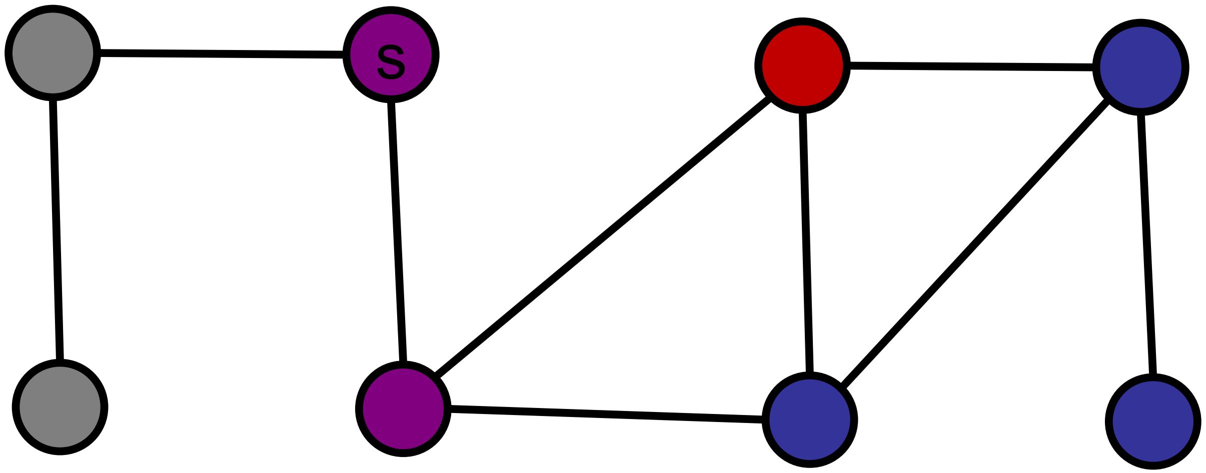
Red = active frontier

Purple = next

Gray = visited

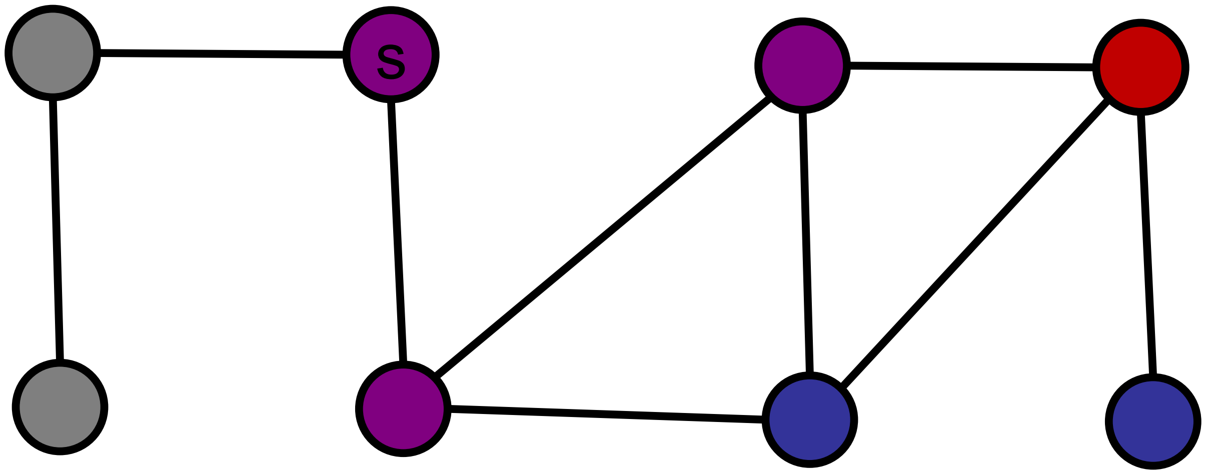
Blue = unvisited

Depth-First Search Example



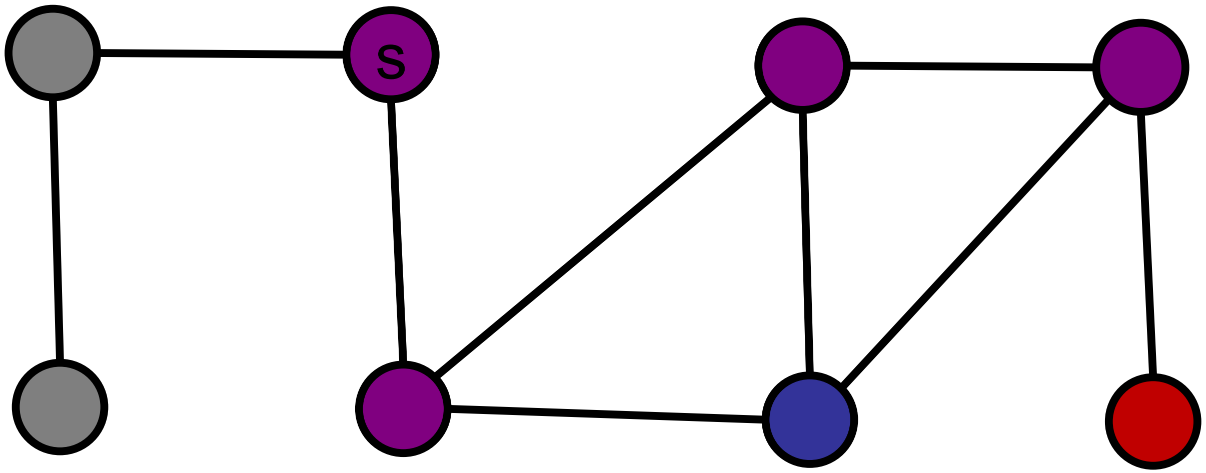
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



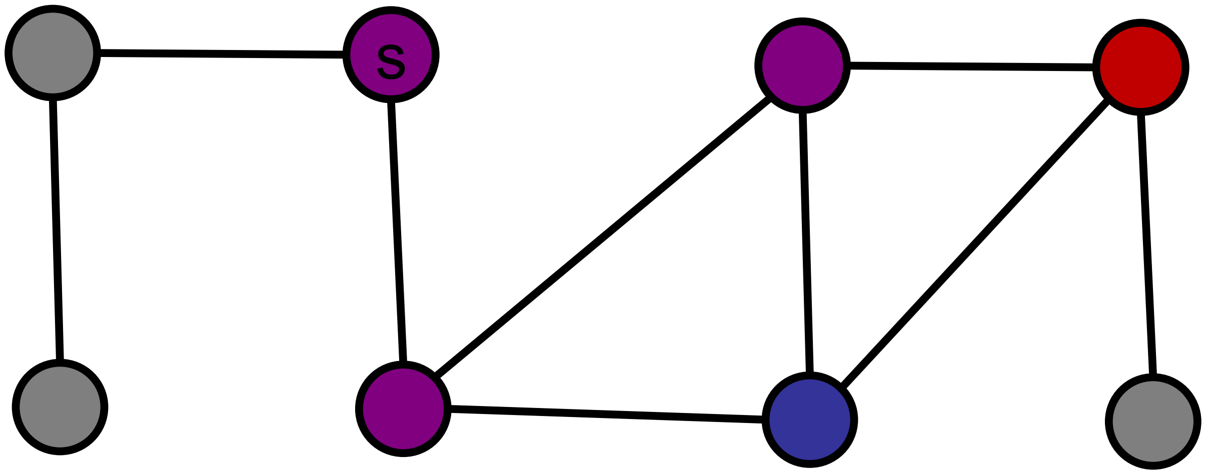
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



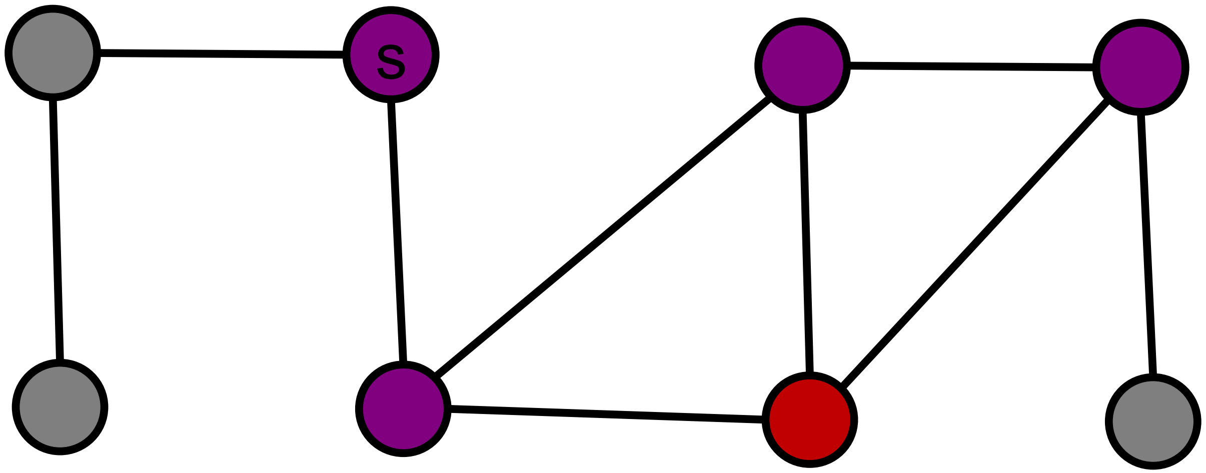
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



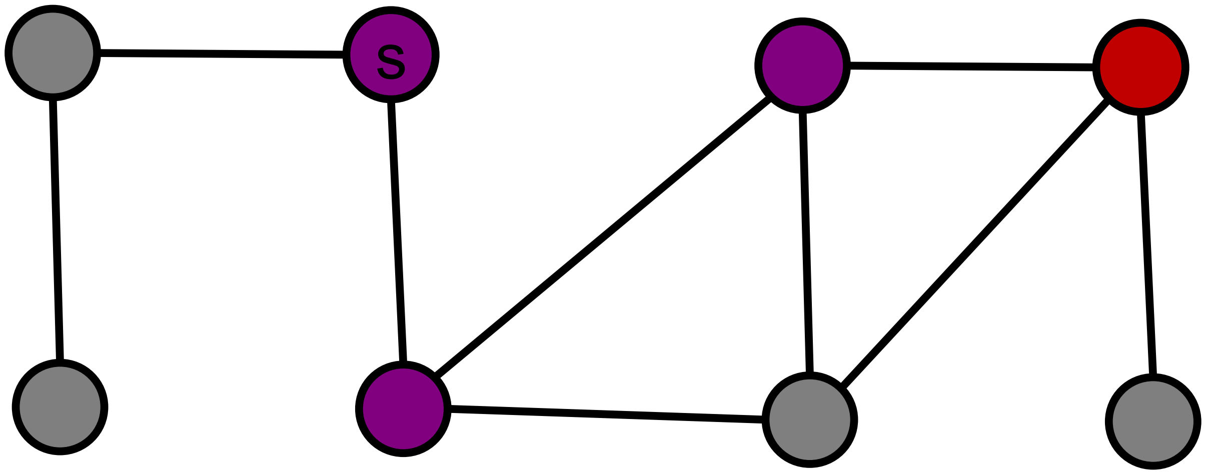
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



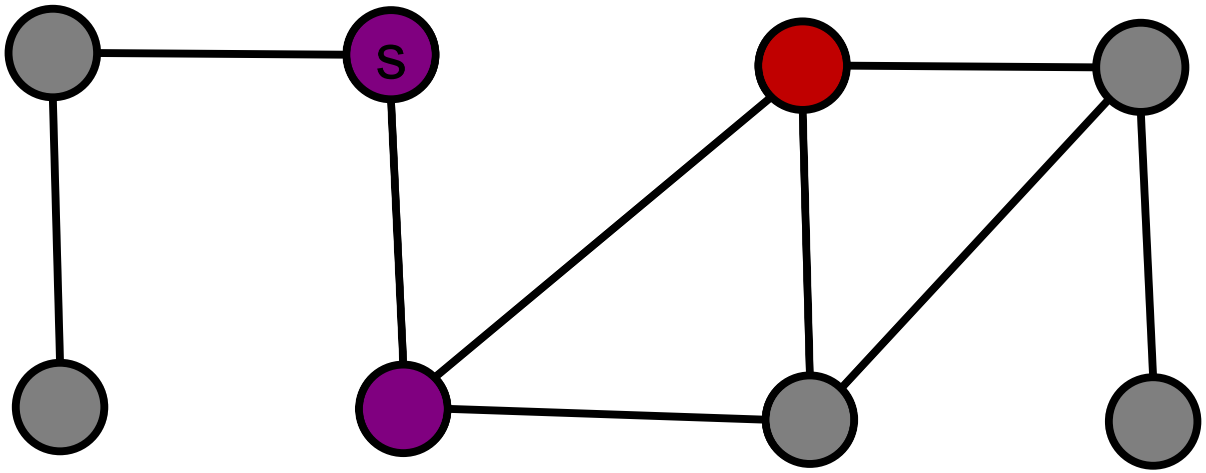
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



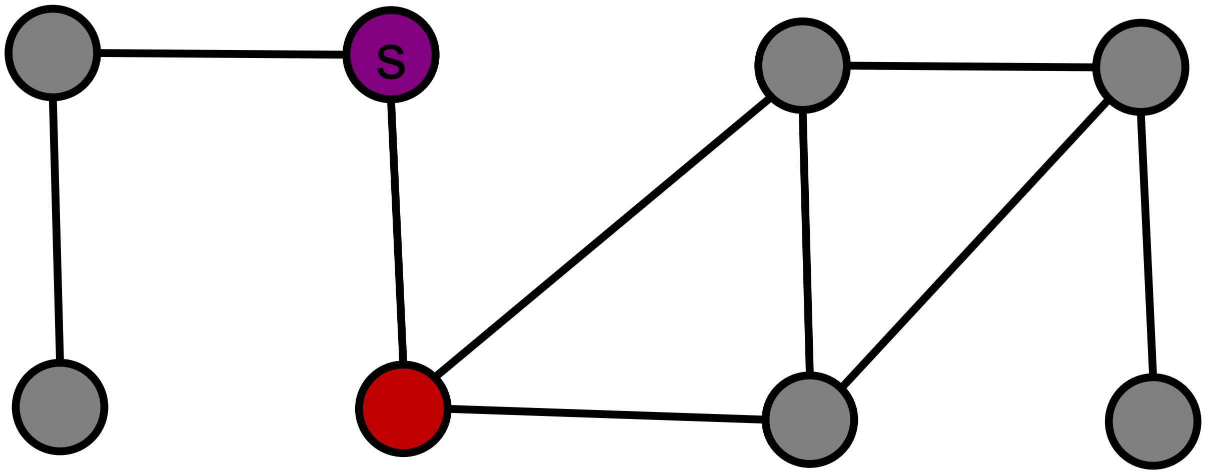
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



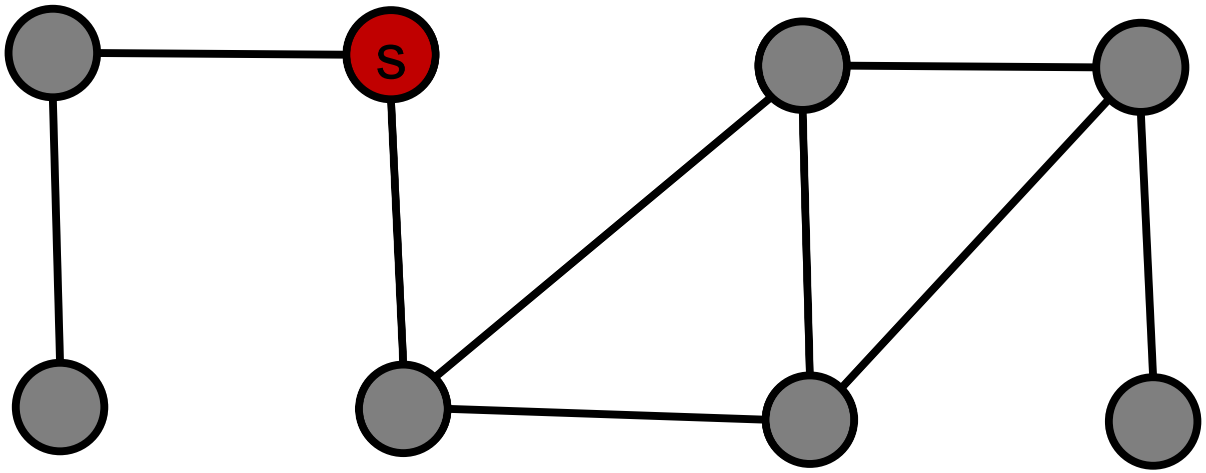
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



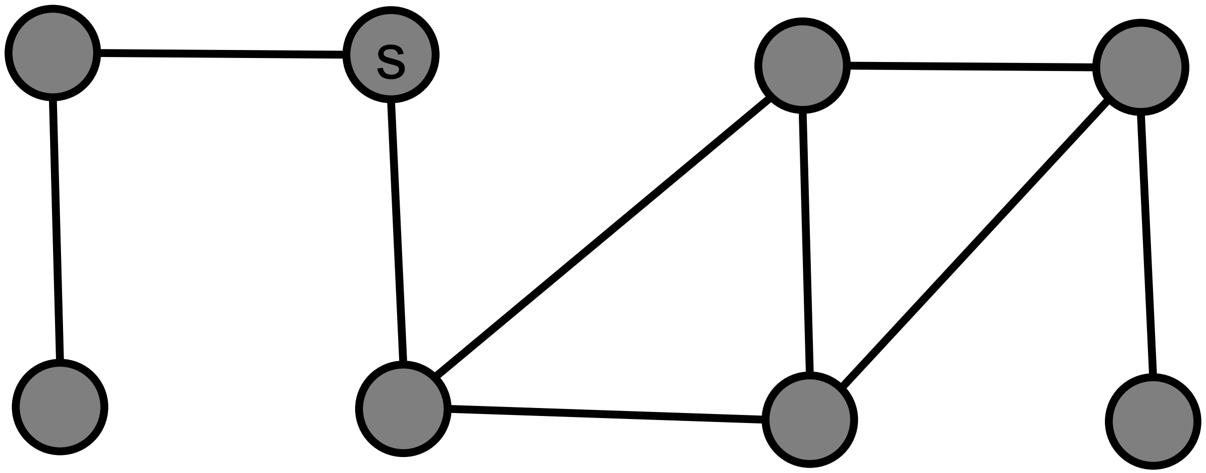
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



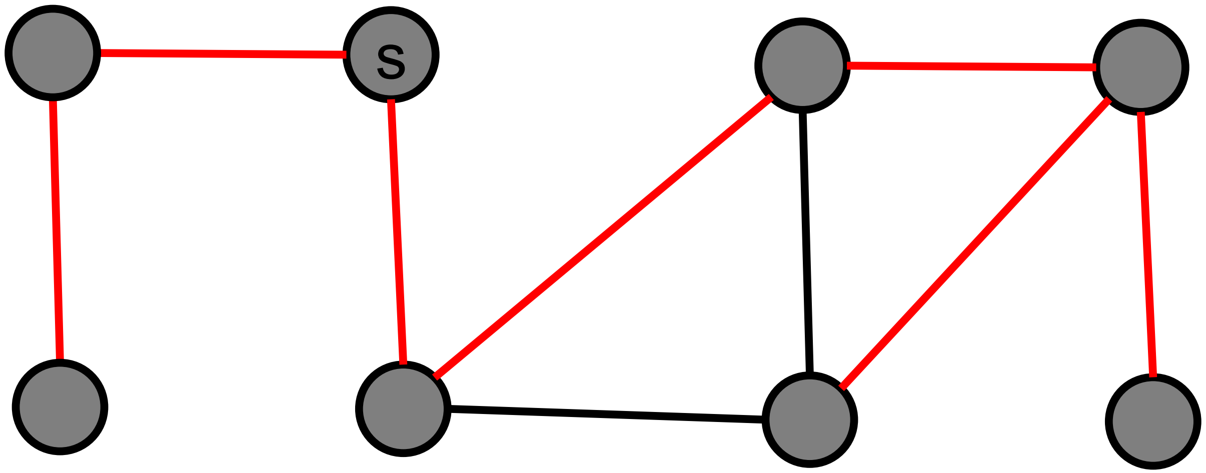
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



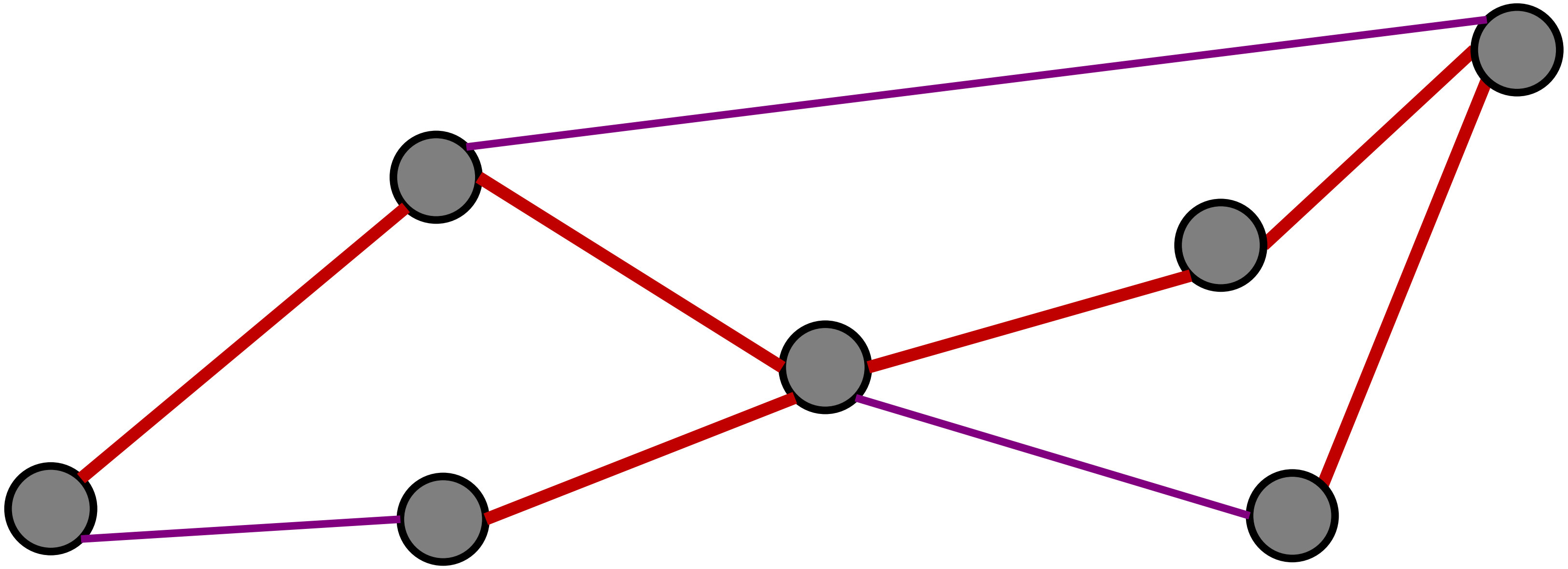
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

DFS parent edges

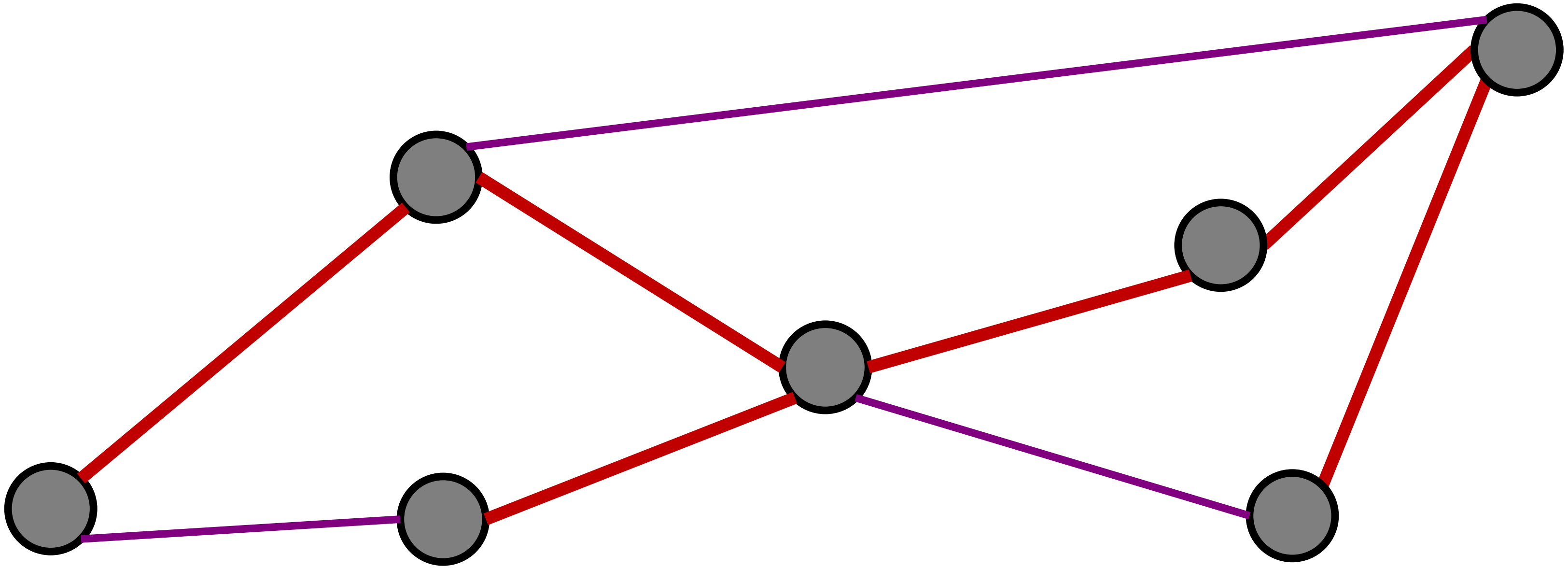


Red = Parent Edges
Purple = Non-parent edges

Which is true? (More than one may apply.)

1. DFS parent graph is a cycle.
2. DFS parent graph is a tree.
3. DFS parent graph has low-degree.
4. DFS parent graph has low diameter.
5. None of the above.

DFS parent edges = tree



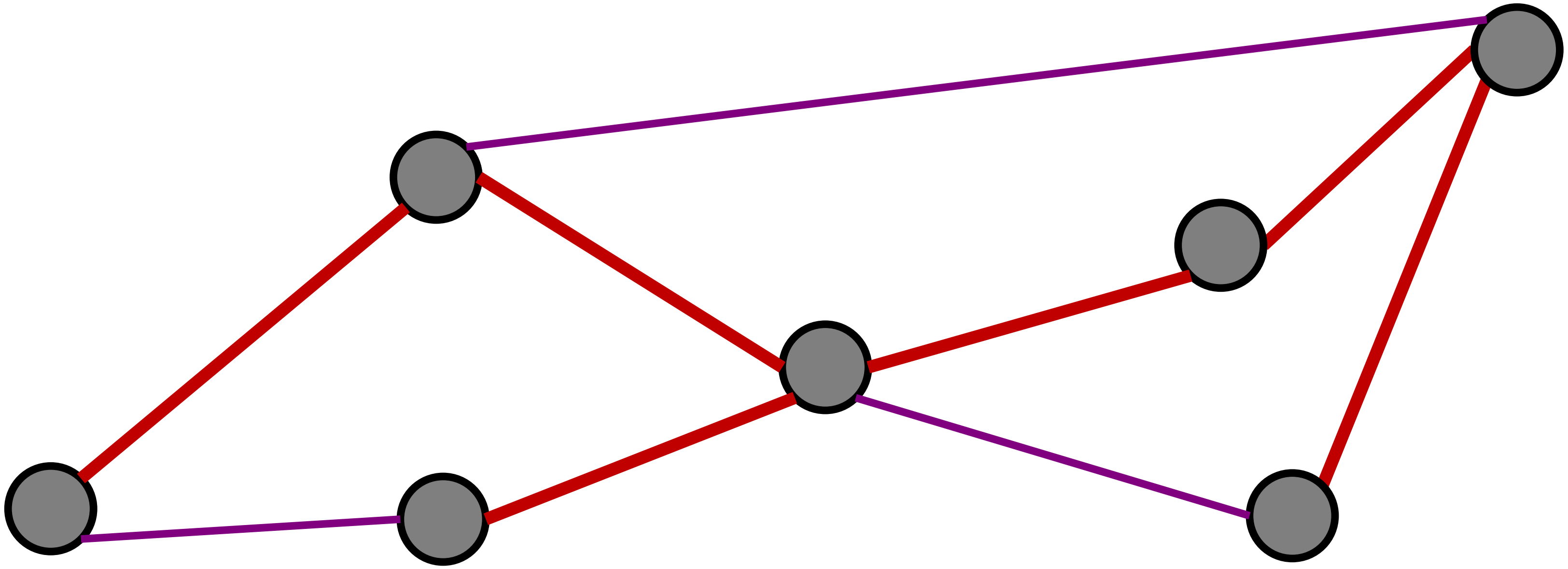
Red = Parent Edges
Purple = Non-parent edges

True or false:
DFS parent graph contains shortest paths.

1. True
2. False

ARCHIPELAGO is open

DFS parent edges = tree



Red = Parent Edges
Purple = Non-parent edges

Note: not shortest paths!

The running time of DFS is:

1. O(V)
2. O(E)
3. O(V+E)
4. O(VE)
5. $O(V^2)$
6. I have no idea.

ARCHIPELAGO

is open

Depth-First Search

$O(V)$

Analysis:

- DFS-visit called only once per node.
  - After visited, never call DFS-visit again.

- In DFS-visit, each neighbor is enumerated.

$O(E)$

If the graph is stored as an adjacency matrix, what is the running time of DFS?

1. $O(V)$
2. $O(E)$
3. $(V+E)$
4. $O(VE)$
5. $O(V^2)$
6. $O(E^2)$

Depth-First Search

Analysis:

- DFS-visit called only once per node.
  - After visited, never call DFS-visit again.

- In DFS-visit, each neighbor is enumerated.

O(V)

O(V)
per
node